An Efficient Generalization of Counting Sort for Large, possibly Infinite Key Ranges

Pasquale Noce
Software Engineer at HID Global, Italy
pasquale dot noce dot lavoro at gmail dot com
pasquale dot noce at hidglobal dot com

December 7, 2022

Abstract

Counting sort is a well-known algorithm that sorts objects of any kind mapped to integer keys, or else to keys in one-to-one correspondence with some subset of the integers (e.g. alphabet letters). However, it is suitable for direct use, viz. not just as a subroutine of another sorting algorithm (e.g. radix sort), only if the key range is not significantly larger than the number of the objects to be sorted.

This paper describes a tail-recursive generalization of counting sort making use of a bounded number of counters, suitable for direct use in case of a large, or even infinite key range of any kind, subject to the only constraint of being a subset of an arbitrary linear order. After performing a pen-and-paper analysis of how such algorithm has to be designed to maximize its efficiency, this paper formalizes the resulting generalized counting sort (GCsort) algorithm and then formally proves its correctness properties, namely that (a) the counters' number is maximized never exceeding the fixed upper bound, (b) objects are conserved, (c) objects get sorted, and (d) the algorithm is stable.

Contents

1 Algorithm's description, analysis, and formalization

theory *Algorithm*
imports *Main*
begin

This paper is dedicated to Gaia, my sweet niece, whose arrival has blessed me and my family with joy and tenderness.

Moreover, I would like to thank my colleague Iacopo Rippa, who patiently listened to the ideas underlying sections 1.1.2 and 1.1.4, and helped me expand those ideas into complete proofs by providing me with valuable hints and test data.

1.1 Introduction

1.1.1 Counting sort

Counting sort is a well-known algorithm that sorts a collection of objects of any kind, as long as each such object is associated with a signed integer key, according to their respective keys (cf. [2], [8]). If xs is the input array containing n objects to be sorted, *out* is the output, sorted array, and *key* is the function mapping objects to keys, counting sort works as follows (assuming arrays to be zero-based):

1. Search the minimum key mi and the maximum key ma occurring within xs (which can be done via a single loop over xs).
2. Allocate an array ns of $ma - mi + 2$ unsigned integers and initialize all its elements to 0.
3. For each i from 0 to $n - 1$, increase $ns[key(xs[i]) - mi + 1]$ by 1.
4. For each i from 2 to $ma - mi$, increase $ns[i]$ by $ns[i - 1]$.
5. For each i from 0 to $n - 1$, set $out[ns[key(xs[i]) - mi]]$ to $xs[i]$ and increase $ns[key(xs[i]) - mi]$ by 1.

Steps 1 and 2 take $O(n)$ and $O(ma - mi)$ time, respectively. Step 3 counts how many times each possible key occurs within xs, and takes $O(n)$ time. Step 4 computes the offset within *out* of the first object in xs, if any, having each possible key, and takes $O(ma-mi)$ time. Finally, step 5 fills *out*, taking $O(n)$ time. Thus, the overall running time is $O(n) + O(ma - mi)$, and the same is obviously true of memory space.

If the range of all the keys possibly occurring within xs, henceforth briefly referred to as the *key range*, is known in advance, the first two steps can be skipped by using the minimum and maximum keys in the key range as mi, ma and pre-allocating (possibly statically, rather than dynamically, in real-world implementations) an array ns of size $ma - mi + 2$. However, this does not affect the asymptotic running time and memory space required by the algorithm, since both keep being $O(n) + O(ma - mi)$ independently of the distribution of the keys actually occurring in xs within the key range.

As a result, counting sort is suitable for direct use, viz. not just as a subroutine of another sorting algorithm such as radix sort, only if the key range is not significantly larger than n. Indeed, if 100 objects with 100,000 possible keys have to be sorted, accomplishing this task by allocating, and iterating over, an array of 100,000 unsigned integers to count keys' occurrences would be quite impractical! Whence the question that this paper will try to answer: how can counting sort be generalized for direct use in case of a large key range?

Solving this problem clearly requires to renounce having one counter per key, rather using a bounded number of counters, independent of the key range's cardinality, and partitioning the key range into some number of intervals compatible with the upper bound on the counters' number. The resulting key intervals will then form as many *buckets*, and what will have to be counted is the number of the objects contained in each bucket.

Counting objects per bucket, rather than per single key, has the following major consequences, the former good, the latter bad:

- *Keys are no longer constrained to be integers, but may rather be elements of any linear order, even of infinite cardinality.*
  In fact, in counting sort keys must be integers – or anything else in one-to-one correspondence with some subset of the integers, such as alphabet letters – since this ensures that the key range contains finitely many keys, so that finitely many counters are needed. Thus, the introduction of an upper bound for the number of counters makes this constraint vanish. As a result, keys of any kind are now allowed and the key range can even be infinite (mathematically, since any representation of the key range on a computer will always be finite). Notably, rational or real numbers may be used as keys, too.
  This observation considerably extends the scope of application of the

special case where function *key* matches the identity function. In counting sort, this option is viable only if the objects to be sorted are themselves integers, whereas in the generalized algorithm it is viable whenever they are elements of any linear order, which also happens if they are rational or real numbers.

- *Recursion needs to be introduced, since any bucket containing more than one object is in turn required to be sorted.*
  In fact, nothing prevents multiple objects from falling in the same bucket, and while this happens sorting is not accomplished. Therefore, the generalized algorithm must provide for recursive rounds, where each round splits any bucket containing multiple objects into finer-grained buckets containing fewer objects. Recursion will then go on until every bucket contains at most one object, viz. until there remains no counter larger than one.

Of course, the fewer recursive rounds are required to complete sorting, the more the algorithm will be efficient, whence the following, fundamental question: how to minimize the number of the rounds? That is to say, how to maximize the probability that, as a result of the execution of a round, there be at most one object in each bucket, so that no more rounds be required?

The intuitive answer is: first, by making the buckets equiprobable – or at least, by making their probabilities as much uniform as possible –, and second, by increasing the number of the buckets as much as possible. Providing pen-and-paper proofs of both of these statements, and showing how they can be enforced, is the purpose of the following sections.

1.1.2 Buckets' probability – Proof

Suppose that k objects be split randomly among n equiprobable buckets, where $k \leq n$. This operation is equivalent to selecting at random a sequence of k buckets, possibly with repetitions, so that the first object be placed into the first bucket of the sequence, the second object into the second bucket, and so on. Thus, the probability P that each bucket will contain at most one object – which will be called *event* E in what follows – is equal to the probability of selecting a sequence without repetitions among all the possible sequences of k buckets formed with the n given ones.

Since buckets are assumed to be equiprobable, so are all such sequences. Hence, P is equal to the ratio of the number of the sequences without repetitions to the number of all sequences, namely:

$$P = \frac{n!}{(n-k)!n^k} \tag{1}$$

In the special case where $k = n$, this equation takes the following, simpler form:

$$P = \frac{n!}{n^n} \tag{2}$$

Now, suppose that the n buckets be no longer equiprobable, viz. that they no longer have the same, uniform probability $1/n$, rather having arbitrary, nonuniform probabilities p_1, ..., p_n. The equation for probability P applying to this case can be obtained through an iterative procedure, as follows.

Let i be an index in the range 1 to n such that p_i is larger than $1/n$. After swapping index i for 1, let x_1 be the increment in probability p_1 with respect to $1/n$, so that $p_1 = a_0/n + x_1$ with $a_0 = 1$ and $0 < x_1 \leq a_0(n-1)/n$ (as $p_1 = 1$ for $x_1 = a_0(n-1)/n$). Then, let P_1 be the probability of event E in case the first bucket has probability p_1 and all the other $n-1$ buckets have the same, uniform probability $q_1 = a_0/n - x_1/(n-1)$.

If $k < n$, event E occurs just in case either all k objects fall in as many distinct buckets with probability q_1, or $k-1$ objects do so whereas the remaining object falls in the bucket with probability p_1. As these events, say E_A and E_B, are incompatible, P_1 matches the sum of their respective probabilities.

Since all the possible choices of k distinct buckets are mutually incompatible, while those of the buckets containing any two distinct objects are mutually independent, the probability of event E_A is equal to the product of the following factors:

- The number of the sequences without repetitions of k buckets formed with the $n-1$ ones with probability q_1, i.e. $(n-1)!/(n-1-k)! = (n-k)(n-1)!/(n-k)!$.
- The probability of any such sequence, i.e. q_1^k.

By virtue of similar considerations, the probability of event E_B turns out to match the product of the following factors:

- The number of the sequences without repetitions of $k-1$ buckets formed with the $n-1$ ones with probability q_1, i.e. $(n-1)!/(n-1-k+1)! = (n-1)!/(n-k)!$.
- The probability of any such sequence, i.e. q_1^{k-1}.
- The number of the possible choices of the object falling in the first bucket, i.e. k.
- The probability of the first bucket, i.e. p_1.

Therefore, P_1 is provided by the following equation:

$$\begin{aligned} P_1 = & \frac{(n-k)(n-1)!}{(n-k)!}\left(\frac{a_0}{n} - \frac{x_1}{n-1}\right)^k \\ & + k\frac{(n-1)!}{(n-k)!}\left(\frac{a_0}{n} - \frac{x_1}{n-1}\right)^{k-1}\left(\frac{a_0}{n} + x_1\right) \end{aligned} \tag{3}$$

The correctness of this equation is confirmed by the fact that its right-hand side matches that of equation (1) for $x_1 = 0$, since P_1 must degenerate to P in this case. In fact, being $a_0 = 1$, it results:

$$\begin{aligned} & \frac{(n-k)(n-1)!}{(n-k)!}\left(\frac{a_0}{n} - 0\right)^k + k\frac{(n-1)!}{(n-k)!}\left(\frac{a_0}{n} - 0\right)^{k-1}\left(\frac{a_0}{n} + 0\right) \\ & = (n - \cancel{k} + \cancel{k})\frac{(n-1)!}{(n-k)!}\left(\frac{a_0}{n}\right)^k \\ & = \frac{n!}{(n-k)!n^k} \end{aligned}$$

If $k = n$, event E_A is impossible, as there is no way to accommodate n objects within $n-1$ buckets without repetitions. Thus, P_1 is given by the following equation, derived by deleting the first addend and replacing k with n in the right-hand side of equation (3):

$$P_1 = n!\left(\frac{a_0}{n} - \frac{x_1}{n-1}\right)^{n-1}\left(\frac{a_0}{n} + x_1\right) \tag{4}$$

Likewise, the right-hand side of this equation matches that of equation (2) for $x_1 = 0$, which confirms its correctness.

The conclusions reached so far can be given a concise form, suitable for generalization, through the following definitions, where i and j are any two natural numbers such that $0 < k - j \leq n - i$ and a_i is some assigned real number:

$$\begin{aligned} A_{i,j} &\equiv \frac{(n-i)!}{(n-i-k+j)!}\left(\frac{a_i}{n-i}\right)^{k-j} \\ F_{i,j} &\equiv (k-j+1)A_{i,j}p_i \\ G_{i,j} &\equiv A_{i,j-1} + F_{i,j} \end{aligned}$$

Then, denoting the value of P in the uniform probability case with P_0, and $a_0(n-1)/n - x_1$ with a_1, so that $q_1 = a_1/(n-1)$, equations (1), (3), and (4) can be rewritten as follows:

$$P_0 = A_{0,0} \tag{5}$$

$$P_1 = \begin{cases} G_{1,1} = A_{1,0} + kA_{1,1}p_1 & \text{if } k < n, \\ F_{1,1} = kA_{1,1}p_1 & \text{if } k = n. \end{cases} \tag{6}$$

Even more than for their conciseness, these equations are significant insofar as they show that the right-hand side of equation (6) can be obtained from the one of equation (5) by replacing $A_{0,0}$ with either $G_{1,1}$ or $F_{1,1}$, depending on whether $k < n$ or $k = n$.

If p_i matches q_1 for any i in the range 2 to n, $P = P_1$, thus P is given by equation (6). Otherwise, the procedure that has led to equation (6) can be applied again. For some index i in the range 2 to n such that p_i is larger than q_1, swap i for 2, and let $x_2 = p_2 - a_1/(n-1)$, $a_2 = a_1(n-2)/(n-1) - x_2$, with $0 < x_2 \leq a_1(n-2)/(n-1)$. Moreover, let P_2 be the probability of event E if the first two buckets have probabilities p_1, p_2 and the other $n-2$ buckets have the same probability $q_2 = a_2/(n-2)$.

Then, reasoning as before, it turns out that the equation for P_2 can be obtained from equation (6) by replacing:

- $A_{1,0}$ with $G_{2,1}$ or $F_{2,1}$, depending on whether $k < n-1$ or $k = n-1$, and
- $A_{1,1}$ with $G_{2,2}$ or $F_{2,2}$, depending on whether $k-1 < n-1$, i.e. $k < n$, or $k-1 = n-1$, i.e. $k = n$.

As a result, P_2 is provided by the following equation:

$$P_2 = \begin{cases} G_{2,1} + kG_{2,2}p_1 = A_{2,0} + kA_{2,1}p_2 + k[A_{2,1} + (k-1)A_{2,2}p_2]p_1 \\ \quad \text{if } k < n-1, \\ F_{2,1} + kG_{2,2}p_1 = kA_{2,1}p_2 + k[A_{2,1} + (k-1)A_{2,2}p_2]p_1 \\ \quad \text{if } k = n-1, \\ kF_{2,2}p_1 = k(k-1)A_{2,2}p_2p_1 \\ \quad \text{if } k = n. \end{cases} \tag{7}$$

Since the iterative procedure used to derive equations (6) and (7) can be further applied as many times as required, it follows that for any nonuniform probability distribution p_1, ..., p_n, the equation for P can be obtained from equation (5) with $n-1$ steps at most, where each step consists of replacing terms of the form $A_{i,j}$ with terms of either form $G_{i+1,j+1}$ or $F_{i+1,j+1}$, depending on whether $k - j < n - i$ or $k - j = n - i$.

Let us re-use letters n, k in lieu of $n - i$ and $k - j$, and use letters a, x as aliases for a_i and x_{i+1}. Then, any aforesaid replacement is equivalent to the insertion of either of the following expressions, regarded as images of as many functions G, F of real variable x:

$$G(x) = \frac{(n-k)(n-1)!}{(n-k)!}\left(\frac{a}{n} - \frac{x}{n-1}\right)^k + k\frac{(n-1)!}{(n-k)!}\left(\frac{a}{n} - \frac{x}{n-1}\right)^{k-1}\left(\frac{a}{n} + x\right) \qquad \text{for } k < n, \tag{8}$$

$$F(x) = n!\left(\frac{a}{n} - \frac{x}{n-1}\right)^{n-1}\left(\frac{a}{n} + x\right) \qquad \text{for } k = n \tag{9}$$

in place of the following expression:

$$\frac{n!}{(n-k)!}\left(\frac{a}{n}\right)^k = \begin{cases} G(0) & \text{if } k < n, \\ F(0) & \text{if } k = n. \end{cases} \tag{10}$$

Equation (10) can be obtained from equations (8) and (9) in the same way as equations (3) and (4) have previously been shown to match equations (1) and (2) for $x_1 = 0$.

Since every such replacement takes place within a sum of nonnegative terms, P can be proven to be increasingly less than P_0 for increasingly nonuniform probability distributions – which implies that the probability of event E is maximum in case of equiprobable buckets – by proving that functions G and F are strictly decreasing in $[0, b]$, where $b = a(n-1)/n$.

The slopes of the segments joining points $(0, G(0))$, $(b, G(b))$ and $(0, F(0))$, $(b, F(b))$ are:

$$\frac{G(b) - G(0)}{b - 0} = \frac{0 - \dfrac{n!}{(n-k)!}\left(\dfrac{a}{n}\right)^k}{b} < 0,$$

$$\frac{F(b) - F(0)}{b - 0} = \frac{0 - n!\left(\dfrac{a}{n}\right)^n}{b} < 0.$$

Therefore, by Lagrange's mean value theorem, there exist $c, d \in (0, b)$ such that $G'(c) < 0$ and $F'(d) < 0$. On the other hand, it is:

$$
\begin{aligned}
G'(x) &= -k\frac{(n-1)!}{(n-k)!}\frac{n-k}{n-1}\left(\frac{a}{n}-\frac{x}{n-1}\right)^{k-1} \\
&\quad - k\frac{(n-1)!}{(n-k)!}\frac{k-1}{n-1}\left(\frac{a}{n}-\frac{x}{n-1}\right)^{k-2}\left(\frac{a}{n}+x\right) \\
&\quad + k\frac{(n-1)!}{(n-k)!}\left(\frac{a}{n}-\frac{x}{n-1}\right)^{k-1}, \\
F'(x) &= -n!\left(\frac{a}{n}-\frac{x}{n-1}\right)^{n-2}\left(\frac{a}{n}+x\right)+n!\left(\frac{a}{n}-\frac{x}{n-1}\right)^{n-1}.
\end{aligned}
$$

Thus, solving equations $G'(x)=0$ and $F'(x)=0$ for $x \neq b$, viz. for $a/n - x/(n-1) \neq 0$, it results:

$$
\begin{aligned}
&G'(x)=0 \\
&\Rightarrow \cancel{k\frac{(n-1)!}{(n-k)!}\left(\frac{a}{n}-\frac{x}{n-1}\right)^{k-2}}\left[\frac{k-n}{n-1}\left(\frac{a}{n}-\frac{x}{n-1}\right)\right. \\
&\qquad \left.+\frac{1-k}{n-1}\left(\frac{a}{n}+x\right)+\frac{a}{n}-\frac{x}{n-1}\right]=0 \\
&\Rightarrow \cancel{\frac{1}{n(n-1)^2}}\big\{(k-n)[(n-1)a-nx]+(1-k)[(n-1)a+n(n-1)x] \\
&\qquad +(n-1)^2a-n(n-1)x\big\}=0 \\
&\Rightarrow \cancel{k(n-1)a}-knx-n(n-1)a+n^2x+(n-1)a+\cancel{n(n-1)x} \\
&\qquad -\cancel{k(n-1)a}-kn(n-1)x+(n-1)^2a-\cancel{n(n-1)x}=0 \\
&\Rightarrow \cancel{-knx}-\cancel{n^2a}+\cancel{na}+n^2x+\cancel{na}-\cancel{a}-kn^2x+\cancel{knx}+\cancel{n^2a}-\cancel{2na}+\cancel{a}=0 \\
&\Rightarrow \cancel{n^2(1-k)}x=0 \\
&\Rightarrow x=0,
\end{aligned}
$$

$$
\begin{aligned}
&F'(x)=0 \\
&\Rightarrow \cancel{n!\left(\frac{a}{n}-\frac{x}{n-1}\right)^{n-2}}\left(-\cancel{\frac{a}{n}}-x+\cancel{\frac{a}{n}}-\frac{x}{n-1}\right)=0 \\
&\Rightarrow \cancel{\frac{n}{1-n}}x=0 \\
&\Rightarrow x=0.
\end{aligned}
$$

Hence, there is no $x \in (0,b)$ such that $G'(x)=0$ or $F'(x)=0$. Moreover, if there existed $y,z \in (0,b)$ such that $G'(y)>0$ or $F'(z)>0$, by Bolzano's theorem there would also exist u,v in the open intervals with endpoints c,y

and d, z, both included in $(0, b)$, such that $G'(u) = 0$ or $F'(v) = 0$, which is not the case. Therefore, $G'(x)$ and $F'(x)$ are negative for any $x \in (0, b)$, so that functions G and F are strictly decreasing in $[0, b]$, Q.E.D..

1.1.3 Buckets' probability – Implementation

Given $n > 1$ buckets, numbered with indices 0 to $n-1$, and a finite set A of objects having minimum key mi and maximum key ma, let $E(k)$, $I(mi, ma)$ be the following events, defined as subsets of the whole range R of function key, with k varying over R:

$$E(k) \equiv \{k' \in R.\ k' \leq k\}$$
$$I(mi, ma) \equiv \{k' \in R.\ mi \leq k' \leq ma\}$$

Furthermore, define functions r, f as follows:

$$r(k, n, mi, ma) \equiv (n-1) \cdot P(E(k) \mid I(mi, ma))$$
$$f(k, n, mi, ma) \equiv floor(r(k, n, mi, ma))$$

where $P(E(k) \mid I(mi, ma))$ denotes the conditional probability of event $E(k)$, viz. for a key not to be larger than k, given event $I(mi, ma)$, viz. if the key is comprised between mi and ma.
Then, the buckets' probabilities can be made as much uniform as possible by placing each object $x \in A$ into the bucket whose index matches the following value:

$$index(key, x, n, mi, ma) \equiv f(key(x), n, mi, ma)$$

For example, given $n = 5$ buckets, suppose that the image of set A under function key consists of keys $k_1 = mi$, k_2, ..., $k_9 = ma$, where the conditional probabilities for a key comprised between k_1 and k_9 to match each of these keys have the following values:

$P_1 = 0.05,$
$P_2 = 0.05,$
$P_3 = 0.15,$
$P_4 = 0.075,$
$P_5 = 0.2,$
$P_6 = 0.025,$
$P_7 = 0.1,$
$P_8 = 0.25,$
$P_9 = 0.1$

Evidently, there is no way of partitioning set $\{k_1, ..., k_9\}$ into five equiprobable subsets comprised of contiguous keys. However, it results:

$$floor\left(4 \cdot \sum_{i=1}^{n} P_i\right) = \begin{cases} 0 & \text{for } n = 1, 2, \\ 1 & \text{for } n = 3, 4, \\ 2 & \text{for } n = 5, 6, 7, \\ 3 & \text{for } n = 8, \\ 4 & \text{for } n = 9. \end{cases}$$

Hence, in spite of the highly nonuniform distribution of the keys' probabilities – key k_8's probability is 10 times that of key k_6 –, function *index* manages all the same to split the objects in A so as to make the buckets' probabilities more uniform – with the maximum one being about 3 times the minimum one –, as follows:

- Bucket 0 has probability 0.1, as it collects the objects with keys k_1, k_2.
- Bucket 1 has probability 0.225, as it collects the objects with keys k_3, k_4.
- Bucket 2 has probability 0.325, as it collects the objects with keys k_5, k_6, k_7.
- Bucket 3 has probability 0.25, as it collects the objects with key k_8.
- Bucket 4 has probability 0.1, as it collects the objects with key k_9.

Remarkably, function *index* makes the buckets' probabilities exactly or almost uniform – meaning that the maximum one is at most twice the minimum nonzero one, possibly except for the last bucket alone – in the following most common, even though special, cases:

1. $I(mi, ma)$ is a finite set of equiprobable keys.

2. $I(mi, ma)$ is a closed interval of real numbers, i.e. $I(mi, ma) = [mi, ma] \subset \mathbb{R}$, with $P(\{mi\}) = 0$, and function r is continuous for $k \in [mi, ma]$.

In case 1, let m be the cardinality of $I(mi, ma)$. It is $m > 0$ since $mi \in I(mi, ma)$, so that each key in $I(mi, ma)$ has probability $1/m$.

If $m \leq n-1$, then $(n-1)/m \geq 1$, thus f is nonzero and strictly increasing for $k \in I(mi, ma)$. Thus, in this subcase function *index* fills exactly m buckets, one for each single key in $I(mi, ma)$, whereas the remaining $n - m$ buckets, particularly the first one, are left unused. Therefore, every used bucket has probability $1/m$.

If $m > n - 1$ and m is divisible by $n - 1$, let $q > 1$ be the quotient of the division, so that $m = q(n - 1)$. Dividing both sides of this equation by $m(n - 1)$, it turns out that $1/(n - 1) = q/m$, and then $1/(n - 1) - 1/m = (q - 1)/m$. Hence, f matches zero for the first $q - 1$ keys in $I(mi, ma)$, increases by one for each of the $n - 2$ subsequent groups of q contiguous keys, and reaches value $n - 1$ in correspondence with the last key. Indeed, $q - 1 + q(n - 2) + 1 = q + q(n - 2) = q(n - 1) = m$.

Consequently, in this subcase function *index* places the objects mapped to the first $q - 1$ keys into the first bucket – which then has probability $(q-1)/m$ –, the objects mapped to the i-th subsequent group of q keys, where $1 \leq i \leq n-2$, into the bucket with index i – which then has probability q/m – and the objects mapped to the last key into the last bucket – which then has probability $1/m$ –. Since $2(q - 1)/m = 2q/m - 2/m \geq 2q/m - q/m = q/m$, the maximum probability is at most twice the minimum one, excluding the last bucket if $q > 2$.

If $m > n-1$ and m is not divisible by $n-1$, let q, r be the quotient and the remainder of the division, where $q > 0$ and $n - 1 > r > 0$. For any $i > 0$, it is:

$$
\begin{aligned}
& m = q(n-1) + r \\
& \quad \Rightarrow \frac{\cancel{m}}{\cancel{m}(n-1)} = \frac{\cancel{q(n-1)}}{m\cancel{(n-1)}} + \frac{r}{m(n-1)} \\
& \quad \Rightarrow \frac{i}{n-1} = \frac{iq}{m} + i\frac{r}{m(n-1)} \\
& \quad \Rightarrow \frac{iq}{m} = \frac{i}{n-1} - \left(i\frac{r}{n-1}\right)\frac{1}{m} \qquad (11) \\
& \quad \Rightarrow \frac{iq+1}{m} = \frac{i}{n-1} + \left(1 - i\frac{r}{n-1}\right)\frac{1}{m} \qquad (12)
\end{aligned}
$$

Both equations (11) and (12) have something significant to say for $i = 1$.

Equation (11) takes the following form:

$$\frac{q}{m} = \frac{1}{n-1} - \left(\frac{r}{n-1}\right)\frac{1}{m}$$

where $r/(n-1) > 0$, so that $q/m < 1/(n-1)$. This implies that, if k is the first key in $I(mi, ma)$ for which f matches any given value, the subsequent $q-1$ keys are never sufficient to increase f by one. Thus, function *index* fills every bucket but the last one – which collects the objects mapped to the last key only – with the objects mapped to $1+q-1 = q$ keys at least. For its part, equation (12) takes the following form:

$$\frac{q+1}{m} = \frac{1}{n-1} + \left(1 - \frac{r}{n-1}\right)\frac{1}{m}$$

where $1 - r/(n-1) > 0$, so that $(q+1)/m > 1/(n-1)$. Therefore, the q keys following any aforesaid key k are always sufficient to increase f by one. Hence, function *index* fills every bucket with the objects mapped to $1+q = q+1$ keys at most. A further consequence is that f changes from zero to one for k matching the $(q+1)$-th key in $I(mi, ma)$, which entails that the first bucket collects the objects mapped to exactly the first q keys. Which is the first i_1, if any, such that the bucket with index i_1 collects the objects mapped to $q+1$, rather than q, keys? Such bucket, if any, is preceded by i_1 buckets (as indices are zero-based), whose total probability is $i_1 q/m$ (as each of those buckets accommodates a group of q keys). So, i_1 is the least index, if any, such that $0 < i_1 < n-1$ and $[(i_1+1)q+1]/m < (i_1+1)/(n-1)$. Rewriting the latter inequality using equation (12), it results:

$$\begin{aligned}
&\frac{\cancel{i_1+1}}{\cancel{n-1}} + \left[1 - (i_1+1)\frac{r}{n-1}\right]\frac{1}{m} < \frac{\cancel{i_1+1}}{\cancel{n-1}} \\
&\quad\Rightarrow \left[1 - (i_1+1)\frac{r}{n-1}\right]\frac{\cancel{1}}{\cancel{m}} < 0 \\
&\quad\Rightarrow (i_1+1)\frac{r}{n-1} > 1 \\
&\quad\Rightarrow i_1 > \frac{n-1}{r} - 1
\end{aligned}$$

where $(n-1)/r - 1 > 0$ since $r < n-1$. Hence, index i_1 there exists just in case:

$$\begin{aligned}
&\frac{n-1}{r} - 1 < n - 2 \\
&\quad \Rightarrow \frac{\cancel{n-1}}{r} < \cancel{n-1} \\
&\quad \Rightarrow r > 1
\end{aligned}$$

Likewise, let i_2 be the next index, if any, such that the bucket with index i_2 accommodates a group of $q+1$ keys. Such bucket, if any, is preceded by $i_2 - 1$ buckets accommodating q keys and one bucket accommodating $q+1$ keys, whose total probability is $(i_2 q + 1)/m$. Thus, i_2 is the least index, if any, such that $i_1 < i_2 < n-1$ and $[(i_2+1)q + 2]/m < (i_2+1)/(n-1)$. Adding term $1/m$ to both sides of equation (12), the latter inequality can be rewritten as follows:

$$\begin{aligned}
&\cancel{\frac{i_2+1}{n-1}} + \left[2 - (i_2+1)\frac{r}{n-1}\right]\frac{1}{m} < \cancel{\frac{i_2+1}{n-1}} \\
&\quad \Rightarrow \left[2 - (i_2+1)\frac{r}{n-1}\right]\cancel{\frac{1}{m}} < 0 \\
&\quad \Rightarrow (i_2+1)\frac{r}{n-1} > 2 \\
&\quad \Rightarrow i_2 > \frac{2(n-1)}{r} - 1
\end{aligned}$$

where $2(n-1)/r - 1 > [(n-1)/r - 1] + 1 \geq i_1$. Hence, index i_2 there exists just in case:

$$\begin{aligned}
&\frac{2(n-1)}{r} - 1 < n - 2 \\
&\quad \Rightarrow \frac{\cancel{2(n-1)}}{r} < \cancel{n-1} \\
&\quad \Rightarrow r > 2
\end{aligned}$$

To sum up, in this subcase function *index* turns out to work as follows:

- The $r-1$ buckets whose indices i_j match the least solutions of inequalities $i_j > j(n-1)/r - 1$, for $1 \leq j \leq r-1$, accommodate a group of $q+1$ contiguous keys each, so that each one has probability $(q+1)/m$.
- The other $n - 1 - (r-1) = n - r$ buckets excluding the last one, particularly the first bucket, accommodate a group of q contiguous keys each, so that each one has probability q/m.

- The last bucket accommodates the last key alone, so that its probability is $1/m$.

Indeed, $(q+1)(r-1)+q(n-r)+1 = \cancel{qr}-q+r-\cancel{1}+qn-\cancel{qr}+\cancel{1} = q(n-1)+r = m$. Furthermore, being $2q/m \geq (q+1)/m$, the maximum value among buckets' probabilities is at most twice the minimum one, excluding the last bucket if $q > 2$.

Two further observations can be made concerning case 1. First, if $m > n-1$, then the larger q gets, the more efficient it becomes to use the buckets' number n itself instead of $n-1$ within function r, placing the objects with index n, viz. mapped to the last key, into the bucket with index $n-1$. In fact, this ensures that all the buckets have almost uniform probabilities rather than leaving a bucket, the last one, with a small, or even negligible, probability.

Second, if keys are integers and $I(mi, ma)$ includes all the integers comprised between mi and ma, it is $m = ma - mi + 1$, whereas the cardinality of set $E(k) \cap I(mi, ma)$ is $k - mi + 1$ for any $k \in I(mi, ma)$. Therefore, it results:

$$r(k, n, mi, ma) = (n-1)\frac{k - mi + 1}{ma - mi + 1},$$

so that function r resembles the approximate rank function R described in [1].

In case 2, let Z be the set of the integers i such that $0 \leq i \leq n-1$. As $r(k, n, mi, ma)$ matches 0 for $k = mi$ and $n-1$ for $k = ma$, by the intermediate value theorem, for each $i \in Z$ there exists a least $k_i \in [mi, ma]$ such that $r(k_i, n, mi, ma) = i$, where $k_0 = mi$. Then, let $B_i = [k_i, k_{i+1})$ for each $i \in Z - \{n-1\}$ and $B_{n-1} = [k_{n-1}, ma]$.

For any $i \in Z - \{n-1\}$, $k \in B_i$, it is $r(k, n, mi, ma) \neq i+1$, since otherwise there would exist some $k < k_{i+1}$ in $[mi, ma]$ such that $r(k, n, mi, ma) = i+1$. On the other hand, being $k < k_{i+1}$, it is $r(k, n, mi, ma) \leq i+1$, since function r is increasing with respect to variable k. Hence, it turns out that $r(k, n, mi, ma) < i+1$. Moreover, the monotonicity of r also implies that $r(k, n, mi, ma) \geq i$. Therefore, it is $f(k, n, mi, ma) = i$, so that for any $i \in Z$, function *index* fills the bucket with index i with the objects mapped to the keys in B_i.

Consequently, for each $i \in Z - \{n-1\}$, the probability of the bucket with index i is:

$$
\begin{aligned}
&P(B_i \mid I(mi, ma)) \\
&\quad = \frac{P(B_i \cap I(mi, ma))}{P(I(mi, ma))} \\
&\quad = \frac{P((k_i, k_{i+1}] \cap I(mi, ma))}{P(I(mi, ma))} \\
&\quad = \frac{P(E(k_{i+1}) \cap I(mi, ma)) - P(E(k_i) \cap I(mi, ma))}{P(I(mi, ma))} \\
&\quad = \frac{P(E(k_{i+1}) \cap I(mi, ma))}{P(I(mi, ma))} - \frac{P(E(k_i) \cap I(mi, ma))}{P(I(mi, ma))} \\
&\quad = P(E(k_{i+1}) \mid I(mi, ma)) - P(E(k_i) \mid I(mi, ma)) \\
&\quad = \frac{(n-1) \cdot P(E(k_{i+1}) \mid I(mi, ma)) - (n-1) \cdot P(E(k_i) \mid I(mi, ma))}{n-1} \\
&\quad = \frac{r(k_{i+1}, n, mi, ma) - r(k_i, n, mi, ma)}{n-1} \\
&\quad = \frac{\cancel{i} + 1 - \cancel{i}}{n-1} \\
&\quad = \frac{1}{n-1}
\end{aligned}
$$

Observe that the computation uses:

- The definition of conditional probability.
- The fact that events B_i and $(k_i, k_{i+1}]$ differ by singletons $\{k_i\}$ and $\{k_{i+1}\}$, whose probability is zero.
  Indeed, it is $P(\{k_0\}) = P(\{mi\}) = 0$ by hypothesis, whereas for any $k \in (mi, ma]$, it is $P(\{k\}) = 0$ due to the continuity of function r, and then of function $P(E(k) \cap I(mi, ma))$, in point k. In fact, for any $k' \in (mi, k)$ it is $E(k') \cap I(mi, ma) = [mi, k'] \subset [mi, k)$, so that $P(E(k') \cap I(mi, ma)) \leq P([mi, k))$. However, it is also $E(k) \cap I(mi, ma) = [mi, k] = [mi, k) \cup \{k\}$, so that $P(E(k) \cap I(mi, ma)) = P([mi, k)) + P(\{k\})$. Thus, if $P(\{k\}) > 0$, then $P(E(k) \cap I(mi, ma)) > P([mi, k))$, in contradiction with the assumption that:

$$\lim_{k' \to k^-} P(E(k') \cap I(mi, ma)) = P(E(k) \cap I(mi, ma))$$

- The fact that event $E(k_{i+1}) \cap I(mi, ma)$ is equal to the union of the disjoint events $E(k_i) \cap I(mi, ma)$ and $(k_i, k_{i+1}] \cap I(mi, ma)$, so that the probability of the former event is equal to the sum of the probabilities of the latter ones.

As a result, all the buckets but the last one are equiprobable, whereas the last one has probability zero. Thus, in this case it is again more efficient to replace $n-1$ with n within function r, assigning the objects with index n, viz. mapped to the keys falling in B_n, to the bucket with index $n-1$, which ensures that all the buckets have uniform probabilities.

If function r is linear for $k \in [mi, ma]$, viz. if interval $[mi, ma]$ is endowed with a constant probability density, then the function's graph (with factor $n-1$ replaced by n) is the straight line passing through points $(mi, 0)$ and (ma, n). Therefore, it results:

$$r(k, n, mi, ma) = n\frac{k - mi}{ma - mi},$$

so that function r matches the approximate rank function R described in [1].

1.1.4 Buckets' number – Proof

Given n equiprobable buckets and k objects to be partitioned randomly among such buckets, where $1 < k \leq n$, the probability $P_{n,k}$ that each bucket will contain at most one object is given by equation (1), namely:

$$P_{n,k} = \frac{n!}{(n-k)!n^k}$$

Thus, it is:

$$\begin{aligned} & P_{n+1,k} - P_{n,k} \\ & \quad = \frac{(n+1)!}{(n-k+1)(n-k)!(n+1)^k} - \frac{n!}{(n-k)!n^k} \\ & \quad = \frac{(n+1)!n^k - n!(n-k+1)(n+1)^k}{(n-k+1)(n-k)!n^k(n+1)^k} \end{aligned}$$

Using the binomial theorem and Pascal's rule, the numerator of the fraction in the right-hand side of this equation can be expressed as follows:

$$
\begin{aligned}
&(n+1)!n^k-(n-k+1)n!(n+1)^k\\
&\quad=n!(n+1)n^k+n!k(n+1)^k-n!n(n+1)^k-n!(n+1)^k\\
&\quad=n!n^{k+1}+n!n^k\\
&\qquad+n!k\left[n^k+\binom{k}{1}n^{k-1}+\binom{k}{2}n^{k-2}+...+\binom{k}{k-1}n+\binom{k}{k}\right]\\
&\qquad-n!n\left[n^k+\binom{k}{1}n^{k-1}+\binom{k}{2}n^{k-2}+...+\binom{k}{k-1}n+\binom{k}{k}\right]\\
&\qquad-n!\left[n^k+\binom{k}{1}n^{k-1}+\binom{k}{2}n^{k-2}+...+\binom{k}{k-1}n+\binom{k}{k}\right]\\
&\quad=\cancel{n!n^{k+1}}+\cancel{n!n^k}+\cancel{n!kn^k}\\
&\qquad+n!k\binom{k}{1}n^{k-1}+n!k\binom{k}{2}n^{k-2}+...+n!k\binom{k}{k-1}n+n!k\\
&\qquad-\cancel{n!n^{k+1}}-\cancel{n!kn^k}-n!\binom{k}{2}n^{k-1}-...-n!\binom{k}{k-1}n^2-n!\binom{k}{k}n\\
&\qquad-\cancel{n!n^k}-n!\binom{k}{1}n^{k-1}-n!\binom{k}{2}n^{k-2}-...-n!\binom{k}{k-1}n-n!\\
&\quad=n!(k-1)+n!n^{k-1}\left[k\binom{k}{1}-\binom{k}{1}-\binom{k}{2}\right]+...\\
&\qquad+n!n\left[k\binom{k}{k-1}-\binom{k}{k-1}-\binom{k}{k}\right]\\
&\quad=n!(k-1)+n!\cdot\sum_{i=1}^{k-1}n^{k-i}\left\{k\binom{k}{i}-\left[\binom{k}{i}+\binom{k}{i+1}\right]\right\}\\
&\quad=n!(k-1)+n!\cdot\sum_{i=1}^{k-1}n^{k-i}\left[k\binom{k}{i}-\binom{k+1}{i+1}\right]\\
&\quad=n!(k-1)+n!\cdot\sum_{i=1}^{k-1}n^{k-i}\left[\frac{kk!}{i!(k-i)!}-\frac{(k+1)!}{(i+1)i!(k-i)!}\right]\\
&\quad=n!(k-1)+n!\cdot\sum_{i=1}^{k-1}n^{k-i}k!\frac{(i+1)k-(k+1)}{(i+1)i!(k-i)!}\\
&\quad=n!(k-1)+n!\cdot\sum_{i=1}^{k-1}n^{k-i}k!\frac{ik-1}{(i+1)i!(k-i)!}>0
\end{aligned}
$$

Therefore, for any fixed $k>1$, sequence $(P_{n,k})_{n\geq k}$ is strictly increasing, viz. the larger n is, such is also the probability that each of the n equiprobable buckets will contain at most one of the k given objects.

Moreover, it is $P_{n,k}=[n(n-1)(n-2)(n-3)...(n-k+1)]/n^k<n^k/n^k=1$,

as the product enclosed within the square brackets comprises k factors, one equal to n and the other ones less than n.
On the other hand, it turns out that:

$$\begin{aligned}
&n(n-1)(n-2)(n-3)...(n-k+1) \\
&\quad = n^2[(n-2)(n-3)...(n-k+1)] - n[(n-2)(n-3)...(n-k+1)] \\
&\quad \geq n^2[(n-2)(n-3)...(n-k+1)] - n \cdot n^{k-2} \\
&\quad = n[n(n-2)(n-3)...(n-k+1)] - n^{k-1} \\
&\quad = n \cdot n^2[(n-3)...(n-k+1)] - n \cdot 2n[(n-3)...(n-k+1)] - n^{k-1} \\
&\quad \geq n^3[(n-3)...(n-k+1)] - 2n^2 \cdot n^{k-3} - n^{k-1} \\
&\quad = n^2[n(n-3)...(n-k+1)] - (1+2)n^{k-1} \ ...
\end{aligned}$$

Hence, applying the same line of reasoning until the product within the square brackets disappears, it results:

$$\begin{aligned}
&n(n-1)(n-2)(n-3)...(n-k+1) \\
&\quad \geq n^k - [1+2+...+(k-1)]n^{k-1} \\
&\quad = n^k - \frac{k(k-1)}{2}n^{k-1},
\end{aligned}$$

so that:

$$P_{n,k} = \frac{n(n-1)(n-2)(n-3)...(n-k+1)}{n^k} \geq 1 - \frac{k(k-1)}{2n}$$

Therefore, for any fixed $k > 1$, the terms of sequence $(P_{n,k})_{n \geq k}$ are comprised between the corresponding terms of sequence $(1 - k(k-1)/2n)_{n \geq k}$ and constant sequence $(1)_{n \geq k}$. Since both of these sequences converge to 1, by the squeeze theorem it is:

$$\lim_{n \to \infty} P_{n,k} = 1,$$

viz. the larger n is, the closer to 1 is the probability that each of the n equiprobable buckets will contain at most one of the k given objects.
As a result, the probability of placing at most one object into each bucket in any algorithm's round is maximized by increasing the number of the buckets as much as possible, Q.E.D..

1.1.5 Buckets' number – Implementation

Let n be the number of the objects to be sorted, and p the upper bound on the counters' number – and then on the buckets' number as well, since there must be exactly one counter per bucket –. This means that before the round begins, the objects to be split are located in m buckets B_1, ..., B_m, where $0 < m \leq p$, respectively containing n_1, ..., n_m objects, where $n_i > 0$ for each i from 1 to m and $n_1 + ... + n_m = n$.

Moreover, let c be the number of the objects known to be the sole elements of their buckets, viz. to require no partition into finer-grained buckets, at the beginning of a given algorithm's round. Then, the number of the objects requiring to be split into finer-grained buckets in that round is $n-c$, whereas the number of the available buckets is $p-c$, since c counters must be left to store as many 1s, one for each singleton bucket.

How to compute c? At first glance, the answer seems trivial: by counting, among the counters input (either by the algorithm's caller or by the previous round) to the round under consideration, those that match 1. However, this value would not take into account the fact that, for each non-singleton bucket, the algorithm must find the leftmost occurrence of the minimum key, as well as the rightmost occurrence of the maximum key, and place the corresponding objects into two new singleton buckets, which shall be the first and the last finer-grained bucket, respectively.

The most fundamental reason for this is that, as a result of the partition of such a bucket, nothing prevents all its objects from falling in the same finer-grained bucket – particularly, this happens whenever all its objects have the same key –, in which case the algorithm does not terminate unless some object is removed from the bucket prior to the partition, so as to reduce its size. Just as clearly, the algorithm must know where to place the finer-grained buckets containing the removed objects with respect to the finer-grained buckets resulting from the partition. This is exactly what is ensured by removing objects with minimum or maximum keys, whereas selecting the leftmost or the rightmost ones, respectively, preserves the algorithm's stability.

Actually, the algorithm's termination requires the removal of at least one object per non-singleton bucket, so the removal of one object only, either with minimum or maximum key, would be sufficient. Nonetheless, the leftmost minimum and the rightmost maximum can be searched via a single loop, and finding both of them enables to pass them as inputs to the function *index* described in section 1.1.3, or to whatever other function used to split buckets into finer-grained ones. Moreover, non-singleton buckets whose objects all have the same key can be detected as those whose minimum and maximum keys are equal. This allows to optimize the algorithm by preventing it from unnecessarily applying multiple recursive rounds to any such bucket; it shall

rather be left as is, just replacing its counter with as many 1s as its size to indicate that it is already sorted.

Therefore, as the round begins, the objects already known to be placed in singleton buckets are one for each bucket whose counter matches 1, and two for each bucket whose counter is larger than 1. As a result, c shall be computed as follows. First, initialize c to zero. Then, for each i from 1 to m, increase c by one if $n_i = 1$, by two otherwise.

Conversely, for any such i, the number N_i of the objects contained in bucket B_i having to be partitioned into finer-grained buckets is 0 if $n_i = 1$, $n_i - 2$ otherwise, so that $N_1 + ... + N_m = n - c$. According to the findings of section 1.1.4, the number N'_i of the resulting finer-grained buckets should be maximized, and the most efficient way to do this is to render N'_i proportional to N_i, since otherwise, viz. if some buckets were preferred to some other ones, the unprivileged buckets would form as many bottlenecks.

This can be accomplished by means of the following procedure. First, initialize integers R and U to 0. Then, for each i from 1 to m, check whether $N_i \leq 1$:

- If so, set N'_i to N_i.
  In fact, no finer-grained bucket is necessary if there are no objects to be split, while a single finer-grained bucket is sufficient for a lonely object.

- Otherwise, perform the integer division of $N_i \cdot (p - c) + R$ by $n - c$, and set integer Q to the resulting quotient and R to the resulting remainder. Then, if the minimum and maximum keys occurring in bucket B_i are equal, increase U by $Q - N_i$, otherwise set N'_i to $U + Q$ and reset U to 0.
  In fact, as observed above, if its minimum and maximum keys are equal, bucket B_i can be split into $n_i = N_i + 2$ singleton buckets. Hence, the difference $Q - N_i$ between the number of the available finer-grained buckets, i.e. $Q + 2$ (where 2 is the number of the buckets containing the leftmost minimum and the rightmost maximum), and the number of those being used, i.e. $N_i + 2$, can be added to the total number U of the available finer-grained buckets still unused in the current round. Such buckets can then be utilized as soon as a bucket B_j with $N_j > 1$ whose minimum and maximum keys do not match is encountered next, in addition to those already reserved for B_j.

Of course, for any i from 1 to m such that $N_i > 1$, it is $N_i \leq Q$ – viz. the number of the objects in B_i to be split is not larger than that of the finer-grained buckets where they are to be placed even if $U = 0$, so that the probability of placing at most one object into each bucket is nonzero – just in case $n - c \leq p - c$, i.e. $n \leq p$. Indeed, it will be formally proven that

for $n \leq p$, this procedure is successful in maximizing the buckets' number in each round never exceeding the upper bound p.

1.1.6 Generalized counting sort (GCsort)

The conclusions of the efficiency analysis performed so far, put together, result in the following *generalized counting sort (GCsort)* algorithm.

Let xs be the input array containing n objects to be sorted, and ns an array of p integers, where $0 < p$ and $n \leq p$. Moreover, let xs' and ns' be two further arrays of the same type and size of xs and ns, respectively, and let i, i', and j be as many integers.

Then, GCsort works as follows (assuming arrays to be zero-based):

1. Initialize the first element of ns to n and any other element to 0.
2. Check whether ns contains any element larger than 1.
   If not, terminate the algorithm and output xs as the resulting sorted array.
3. Initialize i, i', and j to 0.
4. Check whether $ns[i] = 1$ or $ns[i] > 1$:
   (a) In the former case, set $xs'[j]$ to $xs[j]$ and $ns'[i']$ to 1.
   Then, increase i' and j by 1.
   (b) In the latter case, partition the bucket comprised of objects $xs[j]$ to $xs[j + ns[i] - 1]$ into finer-grained buckets according to section 1.1.5, storing the resulting n' buckets in $xs'[j]$ to $xs'[j + ns[i] - 1]$ and their sizes in $ns'[i']$ to $ns'[i' + n' - 1]$.
   Then, increase i' by n' and j by $ns[i]$.
5. Increase i by 1, and then check whether $i < p$.
   If so, go back to step 4.
   Otherwise, perform the following operations:
   (a) If $i' < p$, set integers $ns'[i']$ to $ns'[p - 1]$ to 0.
   (b) Swap addresses xs and xs', as well as addresses ns and ns'.
   (c) Go back to step 2.

Since the algorithm is tail-recursive, the memory space required for its execution matches the one required for a single recursive round, which is $O(n) + O(p)$.

The best-case running time can be computed easily. The running time taken by step 1 is equal to p. Moreover, the partition of a bucket into finer-grained

ones is performed by determining their sizes, computing the cumulative sum of such sizes, and rearranging the bucket's objects according to the resulting offsets. All these operations only involve sequential, non-nested loops, which iterate through either the objects or the finer-grained counters pertaining to the processed bucket alone. Hence, the running time taken by a single recursive round is $O(n) + O(p)$, so that in the best case where at most one round is executed after step 1, the running time taken by the algorithm is $O(n) + O(p)$, too.

The asymptotic worst-case running time can be computed as follows. Let $t_{n,p}$ be the worst-case running time taken by a single round. As $t_{n,p}$ is $O(n) + O(p)$, there exist three real numbers $a > 0$, $b > 0$, and c such that $t_{n,p} \leq an + bp + c$. Moreover, let $U_{n,p}$ be the set of the p-tuples of natural numbers such that the sum of their elements matches n, and $max(u)$ the maximum element of a given $u \in U_{n,p}$. Finally, let $T_{n,p,u}$ be the worst-case running time taken by the algorithm if it starts from step 2, viz. skipping step 1, using as initial content of array ns the p-tuple $u \in U_{n,p}$.

Then, it can be proven by induction on $max(u)$ that:

$$T_{n,p,u} \leq \begin{cases} a\dfrac{max(u)}{2}n + \left[b\dfrac{max(u)}{2} + 1\right]p + c\dfrac{max(u)}{2} \\ \quad \text{if } max(u) \text{ is even,} \\ a\dfrac{max(u)-1}{2}n + \left[b\dfrac{max(u)-1}{2} + 1\right]p + c\dfrac{max(u)-1}{2} \\ \quad \text{if } max(u) \text{ is odd} \end{cases} \tag{13}$$

In fact, if $max(u) = 0$, the initial p-tuple u matches the all-zero one. Hence, the algorithm executes step 2 just once and then immediately terminates. Therefore, the running time is p, which matches the right-hand side of inequality (13) for $max(u) = 0$.

If $max(u) = 1$, u contains no element larger than 1. Thus, again, the algorithm terminates just after the first execution of step 2. As a result, the running time is still p, which matches the right-hand side of inequality (13) for $max(u) = 1$.

If $max(u) = 2$, u contains some element larger than 1, so that one round is executed, taking time $t_{n,p}$ in the worst case. Once this round is over, array ns will contain a p-tuple $u' \in U_{n,p}$ such that $max(u') = 1$. Hence, step 2 is executed again, taking time p, and then the algorithm terminates. As a result, it is:

$$T_{n,p,u} \leq an + bp + c + p = an + (b+1)p + c,$$

which matches the right-hand side of inequality (13) for $max(u) = 2$.

Finally, if $max(u) > 2$, u has some element larger than 1, so one round is executed, taking time $t_{n,p}$ in the worst case. Once this round is over, array ns will contain a p-tuple $u' \in U_{n,p}$ such that $max(u') \leq max(u) - 2$, because of the removal of the leftmost minimum and the rightmost maximum from any non-singleton bucket. By the induction hypothesis, the worst-case time $T_{n,p,u'}$ taken by the algorithm from this point onward complies with inequality (13), whose right-hand side is maximum if such is $max(u')$, viz. if $max(u') = max(u) - 2$.

As a result, if $max(u)$ is even, it is:

$$\begin{aligned} T_{n,p,u} &\leq an + bp + c \\ &\quad + a\frac{max(u) - 2}{2}n + \left[b\frac{max(u) - 2}{2} + 1\right]p + c\frac{max(u) - 2}{2} \\ &= a\frac{max(u)}{2}n + \left[b\frac{max(u)}{2} + 1\right]p + c\frac{max(u)}{2}, \end{aligned}$$

which matches the right-hand side of inequality (13) for an even $max(u)$.

Similarly, if $max(u)$ is odd, it is:

$$\begin{aligned} T_{n,p,u} &\leq an + bp + c \\ &\quad + a\frac{max(u) - 3}{2}n + \left[b\frac{max(u) - 3}{2} + 1\right]p + c\frac{max(u) - 3}{2} \\ &= a\frac{max(u) - 1}{2}n + \left[b\frac{max(u) - 1}{2} + 1\right]p + c\frac{max(u) - 1}{2}, \end{aligned}$$

which matches the right-hand side of inequality (13) for an odd $max(u)$.

With this, the proof of inequality (13) is complete. Now, let $T_{n,p}$ be the worst-case time taken by the algorithm executed in full. Step 1 is executed first, taking time p. Then, array ns contains a p-tuple u such that $max(u) = n$, and by definition, the worst-case time taken by the algorithm from this point onward is $T_{n,p,u}$. Therefore, applying inequality (13), it turns out that:

$$\begin{aligned} T_{n,p} &= p + T_{n,p,u} \\ &\leq \begin{cases} a\dfrac{n^2}{2} + \left(b\dfrac{n}{2} + 2\right)p + c\dfrac{n}{2} & \text{if } n \text{ is even,} \\ a\dfrac{n^2 - n}{2} + \left(b\dfrac{n-1}{2} + 2\right)p + c\dfrac{n-1}{2} & \text{if } n \text{ is odd} \end{cases} \end{aligned}$$

As a result, the asymptotic worst-case running time taken by the algorithm is $O(n^2) + O(np)$.

1.2 Formal definitions

Here below, a formal definition of GCsort is provided, which will later enable to formally prove the correctness of the algorithm. Henceforth, the main points of the formal definitions and proofs are commented. For further information, see Isabelle documentation, particularly [6], [5], [4], and [3].

The following formalization of GCsort does not define any specific function *index* to be used to split buckets into finer-grained ones. It rather defines only the type *index-sign* of such functions, matching the signature of the function *index* described in section 1.1.3, along with three predicates that whatever chosen *index* function is required to satisfy for GCsort to work correctly:

- Predicate *index-less* requires function *index* to map any object within a given bucket to an index less than the number of the associated finer-grained buckets (*less than* instead of *not larger than* since type *'a list* is zero-based).
- Predicate *index-mono* requires function *index* to be monotonic with respect to the keys of the objects within a given bucket.
- Predicate *index-same* requires function *index* to map any two distinct objects within a given bucket that have the same key to the same index (premise *distinct* is added to enable this predicate to be used by the simplifier).

type-synonym ('a, 'b) *index-sign* = ('a ⇒ 'b) ⇒ 'a ⇒ *nat* ⇒ 'b ⇒ 'b ⇒ *nat*

definition *index-less* :: ('a, 'b::*linorder*) *index-sign* ⇒ ('a ⇒ 'b) ⇒ *bool*
where
index-less index key ≡
  ∀ *x n mi ma*. *key x* ∈ {*mi*..*ma*} ⟶ *0* < *n* ⟶
    index key x n mi ma < *n*

definition *index-mono* :: ('a, 'b::*linorder*) *index-sign* ⇒ ('a ⇒ 'b) ⇒ *bool*
where
index-mono index key ≡
  ∀ *x y n mi ma*. {*key x*, *key y*} ⊆ {*mi*..*ma*} ⟶ *key x* ≤ *key y* ⟶
    index key x n mi ma ≤ *index key y n mi ma*

definition *index-same* :: ('a, 'b::*linorder*) *index-sign* ⇒ ('a ⇒ 'b) ⇒ *bool*
where

```
index-same index key ≡
  ∀ x y n mi ma. key x ∈ {mi..ma} ⟶ x ≠ y ⟶ key x = key y ⟶
    index key x n mi ma = index key y n mi ma
```

Functions *bn-count* and *bn-comp* count, respectively, the objects known to be placed in singleton buckets in a given round, and the finer-grained buckets available to partition a given non-singleton bucket, according to section 1.1.5.

```
fun bn-count :: nat list ⇒ nat where
bn-count [] = 0 |
bn-count (Suc (Suc (Suc (Suc n))) # ns) = Suc (Suc (bn-count ns)) |
bn-count (n # ns) = n + bn-count ns

fun bn-comp :: nat ⇒ nat ⇒ nat ⇒ nat ⇒ nat × nat where
bn-comp (Suc (Suc n)) p q r =
  ((Suc (Suc n) * p + r) div q, (Suc (Suc n) * p + r) mod q) |
bn-comp n p q r = (n, r)

fun bn-valid :: nat ⇒ nat ⇒ nat ⇒ bool where
bn-valid (Suc (Suc n)) p q = (q ∈ {0<..p}) |
bn-valid n p q = True
```

Functions *mini* and *maxi* return the indices of the leftmost minimum and the rightmost maximum within a given non-singleton bucket.

```
primrec (nonexhaustive) mini :: 'a list ⇒ ('a ⇒ 'b::linorder) ⇒ nat where
mini (x # xs) key =
  (let m = mini xs key in if xs = [] ∨ key x ≤ key (xs ! m) then 0 else Suc m)

primrec (nonexhaustive) maxi :: 'a list ⇒ ('a ⇒ 'b::linorder) ⇒ nat where
maxi (x # xs) key =
  (let m = maxi xs key in if xs = [] ∨ key (xs ! m) < key x then 0 else Suc m)
```

Function *enum* counts the objects contained in each finer-grained bucket reserved for the partition of a given non-singleton bucket.

```
primrec enum :: 'a list ⇒ ('a, 'b) index-sign ⇒ ('a ⇒ 'b) ⇒
  nat ⇒ 'b ⇒ 'b ⇒ nat list where
enum [] index key n mi ma = replicate n 0 |
enum (x # xs) index key n mi ma =
  (let i = index key x n mi ma;
     ns = enum xs index key n mi ma
   in ns[i := Suc (ns ! i)])
```

Function *offs* computes the cumulative sum of the resulting finer-grained buckets' sizes so as to generate the associated offsets' list.

```
primrec offs :: nat list ⇒ nat ⇒ nat list where
offs [] i = [] |
offs (n # ns) i = i # offs ns (i + n)
```

Function *fill* fills the finer-grained buckets with their respective objects.

```
primrec fill :: 'a list ⇒ nat list ⇒ ('a, 'b) index-sign ⇒ ('a ⇒ 'b) ⇒
  nat ⇒ 'b ⇒ 'b ⇒ 'a option list where
fill [] ns index key n mi ma = replicate n None |
fill (x # xs) ns index key n mi ma =
  (let i = index key x (length ns) mi ma;
     ys = fill xs (ns[i := Suc (ns ! i)]) index key n mi ma
   in ys[ns ! i := Some x])
```

Then, function *round* formalizes a single GCsort's recursive round.

```
definition round-suc-suc :: ('a, 'b::linorder) index-sign ⇒ ('a ⇒ 'b) ⇒
  'a list ⇒ nat ⇒ nat ⇒ nat ⇒ nat × nat list × 'a list where
round-suc-suc index key ws n n' u ≡
  let nmi = mini ws key; nma = maxi ws key;
    xmi = ws ! nmi; xma = ws ! nma; mi = key xmi; ma = key xma
  in if mi = ma
    then (u + n' − n, replicate (Suc (Suc n)) (Suc 0), ws)
    else
      let k = case n of Suc (Suc i) ⇒ u + n' | - ⇒ n;
        zs = nths ws (− {nmi, nma}); ms = enum zs index key k mi ma
      in (u + n' − k, Suc 0 # ms @ [Suc 0],
        xmi # map the (fill zs (offs ms 0) index key n mi ma) @ [xma])

fun round :: ('a, 'b::linorder) index-sign ⇒ ('a ⇒ 'b) ⇒ nat ⇒ nat ⇒ nat ⇒
  nat × nat list × 'a list ⇒ nat × nat list × 'a list where
round index key p q r (u, [], xs) = (u, [], xs) |
round index key p q r (u, 0 # ns, xs) = round index key p q r (u, ns, xs) |
round index key p q r (u, Suc 0 # ns, xs) =
  (let (u', ns', xs') = round index key p q r (u, ns, tl xs)
   in (u', Suc 0 # ns', hd xs # xs')) |
round index key p q r (u, Suc (Suc n) # ns, xs) =
  (let ws = take (Suc (Suc n)) xs; (n', r') = bn-comp n p q r;
     (v, ms', ws') = round-suc-suc index key ws n n' u;
     (u', ns', xs') = round index key p q r' (v, ns, drop (Suc (Suc n)) xs)
   in (u', ms' @ ns', ws' @ xs'))
```

Finally, function *gcsort-aux* formalizes GCsort. Since the algorithm is tail-recursive, this function complies with the requirements for an auxiliary tail-recursive function applying to step 1 of the proof method described in [7] – henceforth briefly referred to as the *proof method* –. This feature will later enable to formally prove the algorithm's correctness properties by means of such method.

abbreviation *gcsort-round* :: ('*a*, '*b*::*linorder*) *index-sign* ⇒ ('*a* ⇒ '*b*) ⇒
nat ⇒ *nat list* ⇒ '*a list* ⇒ *nat* × *nat list* × '*a list* **where**
gcsort-round index key p ns xs ≡
round index key (*p* − *bn-count ns*) (*length xs* − *bn-count ns*) *0* (*0*, *ns*, *xs*)

function *gcsort-aux* :: ('*a*, '*b*::*linorder*) *index-sign* ⇒ ('*a* ⇒ '*b*) ⇒ *nat* ⇒
nat × *nat list* × '*a list* ⇒ *nat* × *nat list* × '*a list* **where**
gcsort-aux index key p (*u*, *ns*, *xs*) = (*if find* (λ*n*. *Suc 0* < *n*) *ns* = *None*
then (*u*, *ns*, *xs*)
else gcsort-aux index key p (*gcsort-round index key p ns xs*))
by *auto*

First of all, even before accomplishing step 2 of the proof method, it is necessary to prove that function *gcsort-aux* always terminates by showing that the maximum bucket's size decreases in each recursive round.

lemma *add-zeros*:
foldl (+) (*m* :: *nat*) (*replicate n 0*) = *m*
by (*induction n*, *simp-all*)

lemma *add-suc*:
foldl (+) (*Suc m*) *ns* = *Suc* (*foldl* (+) *m ns*)
by (*induction ns arbitrary*: *m*, *simp-all*)

lemma *add-update*:
i < *length ns* ⟹ *foldl* (+) *m* (*ns*[*i* := *Suc* (*ns* ! *i*)]) = *Suc* (*foldl* (+) *m ns*)
by (*induction ns arbitrary*: *i m*, *simp-all add*: *add-suc split*: *nat.split*)

lemma *add-le*:
(*m* :: *nat*) ≤ *foldl* (+) *m ns*
by (*induction ns arbitrary*: *m*, *simp-all*, *rule order-trans*, *rule le-add1*)

lemma *add-mono*:
(*m* :: *nat*) ≤ *n* ⟹ *foldl* (+) *m ns* ≤ *foldl* (+) *n ns*
by (*induction ns arbitrary*: *m n*, *simp-all*)

lemma *add-max* [*rule-format*]:
ns ≠ [] ⟶ *Max* (*set ns*) ≤ *foldl* (+) (*0* :: *nat*) *ns*

by (*induction ns*, *simp-all add*: *add-le*, *erule impCE*, *simp*, *rule ballI*, *drule bspec*, *assumption*, *rule order-trans*, *assumption*, *rule add-mono*, *simp*)

lemma *enum-length*:
length (*enum xs index key n mi ma*) = *n*
by (*induction xs*, *simp-all add*: *Let-def*)

lemma *enum-add-le*:
foldl (+) *0* (*enum xs index key n mi ma*) ≤ *length xs*
proof (*induction xs*, *simp-all add*: *Let-def add-zeros*)
fix *x xs*
let *?i* = *index key x n mi ma*
assume *foldl* (+) *0* (*enum xs index key n mi ma*) ≤ *length xs*
(**is** *foldl - - ?ns* ≤ -)
thus *foldl* (+) *0* (*?ns*[*?i* := *Suc* (*?ns* ! *?i*)]) ≤ *Suc* (*length xs*)
by (*cases ?i* < *length ?ns*, *simp-all add*: *add-update*)
qed

lemma *enum-max-le*:
0 < *n* ⟹ *Max* (*set* (*enum xs index key n mi ma*)) ≤ *length xs*
(**is** - ⟹ *Max* (*set ?ns*) ≤ -)
by (*insert add-max* [*of ?ns*], *insert enum-add-le* [*of xs index key n mi ma*], *simp only*: *length-greater-0-conv* [*symmetric*] *enum-length*, *simp*)

lemma *mini-less*:
0 < *length xs* ⟹ *mini xs key* < *length xs*
by (*induction xs*, *simp-all add*: *Let-def*)

lemma *maxi-less*:
0 < *length xs* ⟹ *maxi xs key* < *length xs*
by (*induction xs*, *simp-all add*: *Let-def*)

lemma *mini-lb*:
x ∈ *set xs* ⟹ *key* (*xs* ! *mini xs key*) ≤ *key x*
by (*induction xs*, *simp-all add*: *Let-def*, *auto*)

lemma *maxi-ub*:
x ∈ *set xs* ⟹ *key x* ≤ *key* (*xs* ! *maxi xs key*)
by (*induction xs*, *simp-all add*: *Let-def*, *auto*)

lemma *mini-maxi-neq* [*rule-format*]:
Suc 0 < *length xs* ⟶ *mini xs key* ≠ *maxi xs key*
proof (*induction xs*, *simp-all add*: *Let-def*, *rule conjI*, (*rule impI*)+, (*rule-tac* [*2*] *impI*)+, *rule-tac* [*2*] *notI*, *simp-all*, *rule ccontr*)
fix *x xs*
assume *key* (*xs* ! *maxi xs key*) < *key x* **and** *key x* ≤ *key* (*xs* ! *mini xs key*)
hence *key* (*xs* ! *maxi xs key*) < *key* (*xs* ! *mini xs key*) **by** *simp*
moreover assume *xs* ≠ []
hence *0* < *length xs* **by** *simp*

hence *mini xs key* < *length xs*
by (*rule mini-less*)
hence *xs* ! *mini xs key* ∈ *set xs* **by** *simp*
hence *key* (*xs* ! *mini xs key*) ≤ *key* (*xs* ! *maxi xs key*)
by (*rule maxi-ub*)
ultimately show *False* **by** *simp*
qed

lemma *mini-maxi-nths*:
length (*nths xs* (− {*mini xs key*, *maxi xs key*})) =
(*case length xs of* 0 ⇒ 0 | *Suc* 0 ⇒ 0 | *Suc* (*Suc n*) ⇒ *n*)
proof (*simp add*: *length-nths split*: *nat.split*, *rule allI*, *rule conjI*, *rule-tac* [*2*] *allI*,
(*rule-tac* [!] *impI*)+, *simp add*: *length-Suc-conv*, *erule exE*, *simp*, *blast*)
fix *n*
assume *A*: *length xs* = *Suc* (*Suc n*)
hence *B*: *Suc* 0 < *length xs* **by** *simp*
hence *C*: 0 < *length xs* **by** *arith*
have {*i*. *i* < *Suc* (*Suc n*) ∧ *i* ≠ *mini xs key* ∧ *i* ≠ *maxi xs key*} =
{..<*Suc* (*Suc n*)} − {*mini xs key*} − {*maxi xs key*}
by *blast*
thus *card* {*i*. *i* < *Suc* (*Suc n*) ∧ *i* ≠ *mini xs key* ∧ *i* ≠ *maxi xs key*} = *n*
by (*simp add*: *card-Diff-singleton-if*, *insert mini-maxi-neq* [*OF B*, *of key*],
simp add: *mini-less* [*OF C*] *maxi-less* [*OF C*] *A* [*symmetric*])
qed

lemma *mini-maxi-nths-le*:
length xs ≤ *Suc* (*Suc n*) ⟹ *length* (*nths xs* (− {*mini xs key*, *maxi xs key*})) ≤ *n*
by (*simp add*: *mini-maxi-nths split*: *nat.split*)

lemma *round-nil*:
(*fst* (*snd* (*round index key p q r t*)) ≠ []) = (∃ *n* ∈ *set* (*fst* (*snd t*)). 0 < *n*)
by (*induction index key p q r t rule*: *round.induct*,
simp-all add: *round-suc-suc-def Let-def split*: *prod.split*)

lemma *round-max-eq* [*rule-format*]:
fst (*snd t*) ≠ [] ⟶ *Max* (*set* (*fst* (*snd t*))) = *Suc* 0 ⟶
Max (*set* (*fst* (*snd* (*round index key p q r t*)))) = *Suc* 0
proof (*induction index key p q r t rule*: *round.induct*, *simp-all add*: *Let-def split*:
prod.split del: *all-simps*, *rule impI*, (*rule-tac* [*2*] *allI*)+, (*rule-tac* [*2*] *impI*)+,
(*rule-tac* [*3*] *allI*)+, (*rule-tac* [*3*] *impI*)+, *rule-tac* [*3*] *FalseE*)
fix *index p q r u ns xs* **and** *key* :: *'a* ⇒ *'b*
let *?t* = *round index key p q r* (*u*, *ns*, *xs*)
assume *ns* ≠ [] ⟶ *Max* (*set ns*) = *Suc* 0 ⟶
Max (*set* (*fst* (*snd ?t*))) = *Suc* 0
moreover assume *A*: *Max* (*insert* 0 (*set ns*)) = *Suc* 0
hence *ns* ≠ []
by (*cases ns*, *simp-all*)
moreover from *this* **have** *Max* (*set ns*) = *Suc* 0
using *A* **by** *simp*

```
  ultimately show Max (set (fst (snd ?t))) = Suc 0
    by simp
next
  fix index p q r u ns xs u' ns' xs' and key :: 'a ⇒ 'b
  let ?t = round index key p q r (u, ns, tl xs)
  assume A: ?t = (u', ns', xs') and
   ns ≠ [] ⟶ Max (set ns) = Suc 0 ⟶ Max (set (fst (snd ?t))) = Suc 0
  hence B: ns ≠ [] ⟶ Max (set ns) = Suc 0 ⟶ Max (set ns') = Suc 0
    by simp
  assume C: Max (insert (Suc 0) (set ns)) = Suc 0
  show Max (insert (Suc 0) (set ns')) = Suc 0
  proof (cases ns' = [], simp)
    assume D: ns' ≠ []
    hence fst (snd ?t) ≠ []
      using A by simp
    hence ∃ n ∈ set ns. 0 < n
      by (simp add: round-nil)
    then obtain n where E: n ∈ set ns and F: 0 < n ..
    hence G: ns ≠ []
      by (cases ns, simp-all)
    moreover have n ≤ Max (set ns)
      using E by (rule-tac Max-ge, simp-all)
    hence 0 < Max (set ns)
      using F by simp
    hence Max (set ns) = Suc 0
      using C and G by simp
    ultimately have Max (set ns') = Suc 0
      using B by simp
    thus ?thesis
      using D by simp
  qed
next
  fix n ns
  assume Max (insert (Suc (Suc n)) (set ns)) = Suc 0
  thus False
    by (cases ns, simp-all)
qed

lemma round-max-less [rule-format]:
 fst (snd t) ≠ [] ⟶ Suc 0 < Max (set (fst (snd t))) ⟶
    Max (set (fst (snd (round index key p q r t)))) < Max (set (fst (snd t)))
proof (induction index key p q r t rule: round.induct, simp-all add: Let-def split:
 prod.split del: all-simps, rule impI, (rule-tac [2] allI)+, (rule-tac [2] impI)+,
 (rule-tac [3] allI)+, (rule-tac [3] impI)+, rule-tac [2] ballI)
  fix index p q r u ns xs and key :: 'a ⇒ 'b
  let ?t = round index key p q r (u, ns, xs)
  assume ns ≠ [] ⟶ Suc 0 < Max (set ns) ⟶
    Max (set (fst (snd ?t))) < Max (set ns)
  moreover assume A: Suc 0 < Max (insert 0 (set ns))
```

hence $ns \neq []$
by (*cases ns*, *simp-all*)
moreover from *this* **have** *Suc 0 < Max (set ns)*
using *A* **by** *simp*
ultimately show *Max (set (fst (snd ?t))) < Max (insert 0 (set ns))*
by *simp*
next
fix *index p q r u ns xs u' ns' xs' i* **and** *key :: 'a ⇒ 'b*
let *?t = round index key p q r (u, ns, tl xs)*
assume
?t = (u', ns', xs') **and**
ns ≠ [] ⟶ Suc 0 < Max (set ns) ⟶
Max (set (fst (snd ?t))) < Max (set ns)
hence *ns ≠ [] ⟶ Suc 0 < Max (set ns) ⟶*
Max (set ns') < Max (set ns)
by *simp*
moreover assume *A*: *Suc 0 < Max (insert (Suc 0) (set ns))*
hence *B*: *ns ≠ []*
by (*cases ns*, *simp-all*)
moreover from *this* **have** *Suc 0 < Max (set ns)*
using *A* **by** *simp*
ultimately have *Max (set ns') < Max (set ns)* **by** *simp*
moreover assume *i ∈ set ns'*
hence *i ≤ Max (set ns')*
by (*rule-tac Max-ge*, *simp*)
ultimately show *i < Max (insert (Suc 0) (set ns))*
using *B* **by** *simp*
next
fix *index p q r u n ns n' r' v ms' ws' u' ns' xs'*
and *key :: 'a ⇒ 'b* **and** *xs :: 'a list*
let *?ws = take (Suc (Suc n)) xs*
let *?ys = drop (Suc (Suc n)) xs*
let *?r = λn'. round-suc-suc index key ?ws n n' u*
let *?t = λr' v. round index key p q r' (v, ns, ?ys)*
assume
A: *?r n' = (v, ms', ws')* **and**
B: *?t r' v = (u', ns', xs')*
moreover assume $\bigwedge$*ws a b c d e f g h.*
ws = ?ws ⟹ a = bn-comp n p q r ⟹ (b, c) = bn-comp n p q r ⟹
d = ?r b ⟹ (e, f) = ?r b ⟹ (g, h) = f ⟹
ns ≠ [] ⟶ Suc 0 < Max (set ns) ⟶
Max (set (fst (snd (?t c e)))) < Max (set ns) **and**
bn-comp n p q r = (n', r')
ultimately have *C*: *ns ≠ [] ⟶ Suc 0 < Max (set ns) ⟶*
Max (set ns') < Max (set ns)
by *simp*
from *A* [*symmetric*] **show** *Max (set ms' ∪ set ns') <*
Max (insert (Suc (Suc n)) (set ns))
proof (*simp add*: *round-suc-suc-def Let-def*, *subst Max-less-iff*, *simp-all*,

rule-tac impI, *simp add*: *Let-def split*: *if-split-asm*, *rule-tac ballI*,
erule-tac UnE, *simp add*: *Let-def split*: *if-split-asm*, (*erule-tac* [*1*−*2*] *conjE*)+)
fix *i*
assume *i* = *Suc 0* ∨ *i* = *Suc 0* ∧ *0* < *n*
hence *i* = *Suc 0* **by** *blast*
hence *i* < *Suc* (*Suc n*) **by** *simp*
also have ... ≤ *Max* (*insert* (*Suc* (*Suc n*)) (*set ns*))
by (*rule Max-ge*, *simp-all*)
finally show *i* < *Max* (*insert* (*Suc* (*Suc n*)) (*set ns*)) **.**
next
fix *i*
let *?nmi* = *mini ?ws key*
let *?nma* = *maxi ?ws key*
let *?xmi* = *?ws* ! *?nmi*
let *?xma* = *?ws* ! *?nma*
let *?mi* = *key ?xmi*
let *?ma* = *key ?xma*
let *?k* = *case n of 0* ⇒ *n* | *Suc 0* ⇒ *n* | *Suc* (*Suc i*) ⇒ *u* + *n′*
let *?zs* = *nths ?ws* (− {*?nmi*, *?nma*})
let *?ms* = *enum ?zs index key ?k ?mi ?ma*
assume *i* = *Suc 0* ∨ *i* ∈ *set ?ms*
moreover {
assume *i* = *Suc 0*
hence *i* < *Suc* (*Suc n*) **by** *simp*
}
moreover {
assume *D*: *i* ∈ *set ?ms*
hence *i* ≤ *Max* (*set ?ms*)
by (*rule-tac Max-ge*, *simp*)
moreover have *0* < *length ?ms*
using *D* **by** (*rule length-pos-if-in-set*)
hence *0* < *?k*
by (*simp add*: *enum-length*)
hence *Max* (*set ?ms*) ≤ *length ?zs*
by (*rule enum-max-le*)
ultimately have *i* ≤ *length ?zs* **by** *simp*
moreover have *length ?zs* ≤ *n*
by (*rule mini-maxi-nths-le*, *simp*)
ultimately have *i* < *Suc* (*Suc n*) **by** *simp*
}
ultimately have *i* < *Suc* (*Suc n*) **..**
also have ... ≤ *Max* (*insert* (*Suc* (*Suc n*)) (*set ns*))
by (*rule Max-ge*, *simp-all*)
finally show *i* < *Max* (*insert* (*Suc* (*Suc n*)) (*set ns*)) **.**
next
fix *i*
assume *D*: *i* ∈ *set ns′*
hence *0* < *length ns′*
by (*rule length-pos-if-in-set*)

```
    hence fst (snd (?t r' v)) ≠ []
      using B by simp
    hence E: ∃ n ∈ set ns. 0 < n
      by (simp add: round-nil)
    hence F: ns ≠ []
      by (cases ns, simp-all)
    show i < Max (insert (Suc (Suc n)) (set ns))
    proof (cases Suc 0 < Max (set ns))
      case True
      hence Max (set ns') < Max (set ns)
        using C and F by simp
      moreover have i ≤ Max (set ns')
        using D by (rule-tac Max-ge, simp)
      ultimately show ?thesis
        using F by simp
    next
      case False
      moreover from E obtain j where G: j ∈ set ns and H: 0 < j ..
      have j ≤ Max (set ns)
        using G by (rule-tac Max-ge, simp)
      hence 0 < Max (set ns)
        using H by simp
      ultimately have Max (set ns) = Suc 0 by simp
      hence Max (set (fst (snd (?t r' v)))) = Suc 0
        using F by (rule-tac round-max-eq, simp-all)
      hence Max (set ns') = Suc 0
        using B by simp
      moreover have i ≤ Max (set ns')
        using D by (rule-tac Max-ge, simp)
      ultimately have i < Suc (Suc n) by simp
      also have ... ≤ Max (insert (Suc (Suc n)) (set ns))
        by (rule Max-ge, simp-all)
      finally show ?thesis .
    qed
  qed
qed

termination gcsort-aux
proof (relation measure (λ(index, key, p, t). Max (set (fst (snd t)))),
 simp-all add: find-None-iff, erule exE, erule conjE)
  fix index p ns xs i and key :: 'a ⇒ 'b
  let ?t = gcsort-round index key p ns xs
  assume A: Suc 0 < i and B: i ∈ set ns
  have C: 0 < length ns
    using B by (rule length-pos-if-in-set)
  moreover have ∃ i ∈ set ns. Suc 0 < i
    using A and B ..
  hence Suc 0 < Max (set ns)
    using C by (subst Max-gr-iff, simp-all)
```

ultimately have *Max* (*set* (*fst* (*snd ?t*))) < *Max* (*set* (*fst* (*snd* (*0*, *ns*, *xs*))))
by (*insert round-max-less* [*of* (*0*, *ns*, *xs*)], *simp*)
thus *Max* (*set* (*fst* (*snd ?t*))) < *Max* (*set ns*) **by** *simp*
qed

Now steps 2, 3, and 4 of the proof method, which are independent of the properties to be proven, can be accomplished. Particularly, function *gcsort* constitutes the complete formal definition of GCsort, as it puts the algorithm's inputs and outputs into their expected form.

Observe that the conditional expression contained in the definition of function *gcsort-aux* need not be reflected in the definition of inductive set *gcsort-set* as just one alternative gives rise to a recursive call, viz. as its only purpose is to ensure the function's termination.

definition *gcsort-in* :: *'a list* ⇒ *nat* × *nat list* × *'a list* **where**
gcsort-in xs ≡ (*0*, [*length xs*], *xs*)

definition *gcsort-out* :: *nat* × *nat list* × *'a list* ⇒ *'a list* **where**
gcsort-out ≡ *snd* ∘ *snd*

definition *gcsort* :: (*'a*, *'b*::*linorder*) *index-sign* ⇒ (*'a* ⇒ *'b*) ⇒ *nat* ⇒
'a list ⇒ *'a list* **where**
gcsort index key p xs ≡ *gcsort-out* (*gcsort-aux index key p* (*gcsort-in xs*))

inductive-set *gcsort-set* :: (*'a*, *'b*::*linorder*) *index-sign* ⇒ (*'a* ⇒ *'b*) ⇒ *nat* ⇒
nat × *nat list* × *'a list* ⇒ (*nat* × *nat list* × *'a list*) *set*
for *index key p t* **where**
R0: *t* ∈ *gcsort-set index key p t* |
R1: (*u*, *ns*, *xs*) ∈ *gcsort-set index key p t* ⟹
gcsort-round index key p ns xs ∈ *gcsort-set index key p t*

lemma *gcsort-subset*:
assumes *A*: *t'* ∈ *gcsort-set index key p t*
shows *gcsort-set index key p t'* ⊆ *gcsort-set index key p t*
by (*rule subsetI*, *erule gcsort-set.induct*, *rule A*, *rule R1*)

lemma *gcsort-aux-set*:
gcsort-aux index key p t ∈ *gcsort-set index key p t*
proof (*induction index key p t rule*: *gcsort-aux.induct*, *simp*, *rule conjI*,
rule-tac [!] *impI*, *rule R0*, *simp*)
fix *index p u ns xs* **and** *key* :: *'a* ⇒ *'b*
let *?t* = *gcsort-round index key p ns xs*
assume *gcsort-aux index key p ?t* ∈ *gcsort-set index key p ?t*
moreover have (*u*, *ns*, *xs*) ∈ *gcsort-set index key p* (*u*, *ns*, *xs*)
by (*rule R0*)
hence *?t* ∈ *gcsort-set index key p* (*u*, *ns*, *xs*)
by (*rule R1*)

hence *gcsort-set index key p ?t* ⊆ *gcsort-set index key p* (*u*, *ns*, *xs*)
by (*rule gcsort-subset*)
ultimately show *gcsort-aux index key p ?t*
∈ *gcsort-set index key p* (*u*, *ns*, *xs*) **..**
qed

1.3 Proof of a preliminary invariant

This section is dedicated to the proof of the invariance of predicate *add-inv*, defined here below, over inductive set *gcsort-set*. This invariant will later be used to prove GCsort's correctness properties.

Another predicate, *bn-inv*, is also defined, using predicate *bn-valid* defined above.

fun *bn-inv* :: *nat* ⇒ *nat* ⇒ *nat* × *nat list* × *'a list* ⇒ *bool* **where**
bn-inv p q (*u*, *ns*, *xs*) =
(∀ *n* ∈ *set ns*. *case n of Suc* (*Suc m*) ⇒ *bn-valid m p q* | - ⇒ *True*)

fun *add-inv* :: *nat* ⇒ *nat* × *nat list* × *'a list* ⇒ *bool* **where**
add-inv n (*u*, *ns*, *xs*) =
(*foldl* (+) *0 ns* = *n* ∧ *length xs* = *n*)

lemma *gcsort-add-input*:
add-inv (*length xs*) (*0*, [*length xs*], *xs*)
by *simp*

lemma *add-base*:
foldl (+) (*k* + *m*) *ns* = *foldl* (+) *m ns* + (*k* :: *nat*)
by (*induction ns arbitrary*: *m*, *simp-all*, *subst add.assoc*, *simp*)

lemma *add-base-zero*:
foldl (+) *k ns* = *foldl* (+) *0 ns* + (*k* :: *nat*)
by (*insert add-base* [*of k 0 ns*], *simp*)

lemma *bn-count-le*:
bn-count ns ≤ *foldl* (+) *0 ns*
by (*induction ns rule*: *bn-count.induct*, *simp-all add*: *add-suc*, *subst add-base-zero*,
simp)

Here below is the proof of the main property of predicate *bn-inv*, which states that if the objects' number is not larger than the counters' upper bound, then, as long as there are buckets to be split, the arguments p and q passed by function *round* to function *bn-comp* are such that $0 < q \leq p$.

lemma *bn-inv-intro* [*rule-format*]:
foldl (+) *0 ns* ≤ *p* ⟶

```
    bn-inv (p − bn-count ns) (foldl (+) 0 ns − bn-count ns) (u, ns, xs)
proof (induction ns, simp-all, (rule impI)+, subst (asm) (3) add-base-zero,
 subst (1 2) add-base-zero, simp)
  fix n ns
  assume
    A: ∀ x ∈ set ns. case x of 0 ⇒ True | Suc 0 ⇒ True | Suc (Suc m) ⇒
      bn-valid m (p − bn-count ns) (foldl (+) 0 ns − bn-count ns) and
    B: foldl (+) 0 ns + n ≤ p
  show
   (case n of 0 ⇒ True | Suc 0 ⇒ True | Suc (Suc m) ⇒
      bn-valid m (p − bn-count (n # ns))
        (foldl (+) 0 ns + n − bn-count (n # ns))) ∧
    (∀ x ∈ set ns. case x of 0 ⇒ True | Suc 0 ⇒ True | Suc (Suc m) ⇒
      bn-valid m (p − bn-count (n # ns))
        (foldl (+) 0 ns + n − bn-count (n # ns)))
  using [[simproc del: defined-all]]
  proof (rule conjI, rule-tac [2] ballI, simp-all split: nat.split, rule-tac [!] allI,
   rule-tac [!] impI)
    fix m
    assume C: n = Suc (Suc m)
    show bn-valid m (p − bn-count (Suc (Suc m) # ns))
      (Suc (Suc (foldl (+) 0 ns + m)) − bn-count (Suc (Suc m) # ns))
      (is bn-valid - ?p ?q)
    proof (rule bn-valid.cases [of (m, ?p, ?q)], simp-all, erule conjE, rule conjI)
      fix k
      have bn-count ns ≤ foldl (+) 0 ns
        by (rule bn-count-le)
      thus bn-count ns < Suc (Suc (foldl (+) 0 ns + k)) by simp
    next
      fix k
      assume m = Suc (Suc k)
      hence Suc (Suc (foldl (+) 0 ns + k)) − bn-count ns =
        foldl (+) 0 ns + n − Suc (Suc (bn-count ns))
          using C by simp
      also have ... ≤ p − Suc (Suc (bn-count ns))
        using B by simp
      finally show Suc (Suc (foldl (+) 0 ns + k)) − bn-count ns ≤
        p − Suc (Suc (bn-count ns)) .
    qed
  next
    fix n′ m
    assume n′ ∈ set ns
    with A have case n′ of 0 ⇒ True | Suc 0 ⇒ True | Suc (Suc m) ⇒
      bn-valid m (p − bn-count ns) (foldl (+) 0 ns − bn-count ns) ..
    moreover assume n′ = Suc (Suc m)
    ultimately have bn-valid m (p − bn-count ns) (foldl (+) 0 ns − bn-count ns)
      by simp
    thus bn-valid m (p − bn-count (n # ns))
      (foldl (+) 0 ns + n − bn-count (n # ns))
```

```
    (is bn-valid - ?p ?q)
   proof (rule-tac bn-valid.cases [of (m, ?p, ?q)], simp-all, (erule-tac conjE)+,
    simp)
    fix p' q'
    assume bn-count ns < foldl (+) 0 ns
    moreover assume p − bn-count (n # ns) = p'
    hence p' = p − bn-count (n # ns) ..
    moreover assume foldl (+) 0 ns + n − bn-count (n # ns) = q'
    hence q' = foldl (+) 0 ns + n − bn-count (n # ns) ..
    ultimately show 0 < q' ∧ q' ≤ p'
     using B by (rule-tac bn-count.cases [of n # ns], simp-all)
   qed
  qed
qed
```

In what follows, the invariance of predicate *add-inv* over inductive set *gc-sort-set* is then proven as lemma *gcsort-add-inv*. It holds under the conditions that the objects' number is not larger than the counters' upper bound and function *index* satisfies predicate *index-less*, and states that, if the counters' sum initially matches the objects' number, this is still true after any recursive round.

```
lemma bn-comp-fst-ge [rule-format]:
 bn-valid n p q ⟶ n ≤ fst (bn-comp n p q r)
proof (induction n p q r rule: bn-comp.induct, simp-all del: mult-Suc,
 rule impI, erule conjE)
  fix n p r and q :: nat
  assume 0 < q
  hence Suc (Suc n) = Suc (Suc n) * q div q by simp
  also assume q ≤ p
  hence Suc (Suc n) * q ≤ Suc (Suc n) * p
   by (rule mult-le-mono2)
  hence Suc (Suc n) * q div q ≤ (Suc (Suc n) * p + r) div q
   by (rule-tac div-le-mono, simp)
  finally show Suc (Suc n) ≤ (Suc (Suc n) * p + r) div q .
qed

lemma bn-comp-fst-nonzero:
 bn-valid n p q ⟹ 0 < n ⟹ 0 < fst (bn-comp n p q r)
by (drule bn-comp-fst-ge [where r = r], simp)

lemma bn-comp-snd-less:
 r < q ⟹ snd (bn-comp n p q r) < q
by (induction n p q r rule: bn-comp.induct, simp-all)

lemma add-replicate:
 foldl (+) k (replicate m n) = k + m * n
```

by (*induction m arbitrary*: *k*, *simp-all*)

lemma *fill-length*:
length (*fill xs ns index key n mi ma*) = *n*
by (*induction xs arbitrary*: *ns*, *simp-all add*: *Let-def*)

lemma *enum-add* [*rule-format*]:
assumes
A: *index-less index key* **and**
B: *0* < *n*
shows (∀ *x* ∈ *set xs*. *key x* ∈ {*mi*..*ma*}) ⟶
foldl (+) *0* (*enum xs index key n mi ma*) = *length xs*
proof (*induction xs*, *simp-all add*: *Let-def add-zeros*, *rule impI*, (*erule conjE*)+, *simp*)
fix *x xs*
assume *mi* ≤ *key x* **and** *key x* ≤ *ma*
hence *index key x n mi ma* < *n*
(**is** *?i* < -)
using *A* **and** *B* **by** (*simp add*: *index-less-def*)
hence *?i* < *length* (*enum xs index key n mi ma*)
(**is** - < *length ?ns*)
by (*simp add*: *enum-length*)
hence *foldl* (+) *0* (*?ns*[*?i* := *Suc* (*?ns* ! *?i*)]) = *Suc* (*foldl* (+) *0 ?ns*)
by (*rule add-update*)
moreover assume *foldl* (+) *0 ?ns* = *length xs*
ultimately show *foldl* (+) *0* (*?ns*[*?i* := *Suc* (*?ns* ! *?i*)]) = *Suc* (*length xs*)
by *simp*
qed

lemma *round-add-inv* [*rule-format*]:
index-less index key ⟶ *bn-inv p q t* ⟶ *add-inv n t* ⟶
add-inv n (*round index key p q r t*)
using [[*simproc del*: *defined-all*]]
proof (*induction index key p q r t arbitrary*: *n rule*: *round.induct*, *simp-all add*: *Let-def split*: *prod.split*, (*rule allI*)+, (*rule impI*)+, *erule conjE*, (*rule-tac* [*2*] *allI*)+, (*rule-tac* [*2*] *impI*)+, (*erule-tac* [*2*] *conjE*)+, *rule-tac* [*2*] *ssubst* [*OF add-base-zero*], *simp-all add*: *add-suc*)
fix *n ns ns′* **and** *xs′* :: *′a list*
assume ⋀*n′*. *foldl* (+) *0 ns* = *n′* ∧ *n* − *Suc 0* = *n′* ⟶
foldl (+) *0 ns′* = *n′* ∧ *length xs′* = *n′*
hence *foldl* (+) *0 ns* = *n* − *Suc 0* ⟶
foldl (+) *0 ns′* = *n* − *Suc 0* ∧ *length xs′* = *n* − *Suc 0*
by *simp*
moreover assume *Suc* (*foldl* (+) *0 ns*) = *n*
ultimately show *Suc* (*foldl* (+) *0 ns′*) = *n* ∧ *Suc* (*length xs′*) = *n* **by** *simp*
next
fix *index p q r u m m′ ns v ms′ ws′ ns′ n*
and *key* :: *′a* ⇒ *′b* **and** *xs* :: *′a list* **and** *xs′* :: *′a list* **and** *r′* :: *nat*
let *?ws* = *take* (*Suc* (*Suc m*)) *xs*

assume
A: *round-suc-suc index key ?ws m m′ u* = (*v*, *ms′*, *ws′*) **and**
B: *bn-comp m p q r* = (*m′*, *r′*) **and**
C: *index-less index key* **and**
D: *bn-valid m p q* **and**
E: *length xs* = *n*
assume ⋀*ws a b c d e f g h n′*.
ws = *?ws* ⟹ *a* = (*m′*, *r′*) ⟹ *b* = *m′* ∧ *c* = *r′* ⟹
d = (*v*, *ms′*, *ws′*) ⟹ *e* = *v* ∧ *f* = (*ms′*, *ws′*) ⟹ *g* = *ms′* ∧ *h* = *ws′* ⟹
foldl (+) *0 ns* = *n′* ∧ *n* − *Suc* (*Suc m*) = *n′* ⟶
foldl (+) *0 ns′* = *n′* ∧ *length xs′* = *n′*
moreover assume *Suc* (*Suc* (*foldl* (+) *m ns*)) = *n*
hence *F*: *foldl* (+) *0 ns* + *Suc* (*Suc m*) = *n*
by (*subst* (*asm*) *add-base-zero*, *simp*)
ultimately have
G: *foldl* (+) *0 ns′* = *n* − *Suc* (*Suc m*) ∧ *length xs′* = *n* − *Suc* (*Suc m*)
by *simp*
from *A* **show** *foldl* (+) *0 ns′* + *foldl* (+) *0 ms′* = *n* ∧
length ws′ + *length xs′* = *n*
proof (*subst* (*2*) *add-base-zero*, *simp add*: *round-suc-suc-def Let-def split*:
if-split-asm, (*erule-tac* [!] *conjE*)+, *simp-all*)
assume *Suc 0* # *Suc 0* # *replicate m* (*Suc 0*) = *ms′*
hence *ms′* = *Suc 0* # *Suc 0* # *replicate m* (*Suc 0*) **..**
hence *foldl* (+) *0 ms′* = *Suc* (*Suc m*)
by (*simp add*: *add-replicate*)
hence *foldl* (+) *0 ns′* + *foldl* (+) *0 ms′* = *n*
using *F* **and** *G* **by** *simp*
moreover assume *?ws* = *ws′*
hence *ws′* = *?ws* **..**
hence *length ws′* = *Suc* (*Suc m*)
using *F* **and** *E* **by** *simp*
hence *length ws′* + *length xs′* = *n*
using *F* **and** *G* **by** *simp*
ultimately show *?thesis* **..**
next
let *?nmi* = *mini ?ws key*
let *?nma* = *maxi ?ws key*
let *?xmi* = *?ws* ! *?nmi*
let *?xma* = *?ws* ! *?nma*
let *?mi* = *key ?xmi*
let *?ma* = *key ?xma*
let *?k* = *case m of 0* ⇒ *m* | *Suc 0* ⇒ *m* | *Suc* (*Suc i*) ⇒ *u* + *m′*
let *?zs* = *nths ?ws* (− {*?nmi*, *?nma*})
let *?ms* = *enum ?zs index key ?k ?mi ?ma*
assume *Suc 0* # *?ms* @ [*Suc 0*] = *ms′*
hence *ms′* = *Suc 0* # *?ms* @ [*Suc 0*] **..**
moreover assume
?xmi # *map the* (*fill ?zs* (*offs ?ms 0*) *index key m ?mi ?ma*) @ [*?xma*] = *ws′*
hence *ws′* = *?xmi* # *map the* (*fill ?zs* (*offs ?ms 0*) *index key m ?mi ?ma*)

```
    @ [?xma] ..
  ultimately show ?thesis
  proof (simp add: fill-length, subst (2) add-base-zero, simp, cases m)
    case 0
    moreover from this have length ?ms = 0
      by (simp add: enum-length)
    ultimately show Suc (Suc (foldl (+) 0 ns' + foldl (+) 0 ?ms)) = n ∧
      Suc (Suc (m + length xs')) = n
      using F and G by simp
  next
    case Suc
    moreover from this have 0 < fst (bn-comp m p q r)
      by (rule-tac bn-comp-fst-nonzero [OF D], simp)
    hence 0 < m'
      using B by simp
    ultimately have H: 0 < ?k
      by (simp split: nat.split)
    have foldl (+) 0 ?ms = length ?zs
      by (rule enum-add [OF C H], simp, rule conjI,
       ((rule mini-lb | rule maxi-ub), erule in-set-nthsD)+)
    moreover have length ?ws = Suc (Suc m)
      using F and E by simp
    hence length ?zs = m
      by (simp add: mini-maxi-nths)
    ultimately show Suc (Suc (foldl (+) 0 ns' + foldl (+) 0 ?ms)) = n ∧
      Suc (Suc (m + length xs')) = n
      using F and G by simp
  qed
 qed
qed

lemma gcsort-add-inv:
 assumes A: index-less index key
 shows ⟦t' ∈ gcsort-set index key p t; add-inv n t; n ≤ p⟧ ⟹
   add-inv n t'
by (erule gcsort-set.induct, simp, rule round-add-inv [OF A], simp-all del:
 bn-inv.simps, erule conjE, frule sym, erule subst, rule bn-inv-intro, simp)
```

1.4 Proof of counters' optimization

In this section, it is formally proven that the number of the counters (and then of the buckets as well) used in each recursive round is maximized never exceeding the fixed upper bound.

This property is formalized by theorem *round-len*, which holds under the condition that the objects' number is not larger than the counters' upper bound and states what follows:

- While there is some bucket with size larger than two, the sum of the

number of the used counters and the number of the unused ones – viz. those, if any, left unused due to the presence of some bucket with size larger than two and equal minimum and maximum keys (cf. section 1.1.5) – matches the counters' upper bound.
In addition to ensuring the upper bound's enforcement, this implies that the number of the used counters matches the upper bound unless there is some aforesaid bucket not followed by any other bucket with size larger than two and distinct minimum and maximum keys.

- Once there is no bucket with size larger than two – in which case a round is executed just in case there is some bucket with size two –, the number of the used counters matches the objects' number.
  In fact, the algorithm immediately terminates after such a round since every resulting bucket has size one, so that increasing the number of the used counters does not matter in this case.

```
lemma round-len-less [rule-format]:
 bn-inv p q t ⟶ r < q ⟶
  (r + (foldl (+) 0 (fst (snd t)) − bn-count (fst (snd t))) ∗ p) mod q = 0 ⟶
    (fst (round index key p q r t) +
      length (fst (snd (round index key p q r t)))) ∗ q =
    (fst t + bn-count (fst (snd t))) ∗ q +
      (foldl (+) 0 (fst (snd t)) − bn-count (fst (snd t))) ∗ p + r
using [[simproc del: defined-all]]
proof (induction index key p q r t rule: round.induct, simp-all add: Let-def
 split: prod.split del: all-simps, ((rule allI)+, (rule impI)+, simp add:
 add-suc)+, subst (asm) (3) add-base-zero, subst add-base-zero, erule conjE)
  fix index p q r u n ns n′ r′ v ms′ ws′ u′
    and key :: 'a ⇒ 'b and xs :: 'a list and ns′ :: nat list
  let ?ws = take (Suc (Suc n)) xs
  assume
    A: round-suc-suc index key ?ws n n′ u = (v, ms′, ws′) and
    B: bn-comp n p q r = (n′, r′) and
    C: bn-valid n p q
  have D: bn-count ns ≤ foldl (+) 0 ns
    by (rule bn-count-le)
  assume ⋀ws a b c d e f g h.
    ws = ?ws ⟹ a = (n′, r′) ⟹ b = n′ ∧ c = r′ ⟹
    d = (v, ms′, ws′) ⟹ e = v ∧ f = (ms′, ws′) ⟹ g = ms′ ∧ h = ws′ ⟹
      r′ < q ⟶ (r′ + (foldl (+) 0 ns − bn-count ns) ∗ p) mod q = 0 ⟶
        (u′ + length ns′) ∗ q =
        (v + bn-count ns) ∗ q + (foldl (+) 0 ns − bn-count ns) ∗ p + r′
  moreover assume r < q
  hence snd (bn-comp n p q r) < q
    by (rule bn-comp-snd-less)
  hence r′ < q
    using B by simp
```

moreover assume *E*: (*r* + (*Suc* (*Suc* (*foldl* (+) *0 ns* + *n*)) −
bn-count (*Suc* (*Suc n*) # *ns*)) ∗ *p*) *mod q* = *0*
from *B* [*symmetric*] **have** (*r′* + (*foldl* (+) *0 ns* − *bn-count ns*) ∗ *p*) *mod q* = *0*
proof (*rule-tac trans* [*OF - E*], *rule-tac bn-comp.cases* [*of* (*n*, *p*, *q*, *r*)],
simp-all add: *add-mult-distrib diff-mult-distrib mod-add-left-eq*,
rule-tac arg-cong2 [**where** *f* = (*mod*)], *simp-all*)
fix *n p q r*
have *bn-count ns* ∗ *p* ≤ *foldl* (+) *0 ns* ∗ *p*
using *D* **by** (*rule mult-le-mono1*)
thus *p* + (*p* + (*n* ∗ *p* + (*foldl* (+) *0 ns* ∗ *p* − *bn-count ns* ∗ *p*))) =
p + (*p* + (*foldl* (+) *0 ns* ∗ *p* + *n* ∗ *p*)) − *bn-count ns* ∗ *p*
by *arith*
qed
ultimately have (*u′* + *length ns′*) ∗ *q* =
(*v* + *bn-count ns*) ∗ *q* + (*foldl* (+) *0 ns* − *bn-count ns*) ∗ *p* + *r′*
by *simp*
with *A* [*symmetric*] **and** *B* [*symmetric*] **show**
(*u′* + (*length ms′* + *length ns′*)) ∗ *q* =
(*u* + *bn-count* (*Suc* (*Suc n*) # *ns*)) ∗ *q* +
(*Suc* (*Suc* (*foldl* (+) *0 ns* + *n*)) − *bn-count* (*Suc* (*Suc n*) # *ns*)) ∗ *p* + *r*
proof (*rule-tac bn-comp.cases* [*of* (*n*, *p*, *q*, *r*)],
simp-all add: *round-suc-suc-def Let-def enum-length split*: *if-split-asm*)
fix *m p′ q′ r′*
assume
n = *Suc* (*Suc m*) **and**
p = *p′* **and**
q = *q′* **and**
r = *r′*
moreover have *n* ≤ *fst* (*bn-comp n p q r*)
using *C* **by** (*rule bn-comp-fst-ge*)
ultimately have *Suc* (*Suc m*) ≤ (*p′* + (*p′* + *m* ∗ *p′*) + *r′*) *div q′*
(**is** - ≤ *?a div* -)
by *simp*
hence *F*: *Suc* (*Suc m*) ∗ *q′* ≤ *?a div q′* ∗ *q′*
by (*rule mult-le-mono1*)
moreover assume (*u′* + *length ns′*) ∗ *q′* =
(*u* + *?a div q′* − *Suc* (*Suc m*) + *bn-count ns*) ∗ *q′* +
(*foldl* (+) *0 ns* − *bn-count ns*) ∗ *p′* + *?a mod q′*
ultimately have (*u′* + *length ns′*) ∗ *q′* + *Suc* (*Suc m*) ∗ *q′* =
(*u* + *bn-count ns*) ∗ *q′* + (*foldl* (+) *0 ns* − *bn-count ns*) ∗ *p′* +
?a div q′ ∗ *q′* + *?a mod q′*
(**is** *?c* = *?d*)
proof (*simp add*: *add-mult-distrib diff-mult-distrib*)
have *Suc* (*Suc m*) ∗ *q′* ≤ *?a div q′* ∗ *q′* + *?a mod q′*
using *F* **by** *arith*
hence *q′* + (*q′* + *m* ∗ *q′*) ≤ *?a*
(**is** *?b* ≤ -)
by *simp*
thus

```
      ?a + (u * q' + (bn-count ns * q' +
        (foldl (+) 0 ns * p' - bn-count ns * p'))) - ?b + ?b =
       ?a + (u * q' + (bn-count ns * q' +
        (foldl (+) 0 ns * p' - bn-count ns * p')))
       by simp
    qed
    moreover have ?c = q' + (q' + (u' + (m + length ns')) * q')
      (is - = ?e)
      by (simp add: add-mult-distrib)
    moreover have bn-count ns * p' ≤ foldl (+) 0 ns * p'
      using D by (rule mult-le-mono1)
    hence ?d = (u + bn-count ns) * q' +
      ((Suc (Suc (foldl (+) 0 ns + m)) - bn-count ns) * p' + r')
      (is - = ?f)
      by (simp (no-asm-simp) add: add-mult-distrib diff-mult-distrib)
    ultimately show ?e = ?f by simp
  next
    fix m p' q' r'
    assume (u' + length ns') * q' = bn-count ns * q' +
      (foldl (+) 0 ns - bn-count ns) * p' + (p' + (p' + m * p') + r') mod q'
      (is - = - + - + ?a mod -)
    hence (u' + length ns') * q' + (u + ?a div q') * q' =
      (u + bn-count ns) * q' + (foldl (+) 0 ns - bn-count ns) * p' + ?a
      (is ?c = ?d)
      by (simp add: add-mult-distrib)
    moreover have ?c = (u' + (u + ?a div q' + length ns')) * q'
      (is - = ?e)
      by (simp add: add-mult-distrib)
    moreover have bn-count ns * p' ≤ foldl (+) 0 ns * p'
      using D by (rule mult-le-mono1)
    hence ?d = (u + bn-count ns) * q' +
      ((Suc (Suc (foldl (+) 0 ns + m)) - bn-count ns) * p' + r')
      (is - = ?f)
      by (simp (no-asm-simp) add: add-mult-distrib diff-mult-distrib)
    ultimately show ?e = ?f by simp
  qed
qed

lemma round-len-eq [rule-format]:
 bn-count (fst (snd t)) = foldl (+) 0 (fst (snd t)) ⟶
    length (fst (snd (round index key p q r t))) = foldl (+) 0 (fst (snd t))
using [[simproc del: defined-all]]
proof (induction index key p q r t rule: round.induct, simp-all add: Let-def
 split: prod.split del: all-simps, ((rule allI)+, (rule impI)+, simp add:
 add-suc)+, subst (asm) (3) add-base-zero, subst add-base-zero)
  fix index p q r u n ns n' v ms' ws'
    and key :: 'a ⇒ 'b and xs :: 'a list and ns' :: nat list and r' :: nat
  let ?ws = take (Suc (Suc n)) xs
  assume
```

A: *round-suc-suc index key ?ws n n′ u* = (*v*, *ms′*, *ws′*) **and**
B: *bn-count* (*Suc* (*Suc n*) # *ns*) = *Suc* (*Suc* (*foldl* (+) *0 ns* + *n*))
assume ⋀*ws a b c d e f g h*.
ws = *?ws* ⟹ *a* = (*n′*, *r′*) ⟹ *b* = *n′* ∧ *c* = *r′* ⟹
d = (*v*, *ms′*, *ws′*) ⟹ *e* = *v* ∧ *f* = (*ms′*, *ws′*) ⟹ *g* = *ms′* ∧ *h* = *ws′* ⟹
bn-count ns = *foldl* (+) *0 ns* ⟶ *length ns′* = *foldl* (+) *0 ns*
moreover have *C*: *n* = *0* ∨ *n* = *Suc 0*
using *B* **by** (*rule-tac bn-comp.cases* [*of* (*n*, *p*, *q*, *r*)],
insert bn-count-le [*of ns*], *simp-all*)
hence *bn-count ns* = *foldl* (+) *0 ns*
using *B* **by** (*erule-tac disjE*, *simp-all*)
ultimately have *length ns′* = *foldl* (+) *0 ns* **by** *simp*
with *A* [*symmetric*] **show** *length ms′* + *length ns′* =
Suc (*Suc* (*foldl* (+) *0 ns* + *n*))
by (*rule-tac disjE* [*OF C*], *simp-all*
add: *round-suc-suc-def Let-def enum-length split*: *if-split-asm*)
qed

theorem *round-len*:
assumes
A: *length xs* = *foldl* (+) *0 ns* **and**
B: *length xs* ≤ *p*
shows *if bn-count ns* < *length xs*
then fst (*gcsort-round index key p ns xs*) +
length (*fst* (*snd* (*gcsort-round index key p ns xs*))) = *p*
else length (*fst* (*snd* (*gcsort-round index key p ns xs*))) = *length xs*
(**is** *if - then fst ?t* + - = - *else* -)
proof (*split if-split*, *rule conjI*, *rule-tac* [!] *impI*)
assume *C*: *bn-count ns* < *length xs*
moreover have
bn-inv (*p* − *bn-count ns*) (*foldl* (+) *0 ns* − *bn-count ns*) (*0*, *ns*, *xs*)
using *A* **and** *B* **by** (*rule-tac bn-inv-intro*, *simp*)
ultimately have
(*fst ?t* + *length* (*fst* (*snd ?t*))) ∗ (*length xs* − *bn-count ns*) =
bn-count ns ∗ (*length xs* − *bn-count ns*) +
(*p* − *bn-count ns*) ∗ (*length xs* − *bn-count ns*)
(**is** *?a* ∗ *?b* = *?c*)
using *A* **by** (*subst round-len-less*, *simp-all*)
also have *bn-count ns* ≤ *p*
using *B* **and** *C* **by** *simp*
hence *bn-count ns* ∗ *?b* ≤ *p* ∗ *?b*
by (*rule mult-le-mono1*)
hence *?c* = *p* ∗ *?b*
by (*simp* (*no-asm-simp*) *add*: *diff-mult-distrib*)
finally have *?a* ∗ *?b* = *p* ∗ *?b* **.**
thus *?a* = *p*
using *C* **by** *simp*
next
assume ¬ *bn-count ns* < *length xs*

moreover have *bn-count ns* $\leq$ *foldl* (+) *0 ns*
by (*rule bn-count-le*)
ultimately show *length* (*fst* (*snd ?t*)) = *length xs*
using *A* **by** (*subst round-len-eq*, *simp-all*)
qed

end

2 Proof of objects' conservation

theory *Conservation*
imports
Algorithm
HOL−Library.Multiset
begin

In this section, it is formally proven that GCsort *conserves objects*, viz. that the objects' list returned by function *gcsort* contains as many occurrences of any given object as the input objects' list.

Here below, steps 5, 6, and 7 of the proof method are accomplished. Particularly, *count-inv* is the predicate that will be shown to be invariant over inductive set *gcsort-set*.

fun *count-inv* :: ('*a* $\Rightarrow$ *nat*) $\Rightarrow$ *nat* $\times$ *nat list* $\times$ '*a list* $\Rightarrow$ *bool* **where**
count-inv f (*u*, *ns*, *xs*) = ($\forall$ *x*. *count* (*mset xs*) *x* = *f x*)

lemma *gcsort-count-input*:
count-inv (*count* (*mset xs*)) (*0*, [*length xs*], *xs*)
by *simp*

lemma *gcsort-count-intro*:
count-inv f t $\Longrightarrow$ *count* (*mset* (*gcsort-out t*)) *x* = *f x*
by (*cases t*, *simp add*: *gcsort-out-def*)

The main task to be accomplished to prove that GCsort conserves objects is to prove that so does function *fill* in case its input offsets' list is computed via the composition of functions *offs* and *enum*, as happens within function *round*.

To achieve this result, a multi-step strategy will be adopted. The first step, addressed here below, opens with the definition of predicate *offs-pred*, satisfied by an offsets' list ns and an objects' list xs just in case each bucket delimited by ns is sufficiently large to accommodate the corresponding objects in xs. Then, lemma *offs-pred-cons* shows that this predicate, if satisfied initially, keeps being true if each object in xs is consumed as happens within

function *fill*, viz. increasing the corresponding offset in *ns* by one.

definition *offs-num* :: *nat* ⇒ *'a list* ⇒ (*'a*, *'b*) *index-sign* ⇒
(*'a* ⇒ *'b*) ⇒ *'b* ⇒ *'b* ⇒ *nat* ⇒ *nat* **where**
offs-num n xs index key mi ma i ≡
length [*x*←*xs*. *index key x n mi ma* = *i*]

abbreviation *offs-set-next* :: *nat list* ⇒ *'a list* ⇒ (*'a*, *'b*) *index-sign* ⇒
(*'a* ⇒ *'b*) ⇒ *'b* ⇒ *'b* ⇒ *nat* ⇒ *nat set* **where**
offs-set-next ns xs index key mi ma i ≡
{*k*. *k* < *length ns* ∧ *i* < *k* ∧ *0* < *offs-num* (*length ns*) *xs index key mi ma k*}

abbreviation *offs-set-prev* :: *nat list* ⇒ *'a list* ⇒ (*'a*, *'b*) *index-sign* ⇒
(*'a* ⇒ *'b*) ⇒ *'b* ⇒ *'b* ⇒ *nat* ⇒ *nat set* **where**
offs-set-prev ns xs index key mi ma i ≡
{*k*. *i* < *length ns* ∧ *k* < *i* ∧ *0* < *offs-num* (*length ns*) *xs index key mi ma k*}

definition *offs-next* :: *nat list* ⇒ *nat* ⇒ *'a list* ⇒ (*'a*, *'b*) *index-sign* ⇒
(*'a* ⇒ *'b*) ⇒ *'b* ⇒ *'b* ⇒ *nat* ⇒ *nat* **where**
offs-next ns ub xs index key mi ma i ≡
if offs-set-next ns xs index key mi ma i = {}
then ub else ns ! *Min* (*offs-set-next ns xs index key mi ma i*)

definition *offs-none* :: *nat list* ⇒ *nat* ⇒ *'a list* ⇒ (*'a*, *'b*) *index-sign* ⇒
(*'a* ⇒ *'b*) ⇒ *'b* ⇒ *'b* ⇒ *nat* ⇒ *bool* **where**
offs-none ns ub xs index key mi ma i ≡
(∃*j* < *length ns*. *0* < *offs-num* (*length ns*) *xs index key mi ma j* ∧
i ∈ {*ns* ! *j* + *offs-num* (*length ns*) *xs index key mi ma j*..<
offs-next ns ub xs index key mi ma j}) ∨
offs-num (*length ns*) *xs index key mi ma 0* = *0* ∧
i < *offs-next ns ub xs index key mi ma 0* ∨
0 < *offs-num* (*length ns*) *xs index key mi ma 0* ∧
i < *ns* ! *0*

definition *offs-pred* :: *nat list* ⇒ *nat* ⇒ *'a list* ⇒ (*'a*, *'b*) *index-sign* ⇒
(*'a* ⇒ *'b*) ⇒ *'b* ⇒ *'b* ⇒ *bool* **where**
offs-pred ns ub xs index key mi ma ≡
∀ *i* < *length ns*. *offs-num* (*length ns*) *xs index key mi ma i* ≤
offs-next ns ub xs index key mi ma i − *ns* ! *i*

lemma *offs-num-cons*:
offs-num n (*x* # *xs*) *index key mi ma i* =
(*if index key x n mi ma* = *i then Suc else id*) (*offs-num n xs index key mi ma i*)
by (*simp add*: *offs-num-def*)

lemma *offs-next-prev*:
(*0* < *offs-num* (*length ns*) *xs index key mi ma i* ∧
offs-set-next ns xs index key mi ma i ≠ {} ∧
Min (*offs-set-next ns xs index key mi ma i*) = *j*) =

```
  (0 < offs-num (length ns) xs index key mi ma j ∧
    offs-set-prev ns xs index key mi ma j ≠ {} ∧
    Max (offs-set-prev ns xs index key mi ma j) = i)
  (is ?P = ?Q)
proof (rule iffI, (erule-tac [!] conjE)+)
  let ?A = offs-set-next ns xs index key mi ma i
  let ?B = offs-set-prev ns xs index key mi ma j
  assume
    A: 0 < offs-num (length ns) xs index key mi ma i and
    B: ?A ≠ {} and
    C: Min ?A = j
  have Min ?A ∈ ?A
    using B by (rule-tac Min-in, simp)
  hence D: j ∈ ?A
    using C by simp
  hence E: i ∈ ?B
    using A by simp
  show ?Q
  proof (rule conjI, rule-tac [2] conjI)
    show 0 < offs-num (length ns) xs index key mi ma j
      using D by simp
  next
    show ?B ≠ {}
      using E by blast
  next
    from E show Max ?B = i
    proof (subst Max-eq-iff, simp, blast, simp, rule-tac allI, rule-tac impI,
     (erule-tac conjE)+, rule-tac ccontr, simp)
      fix k
      assume F: k < j and j < length ns
      hence k < length ns by simp
      moreover assume
       ¬ k ≤ i and
       0 < offs-num (length ns) xs index key mi ma k
      ultimately have k ∈ ?A by simp
      hence Min ?A ≤ k
        by (rule-tac Min-le, simp)
      thus False
        using C and F by simp
    qed
  qed
next
  let ?A = offs-set-prev ns xs index key mi ma j
  let ?B = offs-set-next ns xs index key mi ma i
  assume
    A: 0 < offs-num (length ns) xs index key mi ma j and
    B: ?A ≠ {} and
    C: Max ?A = i
  have Max ?A ∈ ?A
```

```
    using B by (rule-tac Max-in, simp)
  hence D: i ∈ ?A
    using C by simp
  hence E: j ∈ ?B
    using A by simp
  show ?P
  proof (rule conjI, rule-tac [2] conjI)
    show 0 < offs-num (length ns) xs index key mi ma i
      using D by simp
  next
    show ?B ≠ {}
      using E by blast
  next
    from E show Min ?B = j
    proof (subst Min-eq-iff, simp, blast, simp, rule-tac allI, rule-tac impI,
     (erule-tac conjE)+, rule-tac ccontr, simp)
      fix k
      assume
       j < length ns and
       ¬ j ≤ k and
       0 < offs-num (length ns) xs index key mi ma k
      hence k ∈ ?A by simp
      hence k ≤ Max ?A
        by (rule-tac Max-ge, simp)
      moreover assume i < k
      ultimately show False
        using C by simp
    qed
  qed
qed

lemma offs-next-cons-eq:
  assumes
    A: index key x (length ns) mi ma = i and
    B: i < length ns and
    C: 0 < offs-num (length ns) (x # xs) index key mi ma j
  shows
   offs-set-prev ns (x # xs) index key mi ma i = {} ∨
     Max (offs-set-prev ns (x # xs) index key mi ma i) ≠ j ⟹
    offs-next (ns[i := Suc (ns ! i)]) ub xs index key mi ma j =
      offs-next ns ub (x # xs) index key mi ma j
  (is ?P ∨ ?Q ⟹ -)
proof (simp only: disj-imp, cases j < i)
  let ?A = offs-set-prev ns (x # xs) index key mi ma i
  let ?B = offs-set-next (ns[i := Suc (ns ! i)]) xs index key mi ma j
  let ?C = offs-set-next ns (x # xs) index key mi ma j
  case True
  hence D: j ∈ ?A
    using B and C by simp
```

hence $j \leq Max\ ?A$
by (*rule-tac Max-ge*, *simp*)
moreover assume $\neg\ ?P \longrightarrow\ ?Q$
hence *?Q*
using *D* **by** *blast*
ultimately have *E*: $j < Max\ ?A$ **by** *simp*
have *F*: $Max\ ?A \in ?A$
using *D* **by** (*rule-tac Max-in*, *simp*, *blast*)
have *G*: $Max\ ?A \in ?B$
proof (*simp*, *rule conjI*, *rule-tac* [*2*] *conjI*)
show $Max\ ?A < length\ ns$
using *F* **by** *auto*
next
show $j < Max\ ?A$
using *E* **.**
next
have $0 < offs\text{-}num\ (length\ ns)\ (x \#\ xs)\ index\ key\ mi\ ma\ (Max\ ?A)$
using *F* **by** *blast*
moreover have $Max\ ?A < i$
using *F* **by** *blast*
ultimately show $0 < offs\text{-}num\ (length\ ns)\ xs\ index\ key\ mi\ ma\ (Max\ ?A)$
using *A* **by** (*simp add*: *offs-num-cons*)
qed
hence *H*: $offs\text{-}next\ (ns[i := Suc\ (ns\ !\ i)])\ ub\ xs\ index\ key\ mi\ ma\ j =$
$ns[i := Suc\ (ns\ !\ i)]\ !\ Min\ ?B$
by (*simp only*: *offs-next-def split*: *if-split*, *blast*)
have $Min\ ?B \leq Max\ ?A$
using *G* **by** (*rule-tac Min-le*, *simp*)
moreover have $Max\ ?A < i$
using *F* **by** *blast*
ultimately have *I*: $Min\ ?B < i$ **by** *simp*
hence *J*: $offs\text{-}next\ (ns[i := Suc\ (ns\ !\ i)])\ ub\ xs\ index\ key\ mi\ ma\ j =$
$ns\ !\ Min\ ?B$
using *H* **by** *simp*
have $Min\ ?B \in ?B$
using *G* **by** (*rule-tac Min-in*, *simp*, *blast*)
hence *K*: $Min\ ?B \in ?C$
using *A* **and** *I* **by** (*simp add*: *offs-num-cons*)
hence *L*: $Min\ ?C \leq Min\ ?B$
by (*rule-tac Min-le*, *simp*)
have $Min\ ?C \in ?C$
using *K* **by** (*rule-tac Min-in*, *simp*, *blast*)
moreover have $Min\ ?C < i$
using *L* **and** *I* **by** *simp*
ultimately have $Min\ ?C \in ?B$
using *A* **by** (*simp add*: *offs-num-cons*)
hence $Min\ ?B \leq Min\ ?C$
using *G* **by** (*rule-tac Min-le*, *simp*)
hence $Min\ ?B = Min\ ?C$

```
    using L by simp
  moreover have offs-next ns ub (x # xs) index key mi ma j = ns ! Min ?C
    using K by (simp only: offs-next-def split: if-split, blast)
  ultimately show ?thesis
    using J by simp
next
  let ?A = offs-set-next (ns[i := Suc (ns ! i)]) xs index key mi ma j
  let ?B = offs-set-next ns (x # xs) index key mi ma j
  case False
  hence ?A = ?B
    using A by (rule-tac set-eqI, simp add: offs-num-cons)
  thus ?thesis
  proof (simp only: offs-next-def split: if-split,
   (rule-tac conjI, blast, rule-tac impI)+)
    assume ?B ≠ {}
    hence Min ?B ∈ ?B
      by (rule-tac Min-in, simp)
    hence i < Min ?B
      using False by simp
    thus ns[i := Suc (ns ! i)] ! Min ?B = ns ! Min ?B by simp
  qed
qed

lemma offs-next-cons-neq:
  assumes
    A: index key x (length ns) mi ma = i and
    B: offs-set-prev ns (x # xs) index key mi ma i ≠ {} and
    C: Max (offs-set-prev ns (x # xs) index key mi ma i) = j
  shows offs-next (ns[i := Suc (ns ! i)]) ub xs index key mi ma j =
    (if 0 < offs-num (length ns) xs index key mi ma i
     then Suc (ns ! i)
     else offs-next ns ub (x # xs) index key mi ma i)
proof (simp, rule conjI, rule-tac [!] impI)
  let ?A = offs-set-next ns (x # xs) index key mi ma j
  assume 0 < offs-num (length ns) xs index key mi ma i
  with A have offs-set-next (ns[i := Suc (ns ! i)]) xs index key mi ma j = ?A
    by (rule-tac set-eqI, rule-tac iffI, simp-all add: offs-num-cons split:
     if-split-asm)
  moreover have 0 < offs-num (length ns) (x # xs) index key mi ma i
    using A by (simp add: offs-num-def)
  hence 0 < offs-num (length ns) (x # xs) index key mi ma j ∧ ?A ≠ {} ∧
    Min ?A = i
    using B and C by (subst offs-next-prev, simp)
  ultimately show offs-next (ns[i := Suc (ns ! i)]) ub xs index key mi ma j =
    Suc (ns ! i)
    using B by (simp only: offs-next-def, simp, subst nth-list-update-eq, blast,
     simp)
next
  let ?A = offs-set-prev ns (x # xs) index key mi ma i
```

assume *offs-num* (*length ns*) *xs index key mi ma i* = *0*
with *A* **have** *offs-set-next* (*ns*[*i* := *Suc* (*ns* ! *i*)]) *xs index key mi ma j* =
offs-set-next ns (*x* # *xs*) *index key mi ma i*
proof (*rule-tac set-eqI*, *rule-tac iffI*, *simp-all add*: *offs-num-cons split*:
if-split-asm, *rule-tac conjI*, *rule-tac notI*, *simp*, *rule-tac impI*,
(*erule-tac* [!] *conjE*)+, *rule-tac ccontr*, *simp-all*)
fix *k*
have *i* < *length ns*
using *B* **by** *blast*
moreover assume *i* ≠ *k* **and** ¬ *i* < *k*
hence *k* < *i* **by** *simp*
moreover assume *0* < *offs-num* (*length ns*) *xs index key mi ma k*
ultimately have *k* ∈ *?A*
by (*simp add*: *offs-num-cons*)
hence *k* ≤ *Max ?A*
by (*rule-tac Max-ge*, *simp*)
moreover assume *j* < *k*
ultimately show *False*
using *C* **by** *simp*
next
fix *k*
have *Max ?A* ∈ *?A*
using *B* **by** (*rule-tac Max-in*, *simp-all*)
hence *j* < *i*
using *C* **by** *simp*
moreover assume *i* < *k*
ultimately show *j* < *k* **by** *simp*
qed
with *B* **show** *offs-next* (*ns*[*i* := *Suc* (*ns* ! *i*)]) *ub xs index key mi ma j* =
offs-next ns ub (*x* # *xs*) *index key mi ma i*
proof (*simp only*: *offs-next-def split*: *if-split*, (*rule-tac conjI*, *rule-tac* [!] *impI*,
simp)+, *subst nth-list-update*, *blast*, *simp* (*no-asm-simp*))
assume *offs-set-next ns* (*x* # *xs*) *index key mi ma i* ≠ {}
hence *Min* (*offs-set-next ns* (*x* # *xs*) *index key mi ma i*)
∈ *offs-set-next ns* (*x* # *xs*) *index key mi ma i*
by (*rule-tac Min-in*, *simp*)
thus *i* ≠ *Min* (*offs-set-next ns* (*x* # *xs*) *index key mi ma i*) **by** *simp*
qed
qed

lemma *offs-pred-ub-aux* [*rule-format*]:
assumes *A*: *offs-pred ns ub xs index key mi ma*
shows *i* < *length ns* ⟹
∀ *j* < *length ns*. *i* ≤ *j* ⟶ *0* < *offs-num* (*length ns*) *xs index key mi ma j* ⟶
ns ! *j* + *offs-num* (*length ns*) *xs index key mi ma j* ≤ *ub*
proof (*erule strict-inc-induct*, *rule-tac* [!] *allI*, (*rule-tac* [!] *impI*)+,
drule le-less-Suc-eq, *simp-all*)
fix *i*
assume *B*: *length ns* = *Suc i*

hence *offs-num* (*Suc i*) *xs index key mi ma i* ≤
offs-next ns ub xs index key mi ma i − *ns* ! *i*
using *A* **by** (*simp add*: *offs-pred-def*)
moreover have *offs-next ns ub xs index key mi ma i* = *ub*
using *B* **by** (*simp add*: *offs-next-def*)
ultimately have *offs-num* (*Suc i*) *xs index key mi ma i* ≤ *ub* − *ns* ! *i* **by** *simp*
moreover assume *0* < *offs-num* (*Suc i*) *xs index key mi ma i*
ultimately show *ns* ! *i* + *offs-num* (*Suc i*) *xs index key mi ma i* ≤ *ub* **by** *simp*
next
fix *i j*
assume *j* < *length ns*
hence *offs-num* (*length ns*) *xs index key mi ma j* ≤
offs-next ns ub xs index key mi ma j − *ns* ! *j*
using *A* **by** (*simp add*: *offs-pred-def*)
moreover assume
B: ∀ *k* < *length ns*. *Suc i* ≤ *k* ⟶
0 < *offs-num* (*length ns*) *xs index key mi ma k* ⟶
ns ! *k* + *offs-num* (*length ns*) *xs index key mi ma k* ≤ *ub* **and**
C: *i* ≤ *j*
have *offs-next ns ub xs index key mi ma j* ≤ *ub*
proof (*simp only*: *offs-next-def split*: *if-split*, *rule conjI*, *simp*, *rule impI*)
let *?A* = *offs-set-next ns xs index key mi ma j*
assume *?A* ≠ {}
hence *Min ?A* ∈ *?A*
by (*rule-tac Min-in*, *simp*)
hence *ns* ! *Min ?A* + *offs-num* (*length ns*) *xs index key mi ma* (*Min ?A*) ≤ *ub*
using *B* **and** *C* **by** *simp*
thus *ns* ! *Min ?A* ≤ *ub* **by** *simp*
qed
ultimately have *offs-num* (*length ns*) *xs index key mi ma j* ≤ *ub* − *ns* ! *j*
by *simp*
moreover assume *0* < *offs-num* (*length ns*) *xs index key mi ma j*
ultimately show *ns* ! *j* + *offs-num* (*length ns*) *xs index key mi ma j* ≤ *ub*
by *simp*
qed

lemma *offs-pred-ub*:
⟦*offs-pred ns ub xs index key mi ma*; *i* < *length ns*;
0 < *offs-num* (*length ns*) *xs index key mi ma i*⟧ ⟹
ns ! *i* + *offs-num* (*length ns*) *xs index key mi ma i* ≤ *ub*
by (*drule offs-pred-ub-aux*, *assumption*+, *simp*)

lemma *offs-pred-asc-aux* [*rule-format*]:
assumes *A*: *offs-pred ns ub xs index key mi ma*
shows *i* < *length ns* ⟹
∀ *j k*. *k* < *length ns* ⟶ *i* ≤ *j* ⟶ *j* < *k* ⟶
0 < *offs-num* (*length ns*) *xs index key mi ma j* ⟶
0 < *offs-num* (*length ns*) *xs index key mi ma k* ⟶
ns ! *j* + *offs-num* (*length ns*) *xs index key mi ma j* ≤ *ns* ! *k*

proof (*erule strict-inc-induct*, *simp*, (*rule allI*)+, (*rule impI*)+, *simp*)
fix *i j k*
assume
B: $k < \mathit{length}\ \mathit{ns}$ **and**
C: $j < k$
hence $j < \mathit{length}\ \mathit{ns}$ **by** *simp*
hence $\mathit{offs\text{-}num}\ (\mathit{length}\ \mathit{ns})\ \mathit{xs}\ \mathit{index}\ \mathit{key}\ \mathit{mi}\ \mathit{ma}\ j \leq$
$\mathit{offs\text{-}next}\ \mathit{ns}\ \mathit{ub}\ \mathit{xs}\ \mathit{index}\ \mathit{key}\ \mathit{mi}\ \mathit{ma}\ j - \mathit{ns}\ !\ j$
using *A* **by** (*simp add*: *offs-pred-def*)
moreover assume
D: $\forall\, j\ k.\ k < \mathit{length}\ \mathit{ns} \longrightarrow \mathit{Suc}\ i \leq j \longrightarrow j < k \longrightarrow$
$0 < \mathit{offs\text{-}num}\ (\mathit{length}\ \mathit{ns})\ \mathit{xs}\ \mathit{index}\ \mathit{key}\ \mathit{mi}\ \mathit{ma}\ j \longrightarrow$
$0 < \mathit{offs\text{-}num}\ (\mathit{length}\ \mathit{ns})\ \mathit{xs}\ \mathit{index}\ \mathit{key}\ \mathit{mi}\ \mathit{ma}\ k \longrightarrow$
$\mathit{ns}\ !\ j + \mathit{offs\text{-}num}\ (\mathit{length}\ \mathit{ns})\ \mathit{xs}\ \mathit{index}\ \mathit{key}\ \mathit{mi}\ \mathit{ma}\ j \leq \mathit{ns}\ !\ k$ **and**
E: $i \leq j$ **and**
F: $0 < \mathit{offs\text{-}num}\ (\mathit{length}\ \mathit{ns})\ \mathit{xs}\ \mathit{index}\ \mathit{key}\ \mathit{mi}\ \mathit{ma}\ k$
have $\mathit{offs\text{-}next}\ \mathit{ns}\ \mathit{ub}\ \mathit{xs}\ \mathit{index}\ \mathit{key}\ \mathit{mi}\ \mathit{ma}\ j \leq \mathit{ns}\ !\ k$
proof (*simp only*: *offs-next-def split*: *if-split*, *rule conjI*, *rule-tac* [!] *impI*, *rule ccontr*, *simp*)
assume $\forall\, n > j.\ n < \mathit{length}\ \mathit{ns} \longrightarrow$
$\mathit{offs\text{-}num}\ (\mathit{length}\ \mathit{ns})\ \mathit{xs}\ \mathit{index}\ \mathit{key}\ \mathit{mi}\ \mathit{ma}\ n = 0$
hence $\mathit{offs\text{-}num}\ (\mathit{length}\ \mathit{ns})\ \mathit{xs}\ \mathit{index}\ \mathit{key}\ \mathit{mi}\ \mathit{ma}\ k = 0$
using *B* **and** *C* **by** *simp*
thus *False*
using *F* **by** *simp*
next
let $?A = \mathit{offs\text{-}set\text{-}next}\ \mathit{ns}\ \mathit{xs}\ \mathit{index}\ \mathit{key}\ \mathit{mi}\ \mathit{ma}\ j$
have *G*: $k \in ?A$
using *B* **and** *C* **and** *F* **by** *simp*
hence $\mathit{Min}\ ?A \leq k$
by (*rule-tac Min-le*, *simp*)
hence $\mathit{Min}\ ?A < k \vee \mathit{Min}\ ?A = k$
by (*simp add*: *le-less*)
moreover {
have $\mathit{Suc}\ i \leq \mathit{Min}\ ?A \longrightarrow \mathit{Min}\ ?A < k \longrightarrow$
$0 < \mathit{offs\text{-}num}\ (\mathit{length}\ \mathit{ns})\ \mathit{xs}\ \mathit{index}\ \mathit{key}\ \mathit{mi}\ \mathit{ma}\ (\mathit{Min}\ ?A) \longrightarrow$
$\mathit{ns}\ !\ \mathit{Min}\ ?A + \mathit{offs\text{-}num}\ (\mathit{length}\ \mathit{ns})\ \mathit{xs}\ \mathit{index}\ \mathit{key}\ \mathit{mi}\ \mathit{ma}\ (\mathit{Min}\ ?A) \leq \mathit{ns}\ !\ k$
using *B* **and** *D* **and** *F* **by** *simp*
moreover assume $\mathit{Min}\ ?A < k$
moreover have $\mathit{Min}\ ?A \in ?A$
using *G* **by** (*rule-tac Min-in*, *simp*, *blast*)
ultimately have $\mathit{ns}\ !\ \mathit{Min}\ ?A < \mathit{ns}\ !\ k$
using *E* **by** *simp*
}
moreover {
assume $\mathit{Min}\ ?A = k$
hence $\mathit{ns}\ !\ \mathit{Min}\ ?A = \mathit{ns}\ !\ k$ **by** *simp*
}
ultimately show $\mathit{ns}\ !\ \mathit{Min}\ ?A \leq \mathit{ns}\ !\ k$

```
    by (simp add: le-less, blast)
  qed
  ultimately have offs-num (length ns) xs index key mi ma j ≤ ns ! k − ns ! j
    by simp
  moreover assume 0 < offs-num (length ns) xs index key mi ma j
  ultimately show ns ! j + offs-num (length ns) xs index key mi ma j ≤ ns ! k
    by simp
qed

lemma offs-pred-asc:
 ⟦offs-pred ns ub xs index key mi ma; i < j; j < length ns;
   0 < offs-num (length ns) xs index key mi ma i;
   0 < offs-num (length ns) xs index key mi ma j⟧ ⟹
    ns ! i + offs-num (length ns) xs index key mi ma i ≤ ns ! j
by (drule offs-pred-asc-aux, erule less-trans, assumption+, rule order-refl)

lemma offs-pred-next:
  assumes
    A: offs-pred ns ub xs index key mi ma and
    B: i < length ns and
    C: 0 < offs-num (length ns) xs index key mi ma i
  shows ns ! i < offs-next ns ub xs index key mi ma i
proof (simp only: offs-next-def split: if-split, rule conjI, rule-tac [!] impI)
  have ns ! i + offs-num (length ns) xs index key mi ma i ≤ ub
    using A and B and C by (rule offs-pred-ub)
  thus ns ! i < ub
    using C by simp
next
  assume offs-set-next ns xs index key mi ma i ≠ {}
  hence Min (offs-set-next ns xs index key mi ma i)
    ∈ offs-set-next ns xs index key mi ma i
    by (rule-tac Min-in, simp)
  hence ns ! i + offs-num (length ns) xs index key mi ma i ≤
    ns ! Min (offs-set-next ns xs index key mi ma i)
    using C by (rule-tac offs-pred-asc [OF A], simp-all)
  thus ns ! i < ns ! Min (offs-set-next ns xs index key mi ma i)
    using C by simp
qed

lemma offs-pred-next-cons-less:
  assumes
    A: offs-pred ns ub (x # xs) index key mi ma and
    B: index key x (length ns) mi ma = i and
    C: offs-set-prev ns (x # xs) index key mi ma i ≠ {} and
    D: Max (offs-set-prev ns (x # xs) index key mi ma i) = j
  shows offs-next ns ub (x # xs) index key mi ma j <
    offs-next (ns[i := Suc (ns ! i)]) ub xs index key mi ma j
  (is ?M < ?N)
proof −
```

```
  have E: 0 < offs-num (length ns) (x # xs) index key mi ma i
    using B by (simp add: offs-num-cons)
  hence 0 < offs-num (length ns) (x # xs) index key mi ma j ∧
    offs-set-next ns (x # xs) index key mi ma j ≠ {} ∧
    Min (offs-set-next ns (x # xs) index key mi ma j) = i
    using C and D by (subst offs-next-prev, simp)
  hence F: ?M = ns ! i
    by (simp only: offs-next-def, simp)
  have ?N = (if 0 < offs-num (length ns) xs index key mi ma i
    then Suc (ns ! i)
    else offs-next ns ub (x # xs) index key mi ma i)
    using B and C and D by (rule offs-next-cons-neq)
  thus ?thesis
  proof (split if-split-asm)
    assume ?N = Suc (ns ! i)
    thus ?thesis
      using F by simp
  next
    assume ?N = offs-next ns ub (x # xs) index key mi ma i
    moreover have ns ! i < offs-next ns ub (x # xs) index key mi ma i
      using C by (rule-tac offs-pred-next [OF A - E], blast)
    ultimately show ?thesis
      using F by simp
  qed
qed

lemma offs-pred-next-cons:
  assumes
    A: offs-pred ns ub (x # xs) index key mi ma and
    B: index key x (length ns) mi ma = i and
    C: i < length ns and
    D: 0 < offs-num (length ns) (x # xs) index key mi ma j
  shows offs-next ns ub (x # xs) index key mi ma j ≤
    offs-next (ns[i := Suc (ns ! i)]) ub xs index key mi ma j
  (is ?M ≤ ?N)
proof −
  let ?P = offs-set-prev ns (x # xs) index key mi ma i ≠ {} ∧
    Max (offs-set-prev ns (x # xs) index key mi ma i) = j
  have ?P ∨ ¬ ?P by blast
  moreover {
    assume ?P
    hence ?M < ?N
      using B by (rule-tac offs-pred-next-cons-less [OF A], simp-all)
    hence ?thesis by simp
  }
  moreover {
    assume ¬ ?P
    hence ?N = ?M
      by (rule-tac offs-next-cons-eq [OF B C D], blast)
```

```
    hence ?thesis by simp
  }
  ultimately show ?thesis ..
qed

lemma offs-pred-cons:
  assumes
    A: offs-pred ns ub (x # xs) index key mi ma and
    B: index key x (length ns) mi ma = i and
    C: i < length ns
  shows offs-pred (ns[i := Suc (ns ! i)]) ub xs index key mi ma
  using A
proof (simp add: offs-pred-def, rule-tac allI, rule-tac impI, simp)
  let ?ns' = ns[i := Suc (ns ! i)]
  fix j
  assume
   ∀j < length ns. offs-num (length ns) (x # xs) index key mi ma j ≤
      offs-next ns ub (x # xs) index key mi ma j − ns ! j and
   j < length ns
  hence D: offs-num (length ns) (x # xs) index key mi ma j ≤
    offs-next ns ub (x # xs) index key mi ma j − ns ! j
    by simp
  from B and C show offs-num (length ns) xs index key mi ma j ≤
    offs-next ?ns' ub xs index key mi ma j − ?ns' ! j
  proof (cases j = i, case-tac [2] 0 < offs-num (length ns) xs index key mi ma j,
   simp-all)
    assume j = i
    hence offs-num (length ns) xs index key mi ma i ≤
      offs-next ns ub (x # xs) index key mi ma i − Suc (ns ! i)
      using B and D by (simp add: offs-num-cons)
    moreover have offs-next ns ub (x # xs) index key mi ma i ≤
      offs-next ?ns' ub xs index key mi ma i
      using B by (rule-tac offs-pred-next-cons [OF A - C], simp-all add:
       offs-num-cons)
    ultimately show offs-num (length ns) xs index key mi ma i ≤
      offs-next ?ns' ub xs index key mi ma i − Suc (ns ! i)
      by simp
  next
    assume j ≠ i
    hence offs-num (length ns) xs index key mi ma j ≤
      offs-next ns ub (x # xs) index key mi ma j − ns ! j
      using B and D by (simp add: offs-num-cons)
    moreover assume 0 < offs-num (length ns) xs index key mi ma j
    hence offs-next ns ub (x # xs) index key mi ma j ≤
      offs-next ?ns' ub xs index key mi ma j
      using B by (rule-tac offs-pred-next-cons [OF A - C], simp-all add:
       offs-num-cons)
    ultimately show offs-num (length ns) xs index key mi ma j ≤
      offs-next ?ns' ub xs index key mi ma j − ns ! j
```

```
    by simp
  qed
qed
```

The next step consists of proving, as done in lemma *fill-count-item* in what follows, that if certain conditions hold, particularly if offsets' list ns and objects' list xs satisfy predicate *offs-pred*, then function *fill* conserves objects if called using xs and ns as its input arguments.

This lemma is proven by induction on xs. Hence, lemma *offs-pred-cons*, proven in the previous step, is used to remove the antecedent containing predicate *offs-pred* from the induction hypothesis, which has the form of an implication.

```
lemma offs-next-zero:
  assumes
    A: i < length ns and
    B: offs-num (length ns) xs index key mi ma i = 0 and
    C: offs-set-prev ns xs index key mi ma i = {}
  shows offs-next ns ub xs index key mi ma 0 =
    offs-next ns ub xs index key mi ma i
proof −
  have offs-set-next ns xs index key mi ma 0 =
    offs-set-next ns xs index key mi ma i
  proof (rule set-eqI, rule iffI, simp-all, (erule conjE)+, rule ccontr, simp add:
   not-less)
    fix j
    assume D: 0 < offs-num (length ns) xs index key mi ma j
    assume j ≤ i
    hence j < i ∨ j = i
      by (simp add: le-less)
    moreover {
      assume j < i
      hence j ∈ offs-set-prev ns xs index key mi ma i
        using A and D by simp
      hence False
        using C by simp
    }
    moreover {
      assume j = i
      hence False
        using B and D by simp
    }
    ultimately show False ..
  qed
  thus ?thesis
    by (simp only: offs-next-def)
qed
```

lemma *offs-next-zero-cons-eq*:
assumes
A: *index key x* (*length ns*) *mi ma* = *i* **and**
B: *offs-num* (*length ns*) (*x* # *xs*) *index key mi ma 0* = *0* **and**
C: *offs-set-prev ns* (*x* # *xs*) *index key mi ma i* ≠ {}
(**is** *?A* ≠ -)
shows *offs-next* (*ns*[*i* := *Suc* (*ns* ! *i*)]) *ub xs index key mi ma 0* =
offs-next ns ub (*x* # *xs*) *index key mi ma 0*
proof −
have *D*: *Min ?A* ∈ *?A*
using *C* **by** (*rule-tac Min-in*, *simp*)
moreover have *E*: *0* < *Min ?A*
proof (*rule ccontr*, *simp*)
assume *Min ?A* = *0*
hence *offs-num* (*length ns*) (*x* # *xs*) *index key mi ma* (*Min ?A*) = *0*
using *B* **by** *simp*
moreover have *0* < *offs-num* (*length ns*) (*x* # *xs*) *index key mi ma* (*Min ?A*)
using *D* **by** *auto*
ultimately show *False* **by** *simp*
qed
ultimately have *Min ?A* ∈ *offs-set-next ns* (*x* # *xs*) *index key mi ma 0*
(**is** - ∈ *?B*)
by *auto*
moreover from *this* **have** *Min ?B* = *Min ?A*
proof (*subst Min-eq-iff*, *simp*, *blast*, *simp*, *rule-tac allI*, *rule-tac impI*,
(*erule-tac conjE*)+, *rule-tac ccontr*, *simp add*: *not-le*)
fix *j*
assume *F*: *j* < *Min ?A*
have *i* < *length ns*
using *D* **by** *simp*
moreover have *Min ?A* < *i*
using *D* **by** *auto*
hence *j* < *i*
using *F* **by** *simp*
moreover assume *0* < *offs-num* (*length ns*) (*x* # *xs*) *index key mi ma j*
ultimately have *j* ∈ *?A* **by** *simp*
hence *Min ?A* ≤ *j*
by (*rule-tac Min-le*, *simp*)
thus *False*
using *F* **by** *simp*
qed
moreover have *Min ?A* ∈ *offs-set-next* (*ns*[*i* := *Suc* (*ns* ! *i*)])
xs index key mi ma 0
(**is** - ∈ *?C*)
using *D* **and** *E* **by** (*auto*, *simp add*: *offs-num-cons A* [*symmetric*])
moreover from *this* **have** *Min ?C* = *Min ?A*
proof (*subst Min-eq-iff*, *simp*, *blast*, *simp*, *rule-tac allI*, *rule-tac impI*,
(*erule-tac conjE*)+, *rule-tac ccontr*, *simp add*: *not-le*)

```
    fix j
    assume F: j < Min ?A
    have i < length ns
      using D by simp
    moreover have Min ?A < i
      using D by auto
    hence j < i
      using F by simp
    moreover assume 0 < offs-num (length ns) xs index key mi ma j
    hence 0 < offs-num (length ns) (x # xs) index key mi ma j
      by (simp add: offs-num-cons)
    ultimately have j ∈ ?A by simp
    hence Min ?A ≤ j
      by (rule-tac Min-le, simp)
    thus False
      using F by simp
  qed
  moreover have Min ?A < i
    using D by auto
  ultimately show ?thesis
    by (simp only: offs-next-def split: if-split,
     (rule-tac conjI, blast, rule-tac impI)+, simp)
qed

lemma offs-next-zero-cons-neq:
  assumes
    A: index key x (length ns) mi ma = i and
    B: i < length ns and
    C: 0 < i and
    D: offs-set-prev ns (x # xs) index key mi ma i = {}
  shows offs-next (ns[i := Suc (ns ! i)]) ub xs index key mi ma 0 =
    (if 0 < offs-num (length ns) xs index key mi ma i
     then Suc (ns ! i)
     else offs-next ns ub (x # xs) index key mi ma i)
proof (simp, rule conjI, rule-tac [!] impI)
  let ?ns' = ns[i := Suc (ns ! i)]
  assume 0 < offs-num (length ns) xs index key mi ma i
  with A have offs-set-next ?ns' xs index key mi ma 0 =
    offs-set-next ns (x # xs) index key mi ma 0
    by (rule-tac set-eqI, rule-tac iffI, simp-all add: offs-num-cons split:
     if-split-asm)
  moreover have i ∈ offs-set-next ns (x # xs) index key mi ma 0
    using A and B and C by (simp add: offs-num-cons)
  moreover from this have
   Min (offs-set-next ns (x # xs) index key mi ma 0) = i
  proof (subst Min-eq-iff, simp, blast, simp, rule-tac allI, rule-tac impI,
   (erule-tac conjE)+, rule-tac ccontr, simp add: not-le)
    fix j
    assume j < i and 0 < offs-num (length ns) (x # xs) index key mi ma j
```

```
    hence j ∈ offs-set-prev ns (x # xs) index key mi ma i
      using B by simp
    thus False
      using D by simp
  qed
  ultimately show offs-next ?ns' ub xs index key mi ma 0 = Suc (ns ! i)
    by (simp only: offs-next-def split: if-split, rule-tac conjI, rule-tac [!] impI,
     simp-all)
next
  let ?ns' = ns[i := Suc (ns ! i)]
  assume offs-num (length ns) xs index key mi ma i = 0
  moreover have offs-set-prev ?ns' xs index key mi ma i = {}
    using D by (simp add: offs-num-cons split: if-split-asm, blast)
  ultimately have offs-next ?ns' ub xs index key mi ma 0 =
    offs-next ?ns' ub xs index key mi ma i
    using B by (rule-tac offs-next-zero, simp-all)
  moreover have offs-next ?ns' ub xs index key mi ma i =
    offs-next ns ub (x # xs) index key mi ma i
    using A and B and D by (rule-tac offs-next-cons-eq, simp-all add:
     offs-num-cons)
  ultimately show offs-next ?ns' ub xs index key mi ma 0 =
    offs-next ns ub (x # xs) index key mi ma i
    by simp
qed

lemma offs-pred-zero-cons-less:
  assumes
    A: offs-pred ns ub (x # xs) index key mi ma and
    B: index key x (length ns) mi ma = i and
    C: i < length ns and
    D: 0 < i and
    E: offs-set-prev ns (x # xs) index key mi ma i = {}
  shows offs-next ns ub (x # xs) index key mi ma 0 <
    offs-next (ns[i := Suc (ns ! i)]) ub xs index key mi ma 0
  (is ?M < ?N)
proof -
  have i ∈ offs-set-next ns (x # xs) index key mi ma 0
    using B and C and D by (simp add: offs-num-cons)
  moreover from this have
   Min (offs-set-next ns (x # xs) index key mi ma 0) = i
  proof (subst Min-eq-iff, simp, blast, simp, rule-tac allI, rule-tac impI,
   (erule-tac conjE)+, rule-tac ccontr, simp add: not-le)
    fix j
    assume j < i and 0 < offs-num (length ns) (x # xs) index key mi ma j
    hence j ∈ offs-set-prev ns (x # xs) index key mi ma i
      using C by simp
    thus False
      using E by simp
  qed
```

```
  ultimately have F: ?M = ns ! i
    by (simp only: offs-next-def split: if-split, blast)
  have ?N = (if 0 < offs-num (length ns) xs index key mi ma i
    then Suc (ns ! i)
    else offs-next ns ub (x # xs) index key mi ma i)
    using B and C and D and E by (rule offs-next-zero-cons-neq)
  thus ?thesis
  proof (split if-split-asm)
    assume ?N = Suc (ns ! i)
    thus ?thesis
      using F by simp
  next
    assume ?N = offs-next ns ub (x # xs) index key mi ma i
    moreover have ns ! i < offs-next ns ub (x # xs) index key mi ma i
      using B by (rule-tac offs-pred-next [OF A C], simp add: offs-num-cons)
    ultimately show ?thesis
      using F by simp
  qed
qed

lemma offs-pred-zero-cons:
  assumes
    A: offs-pred ns ub (x # xs) index key mi ma and
    B: index key x (length ns) mi ma = i and
    C: i < length ns and
    D: offs-num (length ns) (x # xs) index key mi ma 0 = 0
  shows offs-next ns ub (x # xs) index key mi ma 0 ≤
    offs-next (ns[i := Suc (ns ! i)]) ub xs index key mi ma 0
  (is ?M ≤ ?N)
proof (cases offs-set-prev ns (x # xs) index key mi ma i = {})
  case True
  have 0 < i
    using B and D by (rule-tac ccontr, simp add: offs-num-cons)
  hence ?M < ?N
    using True and B by (rule-tac offs-pred-zero-cons-less [OF A - C])
  thus ?thesis by simp
next
  case False
  hence ?N = ?M
    by (rule offs-next-zero-cons-eq [OF B D])
  thus ?thesis by simp
qed

lemma replicate-count:
 count (mset (replicate n x)) x = n
by (induction n, simp-all)

lemma fill-none [rule-format]:
  assumes A: index-less index key
```

shows
(∀ *x* ∈ *set xs*. *key x* ∈ {*mi*..*ma*}) ⟶
ns ≠ [] ⟶
offs-pred ns ub xs index key mi ma ⟶
offs-none ns ub xs index key mi ma i ⟶
fill xs ns index key ub mi ma ! *i* = *None*
proof (*induction xs arbitrary*: *ns*, *simp add*: *offs-none-def offs-num-def offs-next-def*, (*rule impI*)+, *simp add*: *Let-def*, (*erule conjE*)+)
fix *x xs* **and** *ns* :: *nat list*
let *?i′* = *index key x* (*length ns*) *mi ma*
let *?ns′* = *ns*[*?i′* := *Suc* (*ns* ! *?i′*)]
assume
B: *offs-pred ns ub* (*x* # *xs*) *index key mi ma* **and**
C: *offs-none ns ub* (*x* # *xs*) *index key mi ma i*
assume
D: *ns* ≠ [] **and** *mi* ≤ *key x* **and** *key x* ≤ *ma*
moreover from *this* **have**
E: *?i′* < *length ns*
using *A* **by** (*simp add*: *index-less-def*)
hence *offs-pred ?ns′ ub xs index key mi ma*
by (*rule-tac offs-pred-cons* [*OF B*], *simp*)
moreover assume ⋀*ns*. *ns* ≠ [] ⟶ *offs-pred ns ub xs index key mi ma* ⟶
offs-none ns ub xs index key mi ma i ⟶
fill xs ns index key ub mi ma ! *i* = *None*
ultimately have
F: *offs-none ?ns′ ub xs index key mi ma i* ⟶
fill xs ?ns′ index key ub mi ma ! *i* = *None*
by *simp*
show (*fill xs ?ns′ index key ub mi ma*)[*ns* ! *?i′* := *Some x*] ! *i* = *None*
proof (*insert C*, *simp add*: *offs-none-def*, *erule disjE*, *erule-tac* [*2*] *disjE*, *simp-all del*: *subst-all*
add: *offs-num-cons split*: *if-split-asm*, *erule conjE*, *rule case-split*, *drule mp*, *assumption*, *simp-all del*: *subst-all*, (*erule conjE*)+, (*erule-tac* [*2*] *conjE*)+, *erule-tac* [*3*] *conjE*, *erule-tac* [*5*] *conjE*)
fix *j*
assume
G: *?i′* = *j* **and**
H: *j* < *length ns* **and**
I: *Suc* (*ns* ! *j* + *offs-num* (*length ns*) *xs index key mi ma j*) ≤ *i* **and**
J: *i* < *offs-next ns ub* (*x* # *xs*) *index key mi ma j*
show *fill xs* (*ns*[*j* := *Suc* (*ns* ! *j*)]) *index key ub mi ma* ! *i* = *None*
proof (*cases* 0 < *offs-num* (*length ns*) *xs index key mi ma j*, *case-tac* [*2*] *offs-set-prev ns* (*x* # *xs*) *index key mi ma j* ≠ {}, *simp-all only*: *not-not not-gr0*)
have *j* < *length ns* ∧ 0 < *offs-num* (*length ns*) *xs index key mi ma j* ∧
?ns′ ! *j* + *offs-num* (*length ns*) *xs index key mi ma j* ≤ *i* ∧
i < *offs-next ?ns′ ub xs index key mi ma j* ⟶
fill xs ?ns′ index key ub mi ma ! *i* = *None*
using *F* **by** (*simp add*: *offs-none-def*, *blast*)

moreover assume *0 < offs-num (length ns) xs index key mi ma j*
moreover have *?ns' ! j + offs-num (length ns) xs index key mi ma j ≤ i*
using *G* **and** *H* **and** *I* **by** *simp*
moreover have *0 < offs-num (length ns) (x # xs) index key mi ma j*
using *G* **by** (*simp add*: *offs-num-cons*)
hence *offs-next ns ub (x # xs) index key mi ma j ≤*
offs-next ?ns' ub xs index key mi ma j
using *G* **and** *H* **by** (*rule-tac offs-pred-next-cons* [*OF B*], *simp-all*)
hence *i < offs-next ?ns' ub xs index key mi ma j*
using *J* **by** *simp*
ultimately have *fill xs ?ns' index key ub mi ma ! i = None*
using *H* **by** *blast*
thus *?thesis*
using *G* **by** *simp*
next
let *?j' = Max (offs-set-prev ns (x # xs) index key mi ma j)*
have *?j' < length ns ∧ 0 < offs-num (length ns) xs index key mi ma ?j' ∧*
?ns' ! ?j' + offs-num (length ns) xs index key mi ma ?j' ≤ i ∧
i < offs-next ?ns' ub xs index key mi ma ?j' ⟶
fill xs ?ns' index key ub mi ma ! i = None
using *F* **by** (*simp add*: *offs-none-def*, *blast*)
moreover assume *K*: *offs-set-prev ns (x # xs) index key mi ma j ≠ {}*
hence *?j' ∈ offs-set-prev ns (x # xs) index key mi ma j*
by (*rule-tac Max-in*, *simp*)
hence *L*: *?j' < length ns ∧ ?j' < j ∧*
0 < offs-num (length ns) xs index key mi ma ?j'
using *G* **by** (*auto*, *subst* (*asm*) (*2*) *offs-num-cons*, *simp*)
moreover have *ns ! ?j' + offs-num (length ns) (x # xs)*
index key mi ma ?j' ≤ ns ! j
using *G* **and** *H* **and** *L* **by** (*rule-tac offs-pred-asc* [*OF B*], *simp-all add*:
offs-num-cons)
hence *?ns' ! ?j' + offs-num (length ns) xs index key mi ma ?j' ≤ ns ! j*
using *G* **and** *H* **and** *L* **by** (*subst nth-list-update*, *simp-all add*: *offs-num-cons*)
hence *?ns' ! ?j' + offs-num (length ns) xs index key mi ma ?j' ≤ i*
using *I* **by** *simp*
moreover assume *M*: *offs-num (length ns) xs index key mi ma j = 0*
have *offs-next (ns[j := Suc (ns ! j)]) ub xs index key mi ma ?j' =*
(if 0 < offs-num (length ns) xs index key mi ma j
then Suc (ns ! j)
else offs-next ns ub (x # xs) index key mi ma j)
using *G* **and** *K* **by** (*rule-tac offs-next-cons-neq*, *simp-all*)
hence *offs-next ?ns' ub xs index key mi ma ?j' =*
offs-next ns ub (x # xs) index key mi ma j
using *G* **and** *M* **by** *simp*
hence *i < offs-next ?ns' ub xs index key mi ma ?j'*
using *J* **by** *simp*
ultimately have *fill xs ?ns' index key ub mi ma ! i = None* **by** *blast*
thus *?thesis*
using *G* **by** *simp*

```
  next
    have offs-num (length ns) xs index key mi ma 0 = 0 ∧
      i < offs-next ?ns' ub xs index key mi ma 0 ⟶
       fill xs ?ns' index key ub mi ma ! i = None
      using F by (simp add: offs-none-def)
    moreover assume
      K: offs-set-prev ns (x # xs) index key mi ma j = {} and
      L: offs-num (length ns) xs index key mi ma j = 0
    have offs-set-prev ns (x # xs) index key mi ma j =
      offs-set-prev ?ns' xs index key mi ma j
      using G by (rule-tac set-eqI, rule-tac iffI,
       simp-all add: offs-num-cons split: if-split-asm)
    hence M: offs-set-prev ?ns' xs index key mi ma j = {}
      using K by simp
    hence offs-num (length ns) xs index key mi ma 0 = 0
      using H and L by (cases j, simp-all)
    moreover have N: offs-next ?ns' ub xs index key mi ma 0 =
      offs-next ?ns' ub xs index key mi ma j
      using H and L and M by (rule-tac offs-next-zero, simp-all)
    have offs-next ?ns' ub xs index key mi ma j =
      offs-next ns ub (x # xs) index key mi ma j
      using G and H and K by (subst offs-next-cons-eq, simp-all add:
       offs-num-cons)
    hence i < offs-next ?ns' ub xs index key mi ma 0
      using J and N by simp
    ultimately have fill xs ?ns' index key ub mi ma ! i = None by blast
    thus ?thesis
      using G by simp
  qed
next
  fix j
  assume
    G: ?i' ≠ j and
    H: j < length ns and
    I: ns ! j + offs-num (length ns) xs index key mi ma j ≤ i and
    J: i < offs-next ns ub (x # xs) index key mi ma j and
    K: 0 < offs-num (length ns) xs index key mi ma j
  from G have ns ! ?i' ≠ i
  proof (rule-tac notI, cases ?i' < j, simp-all add: not-less le-less)
    assume ?i' < j
    hence ns ! ?i' + offs-num (length ns) (x # xs) index key mi ma ?i' ≤ ns ! j
      using H and K by (rule-tac offs-pred-asc [OF B], simp-all add:
       offs-num-cons)
    moreover assume ns ! ?i' = i
    ultimately have i < ns ! j by (simp add: offs-num-cons)
    thus False
      using I by simp
  next
    assume j < ?i'
```

```
    hence L: ?i' ∈ offs-set-next ns (x # xs) index key mi ma j
      (is - ∈ ?A)
      using E by (simp add: offs-num-cons)
    hence Min ?A ≤ ?i'
      by (rule-tac Min-le, simp)
    hence Min ?A < ?i' ∨ Min ?A = ?i'
      by (simp add: le-less)
    hence ns ! Min ?A ≤ ns ! ?i'
    proof (rule disjE, simp-all)
      assume Min ?A < ?i'
      moreover have Min ?A ∈ ?A
        using L by (rule-tac Min-in, simp, blast)
      hence 0 < offs-num (length ns) (x # xs) index key mi ma (Min ?A)
        by simp
      ultimately have ns ! Min ?A + offs-num (length ns) (x # xs)
        index key mi ma (Min ?A) ≤ ns ! ?i'
        using E by (rule-tac offs-pred-asc [OF B], simp-all add: offs-num-cons)
      thus ?thesis by simp
    qed
    hence offs-next ns ub (x # xs) index key mi ma j ≤ ns ! ?i'
      using L by (simp only: offs-next-def split: if-split, blast)
    moreover assume ns ! ?i' = i
    ultimately show False
      using J by simp
  qed
  thus (fill xs ?ns' index key ub mi ma)[ns ! ?i' := Some x] ! i = None
  proof simp
    have j < length ns ∧ 0 < offs-num (length ns) xs index key mi ma j ∧
      ?ns' ! j + offs-num (length ns) xs index key mi ma j ≤ i ∧
      i < offs-next ?ns' ub xs index key mi ma j ⟶
        fill xs ?ns' index key ub mi ma ! i = None
      using F by (simp add: offs-none-def, blast)
    moreover have offs-next ns ub (x # xs) index key mi ma j ≤
      offs-next ?ns' ub xs index key mi ma j
      using E and K by (rule-tac offs-pred-next-cons [OF B], simp-all add:
       offs-num-cons)
    hence i < offs-next ?ns' ub xs index key mi ma j
      using J by simp
    ultimately show fill xs ?ns' index key ub mi ma ! i = None
      using G and H and I and K by simp
  qed
next
  assume
    G: 0 < ?i' and
    H: offs-num (length ns) xs index key mi ma 0 = 0 and
    I: i < offs-next ns ub (x # xs) index key mi ma 0
  have ns ! ?i' ≠ i
  proof
    have 0 < offs-num (length ns) (x # xs) index key mi ma ?i'
```

```
    by (simp add: offs-num-cons)
  hence L: ?i' ∈ offs-set-next ns (x # xs) index key mi ma 0
    (is - ∈ ?A)
    using E and G by simp
  hence Min ?A ≤ ?i'
    by (rule-tac Min-le, simp)
  hence Min ?A < ?i' ∨ Min ?A = ?i'
    by (simp add: le-less)
  hence ns ! Min ?A ≤ ns ! ?i'
  proof (rule disjE, simp-all)
    assume Min ?A < ?i'
    moreover have Min ?A ∈ ?A
      using L by (rule-tac Min-in, simp, blast)
    hence 0 < offs-num (length ns) (x # xs) index key mi ma (Min ?A)
      by simp
    ultimately have ns ! Min ?A + offs-num (length ns) (x # xs)
      index key mi ma (Min ?A) ≤ ns ! ?i'
      using E by (rule-tac offs-pred-asc [OF B], simp-all add: offs-num-cons)
    thus ?thesis by simp
  qed
  hence offs-next ns ub (x # xs) index key mi ma 0 ≤ ns ! ?i'
    using L by (simp only: offs-next-def split: if-split, blast)
  moreover assume ns ! ?i' = i
  ultimately show False
    using I by simp
 qed
 thus (fill xs ?ns' index key ub mi ma)[ns ! ?i' := Some x] ! i = None
 proof simp
  have offs-num (length ns) xs index key mi ma 0 = 0 ∧
    i < offs-next ?ns' ub xs index key mi ma 0 ⟶
      fill xs ?ns' index key ub mi ma ! i = None
    using F by (simp add: offs-none-def)
  moreover have offs-next ns ub (x # xs) index key mi ma 0 ≤
    offs-next ?ns' ub xs index key mi ma 0
    using E and G and H by (rule-tac offs-pred-zero-cons [OF B],
     simp-all add: offs-num-cons)
  hence i < offs-next ?ns' ub xs index key mi ma 0
    using I by simp
  ultimately show fill xs ?ns' index key ub mi ma ! i = None
    using H by simp
 qed
next
 assume
  G: ?i' = 0 and
  H: i < ns ! 0
 show fill xs (ns[0 := Suc (ns ! 0)]) index key ub mi ma ! i = None
 proof (cases 0 < offs-num (length ns) xs index key mi ma 0, simp-all)
  have 0 < offs-num (length ns) xs index key mi ma 0 ∧ i < ?ns' ! 0 ⟶
    fill xs ?ns' index key ub mi ma ! i = None
```

```
    using F by (simp add: offs-none-def)
  moreover assume 0 < offs-num (length ns) xs index key mi ma 0
  moreover have i < ?ns' ! 0
    using D and G and H by simp
  ultimately show ?thesis
    using G by simp
next
  have offs-num (length ns) xs index key mi ma 0 = 0 ∧
    i < offs-next ?ns' ub xs index key mi ma 0 ⟶
      fill xs ?ns' index key ub mi ma ! i = None
    using F by (simp add: offs-none-def)
  moreover assume offs-num (length ns) xs index key mi ma 0 = 0
  moreover have I: offs-next ?ns' ub xs index key mi ma 0 =
    offs-next ns ub (x # xs) index key mi ma 0
    using D and G by (rule-tac offs-next-cons-eq, simp-all add:
     offs-num-cons)
  have ns ! 0 < offs-next ns ub (x # xs) index key mi ma 0
    using D and G by (rule-tac offs-pred-next [OF B], simp-all add:
     offs-num-cons)
  hence i < offs-next ?ns' ub xs index key mi ma 0
    using H and I by simp
  ultimately show ?thesis
    using G by simp
qed
next
  assume
    G: 0 < ?i' and
    H: 0 < offs-num (length ns) xs index key mi ma 0 and
    I: i < ns ! 0
  have ns ! ?i' ≠ i
  proof −
    have ns ! 0 + offs-num (length ns) (x # xs) index key mi ma 0 ≤ ns ! ?i'
      using H by (rule-tac offs-pred-asc [OF B G E], simp-all add:
       offs-num-cons)
    moreover have 0 < offs-num (length ns) (x # xs) index key mi ma 0
      using H by (simp add: offs-num-cons)
    ultimately show ?thesis
      using I by simp
  qed
  thus (fill xs ?ns' index key ub mi ma)[ns ! ?i' := Some x] ! i = None
  proof simp
    have 0 < offs-num (length ns) xs index key mi ma 0 ∧ i < ?ns' ! 0 ⟶
      fill xs ?ns' index key ub mi ma ! i = None
      using F by (simp add: offs-none-def)
    moreover have i < ?ns' ! 0
      using G and I by simp
    ultimately show fill xs ?ns' index key ub mi ma ! i = None
      using H by simp
  qed
```

```
  qed
qed

lemma fill-index-none [rule-format]:
  assumes
    A: index-less index key and
    B: key x ∈ {mi..ma} and
    C: ns ≠ [] and
    D: offs-pred ns ub (x # xs) index key mi ma
  shows ∀ x ∈ set xs. key x ∈ {mi..ma} ⟹
    fill xs (ns[(index key x (length ns) mi ma) :=
      Suc (ns ! index key x (length ns) mi ma)]) index key ub mi ma !
        (ns ! index key x (length ns) mi ma) = None
  (is - ⟹ fill - ?ns' - - - - - ! (- ! ?i) = -)
  using A and B and C and D
proof (rule-tac fill-none, simp-all, rule-tac offs-pred-cons,
 simp-all, simp add: index-less-def, cases 0 < ?i,
 cases offs-set-prev ns (x # xs) index key mi ma ?i = {},
 case-tac [3] 0 < offs-num (length ns) xs index key mi ma 0)
  assume
    E: 0 < ?i and
    F: offs-set-prev ns (x # xs) index key mi ma ?i = {}
  have G: ?i < length ns
    using A and B and C by (simp add: index-less-def)
  hence offs-num (length ns) (x # xs) index key mi ma 0 = 0
    using E and F by (rule-tac ccontr, simp)
  hence offs-num (length ns) xs index key mi ma 0 = 0
    by (simp add: offs-num-cons split: if-split-asm)
  moreover have offs-next ?ns' ub xs index key mi ma 0 =
   (if 0 < offs-num (length ns) xs index key mi ma ?i
    then Suc (ns ! ?i)
    else offs-next ns ub (x # xs) index key mi ma ?i)
    using E and F and G by (rule-tac offs-next-zero-cons-neq, simp-all)
  hence ns ! ?i < offs-next ?ns' ub xs index key mi ma 0
    by (simp split: if-split-asm, rule-tac offs-pred-next [OF D G], simp add:
     offs-num-cons)
  ultimately show offs-none ?ns' ub xs index key mi ma (ns ! ?i)
    by (simp add: offs-none-def)
next
  assume
    E: 0 < ?i and
    F: offs-set-prev ns (x # xs) index key mi ma ?i ≠ {}
      (is ?A ≠ -)
  have G: ?i < length ns
    using A and B and C by (simp add: index-less-def)
  have H: Max ?A ∈ ?A
    using F by (rule-tac Max-in, simp)
  hence I: Max ?A < ?i by blast
  have Max ?A < length ns
```

```
    using H by auto
  moreover have 0 < offs-num (length ns) (x # xs) index key mi ma (Max ?A)
    using H by auto
  hence 0 < offs-num (length ns) xs index key mi ma (Max ?A)
    using I by (subst (asm) offs-num-cons, split if-split-asm, simp-all)
  moreover have ns ! Max ?A + offs-num (length ns) (x # xs)
    index key mi ma (Max ?A) ≤ ns ! ?i
   using G and H by (rule-tac offs-pred-asc [OF D], simp-all add: offs-num-cons)
  hence ?ns' ! Max ?A + offs-num (length ns) xs
    index key mi ma (Max ?A) ≤ ns ! ?i
    using I by (simp add: offs-num-cons)
  moreover have offs-next ?ns' ub xs index key mi ma (Max ?A) =
   (if 0 < offs-num (length ns) xs index key mi ma ?i
    then Suc (ns ! ?i)
    else offs-next ns ub (x # xs) index key mi ma ?i)
    using F and I by (rule-tac offs-next-cons-neq, simp-all)
  hence ns ! ?i < offs-next ?ns' ub xs index key mi ma (Max ?A)
    by (simp split: if-split-asm, rule-tac offs-pred-next [OF D G], simp add:
     offs-num-cons)
  ultimately show offs-none ?ns' ub xs index key mi ma (ns ! ?i)
    by (simp add: offs-none-def, blast)
next
  assume 0 < offs-num (length ns) xs index key mi ma 0 and ¬ 0 < ?i
  moreover have ?i < length ns
    using A and B and C by (simp add: index-less-def)
  ultimately show offs-none ?ns' ub xs index key mi ma (ns ! ?i)
    by (simp add: offs-none-def)
next
  assume
    E: ¬ 0 < ?i and
    F: ¬ 0 < offs-num (length ns) xs index key mi ma 0
  have G: ?i < length ns
    using A and B and C by (simp add: index-less-def)
  have offs-num (length ns) xs index key mi ma 0 = 0
    using F by simp
  moreover have offs-next ?ns' ub xs index key mi ma ?i =
    offs-next ns ub (x # xs) index key mi ma ?i
    using E and G by (rule-tac offs-next-cons-eq, simp-all add: offs-num-cons)
  hence ns ! ?i < offs-next ?ns' ub xs index key mi ma ?i
    by (simp, rule-tac offs-pred-next [OF D G], simp add: offs-num-cons)
  hence ns ! ?i < offs-next ?ns' ub xs index key mi ma 0
    using E by simp
  ultimately show offs-none ?ns' ub xs index key mi ma (ns ! ?i)
    by (simp add: offs-none-def)
qed

lemma fill-count-item [rule-format]:
  assumes A: index-less index key
  shows
```

```
    (∀ x ∈ set xs. key x ∈ {mi..ma}) ⟶
     ns ≠ [] ⟶
     offs-pred ns ub xs index key mi ma ⟶
     length xs ≤ ub ⟶
       count (mset (map the (fill xs ns index key ub mi ma))) x =
       count (mset xs) x + (if the None = x then ub − length xs else 0)
proof (induction xs arbitrary: ns, simp add: replicate-count, (rule impI)+,
 simp add: Let-def map-update del: count-add-mset mset-map split del: if-split,
 (erule conjE)+, subst add-mset-add-single, simp only: count-single count-union)
 fix y xs and ns :: nat list
 let ?i = index key y (length ns) mi ma
 let ?ns' = ns[?i := Suc (ns ! ?i)]
 assume
   B: ∀ x ∈ set xs. mi ≤ key x ∧ key x ≤ ma and
   C: mi ≤ key y and
   D: key y ≤ ma and
   E: ns ≠ [] and
   F: offs-pred ns ub (y # xs) index key mi ma and
   G: Suc (length xs) ≤ ub
 have H: ?i < length ns
   using A and C and D and E by (simp add: index-less-def)
 assume ⋀ns. ns ≠ [] ⟶ offs-pred ns ub xs index key mi ma ⟶
   count (mset (map the (fill xs ns index key ub mi ma))) x =
   count (mset xs) x + (if the None = x then ub − length xs else 0)
 moreover have offs-pred ?ns' ub xs index key mi ma
   using F and H by (rule-tac offs-pred-cons, simp-all)
 ultimately have count (mset (map the (fill xs ?ns' index key ub mi ma))) x =
   count (mset xs) x + (if the None = x then ub − length xs else 0)
   using E by simp
 moreover have ns ! ?i + offs-num (length ns) (y # xs)
   index key mi ma ?i ≤ ub
   using F and H by (rule offs-pred-ub, simp add: offs-num-cons)
 hence ns ! ?i < ub
   by (simp add: offs-num-cons)
 ultimately show count (mset ((map the (fill xs ?ns' index key ub mi ma))
   [ns ! ?i := y])) x = count (mset xs) x + (if y = x then 1 else 0) +
   (if the None = x then ub − length (y # xs) else 0)
 proof (subst mset-update, simp add: fill-length, subst add-mset-add-single, simp
  only: count-diff count-single count-union, subst nth-map, simp add: fill-length,
  subst add.assoc, subst (3) add.commute, subst add.assoc [symmetric],
  subst add-right-cancel)
   have fill xs ?ns' index key ub mi ma ! (ns ! ?i) = None
     using B and C and D and E by (rule-tac fill-index-none [OF A - - F],
      simp-all)
   thus count (mset xs) x + (if the None = x then ub − length xs else 0) −
     (if the (fill xs ?ns' index key ub mi ma ! (ns ! ?i)) = x then 1 else 0) =
     count (mset xs) x + (if the None = x then ub − length (y # xs) else 0)
     using G by simp
 qed
```

qed

Finally, lemma *offs-enum-pred* here below proves that, if ns is the offsets' list obtained by applying the composition of functions *offs* and *enum* to objects' list xs, then predicate *offs-pred* is satisfied by ns and xs.

This result is in turn used, together with lemma *fill-count-item*, to prove lemma *fill-offs-enum-count-item*, which states that function *fill* conserves objects if its input offsets' list is computed via the composition of functions *offs* and *enum*.

lemma *enum-offs-num*:
$i < n \Longrightarrow$ *enum xs index key n mi ma* ! $i =$ *offs-num n xs index key mi ma i*
by (*induction xs*, *simp add*: *offs-num-def*, *simp add*: *Let-def offs-num-cons*,
subst nth-list-update-eq, *simp-all add*: *enum-length*)

lemma *offs-length*:
length (*offs ns i*) = *length ns*
by (*induction ns arbitrary*: *i*, *simp-all*)

lemma *offs-add* [*rule-format*]:
$i <$ *length ns* $\longrightarrow$ *offs ns k* ! $i =$ *foldl* (+) *k* (*take i ns*)
by (*induction ns arbitrary*: *i k*, *simp*, *simp add*: *nth-Cons split*: *nat.split*)

lemma *offs-mono-aux*:
$i \leq j \Longrightarrow j <$ *length ns* $\Longrightarrow$ *offs ns k* ! $i \leq$ *offs ns k* ! $(i + (j - i))$
by (*simp only*: *offs-add take-add*, *simp add*: *add-le*)

lemma *offs-mono*:
$i \leq j \Longrightarrow j <$ *length ns* $\Longrightarrow$ *offs ns k* ! $i \leq$ *offs ns k* ! j
by (*frule offs-mono-aux*, *simp-all*)

lemma *offs-update*:
$j <$ *length ns* $\Longrightarrow$
offs (*ns*[$i :=$ *Suc* (*ns* ! i)]) *k* ! $j =$ (*if* $j \leq i$ *then id else Suc*) (*offs ns k* ! j)
by (*simp add*: *offs-add not-le take-update-swap*, *rule impI*, *subst nth-take* [*symmetric*],
assumption, *subst add-update*, *simp-all*)

lemma *offs-equal-suc*:
assumes
A: *Suc i* $<$ *length ns* **and**
B: *ns* ! $i = 0$
shows *offs ns m* ! $i =$ *offs ns m* ! *Suc i*
proof −
have *offs ns m* ! $i =$ *foldl* (+) *m* (*take i ns*)
using *A* **by** (*subst offs-add*, *simp-all*)
also have ... = *foldl* (+) *m* (*take i ns* @ [*ns* ! *i*])
using *B* **by** *simp*

also have ... = *foldl* (+) *m* (*take* (*Suc i*) *ns*)
using *A* **by** (*subst take-Suc-conv-app-nth*, *simp-all*)
also have ... = *offs ns m* ! *Suc i*
using *A* **by** (*subst offs-add*, *simp-all*)
finally show *?thesis* **.**
qed

lemma *offs-equal* [*rule-format*]:
i < *j* ⟹ *j* < *length ns* ⟹
(∀ *k* ∈ {*i*..<*j*}. *ns* ! *k* = *0*) ⟶ *offs ns m* ! *i* = *offs ns m* ! *j*
proof (*erule strict-inc-induct*, *rule-tac* [!] *impI*, *simp-all*, *erule offs-equal-suc*, *simp*)
fix *i*
assume *A*: *i* < *j* **and** *j* < *length ns*
hence *Suc i* < *length ns* **by** *simp*
moreover assume ∀ *k* ∈ {*i*..<*j*}. *ns* ! *k* = *0*
hence *ns* ! *i* = *0*
using *A* **by** *simp*
ultimately have *offs ns m* ! *i* = *offs ns m* ! *Suc i*
by (*rule offs-equal-suc*)
also assume ... = *offs ns m* ! *j*
finally show *offs ns m* ! *i* = *offs ns m* ! *j* **.**
qed

lemma *offs-enum-last* [*rule-format*]:
assumes
A: *index-less index key* **and**
B: *0* < *n* **and**
C: ∀ *x* ∈ *set xs*. *key x* ∈ {*mi*..*ma*}
shows *offs* (*enum xs index key n mi ma*) *k* ! (*n* − *Suc 0*) +
offs-num n xs index key mi ma (*n* − *Suc 0*) = *length xs* + *k*
proof −
let *?ns* = *enum xs index key n mi ma*
from *B* **have** *D*: *last ?ns* = *offs-num n xs index key mi ma* (*n* − *Suc 0*)
by (*subst last-conv-nth*, *subst length-0-conv* [*symmetric*], *simp-all add*:
enum-length, *subst enum-offs-num*, *simp-all*)
have *offs ?ns k* ! (*n* − *Suc 0*) = *foldl* (+) *k* (*take* (*n* − *Suc 0*) *?ns*)
using *B* **by** (*rule-tac offs-add*, *simp add*: *enum-length*)
also have ... = *foldl* (+) *k* (*butlast ?ns*)
by (*simp add*: *butlast-conv-take enum-length*)
finally have *offs ?ns k* ! (*n* − *Suc 0*) + *offs-num n xs index key mi ma*
(*n* − *Suc 0*) = *foldl* (+) *k* (*butlast ?ns* @ [*last ?ns*])
using *D* **by** *simp*
also have ... = *foldl* (+) *k ?ns*
using *B* **by** (*subst append-butlast-last-id*, *subst length-0-conv* [*symmetric*],
simp-all add: *enum-length*)
also have ... = *foldl* (+) *0 ?ns* + *k*
by (*rule add-base-zero*)
also have ... = *length xs* + *k*
using *A* **and** *B* **and** *C* **by** (*subst enum-add*, *simp-all*)

finally show *?thesis* **.**
qed

lemma *offs-enum-ub* [*rule-format*]:
assumes
A: *index-less index key* **and**
B: $i < n$ **and**
C: $\forall x \in set\ xs.\ key\ x \in \{mi..ma\}$
shows $offs\ (enum\ xs\ index\ key\ n\ mi\ ma)\ k\ !\ i \leq length\ xs + k$
proof −
let *?ns* = *enum xs index key n mi ma*
have $offs\ ?ns\ k\ !\ i \leq offs\ ?ns\ k\ !\ (n - Suc\ 0)$
using *B* **by** (*rule-tac offs-mono*, *simp-all add*: *enum-length*)
also have $\ldots \leq offs\ ?ns\ k\ !\ (n - Suc\ 0) +$
offs-num n xs index key mi ma $(n - Suc\ 0)$
by *simp*
also have $\ldots = length\ xs + k$
using *A* **and** *B* **and** *C* **by** (*rule-tac offs-enum-last*, *simp-all*)
finally show *?thesis* **.**
qed

lemma *offs-enum-next-ge* [*rule-format*]:
assumes
A: *index-less index key* **and**
B: $i < n$
shows $\forall x \in set\ xs.\ key\ x \in \{mi..ma\} \Longrightarrow$
$offs\ (enum\ xs\ index\ key\ n\ mi\ ma)\ k\ !\ i \leq$
$offs\text{-}next\ (offs\ (enum\ xs\ index\ key\ n\ mi\ ma)\ k)\ (length\ xs + k)$
xs index key mi ma i
(**is** - $\Longrightarrow$ $offs\ ?ns$ - ! - $\leq$ -)
proof (*simp only*: *offs-next-def split*: *if-split*, *rule conjI*, *rule-tac* [!] *impI*,
rule offs-enum-ub [*OF A B*], *simp*)
assume $offs\text{-}set\text{-}next\ (offs\ ?ns\ k)\ xs\ index\ key\ mi\ ma\ i \neq \{\}$
(**is** *?A* ≠ -)
hence *C*: *Min ?A* ∈ *?A*
by (*rule-tac Min-in*, *simp*)
hence $i \leq Min\ ?A$ **by** *simp*
moreover have $Min\ ?A < length\ ?ns$
using *C* **by** (*simp add*: *offs-length*)
ultimately show $offs\ ?ns\ k\ !\ i \leq offs\ ?ns\ k\ !\ Min\ ?A$
by (*rule offs-mono*)
qed

lemma *offs-enum-zero-aux* [*rule-format*]:
⟦*index-less index key*; $0 < n$; $\forall x \in set\ xs.\ key\ x \in \{mi..ma\}$;
$offs\text{-}num\ n\ xs\ index\ key\ mi\ ma\ (n - Suc\ 0) = 0$⟧ ⟹
$offs\ (enum\ xs\ index\ key\ n\ mi\ ma)\ k\ !\ (n - Suc\ 0) = length\ xs + k$
by (*subst offs-enum-last* [**where** *key* = *key* **and** *mi* = *mi* **and** *ma* = *ma*,
symmetric], *simp*+)

lemma *offs-enum-zero* [*rule-format*]:
assumes
A: *index-less index key* **and**
B: $i < n$ **and**
C: $\forall x \in set\ xs.\ key\ x \in \{mi..ma\}$ **and**
D: *offs-num n xs index key mi ma i* = *0*
shows *offs* (*enum xs index key n mi ma*) *k* ! *i* =
offs-next (*offs* (*enum xs index key n mi ma*) *k*) (*length xs* + *k*)
xs index key mi ma i
proof (*simp only*: *offs-next-def split*: *if-split*, *rule conjI*, *rule-tac* [!] *impI*,
cases i = *n* − *Suc 0*, *simp*)
assume *i* = *n* − *Suc 0*
thus *offs* (*enum xs index key n mi ma*) *k* ! (*n* − *Suc 0*) = *length xs* + *k*
using *A* **and** *B* **and** *C* **and** *D* **by** (*rule-tac offs-enum-zero-aux*, *simp-all*)
next
let *?ns* = *enum xs index key n mi ma*
assume *E*: *offs-set-next* (*offs ?ns k*) *xs index key mi ma i* = {}
(**is** *?A* = -)
assume *i* ≠ *n* − *Suc 0*
hence *F*: *i* < *n* − *Suc 0*
using *B* **by** *simp*
hence *offs ?ns k* ! *i* = *offs ?ns k* ! (*n* − *Suc 0*)
proof (*rule offs-equal*, *simp-all add*: *enum-length le-less*,
erule-tac conjE, *erule-tac disjE*, *rule-tac ccontr*, *drule-tac* [*2*] *sym*, *simp-all*)
fix *j*
assume *G*: *j* < *n* − *Suc 0*
hence *j* < *length* (*offs ?ns k*)
by (*simp add*: *offs-length enum-length*)
moreover assume *i* < *j*
moreover assume *0* < *?ns* ! *j*
hence *0* < *offs-num* (*length* (*offs ?ns k*)) *xs index key mi ma j*
using *G* **by** (*subst* (*asm*) *enum-offs-num*, *simp-all add*:
offs-length enum-length)
ultimately have *j* ∈ *?A* **by** *simp*
thus *False*
using *E* **by** *simp*
next
show *?ns* ! *i* = *0*
using *B* **and** *D* **by** (*subst enum-offs-num*, *simp-all*)
qed
also from *A* **and** *B* **and** *C* **have** ... = *length xs* + *k*
proof (*rule-tac offs-enum-zero-aux*, *simp-all*, *rule-tac ccontr*, *simp*)
have *n* − *Suc 0* < *length* (*offs ?ns k*)
using *B* **by** (*simp add*: *offs-length enum-length*)
moreover assume *0* < *offs-num n xs index key mi ma* (*n* − *Suc 0*)
hence *0* < *offs-num* (*length* (*offs ?ns k*)) *xs index key mi ma* (*n* − *Suc 0*)
by (*simp add*: *offs-length enum-length*)
ultimately have *n* − *Suc 0* ∈ *?A*

```
    using F by simp
  thus False
    using E by simp
 qed
 finally show offs (enum xs index key n mi ma) k ! i = length xs + k .
next
 let ?ns = enum xs index key n mi ma
 assume offs-set-next (offs ?ns k) xs index key mi ma i ≠ {}
  (is ?A ≠ -)
 hence Min ?A ∈ ?A
  by (rule-tac Min-in, simp)
 thus offs ?ns k ! i = offs ?ns k ! Min ?A
 proof (rule-tac offs-equal, simp-all add: le-less, simp add: offs-length,
  (erule-tac conjE)+, erule-tac disjE, rule-tac ccontr, drule-tac [2] sym, simp-all)
  fix j
  assume E: j < Min ?A and Min ?A < length (offs ?ns k)
  hence F: j < length (offs ?ns k) by simp
  moreover assume i < j
  moreover assume 0 < ?ns ! j
  hence 0 < offs-num (length (offs ?ns k)) xs index key mi ma j
   using F by (subst (asm) enum-offs-num, simp-all add:
    offs-length enum-length)
  ultimately have j ∈ ?A by simp
  hence Min ?A ≤ j
   by (rule-tac Min-le, simp)
  thus False
   using E by simp
 next
  show ?ns ! i = 0
   using B and D by (subst enum-offs-num, simp-all)
 qed
qed

lemma offs-enum-next-cons [rule-format]:
 assumes
  A: index-less index key and
  B: ∀ x ∈ set xs. key x ∈ {mi..ma}
 shows (if i < index key x n mi ma then (≤) else (<))
  (offs-next (offs (enum xs index key n mi ma) k)
   (length xs + k) xs index key mi ma i)
  (offs-next (offs ((enum xs index key n mi ma) [index key x n mi ma :=
   Suc (enum xs index key n mi ma ! index key x n mi ma)]) k)
   (Suc (length xs + k)) (x # xs) index key mi ma i)
 (is (if i < ?i' then - else -)
  (offs-next (offs ?ns -) - - - - - - -)
  (offs-next (offs ?ns' -) - - - - - - -))
proof (simp-all only: offs-next-def not-less split: if-split, (rule conjI, rule impI)+,
 simp, simp, (rule-tac [!] impI, (rule-tac [!] conjI)?)+, rule-tac [2-3] FalseE,
 rule-tac [4] conjI, rule-tac [4-5] impI)
```

assume
C: *offs-set-next* (*offs ?ns k*) *xs index key mi ma i* = {}
(**is** *?A* = -) **and**
D: *offs-set-next* (*offs ?ns′ k*) (*x # xs*) *index key mi ma i* ≠ {}
(**is** *?A′* ≠ -) **and**
E: *i* < *?i′*
from *C* **have** *F*: ∀ *j* ≠ *?i′*. *j* ∉ *?A′*
by (*rule-tac allI*, *rule-tac impI*, *rule-tac notI*, *simp add*: *enum-length*
offs-length offs-num-cons split: *if-split-asm*, (*erule-tac conjE*)+, *simp*)
from *D* **have** *Min ?A′* ∈ *?A′*
by (*rule-tac Min-in*, *simp*)
hence *G*: *Min ?A′* < *n*
by (*simp add*: *offs-length enum-length*)
have *H*: *Min ?A′* = *?i′*
proof (*rule Min-eqI*, *simp*, *rule eq-refl*, *erule contrapos-pp*, *insert F*, *simp*)
have ∃ *j*. *j* ∈ *?A′*
using *D* **by** *blast*
then obtain *j* **where** *j* ∈ *?A′* **..**
moreover from *this* **have** *j* = *?i′*
by (*erule-tac contrapos-pp*, *insert F*, *simp*)
ultimately show *?i′* ∈ *?A′* **by** *simp*
qed
with *G* **have** *offs ?ns′ k* ! *Min ?A′* = *offs ?ns k* ! *Min ?A′*
by (*subst offs-update*, *simp-all add*: *enum-length*)
also from *A* **and** *B* **and** *G* **and** *H* **have**
... = *offs-next* (*offs ?ns k*) (*length xs* + *k*) *xs index key mi ma* (*Min ?A′*)
proof (*rule-tac offs-enum-zero*, *simp-all*, *rule-tac ccontr*, *simp*)
assume *?i′* < *n* **and** *0* < *offs-num n xs index key mi ma ?i′*
hence *?i′* ∈ *?A*
using *E* **by** (*simp add*: *offs-length enum-length*)
thus *False*
using *C* **by** *simp*
qed
also have ... = *length xs* + *k*
proof (*simp only*: *offs-next-def split*: *if-split*, *rule conjI*, *simp*, *rule impI*,
rule FalseE, *simp*, *erule exE*, (*erule conjE*)+)
fix *j*
assume *j* < *length* (*offs ?ns k*)
moreover assume *Min ?A′* < *j*
hence *i* < *j*
using *E* **and** *H* **by** *simp*
moreover assume *0* < *offs-num* (*length* (*offs ?ns k*)) *xs index key mi ma j*
ultimately have *j* ∈ *?A* **by** *simp*
thus *False*
using *C* **by** *simp*
qed
finally show *length xs* + *k* ≤ *offs ?ns′ k* ! *Min ?A′* **by** *simp*
next
assume

```
    C: offs-set-next (offs ?ns k) xs index key mi ma i = {}
      (is ?A = -) and
    D: offs-set-next (offs ?ns' k) (x # xs) index key mi ma i ≠ {}
      (is ?A' ≠ -) and
    E: ?i' ≤ i
  have ∃j. j ∈ ?A'
    using D by blast
  then obtain j where F: j ∈ ?A' ..
  hence j < length (offs ?ns k)
    by (simp add: offs-length)
  moreover have i < j
    using F by simp
  moreover from this have
   0 < offs-num (length (offs ?ns k)) xs index key mi ma j
    using E and F by (simp add: offs-length enum-length offs-num-cons)
  ultimately have j ∈ ?A by simp
  thus False
    using C by simp
next
  assume
    C: offs-set-next (offs ?ns k) xs index key mi ma i ≠ {}
      (is ?A ≠ -) and
    D: offs-set-next (offs ?ns' k) (x # xs) index key mi ma i = {}
      (is ?A' = -)
  have ∃j. j ∈ ?A
    using C by blast
  then obtain j where E: j ∈ ?A ..
  hence j < length (offs ?ns' k)
    by (simp add: offs-length)
  moreover have i < j
    using E by simp
  moreover have 0 < offs-num (length (offs ?ns' k)) (x # xs) index key mi ma j
    using E by (simp add: offs-length enum-length offs-num-cons)
  ultimately have j ∈ ?A' by simp
  thus False
    using D by simp
next
  assume offs-set-next (offs ?ns k) xs index key mi ma i ≠ {}
    (is ?A ≠ -)
  hence Min ?A ∈ ?A
    by (rule-tac Min-in, simp)
  hence C: Min ?A < n
    by (simp add: offs-length enum-length)
  assume offs-set-next (offs ?ns' k) (x # xs) index key mi ma i ≠ {}
    (is ?A' ≠ -)
  hence D: Min ?A' ∈ ?A'
    by (rule-tac Min-in, simp)
  hence E: Min ?A' < n
    by (simp add: offs-length enum-length)
```

have *offs ?ns k ! Min ?A ≤ offs ?ns k ! Min ?A′*
proof (*cases offs-num n xs index key mi ma* (*Min ?A′*) = *0*)
case *True*
have *offs ?ns k ! Min ?A′ =*
offs-next (*offs ?ns k*) (*length xs* + *k*) *xs index key mi ma* (*Min ?A′*)
using *A* **and** *B* **and** *E* **and** *True* **by** (*rule-tac offs-enum-zero*, *simp-all*)
also from *A* **and** *B* **and** *C* **have** *... ≥ offs ?ns k ! Min ?A*
proof (*simp only*: *offs-next-def split*: *if-split*, *rule-tac conjI*, *rule-tac* [!] *impI*,
rule-tac offs-enum-ub, *simp*, *simp*, *simp*)
assume *offs-set-next* (*offs ?ns k*) *xs index key mi ma* (*Min ?A′*) ≠ {}
(**is** *?B* ≠ -)
hence *Min ?B* ∈ *?B*
by (*rule-tac Min-in*, *simp*)
hence *Min ?B* ∈ *?A*
using *D* **by** *simp*
moreover from *this* **have** *Min ?A ≤ Min ?B*
by (*rule-tac Min-le*, *simp*)
ultimately show *offs ?ns k ! Min ?A ≤ offs ?ns k ! Min ?B*
by (*rule-tac offs-mono*, *simp-all add*: *offs-length*)
qed
finally show *?thesis* **.**
next
case *False*
hence *Min ?A′* ∈ *?A*
using *D* **by** (*simp add*: *offs-length enum-length*)
hence *Min ?A ≤ Min ?A′*
by (*rule-tac Min-le*, *simp*)
thus *?thesis*
by (*rule offs-mono*, *simp-all add*: *enum-length E*)
qed
also have *... ≤ offs ?ns′ k ! Min ?A′*
using *E* **by** (*subst offs-update*, *simp-all add*: *enum-length*)
finally show *offs ?ns k ! Min ?A ≤ offs ?ns′ k ! Min ?A′* **.**
next
let *?A* = *offs-set-next* (*offs ?ns k*) *xs index key mi ma i*
assume *offs-set-next* (*offs ?ns′ k*) (*x # xs*) *index key mi ma i* ≠ {}
(**is** *?A′* ≠ -)
hence *C*: *Min ?A′* ∈ *?A′*
by (*rule-tac Min-in*, *simp*)
hence *D*: *Min ?A′* < *n*
by (*simp add*: *offs-length enum-length*)
assume *?i′* ≤ *i*
hence *E*: *?i′* < *Min ?A′*
using *C* **by** *simp*
hence *0* < *offs-num n xs index key mi ma* (*Min ?A′*)
using *C* **by** (*simp add*: *offs-length enum-length offs-num-cons*)
hence *Min ?A′* ∈ *?A*
using *C* **by** (*simp add*: *offs-length enum-length*)
hence *Min ?A ≤ Min ?A′*

```
    by (rule-tac Min-le, simp)
  hence offs ?ns k ! Min ?A ≤ offs ?ns k ! Min ?A'
    by (rule offs-mono, simp-all add: enum-length D)
  also have ... < offs ?ns' k ! Min ?A'
    using E by (subst offs-update, simp-all add: enum-length D)
  finally show offs ?ns k ! Min ?A < offs ?ns' k ! Min ?A' .
qed

lemma offs-enum-pred [rule-format]:
  assumes A: index-less index key
  shows (∀ x ∈ set xs. key x ∈ {mi..ma}) ⟶
    offs-pred (offs (enum xs index key n mi ma) k) (length xs + k)
      xs index key mi ma
proof (induction xs, simp add: offs-pred-def offs-num-def,
 simp add: Let-def offs-pred-def offs-length enum-length, rule impI, (erule conjE)+,
 simp, rule allI, rule impI, erule allE, drule mp, assumption)
  fix x xs i
  let ?i' = index key x n mi ma
  let ?ns = enum xs index key n mi ma
  let ?ns' = ?ns[?i' := Suc (?ns ! ?i')]
  assume
    B: ∀ x ∈ set xs. mi ≤ key x ∧ key x ≤ ma and
    C: i < n and
    D: offs-num n xs index key mi ma i ≤
      offs-next (offs ?ns k) (length xs + k) xs index key mi ma i − offs ?ns k ! i
  have E: (if i < ?i' then (≤) else (<))
    (offs-next (offs ?ns k) (length xs + k) xs index key mi ma i)
    (offs-next (offs ?ns' k) (Suc (length xs + k)) (x # xs) index key mi ma i)
    using A and B by (subst offs-enum-next-cons, simp-all)
  show offs-num n (x # xs) index key mi ma i ≤
    offs-next (offs ?ns' k) (Suc (length xs + k)) (x # xs) index key mi ma i −
      offs ?ns' k ! i
  proof (subst offs-update, simp add: enum-length C, rule linorder-cases [of i ?i'],
   simp-all add: offs-num-cons)
    assume i < ?i'
    hence offs-next (offs ?ns k) (length xs + k) xs index key mi ma i ≤
      offs-next (offs ?ns' k) (Suc (length xs + k)) (x # xs) index key mi ma i
      using E by simp
    thus offs-num n xs index key mi ma i ≤
      offs-next (offs ?ns' k) (Suc (length xs + k)) (x # xs) index key mi ma i −
        offs ?ns k ! i
      using D by arith
  next
    assume F: i = ?i'
    hence Suc (offs-num n xs index key mi ma ?i') ≤
      Suc (offs-next (offs ?ns k) (length xs + k) xs index key mi ma ?i' −
        offs ?ns k ! ?i')
      using D by simp
    also from A and B and C and F have ... =
```

```
      Suc (offs-next (offs ?ns k) (length xs + k) xs index key mi ma ?i') −
        offs ?ns k ! ?i'
      by (rule-tac Suc-diff-le [symmetric], rule-tac offs-enum-next-ge, simp-all)
    finally have Suc (offs-num n xs index key mi ma ?i') ≤
      Suc (offs-next (offs ?ns k) (length xs + k) xs index key mi ma ?i') −
        offs ?ns k ! ?i' .
    moreover have
     Suc (offs-next (offs ?ns k) (length xs + k) xs index key mi ma ?i') ≤
      offs-next (offs ?ns' k) (Suc (length xs + k)) (x # xs) index key mi ma ?i'
      using E and F by simp
    ultimately show Suc (offs-num n xs index key mi ma ?i') ≤
      offs-next (offs ?ns' k) (Suc (length xs + k)) (x # xs) index key mi ma ?i' −
        offs ?ns k ! ?i'
      by arith
  next
    assume ?i' < i
    hence Suc (offs-next (offs ?ns k) (length xs + k) xs index key mi ma i) ≤
      offs-next (offs ?ns' k) (Suc (length xs + k)) (x # xs) index key mi ma i
      using E by simp
    thus offs-num n xs index key mi ma i ≤
      offs-next (offs ?ns' k) (Suc (length xs + k)) (x # xs) index key mi ma i −
        Suc (offs ?ns k ! i)
      using D by arith
  qed
qed

lemma fill-offs-enum-count-item [rule-format]:
 ⟦index-less index key; ∀ x ∈ set xs. key x ∈ {mi..ma}; 0 < n⟧ ⟹
    count (mset (map the (fill xs (offs (enum xs index key n mi ma) 0)
      index key (length xs) mi ma))) x =
    count (mset xs) x
by (subst fill-count-item, simp-all, simp only: length-greater-0-conv [symmetric]
 offs-length enum-length, insert offs-enum-pred [of index key xs mi ma n 0], simp)
```

Using lemma *fill-offs-enum-count-item*, step 9 of the proof method can now be dealt with. It is accomplished by proving lemma *gcsort-count-inv*, which states that the number of the occurrences of whatever object in the objects' list is still the same after any recursive round.

```
lemma nths-count:
 count (mset (nths xs A)) x =
    count (mset xs) x − card {i. i < length xs ∧ i ∉ A ∧ xs ! i = x}
proof (induction xs arbitrary: A, simp-all add: nths-Cons)
  fix v xs A
  let ?B = {i. i < length xs ∧ Suc i ∉ A ∧ xs ! i = x}
  let ?C = λv. {i. i < Suc (length xs) ∧ i ∉ A ∧ (v # xs) ! i = x}
  have A: ⋀v. ?C v = ?C v ∩ {0} ∪ ?C v ∩ {i. ∃j. i = Suc j}
    by (subst Int-Un-distrib [symmetric], auto, arith)
```

```
  have ⋀v. card (?C v) = card (?C v ∩ {0}) + card (?C v ∩ {i. ∃j. i = Suc j})
    by (subst A, rule card-Un-disjoint, simp-all, blast)
  moreover have ⋀v. card ?B = card (?C v ∩ {i. ∃j. i = Suc j})
    by (rule bij-betw-same-card [of Suc], auto)
  ultimately show
   (0 ∈ A ⟶
     (v = x ⟶ Suc (count (mset xs) x − card ?B) =
       Suc (count (mset xs) x) − card (?C x)) ∧
     (v ≠ x ⟶ count (mset xs) x − card ?B =
       count (mset xs) x − card (?C v))) ∧
    (0 ∉ A ⟶
     (v = x ⟶ count (mset xs) x − card ?B =
       Suc (count (mset xs) x) − card (?C x)) ∧
     (v ≠ x ⟶ count (mset xs) x − card ?B =
       count (mset xs) x − card (?C v)))
  proof ((rule-tac [!] conjI, rule-tac [!] impI)+, simp-all)
    have card ?B ≤ count (mset xs) x
      by (simp add: count-mset length-filter-conv-card, rule card-mono,
       simp, blast)
    thus Suc (count (mset xs) x − card ?B) = Suc (count (mset xs) x) − card ?B
      by (rule Suc-diff-le [symmetric])
  qed
qed

lemma round-count-inv [rule-format]:
 index-less index key ⟶ bn-inv p q t ⟶ add-inv n t ⟶ count-inv f t ⟶
    count-inv f (round index key p q r t)
proof (induction index key p q r t arbitrary: n f rule: round.induct,
 (rule-tac [!] impI)+, simp, simp, simp-all only: simp-thms)
  fix index p q r u ns xs n f and key :: 'a ⇒ 'b
  let ?t = round index key p q r (u, ns, tl xs)
  assume
   ⋀n f. bn-inv p q (u, ns, tl xs) ⟶ add-inv n (u, ns, tl xs) ⟶
      count-inv f (u, ns, tl xs) ⟶ count-inv f ?t and
   bn-inv p q (u, Suc 0 # ns, xs) and
   add-inv n (u, Suc 0 # ns, xs) and
   count-inv f (u, Suc 0 # ns, xs)
  thus count-inv f (round index key p q r (u, Suc 0 # ns, xs))
  proof (cases ?t, simp add: add-suc, rule-tac allI, cases xs,
   simp-all add: disj-imp split: if-split-asm)
    fix y ys x and xs' :: 'a list
    assume ⋀n f. foldl (+) 0 ns = n ∧ length ys = n ⟶
      (∀x. count (mset ys) x = f x) ⟶ (∀x. count (mset xs') x = f x)
    moreover assume Suc (foldl (+) 0 ns) = n and Suc (length ys) = n
    ultimately have ⋀n f. (∀x. count (mset ys) x = f x) ⟶
      (∀x. count (mset xs') x = f x)
      by blast
    moreover assume A: ∀x. (y = x ⟶ Suc (count (mset ys) x) = f x) ∧
      (y ≠ x ⟶ count (mset ys) x = f x)
```

```
    have ∀x. count (mset ys) x = (f(y := f y − Suc 0)) x
      (is ∀x. - = ?f' x)
      by (simp add: A, insert spec [OF A, where x = y], simp)
    ultimately have ∀x. count (mset xs') x = ?f' x ..
    thus (y = x ⟶ Suc (count (mset xs') x) = f x) ∧
      (y ≠ x ⟶ count (mset xs') x = f x)
      by (simp, insert spec [OF A, where x = y], rule-tac impI, simp)
  qed
next
  fix index p q r u m ns n f and key :: 'a ⇒ 'b and xs :: 'a list
  let ?ws = take (Suc (Suc m)) xs
  let ?ys = drop (Suc (Suc m)) xs
  let ?r = λm'. round-suc-suc index key ?ws m m' u
  let ?t = λr' v. round index key p q r' (v, ns, ?ys)
  assume A: index-less index key
  assume
   ⋀ws a b c d e g h i n f.
     ws = ?ws ⟹ a = bn-comp m p q r ⟹ (b, c) = bn-comp m p q r ⟹
     d = ?r b ⟹ (e, g) = ?r b ⟹ (h, i) = g ⟹
       bn-inv p q (e, ns, ?ys) ⟶ add-inv n (e, ns, ?ys) ⟶
         count-inv f (e, ns, ?ys) ⟶ count-inv f (?t c e) and
   bn-inv p q (u, Suc (Suc m) # ns, xs) and
   add-inv n (u, Suc (Suc m) # ns, xs) and
   count-inv f (u, Suc (Suc m) # ns, xs)
  thus count-inv f (round index key p q r (u, Suc (Suc m) # ns, xs))
  using [[simproc del: defined-all]]
  proof (simp split: prod.split, ((rule-tac allI)+, ((rule-tac impI)+) ?)+,
   (erule-tac conjE)+, subst (asm) (2) add-base-zero, simp)
    fix m' r' v ms' ws' xs' x
    assume
      B: ?r m' = (v, ms', ws') and
      C: bn-comp m p q r = (m', r') and
      D: bn-valid m p q and
      E: Suc (Suc (foldl (+) 0 ns + m)) = n and
      F: length xs = n
    assume ⋀ws a b c d e g h i n' f.
      ws = ?ws ⟹ a = (m', r') ⟹ b = m' ∧ c = r' ⟹
      d = (v, ms', ws') ⟹ e = v ∧ g = (ms', ws') ⟹ h = ms' ∧ i = ws' ⟹
        foldl (+) 0 ns = n' ∧ n − Suc (Suc m) = n' ⟶
          (∀x. count (mset ?ys) x = f x) ⟶ (∀x. count (mset xs') x = f x)
    hence foldl (+) 0 ns = n − Suc (Suc m) ⟶
      (∀x. count (mset xs') x = count (mset ?ys) x)
      by simp
    hence count (mset xs') x = count (mset ?ys) x
      using E by simp
    moreover assume ∀x. count (mset xs) x = f x
    ultimately have f x = count (mset ?ws) x + count (mset xs') x
      by (subst (asm) append-take-drop-id [of Suc (Suc m), symmetric],
       subst (asm) mset-append, simp)
```

with B [*symmetric*] **show** *count* (*mset ws'*) x + *count* (*mset xs'*) x = $f\ x$
proof (*simp add*: *round-suc-suc-def Let-def del*: *count-add-mset mset-map*
split: *if-split-asm*, *subst* (*1 2*) *add-mset-add-single*, *simp*
only: *count-single count-union*)
let *?nmi* = *mini ?ws key*
let *?nma* = *maxi ?ws key*
let *?xmi* = *?ws* ! *?nmi*
let *?xma* = *?ws* ! *?nma*
let *?mi* = *key ?xmi*
let *?ma* = *key ?xma*
let *?k* = *case m of 0* ⇒ *m* | *Suc 0* ⇒ *m* | *Suc* (*Suc i*) ⇒ *u* + *m'*
let *?zs* = *nths ?ws* (− {*?nmi*, *?nma*})
let *?ms* = *enum ?zs index key ?k ?mi ?ma*
let *?A* = {*i*. *i* < *Suc* (*Suc m*) ∧ (*i* = *?nmi* ∨ *i* = *?nma*) ∧ *?ws* ! *i* = *x*}
have *G*: *length ?ws* = *Suc* (*Suc m*)
using *E* **and** *F* **by** *simp*
hence *H*: *card ?A* ≤ *count* (*mset ?ws*) *x*
by (*simp add*: *count-mset length-filter-conv-card*, *rule-tac card-mono*,
simp, *blast*)
show *count* (*mset* (*map the* (*fill ?zs* (*offs ?ms 0*) *index key m ?mi ?ma*))) *x*
+ (*if ?xma* = *x then 1 else 0*) + (*if ?xmi* = *x then 1 else 0*) =
count (*mset ?ws*) *x*
proof (*cases m* = *0*)
case *True*
hence *I*: *length ?zs* = *0*
using *G* **by** (*simp add*: *mini-maxi-nths*)
have *count* (*mset ?zs*) *x* = *count* (*mset ?ws*) *x* − *card ?A*
using *G* **by** (*subst nths-count*, *simp*)
hence *J*: *count* (*mset ?ws*) *x* = *card ?A*
using *H* **and** *I* **by** *simp*
from *I* **show** *?thesis*
proof (*simp*, (*rule-tac* [!] *conjI*, *rule-tac* [!] *impI*)+,
simp-all (*no-asm-simp*) *add*: *True*)
assume *?xmi* = *x* **and** *?xma* = *x*
with *G* **have** *?A* = {*?nmi*, *?nma*}
by (*rule-tac set-eqI*, *rule-tac iffI*, *simp-all*, *erule-tac disjE*,
insert mini-less [*of ?ws key*], *insert maxi-less* [*of ?ws key*],
simp-all)
with *G* **have** *card ?A* = *Suc* (*Suc 0*)
by (*simp*, *subst card-insert-disjoint*, *simp-all*,
rule-tac mini-maxi-neq, *simp*)
thus *Suc* (*Suc 0*) = *count* (*mset* (*take* (*Suc* (*Suc 0*)) *xs*)) *x*
using *True* **and** *J* **by** *simp*
next
assume *?xmi* ≠ *x* **and** *?xma* = *x*
with *G* **have** *?A* = {*?nma*}
by (*rule-tac set-eqI*, *rule-tac iffI*, *simp-all*, (*erule-tac conjE*)+,
erule-tac disjE, *insert maxi-less* [*of ?ws key*], *simp-all*)
thus *Suc 0* = *count* (*mset* (*take* (*Suc* (*Suc 0*)) *xs*)) *x*

```
    using True and J by simp
  next
    assume ?xmi = x and ?xma ≠ x
    with G have ?A = {?nmi}
      by (rule-tac set-eqI, rule-tac iffI, simp-all, (erule-tac conjE)+,
       erule-tac disjE, insert mini-less [of ?ws key], simp-all)
    thus Suc 0 = count (mset (take (Suc (Suc 0)) xs)) x
      using True and J by simp
  next
    assume ?xmi ≠ x and ?xma ≠ x
    hence ?A = {}
      by (rule-tac set-eqI, rule-tac iffI, simp-all, (erule-tac conjE)+,
       erule-tac disjE, simp-all)
    hence count (mset ?ws) x = 0
      using J by simp
    thus x ∉ set (take (Suc (Suc 0)) xs)
      using True by simp
  qed
next
  case False
  hence 0 < ?k
    by (simp, drule-tac bn-comp-fst-nonzero [OF D], subst (asm) C,
     simp split: nat.split)
  hence count (mset (map the (fill ?zs (offs ?ms 0) index key
    (length ?zs) ?mi ?ma))) x = count (mset ?zs) x
    by (rule-tac fill-offs-enum-count-item [OF A], simp, rule-tac conjI,
     ((rule-tac mini-lb | rule-tac maxi-ub), erule-tac in-set-nthsD)+)
  with G show ?thesis
  proof (simp, (rule-tac [!] conjI, rule-tac [!] impI)+,
   simp-all add: mini-maxi-nths nths-count)
    assume ?xmi = x and ?xma = x
    with G have ?A = {?nmi, ?nma}
      by (rule-tac set-eqI, rule-tac iffI, simp-all, erule-tac disjE,
       insert mini-less [of ?ws key], insert maxi-less [of ?ws key],
       simp-all)
    with G have card ?A = Suc (Suc 0)
      by (simp, subst card-insert-disjoint, simp-all,
       rule-tac mini-maxi-neq, simp)
    thus Suc (Suc (count (mset ?ws) x - card ?A)) = count (mset ?ws) x
      using H by simp
  next
    assume ?xmi ≠ x and ?xma = x
    with G have ?A = {?nma}
      by (rule-tac set-eqI, rule-tac iffI, simp-all, (erule-tac conjE)+,
       erule-tac disjE, insert maxi-less [of ?ws key], simp-all)
    thus Suc (count (mset ?ws) x - card ?A) = count (mset ?ws) x
      using H by simp
  next
    assume ?xmi = x and ?xma ≠ x
```

```
      with G have ?A = {?nmi}
        by (rule-tac set-eqI, rule-tac iffI, simp-all, (erule-tac conjE)+,
         erule-tac disjE, insert mini-less [of ?ws key], simp-all)
      thus Suc (count (mset ?ws) x − card ?A) = count (mset ?ws) x
        using H by simp
    next
      assume ?xmi ≠ x and ?xma ≠ x
      hence ?A = {}
        by (rule-tac set-eqI, rule-tac iffI, simp-all, (erule-tac conjE)+,
         erule-tac disjE, simp-all)
      thus count (mset ?ws) x − card ?A = count (mset ?ws) x
        by (simp (no-asm-simp))
    qed
  qed
 qed
qed

lemma gcsort-count-inv:
 assumes
  A: index-less index key and
  B: add-inv n t and
  C: n ≤ p
 shows ⟦t' ∈ gcsort-set index key p t; count-inv f t⟧ ⟹
  count-inv f t'
by (erule gcsort-set.induct, simp, drule gcsort-add-inv [OF A - B C],
 rule round-count-inv [OF A], simp-all del: bn-inv.simps, erule conjE,
 frule sym, erule subst, rule bn-inv-intro, insert C, simp)
```

The only remaining task is to address step 10 of the proof method, which is done by proving theorem *gcsort-count*. It holds under the conditions that the objects' number is not larger than the counters' upper bound and function *index* satisfies predicate *index-less*, and states that for any object, function *gcsort* leaves unchanged the number of its occurrences within the input objects' list.

```
theorem gcsort-count:
 assumes
  A: index-less index key and
  B: length xs ≤ p
 shows count (mset (gcsort index key p xs)) x = count (mset xs) x
proof −
 let ?t = (0, [length xs], xs)
 have count-inv (count (mset xs)) (gcsort-aux index key p ?t)
  by (rule gcsort-count-inv [OF A - B], rule gcsort-add-input,
   rule gcsort-aux-set, rule gcsort-count-input)
 hence count (mset (gcsort-out (gcsort-aux index key p ?t))) x =
```

(*count* (*mset xs*)) *x*
by (*rule gcsort-count-intro*)
moreover have *?t* = *gcsort-in xs*
by (*simp add*: *gcsort-in-def*)
ultimately show *?thesis*
by (*simp add*: *gcsort-def*)
qed

end

3 Proof of objects' sorting

theory *Sorting*
imports *Conservation*
begin

In this section, it is formally proven that GCsort actually sorts objects.
Here below, steps 5, 6, and 7 of the proof method are accomplished. Predicate *sort-inv* is satisfied just in case, for any bucket delimited by the input counters' list *ns*, the keys of the corresponding objects within the input objects' list *xs* are not larger than those of the objects, if any, to the right of that bucket. The underlying idea is that this predicate:

- is trivially satisfied by the output of function *gcsort-in*, which places all objects into a single bucket, and
- implies that *xs* is sorted if every bucket delimited by *ns* has size one, as happens when function *gcsort-aux* terminates.

fun *sort-inv* :: ('*a* ⇒ '*b*::*linorder*) ⇒ *nat* × *nat list* × '*a list* ⇒ *bool* **where**
sort-inv key (*u*, *ns*, *xs*) =
(∀ *i* < *length ns*. ∀ *j* < *offs ns 0* ! *i*. ∀ *k* ∈ {*offs ns 0* ! *i*..<*length xs*}.
key (*xs* ! *j*) ≤ *key* (*xs* ! *k*))

lemma *gcsort-sort-input*:
sort-inv key (*0*, [*length xs*], *xs*)
by *simp*

lemma *offs-nth*:
assumes
A: *find* (λ*n*. *Suc 0* < *n*) *ns* = *None* **and**
B: *foldl* (+) *0 ns* = *n* **and**
C: *k* < *n*
shows ∃ *i* < *length ns*. *offs ns 0* ! *i* = *k*
proof (*cases ns*, *insert B C*, *simp*, *cases k* = *0*, *rule exI* [*of* - *0*], *simp*,

rule ccontr, *simp* (*no-asm-use*))
fix *m ms*
assume
D: *ns* = *m* # *ms* **and**
E: *0* < *k* **and**
F: ∀ *i* < *length ns*. *offs ns 0* ! *i* ≠ *k*
have *G*: ∀ *n* ∈ *set ns*. *n* ≤ *Suc 0*
using *A* **by** (*auto simp add*: *find-None-iff*)
let *?A* = {*i*. *i* < *length ns* ∧ *offs ns 0* ! *i* < *k*}
have *H*: *Max ?A* ∈ *?A*
using *D* **and** *E* **by** (*rule-tac Max-in*, *simp-all*, *rule-tac exI* [*of - 0*], *simp*)
hence *I*: *Max ?A* < *length ns*
by *simp*
hence *J*: *offs ns 0* ! *Max ?A* = *foldl* (+) *0* (*take* (*Max ?A*) *ns*)
by (*rule offs-add*)
have *Max ?A* < *length ns* − *Suc 0* ∨ *Max ?A* = *length ns* − *Suc 0*
(**is** *?P* ∨ *?Q*)
using *H* **by** (*simp*, *arith*)
moreover {
assume *?P*
hence *K*: *Suc* (*Max ?A*) < *length ns* **by** *simp*
hence *offs ns 0* ! *Suc* (*Max ?A*) = *foldl* (+) *0* (*take* (*Suc* (*Max ?A*)) *ns*)
by (*rule offs-add*)
moreover have *take* (*Suc* (*Max ?A*)) *ns* = *take* (*Max ?A*) *ns* @ [*ns* ! *Max ?A*]
using *I* **by** (*rule take-Suc-conv-app-nth*)
ultimately have *offs ns 0* ! *Suc* (*Max ?A*) =
offs ns 0 ! *Max ?A* + *ns* ! *Max ?A*
using *J* **by** *simp*
moreover have *offs ns 0* ! *Max ?A* < *k*
using *H* **by** *simp*
moreover have *ns* ! *Max ?A* ∈ *set ns*
using *I* **by** (*rule nth-mem*)
with *G* **have** *ns* ! *Max ?A* ≤ *Suc 0* **..**
ultimately have *offs ns 0* ! *Suc* (*Max ?A*) ≤ *k* **by** *simp*
moreover have *offs ns 0* ! *Suc* (*Max ?A*) ≠ *k*
using *F* **and** *K* **by** *simp*
ultimately have *offs ns 0* ! *Suc* (*Max ?A*) < *k* **by** *simp*
hence *Suc* (*Max ?A*) ∈ *?A*
using *K* **by** *simp*
hence *Suc* (*Max ?A*) ≤ *Max ?A*
by (*rule-tac Max-ge*, *simp*)
hence *False* **by** *simp*
}
moreover {
assume *?Q*
hence *offs ns 0* ! *Max ?A* = *foldl* (+) *0* (*take* (*length ns* − *Suc 0*) *ns*)
using *J* **by** *simp*
moreover have *K*: *length ns* − *Suc 0* < *length ns*
using *D* **by** *simp*

```
    hence take (Suc (length ns − Suc 0)) ns =
      take (length ns − Suc 0) ns @ [ns ! (length ns − Suc 0)]
      by (rule take-Suc-conv-app-nth)
    hence foldl (+) 0 ns =
      foldl (+) 0 (take (length ns − Suc 0) ns @ [ns ! (length ns − Suc 0)])
      by simp
    ultimately have n = offs ns 0 ! Max ?A + ns ! (length ns − Suc 0)
      using B by simp
    moreover have offs ns 0 ! Max ?A < k
      using H by simp
    moreover have ns ! (length ns − Suc 0) ∈ set ns
      using K by (rule nth-mem)
    with G have ns ! (length ns − Suc 0) ≤ Suc 0 ..
    ultimately have n ≤ k by simp
    with C have False by simp
  }
  ultimately show False ..
qed

lemma gcsort-sort-intro:
 ⟦sort-inv key t; add-inv n t; find (λn. Suc 0 < n) (fst (snd t)) = None⟧ ⟹
    sorted (map key (gcsort-out t))
proof (cases t, simp add: sorted-iff-nth-mono-less gcsort-out-def,
 erule conjE, (rule allI)+, (rule impI)+)
  fix ns xs j k
  assume find (λn. Suc 0 < n) ns = None and foldl (+) 0 ns = n
  moreover assume A: k < n
  ultimately have ∃ i < length ns. offs ns 0 ! i = k
    by (rule offs-nth)
  then obtain i where i < length ns ∧ offs ns 0 ! i = k ..
  moreover assume ∀ i < length ns. ∀ j < offs ns 0 ! i. ∀ k ∈ {offs ns 0 ! i..<n}.
    key (xs ! j) ≤ key (xs ! k)
  hence i < length ns ⟶ j < offs ns 0 ! i ⟶ k ∈ {offs ns 0 ! i..<n} ⟶
    key (xs ! j) ≤ key (xs ! k)
    by simp
  ultimately have j < k ⟶ k < n ⟶ key (xs ! j) ≤ key (xs ! k)
    by simp
  moreover assume j < k
  ultimately show key (xs ! j) ≤ key (xs ! k)
    using A by simp
qed
```

As lemma *gcsort-sort-intro* comprises an additional assumption concerning the form of the fixed points of function *gcsort-aux*, step 8 of the proof method is necessary this time to prove that such assumption is satisfied.

```
lemma gcsort-sort-form:
 find (λn. Suc 0 < n) (fst (snd (gcsort-aux index key p t))) = None
```

by (*induction index key p t rule*: *gcsort-aux.induct*, *simp*)

Here below, step 9 of the proof method is accomplished.
In the most significant case of the proof by recursion induction of lemma *round-sort-inv*, namely that of a bucket B with size larger than two and distinct minimum and maximum keys, the following line of reasoning is adopted. Let x be an object contained in a finer-grained bucket B' resulting from B's partition, and y an object to the right of B'. Then:

- If y is contained in some other finer-grained bucket resulting from B's partition, inequality *key* $x \leq$ *key* y holds because predicate *sort-inv* is satisfied by a counters' list generated by function *enum* and an objects' list generated by function *fill* in case *fill*'s input offsets' list is computed via the composition of functions *offs* and *enum*, as happens within function *round*.
  This is proven beforehand in lemma *fill-sort-inv*.
- Otherwise, inequality *key* $x \leq$ *key* y holds as well because object x was contained in B by lemma *fill-offs-enum-count-item*, object y occurred to the right of B by lemma *round-count-inv*, and by hypothesis, the key of any object in B was not larger than that of any object to the right of B.

Using lemma *round-sort-inv*, the invariance of predicate *sort-inv* over inductive set *gcsort-set* is then proven in lemma *gcsort-sort-inv*.

lemma *mini-maxi-keys-le*:
$x \in$ *set xs* $\Longrightarrow$ *key* (*xs* ! *mini xs key*) $\leq$ *key* (*xs* ! *maxi xs key*)
by (*frule mini-lb*, *drule maxi-ub*, *erule order-trans*)

lemma *mini-maxi-keys-eq* [*rule-format*]:
key (*xs* ! *mini xs key*) = *key* (*xs* ! *maxi xs key*) $\longrightarrow$ $x \in$ *set xs* $\longrightarrow$
key x = *key* (*xs* ! *maxi xs key*)
by (*induction xs*, *simp-all add*: *Let-def*, (*rule-tac* [!] *impI*, (*rule-tac* [!] *conjI*)*?*)+,
rule-tac [*2*−*4*] *impI*, *frule-tac* [*1*−*3*] *mini-maxi-keys-le* [**where** *key* = *key*], *simp-all*)

lemma *offs-suc*:
$i <$ *length ns* $\Longrightarrow$ *offs ns* (*Suc k*) ! i = *Suc* (*offs ns k* ! i)
by (*simp add*: *offs-add add-suc*)

lemma *offs-base-zero*:
$i <$ *length ns* $\Longrightarrow$ *offs ns k* ! i = *offs ns 0* ! $i + k$
by (*simp add*: *offs-add*, *subst add-base-zero*, *simp*)

lemma *offs-append*:

offs (*ms* @ *ns*) *k* = *offs ms k* @ *offs ns* (*foldl* (+) *k ms*)
by (*induction ms arbitrary*: *k*, *simp-all*)

lemma *offs-enum-next-le* [*rule-format*]:
assumes
A: *index-less index key* **and**
B: *i* < *j* **and**
C: *j* < *n* **and**
D: ∀ *x* ∈ *set xs*. *key x* ∈ {*mi*..*ma*}
shows *offs-next* (*offs* (*enum xs index key n mi ma*) *k*) (*length xs* + *k*)
xs index key mi ma i ≤ *offs* (*enum xs index key n mi ma*) *k* ! *j*
(**is** - ≤ *offs ?ns* - ! -)
proof (*rule ccontr*, *simp add*: *not-le*)
assume *E*: *offs ?ns k* ! *j* <
offs-next (*offs ?ns k*) (*length xs* + *k*) *xs index key mi ma i*
from *B* **have** *offs-set-next* (*offs ?ns k*) *xs index key mi ma i* =
offs-set-next (*offs ?ns k*) *xs index key mi ma j*
proof (*rule-tac set-eqI*, *rule-tac iffI*, *simp-all*, *rule-tac ccontr*, *simp add*: *not-less*)
fix *m*
assume *m* < *length* (*offs ?ns k*) ∧ *i* < *m* ∧
0 < *offs-num* (*length* (*offs ?ns k*)) *xs index key mi ma m*
hence *F*: *m* ∈ *offs-set-next* (*offs ?ns k*) *xs index key mi ma i*
(**is** - ∈ *?A*)
by *simp*
hence *Min ?A* ≤ *m*
by (*rule-tac Min-le*, *simp*)
moreover assume *m* ≤ *j*
ultimately have *offs ?ns k* ! *Min ?A* ≤ *offs ?ns k* ! *j*
using *C* **by** (*rule-tac offs-mono*, *simp-all add*: *enum-length*)
hence *offs-next* (*offs ?ns k*) (*length xs* + *k*) *xs index key mi ma i* ≤
offs ?ns k ! *j*
using *F* **by** (*simp only*: *offs-next-def split*: *if-split*, *rule-tac conjI*,
blast, *simp*)
thus *False*
using *E* **by** *simp*
qed
hence *offs ?ns k* ! *j* <
offs-next (*offs ?ns k*) (*length xs* + *k*) *xs index key mi ma j*
using *E* **by** (*simp only*: *offs-next-def*)
moreover have *offs-num n xs index key mi ma j* = *0*
proof (*rule ccontr*, *simp*)
assume *0* < *offs-num n xs index key mi ma j*
hence *j* ∈ *offs-set-next* (*offs ?ns k*) *xs index key mi ma i*
(**is** - ∈ *?A*)
using *B* **and** *C* **by** (*simp add*: *offs-length enum-length*)
moreover from *this* **have** *Min ?A* ≤ *j*
by (*rule-tac Min-le*, *simp*)
hence *offs ?ns k* ! *Min ?A* ≤ *offs ?ns k* ! *j*
using *C* **by** (*erule-tac offs-mono*, *simp add*: *enum-length*)

```
    ultimately have offs-next (offs ?ns k) (length xs + k) xs index key mi ma i ≤
      offs ?ns k ! j
      by (simp only: offs-next-def split: if-split, rule-tac conjI, blast, simp)
    thus False
      using E by simp
  qed
  hence offs ?ns k ! j =
    offs-next (offs ?ns k) (length xs + k) xs index key mi ma j
    using A and C and D by (rule-tac offs-enum-zero, simp-all)
  ultimately show False by simp
qed

lemma offs-pred-ub-less:
 ⟦offs-pred ns ub xs index key mi ma; i < length ns;
   0 < offs-num (length ns) xs index key mi ma i⟧ ⟹
    ns ! i < ub
by (drule offs-pred-ub, assumption+, simp)

lemma fill-count-none [rule-format]:
  assumes A: index-less index key
  shows
   (∀ x ∈ set xs. key x ∈ {mi..ma}) ⟶
    ns ≠ [] ⟶
    offs-pred ns ub xs index key mi ma ⟶
    length xs ≤ ub ⟶
      count (mset (fill xs ns index key ub mi ma)) None = ub − length xs
  using A
proof (induction xs arbitrary: ns, simp-all add: replicate-count Let-def,
 (rule-tac impI)+, (erule-tac conjE)+, subst mset-update, simp add: fill-length,
 erule-tac offs-pred-ub-less, simp-all add: index-less-def offs-num-cons del:
 not-None-eq, subst conj-commute, rule-tac conjI, rule-tac [!] impI, rotate-tac,
 rotate-tac, erule-tac contrapos-np, rule-tac fill-index-none, simp-all)
  fix x xs and ns :: nat list
  let ?i = index key x (length ns) mi ma
  let ?ns′ = ns[?i := Suc (ns ! ?i)]
  assume ⋀ns. ns ≠ [] ⟶ offs-pred ns ub xs index key mi ma ⟶
    count (mset (fill xs ns index key ub mi ma)) None = ub − length xs
  moreover assume B: ns ≠ []
  moreover assume
   offs-pred ns ub (x # xs) index key mi ma and
   mi ≤ key x and
   key x ≤ ma
  hence offs-pred ?ns′ ub xs index key mi ma
    using A and B by (rule-tac offs-pred-cons, simp-all add: index-less-def)
  ultimately show count (mset (fill xs ?ns′ index key ub mi ma)) None −
    Suc 0 = ub − Suc (length xs)
    by simp
qed
```

lemma *fill-offs-enum-count-none* [*rule-format*]:
⟦*index-less index key*; ∀ *x* ∈ *set xs*. *key x* ∈ {*mi*..*ma*}; *0* < *n*⟧ ⟹
count (*mset* (*fill xs* (*offs* (*enum xs index key n mi ma*) *0*)
index key (*length xs*) *mi ma*)) *None* = *0*
by (*subst fill-count-none*, *simp-all*, *simp only*: *length-greater-0-conv* [*symmetric*]
offs-length enum-length, *insert offs-enum-pred* [*of index key xs mi ma n 0*], *simp*)

lemma *fill-index* [*rule-format*]:
assumes *A*: *index-less index key*
shows
(∀ *x* ∈ *set xs*. *key x* ∈ {*mi*..*ma*}) ⟶
offs-pred ns ub xs index key mi ma ⟶
i < *length ns* ⟶
0 < *offs-num* (*length ns*) *xs index key mi ma i* ⟶
j ∈ {*ns* ! *i*..<*offs-next ns ub xs index key mi ma i*} ⟶
fill xs ns index key ub mi ma ! *j* = *Some x* ⟶
index key x (*length ns*) *mi ma* = *i*
proof (*induction xs arbitrary*: *ns*, *simp add*: *offs-num-def*, *simp add*: *Let-def*,
(*rule impI*)+, (*erule conjE*)+, *simp*)
fix *y xs* **and** *ns* :: *nat list*
let *?i* = *index key x* (*length ns*) *mi ma*
let *?i'* = *index key y* (*length ns*) *mi ma*
let *?ns'* = *ns*[*?i'* := *Suc* (*ns* ! *?i'*)]
let *?P* = λ*ns*.
offs-pred ns ub xs index key mi ma ⟶
i < *length ns* ⟶
0 < *offs-num* (*length ns*) *xs index key mi ma i* ⟶
ns ! *i* ≤ *j* ∧ *j* < *offs-next ns ub xs index key mi ma i* ⟶
fill xs ns index key ub mi ma ! *j* = *Some x* ⟶
index key x (*length ns*) *mi ma* = *i*
assume
B: ∀ *x* ∈ *set xs*. *mi* ≤ *key x* ∧ *key x* ≤ *ma* **and**
C: *mi* ≤ *key y* **and**
D: *key y* ≤ *ma* **and**
E: *offs-pred ns ub* (*y* # *xs*) *index key mi ma* **and**
F: *i* < *length ns* **and**
G: *0* < *offs-num* (*length ns*) (*y* # *xs*) *index key mi ma i* **and**
H: *ns* ! *i* ≤ *j* **and**
I: *j* < *offs-next ns ub* (*y* # *xs*) *index key mi ma i* **and**
J: ⋀*ns*. *?P ns* **and**
K: (*fill xs ?ns' index key ub mi ma*)[*ns* ! *?i'* := *Some y*] ! *j* = *Some x*
have *0* < *length ns*
using *F* **by** *arith*
hence *L*: *?i'* < *length ns*
using *A* **and** *C* **and** *D* **by** (*simp add*: *index-less-def*)
hence *ns* ! *?i'* + *offs-num* (*length ns*) (*y* # *xs*) *index key mi ma ?i'* ≤ *ub*
by (*rule-tac offs-pred-ub* [*OF E*], *simp-all add*: *offs-num-cons*)
hence *ns* ! *?i'* < *ub*
by (*simp add*: *offs-num-cons*)

with *K* **show** *?i = i*
proof (*cases j = ns ! ?i'*, *simp*, *subst* (*asm*) *nth-list-update-eq*, *simp-all add*: *fill-length*)
assume
M: *j = ns ! ?i* **and**
N: *y = x*
show *?thesis*
proof (*rule ccontr*, *erule neqE*)
assume *?i < i*
hence *ns ! ?i + offs-num* (*length ns*) (*y # xs*) *index key mi ma ?i ≤ ns ! i*
using *F* **and** *G* **and** *N* **by** (*rule-tac offs-pred-asc* [*OF E*], *simp-all add*: *offs-num-cons*)
hence *j < ns ! i*
using *M* **and** *N* **by** (*simp add*: *offs-num-cons*)
thus *False*
using *H* **by** *simp*
next
let *?A = offs-set-next ns* (*y # xs*) *index key mi ma i*
assume *i < ?i*
hence *O*: *?i ∈ ?A*
using *N* **and** *L* **by** (*simp add*: *offs-num-cons*)
hence *P*: *Min ?A ∈ ?A*
by (*rule-tac Min-in*, *simp*, *blast*)
have *Min ?A ≤ ?i*
using *O* **by** (*rule-tac Min-le*, *simp*)
moreover have *?A ≠ {}*
using *O* **by** *blast*
ultimately have *offs-next ns ub* (*y # xs*) *index key mi ma i ≤ j*
using *M* **proof** (*simp only*: *offs-next-def*, *simp*, *subst* (*asm*) *le-less*, *erule-tac disjE*, *simp-all*)
assume *Min ?A < ?i*
hence *ns ! Min ?A + offs-num* (*length ns*) (*y # xs*) *index key mi ma* (*Min ?A*) *≤ ns ! ?i*
using *O* **and** *P* **by** (*rule-tac offs-pred-asc* [*OF E*], *simp-all*)
thus *ns ! Min ?A ≤ ns ! ?i* **by** *simp*
qed
thus *False*
using *I* **by** *simp*
qed
next
assume
M: *j ≠ ns ! ?i'* **and**
N: *fill xs ?ns' index key ub mi ma ! j = Some x*
have *?P ?ns'* **using** *J* **.**
moreover from *D* **and** *F* **have** *offs-pred ?ns' ub xs index key mi ma*
using *L* **by** (*rule-tac offs-pred-cons* [*OF E*], *simp*)
moreover have *i < length ?ns'*
using *F* **by** *simp*
moreover have *0 < offs-num* (*length ?ns'*) *xs index key mi ma i*

```
    proof (rule ccontr, simp)
      assume O: offs-num (length ns) xs index key mi ma i = 0
      hence P: offs-num (length ns) (y # xs) index key mi ma i = Suc 0
        using G by (simp add: offs-num-cons split: if-split-asm)
      hence i = ?i'
        using O by (simp add: offs-num-cons split: if-split-asm)
      hence ns ! i < j
        using H and M by simp
      hence ns ! i + offs-num (length ns) (y # xs) index key mi ma i ≤ j
        using P by simp
      with F and G and I have offs-none ns ub (y # xs) index key mi ma j
        by (simp add: offs-none-def, rule-tac disjI1, rule-tac exI
          [where x = i], simp)
      with B and C and D and E and F have
       fill (y # xs) ns index key ub mi ma ! j = None
        by (rule-tac fill-none [OF A], simp-all, erule-tac disjE, simp-all, auto)
      thus False
        using K by (simp add: Let-def)
    qed
    moreover have ?ns' ! i ≤ j
      using F and H and M by (cases ?i' = i, simp-all)
    moreover have offs-next ns ub (y # xs) index key mi ma i ≤
      offs-next ?ns' ub xs index key mi ma i
      using G and L by (rule-tac offs-pred-next-cons [OF E], simp)
    hence j < offs-next ?ns' ub xs index key mi ma i
      using I by simp
    ultimately show ?thesis
      using N by simp
  qed
qed

lemma fill-offs-enum-index [rule-format]:
 index-less index key ⟹
  ∀ x ∈ set xs. key x ∈ {mi..ma} ⟹
  i < n ⟹
  0 < offs-num n xs index key mi ma i ⟹
  j ∈ {offs (enum xs index key n mi ma) 0 ! i..<
    offs-next (offs (enum xs index key n mi ma) 0) (length xs)
      xs index key mi ma i} ⟹
  fill xs (offs (enum xs index key n mi ma) 0) index key (length xs)
    mi ma ! j = Some x ⟹
    index key x n mi ma = i
by (insert fill-index [of index key xs mi ma offs (enum xs index key n mi ma) 0
 length xs i j x], insert offs-enum-pred [of index key xs mi ma n 0],
 simp add: offs-length enum-length)

lemma fill-sort-inv [rule-format]:
  assumes
    A: index-less index key and
```

B: *index-mono index key* **and**
C: $\forall\, x \in$ *set xs*. *key x* $\in \{mi..ma\}$
shows *sort-inv key* (*u*, *enum xs index key n mi ma*,
map the (*fill xs* (*offs* (*enum xs index key n mi ma*) *0*)
index key (*length xs*) *mi ma*))
(**is** *sort-inv* - (-, *?ns*, -))
proof (*simp*, (*rule allI*, *rule impI*)+, *rule ballI*, *simp add*: *enum-length fill-length*,
erule conjE)
fix *i j k*
let *?A* = $\{i.\ i < n \wedge 0 <$ *?ns* ! $i\}$
let *?B* = $\{i.\ i < n \wedge$ *offs ?ns 0* ! $i \leq j \wedge 0 <$ *offs-num n xs index key mi ma i*$\}$
let *?C* = $\{i.\ i < n \wedge$ *offs ?ns 0* ! $i \leq k \wedge 0 <$ *offs-num n xs index key mi ma i*$\}$
assume
D: $i < n$ **and**
E: $j <$ *offs ?ns 0* ! i **and**
F: *offs ?ns 0* ! $i \leq k$ **and**
G: $k <$ *length xs*
have *H*: $\forall\, i <$ *length xs*.
$\exists\, x$. *fill xs* (*offs ?ns 0*) *index key* (*length xs*) *mi ma* ! $i =$ *Some x*
proof (*rule allI*, *rule impI*, *rule ccontr*, *simp*)
fix *m*
assume *fill xs* (*offs ?ns 0*) *index key* (*length xs*) *mi ma* ! $m =$ *None*
moreover assume $m <$ *length xs*
hence *fill xs* (*offs ?ns 0*) *index key* (*length xs*) *mi ma* ! *m*
$\in$ *set* (*fill xs* (*offs ?ns 0*) *index key* (*length xs*) *mi ma*)
by (*rule-tac nth-mem*, *simp add*: *fill-length*)
ultimately have *None* $\in$ *set* (*fill xs* (*offs ?ns 0*) *index key* (*length xs*) *mi ma*)
by *simp*
moreover have
count (*mset* (*fill xs* (*offs ?ns 0*) *index key* (*length xs*) *mi ma*)) *None* = *0*
using *C* **and** *D* **by** (*rule-tac fill-offs-enum-count-none* [*OF A*], *simp-all*)
ultimately show *False* **by** *simp*
qed
have $\exists\, y\ ys.\ xs = y \#\ ys$
using *G* **by** (*rule-tac list.exhaust* [*of xs*], *simp-all*)
then obtain *z* **and** *zs* **where** *I*: $xs = z \#\ zs$ **by** *blast*
hence *index key z n mi ma* $< n$
using *A* **and** *C* **and** *D* **by** (*simp add*: *index-less-def*)
moreover from *this* **have** *0* $<$ *?ns* ! *index key z n mi ma*
using *I* **by** (*simp add*: *Let-def*, *subst nth-list-update-eq*, *simp-all add*:
enum-length)
ultimately have *index key z n mi ma* $\in$ *?A* **by** *simp*
hence *J*: *Min ?A* $\in$ *?A*
by (*rule-tac Min-in*, *simp*, *blast*)
hence *offs ?ns 0* ! *0* = *offs ?ns 0* ! *Min ?A*
proof (*cases Min ?A* = *0*, *simp-all*, *erule-tac offs-equal*, *simp-all add*:
enum-length, *rule-tac ccontr*, *erule-tac conjE*, *simp*)
fix *m*
assume $m <$ *Min ?A* **and** *Min ?A* $< n$ **and** *0* $<$ *?ns* ! *m*

moreover from *this* **have** $Min\ ?A \leq m$
by (*rule-tac Min-le*, *simp-all*)
ultimately show *False* **by** *simp*
qed
moreover have $\exists\, m\ ms.\ ?ns = m \#\ ms$
using *D* **by** (*rule-tac list.exhaust* [*of ?ns*], *simp-all*,
simp only: *length-0-conv* [*symmetric*] *enum-length*)
then obtain *m* **and** *ms* **where** $?ns = m \#\ ms$ **by** *blast*
ultimately have $offs\ ?ns\ 0\ !\ Min\ ?A = 0$ **by** *simp*
hence $Min\ ?A \in ?B$
using *J* **by** (*simp*, *subst enum-offs-num* [*symmetric*], *simp-all*)
hence *K*: $Max\ ?B \in ?B$
by (*rule-tac Max-in*, *simp*, *blast*)
moreover have $j < offs\text{-}next\ (offs\ ?ns\ 0)\ (length\ xs)$
xs index key mi ma (*Max ?B*)
proof (*simp only*: *offs-next-def split*: *if-split*, *rule conjI*, *rule-tac* [!] *impI*,
rule-tac [*2*] *ccontr*, *simp-all only*: *not-less*)
show $j < length\ xs$
using *E* **and** *F* **and** *G* **by** *simp*
next
assume $offs\text{-}set\text{-}next\ (offs\ ?ns\ 0)\ xs\ index\ key\ mi\ ma\ (Max\ ?B) \neq \{\}$
(**is** $?Z \neq$ -)
hence *L*: $Min\ ?Z \in ?Z$
by (*rule-tac Min-in*, *simp*)
moreover assume $offs\ ?ns\ 0\ !\ Min\ ?Z \leq j$
ultimately have $Min\ ?Z \in ?B$
by (*simp add*: *offs-length enum-length*)
hence $Min\ ?Z \leq Max\ ?B$
by (*rule-tac Max-ge*, *simp*)
thus *False*
using *L* **by** *simp*
qed
moreover have
$\exists\, x.\ fill\ xs\ (offs\ ?ns\ 0)\ index\ key\ (length\ xs)\ mi\ ma\ !\ j = Some\ x$
using *E* **and** *F* **and** *G* **and** *H* **by** *simp*
then obtain *x* **where**
L: $fill\ xs\ (offs\ ?ns\ 0)\ index\ key\ (length\ xs)\ mi\ ma\ !\ j = Some\ x$ **..**
ultimately have *M*: $index\ key\ x\ n\ mi\ ma = Max\ ?B$
using *C* **by** (*rule-tac fill-offs-enum-index* [*OF A*], *simp-all*)
have *N*: $Max\ ?B \in ?C$
using *E* **and** *F* **and** *K* **by** *simp*
hence $Max\ ?C \in ?C$
by (*rule-tac Max-in*, *simp*, *blast*)
moreover have *O*: $k < offs\text{-}next\ (offs\ ?ns\ 0)\ (length\ xs)$
xs index key mi ma (*Max ?C*)
proof (*simp only*: *offs-next-def split*: *if-split*, *rule conjI*, *rule-tac* [!] *impI*,
rule-tac [*2*] *ccontr*, *simp-all only*: *not-less*)
show $k < length\ xs$ **using** *G* **.**
next

assume *offs-set-next* (*offs ?ns 0*) *xs index key mi ma* (*Max ?C*) ≠ {}
(**is** *?Z* ≠ -)
hence *P*: *Min ?Z* ∈ *?Z*
by (*rule-tac Min-in*, *simp*)
moreover assume *offs ?ns 0* ! *Min ?Z* ≤ *k*
ultimately have *Min ?Z* ∈ *?C*
by (*simp add*: *offs-length enum-length*)
hence *Min ?Z* ≤ *Max ?C*
by (*rule-tac Max-ge*, *simp*)
thus *False*
using *P* **by** *simp*
qed
moreover have
∃ *x*. *fill xs* (*offs ?ns 0*) *index key* (*length xs*) *mi ma* ! *k* = *Some x*
using *G* **and** *H* **by** *simp*
then obtain *y* **where**
P: *fill xs* (*offs ?ns 0*) *index key* (*length xs*) *mi ma* ! *k* = *Some y* **..**
ultimately have *Q*: *index key y n mi ma* = *Max ?C*
using *C* **by** (*rule-tac fill-offs-enum-index* [*OF A*], *simp-all*)
have *Max ?B* ≤ *Max ?C*
using *N* **by** (*rule-tac Max-ge*, *simp*)
hence *Max ?B* < *Max ?C*
proof (*rule-tac ccontr*, *simp*)
assume *Max ?B* = *Max ?C*
hence *offs ?ns 0* ! *i* < *offs-next* (*offs ?ns 0*) (*length xs*)
xs index key mi ma (*Max ?B*)
using *F* **and** *O* **by** *simp*
moreover have *offs ?ns 0* ! *Max ?B* < *offs ?ns 0* ! *i*
using *E* **and** *K* **by** *simp*
with *K* **have** *Max ?B* < *i*
by (*erule-tac contrapos-pp*, *subst not-less*, *subst* (*asm*) *not-less*,
erule-tac offs-mono, *simp add*: *enum-length*)
hence *offs-next* (*offs ?ns 0*) (*length xs* + *0*) *xs index key mi ma* (*Max ?B*) ≤
offs ?ns 0 ! *i*
using *C* **and** *D* **by** (*rule-tac offs-enum-next-le* [*OF A*], *simp-all*)
ultimately show *False* **by** *simp*
qed
hence *R*: *index key x n mi ma* < *index key y n mi ma*
using *M* **and** *Q* **by** *simp*
have *count* (*mset* (*map the* (*fill xs* (*offs ?ns 0*) *index key*
(*length xs*) *mi ma*))) *x* = *count* (*mset xs*) *x*
using *C* **and** *D* **by** (*rule-tac fill-offs-enum-count-item* [*OF A*], *simp-all*)
moreover have *S*: *j* < *length* (*map the* (*fill xs* (*offs ?ns 0*) *index key*
(*length xs*) *mi ma*))
using *E* **and** *F* **and** *G* **by** (*simp add*: *fill-length*)
hence *map the* (*fill xs* (*offs ?ns 0*) *index key* (*length xs*) *mi ma*) ! *j*
∈ *set* (*map the* (*fill xs* (*offs ?ns 0*) *index key* (*length xs*) *mi ma*))
by (*rule nth-mem*)
hence *0* < *count* (*mset* (*map the* (*fill xs* (*offs ?ns 0*) *index key*

```
    (length xs) mi ma))) x
    using L and S by simp
  ultimately have T: x ∈ set xs by simp
  have count (mset (map the (fill xs (offs ?ns 0) index key
    (length xs) mi ma))) y = count (mset xs) y
    using C and D by (rule-tac fill-offs-enum-count-item [OF A], simp-all)
  moreover have U: k < length (map the (fill xs (offs ?ns 0) index key
    (length xs) mi ma))
    using G by (simp add: fill-length)
  hence map the (fill xs (offs ?ns 0) index key (length xs) mi ma) ! k
    ∈ set (map the (fill xs (offs ?ns 0) index key (length xs) mi ma))
    by (rule nth-mem)
  hence 0 < count (mset (map the (fill xs (offs ?ns 0) index key
    (length xs) mi ma))) y
    using P and U by simp
  ultimately have V: y ∈ set xs by simp
  have key y ≤ key x ⟶ index key y n mi ma ≤ index key x n mi ma
    using B and C and T and V by (simp add: index-mono-def)
  hence key x < key y
    by (rule-tac contrapos-pp [OF R], simp add: not-less)
  thus key (the (fill xs (offs ?ns 0) index key (length xs) mi ma ! j)) ≤
    key (the (fill xs (offs ?ns 0) index key (length xs) mi ma ! k))
    using L and P by simp
qed

lemma round-sort-inv [rule-format]:
 index-less index key ⟶ index-mono index key ⟶ bn-inv p q t ⟶
    add-inv n t ⟶ sort-inv key t ⟶ sort-inv key (round index key p q r t)
proof (induction index key p q r t arbitrary: n rule: round.induct,
 (rule-tac [!] impI)+, simp, simp-all only: simp-thms)
  fix index p q r u ns xs n and key :: 'a ⇒ 'b
  let ?t = round index key p q r (u, ns, xs)
  assume ⋀n. bn-inv p q (u, ns, xs) ⟶ add-inv n (u, ns, xs) ⟶
    sort-inv key (u, ns, xs) ⟶ sort-inv key ?t
  hence bn-inv p q (u, ns, xs) ⟶ add-inv n (u, ns, xs) ⟶
    sort-inv key (u, ns, xs) ⟶ sort-inv key ?t .
  moreover assume
   bn-inv p q (u, 0 # ns, xs)
   add-inv n (u, 0 # ns, xs) and
   sort-inv key (u, 0 # ns, xs)
  ultimately show sort-inv key (round index key p q r (u, 0 # ns, xs))
    by auto
next
  fix index p q r u ns xs n and key :: 'a ⇒ 'b
  let ?t = round index key p q r (u, ns, tl xs)
  assume
    A: index-less index key and
    B: bn-inv p q (u, Suc 0 # ns, xs)
  moreover assume
```

```
 ⋀n. bn-inv p q (u, ns, tl xs) ⟶ add-inv n (u, ns, tl xs) ⟶
    sort-inv key (u, ns, tl xs) ⟶ sort-inv key ?t and
 add-inv n (u, Suc 0 # ns, xs) and
 sort-inv key (u, Suc 0 # ns, xs)
ultimately show sort-inv key (round index key p q r (u, Suc 0 # ns, xs))
proof (cases ?t, cases xs, simp-all add: add-suc, (rule-tac allI, rule-tac impI)+,
 rule-tac ballI, simp, (erule-tac conjE)+, simp)
  fix i j k y ys u' ns' xs'
  assume
    C: round index key p q r (u, ns, ys) = (u', ns', xs') and
    D: Suc (foldl (+) 0 ns) = n and
    E: Suc (length ys) = n and
    F: ∀ i < Suc (length ns).
      ∀ j < (0 # offs ns (Suc 0)) ! i. ∀ k ∈ {(0 # offs ns (Suc 0)) ! i..<n}.
        key ((y # ys) ! j) ≤ key (ys ! (k − Suc 0))
  assume A': j < (0 # offs ns' (Suc 0)) ! i
  hence ∃ i'. i = Suc i'
    by (rule-tac nat.exhaust [of i], simp-all)
  then obtain i' where B': i = Suc i' ..
  assume i < Suc (length ns')
  hence G: i' < length ns'
    using B' by simp
  hence H: j < Suc (offs ns' 0 ! i')
    using A' and B' by (simp add: offs-suc)
  assume (0 # offs ns' (Suc 0)) ! i ≤ k
  hence Suc (offs ns' 0 ! i') ≤ k
    using B' and G by (simp add: offs-suc)
  moreover from this have ∃ k'. k = Suc k'
    by (rule-tac nat.exhaust [of k], simp-all)
  then obtain k' where I: k = Suc k' ..
  ultimately have J: offs ns' 0 ! i' ≤ k' by simp
  assume k < Suc (length xs')
  hence K: k' < length xs'
    using I by simp
  let ?P = λn. foldl (+) 0 ns = n ∧ length ys = n
  let ?Q = λn. ∀ i < length ns.
    ∀ j < offs ns 0 ! i. ∀ k ∈ {offs ns 0 ! i..<n}.
      key (ys ! j) ≤ key (ys ! k)
  let ?R = ∀ i < length ns'.
    ∀ j < offs ns' 0 ! i. ∀ k ∈ {offs ns' 0 ! i..<length xs'}.
      key (xs' ! j) ≤ key (xs' ! k)
  assume ⋀n. ?P n ⟶ ?Q n ⟶ ?R
  hence ?P (n − Suc 0) ⟶ ?Q (n − Suc 0) ⟶ ?R .
  hence ?Q (n − Suc 0) ⟶ ?R
    using D and E by auto
  moreover have ?Q (n − Suc 0)
  proof ((rule allI, rule impI)+, rule ballI, simp, erule conjE)
    fix i j k
    have Suc i < Suc (length ns) ⟶ (∀ j < (0 # offs ns (Suc 0)) ! Suc i.
```

```
    ∀ k ∈ {(0 # offs ns (Suc 0)) ! Suc i..<n}.
      key ((y # ys) ! j) ≤ key (ys ! (k − Suc 0)))
    using F ..
  moreover assume i < length ns
  ultimately have Suc j < Suc (offs ns 0 ! i) ⟶
    (∀ k ∈ {Suc (offs ns 0 ! i)..<n}.
      key ((y # ys) ! Suc j) ≤ key (ys ! (k − Suc 0)))
    by (auto simp add: offs-suc)
  moreover assume j < offs ns 0 ! i
  ultimately have ∀ k ∈ {Suc (offs ns 0 ! i)..<n}.
    key (ys ! j) ≤ key (ys ! (k − Suc 0))
    by simp
  moreover assume offs ns 0 ! i ≤ k and k < n − Suc 0
  hence Suc k ∈ {Suc (offs ns 0 ! i)..<n} by simp
  ultimately have key (ys ! j) ≤ key (ys ! (Suc k − Suc 0)) ..
  thus key (ys ! j) ≤ key (ys ! k) by simp
qed
ultimately have ?R ..
from I show key ((y # xs') ! j) ≤ key (xs' ! (k − Suc 0))
proof (cases j, simp-all)
  have bn-inv p q (u, ns, ys)
    using B by simp
  moreover have add-inv (n − Suc 0) (u, ns, ys)
    using D and E by auto
  moreover have count-inv (λx. count (mset ys) x) (u, ns, ys) by simp
  ultimately have count-inv (λx. count (mset ys) x)
    (round index key p q r (u, ns, ys))
    by (rule round-count-inv [OF A])
  hence count (mset xs') (xs' ! k') = count (mset ys) (xs' ! k')
    using C by simp
  moreover have 0 < count (mset xs') (xs' ! k')
    using K by simp
  ultimately have xs' ! k' ∈ set ys by simp
  hence ∃ m < length ys. ys ! m = xs' ! k'
    by (simp add: in-set-conv-nth)
  then obtain m where L: m < length ys ∧ ys ! m = xs' ! k' ..
  have Suc 0 < Suc (length ns) ⟶ (∀ j < (0 # offs ns (Suc 0)) ! Suc 0.
    ∀ k ∈ {(0 # offs ns (Suc 0)) ! Suc 0..<n}.
      key ((y # ys) ! j) ≤ key (ys ! (k − Suc 0)))
    using F ..
  moreover have M: 0 < length ns
    using D [symmetric] and E and L by (rule-tac ccontr, simp)
  ultimately have ∀ k ∈ {Suc (offs ns 0 ! 0)..<n}.
    key y ≤ key (ys ! (k − Suc 0))
    by (auto simp add: offs-suc)
  hence ∀ k ∈ {Suc 0..<n}. key y ≤ key (ys ! (k − Suc 0))
    using M by (cases ns, simp-all)
  moreover have Suc m ∈ {Suc 0..<n}
    using E and L by simp
```

```
      ultimately have key y ≤ key (ys ! (Suc m − Suc 0)) ..
      thus key y ≤ key (xs' ! k')
        using L by simp
    next
      case (Suc j')
      moreover have i' < length ns' ⟶ (∀ j < offs ns' 0 ! i'.
        ∀ k ∈ {offs ns' 0 ! i'..<length xs'}. key (xs' ! j) ≤ key (xs' ! k))
        using ‹?R› ..
      hence j' < offs ns' 0 ! i' ⟶
        (∀ k ∈ {offs ns' 0 ! i'..<length xs'}. key (xs' ! j') ≤ key (xs' ! k))
        using G by simp
      ultimately have ∀ k ∈ {offs ns' 0 ! i'..<length xs'}.
        key (xs' ! j') ≤ key (xs' ! k)
        using H by simp
      moreover have k' ∈ {offs ns' 0 ! i'..<length xs'}
        using J and K by simp
      ultimately show key (xs' ! j') ≤ key (xs' ! k') ..
    qed
  qed
next
  fix index p q r u m ns n and key :: 'a ⇒ 'b and xs :: 'a list
  let ?ws = take (Suc (Suc m)) xs
  let ?ys = drop (Suc (Suc m)) xs
  let ?r = λm'. round-suc-suc index key ?ws m m' u
  let ?t = λr' v. round index key p q r' (v, ns, ?ys)
  assume
    A: index-less index key and
    B: index-mono index key and
    C: bn-inv p q (u, Suc (Suc m) # ns, xs)
  assume
   ⋀ws a b c d e f g h n.
      ws = ?ws ⟹ a = bn-comp m p q r ⟹ (b, c) = bn-comp m p q r ⟹
      d = ?r b ⟹ (e, f) = ?r b ⟹ (g, h) = f ⟹
        bn-inv p q (e, ns, ?ys) ⟶ add-inv n (e, ns, ?ys) ⟶
          sort-inv key (e, ns, ?ys) ⟶ sort-inv key (?t c e) and
   add-inv n (u, Suc (Suc m) # ns, xs) and
   sort-inv key (u, Suc (Suc m) # ns, xs)
  with C show sort-inv key (round index key p q r (u, Suc (Suc m) # ns, xs))
  using [[simproc del: defined-all]]
  proof (simp split: prod.split, ((rule-tac allI)+, (rule-tac impI)+)+,
   rule-tac ballI, simp, (erule-tac conjE)+, subst (asm) (2) add-base-zero, simp)
    fix m' r' v ms' ws' u' ns' xs' i j k
    let ?nmi = mini ?ws key
    let ?nma = maxi ?ws key
    let ?xmi = ?ws ! ?nmi
    let ?xma = ?ws ! ?nma
    let ?mi = key ?xmi
    let ?ma = key ?xma
    let ?k = case m of 0 ⇒ m | Suc 0 ⇒ m | Suc (Suc i) ⇒ u + m'
```

let *?zs* = *nths ?ws* (− {*?nmi*, *?nma*})
let *?ms* = *enum ?zs index key ?k ?mi ?ma*
let *?P* = λ*n′*. *foldl* (+) *0 ns* = *n′* ∧ *n* − *Suc* (*Suc m*) = *n′*
let *?Q* = λ*n′*. ∀ *i* < *length ns*.
∀ *j* < *offs ns 0* ! *i*. ∀ *k* ∈ {*offs ns 0* ! *i*..<*n′*}.
key (*xs* ! *Suc* (*Suc* (*m* + *j*))) ≤ *key* (*xs* ! *Suc* (*Suc* (*m* + *k*)))
let *?R* = ∀ *i* < *length ns′*.
∀ *j* < *offs ns′ 0* ! *i*. ∀ *k* ∈ {*offs ns′ 0* ! *i*..<*length xs′*}.
key (*xs′* ! *j*) ≤ *key* (*xs′* ! *k*)
assume
D: *?r m′* = (*v*, *ms′*, *ws′*) **and**
E: *?t r′ v* = (*u′*, *ns′*, *xs′*) **and**
F: *bn-comp m p q r* = (*m′*, *r′*) **and**
G: *bn-valid m p q* **and**
H: *Suc* (*Suc* (*foldl* (+) *0 ns* + *m*)) = *n* **and**
I: *length xs* = *n* **and**
J: ∀ *i* < *Suc* (*length ns*). ∀ *j* < (*0* # *offs ns* (*Suc* (*Suc m*))) ! *i*.
∀ *k* ∈ {(*0* # *offs ns* (*Suc* (*Suc m*))) ! *i*..<*n*}.
key (*xs* ! *j*) ≤ *key* (*xs* ! *k*) **and**
K: *i* < *length ms′* + *length ns′* **and**
L: *j* < *offs* (*ms′* @ *ns′*) *0* ! *i* **and**
M: *offs* (*ms′* @ *ns′*) *0* ! *i* ≤ *k* **and**
N: *k* < *length ws′* + *length xs′*
have *O*: *length ?ws* = *Suc* (*Suc m*)
using *H* **and** *I* **by** *simp*
with *D* [*symmetric*] **have** *P*: *length ws′* = *Suc* (*Suc m*)
by (*simp add*: *round-suc-suc-def Let-def fill-length split*: *if-split-asm*)
have *Q*: ⋀*i*. *i* < *m* ⟹
the (*fill ?zs* (*offs ?ms 0*) *index key m ?mi ?ma* ! *i*) ∈ *set ?ws*
proof −
fix *i*
let *?x* = *the* (*fill ?zs* (*offs ?ms 0*) *index key m ?mi ?ma* ! *i*)
assume *R*: *i* < *m*
hence *0* < *m* **by** *simp*
hence *0* < *?k*
by (*drule-tac bn-comp-fst-nonzero* [*OF G*], *subst* (*asm*) *F*,
simp split: *nat.split*)
hence *count* (*mset* (*map the* (*fill ?zs* (*offs ?ms 0*)
index key (*length ?zs*) *?mi ?ma*))) *?x* = *count* (*mset ?zs*) *?x*
by (*rule-tac fill-offs-enum-count-item* [*OF A*], *simp*, *rule-tac conjI*,
((*rule-tac mini-lb* | *rule-tac maxi-ub*), *erule-tac in-set-nthsD*)+)
hence *count* (*mset* (*map the* (*fill ?zs* (*offs ?ms 0*)
index key m ?mi ?ma))) *?x* = *count* (*mset ?zs*) *?x*
using *O* **by** (*simp add*: *mini-maxi-nths*)
moreover have *0* < *count* (*mset* (*map the* (*fill ?zs* (*offs ?ms 0*)
index key m ?mi ?ma))) *?x*
using *R* **by** (*simp add*: *fill-length*)
ultimately have *?x* ∈ *set ?zs* **by** *simp*
thus *?x* ∈ *set ?ws*

by (*rule in-set-nthsD*)
qed
have *R*: $\bigwedge i.\ i < Suc\ (Suc\ m) \Longrightarrow Suc\ (Suc\ m) \leq k \Longrightarrow$
$key\ (?ws\ !\ i) \leq key\ (xs'\ !\ (k - Suc\ (Suc\ m)))$
proof −
fix *i*
have *bn-inv p q* (*v*, *ns*, *?ys*)
using *C* **by** *simp*
moreover have *add-inv* $(n - Suc\ (Suc\ m))$ (*v*, *ns*, *?ys*)
using *H* **and** *I* **by** *simp*
moreover have *count-inv* $(\lambda x.\ count\ (mset\ ?ys)\ x)$ (*v*, *ns*, *?ys*)
by *simp*
ultimately have *count-inv* $(\lambda x.\ count\ (mset\ ?ys)\ x)$ $(?t\ r'\ v)$
by (*rule round-count-inv* [*OF A*])
hence *S*: $count\ (mset\ xs')\ (xs'\ !\ (k - Suc\ (Suc\ m))) =$
$count\ (mset\ ?ys)\ (xs'\ !\ (k - Suc\ (Suc\ m)))$
using *E* **by** *simp*
have $k < Suc\ (Suc\ (m + length\ xs'))$
using *N* **and** *P* **by** *simp*
moreover assume $Suc\ (Suc\ m) \leq k$
ultimately have $0 < count\ (mset\ xs')\ (xs'\ !\ (k - Suc\ (Suc\ m)))$
by *simp*
hence $xs'\ !\ (k - Suc\ (Suc\ m)) \in set\ ?ys$
using *S* **by** *simp*
hence $\exists p < length\ ?ys.\ ?ys\ !\ p = xs'\ !\ (k - Suc\ (Suc\ m))$
by (*simp add*: *in-set-conv-nth*)
then obtain *p* **where**
T: $p < length\ ?ys \wedge ?ys\ !\ p = xs'\ !\ (k - Suc\ (Suc\ m))$ **..**
have $Suc\ 0 < Suc\ (length\ ns) \longrightarrow$
$(\forall j < (0\ \#\ offs\ ns\ (Suc\ (Suc\ m)))\ !\ Suc\ 0.$
$\forall k \in \{(0\ \#\ offs\ ns\ (Suc\ (Suc\ m)))\ !\ Suc\ 0..<n\}.$
$key\ (xs\ !\ j) \leq key\ (xs\ !\ k))$
using *J* **..**
moreover have *U*: $0 < length\ ns$
using *H* [*symmetric*] **and** *I* **and** *T* **by** (*rule-tac ccontr*, *simp*)
ultimately have $\forall j < offs\ ns\ (Suc\ (Suc\ m))\ !\ 0.$
$\forall k \in \{offs\ ns\ (Suc\ (Suc\ m))\ !\ 0..<n\}.\ key\ (xs\ !\ j) \leq key\ (xs\ !\ k)$
by *simp*
moreover assume $i < Suc\ (Suc\ m)$
moreover have $offs\ ns\ (Suc\ (Suc\ m))\ !\ 0 = Suc\ (Suc\ m)$
using *U* **by** (*subst offs-base-zero*, *simp*, *cases ns*, *simp-all*)
ultimately have $\forall k \in \{Suc\ (Suc\ m)..<n\}.\ key\ (?ws\ !\ i) \leq key\ (xs\ !\ k)$
by *simp*
moreover have $Suc\ (Suc\ m) + p \in \{Suc\ (Suc\ m)..<n\}$
using *I* **and** *T* **by** (*simp*, *arith*)
ultimately have $key\ (?ws\ !\ i) \leq key\ (xs\ !\ (Suc\ (Suc\ m) + p))$ **..**
thus $key\ (?ws\ !\ i) \leq key\ (xs'\ !\ (k - Suc\ (Suc\ m)))$
using *O* **and** *T* **by** *simp*
qed

assume $\bigwedge ws\ a\ b\ c\ d\ e\ f\ g\ h\ n'.$
$ws = ?ws \Longrightarrow a = (m',\ r') \Longrightarrow b = m' \wedge c = r' \Longrightarrow$
$d = (v,\ ms',\ ws') \Longrightarrow e = v \wedge f = (ms',\ ws') \Longrightarrow g = ms' \wedge h = ws' \Longrightarrow$
$?P\ n' \longrightarrow ?Q\ n' \longrightarrow ?R$
hence $?P\ (n - Suc\ (Suc\ m)) \longrightarrow ?Q\ (n - Suc\ (Suc\ m)) \longrightarrow ?R$
by *simp*
hence $?Q\ (n - Suc\ (Suc\ m)) \longrightarrow ?R$
using H **by** *simp*
moreover have $?Q\ (n - Suc\ (Suc\ m))$
proof ((*rule allI*, *rule impI*)+, *rule ballI*, *simp*, *erule conjE*,
subst (*1 2*) *append-take-drop-id* [*of Suc* (*Suc m*), *symmetric*],
simp only: *nth-append O*, *simp*)
fix $i\ j\ k$
have $Suc\ i < Suc\ (length\ ns) \longrightarrow$
$(\forall j < (0\ \#\ offs\ ns\ (Suc\ (Suc\ m)))\ !\ Suc\ i.$
$\forall k \in \{(0\ \#\ offs\ ns\ (Suc\ (Suc\ m)))\ !\ Suc\ i..<n\}.$
$key\ ((xs)\ !\ j) \leq key\ (xs\ !\ k))$
using J **..**
moreover assume $i < length\ ns$
ultimately have $j < offs\ ns\ 0\ !\ i \longrightarrow$
$(\forall k \in \{offs\ ns\ 0\ !\ i + Suc\ (Suc\ m)..<n\}.$
$key\ (xs\ !\ (Suc\ (Suc\ m) + j)) \leq key\ (xs\ !\ k))$
by (*simp*, *subst* (*asm*) *offs-base-zero*, *simp*,
subst (*asm*) (*2*) *offs-base-zero*, *simp-all*)
moreover assume $j < offs\ ns\ 0\ !\ i$
ultimately have $\forall k \in \{offs\ ns\ 0\ !\ i + Suc\ (Suc\ m)..<n\}.$
$key\ (?ys\ !\ j) \leq key\ (xs\ !\ k)$
using O **by** *simp*
moreover assume $offs\ ns\ 0\ !\ i \leq k$ **and** $k < n - Suc\ (Suc\ m)$
hence $Suc\ (Suc\ m) + k \in \{offs\ ns\ 0\ !\ i + Suc\ (Suc\ m)..<n\}$
by *simp*
ultimately have $key\ (?ys\ !\ j) \leq key\ (xs\ !\ (Suc\ (Suc\ m) + k))$ **..**
thus $key\ (?ys\ !\ j) \leq key\ (?ys\ !\ k)$
using O **by** *simp*
qed
ultimately have $?R$ **..**
show $key\ ((ws' @ xs')\ !\ j) \leq key\ ((ws' @ xs')\ !\ k)$
proof (*simp add*: *nth-append not-less P*, (*rule-tac* [!] *conjI*,
rule-tac [!] *impI*)+, *rule-tac* [*3*] *FalseE*)
assume
S: $j < Suc\ (Suc\ m)$ **and**
T: $k < Suc\ (Suc\ m)$
from D [*symmetric*] **show** $key\ (ws'\ !\ j) \leq key\ (ws'\ !\ k)$
proof (*simp add*: *round-suc-suc-def Let-def split*: *if-split-asm*,
(*erule-tac* [*2*] *conjE*)+, *simp-all*)
assume U: $?mi = ?ma$
have $j < length\ ?ws$
using S **and** O **by** *simp*
hence $?ws\ !\ j \in set\ ?ws$

```
    by (rule nth-mem)
  with U have key (?ws ! j) = ?ma
    by (rule mini-maxi-keys-eq)
  moreover have k < length ?ws
    using T and O by simp
  hence ?ws ! k ∈ set ?ws
    by (rule nth-mem)
  with U have key (?ws ! k) = ?ma
    by (rule mini-maxi-keys-eq)
  ultimately show key (?ws ! j) ≤ key (?ws ! k) by simp
next
  assume U: ms' = Suc 0 # ?ms @ [Suc 0]
  hence V: j < (0 # offs (?ms @ Suc 0 # ns') (Suc 0)) ! i
    using L by simp
  hence ∃ i'. i = Suc i'
    by (rule-tac nat.exhaust [of i], simp-all)
  then obtain i' where W: i = Suc i' ..
  have i < Suc (Suc (?k + length ns'))
    using K and U by (simp add: enum-length)
  hence X: i' < Suc (?k + length ns')
    using W by simp
  hence Y: j < Suc (offs (?ms @ Suc 0 # ns') 0 ! i')
    using V and W by (simp add: enum-length offs-suc)
  have (0 # offs (?ms @ Suc 0 # ns') (Suc 0)) ! i ≤ k
    using M and U by simp
  hence Suc (offs (?ms @ Suc 0 # ns') 0 ! i') ≤ k
    using W and X by (simp add: enum-length offs-suc)
  moreover from this have ∃ k'. k = Suc k'
    by (rule-tac nat.exhaust [of k], simp-all)
  then obtain k' where Z: k = Suc k' ..
  ultimately have AA: offs (?ms @ Suc 0 # ns') 0 ! i' ≤ k'
    by simp
  have AB: j ≤ k'
    using Y and AA by simp
  with Z show
   key ((?xmi # map the (fill ?zs (offs ?ms 0) index key m ?mi ?ma) @
     [?xma]) ! j) ≤
    key ((?xmi # map the (fill ?zs (offs ?ms 0) index key m ?mi ?ma) @
     [?xma]) ! k)
  proof (cases j, case-tac [2] j < Suc m, simp-all add: nth-append
   not-less fill-length, rule-tac [1-2] conjI, rule-tac [1-4] impI,
   rule-tac [5] FalseE, simp-all)
    assume k' < m
    hence the (fill ?zs (offs ?ms 0) index key m ?mi ?ma ! k') ∈ set ?ws
      by (rule Q)
    thus ?mi ≤ key (the (fill ?zs (offs ?ms 0) index key m ?mi ?ma ! k'))
      by (rule mini-lb)
  next
    assume m ≤ k'
```

```
  hence k' = m
    using T and Z by simp
  thus ?mi ≤ key ([?xma] ! (k' − m))
    by (simp, rule-tac mini-maxi-keys-le, rule-tac nth-mem [of 0],
     subst O, simp)
next
  fix j'
  have sort-inv key (0, ?ms, map the (fill ?zs (offs ?ms 0) index key
    (length ?zs) ?mi ?ma))
    by (rule fill-sort-inv [OF A B], simp, rule conjI,
     ((rule mini-lb | rule maxi-ub), erule in-set-nthsD)+)
  hence sort-inv key (0, ?ms, map the (fill ?zs (offs ?ms 0) index key
    m ?mi ?ma))
    using O by (simp add: mini-maxi-nths)
  hence ∀ i < ?k. ∀ j < offs ?ms 0 ! i. ∀ k ∈ {offs ?ms 0 ! i..<m}.
    key (the (fill ?zs (offs ?ms 0) index key m ?mi ?ma ! j)) ≤
    key (the (fill ?zs (offs ?ms 0) index key m ?mi ?ma ! k))
    by (simp add: enum-length fill-length)
  moreover assume AC: k' < m
  hence AD: offs (?ms @ Suc 0 # ns') 0 ! i' < m
    using AA by simp
  have AE: i' < ?k
  proof (rule contrapos-pp [OF AD], simp add: not-less offs-append
   nth-append offs-length enum-length)
    have 0 < m
      using AC by simp
    hence 0 < ?k
      by (drule-tac bn-comp-fst-nonzero [OF G], subst (asm) F,
       simp split: nat.split)
    with A have foldl (+) 0 ?ms = length ?zs
      by (rule enum-add, simp, rule-tac conjI,
       ((rule-tac mini-lb | rule-tac maxi-ub), erule-tac in-set-nthsD)+)
    hence foldl (+) 0 ?ms = m
      using O by (simp add: mini-maxi-nths)
    with X show m ≤ (foldl (+) 0 ?ms #
      offs ns' (Suc (foldl (+) 0 ?ms))) ! (i' − ?k)
      by (cases i' − ?k, simp-all, subst offs-base-zero, simp-all)
  qed
  ultimately have ∀ j < offs ?ms 0 ! i'. ∀ k ∈ {offs ?ms 0 ! i'..<m}.
    key (the (fill ?zs (offs ?ms 0) index key m ?mi ?ma ! j)) ≤
    key (the (fill ?zs (offs ?ms 0) index key m ?mi ?ma ! k))
    by simp
  moreover assume j = Suc j'
  with Y and AE have j' < offs ?ms 0 ! i'
    by (simp add: offs-append nth-append offs-length enum-length)
  ultimately have ∀ k ∈ {offs ?ms 0 ! i'..<m}.
    key (the (fill ?zs (offs ?ms 0) index key m ?mi ?ma ! j')) ≤
    key (the (fill ?zs (offs ?ms 0) index key m ?mi ?ma ! k))
    by simp
```

```
      moreover from AA and AE have offs ?ms 0 ! i' ≤ k'
        by (simp add: offs-append nth-append offs-length enum-length)
      hence k' ∈ {offs ?ms 0 ! i'..<m}
        using AC by simp
      ultimately show
       key (the (fill ?zs (offs ?ms 0) index key m ?mi ?ma ! j')) ≤
        key (the (fill ?zs (offs ?ms 0) index key m ?mi ?ma ! k')) ..
    next
      fix j'
      assume j' < m
      hence the (fill ?zs (offs ?ms 0) index key m ?mi ?ma ! j') ∈ set ?ws
        (is ?x ∈ -)
        by (rule Q)
      moreover assume m ≤ k'
      hence k' = m
        using T and Z by simp
      ultimately show key ?x ≤ key ([?xma] ! (k' − m))
        by (simp, rule-tac maxi-ub)
    next
      fix j'
      assume j = Suc j' and m ≤ j'
      hence Suc m ≤ j by simp
      hence Suc (Suc m) ≤ k
        using Z and AB by simp
      thus False
        using T by simp
    qed
  qed
next
  assume
    S: j < Suc (Suc m) and
    T: Suc (Suc m) ≤ k
  from D [symmetric] show key (ws' ! j) ≤ key (xs' ! (k − Suc (Suc m)))
  proof (simp add: round-suc-suc-def Let-def split: if-split-asm,
   rule-tac R [OF S T], cases j, case-tac [2] j < Suc m,
   simp-all add: nth-append not-less fill-length)
    have ?nmi < length ?ws
      using O by (rule-tac mini-less, simp)
    hence ?nmi < Suc (Suc m)
      using O by simp
    thus ?mi ≤ key (xs' ! (k − Suc (Suc m)))
      using T by (rule R)
  next
    fix j'
    assume j' < m
    hence U: the (fill ?zs (offs ?ms 0) index key m ?mi ?ma ! j') ∈ set ?ws
      by (rule Q)
    have ∃ p < Suc (Suc m). ?ws ! p =
      the (fill ?zs (offs ?ms 0) index key m ?mi ?ma ! j')
```

```
    using O and U by (simp add: in-set-conv-nth)
   then obtain p where V: p < Suc (Suc m) ∧ ?ws ! p =
    the (fill ?zs (offs ?ms 0) index key m ?mi ?ma ! j') ..
   hence key (?ws ! p) ≤ key (xs' ! (k − Suc (Suc m)))
    using T by (rule-tac R, simp)
   thus key (the (fill ?zs (offs ?ms 0) index key m ?mi ?ma ! j')) ≤
    key (xs' ! (k − Suc (Suc m)))
    using V by simp
  next
   fix j'
   assume j = Suc j' and m ≤ j'
   moreover from this have Suc m ≤ j by simp
   hence j = Suc m
    using S by simp
   ultimately have j' = m by simp
   thus key ([?xma] ! (j' − m)) ≤ key (xs' ! (k − Suc (Suc m)))
   proof simp
    have ?nma < length ?ws
     using O by (rule-tac maxi-less, simp)
    hence ?nma < Suc (Suc m)
     using O by simp
    thus ?ma ≤ key (xs' ! (k − Suc (Suc m)))
     using T by (rule R)
   qed
  qed
next
  assume k < Suc (Suc m) and Suc (Suc m) ≤ j
  hence k < j by simp
  moreover have j < k
   using L and M by simp
  ultimately show False by simp
next
  assume
   S: Suc (Suc m) ≤ j and
   T: Suc (Suc m) ≤ k
  have U: 0 < length ns' ∧ 0 < length xs'
  proof (rule ccontr, simp, erule disjE)
   have bn-inv p q (v, ns, ?ys)
    using C by simp
   moreover have add-inv (n − Suc (Suc m)) (v, ns, ?ys)
    using H and I by simp
   ultimately have add-inv (n − Suc (Suc m)) (?t r' v)
    by (rule round-add-inv [OF A])
   hence length xs' = foldl (+) 0 ns'
    using E by simp
   moreover assume ns' = []
   ultimately have length xs' = 0 by simp
   hence k < Suc (Suc m)
    using N and P by simp
```

```
  thus False
    using T by simp
next
  assume xs' = []
  hence k < Suc (Suc m)
    using N and P by simp
  thus False
    using T by simp
qed
hence V: i − length ms' < length ns'
  using K by arith
hence W: ∀ j < offs ns' 0 ! (i − length ms').
  ∀ k ∈ {offs ns' 0 ! (i − length ms')..<length xs'}.
    key (xs' ! j) ≤ key (xs' ! k)
  using ‹?R› by simp
from D [symmetric] have X: foldl (+) 0 ms' = Suc (Suc m)
proof (simp add: round-suc-suc-def Let-def add-replicate add-suc
 split: if-split-asm, cases m = 0)
  case True
  hence ?k = 0 by simp
  hence length ?ms = 0
    by (simp add: enum-length)
  thus foldl (+) 0 ?ms = m
    using True by simp
next
  case False
  hence 0 < ?k
    by (simp, drule-tac bn-comp-fst-nonzero [OF G], subst (asm) F,
     simp split: nat.split)
  with A have foldl (+) 0 ?ms = length ?zs
    by (rule enum-add, simp, rule-tac conjI,
     ((rule-tac mini-lb | rule-tac maxi-ub), erule-tac in-set-nthsD)+)
  thus foldl (+) 0 ?ms = m
    using O by (simp add: mini-maxi-nths)
qed
have length ms' < i
proof (rule ccontr, simp add: not-less)
  assume i ≤ length ms'
  moreover have length ms' < length (ms' @ ns')
    using U by simp
  ultimately have offs (ms' @ ns') 0 ! i ≤
    offs (ms' @ ns') 0 ! (length ms')
    by (rule offs-mono)
  also have ... = Suc (Suc m)
    using U and X by (simp add: offs-append nth-append offs-length,
     cases ns', simp-all)
  finally have offs (ms' @ ns') 0 ! i ≤ Suc (Suc m) .
  hence j < Suc (Suc m)
    using L by simp
```

```
      thus False
        using S by simp
    qed
    hence Y: offs (ms' @ ns') 0 ! i =
      Suc (Suc m) + offs ns' 0 ! (i − length ms')
      using X by (simp add: offs-append nth-append offs-length,
       subst offs-base-zero [OF V], simp)
    hence j < Suc (Suc m) + offs ns' 0 ! (i − length ms')
      using L by simp
    moreover from this have 0 < offs ns' 0 ! (i − length ms')
      using S by (rule-tac ccontr, simp)
    ultimately have j − Suc (Suc m) < offs ns' 0 ! (i − length ms')
      by simp
    hence Z: ∀ k ∈ {offs ns' 0 ! (i − length ms')..<length xs'}.
      key (xs' ! (j − Suc (Suc m))) ≤ key (xs' ! k)
      using W by simp
    have offs ns' 0 ! (i − length ms') ≤ k − Suc (Suc m)
      using M and Y by simp
    moreover have k − Suc (Suc m) < length xs'
      using N and P and U by arith
    ultimately have k − Suc (Suc m) ∈
      {offs ns' 0 ! (i − length ms')..<length xs'}
      by simp
    with Z show key (xs' ! (j − Suc (Suc m))) ≤
      key (xs' ! (k − Suc (Suc m))) ..
  qed
 qed
qed

lemma gcsort-sort-inv:
 assumes
   A: index-less index key and
   B: index-mono index key and
   C: add-inv n t and
   D: n ≤ p
 shows ⟦t' ∈ gcsort-set index key p t; sort-inv key t⟧ ⟹
   sort-inv key t'
by (erule gcsort-set.induct, simp, drule gcsort-add-inv [OF A - C D],
 rule round-sort-inv [OF A B], simp-all del: bn-inv.simps, erule conjE,
 frule sym, erule subst, rule bn-inv-intro, insert D, simp)
```

The only remaining task is to address step 10 of the proof method, which is done by proving theorem *gcsort-sorted*. It holds under the conditions that the objects' number is not larger than the counters' upper bound and function *index* satisfies both predicates *index-less* and *index-mono*, and states that function *gcsort* is successful in sorting the input objects' list.

theorem *gcsort-sorted*:
assumes
A: *index-less index key* **and**
B: *index-mono index key* **and**
C: *length xs* $\leq$ *p*
shows *sorted* (*map key* (*gcsort index key p xs*))
proof −
let *?t* = (*0*, [*length xs*], *xs*)
have *sort-inv key* (*gcsort-aux index key p ?t*)
by (*rule gcsort-sort-inv* [*OF A B - C*], *rule gcsort-add-input*,
rule gcsort-aux-set, *rule gcsort-sort-input*)
moreover have *add-inv* (*length xs*) (*gcsort-aux index key p ?t*)
by (*rule gcsort-add-inv* [*OF A - - C*],
rule gcsort-aux-set, *rule gcsort-add-input*)
ultimately have *sorted* (*map key* (*gcsort-out* (*gcsort-aux index key p ?t*)))
by (*rule gcsort-sort-intro*, *simp add*: *gcsort-sort-form*)
moreover have *?t* = *gcsort-in xs*
by (*simp add*: *gcsort-in-def*)
ultimately show *?thesis*
by (*simp add*: *gcsort-def*)
qed

end

4 Proof of algorithm's stability

theory *Stability*
imports *Sorting*
begin

In this section, it is formally proven that GCsort is *stable*, viz. that the sublist of the output of function *gcsort* built by picking out the objects having a given key matches the sublist of the input objects' list built in the same way.

Here below, steps 5, 6, and 7 of the proof method are accomplished. Particularly, *stab-inv* is the predicate that will be shown to be invariant over inductive set *gcsort-set*.

fun *stab-inv* :: (*'b* ⇒ *'a list*) ⇒ (*'a* ⇒ *'b*) ⇒ *nat* × *nat list* × *'a list* ⇒
bool **where**
stab-inv f key (*u*, *ns*, *xs*) = (∀ *k*. [*x*←*xs*. *key x* = *k*] = *f k*)

lemma *gcsort-stab-input*:
stab-inv (λ*k*. [*x*←*xs*. *key x* = *k*]) *key* (*0*, [*length xs*], *xs*)
by *simp*

lemma *gcsort-stab-intro*:
stab-inv f key t $\Longrightarrow$ [*x*←*gcsort-out t*. *key x* = *k*] = *f k*
by (*cases t*, *simp add*: *gcsort-out-def*)

In what follows, step 9 of the proof method is accomplished.

First, lemma *fill-offs-enum-stable* proves that function *fill*, if its input offsets' list is computed via the composition of functions *offs* and *enum*, does not modify the sublist of its input objects' list formed by the objects having a given key. Moreover, lemmas *mini-stable* and *maxi-stable* prove that the extraction of the leftmost minimum and the rightmost maximum from an objects' list through functions *mini* and *maxi* is endowed with the same property.

These lemmas are then used to prove lemma *gcsort-stab-inv*, which states that the sublist of the objects having a given key within the objects' list is still the same after any recursive round.

lemma *fill-stable* [*rule-format*]:
assumes
A: *index-less index key* **and**
B: *index-same index key*
shows
(∀ *x* ∈ *set xs*. *key x* ∈ {*mi*..*ma*}) ⟶
ns ≠ [] ⟶
offs-pred ns ub xs index key mi ma ⟶
map the [*w*←*fill xs ns index key ub mi ma*. ∃ *x*. *w* = *Some x* ∧ *key x* = *k*] =
[*x*←*xs*. *k* = *key x*]
proof (*induction xs arbitrary*: *ns*, *simp-all add*: *Let-def*, *rule conjI*,
(*rule-tac* [!] *impI*)+, (*erule-tac* [!] *conjE*)+, *simp-all*)
fix *x xs* **and** *ns* :: *nat list*
let *?i* = *index key x* (*length ns*) *mi ma*
let *?ns′* = *ns*[*?i* := *Suc* (*ns* ! *?i*)]
let *?ws* = [*w*←*fill xs ?ns′ index key ub mi ma*.
∃ *y*. *w* = *Some y* ∧ *key y* = *key x*]
let *?ws′* = [*w*←(*fill xs ?ns′ index key ub mi ma*)[*ns* ! *?i* := *Some x*].
∃ *y*. *w* = *Some y* ∧ *key y* = *key x*]
assume
C: ∀ *x* ∈ *set xs*. *mi* ≤ *key x* ∧ *key x* ≤ *ma* **and**
D: *mi* ≤ *key x* **and**
E: *key x* ≤ *ma* **and**
F: *ns* ≠ [] **and**
G: *offs-pred ns ub* (*x* # *xs*) *index key mi ma*
have *H*: *?i* < *length ns*
using *A* **and** *D* **and** *E* **and** *F* **by** (*simp add*: *index-less-def*)
assume ⋀*ns*. *ns* ≠ [] ⟶ *offs-pred ns ub xs index key mi ma* ⟶
map the [*w*←*fill xs ns index key ub mi ma*.
∃ *y*. *w* = *Some y* ∧ *key y* = *key x*] =

$[y \leftarrow xs.\ key\ x = key\ y]$
moreover have I: *offs-pred ?ns′ ub xs index key mi ma*
using G **and** H **by** (*rule-tac offs-pred-cons*, *simp-all*)
ultimately have J: *map the ?ws* $= [y \leftarrow xs.\ key\ x = key\ y]$
using F **by** *simp*
have *ns* ! *?i* + *offs-num* (*length ns*) (*x* # *xs*) *index key mi ma ?i* $\leq$ *ub*
using G **and** H **by** (*rule offs-pred-ub*, *simp add*: *offs-num-cons*)
hence K: *ns* ! *?i* $<$ *ub*
by (*simp add*: *offs-num-cons*)
moreover from *this* **have**
L: (*fill xs ?ns′ index key ub mi ma*)[*ns* ! *?i* := *Some x*] ! (*ns* ! *?i*) = *Some x*
by (*subst nth-list-update-eq*, *simp-all add*: *fill-length*)
ultimately have *0* $<$ *length ?ws′*
by (*simp add*: *length-filter-conv-card card-gt-0-iff*,
rule-tac exI [**where** *x* = *ns* ! *?i*], *simp add*: *fill-length*)
hence $\exists\, w\ ws.$ *?ws′* $= w \#\ ws$
by (*rule-tac list.exhaust* [*of ?ws′*], *simp-all*)
then obtain *w* **and** *ws* **where** *?ws′* $= w \#\ ws$ **by** *blast*
hence $\exists\, us\ vs\ y.$
(*fill xs ?ns′ index key ub mi ma*)[*ns* ! *?i* := *Some x*] = *us* @ *w* # *vs* $\wedge$
$(\forall\, u \in set\ us.\ \forall\, y.\ u = Some\ y \longrightarrow key\ y \neq key\ x) \wedge$
$w = Some\ y \wedge key\ y = key\ x \wedge$
$ws = [w \leftarrow vs.\ \exists\, y.\ w = Some\ y \wedge key\ y = key\ x]$
(**is** $\exists\, us\ vs\ y.$ *?P us vs y*)
by (*simp add*: *filter-eq-Cons-iff*, *blast*)
then obtain *us* **and** *vs* **and** *y* **where** M: *?P us vs y* **by** *blast*
let *?A* = $\{i.\ i < length\ ns \wedge 0 < offs\text{-}num\ (length\ ns)\ xs\ index\ key\ mi\ ma\ i \wedge$
ns ! *i* $\leq$ *length us*$\}$
have *length us* = *ns* ! *?i*
proof (*rule ccontr*, *erule neqE*, *cases ?A* = {},
cases 0 $<$ *offs-num* (*length ns*) *xs index key mi ma 0*, *simp-all only*: *not-gr0*)
assume
N: *length us* $<$ *ns* ! *?i* **and**
O: *?A* = {} **and**
P: *0* $<$ *offs-num* (*length ns*) *xs index key mi ma 0*
have *length us* $<$ *ns* ! *0*
proof (*rule ccontr*, *simp add*: *not-less*)
assume *ns* ! *0* $\leq$ *length us*
with F **and** P **have** *0* $\in$ *?A* **by** *simp*
with O **show** *False* **by** *blast*
qed
hence *length us* $<$ *?ns′* ! *0*
using F **by** (*cases ?i*, *simp-all*)
hence *offs-none ?ns′ ub xs index key mi ma* (*length us*)
using P **by** (*simp add*: *offs-none-def*)
hence *fill xs ?ns′ index key ub mi ma* ! (*length us*) = *None*
using C **and** F **and** I **by** (*rule-tac fill-none* [*OF A*], *simp-all*)
hence (*fill xs ?ns′ index key ub mi ma*)[*ns* ! *?i* := *Some x*] ! (*length us*) = *None*
using N **by** *simp*

```
  thus False
    using M by simp
next
  assume
    N: length us < ns ! ?i and
    O: ?A = {} and
    P: offs-num (length ns) xs index key mi ma 0 = 0
  from K and N have length us < offs-next ?ns' ub xs index key mi ma 0
  proof (rule-tac ccontr, simp only: not-less offs-next-def split: if-split-asm)
    assume offs-set-next ?ns' xs index key mi ma 0 ≠ {}
      (is ?B ≠ -)
    hence Min ?B ∈ ?B
      by (rule-tac Min-in, simp)
    moreover assume ?ns' ! Min ?B ≤ length us
    hence ns ! Min ?B ≤ length us
      using H by (cases Min ?B = ?i, simp-all)
    ultimately have Min ?B ∈ ?A by simp
    with O show False by blast
  qed
  hence offs-none ?ns' ub xs index key mi ma (length us)
    using P by (simp add: offs-none-def)
  hence fill xs ?ns' index key ub mi ma ! (length us) = None
    using C and F and I by (rule-tac fill-none [OF A], simp-all)
 hence (fill xs ?ns' index key ub mi ma)[ns ! ?i := Some x] ! (length us) = None
    using N by simp
  thus False
    using M by simp
next
  assume N: length us < ns ! ?i
  assume ?A ≠ {}
  hence Max ?A ∈ ?A
    by (rule-tac Max-in, simp)
  moreover from this have O: Max ?A ≠ ?i
    using N by (rule-tac notI, simp)
  ultimately have index key y (length ?ns') mi ma = Max ?A
  using C proof (rule-tac fill-index [OF A - I, where j = length us], simp-all,
   rule-tac ccontr, simp only: not-less not-le offs-next-def split: if-split-asm)
    assume ub ≤ length us
    with N have ub < ns ! ?i by simp
    with K show False by simp
  next
    let ?B = offs-set-next ?ns' xs index key mi ma (Max ?A)
    assume ?ns' ! Min ?B ≤ length us
    hence ns ! Min ?B ≤ length us
      using H by (cases Min ?B = ?i, simp-all)
    moreover assume ?B ≠ {}
    hence P: Min ?B ∈ ?B
      by (rule-tac Min-in, simp)
    ultimately have Min ?B ∈ ?A by simp
```

```
    hence Min ?B ≤ Max ?A
      by (rule-tac Max-ge, simp)
    thus False
      using P by simp
  next
    have fill xs ?ns' index key ub mi ma ! length us =
      (fill xs ?ns' index key ub mi ma)[ns ! ?i := Some x] ! length us
      using N by simp
    thus fill xs ?ns' index key ub mi ma ! length us = Some y
      using M by simp
  qed
  moreover have index key y (length ?ns') mi ma = ?i
    using B and D and E and M by (cases y = x, simp-all add:
     index-same-def)
  ultimately show False
    using O by simp
next
  assume N: ns ! ?i < length us
  hence (fill xs ?ns' index key ub mi ma)[ns ! ?i := Some x] ! (ns ! ?i) =
    us ! (ns ! ?i)
    using M by (simp add: nth-append)
  hence (fill xs ?ns' index key ub mi ma)[ns ! ?i := Some x] ! (ns ! ?i) ∈ set us
    using N by simp
  thus False
    using L and M by blast
qed
hence take (ns ! ?i) ((fill xs ?ns' index key ub mi ma)[ns ! ?i := Some x]) = us
  using M by simp
hence N: take (ns ! ?i) (fill xs ?ns' index key ub mi ma) = us by simp
have fill xs ?ns' index key ub mi ma =
  take (ns ! ?i) (fill xs ?ns' index key ub mi ma) @
  fill xs ?ns' index key ub mi ma ! (ns ! ?i) #
  drop (Suc (ns ! ?i)) (fill xs ?ns' index key ub mi ma)
  (is - = ?ts @ - # ?ds)
  using K by (rule-tac id-take-nth-drop, simp add: fill-length)
moreover have fill xs ?ns' index key ub mi ma ! (ns ! ?i) = None
  using C and D and E and F and G by (rule-tac fill-index-none [OF A],
   simp-all)
ultimately have O: fill xs ?ns' index key ub mi ma = us @ None # ?ds
  using N by simp
have (fill xs ?ns' index key ub mi ma)[ns ! ?i := Some x] =
  ?ts @ Some x # ?ds
  using K by (rule-tac upd-conv-take-nth-drop, simp add: fill-length)
hence (fill xs ?ns' index key ub mi ma)[ns ! ?i := Some x] =
  us @ Some x # ?ds
  using N by simp
hence ?ws' = Some x # [w←?ds. ∃ y. w = Some y ∧ key y = key x]
  using M by simp
also have ... = Some x # ?ws
```

using M **by** (*subst* (*2*) O, *simp*)
finally show *map the ?ws′* = x # [y←*xs*. *key x* = *key y*]
using J **by** *simp*
next
fix x *xs* **and** *ns* :: *nat list*
let *?i* = *index key x* (*length ns*) *mi ma*
let *?ns′* = *ns*[*?i* := *Suc* (*ns* ! *?i*)]
let *?ws* = [w←*fill xs ?ns′ index key ub mi ma*. ∃ y. w = *Some y* ∧ *key y* = k]
let *?ws′* = [w←(*fill xs ?ns′ index key ub mi ma*)[*ns* ! *?i* := *Some x*].
∃ y. w = *Some y* ∧ *key y* = k]
assume
C: ∀ x ∈ *set xs*. *mi* ≤ *key x* ∧ *key x* ≤ *ma* **and**
D: *mi* ≤ *key x* **and**
E: *key x* ≤ *ma* **and**
F: *ns* ≠ [] **and**
G: *offs-pred ns ub* (x # *xs*) *index key mi ma* **and**
H: k ≠ *key x*
have I: *?i* < *length ns*
using A **and** D **and** E **and** F **by** (*simp add*: *index-less-def*)
assume ⋀*ns*. *ns* ≠ [] ⟶ *offs-pred ns ub xs index key mi ma* ⟶
map the [w←*fill xs ns index key ub mi ma*.
∃ y. w = *Some y* ∧ *key y* = k] =
[y←*xs*. k = *key y*]
moreover have *offs-pred ?ns′ ub xs index key mi ma*
using G **and** I **by** (*rule-tac offs-pred-cons*, *simp-all*)
ultimately have J: *map the ?ws* = [y←*xs*. k = *key y*]
using F **by** *simp*
have *ns* ! *?i* + *offs-num* (*length ns*) (x # *xs*) *index key mi ma ?i* ≤ *ub*
using G **and** I **by** (*rule offs-pred-ub*, *simp add*: *offs-num-cons*)
hence K: *ns* ! *?i* < *ub*
by (*simp add*: *offs-num-cons*)
have *fill xs ?ns′ index key ub mi ma* =
take (*ns* ! *?i*) (*fill xs ?ns′ index key ub mi ma*) @
fill xs ?ns′ index key ub mi ma ! (*ns* ! *?i*) #
drop (*Suc* (*ns* ! *?i*)) (*fill xs ?ns′ index key ub mi ma*)
(**is** - = *?ts* @ - # *?ds*)
using K **by** (*rule-tac id-take-nth-drop*, *simp add*: *fill-length*)
moreover have *fill xs ?ns′ index key ub mi ma* ! (*ns* ! *?i*) = *None*
using C **and** D **and** E **and** F **and** G **by** (*rule-tac fill-index-none* [*OF A*],
simp-all)
ultimately have L: *fill xs ?ns′ index key ub mi ma* = *?ts* @ *None* # *?ds*
by *simp*
have (*fill xs ?ns′ index key ub mi ma*)[*ns* ! *?i* := *Some x*] =
?ts @ *Some x* # *?ds*
using K **by** (*rule-tac upd-conv-take-nth-drop*, *simp add*: *fill-length*)
moreover have *?ws* = [w←*?ts*. ∃ y. w = *Some y* ∧ *key y* = k] @
[w←*?ds*. ∃ y. w = *Some y* ∧ *key y* = k]
by (*subst L*, *simp*)
ultimately have *?ws′* = *?ws*

```
    using H by simp
  thus map the ?ws' = [y←xs. k = key y]
    using J by simp
qed

lemma fill-offs-enum-stable [rule-format]:
  assumes
    A: index-less index key and
    B: index-same index key
  shows
   ∀ x ∈ set xs. key x ∈ {mi..ma} ⟹
    0 < n ⟹
      [x←map the (fill xs (offs (enum xs index key n mi ma) 0)
        index key (length xs) mi ma). key x = k] = [x←xs. k = key x]
  (is - ⟹ - ⟹ [-←map the ?ys. -] = -
   is - ⟹ - ⟹ [-←map the (fill - ?ns - - - - -). -] = -)
proof (subst fill-stable [symmetric, OF A B, of xs mi ma ?ns], simp,
 simp only: length-greater-0-conv [symmetric] offs-length enum-length,
 rule offs-enum-pred [OF A], simp, subst filter-map,
 simp add: filter-eq-nths fill-length)
  assume
    C: ∀ x ∈ set xs. mi ≤ key x ∧ key x ≤ ma and
    D: 0 < n
  have {i. i < length xs ∧ key (the (?ys ! i)) = k} =
    {i. i < length xs ∧ (∃ x. ?ys ! i = Some x ∧ key x = k)}
    (is ?A = ?B)
  proof (rule set-eqI, rule iffI, simp-all, erule-tac [!] conjE, erule-tac [2] exE,
   erule-tac [2] conjE, simp-all)
    fix i
    assume E: i < length xs and F: key (the (?ys ! i)) = k
    have ∃ x. ?ys ! i = Some x
    proof (cases ?ys ! i, simp-all)
      assume ?ys ! i = None
      moreover have ?ys ! i ∈ set ?ys
        using E by (rule-tac nth-mem, simp add: fill-length)
      ultimately have None ∈ set ?ys by simp
      moreover have count (mset ?ys) None = 0
        using C and D by (rule-tac fill-offs-enum-count-none [OF A], simp)
      ultimately show False by simp
    qed
    then obtain x where ?ys ! i = Some x ..
    moreover from this have key x = k
      using F by simp
    ultimately show ∃ x. ?ys ! i = Some x ∧ key x = k by simp
  qed
  thus map the (nths ?ys ?A) = map the (nths ?ys ?B) by simp
qed

lemma mini-first [rule-format]:
```

xs $\neq$ [] $\longrightarrow$ *i* < *mini xs key* $\longrightarrow$
key (*xs* ! *mini xs key*) < *key* (*xs* ! *i*)
by (*induction xs arbitrary*: *i*, *simp-all add*: *Let-def*, (*rule impI*)+,
erule conjE, *simp add*: *not-le nth-Cons split*: *nat.split*)

lemma *maxi-last* [*rule-format*]:
xs $\neq$ [] $\longrightarrow$ *maxi xs key* < *i* $\longrightarrow$ *i* < *length xs* $\longrightarrow$
key (*xs* ! *i*) < *key* (*xs* ! *maxi xs key*)
by (*induction xs arbitrary*: *i*, *simp-all add*: *Let-def*, (*rule impI*)+,
rule le-less-trans, *rule maxi-ub*, *rule nth-mem*, *simp*)

lemma *nths-range*:
nths xs A = *nths xs* (*A* $\cap$ {..<*length xs*})
proof (*induction xs arbitrary*: *A*, *simp-all add*: *nths-Cons*)
fix *xs* :: *'a list* **and** *A*
assume $\bigwedge$*A*. *nths xs A* = *nths xs* (*A* $\cap$ {..<*length xs*})
hence *nths xs* {*i*. *Suc i* $\in$ *A*} = *nths xs* ({*i*. *Suc i* $\in$ *A*} $\cap$ {..<*length xs*}) **.**
moreover have
{*i*. *Suc i* $\in$ *A*} $\cap$ {..<*length xs*} = {*i*. *Suc i* $\in$ *A* $\wedge$ *i* < *length xs*}
by *blast*
ultimately show
nths xs {*i*. *Suc i* $\in$ *A*} = *nths xs* {*i*. *Suc i* $\in$ *A* $\wedge$ *i* < *length xs*}
by *simp*
qed

lemma *filter-nths-diff*:
assumes
A: *i* < *length xs* **and**
B: $\neg$ *P* (*xs* ! *i*)
shows [*x*$\leftarrow$*nths xs* (*A* − {*i*}). *P x*] = [*x*$\leftarrow$*nths xs A*. *P x*]
proof (*cases i* $\in$ *A*, *simp-all*)
case *True*
have *C*: *xs* = *take i xs* @ *xs* ! *i* # *drop* (*Suc i*) *xs*
(**is** - = *?ts* @ - # *?ds*)
using *A* **by** (*rule id-take-nth-drop*)
have *nths xs A* = *nths ?ts A* @ *nths* (*xs* ! *i* # *?ds*) {*j*. *j* + *length ?ts* $\in$ *A*}
by (*subst C*, *simp add*: *nths-append*)
moreover have *D*: *length ?ts* = *i*
using *A* **by** *simp*
ultimately have *E*: *nths xs A* = *nths ?ts* (*A* $\cap$ {..<*i*}) @ *xs* ! *i* #
nths ?ds {*j*. *Suc j* + *i* $\in$ *A*}
(**is** - = *?vs* @ - # *?ws*)
using *True* **by** (*simp add*: *nths-Cons*, *subst nths-range*, *simp*)
have *nths xs* (*A* − {*i*}) = *nths ?ts* (*A* − {*i*}) @
nths (*xs* ! *i* # *?ds*) {*j*. *j* + *length ?ts* $\in$ *A* − {*i*}}
by (*subst C*, *simp add*: *nths-append*)
moreover have (*A* − {*i*}) $\cap$ {..<*i*} = *A* $\cap$ {..<*i*}
by (*rule set-eqI*, *rule iffI*, *simp-all*)
ultimately have *nths xs* (*A* − {*i*}) = *?vs* @ *?ws*

```
    using D by (simp add: nths-Cons, subst nths-range, simp)
  thus ?thesis
    using B and E by simp
qed

lemma mini-stable:
  assumes
    A: xs ≠ [] and
    B: mini xs key ∈ A
      (is ?nmi ∈ -)
  shows [x←[xs ! ?nmi] @ nths xs (A − {?nmi}). key x = k] =
    [x←nths xs A. key x = k]
    (is [x←[?xmi] @ -. -] = -)
proof (simp, rule conjI, rule-tac [!] impI, rule-tac [2] filter-nths-diff,
 rule-tac [2] mini-less, simp-all add: A)
  assume C: key ?xmi = k
  moreover have ?nmi < length xs
    using A by (rule-tac mini-less, simp)
  ultimately have 0 < length [x←xs. key x = k]
    by (simp add: length-filter-conv-card card-gt-0-iff, rule-tac exI, rule-tac conjI)
  hence ∃ y ys. [x←xs. key x = k] = y # ys
    by (rule-tac list.exhaust [of [x←xs. key x = k]], simp-all)
  then obtain y and ys where [x←xs. key x = k] = y # ys by blast
  hence ∃ us vs. xs = us @ y # vs ∧
    (∀ u ∈ set us. key u ≠ k) ∧ key y = k ∧ ys = [x←vs. key x = k]
    (is ∃ us vs. ?P us vs)
    by (simp add: filter-eq-Cons-iff)
  then obtain us and vs where D: ?P us vs by blast
  have E: length us = ?nmi
  proof (rule ccontr, erule neqE)
    assume length us < ?nmi
    with A have key ?xmi < key (xs ! length us)
      by (rule mini-first)
    also have ... = k
      using D by simp
    finally show False
      using C by simp
  next
    assume F: ?nmi < length us
    hence ?xmi = us ! ?nmi
      using D by (simp add: nth-append)
    hence ?xmi ∈ set us
      using F by simp
    thus False
      using C and D by simp
  qed
  moreover have xs ! length us = y
    using D by simp
  ultimately have F: ?xmi = y by simp
```

have *nths xs A = nths us A @ nths (y # vs) {i. i + ?nmi ∈ A}*
using *D* **and** *E* **by** (*simp add*: *nths-append*)
also have *... = nths us A @ y # nths vs {i. Suc i + ?nmi ∈ A}*
(**is** - = - @ - # *?ws*)
using *B* **by** (*simp add*: *nths-Cons*)
finally have *G*: *[x←nths xs A. key x = k] = y # [x←?ws. key x = k]*
using *D* **by** (*simp add*: *filter-empty-conv*, *rule-tac ballI*,
drule-tac in-set-nthsD, *simp*)
have *nths xs (A − {?nmi}) = nths us (A − {?nmi}) @*
nths (y # vs) {i. i + ?nmi ∈ A − {?nmi}}
using *D* **and** *E* **by** (*simp add*: *nths-append*)
also have *... = nths us (A − {?nmi}) @ ?ws*
by (*simp add*: *nths-Cons*)
finally have *H*: *[x←nths xs (A − {?nmi}). key x = k] = [x←?ws. key x = k]*
using *D* **by** (*simp add*: *filter-empty-conv*, *rule-tac ballI*,
drule-tac in-set-nthsD, *simp*)
show *?xmi # [x←nths xs (A − {?nmi}). key x = k] =*
[x←nths xs A. key x = k]
using *F* **and** *G* **and** *H* **by** *simp*
qed

lemma *maxi-stable*:
assumes
A: *xs ≠ []* **and**
B: *maxi xs key ∈ A*
(**is** *?nma ∈* -)
shows *[x←nths xs (A − {?nma}) @ [xs ! ?nma]. key x = k] =*
[x←nths xs A. key x = k]
(**is** *[x←- @ [?xma]. -] =* -)
proof (*simp*, *rule conjI*, *rule-tac* [!] *impI*, *rule-tac* [*2*] *filter-nths-diff*,
rule-tac [*2*] *maxi-less*, *simp-all add*: *A*)
assume *C*: *key ?xma = k*
moreover have *D*: *?nma < length xs*
using *A* **by** (*rule-tac maxi-less*, *simp*)
ultimately have *0 < length [x←rev xs. key x = k]*
by (*simp add*: *length-filter-conv-card card-gt-0-iff*,
rule-tac exI [**where** *x = length xs − Suc ?nma*], *simp add*: *rev-nth*)
hence *∃ y ys. [x←rev xs. key x = k] = y # ys*
by (*rule-tac list.exhaust* [*of* [*x←rev xs. key x = k*]], *simp-all*)
then obtain *y* **and** *ys* **where** *[x←rev xs. key x = k] = y # ys* **by** *blast*
hence *∃ us vs. rev xs = us @ y # vs ∧*
(∀ u ∈ set us. key u ≠ k) ∧ key y = k ∧ ys = [x←vs. key x = k]
(**is** *∃ us vs. ?P us vs*)
by (*simp add*: *filter-eq-Cons-iff*)
then obtain *us* **and** *vs* **where** *E*: *?P us vs* **by** *blast*
hence *F*: *xs = rev vs @ y # rev us*
by (*simp add*: *rev-swap*)
have *G*: *length us = length xs − Suc ?nma*
proof (*rule ccontr*, *erule neqE*)

assume *length us* < *length xs* − *Suc ?nma*
hence *?nma* < *length xs* − *Suc* (*length us*) **by** *simp*
moreover have *length xs* − *Suc* (*length us*) < *length xs*
using *D* **by** *simp*
ultimately have *key* (*xs* ! (*length xs* − *Suc* (*length us*))) < *key ?xma*
by (*rule maxi-last* [*OF A*])
moreover have *length us* < *length xs*
using *F* **by** *simp*
ultimately have *key* (*rev xs* ! *length us*) < *key ?xma*
by (*simp add*: *rev-nth*)
moreover have *key* (*rev xs* ! *length us*) = *k*
using *E* **by** *simp*
ultimately show *False*
using *C* **by** *simp*
next
assume *H*: *length xs* − *Suc ?nma* < *length us*
hence *rev xs* ! (*length xs* − *Suc ?nma*) = *us* ! (*length xs* − *Suc ?nma*)
using *E* **by** (*simp add*: *nth-append*)
hence *rev xs* ! (*length xs* − *Suc ?nma*) ∈ *set us*
using *H* **by** *simp*
hence *?xma* ∈ *set us*
using *D* **by** (*simp add*: *rev-nth*)
thus *False*
using *C* **and** *E* **by** *simp*
qed
moreover have *rev xs* ! *length us* = *y*
using *E* **by** *simp*
ultimately have *H*: *?xma* = *y*
using *D* **by** (*simp add*: *rev-nth*)
have *length xs* = *length us* + *Suc ?nma*
using *D* **and** *G* **by** *simp*
hence *I*: *length vs* = *?nma*
using *F* **by** *simp*
hence *nths xs A* = *nths* (*rev vs*) *A* @ *nths* (*y* # *rev us*) {*i*. *i* + *?nma* ∈ *A*}
using *F* **by** (*simp add*: *nths-append*)
also have . . . = *nths* (*rev vs*) (*A* ∩ {..<*?nma*}) @ *y* #
nths (*rev us*) {*i*. *Suc i* + *?nma* ∈ *A*}
(**is** - = *?ws* @ - # -)
using *B* **and** *I* **by** (*simp add*: *nths-Cons*, *subst nths-range*, *simp*)
finally have *J*: [*x*←*nths xs A*. *key x* = *k*] = [*x*←*?ws*. *key x* = *k*] @ [*y*]
using *E* **by** (*simp add*: *filter-empty-conv*, *rule-tac ballI*,
drule-tac in-set-nthsD, *simp*)
have *nths xs* (*A* − {*?nma*}) = *nths* (*rev vs*) (*A* − {*?nma*}) @
nths (*y* # *rev us*) {*i*. *i* + *?nma* ∈ *A* − {*?nma*}}
using *F* **and** *I* **by** (*simp add*: *nths-append*)
hence *nths xs* (*A* − {*?nma*}) = *nths* (*rev vs*) ((*A* − {*?nma*}) ∩ {..<*?nma*}) @
nths (*rev us*) {*i*. *Suc i* + *?nma* ∈ *A*}
using *I* **by** (*simp add*: *nths-Cons*, *subst nths-range*, *simp*)
moreover have (*A* − {*?nma*}) ∩ {..<*?nma*} = *A* ∩ {..<*?nma*}

```
    by (rule set-eqI, rule iffI, simp-all)
  ultimately have K: [x←nths xs (A − {?nma}). key x = k] =
    [x←?ws. key x = k]
    using E by (simp add: filter-empty-conv, rule-tac ballI,
     drule-tac in-set-nthsD, simp)
  show [x←nths xs (A − {?nma}). key x = k] @ [?xma] =
    [x←nths xs A. key x = k]
    using H and J and K by simp
qed

lemma round-stab-inv [rule-format]:
 index-less index key ⟶ index-same index key ⟶ bn-inv p q t ⟶
    add-inv n t ⟶ stab-inv f key t ⟶ stab-inv f key (round index key p q r t)
proof (induction index key p q r t arbitrary: n f rule: round.induct,
 (rule-tac [!] impI)+, simp, simp, simp-all only: simp-thms)
  fix index p q r u ns xs n f and key :: 'a ⇒ 'b
  let ?t = round index key p q r (u, ns, tl xs)
  assume
   ⋀n f. bn-inv p q (u, ns, tl xs) ⟶ add-inv n (u, ns, tl xs) ⟶
      stab-inv f key (u, ns, tl xs) ⟶ stab-inv f key ?t and
   bn-inv p q (u, Suc 0 # ns, xs) and
   add-inv n (u, Suc 0 # ns, xs) and
   stab-inv f key (u, Suc 0 # ns, xs)
  thus stab-inv f key (round index key p q r (u, Suc 0 # ns, xs))
  proof (cases ?t, cases xs, simp-all add: add-suc, arith, erule-tac conjE,
   rule-tac allI, simp)
    fix k y ys xs'
    let ?f' = f(key y := tl (f (key y)))
    assume ⋀n' f'. foldl (+) 0 ns = n' ∧ length ys = n' ⟶
      (∀k'. [x←ys. key x = k'] = f' k') ⟶ (∀k'. [x←xs'. key x = k'] = f' k')
    moreover assume Suc (foldl (+) 0 ns) = n and Suc (length ys) = n
    ultimately have (∀k'. [x←ys. key x = k'] = ?f' k') ⟶
      (∀k'. [x←xs'. key x = k'] = ?f' k')
      by blast
    moreover assume A: ∀k'. (if key y = k'
      then y # [x←ys. key x = k'] else [x←ys. key x = k']) = f k'
    hence B: f (key y) = y # [x←ys. key x = key y]
      by (drule-tac spec [where x = key y], simp)
    hence ∀k'. [x←ys. key x = k'] = ?f' k'
    proof (rule-tac allI, simp, rule-tac impI)
      fix k'
      assume k' ≠ key y
      thus [x←ys. key x = k'] = f k'
        using A by (drule-tac spec [where x = k'], simp)
    qed
    ultimately have C: ∀k'. [x←xs'. key x = k'] = ?f' k' ..
    show (key y = k ⟶ y # [x←xs'. key x = k] = f k) ∧
      (key y ≠ k ⟶ [x←xs'. key x = k] = f k)
    proof (rule conjI, rule-tac [!] impI)
```

```
      assume key y = k
      moreover have tl (f (key y)) = [x←xs'. key x = key y]
        using C by simp
      hence f (key y) = y # [x←xs'. key x = key y]
        using B by (subst hd-Cons-tl [symmetric], simp-all)
      ultimately show y # [x←xs'. key x = k] = f k by simp
    next
      assume key y ≠ k
      thus [x←xs'. key x = k] = f k
        using C by simp
    qed
  qed
next
  fix index p q r u m ns n f and key :: 'a ⇒ 'b and xs :: 'a list
  let ?ws = take (Suc (Suc m)) xs
  let ?ys = drop (Suc (Suc m)) xs
  let ?r = λm'. round-suc-suc index key ?ws m m' u
  let ?t = λr' v. round index key p q r' (v, ns, ?ys)
  assume
    A: index-less index key and
    B: index-same index key
  assume
   ⋀ws a b c d e g h i n f.
      ws = ?ws ⟹ a = bn-comp m p q r ⟹ (b, c) = bn-comp m p q r ⟹
      d = ?r b ⟹ (e, g) = ?r b ⟹ (h, i) = g ⟹
        bn-inv p q (e, ns, ?ys) ⟶ add-inv n (e, ns, ?ys) ⟶
          stab-inv f key (e, ns, ?ys) ⟶ stab-inv f key (?t c e) and
   bn-inv p q (u, Suc (Suc m) # ns, xs) and
   add-inv n (u, Suc (Suc m) # ns, xs) and
   stab-inv f key (u, Suc (Suc m) # ns, xs)
  thus stab-inv f key (round index key p q r (u, Suc (Suc m) # ns, xs))
  using [[simproc del: defined-all]]
  proof (simp split: prod.split, ((rule-tac allI)+, ((rule-tac impI)+)?)+,
   (erule-tac conjE)+, subst (asm) (2) add-base-zero, simp)
    fix m' r' v ms' ws' xs' k
    let ?f = λk. [x←?ys. key x = k]
    let ?P = λf. ∀k. [x←?ys. key x = k] = f k
    let ?Q = λf. ∀k. [x←xs'. key x = k] = f k
    assume
      C: ?r m' = (v, ms', ws') and
      D: bn-comp m p q r = (m', r') and
      E: bn-valid m p q and
      F: Suc (Suc (foldl (+) 0 ns + m)) = n and
      G: length xs = n
    assume ⋀ws a b c d e g h i n' f.
      ws = ?ws ⟹ a = (m', r') ⟹ b = m' ∧ c = r' ⟹
      d = (v, ms', ws') ⟹ e = v ∧ g = (ms', ws') ⟹ h = ms' ∧ i = ws' ⟹
        foldl (+) 0 ns = n' ∧ n − Suc (Suc m) = n' ⟶ ?P f ⟶ ?Q f
    hence ⋀f. foldl (+) 0 ns = n − Suc (Suc m) ⟶ ?P f ⟶ ?Q f
```

by *simp*
hence $\bigwedge f$. *?P f* $\longrightarrow$ *?Q f*
using *F* **by** *simp*
hence *?P ?f* $\longrightarrow$ *?Q ?f* **.**
hence [*x*←*xs'*. *key x* = *k*] = [*x*←*?ys*. *key x* = *k*] **by** *simp*
moreover assume $\forall$ *k*. [*x*←*xs*. *key x* = *k*] = *f k*
hence *f k* = [*x*←*?ws* @ *?ys*. *key x* = *k*] **by** *simp*
ultimately have *f k* = [*x*←*?ws*. *key x* = *k*] @ [*x*←*xs'*. *key x* = *k*]
by (*subst* (*asm*) *filter-append*, *simp*)
with *C* [*symmetric*] **show** [*x*←*ws'*. *key x* = *k*] @ [*x*←*xs'*. *key x* = *k*] = *f k*
proof (*simp add*: *round-suc-suc-def Let-def del*: *filter.simps*
split: *if-split-asm*)
let *?nmi* = *mini ?ws key*
let *?nma* = *maxi ?ws key*
let *?xmi* = *?ws* ! *?nmi*
let *?xma* = *?ws* ! *?nma*
let *?mi* = *key ?xmi*
let *?ma* = *key ?xma*
let *?k* = *case m of 0* ⇒ *m* | *Suc 0* ⇒ *m* | *Suc* (*Suc i*) ⇒ *u* + *m'*
let *?zs* = *nths ?ws* (− {*?nmi*, *?nma*})
let *?ms* = *enum ?zs index key ?k ?mi ?ma*
have *H*: *length ?ws* = *Suc* (*Suc m*)
using *F* **and** *G* **by** *simp*
hence *I*: *?nmi* ≠ *?nma*
by (*rule-tac mini-maxi-neq*, *simp*)
have [*x*←(([*?xmi*] @ *map the* (*fill ?zs* (*offs ?ms 0*) *index key m ?mi ?ma*))
@ [*?xma*]). *key x* = *k*] = [*x*←*?ws*. *key x* = *k*]
proof (*cases m* = *0*)
case *True*
have *?nmi* < *length ?ws*
using *H* **by** (*rule-tac mini-less*, *simp*)
hence *J*: *?nmi* < *Suc* (*Suc 0*)
using *True* **by** *simp*
moreover have *?nma* < *length ?ws*
using *H* **by** (*rule-tac maxi-less*, *simp*)
hence *K*: *?nma* < *Suc* (*Suc 0*)
using *True* **by** *simp*
ultimately have *card* ({..<*Suc* (*Suc 0*)} − {*?nma*} − {*?nmi*}) = *0*
using *I* **by** *auto*
hence *L*: {..<*Suc* (*Suc 0*)} − {*?nma*} − {*?nmi*} = {} **by** *simp*
have *length* (*fill ?zs* (*offs ?ms 0*) *index key m ?mi ?ma*) = *0*
using *True* **by** (*simp add*: *fill-length*)
hence *M*: *map the* (*fill ?zs* (*offs ?ms 0*) *index key m ?mi ?ma*) =
nths ?ws ({..<*Suc* (*Suc 0*)} − {*?nma*} − {*?nmi*})
by (*subst L*, *simp*)
show *?thesis*
proof (*subst M*, *subst filter-append*)
show [*x*←[*?xmi*] @ *nths ?ws* ({..<*Suc* (*Suc 0*)} − {*?nma*} − {*?nmi*}).
key x = *k*] @ [*x*←[*?xma*]. *key x* = *k*] = [*x*←*?ws*. *key x* = *k*]

```
        proof (subst mini-stable, simp only: length-greater-0-conv
         [symmetric] H, simp add: I J, subst filter-append [symmetric])
          show [x←nths ?ws ({..<Suc (Suc 0)} − {?nma}) @ [?xma].
            key x = k] = [x←?ws. key x = k]
            by (subst maxi-stable, simp only: length-greater-0-conv
             [symmetric] H, simp add: K, simp add: True)
        qed
      qed
    next
      case False
      hence 0 < ?k
        by (simp, drule-tac bn-comp-fst-nonzero [OF E], subst (asm) D,
         simp split: nat.split)
      hence [x←map the (fill ?zs (offs ?ms 0) index key (length ?zs) ?mi ?ma).
        key x = k] = [x←?zs. k = key x]
        by (rule-tac fill-offs-enum-stable [OF A B], simp, rule-tac conjI,
         ((rule-tac mini-lb | rule-tac maxi-ub), erule-tac in-set-nthsD)+)
      moreover have [x←?zs. k = key x] = [x←?zs. key x = k]
        by (rule filter-cong, simp, blast)
      ultimately have
        J: [x←map the (fill ?zs (offs ?ms 0) index key m ?mi ?ma).
          key x = k] = [x←?zs. key x = k]
        using H by (simp add: mini-maxi-nths)
      show ?thesis
      proof (simp only: filter-append J, subst Compl-eq-Diff-UNIV,
       subst Diff-insert, subst filter-append [symmetric])
        show [x←[?xmi] @ nths ?ws (UNIV − {?nma} − {?nmi}). key x = k]
          @ [x←[?xma]. key x = k] = [x←?ws. key x = k]
        proof (subst mini-stable, simp only: length-greater-0-conv
         [symmetric] H, simp add: I, subst filter-append [symmetric])
          show [x←nths ?ws (UNIV − {?nma}) @ [?xma]. key x = k] =
            [x←?ws. key x = k]
            by (subst maxi-stable, simp only: length-greater-0-conv
             [symmetric] H, simp, subst nths-range, subst H, simp)
        qed
      qed
    qed
    thus [x←?xmi # map the (fill ?zs (offs ?ms 0) index key m ?mi ?ma) @
      [?xma]. key x = k] = [x←?ws. key x = k]
      by simp
  qed
 qed
qed

lemma gcsort-stab-inv:
  assumes
    A: index-less index key and
    B: index-same index key and
    C: add-inv n t and
```

```
    D: n ≤ p
  shows ⟦t' ∈ gcsort-set index key p t; stab-inv f key t⟧ ⟹
    stab-inv f key t'
by (erule gcsort-set.induct, simp, drule gcsort-add-inv [OF A - C D],
 rule round-stab-inv [OF A B], simp-all del: bn-inv.simps, erule conjE,
 frule sym, erule subst, rule bn-inv-intro, insert D, simp)
```

The only remaining task is to address step 10 of the proof method, which is done by proving theorem *gcsort-stable*. It holds under the conditions that the objects' number is not larger than the counters' upper bound and function *index* satisfies both predicates *index-less* and *index-same*, and states that function *gcsort* leaves unchanged the sublist of the objects having a given key within the input objects' list.

```
theorem gcsort-stable:
  assumes
    A: index-less index key and
    B: index-same index key and
    C: length xs ≤ p
  shows [x←gcsort index key p xs. key x = k] = [x←xs. key x = k]
proof −
  let ?t = (0, [length xs], xs)
  have stab-inv (λk. [x←xs. key x = k]) key (gcsort-aux index key p ?t)
    by (rule gcsort-stab-inv [OF A B - C], rule gcsort-add-input,
     rule gcsort-aux-set, rule gcsort-stab-input)
  hence [x←gcsort-out (gcsort-aux index key p ?t). key x = k] =
    [x←xs. key x = k]
    by (rule gcsort-stab-intro)
  moreover have ?t = gcsort-in xs
    by (simp add: gcsort-in-def)
  ultimately show ?thesis
    by (simp add: gcsort-def)
qed

end
```

References

[1] P. E. Black. Histogram sort — Dictionary of Algorithms and Data Structures, Feb. 2019. https://www.nist.gov/dads/HTML/histogramSort.html.

[2] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

[3] A. Krauss. *Defining Recursive Functions in Isabelle/HOL*. https://isabelle.in.tum.de/website-Isabelle2019/dist/Isabelle2019/doc/functions.pdf.

[4] T. Nipkow. *A Tutorial Introduction to Structured Isar Proofs*. https://isabelle.in.tum.de/website-Isabelle2011/dist/Isabelle2011/doc/isar-overview.pdf.

[5] T. Nipkow. *Programming and Proving in Isabelle/HOL*, June 2019. https://isabelle.in.tum.de/website-Isabelle2019/dist/Isabelle2019/doc/prog-prove.pdf.

[6] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, June 2019. https://isabelle.in.tum.de/website-Isabelle2019/dist/Isabelle2019/doc/tutorial.pdf.

[7] P. Noce. A General Method for the Proof of Theorems on Tail-recursive Functions. *Archive of Formal Proofs*, Dec. 2013. http://isa-afp.org/entries/Tail_Recursive_Functions.html, Formal proof development.

[8] Wikipedia contributors. Counting sort — Wikipedia, The Free Encyclopedia, Sept. 2019. https://en.wikipedia.org/w/index.php?title=Counting_sort&oldid=915065502.