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

May 26, 2024

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *add-suc*:
foldl (+) (*Suc m*) *ns* = *Suc* (*foldl* (+) *m ns*)
⟨*proof*⟩

lemma *add-update*:
i < *length ns* ⟹ *foldl* (+) *m* (*ns*[*i* := *Suc* (*ns* ! *i*)]) = *Suc* (*foldl* (+) *m ns*)
⟨*proof*⟩

lemma *add-le*:
(*m* :: *nat*) ≤ *foldl* (+) *m ns*
⟨*proof*⟩

lemma *add-mono*:
(*m* :: *nat*) ≤ *n* ⟹ *foldl* (+) *m ns* ≤ *foldl* (+) *n ns*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *enum-length*:
length (*enum xs index key n mi ma*) = *n*
⟨*proof*⟩

lemma *enum-add-le*:
foldl (+) *0* (*enum xs index key n mi ma*) ≤ *length xs*
⟨*proof*⟩

lemma *enum-max-le*:
0 < *n* ⟹ *Max* (*set* (*enum xs index key n mi ma*)) ≤ *length xs*
(**is** - ⟹ *Max* (*set ?ns*) ≤ -)
⟨*proof*⟩

lemma *mini-less*:
0 < *length xs* ⟹ *mini xs key* < *length xs*
⟨*proof*⟩

lemma *maxi-less*:
0 < *length xs* ⟹ *maxi xs key* < *length xs*
⟨*proof*⟩

lemma *mini-lb*:
x ∈ *set xs* ⟹ *key* (*xs* ! *mini xs key*) ≤ *key x*
⟨*proof*⟩

lemma *maxi-ub*:
x ∈ *set xs* ⟹ *key x* ≤ *key* (*xs* ! *maxi xs key*)
⟨*proof*⟩

lemma *mini-maxi-neq* [*rule-format*]:
Suc 0 < *length xs* ⟶ *mini xs key* ≠ *maxi xs key*
⟨*proof*⟩

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*⟩

lemma *mini-maxi-nths-le*:
length xs ≤ *Suc* (*Suc n*) ⟹ *length* (*nths xs* (− {*mini xs key*, *maxi xs key*})) ≤ *n*
⟨*proof*⟩

lemma *round-nil*:
(*fst* (*snd* (*round index key p q r t*)) ≠ []) = (∃ *n* ∈ *set* (*fst* (*snd t*)). *0* < *n*)
⟨*proof*⟩

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*⟩

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*⟩

termination *gcsort-aux*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *gcsort-aux-set*:
gcsort-aux index key p t ∈ gcsort-set index key p t
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *add-base*:
foldl (+) (*k* + *m*) *ns* = *foldl* (+) *m ns* + (*k* :: *nat*)
⟨*proof*⟩

lemma *add-base-zero*:
foldl (+) *k ns* = *foldl* (+) *0 ns* + (*k* :: *nat*)
⟨*proof*⟩

lemma *bn-count-le*:
bn-count ns ≤ *foldl* (+) *0 ns*
⟨*proof*⟩

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*⟩

In what follows, the invariance of predicate *add-inv* over inductive set *gcsort-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 $\longrightarrow$ *n* $\leq$ *fst* (*bn-comp n p q r*)
⟨*proof*⟩

lemma *bn-comp-fst-nonzero*:
bn-valid n p q $\Longrightarrow$ *0* $<$ *n* $\Longrightarrow$ *0* $<$ *fst* (*bn-comp n p q r*)
⟨*proof*⟩

lemma *bn-comp-snd-less*:
r $<$ *q* $\Longrightarrow$ *snd* (*bn-comp n p q r*) $<$ *q*
⟨*proof*⟩

lemma *add-replicate*:
foldl (+) *k* (*replicate m n*) = *k* + *m* $*$ *n*
⟨*proof*⟩

lemma *fill-length*:
length (*fill xs ns index key n mi ma*) = *n*
⟨*proof*⟩

lemma *enum-add* [*rule-format*]:
assumes
A: *index-less index key* **and**
B: *0* $<$ *n*
shows (∀ *x* ∈ *set xs*. *key x* ∈ {*mi*..*ma*}) $\longrightarrow$
foldl (+) *0* (*enum xs index key n mi ma*) = *length xs*
⟨*proof*⟩

lemma *round-add-inv* [*rule-format*]:
index-less index key $\longrightarrow$ *bn-inv p q t* $\longrightarrow$ *add-inv n t* $\longrightarrow$
add-inv n (*round index key p q r t*)
⟨*proof*⟩

lemma *gcsort-add-inv*:
assumes *A*: *index-less index key*
shows ⟦*t'* ∈ *gcsort-set index key p t*; *add-inv n t*; *n* $\leq$ *p*⟧ $\Longrightarrow$
add-inv n t'
⟨*proof*⟩

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 $\longrightarrow$ *r* $<$ *q* $\longrightarrow$
(*r* + (*foldl* (+) *0* (*fst* (*snd t*)) − *bn-count* (*fst* (*snd t*))) ∗ *p*) *mod q* = *0* $\longrightarrow$
(*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*
⟨*proof*⟩

lemma *round-len-eq* [*rule-format*]:
bn-count (*fst* (*snd t*)) = *foldl* (+) *0* (*fst* (*snd t*)) $\longrightarrow$
length (*fst* (*snd* (*round index key p q r t*))) = *foldl* (+) *0* (*fst* (*snd t*))
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *gcsort-count-intro*:
count-inv f t $\Longrightarrow$ *count* (*mset* (*gcsort-out t*)) *x* = *f x*
⟨*proof*⟩

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* :: $nat \Rightarrow 'a\ list \Rightarrow ('a, 'b)\ index\text{-}sign \Rightarrow$
$('a \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'b \Rightarrow nat \Rightarrow nat$ **where**
offs-num n xs index key mi ma i ≡
length [*x*←*xs*. *index key x n mi ma* = *i*]

abbreviation *offs-set-next* :: $nat\ list \Rightarrow 'a\ list \Rightarrow ('a, 'b)\ index\text{-}sign \Rightarrow$

$('a \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'b \Rightarrow nat \Rightarrow nat\ set$ **where**
offs-set-next ns xs index key mi ma i ≡
$\{k.\ k < length\ ns \wedge i < k \wedge 0 < offs\text{-}num\ (length\ ns)\ xs\ index\ key\ mi\ ma\ k\}$

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

definition *offs-next* :: $nat\ list \Rightarrow nat \Rightarrow 'a\ list \Rightarrow ('a, 'b)\ index\text{-}sign \Rightarrow$
$('a \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'b \Rightarrow nat \Rightarrow nat$ **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* :: $nat\ list \Rightarrow nat \Rightarrow 'a\ list \Rightarrow ('a, 'b)\ index\text{-}sign \Rightarrow$
$('a \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'b \Rightarrow nat \Rightarrow bool$ **where**
offs-none ns ub xs index key mi ma i ≡
$(\exists j < length\ ns.\ 0 < offs\text{-}num\ (length\ ns)\ xs\ index\ key\ mi\ ma\ j \wedge$
$i \in \{ns\ !\ j + offs\text{-}num\ (length\ ns)\ xs\ index\ key\ mi\ ma\ j..<$
offs-next ns ub xs index key mi ma j$\}) \vee$
offs-num (*length ns*) *xs index key mi ma 0* = *0* ∧
$i < offs\text{-}next\ ns\ ub\ xs\ index\ key\ mi\ ma\ 0 \vee$
$0 < offs\text{-}num\ (length\ ns)\ xs\ index\ key\ mi\ ma\ 0 \wedge$
$i < ns\ !\ 0$

definition *offs-pred* :: $nat\ list \Rightarrow nat \Rightarrow 'a\ list \Rightarrow ('a, 'b)\ index\text{-}sign \Rightarrow$
$('a \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'b \Rightarrow bool$ **where**
offs-pred ns ub xs index key mi ma ≡
$\forall i < length\ ns.\ offs\text{-}num\ (length\ ns)\ xs\ index\ key\ mi\ ma\ i \leq$
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*)
⟨*proof*⟩

lemma *offs-next-prev*:
$(0 < offs\text{-}num\ (length\ ns)\ xs\ index\ key\ mi\ ma\ i \wedge$
$offs\text{-}set\text{-}next\ ns\ xs\ index\ key\ mi\ ma\ i \neq \{\} \wedge$
$Min\ (offs\text{-}set\text{-}next\ ns\ xs\ index\ key\ mi\ ma\ i) = j) =$
$(0 < offs\text{-}num\ (length\ ns)\ xs\ index\ key\ mi\ ma\ j \wedge$
$offs\text{-}set\text{-}prev\ ns\ xs\ index\ key\ mi\ ma\ j \neq \{\} \wedge$
$Max\ (offs\text{-}set\text{-}prev\ ns\ xs\ index\ key\ mi\ ma\ j) = i)$
(**is** *?P* = *?Q*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*

⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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: $offs\text{-}num\ (length\ ns)\ xs\ index\ key\ mi\ ma\ i = 0$ **and**
C: $offs\text{-}set\text{-}prev\ ns\ xs\ index\ key\ mi\ ma\ i = \{\}$
shows $offs\text{-}next\ ns\ ub\ xs\ index\ key\ mi\ ma\ 0 =$
$offs\text{-}next\ ns\ ub\ xs\ index\ key\ mi\ ma\ i$
⟨*proof*⟩

lemma *offs-next-zero-cons-eq*:
assumes
A: $index\ key\ x\ (length\ ns)\ mi\ ma = i$ **and**
B: $offs\text{-}num\ (length\ ns)\ (x \# xs)\ index\ key\ mi\ ma\ 0 = 0$ **and**
C: $offs\text{-}set\text{-}prev\ ns\ (x \# xs)\ index\ key\ mi\ ma\ i \neq \{\}$
(**is** *?A* ≠ -)
shows $offs\text{-}next\ (ns[i := Suc\ (ns\ !\ i)])\ ub\ xs\ index\ key\ mi\ ma\ 0 =$
$offs\text{-}next\ ns\ ub\ (x \# xs)\ index\ key\ mi\ ma\ 0$
⟨*proof*⟩

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: $offs\text{-}set\text{-}prev\ ns\ (x \# xs)\ index\ key\ mi\ ma\ i = \{\}$
shows $offs\text{-}next\ (ns[i := Suc\ (ns\ !\ i)])\ ub\ xs\ index\ key\ mi\ ma\ 0 =$
$(if\ 0 < offs\text{-}num\ (length\ ns)\ xs\ index\ key\ mi\ ma\ i$
$then\ Suc\ (ns\ !\ i)$
$else\ offs\text{-}next\ ns\ ub\ (x \# xs)\ index\ key\ mi\ ma\ i)$
⟨*proof*⟩

lemma *offs-pred-zero-cons-less*:
assumes
A: $offs\text{-}pred\ ns\ ub\ (x \# xs)\ index\ key\ mi\ ma$ **and**
B: $index\ key\ x\ (length\ ns)\ mi\ ma = i$ **and**
C: $i < length\ ns$ **and**
D: $0 < i$ **and**
E: $offs\text{-}set\text{-}prev\ ns\ (x \# xs)\ index\ key\ mi\ ma\ i = \{\}$
shows $offs\text{-}next\ ns\ ub\ (x \# xs)\ index\ key\ mi\ ma\ 0 <$
$offs\text{-}next\ (ns[i := Suc\ (ns\ !\ i)])\ ub\ xs\ index\ key\ mi\ ma\ 0$
(**is** *?M* < *?N*)
⟨*proof*⟩

lemma *offs-pred-zero-cons*:
assumes
A: $offs\text{-}pred\ ns\ ub\ (x \# xs)\ index\ key\ mi\ ma$ **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* $ns\ ub\ (x \# xs)\ index\ key\ mi\ ma\ 0 \leq$
offs-next $(ns[i := Suc\ (ns\ !\ i)])\ ub\ xs\ index\ key\ mi\ ma\ 0$
(**is** $?M \leq ?N$)
⟨*proof*⟩

lemma *replicate-count*:
$count\ (mset\ (replicate\ n\ x))\ x = n$
⟨*proof*⟩

lemma *fill-none* [*rule-format*]:
assumes *A*: *index-less index key*
shows
$(\forall x \in set\ xs.\ key\ x \in \{mi..ma\}) \longrightarrow$
$ns \neq [] \longrightarrow$
offs-pred ns ub xs index key mi ma $\longrightarrow$
offs-none ns ub xs index key mi ma i $\longrightarrow$
fill xs ns index key ub mi ma ! $i = None$
⟨*proof*⟩

lemma *fill-index-none* [*rule-format*]:
assumes
A: *index-less index key* **and**
B: $key\ x \in \{mi..ma\}$ **and**
C: $ns \neq []$ **and**
D: *offs-pred ns ub* $(x \# xs)$ *index key mi ma*
shows $\forall x \in set\ xs.\ key\ x \in \{mi..ma\} \Longrightarrow$
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** - $\Longrightarrow$ *fill* - *?ns′* - - - - - ! (- ! *?i*) = -)
⟨*proof*⟩

lemma *fill-count-item* [*rule-format*]:
assumes *A*: *index-less index key*
shows
$(\forall x \in set\ xs.\ key\ x \in \{mi..ma\}) \longrightarrow$
$ns \neq [] \longrightarrow$
offs-pred ns ub xs index key mi ma $\longrightarrow$
$length\ xs \leq ub \longrightarrow$
$count\ (mset\ (map\ the\ (fill\ xs\ ns\ index\ key\ ub\ mi\ ma)))\ x =$
$count\ (mset\ xs)\ x + (if\ the\ None = x\ then\ ub - length\ xs\ else\ 0)$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *offs-length*:
length (*offs ns i*) = *length ns*
⟨*proof*⟩

lemma *offs-add* [*rule-format*]:
$i <$ *length ns* $\longrightarrow$ *offs ns k* ! $i =$ *foldl* (+) *k* (*take i ns*)
⟨*proof*⟩

lemma *offs-mono-aux*:
$i \leq j \Longrightarrow j <$ *length ns* $\Longrightarrow$ *offs ns k* ! $i \leq$ *offs ns k* ! $(i + (j - i))$
⟨*proof*⟩

lemma *offs-mono*:
$i \leq j \Longrightarrow j <$ *length ns* $\Longrightarrow$ *offs ns k* ! $i \leq$ *offs ns k* ! j
⟨*proof*⟩

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)
⟨*proof*⟩

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*⟩

lemma *offs-equal* [*rule-format*]:
$i < j \Longrightarrow j <$ *length ns* $\Longrightarrow$
$(\forall\, k \in \{i..<j\}.$ *ns* ! $k = 0) \longrightarrow$ *offs ns m* ! $i =$ *offs ns m* ! j
⟨*proof*⟩

lemma *offs-enum-last* [*rule-format*]:
assumes
A: *index-less index key* **and**
B: $0 < n$ **and**
C: $\forall\, x \in$ *set xs*. *key* $x \in \{mi..ma\}$
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*⟩

lemma *offs-enum-ub* [*rule-format*]:
assumes
A: *index-less index key* **and**
B: *i < n* **and**
C: ∀ *x* ∈ *set xs. key x* ∈ {*mi..ma*}
shows *offs* (*enum xs index key n mi ma*) *k* ! *i* ≤ *length xs + k*
⟨*proof*⟩

lemma *offs-enum-next-ge* [*rule-format*]:
assumes
A: *index-less index key* **and**
B: *i < n*
shows ∀ *x* ∈ *set xs. key x* ∈ {*mi..ma*} ⟹
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
(**is** - ⟹ *offs ?ns* - ! - ≤ -)
⟨*proof*⟩

lemma *offs-enum-zero-aux* [*rule-format*]:
⟦*index-less index key*; *0 < n*; ∀ *x* ∈ *set xs. key x* ∈ {*mi..ma*};
offs-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*
⟨*proof*⟩

lemma *offs-enum-zero* [*rule-format*]:
assumes
A: *index-less index key* **and**
B: *i < n* **and**
C: ∀ *x* ∈ *set xs. key x* ∈ {*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*⟩

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*⟩

lemma *offs-enum-pred* [*rule-format*]:
assumes A: *index-less index key*
shows $(\forall x \in set\ xs.\ key\ x \in \{mi..ma\}) \longrightarrow$
offs-pred (*offs* (*enum xs index key n mi ma*) *k*) (*length xs* + *k*)
xs index key mi ma
⟨*proof*⟩

lemma *fill-offs-enum-count-item* [*rule-format*]:
$[\![$*index-less index key*; $\forall x \in set\ xs.\ key\ x \in \{mi..ma\}$; $0 < n]\!] \Longrightarrow$
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*
⟨*proof*⟩

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* $\{i.\ i < length\ xs \wedge i \notin A \wedge xs\ !\ i = x\}$
⟨*proof*⟩

lemma *round-count-inv* [*rule-format*]:
index-less index key $\longrightarrow$ *bn-inv p q t* $\longrightarrow$ *add-inv n t* $\longrightarrow$ *count-inv f t* $\longrightarrow$
count-inv f (*round index key p q r t*)
⟨*proof*⟩

lemma *gcsort-count-inv*:
assumes
A: *index-less index key* **and**
B: *add-inv n t* **and**
C: $n \leq p$
shows $[\![t' \in gcsort\text{-}set\ index\ key\ p\ t;\ count\text{-}inv\ f\ t]\!] \Longrightarrow$
count-inv f t'
⟨*proof*⟩

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* $\leq$ *p*
  shows *count* (*mset* (*gcsort index key p xs*)) *x* = *count* (*mset xs*) *x*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *gcsort-sort-intro*:
⟦*sort-inv key t*; *add-inv n t*; *find* (λ*n*. *Suc 0* < *n*) (*fst* (*snd t*)) = *None*⟧ ⟹
sorted (*map key* (*gcsort-out t*))
⟨*proof*⟩

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 (λ*n*. *Suc 0* < *n*) (*fst* (*snd* (*gcsort-aux index key p t*))) = *None*
⟨*proof*⟩

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*:

$x \in set\ xs \Longrightarrow key\ (xs\ !\ mini\ xs\ key) \leq key\ (xs\ !\ maxi\ xs\ key)$
⟨*proof*⟩

lemma *mini-maxi-keys-eq* [*rule-format*]:
$key\ (xs\ !\ mini\ xs\ key) = key\ (xs\ !\ maxi\ xs\ key) \longrightarrow x \in set\ xs \longrightarrow$
$key\ x = key\ (xs\ !\ maxi\ xs\ key)$
⟨*proof*⟩

lemma *offs-suc*:
$i < length\ ns \Longrightarrow offs\ ns\ (Suc\ k)\ !\ i = Suc\ (offs\ ns\ k\ !\ i)$
⟨*proof*⟩

lemma *offs-base-zero*:
$i < length\ ns \Longrightarrow offs\ ns\ k\ !\ i = offs\ ns\ 0\ !\ i + k$
⟨*proof*⟩

lemma *offs-append*:
$offs\ (ms\ @\ ns)\ k = offs\ ms\ k\ @\ offs\ ns\ (foldl\ (+)\ k\ ms)$
⟨*proof*⟩

lemma *offs-enum-next-le* [*rule-format*]:
assumes
A: *index-less index key* **and**
B: $i < j$ **and**
C: $j < n$ **and**
D: $\forall x \in set\ xs.\ key\ x \in \{mi..ma\}$
shows $offs\ (enum\ xs\ index\ key\ n\ mi\ ma)\ k)\ (length\ xs + k)$
$xs\ index\ key\ mi\ ma\ i \leq offs\ (enum\ xs\ index\ key\ n\ mi\ ma)\ k\ !\ j$
(**is** - ≤ *offs ?ns* - ! -)
⟨*proof*⟩

lemma *offs-pred-ub-less*:
⟦*offs-pred ns ub xs index key mi ma*; $i < length\ ns$;
$0 < offs\text{-}num\ (length\ ns)\ xs\ index\ key\ mi\ ma\ i$⟧ ⟹
$ns\ !\ i < ub$
⟨*proof*⟩

lemma *fill-count-none* [*rule-format*]:
assumes *A*: *index-less index key*
shows
$(\forall x \in set\ xs.\ key\ x \in \{mi..ma\}) \longrightarrow$
$ns \neq [] \longrightarrow$
offs-pred ns ub xs index key mi ma $\longrightarrow$
$length\ xs \leq ub \longrightarrow$
$count\ (mset\ (fill\ xs\ ns\ index\ key\ ub\ mi\ ma))\ None = ub - length\ xs$
⟨*proof*⟩

lemma *fill-offs-enum-count-none* [*rule-format*]:
⟦*index-less index key*; $\forall x \in set\ xs.\ key\ x \in \{mi..ma\}$; $0 < n$⟧ ⟹

count (*mset* (*fill xs* (*offs* (*enum xs index key n mi ma*) *0*)
index key (*length xs*) *mi ma*)) *None* = *0*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *fill-sort-inv* [*rule-format*]:
assumes
A: *index-less index key* **and**
B: *index-mono index key* **and**
C: ∀ *x* ∈ *set xs*. *key x* ∈ {*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*⟩

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*⟩

lemma *gcsort-sort-inv*:
assumes
A: *index-less index key* **and**
B: *index-mono index key* **and**

C: *add-inv n t* **and**
D: $n \leq p$
shows $[\![t' \in$ *gcsort-set index key p t*; *sort-inv key t*$]\!] \Longrightarrow$
sort-inv key t'
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *gcsort-stab-intro*:
stab-inv f key t $\Longrightarrow$ [*x*←*gcsort-out t. key x* = *k*] = *f k*
⟨*proof*⟩

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*⟩

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*} $\Longrightarrow$
0 < *n* $\Longrightarrow$
[*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** - $\Longrightarrow$ - $\Longrightarrow$ [-←*map the ?ys*. -] = -
is - $\Longrightarrow$ - $\Longrightarrow$ [-←*map the* (*fill* - *?ns* - - - - -). -] = -)
⟨*proof*⟩

lemma *mini-first* [*rule-format*]:
xs ≠ [] ⟶ *i* < *mini xs key* ⟶
key (*xs* ! *mini xs key*) < *key* (*xs* ! *i*)
⟨*proof*⟩

lemma *maxi-last* [*rule-format*]:
$xs \neq [] \longrightarrow maxi\ xs\ key < i \longrightarrow i < length\ xs \longrightarrow$
$key\ (xs\ !\ i) < key\ (xs\ !\ maxi\ xs\ key)$
⟨*proof*⟩

lemma *nths-range*:
$nths\ xs\ A = nths\ xs\ (A \cap \{..<length\ xs\})$
⟨*proof*⟩

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*⟩

lemma *mini-stable*:
assumes
A: $xs \neq []$ **and**
B: $mini\ xs\ key \in A$
(**is** *?nmi* ∈ -)
shows $[x{\leftarrow}[xs\ !\ ?nmi]\ @\ nths\ xs\ (A - \{?nmi\}).\ key\ x = k] =$
$[x{\leftarrow}nths\ xs\ A.\ key\ x = k]$
(**is** $[x{\leftarrow}[?xmi]\ @\ \text{-}.\ \text{-}] = \text{-}$)
⟨*proof*⟩

lemma *maxi-stable*:
assumes
A: $xs \neq []$ **and**
B: $maxi\ xs\ key \in A$
(**is** *?nma* ∈ -)
shows $[x{\leftarrow}nths\ xs\ (A - \{?nma\})\ @\ [xs\ !\ ?nma].\ key\ x = k] =$
$[x{\leftarrow}nths\ xs\ A.\ key\ x = k]$
(**is** $[x{\leftarrow}\text{-}\ @\ [?xma].\ \text{-}] = \text{-}$)
⟨*proof*⟩

lemma *round-stab-inv* [*rule-format*]:
$index\text{-}less\ index\ key \longrightarrow index\text{-}same\ index\ key \longrightarrow bn\text{-}inv\ p\ q\ t \longrightarrow$
$add\text{-}inv\ n\ t \longrightarrow stab\text{-}inv\ f\ key\ t \longrightarrow stab\text{-}inv\ f\ key\ (round\ index\ key\ p\ q\ r\ t)$
⟨*proof*⟩

lemma *gcsort-stab-inv*:
assumes
A: $index\text{-}less\ index\ key$ **and**
B: $index\text{-}same\ index\ key$ **and**
C: $add\text{-}inv\ n\ t$ **and**
D: $n \leq p$
shows $[\![t' \in gcsort\text{-}set\ index\ key\ p\ t;\ stab\text{-}inv\ f\ key\ t]\!] \Longrightarrow$
$stab\text{-}inv\ f\ key\ t'$

⟨*proof*⟩

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*⟩

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.