# 1

# The smallest free number

## Introduction

Consider the problem of computing the smallest natural number not in a given finite set $X$ of natural numbers. The problem is a simplification of a common programming task in which the numbers name objects and $X$ is the set of objects currently in use. The task is to find some object not in use, say the one with the smallest name.

The solution to the problem depends, of course, on how $X$ is represented. If $X$ is given as a list without duplicates and in increasing order, then the solution is straightforward: simply look for the first gap. But suppose $X$ is given as a list of distinct numbers in no particular order. For example,

$$[08, 23, 09, 00, 12, 11, 01, 10, 13, 07, 41, 04, 14, 21, 05, 17, 03, 19, 02, 06]$$

How would you find the smallest number not in this list?

It is not immediately clear that there is a linear-time solution to the problem; after all, sorting an arbitrary list of numbers cannot be done in linear time. Nevertheless, linear-time solutions do exist and the aim of this pearl is to describe two of them: one is based on Haskell arrays and the other on divide and conquer.

## An array-based solution

The problem can be specified as a function *minfree*, defined by

$$\begin{array}{lcl} \mathit{minfree} & :: & [\mathit{Nat}] \to \mathit{Nat} \\ \mathit{minfree}\ \mathit{xs} & = & \mathit{head}\ ([0\,..\,] \setminus\setminus \mathit{xs}) \end{array}$$

The expression $us \setminus\setminus vs$ denotes the list of those elements of $us$ that remain after removing any elements in $vs$:

$$\begin{array}{lcl} (\setminus\setminus) & :: & \mathit{Eq}\ a \Rightarrow [a] \to [a] \to [a] \\ \mathit{us} \setminus\setminus \mathit{vs} & = & \mathit{filter}\ (\notin \mathit{vs})\ \mathit{us} \end{array}$$

The function *minfree* is executable but requires $\Theta(n^2)$ steps on a list of length $n$ in the worst case. For example, evaluating *minfree* $[n{-}1, n{-}2\,..\,0]$ requires evaluating $i \notin [n{-}1, n{-}2\,..\,0]$ for $0 \leq i \leq n$, and thus $n(n+1)/2$ equality tests.

The key fact for both the array-based and divide and conquer solutions is that not every number in the range $[0\,..\,\mathit{length}\ xs]$ can be in $xs$. Thus the smallest number not in $xs$ is the smallest number not in $\mathit{filter}\,(\leq n)\,xs$, where $n = \mathit{length}\ xs$. The array-based program uses this fact to build a checklist of those numbers present in $\mathit{filter}\ (\leq n)\ xs$. The checklist is a Boolean array with $n+1$ slots, numbered from 0 to $n$, whose initial entries are everywhere *False*. For each element $x$ in $xs$ and provided $x \leq n$ we set the array element at position $x$ to *True*. The smallest free number is then found as the position of the first *False* entry. Thus, $\mathit{minfree} = \mathit{search} \cdot \mathit{checklist}$, where

$$\begin{array}{lcl} \mathit{search} & :: & \mathit{Array\ Int\ Bool} \to \mathit{Int} \\ \mathit{search} & = & \mathit{length} \cdot \mathit{takeWhile\ id} \cdot \mathit{elems} \end{array}$$

The function *search* takes an array of Booleans, converts the array into a list of Booleans and returns the length of the longest initial segment consisting of *True* entries. This number will be the position of the first *False* entry.

One way to implement *checklist* in linear time is to use the function *accumArray* in the Haskell library *Data.Array*. This function has the rather daunting type

$$\mathit{Ix}\ i \Rightarrow (e \to v \to e) \to e \to (i, i) \to [(i, v)] \to \mathit{Array}\ i\ e$$

The type constraint $\mathit{Ix}\ i$ restricts $i$ to be an *Index* type, such as *Int* or *Char*, for naming the indices or positions of the array. The first argument is an "accumulating" function for transforming array entries and values into new entries, the second argument is an initial entry for each index, the third argument is a pair naming the lower and upper indices and the fourth is an association list of index–value pairs. The function *accumArray* builds an array by processing the association list from left to right, combining entries and values into new entries using the accumulating function. This process takes linear time in the length of the association list, assuming the accumulating function takes constant time.

The function *checklist* is defined as an instance of *accumArray*:

$$\begin{array}{lcl} \mathit{checklist} & :: & [\mathit{Int}] \to \mathit{Array\ Int\ Bool} \\ \mathit{checklist\ xs} & = & \mathit{accumArray}\ (\vee)\ \mathit{False}\ (0, n) \\ & & (\mathit{zip}\ (\mathit{filter}\ (\leq n)\ xs)\ (\mathit{repeat\ True})) \\ & & \textbf{where}\ n = \mathit{length\ xs} \end{array}$$

This implementation does not require the elements of $xs$ to be distinct, but does require them to be natural numbers.

It is worth mentioning that *accumArray* can be used to sort a list of numbers in linear time, provided the elements of the list all lie in some known range $(0, n)$. We replace *checklist* by *countlist*, where

$$\begin{array}{lcl} \mathit{countlist} & :: & [\mathit{Int}] \to \mathit{Array\ Int\ Int} \\ \mathit{countlist\ xs} & = & \mathit{accumArray}\ (+)\ 0\ (0, n)\ (\mathit{zip\ xs}\ (\mathit{repeat}\ 1)) \end{array}$$

Then $\mathit{sort\ xs} = \mathit{concat}\ [\mathit{replicate}\ k\ x \mid (x, k) \leftarrow \mathit{countlist\ xs}]$. In fact, if we use *countlist* instead of *checklist*, then we can implement *minfree* as the position of the first 0 entry.

The above implementation builds the array in one go using a clever library function. A more prosaic way to implement *checklist* is to tick off entries step by step using a constant-time update operation. This is possible in Haskell only if the necessary array operations are executed in a suitable monad, such as the state monad. The following program for *checklist* makes use of the library *Data.Array.ST*:

$$\begin{array}{lcl} \mathit{checklist\ xs} & = & \mathit{runSTArray}\ (\mathbf{do} \\ & & \quad \{a \leftarrow \mathit{newArray}\ (0, n)\ \mathit{False}; \\ & & \quad\ \mathit{sequence}\ [\mathit{writeArray}\ a\ x\ \mathit{True} \mid x \leftarrow xs,\ x \leq n]; \\ & & \quad\ \mathit{return}\ a\}) \\ & & \mathbf{where}\ n = \mathit{length\ xs} \end{array}$$

This solution would not satisfy the pure functional programmer because it is essentially a procedural program in functional clothing.

## A divide and conquer solution

Now we turn to a divide and conquer solution to the problem. The idea is to express $\mathit{minfree}\,(xs +\!\!+ ys)$ in terms of $\mathit{minfree}\,xs$ and $\mathit{minfree}\,ys$. We begin by recording the following properties of $\backslash\backslash$:

$$\begin{array}{lcl} (as +\!\!+ bs) \backslash\backslash cs & = & (as \backslash\backslash cs) +\!\!+ (bs \backslash\backslash cs) \\ as \backslash\backslash (bs +\!\!+ cs) & = & (as \backslash\backslash bs) \backslash\backslash cs \\ (as \backslash\backslash bs) \backslash\backslash cs & = & (as \backslash\backslash cs) \backslash\backslash bs \end{array}$$

These properties reflect similar laws about sets in which set union $\cup$ replaces $+\!\!+$ and set difference $\setminus$ replaces $\backslash\backslash$. Suppose now that $as$ and $vs$ are disjoint, meaning $as \backslash\backslash vs = as$, and that $bs$ and $us$ are also disjoint, so $bs \backslash\backslash us = bs$. It follows from these properties of $+\!\!+$ and $\backslash\backslash$ that

$$(as +\!\!+ bs) \backslash\backslash (us +\!\!+ vs) \quad = \quad (as \backslash\backslash us) +\!\!+ (bs \backslash\backslash vs)$$

Now, choose any natural number $b$ and let $as = [0\,..\,b{-}1]$ and $bs = [b..]$. Furthermore, let $us = filter\ (<b)\ xs$ and $vs = filter\ (\geq b)\ xs$. Then $as$ and $vs$ are disjoint, and so are $bs$ and $us$. Hence

$$\begin{array}{lcl} [0\,..\,] \setminus\setminus xs & = & ([0\,..\,b{-}1] \setminus\setminus us) +\!\!+ ([b\,..\,] \setminus\setminus vs) \\ & & \textbf{where}\ (us, vs) = partition\ (<b)\ xs \end{array}$$

Haskell provides an efficient implementation of a function $partition\ p$ that partitions a list into those elements that satisfy $p$ and those that do not. Since

$$head\ (xs +\!\!+ ys) \quad = \quad \textbf{if}\ null\ xs\ \textbf{then}\ head\ ys\ \textbf{else}\ head\ xs$$

we obtain, still for any natural number $b$, that

$$\begin{array}{lcl} minfree\ xs & = & \textbf{if}\ null\ ([0\,..\,b{-}1] \setminus\setminus us) \\ & & \textbf{then}\ head\ ([b\,..\,] \setminus\setminus vs) \\ & & \textbf{else}\ head\ ([0\,..\,] \setminus\setminus us) \\ & & \textbf{where}\ (us, vs) = partition\ (<b)\ xs \end{array}$$

The next question is: can we implement the test $null\ ([0\,..\,b{-}1] \setminus\setminus us)$ more efficiently than by direct computation, which takes quadratic time in the length of $us$? Yes, the input is a list of distinct natural numbers, so is $us$. And every element of $us$ is less than $b$. Hence

$$null\ ([0\,..\,b{-}1] \setminus\setminus us \quad \equiv \quad length\ us == b$$

Note that the array-based solution did not depend on the assumption that the given list did not contain duplicates, but it is a crucial one for an efficient divide and conquer solution.

Further inspection of the above code for *minfree* suggests that we should generalise *minfree* to a function, *minfrom* say, defined by

$$\begin{array}{lcl} minfrom & :: & Nat \to [Nat] \to Nat \\ minfrom\ a\ xs & = & head\ ([a\,..\,] \setminus\setminus xs) \end{array}$$

where every element of $xs$ is assumed to be greater than or equal to $a$. Then, provided $b$ is chosen so that both $length\ us$ and $length\ vs$ are less than $length\ xs$, the following recursive definition of *minfree* is well-founded:

$$\begin{array}{lcl} minfree\ xs & = & minfrom\ 0\ xs \\ minfrom\ a\ xs & | & null\ xs \quad = \quad a \\ & | & length\ us == b - a \quad = \quad minfrom\ b\ vs \\ & | & \textbf{otherwise} \quad = \quad minfrom\ a\ us \\ & & \textbf{where}\ (us, vs) = partition\ (<b)\ xs \end{array}$$

It remains to choose $b$. Clearly, we want $b > a$. And we would also like to choose $b$ so that the maximum of the lengths of $us$ and $vs$ is as small as possible. The right choice of $b$ to satisfy these requirements is

$$b \quad = \quad a + 1 + n \operatorname{div} 2$$

where $n = \mathit{length}\ xs$. If $n \neq 0$ and $\mathit{length}\ us < b - a$, then

$$\mathit{length}\ us \leq n \operatorname{div} 2 < n$$

And, if $\mathit{length}\ us = b - a$, then

$$\mathit{length}\ vs = n - n \operatorname{div} 2 - 1 \leq n \operatorname{div} 2$$

With this choice the number of steps $T(n)$ for evaluating $\mathit{minfrom}\ 0\ xs$ when $n = \mathit{length}\ xs$ satisfies $T(n) = T(n \operatorname{div} 2) + \Theta(n)$, with the solution $T(n) = \Theta(n)$.

As a final optimisation we can avoid repeatedly computing *length* with a simple data refinement, representing $xs$ by a pair $(\mathit{length}\ xs, xs)$. That leads to the final program

$$\begin{array}{lcl}
\mathit{minfree}\ xs & = & \mathit{minfrom}\ 0\ (\mathit{length}\ xs, xs) \\
\mathit{minfrom}\ a\ (n, xs) & | & n == 0 \quad = \quad a \\
& | & m == b - a \quad = \quad \mathit{minfrom}\ b\ (n - m, vs) \\
& | & \textbf{otherwise} \quad = \quad \mathit{minfrom}\ a\ (m, us) \\
& & \textbf{where}\ (us, vs) \quad = \quad \mathit{partition}\ (< b)\ xs \\
& & \qquad\quad b \quad = \quad a + 1 + n \operatorname{div} 2 \\
& & \qquad\quad m \quad = \quad \mathit{length}\ us
\end{array}$$

It turns out that the above program is about twice as fast as the incremental array-based program, and about 20% faster than the one using *accumArray*.

## Final remarks

This was a simple problem with at least two simple solutions. The second solution was based on a common technique of algorithm design, namely divide and conquer. The idea of partitioning a list into those elements less than a given value, and the rest, arises in a number of algorithms, most notably Quicksort. When seeking a $\Theta(n)$ algorithm involving a list of $n$ elements, it is tempting to head at once for a method that processes each element of the list in constant time, or at least in amortized constant time. But a recursive process that performs $\Theta(n)$ processing steps in order to reduce the problem to another instance of at most half the size is also good enough.

One of the differences between a pure functional algorithm designer and a procedural one is that the former does not assume the existence of arrays with a constant-time update operation, at least not without a certain amount of plumbing. For a pure functional programmer, an update operation takes logarithmic time in the size of the array.[1] That explains why there sometimes seems to be a logarithmic gap between the best functional and procedural solutions to a problem. But sometimes, as here, the gap vanishes on a closer inspection.

[1] To be fair, procedural programmers also appreciate that constant-time indexing and updating are only possible when the arrays are small.

# 2

# A surpassing problem

## Introduction

In this pearl we solve a small programming exercise of Martin Rem (1988a). While Rem's solution uses binary search, our solution is another application of divide and conquer. By definition, a *surpasser* of an element of an array is a greater element to the right, so $x[j]$ is a surpasser of $x[i]$ if $i < j$ and $x[i] < x[j]$. The *surpasser count* of an element is the number of its surpassers. For example, the surpasser counts of the letters of the string GENERATING are given by

| G | E | N | E | R | A | T | I | N | G |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 2 | 5 | 1 | 4 | 0 | 1 | 0 | 0 |

The maximum surpasser count is six. The first occurrence of letter E has six surpassers, namely N, R, T, I, N and G. Rem's problem is to compute the maximum surpasser count of an array of length $n > 1$ and to do so with an $O(n \log n)$ algorithm.

## Specification

We will suppose that the input is given as a list rather than an array. The function *msc* (short for maximum surpasser count) is specified by

$$\begin{array}{lcl} msc & :: & Ord\ a \Rightarrow [a] \rightarrow Int \\ msc\ xs & = & maximum\ [scount\ z\ zs \mid z : zs \leftarrow tails\ xs] \\ scount\ x\ xs & = & length\ (filter\ (x <)\ xs) \end{array}$$

The value of $scount\ x\ xs$ is the surpasser count of $x$ in the list $xs$ and *tails* returns the *nonempty* tails of a nonempty list in decreasing order of length:[1]

$$\begin{array}{lcl} tails\ [\,] & = & [\,] \\ tails\ (x : xs) & = & (x : xs) : tails\ xs \end{array}$$

The definition of *msc* is executable but takes quadratic time.

[1] Unlike the standard Haskell function of the same name, which returns the possibly empty tails of a possibly empty list.

### Divide and conquer

Given the target complexity of $O(n \log n)$ steps, it seems reasonable to head for a divide and conquer algorithm. If we can find a function *join* so that

$$msc\ (xs +\!\!+ ys) \quad = \quad join\ (msc\ xs)\ (msc\ ys)$$

and *join* can be computed in linear time, then the time complexity $T(n)$ of the divide and conquer algorithm for computing *msc* on a list of length $n$ satisfies $T(n) = 2T(n/2) + O(n)$, with solution $T(n) = O(n \log n)$. But it is fairly obvious that no such *join* can exist: too little information is provided by the single number *msc xs* for any such decomposition.

The minimal generalisation is to start out with the table of *all* surpasser counts:

$$table\ xs \quad = \quad [(z, scount\ z\ zs) \mid z : zs \leftarrow tails\ xs]$$

Then $msc = maximum \cdot map\ snd \cdot table$. Can we find a linear-time *join* to satisfy

$$table\ (xs +\!\!+ ys) \quad = \quad join\ (table\ xs)\ (table\ ys)$$

Well, let us see. We will need the following divide and conquer property of *tails*:

$$tails\ (xs +\!\!+ ys) \quad = \quad map\ (+\!\!+ys)\ (tails\ xs) +\!\!+ tails\ ys$$

The calculation goes as follows:

$$\begin{array}{ll}
 & table\ (xs +\!\!+ ys) \\
= & \quad \{\text{definition}\} \\
 & [(z, scount\ z\ zs) \mid z : zs \leftarrow tails\ (xs +\!\!+ ys)] \\
= & \quad \{\text{divide and conquer property of } tails\} \\
 & [(z, scount\ z\ zs) \mid z : zs \leftarrow map\ (+\!\!+ys)\ (tails\ xs) +\!\!+ tails\ ys] \\
= & \quad \{\text{distributing } \leftarrow \text{ over } +\!\!+\} \\
 & [(z, scount\ z\ (zs +\!\!+ ys)) \mid z : zs \leftarrow tails\ xs] +\!\!+ \\
 & [(z, scount\ z\ zs) \mid z : zs \leftarrow tails\ ys]) \\
= & \quad \{\text{since } scount\ z\ (zs +\!\!+ ys) = scount\ z\ zs + scount\ z\ ys\} \\
 & [(z, scount\ z\ zs + scount\ z\ ys) \mid z : zs \leftarrow tails\ xs] +\!\!+ \\
 & [(z, scount\ z\ zs) \mid z : zs \leftarrow tails\ ys]) \\
= & \quad \{\text{definition of } table \text{ and } ys = map\ fst\ (table\ ys)\} \\
 & [(z, c + scount\ z\ (map\ fst\ (table\ ys))) \mid (z, c) \leftarrow table\ xs] +\!\!+ table\ ys
\end{array}$$

Hence *join* can be defined by

$$\begin{array}{lcl} join\ txs\ tys & = & [(z, c + tcount\ z\ tys) \mid (z, c) \leftarrow txs] +\!\!+ tys \\ tcount\ z\ tys & = & scount\ z\ (map\ fst\ tys) \end{array}$$

The problem with this definition, however, is that *join txs tys* does not take linear time in the length of *txs* and *tys*.

We could improve the computation of *tcount* if *tys* were sorted in ascending order of first component. Then we can reason:

$$\begin{array}{ll} & tcount\ z\ tys \\ = & \quad \{\text{definition of } tcount \text{ and } scount\} \\ & length\ (filter\ (z <)\ (map\ fst\ tys)) \\ = & \quad \{\text{since } filter\ p \cdot map\ f = map\ f \cdot filter\ (p \cdot f)\} \\ & length\ (map\ fst\ (filter\ ((z <) \cdot fst)\ tys)) \\ = & \quad \{\text{since } length \cdot map\ f = length\} \\ & length\ (filter\ ((z <) \cdot fst)\ tys) \\ = & \quad \{\text{since } tys \text{ is sorted on first argument}\} \\ & length\ (dropWhile\ ((z \geq) \cdot fst)\ tys) \end{array}$$

Hence

$$tcount\ z\ tys \quad = \quad length\ (dropWhile((z \geq) \cdot fst)\ tys) \tag{2.1}$$

This calculation suggests it would be sensible to maintain *table* in ascending order of first component:

$$table\ xs \quad = \quad sort\ [(z, scount\ z\ zs) \mid z : zs \leftarrow tails\ xs]$$

Repeating the calculation above, but for the sorted version of *table*, we find that

$$join\ txs\ tys \quad = \quad [(x, c + tcount\ x\ tys) \mid (x, c) \leftarrow txs] \mathbin{\curlywedge} tys \tag{2.2}$$

where $\curlywedge$ merges two sorted lists. Using this identity we can now calculate a more efficient recursive definition of *join*. One of the base cases, namely $join\ [\,]\ tys = tys$, is immediate. The other base case, $join\ txs\ [\,] = txs$, follows because $tcount\ x\ [\,] = 0$. For the recursive case we simplify

$$join\ \ txs@((x, c) : txs')\ \ tys@((y, d) : tys') \tag{2.3}$$

by comparing $x$ and $y$. (In Haskell, the @ sign introduces a synonym, so *txs* is a synonym for $(x, c) : txs'$; similarly for *tys*.) Using (2.2), (2.3) reduces to

$$((x, c + tcount\ x\ tys) : [(x, c + tcount\ x\ tys) \mid (x, c) \leftarrow txs']) \mathbin{\curlywedge} tys$$

To see which element is produced first by $\curlywedge$ we need to compare $x$ and $y$. If $x < y$, then it is the element on the left and, since $tcount\ x\ tys = length\ tys$ by (2.1), expression (2.3) reduces to

$$(x, c + length\ tys) : join\ txs'\ tys$$

If $x = y$, we need to compare $c + tcount\ x\ tys$ and $d$. But $d = tcount\ x\ tys'$ by the definition of *table* and $tcount\ x\ tys = tcount\ x\ tys'$ by (2.1), so (2.3) reduces to $(y, d) : join\ txs\ tys'$. This is also the result in the final case $x > y$.

Putting these results together, and introducing $length\ tys$ as an additional argument to *join* in order to avoid repeating length calculations, we arrive at the following divide and conquer algorithm for *table*:

$$\begin{array}{lcl}
table\ [x] & = & [(x, 0)] \\
table\ xs & = & join\ (m - n)\ (table\ ys)\ (table\ zs) \\
 & & \quad \textbf{where}\ m = length\ xs \\
 & & \qquad\ \ n = m\ \mathrm{div}\ 2 \\
 & & \qquad\ \ (ys, zs) = splitAt\ n\ xs \\
join\ 0\ txs\ [\,] & = & txs \\
join\ n\ [\,]\ tys & = & tys \\
join\ n\ \ txs@((x, c) : txs')\ \ tys@((y, d) : tys') & & \\
\quad \mid\ x < y & = & (x, c + n) : join\ n\ txs'\ tys \\
\quad \mid\ x \geq y & = & (y, d) : join\ (n-1)\ txs\ tys'
\end{array}$$

Since *join* takes linear time, *table* is computed in $O(n \log n)$ steps, and so is *msc*.

## Final remarks

It is not possible to do better than an $O(n \log n)$ algorithm for computing *table*. The reason is that if *xs* is a list of distinct elements, then *table xs* provides enough information to determine the permutation of *xs* that sorts *xs*. Moreover, no further comparisons between list elements are required. In fact, *table xs* is related to the *inversion table* of a permutation of $n$ elements; see Knuth (1998): *table xs* is just the inversion table of *reverse xs*. Since comparison-based sorting of $n$ elements requires $\Omega(n \log n)$ steps, so does the computation of *table*.

As we said in the Introduction for this pearl, the solution in Rem (1998b) is different, in that it is based on an iterative algorithm and uses binary search. A procedural programmer could also head for a divide and conquer algorithm, but would probably prefer an in-place array-based algorithm simply because it consumes less space.