5

Sorting pairwise sums

Introduction

Let A be some linearly ordered set and $(\oplus) :: A \to A \to A$ some monotonic binary operation on A, so $x \leq x' \wedge y \leq y' \Rightarrow x \oplus y \leq x' \oplus y'$. Consider the problem of computing

$$
\begin{array}{lcl}
\mathit{sortsums} & :: & [A] \to [A] \to [A] \\
\mathit{sortsums}\ \mathit{xs}\ \mathit{ys} & = & \mathit{sort}\ [x \oplus y \mid x \leftarrow \mathit{xs},\ y \leftarrow \mathit{ys}]
\end{array}
$$

Counting just comparisons, and supposing *xs* and *ys* have the same length n, how long does *sortsums xs ys* take?

Certainly $O(n^2 \log n)$ comparisons are sufficient. There are n^2 sums and sorting a list of length n^2 can be done with $O(n^2 \log n)$ comparisons. This upper bound does not depend on $\oplus$ being monotonic. In fact, without further information about $\oplus$ and A this bound is also a lower bound. The assumption that $\oplus$ is monotonic does not reduce the asymptotic complexity, only the constant factor.

But now suppose we know more about $\oplus$ and A: specifically that $(\oplus, A)$ is an *Abelian group*. Thus, $\oplus$ is associative and commutative, with identity element e and an operation $\mathit{negate} :: A \to A$ such that $x \oplus \mathit{negate}\ x = e$. Given this extra information, Jean-Luc Lambert (1992) proved that *sortsums* can be computed with $O(n^2)$ comparisons. However, his algorithm also requires $Cn^2 \log n$ additional operations, where C is quite large. It remains an open problem, some 35 years after it was first posed by Harper *et al.* (1975), as to whether the total cost of computing *sortsums* can be reduced to $O(n^2)$ comparisons and $O(n^2)$ other steps.

Lambert's algorithm is another nifty example of divide and conquer. Our aim in this pearl is just to present the essential ideas and give an implementation in Haskell.

Lambert's algorithm

Let's first prove the $\Omega(n^2 \log n)$ lower bound on *sortsums* when the only assumption is that $(\oplus)$ is monotonic. Suppose xs and ys are both sorted into increasing order and consider the $n \times n$ matrix

$$[[x \oplus y \mid y \leftarrow ys] \mid x \leftarrow xs]$$

Each row and column of the matrix is therefore in increasing order. The matrix is an example of a standard Young tableau, and it follows from Theorem H of Section 5.1.4 of Knuth (1998) that there are precisely

$$E(n) = (n^2)! \Big/ \left(\frac{(2n-1)!}{(n-1)!} \frac{(2n-2)!}{(n-2)!} \cdots \frac{n!}{0!} \right)$$

ways of assigning the values 1 to n^2 to the elements of the matrix, and so exactly $E(n)$ potential permutations that sort the input. Using the fact that $\log E(n) = \Omega(n^2 \log n)$, we conclude that at least this number of comparisons is required.

Now for the meat of the exercise. Lambert's algorithm depends on two simple facts. Define the subtraction operation $(\ominus) :: A \to A \to A$ by $x \ominus y = x \oplus \mathit{negate}\ y$. Then:

$$x \oplus y \quad = \quad x \ominus \mathit{negate}\ y \tag{5.1}$$
$$x \ominus y \leq x' \ominus y' \quad \equiv \quad x \ominus x' \leq y \ominus y' \tag{5.2}$$

Verification of (5.1) is easy, but (5.2), which we leave as an exercise, requires all the properties of an Abelian group. In effect, (5.1) says that the problem of sorting sums can be reduced to the problem of sorting subtractions and (5.2) says that the latter problem is, in turn, reducible to the problem of sorting subtractions over a single list.

Here is how (5.1) and (5.2) are used. Consider the list $\mathit{subs}\ xs\ ys$ of *labelled* subtractions defined by

$$\begin{array}{lcl} \mathit{subs} & :: & [A] \to [A] \to [\mathit{Label}\ A] \\ \mathit{subs}\ xs\ ys & = & [(x \ominus y, (i,j)) \mid (x,i) \leftarrow \mathit{zip}\ xs\ [1..],\ (y,j) \leftarrow \mathit{zip}\ ys\ [1..]] \end{array}$$

where $\mathit{Label}\ a$ is a synonym for $(a, (\mathit{Int}, \mathit{Int}))$. Thus, each term $x \ominus y$ is labelled with the position of x in xs and y in ys. Labelling information will be needed later on. The first fact (5.1) gives

$$\begin{array}{lcl} \mathit{sortsums}\ xs\ ys & = & \mathit{map}\ \mathit{fst}\ (\mathit{sortsubs}\ xs\ (\mathit{map}\ \mathit{negate}\ ys)) \\ \mathit{sortsubs}\ xs\ ys & = & \mathit{sort}\ (\mathit{subs}\ xs\ ys) \end{array}$$

The sums are sorted by sorting the associated labelled subtractions and throwing away the labels.

The next step is to exploit (5.2) to show how to compute *sortsubs xs ys* with a quadratic number of comparisons. Construct the list *table* by

$$
\begin{array}{lcl}
\mathit{table} & :: & [A] \to [A] \to [(\mathit{Int}, \mathit{Int}, \mathit{Int})] \\
\mathit{table}\ \mathit{xs}\ \mathit{ys} & = & \mathit{map}\ \mathit{snd}\ (\mathit{map}\ (\mathit{tag}\ 1)\ \mathit{xxs} \curlywedge \mathit{map}\ (\mathit{tag}\ 2)\ \mathit{yys}) \\
 & & \quad \mathbf{where}\ \mathit{xxs} = \mathit{sortsubs}\ \mathit{xs}\ \mathit{xs} \\
 & & \quad \phantom{\mathbf{where}\ } \mathit{yys} = \mathit{sortsubs}\ \mathit{ys}\ \mathit{ys} \\
\mathit{tag}\ i\ (x, (j, k)) & = & (x, (i, j, k))
\end{array}
$$

Here, $\curlywedge$ merges two sorted lists. In words, *table* is constructed by merging the two sorted lists *xxs* and *yys* after first tagging each list in order to be able to determine the origin of each element in the merged list. According to (5.2), *table* contains sufficient information to enable *sortsubs xs ys* to be computed with *no* comparisons over A. For suppose that $x \ominus y$ has label (i, j) and $x' \ominus y'$ has label (k, ℓ). Then $x \ominus y \leq x' \ominus y'$ if and only if $(1, i, k)$ precedes $(2, j, \ell)$ in *table*. No comparisons of elements of A are needed beyond those required to construct *table*.

To implement the idea we need to be able to compute precedence information quickly. This is most simply achieved by converting *table* into a Haskell array:

$$
\begin{array}{lcl}
\mathit{mkArray}\ \mathit{xs}\ \mathit{ys} & = & \mathit{array}\ b\ (\mathit{zip}\ (\mathit{table}\ \mathit{xs}\ \mathit{ys})\ [1..]) \\
 & & \quad \mathbf{where}\ b = ((1, 1, 1), (2, p, p)) \\
 & & \quad \phantom{\mathbf{where}\ } p = \mathit{max}\ (\mathit{length}\ \mathit{xs})\ (\mathit{length}\ \mathit{ys})
\end{array}
$$

The definition of *mkArray* makes use of the library *Data.Array* of Haskell arrays. The first argument b of *array* is a pair of bounds, the lowest and highest indices in the array. The second argument of *array* is an association list of index–value pairs. With this representation, $(1, i, k)$ precedes $(2, j, \ell)$ in *table* if $a!(1, i, k) < a!(2, j, \ell)$, where $a = \mathit{mkArray}\, \mathit{xs}\, \mathit{ys}$. The array indexing operation $(!)$ takes constant time, so a precedence test takes constant time. We can now compute *sortsubs xs ys* using the Haskell utility function *sortBy*:

$$
\begin{array}{lcl}
\mathit{sortsubs}\ \mathit{xs}\ \mathit{ys} & = & \mathit{sortBy}\ (\mathit{cmp}\ (\mathit{mkArray}\ \mathit{xs}\ \mathit{ys}))\ (\mathit{subs}\ \mathit{xs}\ \mathit{ys}) \\
\mathit{cmp}\ a\ (x, (i, j))\ (y, (k, \ell)) & & \\
 & = & \mathit{compare}\ (a\ !\ (1, i, k))\ (a\ !\ (2, j, \ell))
\end{array}
$$

The function *compare* is a method in the type class *Ord*. In particular, $\mathit{sort} = \mathit{sortBy}\ \mathit{compare}$ and $(\curlywedge) = \mathit{mergeBy}\ \mathit{compare}$. We omit the divide and conquer definition of *sortBy* in terms of *mergeBy*.

The program so far is summarised in Figure 5.1. It is complete apart from the definition of $\mathit{sortsubs}'$, where $\mathit{sortsubs}'\ \mathit{xs} = \mathit{sortsubs}\ \mathit{xs}\ \mathit{xs}$. However, this definition cannot be used in *sortsums* because the recursion would not be

$$
\begin{array}{lcl}
\mathit{sortsums}\ \mathit{xs}\ \mathit{ys} & = & \mathit{map}\ \mathit{fst}\ (\mathit{sortsubs}\ \mathit{xs}\ (\mathit{map}\ \mathit{negate}\ \mathit{ys}))\\
\mathit{sortsubs}\ \mathit{xs}\ \mathit{ys} & = & \mathit{sortBy}\ (\mathit{cmp}\ (\mathit{mkArray}\ \mathit{xs}\ \mathit{ys}))\ (\mathit{subs}\ \mathit{xs}\ \mathit{ys})
\end{array}
$$

$$
\mathit{subs}\ \mathit{xs}\ \mathit{ys} \;=\; [(x \ominus y, (i,j)) \mid (x,i) \leftarrow \mathit{zip}\ \mathit{xs}\ [1..], (y,j) \leftarrow \mathit{zip}\ \mathit{ys}\ [1..]]
$$

$$
\mathit{cmp}\ a\ (x,(i,j))\ (y,(k,\ell)) \;=\; \mathit{compare}\ (a\,!\,(1,i,k))\ (a\,!\,(2,j,\ell))
$$

$$
\begin{array}{lcl}
\mathit{mkArray}\ \mathit{xs}\ \mathit{ys} & = & \mathit{array}\ b\ (\mathit{zip}\ (\mathit{table}\ \mathit{xs}\ \mathit{ys})\ [1..])\\
 & & \quad \mathbf{where}\ b \;=\; ((1,1,1),(2,p,p))\\
 & & \quad \phantom{\mathbf{where}}\ p \;=\; \mathit{max}\ (\mathit{length}\ \mathit{xs})\ (\mathit{length}\ \mathit{ys})\\
\mathit{table}\ \mathit{xs}\ \mathit{ys} & = & \mathit{map}\ \mathit{snd}\ (\mathit{map}\ (\mathit{tag}\ 1)\ \mathit{xxs} \wedge\!\!\!\wedge\ \mathit{map}\ (\mathit{tag}\ 2)\ \mathit{yys})\\
 & & \quad \mathbf{where}\ \mathit{xxs} \;=\; \mathit{sortsubs}'\ \mathit{xs}\\
 & & \quad \phantom{\mathbf{where}}\ \mathit{yys} \;=\; \mathit{sortsubs}'\ \mathit{ys}\\
\mathit{tag}\ i\ (x,(j,k)) & = & (x,(i,j,k))
\end{array}
$$

Fig. 5.1 The complete code for *sortsums*, except for *sortsubs′*

well founded. Although computing *sortsubs xs ys* takes $O(mn \log mn)$ steps, it uses no comparisons on A beyond those needed to construct *table*. And *table* needs only $O(m^2 + n^2)$ comparisons plus those comparisons needed to construct *sortsubs′ xs* and *sortsubs′ ys*. What remains is to show how to compute *sortsubs′* with a quadratic number of comparisons.

Divide and conquer

Ignoring labels for the moment and writing $\mathit{xs} \ominus \mathit{ys}$ for $[x \ominus y \mid x \leftarrow \mathit{xs},\ y \leftarrow \mathit{ys}]$, the key to a divide and conquer algorithm is the identity

$$
\begin{array}{l}
(\mathit{xs} +\!\!+ \mathit{ys}) \ominus (\mathit{xs} +\!\!+ \mathit{ys})\\
\quad = \quad (\mathit{xs} \ominus \mathit{xs}) +\!\!+ (\mathit{xs} \ominus \mathit{ys}) +\!\!+ (\mathit{ys} \ominus \mathit{xs}) +\!\!+ (\mathit{ys} \ominus \mathit{ys})
\end{array}
$$

Hence, to sort the list on the left, we can sort the four lists on the right and merge them together. The presence of labels complicates the divide and conquer algorithm slightly because the labels have to be adjusted correctly. The labelled version reads

$$
\begin{array}{l}
\mathit{subs}\ (\mathit{xs} +\!\!+ \mathit{ys})\ (\mathit{xs} +\!\!+ \mathit{ys})\\
\quad = \quad \mathit{subs}\ \mathit{xs}\ \mathit{xs} +\!\!+ \mathit{map}\ (\mathit{incr}\ m)\ (\mathit{subs}\ \mathit{xs}\ \mathit{ys}) +\!\!+\\
\qquad\quad\ \ \mathit{map}\ (\mathit{incl}\ m)\ (\mathit{subs}\ \mathit{ys}\ \mathit{xs}) +\!\!+ \mathit{map}\ (\mathit{incb}\ m)\ (\mathit{subs}\ \mathit{ys}\ \mathit{ys})
\end{array}
$$

where $m = \mathit{length}\ \mathit{xs}$ and

$$
\begin{array}{lcl}
\mathit{incl}\ m\ (x,(i,j)) & = & (x,(m{+}i,j))\\
\mathit{incr}\ m\ (x,(i,j)) & = & (x,(i,m{+}j))\\
\mathit{incb}\ m\ (x,(i,j)) & = & (x,(m{+}i,m{+}j))
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{sortsubs}'\,[\,] & = & [\,] \\
\mathit{sortsubs}'\,[w] & = & [(w \ominus w, (1,1))] \\
\mathit{sortsubs}'\,\mathit{ws} & = & \mathit{foldr1}\;(\wedge\!\!\!\wedge)\;[\mathit{xxs}, \mathit{map}\;(\mathit{incr}\;m)\;\mathit{xys}, \\
 & & \qquad\qquad \mathit{map}\;(\mathit{incl}\;m)\;\mathit{yxs}, \mathit{map}\;(\mathit{incb}\;m)\;\mathit{yys}] \\
\quad \textbf{where}\;\mathit{xxs} & = & \mathit{sortsubs}'\;\mathit{xs} \\
\qquad\quad \mathit{xys} & = & \mathit{sortBy}\;(\mathit{cmp}\;(\mathit{mkArray}\;\mathit{xs}\;\mathit{ys}))\;(\mathit{subs}\;\mathit{xs}\;\mathit{ys}) \\
\qquad\quad \mathit{yxs} & = & \mathit{map}\;\mathit{switch}\;(\mathit{reverse}\;\mathit{xys}) \\
\qquad\quad \mathit{yys} & = & \mathit{sortsubs}'\;\mathit{ys} \\
\qquad\quad (\mathit{xs}, \mathit{ys}) & = & \mathit{splitAt}\;m\;\mathit{ws} \\
\qquad\quad m & = & \mathit{length}\;\mathit{ws}\;\mathrm{div}\;2
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{incl}\;m\;(x,(i,j)) & = & (x,(m+i,j)) \\
\mathit{incr}\;m\;(x,(i,j)) & = & (x,(i,m+j)) \\
\mathit{incb}\;m\;(x,(i,j)) & = & (x,(m+i,m+j))
\end{array}
$$

$$
\mathit{switch}\;(x,(i,j)) = (\mathit{negate}\;x,(j,i))
$$

Fig. 5.2 The code for *sortsubs′*

To compute $\mathit{sortsubs}'\,\mathit{ws}$ we split ws into two equal halves xs and ys. The lists $\mathit{sortsubs}'\,\mathit{xs}$ and $\mathit{sortsubs}'\,\mathit{ys}$ are computed recursively. The list $\mathit{sortsubs}\,\mathit{xs}\,\mathit{ys}$ is computed by applying the algorithm of the previous section. We can also compute $\mathit{sortsubs}\;\mathit{ys}\;\mathit{xs}$ in the same way, but an alternative is simply to reverse $\mathit{sortsubs}\;\mathit{xs}\;\mathit{ys}$ and negate its elements:

$$
\begin{array}{lcl}
\mathit{sortsubs}\;\mathit{ys}\;\mathit{xs} & = & \mathit{map}\;\mathit{switch}\;(\mathit{reverse}\;(\mathit{sortsubs}\;\mathit{xs}\;\mathit{ys})) \\
\mathit{switch}\;(x,(i,j)) & = & (\mathit{negate}\;x,(j,i))
\end{array}
$$

The program for $\mathit{sortsubs}'$ is given in Figure 5.2. The number $C(n)$ of comparisons required to compute $\mathit{sortsubs}'$ on a list of length n satisfies the recurrence $C(n) = 2C(n/2) + O(n^2)$ with solution $C(n) = O(n^2)$. That means $\mathit{sortsums}$ can be computed with $O(n^2)$ comparisons. However, the total time $T(n)$ satisfies $T(n) = 2T(n/2) + O(n^2 \log n)$ with solution $T(n) = O(n^2 \log n)$. The logarithmic factor can be removed from $T(n)$ if $\mathit{sortBy}\;\mathit{cmp}$ can be computed in quadratic time, but this result remains elusive. In any case, the additional complexity arising from replacing comparisons by other operations makes the algorithm very inefficient in practice.

Final remarks

The problem of sorting pairwise sums is given as Problem 41 in the Open Problems Project (Demaine *et al.*, 2009), a web resource devoted to recording open problems of interest to researchers in computational geometry and related fields. The earliest known reference to the problem is Fedman (1976), who attributes the problem to Elwyn Berlekamp. All these references consider the problem in terms of numbers rather than Abelian groups, but the idea is the same.

References

Demaine, E. D., Mitchell, J. S. B. and O'Rourke, J. (2009). The Open Problems Project. http://mave,smith.edu/~orourke/TOPP/.

Fedman, M. L. (1976). How good is the information theory lower bound in sorting? *Theoretical Computer Science* **1**, 355–61.

Harper, L. H., Payne, T. H., Savage, J. E. and Straus, E. (1975). Sorting $X + Y$. *Communications of the ACM* **18** (6), 347–9.

Knuth, D. E. (1998). *The Art of Computer Programming: Volume 3, Sorting and Searching*, second edition. Reading, MA: Addison-Wesley.

Lambert, J.-L. (1992). Sorting the sums (x_i+y_j) in $O(n^2)$ comparisons. *Theoretical Computer Science* **103**, 137–41.