COMP50001: Algorithm Design & Analysis

Sheet 3 (Week 4)

Exercise 3.1

qsort uses $+\!\!+$ on lists to build up the output list; reimplement it to use *DList*s. Compare the running time (both asymptotic and absolute) of the new *qsort* to the old.

$$
\begin{aligned}
&qsort :: [Int] \to [Int]\\
&qsort\ [\,] = [\,]\\
&qsort\ [x] = [x]\\
&qsort\ (x : xs) = qsort\ us +\!\!+ [x] +\!\!+ qsort\ vs\\
&\quad \mathbf{where}\\
&\qquad (us, vs) = partition\ (\leqslant x)\ xs
\end{aligned}
$$

Exercise 3.2

Identify and prove that one of the base cases in the given definition of *qsort* is unnecessary.

Exercise 3.3

Recall that quick sort is $\mathcal{O}(n^2)$ on sorted lists. With this in mind, often we employ more sophisticated methods of choosing a pivot: the median-of-three approach, for instance, picks the first, middle, and last elements in the input list and chooses the median of those three as the pivot. Calculate the time complexity of quick sort on sorted lists when median-of-three is used. Describe a pathological case, if one exists, where quick sort is $\mathcal{O}(n^2)$ even when median-of-three is used.

Exercise 3.4

Consider the following implementation of merge sort on nonempty lists (equivalent to the implementation given in lectures):

$$
\begin{aligned}
&merge :: [Int] \to [Int] \to [Int]\\
&merge\ [\,]\ ys = ys\\
&merge\ xs\ [\,] = xs\\
&merge\ xxs@(x : xs)\ yys@(y : ys)\\
&\quad |\ x \leqslant y = x : merge\ xs\ yys\\
&\quad |\ otherwise = y : merge\ xxs\ ys
\end{aligned}
$$

$$
\begin{aligned}
&msort :: [Int] \to [Int]\\
&msort = foldt\ merge \circ map\ single\\
&foldt :: (a \to a \to a) \to [a] \to a\\
&foldt\ \_\ [x] = x\\
&foldt\ f\ xs = f\ (foldt\ f\ ys)\ (foldt\ f\ zs)\\
&\quad \mathbf{where}\\
&\qquad (ys, zs) = splitAt\ (length\ xs\ `div`\ 2)\ xs
\end{aligned}
$$

Rewrite *foldt* to be bottom-up (rather than top-down): it should merge adjacent elements in the input list repeatedly until only one is left, and then return it.

Exercise 3.5

Calculate the time complexity of *msort* if the following definition of *foldt* had been used:

$$
\begin{aligned}
&foldt\ f\ [x] = x\\
&foldt\ f\ (x : xs) = f\ x\ (foldt\ f\ xs)
\end{aligned}
$$

State the name of the sorting algorithm implemented by *msort* if this definition of *foldt* is used.

Exercise 3.6

Recall that we defined $minimum = head \circ isort$. Calculate the (worst-case) running time of $minimum = head \circ qsort$ (ADWH, p111, Exercise 5.5) and $minimum = head \circ msort$.

Exercise 3.7

(ADWH, p112, Exercise 5.7) The number of comparisons $T(m, n)$ required by *merge* to merge two lists of lengths m and n in the worst case satisfies

$$\begin{aligned} T(0, n) &= 0 \\ T(m,0) &= 0 \\ T(m,n) &= 1 + T(m-1, n) \max T(m, n-1) \end{aligned}$$

Prove that $T(m, n) \leqslant m + n$.

Exercise 3.8

lcs finds the longest common subsequence of two lists of *Int*s.

$$\begin{aligned} &lcs\ [1,2,3]\ [1,3] &&\equiv [1,3] \\ &lcs\ [1,2,3]\ [4,5,6] &&\equiv [\,] \\ &lcs\ [1,2,3]\ [3,2,1] &&\equiv [3] \end{aligned}$$

Reimplement *lcs* to use memoisation.

```
lcs :: [Int] → [Int] → [Int]
lcs xxs@(x : xs) yys@(y : ys)
  | x ≡ y                    = x : lcs xs ys
  | length us ⩽ length vs    = vs
  | otherwise                = us
  where
    us = lcs xxs ys
    vs = lcs xs yys
lcs _ _ = []
```

Exercise 3.9

Given a list of integers *xs*, and some index *i* into *xs*, we define *splitDiff xs i* to be the sum of all the integers up to *i* minus the sum of those from *i* onwards.

Write an algorithm which, given some *xs*, finds an *i* which maximises *splitDiff xs i*. Your algorithm should run in linear time. The following is a quadratic-time solution:

```
splitDiff :: [Int] → Int → Int
splitDiff xs i = sum lhs − sum rhs
  where
    (lhs, rhs) = splitAt i xs
```

```
maximise :: [Int] → Int
maximise xs =
  maximumBy (comparing (splitDiff xs)) [0 .. length xs − 1]
```

Exercise 3.10

Consider the following implementation of *fib* which uses the *fixer* helper function:

```
fixer :: ((Int → a) → (Int → a)) → Int → a
fixer f = f (fixer f)
```

```
fib :: Int → Int
fib = fixer go
  where
    go r 0 = 0
    go r 1 = 1
    go r n = r (n − 1) + r (n − 2)
```

Reimplement *fixer* such that the resulting *fib* will be memoised.

Solutions to the Exercises

Solution 3.1

Using *DList* directly, we get the following:

$$
\begin{aligned}
&\mathit{qsort} :: [\mathit{Int}] \to [\mathit{Int}] \\
&\mathit{qsort} = \mathit{toList} \circ \mathit{go} \\
&\quad \textbf{where} \\
&\qquad \mathit{go} :: [\mathit{Int}] \to \mathit{DList}\ \mathit{Int} \\
&\qquad \mathit{go}\ [\,] \qquad\quad = \mathit{empty} \\
&\qquad \mathit{go}\ (x : \mathit{xs}) = \mathit{go}\ \mathit{us} +\!\!+ \mathit{single}\ x +\!\!+ \mathit{go}\ \mathit{vs} \\
&\qquad\quad \textbf{where} \\
&\qquad\qquad (\mathit{us}, \mathit{vs}) = \mathit{partition}\ (\leqslant x)\ \mathit{xs}
\end{aligned}
$$

If we replaced all of the class methods with their implementations, we would arrive at the following function:

$$
\begin{aligned}
&\mathit{qsort} :: [\mathit{Int}] \to [\mathit{Int}] \\
&\mathit{qsort}\ \mathit{xs} = \mathit{go}\ \mathit{xs}\ [\,] \\
&\quad \textbf{where} \\
&\qquad \mathit{go}\ [\,] \qquad\quad = \mathit{id} \\
&\qquad \mathit{go}\ (x : \mathit{xs}) = \mathit{go}\ \mathit{us} \circ (x{:}) \circ \mathit{go}\ \mathit{vs} \\
&\qquad\quad \textbf{where} \\
&\qquad\qquad (\mathit{us}, \mathit{vs}) = \mathit{partition}\ (\leqslant x)\ \mathit{xs}
\end{aligned}
$$

And if we eta-expand the *go* helper, we arrive at the following:

$$
\begin{aligned}
&\mathit{qsort} :: [\mathit{Int}] \to [\mathit{Int}] \\
&\mathit{qsort}\ \mathit{xs} = \mathit{go}\ \mathit{xs}\ [\,] \\
&\quad \textbf{where} \\
&\qquad \mathit{go}\ [\,] \qquad\quad \mathit{ks} = \mathit{ks} \\
&\qquad \mathit{go}\ (x : \mathit{xs})\ \mathit{ks} = \mathit{go}\ \mathit{us}\ (x : \mathit{go}\ \mathit{vs}\ \mathit{ks}) \\
&\qquad\quad \textbf{where} \\
&\qquad\qquad (\mathit{us}, \mathit{vs}) = \mathit{partition}\ (\leqslant x)\ \mathit{xs}
\end{aligned}
$$

This final function reveals that the *DList* optimisation is analogous to adding an accumulator to the recursive function, similarly to how the naive $\mathcal{O}(n^2)$ *reverse* on lists can be improved to $\mathcal{O}(n)$ with an accumulator.

With regards to the running time, the line of interest is the following:

$$\mathit{qsort}\ (x : \mathit{xs}) = \underline{\mathit{qsort}\ \mathit{us}} +\!\!+ [x] +\!\!+ \mathit{qsort}\ \mathit{vs}$$

In the list-based algorithm, we would need to pay for the traversal of the underlined portion, since $+\!\!+$ is linear in its first argument. In the *DList*-based solution, on the other hand, $+\!\!+$ is constant-time, so we don't have to pay the extra cost. There is an extra linear-time step in the *DList* solution (the *toList* at the end), but we can skip this step by using the final version of *qsort* above with the accumulator, meaning that overall the new *qsort* is strictly better than the old in terms of absolute time.

The asympotics, however, do not change: this is because $qsort\ us$ already costs $\mathcal{O}(n\log(n))$, so the extra linear traversal incurred by $+\!\!+$ doesn't make a difference.

Solution 3.2

The second clause ($qsort\ [x] = [x]$) is unnecessary. This means that we can write *qsort* as follows:

$$
\begin{aligned}
&qsort' :: [Int] \to [Int]\\
&qsort'\ [\,] \qquad = [\,]\\
&qsort'\ (x : xs) = qsort'\ us +\!\!+ [x] +\!\!+ qsort'\ vs\\
&\quad \textbf{where}\\
&\qquad (us, vs) = partition\ (\leqslant x)\ xs
\end{aligned}
$$

To prove that the clause is unnecessary we need only show that, on the input that matches that clause, $qsort'$ and $qsort$ are equal.

$$
\begin{aligned}
&\quad qsort'\ [x]\\
\equiv&\quad \{\text{ Evaluate } qsort'\ [x]\ \}\\
&\quad qsort'\ us +\!\!+ [x] +\!\!+ qsort'\ vs\\
&\qquad \textbf{where}\\
&\qquad\quad (us, vs) = partition\ (\leqslant x)\ [\,]\\
\equiv&\quad \{\text{ Evaluate } partition\ (\leqslant x)\ [\,]\ \}\\
&\quad qsort'\ us +\!\!+ [x] +\!\!+ qsort'\ vs\\
&\qquad \textbf{where}\\
&\qquad\quad (us, vs) = ([\,], [\,])\\
\equiv&\quad \{\text{ Variable substitution for } us \text{ and } vs\ \}\\
&\quad qsort'\ [\,] +\!\!+ [x] +\!\!+ qsort'\ [\,]\\
\equiv&\quad \{\text{ Evaluate } qsort'\ [\,]\ \}\\
&\quad [\,] +\!\!+ [x] +\!\!+ [\,]\\
\equiv&\quad \{\ +\!\!+ \text{ left and right identities }\}\\
&\quad [x]\\
\equiv&\quad \{\text{ Definition of } qsort\ \}\\
&\quad qsort\ [x]
\end{aligned}
$$

Solution 3.3

Quick sort with median-of-three pivot selection is $\mathcal{O}(n\log(n))$ on sorted lists.

The key to generating a pathological case for median-of-three is to try and make it so that the pivot chosen is always either larger or smaller than most other elements in the list, so the split is skewed one way or the other. As such, we should place the two smallest (or largest) elements in the first and middle positions in the list. A function which would produce this pathological case (from a sorted list with distinct elements) is the following:

```haskell
pathologise :: [Int] -> [Int]
pathologise xxs@(x1 : x2 : xs) = [x1] ++ lhs ++ [x2] ++ rhs
  where
    n = length xxs
    ys = pathologise xs
    (lhs, rhs) = splitAt (n `div` 2 - 1) ys
pathologise xs = xs
```

Running the median-of-three version of *qsort* on *pathologise* $[x_1, \ldots, x_n]$ gives rises to

The *pathologise* function assumes that the middle element is picked by *xs* !! (*length xs* ‘*div*’ 2).

$$
\begin{aligned}
& \mathit{qsort}_{m3}\ (\mathit{pathologise}\ [x_1, x_2, \ldots, x_n]) \\
\equiv\ & \mathit{qsort}_{m3}\ ([x_1] +\!\!+ \mathit{lhs} +\!\!+ [x_2] +\!\!+ \mathit{rhs}) \\
\equiv\ & \text{\{-The index of } x_2 \text{ is exactly } n \text{ `div` } 2 \text{ and any elements -\}} \\
& \text{\{-in } \mathit{rhs} \text{ is greater than } x_2\text{. -\}} \\
& \mathit{qsort}_{m3}\ [x_1] +\!\!+ [x_2] +\!\!+ \mathit{qsort}_{m3}\ (\mathit{lhs} +\!\!+ \mathit{rhs}) \\
\equiv\ & \mathit{qsort}_{m3}\ [x_1] +\!\!+ [x_2] +\!\!+ \mathit{qsort}_{m3}\ (\mathit{pathologise}\ [x_3, \ldots, x_n]) \\
\equiv\ & \ldots
\end{aligned}
$$

and it indicates that qsort_{m3} only shrinks the size of the input by 2 for each step.

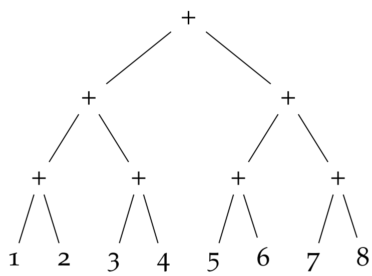
Solution 3.4

There are many different implementations of a bottom-up *foldt*. Here is a simple implementation:

```haskell
foldt :: (a -> a -> a) -> [a] -> a
foldt f [x] = x
foldt f xs  = foldt f (pairMap f xs)
  where
    pairMap f (x1 : x2 : xs) = f x1 x2 : pairMap f xs
    pairMap _ xs             = xs
```

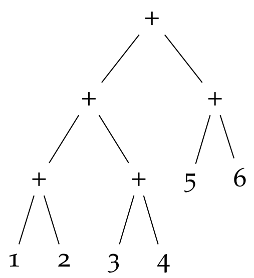
It’s important that the implementation of *foldt* builds a balanced tree in order to maintain the asymptotic complexity of *msort*. For instance, for the given *foldt*, we have:

$\mathit{foldt}\ (+)\ [1, 2, 3, 4, 5, 6, 7, 8] =$



In the cases where a perfect tree is not possible, it balances the tree as follows:

$foldt\ (+)\ [1,2,3,4,5,6] =$



However, even though the second tree here is unbalanced, the depth of the tree is always of order $\mathcal{O}(\log(n))$, so the asymptotic complexity of *msort* is $\mathcal{O}(n \log(n))$.

Solution 3.5

This version of *msort* is actually insertion sort, and so has a time complexity of $\mathcal{O}(n^2)$.

Solution 3.6

In a strict evaluation context, both of these implementations for *minimum* have the same running time as their corresponding sort function. So $head \circ qsort \in \mathcal{O}(n^2)$ and $head \circ msort \in \mathcal{O}(n \log n)$.

In a lazy context the question is more complicated. First, the worst-case time for $head \circ qsort$ is still $\mathcal{O}(n^2)$: to see this, consider the pathological case of a list sorted in reverse, where we always choose a pivot larger than all the other elements. In this case, the call to *partition* will always partition the list into an empty list and a list of all the elements smaller than the pivot. This means that we will have n recursive calls, with each call performing the linear-time partition on its input list, giving us an $\mathcal{O}(n^2)$ function overall. Crucially, no cons-cell is constructed until we perform all of these recursive calls; as a result, laziness can't save us from the quadratic cost.

Interestingly, in the case of a list sorted in ascending order $minimum = head \circ qsort$ is actually $\mathcal{O}(n)$, despite the fact that this is a pathological $\mathcal{O}(n^2)$ case for *qsort*.

To deduce the complexity of $head \circ msort$ we will first look at *merge*.

$$
\begin{array}{l}
merge :: [Int] \to [Int] \to [Int] \\
merge\ [\,]\ ys = ys \\
merge\ xs\ [\,] = xs \\
merge\ xxs@(x : xs)\ yys@(y : ys) \\
\quad \mid x \leqslant y \qquad\ \ = x : merge\ xs\ yys \\
\quad \mid otherwise = y : merge\ xxs\ ys
\end{array}
$$

We know that we are eventually going to call *head* on the returned list, so we can actually *ignore* the second argument to the (:) constructor here (since *head* doesn't look at it). This changes the function to the following:

$$
\begin{aligned}
&\mathit{merge}\ [\,]\ \mathit{ys} = \mathit{ys}\\
&\mathit{merge}\ \mathit{xs}\ [\,] = \mathit{xs}\\
&\mathit{merge}\ (x{:}\ \_)\ (y{:}\ \_)\\
&\quad \mid x \leqslant y \quad\ \ = [x]\\
&\quad \mid \mathit{otherwise} = [y]
\end{aligned}
$$

Now, since we're not calling *merge* on the tail of either of the lists, we know that it will never be called on an empty list, so we can discard the first two clauses:

$$
\begin{aligned}
&\mathit{merge}\ (x{:}\ \_)\ (y{:}\ \_)\\
&\quad \mid x \leqslant y \quad\ \ = [x]\\
&\quad \mid \mathit{otherwise} = [y]
\end{aligned}
$$

Let's now sub in the implementation of *msort* into the definition of *minimum*:

$$\mathit{head} \circ \mathit{msort} = \mathit{head} \circ \mathit{foldt}\ \mathit{merge} \circ \mathit{map}\ \mathit{single}$$

We can see that the *map single* is simply converting every element to a singleton list, and the *head* is converting *from* a singleton list, so we can actually remove these transformations:

We're allowed to remove *map single* and *head* without worrying about how they affect the complexity because *map single* is linear and *head* is constant-time, and we know that *minimum* is at best linear-time, so neither function will make the asymptotic complexity worse.

$$
\begin{aligned}
&\mathit{merge}\ x\ y\\
&\quad \mid x \leqslant y \quad\ \ = x\\
&\quad \mid \mathit{otherwise} = y\\
&\mathit{head} \circ \mathit{msort} = \mathit{foldt}\ \mathit{merge}
\end{aligned}
$$

At this point it's clear that *merge* is just an incorrectly-named *min*:

$$\mathit{head} \circ \mathit{msort} = \mathit{foldt}\ \mathit{min}$$

This final expression will give us our complexity: *foldt f xs* calls *f* *length xs* $- 1$ times, giving *foldt min* linear complexity overall.

Solution 3.7

To prove that $T(m, n) \leqslant m + n$, we proceed by case analysis on m and n. In the case that $m = 0$, the property holds, as $T(0, n) = 0$, and $0 \leqslant m + 0$. Similarly, when $n = 0$, $T(m, 0) = 0$ and $0 \leqslant 0 + n$. In the recursive case, we must prove

$$1 + T(m-1, n)\ \max\ T(m, n-1) \leqslant m + n$$

By induction we know that

$$T(m-1, n) \leqslant (m-1) + n$$

and

$$T(m, n-1) \leqslant m + (n-1)$$

so

$$T(m-1, n)\ \max\ T(m, n-1) \leqslant m + n - 1$$

By adding one to both sides we get

$$1 + T(m-1, n)\ \max\ T(m, n-1) \leqslant m + n$$

Which means the property holds in all cases.

Solution 3.8

The solution uses the *tabulate* function:

$$\begin{array}{l}
tabulate :: Ix\ i \Rightarrow (i,i) \rightarrow (i \rightarrow a) \rightarrow Array\ i\ a \\
tabulate\ (u,v)\ f = array\ (u,v)\ [(i, f\ i) \mid i \leftarrow range\ (u,v)]
\end{array}$$

The first step is to transform *lcs* so that the lists stay constant and to use indices to track how far through the list progress is being made. Here the indices i and j represent the values at the head of the lists being considered. The empty cases are being considered when $i \equiv m$ or $j \equiv n$.

$$\begin{array}{l}
lcs' :: [Int] \rightarrow [Int] \rightarrow Int \rightarrow Int \rightarrow [Int] \\
lcs'\ xs\ ys\ i\ j \\
\quad \mid i \equiv m \vee j \equiv n \qquad\qquad = [\,] \\
\quad \mid x \equiv y \qquad\qquad\qquad\quad\ \ = x : lcs'\ xs\ ys\ (i+1)\ (j+1) \\
\quad \mid length\ us \leqslant length\ vs = vs \\
\quad \mid otherwise \qquad\qquad\quad\ \ \ = us \\
\quad \textbf{where} \\
\qquad us = lcs'\ xs\ ys\ i\ (j+1) \\
\qquad vs = lcs'\ xs\ ys\ (i+1)\ j \\
\qquad x = xs\ !!\ i \\
\qquad y = ys\ !!\ j \\
\qquad m = length\ xs \\
\qquad n = length\ ys
\end{array}$$

From here, the transformation is routine: the *memo* function is essentially lcs', but with recursive calls replaced with a lookup in the *table*. The solution is found in the $(0,0)$ entry.

$$\begin{array}{l}
lcs'' :: [Int] \rightarrow [Int] \rightarrow [Int] \\
lcs''\ xs\ ys = table\ !\ (0,0) \\
\quad \textbf{where} \\
\qquad table = tabulate\ ((0,0),(m,n))\ memo \\
\qquad memo\ (i,j) \\
\qquad\quad \mid i \equiv m \vee j \equiv n \qquad\qquad = [\,] \\
\qquad\quad \mid x \equiv y \qquad\qquad\qquad\quad\ \ = x : table\ !\ (i+1, j+1) \\
\qquad\quad \mid length\ us \leqslant length\ vs = vs \\
\qquad\quad \mid otherwise \qquad\qquad\quad\ \ \ = us \\
\qquad\quad \textbf{where} \\
\qquad\qquad x = xs\ !!\ i \\
\qquad\qquad y = ys\ !!\ j \\
\qquad\qquad us = table\ !\ (i, j+1) \\
\qquad\qquad vs = table\ !\ (i+1, j) \\
\qquad m = length\ xs \\
\qquad n = length\ ys
\end{array}$$

The complexity is then further improved by replacing the calls to *length us* and *length vs* with a lookup, rather than recalculation. This involves modifying the table to store a pair rather than a single value, where one component of the pair is the list, and the other is the length of that list. In addition, the lists *xs* and *ys* can be used to create array version *axs* and *ays* so that looking up values there also takes constant time.

With all this completed, the complexity is easy enough to see: given lists of size m and n each entry in the table takes constant time to compute, and there are $m \times n$ entries in the table. Thus, this is an $O(m \times n)$ algorithm.

Solution 3.9

As in question 3.8, the first step is to redefine the function in question to be more amenable to memoisation. This means replacing its parameters with values that can be keys into some memo table. The function we want to optimise here is actually *splitDiff*: the quadratic complexity comes from calling it a linear amount of times, so if we can build a memo table in $\mathcal{O}(n)$ time *splitDiff* will become $\mathcal{O}(1)$, and the whole function will be linear.

$$
\begin{aligned}
&\mathit{maximise} :: [\mathit{Int}] \rightarrow \mathit{Int} \\
&\mathit{maximise}\ \mathit{xs} = \\
&\qquad \mathit{maximumBy}\ (\mathit{comparing}\ \mathit{splitDiff}')\ [0\,..\,\mathit{length}\ \mathit{xs} - 1] \\
&\quad \textbf{where} \\
&\qquad \mathit{splitDiff}'\ 0 = -\mathit{sum}\ \mathit{xs} \\
&\qquad \mathit{splitDiff}'\ i\ = \mathit{splitDiff}'\ (i-1) + (\mathit{xs}\ !!\ (i-1)) * 2
\end{aligned}
$$

splitDiff' here has two cases: the first (when $i = 0$) corresponds to *splitDiff xs* 0:

$$
\begin{aligned}
& \mathit{splitDiff}\ \mathit{xs}\ 0 \\
\equiv\ & \{\ \text{Definition of } \mathit{splitDiff}\ \} \\
& \mathit{sum}\ \mathit{lhs} - \mathit{sum}\ \mathit{rhs} \\
& \quad \textbf{where} \\
& \qquad (\mathit{lhs}, \mathit{rhs}) = \mathit{splitAt}\ 0\ \mathit{xs} \\
\equiv\ & \{\ \text{Evaluate } \mathit{splitAt}\ \} \\
& \mathit{sum}\ \mathit{lhs} - \mathit{sum}\ \mathit{rhs} \\
& \quad \textbf{where} \\
& \qquad (\mathit{lhs}, \mathit{rhs}) = ([\,], \mathit{xs}) \\
\equiv\ & \{\ \text{Sub in for } \mathit{lhs} \text{ and } \mathit{rhs}\ \} \\
& \mathit{sum}\ [\,] - \mathit{sum}\ \mathit{xs} \\
\equiv\ & \{\ \text{Evaluate } \mathit{sum}\ [\,]\ \} \\
& -\mathit{sum}\ \mathit{xs} \\
\equiv\ & \{\ \text{Definition of } \mathit{splitDiff}'\ \} \\
& \mathit{splitDiff}'\ 0
\end{aligned}
$$

The second case (the recurrence relation) is a little more complex. We need to build the result of *splitDiff' i* from *splitDiff'* $(i-1)$. To do that we will use the following identity (where x_i means *xs* !! *i*):

$$splitDiff'\ i = sum\ [..., x_{i-3}, x_{i-2}, x_{i-1}] - sum\ [x_i, x_{i+1}, x_{i+2}, ...]$$

From which we can derive the following:

$$splitDiff'\ (i-1) = sum\ [..., x_{i-4}, x_{i-3}, x_{i-2}] - sum\ [x_{i-1}, x_i, x_{i+1}, ...]$$

From these we can derive the recurrence relation we need:

$$splitDiff'\ i = splitDiff'\ (i-1) + x_{i-1} * 2$$

With this form of *splitDiff′* the memoisation is much easier to figure out (we also will make an array from the input list to give us quicker lookups):

```
maximise :: [Int] → Int
maximise xs =
    maximumBy (comparing (table!)) [0 .. length xs − 1]
  where
    table = tabulate (0, length xs − 1) memo
    memo 0 = −sum xs
    memo i  = table ! (i − 1) + axs ! (i − 1) ∗ 2
    axs = listArray (0, length xs − 1) xs
```

Finally, there is actually a far simpler function which accomplishes the same task as the two above, also in linear time. Technically it does so using a form of memoisation, although it's probably a little more difficult to see. Its definition is as follows:

```
maximise :: [Int] → Int
maximise xs = fst (maximumBy (comparing snd) (zip [0 ..] sums))
  where
    sums = init (zipWith (−) (scanl (+) 0 xs) (scanr (+) 0 xs))
```

Solution 3.10

Here is a version that uses the *tabulate* function:

```
fixer :: ((Int → a) → (Int → a)) → Int → a
fixer f n = table ! n
  where
    table    = tabulate (0, n) (f memo)
    memo i = table ! i
```

And here is one which uses an infinite list:

```
fixer f = memo
  where
    table    = map (f memo) [0 ..]
    memo i = table !! i
```

These functions have quite different performance characteristics: the former has the advantage of avoiding linear-time lookups in singly-linked lists, meaning that it is much faster in the general case. The latter has the (small) advantage of being able to store results between calls, meaning that *fib* 3 and *fib* 5 in different parts of the program would use the same memo table.

The latter function is able to store results because it first creates an unbounded table for results: laziness ensures that this table is only computed as needed. The former function needs to allocate the space for an array ahead of time, and as a result needs to know a bound on the table; it uses the input to *fib* to do this.

There are data structures which achieve something of the best of both worlds: they can be unbounded and computed on-demand, but with reasonably efficient lookups. In the case of integer keys, we could have used a trie instead of a list to store results. This would have given logarithmic lookups, while also allowing on-demand growth and allocation. In some scenarios this second property is quite important: we don't always know how big our memo table will need to be before we start.