Algorithms I – supervision 2

James Wood

February 2, 2015

1 Heap sort

(a) A heap of height h must contain at least $\left(\sum_{i \in [0..h)} 2^i\right) + 1$, which is 2^h, elements; it can contain at most $\sum_{i \in [0..h]} 2^i$, which is $2^{1+h} - 1$, elements.

(b)
- Heap sort is not stable, because `heapify` tends to mix up elements found at the bottom of the heap.
- Insert sort, selection sort, bubble sort, mergesort, quicksort, counting sort, bucket sort and radix sort are stable.
- Binary insert sort is not stable, but can be made stable by making sure that inserted elements are inserted at the end of a run of elements equal to it (costing a few comarisons, but saving on swaps).

(c) When given a list that is already sorted, construction of the maxheap will run in $\Theta(n \cdot \lg n)$ time, since each of the n elements has to be moved one level at a time from the bottom to the top of a binary tree of height approaching $\lg n$. The worst case is at least as bad as this, and so runs in $\Omega(n \cdot \lg n)$ time.

2 Counting sort

The complexity of counting sort can be described as $O(r \cdot n)$, where r is the number of elements in the range. When $r \gg n$, as in many situations, counting sort will have worse than quadratic complexity.

3 Fibonacci's sequence

```
-- Idris
fib : Nat → Nat
fib Z          = 1
fib (S Z)      = 1
fib (S (S n)) = fib n + fib (S n)
```

This can be made to count how many times it calls itself by modifying it as follows.

```
data Counted a = MkCounted Nat a
instance Functor Counted where
  map m (MkCounted k x) = MkCounted k (m x)
instance Applicative Counted where
  pure x = MkCounted 0 x
  (<$>) (MkCounted k x) (MkCounted j y) = MkCounted (S k + j) (x y)

fibC : Nat → Counted Nat
fibC Z          = [| 1 |]
fibC (S Z)      = [| 1 |]
fibC (S (S n)) = [| fibC n + fibC (S n) |]
```

`fibC 10` yields `MkCounted 176 89` (176 recursive calls) and `fibC 20` (after some time) yields `MkCounted 21890 10946` (21890 recursive calls). Trying to evaluate `fibC 30` has already frozen my computer for long periods twice, so instead I will write a new function to calculate it:

```
fibC' : Nat → Int
fibC' = f (0,0)
  where
    f : (Int,Int) → Nat → Int
    f (a,b) Z = a
    f (a,b) (S k) = f (b,2 + a + b) k
```

With this, `fibC' 30` yields `2692536`.

4 Dynamic programming

(a) The expression $\begin{pmatrix} x_0 & x_1 & x_2 \end{pmatrix} \cdot \left(\begin{pmatrix} y_0 & y_1 \\ y_2 & y_3 \\ y_4 & y_5 \end{pmatrix} \cdot \begin{pmatrix} z_0 \\ z_1 \end{pmatrix} \right)$ reduces to $(x_0 \cdot (y_0 \cdot z_0 + y_1 \cdot z_1) + x_1 \cdot (y_2 \cdot z_0 + y_3 \cdot z_1) + x_2 \cdot (y_4 \cdot z_0 + y_4 \cdot z_1))$, which contains

9 multiplications. The expression $\left(\begin{pmatrix} x_0 & x_1 & x_2 \end{pmatrix} \cdot \begin{pmatrix} y_0 & y_1 \\ y_2 & y_3 \\ y_4 & y_5 \end{pmatrix}\right) \cdot \begin{pmatrix} z_0 \\ z_1 \end{pmatrix}$ reduces to $((x_0 \cdot y_0 + x_1 \cdot y_2 + x_2 \cdot y_4) \cdot z_0 + (x_0 \cdot y_1 + x_1 \cdot y_3 + x_2 \cdot y_5) \cdot z_1)$, which contains 8 multiplications.

(b) A full parenthesization of a 1-element expression uses no parentheses, since there are no operations to surround. A full parenthesization of an n-element expression, where $n > 1$, contains a full parenthesization of the first m elements and the last $n - m$ elements ($m \in (0..n)$), plus a pair of parentheses around the entire expression. Assuming that the statement is true for all suitable m, this gives a total of $m - 1 + n - m - 1 + 1$, which is $n - 1$, pairs of parentheses. Hence, the statement is true by strong induction.

(c)

```
data MonotonicFlag a = Increasing a | Decreasing a

def lmis(xs):
    n = len(xs)
    cache = []

    def fitIn(x, ys):
        # Modifies cache entry
        case ys:
            Increasing zs:
                if zs[$-1$] $\leq$ x:
                    zs = zs ++ [x]
                    (True, False)
                else if len(zs) > 1 $\wedge$ zs[$-2$] < x:
                    zs[$-1$] = x
                    (True, False)
                else:
                    (False, False)
            Decreasing zs:
                if zs[$-1$] $\geq$ x:
                    zs = zs ++ [x]
                    (False, True)
                else if len(zs) > 1 $\wedge$ zs[$-2$] > x:
                    zs[$-1$] = x
                    (False, True)
                else:
                    (False, False)

    for x in xs:
        inIncreasing = False
        inDecreasing = False
        for ys in cache:
            (i, d) = fitIn(x, ys)  # cache modified
            inIncreasing = i $\vee$ inIncreasing
            inDecreasing = d $\vee$ inDecreasing
        if not inIncreasing:
            cache = Increasing [x] :: cache
        if not inDecreasing:
            cache = Decreasing [x] :: cache
```

I don't know whether that runs in $O(n \cdot \lg n)$, but `cache` should grow more slowly than `n`, suggesting a $\lg n$ size.

5 Comparisons and sorting

(a) First, compare non-overlapping pairs of elements ($\lfloor n/2 \rfloor$ comparisons). $\lfloor n/2 \rfloor$ elements are deemed smaller, and are candidates for being the minimum; and $\lfloor n/2 \rfloor$ elements are deemed bigger, and are candidates for being the maximum. If the list has an odd number of items, the uncompared item can go in either group, giving that group $\lceil n/2 \rceil$ elements. Now, the minimum of the smaller elements and the maximum of the larger elements are found, taking $\lfloor n/2 \rfloor - 1$ and $\lceil n/2 \rceil - 1$ comparisons. Hence, there is a total of $\lfloor n/2 \rfloor + (\lfloor n/2 \rfloor - 1) + (\lceil n/2 \rceil - 1)$, which is $2 \cdot \lfloor n/2 \rfloor + \lceil n/2 \rceil - 2$, comparisons.

(b) An ordered list has $n!$ possible permutations, and a sorting algorithm must gain enough information to choose one of these. After k swaps, an optimal algorithm has enough information to distinguish 2^k permutations. Hence, solve $2^k = n!$ for k, giving:

$$k \approx \lg(n!)$$
$$k \approx \lg(n^n)$$
$$k \approx n \cdot \lg n$$

QED.

6 Greedy algorithms

(a) Choose the furthest-away gas station within n miles of the start. Then, plan the journey from this gas station onwards (recursively). This succeeds because the furthest-away gas station is n miles or less before all of the gas stations that any preceeding gas station is n miles or less before. Hence, the options after greedily choosing this station are a superset of any of the options yeilded by picking any other gas station to stop at.

(b)

```
def knapsack(wvs, w):
    # wvs is a dictionary stored as a binary search tree where weights are the keys.
    # cache stores the maximum value achived and the indices of the items used to achieve
    # it for the given weight using the given item
    cache = w * [len(wvs) * [(0, [])]]

    for i in [1:w + 1]:
        for (w, v) in wvs indexed by j:
            if w == i:
                cache[i][j] = (v, [j])
            else:
                (va, pa) = maximumBy(\(vx, _) (vy, _). compare(vx, vy),
                                     filter(\(_, p). j not in p, cache[i - w]))
                (vb, pb) = cache[i - 1][j]
                if vb ≤ v + va:
                    cache[i][j] = (vb, pb)
                else:
                    cache[i][j] = (v + va, j :: pa)

    return maximum(map(\(v, p). v, cache[w]))
```

It's nearly 17:00, so I appologise for this answer being incomplete.