MATH 2112/CSCI 2112, Discrete Structures I
Winter 2007

Toby Kenney

Homework Sheet 8
Model Solutions

Compulsory questions

1 (a) Consider the following algorithm for finding the nth fibonacci number:

Input: natural number n
Output: nth Fibonacci number
  if n=0 **then**
    return 0
  end if
  if n=1 **then**
    return 1
  end if
  Find the $n-1$th Fibonacci number {using this algorithm}
  Find the $n-2$th Fibonacci number {using this algorithm}
  Add them together and
  return the result.

Find a recurrence relation for the number of additions required to calculate the nth fibonacci number using this algorithm and solve it.

Let the number of additions required to calculate F_n be a_n, then $a_n = a_{n-1} + a_{n-2} + 1$, while $a_0 = 0, a_1 = 0$. This gives the following first few values:

n	a_n
2	1
3	2
4	4
5	7
6	12

This leads us to guess that $a_n = F_{n+1} - 1$. We now prove this by induction. We have already checked the base cases. Suppose it holds for $n-2$ and $n-1$; we want to prove it holds for n. By our inductive hypothesis, $a_{n-2} = F_{n-1} - 1$, $a_{n-1} = F_n - 1$, so by our recurrence relation, $a_n = a_{n-1} + a_{n-2} + 1 = F_{n-1} - 1 + F_n - 1 + 1 = F_{n-1} + F_n - 1 = F_{n+1} - 1$, as required, so by induction, the formula works for all n.

(b) Now consider the following algorithm to find both F_n and F_{n+1}:

Input: natural number n
Output: nth Fibonacci number
  if n=0 **then**
    return 0 and 1
  end if
  Find F_{n-1} and F_n, the $n-1$th and nth fibonacci numbers {using this algorithm}
  return F_n and $F_{n-1} + F_n$.

How many additions does this algorithm need to calculate F_n and F_{n+1}?

Let a_n be the number of additions that this algorithm needs to calculate F_n and F_{n+1}. a_n satisfies the recurrence relation $a_{n+1} = a_n + 1$, and the base case is $a_0 = 0$, so we get $a_n = n$, i.e the algorithm needs just n additions to find F_n and F_{n+1}.

2 *Which of the following functions are $\Theta(n^a)$ for some $0 < a < \infty$. For functions which are $\Theta(n^a)$ for some a, give the value of a. For function which are not, are they $O(n^a)$ for all a? are they $\Omega(n^a)$ for all a? Justify your answers. You may use any of the results about O, Ω and Θ proved in the lectures.*

(a) $f(n) = n^7 - 3n^{3.6} + 4$

f is $\Theta(n^7)$. It was proved in lectures that for $a_1 \neq 0$, and $b_1 > b_2 > \ldots > b_k > 0$, $a_1 n^{b_1} + a_2 n^{b_2} + \ldots + a_k n^{b_k}$ is $\Theta(n_1^b)$: this is a special case of that result.

(b) $f(n) = e^{2n}$

f is $\Omega(n^a)$ for all a, since $e^{2n} = (e^2)^n$, and as was shown in lectures, x^n is $\Omega(n^a)$ for any a.

(c) $f(n) = 6$

This is $O(n^a)$ for any $a > 0$, since for any K, if $n > (6K)^{\frac{1}{a}}$, $n^a > 6K$, so $6 \leqslant \frac{n^a}{K}$.

(d) $f(n) = (n+3)\log(n)$

This is not $\Theta(n^a)$ for any a: $n+3$ is $O(n)$, while $\log(n)$ is $O(n^a)$ for any $a > 0$, so f is $O(n^a)$ for any $a > 1$. On the other hand, f is not $O(n)$, since if $n > e^K$, then $f(n) > Kn$, for any K, so we can't choose a K and

an N such that $(\forall n \geqslant N)(f(n) \leqslant Kn)$. Therefore, f cannot be $\Theta(n^a)$ for any a, since if $a \leqslant 1$, f is not $O(n^a)$, while if $a > 1$, then f is $O\left(n^{\frac{a+1}{2}}\right)$, so if f were $\Omega(n^a)$, then $n^{\frac{a+1}{2}}$ would also be $\Omega(n^a)$, which it isn't.

(e) $f(n) = n^3 + n(\log(n))^2$

f is $\Theta(n^3)$. Since $\log(n)$ is $O(n^{0.5})$, $n(\log(n))^2$ is $O(n^2)$, so f is also $O(n^3)$. On the other hand, for any K, we can choose N so that $(\forall n \geqslant N)(|n^3| > Kn(\log(n))^2)$. Therefore, $|f| > (K-1)n(\log(n))^2$, so, we get that $n(\log(n))^2$ is $O(f)$. Thus, n^3 is $O(f)$, so f is $\Omega(n^3)$, and therefore, f is $\Theta(n^3)$.

(f) $f(n) = \sqrt{n} - \log(n^2 + 5)$

f is $\Theta(n^{\frac{1}{2}})$. $log(n^2 + 5)$ is $O(\log(2n^2))$, and $\log(2n^2) = 2\log n + \log 2$, which is $O(logn)$, and therefore, $O(n^a)$ for any $a > 0$. Thus, $\log(n^2 + 5)$ is $O(n^{\frac{1}{2}})$, and as in (e), it is $O(f)$, so as in (e), f us $\Theta(n^{\frac{1}{2}})$.

3 Consider the following algorithm for finding an element in a sorted list $a[1], a[2], \ldots, a[n]$ of length n.

Input: x – item for which to search
Output: index at which x occurs in the list (or **false** if it doesn't occur)

```
if n = 0 then
  return false
else
  Compare x to a[n/2] {rounding n/2 up to the nearest integer}
  if x = a[n/2] then
    return n/2
  else if x < a[n/2] then
    use this algorithm to find x in the list a[1], a[2], ..., a[n/2 − 1], and
    return the result.
  else if x > a[n/2] then
    use this algorithm to find x in the list a[n/2+1], a[n/2+2], ..., a[n], and
    return the result plus n/2.
  end if
end if
```

(a) How many comparisons does this algorithm take to find x:

(i) in the best case?

In the best case, x is the $n/2$th number in the list, so the algorithm makes only 1 comparison.

(ii) in the worst case?

In the worst case, the algorithm never finds x, and at each stage searches the longer sublist for x. If the list has length n, the longer sublist (or either sublist if n is odd) has at most $\frac{n}{2}$ elements, and the algorithm makes one comparison before searching the sublist. Therefore, the algorithm makes $k+1$ comparisons, where k is the smallest natural number such that $n < 2^k$. This is $\Theta(\log n)$ comparisons.

(b) If the list is not sorted, the best search algorithm takes $O(n)$ comparisons to find x on average. How many searches must a program perform in order for it to be faster to sort the list with a merge-sort than to simply use an unsorted list? (Give the order of magnitude, i.e. something like "$\Omega(n^3(\log(n))^2)$ searches".) Justify your answer.

Recall that sorting the list takes $\Theta(n \log n)$ operations, so sorting the list then performing $f(n)$ searches takes a total of $\Theta(n \log n + f(n) \log n)$ operations. On the other hand, performing $f(n)$ searches on an unsorted list takes $O(nf(n))$ operations. We want to find what $f(n)$ makes these two functions have the same order of magnitude. Clearly, we need $nf(n)$ to be $\Omega(n \log n)$, since $n \log n + f(n) \log n$ is $\Omega(n \log n)$. To get $nf(n)$ to be $\Omega(n \log n)$, we need $f(n)$ to be $\Omega(\log n)$. On the other hand, if $f(n)$ is $\Omega(\log n)$, then $n \log n$ is $O(nf(n))$, and $f(n) \log n$ is $O(nf(n))$, so we get that $n \log n + f(n) \log n$ is $O(nf(n))$, so we need to perform $\Omega(\log n)$ searches in order for it to be faster to sort the list before searching.

4 Recall the insertion sort: (This version is slightly different from the version in the textbook.)

---

for $i = 1$ to n **do**
  for $j = i - 1$ to 0 **do**
    if $j = 0$ **then**
      move $a[i]$ to the front of the list. {This requires i swaps.}
    else
      compare $a[i]$ and $a[j]$.
      if $a[i] \geqslant a[j]$ **then**
        insert $a[i]$ just after $a[j]$ {This requires $i - j - 1$ swaps.}
        go to next i.
      end if
    end if
  end for
end for

Suppose the list $a[1], \ldots, a[n]$ is initially sorted, then 100 of its values are changed at random. How many comparisons and swaps will be needed for

the insertion sort to sort the changed list? Explain your answer. [You only need to give the order of magnitude, e.g. $\Theta(n \log n)$*.]*

The number of comparisons needed is $\Theta(n)$, and the number of swaps needed is also $\Theta(n)$. Note that if $a[i]$ and $a[j]$ $(j < i)$ are unchanged, then $a[j] < a[i]$, since the list was initially sorted. Therefore, we only compare each unchanged element to at most 1 earlier unchanged element. Therefore, the number of comparisons we make is at most $101(n-100) + 100(n-1)$, since we compare each of the $n-100$ unchanged elements to at most 101 earlier elements, and we compare each of the 100 changed elements to at most $n-1$ elements, since there are only n elements in the list. (In fact, we can get a smaller upper bound, of about $\frac{1}{2}$ this upper bound, but since we're only worried about the order of magnitude, that's not important.) We therefore perform $O(n)$ comparisons. Also, We compare every element except the first one to the preceding element, so we make at least n comparisons, so we make $\Theta(n)$ comparisons.

For swaps, note that we never make any swaps that swap elements that are in the correct relative order, so every swap involves a changed element. Therefore, we make at most $100n$ swaps. If the list is already sorted, we make no swaps, so the number of swaps we make is $O(n)$.

Bonus question

5 *Prove that any algorithm for sorting a list using only comparisons and swaps, must use* $\Omega(n \log n)$ *comparisons in the worst case. [Hint: There are* $n!$ *possible orders the list can start in. The comparisons made must distinguish between all of these possibilities. You may use the fact that* $\log(n!)$ *is* $\Theta(n \log n)$*.]*

Our algorithm must sort the list in less than $\Theta(n \log n)$ comparisons, from any initial permutation (rearrangement) of the list. Each comparison made has 3 possible outcomes, (less, equal, or greater). Therefore, every time our algorithm makes a comparison, it partitions the set of possible starting positions into 3 distinct sets – the starting positions where this comparison gives a less than result, the starting positions where it gives a greater than result, and the starting positions where this comparison gives an equality result. If the number of possible starting positions before this comparison is N, then one of these partitions must have at least $\frac{N}{3}$ members. In order for our sort algorithm to work, it must partition the set of possible starting permutations into sets of size 1, since any two different partitions must have a different collection of swaps made to sort them, and therefore, must give different results for one of the comparisons. If the number of comparisons made is at most k, then the set of possible starting permutations is partitioned into at most 3^k sets. In order for

these sets to all have at most one element, we must have that $3^k \geqslant n!$. Therefore, $k \geqslant \log(n!)$. But $\log(n!)$, is $\Theta(n \log n)$, so k must be $\Omega(n \log n)$.