ALGORITHMS AND COMPLEXITY

1. a) For each of the following statements, state whether it is true or false and provide a supporting proof.

i) $2^n = O(3^n)$ [3]

Answer:
True. $2^n \leq 3^n$ for all n.
Almost all students solved this correctly, although many made it much more difficult than it needed to be, taking logs and doing proofs by induction, etc.

ii) $n! = \Omega(2^n)$ [3]

Answer:
True. $n! = n(n-1)(n-2)\cdots(3)(2)(1)$, there are n terms and each (except for the last one) is ≥ 2, so $n! \geq 2^{n-1}$.
Almost all students got the correct answer, but not all justified it. Some students gave a table of the first few values, which wasn't sufficient.

iii) If $f(n) > 0$, $g(n) > 0$ and $f(n) = O(g(n))$ then $g(n) = \Omega(f(n))$. [4]

Answer:
True. $f(n) = O(g(n))$ means that for some constants $c > 0$, n_0 we have that whenever $n \geq n_0$, $f(n) < cg(n)$. Therefore, $g(n) > (1/c)f(n)$, which is the definition of $g(n) = \Omega(f(n))$.
Most students got the answer correct, although many thought that the definition was $f(n) < g(n)$ and forgot the role of the constants.

b) Give a tight bound for each of the following recurrence relations, or explain why it's not possible to do so.

Carefully justify your answers.

i) $T(n) = 16T(n/4) + O(n)$ [3]

Answer:
$a = 16, b = 4, d = 1$. So $\log_b a = \log_4 16 = 2 > 1 = d$ and the complexity is $O(n^{\log_b a}) = O(n^2)$.
Almost all students got this correct.

ii) $T(n) = 9T(n/3) + n(n+1)$ [3]

Answer:
$a = 9, b = 3, d = 2$. So $\log_b a = \log_3 9 = 2 = d$ and the complexity is $O(n^d \log n) = O(n^2 \log n)$.
Most students got this correct. A few didn't notice that $n(n+1) = O(n^2)$ and so thought that the Master theorem didn't apply.

iii) $T(n) = 2T(n-1) + 2^{-n}$ [4]

Answer:
The Master Theorem doesn't apply, however it can be expanded out to get $T(n) = 2T(n-1) + 2^{-n} = 2(2T(n-2) + 2^{-(n-1)}) + 2^{-n} = 2^2 T(n-2) + 2^{-n} + 2^{-n+2}$. Continue to get $T(n-k) = 2^k T(n-k) + 2^{-n}(1 + 4 + 4^2 + \cdots + 4^{k-1})$ so $T(n) = 2^n T(0) + 2^{-n}(1 + 4 + 4^2 + \cdots + 4^{n-1})$. The sum is $O(4^n)$ so $T(n) = 2^n T(0) + 2^{-n} O(2^{2n}) = 2^n (T(0) + O(1)) = 2^n O(1) = O(2^n)$. Alternatively, note that $T(n) \leq 2T(n-1) + 1$ so (by

expanding it out similarly) $T(n) \leq 2^n T(0) + 1 + 2 + 4 + \cdots + 2^{n-1} = 2^n T(0) + 2^n - 1 = O(2^n)$.
Most students noticed that the Master Theorem doesn't apply. A good number got the correct answer $O(2^n)$ *although very few managed to get all the expansion and algebra correct.*

Master Theorem. If $T(n)$ satisfies

$$T(n) = a\,T(\,n/b\,) \;+\; O(n^d)$$

for some $a > 0$, $b > 1$ and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

2. The programming language Earthworm has only one function `split(s, i)` for splitting a string s of length n at position i. It returns two substrings `left` and `right`, so that `left` consists of the first i characters of the string, and `right` consists of the last $n-i$ characters. For example, `split('abcd', 1)` returns `'a', 'bcd'`. The `split` function takes n units of time to run on a string of length n.

You are given the task of using the `split` function to create an efficient implementation of the function `multisplit(s, b)`. This takes a string s of length n and a sorted array of integers b of length k, and returns a sequence of strings corresponding to splitting s at b_1, b_2, etc. For example, `multisplit('abcd', [1, 3])` would return `'a', 'bc', 'd'`.

a) The "right-first" algorithm for implementing `multisplit` is to first split at b_k, then to split the left part at position b_{k-1}, then the left part of that at position b_{k-2}, and so on to the last split at b_1. How many units of time will this algorithm take? Express your answer in terms of n and the values in the array b. [5]

Answer:

The first split will take n units. The second split is on a string of length b_k and so will take b_k units, and so on. The last split will be at position b_1 and this will take b_2 units of time. The total time will be $n + \sum_{i=2}^{k} b_i$.

A good number of students got the correct answer, although many summed from b_1. Quite a few assumed each step would have time cost n so gave the answer nk. Quite a few also tried something equivalent to left-first, assuming that the number of remaining characters after the first step was $n - b_k$ rather than b_k.

b) In the worst case where the string has to be split into n parts of length 1, i.e. when $b_i = i$ for $i = 1, \ldots, n-1$, how many units of time will the right-first algorithm take? [5]

Answer:

Applying the formula above it will take $n + \sum_{i=2}^{n-1} i = 2 + 3 + \ldots + n = n(n+1)/2 - 1$.

Most students got the correct answer or close to it (if they included 1 in the sum instead of starting from 2).

c) The "binary-split" algorithm repeatedly divides the left and right parts into two equal subparts. So for `'abcd'` it would first split into `'ab'` and `'cd'`, then split `'ab'` into `'a'`, `'b'`, and `'cd'` into `'c'`, `'d'`. How many units of time will this algorithm take for the worst-case considered in part (b) when n is a power of 2? [5]

Answer:

At stage i of the algorithm there will be 2^i strings remaining of length $n/2^i$. Each split operation therefore takes $n/2^i$ steps, and there are 2^i such operations for stage i. Therefore, each stage takes n steps, and there are $\log_2 n$ stages so the total time is $n \log_2 n$.

Most students got this correct, although quite a few assumed that the cost of each step was $O(1)$ and so gave the answer as just $O(\log n)$. Many students used the Master Theorem to derive the answer rather than directly analysing the tree.

d) For the general case, use dynamic programming to find how long the optimal sequence of splits would take. Write your answer using pseudocode. You do not need to compute the complexity of the dynamic program.

Hint: you may find it useful to consider the subproblem of finding the optimal number of units of time $C(i : j)$ taken to split the substring of characters in positions i to $j-1$. [10]

Answer:
$C(i:j)$ will either be 0 if there are no splits between i and j, or it will cost $j-i$ (the next split) plus the minimum cost of the remaining splits over all the possible next splits:

$$C(i:j) = \begin{cases} 0 & \text{there is no } i < b_k < j \\ j - i + \min_{i<b_k<j} C(i:b_k) + C(b_k:j) & \text{otherwise} \end{cases}.$$

The following Python code implements this:

```
def C(i, j, b):
    b = [bk for bk in b if i<bk<j]
    if len(b)==0:
        return 0
    return j-i+min([C(i, bk, b)+C(bk, j, b) for bk in b])
```

Note that a more efficient implementation of this dynamic program would be to initially include in b a split at positions 0 and n, sort the split positions in ascending order, and then consider $D(i:j)$ the optimal time taken to split characters b_i to $b_j - 1$, which will be only the split positions b_{i+1} to b_{j-1}. This avoids searching through the array b each time. However, the question does not ask for the most efficient algorithm, so this isn't necessary.
Few students were able to do this correctly. Some didn't understand the question and put code for split/multisplit. Many didn't use dynamic programming but just did an implementation that they thought was intuitively correct. Some didn't consider the general case but only the worst case. A surprisingly large number wrote down the code for the rod cutting example from the lectures which doesn't work at all. In general, it was also very hard to understand the pseudocode that people wrote.

e) For the worst case considered in parts (b) and (c), use the algorithm in (d) to compute the optimal number of units of time to split strings of length 2, 4 and 8. How does the binary-split algorithm compare to these optimal times? [5]

Answer:
We simplify by noting that in the worst case considered above, $C(i:j)$ only depends on $j-i$ and so we write it as C_{j-i}. Now, $C_0 = C_1 = 0$ and the formula above translates to $C_n = n + \min_{k=1}^{n-1} C_k + C_{n-k}$. So:
$C_2 = 2 + C_1 + C_1 = 2$.
$C_3 = 3 + \min\{C_1 + C_2, C_2 + C_1\} = 3 + C_1 + C_2 = 5$.
$C_4 = 4 + \min\{C_1 + C_3, C_2 + C_2\} = 4 + \min\{5, 4\} = 8$.
$C_5 = 5 + \min\{C_1 + C_4, C_2 + C_3\} = 5 + \min\{8, 7\} = 12$.
$C_6 = 6 + \min\{C_1 + C_5, C_2 + C_4, C_3 + C_3\} = 6 + \min\{12, 10, 10\} = 16$.
$C_7 = 7 + \min\{C_1 + C_6, C_2 + C_5, C_3 + C_4\} = 7 + \min\{16, 14, 13\} = 20$.
$C_8 = 8 + \min\{C_1 + C_7, C_2 + C_6, C_3 + C_5, C_4 + C_4\} = 8 + \min\{20, 18, 17, 16\} = 24$.

So $C_n = n \log_2 n$ when n is a power of 2, and the binary-split algorithm is optimal in this case.
Almost nobody gave a complete answer to this question with all the reasoning above, although most guessed that the answers would be 2, 8, 24 from the binary split algorithm.