ALGORITHMS AND COMPLEXITY

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

i) $2n^2 + 3n^3 + 4n^4 = O(n^4)$. [3]

Answer:

True. Easy to show that for all n, we have $n^2 \leq n^4$ and $n^3 \leq n^4$. So $2n^2 + 3n^3 + 4n^4 \leq 2n^4 + 3n^4 + 4n^4 = 9n^4$. So let $n_0 = 1$ and $c = 9$ in definition of O.

Most students solved this correctly, although quite a few just asserted that there was a c without giving a value. Quite a few students used that the complexity of a polynomial is the highest term, which was fine since it had been stated in lectures.

ii) $n^2 + 2^{-n} = O(n^2)$. [3]

Answer:

True. For $n \geq 1$ we have $2^{-n} \leq 1/2 \leq n^2$ and therefore $n^2 + 2^{-n} \leq 2n^2$ so set $n_0 = 1$ and $c = 2$.

Most students solved this correctly, although quite a few just asserted that there was a c without giving a value.

b) Describe an algorithm (using pseudocode or a precise description in words) for each of the following tasks, and give a tight upper bound for its running time in O notation. You may assume and state running times for standard algorithms without proving them.

i) Calculate the mean of an array of n values. [3]

Answer:

Set $S = 0$. Iterate through each item x of the array X, and compute $S = S + x$. Return S/n. Within each iteration, fixed number of operations so $O(1)$. First and final steps are fixed number of operations so $O(1)$. n iterations, so total cost is $O(n)$.

Almost all students got this right. A very few used divide and conquer, which is fine, although these students tended to get the analysis wrong giving $T(n) = 2T(n/2) + O(n)$ instead of the correct $T(n) = 2T(n/2) + O(1)$.

ii) Count the number of unique items in an array of n values. So, for example, the array $\{1,5,3,1,3,1,3\}$ has $n = 7$ but only 3 unique items. [4]

Answer:

Three equally good answers:

For each i, iterate over all $j < i$ and look for $X[i] == X[j]$. If you don't find one, increment count of unique. Only increments counter the first time a value is seen, and so counts all unique items once. Complexity is $O(n^2)$ because iterating over i and j up to maximum values of n.

Sort the list $O(n \log n)$ using merge sort or other standard algorithm. Set the count of unique items $U = 0$. Iterate from $i = 0$ to $n-1$. For each i, if $i = 0$ or $X[i-1] \neq X[i]$, set $U = U + 1$. Return U. Each *iteration is $O(1)$ and there are n iterations so the second stage of the algorithm is $O(n)$. $O(n)+O(n\log n)=O(n\log n)$ so the algorithm is $O(n\log n)$. The algorithm works because when the array is sorted, duplicate copies of a value will be adjacent in the sorted array.*

Initialise a hash table T with values being booleans (for example). Set $U=0$. Iterate through each item x in the array X. If x is not in T, set $U=U+1$ and set $T[x]=\mathrm{True}$. Return U. Each hash table operation is $O(1)$ and there are n iterations so this algorithm is $O(n)$.
Most students got this right with one of the solutions above. Most used the last solution with a less efficient data structure (list or vector) making the algorithm $O(n^2)$. Many didn't correctly analyse the data structure operations (e.g. not accounting for the cost of checking if an element is in the data structure, or the cost of adding an element to the data structure).

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

i) $T(n)=8T(n/4)+n\sqrt{n}$. [3]
Answer:
$a=8$, $b=4$, $d=1.5$. $\log_b a=\log_4 8=1.5=d$. So, $T(n)=O(n^d\log n)=O(n^{1.5}\log n)$.
Almost all students got this.

ii) $T(n)=aT(n/3)+O(n^3)$ assuming $a<27$. [4]
Answer:
To apply the master theorem, we need to know which is larger, d or $\log_b a$. Since $a<27$ we have $\log_3 a<\log_3 27=3=d$. Therefore, $T(n)=O(n^d)=O(n^3)$.
Most students got this.

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. Moria Mining Corporation (MMC) holds the rights to n mine shafts. They estimate that mine shaft i ($0 \le i < n$) has a total amount of gold g_i. Their competitor, Balrog Incorporated (BI), is going to mount a hostile takeover of these mine shafts in T days. MMC wants to maximise the amount of gold they extract before this happens. Each day, only a single shaft can be mined and the amount of gold they can extract from mine i if they spend t days mining it is

$$m_i(t) = g_i \cdot \left(1 - \frac{1}{t+1}\right).$$

MMC have asked you to devise a dynamic programming algorithm to maximise the total amount of gold they can extract in the time remaining.

We will define $G(n,T)$ as the maximum amount of gold that can be mined from mine shafts 0 to $n-1$ in T days.

a) Explain what each of the following special cases for $G(n,T)$ means in a single non-mathematical sentence, and find the solution in each case:

i) $G(n,0)$. [2]

Answer:
If $T = 0$ there is no time remaining, so no gold can be mined, and $G(n,0) = 0$.
Almost all got this.

ii) $G(1,T)$. [2]

Answer:
If $n = 1$ there is only one mine shaft remaining, so all the remaining time T should be used on this shaft. This gives $G(1,T) = m_0(T)$.
Most got this, although quite a few thought that either it was asking for the best solution for 2 mines remaining, or thought that it was the best solution for one mine but you could pick which mine.

iii) $G(n,1)$. [2]

Answer:
If $T = 1$ there is only one day remaining, so it should be spent on the mine with the largest value of g_i to give $G(n,1) = \max_i m_i(1) = (\max_i g_i)(1 - 1/2) = (\max_i g_i)/2$.
Most got this, although quite a few didn't write max *in their solutions.*

b) Write an equation for $G(n,T)$, by considering the options for the number of days t that MMC should spend mining shaft $n-1$, and expressing the optimal solution in terms of $m_{n-1}(0)$, $m_{n-1}(1)$, $m_{n-1}(2)$, …, and $G(n-1,T)$, $G(n-1,T-1)$, $G(n-1,T-2)$, …. [6]

Answer:
The options available are to spend 0, 1, 2, …, T days on shaft $n-1$. If they spend t days, then they will mine $m_{n-1}(t)$ gold from shaft $n-1$. In the remaining $T-t$ days they can mine $G(n-1,T-t)$ gold. So, if they spend t days on shaft $n-i$ they will make $m_{n-1}(t) + G(n-1,T-t)$ gold. To find the maximum amount of gold they can mine, we maximise over all the possible choices of t to get

$$G(n,T) = \max_{0 \le t \le T} m_{n-1}(t) + G(n-1,T-t).$$

The majority of students got this, or something close to it. A number included some unnecessary extra terms, or an unnecessary max over i. Quite a few got the idea of the expression inside the max, but got confused about how to write an equation with a max.

c) Write pseudocode for an efficient algorithm to compute $G(n,T)$, the maximum amount of gold they can mine. Your algorithm does not need to return the amount of time they should spend mining each shaft, and your pseudocode should not be substantially longer than 20 lines. [10]

Answer:
The following Python code solves this problem.

```
solutions = {}
def G(g, n, T):
    if (n, T) in solutions:
        return solutions[n, T]
    if T==0:
        return 0
    if n==1:
        return g[0]*f(T)
    best_G = G(g, n-1, T)
    for t in range(0, T+1):
        cur_G = g[n-1]*f(t)+G(g, n-1, T-t)
        if cur_G>best_G:
            best_G = cur_G
    solutions[n, T] = best_G
    return best_G
```

Quite a few students managed a perfect answer here, but the majority made some errors. The most common errors were: forgetting the base cases; forgetting to use memoisation for efficiency; inserting an unnecessary loop over the mine shafts; some minor error in logic; writing something too confusing to be understood. A very small number wrote a bottom-up solution rather than a recursive solution, and this was fine.

This year, greedy algorithms were not covered, however a very small number of students attempted a greedy algorithm instead of a dynamic programming algorithm. For interest, the following is an efficient implementation of a greedy algorithm solution using a Fibonacci heap priority queue. It has running time $O(T \log n + n)$ which is much better than the dynamic programming solution.

```
import heapq
def G(g, n, T):
    if T==0 or n==0:
        return 0
    timesofar = [0]*n # store time spent in each mine
    # priority queue consisting of pairs (priority, mineshaft)
    # priority is -gold_for_next_day because heapq is a min
    # queue
    pq = [(-g[i]/2, i) for i in range(n)]
    heapq.heapify(pq)
    totalgold = 0
    while T>0:
        # best mine for next day is one with min priority
        # (max gold for one extra day of mining)
        priority, nextmine = heapq.heappop(pq)
        nextgold = -priority
        totalgold += nextgold
```

```
            timesofar[nextmine] += 1
            t = timesofar[nextmine]
            # update the amount of gold for the next day and push
            # it back on to the queue
            nextgold = g[nextmine]/((t+1)*(t+2))
            heapq.heappush(pq, (-nextgold, nextmine))
            T -= 1
        return totalgold
```

d) Compute the time complexity of your algorithm in terms of n and T.

Note that complexity notation for two variables behaves as you would expect for one variable. In particular $O(f(x)) \times O(g(y)) = O(f(x)g(y))$. [4]

Answer:

Each call to G is at most $O(T)$ due to the loop $0 \le t \le T$. At the end of the computation, nT different values of $G(n,T)$ will have been computed, taking $O(T)$ each so the total time is $O(nT^2)$.

Few students got this right. Most gave $O(nT)$ as the answer reasoning that $G(n,T)$ should be called n times and has complexity $O(T)$.

e) Using your answer to parts (a) and (b), find the maximum amount of gold that could be mined for $n = 3$ mines each of which have a total amount of gold $g_i = 600$ in $T = 3$ days. [4]

Answer:

We drop the subscripts i because g_i and m_i are the same for each i. We start by computing $m(t)$ for $0 \le t \le 3$.

$m(0) = 0$

$m(1) = 600(1 - 1/2) = 300$

$m(2) = 600(1 - 1/3) = 400$

$m(3) = 600(1 - 1/4) = 450.$

Now we want to know $G(3,3)$ which we compute as

$$G(3,3) = \max \begin{cases} G(2,3) & t = 0 \\ G(2,2) + m(1) = G(2,2) + 300 & t = 1 \\ G(2,1) + m(2) = 300 + 400 = 700 & t = 2 \\ G(2,0) + m(3) = 0 + 450 = 450 & t = 3 \end{cases} \tag{2.1}$$

To compute this then, we need to compute $G(2,3)$ and $G(2,2)$. Let's start with $G(2,2)$.

$$G(2,2) = \max \begin{cases} G(1,2) = m(2) = 400 & t = 0 \\ G(1,1) + m(1) = m(1) + m(1) = 600 & t = 1 \\ G(1,0) + m(2) = 0 + 400 = 400 & t = 2 \end{cases} \tag{2.2}$$

So $G(2,2) = 600$. Now we compute $G(2,3)$:

$$G(2,3) = \max \begin{cases} G(1,3) = m(3) = 450 & t = 0 \\ G(1,2) + m(1) = m(2) + m(1) = 400 + 300 = 700 & t = 1 \\ G(1,1) + m(2) = m(1) + m(2) = 300 + 400 = 700 & t = 2 \\ G(1,0) + m(3) = 0 + 450 = 450 & t = 3 \end{cases} \tag{2.3}$$

So $G(2,3) = 700$. *We can now put this into our equation for* $G(3,3)$ *to get* $G(3,3) = \max\{700, 900, 700, 450\} = 900$.

Most students got the correct answer, and a good number managed to give the correct reasoning. Many just guessed (correctly) that an equal number of days for each would be the correct solution.