Assignment 1 3340

Justin Grant

1. Find the formula for the summation $\sum_1^n i(i+1)$. $Hint: i(i+1) = \frac{(i+1)^3 - i^3 - 1}{3}$

Proof:

$$USING\ THE\ HINT: \sum_1^n i(i+1) = (\frac{1}{3})\left(\sum_1^n (i+1)^3 - \sum_1^n i^3 - \sum_1^n 1\right)$$

$$= \left(\frac{1}{3}\right)\left(\sum_1^{n+1} i^3 - \sum_1^n i^3 - \sum_1^n 1\right)$$

$$= \left(\frac{1}{3}\right)((n+1)^3 - n)$$

$$= \frac{\mathrm{n}^3 + 3n^2 + 3n + 1 - n}{3}$$

$$= \frac{\mathrm{n}^3 + 3n^2 + 2n + 1}{3}$$

2. A complete binary tree is defined inductively as follows. A complete binary tree of height 0 consists of 1 node which is the root. A complete binary tree of height h+1 consists of two complete binary trees of height h whose roots are connected to a new root. Let T be a complete binary tree of height h. Prove that the number of leaves of the tree is 2^h and the size of the tree (number of nodes in T) is $2^{h+1} - 1$.

PART 1

The base case is when H=0

$1 = 2^0$, just like our induction hypothesis which says the leaves should be equal to 2^h

With the height k=1, we can examine the left and right subtrees, which are leaves. $2 = 2^1$.

When we examine the trees when height is k+1, the left and right subtrees have height k. We determined that trees of height k have leaves equal to 2^h. The number of leaves we have at k+1 must be equal to 2^k+2^k and equal to 2^{k+1}, which is the number of the current height of the tree. Since we are using variables and proved this for the value k, this is true for any value of k>1, and thus it is proved.

PART 2

The base case is when H=0

Our induction hypothesis states that the number of nodes in a tree T should be equal to $2^{h+1} - 1$ This is true when H = 0, as there is only 1 node. As well as true when H = 1, since there is 3 nodes. We can examine the tree when the height is equal to k+1. At level k+1 all the nodes are all leaves, so we know

there should be 2^{h+1} number of leaves. If we were to remove those leaves we would have a tree of size 2^h, as the height of the tree would be h.

The total number of nodes in T is $2^{(k+1)} + 2^{(k+1)} - 1 = 2(2^{k+1+1}) - 1$ The hypothesis holds for k+1 thus it is proved.

3.

Question R-1.7 (pp. 42) in the text book.

Similar Thetas are highlighted

The number orders from slowest growing to fastest growing.

1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
$O\left(\frac{1}{n}\right)$	$O(1)$	$O(\log\log n)$	$O(\sqrt{\log n})$	$O(\log^2 n)$	$O(\sqrt[100]{n})$	$O(\sqrt{n})$	$\lceil O(\sqrt{n}) \rceil$	$O(n)$	$O(n \log n)$
$\frac{1}{n}$	2^{100}	$\log\log n$	$\sqrt{\log n}$	$\log^2 n$	$n^{0.01}$	$3n^{0.5}$	$\lceil \sqrt{n} \rceil$	$5n$	$6n \log n$
10th	11th	12th	13th	14th	15th	16th	17th	18th	18th
$O(n \log_4 n)$	$O(\lfloor n \log^2 n \rfloor)$	$O(\sqrt{n^3})$	$O(n^2 \log n)$	$O(n^3)$	$O(2^{\log n})$	$O(2^n)$	$O(4^{\log n})$	$O(4^n)$	$O(4^n)$
$n \log_4 n$	$2n \log^2 n$	$4n^{\frac{3}{2}}$	$n^2 \log n$	n^3	$2^{\log n}$	2^n	$4^{\log n}$	2^{2^n}	4^n

4. Question C-1.13 (pp. 46) in the text book.

C-1.13 Describe a method for finding both the minimum and maximum of n number using fewer than 3n/2 comparisons.

We first make our minimum number negative infinity, and our maximum number positive infinity. We first look at the first two elements in the group. We compare them to each other to determine which the bigger number is. After finding the bigger number we that larger number to our maximum, and if it is larger, then we store that value as our new maximum. We compare the smaller number to our minimum number, and if it is smaller that is our new minimum number. After this we get the next two numbers, and repeat the process.

This takes 3n/2 comparisons because when we grab the numbers we make 3 comparisons. The comparison against each other, against the minimum and against the maximum. We must do this n/2 times because we go through the group/array 2 numbers at a time. In the end doing 3n/2 comparisons.

5. Question A-1.12 (pp. 49) in the text book.

A-1.12 Given an array, A, of n – 2 unique integers in the range from 1 to n, describe an O(n)-time method for finding the two integers in the range from 1 to n that are not in A. You may only use O(1) space) in addition to the space used by A.

We would go through the array once and sum all of the elements in the array, and we get S. Now we know that the sum S plus the missing integers a and b is equal to $\sum_1^n i$ or (n(n+1))/2, which we could say is T. So a+b = T-S. Next we go through the array and get the product of all the integers in the array, which is P, and then find the product of all the elements that should be in the array which is G. So we know that ab = G / P. Now we have two formulas including a and b, which we can then solve for the missing integers. The reason this is in O(n) time is that we make two passes through the array to find the sum and product of the elements in the array, which takes 2n accesses. Finding the sum and product of all the elements that should be in the array is easy because there is a formula for that as well.

6. Suppose that the running time of a recursive program is represented by the following recurrence relation:

$$\mathrm{T}(2) = 1$$

$$\mathrm{T(n)} \leq 2T\left(\frac{n}{2}\right) + nlog_2(n)$$

Determine the time complexity of the program using recurrence tree method and then prove your answer.

Proof

Base Case$T(2) = 1$

$$\mathrm{T(n)} \leq 2T\left(\frac{n}{2}\right) + nlog_2(n)$$

$$= 4\mathrm{T}\left(\frac{\mathrm{n}}{4}\right) + nlog_2(n) + nlog_2(n)$$

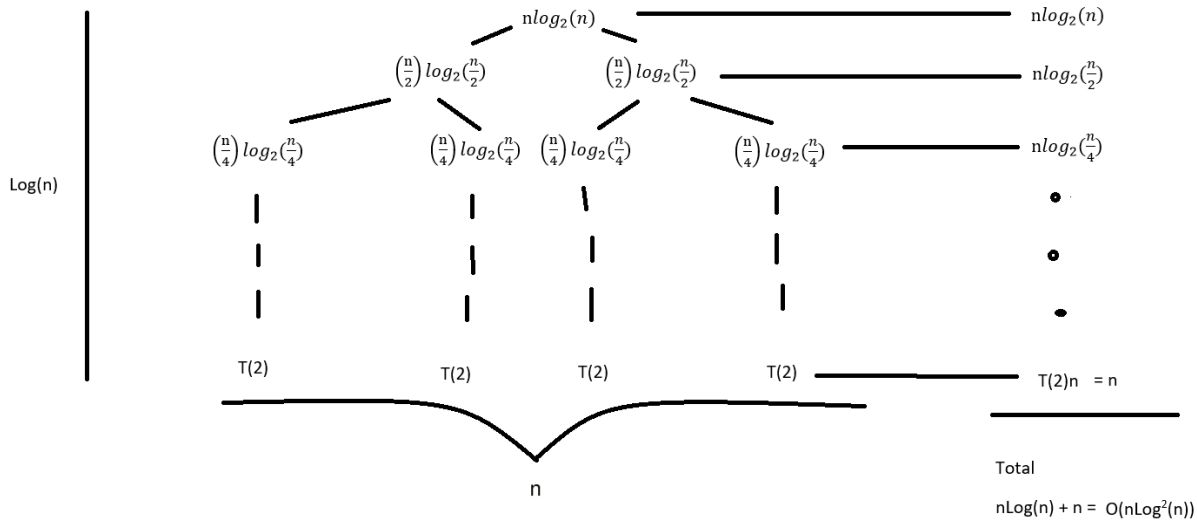
$$= 8\mathrm{T}\left(\frac{\mathrm{n}}{8}\right) + nlog_2(n) + nlog_2(n) + nlog_2(n)$$

$$= 2^{\mathrm{i}}T\left(\frac{n}{2^i}\right) + i\big(nlog_2(n)\big)\ \ i = logn - 1$$

$$\ldots = \mathrm{n}log_2(n)\log(n-1) + n \Rightarrow 2^iT\left(\frac{n}{2}\right) = n$$

$$k * nlog_2(n) = (nlogn)(logn - 1)$$

$$nlog^2n - nlogn + n = O(nlog^2n)$$



c) Use the Unix time facility (/usr/bin/time) to track the time needed to compute for each run and for each algorithm. Compare the results and state your conclusion in two or three sentences.

The second program runs slightly faster than the first program, even though it is calculating a much larger Fibonacci sum. This is because they operate in completely different time orders. The first program runs in O(n^2) time, and the seconds one runs in O(n) time.

e) Can you use your program in a) to compute F(50)? Briefly explain your answer. Explain why your program in b) computes F(300) precisely?

I cannot use my first program to compute F(50). The program has an order of O(n^2), which means that it grows very fast. The length of time that it takes each time we increase the input increases tremendously, as well as the amount of memory it needs as well. On the other hand program 2 runs F(300) fine because it runs in O(n) time, which is a constant time. For each new input there is only a constant increase in how long it takes. If it takes a second to run F(1), then it will take 300 seconds to run F(300), so it is guaranteed to finish.

The first one works be recursively calling itself. The function either returns itself(n-1) + itself(n-2), returns 1, or returns 0 depending on the parameter passed to it. The second one stores each value it calculates into an array so we do not have to re-calculate any values. Each call branches exponentially, and ends up running in O(n^2) time Thus it calculates each value one step at a time in one statement, and runs in O(n) time.