Homework 2 COM S 311

Alec Meyer

September 11, 2020

Question 1

a.

$2^{2n} \in O(2^n)$

Claim: $2^{2n} \notin O(2^n)$

$f(n) \leq c \times g(n)$

f(n) = 2^{2n}, g(n) = 2^n

plug in f(n) and g(n)
$2^{2n} \leq c \times 2^n$
$\frac{2^{2n}}{2^n} \leq c$
$2^n \leq c$

Here we have a constant 'c' being greater than or equal to a function of n. This is not possible as 2^n will surpass the value of c. Therefore, since there is no constant value c to satisfy this claim we know that this statement is false.

$\therefore 2^{2n} \notin O(2^n)$

b.

Claim:
$f_1(n) \in O(g_1(n)) \wedge f_2(n) \in O(g_2(n)) \Rightarrow f_1(n) \times f_2(n) \in O(g_1(n) \times g_2(n))$

We know that:
$\frac{f_1(n)}{g_1(n)} \leq c_1$ and $\frac{f_2(n)}{g_2(n)} \leq c_2$

$f_1(n) \times f_2(n) \in O(g_1(n) \times g_2(n))$ goes to:
$f_1(n) \times f_2(n) \leq c_1 g_1(n) \times c_2 g_2(n)$
$=\frac{f_1(n)}{g_1(n)} \times \frac{f_2(n)}{g_2(n)} \leq c_1 c_2$

Therefore, we know that that these values of c are constant which proves that this statement is true.
$\therefore f_1(n) \in O(g_1(n)) \wedge f_2(n) \in O(g_2(n)) \Rightarrow f_1(n) \times f_2(n) \in O(g_1(n) \times g_2(n))$

Question 2

a.

Inner:
$\sum_{k=1}^{j} c_1 = c_1 j$

Middle:
$\sum_{j=i}^{n} (c_2 + c_1 j)$
$\sum_{j=i}^{n} c_2 + \sum_{j=i}^{n} c_1 j$
$\sum_{j=i}^{n} c_1 j = c_1 (\frac{n(n+1)}{2} - \frac{i(i-1)}{2}$
$\sum_{j=i}^{n} c_2 = c_2 (n - i + 1)$

Outer:
$\sum_{i=1}^{n-1} [c_2(n - i + 1) + c_1(\frac{n(n+1)}{2} - \frac{i(i-1)}{2})]$
$= \sum_{i=1}^{n-1} [c_2 n - c_2 i + c_2 + \frac{c_1 n^2}{2} + \frac{c_1 n}{2} - \frac{c_1 i^2}{2} - \frac{c_1 i}{2}]$
$= c_2 n^2 - \frac{c_2 n(n+1)}{2} - 1 + c_2 + \frac{c_1 n^3}{2} + \frac{c_1 n^2}{2} - \frac{n^2(n+1)}{2} - 1 - \frac{n(n+1)}{2} - 1$
$\in O(n^3)$

b.

$\sum_{i=0}^{j}[\sum_{x=i}^{y} c + \sum_{a=j}^{b} c]$
The two inner loops are independant of from the outer loop so their time will be O(n)
The outer for loop: $\sum_{i=0} jc$ has a runtime of O(n) as well:
Therefore, the final runtime of this algorithm is $O(n^2)$

Question 3

a.

```
{
for(int i = 0; i < A.size; i++)
        if(binarySearch(T-A[i]))
                TRUE
FALSE
}
```

Outer loop time:
$\sum_{i=0}^{n} BinarySearch$

We know that Binary Search has a runtime of O(logn) and runs n times in this algorithm, therefore the runtime of this algorithm is O(nlogn)

b.

```
{
sum = 0;
for(int i = 0; i < A.size; i++)
        sum = 0;
        for(int j = i; j < A.size; j++)
                sum += A[j];
                newArray[i][j] += sum;
return newArray;
}
```

Inner loop:

$\sum_{j=i}^{A.size} c_1$

Outer loop:

$\sum_{i=0}^{A.size} c_2$

These two for loops run at O(n), therefore the total runtime will be $O(n^2)$.

c.

```
{
left = 0;
right = A.length - 1;

//Checking edge cases
if(A[0] == 1)
        return 0;
else if(A[1] == 1)
        return 1;

while(left <= right)
        i = middle of A;

        if(A[i] - A[i - 1] == 1)
                return i;

        else if (i - 1 < 0 OR A[i - 1] == 0)
                left = i + 1;

        else if (i - 1 < 0 OR A[i - 1] == 1)
                right = i - 1;
                }
```

This is an augmentation of a Binary Search that does not affect the runtime. We know that Binary Search runs at $O(logn)$ time, therefore this algorithm has a runtime of $O(logn)$.

d.

Our current understanding of a Binary Heap (minHeap) is:

-a complete Binary Search Tree

-every nodes parent is less than or equal to that node

I am going to add another property to this tree:

-every element added to the heap will also be added to a hashmap with its corrisponding index

What I mean by this is when making a minHeap it takes O(n) time to build since it iterates through an array of size n adding elements to a heap. While it is iterating it will also take the index and index value and add them to a hashmap, which is a constant time proccess. Once there is a corrisponding hashmap to the heap we will be able to search the tree by index and index value at constant time. Using this property my implementation of a deleteValue(x) method will look something like:

```
deleteValue(x)
{
        i = hashmap.get(x) //will return the index of x in the heap
        heap.delete(i) //this will delete the index i at O(logn)
}
```

This implimentation will still run at $O(logn)$ time since heap's 'delete' method has not been altered.