CSE 3044: Design and Analysis of Algorithms Homework 2

Remy Mujynya
Kassidy Maberry Date: 2024/10/07

Statement

I certify that every answer in this assignment is the result of my own work; that I have neither copied off the Internet nor from any one else's work; and I have not shared my answers or attempts at answers with anyone else.

Problem 1

T/F There is no difference between $O(1)$ and $\Theta(1)$.

No, consider $f(n) = 0$. While $f(n) = O(1)$ it is not the case $f(n) = 1\Theta(1)$ this is because of how $Theta$ is defined. $c_1 > 0$ however, in this case we cannot multiply such that $1c_1 = 0$. To redefine this question There is no difference between $O(1)$ and $\Theta(1)$ when $f(n) = c$ for some $c > 0$.

Problem 2

Consider the following algorithm SORTX. Create columns Cost and # Times and add entries under them. Use those entries to compute the worst-case time complexity of SORTX. Assume a cost of 3c for the call to EXCHANGE, c for all other executable statements, and zero for comments.
Arrive at a closed form expression for T(n). Infer an asymptotic bound.
Separately, prove any sums of sequences you are using.

Line	Cost	Times
1	c	1
2	c	n
3	c	n - 1
4	c	X
5	c	Y
6	c	Y
7	3c	n - 1

Since this is worst case time complexity we will assume that The if statement will always be true upon execution thus lines 5 and 6 have the same number of times ran.

We'll start with the following expression for $T(n)$:
$T(n) = c + cn + c(n-1) + cX + 2cY + 3c(n-1)$.
$T(n) = cX + 2cY + cn + 4c(n-1) + c$.
First we will need to determine the runtime of X.
We know that at minimum line 4 will execute n times to check the for loop to execute.
X will then execute $Z = (n-1) + (n-2) + (n-3) + ... + 1$.
As the required amount of comparisons will decrease by 1 each time.
Thus, we can rewrite X as $X = \Sigma_{i=1}^{n-1}(n-i)$.
We can compute the sum of X as $X = \frac{n(n-1)}{2}$.
Now we just need to compute the sum of Y.
Each time, lines 5, 6 are executed we know it will execute one less time than X because the final execution of X is the terminating condition.
$Y = \Sigma_{i=1}^{n-1}((n-i)-1)$
Or $Y = \Sigma_{i=1}^{n-1}(n-i) - \Sigma_{i=1}^{n-1}1$.
Then $Y = \frac{n(n-1)}{2} - \Sigma_{i=1}^{n-1}1$.
Summing 1 n-1 times gives us n-1 with a final result of.
$Y = \frac{n(n-1)}{2} - n + 1$.
$Y = \frac{n^2-n}{2} - n + 1$.
$Y = \frac{n^2-3}{2} + 1$.
Finally, we can substitue back in giving us T(n).
$T(n) = c(\frac{n^2-n}{2}) + c(n^2-3n) + 2c + cn + 4c(n-1) + c$ $T(n) = 3c\frac{n^2+n}{2} - 5c$.

Problem 3

Part a

Using our pseudo-code notation, write an algorithm HEAPDELETE(A, i) that deletes the data in node number i and runs in O(lg n) time for an n-element heap. Of course, A remains a heap after deletion. It is crucial that you explain your algorithm clearly using comments. The grader will assume that the algorithm is incorrect if it is not clear. (By default, heaps are max-heaps, i.e., descending heaps.)

HEAPDELETE(A, i)

```
1: $s \leftarrow A.size$
2: $A[i] \leftarrow A[s]$ ▷ Assign A[i] as the last most element of the heap
3: $A.size \leftarrow A.size - 1$ ▷ Resize heap
4: while $i > 1$ do ▷ We know the root will always be the largest element
```

```
5:     Heapify(A,i) ▷ Restore the heap property if needed
6:     $i \leftarrow \lfloor \frac{i}{2} \rfloor$ ▷ Get next parent
```

Part b

We know heapify has a runtime of $O(lg(n))$ if the heap property needs to be restored initially at a given node i. If not heapify has a runtime of $O(1)$. If the heap property is still intact from the switch it'll call itself as many times as the current height - total height. Which will be lg(n). Otherwise, we'll need to restore the heap property. If it is on the first node it'll be $O(lg(n))$ and the reamining calls would have a runtime of $O(1)$. If the property needs to be restored within one of the parent nodes heapify will only recurse in each call once as before the deletion we had the heap property swapping the nodes once will restore it. Thus it'll run h times giving us $O(lg(n))$.

4

Full proof for heapsort

First let us look at the heap sort algorithm.
Heapsort(A)

```
1: $n \leftarrow A.heapsize$
2: BuildHeap(A)
3: for $i \leftarrow n$ downto 2 do
4:     Exchange(A, 1, i)
5:     A.heapsize *gets* A.heapsize - 1
6:     Heapify(A, 1)
```

We can construct a $T(n)$ of Heapsort with our given algorithm. We will represent BuildHeap as $B(n)$ and heapify as $H(n)$. Assuming each statement has a cost of c.

$T(n) = nc + (n-1)H(n)c + 2c(n-1) + b(n)c + c.$

Let's begin by deteriming what $H(n)$ is. Starting by looking at the heapify algorithm

Heapify(A ,i)

```
1: $l \leftarrow left(i)$
2: $r \leftarrow right(i)$
3: if $l \geq A.heapsize$ and $A[l] > A[i]$ then $largest \leftarrow l$
4:     if $r \geq A.heapsize$ and $A[r] > A[largest]$ then $largest \leftarrow l$
5:         if $largest <> i$ then swap(A, i, largest) heapify(A, largest)
```

Since the heap is arranged as a tree and we go down by height rather than by node we will instead use H(h).
$H(h) = H(h-1) + 8c.$
$H(0) = 5c$ So $H(h) = H(h-2) + 16c$ Repeating k times we get $H(h) = H(h-k) + 8kc.$
If $k = h$ then $H(h) = H(0) + 8kc.$
Finally $H(h) = 5c + 8hc.$
Since the height in any tree is equal to $log_2(n) = h.$
Finally, $H(h) = 5c + 8log_2(n)c = O(log_2(n)).$
Now let us look at buildheap.
Buildheap(A)

```
1: $n \leftarrow A.length$
2: $A.heapsize \leftarrow n$
3: for $i \leftarrow \lfloor \frac{n}{2} \rfloor$ downto 1 do
4:     Heapify(A,i)
```

Since the cost of heapify depends on the height of the tree we are heapifying. Since majority of our calls will have a low cost. We can arrange our cost as
$\frac{n+1}{4}(x) + \frac{n+1}{8}(2x) + \frac{n+1}{16}(3x) + ... + 1(c(log_2(n+1))).$
$(n+1)(\frac{1}{4} + \frac{1}{8} + \frac{1}{16}) + ... + \frac{1}{n+1}$
$\frac{n+1}{2}(\frac{1}{2} + \frac{2}{4} + \frac{3}{8}) + ... + \frac{log(\frac{n+1}{2})}{\frac{n+1}{2}}$
So $S = \Sigma_{i=1}^{h} \frac{i}{2^h}.$
And $2S = \frac{2i}{2^i} = \frac{i}{2^{h-1}}.$
$2S - S = \frac{i}{2^{i-1}} - \frac{i}{2^i}.$
$2S - S = \Sigma_{j=1}^{h} \frac{j}{2^{j-1}} - \frac{j-1}{2^{j-1}}.$
$S = 1.$
Thus $B(n) = O(n).$

Finally we can plug into our original equation and get the following.
$T(n) = nc + (n-1)log(n)c + 2c(n-1) + nc + c.$
For $O(n)$ we will take the largest term thus,
$T(n) = O(nlog(n)).$