CSE 3044: Design and Analysis of Algorithms

Kassidy Maberry

New Mexico Institute of Mining and Technology
Socorro, NM 87801, USA
E-mail: kassidy.maberry@student.nmt.edu

Date: 2024/10/07

Problem 1

Proof:
Think of when f(n) = 0...

Problem 2

Proof:

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 3a 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.)

Proof:

```
HEAPDELETE(A, i)
$s \leftarrow A.size$
$A[i] \leftarrow A[s]$ Assign A[i] as the last most element of the heap
$A.size \leftarrow A.size - 1$ Resize heap
**while** $i > 1$ **do** We know the root will always be the largest element
    Heapify(A,i) Restore the heap property if needed
    $i \leftarrow \lfloor \frac{i}{2} \rfloor$ Get next parent
```

■

Problem 3b Explain (no need to prove) why your algorithm attains the O(lg n) bound.

Proof:

We know heapify has a runtime of $O(lg(n))$ if the heap property needs to be restored initially at a given node i. Otherwise, we need to check the parent of i. Since the heap property is the previous lists at this point is assumed to have held true the runtime of heapify is then $O(1)$ and instead relies on the amount of nodes we need to run heapify on. We know at most the amount of nodes required will be the height which is $lg(n)$. Thus, $O(lg(n))$ is our runtime in that case. Thus, in both cases our runtime is $O(log(n))$.

■

Problem 4 Full proof for heapsort

Proof:

First let us look at the heap sort algorithm.

```
Heapsort(A)
  $n \leftarrow A.heapsize$
  BuildHeap(A)
  **for** $i \leftarrow n$ downto 2 **do**
```

```
    Exchange(A, 1, i)
    A.heapsize $gets$ A.heapsize - 1
    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)
$l \leftarrow left(i)$
$r \leftarrow right(i)$
if $l \geq A.heapsize$ and $A[l] > A[i]$ then $largest \leftarrow l$
    if $r \geq A.heapsize$ and $A[r] > A[largest]$ then $largest \leftarrow l$
        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, $h(h) = 5c + 8log_2(n)c.$
Now let us look at buildheap.
Buildheap(A)

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

Since the cost of heapify depends on the height of the tree we are heapifying. Let's define the cost of heapify as x.
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,
$T(n) = O(nlog(n))$.

■