UWI

Design and Analysis of Algorithms

Tutorial 6- Sample Solutions

Setting Up & Solving Recurrence Relations

1. Setup and solve the recurrence relation for the following algorithm.

```
Algorithm F(n)
if n = 0 return 1
else return F(n - 1) × n
```

The basic operation in the above function is multiplication. Now, we can rewrite the function $F(n)$ as:

$$F(n) = F(n-1) * n, \text{ for } n > 0 \tag{1}$$
$$F(0) = 1 \tag{2}$$

This can be rewritten with regards to the multiplication operation denoted by M(n). This is the recurrence relation.

$$M(n) = M(n-1) + 1, \text{ for } n > 0 \tag{3}$$
$$M(0) = 0 \tag{4}$$

Now, we proceed by using backward substitution:

$$M(n) = M(n-1) + 1 \tag{5}$$
$$\text{sub } n-1 \text{ into } M \quad = [M((n-1)-1)+1] + 1 = M(n-2) + 2 \tag{6}$$
$$\text{sub } n-2 \text{ into } M \quad = [M((n-2)-1)+1] + 2 = M(n-3) + 3 \tag{7}$$
$$\vdots \tag{8}$$
$$\text{we notice a pattern} \quad = M(n-i) + i \tag{9}$$
$$\vdots \tag{10}$$
$$\text{we generalize for n} \quad = M(n-n) + n \tag{11}$$
$$= M(0) + n \tag{12}$$
$$= 0 + n \tag{13}$$

Therefore, the time complexity for the above algorithm is $\Theta(n)$.

2. Setup and solve the recurrence relation for the following algorithm.

```
Algorithm S(n)
//Input: A positive integer n
//Output: The sum of the first n cubes
if n = 1 return 1
else return S(n - 1) + n * n * n
```

The basic operation in the algorithm is multiplication, whose number of executions we denote as M(n). From the analysis of the algorithm we can derive the following function of S.

$$S(n) = S(n-1) + n * n * n, \text{ for } n > 1 \tag{14}$$
$$S(1) = 1 \tag{15}$$

This can be rewritten in terms of the number of multiplication operations made by the algorithm as follows:

$$M(n) = M(n-1) + 2, \text{ for } n > 1 \tag{16}$$
$$M(1) = 0 \tag{17}$$
$$M(0) = 0, \text{ by extension} \tag{18}$$

This is because there are two multiplication operations performed if $n > 1$ and none are performed if it is equal to zero.

Now, we proceed by using backward substitution:

$$M(n) = M(n-1) + 2 \tag{19}$$
$$\text{sub } n-1 \text{ into } M \quad = [M((n-1)-1)+2] + 2 = M(n-2) + 2 \cdot 2 \tag{20}$$
$$\text{sub } n-2 \text{ into } M \quad = [M((n-2)-1)+2] + 2 \cdot 2 = M(n-3) + 2 \cdot 3 \tag{21}$$
$$\vdots \tag{22}$$
$$\text{we notice a pattern} \quad = M(n-i) + 2i \tag{23}$$
$$\vdots \tag{24}$$
$$\text{we generalize for n} \quad = M(n-n) + 2n \tag{25}$$
$$= M(0) + 2n \tag{26}$$
$$= 0 + 2n \tag{27}$$

Therefore, the time complexity of the above algorithm is $\Theta(n)$ if we ignore all constants.

3. Setup the recurrence relation for the following algorithm.

```
Algorithm BinarySearch(A[0...n - 1], K)
//Implements non-recursive binary search
//Input: An array A[0...n - 1] sorted in ascending order and a search key K
//Output: Index of the array's element that is equal to K or -1
l = 0;
r = n - 1
while l <= r do
    m = floor((l + r)/2)
    if K = A[m] return m
```

```
        else if K < A[m] r = m - 1
        else l = m + 1
    return -1
```

Since we are defining the number of comparisons made by the algorithm in the worst case, then we have to consider the case where the array A does not contain the key K. For the binary search algorithm, there is at least one comparison made at each iteration of the loop. This comparison determines whether K is equal to, less than or greater than a given element in the array. If K is not equal to the given element in the array, then a lower or upper bound is set for the new subset of elements in the array is created. This effectively halves the size of the array to consider at each iteration of the loop, given that the key is not found. This can be represented as $\frac{n}{2} + 1$. The additional 1 is for the key comparison each time the array is halved. This therefore means that we can derive the following recurrence relation:

$$C_w(n) = C_w(\frac{n}{2}) + 1, \text{ for } n > 1 \tag{28}$$

$$C_w(1) = 1 \tag{29}$$

Transform and Conquer Approach

It is important that we learn the theory behind the transform and conquer approach and its algorithms.

1. Explain the theory behind the transform and conquer approach
   (a) What is meant by the transform and conquer approach?
   The transform and conquer approach follows two stages:
      1. Transformation stage: problem's instance is modified in some way.
      2. Conquering stage: Solves transformed problem's instance.

2. Explain the theory behind heaps and heap sort.
   (a) What is a heap?
   A heap is an almost-complete binary tree with each node satisfying the heap property.
   (b) What is the heap property?
   If v and p(v) are a node and its parent, respectively, then the key of the item stored in p(v) is greater than or equal to the key of the item stored in v.
   (c) What are the two stages of Heap sort?
      1. Stage 1 (heap construction): Construct heap for given array by calling HeapBottomUp(H[1..n]). ($C_{wh}(n) \in O(n)$)
      2. Stage 2 (maximum deletions): Apply the root deletion operation n - 1 times to the remaining heap. ($C_w(n) \in O(nlogn)$)
   (d) What is the overall time complexity of heap sort?
   Its overall time complexity is: (recall the properties of asymptotic notations)

$$T_w(n) = T_{Stage1}(n) + T_{Stage2}(n) \tag{30}$$

$$= O(n) + O(nlogn) \tag{31}$$

$$\in O(nlogn) \tag{32}$$

   (e) Calculate the time complexity of the worst case scenario of the heapBottomUp operation, given that:

$$C_{wh}(n) = \sum_{i=0}^{h-1} 2(h-i)2^i, \text{ where } h = \lfloor log_2 n \rfloor \tag{33}$$

$$C_{wh}(n) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2\sum_{i=0}^{h-1}(h-i)2^i \tag{34}$$

$$= 2\left[\sum_{i=0}^{h-1} h2^i - \sum_{i=0}^{h-1} i2^i\right] = 2\left[h\sum_{i=0}^{h-1} 2^i - \sum_{i=0}^{h-1} i2^i\right] \tag{35}$$

see summation rules for 2^i and $i2^i$

$$= 2\left[(h(2^{h-1+1}-1)) - ((h-1-1)2^{h-1+1}+2)\right] \tag{36}$$

$$= 2\left[(h(2^h-1)) - ((h-2)2^h+2)\right] \tag{37}$$

$$= 2\left[h2^h - h - h2^h + 2^{h+1} - 2\right] \tag{38}$$

$$= 2\left[2^{h+1} - h - 2\right] \tag{39}$$

we rewrite as

$$= 2\left[2^{h+1} - (h+1) - 1\right] \tag{40}$$

we sub $h+1$ for $\lceil log_2(n+1) \rceil$

$$= 2\left[2^{log_2(n+1)} - (log_2(n+1)) - 1\right] \tag{41}$$

$$= 2\left[(n+1) - log_2(n+1) - 1\right] \tag{42}$$

$$= 2\left[n - log_2(n+1)\right] \tag{43}$$

Recall the proof from an earlier tutorial, where we proved that $\lfloor log_2(n) \rfloor + 1 = \lceil log_2(n+1) \rceil$. We know that $h = \lfloor log_2 n \rfloor$, therefore we can say $h + 1 = \lfloor log_2(n) \rfloor + 1$ and use the previously proven value. From the above simplification, we get $C_w(n) = 2(n - log_2(n+1))$, which means that $C_w \in O(n)$ since it makes at most $2n$ comparisons (recall the properties of asymptotic notations).