CIS 350/3501 Summer 2020
Data Structures and Algorithm Analysis
Homework # 1

Student Name: Demetrius Johnson
Due: 6/3/2021
Total points: 90

<u>Problem 1</u> (10 points)
Order the following functions by growth rate: $\sqrt{N}$, N, $2/N$, 2^N, N^3, $N \log N$, $N \log^2 N$, 37, $N^2 \log N$, $N^{1.5}$, $N \log\log N$, N^2, $2^{N/2}$, $N \log(N^2)$. Also, indicate which functions asymptotically grow at the same rate.

Growth Rate (smallest to largest):

1. 37 //constant time
2. $2/N$ //**Grows (decays) asymptotically towards x-axis as $N \rightarrow$ Infinity
3. $\sqrt{N}$
4. N
5. $N \log(\log N)$
6. $N \log N$
7. $N \log(N^2)$
8. $N \log^2 N$
9. $N^{1.5}$
10. N^2
11. $N^2 \log N$
12. N^3
13. $2^{N/2}$
14. 2^N

Problem 2 (16 points)
For each of the following six program fragment, give an analysis of the running time in Big-Oh (worst case) notation.

(1)
```
sum = 0;
for(i = 0; i < n; i++)
    sum++;
```

- **Worst Case: O(N)**
  - There is a loop that will execute a single instruction that will take constant time (sum++;), N times.

(2)
```
sum = 0;
for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
        sum++;
```

- **Worst Case: $O(N^2)$**
  - There are two loops, with one of the loops nested inside of the other loop. Both loops are controlled by and will execute N times, however, the inner loop executes N times for every time the outer loop executes. Thus, $N_{outer}*N_{inner} = N^2$ → inner loop executes N times every iteration of the outer loop, which will execute N times.

(3)
```
sum = 0;
for(i = 0; i < n; i++)
    for(j = 0; j < n * n; j++)
        sum++;
```

- **Worst Case: $O(N^3)$**
  - There are two loops, an i-loop and a j-loop. The j-loop is nested inside of the i-loop. The i-loop executes N times, and for every iteration of the i-loop, the j-loop executes N*N times. Thus, the total number of executions will be N*(N*N) times == N^3 times.

(4) sum = 0;

```
for(i = 0; i < n; i++)
    for(j = 0; j < i; j++)
        sum++;
```

- **Worst Case: $O(N^2)$**
    - There are two loops, an i-loop and a j-loop. The j-loop is nested inside of the i-loop. The i-loop executes N times, and for every iteration of the i-loop, the j-loop executes up to one less than the current iteration of the i-loop; for example, if n = 4, the i-loop executes 4 times, and the j-loop executes 0+1+2+3 = 6 times.
    - Thus, the total number of executions will be:
        - n-1 + n-2 + n-3 +...+ n-n times.
    - More formally, total executions will be:
        - The $\sum_{[i=1,\ i=n]}\{(n-i)\}$
        - Or similarly, $\sum_{[i=n-1,\ i=0]}\{i\}$
    - Now, after doing some simplification, the sum of total executions in the inner loop for a given n reduces to n((n-1)*0.5) = n((n-1)/2) = $(n^2-n)/2$ → which ultimately grows as N^2 as n approaches infinity. I discovered this simplification by noticing how each subsequent sum for a given n will produce a sum that is an additional 0.5 times bigger than the n value.
    -

```
(5) sum = 0;     //multiply the highest orders of each tree together
    for(i = 0; i < n; i++)              //n
        for(j = 0; j < i * i; j++)      //n²
            for(k = 0; k < j; k++)      //n²
                sum++;
```

- **Worst Case: $O(N^5)$**
  - Here we have 3 loops, with each subsequent loop nested inside of the other (i→j→k). The i-loop will iterate N times, the j loop will iterate the square of one less than the current iteration of the i-loop. The k-loop will iterate j times for every iteration of the j-loop.
  - For example, for n = 4, the i-loop will iterate N times, the j loop will iterate $0^2 + 1^2 + 2^2 + 3^2$ times, and the k-loop will iterate $0 + (0+\ldots+1^2) + (0+\ldots+2^2) + (0+\ldots+3^2)$.
  - Formally, the summation for the j-loop will simplify to:
    - The $\sum_{[i=1,\ i=n]}\{(n-i)^2\}$
    - or similarly, The $\sum_{[i=n-1,\ i=0]}\{i^2\}$
  - Using summation properties, we can simplify:
    - $\sum_{[i=n-1,\ i=0]}\{i^2\}$ =
    - $n(n-1)(2n-1)/6$
    - which simplifies to: $(2n^3-3n^2+n)/6$
  - Now, for every element in the above j-sum, you need to account for the k-loop which will provide a number of executions for every j-loop element of the above sum, namely:
    - The $\sum_{[i=1,\ i=n]}\{(n-i)\}$, for each j-sum element.
    - Which we know from the previous problem, simplifies to: $(n^2-n)/2$
  - Since each element evaluated in the j-sum must be summed up using $(n^2-n)/2$ in order to determine the total number of k-loop executions, we can use our j-sum and plug it into this equation:
    - The $\sum_{[i=1,\ i=n]}\{[((n-i)^2)^2 - (n-i)^2]/2\}$
    - Which simplifies to:
    - $\sum_{[i=1,\ i=n]}\{[(n-i)^4 - (n-i)^2]/2\}$
  - Now, as I was doing this I realized that I can also rewrite the above sum in another way:

    - $\sum_{[i=n-1,\ i=0]}\{i\}$ = $(n^2-n)/2$ is what needs to be resolved for every j element in the sum $\sum_{[i=n-1,\ i=0]}\{i^2\}$
    - Thus, $\sum_{[i=n-1,\ i=0]}\{[(i^2)^2-(i^2)]/2\}$
    - Which simplifies to $\sum_{[i=n-1,\ i=0]}\{[(i^4-i^2]/2\}$
    - $(½)[\sum_{[i=n-1,\ i=0]}\{i^4\} - \sum_{[i=n-1,\ i=0]}\{i^2\}$
  - This ultimately simplifies to:
    - $[(2n^3-3n^2+n)/6] * [(n^2-n)/2]$ =
    - **$(2n^5-5n^4+4n^3-n^2)/12$** → total executions of the k-loop == total executions of the function.
  - Thus, $(2n^5-5n^4+4n^3-n^2)/12$ has the highest power of n^5, so big O complexity of the function is **$O(N^5)$**.

```
(6) sum = 0;
    for(i = 1; i < n; i++)                    // n
        for(j = 1; j < i * i; j++)            // n²
            if (j % i == 0 )
                for(k = 0; k < j; k++)        // n
                    sum++;
```

- **Worst Case: $O(N^4)$**
  - The i-loop will execute n times.
  - The j-loop will execute up to i*i times, which at its largest degree, means that the j-loop execute n^2 times.
  - The k-loop will execute depending on the if-statement, which is determined by j%i == 0 ; when j is at its largest (i*i = n^2), then you have i^2%i = 0, so the k-loop will execute up to a max when j = i^2, thus k-loop executes at a largest degree of n^2 : n + 2n + 3n + .. n^2. Instead of k executing n^2 times however, because of the % operator n loops are skipped, thus we can reduce n^2 to n.
  - So, worst case becomes: n * n^2 * n = n^4

(7)

```
int sum (int n)   {
    if n == 1 {
        return 1;
    }
    return n + sum(n-1);
}
```

- **Worst Case: O(N)**
  - There will be a recursive call of the sum function:
  - n will be reduced by 1 every recursive call; thus there will be approximately n recursive calls for this function.

(8)

```
int sum (int n)
{
   if (n <= 1 )
      return 1;
   else
      return n + sum((3*n)/5);
}
```

- **Worst Case: O(N)**
  - this function will execute recursive iterations, with each iteration reducing n by 3/5ths, until n is <= 1. With that being considered, as n approaches infinity, the 3/5 factor become irrelevant, and so we see that the growth of the function is n.

<u>Problem 3</u> (9 points)
An algorithm takes 0.5 ms for input size 100. How long will it take for input size 500 if the running time is the following (assuming low-order terms are negligible)?

*Since run time is 100/0.5 = 200 executions in 1 millisecond.

a) Linear
- *x, x = 500; thus, 500 executions will occur; it will take 500/200 =* ***2.5ms***

b) O(NLogN)
- *xlog(x), x =500; thus, 500*log(500) = 1350 executions will occur; it will take 1350/200 =* ***6.75ms***

c) Quadratic
- *x^2, x = 500; thus, 500^2 = 250,000 executions will occur; it will take 250,000/200 =* ***1,250ms***

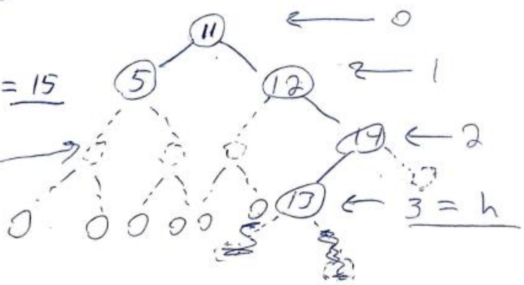
d) Cubic
- *x^2, x = 500; thus, 500^3 = 125,000,000 executions will occur; it will take 125,000,/200 =* ***625,000ms***

Problem 4 (10 points)
Show that the maximum number of nodes in a binary tree of height h is $2^{h+1}-1$.

HW1 problem 4

Show max # of nodes in a BST of height h is $2^{h+1}-1$

- Lets pick a height: $h = 3$
- Formula says max nodes is $2^{h+1}-1$.
- Lets see: $2^{3+1}-1 = 2^4-1 = 15$
- Lets count
- Yes! If we generate a full tree at height 3, we get 15 nodes total.



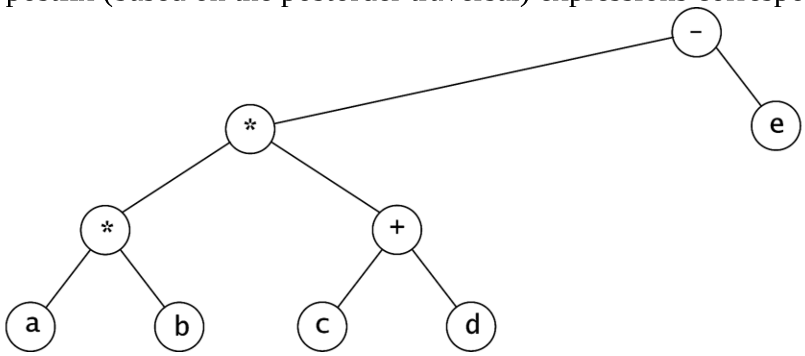
- Also, we know max nodes of each level will be twice the number of nodes in the previous level;

$L_0 = 1$
$L_1 = 2\cdot1$
$L_2 = 2\cdot2$
$L_3 = 2\cdot2\cdot2$
$L_4 = 2\cdot2\cdot2\cdot2$

- Each level can hold 2^{level} nodes max.
- So we can say level $= h+1$, and subtract 1 for the 0-level; the $2^{h+1}-1$ = MAX nodes

<u>Problem 5</u> (10 points)

Give the prefix (based on the preorder traversal), infix (based on the inorder traversal), and postfix (based on the postorder traversal) expressions corresponding to the following tree:



HW 1 Problem 5

Demetrius Johnson

Prefix (preorder): - * * a b + c d e

Infix (inorder): a * b * c + d - e

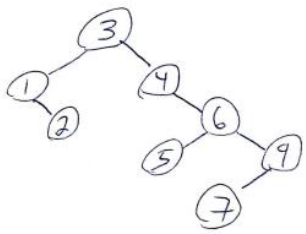
Postfix (post order): a b * c d + * e -

<u>Problem 6</u> (10 points)

a. Show the result of inserting 3, 1, 4, 6, 9, 2, 5, 7 into an initially empty binary search tree.
b. Show the result of deleting the root.

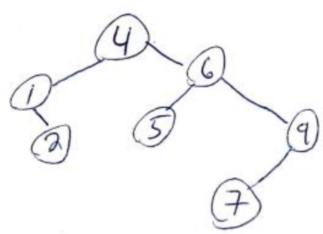
HW 1 Problem 6

(a) Insert: 3, 1, 4, 6, 9, 2, 5, 7 into a BST



(b) show result of deleting root:

I will replace root with smallest value (left-most) in right subtree.
So, 3 gets overwritten with 4.



Problem 7 (15 points)

Write efficient functions that take only a pointer to the root of a binary tree, T, and compute:

a. The number of nodes in T.

```
Int numNodes(BinaryNode* t){

If(t == NULL)
        Return 0;
If(t→left!= NULL || t→right != NULL)
        Return 1+ (numNodes(t→left) + numNodes(t→right));
}
```

b. The number of leaves in T.

```
Int numLeaves(BinaryNode* t){

If(t == NULL)
        Return 0;
If(t→left!= NULL && t→right != NULL)
        Return 1;
Return countLeaves(t→left) + countLeaves(t→right);
}
```

c. The number of full nodes in T.

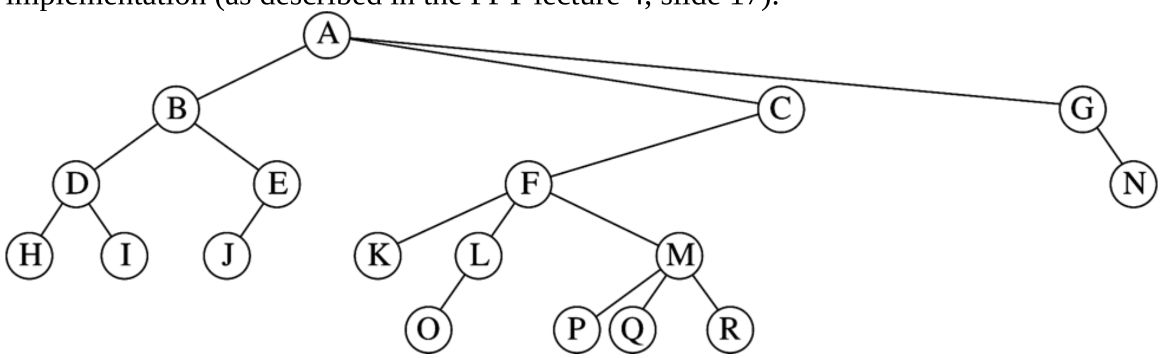
```
Int numFullNodes(BinaryNode* t){

    If(t == NULL)
            Return 0;
    If(t→left!= NULL && t→right != NULL)
            Return 1 + numFullNodes(t→left) + numFullNodes(t→right);
    Return 0;
    }
```

What is the running time of your functions.

O(logN) since they are all binary-search based

Problem 8 (10 points)
Show how the tree in the figure below is represented using a firstChild/nextSibling link implementation (as described in the PPT lecture 4, slide 17).



HW 1 Problem 8

Demetrius Johnson