CSCE 221 Cover Page

Homework Assignment #2

First Name Raymond Last Name Zhu UIN 923008555

User Name rawrbyte E-mail address rawrbyte@tamu.edu

Please list all sources in the table below including web pages which you used to solve or implement the current homework. If you fail to cite sources you can get a lower number of points or even zero, read more on Aggie Honor System Office website: `http://aggiehonor.tamu.edu/`

Type of sources				
People				
Web pages (provide URL)		chegg		
Printed material	CSCE 221 Slide			
Other Sources				

I certify that I have listed all the sources that I used to develop the solutions/codes in the submitted work.
On my honor as an Aggie, I have neither given nor received any unauthorized help on this academic work.

Your Name Raymond Zhu Date 10/18/15

Homework 2

due October 18 at 11:59 pm.

1. (20 points) Linked list questions.

   (a) Write a recursive function in C++ that counts the number of nodes in a singly linked list.

```
int count(ListNode* node){
        if (node == NULL)
                return 0;
        else
                return 1+count(node->next);
}
```

   (b) Write a recurrence relation that represents the running time for your algorithm.

   $f(n) = f(n-1) + 1$
   $f(n-1) = f(n-2) + 1$
   $f(n-2) = f(n-3) + 1$
   $f(n-3) = f(n-4) + 1$
   $f(n-4) = f(n-5) + 1$
   .
   .
   .
   $f(n-k) = f(n-(k+1)) + 1$

   (c) Solve this relation and provide the classification of the algorithm using the Big-O asymptotic notation.

   $f(1) = 1$
   $f(n) = n = O(n)$
   For a node list size of 1, the runtime would simply be running at constant time. In the case of a singly link list of n nodes, the function will run at n times. Given n, the function runs in linear time, where the big-O notation is $O(n)$.

2. (20 points) Write a recursive function that finds the maximum value in an array of int values without using any loops.

```
int find_max(int A[] , int size, int max) {
        if(size==1){
                if(A[0]>max) return A[0];
                else return max;
        }
        else if(size>1){
                if(A[size-1]>max)
                        return find_max(A , size-1, A[size-1]);
                else
                        return find_max(A , size-1, max);
        }
        else{
                cout << "range error"<<endl;
                return -1;
        }
}
```

(a) Write a recurrence relation that represents running time of your algorithm.

$f(n) = f(n-1) + 1$
$f(n-1) = f(n-2) + 1$
$f(n-2) = f(n-3) + 1$
$f(n-3) = f(n-4) + 1$
$f(n-4) = f(n-5) + 1$
.
.
.
$f(n-k) = f(n-(k+1)) + 1$

(b) Solve this relation and classify the algorithm using the Big-O asymptotic notation.

$f(1) = 1$
$f(n) = n = O(n)$

For an array size of 1, the runtime would simply be running at constant time. In the case of an array size of n integers, the function will run at n times. Given n, the function runs in linear time, where the big-O notation is $O(n)$.

1. (10 points) What data structure is the most suitable to determine if a string is a palindrome? A string is a palindrome if it is equal to its reverse. For example, "racecar" and "so many dynamos" are palindromes (spaces are removed from many word strings). Justify your answer. Use Big-O notation to classify the running time of your algorithm.

   We can consider the length of a string is of length n. Using two stacks, one of size n and the other of size n/2. The second stack of size n/2 is a ceiling. If we push the the string into a stack of chars, this will require a runtime of O(n). We can then pop half of the stack size and push the chars into a second stack of size n/2 (the other half of the string). Now that we have both halves of the string in two stacks. For every pop, we can compare the chars of each stack simultaneously. The pop function of a stack data structure requries a runtime of O(n). If the string n is of odd length, middle character should be removed by pop before comparing the two stacks. Overall, the runtime complexity of using two stacks to compare if a string is a palindrome requires a total runtime of O(n).

2. (10 points) Solve C-5.2 on p. 224

   The solution is to actually use the queue Q to process the elements in two phases. In the first phase, we iteratively pop each of the elements from S and enqueue it in Q, and then we iteratively dequeue each element from Q and push it into S. This reverses the elements in S. Then we repeat this same process, but this time we also look for the element x. By passing the elements through Q and back to S a second time, we reverse the reversal, thereby putting the elements back into S in their original order.

1. (20 points) What is the amortized cost of the stack push operation when the additional stack-array memory is allocated by each of these two strategies? Do calculations to support your answer.

   (a) Doubling strategy – double the size of the stack-array memory if more memory is needed.

   Assuming that the initial capacity of a dynamic array is 1. If the array is full we double its capacity so that the sequence goes 1, 2, 4, 8, 16, ... The total cost of n push operations is $\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{log_2 n} 2^j < n + 2n = 3n$, since there are at most n operations that cost 1 and the cost of the copy operations form a geometric series. The amortized cost of a single push operation is 3n/n = 3, that is, it is O(1).

   (b) Incremental strategy – increase the size of the stack-array by a positive constant c if more memory is needed.

   If there is no space for a new object, we can increase the size by a constant c. To insert n objects, we need to replace the array k = n/c times. The total time T(n) of a series of n insert operations is proportional to n + c + 2c + 3c + 4c + ... + kc = n + c(1 + 2 + 3 + ... + k) = n + ck(k + 1)/2.

   Since c is a constant, T(n) is O(n+k^2), i. e., O(n^2). The amortized time of an insert operation is O(n)

1. (10 points) Describe (in pseudo code) how to implement the stack ADT using two queues. What is the running time of the push and pop functions in this implementation?

```
void push(T obj){
        Q1.enqueue(obj);
}
T pop(){
        queue Q2;
        while(!Q1.size()>1){ // copy n-1 elements from Q1 to Q2
                Q2.enqueue(Q1.dequeue());
        }
        T obj= Q1.dequeue(); // pop last element in Q1
        while(!Q2.isEmpty()){ // copy Q2 to Q1
                Q1.enqueue(Q2.dequeue());
        }
        return obj;
}
```

The runtime function of the can be written as $f(n) = 4n + 1$.

To write the runtime function in big-O notation, we can see that its a linear runtime

We can write the runtime as $O(n)$.

2. (10 points) Solve C-5.8 on p. 224

```
Input: A string inputString[]
Output: Postfix notation of inputString[]
Assume that the operands have the following precendencies:

Operands %, *, / have precendence 2
Operands +,- have precendence 1

Algorithm EvaluatePostfix(inputString[]):
for count = 0 to inputString[count]!= null, do
If the value at inputString[count] is an operand, print it.

if inputString[count] is an operand (+, -, *, /, %), then
        print inputString[count]
        else Operand '(' has lowest precendence in the stack, whereas it
           has the highest precendence outside.
        if inputString[count] = '(', then
StackS.push inputString[count]
        else if inputString[count] = ')' then with the help of a while
           loop check whether the stack still has elements and the top of
            the stack is not the operand '('. Inside the loop, pring the
           top of the stack and remove the top of the stack.

while !StackS.empty() && stackS.top() != '(' do
print StackS.top()
StackS.pop()

If the stack stack is not empty, remove the top elements.
```

```
if !StackS.empty(), then
StackS.pop()
If the character doesn't match '(', print an error.
else print error

If the stack has more elements and the precendence of the top of the stack
     is less than the precedence of inputString[count], then
         insert inputString[count] into the stack.
         else if StackS.empty() || precedence of StackS.top() < precedence
            of inputString[count], then StackS.push(inputString[count])

if the precedence of the top of the stack is higher than inputString[count
   ],
         then print the top of the stack and remove it from its position,
            till either the stack is empty or the precedence of
            inputString[count] becomes higher than that of StackS.top()
         else if precedence of StackS.top() >= precedence of inputString[
            count], then
while !StackS. empty() && precedence of StackS.top()>= precedence of
   inputString[count] do
         print StackS.top()
         StackS.pop()
Insert inputString[count] in the stack with the function StackS.push()
StackS.push(inputString[count])
else check for errors
Print all the elements from the top of th estack and remove them from the
   stack, till the stack contains no more elements

while !StackS.empty() do
         print StackS.top()
         StackS.pop()
```

1. (20 points) Consider the quick sort algorithm.

   (a) Provide an example of the inputs and the values of the pivot point for the best, worst and average cases for the quick sort.

   - Suppose the size of the data is n.
     - Best Case (The pivot is the middle of all the data, thus same number of elements on each sides of the pivot)
     - Average Case (suppose the left side of pivot has k*n elements and right side of pivot has (1-k)*n elements)
     - Worst Case (when the pivot is the minimum or maximum element in the data set)

     For example take array A with elements: 1, 2, 3, 4, 5. If the pivot were 5 (max value) then it would have to sort 4, 3, 2, 1 because they are all values less than 5. The best case for quick sort is when the pivot point always splits the array evenly. For example, 1, 2, 3, 4, 5, the pivot point 3 would separate it the array evenly and so forth. The average case for quick sort is when the pivot point is chosen at random.

   (a) Write a recursive relation for running time function and its solution for each case.

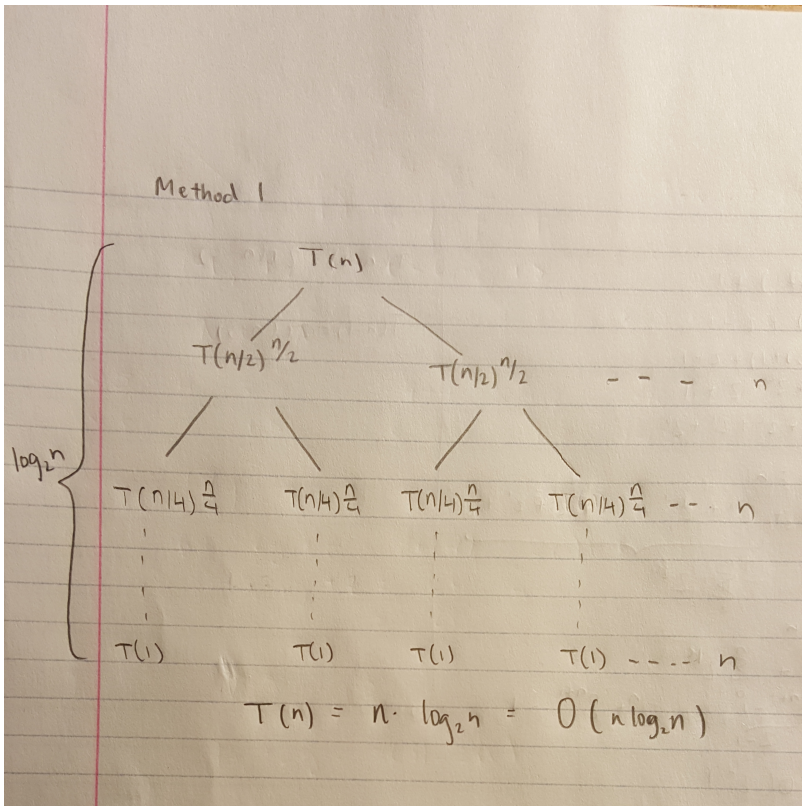
   - Best
     - $f(n) = f(n/2) + f(n/2) + n$
     - $f(n) = 2 * f(n/2) + n$
     - $f(1) = 0$
     - $f(n) = O(n * log(n))$
   - Average
     - $f(n) = f(k * n) + f((1 - k) * n) + n$ where k <0.5
     - $f(n) = O(n * log(n))$
   - Worst
     - $f(n) = f(n - 1) + f(1) + n$
     - $f(n) = f(n - 2) + (n - 1) + n$
     - $f(n) = f(n - 3) + (n - 2) + (n - 1) + n$
     - $f(n) = 0$
     - $f(n) = n(n + 1)/2 = O(n^2)$

1. (15 points) Consider the merge sort algorithm.

   (a) Write a recurrence relation for running time function for the merge sort.
   $T(n) = 2T(n/2) + \Theta(n)$
   $T(1) = \Theta(1)$

   (b) Use two methods to solve the recurrence relation.

   - Method I



   - Method 2
     - According to the Master Method:
       * $T(n) = aT(n/b) + f(n)$
       * Then T(n) can be bounded asymptotically as follows:
         · $T(n) = \Theta(n^{(logb(a))} * log(n))$ if $f(n) = \Theta(n^{(logb(a))})$
         · We have a = 2, b = 2, $f(n) = \Theta(n)$. log2 = 1 and $n^1 = n$. So the merge sort requires $T(n) = nlog(n) = O(nlog(n))$ comparisons

   (c) What is the best, worst and average running time of the merge sort algorithm? Justify your answer.

   There is no best case or worst case for the merge sort. This is because the merge sort needs a duplicate copy of the array being sorted therefore in many situations it can be impractical