Assignment 3 - CS5343.001. (PRD190001)

1. Linked lists and arrays:

a. What are some advantages of linked lists versus arrays?

- Size of a linked list is dynamic as it doesn't have ordered storage, that is it can grow or shrink at run time and doesn't have a fixed compile time contiguous memory allocation like array.
- Insertion and deletion of the elements to the front of the list takes, O(1), time whereas adding elements to the front of the array takes O(N). As an array needs to be shifted every time an element is added or deleted, doing this by managing pointers is more efficient in linked list.
- Linked list is also is more space efficient as it grows dynamically in runtime.

b. What are some advantages of arrays versus linked lists?

- As array is a data structure that has contiguous memory allocation, it supports random access, that is retrieval of any item in an array is possible in constant time, whereas this operation in a linked list is a linear/sequential.
- Binary search becomes easier in an array due to the contiguous memory allocation.
- Due to the indexing, there's no need to store pointers to next and previous elements, thus not wasting memory in that front.

2. What is the Big-O running time of the following code fragment?

Assume lst1 has N items, and lst2 is initially empty.

```
public static void add( List<Integer> lst1, List<Integer> lst2)
{
  for ( Integer x : lst1 )
    lst2.add(0, x);       // add to front
}
```

a. If an ArrayList is passed for lst1 and lst2. Explain your answer.
Answer: O(N^2), as adding to the 0'th position in an array involves shifting the entire list towards the right. Repeating the adding as the first element for n times leads to this quadratic time complexity.

b. If a LinkedList is passed for lst1 and lst2. Explain your answer.
Answer: O(N), As linked list stores the head pointer, and adding another node towards the front of a list is a constant time operation, thus adding n times to 0'th location, would lead to this linear time complexity.

3. What is the Big-O running time of the following code fragment?

```
public static void erase( List<Integer> lst )
{
  Iterator<Integer> itr = lst.iterator();

  while ( itr.hasNext() )
  {
    Integer x = itr.next();
    itr.remove();
  }

}
```

a. If an ArrayList is passed for lst. Explain your answer.
Answer: O(N^2), as the while loop will execute for n times, and the removal of one element from the front takes linear time in an array, as it will delete one element and then shift every other element. This linear time multiplied with the linear time of the while loop leads to a quadratic time complexity for the above code.

b. If a LinkedList is passed for lst. Explain your answer.
Answer: O(N), as the while loop will execute for n times, and the removal of one element from the front takes constant time, as just the pointer allocation needs to be changed. The above code when passed a LinkedList would execute in linear time complexity.

4. What is the Big-O running time of the following code fragment?

Assume lst1 has N items, and lst2 has N items.

```
public static int Count( List<Integer> lst1, List<Integer> lst2)
{
  Iterator<Integer> itr1 = lst1.iterator();

  int count=0;
  while ( itr1.hasNext() )
  {
    Integer x = itr1.next();
    Iterator<Integer> itr2 = lst2.iterator();
    while ( itr2.hasNext() )
      if ( x.equals( itr2.next()) )
        count++;
  }

  return count;
}
```

The above code is taking two lists and comparing every element of list one with every element in list two and is increasing the count if the two elements are same. So essentially the above code works as a nested for loop.

a. If an ArrayList is passed for lst1 and lst2. Explain your answer.
Answer: O(N^2) As the above code acts like a nested for loop, the internal while loop will go from 0 to N for each element in the iter1, thus leading to a quadratic time complexity.

b. If a LinkedList is passed for lst1 and lst2. Explain your answer.
Answer: O(N^2), same is the case with linked list, as the iter1 will go from 0 to N and the internal while loop will take iter2 from 0 to N for each element of iter1 thus leading to this quadratic time complexity.

5. What is the Big-O running time of the following code fragment?

```
public static int calc( List<Integer> lst )
{
  int count = 0;
  int N = lst.size();

  for ( int i=0; i<N; i++)
  {
    if (lst.get(i) > 0)
      sum += lst.get(i);
    else
      sum += lst.get(i) * lst.get(i);
  }
  return sum;
}
```

a. If an ArrayList is passed for lst. Explain your answer.
Answer: O(N), the get(i) in an array is just using random access, which is O(1) thus everything inside the for loop is constant time, hence for an array list this is linear time complexity.

b. If a LinkedList is passed for lst. Explain your answer.
Answer: O(N^2), the get(i) for a linked list takes linear time, so further evaluating the if condition we get O(N) + O(N), which is O(N) and in the else also leading to O(N) from evaluating O(N) + O(1) + O(N). Thus, multiplying the O(N) of the for loop by the bigger branch that is O(N) in this case, we get O(N^2) that is a quadratic time complexity.

6. Suppose a Java method receives a List<Integer> and reverses the order of the items it contains by removing each item from the front of the list and pushing it onto a Stack<Integer>, and then popping the items from the stack and inserting each item to the end of the list.
What is the expected Big-O running time if:

a. If an ArrayList is passed. Explain your answer.
Answer: O(N^2)
The above code for array list would look like:

1. For every element in array:
    a. Get the element. O(1)
    b. Adjust the array list. O(N)
    c. Push element in stack. O(1)
2. For every element in the stack: O(N)
    a. Pop the element O(1)
    b. Push it at the end of the array list. O(1)

Thus step one takes O(N) multiplied by O(N) that is O(N^2) and step two takes O(N) thus resulting in the overall time complexity dominated by step 1 that is O(N^2) or quadratic time complexity.

b. If a LinkedList is passed. Explain your answer.
Answer: O(N)
The above code for linked list would look like:

1. While list is not empty: O(N)
    a. Get the first element O(1)
    b. Remove the first element O(1)
    c. Push the first element in stack O(1)
2. While the stack is not empty: O(N)
    a. Pop the element from the stack O(1)
    b. Attach the element to the tail O(1) (assuming we have reference to the tail)

Thus as both step one and two take O(N) the time complexity for the above code is a linked list is passed is O(N), or linear time complexity.

7. Show each step of converting a+b*c+(d-e) from infix to postfix notation, using the algorithm described in the textbook that uses a stack.
Answer:

Expression: a+b*c+(d-e) (Treat the stack left to right as bottom to up)

Step 1:
Output: a
Stack:

Step 2:
Output: ab
Stack:

Step 3:
Output: ab
Stack: +

Step 4:
Output: ab
Stack: +*

Step 5:
Output: abc*
Stack: +

Step 6:
Pop the + in the stack and put a new + in the stack
Output: abc*+
Stack: +

Step 7:
Output: abc*+
Stack: +(

Step 8:
Output: abc*+d
Stack: +(-

Step 9:
Output: abc*+de-
Stack: +(

Step 10:
Output: abc*+de-
Stack: +

Step 11:
Output: **abc*+de-+**
Stack:

The postfix expression is **abc*+de-+**