CSCE 221 Cover Page

Homework #1

Due February 11 at midnight to eCampus

First Name: Pratik Last Name: Patel UIN: 527004337

User Name: p.pratik99 E-mail address: p.pratik99@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: Aggie Honor System Office

Type of sources			
People			
Web pages (provide URL)			
Printed material	Textbook		
Other Sources	Class Notes		

I certify that I have listed all the sources that I used to develop the solutions/codes to the submitted work.

"On my honor as an Aggie, I have neither given nor received any unauthorized help on this academic work."

Your Name Pratik Patel Date 02/11/2019

Type solutions to the homework problems listed below using preferably LyX/LaTeX word processors, see the class webpage for more information about their installation and tutorials.

1. (10 points) Write the C++ classes called `ArithmeticProgression` and `GeometricProgression` that are derived from the abstract class `Progression`, with two pure virtual functions, `getNext()` and `sum()`, see the course textbook p. 87–90 for more details. Each subclass should implement these functions in order to generate elements of the sequences and their sums. Test your program for the different values of d, r and the number of elements n in each progression.
What is the classification of those functions: `getNext()` and `sum()` in terms of the Big-O notation?

   Recall the definitions of the arithmetic and geometric progressions.

   Definition: An *arithmetic progression* with the initial term a and the common real difference d is a sequence of the form

   $a, a+d, a+2d, \ldots, a+nd, \ldots$

   Definition: A *geometric progression* with the initial term a and the common real ratio r is a sequence of the form

   $a, ar, ar^2, \ldots, ar^n, \ldots$

   Solution:

```
#include <iostream>
#include <math.h>

using namespace std;

class Progression
{
        Progression(a, d)
        {
                this.a = a;
                this.d = d;
        }
public:
        virtual int sum() = 0;
        virtual int getNext(int current) = 0;

private:
        int a;
        int d;
}

class ArithmeticProgession : public Progression
{
        ArithmeticProgession(a, d) : Progression(a, d){ ;}

        int sum(n)
        {
                int totalA = n*a;
                int totalDifference = ((n * (n + 1)) / 2) * d;

                return totalA + totalDifference;
        }
        int getNext(int current)
        {
                return current + d;
        }
```

```
}

class GeometricProgession : public Progression
{
        GeometricProgession(a, d) : Progression(a, d){ ;}

        int sum(n)
        {
                return ((a * (1 - pow(d, n))) / (1 - d));
        }
        int getNext(int current)
        {
                return current * d;
        }
}
```

Both the functions' for both class have Big-O notation of $O(1)$.

2. (10 points) Use the STL class `vector<double>` to write a C++ function that takes two vectors, `a` and `b`, of the same size and returns a vector `c` such that $c[i] = a[i] \cdot b[i]$. How many scalar multiplications are used to create elements of the vector `c` of size n? What is the classification of this algorithm in terms of the Big-O notation?

Solution:

```
#include <vector>

using namespace std;

vector<int> multiplyVectors(vector<int> a, vector<int> b)
{
        vector<int> c;
        if(a.size() == b.size())
        {
                for(int i = 0, i < a.size(); ++i)
                        c.push_back(a.at(i) * b.at(i));
        }
        else
                cout << "The vectors passed in this function are not of same size." <<

        return c;
}
```

There are n numbers of scalar multiplications. The Big-O notation of this alogorithm is $O(n)$.

3. (10 points) Use the STL class `vector<int>` to write a C++ function that returns true if there are two elements of the vector for which their product is odd, and returns false otherwise. Provide a formula on the number of scalar multiplications in terms of n, the length of the vector, to solve the problem in the best and the worst cases. Describe the situations of getting the best and worst cases. What is the classification of the algorithm in the best and worst cases in terms of the Big-O notation?

Solution:

```
#include <iostream>
#include <vector>

using namespace std;

bool oddVector(vector<int> a)
{
        for(int i = 0; i < a.size(); ++i)
        {
                if(a.at(i) % 2 == 0)
                        return false;
        }

        return true;
}
```

We can get the result we want, without any multiplications. Because, if we multiply an even number with any number, the result will be even. Therfore, if our vector does not contain any even number than the result will be true, else it will be false. The best case for this algorithm is that the first element in the vector is an even number. The worst case is that the last element is an even number or all the elements are odd. The Big-O notation in the best case is $O(1)$ and in the worst case is $O(n)$.

4. (20 points) Write a templated C++ function called `BinarySearch` which searches for a target `x` of any numeric type `T`, and test it using a sorted vector of type `T`. Provide the formulae on the number of comparisons in terms of n, the length of the vector, when searching for a target in the best and the worst cases. Describe the situations of getting the best and worst cases. What is the classification of the algorithm in the best and worst cases in terms of the Big-O notation?

Solution:

```
#include <iostream>
#include <vector>

using namespace std;

<template typename T>
int BinarySearch(T x, vector<T> a)
{
        int middle;
        int lower = 0;
        int higher = (int) a.size() - 1;

        while(lower <= higher)
        {
                middle = (lower + higher) / 2;

                if(a.at(middle) < x)
                        lower = middle + 1;
                else if (a.at(middle) > x)
                        higher = middle - 1;
                else
                        return middle;
        }

        return -1;
}
```

Best case for this algorithm is that the target is in the middle of the array. In the best case there will be only 3 comparisions. In the worst case scenario it will have $3nlog(n)$ comparisons. In the best case, the Big-O notation is $O(1)$. In the worst case, it is $O(nlog(n))$

5. (10 points) **(R-4.7 p. 185)** The number of operations executed by algorithms A and B is $8n \log n$ and $2n^2$, respectively. Determine n_0 such that A is better than B for $n \geq n_0$.

Solution:

$n_0 = 1$. Because for every $n \geq 1, 8nlog(n) < 2n^2$

6. (10 points) **(R-4.21 p. 186)** Bill has an algorithm, `find2D`, to find an element x in an $n \times n$ array A. The algorithm `find2D` iterates over the rows of A, and calls the algorithm `arrayFind`, see Code Fragment 4.5, p. 184, on each row, until x is found or it has searched all rows of A. What is the worst-case running time of `find2D` in terms of n? What is the worst-case running time of `find2D` in terms of N, where N is the total size of A? Would it be correct to say that `find2D` is a linear-time algorithm? Why or why not?

Solution:

The worst case running time of the algorithm, in terms of n, is $O(n^2)$. If $N = n^2$, then the Big-O notation is $O(N)$. Here, Big-O is linear because N is only of power 1.

7. (10 points) **(R-4.39 p. 188)** Al and Bob are arguing about their algorithms. Al claims his $O(n \log n)$-time method is **always** faster than Bob's $O(n^2)$-time method. To settle the issue, they perform a set of experiments. To Al's dismay, they find that if $n < 100$, the $O(n^2)$-time algorithm runs faster, and only when $n \geq 100$ is the $O(n \log n)$-time algorithm better. Explain how this is possible.

Solution:

If Al's algorithm does $10nlog(n)$ comparisions and Bob's algorithm does n^2 comparisions, then $10nlog(n) > n^2$ for $n < 100$.

8. (20 points) Find the running time functions for the algorithms below and write their classification using Big-O asymptotic notation. The running time function should provide a formula on the number of operations performed on the variable s. Note that array indices start from 0.

```
Algorithm Ex1(A):
  Input: An array A storing n ≥ 1 integers.
  Output: The sum of the elements in A.
s ← A[0]
for i ← 1 to n − 1 do
    s ← s + A[i]
end for
return s
```

Solution:

Number of Operation = $2(n-1)+1$
Big-O Notation = $O(n)$

```
Algorithm Ex2(A):
  Input: An array A storing n ≥ 1 integers.
  Output: The sum of the elements at even positions in A.
s ← A[0]
for i ← 2 to n − 1 by increments of 2 do
  s ← s + A[i]
end for
return s
```

Solution:

Number of Operation = $2(\frac{(n-1-2+1)}{2})+1$
Number of Operation = $n-1$
Big-O Notation = $O(n)$

```
Algorithm Ex3(A):
   Input: An array A storing n ≥ 1 integers.
   Output: The sum of the partial sums in A.
s ← 0
for i ← 0  to n − 1 do
   s ← s + A[0]
   for j ← 1 to i do
     s ← s + A[j]
   end for
end for
return s
```

Solution:

Number of Operations = $2n + 2(\frac{n(n-1)}{2})+1$
Number of Operations = n^2+n+1
Big-O Notation = $O(n^2)$

Algorithm Ex4(A):
    Input: An array A storing $n \geq 1$ integers.
    Output: The sum of the partial sums in A.
$t \leftarrow 0$
$s \leftarrow 0$
for $i \leftarrow 1$ **to** $n-1$ **do**
    $s \leftarrow s + A[i]$
    $t \leftarrow t + s$
end for
return t

Solution:

Number of Operations: $2(n-1)+1$
Big-O Notation = $O(n)$