# CSCE 221 Cover Page

## Homework #1

Due February 11 at midnight to eCampus

First Name: Asa Last Name: Hayes UIN: 525003952

User Name: AsaHayes E-mail address: asahayes@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 | Josiah Egner | | |
| Web pages (provide URL) | | | |
| Printed material | | | |
| Other Sources | | | |

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 Asa R Hayes Date 10 February, 2018

**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.**

I apologize for the improper numbering of the questions, I'm not sure what caused it or how to fix it.

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

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

```
#include <cmath> // for Math.pow in the geometric progression
class ArithmeticProgression : Progression{

    double initTerm, curTerm, comRealDif;
    int n;
    ArithmeticProgression() : initTerm(1), comRealDif(1), n(1){
        curTerm = initTerm;
    }
    ArithmeticProgression(double a, double d){
        initTerm = a;
        comRealDif = d;
        curTerm = initTerm;
    }
    double getNext(){
        n++;
        curTerm = a + (n * comRealDif);
        return curTerm;
    }
    double sum(){
        double sum = a;
        for(i = 0; i < n-1; i++){
            sum += a + (i * comRealDif);
        }
        return sum;
    }

}
```

```
class GeometricProgression : Progression{
    double initTerm, curTerm, comRealRat;
    int n;
    GeometricProgression() : initTerm(1), comRealRat(1), n(1){
       curTerm = initTerm;
    }
    GeometricProgression(double a, double r){
       initTerm = a;
       comRealRat = r;
       curTerm = initTerm;
    }
    double getNext(){
       n++;
       curTerm = a * Math.pow(comRealRat, n);
       return curTerm;
    }
    double sum(){
       double sum = a;
       for(i = 0; i < n-1; i++){
          sum += a * Math.pow(comRealRat, i);
       }
       return sum;
    }
}
```

The function getNext() is O(1) as it does the same amount of operations regardless of input. The function sum() is O(n) because it loops thru the same set of operations n times.

1. (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?

```
vector<int> MultVectors(vector<int> a, vector<int> b){
    vector<int> c;
    for(int i = 0; i < a.size(); i++){
       c.push_back(a[i]*b[i]);
    }
    return c;
}
```

If n is the size of a and b, then this algorithm is O(n). The loop performs 1 scalar multiplication for every element in a and b, so there are a total of n scalar multiplications.

1. (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?

```
boolean oddCheck(vector<int> a){
    for(int i = 0; i < a.size() - 1; i++){
       for(int j = i + 1; j < a.size(); j++){
          if( ( (a[i] * a[j]) % 2 ) != 0 ){
            return true;
          }
       }
    }
    return false;
}
```

The best-case formula for the number of scalar multiplications would be if the first two items in "a" had an odd product, where the formula would be f(n) = 1 (O(1)). The worst-case formula would be if either only the last two elements of "a" had an odd product or if no elements of a had an odd product, so f(n) = n! (O(n!)).

1. (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?

```
template <typename T>
int GenBinSearch(vector<T> a, T x){
    int bot = 0, top = a.size()-1, mid = (a.size()-1) / 2;
    // early return if element is not within vector's sorted range
    if(x < a[0] || x > a[a.size()-1]){
       return -1;
    }
    // main sort loop
    while(top > bot){
       mid = bot + ( (top - bot) / 2 );
       if(x == a[mid]){
          return mid;
       }else if(x > a[mid]){
          bot = mid+1;
       }else{
          top = mid-1;
       }
    }
return -1;
}
```

Best Case: The best case would be if x was the center element of "a". As it would return instantly, the running time would be O(1).

Worst Case: The worst case would be if x was either the first or last element of "a" or not within a. If n is the size of the array, this function would be O(log n).

1. (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 $n \geq n_0$.

As the graph lines for these algorithms intersect before $2n^2$ grows noticeably faster, we can determine $n_0$ as the point where the two lines intersect.

$8n \log n = 2n^2$
$4 \log n = n$
$\frac{n}{\log n} = 4$
$\frac{16}{\log 16} = 4$
16 = n

Because they are equal at n = 16, $2n^2$ (i.e. B) starts growing faster at n = 17.

1. (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?

find2D by itself would have a worst case time of O(n), as it only loops from 0 to n while performing just 1 operation (comparison) per iteration. This worst case would be if x is the last element or is not contained within A.

The worst case for find2D over N would be $O(n^2)$, as it would be looping n times through arrays of size n, doing the same number of operations per iteration. This worst case would be if x were A[n-1][n-1] (i.e. the last element of A) or not in A.

By itself, the function find2D runs in linear time, as it is O(n). The number of operations increases linearly and directly corresponds to the size of input array A.

1. (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.

As can be seen in Problem 5 of this assignment, one can have a function that is $O(n^2)$ that runs better than $O(n \log n)$. In Problem 5, the algorithm of time $2n^2$ performs better than the algorithm of time $8n \log n$ from n = 0 to n = 16, which has been achieved by changing the coefficient of each algorithm's time function. Through some trial and error on graphing software, the $O(n \log n)$ algorithm for this question could be around $65n \log n$ and the $O(n^2)$ around $3n^2$ to meet the conditions described.

1. (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
```

The running time function for this would be f(n) = n - 1, as it iterates through n, but does the first comparison out of the loop. This function is 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
```

The running time algorithm for Ex2 would be f(n) = $\frac{n}{2}$, as it covers the same circumstances as Ex1 but with half the elements. This function is also O(n). This function would not work properly for arrays of size 1 or 2 due to the indexing used.

```
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
```

The running time algorithm for Ex3 is f(n) = n + n!. For each iteration i of the main loop (from 0 to n-1), there is also another loop (from 1 to i) added on. This function is O(n!).

```
Algorithm Ex4(A):
   Input: An array A storing n ≥ 1 integers.
   Output: The sum of the partial sums in A.
t ← 0
s ← 0
for i ← 1 to n − 1 do
   s ← s + A[i]
   t ← t + s
end for
return t
```

The running time algorithm for Ex4 is f(n) = 2n - 1. It is the same as Ex1, but does one more operation per iteration through n. This function is O(n).