# CSCE 221 Cover Page

## Homework #1

**Check the Canvas Calendar for deadlines. The homework submission is to Gradescope**

First Name Chase Last Name Albright

UIN 529008060

E-mail address Chasealbright@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 | | | |
| 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 Chase albright Date 2/1/2022

**Homework 1 Objectives:**

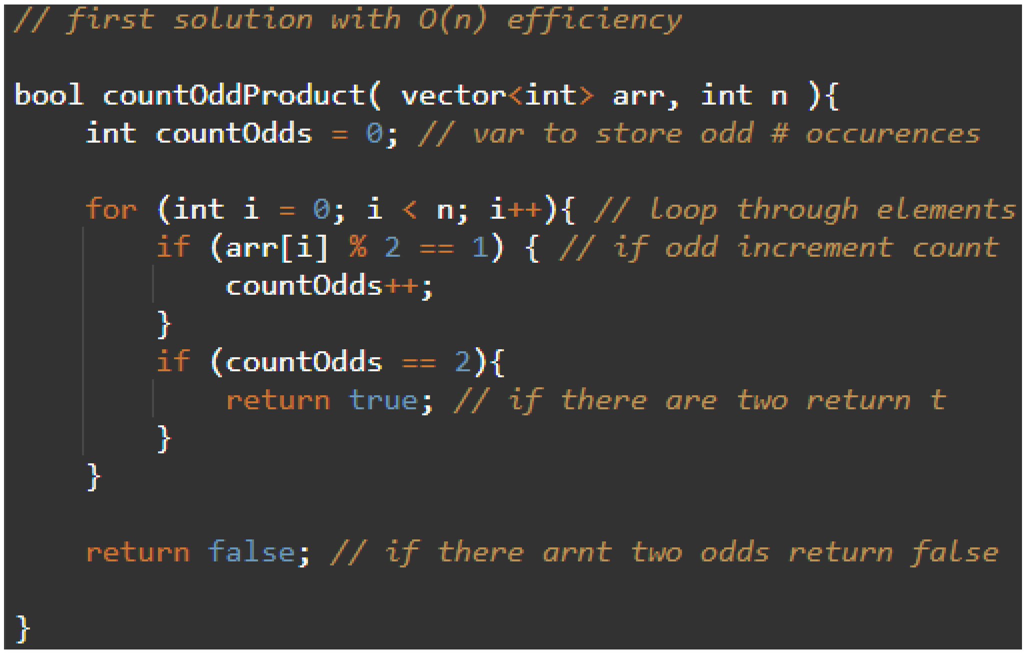
1. Developing the C++ programming skills by using
   (a) templated dynamic arrays and STL vectors.
   (b) exceptions for reporting the logical errors or unsuccessful operations.
   (c) tests for checking correctness of a program.
2. Comparing theory with a computation experiment in order to classify algorithm.
3. Preparing reports/documents using the professional software LyX or LaTeX.

**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. (25 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 two algorithms for solving this problem with the efficiency of $O(n)$ for the first one and $O(n^2)$ for the second one where $n$ is the size of the vector.

   Justify your answer by writing the running time functions in terms of $n$ for both the algorithms.

   - What do you consider as an operation for each algorithm?
   - Are the best and worst cases for both the algorithms the same in terms of Big-O notation? Justify your answer.
   - Describe the situations of getting the best and worst cases, give the samples of the input for each case and check if your running time functions match the number of operations.

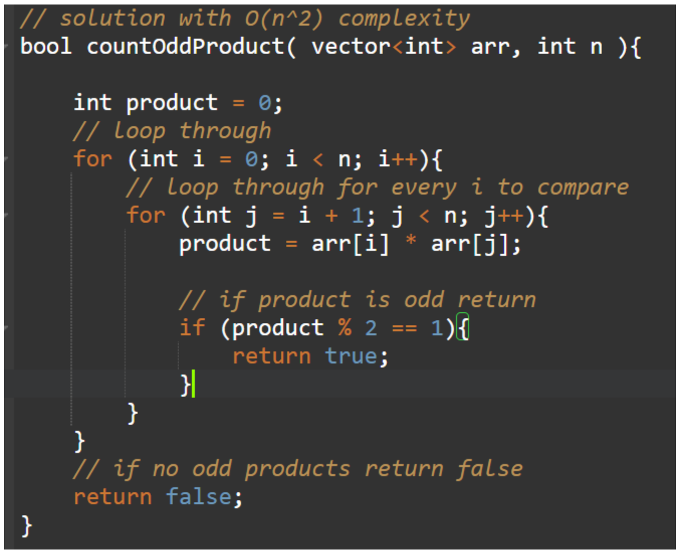


```
// first solution with O(n) efficiency

bool countOddProduct( vector<int> arr, int n ){
    int countOdds = 0; // var to store odd # occurences

    for (int i = 0; i < n; i++){ // loop through elements
        if (arr[i] % 2 == 1) { // if odd increment count
            countOdds++;
        }
        if (countOdds == 2){
            return true; // if there are two return t
        }
    }

    return false; // if there arnt two odds return false

}
```

1) For this algorithm, the i < n operation is the operation impacting the time complexity all other operations are constant and have little impact.

2) For this algorithm, the best case is constant O(1) if the first two elements are odds, the worst case is if the loop traverse through all i < n times giving it O(n) complexity

3) An example of O(1) complexity could be [3,5,4,2] the first two values are odd so the function terminates and returns true. An example of O(n) complexity would be a much longer array of n size with odds at the last two indexes, or no odds at all.



```
// solution with O(n^2) complexity
bool countOddProduct( vector<int> arr, int n ){

    int product = 0;
    // loop through
    for (int i = 0; i < n; i++){
        // loop through for every i to compare
        for (int j = i + 1; j < n; j++){
            product = arr[i] * arr[j];

            // if product is odd return
            if (product % 2 == 1){
                return true;
            }
        }
    }
    // if no odd products return false
    return false;
}
```

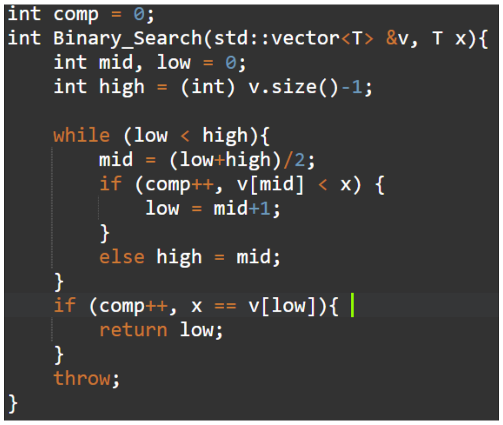
1) Since there are two for loops, each iteration and comparison through the array counts as an operation ( i*j) = O(n^2)

2) For this algorithm the best case is if the first two comparisons are odd giving it O(1) time. The worst case scenario is if there is no odd pair and we go through two nested loops, giving us O(n^2) time.

3) An example of the best case scenario would be an array with the first two elements are odd [3,5,4,4,2] the time would be constant O(1) since it compares the first two elements. An example of the worst case would be if the array contains no odds and it loops through both loops giving it O(n^2) time.

2. (50 points) The binary search algorithm problem.

   (a) (5 points) Implement a templated C++ function for the binary search algorithm based on the set of the lecture slides *"Analysis of Algorithms"*.



```
int comp = 0;
int Binary_Search(std::vector<T> &v, T x){
    int mid, low = 0;
    int high = (int) v.size()-1;

    while (low < high){
        mid = (low+high)/2;
        if (comp++, v[mid] < x) {
            low = mid+1;
        }
        else high = mid;
    }
    if (comp++, x == v[low]){
        return low;
    }
    throw;
}
```

   Be sure that before calling `Binary_Search` elements of the vector `v` are arranged in increasing order. The function should also keep track of the number of comparisons used to find the target `x`. The (global) variable `num_comp` keeps the number of comparisons and initially should be set to zero.

   (b) (10 points) Test your algorithm for correctness using a vector of data with 16 elements sorted in ascending order. An exception should be thrown when the input vector is unsorted or the search is unsuccessful.
   What is the value of `num_comp` in the cases when **This is calculated using the algorithm above and a global variable.**

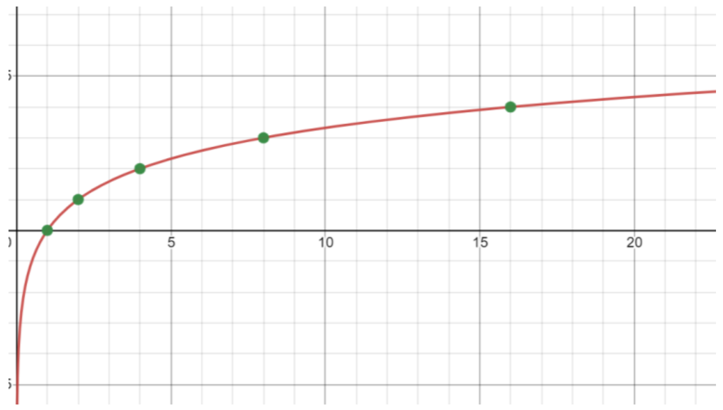
      i. the target `x` is the first element of the vector `v` **`numb_comp = 5`**
      ii. the target `x` is the last element of the vector `v` **`numb_comp = 5`**
      iii. the target `x` is in the middle of the vector `v` **`numb_comp = 2`**

   What is your conclusion from the testing for $n = 16$? **In conclusion values towards the middle of a vector are quicker to find then then values at the beginning or end.**

   (c) (10 points) Test your program using vectors of size $n = 2^k$ where $k = 0, 1, 2, \ldots, 11$ populated with consecutive increasing integers in these ranges: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048. Select the target as the last element in the vector. Record the value of `num_comp` for each vector size in the table below. **This is comparisons are counted using my code from part a. This algorithm follows a log{base2}(n) +1 function very closely.**

| Range $[1,n]$ | Target | # comp. |
|---|---|---|
| [1, 1] | 1 | 1 |
| [1, 2] | 2 | 2 |
| [1, 4] | 4 | 3 |
| [1, 8] | 8 | 4 |
| [1, 16] | 16 | 5 |
| ... | | |
| [1, 2048] | 2048 | 12 |

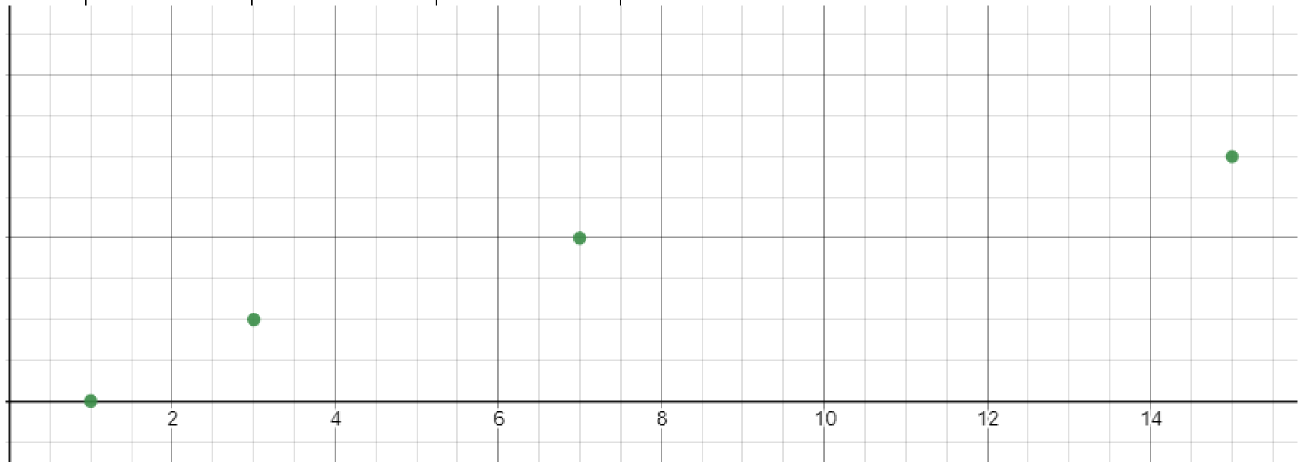
(d) (5 points) Plot the number of comparisons for the vector size $n = 2^k$, $k = 1, 2, \dots, 11$. You can use a spreadsheet or any graphical package. This is the graph of log base2(x)



(e) (5 points) Provide a mathematical formula/function which takes $n$ as an argument, where $n$ is the vector size and returns as its value the number of comparisons. Does your formula match the computed output for any input? Justify your answer. **The function that I found works for this situation would be $f(n) = \log_2 n$ If we count only operations that occur within the while loop, then this function is applicable for any input.**

(f) (5 points) How can you modify your formula/function if the largest number in a vector is not the exact power of two? Test your program using input in ranges from 1 to $n = 2^k - 1$, $k = 1, 2, \dots, 11$ and plot the number of comparisons vs. the size of the vector. **The #comparisons will be same as n+1 for n>1 this is due to the floor logic within the algorithm. This also can be represented by log base2.**

| Range [1,$n$] | Target | # comp. |
|---|---|---|
| [1, 1] | 1 | 1 |
| [1, 3] | 3 | 2 |
| [1, 7] | 7 | 3 |
| [1, 15] | 15 | 4 |
| [1, 31] | 31 | 5 |
| ... | | |
| [1, 2047] | 2047 | 12 |



(g) (5 points) Do you think the number of comparisons in the experiment above are the same for a vector of strings or a vector of doubles? Justify your answer. **The number of comparisons for strings would be much greater than doubles because a vector of strings is essentially an array of arrays. This would drastically increase the comparisons compared to double which is a primitive type and takes only one operation per comparison.**

(h) (5 points) Use the Big-O asymptotic notation to classify binary search algorithm and justify your answer. **The complexity is O(log(n)) this is because after every iteration the area searched is halved mathematically following the logbase2 function.**

(i) (Bonus question—10 points) Read the sections 1.6.3 and 1.6.4 from the textbook and modify the algorithm using a functional object to compare vector elements. How can you modify the binary search algorithm to handle the vector of decreasing elements? What will be the value of `num_comp`? Repeat the search experiment for the smallest number in the integer arrays. Tabulate the results and write a conclusion of the experiment with your justification.

3. (25 points) Find running time functions for the algorithms below and write their classification using Big-O asymptotic notation in terms of $n$. A running time function should provide a formula on the number of arithmetic operations and assignments performed on the variables $s$, $t$, or $c$ (the return value). Note that array indices start from 0.

```
Algorithm Ex1(A):
  Input: An array A storing $n \geq 1$ integers.
  Output: The sum of the elements in A.
$s \leftarrow A[0]$ assignment of s +1 operation
for $i \leftarrow 1$ to $n-1$ do    loop goes n-1 times giving the operation +(n-1)
   $s \leftarrow s + A[i]$
end for
return $s$    Returning s is not an operation in my opinion
```

For this algorithm I counted the assignment of s as an operation and the loop that executes n-1 times. I used this information to find the running time function below. The operation that has the greatest impact on time is n-1 giving a big o of O(n)

A running time function I believe fits is **f(n) = Cs + (n-1)**
In big O notation this would be equal to = **O(n) complexity**

```
Algorithm Ex2(A):
  Input: An array A storing $n \geq 1$ integers.
  Output: The sum of the elements at even positions in A.
$s \leftarrow A[0]$ Assignment of s +1 operation
for $i \leftarrow 2$ to $n-1$ by increments of 2 do loop goes (n-1)*1/2 times
  $s \leftarrow s + A[i]$
end for
return $s$  Returning s is not an operation in my opinion
```

Similar to the first algorithm, the running time is impacted most by the for loop which executes n-1 time but by an increment of 2. This will give us the running time function below and a the big o notation.

A running time function I believe fits is **f(n) = Cs + (n-1)/2**
In big O notation this would be equal to =**O(n)**

```
Algorithm Ex3(A):
    Input: An array A storing $n \geq 1$ integers.
    Output: The sum of the partial sums in A.
$s \leftarrow 0$ assignment operation of s +1 operation
for $i \leftarrow 0$  to $n-1$ do loop of (n-1)operations
    $s \leftarrow s + A[0]$
    for $j \leftarrow 1$ to $i$ do loop of (j < i) times
      $s \leftarrow s + A[j]$
    end for
end for
return $s$       Returning s is not an operation in my opinion
```

This algorithm has a constant assignment of s but has nested for loops. The there is another assignment of s that happens n-1 times and the second for loop that executes when i < j. this will give us the run time function below.

A running time function I believe fits is **f(n) = Cs + Cs(n-1)+ (n-1)(j)**
A Big O notation would be = **O(n^2)**

```
Algorithm Ex4(A):
    Input: An array A storing $n \geq 1$ integers.
    Output: The sum of the partial sums in A.
$t \leftarrow A[0]$ assignment of t
$s \leftarrow A[0]$ assignment of s
for $i \leftarrow 1$ to $n-1$ do loop that executes n-1 times
    $s \leftarrow s + A[i]$
    $t \leftarrow t + s$
end for
return $t$        Returning t is not an operation in my opinion
```

In this algorithm there is two assignments in the beginning and then two assignments within the for loop of s and t. I am only counting the assignment of the variables in the for loop. This will lead to the run time function below.

A running time function I believe fits is **f(n) = Ct + Cs + (Cs+Ct)(n-1)**
In Big O notation this would be = **O(n)**

```
Algorithm Ex5(A, B):
    Input: Arrays A and B storing $n \geq 1$ integers.
    Output: The number of elements in B equal to the partial sums in A.
$c \leftarrow 0$ //counter assignment of c
for $i \leftarrow 0$ to $n-1$ do loop that executes n-1 times
    $s \leftarrow 0$ //partial sum assignment of s n-1 times
    for $j \leftarrow 0$ to $n-1$ do loop that executes n-1 times
        $s \leftarrow s + A[0]$ assignment of s n-1 times
        for $k \leftarrow 1$ to $j$ do loop that executes k < j times
            $s \leftarrow s + A[k]$ assignment of s k<j times
        end for
    end for
    if $B[i] = s$ then
        $c \leftarrow c + 1$ assignment of c and comparison b[i] = s n-1 times
    end if
end for
return $c$       Returning c is not an operation in my opinion
```

This is a complex algorithm with 3 for nested for loops. I counted assignment at the beginning and there was an assignment in almost every loop. If the algorithm reaches the 3rd for loop, this would have the greatest impact on time giving it n cubed time.

A running time function I believe fits is
**f(n) = Cc + Cs(n-1) + Cs(n-1) + Cs(j) + Cc(n-1)+(n-1)(n-1) + (n-1)(n-1)(j)**
This in big O notation would be **= O(n^3)**