CSCE 221 Cover Page

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 https://aggiehonor.tamu.edu/

Name	Blake Lun
UIN	131001178
Email address	blakelunnn@tamu.edu

Cite your sources using the table below. Interactions with TAs and resources presented in lecture do not have to be cited.

People	1. None
Webpages	1. https://www.geeksforgeeks.org/initialize-a-vector-in-cpp-different-ways/ 2. https://www.tutorialspoint.com/What-are-global-variables-in-Cplusplus 3. https://www.youtube.com/watch?v=EfPAxNSZL2U (this is Alex's runtime analysis video) 4. https://adrianmejia.com/how-to-find-time-complexity-of-an-algorithm-code-big-o-notation/
Printed Materials	1. None
Other Sources	1. None

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."*

Name	Blake Lun
Date	9/27/2021

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.
4. Understanding the definition of the big-O asymptotic notation.
5. Classifying algorithms based on pseudocode.

Homework 1

Due September 27 at 11:59 PM

Typeset your solutions to the homework problems preferably in LaTeXor LyX. See the class webpage for 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.

```
//O(n) solution:
bool isOdd(vector<int> numbers, int n)
{
    int count = 0;
    for(int x = 0; x < n; x++)
    {
        if(numbers.at(x) % 2 == 1)
            count++;
        if(count == 2)
            return true;
    }
    return false;
}

//O(n^2) solution:
bool isOddTwo(vector<int> numbers, int n)
{
    int product = 0;
    int count = 0;
    for(int x = 0; x < n; x++)
        for(int y = x + 1; y < n; y++)
        {
            product = numbers.at(x) * numbers.at(y);
                if(product % 2 == 1)
                    return true;
        }
    return false;
}
```

The run time function of the first algorithm is f(n) = 2n + 1. The run time for the second algorithm is $f(n) = (2n^2) + 2$.

For each algorithm, I am counting each line counts as an operation. For example, int count = 0; is one operation and so is product = numbers.at(x) * numbers.at(y); even though there's both multiplication and assigning in this line. This is just to make it more consistent for me.

The best case for both algorithms in Big-O notation is the same. However, the worst case for algorithm two is worse than that of algorithm 1.

For the first algorithm, the best is O(1) where the first two numbers we look at are odd meaning we don't have to go through anymore of the vector. An example of this would be if we had a vector with elements: {1, 1, 2, 2, 2, 2, 2, 2, 2, 2}. The first two elements are odd so there is no need to check the rest of the vector giving us the best possible run time. The worst case for the first algorithm is when there's only one odd number or no odd numbers which gives us time O(n), n being the size of the vector. This is the case because if we never returned true, that means we ran through the whole vector meaning the run time depends on the size or n. An example of this would be if we had a vector with elements: {2, 2, 2, 2, 2, 2, 2, 2, 2, 2}. None of the numbers are odd and we run through each index of the vector giving us the worst possible run time.

For the second algorithm, the best case is O(1) just like algorithm 1. This happens when the first two numbers in the vector are both odd meaning we only have one check just like algorithm 1. An example of this would be if we had a vector with elements: {1, 1, 2, 2, 2, 2, 2, 2, 2, 2} just like algorithm 1. The first two elements are odd so there is no need to check the rest of the vector giving us the best possible run time. The worst case for the second algorithm is when there's only one odd number or no odd numbers just like algorithm 1, except, since this algorithm has a nested loop, there are more comparisons being done. Since there is a nested loop, the worst case is $O(n^2)$. The first loop iterates through each element and the second loops begins iterating one element after where the first loop started making there be more comparisons than the first algorithm. An example of this worst case would be if we had a vector with elements: {2, 2, 2, 2, 2, 2, 2, 2, 2, 2}. The first loop begins at the first number 2 and compares the product of it with all the other elements within the vector. Since all the numbers are even and an even multiplied with an even gives an even, the algorithm will never return true.

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 Binary_Search(vector<int> &v, int x) {
    int mid, low = 0;
    int high = (int) v.size()-1;
    while (low < high) {
        mid = (low+high)/2;
        if (num_comp++, v[mid] < x) low = mid+1;
        else high = mid;
    }
    if (num_comp++, x == v[low]) return low; //OK:
        found
    throw Unsuccessful_Search(); //exception: not
        found
}
```

Be sure that 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.

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

using namespace std;

int num_comp;

class Unsuccessful_Search{};

bool binarySearch(vector<int> v, int x)
{
    num_comp = 0;
    bool found = false;
    int low = 0;
    int mid = 0;
    int high = v.size() - 1;
    while (low <= high)
    {
        num_comp++;
        mid = low + (high - low) / 2;
        if (v[mid] == x)
```

```
            found = true;
        else if (x < v[mid])
            high = mid - 1;
        else
            low = mid + 1;
        if(found)
        {
            cout << "value␣of␣num_comp␣is␣" << num_comp
                << endl;
            return true;
        }
    }
    throw Unsuccessful_Search();
}

int main()
{
    int arr[] = {1, 3, 5, 6, 9, 10, 15, 18, 20, 22, 25,
        30, 36, 41, 55, 60};
    vector<int> vector(arr, arr + (sizeof(arr) / sizeof(
        arr[0])) );

    binarySearch(vector, 1); // first element case
    binarySearch(vector, 60); // end element case
    binarySearch(vector, 18); // middle element case

    return 0;
}
```

(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

(i) the target x is the first element of vector v
num_comp is 4

(ii) the target x is the last element of vector v
num_comp is 5

(iii) the target x is the middle element of vector v
num_comp is 1

What is your conclusion from the testing $n = 16$?
This biggest number of runs that binary search does for a vector the size of 16 is 5 and the smallest is 1.

(c) (10 points) Test your program using vectors of size $n = 2^k$ where $k = 0$, 1, 2, . . . , 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.

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 $n = 2^k$, $k = 1, 2, ..., 11$. You can use a spreadsheet or any graphical package.

(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. log(n) + 1. This formula follows what num_comps turns out to be at the end of the code based on the varying size of the vector. Binary search follows a log(x) pattern and the + 1 is added since if there is only 1 element, log(n) becomes 0, but we just said there's 1 element so the +1 must be added

(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 $n = 2^k - 1, k = 1, 2, ..., 11$ and plot the number of comparisons vs. the size of the vector.

Range [1,n]	Target	# comp.
[1,1]	1	1
[1,2]	3	1
[1,4]	5	2
[1,8]	7	3
[1,16]	15	5
[1,31]	31	5
...		
[1,2047]	2047	11

(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. Comparisons for a vector of strings would not be the same as a comparison of a vector of strings. Generally, integers are faster to compare since integers are an array of bits. Strings are an array of characters/character arrays. There is more to compare when dealing with strings.

(h) (5 points) Use the Big-O asymptotic notation to classify binary search algorithm and justify your answer. Big-O notation of binary search is Olog(n). You can see with the graph in part d. The graph directly follows what a log(n) graph is like.

(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.

IMPORTANT NOTE: For each algorithm, I am counting each line counts as an operation. For example, int count = 0; is one operation and so is product = numbers.at(x) * numbers.at(y); even though there's both multiplication and assigning in this line. This is just to make it more consistent for me. I will be following this for the corresponding questions below.

```
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
```

Running time: f(n) = (n-1) + 1
The loops run until n - 1 which gives the n-1 within the function. There is only one operation following the for loop so no additionally additions to that part. The first assignment operation gives us the + 1 outside of the parenthesis. Big O notation: O(n), the run time function is linear. For any number of "n", there is a set number of times we run through the loop making it linear.

```
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
```

Running time: f(n) = ((n-1) / 2) + 1
The loops run until n - 1 which gives the n-1 within the function. Since we are increasing by increments of 2, we are essentially traversing through n at double the speed meaning we half the time, this is why there is a "/ 2" inside the parenthesis associated with the n - 1. The first assignment operation gives us

the + 1 outside of the parenthesis. Big O notation: O(n), the run time is still linear even though we increased the increment. For any number of "n", there is a set number of times we run through the loop making it linear.

```
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
```

Running time: f(n) = ((n*(n-1)) / 2) + 1
The loops run until n - 1 which gives the n-1 within the function. The next for loop, which is nested, iterates through "i" which begins at 0 until n - 1. That is the value "n" since (n - 1) + 1 for the 0 that isn't included makes it n. However, since j begins at 1, the second loop doesn't always run through what "n - 1" is. For example, in the first iteration, j = 1 and i = 0. The second loop's conditional statement doesn't even pass. The number of elements that the second loop goes through increases with more runs which gives us the / 2. The first assignment operation gives us the + 1 outside of the parenthesis. Big O notation: O(n^2)

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

Running time: f(n) = (2*(n-1)) + 2
The loops run until n - 1 which gives the n-1 within the function. The first two assignment operation gives us the + 1 outside of the parenthesis. Big O notation: O(n)

```
Algorithm Ex5 (A, B):
    Input: Arrays A and B storing n >= 1 integers.
    Output: The number of elements in B equal to the
        partial sums in A.
c <- 0 //counter
for i <- 0 to n - 1 do
    s <- 0 //partial sum
    for j <- 0 to n - 1 do
        s <- s + A[0]
        for k <- 1 to j do
            s <- s + A[k]
        end for
    end for
    if B[i] = s then
        c <- c + 1
    end if
end for
return c
```