CSCE 221 Cover Page

Manas Navale 333006797

msn0083-tamu msn0083@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)	https://flexiple.com/algorithms /big-o-notation-cheat-sheet	https://algorithmtutor.com/Analysis-of-Algorithm/ Running-Time-Growth-of-Function-and-Asymptotic-Notati
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.*"

Manas Navale 9/20/23

Homework 1

Check the Canvas calendar for the deadliness.
The homework submission to Gradescope only.

Typeset your solutions to the homework problems listed below using LaTeX (or LyX). See the class webpage for information about their installation and tutorials.

Homework 1 Objectives:

1. Developing the C++ programming skills by using
   (a) templated dynamic arrays and STL vectors
   (b) 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 LaTeX or LyX.
4. Understanding the definition of the big-O asymptotic notation.
5. Classifying algorithms based on pseudocode.

---

1. (25 points) Include your C++ code in the problem solution—**do not use attachments or screenshots.**

   (a) (10 points) Use the STL class `vector<int>` to write two C++ functions that return true if there exist **two** elements of the vector such that their **product** is divisible by 12, and return false otherwise. The efficiency of the first function should be $O(n)$ and the efficiency of second one should be $O(n^2)$ where n is the size of the vector.

```
   //first function
   bool LinearFunction(const vector<int>& nums) {
      unordered_set<int> seen;
      for (int num : nums) {
          if (num % 12 == 0 || seen.count(12 - (num % 12)) > 0) {
              return true;
          }
          seen.insert(num % 12);
      }
      return false;
  }

//second function
bool QuadFunction(const vector<int>& nums) {
    int n = nums.size();
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if ((nums[i] * nums[j]) % 12 == 0) {
                return true;
            }
        }
    }
    return false;
}
```

(b) (6 points) Justify your answer by writing the running time functions in terms of n for both the algorithms and their classification

The first function has a running time function of $T(n) = O(n)$ and a linear time complexity. The second function has a running time function of $T(n) = O(n^2)$ and a quadratic time complexity.

(c) (2 points) What do you consider as an operation for each algorithm?
An operation in the first algorithm would be the insertion and the checking:

```
unordered_set<int> seen;
for (int num : nums) {
    // Insertion operation (O(1) average time complexity)
    seen.insert(num % 12);
}
// Checking for an element operation (O(1) average time complexity)
if (num % 12 == 0 || seen.count(12 - (num % 12)) > 0) {
    return true;
}
```

in the second algorithm it would be the nested loops:

```
int n = nums.size();
for (int i = 0; i < n; i++) {
   // inner loop  iteratres over pairs of elements in the 'nums' vector.
    for (int j = i + 1; j < n; j++) {
        //Constant time operation within the inner loop
        if ((nums[i] * nums[j]) % 12 == 0) {
            return true;
        }
    }
}
```

(d) (2 points) Are the best and worst cases for both the algorithms the same in terms of big-O notation? Justify your answer.

In both algorithms, the time complexity does not change based on the input and remain $O(n)$ and $O(n^2)$. So the best and worst cases are the same.

(e) (5 points) 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.

The best case scenario for the algorithms would be if a pair divisible is found early so something like $[12, 6, 2, 8, 4, 9]$, while the worst case scenario would be $[1, 2, 3, 4, 5, 6]$ where there algorith examines each input without finding a pair divisible by 12.

2. (50 points + bonus) 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
   return -1; //not found
}
```

   Be sure that before calling `Binary_Search`, elements of the vector `v` are arranged in **ascending** order. The function should also keep track of the number of comparisons used to find the num `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 = 0;

template <typename T>
int Binary_Search(vector<T> &v, T x) {
    int mid, low = 0;
    int high = static_cast<int>(v.size()) - 1;

    while (low <= high) {
        mid = (low + high) / 2;
        num_comp++;

        if (v[mid] == x) {
            return mid;
        } else if (v[mid] < x) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}

int main() {
    vector<int> sorted_vector = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};

    for (int i = 1; i < sorted_vector.size(); i++) {
        if (sorted_vector[i] < sorted_vector[i - 1]) {
            cout << "Not sorted vector." << endl;
            return 1;
        }
    }

    int num1 = 1;
    int num2 = 16;
    int num3 = 8;

    int location1 = Binary_Search(sorted_vector, num1);
    int location2 = Binary_Search(sorted_vector, num2);
    int location3 = Binary_Search(sorted_vector, num3);

    cout << num1 << " found at index " << location1 << ", comparisons: " << num_comp << endl
    cout << num2 << " found at index " << location2 << ", comparisons: " << num_comp << endl
    cout << num3 << " found at index " << location3 << ", comparisons: " << num_comp << endl

    return 0;
}
```

(b) (10 points) Test your algorithm for correctness using a vector of data with 16 elements sorted in ascending order. An error message should be printed or exception should be thrown when the input vector is unsorted.
What is the value of `num_comp` in the cases when

i. the num `x` is the first element of the vector `v`

   num comp is 1

ii. the num `x` is the last element of the vector `v`

iii.

   num comp is 4

iv. the num `x` is in the middle of the vector `v`

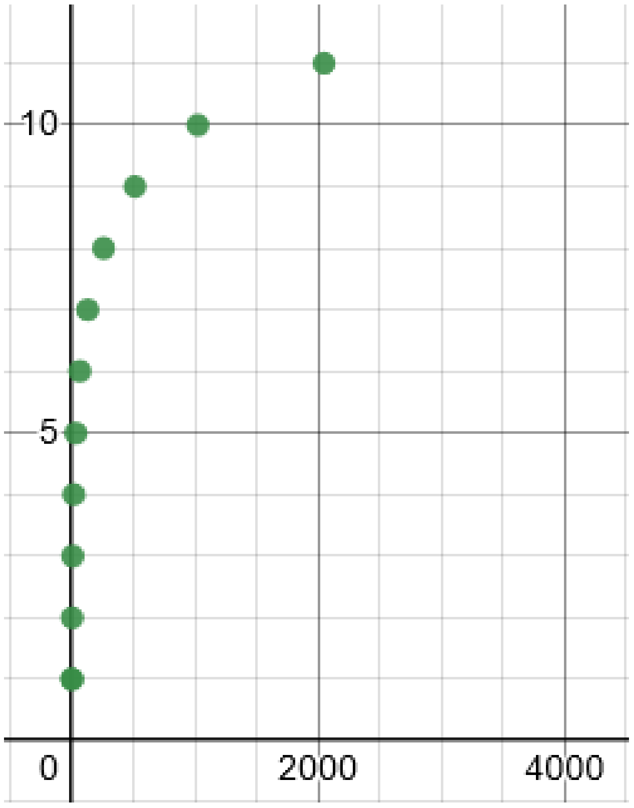
   num comp is 3

What is your conclusion from the testing for $n = 16$?

My conclusion is that the binary search algorithm performs a small number of comparisons becasue it is capable of narrowing down the search range effectively.

(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 num as the last element in the vector. Record the value of `num_comp` for each vector size in the table below.

Range [1,n]	num	num_comp
[1, 1]	1	1
[1, 2]	2	1
[1, 4]	4	2
[1, 8]	8	3
[1, 16]	16	4
[1, 32]	32	5
[1, 64]	64	6
[1, 128]	128	7
[1, 256]	256	8
[1, 512]	512	9
[1, 1024]	1024	10
[1, 2048]	2048	11



(d) (5 points) Plot the number of comparisons for the vector size $n = 2^k$, $k = 0, 1, 2, \ldots, 11$. 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.

The formula would be $C = \log_2(n)$. The following formula matches the computed output for each input size, except 1, where the number of comparisons was 1, and not 0.

(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 $n = 2^k - 1$, $k = 0, 1, 2, \ldots, 11$ and plot the number of comparisons vs. the size of the vector.

The modified formula would be $C = \lceil \log_2(n) \rceil$

Range [1,n]	num	num_comp
[1, 1]	1	0
[1, 3]	3	2
[1, 7]	7	3
[1, 15]	15	4
[1, 31]	31	5
[1, 63]	63	6
[1, 127]	127	7
[1, 255]	255	8
[1, 511]	511	9
[1, 1023]	1023	10
[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.

It would not be the same, as vector of strings, would depend on different variables like the number of characters and the lenght of the string, meaning the comparisons would vary based on input data. For a vector of doubles, it would vary on the specific double value.

(h) (5 points) Use the big-O asymptotic notation to classify binary search algorithm and justify your answer.

Time Complexity: $O(\log_2(n))$
The key component of this algorithm is the way it reduces the search space in half, and the way the number of comparisons is proportional to the size of the input data. As n grows, the number of comparisons grows(but at a slower rate.)

(i) (Bonus 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 descending 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. Note that array indices start from 0.

   (a) **Algorithm Ex1(A):**

```
  Input: An array A storing $n \geq 1$ integers.
  Output: The sum of the elements in A.
$s \leftarrow A[0]$
for $i \leftarrow 1$ to $n-1$ do
   $s \leftarrow s + A[i]$
end for
return $s$
```

   Running Time Function: $f(n) = n$
   Big O: $O(n)$

   (b) **Algorithm Ex2(A):**

```
  Input: An array A storing $n \geq 1$ integers.
  Output: The sum of the elements at even positions in A.
$s \leftarrow A[0]$
for $i \leftarrow 2$ to $n-1$ by increments of 2 do
  $s \leftarrow s + A[i]$
end for
return $s$
```

   Running Time Function: $f(n) = \frac{n}{2}$
   Big O: $O(n)$

(c) **Algorithm Ex3(A):**

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

Running Time Function:$f(n) = n + \frac{n(n-1)}{2}$
Big O: $O(n^2)$

(d) **Algorithm Ex4(A):**

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

Running Time Function: $f(n) = 2n$
Big O: $O(n)$

(e) **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
    for $i \leftarrow 0$ to $n-1$ do
        $s \leftarrow 0$ //partial sum
        for $j \leftarrow 0$ to $n-1$ do
            $s \leftarrow s + A[0]$
            for $k \leftarrow 1$ to $j$ do
                $s \leftarrow s + A[k]$
            end for
        end for
        if $B[i] = s$ then
            $c \leftarrow c + 1$
        end if
    end for
    return $c$
```

Running Time Function: $f(n) = n + \frac{n^2(n-1)}{2}$
Big O: $O(n^3)$