CSCE 221 Cover Page

Vishal Suresh 532004218

suresh06192004 suresh06192004@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://slaystudy.com/binary-search-real-life-examples/		
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.*"

Vishal Suresh 06/18/2023

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.**

   - (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.

     O(n^2):

```
bool On2(vector<int> v) {
    for (int i = 0;i < v.size();++i)
        for (int j = i+1;j < v.size();++j)
            if (v[i] * v[j] % 12 == 0)
                return true;
     return false;
 }
```

     O(n):

```
bool On1(vector<int> v) {
    std::unordered-set<int> nums;
    bool hasNonZero = false;
    bool has12Num = false;

    for (int num : v) {
        if (num%12 == 0)
            has12Num = true;
        else if (num != 0)
            hasNonZero = true;

        if (hasNonZero && has12Num)
```

```
            return true;

        if (num % 3 == 0) {
            if (nums.find(num / 3) != nums.end())
                return true;
            if (nums.find(num / 4) != nums.end())
                return true;
            }
             nums.insert(num)
}
```

- (6 points) Justify your answer by writing the running time functions in terms of n for both the algorithms and their classification in terms of big-O asymptotic notation.

  O(n):

  The runtime of the function is O(n) because there is one loop in the function that accounts for most of the assignments and operations.

  O(n^2):

  The runtime of the function is O(n^2) as there are two nested loops which take up most of the operations and assignments.

- (2 points) What do you consider as an operation for each algorithm?

  An operation for each algorithim would either be to change its value or create a new variable. this makes sense as the loop itself does not increase runtime but rather the operations done within the loop as these take some amount of time and resources.

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

  The best case case for both algorithms would be O(1) as if the first two elements have a product that would be divisible by 12. The worst case would be O(n) and O(n^2) for their respective algorithms as this would mean that the elements that would be divisible by 12 are at the end of the array.

- (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.

  O(n^2):

  Best Case: If the first and second elements in the array happen to be 3 and 4 then the if statement in the nested for loop would be true as (3 * 4 % 12) does equal 0 making the function return true;

  Worst Case: If the entire array was filled with 0's or some other numbers that do not happen to multiply out to a multiple of 12, and the last two element's product happens to be divisible by 12, then the array would have not iterate through n^2 times before finding those last two elements. For example vector<int> v 0,0,1,5,1,4,2,6.

  O(n):

  Best Case: If the first two elements happened to be 12 and any other number the loop would only have to run twice before returning true as 12 times any number is a multiple of 12.

  Worst Case: If the last element was a multiple of 3 and and did not appear in the end of an array, then the loop would have to go through the entire array to 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
   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 target `x`. The (global) variable `num_comp` keeps the number of comparisons and initially should be set to zero.

   (b) (6 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 target `x` is the first element of the vector `v`
      5

   ii. the target `x` is the last element of the vector `v`
      5

   iii. the target `x` is in the middle of the vector `v`
      5

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

   The program will always make 5 comparisons before getting a result

   (c) (6 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.

Range [1,n]	Target	num_comp
[1, 1]	1	1
[1, 2]	2	2
[1, 4]	4	3
[1, 8]	8	4
[1, 16]	16	5
[1, 32]	32	6
[1, 64]	64	7
[1, 128]	128	8
[1, 256]	256	9
[1, 512]	512	10
[1, 1024]	1024	11
[1, 2048]	2048	12

   (d) (4 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.

num_comp= $log_2(n) - 1$. Since we know that the number of elements is equal to 2 to the power of k where k is the number of comparisons minus 1. We can just take the log of both sides and solve for the number of comparisons.

(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.

Range [1,n]	Target	num_comp
[1, 1]	1	1
[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.

I think when comparing a string more comparison will occur because they are essentially an array of character so when comparing an entire string, each of the individual character will have to be compared leading to more comparisons. However I think when comparing a doubles the amount on comparisons will stay the same as a double is its own type and not made up of smaller types like a string.

(h) (4 points) Use the big-O asymptotic notation to classify binary search algorithm and justify your answer.

The runtime of a binary search algorithm is O(log$n$). This is because every time a comparison is made, the number of elements that need to be search is divided by two. Therefore with every iteration of the loop, the size exponentially gets smaller.

(i) (10 points bonus) 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.

(j) (10 points) Write about two real life applications of this algorithm. Justify your answer.

The first application of a binary search is for a library. A library contains thousands of books so it would be inefficient to search for it using a linear fashion. Library books are generally sorted alphabetically or using some integer code. Using that allows books to be sorted effectively. The second application is a online dictionary. These dictionaries typically involve thousands of words, and since these words can be stored alphabetically, they can also use the binary search algorithm. Both of these examples can use the order of the alphabet to check which letter is "greater" than the other in a similar way you could compare the size of two numbers.

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.

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

The total operations: f(n) = 1 + (n-1)*2 → Big O is O(n)

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

The total operations: f(n) = 1 + (n/2)*2 → Big O is O(n)

Algorithm Ex3(A):
  Input: An array A storing $n \geq 1$ integers.
  Output: The sum of the partial sums in A.
$s \leftarrow 0$
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
The total operations: f(n) = 1 + 2n + n*(n/2) → Big O is $O(n^2)$

Algorithm Ex4(A):
Input: An array A storing $n \geq 1$ integers.
Output: The sum of the partial sums in A.

$t \leftarrow A[0]$
$s \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  $s \leftarrow s + A[i]$
  $t \leftarrow t + s$
end for
return t

The total operations: f(n) = 2 + 4(n-1) → Big O is $O(n)$

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
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

The total operations: f(n) = 1 + n * n(1 + 2(n-1)) → Big O is $O(n^3)$