CSCE 221 Cover Page

Homework #1

Due September 14 at midnight to eCampus

First Name Rong Last Name Xu UIN 928009312

User Name Abby-xu E-mail address rongx0915@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 Rong Xu Date 2020/09/13

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.

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.sfvsf
     - An operation such as a if statement, add the value at a varaible... For a for loop of n elements, the operations of this for loop is n if there isn't any more nested for loop. If there is a nested for loop in a for loop, (let the first for loop includes n elements and the second for loop includes m elements, then we have m * n operations for this system.)
     - Solution 1: code for $O(n)$.....

```
bool if_Product_odd (vector<int> &v) {
        \\ The product of two elements is odd if and only if both two elements are odd
        \\ For example, 3 * 5 = 15 is odd; 5 * 6 = 30 is even; 6 * 8 = 48 is even
        bool if_odd = false;
        int count = 0, operation = 0; \\ Set a integer named operation for testing
        for (int i = 0; i < v.size(); i++) {
                if (v.at(i) % 2 == 1) count++;
                operation++;
        }
        if (count >= 2) if_odd = true;
        cout << "The operation of this algorithm is " << operation << endl;
        return if_odd;
}
```

       * The best case of this algorithm is $O(n)$, and as same as the worst case and average case.
       * Test cases:
         · Input vector_1 = [1, 3] for best case, the operation is 2;
         · Input vector_2 = [2, 4, 6, 7, 9] for worst case, the operation is 5;
         · As two test cases above, both of the best case and worst case have the same in terms of Big-O notation.
     - Solution 2: code for $O(n^2)$.....

```
\\ Version_1
bool if_Product_odd (vector<int> &v) {
        \\The product of two elements is odd
        \\if and only if both two elements are odd
        bool if_odd = false;
        int operation = 0; \\ Set a integer named operation for testing
        for (int i = 0; i < v.size(); i++) {
                for (int j = 0; j < v.size() - 1; j++) {
                        if (v.at(i) % 2 == 1 && v.at(j) % 2 == 1) {
                                if_odd = true;
                        }
                        operation++;
                }
        }
        cout << "The operation of this algorithm is " << operation << endl;
        return if_odd;
}
```

```
\\ Version_2
bool if_Product_odd (vector<int> &v) {
        \\The product of two elements is odd
        \\if and only if both two elements are odd
        int operation = 0; \\ Set a integer named operation for testing
        for (int i = 0; i < v.size(); i++) {
                for (int j = 0; j < v.size(); j++) {
                        operation++;
                        if (i != j && v.at(i) % 2 == 1 && v.at(j) % 2 == 1) {
                                cout << "The operation of this algorithm is "
                                        << operation << endl;
                                return true;
                        }
                }
        }
        return false;
}
```

* The best case of this algorithm is $O(1)$; the worst case of this algorithm is $O(n^2)$.
  - Input vector_1 = [1, 3] for best case, the operation is 2 for version_1, 1 for version_2;
  - Input vector_2 = [2, 4, 6, 7, 9] for worst case, the operation is 20 for both version_1 and version_2;
  - Big-O for best case and worst case depends on different version of code.

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 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>
#include<math.h>
using namespace std;
int num_comp = 0;
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 "sth␣wrong..."; // Unsuccessful_Search(); //exception: not found
}

int main() {
        vector<int> v;
        int k;
        cin >> k;
        int n = pow(2, k) - 1;
        cout << "The␣number␣of␣vector:␣" << n;
        for (int i = 0; i < n; i++)
                v.push_back(i);
        cout << "Last␣element:␣" << v.at(n - 1) << endl;
        cout << "position:␣" << Binary_Search(v, n - 1);
        cout << "\n" << "num_comp:␣" << num_comp;
        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 the vector `v`
      A. 5

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

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

A. 5

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

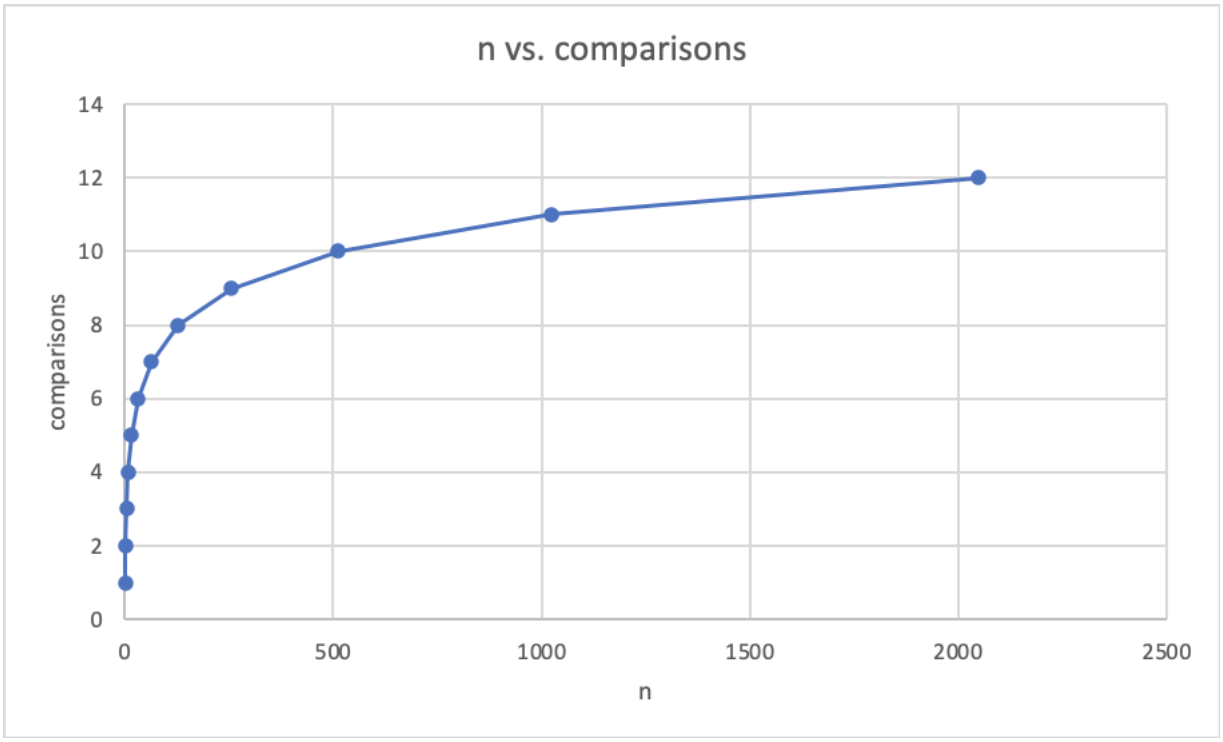
i. In conclusion, whatever index is, the number of comparison will always be 5

(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.

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 $n = 2^k$, $k = 1, 2, \ldots, 11$. You can use a spreadsheet or any graphical package. Include graphs for each case.

i.



(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.

i. $f(n) = log_2 n + 1$

ii. Yes, the fomula matches the computed output for any input. For example, when n = 4, this function will return 3, which matches the fomula I got.

(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 $2^k - 1$, $k = 1, 2, 3, \ldots, 11$.

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	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. Yes, I think so. The running time of the algorithm itself does not depend on the type of inputs.

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

i. The Big-O of binary search is O(log n). In this case, the base of log is 2.

(a) (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.

i.
```
#include <iostream>
#include <vector>
#include<math.h>
using namespace std;
class Binary_Search {
private:
        int num_comp = 0;
        int num_find;
public:
        std::vector<int> vec;
        // constructor
        Binary_Search(const std::vector<int> &v, int x) : num_find(x) {
                for (int i = 0; i < v.size(); i++)
                        vec.push_back(v[i]);
        }
        int Bi_Search() {
                int mid;
                int low = 0;
                int high = (int) vec.size()-1;
                while (low < high) {
                        mid = (low + high) / 2;
                        cout << "mid:␣" << mid << endl;
                        if (num_comp++, vec[mid] < get_num_find()) high = mid - 1;
                        else if (vec[mid] == get_num_find()) high = mid;
                        else low = mid + 1;                        }
                        if (num_comp++, num_find == vec[high]) return high; //OK: found
                        throw "sth␣wrong..."; // Unsuccessful_Search(); //exception: not
                }
        int get_num_find() const {
                return num_find;
        }
        int get_num_comp() const {
                return num_comp;
```

```
        }
        void print_out() {
                for (int i = 0; i < vec.size(); i++)
                        cout << vec[i] << endl;
                cout << "finished printing" << endl;
        }
};

int main() {
        vector<int> v;
        int k;
        cin >> k;
        int n = pow(2, k);
        cout << "The number of vector: " << n << endl;
        for (int i = n; i >= 1; i--)
                v.push_back(i);
        Binary_Search bs(v, n);
        cout << "finished constructor" << endl;
        bs.print_out();
        bs.Bi_Search();
        cout << "finished function" << endl;
        cout << bs.get_num_comp() << endl;
        return 0;
}
```

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 ≥ 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 Big-O is O(n)
// Running time function is 2(n - 1) + 1 //
```

```
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 Big-O is O(n / 2) = O(n)
// Running time function is 2(n / 2) + 2
```

```
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 Big-O is O(2n(2n + 1) / 2 + 2) = O(n ^ 2)
// Running time function is 2n(2n + 1) / 2 + 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
// The Big-O is O(n)
// Running time function is 4(n - 1) + 3
```

```
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
// The Big-O is O(n^3)
// Running time function is n * 2n * 2 (2n + 1) / 2 + 3
if statement: 3
first for loop: n
second for loop: 2n
third for loop: (2n + 1) / 2
```