CSCE 221 Cover Page
Homework #1
Due July 12 at midnight to CSNet

First Name Carson Last Name Hanel UIN 826003149

User Name Chingy1510 E-mail address chingy1510@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	
Web pages (provide URL)	https://www.wolframalpha.com/input/?i=graph+(8n+*+log_2(n))++and+2*(n%5E2) Wolfram graph for 8n*logn and 2n^2
Web pages (provide URL)	https://www.wolframalpha.com/input/?i=graph+(n*log_2(n))++and+(n%5E2)+from+0+t Wolfram graph for nlogn vs. n^2
Printed material	Data Structures and Algorithms Ed. 2 by Goodrich, Tamassia, and Mount
Other Sources	Lecture Notes

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 Carson Hanel Date 7/12/2017

Type the solutions to the homework problems listed below using preferably LyX/LaTeX word processors, see the class webpage for more information about their installation and tutorial.

1. (10 points) Write one C++ function for the Binary Search algorithm based on the pseudocode in the textbook on page 396. to search a target element in a sorted, ascending or descending, order vector. Your function should also keep track of the number of comparisons used to find the target.

```
int binary_search(const vector<int>& vec, const int val, int low, int high, bool asc){
    int mid, temp;

    if(asc){ //Only runs if the function check_asc returns TRUE, or rather ascending.
        if(/*count++,*/ low > high){ //I chose not to count non-key comparisons. I can change thi
            cout << "It took " << count << " steps to NOT find " << val << endl;
            count = 0;
            throw runtime_error("The sought value doesn't exist.");
        }
        else{
            mid = (low + high) / 2;
            temp = vec.at(mid);
            if(count++, temp == val){
                cout << "It took " << count << " steps to find " << temp << endl;
                cout << "It should've taken " << abs(2 * floor(log2(val)) - 1)<< " iterations." <
                count =   0;
                return temp;
            }
            else if(count++, temp < val){
                return binary_search(vec, val, ++mid, high, asc);
            }
            else{
                return binary_search(vec, val, low, --mid, asc);
            }
        }
    }
    else{ //Only runs if the function check_asc returns FALSE, or rather descending.
        if(/*count++,*/ low > high){ //I chose not to count non-key comparisons. I can change thi
            cout << "It took " << count << " steps to NOT find " << val << endl;
            count = 0;
            throw runtime_error("The sought value doesn't exist.");
        }
        else{
            mid = (low + high) / 2;
            temp = vec.at(mid);
            if(count++, temp == val){
                cout << "It took " << count << " steps to find " << temp << endl;
                cout << "It should've taken " << abs(2 * floor(log2(val)) - 1)<< " iterations." <
                count =   0;
                return temp;
            }
            else if(count++, temp < val){
                return binary_search(vec, val, low, --mid, asc);
            }
            else{
                return binary_search(vec, val, ++mid, high, asc);
            }

        }
    }
}
```

2.

   (a) (5 points) To ensure the correctness of the algorithm the input data should be sorted in ascending or descending order. An exception should be thrown when an input vector is unsorted.

i. I handled this portion by checking the vector with a linear comparison algorithm to make sure it was either all ascending or descending relationships, and returned true for ascension, and false for descension.

ii.

```
bool check_sorted(const vector<int>& vec){
    bool asc = (vec.at(0) < vec.at(1)); //true if ascending
    if(asc){
        for(int i = 0; i < vec.size() - 1; i++){
            if(!(vec.at(i + 1) > vec.at(i))){
                throw runtime_error("Vector is unsorted!"); //Auto-throws exception that
            }
        }
        return true; //Returns true, the vector is sorted ascending
    }else{
        for(int i = 0; i < vec.size() - 1; i++){
            if(!(vec.at(i + 1) < vec.at(i))){
                throw runtime_error("Vector is unsorted!"); //Auto-throws exception that
            }
        }
        return false; //Returns false, the vector is sorted descending
    }
}
```

(b) (10 points) Test your program using vectors populated with:

i. consecutive increasing integers in the ranges from 1 to powers of 2, that is, to these numbers: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048.
Select the target as the last integer in the vector.

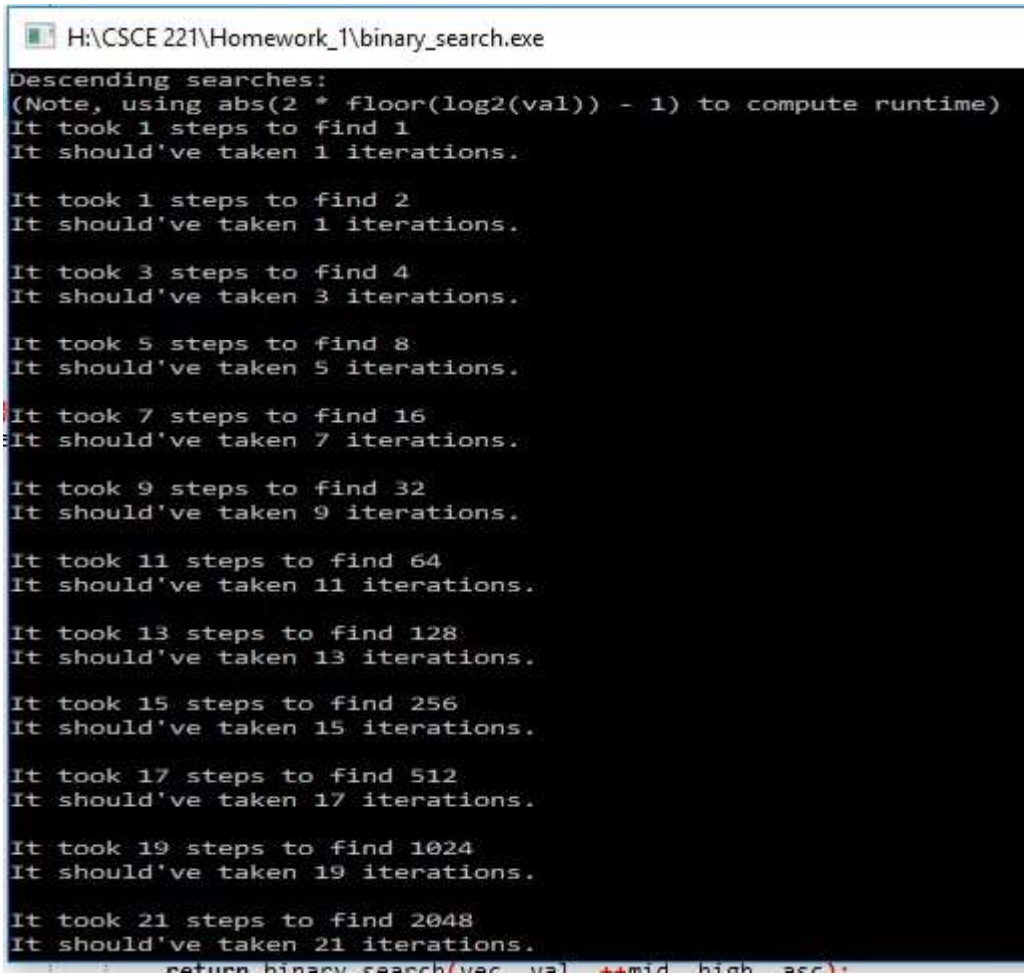
A.

```
H:\CSCE 221\Homework_1\binary_search.exe
Ascending searches:
(Note, using abs(2 * floor(log2(val)) - 1) to compute runtime)
It took 1 steps to find 1
It should've taken 1 iterations.

It took 1 steps to find 2
It should've taken 1 iterations.

It took 3 steps to find 4
It should've taken 3 iterations.

It took 5 steps to find 8
It should've taken 5 iterations.

It took 7 steps to find 16
It should've taken 7 iterations.

It took 9 steps to find 32
It should've taken 9 iterations.

It took 11 steps to find 64
It should've taken 11 iterations.

It took 13 steps to find 128
It should've taken 13 iterations.

It took 15 steps to find 256
It should've taken 15 iterations.

It took 17 steps to find 512
It should've taken 17 iterations.

It took 19 steps to find 1024
It should've taken 19 iterations.

It took 21 steps to find 2048
It should've taken 21 iterations.
```

ii. consecutive decreasing integers in the ranges from powers of 2 to 1 , that is, to these numbers: 2048, 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1.

Select the target as the last integer in the vector.

A.

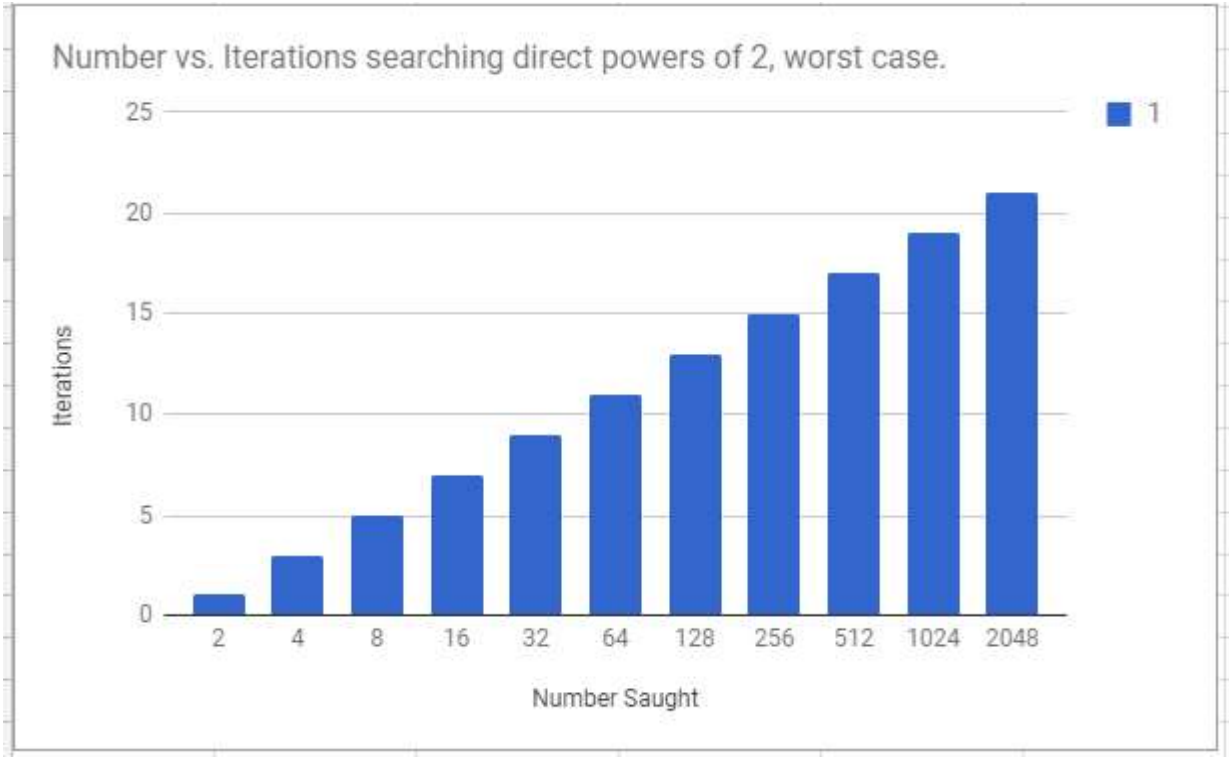


(c) (5 points) Tabulate the number of comparisons to find the target in each range.

Range [1,n]	Target for incr. values	# comp. for incr. values	Target for decr. values	# comp. for decr. values	Result of the formula in item 5 (e)
[1,1]	1	1	1	1	abs(2*floor(log(1))- 1) = 1
[1,2]	2	1	2	1	abs(2*floor(log(2)) - 1) = 1
[1,4]	4	3	4	3	abs(2*floor(log(4)) - 1) = 3
[1,8]	8	5	8	5	abs(2*floor(log(8)) - 1) = 5
[1,16]	16	7	16	7	abs(2*floor(log(16)) -1) = 7
...					
[1,2048]	2048	21	2048	21	abs(2*floor(log(2048)) - 1) = 21

(d) (5 points) Plot the number of comparisons to find a target where the vector size $n = 2^k$, $k = 1, 2, \ldots, 11$ in each increasing/decreasing case. You can use any graphical package (including a spreadsheet). Include graphs for each case.

i.



ii. Technically, the above graph holds true for the non-direct powers of 2 as well, but shift the number of iterations left one "number saught".

(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 a given input? Justify your answer.

i. Best case: the target is at (low + high) / 2 (mid). In this case, it takes 1 comparison.

ii. Worst case: the target is at either the right-most or left-most slot of the array. In this case, it takes 3*log(n) + 2 because for every iteration, there are at most 3 comparisons. log(1) is 0, and the best case takes 2 comparisons, so +2.

iii. Average case: the average case is (1 + 2*log(n) + 2) / 2 which is log(n) + 1.5 in the average case.

iv. This search is in O(log(n)).

v. So, to find 2048 in the largest array, it took 21 comparisons (not including low > high). The case is 2*log(n) - 1

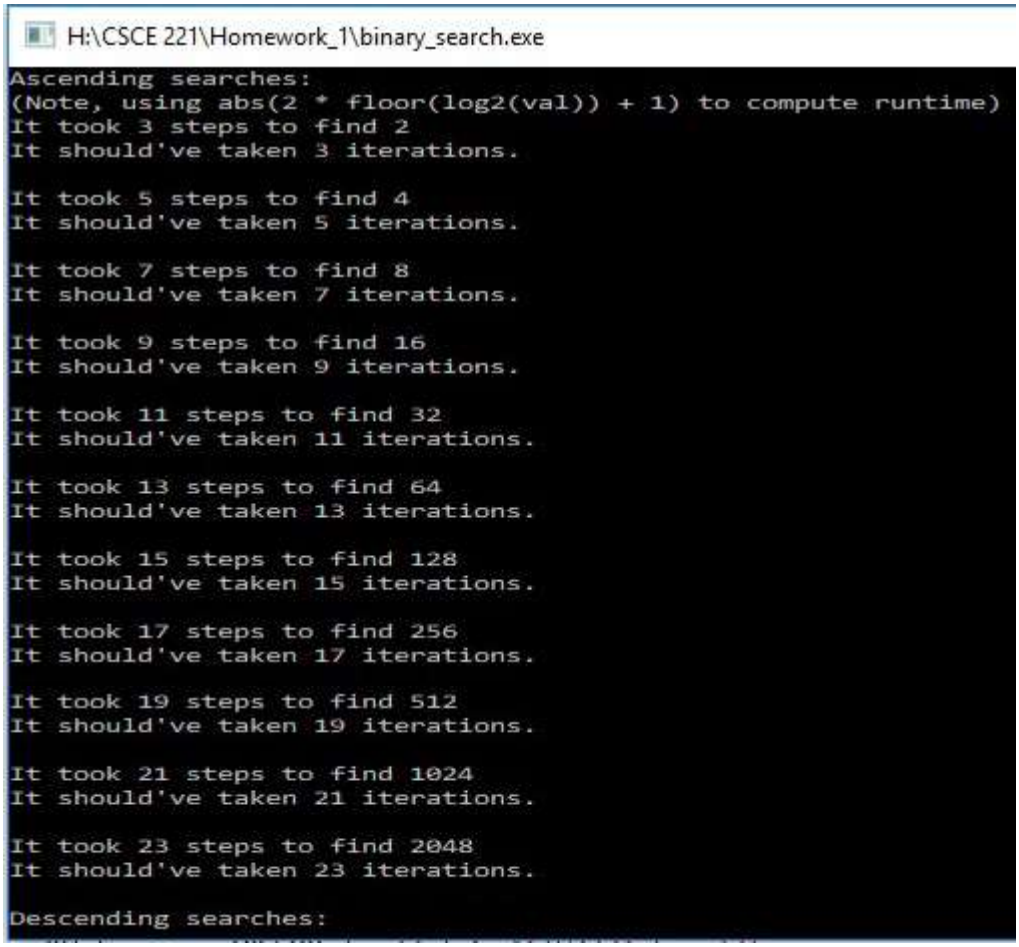
vi. So, the BEST 1:1 matching case I have found is abs(2 * floor(log(n)) - 1) with the abs being for the n = 1 place.

vii. NOTE: This runtime formula only works if the integer exists within the range.

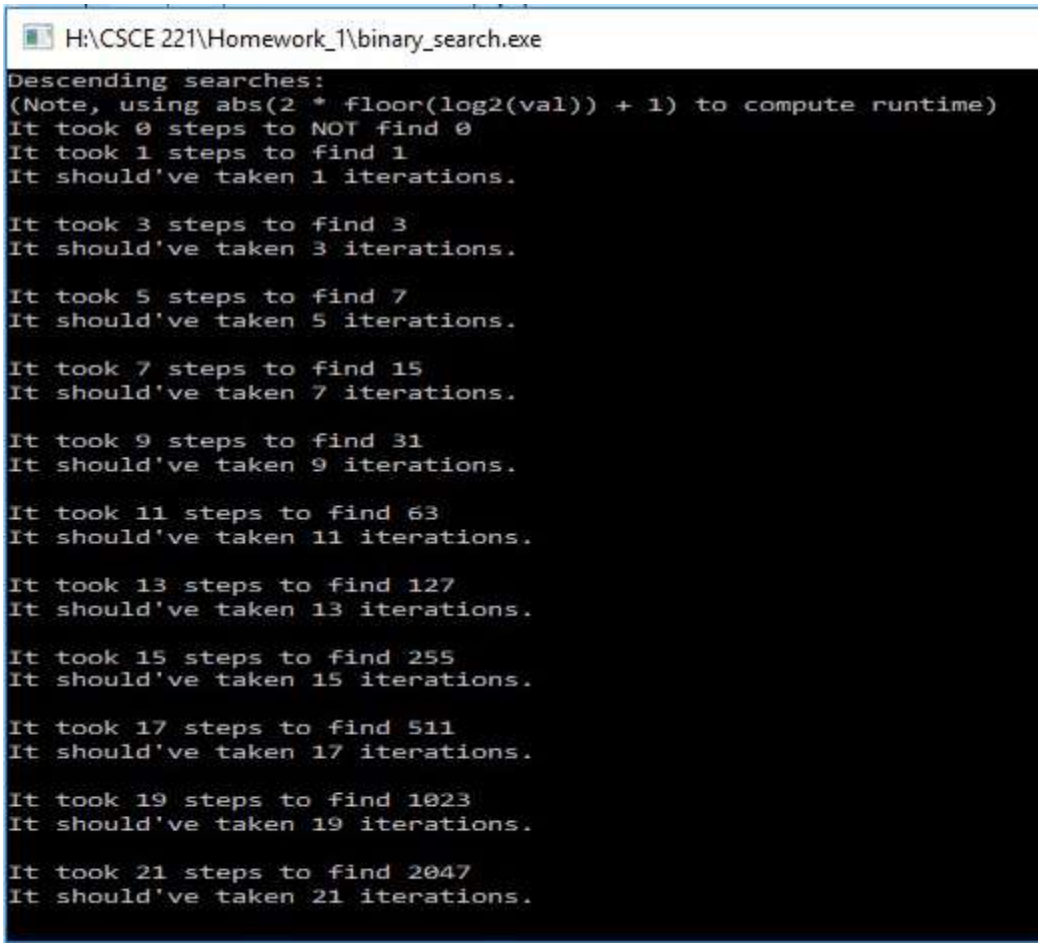
viii. NOTE 2: If the value DOESN'T exist, this is a perfect floor(2*log(n)) for n > 1.

(f) (5 points) How can you modify your formula/function if the largest number in a vector is not an exact power of two? Test your program using input in ranges from 1 to $2^k - 1$, $k = 1, 2, 3, \ldots, 11$.

i.


H:\CSCE 221\Homework_1\binary_search.exe
Ascending searches:
(Note, using abs(2 * floor(log2(val)) + 1) to compute runtime)
It took 3 steps to find 2
It should've taken 3 iterations.
It took 5 steps to find 4
It should've taken 5 iterations.
It took 7 steps to find 8
It should've taken 7 iterations.
It took 9 steps to find 16
It should've taken 9 iterations.
It took 11 steps to find 32
It should've taken 11 iterations.
It took 13 steps to find 64
It should've taken 13 iterations.
It took 15 steps to find 128
It should've taken 15 iterations.
It took 17 steps to find 256
It should've taken 17 iterations.
It took 19 steps to find 512
It should've taken 19 iterations.
It took 21 steps to find 1024
It should've taken 21 iterations.
It took 23 steps to find 2048
It should've taken 23 iterations.
Descending searches:

ii.



Range [1,n]	Target for incr. values	# comp. for incr. values	Target for decr. values	# comp. for decr. values	Result of the formula in item 5 (e)
[1,1]	1	1	1	1	abs(2*floor(log(1))- 1) = 1
[1,3]	3	3	1	3	abs(2*floor(log(3)) - 1) = 1
[1,7]	7	5	1	5	abs(2*floor(log(7)) - 1) = 3
[1,15]	15	7	1	7	abs(2*floor(log(15)) - 1) = 5
[1,31]	31	9	1	9	abs(2*floor(log(31)) - 1) = 7
...					
[1,2047]	2047	21	1	21	abs(2*floor(log(2047)) - 1) = 21

(g) (5 points) Use Big-O asymptotic notation to classify this algorithm and justify your answer.

i. This algorithm is in O(log(n))

A. 0 < |f(n)| <= c*|g(n)|

B. f(n) = abs(2*floor(log(31)) + 1) NOTE: I figured this out for the non-powers of 2.

C. g(n) = log(n)

D. c = 3

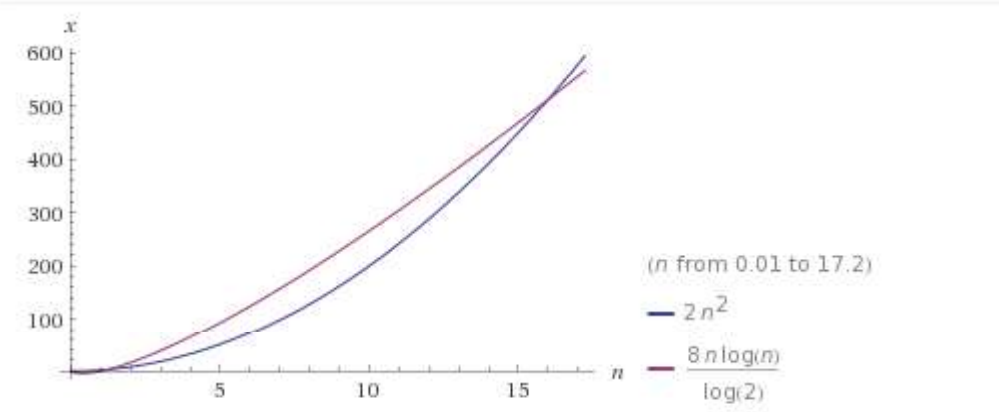
E. since (3*log(n) > 2*log(n) + 1) is true in all cases n > 1, the proof holds true.

F. Thus, the recursive binary search is O(log(n))

(h) Submit to CSNet an electronic copy of your code, testing results of all your experiments, and answer to the questions above for grading.

3. (10 points) **(R-4.7 p. 185)** The number of operations executed by algorithms A and B is $8n\log n$ and $2n^2$, respectively. Determine n_0 such that A is better than B for $n \geq n_0$.

   (a) This is 8nlogn and 2n^2 graphed together:



   (b)

   (c) Looking at the graph I made with wolframalpha, you can see that there exists some point less than 17.2 and greater than 16 in which the n^2 algorithm performs more poorly.

   (d) Thus, 16 <=n_0 <= 17.2

   (e) As a mathematical approach:

      i. 8n*log(n) = 2(n)^2 //Divide by 2
      ii. 4n*log(n) = n^2 //Divide by n
      iii. 4*log(n) = n //Divide n by log(n)
      iv. 4 = n/log(n)
      v. 16 / log(16) = 16/4 = 4 //I chose 16 because of the graph placement, but also knowing that 2^4 = 16 and 16/4 is indeed 4 because 16 = 4^2.
      vi. Thus, n_0= 16 is the point at which the two functions overlap, never meeting again (in the positive range).

4. (10 points) **(R-4.21 p. 186)** Bill has an algorithm, `find2D`, to find an element x in an $n \times n$ array A. The algorithm `find2D` iterates over the rows of A, and calls the algorithm `arrayFind`, of code fragment 4.5, on each row, until x is found or it has searched all rows of A. What is the worst-case running time of `find2D` in terms of n? What is the worst-case running time of `find2D` in terms of N, where N is the total size of A? Would it be correct to say that `find2D` is a linear-time algorithm? Why or why not?

   (a) At first I was confused, and then I found page 184.

   (b) Basically what this code fragment does is iterates linearily until the value is found.

   (c) In terms of n (thinking of each new row as a new iteration of an outside for-loop) given that place values 0, 1, ... n exist on one row of A:

      i. The worst case is technically O(n^2) when in terms of n.

   (d) In terms of N (rather, thinking of the entire 2D array as on linear array of size n^2) given that place values 0, 1, ... N exist in the entirety of A:

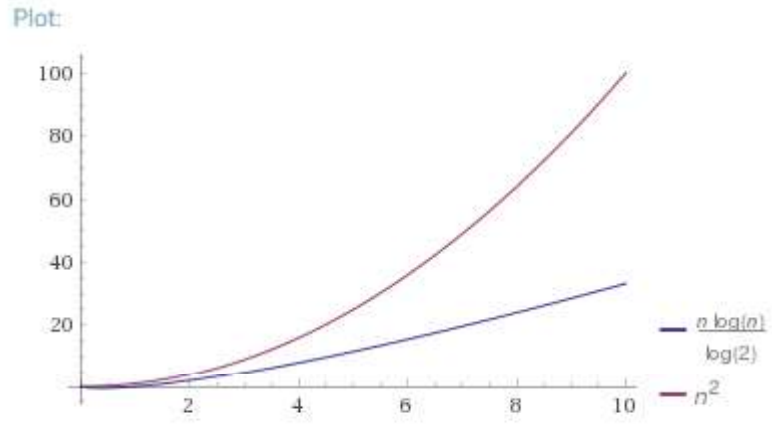
      i. The worst case is technically O(n) when in terms of N.

   (e) If you don't think in terms of N or n, find2D is absolutely an O(n) algorithm.

      i. This is because, when doing upper bounds, you assume the input is uniform. At its core, find2D iterates through and makes n-1 comparisons.

5. (10 points) **(R-4.39 p. 188)** Al and Bob are arguing about their algorithms. Al claims his $O(n\log n)$-time method is always faster than Bob's $O(n^2)$-time method. To settle the issue, they perform a set of experiments. To Al's dismay, they find that if $n < 100$, the $O(n^2)$-time algorithm runs faster, and only when $n \geq 100$ then the $O(n\log n)$-time one is better. Explain how this is possible.

   (a) Just like problem 3, here are the two graphed on wolframalpha, but rather than having the range as where they intersect, the range is 10:



   (b)

   (c) As you can observe, n*log(n) is much more efficient as a representative function.

   (d) The way that the O(nlogn) could be slower for indices n < 100 could be for if either Bob has a much lower coefficient to his n^2, or if Al has a very high coefficient to his algorithm.

   (e) A way to represent this is the comparison of, say 10(n*log(n)) + 2 vs. 1/10(n^2) + 1/10 or something.

   (f) Another explanation could have to do with the lower bound functions of both algorithms. Big-O notation is very broad.

   (g) Personal definition for Big O - The set of all categorical functions with respect to maximal runtime modified by a constant.

6. (20 points) Find the running time functions for the algorithms below and write their classification using Big-O asymptotic notation. The running time function should provide a formula on the number of operations performed on the variable s.

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**
  $s \leftarrow s + A[i]$
return s
The above runtime complexity: f(n) = 1 + (n - 1) + 1 = O(n)
Note: The reason this is n-1 is because the range starting at 1.
Note: The 1 at the beginning is for assignment on s, and the final +1 is the return

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**
  $s \leftarrow s + A[i]$
return s
The above runtime complexity: f(n) = 1 + ((n - 1) / 2) + 1 = O(n) because n/2 runti
Note: This is because indices 0 and 1 are overlooked by the algorithm.
Note: The 1 at the beginning is for assignment on s, and the final +1 is the return

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**
    $s \leftarrow s + A[j]$
return s
The above runtime complexity: f(n) = 1 + (n(n - 1))/2 + 1 = O(n^2)
Note: The -1 is because j is from 1 to n-1 rather than 0 to n-1.
Note: The 1 at the beginning is for assignment on s, and the final +1 is the return

Algorithm Ex4(A):
  Input: An array A storing $n \geq 1$ integers.
  Output: The sum of the partial sums in A.
$t \leftarrow 0$
$s \leftarrow 0$
for $i \leftarrow 1$ **to** $n-1$ **do**
  $s \leftarrow s + A[i]$
  $t \leftarrow t + s$
return t
The above runtime complexity: f(n) = 1 + (n - 1) = O(n)
Note: Array starts at index 1 rather than 0 to n-1.
Note: Because t = t + s is an assignment on the variable t rather than s, it wasn't
Note: Since s is not returned, there's no +1 for return, but still +1 for assignmen