CSCE 221 Cover Page
Homework #1
Due Sept. 23 by midnight to CSNet

Oneal Abdulrhaim UIN 324007937

User Name oneal.abdulrahim E-mail address oneal.abdulrahim@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	TA's in lab Dr. Leyk		
Web pages (provide URL)	C++ Binary Search		
Printed material	Our Textbook		
Other Sources	Lecture Slides		

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 Oneal Abdulrahim Date 23 September 2016

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 a C++ program to implement the Binary Search algorithm for searching a target element in a sorted vector. Your program should keep track of the number of comparisons used to find the target.

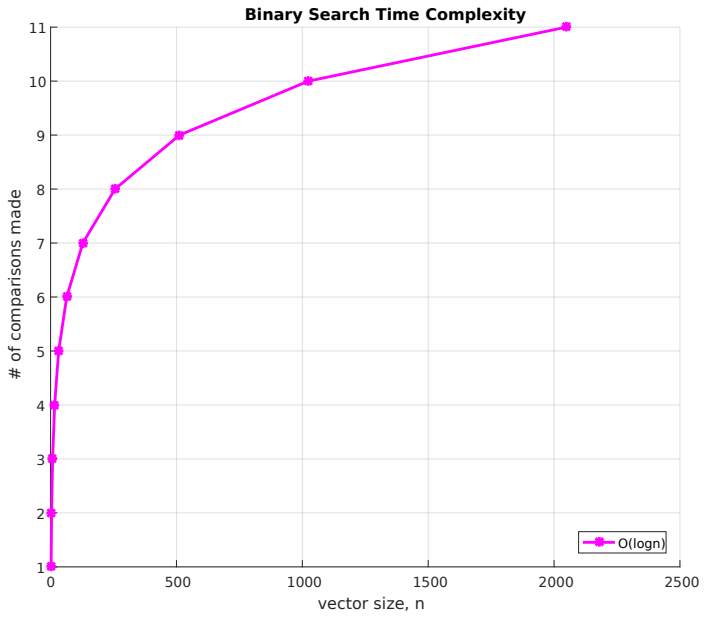
   (a) (5 points) To ensure the correctness of the algorithm the input data should be sorted in ascending or descending order. An exception should be thrown when an input vector is unsorted.

   (b) (10 points) Test your program using vectors populated with consecutive (increasing or decreasing) 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.

   (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
[1,1]	1	0	1	0	0
[1,2]	2	1	1	1	1
[1,4]	4	2	1	2	2
[1,8]	8	3	1	3	3
[1,16]	16	4	1	4	4
[1,32]	32	5	1	5	5
[1,64]	64	6	1	6	6
[1,128]	128	7	1	7	7
[1,256]	256	8	1	8	8
[1,512]	512	9	1	9	9
[1,1024]	1024	10	1	10	10
[1,2048]	2048	11	1	11	11

   (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).



(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.

The mathematical formula for the number of comparisons k for vector size n is $k = log_2(n)$. For each iteration of binary search, the number of items left to compare halves. It is like searching for a word in the dictionary, you open it to the middle, then see if the first letter is before or after the letter of the page you are on. Also, our binary search tree is perfectly balanced, so our algorithm expects the next node to be the midpoint of exactly half of the remaining elements.

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

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
[1,1]	1	0	1	0	0
[1,3]	2	1	1	1	1
[1,7]	4	2	1	2	2
[1,15]	8	3	1	3	3
[1,31]	16	4	1	4	4
[1,63]	32	5	1	5	5
[1,127]	64	6	1	6	6
[1,255]	128	7	1	7	7
[1,511]	256	8	1	8	8
[1,1023]	512	9	1	9	9
[1,2047]	1024	10	1	10	10

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

$O(\log n)$. In my particular algorithm's case, it is $k = log_2 n$. The algorithm is recursive, so it calls binary search again with adjusted bounds. For each iteration, there is a maximum of two comparisons that occur. The worst case for binary search is actually good for number of comparisons, because in recursive implementation, only one comparison is made until the final index is reached.

(h) Submit to CSNet an electronic copy of your code and results of all your experiments for grading.

2. (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 $n \geq n_0$.

   At $n \geq 16$, algorithm A is better than algorithm B.

3. (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?

   For each row iteration, there are n comparisons. But, the array has number of rows n, so we are actually performing n comparisons n times. The worst-case running scenario would be searching for an element that is located in the n-th row and the$n$-th column. Therefore, the running time in terms of n is $O(n^2)$. In terms of N, the worst-case scenario would be $O(N)$. However, it would be incorrect to say that this method is of linear time complexity, simply because the calculation of Big-O is based on input size n and number of comparisons, and $N = n^2$.

4. (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.

   For smaller input sizes (namely $n < 100$), the algorithm with $O(n^2)$ time complexity outperforms $O(n \log n)$. However, since $n^2$is strictly increasing for all values of $n \geq 0$ and $n \log n$ increases slower. Eventually, n^2 grows too big, leaving $n \log n$ at a more efficient time complexity.

5. (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`.

   (a) `Ex1:`
   $k = 1 + (n - 2)$
   $O(n)$

   (b) `Ex2:`
   $k = 1 + (\log_2(n))$ `I interpreted the forloop to change i by *= 2 on each iteration`
   $O(\log n)$

   (c) `Ex3:`
   $k = 1 + (2n - 1)$ `Since the j forloop goes from j to i`
   $O(n)$

   (d) `Ex4:`
   $k = 1 + (n - 1)$
   $O(n)$