CSCE 221 Cover Page

Homework #1

Due February 13 at midnight to eCampus

First Name: Leilani Last Name: Horlander-Cruz UIN: 523008771

User Name: leilanihc112 E-mail address: leilanihc112@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. According to the University Regulations, Section 42, scholastic dishonesty are including: acquiring answers from any unauthorized source, working with another person when not specifically permitted, observing the work of other students during any exam, providing answers when not specifically authorized to do so, informing any person of the contents of an exam prior to the exam, and failing to credit sources used. Disciplinary actions range from grade penalties to expulsion read more: Aggie Honor System Office

Type of sources	Website		
People			
Web pages (provide URL)			
Printed material			
Other Sources	Powerpoints from class		

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	Leilani Horlander-Cruz			Date	2-13-17

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**	**1**	**1**	**1**	**1**
[1,2]	**2**	**2**	**1**	**2**	**2**
[1,4]	**4**	**3**	**1**	**3**	**3**
[1,8]	**8**	**4**	**1**	**4**	**4**
[1,16]	**16**	**5**	**1**	**5**	**5**
...					
[1,2048]	**2048**	**12**	**1**	**12**	**12**

   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.

$\lfloor \log_2(n) \rfloor+1$. Yes, this formula matches the computed output for a given input. For each step in the array, the remaining pieces are being cut in size to $(n+1)/2$. Therefore, we have a maximum of $\lfloor \log_2(n) \rfloor+1$ steps.

   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**	**1**	**1**	**1**	**1**
[1,3]	**3**	**2**	**1**	**2**	**2**
[1,7]	**7**	**3**	**1**	**3**	**3**
[1,15]	**15**	**4**	**1**	**4**	**4**
[1,31]	**31**	**5**	**1**	**5**	**5**
...					
[1,2047]	**2047**	**11**	**1**	**11**	**11**

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

$O(\log_2 n)$. If $T(n)$ is the time taken at the nth level, $T(n)=T(n/2)+O(1)$. At each level, a comparison is made with a midpoint. This takes $O(1)$ time. Then, a subproblem is computed of searching a value within the $n/2$ values. Given this recurrence relation, the Master Theorem can be applied to deduce that $T(n)=O(\log_2 n)$.

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 $n \geq n_0$

$f(n) = 8n\log n$

$g(n) = 2n^2$

n_0 can be determined by equating $f(n)$ and $g(n)$

$8n\log n = 2n^2$

Divide both sides by n

$8\log n = 2n$

Divide both sides by 2

$4\log n = n$

Solve for $n = 16$

$(16)/(\log_2 16) = 4$

$n_0 = 17$ since for all $n \geq 17$, A will be faster than B (at 16 they're equal).

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?

The worst-case running time of find2D in terms of n is $O(n^2)$. This is because, in the worst case, the element x is the very last item and is searched in an array of n rows and n columns. arrayFind is called n times by find2D. arrayFind will then search all n elements for each call until the final call when x is found. Therefore, n comparisons are done for each arrayFind call. arrayFind will be called n times, so that means there will be $n \times n$ operations, which is an $O(n^2)$ running time. The worst-case running time of the algorithm in terms of N is $O(N)$. The array is of size N which is n^2. In the worst case, the element x is the last item in the array A and the entire size of the array is searched through, which is N. This is done one time, so this is an $O(N)$ running time. The algorithm is not a linear-time algorithm because it would need to have a worst-case running time proportional to its input size, which is n, but its worst-case running time is $O(n^2)$, which makes it a quadratic-time algorithm.

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.
In Big-O notation, there's only an upper bound on the algorithm. The case may be that Bob's algorithm has a better lower bound than Al's algorithm. Also, the $O(n^2)$ algorithm is only effective for small amounts of input n. As the input increases, the efficiency decreases. Typically, asymptotic analysis is done for large inputs. Also, in Big-O notation, the coefficients and lower terms are stripped away. Both contribute to the actual timings.
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*.

```
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]
return s
 f(n)= 2+3(n-1) and O(n)
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]
return s
 f(n)= 2+3(n-1)/2 and O(n)
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]
return s
f(n)=1+3(n-1)(2n*3) and O(n^2)
Algorithm Ex4(A):
   Input: An array A storing n≥1 integers.
   Output: The sum of the partial sums in A.
t←0
s←0
for i←1 to n−1 do
   s←s+A[i]
   t←t+s
return t
```

$f(n)=2+5(n-1)$ and $O(n)$