# CSC 532 Homework 1

## Problem 1

Complete this meeting chart (add rows as appropriate).

Remember: The entire team must meet at least twice.

|  |  |  |
| --- | --- | --- |
| **Meeting Date** | **Who was in attendance** | **What was done (1-3 sentences)** |
| 1/30/24 | 1. David Begu  2. Austin Baird  3. Liam Coyle | Delegation of tasks/planning for individual contributions |
| 2/6/24 | 1. David Begu  2. Austin Baird  3. Liam Coyle | Overview of completed work/combining what was created and answered. |
|  |  |  |
|  |  |  |

## Problem 2

Complete this chart. A little digging may be required.

I’ve done the first one.

If a square is blacked out, don’t worry about it.

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Worst case analysis of execution time in terms of ϴ** | **Average case analysis (if different from the worst case – almost never the case)** |
| Retrieving an object at index k from an array | 1 |  |
| Inserting at the end of a Python list |  |  |
| Inserting at index 0 in a Python list |  |  |
| Removing an item from the end of a Python list |  |  |
| Removing an item from index 0 of a Python list |  |  |
| Finding a value in an unsorted Python list using sequential search |  |  |
| Retrieving an object at index k from a linked list |  |  |
| Inserting at the beginning of a linked list |  |  |
| Removing an item at the beginning of a linked list |  |  |
| Finding a value in an unsorted linked list |  |  |
| Finding a value in a sorted Python list using binary search |  |  |
| The Python in operator such as  x in myList  where x is an object and myList is a Python list |  |  |
| The Python list method index such as  myList.index(“dog”)  which returns the index of the first element whose value is “dog” |  |  |
| Finding a value in a sorted Python list using interpolation search |  |  |
| Sorting a list of numbers using insertion sort |  |  |
| Sorting a list of numbers using merge sort |  |  |
| Sorting a list of numbers using quick sort |  |  |
| Computing the greatest common divisor using Euclid’s algorithm |  |  |
| Brute force matrix multiplication of two n x n matrices |  |  |
| Inserting a key/value pair into a hash table assuming uniform hashing and a load factor < 0.5 |  |  |
| Retrieving a value given a key from a hash table assuming uniform hashing and a load factor < 0.5 |  |  |
| Deleting a key/value pair from a hash table using lazy removal assuming uniform hashing and a load factor of 0.5 |  |  |
| An algorithm whose recurrence relation is T(1) = 1 and T(N) = 2T(N/2) + n |  |  |
| An algorithm whose recurrence relation is T(1) = 1 and T(N) = T(N/2) + 1 |  |  |
| An algorithm whose recurrence relation is T(1) = 1 and T(N) = 2T(N/2) + 1 |  |  |
| An algorithm whose recurrence relation is T(1) = 1 and T(N) = 4T(N/2) + n |  |  |
| An algorithm whose recurrence relation is T(1) = 1 and T(N) = 3T(N/2) + n |  |  |
| An algorithm whose recurrence relation is T(1) = 1 and T(N) = 8T(N/2) + n2 |  |  |
| An algorithm whose recurrence relation is T(1) = 1 and T(N) = 7T(N/2) + n2 |  |  |
| An algorithm whose recurrence relation is T(1) = 1 and T(N) = T(N-1) + 1 |  |  |
| An algorithm whose recurrence relation is T(1) = 1 and T(N) = T(N-1) + n |  |  |
| Assume myList is a Python list of integers of length n.  def mystery1(mylist):  x = mylist[0]  for i in range(0, len(mylist), 2):  if mylist[i] > x:  x = mylist[i]  return x |  |  |
| Assume myList is a Python list of length n.  def mystery2(mylist):  for i in range(len(mylist) - 1):  j = i  while j > 0:  if mylist[i] == mylist[j]:  return False  j = j // 2  return True |  |  |
| Assume matrixA and matrixB are n x n list of lists.  def mystery3(matrixA, matrixB):  matrixC = x = [[0 for i in  range(len(matrixA))] for j in  range(len(matrixA))]  for i in range(len(matrixA)):  for j in range(len(matrixA)):  for k in range(len(matrixA)):  matrixC[i][j] = matrixC[i][j] +  matrixA[i][k]\*matrixB[k][j]  return matrixC |  |  |
| Assume n is a positive integer.  def mystery4(n):  c = 1  k = 1  while k <= n:  c += 1  k = 2 \* k  return c |  |  |
| Assume n is a positive integer.  def mystery5(n):  if n <= 1:  return n  else:  return n\*mystery5(n-1) |  |  |
| Assume myList is a python list of integers with n elements.  def mystery6(myList):  reversedList = []  for j in range(len(myList)):  reversedList.insert(0, myList[j]))  return reversedList |  |  |

## Problem 3

Implement the **book’s** version of Insertion-Sort **and** the **book’s** version of Merge and MergeSort. It’s important to do the book’s version of these algorithms – the book does some unique things.

Be very, very careful. The book assumes the first index is **1**. How crazy is that?!?!? You will have to modify the algorithm to account for the fact that indices start at 0 in Python (or any other programming language).

Also, when using Python for loops, remember that those loops go up to, but do not include, the last value you specify. In other words “for x in range(x, y)”, will only go up to and include **y – 1**.

Be sure to test your methods extensively to make sure they are working. Don’t assume that because they don’t crash that they are working. Don’t assume that because it works on one or two test examples, it is working.

You may use either Python or Java, but I think you find Python to be easier for this problem.

Conduct an experiment that compares the performance of the two sorting algorithms for randomly filled, but unsorted, arrays of sizes 100 up to 10000 stepsize 250 [or similar].

Produce a graph (Excel is a nice tool for doing that) showing the growth rates of the two implemented sorting algorithms, in practice. If the curves that are generated do not look like the expected big-theta of the two algorithms, something is wrong.

Now conduct a similar experiment but pass in already sorted arrays to the two algorithms. Isn’t that interesting?!

The team will turn in the code and the accompanying graphs.

## Problem 4

Implement the **book’s** version of solving the maximum subarray problem.

Also, implement a **brute force** algorithm for solving the maximum subarray problem. The book gives a one-sentence description of this algorithm: “We can easily devise a brute-force solution to this problem: just try every possible pair [of starting and ending indices].” This method would use a pair of nested loops where the outer loop iterates through all the possible starting indices (0 through the length of the list) and the inner loop iterates through all the possible ending indices (the starting index through the length of the list). Compute the sum of the values between those two indices [inclusive] and keep track of the biggest sum you’ve found so far.

Make sure the two algorithms give the same (and correct) answer.

Conduct an experiment that shows the growth in execution time of these two algorithms as the array sizes increases by timing how long it takes to solve randomly filled lists of numbers between [-1000, 1000] of sizes 100 up to 10000 stepsize 500 [or similar]. Produce a graph.

Do the curves on the graph match the expected big-O of the algorithms as described by the book? If not, there is likely an error in your code.

The team will turn in the code and the accompanying graphs.