Questions and Exercises to work out and turn in:

Grading Guidelines:

A right answer will get full credit when:

1. It is right (worth 25%)
2. It is right **AND** neatly presented making it easy and pleasant to read. (worth an **extra** 15%)
3. There is an **obvious and clear link[[1]](#footnote-1)** between 1) the information provided in the exercise and in class and 2) the final answer. A clear link is built by properly writing, justifying, and documenting an answer (worth an **extra** 60%).
4. Calculation mistakes will be minimally penalized (2 to 5% of full credit) while errors on units will be more heavily penalized.

You are welcome/encouraged to discuss exercises with other students or the instructor. But, ultimately, **personal** writing is expected.

* USE THIS FILE AS THE STARTING DOCUMENT YOU WILL TURN IN. **DO NOT DELETE ANYTHING FROM THIS FILE:** JUST **INSERT** YOUR ANSWERS.
* IF USING HAND WRITING (STRONGLY DISCOURAGED), **USE THIS FILE** BY CREATING SUFFICIENT SPACE AND WRITE IN YOUR ANSWERS.
* FAILING TO FOLLOW TURN IN DIRECTIONS /GUIDELINES WILL COST **A 30% PENALTY.**

Objectives of this assignment:

* to use and manipulate the concepts presented in this module
* to propose and write algorithms in pseudocode
* to analyze the time complexity of algorithms
* to analyze the space complexity of algorithms
* to learn autonomously new concepts

What you need to do:

Answer the questions and/or solve the exercises described below.

Questions (15 points):

1. For each of the following problems, propose the key “action/operation” you would use to determine the time complexity (short answers are fine here).
   1. Compute the trace of an n x n matrice A. (The trace of a square matrix is the sum of elements on the diagonal of the matrix.
      1. I would create a function *int traceSum(int mat[][], int n) {…}*.
         1. This function takes in the matrix and dimensions (*n*) as parameters.
         2. I could set the function to take two integer parameters if I knew that not all matrices sent to this function are squares or I could have an override for both instances.
      2. Afterwards, I would initialize the integer *sum = 0*.
         1. This is the actual return integer for the end of the function.
      3. Then, I would start a *for* loop with the statement
         1. *for (int i = 0; i < n; i++){…}*
      4. Within the loop will be the statement:
         1. *{ sum += mat[i][i]; }*
      5. Following the loop would be:
         1. *return sum; }*
      6. Code:
         1. *int traceSum(int mat[][], int n) { // assumes all traces are squares*
         2. *int sum = 0; // initialization of trace int*
         3. *for (int i = 0; i < n; i++) { // assumes n = matrix length*
         4. *sum += mat[i][i]; // sums up the trace*
         5. *} // loops through matrix until i = n, then breaks*
         6. *return sum; } // returns the trace*

Line 2: 1 constant operation

Line 3: An operation loop of size n comparisons

Line 4: 1 operation inside the loop

Total: Time Complexity = *n(1)+1 = n+1*

Big O Notation = *O(n)*

* 1. Find the position (index) of the first occurrence some character in a string s of characters.
     1. The code I would likely be tempted to use:
        1. *int getIndex(String s, char e) {*
        2. *// function with the String to search & char to find parameters*
        3. *int index = 0; // initialization of index*
        4. *while ((index < s.length) && (s.charAt[index] != e)) {*
        5. *index++; // increments if char is not e*
        6. *} // while index <= n, check if this char[] is e*
        7. *if (index > s.length) { return -1; } // char[index] not found*
        8. *return index; } // char[index] found*

Line 3: 1 constant operation

Line 4: A loop of size n (aka s.length) with 2 comparisons

Line 5: 1 operation inside the loop

Line 7: 1 operation post-loop

Total: Time Complexity = *1+2n+1+1 = 2n+3 = 2n+3*

Big O Notation = *O(n)*

* + 1. My above algorithm is not the most efficient algorithm. It would have 2n+3 comparisons (if the letter searched has the index of *n*), but the average would likely be found by the halfway point of any input.
    2. There are many prebuilt functions available in most languages that find this information, such as the much more efficient recursive function *indexOf(array, element, index)*, but the above would be plenty capable of finding the index as well.
  1. Insert a string of characters A into another string B starting from some position.
     1. The code I would likely be tempted to use:
        1. *String insertString(String A, String B, int insert) {*
        2. *// function with Strings A inserted at index insert of String B*
        3. *String strB1 = B.substring(0, insert); // pre-insert A*
        4. *String strB2 = B.substring(insert+1, B.length); // post-insert A*
        5. *return strB1 + A + strB2; } // combined new string*
     2. Admittedly, it is sort of cheating to use the substring function. Upon research I found that most languages have their substring functions set up with *O(n)* time complexity since the substring function simply copies all the characters given as parameters and concatenates them into a new string. Substring functions used to maintain O(1) constant time, but that led to memory leaks.
  2. Compute !n (factorial).
     1. Pseudo code for this one:
        1. *function getFactorial(int n) {*
        2. *if n is 0*
        3. *return 1*
        4. *return n \* getFactorial(n-1)*
        5. *}*
     2. I will doing this if at all possible, because clearly it has a factorial time complexity of !n ---- For example, if n = 5, then this would return an integer of 120.
     3. If n = 20, then this would return attempt and fail to return

2,432,902,008,176,640,000 since the integer data type maxes out far before then.

* + 1. Simple calculations quickly get out of hand. Your computer would likely freeze and or power off from hitting a thermal limit if this went on for too long haha.
  1. Traverse (to display e.g.,) a binary tree (represented as a linked structure where each node contains pointers to its left and right children)
     1. *BinaryTree tree = new BinaryTree();*
     2. *tree.head = new Node(5);*
     3. *tree.head.next = new Node(8);*
     4. *tree.head.next.prev = tree.head;*
     5. *tree.head.prev = new Node(2);*
     6. *tree.head.prev.next = tree.head;*
     7. *tree.head.next.next = new Node(10);*
     8. *tree.head.next.next.prev = tree.head.next;*
     9. This could be put into a called function that recursively applies the pointers of next and prev until the base case of (pointer = null) happens.

1. Research on line and in the textbook what an *in-place* algorithm. Use then your own words to explain the definition of an in-place algorithm and provide an example of an algorithm that is in-place.
   1. That is an interesting algorithm! I have never used something like that before. It appears to utilize space complexity at a near constant. From my prior understanding, this algorithm style would likely fail to compile if you did not temporarily store one of the values in a third variable. One example I saw that circumvented this was to do the following:
      1. *void addSwap(int a, int b)*
      2. *if (a != b) {*
      3. *a = a + b; // a = a + b*
      4. *b = a – b; // b = a + b – b = a*
      5. *a = a – b; // a = a + b – a = b*
   2. That is ingenious as long as the data you are summing fits inside the limits of the data type.

Exercise 1 (25 points)

This exercise explores the concept of best, average, and worst cases for the time complexity.

Consider the following problem: ***insertInSortedLis****t*

**Input**: a number *a* and a **sorted** linked list *L* of numbers

**Output**: a sorted list including *a* and *L* (increasing order)

1. Propose in pseudocode an algorithm for *insertInSortedList*.
   1. void insertInSortedList(int a, Node L) {
   2. Node n = new Node(a, null);
   3. Node itr;
   4. if (L is empty || L.value >= n.value)
   5. n.next = L.head;
   6. L.head = n; *// if 0/1 values in L*
   7. else {
   8. itr = L.head;
   9. while ((itr.next != null) && (itr.next.value < n.value)) {
   10. itr = itr.next;
   11. n.next = itr.next;
   12. itr.next = n; }}} *// loops until finds insert*
2. Propose an action to analyze the time complexity of *insertInSortedList*
   1. Iterating from the head to the end of the list will find the time complexity O(n).
3. Is the time complexity constant for any problem instance? If not, determine and justify the time complexity for the best, average, and worst case.
   1. Only if the list contains 1 element or is empty would it be constant.
   2. Best case – element inserted during first iteration – O(1)
   3. Average case – More than likely, the element will be inserted by halfway into the algorithm – O(n)
   4. Worst case – element inserted during the final iteration – O(n)
4. How do the time complexities for the best case, average case, and worst case grow?
   1. Best case is constant, worst and average would be linear.
5. Analyze the space complexity. Is your algorithm in-place?
   1. I believe so. Nodes are pointers that link to data locations in memory, so any changes should happen in-place since new data isn’t created. My itr node might be unnecessary, though. I only used it to ensure I didn’t modify L’s data that it pointed to. I suppose that the L.head is already set, though, so it might have been fine! I’d have to run the code and test how the data behaves to know for sure.

Exercise 2 (60 points) Analyze "Sum Upper Triangular Matrix" Algorithm

Consider the problem to compute the sum of the upper triangular matrix an n x n matrix A.

Below is the pseudocode of the algorithm to compute this sum

**SumUpperTriangularMatrix(A)**

1: S = 0

2: for j = 1 to n

3: for i = 1 to j

3: // Add

4: S = S + A[i][j]

5: return S

All the questions in this exercise are related to the **SumUpperTriangularMatrix(A)** algorithm. The objective of this **exercise is to explore whether the time complexity will change if we count different “actions”.**

1. **(16 points) Comparison** Action

In this case, we count the number of *comparisons* performed by the “for loops” (Lines 2 and 3). Answer the following questions to determine the total number of comparisons performed by the algorithm *SumUpperTriangularMatrix(A)*.

* 1. How many comparisons are performed by the “for loop” in Line 2? ***n + 1***
  2. Let us call tj the number of comparisons performed by “for loop” in Line 3 for a given value of j. Fill in this table (**Justify** how you find tj **only** for j= 1 and j=n):

|  |  |
| --- | --- |
| j | tj |
| 1 | 2; loop runs from 1 to 1 and breaks when tj = 2 (ie j+1) |
| 2 | 3 |
| 3 | 4 |
| j | j + 1 |
| n | n + 1; loop breaks once i = j+1 |

* 1. Express the **total** number of comparisons performed by the “for loop” in Line 3 during the execution of *SumUpperTriangularMatrix(A).* 
     1. ***n(n+1)/2 comparisons***
  2. Express the function fc(n) that represents the overall total number of comparisons performed by the “for loops” in Lines 2 and 3 during the execution of *SumUpperTriangularMatrix(A).*
     1. ***(n+1)+(n(n+1)/2) = (n+1)(n+1)/2 = comparisons***
  3. The function fc(n) grows like which function?
     1. ***O(n2) – quadratically***

1. **(16 points) Sum** Action (only on Line 4)

In this case, we count the number of *additions* performed by Line 4. Answer the following questions to determine the number of additions performed by Line 4 during the execution of the algorithm *SumUpperTriangularMatrix(A).*

* 1. Let us call aj the number of *additions* performed by Line 4 for a given value of j. Fill in this table (**Justify** how you find tj **only** for j= 1 and j=n):

|  |  |
| --- | --- |
| j | aj |
| 1 | 1 ; loop breaks after j=1 |
| 2 | 2 |
| 3 | 3 |
| j | j |
| n | n; loop breaks after j = n comparison |

* 1. Express the function fa(n) that represents the overall total number of additions performed by Line 4 during the execution of *SumUpperTriangularMatrix(A).* ***n(n+1)/2***
  2. The function fa(n) grows like which function?
     1. ***O(n2) - quadratically***

1. **(20 points) Sum** Action (only on Lines 2 and 3)

In this case, we count the number of *additions* performed by Lines 2 and 3. Answer the following questions to determine the number of additions performed by Lines 2 and 3 during the execution of the algorithm *SumUpperTriangularMatrix(A).*

* 1. Let us call aj the number of *additions* performed by Line 3 for a given value of j. Fill in this table (**Justify** how you find tj **only** for j= 1 and j=n):

|  |  |
| --- | --- |
| j | aj |
| 1 | 1; breaks after i = j |
| 2 | 2 |
| 3 | 3 |
| j | j |
| n | n; loop breaks after i = j |

* 1. Express the function fy(n) that represents the overall total number of additions performed by Lines 2 and 3 during the execution of *SumUpperTriangularMatrix(A).* ***n(n+1)/2***
  2. The function fy(n) grows like which function? ***O(n2)***

1. **(4 points) Compare** the functions fc(n), fa(n), and fy(n) and discuss their “growth”.
   1. **They all have the same “growth”, where their Big O all relies on the rate that n grows**
2. **(4 points) Space complexity**: answer the following questions. Is this algorithm is in-place? What is its space complexity? How does its space complexity “grow”?
   1. **Yes, it is in-place since it is taking a variable (matrix) and just increasing value of the index looping throughout it.**

**What you need to turn in:**

* Electronic copy of this file (including your answers) (standalone). Submit the file as a Microsoft Word or PDF file.
* Recall that answers must be well written, documented, justified, and presented to get full credit.

**Reminder: How this assignment will be graded:**

* A right answer will get full credit when:
* It is right (worth 25%)
* It is right AND neatly presented making it easy and pleasant to read. (worth 15%)
* There is an obvious and clear link between 1) the information provided in the exercise and in class and 2) the final answer. A clear link is built by properly writing, justifying, and documenting an answer (worth 60%).
* Calculation mistakes will be minimally penalized (2 to 5% of full credit) while errors on units will be more heavily penalized.
* You are welcome/encouraged to discuss exercises with other students or the instructor. But, ultimately, personal writing is expected.

**Appendix**: Grading: What is an OBVIOUS and CLEAR LINK?

Here is an example to explain what an **obvious and clear link** is and how we grade your work.

Consider the following problem:

"(100 points) John travels from Auburn to Atlanta in his car at a speed of 50 mph. Leaving at 8am, at what time will John reach Atlanta".

Here are the answers of three students and their scores:

**Student 1** answers: "10am". Student 1 will get 25 points.

**Student 2**answers : "John will reach Atlanta at 10am". Student 2 will get 25+15 = 40 points

**Student 3** answers: "The time t to travel a distance d at speed v is equal to d/v = d/50mph. The problem does not provide the distance d from Auburn to Atlanta. Based on Google, the distance from Auburn to Atlanta is approximately 100 miles (**document is here**). Therefore, the time t = 100 miles/50mph = 2 hours. Since John left at 8am, he will then reach Atlanta at 8am + 2 hours = 10 am".

**Student 3** will get 25 + 15 + 60 = 100 points

Do you see the **direct** **link** going from the data provided in the question to the final answer, using general knowledge/formula and documents?.... Can you now solve the following problem and get 100 points?

"(100 points) Alice travels from Auburn to Atlanta in her car at a speed of 50 mph. Leaving at 8am, at what time will Alice reach Atlanta assuming that she had a flat tire that delayed her 30 minutes".

1. See the appendix for what an obvious and clear link is. [↑](#footnote-ref-1)