Problem	Points
Problem #1 - Short Answer Questions	/20
Problem #2 - BubbleSort	/20
Problem #3 - Recursion	/20
Problem #4 - Guess My Word	/20
Problem #5 - Merge Sort Comparisons	/20
Problem #6 - Binary Trees	/20
Total	/80

Instructions

- This Synthesis will be marked out of 80. The short answer question (**Problem #1**) as well as **Problem #2 and Problem #3** are compulsory. For the rest three (3) Problems (i.e. Problem #4, #5, #6), attempt as many of them as you wish, but only your *Best problem* will be counted. For example, if your marks on the *Problem*#4, #5, #6 are 8, 0, 10 and you got 16/20 for your short answer question and 15 and 20 on Problem #2 and Problem #3 respectively, then your grade will be 10+16+15+20 = 61 out of 80, since your lowest scores will be dropped.
- Think of this Course Synthesis as a week-long individual take-home exam where you may consult your notes, the course textbook, anything on the Canvas Page, and any websites linked from the Canvas Page. However, you may NOT consult your classmates or look at any other online resources. **Any issues with academic integrity will lead to a zero (0) in the entire synthesis**.
- To make it easier for the TAs, submit individual .pdf files for each of the problems and make sure your write the **Problem Number** on top of each page. Notice that Gradescope will help break your solutions to individual pdfs for each question.

(20 pts.) Problem #1 - Short Answer Questions

In this question, you only need to submit your final ANSWERS to these 10 questions. No additional work is required or requested. No justification is required – all I want is your final answer.

Submit a single .pdf file (ideally one page long) on which you will provide your answers to the 10 questions below.

Each correct answer will be worth 2 points. For each incorrect answer, the TAs will decide whether your response will be awarded 1 point (for an answer that is almost correct) or 0 points.

(1) Of the four options below, exactly one lists the various running times from fastest to slowest.

A. $O(1),\ O(n),\ O(n\log n),\ O(n^2),\ O(2^n),\ O(n!),\ O(n^{1000})$
B. $O(1),\ O(n),\ O(n\log n),\ O(n^2),\ O(n!),\ O(2^n),\ O(n^{1000})$
C. $O(1),\ O(n),\ O(n\log n),\ O(n^2),\ O(n^{1000}),\ O(n!),\ O(2^n)$
D. $O(1),\ O(n),\ O(n\log n),\ O(n^2),\ O(n^{1000}),\ O(2^n),\ O(n!)$

Determine which list is correct. Answer either A or B or C or D.

(2) $f(n) = 5n^3 + 10n^2 + 15n + 1000$. Consider the following statements:

i. $f(n) = O(n^1)$
ii. $f(n) = O(n^2)$
iii. $f(n) = O(n^3)$
iv. $f(n) = O(n^4)$

Determine how many of these four statements are TRUE.

(3) Let $T(n)$ be defined by the recurrence relation $T(1) = 1$, and $T(n) = 32T(n/2) + n^k$ for all $n > 1$.

Determine the integer k for which $T(n) = \Theta(n^5 \log n)$.

(4) Let $f(n) = 2^{logn}$ and $g(n) = n^{\sqrt{n}}$. Determine whether $f(n) = O(g(n))$ or whether $g(n) = O(f(n))$

(5) Let A be an array of numbers. In the maximum sub-array problem, your goal is to determine the sub-array $A[x \dots y]$ of consecutive terms for which the sum of the entries is as large as possible.

For example, if $A = [-2, -3, 4, -1, -2, 1, 5, -3]$, the maximum sub-array is $[4, -1, 2, 1, 5]$, and the largest possible sum is $S = 4 - 1 - 2 + 1 + 5 = 7$.

Suppose $A = [1, 2, -4, 8, 16, -32, 64, 128, -256, 512, 1024, -2048]$.

Determine S, the largest possible sum of a sub-array of A.

(6) Suppose we want to use the Heapsort algorithm to sort a large list of numbers.

Our first step is to convert the input list to a heap, and then run BUILD-MAX-HEAP, which applies MAX-HEAPIFY on all the nodes in the heap, starting at the bottom and moving towards the top.

For example, if $A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$, then BUILD-MAX-HEAP(A) makes a total of 7 swaps, and returns the array $[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$.

The swaps are made in this order: $(14, 2), (10, 3), (16, 1), (7, 1), (16, 4), (14, 4), (8, 4)$.

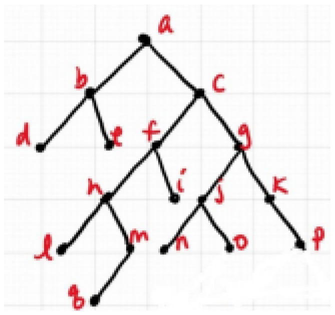
Suppose $A = [2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15]$.

Determine the total number of swaps made by BUILD-MAX-HEAP(A).

(7) In class, we spoke about a a Quick Sort Partition Technique introduced by C.A.R Hoare. We referred to this as the 'Hoare Partition Technique'. This PARTITION(A) procedure inputs an array A, and takes the final element x = A[r] as the pivot element. The output is an array where all elements to the left of x are less than x, and all elements to the right of x are greater than x. Let A = [1,2,7,8,9,6,3,4,5]. Determine PARTITION(A). **You only need to show the first partition.**

(8) A binary tree is a tree where each node has at most 2 child nodes. Clearly and carefully give a correct definition for the (a) Height of a node and (b) Depth of a node .

(9) Consider the binary tree given below. Give the output achieved using (a) Preorder Traversal (b) In order Traversal and (c) Post Order Traversal



(10) A particular balanced binary search tree has the following outputs from tree traversals:
Inorder traversal: A, B, C, D, E, F, G, H, I, J
Post Order Traversal: B, A, D, C, G, F, I, J, H, E
Determine and provide the original balanced binary search tree that gave these outputs.

(20 pts.) Problem #2 - BubbleSort

This question is inspired by our discussion regarding bubblesort in the first week of the CS5800 class. The figures below give pseudocode for the basic bubblesort algorithm and two variations: (a) "early-stopping-bubblesort" and (b) "forward-backward-bubblesort". In this question, you will explore whether there is any asymptotic improvements arising from these two variations.

```
bubbleSort($A[1 \dots n]$)
for $i := 1$ to $n$:
    for $j := 1$ to $n - i$:
        if $A[j] > A[j+1]$ then
            swap $A[j]$ and $A[j+1]$
```

```
early-stopping-bubbleSort($A[1 \dots n]$):
1 for $i := 1$ to $n$:
2     swapped := False
3     for $j := 1$ to $n - i$:
4         if $A[j] > A[j+1]$ then
5             swap $A[j]$ and $A[j+1]$
6             swapped := True
7     if swapped = False then
8         return $A$
```

```
forward-backward-bubbleSort($A[1 \dots n]$):
Construct $R[1 \dots n]$, where $R[i] := A[n - i + 1]$ for each $i$.
for $i := 1$ to $n$:
    Run one iteration of lines 2–8 of early-stopping-bubbleSort on $A$.
    Run one iteration of lines 2–8 of early-stopping-bubbleSort on $R$.
    if either $A$ or $R$ is now sorted then
        return whichever is sorted
```

(a) Compute the worst-case running time of the **early-stopping-bubblesort**. Note that you will need to carefully show an examination of the technique and an entire computation of the running time.

(b) Provide a prove that shows that the best-case running time of the **early-stopping-bubblesort** is asymptotically better than the best-case running time of the basic **bubblesort**

(c) Show that the running time of **forward-backward-bubblesort** on a reverse-sorted array A[1..n] is $\Theta(n)$. Note that the reverse-sorted input is the worst case for both basic **bubblesort** and the **early-stopping-bubblesort**.

(d) Prove that the worst-case running time of **forward-backward-bubblesort** is $O(n^2)$

(e) Prove that despite the improvement shown by the **forward-backward-bubblesort**, its worst case running time is $\Omega(n^2)$. Note that to prove this claim, you will need to explicitly describe an array A[1...n] for which the **early-stopping-bubblesort** requires $\Omega(n^2)$ on both array A and the reverse of A.

(20 pts.) Problem #3 - Recursion

Consider the following three recursive algorithms that all solve the same problem. In this question, you will formulate the running time for each of the algorithm as a recurrence relation and then you shall use different techniques to prove that each of the algorithm requires $\Theta(n)$ time. For each of these assume that selecting a subarray takes $\Theta(1)$ time.

```
B(A[1...n]):
if $n = 0$ then
    return 0
else if $A[1] < 0$ then
    return $1 + B(A[2 \dots n])$
else
    return $B(A[2 \dots n])$
```

```
C(A[1...n]):
if $n = 0$ or ($n = 1$ and $A[1] \geq 0$) then
    return 0
else if $n = 1$ and $A[1] < 0$ then
    return 1
else
    $count := 0$
    $count := count + C(A[1 \dots \lfloor \frac{n}{2} \rfloor])$
    $count := count + C(A[\lfloor \frac{n}{2} \rfloor + 1 \dots n])$
    return $count$
```

```
D(A[1...n]):
if $n = 0$ or ($n = 1$ and $A[1] \geq 0$) then
    return 0
else if $n = 1$ and $A[1] < 0$ then
    return 1
else
    $count := 0$
    $count := count + D(A[1 \dots \lfloor \frac{n}{4} \rfloor])$
    $count := count + D(A[\lfloor \frac{n}{4} \rfloor + 1 \dots \lfloor \frac{3n}{4} \rfloor])$
    $count := count + D(A[\lfloor \frac{3n}{4} \rfloor + 1 \dots n])$
    return $count$
```

(a) Give a recurrence relation for B (and the base case) and then use the change of variable method and the master method to show that the running time is indeed $\Theta(n)$.

(b) Give a recurrence relation for C (and the base case) and then use Guess and Confirm (also known as substitution method) to show that the running time is indeed $\Theta(n)$. For ease, use n as a power of 2. Note that this requires to Guess the answer (which is already given as $\Theta(n)$) and then use Induction to prove that the answer is correct.

(b) Give a recurrence relation for D (and the base case) and then use the backward substitution method or the recursive tree method to show that the running time is indeed $\Theta(n)$.

(c) What problem does A, B, C solve?

(20 pts.) Problem #4 - Guess My Word

This question is inspired by the online Guess My Word challenge, whose URL is:

https://hryanjones.com/guess-my-word/

Each time you enter a guess, the program will tell you whether the secret word is alphabetically *before* your guess, alphabetically *after* your guess, or *exactly* matches your guess.

Each secret word is randomly chosen from a dictionary with exactly $267,751$ words.

(a) Post a screenshot of you winning this game.

You receive full credit if you require at most 20 guesses *or* guess the word within 2 minutes. If you require more than 20 guesses *and* require more than 2 minutes, you will receive partial credit. (You can play this game as often as you'd like! Please submit a screenshot of your best result.)

(b) Suppose the secret word is randomly chosen from a dictionary with exactly $2^k - 1$ words, where k is a positive integer. Describe an algorithm that guarantees that you can identify the secret word in at most k guesses. Clearly justify how and why your algorithm works.

(c) Let $T(n)$ be the maximum number of guesses required to correctly identify a secret word that is randomly chosen from a dictionary with exactly n words.

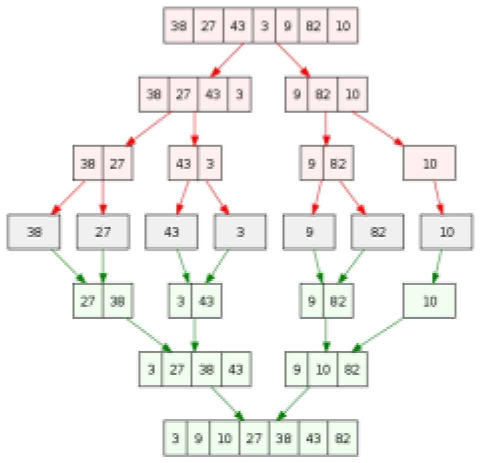
Determine a recurrence relation for $T(n)$, explain why the recurrence relation is true, and then apply the Master Theorem to show that $T(n) = \Theta(\log n)$.

(d) Suppose I give you \$15 to play the online Guess My Word game. Every time you make a guess, you give me \$1. If you agree to play this game with me, do you expect to *win* money or *lose* money? Clearly justify your answer. (Assume that each of the $267,751$ words is equally likely to be chosen.)

(20 pts.) Problem #5 - Merge Sort Comparisons

Merge Sort is a powerful divide-and-conquer algorithm that recursively sorts an array by breaking it into two approximately-equal pieces, applying Merge Sort to each, and then merging the two sorted sub-arrays to produce the final sorted array.

This picture provides a visual illustration of Merge Sort, on the unsorted array [38,27,43,3,9,82,10].



When we merge two sorted sub-arrays, we only compare the left-most element of each sub-array, since one of these two elements is guaranteed to be the smallest. We then repeat the process until one of the two sub-arrays is empty. Then there is nothing left to compare, and we will have our desired merged array.

Let's count the number of *comparisons* needed to produce the combined (and sorted!) merged array.

To get the first green level, we require 3 comparisons (38-27, 43-3, 9-82).
To get the second green level, we require 5 comparisons (27-3, 27-43, 38-43, 9-10, 82-10)
To get the third and final green level, we require 6 comparisons (3-9, 27-9, 27-10, 27-82, 38-82, 43-82)

Thus, Merge Sort requires 3+5+6 = 14 total comparisons to sort the array [38,27,43,3,9,82,10].

(a) $A = [6, 5, 8, 7, 3, 4, 2, 1]$. Perform the Merge Sort algorithm on this array, using a visual illustration of each step to generate the sorted array $[1, 2, 3, 4, 5, 6, 7, 8]$. Show that exactly 12 comparisons are needed to sort this input array A.

(b) Let $M(n)$ be the *minimum* number of comparisons needed to sort an array A with exactly n elements. For example, $M(1) = 0$, $M(2) = 1$, and $M(4) = 4$. If n is an even number, clearly explain why $M(n) = 2M(n/2) + n/2$.

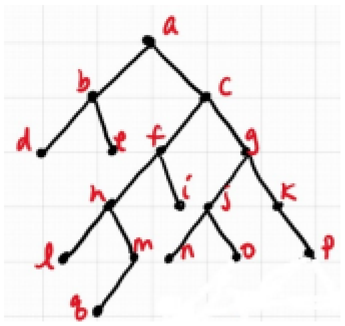
(c) If n is a power of 2, prove that $M(n) = \dfrac{n \log n}{2}$, using any method of your choice. Show all your steps.

(d) Let A be a random permutation of $[1, 2, 3, 4, 5, 6, 7, 8]$. Determine the probability that *exactly* 12 comparisons are required by Merge Sort to sort the input array A. Clearly and carefully justify your answer.

(20 pts.) Problem #6 - Binary Trees

In this question you will explore algorithms on undirected Binary Tree Graphs.

(a) Consider the following undirected binary tree T with 17 vertices.



Starting with the root vertex a, we can use Breadth-First Search (BFS) or Depth-First Search (DFS) to pass through all of the vertices in this tree.

Whenever we have more than one option, we always pick the vertex that appears earlier in the alphabet. For example, from vertex a, we go to b instead of c.

Clearly explain the difference between Breadth-First Search and Depth-First Search, and determine the order in which the 17 vertices are reached using each algorithm.

(b) Let T be an undirected binary tree with n vertices. Show how you can walk through the tree by crossing each edge of T exactly twice: once in each direction.

Clearly explain how your algorithm works, why each edge is guaranteed to be crossed exactly twice, and determine the running time of your algorithm.

(c) Let T be an undirected binary tree. For each pair of vertices, we can compute the distance between these vertices. In the binary tree above, we have $dist(d, i) = 5$ and $dist(l, o) = 6$.

We define the *diameter* of T to be the <u>maximum</u> value of $dist(x, y)$, chosen over all pairs of vertices x and y in the tree.

Clearly explain why the diameter of the above tree is 7.

(d) Let T be an undirected binary tree with n vertices. Create an algorithm to compute the diameter of T. Clearly explain how your algorithm works, why it guarantees the correct output, and determine the running time of your algorithm.