Synthesis 1

Q1 **Short Answer**

10 Points

In each question for Q1, you only need to submit your final ANSWERS to these 5 questions. No additional work is required or requested.

Directions: No partial credits would be awarded for Q1 questions.

Q1.1

2 Points

Of the four options below, exactly one lists the various running times from fastest to slowest. Submit your answer a,b,c or d

Directions: The question is asking for the runtime complexity from best to worst. (left to right)

The following multiple-choice options contain math elements, so you may need to read them in your screen reader's "reading" or "browse" mode instead of "forms" or "focus" mode.

a) O (1), O(n), O(nlogn), O(n^2), O(2^n), O(n!), O(n^{1000})
b) O (1), O(n), O(nlogn), O(n^2), O(n!), O(2^n), O(n^{1000})
c) O (1), O(n), O(nlogn), O(n^2), O(n^{1000}), O(n!), O(2^n)
d) O (1), O(n), O(nlogn), O(n^2), O(n^{1000}), O(2^n), O(n!) - Correct

Q1.2

2 Points

f(n)=$5n^3+10n^2+15n+1000$. Consider the following statements: Determine how many of these four statements are TRUE.

The following checkbox options contain math elements, so you may need to read them in your screen reader's "reading" or "browse" mode instead of "forms" or "focus" mode.

Choice 1 of 4: f(n)=O(n^1)
Choice 2 of 4: f(n)=O(n^2)
Choice 3 of 4: f(n)=O(n^3) - True
Choice 4 of 4: f(n)=O(n^4) - True

Q1.3

2 Points

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)=Θ(n^5logn).

K=5

Q1.4

2 Points

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 77 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).

The total number of swaps will be 19 swaps.

Synthesis 1

Q1.5

2 Points

You have a knapsack that can hold 10 pounds, which you can fill with any of these items.

Object	A	B	C	D	E
Weight (lb)	1	2	3	4	5
Value ($)	5	15	24	30	35

(e.g. Object A has Weight of 1 lb and Value of $5, Object B has Weight 2 lbs and Value of $15, etc)

In the Fractional Knapsack Problem, you are allowed to take *f* of each object, where *f* is some real number between 0 and 1. Your goal is to pick the objects that maximize the total value of your knapsack, with the condition that the chosen objects weigh at most 10 pounds.

Determine the maximum total value of your knapsack.

The maximum value is 76

Q2 Guess My Word

20 Points

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.

Directions: The words are ordered as per the dictionary; a word needs to be a valid English word and the length of a word is << number of words in the dictionary. Note: The system performs a lexicographical comparison between your guess and the hidden word

Q2.1

5 Points

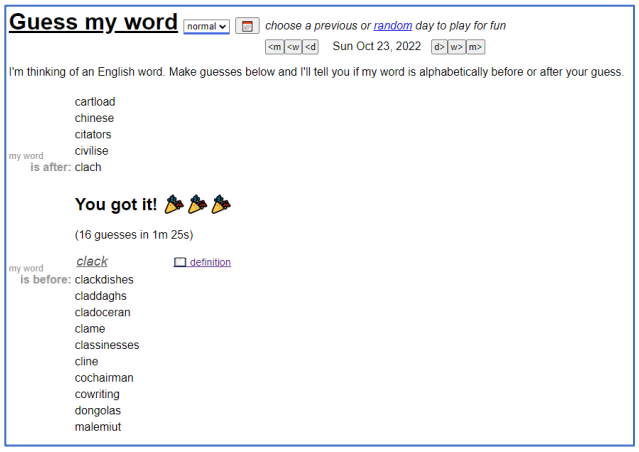
Post a screenshot of you winning this game.

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 *it* 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.)

Directions: You will get full credits if you can guess the word in at most 20 guesses or if you guessed in 2 minutes despite the number of guesses used.

Total Time: 1m 25s

Total Guesses: 16



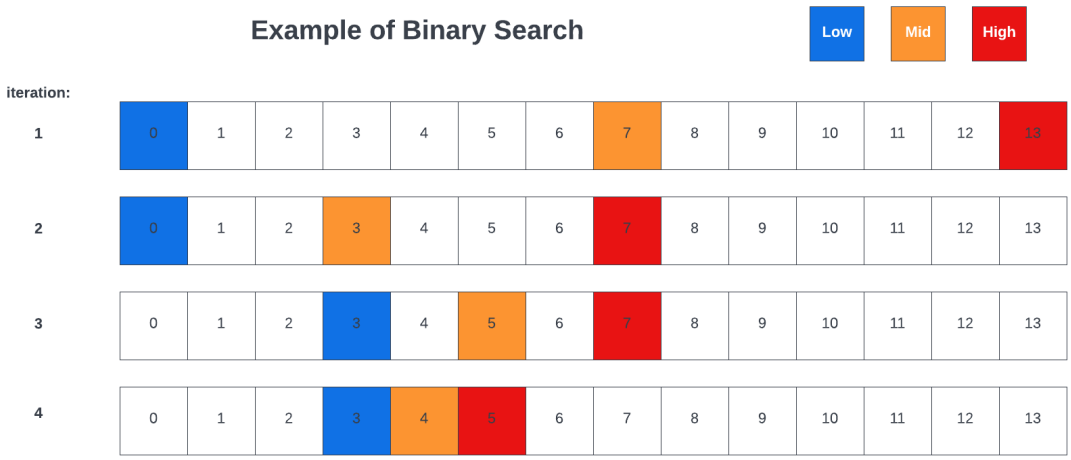
Synthesis 1

Q2.2

5 Points

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.

An ideal answer should have Algorithm/Pseudocode, correct return/print statement, correctness, and runtime of your algorithm. Please clearly state your assumptions regarding the structure of the input.



One algorithm which can use for such an occasion is **Binary search**, which operates by defining a high h, medium m, and low l, point and essentially halving the search space every time a guess is made. Since this dictionary is similar to the one outline in Q2.1, we can **assume** that this dictionary is ordered and we assume that we will know whether the guess that is made is either before or after the randomly selected word. One can use this divide-and-conquer methodology on the dictionary, and the algorithm will call itself on a dictionary half the size of the one in the previous step. The figure above demonstrates this concept as the high and low points shift with each iteration. Some sample pseudocode can be seen below:

- A random word r is selected from the list of words in the dictionary d, containing 2^k-1 words.
- If the dictionary is empty, it should return none, and if it contains a single word, it should return that one word.
- The algorithm then compares the random word r with the middle word of the dictionary
- If the random word matches the middle word, the word is returned/printed and the function ends.
- Else if the random word is after the middle word, then we can call the function, recursively, on the right-half of the list
- Else the random word is before the middle word, then we can call the function, recursively, on the left-half of the list
- The process is repeated and the list off possible answers will shrink until the correct word is found, and returned/printed
- This yields an algorithm with a time complexity of O(log n)

Proof.

- We assume that we are given a sorted dictionary d, containing words.
- If the dictionary is empty, it will return none, which is the correct answer for this case.
- We assume here that if the algorithm is true for an array of strings from lengths 0 to n, then it will also be true for length n+1
- Since the dictionary is sorted, if r <d[m], then we can say r < d[k] for all values of k > m. This will limit the search space to the half of interest, which would be [m, h]
- Given that the range is one of the smaller halves of the initial range of the dictionary [l, h], we can assume that it will work on the other.

Q2.3

5 Points

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)=Θ(logn).

Directions: Please stick to the master's theorem provided in the book (CLRS) - The material is available from the sub-module where master's theorem is discussed in Canvas.

- Since T(n) represents the maximum number of guesses to correctly identify a word, we can specify it asl follows:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

- We can define the relation like this because T(n/2) is the time required to sort the first half of the dictionary array, whereas the later is used on the second half of the dictionary array.
- Using the Master Theorem from CLRS:

$$If\ f(n) = O(n^{log_b a - \varepsilon})\ for\ some\ constant\ \varepsilon > 0, then\ T(n) = \theta(n^{log_b a})$$

$$If\ f(n) = \theta(n^{log_b a}), then\ T(n) = \theta(n^{log_b a} \lg(n))$$

$$If\ f(n) = \Omega(n^{log_b a - \varepsilon}) for\ some\ constant\ \varepsilon > 0, then\ T(n) = \theta(f(n))$$

- Given our recurrence of T(n) = (n/2)+1, we can see that this aligns with the master theorem format of T(n) = a*T(n/b)+n^c
- Given that a=1, b=2, and we can see that c=$\log_2(1)$ = 0, we can see that we default to $\theta(n^0 \log n) = \theta(\log n)$

Q2.4

5 Points

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.)

Directions: Only justification is required. Please show your work.

- Referring to some of the topics covered in CS5002, there are a few different ways one can address this question.
- It is given that the player received $15 to play, but for each guess, there is a cost of $1. We assume that the probability of each of the 267,751 words are the same.
- Let us approach this via the perspective of binary search to start. If we pursue the same method outline in Q2.2, we will require more than 15 guesses to ensure that the correct guess is made.
- With BinarySearch, we half the amount of possible answers until a final result is achieved
- If given $15, there would still be a list of 16 possible options left
- Even if the player happens to randomly select the correct answer the first time, over time they will lose this game.
- Therefore, the player should expect to lose money.

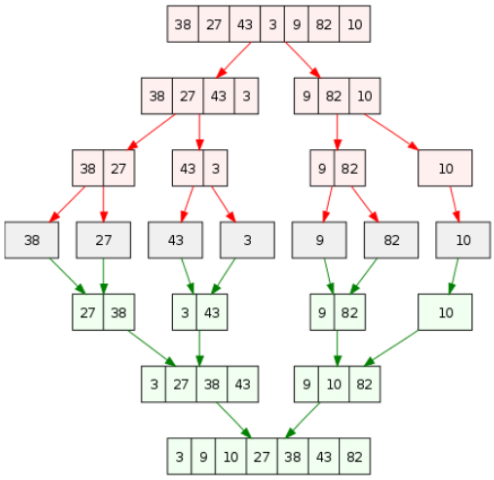
1	267751
2	133876
3	66938
4	33469
5	16734
6	8367
7	4184
8	2092
9	1046
10	523
11	261
12	131
13	65
14	33
15	16
16	8
17	4
18	2
19	1

Q3 **Merge Sort**

20 Points

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.

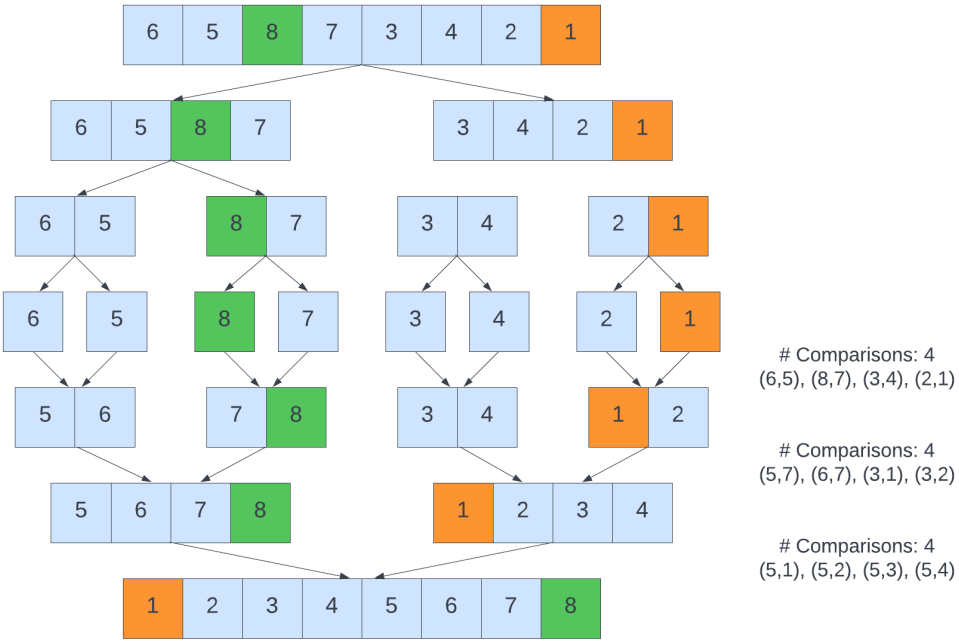
Let's count the number of *comparisons* needed to produce the combined (and sorted!) merged array. To get the first green level, we require 3 comparisons (38–27, 43–3, 9–82). To get the second green level, we require 5 comparisons (27–3, 27–43, 38–43, 9–10, 82–10)
To get the third and final green level, 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].

Q3.1

5 Points

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

Directions: Show step by step visualization of the merge sort algorithm on the given array.



The diagram above outlines the number of comparisons, as well as the specific comparisons needed to answer this question. In each of the layers, exactly four steps are needed to complete the Merge Sort algorithm leading to a total of 12 comparisons. It is worth noting that this is achieved because when a list becomes empty, the values of the other (already sorted list) can simply be copied over into the array without any comparisons.

Q3.2

5 Points

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

Note: the question asks you for the number of comparisons (M) and not the runtime of the equation.

- Since we are looking at an array A with n elements, we notice that the equation divides n by 2
- We can infer from this that the number of comparisons is equal to 2x the comparisons with n/2 comparisons, plus an additional n/2 comparisons.
- We notice that given an array such as the one earlier A=[6,5,8,7,3,4,2,1], once divided into two segments, one would need to compare the elements in each segment to sort it, corresponding to n/2 comparisons
- Using the array A, comprising 8 elements, we can use the equation M(n)=2M(n/2)+n/2 to calculate the number of comparisons: M(8)=2M(8/2)+8/2 = 2*4+4 = 8+4 = 12

Synthesis 1

Q3.3

5 Points

If n is a power of 2, prove that M(n)=(n log n)/2, using any method of your choice. Show all your steps.

Note : n and M are as used in the previous questions

- In this case we wish to prove that M(n) = (n log n)/2 in which n is the number of elements in an array and is a power of 2.
- Recall that powers of 2 comprise {1, 2, 4, 8, ...}
- Note that from the previous question, $M(n) = 2M(\frac{n}{2}) + \frac{n}{2}$
- From CLRS, we see that the Masters Theorem takes the form of:

$$T(n) = aT\left(\frac{n}{b}\right) + \theta\left(n^k (\log n)^i\right)$$

$$where\ in\ our\ case\ a = 2, b = 2, k = 1, and\ f(n) = n/2$$

- With the Master's theorem, we see that it resembles case 2, and therefore $T\epsilon\ \theta\ \frac{(n \log n)}{2}$

Q3.4

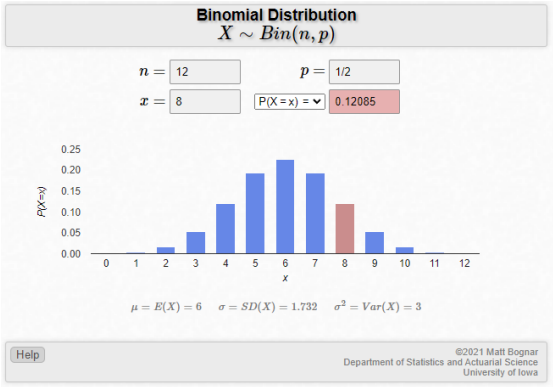
5 Points

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.

- We have an array which is a random permutation of elements that are already sorted in ascending order
- Recall that the MergeSort algorithm (discussed in depth within this Synthesis), is a D&C sorting method by which the array is divided into halves and then arranged back together recursively.
- Given that the length of the array is 8, our objective is to determine the probability of needing 12 comparisons.
- In this case, we can use a Binomial Distribution, as defined by CLRS:

$$\Pr\{X = k\} = \binom{n}{k} p^k q^{n-k}$$

- Since we have a permutation of 8 elements, and are considering that exactly 12 comparisons are made, and through the use of MergeSort we half the array in each iteration, we can infer that n=12, x=8, and p=1/2
- Using a UI Calculator for simplicity (since binomial distributions are not generally part of this course):



- Given the calculation, we arrive at a probability of 0.12085 as a final result.

Q4 **Sorting algorithms**

20 Points

Sorting algorithms are incredibly important and used by us on a daily basis.

In each of the following questions, you are given an input array A with n elements, where n is a very large positive integer. You will then compare two sorting algorithms and justify which of the two algorithms is faster in sorting this array A.

In your responses, you must clearly explain which of the two algorithms runs faster. Make sure you rigorously justify your answer, carefully proving whether each algorithm is O(n), O(nlogn), or $O(n^2)$.

Directions: "Faster" implies better runtime complexity. An ideal answer should include reasoning to why the chosen sorting algorithm would work better than the other. The Log is with base 2.

Q4.1

5 Points

Let A=[5,5,5,5,5,...,5] be an array where all of the elements are equal to 5.

Determine whether SelectonSort or InsertionSort sorts this array faster.

- We can see that the elements are the same throughout the array, inferring that it is already sorted.
- Since the array is already sorted, no swaps amongst the elements will be needed.
- Given those details, InsertionSort is more likely a better option relative to SelectionSort, because InsertionSort operates such that when given a list of elements, you taken an element and insert it into an appropriate position within the list, based on a comparison.
- With that in mind, we will see that InsertionSort will have a time complexity of O(n), whereas its SelectionSort counterpart will have a time complexity of $O(n^2)$. This is because equal items, in the eyes of InsertionSort, will preserve their order as seen in CLRS:
    - Iterate from j=2 to the length of the array A
    - Set a key to be A[j]
    - Insert A[j] into the sorted sequence
    - Assign i as j-1
    - Iterate over A while i>0 and A[i] > key
    - And then assign a[i+1] = A[i]
    - Then, assign i = i-1
    - Finally, we set A[i+1] = key
- We can see from the fact that the order is preserved, which means less comparisons, and so given the linear nature of this, we see that this means InsertionSort grows at O(n)
- We can prove this via a loop invariant:
    - Initialization: Let A but the input array, which is already sorted and contains the same number repeatedly.
    - Maintenance: Using the algorithm above, we notice that no swaps occur, and the other is preserved, with far fewer comparisons giving a linear nature to the algorithm.
    - Termination: We see that the order of the array has not changed, and remains the same. We can see from this preservation of order that the best-case scenario for InsertionSort is applicable, giving us a time complexity of O(n)

Synthesis 1

Q4.2

5 Points

Let A=[n, n−1, n−2,...,3,2,1] be an array where the first n positive integers are listed in decreasing order.

Determine whether Heapsort or Quicksort sorts this array faster.

For this question, assume the Quicksort pivot is always the right-most element.

- Looking at the array we see that it is in a descending order where the values on the left and decrementing by 1 as the move to the right.
- The average time complexities of HeapSort and QuickSort are generally considered at O(n logn) and are in many cases equivalent in their performance.
- Given that QuickSort algorithm, which uses a pivot which for the purposes of this question we are considering the right-most element, this poses a problem with the given array. If fact, this represents a worst-case time complexity for the QuickSort algorithm, giving us a time complexity of $O(n^2)$. QuickSort relies on this pivot being in a favorable location, which within the confines of this descending list, is not favorable at all.
    - We can see this within the context of the recurrence relation T(n) = T(n-1) + n:

$$
\begin{aligned}
&T(n) = T(n-1) + n \\
&t(n-1) = T(n-2) + n - 1 \\
&T(n-2) = T(n-3) + n - 2 \\
&\dots \\
&T(n) = T(n-k) + kn - \frac{k(k-1)}{2} \\
&\dots substitution \\
&T(n) = T(1) + (n-1)n - \frac{(n-1)(n-2)}{n} \\
&\dots which\ renders\ to\ O(n^2)
\end{aligned}
$$

- This value represents the worst case time complexity for QuickSort
- However, we can see that HeapSort has a worst case time complexity of O(n log n)
- Therefore, by comparing the two, we see that HeapSort is more favorable.
- We can prove this via a loop invariant:
    - Let us assume for the purpose of this HeapSort algorithm, we are discussing a heap such as the same one we discussed in lecture
    - We can use our Heap algorithm to prove that we maintain an invariant
    - Initialization: Before our initialization and iteration of our loop, we can assume that we have a heap
    - Maintenance: The children of reach node, or element n, are numbered iteratively and decreasing as n-1 from left to right in the input array. As a heap, the MaxHeap would preserve the loop invariant here and maintain it over the course of each iteration or step.
    - Termination: The process terminates depending on the given graph, or when the iteration is complete.

Q4.3

5 Points

Let A be an array, where each of the n elements is a randomly chosen integer between 1 and n. For example, if n=12, this array could be

A=[3,5,1,10,5,7,9,12,2,8,8,6].

Determine whether **Bubble Sort** or **Bucket Sort** sorts this array faster.

- In this case we see that array A contains elements chosen at random between 1 and n.
- BucketSort involves the idea of grouping elements in which the elements are grouped into buckets, and each bucket sorts the elements within it, with the buckets being recombined together in the end.
- On the other hand, BubbleSort operates by swapping two elements at each interval or comparison, until the entire list is sorted.
- The time complexities of BubbleSort and BucketSort are $O(n^2)$ and O(n+k), respectively.
- Given the even distribution of the numbers and the fact that the range of the values is known from 1-n, then BucketSort will be faster relative to BubbleSort.
    - Proof
    - Let us assume we have a number of integers n, which can be divided by k bins, taking O(n) time
    - The next step would be the process of sorting each of the bins respectively, using an algorithm such as QuickSort or InsertionSort.
    - Next, the assembling of the array would take O(n+k) time as it depends on the number of elements and buckets
    - We can see this as well in the relation in which we represent the time to insert elements + time to go through the array: T(n) = O(n) + (n-1) = O(n)

Q4.4

5 Points

Let A be an array, where each of the n elements is a randomly chosen digit between 0 and 9. For example, if n=12, this array could be

A=[3,5,1,0,5,7,9,2,2,8,8,6].

Determine whether **Counting Sort** or **Merge Sort** sorts this array faster.

- We are given an array A with n elements chosen at random from between 0 and 9, and therefore we know the range of possible values within this list.
- If we take a look at the two algorithms, we know that CountSort performs well when a range is known, giving us a time complexity of roughly O(n), whereas MergeSort which can be used on any given array, has a time complexity of O(n log(n)).
- The key idea here is that CountSort relies on knowing the range of elements ahead of time. It is not a comparison-based algorithm, but instead counts or hashes the values as a count and then sorts them accordingly.
- Given that the array of length n consists of k distinct elements, which in this case k=10, then the time complexity would be O(n+k) = O(n+10) = O(n)
- Given that O(n) is 'faster' than O(n log n), then we can conclude that CountSort is more appropriate here.

Q5 Graph Coloring

30 Points

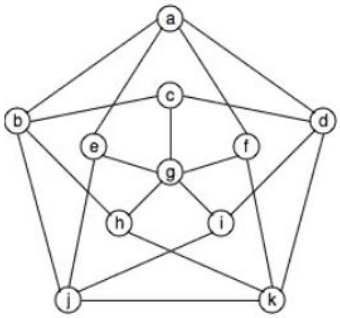
Given a graph G, we say that G is k-colorable if every vertex of G can be assigned one of k colours so that for every pair u,v of adjacent vertices, u and v are assigned different colors.

The `chromatic number` of a graph G, denoted by χ(G), is the smallest integer k for which graph G is k-colorable. To show that χ(G)=k, you must show that the graph is k-colorable *and* that the graph is not (k−1)-colorable.

Q5.1

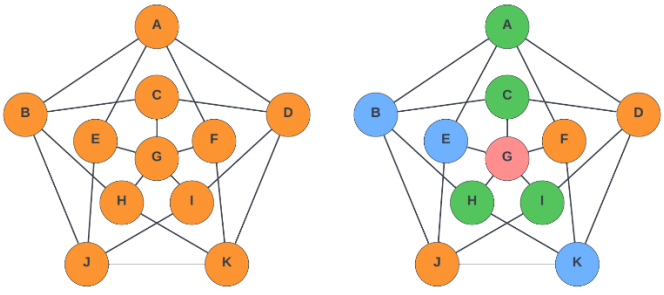
5 Points

Let G be the graph below, with 11 vertices and 20 edges. Clearly explain why χ(G)=4.



Directions: You can choose to show it visually or write it in words.

- We define the chromatic number as the minimal number of colors needed to color the nodes of a given graph, which we denote χ(G) where G is our graph. The key idea here is that no two adjacent vertices/nodes have the same color.
- Since the graph contains 11 vertices/nodes. And 20 edges, our objective is to show that χ(G) = 4.
- Starting at node A, we can color that node green, and then color its adjacent nodes in different colors to abide by the given constraint. If we follow this pattern of color items as either a new color when needed, or a previous color when able to, we can minimize the number of colors while still ending that no two nodes have the same color when adjacent.



- If you look at the diagram above, you can see the graph colored such that four colors are used (Red, Green, Blue and Orange).
- Given the shape of the graph, this can be represented in multiple ways.
- Since we can color the graph in only four colors, we can say that for graph G, χ(G) = 4.
- This is because we bucketed the nodes into the follow buckets of colors:
  - Blue : B, E, K
  - Green: A, C, H, I
  - Orange: D, F, J
  - Red: G

Q5.2

5 Points

Create a simple greedy algorithm for coloring the vertices of any graph G, ideally using as few colors as possible. Explain how your algorithm works, i.e., the order in which your algorithm chooses the vertices of a given graph, and how a color is assigned to each vertex.

Directions: You need to provide the greedy criteria, algorithm/pseudocode, and the correct runtime analysis of your algorithm.

- Since the objective is to create an algorithm that will traverse the graph and color each of the vertices/nodes with a color, such that the minimum number of colors are used and no two adjacent vertices have the same color, we can use the following algorithm:

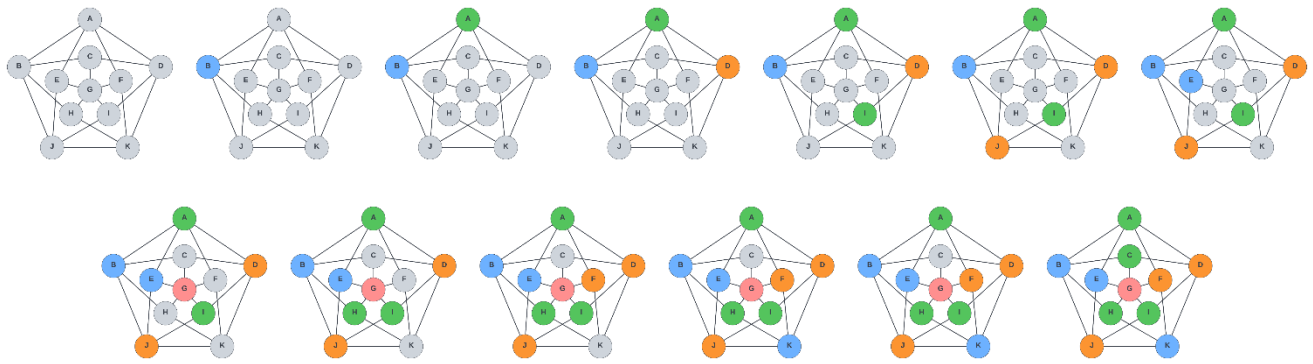
    1. First, we will create a list that will allow us to track the colors for each of the nodes/vertices
    2. We can begin by selecting the node that contains the fewest number of edges, if that exists, otherwise a random node
    3. We color that node with a given color and assign that in our list
    4. Next, we select a neighboring node with the least number of nodes that are assigned a color
    5. We then color that node with a color different than its neighbors, based on the list we kept
    6. We can repeat steps 4 and 5 until all the nodes are colored.

- Given the nature of this algorithm and the fact that we iterate over all of the nodes with the graph, and in each case assign a color based on the adjacent nodes, we will see a runtime of roughly $O(n^2)$.

Q5.3

5 Points

Apply your algorithm (from 5.2) to the graph in 5.1. Show step by step. How many colors did your algorithm use?



- In the illustration above, you will see the steps taken, based on the algorithm to color the diagram in graph G. The algorithm managed to color it using 4 colors.

Q5.4

5 Points

For any graph G, does your algorithm always use exactly χ(G) colours? If so, explain why. If not, provide a graph G for which your algorithm requires more than χ(G) colours.

Directions - You need to provide a proof of correctness if your algorithm gives the correct chromatic number in all the cases. Otherwise, just a counter example is enough.

- Using the algorithm developed above, it should still give an answer of 4 in the sense that it will always use χ(G) colours.
- We use a similar algorithm in the field of chemistry to make arguments about the symmetry of small molecules, most notably Phenol, in cases similar as this.
- We can use a contradiction to prove this.
    - Let us assume that the graph only requires 3 colors instead of 4.
    - Let us also assume that the central vertex is in its own category of color, such that no other vertex shares the same color. The other vertices are split across the other two colors
    - For every vertex v_{out} in the outermost ring, let us say that is some other vertex in the inner ring v_{inn} such that they have the same neighbour within the graph
    - Looking over the graph, for every vertex, if the vertices v_{out} are colored with the other colors, we can ignore them for now, whereas if they match the color of the central vertex we change their color to match that of $v_{inn.}$ Upon this change, we notice that this now contradicts the chromatic number of 3 since we now have an outer ring comprising of two colors.
    - Therefore, we can see that via this contradiction, we must arrive at the fact that the answer must indeed be 4 or χ(G) colors.

Q5.5

10 Points

The problem of determining whether an arbitrary graph has chromatic number k, where k≥3 is a very hard problem (this problem falls under NP-Complete and means there does not currently exists an efficient algorithm for solving the problem). However, determining whether an arbitrary graph has chromatic number 2 is much easier (there does exist efficient algorithms to do so, we say that they fall under **P**).

Given a graph G on n vertices, create an algorithm that will return TRUE if χ(G)=2 and FALSE if χ(G)! =2. Clearly explain how your algorithm works, why it guarantees the correct output, and determine the running time of your algorithm.

- Our objective is create an algorithm concerning the chromatic number that will return TRUE if χ(G)=2 and FALSE if χ(G)! =2
- Recall that bipartite graph, as defined by CLRS, is a graph where the vertices can be observed as two disjoint sets where edges connect the vertices of sets u and v, when the value of the chromatic number is 2.
- We can develop an algorithm to accomplish this:
    - Initialize two distinct empty sets which we will call v and u, respectively.
    - Select a vertex at random, and initialize that as a variable
    - Using the Breadth-First Searching algorithm, we will identify the vertices, and populate set V with them
    - Next, we add all the neighbors of that vertex into set U
    - We must add a conditional here that if a vertex is populated into both sets, we return a false
    - Finally, once both sets u and v are populated, we return a true
- Given the traversal nature of this algorithm we can see that the time complexity given the vertices v and edges e, would not only be O(V+E), but we also need to incorporate the additional rigor in step 3, giving us $O((V+E)V) = O(n^3)$

References:

[1] CS5800 Course

[2] Introduction to Algorithms, Cormen, Third Edition. (CLRS)

[3] LucidChart Drawing Tool (https://lucid.app/lucidchart)

[4] Binomial Distribution Calculator (https://homepage.divms.uiowa.edu/~mbognar/applets/bin.html)