Synthesis 1

Question 1

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.

Choice 1 of 4:

a) O (1), O(n), O(nlogn), O(n2), O(2^n), O(n!), O(n^1000)

b) O (1), O(n), O(nlogn), O(n2), O(n!), O(2^n), O(n^1000)

c) O (1), O(n), O(nlogn), O(n2), O(n^1000), O(n!), O(2^n)

d) O (1), O(n), O(nlogn), O(n2), O(n^1000), O(2^n), O(n!)

Answer: d

Question 1.2

f(n)=5n3+10n2+15n+1000. Consider the following statements: Determine which 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(n1)

Choice 2 of 4:f(n)=$O(n^2)$

Choice 3 of 4:f(n)=$O(n^3)$

Choice 4 of 4:f(n)=$O(n^4)$

Answer:

3. f(n)=$O(n^3)$,f(n)=$O(n^4)$

Question 1.3

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 $T(n) = \Theta(n^5 * logn).$

Answer: 5

Q1.4

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 .[16,14,10,9,8,7,4,3,2,1]

The swaps are made in this order: .

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).

Answer:

8

Q1.5

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.

Answer:

To get the higher maximum value we have to follow a greedy approach of taking the object which is having highest value/weight ratio. so we have to sort the given array using value/weight ratio we get the following table:

Object	**Value ($)**	**Weight (lb)**	**Value-to-Weight Ratio**
C	24	3	8
B	15	2	7.5
D	30	4	7.5
E	35	5	7
A	5	1	5

Now accordingly,

We choose 3 pounds of C, 2 pounds of B, 4 pounds of D, 1 Pound of E that gives you **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/Links to an external site.

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.

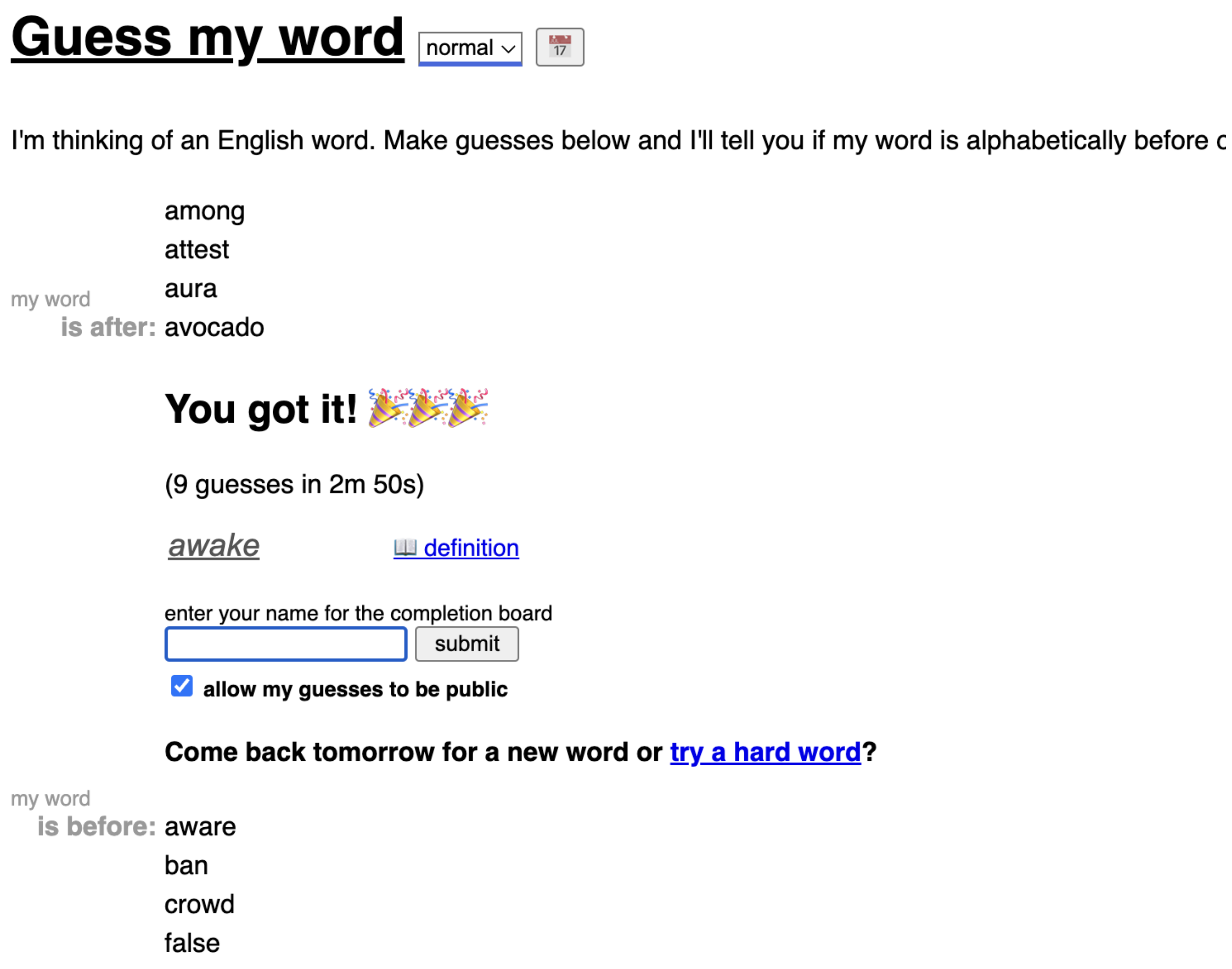
Directions: The words are ordered as per the dictionary; a word needs to be a valid English word and the length of a word is << number of words in the dictionary. Note: The system performs a lexicographical comparison between your guess and the hidden word

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.)



I'm thinking of an English word. Make guesses below and I'll tell you if my word is alphabetically before c

among
attest
my word aura
is after: avocado

You got it! 🎉🎉🎉

(9 guesses in 2m 50s)

awake 📖 definition

enter your name for the completion board

submit

☑ allow my guesses to be public

Come back tomorrow for a new word or try a hard word?

my word
is before: aware
ban
crowd
false

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.

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.

Algorithm to Identify Secret Word in at Most k Guesses:

Assumptions:

- The secret word is randomly chosen from a dictionary with exactly 2^{k-1} words.
- The dictionary is ordered in some way (e.g., alphabetical order).
- We have access to the dictionary and can perform binary searches.

Pseudocode:

1. Set the range of possible words as the entire dictionary.
    - Let the lower bound be "start" initially pointing to the first word in the dictionary.
    - Let the upper bound be "end" initially pointing to the last word in the dictionary.
2. Initialize the variable "guesses" as 0.
3. Repeat the following steps while start <= end:
   a. Increment "guesses" by 1.
   b. Compute the mid index as (start + end) / 2.
   c. Guess the word at the mid-index.
   d. If the guessed word is the secret word, return/print the secret word and the number of guesses.
   e. If the guessed word comes before the secret word in the dictionary order:
   - Update the lower bound "start" to mid + 1.
   f. If the guessed word comes after the secret word in the dictionary order:
   - Update the upper bound "end" to mid-1.
4. Return/print that the secret word was not found (optional).

Justification:
This algorithm is based on the idea of binary search. By dividing the range of possible words in half with each guess, it narrows down the search space exponentially. Since

the dictionary has exactly 2^{k-1} words, and we divide the search space in half at each step, it will take at most k guesses to find the secret word.

Correctness:
The algorithm guarantees that the secret word will be identified in at most k guesses because it exploits the property of binary search to efficiently search a sorted list. By dividing the search space in half at each step, we eliminate half of the remaining words that cannot be the secret word. Eventually, we narrow down the search to only one word, which is the secret word.

Runtime:
The runtime complexity of this algorithm is O(k) because at each step, we halve the search space. Since the number of words in the dictionary is 2^{k-1}, the algorithm performs k iterations at most to identify the secret word. Therefore, the algorithm runs in linear time with respect to k.

Note: The assumption of having access to the dictionary and being able to perform binary searches is essential for this algorithm to work.

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.

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.

Answer:

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).

To determine the recurrence relation for T(n), let's analyze the binary search process used to identify the secret word from a dictionary with n words.

In each guess, we divide the search space in half by selecting the middle word. If the guessed word is not the secret word, we eliminate half of the remaining words and continue the search in the remaining half. Therefore, the recurrence relation for T(n) can be expressed as follows:

T(n) = 1 + T(n/2)

Explanation:

- The "1" term represents the initial guess we make, which counts as one guess.
- T(n/2) represents the maximum number of guesses required to find the secret word in the remaining half of the dictionary after the initial guess.

This recurrence relation is true because, in each step, we divide the search space in half, reducing the problem size to half of the previous search space. The additional "1" term accounts for the initial guess made at each level of the binary search.

Now, let's apply the Master Theorem to show that T(n) = Θ(logn):

The recurrence relation is T(n) = 1 + T(n/2).

Comparing this to the standard form of the Master Theorem:
T(n) = aT(n/b) + f(n),

where a = 1, b = 2, and f(n) = 1.

The value of f(n) is constant and satisfies f(n) = O(n^c) for c = 0, which means f(n) falls under Case 1 of the Master Theorem.

Since log_b(a) = log_2(1) = 0, which is equal to c, the Master Theorem Case 2 applies.

According to Case 2, if f(n) = O(n^c), where c >= 0, and log_b(a) = c, then T(n) = Θ(n^c * logn).

In our case, c = 0, so T(n) = Θ(logn).

Therefore, by applying the Master Theorem, we have shown that T(n) = Θ(logn), indicating that the maximum number of guesses required to identify the secret word from a dictionary with n words grows logarithmically with the size of the dictionary.

Q2.4

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.

In the Guess My Word game, there are 267,751 possible words that can be chosen, each with an equal probability of being selected. For each guess, we pay $1.

To determine whether we expect to win or lose money, we need to consider the average number of guesses required to correctly guess the word. The worst-case scenario is when we have to make the maximum number of guesses, which is equal to the total number of possible words.

if you use the above algorithm correctly the maximum number of chances would be $log_2(267,751)$ that is approximately 18.3 ~ 19 chances but as we have only 15 dollars only number of chances to guess the word is 15 so in this case we are very likely not to win the money as we exhaust the money.

Using Probability:

Considering that my guesses are independent and every word is equally likely to be chosen, the
possibility of failing to guess the correct word in 15 attempts can be calculated as follows:

$$(1-1/267,751)^{15} = 0.9945$$

This implies that there is a 99.45% likelihood of not guessing the word correctly within 15 tries,

ending the game and resulting in a loss of all $15.
Moreover, the probability of correctly guessing the word in one try is exceptionally low at 0.000373%.

As a result, the expected value of my earnings per guess is:

(0.000373% * $1) - ((1 - 0.000373%) * $1) = -$0.999

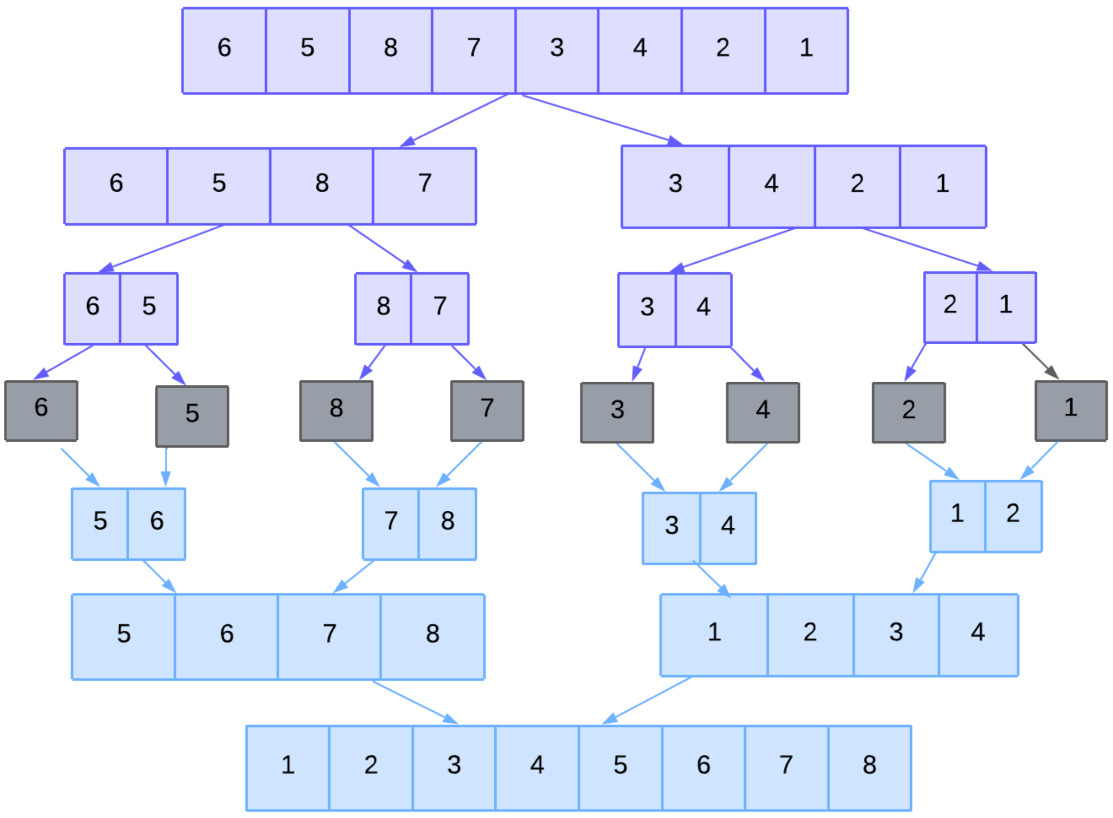
Therefore, for every guess I make, I should expect to lose approximately $1.

In conclusion, it is highly probable that I will lose money if I choose to play this game with you,
and there is no possibility of winning money in the long run.

Q3 Merge Sort

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 visually illustrates the Merge sort on the unsorted array [6,5,8,7,3,4,2,1].



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 blue level, we require comparisons (6-5,8-7,3-4,2-1). To get the second green level, we require comparisons (5-7,6-8,3-1,3-2)

To get the third and final green level, we require comparisons (5-1,5-2,5-3,5-4)

Thus, Merge Sort requires 4+4+4=12 total comparisons to sort the array [6,5,8,7,3,4,2,1].

Q3.2

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.

When we have an even number of elements, we can divide the array into two equal halves. Let's assume the two halves are A1 and A2.

To sort the entire array A, we can first sort A1 and A2 separately, and then merge them back together. The process of merging the two sorted halves requires comparing the elements from both halves.

Here's a step-by-step breakdown:

1. Sort A1: To sort the first half, we need M(n/2) comparisons according to the definition of M(n).
2. Sort A2: Similarly, to sort the second half, we also need M(n/2) comparisons.
3. Merge A1 and A2: When merging the two sorted halves, we need to compare the elements from both halves to determine their order. Since A1 and A2 each have n/2 elements, we need n/2 comparisons to merge them.

Overall, the total number of comparisons required to sort the entire array A is the sum of the comparisons made during the sorting of A1, A2, and the merging step:

M(n) = M(n/2) + M(n/2) + n/2 = 2M(n/2) + n/2.

Therefore, when n is an even number, the minimum number of comparisons needed to sort an array A with n elements is given by the recurrence relation M(n) = 2M(n/2) + n/2.

Q3.3

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

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

To prove that M(n) = (nlogn)/2 when n is a power of 2, we can use mathematical induction.

Base Case:
For n = 2, we have M(2) = 1 (as given in the question).

Inductive Hypothesis:
Assume that for any power of 2, say k, the statement $M(k) = (klogk)/2$ holds true.

Inductive Step:
We need to prove that if$M(k) = (klogk)/2$ holds true, then $M(2k) = (2klog2k)/2$ holds true as well.

Using the recurrence relation for M(n), which is $M(n) = 2M(n/2) + n/2,$ let's substitute $n = 2k$:

$M(2k) = 2M(2k/2) + (2k)/2$
= 2M(k) + k

Using the inductive hypothesis, we can replace M(k) with (klogk)/2:

$M(2k) = 2(klogk)/2 + k$
$= klogk + k$

Now, let's simplify the expression further:
$M(2k) = k(logk + 1)$

We know that$logk + 1 = log2k$ (since k is a power of 2).

Therefore,
$M(2k) = klog2k = (2klog2k)/2$

Hence, we have shown that if the statement $M(k) = (klogk)/2$ holds true for any power of 2 k, then the statement $M(2k) = (2klog2k)/2$ holds true as well.

By the principle of mathematical induction, we have proven that $M(n) = (nlogn)/2$ for any power of 2, where n is a positive integer.

Q3.4

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.

Answer

To determine the probability of exactly 12 comparisons to sort input array, we need to count the number of permutations of A that require exactly 12 comparisons and divide by total number of permutations possible for A.

1. For an array of length 8, to divide an array into two halves. 2. Total number of permutations to arrange 8 elements in an array is 8! = 40,320.
2. To get the minimum number of operations, let us assume that dividing an array of size 8 into halves, such that each half is of size 4, and each half is already sorted. The array is already sorted since it will have minimum number of operations. There are two ways of doing this either left half will be greater than right half or vice-versa (since both halves are sorted in themselves);

for example left half can be, left = (1,2,3,4) and right half can be right =(5,6,7,8) or vice-versa. This leads to 2 ways of dividing an array.

1. Following the same logic from step 3, again divide the array of size 4 into sizes of 2 each (left subarray and right subarray), such that they are already sorted. For example, 1,2,3,4 can be divided into left = (1,2) and right = (3,4) or vice-versa. To do this, there are four ways. Way 1= (1,2), (3,4); Way 2= (3,4), (1,2); Way 3= (5,6), (7,8); Way 4= (7,8),(5,6). This leads to 22 = 4 combinations.
2. Similarly, each subarray of size 2, can be further divided into subarray of size 1, and there will be total 16 ways to arrange them, that is 24 = 16 combinations. (1,2 or 2,1 and so on...)
3. Combining total ways from step 3,4, and 5, total ways to get exactly 12 comparisons (12 are minimum number of comparisons on size 8) for array of size 8 will be 24.16=128 7. Therefore, from step 6 and 7, the probability that exactly 12 comparisons are required by Merge Sort to sort the input array A is 128/40320 = 0.00317.

Question 4.1

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

Determine whether Selection Sort or Insertion Sort sorts this array faster.

To determine whether Selection Sort or Insertion Sort is faster for sorting the given array A, let's analyze the time complexity of each sorting algorithm.

1. Selection Sort:
   Selection Sort is a comparison-based sorting algorithm that divides the input array

into a sorted and an unsorted portion. It repeatedly selects the smallest or largest element from the unsorted portion and swaps it with the first element of the unsorted portion. This process continues until the entire array is sorted.

The time complexity of the Selection Sort is $O(n^2)$ in all cases, regardless of the order or distribution of the elements. This is because it always requires n iterations to find the smallest or largest element in each pass and perform swaps.

1. Insertion Sort:
   Insertion Sort is a comparison-based sorting algorithm that works by dividing the input array into a sorted and an unsorted portion. It iterates through the unsorted portion, selects an element, and inserts it into its correct position within the sorted portion. This process continues until the entire array is sorted.

The time complexity of Insertion Sort is also $O(n^2)$ in the worst case, but it can perform better than Selection Sort for certain types of input data. In the best-case scenario, when the array is already sorted, the time complexity reduces to $O(n)$, making Insertion Sort more efficient.

Comparing the time complexities of Selection Sort and Insertion Sort, we can see that both algorithms have the same worst-case time complexity of $O(n^2)$. However, in the case of an array where all elements are equal (as in A=[5,5,5,5,5,...,5]), Insertion Sort would perform better.

Insertion Sort benefits from the fact that it has a best-case time complexity of $O(n)$ when the input array is already sorted. Since the given array A is already sorted, Insertion Sort can quickly determine that each element is already in its correct position, resulting in fewer unnecessary comparisons and swaps.

Therefore, for sorting an array A where all elements are equal to 5, Insertion Sort would be expected to sort the array faster than Selection Sort.

Question 4.2

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.`

Heapsort and Quicksort are both sorting algorithms with a worst-case time complexity of O(n log n). However, the average-case time complexity of Heapsort is O(n log n), while the average-case time complexity of Quicksort is O(n log n) in the best case and O(n^2) in the worst case.

In the case of the array A, where the elements are in decreasing order, the worst-case scenario for Quicksort occurs. This is because the pivot element will always be the smallest element in the array, which means that the algorithm will have to divide the array into two subarrays of equal size. This will happen recursively until the array is sorted.

In contrast, the heapsort algorithm will always be able to extract the minimum element from the heap in O(log n) time. This means that the heapsort algorithm will be able to sort the array in O(n log n) time, even in the worst case.

Therefore, heapsort will sort the array A faster than quicksort.

Here is a table that summarizes the time complexity of heapsort and quicksort for the array A:

Algorithm	**Worst Case**	**Average Case**
Heapsort	O(n log n)	O(n log n)
Quicksort	O(n^2)	O(n log n)

drive_spreadsheetExport to Sheets

As you can see, heapsort has a better worst-case time complexity than quicksort. In addition, heapsort has the same average-case time complexity as quicksort, but it is not as sensitive to the arrangement of the elements in the array. This means that heapsort is a more robust sorting algorithm than quicksort.

Therefore, heapsort is the better choice for sorting the array A.

Question 4.3

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.

Bucket sort will be faster than bubble sort as we got range

Bucket Sort will sort the array faster than **Bubble Sort**.

To determine whether Bubble Sort or Bucket Sort is faster for sorting the given array A, we need to analyze the time complexity of each sorting algorithm.

1. Bubble Sort:
   Bubble Sort is a simple comparison-based sorting algorithm that repeatedly steps through the array, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the array is sorted.

The worst-case time complexity of Bubble Sort is $O(n^2)$, where n is the number of elements in the array. In the worst case, when the array is in reverse order, Bubble Sort needs to make n passes and perform comparisons and swaps for each element.

1. Bucket Sort:
   Bucket Sort is an efficient sorting algorithm that distributes the elements of an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm or recursively applying Bucket Sort.

The time complexity of Bucket Sort depends on the sorting algorithm used to sort each bucket. In this case, since the elements in array A are randomly chosen integers between 1 and n, we can use a comparison-based sorting algorithm like Insertion Sort to sort each bucket. The average time complexity of the Insertion Sort is $O(n^2)$, but in this case, the average case is improved due to the relatively uniform distribution of elements.

The overall time complexity of Bucket sorting can be expressed as $O(n + k$), where n is the number of elements in the array and k is the number of buckets. Since k is typically chosen to be a small constant relative to n, the time complexity of Bucket Sort is often considered to be linear, O(n).

Comparing the time complexities of Bubble Sort and Bucket Sort, we can see that in general, Bucket Sort has a better time complexity of $O(n)$, which is faster than Bubble Sort's worst-case time complexity of $O(n^2)$.

Therefore, for a large enough array, Bucket Sort would be expected to sort the given array A faster than Bubble Sort.

Question 4.4

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

To determine whether Counting Sort or Merge Sort is faster for sorting the given array A, we need to analyze the time complexity of each sorting algorithm.

1. Counting Sort:
   Counting Sort is an efficient sorting algorithm that works well when the range of input elements is small. It creates a count array to store the count of each unique element in the input array. Then, it modifies the count array to calculate the position of each element in the sorted output. Finally, it builds the sorted array by placing each element in its correct position based on the count array.

The time complexity of Counting Sort is O(n + k), where n is the number of elements in the array and k is the range of input elements. In this case, the range is 10 (digits 0-9), so the time complexity becomes O(n + 10), which simplifies to O(n).

1. Merge Sort:
   Merge Sort is a divide-and-conquer algorithm that divides the array into two halves, recursively sorts each half, and then merges the sorted halves to produce a sorted array.

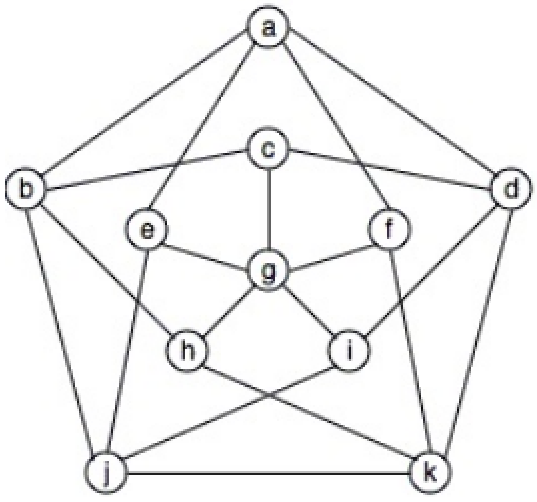
The time complexity of Merge Sort is O(n log n), where n is the number of elements in the array. It has a consistent time complexity regardless of the range or distribution of the input elements.

Comparing the time complexities of Counting Sort and Merge Sort, we can see that Counting Sort has a linear time complexity of O(n), which is faster than Merge Sort's time complexity of O(n log n).

Therefore, for sorting an array of randomly chosen digits between 0 and 9, Counting Sort would be expected to sort the given array A faster than Merge Sort.

Q5.1

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.

g→ f,c,e,h,i

f→ a,k,g

c→b,d,g

e→a,j,g

h→b,k,g

i→d,j,g

a→b,d,e,f

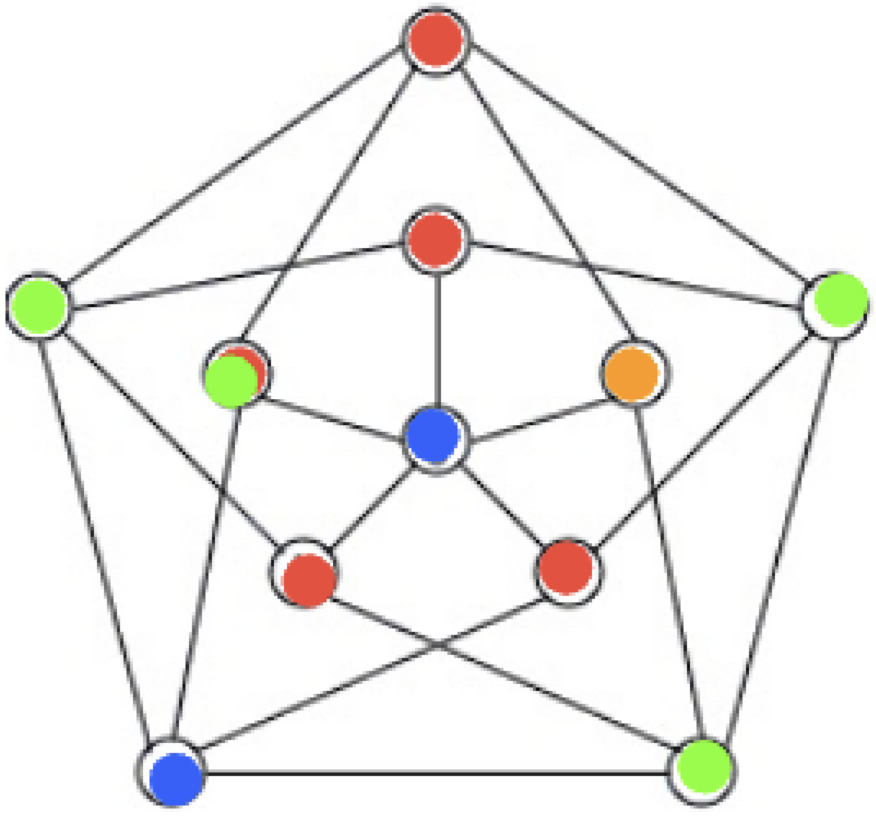
b→a,j,c,h

j→e,i,b,k

k→h,d,f,j

d→c,i,k,a

The conditions that no adjacent c



Q5.2

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.

The algorithm for graph coloring with the minimum number of colors can be summarized as follows:

1. Create a color array `color[]` and initialize it with `1` to indicate that no color has been assigned to any vertex yet.

2. Assign the first vertex a color `0` and mark it as colored.
3. For each vertex `v` from the second vertex to the last vertex:
    - Reset the available colors array `available_colors[]` to indicate all colors are available.
    - For each adjacent vertex `u` of `v`, if `u` is already colored, mark the color `color[u]` as unavailable in `available_colors[]`.
    - Select the smallest available color from `available_colors[]` and assign it to vertex `v`.
4. Find the maximum color assigned, which represents the minimum number of colors used in the graph.
5. Output the minimum number of colors used and the color assigned to each vertex.

The algorithm uses a greedy approach, where at each step, it chooses the smallest available color to color the current vertex while considering the neighboring vertices. By iterating through the vertices and considering the neighboring colored vertices, it ensures that adjacent vertices do not have the same color.

The implementation provided earlier follows this algorithm and demonstrates how it can be coded in Python.

Here is the code:

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0] * self.V for _ in range(self.V)]

    def is_safe(self, v, color, c):
        for i in range(self.V):
            if self.graph[v][i] == 1 and color[i] == c:
                return False
        return True

    def graph_coloring(self):
        color = [-1] * self.V
        color[0] = 0

        available_colors = [True] * self.V
        available_colors[0] = False

        for v in range(1, self.V):
            for u in range(self.V):
                if self.graph[v][u] == 1 and color[u] != -1:
                    available_colors[color[u]] = False

            c = 0
            while not available_colors[c]:
                c += 1

            color[v] = c
            available_colors = [True] * self.V

        num_colors = max(color) + 1

        print("Minimum number of colors required:", num_colors)
        print("Vertex colors:")
        for vertex, col in enumerate(color):
            print(f"Vertex {vertex} --> Color {col}")

        return num_colors


# Example usage
g = Graph(5)
g.graph = [
    [0, 1, 1, 1, 0],
    [1, 0, 1, 0, 1],
    [1, 1, 0, 1, 0],
    [1, 0, 1, 0, 1],
    [0, 1, 0, 1, 0]
]
g.graph_coloring()
```

The time complexity of this algorithm is as follows:

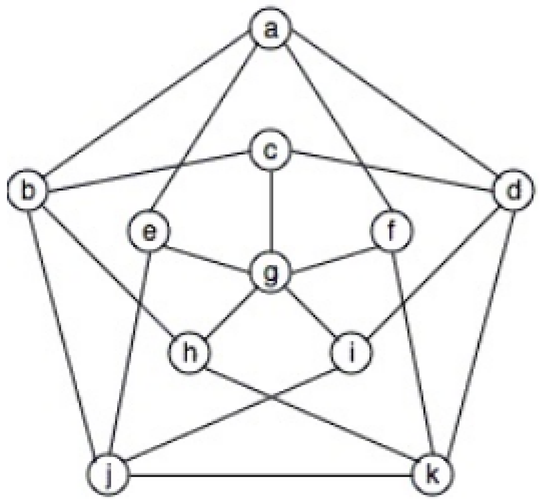
1. Constructing the graph: $O(V^2)$, where V is the number of vertices. This step involves initializing the adjacency matrix.
2. Coloring the graph: $O(V^2)$, where V is the number of vertices. This step involves iterating over all vertices and their adjacent vertices.
   - For each vertex, the algorithm iterates through all vertices to check adjacent vertices in the adjacency matrix, resulting in O(V) iterations.
   - For each iteration, the algorithm performs a while loop that goes through all available colors, resulting in O(V) iterations in the worst case.
   - Overall, the time complexity of this step is $O(V^2)$.

Therefore, the overall time complexity of the algorithm is $O(V^2)$.

It's important to note that this algorithm provides a heuristic solution, which may not always result in the optimal coloring in terms of the minimum number of colors used. Finding the exact minimum number of colors required for graph coloring is an NP-hard problem and typically requires more advanced algorithms.

Q5.3

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



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

g→ f,c,e,h,i

f→ a,k,g

c→b,d,g

e→a,j,g

h→b,k,g

i→d,j,g

a→b,d,e,f

b→a,j,c,h

j→e,i,b,k

k→h,d,f,j

d→c,i,k,a

traversing vertex by the above order and filling the minimum colour that is not used in the adjacent vertexes

lets assign a colour red to g which is kind of first colour which can be treated as colour 0

then go to f as a and k are not coloured and only g is coloured use a new colour blue can be treated as colour 1

then go to c as b and d are not coloured and only g is coloured use a new colour blue can be treated as colour 1

then go to e as a and j are not coloured and only g is coloured use a new colour blue can be treated as colour 1

then go to h as b and k are not coloured and only g is coloured use a new colour blue can be treated as colour 1

then go to i as d and j are not coloured and only g is coloured use a new colour blue can be treated as colour 1

then go to a as the only coloured one is e instead of blue colour we could give the colour of g which is not adjacent which is red can be treated as colour 0

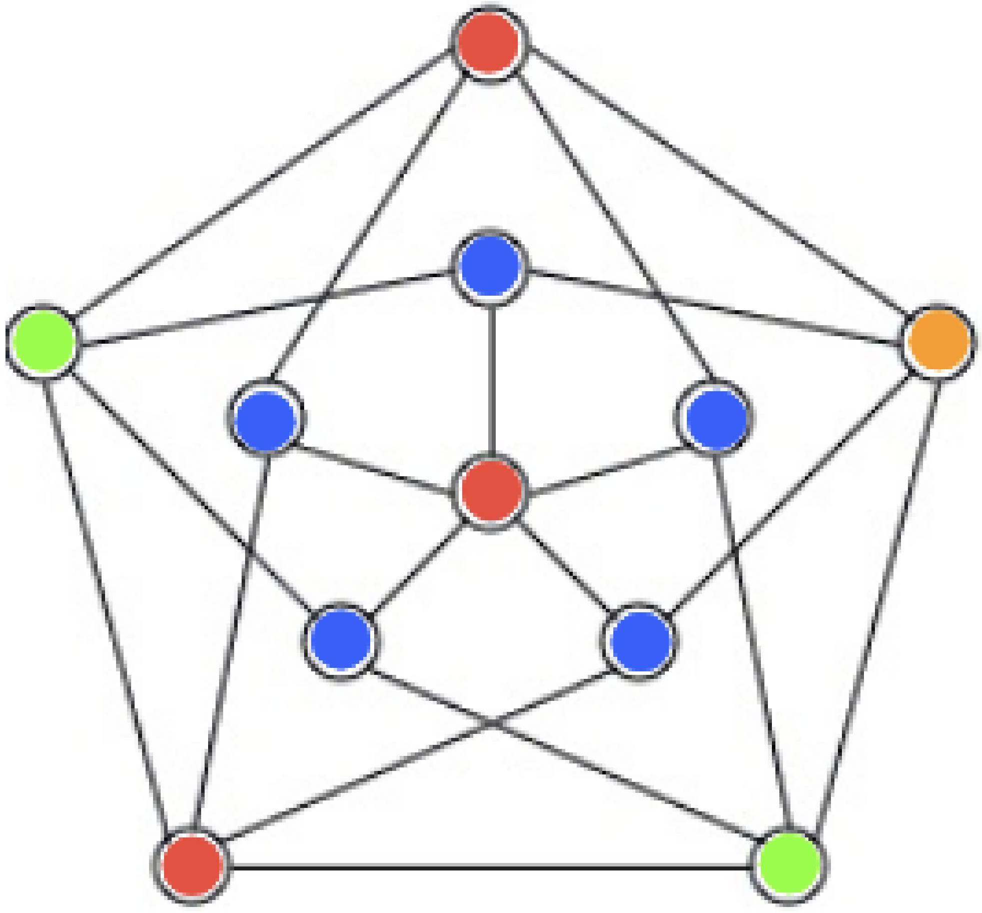
then go to b the adjacent nodes are a,j,c,h and a is coloured with red,c and h is coloured with blue so we give a new colour green can be treated as colour 2

then go to j the adjacent nodes are e,j,b,k and we know e,i are colured blue and b as green so we could assign red colour to the e can be treated as colour 0

then go to k the adjacent nodes are h,d,f,j and we know h,f are coloured in blue d is and j is coloured in red so we can give you green can be treated as colour 2

then go to d the adjacent nodes are c,i,k,a and we know c and i are coloured in blue a in red and k in green so we have to use a new colour orange can be treated as colour 3

so by this we used X(G)=4 as we used colours red, blue, green, orange



Q5.4

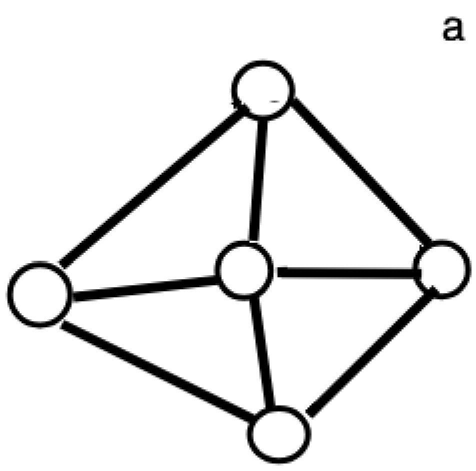
For any graph G, does your algorithm always use exactly χ(G) colours? If so, explain why. If not, provide a graph G for which your algorithm requires more than χ(G) colours.

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.

No the greedy algorithm for coloring the vertices of a graph does not always use exactly X(g) colors.
In fact, there exists graphs for which the greedy algorithm requires more than X(g)

colors. This is
an example of a graph that uses greedy algorithm requires more than X(g) colors.

Consider a graph with 5 vertices.
While using the greedy algorithm for this graph, there will be a time where 2 vertices are uncolored



and adjacent to the same vertex. Here, instead of using 3 colors, we will have to use 4. This shows that the greedy algorithm for coloring the vertices of a graph does not always use exactly X(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 and means there does not currently exists an efficient algorithm for solving the problem). However, determining whether an arbitrary graph has chromatic number 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

Algorithm:

1. For each vertex V in G, mark it with a first color.

2. For each uncolored vertex V in G, mark it with a second color.
3. For each edge (U,V) in G, check if U and V are the same color. If they are the same, return
FALSE.
4. If all the edges in G have vertices with different colors, return TRUE.
The correctness of this algorithm is tested based on what value is returned. If FALSE is returned,
it means that there is atleast one edge with vertices having the same color which means that the
graph cannot be colored with 2 colors. If TRUE is returned, we can conclude the opposite. The
algorithm performs a linear scan of all vertices and edges in the graph, so the runtime is O(|V |+|E|)
where |V | and |E| are the number of vertices and edges in the graph respectively.

The time complexity of the algorithm is O(|V| + |E|), where |V| is the number of vertices and |E| is the number of edges in the graph G.

Explanation:

1. The algorithm starts by marking the first vertex with the first color.
2. It then proceeds to color the remaining uncolored vertices with the second color.
3. Next, it checks each edge in the graph to ensure that the vertices connected by the edge have different colors.
4. If any edge is found where the vertices have the same color, the algorithm immediately returns FALSE, indicating that the graph cannot be colored with only two colors.
5. If all the edges are checked and none of them have vertices with the same color, the algorithm returns TRUE, indicating that the graph can be colored with only two colors.

The algorithm guarantees the correct output because it checks all the edges in the graph to ensure that no two connected vertices have the same color. If there exists an edge with vertices of the same color, it means that the graph cannot be colored with only two colors, which is correctly detected by the algorithm.

The runtime of the algorithm is determined by the number of vertices and edges in the graph. It performs a linear scan of all the vertices and edges, resulting in a time complexity of $O(|V| + |E|)$. This time complexity is efficient for most practical graphs, making the algorithm suitable for determining if $\chi(G)$ is 2 or not.