Algorithms: Synthesis #1

Tanvi Magdum
magdum.t@northeastern.edu

March 6, 2023

Answers

1 Problem 1 Solution :

1.1 Problem 1.1 Solution :

Runtime complexity from best to worst - b). O(1), O(n), O($n \log n$), O(n^2), O(n!), O(2^n), O(n^{1000})

1.2 Problem 1.2 Solution :

For $f(n) = 5n^3 + 10n^2 + 15n + 1000$, $f(n) = O(n^3)$ is true.

1.3 Problem 1.3 Solution :

The value of integer k is 5 if $T(n) = \Theta(n^{5 \log n})$ for recurrence relation $T(n) = 32T(n/2) + n^k$ for all n>1.

1.4 Problem 1.4 Solution :

For A = [2,4,6,8,10,12,14,1,3,5,7,9,11,13,15], the number of swaps made by BUILD-MAX-HEAP(A) is 8. The returned array is [15,10,14,8,7,12,13,1,3,5,4,9,11,2,6].

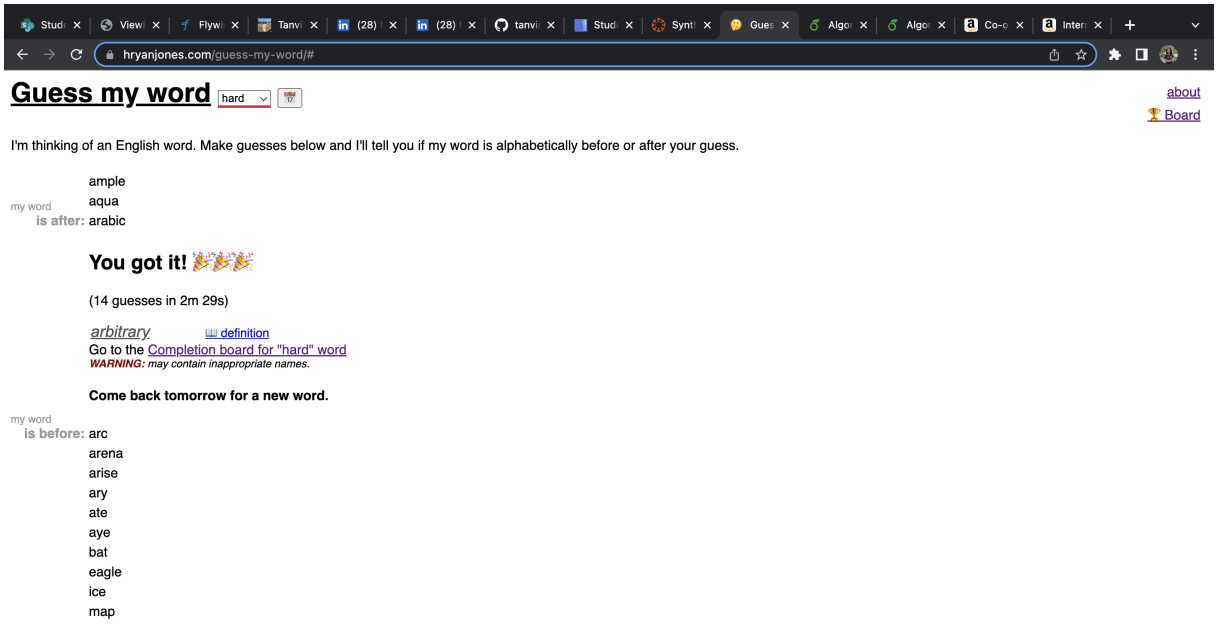
Following are the swaps -
(15,14), (10,4), (7,4), (15,6), (14,6), (15,2), (14,2), (13,2)

1.5 Problem 1.5 Solution :

The maximum total value of your knapsack is 76$ by using D+C+B+(1/5)E = 30+24+15+7 = 76, with total weight of 10 lb.

2 Problem 2 Solution :

2.1 Problem 2.1 Solution :



2.2 Problem 2.2 Solution :

Let us assume that, the dictionary of words is stored in a string array (alphabetically) and the secret word is guaranteed to be found in the stored dictionary. We will use approach as follows -

We will construct a function GuessWord, which takes the guess word as an argument. If the guess word is secret word, then it returns 0. If the secret word lies after the guess word, then the function returns 1 and if the secret word lies before the guess word, then function returns -1. This is similar to binary search approach.

Algorithm-

1. Define function GuessWord(word) which returns int 0, 1 or -1 depending upon whether the guess word is secret word or secret key lies after guess word or before guess word.
2. Set left to 0 and right to $2^k - 1$.
3. While left <= right do the following-
   (a) Set mid to average of left and right.
   (b) Call GuessWord by passing mid as argument.
   (c) If GuessWord returns 0, return the word at position mid as the identified secret word.
   (d) If GuessWord returns 1, set left to mid+1.
   (e) If GuessWord returns -1, set right to mid-1.

Pseudocode-

```
secretWord : the secret word randomly chosen from a dictionary
word : the guessed word
```

```
left : lowest index of the dictionary (set/subset) being searched
right : highest index of the dictinary (set/subset) being searched

GuessWord(word):
    if (word == secretWord):
        return 0
    else if (word > secretWord):
        return 1
    else if (word < secretWord):
        return -1

Find_Secret_Word(dictionary, k):
    left = 0
    right = 2^k - 1
    while left <= right:
        mid = (left + right)/2
        result = GuessWord(dictionary[mid])
        if (result == 0):
            return dictionary[mid]
        else if (result == 1):
            left = mid + 1
        else if (result == -1):
            right = mid - 1
```

We have assumed that the secret key is guaranteed to be found in the dictionary. If this assumption is excluded, then the function Find_Secret_Word has to return something ("Incorrect secret key" or "Null") so that user knows that secret key is not present in the given dictionary set.

The above algorithm is binary search algorithm, which operates on the assumption that the secret word is randomly chosen from a dictionary set containing exactly $2^k - 1$ words. Due to this consideration, the dictionary can be split into two halves at each step and eliminate half of the remaining words based upon the guess word. This guarantees that the secret word can be identified in atmost k steps as each step will reduce the size of the dictionary set by half. In short, the number of guesses required is atmost the number of times we can divide the dictionary set by 2 until we reach a single word set, that is, k times.

The runtime complexity of this algorithm is same as binary search algorithm, $O(\log n)$ in the worst case, where n is the number of words in the dictionary. Here, $n = 2^k - 1$. So,

$O(\log n) = O(\log 2^k) = O(k \log 2) = O(k)$

Thus, the algorithm ensures that the secret word can be identifies in atmost k guesses with runtime complexity of O(k).

2.3 Problem 2.3 Solution :

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, where $n = 2^k - 1$.

Suppose that we have a dictionary and we need to find our secret key in it. According to algorithm, first guess would be the word exactly in middle of the dictionary set. If that guess is correct, then we are done. But if it is wrong, then depending upon if the secret key lies after or before the middle word, we have to eliminate half of the dictionary, and repeat the same process on remaining half. In every step, we are dividing the set by 2, repeating process on one half and discarding other half. We are continuously narrowing down our search and after k steps, we will have made k guesses. Thus, the maximum number of guesses will be k, that is $T(n) = k$.

For each division T(n) will become T(n/2), T(n/4) and so on. So, the recurrence relation we obtain is

$T(n) = T(\frac{n}{2}) + 1$, where 1 is the current guess.

Now, applying Master's theorem on this recurrence,

Comparing this relation with, $T(n) = aT(\frac{n}{b}) + f(n)$, a = 1, b = 2 and f(n) = 1

$\therefore n^{\log_b a} = n^{\log_2 1} = n^0 = 1$

$f(n) = n^{\log_b a} = 1$, this is Case 2 of Master's theorem.

Hence, according to case 2, $T(n) = \Theta(1. \log n)$, that is, $T(n) = \Theta(\log n)$

Therefore, the maximum number of guesses required to identify a secret word in a dictionary of size n is logarithmic in n.

2.4 Problem 2.4 Solution :

We assume that there are total 267,751 words in dictionary and each of those words is equally likely to be chosen. Then probability of guessing the correct word in one guess is (1/267751). It means, in best case it is possible we require 1 guess to find correct word, or in worst case we might require 267751 guesses to find the correct word. If we guess the word incorrectly, then we have to spend an extra dollar to get a chance to guess once again, and this happens until we guess the word correctly.

But, if we use binary search approach to guess the word, we can narrow down our search and we can guess the correct word in less number of guesses compared to the previous approach. The program tells us whether the word lies before or after the guessed word. This will help eliminate all the words before or after the guessed word. As stated before, if the size of set, $n = 2^k - 1$, then we require at most k guesses to find the correct word, assuming that the word is present in the given set of dictionary.

$267751 = 2^k - 1$

$267752 = 2^k$

$k = \log_2 267752$

$k \approx 18$

We require at most 18 guesses in worst scenario to find the correct word in the dictionary using binary search method.

Now, we need to find out if we have a chance to win or lose the game. Since we have 15$, we have two possible outcomes -

1. We make 18 guesses to find the word, resulting in a loss of 3 extra dollars.
2. We make 15 or fewer guesses to find the word, resulting in retaining the money or exhausting it completely without any loss.

To calculate the probability of each outcome, we need to know the probability of finding the correct word in 15 or fewer guesses. We have a 1/267,751 chance of guessing correctly in a single try. Let p = 1/267,751.

For P(correct guess in 15 or fewer tries) = 1 - P(correct guess in 16 tries) - P(correct guess in 17 tries) - P(correct guess in 18 tries)

Now, P(correct guess in 16 tries) $= \binom{15}{1} * p^1 * (1-p)^{14}$

$$= 15 * \frac{1}{267751} * (\frac{267750}{267751})^{14}$$

$$\approx 0.0048$$

For, P(correct guess in 17 tries) $= \binom{15}{2} * p^2 * (1-p)^{13}$

$$= 105 * (\tfrac{1}{267751})^2 * (\tfrac{267750}{267751})^{13}$$

$$\approx 0.0032$$

For, P(correct guess in 18 tries) $= \binom{15}{3} * p^3 * (1-p)^{12}$

$$= 455 * (\tfrac{1}{267751})^3 * (\tfrac{267750}{267751})^{12}$$

$$\approx 0.0011$$

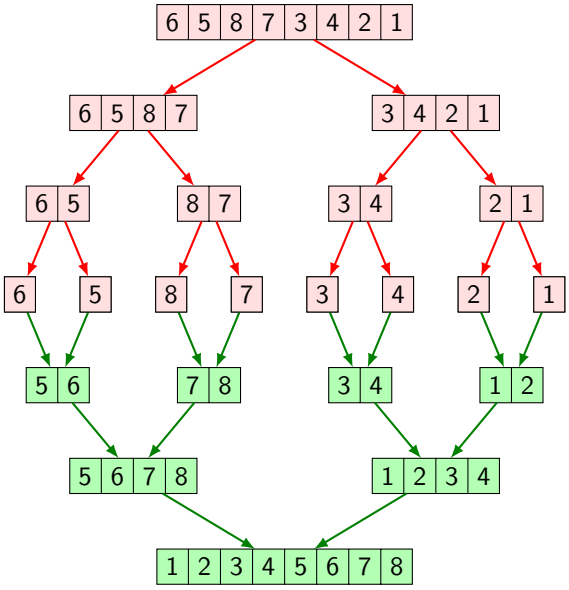
$\therefore$ P(correct guess in 15 or fewer tries) = 1 - 0.0048 - 0.0032 - 0.0011 = 0.9910 = 99.10%

So the probability of guessing the word in 15 or fewer tries is approximately 0.9910 or 99.1%. Thus, the probability of retaining money is more than probability of losing money by making 15+ guesses, hence chances of winning are more.

3 Problem 3 Solution :

3.1 Problem 3.1 Solution :

We have A=[6,5,8,7,3,4,2,1]. To perform Merge Sort algorithm on this array, we need to split this array into 2 halves for 2nd pink step. Then again for 3rd pink step, break each of the 2 sub-arrays into 2 halves to get total 4 sub-arrays with 2 elements for each. Breakdown these 4 sub-arrays into 8 single element arrays to get 4th pink step.



From the depicted breakdown, total 4 comparisons would occur to get 1st green level, that is between 6-5, 8-7, 3-4, 2-1. The sorted pairs form 4 different arrays on this level.

As we go down to get 2nd green level, we merge sub-arrays into pairs, while maintaining the sorted order of elements. We merge arrays [5,6] and [7,8] in order to get [5,6,7,8] and [3,4] and [1,2] to get [1,2,3,4]. We make a comparison between first element of each array and select the smaller element to add to merged array. We repeat this process until all elements are merged into sub-array. So, comparisons occur between 5-7, 6-7, 3-1, 3-2. The sorted pairs form 2 different sorted arrays on 2nd green level.

To get final sorted whole array, we have to merge 2 sub-arrays. We compare smallest element of first element of first array [5,6,7,8] with first element of second array [1,2,3,4]. Add smallest into final array and repeat this until all elements are merged into one array. Comparisons occur between 5-1, 5-2, 5-3, 5-4. After comparing this and adding 1,2,3,4 to new array, 5,6,7,8 will simply append into final array.

At the end, we see that total 12 comparisons take place, 4 in last pink step, 4 in 1st green step and

4 in 2nd green step. After these comparisons we get a sorted array [1,2,3,4,5,6,7,8] using Merge Sort algorithm.

3.2 Problem 3.2 Solution :

In Merge Sort algorithm, the sub-array merging decides the number of comparisons made to form a final merged sorted array. When we merge two sub-arrays of size n/2, we need to compare their elements to determine their relative order. As there are n elements in the combined sub-arrays, we make total n number of comparisons to merge them. Given any array A with n elements, we first divide it into 2 halves each containing n/2 elements and again divide each half into more 2 halves. The division and merging of sub-arrays occurs recursively, that is, we can apply Merge Sort algorithm recursively to each sub-array to obtain the minimum number of comparisons required to sort. We need M(n/2) comparisons to sort each sub-array and once arrays are sorted we merge them by comparing n times. Hence total number of comparisons can be represented as -

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

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

This applies only when n is an even number. If n is even the recursion will take place and we can substitute values in the above equation to get minimum number of comparisons. Suppose n=8, then-

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

M(8) = 2M(4) + 4

M(8) = 2[2M(2) + 2] + 4

M(8) = 2[2[2M(1) + 1] + 2] + 4

If n=1, then there would be no comparison so, M(1) = 0.

M(8) = 2[2[2*0 + 1] + 2] + 4

M(8) = 2[2 + 2] + 4 = 8 + 4 = 12

In this example, total 12 minimum comparisons will be required to sort an array of 8 elements using Merge Sort.

3.3 Problem 3.3 Solution :

To prove $M(n) = (n \log n)/2$, for n being a power of 2, we can use induction method.

Base case - When n = 2, $M(2) = (2 \log 2)/2 = 1$

Inductive hypothesis - Assume that $M(k) = (k \log k)/2$ for all powers of 2 less than or equal to k.

Inductive step - Prove $M(2k) = (2k \log 2k)/2$ for all powers of 2.

We know, $M(n) = 2M(n/2) + n/2$

Substitute n with 2k,

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

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

Now, substitute the inductive hypothesis, that is, $M(k) = (k \log k)/2$

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

$M(2k) = k \log k + k$

multiply by $\log 2$, since it's value is 1 and will not affect the equation

$M(2k) = k[\log k * \log 2 + \log 2]$

Now for $\log k * \log 2$, put $\log 2 = 1$

$M(2k) = k[\log k + \log 2]$

$M(2k) = k[\log 2k]$ according to logarithmic rules

multiply and divide by 2

$M(2k) = 2k(\log 2k)/2$

Thus, we have proved the induction $M(2k) = (2k \log 2k)/2$ for all powers of 2.

Now re-substitute 2k by n

$M(n) = (n \log n)/2$

Hence, we have proved $M(n) = (n \log n)/2$ for all powers of 2.

3.4 Problem 3.4 Solution :

To determine the probability that exactly 12 comparisons are required, we need to find conditions under which 12 comparisons occur for Merge sorting an array of 8 elements. We know that merge sort requires exactly n-1 comparisons to sort an array of n elements.

We need to find the number of permutations of A for which Merge Sort takes 12 comparisons. Consider the number of permutations for which first comparison is made between 2 elements that are not adjacent to each other in sorted array. There are such 7 pairs of non-adjacent : (1,3), (1,4), (1,5), (1,6), (1,7), (1,8) and (2,4). We have total pairs as 28 (adjacent + non-adjacent). Hence, the probability that the first comparison is made between 2 non-adjacent elements is 7/28 = 1/4.

Once we have chosen 2 non-adjacent elements for first comparison, we can chose other 2 non-adjacent elements in 4 ways. From previous comparison, it is clear that one of the 2 elements have a fixed position. So, now remaining 3 elements can also be arranged in any of 3!=6 ways. Once those two elements are chosen, the remaining 4 elements can be arranged in any order, giving us 6! possibilities. Hence the total number of permutations for A for which exactly 12 comparisons are required by Merge Sort is-

(7/28) * 2 * 4 * 3! * (4 * 6!) = 34560

The total number of permutations for A is 8! = 40320. Therefore, the probability that exactly 12 comparisons are required by Merge Sort to sort the input array A is -

P(12 comparisons) = 34560/40320 = 0.8571 = 85.71%

Therefore, the probability that Merge Sort requires exactly 12 comparisons is 85.71%.

4 Problem 4 Solution :

4.1 Problem 4.1 Solution :

The Selection sort and Insertion sort both have a worst-time complexity of $O(n^2)$. This means the time it takes to sort an array with size n, increases quadratically as n increases. Selection sort works by repetitively finding the minimum from the unsorted part of array and swapping it with the first element of unsorted array part. This process continues until the whole array is sorted. Insertion sort works by sorting the elements one by one and inserting each new element in its proper place in the sorted part of the array. This continues until all elements are sorted.

Algorithm for Selection sort-

1. Start with unsorted array A=[5,5,5,5,5,...,5].
2. Find minimum element in unsorted part of array A and swap it with the first element.
3. Move to next element and again swap.
4. Continue until the whole array is sorted.

Algorithm for Insertion sort-

1. Start with unsorted array A=[5,5,5,5,5,...,5].
2. Set the first element as sorted part of array.
3. For each unsorted element, compare it to elements in sorted part and then insert it into correct position.
4. Continue until the whole array is sorted.

For given array, where all elements are equal to 5, Selection sort would still need to compare every element with other elements of array to find the minimum, though all elements are same. For array of size n, there would be (n-1) swaps for Selection sort. On the other hand, in Insertion sort, since all the elements are same and every new element will be equal to previous sorted element, there would be no swapping. It would just compare each element with the sorted part of the array and array is already sorted. This sort will go through array once and that is it.

Evidently, the Insertion sort is faster for sorting the array A=[5,5,5,5,5,...,5], as it take just one go through for given array and Selection sort takes (n-1) swaps .

4.2 Problem 4.2 Solution :

Quicksort is a divide-and-conquer algorithm that sorts an array by partitioning it into 2 sub-arrays around a pivot element, such that all elements in left sub-array are less than or equal to pivot and all elements in right sub-array are greater than pivot. The algorithm recursively sorts left and right arrays.

Algorithm for rightmost Quicksort -

1. Choose the rightmost element of array as the pivot.
2. Partition the array around the pivot such that all elements those are smaller than pivot belong to left sub-array and greater than pivot belong to right sub-array.
3. Sort both left and right sub-arrays recursively.
4. Combine the sorted sub-arrays to produce final sorted array.

When we choose the rightmost element as pivot, the right sub-array is always going to remain empty and left sub-array will contain all the elements of an array or sub-array. For given example, the rightmost element 1 would be the first pivot. After partitioning, the array would look like [1,n-1,n-2,. . . ,3,2,n], since the pivot will acquire its desired position. So, now 1 is sorted, we don't have any left sub-array and for remaining right sub-array, n is the pivot. We know that n too is at the right place but still the swapping would take place resulting in no difference in array. After this iteration, 2 will become the pivot and it will swap with n-1. Again for next iteration n-1 is pivot and yet it is at correct place swapping occurs. We can figure out that half of the elements of array get sorted automatically due to previous pivot yet the swapping occurs unnecessarily. The time complexity in this case for Quicksort is $O(n^2)$.

Let us look at Heapsort. It sorts an array by first building a heap, which is a binary tree satisfying heap properties. The algorithm extracts the max element repetitively from heap and places it at the end of the sorted array until the whole array is sorted.

Algorithm for Heapsort -

1. Build a max heap for array A.
2. Extract max element from heap and place it at the end of the array.
3. Heapify the remaining elements to satisfy heap properties.
4. Repeat previous 2 steps until the entire array is sorted.

In this case, building a max heap takes $O(n)$ time. Here, A[0] will be the root of the heap, as it is the largest element. Hence, the first step of algorithm can be skipped and we can proceed to further steps. As the largest element of array is always at the root, we can extract the max element and place it at the end of the array in constant time. Heapifying the rest of the elements will take $O(\log n)$ time. Hence, the total time complexity of this algorithm is $O(n \log n)$. Also, the worst case complexity of Heapsort is always $O(n \log n)$.

Thus, we can say that for this case, Heapsort would be a better choice over Quicksort, as the former is faster and more efficient than the latter one.

4.3 Problem 4.3 Solution :

Bubble sort algorithm -

1. Start with the first element in array.
2. Compare the first element with the next element in array.
3. If the first element is greater than the next element then swap both elements.
4. Move to the next pair of elements in array and repeat steps 2 and 3.
5. Continue this process until whole array is sorted.

Using this algorithm, we can sort the array as-

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

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

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

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

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

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

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

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

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

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

Bubble sort has worst-case time complexity of $O(n^2)$ because it contains a nested loop which iterates n*n number of times to sort the array, that means, this algorithm can take a long time to sort an array of size n, especially if the array is partially sorted.

Bucket sort algorithm -

1. Create n empty buckets based on some criteria.
2. Iterate through the array and add each element to its corresponding bucket depending upon its value.

3. Sort each bucket using any efficient algorithm, we will use Insertion sort here since it is considered to be an efficient algorithm to sort an an array with less elements.

4. Concatenate the sorted buckets to form a single sorted array.

Using this algorithm we can sort the array as -

Create 4 buckets, Bucket 1 containing elements with values 1-3, Bucket 2 with 4-6, Bucket 3 with 7-9 and Bucket 4 with 10-12.

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

Bucket 1 : [3,1,2]

Bucket 2 : [5,5,6]

Bucket 3 : [7,9,8,8]

Bucket 4 : [10,12]

Sorting these buckets individually, we get -

Bucket 1 : [1,2,3]

Bucket 2 : [5,5,6]

Bucket 3 : [7,8,8,9]

Bucket 4 : [10,12]

Just concatenate all buckets to get a sorted array - [1,2,3,5,5,6,7,8,8,9,10,12].

Bucket sort has worst case complexity of $O(n+k)$, where k is the number of buckets used in this case. We know that the elements of an array lie between 1 to n, we have an understanding of the input, so Bucket sort fits perfectly in this case. Since we know the input lies between 1 to n and we expect that the distribution is fairly uniform based on the given input array, we can design buckets so that k is smaller than n. Even if we consider k=n, that is, create total n=12 buckets, then also, worst case complexity in this case turns out to be $O(n)$. Thus, Bucket sort can be much faster than Bubble sort.

4.4 Problem 4.4 Solution :

Counting sort algorithm -

1. Create an auxiliary array of size 10 to maintain the count. Initialize it to 0.

2. Iterate over A and increment the corresponding element of C for each digit in A.

3. Iterate over C and update each element to be sum of itself and the previous element. For example, if C[3] = 4 and C[4] = 2, then modify C[4] = 6.

4. Create a new array B of size n.

5. Iterate over A from right to left, placing each element in its sorted position in B using the corresponding value in C as the index.

6. Return B as the sorted array.

Counting sort has a time complexity of $O(n+k)$, where n is the number of elements in array and k, the range of values that elements can take. Counting sort is efficient when range of values is smaller compared to number of elements in the array. In this case, we know that elements lie between a range of 0-9 and this range is small.

Merge sort algorithm -

1. Divide the array into two equal halves.
2. Sort the left half recursively using Merge sort.
3. Sort the right half recursively using Merge sort.
4. Merge the 2 halves into a single sorted array.
   (a) Create an auxiliary array B.
   (b) Initialize 2 variables i and j to 0, one for each half of the array.
   (c) Compare the current elements of both halves and put smaller element into B.
   (d) If one of the halves are done, copy the remaining part of other half into B.
   (e) Repeat 3-4 until all elements are placed into B.
5. Return the sorted array B.

Merge sort has time complexity of $O(n \log n)$ in the worst case, and this sort is considered to be the best sorting algorithm. But in our case, we know that we are operating on a certain range of values and merge sort performs requires the same time regardless the input range. Merge sort would have been an efficient algorithm here if the number of elements was large or the array was completely unsorted. Though here the array is completely unsorted, the number of elements n=12 is very small, and we have an option to go for better algorithms than merge sort.

Counting sort is more efficient than merge sort in this case because we are known with the user input and the time complexity is linear.

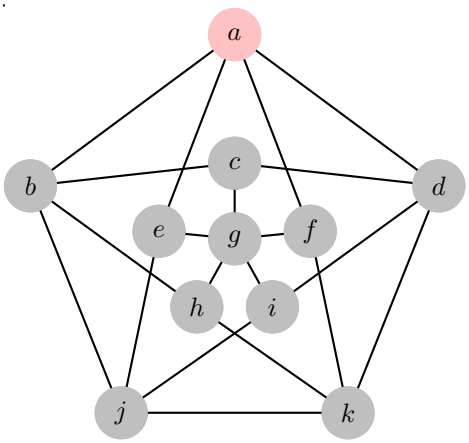
5 Problem 5 Solution :

5.1 Problem 5.1 Solution :

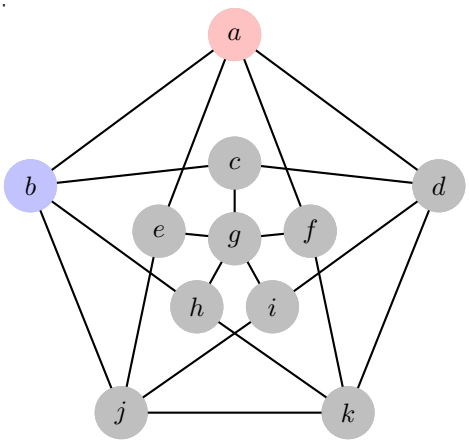
To show that $\chi(G) = 4$ in the given graph, we can use the greedy algorithm. We will start by an arbitrary vertex, suppose A and assign it the first color, suppose "red". We then proceed with coloring other vertices with the available color that is not similar to any other adjacent vertex colors. Since, each vertex has at most (k-1) neighbors, this algorithm is guaranteed to produce a valid k-coloring if one exists. Hence, G is k-colorable.

The given graph has total 11 vertices and 20 edges. To determine the chromatic number of this graph, we need to find the smallest number of colors required to color the vertices such that no two adjacent vertices have same color. We have been given that the chromatic number for this graph is $\chi(G) = 4$. We need to prove that the smallest number of colors required is 4.

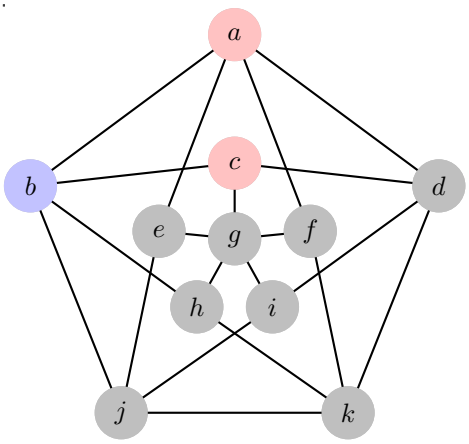
As we mentioned, we will start from vertex a. Let us assume that it is colored by red. Now a is connected to b, d, e, f. So if vertex a is red, then b, d, e, f cannot be red.



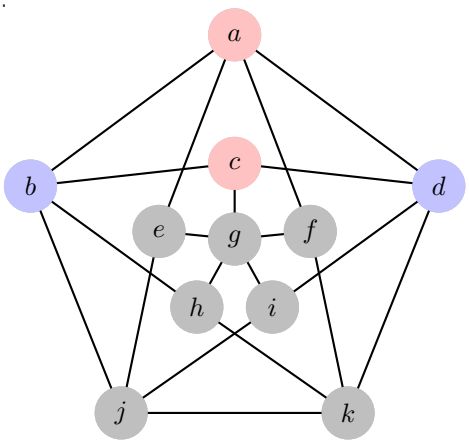
Now suppose since, vertex b cannot be colored red as it lies adjacent to vertex a which is already red, we can use blue color for vertex b. The adjacent vertices for b are vertices a, c, h, j. All of these vertices cannot acquire blue color. Vertex a is already colored red so we need to care about other 3 vertices.



For vertex c, since it is adjacent to vertex b, it cannot be blue. Till now we have used 2 colors red and blue. If blue is out, let us check if we can use red. We have vertex a in red and vertex c is not adjacent to vertex a. So, c can definitely be colored red. We have keep in mind that since c is red now, its adjacent vertices b, d, g cannot be red.

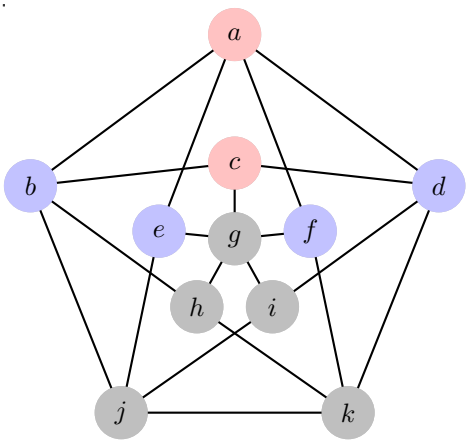


For vertex d, since a and c are its neighbors, it cannot be colored red. Let's check if we can color it blue. Till now we have vertex b in blue color, and d is not connected directly to b. So we can use blue color for vertex d. Vertices c, a, k and i cannot be colored blue as they are d's neighbors.

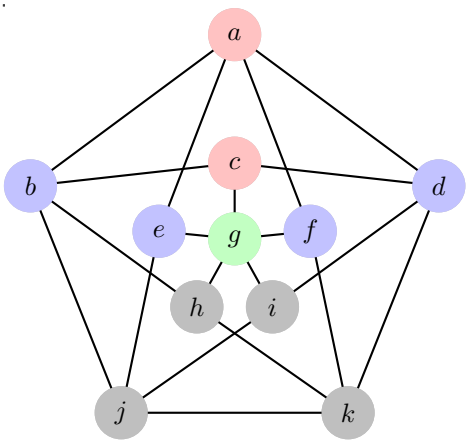


Take vertex e. It has neighbors a, g, j which are not supposed to be colored same as e. Till now, a is

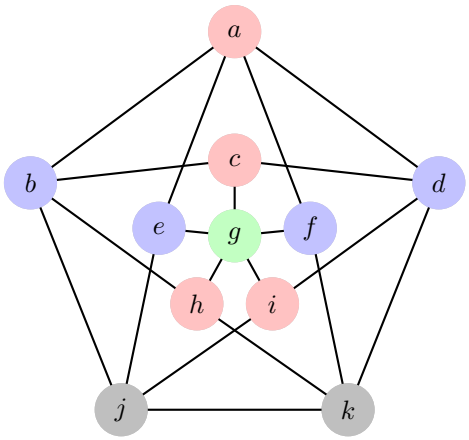
red and e's neighbor so, red is out. Blue fits correctly for e because from the graph, e is not adjacent to b or d which are blue. Similarly for vertex f, blue color can be used as it has neighbors a, g and k and none of them are blue.We cannot use red for it because a is red.



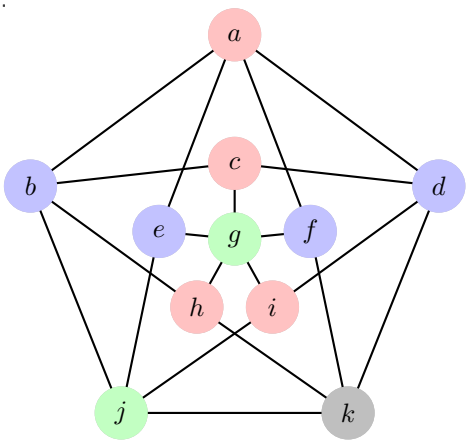
Consider vertex g. It has adjacent vertices as c, e, f, h and i. Out of these, c is red and e and f are blue. We have exhausted the colors we have used till now. We need to use a new color as the already existing colors are occupied by g's adjacent nodes and hence, g cannot use them. We will use green color for g. It is clear that remaining adjacent nodes h and i cannot be green now.



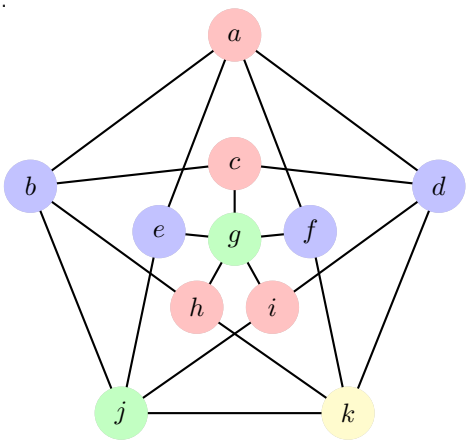
For vertex h, adjacent edges are b,g and k. Vertex b is blue and g is green. Out of the colors we have used till now, we have red color that we can use for h. Let us check if red colored node is connected to h. Till now vertices a and c are red colored and not connected to h. So, we can use red for h. Similarly, for i, adjacent edges are d, g and j. Vertex d is blue and g is green. Try red for i. Vertices a and c which are red are not adjacent to vertex i. Also the newly colored vertex h is not connected to i. So, we can color i to red. Note that we cannot use red for vertices j and k which are uncolored yet.



For vertex j, neighboring vertices are b, e, i and k. Till now b and e are colored blue and i is red. Picking the green color from previously used colors let us check if we can use it. We have node g as green and it is not connected to j. So, yes we can use green for j.



For the last vertex k, the adjacent vertices are j, h, f and d. Vertex j is green, h is red and vertices f and d are blue colored. So, the adjacent edges of k are occupied with all the colors which we have used till now. Hence, we cannot use those colors and need to use a new one. Let's take yellow.



Thus, from the last graph, we see that we have used 4 colors to for all the vertices in the given graph so that adjacent vertices don't have same color. The minimum colors required to satisfy the condition are 4 in this case, otherwise we would have ended up having 2 or more adjacent vertices with same color. Hence, the chromatic number for this graph is $\chi(G) = 4$.

5.2 Problem 5.2 Solution :

The greedy algorithm for vertex coloring is -

1. Sort the vertices in the descending order of their degree. Sort it according to which vertices have more connected vertices to less connected vertices. This is because many times high-degree vertices are likely to have more conflicts with its other vertices.
2. Initialize an empty dictionary "colors". The keys in this dictionary are the vertices of the graph and values are its colors.
3. For each vertex v in the sorted list of vertices, assign the lowest possible color that is not already assigned to its neighbors. If all colors are utilized, assign a new color.
4. Once all vertices are colored, the number of colors used is the chromatic number of the graph.

The algorithm starts by sorting the vertices of the graph in descending order. This is because high degree vertices tend to have many neighbors and by sorting in descending order we would increase the likelihood that each vertex would have neighbors that are already colored. This would allow us to use fewer colors. The algorithm then initializes an empty dictionary called "colors" that keeps track of every vertex-color pair. For each vertex, the algorithm checks colors assigned to its neighbors. It skips those colors and assigns the next available color to the current vertex. If there are no colors assigned to any of the neighbors, the algorithm assign vertex as 0. Using the smallest possible color that has not been yet assigned to its neighbors, the algorithm tries to minimize the number of colors needed to color the whole graph.

Psuedocode :

```
Chromatic_Number():
    sort the vertices in descending order and store in sortedList
    colors = {}
    for vertex v in sortedList:
        available_colors = {}
        for u in G.adjacent(v):
            if colors.containsKey(u):
                available_color.add(colors.get(u))
        col = 0 //col = 0,1,2,3... resemble to col = red, blue, green,...
        while available_color.contains(col):
            col++
        colors.put(v,col)
    return colors
```

The runtime complexity of this algorithm is $O(n^2)$, where n is the number of vertices in the graph. We need to iterate over each vertex, and for each vertex we need to iterate over its neighbor. Since we are sorting the vertices first, the time complexity also depending upon the sorting algorithm used. The main loop iterates through each vertex and its neighbors and for each vertex, it looks up in the dictionary to choose the colors for its neighbors. This can be implemented using a hash table and thus the time required for this operation can be reduced to $O(1)$ in average. In the worst case, each vertex may be connected to every other vertex int he graph, thus each neighbor may have different color for a particular vertex. It would require iterating through all previously used colors to find an available color for current vertex. It might result in worst-case time complexity of $O(n^2)$, where n is number of vertices. In practice the time complexity might be lower depending upon the number of vertices and density in a graph.

5.3 Problem 5.3 Solution :

If we apply the above algorithm to the first problem graph, we can go step by step as follows -

1. Sort the vertices in descending order of their degrees. The vertex with maximum number of adjacent vertices will be the first element of the sorted array and one with least number of adjacent nodes will be last element of array.

For the given graph, the vertex g has maximum, that is, 5 adjacent nodes. Then a, b, d, j, k all have 4 adjacent nodes each and remaining nodes c, e, f, h, i all have 3 adjacent nodes each. The sorted list would be-

```
sortedList = {g, a, b, d, j, k, c, e, f, h, i}
```

2. Initialize an empty dictionary "colors", to store colors assigned to each vertex.

```
colors = new HashMap<>();
```

3. We iterate over each vertex in the sortedList. For each vertex v, we find a set of colors used by its neighbors and use the smallest unused color.

a. For first vertex in the array g, it doesn't have neighbors which are already colored. So, we can assign 0, the first color to it. We are using numbers for simplicity. Instead of numbers, in actual code we can use an array of actual colors as strings or any kind of set storing various colors.

```
available\_colors = {}
col = 0
colors[g] = col
```

After adding this pair of vertex:color to dictionary, it will look like-

```
colors = {g:0}
```

b. For next vertex in the array a, its neighbors are b, d, e, f. None of them have been previously colored. Again assign 0 to vertex a, as it the smallest color that can be assigned.

```
available\_colors = {}
col = 0
colors[a] = col
```

After adding this pair of vertex:color to dictionary, it will look like-

```
colors = {g:0, a:0}
```

c. For next vertex in the array b, its neighbors are a, c, h, j. Now we have a:0. Other adjacent nodes are uncolored. Now since the smallest possible color 0 is used by a, we need to use a different color for b.

```
available\_colors = {0}
col = 1
colors[b] = col
```

After adding this pair of vertex:color to dictionary, it will look like-

```
colors = {g:0, a:0, b:1}
```

d. For next vertex in the array d, its neighbors are a, c, i, k. Now we have a:0. Other adjacent nodes are uncolored. Now since the smallest possible color 0 is used by a, we need to use a different color for d.

```
available\_colors = {0}
col = 1
colors[d] = col
```

After adding this pair of vertex:color to dictionary, it will look like-

```
colors = {g:0, a:0, b:1, d:1}
```

e. For next vertex in the array j, its neighbors are b, e, i, k. Now we have b:1. Other adjacent nodes are uncolored. Now, the smallest possible color is 0 and it is not used by any of the adjacent nodes of j. So, we can use 0 for j.

```
available\_colors = {1}
col = 0
colors[j] = col
```

After adding this pair of vertex:color to dictionary, it will look like-

```
colors = {g:0, a:0, b:1, d:1, j:0}
```

f. For next vertex in the array k, its neighbors are j, h, f, d. Now we have d:1 and j:0. Other adjacent nodes are uncolored. Now since colors 0 and 1 are used by the neighboring vertices, we will have to use a new color for k.

```
available\_colors = {0, 1}
col = 2
colors[k] = col
```

After adding this pair of vertex:color to dictionary, it will look like-

```
colors = {g:0, a:0, b:1, d:1, j:0, k:2}
```

g. For next vertex in the array c, its neighbors are g, b, d. Now we have g:0, d:1 and b:1. The smallest possible color and not used by any of the adjacent nodes of c is 2.

```
available\_colors = {0, 1}
col = 2
colors[c] = col
```

After adding this pair of vertex:color to dictionary, it will look like-

```
colors = {g:0, a:0, b:1, d:1, j:0, k:2, c:2}
```

h. For next vertex in the array e, its neighbors are g, a, j. Now we have g:0, a:0 and j:0. The smallest possible color and not used by any of the adjacent nodes of e is 1.

```
available\_colors = {0}
col = 1
colors[e] = col
```

After adding this pair of vertex:color to dictionary, it will look like-

```
colors = {g:0, a:0, b:1, d:1, j:0, k:2, c:2, e:1}
```

i. For next vertex in the array f, its neighbors are g, a, k. Now we have g:0, a:0 and k:2. The smallest possible color and not used by any of the adjacent nodes of f is 1.

```
available\_colors = {0, 2}
col = 1
colors[f] = col
```

After adding this pair of vertex:color to dictionary, it will look like-

```
colors = {g:0, a:0, b:1, d:1, j:0, k:2, c:2, e:1, f:1}
```

j. For next vertex in the array h, its neighbors are g, b, k. Now we have g:0, b:1 and k:2. Now since all available colors are exhausted, we need to use a different color. The smallest possible color now is 3, which is also different from the colors of neighboring vertices.

```
available\_colors = {0, 1, 2}
col = 3
colors[h] = col
```

After adding this pair of vertex:color to dictionary, it will look like-

```
colors = {g:0, a:0, b:1, d:1, j:0, k:2, c:2, e:1, f:1, h:3}
```

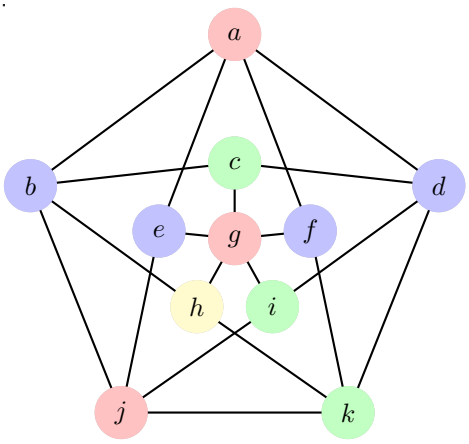
k. For next vertex in the array i, its neighbors are g, d, j. Now we have g:0, d:1 and j:0. The smallest possible color and not used by any of the adjacent nodes of i is 2.

```
available\_colors = {0, 1}
col = 2
colors[i] = col
```

After adding this pair of vertex:color to dictionary, it will look like-

```
colors = {g:0, a:0, b:1, d:1, j:0, k:2, c:2, e:1, f:1, h:3, i:2}
```

4. Finally the colors will be - colors = {g:0, a:0, b:1, d:1, j:0, k:2, c:2, e:1, f:1, h:3, i:2}. Let 0 corresponds to red, 1 to blue, 2 to green and 3 to yellow. The colored graph will look like -



Hence, this graph uses total 4 colors. The chromatic number for this graph is 4. We see in 5.1 we got a graph where 4 vertices are red, 4 are blue, 2 are green and 1 is yellow and in this graph we have 3 reds, 4 blues, 3 greens and 1 yellow. This change can occur due to the algorithm we are using for calculation. In this algorithm, we sorted vertices in descending order of their degrees, hence it affected this change. But anyhow, the maximum number of colors used will remain constant for a particular graph.

5.4 Problem 5.4 Solution :

For any graph G, the above algorithm always use exactly $\chi(G)$ colours. Let us breakdown the algorithm step by step.

Step 1: Sorting the vertices in descending order of their degrees.

We are sorting the vertices in descending order of degrees because, the vertices with more number of adjacent vertices tend to have more conflicts with their neighbors, making them difficult to color. Also, by coloring these vertices first, we increase the likelihood of getting colored neighboring vertices for other vertices. This would allow us to use fewer colors.

Step 2: Initialize an empty dictionary "colors".

We use a dictionary to track vertex:color pairs. The keys are vertices and values are their respective colors.

Step 3: For each vertex v in the sorted list of vertices, assign the lowest possible color that is not already assigned to its neighbors. If all colors are utilized, assign a new color.

This is the crucial step of the algorithm, where we apply greedy strategy. For each vertex v, we consider the colors assigned to its neighbors and store those colors in a set or an array. Then we choose the lowest possible color that is not used by any of its adjacent nodes. If all colors are exhausted, we use a new color. We want to minimize the number of colors used, so we try to reuse colors wherever possible. By selecting the lowest possible color, we avoid the chance of running into conflicts for other vertices.

Step 4: Once all vertices are colored, the number of colors used is the chromatic number of the graph.

The chromatic number of graphs will be accessed by counting the unique color values in the colors dictionary.

Now, to prove that this algorithm gives the correct chromatic number always, we need to show that it always assigns a valid color to each vertex and always uses a minimum number of colors necessary to color a graph.

Case 1: This algorithm always assigns a valid color to each vertex.

Suppose that the algorithm is assigning colors to a vertex v. Let available_colors be the set of colors assigned to the neighbors of v. Then, the algorithm assign a lowest possible color that doesn't belong to available_colors set. As the degree of v is finite, the length of available_colors is also finite. Hence, the algorithm is guaranteed to find a valid color for v.

Case 2: This algorithm always uses the minimum number of colors necessary to color the graph.

Suppose the algorithm colors the graph with k colors, where k is not the chromatic number of the graph. Then there must exist at least 2 adjacent vertices with same color. Let u and v are adjacent vertices. Let available_colors_u and available_colors_v be two sets of colors assigned to neighbors of u and v. Since u and v are adjacent they cannot have same color. The union of both the sets contains atleast k colors. We know that algorithm only used k colors. Hence, there must be one color in the the union set, which was not used by the algorithm. Suppose w is a vertex adjacent to both u and v and it has this unused color. Then, the algorithm would have assigned this color to w, instead of the color it actually used for w. This leads to a contradiction. Therefore, the algorithm must use at least k colors, the chromatic number of the graph to color it.

Thus, we have shown that, for any graph G, this algorithm always use exactly $\chi(G)$ colours.

5.5 Problem 5.5 Solution :

Algorithm :

1. Select a vertex v in the graph G.
2. Color v with color 0.
3. Color all the neighbors of v with color 1.
4. For each uncolored vertex u in G, repeat:
   (a) If u has no adjacent nodes with color 1, then assign u with 1.
   (b) Else, assign u with 0.
5. For each edge (u,v) in G, if u and v are same colored, return false.
6. If all vertices are colored, return true.
7. Else, go to step 1, choose different vertex as v.

This algorithm works by trying to color the graph using just 2 colors. Let us breakdown the algorithm step by step.

Step 1 selects an arbitrary vertex v in G to start the coloring process. Steps 2 and 3 assign 2 different colors to v and its neighbors. Thus the graph is bifurcated into 2 sets of vertices such that no two vertices in the same set are adjacent. Step 4 lets the algorithm iterate through the uncolored vertices in the graph. For each uncolored vertex u, it checks if its any of the neighbors are colored with 1. If none of them is colored with 1, assign 1 to u. Otherwise assign it color 0. Step 5 checks the edges in graph to make sure that no 2 adjacent vertices in graph have same color. If any such pair is found, $\chi(G)! = 2$, and algorithm returns FALSE. Otherwise if all vertices are colored, algorithm return TRUE.

For example, take a cyclic graph with 4 vertices A, B, C, D and 4 edges A-B, B-C, C-D, D-A. Now, select vertex A first. Assign color 0 to it. Find its neighbors. We have vertices B and D as A's neighbors. Color them with color 1. Now uncolored vertex we have is vertex C. Check if the neighbors of C are colored with color 1, if yes assign C with color 0, else with color 1. We know that B and D are also C's neighbors and both are already colored in color 1. So, C is colored in color 0. Check for each edge, if both the vertices have same colors. Till now we have, A:0, B:1, C:0, D:1. So, none of these edges have vertices with same colors. thus, algorithm returns TRUE. The chromatic number of this graph can be 2.

Let us modify this example by adding a vertex E so that A, B, C, D, E form a cyclic graph with edges A-B, B-C, C-D, D-E and E-A. Perform the same procedure. Select vertex A and assign it 0. Assign its neighbors B and E with 1. Now we have C and D as uncolored vertices. Select C. C has neighbors B and D where B is colored 1 and D is uncolored. C has no adjacent nodes with color 0, so assign C with 0. Select D now. D now has neighboring nodes C with color 0 and E with color 1. It defies condition 1, it has an adjacent node with color 1. So, assign D with 1. Now check the edges. We clearly know that for edge D-E, both vertices are colored same. The algorithm returns FALSE. Thus, this graph needs a chromatic number more than 2, thus $\chi(G)! = 2$.

The correctness of the algorithm is guaranteed because, a graph will have $\chi(G) = 2$, if and and only if it can be colored using only 2 colors. The algorithm starts with an arbitrary vertex and divides the graph into 2 sets of vertices, one with color 0 and another with color 1. Then it colors remaining vertices by checking their neighbors' colors and it ensures that no 2 adjacent vertices have same color. If $\chi(G) = 2$, then algorithm will color it with 2 different colors and return TRUE, else algorithm will find 2 adjacent vertices with same color so, $\chi(G)! = 2$ and algorithm returns FALSE.

The running time complexity of this algorithm is $O(n + m)$, where n is the number of vertices and m is the number of edges in graph G, as it iterates through all vertices and edges once once.