Final Exam

Due: March 15, 2022

Submitted by:- PARTH PARASHAR (PSU ID:- 923928157)

Your solutions must be typed and submitted as a PDF through Canvas on the day it's due. If your answer requires or would be improved with a drawing, feel free to create that by hand and simply include it with your submission, either in the PDF or as a separate file.

Problem 1: Divide and Conquer Consider a monotonously decreasing function $f : \mathbb{N} \rightarrow \mathbb{Z}$, that is a function defined on the natural numbers that outputs integers such that $\forall i \in \mathbb{N} f(i) > f(i+1)$. Assume that we can evaluate $f(i)$ at any i in constant time, we want to find $n = min(i \in \mathbb{N} \mid f(i) \leq 0)$, in other words we want to find the input where f first becomes negative. We can obviously solve this problem in $O(n)$ time by simply evaluating $f(1), f(2), f(3), \ldots, f(n)$.

[10 points] Describe a $O(\log n)$ time algorithm for this problem. (Remember that you do not know n in advance)

Answer: - **Now,** According to the question, we don't know the value of n. However, we can narrow down to a subset of indices by iteratively doubling a variable until the first negative value is observed.

The steps that we can follow are: -

1) Let this index obtained by iteratively doubling a variable = high

2) Therefore, Lowest possible index where a negative value might be present will be at high/2

3) From the above two steps, we can say that the first negative value will be present in between the indices high/2 and high.

4) The operation mentioned above will be done in O(log n)

5) When combining both the operations mentioned above, it will take O(log n)

The algorithm for the same is given below: -

```
Binary_Search(low_element, high_element)
Middle_element=(low_element+high_element)/2
If(f(middle_element) <0 and f(middle_element-1)>=0
    Return middle_element
If (f(middle_element)<=0:
    Return Binary_Search(low_element,middle_element-1)
Else:
    Return Binary_Search(middle_element+1, high_element)
Call-the-function Find_Min() which is defined as below
Iterator i=0
While(f(iterator)>0)
Iterator = iterator * 2;
Return Binary_Search(iterator/2, iterator)
```

Now, we know that the steps which are involved in time-complexity analysis are: -

1) Performing the while with the iterator While(f(iterator)>0)
   This operation takes O(log n) time

2) From this while loop, the iterator is used to iteratively call Binary_Search method Return Binary_Search(iterator/2, iterator)
   This operation takes O(log n) time.

   Hence the time complexity for this algorithm becomes as: - O(log n).

Problem 2: Median Finding Finding the median element of a sorted array is easy, simply return the middle element. What if we are given two sorted arrays, *A* and *B*, of lengths *m* and *n* respectively, and you want to find the median of all the numbers in *A* and *B*?

(a) [5 points] Briefly describe a naive $\Theta(n + m)$ time algorithm for this and explain your answer. (You do not need to provide pseudocode for this, just enough detail that the complexity can be clearly established.)

Answer: - Now, we know that the two arrays which are given to us are sorted.
Let the two sorted arrays be A and B.
To find the median of all the numbers in the arrays A and B, we will follow the steps mentioned below: -

1) Merge the arrays to create a new sorted array. This will be done in O(n+m) time with the help of the two-pointers method.
2) After we have merged the two arrays, the length of the newly created array (which is sorted) would be n+m.
3) Now, we will check the value of (n+m) and determine whether it is odd or even.
4) If the value of n+m is odd, then median will be an index (n+m)/2
5) Or if the value of n+m is even, then median is the average of the elements at indices ((n+m)/2)-1 and (n+m)/2

Finding the median from the above two steps will take O(1) time.

This means that since there are n+m elements and the finding the median from the above steps take O(1) time, therefore,
The time complexity for this algorithm will be O(n+m)

(b) [5 points] If $m = n$, give an algorithm that runs in $\Theta(\log n)$ and explain your answer. (You do not need to provide pseudocode for this, just enough detail that the complexity can be clearly established.)

Answer: - Now, it is given in the problem that both the arrays are of the same size.
So, the steps which needs to be followed are: -

1) We will find the medians of both the arrays A and B. Let those medians be named as m1 and m2 respectively.
2) Now, if m1==m2, then either of m1 or m2 would be the median.
3) Otherwise, if m1<m2, then median is either in the second half of A or in the first half of B.
4) Or if m1>m2, then it is clear that the median is either in the first half of A or in the second half of B

5) Following this recursively, we come down to two arrays of size 2 each.
6) Now, to find the median among these, we will simply find the average of max(A[0],B[0]) and min(A[1],B[1]).
7) Upon analyzing the recursion, we can see that we are getting a binary tree whose root is at the bottom and the height of this tree is log n.
8) Therefore, the whole recursive steps take 0(log n) and the last step after this takes o(1) time.
9) Therefore, the total time taken in O(log n).

(c) [5 points] Give an algorithm that runs in $\Theta(\log min(m, n))$ and explain your answer. (You do not need to provide pseudocode for this, just enough detail that the complexity can be clearly established.)

Answer: - For solving this problem, we will follow the steps mentioned below: -

1) We will consider the array which is smaller, and for better usability let's assume $m < n$.
2) Consider a position for partition in array A (It would be great if it is considered to be the position of median).
3) Consider the same position for partitioning array B, so that we can make sure that both the left partitions of A and B combined have same number of elements as right partitions of A and B combined.
4) Now, if we can show that the max element of left partition of A is less than or equal to min element of right partition of B and max element of left partition of B is less than or equal to max element of right partition of A, then we can say that every element in left partitions is less than every element in right partitions.

5) In that case, we can find the median in $O(1)$ time. i.e., if the size of the combined is even, the median would be Average of max (max elements of left partitions of A and B) and min (min elements of right partitions of A and B), else if it is odd, the median is max (max elements of left partitions of A and B).

6) The next challenge that we have is how to partition the arrays to get into such arrangement as mentioned above, i.e., every element in left partitions is less than every element in right partitions.

We can do this iteratively by, moving partition A to the left (moving the binary search to the left) if max of max elements of left partitions of A is greater than min of min elements of right partitions of B, else otherwise moving the partition A to the right. Eventually, this gets converged and will meet the ideal median condition described above in 2nd and 3rd point.

Here we are considering the array that is less in size. Iteratively performing a version of binary search. Hence, overall, it would take a time complexity of $O(log\ min(m, n))$.

Problem 3: Dynamic Programming Suppose a dynamic programming algorithm creates an $n \times m$ table and to compute each entry of the table it takes a minimum over at most m (previously computed) other entries.

[5 points] What would the running time of this algorithm be, assuming there is no other computa- tions? Explain your answer.

Answer: - Now as we know that there is no other computation, we can say that

1) Creating a n*m table needs n*m operations
2) Filling each entry takes m operations
3) Since there are n*m entries to be filled, so in total it takes n*m*m operations

Because of these steps, the final number of operations will be: -

Total operations = n*m + n*m*m

Because of these total operations, it can be said that the time complexity will be:- O(n*m*m).

For example: -
Suppose we have 5 * 6 table,
So the number of entries will be 30 and the number of operations that will be required will be 5*6*6=180 operations, This is because there will be 30 entries and each entry will have to be computed and compared for the values pertaining to that bucket.

Therefore, we can say that the time complexity will be: - O(n*m*m).

Problem 4: Greedy Algorithms Consider the *Fractional Knapsack Problem*: 'Given *n* items with values and weights and a knapsack with capacity *C*. You may take fractions of each item and as many of each item as you want. Find the selection of items that maximizes the possible value that can be fit in your knapsack.'

[10 points] Describe an efficient greedy strategy for this problem and prove that your greedy choice strategy finds an optimal solution in all cases.

Answer: - According to the Fractional Knapsack problem, given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

For fractional Knapsack, we can break the items for maximizing the total value of the knapsack. The problem in which we break an item is also called the fractional knapsack problem.

The best approach to use greedy approach to solve the problem.

1) The basic idea of the greedy approach is to calculate the ratio: - Value / Weight for each item

2) The next step would be to sort the items based on this ratio calculated above.

3) Take the item with the highest ratio and add them until we cannot add the next item as an whole

4) at the end add the next item as much as we can

The algorithm/ pseudo code for the same is given below: -

```
Fractional Knapsack (Array W, Array V, int M)
for i <- 1 to size (V)
        calculate cost[i] <- V[i] / W[i]
Sort-Descending (cost)
i ← 1
 while (i <= size(V))
        if  W[i] <= M
                M ← M – W[i]
                total ← total + V[i];
        if  W[i] > M
                i ← i+1
```

the complexity of this algorithm can be defined as: -

1) If simple sort algorithm (selection, bubble…) then the complexity of the whole problem is O(n2).
2) If using quick sort or merge sort then the complexity of the whole problem is O(nlogn).

Problem 5: Hash Tables For a hash table of size m consider the following obvious hash function $h(x) = x \mod m$.

(a) [5 points] Assuming that each entry in the table stores an unordered linked list, and that when a new element is added it is inserted at the beginning of the list. In the worst case, what is the time complexity to insert n keys into the table if chaining is used to resolve collisions? Briefly explain your answer.

Answer: - In the *absence* of collisions, inserting a key into a hash table/map is O(1), since looking up the bucket is a constant time operation.
I would not expect this to vary in the case of collisions, assuming that collisions are resolved using a linked list *and* that the new element is inserted to the head of the list.
The reason for this is that adding an new element to the head of a linked list it also basically O(1).
So, inserting under these assumptions should also be O(1), and therefore inserting n keys should be O(n)

(b) [5 points] What if instead of using an unordered link list to represent each bucket we want to maintain a sorted linked list. In the worst case, what is the time complexity to insert n keys into the table given this modification? Briefly explain your answer.

Answer:- In case of a sorted linked list:- The keys are checked before insertion and sorted into the array, thereby reducing the number of comparisons required to perform the operations, hence,
Inserting n keys will take O(n) time.

Problem 6: Amortized Analysis In class we analyzed Dynamic Table Expansion under the assump- tion that if we want to insert a new element in a table T that is full, we first copy all the elements of T into a new table T of size $|T| = 2|T|$, and then enter the new element in T. In this question, we consider cases where the size of T is not double the size of T. Assume (as in class) that entering an element in an empty slot of a table costs 1, and copying an element from a table into a new table also costs 1. Suppose that $|T| = |T| + 1000$, i.e, each new table has 1000 more slots than the previous one.

[10 points] Starting with an empty table T with 1000 slots, we insert a sequence of n elements. What is the amortized cost per insertion? Explain your answer. (Hint: aggregate analysis might help)

<u>Answer: - On next Page</u>

Ans 6) For the amortized cost analysis of this problem, I will be using the "aggregate" method.

$\Rightarrow$ The cost of ith operation $= \frac{1}{1001}$ if $i-1$ = multiple of 1000

$\Rightarrow$ Total cost of iterations for n insertions would be

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=1}^{n/1000} 1000$$

$$< n + n$$

$$\simeq 2n$$

$\Rightarrow$ That is the cheaper operation (insertion costs) take a total cost 'n'

Expensive costs for copying the elements each time the slots are increased takes

$1000\left(\frac{n}{1000}\right)$ which is approximately equal to

$$1000\left(\frac{n}{1000}\right) \simeq n$$

for example :-

for 3001 elements → it would take 1000

for each 1001th, 2001th, 3001th operation

⇒ Amortized cost per inserting is 2. This is

because $\frac{2\cancel{n}}{\cancel{n}} = 2$. which is of the order O(1)

Problem 7: Data Structures Disjoint Sets: Consider the set of initially unrelated elements:

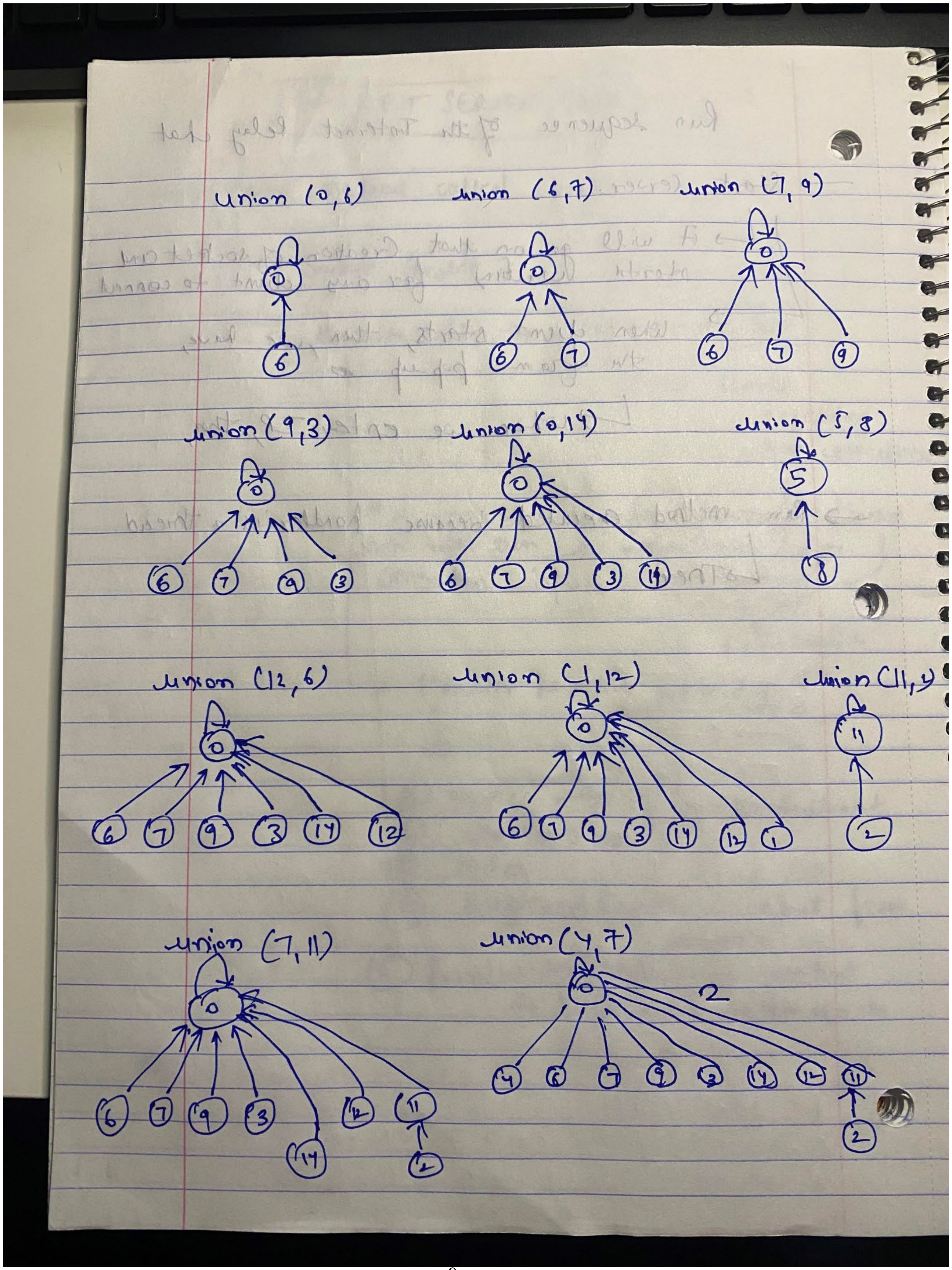
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14

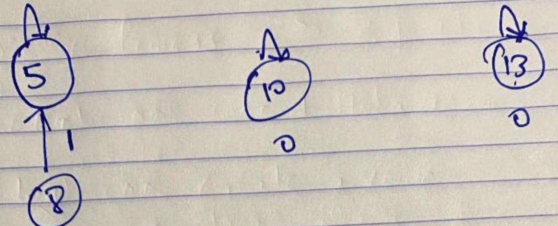
Draw the final forest of trees that results from the following sequence of operations using union as described in lecture. When unioning trees of the same size use the representative element from the first argument as representative for the resulting tree. Please include the rank of the root node in your answer.

14

[10 points] Union(0, 6), Union(6, 7), Union(7, 9), Union(9, 3), Union(0, 14), Union(5, 8), Union(12, 6), Union(1, 12), Union(11, 2), Union(7, 11), Union(4, 7)

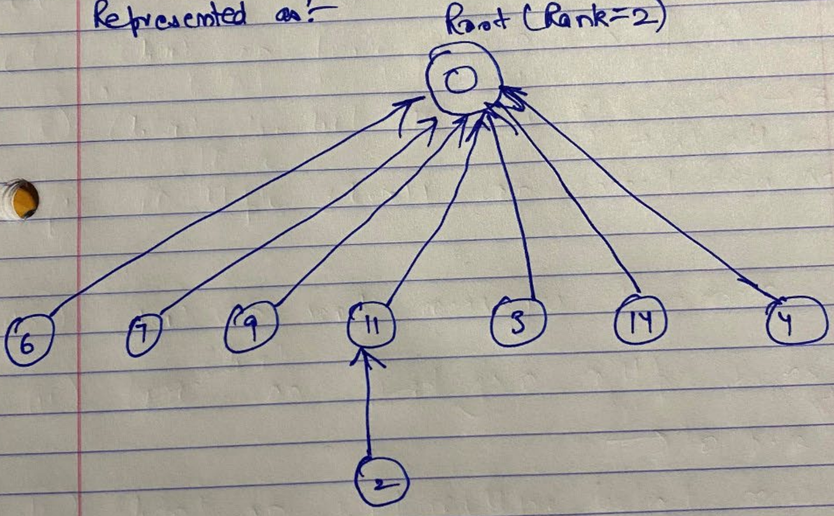
<u>Answer: - On Next Page</u>

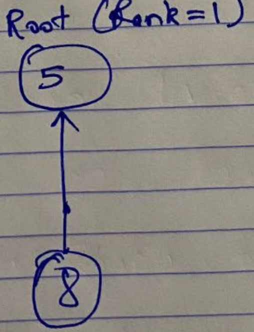

union (0,6)
union (6,7)
union (7,9)
union (9,3)
union (0,14)
union (5,8)
union (12,6)
union (1,12)
union (11,2
union (7,11)
union (4,7)
2



The rank of the nodes are 2, 1, 0, 0

Represented as:-





Problem 8: Lower Bounds

[10 points] Prove that finding the sum of an array of n integers requires $\Omega(n)$ time.

Answer: - The question is to find the **sum of all the values** so iterate through each element in the array and add each element to a temporary sum value.

```
temp_sum = 0
for i in 1 ...array.length
   temp_sum = temp_sum + array[i]
```

Since you need to go through all the elements in the array, this **program depends linearly to the number of elements**. If you have 10 elements, iterate through 10 elements, if you have a million you have no choice other than to go through all the million elements and add each of them. Thus the **time complexity is Θ(n)**.

If you are finding the sum of all the elements and you don't know any thing about the data then you need to look at all the elements at least once. Thus n is the lower bound. You also need not look at the element more than once. n is also the upper bound. Hence the complexity is Θ(n).

This can ***also be Proved*** in the following form by using the Theoretic Lower Bounds Technique which means I would prove this by disproving that there exists an algorithm that finds the sum of n integers in O(n) time.

Let us suppose that the n integers are unordered.

Steps: - 1) Let us assume that there exists an algorithm that finds the sum of all integers in O(n) time

2) It is safe to say that the sum of n integers cannot be calculated without visiting every integer. Therefore, Ω (n) algorithm does visit all the n integers

3) We know that an algorithm that runs in O(n) cannot visit every integer of all the available n integers. That leaves us with the inference that there exists an integer that is never accessed by our hypothetical algorithm. Let this integer be x

4) Now, considering the integer x is never accessed and if we compute the sum of all integers excluding x, the algorithm will give an incorrect sum unless integer x is 0.

Therefore, this $o(n)$ algorithm will give us an incorrect answer.

Hence, an algorithm for finding the sum of all integers does require Ω (n), assuming the elements are unordered/not sequential.

However, If the given input has some sort of pattern or if the input is sorted, then, if we supply this information to the algorithm, then the algorithm will take O(1) for finding the sum for any value of 'n' number of inputs.

Problem 9: NP-Complete Problems Consider the two decision problems below. One of these prob- lems is in P and the other is N P-Complete. (You should use the fact that the general *clique* problem is NP-Complete in your answer)

Problem 1 Given an undirected graph $G = (V, E)$ with $|V|$ even, does G contain a clique with at least $\frac{|V|}{2}$ vertices?

Problem 2 Given an undirected graph $G = (V, E)$, does G contain a clique with at least $|V|-2$ vertices?

Answer: - On next page

(a) **[5 points]** Identify which problem is in P and briefly describe how it can be solved in poly time.

Answer: - Problem-2 is in P, as it can be solved in polynomial time. This is because of the way Problem 2 is defined. Since we know that in problem-2, we are given an undirected graph G=(V,E) which does contain a clique with at least |V|-2 vertices. Now for this graph problem, we know that

E= edges

V= vertices,

Then for any undirected graph, the time complexity will be in the range of O(E*V)

Replacing the values of E and V in the equation, we have,

O(E*V) = O(E*(|V|-2))

Now, we know that even if the size of the edge is E=V^2 (for maximum edge and vertex relationship),

The time complexity is still in the range of O(V^2 * (|v|-2)) which still is a polynomial.

Therefore, we are still in the P.

(b) **[5 points]** For the other problem, use a reduction to show that the problem is NP-hard.

Answer: - The problem-1 is an NP-Complete problem.

Now, according to the question, we have, an undirected graph $G = (V, E)$ with $|V|$ even, does G contain a clique with at least -| V | / 2 vertices

Given the problem, we will create a new graph G' = G/vertex

According to the definition, if there is a boundary between them at G, there will be no boundaries between those vertices in G'.

If there is no limit between the given pair and we insert edges between them in G', the set is same for all vertices.

This is dependent on the polynomial size and the input size.

If this new Graph G' contains a set of of size K, then there is a private collection of size k.

As a result, in this issue presented, there is a decrease in the problem set.

Now, we know that for any NP Complete problem, there exists a polynomial – time algorithm that can transform the problem into any other NP-complete problem.

In the above scenario, the keys are given in the form of Vertices and edges. G=(V,E)

V= Vertices

E= Edges.

Now, for this problem, we have, the time complexity in the range of O(V*E)

Replacing the values of E and V, we have

Time complexity lies in the range of O(|V|/2 * V). This is again representing a NP problem polynomial which can be reduced further for obtaining the required results.

Hence the condition for NP-Complete is satisfied for this graph problem and thus, it can be classified as an NP-Complete problem.

Problem 10: Approximation Algorithms You've been asked to help a friend move. For simplicity, let us assume that your car has size 1, and your friend's n possessions $x_1, x_2, ..., x_n$ have real number sizes between 0 and 1. The goal is to divide those items into as few carloads as possible, such that all items arrive at the new location, and the car is never overpacked. Consider the following **First-Fit** approximation algorithm. Put item x_1 in the first carload. Then, for $i = 2, 3, ..., n$, put x_i in the first carload that has room for it, starting a new carload if necessary. For example: if $x_1 = 0.2, x_2 = 0.4, x_3 = 0.6$, and $x_4 = 0.3$, the **First-Fit** algorithm would place x_1 and x_2 into the first carload, x_3 in the second carload, and then x_4 in the first carload where there is still room. Note that all decision are made offline; we divide the n possessions into carloads before any trips are actually made.

(a) [5 points] Give an example input where this **First-Fit** algorithm would fail to *minimize* the number of carloads.

Answer: - Let us consider that

X1 = 0.7
X2=0.8
X3 = 0.2
X4=0.3

Now according to the first fit algorithm, we have: -
X1 = Goes into the first carload
X2= Goes into the second carload
X3= Goes into the first carload
X4= Goes into the third carload

Therefore, we can say that the first fit gives us 3 carloads.

However, an optimal one would be 2 carloads i.e. X1 and X4 go into the carload 1.
X2 and X3 go into the carload 3.

So, First-Fit algorithm fails to give the optimal solution for this input.

(b) [5 points] Prove that the **First-Fit** algorithm is a 2-approximation algorithm. (Hint: How many carloads can be less than half full)

Answer: - First let us consider a carload whose size is less than 0.5 and it is represented by c1.
Now, let us assume that the next item Xi goes into a new carload c2.
Going by the first search algorithm, this scenario happens when the Xi+Ci > 1.
We can also infer that Xi>0.5 because C1<0.5

Now, for the sake of simplicity, let us rearrange the weights in such a way that the first carload C1 is exactly 0.5 and carload C2 is at least 0.5 . Let's see if this is possible by considering a normal example: 7-

Let C_1 be 0.4, next item X_i be 0.7. By following First-Fit we will X_i will be going into a new carload c_2.
We can take 0.1 of C_2 and put it in C_1 to make $C_1 = 0.5$.
Therefore, C_2 becomes 0.6.
Now, both carloads are at least half i.e., ≥ 0.5.
So, it is possible to make both carloads at least half full.

So, if C is the total number of carloads, we can say that total sizes combined of all carloads, let it be S, is at

least $0.5C$ i.e., $S \geq 0.5C$,
and most optimal carload (Let it be OP) will at least be S.
From both the equations, we can say that:

$$S \geq 0.5C$$

$$OP \geq S$$

Therefore, we have: -
$1/2 < OP/C$

Rearranging these terms will give us: -
$C/OP <= 2$

This proves that the First-Fit Algorithm is a 2-Approximation Algorithm.