- The instructions are the same as in Homework-0, 1, 2.

There are 6 questions for a total of 100 points.

---

1. (5 points) Solve the following recurrence relations. You may use the Master theorem wherever applicable.

   (a) $T(n) = 3T(n/3) + cn$, T(1) = c

   We can write c*n = O(n)
   Then T(n) = 3 * T(n/3) + O(n)
   Now applying master theorem with a = 3, b = 3 and d = 1, we see that a = b^d
   Hence, T(n) = O(n^1 * log(n)) = O(nlog(n))
   Answer : O(nlog(n))

   (b) $T(n) = 3T(n/3) + cn^2$, T(1) = c

   We can write c*n^2 = O(n^2)
   Then T(n) = 3 * T(n/3) + O(n^2)
   Now applying master theorem with a = 3, b = 3 and d = 2, we see that a ¡ b^d
   Hence, T(n) = O(n^2)
   Answer : O(n^2)

   (c) $T(n) = 3T(n-1) + 1$, T(1) = 1
   T(n) = 3 * T(n-1) + 1
   T(n) = 3 * [3 * T(n-2) + 1] + 1 = 3^2 * T(n-2) + 3 + 1
   ...
   T(n) = 3^{n-1} * T(1) + 3^{n-2} + ... + 3 + 1
   = 3^{n-1} + 3^{n-2} + ... + 3 + 1
   = (3^n - 1) / 2
   = O(3^n)
   Answer : O(3^n)

2. (20 points) Consider the following problem: You are given a pointer to the root r of a binary tree, where each vertex v has pointers $v.lc$ and $v.rc$ to the left and right child, and a value $Val(v) > 0$. The value NIL represents a null pointer, showing that v has no child of that type. You wish to find the path from r to some leaf that minimizes the total values of vertices along that path. Give an algorithm to find the minimum sum of vertices along such a path along with a proof of correctness and runtime analysis.

   Main idea: We will divide the tree into left and right subtrees, find the minimum cost path from the both, and add the weight of root node to the lesser one to get the final minimum.
   An algorithm implementing the above solution recursively is as follows:

   findMinimumPath(root):
   - if Val(root.lc) == NIL and Val(root.rc) == NIL:
     - **return** Val(root)
   - if Val(root.lc) == NIL and not(Val(root.rc) == NIL):

- rcval = findMinimumPath(root.rc)
- **return** rcval + Val(root)

- if not(Val(root.lc) == NIL) and Val(root.rc) == NIL:
  - lcval = findMinimumPath(root.lc)
  - **return** lcval + Val(root)
- if not(Val(root.lc) == NIL) and not(Val(root.rc) == NIL):
  - rcval = findMinimumPath(root.rc)
  - lcval = findMinimumPath(root.lc)
  - **return** min(rcval,lcval) + Val(root)

Proof of correctness:
Induction statement: The algorithm returns right value for binary tree with all heights 1,2,3,...
Base Case: Height=1. Our algorithm will return the root value, which should be the answer.
Induction hypothesis: Let our algorithm give right values for height h=1,2,....k. Let us prove that it gives the right answer for trees of height k+1
Proof:
Let us consider a tree of height k+1, and let's see what happens at the root.

Case 1: left child of root is null but right child is not null.
In this case, the path will go from root to right child to get to the leaf which gives minimum value path. Now, the right subtree is of height k, and we have already stated that the algo will give right answer for a tree of height k. Adding the root value to the value given by right subtree, we get the minimum value path from root to a certain leaf in the right subtree.

Case 2: right child of root is null but left child is not null.
Similar to the above case

Case 3: both left and right children of root are not null
The algorithm will calculate minimum value of such path in the left subtree and the right subtree correctly according to out hypothesis. Now, to get the minimum of current tree, we are taking the minimum value among left subtree path and right subtree path, and adding root value to it, which gives us the minimum value for such path in current tree.
Hence, our algorithm works for a tree of height k+1 as well.
Therefore, by induction, the algorithm always gives the minimum value.

Running time: The recurrence relation for the above algorithm is given as:
T(n) = 2*T($\frac{n}{2}$) + O(1) (assuming addition to be O(1) operation)
Using master theorem, a=2,b=2,d=0, we get the running time as O(n)

3. (20 points) Let S and T be sorted arrays each containing n elements. Design an algorithm to find the n^{th} smallest element out of the $2n$ elements in S and T . Discuss running time, and give proof of correctness.

We will write a general algorithm for finding the kth smallest element given 2 sorted arrays S and T of length n each. Then we can replace k with n to find the nth smallest element out of the given 2n elements

The main idea behind the algorithm is divide and conquer strategy. The arrays S and T are divided into segments of length (k/2) to make smaller sub-problems and compute the result.

Pseudo-Code for the algorithm :

```
procedure kth(S, T, n, k, st1, st2):
    ## st1 and st2 indicate the starting indices of S and T
    ## The search space is S[s1:n] and T[s2:n] and we are finding
    ## the kth smallest element out of these 2n - s1 - s2 elements

    if (st1 == n):
        return T[st2 + k - 1]

    if (st2 == n):
        return S[st1 + k - 1]

    ## Base Case
    if (k == 1):
        if(S[st1] < T[st2]):
            return S[st1]
        else:
            return T[st2]

    curr = [k / 2] # integer value of (k/2)

    ## Block 1
    if(curr - 1 ≥ n - st1):

        ## Block 1A
        if (S[n - 1] < T[st2 + curr - 1]):
            return T[st2 + (k - (n - st1) - 1)]

        ## Block 1B
        else:
            return kth(S, T, n, k - curr, st1, st2 + curr)

    ## Block 2
    else if (curr - 1 ≥ n - st2):

        ## Block 2A
        if (T[n - 1] < S[st1 + curr - 1]):
            return S[st1 + (k - (n - st2) - 1)]

        ## Block 2B
        else:
            return kth(S, T, n, k - curr,st1 + curr, st2)

    ## Block 3
    else:

        ## Block 3 A
        if (S[curr + st1 - 1] < T[curr + st2 - 1]):
            return kth(S, T, n, k - curr, st1 + curr, st2)
```

```
## Block 3 B
else:
    return kth(S, T, n, k - curr, st1, st2 + curr)
```

answer (kth smallest element in S and T of n elements each) = kth(S,T,n,k,0,0)

Running Time Analysis :
(We can approximately write k - [(k/2)] as (k/2))
The algorithm divides the arrays into sub arrays of size (k/2), so running time also has dependency on k.

T(k) = T(k/2) + O(1)
Reason : (k/2) for the recursive call and there are a fixed number of 'if-else' conditional blocks

Applying master theorem with a = 1, b = 2 and d = 0, hence a = b^d
So, T(k) = O(k^0 * log(k)) = O(log(k))
Now since k = n, Running time = O(log(n))

Proof of correctness :

Base case : k = 1. In this case, we are looking for the smallest element out of all elements in S and T. So, the algorithm will compare S[0] and T[0] (since initially st1 = st2 = 0) and the smaller of the two will be returned as the answer. Hence, we will get the smallest of the smallest element of S and T, which is the correct result.

Proof of combination step :
We are dealing with subarray S' of S and T' of T, which are not explicitly written in the pseudocode because we are using indices st1 and st2 to indicate the subarrays.
Length of S' = n - st1 and length of T' = n - st2 [st1 and st2 are the indices representing the starting point of searching in S and T in the sub-problem which reduces to searching in arrays S' and T'].
Basically S' = S[st1:n] and T' = T[st2:n] and we search for the kth smallest among all elements in S' and T'.
Please note that [r] : denotes integer part of r

Case 1 : Length of S' is less than [k/2].

Case 1 a : Last element in S' is less than the [k/2]th element of T'.

Then all elements of S' are less than the kth element we require. So, the kth smallest element will logically be the (k - (n-st1))th element in T'. This is what is returned by the algorithm above according to block 1A.

Case 1 b : Last element in S' is not less than (k/2)th element of T'.

The kth smallest element will be logically the (k/2)th smallest element found in T'[(k/2):n] and S' since the first (k/2) elements in T' are smaller than the last element in S' and S' has less than [k/2] elements. The algorithm returns this [k/2] the smallest element among all elements in S' and T'[[(k/2)]:n] as shown in block 1 B.

Case 2 : Length of T' is less than [k/2].

Case 2 a : Last element in T' is less than the [k/2]th element of S'.

Then all elements of T' are less than the kth element we require. So, the kth smallest element will logically be the (k - (n-st2))th element in S'. This is what is returned by the algorithm

above according to block 2A.

Case 2 b : Last element in T' is not less than [k/2]th element of S'.
The kth smallest element will be logically the [k/2]th smallest element found in S'[[k/2]:n] and T' since the first [k/2] elements in S' are smaller than the last element in T' and T' has less than [k/2] elements. The algorithm returns this (k/2) the smallest element among all elements in T' and S'[[k/2]:n] as shown in block 2 B.

Case 3 : Length of both S' and T' is more than [k/2].
Case 3 a : [k/2]th element of S' is less then [k/2]th element of T'.
The kth smallest element amongst elements in S' and T' will logically be the [k/2]th smallest element among the elements in S'[[k/2]:] and T' since the first [k/2] elements in S' are definitely smaller than the kth smallest element in S' and T'. The algorithm checks this condition and returns the correct answer by formulating the required subproblem as shown in block 3 A.

Case 3 b : [k/2]th element of T' is less then [k/2]th element of S'.
The kth smallest element amongst elements in S' and T' will logically be the [k/2]th smallest element among the elements in T'[[k/2]:] and S' since the first [k/2] elements in T' are definitely smaller than the kth smallest element in S' and T'. The algorithm checks this condition and returns the correct answer by formulating the required subproblem as shown in block 3 B.

After exhausting all possible cases, we can conclude that the algorithm is proven to be correct for any given S and T sorted arrays of length n, and the answer = kth(S,T,n,k,0,0) returns the kth smallest element among all elements in S and T.
So, the nth smallest element can be obtained as answer = kth(S,T,n,n,0,0).

4. An array $A[1...n]$ is said to have a majority element if more than half (i.e., $> n/2$) of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $A[i] \geq A[j]$?" (Think of the array elements as GIF files, say.) However you can answer questions of the form: "is $A[i] = A[j]$?" in constant time.

   (a) (10 points) Show how to solve this problem in $O(n \log n)$ time. Provide a runtime analysis and proof of correctness.

   (*Hint: Split the array A into two arrays A1 and A2 of half the size. Does knowing the majority elements of A1 and A2 help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.*)

   Answer.
   Proof.
   Base Case: If len(A) is 1 then the single element is the majority element. Since, count = 1 (>len(A)/2 = 1/2).
   Proof of Combination Step:
   Using the hint provided, we will split the array A into $A1$ and $A2$ of half the size and run the algorithm on each half recursively. Then we will find the majority element using the following case analysis.
   Case 1: There is no majority element in either $A1$ or $A2$
   This means that there is no element in either of the halves with count > n/4. Let the element with

max count(not majority element since it does not exist) in $A1$ have count n_1 and element with max count in $A2$ have count n_2. If they are the same element,

$$\begin{aligned} n_1 &\leq n/4 \\ n_2 &\leq n/4 \\ \implies n_1 + n_2 &\leq n/2 \end{aligned}$$

This implies that even if the max count element is same in both it cannot be the majority element of the combined array.
Hence, A has no majority element
Case 2: WLOG,Consider case where $A1$ has a majority element and $A2$ does not
This means that there is an element in $A1$ with count > n/4, let the element be e and its count be n_1. Let the count of e in $A2$ be n_2.Then, if $n_1 + n_2 > n/2$ then the it will be the majority element. If not, then there is no majority element. Since, for any other element it's count is < n/4 in both $A1$ and $A2$, hence with the reasoning in Case1 it cannot be a majority element.
Case 3: Both $A1$ and $A2$ have majority elements
This means that there is an element in both $A1$ and $A2$ with count > n/4. Let the majority element of $A1$ be m_1 and its count be n_1. Let the majority element of $A2$ be m_2 and its count be n_2. We know that the majority element of A can be only either m_1 or m_2. (From Case1 reasoning for other elements).
Let the total count of m_1 in $A1$ and $A2$ be t_1 and the total count of m_2 in $A1$ and $A2$ be t_2.Then,
If $t1$ ¿ $t2$ and $t_1 > n/2$ then m_1 is the majority element.
Else if $t2$ ¿ $t1$ and $t2 > n/2$ then m_2 is the majority element.
Else, there is no majority element.

Algorithm:

```
procedure MajorityElem(A):                n = length(A)
    return FindMajorityElem(A,n)[0]

procedure FindMajorityELem(A, n):
    if(n==1):
        return A[0],1
    Split A by half into A1 and A2
    m1,c1 = FindMajorityElem(A1)
    m2,c2 = FindMajorityElem(A2)
    if(m1 == NULL and m2 == NULL):
        return NULL,0
    else if(m1 == NULL):
        count = c2
        for e in A1:
            if (e==m2):
                count +=1
        if(count ¿ n/2):
            return m2,count
        else:
            return NULL,0
    else if(m2 == NULL):
        count = c1
        for e in A2:
            if (e==m1):
                count +=1
```

```
        if(count > n/2):
            return m1,count
        else:
            return NULL,0
    else:
        count1 = c1
        count2 = c2
        for e in A2:
            if (e==m1):
                count1 +=1
        for e in A1:
            if (e==m2):
                count2 +=1
        if(count1 >= count2 and count1 > n/2):
            return m1,count1
        else if(count2 > count1 and count2 > n/2):
            return m2,count2
        else:
            return NULL,0
```

Runtime Analysis:
The time complexity of the combination step is O(n) since the maximum number of steps traversed is in the last case - where the entire array is traversed. The recursive relation is

$$T(n) = 2T(n/2) + O(n)$$

Using Master Theorem, the time complexity is O($nlog_2n$).

(b) (10 points) Design a linear time algorithm. Provide a runtime analysis and proof of correctness. (*Hint: Here is another divide-and-conquer approach:*

- *Pair up the elements of A arbitrarily, to get $n/2$ pairs (say n is even)*
- *Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them*
- *Show that after this procedure there are at most $n/2$ elements left, and that if A has a majority element then it's a majority in the remaining set as well)*

Answer.
According to their algorithm, we pair the elements up and if the two elements are different, discard both of them; if they are the same, keep just one of them.
Proof:
Claim: After above procedure there are at most $n/2$ elements left.
Proof: The maximum number of elements will remain when the least number of elements are deleted.
This means that, if all the pairs are identical pairs, then one of each will be discarded. If
Number of elements = n
Number of pairs = $n/2$
Number of elements discarded = Number of pairs(in worst case) = $n/2$
Number of elements remaining = $n - n/2 = n/2$.
Therefore, there are atmost $n/2$ elements remaining.

Claim: if A has a majority element then it's a majority in the remaining set as well
Proof:
Base Case: For n = 1, the majority element is the element itself. This is represented in our

algorithm as the case where n=0 and the oddelement carried over is the single element in the array. This returns the oddelement. Hence the base case is true.
Consider the two subsets $A1$ and $A2$ of set A, having elements equal to m and elements $\neq$ m respectively. If A has a m as a majority element, number of elements in $A1$ is greater than that of $A2$. Since the number of elements = m is greater than $n/2$, there will exist atleast one pair of elements = m. This means the resulting array will have atleast one m and will have atmost $n/2$ elements. Out of all possible pairs (p,q), the following scenarios can occur:

1. p = q
2. p = m and q $\neq$ m
3. p $\neq$ m and q = m
4. p $\neq$ m and q $\neq$ m and p $\neq$ q

For Case 2 and 3, the majority status of m is maintained as when we remove a majority element we also remove a non-majority element.
For Case 4, we remove 2 non-majority elements, hence majority status of m is maintained.
For Case 1, whether or not the element is equal to m, keeping one of them will ensure that majority status of m is not affected(since size of array is also reduced to half in one iteration).
If at any recursive call, an odd element exists(i.e if the number of elements is odd and one element is left out), after recursive call it is included in the array and checked if it breaks a tie that existed.(This covers the case where the recursive call predicts a Majority element that occurs exactly $n/2$ times and hence will not be considered a majority element if the oddelement is not equal to it)
Hence proved.

Algorithm.

```
procedure GetMajorityElem(A, oddelement = NULL):
    n = length(A)
    if(n==0):
        return oddelement
    nextArr = []
    i = 0
    while(i<n-1):
        if(A[i]==A[i+1]):
            nextArr.append(A[i])
    i+=2
    if (n%2==1):
        oddelement = A[n-1]
    majorityElem = GetMajorityElem(nextArr,oddelement)
    MajorityCount = FindFrequency(majorityElem,A)
    if(2*MajorityCount > n or (2*MajorityCount == n and oddelement == majorityElem))
:
        return majorityElem
    else:
        return NULL
```

Runtime Analysis:
The above algorithm calls makes a recursive call on an array that is in worst case of size $n/2$. The remaining computation of pairing and finding frequency is O(n). The recurrence relation is hence

$$T(n) = T(n/2) + O(n)$$

Therefore, the time complexity of the algorithm is O(n).

5. Consider the following algorithm for deciding whether an undirected graph has a *Hamiltonian Path* from x to y, i.e., a simple path in the graph from x to y going through all the nodes in G exactly once. ($N(x)$ is the set of neighbors of x, i.e. nodes directly connected to x in G).

   `HamPath(graph` G`, node` x`, node` y`)`
   - If $x = y$ is the only node in G return $True$.
   - If no node in G is connected to x, return $False$.
   - For each $z \in N(x)$ do:
     - If `HamPath(`$G - \{x\}, z, y$`)`, return $True$.
   - return $False$

   (a) (7 points) Give proof of correctness of the above backtracking algorithm.

   Proof by Induction:
   We will prove the correctness of above algorithm using induction on the number of vertices in the graph n.
   Base Case: When there is just one vertex, i.e. n=1. In this case, there obviously exists a path covering the whole graph and thats the vertex itself. Therefore, the algorithm returns true.
   Induction Hypothesis: Let us assume that our algorithm works for n=1,2,...k-1,k.
   Let us prove that the algorithm works for graph with k+1 nodes as well.
   As we start from any starting node x, two things can happen. If x is isolated, the algorithm will return false, which should be the case.
   Otherwise, it will run the function HampPath in the for loop for all neighbours of x, with a graph of k nodes(as x is subtracted). And, as we assumed that the algorithm returns right answer for k node graphs, the algo for k+1 will return right answer as well. If a Hamiltonian path exists from any of the neighbours of x, say z, to y in Graph G-x, say P. edge(x,z)+P will be the Hamiltonian Path from x to y in graph G.
   Hence, our algorithm works correctly for any graph.

   (b) (8 points) If every node of the graph G has degree (number of neighbors) at most 4, how long will this algorithm take at most? *(Hint: you can get a tighter bound than the most obvious one.)*
   Answer:
   The recurrence tree for the given algorithm will have **depth n**, for a graph with n nodes, as we have to cover all the nodes in the graph, exactly once for the HamPath.
   If the maximum degree for every node is 4, then an upper bound is (4^n), as each node will have 4 children in the tree. Now, if we look closely, each of the nodes after the first one are called after one of their neighbours are deleted from the graph. So, a better bound is $O(4*3^{n-1}) = O(3^n)$

   Now, as we know that maximum number of edges in a graph is (sum of degrees of all vertices)/2, so considering n nodes and given max degree 4, we get maximum number of edges as ((4*n)/2) = 2n.
   Now, consider removing vertices from the recursion tree. If we remove a vertex of degree d from the tree, it will lead to d new vertices, as it has d children remaining, and the recursion equation will be given as :
   T(n')=d*T(n'-d)+O(poly(n')) (where n' is the number of edges)
   Now putting the values of degree as 1,2,3 and 4, and assuming same degree throughout the graph to get worst case, we get running time for them as O(n'), $O(2^{n'/2})$, $O(3^{n'/3})$, and $O(4^{n'/4})$.

   Putting n'=2n as the maximum edges, we get O(n), $O(2^n)$, $O(3^{2n/3})$ and $O(2^{n/2})$.
   From here, we can clearly see that the worst case running time is $O(3^{2n/3})$.
   Therefore, the algorithm can take at most $O(3^{2n/3})$

6. (20 points) Design an $O(poly(n) \cdot 2^{n/4})$-time backtracking algorithm for the maximum independent set problem for graphs with bounded degree 3 (these are graphs where all vertices have degree at most 3). Give running time analysis and proof of correctness.

Answer:
Main idea: We will include all the vertices with degree 0 and 1, and will apply the given algoritm for vertices with degree 2 and 3.
Algorithm:

1. Include all vertices with degree 0 and 1.
2. For every vertex of degree 2, if it's neighbours have an edge between, then include the vertex and remove this vertex and it's 2 neighbours from the graph. If they do not have an edge, remove the 2 neighbours(say u1 and u2) and v and replace with new vertex and join this vertex to all the neighbours of u1 and u2.
3. For all vertices of degree 3, call the MIS algorithm recursively.

Proof of correctness:
Claim: Our algorithm gives the maximum independent subset.
Proof of the claim:

Degree 0 and 1
Consider the case of vertices with degree 0, 1.
There will be no edges to other vertex from a vertex of degree 0, and hence no other node will be rejected. So, taking vertex of degree 0 doesn't affect the remaining graph, and adds to the subset, giving us maximum independent subset of graph.
Now, for vertices with degree 1, let us assume a vertex v in our graph G, connected to any other vertex u of the remaining graph.
We claim that v should be a part of the best solution. Let us suppose we don't add v to our solution, then there can be two cases:

1. we didn't add u to the solution: In this case, a better solution exists, containing v, as it won't affect the remaining solution because of the absence of it's neighbour u.
2. we did add u to the solution: Now, since u was connected to v, and can be any vertex in the remaining graph, it is true to say that degree of u should be greater than 1, and can go upto 3. So, if degree u is 1(connected to v only), v will be removed in the children state, and taking u or v wont make much difference. In case degree of u is 2 or 3, taking u will end up removing 2 nodes or 3 nodes from remaining graph, which is worse than taking v and just removing u from graph.

Hence, it is safe to say that taking all the vertices with degree 0 and 1 will give best solution.

Degree = 2 vertices
Let the neighbours be u1 and u2 and the vertex be v
Case1: edge(u1,u2) exists
Claim: if edge exists then removing u1 and u2 and considering v gives MIS.
Proof by contradiction:
WLOG, let us pick u1 and let the set be S. Then we cannot pick v,u2 and other neighbours of u1. Consider Snew = S+v-u1. This has same number of nodes and no restriction on other neighbours of u1. Hence is a better choice. Contradiction.
Case 2: edge(u1,u2) doesn't exist
If the new vertex is included in the graph, then we can replace it with u1 and u2. (to maximise cardinality). If it is excluded from the graph we replace it with v, whose neighbours u1 and u2 will not be

included. Hence, the properties of MIS are not violated.

Degree = 3 vertices
We have a vertex u of degree 3
Case 1 : u is included
Now if u is taken, its neighbours are not taken according to the algo, hence the graph size is reduced by 4 vertices. So, we get T(n-4) time for computation.
Case 2 : u is excluded
Vertex u is excluded and also removed from the graph. So, the degree of all neighbours of u is now = 2. And we know that degree = 2 can be done in time which has a polynomial complexity with respect to the number of vertices in the graph. Therefore, this will also take $\leq$ T(n-4) time.

Running Time analysis :
T(n) = 2*T(n-4) + some polynomial expression in n
Let the polynomial expression be P(n).
This cannot be solved using Master Theorem, so we approximate it as T(n) = O(P(n) * $2^{n/4}$) since T(n-4) term has a greater contribution compared to P(n) in computation of T(n).