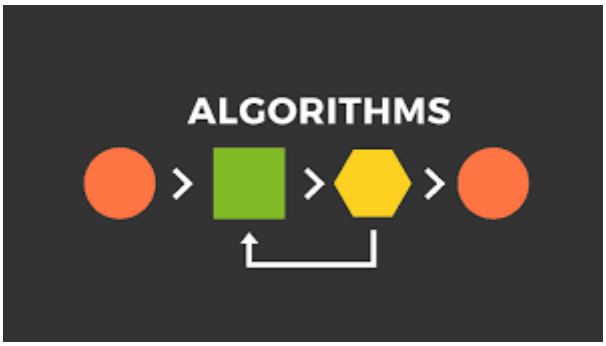


Divide & Conquer

MD Atiqur Rahman | 180041123 | CSE 1 | CSE 4304 | 6 oct,2020

Problem 1:

MD Mushfiqul Haque – 180041140

Farhan Ishmam – 180041120

Nahian Ibn Asad - 180041136

Problem 2 :

What is uppertangent?

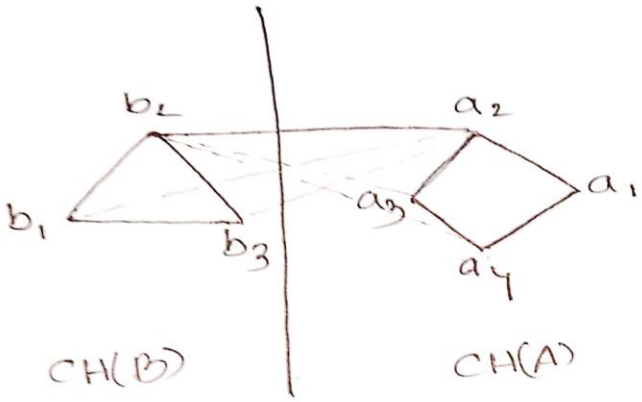
Given that, CH(A)={ a1,a2,a3,....,ap} and CH(B)={b1,b2,.....,bq}, which means we have a convex hull A with {a1,a2,a3,.....,ap} points and another convex hull B with {b1,b2,.....,bq} points. Now if we join any points of convex hull A with any points of convex hull B, we will get straight lines connecting every combination of (ai,bj) points, here ai means all points of CH(A) and bj means all points of CH(B). Now the line which is above all the other lines is called the uppertangent.

Now, let's come to the argument. Given that, there is vertical line L between those convex hulls CH(A) and CH(B). As we are connecting every points we can easily depict that all the line segment (ai,bj) will intersect the line L and the y co-ordinate of the intersection points are y(i,j). Is y(i,j) is the maximum among all the other points if and only if line segment (ai,bj) is the uppertangent?

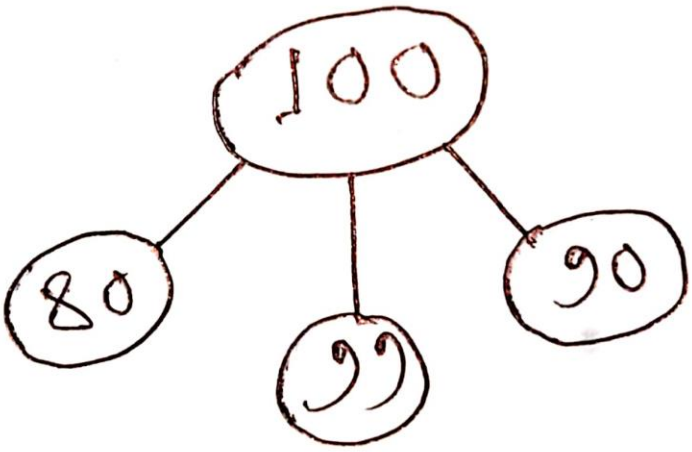
Let's assume line segment (ai,bj) is the uppertangent, then by the defination it is above all the other lines and y(i,j) must be the maximum one else (ai,bj) can't be the uppertangent.

Again, let's say y(i,j) is the maximum one then it is impossible to have uppertangent other than (ai,bj) cause if (ak,bq) is the uppertangent then y(k,q) is the maximum one which is not.

Therefore, we can say that (ai,bj) is the uppertangent if and only if y(i,j) is the maximum one.



Problem 3.a :



If we follow the given greedy algorithm, then we will get a sorted list like,

100 99 90 80.

Now if we add 100 to set O and delete all it's neighbour 99,80 and 90, then V is empty. So our answer will have set O which has only 100. But we can have (99+80+90)=269>100 as 80 99 and 90 are not neighbour.

Therefore, for the above example the given greedy algorithm doesn't work.

Problem 3.b :

As the above graph G=(V,E) is an undirected acyclic graph, we can rephrase it to a tree. So we have to find out the subset of vertices which are not adjacent (where is no child parent relationship) but the sum of the profit assigned to those vertices will be maximum.

If we think then we will find that it is a lot like that problem where we have to find the maximum sum of the elements of the given array and the elements can't be adjacent. Like 10 2 3 20 is given the maximum sum will be 30 as we can't take the adjacent elements. But the problem given is about tree. So if the tree is 1-D like 10->2->3->20 where -> is denoting an edge then the answer will be same 30. So our approach should be same like before, we will just child-grandchild relationship in between.

First, let's think about the solution of the array, which is relatively easy.

Dp[i] = max(dp[i-1] , a[i] + dp[i-2]) here we are taking that element or not. How are we differentiating between them? We will take it and the one which is two step before it (as two step before element is clearly not the adjacent element) or we will not take it and take the adjacent one.

If we insert child-grandchild relationship in the above idea we will get, we will take a vertex and all of it's grand children , we will again only take the children of that vertex and save the maximum one between the two of them in dp[that vertex we have taken at first]. This way though we will not be taking any adjacent vertex ,we will be taking the maximum one.

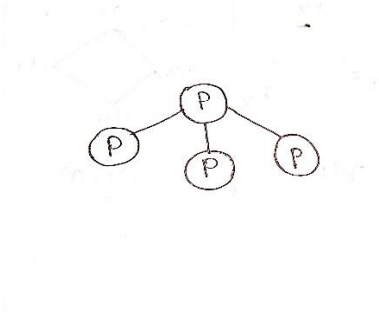
Therefore, our approach will be dp and we will start counting from leaf nodes and we also know who is whose child. So we will be going from leafnode to the root. Hence dp[root] is our answer.

Our algorithm is ,

1. We will run a bfs and find every vertex's children and store it in a 2D vector like child[v]={child1,child2,child3...,childn} and we will also push every vertex we are visiting in a stack.
2. So, the vertices we are visiting last are the leaf nodes and at the top of the stack. We will pop a vertex v and do the following untill the stack is empty,
   dp[v]=max((sum of dp[child[v][i]] where i=0 to number of children of v) , (Pv + sum of dp[child[child[v][i]][j]] where Pv is the profit of v and i=0 to number of children of v and j=0 to number of children of child[v][i]))
   At first we are taking only the children of v or we are taking v and it's children's children which are the grandchild of v.
3. Lastly print dp[root] cause it's our answer maximum profit without taking any adjacent vertices.

Problem 3.c :

Let's think about a tree with two levels.



So, here all vertices have same profit, so will take the maximum number of vertices so that those vertices are not adjacent to each other. How can we take it? Let's take all the leaf node's and not the parent of all those leaf nodes. Yes it is the correct answer only for the trees which have exactly two levels. Now how will we implement this? We will start from leafnode and for that leafnode we will not take it's parent.

But if there are more than two levels what will we do? Let assume that we have a indicator which tell us if we have taken a vertex or discarded it. For every discarded vertex we will not mark it's parent as discarded. Cause if a vertex is not taken we can take it's adjacent vertex like it's parent and it's children. After traversing all the vertex we will count the number of vertex which are not discarded and it will be our answer.

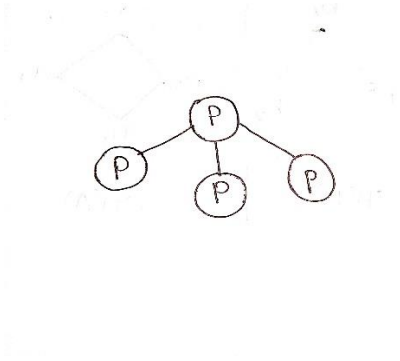
So our algorithm will be,

1. We will run a bfs and find every vertices' parent and push every vertex which we are visiting in a stack.
2. So the leaves are at the top in the stack. We will pop a vertex v and do the following untill the stack is empty,
   If v is not discarded then we will discard it's parent.
3. Count the not discarded vertices and print the counted value.

PSUEDO CODE:

```
stack<lld> st
lld bfs(lld source)
queue<lld> q
q.push(source)
while(! q.empty())
  lld v=q.front()
  q.pop()
  visited[v]=true
  st.push(v)
  for(lld i=0;i<adj[v].size();i++)
       if(visited[ adj[v][i] ]==false)
         par[ adj[v][i] ]=v
         q.push( adj[v][i] )
while(! st.empty() )
  lld v=st.top()
  st.pop()
  if( flag[v] == false)
     flag[ par[v] ] = true
lld ans=0
for(lld i=1; i<=n ; i++)
     if( flag[i] == false ) ans++
return ans
```

CORRECTNESS :



Let's get back to the first example we have given. Now we know every parent must have atleast one child to be a parent, normal logic. But we can only take either the parent or the child not both. So if we take the child not the parent or the parent not the child the end result will be same for both of the cases. As 1=1. But if the parent has children (more than one) then it is better to take the children rather than parent as parent(1)<children(more than one by defination). So we will take the child or children and discard the parent.

We are following this optimal substructure for all the cases from the leaves to the root. As we know if the solutions for all the sub problem are correct than the overall solution of the main problem is also correct.

Hence our algorithm is correct.

Problem 3.d :

As our tree has become graph by cycles, we can't find the solution in dp approach, cause we can only appy dp when there is a DAG. Again we also can't go with greedy approach cause there will be one state which will depend in another unsolved state for it's own solution.

So to find answer we will go with brute force approach and find all 2^v combinations and find the maximum one among them.