ISLAMIC UNIVERSITY OF TECHNOLOGY

Board Bazar, Gazipur, Dhaka, 1704, Bangladesh

ASSIGNMENT

Course No.: CSE 4404

Name of the Assignment: **Divide and Conquer**

Name : **Syed Rifat Raiyan**
Student ID : **180041205**
Department : **Computer Science and Engineering (CSE)**
Lab Group : **2A**

Date of Submission : 1st November, 2020.

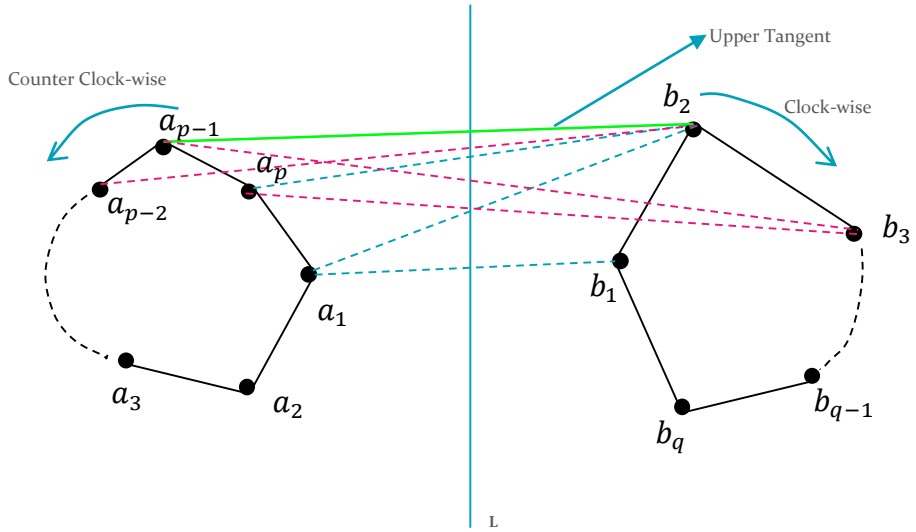
Student ID of Partners: **N/A.**

Problem 1: Collaborators

None.

Problem 2: Go Off on a Tangent

The figure assumed in the problem statement looks like:



Here, a_1 and b_1 are the right-most and left-most points respectively, and the polygons formed by points $a_1, a_2, \dots, a_p$ and $b_1, b_2, \dots, b_p$ are convex hulls themselves.

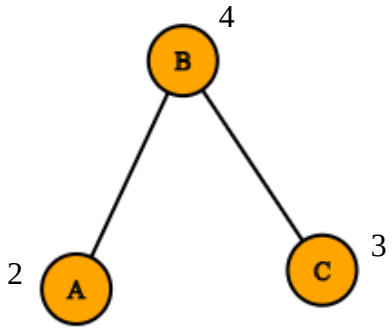
The simplest and most obvious argument as to why the given statement is **valid** is that, if the pair of points a_i and b_j is such that rotating from either side results in decreasing of the value of $y(i,j)$, then there must be no other points of either polygon $CH(A)$ or $CH(B)$ above the former straight line $(a_i — b_j)$. So, the pair of points that were considered prior to this must be the two points of the required upper-tangent. And, since we are checking to see if $y(i,j)$ is decreasing or not, then the pair of points that were considered prior to the last two breaking condition checks must have generated the maximum $y(i,j)$ value.

In the given figure, it can be seen that the breaking condition is satisfied when both lines $(a_{p-2} — b_2)$ and $(a_{p-1} — b_3)$ both yield $y(p-2, 2)$ and $y(p-1, 3)$ which are values that are less than $y(p-1, 2)$.
So, in the case of a "convex" hull, proceeding further is pointless, since the angle between adjacent sides of a convex polygon is $< 180°$. The correct upper-tangent is the line $(a_{p-1} — b_2)$.

This approach of merging two disjoint Convex Hulls is known as the Two Finger Algorithm.

Problem 3: GiveIn: Shakes, Fries, Burgers

(a) A counter-example is shown below:



Here, the profits are $p_A = 2$, $p_B = 4$ and $p_C = 3$. After sorting the nodes in the descending order of their profit, we get, $V = \{B, C, A\}$.
Now, following the steps of the given algorithm, we have to insert B in the set O and delete its neighbors from V. V becomes an empty set and the loop is broken.

$$O = \{B\}$$

The maximum profit obtained is $ans = p_B = 4$, since B is the only node is the set O.

This approach is clearly wrong. The optimum choice would've been picking the non-adjacent nodes A and C.

$$O = \{C, A\}$$

The maximum profit earned becomes, $p_A + p_C = 2 + 3 = 5$.

(b) Let's use Dynamic Programming to solve this.
At first, make the set of nodes V topologically sorted. A simple DFS traversal taking an arbitrary root x should work. Now, just reverse the set to ensure that the algorithm starts with the leaf nodes. Assume that the graph is already represented using an adjacency list. Here, for each node u, let's define,
$dp1(u)$ ≡ Maximum profit obtained from subtree whose root is u, including u.
$dp2(u)$ ≡ Maximum profit obtained from subtree whose root is u, excluding u.

Now, let's establish a recurrence relationship between them.
For each node $v \epsilon V$

If v is a not a leaf node

$$dp1(v) = p_v + \sum_{i=0}^{adj[v].size()-1} dp2(adj[v][i])$$
$$dp2(v) = \sum_{i=0}^{adj[v].size()-1} max\ (dp1(adj[v][i]), dp2(adj[v][i]))$$

Else

$$dp1(v) = p_v \quad \textit{[Base Cases]}$$
$$dp2(v) = 0$$

The final answer will be $\max(dp1(x), dp2(x))$.

The algorithm works with the child nodes before the parent nodes due to the reverse DFS traversal alignment we ensured in the set of nodes V. The recurrence relationship can be explained in simple words as, we will either choose a node and exclude its parent node and proceed to explore further above, or we won't choose a node and proceed to explore its parent node. Along the process, the maximum profit is recorded and updated.

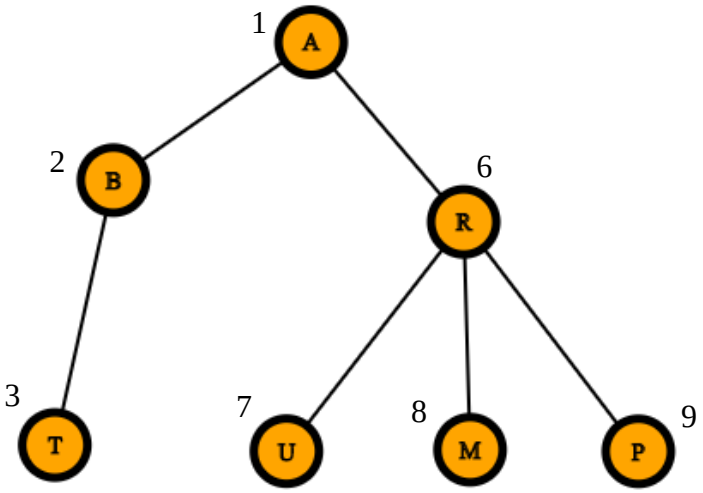
The resultant set of nodes O can also be recursively determined. Starting from the root x, we can insert x into the set only if we can yield a higher profit by opening a *GiveIn* in the node x. That is, if $dp1(x) > dp2(x)$ then x should be in the resultant set. Otherwise, we can exclude x. Repeat the same process with x's grandchildren if x is selected, and if x isn't selected, repeat the same process with x's child nodes.

Let's analyze the time complexity.

Here, for DFS the time taken is $\boldsymbol{O(V+E)}$, where E is at most $V-1$. The time to generate `dp1[]` and `dp2[]` memo tables is proportional to the aggregate of the degrees of all the nodes, which results in a complexity of $\boldsymbol{O(E)}$. The overall complexity now becomes,

$\boldsymbol{O(V+E+E) \approx O(3V-2) \approx O(V)}$; $\boldsymbol{V}$ is the size of the set of vertices.

Let's take an acyclic graph (arranged as a tree), and see how the algorithm works on it.



The DFS Traversal is **A B T R U M P**, and after flipping, it becomes **P M U R T B A**.

Now, after traversing through this list, the final memo table looks something like this:

Memo \ Nodes	**A**	**B**	**M**	**P**	**R**	**T**	**U**
$dp1$	28	2	8	9	6	3	7
$dp2$	27	3	0	0	24	0	0

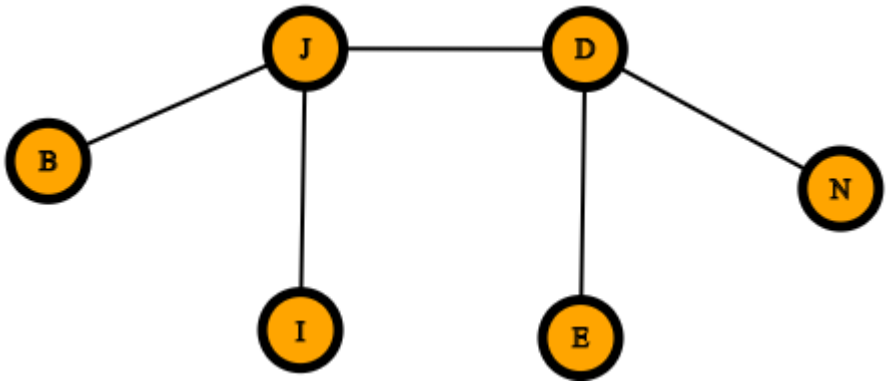
Here the answer is $max(28, 27) = 28$ since we considered A as the root node.

So, opening *GiveIn* restaurants in the nodes A, T, U, M and P will result in the maximum profit which is $1 + 3 + 7 + 8 + 9 = 28$.

(c) Performing DFS and then iterating through V should work for this case as well.
Just like in (b), rearrange the set of nodes V in the reverse DFS traversal order. This ensures that the starting node must be a leaf node.
For each node $v \epsilon V$
Insert v to set O
Delete parent of v from set V [if the parent is present in the set, of course]

Let's try to prove the claim that the aforementioned greedy algorithm will work.
Since **all the nodes have equal profit margins**, we can say for sure, that **the resultant set O must contain all the leaf nodes**. (Lemma)
And the order in which we are working with the nodes ensures that the leaf nodes are coming first and we can insert them to O and delete their parent nodes from the set V.
After every iteration of the for-loop, we delete the parent nodes, thus guaranteeing that the last node of the shrunk set V is a leaf node considering the order in which we arranged the set V in the first place.
So, this algorithm will work. Let's analyze its time complexity. At first, we perform DFS in $\boldsymbol{O(V+E) \approx O(V+V-1) \approx O(V)}$. The overall complexity of the insertion and deletion for all $|V|$ nodes is $\boldsymbol{O(V)}$ in the worst case.
So, the overall time complexity of the whole algorithm is $\boldsymbol{O(V)}$.

Let's try out this algorithm on an example as well.



The DFS Traversal, taking J as the root node, is **J B I D E N**, and after flipping, it becomes **N E D I B J**. After following the steps of the aforementioned algorithm, the set O looks something like:

$$O = \{N, E, I, B\}$$

Since all the nodes hold equal values, the maximum profit will be the size of the set O, which is 4.

(d) Since the question isn't asking for an efficient solution, the algorithm that I am going to suggest involves checking all possible subsets of nodes of V.
Keep a boolean unordered map/2D array to keep track of which 2 nodes have an edge between them *i.e.* `edge[i][j]=true, edge[{i,j}]=true`.
Create all possible $2^{|V|}$ subsets of the mother set V. Let's create a Power Set P consisting of all the found subsets. Now, the algorithm will check every subset and see if the constituent nodes are non-adjacent or not. If the set turns out to be valid, record the size of the set in a variable. If not, then discard it. Then, proceed to the next subset and continue until all $2^{|V|}$ iterations are completed.

$ans := \emptyset, maxm := -\infty, sum := 0$
For each subset $s \epsilon P$
    For each node uϵs
        For each node $v \epsilon s$
            If $edge[\{u, v\}] == false$
                Continue;
            Else
                Go to $bottom$;
    $maxm = \max(maxm, s.size())$; //Assuming all nodes have equal profits. $O(1)$
    For each node $u \epsilon s$ //Assuming nodes with different profits. $O(V)$
        $sum += p_u$;
    $maxm = \max(maxm, sum)$
    If $s.size() > maxm$
        $ans = s$;
    $bottom$:

Here, I have assumed that the nodes have equal values. If this is not the case, then, an additional $\boldsymbol{O(V)}$ time has to be spent to calculate the total profit yielded from the subset in the worst case. There is a total of E edges, so in the worst case, the inner for-loops will work in $\boldsymbol{O(E)}$ time. All the other operations take constant time.
So, the overall complexity of this problem is $\mathbf{O(2^{|V|} \times (|E| + |V|))}$, where |E| can be at most $\frac{|V|(|V|-1)}{2}$ in the case of a Complete Graph. So, the asymptotic time complexity can now be written as $\boldsymbol{O(2^{|V|} \times |E|)}$. This is a Non-Deterministic Polynomial Time Complete (NP-Complete) Algorithm.

—×—