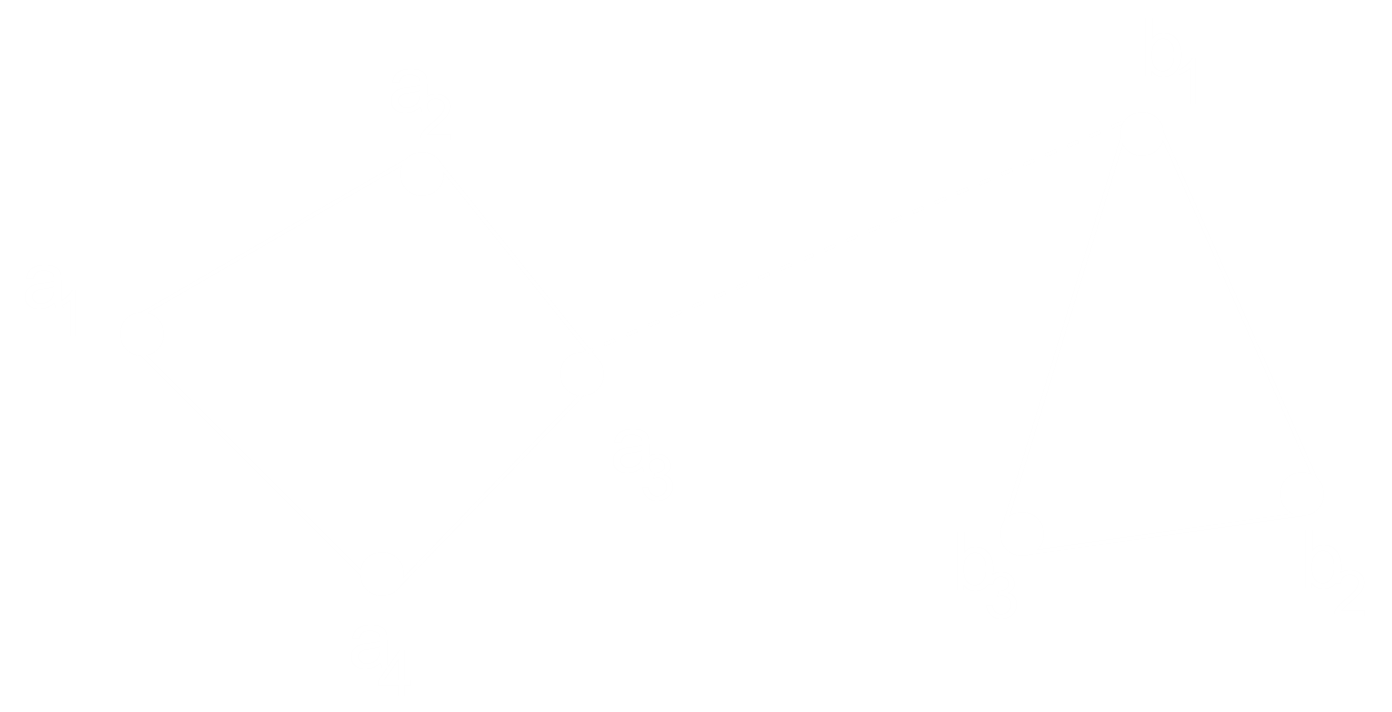
**Problem 1: Collaborators**

**Problem 2: Go Off on a Tangent**

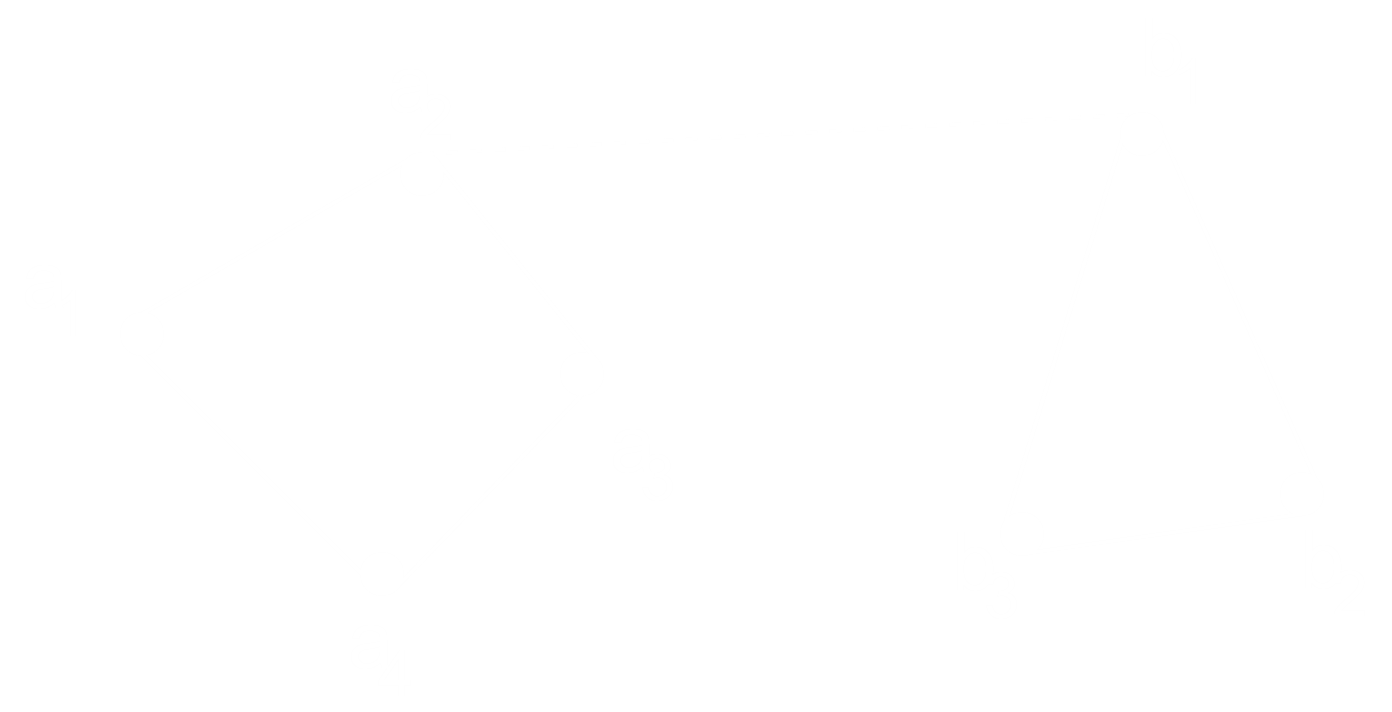
The concept of a convex hull is that it bounds all the points inside it. If we consider to be the upper tangent such that it does not maximize , then there will be points that are above this line, which means they would not be bound by the resulting ‘convex hull’.

It is easiest to show this using an example. Consider the figure below:



Quite obviously, is not the upper tangent. The point for example, is above this line, which means it is not bound by the ‘convex hull’ created. More importantly to this argument, the point of intersection of this line and the vertical line separating the two parts is not the highest possible point, i.e. is not maximized here.

Now consider this figure instead:

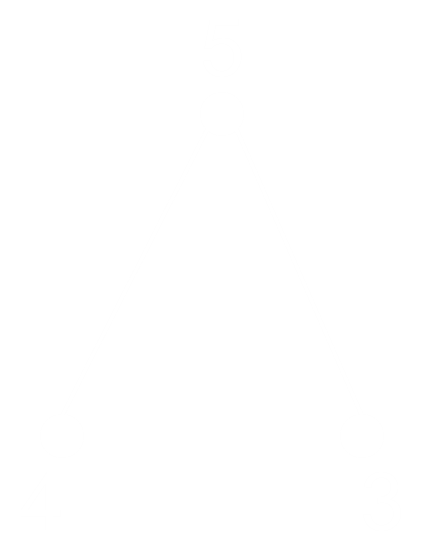


It is clear that this is the correct upper tangent, since all points on the figure are below this line, which means they are guaranteed to be bound by the resulting convex hull. As can be seen, is also maximum in this scenario.

**Problem 3: GiveIn: Shakes, Fries, Burgers**

a)

Consider the graph below:



For this graph, the vertices sorted in descending order are:

According to the suggested greedy approach, we take the vertex with the highest profit and remove all its neighbours. In this case, we would take and remove and . However, this will fail to give us the maximum total value. If we had not taken , we could have taken and , since they are not adjacent to each other. This would have given us a total profit of , the maximum profit.

b)

We can solve this problem by making a decision about whether or not to include each vertex as we traverse over it.

Since we are assuming the graph is acyclic, we can consider any node to be the ‘root’ and start working from there. At the -th node, which has the value we can either

* Choose to not take the node, which would allow us to check all of its child vertices, a set we can denote as

or

* Choose to take the node, which would forbid us from checking any of the nodes in , but would allow us to check all the nodes that are children to the nodes in , i.e. the grandchildren of the -th node. Let these nodes all be in a set denoted by .

Thus,

i.e. the result is the maximum value obtained by either calling the function recursively on all of the children and not taking the value of the node itself, or by taking the value of the node itself and calling the function recursively on all the grandchildren. If the prior is taken, all the children are added to and if the latter is taken, all the grandchildren and the node itself are added to .

The base case in this scenario would be a node that has not children. Such a node would just return its own value.

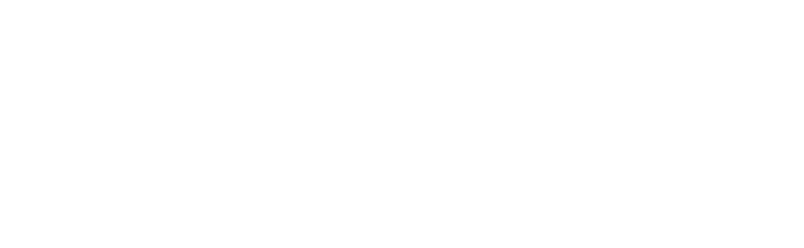
c)

First, we sort the vertices in ascending order based on the number of adjacent vertices it has. We take the first vertex in this sorted list, add it to , and remove all its immediate neighbours. This first vertex is a ‘leaf’ node. While removing the neighbours, we add the vertices that are adjacent to each of those neighbours to a queue, since we will be using these next. After all the neighbours of the first vertex have been removed, we repeat this entire process on each of the vertices in the queue we just created.

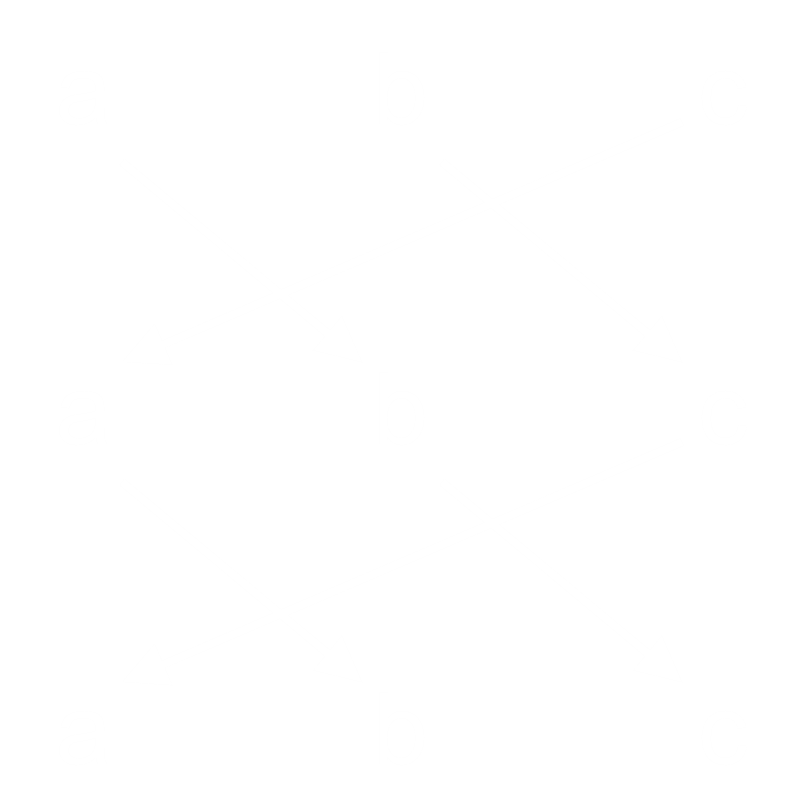
d)

We can create an acyclic version of the graph and then use the same algorithm that we have already created.

Consider this cyclic graph:



We can make a acyclic by creating copies of the graph, where is the number of nodes. In this new set of graphs, we make the same connections, but only from one level to the next. This is shown below:



Now we can apply the same algorithm that we already have. The only shortcoming is that the time complexity will increase. The algorithm we had before had subproblems for each of the vertices, with each subproblem taking constant time. Thus, the algorithm had a time complexity of . After making this change, there will be subproblems, so the algorithm will have a time complexity of .