Question 1 *Yellow-Purple Tree*

[7 marks] You are given an unrooted tree with n vertices. Each vertex has a colour, yellow or purple. Your task is to design an algorithm in pseudo-code which finds the longest path with the same colour in $O(n)$ time. []

```
Function find_longest_path(v):
    visited[i] ← False, i = 1...n
    path_start, path_longest ← find_longest_path_recursive(v)
    return path_longest
Function find_longest_path_recursive(v, visited):
    A[i] = null, i = 1, 2
    cur_longest_path = null
    foreach unvisited adjacent vertex u do
        path_start_u, path_longest_u ← find_longest_path_recursive(u)
        if color(v) = color(u) ∧ length(A[i]) < length(path_start_u) for first i ∈ [1, 2] then
            Replace A[i] with path_start_u
        if length(path_longest_u) > length(cur_longest_path) then
            cur_longest_path = path_longest_u
        visited[u] ← True
    path_inclu_v ← append(A[0], v, A[1])
    if length(nPath) > length(cur_longest_path) then
        cur_longest_path ← path_inclu_v
    return append(v, A[0]), cur_longest_path
```

We treat the vertex we entered from as root. The algorithm uses a modified DFT. For each vertex v, there are two sub-problems.

[A] P1. Longest path with same color in this sub-tree that starts at v (longest path downward starting from v).

[B] P2. Longest path with same color in this sub-tree (doesn't need to be a path that starts from v).

By solving P1 and P2 for all child, we can solve P1 and P2 for v. Let m1 and m2 be the solution for P1 and P2 respectively. For m1(v), choose the longest m1 from all child that has the same color and append v. For m2(v), it is either a path that includes v or not include v. We choose max{the longest m2 from all child, *path_inclu_v*}, where *path_inclu_v* is the longest path that include v, created by appending v and the two longest m1 from all child that has the same color. Our wrapper function returns m2 of the root, which is the longest path with the same color of this tree.

Assume checking color of nodes is $O(1)$ and using edge representation, we can use a doubly linked list for storing path so checking path length and appending two paths is $O(1)$. Array A in the algorithm stores the two pointers for the two longest m1 from childs that have the same color in descending order. The two return values from the recursive function is simply m1 and m2 of v. Since this algorithm is a modified DFT and the DFT of a tree is $O(n)$, this algorithm is $O(n)$

Question 2

[10 marks] You have n identical marbles, to be divided into groups. All groups must be of different sizes. For example, if $n = 6$, there are three ways:

- a group of 5, a group of 1, and a group of 0,
- a group of 4, a group of 2, and a group of 0, and

- four groups of 3, 2, 1 and 0.

Let $q(n)$ be the number of ways to divide n marbles in this way.

2.1 [6 marks] We want to build a dynamic programming algorithm to compute $q(n)$. Define the subproblems, the base cases and the recurrence relation between the subproblems, and explain the recurrence relation in words.

From the updated version of this question mentioned in forum, I will assume for $n = k$, a division of a single group k is allowed, and there is no group of 0 (except in base case mentioned later). So in the question above, a group of 6 is also a valid division and there are four ways of dividing $n = 6$, $\{(6), (5, 1), (4, 2), (3, 2, 1)\}$.

Define sub-problem $S(i, k)$ as the number of ways of dividing i marbles which must contain a group of k, where k is the largest group for the way of dividing, let $m(i, k)$ be the solution of $S(i, k)$.

The base case is $m(0, j) = 0 \; \forall j = [1, n]$, $m(j, 0) = 0 \; \forall j = [1, n]$, and $m(0, 0) = 1$. Obviously, there is no division that divides 0 marbles that contains a group larger or equal to 1, and there is no division marbles greater or equal to 1 marble but the maximum group is 0. We treat $m(0.0) = 1, \{(0)\}$ as a valid division for 0 marbles to make the recurrence relation work.

The recurrence relation is $m(i, k) = \sum_{j=0}^{k-1} m(i - k, j)$, with any out of bounds cases equal to 0, i.e. $m(i, k) = 0, \forall i, k < 0$. Number of ways of dividing i marbles that has the largest group k is equal to the total number of ways to divide $i - k$ marbles with largest group being smaller than k, as we simply add a group k to such division $i - k$. Notice that the divisions accounted in each $m(i, k)$ is different (divisions of different i means groups sum up to different value, and divisions of different k means they have a different maximum group), this means adding different $m(i, k)$ won't result in repeated counting.

We can solve this using 2D array. The table shows $m(i, k), 0 \leq i \leq n, 0 \leq k \leq n$ for the first $n = 8$, row is i and column is k:

	0	1	2	3	4	5	6	7	8
0	1	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0
3	0	0	1	1	0	0	0	0	0
4	0	0	0	1	1	0	0	0	0
5	0	0	0	1	1	1	0	0	0
6	0	0	0	1	1	1	1	0	0
7	0	0	0	0	2	1	1	1	0
8	0	0	0	0	1	2	1	1	1

To find $q(n)$, we add up all columns in the row number n, i.e. $q(n) = \sum_{j=0}^{n} m(n, j)$

2.2 [2 marks] What would be the worst-case time complexity of an algorithm based on your approach? Justify your answer.

There are $(n + 1)^2$ cells (sub-problems) in our 2D matrix (including row 0 and column 0), worst case for solving a sub-problem (sub-problems at column n) is $O(n)$ as it sums up a row of values. This takes $O(n^3)$. Finding $q(n)$ is $O(n)$ as it sums up the last row. Thus the worst case time complexity is $O(n^3)$. This is a pseudopolynomial time as n is the numerical value, not in terms of number of bits.

2.3 [0 marks] Find a closed-form formula for $q(n)$ and prove rigorously that it is correct. If Question 2.3 is completely correct, you do not need to provide separate solutions for Question 2.1

and Question 2.2.

Question 3

[9 marks] In this exercise, we are interested in building "good" supersequences of a special kind of input sequences of characters.

Given a reward/cost matrix m associating to every pair of characters a value, we define the *similarity* σ of two sequences A' and B' of the same length $\ell = |A'| = |B'|$ as:

$$\sigma_m(A', B') = \sum_{i=1}^{\ell} m(A_i, B_i)$$

A sequence A' is *#-inserted supersequence* of a sequence A if it contains the same characters as A in the same order, and also possible # characters anywhere in it. For instance $A' = \#\#aba\#c$ is a #-inserted supersequence of $A = abac$ and $B' = \#b\#\#aa\#$ is a #-inserted supersequence of $B = baa$.

Assume the following cost matrix: $m =$

	#	a	b	c
#	−10	10	10	15
a	10	5	40	−30
b	10	40	80	10
c	15	−30	10	20

we can compute the similarity between A' and B' as

$$\begin{aligned}\sigma_m(A', B') &= m(\#, \#) + m(\#, b) + m(a, \#) + m(b, \#) + m(a, a) + m(\#, a) + m(c, \#) \\ &= -10 + 10 + 10 + 10 + 5 + 10 + 15 \\ &= 50.\end{aligned}$$

Another way to visualize this is with the following table.

A'	#	#	a	b	a	#	c	total
B'	#	b	#	#	a	a	#	
m	−10	10	10	10	5	10	15	50

3.1 **[1 mark]** Reusing the example cost matrix and sequences above, find two #-inserted supersequences A^+ and B^+ of A and B respectively such that $\sigma_m(A^+, B^+) \geq 120$. In your answer, provide the two supersequences and their similarity. You do not need to explain how you found them. You can use any method you want, including guessing, manual computating, programming, but not including plagiarising.

$A^+ = ab\#\#ac$
$B^+ = \#baa\#\#$
$\sigma_m(A^+, B^+) = 135$

3.2 **[1 mark]** For any two finite, non-empty, sequences A, B, and cost matrix m, we are interested in maximizing σ_m. That is, we want to find the largest value M such that there exist #-inserted supersequences A^* and B^* such that $M = \sigma_m(A^*, B^*)$. Under which conditions on A, B, m is M guaranteed to be a finite number? If you think M is always guaranteed to be finite then just write "no particular condition is required" as your answer.

Assume the entries of the cost matrix is finite as they are inputs to the program. Then $M = \infty$ when $m(\#, \#)$ is positive. If $m(\#, \#)$ is positive, then we can pad A and B indefinitely. $M \neq -\infty$

for any sequence A and B as we can create A' and B' of length $|A|+|B|$), where each character in a sequence is paired with a # in another sequence. As we assumed the entries of the cost matrix is finite, $\sigma_m(A', B')$ must be finite and cannot be $-\infty$.

3.3 [7 marks] Design an algorithm that takes as input a cost matrix m and two finite sequences A and B and outputs the largest value M as defined previously. If M is unbounded, then your algorithm should just output $+\infty$ or $-\infty$ accordingly. To get full marks, your algorithm needs to run in $O(n^2)$ where $n = \max(|A|, |B|)$. Describe it in plain English.

First, check if $m(\#, \#)$ is positive. $M = \infty$ if $m(\#, \#)$ is positive. Define $A(n)$ as the sub-sequence of A of size n built from the first n characters (Our array starts at index 1, that means $A(1)$ is a sequence with 1 character $A[1]$, $A[0]$ is the empty character and $A(0)$ is the empty sequence). We need to define $A(0)$ as this will be helpful when preforming the recurrence relation.

Define sub-problem $S(i, j) = \sigma_m(A(i), B(i)), 0 \leq i \leq |A|$ and $0 \leq j \leq |B|$. Define the solution to $S(i, j)$ as $s(i, j)$. The base case is $s(i, 0) = \sum_{k=1}^{i} m(A[k], \#)\ \forall i \in [1, |A|]$, $s(0, j) = \sum_{k=1}^{j} m(\#, B[k])\ \forall j \in [1, |B|]$ and $s[0, 0] = 0$. If we have a non empty sequence and an empty sequence, then to get $\sigma_m = M$ we pad the empty sequence with # until the two sequence are of same size, and note that padding more # will not increase σ_m as $m(\#, \#)$ is non-positive. Obviously, two empty sequence will have $M = 0$.

The recurrence relation for non base case is

$$s(i, j) = max \begin{pmatrix} m(A[i], \#) + s(i-1, j) \\ m(\#, B[j]) + s(i, j-1) \\ m(A[i], B[j]) + s(i-1, j-1) \end{pmatrix}, 1 \leq i \leq |A|, 1 \leq j \leq |B| \tag{0.1}$$

For sub-sequence $A(i)$ and $B(j)$, three possible cases can occur for the last entry pairing of $A(i)^*$ and $B(j)^*$, it is either $(A[i], \#)$, $(\#, B[j])$ or $(A[i], B[j])$. $(\#, \#)$ is not possible due to reasons mentioned above. Thus we only need to solve the sub-problem of the sub-sequence for all three cases to find $s(i, j)$.

We can solve this using 2D array. The following table shows how the algorithm finds M for question 3.1. Each cell is $s(i, j)$, row is i and column is j.

	0	1	2	3
0	0	10	20	30
1	10	40	50	60
2	20	90	100	110
3	30	100	110	120
4	45	115	125	135

Notice that the algorithm is never out of bound for non base cases, as $s(i, j)$ only needs to look at top, left and top-left entries.

There are $(|A| + 1)(|B| + 1)$ sub-problems to solve, and solving $s(i, j)$ is $O(1)$ given previous sub-problems are solved. This algorithm runs in $O((|A| + 1)(|B| + 1))$, which is $O(n^2)$

Question 4

[8 marks] You are given a tree with n nodes. In this tree, each edge has a given cost and each leaf has a given reward. You can spend a cost to activate an edge. A leaf is activated if and only if there is a path to it from root via activated edges only. Define P to be the profit (the total reward of activated leaves minus the total cost of activated edges). Design an algorithm in plain English that finds the maximum number of activated leaves with non-negative P in $O(n^3)$.

The algorithm solve the problem by pruning the tree. It finds the profit of the tree if all edges are activated, which will most likely be negative, then finds the profit gained by pruning the tree in a way that removes the least number of leaves. The algorithm stops when the profit gained by pruning the trees exceeds the debt of activating all edges. The implementation of the algorithm is in two parts:

Assume all edges are activated. We can prune an edge e by deactivating e and all of it's child edges. The algorithm first finds the profit gained by pruning e, denoted by $p(e)$, and also finding the number of leaves removed $r(e)$ as a result of pruning e. All values can be found using a modified depth-first-traversal. When traversing into a vertex v, the edge we came from (the edge connecting v and it's parent) is passed as argument, denote this edge as i and it's cost is some negative number $c(i)$. Then $p(i) = (\sum p(k)) - c(i)$ and $r(i) = \sum r(k)$, where k are the child edges connecting v. When traversing to vertex v which do not have any child (leaf node), then $p(i) = -(c(v) + c(i))$ (we assume $c(v)$ is positive) and $r(i) = 1$. In addition, we need to obtain $s(i)$, the number of edges removed as a result of pruning i, which is simply $(\sum s(k)) + 1$, where k are the child edges connecting v. We also want to find the number of leaves $|L|$. We also want the list E, which is the edges visited in post-order of the tree. This can all be done while traversing. Finally, we obtain debt d for activating all edges (the debt without pruning any edges), this is simply $\sum p(i)$, where i is edges connecting the root, if d is found to be negative, then the algorithm terminates and return $|L|$. Since this part is just a modified DFT on a tree, it runs in $O(n)$.

The values we obtained from the traversal:

- [A] Array p, where $p(e)$ =profit gained for pruning e
- [B] Array r, where $r(e)$ =number of leaves removed as a result of pruning e
- [C] Array s, where $s(e)$ =number of edges removed as a result of pruning e
- [D] List E, which contains the edges in post-order of the tree (Any ordering that guarantees child edges directly precedes parent edges in E is sufficient).
- [E] Value $|L|$, which is the number of leaves
- [F] Value d, which is the debt for activating all edges

The algorithm now decides which edges to prune. This can be done by a method similar to 0/1 knapsack without duplicate, where each edge is equivalent to an item in knapsack. Profit gained by pruning e equivalent to value gained for picking an item, $r(e)$ equivalent to weight of an item, and we can limit maximum leafs removed (capacity in 0/1 knapsack). The difference between picking items and pruning edges is that we won't be able to prune edges once it's parent edges is already pruned. However, this won't be an issue by the way we defined the edge list.

Define $E(k)$ to be the list that contains first k edges in E, and e_k to be the last element in $E(k)$. Notice that all child edges of e_k must be in $E(k)$, and in particular directly preceding e_k as E is in post-order. To remove the edge e_k and all of it's child edges from $E(k)$ (i.e. to prune e_k in $E(k)$) is same as finding $E(k - s(e_k))$, which is the list that exactly excludes e_k and all of it's child edges. This ensures the algorithm can't prune edges when it's parent edge is already pruned.

Define our sub-problem $M(k, i)$ as the maximum profit obtained by pruning some edges in $E(k)$ such that at most i leaves is removed, and let $m(k, i)$ to be it's solution. The base case is $m(k, 0) = 0,\ 0 \leq k \leq |E|$, as pruning any edges results in removal of at least 1 leaf, the maximum profit gained in which we don't remove any leaves is 0. Another base case is $m(0, i) = 0,\ 0 \leq i \leq |L|$, as $E(0)$ is the empty list and not pruning any edges has maximum profit 0.

The recurrence relation for non base case is

$$m(k, i) = max \begin{pmatrix} m(k-1, i) \\ m(k - s(e_k), i - r(e_k)) + p(e_k) \end{pmatrix}, 1 \leq k \leq |E|, 1 \leq i \leq |L| \tag{0.2}$$

To find $m(k, i)$, the maximum profit gained by pruning edges from $E(k)$ with at most i leaves removed, we need to consider two cases. Either e_k is not pruned or is pruned. If e_k is not pruned (first case in *max* function), then the solution is same as $m(k-1, i)$. If e_k is pruned (second case in *max* function), then the maximum profit is equal to profit gained by pruning e_k plus the maximum profit gained by pruning edges from $E(k - s(e_k))$ with at most $i - r(e_k)$ leaves removed. It have covered all possible cases by choosing the maximum of these two cases. For out of bounds cases, $m(k, i) = -\infty, k < 0, i < 0$ so it is never chosen in the recursion.

The algorithm first finds $m(k, i)$ for all k increasingly with $i = 0$ (base case), then repeat with $i = 1, i = 2...$ and so on. After finding the first $m(k, i) \geq d$, return $|L| - i$ as it is the maximum number of activated leaves with non-negative profit.

There are $(|E| + 1)(|L| + 1) < n^2$ sub-problems, each requiring $O(1)$ to solve assuming previous sub-problems are solved, so this part is in $O(n^2)$. Combining with the first part, the algorithm runs in $O(n^2)$.