**Take-home final**

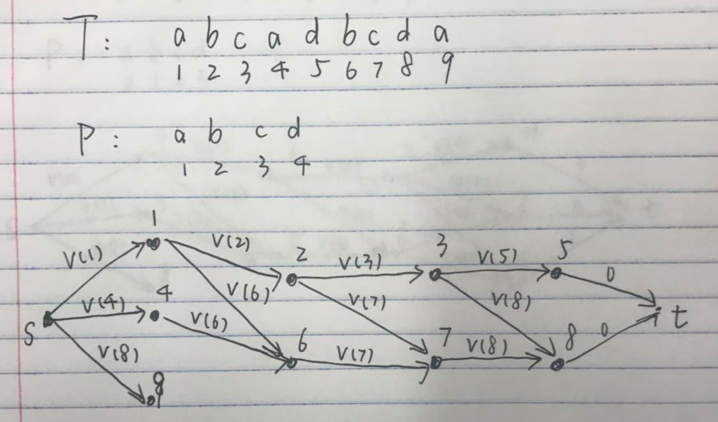
Wenjun Zhang

A53218995

**Problem 1: Sequence matching**

**Algorithm description:**

We can convert this problem into finding the largest weighted path in a DAG. To begin with, we can construct a graph . For the vertex set, we first define two nodes and as the start and end point. There will be layers between and . For layer , there are all the possible nodes such that . For the edge set, we first link to all nodes in the first layer, and link all the nodes in the last layer to . Then, between two consecutive layer, in the previous layer link to in the next layer if . Next, we assign all the edges with weight 0, and for the other edges, we assign edge with weight . An example graph is shown as below.



After constructing the graph, we need to get the topological order for all the nodes. Next, we can use the Dynamic Programming to solve this problem. We define as the largest weighted path from to , with base case = 0 and the recurrence is as below:

where represents the longest weighted path from to , and are all nodes that have a directed edge to and are in front of in Topological sorting order.

is the maximum sum that we are looking for. And we can iterate from to to find the path, which will be the sequence we want.

**Proof of correctness:**

To prove the correctness of the algorithm, we first prove that the graph we construct is a DAG. Since combining and , there are layers in this graph, and edges only exist from the previous layer to the next layer, there won’t be a circle in this graph.

Next, we prove that the largest weighted path in this DAG represents the sequence we want in this question. Since the value are all non-negative, the maximum weight path in the graph must be from to . And for each layer between and , only nodes such that are in layer , so they are candidates for . And because only links to in the next layer where , this path is bounded that the node index number in the next layer will always be larger than the previous one, which corresponds to the request that . For edges and , the weight of each edge is assigned with , which is the value of using that node . In addition, all the edges are weighted 0 so there won’t be additional weights put into the path.

Finally, we prove that the Dynamic Programming method can return the largest weighted path in this DAG. We first assume represents the largest weighted path from to and a base case . For the recurrence,

Since we traverse the nodes in the vertex set in Topological order, if is in front of in Topological sorting order and have a directed edge to , will have a value, and the recurrence will consider all the candidates that have a directed edge to and return the maximum. So will be the largest weighted path to , and will be the largest weighted path in the DAG.

**Time complexity:**

There are at most nodes and edges in the graph, so the construction of the graph takes . Topological sort takes . Besides, the DP traversal takes at most in each layer. Therefore, the time complexity of the algorithm is

**Problem 2: Covering points with gain**

**Algorithm description:**

We can solve this problem by applying a two-dimensional Dynamic Programming method. Firstly, sort in non-decreasing order. Then we also sort the intervals based on the following rules:

1. If , interval is placed in front of .
2. If and , interval is placed in front of.

We define as the maximum gain from the interval to , considering the point set is . The base case are for . Assuming that for , there are points in that are within the interval, the recurrence is shown as below:

We can traverse the row by row from left to right. When we start to meet the point , the rest of in this row is computed as

.

We finally output as the maximum gain of all intervals. And we can look back from to to find which intervals are used.

**Proof of correctness:**

We first define as the maximum gain until interval that cover points until . The base case are for , meaning the gain until any point when using no interval is zero.

Next, we are going to prove the recurrence.When a new interval comes, we will have two options: to use this interval or not to use. If we choose not to use this interval, the maximum gain until intervals using sorted points will be the same as . If we choose to use this interval, then this interval at most cover consecutive points. So we should extract the points that this interval covers, and use the previous intervals to cover the previous points. There will be or no points that the previous intervals need to cover if . In this case,

Because is a maximizer, we choose the larger one of the two terms to be . In addition, when we start to meet the point , then the range until interval [] cannot cover the point and those after , so we assign

.

Because we traverse row by row and from left to right, there will always be a value for every . At the end, represents the maximum gain until the last interval that tries to cover all the points, and it corresponds to the solution of the problem.

**Time complexity:**

The two sorting each takes and . We need to traverse the whole matrix once, so here it is . In each iterations, we need comparison and to compute the sum of gain of points. Hence, the time complexity of the algorithm is

**Problem 3: Round-robin tournament**

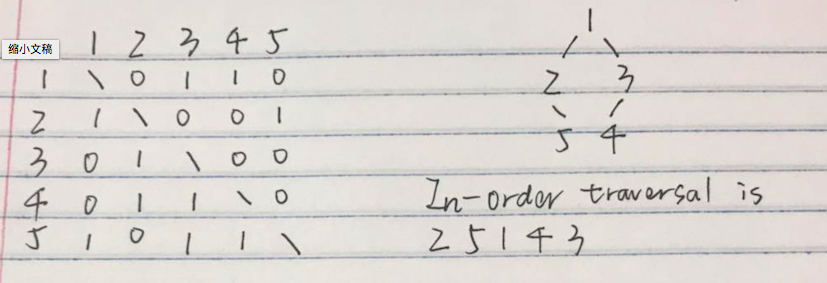
**Algorithm description:**

At first I want to try Topological sorting or divide and conquer method to solve this problem, but this problem is not necessarily a DAG and I couldn’t figure out a divide and conquer algorithm to solve this problem in . Here it’s an algorithm with best case time but worst case in .

We can use binary tree and In-order traversal to solve this problem. Assume the players are marked with . We traverse the players from to and build a graph mentioned as below:

1. We define as the root of a binary tree.
2. At each iteration we meet a new player , we begin by comparing with the root . If defeats , we assume as the left child of and we continue to compare with . Similarly, if defeats , we assume as the right child of and we continue to compare with .
3. We stop the last step until we reach the leaves of the tree. If defeats this leaf , we assign as the left child of . If is defeated by this leaf , we assign as the right child of .
4. After assigning as a leaf of the binary tree, we continue to compare the next player until we traverse all the players.

A possible binary tree is shown as below. Finally, we can return the In-order traversal of this binary tree as a solution to the problem.



**Proof of correctness:**

We can prove the correctness by looking at the rules we set to build the binary tree. According to the rules, we can ensure that the players in the left subtree of a node all defeat , and the all the players in the right subtree of a node are defeated by . We traverse all the players and we build a binary tree with nodes, so the In-order traversal can ensure there are players. Since the order of In-order traversal is left, middle, right, we can assure that the In-order traversal corresponds to a feasible solution of the problem.

We can also prove the correctness by assuming In-order traversal is not a feasible solution. In this case we will have a player in the left subtree is defeated by the root , which contradicts the fact that the player who is defeated by the root be put into the right subtree.

Hence, we prove the correctness of this algorithm.

**Time complexity:**

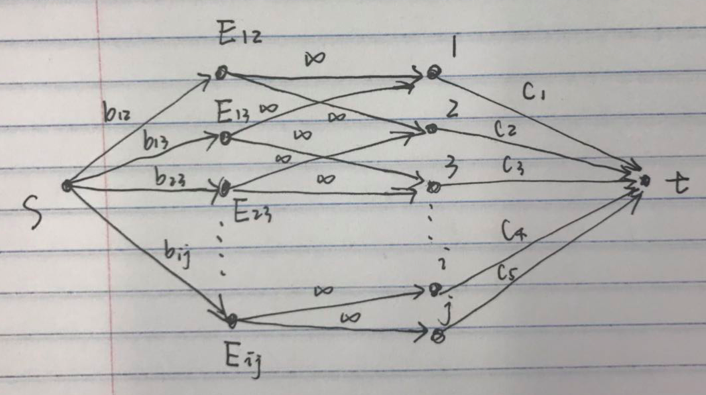
The worst case is to meet new player who always defeat or be defeated by the previous players. In this case the time complexity is

The best case of this problem is that the binary tree is a complete binary tree. In this case the maximum times of comparison is , so the time complexity is

**Problem 4: Cellular network (Revised problem)**

**Algorithm description:**

We can use min-cut to solve this problem. We first construct a network . For the vertex set , it includes a source and sink , and the vertices in the provided undirected vertex set . In addition, for every edge , a node is included in V. For the edges set , we first link to all the nodes in , then we link every to nodes and . Besides, we also link every node in V to the sink . For the capacities of the edges, we define , and . An example network is shown as below:



The capacities can be irrational, so we need to apply Preflow-push method to find the min cut. Suppose the min cut where the source *s* belongs to and the destination *t* belongs to , then the set of all the vertices in will be the subset we want.

**Proof of correctness:**

To prove the correctness of the algorithm, we need to show that the min cut solution corresponds to the solution of the original problem.

In this problem, we are looking for a set A such that is maximized. We assume , then this problem is equivalent to looking for the minimized .

There are three types of edges in , where , so the min-cut will not cross any edge, or it will lead to infinite cut value. Therefore, the min-cut only involves the other two types of edges, where we have and . Suppose the min-cut is where the source *s* belongs to and the destination *t* belongs to . Then the min-cut value is

Since the min-cut doesn’t involve any edge, must be in the same set with node and . Hence, if we assume is the set of all the vertices in , we can rewrite the above equation as

Since this is the min cut, we cannot achieve a cut that has a smaller value, is exactly the set we are looking for. Therefore, if the min cut is where the source *s* belongs to and the destination *t* belongs to , the set of all the vertices in will be the subset we want.

**Time complexity:**

We first need to construct the network, which has vertices and edges . So the construction takes . We use FIFO preflow-push algorithm, whose time complexity is , so it takes . Thus, the total complexity of the algorithm is

.

**Problem 6: Approximation algorithm**

Suppose the optimal solution is , and the sum of all the is , which can be written as . Since, we are looking for the bigger sum of the partitioned set, it is trivial to know that .

The algorithm first chooses largest ’s and find the optimal partition of these integers. We can without loss assume that are the largest ’s. Suppose the two sets are obtain using this algorithm are and , and for convenience, we write , and we can also without loss suppose .

Let be the last one that is put into that makes . By adding to both sides, we have

There are two periods that can be put into . If it happens during the period when we partition the largest ’s, then is the optimal solution. We can prove this by looking at the description of the algorithm. We first partition the largest ’s and obtain the optimal solution. In the second period when we partition the rest elements, there is no such element that can make minus its value to be smaller than . According to the algorithm, we only put the element to a set if the sum in that set is smaller. Hence, no element is put into in the second period. Since and is derived by find the optimal solution in the first period, is the optimal solution, and the approximation ratio is 1 in this case.

If is put into in the second period, then we know that

Hence, there are at least with value no smaller than in the whole set, and we can write it as

So the approximation ratio can be written as

We first need to choose the largest ’s. We can solve this by traversing the whole set once, so it takes . Next, we are going to partition the largest ’s, which will take by checking all the possibilities. Next, we simply traverse the rest set and partition them by checking which set has larger sum, so it takes . Therefore, the whole algorithm takes time complexity .

As mentioned all above, we prove that this algorithm with time complexity outputs a sum that is within fraction of the optimal output.