CS 180 Homework 6

Name: Wang, Zheng CS 180 Dis 1E

UID: 404855295 Time: F 10:00-11:50

TA: Orpaz Goldstein

**Question 1 (Exercise 16, page 327):**

**Explanation:**

The idea is to use a combination of the greedy algorithm and dynamic programming to solve this question.

Suppose a tree has root and has children . Let the subtree rooted on be . Now, assume the minimum round needed to notify everyone in each subtree is . Without loss of generality, assume .

*Claim:* if notify all its children according to the order (call this scheduling ), the total round taken will be minimized.

*Proof.* Let’s suppose there is some other scheduling for with a different ordering of ’s (not sorted). Then, there must exist some neighboring pair such that is before , but .

Let’s say that the maximum round taken for everyone except for , and their decedents is . Then, right now the total round to notify everyone in is

Now suppose we switch the order of and —notify and then . Then, the total round taken to notify everyone is

We notice that and also . Thus, we have

So

Thus, and we will not worsen the total round take to notify everyone if we switch and . If we keep doing so, we will eventually get . Thus, is optimal.

Using the idea of dynamic programming, we then need to find . But this is easy and we find (Assuming ). By the *claim*, we know that if we call subordinates of in the order , we can minimize the total round need to notify everyone in . Thus, the total round taken to notify everyone is going to be the maximum round for a particular subtree to be all notified, AKA .

The base case here will be when is a leaf, here we use .

Finally, use the idea of dynamic programming, we can start from the bottom layer, propagate above, until we get the ultimate root representing the ranking officer.

To output the sequence the officers should make the phone call, just output the sorted subordinate list of each officer when that officer is notified

**Algorithm (suppose the root of**  **is ):**

* Define function output\_calling\_sequence
  + if is a leaf
    - stop
  + output the sorted subordinate list of
  + for each subordinate in the sorted subordinate list of
    - output\_calling\_sequence
* Initialize the tree with officer along with as each node
* Run BFS on tree to get the level each officer is located on
* for each leaf of
  + set
* for each level of from bottom to top
  + for each officer at that level, if is NOT initialized
    - sort the subordinates of the officer by in decreasing order
    - for each subordinate in the sorted list ()
  + end for
* end for
* call output\_calling\_sequence()
* output

**Complexity:**

1. To construct the tree will take time to run as there are nodes and edges
2. Initialize the leaves will take time to run as there can be in maximum leaves
3. The nested for loop will take time to run. The reason is as follows:
   1. For each node in the tree, suppose it has children, then sorting and finding the maximum will take step
   2. Since , we have . Thus, the overall complexity is
4. Output the calling sequence will output all the nodes of , so that will take step to run

Thus, the overall complexity of this algorithm is .

**Question 2 (Exercise 21, page 330):**

**Explanation:**

The idea is to use two dynamic programming in this process.

We will first define being the maximum money we can earn in the interval from day to day by making a single transaction.

How to find ? We see that there are two cases:

1. If we buy the shares on day and sell it on day . Then the amount of money we can obtain is
2. Otherwise, the transaction happens either in the interval or . The maximum money we can earn is thus and resepectively.

Thus, ,

For the base cases, we know that for all . Using the idea of dynamic programming, we can easily calculate all from the base cases.

The next step is to calculate the optimal money we can earn from day to day , where , by making exactly () transactions. Let denote this value.

Then, we see . The reason to get this equation is that we can divide the interval into to two part and , we define that in we are making the last transaction. Then, we see that the best profit we can achieve for the last transaction in interval is . The best profit we can achieve for all rest transaction is After trying all possible , we will find the best profit we can achieve by making transaction in interval , i.e. .

For the base cases, we see that , since we cannot earn anything or loss anything with in a single day (from day 1 to day 1). We also see that since if we do not make transaction, we will not earn nor loss any money. Finally, when we have to much transaction (, set such .

In order to output the k-shot we are going to use, simply back track and output each transaction happened whenever a is included for .

**Algorithm (assume 1 indexing):**

* Initialize 2d array that is of size
* Initialize 2d array of size
* Initialize 2d array of size
* For
  + For
  + End for
* End for
* For
  + Set
* End for
* For ++
  + For ++, ++

,

* + - if is either or
      * let points to the one that is larger
    - else
      * let points to
  + end for
* end for
* for
  + for
    - if is 1
    - Else if
    - Else if is 1
      * Let points to
    - Else
      * Let points to and that maximize
  + End for
* End for
* Output
* If we output 0
  + output “no transaction made”
* Else
  + Initialize a stack
  + Follow the pointer of ( is the maximum of ) to an element of and trace back from that until we get an element from . Push that element in a stack
  + Follow the pointer of to the previous element of , repeat above until there is no previous element
* Output and pop out each element in to show the k-shot found

**Complexity:**

1. Initializing and will take to complete
2. To fill , each element can be filled with , and we are filling elements. Thus, filling will take to complete
3. Outputting the maximum return will take to complete
4. Backtracking to find the each element will take to complete. This is because we will in maximum follow pointer in before reaching an element. Since we are finding in maximum elements, this step will take to complete.

Thus, the overall time complexity of this algorithm will be

**Question 3 (Exercise 24, page 331):**

**Explanation:**

Let’s denote the two parties and , the two districts and , and the precincts be .

Without loss of generality, suppose there is a way partition into and such that party can win in both districts, then the vote obtained by , satisfies votes. This is because to win in both districts, we must have ( are votes gets for and respectively).

Thus, with the method mentioned above, we can determine which party can possibly take advantage of gerrymander (only the party with majority vote can take advantage of gerrymander). If none of the parties can satisfy the condition above, then of course the precincts are not susceptible to gerrymander.

Next, we will show how to check if the precincts are susceptible to gerrymander. Without loss of generality, assume that has already satisfied.

We see that the precincts are susceptible to gerrymander if and only if there is subset of size of such that satisfies:

*Proof.*

If the precincts are susceptible to gerrymander, by definition, there is a subset of size of and such that and . Thus, .

Conversely, if there exist subset of size of such that satisfies , then let , we have and . By definition, the precincts are susceptible to gerrymander.

Now, it is time to give the actual algorithm explanation. We will first see if a party satisfies . If neither of the two parties can achieve this, then the set of precincts are not susceptible to gerrymander. Otherwise, we continue with the following.  
Let be if for a set of precincts , there exist some subset of size such that party can win votes in will be otherwise.Then, we see that when we include in the set of precincts, there are two cases.

1. We include it in
2. We do not include it in

Thus, if at least one of the following is and is otherwise:

1. is (when we do not include in )
2. is ( the number of votes party gets, this correspond to the case when we include in )

Then, we see that the base cases are if and otherwise.

Also, , otherwise, if any of less or equal to 0, .

**Algorithm (assume that**  **is the number of votes party**  **get at precincts , we define also and**  **in a similar fashion):**

* Initialize a 3d array with size
* Set
* Set
* If and
  + Return
  + Exit the program
* Else set to be the PARTY with larger total vote ()
* For
  + For :
    - Set
  + End for
* End for
* Set
* For
  + For
    - For
      * If
        + Then
      * Else if and
        + Then
      * Else if and

and

* + - * + Then
      * Else
        + Then
    - End for
  + End for
* End for
* Let be the larger one of and
* For
  + If any of the element in is
    - Return and end the program
* End for
* Return

**Complexity:**

1. Initialize all the elements in will take to complete.
2. The nested for loop to fill the recursive cases will take to complete
3. To check If any of the element in is , where will take to complete

Thus, the overall complexity of this algorithm is .

**Question 4 (Exercise 6, page 417):**

**Explanation:**

Let the fixtures be , the switches be , and the wall be .

The idea to solve this problem is to create a directed weighted graph with nodes ,, where is a pseudo starting node and being a pseudo target node.

The edges are then draw according the the following rules:

1. Draw an edge from to each of ; all of these edges have capacity 1
2. Draw an edge from each of to ; all of these edges have capacity 1
3. If the line defined by end points and does not cross any wall, then draw an edge from to ; all of these edges have capacity 1

With the graph constructed above, we then run Ford-Fulkerson algorithm to find the maximum flow from to .

*Claim:*

If and only if the maximum flow from to is equal to , there is a perfect matching of all fixtures and switches.

*Proof:*

We see that the maximum flow out of is , so that we cannot get a maximum flow larger than .

Then if maximum flow is smaller than , then that means at least one has no incoming flow (a match means that there is a flow of 1 from to ), then that means not all has been matched with a .

Similarly, if there is no perfect match, the maximum flow must be less than .

Thus, by argument above, the claim is true.

Then, if there is a perfect match of each lamp and with a switch, there is a way to match each switch with a lamp with an edge we draw. By the rules of how we draw edges, no wiring of lamp and switch cross the walls, so it is ergonomic.

Also, we see that if the room is indeed ergonomic, then there will be perfect match and our algorithm is guaranteed to find it. So, this algorithm will work.

**Algorithm:**

* Initialize with vertex ,
* put an edge from to each of ; all of these edges have capacity 1
* put an edge from each of to ; all of these edges have capacity 1
* For each of in
  + For each of in
    - For each of in .
      * If line intersect with
        + Continue
    - End for
    - Add edge from to with capacity 1
  + End for
* End for
* Run Ford-Fulkerson Algorithm on , from to , to get maximum flow,
* If is equal to
  + Return ergonomic
* Else
  + Return not ergonomic

**Complexity:**

1. To construct the graph , we have to add all vertices, which take to complete. We will add all edges and edges, which take to complete. Finally, we for each of the pair we check it against all of the walls to determine if we put an edge or not. There are pairs and there are walls. Thus, this step will take to complete.
2. Then, we simply run Ford-Fulkerson Algorithm on Since there are edges in maximum in ; the maximum flow out of is . Thus, Ford-Fulkerson Algorithm will take to run.

Thus, the overall complexity will be

**Question 5 (Exercise 8, page 418):**

**(a)**

**Explanation:**

To solve this problem, we will construct a weighted directed graph with nodes .

We first create edges from to each of , the capacity of each edge is the supply of that type of blood. For example, edge has capacity and edge has capacity .

Then, we create edges from each of to . The capacity of each edge is the demand of that type of blood. For example, edge has capacity .

Finally, we add edges from to . If a blood type in can donate blood to a blood type in , then draw and edge from to . Each of the edges in this step has capacity .

With this graph constructed, we run Ford-Fulkerson Algorithm on to find the maximum flow from to .

*Claim*:

If and only if the maximum flow is the sum of the demand, (), the supplies will fulfill the demands.

*Proof.*

If there is enough supply, then each of the node in can receive at least the demand they need. So, for all . Thus, the flow out of will saturate the edge for all . In another word, is equal to maximum flow.

Conversely, if is equal to maximum flow, then it is necessary that the edge for all is saturated. Thus, for all . Thus, all of the blood type in (which represent receiver of the blood) will get amount that satisfy the demand.

**Algorithm:**

* Initialize graph with nodes .
* For in
  + Add edge from to in
* For in
  + Add edge from to in
* For in
  + For in
    - If blood type can receive blood type
      * Add an edge from to in
  + End for
* End for
* Run Ford-Fulkerson Algorithm on , from to , to get maximum flow,
* If is equal to
  + Return satisfied
* Else
  + Return not satisfied

**Complexity:**

1. We see that the graph is constant. Thus, creating the graph take time to compete.
2. We then run the Ford-Fulkerson Algorithm, this will take , where is the number of edges in , and is the maximum flow out of , which is the sum of supplies of all types of blood (i.e. ). Thus, as the graph is a constant, the Ford-Fulkerson Algorithm will run with time complexity .

Thus, the overall time complexity is .

**(b)**

The cut can be and . This cut has capacity , and thus cannot fulfill the maximum flow of required.

The explanation: since the demand of and can only be fulfilled by type and . As there are in total 86 unit of supply of blood and . But there are unit of demand of blood and . Thus, we cannot full fill all the demands.

**Question 6 (alternating sequence):**

**Explanation:**

The idea to solve this question is to use dynamic programming. Let the original sequence be and let denote the th element of .

We first notice that if we know the maximum length of the alternating sequence that end up all , then we can find the global maximum length of the alternating sequence.

We also notice that if we force the sequence to end with an element , then there are only two cases:

Thus, we can come up with a recursive call to solve the maximum length of the alternating sequence that end up all We will first define some terms.

1. Let denote the maximum length of the alternating sequence that end up with the th element of , given that the alternating subsequence satisfies ( denote the th element of )
2. Let denote the maximum length of the alternating sequence that end up with the th element of , given that the alternating subsequence satisfies ( denote the th element of )

Then, we see that . This is because for alternating subsequence to end up with , the subsequence before it must be . Thus, we search elements in .

Similarly, we see that .

Then, we see that the base cases here is and .

**Algorithm**

* Initialize a 2d array with size
* Set
* Set
* For
  + Set
  + Set
* For
  + For
    - If
      * If has been updated
        + Let points to
    - If
      * If has been updated
        + Let points to
  + End for
* End for
* Output the maximum element in table
* Trace back all of the pointers of the maximum element. The result should be a sequence of elements. Reverse this sequence of (call this ). Then get the second index of all elements in (i.e. the index number in the second of ). The result is an index sequence , where .
* Output the subsequence by output .

**Complexity:**

1. Initialization will take to run
2. The nested for loop will take time to complete since for each we have in maximum ’s to check, and there are ’s.
3. To find the maximum element will take
4. To output the subsequence will take

Thus, the overall time complexity of this algorithm will be .