Sum Yi Li
(UID: 505146702)

① Exercise 19 on page 329

- Let assume that s has in total n number of character
- Let consider there are a total n characters for the repetition of x' of x. There are also a total of n characters for the repetition of y' of y
- The problem we want to investigate is whether s is considered to be interleaving between x' and y'.
- When we are considering the string x' & y', we do not need to worry about wrapping around and several period of x' and y'
- Let assume that w[j] refer to the jth character of the string s
- Let assume w[1 : j] refer to the first j characters of s.
- If s is an interleaving of x' & y', the last character comes from either x' or y'.
- We will have a smaller recursive call on w[1 : n-1] with prefixes of x' & y'. if we remove the last character with the corresponding index.
- We will focus on the sub-problems defined by the prefixes x' and y'.
- Let assume the slot of our table denote as M[i, j] to be yes if w[1 : i+j] is considered as interleaving of prefixes x'[1 : i] & y'[1 : j]

- If there is an interleaving, it means that the final character is either $x'[i]$ or $y'[j]$
- The recurrence statement will be:

  $M[i,j]$ equal to yes iff $M[i-1,j]$ equal to yes and $w[i+j] = x'[i]$ or $M[i,j-1]$ equal to yes and $w[i+j] = y'[j]$ (the last character comes from either x' or y')
- The algorithm will be conducted as follow:
- First, we will set $M[0,0]$ to equal to yes.
- Then, we are looping from $k=1$ to $k=n$.
  - We are looping for all the pairs so $i+j$ equal to k.
  - If $M[i-1,j]$ equal to yes and $w[i+j] = x'[i]$
    - Then $M[i,j]$ equal to yes
  - Else if $M[i,j-1]$ equal to yes and $w[i+j] = y'[i]$
    - Then $M[i,j]$ equal to yes.
  - Else
    - Then $M[i,j]$ equal to no.
  - End of inner for loop
- End of outer for loop
- Function return yes if there is some pair (i,j) with $i+j-n$, therefore $M[i,j]$ equal to yes.

① Prove of Correctness by induction

- The base case is $M[0,0] = yes$.
- We have defined a recurrence statement :

$M[i,j] = yes$ iff

$M[i-1,j] = yes$ and $w[i+j] = x'[i]$ or

$M[i,j-1] = yes$ and $w[i+j] = y'[i]$

- For the inductive steps, we keep on removing the last character of our string to show that the last character is either from x' or y' if s is considered as interleaving of x' & y'.
- We will look at the smaller recursive calls of the sub-problems
- Therefore, $M[i,j] = yes$ if $w[1:i+j]$ is interleaving of prefixes $x'[1:i]$ & $y[1:j]$
- However, the inductive set will output whether the string is an interleaving of x' & y' for the first $i+j$ characters.
- As a result, $M[i,j]$ plus the last character whether it is from x' or y' will give us whether s is an interleaving of x & y.

① Time Analysis

- For our table M[i, j], it will takes $O(n^2)$ to build it.
- Each slot will take constant time to fill in the result on the separate subsets
- Therefore, the total running time is $O(n^2)$

② Exercise 22, page 330

- Let C_a be the cost of a edge
- Let C_{gh} to be the cost of the edge between the node g & node h
- Let $opt(i, g)$ be the optimal cost of the shortest path to g using exactly i edges
- Let $N(i, g)$ be the number of shortest paths to g.
- We set $opt(i, v) = 0$ because the shortest path from v to itself is zero.
- We set $opt(i, v') = \infty$ for all $v' \neq v$ because we still don't know the shortest path to node v' yet.
- Set $N(i, v) = 1$ since the shortest path is to itself.
- Set $N(i, v') = 0$ for all $v' \neq v$ since we haven't find any shortest path yet.
- Therefore, we currently know source v is reachable, but there is only one path to get this.
- The recurrence of the algorithm is:

$$opt(i, g) = \min_{h, (h,g) \in E} \{opt(i-1, h) + C_{hg}\}$$

- For traveling to node g with i edges, we need to go through a node h that is before node g with $i-1$ edges. & account for the edge from node h to node g.
- After we compute the compute the shortest path to node g. We can repeat the same thing for finding out the total number of shortest paths to node g.

- The recurrence for this step :

$$N(i,g) = \sum_{h,\ (h,g)\in E \text{ and } opt(i,g)=opt(i-1,h)+C_{hg}} N(i-1,h) .$$

- Therefore, we examine all the predecessor nodes which are able to fulfill the shortest path cost & keep a count & add them up.
- The algorithm could be formed by a double loop
  The outer loop is looping through i which is different number of possible edges to get to the corresponding destination nodes
- The inner loop is looping through all the possible destination node g.
- After we compute the paths with a specific length to a node c, then we find the optimal path to node c by taking the minimum among all the possible paths to node c with different lengths.
- The recurrence for the step :

$$opt(c) = \min_i \{opt(i,c)\}$$

- The total number of the shortest path could be computed by summing up all the paths that has this specific amount of minimum cost.
- The recurrence for the step :

$$N(c) = \sum_{i,\ opt(i,c)=opt(c)} N(i,c)$$

② Prove of Correctness by induction.

- The base case is $opt(i, v) = 0$, the shortest path from v to itself & $N(i, v)$, there are only one path to there.
- We have define a function to find the optimal path to a node g with i edges by considering all the possible length & take the minimum of it.
- For the inductive steps, assume we have the optimal path up to the node h, the predecessor node of node g.
- We are picking the minimum edge from node h to node g to finish computing the shortest path to node g.
- Based from the assumption & inductive steps, we are able to compute the shortest path to any nodes g with i number of edges.
- Since we keep track of of all possible paths to a specific node, we just need to take out the ones that fulfill minimum cost among different lengths to the node & make a counter to sum up all those paths.
- The counter would be the total number of the shortest paths from a source node to a specific node.

② Time Analysis

- The outer loop is looping through different number of possible edges to get to the destination node. This takes $O(n)$
- The inner loop is looping through all the possible destination nodes. This takes $O(n)$.
- Therefore, it takes $O(n^2)$ to compute the all the possible paths to node c with different lengths.
- It takes $O(1)$ to find the minimum path to node c.
- It takes $O(n)$ to sum up all the shortest paths to node c
- Therefore, the time complexity:

$$O(n^2) + O(1) + O(n) = \boxed{O(n^2)}$$

(2) Exercise 24, page 331

- There are n precincts & each has m voters
- The total number of voters are nm.
- Each district will have $\frac{nm}{2}$ voters
- For either part A or party B to win a district, it need to have at least $\frac{nm}{4}+1$ voters in the district.
- Let a = total number of party A voters
- In order for party A to have the possibility of gerrymandering, $a \geq \frac{nm}{2}+2$
- In other words, party A will win both party if there is a subset S of $n/2$ precincts st. the total number of party A voters in subset S is refer to $s \geq \frac{nm}{4}+1$
- The total number of part A voters in the remaining precincts is $a-s \geq \frac{nm}{4}+1$
- If there is a subset S of s as total number of part A voters that fulfill the following property:

$$\frac{nm}{4}+1 \leq s \leq a-\frac{nm}{4}-1$$

- The goal is to create a district that contain $n/2$ precincts & s total party A voters.
- Let consider precinct n, and suppose precinct n has a_n party A voters. We either include precinct n in the district or not.
- If we include precinct n, the system will be suspect to bias if we have a set of $n/2-1$ precincts from the the remaining precincts $1, \ldots, n-1$ which has $s-a_k$ total part A voters.

- If we do not include the precinct n, the system will be suspect to bias if we have a set of $n/2$ precincts from the remaining precincts $1, \dots, n-1$ that has s total part A voters.
- Let $G(x, y, z) \equiv$ possible to form a set of y precincts from among the first x precincts that has exactly z total part A voters.

$$G(x,y,z) = \begin{cases} \text{true} & \text{if } G(x-1, y-1, z-a_x) = \text{true} \\ \text{true} & \text{if } G(x-1, y, z) = \text{true} \\ \text{true} & \text{if } y = 0,\ z = 0 \\ \text{true} & \text{if } x = 1,\ y = 1,\ z = a_1 \\ \text{false} & \text{otherwise} \end{cases}$$

- The algorithm follows:
- Let's create a new array: $G[n][n/2][a - \frac{nm}{4} - 1]$
- Loop from $x = 1$ to $x = n$
    - Put $G[x][0][0]$ equal to true.
- Loop from $x = 1$ to $x = n$
    - Loop from $y = 1$ to $y = n/2$
        - Loop from $z = 1$ to $z = a - \frac{nm}{4} - 1$
            - if $x = 1$, $y = 1$, $z = a_1$
                put $G[x][y][z]$ to be true
            - else if $G[x-1][y-1][z-a_k]$
                put $G[x][y][z]$ to be true
            - else if $G[x-1][y][z]$
                put $G[x][y][z]$ to be true
            - else
                put $G[x][y][z]$ to be false
        - end loop
    - end loop
- end loop.

- Then loop $s = \frac{nm}{4} + 1$ to $s = a - \frac{nm}{4} - 1$
  - if $G[n][n/2][s]$
    return true
- end for
- return false.

Prove of Correctness

- According to the recurrence, the first case refers to the case in which we choose to include precinct x in the district.
- The second case refers to the case that we did not include precinct x in the district
- The third case is a base case that we are able to form a subset of $y = 0$ precincts that contain $z = 0$ part A voters, no matter the value of x.
- The fourth case is the second base case that we can form a set of 1 precinct with w party A voters from the first 1 precinct if precinct 1 has w party A - voters.
- The last case is if none of the above condition is satisfied, then there is no bias in the partition
- For the inductive cases, the algorithm will fill up the $n \times n/2 \times (a - \frac{nm}{4} - 1)$ table
- Each of the slot in the table store the answer correspond to the answer of the recurrence mention above.
- After the table is filled, to find out whether the system is biased, find out all the s from the rows $G(n, \frac{n}{2}, s)$ that satisfy $\frac{nm}{4} + 1 \leq s \leq a - \frac{nm}{4} - 1$
- If any slot from all the inspected rows is true, then the algorithm find the system is biased.

Time Analysis

- The table is a 3D table, it takes $O(n^3 m)$ to fill in all the entries.
- The lost for loop that check the rows $G(n, \frac{n}{2}, s)$ takes $O(nm)$
- So, total takes $O(n^3 m) + O(nm) = \boxed{O(n^3 m)}$

④ Exercise 7, page 417

- We are applying the idea of network flow here.
- There is a node a_i for every single client i and a node b_j for every base station j
- The edge that connect between the two node a & b are denoted as (a_i, b_j) & the capacity of the edge is 1 if client i locates within the range of base station j.
- Then, we have a source node s & it is connected to every client node with an edge of capacity 1.
- We repeat the same thing for the base station nodes, we will have a sink node t & it is connected to every base station node with an edge of capacity L.
- We set up a claim that there is a convenient way to connect every client to the base stations if there is a $c-d$ flow of value e
- If there is a workable connection between client i & base station j, then it will send one unit of workflow to each of the path c, a_i, b_j, d
- Due to the specify load constraint, the edge (b_j, d) does not violate the capacity condition.
- Reversely, if we have a flow value of n, then it means that one of the flow value has to be integer.
- We will look at the capacity condition to ensure no overloaded base station whenever we connect client i to base station j if the edge (a_i, b_j) contains one unit of flow.

Time Analysis

- For this graph, we have n clients & k base stations, so we have n+k nodes, so it takes O(n+k)
- There are nk edges, because each client need to connect to one of several possible base stations. There are a total of nk possible combinations between n clients & k base stations.
- Therefore, the running time is the time it takes to find a max flow on a graph of O(n+k) nodes with O(nk) edges.

⑤ Exercise 9, page 419

- We will apply the concept of flow network.
- There is a node a_i for every single patient i and there is a node b_j for every single hospital j
- There will be an edge between node a_i & b_j with capacity value of 1 if patient i is within the half hour drive of hospital j.
- For each of the patient node, we will connect it to the same & only one source node with an edge of capacity of 1
- For each of the hospital node, we will connect it to the same & only one sink node with an edge of capacity of $\lceil n/k \rceil$
- We claim that there is workable way to send all patients to hospitals if there is an x-y flow with value n.
- If we find a workable to send patients, then we send one unit of flow from x to y for each the path x, a_i, b_j, y which refers to patient i is sent to hospital j.
- Because of specify load constraint, the edge (b_j, y) does not violate the capacity condition.
- Reversely, if we have a flow value of n, then it means that one of the flow value has to be integer.
- We will look at the capacity condition to make sure no overloaded hospital when we send patient i to hospital j if the edge between (a_i, b_j) contains one unit of flow.

Time Analysis

- For this graph, we have n injured people & k hospital, so we have n+k nodes, so it takes $O(n+k)$
- There are nk edges, because each patient need to be brought to one of several possible hospitals. Therefore, there are a total of nk possible combinations between n injured people & k hospitals
- Therefore, the running time is the time it takes to find a max flow on a graph of $O(n+k)$ nodes with $O(nk)$ edges.

(6) - We will use Dynamic Programming approach to solve this problem

- Let M to be an array of length n of integers.
- Then, we define a 2D array to be sub[n][2]
- For sub[n][0], it contains the length of the longest alternating subsequence ending at index i and the last element is greater than its own previous element.
- For sub[n][1], it contains the length of the longest alternating subsequence ending at index i and the last element is smaller than its own previous element.
- Based on the above definition, we can set up the two recursive formulation.
- The first recurrence relation is:

$$sub[i][0] = \max(sub[i][0], sub[j][1]+1)$$
$$\text{for all } j<i \text{ and } M[j] < M[i]$$

- The second recurrence relation is:

$$sub[i][1] = \max(sub[i][1], sub[j][0]+1)$$
$$\text{for all } j<1 \text{ and } M[j] > M[i]$$

- For the first recurrence relation, if we are currently at position i and the element need to be bigger than the previous element. Then, for the sequence, we need to choose an element j such that M[j] < M[i]
- If M[j] is the previous element and sub[j][1]+1 is larger than sub[i][0], then update the value of sub[i][0]
- The sub[j][1] + 1 fulfill the alternating property.

- For the second recurrence relation, if we are currently at position i and the element need to be smaller than the previous element. Then for the sequence, we need to choose an element j such that M[j] > M[i]
- If M[j] is the previous element and sub[j][0]+1 is smaller than sub[i][1], then update the value of sub[i][1]
- The algorithm :
  - Given array A & n length of integers as input (A: store all the elements.)
  - int sub[n][2]; int result = 1;
  - Then initialize all the slots to be 1

```
for (int i=0; i<n; ++i)
    for (int j=0; j<i; j++)
        if (A[j] < A[i] and sub[i][0] < sub[j][1]+1)
            sub[i][0] = sub[j][1]+1;
          ↳ If A[i] is greater, then we need to check sub[j][1]
        if (A[j] > A[i] and sub[i][1] < sub[j][0]+1)
            sub[i][1] = sub[j][0]+1
          ↳ If A[i] is smaller, then we need to check sub[j][0]
    end of inner for loop

    if (result < maximum (sub[i][0], sub[i][1]))
        result = max (sub[i][0], sub[i][1])
end of outer for loop

return result.
```

(6) cont.

Prove of Correctness

- The base cases are: initialize to one

$sub[i][0] = 1$ for all i; it is the length of the longest alternating subsequence ending at index i and last element is greater than its previous element.

$sub[i][1] = 1$ for all i, it is the length of the longest alternating subsequence ending at index i and last element is smaller than its previous element.

- We have define 2 recurrence relation previously.
- For the inductive steps, we want to show that $\max(sub[i][0], sub[i][1])$ will output the longest alternating subsequence.
- For each recurrence relation during the inductive steps, we are updating $sub[i][0]$ & $sub[i][1]$ with a longer alternating subsequence if the current stored subsequence is less than the compared one.
  - $sub[i][0] = \max(sub[i][0], sub[j][1] + 1)$
  - $sub[i][1] = \max(sub[i][1], sub[j][0] + 1)$
- Therefore, the inductive steps will store the most updated longest alternating subsequence in both $sub[i][0]$ & $sub[i][1]$ correspondingly.
- As a result, the inductive steps consider every possible alternating subsequence whether the last element is smaller or greater than the previous element. So picking from $\max(sub[i][0], sub[i][1])$ will give us the longest alternating subsequence.

(b) Time Analysis

- Looping from 1 to n number of integers : $O(n)$
- Looping through all the elements that are previous of $A[i]$: $O(n)$
- If $A[i]$ is greater, check with sub[j][1] : $O(1)$
- If $A[i]$ is smaller, check with sub[j][0] : $O(1)$
- Pick the maximum of both values at index i : $O(1)$
- Therefore $n(n+1+1+1) = \boxed{O(n^2)}$