Fulati Aizihaer

Discussion: 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

1. Exercise 19 on page 329

Algorithm:

- Set **n** as the length of string **s**
- Set string **x’** as a repetition of string **x**
- Set string **y’** as a repetition of string **y**
- Set an array **opt[i,j]**, where **i** is each index of **x’** and j is each index of **y’**
- Set **opt[0,0]** = **true**
- For **i** from **1** to **n**:
    - For **j** from **1** to **n**:
        - If **opt[i - 1, j]** == **true** AND **s[i+j]** == **x’[i]**:
            - **opt[i, j] = true**
            - If **(i + j) = n**:
                - **Return true**
        - Else If **opt[i, j - 1]** == **true** AND **s[i+j]** == **y’[j]**:
            - **opt[i, j] = true**
            - If **(i + j) = n**:
                - **Return true**
        - Else:
            - **opt[i, j] = false**
- **Return false**

Proof:

- We have a base case of **opt[0,0] = true**
- Inductive step is that at each **i**, we go through the **j** and check if our conditions are met.
- If the condition **opt[i - 1, j]** or **opt[i, j - 1]** is **true**, it means that our previous **s** character was equal to previous **x’** or **y’** value at previous index **i** or **j**
    - While inside the condition, if the sum of **i** and **j** is equal to the length of the string **s**, it means that we have concluded that **s** is an interleaving of **x** and **y**.
        - we return true

- ○ In case we reach the end of the loop and return false, it means that **s** is not an interleaving of **x** and **y.**

Time Complexity:

- The nested loop takes O(n^2) time
- Array lookups and assigning value of true and false takes O(1) time
- Overall we get O(n^2) time

Fulati Aizihaer

Discussion: 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

2. Exercise 22 on page 330

Algorithm:

- Given **G = (V,E)**, where **V** are the vertices and **E** are the edges
- **n** = number of nodes in G
- Set C_{st} as the cost of the edge between node **s** and **t**
- Set array **opt[0…n-1, V],** first value represents # of edges and second value represents the nodes
- Set **opt[0,v] = 0** and **opt[0, s] = infinity** for all other vertices **s** in **V**
- Set **minDist** to hold the minimum distance value
- Set **count** = 0
- For **i = 1** to **n - 1**:
    - For **s** in **V** in any order:
        - **opt(i,s)** = min(**opt(i-1, s)**, min(**opt(i-1,t)** + C_{st})),   node **t** is in **V**
        - If node **s** == node **w:**
            - If **minDist** == **opt(i,s):**
                - **count++**
            - Else if **minDist** > **opt(i,s):**
                - **minDist** = **opt(i,s)**
                - **count** = 1
- Return **count**

Proof:

- We have a base case of **opt[0,v] = 0,** since no edge means no distance
- Inductive step is that at each **i**, we go through all the vertices **s** in **V.**
- We then set our **opt(i,s)** value as the **opt(i-1,s)** value or minimum of opt(i-1, t) + C_{st} value, where C_{st} represents the weight of the path from **s** to **t**.
- After selecting the minimum distance value for the opt(i,s), we then begin to check whether node **s** is our final destination node **w**.
- If **s** is **w**, we then go through the condition to check if opt(i,s) is the shortest path,

  - If it is the shortest path, then we increment the counter
  - If it is smaller than the shortest path, meaning that it takes less distance, then we assign that opt(i,s) as our shortest path, and restart the counter from 1.
- In the end, after we finish the loop, we return the counter to display the number of shortest paths from v to w.

Time Complexity:

- We use Bellman Ford algorithm which takes O(V*E) where **V** is the number of vertices and **E** is the number of edges.
- Comparing the distance values and incrementing the count values takes O(1) time
- Overall, time complexity is O(V*E)

Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

3. Exercise 24 on page 331

Algorithm:

- Set **n** as number of precincts
- Set **m** as number of registered voters in each precinct
- Set **p** as each precinct
- Set **d** as each precinct in a district
- Set **v_1** as votes in District 1
- Set **v_2** as votes in District 2
- Set **v_a** as votes for party A
- Set array **opt[p, d, v_1, v_2]**
- Initialize **opt[0, 0, 0, 0] = true** and everything else as false.
- For **p** = 1 to n:
    - For **d** = 1 to n:
        - For **v_1** = 0 to m*n:
            - For **v_2** = 0 to m*n:
                - If (**opt[p-1, d-1, v_1 - v_a, v_1]** == true):
                    - **opt[p,d,v_1**,**v_2**] = true
                - Else if (**opt[p-1, d, v_1, v_1 - v_a]** == true):
                    - **opt[p,d,v_1**,**v_2**] = true
- If there is a **opt[n, n/2, v_1, v_2] = true**:
    - If **v_1 > mn/4** and **v_2 > mn/4**:
        - Return **true**

Proof:

- We have a base case of **opt[0,0,0,0] = true**
- Inductive step is that:
    - We go through the nested loops **p** → **d** → **v_1** → **v_2**
    - We then loop at the condition where if
    - At v_2, we go through the conditions to check conditions
        - If **opt[p-1, d-1, v_1 - v_a, v_1]** == true
            - If yes, then we set **opt[p,d,v_1**,**v_2**] = true

        - Else if **opt[p-1, d, v_1, v_1 - v_a]** == true
            - If yes, then we set **opt[p,d,v_1**,**v_2**] = true
- After we finish the loop, we check if **opt[n, n/2, v_1, v_2] = true**
    - If true, we then check **v_1 > mn/4** and **v_2 > mn/4**
        - If true, then return **TRUE**

Time Complexity:

- Array lookups and assigning value of true and false takes O(1) time
- We have a nested for loop which will take n * n * mn * mn time which is O(n^2 * mn^2)
- Overall, our run time is O(n^4 * m^2)

Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

4. Exercise 7 on page 417

Algorithm:

- Set **S** as the **source node**
- Set **T** as the **sink node**
- Set C_i to represent the **client i**
- Set B_j to represent the **base station j**
- Connect all the C_i with **S** with edge **capacity** of **1**
- Connect all the B_j with **T** with edge **capacity** of **L**
- if C_i is within range **r** of the base station B_j
    - Add edge (C_i , B_j) with capacity 1
- Run the maxflow(Ford Fulkerson) algorithm on the graph, return the max flow of that graph
- While there is a path from nodes **s** to **t:**
    - Find a path from s → t:
    - Create the augmented graph
    - Find another path from s → t on the augmented graph we just created
    - Create the next augmented graph
    - Repeat bullet points 3 and 4 until no more flow can be added from s → t
    - Return the max flow
- If the max flow is equal to **n**, then every **client** can be connected simultaneously to a **base station**.

Proof:

- Like we did on the cell phone tower problem, we first prepare our problem to be able to run max flow algorithm
- We connect all the clients with the source node with capacity of 1, and all the base station with the sink node with capacity of L
- We then check if we can connect the all the clients with the base stations with an edge that has capacity of 1 (making sure that we only connect each client to only one base station

- We then use the Ford Fulkerson algorithm to find the max flow, which we proved in class that it works
- After we run the Ford Fulkerson algorithm and return the max flow, we check if max flow is equal to the number of clients.
- If it is, it means that all the clients can be connected simultaneously.
- Therefore, the algorithm above works

Time Complexity:

- Running the Ford Folkerson algorithm takes O(|f| * E), where f is the max flow and E is the number of edges
- |f| is equal to **n** because that is the number of clients, and E is equal to O(**n + nk + k**) = O(**nk**) edges
- Overall, the runtime is O(n^2 * k)

Fulati Aizihaer

Discussion: 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

5. Exercise 9 on page 419

Algorithm:

- Set **S** as the **source node**
- Set **T** as the **sink node**
- Set P_i to represent the **patient i**
- Set H_j to represent the **hospital j**
- Connect all the P_i with **S** with edge **capacity** of **1**
- Connect all the H_j with **T** with edge **capacity** of **n/k**
- if P_i is within half-hour's driving time of the H_j:
    - Add edge (P_i , H_j) with capacity 1
- Run the maxflow(Ford Fulkerson) algorithm on the graph, return the max flow of that graph
- While there is a path from nodes **s** to **t:**
    - Find a path from s → t:
    - Create the augmented graph
    - Find another path from s → t on the augmented graph we just created
    - Create the next augmented graph
    - Repeat bullet points 3 and 4 until no more flow can be added from s → t
    - Return the max flow
- If the max flow is equal to **n**, then every **patient** can be sent to a **hospital**

Proof:

- Like we did on the cell phone tower problem, we first prepare our problem to be able to run max flow algorithm
- We connect all the patients with the source node with capacity of 1, and all the hospitals with the sink node with capacity of L
- We then check if we can connect the all the patients with the hospitals with an edge that has capacity of 1 (making sure that we only connect each patient to only one hospital
- We then use the Ford Fulkerson algorithm to find the max flow, which we proved in class that it works

Fulati Aizihaer

- After we run the Ford Fulkerson algorithm and return the max flow, we check if max flow is equal to the number of patients
- If it is, it means that all the patients can be sent to a hospital
- Therefore, the algorithm above works

Time Complexity:

- Running the Ford Folkerson algorithm takes O(|f| * E), where f is the max flow and E is the number of edges
- |f| is equal to **n** because that is the number of patients, and E is equal to O(**n + nk + k**) = O(**nk**) edges
- Overall, the runtime is O(n^2 * k)

Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

6. Given a sequence of numbers, find a subsequence of alternating order, find the length where the subsequence is as long as possible. (That is, find a longest subsequence with alternate low and high elements).

Example
Input: 8, 9, 6, 4, 5, 7, 3, 2, 4
Output: 8, 9, 6, 7, 3, 4 (of length 6)
Explanation: 8 < 9 > 6 < 7 > 3 < 4 (alternating < and >)

Algorithm:

- Given input array **arr**
- Set **n** as the length of the input array
- Set two arrays **opt_low[1…n]** and **opt_high[1…n]**
  - **opt_low** contains the length value of when the last element is smaller than the previous element
  - **opt_high** contains the length value of when the last element is larger than the previous element.
- Set two arrays **sub_low[1…n]** and **sub_high[1…n]**
  - **sub_low** contains the subsequence of when the last element is smaller than the previous element
  - **sub_high** contains the subsequence of when the last element is larger than the previous element.
- Set **opt_low[i]** = 1 and **opt_high[i]** = 1, for all **i**
- Set **sub_low[i]** = a[i] and **sub_high[i]** = a[i], for all **i**
- For **i** from **1** to **n**:
  - For **j** from **1** to **i - 1**:
    - If **arr[i] > arr[j]**:
      - **opt_high[i]** = max(**opt_high[i]**, **opt_low[j] + 1**)
      - **sub_high[i]** = **sub_low[j] + a[i]**
    - Else if **arr[i] < arr[j]**:
      - **opt_low[i]** = max(**opt_low[i]**, **opt_high[j] + 1**)
      - **sub_low[i]** = **sub_high[j] + a[i]**
- Print max**(**max length**(sub_low),** max length**(sub_high))**
- Print max**(**max(**opt_low),** max**(opt_high))**

Fulati Aizihaer **Discussion:** 1F

Time: 2:00pm - 3:50pm, Friday

TA name: Parshan

Proof:

- We have a base case of:
    - **opt_low[i]** = 1 and **opt_high[i]** = 1, for all **i**
    - **sub_low[i]** = a[i] and **sub_high[i]** = a[i], for all **i**
- Inductive step is that at each **i**, we go through the **j** and check if our **arr[i] > arr[j]** or **arr[i] < arr[j]** conditions are met.
- If the condition **arr[i] > arr[j]** is **true**:
    - we then assign **opt_high[i]** as the maximum value between what we already have for **opt_high[i]** or **opt_low[j] + 1**
    - We assign **sub_high[i]** = **sub_low[j] + a[i]**
- Else if the condition **arr[i] < arr[j]** is true:
    - we assign **opt_low[i]** as the maximum value between what we already have for **opt_low[i]** or **opt_high[j] + 1**
    - We assign **sub_low[i]** = **sub_high[j] + a[i]**
- After we finish the loop:
    - we then print the maximum value between maximum length value in **sub_low** and maximum length value in **sub_high**
    - we then print the maximum value between maximum value in **opt_low** and maximum value in **opt_high**

Time Complexity:

- The nested loop takes O(n^2) time
- Array lookups and assigning values takes O(1) time
- Overall we get O(n^2) time