HW6 Solutions

1. Exercise 19 on page 329 (15 points)

Algorithm (10 points)

n: length of s
x_{cat}: concatenation of x (up to n characters suffice)
y_{cat}: same as x_{cat}
As per usual, for Dynamic Programming questions we try to find a recursion resulting in smaller subproblems.

Define opt(x_index=i, y_index=j):

- It is true if the first (i+j) characters of s are an interleaving of
  - first i characters of xcat and
  - firt j characters of ycat
- else it is False

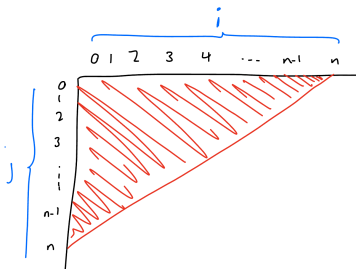
Base case: opt(0,0) = True

For index i+j we have:

- Two cases: the last character s[i+j] can be from x_{cat} or y_{cat}
  - if it is from x_{cat}
    - s[i-1, j] == True and s[i+j] == x_{cat}[i]
  - if from y_{cat}
    - x[i, j-1] == True and s[i+j] == y_{cat}[j]
- if none of these conditions are met, it means that the first i+j characters cannot be built with i characters of x_{cat} and j characters of y_{cat}, hence opt[i,j] must be False.

Time Complexity (3 points)

- we need the value of opt[i,j] for all i+j<n.
  - $O(n^2)$



Proof (2 points)

Induction. The logic is the same as the algorithm section. We just write down the base case, and prove that the recursion step (combining subproblems) covers all the possible ways for the main problem. Here, for s of size n, the only possible ways of building s is by opt[0, n], opt[1, n-1], … , opt[n, 0]: all cases will be checked.

2. Exercise 22 on page 330 (20 points)

Algorithm (14 points)
Edges might have negative weights, so we cannot use Dijkstra's algorithm.

Notation: We want to find the shortest path from node v to w

- Create an array dist of size |V|
  - Initialize it with dist[v] = 0 and dist[u] = ∞ for all other nodes u
- Create another list of size |V| to store all the shortest paths to nodes: sp
  - Let it be empty for all nodes
- Do this process |V|-1 times:
  - Traverse each edge e=[s,t] once and update dist[t] and sp[t] if a new shortest path is found
    - If dist[t] > dist[s] + $cost_e$:
      dist[t] = dist[s] + $cost_e$
      sp[t] = sp[s] + 'e'
      - Explanation: we append 'e' to the end of **all the** shortest paths to s that existed
    - If dist[t] = dist[s] + $cost_e$:
      Sp[t].append(sp[s] + 'e')
      - Explanation: another shortest path is found. Same as the previous case, but keep the previous paths as well.

Time Complexity (2 points)
The loop runs for |V|-1 times and checks |E| edges every time: O(|V||E|)

Proof (4 points)
The shortest path between any two nodes can have at most |V|-1 edges, or there would be a cycle.

- We know there is no negative cycle, therefore we can remove some edges from that cycle and find a shorter path: Contradiction: no shortest path with cycle can exist.

After the first traversal over all the edges, the shortest paths to all vertices that are 1 edges away of from s will be found.
Use induction

- Inductive Argument: after k^{th} iterations: the shortest paths to all vertices that are k edges away
- Base case: iteration 0
- Induction step: if the distance is correctly selected for all nodes that are k-1 edges away from the source node, in the k^{th} step, the shortest distance to all the next nodes will be found.

3. Exercise 24 on page 331 (20 points)

Algorithm (14 points)

opt[p_idx, num1, A_count1, A_count2]

- p_idx: precincts 1 to p_idx are being considered
- Num1: number of precincts in district 1
    - There would be (p_idx – num1) precincts in district 2 obviously: don't need to add it as an argument to opt
- A_count1: number of votes in favor of A in district 1
- A_count2: number of votes in favor of A in district 2

opt[p_idx, num1, A_count1, A_count2] there are two cases:

- We denote the number of votes that A has in precinct[p_idx] as: va

1. We either put the precinct[p_idx] in district one
    - opt[p_idx - 1, num1 - 1, A_count1 – va, A_count2]
    - Explanation: after adding precinct[p_idx] to district 1, we want the total votes to be "num1" and the number of distincts to be "A_count1"
2. Or we put it in district 2
    - opt[p_idx - 1, num1 , A_count1, A_count2 - va]

To find the answer, we check opt[n, n/2, A_count1, A_count2] for all A_count1 > mn/4, A_count2 > mn/4

- A will win each district having at least mn/4 votes in that district.

Base case:

- OPT[0,0,0,0] = True
- OPT[0, x, y, z] = False for all other x, y, z

Time Complexity (4 points)

We have these conditions over variables

- $p_{idx} \in [0, n]$
- $num1 \in [0, n/2]$
- $A\_count1 \in [0, mn]$
- $A\_count2 \in [0, mn]$

We can calculate time complexity now:

- Number of subproblems: $O(n^4m^2)$
- each subproblem can be solved in O(1)

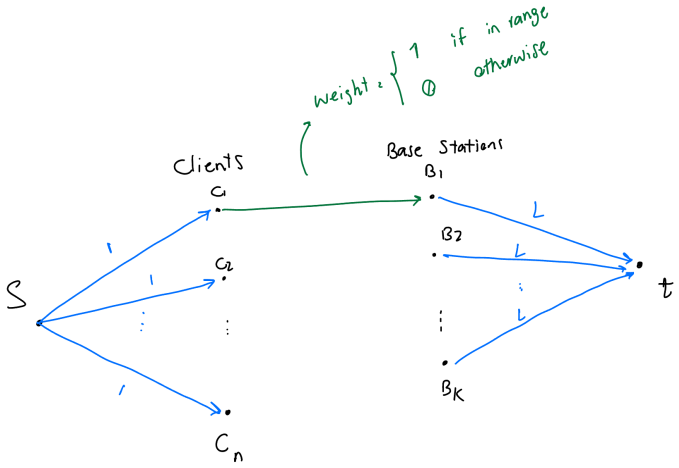
Proof (2 points)

Induction. Use the logic of the algorithms section

4. Exercise 7 on page 417 (15 points)

Algorithm (10 points)

Form the s-t network as follows



Run Ford-Fulkerson and find the maximum flow. If $|f| < n$, then it means all the clients cannot be connected to exactly one station.

Time Complexity (3 points)

Forming the network: nk

Time complexity of Ford-Fulkerson is $O(|f|e)$

Calculating e

- Source-client edges: n
- Client-station edges: O(nk)
- Station-target node edges: k

We also know that $|f| \leq Lk$

- Equal if all edges to target are saturated

So, $O(nLk^2)$ for running Ford-Fulkerson

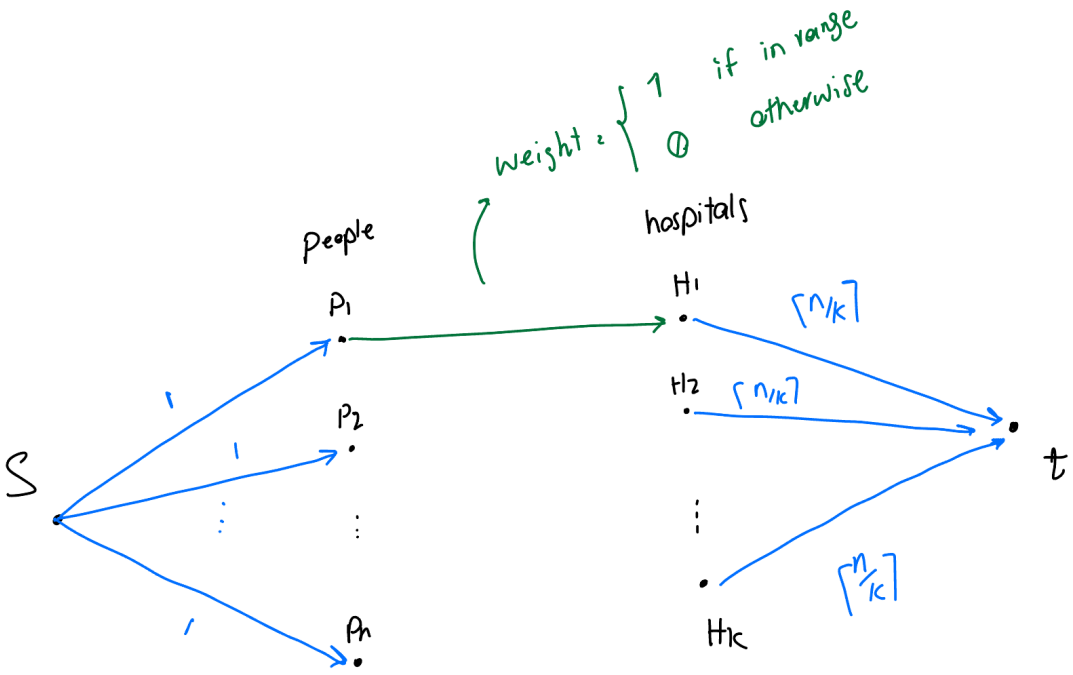
Proof (2 points)

Conditions:

- Range: condition satisfied because of the weight of client-station edges
- Load: Satisfied because of weight of station-t edges
- Each client must be connected to **exactly** one base station:
    - Clients cannot be connected to more than one base station because the incoming flow to clients is one (and conservation rule)
    - Clients must at least be connected to one station, or we notice it when checking the final flow.

5. Exercise 9 on page 419 (15 points)

Exactly the same as problem 4.



6. (15 points) Given a sequence of numbers find a subsequence of alternating order, 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 (8 points)

For index i:

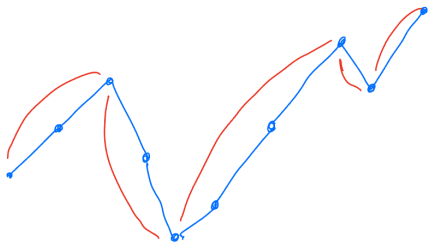
- If it was larger than element i-1
  - Opt(i, increasing=True) = opt(i-1, increasing=False) + 1
  - Opt(i, increasing=False) = opt(i-1, increasing=False)
- when element[i] < element[i-1]:
  - Opt(i, increasing=True) = opt(i-1, increasing=True)
  - Opt(i, increasing=False) = opt(i-1, increasing=True) + 1

Base case: opt(1, True) = opt(1, False) = 1

Proof (5 points)

A few points:

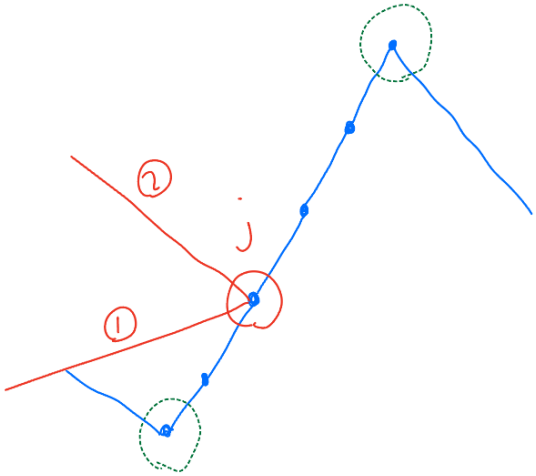
- Any input can be separated into contiguous subsequences of non-decreasing and non-increasing order. We call these monotonic subsequences.



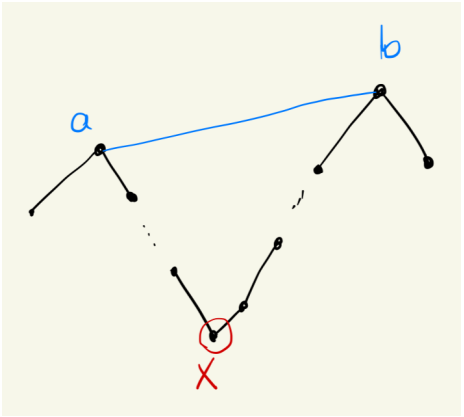
  - Our algorithm is selecting the start and end of monotonic subsequences.
- At most two points from each monotonic subsequence can be selected.
  - Three of them in a row will not form an alternating subsequence

- On a monotonic subsequence, if a middle node is selected for the longest alternating subsequence (LAS), we can safely replace it with the start or the end node of that subsequence.
  - Without loss of generality: assume this monotonic subsequence is non-decreasing.
  - Assume node j is selected

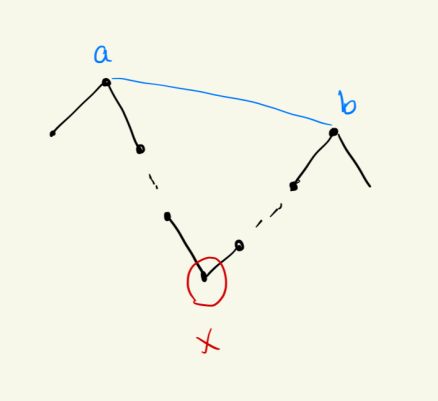
- Case 1: the node before j on LCA is smaller than j
    - Replace j with the last node on the monotonic subseq
- Case 2: the node before j is larger than j
    - Replace j with the start of mon. subseq.



- All of the starts and ends of mon. subseqs must be selected.
    - Proof by contradiction
    - Assume a node x with this property is not selected
        - Remember that we already proved we only need to consider peak or a valley nodes.
        - a: the last selected node before x
        - b: the first node after x
            - b > a is a contradiction
                - adding x increases the length of las: it cannot be longest



- So we have b < a:



- We can safely replace b with x

Time complexity analysis (2 points)
One traversal, O(1) operation per element: O(n) algorithm

You can solve this with pure dynamic programming is $O(n^2)$. Hint:
las(i, increasing=False)

- for all j < i where A[j] < A[i] find the maximum
  las(i, increasing=True) + 1

las(i, increasing=True)

- exact opposite of previous case

proof: use induction

Rubric explanation: both O(n) algorithm (greedy based) and $O(n^2)$ pure dynamic programming solutions get full credit.