CS 180 Homework 6

Name: Wang, Zheng
UID: 404855295

CS 180 Dis 1E
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 T has root r and r has children $v_1, v_2, \dots, v_k$. Let the subtree rooted on v_i be T_{v_i}. Now, assume the minimum round needed to notify everyone in each subtree T_{v_i} is $opt(T_{v_i})$. Without loss of generality, assume $opt(T_{v_1}) \geq opt(T_{v_2}) \geq \cdots \geq opt(T_{v_k})$.

Claim: if r notify all its children according to the order $v_1, v_2, \dots, v_k$ (call this scheduling G), the total round taken will be minimized.

Proof. Let's suppose there is some other scheduling G' for r with a different ordering of v_i's (not sorted). Then, there must exist some neighboring pair v_s, v_t such that v_s is before v_t, but $opt(T_{v_s}) < opt(T_{v_t})$.

Let's say that the maximum round taken for everyone except for v_s, v_t, and their decedents is M. Then, right now the total round to notify everyone in T is

$$\mathrm{R} = \max(\, M, \max\left(s + opt(T_{v_s}), s + 1 + opt(T_{v_t})\right)\,)$$

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

$$\mathrm{R}' = \max(\, M, \max\left(s + opt(T_{v_t}), s + 1 + opt(T_{v_s})\right)\,)$$

We notice that $s + opt(T_{v_t}) < s + 1 + opt(T_{v_t})$ and also $s + 1 + opt(T_{v_s}) < s + 1 + opt(T_{v_t})$. Thus, we have

$$\max\left(s + opt(T_{v_t}), s + 1 + opt(T_{v_s})\right) < \max\left(s + opt(T_{v_s}), s + 1 + opt(T_{v_t})\right)$$

So

$$\max(M, \max\left(s + opt(T_{v_t}), s + 1 + opt(T_{v_s})\right))$$
$$\leq \max(M, \max\left(s + opt(T_{v_s}), s + 1 + opt(T_{v_t})\right))$$

Thus, $R' \leq R$ and we will not worsen the total round take to notify everyone if we switch v_s and v_t. If we keep doing so, we will eventually get G. Thus, G is optimal.

Using the idea of dynamic programming, we then need to find $opt(T)$. But this is easy and we find $opt(T) = \max_{1 \leq i \leq k}(i + opt(T_{v_i}))$ (Assuming $opt(T_{v_1}) \geq opt(T_{v_2}) \geq \cdots \geq opt(T_{v_k})$). By the *claim*, we know that if we call subordinates of r in the order $v_1, v_2, \ldots, v_k$, we can minimize the total round need to notify everyone in T. Thus, the total round taken to notify everyone is going to be the maximum round for a particular subtree to be all notified, AKA $\max_{1 \leq i \leq k}(i + opt(T_{v_i}))$.
The base case here will be when T_L is a leaf, here we use $opt(T_L) = 0$.
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 T is r):

```
➢ Define function output_calling_sequence($O$)
    o if $n$ is a leaf
        ▪ stop
    o output the sorted subordinate list of $O$
    o for each subordinate $s_i$ in the sorted subordinate list of $O$
        ▪ output_calling_sequence($s_i$)
➢ Initialize the tree $T$ with officer $v$ along with $opt(T_v)$ as each node
➢ Run BFS on tree $T$ to get the level each officer is located on $T$
➢ for each leaf $L_i$ of $T$
    o set $opt(T_{L_i}) = 0$
➢ for each level of $T$ from bottom to top
    o for each officer $O$ at that level, if $opt(T_O)$ is NOT initialized
        ▪ sort the subordinates $s_i$ of the officer $O$ by $opt(T_{s_i})$ in decreasing order
        ▪ for each subordinate $s_i$ in the sorted list ($i = 1, \ldots, k$)
            • $opt(O) = \max_{1 \leq i \leq k}(i + opt(T_{s_i}))$
    o end for
```

- end for
- call output_calling_sequence(r)
- output $opt(T_r)$

Complexity:

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

Thus, the overall complexity of this algorithm is $O(n\log n)$.

Question 2 (Exercise 21, page 330):

Explanation:

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

We will first define $S[i][j]$ $(i \le j)$ being the maximum money we can earn in the interval from day i to day j by making a single transaction.

How to find $S[i][j]$? We see that there are two cases:

1. If we buy the shares on day i and sell it on day j. Then the amount of money we can obtain is $1000 \times (p(j) - p(i))$
2. Otherwise, the transaction happens either in the interval $(i, j-1)$ or $(i+1, j)$. The maximum money we can earn is thus $S[i][j-1]$ and $S[i+1][j]$ resepectively.

Thus, $S[i][j] = \max(1000 \times \big(p(j) - p(i)\big),\ S[i][j-1],\ S[i+1][j])$

For the base cases, we know that $S[i][i] = 0$ for all i. Using the idea of dynamic programming, we can easily calculate all $S[i][j]$ from the base cases.

The next step is to calculate the optimal money we can earn from day 1 to day j, where $j = 1,2,\dots,n$, by making exactly m $(m = 1,2,\dots,k)$ transactions. Let $opt[j][m]$ denote this value.

Then, we see $opt[j][m] = \max\limits_{2\le q\le j}(opt[q-1][m-1] + S[q][j])$. The reason to get this equation is that we can divide the interval $(1, j)$ into to two part $(1, q-1)$ and (q, j), we define that in (q, j) we are making the last transaction. Then, we see that the best profit we can achieve for the last transaction in interval (q, j) is $S[q][j]$. The best profit we can achieve for all rest transaction is $opt[q-1][m-1]$ After trying all possible q, we will find the best profit we can achieve by making m transaction in interval $(1, j)$, i.e. $opt[j][m]$.

For the base cases, we see that $opt[1][m] = 0 \quad \forall m$, since we cannot earn anything or loss anything with in a single day (from day 1 to day 1). We also see that $opt[j][1] = S[1][j] \quad \forall j$ since if we do not make transaction, we will not earn nor loss any money. Finally, when we have to much transaction ($m > j/2$), set such $opt[j][m] = 0$.

In order to output the k-shot we are going to use, simply back track and output each transaction happened whenever a $S[q][j]$ is included for $opt[j][m]$.

Algorithm (assume 1 indexing):

```
➢ Initialize 2d array $interval$ that is of size $n * n$
➢ Initialize 2d array $S$ of size $n * n$
➢ Initialize 2d array $opt$ of size $n * k$
➢ For $i = 1,2,\dots,n$
    o For $j = 1,2,\dots,n$
        ▪ $interval[i][j] = (i, j)$
    o End for
➢ End for
➢ For $i = 1,2,\dots,n$
    o Set $S[i][i] = 0$
➢ End for
➢ For $(diag = 3;\ diag \le n;\ diag{+}{+})$
    o For $(i = 1, j = diag;\ \ j \le n;\ \ i{+}{+},\ j{+}{+})$
        ▪ $S[i][j] = \max(1000 \times (p(j) - p(i)),$
                    $S[i][j-1],\ S[i+1][j]\ )$
        ▪ if $S[i][j]$ is either $S[i][j-1]$ or $S[i+1][j]$
            • let $S[i][j]$ points to the one that is larger
        ▪ else
            • let $S[i][j]$ points to $interval[i][j]$
    o end for
➢ end for
➢ for $j = 1,2,\dots,n$
    o for $m = 1,2,\dots,k$
        ▪ if $j$ is 1
            • $opt[j][m] = 0$
        ▪ Else if $m > j/2$
            • $opt[j][m] = 0$
        ▪ Else if $m$ is 1
            • $opt[j][m] = S[1][j]$
            • Let $opt[j][m]$ points to $S[1][j]$
        ▪ Else
```

```
                • $opt[j][m] = \max_{2\le q\le j}(opt[q-1][m-1] + S[q][j])$
                • Let $opt[j][m]$ points to $S[q][j]$ and $opt[q-1][m-1]$ that maximize $opt[q-1][m-1] + S[q][j]$
        o End for
  ➢ End for
  ➢ Output $\max(0, \max_{1\le m\le k}(opt[n][m]))$
  ➢ If we output 0
        o output "no transaction made"
  ➢ Else
        o Initialize a stack $opt$
        o Follow the pointer of $opt[n][m^*]$ ( $m^*$ is the maximum of $opt[n][m]$ $\forall m$) to an element of $S$ and trace back from that until we get an element from $interval$. Push that element in a stack $out$
        o Follow the pointer of $opt[n][m^*]$ to the previous element of $opt$, repeat above until there is no previous $opt$ element
  ➢ Output and pop out each element in $out$ to show the k-shot found
```

Complexity:

1. Initializing $interval$ and S will take $O(n^2)$ to complete
2. To fill opt, each element can be filled with $O(n)$, and we are filling $n*k$ elements. Thus, filling opt will take $O(kn^2)$ to complete
3. Outputting the maximum return will take $O(n)$ to complete
4. Backtracking to find the each $interval$ element will take $O(n)$ to complete. This is because we will in maximum follow n pointer in S before reaching an $interval$ element. Since we are finding in maximum $O(n)$ $interval$ elements, this step will take $O(n^2)$ to complete.

Thus, the overall time complexity of this algorithm will be $O(kn^2)$

Question 3 (Exercise 24, page 331):

Explanation:

Let's denote the two parties A and B, the two districts D_1 and D_2, and the precincts be $P = \{P_1, P_2, \dots, P_n\}$.

Without loss of generality, suppose there is a way partition $\{P_1, P_2, \dots, P_n\}$ into D_1 and D_2 such that party A can win in both districts, then the vote obtained by A, V_A, satisfies $V_A \ge \frac{mn}{2} + 2$ votes. This is because to win in both districts, we must have $V_A = V_A^{D_1} + V_A^{D_2} \ge \frac{mn}{4} + 1 + \frac{mn}{4} + 1 = \frac{mn}{2} + 1$ ($V_A^{D_1}, V_A^{D_2}$ are votes A gets for D_1 and D_2 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 $V_A \geq \frac{mn}{2} + 2$ has already satisfied.

We see that the precincts are susceptible to gerrymander if and only if there is subset D_1 of size $n/2$ of $P = \{P_1, P_2, \dots, P_n\}$ such that $V_A^{D_1}$ satisfies:

$$\frac{mn}{4} < V_A^{D_1} < V_A - \frac{mn}{4}$$

Proof.

If the precincts are susceptible to gerrymander, by definition, there is a subset D_1 of size $n/2$ of P and $D_2 = P \backslash D_1$ such that $V_A^{D_1} > \frac{mn}{4}$ and $V_A^{D_2} = V_A - V_A^{D_1} > \frac{mn}{4}$. Thus, $\frac{mn}{4} < V_A^{D_1} < V_A - \frac{mn}{4}$.

Conversely, if there exist subset D_1 of size $n/2$ of P such that $V_A^{D_1}$ satisfies $\frac{mn}{4} < V_A^{D_1} < V_A - \frac{mn}{4}$, then let $D_2 = P \backslash D_1$, we have $V_A^{D_1} > \frac{mn}{4}$ and $V_A^{D_2} = V_A - V_A^{D_1} > \frac{mn}{4}$. 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 T satisfies $V_T \geq \frac{mn}{2} + 2$. 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 $G[p][s][v]$ be $true$ if for a set of precincts $\{P_1, P_2, \dots, P_p\}$, there exist some subset D_1 of size s such that party T can win v votes in D_1. $G[p][s][v]$ will be $false$ otherwise.
Then, we see that when we include P_p in the set of precincts, there are two cases.

1. We include it in D_1
2. We do not include it in D_1

Thus, $G[p][s][v] = true$ if at least one of the following is $true$ and is $false$ otherwise:

1. $G[p-1][s][v]$ is $true$ (when we do not include P_p in D_1)
2. $G[p-1][s-1][v-t_p]$ is $true$ (t_p the number of votes party T gets, this correspond to the case when we include P_p in D_1)

Then, we see that the base cases are $G[1][s][v] = true$ if $s = 1, v = t_1$ and $false$ otherwise.
Also, $G[p][0][0] = true$, otherwise, if any of p, s, v less or equal to 0, $G[p][s][v] = false$.

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

```
Initialize a 3d array $G$ with size $n * n * mn$
Set $V_A = \sum_{\forall i} a_i$
Set $V_B = \sum_{\forall i} b_i$
If $V_A < \frac{mn}{2} + 2$ and $V_B < \frac{mn}{2} + 2$
    Return $false$
    Exit the program
Else set $T$ to be the PARTY with larger total vote ($V$)
For $s = 1,2, \dots, n$
    For $v = 1,2, \dots, mn$:
        Set $G[1][s][v] = false$
    End for
End for
Set $G[1][1][t_1] = true$
For $p = 2,3, \dots, n$
    For $s = 1,2, \dots, n$
        For $v = 1,2,3, \dots, mn$
            If $G[p-1][s][v] = true$
                Then $G[p][s][v] = true$
            Else if $s - 1 = 0$ and $v - t_p = 0$
                Then $G[p][s][v] = true$
            Else if $s - 1 > 0$ and $v - t_p > 0$
            and $G[p-1][s-1][v - t_p] = true$
                Then $G[p][s][v] = true$
            Else
                Then $G[p][s][v] = false$
        End for
    End for
End for
Let $V_T$ be the larger one of $V_A$ and $V_B$
For $\frac{mn}{4} < v < V_T - \frac{mn}{4}$
    If any of the element in $G[n]\left[\frac{n}{2}\right][v]$ is $true$
        Return $true$ and end the program
End for
Return $false$
```

Complexity:

1. Initialize all the elements in $p = 1$ will take $O(n^2m)$ to complete.
2. The nested for loop to fill the recursive cases will take $O(n^3m)$ to complete
3. To check If any of the element in $G[n]\left[\frac{n}{2}\right][v]$ is $true$, where $\frac{mn}{4} < v < V_T - \frac{mn}{4}$ will take $O(nm)$ to complete

Thus, the overall complexity of this algorithm is $O(n^3m)$.

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

Let the fixtures be $f_1, f_2, \dots, f_n$, the switches be $s_1, s_2, \dots, s_n$, and the wall be $w_1, w_2, \dots, w_m$.

The idea to solve this problem is to create a directed weighted graph G with nodes $S, T, f_1, f_2, \dots, f_n$, $s_1, s_2, \dots, s_n$, where S is a pseudo starting node and T being a pseudo target node.

The edges are then draw according the the following rules:

1. Draw an edge from S to each of $f_1, f_2, \dots, f_n$; all of these edges have capacity 1
2. Draw an edge from each of $s_1, s_2, \dots, s_n$ to T; all of these edges have capacity 1
3. If the line defined by end points s_i and f_j does not cross any wall, then draw an edge from f_j to s_i; all of these edges have capacity 1

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

Claim:

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

Proof:

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

Then if maximum flow is smaller than n, then that means at least one s_k has no incoming flow (a match means that there is a flow of 1 from $f_{()}$ to $s_{()}$), then that means not all s has been matched with a f.

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

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 $G$ with vertex $S, T, f_1, f_2, \ldots, f_n, s_1, s_2, \ldots, s_n$
➢ put an edge from $S$ to each of $f_1, f_2, \ldots, f_n$; all of these edges have capacity 1
➢ put an edge from each of $s_1, s_2, \ldots, s_n$ to $T$; all of these edges have capacity 1
➢ For each of $f_i$ in $f_1, f_2, \ldots, f_n$
    o For each of $s_j$ in $s_1, s_2, \ldots, s_n$
        ▪ For each of $w_k$ in $w_1, w_2, \ldots, w_m$.
            • If line $(f_i, s_j)$ intersect with $w_k$
                o Continue
        ▪ End for
        ▪ Add edge from $f_i$ to $s_j$ with capacity 1
    o End for
➢ End for
➢ Run Ford-Fulkerson Algorithm on $G$, from $S$ to $T$, to get maximum flow, $flow$
➢ If $flow$ is equal to $n$
    o Return ergonomic
➢ Else
    o Return not ergonomic
```

Complexity:

1. To construct the graph G, we have to add all vertices, which take $O(n)$ to complete. We will add all $S - f_{()}$ edges and $s_{()} - T$ edges, which take $O(n)$ to complete. Finally, we for each of the pair $f_i - s_j$ we check it against all of the walls to determine if we put an edge or not. There are $O(n^2)$ pairs and there are $O(m)$ walls. Thus, this step will take $O(n^2m)$ to complete.
2. Then, we simply run Ford-Fulkerson Algorithm on G. Since there are $n^2 + 2n$ edges in maximum in G; the maximum flow out of S is n. Thus, Ford-Fulkerson Algorithm will take $O\left(n(n^2 + 2n)\right) = O(n^3)$ to run.

Thus, the overall complexity will be $O(n^3 + mn^2)$

Question 5 (Exercise 8, page 418):

(a)

Explanation:

To solve this problem, we will construct a weighted directed graph with nodes $S, T, O, A, B, AB, O', A', B', AB'$.

We first create edges from S to each of O, A, B, AB, the capacity of each edge is the supply of that type of blood. For example, edge (S, O) has capacity s_O and edge (S, AB) has capacity s_{AB}.

Then, we create edges from each of O', A', B', AB' to T. The capacity of each edge is the demand of that type of blood. For example, edge (O', T) has capacity d_O.

Finally, we add edges from $S = \{O, A, B, AB\}$ to $S' = \{O', A', B', AB'\}$. If a blood type Y in S can donate blood to a blood type X' in S', then draw and edge from Y to X'. Each of the edges in this step has capacity ∞.

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

Claim:

If and only if the maximum flow is the sum of the demand, Σ_d ($\Sigma_d = d_O + d_A + d_B + d_{AB}$), the supplies will fulfill the demands.

Proof.

If there is enough supply, then each of the node in $S' = \{O', A', B', AB'\}$ can receive at least the demand they need. So, $flow_{in}(X') \geq d_{X'}$ for all $X' \in S'$. Thus, the flow out of S' will saturate the edge (X', T) for all $X' \in S'$. In another word, Σ_d is equal to maximum flow.

Conversely, if Σ_d is equal to maximum flow, then it is necessary that the edge (X', T) for all $X' \in S'$ is saturated. Thus, $flow_{in}(X') = flow_{out}(X') = d_{X'}$ for all $X' \in S'$. Thus, all of the blood type in S' (which represent receiver of the blood) will get amount that satisfy the demand.

Algorithm:

```
➢ Initialize graph $G$ with nodes $S, T, O, A, B, AB, O', A', B', AB'$.
➢ For $x$ in $O, A, B, AB$
    o Add edge from $S$ to $x$ in $G$
➢ For $x'$ in $O', A', B', AB'$
    o Add edge from $x$ to $T$ in $G$
➢ For $x$ in $O, A, B, AB$
    o For $x'$ in $O', A', B', AB'$
        ▪ If blood type $x'$ can receive blood type $x$
            • Add an edge from $x$ to $x'$ in $G$
    o End for
➢ End for
➢ Run Ford-Fulkerson Algorithm on $G$, from $S$ to $T$, to get maximum flow, $flow$
➢ If $flow$ is equal to $\Sigma_d = d_O + d_A + d_B + d_{AB}$
    o Return satisfied
➢ Else
    o Return not satisfied
```

Complexity:

1. We see that the graph is constant. Thus, creating the graph take $O(1)$ time to compete.
2. We then run the Ford-Fulkerson Algorithm, this will take $O(e * C)$, where e

is the number of edges in G, and C is the maximum flow out of S, which is the sum of supplies of all types of blood (i.e. $\sum_{x\in\{O,A,B,AB\}} s_x$). Thus, as the graph is a constant, the Ford-Fulkerson Algorithm will run with time complexity $O(\sum_{x\in\{O,A,B,AB\}} s_x)$.

Thus, the overall time complexity is $\sum_{x\in\{O,A,B,AB\}} s_x$.

(b)

The cut can be $\{S, AB, B, AB', B'\}$ and $\{T, O, O', A, A'\}$. This cut has capacity $56 + 36 + 10 + 3 = 99$, and thus cannot fulfill the maximum flow of 100 required.

The explanation: since the demand of O and A can only be fulfilled by type O and A. As there are in total 86 unit of supply of blood O and A. But there are 87 unit of demand of blood O and A. 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 S and let s_i denote the ith element of S.

We first notice that if we know the maximum length of the alternating sequence that end up all $s \in S$, 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 s, then there are only two cases:

1. $\cdots\cdots\cdots < s$
2. $\cdots\cdots\cdots > s$

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

1. Let $opt[1][i]$ denote the maximum length of the alternating sequence that end up with the ith element of S, given that the alternating subsequence satisfies $\cdots\cdots\cdots < s_i$ (s_i denote the ith element of S)
2. Let $opt[2][i]$ denote the maximum length of the alternating sequence that end up with the ith element of S, given that the alternating subsequence satisfies $\cdots\cdots\cdots > s_i$ (s_i denote the ith element of S)

Then, we see that $opt[1][i] = \max\limits_{1\le j<i \text{ and } s_j<s_i}(opt[2][j]) + 1$. This is because for alternating subsequence to end up with $\cdots\cdots\cdots < s_i$, the subsequence before it must be $\cdots\cdots\cdots > s_j < s_i$. Thus, we search elements in $opt[2][*]$.

Similarly, we see that $opt[2][i] = \max\limits_{1\le j<i \text{ and } s_j>s_i}(opt[1][j]) + 1$.

Then, we see that the base cases here is $opt[1][1] = 1$ and $opt[2][1] = 1$.

Algorithm

- Initialize a 2d array opt with size $2 * n$
- Set $opt[1][1] = 1$

```
➢ Set $opt[2][1] = 1$
➢ For $i = 2,3,\dots,n$
      o Set $opt[1][i] = 1$
      o Set $opt[2][i] = 1$
➢ For $i = 2,3,\dots,n$
      o For $j = 1,2,\dots,i$
            ▪ If $s_j < s_i$
                  • $opt[1][i] = \max(opt[1][i], opt[2][j] + 1)$
                  • If $opt[1][i]$ has been updated
                        o Let $opt[1][i]$ points to $opt[2][j]$
            ▪ If $s_j > s_i$
                  • $opt[2][i] = \max(opt[2][i], opt[1][j] + 1)$
                  • If $opt[2][i]$ has been updated
                        o Let $opt[2][i]$ points to $opt[1][j]$
      o End for
➢ End for
➢ Output the maximum element in $opt$ table
➢ Trace back all of the pointers of the maximum element. The result should be a
  sequence of $opt$ elements. Reverse this sequence of $opt$ (call this $out\_opt$).
  Then get the second index of all elements in $out\_opt$ (i.e. the index number
  in the second [] of $opt$). The result is an index sequence $i_1, i_2, \dots, i_k$, where
  $i_1 < i_2 < \cdots < i_k$.
➢ Output the subsequence by output $s_{i_1}, s_{i_2}, \dots, s_{i_k}$.
```

Complexity:

1. Initialization will take $O(n)$ to run
2. The nested for loop will take $O(n^2)$ time to complete since for each i we have in maximum n j's to check, and there are n i's.
3. To find the maximum element will take $O(n)$
4. To output the subsequence will take $O(n)$

Thus, the overall time complexity of this algorithm will be $O(n^2)$.