Brief solutions to hw3

1. **Solution to problem 1**

   Run Dijkstra's algorithm twice, once from s and once from t to compute shortest s-v paths and shortest t-v paths in G. Store the computed distances in arrays d_s and d_t, respectively.

   Then iterate over all pairs of cities in the list E' to find the pair (x, y) that incurs the minimum s-t distance. For a fixed pair (x, y), the latter is given by

   $$\min\{d_s(x) + \ell(x, y) + d_t(y), d_s(y) + \ell(x, y) + d_t(x)\}$$

   Running time: the total running time is Dijkstra's plus $O(|E'|)$. Depending on $|E'|$, we can decide which implementation to use.

2. **Solution to problem 2**

   Suppose n boxes arrive in the order $b_1, b_2, \ldots, b_n$ and weights $w_1, w_2, \ldots, w_n$.

   Greedy algorithm: Pack the boxes in the order they arrive; when the next box does not fit, send the truck on its way.

   Correctness: We will show that the greedy algorithm "always stays ahead" of any other solution (similarly to the proof of Dijkstra's algorithm). To do so, it suffices to prove the following claim *(why?)*: If the greedy algorithm loaded the first m trucks with p boxes, then no other algorithm can fit more boxes in the first m trucks. The proof of the claim is by a straightforward induction on m (left as an exercise).

3. **Solution to problem 3**

   Let

   $OPT(n)$ = min total cost to place copies in n libraries, *given that* the book is placed in L_n.

   Suppose we knew that in the optimal solution, L_i is the last library before L_n that contains the book, where $0 \leq i \leq n-1$. Then a request at library L_k with $i+1 \leq k \leq n-1$ will incur a delay of $n-k$. So the total user delay associated with libraries $L_{i+1}, \ldots, L_{n-1}$ is

   $$\sum_{k=i+1}^{n-1}(n-k) = \sum_{r=1}^{n-i-1} r = \frac{(n-i-1)(n-i)}{2} = \binom{n-i}{2}$$

   Then the minimum total cost is given by the following recurrence:

   $$OPT(n) = c_n + \binom{n-i}{2} + OPT(i)$$

   Since we don't know the index i a priori, we will look for the i that minimizes the above expression over all possible values for i. Therefore the recurrence is

   $$OPT(j) = c_j + \min_{0 \leq i < j}\left\{\binom{j-i}{2} + OPT(i)\right\}$$

   - We want $OPT(n)$.
   - Boundary conditions: $OPT(0) = 0$.

- Running time: $O(n^2)$; there are n subproblems, each requiring $O(n)$ time to fill in.
- Space: $O(n)$; maintain an array M such that $M[i] = OPT(i)$.

An algorithm similar to the OPTSegmentation algorithm from the lecture slides can be used to reconstruct an optimal solution given M.

4. **Solution to problem 4**

Let $OPT(n, m)$ be the distance between $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$.

Then either a_n is not mapped to b_m in the optimal solution or it is. Hence

- $OPT(n, m) = OPT(n, m-1)$, if a_n is not mapped to b_m; or
- $OPT(n, m) = |a_n - b_m| + OPT(n-1, m)$, if a_n is mapped to b_m

Therefore

$$OPT(n, m) = \min\{OPT(n, m-1), |a_n - b_m| + OPT(n-1, m)\}$$

- We want $OPT(n, m)$.
- Boundary conditions: $OPT(i, 1) = \sum_{k=1}^{i} |a_k - b_1|$ for $i \geq 0$, $OPT(0, j) = 0$ for $j \geq 1$
- Time: $O(nm)$ (there are nm subproblems, each requires $O(1)$ time)
- Space: $O(nm)$ (DP table $M[0..n, 1..m]$; *(space can be improved)*)
- Order to fill in table: column-by-column.

5. **Solution to problem 5**

We want to cut a string $s_1 s_2 \ldots s_n$ of length n into $m+1$ pieces, where we know the positions of the m cuts in advance: the cuts will be made at positions $p_1, \ldots, p_m$ with $0 < p_1 \leq p_2 \leq \ldots \leq p_m < n$. If we let $p_0 = 0$ and $p_{m+1} = n$, then $s_1 \ldots s_n = s_{p_0+1} \ldots s_{p_{m+1}}$.

Let

$OPT(0, m+1) =$ min cost to cut the entire string $s_{p_0+1} \ldots s_{p_{m+1}}$ at *fixed* positions $p_1, \ldots, p_m$.

More generally, for $0 \leq i < j \leq m+1$, let

$$OPT(i, j) = \text{ min cost to cut the string } s_{p_i+1} \ldots s_{p_j} \text{ at } \textit{fixed} \text{ positions } p_{i+1}, \ldots, p_{j-1}$$

Consider the overall optimal solution: the string consists of two pieces since every time we cut the string into two parts. Suppose we knew that the string was first cut at position p_{k^*} in the optimal solution. Then

$$OPT(0, m+1) = OPT(0, k^*) + OPT(k^*, m+1) + (n - 0)$$

Since we do not know the index k^*, and we want to minimize the overall cost, we have

$$OPT(0, m+1) = \min_{0<k<m+1} \{OPT(0, k) + OPT(k, m+1) + (p_{m+1} - p_0)\}$$

For subproblem $OPT(i, j)$, the recurrence becomes

$$OPT(i, j) = \min_{i<k<j} \{OPT(i, k) + OPT(k, j) + (p_j - p_i)\}$$

- Boundary conditions: $OPT(i, i+1) = 0$;
- Time: $O(m^3)$; there are $O(m^2)$ subproblems, each requiring $O(m)$ time
- Space: $O(m^2)$; need fill in the upper half of an $(m+1) \times (m+1)$ matrix; each entry in the matrix corresponds to a subproblem $OPT(i, j)$ for $0 \leq i < j \leq m+1$.
- Fill in the matrix diagonal by diagonal.

Solutions to Recommended Exercises

1. **Solution to recommended exercise 1**

   A greedy algorithm that sorts the customers by increasing order of service times and services them in this order is correct.

   Running time: $O(n \log n)$.

   Correctness: by an exchange argument; for any ordering of the customers, let c_j denote the j-th customer in the ordering. Then the total time is given by

   $$T = \sum_{i=1}^{n} \sum_{j=1}^{i-1} t_{c_j} = \sum_{i=1}^{n} (n-i) t_{c_i}$$

   One can observe that if $t_{c_i} > t_{c_j}$ for $i < j$, then swapping the positions of the two customers gives a better ordering. Then the ordering $t_{c_1} \leq t_{c_2} \leq \ldots \leq t_{c_n}$ provided by the greedy algorithm above is optimal.

2. **Solution to recommended exercise 3**

   Let $OPT(i)$ be the maximum revenue achievable up to week i.

   $$OPT(i) \quad = \quad \max\{OPT(i-1) + \ell_i, OPT(i-2) + h_i\}$$

   - We want $OPT(n)$.
   - Boundary conditions: $OPT(1) = \max\{\ell_1, h_1\}$.
   - Time complexity: $O(n)$; there are n subproblems, each requiring $O(1)$ time to fill in.
   - Space: maintain DP array M of size n *(can be improved)*

3. **Solution to recommended exercise 4**

   Let $OPT(i)$ = length of longest monotonically increasing subsequence **ending with** a_i

   Since the subsequence includes a_i, we have

   $$OPT(i) = 1 + \max_{\substack{1 \leq j < i \\ a_j < a_i}} OPT(j)$$

   - We want $\max\limits_{1 \leq i \leq n} OPT(i)$.
   - Boundary conditions: $OPT(0) = 0$.
   - Time complexity: $O(n^2)$; there are n subproblems to fill in, each requires $O(n)$ time in a bottom-up fashion; return their maximum in $O(n)$ time.
   - Space: $O(n)$

4. **Solution to recommended exercise 5**

   Let $OPT(j)$ = max sum of subarray **ending at** j. Then

   $$OPT(n) = \max\Big\{OPT(n-1) + A[n], 0\Big\}$$

   and

   $$OPT(j) = \max\Big\{OPT(j-1) + A[j], 0\Big\}.$$

- Boundary condition: $OPT(0) = 0$.
- We want the maximum length L of a contiguous subarray of A given by

$$L = \max_{1 \le j \le n} OPT(j)$$

- There are $n + 1$ subproblems, each requiring $O(1)$ time in a bottom-up computation. Thus computing L takes $O(n)$ time.
- Space: $O(n)$

5. **Solution to recommended exercise 6**

Let $OPT(i, j)$ = longest common substring of x, y **terminating at** x_i, y_j. Then

$$OPT(i, j) = \begin{cases} 1 + OPT(i-1, j-1) & \text{, if } x_i = y_j \\ 0 & \text{, otherwise} \end{cases}$$

- We want the subproblem of maximum value.
- Boundary conditions: $OPT(i, 0) = 0$ for all i, $OPT(0, j) = 0$ for all j.
- We can construct an $(m+1) \times (n+1)$ DP table to compute each $OPT(i, j)$. The longest common substring of x, y is given by the max entry in the table.
- Running time: $O(mn)$. There are $\Theta(mn)$ subproblems, each requires $O(1)$ time when computed in a bottom-up fashion. Returning the subproblem of maximum value takes $O(mn)$ time.
- Space: $O(mn)$ *(can be improved)*
- Fill in the DP table row-by-row.

Solutions to EdStem Further Practice Problems

1. **Solution to EdStem Further Practice Problem 1**

   (a) Let $OPT(W)$ be the max value we can get by using total weight at most W. The last item in an optimal solution achieving value $OPT(W)$ can be any of the input items, as long as its weight doesn't exceed W. Hence

   $$OPT(W) = \max_{1 \leq i \leq n} \{OPT(W - w_i)) + v_i, \text{ if } w_i \leq W\}$$

   More generally,

   $$OPT(w) = \max_{1 \leq i \leq n} \{OPT(w - w_i)) + v_i, \text{ if } w_i \leq w\}$$

   - We want $OPT(W)$.
   - Boundary conditions: $OPT(w) = 0$ for $w = 0$.
   - Time: $O(nW)$ (there are $O(W)$ subproblems, each requiring $O(n)$ time)
   - Space: $O(W)$

   (b) Let $OPT(n, W)$ be the max value we can get by selecting any subset of the first n items with total weight at most W. Let S^* be an optimal solution. Then the n-th item either appears in S^* or it doesn't; note that if $w_n > W$, then it cannot appear in S^*. Hence

   $$OPT(n, W) = \begin{cases} \max\left\{OPT(n-1, W), OPT(n-1, W - w_n) + v_n\right\} & \text{, if } w_n \leq W \\ OPT(n-1, W) & \text{, otherwise} \end{cases}$$

   More generally, for $0 \leq i \leq n$, $0 \leq w \leq W$, we have

   $$OPT(i, w) = \begin{cases} \max\left\{OPT(i-1, w), OPT(i-1, w - w_i) + v_i\right\} & \text{, if } w_i \leq w \\ OPT(i-1, w) & \text{, otherwise} \end{cases}$$

   - We want $OPT(n, W)$.
   - Boundary conditions: $OPT(0, w) = 0$ for $0 \leq w \leq W$, $OPT(i, 0) = 0$ for $1 \leq i \leq n$.
   - Time: $O(nW)$ (there are $O(nW)$ subproblems, each requiring $O(1)$ time)
   - Space: $O(nW)$ // *can be improved if we only care about optimal value— how?*
   - Order to fill in table: row by row

2. **Solution to EdStem Further Practice Problem 2**

   Let $OPT(i, j, k)$ be the length of the longest common subsequence of the prefixes of X, Y, Z ending at i, j, k respectively. Then

   $$OPT(i, j, k) = \begin{cases} 1 + OPT(i-1, j-1, k-1) & \text{, if } x_i = y_j = z_k \\ \max\{OPT(i-1, j, k), OPT(i, j-1, k), OPT(i, j, k-1)\} & \text{, otherwise} \end{cases}$$

   - We want $OPT(m, n, p)$.

- Boundary conditions: $OPT(0, j, k) = 0$ for all j, k; $OPT(i, 0, k) = 0$ for all i, k; $OPT(i, j, 0) = 0$ for all i, j.
- Time: $O(n^3)$, for $m, p = O(n)$; $O(n^3)$ subproblems, $O(1)$ time to fill in in a bottom-up fashion
- Space: $O(n^3)$, for $m, p = O(n)$ *// can be improved.*
- Fill in 3D matrix by increasing i, j, k in three nested for loops.

The reconstruction algorithm is recursive and linear in n for $m, p = O(n)$.