Assignment 3 Solutions

1 Longest common subsequence of many strings [10 marks]

Recall that a string $z = z_1 z_2 \cdots z_k$ is a *subsequence* of a string $x = x_1 x_2 \cdots x_n$ if there are indices $1 \leq i_1 < i_2 < \cdots i_k \leq n$ for which $z_1 = x_{i_1}$, $z_2 = x_{i_2}$, ..., $z_k = x_{i_k}$. The string z is a *common subsequence* of a set of m strings $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$ if and only if z is a subsequence of all m of these strings. For example, the string `ONI` is a common subsequence of the 3 strings

$$\begin{aligned} x^{(1)} &= \texttt{POLYNOMIAL} \\ x^{(2)} &= \texttt{EXPONENTIAL} \\ x^{(3)} &= \texttt{CONSTANTTIME} \end{aligned}$$

but the string `PAL` is not.

Design an dynamic programming algorithm that, given as input a set of m strings $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$ of lengths $n_1, \ldots, n_m$, respectively, outputs the length of the longest common subsequence of $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$.

Solution. Given a string $z = z_1, z_2, \ldots, z_n$ and some parameter $0 \leq k \leq n$, let $z_{[k]} = z_1, z_2, \ldots, z_k$ be the substring that includes only the first k characters of z. We will also write $[k] = \{1, 2, \ldots, k\}$.

Algorithm description. We obtain a dynamic programming algorithm by considering the following subproblems.

Subproblems. For $1 \leq i_1 \leq n_1$, ..., $1 \leq i_m \leq n_m$ define

$$M(i_1, \ldots, i_m) = \text{ length of the LCS of } x^{(1)}_{[i_1]}, \ldots, x^{(m)}_{[i_m]}.$$

Base cases. When any parameter $i_j = 0$, then

$$M(i_1, \ldots, i_{j-1}, 0, i_{j+1}, \ldots, i_m) = 0.$$

Recurrence. (When $i_1 > 0, \ldots, i_m > 0$.) There are two possibilities. If $x^{(1)}_{i_1} = x^{(2)}_{i_2} = \cdots = x^{(m)}_{i_m}$ then we can match those symbols and the length of the LCS satisfies

$$M(i_1, i_2, \ldots, i_m) = M(i_1 - 1, i_2 - 1, \ldots, i_m - 1) + 1.$$

Otherwise,

$$M(i_1, i_2, \ldots, i_m) = \max \begin{cases} M(i_1 - 1, i_2, \ldots, i_m) \\ M(i_1, i_2 - 1, \ldots, i_m) \\ \vdots \\ M(i_1, i_2, \ldots, i_m - 1). \end{cases}$$

Pseudocode. The algorithm is obtained by solving the subproblems in increasing order of the indices and returning the value $M(n_1, \ldots, n_m)$.

Algorithm 1: $\mathrm{LCS}_m(x^{(1)}, \ldots, x^{(m)})$

---

for $(i_1, i_2 \ldots, i_m) \in [n_1] \times [n_2] \times \cdots \times [n_m]$ **do**
  $M[0, i_2, \ldots, i_m] \leftarrow 0$;
  $M[i_1, 0, \ldots, i_m] \leftarrow 0$;
  $\vdots$
  $M[i_1, i_2, \ldots, 0] \leftarrow 0$;

for $(i_1, i_2 \ldots, i_m) \in [n_1] \times [n_2] \times \cdots \times [n_m]$ **do**
  if $x_{i_1}^{(1)} = x_{i_2}^{(2)} = \cdots = x_{i_m}^{(m)}$ **then**
    $M[i_1, i_2, \ldots, i_m] \leftarrow M[i_1 - 1, i_2 - 1, \ldots, i_m - 1] + 1$;
  else

$$M[i_1, i_2, \ldots, i_m] \leftarrow \max \begin{cases} M[i_1 - 1, i_2, \ldots, i_m] \\ M[i_1, i_2 - 1, \ldots, i_m] \\ \quad \vdots \\ M[i_1, i_2, \ldots, i_m - 1] \end{cases};$$

return $M[n_1, n_2, \ldots, n_m]$;

In the pseudocode, the notation in the for loops is shorthand for the m nested for loops

for $i_1 = 1, \ldots, n_1$
  for $i_2 = 1, \ldots, n_2$
    $\ddots$
      for $i_m = 1, \ldots, n_m$
        (Loop contents)

Proof of correctness. The base case computations are correct because if any string is empty, the set of strings contains no non-empty common subsequence.

The recurrence is correct in the case when $x_{i_1}^{(1)} = \cdots = x_{i_m}^{(m)} = s$ for some symbol s because in this case we can get a longest common subsequence by first getting the longest common subsequence of $x_{[i_1-1]}^{(1)}, \ldots, x_{[i_m-1]}^{(m)}$ and then appending the symbol s to it. This remains a subsequence of the m strings and it has length $M(i_1 - 1, i_2 - 1, \ldots, i_m - 1) + 1$. The recurrence is also correct when the last symbols of the m strings are not all identical because in this case the longest common subsequence of the m strings must skip the last symbol of at least one of those strings.

Time complexity analysis. Both **for** loops in the algorithm execute the code inside the loops $n_1 n_2 \cdots n_m$ times. The contents of each loop has time complexity $\Theta(m)$. The final time complexity of the algorithm is therefore $\Theta(m\, n_1\, n_2 \cdots n_m)$. (If all m strings have the same length n, this expression simplifies to $\Theta(m\, n^m)$.)

2 Bookshelf [10 marks]

In the LibraryBookshelf problem, you are given as input a set of n books and the thickness $t_i > 0$ of each book $i \in \{1, 2, \ldots, n\}$, along with the width W of the bookshelves that will be used to build the library.

Design a greedy algorithm that determines the minimum number of bookshelves required to store the books, keeping in mind that since the books have call numbers, you must place them in order in the bookshelves, but that you can choose how many go on a shelf.

Solution.

Algorithm description. The algorithm is remarkably simple: start filling the first shelf and keep adding books to it (in order) until the next book does not fit in the shelf. Then start a new shelf with this book, and continue adding books to it again until the next book does not fit in the current shelf. Repeat this process until all the books have been shelved then return the total number of shelves that were used.

Pseudocode. The algorithm itself is as follows.

Algorithm 2: Bookshelf$(t_1, \ldots, t_n, W)$

```
if n = 0 return 0;
numshelves ← 1;
curwidth ← t_1;
for i = 2, ..., n do
    if curwidth + t_i ≤ W then
        curwidth ← curwidth + t_i;
    else
        numshelves ← numshelves + 1;
        curwidth ← t_i;
return numshelves;
```

Proof of correctness.

1. (Exchange.) Let $A : \{1, \ldots, n\} \to \{1, \ldots, k\}$ be an optimal solution of size k (i.e. a mapping from books $\{1, \ldots, n\}$ to shelves $\{1, \ldots, k\}$) and let $G : \{1, \ldots, n\} \to \mathbb{Z}$ be the greedy solution. Suppose i is the first book such that $A(i) \neq G(i)$.

   Suppose $A(i) = A(i-1)$ (i.e. book i is on the same shelf as the previous book). Let $j < i$ be the first book assigned to the same shelf as i. Then $A(j) = G(j)$ and $\sum_{\ell=j}^{i} t_\ell \leq W$ so G could have assigned book i to shelf $A(i-1)$; this is a contradiction so we must have $A(i) \neq A(i-1)$.

   Suppose $A(i) = A(i-1) + 1$ so book i starts a new shelf in A; then since $G(i) \neq A(i)$ we must have $G(i) = G(i-1) = A(i-1)$. Let $j < i$ be the first book such that $G(j) = G(i)$.

$A(j) = G(j)$ and $\sum_{\ell=j}^{i} t_\ell \leq W$ so we can get a new optimal solution A' be setting $A'(i) \leftarrow G(i)$; thus A' is an optimal solution with one fewer difference from G.

2. (Induction.) By induction on the size k of the optimal solution.

   Base case: Suppose $k = 1$. Then $\sum_{i=1}^{n} t_i \leq W$ so the algorithm will never increment `numshelves`, and will return 1.

   Inductive step: Suppose the algorithm is correct for all inputs having optimal solution at most k, and let $t_1, \ldots, t_n$ be an input with optimal solution $k+1$. Let $A : \{1, \ldots, n\} \to \{1, \ldots, k+1\}$ be any optimal assignment of books to shelves, so book i is assigned to shelf $A(i)$, and let $A_1 = \{1, \ldots, m\}$ be the set of books assigned to the first shelf. Then $\sum_{i=1}^{m} t_i \leq W$ so the algorithm will not increment `numshelves` until after $i = m$. The books $\{m+1, \ldots, n\}$ fit on k shelves in the optimal solution so by the inductive hypothesis, the algorithm will find an assignment placing these books on at most k shelves. So the algorithm will return $k+1$.

Time complexity analysis. The **for** loop has $n-1$ iterations and each iteration of the loop has constant time complexity, so the total time complexity of the algorithm is $\Theta(n)$.

3 Hiring [10 marks]

In the Hiring problem, you own two companies that have salary budgets $B_1 > 0$ and $B_2 > 0$, respectively. There are n people $1, 2, \ldots, n$ applying to work at one of your companies, where person i requires salary s_i and would generate revenues $r_1(i) > 0$ and $r_2(i) > 0$ to companies 1 and 2, respectively.

A valid solution to the Hiring problem is the maximum value R for which there exists a pair of disjoint sets $S_1, S_2 \subseteq \{1, 2, \ldots, n\}$ that satisfy

(i) $\sum_{i \in S_1} s_i \leq B_1$,

(ii) $\sum_{i \in S_2} s_i \leq B_2$, and

(iii) $\sum_{i \in S_1} r_1(i) + \sum_{i \in S_2} r_2(i) = R$.

Design a dynamic programming algorithm to solve the Hiring problem. (Note that to solve the problem, the algorithm needs to find the maximum revenue R as described above, but it does not need to identify the sets S_1 and S_2.) In your description of the algorithm, make sure you clearly indicate what are the subproblems and the order in which they are solved. Also: your time complexity analysis should be in terms of n, B_1, and B_2.

Solution.

Algorithm description. The key idea is that we need to consider subproblems where we look at only the first $k \leq n$ *and* where we consider company budgets $b_1 \leq B_1$ and $b_2 \leq B_2$. This solution will only be valid when the budgets and salaries are non-negative integers.

Subproblems. Define $R(k, b_1, b_2)$ to be the maximum revenue of the two companies when $S_1, S_2 \subseteq \{1, 2, \ldots, k\}$ and when the companies have budgets b_1 and b_2, respectively.

Base cases. When $k = 0$ then for every $b_1 \leq B_1$ and $b_2 \leq B_2$,

$$R(0, b_1, b_2) = 0.$$

Recurrence. There are three possibilities when we consider the kth candidate. We can decide not to hire the candidate at all. We can hire the candidate in company 1 if $b_1 \geq s_k$. Or we can hire the candidate in company 2 if $b_2 \geq s_k$. So

$$R(k, b_1, b_2) = \max \begin{cases} R(k-1, b_1, b_2) \\ R(k-1, b_1 - s_k, b_2) + r_1(k) & \text{if } s_k \leq b_1 \\ R(k-1, b_1, b_2 - s_k) + r_2(k) & \text{if } s_k \leq b_2. \end{cases}$$

Pseudocode. We solve the subproblems in increasing order of k, b_1, and b_2, as follows.

Algorithm 3: Hiring$(s_1, \ldots, s_n, r_1, r_2, B_1, B_2)$

```
for b_1 = 0, 1, 2, ..., B_1 do
    for b_2 = 0, 1, 2, ..., B_2 do
        R[0, b_1, b_2] ← 0;

for k = 1, 2, ..., n do
    for b_1 = 0, 1, 2, ..., B_1 do
        for b_2 = 0, 1, 2, ..., B_2 do
            R[k, b_1, b_2] ← R[k − 1, b_1, b_2];
            if s_k ≤ b_1 and R[k − 1, b_1 − s_k, b_2] + r_1(k) > R[k, b_1, b_2] then
                R[k, b_1, b_2] ← R[k − 1, b_1 − s_k, b_2] + r_1(k);
            if s_k ≤ b_2 and R[k − 1, b_1, b_2 − s_k] + r_2(k) > R[k, b_1, b_2] then
                R[k, b_1, b_2] ← R[k − 1, b_1, b_2 − s_k] + r_2(k);
return R[k, B_1, B_2];
```

Proof of correctness. The base case is correct because with no candidates to hire, neither company can make any revenue.

The recurrence is correct because there are only three possibilities for our decision regarding the candidate k, and the candidate can be hired in company 1 (resp., company 2) only if the candidate's salary is no larger than the budget b_1 (resp., b_2) of the company.

Time complexity analysis. The contents of both loops have constant time complexity. The content of the initialization nested loops is run a total of $(B_1 + 1)(B_2 + 1)$ times, while the main loop is executed a total of $n(B_1 + 1)(B_2 + 1)$ times so the total time complexity of the algorithm is $\Theta(n\, B_1\, B_2)$.

4 Scriptio continua [10 marks]

Let $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{z}\}$ denote the alphabet, and let $S \subset \Sigma^*$ be the set of all words in the English language. In the ScriptioContinua problem, you are given a string $s \in \Sigma^n$ (that does not have any spaces or punctuation marks) and you must output `True` if there exists a sequence of words $w_1, \ldots, w_m \in S$ such that $s = w_1 w_2 \cdots w_m$ is the concatenation of the words $w_1, \ldots, w_m$, and `False` otherwise.

For example, for the input

$$s = \texttt{thisstringisasequenceofwordswrittenwithoutanyspaces}$$

the valid solution is `True` since we can partition s into the sequence of 11 words

$$s = \underbrace{\texttt{this}}_{w_1}\ \underbrace{\texttt{string}}_{w_2}\ \underbrace{\texttt{is}}_{w_3}\ \underbrace{\texttt{a}}_{w_4}\ \underbrace{\texttt{sequence}}_{w_5}\ \underbrace{\texttt{of}}_{w_6}\ \underbrace{\texttt{words}}_{w_7}\ \underbrace{\texttt{written}}_{w_8}\ \underbrace{\texttt{without}}_{w_9}\ \underbrace{\texttt{any}}_{w_{10}}\ \underbrace{\texttt{spaces}}_{w_{11}}$$

with $w_1, w_2, \ldots, w_{11} \in S$.

(i) Design a dynamic programming algorithm for solving the ScriptioContinua problem. In your solution, you can assume that you have access to an algorithm IsWord that, on any input $w \in \Sigma^\ell$, outputs `True` if $w \in S$ and `False` otherwise, and has time complexity $\Theta(1)$. You should aim for an algorithm with runtime $O(n^2)$.

(ii) Modify the algorithm in part (i) so that when the answer is `True`, the algorithm also outputs $w_1, \ldots, w_m \in S^m$ for which $s = w_1 w_2 \cdots w_m$.

Solution.

(i) Algorithm description. The key insight is that we get easier subproblems by considering only the prefixes of the input string.

Subproblems. For each $0 \le k \le n$, define $W(k)$ to be `True` if the string $s_1, \ldots, s_k$ can be split into words in S and `False` otherwise.

Base cases. The empty string is trivially splittable into (zero) words, so

$$W(0) = \texttt{True}.$$

Recurrence. For $1 \le k \le n$, the string $s_1, \ldots, s_k$ can be split into words if there is any $0 \le \ell < k$ such that $s_1, \ldots, s_\ell$ can be split into words and $s_{\ell+1}, \ldots, s_k$ is a word in S. So

$$W(k) = \bigvee_{0 \le \ell < k} W(\ell) \wedge \mathbf{1}[s_{\ell+1}, \ldots, s_k \in S].$$

Pseudocode. We solve the subproblems $W(1), W(2), \ldots, W(n)$ in that order to obtain the final result.

Algorithm 4: IsWords$(s_1, \ldots, s_n, S)$

```
W[0] ← True;
for k = 1, 2, ..., n do
    W[k] ← False;
    for ℓ = 0, 2, ..., k − 1 do
        if W[ℓ] and s_{ℓ+1} ··· s_k ∈ S then
            W[k] ← True;
return W[n];
```

Proof of correctness. The base case is correct because an empty string is trivially splittable into words.

For values $k \geq 1$, the recurrence is correct because if the string $s_1 \cdots s_k$ is splittable into a sequence of m words, then removing the last word $s_{\ell+1} \cdots s_k$ leaves a string $s_1 \cdots s_\ell$ that can be split into a sequence of $m - 1$ words. (Note that there may be multiple ways to choose the last word where this is true.)

Time complexity analysis. By the assumption that we can check whether a word is in S or not in constant time, we have that the contents of the inner for loop has time complexity $\Theta(1)$. The inner loop is run k times, with k being the index of the outer loop, so the total time complexity of the algorithm is

$$\sum_{k=1}^{n} k \cdot \Theta(1) = \Theta(1 + 2 + 3 + \cdots + n) = \Theta(n^2).$$

(ii) Algorithm description. We modify the algorithm from part (i) to remember the last word in the split of the string $s_1, \ldots, s_k$ into valid words when such a split exists. This is easily done by recalling the index of the first character of the last word whenever $W[k]$ is true. After computing all the values $W[0], W[1], \ldots, W[n]$, we can then use those recalled characters to extract the words one by one (from the last one to the first).

Pseudocode. The following algorithm returns the words in a valid split of the string, when one exists, in a stack. Popping the words from the stack will produce them in order.

Algorithm 5: SplitInWords$(s_1, \ldots, s_n, S)$

```
W[0] ← True;
for k = 1, 2, ..., n do
    W[k] ← False;
    for ℓ = 0, 2, ..., k − 1 do
        if W[ℓ] and s_{ℓ+1} ··· s_k ∈ S then
            W[k] ← True;
            start[k] ← ℓ + 1;
if W[n] = False return False;
WordStack ← EmptyStack();
k ← n;
while k > 0 do
    WordStack.Push(s_{start[k]}, ..., s_k);
    k ← start[k] − 1;
return WordStack;
```

Proof of correctness. The proof of correctness is essentially identical to that in part (i): the only extra observation that we make is that indeed when we set $W[k]$ to `True` we also know the index $\ell + 1$ of the first character in the last word of the valid split of the string.

Time complexity. The first part of the algorithm, with the nested for loops, still has total time complexity $\Theta(n^2)$. The second part of the algorithm, where we push the words onto the stack, has total time complexity $O(n)$ so the runtime is dominated by the first part of the algorithm and the algorithm has total time complexity $\Theta(n^2)$.

5 Shipping paper [10 marks]

Note: For this assignment only, you do not need to program your solution to the problem; only the written solution needs to be submitted.

There are two warehouses, V and W, which ship supplies of paper to n factories, denoted $F_1, \ldots, F_n$. The factory F_i requires exactly d_i units of paper, it costs v_i to ship a unit of paper from V to F_i, and it costs w_i to ship a unit of paper from W to F_i, for each $i \in \{1, 2, \ldots, n\}$. The total amount of paper available at V and W is r_V and r_W, respectively, where $r_V + r_W = d_1 + \cdots + d_n$.

We want to determine how much paper to ship from V and W to each of the factories so that the total shipping cost is minimized. Formally, let x_i denote the number of units of paper that are shipped from V to F_i and y_i denote the units of paper that are shipped from W to F_i, for $1 \leq i \leq n$. The following constraints must be satisfied:

- $x_1, \ldots, x_n, y_1, \ldots, y_n$ are non-negative integers,
- $x_1 + \cdots + x_n = r_V$,
- $y_1 + \cdots + y_n = r_W$, and
- $x_i + y_i = d_i$ for each $i \in \{1, 2, \ldots, n\}$.

The total shipping cost is

$$C = \sum_{i=1}^{n} (v_i x_i + w_i y_i).$$

An instance of the Shipping problem is specified by the values r_V, r_W, $d_1, \ldots, d_n$, $v_1, \ldots, v_n$, and $w_1, \ldots, w_n$. A valid solution to such an instance is an n-tuple $(x_1, \ldots, x_n)$ that satisfies the constraints above and minimizes C. (Since the value of x_i determines the value of y_i uniquely for each $i \in \{1, 2, \ldots, n\}$, the n-tuple $(y_1, \ldots, y_n)$ does not need to be provided explicitly in the solution.)

Design a greedy algorithm that solves the Shipping problem.

Solution.

Algorithm description. The key factor that impacts how much we would rather ship paper from V than from W to the factory F_i is not really the costs v_i and w_i by themselves, but rather the *difference* $v_i - w_i$ in how much more expensive it is to ship paper from V to F_i than to ship it from W. So the right approach to the greedy algorithm is to sort the factories by this difference so that $v_i - w_i \leq v_j - w_j$ for each $i < j$ after sorting, then to consider the factories in this order and to fill all their orders from V until no paper remains at that warehouse; the rest of the orders are filled entirely from W.

Pseudocode. The implementation of the algorithm described above is as follows.

Algorithm 6: ShippingPaper$(v_1, \ldots, v_n, w_1, \ldots, w_n, d_1, \ldots, d_n, r_V, r_W)$

```
(Assume the factories are sorted in an order F_1, ..., F_n that satisfies
v_1 − w_1 ≤ ··· ≤ v_n − w_n);
r ← r_V;
for i = 1, 2, ..., n do
    if r = 0 then
        x_i ← 0;
        y_i ← d_i;
    else if r ≥ d_i then
        x_i ← d_i;
        y_i ← 0;
        r ← r − x_i;
    else
        x_i ← r;
        y_i ← d_i − r;
        r ← 0;
return x, y;
```

If the assumption is not satisfied, we modify the algorithm to sort the factories in the desired order in the first line and to "unsort" the values x_i, y_i returned at the end.

Proof of correctness. Let $(x^*, y^*) = (x_1^*, \ldots, x_n^*, y_1^*, \ldots, y_n^*)$ be the solution returned by the greedy algorithm and let $C^* = \sum_{i \le n} v_i x_i^* + w_i y_i^*$ be its cost; let (x, y) be any solution with minimal cost C. We prove that $C^* = C$ by induction on the number k of coordinates in which $x_i^* \neq x_i$.

In the base case, when $k = 0$ then $(x^*, y^*) = (x, y)$ so $C^* = C$ as well.

For the induction step, consider any $k \ge 1$ and let (x, y) be an optimal solution that differs with (x^*, y^*) on exactly k coordinates. Let $i \le n$ be the minimal index where $x_i^* \neq x_i$. By the greedy algorithm's strategy, it must be that $x_i^* > x_i$ (otherwise this would contradict the fact that the greedy algorithm maximizes x_i^* for indices $i = 1, 2, 3, \ldots$ in this order) and since $\sum_\ell x_\ell^* = \sum_\ell x_\ell = r_V$ there must be another index $j > i$ for which $x_j^* < x_j$. Define $\delta = \min\{x_i^* - x_i, x_j - x_j^*\}$ and define the solution (x', y') by setting $x_i' = x_i + \delta$, $x_j' = x_j - \delta$, and $x_\ell' = x_\ell$ for each $\ell \notin \{i, j\}$ and fixing all $y_\ell' = d_\ell - x_\ell'$. Then the cost C' of the solution (x', y') satisfies

$$C' = \sum_{\ell \le n} v_\ell x_\ell' + w_\ell y_\ell' = C + \delta(v_i - w_i) - \delta(v_j - w_j).$$

But $v_i - w_i \le v_j - w_j$ since $i < j$, so $C' \le C$ and, since C is minimal, we in fact have $C' = C$. Furthermore, our choice of δ guarantees that $x_i' = x_i^*$ or that $x_j' = x_j^*$ so that it is a solution with optimal cost that differs from our greedy solution on at most $k - 1$ coordinates. By the induction hypothesis, we therefore have $C^* = C' = C$, as we wanted to show.

Time complexity analysis. Each iteration of the **for** loop runs in constant time, so the total time complexity of the algorithm is $\Theta(n)$. If the algorithm has to be modified to sort the factories itself, then the sorting operation dominates the runtime and the total time complexity is $\Theta(n \log n)$.