COMP 3804 — Winter 2026
Solutions Problem Set 5

Some useful facts:

1. $1 + 2 + 3 + \cdots + n = n(n+1)/2$.
2. for any real number $x > 0$, $x = 2^{\log x}$.
3. For any real number $x \neq 1$ and any integer $k \geq 1$,
$$1 + x + x^2 + \cdots + x^{k-1} = \frac{x^k - 1}{x - 1}.$$
4. For any real number $0 < \alpha < 1$,
$$\sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}.$$

Master Theorem:

1. Let $a \geq 1$, $b > 1$, $d \geq 0$, and
$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ a \cdot T(n/b) + \Theta(n^d) & \text{if } n \geq 2. \end{cases}$$
2. If $d > \log_b a$, then $T(n) = \Theta(n^d)$.
3. If $d = \log_b a$, then $T(n) = \Theta(n^d \log n)$.
4. If $d < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$.

Question 1: In class, we have seen a data structure for the UNION-FIND problem that stores each set in a linked list, with the header of the list storing the name and size of the set, and each node storing a back pointer to the header. If we start with n sets, each having size one, then we have seen in class that, using this data structure, any sequence of $n-1$ UNION-operations can be processed in $O(n \log n)$ time.

Give an example of a sequence of $n-1$ UNION-operations, for which the algorithm takes $\Omega(n \log n)$ time.

Solution: We have seen in class that the operation UNION(A, B) takes time

$$\Theta\left(\min(|A|, |B|)\right).$$

We assume for simplicity that n is a power of two, say $n = 2^k$. We start with n sets, each of size 1.

- We do $n/2$ UNION-operations, each one on two sets of size 1. Afterwards, we have $n/2$ sets, each of size 2. The total time for these $n/2$ UNION-operations is $\Theta(n)$.
- We do $n/2^2$ UNION-operations, each one on two sets of size 2. Afterwards, we have $n/2^2$ sets, each of size 2^2. The total time for these $n/2^2$ UNION-operations is $\Theta(n)$.
- We do $n/2^3$ UNION-operations, each one on two sets of size 2^2. Afterwards, we have $n/2^3$ sets, each of size 2^3. The total time for these $n/2^3$ UNION-operations is $\Theta(n)$.
- We do $n/2^4$ UNION-operations, each one on two sets of size 2^3. Afterwards, we have $n/2^4$ sets, each of size 2^4. The total time for these $n/2^4$ UNION-operations is $\Theta(n)$.
- Etc. etc.
- We do $n/2^{k-1} = 2$ UNION-operations, each one on two sets of size $2^{k-2} = n/4$. Afterwards, we have $n/2^{k-1} = 2$ sets, each of size $2^{k-1} = n/2$. The total time for these $n/2^{k-1}$ UNION-operations is $\Theta(n)$.
- We do $n/2^k = 1$ UNION-operation, on two sets of size $2^{k-1} = n/2$. Afterwards, we have $n/2^k = 1$ set of size n. The time for this UNION-operation is $\Theta(n)$.
- Overall, during all these k stages, the amount of time is

  $$\Theta(kn) = \Theta(n \log n),$$

  which is

  $$\Omega(n \log n).$$

Question 2: You are given two strings $X = x_1x_2\dots x_m$ and $Y = y_1y_2\dots y_n$ over some finite alphabet. We consider the problem of converting X to Y, using the following operations:

1. Substitution: replace one symbol by another one.
2. Insertion: insert one symbol.
3. Deletion: delete one symbol.

For example, if X = "logarithm" and Y = "algorithm", we can convert X to Y in the following way:

1. Start with "logarithm".
2. Inserting "a" at the front gives "alogarithm".
3. Deleting "o" gives "algarithm".
4. Replacing the second "a" by "o" gives "algorithm".

The *similarity* between the strings X and Y is defined to be the minimum number of operations needed to convert X to Y. For example, the similarity between X = "logarithm" and Y = "algorithm" is three, because X can be converted to Y using three operations, but not using two operations. If the string X has length m and the string Y is empty, then the similarity between X and Y is equal to m.

Give a dynamic programming algorithm (in pseudocode) that computes, in $O(mn)$ time, the similarity between the strings X and Y. Argue why your algorithm is correct. Follow the three dynamic programming steps that we have seen in class.

Solution: We want to apply dynamic programming, so we have to go through the three steps, as we did in class.

Step 1: Structure of the optimal solution.

(Note that there are several ways to do this. We will give one possible way.)

Consider the optimal way to convert $X = x_1x_2\dots x_m$ to $Y = y_1y_2\dots y_n$. There are three possibilities:

1. y_n has been inserted. Then, the optimal conversion of X to Y consists of the optimal conversion of $x_1x_2\dots x_m$ to $y_1y_2\dots y_{n-1}$, followed by the insertion of y_n.
2. x_m has been deleted. Then, the optimal conversion of X to Y consists of the optimal conversion of $x_1x_2\dots x_{m-1}$ to $y_1y_2\dots y_n$, followed by the deletion of x_m.
3. Neither y_n has been inserted, nor x_m has been deleted. There are two possibilities:
   (a) If $x_m = y_n$: In this case, the optimal conversion of X to Y is just the optimal conversion of $x_1x_2\dots x_{m-1}$ to $y_1y_2\dots y_{n-1}$.

(b) If $x_m \neq y_n$: In this case, the optimal conversion of X to Y consists of the optimal conversion of $x_1 x_2 \ldots x_{m-1}$ to $y_1 y_2 \ldots y_{n-1}$, followed by replacing x_m by y_n.

Thus, the optimal solution contains optimal solutions to subproblems.

Step 2: Set up a recurrence relation for the optimal solution.
We define, for $0 \leq i \leq m$ and $0 \leq j \leq n$:

$$S(i,j) = \text{ the similarity between } x_1 \ldots x_i \text{ and } y_1 \ldots y_j.$$

Thus, we have to compute the value of $S(m,n)$. We obtain the following recurrences:

$$S(i,0) = i \text{ for } 0 \leq i \leq m,$$

$$S(0,j) = j \text{ for } 0 \leq j \leq n,$$

$$S(i,j) = \min(S(i,j-1)+1, S(i-1,j)+1, S(i-1,j-1)) \text{ if } i > 0,\ j > 0, \text{ and } x_i = y_j,$$

$$S(i,j) = \min(S(i,j-1)+1, S(i-1,j)+1, S(i-1,j-1)+1) \text{ if } i > 0,\ j > 0, \text{ and } x_i \neq y_j.$$

Step 3: Solve the recurrence, in a bottom-up order.

```
for $i = 0$ to $m$ do $S(i,0) = i$ endfor;
for $j = 1$ to $n$ do $S(0,j) = j$ endfor;
for $i = 1$ to $m$
do for $j = 1$ to $n$
    do if $x_i = y_j$
        then $S(i,j) = \min(S(i,j-1)+1, S(i-1,j)+1, S(i-1,j-1))$
        else $S(i,j) = \min(S(i,j-1)+1, S(i-1,j)+1, S(i-1,j-1)+1)$
        endif;
    endfor;
endfor;
return $S(m,n)$
```

Question 3: Let $S = (a_1, a_2, \ldots, a_n)$ be a sequence of n positive numbers. A subsequence T of S is called *awesome*, if for every i with $1 \leq i \leq n-1$, a_i and a_{i+1} are not both in T. In other words, whenever a number is in T, none of its neighbors in S is in T. The weight of the subsequence T is the sum of all numbers in T.

Give a dynamic programming algorithm that computes, in $O(n)$ time, the maximum weight of any awesome subsequence of T.

As always, justify your answer. Follow the three dynamic programming steps that we have seen in class.

Solution: We want to apply dynamic programming, so we have to go through the three steps, as we did in class.

Step 1: Show that there is optimal substructure.
Assume we know the optimal solution for the entire problem.

- Assume that a_n is not included in the optimal solution. Then the weight of the optimal solution is the same as the weight of the optimal solution for the numbers $a_1, a_2, \ldots, a_{n-1}$.
- Assume that a_n is included in the optimal solution. Then the optimal solution does not contain a_{n-1}. The value of the optimal solution is equal to a_n plus the weight of the optimal solution for the numbers $a_1, a_2, \ldots, a_{n-2}$.
- Since we do not know which of these two cases holds, we take the larger of them.

Step 2: Set up a recurrence relation.
For $i = 0, 1, 2, \ldots, n$, let $W(i)$ be the weight of the optimal solution for the numbers $a_1, a_2, \ldots, a_i$.

We want to compute the value of $W(n)$.

- $W(0) = 0$.
- $W(1) = a_1$. (Here we use the fact that $a_1 > 0$.)
- For $i = 2, 3, \ldots, n$, $W(i) = \max(W(i-1), W(i-2) + a_i)$.

Step 3: Solve the recurrence, in a bottom-up order.

```
$W(0) = 0$;
$W(1) = a_1$;
for $i = 2, 3, \ldots, n$
do $W(i) = \max(W(i-1), W(i-2) + a_i)$
endfor;
return $W(n)$
```

It is clear that the running time is $O(n)$.

Question 4: Zoltan is not only your friendly TA, he is also the CEO of *Zoltan Enterprises*. This company buys long copper wires, cuts them into subwires, and then sells these subwires. Zoltan Enterprises only buys long copper wires having integer lengths, and cuts them such that each subwire has an integer length.

Let n be a large integer and let $p_1, p_2, \ldots, p_n$ be a sequence of positive numbers. For each i with $1 \leq i \leq n$, Zoltan Enterprises sells a subwire of length i for p_i dollars.

Consider a copper wire of length n. Give a dynamic programming algorithm that computes, in $O(n^2)$ time, the maximum revenue that can be obtained by cutting the length-n wire into subwires.

As always, justify your answer. Follow the three dynamic programming steps that we have seen in class.

For example, let $n = 4$. Here are the different options to cut a length-4 wire:

- The wire is not cut. Then the revenue is p_4.

- The wire is cut into one subwire of length 1 and one subwire of length 3. Then the revenue is $p_1 + p_3$.
- The wire is cut into two subwires of length 1 and one subwire of length 2. Then the revenue is $2 \cdot p_1 + p_2$.
- The wire is cut into two subwires of length 2. Then the revenue is $2 \cdot p_2$.
- The wire is cut into four subwires of length 1. Then the revenue is $4 \cdot p_1$.

Solution: We want to apply dynamic programming, so we have to go through the three steps, as we did in class.

Step 1: Show that there is optimal substructure.
Assume we know the optimal solution for the entire problem.

- Consider the last subwire in the optimal cutting. Let i be the length of this last subwire. Note that $1 \leq i \leq n$. Then the optimal revenue for the length-n wire is equal to p_i plus the optimal revenue for a wire of length $n - i$.
- Since we do not know the value of i, we consider all possible values for i, and take the largest revenue.

Step 2: Set up a recurrence relation.
For $m = 0, 1, 2, \dots, n$, let $R(m)$ be the optimal revenue for a wire of length m.
We want to compute the value of $R(n)$.

- $R(0) = 0$.
- For $m = 1, 2, \dots, n$,
$$R(m) = \max_{1 \leq i \leq m} (p_i + R(m - i)).$$

Step 3: Solve the recurrence, in a bottom-up order.

```
$R(0) = 0$;
**for** $m = 1, 2, \dots, n$
**do** $R(m) = -\infty$;
    **for** $i = 1, 2, \dots, m$
    **do** $R(m) = \max(R(m), p_i + R(m - i))$
    **endfor**
**endfor**;
return $R(n)$
```

The outer for-loop makes n iterations. For each such iteration, the inner for-loop makes at most n iterations. Therefore, the total running time is $O(n^2)$.