CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 13

For: April 14–April 18

Goals:

- Dynamic Programming

Problem 1. Fibonacci As Warmup

To get you warmed up, let's quickly re-cap Fibonacci yet again.

Problem 1.a. Here's some pseudocode for computing Fibonacci:

```
int fibVer1(int i) {
    if (i == 0 || i == 1) {
        return 1;
    }
    return fibVer1(i - 1) + fibVer1(i - 2);
}
```

Quick revision: What is the running time of `fibVer1`?

Solution: We've talked before in tutorial 1 about how this is $O(\phi^n)$ where ϕ is the golden ratio, but also a bound like $O(2^n)$ is perfectly acceptable in CS2040S.

Problem 1.b. Let's consider the following version instead:

```
int fibVer2Helper(int i, int[] arr) {
    if (arr[i] != -1) {
        return arr[i];
    }
    if (i == 0 || i == 1) {
        arr[i] = 1;
        return 1;
    }
```

```
        int answer = fibVer2Helper(i - 1, arr) + fibVer2Helper(i - 2, arr);
        arr[i] = answer;
        return answer;
    }

    int fibVer2(int i) {
        int[] memo_table = new int[i + 1];
        for (int idx = 0; idx <= i; ++idx){
            memo_table[idx] = -1; // mark as unsolved
        }
        return fibVer2Helper(i, memo_table);
    }
```

What is the running time of `fibVer2`? What is the space complexity of `fibVer2`?

To help you answer the question on running time, focus on the 2 following things:

1. To solve for input i, how many sub-problems are we solving?
2. How much does it cost to solve each subproblem?

Solution: Even though fibVer2Helper is recursive, notice here that we really only recurse on input i when `memo_table[i]` has not yet been set. After we recurse once, the value will have been set, and we do not recurse again.
To analyse this, we need to focus instead on how many subproblems we have. Remember, after we solve each subproblem once, we're storing it in an array and re-retrieving it in $O(1)$ time.
We have i sub-problems, and solving each requires $O(1)$ time, because we just need to add two numbers together (retrived from our memo table in $O(1)$ time each).
So $O(i) \times O(1) = O(i)$.
The space complexity is the size of the array itself, which is $O(i)$.

Problem 1.c. Try coming up with an iterative version instead, without recursion.

Solution:
```
int fibVer3(int i) {
    int[] memo_table = new int[i + 1];
    memo_table[0] = 1;
    memo_table[1] = 1;

    for (int idx = 2; idx <= i; ++idx) {
        memo_table[idx] = memo_table[idx - 1] + memo_table[idx - 2];
    }

    return memo_table[i];
}
```

As a bonus, notice this solution still uses $O(n)$ space. Can we think of a solution that takes $O(1)$ space?

Problem 2. Fancy Paintings

Xenon, who recently earned a fortune from teaching CS2040S, has decided to purchase n paintings, where the i-th painting has height h_i metres and width w_i metres. He now wishes to build a building to display his paintings. Unfortunately, he has quite limited land, and thus he needs to build multiple floors to display all the paintings.

Each floor of the building has one display wall k metres long. He can fit as many paintings side by side as long as the total width of the paintings does not exceed k metres. Furthermore, Xenon wishes to provide a curated experience, and so there are two constraints for the paintings:

- We cannot reorder the paintings. Painting i must come before painting $i + 1$.
- We cannot stack paintings. Each floor will only have one row of paintings.

The height of each floor is solely determined by the height of the tallest painting for that floor. To save on construction costs, he needs to minimize the height of the building (i.e. the sum of the height of all the floors).

Example: Given $k = 10$ and the following paintings:

- Painting 1: $h_1 = 5, w_1 = 2$
- Painting 2: $h_2 = 3, w_2 = 6$
- Painting 3: $h_3 = 4, w_3 = 4$

All three paintings can't be on the same floor, since $w_1 + w_2 + w_3 = 12 > k$. Thus, there are three possible arrangments:

- All three paintings on different floors. Then the total height is $5 + 3 + 4 = 12$.
- Paintings 1 and 2 on the first floor, and painting 3 on the second. Then total height is $5 + 4 = 9$
- Painting 1 on the first floor, and paintings 2 and 3 on the second. Then total height is $5 + 4 = 9$.

Thus, the minimal height of the building is 9.

Problem 2.a. Show that the following greedy approach to this problem does not work:

- Keep inserting paintings to the current floor.
- If the next painting does not fit, create a new floor.

Solution: Consider the following three paintings and $k = 10$:

- Painting 1, $h_1 = 1, w_1 = 5$
- Painting 2, $h_2 = 10, w_2 = 5$
- Painting 3, $h_3 = 10, w_3 = 5$

The greedy algorithm would put the first two paintings on the same floor, resulting in two floors of height 10 each. However, the optimal soultion would be to put the second and third paintings together, resulting in a total height of $h_1 + \max h_2, h_3 = 11$.

Problem 2.b. Let $dp(i)$ be the height of the building if we only consider the first i paintings. Write the recurrence relation and initial condition for $dp(i)$. If you need additional variables, please state them clearly.

Solution: We can first define $dp(0) = 0$, i.e. we need a building of zero height if we do not have any paintings.

To write a recurrence relation for $dp(i)$, we need to consider all possible arrangements for the paintings on the last floor. Since we only consider the first i paintings, the last painting on the last floor has to be the i-th painting. Thus, we just have to consider every other painting that should be on the same floor.

Let $j > 0$ be the one-based index of the painting such that $w_j + w_{j+1} + ... + w_i = \sum_{r=j}^{i} w_r \leq k$. In other words, if painting i is the last painting for the floor, the most paintings we can include is $j, j+1, ..., i-1, i$, as adding painting $j-1$ would result in the total width of the paintings exceeding k.

Say that the last floor began with painting r and ended with painting i. Then the height of the floor would be $\max_{q=r}^{i} h_q$. The total height of the previous floors would be $dp(r-1)$. Thus, the recurrence relation can then be written as

$$dp(i) = \min_{r=j}^{i} \left(dp(r-1) + \max_{q=r}^{i} h_q \right)$$

Problem 2.c. Write pseudocode for your recurrence relation. What is the worst case time and space complexity for your code?

Solution: Note that, excluding the recursive calls, we can implement $dp(i)$ to take $O(i)$ time:

```
int dp(int i) {
    if (i == 0) return 0;

    int min_building_height = MAX_INT;
    int total_width = 0;
    int max_painting_height = 0;

    for (int r = i; r >= 1; r--) {
        total_width += w[r];
        if (total_width > k) break;

        max_painting_height = max(max_painting_height, h[r]);
        min_building_height = min(min_building_height, dp(r - 1) + max_painting_height);
    }

    return min_building_height;
}
```

By using top-down DP (memoization) or bottom-up DP (table method), we can implement this in $O(n)$ space. In the worst case scenario that every painting can fit in one floor, the total time complexity would be $O(n^2)$.

Problem 3. Road Trip

Relevant Kattis Problems:

- https://open.kattis.com/problems/adventuremoving4
- https://open.kattis.com/problems/highwayhassle
- https://open.kattis.com/problems/roadtrip

You are going on a road trip. You get in your trusty car and drive to Panglossia, where you will spend a nice vacation by the beach.

You have already found the best route from your home to Panglossia (using Dijkstra's Algorithm). Next, you need to determine where you can buy petrol along the way. On the road between your home and Panglossia, there are a set of n petrol stations, the last of which is in Panglossia itself. Assume that your trip is complete when you reach the last station.

By searching on the internet, you find for each station, its location on the road and the cost of petrol. That is, the input to the problem is n stations, $s_0, s_1, \ldots, s_{n-1}$, along with $c(s_i)$, which specifies the cost of petrol at station s_i, and $d(s_i)$, which specifies the distance from your home to station s_i.

The tank of your car has a capacity of L liters. You begin your trip at home with a full tank of petrol. Your car uses exactly 1 liter per kilometer. Along the way, you must ensure that your car always has petrol (though you may arrive at a station just as you run out of petrol). Note that you do not need to fill your tank at a station. You can buy any amount of petrol, as long as it's within the capacity of your tank.

Your job is to determine **how much petrol to buy at each station** along the way so as to minimize the cost of your trip.

You may assume, for simplicity, that L and all the given distances are integers, i.e., all the quantities are integers.

Here are two examples, a simple one and a more complicated one.

Example 1: Your tank holds 6 liters, and there are 3 stations:

```
5km: $1
6km: $2
7km: $4
```

In this case, the best solution costs one dollar, where you purchase one liter of petrol at the first station at a cost of 1 per liter.

Example 2: Your tank holds 20 liters, and there are 10 stations:

```
7km:  $3
17km: $5
20km: $2
23km: $1
39km: $5
48km: $4
66km: $9
83km: $9
88km: $4
92km: $1
```

In this case, the best solution is to purchase petrol at each station as follows, for a total cost of 327 dollars:

```
0
0
3
20
5
20
15
5
4
0
```

Hint: To solve this problem, think about how to calculate $DP(s_j, k)$, the minimum cost to get from station s_j to the destination, assuming you have k liters of petrol left.

Solution: This is a typical dynamic programming style problem where the subproblems involve computing $DP(s_j, k)$ (the minimum cost to get from station j to station $n-1$) for each station s_j and each value of k. Let's use s_{-1} to denote your home.

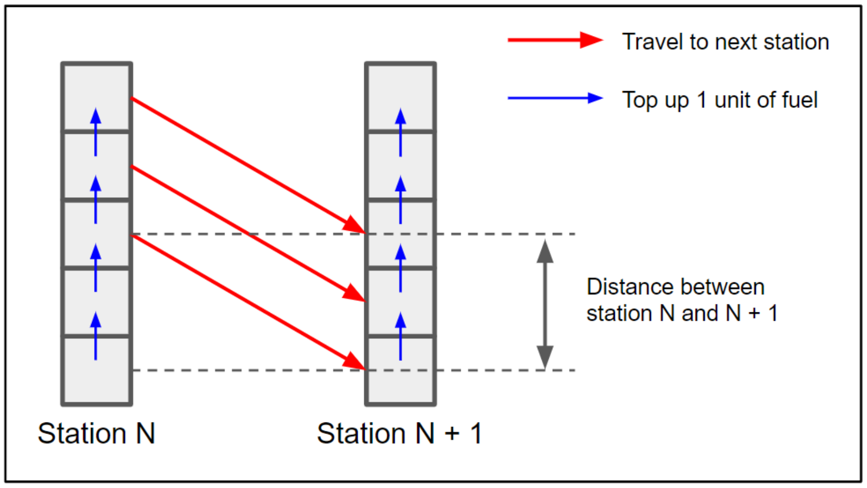
You can observe that if you have already solved the problem for all $j' > j$, then it is not hard to compute $DP(s_j, k)$. Since you have k liters and it is $h = d(s_{j+1}) - d(s_j)$ kilometers to the next station, for all $i \leq L$, lookup $DP(s_{j+1}, k - h + i)$, which is the minimum cost to go from s_{j+1} to s_{n-1} with $k - h + i$ liters of petrol, add $c(s_j) \times i$ to it, and keep track of the i that gives you the lowest cost. The value of i is the amount of petrol you should buy at station s_j. This gives the following state transfer equation:

$$DP(s_j, k) = \min_{max(0,h-k) \leq i \leq L-k} (DP(s_{j+1}, k - h + i) + c(s_j) \times i)$$

Run your DP backward from s_{n-1} until you get back to s_{-1}, you will discover the cheapest way to reach your destination.

The complexity of this solution is $O(nL^2)$, which is essentially exponential on the number of bits in L.

You may refer to `roadtrip_dp.java` for a Java implementation of this DP solution for Kattis problem Road Trip.



Alternatively, you may consider running your DP forward (from s_{-1} to s_{n-1}) and define $DP_1(s_j, k)$ to be the minimum cost to get from your home to station s_j. To reach s_j with k litres remaining, you can decide to either go from s_{j-1} with $k + h$ litres or pump 1 liter of petrol at s_j with $k - 1$ litres remaining. In other words:

$$DP'(s_j, k) = min(DP'(s_{j-1}, k - h), DP'(s_j, k - 1) + c(s_j))$$

The complexity of this solution is $O(nL)$, which still depends on L.

Solution:
Extension: In fact, there exists a greedy solution which is much more efficient than the DP solution presented above. The idea is to maintain a priority queue of petrol prices, which allows us to keep track of the price of cheapest petrol available nearby each station. As we traverse the n stations from s_0 to s_{n-1}, by greedily picking the cheapest petrol available and only making the purchase needed to reach the current location, we will end up with the cheapest total cost when we successfully reach the last station.

The complexity of this solution is $O(n \log n)$, which is no longer dependent on L.

You can further optimize this idea by using a deque instead of a priority queue, which leads to an $O(n)$ solution.

This technique is called Sweep Line, which is commonly used in competitive programming.

You may refer to `roadtrip_sl.java` for a Java implementation of this greedy solution for Kattis problem Road Trip.

Problem 4. Plagiarism Panic!

By now, you should be well aware of the Collaboration Policy on solving the Problem Sets. We hope that you've been following this policy throughout the semester!

Let's say (hypothetically!) that two students decide to cheat anyway. How might we detect that their codes are similar? In other words, given two strings of code A and B, how do we determine their similarity factor?

One way we can define the similarity factor is through shared substrings. We say the string S is a *substring* of string T if S appears as a consecutive sequence of characters within T. For example, for the string `CS2040S`, some possible substrings are `CS`, `2040`, `S204`, `CS2040S`, etc.

Two strings A and B are said to have a shared substring C if C is a substring of A and C is a substring of B. For example, `CS2040S` and `CS2030S` both share the substring `S20`.

Problem 4.a. Given two strings A and B, come up with an algorithm to determine the length of their longest shared substring. For example, the length of the longest shared substring of `dynamic` and `programming` is 2. There are two such shared substrings: `am` and `mi`.

Let $f(i, j)$ be the length of the longest shared substring that ends at the i-th character of string A and ends at the j-th character of string B.

Write the recurrence relation and base cases for $f(i, j)$. What is the answer to the original problem?

Write pseudocode for your recurrence relation. What is the time complexity of your code?

Solution: For the remainder of the solution, all positions are 1-indexed.

If $A[i] \neq B[j]$, then $f(i, j) = 0$, since no shared substring can end at $A[i]$ and $B[j]$.

If $A[i] = B[j]$, then there are 3 cases:

- $i = 1, j \geq 1$: This means the shared substring ends at the 1st character of A, so its length must be exactly 1. Thus, $f(i, j) = 1$. This is a base case.
- $i \geq 1, j = 1$: Similarly, $f(i, j) = 1$. This is also a base case.
- $i > 1, j > 1$: If the shared substring has length at least 2, then when we remove the last character of our shared substring, we now have a shorter substring that ends at the $(i-1)$-th character of A and ends at the $(j-1)$-th character of B. Thus, $f(i, j) = f(i-1, j-1) + 1$.

  Notice that this formula still works if the substring has length 1. When $f(i-1, j-1) = 0$, it means that $A[i-1] \neq B[j-1]$, so our substring cannot be extended further.

The answer is then the maximum of $f(i, j)$ over all possible choices of i and j.

Let n be the length of A and m be the length of B. Pseudocode (table method):

```
answer = 0
for each i from 1 to n:
    for each j from 1 to m:
        if A[i] != B[j]:
            f[i][j] = 0
        else if i == 1 or j == 1:
            f[i][j] = 1
        else:
            f[i][j] = f[i - 1][j - 1] + 1
        answer = max(answer, f[i][j])
```

You can also implement it recursively with memoization. Either way, the time complexity is $O(nm)$.

Problem 4.b. Another way we can define the similarity factor is through shared subsequences. We say the string S is a *subsequence* of string T if S can be obtained by removing some (or no) characters of T. For example, for the string `CS2040S`, some possible subsequences are `C4`, `SS`, `S40S`, `CS2040S`, etc.

Given two strings A and B, come up with an algorithm to determine the length of their longest shared subsequence. For example, the length of the longest shared subsequence of `dynamic` and `programming` is 3 (the string `ami`).

Similar to the previous part, let $g(i, j)$ be the length of the longest shared subsequence that ends at the i-th character of string A and ends at the j-th character of string B. What is the recurrence relation now?

Write pseudocode for your recurrence relation. What is the time complexity of your code?

Solution: The recurrence is mostly the same as before, but with a small change.

If $A[i] \neq B[j]$, then $g(i, j) = 0$.

If $A[i] = B[j]$, then there are 3 cases:

- $i = 1$ or $j = 1$: Then $g(i, j) = 1$. This is the base case.
- $i > 1, j > 1$: Since subsequences do not need to be consecutive, we don't know where the previous characters are. So we have to iterate through all possible choices. In other words, $g(i, j)$ is the maximum of $g(x, y) + 1$ over all x and y such that $x < i$, $y < j$ and $A[x] = B[y]$. Mathematically,

$$g(i, j) = \max_{1 \leq x < i, 1 \leq y < j, A[x] = B[y]} (g(x, y) + 1)$$

The answer is the maximum of $g(i, j)$ over all possible choices of i and j.

Pseudocode:

```
answer = 0
for each i from 1 to n:
    for each j from 1 to m:
        if A[i] != B[j]:
            g[i][j] = 0
        else:
            g[i][j] = 1
            for each x from 1 to i - 1:
                for each y from 1 to j - 1:
                    if A[x] == B[y]:
                        g[i][j] = max(g[i][j], g[x][y] + 1)
        answer = max(answer, g[i][j])
```

The time complexity of the code is $O(n^2 m^2)$.

Notice how we don't explicitly handle the base cases. This is because the inner loop never executes if $i = 1$ or $j = 1$.

Here's something to think about: Do we really need to check if $A[x] = B[y]$ in our inner loop?

Problem 4.c. Let's use a different approach for the DP. Define $h(i, j)$ to be the length of the longest shared subsequence that ends *before* (or at) the i-th character of string A and ends *before* (or at) the j-th character of string B.

Write the recurrence relation and pseudocode for $h(i, j)$. What is the time complexity of your code?

Note: The moral of the story is that choosing the right subproblem can make your algorithm more efficient!

Solution: For this part, positions are still 1-indexed, but we will allow i and j to be 0. Intuitively, you can think of it as having a subsequence that ends *before* the start of either string A or string B. This can only happen if the subsequence is empty, so $h(i, j) = 0$ if $i = 0$ or $j = 0$.

These are the base cases, and allowing for 0 arguments makes the implementation nicer.

To compute $h(i, j)$, there are two cases:

- If $A[i] = B[j]$, then the subsequence will end with this character. The previous character is somewhere before (or at) the $(i-1)$-th character in A and the $(j-1)$-th character in B. Thus, $h(i, j) = h(i-1, j-1) + 1$.
- If $A[i] \neq B[j]$, then we are forced to discard either the i-th character of A or the j-th character of B. Taking the maximum of these two options, we get $h(i, j) = \max(h(i-1, j), h(i, j-1))$.

The answer is now $h(n, m)$, because the shared subsequence ends somewhere before (or at) the n-th character of A and the m-th character of B.

Pseudocode:

```
for each i from 0 to n:
    h[i][0] = 0
for each j from 0 to m:
    h[0][j] = 0

for each i from 1 to n:
    for each j from 1 to m:
        if A[i] == B[j]:
            h[i][j] = h[i - 1][j - 1] + 1
        else:
            h[i][j] = max(h[i - 1][j], h[i][j - 1])

answer = h[n][m]
```

The time complexity of the code is $O(nm)$.

- This is it for CS2040S this semester. All the best! -