Week 13

Dynamic Programming!

Qn 1 - Fibonacci as Warmup

a) What is the runtime of fibVer1?

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

Qn 1 - Fibonacci as Warmup

a) What is the runtime of fibVer1?

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

As mentioned in tutorial one, the runtime is $O(\Phi^n)$ but a bound like 2^n is perfectly acceptable in CS2040S :)

Qn 1 - Fibonacci as Warmup

b) What is the runtime of fibVer2? What is the space complexity?

FibVer2

```
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);
}
```

FibVer2

```
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);
}
```

Notice that although FibVer2 is recursive, we only recurse when i has not been set. After we recurse once, value has been set and we do not need to recurse again

FibVer2

```
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);
}
```

We need to ask ourselves:

1) To solve for input i, how many sub-problems are we solving?

2) How much does it cost to solve each subproblem?

FibVer2

```
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);
}
```

Number of subproblems = i
Cost of solving each problem = O(1)
Time Complexity: O(i) x O(1) = O(i)
Space Complexity: O(i) (size of array)

Qn 1 - Fibonacci as Warmup

c) What about an iterative version?

FibVer3

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];
}
```

Number of subproblems = i
Cost of solving each problem = O(1)
Time Complexity: O(i) x O(1) = O(i)
Space Complexity: O(i) (size of array)

Qn 2 - Fancy Paintings

Xenon has purchased **n paintings,** where the **i-th painting** has height $\mathbf{h_i}$ and width $\mathbf{w_i}$ He now wishes to display his paintings in a building with multiple floors.

Each floor has one wall **k** metres long, and can fit as many paintings side-by-side so long as the total width does not exceed **k** metres.

We cannot reorder, and we cannot stack the paintings. Moreover, the height of each floor is determined by the **height of the tallest painting**, and he wants to **minimise the height of the building.**

Qn 2 - Fancy Paintings

2a) **Show that the following greedy algorithm will not work:**
Keep inserting paintings to the current floor, and if it does not fit, create a new floor.

Qn 2 - Fancy Paintings

2a) **Show that the following greedy algorithm will not work:**
Keep inserting paintings to the current floor, and if it does not fit, create a new floor.

Simple Counter-Example: Suppose k = 10
Painting 1: h = 1, w = 5
Painting 2: h = 10, w = 5
Painting 3: h = 10, w = 5

Following the above algorithm, you would put painting 1 and 2 on the first floor, and painting 3 on the second, resulting in a total height of 20m. But you can see how if we were to put painting 1 on the first, 2 and 3 on the second, it would result in a min height of 1 + 10 = 11m

Qn 2 - Fancy Paintings

2b) Let *dp(i)* be the height of the building if we only consider the first i paintings. Write the recurrence relation and the initial condition for *dp(i)*. State additional variables clearly if necessary.

Paintings

First, we define **dp(0) = 0.** We need a building of 0 height of we do not have any paintings.

Next, to write a recurrence for *dp(i)*, we need to consider **all possible arrangements on the LAST floor**, since the **last painting on the last floor has to be the i-th painting**. Thus, we need 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.

Paintings

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.

This just means: We can choose the number of paintings that we can add onto the last floor with the last painting, but the maximum is reached when adding one more would exceed width k (and recall we have to go in sequence! based on restrictions given in the question)

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

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

Paintings

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

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

Finding the starting index that minimises the building height

Total height of previous floors (already minimised)

Height of the tallest painting

Qn 2 - Fancy Paintings

2c) Write pseudocode for the recurrence, and identify worst-case time and space complexity

Paintings

```
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;
}
```

Excluding the recursive calls, finding dp(i) can take O(i) time.

Paintings

```
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;
}
```

Excluding the recursive calls, finding dp(i) can take O(i) time.

Note that this pseudocode is only for showcasing the recurrence! There is NO dynamic programming implemented YET, but you can see how you can either use memoisation (top-down) or table approach (bottom-up) to avoid recomputing repeated values of dp(r - 1)

Paintings

```
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;
}
```

Space complexity: O(n)

Time complexity: $O(n^2)$, when every painting can fit on the same floor (why? because you need to test more values of r (starting index))

Qn 3 - Road Trip

Set of **n petrol stations** (including destination). Trip is complete when reached destination. Each petrol station has a **cost**, and a **distance.**

Tank has **L litres**. **Each km uses 1 litre.** Ensure that your car always has petrol (but you can reach a station with 0 petrol). Determine how much petrol to buy at each station to **minimise the cost of the trip**

First step: Identify what is the subproblem!

Qn 3 - Road Trip

Sub-problem: Compute **DP(s_j,k)** i.e. the minimum cost to get from station j to station n-1 (destination) **for each station and for each value of k**.

Note that if you have already solved the problem for ALL j' > j, then it is not hard to compute DP(s_j, k) because, suppose you have k liters left, and you know the distance (h) to the next station, then you just have to **lookup DP(s_{j+1}, h - k + i) for all i ≤ L,** add **c(s_j) x i** to it (which simulates buying petrol at the station) and select the **minimum cost.** Then the value of i is the amount of petrol you should buy at that station.

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

Run your DP backwards from destination to source to discover the cheapest way.

Qn 3 - Road Trip

Runtime: **$O(nL^2)$**, where n is the number of stations and L is the number of litres.

How is this runtime derived? At each station **(n)**, for each possible fuel level **(L)**, try buying various levels of petrol **(L)** and find the minimum.

Qn 3 - Road Trip

Runtime: **$O(nL^2)$**, where n is the number of stations and L is the number of litres.

How is this runtime derived? At each station **(n)**, for each possible fuel level **(L)**, try buying various levels of petrol **(L)** and find the minimum.

But, can we do better? Hint: Run your DP Forward!

Qn 3 - Road Trip

But, can we do better? Hint: Run your DP Forward!

Alternatively, define DP(s_j, k) to be minimum cost to get from home to station s_j. To reach s_j with k litres remaining, can decide to go from s_{j-1} with k + h litres or pump 1 litre 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.

Why is it O(nL)? For each station **(n)**, compute minimum cost for each possible amount of fuel k **(L).**

Qn 4 - Plagiarism

4a) Given two substrings A and B, come up with an algorithm to determine the length of the longest shared substring.

Let **f(i,j)** be the length of the longest shared substring that ends at the i-th character of string A and 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 and identify time complexity of your code.

Plagiarism

Let f(i,j) be the length of the longest shared substring that ends at the i-th character of string A and 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 and identify time complexity of your code.

If A[i] != B[j], then f(i, j) = 0 since there is no way a shared substring can end at A[i] and B[j].

If A[i] = B[j], there are three cases:

1. i = 1, j ≥ 1, means f(i, j) = 1. This is a base case
2. i ≥ 1, j = 1, means f(i, j) = 1. This is another base case.
3. i > 1, j > 1. If the shared substring has length at least 2, when we remove the last character, we have a shorter substring that ends at the (i - 1)th character of A and (j - 1)th character of B. Thus, **f(i, j) = f(i - 1, j - 1) + 1 (recurrence!)**

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

Plagiarism

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 $O(nm)$.

Qn 4 - Plagiarism

4b) Given two substrings A and B, come up with an algorithm to determine the length of the longest shared subsequence. (Subsequence = can remove some characters in between)

Let **g(i,j)** be the length of the longest shared subsequence that ends at the i-th character of string A and 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 and identify time complexity of your code.

Plagiarism

Let g(i,j) be the length of the longest shared subsequence that ends at the i-th character of string A and the j-th character of string B.
Write the recurrence relation and base cases for g(i, j). What is the answer to the original problem?
Write pseudocode and identify time complexity of your code.

Similar to before,
If A[i] != B[j], then g(i, j) = 0

If A[i] = B[j], there are three cases:

1. i = 1, j ≥ 1, means f(i, j) = 1. This is a base case
2. i ≥ 1, j = 1, means f(i, j) = 1. This is another base case.
3. i > 1, j > 1. Since subsequences **do not need to be consecutive,** we don't know where previous characters are, so we need to consider 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].**

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

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

Plagiarism

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 $O(n^2m^2)$.

Qn 4 - Plagiarism

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

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

Plagiarism

Define h(i,j) to be the length of the longest shared subsequence that **ends before (or at) the i-th character of A** and **ends before(or at) the j-th character of B.**
Write the recurrence relation and pseudocode for h(i, j). What's the time complexity of the code?

In this case, we will allow i and j to be equal to 0 (you can think of it as having a subsequence before the start of string A or B). This can only happen if the subsequence is empty, so h(i, j) == 0 if i == 0 or j == 0. These are our base cases.

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

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

Plagiarism

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 $O(nm)$.