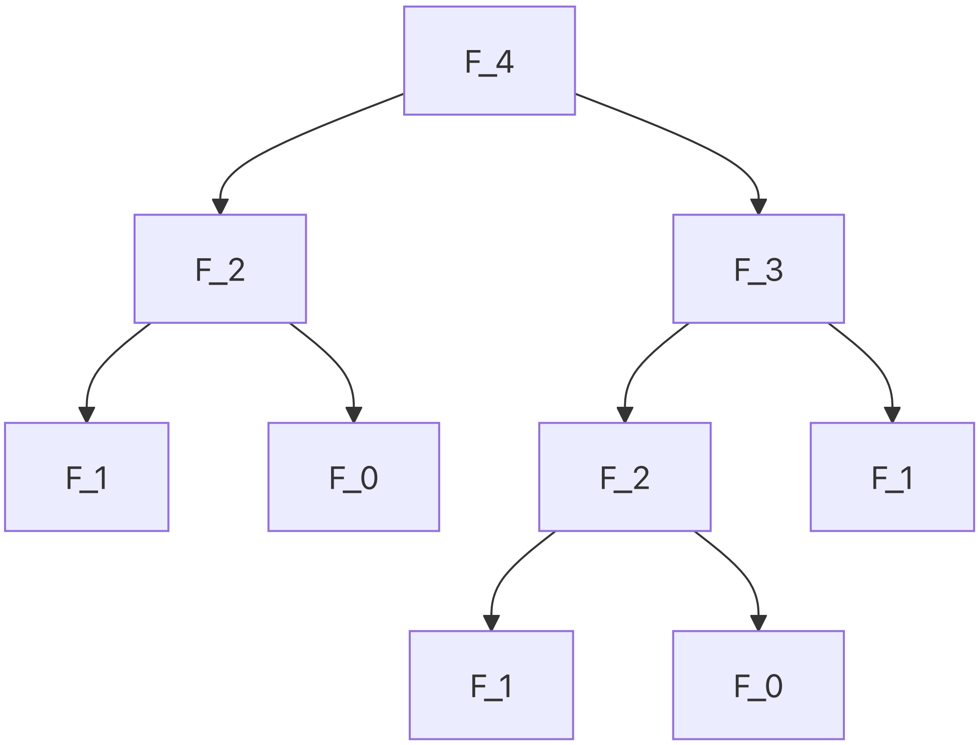
Chapter 5 - Dynamic Programming

Dynamic Programming

> *A technique that use a lookup table to store result of each sub-problem and immediately use it if any sub-problem is required multiple times.*



(Sub-problems overlap, and we can use a lookup table to memorize results)

> *Dynamic Programing = Divide and Conquer + Lookup Table.*

We can solve dynamic programming using 2 approaches:

- Top-Down Approach (Memoization): Start from whole problem and then break it down into sub-problems.
- Bottom-Up Approach: Start from sub-problems and build up to solve the whole problem.

Fibonacci Number

> *The Fibonacci number is defined by $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$, with the base cases $F(0) = 0, F(1) = 1$*

$$F(n) = \begin{cases} 0 & ; n = 0 \\ 1 & ; n = 1 \\ F(n-1) + F(n-2) & ; n \geq 2 \end{cases}$$

Top-Down Approach

```
int fibonacci(int n) {
    if (n == 0 || n == 1) return n;
    if (dp[n]) return dp[n]; // We can use dp[n] to check whether dp[n] 
has been calculated because fibonacci(n) for n >= 2 cannot be zero.
    return dp[n] = fibonacci(n-1) + fibonacci(n-2);
}
```

Bottom-Up Approach

```
dp[0] = 0;
dp[1] = 1;
for (int i = 2; i <= n; i++) {
    dp[i] = dp[i-1] + dp[i-2];
}
```

Time Complexity: $O(n)$

Binomial Coefficient

> *To calculate the number of ways to choose r items from n items. We have closed form solution $C(n, r) = \frac{n!}{r!(n-r)!}$*

Using the following property:

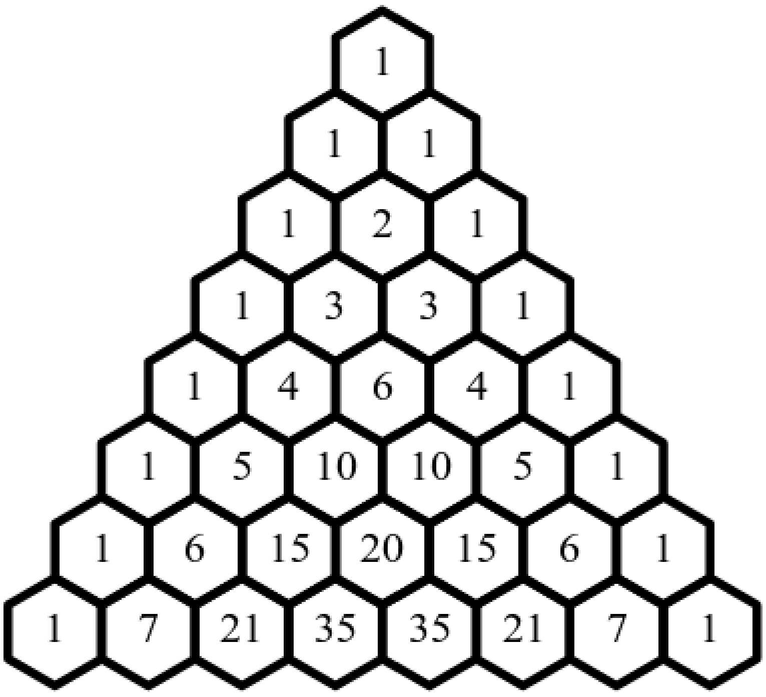
$$C(n, r) = \begin{cases} 1 & r = 0 \text{ or } n = r \\ C(n-1, r) + C(n-1, r-1) & \text{otherwise} \end{cases}$$

We can calculate $C(n, r)$ recursively.

Pascal's Triangle

> *A pascal's triangle is an arrangement of numbers in a triangular array such that the numbers at the end of each row are 1 and the remaining numbers are the sum of the nearest two numbers in the above row.*

The number in Pascal's triangle at the j^{th} row and i^{th} column can be calculated using $\binom{i}{j}$ for $i \geq 0$ and $j \geq 0$



Top-Down Approach

```
int binomial(int n, int r) {
    if (r == 0 || n == r) return 1;
    if (dp[n][r]) return dp[n][r]; // We can use dp[n][r] to check whether dp[n][r] has been calculated because binomial(n, r) cannot be zero.
    return dp[n][r] = binomial(n - 1, r) + binomial(n - 1, r - 1);
}
```

Bottom-Up Approach

```
for (int i = 0; i <= n; i++) {
    for (int j = 0; j<=i; j++) {
        if (j == 0 || i == j) dp[i][j] = 1;
        else dp[i][j] = dp[i-1][j] + dp[i-1][j-1];
    }
}
```

Time Complexity: $O(nr)$

Kadane's Algorithm

> *The same ideal as Maximum Subarray sum in Divide and Conquer, but faster. It reduces the time complexity from $O(n\ log\ n)$ to $O(n)$.*

The original problem is proposed by Ulf Grenander.

- $O(n\ log\ n)$ solution is proposed by Micheal Shamos.

- $O(n)$ solution is proposed by Joseph Born Kadane.

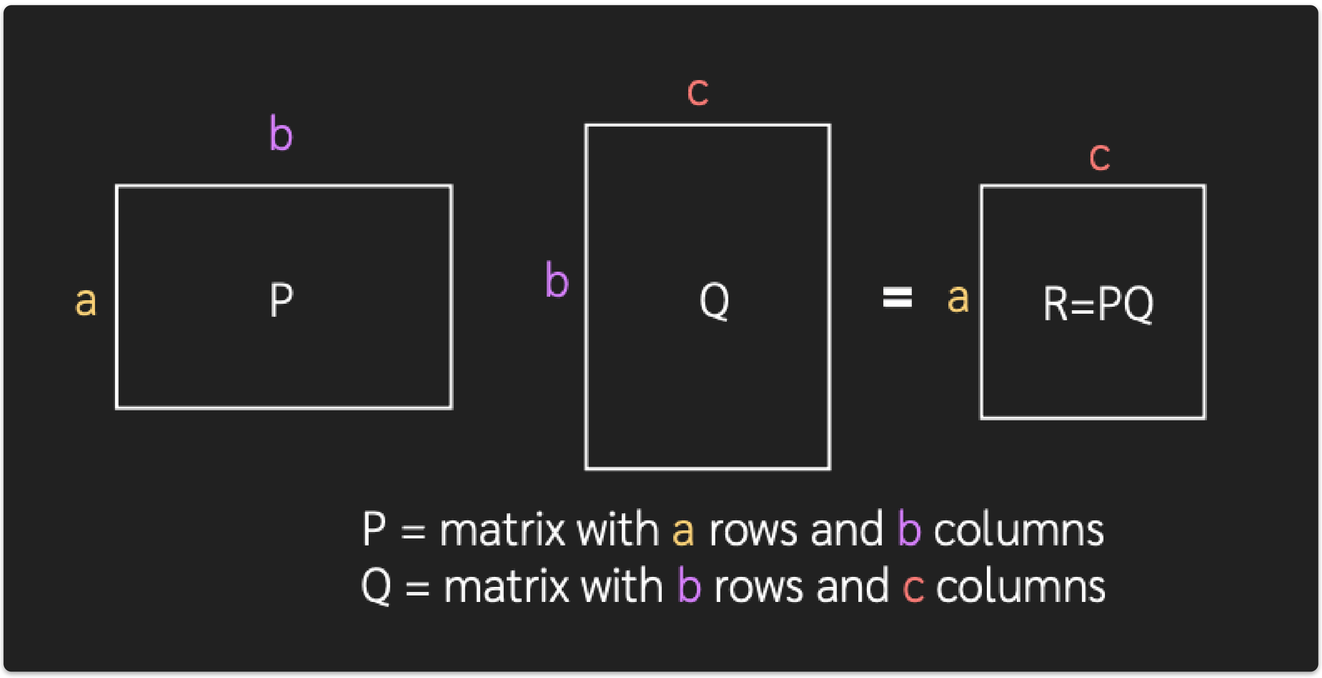
Kadane's Algorithm

```
int kadane(int arr[]) {
    int sum = arr[1];
    int ans = arr[1];
    for (int i = 2; i <= n; i++) {
        sum = max(arr[i], sum+arr[i]);
        ans = max(ans, sum);
    }
    return ans;
}
```

Time Complexity: $O(n)$

Matrix Chain Multiplication

> *Find an efficient way to multiply matrices with the fewest number of multiplications.*


b
a
P
c
b
Q
=
c
a
R=PQ
P = matrix with a rows and b columns
Q = matrix with b rows and c columns

The time complexity to multiply matrix P and Q is $\Theta(abc)$.

Due to the associative property of matrix multiplication, $A(BC) = (AB)C$, the order of multiplication matters. We need to find a way to compute the fewest matrix multiplications.

Example

- $(B_1B_2)((B_3B_4)B_5)$ is equivalent to $(B_1B_2)(B_3(B_4B_5))$
- $((B_1B_2)(B_3B_4))B_5$ is equivalent to $(((B_1B_2)B_3)B_4)B_5$

  i We can count those cases as the same sub-problem.

Top-Down Approach

$$MCM(l,r) = \begin{cases} 0 & ; l = r \\ min(MCM(l,i) + MCM(i+1,r) + A[l-1] * A[i] * A[r]) & ; l < r \text{ and } l \le i < r \end{cases}$$

```
int mcm(int l, int r) {
    if (l == r) return 0;
    if (chosen[l][r]) return dp[l][r];

    int ans = 1e9;
    for (int i = l; i < r; i++) {
        int res = solve(l, i) + solve(i+1, r) + arr[l-1]*arr[i]*arr[r];
        ans = min(ans, res);
    }

    chosen[l][r] = true;
    return dp[l][r] = ans;
}
```

Bottom-Up Approach

```
for (int k = 2; k <= n; k++) {
    for (int i = 1; i <= n - k + 1; i++) {
        dp[i][i+k-1] = 1e9;
        for (int j = i; j <= i+k-1; j++) {
            int val = dp[i][j] + dp[j+1][i+k-1] + arr[i-1]*arr[j]*arr[i+k-
1];
            if (dp[i][i+k-1] > val) {
                dp[i][i+k-1] = val;
                from[i][i+k-1] = j;
            }
        }
    }
}

cout << dp[2][n]; // dp[2][n] is the answer
```

Retrieve the Way the Algorithm Splits the Problem

```
string retrieve_mcm(int l, int r) {
    if (l == r) {
        return "A" + to_string(l-1);
    }

    int retrieve_index = from[l][r];
    return "(" + retrieve_mcm(l, retrieve_index) + " " +
retrieve_mcm(retrieve_index+1, r) + ")";
}
```

Time Complexity: $O(n^3)$ where n is the number of matrices.

Knapsack Problem

> *Given a sack, able to hold W kg and a list of objects each has a value v_i and a weight w_i. Try to pack the object in the sack so that the total value is maximized.*

Variation

- Rational Knapsack: We can cut the object into pieces, each has the same value/weight ratio
- 0-1 Knapsack: We cannot cut the object, we can only choose (1) or leave (2) the object

Top-Down Approach

$$K(i, W) = \begin{cases} -\infty & ; W < 0 \\ 0 & ; i > n \\ K(i+1, W) + W(i+1, W - weight[i]) + value[i] & \text{otherwise} \end{cases}$$

```
int knapsack(int index, int W) {
    if (W < 0) return -1e9;
    if (index > n) return 0;
    if (chosen[index][W]) return dp[index][W];

    int r1 = knapsack(index+1, W);
    int r2 = knapsack(index+1, W-items[index].weight) +
items[index].value;

    chosen[index][W] = true;
    return dp[index][W] = max(r1, r2);
}
```

Bottom-Up Approach

```
for (int i = 1; i <= n; i++) {
    for (int j = 0; j <= W; j++) {
        if (j >= weight[i]) {
            dp[i][j] = max(dp[i-1][j], dp[i-1][j-weight[i]] + value[i]);
        } else {
            dp[i][j] = dp[i-1][j];
        }
    }
}
```

```
cout << dp[n][W];
```

Time Complexity: $O(Wn)$

Coin Change

> *We want to pay a change of W baht by the set of coins using smallest number of coins.*

Top-Down Approach

```
int coin_change(int x, int coins[]) {
    if (x < 0) return 1e9;
    if (x == 0) return 0;
    if (dp[x]) return dp[x];

    int ans = 1e9;
    for (int i = 1; i <= n; i++) {
        ans = min(ans, coin_change(x-coins[i], coins) + 1);
    }

    return dp[x] = ans;
}
```

Bottom-Up Approach

```
// Need to set all indices in array to infinity except for index 0
dp[0] = 0;

for(int i=0 ; i<=x ; i++){
    for(int j=1 ; j<=n ; j++){
        if(i+arr[j]<=x){
            dp[i+arr[j]] = min(dp[i+arr[j]], dp[i]+1);
        }
    }
}

cout << dp[x];
```

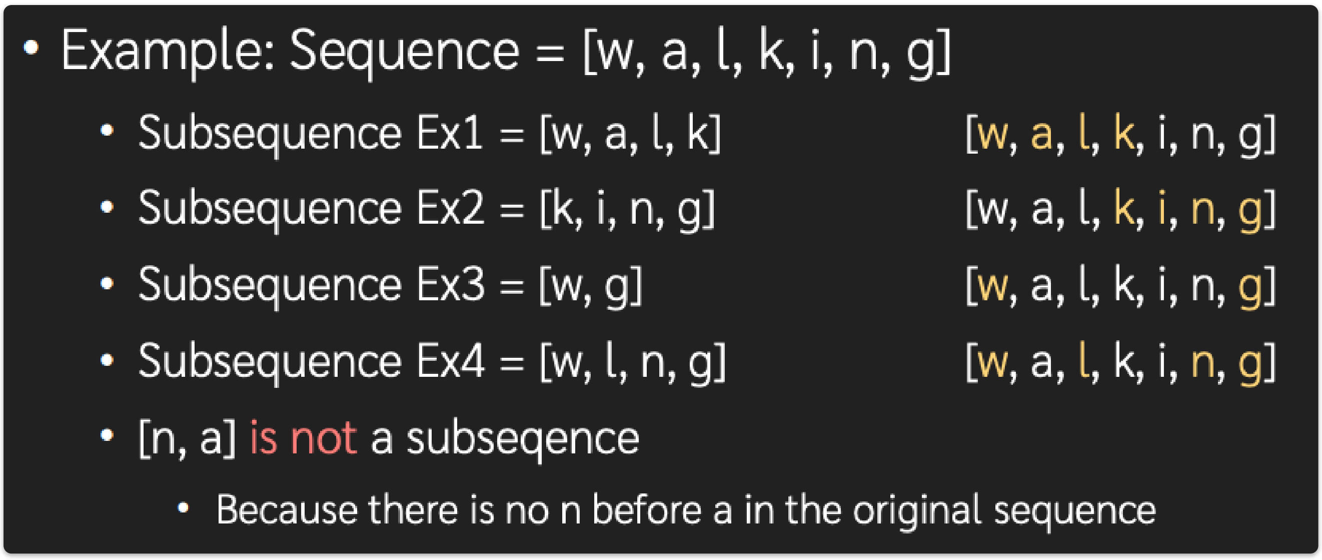
Time Complexity: $O(xn)$

Longest Common Subsequence

> *To find the longest common subsequence of both A and B*

Subsequence

> *An ordered combination of each member of the sequence*



- Example: Sequence = [w, a, l, k, i, n, g]
  - Subsequence Ex1 = [w, a, l, k] [w, a, l, k, i, n, g]
  - Subsequence Ex2 = [k, i, n, g] [w, a, l, k, i, n, g]
  - Subsequence Ex3 = [w, g] [w, a, l, k, i, n, g]
  - Subsequence Ex4 = [w, l, n, g] [w, a, l, k, i, n, g]
  - [n, a] is not a subseqence
    - Because there is no n before a in the original sequence

Example

A = [w, a, l, k, i, n, g]
B = [a, l, i, e, n]

LCS(A, B) = [a, l, i, n]

Top-Down Approach

```
int lcs(int n, int m, string s1, string s2) {
    if (n < 0 || m < 0) return 0;
    if (chosen[n][m]) return dp[n][m];
    if (s1[n] == s2[m]) return lcs(n-1, m-1, s1, s2) + 1;

    chosen[n][m] = true;
    return dp[n][m] = max(lcs(n-1, m, s1, s2), lcs(n, m-1, s1, s2));
}
```

Bottom-Up Approach

```
for (int i = 0; i <= n; i++) {
    for (int j = 0; j <= m; j++) {
        if (i == 0 || j == 0) dp[i][j] = 0;
        else if (s1[i-1] == s2[j-1]) dp[i][j] = dp[i-1][j-1] + 1;
        else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    }
}
```

Time Complexity: $O(nm)$