Dynamic Programming

What is Dynamic Programming?

A very powerful and general tool for solving certain optimization problems

Dynamic Programming VS Recursion

- Both are using sub-problems to solve the main problem
- Recursion is much more general
- If it can be solved by dynamic programming, DP is much faster
- Usually DP requires more space
- SPACE VS TIME

When we use DP?

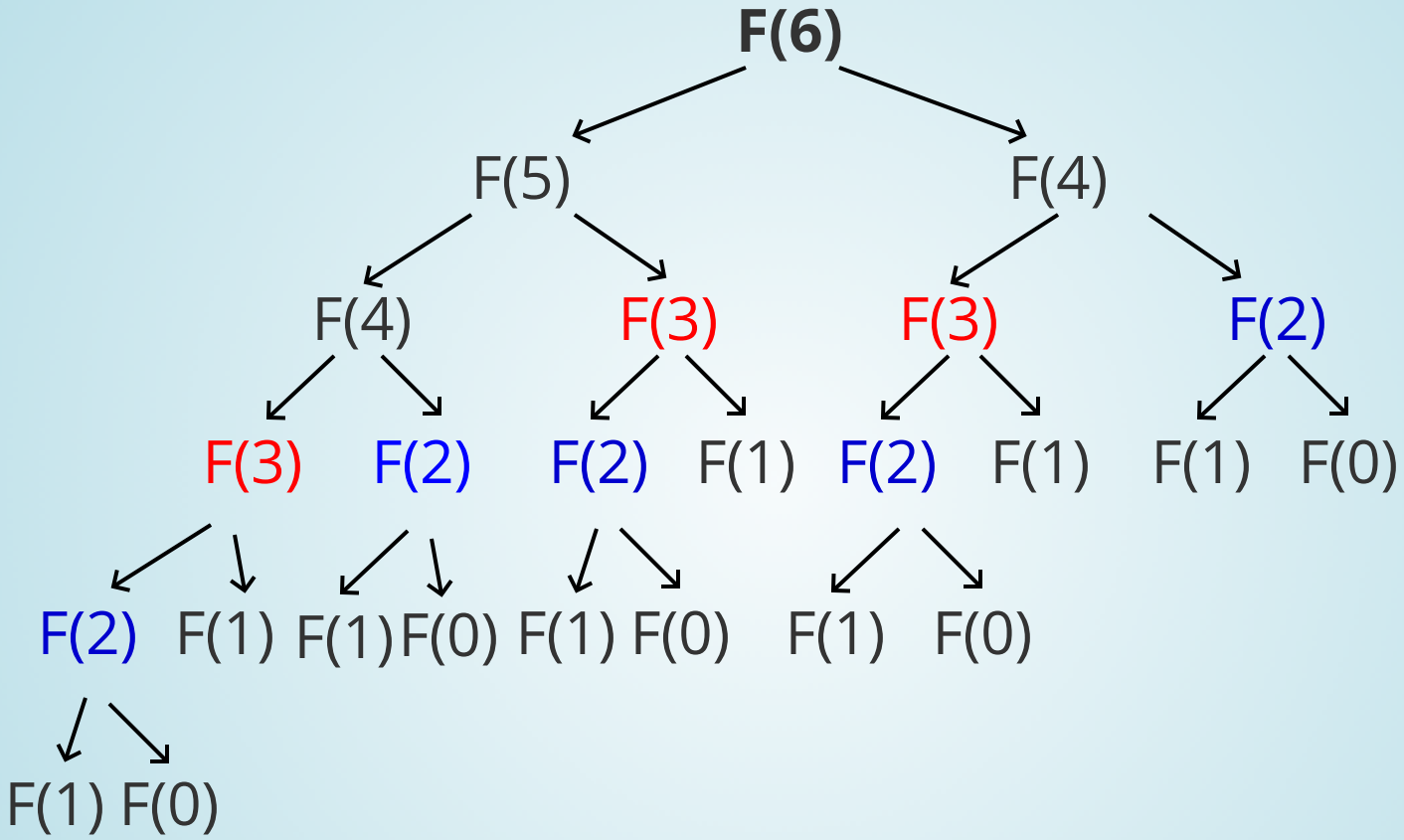
- It can be resolved by recursion and not tail recursion
- Recursion is making some redundant computation
- It has to have optimal sub-structures:
  optimal solutions to the original problem contains
  optimal solutions to sub-problems

How to use DP?

- Think about the recursion solution
- Find sub problems that have been computed multiple times
- See if the sub problems are optimal sub-structure
- Try to memorize them to avoid the redundant computing

Fibonacci Numbers

```
int Fabonacci(int n) {
    if(n == 0 || n == 1) return 1;
    return F(n-1) + F(n-2);
}
```



Need some space to avoid the computation

```
int Fabonacci(int n) {
    if(n <= 1) return 1;
    int[] result = new int[n + 1];
    result[0] = 1;
    result[1] = 1;
    for(int i = 2; i < n + 1; i ++) {
        result[i] = result[i-1] + result[i-2];
    }
    return result[n];
}
```

We use an Array to store the temp result

What is the optimal sub-structure?

Unique Paths

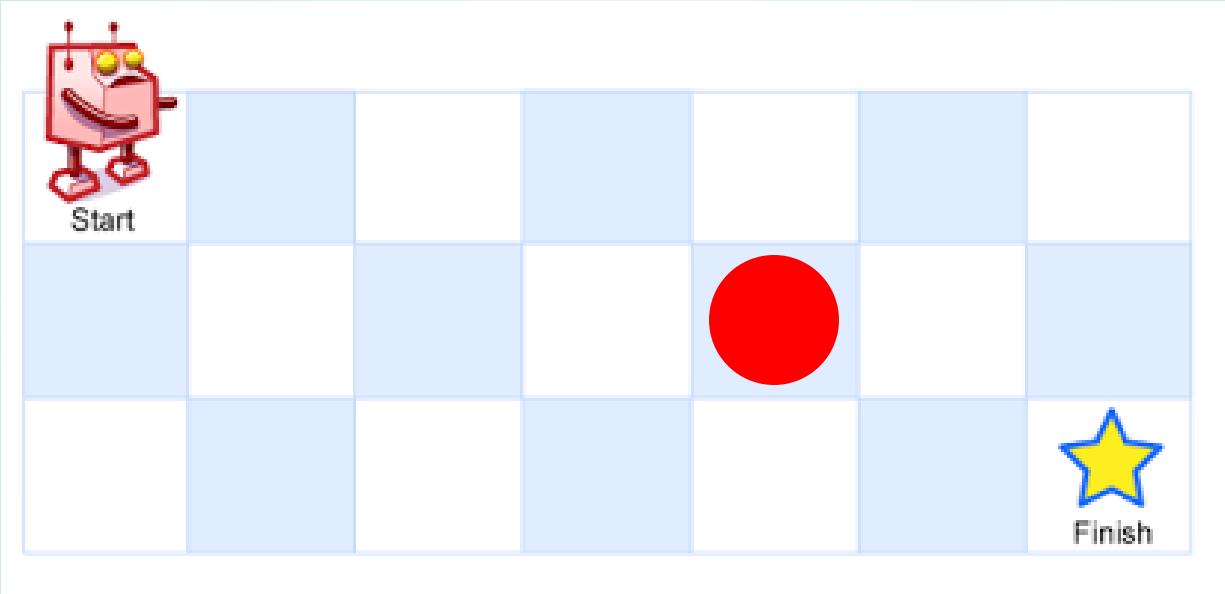
Unique Paths

```
int uniquePaths(int m, int n) {
      if(m==1 || n==1) return 1;
      return uniquePaths(m-1, n) + uniquePaths(m, n-1);
}
```

Where is the redundancy?

What is the optimal sub-structure?

Unique Paths



Optimal Sub-structure

$$a(i,j) = a(i-1,j) + a(i, j-1)$$

Unique Paths

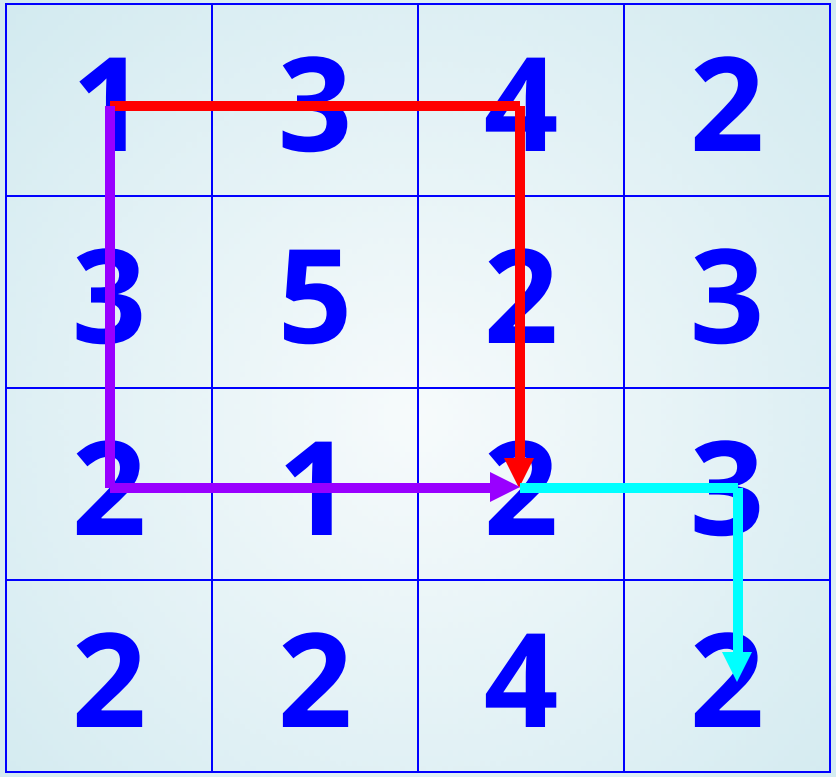
```
public int uniquePaths(int m, int n) {
    int[][] a = new int[m][n];
    for (int i = 0; i < m; i++) {
        a[i][0] = 1;
    }
    for (int i = 0; i < n; i++) {
        a[0][i] = 1;
    }
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            a[i][j] = a[i-1][j] + a[i][j-1];
        }
    }
    return a[m-1][n-1];
}
```

Minimum Path Sum

1	3	4	2
3	5	2	3
2	1	2	3
2	2	4	2

Best optimal sub-structure?

Minimum Path Sum



PathSum(m,n) = MIN(PathSum(m,n-1),PathSum(m-1,n)) + matrix(m,n)

Minimum Path Sum

```
public int minPathSum(int[][] grid) {
    if(grid == null || grid.length==0)
        return 0;
    int m = grid.length;
    int n = grid[0].length;
    int[][] dp = new int[m][n];
    dp[0][0] = grid[0][0];
    for(int i=1; i<n; i++){
        dp[0][i] = dp[0][i-1] + grid[0][i];
    }
    for(int j=1; j<m; j++){
        dp[j][0] = dp[j-1][0] + grid[j][0];
    }
    for(int i=1; i<m; i++){
        for(int j=1; j<n; j++){
            if(dp[i-1][j] > dp[i][j-1]){
                dp[i][j] = dp[i][j-1] + grid[i][j];
            }else{
                dp[i][j] = dp[i-1][j] + grid[i][j];
            }
        }
    }
    return dp[m-1][n-1];
}
```

Minimum Path Sum

Any improvement?

We can use less space to get the same result without hurting time complexity

Minimum Path Sum II

```
public static int minPathSum(int[][] grid) {
    if(grid == null || grid.length==0)
        return 0;
    int m = grid.length;
    int n = grid[0].length;
     int[] newline = new int[n];
     int[] oldline = new int[n];
     oldline[0] = grid[0][0];
     for(int i=1; i<n; i++){
         oldline[i] = oldline[i-1] + grid[0][i];
     }
     for(int i=1; i<m; i++){
         newline[0] = grid[i][0] + oldline[0];
         for(int j=1; j<n; j++){
             if(oldline[j] > newline[j-1]){
                 newline[j] = newline[j-1] + grid[i][j];
             }else{
                 newline[j] = oldline[j] + grid[i][j];
             }
         }
         oldline = newline;
     }
     return newline[n-1];
}
```

Minimum Path Sum II

We can even only use one array

0-1 Knapsack

Given a knapsack which can hold ***w*** pounds of items, and a set of items with weight ***w1, w2, ... wn***. Each item has its value **s1,s2,...,sn**. Try to select the items that could put in knapsack and contains most value.

What is the optimal sub-structure?

0-1 Knapsack

w[i][j]: for the previous total i items, the max value it can have for capacity j

Which two we need to use to compare?

w[i][j]: for the previous total i items, the max value it can have for capacity j

When you iterate i, and j, you need to try:

1. If the new i could be added into j
2. if it could and added, could it be better

w[i][j]: for the previous total i items, the max value it can have for capacity j

Example: weights{1,3,4,5} values{3,8,4,7}

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	0	0	0	0	0	0	0
1	0	3	3	3	3	3	3	3
2	0	3	3	8	11	11	11	11
3	0	3	3	8	11	11	11	12
4	0	3	3	8	11	11	11	12

```java
public int knapsack(int capacity, int[] weights, int[] values) {
        int length = weights.length;
        if (capacity == 0 || length == 0)
                return 0;
        int[][] w = new int[length + 1][capacity + 1];
        for (int i = 1; i <= length; i++) {
                int index = i - 1;
                for (int j = 1; j <= capacity; j++) {
                        if (j < weights[index]) {
                                w[i][j] = w[i - 1][j];
                        } else if (w[i - 1][j - weights[index]] + values[index] > w[i - 1][j]) {
                                w[i][j] = w[i - 1][j - weights[index]] + values[index];
                        } else {
                                w[i][j] = w[i - 1][j];
                        }
                }
        }
        return w[length][capacity];
}
```

Coin Change

We mentioned this question in DFS and use DFS will be TLE

The question is similar to knapsack

coin and amount will be the two dimension

How we update the result?

```java
public int coinChange(int[] coins, int amount) {
    Arrays.sort(coins);
    int length = coins.length;
    int[][] dp = new int[length][amount + 1];
    for (int j = 0; j <= amount; j ++) {
        if (j % coins[0] == 0) {
            dp[0][j] = j / coins[0];
        } else {
            dp[0][j] = -1;
        }
    }
    for (int i = 1; i < length; i ++) {
        for (int j = 0; j <= amount; j ++) {
            if (j < coins[i]) {
                dp[i][j] = dp[i-1][j];
            } else {
                int temp = Integer.MAX_VALUE;
                for (int k = 0; k <= j / coins[i]; k ++) {
                    int remaining = j - coins[i] * k;
                    if (dp[i-1][remaining] != -1 && dp[i-1][remaining] + k < temp) {
                        temp = dp[i-1][remaining] + k;
                    }
                }
                dp[i][j] = temp < Integer.MAX_VALUE ? temp : -1;
            }
        }
    }
    return dp[length - 1][amount];
}
```

Longest Increasing Subsequence

3, 1, 4, 5, 7, 6, 8, 2

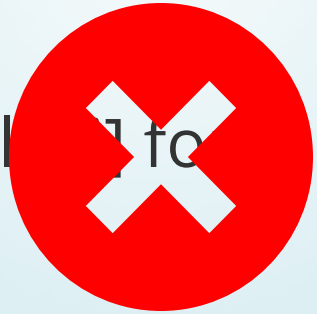
1, 4, 5, 6, 8 (Or 1, 4, 5, 7, 8)

Longest Increasing Subsequence

What is the optimal sub-structure?

Longest Increasing Subsequence

We store l[i] for the LIS by i?



We store lis[i] for the LIS using sequence[i]

Longest Increasing Subsequence

```
public int longestIncreasingSubsequence(int[] nums) {
    if(nums.length == 0){
        return 0;
    }
    int[] lis = new int[nums.length];
    int max = 0;
    for (int i = 0; i < nums.length; i++){
        int localMax = 0;
        for (int j = 0; j < i; j++){
            if (lis[j] > localMax && nums[j] <= nums[i]){
                localMax = lis[j];
            }
        }
        lis[i] = localMax + 1;
        max = Math.max(max, lis[i]);
    }
    return max;
}
```

What is the Space Complexity?

What is the Time Complexity?

Can we do better?

Patient Sort

1, 3, 5, 2, 8, 4, 7, 6, 0, 9, 10

1 -> 0
1,3 -> 1,2
1,3,5 -> 1,3,4
1,3,5,8 -> 1,3,5,7 -> 1,3,5,6
1,3,5,6,9
1,3,5,6,9,10

```java
public int longestIncreasingSubsequence(int[] nums) {
    if(nums.length == 0){
        return 0;
    }
    int len = 0;
    int[] tails = new int[nums.length];
    tails[0] = nums[0];
    for(int i = 1; i < nums.length; i++){
        if(nums[i] < tails[0]){
            tails[0] = nums[i];
        } else if (nums[i] >= tails[len]){
            tails[++len] = nums[i];
        } else {
            tails[binarySearch(tails, 0, len, nums[i])] = nums[i];
        }
    }
    return len + 1;
}
private int binarySearch(int[] tails, int min, int max, int target){
    while(min < max){
        int mid = min + (max - min) / 2;
        if(tails[mid] == target){
            return mid;
        }
        else if(tails[mid] < target){
            min = mid + 1;
        }
        else max = mid;
    }
    return min;
}
```

Longest Common Sequence

Longest Common Sequence

Example: abcfbc abfcab

return 4 (abcb)

Longest Common Sequence

Example: abcfbc abfcab

return 4 (abcb)

Longest Common Sequence

What is the optimal sub-structure?

maxCommon(i,j): longest common string
for String A(0,i) and String B(0,j)

We finally need to get
maxCommon(stringA.length, stringB.length)

Longest Common Sequence

What is the relationship between maxCommon(i,j) and maxCommon(i-1,j-1)?

If(A[i-1] = B[j-1]) ?

If(A[i-1] != B[j-1])?

Longest Common Sequence

What is the relationship between
maxCommon(i,j) and maxCommon(i-1,j-1)?

If(A[i-1] = B[j-1]) ?

maxCommon(i,j) = maxCommon(i-1,j-1) + 1

If(A[i-1] != B[j-1])?

maxCommon(i,j) = max(maxCommon(i-1,j), maxCommon(i,j-1))

Longest Common Sequence

```
public static int longestCommonString(String a, String b) {
        int m = a.length();
        int n = b.length();
        int[][] maxCommon = new int[m+1][n+1];
        for(int i = 0; i <= m; i ++) {
                maxCommon[i][0] = 0;
        }
        for(int j = 0; j <= n; j ++) {
                maxCommon[0][j] = 0;
        }
        for(int i = 1; i <= m; i ++) {
                for(int j = 1; j <= n; j ++) {
                  if(a.charAt(i-1) == b.charAt(j-1)) {
                    maxCommon[i][j] = maxCommon[i-1][j-1] + 1;
                  }
                  else {
                    maxCommon[i][j] = Math.max(maxCommon[i][j-1], maxCommon[i-1][j]);
                  }
                }
        }
        return maxCommon[m][n];
}
```

Matrix Multiplication

Matrix **A** m*n

Matrix **B** n*p

C = **A** * **B** will need **m*n*p** times multiplication

A 100 * 10, B 10 * 100, C 100 * 5

D = A * B * C

If we do (AB)C, need 100*10*100 + 100*100*5 = 150000 times

If we do A(BC), need 100*10*5 + 10*100*5 = 10000 times

Matrix Multiplication

Give A0,A1,.....,An-1 n different matrixs

Find minimum of multiplication it needs to get the result

input: An array P with n+1 numbers

A0 = p0*p1, An-1 = Pn-1*Pn

Matrix Multiplication

What is the optimal sub-structure?

For the result A1*...*An, if we split from Ak

It becomes (A1*...Ak)(Ak+1*...An)

T(1,n) = T(1,k) + T(k+1,n) + p0*pk*pn

So if T(1,n) is the best, T(1,k) and T(k+1,n) must be the best

Matrix Multiplication

So how do we get the T(1,k)?

We need to start from the chain with length 1 to finally get to length n

```
public static int MatrixChain(int[] p)
{
    int n = p.length;
    n --;
    int[][] m = new int[n][n];
    for(int i = 0; i < n; i++)
        m[i][i] = 0;
    for(int r = 2; r <= n; r++)
    {
        for(int i = 0; i < n - r + 1; i ++)
        {
            int j = i + r - 1;
            m[i][j] = m[i + 1][j]  + p[i] * p[i+1] * p[j + 1];
            for(int k = i + 1; k < j; k++)
            {
                int t = m[i][k] + m[k + 1][j] + p[i] * p[k+1] * p[j+1];
                if( t < m[i][j])
                    m[i][j] = t;
            }
        }
    }
    return m[0][n-1];
}
```

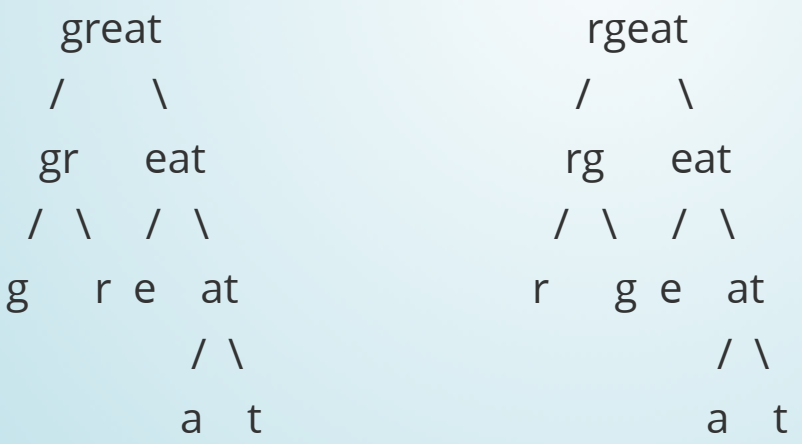
Scramble String

Given a string s1, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Below is one possible representation of s1 = "great":

To scramble the string, we may choose any non-leaf node and swap its two children.

Given two strings *s1* and *s2* of the same length, determine if *s2* is a scrambled string of *s1*.



Scramble String

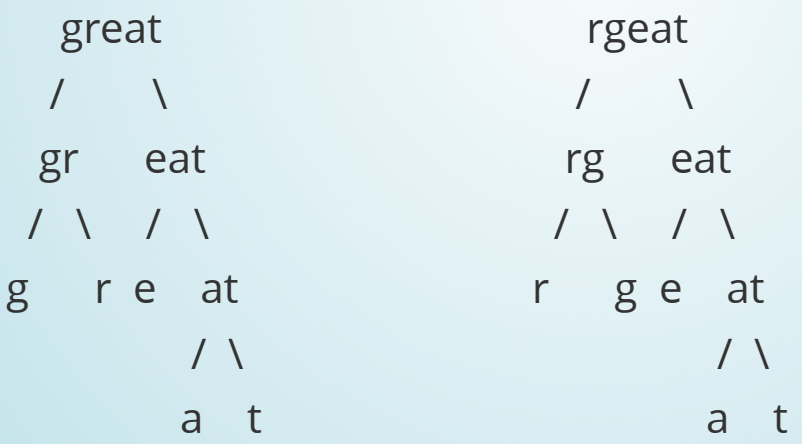
For length = 1. We know the only scramble string stands if these two are the same char

For length = 2. We know ab's scramble String could be ab or ba

Two Strings are scrambles, if (assume the string are A and B)

If any cut for A and B to A1 A2 and B1 B2. A1 is scramble of B1, A2 is scramble of B2

or A1 is scramble of B2, A2 is scramble of B1



Scramble String

```
public boolean isScramble(String s1, String s2) {
    if (s1.length() != s2.length()) return false;
    if (s1.equals(s2)) return true;
    char[] c1 = s1.toCharArray();
    char[] c2 = s2.toCharArray();
    Arrays.sort(c1);
    Arrays.sort(c2);
    if (!Arrays.equals(c1, c2)) return false;
    for (int i = 1; i < s1.length(); i++) {
        if (isScramble(s1.substring(0, i), s2.substring(0, i))
            && isScramble(s1.substring(i), s2.substring(i))) {
            return true;
        }
        if (isScramble(s1.substring(0, i), s2.substring(s2.length() - i))
            && isScramble(s1.substring(i), s2.substring(0, s2.length() - i))) {
            return true;
        }
    }
    return false;
}
```

Scramble String

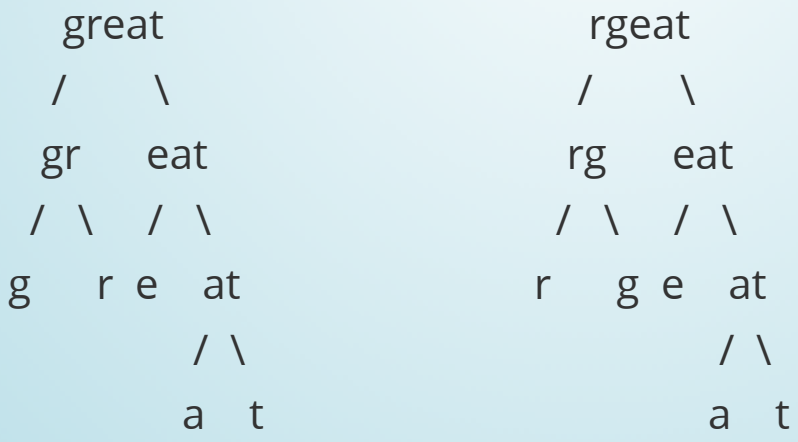
For DP. We need a three dimension Array to help us to log the information:

dp[i][j][k] means

Starting from s1(i) and starting from s2(j), substring with length k, if they are scramble

dp[i][j][1] = true if s1.charAt(i) == s2.charAt(j)

we finally want to get dp[0][0][s1.length()]



Scramble String

```
public boolean isScramble(String s1, String s2) {
    if (s1.length() != s2.length()) return false;
    if (s1.equals(s2)) return true;

    boolean[][][] dp = new boolean[s1.length()][s2.length()][s1.length() + 1];
    for (int i = 0; i < s1.length(); i++) {
        for (int j = 0; j < s2.length(); j++) {
            dp[i][j][1] = s1.charAt(i) == s2.charAt(j);
        }
    }

    for (int len = 2; len <= s1.length(); len++) {
        for (int i = 0; i < s1.length() - len + 1; i++) {
            for (int j = 0; j < s2.length() - len + 1; j++) {
                for (int k = 1; k < len; k++) {
                    dp[i][j][len] = dp[i][j][len] ||
                        (dp[i][j][k] && dp[i + k][j + k][len - k] ||
                        dp[i][j + len - k][k] && dp[i + k][j][len - k]);
                }
            }
        }
    }

    return dp[0][0][s1.length()];
```

Homework

House Robber

Maximum Subarray

Triangle

Interleaving String

House Robber II

Homework (Optional)

Distinct Subsequences

Dungeon Game

Ones and Zeroes

Maximal Square

Can I Win