Recursion - III Complete

Permutation

To print all the permutations of a string.

Idea: for each character s[i] in the given string, we add a character in the ans string and then solve s.substr(0,i) + s.substr(i+1)

Sample Input:

ABC

Sample Output:

ABC

ACB

BAC

BCA

CAB

CBA

Time Complexity: $O(N*2^n)$

Space Complexity: $O(2^n)$

```
void permutation(string s, string ans) {

    if (s.length() == 0) {
        cout << ans << endl;
        return;
    }

    for (int i = 0; i < s.length(); i++) {
        char ch = s[i];
        string ros = s.substr(0, i) + s.substr(i + 1);

        permutation(ros, ans + ch);
    }
}
```

If n =4

1) - when we place 1 horiz than above it has to be horiz
so, remaining two cols - both vertical or both horiz
so this leads us to ways - way1 - all horiz
- way 2 - 1 horiz and 2 vert
2) - way 3 - all vert.

Note when we place a tile vert it only subtracts one col, but when we place a tile horiz it occupies 2 cols

Tiling problem

Find the number of ways to tile the floor with 1x2 and 1x1 tiles.

Idea: Tile[i] = Tile[i-1] (1x1) + Tile[i-2](1x2)

Time Complexity: $O(2^n)$

Space Complexity: $O(2^n)$ //Memory is used for call stack as well

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

Knapsack [IMP]

Given n items, each item has a certain value and weight. We have to maximize the value of the objects we can accommodate in a bag of weight W.

Idea: For each item, we have two choices, either to include it or exclude it.

Time Complexity: $O(2^n)$

Space Complexity: $O(2^n)$ //space for call stack

```
int knapsack(int value[], int wt[], int n, int W) {

    if (n == 0 || W == 0) {
        return 0;
    }

    if (wt[n - 1] > W) {
        return knapsack(value, wt, n - 1, W);
    }

    return max(knapsack(value, wt, n - 1, W - wt[n - 1]) + value[n - 1],
               knapsack(value, wt, n - 1, W));
}
```

Friends Pairing Problem

There are n friends, we have to find all the pairings possible. Each person can be paired with only one person or does not pair with anyone.

Idea: we have two options, i'th friend does not get paired or we have n-1 options to pair it with anyone.

Time Complexity: O(n)

Space Complexity: O(n)

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

CountPaths

Find the number of ways to reach e from s.

Idea:

We have 6 ways to go forward (1,2,3,4,5,6).

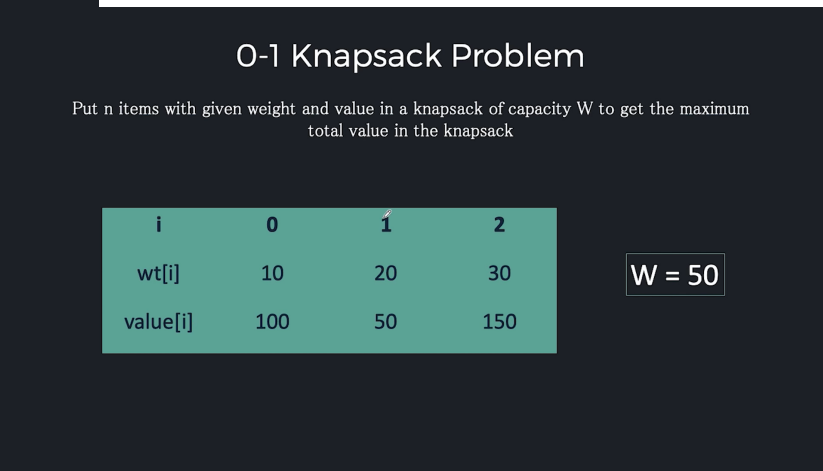
At the starting point s,

Current answer = countPath(s+1,e) + countPath(s+2,e) + countPath(s+3,e) + countPath(s+4,e) + countPath(s+5,e) + countPath(s+6,e)

Time Complexity: $O(2^n)$

Space Complexity: $O(2^n)$

```
int countPath(int s, int e) {
    if (s == e) {
        return 1;
    }
    if (s > e) {
        return 0;
    }
    int count = 0;
    for (int i = 1; i <= 6; i++) {
        count += countPath(s + i, e);
    }
    return count;
}
```



CountPathMaze

Given a 2D grid, find the number of ways to reach (n-1, n-1).

You can go to (i,j) from (i-1,j) and (i,j-1).

Time Complexity: $O(2^n)$

Space Complexity: $O(2^n)$

```
int countPathMaze(int n, int i, int j) {
    if (i == n - 1 && j == n - 1) {
        return 1;
    }
    if (i >= n || j >= n) {
        return 0;
    }

    return countPathMaze(n, i + 1, j) +
           countPathMaze(n, i, j + 1);
}
```