- Expectation
- Production ready code
  - naming convention
  - boundary conditions
    - if else
  - OO Code
    - exceptions
  - language
- Working code
- TC, SC: optimization
- Non-Algorithmic: working code -> test cases -> (TDD)

Before Interview:

- Online Test/Assessment
- Assignments

Soft Skills

- Clarifying things, are you asking questions ?
- Think of things and make appropriate assumptions (Decision based on trade offs,)
- Don't talk, talk but verbose.
- Correct
- Time ?
- Do you listen: hint/don't do this/what if I do this ?

In [ ]:

```
```

Knapsack problem

- Given max weight W
- array of weights and values
- optimize for max value, staying within the limit of max weight

Return max value that can be put in knapsack

```
values  [10 20 60]
weights [20 40 30]

W = 60
```

In [4]:

```
## Brute force recursive solution

def knapsack(w, values, weights):
    return knapsack_util(w, values, weights, 0)


def knapsack_util(w, values, weights, curr):
    if curr >= len(values):
        return 0

    if weights[curr] > w:
        return knapsack_util(w,values, weights, curr+1)

    include = values[curr] + knapsack_util(w-weights[curr],values, weights, cu
    exclude = knapsack_util(w,values, weights, curr+1)

    return max(include, exclude)

values  = [10, 20, 60]
weights = [20, 40, 30]

print(knapsack(60, values, weights))

# N=len(weights)
# TC: O(2^N)
# SC: O(N)
```

```
70
```

Memoization

```
int knapsack_util(int w, vector<int>values, vector<int> weights, int
curr,map<pair<int,int>,int>& dp)
{
    if (curr >= values.size())
        return 0;

    if (weights[curr] > w)
        return knapsack_util(w,values, weights, curr+1,dp);

    if(dp.find({w,curr})!=dp.end()) return dp[{w,curr}];

    int include = values[curr] + knapsack_util(w-weights[curr],value
s, weights, curr+1,dp);
    int exclude = knapsack_util(w,values, weights, curr+1,dp);

    int w1=weights[curr];
    dp[{w,curr}] = max(include,exclude);

    return max(include, exclude);

}
```

Tabulation

```
public int bottomUp(int w, int [] weights, int [] values){
        int n = values.length;

        int [][] dp = new int [n+1][w+1];
        for(int i = 1 ; i <= n;  i++){
            for(int j = 0 ; j <= w ; j++){
                if(weights[i-1] > j ) dp[i][j] = dp[i-1][j];
                else{
                 dp[i][j] = Math.max(values[i-1]+dp[i-1][j-weights[i-
1]] , dp[i-1][j]);
                }
            }
        }
```

In [ ]:

In [ ]:

Subset Sum

Given a list of numbers. Figure out if it is possible to have the given sum S. Using any subset of given numbers. Use a number from the subset only once.

Numbers: [2,4,6,4] S: 10

def isSubsetSumPossible(numbers, s): pass

In [ ]:

https://leetcode.com/problems/coin-change/ (https://leetcode.com/problems/coin-change/)

In [ ]: