**BackTracking**

Subsets : https://leetcode.com/problems/subsets/

```
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> list = new ArrayList<>();
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, 0);
    return list;}
private void backtrack(List<List<Integer>> list , List<Integ
er> tempList, int [] nums, int start){
    list.add(new ArrayList<>(tempList));
    for(int i = start; i < nums.length; i++){
        tempList.add(nums[i]);
        backtrack(list, tempList, nums, i + 1);
        tempList.remove(tempList.size() - 1);}}
```

Subsets II (contains duplicates) : https://leetcode.com/problems/subsets-ii/

```
public List<List<Integer>> subsetsWithDup(int[] nums) {
    List<List<Integer>> list = new ArrayList<>();
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, 0);
    return list;}
private void backtrack(List<List<Integer>> list, List<Intege
r> tempList, int [] nums, int start){
    list.add(new ArrayList<>(tempList));
    for(int i = start; i < nums.length; i++){
        if(i > start && nums[i] == nums[i-1]) continue;
        tempList.add(nums[i]);
        backtrack(list, tempList, nums, i + 1);
        tempList.remove(tempList.size() - 1);}}
```

Permutations : https://leetcode.com/problems/permutations/

```
public List<List<Integer>> permute(int[] nums) {
   List<List<Integer>> list = new ArrayList<>();
   // Arrays.sort(nums); // not necessary
   backtrack(list, new ArrayList<>(), nums);
   return list;
}


private void backtrack(List<List<Integer>> list, List<Intege
r> tempList, int [] nums){
   if(tempList.size() == nums.length){
      list.add(new ArrayList<>(tempList));
   } else{
      for(int i = 0; i < nums.length; i++){
         if(tempList.contains(nums[i])) continue; // element
already exists, skip
         tempList.add(nums[i]);
         backtrack(list, tempList, nums);
         tempList.remove(tempList.size() - 1);
      }
   }
}
```

Permutations II (contains duplicates) : https://leetcode.com/problems/permutations-ii/

```
public List<List<Integer>> permuteUnique(int[] nums) {
    List<List<Integer>> list = new ArrayList<>();
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, new boolean[nums.length]);
    return list;
}


private void backtrack(List<List<Integer>> list, List<Integer> tempList, int [] nums, boolean [] used){
    if(tempList.size() == nums.length){
        list.add(new ArrayList<>(tempList));
    } else{
        for(int i = 0; i < nums.length; i++){
            if(used[i] || i > 0 && nums[i] == nums[i-1] && !used[i - 1]) continue;
            used[i] = true;
            tempList.add(nums[i]);
            backtrack(list, tempList, nums, used);
            used[i] = false;
            tempList.remove(tempList.size() - 1);
        }
    }
}
```

---

Combination Sum : https://leetcode.com/problems/combination-sum/

```
public List<List<Integer>> combinationSum(int[] nums, int target) {
    List<List<Integer>> list = new ArrayList<>();
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, target, 0);
    return list;
}


private void backtrack(List<List<Integer>> list, List<Integer> tempList, int [] nums, int remain, int start){
    if(remain < 0) return;
    else if(remain == 0) list.add(new ArrayList<>(tempList));
    else{
        for(int i = start; i < nums.length; i++){
            tempList.add(nums[i]);
            backtrack(list, tempList, nums, remain - nums[i], i); // not i + 1 because we can reuse same elements
            tempList.remove(tempList.size() - 1);
        }
    }
}
```

Combination Sum II (can't reuse same element) : https://leetcode.com/problems/combination-sum-ii/

```
public List<List<Integer>> combinationSum2(int[] nums, int target) {
    List<List<Integer>> list = new ArrayList<>();
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, target, 0);
    return list;

}

private void backtrack(List<List<Integer>> list, List<Integer> tempList, int [] nums, int remain, int start){
    if(remain < 0) return;
    else if(remain == 0) list.add(new ArrayList<>(tempList));
    else{
        for(int i = start; i < nums.length; i++){
            if(i > start && nums[i] == nums[i-1]) continue; // skip duplicates
            tempList.add(nums[i]);
            backtrack(list, tempList, nums, remain - nums[i], i + 1);
            tempList.remove(tempList.size() - 1);
        }
    }
}
```

Palindrome Partitioning : https://leetcode.com/problems/palindrome-partitioning/

```
public List<List<String>> partition(String s) {
   List<List<String>> list = new ArrayList<>();
   backtrack(list, new ArrayList<>(), s, 0);
   return list;
}


public void backtrack(List<List<String>> list, List<String>
tempList, String s, int start){
   if(start == s.length())
      list.add(new ArrayList<>(tempList));
   else{
      for(int i = start; i < s.length(); i++){
         if(isPalindrome(s, start, i)){
            tempList.add(s.substring(start, i + 1));
            backtrack(list, tempList, s, i + 1);
            tempList.remove(tempList.size() - 1);
         }
      }
   }
}


public boolean isPalindrome(String s, int low, int high){
   while(low < high)
      if(s.charAt(low++) != s.charAt(high--)) return false;
   return true;
}
```

**Dynamic Programming**

There is some frustration when people publish their perfect fine-grained algorithms without sharing any information abut how they were derived. This is an attempt to change the situation. There is not much more explanation but it's rather an example of higher level improvements. Converting a solution to the next step shouldn't be as hard as attempting to come up with perfect algorithm at first attempt.

This particular problem and most of others can be approached using the following sequence:

1. Find recursive relation
2. Recursive (top-down)
3. Recursive + memo (top-down)
4. Iterative + memo (bottom-up)
5. Iterative + N variables (bottom-up)

**Step 1.** Figure out recursive relation.
A robber has 2 options: a) rob current house `i`; b) don't rob current house.
If an option "a" is selected it means she can't rob previous `i-1` house but can safely proceed to the one before previous `i-2` and gets all cumulative loot that follows.
If an option "b" is selected the robber gets all the possible loot from robbery of `i-1` and all the following buildings.
So it boils down to calculating what is more profitable:

- robbery of current house + loot from houses before the previous
- loot from the previous house robbery and any loot captured before that

```
rob(i) = Math.max( rob(i - 2) + currentHouseValue, rob(i - 1) )
```

**Step 2.** Recursive (top-down)
Converting the recurrent relation from Step 1 shound't be very hard.

```
public int rob(int[] nums) {
    return rob(nums, nums.length - 1);}
private int rob(int[] nums, int i) {
    if (i < 0) { return 0;}
    return Math.max(rob(nums, i - 2) + nums[i], rob(nums, i - 1));}
```

This algorithm will process the same `i` multiple times and it needs improvement. Time complexity: [to fill]

**Step 3.** Recursive + memo (top-down).

```
int[] memo;
public int rob(int[] nums) {
    memo = new int[nums.length + 1];
    Arrays.fill(memo, -1);
    return rob(nums, nums.length - 1);
}

private int rob(int[] nums, int i) {
    if (i < 0) {
        return 0;
    }
    if (memo[i] >= 0) {
        return memo[i];
    }
    int result = Math.max(rob(nums, i - 2) + nums[i], rob(num
s, i - 1));
    memo[i] = result;
    return result;
}
```

Much better, this should run in `O(n)` time. Space complexity is `O(n)` as well, because of the recursion stack, let's try to get rid of it.

**Step 4.** Iterative + memo (bottom-up)

```
public int rob(int[] nums) {
    if (nums.length == 0) return 0;
    int[] memo = new int[nums.length + 1];
    memo[0] = 0;
    memo[1] = nums[0];
    for (int i = 1; i < nums.length; i++) {
        int val = nums[i];
        memo[i+1] = Math.max(memo[i], memo[i-1] + val);
    }
    return memo[nums.length];
}
```

**Step 5.** Iterative + 2 variables (bottom-up)

We can notice that in the previous step we use only `memo[i]` and `memo[i-1]`, so going just 2 steps back. We can hold them in 2 variables instead. This optimization is met in Fibonacci sequence creation and some other problems [to paste links].

```
/* the order is: prev2, prev1, num  */
public int rob(int[] nums) {
    if (nums.length == 0) return 0;
    int prev1 = 0;
    int prev2 = 0;
    for (int num : nums) {
        int tmp = prev1;
        prev1 = Math.max(prev2 + num, prev1);
        prev2 = tmp;
    }
    return prev1;
}
```

# 01背包的状态转换方程 f[i,j] = Max{ f[i-1,j-Wi]+Pi( j >= Wi )，f[i-1,j] }

f[i,j]表示在前i件物品中选择若干件放在承重为 j 的背包中，可以取得的最大价值。
Pi表示第i件物品的价值。
决策：为了背包中物品总价值最大化，第 i件物品应该放入背包中吗？

**题目描述：**

**假设山洞里共有a,b,c,d ,e这5件宝物（不是5种宝物），它们的重量分别是2,2,6,5,4，它们的价值分别是6,3,5,4,6，现在给你个承重为10的背包，怎么装背包，可以才能带走最多的财富。**

有编号分别为a,b,c,d,e的五件物品，它们的重量分别是2,2,6,5,4，它们的价值分别是6,3,5,4,6，现在给你个承重为10的背包，如何让背包里装入的物品具有最大的价值总和?

| name | weight | value | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | 2 | 6 | 0 | 6 | 6 | 9 | 9 | 12 | 12 | 15 | 15 | 15 |
| b | 2 | 3 | 0 | 3 | 3 | 6 | 6 | 9 | 9 | 9 | 10 | 11 |
| c | 6 | 5 | 0 | 0 | 0 | 6 | 6 | 6 | 6 | 6 | 10 | 11 |
| d | 5 | 4 | 0 | 0 | 0 | 6 | 6 | 6 | 6 | 6 | 10 | 10 |
| e | 4 | 6 | 0 | 0 | 0 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |

只要你能通过找规律手工填写出上面这张表就算理解了01背包的动态规划算法。

首先要明确这张表是至底向上，从左到右生成的。

为了叙述方便，用e2单元格表示e行2列的单元格，这个单元格的意义是用来表示只有物品e时，有个承重为2的背包，那么这个背包的最大价值是0，因为e物品的重量是4，背包装不了。

对于d2单元格，表示只有物品e，d时,承重为2的背包,所能装入的最大价值，仍然是0，因为物品e,d都不是这个背包能装的。

同理，c2=0，b2=3,a2=6。

对于承重为8的背包，a8=15,是怎么得出的呢?

根据01背包的状态转换方程，需要考察两个值，

一个是f[i-1,j],对于这个例子来说就是b8的值9，另一个是f[i-1,j-Wi]+Pi；

在这里，

f[i-1,j]表示我有一个承重为8的背包，当只有物品b,c,d,e四件可选时，这个背包能装入的最大价值

f[i-1,j-Wi]表示我有一个承重为6的背包（等于当前背包承重减去物品a的重量），当只有物品b,c,d,e四件可选时，这个背包能装入的最大价值

f[i-1,j-Wi]就是指单元格b6,值为9，Pi指的是a物品的价值，即6

由于f[i-1,j-Wi]+Pi = 9 + 6 = 15 大于f[i-1,j] = 9，所以物品a应该放入承重为8的背包

```
public function get01PackageAnswer(bagItems:Array,bagSize:int):Array
{
    var bagMatrix:Array=[],   var i:int,   var item:PackageItem;
    for(i=0;i<bagItems.length;i++)
    {    bagMatrix[i] = [0]; }
    for(i=1;i<=bagSize;i++)
    {
        for(var j:int=0;j<bagItems.length;j++)
        {
            item = bagItems[j] as PackageItem;
            if(item.weight > i)
            {
                if(j==0)   //i 背包转不下 item
                {    bagMatrix[j][i] = 0;}
                else{ bagMatrix[j][i]=bagMatrix[j-1][i]; }
            }
            else{
                var itemInBag:int; //将 item 装入背包后的价值总和
                if(j==0){
                    bagMatrix[j][i] = item.value;
                    continue;}
                else{itemInBag = bagMatrix[j-1][i-item.weight]+item.value;}
        bagMatrix[j][i] = (bagMatrix[j-1][i] > itemInBag ? bagMatrix[j-1][i] : itemInBag)}
        }
    }
    var answers:Array=[],  var curSize:int = bagSize; //find answer
    for(i=bagItems.length-1;i>=0;i--)
    {
        item = bagItems[i] as PackageItem;
        if(curSize==0) break;
        if(i==0 && curSize > 0)
        {
            answers.push(item.name);
            break;
        }
        if(bagMatrix[i][curSize]-bagMatrix[i-1][curSize-item.weight]==item.value)
        {
            answers.push(item.name);
            curSize -= item.weight;}}
    return answers;
}
```