**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**<Integer> 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**<Integer> 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**<Integer> 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(nums, 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;

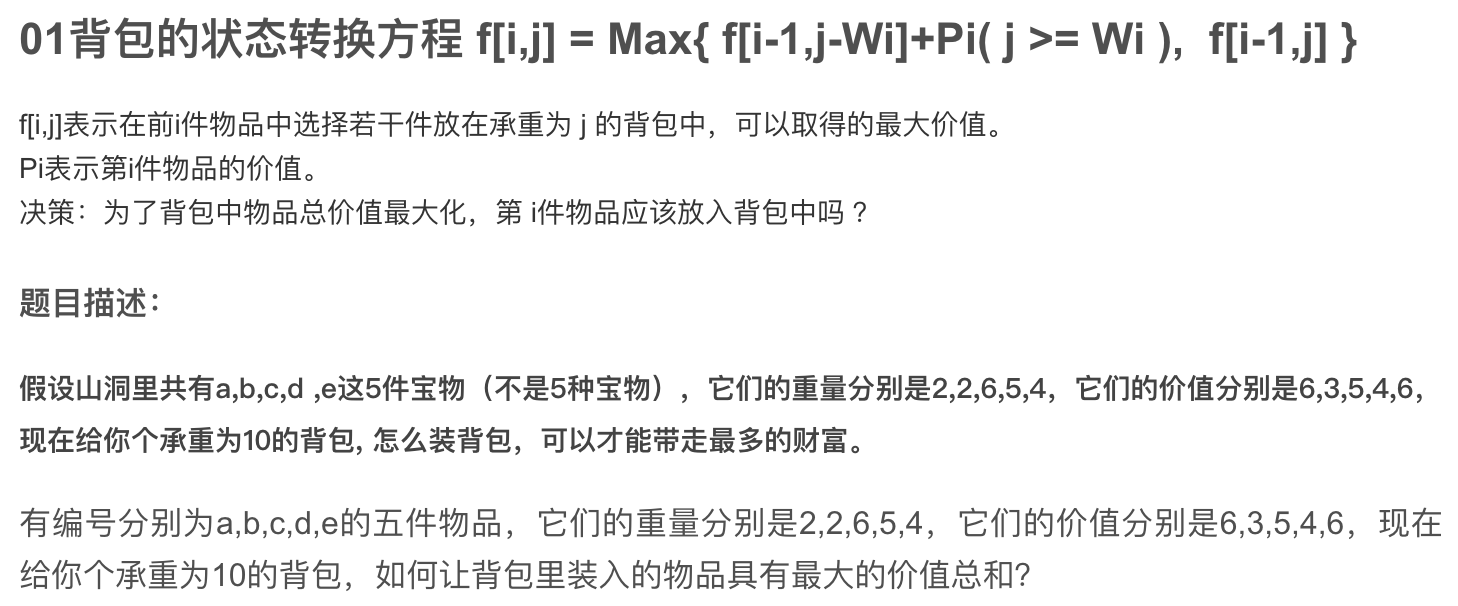
prev1 = Math.max(prev2 + num, prev1);

prev2 = tmp;

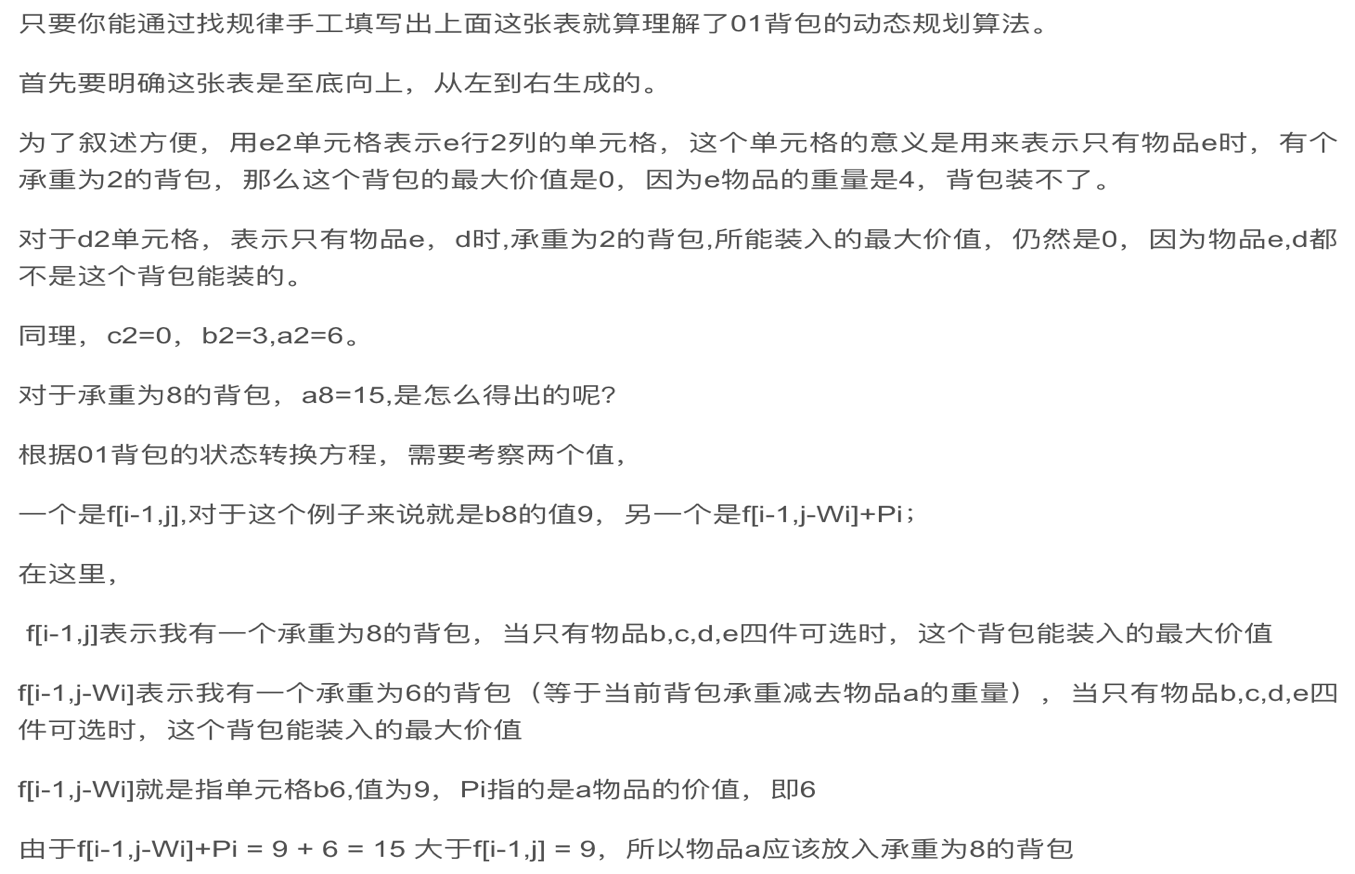
}

return prev1;

}







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;

}