

0-1 knapsack

- **A better implementation**
- $\text{KNAPSACK}(W, w_1, \dots, w_n, v_1, \dots, v_n)$
(assume $\text{opt}[w] = -\infty$ for all w initially)
 $\text{opt}[0] \leftarrow 0$
for $k = 1, \dots, n$ **do**
 for $w = W, \dots, 1$ **do**
 $\text{opt}[w] \leftarrow \max\{\text{opt}[w], \text{opt}[w - w_k] + v_k\}$
return $\max_{i \in \{1, \dots, W\}} \text{opt}[w]$
- Why does this implementation work?
- Time complexity = $O(nW)$

0-1 knapsack

- A_k = the set of items with indices $1, \dots, k$
- $P_{w,k}$ = getting maximum value **using items in A_k** with total weight w
- **$\text{opt}(P_{w,k}) = \max\{\text{opt}(P_{w,k-1}), \text{opt}(P_{w-w_k,k-1}) + v_k\}$**
- **Standard implementation**
- $\text{KNAPSACK}(W, w_1, \dots, w_n, v_1, \dots, v_n)$
(assume $\text{opt}[w, k] = -\infty$ for all w and k initially)
 $\text{opt}[0, k] \leftarrow 0$ for all $k \in \{1, \dots, n\}$
for $k = 1, \dots, n$ **do**
 for $w = 1, \dots, W$ **do**
 $\text{opt}[w, k] \leftarrow \max\{\text{opt}[w, k-1], \text{opt}[w-w_k, k-1] + v_k\}$
return $\max_{w \in \{1, \dots, W\}} \text{opt}[w, n]$

Cut the cake!

- Each piece in \mathcal{K} can be described by a tuple (i^-, i^+, j^-, j^+) .
 i^-/i^+ = index of the topmost/bottommost row
 j^-/j^+ = index of the leftmost/rightmost column
- The original cake is $(1, n, 1, m)$.
- If we cut a piece (i^-, i^+, j^-, j^+) , what are the two resulting pieces?
Horizontal: (i^-, i, j^-, j^+) and $(i+1, i^+, j^-, j^+)$ for some $i \in [i^-, i^+)$.
Vertical: (i^-, i^+, j^-, j) and $(i^-, i^+, j+1, j^+)$ for some $j \in [j^-, j^+)$.
- $\text{opt}_{i^-, i^+, j^-, j^+}^h = \min_{i \in [i^-, i^+)} (j^+ - j^- + 1 + \text{opt}_{i^-, i, j^-, j^+} + \text{opt}_{i+1, i^+, j^-, j^+})$
 $\text{opt}_{i^-, i^+, j^-, j^+}^v = \min_{j \in [j^-, j^+)} (i^+ - i^- + 1 + \text{opt}_{i^-, i^+, j^-, j} + \text{opt}_{i^-, i^+, j+1, j^+})$
- $\text{opt}_{i^-, i^+, j^-, j^+} = \min\{\text{opt}_{i^-, i^+, j^-, j^+}^h, \text{opt}_{i^-, i^+, j^-, j^+}^v\}$
- Boundary case**
 $\text{opt}_{i^-, i^+, j^-, j^+} = 0$ if (i^-, i^+, j^-, j^+) contains at most one cherry.

Expression evaluation

- $P_{i,j}$ = evaluating the expression $x_i + \cdots + x_j$
 $\mathcal{S} = \{P_{i,j} : 1 \leq i \leq j \leq n\}$
- How many subproblems are there? $1 + \cdots + n = O(n^2)$
- Let's write $\Sigma_i^j = x_i + \cdots + x_j$.
- $\text{opt}(P_{i,j}) = \min_{k=i}^{j-1} (\text{COST}(\Sigma_i^k, \Sigma_{k+1}^j) + \text{opt}(P_{i,k}) + \text{opt}(P_{k+1,j}))$
- **Boundary case**
 $\text{opt}(P_{i,j}) = 0$ if $i = j$ (no “+” operation is needed)

- We can solve the problem using **DP** as before.
- P_i = computing the shortest path $A_1 \rightarrow A_i$
 $\mathcal{S} = \{P_1, \dots, P_n\}$
- Note that P_i only depends on P_j for $j < i$.
- **SHORTESTPATH**(n, A_1, \dots, A_n, E)
 - $\text{opt}[1] \leftarrow 0$
 - for** $i = 2, \dots, n$ **do**
 - $E_i \leftarrow \{e \in E : \text{target}(e) = i\}$
 - if** $E_i = \emptyset$ **then** $\text{opt}[i] \leftarrow +\infty$
 - else** $\text{opt}[i] \leftarrow \min_{e \in E_i} (\text{length}(e) + \text{opt}[\text{source}(e)])$
 - return** $\text{opt}[n]$
- Time complexity = $O(n^2)$ or $O(n + |E|)$ if implemented carefully

```
def minDistance(self, word1: str, word2: str) -> int:
    if len(word1) * len(word2) == 0:
        return len(word2) + len(word1)

    dp = [[0] * (len(word2) + 1) for i in range(len(word1) + 1)]
    for i in range(len(word1) + 1):
        ## obvious that if w2 have no element, edit is just i
        dp[i][0] = i
    for j in range(len(word2) + 1):
        dp[0][j] = j

    ## it would be equal to: -> why don't need to worry of push the letter?
    ## add 1 to A
    ## add 1 to B
    ## modify 1 in A/B (same for either)
    for i in range(1, len(word1) + 1):
        for j in range(1, len(word2) + 1):
            ## there would be two situations:
            if word1[i - 1] == word2[j - 1]:
                ## then it would save 1 distance of matching these two
                dp[i][j] = min(dp[i - 1][j] + 1, dp[i][j - 1] + 1, dp[i - 1][j - 1])
            else:
                dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1])
    return dp[len(word1)][len(word2)]
```

Fractional knapsack

- Next let's formally describe the greedy strategy.
- Without loss of generality, suppose $\frac{v_1}{w_1} \geq \dots \geq \frac{v_n}{w_n}$.
- Find the maximum index k satisfying $\sum_{i=1}^k w_i \leq W$.
- $\rho_i = 1$ for all $i \in \{1, \dots, k\}$
 $\rho_{k+1} = (W - \sum_{i=1}^k w_i) / w_{k+1}$
 $\rho_i = 0$ for all $i \in \{k+2, \dots, n\}$
- **Proof of correctness**
Assume there exists an optimal solution whose ρ_i is the same as the greedy solution for all $i < t$, and show the existence of an optimal solution whose ρ_i is the same as the greedy solution for all $i \leq t$.

```
8 }
}
```

扩展欧几里得法和求解 [线性同余方程](#) 是一个原理，在这里不展开解释。

快速幂法


证明

因为 $ax \equiv 1 \pmod b$;

所以 $ax \equiv a^{b-1} \pmod b$ (根据 [费马小定理](#)) ；

所以 $x \equiv a^{b-2} \pmod b$ 。

然后我们就可以用快速幂来求了。

 实现

C++Python

```
1 int qpow(long long a, int b) {
2     int ans = 1;
3     a = (a % p + p) % p;
4     for (; b; b >>= 1) {
5         if (b & 1) ans = (a * ans) % p;
6         a = (a * a) % p;
7     }
8     return ans;
9 }
```

注意：快速幂法使用了 [费马小定理](#)，要求 b 是一个素数；而扩展欧几里得法只要求 $\gcd(a, b) = 1$ 。

目录

定义


如何求逆元

[扩展欧几里得法](#)

快速幂法

证明

线性求逆元



线性求任意 n 个数的逆元

逆元练习题



```
*/  
class Solution {  
public:  
  
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {  
        if (root == nullptr || p == root || q == root){  
            return root;  
       }  
  
        TreeNode* l = lowestCommonAncestor(root->left, p, q);  
        TreeNode* r = lowestCommonAncestor(root->right, p, q);  
  
        return l == nullptr ? r: (r == nullptr ? l : root);  
    }  
};
```

- $\text{LCS}(n_A, A, n_B, B)$
 $\text{opt}[i, 0] \leftarrow 0$ and $\text{opt}[0, j] \leftarrow 0$ for $i, j \in \{0, 1, \dots, n\}$
 for $i = 1, \dots, n_A$ **do**
 for $j = 1, \dots, n_B$ **do**
 $\text{opt}[i, j] \leftarrow \max\{\text{opt}[i - 1, j], \text{opt}[i, j - 1]\}$
 if $A[i] = B[j]$ and $\text{opt}[i, j] < \text{opt}[i - 1, j - 1] + 1$ **then**
 $\text{opt}[i, j] \leftarrow \text{opt}[i - 1, j - 1] + 1$
 return $\text{opt}[n_A, n_B]$
- The above code only returns the **optimum**, i.e., the **length** of an LCS.
 Try to figure out by yourself how to retrieve an **LCS**!
- Time complexity = $O(n_A n_B) = O(n^2)$ where $n = n_A + n_B$

Longest increasing subsequence

- $\text{opt}(P_i) = \max_{j \in C(P_i)} (\text{opt}(P_j) + 1) = \max_{j \in C(P_i)} \text{opt}(P_j) + 1$

- $\text{LIS}(n, A)$

for $i = 1, \dots, n$ **do**

$m \leftarrow 0$ and $k \leftarrow 0$

for $j = 1, \dots, i - 1$ **do**

if $A[j] < A[i]$ and $\text{opt}[j] > m$ **then**

$m \leftarrow \text{opt}[j]$ and $k \leftarrow j$

$\text{opt}[i] \leftarrow m + 1$ and $\text{pos}[i] \leftarrow j$

$i^* = \arg \max_{i \in \{1, \dots, n\}} \text{opt}[i]$

$i \leftarrow i^*$ and $B \leftarrow [A[i]]$

while $\text{pos}[i] > 0$ **do**

$i \leftarrow \text{pos}[i]$ and $B \leftarrow [A[i]] + B$

return B

both_changing > main.cpp > solve()

```
# include <bits/stdc++.h>
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
```

```
using namespace std;
```

```
void solve(){
```

```
    map<char, int> m;
```

```
    m['O'] = 1;
```

```
    m['B'] = 2;
```

```
    m['A'] = 3;
```

```
    m['F'] = 4;
```

```
    m['G'] = 5;
```

```
    m['K'] = 6;
```

```
    m['M'] = 7;
```

- $\text{MAXSUBARRAY}(n, A)$

$\text{opt}[1] \leftarrow A[1]$ and $C[1] \leftarrow \text{"stop"}$

for $i = 2, \dots, n$ **do**

$\text{opt}[i] \leftarrow \max\{0, \text{opt}[i - 1]\} + A[i]$

if $0 < \text{opt}[i - 1]$ **then** $C[i] \leftarrow \text{"go left"}$

else $C[i] \leftarrow \text{"stop"}$

$i^* \leftarrow \arg \max_{i \in \{1, \dots, n\}} \text{opt}[i]$

$j \leftarrow i^*$

while $C[j] = \text{"go left"}$ **do**

$j \leftarrow j - 1$

return $A[j \dots i^*]$

- Why not consider the **configurations** of the **last column** when defining the subproblems?
- Let $P_{i,x}$ be the problem of picking non-adjacent entries in **the first i column of M** such that the i -th column has configuration **x** .
- $\Gamma = \{0000, 1000, 0100, 0010, 0001, 1010, 0101\}$
 $\mathcal{S} = \{P_{i,x} : i \in \{1, \dots, n\} \text{ and } x \in \Gamma\}$
- $\text{opt}(P_{i,x}) = \max_{y \in \Gamma, y \wedge x = 0000} (\text{opt}(P_{i-1,y}) + \sum_{k=1}^4 x_i M_{k,i})$
- Final optimum = $\max_{P_{i,x} \in \mathcal{S}} \text{opt}(P_{i,x})$
- Time complexity = $O(n)$
- Can be generalized to matrix of size $c \times n$ for a **constant c** .

198. 打家劫舍

已解答

中等

🏷 相关标签

🏢 相关企业

Ag

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你不触动警报装置的情况下，一夜之内能够偷窃到的最高金额。

示例 1:

输入: [1,2,3,1]

输出: 4

解释: 偷窃 1 号房屋 (金额 = 1) , 然后偷窃 3 号房屋 (金额 = 3)。

偷窃到的最高金额 = 1 + 3 = 4 。

示例 2:

输入: [2,7,9,3,1]

输出: 12

解释: 偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9), 接着偷窃 5 号房屋 (金额 = 1)。

偷窃到的最高金额 = 2 + 9 + 1 = 12 。

C++ 智能模式

```
1 class Solution {
2 public:
3     int rob(vector<int>& nums) {
4         int n = nums.size();
5         if (n == 1){
6             return nums[0];
7         }
8
9
10
11         int mx = 0, a, b;
12         a = nums[0];
13         b = max(a, nums[1]);
14         mx = max(a, b);
15         for (int i = 2; i < n; i++){
16             int cur = b;
17             b = max(b, a+ nums[i]);
18             a = cur;
19             mx = max(mx,b);
20         }
21
22         return mx;
23     }
24 };
```

已存储

✅ 测试用例 > 测试结果

213. 打家劫舍 II

已解答

中等 相关标签 相关企业 提示 Aa

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都围成一圈，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你 在不触动警报装置的情况下，今晚能够偷窃到的最高金额。

示例 1:

输入: nums = [2,3,2]
输出: 3
解释: 你不能先偷窃 1 号房屋 (金额 = 2)，然后偷窃 3 号房屋 (金额 = 2)，因为他们是相邻的。

示例 2:

输入: nums = [1,2,3,1]
输出: 4
解释: 你可以先偷窃 1 号房屋 (金额 = 1)，然后偷窃 3 号房屋 (金额 = 3)。
偷窃到的最高金额 = 1 + 3 = 4。

示例 3:

输入: nums = [1,2,3]

C++ 智能模式

```
1 class Solution {
2 public:
3     int robRange(vector<int>& nums, int start, int end){
4         int mx, a, b;
5         a = nums[start];
6         b = max(a, nums[start + 1]);
7         mx = b;
8         for (int i = start + 2; i <= end; i++){
9             int cur = b;
10            b = max(a + nums[i], b);
11            a = cur;
12            mx = max(b, mx);
13        }
14        return mx;
15    };
16
17    int rob(vector<int>& nums) {
18        // 8: 47 start
19        // maximum sum without adjacent neighbor
20
21        int n = nums.size();
22        if (n == 1){
23            return nums[0];
24        }else if (n == 2){
25            return max(nums[1], nums[0]);
26        }else{
27            return max(robRange(nums, 0, n-2), robRange(nums, 1, n - 1));
28        }
29    }
30 }
```

已存储