- **动态规划**
  - 线性DP
    - Kanade 算法
      - 最大子数组问题
      - 环形子数组的最大和
    - 乘积最大子数组
    - 乘积为正数的最长子数组长度
    - 最佳观光组合
    - 最长上升子序列LIS系列
      - 最长上升子序列
        - 方法一：动态规划
        - 方法二：贪心 + 二分查找
      - 最长上升子序列的个数
      - 二维LIS（套娃问题）
    - 股票问题
      - 买卖股票的最佳时机
      - 最佳买卖股票时机含冷冻期
      - 买卖股票的最佳时机含手续费
      - 买卖股票的最佳时机 (k次交易)
    - 打家劫舍
      - 打家劫舍1(直线)
      - 打家劫舍2(环)
  - 前缀和
    - 一维前缀和
    - 二维前缀和
  - 区间DP
    - 常数规模子问题
      - 最长回文子串
        - 方法一：区间dp
        - 方法二：中心拓展算法
      - 最长回文子序列
      - 最长公共子序列（LCS）
    - O(n)规模子问题
      - 矩阵链乘
      - 最优二叉搜索树
      - 凸多边形的划分
  - 背包问题
    - 01背包
      - 01背包原理
      - 01背包最优解回溯
      - 01背包是否恰好装满问题
    - 完全背包
    - 多重背包
    - 三种背包函数形式(组合背包)
  - 树形DP
    - 对树根进行状态分析
      - 树形DP初步-真树
  - 其他问题
    - 矩阵类型
      - 流水线问题
      - 不同路径
    - 丑数 II
    - 双调欧几里得TSP问题

动态规划

- dp一般步骤：
    1. 状态的表示（建立模型f(n)）
    2. 阶段的划分（明确状态的几种形式）
    3. **寻找状态转移方程**
    4. 边界条件（需按照初始条件或者边界条件）

线性DP

Kanade 算法

对于一个给定数组 A，Kadane 算法可以用来找到 A 的最大子段和。这里，我们只考虑非空子段。

Kadane 算法基于动态规划。令 dp[j] 为以 A[j] 结尾的最大子段和。也就是，

$$\mathrm{dp}[j] = \max_i(A[i] + A[i+1] + \cdots + A[j])$$

那么，以 `j+1` 结尾的子段（例如 `A[i], A[i+1] + ... + A[j+1]`）最大化了 `A[i] + ... + A[j]` 的和，当这个子段非空那么就等于 dp[j] 否则就等于 0。所以，有以下递推式：

$$\mathrm{dp}[j+1] = A[j+1] + \max(\mathrm{dp}[j], 0)$$

由于一个子段一定从某个位置截止，所以：

$$\max_j dp[j] jmax$$

dp[j] 就是需要的答案。

为了计算 dp 数组更快，Kadane 算法通常节约空间复杂度的形式表示。我们只维护两个变量 ans 等于

$$\max_j dp[j]$$

和 cur 等于 dp[j]。随着 j 从 0 到 A.length−1 遍历。

最大子数组问题

- 最大子数组传统做法的时间按复杂度为O（n*n）,如果此采用教材上的分治方法可以缩减至O（n*lgn）,但是**如果使用动态规划的话，可以直接变成O(n).**
- 股票交易

Kanade 算法

$$MaxSum[i] = Max(MaxSum[i-1] + A[i], A[i])$$

参考代码如下：

```c
int maxSubArray(int* nums, int numsSize) {
    int pre = nums[0], maxAns = nums[0];
    for (int i = 1; i < numsSize; i++) {
        pre = fmax(pre + nums[i], nums[i]);
        maxAns = fmax(maxAns, pre);
    }
    return maxAns;
}
```

环形子数组的最大和

环形数组最大区间和分为两种情况：

1. 不跨越边界，子数组是连续的，求解子数组最大和（一次kadane简单解决）；
2. 跨越边界，则没有加入到子数组最大和中的元素是连续的，这部分即为子数组最小和，可用一次kadane简单解决，最大和问题转化为 **数组之和 - 子数组最小和**；

```c
int maxSubarraySumCircular(int* nums, int numsSize){
    int maxCur = nums[0], maxAns = nums[0];
    int minCur = nums[0], minAns = nums[0];
    int sum=nums[0];
    for (int i = 1; i < numsSize; i++) {
        sum+=nums[i];
        maxCur = fmax(maxCur + nums[i], nums[i]);
        minCur = fmin(minCur + nums[i], nums[i]);
        maxAns = fmax(maxAns, maxCur);
        minAns = fmin(minAns, minCur);
    }
    if(maxAns<0)
        return maxAns;
    else
        return fmax(sum - minAns,maxAns);
}
```

乘积最大子数组

$$fmax(i) = \max_{i=1}^{n}\{fmax(i-1) \times a_i, fmin(i-1) \times a_i, a_i\}$$

$$fmin(i) = \min_{i=1}^{n}\{fmax(i-1) \times a_i, fmin(i-1) \times a_i, a_i\}$$

```c
int maxProduct(int* nums, int numsSize){
    int pre1 = nums[0], pre2 = nums[0],maxAns = nums[0];
    for (int i = 1; i < numsSize; i++) {
        int tmp_1 = pre1,tmp_2 = pre2;
        pre1 = fmax(tmp_1 * nums[i], fmax(tmp_2 * nums[i], nums[i]));
        pre2 = fmin(tmp_1 * nums[i], fmin(tmp_2 * nums[i], nums[i]));
        maxAns = fmax(maxAns, pre1);
    }
    return maxAns;
}
```

乘积为正数的最长子数组长度

当 $i = 0$ 时，以下标 i 结尾的子数组的长度为 1，因此当 $nums[0] > 0$ 时 $positive[0] = 1$，当 $nums[0] < 0$ 时 $negative[0] = 1$。

当 $i > 1$ 时，根据 $nums[i]$ 的值计算 $positive[i]$ 和 $negative[i]$ 的值。

- 当 $nums[i] > 0$ 时，之前的乘积乘以 $nums[i]$ 不会改变乘积的正负性。
  $positive[i]$ 的计算为：

  $$positive[i] = positive[i-1] + 1$$

  $negative[i]$ 的计算为：

  $$negative[i] = \begin{cases} negative[i-1] + 1, & negative[i-1] > 0 \\ 0, & negative[i-1] = 0 \end{cases}$$

  这是因为当 $negative[i-1] = 0$ 时，$negative[i]$ 本身无法形成一个乘积为正数的子数组，所以要特殊判断。
- 当 $nums[i] < 0$ 时，之前的乘积乘以 $nums[i]$ 会改变乘积的正负性。
  $positive[i]$ 的计算为：

  $$positive[i] = \begin{cases} negative[i-1] + 1, & negative[i-1] > 0 \\ 0, & negative[i-1] = 0 \end{cases}$$

  这是因为当 $negative[i-1] = 0$ 时，$positive[i]$ 本身无法形成一个乘积为负数的子数组，所以要特殊判断。
  $negative[i]$ 的计算为：

  $$negative[i] = positive[i-1] + 1$$

- 当 $nums[i] = 0$ 时，以下标 i 结尾的子数组的元素乘积一定为 0，因此有 $positive[i] = 0$ 和 $negative[i] = 0$

```c
int getMaxLen(int* nums, int numsSize){
    int positive = (nums[0] > 0);
    int negative = (nums[0] < 0);
    int maxLength = positive;
    for(int i = 1;i < numsSize;i++)
    {
        if(nums[i]>0)
        {
            positive++;
            negative = (negative > 0 ? negative + 1 : 0);
        }
        else if(nums[i]<0)
        {
            int newPositive = negative > 0 ? negative + 1 : 0;
            int newNegative = positive + 1;
            positive = newPositive;
            negative = newNegative;
        }
        else
        {
            positive = negative = 0;
        }
        maxLength = fmax(maxLength,positive);
    }
    return maxLength;
```

```
}
```

最佳观光组合

- 题目内容：给你一个正整数数组 values，其中 values[i] 表示第 i 个观光景点的评分，并且两个景点 i 和 j 之间的 距离 为 j - i。

  一对景点（i < j）组成的观光组合的得分为 values[i] + values[j] + i - j ，也就是景点的评分之和 减去 它们两者之间的距离。

  返回一对观光景点能取得的最高分。

- 为什么不能遍历两次是因为有可能观光点会重复。

```
int maxScoreSightseeingPair(int* values, int valuesSize){
    int ans = 0, mx = values[0] + 0;
    for(int j = 1;j < valuesSize;j++)
    {
        ans = fmax(ans,mx + values[j] - j);
        mx = fmax(mx , values[j] + j);
    }
    return ans;
}
```

最长上升子序列LIS系列

最长上升子序列

方法一：动态规划

思路与算法

定义 `dp[i]` 为考虑前 `i` 个元素，以第 `i` 个数字结尾的最长上升子序列的长度，**注意** `nums[i]` **必须被选取。**

我们从小到大计算 `dp[]` 数组的值，在计算 `dp[i]` 之前，我们已经计算出 `dp[0…i−1]` 的值

则**状态转移方程**为：

`dp[i]=max(dp[j])+1,其中0≤j<i且num[j]<num[i]`

即考虑往 `dp[0…i−1]` 中最长的上升子序列后面再加一个 `nums[i]` 。由于 `dp[j]` 代表 `nums[0…j]` 中以 `nums[j]` 结尾的最长上升子序列，所以如果能从 `dp[j]` 这个状态转移过来，那么 `nums[i]` 必然要大于 `nums[j]` ，才能将 `nums[i]` 放在 `nums[j]` 后面以形成更长的上升子序列。

最后，整个数组的最长上升子序列即所有 `dp[i]` 中的最大值。

`length =max(dp[i]),其中0≤i<n`

```
int lengthOfLIS(int* nums, int numsSize){
    int maxp = 0;
    int dp[numsSize+1];
    for(int i = 0;i<numsSize;i++)
    {
        dp[i] = 1;
        int max = 0;
        for(int j = 0;j<i;j++)
```

```
        {
            if(nums[j]<nums[i])
            {
                max = fmax(max,dp[j]);
            }
        }
        dp[i] = max + 1;
        maxp = fmax(maxp,dp[i]);
    }
    return maxp;
}
```

方法二：贪心 + 二分查找

```
int lengthOfLIS(vector<int>& nums)
{
    int len = 1, n = (int)nums.size();
    if (n == 0) {
        return 0;
    }
    vector<int> d(n + 1, 0);
    d[len] = nums[0];
    for (int i = 1; i < n; ++i) {
        if (nums[i] > d[len]) {
            d[++len] = nums[i];
        } else {
            int l = 1, r = len, pos = 0; // 如果找不到说明所有的数都比 nums[i] 大，此时要更新 d[1]，所以这里将 pos 设为 0
            while (l <= r) {
                int mid = (l + r) >> 1;
                if (d[mid] < nums[i]) {
                    pos = mid;
                    l = mid + 1;
                } else {
                    r = mid - 1;
                }
            }
            d[pos + 1] = nums[i];
        }
    }
    return len;
}
```

最长上升子序列的个数

- 假设对于以 `nums[i]` 结尾的序列，我们知道最长序列的长度 `length[i]`，以及具有该长度的序列的 `count[i]`。
- 对于每一个 `i<j` 和一个 `A[i]<A[j]`，我们可以将一个 A[j] 附加到以 A[i] 结尾的最长子序列上。
- 如果这些序列比 `length[j]` 长，那么我们就知道我们有 `count[i]` 个长度为 `length` 的序列。如果这些序列的长度与 `length[j]` 相等，那么我们就知道现在有 `count[i]` 个额外的序列（即 `count[j]+=count[i]`）。

```
int findNumberOfLIS(int* nums, int numsSize){
    int len[numsSize];
    int cnt[numsSize];
    int ans = 0;
```

```
    int longest = 0;
    for(int i = 0;i < numsSize; i++)
    {
        len[i] = 1;
        cnt[i] = 1;
        for(int j = 0; j < i; j++)
        {
            if(nums[j]<nums[i])
            {
                if(len[j]>=len[i])
                {
                    len[i] = len[j] + 1;
                    cnt[i] = cnt[j];
                }
                else if(len[j] + 1 == len[i])
                {
                    cnt[i]+=cnt[j];
                }
            }
        }
    }
    for(int i = 0; i < numsSize; i++)
    {
        longest = fmax(longest,len[i]);
    }
    for(int i = 0 ; i < numsSize ; i++)
    {
        if(len[i] == longest)
            ans+=cnt[i];
    }
    return ans;
}
```

二维LIS（套娃问题）

股票问题

一种常用的方法是将「买入」和「卖出」分开进行考虑：「买入」为负收益，而「卖出」为正收益。在初入股市时，你只有「买入」的权利，只能获得负收益。而当你「买入」之后，你就有了「卖出」的权利，可以获得正收益。

我们用 f[i]表示**第 i天结束之后**的「累计最大收益」。

- 一般以f[n][0]表示手上有股票；
- 一般以f[n][1]表示手上没有股票；
- 其他的再多用一个维度

之后找到状态转移方程即可。

买卖股票的最佳时机

```
int maxSubArray(int* nums, int numsSize){
    int max=nums[0],pre = 0;
    for(int i=0;i<numsSize;i++)
    {
        if(pre+nums[i]>nums[i])
            pre = pre+nums[i];
        else
            pre = nums[i];
        if(pre>max)
            max=pre;
    }
    return max;
}
int maxProfit(int* prices, int pricesSize){
    int pre = 0,now = 0;
    for(int i=0;i<pricesSize;i++)
    {
        if(i==0)
        {
            pre = prices[0];
            prices[0] = 0;
        }
        else
        {
            now = prices[i];
            prices[i] = prices[i] - pre;
            pre = now;
        }
    }
    return maxSubArray(prices,pricesSize);
}
```

最佳买卖股票时机含冷冻期

```
int maxProfit(int* prices, int pricesSize){
    if(pricesSize == 0) return 0;
    int f0 = -prices[0];
    int f1 = 0;
    int f2 = 0;
    for(int i = 1;i < pricesSize;i++)
    {
        int newf0 = fmax(f0,f2-prices[i]);
        int newf1 = f0 + prices[i];
        int newf2 = fmax(f1,f2);
        f0 = newf0;
        f1 = newf1;
        f2 = newf2;
    };
    return fmax(f1,f2);
}
```

买卖股票的最佳时机含手续费

```
int maxProfit(int* prices, int pricesSize, int fee){
    int f0 = -prices[0];
    int f1 = 0;
    for(int i = 1;i < pricesSize;i++)
    {
        int newf0 = fmax(f0,f1-prices[i]);
        int newf1 = fmax(f0+prices[i]-fee,f1);
        f0 = newf0;
        f1 = newf1;
    }
    return fmax(f0,f1);
}
```

买卖股票的最佳时机 (k次交易)

```
int maxProfit(int k, int* prices, int pricesSize) {
    int n = pricesSize;
    if (n == 0) {
        return 0;
    }

    k = fmin(k, n / 2);
    int buy[k + 1], sell[k + 1];
    memset(buy, 0, sizeof(buy));
    memset(sell, 0, sizeof(sell));

    buy[0] = -prices[0];
    sell[0] = 0;
    for (int i = 1; i <= k; ++i) {
        buy[i] = sell[i] = INT_MIN / 2;
    }

    for (int i = 1; i < n; ++i) {
        buy[0] = fmax(buy[0], sell[0] - prices[i]);
        for (int j = 1; j <= k; ++j) {
            buy[j] = fmax(buy[j], sell[j] - prices[i]);
            sell[j] = fmax(sell[j], buy[j - 1] + prices[i]);
        }
    }
    int ret = 0;
    for (int i = 0; i <= k; i++) {
        ret = fmax(ret, sell[i]);
    }

    return ret;
}
```

打家劫舍

打家劫舍1(直线)

状态转移方程：

$$dp[i] = max(dp[i-2] + nums[i], dp[i-1])$$

注意边界条件：

$$\begin{cases} dp[0] = nums[0] & 只有一间房屋，则偷窃该房屋 \\ dp[1] = max(nums[0], nums[1]) & 只有两间房屋，选择其中金额较高的房屋进行偷窃 \end{cases}$$

```
int rob(int* nums, int numsSize){
    if(numsSize==1)
        return nums[0];
    else if(numsSize==2)
        return fmax(nums[0],nums[1]);
    int pre = nums[0];
    int cur = fmax(nums[0],nums[1]);
    for(int i=2;i<numsSize;i++)
    {
        int now = fmax(pre + nums[i],cur);
        pre = cur;
        cur = now;
    }
    return cur;
}
```

打家劫舍2(环)

```
int robRange(int* nums, int start ,int end){
    int pre = nums[start];
    int cur = fmax(nums[start],nums[start+1]);
    for(int i=2+start;i<end;i++)
    {
        int now = fmax(pre + nums[i],cur);
        pre = cur;
        cur = now;
    }
    return cur;
}
int rob(int* nums, int numsSize){
    if(numsSize==1)
        return nums[0];
    else if(numsSize==2)
        return fmax(nums[0],nums[1]);
    else return fmax(robRange(nums,0,numsSize-1),robRange(nums,1,numsSize));
}
```

前缀和

一维前缀和

二维前缀和

- 预备知识：设二维数组 A 的大小为 m * n，行下标的范围为 [1, m]，列下标的范围为 [1, n]。

  数组 P 是 A 的前缀和数组，等价于 P 中的每个元素 P[i][j]：

  如果 i 和 j 均大于 0，那么 P[i][j] 表示 A 中以 (1, 1) 为左上角，(i, j) 为右下角的矩形区域的元素之和；

  如果 i 和 j 中至少有一个等于 0，那么`P[i][j]`也等于 0。

  数组 P 可以帮助我们在 O(1)的时间内求出任意一个矩形区域的元素之和。具体地，设我们需要求和的矩形区域的左上角为 (x1, y1)，右下角为 (x2, y2)，则该矩形区域的元素之和可以表示为：

  `sum = A[x1..x2][y1..y2]= P[x2][y2] - P[x1 - 1][y2] - P[x2][y1 - 1] + P[x1 - 1][y1 - 1]`

  `P[i][j] = P[i - 1][j] + P[i][j - 1] - P[i - 1][j - 1] + A[i][j]`

- 二维前缀和的常用算法

```
void getPrefixSum()//获得前缀和
{
    for(i=1;i<=m;i++)
    {
        for(j=1;j<=n;j++)
        {
            p[i][j] = p[i-1][j] + p[i][j-1] - p[i-1][j-1] + mat[i][j];
        }
    }
}
int getRect(int x1,int y1,int x2,int y2)//计算矩形的值
{
    return p[x2][y2] - p[x1 - 1][y2] - p[x2][y1 - 1] + p[x1 - 1][y1 - 1];
}
```

区间DP

1. `dp[i][j]`仅与**常数个更小规模子问题**有关(回文子串等)
   一般是与`dp[i + 1][j], dp[i][j - 1], dp[i + 1][j - 1]`有关。

`dp[i][j] = f(dp[i + 1][j], dp[i][j - 1], dp[i + 1][j - 1])`

代码常见写法:

```
for len = 1..n
    for i = i..len
        j = i + len - 1
        dp[i][j] = max(dp[i][j], f(dp[i+1][j], dp[i][j-1], dp[i+1][j-1]))
```

2. `dp[i][j]`与 **O(n) 个更小规模子问题**有关
   一般是枚举 [i,j] 的分割点，将区间分为 [i,k] 和 [k+1,j]，对每个 k 分别求解（下面公式的 f），再汇总（下面公式的 g）。

`dp[i][j] = g(f(dp[i][k], dp[k + 1][j])) 其中 k = i .. j-1。`

代码常见写法, 以下代码以 f 为 max 为例:

```
for len = 1..n
    for i = i..len
        j = i + len - 1
        for k = i..j
            dp[i][j] = max(dp[i][j], f(dp[i][k], dp[k][j]))
```

常数规模子问题

最长回文子串

方法一：区间dp

根据这样的思路，我们就可以用动态规划的方法解决本题。我们用 $P(i,j)$ 表示字符串 s 的第 i 到 j 个字母组成的串（下文表示成 $s[i:j]$）是否为回文串：

$$P(i,j)=\begin{cases}\text{true}, & \text{如果子串 } S_i \dots S_j \text{ 是回文串}\\ \text{false}, & \text{其它情况}\end{cases}$$

这里的「其它情况」包含两种可能性：

- $s[i,j]$ 本身不是一个回文串；
- $i > j$，此时 $s[i,j]$ 本身不合法。

那么我们就可以写出动态规划的状态转移方程：

$$P(i,j)=P(i+1,j-1)\wedge(S_i==S_j)$$

也就是说，只有 $s[i+1:j-1]$ 是回文串，并且 s 的第 i 和 j 个字母相同时，$s[i:j]$ 才会是回文串。

上文的所有讨论是建立在子串长度大于 2 的前提之上的，我们还需要考虑动态规划中的边界条件，即子串的长度为 1 或 2。对于长度为 1 的子串，它显然是个回文串；对于长度为 2 的子串，只要它的两个字母相同，它就是一个回文串。因此我们就可以写出动态规划的边界条件：

$$\begin{cases}P(i,i)=\text{true}\\ P(i,i+1)=(S_i==S_{i+1})\end{cases}$$

根据这个思路，我们就可以完成动态规划了，最终的答案即为所有 $P(i,j)=\text{true}$ 中 $j-i+1$（即子串长度）的最大值。**注意：在状态转移方程中，我们是从长度较短的字符串向长度较长的字符串进行转移的，因此一定要注意动态规划的循环顺序。**

```
class Solution {
public:
    string longestPalindrome(string s) {
        int n = s.size();
        if (n < 2) {
            return s;
        }

        int maxLen = 1;
        int begin = 0;
        // dp[i][j] 表示 s[i..j] 是否是回文串
        vector<vector<int>> dp(n, vector<int>(n));
        // 初始化：所有长度为 1 的子串都是回文串
        for (int i = 0; i < n; i++) {
            dp[i][i] = true;
```

```
        }
        // 递推开始
        // 先枚举子串长度
        for (int L = 2; L <= n; L++) {
            // 枚举左边界，左边界的上限设置可以宽松一些
            for (int i = 0; i < n; i++) {
                // 由 L 和 i 可以确定右边界，即 j - i + 1 = L 得
                int j = L + i - 1;
                // 如果右边界越界，就可以退出当前循环
                if (j >= n) {
                    break;
                }

                if (s[i] != s[j]) {
                    dp[i][j] = false;
                } else {
                    if (j - i < 3) {
                        dp[i][j] = true;
                    } else {
                        dp[i][j] = dp[i + 1][j - 1];
                    }
                }

                // 只要 dp[i][L] == true 成立，就表示子串 s[i..L] 是回文，此时记录回文长
度和起始位置
                if (dp[i][j] && j - i + 1 > maxLen) {
                    maxLen = j - i + 1;
                    begin = i;
                }
            }
        }
        return s.substr(begin, maxLen);
    }
};
```

方法二：中心拓展算法

```
struct Palindrome{
    int start;
    int end;
};

char * longestPalindrome(char * s){
    struct Palindrome temp,max;//先声明两个变量用于存储当前回文串和最优回文串
    int i;
    int lenth=strlen(s);
    max.start=max.end=0;//最优回文串初始化为0
    for(i=0;i<lenth;)
    {
        temp.start=temp.end=i;
        while(s[temp.start]==s[temp.end+1])//先找相同的字符作为回文串的中心主体
            (temp.end)++;
        i=temp.end+1;
        while((temp.start-1)>=0&&(temp.end+1)<lenth)//两边延伸
        {
```

```
            if(s[temp.start-1]==s[temp.end+1])
            {
                (temp.start)--;
                (temp.end)++;
            }
            else
                break;
        }
        if(temp.end-temp.start>max.end-max.start)
            max=temp;
    }
    s[max.end+1]='\0';
    return &s[max.start];
}
```

最长回文子序列

- 给你一个字符串 s，找出其中最长的回文子序列，并返回该序列的长度。

  子序列定义为：不改变剩余字符顺序的情况下，删除某些字符或者不删除任何字符形成的一个序列。

  > 对于一个子序列而言，如果它是回文子序列，并且长度大于 2，那么将它首尾的两个字符去除之后，它仍然是个回文子序列。因此可以用动态规划的方法计算给定字符串的最长回文子序列。
  >
  > 用 $dp[i][j]$ 表示字符串 s 的下标范围 $[i,j]$ 内的最长回文子序列的长度。假设字符串 s 的长度为 n，则只有当 $0 \le i \le j < n$ 时，才会有 $dp[i][j] > 0$，否则 $dp[i][j] = 0$。
  >
  > 由于任何长度为 1 的子序列都是回文子序列，因此动态规划的边界情况是，对任意 $0 \le i < n$，都有 $dp[i][i] = 1$。
  >
  > 当 $i < j$ 时，计算 $dp[i][j]$ 需要分别考虑 $s[i]$ 和 $s[j]$ 相等和不相等的情况：
  >
  > - 如果 $s[i] = s[j]$，则首先得到 s 的下标范围 $[i+1, j-1]$ 内的最长回文子序列，然后在该子序列的首尾分别添加 $s[i]$ 和 $s[j]$，即可得到 s 的下标范围 $[i,j]$ 内的最长回文子序列，因此 $dp[i][j] = dp[i+1][j-1] + 2$；
  > - 如果 $s[i] \ne s[j]$，则 $s[i]$ 和 $s[j]$ 不可能同时作为同一个回文子序列的首尾，因此 $dp[i][j] = \max(dp[i+1][j], dp[i][j-1])$。
  >
  > 由于状态转移方程都是从长度较短的子序列向长度较长的子序列转移，因此需要注意动态规划的循环顺序。
  >
  > 最终得到 $dp[0][n-1]$ 即为字符串 s 的最长回文子序列的长度。

```
int longestPalindromeSubseq(char* s) {
    int n = strlen(s);
    int dp[n][n];
    memset(dp, 0, sizeof(dp));
    for (int i = n - 1; i >= 0; i--) {
        dp[i][i] = 1;
        char c1 = s[i];
        for (int j = i + 1; j < n; j++) {
            char c2 = s[j];
            if (c1 == c2) {
                dp[i][j] = dp[i + 1][j - 1] + 2;
            } else {
                dp[i][j] = fmax(dp[i + 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[0][n - 1];
```

```
}
```

```
int longestPalindromeSubseq(char * s){
    int n = strlen(s);
    int dp[n][n];
    memset(dp,0,sizeof(dp));
    for(int i=0;i<n;i++)
        dp[i][i] = 1;
    for(int l = 2;l<=n;l++)
    {
        for(int i = 0;i<n;i++)
        {
            int j = l + i - 1;
            if(i>j || j>=n) break;
            if(s[i]==s[j])
            {
                dp[i][j] = dp[i+1][j-1] + 2;
            }
            else
                dp[i][j] = fmax(dp[i+1][j],dp[i][j-1]);
        }
    }
    return dp[0][n-1];
}
```

最长公共子序列（LCS）

Longest Common Subsequence

暗号

状态转移方程：

$$\begin{cases} dp[i][j] = dp[i-1][j-1] + 1 & A[i] == B[i] \\ dp[i][j] = max(dp[i-1][j], dp[i][j-1]) & A[i] != B[i] \end{cases}$$

```
#include<iostream>
#include<string>
#include<cstring>
#include<algorithm>
#include<set>
using namespace std;
char A[110],B[110];
int dp[110][110];
int n,m;
int targetlen;
set<string> ans;
void init()
{
    memset(dp,0,sizeof(dp));
    ans.clear();
}
void LCS()
{
    //preMark = 0; nowMark = 1;
```

```
    for(int i = 1;i<=n;i++)
    {
        for(int j = 1;j<=m;j++)
        {
            /*dp[i][j] = max(dp[i-1][j],dp[i][j-1]);
            if(A[i] == B[j])
            {
                dp[i][j] = max(dp[i][j],dp[i-1][j-1] + 1);
            }*/
            if(A[i] == B[j])
            {
                dp[i][j] = dp[i-1][j-1] + 1;
            }
            else
            {
                dp[i][j] = max(dp[i-1][j],dp[i][j-1]);
            }
             //nowMark = preMark; preMark = 1 - preMark;
            //更新，当然了，最后结果是dp[preMark][m]
        }
    }
    targetlen = dp[n][m];
}
void dfs(int i, int j, string s)
{
    if(i<=0 || j<=0)    return ;
    if(A[i] == B[j])
    {
        s.push_back(A[i]);
        if(s.length() == targetlen)
        {
            reverse(s.begin(),s.end());
            ans.insert(s);
        }
        else   dfs(i-1,j-1,s);
    }
    else{
        if(dp[i-1][j] >= dp[i][j-1])   dfs(i-1,j,s);
        if(dp[i-1][j] <= dp[i][j-1])    dfs(i,j-1,s);
    }
}
int main(){
    while(~scanf("%s%s",A+1,B+1))
    {
        init();
        n = strlen(A+1);
        m = strlen(B+1);
        LCS();
        dfs(n,m,"");
        for(string s : ans)
        {
            puts(s.c_str());
        }
    }
    return 0;
}
```

- 滚动数组空间优化

```
int longestCommonSubsequence(char * text1, char * text2){
    int n = strlen(text1),m = strlen(text2);
    int dp[m+1];
    memset(dp,0,sizeof(dp));
    for(int i = 1;i<=n;i++)
    {
        int upleft = dp[0];
        for(int j=1;j<=m;j++)
        {
            int tmp = dp[j];
            if(text1[i-1] == text2[j-1])
                dp[j] = upleft +1;
            else
                dp[j] = fmax(dp[j-1],dp[j]);
            upleft = tmp;
        }
    }
    return dp[m];
}
```

O(n)规模子问题

矩阵链乘

零崎的朋友很多Ⅲ

```
#include <iostream>
#define maxn 310
#define INF 0x3f3f3f3f
using namespace std;
int m[maxn][maxn],s[maxn][maxn];
int a[maxn],n;
void Matrixchain()
{
    for(int l=2;l<=n;l++) //l是矩阵链长度
        for(int i=0;i<=n-l+1;i++)   //i是矩阵链最左边(长度限制在最总长度减去矩阵链长度之内)
        {
            int j = i+l-1;  //j是矩阵链最右边 (l=j-i+1)
            m[i][j] = INF;
            for(int k = i;k<j;k++)  //k是i和j中间的分界点 (任意一点)
            {
                int tmp = m[i][k] + m[k+1][j] + a[i-1] * a[k] * a[j];   //相乘代价表达式
                if(tmp<=m[i][j])    m[i][j] = tmp,s[i][j] = k;  //更新数据
            }
        }
}
void print_Matrix(int i,int j)
{
    if(i==j) printf("A%d",i);
    else
    {
```

```
        putchar('(');
        print_Matrix(i,s[i][j]);
        print_Matrix(s[i][j]+1,j);
        putchar(')');
    }
}
int main()
{
    while(~scanf("%d",&n))
    {
        for(int i=0;i<=n;i++)
            scanf("%d",a+i);
        for(int i=0;i<=n;i++)
            m[i][i] = 0;
        Matrixchain();
        printf("%d\n", m[1][n]);
        print_Matrix(1, n);
        putchar('\n');
    }
}
```

最优二叉搜索树

最优二叉搜索树

C3-Zexal的OBST

最优二叉搜索树的搜索期望:
n个关键字的搜索概率为p1到pn
叶子节点n+1个，概率q0到qn

每个节点的期望是(搜索深度+1)*概率

最优子结构:
设dp[i][j]为包含ki...kj关键字树的搜索代价
设这几个关键字组成的树以kr为根，则左右子树为ki,ki+1,...,kr-1和kr+1...kj对应叶子分别为di-1...dr-1和dr...dj-1，只有一个叶子组成的子树j=i-1只有di-1则e[i][j]=qi-1 这个子树为一个叶节点，不再划分左右树还有关键字的话就接着划分左右子树i<=je[i][j]=min{e[i][r-1]+e[r+1][j]+w[i][j]}

状态转移方程:

$$e[i][j] = \begin{cases} q[i-1] & j == i-1 \\ min(e[i][r-1] + e[r+1][j] + w[i][j]) & i <= j \end{cases}$$

$$w[i][j] = w[i][j-1] + p[j] + q[j]$$

```
#include<iostream>
#include<cstring>
#define ms(a,b) memset(a,b,sizeof(a))
#define maxn 510
using namespace std;
double e[maxn][maxn], w[maxn][maxn], p[maxn], q[maxn];
int n;
void init()
{
    ms(p,0.0);
    ms(q,0.0);
    ms(e,0.0);
```

```
    ms(w,0.0);
}
void OBST()
{
    for(int i=1;i<=n+1;i++)
        e[i][i-1] = w[i][i-1] = q[i-1];
    for(int k=1;k<=n;k++)
    {
        for(int i=1;i<=n-k+1;i++)
        {
            int j = i+k-1;
            e[i][j] = 0x3f3f;
            w[i][j] = w[i][j-1] + p[j] +q[j];
            for(int r = i;r<=j;r++)
            {
                double t = e[i][r-1] + e[r+1][j] + w[i][j];
                if(t < e[i][j]) e[i][j] = t;
            }
        }
    }
}
int main(){
    while(~scanf("%d",&n))
    {
        init();
        for(int i = 1;i<=n;i++) scanf("%lf",&p[i]);
        for(int i = 0;i<=n;i++) scanf("%lf",&q[i]);
        OBST();
        printf("%.3lf\n",e[1][n]);
    }
    return 0;
}
```

凸多边形的划分

划分多边形

那么对于区间类动态规划问题，往往可以将问题分解成为两两合并的形式。其解决方法是对整个问题设最优解，枚举分割点，维护最优值。
`dp[i][j] = max{dp[i][k] + dp[k+1][j] + 合并时需要计算的值 }`,其中k为区间[i,j]内一分割点。

对于多边形，我们将其顶点按顺时针编号，那么所求问题为1~n顺时针连线组成的多边形的三角形划分后的**权值乘积之和的最小值**。按照求什么设什么的思考原则，设`dp[i][j] (i < j)`**表示从i~j顺时针连线组成的多边形的三角形剖分后所得的顶点权值乘积和的最小值。（第一步）**

状态转移方程：

$$dp[i][j] = min(dp[i][k] + a[i] * a[j] * a[k] + dp[k][j])(1 <= i < k < j <= n);$$

边界条件：

`初始dp[i][i+1] = 0;目标状态在dp[1][n];`

```
#include <bits/stdc++.h>
#define mp make_pair
#define ms(a, b) memset(a, b, sizeof(a))
#define maxn 1007
typedef long long LL;
```

```
using namespace std;
inline LL read()
{
    LL X = 0, w = 1;
    char ch = 0;
    while (ch < '0' || ch > '9')
    {
        if (ch == '-')
            w = -1;
        ch = getchar();
    }
    while (ch >= '0' && ch <= '9')
        X = (X << 3) + (X << 1) + ch - '0', ch = getchar();
    return X * w;
}
inline void write(LL x)
{
    if (x < 0)
        putchar('-'), x = -x;
    if (x > 9)
        write(x / 10);
    putchar(x % 10 + 48);
}
LL n, dp[maxn][maxn], a[maxn];
int main()
{
    n = read();
    ms(dp,0x3f3f3f3f);
    for (LL i = 1; i <= n; i++)
        a[i] = read();
    for (int i = 1; i <= n; i++)
        dp[i][i + 1] = 0;
    for (int len = 2; len < n; len++)
    {
        for (int i = 1; i < n-len+1; i++)
        {
            int j = i+len;
            for (int k = i + 1; k < j; k++)
            {
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j] + a[i] * a[j] *
a[k]);
            }
        }
    }
    write(dp[1][n]);
    return 0;
}
```

背包问题

01背包

[零崎的补番计划II]

[Magry的朋友很多 - 零食篇]

0-1背包问题是指每一种物品都只有一件，可以选择放或者不放。现在假设有n件物品，背包承重为m。dp(n,m)就是我们的最终结果了。

01背包原理

在解决问题之前，为描述方便，首先定义一些变量：**Vi表示第 i 个物品的价值，Wi表示第 i 个物品的体积，定义dp(i,j)：当前背包容量 j，前 i 个物品最佳组合对应的价值**，同时背包问题抽象化（X1，X2，...，Xn，其中 Xi 取0或1，表示第 i 个物品选或不选）。

1. 建立模型，即求max(V1X1+V2X2+...+VnXn);
2. 寻找约束条件，W1X1+W2X2+...+WnXn<capacity;
3. 寻找递推关系式，面对当前商品有两种可能性：

- **包的容量比该商品体积小，装不下，此时的价值与前i-1个的价值是一样的，即dp(i,j)=dp(i-1,j);**
- **还有足够的容量可以装该商品，但装了也不一定达到当前最优价值，所以在装与不装之间选择最优的一个，即dp(i,j)=max｛dp(i-1,j)，dp(i-1,j-w(i))+v(i)｝。**

- **注意，dp里的j - w[i]必须是同类型的。**
- 二维

```
void findMax() {                    //动态规划
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (j < w[i])
                dp[i][j] = dp[i - 1][j];
            else
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i]] + v[i]);
        }
    }
}
```

- 一维

```
for (int i = 1; i <= n; i++) {
    for (int j = m; j >= w[i]; j--) {
         dp[j] = dp[j] > dp[j - w[i]] + v[i] ? dp[j] : dp[j - w[i]] + v[i];
    }
}
```

01背包最优解回溯

通过上面的方法可以求出背包问题的最优解，但还不知道这个最优解由哪些商品组成，故要根据最优解回溯找出解的组成，根据填表的原理可以有如下的寻解方式：

- V(i,j)=V(i-1,j)时，说明没有选择第i 个商品，则回到V(i-1,j)；
- V(i,j)=V(i-1,j-w(i))+v(i)时，说明装了第i个商品，该商品是最优解组成的一部分，随后我们得回到装该商品之前，即回到V(i-1,j-w(i))；
- 一直遍历到i = 0结束为止，所有解的组成都会找到。

```
void findwhat(int i, int j) {                //最优解情况
    if (i >= 0) {
        if (dp[i][j] == dp[i - 1][j]) {
            item[i] = 0;
            findwhat(i - 1, j);
        }
        else if (j - w[i] >= 0 && dp[i][j] == dp[i - 1][j - w[i]] + v[i]) {
            item[i] = 1;
            findwhat(i - 1, j - w[i]);
        }
    }
}
```

01背包是否恰好装满问题

[零崎的朋友很多II]

恰好装满：

1. 求最大值时，除了**dp[0] 为0**，其他都初始化为无穷小 -0x3f3f3f3f

   `dp[i] = max(dp[i],dp[i-weight]+value);`

2. 求最小值时，除了**dp[0] 为0**，其他都初始化为无穷大 0x3f3f3f3f

   `dp[i] = min(dp[i],dp[i-weight]+value);`

3. 不必恰好装满： 全初始化为0。

完全背包

[零崎的朋友很多 I]

- 一般状态转移方程：

  $$dp[i][j] = max(dp[i-1, j-kw_i] + kv[i])0 \leq kw_i \leq j$$

- 采用二进制思想之后新的转移方程：

  $$dp[i][j] = max(dp[i-1][j], dp[i][j-w_i] + v_i$$

```
for (int i = 1; i <= n; i++) {
        for (int j = w[i]; j <= m; j++) {
            dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
        }
    }
```

多重背包

DP大作战—多重背包

- 状态转移方程为：

  $dp[i][j] = max(dp[i-1][j-k*w[i]] + v[i])0 <= k <= n[i]$

多重背包问题限定了一种物品的个数，解决多重背包问题，只需要把它转化为0-1背包问题即可。比如，有2件价值为5，重量为2的同一物品，我们就可以分为物品a和物品b，a和b的价值都为5，重量都为2，但我们把它们视作不同的物品。

```
int k = n + 1;
    for (int i = 1; i <= n; i++) {
        while (num[i] != 1) {
            w[k] = w[i];
            v[k] = v[i];
            k++;
            num[i]--;
        }
    }
for (int i = 1; i <= k; i++) {
        for (int j = m; j >= 1; j--) {
            if (w[i] <= j) dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
        }
    }
```

三种背包函数形式(组合背包)

```
int dp[MAX];//存储最后背包最大能存多少
int value[MAX],weight[MAX],number[MAX];//分别存的是物品的价值，每一个的重量以及数量
int bag;

void ZeroOnePack(int weight,int value )//01背包
{
    int i;
    for(i = bag; i>=weight; i--)
    {
        dp[i] = max(dp[i],dp[i-weight]+value);
    }
}
void CompletePack(int weight,int value)//完全背包
{
    int i;
    for(i = weight; i<=bag; i++)
    {
        dp[i] = max(dp[i],dp[i-weight]+value);
    }
```

```
}

void MultiplePack(int weight,int value,int number)//多重背包
{
    if(bag<=number*weight)//如果总容量比这个物品的容量要小，那么这个物品可以直到取完，相当于完全背包
    {
        CompletePack(weight,value);
        return ;
    }
    else//否则就将多重背包转化为01背包
    {
        int k = 1;
        while(k<=number)
        {
            ZeroOnePack(k*weight,k*value);
            number = number-k;
            k = 2*k;//这里采用二进制思想
        }
        ZeroOnePack(number*weight,number*value);
    }
}
```

树形DP

对树根进行状态分析

树形DP初步-真树

状态表示：

1. `dp[u][0]`表示不选择结点u时，在以节点u为根的子树中选择的最大人数
2. `dp[u][1]`表示选择结点u时，在以节点u为根的子树中选择的最大人数

状态转移方程：

1. `dp[u][0]+=max(dp[v][0],dp[v][1])`
2. `dp[u][1]+=dp[v][0]`

考虑边界及其初始值：

1. `dp[u][0] = 0`
2. `dp[u][1] = 权值`

注意要点：

1. 要用并查集寻找真正的树根，所以需要`fa[maxn]`数组进行存储
2. 如果数据给的是具体的名字而不是现成的编号，我们可以采用`STL`里面的`map<string,int>`进行编码

```
#include <bits/stdc++.h>
#define mp make_pair
#define ms(a, b) memset(a, b, sizeof(a))
#define maxn 10007
typedef long long LL;
using namespace std;
```

```cpp
inline int read()
{
    int X = 0, w = 1;
    char ch = 0;
    while (ch < '0' || ch > '9')
    {
        if (ch == '-')
            w = -1;
        ch = getchar();
    }
    while (ch >= '0' && ch <= '9')
        X = (X << 3) + (X << 1) + ch - '0', ch = getchar();
    return X * w;
}
inline void write(int x)
{
    if (x < 0)
        putchar('-'), x = -x;
    if (x > 9)
        write(x / 10);
    putchar(x % 10 + 48);
}
int n;
int happy[maxn], fa[maxn];
int dp[maxn][2];
vector<int> E[maxn];
void dfs(int u)
{
    dp[u][0] = 0;
    dp[u][1] = happy[u];
    for (int i = 0; i < E[u].size(); i++)
    {
        int v = E[u][i];
        dfs(v);
        dp[u][0] += max(dp[v][0], dp[v][1]);
        dp[u][1] += dp[v][0];
    }
}
int main()
{
    ms(fa, -1);
    n = read();
    for (int i = 1; i <= n; i++)
    {
        happy[i] = read();
    }
    while (1)
    {
        int a = read(), b = read();
        if (a == 0 && b == 0)
            break;
        else
        {
            E[b].push_back(a);
            fa[a] = b;
        }
    }
    int rt = 1;
```

```
    while (fa[rt] != -1)
        rt = fa[rt];

    dfs(rt);
    write(max(dp[rt][0],dp[rt][1]));
}
```

其他问题

矩阵类型

流水线问题

[说好的ALS呢?](#)

- 状态状态转移方程:

  $dp[i][j] = min(dp[i][j], dp[k][j-1] + t[k][i] + p[i][j]);$

```
#include<iostream>
#include<cstring>
#include<algorithm>
#define ms(a,b) memset(a,b,sizeof(a))
typedef long long LL;
using namespace std;
inline LL read()
{
    LL X=0,w=1; char ch=0;
    while(ch<'0' || ch>'9') {if(ch=='-') w=-1;ch=getchar();}
    while(ch>='0' && ch<='9') X=(X<<3)+(X<<1)+ch-'0',ch=getchar();
    return X*w;
}
inline void write(LL x) {
    if(x < 0)putchar('-'),x=-x;
    if (x > 9)write(x / 10);
    putchar(x % 10 + 48);
}
LL n,m,i,j,k;
LL p[101][101],t[101][101],dp[101][101];
int main(){
    while(~scanf("%lld %lld",&n,&m))
    {
        ms(p,0);
        ms(t,0);
        ms(dp,0x3f3f3f3f);//一定要这么大
        for(i=1;i<=n;i++)
        {
            for(j=1;j<=m;j++)
            {
                p[i][j] = read();
                dp[i][1] = p[i][1];
            }
        }
        for(i=1;i<=n;i++)
        {
            for(j=1;j<=n;j++)
```

```
            {
                t[i][j] = read();
            }
        }


        for(i=2;i<=m;i++)
        {
            for(j=1;j<=n;j++)
            {
                for(k=1;k<=n;k++)
                {
                    dp[j][i] = min(dp[j][i],dp[k][i-1] + t[k][j] +p[j][i]);
                }
            }
        }
        LL mindp = 0x3f3f3f3f;//一定要这么大
        for(i=1;i<=n;i++)
        {
            mindp = min(dp[i][m],mindp);
        }
        write(mindp);
        putchar('\n');
    }
    return 0;
}
```

不同路径

一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为“Start” ）。

机器人每次**只能向下或者向右移动一步**。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

现在考虑网格中**有障碍物**。那么从左上角到右下角将会有多少条不同的路径?

- 主要是学习二维数组怎么滚动至一维。

$$f(i,j) = \begin{cases} 0, & u(i,j) = 0 \\ f(i-1,j) + f(i,j-1), & u(i,j) \neq 0 \end{cases}$$

```
for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            if (obstacleGrid[i][j] == 1) {
                f[j] = 0;
                continue;
            }
            if (j - 1 >= 0 && obstacleGrid[i][j - 1] == 0) {
                f[j] += f[j - 1];
            }
        }
    }
```

丑数 II

- 给你一个整数 `n` ，请你找出并返回第 `n` 个 **丑数** 。

  丑数 就是只包含质因数 `2`、`3` 和/或 `5` 的正整数。

- 对于质因数这类的问题，最好采用从头开始分别乘以这些质因数，这样就可以简单而又轻松地得到只包含这些质因数的数字。因此采用多指针的算法技巧很方便，

```
int nthUglyNumber(int n){
    int dp[n+1];
    dp[1] = 1;
    int p2 = 1,p3 = 1,p5 = 1;
    for(int i = 2;i<=n;i++)
    {
        int num2 = dp[p2]*2, num3 = dp[p3]*3, num5 = dp[p5]*5;
        dp[i] = fmin(fmin(num2,num3),num5);
        if(dp[i] == num2)    p2++;
        if(dp[i] == num3)  p3++;
        if(dp[i] == num5)  p5++;
    }
    return dp[n];
}
```

双调欧几里得TSP问题

身可死，武士之名不可弃