算法强化

我们称一个长度为n的序列为正则序列，当且仅当该序列是一个由1~n组成的排列，即该序列由n个正整数组成，取值在[1,n]范围，且不存在重复的数，同时正则序列不要求排序

有一天小团得到了一个长度为n的任意序列s，他需要在有限次操作内，将这个序列变成一个正则序列，每次操作他可以任选序列中的一个数字，并将该数字加一或者减一。

请问他最少用多少次操作可以把这个序列变成正则序列?

进阶：时间复杂度O(nlogn)

示例：

```
输入：-1 2 3 10 100
输出：103
```

代码：

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    int n, ans = 0;;
    cin >> n;
    vector<int>que(n, 0);
    for (int i = 0; i < n; ++i) cin >> que[i];
    sort(que.begin(), que.end());
    for (int i = 1; i <= n; ++i) {
        ans += abs(i - que[i - 1]);
    }
    cout << ans;
    return 0;
}
```

分治法

1. 给你一个整数数组 nums ，其中元素已经按 升序 排列，请你将其转换为一棵 高度平衡 二叉搜索树。

高度平衡 二叉树是一棵满足「每个节点的左右两个子树的高度差的绝对值不超过 1 」的二叉树。

思路：选择中间位置左边的数字作为根节点，确定平衡二叉搜索树的根节点之后，其余的数字分别位于平衡二叉搜索树的左子树和右子树中，左子树和右子树分别也是平衡二叉搜索树，因此可以通过递归的方式创建平衡二叉搜索树。

代码：

```
BTNode *helper(int nums[], int left, int right)
{
    if (left > right)
    {
        return NULL;
    }
    // 总是选择中间位置左边的数字作为根节点
    int mid = (left + right) / 2;

    BTNode *root = (BTNode *)malloc(sizeof(BTNode));
    root->data = nums[mid];
    root->lchild = helper(nums, left, mid - 1);
    root->rchild = helper(nums, mid + 1, right);
    return root;
}
```

2. 输入一个长度为n的整型数组array，数组中的一个或连续多个整数组成一个子数组，子数组最小长度为1。求所有子数组的和的最大值， 当所有整数均为负数时，定义最大字段和为0，要求用分治法求解

代码：

```
int MaxSubSum(int array[], int left, int right)
{
    int sum = 0;
    if (left == right)
        sum = array[left] > 0 ? array[left] : 0;
    else
    {
        int mid = (left + right) / 2;
        int leftsum = MaxSubSum(array, left, mid);
        int rightsum = MaxSubSum(array, mid + 1, right);
        int s1 = 0, lefts = 0, i;
        for (i = mid; i >= left; i--)
        {
            lefts += array[i];
            if (lefts > s1)
                s1 = lefts;
        }
        int s2 = 0, rights = 0;
        for (i = mid + 1; i <= right; i++)
        {
            rights += array[i];
            if (rights > s2)
                s2 = rights;
        }
        sum = s1 + s2;
        if (sum < leftsum)
            sum = leftsum;
        if (sum < rightsum)
            sum = rightsum;
        return sum;
    }
```

```
}
```

3. 编写一个高效的算法来搜索 m x n 矩阵 matrix 中的一个目标值 target 。该矩阵具有以下特性：

   每行的元素从左到右升序排列。 每列的元素从上到下升序排列。

实例：

```
输入：
[[1,2,3],[4,5,6]],2,3,6
返回值：[1,2]
```

思路：由于矩阵中每一行的元素都是升序排列的，因此我们可以对每一行都使用一次二分查找，判断 target 是否在该行中，从而判断target 是否出现。

代码：

```
bool searchMatrix(int matrix[][MAXSIZE], int n, int target)
{
    int mid, low, high;
    for (int i = 0; i < n; i++)
    {
        low = 0;
        high = MAXSIZE - 1;
        while (low <= high)
        {
            mid = (low + high) / 2;
            if (matrix[i][mid] == target)
                return true;
            else if (matrix[i][mid] > target)
                high = mid - 1;
            else
                low = mid + 1;
        }
    }
    return false;
}
```

4. 在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组,求出这个数组中的逆序对的总数P，题目保证输入的数组中没有的相同的数字

   要求：空间复杂度 O(n)，时间复杂度 O(nlogn)

   示例：

```
输入：[1,2,3,4,5,6,7,0]
返回值：7
```

思路：

因为我们在归并排序过程中会将数组划分成最小为1个元素的子数组，然后依次比较子数组的每个元素的大小，依次取出较小的一个合并成大的子数组。

具体做法：

- step 1： 划分阶段：将待划分区间从中点划分成两部分，两部分进入递归继续划分，直到子数组长度为1.
- step 2： 排序阶段：使用归并排序递归地处理子序列，同时统计逆序对，因为在归并排序中，我们会依次比较相邻两组子数组各个元素的大小，并累计遇到的逆序情况。而对排好序的两组，右边大于左边时，它大于了左边的所有子序列，基于这个性质我们可以不用每次加1来统计，减少运算次数。
- step 3： 合并阶段：将排好序的子序列合并，同时累加逆序对。

代码：

```
int mergeSort(vector<int> &data, int low, int high)
{
    if (low >= high)
        return 0; //递归出口
    int mid = (low + high) / 2, len = high - low + 1;
    int lsum = mergeSort(data, low, mid);      // 左半边逆序对
    int rsum = mergeSort(data, mid + 1, high); // 右半边逆序对
    int res = lsum + rsum;
    int *temp = (int *)malloc(sizeof(int) * len);
    int i = low, j = mid + 1, k = 0;
    while (i <= mid && j <= high)
    {
        if (data[i] < data[j])
            temp[k++] = data[i++];
        else
        {
            // 左边比右边大，统计逆序对
            temp[k++] = data[j++];
            res += mid - i + 1;
        }
    }
    while (i <= mid)
        temp[k++] = data[i++];
    while (j <= high)
        temp[k++] = data[j++];
    i = low;
    j = 0;
    while (j < k) // 复制回原数组
        data[i++] = temp[j++];
    return res;
}
```

动态规划

1. 给定一个整数数组 cost ， 其中 cost[i]是从楼梯第i 个台阶向上爬需要支付的费用，下标从0开始。一旦你支付此费用，即可选择向上爬一个或者两个台阶。你可以选择从下标为 0 或下标为 1 的台阶开始爬楼梯。

   请你计算并返回达到楼梯顶部的最低花费。

   示例：

   ```
   输入：[2,5,20]
   返回值：5
   说明：你将从下标为1的台阶开始，支付5 ，向上爬两个台阶，到达楼梯顶部。总花费为5
   ```

   思路：可以用一个数组记录每次爬到第i阶楼梯的最小花费，然后每增加一级台阶就转移一次状态，最终得到结果。

- **初始状态：** 因为可以直接从第0级或是第1级台阶开始，因此这两级的花费都直接为0.
- **状态转移：** 每次到一个台阶，只有两种情况，要么是它前一级台阶向上一步，要么是它前两级的台阶向上两步，因为在前面的台阶花费我们都得到了，因此每次更新最小值即可，转移方程为：dp[i]=min(dp[i−1]+cost[i−1], dp[i−2]+cost[i−2])

代码：

```
int minCostClimbingStairs(int cost[], int n)
{
    int *dp = (int *)malloc(sizeof(int) * (n + 1));
    for (int i = 2; i <= n; i++)
        dp[i] = min(dp[i - 1] + cost[i - 1], dp[i - 2] + cost[i - 2]);
    return dp[n];
}
```

2. 给定两个字符串str1和str2,输出两个字符串的最长公共子串, 题目保证str1和str2的最长公共子串存在且唯一。

   要求： 空间复杂度 O(n^2)，时间复杂度 O(n^2)

   思路：

- step 1：我们可以用dp(i, j)表示在str1中以第i个字符结尾在str2中以第j个字符结尾时的公共子串长度，
- step 2：遍历两个字符串填充dp数组，转移方程为：如果遍历到的该位两个字符相等，则此时长度等于两个前一位长度+1，dp(i, j)=dp(i-1, j-1)+1, 如果遍历到该位时两个字符不相等，则置为0，因为这是子串，必须连续相等，断开要重新开始。
- step 3：每次更新dp(i, j)后，我们维护最大值，并更新该子串结束位置。
- step 4：最后根据最大值结束位置即可截取出子串。

```
string LCS(string str1, string str2)
{

    int m = str1.size();
```

```
    int n = str2.size();
    int maxlen = 0, end = 0;
    // dp[i][j]表示str1前i个字符和str2的前j个字符的最长公共子串的长度
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            if (str1[i - 1] == str2[j - 1])
                dp[i][j] = dp[i - 1][j - 1] + 1;
            else
                dp[i][j] = 0;
            if (dp[i][j] > maxlen)
            {
                maxlen = dp[i][j];
                end = j - 1;
            }
        }
    }
    string r;
    if (maxlen == 0)
        return "-1";
    else
        r = str2.substr(end - maxlen + 1, maxlen);
    return r;
}
```

3. 给定一个 n * m 的矩阵 a，从左上角开始每次只能向右或者向下走，最后到达右下角的位置，路径上所有的数字累加起来就是路径和，输出所有的路径中最小的路径和。

   要求：时间复杂度 O(nm)

   示例：

   ```
   输入：[[1,3,5,9],[8,1,3,4],[5,0,6,1],[8,8,4,0]]
   返回值：12
   ```

   思路：

   最朴素的解法莫过于枚举所有的路径，然后求和，找出其中最大值。但是像这种有状态值可以转移的问题，我们可以尝试用**动态规划**。

   具体做法：

- step 1：我们可以构造一个与矩阵同样大小的二维辅助数组，其中dp(i, j)表示以(i,j)位置为终点的最短路径和，则dp(0, 0)=matrix(0, 0);
- step 2：很容易知道第一行与第一列，只能分别向右或向下，没有第二种选择，因此第一行只能由其左边的累加，第一列只能由其上面的累加。
- step 3：边缘状态构造好以后，遍历矩阵，补全矩阵中每个位置的dp数组值：如果当前的位置是(i，j)，上一步要么是(i−1, j)往下，要么就是(i, j−1)往右，那么取其中较小值与当前位置的值相加就是到当前位置的最小路径和，因此状态转移公式为dp(i, j)=min(dp(i-1，j)，dp(i，j-1))+matrix(i，j);
- step 4：最后移动到(n−1,m−1)的位置就是到右下角的最短路径和。

代码：

```
int minPathSum(vector<vector<int>> &matrix)
{
    int m = matrix.size();
    int n = matrix[0].size();
    vector<vector<int>> dp(m, vector<int>(n, 0));
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (i == 0 && j == 0)
                dp[i][j] = matrix[0][0];
            else if (i == 0)
                dp[i][j] = dp[i][j - 1] + matrix[i][j];
            else if (j == 0)
                dp[i][j] = dp[i - 1][j] + matrix[i][j];
            else
                dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + matrix[i][j];
        }
    }
    return dp[m - 1][n - 1];
}
```

4. 在下面的数字三角形中寻找一条从顶部到底部的路径，使得路径上所经过的数字之和最小。路径上的每一步都只能向下或者右下走，求出最大和即可

```
输入：
4 //三角形行数
2
3 4
6 5 7
4 1 8 3
返回值：11
说明：最小路径是 2 ， 3 ，5 ， 1
```

思路：用二维数组triangle(i, j)来存放数字三角形，然后用dp(i, j)表示从triangle(i, j)到底部的最小路径和，则

if(i==n-1)

dp(i, j)=triangle(i, j);

else

dp(i, j)=max(dp(i+1, j), dp(i+1, j+1))+triangle(i, j);

代码：

```
int minTrace(vector<vector<int>> &triangle)

{
```

```
    int n = triangle.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));
    for (int i = n - 1; i >= 0; i--)
    {
        for (int j = 0; j <= i; j++)
        {
            if (i == n - 1)
                dp[i][j] = triangle[i][j];
            else
                dp[i][j] = min(dp[i + 1][j], dp[i + 1][j + 1]) + triangle[i][j];
        }
    }
    return dp[0][0];
}
```

5. 输入一个长度为n的整型数组array，数组中的一个或连续多个整数组成一个子数组，子数组最小长度为1。求所有子数组的和的最大值， 当所有整数均为负数时，定义最大字段和为0，要求用动态规划求解

具体做法：

- step 1：可以用dp数组表示以下标i为终点的最大连续子数组和。
- step 2：遍历数组，每次遇到一个新的数组元素，连续的子数组要么加上变得更大，要么这个元素本身就更大，要么会更小，更小我们就舍弃，因此状态转移为dp[i]=max(dp[i−1]+array[i], array[i])。
- step 3：因为连续数组可能会断掉，每一段只能得到该段最大值，因此我们需要维护一个最大值。

代码：

```
int MaxSum(int array[], int n)
{
    int sum = 0, b = 0;
    for (int i = 0; i < n; i++)
    {
        if (b > 0)
            b += array[i];
        else
            b = array[i];
        if (b > sum)
            sum = b;
    }
    return sum;
}
```

6. 给定数组arr，arr中所有的值都为正整数且不重复。每个值代表一种面值的货币，每种面值的货币可以使用任意张，再给定一个aim，代表要找的钱数，求组成aim的最少货币数。如果无解，请返回-1.

要求：时间复杂度 O(n×aim) ，空间复杂度 O(aim)

具体做法：

- step 1：可以用dp[i]表示要凑出i元钱需要最小的货币数。

- step 2：一开始都设置为最大值aim+1，因此货币最小1元，即货币数不会超过aim.
- step 3：初始化dp[0]=0
- step 4：后续遍历1元到aim元，枚举每种面值的货币都可能组成的情况，取每次的最小值即可，转移方程为dp[i]=min(dp[i], dp[i−arr[j]]+1)
- step 5：最后比较dp[aim]的值是否超过aim，如果超过说明无解，否则返回即可。

代码：

```
int minMoney(vector<int> &arr, int aim)
{
    int n = arr.size();
    vector<int> dp(aim + 1);
    int i, j;
    dp[0] = 0;
    for (i = 1; i <= aim; i++)
        dp[i] = aim + 1;
    for (i = 1; i <= aim; i++)
    {
        for (j = 0; j < n; j++)
            if (i >= arr[j])
                dp[i] = min(dp[i], dp[i - arr[j]] + 1);
    }
    if (dp[aim] > aim)
        return -1;
    return dp[aim];
}
```

7. 给定一个长度为 n 的数组 arr，求它的最长严格上升子序列的长度。

要求：时间复杂度 O(n^2)， 空间复杂度 O(n)

思路：

要找到最长的递增子序列长度，每当我们找到一个位置，它是继续递增的子序列还是不是，它选择前面哪一处接着才能达到最长的递增子序列，这类有状态转移的问题常用方法是动态规划。

具体做法：

- step 1：用dp[i]表示到元素i结尾时，最长的子序列的长度，初始化为1，因为只有数组有元素，至少有一个算是递增。
- step 2：第一层遍历数组每个位置，得到n个长度的子数组。
- step 3：第二层遍历相应子数组求对应到元素i结尾时的最长递增序列长度，期间维护最大值。
- step 4：对于每一个到i结尾的子数组，如果遍历过程中遇到元素j小于结尾元素，说明以该元素结尾的子序列加上子数组末尾元素也是严格递增的，因此转移方程为dp[i]=dp[j]+1

代码：

```
int LIS(vector<int> &arr)
{
    int i, j, k, maxL = 0;

    int n = arr.size();
```

```
    if (n == 0)
        return 0;
    vector<int> dp(n);
    dp[0] = 1;
    for (i = 1; i < n; i++)
    {
        k = 0;
        for (j = 0; j < i; j++)
        {
            if (arr[j] <= arr[i] && k < dp[j])
                k = dp[j];
        }
        dp[i] = k + 1;
    }
    for (i = 0; i < n; i++)
        if (dp[i] > maxL)
            maxL = dp[i];
    return maxL;
}
```

8. 给你一个字符串 s，找到 s 中最长的回文子串

思路：对于一个子串而言，如果它是回文串，并且长度大于 2，那么将它首尾的两个字母去除之后，它仍然是个回文串。 我们用 dp(i,j) 表示字符串 s 的第 i 到 j 个字母组成的串是否为回文串，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。

代码：

```
string longest(string s)
{
    int n = s.size();
    int i, j, max = 1, l = 0;
    vector<vector<int>> dp(n, vector<int>(n, 0));
    for (i = 0; i < n; i++)
    {
        dp[i][i] = 1;
    }
    for (int r = 2; r <= n; r++) //回文串长度
    {
        for (i = 0; i <= n - r; i++)
        {
            j = i + r - 1; //回文串的结束位置
            if (s[i] == s[j])
            {
                if (r == 2)
                    dp[i][j] = 1;
                else
                    dp[i][j] = dp[i + 1][j - 1];
            }
            else

                dp[i][j] = 0;
```

```
            if (dp[i][j])
            {
                if (r > max)
                {
                    max = r;
                    l = i;
                }
            }
        }
    }
    return s.substr(l, max);
}
```

9. 你是一个经验丰富的小偷，准备偷沿街的一排房间，每个房间都存有一定的现金，为了防止被发现，你不能偷相邻的两家，即，如果偷了第一家，就不能再偷第二家；如果偷了第二家，那么就不能偷第一家和第三家。给定一个整数数组nums，数组中的元素表示每个房间存有的现金数额，请你计算在不被发现的前提下最多的偷窃金额。

示例：

```
输入：[1,2,3,4]
返回值：6
说明：最优方案是偷第 2，4 个房间
```

具体做法：

- step 1：用dp[i]表示长度为i的数组，最多能偷取到多少钱，只要每次转移状态逐渐累加就可以得到整个数组能偷取的钱。
- step 2：**（初始状态）** 如果数组长度为1，只有一家人，肯定是把这家人偷了，收益最大，因此dp[1]=nums[0]
- step 3：**（状态转移）** 每次对于一个人家，我们选择偷他或者不偷他，如果我们选择偷那么前一家必定不能偷，因此累加的上上级的最多收益，同理如果选择不偷他，那我们最多可以累加上一级的收益。因此转移方程为dp[i]=max(dp[i−1], nums[i−1]+dp[i−2])。这里的i在dp中为数组长度，在nums中为下标。

代码：

```
int rob(vector<int> &nums)
{
    int n = nums.size();
    // dp[i]表示长度为i的数组，最多能偷取多少钱
    vector<int> dp(n + 1, 0);
    dp[1] = nums[0];
    for (int i = 2; i <= n; i++)
    {
        //对于每家可以选择偷或者不偷
        dp[i] = max(dp[i - 1], nums[i - 1] + dp[i - 2]);
    }
    return dp[n];
}
```

10. 小红拿到了一个数组。她想取一些不相邻的数，使得取出来的数之和尽可能大。你能帮帮她吗?

示例:

```
输入：4
     2 6 4 1
输出：7
说明：取 6 和 1 即可
```

思路: 设 dp(i, 0)为不取第 i个数的情况下，前 i 个数不相邻取数的最大值；dp(i, 1)为取第 i个数的情况下，前 i个数不相邻取数的最大值。那么有转移方程:

1) 不取当前数，那么前一个数取不取都行:

dp(i, 0)=max(dp(i-1, 1), dp(i-1, 0))

2) 若取当前数，那么前一个数一定不能取:

dp(i, 1)=dp(i-1, 0) + a[i]

最后需要输出 max(dp(n, 0), dp(n, 1))

代码:

```
int MaxSum(vector<int>nums) {
    int n = nums.size();
    vector<vector<int>>dp(n, vector<int>(2, 0));
    dp[0][0] = 0;
    dp[0][1] = nums[0];
    for (int i = 1; i < n; i++) {
        dp[i][0] = max(dp[i - 1][0], dp[i - 1][1]);
        dp[i][1] = dp[i - 1][0] + nums[i];
    }
    int sum = max(dp[n - 1][0], dp[n - 1][1]);
    return sum;
}
```

11. 假设你有一个数组prices，长度为n，其中prices[i]是股票在第i天的价格，请根据这个价格数组，返回买卖股票能获得的最大收益

    (1) 你可以买入一次股票和卖出一次股票，并非每天都可以买入或卖出一次，总共只能买入和卖出一次，且买入必须在卖出的前面的某一天

    (2) 如果不能获取到任何利润，请返回0

    (3) 假设买入卖出均无手续费

示例:

```
输入：[8,9,2,5,4,7,1]
返回值：5
说明：在第3天(股票价格 = 2)的时候买入，在第6天(股票价格 = 7)的时候卖出，最大利润 = 7-2 = 5 ，不能选择在第2天买入，第3天卖出，这样就亏损7了；同时，你也不能在买入前卖出股票。
```

具体做法：

- step 1：用dp(i, 0)表示第i天不持股到该天为止的最大收益，dp(i, 1)表示第i天持股，到该天为止的最大收益;
- step 2：**（初始状态）** 第一天不持股，则总收益为0，dp(0, 0)=0；第一天持股，则总收益为买股票的花费，此时为负数，dp(0, 1)=−prices[0];
- step 3：**（状态转移）** 对于之后的每一天，如果当天不持股，有可能是前面的若干天中卖掉了或是还没买，因此到此为止的总收益和前一天相同，也有可能是当天才卖掉，我们选择较大的状态

  dp(i, 0)=max(dp(i-1, 0), dp(i-1, 1) + prices[i])
- step 4：如果当天持股，有可能是前面若干天中买了股票，当天还没卖，因此收益与前一天相同，也有可能是当天买入，此时收益为负的股价，同样是选取最大值：dp(i, 1)=max(dp(i-1, 1), -prices[i])

代码：

```
int maxProfit(vector<int>& prices) {
        int n = prices.size();
        //dp[i][0]表示某一天不持股到该天为止的最大收益，dp[i][1]表示某天持股，到该天为止的最大收益
        vector<vector<int>>dp(n, vector<int>(2, 0));
        dp[0][0] = 0;
        dp[0][1] = -prices[0];
        for (int i = 1; i < n; i++) {
            dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
            dp[i][1] = max(dp[i - 1][1], -prices[i]);
        }
        return dp[n - 1][0];
}
```

12. 假设你有一个数组prices，长度为n，其中prices[i]是某只股票在第i天的价格，请根据这个价格数组，返回买卖股票能获得的最大收益

    1. 你可以多次买卖该只股票，但是再次购买前必须卖出之前的股票
    2. 如果不能获取收益，请返回0
    3. 假设买入卖出均无手续费

    要求：空间复杂度 O(n)O(n)，时间复杂度 O(n)

    进阶：空间复杂度 O(1)O(1)，时间复杂度 O(n)

    示例：

    ```
    输入：[8,9,2,5,4,7,1]
    返回值：7
    说明：在第1天(股票价格=8)买入，第2天(股票价格=9)卖出，获利9-8=1
          在第3天(股票价格=2)买入，第4天(股票价格=5)卖出，获利5-2=3
          在第5天(股票价格=4)买入，第6天(股票价格=7)卖出，获利7-4=3
          总获利1+3+3=7，返回7
    ```

    具体做法：

    - step 1：用dp(i, 0)表示第i天不持股到该天为止的最大收益，dp(i, 1)表示第i天持股，到该天为止的最大收益;

- step 2： **（初始状态）** 第一天不持股，则总收益为0，dp(0, 0)=0；第一天持股，则总收益为买股票的花费，此时为负数，dp(0, 1)=−prices[0];
- step 3： **（状态转移）** 对于之后的每一天，如果当天不持股，有可能是前面的若干天中卖掉了或是还没买，因此到此为止的总收益和前一天相同，也有可能是当天才卖掉，我们选择较大的状态

  dp(i, 0)=max(dp(i-1, 0), dp(i-1, 1) + prices[i])
- step 4：如果当天持股，有可能是前面若干天中买了股票，当天还没卖，因此收益与前一天相同，也有可能是当天买入，此时收益为负的股价，同样是选取最大值：dp(i, 1)=max(dp(i-1, 1), dp(i-1, 0) - prices[i])

代码：

```
int maxProfit(vector<int>& prices) {
        int n = prices.size();
        //dp[i][0]表示某一天不持股到该天为止的最大收益，dp[i][1]表示某天持股，到该天为止的最大收益
        vector<vector<int>>dp(n, vector<int>(2, 0));
        dp[0][0] = 0;
        dp[0][1] = -prices[0];
        for (int i = 1; i < n; i++) {
            dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
            dp[i][1] = max(dp[i - 1][1], dp[i - 1][0] - prices[i]);
        }
        return dp[n - 1][0];
}
```

13. 给定一个正整数n，请找出最少个数的完全平方数，使得这些完全平方数的和等于n。

完全平方指用一个整数乘以自己例如11，22，3*3等，依此类推。若一个数能表示成某个整数的平方的形式，则称这个数为完全平方数。例如:1，4，9，和16都是完全平方数，但是2，3，5，8，11等等不是

示例：

```
输入：5
返回值：2
说明：5=1+4
```

思路：设置 dp[i] 表示能组成 i 的最少完全平方数,

dp[0] = 0

dp[i] = min(dp[i - j * j] + 1, dp[i]) j < i ，

dp[i - j * j] 表示上一个组成完全平方数的最少值,因为每个正数都可以是完全平方数(1 + 1 + ... + 1),最多次数可以为 n，遍历数组，根据转移方程得出每个状态位的dp值

代码：

```
int numSquares(int n)
{
    vector<int> dp(n + 1, 0);
    int minn;
    for (int i = 1; i <= n; i++)
    {
        minn = dp[i - 1];
        for (int j = 2; j * j <= i; j++)
            minn = min(minn, dp[i - j * j]);
        dp[i] = minn + 1;
    }
    return dp[n];
}
```

14. 给定两个字符串 str1 和 str2 ， 请你算出将 str1 转为 str2 的最少操作数。

    你可以对字符串进行3种操作：

    1.插入一个字符

    2.删除一个字符

    3.修改一个字符。

    保证字符串中只出现小写英文字母。

    示例：

    ```
    输入："abc","adc",5,3,2
    返回值：2
    ```

    思路：

    把第一个字符串变成第二个字符串，我们需要逐个将第一个字符串的子串最少操作下变成第二个字符串，这就涉及了第一个字符串增加长度，状态转移，那可以考虑动态规划。用 `dp[i][j]` 表示从两个字符串首部各自到 `str1[i]` 和 `str2[j]` 为止的子串需要的编辑距离，那很明显 `dp[str1.length][str2.length]` 就是我们要求的编辑距离。（下标从1开始）

    具体做法：

    - step 1：**初始条件：** 假设第二个字符串为空，那很明显第一个字符串子串每增加一个字符，编辑距离就加1，这步操作是删除；同理，假设第一个字符串为空，那第二个字符串每增加一个字符，编剧距离就加1，这步操作是添加。
    - step 2：**状态转移：** 状态转移肯定是将dp矩阵填满，那就遍历第一个字符串的每个长度，对应第二个字符串的每个长度。如果遍历到 `str1[i]` 和 `str2[j]` 的位置，这两个字符相同，这多出来的字符就不用操作，操作次数与两个子串的前一个相同，因此有 `str1[i][j]=dp[i-1][j-1]` ；如果这两个字符不相同，那么这两个字符需要编辑，但是此时的最短的距离不一定是修改这最后一位，也有可能是删除某个字符或者增加某个字符，因此我们选取这三种情况的最小值增加一个编辑距离，即d `str1[i][j]=min(dp[i-1][j-1], min(dp[i-1][j], dp[i][j-1]))+1`

代码：

```
int editDistance(string str1, string str2)
```

```
{
    int m = str1.size(), n = str2.size();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    int i, j;
    for (i = 0; i <= m; i++)
        dp[i][0] = i;
    for (j = 0; j <= n; j++)
        dp[0][j] = j;
    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            if (str1[i - 1] == str2[j - 1])
                dp[i][j] = dp[i - 1][j - 1];
            else
                dp[i][j] = min(dp[i - 1][j - 1], min(dp[i - 1][j], dp[i][j - 1])) + 1;
        }
    }
    return dp[m][n];
}
```