一.动态规划问题的简介

1.动态规划问题背景

动态规划问题与分治思想类似，都是将原问题分解为若干个子问题，通过求解子问题得到原问题的解。

递归形式简明直观，但是不代表在任何情况下都是有效的，以斐波那契数列为例，很多子问题的计算重复了多次。

1）斐波那契数列

在求斐波那契数列f(n)时，我们很容易得到递归定义

$$f(n)=\begin{cases}1 & n=1,2\\ f(n-1)+f(n-2) & n>=3\end{cases}$$

我们很容易写出以下代码

```
int fib(int n){
  if(n==1||n==2) return 1;
  return fib(n-1)+fib(n-2);
}
```

时间复杂度：

$$T(n)=\begin{cases}1 & n=1,2\\ T(n-1)+T(n-2) & n>=3\end{cases}$$

运行时间是指数级别的。同时，我们发现，中间很多f(k)的计算重复了多次，其实是不必要的。递归是一种自顶向下再自底向上的设计思想，如果将递归改为递推的形式，自底向上分析问题，可能会减少子过程的重复计算，在斐波那契数列的计算中，使用递推进行求解，时间复杂度为O(n)，空间复杂度为O(1)

```
int fib(int n){
    int fib1 = 1;
    int fib2 = 1;
    int temp;
    for(int i=3;i<=n;i++){
        temp = fib2;
        fib2 = fib1+fib2;
        fib1 = temp;
    }
    return fib2;
}
```

2）二项式系数计算

二项式系数递归求解呈指数级增加：

$$C_n^k=\begin{cases}1 & n=k\\ 1 & k=0\\ C_{n-1}^{k-1}+C_{n-1}^k & others\end{cases}$$

利用帕斯卡三角改为递推的形式：

```
int c(int n,int k){
    if(n==k||k==0) return 1;
    tri[0] = 1;
    tri[1] = 1;
    for(int i=2;i<n;i++){
        tri[0] = 1;
        tri[i] = 1;
        for(int j=i-1;j>=1;j--){
            tri[j] = tri[j-1] + tri[j];
        }
    }
    return tri[k-1]+tri[k];
}
```

3）总结

分治法是把大问题分解成一些相互独立的子问题，递归的求解这些子问题然后将他们合并来得到整个问题的解
动态规划是通过组合子问题的解来解决整个大问题。各个子问题不是独立的，也就是各个子问题包含公共子问题。它可以避免遇到的子问题的重复求解。

应用这种算法思想解决问题的可行性，对子问题与原问题的关系，以及子问题之间的关系这两方面有一些要求，它们分别对应了最优子结构和重复子问题。

- 最优子结构：最优子结构规定的是子问题与原问题的关系

  当我们在求一个问题最优解的时候，如果可以把这个问题分解成多个子问题，然后递归地找到每个子问题的最优解，最后通过一定的数学方法对各个子问题的最优解进行组合得出最终的结果。

- 重复子问题：规定的是子问题与子问题的关系。

  当我们在递归地寻找每个子问题的最优解的时候，有可能会重复地遇到一些更小的子问题，而且这些子问题会重叠地出现在子问题里，出现这样的情况，会有很多重复的计算，动态规划可以保证每个重叠的子问题只会被求解一次。当重复的问题很多的时候，动态规划可以减少很多重复的计算。

2.解决动态规划问题的思考过程

1）最长递增子序列

https://leetcode-cn.com/problems/longest-increasing-subsequence/

- 考虑能否将问题规模减小
  - 对半分：前后最长递增串可能没什么关系，这样划分有问题
  - 每次减少一个：对于数组A[n],其最长递增子序列的最后一个元素一定是以某个元素A[i]结尾的，那么我们可以定义f(n)为以a[n]结尾的最长递增子序列，在f(0..n)中，最大值f(i)就是所求结果，那么状态转移方程为：

$$f(n) = max f(i) + 1 \quad a[i] < a[n] 且 i = 0..n-1$$

  - 动态规划问题的难点：

    - 怎么定义f(n)
    - 怎么找出状态转移方程
- 考虑用递归的方式求解

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        return LISatN(nums,nums.size()-1);
    }
    int LISatN(vector<int>& nums,int n){
        if(n==0) return 1;
        int max=0;
        int temp;
        for(int i=0;i<n;i++){
            if(nums[n]>nums[i]){
                temp = LISatN(nums,i);
                max = max>temp?max:temp;
            }
        }
        return max+1;
    }
};
```

    - 在每次计算时，会重复调用很多函数，考虑用数组保留其现场

- 动态规划

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        vector<int> dp(nums.size());
        dp[0]=1;
        int max=0;
        for(int i=1;i<nums.size();i++){
            max=0;
            for(int j=0;j<i;j++){
                if(nums[j]<nums[i]){
                    max=max>dp[j]?max:dp[j];
                }
            }
            dp[i]=max+1;
        }
        max = 0;
        for(int i=0;i<nums.size();i++){
            max=max>dp[i]?max:dp[i];
        }
        return max;
    }
};
```

二.线性动态规划

1.线性动态规划简介

- 状态定义：`dp[n] := [0..n] 上问题的解`
- 状态转移：`dp[n] = f(dp[n-1], ..., dp[0])`

从以上状态定义和状态转移可以看出，大规模问题的状态只与较小规模的问题有关，而问题规模完全用一个变量 i 表示，i 的大小表示了问题规模的大小，因此从小到大推 i 直至推到 n，就得到了大规模问题的解，这就是线性动态规划的过程。

按照问题的输入格式，线性动态规划解决的问题主要是单串，双串，矩阵上的问题，因为在单串，双串，矩阵上问题规模可以完全用位置表示，并且位置的大小就是问题规模的大小。因此从前往后推位置就相当于从小到大推问题规模。

2.单串问题

1）单串问题简介

- 状态定义：dp[i] :=考虑[0..i]上原问题的解

  一般分为以下两种类型：

  - dp[i]必须考虑nums[i]
  - dp[i]无需考虑nums[i]

- 状态转移方程

  根据dp[i]的依赖，分为以下两类：

  - dp[i]依赖于比i小的O(1)个子问题：如`dp[i]=dp[i-1]+nums[i]`，如第一节的斐波那契数列
    - 时间复杂度为O(n),空间复杂度经优化可为O(1)
  - dp[i]依赖于比i小的O(n)个子问题：如上一节的例题
    - 时间复杂度为O(n^2),空间复杂度为O(n)

2）dp[i]需要考虑nums[i]（大多数情况）

①爬楼梯问题

https://leetcode-cn.com/problems/climbing-stairs/

- 状态定义：dp[i]：走i级台阶有几种方法
- 状态转移函数：dp[i] = dp[i-2]+dp[i-1]

  dp[1]=1,dp[0]=1

```
class Solution {
public:
    int climbStairs(int n) {
        int stair0 = 1;
        int stair1 = 1;
        int sum;
        for(int i=2;i<=n;i++){
```

```
            sum = stair0 + stair1;
            stair0 = stair1;
            stair1 = sum;
        }
        return stair1;
    }
};
```

- 由于dp[i]只取决于dp[i-1]和dp[i-2]，只需要保留两个变量，不需要保存整个dp数组

3）LIS问题

①最长递增子序列的个数

- https://leetcode-cn.com/problems/number-of-longest-increasing-subsequence/
- 思路
  - 类型：需要考虑nums[i]，dp[i]依赖于O(n)的问题规模
  - 状态定义和状态转移函数见1.2.1

```
struct Path{
    int maxpath;
    int num;
};
class Solution {
public:
    int findNumberOfLIS(vector<int>& nums) {
        vector<Path> path(nums.size());
        path[0].maxpath=1;
        path[0].num=1;
        int max;
        int num=1;
        for(int i=1;i<nums.size();i++){
            max=0;
            num=1;
            for(int j=0;j<i;j++){
                if(nums[i]>nums[j]){
                    if(path[j].maxpath>max){
                        max=path[j].maxpath;
                        num=path[j].num;
                    }else if(path[j].maxpath==max){
                        num+=path[j].num;
                    }
                }
            }
            path[i].maxpath=max+1;
            path[i].num=num;
        }
        max=0;
        num=1;
```

```
        for(int i=0;i<path.size();i++){
            if(path[i].maxpath>max){
                max=path[i].maxpath;
                num=path[i].num;
            }else if(path[i].maxpath==max){
                num+=path[i].num;
            }
        }
        return num;
    }
};
```

- 这里需要额外的空间记录以nums[i]结尾的最长序列的个数，其实就是在统计dp[0..i-1]最大序列的过程中，如果遇到dp[j]等于max的情况，就需要添加以nums[j]结尾的最长序列的个数。

②俄罗斯套娃信封问题

- https://leetcode-cn.com/problems/russian-doll-envelopes/
- 思路：看到这个题第一反应就是先排序，本以为排序可以优化动态规划的时间复杂度，但是如果按w排序后，实际上h还是无序的，只是根据w的大小关系硬性规定了信封的先后次序，本质上还是最长递增子序列的问题

```
bool cmp(vector<int> a,vector<int> b){
    if(a[0]!=b[0]){
        return a[0]<b[0];
    }else{
        return a[1]<b[1];
    }
}
class Solution {
public:
    int maxEnvelopes(vector<vector<int>>& envelopes) {
        vector<int> dp(envelopes.size());
        sort(envelopes.begin(),envelopes.end(),cmp);
        dp[0]=1;
        int max;
        for(int i=1;i<envelopes.size();i++){
            max=0;
            for(int j=0;j<i;j++){
                if(envelopes[i][0]>envelopes[j][0]&&envelopes[i][1]>envelopes[j]
[1]&&dp[j]>max){
                    max=dp[j];
                }
            }
            dp[i]=max+1;
        }
        max=0;
        for(int i=0;i<envelopes.size();i++){
            if(dp[i]>max){
```

```
                max=dp[i];
            }
        }
        return max;
    }
};
```

4）最大子数组和系列

①乘积最大子数组

- https://leetcode-cn.com/problems/maximum-product-subarray/
- 思路：设B[i...j]是乘积最大的连续子数组，那么B一定是以nums数组的某个元素nums[j]结尾，设dp[j]为以nums[j]结尾的连续子数组乘积最大值，那么dp[1...n]中一定有原问题的解，怎么样得到dp[i]呢，首先dp[1]=nums[1]，dp[i]一定以nums[i]结尾，该连续子数组有两种情况，要么和之前的数组连接，要么自成一派，dp[i]=max(nums[i],nums[i]*之前的某个连续数组),nums[i]为定值，要让`nums[i]*之前的某个连续数组`最大，要根据nums[i]的情况进行讨论（之前连续和问题可以直接断定需要让之前某个连续数组和最大）若nums[i]>0,之前的某个连续数组值越大，整体越大，若nums[i]<0,之前的某个连续数组值越大，整体越小；那么如果要求最大值，需要求出以nums[i-1]结尾的连续数组乘积最大值和最小值，则从开始递推时，需要记录最大值和最小值。最后从最大值中选出全局最优解。

```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int dp_max=nums[0];
        int dp_min=nums[0];
        int Max=dp_max;
        int temp;
        for(int i=1;i<nums.size();i++){
            if(nums[i]==0){
                dp_max=0;
                dp_min=0;
            }else if(nums[i]>0){
                dp_max=max(dp_max*nums[i],nums[i]);
                dp_min=min(dp_min*nums[i],nums[i]);
            }else{
                temp = dp_max;
                dp_max=max(dp_min*nums[i],nums[i]);
                dp_min=min(temp*nums[i],nums[i]);
            }
            Max=max(Max,dp_max);
        }
        return Max;
    }
};
```

②最大子矩阵

- https://leetcode-cn.com/problems/max-submatrix-lcci/
- 思路：参考题解https://leetcode-cn.com/problems/max-submatrix-lcci/solution/zhe-yao-cong-zui-da-zi-xu-he-shuo-qi-you-jian-dao-/
- 代码：

```
class Solution {
public:
    vector<int> getMaxMatrix(vector<vector<int>>& matrix) {
        int n = matrix.size();
        int m = matrix[0].size();
        vector<int> ans(4);
        vector<int> arr(m);
        int Max = matrix[0][0];
        ans[0]=0;
        ans[1]=0;
        ans[2]=0;
        ans[3]=0;
        for(int i=0;i<n;i++){
            for(int k=0;k<m;k++) arr[k]=0;
            for(int j=i;j<n;j++){
                for(int k=0;k<m;k++) arr[k] += matrix[j][k];
                // dp
                int dp0=arr[0];
                int index_begin = 0;
                for(int k=1;k<m;k++){
                    if(dp0<=0){
                        index_begin=k;
                        dp0=arr[k];
                    }else{
                        dp0+=arr[k];
                    }
                    if(dp0>Max){
                        ans[0]=i;
                        ans[1]=index_begin;
                        ans[2]=j;
                        ans[3]=k;
                        Max=dp0;
                    }
                }
            }
        }
        return ans;
    }
};
```

5）不相邻子序列最大和问题

①打家劫舍

- https://leetcode-cn.com/problems/house-robber/
- 思路：nums[n]分为拿与不拿两种情况，定义get[n]为必须拿nums[n]时总金额最大，notget[n]为不许拿nums[n]时总金额最大，可得递推式：(也可以将两种情况归为金额最大的情况，dp[n]=max(dp[n-2]+nums[n],dp[n-1]))

$$get(n)=\begin{cases} nums[1] & n=1 \\ nums[2] & n=2 \\ get(n)=nums[n]+max(get(n-2),notget(n-2)) & n>=3 \end{cases}$$

$$notget(n)=\begin{cases} 0 & n=1 \\ notget(n)=max(get(n-1),notget(n-1)) & n>=2 \end{cases}$$

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if(nums.size()==1) return nums[0];
        int get1,get2,notget1,notget2,temp1,temp2;
        get1=nums[0];
        get2=nums[1];
        notget1=0;
        notget2=get1;
        for(int i=2;i<nums.size();i++){
            temp1 = nums[i]+max(get1,notget1);
            temp2 = max(get2,notget2);
            get1=get2;
            notget1=notget2;
            get2=temp1;
            notget2=temp2;
        }
        return max(get2,notget2);
    }
};
```

②打家劫舍2

- https://leetcode-cn.com/problems/house-robber-ii/
- 思路：相比于上一题，增加了一个约束条件，那就是第一家和最后一家不能同时偷，假设不偷第一家，那么问题相当于在nums[2...n]上找最优解，假设不偷最后一家，问题相当于在nums[1...n-1]上找最优解，比较两个最优解得出最终结果即可。

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if(nums.size()==1) return nums[0];
        if(nums.size()==2) return max(nums[0],nums[1]);
```

```
        int dp0_1=nums[0];
        int dp1_1=max(nums[1],nums[0]);
        int dp0_2=nums[1];
        int dp1_2=max(nums[1],nums[2]);
        int temp1,temp2;
        for(int i=2;i<nums.size()-1;i++){
            temp1=max(nums[i]+dp0_1,dp1_1);
            temp2=max(nums[i+1]+dp0_2,dp1_2);
            dp0_1=dp1_1;
            dp1_1=temp1;
            dp0_2=dp1_2;
            dp1_2=temp2;
        }
        return max(dp1_1,dp1_2);
    }
};
```

6）单串问题的变形

①最长的斐波那契子序列长度

- https://leetcode-cn.com/problems/length-of-longest-fibonacci-subsequence/
- 思路：如果存在这样最长的斐波那契序列，一定是以nums[i]，i=3...n（数组角标从0开始）结尾，那么很容易想到定义状态为dp[i]=以nums[i]结尾的最长的斐波那契序列长度，对于dp[i]的求解，如果从dp[1...i-1]中挑选最长的斐波那契序列，并不保证加上nums[i]后仍为斐波那契序列，假设nums[x1],nums[x2],nums[x3],nums[x4],nums[x5]构成斐波那契数列，dp[x3]最长不一定为3，可能为4，但是由于最后两个数字没办法添加一个新的nums，最长序列为4，由此，部分最长的斐波那契数列不一定是解的组。

  修改定义：dp[i][j]表示以nums[i]和nums[j]结尾的斐波那契数列最长长度，dp[k][i]表示nums[k]和nums[i]结尾的斐波那契数列最长长度

  nums[k] = nums[j] - nums[i];

  dp[i][j]=dp[k][i]+1;

  这里会不断涉及按值查找下标，我们维护一个哈希映射保存<value,index>

```
class Solution {
public:
    int lenLongestFibSubseq(vector<int>& arr) {
        vector<vector<int>> dp(arr.size());
        unordered_map<int, int> hashmap;
        for(int i=0;i<arr.size();i++){
            dp[i].resize(arr.size());
        }
        dp[0][0] = 1;
        hashmap.insert(make_pair(arr[0],0));
        for(int i=1;i<arr.size();i++){
            dp[i][i]=1;
            dp[0][i]=2;
```

```
            hashmap.insert(make_pair(arr[i],i));
        }
        for(int i=1;i<arr.size();i++){
            for(int j=i+1;j<arr.size();j++){
                int temp = arr[j] - arr[i];
                if(temp<arr[i]&&hashmap.count(temp)>0){
                    dp[i][j] = dp[hashmap[temp]][i] + 1;
                }else dp[i][j] = 2;
            }
        }
        int longestres=0;
        for(int i=0;i<arr.size();i++){
            for(int j=i;j<arr.size();j++){
                longestres = max(longestres,dp[i][j]);
            }
        }
        if(longestres>=3) return longestres;
        else return 0;
    }
};
```

3.多串问题

1）LCS系列

①最长公共子序列

- https://leetcode-cn.com/problems/longest-common-subsequence/
- 优化过程

```
class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {
        int m = text1.length();
        int n = text2.length();
        vector<vector<int>> dp(m+1);
        for(int i=0;i<=m;i++){
            dp[i].resize(n+1);
            dp[i][0] = 0;
        }
        for(int i=0;i<=n;i++){
            dp[0][i] = 0;
        }
        for(int i=1;i<=m;i++){
            for(int j=1;j<=n;j++){
                if(text1[i-1]==text2[j-1]){
                    dp[i][j]=dp[i-1][j-1]+1;
                }else{
                    dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
```

```
                }
            }
        }
        return dp[m][n];
    }
};
```

- 输出最长公共子序列：（创建路径矩阵）

```
class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {
        int m = text1.length();
        int n = text2.length();
        vector<vector<int>> dp(m+1);
        vector<vector<char>> path(m+1);
        for(int i=0;i<=m;i++){
            dp[i].resize(n+1);
            path[i].resize(n+1);
            dp[i][0] = 0;
        }
        for(int i=0;i<=n;i++){
            dp[0][i] = 0;
        }
        for(int i=1;i<=m;i++){
            for(int j=1;j<=n;j++){
                if(text1[i-1]==text2[j-1]){
                    dp[i][j]=dp[i-1][j-1]+1;
                    path[i][j]= '\\';
                }else{
                    dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
                    if(dp[i][j]==dp[i-1][j]) path[i][j]='|';
                    else path[i][j]='-';
                }
            }
        }
        // 输出path[i][j]
        for(int i=1;i<=m;i++){
            for(int j=1;j<=n;j++){
                cout << path[i][j] << '\t';
            }
            cout << endl;
        }
        printLCS(text1, path, m,n);
        return dp[m][n];
    }
    void printLCS(string& arr,vector<vector<char>>& path,int m,int n){
        if(m==0||n==0) return;
        if(path[m][n]=='|') printLCS(arr,path, m-1, n);
```

```
        else if(path[m][n]=='\\') {
            printLCS(arr,path, m-1, n-1);
            cout << arr[m-1];
        }else{
            printLCS(arr, path, m, n-1);
        }
    }
};
```

2）字符串匹配系列

①正则表达式

- https://leetcode-cn.com/problems/regular-expression-matching/

```
class Solution {
public:
    bool isMatch(string s, string p) {
        int m = s.size();
        int n = p.size();
        vector<vector<bool>> dp(m+1);
        dp[0].resize(n+1);
        dp[0][0]=1;
        for(int i=1;i<=m;i++){
            dp[i].resize(n+1);
            dp[i][0]=0;
        }
        for(int i=1;i<=n;i++){
            if(p[i-1]=='*') dp[0][i]=dp[0][i-2];
            else dp[0][i]=0;
        }
        for(int i=1;i<=m;i++){
            for(int j=1;j<=n;j++){
                if(p[j-1]=='*'){
                    // 不匹配
                    if(p[j-2]!='.'&&p[j-2]!=s[i-1]) dp[i][j]=dp[i][j-2];
                    else dp[i][j]=dp[i-1][j]||dp[i][j-2];
                }else{
                    if(p[j-1]!='.'&&p[j-1]!=s[i-1]) dp[i][j]=0;
                    else dp[i][j]=dp[i-1][j-1];
                }
            }
        }
        return dp[m][n];
    }
};
```

- 注意：即使子串a和a*可以匹配，在实际匹配时，a*依旧可能为0，需要分类讨论，只要有一种成立即可。

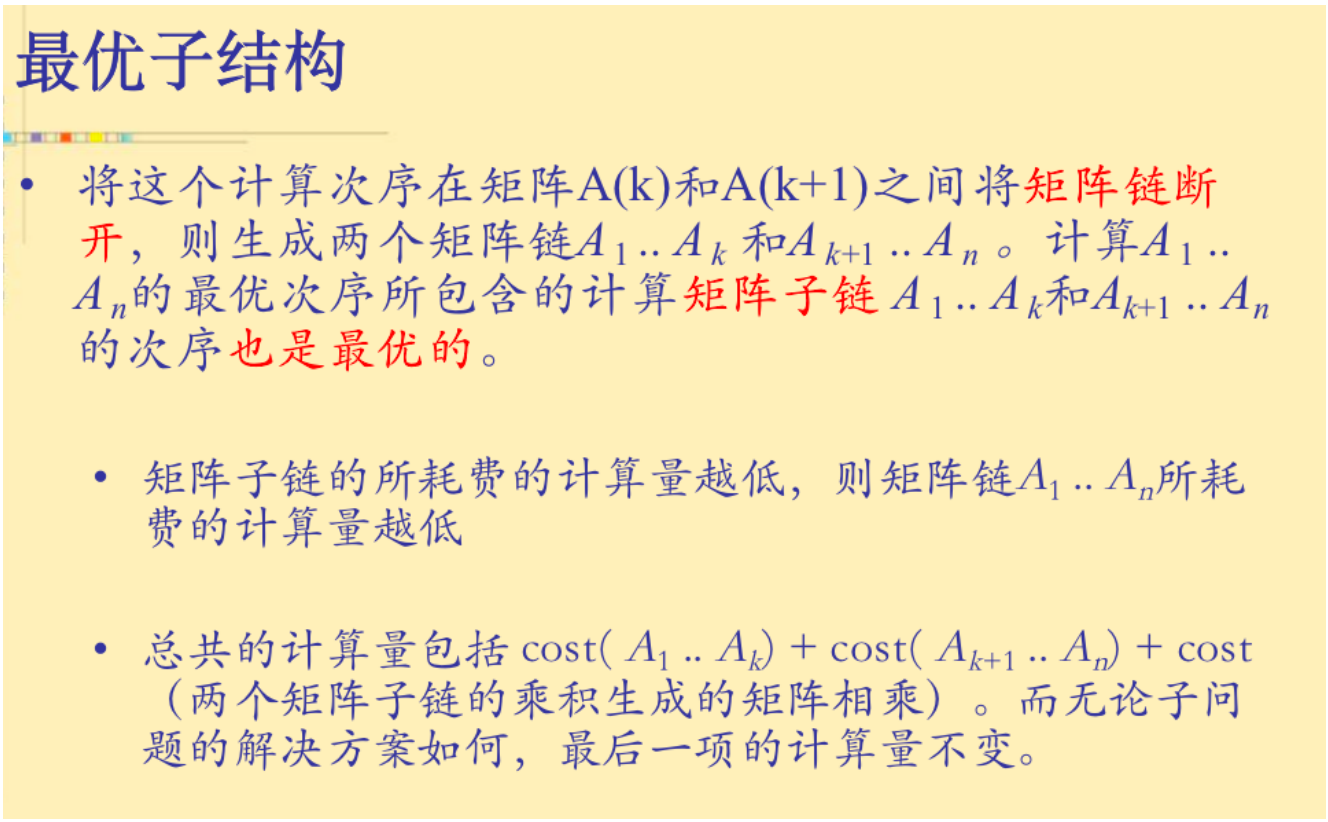
4.矩阵问题

1）矩阵链相乘问题

- 问题描述：给定n个矩阵{A1,A2,...,An}，其中Ai与Ai+1是可乘的，i=1,2,...,n-1。要算出这n个矩阵的连乘积A1A2...An。
  由于矩阵乘法满足结合律，故计算矩阵的连乘积可以有许多不同的计算次序。这种计算次序可以用加括号的方式来确定。若一个矩阵连乘积的计算次序完全确定，也就是说该连乘积已完全加括号，则可以依此次序反复调用2个矩阵相乘的标准算法计算出矩阵连乘积。

  每一种完全加括号的方式对应于一个矩阵连乘积的计算次序，这决定着作乘积所需要的计算量。如对于m*n与n*k矩阵相乘，需要m*n*k次乘法。

  如何确定计算矩阵连乘积A1A2...An的计算次序（完全加括号方式），使得依此次序计算矩阵连乘积需要的数乘次数最少。

- 

动态规划方法

- 将矩阵连乘积A(i)A(i+1)…A(j)简记为A[i:j]，设计算A[i:j](1 <= i <= j <= n)所需要的最少数乘次数m[i,j]，则原问题的最优值为m[1,n]
- 当i＝j时，A[i:j]=Ai，因此，m[i,i]＝0，i＝1,2,…,n
- 当i＜j时，m[i,j] = m[i,k] + m[k+1,j] + p(i-1)p(k)p(j)
  - 这里A(i)的维数为p(i-1)*p (i)(注：p(i-1)为矩阵A(i)的行数，p(i)为矩阵A[i]的列数)
- 可以递归地定义m[i,j]为：

$$m[i,j]=\begin{cases}0 & if\ i=j\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p[i-1]p[k]p[j]\} & if\ i<j\end{cases}$$

```cpp
#include<iostream>
using namespace std;
int main(){
    // 1.对应矩阵为1..N,0处存放第一个矩阵的行
    // 存放矩阵时只存放其列数，一个矩阵的行在前一个单元
    int matrix[6]={5,10,4,6,10,2};
    // 2.A[i,j]表示从矩阵i到矩阵j最小的连乘次数
    // A[i,j]=0
    // 计算时，从j-i=0一直到j-i=N-1
    int A[5][5];
    for(int i=0;i<5;i++) A[i][i]=0;
    for(int k=1;k<5;k++){
        for(int i=k;i<5;i++){
            // 填充A[i-k][i]
            int Min = INT_MAX;
            for(int j=i-k;j<i;j++){
                Min = min(A[i-k][j]+A[j+1][i]+matrix[i-k]*matrix[j+1]*matrix[i+1],Min);
            }
            A[i-k][i]=Min;
        }
    }
    cout << "计算最少次数为" << A[0][4] << endl;

    return 0;
}
```

https://blog.csdn.net/winycg/article/details/68567784

三.前缀和问题

1.前缀和简介

- 考虑数组a[0..n-1]，定义数组S[0...n]，满足`S0=0`，`Si=a[0]+a[1]+..+a[i-1]`,这样的数组称为前缀和数组
- 前缀和数组通常可以解决以下两类问题
  - 求数组前缀和a[0..i]，即S[i+1]
  - 求区间和a[L,R]，即S[R+1]-S[L]，当已有前缀和数组后，此步操作为O(1)
- 假设有大量需要计算区间和的问题，那么就可以维护这样一个前缀和数组，这个数组所缓存的内容并不是子问题的解，而是相当于计算的中间结果。

  先将所有位置的前缀和预处理出来，然后再处理区间和的查询，这是一种先缓存中间结果再处理查询的思路，因为这些中间结果在查询时需要反复用到，缓存之后就不用反复计算了，因此花时间预处理这些信息是有效的。

2.求区间和

1）动态规划表示

- 状态表示：sum[i]=a[0]+a[1]+...+a[i-1]
- 状态转移：sum[i]=sum[i-1]+a[i-1]
- 初始状态：sum[0]=0

```
int rangeSum(int L,int R){
  return sum[R+1]-sum[L];
}
```

2）区间和检索

- https://leetcode-cn.com/problems/range-sum-query-immutable/
- 思路：基本做法很简单，但是注意到这里会`多次`调用区间求和函数，根据最后的数据量：

提示：

- $0 <= nums.length <= 10^4$
- $-10^5 <= nums[i] <= 10^5$
- $0 <= i <= j < nums.length$
- 最多调用 10^4 次 `sumRange` 方法

如果每次单独遍历求和，计算量的数量级最高为10^8，如果最

开始维护前缀和数组，运算数量级为10^4

```
class NumArray {
public:
    vector<int> rangeSum;
    NumArray(vector<int>& nums) {
        rangeSum.push_back(0);
        for(int i=1;i<=nums.size();i++){
            rangeSum.push_back(nums[i-1]+rangeSum[i-1]);
        }
    }
    int sumRange(int left, int right) {
        return rangeSum[right+1]-rangeSum[left];
    }
};

/**
 * Your NumArray object will be instantiated and called as such:
 * NumArray* obj = new NumArray(nums);
 * int param_1 = obj->sumRange(left,right);
 */
```

3.实现前缀和问题

本节重点说明实现前缀和的数据结构，将会在后续问题中反复运用到。前缀和重点解决的是求区间和和子矩阵的问题

1）区间和检索

见`问题3.2.2`

2）二维区域和检索

- https://leetcode-cn.com/problems/range-sum-query-2d-immutable/
- 思路：同一维的区间和问题，可以定义一个二维数组
  - 状态定义：sum[i+1][j+1]表示区域nums[0][0]~nums[i][j]所有元素求和
  - 状态转移：根据维护sum矩阵时的遍历顺序，sum[i][j]=nums[i][j]+sum[i-1][j]+sum[i][j-1]-sum[i-1][j-1]
  - 初始状态：sum[i][0]=0,sum[0][j]=0

```
class NumMatrix {
public:
    vector<vector<int>> sum;
    NumMatrix(vector<vector<int>>& matrix) {
        sum.resize(matrix.size()+1);
        for(int i=0;i<sum.size();i++){
            sum[i].resize(matrix[0].size()+1);
        }
        for(int i=1;i<sum.size();i++){
```

```
            for(int j=1;j<sum[0].size();j++){
                sum[i][j]=matrix[i-1][j-1]+sum[i-1][j]+sum[i][j-1]-sum[i-1][j-1];
            }
        }
    }

    int sumRegion(int row1, int col1, int row2, int col2) {
        return sum[row2+1][col2+1]-sum[row2+1][col1]-sum[row1][col2+1]+sum[row1]
[col1];
    }
};

/**
 * Your NumMatrix object will be instantiated and called as such:
 * NumMatrix* obj = new NumMatrix(matrix);
 * int param_1 = obj->sumRegion(row1,col1,row2,col2);
 */
```

4.数据结构实现前缀和

我们常见的是将前缀和维护在哈希表中

1）问题一

$a_0, a_1, \ldots, a_{n-1}$ 上有没有一个区间，其和为 target 。

在遍历数组的时候存储S[i],同时查询是否存在j使得target=S[i]-S[j]，及值S[i]-target是否在存储的前缀和中出现，涉及大量的按值查询工作，可以用哈希表(unordered_set)存储数组S[i],能在O(1)的时间内完成按值查找。

2）问题二

$a_0, a_1, \ldots, a_{n-1}$ 上有多少个区间，其和为 target 。

只需要将哈希集合换为哈希映射，存储值的同时存储S[i]等于该值的数量。

- 和为k的子数组
- https://leetcode-cn.com/problems/subarray-sum-equals-k/
- 存储键值对时存储<key,count>

```
class Solution {
public:
    int subarraySum(vector<int>& nums, int k) {
        unordered_map<int, int> hashMap;
        hashMap.insert(make_pair(0, 1));
        int sum = 0;
        int ans = 0;
```

```
        for(int i=0;i<nums.size();i++){
            sum += nums[i];
            if(hashMap.count(sum-k)>0) ans += hashMap[sum-k];
            if(hashMap.count(sum)>0){
                hashMap[sum]++;
            }else{
                hashMap.insert(make_pair(sum, 1));
            }
        }
        return ans;
    }
};
```

3）问题三

$a_0, a_1, \ldots, a_{n-1}$ 上有没有一个区间，其和大于/小于 target 。

http://118.190.20.162/view.page?gpid=T130

5.差分问题

适用于每次同时对若干个元素增加相同值，先求差分，更改后求前缀和。

四.参考资料

https://leetcode-cn.com/leetbook/read/dynamic-programming-1-plus/