代码板子：第二弹

陈彦桥

July 28, 2025

0.1 前言

本册主要包含一些模板题的代码板子，主要来源是本人的做题经历。

Contents

0.1 前言 2

第一章 动态规划 **5**

1.1 如何考虑动态规划 5

1.1.1 考虑状态 5

1.1.2 考虑状态转移 5

1.2 背包 DP 6

1.2.1 0-1 背包问题 6

1.2.2 完全背包问题 6

1.2.3 多重背包问题 8

1.2.4 分组背包问题 9

1.3 线性 dp 10

1.3.1 数字三角形 10

1.3.2 最长上升子序列 11

1.3.3 最长公共子序列 13

1.3.4 字符串编辑距离 13

1.4 区间 dp 14

1.4.1 合并石子 14

1.5 计数 dp 16

1.5.1 整数划分 16

1.6 数位统计 dp 17

1.6.1 数位统计 17

第二章 字符串专题 **21**

2.0.1 KMP 21

2.0.2 Manacher 算法 22

第三章 进阶数据结构 **25**

3.0.1 ST 表 25

3.0.2 字典树 26

3.0.3 AC 自动机 27

3.0.4 树状数组 29

3.0.5 线段树 30
3.0.6 Splay 树 33

第一章　动态规划

1.1　如何考虑动态规划

1.1.1　考虑状态

如何考虑题目中的状态？应该首先考虑如下内容：

- 集合，也就是包含哪些元素
- 属性，这些元素具有的属性：最大值、最小值、数量……

其中，我们重点考虑集合的表示：首先我们需要确定集合到底是什么，原本集合的元素？选法？还是值？然后，我们要确定我们的维度的含义。常见的 $f[i][j]$ 可以表示：

- 下标不超过 i，总属性也不超过 j 的各种选法
- 左边界在 i，右边界在 j 的元素

等等，以此类推，需要做题积累经验。

1.1.2　考虑状态转移

考虑特定的某一个状态，讨论所有转移到该状态的前置状态。此时，我们进行的是集合的划分，将当前的集合划分为若干更小的集合，使得前面计算过的集合能够用于计算当前的状态。需要注意，我们不一定只能通过循环求出结果，也可以使用递归的方法，也就是记忆化搜索。

如何得到状态递推？或者说，如何找一些前置集合，使得能够产生递推关系？我们可以考虑比较小规模的问题，然后进行分类讨论，以此得到一些不变的性质；或者，我们可以拿出代表对象，通过对对象位置、大小等性质的讨论和比较进行分类，从而得到更小的子问题。总而言之，我们需要找到当前比当前维度更小的子问题，这一问题通常是通过数学计算和分类讨论得到的。

如果需要维护最大值，我们首先要将其旧值进行比较。对于背包问题，我们常常是讨论选不选当前物品，进而将不选对应到上一个 $f[i-1][j]$，将选该物品对应到 $f[i-1][j-w[i]]+w[i]$。在其它问题中也可以类似地找到类似的讨论对象。例如在数位 dp 中，上一个选择会影响下一

个选择，但是我们只需要考虑当前选择对于选择之前状态的限制以及影响，就可以很好的找到递推公式。

1.2 背包 DP

1.2.1 0-1 背包问题

```
//二维版本
int knapsack(vector<int>& weights,vector<int>& values, int W)
{
int n = weights.size();
//初始化略
vector<vector<int>> dp(W + 1, vector<int>(n));
for (int i = 0; i < n; ++i) {
    for (int j = weights[i]; j <= W; j++) {
        dp[i][j]=max(dp[i][j],dp[i-1][j-weights[i-1]]+values[i-1]);
    }
}
return dp[W][n - 1];
}
```

```
//一维版本
int knapsack(vector<int>& weights,vector<int>& values, int W)
{
int n = weights.size();
vector<int> dp(W + 1, 0);
for (int i = 0; i < n; ++i) {
    for (int w = W; w >= weights[i]; --w) {
        //逆序遍历保证每一个物品只选择一遍
        dp[w] = max(dp[w],dp[w - weights[i]] + values[i]);
    }
}
return dp[W];
}
```

1.2.2 完全背包问题

```
//枚举版本
int knapsack(vector<int>& weights,vector<int>& values, int W)
{
int n = weights.size();
//初始化略
vector<vector<int>> dp(W + 1, vector<int>(n + 1));
for (int i = 1; i <= n; ++i) {
    for (int j = 0; j <= W; ++j) {
        for(int k = 0; k * weights[i] <= j; ++k)
            dp[i][j]=max(dp[i][j],dp[i][j-k * weights[i]]+values[i] * k);
    }
}
return dp[W][n];
}
```

注意到，$dp[i][j] = MAX(dp[i-1][j], dp[i-1][j-w]+v, dp[i-1][j-2w]+2v, \cdots, dp[i-1][j-mw]+mv)$ 中，后面的 $dp[i-1][j-w]+v, dp[i-1][j-2w]+2v, \cdots, dp[i-1][j-mw]+mv$ 与 $dp[i][j-w]$ 十分相似，如下：

$$dp[i][j-w] = MAX(dp[i-1][j-w], \cdots, dp[i-1][j-(m-1)w]+(m-1)v)$$

与上式仅差一个 v，而同时加一个值不影响最大值，从而注意到

$$dp[i][j] = max(dp[i-1][j], dp[i-1][j-w]+v)$$

综上有优化代码：

```
//优化版本
int knapsack(vector<int>& weights,vector<int>& values, int W)
{
int n = weights.size();
//初始化略
vector<vector<int>> dp(W + 1, vector<int>(n + 1));
for (int i = 1; i <= n; ++i) {
    for (int j = 0; j <= W; ++j) {
      dp[i][j] = dp[i-1][j];
      if(j >= weights[i])dp[i][j]=max(dp[i][j],dp[i][j-weights[i]]+values[i]);
    }
}
return dp[W][n];
}
```

1.2.3 多重背包问题

```
//枚举版本
int knapsack(vector<int>& num, vector<int>& weights,vector<int>& values, int
    W)
{
int n = weights.size();
//初始化略
vector<vector<int>> dp(W + 1, vector<int>(n + 1));
for (int i = 1; i <= n; ++i) {
    for (int j = 0; j <= W; ++j) {
        for(int k = 0; k * weights[i] <= j && k <= num[i]; ++k)
            dp[i][j]=max(dp[i][j],dp[i][j-k * weights[i]]+values[i] * k);
    }
}
return dp[W][n];
}
```

我们尝试优化上面的代码，注意到 $dp[i][j] = MAX(dp[i-1][j], dp[i-1][j-w]+v, \cdots, dp[i-1][j-s[i]*w]+s[i]*v)$ 以及 $dp[i][j-w] = MAX(dp[i-1][j-w], \cdots, dp[i-1][j-s[i]*w]+(s[i]-1)*v, dp[i-1][j-(s[i]+1)*w]+s[i]*v)$ 这里，与完全背包不同的是，我们的第二个式子最后会多出一项 $dp[i-1][j-(s[i]+1)*w]+s[i]*v$，这是因为，此时我们有选择的上限，使得我们无法对齐尾部这一项（第一个式子的物品数量不能再加一以对齐了）。因此，这样优化是不行的。所以我们要考虑其它的办法。

我们考虑倍增法，考虑小于等于 s[i] 的每个数的二进制编码，我们将“选 k 个物品”转化为“在数个二进制位中选任意个，且表示的这个数的大小不超过 s[i]”，每一个位的“价值”就是二的任意次幂，这样我们就可以将问题转化为“二进制位的 0-1 背包问题”，而这个二进制位的长度是 $log(max(s[i]))$，从而我们避免了从 0 到 s[i] 枚举每一个数字。

```
#include<iostream>
using namespace std;
const int N=11010,M=2010;
int n,m;
int v[N],w[N];
int f[M];
int main()
{
    cin>>n>>m;
```

```
    int cnt=1;
    for(int i=1;i<=n;i++){//拆分打包
        int a,b,s;
        cin>>a>>b>>s;
        int k=1;
        while(k<=s){
            v[cnt]=a*k;
            w[cnt]=b*k;
            s-=k;
            k*=2;
            cnt++;//将s[i]个物品拆分为1个，2个，4个……分别打包为一个物品
            //最终就是在打包好的物品中进行0-1背包问题，从而保证了正确性，因为二进制
            //可以表示小于编码长度的所有整数
        }
        if(s>0){
            v[cnt]=s*a;
            w[cnt]=s*b;
            cnt++;
        }
    }
    for(int i=1;i<=cnt;i++){
        for(int j=m;j>=v[i];j--)
        {
            f[j]=max(f[j],f[j-v[i]]+w[i]);
        }
    }
    cout<<f[m]<<endl;
    return 0;
}
```

1.2.4 分组背包问题

```
int knapsack(vector<vector<int>>& weights, vector<vector<int>>& values, vector<int>& nums,int W)
{
int n = weights.size();
//初始化略
```

```
vector<int> dp(W + 1);
for (int i = 1; i <= n; ++i) {
    for (int j = W; j >= 0; --j) {
        for(int k = 0; k < nums[i]; ++k){
            if(weights[i][k] <= j)dp[j]=max(dp[j],dp[j-weights[i][k]]+values[i
                ][k]);
        }//要么从该组选一个，要么不选
    }
}
return dp[W];
}
```

1.3 线性 dp

1.3.1 数字三角形

数字三角形就是典型的移动类型问题，只需要计算出上一个位置的最大特征，就可以递推下一个状态。

```
#include <iostream>
using namespace std;
const int N = 510;
int n;
int a[N][N];
int f[N][N];
int main () {
    cin >> n;
    for (int i = 1;i <= n;i++) {
        for (int j = 1;j <= i;j++) cin >> a[i][j];
    }
    for (int i = 1;i <= n;i++) f[n][i] = a[n][i];
    for (int i = n - 1;i >= 1;i--) {
        for (int j = 1;j <= i;j++) {
            f[i][j] = max (f[i + 1][j],f[i + 1][j + 1]) + a[i][j];
        }
    }
    cout << f[1][1] << endl;
    return 0;
}
```

1.3.2 最长上升子序列

```cpp
//朴素dp
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        int l=nums.size();
        vector<int>dp(l);//dp[i]表示以i结尾的递增序列长度
        dp[0]=1;
        for(int i=0;i<l;i++){
            dp[i]=1;
            for(int j=0;j<i;j++){
                if(nums[i]>nums[j]){
                    dp[i]=max(dp[i],dp[j]+1);
                }
            }
        }
        int MAX=INT_MIN;
        for(int i=0;i<l;i++){
            MAX=max(dp[i],MAX);
        }
        return MAX;
    }
};
```

```cpp
//线段树优化，解决间隔不超过k的递增最长子序列
class Solution {
    vector<int> max;

    void modify(int o, int l, int r, int i, int val) {
        if (l == r) {
            max[o] = val;
            return;
        }
        int m = (l + r) / 2;
        if (i <= m) modify(o * 2, l, m, i, val);
```

```
12          else modify(o * 2 + 1, m + 1, r, i, val);
13          max[o] = std::max(max[o * 2], max[o * 2 + 1]);
14      }
15
16      // 返回区间 [L,R] 内的最大值
17      int query(int o, int l, int r, int L, int R) { // L 和 R 在整个递归过程中
            均不变，将其大写，视作常量
18          if (L <= l && r <= R) return max[o];
19          int res = 0;
20          int m = (l + r) / 2;
21          if (L <= m) res = query(o * 2, l, m, L, R);
22          if (R > m) res = std::max(res, query(o * 2 + 1, m + 1, r, L, R));
23          return res;
24      }
25
26  public:
27      int lengthOfLIS(vector<int> &nums, int k) {
28          int u = *max_element(nums.begin(), nums.end());
29          max.resize(u * 4);
30          for (int x: nums) {
31              if (x == 1) modify(1, 1, u, 1, 1);
32              else {
33                  int res = 1 + query(1, 1, u, std::max(x - k, 1), x - 1);
34                  modify(1, 1, u, x, res);
35              }
36          }
37          return max[1];
38      }
39  };
```

1.3.3 最长公共子序列

```
class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {
        int l=text1.length();
        int r=text2.length();
        int dp[l+1][r+1];//dp[i][j]表示，第一个字符串前i个和第二个字符串前j个字母的最长公共序列
        for(int i=0;i<=l;i++)
        {
            dp[i][0]=0;
        }
        for(int i=0;i<=r;i++)
        {
            dp[0][i]=0;
        }
        for(int i=1;i<=l;i++)
        {
            for(int j=1;j<=r;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[l][r];
    }
};
```

1.3.4 字符串编辑距离

```
class Solution {
public:
    int minDistance(string word1, string word2) {
        int n = word1.length();
        int m = word2.length();

        // 有一个字符串为空串
```

```
        if (n * m == 0) return n + m;

        // DP 数组
        vector<vector<int>> D(n + 1, vector<int>(m + 1));

        // 边界状态初始化
        for (int i = 0; i < n + 1; i++) {
            D[i][0] = i;
        }
        for (int j = 0; j < m + 1; j++) {
            D[0][j] = j;
        }

        // 计算所有 DP 值
        // 对于A的前i个字符和B的前j个字符，若最后一个字符相同，那么直接用前i-1和
            前j-1的编辑距离
        // 如果最后一个字符不同，那么只需要修改一个字符即可
        // 对于其它情况，至少需要增加一次编辑距离（也就是插入一个字符）
        for (int i = 1; i < n + 1; i++) {
            for (int j = 1; j < m + 1; j++) {
                int left = D[i - 1][j] + 1;
                int down = D[i][j - 1] + 1;
                int left_down = D[i - 1][j - 1]; //暂存防止意外改变原始值
                if (word1[i - 1] != word2[j - 1]) left_down += 1;
                D[i][j] = min({left, down, left_down});

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

Remark 1.3.1. 可以发现，两个字符串类型的 dp 问题通常可以使用“A 的前 i 个 XXX 和 B 的前 j 个 XXX”作为状态进行递推。

1.4 区间 dp

1.4.1 合并石子

```cpp
#include <iostream>
#include <cstring>
using namespace std;
const int N = 307;

int a[N], s[N];
int f[N][N];

int main() {
    int n;
    cin >> n;

    for (int i = 1; i <= n; i ++) {
        cin >> a[i];
        s[i] += s[i - 1] + a[i];
    }

    memset(f, 0x3f, sizeof f);
    // 区间 DP 枚举套路：长度+左端点
    for (int len = 1; len <= n; len ++) { // len表示[i, j]的元素个数
        for (int i = 1; i + len - 1 <= n; i ++) {
            int j = i + len - 1; // 自动得到右端点
            if (len == 1) {
                f[i][j] = 0; // 边界初始化
                continue;
            }

            for (int k = i; k <= j - 1; k ++) { // 必须满足k + 1 <= j
                f[i][j] = min(f[i][j], f[i][k] + f[k + 1][j] + s[j] - s[i - 1]);
            }
        }
    }
    cout << f[1][n] << endl;
    return 0;
}
```

1.5 计数 dp

1.5.1 整数划分

```cpp
// f[i][j] = f[i - 1][j] + f[i][j - i]
#include <iostream>

using namespace std;

const int N = 1e3 + 7, mod = 1e9 + 7;

int f[N][N];

int main() {
    int n;
    cin >> n;

    for (int i = 0; i <= n; i ++) {
        f[i][0] = 1; // 容量为0时，前 i 个物品全不选也是一种方案
    }

    for (int i = 1; i <= n; i ++) {
        for (int j = 0; j <= n; j ++) {
            f[i][j] = f[i - 1][j] % mod; // 特殊 f[0][0] = 1
            if (j >= i) f[i][j] = (f[i - 1][j] + f[i][j - i]) % mod;
        }
    }

    cout << f[n][n] << endl;
}
```

```cpp
#include <iostream>

using namespace std;

const int N = 1e3 + 7, mod = 1e9 + 7;

int f[N];
```

```
8
9   int main() {
10      int n;
11      cin >> n;
12
13
14      f[0] = 1; // 容量为0时，前 i 个物品全不选也是一种方案
15
16      for (int i = 1; i <= n; i ++) {
17          for (int j = i; j <= n; j ++) {
18              f[j] = (f[j] + f[j - i]) % mod;
19          }
20      }
21
22      cout << f[n] << endl;
23  }
```

实际上，这就是著名的分拆数的计算方法。目前没有一个闭式公式用于计算这个数，但是可以通过渐进的方式估计该数的大小。

1.6 数位统计 dp

1.6.1 数位统计

关键在于分情况讨论，根据题目中的限制条件列出可能的情况进行递归。不妨定义以下两种情况

- Dominate（压制）：高位恒小于上限，低位任取；
- Equal（等于）：高位等于上限，低位讨论；

```
1   #include <iostream>
2   #include <algorithm>
3   #include <cmath>
4
5   using namespace std;
6
7   int get(int n) { // 求数n的位数
8       int res = 0;
9       while (n) res ++, n /= 10;
```

```
10      return res;
11  }
12
13  int count(int n, int i) { // 求从1到数n中数i出现的次数
14      int res = 0, dgt = get(n);
15
16      for (int j = 1; j <= dgt; ++ j) {
17      /* p为当前遍历位次(第j位)的数大小 <10^(右边的数的位数)>, Ps:从左往右(从高位
            到低位)
18          l为第j位的左边的数，r为右边的数，dj为第j位上的数 */
19          int p = pow(10, dgt - j), l = n / p / 10, r = n % p, dj = n / p % 10;
20
21      // ps:下文的xxx、yyy均只为使读者眼熟，并不严格只是三位数啊~ 然后后续的...就代
            表省略的位数啦~
22          /* 求要选的数在i的左边的数小于l的情况:
23  (即视频中的xxx1yyy中的xxx的选法) --->
24      1)、当i不为0时 xxx : 0...0 ~ l - 1, 即 l * (右边的数的位数) == l * p 种选
            法
25      2)、当i为0时 由于不能有前导零 故xxx: 0....1 ~ l - 1,
26                      即 (l-1) * (右边的数的位数) == (l-1) * p 种选法 */
27          if (i) res += l * p;
28          else res += (l - 1) * p;
29
30          /* 求要选的数在i的左边的数等于l的情况: (即视频中的xxx == l 时)
31      (即视频中的xxx1yyy中的yyy的选法) --->
32                          1)、i > dj时 0种选法
33                          2)、i == dj时 yyy : 0...0 ~ r 即 r + 1 种选法
34                          3)、i < dj时 yyy : 0...0 ~ 9...9 即 10^(右边的数的位数)
                                == p 种选法 */
35          if (i == dj) res += r + 1;
36          if (i < dj) res += p;
37      }
38
39      return res; // 返回结果
40  }
41
42  int main() {
43      int a, b;
44      while (cin >> a >> b, a) { // 输入处理，直到输入为0停止
```

```
45          if (a > b) swap(a, b); // 预处理-->让a为较小值，b为较大值
46          for (int i = 0; i <= 9; ++ i) cout << count(b, i) - count(a - 1, i) <<
                '␣';
47          // 输出每一位数字(0 ~ 9)分别在[a,b]中出现的次数<利用前缀和思想：[l,r]的
                和=s[r] - s[l - 1]>
48          cout << endl; //换行
49      }
50
51      return 0;
52  }
```

第二章　字符串专题

2.0.1　KMP

```cpp
// 在文本串 text 中查找模式串 pattern，返回所有成功匹配的位置（pattern[0] 在 text 中的下标）
vector<int> kmp(const string& text, const string& pattern) {
    int m = pattern.size();
    vector<int> pi(m);
    int cnt = 0;
    for (int i = 1; i < m; i++) {
        char b = pattern[i];
        while (cnt && pattern[cnt] != b) {
            cnt = pi[cnt - 1];
        }
        if (pattern[cnt] == b) {
            cnt++;
        }
        pi[i] = cnt;
    }

    vector<int> pos;
    cnt = 0;
    for (int i = 0; i < text.size(); i++) {
        char b = text[i];
        while (cnt && pattern[cnt] != b) {
            cnt = pi[cnt - 1];
        }
        if (pattern[cnt] == b) {
            cnt++;
        }
        if (cnt == m) {
            pos.push_back(i - m + 1);
```

```
            cnt = pi[cnt - 1];
        }
    }
    return pos;
}
```

2.0.2 Manacher 算法

本算法用于高效解决回文串判定。

```
class Solution {
public:
    int expand(const string& s, int left, int right) {
        while (left >= 0 && right < s.size() && s[left] == s[right]) {
            --left;
            ++right;
        }
        return (right - left - 2) / 2;
    }

    string longestPalindrome(string s) {
        int start = 0, end = -1;
        string t = "#";
        for (char c: s) {
            t += c;
            t += '#';
        }
        t += '#';
        s = t;

        vector<int> arm_len;
        int right = -1, j = -1;
        for (int i = 0; i < s.size(); ++i) {
            int cur_arm_len;
            if (right >= i) {
                int i_sym = j * 2 - i;
                int min_arm_len = min(arm_len[i_sym], right - i);
                cur_arm_len = expand(s, i - min_arm_len, i + min_arm_len);
            } else {
```

```
30                cur_arm_len = expand(s, i, i);
31            }
32            arm_len.push_back(cur_arm_len);
33            if (i + cur_arm_len > right) {
34                j = i;
35                right = i + cur_arm_len;
36            }
37            if (cur_arm_len * 2 + 1 > end - start) {
38                start = i - cur_arm_len;
39                end = i + cur_arm_len;
40            }
41        }
42
43        string ans;
44        for (int i = start; i <= end; ++i) {
45            if (s[i] != '#') {
46                ans += s[i];
47            }
48        }
49        return ans;
50    }
51 };
```

第三章　进阶数据结构

3.0.1　ST 表

```
#include<cstdio>
#include<cmath>
using namespace std;
const int N=1e5+5;
int n,m;
int a[N];
int st[N][25];
int main()
{
        scanf("%d%d",&n,&m);
        for(int i=1;i<=n;i++)scanf("%d",&a[i]);//读入数据
        for(int i=1;i<=n;i++)st[i][0]=a[i];//初始化，区间长度为0
        for(int j=1;1<<j<=n;j++){ //j的边界就是右边
                for(int i=1;i+(1<<j)<=n+1;i++){ //i+j的边界是i+2^{j-1}+1
                        st[i][j]=max(st[i][j-1],st[i+(1<<(j-1))][j-1]);//转移处理
                            ST表
        }
    }//读入预处理阶段，构建ST表
        while(m--)
        {
                int l,r;
                scanf("%d%d",&l,&r);
                int k=__lg(r-l+1);//将区间长度分拆为两个2^k
                printf("%d\n",max(st[l][k],st[r-(1<<k)+1][k])); //利用公式计算出
                    左端点
        }
        return 0;
}
```

3.0.2 字典树

```
struct Node {
    Node* son[26]{};
    bool end = false;
};

class Trie {
    Node* root = new Node();

    int find(string word) {
        Node* cur = root;
        for (char c : word) {
            c -= 'a';
            if (cur->son[c] == nullptr) { // 道不同，不相为谋
                return 0;
            }
            cur = cur->son[c];
        }
        // 走过同样的路（2=完全匹配，1=前缀匹配）
        return cur->end ? 2 : 1;
    }

    void destroy(Node* node) {
        if (node == nullptr) {
            return;
        }
        for (Node* son : node->son) {
            destroy(son);
        }
        delete node;
    }

public:
    ~Trie() {
        destroy(root);
    }

    void insert(string word) {
```

```
38          Node* cur = root;
39          for (char c : word) {
40              c -= 'a';
41              if (cur->son[c] == nullptr) { // 无路可走?
42                  cur->son[c] = new Node(); // new 出来!
43              }
44              cur = cur->son[c];
45          }
46          cur->end = true;
47      }
48
49      bool search(string word) {
50          return find(word) == 2;
51      }
52
53      bool startsWith(string prefix) {
54          return find(prefix) != 0;
55      }
56  };
```

3.0.3 AC 自动机

AC 自动机是多字符串匹配的经典算法，是 KMP 思想在多字符上的运用。实际上，KMP 就是链表形式的 AC 自动机。事实上，我们在这个算法中的核心目的，就是找到当前失配字符串后缀对应的最长前缀。该算法的关键在于建立 fail 数组，其中，某一个结点 x 的 fail 数组为：x->fa->fail->(若存在 fa 到 x 的字母，x 的 fail 指向它)-> 否则继续查找 fail。

为了压缩路径，我们可以直接设置转移表，每一个结点对应读到字符的转移路径。如何设置呢？我们只需要将当前节点的 fail 结点读到字符的 fail 转移结点一致即可。

```
1   typedef struct TrieNode {
2       vector<TrieNode *> children;
3       bool isEnd;
4       TrieNode *fail;
5       TrieNode() {
6           this->children = vector<TrieNode *>(26, nullptr);
7           this->isEnd = false;
8           this->fail = nullptr;
9       }
```

```
};

class StreamChecker {
public:
    TrieNode *root;
    TrieNode *temp;
    StreamChecker(vector<string>& words) {
        root = new TrieNode();
        for (string &word : words) {
            TrieNode *cur = root;
            for (int i = 0; i < word.size(); i++) {
                int index = word[i] - 'a';
                if (cur->children[index] == nullptr) {
                    cur->children[index] = new TrieNode();
                }
                cur = cur->children[index];
            }
            cur->isEnd = true;
        }
        root->fail = root;
        queue<TrieNode *> q;
        for (int i = 0; i < 26; i++) {
            if(root->children[i] != nullptr) {
                root->children[i]->fail = root;
                q.emplace(root->children[i]);
            } else {
                root->children[i] = root;
            }
        }
        while (!q.empty()) {
            TrieNode *node = q.front();
            q.pop();
            node->isEnd = node->isEnd || node->fail->isEnd;
            for (int i = 0; i < 26; i++) {
                if(node->children[i] != nullptr) {
                    node->children[i]->fail = node->fail->children[i];
                    q.emplace(node->children[i]);
                } else {
                    node->children[i] = node->fail->children[i];
```

```
                }
            }
        }

        temp = root;
    }

    bool query(char letter) {
        temp = temp->children[letter - 'a'];
        return temp->isEnd;
    }
};
```

3.0.4 树状数组

```
class NumArray {
private:
    vector<int> nums;
    vector<int> tree;

    int prefixSum(int i) {
        int s = 0;
        for (; i > 0; i &= i - 1) { // i -= i & -i 的另一种写法，等价于lowbit;
            访问上一节点，如i=13，则迭代i=13-1 = 12，恰好是9-12这一段;
                                    //继续访问，i=12-4=8，恰好是最左边的一段
            s += tree[i];
        }
        return s;
    }

public:
    NumArray(vector<int> &nums) : nums(nums), tree(nums.size() + 1) {
        for (int i = 1; i <= nums.size(); i++) {
            tree[i] += nums[i - 1];
            int nxt = i + (i & -i); // 下一个关键区间的右端点，例如3，对应地就是
                3+(3&-3)=3+1=4
            if (nxt <= nums.size()) {
                tree[nxt] += tree[i];//关键区间加上当前位置的数
```

```
            }
        }
    }//倍增法，用等长数组维护区间信息，访问下一关键结点，迭代i=i+lowbit(i)即可

    void update(int index, int val) {
        int delta = val - nums[index];//储存更新差值，从下到上传递
        nums[index] = val;//先更新值
        for (int i = index + 1; i < tree.size(); i += i & -i) {
            tree[i] += delta;//注意，迭代从index + 1开始，因为原数组从零开始
        }
    }

    int sumRange(int left, int right) {
        return prefixSum(right + 1) - prefixSum(left);
    }
};
```

3.0.5 线段树

```
template<typename T>
class SegmentTree {
    // 注：也可以去掉 template<typename T>，改在这里定义 T
    // using T = pair<int, int>;

    int n;
    vector<T> tree;

    // 合并两个 val
    T merge_val(T a, T b) const {
        return max(a, b); // **根据题目修改**
    }

    // 合并左右儿子的 val 到当前节点的 val
    void maintain(int node) {
        tree[node] = merge_val(tree[node * 2], tree[node * 2 + 1]);
    }

    // 用 a 初始化线段树
```

```
    // 时间复杂度 O(n)
    void build(const vector<T>& a, int node, int l, int r) {
        if (l == r) { // 叶子
            tree[node] = a[l]; // 初始化叶节点的值
            return;
        }
        int m = (l + r) / 2;
        build(a, node * 2, l, m); // 初始化左子树
        build(a, node * 2 + 1, m + 1, r); // 初始化右子树
        maintain(node);
    }

    void update(int node, int l, int r, int i, T val) {
        if (l == r) { // 叶子（到达目标）
            // 如果想直接替换的话，可以写 tree[node] = val
            tree[node] = merge_val(tree[node], val);
            return;
        }
        int m = (l + r) / 2;
        if (i <= m) { // i 在左子树
            update(node * 2, l, m, i, val);
        } else { // i 在右子树
            update(node * 2 + 1, m + 1, r, i, val);
        }
        maintain(node);
    }

    T query(int node, int l, int r, int ql, int qr) const {
        if (ql <= l && r <= qr) { // 当前子树完全在 [ql, qr] 内
            return tree[node];
        }
        int m = (l + r) / 2;
        if (qr <= m) { // [ql, qr] 在左子树
            return query(node * 2, l, m, ql, qr);
        }
        if (ql > m) { // [ql, qr] 在右子树
            return query(node * 2 + 1, m + 1, r, ql, qr);
        }
        T l_res = query(node * 2, l, m, ql, qr);
```

```
59          T r_res = query(node * 2 + 1, m + 1, r, ql, qr);
60          return merge_val(l_res, r_res);
61      }
62
63  public:
64      // 线段树维护一个长为 n 的数组（下标从 0 到 n-1），元素初始值为 init_val
65      SegmentTree(int n, T init_val) : SegmentTree(vector<T>(n, init_val)) {}
66
67      // 线段树维护数组 a
68      SegmentTree(const vector<T>& a) : n(a.size()), tree(2 << bit_width(a.size
            () - 1)) {
69          build(a, 1, 0, n - 1);
70      }
71
72      // 更新 a[i] 为 merge_val(a[i], val)
73      // 时间复杂度 O(log n)
74      void update(int i, T val) {
75          update(1, 0, n - 1, i, val);
76      }
77
78      // 返回用 merge_val 合并所有 a[i] 的计算结果，其中 i 在闭区间 [ql, qr] 中
79      // 时间复杂度 O(log n)
80      T query(int ql, int qr) const {
81          return query(1, 0, n - 1, ql, qr);
82      }
83
84      // 获取 a[i] 的值
85      // 时间复杂度 O(log n)
86      T get(int i) const {
87          return query(1, 0, n - 1, i, i);
88      }
89  };
90
91  int main() {
92      SegmentTree t(8, 0LL); // 如果这里写 0LL，那么 SegmentTree 存储的就是 long
            long 数据
93      t.update(0, 1LL << 60);
94      cout << t.query(0, 7) << endl;
95
```

```
96      vector<int> nums = {3, 1, 4, 1, 5, 9, 2, 6};
97      // 注意: 如果要让 SegmentTree 存储 long long 数据, 需要传入 vector<long
            long>
98      SegmentTree t2(nums); // 这里 SegmentTree 存储的是 int 数据
99      cout << t2.query(0, 7) << endl;
100     return 0;
101 }
```

3.0.6 Splay 树