

# 算法竞赛入门基础

电气信息系 计算机科学与技术17级 陈 巍

## C、C++中常用函数和STL

### 第一部分 基础算法

#### 1.1 快速排序

##### 1.1.1 算法流程及时间复杂度分析

算法流程

- 1.确定排序对象，排序区间：（确定函数的形参）
- 2.确定分界点  $x = q[(l + r) / 2]$  ,和需要递归处理的两个边界点  $i = l - 1, j = r + 1$
- 3.循环处理分界点  $x$  两侧的不符合条件的数字交换处理
- 4.递归左右两个区间  $[l, j]$  和  $[j + 1, r]$

时间复杂度

$O(n \log n)$

##### 1.1.2 题目及相关代码

题目：

[AcWing785.快速排序](#)

代码示例：

```
1 void quick_sort(int q[], int l, int r)
2 {
3     if (l >= r) return ;    // 递归出口（排序完毕）
4     // 确定分界点x和边界点ij
5     int x = q[(l + r) / 2];
6     int i = l - 1, j = r + 1;
7     while (i < j)
8     {
9         do i++; while (q[i] < x);    // 找到x左侧不符合条件的值并记录下标i
10        do j--; while (q[j] > x);    // 找到x右边不符合条件的值并记录下标j
11        if (i < j) swap(q[i], q[j]);    // 交换x左右两侧的值
12        else break;
13    }
14    // 递归处理
15    quick_sort(q, l, j), quick_sort(q, j + 1, r);
16 }
```

简化版：

```
1 void quick_sort(int q[], int l, int r)
2 {
3     if (l >= r) return ;
4 }
```

```

5     int x = q[(l + r) / 2];
6     int i = l - 1, j = r + 1;
7
8     while (i < j)
9     {
10         while (q[++i] < x);
11         while (q[--j] > x);
12         if (i < r) swap(q[i], q[j]);
13         else break;
14     }
15
16     quick_sort(q, l, j), quick_sort(q, j + 1, r);
17 }

```

注意：

排序的区间范围；

分解点的确定；

递归调用时使用的参数。

### 2.1.3 运用STL中的sort函数

```

1  #include <algorithm>      // include算法库之后才能够使用sort函数
2  sort(q, q + n); // q是待排序的数组,排序区间下标[0, n); 形参中q是数组的地址
3  /*
4  * 注意：必要时，定义 bool cmp() 或者 重载操作符 <
5  */

```

### 2.1.4 相关题目

[AcWing786.第K个数](#)

## 1.2 归并排序

### 1.2.1 算法流程及时间复杂度分析

算法流程：

0.确定递归出口

1.划分（递归）。找到分解点  $mid = (l + r) / 2$ ，然后根据分解点进行递归划分。

2.合并。

首先，确定三个参数  $i$ 、 $j$ 、 $k$ ，其中  $i = l$ 、 $j = mid + 1$  分别表示划分后的两个区间  $[l, mid]$  和  $[mid + 1, r]$  的起始下标， $k = 0$  表示 `tmp` 数组的其实下标；

然后，比较划分后的两个数组中数值大小，按照升序存放到 `tmp` 数组中；

最后，将存放升序结果的 `tmp` 数组 中的值，转存回原数组。

时间复杂度分析：

$O(n \log n)$

### 1.2.2 题目及相关代码

题目：

[AcWing787.归并排序](#)

代码示例：

```
1 void merge_sort(int q[], int l, int r)
2 {
3     // 递归出口
4     if (l >= r) return ;
5
6     // 划分
7     int mid = (l + r) / 2;
8     merge_sort(q, l, mid), merge_sort(q, mid + 1, r);
9
10    // 归并
11    int i = l, j = mid + 1;    // 划分后的两个需要进行比较的数组的起始位置的下标ij
12    int k = 0;    // 数组tmp的起始下标k
13    // 比较划分后的数组的下标对应的值大小，按照升序存到tmp中
14    while (i <= mid && j <= r)
15    {
16        if (q[i] < q[j]) tmp[k++] = q[i++];
17        else tmp[k++] = q[j++];
18    }
19    while (i <= mid)
20    {
21        tmp[k++] = q[i++];
22    }
23    while (j <= r)
24    {
25        tmp[k++] = q[j++];
26    }
27    // 将已经顺序排列后的tmp转存到原数组中
28    for (i = l, j = 0; i <= r; ++i, ++j) q[i] = tmp[j];
29 }
```

注意：

递归出口；

先划分后合并。

## 1.2.3 相关题目

[AcWing788.逆序对的数量](#)

代码示例：

```
1 #include <iostream>
2 #include <cstdio>
3
4 using namespace std;
5
6 typedef long long LL;
7
8 const int N = 1e6 + 10;
9 int q[N];
10 int tmp[N];
11 int n;
12
13 LL merge_sort(int q[], int l, int r)
14 {
15     if (l >= r) return 0;
16
17     int mid = l + r >> 1;
```

```

18     LL res = merge_sort(q, l, mid) + merge_sort(q, mid + 1, r);
19
20     while (i <= mid && j <= r)
21     {
22         if (q[i] <= q[j]) tmp[k++] = q[i++];
23         else
24         {
25             res += mid - i + 1;
26             tmp[k++] = q[j++];
27         }
28     }
29
30     while (i <= mid) tmp[k++] = q[i++];
31     while (j <= r) tmp[k++] = q[j++];
32
33     for (i = l, j = 0; i <= r; ++i, ++j) q[i] = tmp[j];
34
35     return res;
36 }
37 int main()
38 {
39     scanf("%d", &n);
40     for (int i = 0; i < n; ++i) scanf("%d", &q[i]);
41     printf("%d\n", merge_sort(q, 0, n - 1));
42 }

```

## 1.3 二分法

### 1.3.1 整数二分

重点：边界问题 处理

本质：

二分和单调性没有直接的关系。只要满足单调性，我们就可以使用二分法；但是，二分法不仅仅适用于单调性的题目。

本质上是，区间上的性质。二分可以将区间一分为二，比如，左半边是满足性质的，右半边是不满足性质的。也就是，根据某种性质，将区间一分为二，一半满足性质，一半不满足性质，用二分法寻找满足性质/条件的边界点。

整数二分结果：

通过两种不同的二分，可以得到左半边的右边界点和右半边的左边界点。

求解左半边的右边界点：

我们要查找的  $x$  在符合条件的区间右边；

算法流程：

- 1) 首先，找到当前区间的中间值  $mid = (l + r + 1) / 2$ ；
- 2) 然后，判断中间值  $mid$  是否满足性质/条件（区间的左半边）；

当中间值  $mid$  满足条件 (True) 的时候，说明中间值  $mid$  在左半边中的某一个位置；左半边的边界点一定在中间值  $mid$  的右边（ $mid$  可能符合条件，当前  $mid$  是可以取到边界点的），所以得到区间  $[mid, r]$ ；更新区间：  $l = mid$ ，更新后的区间正好是  $[mid, r]$ ；

当中间值 `mid` 不满足条件 (`False`) 的时候, 说明中间值 `mid` 在右半边的某一个位置; 由于 `mid` 一定不在左半边 (中间值 `mid` 一定不符合条件), 所以我们得到区间 `[1, mid-1]`; 更新区间 `[1, mid - 1]`;

如果查找的值 `x` 不存在, 找到的则是小于 `x` 的最大值。

代码:

```
1  /*
2      bool check(int mid) { ... }
3  */
4  int binary_search(int l, int r)
5  {
6      while (l < r)
7      {
8          int mid = (l + r + 1) >> 1; // int mid = (l + r + 1) / 2
9          if (check(mid)) // 不写check()直接判断: if (q[mid] >= x)
10             l = mid; // [mid, r]
11          else
12             r = mid - 1; // [l, mid - 1]
13      }
14      return l; // 小于等于x的最大值
15  }
```

思考: 为什么 `mid = (l + r + 1) / 2`?

求解右半边的左边界点:

我们要查找的 `x` 在符合条件的区间最左边;

算法流程:

- 1) 首先, 找中间值 `mid = (l + r) / 2`;
- 2) 然后, 判断中间值 `mid` 是否满足性质/条件 (区间右半边);

当满足 (`True`) 条件时 (`mid` 在区间右半边), 说明 `mid` 在区间右半边的某一个位置, 这时右半边的左边界点在 `mid` 的左侧 (可能被当前的 `mid` 取到); 那么更新后的区间为 `[1, mid]`; 更新 `r = mid`;

当不满足 (`false`) 条件时 (`mid` 在区间左半边), 说明 `mid` 在区间左半边的一个位置, 这时右半边的左边界点在 `mid` 的右侧 (不可能被当前的 `mid` 取到); 那么更新后的区间为 `[mid+1, r]`; 更新 `l = mid + 1`。

如果查找的值 `x` 不存在, 则返回大于 `x` 的最小值

代码示例:

```
1  /*
2      bool check(int mid) { ... }
3  */
4  int binary_search(int l, int r)
5  {
6      while (l < r)
7      {
8          int mid = l + r >> 1; // int mid = (l + r) / 2;
9          if (check(mid))
10             r = mid; // [1, mid]
11          else
12             l = mid + 1; // [mid+1, r]
13      }
14      return l; // 大于等于x的最小值
```

```
15 | }
```

#### 1.3.1.1 两种二分的选择

方法：

首先，不写 `mid` 而是先编写 `if` 中的 `check()` 函数；

然后，根据编写好的 `check()` 函数 来确定更新区间的方式；

最后，根据更新区间的方式，来确定 `mid = l + r >> 1` 还是 `mid = l + r + 1 >> 1`。

#### 1.3.2.1 题目链接及代码

题目链接：

[AcWing789.数的范围](#)

代码示例：

```
1 #include <iostream>
2 #include <cstdio>
3
4 using namespace std;
5
6 int main()
7 {
8
9     return 0;
10 }
```

### 1.3.2 实数二分

## 1.4 前缀和与差分

## 1.5 高精度算法/大整数算法

### 1.5.1 高精度加

$$C = A + B; \quad A \geq 0, B \geq 0$$

题目链接：

[AcWing.791高精度加法](#)

思路：

使用C++的STL中vector实现，模拟加法运算。

算法流程：

主函数中，使用 `string` 类型的变量存放参与运算的数，然后将“参与运算的数”逆序存放到 `vector` 类型的变量中（作为参数）；

加法运算函数，通过设置一个进位变量 `t` 模拟每一位运算后得到的结果；通过 `t % 10` 得到“当前位”的数值，`t /= 10` 得到参与下一位运算的进位值；最后记得处理“多出来的一位”的进位值。

最后，将从“个位到十位百位...”的逆序组成的结果，顺序输出。

注意：

参数是逆序传参数；

如果是非十进制运算，只需要将  $t \% x$  和  $t \backslash x$  中的  $x$  替换成相应的进制即可；

结果是逆序，输出需要顺序输出；

不需要比较  $A$  和  $B$  的大小关系。

代码示例：

```
1 #include <iostream>
2 #include <cstdio>
3 #include <string>
4 #include <cstring>
5 #include <vector>
6
7 using namespace std;
8
9 string a, b;
10 vector<int> A, B, C;
11
12 vector<int> add(vector<int>& A, vector<int>& B)
13 {
14     vector<int> C;
15     int t = 0; // 进位t
16     for (int i = 0; i < A.size() || i < B.size(); ++i)
17     {
18         if (i < A.size()) t += A[i];
19         if (i < B.size()) t += B[i];
20         C.push_back(t % 10); // 得到当前位的值，并存到C中
21         t /= 10; // 得到进位值
22     }
23     // 在运算结束后，处理进位，防止遗漏
24     if (t) C.push_back(t); // 等价 if(t) C.push_back(1);
25     return C;
26 }
27 int main()
28 {
29     cin >> a >> b;
30     // 将每一位逆序存放到vector中，A和B中第一个元素是a和b中最低位的数字
31     for (int i = a.size() - 1; i >= 0; --i) A.push_back(a[i] - '0');
32     for (int j = b.size() - 1; j >= 0; --j) B.push_back(b[j] - '0');
33
34     vector<int> C = add(A, B);
35
36     // 将逆序存放的结果，输出
37     for (int i = C.size() - 1; i >= 0; --i) cout << C[i];
38     puts("");
39     return 0;
40 }
```

## 1.5.2 高精度减法

## 1.5.3 高精度乘法-大整数乘低精度

$$C = A * b, A > 0, b > 0$$

题目链接：

[AcWing.793高精度乘法](#)

思路：

使用C++的STL中vector实现，模拟乘法运算。

算法流程：

主函数，同高精度加法；

模拟乘法函数，注意对进位的处理。

注意：

除了高精度加法外，模拟乘法函数中没有对额外进位进行特别处理，而是通过巧妙的方法在for循环中一起处理了。

代码示例：

```
1  #include <iostream>
2  #include <cstdio>
3  #include <string>
4  #include <cstring>
5  #include <vector>
6
7  using namespace std;
8
9  string a;
10 int b;
11 vector<int> A, C;
12
13 vector<int> mul(vector<int>& A, int b)
14 {
15     vector<int> C;
16     int t = 0;
17     for (int i = 0; i < A.size() || t; ++i)    // 实现了对进位的处理
18     {
19         if (i < A.size()) t += A[i] * b;
20         C.push_back(t % 10);
21         t /= 10;
22     }
23     return C;
24 }
25 int main()
26 {
27     cin >> a >> b;
28     for (int i = a.size() - 1; i >= 0; --i) A.push_back(a[i] - '0');
29
30     C = mul(A, b);
31
32     for (int i = C.size() - 1; i >= 0; --i) cout << C[i];
33     cout << endl;
34     return 0;
35 }
```

#### 1.5.4 高精度除法

## 第二部分 数据结构



## 2.1 链表

### 2.1.1 单链表

#### 2.1.1.1 相关操作、算法流程及代码实现：

相关操作和算法流程：

1. 初始化单链表
2. 插入元素
  1. 插入头结点x
  2. 在结点k的后面插入结点x
2. 删除元素
  1. 删除结点k后面的一个结点
3. 删除头结点

代码实现：

```
1  /*
2     head表示头指针；
3     e[i]表示编号为i的结点的值；
4     ne[i]表示编号为i的下一个元素的next指针；
5     idx表示临时指针,当前使用的结点编号.
6  */
7  const int N = 1e6 + 10;
8  int head, e[N], ne[N], idx;
9
10 // 初始化单链表
11 void init()
12 {
13     head = -1;    // 表示不指向任何元素
14     idx = 0;      // 从编号0开始存放结点
15 }
16
17 // 插入头结点x: x表示头结点的值
18 void add_to_head(int x)
19 {
20     e[idx] = x;
21     ne[idx] = head;
22     head = idx++;
23 }
24
25 // 插入结点,在编号为k的结点后面插入结点x
26 void add(int k, int x)
27 {
28     e[idx] = x;
29     ne[idx] = h[k];
30     h[k] = idx++;
31 }
32
33 // 删除结点k后面的结点
34 void remove(int k)
35 {
36     ne[k] = ne[ne[k]];
37 }
38
39 // 删除头结点
40 void remove_to_head()
41 {
```

```

42     head = ne[ne[head]];
43 }
44
45 // 遍历单链表并输出
46 for (int i = head; i != -1; i = ne[i])
47 {
48     cout << e[i] << ' ';
49 }
50 cout << endl;

```

#### 2.1.1.2相关题目链接及代码示例：

题目链接：

[AcWing.826单链表](#)

代码示例：

```

1  #include <iostream>
2  #include <cstdio>
3  using namespace std;
4
5  const int N = 1e6 + 10;
6
7  int main()
8  {
9
10     return 0;
11 }

```

## 2.2 栈

## 2.3 队列

## 2.8 并查集

### 2.8.1 并查集功能和原理：

功能：

- 1) 用来判断一个图中是否存在环;
- 2) 将两个集合合并为一个集合;
- 3) 判断两个元素是否在同一个集合中;
- 4) 维护额外信息
- 5) 无向图的连通分量个数、Kruskal算法求最小生成树、最近/最小公共祖先（LCA）

原理：

并查集是一种 **树形数据结构** 又称为 **不相交集** 或 **不相交集的数据结构**; 每一个集合用一颗树来表示。树根的编号就是整个集合的编号。每个节点都存放着自己的父结点（可以路径压缩优化）。

路径压缩-优化：

让集合中的所有结点/路径上的每个结点都指向祖先/根结点，只需要一次遍历，就可以让大部分结点找到根结点，优化结点查找。

基本操作及时间复杂度：

路径压缩-优化后，时间复杂度：接近  $O(1)$

算法流程和相关操作：

首先初始化所有结点的父结点为指向自己；根据要求进行合并操作和查找操作。

```
1 int p[N];    // p[a] = b 表示 结点a的父亲是 结点b
```

求集合的编号/祖先结点/根结点的编号（路径压缩后）：

```
1 while (p[x] != x) x = p[x];
```

Find查找祖先结点 + 路径压缩：

```
1 // 返回结点x的祖先结点编号，即集合的编号
2 int find(int x)
3 {
4     if (p[x] != x) p[x] = find(p[x]);
5     else return p[x];
6 }
```

Union合并操作（利用Find）：

```
1 // 如果ab两个结点不在同一个集合中，将 结点a 所在的集合 合并到 结点b 所在的集合 ： b的祖先
  结点作为 a 的祖先结点的父结点
2
3 if (find(a) != find(b)) p[find(a)] = find(b);
```

题目链接及代码示例：

[AcWing836.合并集合](#)

代码示例：

```
1 #include <iostream>
2 #include <cstdio>
3
4 using namespace std;
5
6 const int N = 1e5 + 10;
7 int n, m;
8 int p[N];
9
10 // 查找操作 + 路径压缩
11 int find(int x)
12 {
13     if (p[x] != x) p[x] = find(p[x]);
14     return p[x];
15 }
16
17
18 int main()
19 {
20     scanf("%d%d", &n, &m);
21     // 初始化所有结点的父结点
22     for (int i = 1; i <= n; ++i) p[i] = i;
```

```

23
24     while (m--)
25     {
26         char op[2];
27         int a, b;
28         scanf("%s%d%d", op, &a, &b);
29
30         if (op[0] == 'M')    // 合并
31         {
32             p[find(a)] = find(b);
33         }
34         else    // 判断 or 查找
35         {
36             if (find(a) != find(b)) puts("No");
37             else puts("Yes");
38         }
39
40     }
41     return 0;
42 }

```

## 2.8.2 连同块中点的数量

# 第三部分 搜索与图论

## 3.0 深度优先遍历DFS 和 宽度优先遍历BFS 简介

### 3.0.1 DFS

DFS是树形结构的 **搜索树**，可以对整个空间进行搜索；

一条路走到黑。

算法流程：

从起点开始，一直往下一层搜索，当遇到叶节点时，**回溯**；

也就是，一直往前走，当走到了尽头再回头；回头之后，看看当前是否可以继续向前走（换一个**树的分叉**）；不能走了之后再回头，再看是否可以向前走；

直到走完 **搜索树** 的所有结点。

### 3.0.2 BFS

BFS同DFS同样是树形结构的搜索树，同样可以对整个空间进行搜索（于DFS的搜索方式不同）；

眼观六路，耳听八方。

算法流程：

从起点开始，先走完搜索树的第一层所有结点；

然后再走完搜索树的第二层结点；

直到走完所有结点。

### 3.0.3 DFS 和 BFS 的对比

数据结构:

DFS : stack 栈

BFS : queue 队列

空间:

DFS :  $O(h)$ ;  $h$  为搜索树的高度

BFS :  $O(2^h)$ ;

DFS 在 "空间复杂度" 上有优势

最短路性质:

DFS : 不具有最短路性质

BFS : 具有最短路性质 (在每一次按层扩展/搜索过程中, 每次都能找到最近的点, 最终能够找到一条最短路径)

BFS 在 "时间复杂度" 上有优势

### 3.0.4 DFS 的难点

回溯原理的理解;

剪枝技巧的使用。

回溯+回复现场: 哪个地方修改了状态, 哪个地方需要恢复现场 (递归函数的位置)

### 3.0.5 BFS 求最短路的条件

图/树的边权为1时, 才能够使用BFS求最短路;

其他情况, 只能使用 最短路算法 求解最短路。

## 3.1 深度优先遍历 DFS

题目链接:

[AcWing842.排列数字](#)

题解:

我们可以将问题转化为, 在三个空格 ( \_ \_ \_ ) 上面填数字;

有三个空位, 从第一个空位开始填数字, 都枚举一遍所有数字, 查找没有使用过的数字, 将没有使用过的数字填到空格中;

注意: 每次都存放当前的路径; 记得回溯和回复现场。

优化: 位运算

关键点:

想清楚 搜索顺序

示例代码:

```
1 #include <iostream>
2 #include <cstdio>
3
4 using namespace std;
5 const int N = 10;
```

```

6  int n;
7  int path[N];    // path[u]表示第u层中存放的数字=>搜索路径中第u个搜索位置
8  bool st[N];    // st[i]表示 i 是否使用过, True 使用过, false 没有使用过
9
10 void dfs(int u)    // u表示当前搜索树的深度
11 {
12     // 当搜索到最后一层, 输出结果
13     if (u == n)
14     {
15         for (int i = 0; i < n; ++i) printf("%d ", path[i]);
16         puts("");
17         return ;
18     }
19     // 搜索, 寻找符合条件的路径 (本层中, 没有被使用过的数字)
20     for (int i = 1; i <= n; ++i)
21     {
22         if(!st[i])
23         {
24             path[u] = i;    // 更新当前层数的数字 (更新被搜索到的数字)
25             st[i] = true;   // 更新 数字i 的状态
26             dfs(u + 1);     // 搜索下一层
27             st[i] = false;  // 回溯 (恢复现场)
28         }
29     }
30 }
31
32 int main()
33 {
34     scanf("%d", &n);
35     dfs(0);
36     return 0;
37 }

```

### 3.1.1 n皇后问题

题目链接:

[AcWing843.n-皇后问题](#)

题目讲解:

dfs + 剪枝技巧

(未剪枝) 从第一行到第n行, 每行皇后放置的列的位置;

(剪枝) 从第一行到第n行, 在摆放每一行的皇后时, 首先判断 当前要摆放皇后的位置 是否和 前几行的皇后位置 产生冲突。如果产生冲突, 停止当前行皇后的摆放, 放弃对后面几个皇后摆放, 停止搜索 (剪枝)。如果没有产生冲突, 继续摆放下一行。

代码示例 (剪枝):

算法流程:

按行摆放, 判断摆放的列位置、正对角线和反对角线位置是否符合摆放条件。

算法细节-正反对角线表示以及判断思路:

正对角线  $dg$ :

对正角线按照 由左上到右下 表示, 从棋盘的左上角算作第一条正对角线;

$$y = -x + b$$

表示列通过截距  $b = y + x$ ，将皇后摆放位置  $(u, i)$  带入到截距公式中，得到

`dg[u + i]`

反对角线  $udg$ ：

反对角线按照 由左上到右下 表示，从棋盘的右上角算作第一条反对角线

$$y = x + b$$

截距公式  $b = y - x$ ，同上，得到 `udg[n - u + i]`

解释  $y - x < 0$  时结果为负数，但是数组的下标没有负数，所以增加 偏移量 `n`。

时间复杂度：

$O(n * n!)$

```
1 #include <iostream>
2 #include <cstdio>
3
4 using namespace std;
5 const int N = 10;
6
7 int n;
8 char g[N][N];    // 棋盘
9 bool col[N];     // 列判断（每一列只能有一个）
10 bool dg[N];      // 两条对角线：正对角线dg（从右上角开始，按照由左上到右下）
11 bool udg[N];     // 反对角线udg（从左上角开始，按照由右上到左下）
12
13 void dfs(int u)
14 {
15     if (u == n)    // 第n行的皇后也摆放完毕，输出结果
16     {
17         for (int i = 0; i < n; ++i) puts(g[i]);
18         puts("");
19         return ;
20     }
21     // 当前第u行皇后的摆放，枚举每一列，找到何时的摆放位置
22     for (int i = 0; i < n; ++i)
23     {
24         // 根据  $y = ax + b$  和  $y = -ax + b$  中截距来判断位置
25         if (!col[i] && !dg[u + i] && !udg[n - u + i]) // 第u行中，第i列没有放过
26             // 而且(u,i)的正对角线和反对角线没有摆放过
27         {
28             g[u][i] = 'Q';
29             col[i] = dg[u + i] = udg[n - u + i] = true;
30             dfs(u + 1);
31             col[i] = dg[u + i] = udg[n - u + i] = false;
32             g[u][i] = '.';
33         }
34     }
35
36 int main()
37 {
38     scanf("%d", &n);
39     for (int i = 0; i < n; ++i)
40         for (int j = 0; j < n; ++j)
41             g[i][j] = '.';
42     dfs(0);
43     return 0;
44 }
```

代码示例（未剪枝）：

算法流程：

枚举每一个格子，每一个格子都有两种选择，摆放皇后，或者不摆放皇后。

时间复杂度：

$O(2^n)$

```
1 #include <iostream>
2 #include <cstdio>
3
4 using namespace std;
5 const int N = 10;
6
7 int n;
8 char g[N][N]; // 棋盘
9 bool row[N]; // 行判断
10 bool col[N]; // 列判断（每一列只能有一个）
11 bool dg[N]; // 两条对角线：正对角线dg（从右上角开始，按照由左上到右下）
12 bool udg[N]; // 反对角线udg（从左上角开始，按照由右上到左下）
13
14 void dfs(int x, int y, int s) // 摆放位置(x,y),当前已经摆放的皇后数量t
15 {
16     if (y == n) y = 0, x++; // 如果当前行按列摆放已经出界，从下一行的第一个位置开始摆
17     if (x == n) // 如果当前行摆放完毕
18     {
19         if (s == n)
20         {
21             for (int i = 0; i < n; ++i) puts(g[i]);
22             puts("");
23         }
24         return ;
25     }
26     dfs(x, y + 1, s); // 当前位置不摆放皇后，判断下一个位置
27
28     // 当前可以摆放皇后
29     if (!row[x] && !col[y] && !dg[x + y] && !udg[n - x + y])
30     {
31         row[x] = col[y] = dg[x + y] = udg[n - x + y] = true;
32         g[x][y] = 'Q';
33         dfs(x, y + 1, s + 1); // 判断下一个位置摆放皇后
34         row[x] = col[y] = dg[x + y] = udg[n - x + y] = false;
35         g[x][y] = '.';
36     }
37 }
38 int main()
39 {
40     scanf("%d", &n);
41     for (int i = 0; i < n; ++i)
42         for (int j = 0; j < n; ++j)
43             g[i][j] = '.';
44     dfs(0, 0, 0);
45     return 0;
46 }
```

## 3.2 宽度优先遍历BFS



### 3.2.1 走迷宫

题目链接：

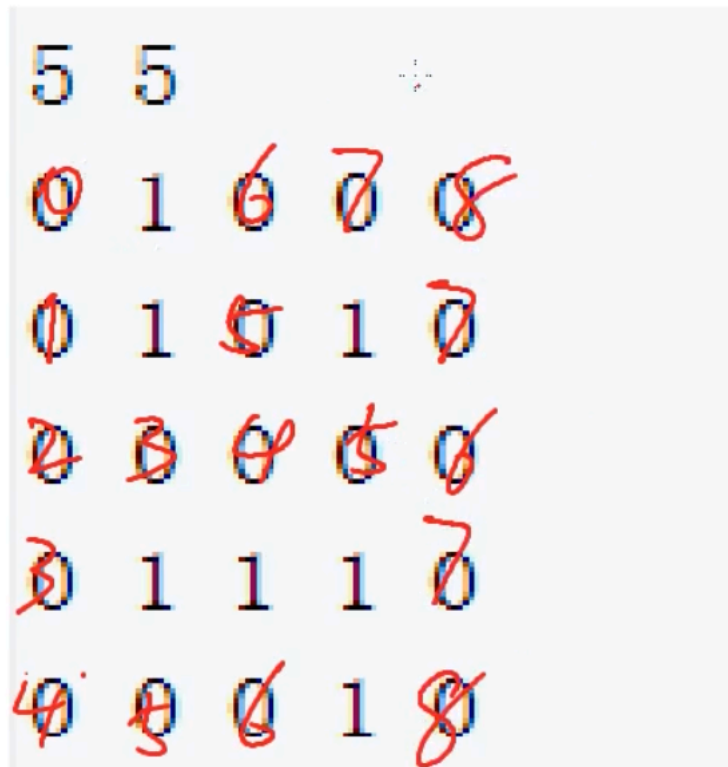
[AcWing844.走迷宫](#)

算法流程：

从左上角  $(0, 0)$  为起点，按照一定的顺序，走到每一个距离为1的所有点；

然后，走到距离为1的点之后接着走到距离起点为2的所有点，直到走到终点 $(n, n)$ 。

样例模拟：



说明：数字即表示 搜索的层数/先后顺序，同时也表示距离。

算法框架

```
1 队头元素 -> queue          // 队头元素放到queue中
2 while (queue不为空)
3 {
4     取当前队头
5     扩展队头
6     // 后续操作
7 }
```

代码示例（模拟队列写法）：

```
1 #include <iostream>
2 #include <cstdio>
3 #include <cstring>
4 #include <algorithm>
5 using namespace std;
6
7 typedef pair<int, int> PII;
8 const int N = 110;
```

```

9
10 int n, m;
11 int g[N][N];    // 存放图
12 int d[N][N];    // 每一个点d[x][y]到起点的距离
13 PII q[N * N];   // 模拟队列q[], 队列的大小是N*N, q[i]有两个元素q[i].first是横坐标
                  // x, q[i].second是纵坐标y
14
15
16 int bfs()
17 {
18     int hh = 0, tt = 0;    // 初始化: 队头hh是0, 队尾tt是0
19     q[0] = {0, 0};        // 初始化起始位置
20
21     memset(d, -1, sizeof(d));    // 初始化距离, 保证每一个位置都是第一次走
22     d[0][0] = 0;    // 从起点开始
23     // 上右下左, 四个方向的xy坐标变换的偏移量
24     int dx[4] = {-1, 0, 1, 0};
25     int dy[4] = {0, 1, 0, -1};
26
27     while (hh <= tt)    // 队列不为空
28     {
29         PII t = q[hh++];    // 取出队头元素
30         // 枚举4个方向
31         for (int i = 0; i < 4; ++i)
32         {
33             int x = t.first + dx[i], y = t.second + dy[i];
34             // 保证xy在正确的范围内, 而且g[x][y] == 0表示当前路可以走; d[x][y] == -1
表示当前路的是第一次走, 保证了最短路的求得
35             if (x >= 0 && x < n && y >= 0 && y < m && g[x][y] == 0 && d[x][y]
== -1)
36             {
37                 d[x][y] = d[t.first][t.second] + 1;    // 更新当前点到起始点的距
离
38                 q[++tt] = {x, y};    // 将当前点入队
39             }
40         }
41     }
42     return d[n - 1][m - 1];    // 返回 (终点[n,m]到起点的距离) 最短距离
43 }
44
45 int main()
46 {
47     scanf("%d%d", &n, &m);
48     for (int i = 0; i < n; ++i)
49         for (int j = 0; j < m; ++j)
50             scanf("%d", &g[i][j]);
51     printf("%d\n", bfs());
52     return 0;
53 }

```

如果想要输出路径, 只需要开一个额外的pair数组, 记录当前坐标(x,y)是从哪一个点走过来的即可。

代码示例 (STL写法) :

```

1 #include <iostream>
2 #include <cstdio>
3 #include <cstring>
4 #include <queue>
5 #include <algorithm>
6

```

```

7 using namespace std;
8 typedef pair<int,int> PII;
9
10 const int N = 110;
11 int n, m;
12 int g[N][N];
13 int d[N][N];
14
15 int bfs()
16 {
17     queue<PII> q;
18     memset(d, -1, sizeof(d));
19     d[0][0] = 0;
20     q.push({0, 0});
21
22     int dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};
23     while (q.size())
24     {
25         auto t = q.front();
26         q.pop();
27
28         for (int i = 0; i < 4; ++i)
29         {
30             int x = t.first + dx[i], y = t.second + dy[i];
31             if (x >= 0 && x < n && y >= 0 && y < m && d[x][y] == -1 && g[x]
[y] == 0)
32             {
33                 d[x][y] = d[t.first][t.second] + 1;
34                 q.push({x, y});
35             }
36         }
37     }
38     return d[n - 1][m - 1];
39 }
40
41 int main()
42 {
43     scanf("%d%d", &n, &m);
44     for (int i = 0; i < n; ++i)
45         for (int j = 0; j < m; ++j)
46             scanf("%d", &g[i][j]);
47     int res = bfs();
48     printf("%d\n", res);
49 }

```

### 3.2.2 八数码问题

## 3.3 树和图的存储

树形结构是一种特殊的图结构。所以我们只需要理解图结构的存储即可。

图有两种类型有向图和无向图。算法中，无向图在相连结点之间建立两条边；有向图根据指向建立一条边。无向图我们也可以看成一种特殊的有向图。

有向图的存储两种方式：

邻接矩阵：二维矩阵；不能存储重边，多条重边只能够存储一条。

空间复杂度： $O(n^2)$

邻接表：多个单链表

代码实现，见后续题目中的代码

通用存储树和图的代码模版

```
1  #include <iostream>
2  #include <cstdio>
3  #include <algorithm>
4
5  using namespace std;
6
7  const int N = 1e6 + 10, M = N * 2;
8
9  int h[N], e[M], ne[M], idx;
10
11 // 加边：添加一条从a指向b的边
12 void add(int a, int b)
13 {
14     e[idx] = b;
15     ne[idx] = h[a];
16     h[a] = idx++;
17 }
18
19 int mian()
20 {
21     // 初始化邻接表的表头
22     memset(h, -1, sizeof(h));
23
24     return 0;
25 }
```

注意：无向图，建立两条方向相反的有向边。

## 3.4 树和图的深度优先遍历

### 3.4.1 树的重心

题目链接

[AcWing846.树的重心](#)

题解

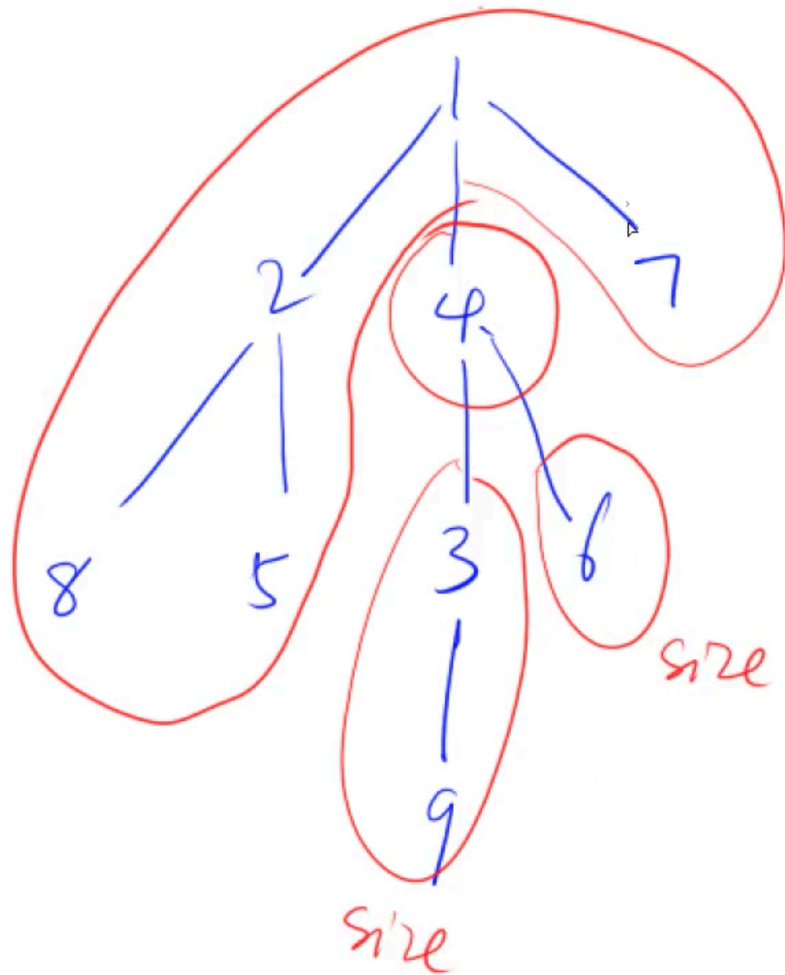
树的遍历，相当于特殊有向图的遍历。

**树的重心：**

树中的一个结点，如果将这个点删除后，剩余各个连通块中点数的最大值最小，那么这个节点被称为树的重心。（换句话说，把一个点删除掉之后，以这个点（被删除的点）为根的各棵子树的结点数量中的最大值，再从所有的最大值中取最小值；把连通块理解成 子树/集合）

**思路：**

遍历整棵树，遍历结点的同时将每一个结点删除掉，删除节点的同时求解连通块的数量最大值，从这些最大值中再找最小值。（求删除每一个点后的连通块的数量最大值）



连通块的求解：

从 4 开始DFS，通过 4 的儿子结点的 dfs 返回值 就能够得到两个 size 的大小；剩下一个连通块的数量为 结点总数 - size[4]，即减去以 4 根的树的点个数。

时间复杂度

$O(n + m)$

时间复杂度和 点数 以及 边数 呈线性关系

代码示例-模拟队列写法

```

1  #include <iostream>
2  #include <cstdio>
3  #include <cstring>
4  #include <algorithm>
5
6  using namespace std;
7
8  const int N = 1e6 + 10, M = 2 * N;
9
10 int n; // 树的结点个数
11 int h[N], e[M], ne[M], idx;
12 bool st[N]; // 每个点只能遍历一次, st用来作标记
13
14 int ans = N; // 记录答案 (最大值)
15
16 // 加边
17 void add(int a, int b)

```

```

18 {
19     e[idx] = b;
20     ne[idx] = h[a];
21     h[a] = idx ++;
22 }
23
24 int dfs(int u)    // 当前遍历的图中结点u
25 {
26     st[u] = true; // 更新u的状态
27
28     int sum = 1; // 以u为根的树中结点个数（当前点算一个，默认为1）
29     int res = 0; // 所有连通块中结点数量的最大值
30     // 遍历结点u的出边
31     for (int i = h[u]; i != -1; i = ne[i])
32     {
33         int j = e[i]; // j表示链表中出边指向的结点，在图中的编号
34         if (!st[j])
35         {
36             int s = dfs(j); // 以u为根的子树的大小
37             res = max(res, s); // u 和子树 的连通块的最大值
38             sum += s; // u的子树的结点，是u这个树中的一部分，所以结点数量要更新
39         }
40     }
41     res = max(res, n - sum); // (n - sum) 表示 删除u为根的树，不算u这个树后剩下的
    连通块点的个数
42
43     ans = min(ans, res); // 连通块的（结点数量）最大值中最小值
44
45     return sum; // 返回 以u为根的树的结点个数
46 }
47
48 int main()
49 {
50     scanf("%d", &n);
51
52     memset(h, -1, sizeof(h)); // （容易遗忘）初始化存图的链表表头
53
54     for (int i = 0; i < n - 1; ++i)
55     {
56         // 无向图:加两条边（反向）
57         int a, b;
58         scanf("%d%d", &a, &b);
59         add(a, b), add(b, a);
60     }
61     dfs(1);
62     printf("%d\n", ans);
63     return 0;
64 }

```

## 代码示例-STL写法

```

1 // 代码注释看"模拟队列写法"
2 #include <iostream>
3 #include <cstdio>
4 #include <cstring>
5 #include <queue>
6
7 using namespace std;
8
9 const int N = 1e6 + 10, M = 2 * N;

```

```

10 int n, m;
11 int h[N], e[M], ne[M], idx;
12 int d[N];
13
14 void add(int a, int b)
15 {
16     e[idx] = b;
17     ne[idx] = h[a];
18     h[a] = idx++;
19 }
20
21 int bfs()
22 {
23     memset(d, -1, sizeof(d));
24     queue<int> q;
25     q.push(1);
26     d[1] = 0;
27     while (q.size())
28     {
29         int t = q.front();
30         q.pop();
31         for (int i = h[t]; i != -1; i = ne[i])
32         {
33             int j = e[i];
34             if (d[j] == -1)
35             {
36                 d[j] = d[t] + 1;
37                 q.push(j);
38             }
39         }
40     }
41     return d[n];
42 }
43
44 int main()
45 {
46     scanf("%d%d", &n, &m);
47     memset(h, -1, sizeof(h));
48     for (int i = 0; i < m; ++i)
49     {
50         int a, b;
51         scanf("%d%d", &a, &b);
52         add(a, b);
53     }
54     printf("%d\n", bfs());
55     return 0;
56 }

```

### 通用dfs代码模版

```

1 #include <iostream>
2 #include <cstdio>
3 #include <cstring>
4 #include <algorithm>
5
6 using namespace std;
7
8 const int N = 1e6 + 10, M = N * 2;

```

```

9
10 int h[N], e[M], ne[M], idx;
11 bool st[N]; // 第i个点时候 被访问过/遍历过
12
13 // 加边操作:给a加一条到b的边
14 void add(int a, int b)
15 {
16     e[idx] = b;
17     ne[idx] = h[a];
18     h[a] = idx++;
19 }
20
21 void dfs(int u) // 当前被访问的结点
22 {
23     st[u] = true; // 更新状态, u被访问
24     // 遍历当前点的所有出边
25     for (int i = h[u]; i != -1; i = ne[i])
26     {
27         int j = e[i]; // j表示链表中的结点在图中的编号
28         if (!st[j]) // 如果没有遍历/搜索过, 继续向下搜索 (“一条道走到黑”)
29             dfs(j);
30     }
31 }
32 int main()
33 {
34     // 初始化 临界表
35     memset(h, -1, sizeof(h)); // 初始化链表表头
36
37     return 0;
38 }

```

注意：不论是DFS还是BFS，每一个点只需要遍历一次即可。

## 3.5 树和图的宽度优先遍历BFS

### 3.5.1 图中点的层次

题目链接

[AcWing847.图中点的层次](#)

重边：a和b之间有超过一条边

自环：存在a指向a自己的边

题解

由于图中边权为1，可以使用BFS来求最短路/最短距离。

代码示例

```

1 #include <iostream>
2 #include <cstdio>
3 #include <cstring>
4 #include <algorithm>
5
6 using namespace std;
7
8 const int N = 1e6 + 10, M = 2 * N;
9
10 int n, m; // n个点 m条边
11 int h[N], e[M], ne[M], idx;

```



```

12 int d[N]; // 表示点i到起点的距离
13 int q[N]; // 模拟队列
14
15 void add(int a, int b)
16 {
17     e[idx] = b;
18     ne[idx] = h[a];
19     h[a] = idx ++;
20 }
21
22 int bfs()
23 {
24     int hh = 0, tt = 0; // hh 队头指针, tt 队尾指针
25     q[0] = 1; // 起点入队
26
27     memset(d, -1, sizeof(d)); // 初始化距离, 所有点距离起点的距离都为-1 (同时可以
    作为是否被遍历过的标志)
28
29     d[1] = 0; // 起点到起点的距离
30     // BFS框架
31     while (hh <= tt)
32     {
33         int t = q[hh++]; // 取队头, 并且队头出队
34         for (int i = h[t]; i != -1; i = ne[i])
35         {
36             int j = e[i];
37             if (d[j] == -1) // j没有被遍历过, 那么距离起点的距离为-1
38             {
39                 // 更新距离, 扩展队头的点x
40                 d[j] = d[t] + 1;
41                 q[++tt] = j;
42             }
43         }
44     }
45     return d[n]; // 返回最短路
46 }
47 int main()
48 {
49     scanf("%d%d", &n, &m);
50     memset(h, -1, sizeof(h)); // 初始化链表表头
51     for (int i = 0; i < m; ++i)
52     {
53         int a, b;
54         scanf("%d%d", &a, &b);
55         add(a, b);
56     }
57     printf("%d\n", bfs());
58     return 0;
59 }

```

## 通用BFS框架

```

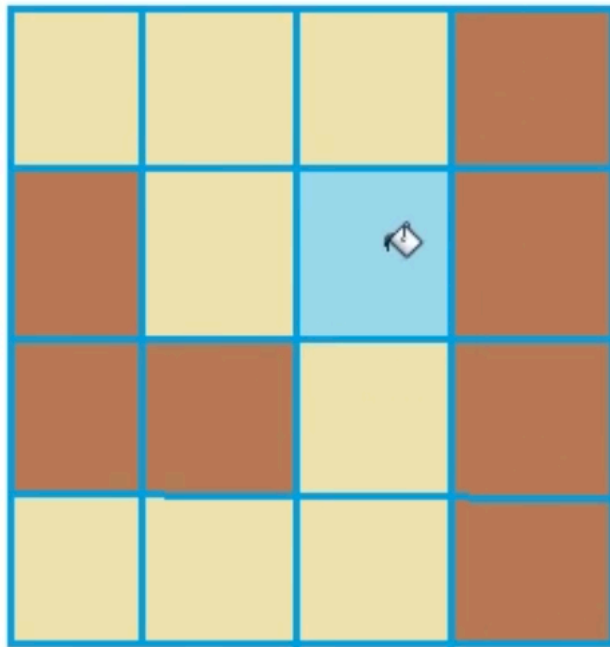
1  #include <queue>
2  // queue为队列
3  1. 将起点放到queue的队头中
4  2.
5  while (queue 不为空)
6  {
7      取出队头元素t,并及时弹出queue中的队头元素
8
9      扩展t能够到达的所有邻点x
10     if (x 没有被遍历过)
11         将x放入queue (x入队)
12     // 接下来的操作 (本题目中 更新距离)
13 }

```

## 3.6 BFS算法拓展

### 3.6.1 Flood Fill--洪水灌溉算法

算法简介：



在一个迷宫/地图中，深色区域为墙，浅色区域为洼地；我们从洼地中找到一个位置开始注水（图中蓝色区域），当洼地中的水被注满时，会向相邻的区域灌溉/填充；当填满所有洼地或者满足某个条件时停止注水。

DFS类似，只不过变为当前位置注满水后，向固定方向的下一个位置继续注水，直到这个方向无法继续注水时，再回溯换一个方向注水，重复上述操作。

算法的作用：

可以在线性时间复杂度内，找到某个点所在的连通块。

注意：虽然Flood-Fill算法同样可以用DFS来实现。为了防止出现DFS中爆栈的问题，尽量使用BFS来实现Flood-Fill算法。

题目链接

[AcWing1097.池塘计数](#)

四连通：某个点和其相邻的上下左右四个点连通

八连通：某个点和其上下左右、左右上、左右下共八个点连通

### 算法思路

使用Flood-Fill算法从左上角开始遍历所有点，只要当前位置是水洼而且第一次被遍历到并且当前位置周围连通起来的水洼共同构成一个池塘，那么就满足条件更新结果。

代码示例：

```
1  #include <iostream>
2  #include <cstdio>
3  #include <cstring>
4  #include <algorithm>
5
6  // 将 pair中的第一个元素和第二个元素的表示简化 first 、 second
7  #define x first
8  #define y second
9
10 using namespace std;
11
12 typedef pair<int, int> pii;    // 使用pair存放“二维下标”
13
14 const int N = 1010, M = N * N;    // N的长宽理论最大值，M图的理论最大值
15
16 int n, m;    // 地图的长宽
17 char g[N][M];
18 pii q[M];    // 使用pair模拟队列写法
19 bool st[N][N]; // st[i][j] 表示 点(i,j)是否被遍历过(判重：防止重复遍历某一个点)
20
21 int bfs(int sx, int sy)
22 {
23
24 }
25
26 int main()
27 {
28     scanf("%d%d", &n, &m);
29
30     // 按行读数据
31     for (int i = 1; i <= n; ++i) scanf("%s", g[i]);
32
33     // Flood-Fill
34     int cnt = 0;
35     for (int i = 1; i <= n; ++i)
36         for (int j = 1; j <= m; ++j)
37             if (g[i][j] == 'W' && !st[i][j]) // 是"水洼"而且"水洼第一次被遍历过"
38             {
39                 // 只要找到了“第一次被遍历过的水洼”，就说明了池塘的存在；先用BFS搜索池塘的大小，然后
40                 bfs(i, j); // 从水洼地开始使用BFS搜索“池塘”
41                 cnt++;
42             }
43
44     // 输出结果
45     printf("%d\n", cnt);
46     return 0;
47 }
```

### 3.6.2 BFS的最短路模型

## 3.7 DFS算法拓展

### 3.X.1 连通性模型

### 3.x.2 图和树的遍历

## 3.x 拓扑排序

## 3.x 最短路算法

## 第五部分 动态规划

动态规划问题，可以从两个方面去考虑：状态表示和状态计算；其中状态表示可以分为集合+属性，状态计算可以分为集合的划分+状态方程；最后通过优化的方法减少时间复杂度或者空间复杂度。

其中，集合为，能够满足"条件"的所有物品/方法；

属性，最大值、最小值或者数量；

集合的划分，把集合划分成若干个更小的子集，然后把每一个子集划分成更小的子集，通过计算出更下的子集的结果，来得到上一层集合的结果，最终求得整个集合；

状态方程，状态转移方程；

优化,代码/状态方程的等价变形。

## 5.1 背包问题

### 5.1.1 问题背景

有  $N$  个物品和一个容量为  $V$  的背包；每一件物品  $i$  都有两个属性  $v_i$  和  $w_i$ ,其中  $w_i$  表示第  $i$  件物品的体积， $v_i$  表示第  $i$  件物品的价值。在不超过背包容量  $V$  的情况下，求解放进背包的物品的最大价值。

### 5.1.2 01背包问题

#### 5.1.2.1 问题描述及题目链接

每件物品只有一件，即每一件物品只能够"使用"一次。

题目链接：

[AcWing2.01背包问题](#)

#### 5.1.2.2 朴素算法

状态表示： $f[i, j]$

集合：从  $i$  件物品中挑选出容量不超过  $j$  的所有物品的价值

属性：最大值

状态计算：

集合的划分：

1) 不选第  $i$  件物品时，物品的总体积不超过  $j$  的选法，表示为  $f[i-1, j]$ ；

$f[i-1, j]$  中的  $i-1$  表示为从前  $i-1$  个物品中选（没有选择第  $i$  件物品）；整体，表示为不选第  $i$  件物品时，容量不超过  $j$  的选法中，最大值；

2) 选择第  $i$  件物品时，物品的总体积不超过  $j$  的选法，表示为  $f[i-1, j-v_i] + w_i$ ；

$f[i-1, j-v_i] + w_i$  表示从前  $i-1$  个物品中选，体积不超过  $j-v_i$ （虽然选择了第  $i$  件物品，但是现考虑前  $i-1$  件物品的选法，然后补上第  $i$  件物品的体积  $v_i$  和价值  $w_i$ ）的选法；表示为  $f[i-1, j-v_i] + w_i$

状态方程：

1) 不选择第  $i$  件物品时，物品的总体积不超过  $j$  的选法：

$$f[i, j] = f[i-1, j]$$

2) 选择第  $i$  件物品时，物品的总体积不超过  $j$  的选法：

$$f[i, j] = f[i-1, j-v_i] + w_i$$

状态转移方程：

$$f[i, j] = \max(f[i-1, j], f[i-1, j-v_i] + w_i)$$

代码示例：

```
1  #include <iostream>
2  #include <cstdio>
3  #include <algorithm>
4
5  using namespace std;
6  const int N = 1010;
7  int n, m;    // n物品的数量， m背包的容量
8  int f[N][N]; // 状态方程，f[i][j]从前i个物品中选，所有选法的最大值
9  int v[N];    // v[i]表示第i件物品的体积
10 int w[N];    // w[i]表示第i件物品的价值
11
12 int main()
13 {
14     scanf("%d%d", &n, &m);
15     for (int i = 1; i <= n; ++i) scanf("%d%d", &v[i], &w[i]);
16
17     // f[0][0~m]表示从前0件物品中选，体积不超过0~m的选法中的最大值，结果是0，默认是0，
    不需要初始化
18     for(int i = 1; i <= n; ++i)
19         for (int j = 0; j <= m; ++j)
20             {
21                 f[i][j] = f[i-1][j];
22                 if (j >= v[i]) f[i][j] = max(f[i][j], f[i-1][j-v[i]] +
23 w[i]);
24             }
25     return 0;
26 }
```

### 5.1.2.3 优化（一维算法）

将状态从二维表示变成一维表示

状态表示:  $f[j]$

集合: 体积不超过  $j$  的情况下的

属性: 最大值

状态计算

状态划分: 滚动数组思想

从朴素算法中的状态转移方程  $f[i, j] = f[i - 1, j] + f[i - 1, j - v_i] + w_i$  我们不难发现, 参数  $i$  仅仅使用了两个, 也就是  $i$  和  $i - 1$ ; 参数  $j$  的范围不论是  $j$  还是  $j - v_i$  都不超过  $j$ ; 所以, 我们可以用一维来表示。

状态方程:

$$f[j] = \max(f[j - v_i] + w_i)$$

注意:

要逆序枚举  $j$ , 防止在更新  $f[j]$  时, 受到  $f[j - v_i]$  的影响

代码示例:

```
1  #include <iostream>
2  #include <cstdio>
3  #include <algorithm>
4
5  using namespace std;
6
7  const int N = 1010;
8  int n, m;
9  int f[N];    // 状态表示 f[j] 容量不超过 j 的情况下选法的最大值
10 int v[N], w[N];
11
12 int main()
13 {
14     scanf("%d%d", &n, &m);
15     for (int i = 1; i <= n; ++i) scanf("%d%d", &v[i], &w[i]);
16
17     for (int i = 1; i <= n; ++i)
18         for (int j = m; j >= v[i]; --j)
19             f[j] = max(f[j], f[j - v[i]] + w[i]);
20     printf("%d\n", f[m]);
21     return 0;
22 }
```

## 5.1.3 完全背包问题

### 5.1.3.1 问题描述及题目链接

每一件物品  $i$  可以选择任意多次。

题目链接:

[AcWing3.完全背包问题](#)

### 5.1.3.2 朴素算法

状态表示:  $f[i, j]$

集合：从  $i$  件物品，选择容量不超过  $j$  的物品的选择

属性：最大值

状态计算：

集合划分：

在对前  $i - 1$  件物品作出选择后，从第  $i$  件物品选择  $k$  件的选法；当前这种选法得到的最大价值；

状态方程：

在对前  $i - 1$  件物品作出选择后，选择  $k$  件第  $i$  件物品的选法的状态方程

1. 当第  $i$  件物品选择了 0 件时，

$$f[i, j] = f[i - 1, j - 0 * v_i] + 0 * w_i$$

2. 当第  $i$  件物品选择了 1 件时，

$$f[i, j] = f[i - 1, j - 1 * v_i] + w_i$$

...

3. 当第  $i$  件物品选择了  $k$  件时，

$$f[i, j] = f[i - 1, j - k * v_i] + k * w_i$$

代码示例：

```
1  #include <iostream>
2  #include <cstdio>
3  #include <algorithm>
4
5  using namespace std;
6
7  const int N = 1010;
8  int f[N][N];    // 状态方程
9  int v[N], w[N]; // v[i]表示第i件物品的体积，w[i]表示第i件物品的价格
10 int n, m;       // n件物品， m表示背包的容量
11
12 int main()
13 {
14     scanf("%d%d", &n, &m);
15     for (int i = 1; i <= n; ++i) scanf("%d%d", &v[i], &w[i]);
16
17     // 状态转移方程：
18     //      f[i][j] = f[i - 1][j - k * v[i]] + k * w[i]
19     for (int i = 1; i <= n; ++i)
20         for (int j = 0; j <= m; ++j)
21             for (int k = 0; k * v[i] <= j; ++k)
22                 f[i][j] = max(f[i][j], f[i - 1][j - k * v[i]] + k * w[i]);
23
24     printf("%d\n", f[n][m]);
25     return 0;
26 }
```

### 5.1.3.3 优化（二维）

状态表示：

同朴素算法

状态计算：

同朴素算法

优化：

首先，观察朴素算法的状态转移方程： $f[i][j] = f[i-1, j - k * v_i] + k * w_i$

然后，展开  $f[i, j]$  和  $f[i-1, j - v_i]$ ：

$f[i, j] = \max(f[i-1, j] + f[i-1, j - v_i] + w_i + f[i-1, j - 2 * v_i] + 2 * w_i + \dots + f[i-1, j - k * v_i] + k * w_i)$   
，其中  $\max()$  里面的  $f[i-1, j]$  表示从前  $i$  件物品中选择，容量不超过  $j$  的情况下，当前选法没有选择第  $i$  件物品的价值； $f[i-1, j - v_i] + w_i$  表示从前  $i$  件物品中选择，容量不超过  $j$  的情况下，当前选法选择1件第  $i$  件物品的价值...

$f[i, j - v_i] = \max(f[i-1, j - v_i] + f[i-1, j - 2 * v_i] + w_i + f[i-1, j - 2 * v_i] + 2 * w_i + \dots + f[i-1, j - k * v_i] + k * w_i)$   
，其中  $\max()$  里面的  $f[i-1, j - v_i]$  表示从前  $i$  件物品中选择，容量不超过  $j - v_i$  的情况下，当前选法没有选择第  $i$  件物品的价值...

最后，将展开后的  $f[i, j]$  和  $f[i, j - v_i]$  对比和代入/替换得到：

$$f[i, j] = \max(f[i-1, j] + f[i, j - v_i] + w_i)$$

代码示例：

```
1  #include <iostream>
2  #include <cstdio>
3  #include <algorithm>
4
5  using namespace std;
6
7  const int N = 1010;
8  int f[N][N];    // 状态方程
9  int v[N], w[N]; // v[i]表示第i件物品的体积，w[i]表示第i件物品的价格
10 int n, m;       // n件物品， m表示背包的容量
11
12 int main()
13 {
14     scanf("%d%d", &n, &m);
15     for (int i = 1; i <= n; ++i) scanf("%d%d", &v[i], &w[i]);
16
17     // 状态转移方程
18     // f[i][j] = max(f[i-1][j], f[i][j-v[i]] + w[i])
19     for (int i = 1; i <= n; ++i)
20         for (int j = 0; j <= m; ++j)
21             {
22                 f[i][j] = f[i-1][j];
23                 if (j >= v[i]) f[i][j] = max(f[i][j], f[i][j - v[i]] + w[i]);
24             }
25     printf("%d\n", f[n][m]);
26     return 0;
27 }
```

#### 5.1.3.4 优化（一维）

优化：

不难发现，优化（二维）方法的状态转移方程以及代码示例和01背包问题很像，我们同样可以使用代码等价变形的方法将代码优化为一维表示。



代码示例：

```
1 #include <iostream>
2 #include <cstdio>
3 #include <algorithm>
4
5 using namespace std;
6
7 const int N = 1010;
8 int n, m;
9 int f[N];
10 int v[N], w[N];
11
12 int main()
13 {
14     scanf("%d%d", &n, &m);
15     for (int i = 1; i <= n; ++i) scanf("%d%d", &v[i], &w[i]);
16
17     for (int i = 1; i <= n; ++i)
18         for (int j = v[i]; j <= m; ++j)
19             f[j] = max(f[j], f[j - v[i]] + w[i]);
20     printf("%d\n", f[m]);
21     return 0;
22 }
```

思考：为什么优化（一维）后的内层循环是顺序枚举  $j$  而不是像“01背包问题”中的优化代码那样逆序枚举  $j$  呢？

观察优化（二维）的状态转移方程  $f[i, j] = \max(f[i - 1, j], f[i, j - v_i + w_i])$ ，优化（一维）中的  $f[j]$  在  $f[j - v_i]$  的更新之后才得到更新，所以不会受到影响。

## 5.2 线性DP

### 5.2.1 数字三角形

#### 5.2.1.1 题目链接：

[AcWing898.数字三角形](#)

#### 5.2.1.2 算法流程：

状态表示： $f[i, j]$ ,  $a[i, j]$  表示数字三角形中，第  $i$  行第  $j$  列的位置的数字

集合：所有从起点走到  $[i, j]$  的路径；（第  $i$  行第  $j$  列的数字;起点是  $(1, 1)$ ）

属性：最大值

状态计算：

集合划分：

1. 来自左上方（从左上方的位置走到  $(i, j)$ ，走到  $(i - 1, j - 1)$  的位置，已经得到最大路径长度和当前位置的数字相加：

$$f[i, j] = f[i - 1, j - 1] + a[i, j]$$

2. 来自右上方（从右上方的位置走到  $(i, j)$ ）：

$$f[i, j] = f[i - 1, j] + a[i, j]$$

为什么上一个点的位置是  $(i - 1, j)$  而不是  $(i + 1, j + 1)$  或者是其他的坐标呢？

因为：我们用数组来存放这个数字三角形，由于存放的方式使得，每一个数字的坐标又一些改变。

状态方程：

$$f[i, j] = \max(f[i - 1, j - 1] + a[i, j], f[i - 1, j] + a[i, j])$$

注意：

数字三角形的存放和坐标位置表示；

通过状态方程我们求解得到的是从起点走到最底下一层的每一个位置的最大路径长度,然后通过比较最底下一层的路径长度来求得最中结果。

代码示例：

```
1 #include <iostream>
2 #include <cstdio>
3 #include <algorithm>
4
5 using namespace std;
6
7 const int N = 510, INF = 1e9;
8
9 int n;
10 int a[N][N];
11 int f[N][N];
12
13 int main()
14 {
15     scanf("%d", &n);
16     for (int i = 1; i <= n; ++i)
17         for (int j = 1; j <= i; ++j)
18             scanf("%d", &a[i][j]);
19     // 初始化状态, 将每一个状态初始化成“负无穷”, 而且额外初始化第0行第0列, 每一列还要额外
    // 初始化一个坐标的状态
20     //
21     for (int i = 0; i <= n; ++i)
22         for (int j = 0; j <= i + 1; ++j)
23             f[i][j] = -INF;
24     // DP
25     f[1][1] = a[1][1]; // 初始化起点
26     for (int i = 2; i <= n; ++i)
27         for (int j = 1; j <= i; ++j)
28             f[i][j] = max(f[i - 1][j - 1] + a[i][j], f[i - 1][j] + a[i][j]);
29     // 遍历最后一层数字三角形的状态, 找最大值
30     int res = -INF;
31     for (int i = 0; i <= n; ++i) res = max(res, f[n][i]);
32     printf("%d\n", res);
33     return 0;
34 }
```

练习与提高：

[AcWing1015.摘花生](#)：

状态方程：

$$f[i, j] = \max(f[i - 1, j] + w[i][j], f[i][j - 1] + w[i][j])$$

[AcWing1018.最低通行费](#)：

注意：

仔细分析  $2N-1$  这个条件；

考虑边界问题，对特殊位置进行特判；

状态方程：

$$f[i, j] = \min(f[i-1, j] + w[i][j], f[i][j-1] + w[i][j])$$

[AcWing1027.方格取数:](#)

思路：

1) 两个人同时走

2) 一个先走，将走过的路上打上标记

状态表示：

两个人同时走

集合： $f[i_1, j_1, i_2, j_2]$  和  $f[k, i_1, i_2]$

$[i_1, j_1, i_1, j_1]$  表示两条路径从  $(1, 1)$  和  $(1, 1)$  分别走到  $(i_1, j_1)$  和  $(i_2, j_2)$  的路径的集合，

$f[i_1, j_1, i_2, j_2]$  表示两条路径走到  $(i_1, j_1)$  和  $(i_2, j_2)$  的长度之和最大值；

化简状态用3个状态来表示：

$[k, i_1, i_2]$  用  $k$  表示两条路径当前走到的格子位置中横纵坐标之和， $f[k, i_1, i_2]$  中表示从  $(1, 1)$

和  $(1, 1)$  分别走到  $(i_1, k - i_1)$  和  $(i_2, k - i_2)$  的长度之和最大值。

其中  $k$  满足的等价关系：

$$k = (i_1 + j_1) = (i_2 + j_2)$$

属性：最大值

状态计算：

集合划分：

从最后一步开始考虑的思想，当重合时，做到不重复处理；

1) 两条路径分别从上面/向下走走到  $(i_1, j_1)$  和从上面/向下走走到  $(i_2, j_2)$

2) 两条路径分别从上面/向下走走到  $(i_1, j_1)$  和从左边/向右走走到  $(i_2, j_2)$

3) 两条路径分别从左边/向右走走到  $(i_1, j_1)$  和从上面/向右走走到  $(i_2, j_2)$

4) 两条路径分别从左边/向右走走到  $(i_1, j_1)$  和从左边/向右走走到  $(i_2, j_2)$

状态方程：

下面以 (1) 为例：

$$f[k, i_1, i_2] = \begin{cases} f[k-1, i_1-1, j_1] + f[k-1, i_2-1, j_1] + w[i_1][j_1] & \text{两个点重合} \\ f[k-1, i_1-1, j_1] + w[i_1][j_1] + f[k-1, i_2-1, j_2] + w[i_2][j_2] & \text{两个点不重合} \end{cases}$$

思考：

如何处理 同一个格子不被重复选择/走过？

答：由于两条路径是同时移动，而且每次移动一个格子；我们不难发现一个关系式：

$$i_1 + j_1 = i_2 + j_2$$

即两条路线在横纵坐标之和相同时，才会出现重合。

为什么  $i_1 + j_1 = i_2 + j_2$  能够说明两个路径走过的格子相同？

答：每次移动两条路径走过的点的方向（向下/向右）只有两个，那么我们能够确定两个点的下一步的移动范围，由  $k = i_1 + j_1 = i_2 + j_2$  只要每一步出现  $i_1 = i_2$  那么  $j_1 = j_2$ ，产生了重合。

代码示例：

```
1 #include <iostream>
2 #include <cstdio>
3 #include <algorithm>
4
5 using namespace std;
6
7 const int N = 15;
8 int n;
9 int f[2 * N][N][N];
10 int w[N][N];
11
12 int main()
13 {
14     scanf("%d", &n);
15     // 输入 (有技巧性)
16     int a, b, c;
17     while (cin >> a >> b >> c, a || b || c) w[a][b] = c;
18
19     // DP f[k][i1][i2]
20     for (int k = 2; k <= n + n; ++k)
21         for (int i1 = 1; i1 <= n; ++i1)
22             for (int i2 = 1; i2 <= n; ++i2)
23             {
24                 int j1 = k - i1, j2 = k - i2;
25                 // 判断是否在合法范围内
26                 if (j1 >= 1 && j1 <= n && j2 >= 1 && j2 <= n)
27                 {
28                     // 特判, 重复
29                     int t = w[i1][j1];
30                     if (i1 != i2) t += w[i2][j2];
31                     // 向下 向下
32                     f[k][i1][i2] = max(f[k][i1][i2], f[k-1][i1-1][i2-1] + t);
33                     // 向下 向右
34                     f[k][i1][i2] = max(f[k][i1][i2], f[k-1][i1-1][i2] + t);
35                     // 向右 向下
36                     f[k][i1][i2] = max(f[k][i1][i2], f[k-1][i1][i2-1] + t);
37                     // 向右 向右
38                     f[k][i1][i2] = max(f[k][i1][i2], f[k-1][i1][i2] + t);
39                 }
40             }
41     printf("%d\n", f[2*n][n][n]);
42     return 0;
43 }
```

## 5.2.2 最长上升子序列 LIS (Longest Increasing Subsequence)

题目链接及解释：

[AcWing895.最长上升子序列](#)

何为子序列？

```
1 | abcd
```

abcd 是我们已知的序列，序列 abcd 的子集组成的集合就是序列 abcd 的子序列，比如：

长度为 1 的子序列 a，长度为 2 的子序列 ab，长度为 3 的子序列 acd。

状态表示：f[i]

集合：从第一个数字开始计算，所有以数字  $a_i$  结尾的上升子序列 属性：最大值

状态计算：

集合划分：

划分依据：按照最后一步来划分，结尾数字  $a_i$  前面的一个数字是不确定的，我们划分出  $i - 1$

种可能性（表示以数字  $a_i$  结尾的序列的前面的数字个数）

划分依据的另一种描述：当前上升子序列中最后一个不同的点/结尾数字的前一个点的值

1) 以  $a_i$  结尾的上升子序列中，前一个数字为  $a_1$

2) 以  $a_i$  结尾的上升子序列中，前一个数字为  $a_2$

...

3) 以  $a_i$  结尾的上升子序列中，前一个数字为  $a_{i-1}$

除了上面  $i - 1$  种情况，还有一个上升子序列中有且仅有  $a_i$  这一个数字，这个情况下上升子序列的长度为 1

状态方程：

已知，数列

...  $a_k$   $a_j$   $a_i$  ...

而且， $f[j]$  表示序列中以数字  $j$  为结尾的最长上升子序列的长度；

当  $a_j < a_i$  时，满足最长上升子序列的条件更新  $f[i] = f[j] + 1$ 。（因为在更新  $f[i]$  之前  $f[j]$  就已经被更新/计算出来了）

状态转移方程：

$$f[i] = f[j] + 1 \quad (j < i)$$

时间复杂度：

$$O(n^2)$$

代码示例：

```
1 | #include <iostream>
2 | #include <cstdio>
3 | #incldue <algorithm>
4 |
5 | using namespace std;
6 |
7 | const int N = 1010;
8 | int f[N];
9 | int a[N];
10 |
11 | int main()
12 | {
13 |     scanf("%d", &n);
14 |     for (int i = 1; i <= n; ++i) scanf("%d", &a[i]);
```

```

15
16     // DP
17     for (int i = 1; i <= n; ++i)
18     {
19         f[i] = 1;    // 当只有a[i]自己, 组成上升子序列时
20         for (int j = 1; j < i; ++j)
21         {
22             if (a[j] < a[i])
23                 f[i] = max(f[i], f[j] + 1);
24         }
25     }
26     // 找最大值
27     int res = 0;
28     for (int i = 1; i <= n; ++i) res = max(res, f[i]);
29     printf("%d\n", res);
30     return 0;
31 }

```

### 优化版 - 贪心算法:

根据样例 3 1 2 1 8 5 6, 我们发现 3 可以接 8 组成最长上升子序列; 同样, 3 后面的 1 同样可以接 8; 可以接到 3 的后面的数字一定能够接到比 3 小的数字的后面组成 最长上升子序列。

根据最长上升子序列的长度来分类; 多个最长上升子序列中求解出长度为  $i$  的数字, 而且每一个长度结尾数值的最小值 (取最小值, 保证了后面能够拼接的数字范围更大);

最长上升子序列的长度越长, 子序列结尾数字的数值大小越大 (不存在, 结尾数字的数值大小相等的情况), 严格单调递增 (当满足单调递增的情况下, 可以使用二分法);

思想: 拼接

根据上面的结论, 数字  $a_i$  可以拼接到一个结尾数字的数值小于  $a_i$  的最长上升子序列的后面, 我们只要将  $a_i$  拼接能够拼接的最长上升子序列结尾数字的最大值的后面 (二分法: 找到小于某一个数的最大的数字), 就能够得到 长度加一 的最长上升子序列。

记得更新当前长度的最长上升子序列结尾数字的最小值。

时间复杂度:  $O(n * \log n)$

代码示例:

```

1  #include <iostream>
2  #include <cstdio>
3  #include <algorithm>
4
5  using namespace std;
6
7  const int N = 1e6 + 10;
8  int n;
9  int a[N];
10 int q[N];    // q[i] 表示当前长度是i的最长上升子序列的末尾元素值
11
12 int main()
13 {
14     scanf("%d", &n);
15     for (int i = 0; i < n; ++i) scanf("%d", &a[i]);
16
17     int len = 0;    // 最长上升子序列的长度
18     q[0] = -2e9;    // 设置边界
19
20     // 通过二分法, 查找最长上升子序列结尾q[i]小于a[i]的最大值

```

```

21     for (int i = 0; i < n; ++i)
22     {
23         // 二分 (区间)
24         int l = 0, r = len;
25         while (l < r)
26         {
27             int mid = (l + r + 1) >> 1;
28             if (q[mid] < a[i]) l = mid;
29             else r = mid - 1;
30         }
31         len = max(len, r + 1);
32         // 拼接
33         q[r + 1] = a[i];
34     }
35     printf("%d\n", len);
36     return 0;
37 }

```

练习和提高：

### [AcWing1017.怪盗基德的滑翔翼](#)

思路 - 如何转化成 LIS 问题： 当怪盗基德从某一栋房子开始向左跳时：

我们可以把这个问题当作 从最左端开始/从左向右看的 LIS 问题；

当怪盗基德从某一栋房子开始向右跳时：

我们可以把这个问题当作 从最右端开始/从右向左看的 LIS 问题；

最终：

两个方向的 LIS 问题 求解

### [AcWing1014.登山](#)

分析题目： 1) 根据 每次浏览的景点海拔都要大于前一个； 说明这个问题涉及到了 子序列。

2) 相邻两个景点的高度不能相同， 一旦开始下降，就不能上升了； 说明子序列满足 先严格单调递增然后严格单调递减。

3) 求最多能浏览多少景点。

目标：

所有形状是 先严格单调递增然后严格单调递减（金字塔形状）的子序列长度的最大值。

状态表示：  $f[i]$

集合表示： 所有以  $a[i]$  为峰值的子序列

属性： 最大值

状态计算：

集合划分：

1) 当峰值为  $a[0]$  时，子序列的最大长度

2) 当峰值为  $a[1]$  时，子序列的最大长度

...

3) 当峰值为  $a[n]$  时，子序列的最大长度

当  $a[k]$  为峰值的时候，左边的上升子序列和右边上升子序列的最大值求解是独立的；可以分别求出左右两边的上升子序列的最大值  $f[k]$  和  $g[k]$ ；

注意,分段求解需要将额外的计算1次  $a[k]$  减掉

状态方程：

$$f[i] = f[j] + 1$$

$$g[i] = g[j] + 1$$

$$sum = f[i] + g[i] - 1$$

优化 - 打表：

从  $O(n^3)$  优化成  $O(n^2)$

```
1 // 未优化
2 #include <iostream>
3 #include <cstdio>
4 #include <algorithm>
5
6 using namespace std;
7
8 const int N = 1010;
9 int n;
10 int a[N], f[N], g[N];
11
12 int main()
13 {
14     scanf("%d", &n);
15     for (int i = 1; i <= n; ++i) scanf("%d", &a[i]);
16     // 左边
17     for (int i = 1; i <= n; ++i)
18     {
19         f[i] = 1;
20         for (int j = 1; j < i; ++j)
21         {
22             if (a[j] < a[i]) f[i] = max(f[i], f[j] + 1);
23         }
24     }
25     // 右边
26     for (int i = n; i >= 1; --i)
27     {
28         g[i] = 1;
29         for (int j = n; j > i; --j)
30         {
31             if (a[j] < a[i]) g[i] = max(g[i], g[j] + 1);
32         }
33     }
34     // 注意：处理额外的一次
35     int res = 0;
36     for (int i = 1; i <= n; ++i) res = max(res, f[i] + g[i] - 1);
37     printf("%d\n", res);
38     return 0;
39 }
```

[AcWing482.合唱队形](#)

思路：同[AcWing1014.登山](#) 仅仅对最后的结果做处理即可。



提示：（对偶）保留的人数尽可能多，剔除的人数就会变少了

## [AcWing1012.友好城市](#)

难点：思路，非代码上

技巧：转化

题目分析：1) 每个城市上只能建一座桥

2) 所有桥与桥之间不能相交

求解建桥的数量最大

状态表示：

集合：

首先，将南北两岸的城市在纸上画出来用两行表示（上面一行对应北岸，下面一行对应南岸）；由于友好城市的关系是一一对应的，我们可以将某一行进行排序（另一行的顺序也会造成改变）；

比如，对下面一行（南岸）城市的位置进行升序排序，然后从下面一行（南岸）的城市向其对应的城市建立友好关系；

根据题目中的要求桥与桥之间不能相交，所以只要保证从下面一行（南岸）的城市与其对应的友好城市（上面一行）位置/坐标是上升子序列（单调的）即可求解出该问题；

通过上面的分析，我们得到了两个集合：所有合法的建桥方式和上升子序列

属性：最大值

状态计算：

参考 最小上升子序列（朴素）

使用了双关键字排序方法，防止TLE。

代码示例：

```
1  #include <iostream>
2  #include <cstdio>
3  #include <algorithm>
4
5  using namespace std;
6  typedef pair<int,int> pii;
7
8  const int N = 5010;
9  int n;
10 pii q[N];
11 int f[N];
12
13 int main()
14 {
15     scanf("%d", &n);
16     for (int i = 0; i <= n; ++i) scanf("%d%d", &q[i].first, &q[i].second);
17
18     sort(q, q + n);    // 对北岸进行排序（与分析中举例相反）
19
20     // DP-最小上升子序列
21     int res = 0;
22     for (int i = 0; i < n; ++i)
23     {
24         f[i] = 1;
25         for (int j = 0; j < i; ++j)
```

```

25         if (q[j].second < q[i].second)
26             f[i] = max(f[i], f[j] + 1);
27         res = max(res, f[i]);
28     }
29     printf("%d\n", res);
30     return 0;
31 }
32 ///////////////另一种写法 (细节上不一样) //////////////////////////
33 #include <iostream>
34 #include <cstdio>
35 #include <algorithm>
36
37 using namespace std;
38 typedef pair<int,int> pii;
39
40 const int N = 5010;
41 int n;
42 int f[N];
43 pii q[N];
44
45 int main()
46 {
47     scanf("%d", &n);
48     for (int i = 1; i <= n; ++i) scanf("%d%d", &q[i].first, &q[i].second);
49     sort(q + 1, q + n + 1);
50     // DP-LIS
51     int res = 0;
52     for (int i = 1; i <= n; ++i)
53     {
54         f[i] = 1;
55         for (int j = 1; j < i; ++j)
56             if (q[j].second < q[i].second)
57                 f[i] = max(f[i], f[j] + 1);
58         res = max(f[i], res);
59     }
60     printf("%d\n", res);
61     return 0;
62 }

```

### [AcWing1016.最大上升子序列和](#)

求解目标：

满足：1) 上升子序列 2) 子序列之和最大值

不一定是最长上升子序列，但是一定要满足上升子序列之和最大值

题目分析：

状态表示：

集合：所有以 $a[i]$ 结尾的上升子序列

属性：每一个上升子序列的 和的最大值

状态计算：

集合的划分：

划分依据：从最后一步考虑它的前一步；

按照以 $a[i]$ 结尾的上升子序列，前一个数的值（倒数第二个数）划分；

- 1)  $a[i]$  的前面是空的;
- 2) 倒数第二个数是  $a[1]$  ;
- 3) 倒数第二个数是  $a[2]$  ;
- ...
- 4) 倒数第二个数是  $a[i - 1]$  ;

状态方程:

当  $a[i]$  前面是空的, 最大值为  $a[i]$ ;

当倒数第二个数是  $a[k]$  时, 上升子序列的最大值就是以  $a[k]$  结尾的上升子序列最大值  $f[k]$  再加  $a[i]$  ;

$$f[i] = f[i] + a[i]$$

```

1  #include <iostream>
2  #include <cstdio>
3  #include <algorithm>
4
5  using namespace std;
6  const int N = 1010;
7  int a[N];
8  int f[N];
9  int n;
10
11 int main()
12 {
13     scanf("%d", &n);
14     for (int i = 1; i <= n; ++i) scanf("%d", &a[i]);
15
16     int res = 0;
17     for (int i = 1; i <= n; ++i)
18     {
19         f[i] = a[i];
20         for (int j = 1; j < i; ++j)
21         {
22             if (a[j] < a[i])
23                 f[i] = max(f[i], f[j] + a[i]);
24         }
25         res = max(f[i], res);
26     }
27     printf("%d\n", res);
28     return 0;
29 }
```

## 5.2.3 最长公共子序列 LCS (Longest Common Subsequence)

### 5.2.3.1 题目链接及样例解释:

题目链接:

[AcWing897.最长公共子序列](#)

样例解释:

```

1  acbd
2  abedc
```

最长公共子序列及长度: abd 3

子序列解释: 同 LIS。

### 5.2.3.2 算法流程

状态表示:

$f[i, j]$  二维的状态

集合: 所有由第一个序列的前  $i$  个字母和第二个序列的前  $j$  个字母能够组成的公共子序列 (所有在第一个序列的前  $i$  个字母中出现, 而且在第二个序列的前  $j$  个字母中同样出现的子序列)

属性: 最大值

状态计算:

集合的划分:

从当前是最后一步来考虑, 或者说, 从上一状态  $f[i-1][j-1]$  到当前状态;

以  $a[i]$  和  $b[j]$  是否存在于公共子序列当中; 一共四种情况:

1)  $a[i]$  不存在,  $b[j]$  不存在, 最长公共子序列的长度不变, 仍然为前第一行  $i-1$  个字母和第二行前  $j-1$  能够构成的最长公共子序列的长度;

$f[i-1, j-1]$

2)  $a[i]$  不存在,  $b[j]$  存在;

不能直接用  $f[i-1, j]$  这个状态来表示, 但是  $f[i-1, j]$  却包含了这种情况;

3)  $a[i]$  存在,  $b[j]$  不存在;

不能直接用  $f[i, j-1]$  这个状态来表示, 但是  $f[i, j-1]$

4)  $a[i]$  存在,  $b[j]$  存在, 公共子序列长度加1;

$f[i-1, j-1] + 1$

状态方程:

最终结果就是, 从上面的四种情况里选出, 最大值即可;

$f[i, j] = \max(f[i-1, j], f[i, j-1], f[i-1][j-1] + 1)$

代码示例:

```
1  #include <iostream>
2  #include <cstdio>
3  #include <algorithm>
4
5  using namespace std;
6  const int N = 1010;
7  int n, m;
8  char a[N], b[N];
9  int f[N][N];
10
11 int main()
12 {
13     scanf("%d%d", &n, &m);
14     scanf("%s%s", a + 1, b + 1);
15
16     for (int i = 1; i <= n; ++i)
17         for (int j = 1; j <= m; ++j)
18             {
```

```
19         f[i][j] = max(f[i - 1][j], f[i][j - 1]);
20         if (a[i] == b[j]) f[i][j] = max(f[i][j], f[i-1]);
21     }
22     printf("%d\n", f[n][m]);
23     return 0;
24 }
```

## 第六部分 贪心算法

---

### 6.1 区间问题

#### 6.1.1

## 第七部分 时空复杂度

---