ACM/ICPC 代码库

2015 年 5 月 5 日

目录

第一章　基本算法

1.1　模拟

1.2　贪心

1.2.1　背包问题

1.2.1.1　最优装载问题

问题　给出 n 个物体，第 i 个物体的重量为 w_i。选择尽量多的物体，使得总重量不超过 C。

算法　只需把所有问题按重量从小到大排序，然后从最轻的开始选，直到无法选择为止。

1.2.1.2　部分背包问题

问题　给出 n 个物体，第 i 个物体的重量为 w_i，价值为 v_i。选择尽量多的物体，使得总重量不超过 C。

算法　把所有物体按照性价比（价值除以重量的值）排序，然后优先拿性价比高的，直到总重量正好为 C。实际上，按照这种取法，除了最后一个被拿的物体外，其他物体要么全部拿走，要么没拿。

1.2.1.3　乘船问题

问题　有 n 个人，第 i 个人重量为 w_i。每艘船最大载重量均为 C，且最多能乘两个人。请用最少的船装载所有人。

算法　从最轻的人开始，每个人都找一个能和他同船，而且重量最重的人。重复这个过程，直到没有人可以找到伙伴为止（剩下的人就一人一条船了）。

1.2.2 区间上的问题

1.2.2.1 选择不相交区间

问题 数轴上有 n 个开区间 (a_i, b_i)。选择尽量多的区间，使这些区间两两没有公共点。

思路 贪心策略如下：

1. 按 b_i 从小到大的顺序排序。
2. 务必选择第一个区间。（原因见图 1.2.2.1）
3. 继续从前向后遍历。每当遇到可以选择的区间（与上一区间没有公共点），就选择它。

图 1.1: 选择不相交区间：如图所示，不论区间 1、2 的相对位置如何，选择区间 1 都会为以后的选择留下更大的剩余空间。

1.2.2.2 区间选点问题

问题 数轴上有 n 个闭区间 $[a_i, b_i]$。取尽量少的点，使得每个区间内都至少有一个点（不同区间内含有的点可以是同一个）。

思路 贪心策略如下：

1. 把所有区间按 b_i 从小到大排序（b_i 相同时，a 从大到小排序[1]）。
2. 然后，一定取第一个区间的右端点。（原因见图 1.2.2.2）
3. 继续从前向后遍历，当遇到覆盖不到的区间时，选取这个区间的右端点。

1.2.2.3 区间覆盖问题

问题 数轴上有 n 个闭区间 $[a_i, b_i]$。选择尽量少的区间来覆盖指定线段 $[s, t]$。

[1]考虑区间包含的情况：小区间被满足时大区间一定被满足。所以我们应当优先选取小区间中的点，这样大区间不必考虑。

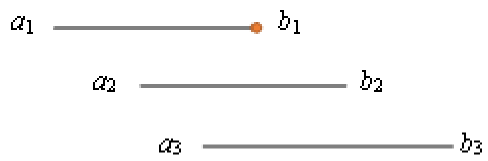

图 1.2: 区间选点问题：对于第 1 个区间来说，显然，选择它的右端点是明智的。因为它比前面的点能覆盖更大的范围。

思路 贪心策略如下：

1. 预处理，扔掉不能覆盖 $[s,t]$ 的区间。如果区间边界超过了 $[s,t]$，就把它变成 s 或 t，以免影响后面的处理。

2. 把各区间按 a 从小到大排序。如果区间 1 的起点不是 s，无解，否则选择起点在 s 的最长区间。

3. 选择此区间 $[a_i,b_i]$ 后，问题就转化成了起点为 b_i 的区间 $[b_i,t]$。继续选择，直到找不到区间覆盖当前起点，或者 b_i 已经到达线段末端。

1.2.3 Huffman 编码

1.2.4 两个调度问题

1.3 递归和分治

1.3.1 分治法的思考方法

1.3.2 二分法与三分法

1.3.3 最近点对问题

1.4 递推

1.5 构造

1.6 回溯法

1.6.1 连续邮资问题

1.7 离散化

第二章　搜索

2.1　盲目搜索

2.1.1　纯随机搜索

2.1.2　深度优先搜索

2.1.3　广度优先搜索

2.1.4　迭代加深搜索

2.1.5　双向广度优先搜索

2.2　启发式搜索

2.2.1　A× 算法

2.2.2　IDA× 算法

2.3　博弈论

2.3.1　极大极小过程

2.3.1.1　Min-Max 算法

2.3.1.2　alpha-beta 剪枝

2.3.2　组合游戏

2.3.3　Nim 过程

2.4　记录路径

2.5　搜索的优化

2.5.1　剪枝

2.5.2　判重

2.5.3　预处理

第三章　数据结构

3.1　串

3.1.1　C++ 中的字符串类

3.1.1.1　string

有些字符串操作本来很简单，但是在 C 语言里就很麻烦。C++ 通过 string 类，在一定程度上缓解了这个问题。string 的头文件为 <string>。

注意，string 类虽然方便，但是速度很慢！

输入/输出　cin、cout 可以直接输入、输出 string 字符串。如果需要整行读入，可以用 getline 函数：

```
string st;
cin>>st;

while (getline(cin, st))
{
    cout<<st;
}
```

基本操作　string 类相当于 vector<char>，所以 string 可以使用 vector 所支持的大部分操作，例如：

```
string st="abcde";
cout << st[1];
st.push_back('f');
```

- 可以用 "+" 连接两个字符串。当然，两个字符串中需要至少一个是 string 类型，否则会编译错误。*由于涉及内存分配，不建议使用。*
- 用 ==、!= 判断两个字符串是否相等。用比较符号比较两个字符串的大小。

- 用 += 运算符或 st.append() 在 st 后面附加字符串。
- 用 st.size() 获得字符串 st 的长度。
- 用 st.find() 在字符串里寻找子串。如果没找到，函数会返回 string::npos。

```
size_t p = st.find("hello", 0); // 第二个参数表示起始位置。
```

- 用 st.substr(5, 10) 来截取从第 5 个字符开始、长度为 10 的字符串。如果长度为 string::npos，则一直截取到字符串结束。
- 用 st.c_str() 获取一个字符串数组，这样就可以继续用 C 语言的处理方式来处理字符串了。

3.1.1.2 stringstream

stringstream 可用于从字符串中读取内容。头文件 <sstream>。

注意，stringstream 的速度非常慢。

以下是“输入若干行整数，求它们的和”的代码：

```
string st;
while (getline(cin, st))
{
    stringstream ss(st);
    int sum=0, p;
    while (ss>>p) sum+=p;
    cout<<p<<endl;
}
```

3.1.2 串匹配

问题：已知原串和模式串，求模式串在原串中的个数和位置。

对于此类问题，很容易想到朴素算法，或者是 strstr() 与 string 的 find()。不过，在字符串很长的情况下，朴素算法会浪费时间。*C 和 C++ 标准也没有对 strstr 和 find 的实现方式做出*规定，而且我们碰到的问题也并不总是单纯的字符串，所以我们需要了解更高效的算法。

常用的串匹配算法有 KMP 算法、BM 算法、KR 算法等。

3.1.2.1 KMP 算法

KMP 算法的原理 假设原串 S 为 abababacabcababacbc，模式串 P 为 abababacb。

按照朴素算法的做法，发现不匹配之后，应该把模式串向右移动一位。而 KMP 算法会根据预处理的结果来决定应该把模式串向右移动多少位。

朴素算法的尝试：

```
S:  abababacabababababacbc
P:  ababacb
P1:  ababacb
P2:   ababacb
P3:    ababacb
......
Pn:             ababacb
```

KMP 算法的尝试：

```
S:  abababacabababababacbc
P1: ababacb                // P[5] 时失败，向右移动 2 位，下次从 P[3] 开始
P2:   ababacb              // P[6] 时失败，向右移动 6 位，下次从 P[0] 开始
P3:         ababacb        // P[5] 时失败，向右移动 2 位，下次从 P[3] 开始
P4:           ababacb      // P[5] 时失败，向右移动 2 位，下次从 P[3] 开始
P5:             ababacb    // 成功
```

现在看一下 KMP 是怎样知道模式串向右移动的位数的。为了看得清楚，我们进行一下处理：

```
S:  ababa bac ababa ba bacbc
P1: ababa|cb
P2:   aba|bac|b
P3:          |ababa|cb
P4:             aba|ba|cb
P5:               a ba|bacb
```

我们在使匹配失败的字母的前面加上了竖线。设 P[0:x] 为"到目前为止，成功匹配部分的长度为 x"。可以看到，在每组竖线的前面，移动后的 P[0:j] (在继续匹配之前，P 只剩 j 个字母) 和移动前的 P[0:i] 的每一个字母仍然是对应的。也就是说，*P[0:j]* 实际上是 *P[0:i]* 的前缀。这样，我们就可以减少很多无效的右移。

所以，我们要做的就是开一个 next 数组。对于 next[j]=k，我们认为 *P[0:k]* 是 *P[0:j]* 的最长的前缀。把 next 数组算好之后，就可以正式开始和原串的匹配了。

实现 以下是 KMP 算法的代码：

```c
int next[100];
int KMP(char *str, char *pat)
{
    int sln=strlen(str);
    int pln=strlen(pat);
    int i,j;

    next[0]=-1;
    for (i=0,j=-1; i<pln; )
        if (j==-1 || (pat[i]==pat[j]))
```

```
            next[++i]=++j;
        else
            j=next[j];

    for (i=j=0; i<sln and j<pln; )
        if (j==-1 || (str[i]==pat[j]))
            ++i,++j;
        else
            j=next[j];

    return (j==patln) ? (i-j) : -1;
}
```

3.1.3 Karp-Rabin 算法

3.1.4 最短公共祖先

3.1.4.1 两个字符串

3.1.4.2 多个字符串

3.2 二分查找

想必大家都知道二分查找是什么东西了，也知道时间复杂度为 $O(\log n)$ 了。

二分查找要求数据单调有序。

3.2.1 实现

```
// 数据范围 a[x..(y-1)]，为左闭右开区间。
int binarySearch(int *a, int x, int y, int v)
{
    while (x<y)
    {
        int mid=x+(y-x)/2;
        if (v==a[mid])
            return mid;
        else if (v<a[mid])
            y=mid;
        else
            x=mid+1;
    }
    return -1;
}
```

3.2.2 二分查找求上下界

【求下界】寻找 a 的区间 $[x, y)$ 中大于等于 v 的第一个数，使得 v 插入到 a 的对应位置以后，a 还是一个有序的数组。

【求上界】二分查找求下界的描述：寻找 a 的区间 $[x, y)$ 中大于 v 的第一个数，使得 v 插入到 a 对应位置的前面一位之后，a 还是一个有序的数组。

实现代码如下：

```
// 左闭右开区间 a[x..y-1]
int lowerBound(int *a, int x, int y, int v)
{
    while (x<y)
    {
        int mid=x+(y-x)/2;
        /*
            v == a[mid]: 至少找到一个等于 v 的，但前面可能还有。
            v < a[mid]: v 肯定不能在 mid 的后面。
            v > a[mid]: mid 和前面都不可以。
        */
        if (v<=a[mid]) y=mid; else x=mid+1;
    }
    return x;
}

int upperBound(int *a, int x, int y, int v)
{
    while (x<y)
    {
        int mid=x+(y-x)/2;
        /*
            v == a[mid]: 答案肯定在后面，因为不可能等于 v。
            v < a[mid]: mid 和后面都不可以。
            v > a[mid]: 应该在 mid 的后面。
        */
        if (v<a[mid]) y=mid; else x=mid+1;
    }
    return y;
}
```

3.2.3 STL 中的二分查找

STL 中有二分查找的算法，位于头文件 <algorithm> 中。

- binary_serach(begin, end, value[, comp])

判断已序区间 [begin, end) 内是否包含和 value 相等的元素。如果省略 comp 则使用“<”运算符进行比较。注意，返回值只说明是否存在，不指明位置。

- lower_bound(begin, end, value[, comp])

  返回第一个大于等于 value 的元素位置，即可以插入 value 且不破坏区间有序性的第一个位置。省略 comp 则使用“<”比较。

- upper_bound(begin, end, value[, comp])

  返回第一个大于 value 的元素位置，即可以插入 value 且不破坏区间有序性的最后一个位置。省略 comp 则使用“<”比较。

3.2.4 二分答案

许多问题不易直接求解，但我们可以用很短的时间来判断某个解是否可行，而且已知候选答案的范围 [min,max]。这时我们大可不必通过计算得到答案，只需在此范围内应用“二分”的过程，逐渐靠近答案！

最大值最小化问题（“使最大值最小”）等常用此做法。

当然不是任何题目都适合使用“二分答案”。能够二分答案的题目有以下几个特点：

- 候选答案必须是离散的，且答案在一个上下界确定的范围里。
- 候选答案在区间 [min,max] 上是有序的。
- 很容易去判断某个值是不是答案。

3.3 排序

3.3.1 快速排序

快速排序是基于比较的排序算法中最快的算法。

```
// 备注：(1) 左闭右开区间 (2) 需要 swap 函数
void quickSort(int *start, int *end)
{
    if (start +1 >= end) return;
    int *high=end, *low=start;

    // 划分：把比 *start 小的数据放到它的左侧，否则放右侧。
    while (low<high)
    {
```

```
        while (++low<end && *low<=*start);
        while (--high>start && *high>=*start);
        if (low<high) swap(*low, *high);
    }
    swap(*high, *start);
    quickSort(start,high);
    quickSort(low,end);
}
```

快速排序的时间复杂度为 $O(n\log n)$，但是极端情况（数据基本有序）下会退化成 $O(n^2)$。因此建议大家使用 STL 的 sort() 函数。STL 的 sort() 与下面代码相比，具有以下特点：

- 数据量大时采用分段递归排序，即快速排序。在取分隔点时，取的是头部、尾部和中央三个元素的中间值。
- 数据量变小的时候，采用插入排序代替快速排序。
- 如果递归层次过深，会改用堆排序。

此外，快速排序是不稳定的排序算法。

3.3.2 归并排序

归并排序的时间复杂度为 $O(n\log n)$，但是空间复杂度很大，为 $O(n)$。

归并排序是稳定的排序算法，即数值相同时，元素的相对位置不会发生改变。

STL 的 stable_sort() 采用了归并排序算法。

```
int temp[N];
void mergeSort(int *start, int *end)
{
    if (start+1>=end) return;
    // 划分阶段、递归
    int *mid = start+(end-start)/2;
    mergeSort(start, mid);
    mergeSort(mid, end);

    // 将 mid 两侧的两个有序表合并为一个有序表。
    int *p=start,*q=mid,*r=temp;
    while (p<mid || q<end)
        if (q>=end || (p<mid && *p<=*q))
            *(r++)=*(p++);
        else
            *(r++)=*(q++);

    for (p=start, r=temp; p<end; p++, r++) *p=*r;
}
```

3.3.3 堆排序

堆排序的时间复杂度为 $O(n \log n)$。但是由于该算法常数因子有些大，因此它比快速排序慢很多。不过它不需要递归，所以不怕爆栈。

堆排序的思路：

1. 将整个数组转化为一个堆。如果想把一串数从小到大排序，则需要使用最大值堆[1]。
2. 将堆顶的最大元素取出，并把它放到数组的最后。
3. 剩余元素重新建堆。
4. 重复第 2 步，直到堆为空。

3.3.4 简单排序算法

3.3.4.1 插入排序

插入排序的时间复杂度为 $O(n^2)$。但是在“数据几乎有序”的情况下，该算法的速度并不是很慢。在数据规模不大时，可用插入排序来代替快速排序。

3.3.4.2 冒泡排序

3.3.4.3 选择排序

选择排序的时间复杂度为 $O(n^2)$。由于该算法最好的时间复杂度也是 $O(n^2)$，并且该算法不稳定，所以很少使用。

3.3.5 非比较排序

在基于比较的排序算法中，最快的速度为 $O(n \log n)$。实际上，在某些情况下，还有更快的算法。

3.3.5.1 基数排序

3.3.5.2 桶排序

3.3.5.3 计数排序

3.3.6 第 k 大元素

思路和快速排序相近，也是找一个数，把比它小的放到它左面，比它大的放到右面。不同的是，进行“递归”时只需对其中一侧进行操作，也就是对有第 k 大 (k 从 1 开始) 元素的那一部分

[1] 如果用最小值堆会占用大量额外空间。

进行操作。理想情况下，该算法的复杂度能达到线性水平。

```
int part(int *a, int start, int end)
{
    int low=start, high=end;
    int temp, check=a[start];
    do
    {
        while (a[high]>=check && low<high) high--;
        if (a[high]<check) temp=a[high], a[high]=a[low], a[low]=temp;
        while (a[low]<=check && low<high) low++;
        if (a[low]>check) temp=a[high], a[high]=a[low], a[low]=temp;
    }
    while (low!=high);
    a[low]=check;
    return low;
}

int find(int *a, int start, int end, int k)
{
    if (start==end) return a[start];
    // 计算p位置的“排名”
    int p = part(a, start, end);
    // 只对包含第k小元素的部分进行查找和排序。
    int q = p-start+1;
    if (k <= q)
        return find(a, start, p, k);
    else
        return find(a, p+1, end, k-q);
}
```

3.3.7 排序与 STL

关于排序的算法均在头文件 <algorithm> 中。排序算法需要动用随机存取迭代器，所以不能对 list、set、map 等使用排序算法 (list 内置排序函数)。

以下各算法，如果不自行提供比较函数，则使用“<”运算符进行比较。区间均为*左闭右开*区间。

- sort(begin, end[, comp])：对区间内所有元素排序。时间复杂度 $O(n \log n)$。
- stable_sort(begin, end[, comp])：同样是全排序，但会保持相等元素原来的相对次序。
- partial_sort(begin, sortEnd, end[, comp])：局部排序。对区间 [begin, end) 内的元素排序，使区间 [begin, sortEnd) 内的元素有序。时间复杂度介于 $O(n)$ 和 $O(n \log n)$。

- nth_element(begin, pos, end[, comp]): 寻找从小到大排名第 k 的元素。pos 也是迭代器，并且 pos = begin+k-1。平均时间复杂度为 $O(n)$。

- partition(begin, end, pred): 对元素进行分类。对于每个元素 x，如果 pred(x) 为真则归入第一组，否则归入第二组。返回值为指向第二组第一个元素的迭代器。平均时间复杂度为 $O(n)$。

- stable_partition(begin, end, pred): 同上，只不过会保持相等元素的相对次序。

以上各算法耗时，从小到大分别为：partition、stable_partition、nth_element、partial_sort、sort、stable_sort。

3.4 哈希表

3.4.1 基本实现

3.4.2 字符串的哈希函数

3.5 并查集

并查集是若干个不相交集合，能够较快地实现合并和判断元素所在集合的操作（可认为是常数）。并查集可用于求无向图的连通分量个数、最小公共祖先、带限制的作业排序，还有最小生成树（Kruskal 算法）。

3.5.1 并查集

问题 有 n 个元素，最初没有哪个集合内有两个或以上的元素。每次进行以下两种操作之一：

1. 将其中两个集合合并。
2. 判断两个元素是否属于同一集合。

3.5.2 朋友 -敌人模型

3.6 二叉树

3.6.1 基本二叉树

二叉树是递归定义的：它要么为空，要么由根结点、一个左子树和一个右子树构成，且左子树和右子树都是二叉树。

3.6.1.1 实现

二叉树可以用数组保存，也可以用链表保存。根据问题的实际需求，链表既可以只保存左右子树结点，也可以把父节点包括进去。

```
const int MAX=1000;

struct node
{
    int value;
    node *parent;        // 根据实际需要来添加
    node *left, *right;
};

int n=0;
node tree[MAX], *head=NULL;

inline node *createNode()
{
    node *t=&tree[++n];
    t->value=0; t->parent=NULL; t->left=t->right=NULL;
    return t;
}

void preOrder(node *p)  // 前序遍历
{
    if (p==NULL) return;
    cout<<p->value<<' '; preOrder(p->leftchild); preOrder(p->rightchild);
}
void inOrder(node *p)   // 中序遍历
{
    if (p==NULL) return;
    inOrder(p->leftchild); cout<<p->value<<' '; inOrder(p->rightchild);
}
void postOrder(node *p) // 后序遍历
{
    if (p==NULL) return;
    postOrder(p->leftchild); postOrder(p->rightchild); cout<<p->value<<' ';
}
```

对于创建结点函数，有人希望能够把树根直接扔到函数参数里，而不是写成赋值的形式。这时可以这样定义函数：

```
    void createNode(node* & p)
```

引用是 C++ 的概念。上面代码表示“代表一个指针的引用”。

3.6.1.2 完全二叉树

一棵深度为 k，且有 2^k-1 个节点的二叉树，称为满二叉树。在一棵二叉树中，除最后一层外，若其余层都是满的，并且最后一层要么是满的，要么是在右边缺少连续若干节点，则此二叉树为完全二叉树。

由于完全二叉树排列密集，所以可以用一个一维数组来保存二叉树而不造成太大浪费。

假设一个完全二叉树的结点数为 n，树根的序号是 0，那么可根据表 3.6.1.2 来定位。

关系	函数	备注
r 的父亲	$(r-1)/2$	$r \neq 0$
r 的左儿子	$r*2+1$	$2r+1<n$
r 的右儿子	$r*2+2$	$2r+2<n$
r 的左兄弟	$r-1$	r 为偶数且 $0<r<n$
r 的右兄弟	$r+1$	r 为奇数且 $r+1<n$
r 是否为叶子?	$r \geqslant n/2$	$r<n$

表 3.1: 完全二叉树各结点关系

3.6.2 哈夫曼树

哈夫曼编码（Huffman Coding）是一种编码方式，是一种用于无损数据压缩的熵编码（权编码）算法。该算法使用变长编码表，概率较高的字母，编码长度较短，反之亦然。

哈夫曼编码值得注意的一点是，一个字符的编码不能是另一个字符编码的前缀，否则会引起歧义。

哈夫曼树又称最优二叉树，是一种带权路径长度最短的二叉树。所谓树的带权路径长度，就是树中所有的叶结点的权值乘上其到根结点的路径长度（若根结点为 0 层，叶结点到根结点的路径长度为叶结点的层数）。树的路径长度是从树根到每一结点的路径长度之和，记为 $W_{PL}=(W_1*L_1+W_2*L_2+W_3*L_3+\cdots+W_n*L_n)$，$N$ 个权值 W_i $(i=1,2,\cdots n)$ 构成一棵有 N 个叶结点的二叉树，相应的叶结点的路径长度为 L_i $(i=1,2,\cdots n)$。可以证明哈夫曼树的 W_{PL} 是最小的。

建立哈夫曼树的算法是贪心算法, 其基本思想: (1) 权越大, 离根越近。(2) 自底向上建树。

具体步骤如下:

1. 首先, 创建 n 个初始的 Huffman 树, 每棵树只包含单一的叶结点, 叶结点记录对应字母。
2. 拿走权最小但没有被处理的两棵树, 再把它们标记为 Huffman 树的叶结点。

3. 把这两个叶结点标记为一个分支结点的两个子结点，而这个结点的权即为两个叶结点的权之和。
4. 重复步骤 2 和 3，直到序列中只剩下一个元素。

调用 makeHuffman 函数时，需要事先将各个元素的权值放到一个数组中。函数的返回值代表 Huffman 树的根结点。

```
const int INF=9999999, M=100;

struct node
{
    int w;
    node *parent, *leftchild, *rightchild;
} h[M];

// weight[i]表示结点i的权值。返回值是Huffman的树根。
node *makeHuffman(int *weight, int n)
{
    node *p1, *p2;

    memset(h,0,sizeof(h));
    for (int i=0; i<n; i++) h[i].w=weight[i];
    int m=2*n-1;

    for (node *np=h+n; np<h+m; np++)
    {
        int min1=INF, min2=INF;
        for (node *op=h; op<np; op++)
        {
            if (op->parent==0)     // 处理未处理节点
            {
                if (op->w < min1) // 选择权值最小的两个点
                    min2=min1, min1=op->w, p2=p1, p1=op;
                else if (op->w < min2)
                    min2=op->w, p2=op;
            }
        }
        p1->parent = p2->parent = np;
        np->leftchild = p1;
        np->rightchild = p2;
        np->w = p1->w + p2->w;
    }
    return &h[2*n-2];
}
```

3.6.3 字典树 (Trie)

3.7 二叉排序树

3.7.1 基本的二叉排序树

3.7.2 Treap

3.7.3 Size Balanced Tree (SBT)

3.7.4 C++ 的 set 和 map

map (映射)、multimap (多重映射)、set (集合)、multiset (多重集合) 属于关联容器，头文件分别位于 <map> 和 <set> 中。

map 和 set 的区别：set 实际上就是一组元素的集合，但其中所包含的元素的值是唯一的，且是按一定顺序排列的。集合中的每个元素被称作集合中的实例。其内部通过链表的方式来组织；而 map 提供一种“键—值”关系的一对一的数据存储能力，类似于字典。其“键”在容器中不可重复，且按一定顺序排列。由于其是按链表的方式存储，它也继承了链表的优缺点。

multiset 和 set 不同之处在于，multiset 中元素的值可以不唯一。multimap 也类似，在 multimap 中“键”可以不唯一。

关联容器的特点：

1. 关联容器对元素的插入和删除操作比 vector 快，但比 list 慢。
2. 关联容器对元素的检索操作比 vector 慢，但是比 list 要快很多。关联容器查找的复杂度基本是 $O(\log n)$。
3. set 是内部排序的，这与序列容器有着本质的区别。

下面是有关 set 和 multiset 的用法：

```
// 1. 定义
set < int, less<int> > s1;  // 集合内部升序排列
set < int, greater<int> > s2; // 集合内部降序排列
// 和其他容器一样，set 也可以用预定义的区间来初始化。

// 2. 查询
s.count(10);  // 返回 s 中值为 10 的具体数目。但对于 set 来说，返回值不是 0 就是
   1。
s.empty();    // 判断集合是否为空集。
s.size();     // 返回集合的元素数量。

// 3. 插入和删除
```

```
s.insert(e);  // 将e插入到set中。
// 注意，insert 的返回值是一个 pair，其 first 是指向插入后元素的迭代器，
// second 表示插入是否成功（如果其 second 为 false，说明元素已经存在）。
s.insert(begin, end); // 将区间 [begin, end) 中的值插入到 s 中。
s.erase(e);           // 将 e 删除并返回剩余的 e 的数量。
s.erase(pos);         // 将 pos 处的元素删除。
s.erase(begin, end); // 将 [begin, end) 处的元素删除。

// 4. 迭代器：begin、end、rbegin、rend 分别返回正向和反向迭代器。
```

下面是有关 map 和 multimap 的用法：

```
// 1. 定义
map<int, string> m;  // 键的类型是 int，值的类型是 string。
// 和其他容器一样，map 也可以用预定义的区间来初始化。

// 2. 查询
// m.at(3) 或 m[3]：返回一个引用，指向键为3时的对应值。注意，它不是数组下标！
// 如果元素不存在，map 会自动建立这个元素。
m.count(3);   // 返回 s 中键为 3 的具体数目。但对于 map 来说，返回值不是 0 就是
    1。
m.find(3);    // 返回指向键为 3 的元素的迭代器。如果不存在，则返回 m.end()。
m.empty();    // 判断映射是否为空映射。
m.size();     // 返回映射的元素数量。

// 3. 插入和删除
m.insert(begin, end) // 将区间 [begin, end) 中的值插入到 s 中。该区间应该是 map
    类型的。
m.insert(make_pair(10, "Hello")); // 将元素插入到map中。需要<utility>
m.erase(e);       // 将键为 e 的元素删除。返回值为被删除的 e 的数量。
m.erase(pos);     // 将 pos 处的元素删除。
m.erase(begin, end); // 将 [begin, end) 处的元素删除。

// 4. 迭代器：begin、end、rbegin、rend 分别返回正向和反向迭代器。
```

3.8 二叉堆

3.8.1 基本二叉堆

二叉堆 二叉堆是完全二叉树。按照树根大小，二叉堆可分为最大值堆和最小值堆。

二叉堆的特点：

1. 最大（小）值堆中，结点一定不小（大）于两个儿子的值。

2. 在堆中，两兄弟的大小没有必然联系。
3. 最大 (小) 值堆的根结点是整个树中的最大 (小) 值。

实现 本节的二叉堆是最大值堆，修改代码中的标记部分可以变成最小值堆。

由于是完全二叉树，所以可以直接用一维数组保存。数组的下标是从 0 开始的。

二叉堆的操作有：

1. 插入：在堆中插入元素，首先要把元素放到末尾，然后通过不断往上“拱，把元素“拱”到正确的位置。
2. 用现有值初始化：最快的方法不是挨个插入，而是直接调整数组元素的顺序，使其符合堆的性质。
3. 查找：查找最值是最快的 ——直接访问树根就可以了。不过，用堆查找其他值就很慢了。因此，可以考虑再使用一个适合查找的辅助数据结构，例如二叉排序树。
4. 删除：把堆中最后一个元素 (就是一维数组存储所对应的最后一个元素) 放到待删除元素的位置，将元素总数减一，然后调整各元素的顺序。

```
// 注意：如果想改成最小值堆，只需调换有 (*) 标记的代码中的不等号的方向。

const int N=1000;
int heap[N], n;

inline int parent(int r) {return (r-1)/2;}
inline int leftChild(int r) {return r*2+1;}
inline int rightChild(int r) {return r*2+2;}
inline bool isLeaf(int r) {return r>=n/2;}
// inline void swap(int &a, int &b) {int t=a; a=b; b=t;}

void insert(int value)
{
    int curr = n++;
    heap[curr] = value;

    while (curr!=0 and heap[curr]>heap[parent(curr)]) // (x)
    {
        swap(heap[curr], heap[parent(curr)]);
        curr = parent(curr);
    }
}

void siftDown(int pos)  // 使元素往下“拱”。你不必手动调用此函数。
{
```

```
    while (not isLeaf(pos))
    {
        int i=leftChild(pos), j=rightChild(pos);
        if (j<n and heap[i]<heap[j]) i = j; // (*) 只改第二个不等号
        if (heap[i]<=heap[pos]) return; // (*)
        swap(heap[i], heap[pos]);
        pos = i;
    }
}

// 建堆。注意：在调用此函数之前，先初始化 heap[] 和 n
void buildHeap() { for (int i = n/2-1; i>=0; i--) siftDown(i); }

int removeMax()
{
    if (n==0) return 0;
    n--;
    swap(heap[0], heap[n]);
    if (n!=0) siftDown(0);
    return heap[n];
}

int removeItem(int pos)
{
    n--;
    swap(heap[pos], heap[n]);
    while (pos!=0 and heap[pos]>heap[parent(pos)]) // (*)
        swap(heap[pos], heap[parent(pos)]);
    siftDown(pos);
    return heap[n];
}

int getMax() { return heap[0]; }
```

3.8.2 左偏树

3.8.3 STL 中的堆算法

头文件：<algorithm>

为了实现堆算法，需要一个支持随机迭代器的容器。当然，一维数组也可以。

下面各函数的 comp 用于代替默认的小于号。如果不需要，可以省略。如果不指明，那么堆中第一个元素的值是*最大值*。区间为*左闭右开*区间。

- make_heap(begin, end, comp)：将某区间内的元素转化为堆。时间复杂度 $O(n)$。

- push_heap(begin, end, comp)：假设 [begin, end-1) 已经是一个堆。现在将 end 之前的那个元素加入堆中，使区间 [begin, end) 重新成为堆。时间复杂度 $O(\log n)$。
- pop_heap(begin, end, comp)：从区间 [begin, end) 取出第一个元素，放到最后位置，然后将区间 [begin, end-1) 重新组成堆。时间复杂度 $O(\log n)$。
- sort_heap(begin, end, comp)：将 heap 转换为一个有序集合。时间复杂度 $O(n \log n)$。

3.8.4 STL 中的优先队列

头文件：<queue>

priority_queue 基于 vector 实现[2]。普通的队列是先进先出，而优先队列是按照优先级出队，即无论入队顺序如何，出队的都是最大 (最小) 值。

priority_queue 位于 <queue>，而 greater 和 less 存在于 <functional>。

可以用以下几种方式定义优先队列 (假设 arr 是一个有 10 个元素的数组)：

```
// 元素为 int 类型，最大值先出列。
priority_queue<int> q1;
// 元素为 int 类型，最小值先出列。注意两个 > 之间有空格。
priority_queue< int,vector<int>,greater<int> > q2;
// 元素为 float 类型，最大值先出列，用现有数组初始化。
priority_queue<float> q3(arr, arr+10);
```

优先队列支持的操作有：push、top (不是 front)、pop、empty 和 size。

如果需要使用自己的结构体，你需要重载复制构造函数和“>”或“<”运算符。less 对应“<”，表示最大值先出列；greater 对应“>”，表示最小值先出列。

```
struct MyStruct
{
   int v;
   MyStruct(int i):v(i) {}

   bool operator < (const MyStruct & b) const {return v < b.v;}
};
priority_queue < MyStruct,vector<MyStruct>,less<MyStruct> > q;
```

3.8.5 线段树

3.8.5.1 一维线段树

问题 有一列长度为 n 的数组，刚开始全是 0。现在执行 m 次操作，每次可以执行以下两种操作中的一个：

[2] 模板接受三个参数，第二个就是容器类型。一般使用 vector，也可使用 deque。

1. 将某个区间的值全部加上指定的数。
2. 询问给定区间中所有数的和。

类似的问题 有一个区间，可以覆盖或删除一个线段。此问题可以转化为上面的问题。

代码 注意代码中所有区间均为左闭右开区间。

```
// 本代码维护的是区间和
// 注意:  请根据实际需要修改 modify、update、mergevalue 和 query 四个函数!
//   N 的取值应该为区间最大值的 3~4 倍。
const int N=1000;

struct segment_tree
{
    struct node
    {
        node *lch, *rch;
        int st, en;
        int value;
        int delta;        // 懒惰修改标记
        void modify(int d) // 懒惰修改
        {
            value+=d*(en-st);
            delta+=d;
        }
        void update()
        {
            if (lch and rch)
            {
                lch->delta+=delta;
                lch->value+=delta*(lch->en - lch->st);
                rch->delta+=delta;
                rch->value+=delta*(rch->en - rch->st);
            }
            delta=0;
        }
        void mergevalue()
        {
            value = lch->value + rch->value;
        }
    } ST[N];
    int node_top;

    node *head() { return ST; }
```

```
void build(node *root, int st, int en)
{
    root->st=st, root->en=en;
    root->value=root->delta=0;
    if (en-st>1)
    {
        int mid=(st+en)>>1;
        root->lch=&ST[++node_top]; root->rch=&ST[++node_top];
        build(root->lch, st, mid);
        build(root->rch, mid, en);
    }
}

segment_tree(int st, int en) { node_top=0; build(ST, st, en); }

void modify(node *root, int st, int en, int delta)
{
    if (st <= root->st and root->en <= en)
    {
        root->modify(delta);
    }
    else
    {
        if (root->delta != 0) root->update();

        int mid = (root->st + root->en)>>1;
        if (st<mid) modify(root->lch, st, en, delta);
        if (mid<en) modify(root->rch, st, en, delta);

        root->mergevalue();
    }
}

int query(node *root, int st, int en)
{
    if (st <= root->st and root->en <= en)
        return root->value;
    else
    {
        if (root->delta != 0) root->update();

        int value=0;
        int mid = (root->st + root->en)/2;
        if (st<mid) value+=query(root->lch, st, en);
        if (mid<en) value+=query(root->rch, st, en);
```

```
            return value;
        }
    }
};
```

如果使用线段树进行动态维护，还需要注意两个重要条件：

- 如果想进行修改操作，那么父亲的值必须可以由左右两个儿子算出。
- 如果想进行统计操作，那么整体的值必须可以由局部推出。

3.8.5.2 二维线段树

3.8.5.3 矩形并

3.8.6 树状数组

备注：不保证以下内容是正确的。

问题 有一个长度为 n 的数组，并且支持以下两种操作：

1. 给其中某一个数增加 delta。
2. 查询，求 $a[1]$ 至 $a[i]$ 的和。

树状数组 树状数组的基本思想：把前缀和划分成一系列的不相交的子集和。子集长度以 2^k 为单位，而且，如果对于下标为 p 的二进制表示中有 x 个“1”，则把它分解为 x 个子集的和。

注意，在本节代码中，数组下标从 1 开始。

3.8.6.1 一维树状数组

单次查询和修改的时间复杂度均为 $O(\log n)$。

```
const int N=1000;

struct BIT
{
    int sum[N], n;

    BIT(int len): n(len) { memset(sum,0,sizeof(sum)); }

    BIT(int *a, int len): n(len)
    {
        int *pre = new int[len+1]; pre[0]=0;
        for (int i=1; i<=len; i++) pre[i]=pre[i-1]+a[i];
```

```
        for (int i=1; i<=len; i++) sum[i]=pre[i]-pre[i-lowbit(i)];
        delete [] pre;
    }

    int lowbit(int x) { return x & -x; }

    void add(int i, int value) { for (; i<n; i+=lowbit(i)) sum[i]+=value; }
    int query(int i)
    {
        int r=0;
        for (; i>0; i-=lowbit(i)) r+=sum[i];
        return r;
    }
    int query(int x, int y) { return query(y)-query(x-1); }
};
```

3.8.6.2 二维树状数组

线段树不易扩展，而树状数组可以容易地扩展到高维。

在二维情况下，单次查询和修改的时间复杂度为 $O(\log m \cdot \log n)$。

```
const int N=1000;

struct BIT
{
    int sum[N][N], m, n; // m 行 n 列

    BIT(int row, int col): m(row), n(len) { memset(sum,0,sizeof(sum)); }

    int lowbit(int x) { return x & -x; }

    void add(int i, int j, int value)
    {
        for (; i<=row; i+=lowbit(i))
            for (int tj=j; tj<=col; tj+=lowbit(tj))
                sum[i][tj]+=value;
    }
    int query(int i, int j)
    {
        int r=0;
        for (; i>0; i-=lowbit(i))
            for (int tj=j; tj>0; tj-=lowbit(tj))
                r+=sum[i][tj];
        return r;
    }
```

```
};
```

3.9 后缀数组

3.10 最长 xx 子序列

3.10.1 最长非降子序列 (LIS)

问题 有一个序列 $a_1, a_2, a_3, \cdots, a_n$ 共 N 个元素。现在要求从序列找到一个长度最长、且前面一项不大于它后面任何一项的子序列，元素之间不必相邻。$N \leqslant 100000$。

思路 如果用动态规划求解，那么状态转移方程为：$f(i) = \max\{f(j)\} + 1(a_j > a_i \text{且} i > j)$。时间复杂度为 $O(n^2)$，很明显，时间不够用。

超时的原因是大部分时间都耗在了寻找 $\max\{f(j)\}$ 上面。

根据这一点，我们可以对算法进行改进。假设数组 $C[]$ 是"到目前为止找到的一个非降子序列"，并且使 $C[]$ 是有序的[3]，那么我们就可以利用二分查找来确定 $f[j]$。时间复杂度由 $O(n^2)$ 降到了 $O(n \log n)$。

```
// 注意:
//   本段代码为最长非降子序列
//   所有数组的下标都是从 1 开始的
//   序列会被记录到 ans[] 里，如果不需要，可以把代码 (*) 处删除
//   如果希望数字不重复，请修改代码中的 (1) 处
//    如果希望寻找下降或非升序列，请修改代码中的 (1)、(2)、(3) 处
const int N=1000;
const int INIT=0x3F3F3F3F;

int C[N], f[N];
int f_max, f_last;
int C_prev[N], C_id[N], ans[N]; // (*) 记录序列

int b_search(int *a, int left, int right, int val)
{
    while (left<right)
    {
        int mid=(left+right)/2;
        if (a[mid]<=val) left=mid+1; else right=mid;
        // (1) 最长上升序列: 把 <= 改成 <
```

[3] 尽管不断加入的元素会破坏这个序列，不过最长序列的最后一个数是不会被覆盖的。因此，这样做可以保证 $C[]$ 的长度与最长序列的长度相等。

```
        //    下降序列改成 > ，非升序列改成 >=
    }
    return left; // (2) 下降和非升序列：改成 right
}

int LIS(int *a, int n)
{
    memset(C, INIT, sizeof(C)); // (3) 下降和非升序列：把 INIT 改成 0
    C[0]=a[0]; f_max=f_last=0;

    for (int i=0; i<n; i++)
    {
        int pos=b_search(C, 0, i, a[i]);
        f[i]=pos+1; C[pos]=a[i];
        if (f[i]>f_max) f_max=f[i], f_last=i;

        // (*) 记录序列
        C_id[pos]=i;
        if (pos==0) C_prev[i]=-1; else C_prev[i]=C_id[pos-1];
    }

    // (*) 记录序列
    for (int p=f_last, i=f_max; i>0; p=C_prev[p], i--) ans[i]=a[p];

    return f_max;
}
```

3.10.2 最长公共子序列 (LCS)

问题 有两个序列 a 和 b。求一个最长的序列 p，使它既是 a 的子序列，又是 b 的子序列，元素之间不必相邻。(a, b 长度小于 1000)

思路 典型的动态规划问题。设 $f(i,j)$ 表示 a 的前 i 个元素、b 的前 j 个元素中最长公共子序列的长度。那么

$$f(i,j)=\begin{cases} f(i-1,j) \\ f(i,j-1) \\ f(i-1,j-1)+k(a[i]=b[j]?k=1:k=0) \end{cases}$$

```
// 注意：下标从 1 开始，m、n 分别表示 a[]、b[] 元素个数。
int f[1001][1001];
int LCS(int *a, int m, int *b, int n)
```

```
{
    memset(f,0,sizeof(f));
    for (int i=1; i<=m; i++)
        for (int j=1; j<=n; j++)
        {
            if (a[i]==b[j]) f[i][j]=f[i-1][j-1]+1;
            f[i][j] = max(f[i][j], max(f[i-1][j], f[i][j-1]));
        }
    return f[m][n];
}
```

3.10.3 最长公共上升子序列

问题 有两个序列 a 和 b。求一个最长的序列 p，使它既是 a 的子序列，又是 b 的子序列，而且，子序列的元素必须是递增的。元素之间不必相邻。(a,b 长度小于 1000)

思路 我们可以先保证它是"公共子序列"。但是，在扩展这个子序列的时候 (也就是上一节中 k=1 的时候) 要好好"审查"一下，不要相等就加 1，而是要看看当前位置的值比前面[4]哪个数大，比它大才能加 1。此时状态转移方程可以写成：

$$f(i,j)=\begin{cases} f(i-1,j) \\ f(i-1,k)+1(1\leqslant k<j, a[i]=b[j]\wedge b[j]>b[k]) \end{cases}$$

时间复杂度为 $O(mn^2)$。注意，根据以上条件，答案不一定是 $f(m,n)$。实际上，$f(m,1)$ 到 $f(m,n)$ 中的最大值才是答案。

优化 在 $a[i]>b[j]$ 时，我们可以记下 $f(i,j)$ 的最大值 max。因为 $b[j]<a[i]$，所以 $b[j']$ 一定可以接在以 $b[j]$ 为结尾的子序列之中。而且，一旦找到与 $a[i]$ 相等的 $b[j']$，那么 $f(i,j')$ 一定等于 $max+1$。

同时，由于我们的上面讨论的 $f(i,j)$ 中的 i 是指的 $a[]$ 的"前 i 个"数字。也就是说 $f(i,j)$ 一定包含了 $f(i-1,j)$ 的最优情况或者是比其更优的情况。于是，我们还可以把 f 数组压缩一维，每次仅仅记录 $f(j)$ 就可以了。

经过优化，最终的时间复杂度为 $O(mn)$。

代码 优化后的代码如下：

```
// 注意:
//   本段代码为上升序列。如果希望逆序，修改 (1)；如果希望有重复项，修改 (2)
```

[4] 因为还有一个"公共"的条件，所以找最长序列时可以只关注其中一个序列。

```
//   所有数组的下标都是从 1 开始的
//   序列会被记录到 ans[] 里，如果不需要，可以把代码 (*) 处删除

const int N=1000;

int f[N];
int prev[N], ans[N]; // (*) 记录序列

int LCIS(int *a, int na, int *b, int nb)
{
    memset(f,0,sizeof(f));
    memset(prev,0,sizeof(prev)); // (*) 记录序列

    for (int i=1;i<=na;i++)
    {
        for (int j=1,k=0;j<=nb;j++)
        {
            if (b[j]<a[i] and f[j]>f[k]) k=j; // (1) 逆序: b[j]>a[i]
            if (b[j]==a[i])
            {
                f[j]=(k>0)?(f[k]+1):1;
                prev[j]=k;
                // (2) 重复项: 此处加入 k=j;
            }
        }
    }

    int pos=0;
    for (int i=1; i<=nb; i++) if (f[i]>f[pos]) pos=i;

    // (*) 记录序列
    for (int q=pos, i=f[pos]; i>0; q=prev[q], i--) ans[i]=b[q];

    return f[pos];
}
```

3.11 最近公共祖先 (LCA)

问题 给出一棵有根树 T，对于任意两个结点 u, v，求一个离根最远的结点 x，使得 x 同时是 u 和 v 的祖先。

3.11.1 在线算法 (DFS+ST)

令 $L(u)$ 为 u 的深度。设 $L(u) \leqslant L(v)$，那么如果 u 是 v 的父亲，那么 $LCA(u,v)=u$，否则 $LCA(u,v)=LCA(u,father(v))$。这样就可以用 $O(n^2)$ 的时间进行预处理，然后用 $O(1)$ 的时间来解决 LCA 问题。

实际上还有更好的在线算法，即转化为 $\pm 1-RMQ$ 问题后再继续。

代码在哪儿?

3.11.2 离线算法 (Tarjan)

求 LCA 的 Tarjan 算法是一个经典的离线算法。

Tarjan 算法用到了并查集。LCA 问题可以用 $O(n+Q)$ 的时间来解决，其中 Q 为询问的次数。

Tarjan 算法基于深度优先搜索的框架。对于新搜索到的一个结点，首先创建由这个结点构成的集合，再对当前结点的每一个子树进行搜索，每搜索完一棵子树，则可确定子树内的 LCA 询问都已解决，其他的 LCA 询问的结果必然在这个子树之外。

这时把子树所形成的集合与当前结点的集合合并，并将当前结点设为这个集合的祖先。之后继续搜索下一棵子树，直到当前结点的所有子树搜索完。这时把当前结点也设为“已被检查过的”，同时可以处理有关当前结点的 LCA 询问，如果有一个从当前结点到结点 v 的询问，且 v 已被检查过，那么，由于进行的是深度优先搜索，所以当前结点与 v 的最近公共祖先一定还没有被检查，而这个最近公共祖先的包含 v 的子树一定已经搜索过了，因此这个最近公共祖先一定是 v 所在集合的祖先。

```
// G[][] 表示树的邻接矩阵（可改写成邻接表），结果保存在 LCA[][] 中
// 注意：求 LCA 之前先调用 memset(parent,-1,sizeof(parent));
const int N=1000;
int G[N][N], n, LCA[N][N];
int parent[N];

int find(int p) { return parent[p]==p ? p : find(parent[p]); }
void unin(int i, int j) { parent[find(i)] = find(j); }

void LCA(int u)
{
   parent[u]=u;

   for (int i=0; i<n; i++)
      if (G[u][i] and parent[i]==-1) { LCA(i); unin(i,u); }

   for (int i=0; i<n; i++)
```

```
        if (parent[i]!=-1) LCA[u][i]=LCA[i][u]=find(i);
}
```

3.11.3 LCA 转 RMQ

对有根树 T 进行 DFS，将遍历到的结点按照顺序记下，即可得到一个长度为 $2n-1$ 的序列 E。在产生 E 的同时，我们还需要记录结点所处深度，最终得到一个与 E 相对应的序列 B。用 $pos[u]$ 表示结点 u 在这个序列中第一次出现的位置。那么，$LCA(T,u,v)=RMQ(B,pos[u],pos[v])$。

将 LCA 问题转化为 RMQ 问题的复杂度是 $O(n)$ 的。由于此 RMQ 为特殊的 $\pm 1-RMQ$，所以最终的时间复杂度为 $O(n)-O(1)$。

3.12 区间最值问题 (RMQ)

RMQ (Range Minimum/Maximum Query) 问题是指：已知长度为 n 的数列 A，有若干个问题，每个问题都是求某个区间的最值。

由于问题很多，并且区间范围不确定，因此采用朴素算法是不可行的。

线段树是可行的。采用线段树，预处理的时间为 $O(n)$，查询的时间是 $O(\log n)$。并且，如果规定数列元素可以更改，那么基本上只能用线段树来处理。这种情况下，下面的算法不可用。

3.12.1 ST 算法

ST 的全称为 Sparse Table。ST 算法利用了动态规划，其基本思想是分块——把区间最值存储到一个块中。那么，一个长度为 2^k 的区间的最值就可以从两个相邻的、长度为 2^{k-1} 的块中取。

设 $f(i,level)$ 是起点为 i、长度为 2^{level} 的区间的最值，则状态转移方程为 (求最小值用 min，求最大值则用 max)：

$$f(i,level)=min\{f(i,level-1),f(i+2^{level-1},level-1)\}$$

边界条件：$level=0$ 时只有一个元素，所以 $f(i,0)$ 自然就等于下标为 i 的数。

询问区间 $[left,right]$ 的最值时，首先令 $k=\text{int}(log_2(right-left+1))$，则区间最小值为

$$\text{RMQ}(left,right)=min\{f(left,k),f(right-2^k+1,k)\}$$

该算法是在线算法，预处理的时间为 $O(n\log n)$，但回答一次询问的仅为 $O(1)$。

```
// 初始化: 数据在data[]中。首先调用initRMQ()进行预处理, 然后就可以用queryRMQ
    来进行查询。
// 注意:  下标从0开始;  下面代码求的是最大值;  头文件: algorithm、cmath
int data[N];
int f[N][20];
const int pow2[20]={1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,
                  16384,32768,65536,131072,262144,524288};

void initRMQ(int n)
{
    for (int i=0; i<n; i++) f[i][0]=data[i];

    int k = int(log(n)/log(2));
    for (int level=1; level<=k; level++)
        for (int i=0; i<n; i++)
            if (i + pow2[level-1]-1 < n)
                f[i][level] = max(f[i][level-1], f[i+pow2[level-1]][level-1]);
            else
                break;
}

int queryRMQ(int left, int right)
{
    int k = int(log(right-left+1)/log(2));
    return max(f[left][k], f[right-pow2[k]+1][k]);
}
```

3.12.2 ±1-RMQ 问题

如果序列相邻两个元素的差是 ±1，那么 RMQ 有一个更优的处理方法。处理完成后，预处理的复杂度为 $O(n)$，询问的复杂度为 $O(1)$。

算法的核心仍然在于分块。把数组划分为每部分为 $L = \log_2 n/2$ 的小块，则一共会产生 $m = 2n/\log_2 n$ 个组。将 m 个块的最小值组成序列 Blocks。用 ST 算法对 Blocks 进行预处理，则预处理的时间为 $O(m \log m) = O(2n/\log_2 n \log(2n/\log_2 n)) = O(n)$。

接下来我们要在 $O(1)$ 的时间内回答 Blocks 上的 RMQ 询问。首先，对于一般的询问 RMQ(i, j)，求出 i 所在的编号 x 和它在块中的下标 a，以及 j 所在的块编号 y 和它在块中的下标 b。

- 如果 $x = y$，那么执行块内 RMQ: In-RMQ(x, a, b)，表示第 x 块中下标 a 到 b 的最小值。

- 如果 $x \neq y$，那么就把区间 $[i, j]$ 分成三部分 ——块 x 中从 i 到块末的最小值 In-RMQ(x, a, L)、在块 y 中从块首到 j 的最小值 In-RMQ$(y, 1, b)$ 以及第 $x + 1$ 块到

$y-1$ 块的最小值 RMQ(A’$,x+1,y-1$)。所以，RMQ 询问的结果为 $\mathrm{RMQ}(A',x,y)=\min\{\text{In-RMQ}(x,a,L),\text{In-RMQ}(y,1,b),\mathrm{RMQ}(A',x+1,y-1)\}$

3.12.3 RMQ 转为 LCA

设序列中最小值为为 A_k，建立优先级为 A_k 的根结点 T_k。接下来将 $A[1..k-1]$ 递归建树，作为 T_k 的左子树；将 $A[k+1..N]$ 递归建树作为 T_k 的右子树。这样，$\mathrm{RMQ}(A,i,j)$ 就转化为 $\mathrm{LCA}(T,i,j)$ 了。

LCA 可以转化为 $\pm1-RMQ$。这样，一般的 RMQ 问题也就可以用 $O(n)-O(1)$ 的时间来处理了。

3.12.4 第 k 小值

如果把“区间最小值”改成“区间第 k 小值”(假设元素不重复)，那么只好用线段树来解决。

不过线段树需要经过一些处理，使得第 k 小值可以以一种递推的方式来求出。因此，我们可以考虑这样处理每一部分 ——假设已经知道可能的第 k 小值 x，并且统计有多少个数比 x 小。

在合并的时候，如果恰好有 $k-1$ 个数比 x 小, 那么它一定是第 k 小数。如果比 x 小的数太少，说明 x 需要增大；如果比 x 小的数太多，则 x 需要减小。这样，只需二分 x，每次判断 x 是否为正确答案即可。

预处理 在“统计有多少个数比 x 小”之前，先用归并排序将数组排序。设 $A[l,r]$ 表示树内区间 $[l,r]$ 所有元素对应的有序数组。

查询 二分答案 x，根据比 x 小和相等的元素个数决定 x 应增大还是减小。统计元素个数只需要把分解后的 $2\log_2 n$ 个区间的统计结果相加，而每个区间的统计需要一次有序数组中的二分查找。假设所有元素均为正整数，且不超过 C，则二分次数不超过 $\log C$ 次，总时间复杂度为 $O(\log C\log_2 n)$[5]。

修改 在一个有序数组里修改元素可能会很慢，所以为了让修改操作变得快速，需要把有序数组改为排序二叉树。这样，我们得到了一个以排序二叉树为结点的线段树。修改时，从元素对应的线段树的叶结点开始，一直到树根，把每个结点所拥有的排序二叉树的元素更新一下。(删除旧的，增加新的)。时间复杂度为 $\log n+\log(n/2)+\log(n/4)+\cdots=O(\log^2 n)$。

[5]实际的次数比它小得多，因为二分次数、树中区间个数和树中区间长度都可以远比估计值小。

3.13 逆序对

对于一个序列 $a_1, a_2, a_3, \cdots, a_n$，如果存在 $i < j$，使 $a_i > a_j$，那么 (a_i, a_j) 就是一个逆序对。现在给你一个有 N 个元素的序列，请求出序列内逆序对的个数。($N \leqslant 100000$ [6])。

3.13.1 二分法

可以考虑将序列分成两部分，分别求出子序列的逆序对个数。

横跨两序列的逆序对个数怎么统计？用枚举法行吗？不行，那样做就失去分治的意义了。

但是如果两区间有序，那么统计就很容易了，因此我们边统计边排序。由于子序列内逆序对个数的统计是在排序之前完成的，排序又不影响两序列之间的逆序对个数，所以排序是不会影响最终结果的。

```
// 注意：需要左闭右开区间！
int c[100002];
int mergeSort(int *a, int l, int r)
{
    int mid, i, j, tmp;
    int cnt = 0;
    if (r>l+1)
    {
        mid = (l+r)/2;
        cnt += mergeSort(l, mid);
        cnt += mergeSort(mid, r);
        tmp = l;
        for (i=l,j=mid; i<mid && j<r; )
        {
            if (a[i]>a[j])
            {
                c[tmp++] = a[j++];
                // 使用排序,就可以方便地数跨“分界”的逆序对个数了
                cnt += mid-i;
            }
            else
            {
                c[tmp++] = a[i++];
            }
        }
        for (; j<r; j++) c[tmp++] = a[j];
        for (; i<mid; i++) c[tmp++] = a[i];
        for (i=l; i<r; i++) a[i]=c[i];
    }
```

[6] 由于最大的逆序数为 $N(N-1)$，所以，如果 N 再大一点，int 就装不下了。

```
    return cnt;
}
```

3.13.2 树状数组

备注：不保证以下内容是正确的。

由于我们只是看两个数之间的大小关系，所以可以对序列中的数进行离散化。即按照大小关系把 a_1 到 a_n 映射到 1 至 num 之间 (num 为不同数字的个数)，保证仍然满足原有的大小关系。

这样，本题就转化成了：对于一个数 a_i，在它后面有多少个比它小的数?

处理的时候，我们从第 n 个数倒着处理，用树状数组维护一个 $cnt[]$ 数组，其前缀和 Query(x) 表示"到当前处理的第 i 个数为止，映射后值为 1 到 x 之间的数字一共有多少个"，换句话说，如果 x 对应的是 $a[i]$，那么比 $a[i]$ 小的数的个数就是 Query($x-1$)。

对于维护，只要在该次查询结束后进行修改即可，即 Change(x,1)。整个算法的时间复杂度为 $O(n\log n)$。

```
// ****************************
// 备注：代码是抄袭的，需要消化和整理
// ****************************

#include <cstdio>
#include <cstring>
#include <algorithm>
const int MAXN=100000;
using namespace std;

struct node
{
    long long v;
    int id;
    bool operator< (const node &p) const {return v<p.v;}
};

node a[MAXN+10];
long long c[MAXN+10];
long long b[MAXN+10];
int n;

inline int lowbit(int x) { return x bitand -x; }
long long Query(int x)
{
    long long ans=0;
    while (x)
```

```
    {
        ans+=c[x];
        x-=lowbit(x);
    }
    return ans;
}
void Change(int x)
{
    while (x<=n)
    {
        c[x]++;
        x+=lowbit(x);
    }
}

int main()
{
    memset(a,0,sizeof(a));
    memset(b,0,sizeof(b));
    memset(c,0,sizeof(c));
    cin>>n;
    for (int i=1; i<=n; i++) { cin>>a[i].v; a[i].id=i; }
    sort(a+1, a+1+n);

    int pre=-1;
    int prevalue=0;
    for (int i=1; i<=n; i++)
    {
        if (pre!=a[i].v)
        {
            pre=a[i].v;
            a[i].v=++prevalue;
        }
        else
            a[i].v=prevalue;
    }

    for (int i=1; i<=n; i++)
        b[a[i].id]=a[i].v;

    long long s=0;
    for (int i=n; i>=1; i--)
    {
        Change(b[i]);
        s+=Query(b[i]-1);
    }
```

```
    cout<<s<<endl;
    return 0;
}
```

3.13.3 逆序对数推排列数

问题 已知一个序列有 n 个不重复元素，而且该序列的逆序对数为 m。符合条件的序列有多少种?

第四章　动态规划

4.1　基本动态规划

4.1.1　区间问题

4.1.2　环形问题

4.1.3　判定性问题

4.1.4　棋盘分割

4.1.5　最长公共子序列 (动态规划)

4.1.6　最长上升子序列 (动态规划)

4.2　背包问题

4.2.1　部分背包

4.2.2　01 背包

4.2.2.1　二维数组表示

4.2.2.2　一维数组表示

4.2.3　完全背包

4.2.4　多重背包

4.2.5　二维费用背包

4.2.6　分组背包

4.2.7　混合背包

4.2.8　泛化物品背包

4.3　概率和期望

4.4　二分判定型问题

4.5　树型动态规划

第五章　图论

5.1　图论的基本概念

5.2　图的遍历

根据问题的需要，可以采用深度优先搜索和广度优先搜索。处理结点时要注意对结点进行标记，以免重复访问或陷入环路。

将广度优先搜索中的队列改成堆栈，广度优先搜索将转变为深度优先搜索。不过要注意，最先递归调用的参数应该最后入栈。

此处应该有代码

5.3　拓扑排序

问题　有 n 项工作，其中某些工作必须在另外一些工作进行之前完成，请安排一个顺序，使得这些工作能够顺利完成。

例：有四项工作 1、2、3、4，其中 3 必须在 2 的前面，4 必须在 3 的前面，那么 1、4、3、2 是一种合法的安排。

拓扑排序　很明显，如果图中存在有向环，则排序无法完成，否则就可以得到一个序列。

5.3.1　DFS

思路：对于每个顶点，都先搜索并输出它的前趋。

```
// 初始化：G 表示邻接矩阵，n 表示结点个数，下标从 0 开始
//        从 i 到 j 如果连通，则 G[i][j]=false，否则 G[i][j]=true
// 调用：toposort()，如果有环则返回 false。排序结果保存在数组 a 中。

const int MAXN=1000;
```

```
bool G[MAXN][MAXN]; // 邻接矩阵
int n;              // 结点个数
int a[MAXN], a_n;

// 结点访问情况：0、1、2分别代表未访问、正在访问、已访问。
//             如果正在访问的点又被访问一次，说明有环。
int vis[MAXN];

bool DFS(int v)
{
    vis[v]=1;
    for (int i=0; i<n; i++)
        if (G[i][v])
        {
            if (vis[i]==1) return false;
            else if (vis[i]==0) if (not DFS(i)) return false;
        }
    vis[v]=2; a[a_n++]=v;
    return true;
}

bool toposort()
{
    memset(vis, 0, sizeof(vis));
    a_n=0;
    for (int i=0;i<n;i++)
        if ((not vis[i]) and (not DFS(i))) return false;
    return true;
}
```

5.3.2 辅助队列

操作步骤如下：

1. 统计每个结点的入度。
2. 将入度为零的点加入队列。
3. 依次对入度为零的点进行删边操作，同时将新得到的入度为零的点加入队列。
4. 继续对队列中未进行操作的点进行操作。
5. 如果排序结束，但仍然存在入度不为 0 的点，说明图中有环。

```
// 初始化： G 表示邻接矩阵，n 表示结点个数，下标从 0 开始
//       从 i 到 j 如果连通，则 G[i][j]=false ，否则 G[i][j]=true
```

```
// 调用： toposort()，如果有环则返回 false 。排序结果保存在数组 a 中。

const int MAXN=1000;

bool G[MAXN][MAXN];  // 邻接矩阵
int n;               // 结点个数
int cnt[MAXN];       // cnt[i] 表示结点 i 的入度
int a[MAXN], a_n;    // 结果保存在数组 a 中
queue<int> q;

bool toposort()
{
    memset(cnt,0,sizeof(cnt));
    a_n=0; q=queue<int>();

    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            if (G[i][j]) cnt[j]++;

    for (int i=0;i<n;i++) if (cnt[i]==0) q.push(i);

    while (not q.empty())
    {
        int i=q.front(); q.pop();
        a[a_n++]=i;
        for (int j=0;j<n;j++)
        {
            if (G[i][j])
            {
                cnt[j]--;
                if (cnt[j]==0) q.push(j);
            }
        }
    }

    for (int i=0;i<n;i++) if (cnt[i]!=0) return false;
    return true;
}
```

5.3.3 拓扑排序个数

已知有 n $(n < 17)$ 个元素，其中某些元素必须在另外一些元素的前面。求有多少种排列方法。

此题是 NPC 问题，没有多项式时间的解法。需要用动态规划的思想来进行搜索，而且需要

进行状态压缩。

设 $f(G)$ 表示图 G 的拓扑排序数。

以图上某一个点 p_0 为例，如果该点入度为 0，那么就会有 $f(\{G-p_0\})$ 种方案 ($\{G-p_0\}$ 指去掉 p_0 之后的子图)，使得该点处于拓扑排序序列的第一位上面。所以需要枚举图上的每一个点 p，求出 $f(G-p)$ 的总和，这个总和即位拓扑排序数。

边界条件：如果子图 S 只有一个点，那么 $f(S)=1$。

```
// 初始化：G表示邻接矩阵，n表示结点个数，下标从0开始
//        从i到j如果连通，则G[i][j]=false，否则G[i][j]=true
// 调用：topocount()
const int MAXN=20, M=100001;

bool G[MAXN][MAXN];
int n;
int f[M];
bool err=false;

int encode(int *vis)    // 状态压缩
{
    int c=0;
    for(int i=0;i<n;i++) if(vis[i]==0) c|=1<<i;
    return c;
}

int DFS(int *vis)
{
    int p=encode(vis);
    if (f[p]) return f[p];

    int num=0, m=0, tmp[MAXN];
    for (int i=0; i<n; i++)
    {
        if (not vis[i])
        {
            num++;      // 用于判断目前的图是否只有一个点
            int pre=0;
            for (int j=0; j<n; j++)
                if ((not vis[j]) and G[j][i]) pre++;
            // 存储入度为0的点，准备到后面删除。
            if (!pre) tmp[m++]=i;
        }
    }

    if (not m) { err=true; return 0; }
```

```
    if (num==1) return f[p]=1;
    else
    {
        int sum=0;
        for (int i=0; i<m; i++)
        {
            vis[tmp[i]]=1;
            sum+=DFS(vis);
            vis[tmp[i]]=0;
            if (err) return 0;
        }
        return f[p]=sum;
    }
}

int topocount()
{
    int vis[MAXN]={0};
    memset(f,0,sizeof(f));
    err=false;
    DFS(vis);
    return f[encode(vis)];
}
```

5.4 欧拉路径

找欧拉路径，直观地讲，就是解决“一笔画”问题。

5.4.1 无向图的欧拉路径

两个定义：在一个无向图中，一条包含所有边，且其中每一条边只经过一次的路径叫做欧拉通路。若这条路径的起点与终点为同一点，则为欧拉回路。

判定一个图是否存在欧拉通路或欧拉回路的根据如下：

1. 一个图有欧拉回路当且仅当它是连通的（即不包括 0 度的结点）且每个结点都有偶数度。
2. 一个图有欧拉通路当且仅当它是连通的且除两个结点外，其他结点都有偶数度。在此条件下，含奇数度的两个结点一定是欧拉通路的起点和终点。

```
// 初始化：G表示邻接矩阵，n表示结点个数，下标从0开始
//        从i到j如果连通，则G[i][j]=false，否则G[i][j]=true
// 调用：find_circuit()，如果有欧拉通路则输出并返回true
```

```
// 注意: 此函数会破坏G的值, 请事先做好备份。
const int MAXN=1000;

bool G[MAXN][MAXN]; // 邻接矩阵
int n;              // 结点个数

// 无向图的邻接矩阵。如果两点连通,则为1,否则为0
int cnt[MAXN];
int circuit[MAXN], pos;

void search(int i)
{
    circuit[pos++]=i;
    for (int j=0; j<n; j++)
        if (G[i][j])
        {
            G[i][j]=G[j][i]=false;
            search(j);
        }
}

bool find_circuit()
{
    int start=0, oddnumber=0;
    memset(cnt,0,sizeof(cnt));

    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            if (G[i][j]) ++cnt[i];    // 统计结点入度

    for (int i=0; i<n; i++)
        if (cnt[i]%2==1) start=i, ++oddnumber;

    if (oddnumber>2 or oddnumber==1) return false;
    else
    {
        pos=0;
        search(start);

        for (int i=0; i<pos; i++) cout<<circuit[i]<<"--->";

        return true;
    }
}
```

5.4.2 有向图的欧拉路径

对于有向图来说，我们可以把“奇点”和“偶点”的定义稍作修改，认为出度和入度相等的点为“偶点”，出度和入度之差为 1 的点为“奇点”。

很明显，如果某个点的出度和入度之差大于 1，那么显然不可能做到“一笔画”。

当然，上面代码中的“G[i][j]=G[j][i]=false;”一句也得跟着改成单向的“G[i][j]=false;”。

5.5 生成树 (森林)

5.5.1 最小生成树

问题 要在 n 个城市之间铺设光缆，要求这 n 个城市的任意两个之间都可以通信，然而，铺设光缆的费用很高，且各个城市之间铺设光缆的费用不同，所以还需要使铺设光缆的总费用最低。请问最低费用是多少?

最小生成树 对于一个无向图来说，最小生成树是一个这样的子图：它内部没有环，各节点之间可以互相抵达，而且把最小生成树上各边的权值相加之后，得到的和是最小的。

5.5.1.1 最小生成树 (Prim)

Prim 算法是贪心算法，贪心策略为：找到目前情况下能连上的权值最小的边的另一端点，加入之，直到所有的顶点加入完毕。

Prim 适用于稠密图。

朴素 Prim 的时间复杂度是 $O(n^2)$。因为在寻找离生成树最近的未加入顶点时浪费了很多时间，所以可以用堆进行优化。堆优化后的 Prim 算法的时间复杂度为 $O(m \log n)$。

不过堆优化 Prim 的代码比较复杂，而并查集优化的 Kruskal 算法与它相比要好很多。

```
// 初始化：G表示邻接矩阵，n表示结点个数，下标从0开始
//        如果边(i,j)不存在，则G[i][j] = INF
// 调用：Prim(start)，其中start是图上某个点的编号，可以任取。
// 注意：  start的出度不能为0!
//         无向图!
//         调用函数之后可以追踪cloest以获得生成树的具体连接情况
const int MAXN=1000, INF=0x3F3F3F3F;
int G[MAXN][MAXN], n;

int minEdge[MAXN];   // 与点N连接的最小边
int cloest[MAXN];    // 追踪

int Prim(int start=0)
```

```
{
    int ans=0, k=0;
    // 加入第一个点
    for (int i=0;i<n;i++)
    {
        minEdge[i]=G[start][i];
        cloest[i]=start;
    }
    minEdge[start]=0;
    for (int i=0;i<n-1;i++)
    {
        int min=INF;
        // 寻找离生成树最近的未加入顶点k
        for (int j=0;j<n;j++)
            if (minEdge[j]!=0 and minEdge[j]<min) min=minEdge[k=j];
        // 把找到的边加入到MST中
        ans+=minEdge[k];

        minEdge[k]=0;
        // 加入完毕。以后不用再处理这个点。
        // 重新计算最短边
        for (int j=0;j<n;j++)
            if (G[k][j]<minEdge[j])
            {
                minEdge[j]=G[k][j];
                cloest[j]=k;
            }
    }
    return ans;
}
```

5.5.1.2 最小生成树 (Kruskal)

Kruskal 算法是贪心算法，贪心策略为：选目前情况下能连上的权值最小的边，若与已经生成的树不会形成环，加入之，直到 $n-1$ 条边加入完毕。

时间复杂度为 $O(n\log m)$，最差情况为 $O(m\log n)$。相比于 Prim，这个算法更常用。

```
// 初始化：G表示邻接矩阵，n表示结点个数，下标从0开始
//        从i到j如果连通，则G[i][j]=false，否则G[i][j]=true
// 调用：find_circuit()，如果有欧拉通路则输出并返回true
// 注意：此函数会破坏G的值，请事先做好备份。
const int MAXN=1000;

bool G[MAXN][MAXN]; // 邻接矩阵
```

```
int n;              // 结点个数

// 无向图的邻接矩阵。如果两点连通,则为1,否则为0
int cnt[MAXN];
int circuit[MAXN], pos;

void search(int i)
{
   circuit[pos++]=i;
   for (int j=0; j<n; j++)
      if (G[i][j])
      {
         G[i][j]=G[j][i]=false;
         search(j);
      }
}

bool find_circuit()
{
   int start=0, oddnumber=0;
   memset(cnt,0,sizeof(cnt));

   for (int i=0; i<n; i++)
      for (int j=0; j<n; j++)
         if (G[i][j]) ++cnt[i];    // 统计结点入度

   for (int i=0; i<n; i++)
      if (cnt[i]%2==1) start=i, ++oddnumber;

   if (oddnumber>2 or oddnumber==1) return false;
   else
   {
      pos=0;
      search(start);

      for (int i=0; i<pos; i++) cout<<circuit[i]<<"--->";

      return true;
   }
}
```

5.5.2 次小生成树

次小生成树可以由最小生成树换一条边得到。这样就有一种比较容易想到的算法 ——枚举删除最小生成树上的边，再求最小生成树，然而算法的效率并不是很高。

有一种更简单的方法：先求最小生成树 T，枚举添加不在 T 中的边，则添加后一定会形成环。找到环上边值第二大的边 (即环中属于 T 中的最大边)，把它删掉，计算当前生成树的权值，取所有枚举修改的生成树的最小值，即为次小生成树。

这种方法在实现时有更简单的方法：首先求最小生成树 T，然后从每个结点 u 遍历最小生成树 T，用一个二维数组 $max[u][v]$ 记录结点 u 到结点 v 的路径上边的最大值 (即最大边的值)。然后枚举不在 T 中的边 (u,v)，计算 $T-max[u][v]+w(u,v)$ 的最小值，即为次小生成树的权值。显然，这种方法的时间复杂度为 $O(N^2+E)$。

没有代码……

5.5.3 最小生成森林

除了连通性的问题以外，实际上和最小生成树是一样的。

5.5.4 最小树形图

5.6 最短路

5.6.1 如何选用

Floyd 是多源最短路算法，而另外几种是单源最短路算法。Floyd 可以一次性计算任意两个点之间的最短路。

下面按照稀疏图和稠密图来进行分类：

稠密图 邻接矩阵比邻接表合适。

- 如果 n 不大，可以用邻接矩阵的话，Dijkstra 是最好的选择，SPFA 也可以。
- n>300 时最好不要用 Floyd 算法。
- n>200 时尽量不要用 Bellman-Ford 算法。

稀疏图 邻接矩阵会造成一定的浪费。可能邻接表更合适。

- 超过 3000 个点就不要用朴素的 Dijkstra 算法了。
- 边数超过 50000 时尽量不要用 Bellman-Ford 算法。

- 超过 500000 条边就不要用 SPFA 加邻接表了。
- 点和边数很多的时候，优先队列优化的 Dijkstra 是更合适的。不过，由于优先队列自身的原因，速度会比堆慢一些。

最短路算法的时间复杂度 见表 5.6.1

算法	图的表示法	时间复杂度
Dijkstra	邻接矩阵（邻接表也可）	$O(n^2)$
优先队列优化的 Dijkstra	邻接表	一般 $O[(m+n)\log m]$，密集图 $O(n^2\log m)$
Bellman-Ford	边目录	$O(mn)$
优化的 Bellman（SPFA）	邻接表	$O(km)$，其中 k 可认为是常数。
Floyd	邻接矩阵	$O(n^3)$

表 5.1: 最短路算法的时间复杂度比较

5.6.2 Dijkstra 算法 (邻接矩阵)

注意！Dijkstra 算法只适用于所有边的权都大于 *0* 的图。

Dijkstra 算法是贪心算法。基本思想是：

设置一个顶点的集合 S，并不断地扩充这个集合，当且仅当从源点到某个点的路径已求出时它才属于集合 S。

开始时 S 中仅有源点, 调整 S 集合之外的点的最短路径长度, 并从中找到当前最短路径点, 将其加入到集合 S, 直到所有的点都在 S 中。

普通算法的时间复杂度为 $P(n^2)$。不过还有优化的余地。

```
// 初始化: G表示邻接矩阵，n表示结点个数，下标从0开始
//        如果边(i,j)不存在，则G[i][j] = INF
// 调用: Dijkstra(start)，其中start是起点。结果保存在d[]中。
// 注意:   不能有负边!
//         调用函数之后可以追踪prev以获得具体的路径
const int MAXN=2000, INF=0x3F3F3F3F;
int G[MAXN][MAXN], n;

bool visited[MAXN]; // 是否被标号
int d[MAXN];        // 从起点到某点的最短路径长度
int prev[MAXN];
```

```
void Dijkstra(int start)
{
    memset(visited, 0, sizeof(visited));
    memset(d,INF,sizeof(d)); // 注意memset的局限性!
    d[start]=0;

    for (int i=0; i<n; i++)
    {
        int x, min=INF;
        // 在所有未标号的结点中,选择一个d值最小的点x。
        for (int a=0; a<n; a++) if (!visited[a] && d[a]<min) min=d[x=a];

        // 标记这个点x。
        visited[x]=true;
        // 对于从x出发的所有边(x,y),更新一下d[y]。
        for (int y=0; y<n; y++)
            if (d[y] > d[x]+G[x][y])
            {
                // y这个最短路径是从x走到y。
                d[y] = d[x]+G[x][y];
                prev[y] = x;
            }
    }
}
```

5.6.3 优先队列优化的 Dijkstra 算法 (邻接表)

朴素的 Dijkstra 算法在“选 d 值最小的点”时要浪费很多时间，所以可以用优先队列来优化。优化之后的时间复杂度为 $O[(n+m)\log m]$，最差情况 (密集图) 则为 $O(n^2\log m)$。

由于优先队列基于 vector，本身存在一定的局限性 (例如内存分配可能会占用一些时间)，所以也可以采用堆来优化。

```
// 初始化: 先调用init, 然后用addedge来添加所有边。
// 调用: Dijkstra(start), 其中start是起点。结果保存在d[]中。
// 注意:  不能有负边!
//         调用函数之后可以追踪prev以获得具体的路径
//         #include <queue>、<vector>、<utility>
const int MAXN=1000, MAXM=100000, INF=0x3F3F3F3F;

struct edge
{
    int u,v,w;
    edge *next;
} mem[MAXM], *adj[MAXN];
```

```
int m=0;
void init() { m=0; memset(adj,0,sizeof(adj)); }

void addedge(int u, int v, int w)
{
    edge *p = &mem[m++];
    p->u=u, p->v=v, p->w=w;
    p->next=adj[p->u];
    adj[p->u]=p;
}

typedef pair<int, int> pii;
priority_queue < pii, vector<pii>, greater<pii> > q;
int d[MAXN], prev[MAXN];

void Dijkstra(int start)
{
    memset(d,INF,sizeof(d)); // 注意memset的局限性!
    d[start]=0;
    q.push(make_pair(d[start], start));

    while (!q.empty())
    {
        // 在所有未标号的结点中,选择一个d值最小的点x。
        pii u=q.top(); q.pop();
        int x=u.second;
        // 已经计算完
        if (u.first!=d[x]) continue;

        for (edge *e=adj[x]; e!=NULL; e=e->next)
        {
            int &v=e->v, &w=e->w;
            if (d[v] > d[x]+ w)
            {
                // 松弛
                d[v] = d[x]+ w;
                prev[v]=x;
                q.push(make_pair(d[v], v));
            }
        }
    }
}
```

5.6.4 Bellman-Ford 算法

Bellman-Ford 算法是迭代法，它不停地调整图中的顶点值 (源点到该点的最短路径值)，直到没有可调整的值。

该算法除了能计算最短路，还可以检查负环 (一个每条边的权都小于 0 的环) 把每条边的权取一个相反数，就可以判断图中是否有正环了。。如果图中有负环，那么这个图不存在最短路。

时间复杂度为 $O(mn)$。

对于下面代码，如果纯粹是为了检查负环，可以把代码中的 start、d[start]=0 和 if (d[x]<INF) 去掉。

```
// 初始化：令m=0，然后使用addedge添加所有边（或直接初始化m,u,v,w）。
// 调用：Ford(start)。start是起点。如果有负环则返回false，否则返回true。
// 注意： 下标从0开始； 结果保存在d[]中
const int MAXN=200, MAXM=10000, INF=0x3F3F3F3F;
int u[MAXM],v[MAXM],w[MAXM];
int m=0, n;

void addedge(int x, int y, int len)
{
    u[m]=x, v[m]=y, w[m]=len;
    m++;
}

int d[MAXN];

bool Ford(int start)
{
    memset(d,INF,sizeof(d));
    d[start]=0;

    for (int k=0;k<n-1;k++)
        for (int i=0; i<m; i++) // 检查每条边
        {
            int &x=u[i], &y=v[i];
            if (d[x]<INF) d[y]=min(d[y],d[x]+w[i]);
        }

    // 检查负环——如果全部松弛之后还能松弛,说明一定有负环
    for (int i=0; i<m; i++)
    {
        int &x=u[i], &y=v[i];
        if (d[y]>d[x]+w[i]) return false;
    }
    return true;
```

```
}
```

5.6.5 SPFA 算法

Bellman-Ford 在迭代时会有一些冗余的松弛操作。SPFA 则通过队列对其进行了一个优化。时间复杂度为 $O(km)$，其中 k 为所有顶点进队的平均次数，一般情况下 $k \leqslant 2$。

```
// 初始化：先调用 init，然后用addedge来添加所有边。
// 调用：SPFA(start)。start是起点。如果有负环则返回false，否则返回true。
// 注意： 下标从0开始； 结果保存在 d[] 中。
const int MAXN=1000, MAXM=100000, INF=0x3F3F3F3F;

struct edge
{
    int u,v,w;
    edge *next;
} mem[MAXM], *adj[MAXN];

int m=0;
void init() { m=0; memset(adj,0,sizeof(adj)); }

void addedge(int u, int v, int w)
{
    edge *p = &mem[m++];
    p->u=u, p->v=v, p->w=w;
    p->next=adj[p->u];
    adj[p->u]=p;
}

queue<int> q;
bool inqueue[MAXN];
int cnt[MAXN];  // 结点进队次数。如果超过n说明有负环。
int d[MAXN];

bool SPFA(int start)
{
    memset(d,INF,sizeof(d)); // 注意memset的局限性!
    memset(cnt,0,sizeof(cnt));

    d[start]=0;
    q=queue<int>();
    q.push(start);   // 源点入队
    cnt[start]++;
    while (!q.empty())
    {
```

```
        int x=q.front(); q.pop();
        inqueue[x]=false;

        // 对队首点出发的所有边进行松弛操作(即更新最小值)
        for (edge *e=adj[x];e!=NULL;e=e->next)
        {
            int &v=e->v, &w=e->w;
            if (d[v]>d[x]+w)
            {
                d[v] = d[x]+w;
                // 将不在队列中的尾结点入队
                if (!inqueue[v])
                {
                    q.push(v); inqueue[v]=true;
                    // 有负环
                    if (++cnt[v]>n) return false;
                }
            }
        }
    }
    return true;
}
```

5.6.6 Floyd 算法

5.6.6.1 多源最短路

Floyd 算法是动态规划算法。设 $f(i,j)$ 表示从 i 到 j 的最短路径长度，则 $f(i,j) = \min\{f(i,k)+f(k,j)\}$ (其中 i 到 k、k 到 j 都是连通的)。

时间复杂度为 $O(n^3)$。适用于反复查询任意两点的最短距离。

```
// 初始化：G表示邻接矩阵，n表示结点个数，下标从0开始
//        如果边(i,j)不存在，则G[i][j] = INF
// 调用：Floyd()。结果保存在f[][]中。
// 注意：调用函数之后可以追踪prev以获得具体的路径
const int MAXN=300, INF=0x3F3F3F3F;
int G[MAXN][MAXN], n;

int f[MAXN][MAXN], prev[MAXN][MAXN];

void Floyd()
{
    memcpy(f,G,sizeof(G));
    memset(prev, -1, sizeof(prev));
```

```
    for (int k=0; k<n; k++)
        for (int i=0; i<n; i++)
            for (int j=0; j<n; j++)
                if (f[i][k] + f[k][j] < f[i][j])
                {
                    f[i][j] = f[i][k] + f[k][j];
                    prev[i][j] = k;
                }
}
```

5.6.6.2 最小环

5.6.7 第 K 短路

5.7 二分图

5.7.1 是否为二分图

5.7.2 最大匹配

5.7.2.1 匈牙利算法 (DFS)

5.7.2.2 匈牙利算法 (BFS)

5.7.2.3 Hopcroft-Carp 算法

5.7.3 最大权匹配

5.7.4 最小点集覆盖

5.7.5 最大独立集

5.7.6 最佳匹配

5.8 网络流

5.8.1 最大流

5.8.1.1 增广路算法 (Edmonds-Karp 算法)

5.8.1.2 Dinic 算法

5.8.1.3 SAP 算法

5.8.1.4 有下界的最大流

5.8.1.5 多源多汇最大流

5.8.2 最小费用流

5.8.2.1 最小费用流

5.8.2.2 最小费用最大流

5.8.3 最小费用最大流

5.8.3.1 最小费用最大流

5.8.3.2 最大费用最大流

5.9 差分约束系统

5.10 连通性

5.10.1 双连通分量

第六章　数学

6.1　组合数学

6.1.1　排列组合的计算

6.1.2　排列组合的生成

6.1.2.1　类循环排列

示例：0000、0001、0002、0010、0011、0012、0020、0021、0022、0100、0101……2222

在数字位数和可用数字个数不确定的情况下，显然无法用多重循环 ——用递归就可以了。

注意，某些问题也可以用位运算来解决。例如只有 0 和 1 的时候，或者数字是 0、1、2、3 的时候。

6.1.2.2　全排列 (无重复)

示例：123、132、213、231、312、321

与上节类似，但是在递归调用之前要把已经填上的数字做好标记，防止重复使用。

```
// full_permutation有三个参数，
//   第一个参数是待排列的字符，
//   第二个参数是字符个数，
//   在调用的时候，第三个参数应该为0

const int N=20;
char result[N];       // 储存结果
bool used[N];         // 表示是否已经被使用

void full_permutation(char *item, int n, int depth=0)
{
   if (depth==n)
   {
      // 输出结果
      for (int i=0; i<n; i++) cout<<result[i];
```

```
        return;
    }

    for (int i=0; i<n; i++)
        if (not used[i])
        {
            used[i]=true;
            result[depth]=item[i];
            full_permutation(item,n,depth+1);
            used[i]=false;
        }
}
```

6.1.2.3 全排列 (有重复)

假设待排列的四个数为 1、1、2、3，则输出序列应该为：1123、1132、1213、1231、1312、1321、2113、2131……3211

实际上，重复排列的产生是由于同一个位置被多次填入了相同的数，并且这多次填入又是在同一次搜索过程中完成的。所以，需要统计每个本质不同的数的个数 (输入时注意剔除重复数值)。在搜索的某个过程中，如果某个数 p 还有剩余，那么就可以再次往里填，否则就不能再填了。

```
// 调用：unrepeat_permutation
//    第一个参数是待排列的字符，
//    第二个参数是字符个数
const int N=20;

void search(char *newitem, int new_n, int *count,
            int depth, int n, char *result)
{
    if (depth==n)
    {
        for (i=0; i<n; i++) cout<<result[i];
        return;
    }
    for (int i=0; i<new_n; i++)
        if (count[i] > 0)
        {
            used[i]--;
            result[depth] = newitem[i];
            search(newitem, new_n, cnt, depth+1, n, result);
            used[i]++;
        }
}
```

```
void unrepeat_permutation(char *item, int n)
{
    // 预处理，统计本质不同的数的个数
    char uniqueitem[N];
    int count[N]={0};
    int m=0;

    for (int i=0; i<n; i++)
    {
        for (int j=0; j<m; j++)
            if (item[i]==uniqueitem[j]) { count[j]++; break; }

        if (j==m)
        {
            uniqueitem[m]=val, count[m]=1;
            m++;
        }
    }

    // 开始搜索
    char result[N];
    search(uniqueitem, m, count, 0, n, result);
}
```

6.1.2.4 一般组合

从 n 个元素中任取 m 个元素，请输出所有取法。假设一共有 5 个元素 (12345)，从中取 3 个，则序列应该为：123、124、125、134、135、145、234、235、245、345

如果输入数据有重复，那么仍然需要统计本质不同的数据。具体实现见上一节。

一个合法的组合有这样一个特点：排在右面的数字一定严格大于左面的数字 (在下面代码中不是数字而是序号)。比如说某一位上取了 3，搜索下一位时应该是从 4 开始，否则就重复了。

```
// combination有五个参数，
//    第一个参数是待排列的字符，
//    第二个和第三个参数分别是组合数的n和m，
//    在调用的时候，第四个和第五个参数应该为0
const int N=20;
char result[N];      // 储存结果

void combination(char *item, int n, int m,
                 int pos=0, int depth=0)
{
    if (depth==m)
```

```
    {
        // 输出结果
        for (int i=0; i<n; i++) cout<<result[i];
    }
    for (int i=pos; i<n-(m-depth); i++)
    {
        result[depth]=item[i];
        combination(item, n, m, i+1, depth+1);
    }
}
```

6.1.2.5 求全部子集

示例：1、12、123、13、2、23、3

代码和求一般组合类似，区别在以下两点：首先，m 和 n 是相等的；其次，不管已经产生了多少位数，只要函数被调用，就立刻输出已经产生的结果。

若输入数据有重复，统计本质不同的数据即可。

```
// full_combination有四个参数，
//   第一个参数是待排列的字符，
//   第二个参数是字符个数，
//   在调用的时候，第三个和第四个参数应该为0
const int N=20;
char result[N];      // 储存结果

void full_combination(char *item, int n,
                int pos=0, int depth=0)
{
    for (int i=0; i<depth; i++) cout<<result[i];
    cout<<endl;

    for (int i=pos; i<n; i++)
    {
        result[depth]=item[i];
        full_combination(item, n, i, depth+1);
    }
}
```

6.1.2.6 由上一排列产生下一排列

示例：153642 经过变换之后应该是 154236。

思路：从右向左扫描，找到第一个破坏顺序的数 p (原来从右向左时数字是从小到大变化的)，

如果找不到，说明此排列已经是最后一个排列了。接下来在 p 的右边找到大于 p 的第一个数，并与 p 交换。之后将原来 p 位置右面的所有数倒序即可。

```
// 返回值含义：如果已经是最后一个序列则返回false。
// 注意：需要引用<algorithm>或自己写一个swap。
bool next_perm(int *item, int n)
{
    int j=(n-1)-1;

    while (j>=0 and item[j]>=item[j+1]) j--;

    if (j>=0)
    {
        int k=n-1;
        while (item[k]<=item[j]) k--;

        swap(item[j], item[k]);

        for (int p=j+1,q=n-1; p<q; p++,q--)
            swap(item[p], item[q]);
    }

    return j>=0;
}
```

6.1.2.7 由上一组合产生下一组合

示例：从 $1 \sim 7$ 这 7 个数字中取 4 个，则 1367 经过变换之后为 1456。

思路：从右向左寻找可以取下一个元素的数，记它的位置为 j，然后令 j 到最后一位重新取元素。注意右侧的数字一定严格大于左侧数字。

```
// 注意：只接受从1开始的*连续*数字
//     如果想使用其他元素，可以考虑索引。
bool next_combination(int *index, int n, int m)
{
    int j=m-1;
    while (j>=0 and index[j]==n-(m-1-j)) j--;
    if (j>=0)
    {
        index[j]++;
        for (int k=j+1; k<m; k++) index[k]=index[k-1]+1;
    }
    return j>=0;
}
```

6.1.3 递推

6.1.4 容斥原理

6.1.5 抽屉原理

6.1.6 置换群

6.1.7 母函数

6.1.8 MoBius 反演

6.1.9 偏序关系理论

6.2 数论

6.2.1 快速幂

假设 a,b 均为整数，且 $b>0$。怎样计算 a^b 的值?

当然可以使用 <cmath> 里的 pow 函数，不过在取整的时候要注意在结果上加一个小数，即[1]

```
int r = int(pow(2,10) + 0.0001);
```

我们也可以使用 for 循环来计算 a^b，时间复杂度 $O(b)$。显然，这个时间很慢。

实际上我们可以把 b 按照二进制来分解。以 $b=77$ 为例，$77=(1001101)_2=2^6+2^3+2^2+2^0$。求和的过程可以通过递推来完成，即 $77=(((((1\times2+0)\times2+0)\times2+1)\times2+1)\times2+0)\times2+1$。这样时间复杂度就缩小到 $O(\log b)$ 了。

知道这件事之后，我们就可以通过递推把 a^1,a^4,a^8,a^{64} 算出来，然后将它们组合成结果。

```
// 相信大家很容易就能把此代码改成求 a^b mod c
typedef long long ll;
ll quickpow(ll a, ll b)
{
    if (a==1 or b==0) return 1;

    ll r=1, t=a;
    while (b)
    {
        if (b%2) r*=t;
        b>>=1;
        t*=t;
```

[1] 比方说 pow(2,10)，正确答案是 1024，但是由于 C++ 的浮点数存在误差，所以最终结果可能是 1024.00000001，也可能是 1023.99999999。对于后者来说，如果直接取整，那么结果就会变成 1023，这是我们不希望看到的。

```
    }
    return r;
}
```

类似的思想还可以用于求 $a^1+a^2+a^3+a^4+\cdots+a^n$ 的和，如：

$$a^1+a^2+a^3+\cdots+a^{16}=a^8(a^1+a^2+\cdots+a^8)+(a^1+a^2+\cdots+a^8)$$

秦九韶算法也是将多项式计算转化为递推，以缩短计算时间、提高计算精度。

6.2.2 GCD 与 LCM

求两个数的最大公约数的基本思想：$\gcd(a,b)=\gcd(b,a \bmod b)$

两个数的最大公约数和最小公倍数的关系为 $\gcd(a,b)\times\operatorname{lcm}(a,b)=ab$，因此可以通过 $\operatorname{lcm}(a,b)=\frac{ab}{\gcd(a,b)}$ 来求出最小公倍数。

```
int gcd(int a, int b) { return b ? gcd(b,a%b) : a; }
int lcm(int a, int b) { return a/gcd(a,b)*b; }
```

如果求多个数值的最大公约数或最小公倍数，可以先求出前两个数的值，然后再将结果和第三个数一起求值。

6.2.3 素数的筛法

6.2.4 素数测试

朴素算法的时间复杂度为 $O(\sqrt{n})$，对于某些问题来说结果肯定是很可怕的。下面是一种更快的素数测试法。

费马小定理 假设 a 是一个整数，p 是一个素数，且 $\gcd(a,p)=1$，则 $a^p\equiv a(\bmod p), a^{p-1}\equiv 1(\bmod p)$。

费马小定理的逆命题是不成立的，然而，由于出错的概率比较低，所以我们仍然可以通过它来进行测试。

Miller-Rabin 素数测试 不断选取（下面代码只选了四个数）不超过 $n-1$ 的底数 a，每次判断是否为 $a^{n-1}\equiv 1(\bmod n)$。如果有一次不成立，则 n 为合数，否则基本上可以断定为素数（仍然有合数的可能）。

```
// 进行单次测试，实际上是计算 a^(n-1) mod n 的结果是否为 1
bool miller(int a, int n)
{
```

```
    int b=n-1, d=1, t=a;
    while (n-1 > 0)
    {
        if (t==1) break;
        if (b%2) d=d*t%n;
        b>>=1; t*=t;
    }
    return d%n==1;
}

bool isprime(int n)
{
    if (n%2==0) return n==2;
    // 直接用 2, 3, 5, 7 做测试
    return miller(2,n) and miller(3,n) and
        miller(5,n) and miller(7,n);
}

// 也可以通过产生随机数来进行测试。注意，下面的 a 不会大于 32768
// int a = rand() * (n-2) / RAND_MAX + 1;
```

单次 Miller-Rabin 测试的的正确率为 75%，不过在 n 不太大 ($n \leqslant 2.5\times10^{10}$) 的时候，可以认为 Miller-Rabin 测试是正确的。如果只测试 $a = 2, 3, 5, 7$，则通过测试的最小合数为 3,215,031,751；通过 $a = 2, 3, 5, 7, 11$ 测试的最小合数为 2,152,302,898,747；而通过 $a = 2, 3, 5, 7, 11, 13, 17$ 测试的最小合数为 341,550,071,728,321。

6.2.5 欧拉函数

欧拉函数 对于正整数 n 来说，欧拉函数 $\varphi(n)$ 表示小于等于 n 且与 n 互质的数的个数。

欧拉函数有个重要性质 (欧拉定理)：如果 $\gcd(k, m) = 1$，则 $k^{\varphi(m)} \equiv 1(\mod m)$

6.2.5.1 欧拉函数的应用

6.2.5.2 单独计算

6.2.5.3 欧拉函数表

欧拉函数有两条性质：

1. 对于素数 p，$\varphi(p) = p - 1, \varphi(p^n) = p^n(p - 1)$
2. 如果 $\gcd(x, y) = 1$，则 $\varphi(xy) = \varphi(x)\varphi(y)$，也就是说 $\varphi(x)$ 是积性函数。

下面的代码就是根据欧拉函数的第二条性质来计算的。

```
void phi_table(int *a, int n)
{
    memset(a,0,sizeof(int)*(n+1));
    a[1]=1;
    for (int i=2; i<=n; i++)
        if (!a[i])
            for (int j=1; j<=n; j+=i)
            {
                if (!a[j]) a[j]=j;
                a[j]=a[j]/i*(i-1)
            }
}
```

6.2.6 扩展欧几里得算法

以下算法用于求 $ax+by=\gcd(a,b)$ 的一组整数解。注意，方程的解不是唯一的，因为使 x 增加若干个 b，使 y 减少同等数量的 a，结果还是不变的。

代码中注释 1 的推导方法：按照 gcd 的基本思想，我们可以得到一个式子。与原方程联立，则有

$$\begin{cases} ax_1+by_1=\gcd(a,b) \\ bx_2+(a\%b)y_2=gcd(b,a\%b) \end{cases}$$

很明显，等号右面的两个值是相等的，而且有 $a\%b=a-(a/b)\times b$，那么

$$ax_1+by_1=bx_2+[a-(a/b)\times b]\times y_2$$

也就是

$$ax_1+by_1=ay_2+b[x_2-(a/b)y_2]$$

由于等式是恒成立的，所以可以得出

$$x_1=y_2, y_1=x_2-(a/b)\times y_2$$

```
// 返回值为 a 和 b 的最大公约数
int exgcd(int a, int b, int &x, int &y)
{
    if (b)   // 1
    {
        int r=exgcd(b,a%b,x,y);
        int t=x; x=y; y=t-a/b*y;
```

```
        return r;
    }
    else
    {
        x=1,y=0;
        return a;
    }
}
```

6.2.7 线性同余方程

如何求 $ax \equiv b(\bmod n)(n > 0)$ 的解？如果有解，有几个解？[2]

设 y 是一个整数，那么该式就可以转化为 $ax - ny = b$，也就是说，如果上面那个方程有解，那么这个二元方程也必须有解。为了利用扩展欧几里得算法，我们要对方程稍作转化。令 $d = \gcd(a, n)$，我们先求解 $ax - ny = d$（若使方程有解，需要 $d|b$），可知这个 x 乘上 $\frac{b}{d}$ 就是最终的一个解。

此方程共有 d 个解，这些解相差 $\frac{n}{d}$ 的整数倍。求出一个解之后，其他解也就能求出来了。

最终算法如下。其中没贴 exgcd 的代码，往前翻几页就有了。

```
// 如果有解则输出解并返回 true，否则返回 false
bool mod_equation(int a, int b, int n)
{
    int x,y;
    int d=exgcd(a,n,x,y);

    if (b%d != 0)
        return false;
    else
    {
        int e = x*(b/d) % n;
        for (int i=0; i<d; i++)
            cout<< (e + i*(n/d)) % n <<endl;
        return true;
    }
}
```

对代码稍作修改，也可用于求 $ax + by = c$ 的解。

[2] 假如存在 x_0 使 $ax_0 \equiv b(\bmod n)$ 成立，那么很明显，$x_0 + kn(k \in \mathbb{Z})$ 也可以使式子成立。在数论中，我们认为这是一个解。

6.2.8 同余方程组中国剩余定理

6.2.8.1 模不互素

6.2.8.2 模两两互素

6.3 高斯消元法

6.4 概率问题

6.5 进位制

6.5.1 快速幂

假设 a, b 均为整数，且 $b > 0$。怎样计算 a^b 的值?

当然可以使用 <cmath> 里的 pow 函数，不过在取整的时候要注意在结果上加一个小数，即[3]

```
int r = int(pow(2,10) + 0.0001);
```

我们也可以使用 for 循环来计算 a^b，时间复杂度 $O(b)$。显然，这个时间很慢。

实际上我们可以把 b 按照二进制来分解。以 $b = 77$ 为例，$77 = (1001101)_2 = 2^6 + 2^3 + 2^2 + 2^0$。求和的过程可以通过递推来完成，即 $77 = (((((1 \times 2 + 0) \times 2 + 0) \times 2 + 1) \times 2 + 1) \times 2 + 0) \times 2 + 1$。这样时间复杂度就缩小到 $O(\log b)$ 了。

知道这件事之后，我们就可以通过递推把 a^1, a^4, a^8, a^{64} 算出来，然后将它们组合成结果。

```
// 相信大家很容易就能把此代码改成求 a^b mod c
typedef long long ll;
ll quickpow(ll a, ll b)
{
    if (a==1 or b==0) return 1;

    ll r=1, t=a;
    while (b)
    {
        if (b%2) r*=t;
        b>>=1;
        t*=t;
    }
    return r;
}
```

[3]比方说 pow(2,10)，正确答案是 1024，但是由于 C++ 的浮点数存在误差，所以最终结果可能是 1024.00000001，也可能是 1023.99999999。对于后者来说，如果直接取整，那么结果就会变成 1023，这是我们不希望看到的。

类似的思想还可以用于求 $a^1+a^2+a^3+a^4+\cdots+a^n$ 的和，如：

$$a^1+a^2+a^3+\cdots+a^{16}=a^8(a^1+a^2+\cdots+a^8)+(a^1+a^2+\cdots+a^8)$$

秦九韶算法也是将多项式计算转化为递推，以缩短计算时间、提高计算精度。

6.5.2 斐波那契进制

6.5.3 康托展开

6.6 数学元素

6.6.1 大数

6.6.2 分数

6.6.3 矩阵

6.7 其他数学问题

6.7.1 约瑟夫环

6.7.2 01 分数规划

第七章 计算几何

7.1 上大学以前学过的知识

7.2 三角形

7.2.1 面积

7.2.1.1 已知三点坐标求面积

7.2.1.2 已知三边长求面积

7.2.1.3 多边形面积

7.2.2 重要的点

7.3 多边形的简单算法

7.4 凸包

7.5 扫描线算法

7.6 多边形的内核

7.7 几何工具的综合应用

7.8 半平面求交

7.9 可视图的建立

7.10 点集最小圆覆盖

7.11 对踵点

第八章　ACM 与 Java

8.1　Hello world

8.2　Java 的输入输出

8.2.1　输入

8.2.2　输出

8.3　Java 的数据类型

8.3.1　基本类型

8.3.2　数组

8.3.3　字符串

8.3.4　大数

8.4　函数和全局变量

8.5　查找和排序

8.6　正则表达式

8.7　其他

第九章　STL

9.1　none

第十章　附录

10.1　C or C++ 语言本身的问题

10.2　超级空白文件

10.3　经验教训

10.4　gdb 常用命令