ACM/ICPC 代码库

2015 年 4 月 30 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。

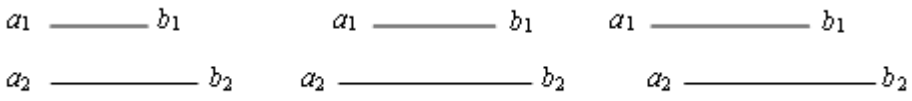

图 1.1: 选择不相交区间：如图所示，不论区间 1、2 的相对位置如何，选择区间 1 都会为以后的选择留下更大的剩余空间。

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 $[s, t]$。

【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。

[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。

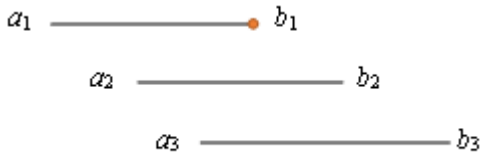

图 1.2: 区间选点问题：对于第 1 个区间来说，显然，选择它的右端点是明智的。因为它比前面的点能覆盖更大的范围。

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

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

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 ababacb。

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:  abababacabababababcbc
P:  ababacb
P1:  ababacb
P2:   ababacb
P3:    ababacb
......
Pn:           ababacb
```

KMP 算法的尝试：

```
S:  abababacabababababcbc
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：

```
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 ±1-RMQ 问题

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。

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

3.6.1.1 实现

3.6.1.2 完全二叉树

3.6.2 哈夫曼树

3.6.3 字典树

3.7 二叉排序树

3.7.1 基本的二叉排序树

3.7.2 C++ 的 set 和 map

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]; }
```

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

5.3 拓扑排序

5.3.1 DFS

5.3.2 辅助队列

5.3.3 拓扑排序个数

5.4 欧拉路径

5.5 生成树 (森林)

5.5.1 最小生成树 (Prim)

5.5.2 最小生成树 (Kruskal)

5.5.3 次小生成树

5.5.4 最小生成森林

5.5.5 最小树形图

5.6 最短路

5.6.1 如何选用

5.6.2 Dijkstra 算法 (邻接矩阵)

5.6.3 优先队列优化的 Dijkstra 算法 (邻接表)

5.6.4 Bellman-Ford 算法

5.6.5 SPFA 算法

5.6.6 Floyd 算法

5.6.6.1 多源最短路

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

6.1.2.2 全排列 (无重复)

6.1.2.3 全排列 (有重复)

6.1.2.4 一般组合

6.1.2.5 求全部子集

6.1.2.6 由上一排列产生下一排列

6.1.2.7 由上一组合产生下一组合

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

6.2.2 GCD 与 LCM

6.2.3 素数的筛法

6.2.4 素数测试

6.2.5 欧拉函数

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