# 数据结构与算法
# (四)
# 贪心、分治算法

杜育根

ygdu@sei.ecnu.edu.cn

# 本章内容

一、 贪心法

常见问题

huffman编码

模拟退火

二、 分治法

归并排序

快速排序

# 贪心算法

- 所谓贪心算法，是指在对问题求解时，总是作出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，它所做出的仅是在某种意义上的局部最优解。
- 贪心算法不是对所有问题都能得到整体最优解，但对范围相当广泛的许多问题它能产生整体最优解或其近似解。
- 基本思路如下：

1.建立数学模型来描述问题；

2.把求解问题分成若干个子问题

3.对每一子问题求解，得到子问题的局部最优解

4.把子问题的局部最优解合成原问题的一个解

# 思考：钱币问题

- 某人带着3种面值的钱币去购物，有1元、2元、5元的，钱币数量不限；

- 他需要支付M元，问怎么支付，才能使钱币数最少？

# 钱币问题思路

```
#include <iostream>
using namespace std;
#define NUM 3
const int Value[NUM] = {5, 2, 1};
int main(){
    int i, money;
    int ans[NUM]={0};
    cout <<  “输入总钱金额：";
    cin >> money;
    for(i= 0; i < NUM; i++){       //求每种钱币的数量
        ans[i] = money/Value[i];
        money = money - ans[i]*Value[i];
    }
    for(i= 0; i < NUM; i++)
        cout<<Value[i]<< “元钱币数："<<ans[i]<<endl;
    return 0;
}
```

# 贪心算法的缺陷

- 虽然每一步选钱币的操作，并没有从整体最优来考虑，而是只在当前步骤选取了局部最优，但是结果是全局最优的。
- 然而，局部最优并不总是能导致全局最优。
- 钱币问题，用贪心法，一定能得到最优解吗？

# 贪心算法的反例

- 在钱币问题中，如果改换一下参数，就不一定能得到最优解。例如：钱币面值比较奇怪，是1、2、4、5、6元，支付9元，如果用贪心法，答案是6 + 2 + 1，需要3个钱币，而最优的5 + 4只需要2个钱币。

- 所以，在钱币问题中，用贪心法是否能得到最优，跟钱币的面值有关。如果是1、2、5这样的面值，贪心是有效的，而对于1、2、4、5、6这样的面值，贪心是无效的。

- 任意面值钱币问题的求解：**动态规划**。

# 思考

- 什么面值的钱币能用贪心法?

一个简单的判别标准是:

对任何一个面值钱币，要大于比它小的所有钱币面值之和。

# 贪心法的设计思想

- 基本思想：看一步走一步，而且只看一步；在每一步，选当前最优的；不回头，不改变已有的选择。
- 贪心法在解决问题的策略上目光短浅，只根据当前已有的信息就做出选择，而且一旦做出了选择，不管将来有什么结果，这个选择都不会改变。换言之，贪心法并不是从整体最优考虑，它所做出的选择只是在某种意义上的局部最优。
- 这种局部最优选择并不总能获得整体最优解（Optimal Solution），但通常能获得近似最优解（Near-Optimal Solution）。

# 贪心法求解的问题的特征

（1）最优子结构性质

当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质，也称此问题满足最优性原理。

（2）贪心选择性质

所谓贪心选择性质是指问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来得到。

- 动态规划法通常以自底向上的方式求解各个子问题，而贪心法则通常以自顶向下的方式做出一系列的贪心选择。

# 活动安排问题（区间调度问题）

有很多电视节目，给出它们的起止时间。有些节目时间冲突。问能完整看完的电视节目最多有多少?

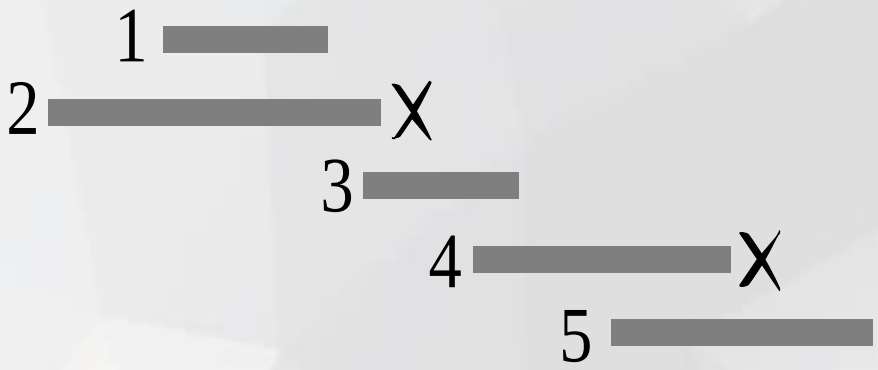
- Input:输入数据包含多个测试实例，每个测试实例的第一行只有一个整数n(n<=100)，表示你喜欢看的节目的总数，然后是n行数据，每行包括两个数据Ti_s,Ti_e (1<=i<=n)，分别表示第i个节目的开始和结束时间，为了简化问题，每个时间都用一个正整数表示。n=0表示输入结束，不做处理。
- Output:对于每个测试实例，输出能完整看到的电视节目的个数，每个测试实例的输出占一行。

Sample Input

```
12
1 3
3 4
0 7
3 8
15 19
15 20
10 15
8 18
6 12
5 10
4 14
2 9
0
```

Sample Output

```
5
```



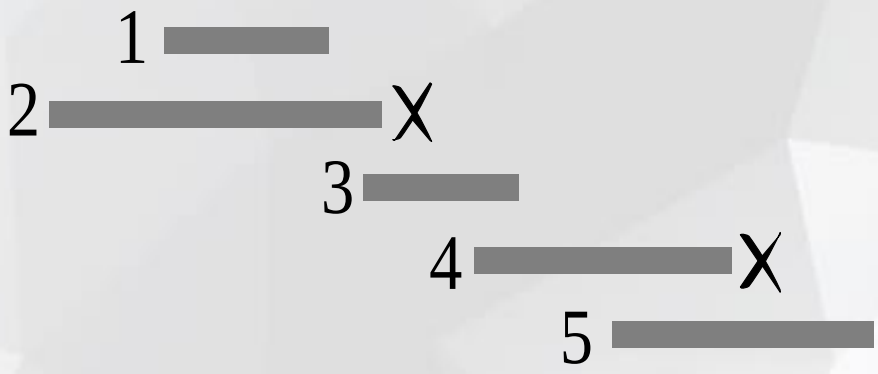
# 选择贪心策略

- 解题的关键在于选择什么贪心策略，才能安排尽量多的活动。由于活动有开始时间和结束时间，考虑三种贪心策略：

(1) 最早开始时间。

(2) 最早结束时间。

(3) 用时最少。

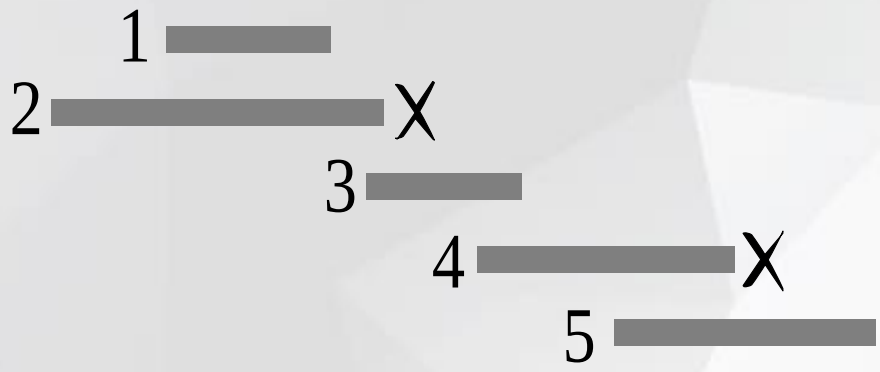


# 三种贪心策略：

（1）最早开始时间：错误，因为如果一个活动迟迟不终止，后面的活动就无法开始。

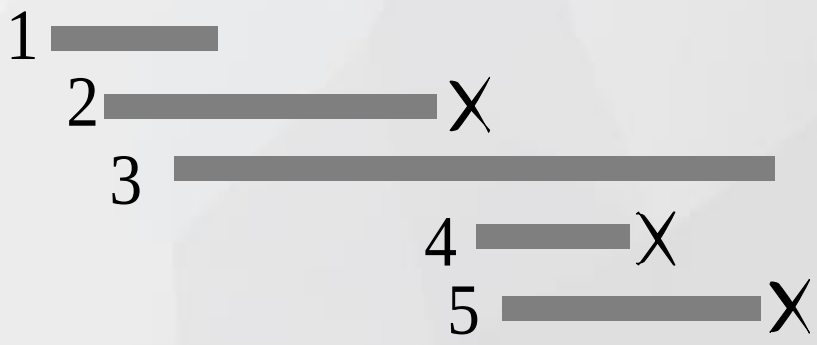
（2）最早结束时间：合理，一个尽快终止的活动，可以容纳更多的后续活动。

（3）用时最少：错误。



# 区间覆盖问题

- 给定一个长度为n的区间，再给出m条线段的左端点（起点）和右端点（终点）。问最少用多少条线段可以将整个区间完全覆盖。



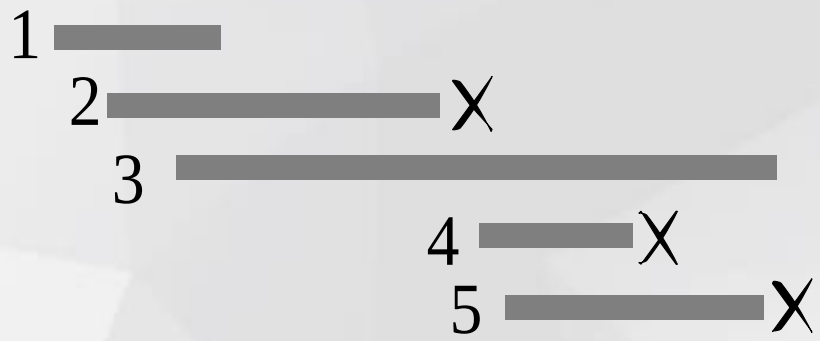
# 贪心策略

- 贪心：尽量找出更长的线段。
- 解题步骤是：

（1）把每个线段按照左端点递增排序。

（2）设已经覆盖的区间是[L，R]，在剩下的线段中，找所有左端点小于等于R，且右端点最大的线段，把这个线段加入到已覆盖区间里，并更新已覆盖区间的[L, R]值。

（3）重复步骤（2），直到区间全部覆盖。



# 练习题：最优装载问题

- 有n种药水，体积都是V，浓度不同。把它们混合起来，得到浓度不大于w%的药水。问怎么混合，才能得到最大体积的药水？注意一种药水要么全用，要么都不用，不能只取一部分。

# 贪心策略

- 有n种药水，体积都是V，浓度不同。把它们混合起来，得到浓度不大于w%的药水。问怎么混合，才能得到最大体积的药水？注意一种药水要么全用，要么都不用，不能只取一部分。
- 要求配置浓度不大于w%的药水，贪心思路：尽量找浓度小的药水。
- 先对药水按浓度从小到大排序，药水的浓度不大于w%就加入，如果药水的浓度大于w%，计算混合后总浓度，不大于w%就加入，否则结束判断。

# 练习题：多机调度问题

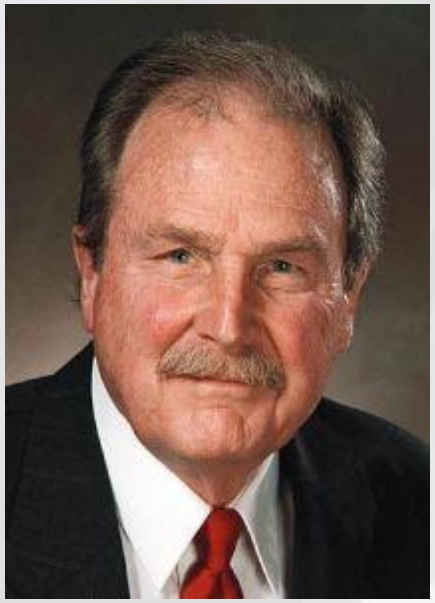
- 有n个独立的作业，由m台相同的机器进行加工。
- 作业i的处理时间为ti，每个作业可在任何一台机器上加工处理，但不能间断、拆分。
- 要求给出一种作业调度方案，在尽可能短的时间内，由m台机器加工处理完成这n个作业。
- 贪心策略：最长处理时间的作业优先，即把处理时间最长的作业分配给最先空闲的机器。让处理时间长的作业得到优先处理，从而在整体上获得尽可能短的处理时间。

# Huffman编码

# 著名贪心算法：Huffman编码

- Huffman编码是贪心思想的典型应用，是一个很有用的、很著名的算法。

- Huffman编码是“前缀”最优编码。



# 什么是编码

- 例：给出一段字符串，它只包含A、B、C、D、E这5种字符。字符出现频率不同。
- 简单编码：

| 字符 | A | B | C | D | E |
|---|---|---|---|---|---|
| 频次 | 3 | 9 | 6 | 15 | 19 |
| 编码 | 000 | 001 | 010 | 011 | 100 |

- 每个字符用3位二进制数表示，存储的总长度是：3*(3+9+6+15+19) = 156。

- 变长编码：出现次数多的字符用短码表示，出现少的用长码表示。

| 字符 | A | B | C | D | E |
|---|---|---|---|---|---|
| 频次 | 3 | 9 | 6 | 15 | 19 |
| 编码 | 1100 | 111 | 1101 | 10 | 0 |

- 存储的总长度是：3*4 + 9*3 + 6*4 + 15*2 + 19*1 = 112。
- 第二种方法相当于对第一种方法，压缩比是：156/112=1.39。

# 编码算法的基本要求：

- 编码后得到的二进制串，能唯一地进行解码还原。

- 第一种方法是正确的，每3位二进制数对应一个字符。
- 第二种方法，也是正确的，

  例如"1100 111 10 0 1101"，

  解码后唯一得到"ABDEC"。

| 字符 | A | B | C | D | E |
|---|---|---|---|---|---|
| 频次 | 3 | 9 | 6 | 15 | 19 |
| 编码 | 1100 | 111 | 1101 | 10 | 0 |

- 胡乱设定编码方案，很可能错误，例如：

| 字符 | A | B | C | D | E |
|---|---|---|---|---|---|
| 频次 | 3 | 9 | 6 | 15 | 19 |
| 编码 | 100 | 10 | 11 | 1 | 0 |

- 编码无法解码还原。例如"100"，是"A"、"BE"还是"DEE"呢?
- 错误的原因是，某个编码是另一个编码的**前缀**(prefix)，即这两个编码有包含关系，导致了混淆。

# 如何找最优编码方案?

- 有没有比第二种编码方法更好的方法?
- 这引出了一个字符串存储的常见问题：给定一个字符串，如何编码，能使得编码后的总长度最小？即如何得到一个最优解?

- Huffman编码是前缀编码算法中的最优算法

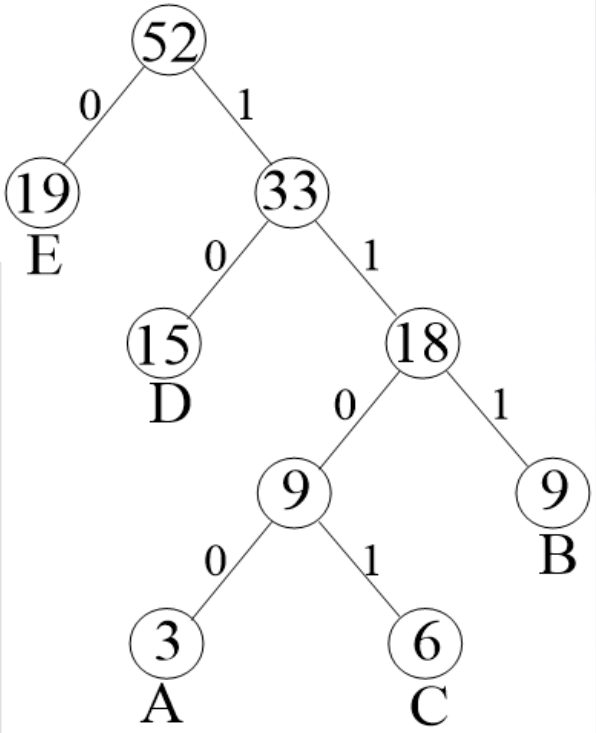
# Huffman编码思想

- Huffman编码是利用贪心思想构造二叉编码树的算法。
- 首先对所有字符按出现频次升序排好位置
- 然后从最少频次的字符开始，用贪心思想安排在二叉树上
- 从底层往顶层生成二叉树，每个节点内数字是这个子树下字符频次之和
- 每一步字符位置都要按频次与节点数字重新排位

| 字符 | A | B | C | D | E |
|---|---|---|---|---|---|
| 频次 | 3 | 9 | 6 | 15 | 19 |
| 编码 | 1100 | 111 | 1101 | 10 | 0 |

这个过程可以保证频次少的字符被放在树底，编码更长；频次高的字符放在上层，编码更短。

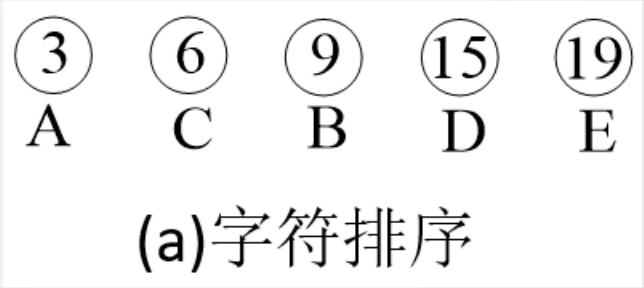
可以证明，Huffman算法符合“最有子结构性质“和”贪心选择性质“。编码结果是最优的。



# 贪心过程

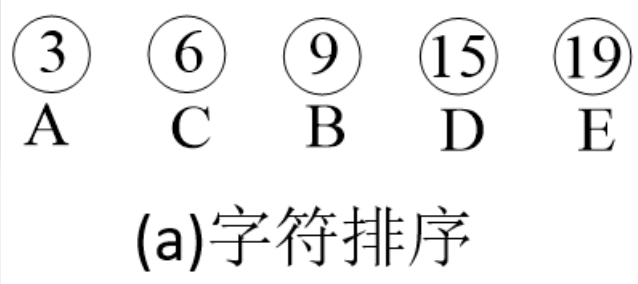
- 对所有字符按频次排序：

| 字符 | A | B | C | D | E |
|---|---|---|---|---|---|
| 频次 | 3 | 9 | 6 | 15 | 19 |

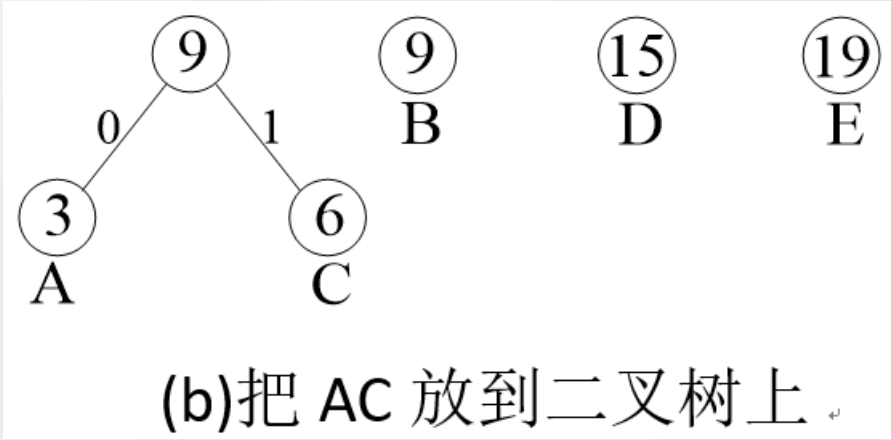



(a)字符排序

- 从最少的字符开始，用贪心思想安排在二叉树上

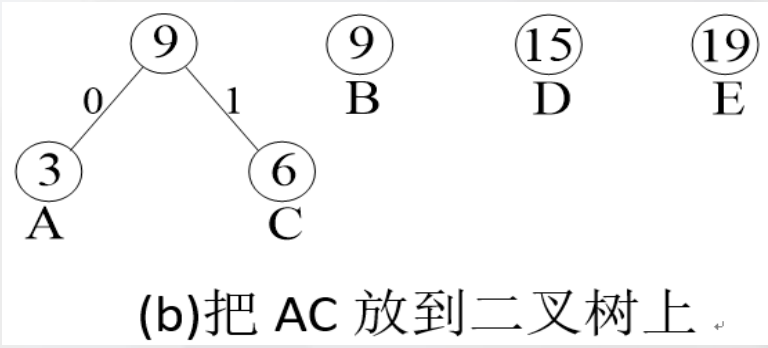


(a)字符排序

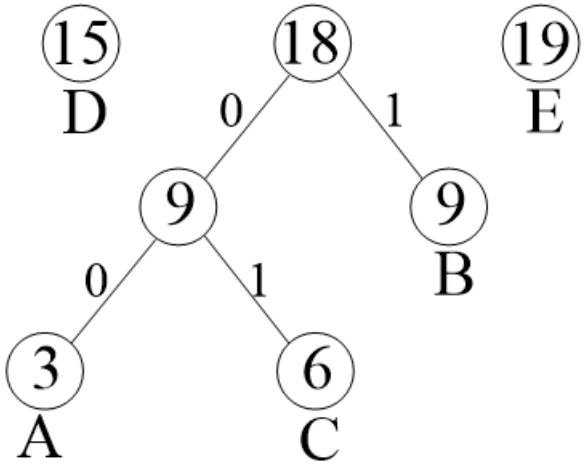


(b)把 AC 放到二叉树上


(a)字符排序

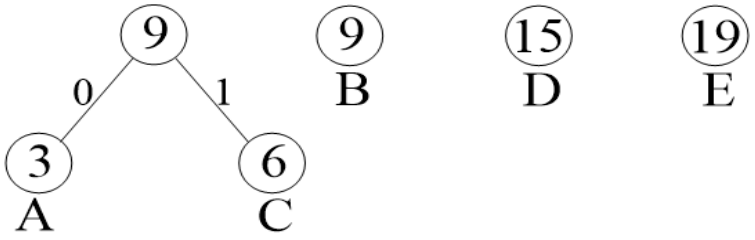



(b)把 AC 放到二叉树上

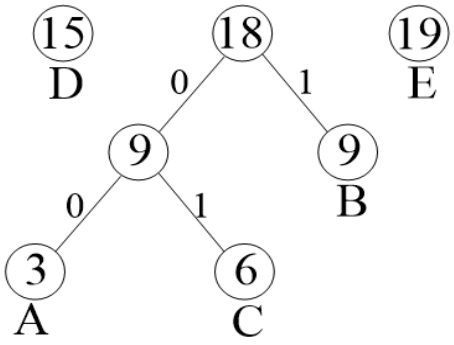



(c) 把 B 放到二叉树上，调整 D

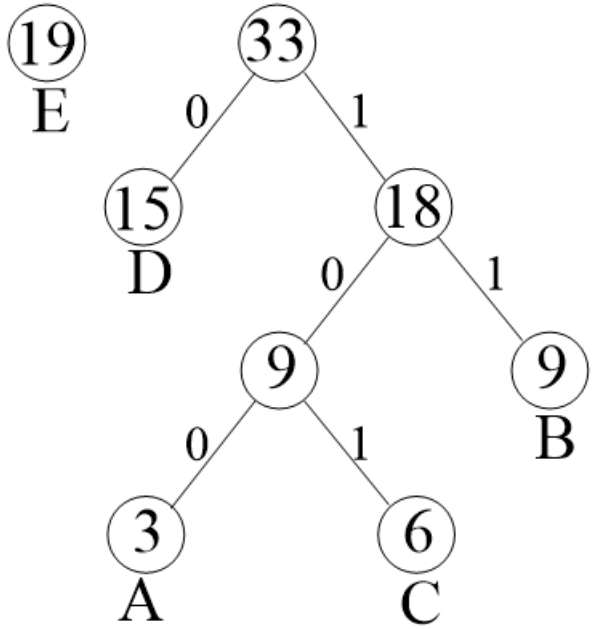

(a)字符排序




(b)把 AC 放到二叉树上

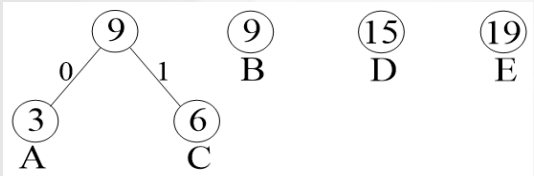



(c) 把 B 放到二叉树上，调整 D

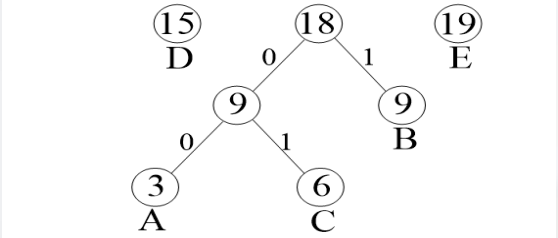



(d)把 D 放到二叉树上，调整 E

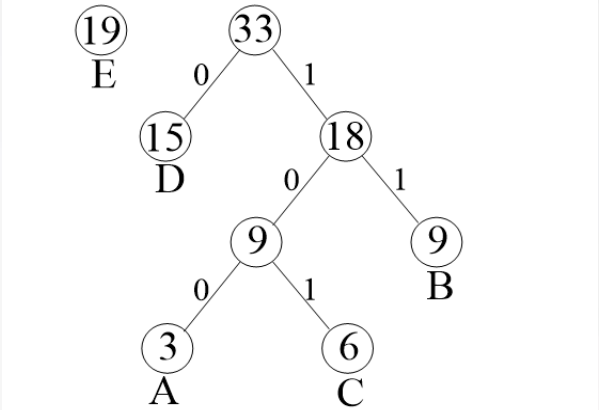

(a)字符排序



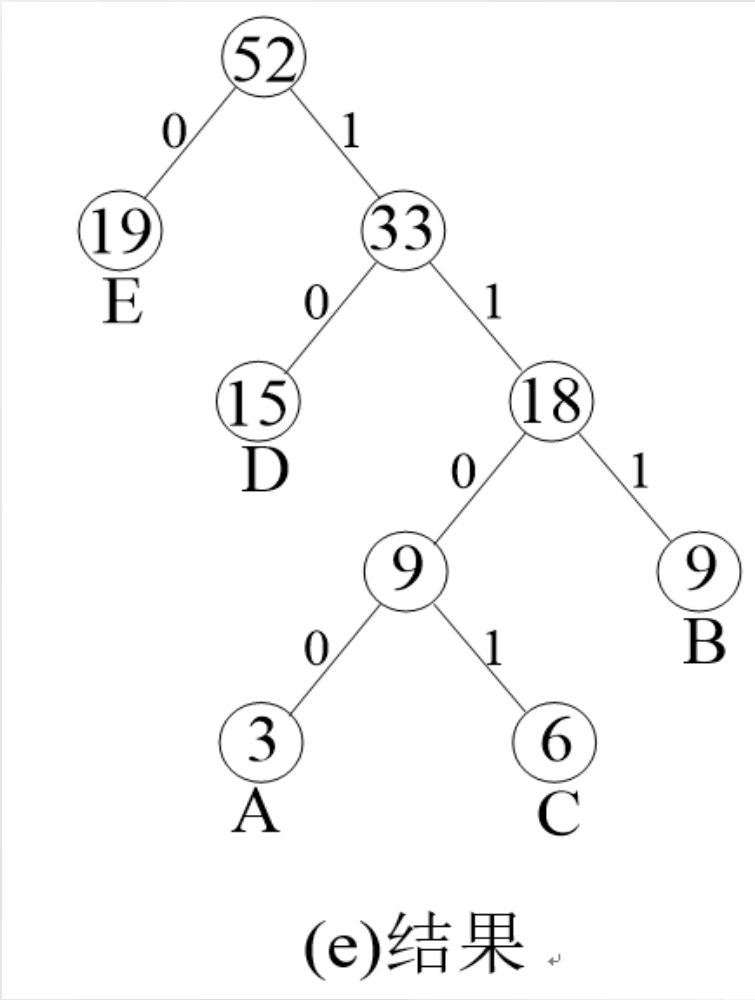

(b)把 AC 放到二叉树上




(c) 把 B 放到二叉树上，调整 D




(d)把 D 放到二叉树上，调整 E



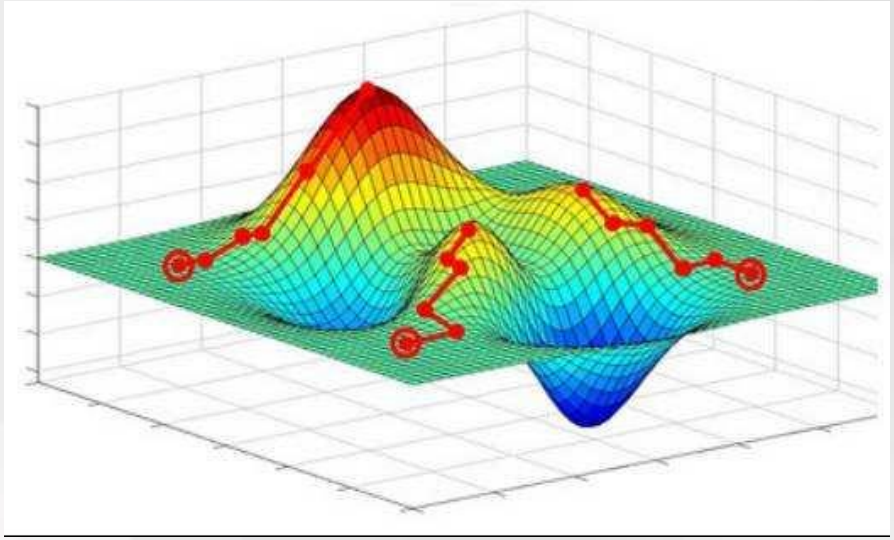

(e)结果

# 练习题：Entropy

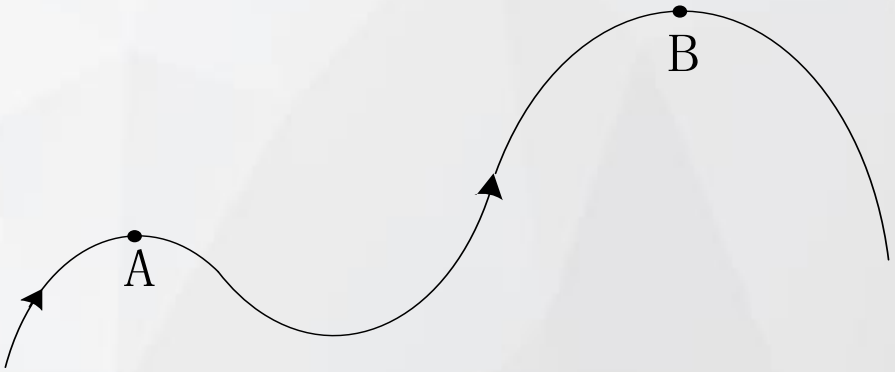
- 输入一个字符串，分别用普通ASCII编码（每个字符8bit）和huffman编码，输出编码后的长度，并输出压缩比。
- Sample Input:
- AAAAABCD
- THE_CAT_IN_THE_HAT
- END
- Sample Output:
- 64 13 4.9
- 144 51 2.8

# 物理的启发：模拟退火

- 模拟退火算法基于这样一个物理原理：
- 一个高温物体降温到常温，温度越高时，降温的概率越大（降温更快），温度越低降温概率越小（降温更慢）。
- 模拟退火算法利用这样一种思想进行搜索，即进行多次降温（迭代），直到获得一个可行解。



# 模拟退火算法：贪心+概率

- 下图中，A是局部最高点，B是全局最高点。
- 

- 普通的贪心算法，如果当前状态在A附近，会一直爬山，最后停在局部最高点A，无法到达B。
- 模拟退火算法能跳出A，得到B。因为它不仅往上爬山，而且以一定概率接受比当前点更低的点，使程序有机会摆脱局部最优而到达全局最优。这个概率会随时间不断减小，从而最后能限制在最优解附近。

# 模拟退火算法的主要步骤

（1）设置一个初始的温度T。

（2）温度下降，状态转移。从当前温度，按降温系数下降到下一个温度。在新的温度，计算当前状态。

（3）如果温度降到设定的温度下界，程序停止。

# 伪代码

```
eps = 1e-8;          //终止温度，接近于0，用于控制精度
T = 100;             //初始温度，应该是高温，以100度为例
delta = 0.98;        //降温系数，控制退火的快慢，小于1
g(x);                //状态x时的评价函数，例如物理意义上的能量
now, next;           //当前状态和新状态
while(T > eps){              //如果温度未降到eps
    g(next), g(now);         //计算能量。
    dE= g(next)-g(now);  //能量差
    if(dE >= 0)              //新状态更优，接受新状态
        now = next;
    else if(exp(dE/T)> rand())
                //如果新状态更差，在一定概率下接受它，e^(dE/T)
        now = next;
    T *= delta;              //降温，模拟退火过程
}
```

# 模拟退火在算法竞赛中的典型应用

- 函数最值问题
- TSP旅行商问题
- 最小圆覆盖
- 最小球覆盖

# 练习题：求函数最值

- 函数F(x) = 6 * x^7+8*x^6+7*x^3+5*x^2-y*x
- 其中x的范围是0 <= x <=100。
- 输入y值，输出F(x)的最小值。

- 思考：数学的求解方法是?
- 下面用模拟退火编程。

# 完整代码

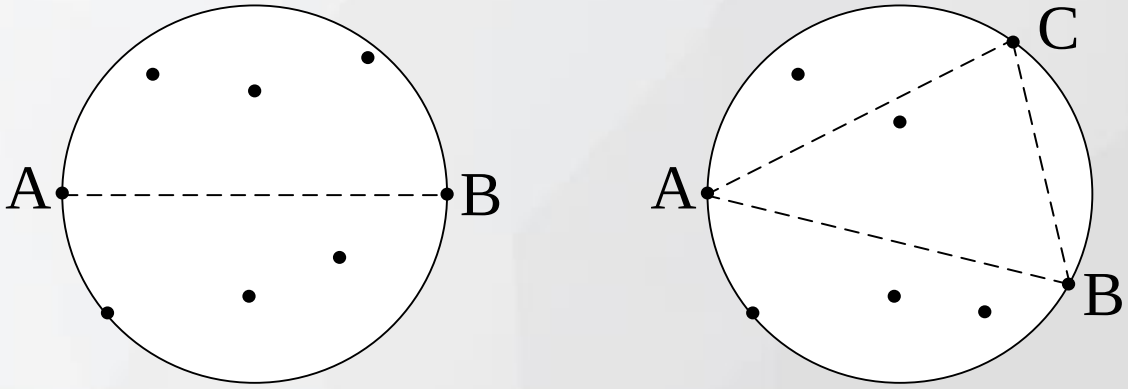
```
double solve(){
    double T = 100;         //初始温度
    double delta = 0.98;   //降温系数
    double x = 50.0;        //x的初始值
    double now = func(x);  //计算初始函数值
    double ans = now;       //返回值
    while(T > eps){         //eps是终止温度
    int f[2]={1,-1};
        double newx = x+f[rand()%2]*T;
            //按概率改变x，随T的降温而减少
        if(newx >= 0 && newx <= 100){
            double next = func(newx);
            ans = min(ans,next);
            if(now - next > eps){x = newx; now = next;}    //更新x
        }
        T *= delta;
    }
  return ans;
}
```

# 模拟退火算法的缺点

- 模拟退火算法用起来非常简单方便，不过也有缺点。
- 它得到的是一个可行解，而不是精确解。
- 例如上面的例题，计算到4位小数点的精度就停止，实际上是一个可行解，所以算法的效率和要求的精度有关。
- 一般情况下，模拟退火算法的复杂度会比其它精确算法差。应用时需要仔细选择初始温度T、降温系数delta、终止温度eps等。

# 练习题：最小圆覆盖

给定n个点的平面坐标，求一个半径最小的圆，把n个点全部包围，部分点在圆上。




两点定圆或三点定圆

# 完整代码

```
void min_cover_circle(Point *p, int n, Point &c, double &r){
    double T = 100.0;      //初始温度
    double delta = 0.98;  //降温系数
    c = p[0];
    int pos;
    while (T > eps){       //eps是终止温度
        pos = 0; r=0;       //初始：  p[0]是圆心，半径是0
        for(int i = 0; i <= n - 1; i++)  //找距圆心最远的点
            if (Distance(c, p[i]) > r){
                r = Distance(c, p[i]);     //距圆心最远的点，肯定在圆周上
                pos = i;
            }
        c.x += (p[pos].x - c.x) / r * T;  //逼近最后的解
        c.y += (p[pos].y - c.y) / r * T;
        T *= delta;
    }
}
```

# 贪心习题

hdu 1789 "Doing Homework again"。活动安排问题。

hdu 1050 "Moving Tables"，空间问题，模型和活动安排问题一样。

hdu 2546 “饭卡”，普通背包问题。

hdu 3348 "coins"，钱币问题。

hdu 4864 "task"，不错的题。

poj 1328 "Radar Installation"，几何问题，建模为活动安排问题。

poj 1089 "Intervals"，区间覆盖问题，给定很多线段，合并线段，使得合并后间隔最小。

# 分治算法

# 分治思想

分治法是广为人知的算法思想，容易理解。

遇到一个难以直接解决的大问题时，自然想到把它划分成一些规模较小的子问题，各个击破，“分而治之（Divide and Conquer）”。

# 分治法需要符合两个特征

（1）平衡子问题：子问题的规模大致相同。能把问题划分成大小差不多相等的k个子问题，最好k = 2，即分成两个规模相等的子问题。子问题规模相等的处理效率，比子问题规模不等的处理效率要高。

（2）独立子问题：子问题之间相互独立。这是区别于动态规划算法的根本特征，在动态规划算法中，子问题是相互联系的，而不是相互独立的。

# 特别说明

分治法不仅能够让问题变得更容易理解和解决，而且能大大优化算法的复杂度，一般情况下，能把O(n)的复杂度优化到O(logn)。

这是因为，局部的优化有利于全局；一个子问题的解决，其影响力扩大了k倍，即扩大到了全局。

一个简单的例子：在一个有序的数列中查找一个数。简单的办法是从头找到尾，复杂度是O(n)。如果用分治法，即“折半查找”，最多只需要logn次就能找到。

# 分治法如何编程

分治法的思想，几乎就是递归的过程，用递归程序实现分治法是很自然的。

用分治法建立模型时，解题步骤分为三步：

（1）分解（Divide）：把问题分解成独立的子问题；

（2）解决（Conquer）：递归解决子问题；

（3）合并（Combine）：把子问题的结果合并成原问题的解。

分治法的经典应用，有汉诺塔、快速排序、归并排序等。

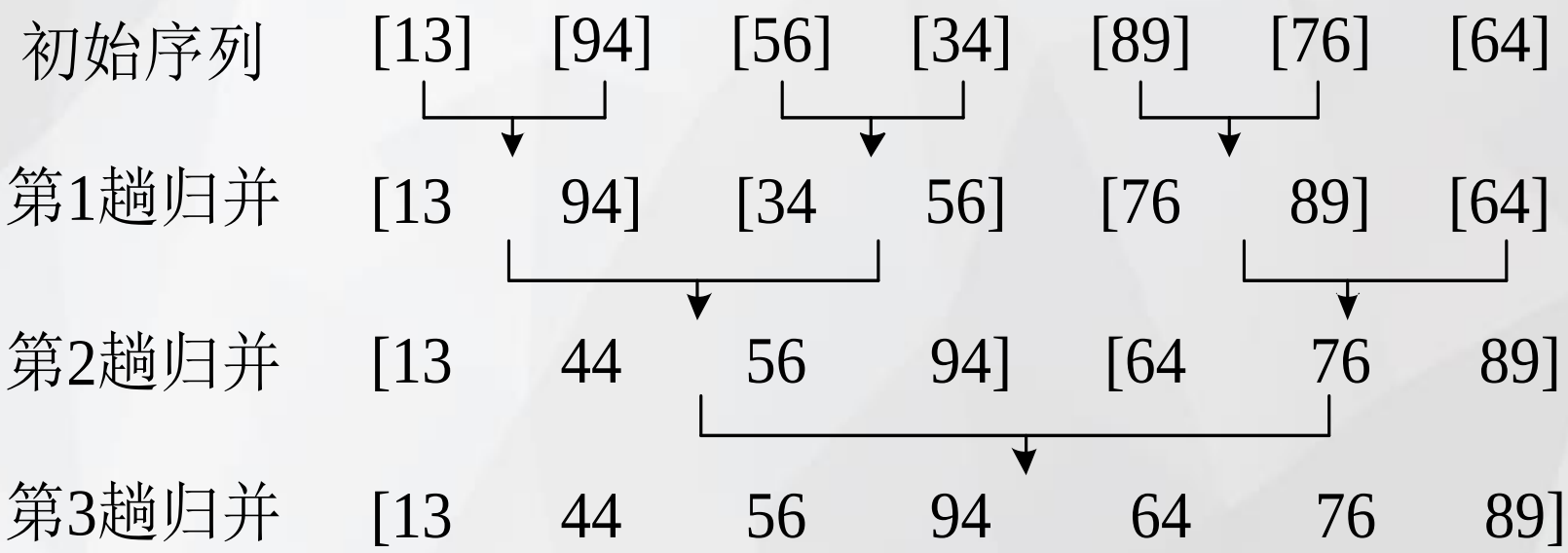
# 如何用分治思想设计排序算法?

（1）分解。把原来无序的数列，分成两部分；对每个部分，再继续分解成更小的两部分……在归并排序中，只是简单地把数列分成两半。在快速排序中，是把序列分成左右两部分，左部分的元素都小于右部分的元素；分解操作是快速排序的核心操作。

（2）解决。分解到最后不能再分解，排序。

（3）合并。把每次分开的两个部分合并到一起。归并排序的核心操作是合并，其过程类似于交换排序。快速排序并不需要合并操作，因为在分解过程中，左右部分已经是有序的。

# 归并排序


初始序列 [13] [94] [56] [34] [89] [76] [64]
第1趟归并 [13 94] [34 56] [76 89] [64]
第2趟归并 [13 44 56 94] [64 76 89]
第3趟归并 [13 44 56 94 64 76 89]

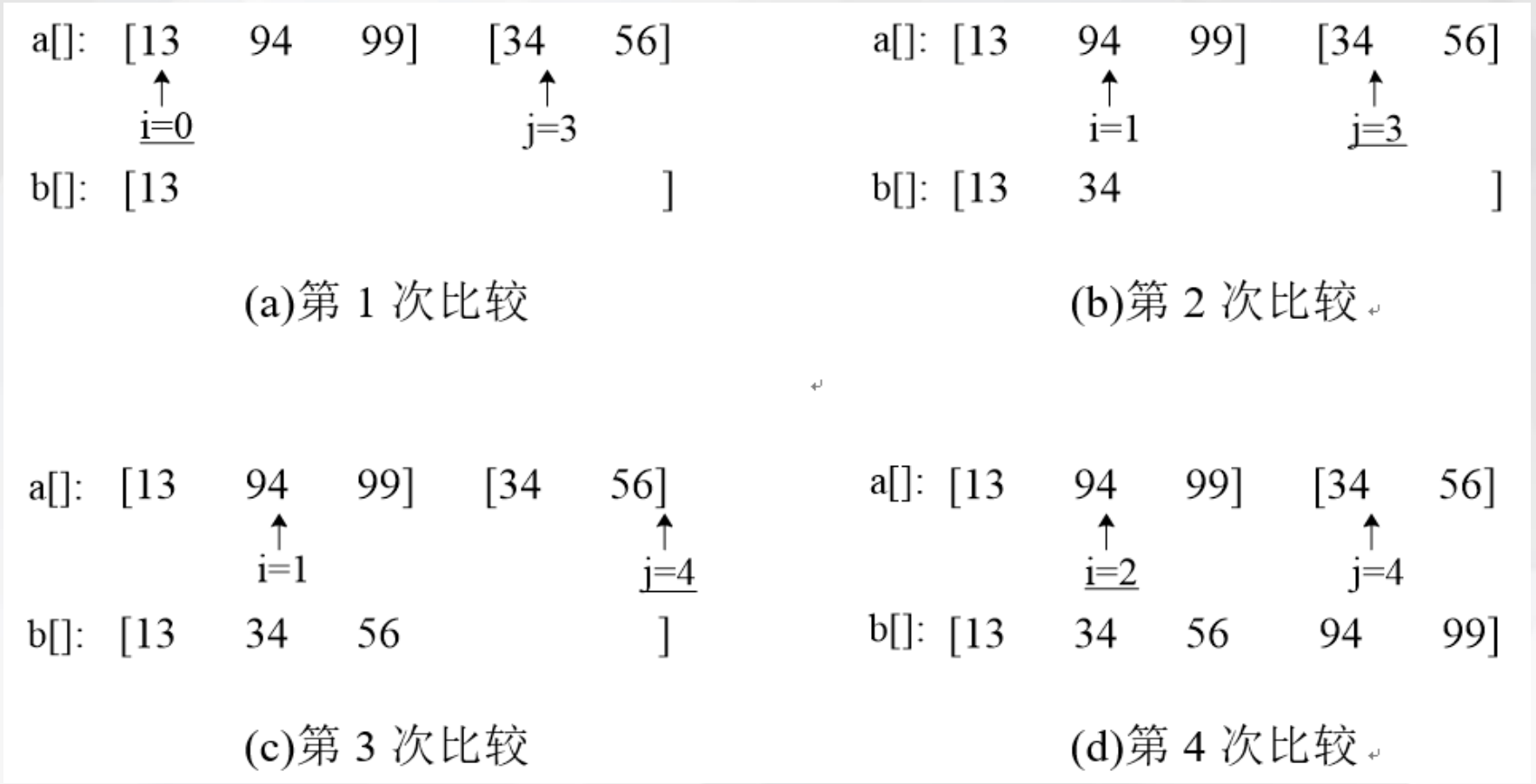
# 归并排序的主要操作

（1）分解。把初始序列分成长度相同的左右两个子序列，然后把每个子序列再分成更小的两个子序列……，直到子序列只包含1个数。用递归实现。

（2）求解子问题，对子序列排序。最底层的子序列只包含1个数，其实不用排序。

（3）合并。归并2个有序的子序列，这是归并排序的主要操作。

# 归并的例子


a[]: [13 94 99] [34 56]
i=0
j=3
b[]: [13 ]
(a)第 1 次比较
a[]: [13 94 99] [34 56]
i=1
j=3
b[]: [13 34 ]
(b)第 2 次比较
a[]: [13 94 99] [34 56]
i=1
j=4
b[]: [13 34 56 ]
(c)第 3 次比较
a[]: [13 94 99] [34 56]
i=2
j=4
b[]: [13 34 56 94 99]
(d)第 4 次比较

# 复杂度

- **对n个数进行归并排序：**

  （1）需要logn趟归并；

  （2）在每一趟归并中，有很多次合并操作，一共需要0(n)次比较。

  所以计算复杂度是0(nlogn)。

- **空间复杂度：需要一个临时的b[]存储结果，所以空间复杂度是O(n)。**

# 分治的优势

- 从归并排序的例子，读者可以体会到，对于整体上O(n)复杂度的问题，通过分治，可以减少为O(logn)复杂度的问题。

# 归并排序是交换排序的升级版

- 交换排序的步骤：

（1）第一轮，检查第一个数a1。把序列中后面所有的数，一个一个跟它比较，如果发现有一个比a1小，就交换。第一轮结束后，最小的数就排在了第一个位置。

（2）第二轮，检查第二个数。第二轮结束后，第2小的数排在了第二个位置。

（3）继续上述过程，直到检查完最后一个数。

- 交换排序的复杂度是$O(n^2)$。
- 在归并排序中，一次合并的操作，和交换排序很相似。只是合并的操作，是基于2个有序的子序列，效率更高。

# 练习题：逆序对问题

- 输入一个序列{a1, a2, a3,..., an}，交换任意两个相邻元素，不超过k次。交换之后，问最少的逆序对有多少个。
- 序列中的一个逆序对，是指存在两个数ai和aj，有ai > aj且1≤i<j≤n。也就是说，大的数排在小的数前面。

输入：第一行是n和k，1 ≤ n ≤ 105，0 ≤ k ≤ 109；第二行包括n个整数{a1, a2, a3,..., an}，0≤ ai ≤109。

输出：最少的逆序对数量。

Sample Input:

3 1

2 2 1

Sample Output:

1

- 当k = 0时，就是求原始序列中有多少个逆序对。
- 求k = 0时的逆序对，用暴力法很容易：先检查第一个数a1，把后面所有数跟它比较，如果发现有一个比a1小，就是一个逆序对；再检查第二个数，第三个数……；直到最后一个数。复杂度是$O(n^2)$。本题n最大是$10^5$，所以暴力法会TLE。

- 考察暴力法的过程，会发现和交换排序很像。那么自然可以想到，能否用交换排序的升级版归并排序，来处理逆序对问题?

- 请课后自己搜资料，解决hdu 4911。

# 快速排序

- 把序列分成左右两部分，使得左边所有的数都比右边的数小；
- 递归这个过程，直到不能再分为止。

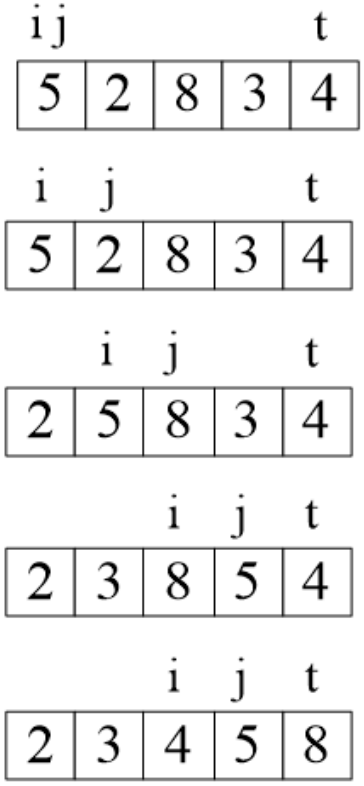
# 如何把序列分成左右两部分?

最简单的办法是:

- 设定两个临时空间X、Y和一个基准数t;
- 检查序列中所有的元素，比t小的放在X中，比t大的放在Y中。

- 不过，其实不用这么麻烦，直接在原序列上操作就行了，不需要X、Y。

# 直接在原序列上操作




尾部的 t 是基准数，i 指向比 t 小的左部分，j 指向比 t 大的右部分。

若 data[j] >= data[t]，j++。

若 data[j] < data[t]，交换 data[j]和 data[i]，然后 i++，j++。

继续。

最后，交换 data[i]和 data[t]，得到结果。i 指向基准数的当前位置。

# 复杂度

- 每一次划分，都把序列分成了左右两部分，在这个过程中，需要比较所有的元素，有O(n)次。
- 如果每次划分是对称的，左右两部分的长度差不多，那么一共需要划分O(logn)次。
- 总复杂度是O(nlogn)。

# 快速排序　不稳定

- 如果划分不是对称的，左部分和右部分的数量差别很大。在极端情况下，例如左部分只有1个数，剩下的全部都在右部分，那么最多可能划分n次，总复杂度是$O(n^2)$。
- 所以，快速排序是不稳定的。

# 但是…


- 一般情况下快速排序效率很高，比稳定的归并排序更好。
- 可以观察到，快速排序的代码比归并排序的代码简洁，代码中的比较、交换、拷贝操作很少。
- 快速排序几乎是目前所有排序法中速度最快的方法。STL的sort()函数就是基于快速排序算法的，并针对快速排序的缺点做了很多优化。

# 快速排序的应用：求第k大数

- 求第k大的数，简单的方法是排序算法进行排序，然后定位第k大的数，复杂度是O(nlogn)。
- 如果用快速排序的思想，可以在O(n)的时间内找到第k大的数 。在快速排序程序中，每次划分的时候，只要递归包含第k个数的那部分就行了。

# 习题

- hdu 1425，求前k大数；
- poj 2388，求中间数。