排序查找

七月算法 邹博

2015年11月1日

主要内容与目标

- 排序
  - 找到一个O(NlogN)的排序算法
  - 插入排序、选择排序、希尔排序、冒泡排序
  - 堆排序及其思考
  - 快速排序及其思考
  - 非比较方案的排序：记数排序、桶排序、基数排序
- 总结与思考
  - 排序的目的是什么？

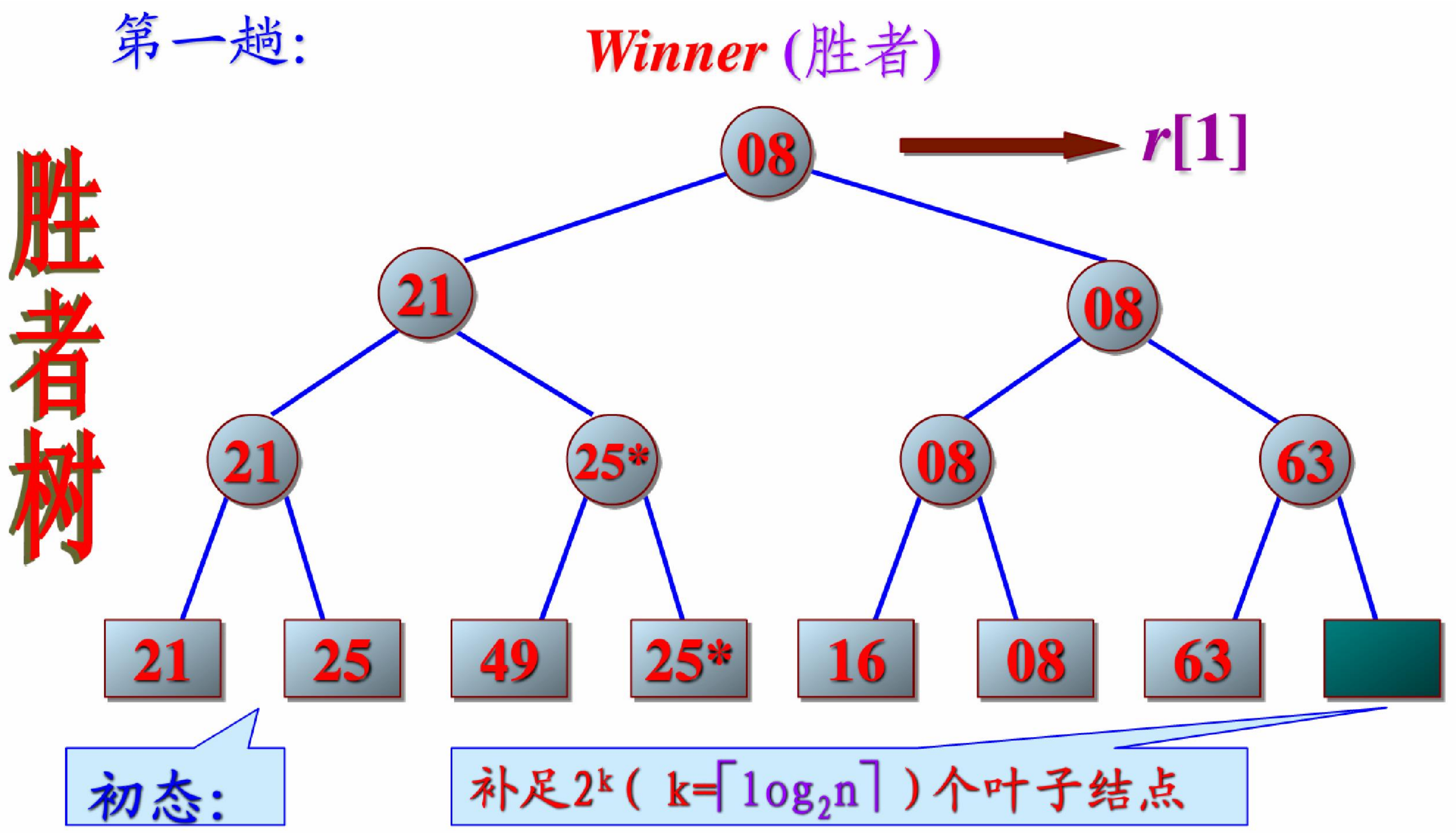
排序问题的提法

- 给定n个元素的集合A，按照某种方法将A中的元素按非降或非增次序排列。
- 分类：内排序，外排序
- 常见内排序方法
  - 插入排序/希尔排序
  - 选择排序/锦标赛排序/堆排序
  - 冒泡排序/快速排序
  - 归并排序
  - 基数排序

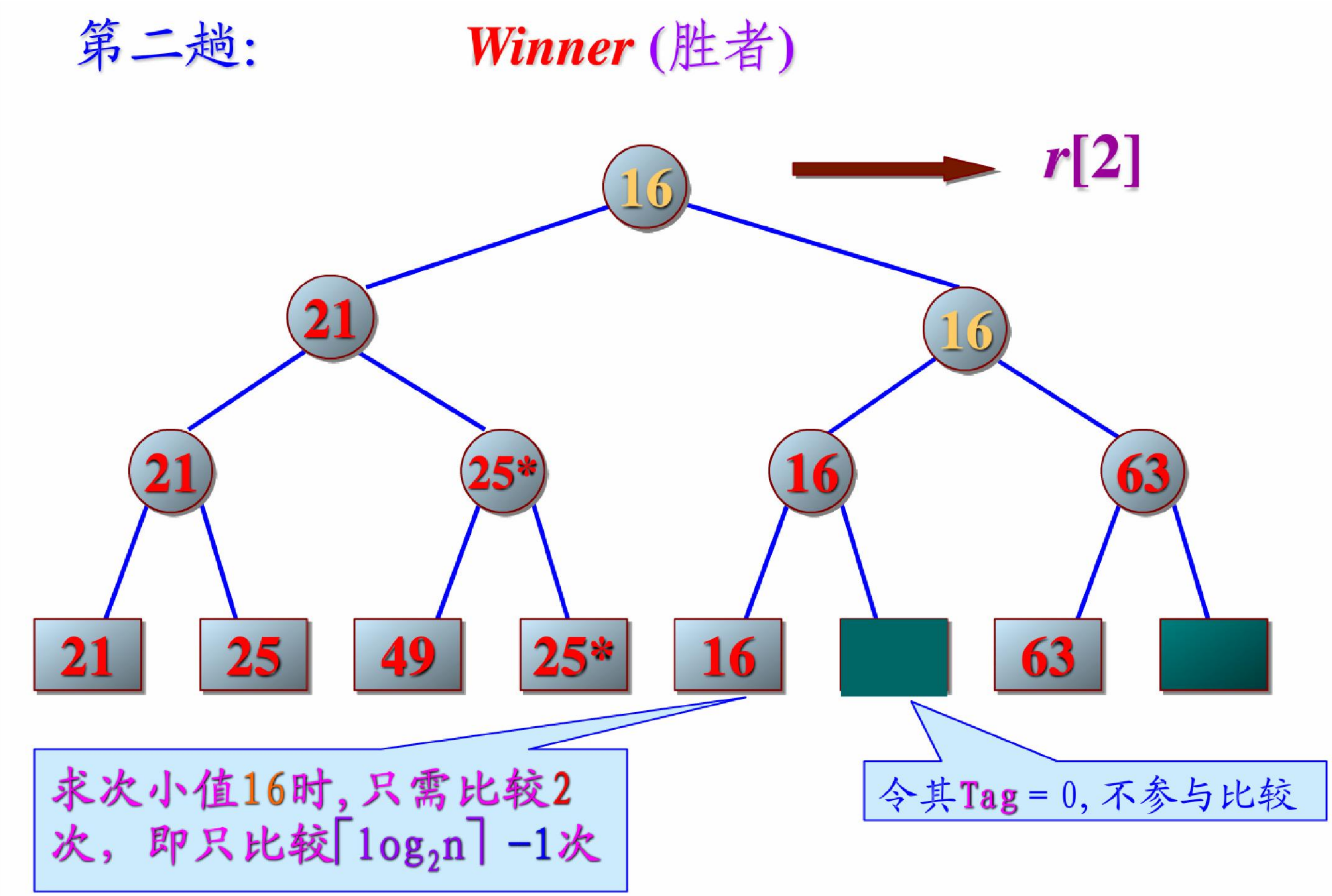
插入排序

- //将A(1：n)中的元素按非降次序分类，n≥1
- procedure INSERTIONSORT(A,n)
  - A(0)←-∞//设置初始边界值
  - for j←2 to n do //A(1:j-1)已分类
    - item←A(j);i←j-1
    - while item<A(i) do //0≤i<j
      - A(i+1)←A(i); i←i-1
    - repeat
    - A(i+1) ←item;
  - repeat
- end INSERTIONSORT

锦标赛排序


第一趟:
Winner (胜者)
r[1]
胜者树
08
21
08
21
25*
08
63
21
25
49
25*
16
08
63
初态:
补足2^k(k=⌈$\log_2 n$⌉)个叶子结点

锦标赛排序



第二趟:
Winner (胜者)
r[2]
16
21
16
21
25*
16
63
21
25
49
25*
16
63
求次小值16时，只需比较2次，即只比较⌈log₂n⌉ -1次
令其Tag = 0，不参与比较

归并排序

- 基本设计思想：将原始数组A[0…n-1]中的元素分成两个子数组：A1[0,n/2]和A2[n/2+1,n-1]。分别对这两个子数组单独排序，然后将已排序的两个子数组归并成一个含有n个元素的有序数组。

Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    int a[] = {3,56,2,7,45,8,1};
    int size = sizeof(a) / sizeof(int);
    MergeSort(a, 0, size-1);
    Print(a, size);
    return 0;
}
```

```
int temp[100];
void Merge(int* a, int low, int mid, int high)
{
    int i = low;
    int j = mid+1;
    int size = 0;
    for(;(i <= mid) && (j <= high); size++)
    {
        if(a[i] < a[j])
            temp[size] = a[i++];
        else
            temp[size] = a[j++];
    }
    while(i <= mid)
        temp[size++] = a[i++];
    while(j <= high)
        temp[size++] = a[j++];

    for(i = 0; i < size; i++)
        a[low+i] = temp[i];
}

void MergeSort(int* a, int low, int high)
{
    if(low >= high)
        return;

    int mid = (low + high) / 2;
    MergeSort(a, low, mid);
    MergeSort(a, mid+1, high);
    Merge(a, low, mid, high);
}
```

归并排序的时间复杂度性能分析

- 算法的递推关系：

$$T(n)=2\cdot T\left(\frac{n}{2}\right)+c\cdot n$$ ，c为常数

- 若 $n=2^k$ ，则有：

$$\begin{aligned}
T(n)&=2\cdot T\left(\frac{n}{2}\right)+c\cdot n\\
&=2\cdot\left(2\cdot T\left(\frac{n}{4}\right)+c\cdot\frac{n}{2}\right)+c\cdot n=4T\left(\frac{n}{4}\right)+2c\cdot n\\
&=4\left(2\cdot T\left(\frac{n}{8}\right)+c\cdot\frac{n}{4}\right)+2c\cdot n=8T\left(\frac{n}{8}\right)+3c\cdot n\\
&=8\left(2\cdot T\left(\frac{n}{16}\right)+c\cdot\frac{n}{8}\right)+3c\cdot n=16T\left(\frac{n}{16}\right)+4c\cdot n\\
&=\cdots\cdots\\
&=2^kT(1)+kc\cdot n=an+cn\log_2 n
\end{aligned}$$

- 若 $2^k<n<2^{k+1}$，则 $T(2^k) < T(n) < T(2^{k+1})$
- 所以得：$T(n) = O(n\log n)$.

归并排序的两点改进

- 在数组长度比较短的情况下，不进行递归，而是选择其他排序方案：如插入排序；
- 归并过程中，可以用记录数组下标的方式代替申请新内存空间；从而避免A和辅助数组间的频繁数据移动。

- *注：基于关键字比较的排序算法的平均时间复杂度的下界为O(nlogn)*

外排序

- 外排序(External sorting)是指能够处理极大量数据的排序算法。通常来说，外排序处理的数据不能一次装入内存，只能放在读写较慢的外存储器(通常是硬盘)上。外排序通常采用的是一种“排序-归并”的策略。在排序阶段，先读入能放在内存中的数据量，将其排序输出到一个临时文件，依此进行，将待排序数据组织为多个有序的临时文件。尔后在归并阶段将这些临时文件组合为一个大的有序文件，也即排序结果。

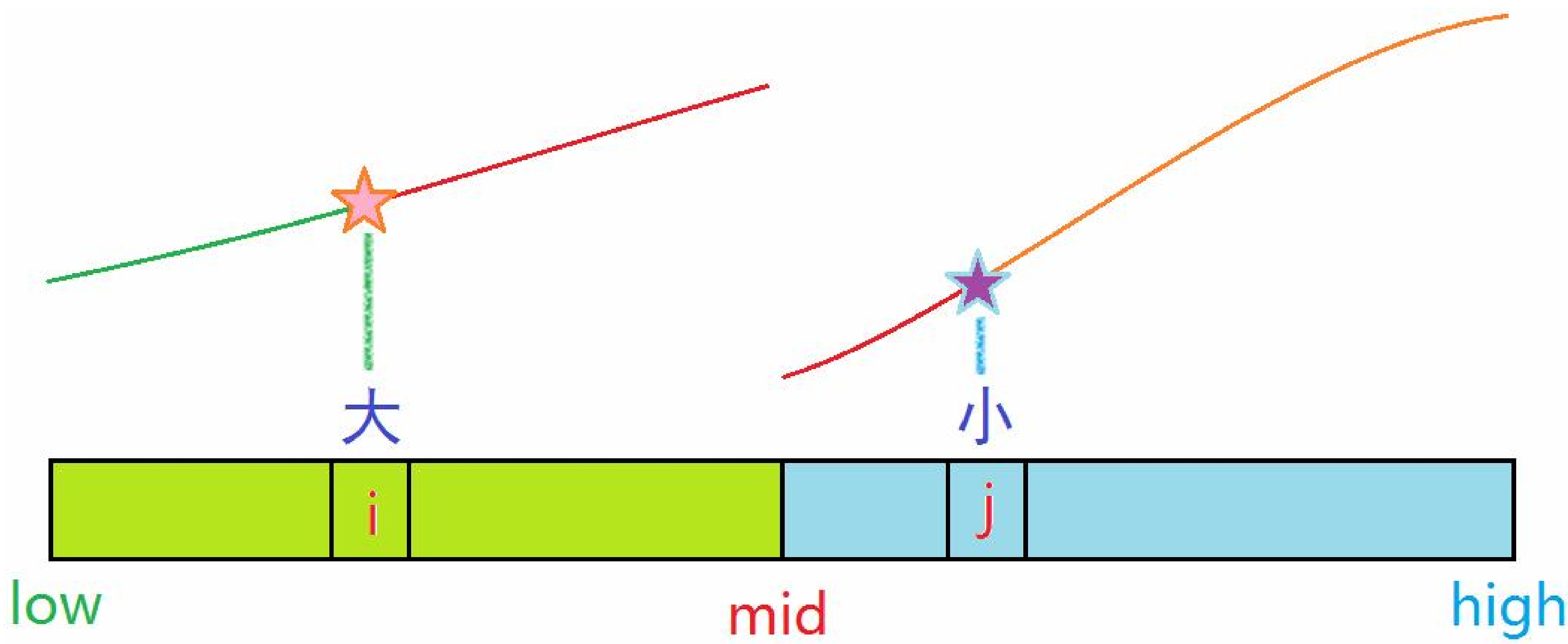
外排序举例

- 外归并排序(External merge sort)，它读入一些能放在内存内的数据量，在内存中排序后输出为一个顺串(即是内部数据有序的临时文件)，处理完所有的数据后再进行归并。
- 对900MB的数据进行排序，但机器上只有100MB的可用内存时，外归并排序按如下方法操作：
  - 读入100MB的数据至内存中，用某种常规方式(如快速排序、堆排序、归并排序等方法)在内存中完成排序。
  - 将排序完成的数据写入磁盘。
  - 重复步骤1和2直到所有的数据都存入了不同的100MB的块(临时文件)中。在这个例子中，有900MB数据，单个临时文件大小为100MB，所以会产生9个临时文件。
  - 读入每个临时文件(顺串)的前10MB(=100MB/(9块+1))的数据放入内存中的输入缓冲区，最后的10MB作为输出缓冲区。(实践中，将输入缓冲适当调小，而适当增大输出缓冲区能获得更好的效果。)
  - 执行九路归并算法，将结果输出到输出缓冲区。一旦输出缓冲区满，将缓冲区中的数据写出至目标文件，清空缓冲区。一旦9个输入缓冲区中的一个变空，就从这个缓冲区关联的文件，读入下一个10M数据，除非这个文件已读完。

逆序数问题

- 给定一个数组A[0…N-1]，若对于某两个元素a[i]、a[j]，若i < j且a[i] > a[j]，则称(a[i],a[j])为逆序对。一个数组中包含的逆序对的数目称为该数组的逆序数。试设计算法，求一个数组的逆序数。
  - 如：3,56,2,7的逆序数为3。

算法分析



Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    int a[] = {3,56,2,7,45,8,1};
    int size = sizeof(a) / sizeof(int);
    int count = 0;
    MergeSort(a, 0, size-1, count);
    cout << count << endl;
    return 0;
}
```

```
int temp[100];
void Merge(int* a, int low, int mid, int high, int& count)
{
    int i = low;
    int j = mid+1;
    int size = 0;
    for(;(i <= mid) && (j <= high); size++)
    {
        if(a[i] < a[j])
        {
            temp[size] = a[i++];
        }
        else
        {
            count += (mid - i + 1);
            temp[size] = a[j++];
        }
    }
    while(i <= mid)
        temp[size++] = a[i++];
    while(j <= high)
        temp[size++] = a[j++];

    for(i = 0; i < size; i++)
        a[low+i] = temp[i];
}

void MergeSort(int* a, int low, int high, int& count)
{
    if(low >= high)
        return;

    int mid = (low + high) / 2;
    MergeSort(a, low, mid, count);
    MergeSort(a, mid+1, high, count);
    Merge(a, low, mid, high, count);
}
```

Code2

```
int temp[100];
void Merge(int* a, int low, int mid, int high, int& count)
{
    int i = low;
    int j = mid+1;
    int size = 0;
    for(;(i <= mid) && (j <= high); size++)
    {
        if(a[i] < a[j])
        {
            temp[size] = a[i++];
        }
        else
        {
            count += (j - mid);
            temp[size] = a[j++];
        }
    }
    while(i <= mid)
        temp[size++] = a[i++];
    while(j <= high)
        temp[size++] = a[j++];

    for(i = 0; i < size; i++)
        a[low+i] = temp[i];
}
```

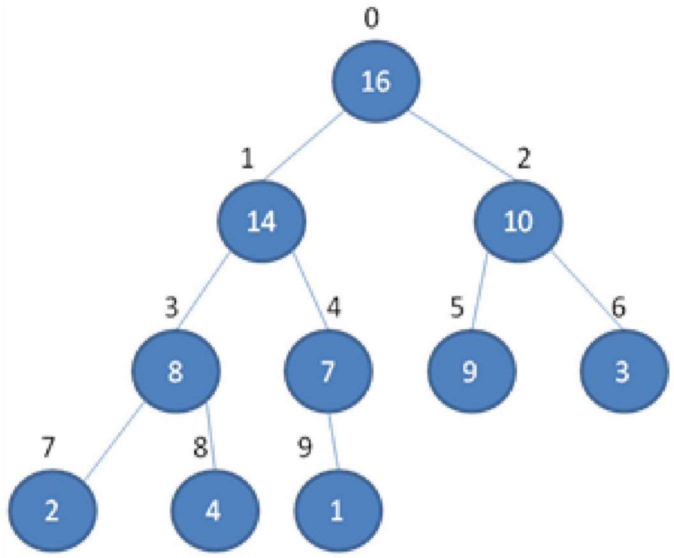
堆的定义和表示

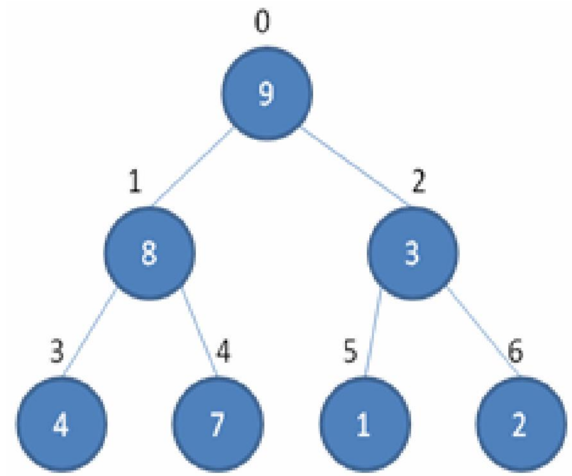
- 定义：对于一棵完全二叉树，若树中任一非叶子结点的关键字均不大于(或不小于)其左右孩子(若存在)结点的关键字，则这棵二叉树，叫做小顶堆(大顶堆)。
- 完全二叉树可以用数组完美存储，对于长度为n的数组a[0…n-1]，若
  - $\forall 0 \leqslant i \leqslant n-1$，$a[i] \leqslant a[2i+1]$ 且 $a[i] \leqslant a[2i+2]$

  那么，a表示一个小顶堆。
- 重要结论：大顶堆的堆顶元素是最大的。

堆的存储和树型表示

- 16,14,10,8,7,9,3,2,4,1
- 9,8,3,4,7,1,2





孩子与父亲的相互索引

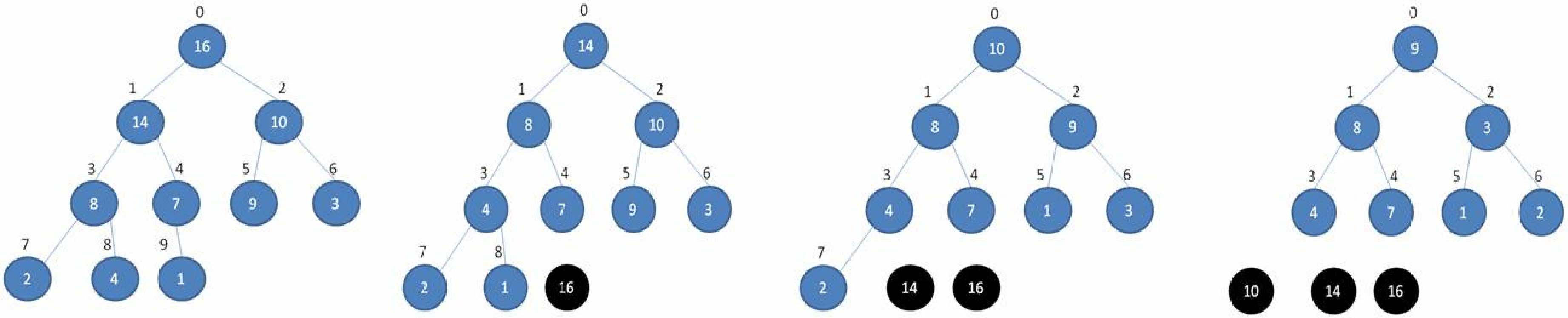
- k的孩子结点是2k+1,2k+2(如果存在)
- k的父结点：
  - 若k为左孩子，则k的父结点为k/2
  - 若k为右孩子，则k的父结点为(k/2) - 1
- 二者公式不一样，十分不便。发现：
  - 若k为左孩子，则k为奇数，则((k+1)/2)-1与k/2相等
  - 若k为右孩子，则k为偶数，则((k+1)/2)-1与(k/2) - 1相等
- 结论：若待考查结点为k，记k+1为K，则k的父结点为：(K/2) - 1

堆排序的整体思路

- ① 初始化操作：将a[0..n-1]构造为堆(如大顶堆)；
- ② 第i(n > i ≥ 1)趟排序：将堆顶记录a[0]和a[n-i] 交换，然后将a[0…n-i-1] 调整为堆(即：重建大顶堆)；
- ③进行n-1趟，完成排序。

- 堆排序的时间复杂度？
  - 初始化堆的过程：O(N)
    - 注意，一般教科书给出的O(NlogN)不是紧的。
  - 调整堆的过程：O(NlogN)

堆排序的调整过程

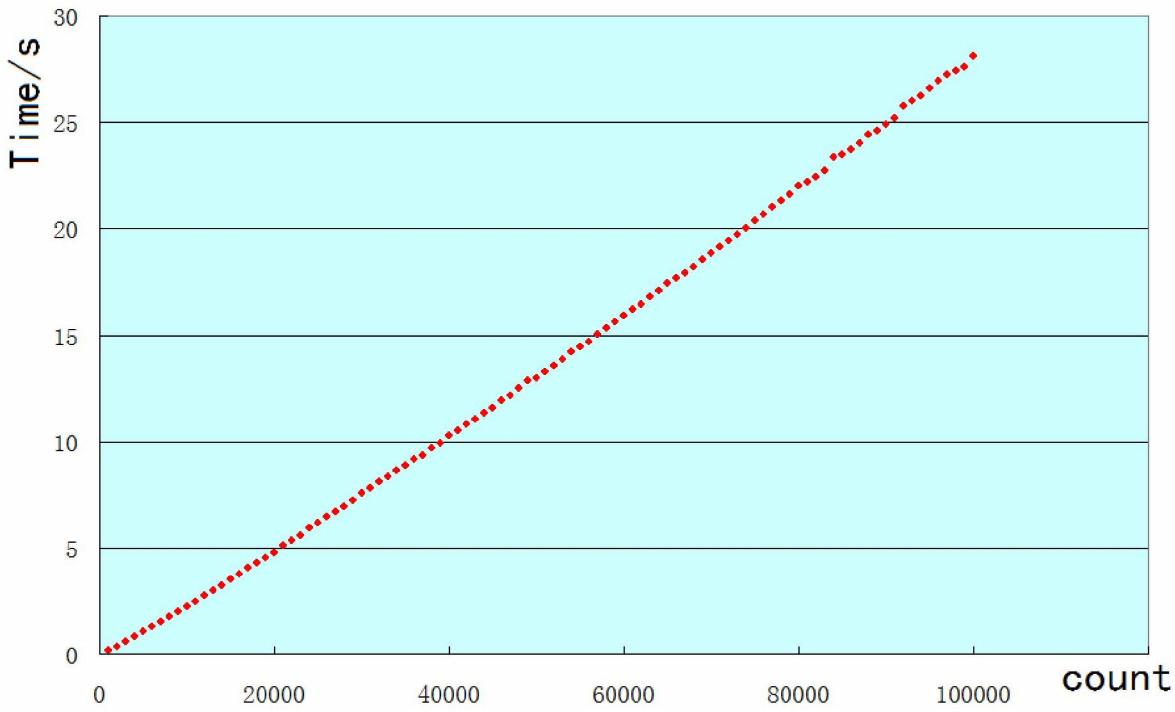


堆排序Code

```
//调用前，n的左右孩子都是大顶堆，调整以n为顶的堆为大顶堆
void HeapAjust(int* a, int n, int size)
{
    int nChild = 2*n+1; //左孩子
    int t;
    while(nChild < size)
    {
        if((nChild+1 < size) && (a[nChild+1] > a[nChild])) //找大孩子
            nChild++;
        if(a[nChild] < a[n])    //孩子比父亲小，说明调整完毕
            break;
        t = a[nChild];
        a[nChild] = a[n];
        a[n] = t;

        n = nChild;
        nChild = 2*n+1;
    }
}

void HeapSort(int* a, int size, int k)  //前k大的
{
    int i;
    for(i = size/2 - 1; i >= 0; i--)    //依次调整堆
        HeapAjust(a, i, size);

    int t;
    int s = size - k;
    while(size > s) //依次找到最大的并放置在数组末尾
    {
        t = a[size-1];
        a[size-1] = a[0];
        a[0] = t;
        size--;
        HeapAjust(a, 0, size);
    }
}
```

堆排序实际运行效率

```
int _tmain(int argc, _TCHAR* argv[])
{
    const int size = 100000;
    int a[size];
    int i;
    for(i = 0; i < size; i++)
        a[i] = i;
    random_shuffle(a, a+size);
    int b[size];
    memcpy(b, a, size*sizeof(int));

    for(int s = 1000; s <= size; s += 1000)
    {
        int dwStart = GetTickCount();
        for(i = 0; i < 1000; i++)
        {
            memcpy(a, b, s*sizeof(int));
            HeapSort(a, s);
        }
        int dwEnd = GetTickCount();
        cout << s << ":\t" << dwEnd - dwStart << endl;
    }
    Print(a, size);
    return 0;
}
```



N个数中，选择前k个最大的数

- 建立一个小顶堆，小顶堆的大小为k
- for 每个数
  - if 这个数比小顶堆的堆顶元素大
    - 弹出小顶堆的最小元素
    - 把这个数插入到小顶堆
- 小顶堆中的k个元素就是所要求的元素

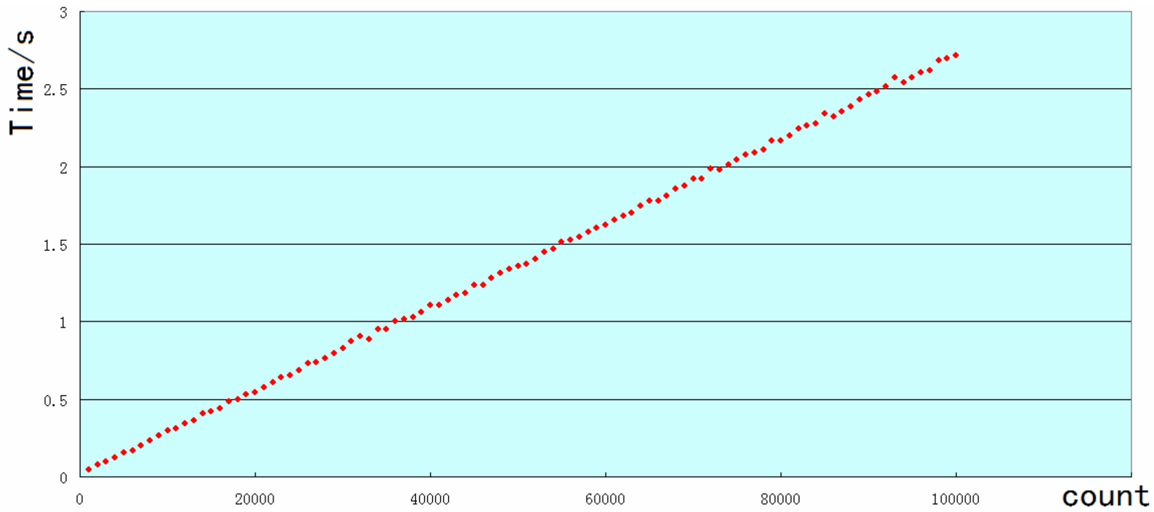
- 小顶堆的作用：
  - 保持始终有k个最大元素——利于最后的输出
  - k个元素中最小的元素在堆顶——利于后续元素的比较
- 时间复杂度：O(N*logk)

对比：选择前k个最大的数

- 算法描述：
  - 1、建立全部n个元素的**大顶堆**；
  - 2、利用堆排序，但得到前k个元素后即完成算法。
- 时间复杂度分析：
  - 1、建堆O(N)
  - 2、选择1个元素的时间是O(logN)，所以，第二步的总时间复杂度为O(klogN)
  - 该算法时间复杂度为O(N+klogN)
- 思考：
  - O(N+klogN)与O(N*logk)哪个更快？

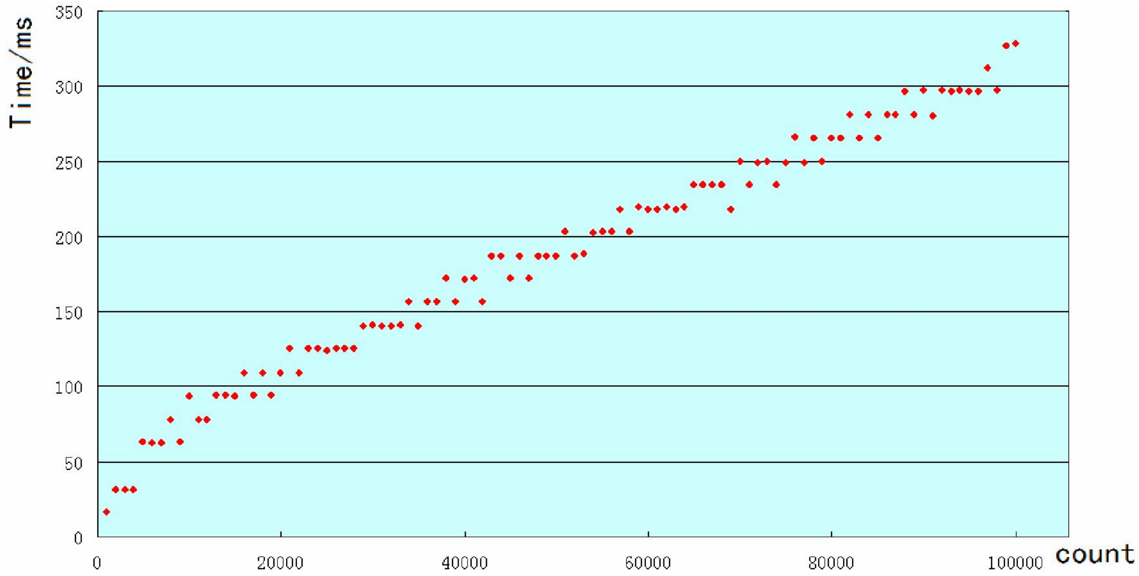
最大的k个数——算法2 Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    const int size = 100000;
    int a[size];
    int i;
    for(i = 0; i < size; i++)
        a[i] = i;
    random_shuffle(a, a+size);
    int b[size];
    memcpy(b, a, size*sizeof(int));

    int k = 100;
    for(int s = 1000; s <= size; s += 1000)
    {
        int dwStart = GetTickCount();
        for(i = 0; i < 1000; i++)
        {
            memcpy(a, b, s*sizeof(int));
            HeapSort(a, s, k);
        }
        int dwEnd = GetTickCount();
        cout << s << ":\t" << dwEnd - dwStart << endl;
    }
    Print(a, size);
    return 0;
}
```



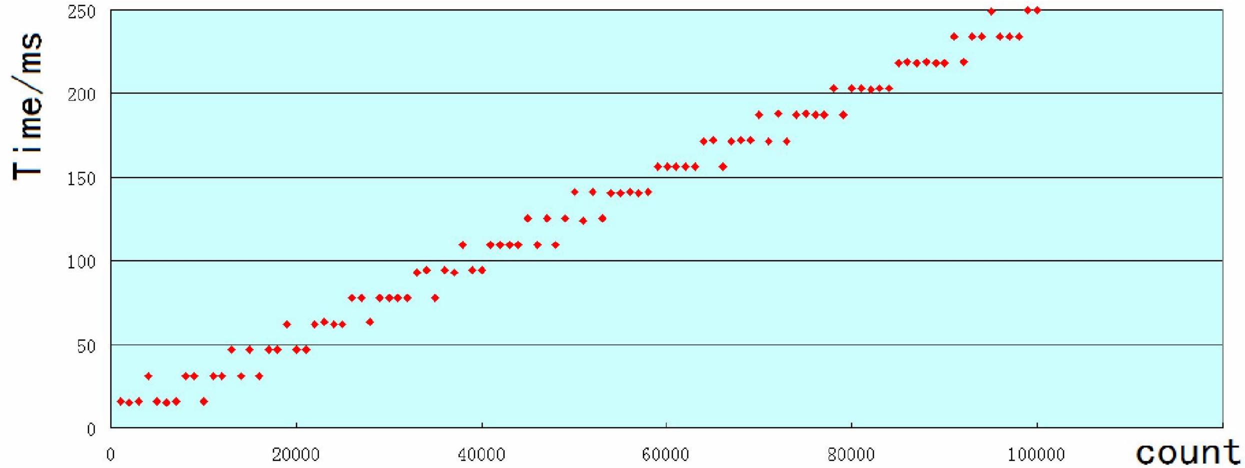
最大的k个数——算法1 Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    const int size = 100000;
    int a[size];
    int i;
    for(i = 0; i < size; i++)
        a[i] = i;
    random_shuffle(a, a+size);
    int b[size];
    memcpy(b, a, size*sizeof(int));

    int k = 100;
    int j;
    for(int s = 1000; s <= size; s += 1000)
    {
        int dwStart = GetTickCount();
        for(i = 0; i < 1000; i++)
        {
            memcpy(a, b, s*sizeof(int));
            HeapSort(a, k, k-1);    //前k个元素堆排序
            for(j = k+1; j < s; j++)
            {
                if(a[j] < a[0]) //新数小于堆顶元素，则新数入堆
                {
                    a[0] = a[j];
                    HeapAjust(a, 0, k);
                }
            }
        }
        int dwEnd = GetTickCount();
        cout << s << ":\t" << dwEnd - dwStart << endl;
    }
    return 0;
}
```



最大的k个数——算法1 Code’

```
int _tmain(int argc, _TCHAR* argv[])
{
    const int size = 100000;
    int a[size];
    int i;
    for(i = 0; i < size; i++)
        a[i] = i;
    random_shuffle(a, a+size);
    int b[size];
    memcpy(b, a, size*sizeof(int));

    int k = 10;
    int j;
    for(int s = 1000; s <= size; s += 1000)
    {
        int dwStart = GetTickCount();
        for(i = 0; i < 1000; i++)
        {
            memcpy(a, b, s*sizeof(int));
            HeapSort(a, k, k-1);    //前k个元素堆排序
            for(j = k+1; j < s; j++)
            {
                if(a[j] < a[0]) //新数小于堆顶元素，则新数入堆
                {
                    a[0] = a[j];
                    HeapAjust(a, 0, k);
                }
            }
        }
        int dwEnd = GetTickCount();
        cout << s << ":\t" << dwEnd - dwStart << endl;
    }
    return 0;
}
```



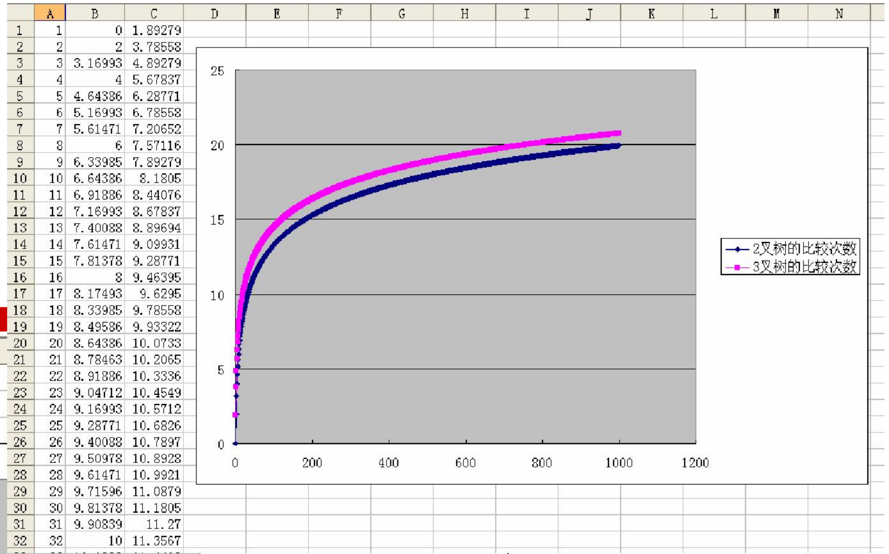
"求前K大的数"算法总结

- 实践证明，算法1的方案相对于算法2更优
- 事实上，有更快的BFPRT算法。
  - 这不能抹杀算法1和算法2在实际中的存在价值
  - 稍后马上介绍

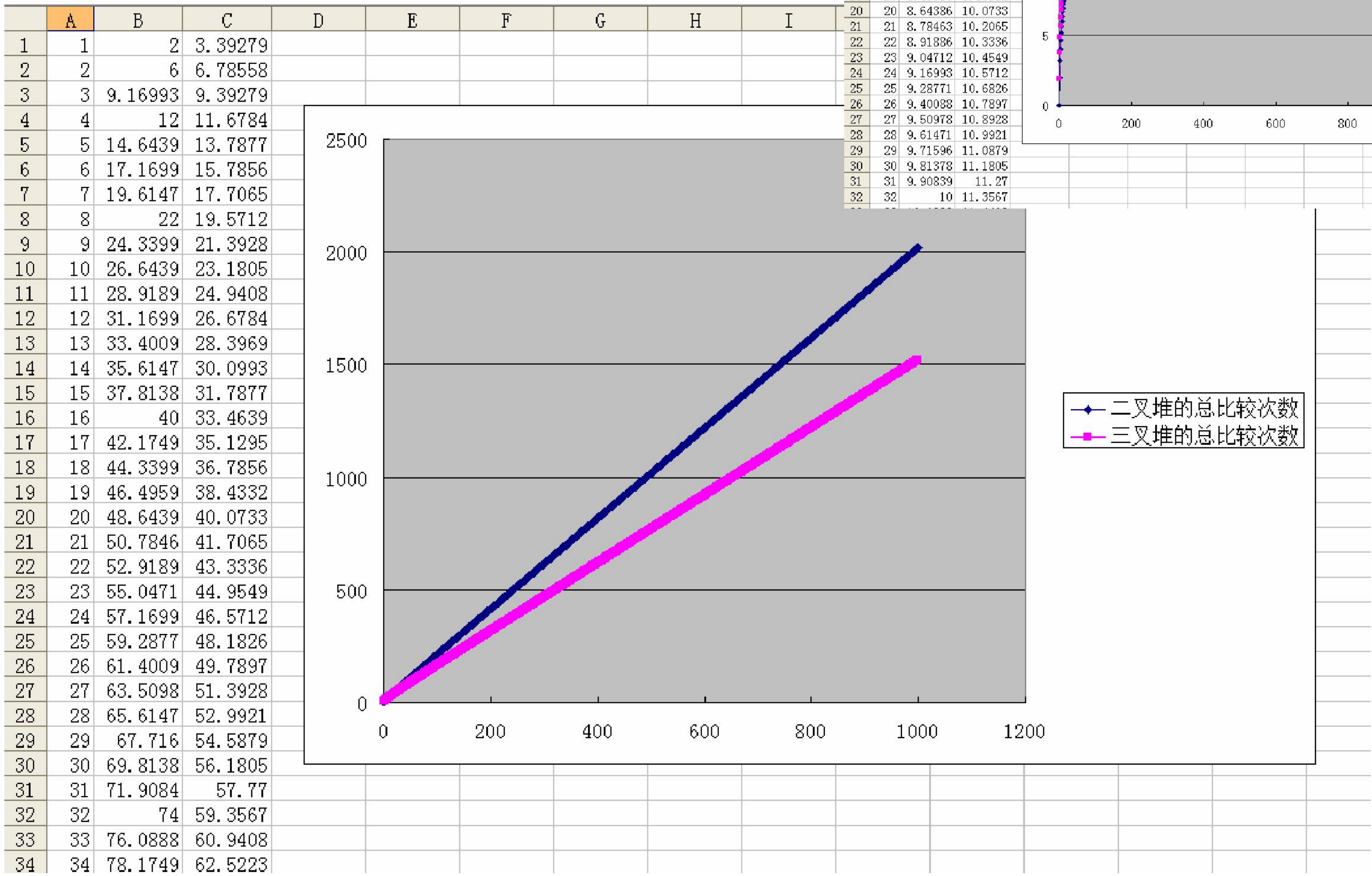
K叉堆的结论

- 对 n 个元素建立初始 K 叉堆的最多比较次数不超过(k/k-1)*n 次；
- 对 n 个元素的 K 叉堆，每次删去堆顶元素并调整使之恢复 K 叉堆，这样 m 次过程的最多比较次数不超过 $m*k*[\log_k((k-1)n]$ 次。

3叉堆与2叉堆



	A	B	C
1	1	0	1.89279
2	2	2	3.78558
3	3	3.16993	4.89279
4	4	4	5.67837
5	5	4.64386	6.28771
6	6	5.16993	6.78558
7	7	5.61471	7.20652
8	8	6	7.57116
9	9	6.33985	7.89279
10	10	6.64386	8.1805
11	11	6.91886	8.44076
12	12	7.16993	8.67837
13	13	7.40088	8.89694
14	14	7.61471	9.09931
15	15	7.81378	9.28771
16	16	8	9.46395
17	17	8.17493	9.6295
18	18	8.33985	9.78558
19	19	8.49586	9.93322
20	20	8.64386	10.0733
21	21	8.78463	10.2065
22	22	8.91886	10.3336
23	23	9.04712	10.4549
24	24	9.16993	10.5712
25	25	9.28771	10.6826
26	26	9.40088	10.7897
27	27	9.50978	10.8928
28	28	9.61471	10.9921
29	29	9.71596	11.0879
30	30	9.81378	11.1805
31	31	9.90839	11.27
32	32	10	11.3567



	A	B	C
1	1	2	3.39279
2	2	6	6.78558
3	3	9.16993	9.39279
4	4	12	11.6784
5	5	14.6439	13.7877
6	6	17.1699	15.7856
7	7	19.6147	17.7065
8	8	22	19.5712
9	9	24.3399	21.3928
10	10	26.6439	23.1805
11	11	28.9189	24.9408
12	12	31.1699	26.6784
13	13	33.4009	28.3969
14	14	35.6147	30.0993
15	15	37.8138	31.7877
16	16	40	33.4639
17	17	42.1749	35.1295
18	18	44.3399	36.7856
19	19	46.4959	38.4332
20	20	48.6439	40.0733
21	21	50.7846	41.7065
22	22	52.9189	43.3336
23	23	55.0471	44.9549
24	24	57.1699	46.5712
25	25	59.2877	48.1826
26	26	61.4009	49.7897
27	27	63.5098	51.3928
28	28	65.6147	52.9921
29	29	67.716	54.5879
30	30	69.8138	56.1805
31	31	71.9084	57.77
32	32	74	59.3567
33	33	76.0888	60.9408
34	34	78.1749	62.5223

堆排序中的思考

- 得到一个堆后，堆排序仅输出堆顶元素，便又重新组织新堆了，没有利用完全堆的全部信息。根据堆的逻辑结构和特征，堆顶结点的左右孩子之一必有一个是数据中的第二大(小)者，完全可以随着堆顶一起交换到末尾。然后，分别对次顶堆和顶堆调整即可。

稳定堆排序？

- 1、建堆的时候，相等则不调整；
- 2、调整堆的时候：
- 2.1 如果与根相等，与左右孩子不相等，则调整到孩子；
- 2.2 如果与根、左孩子都相等，与右孩子不等，则调整到左孩子这一支，递归考察2.1；
- 2.3 如果与根、右孩子都相等，则调整到右孩子这一支，递归考察2.1；
  - 此情况其实包含了根、左孩子、右孩子都相等的情况

稳定与非稳定

- 事实上，任何一个非稳定的排序，如果能够将元素值value与元素所在位置index共同排序，即可得到稳定的排序。

快速排序

- 快速排序是一种基于划分的排序方法；
- 划分Partitioning：选取待排序集合A中的某个元素t，按照与t的大小关系重新整理A中元素，使得整理后的序列中所有在t以前出现的元素均小于t，而所有出现在t以后的元素均大于等于t；元素t称为划分元素。
- 快速排序：通过反复地对A进行划分达到排序的目的。

划分算法

- 对于数组a[0…n-1]
  - 设置两个变量i、j：i=0，j=n-1；
  - 以a[0]作为关键数据，即key=A[0]；
  - 从j开始向前搜索，直到找到第一个小于key的值a[j]，将a[i] = a[j]；
  - 从i开始向后搜索，直到找到第一个大于等于于key的值a[i]，a[j] = a[i]；
  - 重复第3、4步，直到i≥j.

链表划分

- 给定一个链表和一个值x，将链表划分成两部分，使得划分后小于x的结点在前，大于等于x的结点在后。在这两部分中要保持原链表中的出现顺序。
  - 如：给定链表1->4->3->2->5->2和x = 3，返回1->2->2->4->3->5。

问题分析

- 分别申请两个指针p1和p2，小于x的添加到p1中，大于等于x的添加到p2中；最后，将p2链接到p1的末端即可。
- 时间复杂度是O(N)，空间复杂度为O(1)；该问题其实说明：快速排序对于单链表存储结构仍然适用。
  - 注：不是所有排序都方便使用链表存储，如堆排序，将不断的查找数组的n/2和n的位置，用链表做存储结构会不太方便。

Code

```
typedef struct tagSNode
{
    int value;
    tagSNode* pNext;

    tagSNode(int v) : value(v), pNext(NULL) {}
}SNode;
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    SNode* pHead = new SNode(0);
    pHead->pNext = NULL;
    for(int i = 0; i < 10; i++)
    {
        SNode* p = new SNode(rand() % 100);
        p->pNext = pHead->pNext;
        pHead->pNext = p;
    }
    Print(pHead);
    Partition(pHead, 50);
    Print(pHead);
    Destroy(pHead);
    return 0;
}
```

```
void Destroy(SNode* p)
{
    SNode* next;
    while(p)
    {
        next = p->pNext;
        delete p;
        p = next;
    }
}
```

```
void Partition(SNode* pHead, int pivotKey)
{
    //两个链表的头指针
    SNode* pLeftHead = new SNode(0);
    SNode* pRightHead = new SNode(0);

    //两个链表的当前最后一个元素
    SNode* left = pLeftHead;
    SNode* right = pRightHead;
    SNode* p = pHead->pNext;
    while(p)    //遍历原链表
    {
        if(p->value < pivotKey)
        {
            left->pNext = p;
            left = p;
        }
        else
        {
            right->pNext = p;
            right = p;
        }
        p = p->pNext;
    }

    //将right链接到left尾部
    left->pNext = pRightHead->pNext;
    right->pNext = NULL;

    //将整理好的链表赋值给当前链表头部
    pHead->pNext = pLeftHead->pNext;

    delete pLeftHead;
    delete pRightHead;
}
```

快速排序Code

```
void quick_sort(int s[], int l, int r)
{
    if (l < r)
    {
        //Swap(s[l], s[(l + r) / 2]); //将中间的这个数和第一个数交换
        int i = l, j = r, x = s[l];
        while (i < j)
        {
            while(i < j && s[j] >= x) // 从右向左找第一个小于x的数
                j--;
            if(i < j)
                s[i++] = s[j];

            while(i < j && s[i] < x) // 从左向右找第一个大于等于x的数
                i++;
            if(i < j)
                s[j--] = s[i];
        }
        s[i] = x;
        quick_sort(s, l, i - 1); // 递归调用
        quick_sort(s, i + 1, r);
    }
}
```

快速排序与归并排序的联系

- 都是分治的思想；
- 经过一次划分后，实现了对A的调整：其中一个子集合的所有元素均小于等于另外一个子集合的所有元素；
- 按同样的策略对两个子集合进行分类处理。当子集合分类完毕后，整个集合的分类也完成了。这一过程避免了子集合的归并操作。

快速排序的性能分析

- 在最好的情况，每次运行一次分区，我们会把一个数列分为两个几近相等的片段。然后，递归调用两个一半大小的数列。
- 一次分区中，i、j一共遍历了n个数，即O(n)
- 记：快速排序的时间复杂度为T(n)，有，
  - $T(n) = 2*T(n/2) + cn$ c是某常数
- $T(n)=O(n*logn)$

快速排序的性能分析

- 在最坏的情况下，两个子数组的长度为 1 和 n-1
- $T(n) = T(1) + T(n - 1) + cn$
- 演示：计算得到 $T(n) = O(n^2)$

- 思考：如果每次分区，都把数组分成1%和99%的两个子数组，时间复杂度是多少？

附：根据前序中序，计算后序

- 前序遍历：GDAFEMHZ
- 中序遍历：ADEFGHMZ
- 根据前序遍历的特点得知，根结点为G；
- 根结点将中序遍历结果ADEFGHMZ分成ADEF和HMZ两个左子树、右子树。
- 递归确定中序遍历序列ADEF和前序遍历序列DAEF的子树结构；
- 递归确定中序遍历序列HMZ和前序遍历序列MHZ的子树结构；

Code——问：时间复杂度是多少？

```
void InPre2Post(const char* pInOrder, const char* pPreOrder, int nLength, char* pPostOrder, int& nIndex)
{
    if(nLength <= 0)
        return;
    if(nLength == 1)
    {
        pPostOrder[nIndex] = *pPreOrder;
        nIndex++;
        return;
    }
    char root = *pPreOrder;
    int nRoot = 0;
    for(;nRoot < nLength; nRoot++)
    {
        if(pInOrder[nRoot] == root)
            break;
    }
    InPre2Post(pInOrder, pPreOrder+1, nRoot, pPostOrder, nIndex);
    InPre2Post(pInOrder+nRoot+1, pPreOrder+nRoot+1, nLength-(nRoot+1), pPostOrder, nIndex);
    pPostOrder[nIndex] = root;
    nIndex++;
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    char pPreOrder[] = "GDAFEMHZ";
    char pInOrder[] = "ADEFGHMZ";
    int size = sizeof(pInOrder) / sizeof(char);
    char* pPostOrder = new char[size];
    int nIndex = 0;
    InPre2Post(pInOrder, pPreOrder, size-1, pPostOrder, nIndex);
    pPostOrder[size-1] = 0;
    cout << pPostOrder << endl;
    delete[] pPostOrder;
    return 0;
}
```

Heap VS Quick

- 快速排序的最直接竞争者是堆排序。堆排序通常比快速排序稍微慢，但是最坏情况的运行时间总是O(n log n)。快速排序是经常比较快，但仍然有最坏情况性能的机会。
- 堆排序拥有重要的特点：仅使用固定额外的空间，即堆排序是原地排序，而快速排序需要O(log n)的空间。

快速排序为什么这么快？

- 乱数快速排序有一个值得注意的特性，在任意输入数据的状况下，它只需要O(n log n)的期望时间。是什么让随机的基准变成一个好的选择？
- 假设我们排序一个数列，然后把它分为四个部份。在中央的两个部份将会包含最好的基准值；他们的每一个至少都会比25%的元素大，且至少比25%的元素小。如果我们可以一致地从这两个中央的部份选出一个元素，在到达大小为1的数列前，我们可能最多仅需要把数列分区$2\log_2 n$次，产生一个O(nlogn)算法。
- 不幸地，乱数选择只有一半的时间会从中间的部份选择。出人意外的事实是这样就已经足够好了。想像你正在投掷一枚硬币，直到有 k 次国徽朝上。尽管这需要很长的时间，平均来说只需要 2k 次投掷。且在 100k 次投掷中得不到 k 次国徽朝上的概率，是像天文数字一样的非常小[注]。借由同样的论证，快速排序的递归平均只要$2(2\log_2 n)$的调用深度就会终止。
  - 注：该概率小于7.9E-31
- 如果它的平均调用深度是O(log n)且每一阶的调用树状过程最多有 n 个元素，则全部完成的工作量就是 O(n log n)。

BFPRT算法

- 题目：求第k大的数，如何解决？
  - 得到了前k大的数，显然顺便得到第k大的数，即：该问题至少存在O(Nlogk)的算法
  - 事实上，通过快速排序的Partition思想，第k大的数可以在期望是O(N)的算法内解决
    - 最坏情况是$O(N^2)$，但可以使用二次取中的办法避免最坏情况的发生
  - 借鉴第k大的数的思想，如何解决前k大的数
    - Partition之后的前面的k个即为所求。
    - *Blum*、*Floyd*、*Pratt*、*Rivest*、*Tarjan*

n个数中，选择第k大的数

- 数组a[0...n-1]，选择第k大的数
- 利用快速排序的思想，随机选择划分元素t，将数组分成大于t和小于等于t两部分。记为a[0...m-1]和a[m+1...n-1]，若m=k-1，则t即为所求；若m>k-1，则递归计算a[0...m-1]中第k大的数；若m>k-1，则递归计算a[m+1...n-1]中第k-m大的数。
- 平均时间复杂度O(n)，最差O(n^2)。
  - 快排的时候,左右两个分支都要进行递归，找k大的时候只需要对其中一边进行递归。
- 可使用"二次取中"的规则得到最坏情况是O(n)的算法。

如果遇到相等的数，怎么处理

- 数组中M出现次数很多，而恰好选了M作为PivotKey，那么，将导致Partition之后，一部分很长，一部分很短。(比如：极限情况：数组中都是M，划分后，一部分是整体本身，一部分为0)
- 数据分成“大于M、小于M、等于M”三部分，可类比荷兰国旗问题。

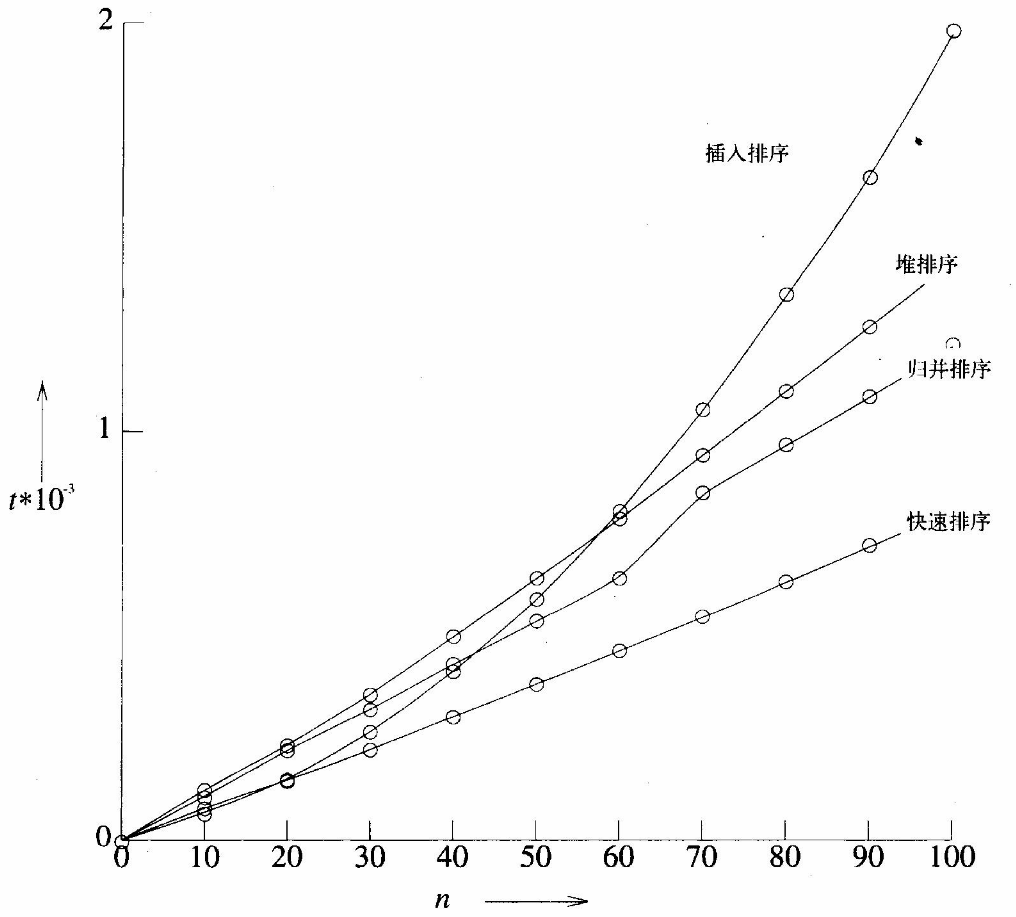
考虑相等元素的O(N)时间选择算法

```
select(L,k)
{
if (L has 10 or fewer elements)
{
    sort L
    return the element in the kth position
}

partition L into subsets S[i] of five elements each
    (there will be n/5 subsets total).

for (i = 1 to n/5) do
    x[i] = select(S[i],3)

M = select({x[i]}, n/10)

partition L into L1<M, L2=M, L3>M
if (k <= length(L1))
    return select(L1,k)
else if (k > length(L1)+length(L2))
    return select(L3,k-length(L1)-length(L2))
else return M
}
```

各种排序算法的时间复杂度

排序方法	最好时间	平均时间	最坏时间	辅助空间	稳定性
直接插入	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
二分插入	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
希 尔		$O(n^{1.25})$		$O(1)$	不稳定
冒 泡	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
快 速	$O(nlgn)$	$O(nlgn)$	$O(n^2)$	$O(lgn)$	不稳定
直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆	$O(nlgn)$	$O(nlgn)$	$O(nlgn)$		不稳定
归 并	$O(nlgn)$	$O(nlgn)$	$O(nlgn)$	$O(n)$	稳定
基 数	$O(d(r+n))$	$O(d(r+n))$	$O(d(r+n))$	$O(rd+n)$	稳定

排序算法效率比较



注：该数据来自网络，可信度低

稳定性

- 一般的说，如果排序过程中，只有相邻元素进行比较，是稳定的，如冒泡排序、归并排序；如果间隔元素进行了比较，往往是非稳定的，如堆排序、快速排序。
  - 归并排序是指针逐次后移，姑且算相邻元素的比较
  - 直接插入排序可以将新增数据放在排序的相等数据的后面，使得直接插入排序是稳定的；但二分插入排序本身不稳定，如果要稳定，需要向后探测
- 一般的说，如果能够方便整理数据，对于不稳定的排序，可以使用(A[i],i)键对来进行算法，可以使得不稳定排序变成稳定排序。

计数排序

- 计数排序的核心思想，是用空间换取时间，本质是建立了基于元素的Hash表。

计数排序

A 数组存储原始数据

	0	1	2	3	4	5	6	7
A:	2	5	3	0	2	3	0	3

C 数组是辅助数组。K=5,则 C 大小为 6。C 数组初始化

	0	1	2	3	4	5
C:	0	0	0	0	0	0

现在 C 数组用作：统计 A 数组中，值为 i 的元素的个数

	0	1	2	3	4	5
C:	2	0	2	3	0	1

现在 C 数组用作：统计 A 数组中，小于等于 i 的元素个数

	0	1	2	3	4	5
C:	2	2	4	7	7	8

现在开始执行最后一个循环：

当 i=7 时，A[7]=3,C[3]=7;B[7-1]=3,C[3]=7-1=6,此时

	0	1	2	3	4	5
C:	2	2	4	6	7	8

	0	1	2	3	4	5	6	7
B:							3	

桶排序/基数排序

- 将元素分到若干个桶中，每个桶分别排序，然后归并
- 由于桶之间往往是有序的(如：洗牌中的1-13个点数，整数按照数位0-9基数排序等)，所以，它们的时间复杂度不是(完全)基于比较的，时间复杂度下限不是O(NlogN)
- 如果桶的个数和待排序数目相同，则退化为记数排序。
  - ——每个桶内只有1个元素
- 思考：如果每个桶内最多有2个元素呢?
  - 求给定N个数的最大间距，要求O(N)的时间复杂度
  - 如：8,3,17,6,14, 4的最大间距为6

附：最大间隔

- 给定整数数组A[0…N-1]，求这N个数排序后最大间隔。如：1,7,14,9,4,13的最大间隔为4。
  - 排序后：1,4,7,9,13,14，最大间隔是13-9=4
  - 显然，对原数组排序，然后求后项减前项的最大值，即为解。
  - 可否有更好的方法？

附： Code

```
typedef struct tagSBucket
{
    bool bValid;
    int nMin;
    int nMax;

    tagSBucket() : bValid(false) {}

    void Add(int n) //将数n加入到桶中
    {
        if(!bValid)
        {
            nMin = nMax = n;
            bValid = true;
        }
        else
        {
            if(nMax < n)
                nMax = n;
            else if(nMin > n)
                nMin = n;
        }
    }
}SBucket;
```

```
int CalcMaxGap(const int* A, int size)
{
    //求最值
    SBucket* pBucket = new SBucket[size];
    int nMax = A[0];
    int nMin = A[0];
    int i;
    for(i = 1; i < size; i++)
    {
        if(nMax < A[i])
            nMax = A[i];
        else if(nMin > A[i])
            nMin = A[i];
    }

    //依次将数据放入桶中
    int delta = nMax - nMin;
    int nBucket;    //某数应该在哪个桶中
    for(i = 0; i < size; i++)
    {
        nBucket = (A[i] - nMin) * size / delta;
        if(nBucket >= size)
            nBucket = size-1;
        pBucket[nBucket].Add(A[i]);
    }

    //计算有效桶的间隔
    i = 0;  //首个桶一定是有效的
    int nGap = delta / size;    //最小间隔
    int gap;
    for(int j = 1; j < size; j++) //i是前一个桶，j是后一个桶
    {
        if(pBucket[j].bValid)
        {
            gap = pBucket[j].nMin - pBucket[i].nMax;
            if(nGap < gap)
                nGap = gap;
            i = j;
        }
    }
    return nGap;
}
```

寻找和为定值的两个数

- 输入一个数组A[0…N-1]和一个数字Sum，在数组中查找两个数A_i,A_j，使得$A_i+A_j=Sum$。

暴力求解

- 从数组中任意选取两个数x,y，判定它们的和是否为输入的数字Sum。时间复杂度为$O(N^2)$，空间复杂度O(1)。

稍好一点的方法

- 两头扫
  - 如果数组是无序的，先排序O(NlogN)，然后用两个指针i，j，各自指向数组的首尾两端，令i=0，j=n-1，然后i++，j--，逐次判断a[i]+a[j]是否等于Sum：
  - 若a[i]+a[j]>sum，则i不变，j--；
  - 若a[i]+a[j]<sum，则i++，j不变；
  - 若a[i]+a[j]==sum，如果只要求输出一个结果，则退出；否则，输出结果后i++，j--；
- 数组无序的时候，时间复杂度最终为O(NlogN+N)=O(NlogN)。

Code

```
bool TwoSum(int* array, int nSize, int nSum, int& a, int& b)
{
    sort(array, array+nSize);

    int nBegin = 0;
    int nEnd = nSize-1;
    int nCur;
    bool bFind = false;
    while(nBegin < nEnd)
    {
        nCur = array[nBegin] + array[nEnd];
        if(nCur > nSum)
            nEnd--;
        else if(nCur < nSum)
            nBegin++;
        else
        {
            bFind = true;
            a = array[nBegin];
            b = array[nEnd];
            break;
        }
    }
    return bFind;
}
```

讨论：Hash方案的可行性

- 算法步骤
  - 选择适当的Hash函数，对原数组建立Hash结构
  - 遍历数组a[i]，计算Hash(Sum-a[i])是否存在
- 算法可行性
  - 时间复杂度，空间复杂度

排序的目的

- 排序本身：得到有序的序列
- 方便查找
  - 如：体会“2-sum问题”的求解过程。
  - 长度为N的有序数组，查找某元素的时间复杂度是多少？
  - 长度为N的有序链表，查找某元素的时间复杂度是多少？
    - 单链表、双向链表
    - 如何解决该问题？

跳跃链表(Skip List)

- AVL-Tree/RB-Tree/BTree
- 跳跃链表是一种随机化数据结构，基于并联的链表，其效率可比拟于二叉查找树(对于大多数操作需要O(logn)平均时间)。具有简单、高效、动态(Simple、Effective、Dynamic)的特点。
- 基本上，跳跃列表是对有序的链表附加辅助链表，增加是以随机化的方式进行的，所以在列表中的查找可以快速的跳过部分结点(因此得名)。查找结点、增加结点、删除结点操作的期望时间都是logN的(with high probability≈1-1/(n^α), W.H.P.)。
  - 将在后面的课程中详细阐述。

我们在这里

七月算法 http://www.julyedu.com/

- 视频/课程/社区

七月题库APP：Android/iOS

- http://www.julyapp.com/

微博

- @研究者July
- @七月题库
- @邹博_机器学习

微信公众号

- julyedu

感谢大家

恳请大家批评指正！