法律声明

- 本课件包括演示文稿、示例、代码、题库、视频和声音等内容，小象学院和主讲老师拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意及内容，我们保留一切通过法律手段追究违反者的权利。

- 课程详情请咨询
  - 微信公众号：小象
  - 新浪微博：ChinaHadoop

查找排序

邹博

主要内容

- 归并排序/逆序对
- 杨氏矩阵的增删改查
- Gantt图
- 2-sum问题
- 素和阶数问题
- 排序本身：
  - 插入排序、选择排序、希尔排序、冒泡排序
  - 堆排序/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

归并排序

- 基本思想：将数组A[0…n-1]中的元素分成两个子数组：A_1[0…n/2]和A_2[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：

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

- 若 $n=2^k$ ，则有：

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

$$=2\cdot\left(2\cdot T\left(\frac{n}{4}\right)+c\cdot\frac{n}{2}\right)+c\cdot n=4T\left(\frac{n}{4}\right)+2c\cdot n$$

$$=4\left(2\cdot T\left(\frac{n}{8}\right)+c\cdot\frac{n}{4}\right)+2c\cdot n=8T\left(\frac{n}{8}\right)+3c\cdot n$$

$$=8\left(2\cdot T\left(\frac{n}{16}\right)+c\cdot\frac{n}{8}\right)+3c\cdot n=16T\left(\frac{n}{16}\right)+4c\cdot n$$

$$=\cdots\cdots$$

$$=2^k T(1)+kc\cdot n=an+cn\log_2 n$$

- 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)是指处理超过内存限度的数据的排序算法。通常将中间结果放在读写较慢的外存储器(通常是硬盘)上。
- 外排序常采用“排序-归并”策略。
  - 排序阶段，读入能放在内存中的数据量，将其排序输出到临时文件，依次进行，将待排序数据组织为多个有序的临时文件。
  - 归并阶段，将这些临时文件组合为大的有序文件。

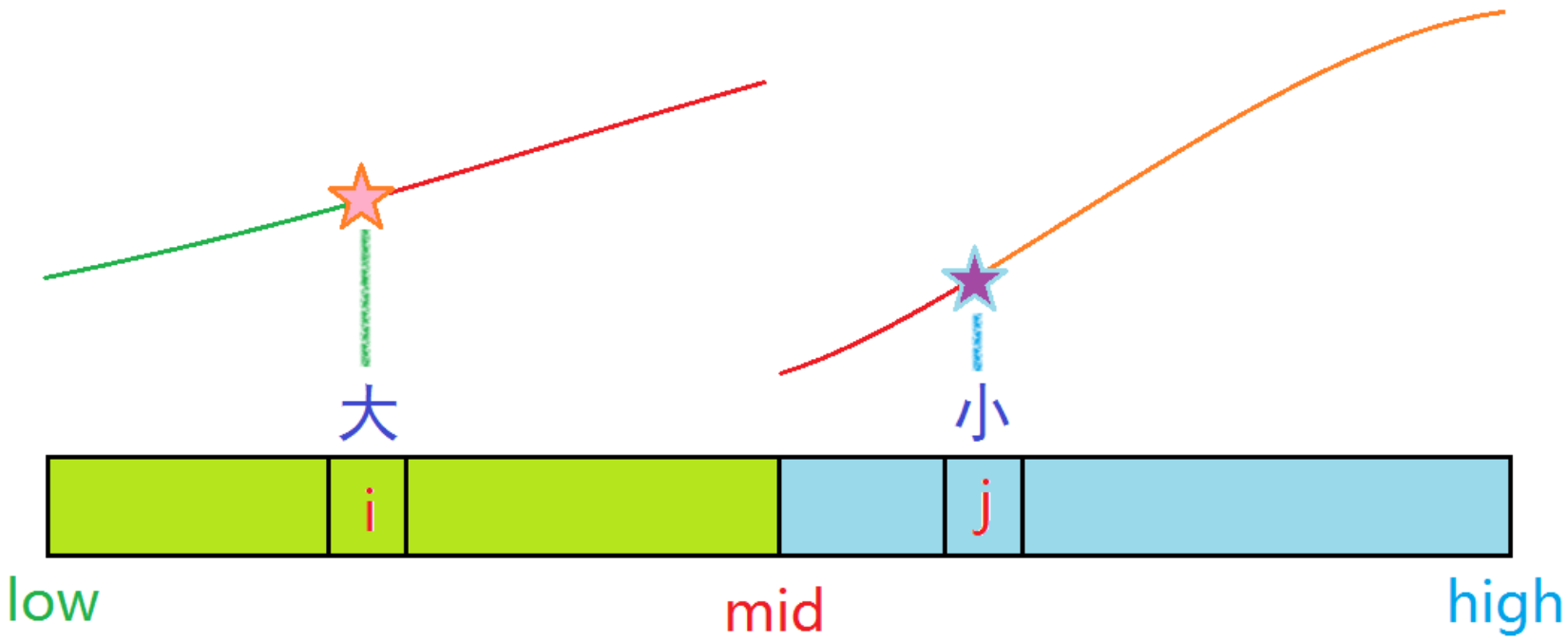
外排序举例

- 使用100M内存对900MB的数据进行排序：
  - 读入100M数据至内存，用常规方式(如堆排序)排序。
  - 将排序后的数据写入磁盘。
  - 重复前两个步骤，得到9个100MB的块(临时文件)中。
  - 将100M内存划分为10份，前9份中为输入缓冲区，第10份为输出缓冲区。
    - 如前9份各8M，第10份18M；或10份大小同时为10M。
  - 执行九路归并算法，将结果输出到输出缓冲区。
    - 若输出缓冲区满，将数据写至目标文件，清空缓冲区。
    - 若输入缓冲区空，读入相应文件的下一份数据。

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 _tmain(int argc, _TCHAR* argv[])
{
    int a[] = {2,4,6,34,9,7,5,44,3};
    int count = 0;
    MergeSort(a, 0, sizeof(a)/sizeof(in
    cout << count << endl;
    return 0;
}
```

(34, 9)
(7, 5)
(44, 3)
(5, 3)
(7, 3)
(4, 3)
(6, 3)
(9, 3)
(34, 3)
(6, 5)
(9, 5)
(34, 5)
(9, 7)
(34, 7)

```
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;
    while((i <= mid) && (j <= high))
    {
        if(a[i] < a[j])
        {
            temp[size++] = a[i++];
        }
        else
        {
            count += (mid - i + 1);
            PrintPair(a, j, i, mid);    //a[i...mid] - a[j]
            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];
}
```

杨氏矩阵 Young tableaus

- 给定M×N的二维数组，每一行、每一列都是有序的，则该二维数组称为杨氏矩阵。
  - 特殊的，行列都是递增排序的。
- 给定杨氏矩阵，如何查找其中的某个元素？
- 继续追问：杨氏矩阵的增删改查。

Code

```
#define INFINITY 100000
using namespace std;
class CYoungTableau
{
private:
    int m_nRow;
    int m_nCol;
    int** m_pData;

public:
    CYoungTableau(int row, int col);
    ~CYoungTableau();
    void Init(int row, int col);
    void Destroy();
    bool Insert(int x);
    bool Insert2(int x);
    void Delete(int row, int col);
    bool Find(int x, int& row, int& col) const;
    void Print() const;
};

CYoungTableau::CYoungTableau(int row, int col)
{
    Init(row, col);
}

CYoungTableau::~CYoungTableau()
{
    Destroy();
}
```

Code

```
bool CYoungTableau::Insert(int x)
{
    int row = m_nRow-1;
    int col = m_nCol-1;
    if(m_pData[row][col] < INFINITY)    //杨氏矩阵已满
        return false;
    m_pData[row][col] = x;
    int r = row;
    int c = col;
    while((row >= 0) || (col >= 0))
    {
        if((row >= 1) && (m_pData[row-1][col] > m_pData[r][c]))
        {
            r = row - 1;
            c = col;
        }
        if((col >= 1) && (m_pData[row][col-1] > m_pData[r][c]))
        {
            r = row;
            c = col - 1;
        }
        if((r == row) && (c == col))
            break;
        swap(m_pData[row][col], m_pData[r][c]);
        row = r;
        col = c;
    }
    return true;
}
```

插入效果

1：

41	67	∞	∞
∞	∞	∞	∞
∞	∞	∞	∞
∞	∞	∞	∞
∞	∞	∞	∞

2：

34	41	67	∞
∞	∞	∞	∞
∞	∞	∞	∞
∞	∞	∞	∞
∞	∞	∞	∞

3：

0	34	41	67
∞	∞	∞	∞
∞	∞	∞	∞
∞	∞	∞	∞
∞	∞	∞	∞

4：

0	34	41	67
69	∞	∞	∞
∞	∞	∞	∞
∞	∞	∞	∞
∞	∞	∞	∞

5：

0	34	41	67
24	69	∞	∞
∞	∞	∞	∞
∞	∞	∞	∞
∞	∞	∞	∞

6：

0	34	41	67
24	69	78	∞
∞	∞	∞	∞
∞	∞	∞	∞
∞	∞	∞	∞

7：

0	34	41	67
24	58	69	78
∞	∞	∞	∞
∞	∞	∞	∞
∞	∞	∞	∞

8：

0	34	41	67
24	58	69	78
62	∞	∞	∞
∞	∞	∞	∞
∞	∞	∞	∞

Code

```
bool IsBig(int a, int b)
{
    if(rand() % 2 == 0)
        return a >= b;
    return a > b;
}

bool CYoungTableau::Insert2(int x)
{
    int row = m_nRow-1;
    int col = m_nCol-1;
    if(m_pData[row][col] < INFINITY)    //杨氏矩阵已满
        return false;
    m_pData[row][col] = x;
    int r = row;
    int c = col;
    while((row >= 0) || (col >= 0))
    {
        if((row >= 1) && m_pData[row-1][col] > m_pData[r][c])
        {
            r = row - 1;
            c = col;
        }
        if((col >= 1) && IsBig(m_pData[row][col-1], m_pData[r][c]))
        {
            r = row;
            c = col - 1;
        }
        if((r == row) && (c == col))
            break;
        swap(m_pData[row][col], m_pData[r][c]);
        row = r;
        col = c;
    }
    return true;
}
```

Code

```
bool CYoungTableau::Find(int x, int& row, int& col) const
{
    row = 0;
    col = m_nCol - 1;
    while((row < m_nRow) && (col >= 0))
    {
        if(m_pData[row][col] == x)
            return true;
        if(x > m_pData[row][col])
            row++;
        else
            col--;
    }
    return false;
}
```

Code

```
void CYoungTableau::Delete(int row, int col)
{
    int r = row;
    int c = col;
    while((row < m_nRow) && (col < m_nCol))
    {
        if(m_pData[row][col] == INFINITY)
            break;
        if(row + 1 < m_nRow)
        {
            r = row + 1;
            c = col;
        }
        if((col + 1 < m_nCol) && (m_pData[row][col+1] < m_pData[r][c]))
        {
            r = row;
            c = col + 1;
        }
        if((row == r) && (col == c))
            break;
        m_pData[row][col] = m_pData[r][c];
        row = r;
        col = c;
    }
    m_pData[m_nRow-1][m_nCol-1] = INFINITY;
}
```

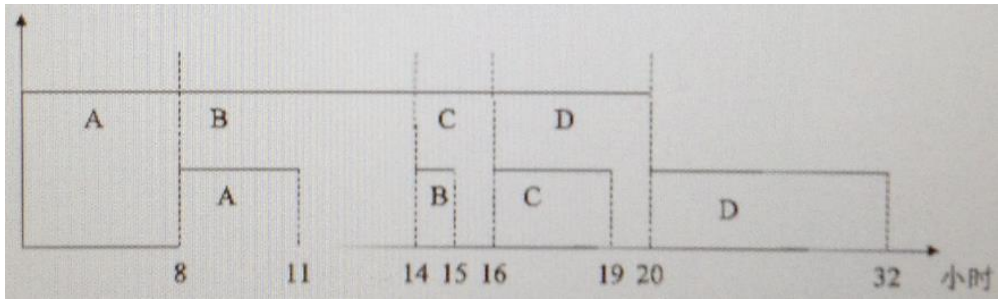
迷离傍地走

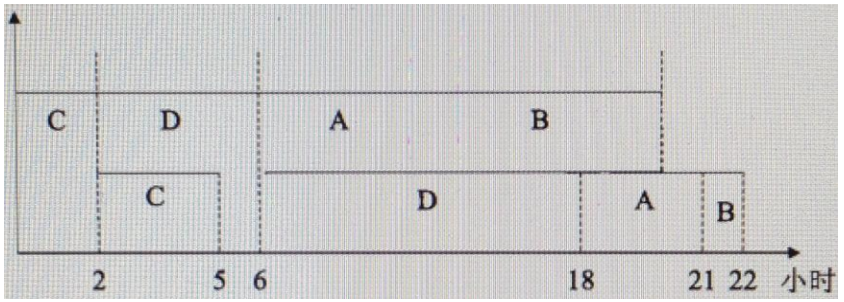
- 为庆祝强汉文武盛世暨废除和亲七百周年，武后决定拜孙武和王翦对春夏秋冬四官细君、昭君、探春、文成做军事训练。
- 孙武和王翦分别负责四官的站军姿和踢正步科目。根据军训要求，只有在学会站军姿之后才能进行踢正步训练。由于四官天资差别，学习时间如下表。
- 问：应该如何安排四官的学习时间，能够使得所有人学会这两项技能的时间最短？

	细君	昭君	探春	文成
站军姿 孙武	8小时	6小时	2小时	4小时
踢正步 王翦	3小时	1小时	3小时	12小时

分析事件

	细君	昭君	探春	文成
站军姿	8小时	6小时	2小时	4小时
踢正步	3小时	1小时	3小时	12小时





- 策略：所有事件中的最短时间是昭君踢正步(1小时)，由于该事件是第二阶段，则将其最后执行；此短时间是探春站军姿(2小时)，由于该事件是第一阶段，则将其优先执行；以此类推。

Code

```
typedef struct tagSItem
{
    int t;
    int nIndex;
    bool bFirst;

    bool operator < (struct tagSItem& item) const
    {
        return t < item.t;
    }
}SItem;

int _tmain(int argc, _TCHAR* argv[])
{
    int a[] = {8,6,2,4};
    int b[] = {3,1,3,12};
    int n = sizeof(a) / sizeof(int);
    int*c = new int[n];
    GanttChart(a, b, c, n);
    Print(c, n);
    delete[] c;
    return 0;
}
```

```
void GanttChart(const int* a, const int* b, int* c, int size)
{
    int i;
    int s2 = 2*size;
    //整理数据
    SItem* pItem = new SItem[s2];
    for(i = 0; i < size; i++)
    {
        pItem[i].t = a[i];
        pItem[i].nIndex = i;
        pItem[i].bFirst = true;

        pItem[size+i].t = b[i];
        pItem[size+i].nIndex = i;
        pItem[size+i].bFirst = false;
    }
    sort(pItem, pItem+s2);

    bool* bArrage = new bool[size]; //d[i]：第i号已经安排
    for(i = 0; i < size; i++)
        bArrage[i] = false;
    int from = 0;
    int to = size-1;
    for(i = 0; i < s2; i++)
    {
        if(bArrage[pItem[i].nIndex]) //pItem[i].nIndex已经确定
            continue;
        bArrage[pItem[i].nIndex] = true;
        if(pItem[i].bFirst)
        {
            c[from] = pItem[i].nIndex;
            from++;
        }
        else
        {
            c[to] = pItem[i].nIndex;
            to--;
        }
        if(to - from < 0)   //提前退出
            break;
    }
    delete[] pItem;
    delete[] bArrage;
}
```

寻找和为定值的两个数

- 给定N个不同的数A[0…N-1]以及某定值sum，找到这N个数中的两个数，使得它们的和为sum。
  - 如给定数组[0, 3, 7, 9, 11,14, 16, 17]，sum=20，则返回[3,17]，[9,11]。

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)。

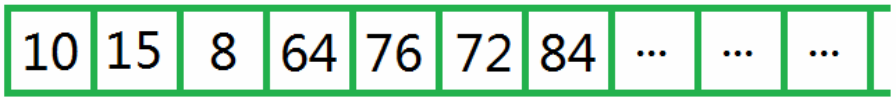
稍好一点的方法

- 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)。

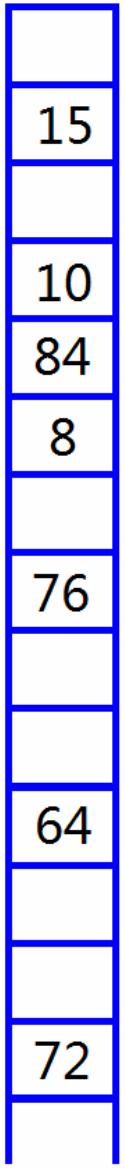
Code

```
void TwoSum(int* a, int N, int sum)
{
    int i = 0;
    int j = N-1;
    while(i < j)
    {
        if(a[i] + a[j] < sum)
            i++;
        else if(a[i] + a[j] > sum)
            j--;
        else    //if(a[i] + a[j] == sum)
        {
            cout << a[i] << ',' << a[j] << '\n';
            i++;
            j--;
        }
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    int a[] = {0, 3, 7, 9, 11, 14, 16, 17};
    //sort(a, a+N); 若给定数组未排序，则先排序
    TwoSum(a, sizeof(a)/sizeof(int), 20);
    return 0;
}
```

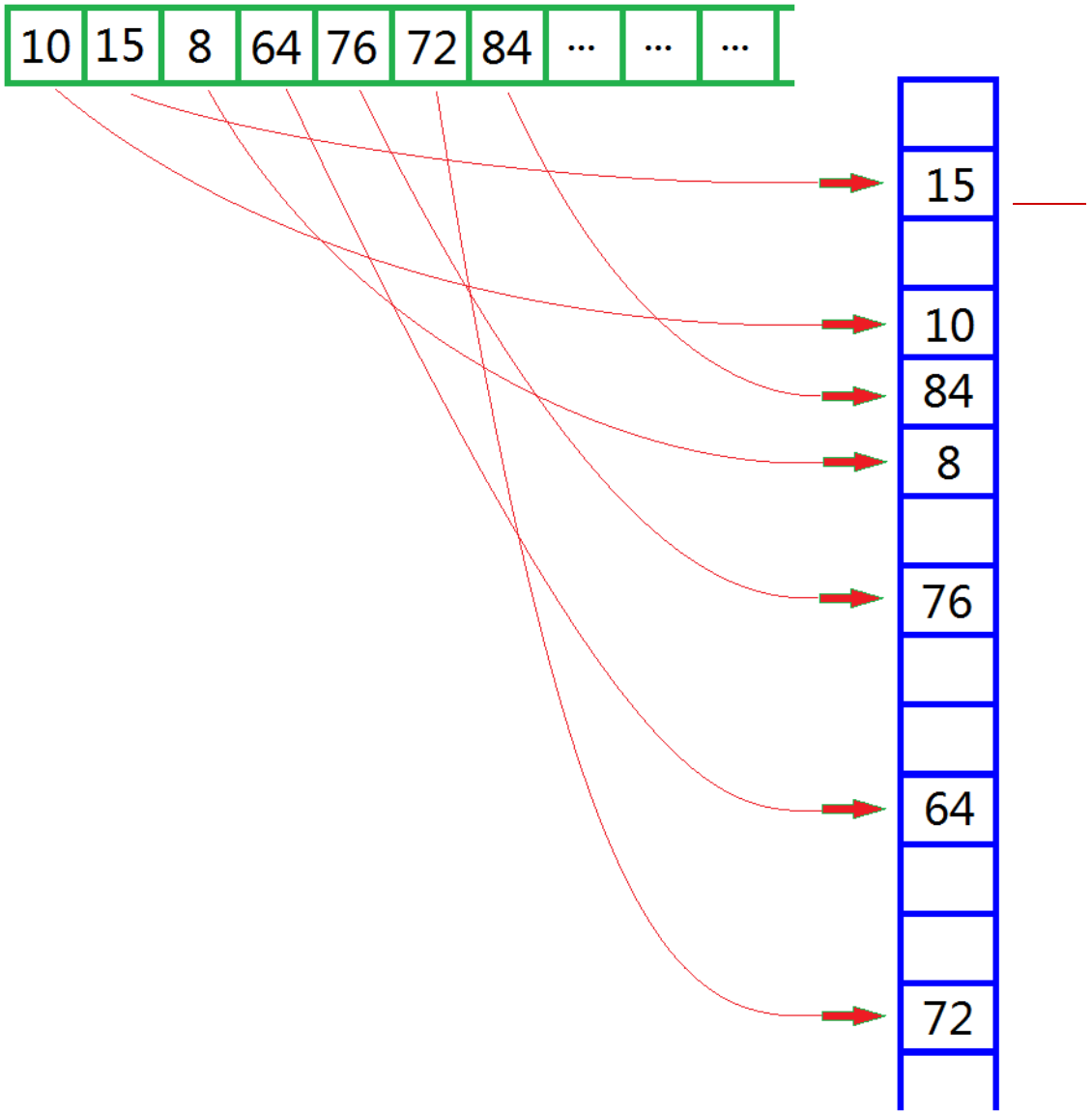
Hash结构



- 事先开辟一个足够大的空间H。以数组元素值x为自变量，通过某函数f，将其映射为某个整数值index，将该整数值x存储在H[index]处，这样，在以后查找元素a是否存在时，只需要计算f(a)，得到一个值y，看H[y]处是否存在a即可。
- 显然，它是通过足够大的存储空间，保证了O(1)的时间找到某元素(如果把函数值的计算认为是常数级的)。



Hash示例



Hash函数

- Hash函数在Hash结构设计中居于核心地位。一个好的Hash函数，能够将数据映射到“杂乱”的位置。
  - dbj2
  - sdbm
  - MurmurHash

djb2 Code

```
unsigned long Hash(unsigned char* str)
{
    unsigned long hash = 5381;
    int c;
    while(c = *str++)
        hash = ((hash << 5) + hash) + c;  //hash * 33 + c
    return hash;
}
```

SDBM：(65599)1 0000 0000 0011 111

```
unsigned long SDBM(unsigned char* str)
{
    unsigned long hash = 0;
    int c;
    while(c = *str++)
        hash = (hash << 6) + (hash << 16) - hash + c; //hash * 65599 + c
    return hash;
}
```

ELFhash算法

```
int ELFhash(char*key)
{
    unsigned long h=0;
    while(*key)
    {
        h = (h << 4) + *key++;
        unsigned long g = h & 0xF0000000L;
        if(g)
            h ^= g >> 24;
        h &= ~g;
    }
    return h % MOD;
}
```

MurmurHash

- MurmurHash是一种非加密型哈希函数，适用于一般的哈希检索操作。由Austin Appleby在2008年发明，并出现了多个变种，都已经发布到了公有领域(public domain)。与其它流行的哈希函数相比，对于规律性较强的key，Murmur Hash的随机分布特征表现更良好。
- 当前的版本是MurmurHash3，能够产生出32-bit或128-bit哈希值。
- 较早的MurmurHash2能产生32-bit或64-bit哈希值。对于大端存储和强制对齐的硬件环境有一个较慢的MurmurHash2可以用。MurmurHash2A变种增加了Merkle–Damgård构造，所以能够以增量方式调用。有两个变种产生64-bit哈希值：MurmurHash64A，为64位处理器做了优化；MurmurHash64B，为32位处理器做了优化。MurmurHash2-160用于产生160-bit哈希值，而MurmurHash1已经不再使用。

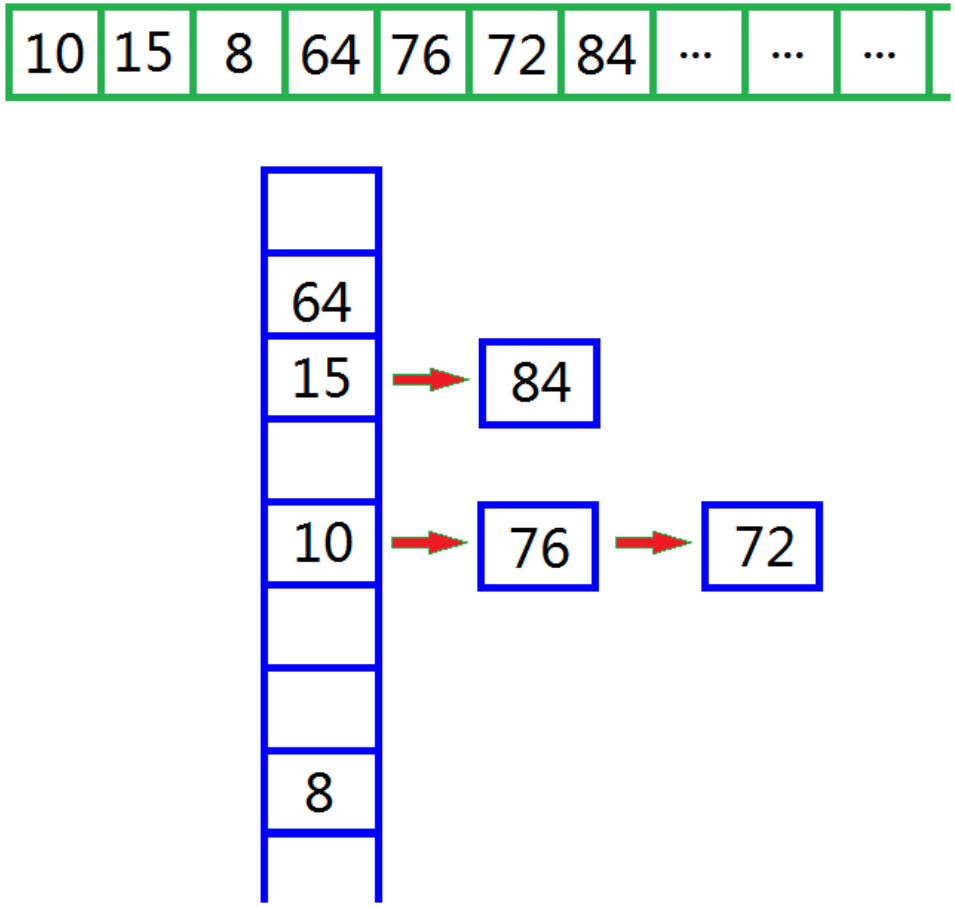
MurmurHash

```
Murmur3_32(key, len, seed)
    c1 ← 0xcc9e2d51
    c2 ← 0x1b873593
    r1 ← 15
    r2 ← 13
    m ← 5
    n ← 0xe6546b64

    hash ← seed

    for each fourByteChunk of key
        k ← fourByteChunk

        k ← k * c1
        k ← (k << r1) OR (k >> (32-r1))
        k ← k * c2

        hash ← hash XOR k
        hash ← (hash << r2) OR (hash >> (32-r2))
        hash ← hash * m + n

    with any remainingBytesInKey
        remainingBytes ← SwapEndianOrderOf(remainingBytesInKey)
        remainingBytes ← remainingBytes * c1
        remainingBytes ← (remainingBytes << r1) OR (remainingBytes >> (32 - r1))
        remainingBytes ← remainingBytes * c2

        hash ← hash XOR remainingBytes

    hash ← hash XOR len

    hash ← hash XOR (hash >> 16)
    hash ← hash * 0x85ebca6b
    hash ← hash XOR (hash >> 13)
    hash ← hash * 0xc2b2ae35
    hash ← hash XOR (hash >> 16)
```

Hash冲突

- 当有两个或以上数量的键被分配到了哈希表数组的同一个索引上面时，我们称这些键发生了冲突(collision)。

Hash冲突



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

探索：素和阶数

- 有些正整数可以写成两个素数的和，如：8=3+5；有些正整数还有多种写法，如：16=3+13=5+11。
- 定义：一个正整数可以被拆分成两素数和的数目为“素和阶数”。请计算100万以内哪个数的素和阶数最大。

一种可行的思路

- 使用Eratosthenes筛法计算100万以内所有素数，存储在数组p[0…size-1]中；
- 正整数Z从1到100万依次遍历：
  - 查找数组p中大于等于Z的最小值，记该最小值的位置为s。
  - 使用2-Sum算法计算p[0,s-1]中和为s的所有组合
- 遍历过程中，记录拆分种类最多的值。

Code

996599	996601	996617	996629	996631	996637	996647	996649	996689
996703	996739	996763	996781	996803	996811	996841	996847	996857
996859	996871	996881	996883	996887	996899	996953	996967	996973
996979	997001	997013	997019	997021	997037	997043	997057	997069
997081	997091	997097	997099	997103	997109	997111	997121	997123
997141	997147	997151	997153	997163	997201	997207	997219	997247
997259	997267	997273	997279	997307	997309	997319	997327	997333
997343	997357	997369	997379	997391	997427	997433	997439	997453
997463	997511	997541	997547	997553	997573	997583	997589	997597
997609	997627	997637	997649	997651	997663	997681	997693	997699
997727	997739	997741	997751	997769	997783	997793	997807	997811
997813	997877	997879	997889	997891	997897	997933	997949	997961
997963	997973	997991	998009	998017	998027	998029	998069	998071
998077	998083	998111	998117	998147	998161	998167	998197	998201
998213	998219	998237	998243	998273	998281	998287	998311	998329
998353	998377	998381	998399	998411	998419	998423	998429	998443
998471	998497	998513	998527	998537	998539	998551	998561	998617
998623	998629	998633	998651	998653	998681	998687	998689	998717
998737	998743	998749	998759	998779	998813	998819	998831	998839
998843	998857	998861	998897	998909	998917	998927	998941	998947
998951	998957	998969	998983	998989	999007	999023	999029	999043
999049	999067	999083	999091	999101	999133	999149	999169	999181
999199	999217	999221	999233	999239	999269	999287	999307	999329
999331	999359	999371	999377	999389	999431	999433	999437	999451
999491	999499	999521	999529	999541	999553	999563	999599	999611
999613	999623	999631	999653	999667	999671	999683	999721	999727
999749	999763	999769	999773	999809	999853	999863	999883	999907
999917	999931	999953	999959	999961	999979	999983		

```
void Eratosthenes(bool* a, int n)
{
    a[1] = false;//a[0]不用
    int i;
    for(i = 2; i <= n; i++) //筛法，默认是素数
        a[i] = true;
    int p = 2;  //第一个筛孔
    int j = p*p;
    int c = 0;
    while(j <= n)
    {
        while(j <= n)
        {
            a[j] = false;
            j += p;
        }
        p++;
        while(!a[p])     //查找下一个素数
            p++;
        j = p*p;
    }
}
```

Code

9240 339
99330： 2168
990990： 15594

```
int _tmain(int argc, _TCHAR* argv[])
{
    ifstream iFile;
    iFile.open("D:\\Prime.txt");     //读入素数
    vector<int> a;
    int i;
    while(iFile >> i)
        a.push_back(i);

    int* p = &a.front();     //100万以内所有的素数
    int size = (int)a.size();   //100万素数的个数
    int j;
    int m = 0;  //素和阶数
    int f = 0;  //哪个整数得到最大的素和阶数
    for(i = 0; i < 1000000; i++)
    {
        j = Find(p, size, i);   //返回p[0,size-1]大于i的最小值索引
        j = TwoSum(i, p, j);    //p[x]+p[y]==i，返回几种拆分方法
        if(m < j)
        {
            m = j;
            f = i;
        }
    }
    cout << f << ":\t" << m << endl;
    return 0;
}
```

Code

```
int Find(const int* a, int size, int x)
{
    int low = 0;
    int high = size-1;
    int mid;
    while(low <= high)
    {
        mid = (low + high) / 2;
        if(a[mid] == x)
            return mid;
        if(a[mid] > x)
            high = mid - 1;
        else
            low = mid + 1;
    }
    return low;
}
```

Code

```
int TwoSum(int sum, const int* a, int size)
{
    int low = 0;
    int high = size-1;
    int s;
    int times = 0;
    while(low < high)
    {
        s = a[low] + a[high];
        if(s == sum)
        {
            times++;
            //cout << a[low] << " + " << a[high] << '\n';
            low++;
            high--;
        }
        else if(s > sum)
            high--;
        else //if(s < sum)
            low++;
    }
    return times;
}
```

排序的稳定性

- 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	

数组的最大间隔

- 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？

问题分析

- 假定N个数的最大最小值为max，min，则这N个数形成N-1个间隔，其最小值是 $\frac{max-min}{N-1}$
  - 如果N个数完全均匀分布，则间距全部是 $\frac{max-min}{N-1}$ 且最小；
  - 如果N个数不是均匀分布，间距不均衡，则最大间距必然大于 $\frac{max-min}{N-1}$

解决思路

- 思路：将N个数用间距 $\frac{max-min}{N-1}$ 分成N-1个区间，则落在同一区间内的数不可能有最大间距。统计后一区间的最小值与前一区间的最大值的差即可。
  - 若没有任何数落在某区间，则该区间无效，不参与统计。
  - 显然，这是借鉴桶排序/Hash映射的思想。

桶的数目

- 同时，N-1个桶是理论值，会造成若干个桶的数目比其他桶大1，从而造成统计误差。
  - 如：7个数，假设最值为10、80，如果适用6个桶，则桶的大小为70/6=11.66，每个桶分别为：[10,21]、[22,33]、[34,44]、[45,56]、[57,68]、[69,80]，存在大小为12的桶，比理论下界11.66大。
- 因此，使用N个桶。

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;
}
```

桶排序/基数排序

- 将元素分于若干桶中，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)
- 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？
  - 最大间隔问题

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？

我们在这里

- http://wenda.ChinaHadoop.cn
  - 视频/课程/社区
- 微博
  - @ChinaHadoop
  - @邹博_机器学习
- 微信公众号
  - 小象
  - 大数据分析挖掘

感谢大家！

恳请大家批评指正！

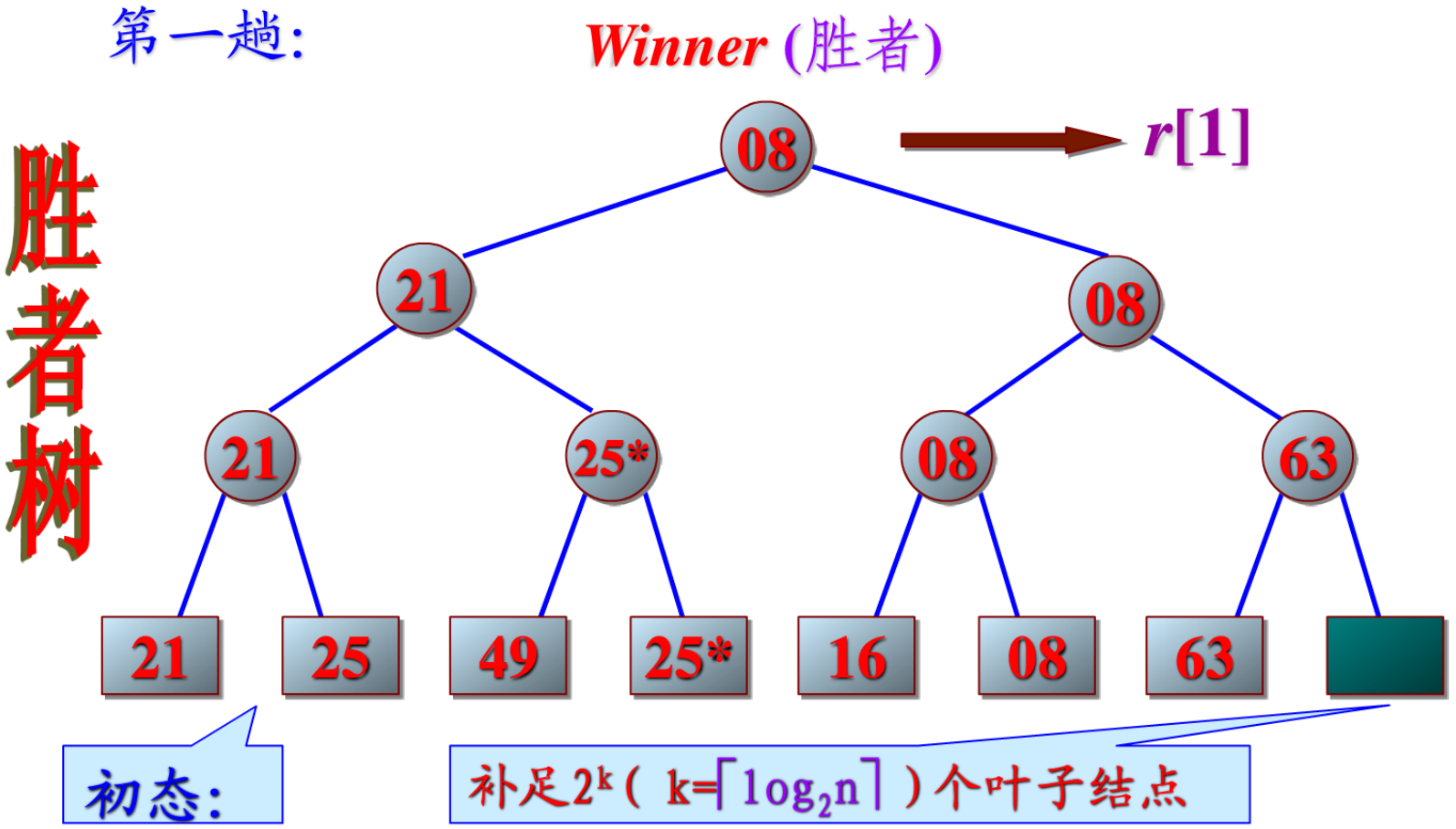
附录：排序问题的提法

- 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

冒泡排序

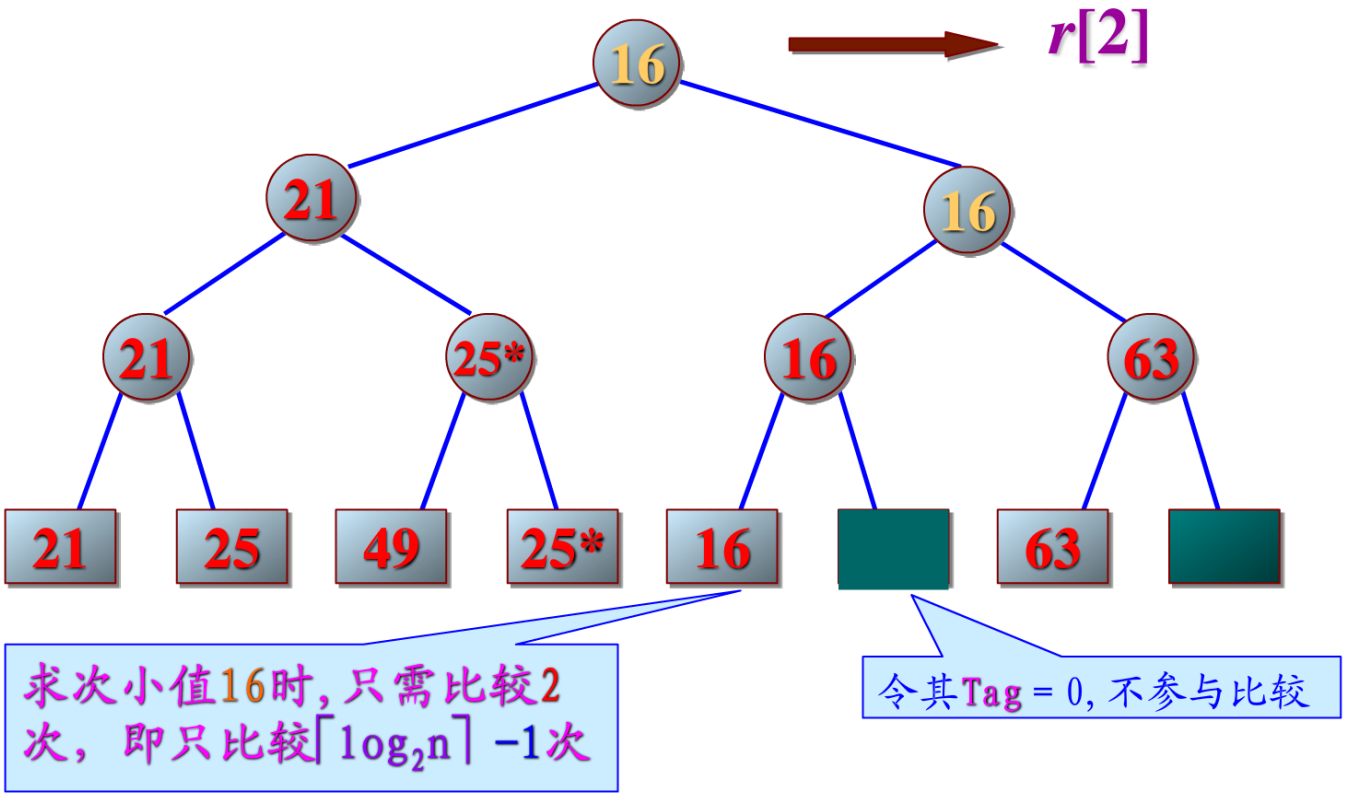
```
void BubbleSort(int* a, int size)
{
    int i,j;
    int t;
    bool bBubble;
    for(i = 0; i < size-1; i++)
    {
        bBubble = false;
        for(j = 0; j < size-i-1; j++)
        {
            if(a[j] > a[j+1])
            {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                bBubble = true;
            }
        }
        if(!bBubble)
            break;
    }
}
```

锦标赛排序


第一趟:
Winner (胜者)
胜者树
08
r[1]
21
08
21
25*
08
63
21
25
49
25*
16
08
63
初态:
补足2^k (k=⌈log2n⌉)个叶子结点

锦标赛排序

第二趟: *Winner* (胜者)



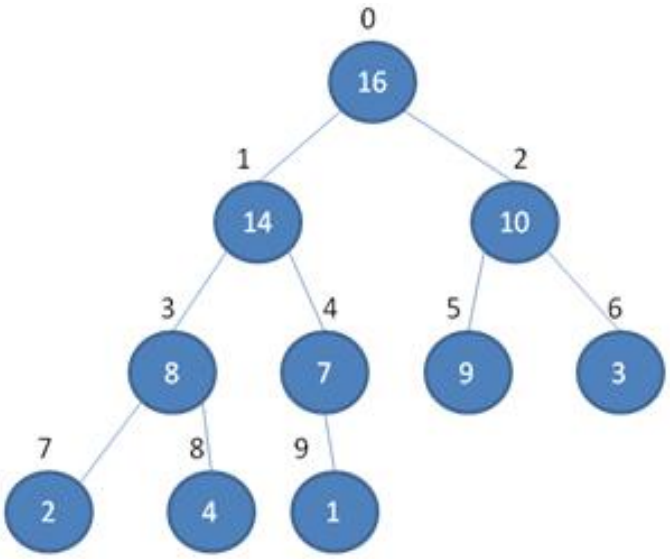
堆的定义和表示

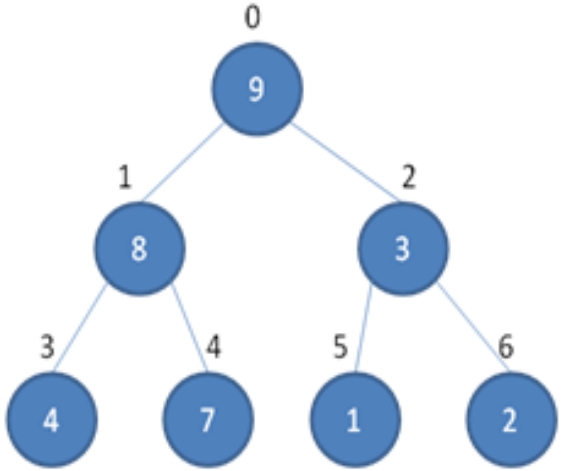
- 定义：对于一棵完全二叉树，若树中任一非叶子结点的关键字均不大于(或不小于)其左右孩子(若存在)结点的关键字，则这棵二叉树，叫做小顶堆(大顶堆)。
- 完全二叉树可以用数组完美存储，对于长度为n的数组a[0…n-1]，若
  - ∀0≤i≤n-1，a[i]≤a[2i+1]且a[i]≤a[2i+2]

  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

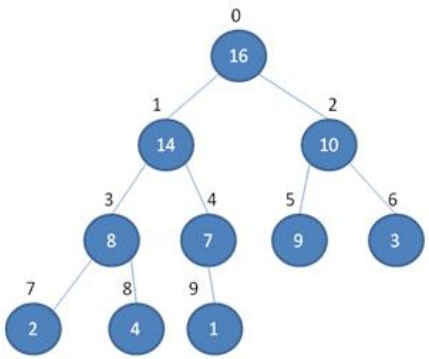
- k的孩子结点是2k+1,2k+2(如果存在)
- k的父结点：
  - 若k为左孩子，则k的父结点为k/2
  - 若k为右孩子，则k的父结点为(k/2) - 1
- 二者公式不一样，十分不便。发现：
  - 若k为左孩子，则k为奇数，则((k+1)/2)-1与k/2相等
  - 若k为右孩子，则k为偶数，则((k+1)/2)-1与(k/2) - 1相等
- 结论：若待考查结点为k，记k+1为K，则k的父结点为：(K/2) - 1

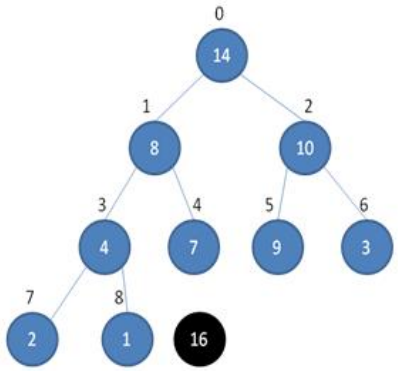
堆排序的整体思路

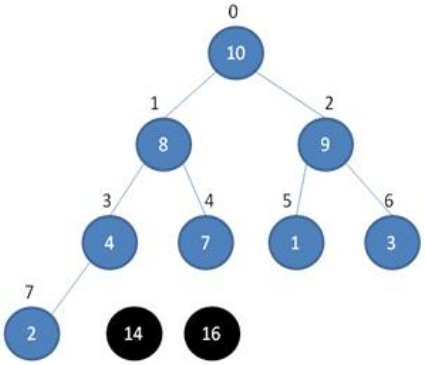
- ① 初始化操作：将a[0..n-1]构造为堆(如大顶堆)；
- ② 第i(n > i≥1)趟排序：将堆顶记录a[0]和a[n-i] 交换，然后将a[0…n-i-1] 调整为堆(即：重建大顶堆)；
- ③进行n-1趟，完成排序。

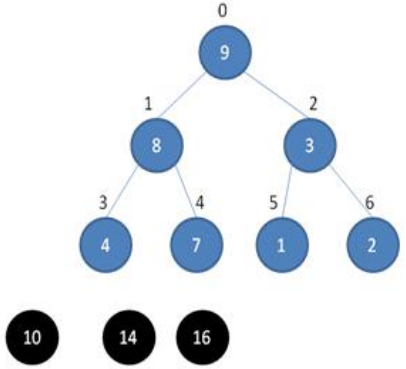
- 堆排序的时间复杂度？
  - 初始化堆的过程：O(N)
    - 注意，一般教科书给出的O(NlogN)不是紧的。
  - 调整堆的过程：O(NlogN)

堆排序的调整过程


0
16
1
14
2
10
3
8
4
7
5
9
6
3
7
2
8
4
9
1


0
14
1
8
2
10
3
4
4
7
5
9
6
3
7
2
8
1
16


0
10
1
8
2
9
3
4
4
7
5
1
6
3
7
2
14
16

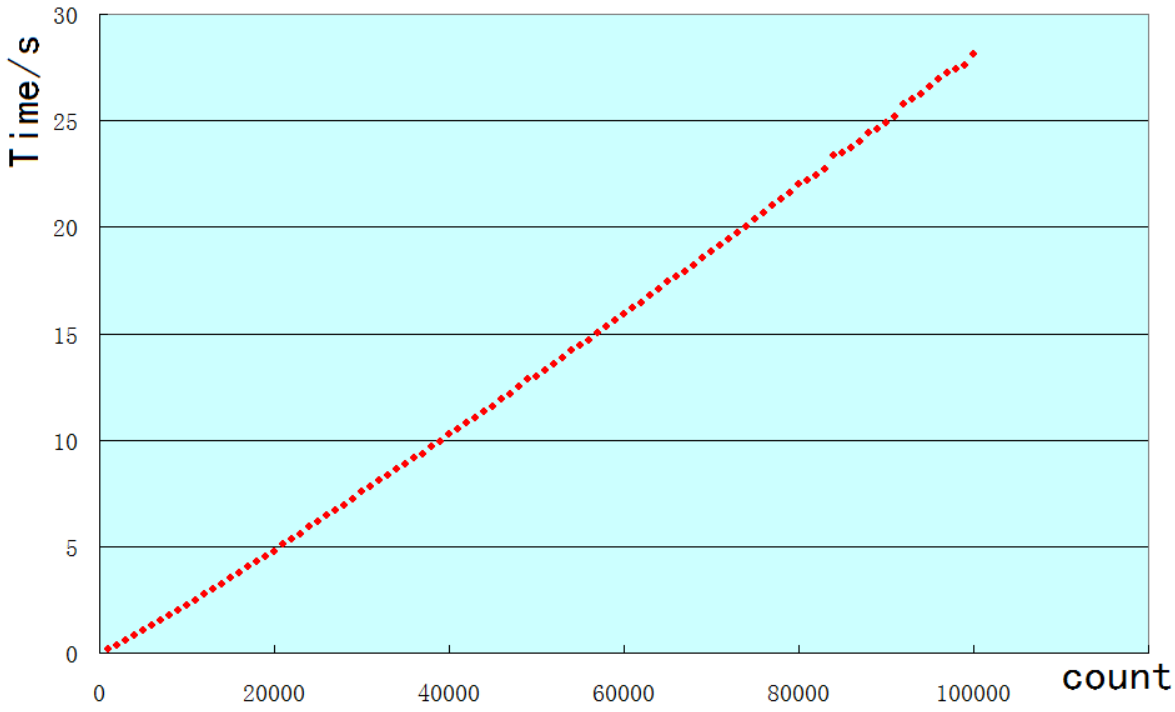

0
9
1
8
2
3
3
4
4
7
5
1
6
2
10
14
16

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

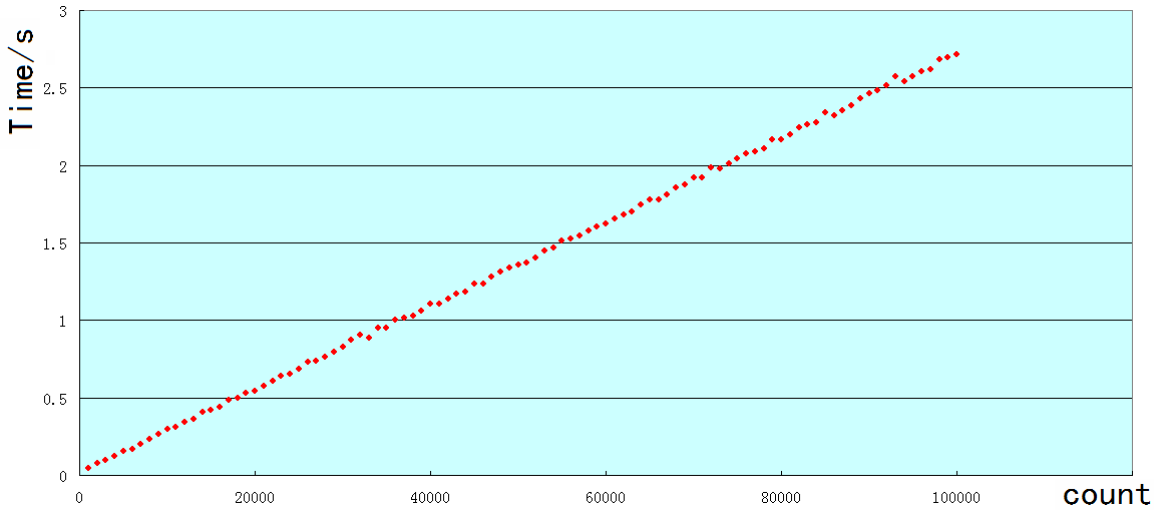
- 小顶堆的作用：
  - 保持始终有k个最大元素——利于最后的输出
  - k个元素中最小的元素在堆顶——利于后续元素的比较
- 时间复杂度：O(N*logk)

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？

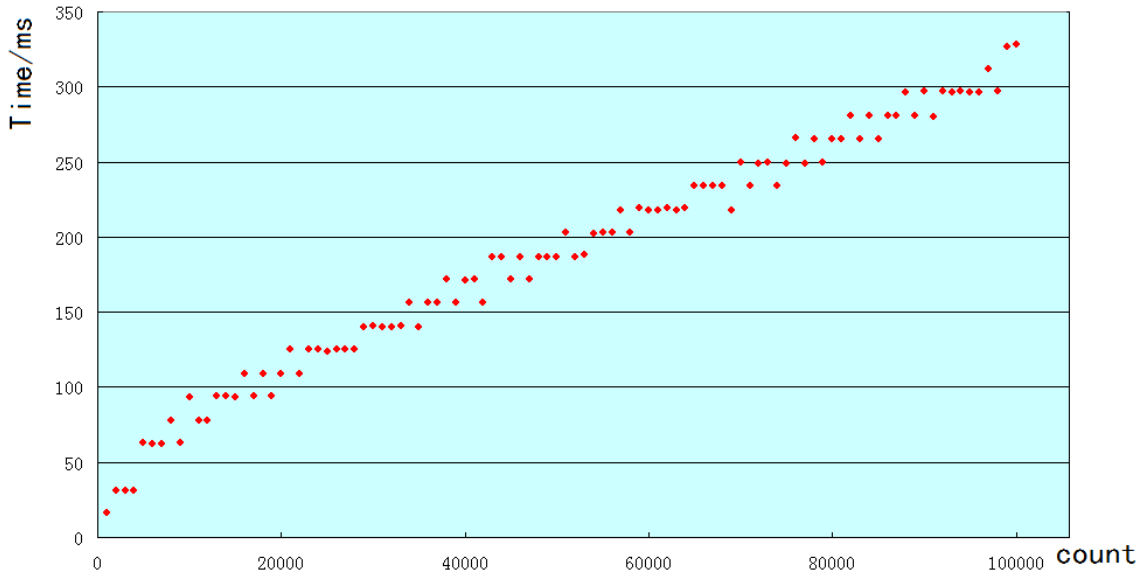
最大的k个数——算法2 Code

```
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;
}
```



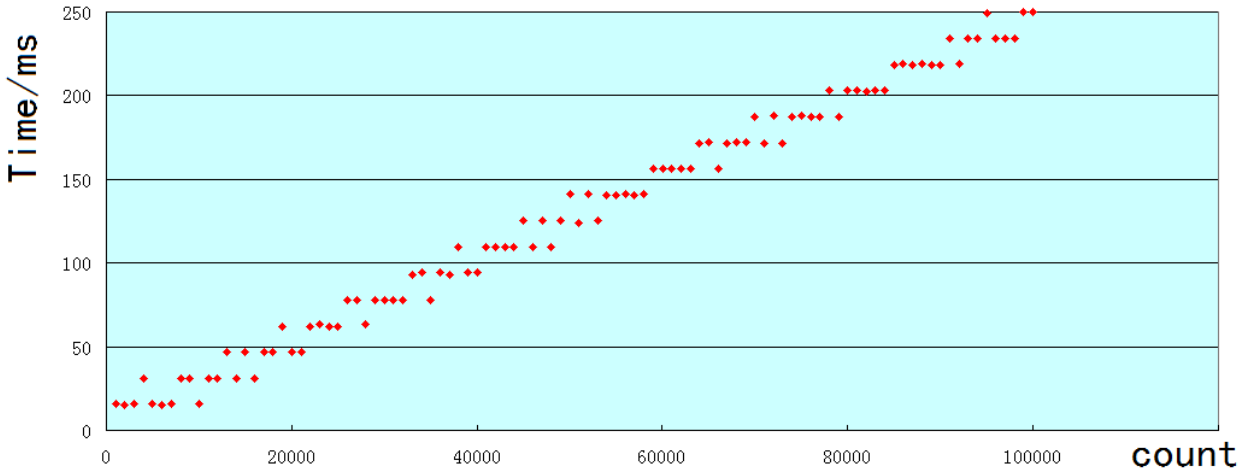
最大的k个数——算法1 Code

```
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

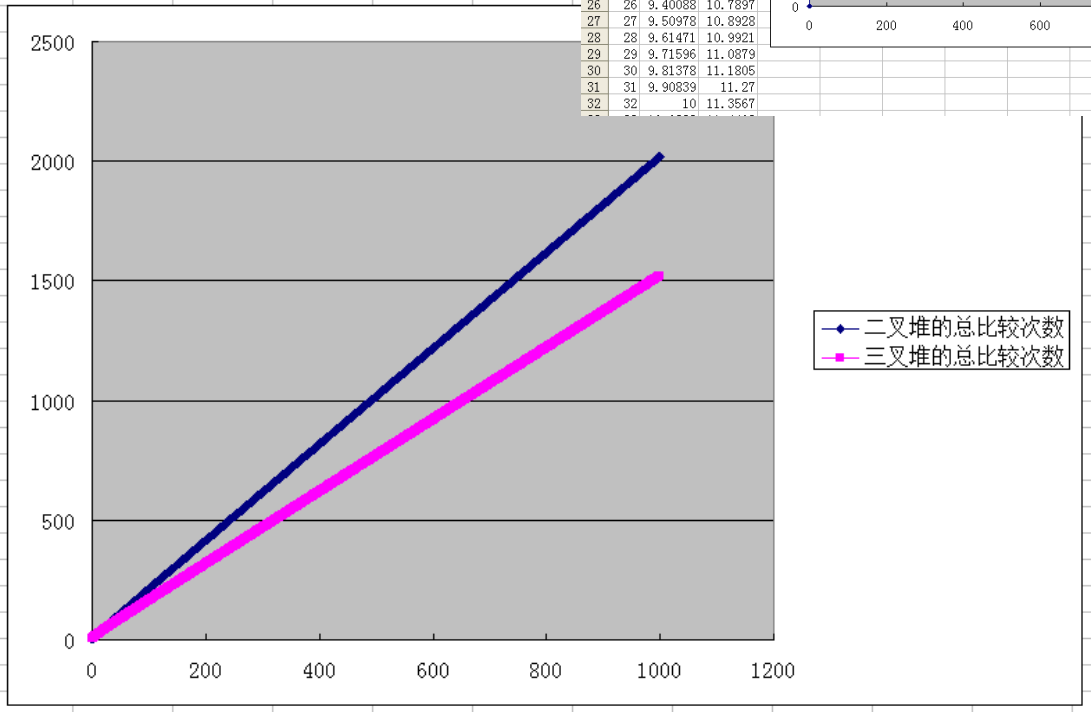
- 实践证明，算法1的方案相对于算法2更优
- 事实上，有更快的BFPRT算法。
  - 这不能抹杀算法1和算法2在实际中的存在价值
  - 稍后马上介绍

K叉堆的结论

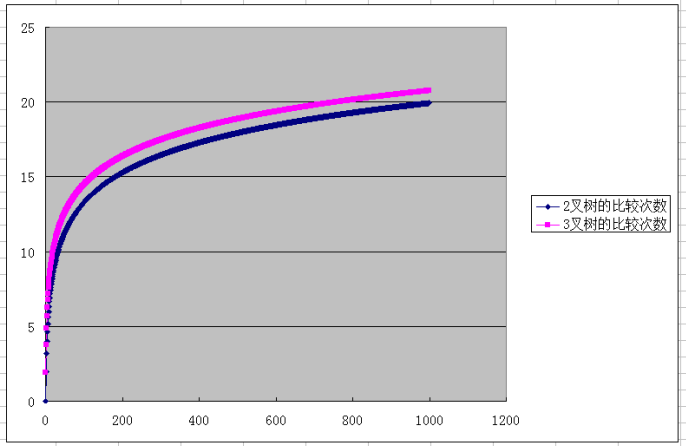
- 对 n 个元素建立初始 K 叉堆的最多比较次数不超过(k/k-1)*n 次；
- 对 n 个元素的 K 叉堆，每次删去堆顶元素并调整使之恢复 K 叉堆，这样 m 次过程的最多比较次数不超过 $m*k*[\log_k ((k-1)n]$ 次。

3叉堆与2叉堆

	A	B	C
1	1	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



	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



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key的值a[i]，a[j] = a[i]；
  - 重复第3、4步，直到i≥j.

附：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。

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;
}
```

Code

```
int Patition(int key, int* a, int from, int to)
{
    int t;
    while(from < to)
    {
        while((from < to) && a[from] < key)
            from++;
        while((from < to) && a[to] >= key)
            to--;

        t = a[from];
        a[from] = a[to];
        a[to] = t;
    }
    return from;
}
```

```
void _QuickSort(int* a, int from, int to)
{
    if(to - from <= 10)
    {
        BubbleSort(a+from, to-from+1);
        return;
    }

    int key = SelectPivotkey(a[from], a[to], a[(from+to)/2]);
    int nIndex = Patition(key, a, from, to);
    _QuickSort(a, from, nIndex-1);
    _QuickSort(a, nIndex, to);
}

void QuickSort(int* a, int size)
{
    _QuickSort(a, 0, size-1);
}
```

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。

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。

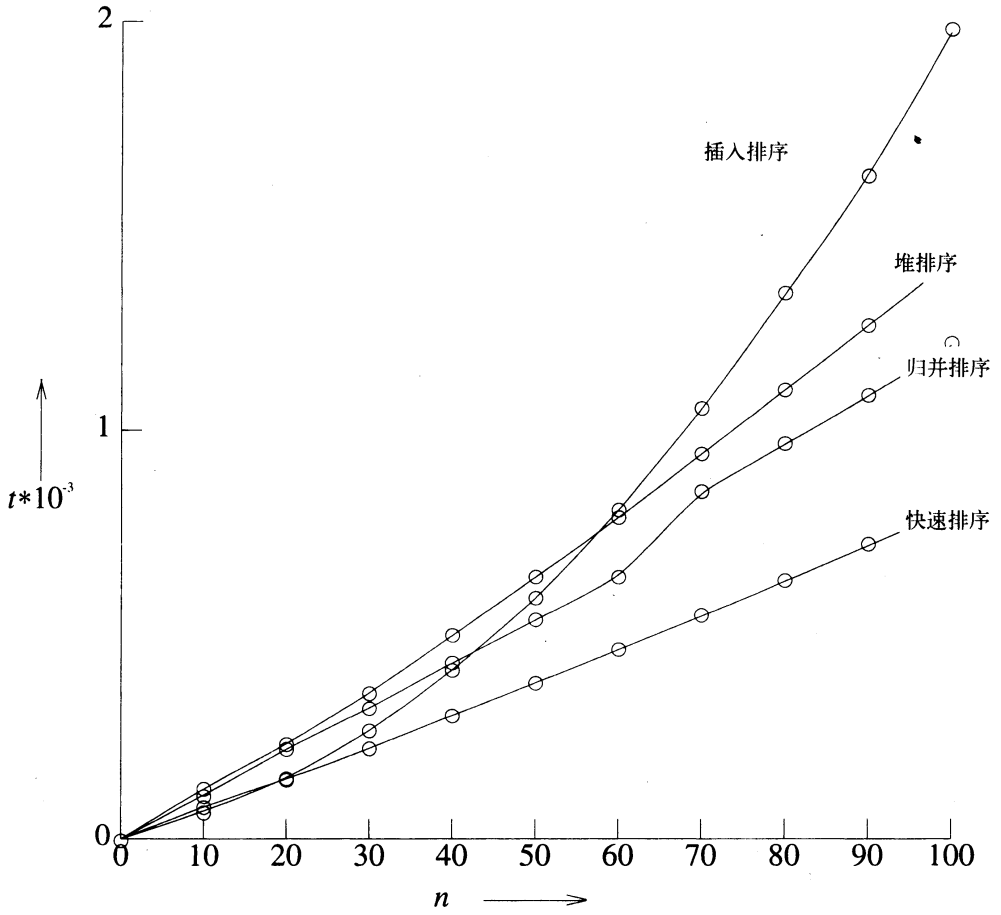
考虑相等元素的O(N)时间选择算法

```
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

感谢大家！

恳请大家批评指正！