北京航空航天大學 BEIHANG UNIVERSITY


数据结构与程序设计（信息类）

Data Structure & Programming

北京航空航天大学 数据结构课程组
软件学院 林广艳
2023年春

Sort Benchmark Home Page

Top Results

	Daytona	Indy
Gray	2016, 44.8 TB/min **Tencent Sort** 100 TB in 134 Seconds 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub	2016, 60.7 TB/min **Tencent Sort** 100 TB in 98.8 Seconds 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub
Cloud	2016, $1.44 / TB **NADSort** 100 TB for $144 394 Alibaba Cloud ECS ecs.n1.large nodes x (Haswell E5-2680 v2, 8 GB memory, 40GB Ultra Cloud Disk, 4x 135GB SSD Cloud Disk) Qian Wang, Rong Gu, Yihua Huang Nanjing University Reynold Xin Databricks Inc. Wei Wu, Jun Song, Junluan Xia Alibaba Group Inc.	2016, $1.44 / TB **NADSort** 100 TB for $144 394 Alibaba Cloud ECS ecs.n1.large nodes x (Haswell E5-2680 v2, 8 GB memory, 40GB Ultra Cloud Disk, 4x 135GB SSD Cloud Disk) Qian Wang, Rong Gu, Yihua Huang Nanjing University Reynold Xin Databricks Inc. Wei Wu, Jun Song, Junluan Xia Alibaba Group Inc.
Minute	2016, 37 TB **Tencent Sort** 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub	2016, 55 TB **Tencent Sort** 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub
Joule 10^{10} recs	2021, 138 KJoules **RezSort** 72 K records sorted / joule Intel i7-10700, 16GB RAM, Nsort, Ubuntu 18.04.5 LTS, 2 SK hynix Gold P31 1TB SSDs, 1 Samsung 980 Pro 2TB SSD Waleed Reda Université catholique de Louvain, KTH Royal Institute of Technology Dejan Kostić KTH Royal Institute of Technology	2019, 89 KJoules **KioxiaSort** 112 K records sorted / joule Intel i9-9900K, 64GB RAM, Ubuntu 19.04 Server, 8 CFD CSSD-M2B1TPG3VNF (1TB), 1 Toshiba XG5-P KXG50PNV2T04 (2TB) Shintaro Sano, Tomoya Suzuki Kioxia Corporation Zaid Mahmoud Princess Sumaya University for Technology

3

内容提要

2

1. 排序（Sort）的基本概念

- 排序是将一个按值任意的数据元素序列转换为一个按值有序的数据元素序列

对于含有n个记录的序列{ R_1, R_2, …, R_n }, 对应的关键字序列为{k_1, k_2, …, k_n }, 确定一种置换关系

$\sigma(1), \sigma(2), \ldots, \sigma(n)$

使得关键字序列满足:

$k_{\sigma(1)} \le k_{\sigma(2)} \le \ldots \le k_{\sigma(n)}$ 或者 $k_{\sigma(1)} \ge k_{\sigma(2)} \ge \ldots \ge k_{\sigma(n)}$

相应文件称为**按关键字值有序**的文件

$\{ R_{\sigma(1)}, R_{\sigma(2)}, \ldots, R_{\sigma(n)} \}$

这一过程称为**排序**。

4

排序的分类

按使用的存储器分

◆ **内排序**

若整个排序过程不需要访问外存便能完成，则称此类排序问题为内部排序

◆ **外排序**

若参加排序的记录数量很大，整个序列的排序过程不可能在内存中完成，则称此类排序问题为外部排序

排序性能

◆ **排序操作的性能评价指标**

✓ **时间性能**—排序过程中元素之间的比较次数与元素的移动次数。

- 本章主要考虑最差情况的比较次数

✓ **空间性能**—除了存放参加排序的元素之外，排序过程中所需要的其他辅助空间。

✓ **稳定性**—对于值相同的两个元素，排序前后的先后次序不变，则称该方法为稳定性排序方法， 否则，称为非稳定性排序方法

- 在所有可能的输入实例中，只要有一个实例使得该排序方法不满足稳定性要求，该排序方法就是非稳定的!

排序性能

◆ **约定**

✓ 只针对一个数据元素(关键字)序列讨论排序方法。

✓ 假设序列中具有n个数据元素(关键字)：

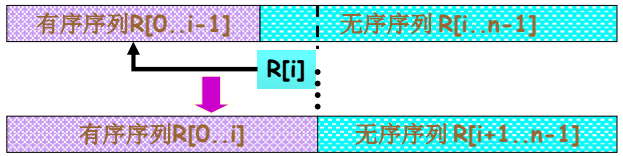
$k_1, k_2, k_3, \ldots, k_{n-1}, k_n$,存放于数组元素K[0],K[1], …, K[n-1]中

✓ 排序结果按照数据元素(关键字值)从小到大排列。

趟 —— 将具有n个数据元素(关键字)的序列转换为一个按照值的大小从小到大排列的序列通常要经过若干趟(Pass)。

2. 插入(insert)排序

◆ **一趟直接插入排序的基本思想**



实现“一趟插入排序”可分三步进行：

1. **在R[0..i-1]中查找R[i]的插入位置，R[0..j].key ≤ R[i].key < R[j+1..i-1].key;**
2. **将R[j+1..i-1]中的所有记录均后移一个位置;**
3. **将R[i] 插入(复制)到R[j+1]的位置上。**

2. 插入(insert)排序

◆ **核心思想**

$k_{i,j}$：表示第i趟排序结束时序列的第j个元素，1≤i≤n-1，1≤j≤n

第 i 趟排序将序列的第 i+1 个元素插入到一个大小为 i、且已经按值有序的子序列($k_{i-1,1}$, $k_{i-1,2}$, …, $k_{i-1,i}$)的合适位置，得到一个大小为 i+1、且仍然按值有序的子序列($k_{i,1}$, $k_{i,2}$, …, $k_{i,i+1}$)。

（1，4，8，12，6，11，…）

一个完整插入排序的过程

从第2个元素开始

初 始:	49	38	97	76	65	13	27	50
第1趟:	38	49	97	76	65	13	27	50
第2趟:	38	49	97	76	65	13	27	50
第3趟:	38	49	76	97	65	13	27	50
第4趟:	38	49	65	76	97	13	27	50
第5趟:	13	38	49	65	76	97	27	50
第6趟:	13	27	38	49	65	76	97	50
第7趟:	13	27	38	49	50	65	76	97

结果

例

temp ≥ K[j] ?
j=0

49 38 97 76 65 13 27 50

K[j+1]=K[j];
j=j−1;

… (若干趟后)

temp

65 38 49 65 76 97 13 27 50

K[j+1]=temp;

38 49 65 76 97 13 27 50

一趟结束了

插入排序算法实现

```
void insertSort(keytype k[], int n)
{
    int i, j;
    keytype temp;
    for (i = 1; i < n; i++)
    {
        temp = k[i];
        for (j = i - 1; j >= 0 && temp < k[j]; j--)
            k[j + 1] = k[j];
        k[j + 1] = temp;
    }
}
```

一趟排序

n-1趟排序

2. 插入排序

◆ 思考

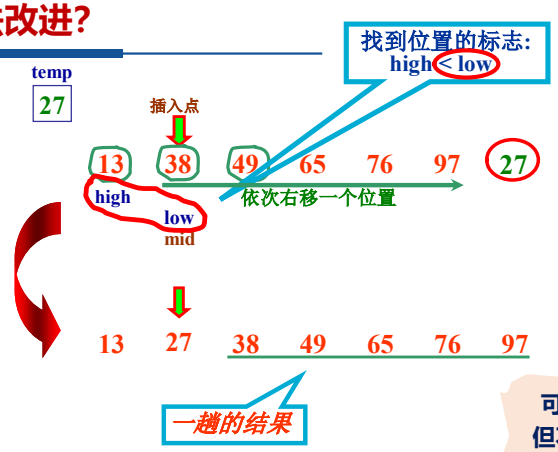
（1）排序的时间效率与什么直接有关?

答案 主要与排序过程中元素之间的比较次数直接有关。

（2）若原始序列为一个按值递增的序列，则排序过程中一共要经过多少次元素之间的比较?

答案 由于每一趟排序只需要经过一次元素之间的比较就可以找到被插入元素的合适位置，因此，整个n-1趟排序一共要经过n-1次元素之间的比较。

算法改进?



2. 插入排序

◆ 思考

（3）若原始序列为一个按值递减的序列，则排序过程中一共要经过多少次元素之间的比较?

答案 由于第i趟排序需要经过i次元素之间的比较才能找到被插入元素的合适位置，因此整个n-1趟排序一共要经过 $\sum_{i=1}^{n-1} i = n(n-1)/2$ 次元素之间的比较。

若以最坏的情况考虑，则插入排序算法的时间复杂度为$O(n^2)$。插入排序法是一种稳定性排序方法。

算法改进：采用折半查找确定插入位置

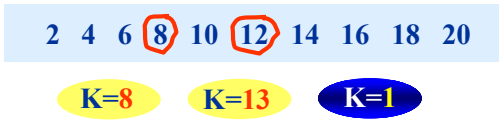
折半插入排序法

```
void insertBSort(keytype k[], int n){
    int i, j, low, high, mid;
    keytype temp;
    for (i = 1; i < n; i++)    {
        temp = k[i];
        low = 0;
        high = i-1;
        while (low <= high){
            mid = (low + high) / 2;
            if (temp < k[mid])
                high = mid-1;
            else
                low = mid + 1;
        }
        for (j = i-1; j >= low; j--)
            k[j + 1] = k[j];
        k[low] = temp;
    }
}
```

采用折半查找方法确定插入位置

练习1

请写一非递归算法，该算法在长度为 n、且元素按值严格递增排列的顺序表A[1..n]中采用折半插入法查找值不大于k的最大元素，若表中存在这样的元素，则算法返回该元素在表中的位置，否则返回0。

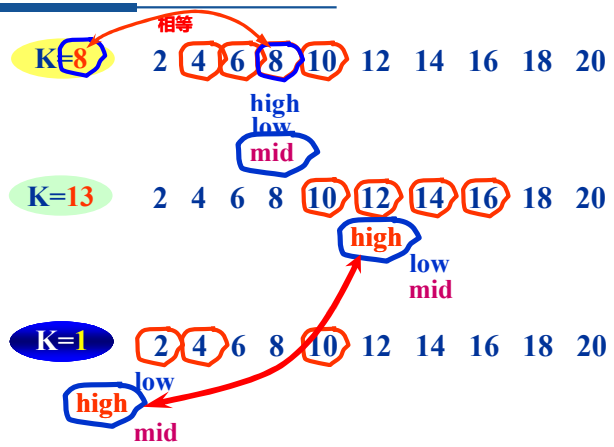


算法实现

```
int searchB(keytype a[], int n, keytype k)
{
    int low = 0, high = n - 1, mid;
    while (low <= high)
    {
        mid = (low + high) / 2;
        if (a[mid] == k)
            return mid;  //返回mid
        if (k > a[mid])
            low = mid + 1; //准备查找后半部分
        else
            high = mid-1; //准备查找前半部分
    }
    return high; //返回high
}
```

19

算法执行过程



18

插入排序算法分析

◆ 插入排序算法分析

稳定性：	稳定
时间代价：	• 最佳情况：n-1次比较，0交换，O(n) • 最差情况：比较和交换次数为$O(n^2)$ • 平均情况：$O(n^2)$
空间代价：	O(1)

20

内容提要

1. 排序的基本概念
2. 插入排序
3. 选择排序
4. 冒泡排序
5. 谢尔排序
6. 堆排序
7. 二路归并排序
8. 快速排序

3. 选择(Select)排序

◆ 核心思想

第 i 趟排序从序列的后 n-i+1个元素中选择一个值最小的元素，将其置于该 n-i+1个元素的最前面。

排序的趟号

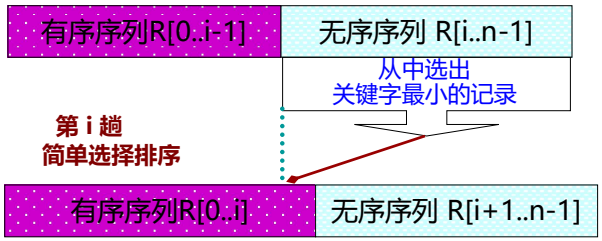
i	n-i+1
1	n
2	n-1
3	n-2
…	…

第 i 趟排序前，序列中未排好序的元素的个数

每一趟排序从序列中未排好序的元素中选择一个值最小的元素，将其置于这些未排好序的元素的最前面。

3. 选择(Select)排序

◆ 假设排序过程中，待排记录序列的状态为：



每一趟排序过程

13 97 38 27 65 35 80 75
j

13 27 38 97 65 35 80 75

13 27 35 38 65 97 80 75
已排好序 j

不必交换

完整的排序过程

初 始:	49	97	38	76	65	13	27	50
第1趟:	13	97	38	76	65	49	27	50
第2趟:	13	27	38	76	65	49	97	50
第3趟:	13	27	38	76	65	49	97	50
第4趟:	13	27	38	49	65	76	97	50
第5趟:	13	27	38	49	50	76	97	65
第6趟:	13	27	38	49	50	65	97	76
第7趟:	13	27	38	49	50	65	76	97

看一个完整的过程

结果

n-1趟

思考

（1）若原始序列为一个按值递增的序列，则排序过程中一共要经过多少次元素之间的比较?

（2）若原始序列为一个按值递减的序列，则排序过程中要经过多少次元素之间的比较?

答案： 无论原始序列为什么状态，第i趟排序都需要经过 n-i 次元素之间的比较，因此，整个排序过程中元素之间的比较次数为$\sum_{i=1}^{n-1}(n-i)=n(n-1)/2$。

结论： 选择排序法的元素之间的比较次数与原始序列中元素的分布状态无关。

时间复杂度为$O(n^2)$。

不稳定排序方法

算法实现

```
void selectSort(keytype k[], int n){
    int i, j, d;
    keytype temp;
    for (i = 0; i < n - 1; i++)
    {
        d = i;
        for (j = i + 1; j < n; j++)
            if (k[j] < k[d])
                d = j;
        if (d != i)
        {
            //最小值元素非未排序元素的第一个元素时
            temp = k[d];
            k[d] = k[i];
            k[i] = temp;
        }
    }
}
```

寻找值最小的元素,并记录其位置

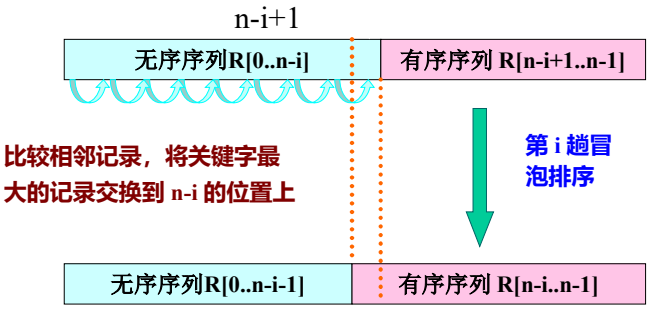
n-1趟排序

选择排序算法分析

稳定性:	不稳定
时间代价:	• 比较次数：$O(n^2)$ • 交换次数：n-1 • 总时间代价：$O(n^2)$
空间代价:	O(1)

4. 冒泡（bubble）排序

- 假设在排序过程中，记录序列R[0..n-1]的状态为：



完整的排序过程

n=8　n-i+1

初 始	49	97	38	13	27	50	76	65
第1趟	49	38	13	27	50	76	65	97
第2趟	38	13	27	49	50	65	76	97
第3趟	13	27	38	49	50	65	76	97
第4趟	13	27	38	49	50	65	76	97

排序结束？！　结果

13 27 38 49 50 65 76 97

j　j+1

排序总趟数可以小于n-1？

4. 冒泡（bubble）排序

特点：值大的元素往后“沉”，值小的元素向前“浮”

- 核心思想

第i趟排序对序列的前 n-i+1个元素从第一个元素开始依次作如下操作:相邻的两个元素比较大小，若前者大于后者,则两个元素交换位置，否则不交换位置。

效果：该n-i+1个元素中最大值元素移到该n-i+1个元素的最后。

设置一标志

Flag = { 0 某趟排序过程中无元素交换位置的动作；1 某趟排序过程中有元素交换位置的动作 }

每一趟趟排序前置flag为0，排序过程中出现元素交换动作，置flag为1。

如何知道某趟排序过程中是否有交换动作？

算法实现

```
void bubbleSort(keytype k[], int n)
{
    int i, j, flag = 1;
    keytype temp;
    for (i = n - 1; i > 0 && flag == 1; i--)
    {
        flag = 0; //每趟排序前标志flag置0
        for (j = 0; j < i; j++)
            if (k[j] > k[j + 1])
            {
                temp = k[j];
                k[j] = k[j + 1];
                k[j + 1] = temp; //交换两个元素的位置
                flag = 1;        //标志flag置1
            }
    }
}
```

33

算法性能分析

泊排序法的排序趟数与原始序列中数据元素的排列有关，因此，排序的趟数为一个范围，即[1..n-1]

什么情况下只要排序一趟

什么情况下要排序n-1趟

$O(n^2)$

泊排序方法比较适合于参加排序的序列的原始状态 基本有序的情况

泊排序法是稳定性排序方法。

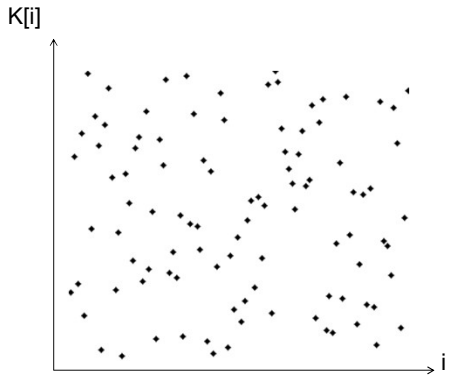
34

（改进）冒泡排序算法性能分析

稳定性：	**稳定**
时间代价：	• **最小时间代价为O(n)，最佳情况下只运行第一轮循环** • **一般情况下为$O(n^2)$**
空间代价：	**O(1)**

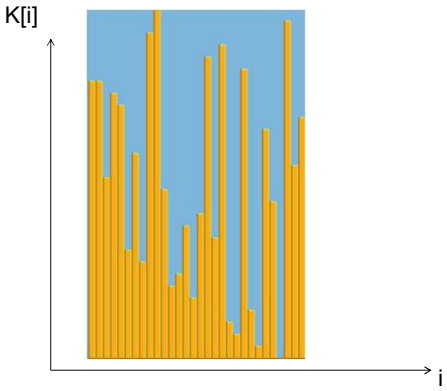
35

思考：什么算法



36

思考：什么算法



内容提要

1. 排序的基本概念
2. 插入排序
3. 选择排序
4. 冒泡排序
5. 谢尔排序
6. 堆排序
7. 二路归并排序
8. 快速排序

5. 谢尔(Shell)排序

- **对待排记录序列先作“宏观”调整，再作“微观”调整。**

将记录序列分成若干子序列，分别对每个子序列进行某种排序。

例如：将 n 个记录分成 d 个子序列：

{ R[1]，R[1+d]，R[1+2d]，…，R[1+kd] }
{ R[2]，R[2+d]，R[2+2d]，…，R[2+kd] }
…
{ R[d]，R[2d]，R[3d]，…，R[kd]，R[(k+1)d] }

其中，d 称为增量，它的值在排序过程中从大到小逐渐缩小，直至最后一趟排序减为 1

5. 谢尔(Shell)排序

缩小增量排序法

- **核心思想**

首先确定一个元素的间隔数gap。

将参加排序的元素按照gap分隔成若干个子序列(即分别把那些位置相隔为gap的元素看作一个子序列),然后对各个子序列采用某一种排序方法进行排序；此后减小gap值，重复上述过程，直到gap<1。

一种减小gap的方法：

$gap_1 = \lfloor n/2 \rfloor$

$gap_i = \lfloor gap_{i-1}/2 \rfloor$　　i = 2, 3, …

排序过程

初 始	49	97	38	50	76	65	13	27	25
第1趟 gap=4	25	65	13	27	49	97	38	50	76
第2趟 gap=2	13	27	25	50	38	65	49	97	76
第3趟 gap=1	13	25	27	38	49	50	65	76	97

结果

gap=1 13 25 27 38 49 50 65 76 97

子序列内采用冒泡排序方法

算法实现

```
void shellSort(keytype k[], int n){
    int i, j, flag, gap = n;
    keytype temp;
    while (gap > 1) {
        gap = gap / 2;
        do {
            flag = 0; //每趟排序前,标志flag置0
            for (i = 0; i < n-gap; i++){
                j = i + gap;
                if (k[i] > k[j]) {
                    temp = k[i];
                    k[i] = k[j];
                    k[j] = temp;
                    flag = 1;
                }
            }
        } while (flag != 0);
    }
}
```

```
void shellSort(int v[], int n)//from K & R
{
    int gap, i, j, temp;
    for (gap = n / 2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j = i - gap;
                 j >= 0 && v[j] > v[j + gap];
                 j -= gap)
            {
                temp = v[j];
                v[j] = v[j + gap];
                v[j + gap] = temp;
            }
}
```

from K & R

算法实现（续）

```
void shellSort(keytype k[], int n)
{
    int i, j, gap = n;
    keytype temp;
    while (gap > 1) {
        gap = gap / 2;

        // 使用插入排序实现子序列排序
        for (i = gap; i < n; i++){
            temp = k[i];
            for (j = i; j >= gap && k[j - gap] > temp; j -= gap)
                k[j] = k[j - gap];
            k[j] = temp
        }
    }
}
```


5. 谢尔(Shell)排序

谢尔排序中gap的取法:

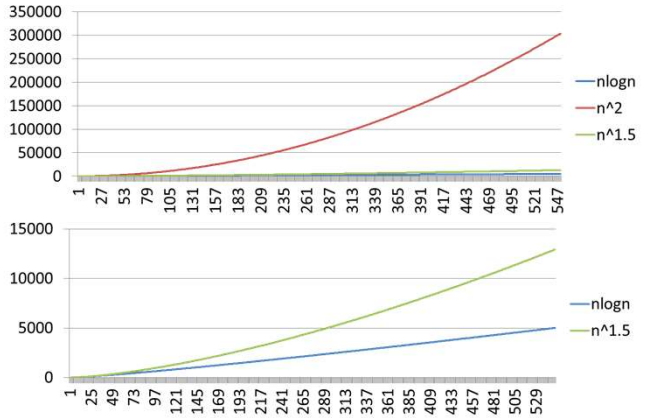
- ✓ Shell最初的方案是 gap= n/2, gap=gap/2，直到gap=1.
- ✓ Knuth的方案是gap = gap/3+1
- ✓ 其它方案有：都取奇数为好；或gap互质为好等等。

Shell排序算法分析

例 设待排序的表有4个记录,其关键字分别为{3 2 2* 5},
说明采用希尔排序方法进行排序的过程。

它是一种不稳定性排序方法

不同时间复杂度的性能分析



Shell排序算法分析

稳定性:	**不稳定**
时间代价:	**$O(n\log_2 n)$与$O(n^2)$之间** **(通常<$O(n^{3/2})$)**
空间代价:	**O(1)**

6. 堆（Heap）排序

由John Williams提出

- **堆(Heap)的定义**

n个元素的序列($k_1, k_2, \ldots, k_n$)，当且仅当满足

(1) $\begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases}$ 或者 (2) $\begin{cases} k_i \leq k_{2i} \\ k_i \leq k_{2i+1} \end{cases}$ 其中， $i=1, 2, 3, \ldots, \lfloor n/2 \rfloor$

称该序列为一个堆积(heap)，简称堆。

称满足条件(1)的堆为大顶堆，称满足条件(2)的堆为小顶堆。

下面以大顶堆为例

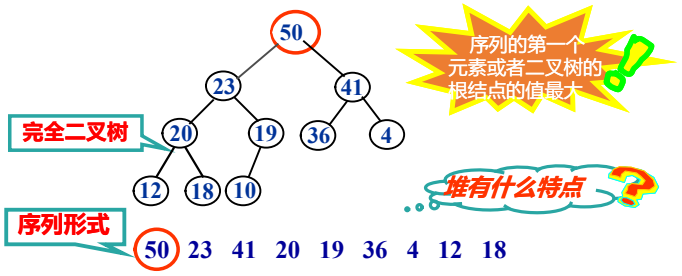
例1

50 23 41 20 19 36 4 12 18

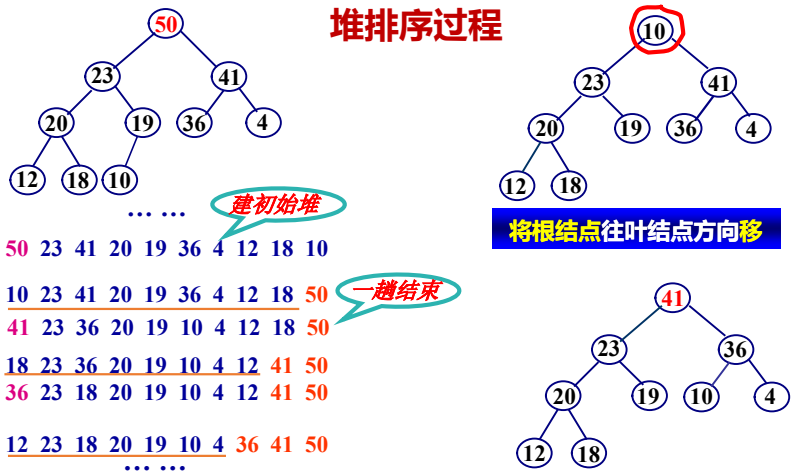
一个(大顶)堆

堆的性质

◆ 堆是一棵完全二叉树，二叉树中任何一个分支结点的值都大于或者等于它的孩子结点的值，并且每一棵子树也满足堆的特性



堆排序过程



堆排序

◆ **核心思想**

第 i 趟排序将序列的前n-i+1个元素组成的子序列转换为一个堆，然后将堆的第一个元素与堆的最后那个元素交换位置。

◆ **步骤**

建初始堆

1. 将原始序列转换为第一个堆。
2. 将堆的第一个元素与堆的最后那个元素交换位置。(即“去掉”最大值元素)
3. 将“去掉”最大值元素后剩下的元素组成的子序列重新转换一个新的堆。
4. 重复上述过程的第2至第3步n−1次。

调整子算法

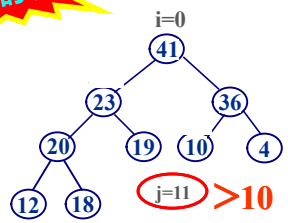
```
void adjust(keytype k[], int i, int n){
    int j;
    keytype temp;
    temp = k[i];
    j = 2 * i + 1;
    while (j < n) {
        if (j + 1 < n && k[j] < k[j + 1])
            j++;
        if (temp < k[j]) {
            k[(j - 1) / 2] = k[j];
            j = 2 * j + 1;
        }
        else
            break;
    }
    k[(j - 1) / 2] = temp;
}
```

功能：向下调整结点 i 的位置，使得其祖先结点值都比其大。如果一棵树仅根结点 i 不满足堆条件，通过该函数可将其调整为一个堆

K：序列
i：被调整的二叉树的根的序号
n：被调整的二叉树的结点数目

调整过程

一个新的堆状



temp
10

n=10

```
void adjust(keytype k[], int i, int n){
    int j;
    keytype temp;
    temp = k[i];
    j = 2 * i + 1;
    while (j < n) {
        if (j + 1 < n && k[j] < k[j + 1])
            j++;
        if (temp < k[j]) {
            k[(j - 1) / 2] = k[j];
            j = 2 * j + 1;
        }
        else
            break;
    }
    k[(j - 1) / 2] = temp;
}
```

功能：向下调整结点i的位置，使得其祖先结点值 都比其大。如果一棵树仅根节点不满足堆条件，通过该函数可将其调整为一个堆。

53

完整的堆排序过程

原始	75	36	18	53	80	30	48	90	15	37
第1趟	37	80	48	53	75	30	18	36	15	90
第2趟	15	75	48	53	37	30	18	36	80	90
第3趟	15	53	48	36	37	30	18	75	80	90
第4趟	18	37	48	36	15	30	53	75	80	90
第5趟	18	37	30	36	15	48	53	75	80	90
第6趟	15	36	30	18	37	48	53	75	80	90
第7趟	15	18	30	36	37	48	53	75	80	90
第8趟	15	18	30	36	37	48	53	75	80	90
第9趟	15	18	30	36	37	48	53	75	80	90

初始堆

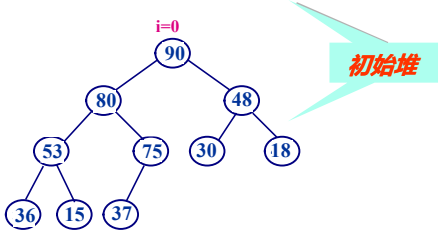
55

五. 建立初始堆

从二叉树的最后那个分支结点(编号为 i=⌊n/2⌋)开始，依次将编号为 i 的结点为根的二叉树**转换**为一个堆，每转换一棵子树，做一次 i -1，重复上述过程，直到将i=1的结点为根的二叉树转换为堆。

75	36	18	53	80	30	48	90	15	37
90	80	48	53	75	30	18	36	15	37

例



初始堆

堆排序算法

```
void heapSort(keytype k[], int n)
{
    int i,
        keytype temp;
    for (i = n / 2 - 1; i >= 0; i--)
        adjust(k, i, n);

    for (i = n - 1; i >= 1; i--)
    {
        temp = k[i];
        k[i] = k[0];
        k[0] = temp;
        adjust(k, 0, i);
    }
}
```

建初始堆

具体排序

n-1趟排序

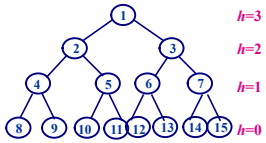
$O(n\log_2 n)$

56

时间复杂度分析

```
for(i=n/2-1;i>=0;i--)
    adjust(K,i,n);
```

建初始堆复杂度



第h层节点数 $\le \left\lceil 2^{(\lg n-h)-1} \right\rceil = \left\lceil \frac{2^{\lg n}}{2^{h+1}} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$

时间复杂度= $\sum_{h=0}^{\lceil \lg n \rceil} \frac{n}{2^{h+1}} O(h) = O\left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^{h+1}} \right)$

$\le O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$

$= O(n)$

堆排序算法分析

稳定性:	不稳定
时间代价:	O(nlog$_2$n)
空间代价:	O(1)

7. 二路归并（Merge）排序

由John von Neumann提出

◆ **什么是二路归并**

将两个位置相邻、并且各自按值有序的子序列合并为一个按值有序的子序列的过程称为二路归并。

$(K_s, K_{s+1}, K_{s+2}, \ldots\ldots, K_u)$ $(K_{u+1}, K_{u+2}, K_{u+3}, \ldots\ldots, K_v)$

$(X_s, X_{s+1}, X_{s+2}, X_{s+3}, \ldots\ldots, X_v)$

其中 $K_s \le K_{s+1} \le K_{s+2} \le \ldots\ldots \le K_u$

$K_{u+1} \le K_{u+2} \le K_{u+3} \le \ldots\ldots \le K_v$

$X_s \le X_{s+1} \le X_{s+2} \le X_{s+3} \le \ldots\ldots \le X_v$

例：归并过程

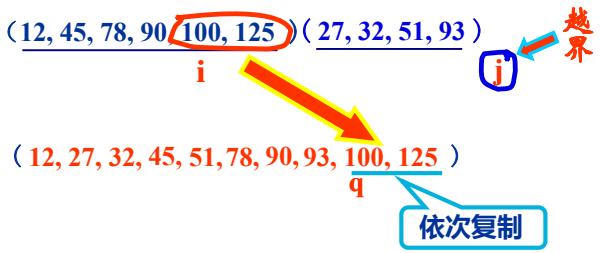
$\ldots K_s, K_{s+1}, K_{s+2}, K_{s+3}, K_{s+4}, K_{s+5}$ $K_{s+6}, K_{s+7}, K_{s+8}, K_{s+9} \ldots$

(12, 45, 78, 90, 100, 125) (27, 32, 51, 93)

(12, 27, 32, 45, 51, 78, 90, 93, 100, 125)

二路归并

例：归并过程（续）

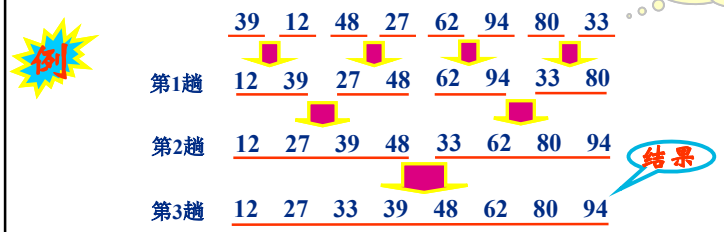


7. 二路归并（Merge）排序

◆核心思想

第 i 趟排序将序列的 $\lfloor \frac{n}{2^{i-1}} \rfloor$ 个长度为 2^{i-1} 的按值有序的子序列依次两两合并为 $\lfloor \frac{n}{2^{i}} \rfloor$ 个长度为 2^{i} 的按值有序的子序列。

分别看成8个大小为1的有序子序列



合并算法实现

```
void merge(keytype x[], keytype tmp[],
           int left, int leftend, int rightend)
{
    int i = left, j = leftend + 1, q = left;
    while (i <= leftend && j <= rightend)
        if (x[i] <= x[j])
            tmp[q++] = x[i++];
        else
            tmp[q++] = x[j++];

    while (i <= leftend)
        tmp[q++] = x[i++];
    while (j <= rightend)
        tmp[q++] = x[j++];
    for (i = left; i <= rightend; i++)
        x[i] = tmp[i];
}
```

功能 将两个位置相邻且按值有序子序列合并为一个按值有序的序列

复制第一个子序列的剩余部分

复制第二个子序列的剩余部分

将合并后内容复制回原数组

归并排序算法

本质上它是一种**分治算法**（divide and conquer algorithm）

```
void mergeSort(keytype k[], int n){
    keytype *tmp;
    tmp = (keytype *)malloc(sizeof(keytype) * n);
    if (tmp != NULL)    {
        mSort(k, tmp, 0, n - 1);
        free(tmp);
    }
    else printf("No space for tmp array!!!\n");
}
void mSort(keytype k[], keytype tmp[], int left, int right){
    int center;
    if (left < right)  {
        center = (left + right) / 2;
        mSort(k, tmp, left, center);
        mSort(k, tmp, center + 1, right);
        merge(k, tmp, left, center, right);
    }
}
```

时间复杂度 $O(n\log_2 n)$

空间复杂度 $O(n)$

merge排序是**稳定的**排序。

7. 二路归并（Merge）排序

◆ 归并排序算法分析

稳定性：	稳定
时间代价：	$O(n\log_2 n)$ （不依赖于原始数据的输入情况，最大、最小及平均时间代价均为$O(n\log_2 n)$）
空间代价：	$O(n)$

8. 快速（Quick）排序

快速排序算法最早由图灵奖获得者Tony Hoare设计出来，被列为20世纪十大算法之一。也是目前实践中已知最快的排序算法。

◆ 核心思想

从当前参加排序的元素中任选一个元素(通常称之为**分界元素pivot**)与当前参加排序的那些元素进行比较,凡是小于分界元素的元素都移到分界元素的前面,凡是大于分界元素的都移到分界元素的后面,分界元素将当前参加排序的元素分成前后两部分,而分界元素处在排序的最终位置。然后，分别对这两部分中大小大于1的部分重复上述过程,直到排序结束。

递归过程

划分元素、基准元素、枢轴、轴值、支点

可以选第一个或者最后一个、或位置居中的那个元素作为分界元素

每一次排序 至少 可以
确定一个元素的 最终位置 !

内容提要

1. 排序的基本概念
2. 插入排序
3. 选择排序
4. 冒泡排序
5. 谢尔排序
6. 堆排序
7. 二路归并排序
8. 快速排序

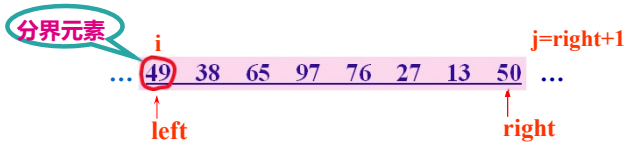
8. 快速（Quick）排序

◆ **算法中用到的变量**

left：当前参加排序的那些元素的第一个元素在序列中的位置，初始值为0

right：当前参加排序的那些元素的最后那个元素在序列中的位置，初始值为n-1

i，j：两个位置变量，初始值分别为left 与right+1。



8. 快速（Quick）排序

◆ **快速排序步骤**

分界元素K[left]

(1) 反复执行动作i+1→i，直到K[left]≤K[i]或者i=right。

反复执行动作j−1→j，直到K[left]≥K[j]或者j=left。

(2) 若i<j，则K[i]与K[j]交换位置，转到第1步。

(3) 若i≥j，则K[left]与K[j]交换位置,到此,分界元素K[left]的最终位置已经确定，然后对被K[left]分成的两部分中个数大于1 的部分重复上述过程，直到排序结束。

j

例

排序结束

				s	t		
13	27	38	49	50	65	76	97
49	38	65	97	76	27	13	50

原始序列

步骤

(1) 反复执行动作i+1→i，直到K[s]≤K[i]或者i=t。
反复执行动作j-1→j，直到K[s]≥K[j]或者j=s。

(2) 若i<j，则K[i]与K[j]交换位置，转到第1步。

(3) 若i≥j，则K[s]与K[j]交换位置，到此,分界元素K[s]的最终位置已经确定，然后对被K[s]分成的两部分中大小大于1的部分重复上述过程，直到排序结束。

算法实现（本例中找中间的元素作为分界元素）

```
void qsort(keytype v[], int left, int right)
{
    int i, last;

    if (left >= right)
        return;
    swap(v, left, (left + right) / 2); //move partition elem to v[0]
    last = left;
    for (i = left + 1; i <= right; i++) //partition
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last); //restore partition elem
    qsort(v, left, last);
    qsort(v, last + 1, right);
}
```

```
void swap(keytype v[], int i, int j)
{
    keytype tmp;
    tmp = v[i];
    v[i] = v[j];
    v[j] = tmp;
}
```

```
//主算法
void quickSort(keytype k[], int n)
{
    qsort(k, 0, n - 1);
}
```


快速排序算法分析

稳定性：	不稳定
最差情况：	• 时间代价：$O(n^2)$ • 空间代价：$O(n)$
最佳情况：	• 时间代价：$O(n\log_2 n)$ • 空间代价：$O(\log_2 n)$
平均情况：	• 时间代价：$O(n\log_2 n)$ • 空间代价：$O(\log_2 n)$

使用库函数qsort进行快速排序

void qsort(void *base, size_t num, size_t wid,
int (*comp)(const void *e1, const void *e2));

qsort：快速排序函数，可以对一组数据进行排序（头文件：stdlib.h）

base: 需要排序的数组首地址

num: 需要排序的数组有效成员的个数

wid: 每个数组成员的字节数

comp: 比较函数指针，传递自己编写的比较函数，根据该函数的返回值决定如何移动数组：返回负数、正数和0，分别表示第一个参数先于、后于和等于第二个参数

七种排序算法总结

排序方法	平均情况	最好情况	最坏情况	辅助空间	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
希尔排序	$O(n\log n) \sim O(n^2)$	$O(n^{1.3})$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	不稳定
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	稳定
快速排序	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n) \sim O(n)$	不稳定

◆ **从算法性质来看**
- 简单算法：冒泡、选择、插入
- 改进算法：谢尔、堆、归并、快速

◆ **从时间复杂度来看**
- 平均情况：后3种改进算法 > 谢尔 > 简单算法
- 最好情况：冒泡和插入排序要更好一些
- 最坏情况：堆和归并排序要好于快速排序及简单排序

◆ **从空间复杂度来看**
- 归并排序有额外空间要求，快速排序也有相应空间要求，堆排序则基本没有。

◆ **从稳定性来看**
- 除了简单排序，归并排序不仅速度快，而且还稳定

其他排序算法：桶(Bucket)排序法*（计数排序）

◆ **核心思想**

假设$a_1,a_2...a_n$由小于M的正整数组成，桶排序的基本原理是使用一个大小为M的数组C(初始化为0，称为桶bucket)，当处理a_i时，使$C[a_i]$增1。最后遍历数组C输出排序后的表。

基本思想是由E.J.Issac R.C.Singleton提出来。桶排序并不是比较排序，不受到O(n log n) 下限的影响，是最简单、最快的排序，时间复杂度为O(M+N)。

77

内容提要

1. 排序的基本概念
2. 插入排序
3. 选择排序
4. 冒泡排序
5. 谢尔排序
6. 堆排序
7. 二路归并排序
8. 快速排序

78

单选题 1分

若对序列(2, 12, 16, 70, 5, 10)按值从小到大进行排序，前三趟排序的结果分别为：

第1趟排序的结果：(2, 12, 16, 5, 10, 70)，

第2趟排序的结果：(2, 12, 5, 10, 16, 70)，

第3趟排序的结果：(2, 5, 10, 12, 16, 70)，

则由此可以断定，该排序过程采用的排序方法是

A 插入排序

C 冒泡排序

B 选择排序

D 快速排序

79

小结：排序

◆ **排序的基本概念**
- ✓ 排序
- ✓ 性能分析：**稳定性**、时间复杂度、空间复杂度

◆ **简单排序算法**
- ✓ 插入排序
- ✓ 选择排序
- ✓ 冒泡排序

◆ **改进排序算法**
- ✓ 谢尔排序
- ✓ 堆排序
- ✓ 二路归并排序
- ✓ 快速排序

80