

## 插入排序-最简单的排序方法

```

package InsertSort;
//从原理上可以看出插入排序是一种内部排序方法，不占用外部空间。
//插入排序的基本思想就是把一个无序的数组看成两部分，一部分是数目为 1 的有序序列，另一部分是数目为 n-1 的无序序列。
//把无序序列的每个元素与有序的比较，找到合适的位置插入进入。
public class InsertSort {
    public static void main(String[] args) {
        int a[] = { 2, 5, 1, 4, 9, 6, 8, 3, 7, 0 };
        sort(a);
    }
    static void sort(int a[]) {
        // 假设第一个是已经排序好的！实际也是如此，一个元素当然是排序好的！
        从后面开始遍历
        for (int i = 1; i < a.length; i++) {
            // 比较从 0 开始到 i 的大小，找到合适的地方插进去
            for (int j = 0; j < i; j++) { //遍历所有有序的数，找个合适的位置
                //这个地方很有讲究。如果上面的 j 是从 i 到 0 的，那么下面你就是
                从

                //大小号能决定排序的方向。
                if (a[i] < a[j]) {
                    int tmp = a[i];
                    // j 后面的元素全部向后移动一位，一直到 i 停止。
                    for (int m = i; m > j; m--) {
                        a[m] = a[m - 1];
                    }
                    a[j] = tmp;
                    break;
                }
            }
        }
        for (int b : a) {
            System.out.print(b + " ");
        }
        //      System.out.print(Arrays.toString(a));
    }
}

```

```

InsertSort.java
8 public static void main(String[] args) {
9     int a[] = { 2, 5, 1, 4, 9, 6, 8, 3, 7, 0 };
10    sort(a);
11 }
12 static void sort(int a[]) {
13     // 假设第一个是已经排序好的! 实际也是如此, 一个元素当然是排序好的! 从后面开始遍历
14     for (int i = 1; i < a.length; i++) {
15         // 比较从0开始到i的大小, 找到合适的地方插进去
16         for (int j = 0; j < i; j++) {
17             // 这个地方很有讲究。如果上面的j是从i到0的, 那么下面你就是从
18             // 有待研究。貌似大小号能决定排序的方向。但是不利于排序的算法执行的效率。
19             if (a[i] < a[j]) {
20                 int tmp = a[i];
21                 // j后面的元素全部向后移动一位, 一直到i停止。
22                 for (int m = i; m > j; m--) {
23                     a[m] = a[m - 1];
24                 }
25                 a[j] = tmp;
26             }
27         }
28     }
29     for (int b : a) {
30         System.out.print(b + " ");
31     }
32 }

```

Problems @ Javadoc Declaration Console

<terminated> InsertSort [Java Application] /usr/local/eclipse/jre/bin/java (2018年8月3日 下午11:54:17)

0 1 2 3 4 5 6 7 8 9

```

8     for(int j=0;j<i;j++){//遍历所有有序的数, 找个合适的位置
9         if(a[i]>a[j]){
10             for(int m=i;m>j;m--){
11                 a[m]=a[m-1];
12             }
13             a[j]=tem;
14             break;
15         }
16     }
17 }
18 System.out.print(Arrays.toString(a));

```

Problems @ Javadoc Declaration Console

<terminated> InsertSort [Java Application] /usr/local/eclipse-java/jre/bin

[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

# 希尔排序

## 1. 问题分析

希尔排序是一种升级版的插入排序！本质上是插入排序，但是优化了一下。

## 2. 具体设计过程

### 2.1 设计思路

按照某种规则将数组分成多组，然后在进行插入排序！有两种方案完成分组合并的问题，第一种是递归算法！第二种是采取 for 循环，使得分组逐渐变化！本实验报告采取递归算法。

### 2.2 程序设计流程图

流程图 + 说明

第一次：d=5，排序一次

第二次：d=2，排序一次

第三次：d=1，排序一次

### 2.3 函数实现说明

此处对程序中的一些关键函数进行说明，例：

(1) void display () 棋盘绘制函数

功能：函数display()通过for循环，对棋盘界面进行了绘制

用法：此函数调用方式为void display(char board[][SIZE]);

说明：参数是一个二维数组，size为定义的长度。值为8

返回值：无

对 ShellSort(a,d);采用递归！并且采用指针传递，保证修改成功。

## 3. 程序运行说明

说明整个程序的运行环境、数据输入的格式及限制、输出、条件及其它相关说明。

VC6.0.发生一特殊情况，在N=11的时候，a[9]=0，最后排序的时候0会出现成11，0消失了。

## 4. 程序运行结果

运行截图 + 说明

```

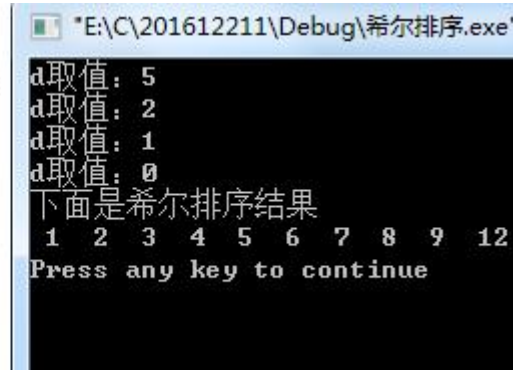
n()

a[N]={NULL,1,5,4,7,8,3,2,9,12,6};
d=N;
lSort(a,d);
tf("下面是希尔排序结果\n");
nt i=1;i<N;i++)

printf(" %d ",a[i]);

f("\n");

```



## 5.结论和心得

### 附录：源代码

```

#include<stdio.h>
#define N 11
int ShellSort(int *a,int n)
{
    int i,j,k;
    int d=n/2;
    printf("d 取值: %d\n",d);
    if (d<1) return 1;
    else
    {
        for(i=1;i<=d;i++)
        {
            for(j=i+d;j<N;j=j+d)
            {
                if(a[j]<a[j-d]){
                    a[0]=a[j];
                    //经过调试发现应该是 k-d>0
                    for(k=j;k>0&&a[k]<a[k-d];k=k-d)
                        a[k]=a[k-d];
                    a[k]=a[0];
                }
            }
        }
        ShellSort(a,d);//程序递归
    }
    return 0;}

int main()
{
    int a[N]={NULL,1,5,4,7,8,3,2,9,12,6};
    int d=N;
    ShellSort(a,d);
    printf("下面是希尔排序结果\n");
    for(int i=1;i<N;i++)
    {

```

```

        printf(" %d ",a[i]);
    }
    printf("\n");
    return 0;
}

```

结论：在 N 为 11 的时候，只要 a[9] 是 0，最后排序的结果就是 11，而且排序正常。

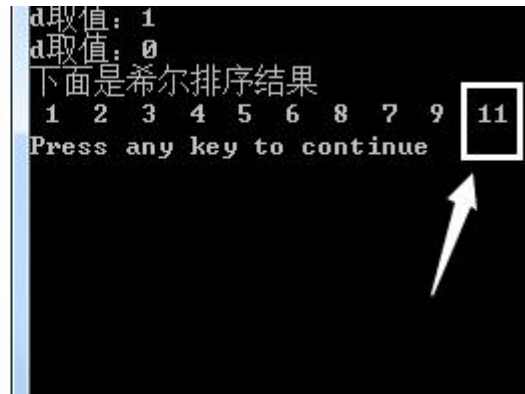
```

i[]={NULL,1,5,4,7,8,3,2,9,0,6};
i;
rt(a,d);
"下面是希尔排序结果\n");
i=1;i<N;i++)

tf(" %d ",a[i]);

\n");

```



```

d取值: 1
d取值: 0
下面是希尔排序结果
1 2 3 4 5 6 8 7 9 11
Press any key to continue

```

下面是 deepin 系统下面的 Code：： Blocks 软件下运行的代码：

```

#include<stdio.h>

#define N 11

int ShellSort(int *a,int n)
{

    for(int i1=1;i1<N;i1++)
    {
        printf(" %d ",a[i1]);
    }

    printf("\n");

    int i=0,j=0,k=0;

    int d=n/2;

    printf("d get value:  %d\n",d);

    if (d<1) return 1;

    else

    {

        for(i=1;j<=d;i++)

```

```
    {
        for(j=i+d;j<N;j=j+d)
        {
            if(a[j]<a[j-d]){
                a[0]=a[j];
                //经过调试发现应该是 k-d>0
                for(k=j;k>0&& a[k]<a[k-d];k=k-d)
                    a[k]=a[k-d];
                a[k]=a[0];
            }
        }
    }
    ShellSort(a,d);//程序递归
}

return 0;}

int main()
{
    int a[N]={0,1,5,4,7,8,3,2,9,12,6};
    int d=N;
    ShellSort(a,d);
    printf("The following is the result of Hill sorting:\n");
    for(int i=1;i<N;i++)
    {
        printf(" %d ",a[i]);
    }

    printf("\n");
    return 0;
}
```

上面的程序修改了 NULL 为 0

运行结果如下：

```

/home/deepin/xiersort
1 5 4 7 8 3 2 9 12 6
d get value: 5
1 32765 4 7 6 3 5 9 12 8
d get value: 2
1 7 4 3 5 9 6 8 12 32765
d get value: 1
1 4 3 5 7 6 8 9 12 32765
d get value: 0
The following is the result of Hill sorting:
Segmentation fault
Process returned 139 (0x8B)   execution time : 0.001 s

```

Deepin 下面的 shell+gcc 也是报错！

```

deepin@deepin-PC: /media/deepin/文件/File/C$ gcc xiersort.c -o shellSort
deepin@deepin-PC: /media/deepin/文件/File/C$ ./shellSort
1 5 4 7 8 3 2 9 12 6
d get value: 5
1 32766 4 7 6 3 5 9 12 8
d get value: 2
1 7 4 3 5 9 6 8 12 32766
d get value: 1
1 4 3 5 7 6 8 9 12 32766
d get value: 0
The following is the result of Hill sorting:
段错误
deepin@deepin-PC: /media/deepin/文件/File/C$

```

上面错误的原因是因为 经过调试发现应该是 `k-d>0`

调试代码如下：

```

#include<stdio.h>

#define N 11

#define null 0

void print(int a[], int n){
    for(int i1=1;i1<n;i1++){
        printf(" %d ",a[i1]);
    }
    // printf("\n");
}

//int ShellSort(int *a, int n){
int ShellSort(int a[], int n){

```

```
print(a, n);
printf("    n:%d:", n);
printf("\n");
    int i=0, j=0, k=0;
    int d=n/2;
    printf("d get value: %d\n", d);
    if (d<1) {
        print(a, n);
        return 1;
    }

    else{

        for(i=1; i<=d; i++) {
            for(j=i+d; j<N; j=j+d) {
                printf("d=%d    j=%d    j-d=%d    a[%d]=%d\n", d, j, j-d, j, a[j], j-d, a[j-d]);
                if(a[j]<a[j-d]) {
                    printf("a[%d]=%d < a[%d]=%d\n", j, a[j], j-d, a[j-d]);

                    a[0]=a[j];
                    printf("a[0]=%d a[%d]=%d\n", a[0], j, a[j]);
                    //下面这个地方不是 k>0, 而是 k-d>0
                    for(k=j; k-d>0&& a[k]<a[k-d]; k=k-d) {
                        printf("%d    a[k=%d]=%d\n", k, a[k], k-d, a[k-d]);
                        a[k]=a[k-d];
                        printf("%d    a[k=%d]=%d\n", k, a[k], k-d, a[k-d]);
                    }
                }
            }
        }
    }
}
```



```
        }
        a[k]=a[0];

        printf("a[%d]=%d\n", k, a[k], k-d, a[k-d]);
    }
}

ShellSort(a, d); //程序递归
}

return 0;
}

int main() {
    int a[N]={0, 1, 5, 4, 7, 8, 3, 2, 9, 12, 6};
    ShellSort(a, N); //a 即表示数组 a 的首地址

    printf("The following is the result of Hill sorting:\n");

    for(int i=1; i<N; i++) {
        printf(" %d ", a[i]);
    }

    printf("\n");

    return 0;
}
```

```

deepin@deepin-PC: /media/deepin/文件/File/C$ g++ xiersort.c -o shellSort
deepin@deepin-PC: /media/deepin/文件/File/C$ ./shellSort
1 5 4 7 8 3 2 9 12 6 n:11:
d get value: 5
d=5 j=6 j-d=1 a[6]=3 a[1]=1
d=5 j=7 j-d=2 a[7]=2 a[2]=5
a[7]=2 < a[2]=5
a[0]=2 a[7]=2
7 a[k=7]=2 a[k-d=2]=5
7 a[k=7]=2 a[k-d=2]=5
2 a[k=2]=5 a[k-d=-3]=32766
2 a[k=2]=5 a[k-d=-3]=32766
a[-3]=32766 a[-8]=5
d=5 j=8 j-d=3 a[8]=9 a[3]=4
d=5 j=9 j-d=4 a[9]=12 a[4]=7
d=5 j=10 j-d=5 a[10]=6 a[5]=8
a[10]=6 < a[5]=8
a[0]=6 a[10]=6
10 a[k=10]=6 a[k-d=5]=8
10 a[k=10]=6 a[k-d=5]=8
a[5]=8 a[0]=6
The following is the result of Hill sorting:
1 5 4 7 8 3 2 9 12 6

```

经过调试发现应该是  $k-d>0$

修改过后排序正常:

```

d get value: 0
The following is the result of Hill sorting:
1 2 3 4 5 6 7 8 9 12

```

Gedit+shell 应该是 linux 系统非常好的调试工具了。可以看到上次的调试结果，互相对比。

Manjaro 下面的代码:

```

#include<stdio.h>

#define N 11

int ShellSort(int *a,int n)
{
    for(int i1=1;i1<N;i1++){
        printf(" %d ",a[i1]);
    }
    printf("\n");

    int i=0,j=0,k=0;

    int d=n/2;

    printf("d get value: %d\n",d);

    if (d<1) return 1;

```

```

else{
    for(i=1;i<=d;i++){
        for(j=i+d;j<N;j=j+d){
            if(a[j]<a[j-d]){
                a[0]=a[j];

                //经过调试发现应该是  $k-d>0$  ,也不得不服, 在 manjaro 下竟然编译执行成功! 而且竟然还排序正确! 真是奇葩

                for(k=j;k>0&& a[k]<a[k-d];k=k-d)

                    a[k]=a[k-d];

                a[k]=a[0];
            }
        }
    }

    ShellSort(a,d);//程序递归
}

return 0;

}

int main()
{
    int a[N]={0,1,5,4,7,8,3,2,9,12,6};

    ShellSort(a,N);

    printf("The following is the result of Hill sorting:\n");

    for(int i=1;i<N;i++)
    {
        printf(" %d ",a[i]);
    }

    printf("\n");

    return 0;
}

```

```
[yz@yz-pc C]$ gcc xiersort.c
[yz@yz-pc C]$ ./a.out
1 5 4 7 8 3 2 9 12 6
d get value: 5
1 2 4 7 6 3 5 9 12 8
d get value: 2
1 2 4 3 5 7 6 8 12 9
d get value: 1
1 2 3 4 5 6 7 8 9 12
d get value: 0
The following is the result of Hill sorting:
1 2 3 4 5 6 7 8 9 12
[yz@yz-pc C]$
```

制定输出文件名

```
[yz@yz-pc C]$ gcc xiersort.c -o shellSort
[yz@yz-pc C]$ ./shellSort
1 5 4 7 8 3 2 9 12 6
d get value: 5
1 2 4 7 6 3 5 9 12 8
d get value: 2
1 2 4 3 5 7 6 8 12 9
d get value: 1
1 2 3 4 5 6 7 8 9 12
d get value: 0
The following is the result of Hill sorting:
1 2 3 4 5 6 7 8 9 12
```

排序正常！

```
[yz@yz-pc C]$ gcc xiersort.c -o shellSort
xiersort.c: 在函数 'main' 中:
xiersort.c:30:12: 警告: initialization of 'int' from 'void *' makes in
pointer without a cast [-Wint-conversion]
    int a[N]={NULL,1,5,4,7,8,3,2,9,12,6};
              ^
xiersort.c:30:12: 附注: (在 'a[0]' 的初始化附近)
[yz@yz-pc C]$ gcc xiersort.c -o shellSort
xiersort.c: 在函数 'main' 中:
xiersort.c:30:12: 错误: 'null' 未声明 (在此函数内第一次使用)
    int a[N]={null,1,5,4,7,8,3,2,9,12,6};
              ^
xiersort.c:30:12: 附注: 每个未声明的标识符在其出现的函数内只报告一次
[yz@yz-pc C]$
```

很显然编译器是不支持 null 和 NULL 的。

## 选择排序

插入排序的基本思想：遍历未排序的所有元素与标记元素（第一次排序可以把首位元素做为标记位，然后下一次排序把标记位推进一位）比较，如果出现比标记元素大或者小的，重新把此元素作为标记元素，一直到排序结束，那么标记元素即为极值，应该放到未排序的第一位，从未排序的元素中取最值，的未排序

步骤如下：

1. 未排序的第一位为标记元素
2. 从未排序的其他元素中找出比标记元素大的元素，记为新标记元素。
3. 一直到未排序的元素遍历完了，把最终的标记元素与未排序的首位元素交换。
4. 重复 1-3 过程

```
public class SelectSort {  
  
    public static void main(String[] args) {  
  
        int a[] = {1,5,6,4,2,3,8,9,7,0};  
  
        selectSort(a);  
  
        for(int b :a){  
  
            System.out.print(b+ " ");  
  
        }  
  
    }  
  
    public static void selectSort(int[]a)  
  
    {  
  
        int minIndex=0;  
  
        int temp=0;
```

```
    if((a==null)|| (a.length==0))

        return;

    for(int i=0;i<a.length;i++)

    {

        minIndex=i;//无序区的最小数据数组下标

        for(int j=i+1;j<a.length;j++)

        {

            //在无序区中找到最小数据并保存其数组下标

            if(a[j]<a[minIndex])

            {

                minIndex=j;

            }

        }

        //将最小元素放到本次循环的前端

        temp=a[i];

        a[i]=a[minIndex];

        a[minIndex]=temp;

    }

}
```

```
#include<stdio.h>
```

```
void select_sort(int*a,int n)

{

    register int i,j,min,t;

    for(i=0;i<n-1;i++)

    {

        min=i;//查找最小值

        for(j=i+1;j<n;j++)

            if(a[min]>a[j])

                min=j;//交换

        if(min!=i)

        {

            t=a[min];

            a[min]=a[i];

            a[i]=t;

        }

    }

}

int main(){

    int a[] ={1,5,3,6,9,7,8,4,0,2};

    select_sort(&a,10);

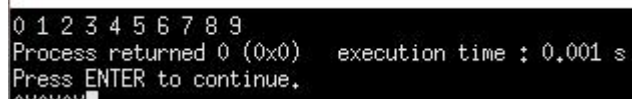
    for(int i=0;i<10;i++){

        printf("%d ",a[i]);

    }

    return 0;

}
```

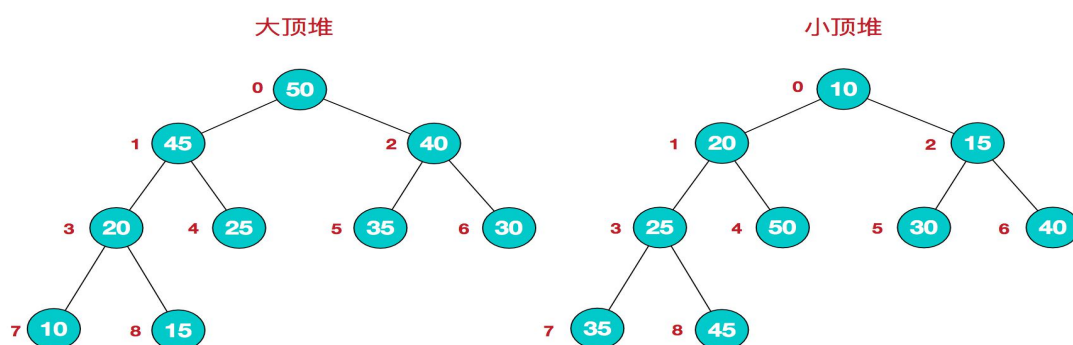


```
0 1 2 3 4 5 6 7 8 9
Process returned 0 (0x0)   execution time : 0.001 s
Press ENTER to continue.
```

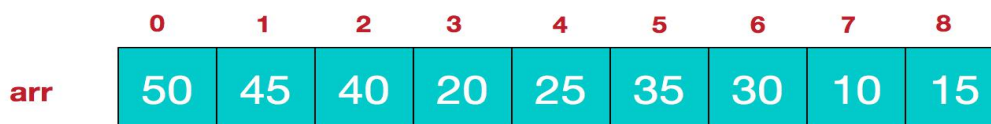
## 图解排序算法(三)之堆排序

堆排序是利用**堆**这种数据结构而设计的一种排序算法,堆排序是一种**选择排序**,它的最坏,最好,平均时间复杂度均为  $O(n\log n)$ ,它也是不稳定排序。首先简单了解下堆结构。

**堆**是具有以下性质的完全二叉树:每个结点的值都大于或等于其左右孩子结点的值,称为**大顶堆**;或者每个结点的值都小于或等于其左右孩子结点的值,称为**小顶堆**。如下图:



同时,我们对堆中的结点按层进行编号,将这种逻辑结构映射到数组中就是下面这个样子



该数组从逻辑上讲就是一个堆结构,我们用简单的公式来描述一下堆的定义就是:



**大顶堆** :  $\text{arr}[i] \geq \text{arr}[2i+1] \ \&\& \ \text{arr}[i] \geq \text{arr}[2i+2]$

**小顶堆** :  $\text{arr}[i] \leq \text{arr}[2i+1] \ \&\& \ \text{arr}[i] \leq \text{arr}[2i+2]$

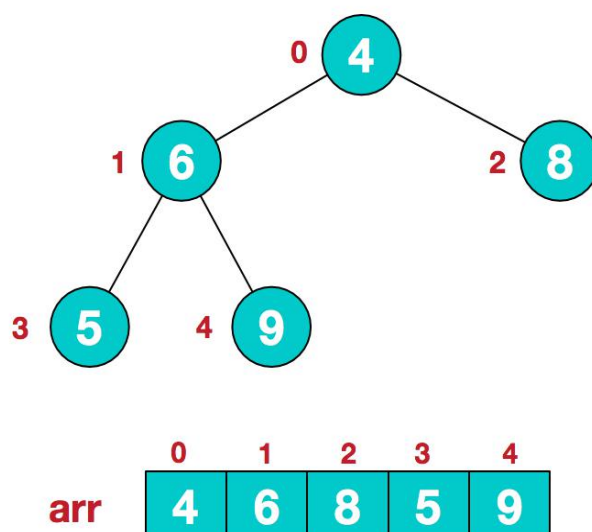
ok，了解了这些定义。接下来，我们来看看堆排序的基本思想及基本步骤：

### 堆排序基本思想及步骤

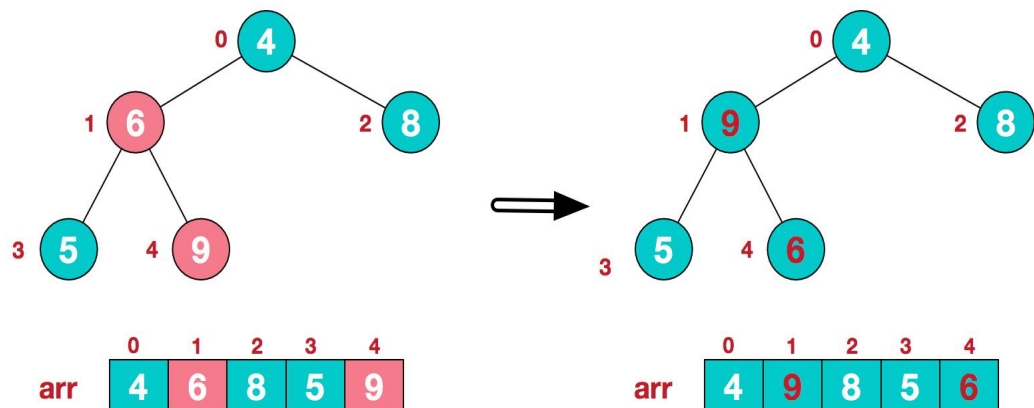
堆排序的基本思想是：将待排序序列构造成一个大顶堆，此时，整个序列的最大值就是堆顶的根节点。将其与末尾元素进行交换，此时末尾就为最大值。然后将剩余  $n-1$  个元素重新构造成一个堆，这样会得到  $n$  个元素的次小值。如此反复执行，便能得到一个有序序列了

**步骤一 构造初始堆。**将给定无序序列构造成一个大顶堆（一般升序采用大顶堆，降序采用小顶堆）。

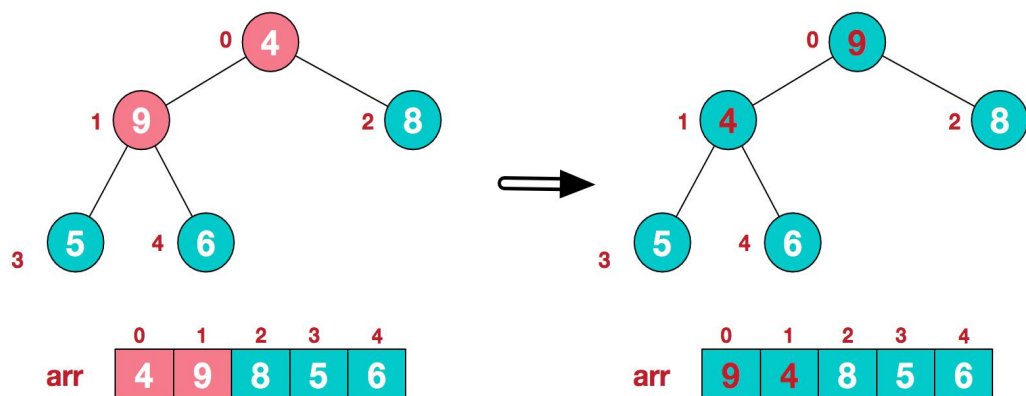
a.假设给定无序序列结构如下



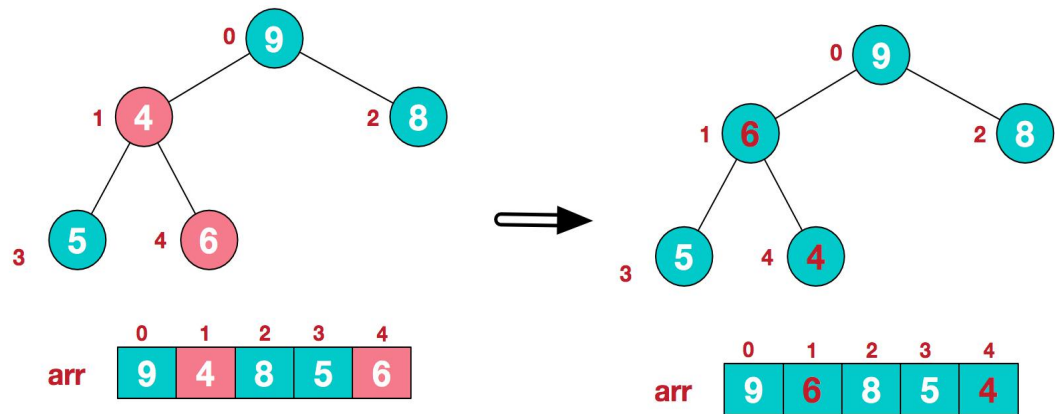
2.此时我们从最后一个非叶子结点开始（叶结点自然不用调整，第一个非叶子结点  $\text{arr.length}/2-1=5/2-1=1$ ，也就是下面的 6 结点），从左至右，从下至上进行调整。



4.找到第二个非叶节点 4，由于[4,9,8]中 9 元素最大，4 和 9 交换。



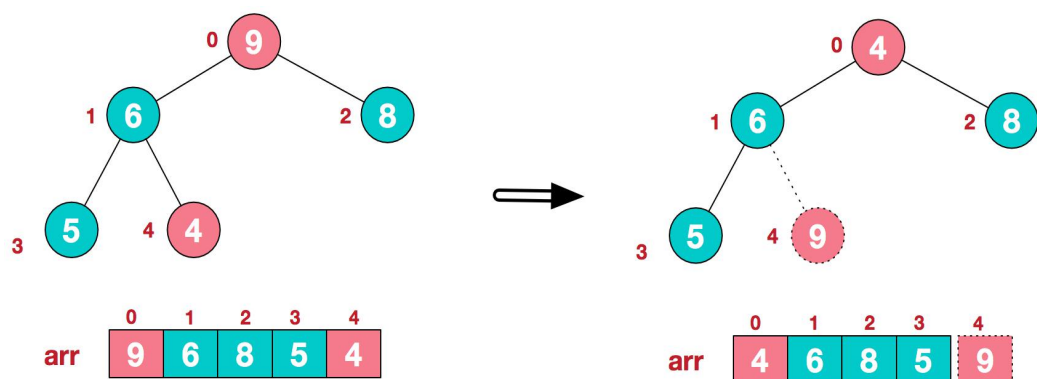
这时，交换导致了子根[4,5,6]结构混乱，继续调整，[4,5,6]中 6 最大，交换 4 和 6。



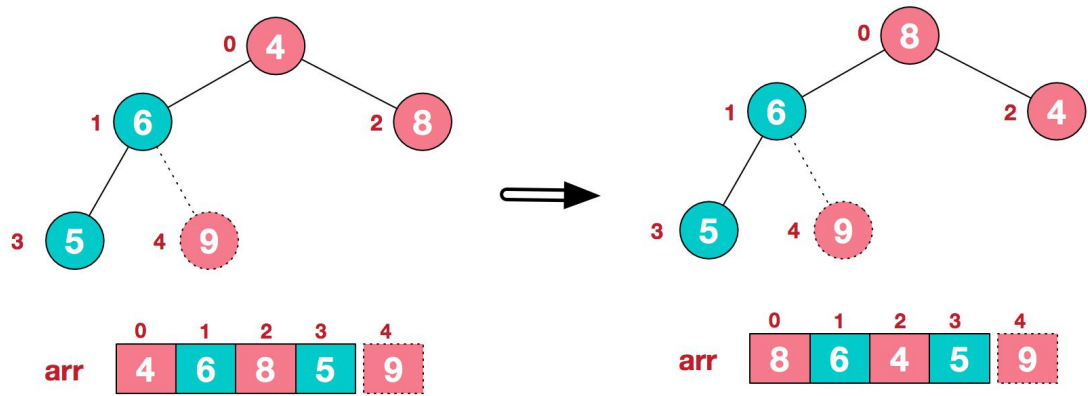
此时，我们就将一个无序序列构造成了一个大顶堆。

**步骤二 将堆顶元素与末尾元素进行交换，使末尾元素最大。然后继续调整堆，再将堆顶元素与末尾元素交换，得到第二大元素。如此反复进行交换、重建、交换。**

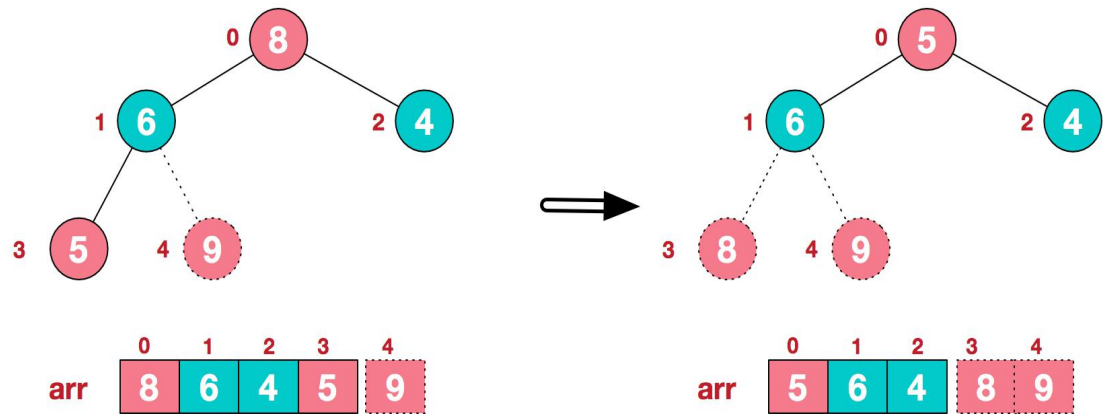
a. 将堆顶元素 9 和末尾元素 4 进行交换



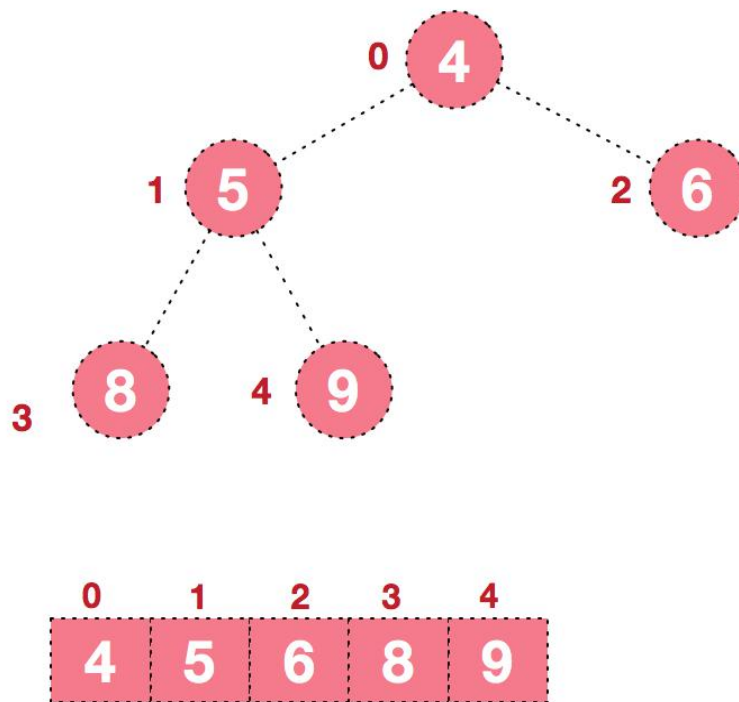
b. 重新调整结构，使其继续满足堆定义



c. 再将堆顶元素 8 与末尾元素 5 进行交换，得到第二大元素 8。



后续过程，继续进行调整，交换，如此反复进行，最终使得整个序列有序



再简单总结下堆排序的基本思路：

- 将无序序列构建成一个堆，根据升序降序需求选择大顶堆或小顶堆;
- 将堆顶元素与末尾元素交换，将最大元素"沉"到数组末端;
- 重新调整结构，使其满足堆定义，然后继续交换堆顶元素与当前末尾元素，反复执行调整+交换步骤，直到整个序列有序。

```
package sortdemo;  
  
import java.util.Arrays;  
  
/**
```

\* Created by chengxiao on 2016/12/17.

\* 堆排序 demo

```
*/public class HeapSort {

    public static void main(String []args){

        int []arr = {9,8,7,6,5,4,3,2,1};

        sort(arr);

        System.out.println(Arrays.toString(arr));

    }

    public static void sort(int []arr){

        //1.构建大顶堆

        for(int i=arr.length/2-1;i>=0;i--){

            //从第一个非叶子结点从下至上，从右至左调整结构
            adjustHeap(arr,i,arr.length);

        }

        //2.调整堆结构+交换堆顶元素与末尾元素

        for(int j=arr.length-1;j>0;j--){

            swap(arr,0,j);//将堆顶元素与末尾元素进行交换

            adjustHeap(arr,0,j);//重新对堆进行调整        }

        }

    /**

     * 调整大顶堆（仅是调整过程，建立在大顶堆已构建的基础上）
```

```
* @param arr

* @param i

* @param length

*/

public static void adjustHeap(int []arr,int i,int length){

    int temp = arr[i];//先取出当前元素 i

    for(int k=i*2+1;k<length;k=k*2+1){//从 i 结点的左子结点开始,
    也就是 2i+1 处开始

        if(k+1<length && arr[k]<arr[k+1]){//如果左子结点小于右子
        结点, k 指向右子结点

            k++;

        }

        if(arr[k] >temp){//如果子节点大于父节点, 将子节点值赋给父节
        点 (不用进行交换)

            arr[i] = arr[k];

            i = k;

        }else{

            break;

        }

    }

    arr[i] = temp;//将 temp 值放到最终的位置 }
```

```
/**  
  
 * 交换元素  
  
 * @param arr  
  
 * @param a  
  
 * @param b  
  
 */  
  
public static void swap(int []arr,int a ,int b){  
  
    int temp=arr[a];  
  
    arr[a] = arr[b];  
  
    arr[b] = temp;  
  
}  
  
}
```

结果    `[1, 2, 3, 4, 5, 6, 7, 8, 9]`

堆排序是一种选择排序，整体主要由构建初始堆+交换堆顶元素和末尾元素并重建堆两部分组成。其中构建初始堆经推导复杂度为  $O(n)$ ，在交换并重建堆的过程中，需交换  $n-1$  次，而重建堆的过程中，根据完全二叉树的性质， $[\log_2(n-1), \log_2(n-2) \dots 1]$  逐步递减，近似为  $n \log n$ 。所以堆排序时间复杂度一般认为就是  $O(n \log n)$  级。

实验报告 C 语言实现



## 堆排序

源程序运行截图：



```
//可以考虑返回数组的
main()
//
int a[N],i;//把a[0]空
printf("请输入%d个自然数:",N);
for(i=1;i<=N-1;i++)
scanf("%d",&a[i]);
printf("\n堆排序结果\n");
Sort(a,N-1);
for(i=1;i<=N-1;i++)
printf(" %d ",a[i]);
printf("\n");
return 0;
```

```
请输入10个自然数:
0
9
8
7
6
5
4
3
2
1
堆排序结果为:
0 1 2 3 4 5 6 7 8 9
Press any key to continue
```

源代码：

采用指针传递！

```
#include<stdio.h>
```

```
#define N 11
```

```
//堆排序:单一元素的寻找合适位置
```

```
void Sort1(int *a,int low ,int hight)
```

```
{
```

```
    int i,j,temp;
```

```
    i=low;
```

```
    j=2*i;
```

```
    temp=a[low];
```

```
while(j<=hight)

{

    if(j<hight&& a[j]<a[j+1]) ++j;

    if(temp<a[j])

    {

        a[i]=a[j];

        i=j;

        j=2*i;

    }

    else break;

}

a[i]=temp;

}

//堆整体排序

void Sort(int *a, int n)

{

    int i,j,temp;

    //初始建堆

    for (i=n/2;i>0;--i)//从最后一个结点开始

        Sort1(a,i,n);

    //堆排序开始

    for(j=n;j>1;--j)
```

```
{

    temp=a[j];

    a[j]=a[1];//每次排序的第一个元素是最大的

    a[1]=temp;

    Sort1(a,1,j-1);

}

//可以考虑返回数组的首地址

}

int main()

{

    int a[N],i;//把 a[0]空置 , 只要 1 到 N-1 的数 ( 共 n-1 个数 )

    printf("请输入%d 个自然数 : \n",N-1);

    for(i=1;i<=N-1;i++)

        scanf("%d",&a[i]);

    printf("\n 堆排序结果为 : \n");

    Sort(a,N-1);

    for(i=1;i<=N-1;i++)

        printf(" %d ",a[i]);

    printf("\n");

    return 0;

}
```

## 实际证明可以不用指针传递！

```
#include<stdio.h>

#define N 11

//堆排序:单一元素的寻找合适位置

void Sort1(int a[],int low ,int hight)
{
    int i,j,temp;

    i=low;

    j=2*i;

    temp=a[low];

    while(j<=hight)
    {
        if(j<hight&& a[j]<a[j+1]) ++j;

        if(temp<a[j])
        {
            a[i]=a[j];

            i=j;

            j=2*i;
        }

        else break;
    }

    a[i]=temp;
```

```
}

//堆整体排序

void Sort(int a[], int n)

{

    int i,j,temp;

    //初始建堆

    for (i=n/2;i>0;--i)

        Sort1(a,i,n);

    //堆排序开始

    for(j=n;j>1;--j)

    {

        temp=a[j];

        a[j]=a[1];//每次排序的第一个元素是最大的

        a[1]=temp;

        Sort1(a,1,j-1);

    }

    //可以考虑返回数组的首地址

}

int main()

{

    int a[N],i;//把 a[0]空置 , 只要 1 到 N-1 的数 ( 共 n-1 个数 )
```

```
printf("请输入%d 个自然数 : \n",N-1);

for(i=1;i<=N-1;i++)

    scanf("%d",&a[i]);

printf("\n 堆排序结果为 : \n");

Sort(a,N-1);

for(i=1;i<=N-1;i++)

    printf(" %d ",a[i]);

printf("\n");

return 0;

}
```

## 归并排序

前面讲了冒泡、选择、插入三种简单排序，时间复杂度都是  $O(n^2)$ ，

我们总是可以将一个数组一分为二，然后二分为四，直到每一组只有两个元素，这可以理解为个递归的过程，然后将两个元素进行排序，之后再将两个元素为一组进行排序。直到所有的元素都排序完成。同样我们来看下边这个动图。

归并排序算法是采用分治法的一个非常典型的应用，且各层分治递归可以同时进行。

### 归并算法的思想

归并算法其实可以分为递归法和迭代法（自底向上归并），两种实现对于最小集合的归并操作思想是一样的。区别在于如何划分数组，我们先介绍下算法最基本的操作：

1. 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列；
2. 设定两个指针，最初位置分别为两个已经排序序列的起始位置；
3. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置；
4. 重复步骤 3 直到某一指针到达序列尾；
5. 将另一序列剩下的所有元素直接复制到合并序列尾。

我们来看看 Java 递归代码是怎么实现的：

```
public class Test09 {  
  
    private static void printArr(int[] arr) {
```

```
        for (int anArr : arr) {

            System.out.print(anArr + " ");

        }
    }

    private static void mergeSort(int[] arr) {

        if (arr == null)

            return;

        mergeSort(arr, 0, arr.length - 1);

    }

    private static void mergeSort(int[] arr, int start, int end)
    {

        if (start >= end)

            return;

        // 找出中间索引

        int mid = start + (end - start >> 1);

        // 对左边数组进行递归
        mergeSort(arr, start, mid);
        // 对右边数组进行递归
        mergeSort(arr, mid + 1, end);

        // 合并
        merge(arr, start, mid, end);

    }

    private static void merge(int[] arr, int start, int mid, int
end) {

        // 先建立一个临时数组，用于存放排序后的数据
```



```
int[] tmpArr = new int[arr.length];

int start1 = start, end1 = mid, start2 = mid + 1, end2 = end;

// 创建一个下标

int pos = start1;

// 缓存左边数组的第一个元素的索引

int tmp = start1;

while (start1 <= end1 && start2 <= end2) {

    // 从两个数组中取出最小的放入临时数组

    if (arr[start1] <= arr[start2])
        tmpArr[pos++] = arr[start1++];
    else
        tmpArr[pos++] = arr[start2++];
}

// 剩余部分依次放入临时数组, 实际上下面两个 while 只会
// 执行其中一个

while (start1 <= end1) {
    tmpArr[pos++] = arr[start1++];
}

while (start2 <= end2) {
    tmpArr[pos++] = arr[start2++];
}

// 将临时数组中的内容拷贝回原来的数组中

while (tmp <= end) {
    arr[tmp] = tmpArr[tmp++];
}

}
```

```
public static void main(String[] args) {  
  
    int[] arr = {6, 4, 2, 1, 8, 3, 7, 9, 5};  
    mergeSort(arr);  
    printArr(arr);  
}  
}
```

归并排序算法总的时间复杂度是  $O(n\log n)$ ，而且这是归并排序算法中最好、最坏、平均的时间性能。

而由于在归并排序过程中需要与原始记录序列同样数量的存储空间存放归并结果以及递归时压入栈的数据占用的空间： $n + \log n$ ，所以空间复杂度为  $O(n)$ 。

## 总结

归并排序虽然比较稳定，在时间上也是非常有效的，但是这种算法很消耗空间，一般来说只有在外部排序才会采用这个方法，但在内部排序不会用这种方法，而是用快速排序。

## 快速排序

快速排序的平均时间复杂度为  $O(n \times \log(n))$ ，最糟糕时复杂度为  $O(n^2)$

```
public class QuickSort {  
  
    public int division(int[] list, int left, int right) {  
  
        // 以最左边的数(left)为基准  
  
        int base = list[left];  
  
        while (left < right) {  
  
            // 从序列右端开始，向左遍历，直到找到小于 base 的数  
  
            while (left < right && list[right] >= base)  
  
                right--;  
  
            // 找到了比 base 小的元素，将这个元素放到最左边的位置  
  
            list[left] = list[right];  
  
  
            // 从序列左端开始，向右遍历，直到找到大于 base 的数  
  
            while (left < right && list[left] <= base)  
  
                left++;  
  
            // 找到了比 base 大的元素，将这个元素放到最右边的位置  
  
            list[right] = list[left];  
  
        }  
    }  
}
```

```
// 最后将 base 放到 left 位置。此时，left 位置的左侧数值应该都比 left 小；

// 而 left 位置的右侧数值应该都比 left 大。

list[left] = base;

return left;

}

private void quickSort(int[] list, int left, int right) {

    // 左下标一定小于右下标，否则就越界了

    if (left < right) {

        // 对数组进行分割，取出下次分割的基准标号

        int base = division(list, left, right);

        System.out.format("base = %d:\t", list[base]);

        printPart(list, left, right);

        // 对“基准标号”左侧的一组数值进行递归的切割，以至于将这些数值完整的排

        quickSort(list, left, base - 1);

        // 对“基准标号”右侧的一组数值进行递归的切割，以至于将这些数值完整的排
```

序

序

```
        quickSort(list, base + 1, right);

    }

}

// 打印序列

public void printPart(int[] list, int begin, int end) {

    for (int i = 0; i < begin; i++) {

        System.out.print("\t");

    }

    for (int i = begin; i <= end; i++) {

        System.out.print(list[i] + "\t");

    }

    System.out.println();

}

public static void main(String[] args) {

    // 初始化一个序列

    // int[] array = new int[]{1, 3, 4, 5, 2, 6, 9, 7, 8, 0};

    int[] array = {1, 3, 4, 5, 2, 6, 9, 7, 8, 0};

    // 调用快速排序方法

    QuickSort quick = new QuickSort();
```

```

System.out.print("排序前:\t\t");

quick.printPart(array, 0, array.length - 1);

quick.quickSort(array, 0, array.length - 1);

System.out.print("排序后:\t\t");

quick.printPart(array, 0, array.length - 1);

    }

}

```

排序前:	1	3	4	5	2	6	9	7	8	0
base = 1:	0	1	4	5	2	6	9	7	8	3
base = 4:			3	2	4	6	9	7	8	5
base = 3:			2	3						
base = 6:						5	6	7	8	9
base = 7:								7	8	9
base = 8:									8	9
排序后:	0	1	2	3	4	5	6	7	8	9