

# 排序算法总结

| 排序算法 | 平均时间复杂度             | 最坏时间复杂度             | 空间复杂度       | 稳定性 |
|------|---------------------|---------------------|-------------|-----|
| 冒泡排序 | $O(n^2)$            | $O(n^2)$            | $O(1)$      | 稳定  |
| 选择排序 | $O(n^2)$            | $O(n^2)$            | $O(1)$      | 不稳定 |
| 插入排序 | $O(n^2)$            | $O(n^2)$            | $O(1)$      | 稳定  |
| 希尔排序 | $O(n^{1.5})$        | $O(n^{1.5})$        | $O(1)$      | 不稳定 |
| 快速排序 | $O(N \cdot \log N)$ | $O(n^2)$            | $O(\log N)$ | 不稳定 |
| 归并排序 | $O(N \cdot \log N)$ | $O(N \cdot \log N)$ | $O(N)$      | 稳定  |
| 堆排序  | $O(N \cdot \log N)$ | $O(N \cdot \log N)$ | $O(1)$      | 不稳定 |
| 基数排序 | $O(d(n+r))$         | $O(d(n+r))$         | $O(N)$      | 稳定  |

## 一. 冒泡排序(BubbleSort)

1. 步骤：

- 1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
- 2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步 做完后，最后的元素会是最大的数。
- 3. 针对所有的元素重复以上的步骤，除了最后一个。
- 4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要 比较为止。

2. 平均时间复杂度： $O(n^2)$

3. java代码实现：

```
public static void BubbleSort(int [] arr){
    int temp; //临时变量
    for(int i=0; i<arr.length-1; i++){    //表示趟数，一共arr.length-1次。
        for(int j=arr.length-1; j>i; j--){
            if(arr[j] < arr[j-1]){
                temp = arr[j];
                arr[j] = arr[j-1];
                arr[j-1] = temp;
            }
        }
    }
}
```

4. 优化：

- 针对问题：  
数据的顺序排好之后，冒泡算法仍然会继续进行下一轮的比较，直到arr.length-1次，后面的比较没有意义的。
- 方案：  
设置标志位flag，如果发生了交换flag设置为true；如果没有交换就设置为false。  
这样当一轮比较结束后如果flag仍为false，即：这一轮没有发生交换，说明数据的顺序已经排好，没有必要继续进行下去。  
``java public static void BubbleSort1(int [] arr){ int temp;//临时变量 boolean flag;//是否交换的标志 for(int i=0;i<arr.length-1;i++){ //表示趟数，一共arr.length-1次。

```
        flag = false;
        for(int j=arr.length-1; j>i; j--){

            if(arr[j] < arr[j-1]){
                temp = arr[j];
                arr[j] = arr[j-1];
                arr[j-1] = temp;
                flag = true;
            }
        }
        if(!flag) break;
    }
}
```

## 二. 选择排序(SelectionSort)

### 1. 步骤：

1. 首先从原始数组中选择最小的一个数据，将其和位于第一个位置的数据进行交换。
2. 接着从剩下的n-1个数据中选择次小的一个元素，将其和第二个位置的数据交换。
3. 然后，这样不断重复，直到最后连个数据完成交换。至此，便完成对原始数组的从小到大的排序。

### 2. 平均时间复杂度：O(n<sup>2</sup>)

### 3. java代码实现：

```
public static void select_sort(int array[],int lenth){

    for(int i=0;i<lenth-1;i++){

        int minIndex = i;
        for(int j=i+1;j<lenth;j++){
            if(array[j]<array[minIndex]){
                minIndex = j;
            }
        }
    }
}
```

```
        if(minIndex != i){
            int temp = array[i];
            array[i] = array[minIndex];
            array[minIndex] = temp;
        }
    }
}
```

### 三. 插入排序(Insertion Sort)

#### 1. 步骤：

1. 从第一个元素开始，该元素可以认为已经被排序
2. 取出下一个元素，在已经排序的元素序列中从后向前扫描
3. 如果该元素（已排序）大于新元素，将该元素移到下一位置
4. 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置
5. 将新元素插入到该位置后
6. 重复步骤2~5

#### 2. 平均时间复杂度： $O(n^2)$

#### 3. java代码实现：

```
public static void insert_sort(int array[],int length){

    int temp;

    for(int i=0;i<length-1;i++){
        for(int j=i+1;j>0;j--){
            if(array[j] < array[j-1]){
                temp = array[j-1];
                array[j-1] = array[j];
                array[j] = temp;
            }else{ //不需要交换
                break;
            }
        }
    }
}
```

### 四. 希尔排序(Shell Sort)

#### 1. 前言：

- 数据序列1：13-17-20-42-28 利用插入排序，13-17-20-28-42. Number of swap:1;
- 数据序列2：13-17-20-42-14 利用插入排序，13-14-17-20-42. Number of swap:3;
- 如果数据序列基本有序，使用插入排序会更加高效。

#### 2. 步骤:

1. 将有 $n$ 个元素的数组分为 $n/2$ 个数字序列，第1个数据和第 $n/2+1$ 个数据为一对。。。
2. 一次循环使每一个序列对排好顺序（对每个序列使用插入排序算法，实质是一种分组插入）
3. 然后，再变为 $n/4$ 个序列，再次排序
4. 不断重复上述过程，随着序列减少最后变为一个，也就完成了整个排序。

3. 平均时间复杂度：

4. java代码实现：

```
public static void shell_sort(int array[],int length){
    int temp = 0;
    int incre = length;
    while(true){
        incre = incre/2;
        for(int k = 0;k<incre;k++){    //根据增量分为若干子序列
            for(int i=k+incre;i<length;i+=incre){
                for(int j=i;j>k;j-=incre){
                    if(array[j]<array[j-incre]){
                        temp = array[j-incre];
                        array[j-incre] = array[j];
                        array[j] = temp;
                    }else{
                        break;
                    }
                }
            }
        }
        if(incre == 1){
            break;
        }
    }
}
```

## 五. 快速排序(Quicksort)

1. 基本思想：（分治）

1. 从数列中挑出一个元素，称为“基准”（pivot）；
2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
3. 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序；

2. 平均时间复杂度： $O(N*\log N)$

3. 代码实现：

```
public static void quickSort(int a[],int l,int r){
    if(l>=r){
        return;
    }
    int i = l; int j = r; int key = a[l]; //选择第一个数为key
    while(i<j){
        while(i<j && a[j]>=key){ //从右向左找第一个小于key的值
            j--;
        }
        if(i<j){
            a[i] = a[j];
            i++;
        }
        while(i<j && a[i]<key){ //从左向右找第一个大于key的值
            i++;
        }
        if(i<j){
            a[j] = a[i];
            j--;
        }
    }
    //i == j
    a[i] = key;
    quickSort(a, l, i-1); //递归调用
    quickSort(a, i+1, r); //递归调用
}
```

key值的选取可以有多种形式，例如中间数或者随机数，分别会对算法的复杂度产生不同的影响。

## 六. 归并排序(Merge Sort)

### 1. 步骤

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 直到某一指针达到序列尾；
5. 将另一序列剩下的所有元素直接复制到合并序列尾

### 2. 平均时间复杂度： $O(N\log N)$

- 长度为 $N$ ，将数列分开成小数列一共要 $\log N$ 步，每步都是一个合并有序数列的过程，时间复杂度可以记为 $O(N)$ ，故一共为 $O(N*\log N)$ 。

### 3. 代码实现：

```
public static void merge_sort(int a[],int first,int last,int temp[]){
    if(first < last){
        int middle = (first + last)/2;
        merge_sort(a,first,middle,temp); //左半部分排好序
        merge_sort(a,middle+1,last,temp); //右半部分排好序
        mergeArray(a,first,middle,last,temp); //合并左右部分
    }
}
```

```
//合并 : 将两个序列a[first-middle],a[middle+1-end]合并
public static void mergeArray(int a[],int first,int middle,int end,int
temp[]){
    int i = first;
    int m = middle;
    int j = middle+1;
    int n = end;
    int k = 0;
    while(i<=m && j<=n){
        if(a[i] <= a[j]){
            temp[k] = a[i];
            k++;
            i++;
        }else{
            temp[k] = a[j];
            k++;
            j++;
        }
    }
    while(i<=m){
        temp[k] = a[i];
        k++;
        i++;
    }
    while(j<=n){
        temp[k] = a[j];
        k++;
        j++;
    }
    for(int ii=0;ii<k;ii++){
        a[first + ii] = temp[ii];
    }
}
```

## 七. 堆排序(HeapSort)

### 1. 步骤

1. 创建一个堆 H[0.....n-1]；
2. 把堆首（最大值）和堆尾互换；

3. 把堆的尺寸缩小 1，并调用 `shift_down(0)`，目的是把新的数组顶端数据调整到相应位置；
4. 重复步骤 2，直到堆的尺寸为 1。

## 2. 平均时间复杂度： $O(N\log N)$

- 由于每次重新恢复堆的时间复杂度为 $O(\log N)$ ，共 $N - 1$ 次重新恢复堆操作，再加上前面建立堆时 $N / 2$ 次向下调整，每次调整时间复杂度也为 $O(\log N)$ 。二次操作时间相加还是 $O(N * \log N)$ 。

## 3. java代码实现：

```
//构建最小堆
public static void MakeMinHeap(int a[], int n){
    for(int i=(n-1)/2 ; i>=0 ; i--){
        MinHeapFixdown(a,i,n);
    }
}
//从i节点开始调整,n为节点总数 从0开始计算 i节点的子节点为 2*i+1, 2*i+2
public static void MinHeapFixdown(int a[],int i,int n){
    int j = 2*i+1; //子节点
    int temp = 0;
    while(j<n){
        //在左右子节点中寻找最小的
        if(j+1<n && a[j+1]<a[j]){
            j++;
        }
        if(a[i] <= a[j]){
            break;
        }
        //较大节点下移
        temp = a[i];
        a[i] = a[j];
        a[j] = temp;
        i = j;
        j = 2*i+1;
    }
}
```

```
public static void MinHeap_Sort(int a[],int n){
    int temp = 0;
    MakeMinHeap(a,n);
    for(int i=n-1;i>0;i--){
        temp = a[0];
        a[0] = a[i];
        a[i] = temp;
        MinHeapFixdown(a,0,i);
    }
}
```

## 八. 基数排序(RadixSort)

## BinSort

### 1. 基本思想：

- BinSort想法非常简单，首先创建数组A[MaxValue]；然后将每个数放到相应的位置上（例如17放在下标17的数组位置）；最后遍历数组，即为排序后的结果。

### 2. 图示：![BinSort](file:///D:/Bao/Pictures/blog/binsort.png "BinSort")

### 3. 问题：

- 当序列中存在较大值时，BinSort 的排序方法会浪费大量的空间开销。

## RadixSort

### 1. 步骤：根据键值的每位数字来分配桶；（桶排序）

### 2. java代码实现：

```
from typing import List

def radix_sort(arr: List[int]):
    n = len(str(max(arr))) # 记录最大值的位数
    for k in range(n): # n轮排序
        # 每一轮生成10个列表
        bucket_list = [[] for i in range(10)] # 因为每一位数字都是0~9，故建立10
        # 个桶
        for i in arr:
            # 按第k位放入到桶中
            bucket_list[i // (10**k) % 10].append(i)
        # 按当前桶的顺序重排列表
        arr = [j for i in bucket_list for j in i]
    return arr
```