第八章 排序

- 算法的稳定性：
  - 若待排序表中有两个元素R_i和R_j对应的关键字相同，并且R_i在R_j前面，若使用某一排序算法之后，R_i依然在R_j前面，则称该排序算法是稳定的，否则就是不稳定的。
- 内部排序和外部排序
  - 1、内部排序：指在排序期间数据对象全部存放在内存的排序。插入排序、选择排序、归并排序等等，这些算法都属于内部排序算法，即排序的整个过程只是在内存中完成。
  - 2、外部排序：当待排序的文件比内存的可使用容量还大时，文件无法一次性放到内存中进行排序，需要借助于外部存储器（例如硬盘、U盘、光盘）就需要使用外部排序。

一、插入排序

1.直接插入排序

- 1.定义：直接插入算法的思想非常简单，把一个无序的数组从头到尾进行遍历，为每一个元素找到属于它的位置。
- 2.算法
  -
```
void InsertSort(int a[],int n){
    for (int i = 1; i < n; ++i) {
        if (a[i] < a[i-1]){ // 若后面这个元素< 前面这个元素 ，就要去前面找到属于他的位置
            int j =i-1;
            int x = a[i];
            // 退出条件 ：1.找到了尽头 也就是 x 比 a[0]还要小，比完之后 j = 0 -1 = -1;
            //              so exit 条件之一： j>-1
            //          2.前面这个数已经比 x 小了
            while (j>-1 && x < a[j]){
                a[j+1] = a[j];   // 前面的到后面去
                j--;
            }
            a[j+1] = x; // 找到位置了
        }
        print(a,n,i);
    }
}
```
- 3.时间复杂度：$O(N^2)$

2.折半插入排序

- 1.定义：折半插入和直接插入排序很相似，直接插入排序中寻找插入位置使用的顺序查找，而我们在查找中学过折半查找，折半插入排序就是利用折半查找 相当于是 直接插入排序的一个优化。
- 2.算法
  -

```
void binary_insertSort(int a[],int n){
    int low,high,mid,x = 0;
    for (int i = 1; i <n ; ++i) {
        low = 0;
        high = i-1;
        x = a[i];
        while (low <= high){
            mid = (low + high)/ 2;
            if (x < a[mid]){
                high = mid-1;
            }else{
                low = mid +1;
            }
        }
        for (int j = i; j >low ; --j) {
            a[j] = a[j-1];
        }
        a[low] = x;
        print(a,n,i);
    }
```

- 3.时间复杂度：$O(N^2)$

3.希尔排序

- 1.定义：希尔排序(Shell Sort)是插入排序的一种，它是针对直接插入排序算法的改进。该方法又称缩小增量排序。它的思路如下：取一个小于n的正整数作为gap(对于gap的取值没有最优，一般来说会取$length/2$)，将gap个元素放入一个组中(取值为0,gap,2gap... 可以想做抽样？not sure)，这样就可以把长度为n的数列分为若干个组，对这些分组各自进行直接插入排序。然后再取gap=gap/2,继续分组内排序，直到gap = 1时，这个数组就是有序的了。
- 2.算法：
  -

```
void shell_sort(int a[],int n){
    for (int gap = n/2; gap > 0 ; gap/=2) {
        cout<<"gap为: "<<gap<<endl;
        // gap个组 进行分组
        for (int i = 0; i < gap; ++i) {
            for (int j = i+gap; j <n ;  j += gap) {
                // 直接插入排序
                if (a[j] <a[j -gap]){
                    int x = a[j];
                    int k = j;
                    while (k>=0 && x < a[k]){
                        a[k+gap] = a[k];    // 前一个元素到后面去
                        k = k - gap;    // k 前移
                    }
                    a[k+gap] = x;
                }
            }
        }
    }
```

- 3.时间复杂度：
  - 时间复杂度会依赖于gap的选取。当n在某个特定范围时，时间复杂度约为$O(n^1.3)$，最坏情况下时间复杂度为$O(n^2)$.
- 4.稳定性
  - 希尔排序是不稳定的算法，对于相同的两个数，可能由于分在不同的组中而导致它们的顺序发生变化。

二、交换排序

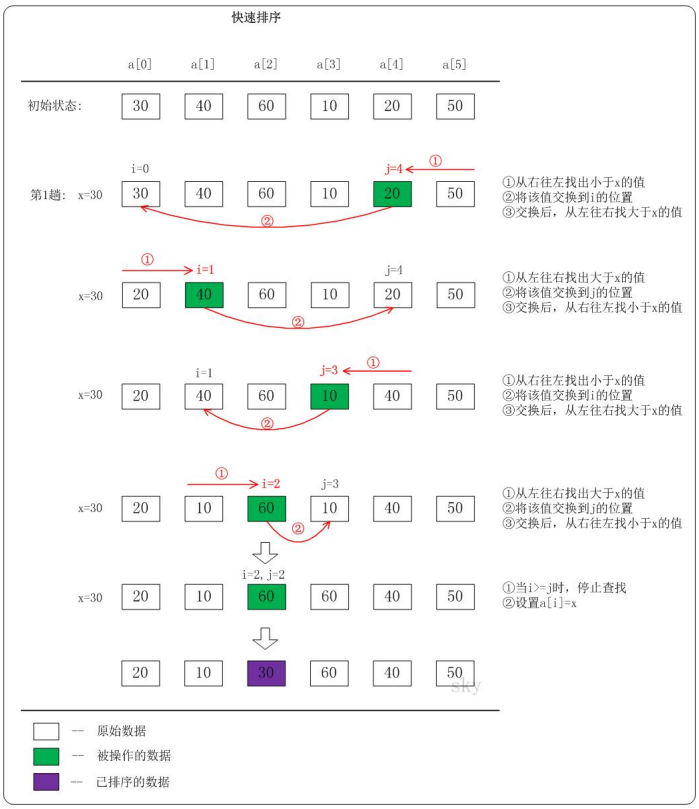
- 交换指的是根据两个元素的比较结果来交换这两个元素的位置从而达到排序的效果。
- 1、冒泡排序
  - 1.代码

```
int* bubble_sort(int a[],int n){
    for (int i = 0; i < n; ++i) {
        for (int j = i+1; j < n; ++j) {
            if (a[i] < a[j])
                swap(&a[i],&a[j]);
        }
    }

    return a;
}
```

  - 2.时间复杂度： $O(n^2)$
  - 3.空间复杂度：$O(1)$

- 2、快速排序
  - 1.定义：快速排序是基于分治法的。它的基本思想是：
    - （1）首先选择一个基准数（一般取首元素）
    - （2）将比基准小的数放到基准前面，比基准大的数放到基准后面，在这个过程结束后，基准就放在了排序结束后的位置。
    - （3）递归的 把基准前的数列 和 基准后的数列 进行排序。
  - 2.排序过程
    -



  - 3.算法
    -

```
void quick_sort(int *a,int l,int r){
    if (l < r){
        int i = l;
        int j = r;
        int x = a[l]; // 基准
        while (i<j){
            while (i<j && a[j]>=x)  // 先从左往右寻找第一个小于X的数
                j--;
            a[i] = a[j];
            while (i<j && a[i]<=x)
                i++;
            a[j] = a[i];
        }
        a[i] = x;
        quick_sort(a,l, r: i-1);
        quick_sort(a, l: i+1,r);
    }
}
```

- 4.时间复杂度
  - 快速排序的时间复杂度在最坏情况下是$O(N^2)$，平均的时间复杂度是$O(N*lgN)$。
  - 这句话很好理解：假设被排序的数列中有N个数。遍历一次的时间复杂度是O(N)，需要遍历多少次呢？至少lg(N+1)次，最多N次。
  - (1) 为什么最少是lg(N+1)次？快速排序是采用的分治法进行遍历的，我们将它看作一棵二叉树，它需要遍历的次数就是二叉树的深度，而根据完全二叉树的定义，它的深度至少是lg(N+1)。因此，快速排序的遍历次数最少是lg(N+1)次。
  - (2) 为什么最多是N次？这个应该非常简单，还是将快速排序看作一棵二叉树，它的深度最大是N。因此，快排的遍历次数最多是N次。
- 5.稳定性
  - 快排是不稳定的。快排是所有内部排序算法中平均性能最优的排序算法。
-

三、选择排序

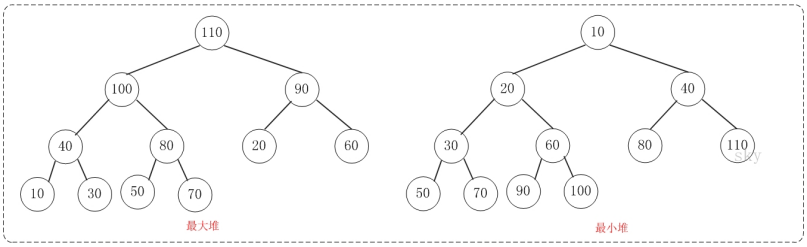
- 选择排序的基本思想就是：第i个元素是其后面元素中最小的元素，经过n-1次排序之后就得到了一个升序的序列。就好比排座位，先从人群中选出最矮的放在第一排，第2矮的放在第二排，以此类推，当排完第n-1个人的时候，最后一个人就不需要进行排序了，他只有最后一个位置可以选。

1.简单选择排序

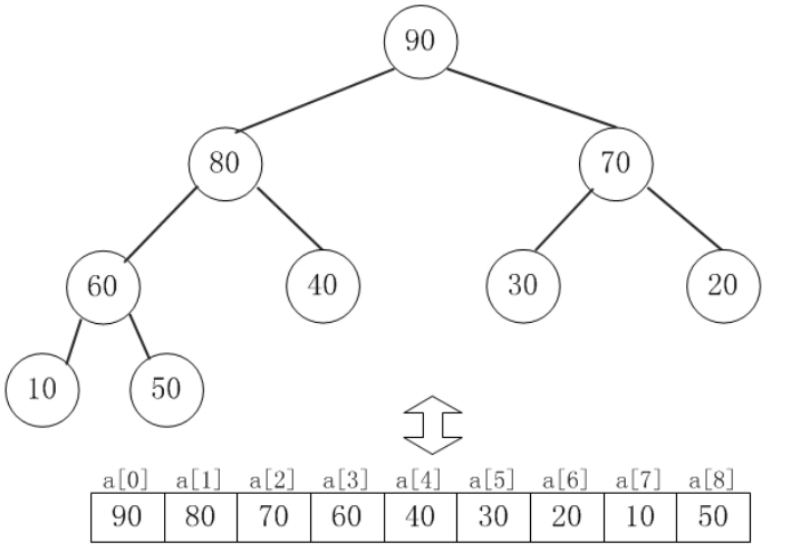
2.堆排序

- 1.堆：堆通常是一个可以被看做一棵树，它满足下列性质：
  - 1）堆中任意节点的值总是不大于或不小于其子节点的值。不大于的叫做最小堆，不小于的叫做最大堆。

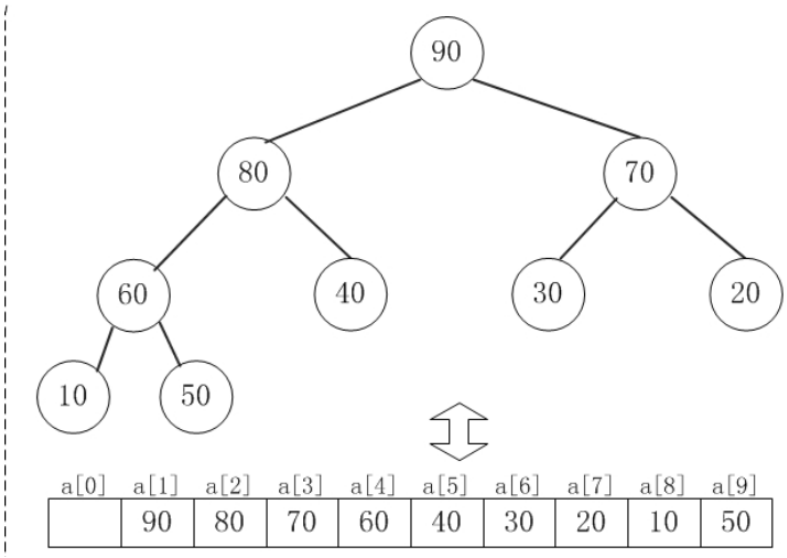
- 2）堆总是一个完全树。常见的堆有二叉堆、左倾堆、斜堆、二项堆、斐波那契堆等等。
- 3）二叉堆的最大堆以及最小堆



- 4）二叉堆
  - 二叉堆一般都通过"数组"来实现。数组实现的二叉堆，父节点和子节点的位置存在一定的关系。有时候，我们将"二叉堆的第一个元素"放在数组索引0的位置，有时候放在1的位置。
  - ①当二叉堆的第一个元素放在0时
    - 下标为i的左孩子下标为:$2*i+1$
    - 下标为i的右孩子下标为:$2*i+2$
    - 下标为i的父节点索引为:$int((i-1)/2)$



  - ②当二叉堆的第一个元素放在1时
    - 下标为i的元素的左孩子下标为：$2*i$
    - 下标为i的元素的右孩子下标为：$2*i+1$
    - 下标为i的元素的父节点下标为：$int(i/2)$



- 2.定义：
  - 通过上面对 堆的介绍，我们知道堆分为了最大堆和最小堆，最大堆的根结点就是堆中最大的那个数，最小堆的根结点就是堆中最小的那个数。堆排序正是根据这一特点，每次找到最大数/最小数，与最后一个数进行交换达到排序的效果。
  - 基本思想：
    - ① 初始化堆：将数列a[1...n]构造成最大堆。
    - ② 交换数据：将a[1]和a[n]交换，使a[n]是a[1...n]中的最大值；然后将a[1...n-1]重新调整为最大堆。 接着，将a[1]和a[n-1]交换，使a[n-1]是a[1...n-1]中的最大值；然后将a[1...n-2]重新调整为最大值。 依次类推，直到整个数列都是有序的。
- 3.代码
  - 构建最大堆(大根堆)

```
void maxHeapDown(int *a, int start, int end){
    int c = start;      // 当前结点
    int l = 2*c + 1;        // 结点的左孩子
    int tmp = a[c];         // 结点的值
    for (;l<=end; c=l,l=2*l+1) {
        // 找到左右孩子的最大值
        if (a[l] < a[l+1])  // 如果左子树的值 < 右孩子的值
            l++;     //l 指向右孩子
        // 如果根结点 > 左右孩子 就可以退出构建了
        // 因为 这是一个最大堆，左右孩子均为一个最大堆。所以 如果根结点已经是最大了 就表明它是这些节点中最大的了
        if (tmp >= a[l])
            break;
        else{   // 不是 就要跟l 交换，而且交换完，a[l]这个结点的值也就不一定比左右孩子都大了，就要继续调整下去
            a[c] = a[l];
            a[l] = tmp;
        }
    }
}
```

```
/**
 * 构建最大堆
 * @param a 数组
 * @param n 数组长度
 */
void buildMaxHeap(int *a,int n){
    // 最后一个下标为 n-1 if最后一个结点是右子树，那么他的父节点满足 2*i+2 = n-1 => i = n/2 - 1
                    //if 最后一个结点是左子树 那么 2*i +1 = n-1 => i = n/2 -1

    // 从最后一个结点的父节点开始
    for (int i = n/2 - 1; i >=0 ; --i) {
        maxHeapDown(a,i, end: n-1);
    }
}
```

- 堆排序

```
/**
 * 堆排序 升序
 * @param a
 * @param n
 */
void heap_sortASC(int *a,int n){
    buildMaxHeap(a,n);  // 构建最大堆
    for (int i = n-1; i > 0; --i) { // n-1次交换
        swap( &: a[0], &: a[i]);    // 将最后一个跟第一个进行交换
        buildMaxHeap(a, n: i-1);
    }
}
```

- 4.时间复杂度
  - 堆排序的时间复杂度是$O(N*lgN)$。
  - 假设被排序的数列中有N个数。遍历一趟的时间复杂度是O(N)，需要遍历多少次呢?
  - 堆排序是采用的二叉堆进行排序的，二叉堆就是一棵二叉树，它需要遍历的次数就是二叉树的深度，而根据完全二叉树的定义，它的深度至少是lg(N+1)。最多是多少呢？由于二叉堆是完全二叉树，因此，它的深度最多也不会超过lg(2N)。因此，遍历一趟的时间复杂度是O(N)，而遍历次数介于lg(N+1)和lg(2N)之间；因此得出它的时间复杂度是O(N*lgN)。
- 5.空间复杂度：O(1)
- 6.稳定性：不稳定

四、归并排序和基数排序

1.归并排序

- 将两个的有序数列合并成一个有序数列，我们称之为"**归并**"。
- 归并排序(Merge Sort)就是利用归并思想对数列进行排序。根据具体的实现，归并排序包括"**从上往下**"和"**从下往上**"2种方式。
- 1)从上往下：将数组一分为二，再二分为四…，最终最小区间的长度为1 递归退出

- 2)从下往上：将数组先分成n份，也就是区间长度为1，每两个相邻的进行合并排序，然后再使区间长度为2，这时就是4个进行合并排序，以此类推，直到分组区间成为n。
- 代码：
  - 1）从上至下
    -
```
void mergeSortUp2Down(int *a,int start,int end){
    if (a==NULL || start >= end)
        return;
    int mid = (start + end) / 2;
    mergeSortUp2Down(a,start,mid);  // 递归进行合并排序 得到的就是一个 [start ... mid] 的 有序数列
    mergeSortUp2Down(a, start: mid+1,end);  // 同理  得到的是 [mid+1 ... end] 有序数列
    merge(a,start,mid,end); // 合并
}

void merge(int *a,int start,int mid,int end){
    int *tmp =new int[end-start+1]; // tmp将两个区间进行合并
    int i = start;  // 区间一的起始下标
    int j = mid +1; // 区间二的起始下标
    int k = 0 ;     // 记录tmp数组的下标
    while (i<=mid && j<=end){
        if (a[i]<a[j])
            tmp[k++] = a[i++];
        else
            tmp[k++] = a[j++];
    }

    // 将还没有放入数组中的放入
    while (i <= mid)
        tmp[k++] = a[i++];
    while (j<=end)
        tmp[k++] = a[j++];

    // 将排序好的tmp 赋值给 a
    for (int l = 0; l < k; ++l) {
        a[start+l] = tmp[l];
    }

    delete[] tmp;    // 释放tmp
}
```
  - 2）从下至上
    -

```
void mergeSortDown2Up(int *a,int n){
    if (a == NULL || n <= 0)
        return;
    for (int i = 1; i < n; i*=2) {
        mergeGroup(a,i,n);
    }
}

void mergeGroup(int *a, int gap, int n) {
    int i ;
    for (i = 0; i + 2*gap <n  ; i += 2*gap) {
        merge(a,i, mid: i+gap-1, end: i+2*gap-1);
    }

    //如果有一个区间没有合并 ， 就让他跟前一个进行合并
    if (i+gap<n){
        merge(a,i, mid: i+gap-1, end: n-1);
    }
}
```

- 时间复杂度
  - 归并排序的时间复杂度是O(N*lgN)。
  - 假设被排序的数列中有N个数。遍历一趟的时间复杂度是O(N)，需要遍历多少次呢?
  - 归并排序的形式就是一棵二叉树，它需要遍历的次数就是二叉树的深度，而根据完全二叉树的可以得出它的时间复杂度是O(N*lgN)。
- 稳定性：稳定

2.计数排序

- 1.定义：计数排序（Counting Sort）是一种针对于特定范围之间的整数进行排序的算法。它通过统计给定数组中不同元素的数量（类似于哈希映射），然后对映射后的数组进行排序输出即可。
- 2.算法思想：计数排序的思想十分简单，就如他的名称一般。
  - 1）首先需要做的就是计数，统计数组中每个元素的数量。在计数排序中我们需要确定数组中的最大值和最小值，然后生成一个 count[min,max]的数组，对整个数组进行遍历，count[a[i]]++ 即为计数。
  - 2）确定每个数的所在位置范围。比如，现在我们有一个数列 [1,1,2,2,2,3,3,3,4],1的范围就是 [0,1],2的范围就是[2,3,4], count[i] + count[i-1] - 1 就是当前元素最后一个的下标值。

- 3) 为了确保是一个稳定排序，就需要从后往前开始遍历。

- 3.算法步骤
  - （1）找出待排序的数组中最大和最小的元素
  - （2）统计数组中每个值为i的元素出现的次数，存入数组C的第i项
  - （3）对所有的计数累加（从C中的第一个元素开始，每一项和前一项相加）
  - （4）反向填充目标数组：将每个元素i放在新数组的第C(i)项，每放一个元素就将C(i)减去1
  - 排序过程

- 4.代码
  -

```
void countSort(int *a,int n){
    int *count_arr = (int *) malloc(sizeof(int) * n);   // 计数数组
    int *result_arr = (int *)malloc(sizeof(int) * n);   // 排序结果数组

    // 计数数组 初始化
    for (int i = 0; i < n; ++i) {
        count_arr[i] = 0;
    }
    for (int j = 0; j < n; ++j) {
        count_arr[a[j]]++;      // 计数
    }

    // 找到每个元素的下标范围
    for (int k = 1; k < n; ++k) {
        count_arr[k] = count_arr[k] + count_arr[k-1];
    }
    for (int i = n-1; i >= 0 ; i--) {
        result_arr[count_arr[a[i]] - 1] = a[i];
        count_arr[a[i]] --;
    }
    for (int i = 0; i < n; ++i) {
        a[i] = result_arr[i];
    }
}
```

- 3.桶排序
  - 定义：三分钟搞懂桶排序
  - 代码：

```
//五分钟学算法：C++代码实现

void bucketSort(double* a,int n)
{
    int i,j;
    Head head[10]={NULL};
    Node * p;
    Node * q;
    Node * node;
    for(i=0;i<=n;i++){
        node=(Node*)malloc(sizeof(Node));
        node->key=a[i];
        node->next=NULL;
        p = q =head[(int)(a[i]*10)].next;
        if(p == NULL){
            head[(int)(a[i]*10)].next=node;
            continue;
        }
        while(p){
            if(node->key < p->key)
                break;
            q=p;
            p=p->next;
        }
        if(p == NULL){
            q->next=node;
        }else{
            node->next=p;
            q->next=node;
        }
    }
    j=0;
    for(i=0;i<10;i++){
        p=head[i].next;
        while(p){
            a[j++]=p->key;
            p=p->next;
        }
    }
```

- 4.基数排序
  - 1.定义：基数排序 (Radix Sort) 是一种非比较型整数排序算法，其原理是将整数按位数切割成不同的数字，然后按每个位数分别比较。基数排序法会使用到桶 (Bucket)，通过将要比较的位（个位、十位、百位...），将要排序的元素分配至 0~9 个桶中，借以达到排序的作用，在某些时候，基数排序法的效率高于其它的比较性排序法。演示链接
  - 2.算法步骤
    - 将所有待比较数值（正整数）统一为同样的数位长度，数位较短的数前面补零
    - 从最低位开始，依次进行一次排序
    - 从最低位排序一直到最高位排序完成以后, 数列就变成一个有序序列
  - 3. 基数排序 vs 计数排序 vs 桶排序
    - 这三种排序算法都利用了桶的概念，但对桶的使用方法上有明显差异：
    - 基数排序：根据键值的每位数字来分配桶；
    - 计数排序：每个桶只存储单一键值；
    - 桶排序：每个桶存储一定范围的数值；
  - 4.算法：
    -

```
void radixSort(int *a,int n){
    int d = getMax(a,n);
    int *tmp = (int *) malloc(sizeof(int) * n);
    int *count = (int *) malloc(sizeof(int) * n);
    int radix = 1;        // 第几位数
     for (int i = 1; i <=d ; ++i) {  // 执行d次， 从个位 到 d位
        // 每次 都要把桶 初始化
        for (int j = 0; j < 10; ++j) {
            count[j] = 0;
        }
        // 提出每个数的第d位，然后使用计数算法
        for (int k = 0; k < n; ++k) {
            int num = (a[k] / radix) % 10;  // 提出个位、十位...
            count[num]++;
        }

        // 计数排序的常规操作
        for (int j = 1; j < 10; ++j) {
            count[j] += count[j-1];
        }

        for (int j = n-1; j >=0 ; --j) {
            int k = (a[j] / radix) % 10;
            tmp[count[k]-1] = a[j];
            count[k]--;
        }
        // 将tmp 转移到 a 中
        for (int j = 0; j < n; ++j) {
            a[j] = tmp[j];
        }
        radix *= 10;     // 进行下一个基数的比较
    }
}
```

五、排序算法比较

- 1.各排序算法复杂度、稳定性

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(log_2n)$	不稳定
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

- 2.排序算法