稳定性：排序前后两个相同的值的前后顺序不会被改变

意义：

1）在实际的应用中，我们交换的不一定只是一个整数，而可能是一个很大的对象，交换元素存在一定的开销；

2）参照基数排序（后面会讲），不稳定排序是无法完成基数排序的，讲述完基数排序后，还会补充这里的原因。

堆排序、快速排序、希尔排序、直接选择排序不是稳定的排序算法，而基数排序、冒泡排序、直接插入排序、折半插入排序、归并排序是稳定的排序算法。

排序算法适用场景

- 若n较小(如n≤50)，可采用直接插入或直接选择排序。当记录规模较小时，直接插入排序较好，否则因为直接选择移动的记录数少于直接插人，应选直接选择排序为宜。
- 若序列初始状态基本有序，则直接插入和冒泡最佳，随机的快速排序也不错。插入排序对部分有序的数组很有效，所需的比较次数平均只有选择排序的一半。
- 若n较大，则应采用时间复杂度为O(nlgn)的排序方法：快速排序、堆排序或归并排序。
    - 快速排序是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；
    - 堆排序所需的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏情况。但这两种排序都是不稳定的。
    - 若要求排序稳定，则可选用归并排序。两两归并的排序算法并不值得提倡，通常可以将它和直接插入排序结合在一起使用。先利用直接插入排序求得较长的有序子文件，然后再两两归并之。因为直接插入排序是稳定的，所以改进后的归并排序仍是稳定的。
- 希尔排序比插入排序和选择排序要快得多，并且数组越大，优势越大。如果需要解决一个排序问题而又没有系统排序函数可用（例如直接接触硬件或者运行于嵌入式系统中的代码），可以先用希尔排序，再考虑是否替换为更复杂的排序算法。而对于部分有序和小规模的数组，应使用插入排序。
- 归并排序可以处理数百万甚至更大规模的数组，但是插入排序和选择排序做不到。归并排序的主要缺点是辅助数组所使用的额外空间和n的大小成正比。
- 快速排序的优点是原地排序（只需要一个很小的辅助栈），但是基准的选取是个问题，对于小数组，快速排序要比插入排序慢。
- 堆排序的优点是在排序时可以将需要排序的数组本身作为堆，无需任何额外空间，与选择排序有些类似，但所需的比较要少得多，堆排序适合例如嵌入式系统或低成本移动设备中容量有限的场景。

时间复杂度分析

> 1、冒泡排序不管序列是怎样，都是要比较n(n-1)/2 次的，最好、最坏、平均时间复杂度都为$O(n^2)$，需要一个临时变量用来交换数组内数据位置，所以空间复杂度为O(1)。有很多人说冒泡排序的最优的时间复杂度为O(n),其实这是在代码中使用一个标志位来判断是否已经排序好的,是冒泡排序的优化版，如果元素已经排序好，那么循环一次就直接退出。

2、选择排序是冒泡排序的改进，同样选择排序无论序列是怎样的都是要比较n(n-1)/2次的，最好、最坏、平均时间复杂度也都为O(n^2)，需要一个临时变量用来交换数组内数据位置，所以空间复杂度为O(1)。

3、插入排序不同，如果序列是完全有序的，插入排序只要比较n次，无需移动时间复杂度为O(n)，如果序列是逆序的，插入排序要比较O(n^2)和移动O(n^2)，所以平均复杂度为O(n^2)，最好为O(n)，最坏为O(n^2)，排序过程中只要一个辅助空间，所以空间复杂度O(1)。

4、快速排序的时间复杂度最好是O(nlog2n),平均也是O(nlog2n),最坏情况是序列本来就是有序的，此时时间复杂度为O(n^2)，快速排序的空间复杂度可以理解为递归的深度，而递归的实现依靠栈，平均需要递归logn次，所以平均空间复杂度为O(log2n)。

5、归并排序需要一个临时temp[]来储存归并的结果，空间复杂度为O(n)，时间复杂度为O(nlog2n)，可以将空间复杂度由 O(n) 降低至 O(1)，然而相对的时间复杂度则由 O(nlogn) 升至 O(n^2)。

6、希尔排序的时间复杂度分析及其复杂，有的增量序列的复杂度至今还没人能够证明出来，只需要记住结论就行，{1,2,4,8,...}这种序列并不是很好的增量序列，使用这个增量序列的时间复杂度（最坏情形）是O(n^2)，Hibbard提出了另一个增量序列{1,3,7，...,2^k-1}，这种序列的时间复杂度(最坏情形)为O(n^1.5)，Sedgewick提出了几种增量序列，其最坏情形运行时间为O（n^1.3）,其中最好的一个序列是{1,5,19,41,109,...}，需要一个临时变量用来交换数组内数据位置，所以空间复杂度为O(1)。

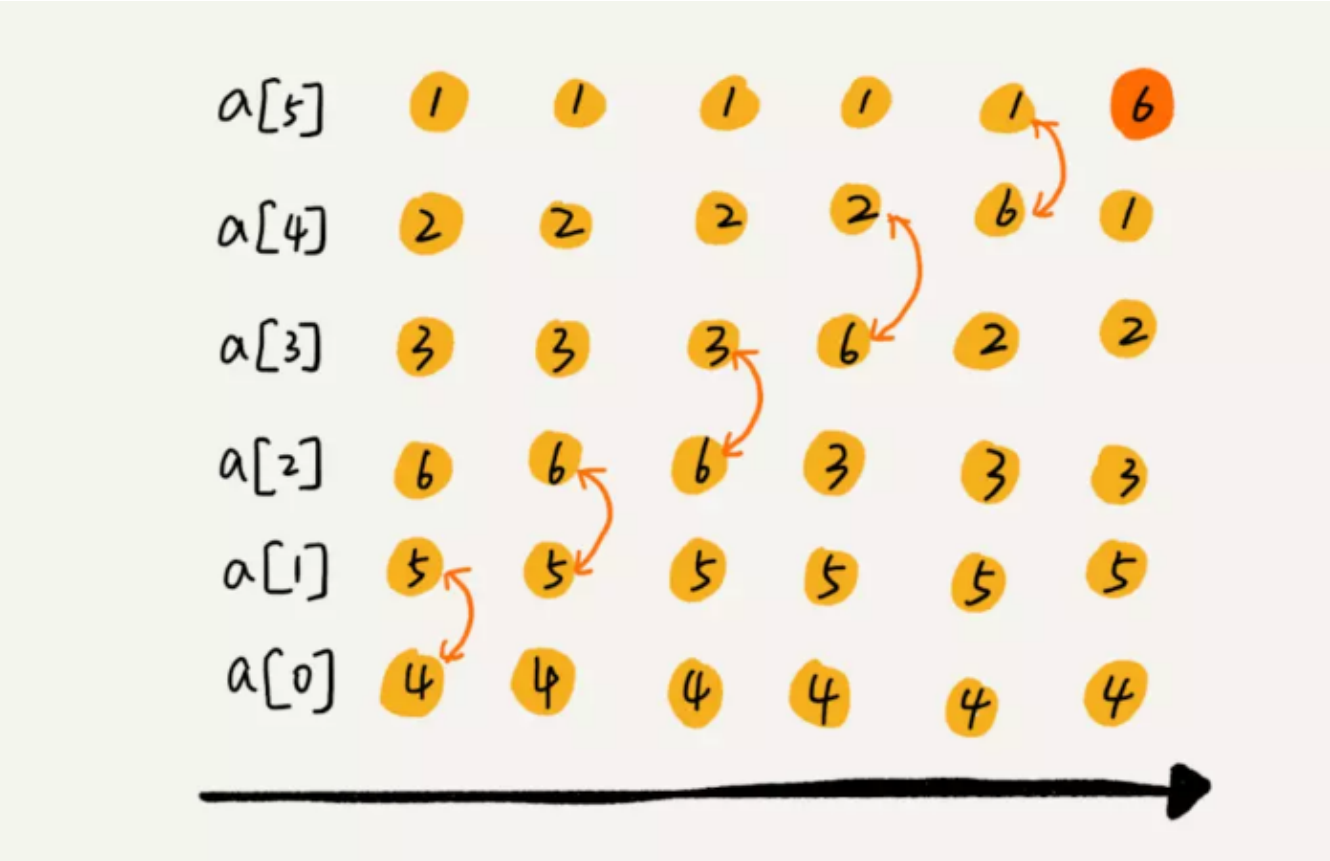
7、堆排序的时间复杂度，主要在初始化堆过程和每次选取最大数后重新建堆的过程，初始化建堆时的时间复杂度为O(n)，更改堆元素后重建堆的时间复杂度为O(nlog2n)，所以堆排序的平均、最好、最坏时间复杂度都为O(nlog2n)，堆排序是就地排序，空间复杂度为常数O(1)。

8、基数排序对于 n 个记录，执行一次分配和收集的时间为O(n+r)，如果关键字有 d 位，则要执行 d 遍，所以总的时间复杂度为 O(d(n+r))。该算法的空间复杂度就是在分配元素时，使用的桶空间，空间复杂度为O(r+n)=O(n)

排序算法	平均时间复杂度	最差时间复杂度	空间复杂度	数据对象稳定性
冒泡排序	O(n2)	O(n2)	O(1)	稳定
选择排序	O(n2)	O(n2)	O(1)	数组不稳定、链表稳定
插入排序	O(n2)	O(n2)	O(1)	稳定
快速排序	O(n*log2n)	O(n2)	O(log2n)	不稳定
堆排序	O(n*log2n)	O(n*log2n)	O(1)	不稳定
归并排序	O(n*log2n)	O(n*log2n)	O(n)	稳定
希尔排序	O(n*log2n)	O(n2)	O(1)	不稳定
计数排序	O(n+m)	O(n+m)	O(n+m)	稳定
桶排序	O(n)	O(n)	O(m)	稳定
基数排序	O(k*n)	O(n2)		稳定

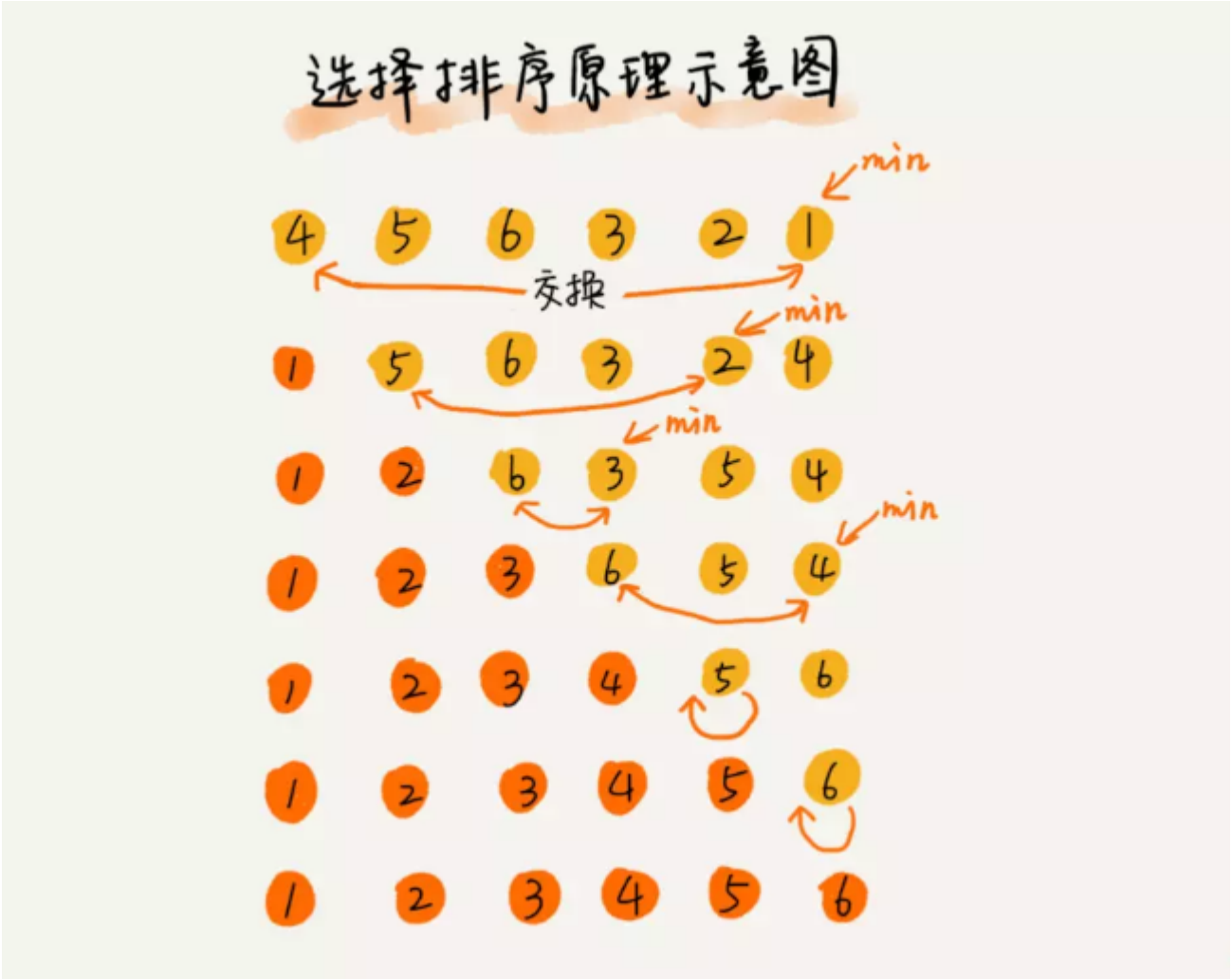
1.冒泡排序

```
void BubbleSort(vector<int>& v)
{
    auto len=v.size();
    for(auto i=0;i<len-1;i++)  //整个大循环执行一次就冒出一个最大的，执行len-1次就决定了len了个元素
    {
        for(auto j=0;j<len-1-i;j++)  //len-1很好理解，因为j+1才是访问数据的边界
        {
            if(v[j]>v[j+1])
                swap(v[j],v[j+1]);
        }
    }
}
```



2.选择排序

```
void SelectionSort(vector<int>& v)
{
    int min, len = v.size();
    for (int i = 0; i < len - 1; ++i)  //每次将最小数放左面，len个元素只需len-1次
    {
        min = i;
        for (int j = i + 1; j < len; ++j)
        {
            if (v[j] < v[min])
            {
                min = j;  //这样就避免了无用功的交换
            }
        }
        if (i != min)
            swap(v[i], v[min]);
    }
}
```



3.插入排序

```
void InsertSort(vector<int>& v)
{
    int len = v.size();
    for (int i = 1; i < len; ++i)
    {
        int temp = v[i];
        for(int j = i - 1; j >= 0; --j)
        {
            if(v[j] > temp)
            {
                v[j + 1] = v[j];
                v[j] = temp;
            }
            else
                break;
        }
    }
}
```

4.快排

```
int division(vector<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;
}

void quickSort(vector<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);

        // 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序
        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
        quickSort(list, base + 1, right);
    }
}
```

5.堆排序

一般升序采用大顶堆，降序采用小顶堆

堆排序的基本思想是：将待排序序列构造成一个大顶堆（这个过程中注意当交换当前上下节点可能导致整个下面的结构不符合堆，又要调整整个下面的结构），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

```
#include <iostream>
```

```cpp
#include <algorithm>
using namespace std;

// 堆排序：（最大堆，有序区）。从堆顶把根卸出来放在有序区之前，再恢复堆。

void max_heapify(int arr[], int start, int end) {
    //建立父節點指標和子節點指標
    int dad = start;
    int son = dad * 2 + 1;
    while (son <= end) { //若子节点指标在范围内才做比较
        if (son + 1 <= end && arr[son] < arr[son + 1]) //先比较两个子节点，选择最大的
            son++;
        if (arr[dad] > arr[son]) //如果父节点大于子节点，直接返回（就该函数来说此时可能左右子树
            //不符合堆，但是要看下该函数被调用的两次情况：第一次是从下往上整理，这时发现父节点本身符合
            // 要求并且下方也符合要求，直接返回没问题。第二次也是下方都符合要求，就根节点被换了
            // 此时如果根节点也符合要求，直接返回也没问题）
            return;
        else { //否则交换父子节点内容并向下整理
            swap(arr[dad], arr[son]);
            dad = son;
            son = dad * 2 + 1;
        }
    }
}

void heap_sort(int arr[], int len) {
    //初始化，i從最後一個父節點開始調整
    for (int i = len / 2 - 1; i >= 0; i--)
        max_heapify(arr, i, len - 1);
    //先將第一個元素和已经排好的元素前一位做交換，再從新調整(刚调整的元素之前的元素)
    //，直到排序完成
    for (int i = len - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        max_heapify(arr, 0, i - 1);
    }
}

int main() {
    int arr[] = { 3, 5, 3, 0, 8, 6, 1, 5, 8, 6, 2, 4, 9, 4, 7, 0, 1, 8, 9, \
        7, 3, 1, 2, 5, 9, 7, 4, 0, 2, 6 };
    int len = (int) sizeof(arr) / sizeof(*arr);
    heap_sort(arr, len);
    for (int i = 0; i < len; i++)
        cout << arr[i] << ' ';
    cout << endl;
    return 0;
}
```

6.二分查找

```
int BinarySearch2(vector<int> v, int value, int low, int high)
{
 if (low > high)
     return -1;
 int mid = low + (high - low) / 2;       //这里可以防止溢出
 if (v[mid] == value)
     return mid;
 else if (v[mid] > value)
     return BinarySearch2(v, value, low, mid - 1);
 else
     return BinarySearch2(v, value, mid + 1, high);
}
```

7.常见数据结构

满二叉树：除了叶结点外每一个结点都有左右子叶且叶结点都处在最底层的二叉树

完全二叉树：在满二叉树基础上可以从右往左减少叶子节点

平衡二叉树（AVL）：或者是一棵空树，或者是具有以下性质的二叉树：

（1）左子树和右子树都是平衡二叉树；

（2）左子树和右子树的深度（高度）之差的绝对值不超过1。

堆：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

大顶堆：arr[i] >= arr[2i+1] && arr[i] >= arr[2i+2]

小顶堆：arr[i] <= arr[2i+1] && arr[i] <= arr[2i+2]

7.