- 直接插入排序

定义：将一个数据插入到已经排好序的数组中，从而得到一个新的个数加一的有序数组。

实现：

```
//插入排序---升序（从后向前查找插入位置）
void InsertSort(int* a,    size_t n)
{
    assert(a);

    for(size_t i=1; i<n; ++i)
    {
        int end = i-1;
        int tmp = a[i];

        while(a[end] > tmp && end >= 0) //挪动数据

        {
            a[end+1] = a[end];
            --end;
        }

        a[++end] = tmp; //插入数据

    }
}
```

时间复杂度：

平均情况下：O(n^2)

最坏情况下：O(n^2)

最优情况下时间复杂度为：O(n). (有序，每次不需要挪动数据)

空间复杂度： O（1）

稳定性：稳定

优化：因为寻找插入时，前面已经是有序的数组，所以可以使用二分查找进行优化。但时间复杂度仍然为 O(n^2)

- 希尔排序

定义：把记录按下标的一定增量分组，对每组记录采用直接插入排序；随着增量不断减少，每组包含的关键词越来越多，当增量减至 1 时，整个文件恰被分成一组，算法便终止。

实现：

```
void ShellSort(int* a, size_t n)
{
    assert(a);

    int gap = n;//表示每隔gap个数据分一组

    while(gap > 1)
    {
        gap = gap/3 + 1;
        for(size_t i=gap; i<n; i++)
        {
            int end = i-gap;
            int tmp = a[end+gap];
            while(a[end] > tmp && end >= 0)
            {
                a[end+gap] = a[end];
                end -= gap;
            }
            a[end+gap] = tmp;
        }
    }
}
```

时间复杂度：

平均情况下：O(n^1.3)

最好情况下：O(n)

最坏情况下：O(n^2)

空间复杂度为：O(1)

稳定性：不稳定（中间各分组交换数据时，可能将已经排序好的数据进行交换）

- 选择排序

定义：每一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，直到全部待排序的数据元素排序完成。

实现：

```
//选择排序
```

```
void SelectSort(int* arr, int n)
{
    //初始化：begin和end的起始位置
    int begin = 0;
    int end = n-1;

    while(begin < end)
    {
        //初始化， min和max , min和max均表示下标
        int min = begin;
        int max = end;

        //寻找最大、最小值
        for(int i=begin; i<=end; i++)
        {
            if(arr[i] > arr[max])
            {
                max = i;
            }

            if(arr[i] < arr[min])
            {
                min = i;
            }
        }

        if(max == begin) //当max == beign时应进行特殊处理
        {
            max = min;
        }

        swap(arr[min],arr[begin]);
        swap(arr[max],arr[end]);
        ++begin;
        --end;
    }
}
```

时间复杂度：

平均情况：O(n^2)

最好情况：O(n^2)

最坏情况：O(n^2)

空间复杂度：O(1)

稳定性：不稳定（两个相等的元素，位置可以不动，但在交换的过程中，可能将数据进行了交换）

● 堆排序

定义：利用堆这种数据结构设计的排序算法，利用数组的快速定位指定索引的元素。（升序→大堆， 降序—>小堆）

思想：

1. 先将数据建成一个大堆
2. 再将对顶元素与最后一个元素进行交换，由于交换后新的堆可能违反性质，故应将堆进行调整。

实现：

```
//堆排序

void AdjustDown(int* arr, int n, int root)//向下调整算法 (n表示一共有多少个元素)
{
    int parent = root;
    int child = 2*parent+1;

    while(child < n)
    {
        //找到左子节点和右子节点的较大结点
        if(child+1<n && arr[child]< arr[child+1])
        {
            ++child;
        }

        if(arr[child] > arr[parent])
        {
            swap(arr[parent],arr[child]);
```

```
            parent = child;
            child = parent*2+1;
        }
        else
        {
            break;
        }

    }
}


void HeapSort(int* arr, int n)
{
    assert(arr);

    for(int i=(n-1-1)/2; i>=0; --i)
    {
        AdjustDown(arr,n,i);
    }


    //排序
    int len = n-1;
    while(len > 0)
    {
        swap(arr[0],arr[len]);
        //因为调整算法中的n代表数组的个数，这里的len代表数组最后元素的
下标，所以这里使用len--
        AdjustDown(arr,len--,0);
    }
}
```

时间复杂度：

平均情况：O(n*lgn)

最好情况：O(n*lgn)

最坏情况：O(n*lgn)

时间复杂度：建堆是通过父节点和子节点两两比较并交换得到，时间复杂度

为 O（n），调整堆需要交换 n-1 次堆顶元素，并调整堆，调整堆的过程就是满二叉树的深度 logN，所以时间复杂度为 O(nlogn),，所以最终时间复杂度为 O(n) + O(nlogn)。

空间复杂度：O(1)

稳定性：不稳定

- 冒泡排序

定义：将大数和小数不断后移的一种思想，比较和交换都发生在两个相邻元素之间。

实现：

```
//冒泡排序
void BubbleSort(int* arr, int n)
{
    assert(arr);

    bool finish = true;//表示是否冒泡完成

    int end = n;//end表示每次冒泡的终止位置

    while(end > 0)
    {
        //单趟冒泡
        for(int i=1; i<end; ++i)
        {
            if(arr[i-1] > arr[i])
            {
                swap(arr[i-1],arr[i]);
                finish = false;
            }
        }

        if(finish == true)
        {
            return ;
        }

        --end;
```

```
    }
}
```

时间复杂度：
平均情况：O(n^2)
最好情况：O(n)
最坏情况：O(n^2)

空间复杂度：O(n)

稳定性：稳定

● 快速排序

定义：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后按此方法对着两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。
首先需要确定 key 值

实现：
a. 左右指针法

```
//快速排序

//左右指针法

int PartSort(int* arr, int begin, int end)//begin,end分别为开始和结束的坐标

{
    int left = begin, right = end;
    int key = arr[right];

    while(begin < end)
    {
        //begin找大于

        while(begin < end && arr[begin]<=key)
        {
            ++begin;
        }


        //end找小于

        while(begin<end && arr[end] >= key)
        {
```

```
            --end;
        }

        swap(arr[begin],arr[end]);
    }
    swap(arr[begin],arr[right]);

    return begin;
}

void QuickSort(int* arr, int left, int right)
{
    if(left >= right)
    {
        return ;
    }

    int div = PartSort(arr,left,right);
    QuickSort(arr,left,div-1);
    QuickSort(arr,div+1,right);
}
```

b. 挖坑法

```
//挖坑法(找到一个坑，从另一端找比它大数的数填充，坑的位置改变，在从另一端找比它小的数)
int PartSort2(int* a, int begin, int end)
{
    int key = a[end];
    while(begin < end)
    {
        while(begin<end && a[begin] <= key)
        {
            ++begin;
        }

        a[end] = a[begin];

        while(begin < end && a[end] >= key)
        {
            --end;
        }
        a[begin] = a[end];
    }
```

```
    a[begin] = key;

    return begin;
}
```

c. 前后指针法

```
//前后指针法 （定义prev 和 cur ） (****算法的很好优化)
int PartSort3(int* a, int begin, int end)
{
    int key = a[end];
    int cur = begin, prev = cur-1; //cur=begin ,不能为0，因为区间不能不一定从0
开始

    while(cur < end)
    {
        if(a[cur] < key && ++prev != cur)
        {
            swap(a[prev], a[cur]);
        }
        ++cur;
    }
    swap(a[++prev],a[end]);
    return prev;
}
```

d. 非递归法

```
//非递归实现（多路递归，借助栈）
//思路：栈中保存左右下标，先压右、再压左，栈不为空取出，栈为空时循环
结束
void QuickSortFD(int* a, int left, int right)
{
    stack<int> s; //栈中存放下标

    if(left < right) //首先要保证下标有意义再压栈
    {
        s.push(right);
        s.push(left);
    }
```

```
    while(s.size()>0)
    {

        int left = s.top();
        s.pop();
        int right = s.top();
        s.pop();
        if(right - left <= 20)
        {
            InsertSort(a+left,right-left+1);
            return;
        }
        else
        {
            int div = PartSort1(a,left,right);


            if(div - 1 >left) //区间不合法时，不入栈
            {
                s.push(div-1);
                s.push(left);
            }

            if(right > div+1)
            {
                s.push(right);
                s.push(left);
            }
        }
    }
}
```

优化：三数取中法

```
//三数取中法(返回值是中间这个数值，还是中间数值的小标)？？？
int GetMidIndex(int* a, int begin, int end)
{
    int mid = begin + ((end-begin)>>1);
    //选择三个数的中间值
    if(a[begin] > a[mid])
    {
        if(a[mid]>a[end])//begin>mid>and
        {
            return mid;
```

```
            }
            else if(a[begin] > a[end])//begin>mid,mid<end,begin>end
            {
                return end;
            }
            else
            {
                return begin;
            }
        }
        else //mid>begin
        {
            if(a[end]>a[mid])
            {
                return mid;
            }
            else if(a[end]>a[begin])
            {
                return end;
            }
            else
            {
                return begin;
            }
        }
}

//使用三数取中法
int PartSort1PMid(int* a, int begin, int end)
{
    int left = begin, right = end; //right赋值为  end  还是  end-1


    int mid = GetMidIndex(a,begin,end); //三数取中法

    swap(a[right],a[mid]);//交换中间与右边的数


    int key = a[right]; //选择最右边为key值


    //单趟排序
```

```
    while(begin < end)
    {
        //begin找大于

        while(begin < end && a[begin] <= key) //注意条件a[begin] <= key
        {
            ++begin;
        }


        //end找小
        while(begin < end && a[end] >= key)
        {
            --end;
        }

        swap(a[begin], a[end]);
    }
    swap(a[begin],a[right]); //a[right]相当于中间划分的位置


    return begin;
}
```

时间复杂度(用求递归时间复杂度进行计算：即递归次数+递归深度)
平均情况时间复杂： O(nlgn)
最好情况： O(nlgn)
最坏情况： O(n^2)

空间复杂度：
首先就地快速排序使用的空间是 O(1)的，也就是个常数级的；而真正消耗空间的就是递归调用了，因为每次递归就要保持一些数据；
最优的情况下空间复杂度为 O(logn)；每一次都平分数组的情况。
最差的情况下空间复杂度为 O(n)；退化为冒泡排序的情况。
稳定性： 不稳定

- 归并排序

定义：
采用分治法，将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序。
实现：

```
//归并排序
```

```
void _MergeSort(int* arr, int left, int right, int* tmp)//tmp临时空间
{
    if(left >= right)//说明区间无效
    {
        return ;
    }

    //划分
    int mid = left+((right-left)>>1);
    _MergeSort(arr,left,mid,tmp);
    _MergeSort(arr,mid+1,right,tmp);

    //归并
    int index =left;
    int begin1 = left, end1 = mid;
    int begin2 = mid+1, end2=right;

    while(begin1<=end1 && begin2<=end2)
    {
        if(arr[begin1] <= arr[begin2])
        {
            tmp[index++] = arr[begin1++];
        }
        else
        {
            tmp[index++] = arr[begin2++];
        }
    }

    //拷贝begin和end剩下元素

    while(begin1<= end1) //注意：应为小于等于，边界的处理
    {
        tmp[index++] = arr[begin1++];
    }

    while(begin2 <= end2)
    {
        tmp[index++] = arr[begin2++];
```

```
    }

    //考回原数组
    index = left;
    while(index <= right)
    {
        arr[index] = tmp[index];
        index++;
    }
}

void MergeSort(int* arr, int n)
{
    assert(arr);

    int* tmp = new int[n];
    _MergeSort(arr,0,n-1,tmp);
    delete[] tmp;
}
```

时间复杂度：
平均情况： O(n*lgN)
最好情况： O(n*lgN)
最坏情况： O(n*lgN)
空间复杂度： O(N)
稳定性： 稳定

注意：各排序算法的边界处理，以及各参数值的设计。

非比较排序

- ## 计数排序

它的优势在与对一定范围内的整数排序时，它的复杂度为 O(n+k)（其中 k 是整数的范围），快于任何比较排序算法。
https://blog.csdn.net/qq_36528114/article/details/78676960

计数排序的思想类似于哈希表中的直接定址法，在给定的一组序列中，先找出该序列中的最大值和最小值，从而确定需要开辟多大的辅助空间，每一个数在对应的辅助空间中都有唯一的下标。

1. 找出序列中最大值和最小值，开辟 Max-Min+1 的辅助空间
2. 最小的数对应下标为 0 的位置，遇到一个数就给对应下标处的值+1,。
3. 遍历一遍辅助空间，就可以得到有序的一组序列

算法分析：

计数排序是一种以空间换时间的排序算法，并且只适用于待排序列中所有的数较为集中时，比如一组序列中的数据为 **0 1 2 3 4 999**;就得开辟 1000 个辅助空间。

时间复杂度

计数排序的时间度理论为 O(n+k),其中 k 为序列中数的范围。

不过当 O(k)>O(n*log(n))的时候其效率反而不如基于比较的排序（基于比较的排序的时间复杂度在理论上的下限是 O(n*log(n)), 如归并排序，堆排序）

实现：

```
void CountSort(int* a, int n)
{
    //统计最大数，最小数方便开空间
    int max = a[0],min = a[0];
    for(int i=0; i<n; i++)
    {
        if(a[i]>max)
        {
            max = a[i];
        }

        if(a[i]<min)
        {
            min = a[i];
        }
    }

    //使用直接地址法
    int range = max - min - 1;
    int* hashtable = new int[range]; //new的数组需不需要进行初始化

    for(size_t i=0; i<n; i++)
    {
        hashtable[a[i]-min]++;
    }

    size_t j=0;
    for(size_t i=0; i<range; i++)
    {
        while(hashtable[i]--)
        {
            a[j] = i+min;
            ++j;
```

```
        }
    }

    delete[] hashtable;

}
```

- **基数排序**

它的基本思想是：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。具体做法是：将所有待比较数值统一为同样的数位长度，数位较短的数前面补零。然后，从最低位开始，依次进行一次排序。这样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。
https://www.cnblogs.com/skywang12345/p/3603669.html

- **桶排序**

将排序的数据放到桶里，初始时设置桶的数量，即排序的范围，如若要对 100 范围内的某 10 个数排序，即设置桶的数量为 100，然后分别编号 1 到 100，将要排序的 10 个数，放到与桶编号匹配的桶中。并将该编号的桶设置一个标志位，标志桶内有数据，输出时只要遍历所有桶，选择有数据的桶，并按编号输出即可。
https://blog.csdn.net/mupengfei6688/article/details/53106267

计数排序与桶排序

https://blog.csdn.net/sunjinshengli/article/details/70738527