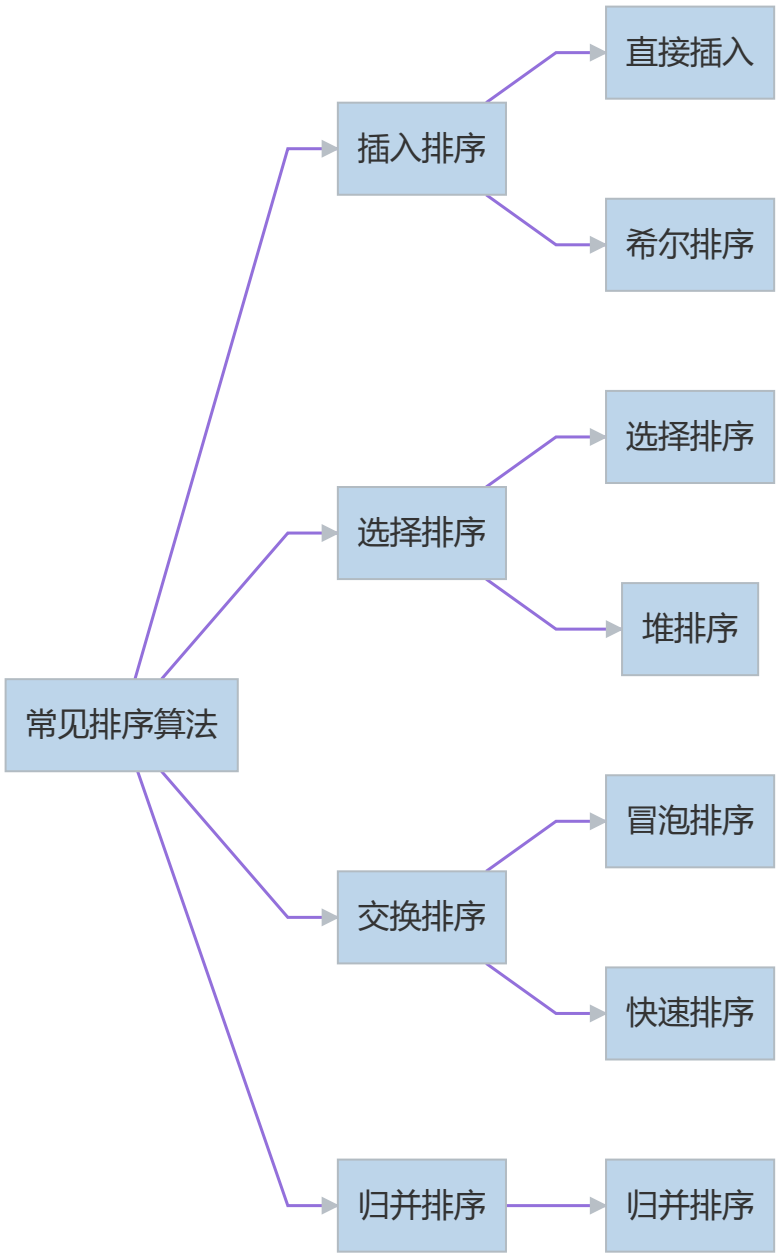
常见的排序算法



时间、空间复杂度和稳定性

排序方法	平均时间	最好时间	最坏时间	空间复杂度	稳定性
插入排序	O(N^2)	O(N)	O(N^2)	O(1)	稳定
希尔排序	O(n^1.3)	O(N)	O(N^2)	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(N*lgN)	O(N*lgN)	O(N*lgN)	O(1)	不稳定
冒泡排序	O(N^2)	O(N)	O(N^2)	O(1)	稳定
快速排序	O(N*lgN)	O(N*lgN)	O(N^2)	O(lgN)	不稳定
归并排序	O(N*lgN)	O(N*lgN)	O(N*lgN)	O(N)	稳定

选择排序算法准则：

每种排序算法都各有优缺点。

影响排序的因素有很多，平均时间复杂度低的算法并不一定就是最优的。相反，有时平均时间复杂度高的算法可能更适合某些特殊情况。同时，选择算法时还得考虑它的可读性，以利于软件的维护。一般而言，需要考虑的因素有以下四点：

1．待排序的记录数目n的大小；

2．记录本身数据量的大小，也就是记录中除关键字外的其他信息量的大小；

3．关键字的结构及其分布情况；

4．对排序稳定性的要求。

设待排序元素的个数为n.
1）当n较大，则应采用时间复杂度为O(nlog2n)的排序方法：快速排序、堆排序或归并排序序。
　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： 如果内存空间允许且要求稳定性的，
　归并排序：它有一定数量的数据移动，所以我们可能过与插入排序组合，先获得一定长度的序列，然后再合并，在效率上将有所提高。
2） 当n较大，内存空间允许，且要求稳定性 ——归并排序
3）当n较小，可采用直接插入或直接选择排序。

　直接插入排序：当元素分布有序，直接插入排序将大大减少比较次数和移动记录的次数。
　直接选择排序： 元素分布有序，如果不要求稳定性，选择直接选择排序
4）一般不使用或不直接使用传统的冒泡排序。

5）基数排序

它是一种稳定的排序算法，但有一定的局限性：

1、关键字可分解。

2、记录的关键字位数较少，如果密集更好

3、如果是数字时，最好是无符号的，否则将增加相应的映射复杂度，可先将其正负分开排序。

插入排序

直接插入排序

思想：将数组中的所有元素依次和前面的已经排好序的元素相比较（依次），如果选择的元素比已排序的元素小，则交换，直到全部元素都比较过。

代码实现：

Sort.h:

```
#pragma once
#include <iostream>
#include <assert.h>
using namespace std;
//直接插入排序
 void InsertSort (int* a,size_t n)
```

```
 {
     assert(a);
     for(size_t i = 1;i < n; ++i)//用end的位置控制边界
     {
         //单趟排序
         int end = i - 1 ;
         int tmp = a[i];
         while( end >= 0 )//循环继续条件
         {
             if( a[end] > tmp )
             {
                 a[end+1] = a[end];
                 --end;
             }
             else
                 break;
         }
         a[end+1] = tmp;
     }
 }
```

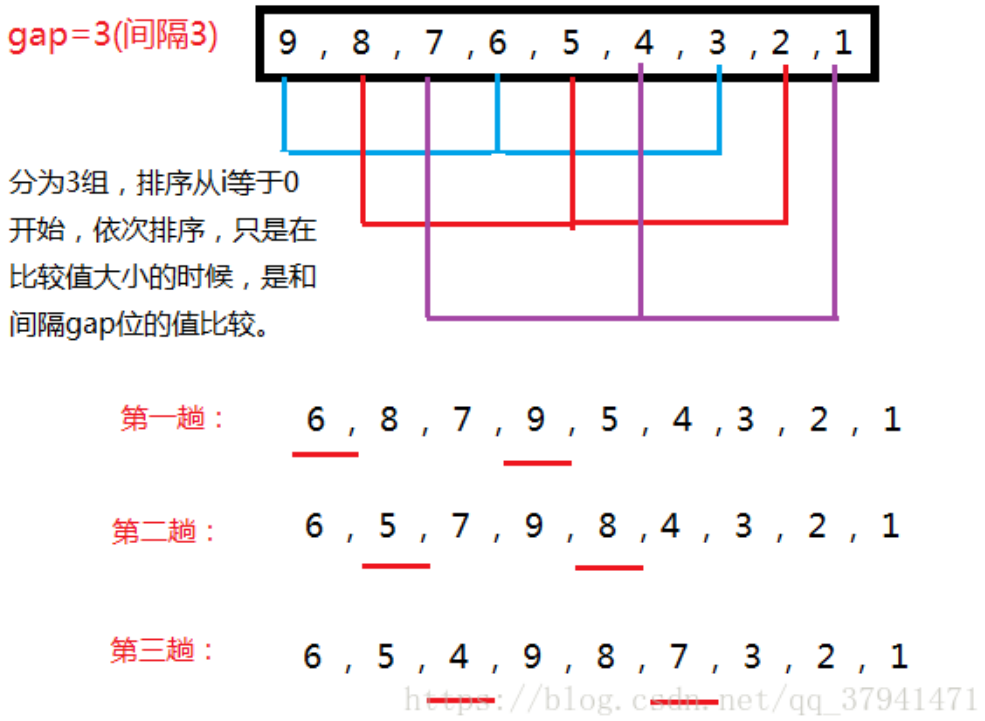
test.cpp :

```
#include "Sort.h"

 void Print(int a[],int len)
{
    for(int i = 0; i < len; ++i)
    {
        cout<<a[i]<<" ";
    }
    cout<<endl;
}

 void test()
 {
     //升序排序
     int a [] = {2,5,7,6,12,4,3,9,0};
     int len = sizeof(a)/sizeof(a[0]);
     cout<<"before sort :";
     Print(a,len);

     InsertSort(a,len);
     cout<<"after sort :";
     Print(a,len);
```

```
 }
int main ()
{
    test();
    return 0;
}
```

希尔排序

思想：希尔排序也称缩小增量排序；希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序，随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时（用gap = gap/3+1 控制），保证了最后一次进行直接插入排序，算法终止。（其中直接插入排序是希尔排序gap=1的特例）另外，gap越大，值越大的容易到最后面，但是不太接近有序。—— 一般gap不要超过数组大小的一半



代码实现：

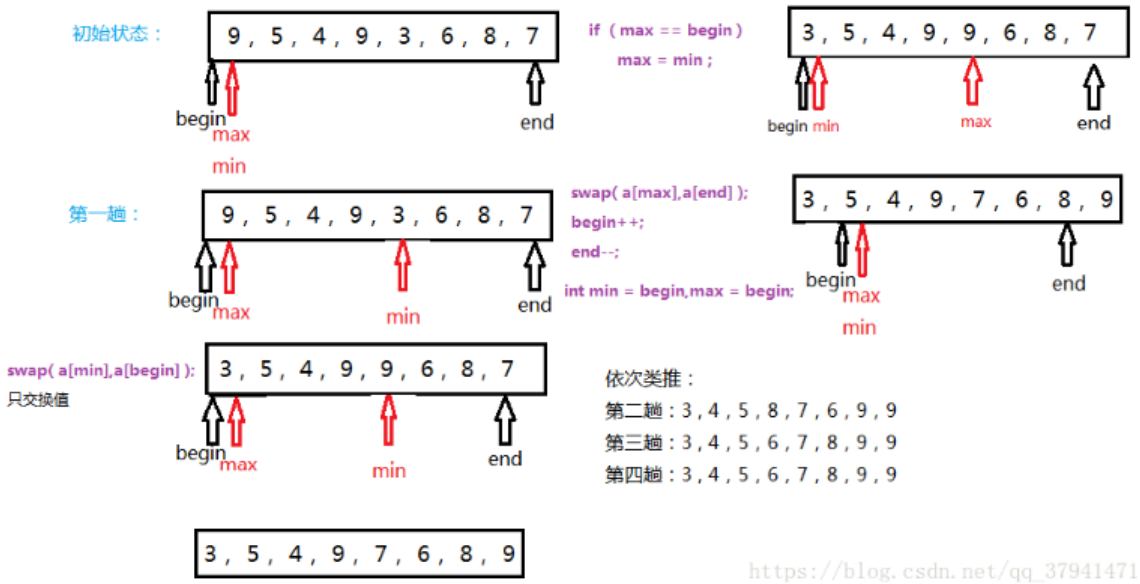
```
void ShellSort(int* a,size_t n)
 {
     assert(a);
     //1. gap > 1 预排序
     //2. gap == 1 直接插入排序
     //3. gap = gap/3 + 1; 保证最后一次排序是直接插入排序
```

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

}
```

选择排序

选择排序

思想：在一个无序数组中选择出每一轮中最值元素，然后把这一轮中最前面的元素和min交换，最后面的元素和max交换；然后缩小范围（开始位置（begin++）++，最后位置（end-）--）,重复上面步骤，最终得到有序序列（升序）。



代码实现：

```
void SelectSort(int* a,size_t n)
{
    assert(a);
    int begin = 0;
    int end = n-1;
    while ( begin < end )
    {
        int min = begin,max = begin;
        for(int i = begin; i <= end; ++i)
        {
            if( a[min] > a[i] )
                min = i;
            if( a[max] < a[i] )
                max = i;
        }
        swap( a[min],a[begin] );
        if( max == begin )//如果首元素是最大的，则需要先把min 和 max的位置一换，再交换，否则经过两次交换，又回到原来的位置
            max = min;
        swap( a[max],a[end] );
        begin++;
        end--;
    }
}
```

堆排序

思想：
堆排序利用了大堆(或小堆)堆顶记录的关键字最大(或最小)这一特征，使得当前无序的序列中选择关键最大(或最小)的记录变得简单。（升序—建大堆，降序—建小堆）

代码实现：

```
class HeapSort {
public:
    void AdjustDown(int *A, int root, int n){
        int parent = root;
        int child = parent*2+1; // 左孩子
        while( child < n ){
```

```
            if( (child+1) < n && A[child+1] > A[child] ){
                ++child;
            }
            if( A[child] > A[parent] ){
                swap(A[child],A[parent]);
                parent = child;
                child = parent*2+1;
            }
            else
                break;
        }
    }
    int* heapSort(int* A, int n) {
        // write code here
        assert(A);
        // 找到倒数第一个非叶子节点----- 建大堆
        for( int i = (n-2)/2; i >=0 ; i-- ){
            AdjustDown(A,i,n);
        }

        int end = n-1;
        while( end > 0 ){
            swap(A[0],A[end]);
            AdjustDown(A,0,end); // end其实就是不算后面的一个元素，原因是最后一个节点已经是最大的
            end--;
        }
        return A;
    }
};
```

交换排序

冒泡排序

基本思想就是：从无序序列头部开始，进行两两比较，根据大小交换位置，直到最后将最大（小）的数据元素交换到了无序队列的队尾，从而成为有序序列的一部分；下一次继续这个过程，直到所有数据元素都排好序。
算法的核心在于每次通过两两比较交换位置，选出剩余无序序列里最大（小）的数据元素放到队尾。

代码实现：

```
void BubbleSort(int* a,size_t n)
{
    assert(a);
    size_t end = n;
    int exchange = 0;
    while( end > 0 )//end作为每趟排序的终止条件
    {
        for( size_t i = 1; i < end ; ++i )
        {
            if( a[i-1] > a[i] )
            {
                swap(a[i-1],a[i]);
                exchange = 1;
            }
        }
        if( 0 == exchange )//数组本身为升序，如果一趟排序结束，并没有进行交换，那么直接跳出循环（减少循环次数，升高效率）
            break;
        --end;
    }
}
```

快速排序

不稳定
速排序算法—左右指针法，挖坑法，前后指针法，递归和非递归
思想：分治法
将问题化为若干个子问题：将子问题排好序，实现最终的数组排序。
最优情况：当关键值位于序列中间时 O(N*lgN)
最坏情况：对已序的序列进行排序 O(N^2)

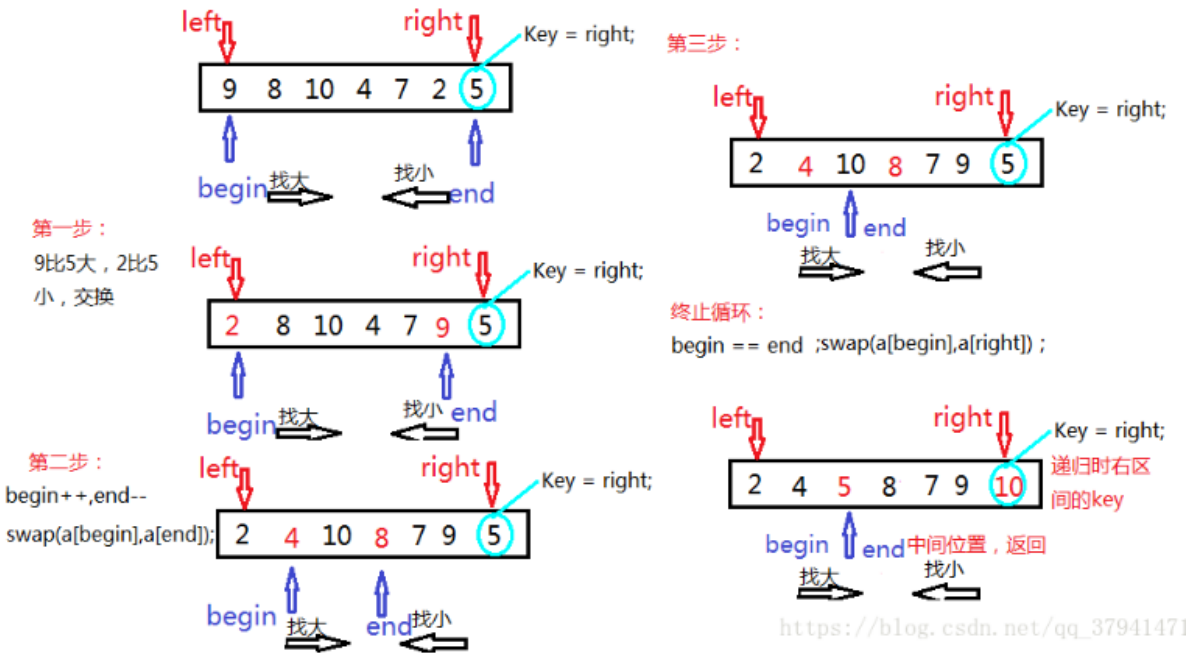
使用场景：
数据量大而杂乱的序列，避免出现最坏的情况（为了让效率更高，使用中间数法和在处理的数组大小小于10的时候，用直接插入排序，减少栈帧）。

递归

左右指针法

基本思路：

1．将数组的最后一个数right作为基准数key。
2．分区过程:从数组的首元素begin开始向后找比key大的数（begin找大）；end开始向前找比key小的数（end找小）；找到后然后两者交换（swap）,知道begin >= end终止遍历。最后将begin和最后一个数交换（这个时候end不是最后一个位置），即 key作为中间数（左区间都是比key小的数，右区间都是比key大的数）
3．再对左右区间重复第二步，直到各区间只有一个数。



代码：

```
int PartSort1(int* a,int left,int right)
{
```

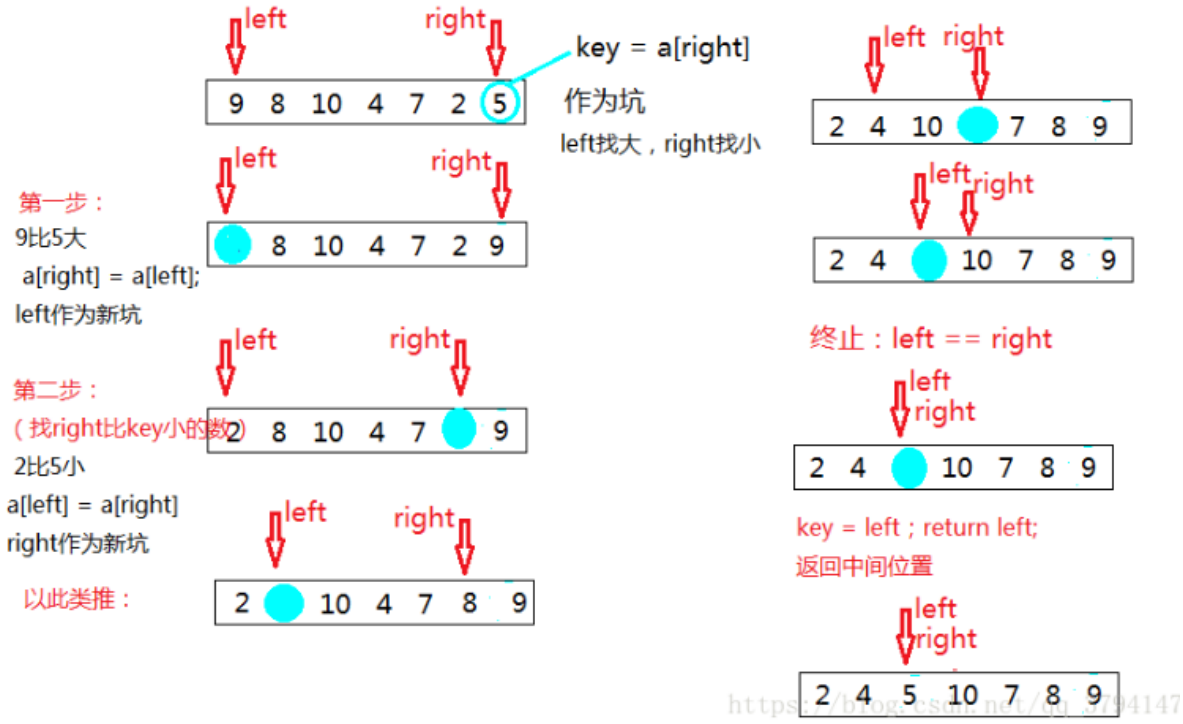
```
    int begin = left;
    int end = right;
    int key = right;

    while( begin < end )
    {
        //begin找大
        while(begin < end && a[begin] <= a[key])
        {
            ++begin;
        }
        //end找小
        while(begin < end && a[end] >= a[key])
        {
            --end;
        }
        swap(a[begin],a[end]);
    }
    swap(a[begin],a[right]);
    return begin;//返回的是中间的位置，swap只是交换值

}
```

挖坑法

基本思路：

定义两个指针left指向起始位置，right指向最后一个元素的位置,然后指定一个基数key（right）,作为坑
left寻找比基数（key）大的数字，找到后将left的数据赋给right，left成为一个坑，然后right寻找比基数（key）小的数字，找到将right的数据赋给left，right成为一个新坑，循环这个过程，直到begin指针与end指针相遇，然后将key返回给那个坑（最终：key的左边都是比key小的数，key的右边都是比key大的数），然后进行递归操

作。



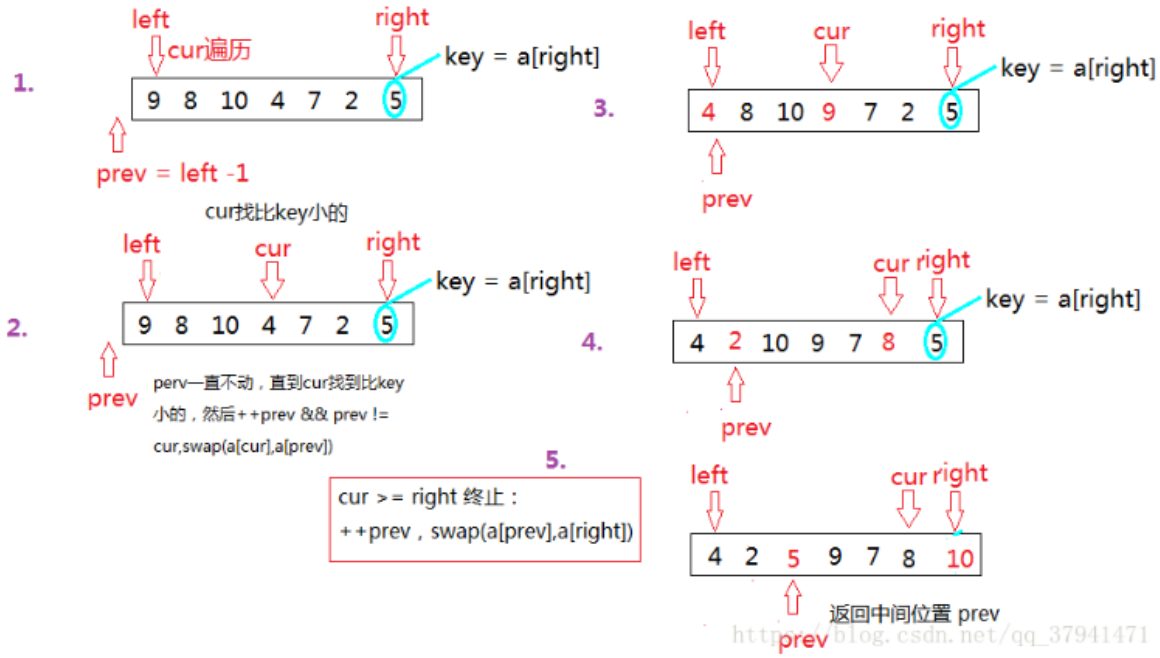
代码：

```
int PartSort2(int* a,int left,int right)
{
    int key = a[right];//初始坑
    while(left<right)
    {
        //left找大
        while(left<right && a[left] <= key )
        {
            left++;
        }
        a[right] = a[left];//赋值，然后left作为新坑
        //right找小
        while(left <right && a[right] >= key)
        {
            right--;
        }
        a[left] = a[right];//right作为新坑
    }
    a[left] = key;//将key赋值给left和right的相遇点，保持key的左边都是比key小的数，key的右边都是比key大的数
    return left;//最终返回中间位置
}
```

前后指针法

基本思路：

定义两个指针，一前一后，cur（前）指向起始位置，prev（后）指向cur的前一个位置；选择数组最后一个元素作为key（right）
实现过程：cur找比key小的数，prev在cur没有找到的情况下，一直不动（即保证prev一直指向比key大的数）；直到cur找到比key小的数（+ +prev && prev != cur时）然后++cur,prev仍然不动。
直到cur走到right前一个位置，终止循环。最后++prev，交换prev和right的值。返回中间位置prev。最后再继续递归。



代码：

```
int PartSort3(int* a,int left,int right)
{
    int cur = left;
    int prev = left -1;
    int key = a[right];
    while(cur < right)
    {
        if(a[cur] < key && ++prev != cur)
        {
            swap(a[cur],a[prev]);
        }
        ++cur;
    }
    swap(a[++prev],a[right]);
    return prev;
```

```
}
```

代码优化

1. 在有序或者接近有序的时候上面的方法效率会非常低，我们可以用 三数取中法解决。
   所谓三数取中法就是：在待排序序列的第一个元素，中间元素和最后一个元素中取大小位于中间的那个元素。

```
int GetMid(int* a,int left,int mid,int right)
{
    if(a[left] >a[mid])
    {
        if(a[mid] > a[right])
        {
            return mid;
        }
        else if(a[left] > a[right])
        {
            return right;
        }
        else
        {
            return left;
        }
    }
    else //a[left] < a[mid]
    {
        if(a[mid] < a[right])
        {
            return mid;
        }
        else if(a[left] > a[right])
        {
            return right;
        }
        else
        {
            return left;
        }
    }
}
//中间数法
int mid = left+((right-left)>>1);
mid = GetMid(a,left,mid,right);
```

```
if( mid != right )//1 2 5 7 8 3其中1 5 3,3是mid(mid == end),就不用交换
swap(a[mid],a[right]);
```

2.在递归子问题的时候在区间内的数据比较少的时候我们可以不再划分区间，直接用直接插入排序效率会更高，因为接着划分又要创建栈桢，没有必要。优化后代码：

```
int GetMid(int* a,int left,int mid,int right)
{
    if(a[left] >a[mid])
    {
        if(a[mid] > a[right])
        {
            return mid;
        }
        else if(a[left] > a[right])
        {
            return right;
        }
        else
        {
            return left;
        }
    }
    else //a[left] < a[mid]
    {
        if(a[mid] < a[right])
        {
            return mid;
        }
        else if(a[left] > a[right])
        {
            return right;
        }
        else
        {
            return left;
        }
    }
}

//1. 左右指针法
int PartSort1(int* a,int left,int right)
{
    //中间数法
```

```
    int mid = left+((right-left)>>1);
    mid = GetMid(a,left,mid,right);
    if( mid != right )//1 2 5 7 8 3其中1 5 3,3是mid(mid == end),就不用交换
        swap(a[mid],a[right]);
    int begin = left;
    int end = right;
    int key = right;

    while( begin < end )
    {
        //begin找大
        while(begin < end && a[begin] <= a[key])
        {
            ++begin;
        }
        //end找小
        while(begin < end && a[end] >= a[key])
        {
            --end;
        }
        swap(a[begin],a[end]);
    }
    swap(a[begin],a[right]);
    return begin;

}
//2.挖坑法
int PartSort2(int* a,int left,int right)
{
    int mid = left+((right-left)>>1);
    mid = GetMid(a,left,mid,right);
    if( mid != right )//1 2 5 7 8 3其中1 5 3,3是mid(mid == end),就不用交换
        swap(a[mid],a[right]);
    int key = a[right];
    while(left<right)
    {
        while(left<right && a[left] <= key )
        {
            left++;
        }
        a[right] = a[left];
        while(left <right && a[right] >= key)
        {
```

```
            right--;
        }
        a[left] = a[right];
    }
    a[left] = key;
    return left;
}

//3.前后指针法
int PartSort3(int* a,int left,int right)
{
    int mid = left+((right-left)>>1);
    mid = GetMid(a,left,mid,right);
    if( mid != right )//1 2 5 7 8 3其中1 5 3,3是mid(mid == end),就不用交换
        swap(a[mid],a[right]);
    int cur = left;
    int prev = left -1;
    int key = a[right];
    while(cur < right)
    {
        if(a[cur] < key && ++prev != cur)
        {
            swap(a[cur],a[prev]);
        }
        ++cur;
    }
    swap(a[++prev],a[right]);
    return prev;
}

void QuickSort(int* a,int left,int right)//递归
{
    assert(a);
    //终止条件
    if( left >= right )
        return;
    if((right-left +1) < 10)  //小区间时不再递归,直接用插入排序，这样会效率更高！减少栈帧
    {
        InsertSort(a+left,right-left+1);
    }
    //int div = PartSort1(a,left,right);
    //int div = PartSort2(a,left,right);
    int div = PartSort3(a,left,right);
```

```
    QuickSort(a,left,div-1);
    QuickSort(a,div+1,right);
}
```

非递归

基本思路： 利用栈保存左右区间

1. 左右区间入栈（先右后左）
2. 取栈顶元素，出栈
3. 排序
4. 入栈，先右后左（直到栈为空，停止循环）

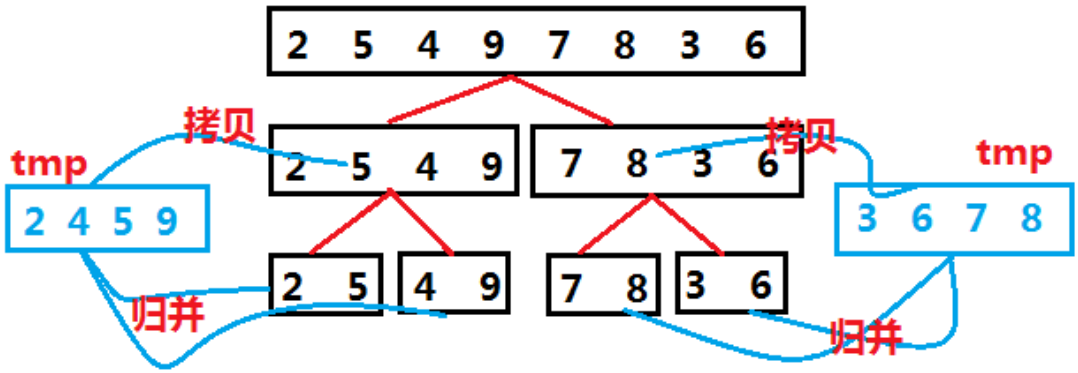
代码：

```
void QuickSortNOR(int* a,int left,int right)
{
    stack<int> s;
    if(left < right)
    {
        s.push(left);
        s.push (right);
    }
    while(!s.empty ())
    {
        int right = s.top ();
        s.pop ();
        int left = s.top ();
        s.pop();
        int div =  PartSort1(a,left,right);
        if((right-left +1) < 20)  //小区间时不再递归
        {
            InsertSort(a+left,right-left+1);
        }
        else
        {
            if(left < div-1)
            {
                s.push (left);
                s.push (div-1);
            }
            if(div+1<right)
            {
                s.push (div+1);
```

```
                s.push (right);
            }
        }
    }
}
```

归并排序

思想：分治法
每个递归过程涉及三个步骤
第一, 分解: 把待排序的 n 个元素的序列分解成两个子序列, 每个子序列包括 n/2 个元素.
第二, 治理: 对每个子序列分别调用归并排序__MergeSort, 进行递归操作
第三, 合并: 合并两个排好序的子序列,生成排序结果.



像似一颗一颗完全二叉树，其高度为lgn,而每层排序加起来需要遍历n次，所以归并排序算法的时间复杂度为O(n*lgn)
而空间复杂度为O(n)，因为需要一个tmp数组

代码实现

```
void __MergeSort( int *a, int left, int right, int * tmp  )
{
    if( left >= right ) //退出条件
        return;
    int mid = left+((right-left)>>1);
    __MergeSort(a,left,mid,tmp); // 递归左半数组
    __MergeSort(a,mid+1,right,tmp); // 递归右半数组
    //将排好序的两部分数组归并（排序）
    int begin1 = left,end1 = mid;
    int begin2 = mid+1,end2 = right;
```

```
    int index = left;
    while( begin1<=end1 && begin2<=end2 )// 循环条件：任一个数组排序完，则终止条件，最后将没有比较完的数组直接一一拷过去
    {
        if( a[begin1] <= a[begin2] )
        {
            tmp[index++] = a[begin1++];
        }
        else
        {
            tmp[index++] = a[begin2++];
        }
    }

    while( begin1 <= end1 )//右半数组走完了
    {
        tmp[index++] = a[begin1++];
    }
    while( begin2 <= end2 )//左半数组走完了
    {
        tmp[index++] = a[begin2++];
    }

    //tmp数组已经排好序,将数组内容拷到原数组，递归向上一层走
    index = left;
    while( index <= right )
    {
        a[index] = tmp[index];
        ++index;
    }

}

void MergeSort( int *a,size_t n )
{
    int *tmp = new int[n]; // 开一个第三方数组来存取左右排好序归并后的序列
    __MergeSort(a,0,n-1,tmp);
    delete[] tmp; // 最后释放第三方空间
}
```

优化：
在递归子问题的时候在区间内的数据比较少的时候我们可以不再划分区间，直接用直接插入排序效率会更高，因为接着划分又要创建栈桢，没有必要

```
void __MergeSort( int *a, int left, int right, int * tmp  )
```

```
{
    if( left >= right ) //退出条件
        return;
    if( right-left+1 <10 )//优化
    {
        InsertSort(a+left,right-left+1);
    }
    int mid = left+((right-left)>>1);
    __MergeSort(a,left,mid,tmp); // 递归左半数组
    __MergeSort(a,mid+1,right,tmp); // 递归右半数组
    //将排好序的两部分数组归并（排序）
    int begin1 = left,end1 = mid;
    int begin2 = mid+1,end2 = right;
    int index = left;

    while( begin1<=end1 && begin2<=end2 )// 循环条件：任一个数组排序完，则终止条件，最后将没有比较完的数组直接一一拷过去
    {
        if( a[begin1] <= a[begin2] )
        {
            tmp[index++] = a[begin1++];
        }
        else
        {
            tmp[index++] = a[begin2++];
        }
    }

    while( begin1 <= end1 )//右半数组走完了
    {
        tmp[index++] = a[begin1++];
    }
    while( begin2 <= end2 )//左半数组走完了
    {
        tmp[index++] = a[begin2++];
    }

    //tmp数组已经排好序,将数组内容拷到原数组，递归向上一层走
    index = left;
    while( index <= right )
    {
        a[index] = tmp[index];
        ++index;
    }
```

```
}
void MergeSort( int *a,size_t n )
{
    int *tmp = new int[n]; // 开一个第三方数组来存取左右排好序归并后的序列
    __MergeSort(a,0,n-1,tmp);
    delete[] tmp; // 最后释放第三方空间
}
```

完整代码:

```
#include <iostream>
using namespace std;

#include <assert.h>

//直接插入排序
 void InsertSort (int* a,size_t n)
 {
     assert(a);
     for(size_t i = 1;i < n; ++i)//用end的位置控制边界
     {
         //单趟排序
         int end = i - 1 ;
         int tmp = a[i];
         while( end >= 0 )//循环继续条件
         {
             if( a[end] > tmp )
             {
                 a[end+1] = a[end];
                 --end;
             }
             else
                 break;
         }
         a[end+1] = tmp;
     }
 }
void __MergeSort( int *a, int left, int right, int * tmp  )
{
    if( left >= right ) //退出条件
        return;
    if( right-left+1 <10 )//优化
    {
        InsertSort(a+left,right-left+1);
    }
```

```
    int mid = left+((right-left)>>1);
    __MergeSort(a,left,mid,tmp); // 递归左半数组
    __MergeSort(a,mid+1,right,tmp); // 递归右半数组

    //将排好序的两部分数组归并（排序）

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

    while( begin1<=end1 && begin2<=end2 )// 循环条件：任一个数组排序
完，则终止条件，最后将没有比较完的数组直接一一拷过去
    {
        if( a[begin1] <= a[begin2] )
        {
            tmp[index++] = a[begin1++];
        }
        else
        {
            tmp[index++] = a[begin2++];
        }
    }

    while( begin1 <= end1 )//右半数组走完了
    {
        tmp[index++] = a[begin1++];
    }
    while( begin2 <= end2 )//左半数组走完了
    {
        tmp[index++] = a[begin2++];
    }

    //tmp数组已经排好序,将数组内容拷到原数组，递归向上一层走
    index = left;
    while( index <= right )
    {
        a[index] = tmp[index];
        ++index;
    }

}

// 归并排序
void MergeSort( int *a,size_t n )
{
```

```
    int *tmp = new int[n]; // 开一个第三方数组来存取左右排好序归并后的序列
    __MergeSort(a,0,n-1,tmp);
    delete[] tmp; // 最后释放第三方空间
}

void Print(int a[],int len)
{
    cout<<endl;
    for(int i = 0; i < len; ++i)
    {
        cout<<a[i]<<" ";
    }
    cout<<endl;
}

void test()
 {
     //升序排序
     int a[] = {9,8,7,6,5,4,3,2,1};
    /*  int a[] = {2,5,4,0,9,3,6,8,7,1};*/
     int len = sizeof(a)/sizeof(a[0]);
     cout<<"before sort :";
     Print(a,len);

     MergeSort(a,len);
     cout<<"after sort :";
     Print(a,len);

 }
int main ()
{
    test();
    return 0;
}
```