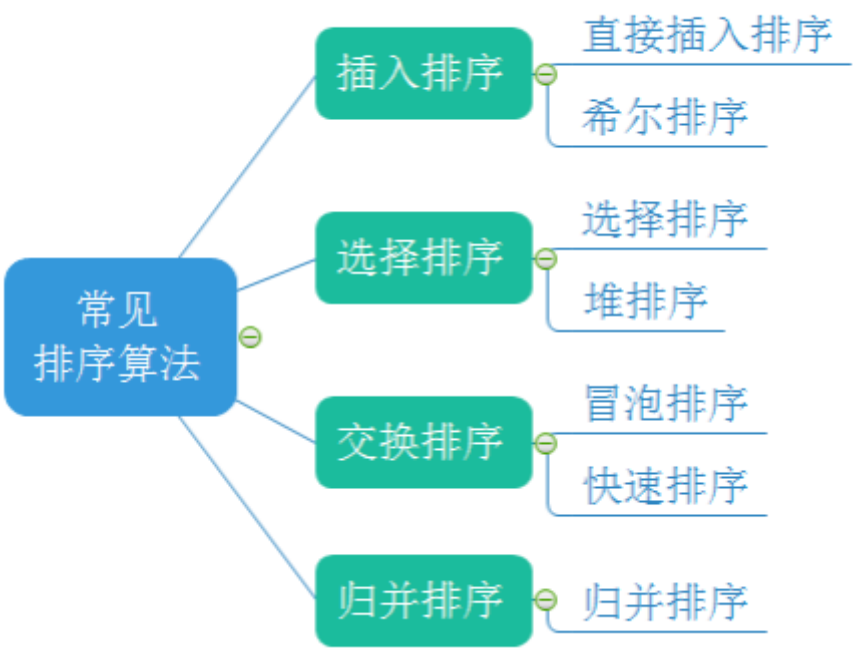
https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

1.1排序的概念

> 排序：所谓排序，就是使一串记录，按照其中的某个或某些关键字的大小，递增或递减的排列起来的操作。
> 稳定性：假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变，即在原序列中，r[i]=r[j]，且r[i]在r[j]之前，而在排序后的序列中，r[i]仍在r[j]之前，则称这种排序算法是稳定的；否则称为不稳定的。
> 内部排序：数据元素全部放在内存中的排序。
> 外部排序：数据元素太多不能同时放在内存中，根据排序过程的要求不能在内外存之间移动数据的排序。

1.2 常见的排序算法




```
// 排序实现的接口
// 插入排序
void InsertSort(int* a, int n);
// 希尔排序
void ShellSort(int* a, int n);
// 选择排序
```

```
void SelectSort(int* a, int n);
// 堆排序
void AdjustDwon(int* a, int n, int root);
void HeapSort(int* a, int n);
// 冒泡排序
void BubbleSort(int* a, int n)
// 快速排序
void QuickSort(int* a, int left, int right);
// 归并排序
void MergeSort(int* a, int n)
```

1. 插入排序

1.1基本思想:

直接插入排序是一种简单的插入排序法，其基本思想是：把待排序的记录按其关键码值的大小逐个插入到一个已经排好序的有序序列中，直到所有的记录插入完为止，得到一个新的有序序列 。实际中我们玩扑克牌时，就用了插入排序的思想

1.2直接插入排序:

当插入第i(i>=1)个元素时，前面的array[0],array[1],...,array[i-1]已经排好序，此时用array[i]的排序码与array[i-1],array[i-2],...的排序码顺序进行比较，找到插入位置即将array[i]插入，原来位置上的元素顺序后移。

```
//插入排序 -- 时间复杂度O(N^2)
void InsertSort(int* a, int n)
{
    //[0,end]有序，把[0,end+1]位置的值插入进去，让[0，end+1]也有序
    for (int i = 0; i < n - 1; i++)//注意n-1不是n
    {
        int end = i;
        int tmp = a[end + 1];
        while (end >= 0)
        {
            if (a[end] > tmp)//升序
            //if (a[end] < tmp)降序
            {
                a[end + 1] = a[end];
                --end;
```

```
            }
            else
            {
                //a[end] >= tmp
                //end<0,比所有数都要小 -- 跳出循环处理
                break;
            }
        }
        a[end + 1] = tmp;
    }
}
void PrintArray(int* a, int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("%d ", a[i]);
    }
    printf("\n");
}


void TestInsertSort()
{
    int a[] = { 3,5,2,7,8,6,1,9,4,0 };
    InsertSort(a,sizeof(a)/sizeof(int));
    PrintArray(a, sizeof(a) / sizeof(int));

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

直接插入排序的特性总结:

1. 元素集合越接近有序，直接插入排序算法的时间效率越高
2. 时间复杂度：O(N^2)
3. 空间复杂度：O(1)，它是一种稳定的排序算法
4. 稳定性：稳定

1.3 希尔排序(缩小增量排序)

希尔排序法又称缩小增量法。希尔排序法的基本思想是：先选定一个整数，把待排序文件中所有记录分成个组，所有距离为的记录分在同一组内，并对每一组内的记录进行排序（组内排序的方法是直接插入排序）。然后，取重复上述分组和排序的工作。当到达gap=1时，所有记录在统一组内排好序

> 对组间隔为gap的预排序，gap由大变小
> gap越大，大的数可以越快的到后面，小的数可以越快的到前面
> gap越大，预排完越不接近有序
> gap越小，越接近有序
> gap==1时，就是直接插入排序

gap很大时，下面预排序时间复杂度O(N)
gap很小时，数组已经很接近有序了，这时差不多也是O(N)
因此整体的时间复杂度大概是：O(N*logN)

```
void ShellSort(int *a,int n)
{
    int gap = n;//就相当于插入排序，只是把1替换成了gap
    while (gap > 1)
    {
        //gap没有固定的取值，一般是除三或除二，但一定要保证最后一次gap是1
        gap = gap / 3 + 1;
        //gap /= 2;
        //把间隔为gap的多组数据同时排
        for (int i = 0; i < n - gap; i++)
        {
            int end = i;
            int tmp = a[end + gap];
            while (end >= 0)
            {
                if (a[end] > tmp)
                {
                    a[end + gap] = a[end];
                    end -= gap;
                }
                //end<0,比所有数都要小 -- 跳出循环处理
                else
                {
                    break;
```

```
                }
            }
            a[end + gap] = tmp;
        }
    }

}
```

希尔排序的特性总结：

1. 希尔排序是对直接插入排序的优化。
2. 当gap > 1时都是预排序，目的是让数组更接近于有序。当gap == 1时，数组已经接近有序的了，这样就会很快。这样整体而言，可以达到优化的效果。我们实现后可以进行性能测试的对比。
3. 希尔排序的时间复杂度不好计算，需要进行推导，推导出来平均时间复杂度： O(N^1.3—N^2)
4. 稳定性：不稳定

2.选择排序

2.1基本思想:

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完 。

2.2 直接选择排序:

- 在元素集合array[i]--array[n-1]中选择关键码最大(小)的数据元素
- 若它不是这组元素中的最后一个(第一个)元素，则将它与这组元素中的最后一个（第一个）元素交换
- 在剩余的array[i]--array[n-2]（array[i+1]--array[n-1]）集合中，重复上述步骤，直到集合剩余1个元素

```
//最简单的排序也是效率最差的排序
void SelectSort(int* a, int n)
{
    int begin = 0;
    int end = n - 1;
    while (begin < end)
    {
        int min = begin;
        int max = begin;
        for (int i = begin; i <= end; i++)
```

```
        {
            if (a[i] < a[min])
                min = i;
            if (a[i] > a[max])
                max = i;
        }
        //先找到一组最大值最小值，一直到循环结束
        Swap(&a[begin], &a[min]);
        //{ 9,3,5,2,7,8,6,-1,1,9,4,0 }这种情况begin和max重合，要对max进行修正
        //{ -1,3,5,2,7,8,6,9,1,9,4,0 }
        if (begin == max)
        {
            max = min;
        }
        Swap(&a[max], &a[end]);
        begin++;
        end--;
    }

}
```

直接选择排序的特性总结：

1. 直接选择排序思考非常好理解，但是效率不是很好。实际中很少使用
2. 时间复杂度：O(N^2)
3. 空间复杂度：O(1)
4. 稳定性：不稳定

2.3 堆排序

下面图片有误，应是：

```
leftchild = parents*2+1
rightchile=parents*2+2
parents=(child-1)/2
```

堆排序(Heapsort)是指利用堆积树（堆）这种数据结构所设计的一种排序算法，它是选择排序的一种。它是通过堆来进行选择数据。**需要注意的是排升序要建大堆，排降序建小堆。**

（向下调整算法）筛选法：左右子树都是小堆（大堆）

> 从根节点开始，选出左右孩子中小的那一个，跟父亲相比，如果比父亲小就交换，然后再继续往下调，调到叶子节点就终止。

如果左右子树不是小堆，就不能直接使用向下调整算法了！

> 倒着走，叶子结点不需要调，从倒数最后一个非叶子的子树开始调

排升序，建大堆还是小堆？建大堆

> 如果是建小堆，最小数在堆顶，已经被选出来了。那么在剩下的数中再去选树，但是剩下的树的结构都乱了，需要重新建堆才能选出下一个数，建堆的时间复杂度是O(N)那么这样不是不可以但是堆排序就没有效率优势了。
> 第一个和最后一个交换，把它不看作堆里面。前n-1一个数向下调整(O(logN))选出次大的数，再跟倒数第2个位置交换...时间复杂度：O(N*logN)

```c
void Swap(int* p1, int* p2)
{
    int tmp = *p1;
    *p1 = *p2;
    *p2 = tmp;
}

//向下调整算法
void AdjustDown(int* a, int n,int root)
{
    int parent = root;
    int child = parent * 2 + 1;//默认是左孩子
    while (child < n)
    {
        //选出左右孩子中大的那一个 -- 小根堆
        //if (child + 1 < n && a[child + 1] < a[child])//只有左孩子或右孩子<左孩子
        if (child + 1 < n && a[child + 1] > a[child])//大根堆
        {
            child += 1;
        }
        if (a[child] > a[parent])//大根堆
        //if (a[child] < a[parent])//小根堆
        {
            Swap(&a[child], &a[parent]);
            parent = child;
            child = parent * 2 + 1;
        }
```

```
        else
        {
            break;
        }
    }
}


void HeapSort(int* a, int n)
{
    //建堆 -- 时间复杂度O(N)
    for (int i = (n - 1 - 1) / 2; i >= 0; i--)//找到父亲结点
    {
        //从最后一个非叶子的子树开始建堆
        AdjustDown(a, n, i);
    }

    // 排升序，建大堆还是小堆？建大堆
    int end = n - 1;
    while (end > 0)
    {
        Swap(&a[0], &a[end]);//把最大的数换到最后一个位置
        AdjustDown(a, end, 0);//剩下的数看成一个堆向下调整
        --end;
    }
}
```

堆排序的特性总结：

1. 堆排序使用堆来选数，效率就高了很多。
2. 时间复杂度：O(N*logN)
3. 空间复杂度：O(1)
4. 稳定性：不稳定

3. 交换排序

基本思想：所谓交换，就是根据序列中两个记录键值的比较结果来对换这两个记录在序列中的位置，交换排序的特点是：将键值较大的记录向序列的尾部移动，键值较小的记录向序列的前部移动。

3.1冒泡排序

```c
void BubbleSort(int* a, int n)
{
    for (int j = 0; j < n; j++)
    {
        int exchange = 0;
        for (int i = 1; i < n - j; i++)
        {
            if (a[i - 1] > a[i])
            {
                Swap(&a[i - 1], &a[i]);
                exchange = 1;

            }
        }
        if (exchange = 0)//没有发生交换
        {
            break;
        }
    }

}
```

冒泡排序的特性总结：

1. 冒泡排序是一种非常容易理解的排序
2. 时间复杂度：O(N^2)
3. 空间复杂度：O(1)
4. 稳定性：稳定

3.2 快速排序

快速排序是Hoare于1962年提出的一种二叉树结构的交换排序方法，其基本思想为：任取待排序元素序列中的某元素作为基准值，按照该排序码将待排序集合分割成两子序列，左子序列中所有元素均小于基准值，右子序列中所有元素均大于基准值，然后最左右子序列重复该过程，直到所有元素都排列在相应位置上为止。

1.挖坑法

```c
//挖坑法
void QuickSort(int* a, int left, int right)//快速排序
```

```
{
    //当划分的区间不存在或区间内只有一个数时说明有序，退出循环
    if (left >= right)
    {
        return;
    }
    int begin = left, end = right;
    //int index = GetMidIndex(a, left, right);//三数取中算法
    //Swap(&a[left], &a[index]);
    int pivot = begin;
    int key = a[begin];
    //单趟排序
    while (begin < end )
    {
        //右边找小，放到左边
        while (begin < end && a[end] >= key)
        //还要再一次判断 begin < end
        {
            end--;
        }
        //小的放到左边的坑里，自己形成新的坑位
        a[pivot] = a[end];
        pivot = end;
        //左边找大，放到右边
        while (begin < end && a[begin]<=key)
        {
            begin++;
        }
        //大的放到右边的坑里，自己形成新的坑位
        a[pivot] = a[begin];
        pivot = begin;
    }
    pivot = begin;
    a[pivot] = key;

    //把[left,right]分成以下区间
    //[left,pivot-1]  pivot  [pivot+1,right]
    //左子区间和右子区间有序，就有序 -- 分治递归
    QuickSort(a, left, pivot - 1);
    QuickSort(a, pivot + 1, right);
```

```
43 }
```

快速排序的特性总结：

1. 快速排序整体的综合性能和使用场景都是比较好的，所以才敢叫快速排序
2. 时间复杂度：O(N*logN)
3. 空间复杂度：O(logN)
4. 稳定性：不稳定

优化1：**三数取中**

什么情况下最坏：

最差的情况就是每一次取到的元素就是数组中最小/最大的，这种情况其实就是冒泡排序了(每一次都排好一个元素的顺序)这种情况时间复杂度就好计算了，就是冒泡排序的时间复杂度：T[n] = n * (n-1) = n^2 + n;

综上所述：快速排序最差的情况下时间复杂度为：O(n^2)

三数取中解决最差情况问题：选中间值作为key

```
int GetMidIndex(int* a, int left, int right)
{
    int mid = (left + right) >> 1;//右移一位相当于除2，(left + right) / 2;
    if (a[left] < a[mid])//顺序
    {
        if (a[mid] < a[right])
            return mid;
        else if(a[left]>a[right])
            return left;
        else
            return right;
    }
    else//a[left] > a[mid] -- 逆序
    {
        if (a[mid] > a[right])
            return mid;
        else if (a[left] < a[right])

            return left;
        else

            return right;

    }
```

```
}
```

优化2：小区间优化

> 当划分的子序列很小的时候(一般认为小于13个元素左右时)，我们在使用快速排序对这些小序列排序反而不如直接插入排序高效。因为快速排序对数组进行划分最后就像一颗二叉树一样，当序列小于13个元素时我们再使用快排的话就相当于增加了二叉树的最后几层的结点数目，增加了递归的次数。所以我们在当子序列小于13个元素的时候就改用直接插入排序来对这些子序列进行排序。

```
//把[left,right]分成以下区间
//[left,pivot-1]  pivot  [pivot+1,right]
//当子序列小于13个元素的时候就改用直接插入排序来对这些子序列进行排序
if (pivot - 1 - left > 10)
    QuickSort_Mid_Minizone(a, left, pivot - 1);
else
    InsertSort(a + left, pivot - 1 - left + 1);
if (right - (pivot + 1) > 10)
    QuickSort_Mid_Minizone(a, pivot + 1, right);
else
    InsertSort(a + pivot + 1, right - (pivot + 1) + 1);
```

2.左右指针法

```
//左右指针法
void QuickSort2(int* a, int left, int right)//快速排序
{
    //当划分的区间不存在或区间内只有一个数时说明有序，退出循环
    if (left >= right)
    {
        return;
    }
    int begin = left, end = right;
```

```
    int key = a[begin];

    while (begin < end)
    {
        //找小
        while (begin < end && a[end] >= a[key])
        {
            end--;
        }
        //找大
        while (begin < end && a[begin] <= a[key])
        {
            begin++;
        }
        Swap(&a[begin], &a[end]);//大小交换
    }
    Swap(&a[begin], &a[key]);

    //把[left,right]分成以下区间
    //[left,begin-1]  begin  [begin+1,right]
    //左子区间和右子区间有序，就有序 -- 分治递归
    QuickSort2(a, left, begin - 1);
    QuickSort2(a, begin + 1, right);
}
```

3.前后指针法

```
void QuickSort3(int* a, int left, int right)//前后指针法
{
    if (left >= right)
        return;
    int key = left;
    int pre = left, cur = left + 1;
    while (cur <= right)
    {
        if (a[cur] < a[key])
        {
```

```
            ++pre;
            Swap(&a[pre], &a[cur]);
        }
        ++cur;
    }
    Swap(&a[key], &a[pre]);
    //把[left,right]分成以下区间
    //[left,begin-1]  begin  [begin+1,right]
    //左子区间和右子区间有序，就有序 -- 分治递归
    QuickSort3(a, left, pre - 1);
    QuickSort3(a, pre + 1, right);
}
```

3.3.归并排序

基本思想:

> 归并排序（MERGE-SORT）是建立在归并操作上的一种有效的排序算法,该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。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，再使子序列段间有序。若将两个有序表合并成一个有序表，称为二路归并。 归并排序核心步骤

```
void _MergeSort(int* a, int left, int right,int* tmp)//归并排序子函数
{
    if (left >= right)
        return;//区间不存在
    int mid = (left + right) >> 1;//右移一位相当于除2
    //假设[left,mid][mid+1,right]有序，那么我们就可以归并
    //无序就递归
    _MergeSort(a, left, mid, tmp);
    _MergeSort(a, mid+1, right, tmp);

    //归并
    int begin1 = left, end1 = mid;
    int begin2 = mid + 1, end2 = right;
    int index = left;
    while (begin1 <= end1 && begin2 <= end2)
    {
```

```
        if (a[begin1] < a[begin2])
        {
            tmp[index++] = a[begin1++];
        }
        else
        {
            tmp[index++] = a[begin2++];
        }
    }
    while (begin1 <= end1)//begin1没结束把begin1拷贝过来
    {
        tmp[index++] = a[begin1++];
    }
    while (begin2 <= end2)//begin2没结束把begin2拷贝过来
    {
        tmp[index++] = a[begin2++];
    }
    //把临时数组中的数拷贝回去
    for (int i = left; i <= right; i++)
    {
    a[i] = tmp[i];
    }
}

void MergeSort(int* a, int n)//归并排序
{
    int* tmp = (int*)malloc(sizeof(int) * n);//空间复杂度O(N)
    _MergeSort(a, 0, n - 1,tmp);
    free(tmp);
}
```

3.4.递归的缺陷

递归的缺陷：栈帧深度太深，栈空间不够用，可能会溢出

递归改非递归：

1. 直接改循环（简单）
2. 借助数据结构栈模拟递归过程（复杂）

快速排序的非递归算法：

```c
//栈里面的区间要分割排序
void QuickSortNonR(int* a, int n)
{
    ST st;
    StackInit(&st);
    //先出左，就要先入右
    StackPush(&st, n - 1);
    StackPush(&st, 0);
    while (!StackEmpty(&st))
    {
        int left = StackTop(&st);
        StackPop(&st);
        int right = StackTop(&st);
        StackPop(&st);
        int keyIndex = PartSort_Trenching(a, left, right);//挖坑法
        //int keyIndex = PartSort_RLHands(a, left, right);//左右指针
        //int keyIndex = PartSort_FBHands(a, left, right);//前后指针

        //[left,keyIndex - 1],ketIndex,[ketIndex+1,right]
        if (keyIndex + 1 < right)//说明还有多个值，还没有序
        {
            //先入右，再入左
            StackPush(&st,right);
            StackPush(&st, keyIndex + 1);
        }
        if (left< keyIndex - 1)//说明还有多个值，还没有序
        {
            StackPush(&st, keyIndex - 1);
            StackPush(&st, left);
        }
    }

    STackDestory(&st);

}
```

归并排序的非递归算法：

归并排序（下）

因此我们需要对区间进行修正：

```
// end1 越界，修正
if (end1 >= n)
    end1 = n - 1;

// begin2 越界，第二个区间不存在
if (begin2 >= n)
{
    begin2 = n;
    end2 = n - 1;
}

// begin2 ok， end2越界，修正end2即可
if (begin2 < n && end2 >= n)
    end2 = n - 1;
```

修正end1：
当end1越界时，让end1等于最后一个元素下标即可。 end1 = n-1
修正begin2：
如果begin2越界，就说明第二个区间并不存在，我们都不需要合并了，因为区间2不存在，区间1就是我们最终的数组，因此我们就让区间2搞成一个不存在 begin = n,end2 = n-1.
修正end2：
当begin2ok,end2越界时,我们修正end2等于最后一个元素下标即可。 end2 = n-1

```
void MergeSortNonR(int* a, int n)
{
    int* tmp = (int*)malloc(sizeof(int) * n);
    int gap = 1;//每组数据的个数
    while (gap < n)
    {
        for (int i = 0; i < n; i += 2 * gap)
        {
            //[i,i+gap-1][i+gap,i+2*gap-1]
            //归并
            int begin1 = i, end1 = i + gap - 1;
            int begin2 = i + gap, end2 = i + 2 * gap - 1;
            // end1 越界，修正
```

```
            if (end1 >= n)
                end1 = n - 1;
            //归并过程右半区间可能不存在
            if (begin2 >= n)
            {
                begin2 = n;
                end2 = n - 1;
            }
            // 归并过程右半区间算多了，begin2 ok， end2越界，修正end2即可
            if (begin2 < n && end2 >= n)
                end2 = n - 1;

            int index = i;
            while (begin1 <= end1 && begin2 <= end2)
            {
                if (a[begin1] < a[begin2])
                    tmp[index++] = a[begin1++];
                else
                    tmp[index++] = a[begin2++];
            }
            while (begin1 <= end1)//begin1没结束把begin1拷贝过来
                tmp[index++] = a[begin1++];
            while (begin2 <= end2)//begin2没结束把begin2拷贝过来
                tmp[index++] = a[begin2++];
        }
        //把临时数组中的数拷贝回去
        for (int j = 0; j < n; j++)
        {
            a[j] = tmp[j];
        }
        gap *= 2;
    }
    free(tmp);
}
```