

前提

2023年8月11日 11:22

前 提

```
void X_Sort ( ElementType A[], int N )
```

- 大多数情况下，为简单起见，讨论从小大的整数排序
- N是正整数
- 只讨论基于比较的排序（> = < 有定义）
- 只讨论内部排序
- 稳定性：任意两个相等的数据，排序前后的相对位置不发生改变

屏幕剪辑的捕获时间: 2023/8/11 11:27

没有一种排序是在任意情况下都最好的

冒泡排序

2023年8月11日 11:27

简单排序

■ 冒泡排序



```
void Bubble_Sort( ElementType A[], int N )
{
    for ( P=N-1; P>=0; P-- ){
        flag = 0;
        for( i=0; i<P; i++ ) { /* 一趟冒泡 */
            if ( A[i] > A[i+1] ) {
                Swap(A[i], A[i+1]);
                flag = 1; /* 标识发生了交换 */
            }
        }
        if ( flag==0 ) break; /* 全程无交换 */
    }
}
```

最好情况：顺序 $T = O(N)$

最坏情况：逆序 $T = O(N^2)$

屏幕剪辑的捕获时间: 2023/8/11 11:32

冒泡排序的好处：

1. 对于单向链表，冒泡排序仍然适用（因为是从头走到尾）
2. 由于是判定前者顺序严格大于后者才交换，所以冒泡排序有稳定性

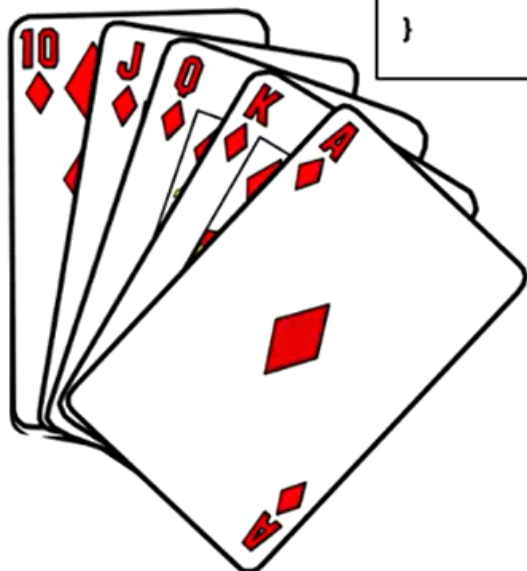
插入排序

2023年8月11日 11:36

就类似于打牌整理手牌的过程，每插入一张牌，从最后一张牌一直往前比，只要比到的这张牌比需要插入的牌小，就把牌插入这个位置

简单排序

■ 插入排序



```
void Insertion_Sort( ElementType A[], int N )
{
    for ( P=1; P<N; P++ ) {
        Tmp = A[P]; /* 摸下一张牌 */
        for ( i=P; i>0 && A[i-1]>Tmp; i-- )
            A[i] = A[i-1]; /* 移出空位 */
        A[i] = Tmp; /* 新牌落位 */
    }
}
```

稳定

最好情况：顺序 $T = O(N)$

最坏情况：逆序 $T = O(N^2)$

屏幕剪辑的捕获时间: 2023/8/11 11:39

时间复杂度下界

- 定理：任意 N 个不同元素组成的序列平均具有 $N(N-1)/4$ 个逆序对。
- 定理：任何仅以交换相邻两元素来排序的算法，其平均时间复杂度为 $\Omega(N^2)$ 。

其平均时间复杂度为 $\Omega(N^2)$ 。

屏幕剪辑的捕获时间: 2023/8/11 14:17

希尔排序 (Shell_Sort)

2023年8月11日 14:24



举个例子

	81	94	11	96	12	35	17	95	28	58	41	75	15
5-间隔	35	17	11	28	12	41	75	15	96	58	81	94	95
3-间隔	28	12	11	35	15	41	58	17	94	75	81	96	95
1-间隔	11	12	15	17	28	35	41	58	75	81	94	95	96

- 定义增量序列 $D_M > D_{M-1} > \dots > D_1 = 1$
- 对每个 D_k 进行 “ D_k -间隔” 排序 ($k = M, M-1, \dots, 1$)
- 注意: “ D_k -间隔” 有序的序列, 在执行 “ D_{k-1} -间隔” 排序后, 仍然是 “ D_k -间隔” 有序的

屏幕剪辑的捕获时间: 2023/8/11 14:33

希尔增量序列

■ 原始希尔排序 $D_M = \lfloor N/2 \rfloor, D_k = \lfloor D_{k+1}/2 \rfloor$

```
void Shell_sort( ElementType A[], int N )
{   for ( D=N/2; D>0; D/=2 ) { /* 希尔增量序列 */
    for ( P=D; P<N; P++ ) { /* 插入排序 */
        Tmp = A[P];
        for ( i=P; i>=D && A[i-D]>Tmp; i-=D )
            A[i] = A[i-D];
        A[i] = Tmp;
    }
}
}
```

屏幕剪辑的捕获时间: 2023/8/11 14:45

这样取序列最坏情况下还是会达到 $O(N^2)$ ，坏的例子如下：

	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
8-间隔	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
4-间隔	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
2-间隔	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
1-间隔	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

屏幕剪辑的捕获时间: 2023/8/11 14:46

增量序列不互质，那么小序列的排序可能根本不起作用

其他序列：Hibbard增量序列、Sedgewick增量序列

■ Hibbard 增量序列

- $D_k = 2^k - 1$ — 相邻元素互质
- 最坏情况: $T = \Theta(N^{3/2})$
- 猜想: $T_{avg} = O(N^{5/4})$

■ Sedgewick 增量序列

- $\{1, 5, 19, 41, 109, \dots\}$
— $9 \times 4^i - 9 \times 2^i + 1$ 或 $4^i - 3 \times 2^i + 1$
- 猜想: $T_{avg} = O(N^{7/6})$, $T_{worst} = O(N^{4/3})$

屏幕剪辑的捕获时间: 2023/8/11 14:47

堆排序

2023年8月11日 15:01

1. 建最小堆，然后一个个删除堆中元素，存到临时数组中，最后复制回去。建堆的 $O(N)$ 方法见[这里](#)。这种排序方法的缺点在于还要开辟一个 $O(N)$ 的空间，很浪费；

■ 算法1

```
void Heap_Sort ( ElementType A[], int N )
{
    BuildHeap(A); /*  $O(N)$  */
    for ( i=0; i<N; i++ )
        TmpA[i] = DeleteMin(A); /*  $O(\log N)$  */
    for ( i=0; i<N; i++ ) /*  $O(N)$  */
        A[i] = TmpA[i];
}
```

$$T(N) = O(N \log N)$$

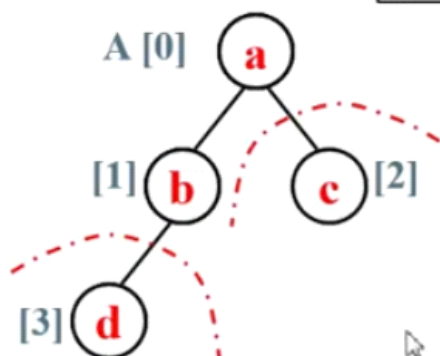
屏幕剪辑的捕获时间: 2023/8/11 15:27

2. 建立最大堆，每次删除最大元素并将其放在堆底，同时堆size-1，调整为堆，重复这个过程
注意到下图中 $i=N/2$ 为初始条件，序号N的父节点为 $N/2$ ，或者其父节点的前一个结点，合理。

堆排序

■ 算法2

```
void Heap_Sort ( ElementType A[], int N )
{
    for ( i=N/2; i>=0; i-- ) /* BuildHeap */
        PercDown( A, i, N );
    for ( i=N-1; i>0; i-- ) {
        Swap( &A[0], &A[i] ); /* DeleteMax */
        PercDown( A, 0, i );
    }
}
```



- 定理：堆排序处理 N 个不同元素的随机排列的平均比较次数是 $2N \log N - O(N \log \log N)$ 。

- 虽然堆排序给出最佳平均时间复杂度，但实际效果不如用

快速排序，快速排序的复杂度为 $O(N^2)$ ，堆排序的复杂度为 $O(N \log N)$ 。



杂度，但实际效果不如用
Sedgewick增量序列的希尔排序。

屏幕剪辑的捕获时间: 2023/8/11 15:50

归并排序

2023年8月11日 16:18

1. 分而治之，递归地解决两半边的排序，最后合并两个子列，合并子列的时间复杂度为 $O(N)$ ，这种排序的时间复杂度严格为 $O(N\log N)$ ，而且是稳定的

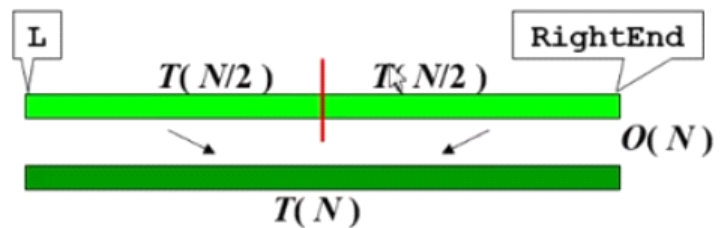
核心：有序子列的归并

```
/* L = 左边起始位置, R = 右边起始位置, RightEnd = 右边终点位置 */
void Merge( ElementType A[], ElementType TmpA[],
            int L, int R, int RightEnd )
{
    LeftEnd = R - 1; /* 左边终点位置。假设左右两列挨着 */
    Tmp = L; /* 存放结果的数组的初始位置 */
    NumElements = RightEnd - L + 1;
    while( L <= LeftEnd && R <= RightEnd ) {
        if ( A[L] <= A[R] ) TmpA[Tmp++] = A[L++];
        else                TmpA[Tmp++] = A[R++];
    }
    while( L <= LeftEnd ) /* 直接复制左边剩下的 */
        TmpA[Tmp++] = A[L++];
    while( R <= RightEnd ) /* 直接复制右边剩下的 */
        TmpA[Tmp++] = A[R++];
    for( i = 0; i < NumElements; i++, RightEnd -- )
        A[RightEnd] = TmpA[RightEnd];
}
```

屏幕剪辑的捕获时间: 2023/8/11 16:33

递归算法

■ 分而治之



```
void MSort( ElementType A[], ElementType TmpA[],
            int L, int RightEnd )
{
    int Center;
    if ( L < RightEnd ) {
        Center = ( L + RightEnd ) / 2;
        MSort( A, TmpA, L, Center );
        MSort( A, TmpA, Center+1, RightEnd );
        Merge( A, TmpA, L, Center+1, RightEnd );
    }
}
```

但是上图中的函数接口比较不友好，需要统一的函数接口：

■ 统一函数接口

```
void Merge_sort( ElementType A[], int N )
{
    ElementType *TmpA;
    TmpA = malloc( N * sizeof( ElementType ) );
    if ( TmpA != NULL ) {
        MSort( A, TmpA, 0, N-1 );
        free( TmpA );
    }
    else Error( "空间不足" );
}
```

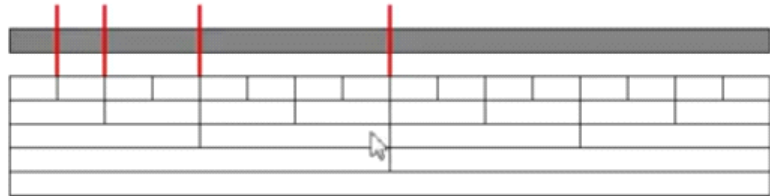
注意到TmpA数组是在Merge_Sort里面声明的，这样递归调用期间，所有操作都在A与TmpA中发生，完成后只free一个TmpA就可以了。如果嫌声明麻烦，在Merge函数中一次次malloc、free空间操作，虽然在正常free所有申请的内存的情况下空间复杂度还是 $O(N)$ ，但是很不合算

■ 如果只在Merge中声明临时数组

- `void Merge(ElementType A[], int L, int R, int RightEnd)`
- `void MSort(ElementType A[], int L, int RightEnd)`

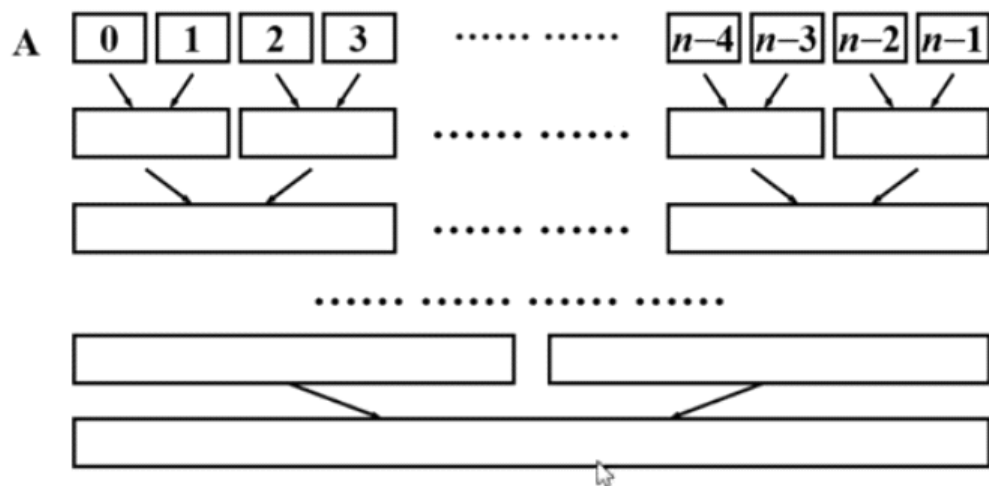
■ 如果只在Merge中声明临时数组

- `void Merge(ElementType A[], int L, int R, int RightEnd)`
- `void MSort(ElementType A[], int L, int RightEnd)`



2. 非递归算法:

非递归算法



额外空间复杂度是??? $O(N)$

具体实现就是将A归并到TmpA中后, $length*2$, 再将TmpA归并到A中, 如此反复, 只不过最终结果可能存在于TmpA中, 这时直接复制TmpA到A里即可

Merge函数与前面的递归归并不同, 最后的尾巴不好处理, 所以只处理到倒数第二段子序列, 然后分类处理: ①如果结尾有两个子列, 那再归并一次; ②如果只有一个子列, 那直接把剩下的A导入到TmpA中

非递归算法

将A中元素归并到TmpA

```
void Merge_pass( ElementType A[], ElementType TmpA[], int N,
                int length ) /* length = 当前有序子列的长度 */
{
    for ( i=0; i <= N-2*length; i += 2*length )
        Merge1( A, TmpA, i, i+length, i+2*length-1 );
    if ( i+length < N ) /* 归并最后2个子列 */
        Merge1( A, TmpA, i, i+length, N-1 );
    else /* 最后只剩1个子列 */
        for ( j = i; j < N; j++ ) TmpA[j] = A[j];
}
```

屏幕剪辑的捕获时间: 2023/8/11 19:44

注意在Merge过程中，由于判断 $i \leq N-2*length$ 的存在，并不会导致数组越界，比如 $length > N/2$ 的时候，这个循环根本进不去，直接判断下一个if了，不要怕

这样对于一个给定的length，相应的merge方法就写好了

接下来实现排序，设定length从1开始，将A归并到TmpA中，然后 $length*2$ ，然后把TmpA归并到A中，

在循环体内进行这两步，退出循环的条件就设定为 $length < N$ ，这样可以保证退出循环时是TmpA归并到A后的结果，无需进行TmpA到A的复制。

Merge_Sort是稳定的，时间复杂度也是 $O(N \log N)$ ，啥都好，就是空间复杂度太高，所以实际应用时如果在内存中可以完成排序，不会用归并排序，一般用于外排序。

非递归算法

```
void Merge_sort( ElementType A[], int N )
{   int length = 1; /* 初始化子序列长度 */
    ElementType *TmpA;
    TmpA = malloc( N * sizeof( ElementType ) );
    if ( TmpA != NULL ) {
        while( length < N ) {
            Merge_pass( A, TmpA, N, length );
            length *= 2;
            Merge_pass( TmpA, A, N, length );
            length *= 2;
        }
        free( TmpA );
    }
    else Error( "空间不足" );
}
```

稳定

屏幕剪辑的捕获时间: 2023/8/11 19:48

Ex: 不同算法的运行情况

2023年8月12日 21:52

给定N个（长整型范围内的）整数，要求输出从小到大排序后的结果。

本题旨在测试各种不同的排序算法在各种数据情况下的表现。各组测试数据特点如下：

- 数据1：只有1个元素；
- 数据2：11个不相同的整数，测试基本正确性；
- 数据3： 10^3 个随机整数；
- 数据4： 10^4 个随机整数；
- 数据5： 10^5 个随机整数；
- 数据6： 10^5 个顺序整数；
- 数据7： 10^5 个逆序整数；
- 数据8： 10^5 个基本有序的整数；
- 数据9： 10^5 个随机正整数，每个数字不超过1000。

输入格式：

输入第一行给出正整数N（ ≤ 105 ），随后一行给出N个（长整型范围内的）整数，其间以空格分隔。

输出格式：

在一行中输出从小到大排序后的结果，数字间以1个空格分隔，行末不得有多余空格。

输入样例：

```
11
4 981 10 -17 0 -20 29 50 8 43 -5
```

输出样例：

```
-20 -17 -5 0 4 8 10 29 43 50 981
```

来自 <<https://pintia.cn/problem-sets/1667128414987735040/exam/problems/1667128415088398341?type=7&page=0>>

题目	用户	提交时间
09-排序1	FSReed	2023/08/11 20:12:41
编译器	内存	用时
C++ (g++)	1484 / 65536 KB	10000 / 10000 ms
状态 ②	分数	评测时间
部分正确	21 / 25	2023/08/11 20:12:42

评测详情					
测试点	提示	内存(KB)	用时(ms)	结果	得分
0		452	3	答案正确	1 / 1
1		444	4	答案正确	10 / 10
2		592	4	答案正确	2 / 2
3		576	129	答案正确	2 / 2
4		944	10000	运行超时	0 / 2

3	576	129	答案正确	2 / 2
4	944	10000	运行超时	0 / 2
5	1484	17	答案正确	2 / 2
6	1392	7136	答案正确	2 / 2
7	1468	268	答案正确	2 / 2
8	896	10000	运行超时	0 / 2

题目	用户	提交时间
09-排序1	FSReed	2023/08/11 20:21:18
编译器	内存	用时
C++ (g++)	1464 / 65536 KB	3392 / 10000 ms
状态 ?	分数	评测时间
答案正确	25 / 25	2023/08/11 20:21:18

评测详情

Insert

测试点	提示	内存(KB)	用时(ms)	结果	得分
0		440	3	答案正确	1 / 1
1		448	4	答案正确	10 / 10
2		452	4	答案正确	2 / 2
3		592	23	答案正确	2 / 2
4		1400	1699	答案正确	2 / 2
5		1464	18	答案正确	2 / 2
6		1404	3392	答案正确	2 / 2
7		1440	33	答案正确	2 / 2
8		1252	1697	答案正确	2 / 2

题目	用户	提交时间
09-排序1	FSReed	2023/08/12 21:46:18
编译器	内存	用时
C++ (g++)	1480 / 65536 KB	201 / 10000 ms
状态 ?	分数	评测时间
答案正确	25 / 25	2023/08/12 21:46:19

评测详情						
测试点	提示	内存(KB)	用时(ms)	结果	得分	
0		440	3	答案正确	1 / 1	
1		452	3	答案正确	10 / 10	
2		444	3	答案正确	2 / 2	
3		572	6	答案正确	2 / 2	
4		1340	113	答案正确	2 / 2	
5		1480	18	答案正确	2 / 2	
6		1464	201	答案正确	2 / 2	
7		1476	19	答案正确	2 / 2	
8		1200	109	答案正确	2 / 2	

题目	用户	提交时间
09-排序1	FSReed	2023/08/12 22:17:10
编译器	内存	用时
C++ (g++)	1464 / 65536 KB	32 / 10000 ms
状态 ?	分数	评测时间
答案正确	25 / 25	2023/08/12 22:17:11

评测详情						
测试点	提示	内存(KB)	用时(ms)	结果	得分	
0		444	3	答案正确	1 / 1	
1		600	4	答案正确	10 / 10	
2		436	3	答案正确	2 / 2	
3		564	6	答案正确	2 / 2	
4		1460	32	答案正确	2 / 2	
5		1464	22	答案正确	2 / 2	
6		1460	22	答案正确	2 / 2	
7		1464	22	答案正确	2 / 2	
8		1084	27	答案正确	2 / 2	

题目	用户	提交时间
09-排序1	FSReed	2023/08/12 22:51:38
编译器	内存	用时
C++ (g++)	1612 / 65536 KB	27 / 10000 ms
状态	分数	评测时间
答案正确	25 / 25	2023/08/12 22:51:38

merge

评测详情					
测试点	提示	内存(KB)	用时(ms)	结果	得分
0		448	3	答案正确	1 / 1
1		440	3	答案正确	10 / 10
2		452	3	答案正确	2 / 2
3		584	5	答案正确	2 / 2
4		1476	27	答案正确	2 / 2
5		1612	19	答案正确	2 / 2
6		1464	20	答案正确	2 / 2
7		1456	22	答案正确	2 / 2
8		1368	25	答案正确	2 / 2

题目	用户	提交时间
09-排序1	FSReed	2023/08/12 23:55:43
编译器	内存	用时
C++ (clang++)	2004 / 65536 KB	31 / 10000 ms
状态 ②	分数	评测时间
答案正确	25 / 25	2023/08/12 23:55:44

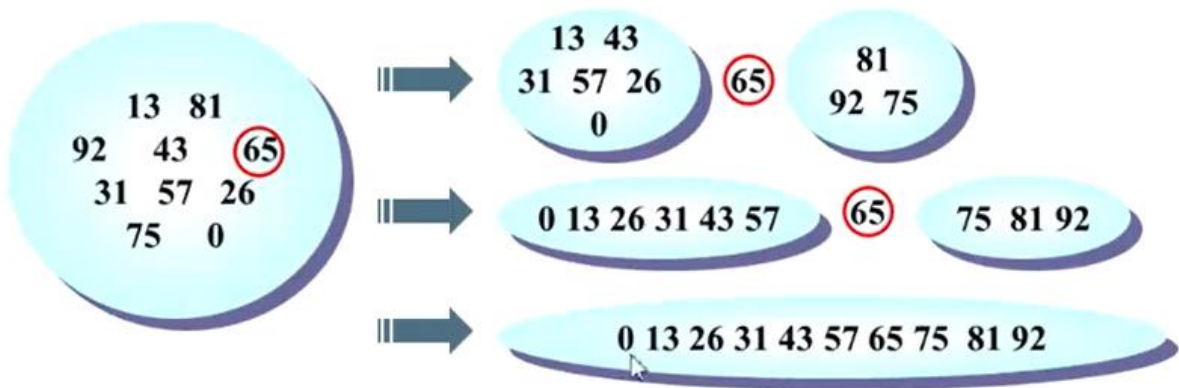
non-recur

评测详情					
测试点	提示	内存(KB)	用时(ms)	结果	得分
0		452	5	答案正确	1 / 1
1		452	4	答案正确	10 / 10
2		440	5	答案正确	2 / 2
3		576	6	答案正确	2 / 2
4		1860	29	答案正确	2 / 2
5		1860	27	答案正确	2 / 2
6		1760	31	答案正确	2 / 2
7		2004	26	答案正确	2 / 2
8		1604	26	答案正确	2 / 2

策略：分而治之

算法概述

■ 分而治之



1. 选择pivot：采用选取头、尾、中间三个位置的中位数，然后把这个中位数放在right-1的位置，接下来划分子集

选主元

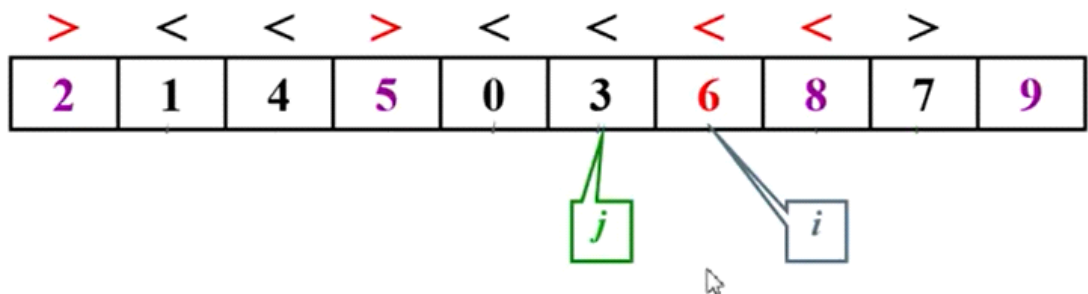
- 随机取 **pivot**? `rand()` 函数不便宜啊!
- 取头、中、尾的中位数
 - 例如 8、12、3的中位数就是8
 - 测试一下**pivot**不同的取法对运行速度有多大影响?

```
ElementType Median3( ElementType A[], int Left, int Right )
{
    int Center = ( Left + Right ) / 2;
    if ( A[ Left ] > A[ Center ] )
        Swap( &A[ Left ], &A[ Center ] );
    if ( A[ Left ] > A[ Right ] )
        Swap( &A[ Left ], &A[ Right ] );
    if ( A[ Center ] > A[ Right ] )
        Swap( &A[ Center ], &A[ Right ] );
    /* A[ Left ] <= A[ Center ] <= A[ Right ] */
    Swap( &A[ Center ], &A[ Right-1 ] ); /* 将pivot藏到右边 */
    /* 只需要考虑 A[ Left+1 ] ... A[ Right-2 ] */
    return
}
```

屏幕剪辑的捕获时间: 2023/8/18 11:38

2. 划分子集:

子集划分



- 如果有元素正好等于**pivot**怎么办?
 - 停下来交换?
 - 不理它, 继续移动指针?



屏幕剪辑的捕获时间: 2023/8/18 11:38

划分时如果遇到与pivot相等的情况, 最好还是选择交换

考虑所有数据都相等的情况, 这样虽然会导致多了许多次无意义的交换, 但是每次划分

完，pivot都比较靠近中间部位，最终时间复杂度接近 $O(N\log N)$
如果选择不交换，那么每次pivot都靠近端点，时间复杂度会达到 $O(N^2)$

而由于快速排序采用递归的方式，所以在处理小规模数据的时候有点不行，这时可以选择划分一个阈值，在数据规模充分小的时候采用简单排序

小规模数据的处理

■ 快速排序的问题

- 用递归.....
- 对小规模的数据（例如 N 不到100）可能还不如插入排序快

■ 解决方案

- 当递归的数据规模充分小，则停止递归，直接调用简单排序（例如插入排序）
- 在程序中定义一个**Cutoff**的阈值 —— 课后去实践一下，比较不同的**Cutoff**对效率的影响

屏幕剪辑的捕获时间: 2023/8/18 11:44

Ex：快速排序设置不同Offset值的差别

2023年8月18日 15:05

题目	用户	提交时间
09-排序1	FSReed	2023/08/18 15:05:17
编译器	内存	用时
C++ (clang++)	1472 / 65536 KB	33 / 10000 ms
状态	分数	评测时间
答案正确	25 / 25	2023/08/18 15:05:17

评测详情					
测试点	提示	内存(KB)	用时(ms)	结果	得分
0		536	5	答案正确	1 / 1
1		476	5	答案正确	10 / 10
2		452	4	答案正确	2 / 2
3		584	7	答案正确	2 / 2
4		1340	27	答案正确	2 / 2
5		1384	33	答案正确	2 / 2
6		1472	20	答案正确	2 / 2
7		1348	21	答案正确	2 / 2
8		1152	23	答案正确	2 / 2

offset = 10

题目	用户	提交时间
09-排序1	FSReed	2023/08/18 15:08:44
编译器	内存	用时
C++ (clang++)	1464 / 65536 KB	26 / 10000 ms
状态 ?	分数	评测时间
答案正确	25 / 25	2023/08/18 15:08:45

offset 50

评测详情					
测试点	提示	内存(KB)	用时(ms)	结果	得分
0		624	6	答案正确	1 / 1
1		460	5	答案正确	10 / 10
2		444	5	答案正确	2 / 2
3		476	7	答案正确	2 / 2
4		1340	26	答案正确	2 / 2
5		1352	20	答案正确	2 / 2
6		1464	20	答案正确	2 / 2
7		1300	21	答案正确	2 / 2
8		1080	24	答案正确	2 / 2

题目	用户	提交时间
09-排序1	FSReed	2023/08/18 15:07:33
编译器	内存	用时
C++ (clang++)	1880 / 65536 KB	26 / 10000 ms
状态 ?	分数	评测时间
答案正确	25 / 25	2023/08/18 15:07:33

offset = 100 .

评测详情					
测试点	提示	内存(KB)	用时(ms)	结果	得分
0		596	4	答案正确	1 / 1
1		664	5	答案正确	10 / 10
2		616	6	答案正确	2 / 2
3		604	7	答案正确	2 / 2
4		1860	26	答案正确	2 / 2
5		1852	19	答案正确	2 / 2
6		1836	24	答案正确	2 / 2
7		1828	26	答案正确	2 / 2
8		1880	25	答案正确	2 / 2

题目	用户	提交时间
09-排序1	FSReed	2023/08/18 15:10:21
编译器	内存	用时
C++ (clang++)	2096 / 65536 KB	32 / 10000 ms
状态 ?	分数	评测时间
答案正确	25 / 25	2023/08/18 15:10:21

200

评测详情					
测试点	提示	内存(KB)	用时(ms)	结果	得分
0		572	4	答案正确	1 / 1
1		580	4	答案正确	10 / 10
2		600	4	答案正确	2 / 2
3		848	7	答案正确	2 / 2
4		1988	28	答案正确	2 / 2
5		2096	19	答案正确	2 / 2
6		1976	21	答案正确	2 / 2
7		2004	32	答案正确	2 / 2
8		1680	25	答案正确	2 / 2

题目	用户	提交时间
09-排序1	FSReed	2023/08/18 15:11:21
编译器	内存	用时
C++ (clang++)	1904 / 65536 KB	30 / 10000 ms
状态	分数	评测时间
答案正确	25 / 25	2023/08/18 15:11:22

评测详情					
测试点	提示	内存(KB)	用时(ms)	结果	得分
0		564	4	答案正确	1 / 1
1		444	4	答案正确	10 / 10
2		728	4	答案正确	2 / 2
3		712	7	答案正确	2 / 2
4		1868	30	答案正确	2 / 2
5		1728	19	答案正确	2 / 2
6		1904	20	答案正确	2 / 2
7		1728	19	答案正确	2 / 2
8		1752	24	答案正确	2 / 2

直接调用库函数

2023年8月18日 15:22

题目	用户	提交时间
09-排序1	FSReed	2023/08/18 15:21:46
编译器	内存	用时
C++ (clang++)	1364 / 65536 KB	40 / 10000 ms
状态	分数	评测时间
答案正确	25 / 25	2023/08/18 15:21:46


qsort

评测详情					
测试点	提示	内存(KB)	用时(ms)	结果	得分
0		452	5	答案正确	1 / 1
1		444	5	答案正确	10 / 10
2		592	6	答案正确	2 / 2
3		564	10	答案正确	2 / 2
4		1364	37	答案正确	2 / 2
5		1284	28	答案正确	2 / 2
6		1308	40	答案正确	2 / 2
7		1328	26	答案正确	2 / 2
8		1152	32	答案正确	2 / 2

算法概述

■ 间接排序

- 定义一个指针数组作为“表”（table）



A	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
key	f	d	c	a	g	b	h	e
table	3	5	2	1	7	0	4	6

如果仅要求按顺序输出，则输出：


$A[\text{table}[0]], A[\text{table}[1]], \dots, A[\text{table}[N-1]]$

基本思想就是当排序的指标只是结构体内的一小部分时，不移动原有数据，而是将各个数据的指针排序，按需访问原数据

如果需要物理排序，有一个很好的结论可以使得算法时间复杂度达到 $O(N)$ 级别：

物理排序

- N个数字的排列由若干个独立的环组成



A	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
key	a	b	c	d	g	f	h	e
table	3	5	2	1	7	0	4	6

Temp = f

如何判断一个环的结束？

```
if ( table[i] == i )
```

此时排序的最坏情况就是有 $N/2$ 个环，每个环都有2个元素待排：

复杂度分析

- 最好情况：初始即有序
- 最坏情况：
 - 有 $\lfloor N/2 \rfloor$ 个环，每个环包含2个元素
 - 需要 $\lfloor 3N/2 \rfloor$ 次元素移动

$T = O(mN)$ ， m 是每个A元素的复制时间。

正是因为复制元素的时间不容忽视，才有了这种最坏情况

基数排序

2023年8月18日

15:52

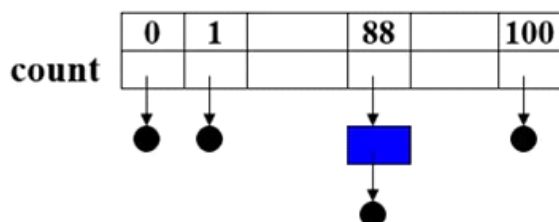
1. 桶排序:

桶排序

中国大学



假设我们有 N 个学生，他们的成绩是0到100之间的整数（于是有 $M = 101$ 个不同的成绩值）。如何在线性时间内将学生按成绩排序？



```
void Bucket_Sort(ElementType A[], int N)
{
    count[] 初始化;
    while (读入1个学生成绩grade)
        将该生插入count[grade]链表;
    for ( i=0; i<M; i++ ) {
        if ( count[i] )
            输出整个count[i]链表;
    }
}
```

$$T(N, M) = O(M + N)$$

当M很小时，这个方法没啥问题，但是如果 $M \gg N$ 呢？

2. 基数排序:

基数排序



假设我们有 $N = 10$ 个整数，每个整数的值在0到999之间（于是有 $M = 1000$ 个不同的值）。还有可能在**线性时间**内排序吗？

输入序列: 64, 8, 216, 512, 27, 729, 0, 1, 343, 125

用“次位优先”（**Least Significant Digit**）

Bucket	0	1	2	3	4	5	6	7	8	9
Pass 1	0	1	512	343	64	125	216	27	8	729
Pass 2	0	512	125		343		64			
	1	216	27							
	8		729							
Pass 3	0	125	216	343		512		729		
	1									
	8									
	27									
	64									

屏幕剪辑的捕获时间: 2023/8/18 16:07

每一次入桶完成后，按照桶的顺序把各个数据串起来，然后新建一组桶，按顺序把所有数据放入桶中，假设是B进制，这样每组有B个桶，需要进行P趟排序，N个数据，时间复杂度就是 $O(P(B+N))$ ，当桶比较少的时候比较划算（一般P都是logB的级别）

如果用最大位优先（MSD），就是按照最大位入桶，然后每个桶创建一个子桶排序后放回最大桶，有点像递归

3. 把基数排序引申到多关键字排序，比如扑克牌排序：

多关键字的排序



一副扑克牌是按**2**种关键字排序的

K^0 [花色]

♣ < ♦ < ♥ < ♠

K^1 [面值]

2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A

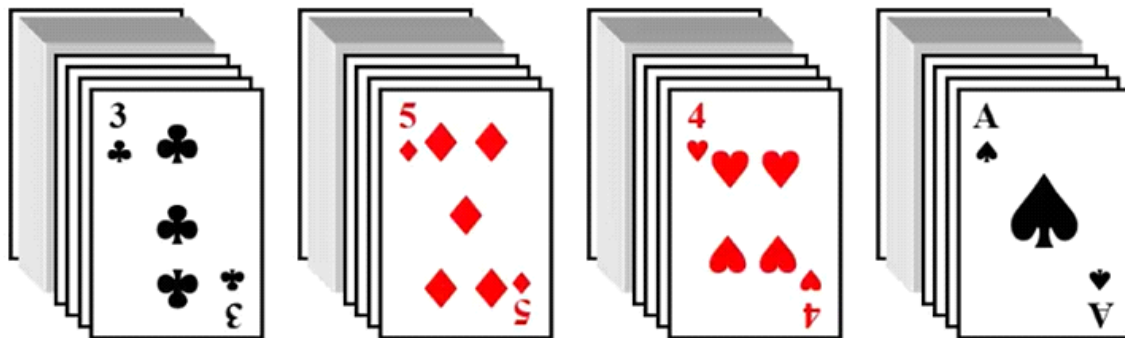
有序结果：

2♣ ... A♣ 2♦ ... A♦ 2♥ ... A♥ 2♠ ... A♠

☞ 用“主位优先”（**Most Significant Digit**）排序：为花色建4个桶

有序结果： 2♣ ... A♣ 2♦ ... A♦ 2♥ ... A♥ 2♠ ... A♠

☞ 用“主位优先” (**Most Significant Digit**) 排序：为花色建4个桶



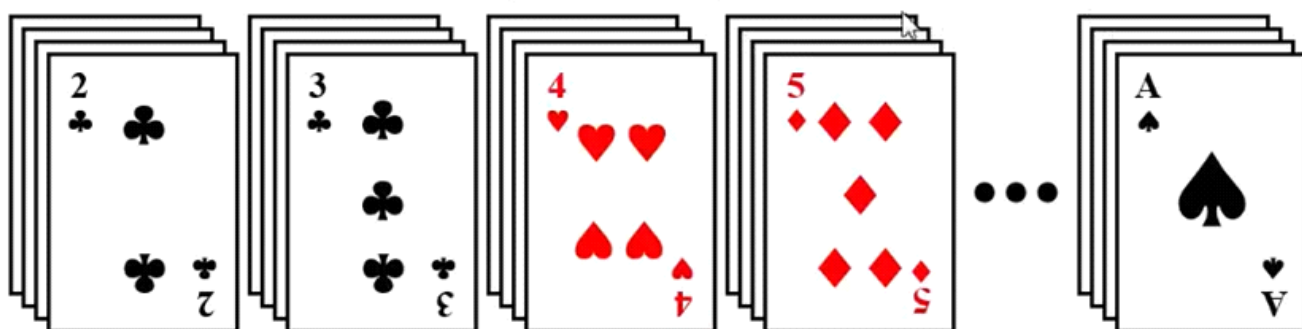
在每个桶内分别排序，最后合并结果。

这种情况更适合用次位优先：

多关键字的排序

中国大学MOOC

■ 用“次位优先” (**Least Significant Digit**) 排序：为面值建13个桶



■ 将结果合并，然后再为花色建4个桶

屏幕剪辑的捕获时间: 2023/8/18 16:24