23 第九讲 排序（上）

笔记本：	浙江大学《数据结构》		
创建时间：	2025/5/5 19:56	**更新时间：**	2025/5/24 18:05
作者：	panhengye@163.com		
URL：	https://www.doubao.com/chat/4267992299182594		

使用建议： 对于小规模数据（n < 50），插入排序通常表现较好 对于中等规模数据，希尔排序是一个不错的选择 对于大规模数据，归并排序和堆排序的性能更稳定 如果内存空间有限，建议使用原地排序算法（如堆排序）

简单排序

前提

Void X_Sort (Element_Type A[], int N)

- 大多数情况下，讨论 从小到大的整数 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：任意两个相等数据，排序前后的相对位置不发生改变

> 没有一种排序设在任何情况下都表现最好的

冒泡排序（稳定）

```
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 */
   }
}
```

```
arr = [34, 8, 64, 51, 32, 21] n = len(arr) swap_count = 0 for i in range(n):
    for j in range(0, n - i - 1):          if arr[j] > arr[j + 1]:
  arr[j], arr[j + 1] = arr[j + 1], arr[j]            swap_count += 1
print("交换次数:", swap_count)
```

最好情况：顺序 T = O(N)

最坏情况：顺序 T = $O(N^2)$

插入排序（稳定）

```
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 */
    }
}
```

```
def insertion_sort(arr):     for i in range(1, len(arr)):         key =
arr[i]         j = i - 1          while j >= 0 and key < arr[j]:
arr[j + 1] = arr[j]             j = j - 1          arr[j + 1] = key
return arr arr = [34, 8, 64, 51, 32, 21] sorted_arr = insertion_sort(arr)
print(sorted_arr)
```

最好情况：顺序 T = O(N)

最坏情况：顺序 T = O(N^2)

时间复杂度下界

对于下标i<j，如果A[i]>A[j]，则称(i j)是一对逆序对(inversion)

- {34, 8, 64, 51, 32, 21}有九个逆序对
- 交换2个相邻元素正好消去1个逆序对

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Ω(N^2)

> 要提高算法效率，必须做到“每次交换相隔较远的逆序对”

希尔排序

Shell sort _ by Donald Shell

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

```
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;         }     }}
```

坏消息是最差情况下，效率是N^2
原因是：增量元素不互质，则小增量可能根本不起作用

> 启发：一个算法可以是如此简单，但是它的复杂度分析却可能异常困难

堆排序

选择排序（Selection Sort)

1. **初始状态**：将待排序序列分为已排序和未排序两部分。初始时，已排序部分为空，未排序部分包含整个待排序序列。
2. **寻找最小元素**：在每一轮排序中，从未排序部分中找出最小（或最大，取决于升序还是降序需求，这里以升序为例）的元素 。比如在给定代码中，ScanForMin(A, i, N - 1)函数就是从A[i]到A[N - 1]这个未排序区间内寻找最小元素的位置。
3. **交换元素**：将找到的最小元素与未排序部分的第一个元素交换位置。在代码里，通过Swap(A[i], A[MinPosition])语句，把未排序部分找到的最小元素A[MinPosition]和当前未排序部分起始位置的A[i]进行交换，这样就使得已排序部分增加一个元素，未排序部分减少一个元素 。
4. **重复操作**：不断重复上述步骤 2 和步骤 3，直到未排序部分为空，此时整个序列就完成了排序。

> 选择排序的时间复杂度在最坏、平均和最好情况下均为(O(n^2)) ，空间复杂度为(O(1)) ，属于不稳定的排序算法

堆排序

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(logN) */     for
( i=0; i<N; i++ ) /* O(N) */         A[i] = TmpA[i];}
```

存在的问题：需要额外O（N）空间，并且复制元素需要时间

算法2

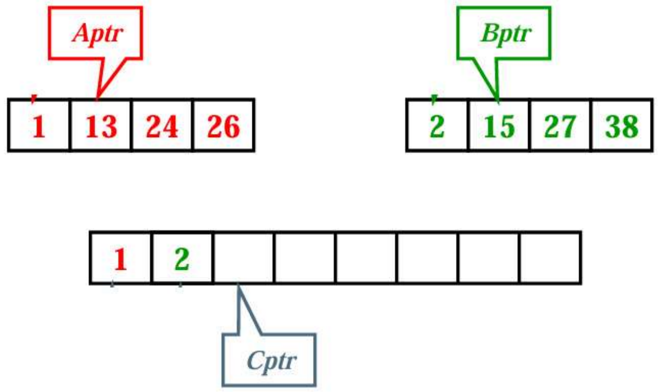
```
void Heap_Sort ( ElementType A[], int N ){     for ( i=N/2-1; i>=0; i-- ) /*
BuildHeap */         PercDown( A, i, N );     for ( i=N-1; i>0; i-- ) {
    Swap( &A[0], &A[i] ); /* DeleteMax */         PercDown( A, 0, i );
}}
```

归并排序

核心：有序子列的合并

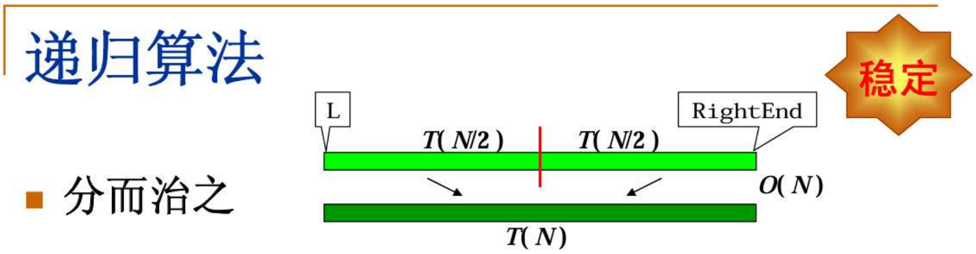
```
/* L = 左边起始位置，R = 右边起始位置，RightEnd = 右边终点位置 */ void Merge(
ElementType A[], ElementType TmpA[],             int L, int R, int RightEnd
) {     LeftEnd = R - 1; /* 左边终点位置。假设左右两列挨着 */     Tmp = L; /*
存放结果的数组的初始位置 */     NumElements = RightEnd - L + 1;     while( L <=
LeftEnd && R <= RightEnd ) {   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

原理展示



具体实现方法1：递归

原理：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 );     } }
```

特点：

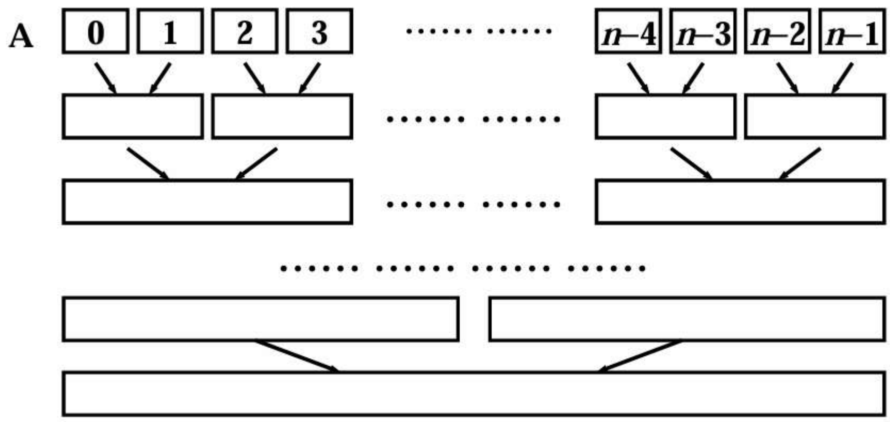
- 时间复杂度：T(N) = O(NlogN)
- 稳定

统一接口

```
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" ); }
```

具体实现方法2：非递归

原理



假设有数组 [4, 2, 1, 3]，算法执行过程如下：

1. 第一轮（step=1）：
    - 合并 [4] 和 [2] → [2, 4]
    - 合并 [1] 和 [3] → [1, 3]
    - 结果：[2, 4, 1, 3]
1. 第二轮（step=2）：
    - 合并 [2, 4] 和 [1, 3] → [1, 2, 3, 4]
    - 结果：[1, 2, 3, 4]

伪码描述

```
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]; }
```

统一接口

```
void Merge_sort( ElementType A[], int N ) {     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"
); }
```