第十讲：排序（下）

10.1 快速排序

10.1.1 算法概述

策略：分而治之。

下面举个例子，假如一组数为 13/81/92/43/65/31/57/26/75/0，我们对其进行排序。那么首先选择出一个主元，这里我们选择为 65，那么将这组数的其他成员分为了两组，一组是小于主元的 13/43/31/57/26/0，一组是大于主元的 81/92/75.然后将其递归处理，两边各选一个主元再进行分组……倒数第二步的时候，我们在第一步选择出来的主元左侧已经排好了顺序，右侧也排好了顺序，这样将它们放在同一个数组中，就完成了排序。

以下是上面这段话的伪码描述：

```
void QuickSort(ElementType A[], int N)
{
    if(N<2)
        return;
    pivot = 从 A[]中选一个主元;
    将 S={A[]\PIVOT}分成 2 个独立子集:
    A1={a∈S|a≤piovt}和
    A2={a∈S|a≥piovt}
    A[]=Quic_Sort(A1,N1)∪(pivot)∪QuickSort(A2,N2);
}
```

这个方法的关键在于**主元的选取**（选取不好，快速算法会很慢）和**子集划分**（这个过程也是耗费时间的一个地方）。

快速排序算法的最好情况就是**每次正好中分**，**T(N)=O(NlogN)**。

10.1.2 选主元

选取头、中、尾的中位数，也可以选取 5 个、7 个等数字的中位数。例如 8/12/3 的中位数就是 8。伪码描述如下：

```
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]);
    Swap(&A[Center],&A[Right-1]);/*将 pivot 藏到右边*/
    /*只需要考虑 A[left+1]到 A[Right-2]，这样来划分左右*/
    return A[Right-1];
}
```

10.1.3 子集划分

为了便于理解，还是举一个例子：

8	1	4	9	0	3	5	2	7	6（M）

定义两个指针 i（指向第一个元素）和 j（指向中位数左侧的元素）。（**继续强调，这里的 i 和 j 不是实际的指针，而是存储所需要元素下标的整数。**）先比较 i 代表的元素和主元的大小，发现 8 大于 6，那么 i 这个指针不变；然后看 j 代表的元素和主元比较，发现 7 大于 6，将 j 减一（即左移一位），然后再比较，发现 2 小于 6，这样就不对了，j 停止移动。在 i 和 j 都停止移动后，将其所指向的两个元素交换位置，变成了下面这个样子。

2	1	4	9	0	3	5	8	7	6（M）

然后 i 加 1（右移一位），1 小于 6，正常，i 继续加 1（右移一位），4 小于 6，正常，i 继续加 1（右移一位），此时 9 大于 6，不正常，i 停止；再看 j 减 1（左移一位），5 小于 6，不正常，j 停止，然后交换此时 i 和 j 所代表的元素，变成下面这个样子。

2	1	4	5	0	3	9	8	7	6（M）

然后 i 加 1（右移一位），发现正常，i 继续加 1，发现正常，i 再加 1，发现 9 小于 6，不正常，i 停止；然后 j 左移一位，发现 3 小于 6，不正常，停止。

此时发现 i-j<0 了，子集划分结束，同时将 i 代表的元素和主元交换，完成了子集的划分。最后结果如下：

2	1	4	5	0	3	6（M）	8	7	9

快速算法的“快速”在于，划分完成后其主元被一次性放到了正确的位置再也不会移动；例如插入算法等都需要一步一步往后移。

如果有元素正好等于 pivot 怎么办？停下来处理。

对于小规模数据还不如用插入排序。当递归的数据规模充分小，则停止递归，直接调用简单排序。在程序中定义一个 Cutoff 的阈值。

10.1.4 算法实现

伪码描述：

```
void Quicksort(ElementType A[], int Left, int Right)
{
    if(Cutoff<=Right-Left)
    {
        Pivot = Median3(A, Left, Right);
        i=Left;
        j= Right-1;
        for(;;)
        {
```

```
            while(A[++i]<Pivot){}
            while(A[--j]>Pivot){}
            if(i<j)
                Swap(&A[i],&A[j]);
            else break;
        }
        Swap(&A[i],&A[Right-1]);
        Quicksort(A,Left,i-1);
        Quicksort(A,i+1,Right);
    }
    else
        Insertion_Sort(A+Left,Right-Left+1);


}
```

为了统一接口，在上段程序后面再加一个：

```
void Quick_Sort(ElementType A[], int N)
{
    Quicksort(A, 0, N-1);
}
```

快速排序算法是**不稳定**算法！

以下给出另外的 C 语言编写的代码，它是直接调用函数库：

```
/* 快速排序 - 直接调用库函数 */

#include <stdlib.h>

/*---------------简单整数排序--------------------*/
int compare(const void *a, const void *b)
{ /* 比较两整数。非降序排列 */
    return (*(int*)a - *(int*)b);
}
/* 调用接口 */
qsort(A, N, sizeof(int), compare);
/*---------------简单整数排序--------------------*/


/*--------------- 一般情况下，对结构体 Node 中的某键值 key 排序 ---------------*/
struct Node {
    int key1, key2;
} A[MAXN];

int compare2keys(const void *a, const void *b)
{ /* 比较两种键值：按 key1 非升序排列；如果 key1 相等，则按 key2 非降序排列 */
    int k;
```

```
    if ( ((const struct Node*)a)->key1 < ((const struct Node*)b)->key1 )
        k = 1;
    else if ( ((const struct Node*)a)->key1 > ((const struct Node*)b)->key1 )
        k = -1;
    else { /* 如果 key1 相等 */
        if ( ((const struct Node*)a)->key2 < ((const struct Node*)b)->key2 )
            k = -1;
        else
            k = 1;
    }
    return k;
}
/* 调用接口 */
qsort(A, N, sizeof(struct Node), compare2keys);
/*--------------- 一般情况下，对结构体 Node 中的某键值 key 排序 ---------------*/
```

10.2 表排序

10.2.1 算法概述

表排序用于：元素不是简简单单的排序，每一个元素都是一个庞大的结构，例如一个结构体等，此时如果我们想要交换两个元素，就不能忽略交换所需要的时间了。表排序就是在移动的时候，并不移动原始的数据，只是移动指向它的指针。

它是一种间接排序的方法，定义一个指针数组作为“表”（table）。

用下面一个例子来举例：

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

利用插入算法，可以得出：

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[table[0]],A[table[1]],……,A[table[N-1]]

10.2.2 物理排序

所谓物理排序就是说，我们不能像前文那样利用指针来移动，而是必须实际移动结构体，那该怎么办呢？

利用这样一个原理：**N 个数字的排列由若干个独立的环组成**。

为了解释这个原理，看 10.2.1 中最后的结果那个表，table[0]值（3）->table[3]值（1）->table[1]的值（5）->table[5](0)返回到了 table[0]。一共有三种环，这三种环互不相交，称

为独立。

在排序的时候，先对一个环内进行排序。那么如何判断一个环的结束呢？

每访问一个空位 i 后，就令 table[i]=i。当发现 table[i]==i 时，环就结束了。

下面分析一下复杂度的情况：

（1）最好的情况：初始即有序。

（2）最坏的情况：有 N/2 个环，每个环包含两个元素，元素需要移动 3N/2 次。

但是无论如何，复杂度都可以写为 T(N)=O(mN)，其中 m 是每个 A 元素复制需要的时间。

10.3 基数排序

10.3.1 桶排序

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

我们可以为每一个成绩值构造一个“**桶**”，于是就有了 101 个桶，如图 1 所示。如果我们有一个 88 分的学生，那么就把学生的信息查到 88 分的这个链表的表头里，伪码描述如下：

```
void Bucket_Sort(ElementType A[], int N)
{
    count[]初始化;
    while (读入读入 11 个学生成绩个学生成绩 graded)
        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;
    }
}
```

时间复杂度：**T(N,M)=O(M+N)**

插入学生的成绩，因为是 N 个学生，所以复杂度为 O(N)；输出成绩，for 循环，M 个成绩，自然复杂度为 O(M)。故总的时间复杂度如上。

如果有 N=40000 个学生，由于 M=101，故这就是一个线性的复杂度。

图 1

但是，如果 M>>N 怎么办？

10.3.2 基数排序

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

这里的基数就是跟进制有关，如果是二进制，基数就是 2；如果是十进制，基数就是 10。这里我们说的是十进制，所以**基数就是 10**。

加入我们的输入序列为：64,8,216,512,27,729,0,1,343,125，用“次位优先”（Least Significant Digit，LSD）算法。主位是指的第一位，剩余的都叫次位。因此这里比较我们先从个位开始。

首先我们建立十个“桶”，然后将它们以个位数为标准放到这 10 个桶里面去，如表中 Pass 1；然后按照十位数为标准放到 10 个桶里，如 Pass 2；在完成之后，我们来做一个收集，收集的过程就是扫描每一个桶，然后把桶里的元素按照 0,1,8,512,216……顺序用一个链表把它们串起来；串起来后，按照主位放到相应的桶里，如 Pass3；最后，再对它们进行一次收集，用链表连起来，得出结果。

表 3.1 基数排序算法流程表

Bucket	0	1	2	3	4	5	6	7	8	9
Pass 1	0	1	512	343	64	125	216	27	8	729
Pass2	0	512	125		343		64			
	1	216	27							
	8		729							
Pass3	0	125	216	343		512		729		
	1									
	8									
	27									
	64									

时间复杂度：T(N,B,P)=O(P(N+B))。其中 P 为基数排序次数（本例子例为 3），N 为输入序列中数字个数，B 为整数进制。

10.3.3 多关键字排序

例如，一副扑克牌是按 2 种关键字排序的，如图 2 所示，花色是它的主关键字，面值是它的次关键字。

在这里，我们可以用“主位优先”（Most Significant Digit, MSD）排序，为花色建立 4 个桶，在每个桶里分别排序，最后合并结果。

也可以使用“次位优先”（LSD）算法排序，为面值建立 13 个桶，就可以直接将结果合并，然后为花色建 4 个桶，放进去就可以了。

在这里，LSD 优点是不需要排序，时间复杂度小的多。

基数排序是**稳定**的算法。

补充

（1）次位优先的 C 语言代码：

```
/* 基数排序 - 次位优先 */

/* 假设元素最多有 MaxDigit 个关键字，基数全是同样的 Radix */
#define MaxDigit 4
#define Radix 10
```

```
/* 桶元素结点 */
typedef struct Node *PtrToNode;
struct Node {
    int key;
    PtrToNode next;
};

/* 桶头结点 */
struct HeadNode {
    PtrToNode head, tail;
};
typedef struct HeadNode Bucket[Radix];

int GetDigit ( int X, int D )
{ /* 默认次位 D=1, 主位 D<=MaxDigit */
    int d, i;

    for (i=1; i<=D; i++) {
        d = X % Radix;
        X /= Radix;
    }
    return d;
}

void LSDRadixSort( ElementType A[], int N )
{ /* 基数排序 - 次位优先 */
     int D, Di, i;
     Bucket B;
     PtrToNode tmp, p, List = NULL;

     for (i=0; i<Radix; i++) /* 初始化每个桶为空链表 */
         B[i].head = B[i].tail = NULL;
     for (i=0; i<N; i++) { /* 将原始序列逆序存入初始链表 List */
         tmp = (PtrToNode)malloc(sizeof(struct Node));
         tmp->key = A[i];
         tmp->next = List;
         List = tmp;
     }
     /* 下面开始排序 */
     for (D=1; D<=MaxDigit; D++) { /* 对数据的每一位循环处理 */
         /* 下面是分配的过程 */
         p = List;
         while (p) {
```

```
            Di = GetDigit(p->key, D); /* 获得当前元素的当前位数字 */
            /* 从 List 中摘除 */
            tmp = p; p = p->next;
            /* 插入 B[Di]号桶尾 */
            tmp->next = NULL;
            if (B[Di].head == NULL)
                B[Di].head = B[Di].tail = tmp;
            else {
                B[Di].tail->next = tmp;
                B[Di].tail = tmp;
            }
        }
        /* 下面是收集的过程 */
        List = NULL;
        for (Di=Radix-1; Di>=0; Di--) { /* 将每个桶的元素顺序收集入 List */
            if (B[Di].head) { /* 如果桶不为空 */
                /* 整桶插入 List 表头 */
                B[Di].tail->next = List;
                List = B[Di].head;
                B[Di].head = B[Di].tail = NULL; /* 清空桶 */
            }
        }
    }
    /* 将 List 倒入 A[]并释放空间 */
    for (i=0; i<N; i++) {
       tmp = List;
       List = List->next;
       A[i] = tmp->key;
       free(tmp);
    }
}
```

（2）主位优先 C 语言代码:

```
/* 基数排序 - 主位优先 */

/* 假设元素最多有 MaxDigit 个关键字，基数全是同样的 Radix */

#define MaxDigit 4
#define Radix 10

/* 桶元素结点 */
typedef struct Node *PtrToNode;
struct Node{
    int key;
    PtrToNode next;
```

```
};

/* 桶头结点 */
struct HeadNode {
    PtrToNode head, tail;
};
typedef struct HeadNode Bucket[Radix];

int GetDigit ( int X, int D )
{ /* 默认次位 D=1, 主位 D<=MaxDigit */
    int d, i;

    for (i=1; i<=D; i++) {
        d = X%Radix;
        X /= Radix;
    }
    return d;
}

void MSD( ElementType A[], int L, int R, int D )
{ /* 核心递归函数: 对 A[L]...A[R]的第 D 位数进行排序 */
     int Di, i, j;
     Bucket B;
     PtrToNode tmp, p, List = NULL;
     if (D==0) return; /* 递归终止条件 */

     for (i=0; i<Radix; i++) /* 初始化每个桶为空链表 */
         B[i].head = B[i].tail = NULL;
     for (i=L; i<=R; i++) { /* 将原始序列逆序存入初始链表 List */
         tmp = (PtrToNode)malloc(sizeof(struct Node));
         tmp->key = A[i];
         tmp->next = List;
         List = tmp;
     }
     /* 下面是分配的过程 */
     p = List;
     while (p) {
         Di = GetDigit(p->key, D); /* 获得当前元素的当前位数字 */
         /* 从 List 中摘除 */
         tmp = p; p = p->next;
         /* 插入 B[Di]号桶 */
         if (B[Di].head == NULL) B[Di].tail = tmp;
         tmp->next = B[Di].head;
         B[Di].head = tmp;
```

```
        }
        /* 下面是收集的过程 */
        i = j = L; /* i, j 记录当前要处理的 A[]的左右端下标 */
        for (Di=0; Di<Radix; Di++) { /* 对于每个桶 */
            if (B[Di].head) { /* 将非空的桶整桶倒入 A[], 递归排序 */
                p = B[Di].head;
                while (p) {
                    tmp = p;
                    p = p->next;
                    A[j++] = tmp->key;
                    free(tmp);
                }
                /* 递归对该桶数据排序, 位数减 1 */
                MSD(A, i, j-1, D-1);
                i = j; /* 为下一个桶对应的 A[]左端 */
            }
        }
}

void MSDRadixSort( ElementType A[], int N )
{ /* 统一接口 */
    MSD(A, 0, N-1, MaxDigit);
}
```

10.4 排序算法的比较

排序方法	平均时间复杂度	最坏情况下时间复杂度	额外空间复杂度	稳定性
简单选择排序	$O(N^2)$	$O(N^2)$	O(1)	不稳定
冒泡排序	$O(N^2)$	$O(N^2)$	O(1)	稳定
直接插入排序	$O(N^2)$	$O(N^2)$	O(1)	稳定
希尔排序	$O(N^d)$	$O(N^2)$	O(1)	不稳定
堆排序	O(NlogN)	O(NlogN)	O(1)	不稳定
快速排序	O(NlogN)	$O(N^2)$	O(logN)	不稳定
归并排序	O(NlogN)	O(NlogN)	O(N)	稳定
基数排序	O(P(N+B))	O(P(N+B))	O(N+B)	稳定

前三种排序算法的共同优点是算法编写简单。冒泡排序和直接插入排序每次都是交换两个元素，因此是慢的，但是它们稳定，简单选择排序是跳着交换，有可能不稳定。

希尔排序算法打破了 N 的平方的复杂度，它的好坏取决于 d（增量序列），因为是跳着排的，所以它也是不稳定的。

堆排序和归并排序的时间复杂度是最好的，无论何时都是一样的。归并排序的缺点是它需要一个额外的空间，当数据量非常大的时候，只能排一半数据，但是它的优点是它是稳定的。堆排序理论上看很美，实际情况是虽然理论上是 O(NlogN)，但是 O 这个常数会比较大，

所以它到底跟快速排序哪个快，就难说了。堆排序和快速排序的共同缺点是不稳定，快速排序总可以构造一种最糟糕的情况是 $O(N^2)$，而且因为是递归的，所以额外空间是需要的，时间复杂度最好时，额外空间复杂度也是 O(logN)。

基数排序在某种情况下，会打破 NlogN 的魔咒，会更快，近乎线性。它需要的额外空间是需要 B 个桶，每个桶设置 B 个数据的位置，所以到底什么情况下合算，看情况，它的好处是它是稳定的。