数据结构之排序

秦新良

2014 年 8 月 20 日

目　录

1 排序

本篇介绍常用的排序算法并给出其实现，最后对各种排序算法作比较。代码1给出了排序算法链表实现的链表定义。

```
#ifndef __LIST_H__
#define __LIST_H__

typedef struct node_s  node_t;
typedef struct node_s *list_t;

struct node_s
{
    int      data;
    node_t *next;
};

#endif /* __LIST_H__ */
```

代码 1: 链表

1.1 插入排序

插入排序的一个非常形象的说明就是打牌。起牌的过程中，每次我们摸起一张牌后，都会和手中已有的牌做比较，然后将牌插到相应的位置，如图1所示。

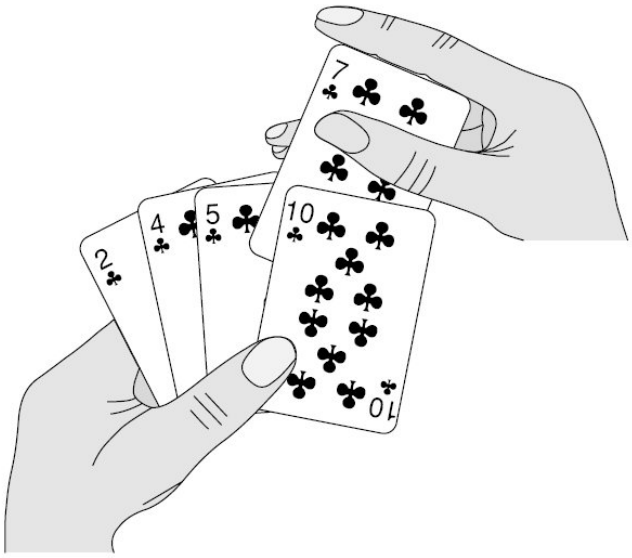


图 1: 打牌

整个起牌的过程，就是对手中的牌的一次插入排序过程。当然，也不排除有的人对手中的牌并不排序，因为这样可以有效的防止“邻居偷窥“，该场景就不在我们的举例范围之内了。

插入排序的时间复杂度为O(n^2)，排序过程中只需O(1)个元素的额外空间即可完成整个数组的排序。代码2给出了插入排序的C代码实现。

```c
void insert_sort(int array[], int size)
{
    int i = 0;
    int j = 0;
    int temp = 0;

    for (i = 1; i < size; ++i)
    {
        for (j = i; j > 0; --j)
        {
            if (array[j] < array[j-1])
            {
                temp = array[j];
                array[j] = array[j-1];
                array[j-1] = temp;
            }
            else
            {
                break;
            }
        }
    }

    return;
}
```

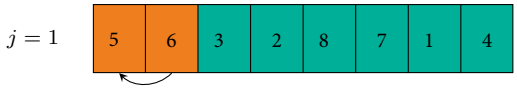
代码 2: 插入排序

为了更清楚的理解该算法，下面以排序6、5、3、2、8、7、1、4这一组数为例来说明插入排序的算法原理。

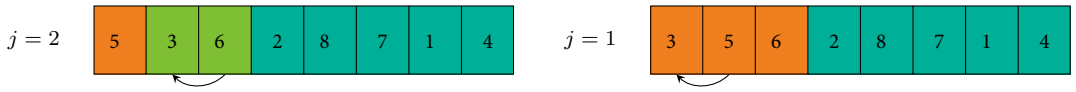
i = 0[1]时：

i = 1 时：

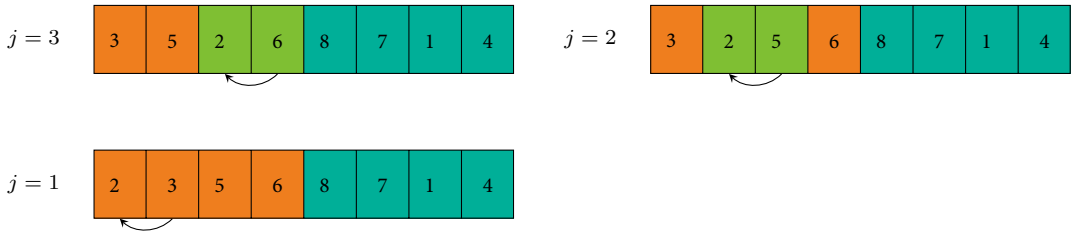
[1]真实的排序不会从i = 0开始，这里用（i = 0）&&（j = 0）表示初始未排序。



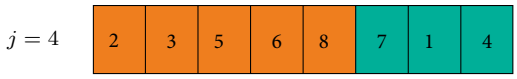
i = 2 时：



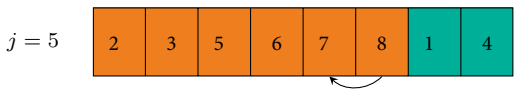
i = 3 时：



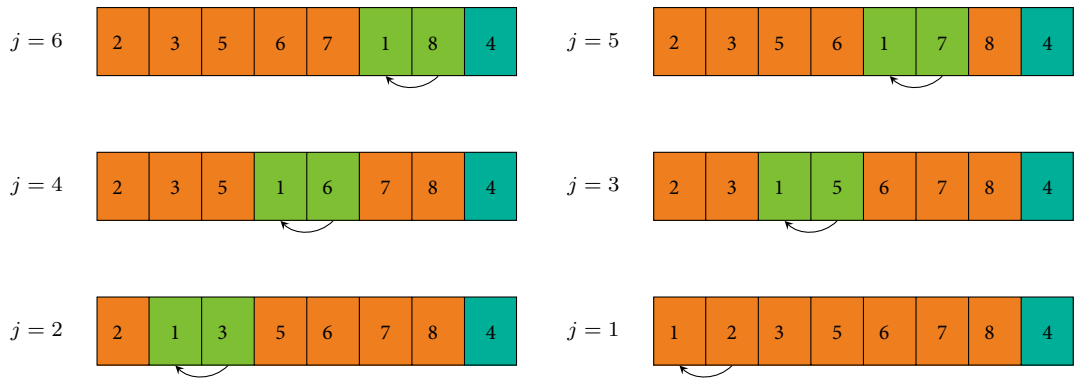
i = 4 时：



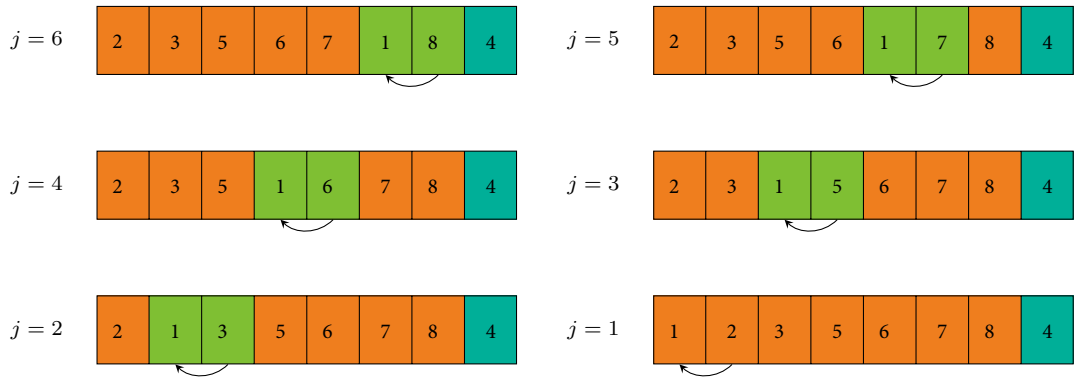
i = 5 时：



i = 6 时：



i = 7 时：



插入排序使用的是增量排序方法，即在排好的子数组 A[0..j-1] 中插入元素 A[j]，形成排好序的子数组 A[0..j]。

插入排序是稳定的[2]排序算法，在待排序元素基本有序的情况下，其最好可达到接近 O(n) 的时间复杂度。在元素基本有序或元素不多的情况下，可选择插入排序。

代码2并不是插入排序的最优实现，还有一种实现方法可减少元素交换时元素之间的拷贝，有兴趣的读者可以尝试实现之。最后，我们以插入排序的链表实现来结束本小节，见代码3。

```
1 void list_insert_sort(list_t *list_head)
2 {
```

[2]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，ri=rj，且 ri 在 rj 之前，而在排序后的序列中，ri 仍在 rj 之前，则称这种排序算法是稳定的；否则称为不稳定的。

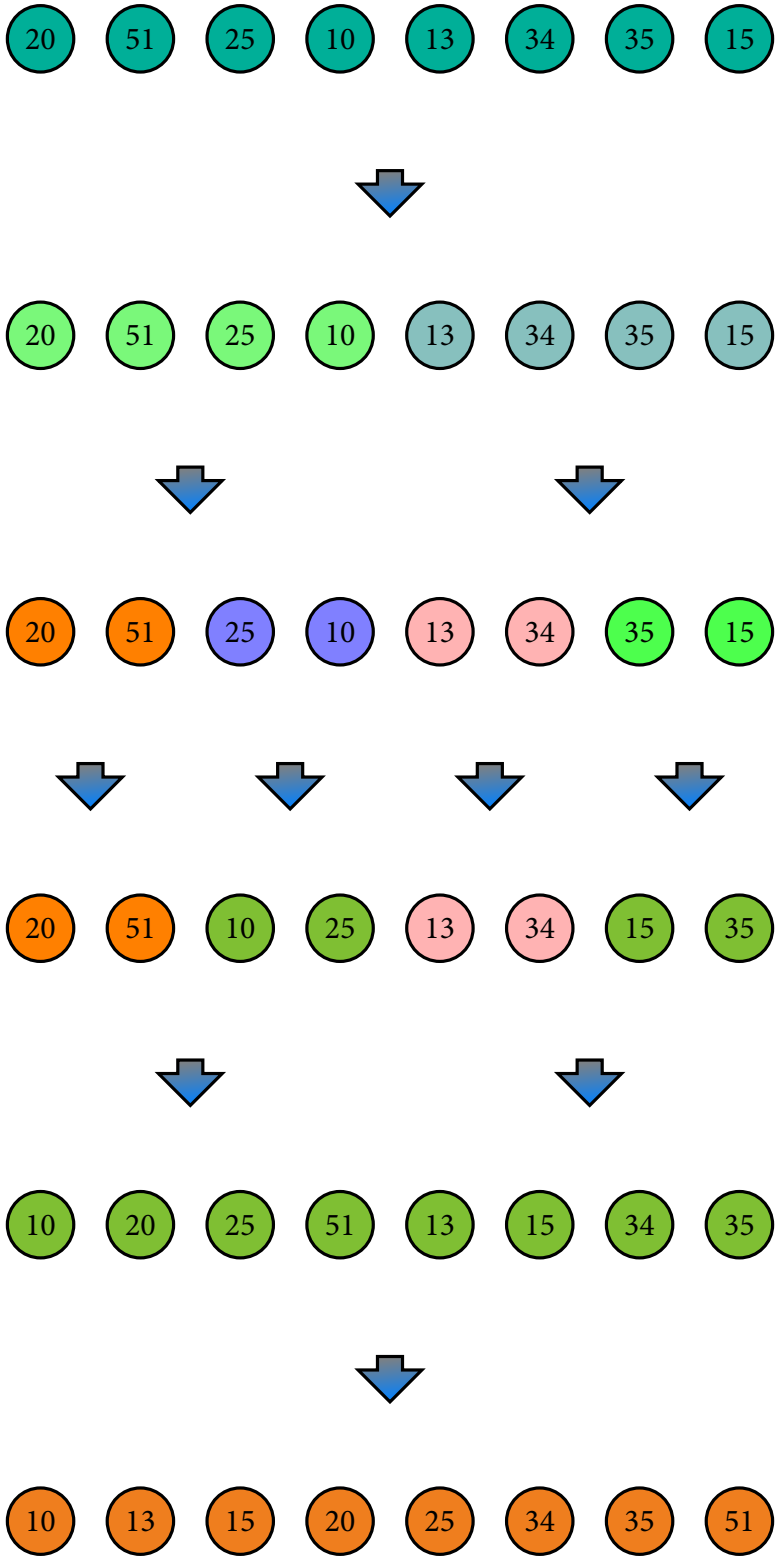
```
    if ((NULL == list_head) || (NULL == *list_head))
    {
        return;
    }

    node_t *p = NULL;
    node_t *c = NULL;
    node_t *n = NULL;
    node_t *t = NULL;

    n = (*list_head)->next;
    (*list_head)->next = NULL;

    while (NULL != n)
    {
        p = c = *list_head;
        do
        {
            if (n->data < c->data)
            {
                t = n;
                n = n->next;

                if (p == c)
                {
                    t->next = c;
                    *list_head = t;
                }
                else
                {
                    t->next = c;
                    p->next = t;
                }

                break;
            }
            else
            {
                if (p == c)
                {
                    c = c->next;
                }
                else
                {
                    p = c;
                    c = c->next;
                }

                if (NULL == c)
                {
```

```
53                      t = n;
54                      n = n->next;
55                      t->next = NULL;
56                      p->next = t;
57                      break;
58                  }
59              }
60          } while (1);
61      }
62
63      return;
64  }
```

代码 3: 插入排序

1.2 归并排序

归并排序采用分而治之的思想,


20 51 25 10 13 34 35 15
20 51 25 10 13 34 35 15
20 51 25 10 13 34 35 15
20 51 10 25 13 34 15 35
10 20 25 51 13 15 34 35
10 13 15 20 25 34 35 51

1.3 冒泡排序

冒泡排序可以说是所有排序算法中最“有名”的一个，因为其名字好记、原理简单：循环对数组中相邻的元素两两比较，每次循环结束后，都会有一个元素在其正确的位置上，即已排序。代码4给出其C语言的实现。

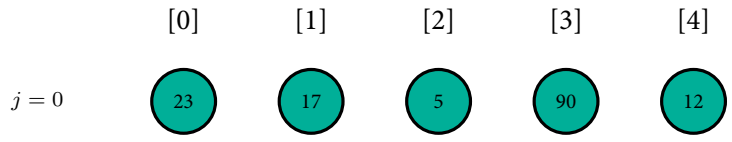
```c
void bubble_sort(int array[], int size)
{
    int i = 0;
    int j = 0;
    int temp = 0;

    for (i = size - 1; i > 0 ; --i)
    {
        for (j = 0 ; j < i; ++j)
        {
            if (array[j + 1] < array[j])
            {
                temp = array[j];
                array[j] = array[j+1];
                array[j+1] = temp;
            }
        }
    }

    return;
}
```

代码 4: 冒泡排序

从代码可以看出，冒泡排序的时间复杂度为 $O(n^2)$。同插入排序一样，排序过程中只需要一个额外的元素空间。但代码4并不是该算法的最优实现，有兴趣的读者可以尝试优化该实现。

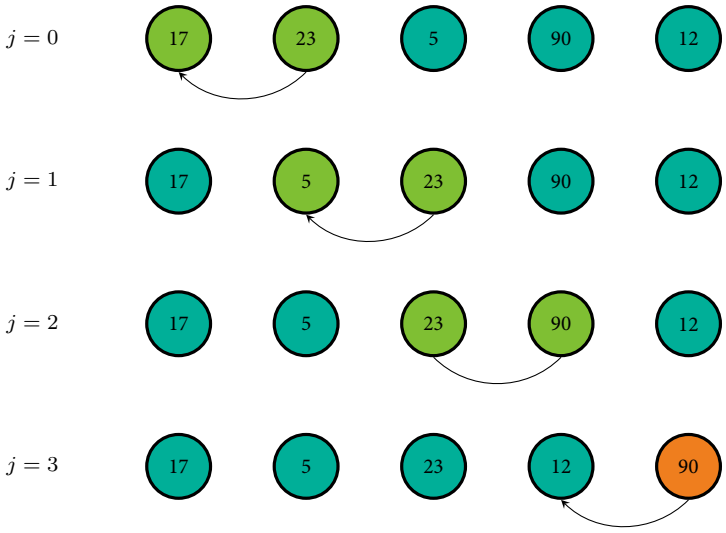
冒泡排序算法虽然简单，我们还是通过对 23、17、5、90、12 这一组数排序来说明该算法的原理。

i = 5[3] 时：

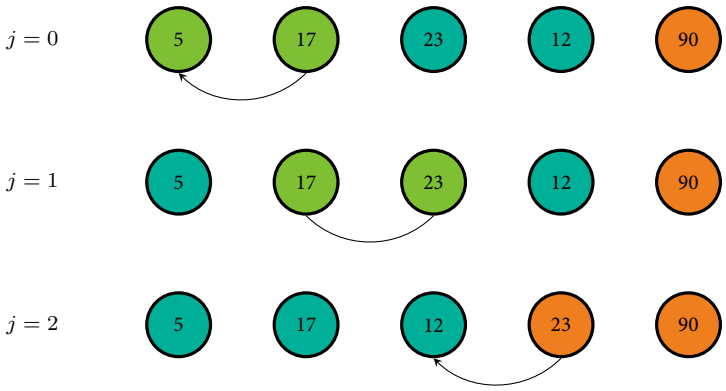


[3] 初始状态，实际从 i = 4 开始。

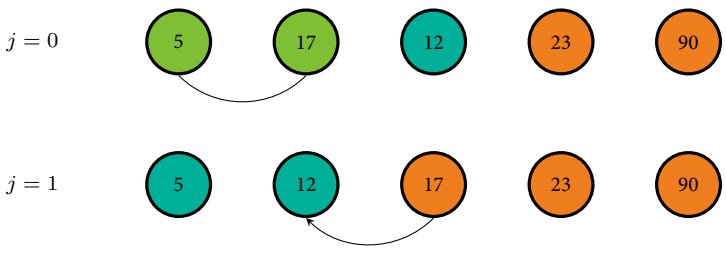
i = 4 时：



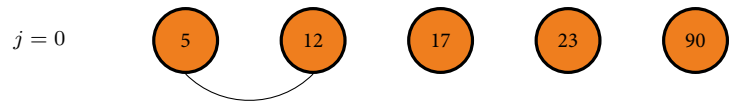
i = 3 时：



i = 2 时：



i = 1 时：



如果觉得如上排序过程不够形象，请移步这里，观看舞动的排序算法之冒泡排序。

冒泡排序是一种稳定的排序算法，前提是排序过程中不对相同的[4]元素元素作交换。最后同样给出冒泡排序的链表实现来作为本小结的结尾，见代码5。

```
/* with the bubble sort, after each iteration, the last element
    is on the right place */
void list_bubble_sort(list_t *list_head)
{
    if ((NULL == list_head) || (NULL == *list_head))
    {
        return;
    }

    node_t *c = NULL;
    node_t *p = NULL;
    node_t *n = NULL;
    node_t *e = NULL;

    while (e != (*list_head)->next)
    {
        p = c = *list_head;
        n = c ->next;
        while (e != n)
        {
            if (n->data < c->data)
            {
                if (p == c)
                {
                    c->next = n->next;
                    n->next = c;
                    *list_head = n;
                    p = n;
                    n = c->next;
                }
                else
                {
                    p->next = n;
                    c->next = n->next;
                    n->next = c;
                    p = n;
                    n = c->next;
```

[4] 即 array[j + 1] == array[j]

```
37                  }
38              }
39              else
40              {
41                  if (p == c)
42                  {
43                      c = n;
44                      n = c->next;
45                  }
46                  else
47                  {
48                      p = c;
49                      c = n;
50                      n = c->next;
51                  }
52              }
53          }
54
55          e = c;
56      }
57
58      return;
59  }
```

代码 5: 冒泡排序

1.4 选择排序

选择排序相对于冒泡排序的一个最大改进就是减少了排序过程中元素的移动，其原理如下：

首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

代码6给出了选择排序的C语言实现。

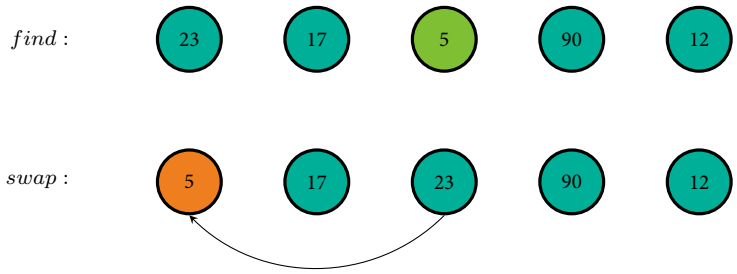
```
1  void selection_sort(int array[], int size)
2  {
3      int i;
4      int j;
5      int idx;
6      int temp;
7
8      for (i = 0; i < size - 1; i++)
```

```
9       {
10          idx = i;
11
12          for (j = i + 1; j < size; j++)
13          {
14              if (array[j] < array[idx])
15              {
16                  idx = j;
17              }
18          }
19
20          if (i != idx)
21          {
22              temp = array[i];
23              array[i] = array[idx];
24              array[idx] = temp;
25          }
26      }
27  }
```

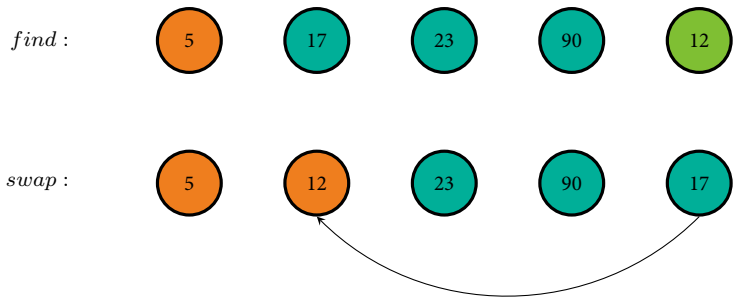
代码 6: 选择排序

从代码6中可以看出，选择排序和冒泡排序相同，其时间复杂度都是O(n^2)，排序过程都只需要一个元素的额外空间；但不同于冒泡排序，每次都交换相邻的满足条件的两个元素，插入排序每次遍历，都是找出最小（大）元素，一次循环结束后再判断是否需要交换。同样，我们以排序23、17、5、90、12这一组数来说明选择排序的原理。

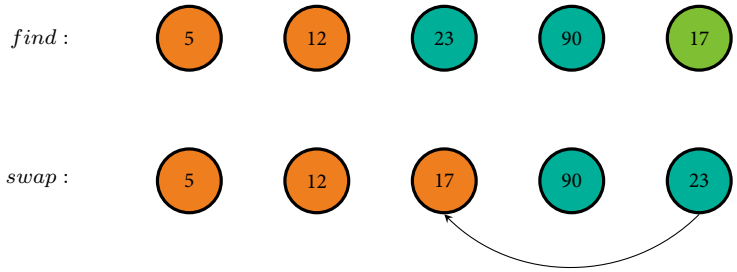
i = 0 时：



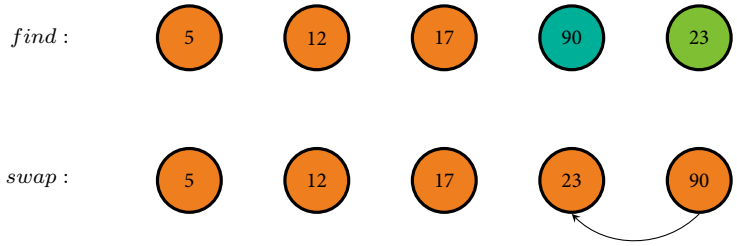
i = 1 时：



i = 2 时：



i = 3 时：



更加生动的排序演示，移步这里，观看舞动的排序算法之选择排序。

选择排序是不稳定的排序算法，如排序 48、38、65、97、76、13、27、48 这一组元素，排序完成后，48 这两个元素的顺序会发生交换。

1.5 希尔排序

希尔排序，又称递减增量排序算法，是一种改进的插入排序。其排序过程如下：

先取一个正整数$g_1 < n$[5]，把所有序号相隔g_1的元素作为一组进行直接插入排序；然后取$g_2 < g_1$，重复上述分组和排序操作，直至$g_i = 1$，即对所有记录进行一次插入排序。

代码7给出了希尔排序的C语言实现。

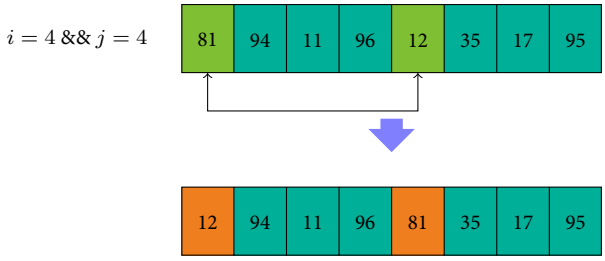
```c
void shell_sort(int array[], int size)
{
    int i = 0;
    int j = 0;
    int temp = 0;
    int step = 0;

    for (step = size/2; step > 0; step /= 2)
    {
        for (i = step; i < size; ++i)
        {
            for (j = i; j >= step; j -= step)
            {
                if (array[j] < array[j - step])
                {
                    temp = array[j];
                    array[j] = array[j - step];
                    array[j - step] = temp;
                }
                else
                {
                    break;
                }
            }
        }
    }
}
```

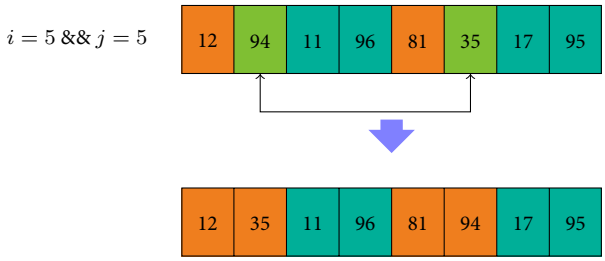
代码 7: 希尔排序

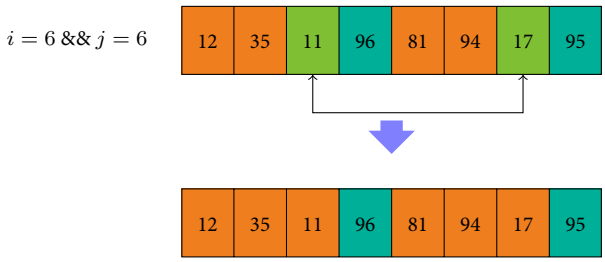
同样，我们以排序一组实际的元素为例来说明希尔排序的排序过程。这次排序的元素为 81、94、11、96、12、35、17、95。

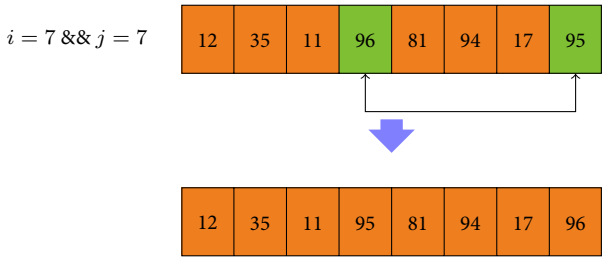
step = 4 时：

[5] n 为待排序元素的个数。

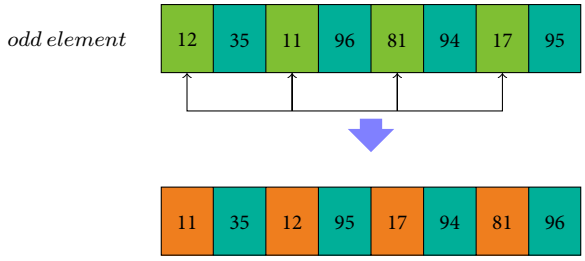

$i = 4 \&\& j = 4$
81 94 11 96 12 35 17 95
12 94 11 96 81 35 17 95

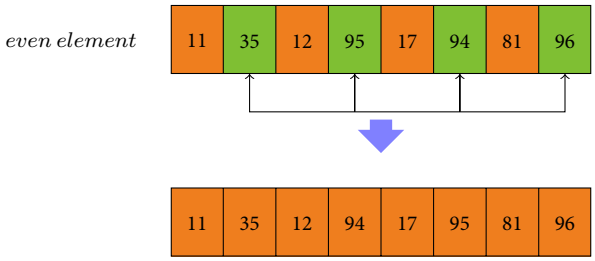

$i = 5 \&\& j = 5$
12 94 11 96 81 35 17 95
12 35 11 96 81 94 17 95


$i = 6 \&\& j = 6$
12 35 11 96 81 94 17 95
12 35 11 96 81 94 17 95

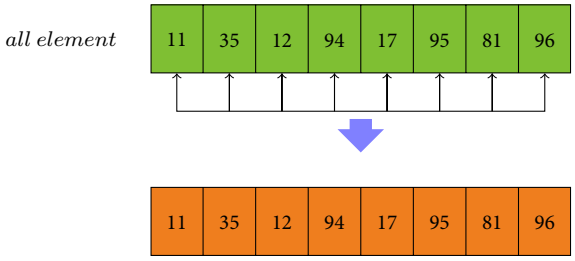

$i = 7 \&\& j = 7$
12 35 11 96 81 94 17 95
12 35 11 95 81 94 17 96

step = 2 时：





step = 1 时：



$step = 2$ 时，最终排序的效果等同于对所有奇数和所有的偶数进行的一次插入排序，但在程序的运行过程中，并不是先排序奇数元素，再排序偶数元素，整个过程和 $step = 4$ 时一样，但 $step = 2$ 和 $step = 1$ 只给出了一个最终排序的效果，没再一一列举。更生动的演示点击这里，观看舞动的排序算法之希尔排序。

影响希尔排序时间复杂度的一个关键因素是步长的选择。关于如何先取步长，请参考这里。

1.6 快速排序

1.7 堆排序

2 总结