目录

湖南工商大学 算法导论 课程教案

授课题目（教学章、节或主题 **课时安排: 2学时**

第七讲：线性时间排序、统计量 **授课时间 :第七周周一第1、2节**

教学内容（包括基本内容、重点、难点）：

基本内容： （1）基本概念：排序问题；

（2）线性时间排序：计数排序、基数排序、桶排序；

（3）统计量：最大最小值;

（4）选择排序算法.

教学重点、难点：重点为动态规划原理、递归算法设计

教学媒体的选择：本章使用大数据分析软件Jupyter教学，Jupyter集课件、Python程序运行、HTML网页制作、Pdf文档生成、Latex文档编译于一身，是算法导论课程教学的最佳选择。

板书设计：黑板分为上下两块，第一块基本定义，推导证明以及例子放在第二块。第一块 整个课堂不擦洗，以便学生随时看到算法流程图以及基本算法理论等内容。

课程过程设计： （1）讲解基本算法理论； （2）举例说明； （3）程序设计与编译； （4）对本课堂进行总 结、讨论； （5）布置作业与实验报告

第七讲 线性时间排序、统计量

一. 线性时间排序

对于比较排序来说，在排序的最终结果中，各元素的次序依赖于它们之间的比较。比较排序可以被抽象为一棵决策树。该模型证明了：对于包含n个元素的输入序列来说，在最坏情况下情况下，时间复杂度至少 $O(nlgn)$。

比较类排序时间复杂度：

- 冒泡排序：$O(n^2)$
- 快速排序：$O(n\lg n)$
- 选择排序： $O(n^2)$
- 直接插入排序：$O(n^2)$
- 希尔排序：$O(n^{1.3})$ $(n^{1.3} > n\lg n)$
- 归并排序：$O(n\lg n)$
- 堆排序：$O(n\lg n)$

排序算法	最坏情况	平均时间	空间需求
插入排序	$O(n^2)$	$\Theta(n^2)$	$O(1)$
归并排序	$O(n\lg n)$	$\Theta(n\lg n)$	$O(n)$
堆排序	$O(n\lg n)$	$\Theta(n\lg n)$	$O(1)$
快速排序	$O(n^2)$	$\Theta(n\lg n)$	$O(\lg n)$

我们可以较为清楚的看到各算法的时间复杂度至少为 $O(n\lg n)$。下面将证明对包含 n 个元素的输入序列来说，在最坏情况下，时间复杂度至少都是 $O(n\lg n)$。相对而言，堆排序是比较理想的排序算法。从算法的稳定性看，插入排序是稳定排序算法，快速排序和堆排序是不稳定算法。

1. 计数排序

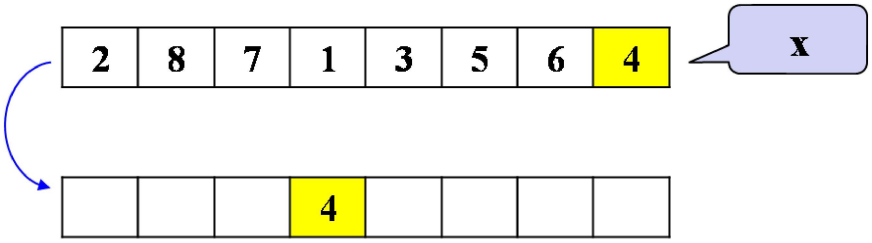
计数排序假设 n 个输入元素中的每一个都是在 0 到 k 区间内的一个整数，其中k为某个整数。当 $k = O(n)$ 时，排序的运行时间为 $O(n)$。

计数排序的思想是： 对每一个输入元素x，确定小于x的元素个数。利用这一信息，可以直接把x放到它在输出数组中的位置上了。（要注意有几个元素相同时的情况）

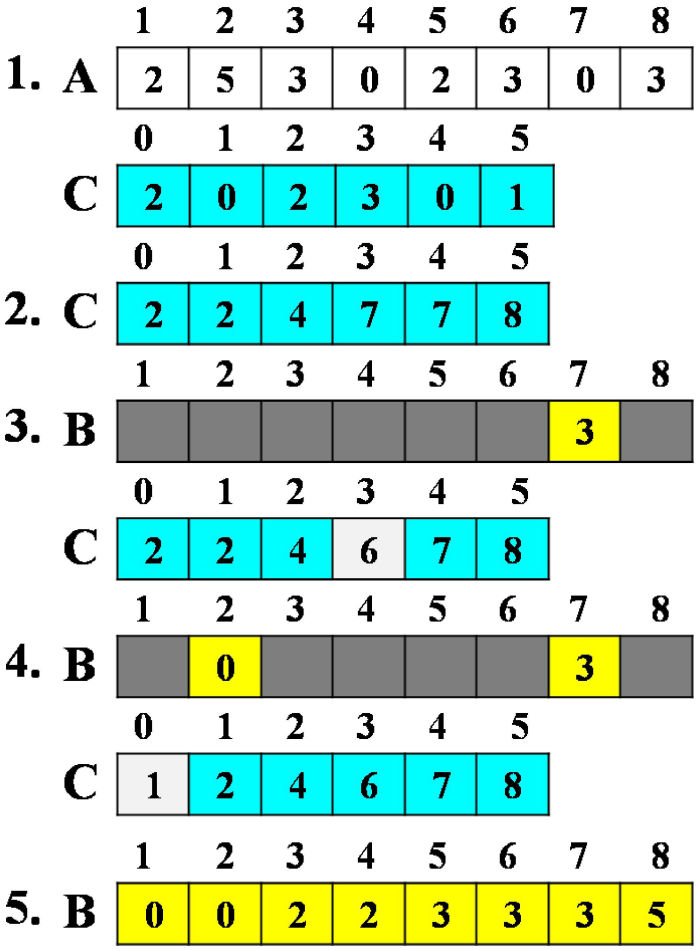
在计数排序算法代码中，假设输入是一个数组A[1…n]，A.length = n。我们还需要两个数组：B[1…n]存放排序数组和C[0…k]提供临时存储，也就是计数。

计数排序的基本步骤：

- 排位：对每一个输入的元素x,确定小于x的元素个数；
- 归位：将x放到输出数组的位置上。



计数排序的操作过程：



计数排序的程序实现：

- 计数排序的伪代码:

```
Counting-Sort(A, B, k)
Let C[0~k] be a new array
For i =0  to  k
     C(i) = 0
For j = 1  to A.length
     C[A[j]] = C[A[j]] + 1
For  I =1  to  k
C[i] = C[i] + C[i-1]
For  j = A.length  downto  1
     B[C[A[j]]] = A[j]
     C[A[j]] = C[A[j]] - 1
```

- 计数排序的C语言代码:

```
#include <istream>
#include <cstdio>
#include <algorithm>
using namespace std;
int c[1000]; //1000代表了a数组中元素的最大值
int b[1000];
int a[1000];
void countSort(int *a, int *b, int k, int n)
{   for(int i = 0; i <= k; i++)
  {  c[i] = 0;  }
  for(int j = 0; j < n; j++)
  {  c[a[j]] ++;    }
  for(int i = 1; i <= k; i++)
  {  c[i] = c[i]+ c[i-1];    }
  for(int j = n-1; j >= 0; j--)
  {  b[c[a[j]]] = a[j];
     c[a[j]] --;    }}
int main()
{    int n;
  int max_x = -100;
  scanf("%d", &n);
  for(int i = 0; i < n; i++)
  {    scanf("%d", &a[i]);
      if(a[i]>max_x)
      max_x = a[i];    }
  countSort(a, b, max_x, n);
  for(int i = 1; i <= n; i++)
  {    printf("%d ", b[i]);    }
  return 0;}
```

- 计数排序的Python代码:

In [7]:

```
import random
def countingSort(alist,k):
    n=len(alist)
    b=[0 for i in range(n)]
    c=[0 for i in range(k+1)]
    for i in alist:
        c[i]+=1
    for i in range(1,len(c)):
        c[i]=c[i-1]+c[i]
    for i in alist:
        b[c[i]-1]=i
        c[i]-=1
    return b
if __name__=='__main__':
    a=[random.randint(0,20) for i in range(20)]
    print(countingSort(a,20))
```

```
[0, 0, 1, 2, 4, 4, 5, 6, 9, 9, 9, 9, 9, 12, 12, 14, 14, 16, 18, 20]
```

计数排序算法分析：

我们从算法的过程来分析算法的时间代价，在第2-3行的for循环所花时间为 $O(k)$,第4-5行的for循环，所花的时间为 $O(n)$,第7-8行的for循环所花时间为 $O(k)$，第10—12行的for循环，所花时间 $O(n)$。这样总的时间代价就是 $O(n+k)$。在实际工作中，当 $k=O(n)$时，我们一般会采用计数排序，这好似的运行时间为 $O(n)$。

计数排序的一个重要性质就是它是稳定的：具有相同值得元素在输出数组中的相对次序与在们在输入数组中的相对次序相同。也就是说，对两个相同的数来说，在输入数组中先出现的数，在输出数组中也位于前面。通常，这种稳定性只有当进行排序的数据还附带卫星数据时才比较重要。计数排序的稳定性很重要的另一个原因是：计数排序通常会被用作基数排序的一个子过程。

2. 基数排序

基数排序(radix sort)用于对n张卡片上的d位数进行排序。与人们直观感受相悖的是，基数排序是先按最低有效位进行排序来解决卡片排序问题的。然后将所有卡片合成一叠，用同样的方法按次低有效位对所有卡片进行排序，并把排好的卡片再次合成一叠。重复这一过程d轮（d位数字）后，卡片可以排好序。

为了保证基数排序的稳定性，选择计数排序作为卡片的排序方法。

基数排序的代码是非常直观的，使用一个稳定排序方法从最低位开始到最高位进行排序。假设输入是一个数组A[1...n]，其中每一个元素都是一个d位数。第1位表示最低位，第d位表示最高位。

稍微修改了计数排序，数组B表示数组A中各元素的每一位，简化了计算每一轮排序后数组A的有效位。

给定n个d位数，其中每一个数位有k个不同的取值。如果基数排序使用的稳定排序方法（计数排序）耗时Θ(n+k)，那么它就可以在Θ(d(n+k))时间内将这些数排好序。当d为常数且k=Θ(n)时，基数排序具有线性的时间代价O(n)。

基数排序的Python代码：

In [8]:

```
class RadixSort:
    def sort(self, A, d):
        temp = A.copy()
        for i in range(d):
            temp = [t // 10 for t in temp]
            self.__count_sort(A, 9, temp)

    def __count_sort(self, A, k, B):
        ''' modified count_sort
        A and B have equal length. And B is used to compute every column of A.
        '''
        keys = [num % 10 for num in B]
        C = [0] * (k + 1)
        for j in range(len(keys)):
            # C[i] contains the number of elements equal to i
            C[keys[j]] = C[keys[j]] + 1
        for i in range(1, k+1):
            # C[i] contains the number of elements less than or equal to i
            C[i] = C[i] + C[i-1]
        # sort A and B
        tempA = [0] * len(A)
        tempB = [0] * len(B)
        for j in range(len(A)-1, -1, -1): # stable sort
            tempA[C[keys[j]]-1] = A[j]
            tempB[C[keys[j]]-1] = B[j]
            C[keys[j]] -= 1
        for i in range(len(A)):
            A[i] = tempA[i]
            B[i] = tempB[i]


def test():
    rs = RadixSort()
    A = [329, 457, 657, 839, 436, 720, 355] # P110例子
    rs.sort(A, 3)
    print(A)

if __name__ == '__main__':
    test()
```

```
[329, 355, 436, 457, 657, 720, 839]
```

3. 桶排序

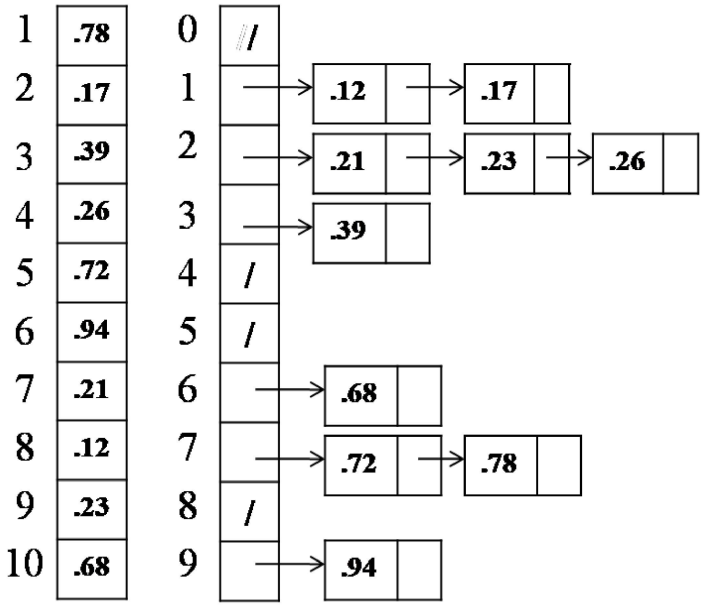
桶排序(bucket sort)假设输入数据服从均匀分布，平均情况下它的时间代价为 $O(n)$。与计数排序类似，因为对输入数据作了某种假设，桶排序的速度也很快。具体来说，计数排序假设输入数据都属于一个小区间内的整数，而桶排序则假设输入数据由一个随机过程产生，该过程将元素均匀、独立地分布在 $[0, 1)$ 区间内。

桶排序将 $[0, 1)$ 区间划分为n个相同大选哦的子区间，或称为桶。然后将n个数字分别放到各个桶中。因为输入数据均匀分布，所以一般不会出现很多数落在同一个桶中的情况。为了得到输出结果，我们先对每个桶中的数进行排序，然后遍历每个桶，按照次序把各个桶中的元素列出来即可。

在桶排序的代码中，我们假设输入是一个包含n个元素的数组A，每个元素位于 $[0, 1)$ 区间。此外，算法中需要一个临时数组B[0...n-1]来存放桶。桶可以用数组或链表来实现。

桶排序的期望运行时间是Θ(n)。即使输入数据不服从均匀分布，桶排序也仍然可以在线性时间内完成。只要输入数据满足下列性质：所有桶的大小的平方和与总的元素个数成线性关系。

桶排序操作过程：



桶排序的程序实现：

- 桶排序的伪代码：

```
BUCKET_SORT(A)
 n = length(A)
 for i= 1 to n
     do insert A[i] into list B
 for i=0 to n-1
     do sort list B[i] with insertion sort
 concatenate the list B[0]、B[1], , , B[n-1] together in order
```

- 桶排序的C语言代码:

```
#include <stdio.h>
void bucket_sort_normal(int source_array[], int source_array_length)
{    int i, j, k;
    // 1. 创建100个桶，并初始化为 0。
    int tmp_bucket[100] = {0};
    // 2. 将桶索引视为数组的“元素”，桶索引对应的值就是数组“该元素的个数”。
    for (i = 0; i < source_array_length; i++)
    {    // 比如， 若原数组是[4, 2, 1, 0]
        // 桶数组初始为 [0, 0, 0, 0, 0],桶的索引对应了原数组的数据区间
        // 当遍历原数组第一个元素4时，则桶数组[4] 加1，桶数组变为 [0, 0, 0, 0, 1]。用映射替代了比较，实现排序。
        tmp_bucket[source_array[i]]++;    }

    // 遍历桶数组(桶数组长度100)，改变原数组
    j = 0;
    for (i = 0; i < 100; i++)
    {    for (k = 0; k < tmp_bucket[i]; k++)
        {    source_array[j] = i;
            j++;   }    }  }

int main()
{    // 生成随机测试列表 0-99
    int test_list[20];
    int test_list_length = sizeof(test_list) / sizeof(int);

    printf("测试列表： \n");
    for (int i = 0; i < test_list_length; i++)
    {    test_list[i] = rand() % 100;
        printf("%d ", test_list[i]);    }
    printf("\n");

    // 普通桶排序
    bucket_sort_normal(test_list, test_list_length);
    printf("普通桶排序结果： \n");
    for (int i = 0; i < test_list_length; i++)
    {   printf("%d ", test_list[i]);    }
    printf("\n");

    return 0;  }
```

- 桶排序的Python代码：

In [9]:

```
class BucketSort:
    def sort(self, A):
        import math
        n = len(A)
        B = []
        for i in range(n):
            B.append([])
        for i in range(n):
            B[math.floor(n*A[i])].append(A[i])
        for i in range(n):
            B[i].sort() # 可以使用插入排序

        i = 0
        for j in range(n):
            for num in B[j]:
                A[i], i = num, i+1

def test():
    bs = BucketSort()
    A = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68] # P112例子
    bs.sort(A)
    print(A)

if __name__ == '__main__':
    test()
```

```
[0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0.94]
```

二. 中位数与顺序统计量

1. 基本概念

中位数：用非形式化的语言描述：中位数表示这样的一位数，它所属集合的“中点元素”。如果集合元素n为奇数，则中位数为(n+1)/2处；如果n为偶数，则中位数出现在n/2（下中位数）和n/2+1（上中位数）处，一般无特殊说明，我们都取下中位数。

顺序统计量：在一个n个元素组成的集合中，第i个顺序统计量是该集合中第i小的元素。

最大值：第1个顺序统计量。

最小值：第n个顺序统计量。

选择问题：给定一个包含n个元素的集合A和一个整数i，$1 \leq i \leq n$，我们需要得到一个整数x，其中有 $i-1$ 个元素小于它，即第i个顺序统计量。

这个问题最直观的解法是通过排序+索引的方式，但排序算法有多种，且时间复杂度略高。我们需要更低时间复杂度来解决这个问题，要求线性时间，即 $O(n)$。我们总结下算法导论上提出的方法，一步步展示如何 $O(n)$ 来解决这个问题。

2. 最小值和最大值

在一个有n个元素的集合中，通过n-1次比较可找到其中的最小值。

- 伪代码如下：

```
MINMUM(A)
  min = A[1]
  for i = 2 to A.length
      if min > A[i]
          min = A[i]
  return min
```

- Python代码如下：

In [ ]:

```
count = int(input('输入数据个数：\n'))
a = 1
while a <= count:
    num = int(input('请输入第{}个数:'.format(a))) #字符串中的方法
    if a == 1:                  #这句一定会执行，而且只执行一次，目的就是让你输入
        max = min = num         #第二个及以后的数都会走else,

    else:                       #第一次走else时，比较中的min和max都是你第一次输入
        if num < min:
            min = num
        elif num > max:
            max = num
    a += 1
print('最大数据是：', max)
print('最小数据是：', min)
```

时间复杂度分析：

当需要同时确定最大最小值时，如果利用上述方法对每个元素进行两次比较，则所需比较次数为2(n-1)。

一种最多需要 3⌊n/2⌋ 次比较就同时确定最大最小值的方法：首先将输入中的一对元素进行比较，然后将较小值与当前的最小值进行比较，将最大值与当前的最大值进行比较。这样，每两个元素仅需要进行三次比较。

3. 期望为线性时间的选择算法

一般选择问题看起来要比找最大、最小值要复杂得多，但令人惊奇的是，这两个问题的渐近运行时间却是相同的，都为$O(n)$。本节介绍的这个算法很强悍，期望的时间复杂度就能达到 $O(n)$，但最坏情况下的时间复杂度却为 $O(n^2)$。该算法采用的是快速排序章节中的Partition过程来得到划分的中点，如果该中点恰好等于选择的点，则即为所求，否则再在左右两个区间中用同样的方法再次寻找.

- 伪代码如下：

```
RANDOMIZED-SELECT(A, p, r, i)
if p = r
   then return A[p]
q = RANDOMIZED-PARTITION(A, p, r)
k = q - p + 1
if i = k       // the pivot value is the answer
   then return A[q]
elseif i < k
   then return RANDOMIZED-SELECT(A, p, q - 1, i)
else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

- Python代码如下：

In [4]:

```
import random
def randomized_select(A,p,r,i):
    if p==r:
        return A[p]
    q=randomized_partition(A,p,r)
    k=q-p+1
    if i==k:
        return A[q]
    elif i < k:
        return randomized_select(A,p,q-1,i)
    else:
        return randomized_select(A,q+1,r,i-k)

def randomized_partition(A,p,r):
    i = random.randint(p,r)
    t=A[r]
    A[r]=A[i]
    A[i]=t
    x=A[r]
    i=p-1
    for j in range(p,r):
        if A[j]<x:
            i+=1
            t=A[i]
            A[i]=A[j]
            A[j]=t
    A[r]=A[i+1]
    A[i+1]=x
    return i+1

A=[90,22,334,55,2,4,56,0,11,11,1]
result=randomized_select(A,0,10,9)
print(result)
```

56

4.简单选择排序

一般选择排序的基本思想：

第 i 趟排序从序列的后n-i+1 (i=1, 2, ..., n-1) 个元素中选择最小的元素，与该n-i+1个元素的最前面那个元素交换,也就是与第i个位置上的元素交换，直到i=n-1.

选择排序的操作过程：

简单选择排序的程序实现：

- 伪代码如下：

```
SelectSort (input ele[],input length)
  for i <- 1 to length step 1
      min <- i
      for j <- i+1 to length step 1
       if ele[j] < ele[min]
          min <- j
          end if
      swap(ele[j],ele[min])
  end
```

- C语言代码如下：

```
void selectsort( keytype k[ ], int n)
{  int  i, j, min;
 keytype  tmp;
 for(i=1; i<=n-1; i++) {
      min = I;
      for (j=i+1; j<=n; j++)
      if (k[j] < k[min])
      min = j
      if (min != i)  {
           tmp = k[min];
           k[min] = k[i];
           k[i] = tmp;  } }}
```

- Python代码如下：

In [6]:

```
def esay_select_sort(data):
    j=0
    while(j<len(data)-1):
        k=j+1
        temp=j
        while(k<len(data)):
            if data[k]<data[temp]:
                temp=k
            k+=1
        if(temp!=j):
            d=data[j]
            data[j]=data[temp]
            data[temp]=d
        j=j+1
data=[4,2,1,5,3]
esay_select_sort(data)
print(data)
```

```
[1, 2, 3, 4, 5]
```

一般选择算法的时间复杂度：

寻找第 i 个顺序统计量耗时 $O(n-i)$，一共要寻找 $n-1$ 次顺序统计量。

$$T(n) = \sum_{i=1}^{n-1} O(n-i) = O(n^2)$$

引用及参考：

[1]《Python数据结构与算法分析》布拉德利.米勒等著，人民邮电出版社，2019年9月.

[2] https://www.cnblogs.com/linxiyue/p/3555175.html (https://www.cnblogs.com/linxiyue/p/3555175.html)

[3] https://blog.csdn.net/lpy00511/article/details/52625481 (https://blog.csdn.net/lpy00511/article/details/52625481)

[4] https://blog.csdn.net/shengchaohua163/article/details/83444059 (https://blog.csdn.net/shengchaohua163/article/details/83444059)

[5] https://blog.csdn.net/zhang_xiaomeng/article/details/72680716 (https://blog.csdn.net/zhang_xiaomeng/article/details/72680716)

课后练习

1. 写出计数排序和桶排序的完整Python代码或 C语言代码。

2. 写出最大值最小值的完整Python代码或 C语言代码，并举例说明。

3. 分析各类算法的时间复杂度。

讨论、思考题、作业：

参考资料（含参考书、文献等）：算法导论. Thomas H. Cormen等，机械工业出版社，2017.

授课类型（请打√）：理论课□ 讨论课□ 实验课□ 练习课□ 其他□

教学过程设计（请打√）：复习□ 授新课□ 安排讨论□ 布置作业□

教学方式（请打√）：讲授□ 讨论□ 示教□ 指导□ 其他□

教学资源（请打√）：多媒体□ 模型□ 实物□ 挂图□ 音像□ 其他□

填表说明：1、每项页面大小可自行添减；2、教学内容与讨论、思考题、作业部分可合二为一。