经典排序算法分析和代码-下篇

这篇文档我们来讲两种非比较排序，计数排序，基数排序

计数排序

计数排序是一种非比较算法，其时间复杂度为 O(N+K)

举例说明

先用一个例子来说明计数排序算法，比如需要排序的集合为{1, 2, 1, 0, 4, 5}，在该集合中，最大的数值为5，那么通过遍历整个集合，

可以得到这样的数组

```
int counter[] = {1, 2, 1, 0, 1, 1}
```

0, 1, 2, 3, 4, 5

counter 数组描述了被排序数组里有1个0， 2个1， 1个2， 0个3， 1个4和1个5，当这个数组形成时，排序也就结束了。

代码设计

1）根据集合中最大的数值 K，来申请辅助空间 counter 数组

2）遍历被排序集合，讲计数记录到 counter 数组中

代码实现

[cpp] view plaincopy

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int data[] = {1, 2, 1, 0, 4, 5};

// 计数排序参数列表
// int d[] 为待排序数据列表
// int n 为待排序数据个数
// int min, max 为待排序数据中最大和最小的数，通过其计算待排数据跨度 K
void sort_counter(int d[], int n, int k)
{
    int i, j = 0;
    k++; // 实际申请空间比 K 大1
    // 申请辅助空间
    int* counter = malloc(sizeof(int)*k);
    memset(counter, 0, sizeof(int)*k);

    // 计数
    for(i=0; i<n; ++i)
    {
        ++counter[d[i]];
    }

    // 讲计数结果保存到待排数据空间
    for(i=0; i<k; ++i)
    {
        while(counter[i]-- > 0)
        {
            d[j++] = i;
        }
    }

    // 释放辅助空间
    free(counter);
}

int main(void)
{
    sort_counter(data, 6, 5);
    int i;
    for(i=0; i<6; ++i)
    {
        printf("%d\n", data[i]);
```

```
    }
    return 0;
}
```

特点

计数排序的特单是时间复杂度，与其他的排序算法不同，它的时间复杂度为 O(N+K)，这个时间复杂度表明了当 K 相对比较大时，不适合使用，比如对集合{1, 0, 100}

但是对于 N 远大于 K 的情况下，是相当适合的

基数排序

在数量大时，计数排序需要大量的辅助空间，并且时间复杂度有可能比较大，所以推出基数排序。

举例说明

假如有待排序数据 data[] = {312, 213, 545, 893};

先排序个位数：排序结果为 312, 213, 893, 545

再排序十位数：排序结果为 321, 213, 545, 893

再排序百位数：排序结果为 213, 312, 545, 893

完毕

对位数的排序，可以使用计数排序，速度快而且稳定。

代码设计

1）获取待排序数据中的最大位数

2）对位数进行循环，按位对待排序数据进行计数排序，并将其中间结果保存到临时空间

3）将临时数据保存到待排序数据，继续步骤2）

代码实现

[cpp] view plaincopy

```cpp
#include <stdio.h>

// int data[] = {1, 100, 321, 121, 333, 586, 1100};

// 该函数计算 data 中最大位数，本例子中，最大位数是1100，所以结果是4
int maxbit(int data[],int n)
{
    int d = 1; //保存最大的位数
    int p =10;
    for(int i = 0;i < n; ++i)
    {
        while(data[i] >= p)
        {
            p *= 10;
            ++d;
        }
    }
    return d;
}

// 基数排序
void sort_radix(int data[],int n)
{
    int d = maxbit(data, n);    // 计算位数
    int * tmp = new int[n];     // 中间变量，用来存储中间排序结果
    int * count = new int[10];  // 计数排序中的计数器
    int i,j,k;
    int radix = 1;

    // 根据最大位数进行循环，对没一位进行计数排序
```

```
78      for(i = 1; i<= d;i++)
79      {
80          // 初始化计数器
81          for(j = 0; j < 10; j++)
82              count[j] = 0;
83
84          // 对位数进行计数排序
85          for(j = 0; j < n; j++)
86          {
87              k = (data[j]/radix)%10; // 注意这里进行取模
88              count[k]++;             // 计数
89          }
90          for(j = 1; j < 10; j++)
91              count[j] = count[j-1] + count[j];
92
93          // 将排序中间结果保存到 tmp
94          for(j = n-1; j >= 0;j--)
95          {
96              k = (data[j]/radix)%10;
97              tmp[count[k]-1] = data[j];
98              count[k]--;
99          }
100
101         // 将中间结果保存到 data
102         for(j = 0;j < n;j++)
103             data[j] = tmp[j];
104
105         // 取模时的被除数，需要提高一位
106         radix = radix*10;
107     }
108     delete [] tmp;
109     delete [] count;
110 }
111
112 int main()
113 {
114     int data[] = {1, 100, 321, 121, 333, 586, 1100};
115     sort_radix(data, 7);
116     for(int i=0; i<7; i++)
117     {
118         printf("%d\n", data[i]);
119     }
120     return 0;
121 }
```

特点

基数排序比较适合对取值很大的数进行排序，也可用来对字符串进行排序。

但基数排序的缺点是不呈现时空局部性，因为在按位对每个数进行排序的过程中，一个数的位置可能发生巨大的变化，所以不能充分利用现代机器缓存提供的优势。同时计数排序作为中间稳定排序的话，不具有原地排序的特点，当内存容量比较宝贵的时候，还是有待商榷。