目 录

1 计数排序

1.1 比较计数

1.1.1 算法思路

针对待排序列表中的每一个元素，算出列表中小于该元素的元素个数，并把结果记录在一张表中，这个个数就指出了该元素在有序列表中的位置。

1.1.2 算法实现代码

```
1  void comparison_counting_sort(int *A, int len)
2  {
3      int i,j;
4      int* count = (int*)malloc(len * sizeof(int));
5      int* result = (int*)malloc(len * sizeof(int));
6      for(i = 0; i < len; ++i)
7          count[i] = 0;
8      for(i = 0; i < len; ++i)
9      {
10         for(j = i+1; j < len; ++j)
11         {
12             if(A[i] < A[j])
13                 ++count[j];
14             else
15                 ++count[i];
16         }
17     }
18     for(i = 0; i < len; ++i)
19         result[count[i]] = A[i];
20     for(i = 0; i < len; ++i)
21         A[i] = result[i];
22     free(count);
23     free(result);
24 }
```

1.1.3 算法分析

算法的基本操作为键值比较操作，基本操作的执行次数为

$$C(n) = \frac{(n-1)n}{2} \tag{1}$$

算法的键值比较次数和选择排序一样多，并且还占用了线性数量的额外空间，所以不推荐它来做实际的应用。

1.2 分布计数

当待排序的元素的值都来自于一个已知的小集合，如果元素的值是位于下界 l 和上界 u 之间的整数，我们可以计算每个这样的值出现的概率，然后把它们存储在数组 F 中。然后把有序列表的前 F[0] 个位置填入 l，接下来的 F[1] 个位置填入 l+1，以此类推。

1.2.1 算法代码实现

```
1  // A中元素的值是位于下界l和上界u之间的整数
2  // 分布计算相当于桶排序
3  void distribution_counting(int *A, int len)
4  {
5      int i,j;
6      int max = A[0], min = A[0];
7      int* bucket = (int*)malloc((max - min + 1) * sizeof(int));
8      int* result = (int*)malloc(len * sizeof(int));
9      for(i = 0; i < len; ++i)
10     {
11         if(max < A[i])
12             max = A[i];
13         if(min > A[i])
14             min = A[i];
15     }
16     for(i = 0; i < max-min+1; ++i)
17         bucket[i] = 0;
18     for(i = 0; i < len; ++i)
19         bucket[A[i] - min] = bucket[A[i] - min] + 1;
20     for(i = 1; i < max-min+1; ++i)
21         bucket[i] = bucket[i-1] + bucket[i];
22     for(i = len-1; i >= 0; --i)
23     {
24         j = A[i] - min;
25         result[bucket[j] - 1] = A[i];
26         --bucket[j];
27     }
28     free(bucket);
29     free(result);
30 }
```

2 串匹配中的输入增强技术

3 散列法

4 B 树