杭州電子科技大学
HANGZHOU DIANZI UNIVERSITY

程序设计实践报告

基于 C 语言的非递归快速排序解题报告

卓越学院
计算机科学英才班
周靖凯

目录

1 引言

快速排序算法，由 C. A. R. Hoare 于 1960 年提出，是计算机科学中最高效和广泛使用的排序算法之一。以其优异的平均时间复杂度——$O(n \log n)$，它在处理大规模数据集时表现出色。然而，传统的快速排序算法通常通过递归实现，这在处理极大数据集时可能导致栈溢出的问题。

为此，本报告采用了非递归方法实现快速排序，有效避免了递归实现中的栈溢出风险。非递归实现通过使用栈结构来模拟递归调用的过程，既保持了算法的核心效率，又提高了其在处理大数据集时的稳定性。接下来，本文将详细介绍非递归快速排序算法的设计、实现，以及性能分析。

2 基本原理与实现

2.1 过程

快速排序的工作原理是通过分治的方式来将一个数组排序。

快速排序分为三个过程：

1. 将数列划分为两部分（要求保证相对大小关系）；
2. 递归到两个子序列中分别进行快速排序；
3. 不用合并，因为此时数列已经完全有序。

和归并排序不同，第一步并不是直接分成前后两个序列，而是在分的过程中要保证相对大小关系。具体来说，第一步要是要把数列分成两个部分，然后保证前一个子数列中的数都小于后一个子数列中的数。为了保证平均时间复杂度，一般是随机选择一个数 m 来当做两个子数列的分界。

之后，维护一前一后两个指针 p 和 q，依次考虑当前的数是否放在了应该放的位置（前还是后）。如果当前的数没放对，比如说如果后面的指针 q 遇到了一个比 m 小的数，那么可以交换 p 和 q 位置上的数，再把 p 向后移一位。当前的数的位置全放对后，再移动指针继续处理，直到两个指针相遇。

其实，快速排序没有指定应如何具体实现第一步，不论是选择 m 的过程还是划分的过程，都有不止一种实现方法。

第三步中的序列已经分别有序且第一个序列中的数都小于第二个数，所以直接拼接起来就好了。

2.2 稳定性

快速排序是一种不稳定的排序算法。

2.3 时间复杂度

快速排序的最优时间复杂度和平均时间复杂度为 $O(n \log n)$，最坏时间复杂度为 $O(n^2)$。

2.4 三路快速排序

三路快速排序（英语：3-way Radix Quicksort）是快速排序和基数排序的混合。它的算法思想基于荷兰国旗问题的解法。

与原始的快速排序不同，三路快速排序在随机选取分界点 m 后，将待排数列划分为三个部分：小于 m、等于 m 以及大于 m。这样做即实现了将与分界元素相等的元素聚集在分界元素周围这一效果。

2.5 递归实现

```
void quick_sort(int a[], const int n)
{
    if (n <= 1)
        return;
    int pivot = a[rand() % n];
    int i = 0, j = 0, k = n;
    while (i < k)
    {
        if (a[i] < pivot)
            swap(a[i++], a[j++]);
        else if (pivot < a[i])
            swap(a[i], a[--k]);
        else
            i++;
    }
    quick_sort(a, j);
    quick_sort(a + k, n - k);
    return;
}
```

2.6 非递归实现

尽管传统的快速排序通常采用递归来实现，但递归方法在处理极大数据集时可能遇到栈溢出的问题。为解决这一问题，非递归实现采用了栈来模拟递归过程，从而避免了递归带来的栈溢出风险。非递归快速排序在逻辑上保持了与递归版本相同的分区过程，但在执行方式上，它通过显式地使用栈来管理分区步骤的序列，从而实现了算法的迭代版本。

```
typedef struct
{
    int start, end;
} Range;

void quick_sort_stack(int a[], const int n)
{
    if (n <= 0)
        return;
    Range *r = (Range *)malloc(sizeof(Range) * n);
    int p = 0;
    r[p++] = (Range){0, n - 1};
    while (p)
    {
        Range range = r[--p];
        int len = range.end - range.start + 1;
        int pivot = a[rand() % len + range.start];
        int i = range.start, j = range.start, k = range.
           end;
        while (i <= k)
        {
            if (a[i] < pivot)
            {
                int t = a[i];
                a[i] = a[j];
```

```
                a[j] = t;
                i++, j++;
            }
            else if (pivot < a[i])
            {
                int t = a[i];
                a[i] = a[k];
                a[k] = t;
                k--;
            }
            else
                i++;
        }
        if (range.start < j - 1)
            r[p++] = (Range){range.start, j - 1};
        if (k + 1 < range.end)
            r[p++] = (Range){k + 1, range.end};
    }
    free(r);
    return;
}
```

3 程序设计

3.1 数据结构

程序中实现了一个简单但高效的栈结构，用于替代递归调用。栈由若干个节点组成，每个节点包含一个范围（`struct Range`），表示待排序数组的一部分。这个范围结构由两个整型字段组成：`start` 和 `end`，它们分别标记了数组中需要排序的段的起始和结束位置。

3.2 核心函数

程序的核心是 `quick_sort` 函数，它使用上述栈来实现快速排序的非递归版本。首先，函数将整个数组的范围压入栈中。然后，在栈不为空的情况下，它连续弹出范围并对其进行排序。排序过程中，选取范围内的随机元素作为枢纽，并基于这个枢纽将范围内的元素分为小于、等于和大于枢纽的三部分。每次分区后，将产生的新范围压回栈中，直到栈为空，表明排序完成。

4 算法实现

4.1 分区策略

算法的核心在于其分区操作，这是快速排序的关键步骤。在每次处理栈中的一个范围时，算法首先选择一个随机枢纽（`pivot`）。这一策略有助于减少对输入数据分布的依赖，从而在平均情况下保持良好的性能。随后，算法将范围内的元素根据它们与枢纽的比较结果进行重新排列：小于枢纽的元素移至其左侧，大于枢纽的元素则移至右侧。

4.2 循环控制

在传统的递归快速排序中，每一次分区都伴随着新的递归调用。而在非递归实现中，这一过程通过显式的循环控制来实现。使用栈结构存储每个分区操作后产生的新范围。当栈不为空时，算法持续弹出栈顶元素（即当前处理的范围），执行分区操作，然后根据分区结果将新的范围压入栈中。这种方法确保了算法能够处理整个数据集，同时避免了递归调用的栈溢出风险。

5 测试与结果

5.1 测试方法

测试分为两部分：正确性验证和性能评估。正确性验证通过将算法的输出与标准排序函数的输出进行比较来进行。这包括了对多种不同大小和特性的数据集（如随机数据、几乎升序数据和几乎逆序数据）的测试。性能评估则通过测量算法在不同大小的数据集上的运行时间来进行，特别关注其在大数据集上的表现。

5.2 测试结果

5.2.1 无序序列

运行时间（秒）	$n = 10^5$	$n = 10^6$	$n = 10^7$
递归实现	0.005587	0.060396	0.697338
非递归实现	0.005690	0.059575	0.665750

5.2.2 几乎升序序列

运行时间（秒）	$n = 10^5$	$n = 10^6$	$n = 10^7$
递归实现	0.003163	0.029129	0.337156
非递归实现	0.003154	0.030876	0.317150

5.2.3 几乎降序序列

运行时间（秒）	$n = 10^5$	$n = 10^6$	$n = 10^7$
递归实现	0.003090	0.030066	0.319865
非递归实现	0.003022	0.028127	0.320204

测试结果表明，非递归快速排序算法在所有测试数据集上均正确地完成了排序任务。在性能方面，算法显示出与递归版本相似的时间效率，特别是在大数据集上，其表现优于或等同于传统的递归实现。这些结果验证了非递归快速排序作为一种有效且可靠的排序方法的实用性。

5.3 性能分析

5.3.1 时间复杂度

在平均情况下，快速排序的时间复杂度为 $O(n \log n)$，这归因于每次分区操作大约将数据集分为两等分，并递归地应用于这些子集。非递归版本维持了这一性能特点，尽管实现方式由递归转变为使用栈进行迭代。对于普通快速排序，在最坏情况下，当数据已经是有序的或接近有序时，时间复杂度会达到 $O(n^2)$。然而，通过随机选择枢纽，算法的平均性能得以优化，这一策略有助于减少对输入数据分布的依赖。

5.3.2 空间复杂度

非递归快速排序的主要空间开销来自于栈的使用。在最坏的情况下，栈的大小可能需要与输入数组的大小成比例，导致空间复杂度达到 $O(n)$。然而，在平均情况下，由于分区操作的效率，栈的大小通常远小于数组的大小，使得实际的空间开销保持在较低水平。

5.3.3 优化可能性

尽管非递归快速排序已经非常高效，但仍有优化空间。例如，改进枢纽选择算法以进一步减少对特定数据分布的依赖，或采用更复杂的数据结构来降低在最坏情况下的空间需求。此外，对于小范围的数据集，可以考虑使用插入排序或其他更适合小数据集的排序算法来优化性能。

6　小结

本报告详细介绍了非递归快速排序算法的设计、实现和性能分析。通过深入探讨，我们可以得出以下几点结论：

非递归实现的快速排序算法在保持了传统快速排序高效性能的同时，成功避免了递归实现可能导致的栈溢出问题。这一点在处理大规模数据集时尤为重要，增强了算法的实用性和稳定性。

非递归实现在空间复杂度方面可能略高于递归版本，但它提供了更高的可靠性，尤其是在系统栈资源有限的环境中。此外，通过采用随机枢纽选择策略，算法成功地减少了对输入数据分布的敏感性，从而提高了其在各种数据集上的平均性能。

综上所述，非递归快速排序算法不仅是一种高效的排序方法，而且是一种适应性强、稳定可靠的解决方案，适用于各种规模的数据排序需求。

A 附录

A.1 完整测试代码

test.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>

typedef struct
{
    int start, end;
} Range;

void quick_sort_stack(int a[], const int n)
{
    if (n <= 0)
        return;
    Range *r = (Range *)malloc(sizeof(Range) * n);
    int p = 0;
    r[p++] = (Range){0, n - 1};
    while (p)
    {
        Range range = r[--p];
        int len = range.end - range.start + 1;
        int pivot = a[rand() % len + range.start];
        int i = range.start, j = range.start, k = range.
           end;
        while (i <= k)
        {
            if (a[i] < pivot)
```

```
            {
                int t=a[i];
                a[i]=a[j];
                a[j]=t;
                i++,j++;
            }
            else if (pivot < a[i])
            {
                int t=a[i];
                a[i]=a[k];
                a[k]=t;
                k--;
            }
            else
                i++;
        }
        if (range.start < j - 1)
            r[p++] = (Range){range.start, j - 1};
        if (k + 1 < range.end)
            r[p++] = (Range){k + 1, range.end};
    }
    free(r);
    return;
}

void quick_sort(int a[], const int n)
{
    if (n <= 1)
        return;
    int pivot = a[rand() % n];
    int i = 0, j = 0, k = n;
```

```
58      while (i < k)
59      {
60          if (a[i] < pivot)
61          {
62              int t = a[i];
63              a[i] = a[j];
64              a[j] = t;
65              i++, j++;
66          }
67          else if (pivot < a[i])
68          {
69              k--;
70              int t = a[i];
71              a[i] = a[k];
72              a[k] = t;
73          }
74          else
75              i++;
76      }
77      quick_sort(a, j);
78      quick_sort(a + k, n - k);
79      return;
80  }
81
82  bool is_sorted(int a[], int n)
83  {
84      for (int i = 1; i < n; i++)
85          if (a[i] < a[i - 1])
86              return false;
87      return true;
88  }
```

```
89  int cmp(const void *a, const void *b)
90  {
91      return (*(int *)a - *(int *)b);
92  }
93
94  void generate(int *a, int *b, int *c, int n)
95  {
96      for (int i = 0; i < n; i++)
97          a[i] = b[i] = rand();
98      qsort(b, n, sizeof(int), cmp);
99      for (int i = 0; i < n; i++)
100         c[i] = b[n - i - 1];
101     int times = 10;
102     for (int i = 1; i <= times; i++)
103     {
104         int l = rand() % n, r = rand() % n;
105         while (l == r)
106             l = rand() % n, r = rand() % n;
107         int t = b[l];
108         b[l] = b[r];
109         b[r] = t;
110     }
111     for (int i = 1; i <= times; i++)
112     {
113         int l = rand() % n, r = rand() % n;
114         while (l == r)
115             l = rand() % n, r = rand() % n;
116         int t = c[l];
117         c[l] = c[r];
118         c[r] = t;
119     }
```

```
120     return;
121 }
122 
123 void test(int a[], int n, void (*sort)(int *, int), const
       char type[], const char name[])
124 {
125     int *c = (int *)malloc(sizeof(int) * n);
126     for (int i = 0; i < n; i++)
127         c[i] = a[i];
128     int st = clock();
129     sort(c, n);
130     int ed = clock();
131     double time = (double)(ed - st) / CLOCKS_PER_SEC;
132     printf("%20s | %25s | time: %.6lf\n", type, name, time
           );
133     if (!is_sorted(c, n))
134     {
135         fprintf(stderr, "Error\n");
136         exit(1);
137     }
138     free(c);
139     return;
140 }
141 
142 int main(int argc, char *argv[])
143 {
144     if (argc < 3)
145     {
146         fprintf(stderr, "Need more parameters");
147         return 1;
148     }
```

```
149     int n = atoi(argv[1]), seed = atoi(argv[2]);
150     printf("n = %d, seed = %d\n", n, seed);
151     srand(seed);
152     int *a = (int *)malloc(sizeof(int) * n), *b = (int *)
            malloc(sizeof(int) * n), *c = (int *)malloc(sizeof(
            int) * n);
153     generate(a, b, c, n);
154     test(a, n, quick_sort, "Quick Sort", "Random Sequence"
            );
155     test(b, n, quick_sort, "Quick Sort", "Almost Ascending
             Sequence");
156     test(c, n, quick_sort, "Quick Sort", "Almost
            Descending Sequence");
157     test(a, n, quick_sort_stack, "Quick Sort Stack", "
            Random Sequence");
158     test(b, n, quick_sort_stack, "Quick Sort Stack", "
            Almost Ascending Sequence");
159     test(c, n, quick_sort_stack, "Quick Sort Stack", "
            Almost Descending Sequence");
160     return 0;
161 }
```