南京郵電大學
Nanjing University of Posts and Telecommunications

《算法设计与分析》课内实验报告

实验题目：分治法

实验日期：2023 年 10 月 28 日

指导教师：余亮

学生姓名：单家俊

学生学号：B21080526

实验成绩：

一、实验目的

理解分治法的算法思想，清楚两路合并排序和快速排序算法的基本原理和实施过程，能将输入的一组无序序列排列为有序序列后输出。比较两路和并排序算法和快速排序算法的时间复杂度及算法的稳定性。

二、实验内容

用分治法实现一组无序序列的两路合并排序和快速排序。

三、实验过程描述

算法思想：

分治法（Divide and Conquer）是一种算法设计策略，它将一个问题分解成更小的子问题，然后递归地解决这些子问题，最后将它们的解合并起来，得到原始问题的解。该策略通常包括三个步骤：

分解（Divide）：将原始问题分解成若干个规模较小但结构与原问题相似的子问题。这一步是问题规模的减小，通常通过将问题划分成两个或更多的子问题。

解决（Conquer）： 递归地解决这些子问题。当子问题的规模足够小，可以直接求解时，这一步直接给出答案。否则，继续分解。

合并（Combine）： 将子问题的解合并成原始问题的解。这一步是将各个子问题的解合并起来，构建出原问题的解。

计算时间复杂度：

合并排序的时间复杂度：

分解： 将数组分成两半，递归地对两个子数组进行排序。解决： 递归地对两个子数组进行排序，直到数组的大小为 1。合并： 将两个有序的子数组合并成一个有序数组。

在每一层递归中，都需要 O(n) 的时间进行合并。由于递归树的高度是 log n，所以总的时间复杂度是 O(n log n)。

快速排序的时间复杂度：

分解： 选择一个基准元素，将数组分成两个子数组，小于基准的放在左边，大于基准的放在右边。解决： 递归地对两个子数组进行排序。合并： 不需要显式的合并操作，因为通过分解和解决步骤已经完成了排序。

在每一层递归中，都需要 O(n) 的时间来进行分割。由于递归树的平均深度是 log n，所以总的时间复杂度是 O(n log n)。但是在最坏情况下（每次划分选择的基准元素总是导致一个空子数组），时间复杂度可能达到 O(n^2)。

代码：

```cpp
#include <iostream>
#include <stdlib.h>
#include <ctime>

#define MaxArray 10000

// 生成最好情况下的数据（已排序）
void generateBestCaseData(int arr[], int n)
{
    for (int i = 0; i < n; i++) {
        arr[i] = i;
    }
}

// 生成最坏情况下的数据（逆序）
void generateWorstCaseData(int arr[], int n)
{
    for (int i = 0; i < n; i++) {
        arr[i] = n - i;
    }
}
void Merge(int A[], int left, int mid, int right)
{
    int* B = new int[right - left + 1];
    int i = left;
    int j = mid + 1;
    int k = 0;

    while (i <= mid && j <= right)
    {
        if (A[i] <= A[j])
            B[k++] = A[i++];
        else
            B[k++] = A[j++];
    }

    while (i <= mid)
        B[k++] = A[i++];

    while (j <= right)
        B[k++] = A[j++];

    for (i = left, k = 0; i <= right; )
        A[i++] = B[k++];
```

```
    delete[] B;
}


void MergeSort(int A[], int left, int right)
{
    if (left < right)
    {
        int mid;
        mid = (left + right) / 2;
        MergeSort(A, left, mid);
        MergeSort(A, mid + 1, right);
        Merge(A, left, mid, right);
    }
}


void Swap(int* i, int* j)
{
    int temp = *i;
    *i = *j;
    *j = temp;
}

int Partition(int* arr, int left, int right) {
    int i = left, j = right + 1;
    do {
        do i++; while (arr[i] < arr[left]);
        do j--; while (arr[j] > arr[left]);
        if (i < j) Swap(&arr[i], &arr[j]);
    } while (i < j);
    Swap(&arr[left], &arr[j]);
    return j;
}

void QuickSort(int* arr, int left, int right) {
    if (left < right) {
        int j = Partition(arr, left, right);
        QuickSort(arr, left, j - 1);
        QuickSort(arr, j + 1, right);
    }

}
```

```cpp
int main()
{
    static double elapsed_time_Average = 0;
    double elapsed_time_Best = 0;
    double elapsed_time_Worst = 0;

    static double Quick_elapsed_time_Average = 0;
    double Quick_elapsed_time_Best = 0;
    double Quick_elapsed_time_Worst = 0;

    int i,j;

    int num[MaxArray];
    int bestnum[MaxArray];
    int worstnum[MaxArray];

    int length = sizeof(num) / sizeof(num[0]);
    clock_t start, end;
    srand((unsigned int)time(0));

    generateBestCaseData(bestnum, MaxArray);
    generateWorstCaseData(worstnum, MaxArray);
//    for ( j = 0; j < length; j++)
//    {
//        num[j] = rand() % 10000 + 1;
//    }
//    std::cout << "生成" << length << "个元素" << std::endl;
//    generateBestCaseData(bestnum, MaxArray);
//    generateWorstCaseData(worstnum, MaxArray);

//    合并排序：生成 20 个数组进行排序获得平均情况下的时间
    for(i = 0;i < 20;i++)
    {
        for ( j = 0; j < length; j++)
        {
            num[j] = rand() % 10000 + 1;
        }
//        std::cout << "生成第" << i+1 << "组数据" << std::endl;
        start = clock();
        MergeSort(num, 0, length -1);
        end = clock();
        elapsed_time_Average += double(end - start) / CLOCKS_PER_SEC
```

```
* 1000;
    }

//   合并排序：生成 20 个数组进行排序获得平均情况下的时间
    for(i = 0;i < 20;i++)
    {
        for ( j = 0; j < length; j++)
        {
            num[j] = rand() % 10000 + 1;
        }
//        std::cout << "生成第" << i+1 << "组数据" << std::endl;
        start = clock();
        QuickSort(num, 0, length -1);
        end = clock();
        Quick_elapsed_time_Average += double(end - start) /
CLOCKS_PER_SEC * 1000;
    }

//合并排序最好情况
    start = clock();
    MergeSort(bestnum, 0, length - 1);
    end = clock();
    elapsed_time_Best = double(end - start) / CLOCKS_PER_SEC * 1000;

//快速排序最好情况
    start = clock();
    QuickSort(bestnum, 0, length - 1);
    end = clock();
    Quick_elapsed_time_Best = double(end - start) / CLOCKS_PER_SEC *
1000;

//合并排序最坏情况
    start = clock();
    MergeSort(worstnum, 0, length - 1);
    end = clock();
    elapsed_time_Worst = double(end - start) / CLOCKS_PER_SEC * 1000;

//快速排序最坏情况
    start = clock();
    QuickSort(worstnum, 0, length - 1);
    end = clock();
    Quick_elapsed_time_Worst = double(end - start) / CLOCKS_PER_SEC
* 1000;
```

```
    std::cout <<"合并排序："<<std::endl;
    std::cout << "Average Case Time: " << elapsed_time_Average/20 <<
" ms" << std::endl;
    std::cout << "Best Case Time: " << elapsed_time_Best << " ms" <<
std::endl;
    std::cout << "Worst Case Time: " << elapsed_time_Worst << " ms"
<< std::endl;

    std::cout <<"快速排序："<<std::endl;
    std::cout << "Average Case Time: " <<
Quick_elapsed_time_Average/20 << " ms" << std::endl;
    std::cout << "Best Case Time: " << Quick_elapsed_time_Best << "
ms" << std::endl;
    std::cout << "Worst Case Time: " << Quick_elapsed_time_Worst <<
" ms" << std::endl;
    return 0;
}
```

四、实验结果

合并排序：

100000 个数据：

```
Average Case Time: 23.65 ms
Best Case Time: 16.6 ms
Worst Case Time: 17.9 ms
```

10000 个数据：

```
Average Case Time: 2.65 ms
Best Case Time: 2.3 ms
Worst Case Time: 3.6 ms
```

快速排序：

1000 个数据：

```
Average Case Time: 0.2 ms
Best Case Time: 0 ms
Worst Case Time: 0 ms
```

10000 个数据：

```
Average Case Time: 1.6 ms
Best Case Time: 78 ms
Worst Case Time: 94 ms
```

五、实验小结

在本次算法实验中，我学习和实践了分治法。以下是我对实验的一些心得体会：

分治法的应用：通过实现两路合并排序和快速排序，我深入理解了分治法的思想和应用。分治法将大问题划分为更小的子问题，然后通过解决子问题来解决整个大问题。这种分而治之的策略有助于提高算法的效率和可读性。

排序算法的性能比较：通过对比两路合并排序和快速排序，我观察到它们在不同场景下的性能差异。两路合并排序具有稳定的时间复杂度（O(nlogn)），适用于大规模数据的排序；而快速排序在平均情况下具有较好的性能，但在最坏情况下可能导致时间复杂度达到O(n^2)。

问题解决的思路：在实验过程中，我遇到了一些问题，例如算法实现错误、时间复杂度过高和结果不符预期等。为了解决这些问题，我采取了仔细检查代码、调试和输出调试信息等方法。这个过程锻炼了我的问题解决能力和调试技巧，并教会了我在面对困难时保持耐心和坚持不懈。

实验收获与思考：通过这次算法实验，我不仅加深了对分治法的理解，还提升了编程能力和算法设计的思维方式。我学会了分析和比较不同算法的优劣，根据问题的特性选择合适的算法。同时，我也意识到算法的正确性、效率和可维护性对实际应用的重要性。

总之，本次算法实验让我对分治法有了更深入的了解，同时培养了我解决问题和优化算法的能力。