算法设计与分析实验报告

一、 实验题目

对 n 个整数使用归并排序和快速排序进行升序排序。

输入文件分别为 mergesort.in 和 quicksort.in，输入有两行，第一行为一个整数 n，第二行包含 n 个整数 n_i，每两个整数用空格隔开。

输出文件名分别为 mergesort.out 和 quicksort.out，输出共一行。包含 n 个整数 n_i，为排序后的升序序列，每两个整数用空格隔开。

二、 实验过程

2.1 归并排序

归并排序算法使用分治策略实现对 n 个元素进行排序的算法，其基本思想是：将待排序序列分为大小大致相同的两个子集合，分别对两个子集合进行排序，最终将排好序的子集合合并成要求的排好序的集合。

本人一共实现了三个版本的归并排序，分别解决了不同的问题。

第一个版本为递归版本的归并排序，优点是结构清晰、可读性强，但耗时和占用空间都大不如非递归版本。以下为递归版本代码实现：

```
template<class Type>
void RecursiveMergeSort(Type lst1[], Type lst2[], const int& left, const int&
right)
{
    if (left < right)
    {
        int middle = (left + right) / 2;
        RecursiveMergeSort(lst1, lst2, left, middle);
        RecursiveMergeSort(lst1, lst2, middle + 1, right);
        Merge(lst2, lst1, left, middle, right);
        for (int i = left; i <= right; i++)
            lst1[i] = lst2[i];
    }
}
template<class Type>
void MergeSort(Type lst[], const int& n)
{
    Type* temp = new Type[n];  //避免每递归调用一次，申请一次辅助数组
    RecursiveMergeSort(lst, temp, 0, n - 1);
    delete(temp);
```

```
}
```

需要注意的一点是辅助数组的创建应在递归函数外进行，避免每调用一次递归申请创建一次辅助数组，造成不必要的资源浪费。因此，可以在递归函数外再分离出一层函数用于申请辅助空间。

RecursiveMergeSort 函数中对自身的两次调用，分别需要 T(n/2)的时间完成，后面的 Merge 函数和迭代复制无论是倒序序列还是正序序列都需要 O(n)时间完成，因此在最坏、平均和最好的情况下所需的计算时间均为

$$T(n) = 2T(n/2) + O(n)$$

解递归方程可得 T(n) = O(nlogn)。

由于使用大小为 n 的辅助数组，因此空间复杂度为 O(n).

第二个版本为迭代版本，相比于递归版本消耗的时间和栈空间都更小，但代码会更加复杂。具体实现为先将数组 a 中相邻元素两两配对，再用合并算法将他们排序。以下为迭代版本代码实现：

```
template<class Type>
void MergePass(Type lst1[], const Type lst2[], const int& s, const int& n)
{
    int i = 0;
    for (; i <= n - 2 * s; i += 2 * s)
        Merge(lst1, lst2, i, i + s - 1, i + 2 * s - 1);
    if (i + s < n)
        Merge(lst1, lst2, i, i + s - 1, n - 1);
    else
        for (int j = i; j < n; j++)
            lst1[j] = lst2[j];
}

template<class Type>
void MergeSort(Type lst[], const int& n)
{
    Type* temp = new Type[n];
    int s = 1;
    while (s < n)
    {
        MergePass(temp, lst, s, n);
        s += s;
        MergePass(lst, temp, s, n);
        s += s;
    }
    delete(temp);
}
```

其中 MergePass 函数无论是倒序序列还是正序序列都需要 O(n)时间完成，MergeSort 函数中共需调用 logn 次 MergePass 函数。因此，在最坏、平均和最优情况下所需的计算时间也均为 T(n) = O(nlogn)，空间复杂度为 O(n)。

第三个版本为自然归并排序，基本思路为首先对数组 a 进行依次线性扫描，找出有序片段，再将有序片段两两合并。该算法对于基本有序序列有着更好地性能。以下为自然归并排序版本代码实现：

```
template<class Type>
void MergePass(Type lst1[], const Type lst2[], const int& s, const int& n, const int divide[], const int& gn)
{
    int i = 0;
    for (; i <= gn - 2 * s; i += 2 * s)
        Merge(lst1, lst2, divide[i], divide[i + s] - 1, divide[i + 2 * s] - 1);
    if (i + s < gn)
        Merge(lst1, lst2, divide[i], divide[i + s] - 1, n - 1);
    else if(i < gn)
        for (int j = divide[i]; j < n; j++)
            lst1[j] = lst2[j];
}

template<class Type>
void MergeSort(Type lst[], const int& n)
{
    int* divide = new int[n];
    divide[0] = 0;
    int GroupNum = 1;  //组数
    for (int i = 0; i < n - 1; i++)
        if (lst[i + 1] < lst[i])
            divide[GroupNum++] = i + 1;        //记录每一组第一个元素位置
    Type* temp = new Type[n];
    int s = 1;
    while (s < GroupNum)
    {
        MergePass(temp, lst, s, n, divide, GroupNum);
        s += s;
        MergePass(lst, temp, s, n, divide, GroupNum);
        s += s;
    }
    delete(temp);
}
```

相比于迭代版本的归并排序，自然归并排序需要对序列进行一次扫描，对于使用 python 随机生成的 1e7 数据并没有更好表现，甚至时间消耗略长与迭代版本。但若将 1e7 数据中每 100 个数据进行排序处理生成 1e5 个有序段，自然归并排序就会有较大的效率提升。

自然归并排序扫描时间为 O(n)，需要一个大小为 n 的 divide 数组，因此算法的平均和最坏时间复杂度依旧为 O(nlogn)，空间复杂度依旧为 O(n)，但对于正序序列只需要一次线性扫描，最优时间复杂度可以提升至 O(n)，。

2.2 快速排序

快速排序是基于分治策略的另一个排序算法，其基本思想是对于输入的子数组 a[p:r]，按照一下三个步骤进行排序。

①分解：以 a[p]为基准元素将 a[p:r]划分为三段 a[p:q-1]，a[q]，a[q+1:r]，使第一段中任何元素均小于等于 a[q]，第三段中任意一元素均大于等于 a[p]。

②递归求解：递归调用快速排序算法，分别对 a[p:q-1]，a[q+1:r]进行排序。

③合并：由于 a[p:q-1]，a[q+1:r]的排序是就地进行的，因此排好序后不需要任何运算 a[p:r]就已经排好序。

本人一共实现了两个版本，第一个版本为随机选择基准元素的快速排序，随机选择划分基准可以期望划分是较对称的，代码如下：

```
template<class Type>
int Partition(Type a[], const int& p, const int& r)
{
    int i = p, j = r + 1;
    Type x = a[p];
    while (true)
    {
        while (a[++i] < x && i < r);
        while (a[--j] > x);
        if (i >= j)
            break;
        Swap(a[i], a[j]);  //交换大于基准元素的元素和小于基准元素的元素
    }
    a[p] = a[j];
    a[j] = x;
    return j;
}

template<class Type>
int RandomizedPartition(Type a[], const int& p, const int& r)
{
    srand(time(0));
    int i = rand() % (r - p + 1) + p;
    Swap(a[p], a[i]);
    return Partition(a, p, r);
}

template<class Type>
void QuickSort(Type a[], const int& p, const int& r)
{
    if(p < r)
    {
        int q = RandomizedPartition(a, p, r);
        QuickSort(a, p, q - 1);
        QuickSort(a, q + 1, r);
```

```
    }
}
```

RandomizedPartition 显然可在 O(n)时间内完成，而 QuickSort 递归调用的效率主要取决于基准元素在划分后的位置。最坏情况下，每次选中的基准元素都为序列最大值或最小值，划分后产生两个序列大小分别为 1 和 n-1，则时间复杂度为

$$T(n) = T(n-1) + O(n)$$

解递归方程后可得 T(n) = $O(n^2)$。

而对于最好情况，每次取得的基准值都为中位数，划分后产生两个大小为 n/2 的序列，此时时间复杂度为

$$T(n) = 2T(n/2) + O(n)$$

解递归方程后可得 T(n) = O(nlogn)。

可以证明，快速排序算法在平均情况下时间复杂度也是 O(nlogn)，这在基于比较的排序算法中算是快速的。

由于快速排序都在原地进行，并未申请辅助数组，空间复杂度为 O(1)。

第二个版本为三路快速排序，若序列中含有大量相同元素，可以使用三路快速排序进一步优化排序效率。具体思路为，将序列划分为三个序列段，第一段和第三段不在按照小于等于和大于等于划分，改为小于和大于，将等于基准元素的部分划分到中间，这样缩小了前后两段序列的大小，降低了递归调用的时间消耗。代码如下：

```
template<class Type>
range Partition(Type a[], const int& p, const int& r)
{
    //划分为三路，第一路小于小于基准元素，第二路等于基准元素，第三路大于基准元素
    int lt = p, i = p + 1, gt = r + 1; //lt 指向第一路的最右边，i 指向第二路的最右边，gt 指向第三路的最左边
    Type x = a[p];
    while (i < gt)
    {
        if (a[i] < x)
        {
            Swap(a[i], a[lt + 1]);
            i++;
            lt++;
        }
        else if (a[i] > x)
        {
            Swap(a[i], a[gt - 1]);
            gt--;
        }
        else
            i++;
    }
    a[p] = a[lt];
```

```
    a[lt] = x;
    return { lt, gt };
}

template<class Type>
void QuickSort(Type a[], const int& p, const int& r)
{
    if (p < r)
    {
        range q = Partition(a, p, r);
        QuickSort(a, p, q.left - 1);
        QuickSort(a, q.right, r);
    }
}
```

三、 实验结果

程序使用 window11 操作系统下 visual studio 2022 编写并编译得到可执行程序，输入文件为相同目录下的 mergesort.in 或 quicksort.in 文件，输入有两行，第一行为一个整数 n，第二行包含 n 个整数 n_i，每两个整数用空格隔开。输出文件名分别为 mergesort.out 和 quicksort.out，输出共一行。包含 n 个整数 n_i，为排序后的升序序列，每两个整数用空格隔开。

测试集可使用 python 快速生成，共生成 4 个测试集，分别为 1e7 个随机生成整数序列，1e7 个整数基本有序序列，其中每 100 个整数进行排序处理生成 1e5 个有序段，1e7 个正序整数序列，1e7 个倒序整数序列，整数范围均是[0, 1e5)。依次运行 4 个测试集后，结果如下：单位为 ms

	1e7 个随机生成整数序列	1e7 个基本有序整数序列	1e7 个正序整数序列	1e7 个倒序整数序列
递归归并排序	1891	1633	1112	1116
迭代归并排序	1340	1110	580	583
自然归并排序	1363	935	24	385
随机选择快速排序	1520	1413	838	855
三路随机选择快速排序	1392	1266	773	796

需要注意的一点是 vs 中 rand()函数的范围仅为[0, 65536]，远小于测试集数量，若不适当扩大其范围，对于顺序和倒序序列得到的基准元素将会始终小于第 65536 个元素，排序效果会非常差。本次测试将 rand()乘以 1000 适当扩大范围后最大值可以取到 1e7。

从表格可以看出，迭代程序效率普遍比递归要好，快速排序平均效果要比递归归并要好，三路快排效果比普通快排效果要好。同时，在基本有序或者正序序列，自然归并效果最好。

四、 实验总结

通过本次实验，本人对于两种排序算法和分治思想有了更深入的了解，主要的困难点在于探索算法优化思路，例如对于归并排序有自然归并的优化，对于快速排序有三路快速排序的优化。

事实上，本人还尝试使用了线性时间选择，选择中位数或仅做一次选择中位数的中位数作为快速排序的基准元素，然而，线性选择虽然时间复杂度可以达到 O(n)水平，但对序列每 5 个进行分组然后简单排序所消耗的时间却比直接随机选择基准元素的快速排序要高得多，因此最终放弃了此方案。

经过测试可以看出，递归版本的快速排序不如迭代版本的归并排序，但优于递归版本的归并排序，可以尝试使用栈模拟递归对快速排序进一步优化，可能可以达到比迭代版本的归并排序更好地执行效率。

五、 算法源代码

merge_sort_test.cpp

```
#include <iostream>
#include <fstream>
#include <time.h>
#include "recursive_merge_sort.cpp"

using namespace std;

int main()
{
    fstream fin("mergesort.in", ios::in), fout("mergesort.out", ios::out);
    int n;
    fin >> n;
    int* lst = new int[n];

    for (int i = 0; i < n; i++)
        fin >> lst[i];

    clock_t start, end;
    start = clock();
    MergeSort<int>(lst, n);
    end = clock();
    cout << "time: " << end - start;

    fout << lst[0];
    for (int i = 1; i < n; i++)
        fout << " " << lst[i];
```

```
    delete(lst);
    return 0;
}
```

recursive_merge_sort.cpp

```
using namespace std;

template<class Type>
void Merge(Type lst1[], const Type lst2[], const int& left, const int& middle,
const int& right)
{
    int i = left, j = middle + 1, k = left;
    while ((i <= middle) && (j <= right))
    {
        if (lst2[i] <= lst2[j])
            lst1[k++] = lst2[i++];
        else
            lst1[k++] = lst2[j++];
    }
    if (i > middle)
        for (; j <= right; j++)
            lst1[k++] = lst2[j];
    else
        for (; i <= middle; i++)
            lst1[k++] = lst2[i];
}

template<class Type>
void RecursiveMergeSort(Type lst1[], Type lst2[], const int& left, const int&
right)
{
    if (left < right)
    {
        int middle = (left + right) / 2;
        RecursiveMergeSort(lst1, lst2, left, middle);
        RecursiveMergeSort(lst1, lst2, middle + 1, right);
        Merge(lst2, lst1, left, middle, right);
        for (int i = left; i <= right; i++)
            lst1[i] = lst2[i];
    }
}

template<class Type>
void MergeSort(Type lst[], const int& n)
```

```
{
    Type* temp = new Type[n];  //避免每递归调用一次，申请一次辅助数组
    RecursiveMergeSort(lst, temp, 0, n - 1);
    delete(temp);
}
```

iterative_merge_sort.cpp

```
using namespace std;

template<class Type>
void Merge(Type lst1[], const Type lst2[], const int& left, const int& middle,
const int& right)
{
    int i = left, j = middle + 1, k = left;
    while ((i <= middle) && (j <= right))
    {
        if (lst2[i] <= lst2[j])
            lst1[k++] = lst2[i++];
        else
            lst1[k++] = lst2[j++];
    }
    if (i > middle)
        for (; j <= right; j++)
            lst1[k++] = lst2[j];
    else
        for (; i <= middle; i++)
            lst1[k++] = lst2[i];
}

template<class Type>
void MergePass(Type lst1[], const Type lst2[], const int& s, const int& n)
{
    int i = 0;
    for (; i <= n - 2 * s; i += 2 * s)
        Merge(lst1, lst2, i, i + s - 1, i + 2 * s - 1);
    if (i + s < n)
        Merge(lst1, lst2, i, i + s - 1, n - 1);
    else
        for (int j = i; j < n; j++)
            lst1[j] = lst2[j];
}

template<class Type>
void MergeSort(Type lst[], const int& n)
```

```
{
    Type* temp = new Type[n];
    int s = 1;
    while (s < n)
    {
        MergePass(temp, lst, s, n);
        s += s;
        MergePass(lst, temp, s, n);    //最终结果一定在 lst 中
        s += s;
    }
    delete(temp);
}
```

natural_merge_sort.cpp

```
using namespace std;

template<class Type>
void Merge(Type lst1[], const Type lst2[], const int& left, const int& middle,
const int& right)
{
    int i = left, j = middle + 1, k = left;
    while ((i <= middle) && (j <= right))
    {
        if (lst2[i] <= lst2[j])
            lst1[k++] = lst2[i++];
        else
            lst1[k++] = lst2[j++];
    }
    if (i > middle)
        for (; j <= right; j++)
            lst1[k++] = lst2[j];
    else
        for (; i <= middle; i++)
            lst1[k++] = lst2[i];
}

template<class Type>
void MergePass(Type lst1[], const Type lst2[], const int& s, const int& n, const
int divide[], const int& gn)
{
    int i = 0;
    for (; i <= gn - 2 * s; i += 2 * s)
        Merge(lst1, lst2, divide[i], divide[i + s] - 1, divide[i + 2 * s] - 1);
    if (i + s < gn)
```

```
        Merge(lst1, lst2, divide[i], divide[i + s] - 1, n - 1);
    else if(i < gn)
        for (int j = divide[i]; j < n; j++)
            lst1[j] = lst2[j];
}

template<class Type>
void MergeSort(Type lst[], const int& n)
{
    int* divide = new int[n];
    divide[0] = 0;
    int GroupNum = 1;  //组数
    for (int i = 0; i < n - 1; i++)
        if (lst[i + 1] < lst[i])
            divide[GroupNum++] = i + 1;        //记录每一组第一个元素位置
    Type* temp = new Type[n];
    int s = 1;
    while (s < GroupNum)
    {
        MergePass(temp, lst, s, n, divide, GroupNum);
        s += s;
        MergePass(lst, temp, s, n, divide, GroupNum);
        s += s;
    }
    delete(temp);
}
```

quick_sort_test.cpp

```
#include <iostream>
#include <fstream>
#include <time.h>
#include "threeway_quick_sort.cpp"

using namespace std;

int main()
{
    fstream fin("quicksort.in", ios::in), fout("quicksort.out", ios::out);
    int n;
    fin >> n;
    int* lst = new int[n];

    for (int i = 0; i < n; i++)
        fin >> lst[i];
```

```
    clock_t start, end;
    start = clock();
    QuickSort<int>(lst, 0, n - 1);
    end = clock();
    cout << "time: " << end - start;

    fout << lst[0];
    for (int i = 1; i < n; i++)
        fout << " " << lst[i];

    delete(lst);
    return 0;
}
```

random_quick_sort.cpp

```
using namespace std;

template<class Type>
inline void Swap(Type& a, Type& b)
{
    Type temp = b;
    b = a;
    a = temp;
}

template<class Type>
int Partition(Type a[], const int& p, const int& r)
{
    int i = p, j = r + 1;
    Type x = a[p];
    while (true)
    {
        while (a[++i] < x && i < r);
        while (a[--j] > x);
        if (i >= j)
            break;
        Swap(a[i], a[j]);  //交换大于基准元素的元素和小于基准元素的元素
    }
    a[p] = a[j];
    a[j] = x;
    return j;
}
```

```
template<class Type>
int RandomizedPartition(Type a[], const int& p, const int& r)
{
    srand(time(0));
    int i = (rand() * 1000) % (r - p + 1) + p;//rand()范围[0, 65536]，依据测试集大小扩大范围
    Swap(a[p], a[i]);
    return Partition(a, p, r);
}

template<class Type>
void QuickSort(Type a[], const int& p, const int& r)
{
    if(p < r)
    {
        int q = RandomizedPartition(a, p, r);
        QuickSort(a, p, q - 1);
        QuickSort(a, q + 1, r);
    }
}
```

threeway_quick_sort.cpp

```
using namespace std;

typedef struct range
{
    int left, right; //左闭右开
}range;

template<class Type>
inline void Swap(Type& a, Type& b)
{
    Type temp = b;
    b = a;
    a = temp;
}

template<class Type>
range Partition(Type a[], const int& p, const int& r)
{
    //划分为三路，第一路小于小于基准元素，第二路等于基准元素，第三路大于基准元素
    int lt = p, i = p + 1, gt = r + 1; //lt 指向第一路的最右边，i 指向第二路的最右边，gt 指向第三路的最左边
    Type x = a[p];
```

```
    while (i < gt)
    {
        if (a[i] < x)
        {
            Swap(a[i], a[lt + 1]);
            i++;
            lt++;
        }
        else if (a[i] > x)
        {
            Swap(a[i], a[gt - 1]);
            gt--;
        }
        else
            i++;
    }
    a[p] = a[lt];
    a[lt] = x;
    return { lt, gt };
}

template<class Type>
range RandomizedPartition(Type a[], const int& p, const int& r)
{
    srand(time(0));
    int i = (rand() * 1000) % (r - p + 1) + p;//rand()范围[0, 65536]，依据测试集大小扩大范围
    Swap(a[p], a[i]);
    return Partition(a, p, r);
}

template<class Type>
void QuickSort(Type a[], const int& p, const int& r)
{
    if (p < r)
    {
        range q = RandomizedPartition(a, p, r);
        QuickSort(a, p, q.left - 1);
        QuickSort(a, q.right, r);
    }
}
```