Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

**Отчет по лабораторной работе**

**«Сортировки массивов»**

**Выполнил**:

студент группы 382003-1

Ларионов А.И.

**Проверил**:

ассистент каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2019

**Содержание**

Постановка задачи 3

Метод решения 4

Руководство пользователя 5

Описание программной реализации 6

Подтверждение корректности 7

Результаты экспериментов 8

Заключение 9

Приложение 10

**Постановка задачи**

В данной лабораторной работе была поставлена задача исследовать алгоритмы сортировок массивов, состоящих из элементов типа double на языке Cи. Провести несколько экспериментов с различными входными данными и подсчетом перестановок и сравнений, чтобы подтвердить теоретическую сложность алгоритмов в худших, лучших и средних случаях.

В работе были рассмотрены сортировки:

1) Сортировка вставками

2) Сортировка Шелла

3) Поразрядная сортировка

4) Сортировка слиянием

**Метод решения**

**1) Сортировка вставками**

Будем считать, что первый элемент уже на своем месте. Тогда каждый последующий элемент массива сравниваем с предыдущим и когда предыдущий оказывается меньше или равен последующему – значит элемент находится на своем месте.

**2) Сортировка Шелла**

Для данной сортировки используются предварительные грубые проходы. Для данных проходов берется переменная STEP указывающая на каком расстоянии друг от друга находятся элементы для сортирования вставками. В моей сортировке для первого прохода используется STEP равная половине длинны массива для первого прохода, а для последующих STEP делится пополам. Если на последнем проходе массив еще не отсортирован и STEP равен 1, то происходит сортировка вставками.

**3) Поразрядная сортировка**

В поразрядной сортировке элементы массива рассматриваются как числа с определенным количеством цифр. Для каждого разряде числа проходит устойчивая сортировка подсчетом для массива цифр в определенном диапазоне. Благодаря устойчивости сортировки подсчетом, в случае если N-ый разряд двух чисел равен, они сохраняют порядок, в котором находились после сравнения по (N-1)-ому разряду.

**4) Сортировка слиянием**

Этот алгоритм рекурсивно разделяет исходный массив на подмассивы меньших размеров, после чего сортирует их. После этого происходит слияние этих подмассивов до тех пор пока размер склеенного массива не равен размеру исходному.

**Руководство пользователя**

1. Введите размер массива который желаете отсортировать.
2. Выберите один из трех вариантов наполнения массива, после чего увидите на экране элементы исходного массива в неотсортированном виде
3. Выберите одну из четырех сортировок для массива, после чего на экране появится число перестановок и сравнений, а так же отсортированные элементы исходного массива.
4. Вы можете нажать 1, чтобы проверить правильно-ли отсортирован массив.

**Описание программной реализации**

Проект состоит из сортировок представленных по-отдельности в .cpp файлах :

1. Insertion\_sort.cpp
2. Shell's\_sort.cpp
3. Поразрядная сортировка.cpp
4. Сортировка\_слиянием.cpp

Файл заголовка, где собраны все сортировки:

1. Lab\_sorts.h

Файл основной программы, в котором реализованы функции создания динамического массива, заполнения этого массива в трех вариантах, применения сортировки к массиву, а так же проверки массива:

1. Final\_Lab.cpp
2. Final\_Lab.exe

**Подтверждение корректности**

Для подтверждения корректности используется часть программы сравнивающая предыдущий и последующий элементы массива.

**Результаты экспериментов** На графике заметно что сортировка вставками подтверждает свою теоретическую сложность о(n^2) в среднем случае. В случае с отсортированным массивом данный алгоритм показывает линейную сложность. Для сортировки Шелла теоретическая сложность может варьироваться в зависимости от выбранного значения STEP. Для разных длин массивов существуют заранее подобранные последовательности значений STEP, помогающие достичь максимальной производительности. В худшем случае сложность алгоритма составляет О(n^2), в моих опытах алгоритм показал результаты близкие к O(n\*(ln(n))^2).

Алгоритм имеет сложность O(m\*(n+k)), где m – количество разрядов, n – количество объектов, которые нужно отсортировать, k – количество возможных значений цифры в числе. По памяти данный алгоритм имеет сложность O(n+m). Сортировка слиянием подтверждает свою теоретическую сложность и идет близко к O(n\*ln(n)). Плюсом для данной сортировки является отсутствие “сложных” входных данных. Однако ей требуется большое количество памяти на вспомогательный массив, а так же сортировка отсортированного массива для нее остается O(n\*ln(n)), что больше O(n), чего можно достичь в других сортировках

**Заключение**

В этой работе я рассмотрел и реализовал 4 алгоритма сортировок из множества возможных, заметно что для разных ситуаций могут подойти разные алгоритмы в зависимости как от входных данных, так и от ограничений по памяти или времени работы.

**Приложение**

void insertion\_sort(double arr[], int n)

{

int a = 0; int b = 0;

for (int i = 1; i < n; i++)

for (int j = i; j > 0 && arr[j - 1] - arr[j] >= 0; j--)

{

double tmp = arr[j - 1];

arr[j - 1] = arr[j];

arr[j] = tmp;

a = a + 3; b++;

}

printf("\n%d%s\n%d%s", a, "swaps", b, "comparisons");

}

void Shell\_sort(double array[], const int n)

{

int step, i, j;

double tmp;

for (step = n / 2; step > 0; step /= 2)

for (i = step; i < n; i++)

for (j = i - step; j >= 0 && array[j] > array[j + step]; j -= step)

{

tmp = array[j];

array[j] = array[j + step];

array[j + step] = tmp;

}

}

void createCounters(double\* data, int\* counters, long N)

{

unsigned char\* bp = (unsigned char\*)data;

unsigned char\* dataEnd = (unsigned char\*)(data + N);

unsigned short i;

memset(counters, 0, 256 \* sizeof(double) \* sizeof(long));

while (bp != dataEnd)

{

for (i = 0; i < sizeof(double); i++)

counters[256 \* i + \*(bp++)]++;

}

}

void radixPass(short Offset, long N, double\* source, double\* dest, int\* count)

{

double\* sp;

int s = 0, c, i, \* cp = count;

unsigned char\* bp;

for (i = 256; i > 0; --i, ++cp)

{

c = \*cp; \*cp = s; s += c;

}

bp = (unsigned char\*)source + Offset;

sp = source;

for (i = N; i > 0; --i, bp += sizeof(double), ++sp)

{

cp = count + \*bp; dest[\*cp] = \*sp; (\*cp)++;

}

}

void radixSort(double\* array, long N)

{

double\* additional\_array = (double\*)malloc(N \* sizeof(double));

int\* counters = (int\*)malloc(2048 \* sizeof(int));

int\* count;

int k = 0;

createCounters(array, counters, N);

for (unsigned short i = 0; i < sizeof(double); i++)

{

count = counters + 256 \* i;

radixPass(i, N, array, additional\_array, count);

for (int j = 0; j < N; j++)

{

array[j] = additional\_array[j];

}

}

while (array[k] >= 0 && k < N && !(k > 0 && array[k] <= 0 && array[k - 1] > 0)) k++;

for (int i = 0; i < N - k; i++)

{

array[i] = additional\_array[N - 1 - i];

}

for (int i = 0; i < k; i++)

{

array[N - k + i] = additional\_array[i];

}

memcpy(array, additional\_array, N \* sizeof(double));

free(additional\_array);

free(counters);

}

void merge(double array[], double spare[], int l, int r)

{

int i = l, q = (l + r) / 2, j = q + 1, k = 0;

while (i <= q && j <= r)

{

if (array[i] <= array[j])

{

spare[k] = array[i];

i++;

}

else

{

spare[k] = array[j];

j++;

}

k++;

}

while (i <= q)

{

spare[k] = array[i];

i++; k++;

}

while (j <= r)

{

spare[k] = array[j];

j++;

k++;

}

for (i = 0; i < k; i++)

array[l + i] = spare[i];

}

int merge\_sort(double array[], double spare[], int l, int r)

{

if (l < r)

{

int q = (l + r) / 2;

merge\_sort(array, spare, l, q);

merge\_sort(array, spare, q + 1, r);

merge(array, spare, l, r);

}

return(0);

=======

#pragma once

#include <iostream>

#include <stdio.h>

int swaps=0, comps=0;

void insertion\_sort(double arr[], int n)

{

int a = 0; int b = 0;

for (int i = 1; i < n; i++)

for (int j = i; j > 0 && arr[j - 1] - arr[j] >= 0; j--)

{

double tmp = arr[j - 1];

arr[j - 1] = arr[j];

arr[j] = tmp;

a = a + 3; b++;

}

printf("\n%d%s\n%d%s\n", a, "swaps", b, "comparisons");

}

void Shell\_sort(double array[], const int n)

{

int step, i, j,a = 0,b = 0;

double tmp;

for (step = n / 2; step > 0; step /= 2)

for (i = step; i < n; i++)

for (j = i - step; j >= 0 && array[j] > array[j + step]; j -= step)

{

tmp = array[j];

array[j] = array[j + step];

array[j + step] = tmp;

b = b + 1; a = a + 1;

}

printf("\n%d%s\n%d%s\n", a, "swaps", b, "comparisons");

}

void createCounters(double\* data, int\* counters, long N)

{

unsigned char\* bp = (unsigned char\*)data;

unsigned char\* dataEnd = (unsigned char\*)(data + N);

unsigned short i;

memset(counters, 0, 256 \* sizeof(double) \* sizeof(long));

while (bp != dataEnd)

{

for (i = 0; i < sizeof(double); i++)

counters[256 \* i + \*(bp++)]++;

}

}

void radixPass(short Offset, long N, double\* source, double\* dest, int\* count)

{

double\* sp;

int s = 0, c, i, \* cp = count;

unsigned char\* bp;

for (i = 256; i > 0; --i, ++cp)

{

c = \*cp; \*cp = s; s += c;

}

bp = (unsigned char\*)source + Offset;

sp = source;

for (i = N; i > 0; --i, bp += sizeof(double), ++sp)

{

cp = count + \*bp; dest[\*cp] = \*sp; (\*cp)++;

}

}

void radixSort(double\* array, long N)

{

double\* additional\_array = (double\*)malloc(N \* sizeof(double));

int\* counters = (int\*)malloc(2048 \* sizeof(int));

int\* count;

int k = 0, a = 0, b = 0;

createCounters(array, counters, N);

for (unsigned short i = 0; i < sizeof(double); i++)

{

count = counters + 256 \* i;

radixPass(i, N, array, additional\_array, count);

for (int j = 0; j < N; j++)

{

array[j] = additional\_array[j]; a++;

}

}

while (array[k] >= 0 && k < N && !(k > 0 && array[k] <= 0 && array[k - 1] > 0)) { k++; b++; }

for (int i = 0; i < N - k; i++)

{

array[i] = additional\_array[N - 1 - i]; a++;

}

for (int i = 0; i < k; i++)

{

array[N - k + i] = additional\_array[i]; a++;

}

memcpy(array, additional\_array, N \* sizeof(double));

free(additional\_array);

free(counters);

printf("\n%d%s\n%d%s\n", a, "swaps", b, "comparisons");

}

void merge(double array[], double spare[], int l, int r)

{

int i = l, q = (l + r) / 2, j = q + 1, k = 0;

while (i <= q && j <= r)

{

if (array[i] <= array[j])

{

swaps = swaps + 1;

comps = comps + 1;

spare[k] = array[i];

i++;

}

else

{

swaps = swaps + 1;

comps = comps + 1;

spare[k] = array[j];

j++;

}

k++;

}

while (i <= q)

{

swaps = swaps + 1;

spare[k] = array[i];

i++; k++;

}

while (j <= r)

{

swaps = swaps + 1;

spare[k] = array[j];

j++;

k++;

}

for (i = 0; i < k; i++)

{

swaps = swaps + 1;

array[l + i] = spare[i];

}

}

int merge\_sort(double array[], double spare[], int l, int r)

{

if (l < r)

{

int q = (l + r) / 2;

merge\_sort(array, spare, l, q);

merge\_sort(array, spare, q + 1, r);

merge(array, spare, l, r);

}

return(0);

}

int mergeSort(double array[], double spare[], int l, int r)

{

merge\_sort(array, spare, l, r);

printf("swaps = "); printf("%d\n", swaps);

printf("comparisons = "); printf("%d\n", comps);

return(0);

}