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

Отчёт по лабораторной работе

«Сортировки»

Выполнил:
студент группы 3822Б1ПМ1
Сомов Я.В.

Проверил:
преподаватель кафедры МОСТ
Волокитин В.Д.

Нижний Новгород
2022 г.

Оглавление

1. Постановка задачи

Сортировка — это процесс упорядочивания данных по определённому критерию. Конечная цель — облегчение поиска, обработки или представления данных. Сортировки часто необходимы для решения прикладных задач: вывод информации о ценах на товары в порядке возрастания, работа с большими базами данных (сложные запросы могут выполняться быстрее, если данные заранее были отсортированы) и т.п.

В настоящее время разработано и применяется на практике множество различных алгоритмов, каждый из которых имеет свои преимущества и недостатки, поэтому особую ценность представляет оценка скорости времени их выполнения. В данной лабораторной работе для изучения выбраны следующие алгоритмы: пузырьковая сортировка, сортировка выбором, сортировка Хоара (quicksort), сортировка слиянием. Предполагается, что алгоритмы будут работать с данными типа double — восьмибайтовыми числами с плавающей запятой.

Цель работы — доказать, что теоретическая оценка сложности данных алгоритмов верна.

Для достижения цели необходимо решить следующие задачи:

- изучить данные алгоритмы сортировки;
- реализовать алгоритмы для работы с числами с плавающей запятой на языке C;
- экспериментальным путём подтвердить теоретическую оценку сложности алгоритмов.

2. Метод решения

2.1 Пузырьковая сортировка

Алгоритм пузырьковой сортировки состоит из повторяющихся проходов по данному массиву. Во время каждого прохода производится сравнение рядом стоящих элементов, и если они не стоят по порядку, то элементы переставляются местами. Алгоритм выполняется до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны.

Можно заметить, что после $i-$ой итерации внешнего цикла i последних элементов в массиве уже отсортированы и находятся на своём месте. Поэтому нет смысла обходить каждый раз массив от начала до конца, и на каждом следующем проходе число обрабатываемых элементов следует уменьшать на 1.

Для пузырьковой сортировки время работы в среднем составляет $T = O(n^2)$.

2.2 Сортировка выбором

Алгоритм условно делит массив на две части: отсортированную, которая выстраивается слева направо в начале массива, и неотсортированную, занимающую остальную часть массива. Изначально подмассив отсортированных элементов пуст, а подмассив неотсортированных элементов — это весь исходный массив. На каждом $i-$м шаге алгоритм находит минимальный элемент в подмассиве неотсортированных элементов и меняет его местами с $i-$м элементом массива, тем самым сдвигая границу между подмассивами вправо. Алгоритм продолжает работу до тех пор, пока подмассив неотсортированных элементов не станет пустым.

Для сортировки выбором время работы в среднем составляет $T = O(n^2)$.

2.3 Сортировка Хоара

Работа алгоритма основана на принципе «разделяй и властвуй». В массиве выбирается так называемый опорный элемент, после чего производится разбиение таким образом, что элементы, меньшие опорного, помещаются перед ним, а большие — после него. К двум подмассивам, образовавшимся слева и справа от опорного элемента, рекурсивно применяется алгоритм сорти-

ровки. Рекурсия не применяется к подмассиву, в котором отсутствуют элементы, или в котором содержится только один элемент.

Для сортировки Хоара время работы в среднем составляет $T = n \log(n)$.

2.4 Сортировка слиянием

Алгоритм сортировки слиянием работает следующим образом. Вначале исходный массив делится на N одноэлементных подмассивов, затем происходит последовательное слияние образованных подмассивов в новые упорядоченные подмассивы до тех пор, пока не останется единственный подмассив. Главная особенность алгоритма сортировки слиянием — необходимость в использовании дополнительной памяти, что в целом замедляет время работы программы и может вызывать проблемы при работе с большими массивами данных.

Для сортировки слиянием время работы в среднем составляет $T = n \log(n)$.

3. Руководство пользователя

Работа пользователя с программой осуществляется посредством интерфейса командной строки (Command line interface, CLI). Вначале программа запрашивает от пользователя размер массива, который впоследствии будет заполнен случайно сгенерированными числами. В случае ввода неверного значения (любое значение, меньшее или равное нулю) программа выведет сообщение об ошибке.

```
Введите размер массива: 100000
```

Затем программа запросит у пользователя количество итераций, то есть число повторений работы программы. На каждой новой итерации программа будет создавать новый массив и заново заполнять его числами. Если пользователь желает, чтобы алгоритм проработал один раз, то он должен ввести значение 1. В случае ввода некорректного значения программа выведет сообщение об ошибке.

```
Введите размер массива: 100000
Введите количество итераций: 5
```

После этого программа спросит у пользователя, какой алгоритм будет применяться. Чтобы выбрать алгоритм, необходимо ввести соответствующую алгоритму цифру. В случае ввода любой цифры, кроме 1, 2, 3, 4, программа выведет сообщение об ошибке.

```
Введите размер массива: 100000
Введите количество итераций: 5
Введите используемую сортировку:
1. Сортировка выбором
2. Пузырьковая сортировка
3. Сортировка Хоара
4. Сортировка слиянием
3
```

В случае, если все данные были введены корректно, программа приступит к созданию массива, заполненного числами с плавающей запятой, отсортирует массив с помощью указанного

алгоритма, повторит этот процесс заданное пользователем число раз. Во время работы программы будет выводиться информация о ходе работы: текущее выполняемое действие, затраченное на выполнение сортировки время. Если по какой-то причине алгоритм не смог отсортировать массив, будет выведено сообщение об ошибке, а выполнение программы будет прервано.

```
Итерация #1: Создание массива размера 100000...
Сортировка...
Проверка...
Массив отсортирован правильно.
Приблизительное затраченное время: 0,019000
Итерация #2: Создание массива размера 100000...
Сортировка...
Проверка...
Массив отсортирован правильно.
Приблизительное затраченное время: 0,018000
Итерация #3: Создание массива размера 100000...
Сортировка...
Проверка...
Массив отсортирован правильно.
Приблизительное затраченное время: 0,019000
Итерация #4: Создание массива размера 100000...
Сортировка...
Проверка...
Массив отсортирован правильно.
Приблизительное затраченное время: 0,019000
Итерация #5: Создание массива размера 100000...
Сортировка...
Проверка...
Массив отсортирован правильно.
Приблизительное затраченное время: 0,018000
```

В случае успешного завершения работы программы будут выведены сообщение об успешном завершении работы программы и статистика по времени работы, включающая в себя лучшее (минимальное), среднее, худшее (максимальное) время выполнения алгоритма.

```
Выполнение программы завершено. Полученные результаты:
Лучшее время:
0,018000
Среднее время:
0,018600
Худшее время: 0,019000
```

4. Описание программной реализации

Проект состоит из единственного файла main.c, содержащего программную реализацию четырёх алгоритмов сортировок: выбором, пузырьковую, Хоара, слиянием. Реализованные алгоритмы работают с числами с плавающей запятой типа double и сортируют массивы по возрастанию.

В таблице 4 содержится основная информация о реализованных функциях: типы возвращаемых значений, прототипы подпрограмм, краткое описание. Важное примечание: для использования алгоритма Хоара или алгоритма сортировки слиянием следует использовать quicksort и mergesort, а не __quicksort и __mergesort.

Таблица 1: Реализованные в программе функции

Тип возвращаемого значения	**Прототип функции**	**Описание**
int	equals(double a, double b)	Проверяет два числа с плавающей запятой на равенство.
int	compare(const void* a, const void* b)	Компаратор, работающий с библиотечной функцией qsort
void	swap(double* a, double* b)	Меняет местами значения переменных
double*	create_array(size_t size)	Создаёт массив указанного размера, заполняет его числами и возвращает указатель на массив
size_t	partition(double* arr, size_t left, size_t right)	Выбирает срединный элемент массива в качестве опорного, затем разбивает массив на две части: подмассивы элементов больших опорного и элементов меньших опорного, возвращает номер позиции опорного элемента после проведения разбиения
void	__quicksort(double* arr, size_t left, size_t right)	Сортирует массив с помощью алгоритма сортировки Хоара

продолжение на следующей странице

Таблица 1 – *продолжение*

Тип возвращаемого значения	**Прототип функции**	**Описание**
void	quicksort(double* arr, size_t size)	Конечная процедура сортировки Хоара
void	merge(double* in, double* out, size_t left, size_t mid, size_t right)	Объединяет два данных массива в один упорядоченный
void	__mergesort(double* in, double* out, size_t size)	Сортирует массив с помощью алгоритма сортировки слиянием
void	mergesort(double* arr, size_t size)	Конечная процедура сортировки слиянием
void	bubblesort(double* arr, size_t size)	Сортирует массив с помощью алгоритма пузырьковой сортировки
void	selectionsort(double* arr, size_t size)	Сортирует массив с помощью алгоритма сортировки выбором
int	check_array(double* arr, double* cpy, size_t size)	Проверяет правильность выполнения алгоритма сортировки

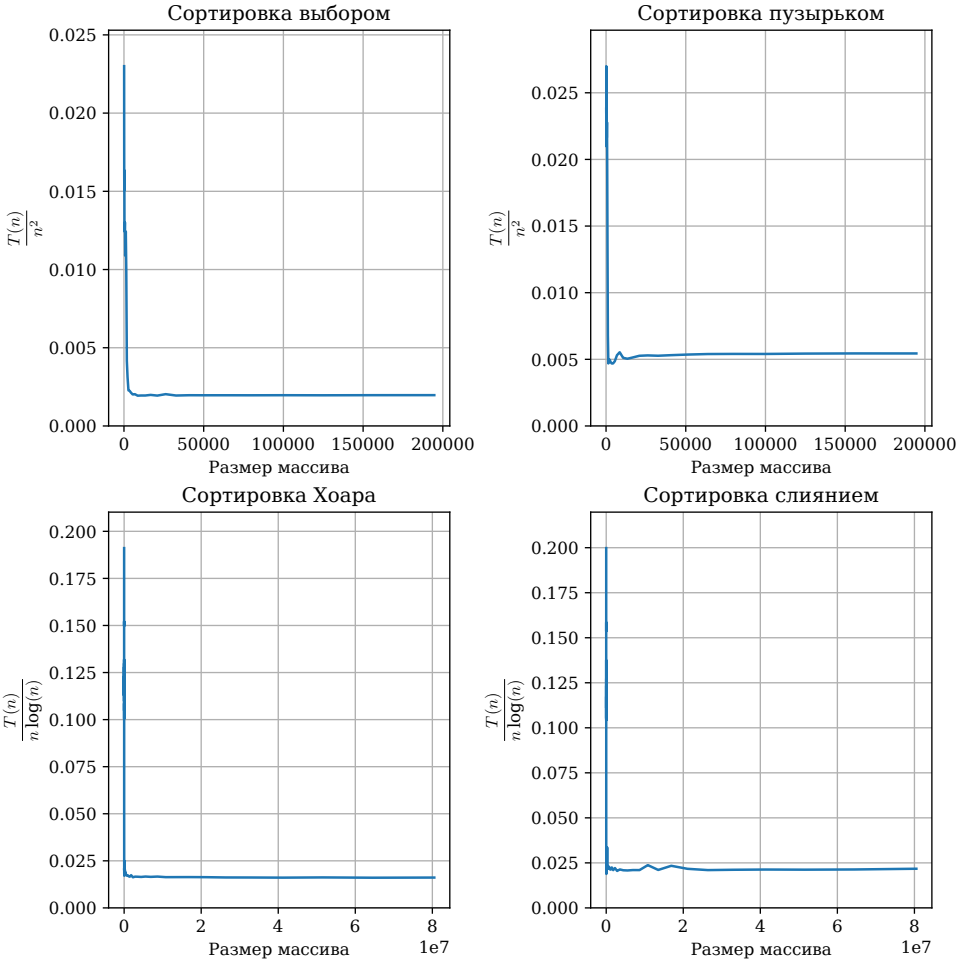
5. Подтверждение корректности

Для подтверждения корректности работы программы была написана функция check_array, использующую встроенную в стандартную библиотеку языка C функцию qsort. На вход подаются два массива: копия изначального неотсортированного массива и собственно отсортированный массив. Копия сортируется с помощью функции qsort, затем запускается поэлементное сравнение массивов. Если найдено хотя бы одно различное значение, значит исходный массив не был отсортирован, работа программы прерывается и выводится сообщение об ошибке. Если различия не были найдены, работа программы продолжается.

6. Результаты экспериментов

Чтобы подтвердить теоретическую оценку сложности алгоритмов, достаточно рассмотреть отношение $\frac{T(n)}{g(n)}$, где n — количество элементов в массиве, $T(n)$ — реально затраченное время на работу алгоритма, $g(n)$ — функция, с помощью которой оценивается время работы алгоритма. Если теоретическая оценка верна, то существует $\lim\limits_{n\to\infty} \frac{T(n)}{g(n)}$, равный некой константе.

Рис. 1: Отношение $\frac{T(n)}{g(n)}$ для различных алгоритмов сортировки



На рисунке 1 представлены результаты эксперимента: поведение $\frac{T(n)}{g(n)}$ при росте размера массива. При достаточно больших n отношение стремится к некой малой константе, большей нуля что является непосредственным подтверждением правильности теоретической оценки времени работы рассмотренных алгоритмов.

7. Заключение

В результате проведения лабораторной работы была достигнута поставленная цель и выполнены следующие исследовательские задачи:

- изучены следующие алгоритмы: пузырьковой сортировки, сортировки выбором, сортировки Хоара, сортировки слиянием
- создана программная реализация выбранных алгоритмов сортировок
- подтверждена в результате проведённого эксперимента теоретическая оценка времени работы указанных алгоритмов.

.

8. Приложение

```c
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "time.h"
#include "locale.h"
#include "math.h"

#define EPSILON 2.220446e-16

// Операция меньше или равно для двух чисел с плавающей запятой
#define less_eq(a, b) ((a - b < 0) || (fabs(a - b) < EPSILON))

// Проверка на равенство двух чисел с плавающей запятой
int equals(double a, double b) { return (fabs(a - b) < EPSILON); }

// Функция compare для корректной работы с функцией qsort из "stdlib.h"
int compare(const void* a, const void* b)
{
        if (*(double*)a - *(double*)b < 0)
                return -1;
        else if (fabs(*(double*)a - *(double*)b) < EPSILON)
                return 0;
        else
                return 1;
}

// Функция перестановки двух переменных типа double
void swap(double* a, double* b)
{
        double temp = *a;
        *a = *b;
        *b = temp;
}

// Создание массива double размерности size и заполнение случайными элементами
double* create_array(size_t size)
{
        double* arr = (double*)malloc(sizeof(double) * size);
        srand(time(NULL));
        for (size_t i = 0; i < size; i++)
                arr[i] = (double)rand() / (double)RAND_MAX;
        return arr;
}

// Вспомогательная функция partition
size_t partition(double* arr, size_t left, size_t right)
{
        double mid = arr[left + (right - left) / 2];
        left--;
        right++;
```

```
51
52          while (1)
53          {
54                  do { left++; } while (arr[left] < mid);
55                  do { right--; } while (arr[right] > mid);
56                  if (left >= right) return right;
57                  swap(&arr[left], &arr[right]);
58          }
59  }
60
61  // Собственно алгоритм сортировки Хоара
62  // Используется внутри функции quicksort
63  void __quicksort(double* arr, size_t left, size_t right)
64  {
65          if (left < right)
66          {
67                  size_t pivot = partition(arr, left, right);
68                  __quicksort(arr, left, pivot);
69                  __quicksort(arr, pivot+1, right);
70          }
71  }
72
73  // Алгоритм сортировки Хоара
74  void quicksort(double* arr, size_t size)
75  {
76          __quicksort(arr, 0, size-1);
77  }
78
79  // Вспомогательная функция merge для слияния массивов в один упорядоченный
80  void merge(double* in, double* out, size_t left, size_t mid, size_t right)
81  {
82          size_t l_curr = left;
83          size_t r_curr = mid;
84
85          for(size_t i = left; i < right; i++)
86          {
87                  if ((l_curr != mid) && (r_curr != right))
88                  {
89                          if (less_eq(in[l_curr], in[r_curr]))
90                                  out[i] = in[l_curr++];
91                          else
92                                  out[i] = in[r_curr++];
93                  }
94
95                  else if (l_curr != mid)
96                          out[i] = in[l_curr++];
97                  else
98                          out[i] = in[r_curr++];
99          }
100 }
101
102 // Собственно сортировка слиянием
103 // Используется внутри функции mergesort
104 void __mergesort(double* in, double* out, size_t size)
105 {
106         size_t step = 1;
107         while (step < size)
108         {
109                 for (size_t i = 0; i < size; i += 2 * step)
110                         merge(in, out, i, min(i + step, size), min(i + 2 * step,
    ↪  size));
```

```
111                 memcpy(in, out, sizeof(double) * size);
112                 step *= 2;
113         }
114 }
115
116 // Алгоритм сортировки слиянием
117 void mergesort(double* arr, size_t size)
118 {
119         double* tmp = (double*)malloc(sizeof(double) * size);
120         memcpy(tmp, arr, sizeof(double) * size);
121         __mergesort(arr, tmp, size);
122         free(tmp);
123 }
124
125 // Алгоритм сортировки пузырьком
126 void bubblesort(double* arr, size_t size)
127 {
128         int flag = 0;
129         for (size_t i = 0; i < size; i++)
130         {
131                 for (size_t j = 0; j < size - 1 - i; j++)
132                         if (less_eq(arr[j + 1], arr[j]))
133                         {
134                                 swap(&arr[j], &arr[j + 1]);
135                                 flag = 1;
136                         }
137                 if (!flag) return;
138         }
139 }
140
141 // Алгоритм сортировки выбором
142 void selectionsort(double* arr, size_t size)
143 {
144         size_t current_min_ind = 0;
145         for (size_t i = 0; i < size - 1; i++)
146         {
147                 current_min_ind = i;
148                 for (size_t j = i + 1; j < size; j++)
149                 {
150                         if (less_eq(arr[j], arr[current_min_ind]))
151                                 current_min_ind = j;
152                 }
153                 swap(&arr[i], &arr[current_min_ind]);
154         }
155 }
156
157 // Проверка правильности работы выбранного алгоритма
158 int check_array(double* arr, double* cpy, size_t size)
159 {
160         qsort((void*)cpy, size, sizeof(double), compare);
161         for (size_t i = 0; i < size; i++)
162                 if (!equals(arr[i], cpy[i]))
163                         return 0;
164         return 1;
165 }
166
167 int main()
168 {
169         double* arr;
170         double* copy;
```

```
171         double approx_time = 0, average_time = 0, best_time = 1.79769e+308, worst_time
    ↪  = 0;
172         size_t array_size;
173         size_t iteration_count;
174         int choice;
175         int flag = 1;
176         clock_t start, end;
177         void (*sort) (double*, size_t) = NULL;
178 
179         setlocale(LC_ALL, "ru_RU.utf8");
180         // Пользователь вводит размер массива,
181         printf("Введите размер массива: ");
182         scanf_s("%Iu", &array_size);
183         // ...затем вводит количество повторений работы программы
184         printf("Введите количество итераций: ");
185         scanf_s("%Iu", &iteration_count);
186         // ...затем выбирает алгоритм сортировки из четырёх предложенных
187         printf("Введите используемую сортировку:\n1. Сортировка выбором\n2. Пузырьковая
    ↪  сортировка\n3. Сортировка Хоара\n4. Сортировка слиянием\n");
188         scanf_s("%d", &choice);
189 
190         // Обработка выбора
191         switch (choice)
192         {
193         case 1:
194                 sort = selectionsort;
195                 break;
196         case 2:
197                 sort = bubblesort;
198                 break;
199         case 3:
200                 sort = quicksort;
201                 break;
202         case 4:
203                 sort = mergesort;
204                 break;
205         default:
206                 flag = 0;
207                 break;
208         }
209 
210         if (array_size <= 0)
211                 flag = 0;
212         if (iteration_count <= 0)
213                 flag = 0;
214 
215         if (flag)
216         {
217                 for (size_t iteration = 0; iteration < iteration_count; iteration++)
218                 {
219                         // На каждой итерации создаётся новый массив и его копия
220                         // Исходный массив сортируется с помощью выбранного алгоритма
221                         // Копия массива сортируется с помощью библиотечной функции
    ↪  qsort
222                         // Копия и исходный массив сравниваются.
223                         // Если между ними нашлись различия, работа программы
    ↪  прерывается и выводится сообщение об ошибке
224 
225                         if (!flag) break;
226                         printf("Итерация #%Iu: Создание массива размера %Iu...\n",
    ↪  iteration + 1, array_size);
```

```
227                         arr = create_array(array_size);
228                         copy = (double*)malloc(sizeof(double) * array_size);
229                         if(copy)
230                                 memcpy(copy, arr, sizeof(double) * array_size);
231                         else
232                         {
233                                 flag = 0;
234                                 break;
235                         }
236
237                         printf("Сортировка...\n");
238                         start = clock();
239                         sort(arr, array_size);
240                         end = clock();
241                         printf("Проверка...\n");
242
243                         if (check_array(arr, copy, array_size))
244                         {
245                                 printf("Массив отсортирован правильно.\n");
246                                 approx_time = (double)(end - start) / CLOCKS_PER_SEC;
247                                 printf("Приблизительное затраченное время: %lf\n",
    ↪  approx_time);
248                                 worst_time = max(approx_time, worst_time);
249                                 best_time = min(approx_time, best_time);
250                                 average_time = (average_time * iteration + approx_time)
    ↪  / (iteration + 1);
251                         }
252                         else
253                         {
254                                 printf("Массив не был отсортирован.\n");
255                                 flag = 0;
256                         }
257
258                         free(arr);
259                         free(copy);
260                 }
261
262                 printf("Выполнение программы завершено. Полученные результаты:\nЛучшее
    ↪  время: %lf\nСреднее время: %lf\nХудшее время: %lf\n", best_time, average_time,
    ↪  worst_time);
263         }
264
265         else
266                 printf("Произошла ошибка.\n");
267
268         return 0;
269 }
```