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»**

**Выполнил**:

студент группы 3823Б1ПМ1

Фамилия И.О.

**Проверил**:

преподаватель каф. ВВСП,

Волокитин В.Д.

Нижний Новгород

2023

**Содержание**

[Постановка задачи 3](#_Toc154710141)

[Метод решения 4](#_Toc154710142)

[Руководство пользователя 6](#_Toc154710143)

[Описание программной реализации 7](#_Toc154710144)

[Подтверждение корректности 12](#_Toc154710145)

[Результаты экспериментов 13](#_Toc154710146)

[Заключение 16](#_Toc154710147)

[Литература 17](#_Toc154710148)

[Приложение 18](#_Toc154710149)

# Постановка задачи

* Реализовать сортировки массива, состоящего из элементов типа double «пузырёк», Хоара, слиянием и поразрядную, в языке программирования Си;
* Создать пользовательский интерфейс для сортировки и анализа сортировок;
* Провести тестирование сортировок, проверить их на корректность;
* Провести ряд экспериментов, в ходе которых доказать асимптотическую сложность алгоритмов, построить график отношения времени к асимптотической сложности и найти константу.

# Метод решения

Описание алгоритмов в исходном виде см. в п. 1 раздела Литература.

1. Сортировка «пузырёк» (Bubble sort)

Средний случай – O

Лучший случай – O

Худший случай – O

В программе реализована вариация «пузырька» – «шейкер». В начале устанавливаются левая и правая границы неотсортированного массива (начало и конец соответственно). Индекс проходит от левой границы до правой сравнивая текущий элемент со следующим, если нужно, меняет их местами и запоминает/перезаписывает позицию замены. По окончании прохода по массиву правая граница сдвигается на записанную позицию (после неё замен не было, а значит все элементы упорядочены). Те же действия выполняются при обратном ходе, но сравнивается текущий элемент с предыдущим и сдвигается левая граница. Цикл проходов повторяется до тех пор, пока левая и правая границы не будут указывать на один элемент.

1. Сортировка Хоара (Quicksort)

Средний случай – O

Лучший случай – O

Худший случай – O

Идея быстрой сортировки в том, чтобы выбрать опорный элемент и разделить массив на две части: в левой собрать все элементы меньшие или равные опорному, а в правой – большие или равные опорному. Затем провести те же действия над каждой из частей. Если размер одной из частей равен 1, то над ней никаких действий не производится.

1. Сортировка слиянием (Merge sort).

Средний случай – O

Лучший случай – O

Худший случай – O

В сортировке слиянием исходный массив разбивается на массивы единичной длины. Все соседствующие пары массивов слияются в один упорядоченный по возрастанию. Над полученными массивами проводятся те же операции, до тех пор, пока все части не сольются в один отсортированный.

1. Поразрядная сортировка (Radix sort)

Средний случай – O

Лучший случай – O

Худший случай – O

(Здесь k – количество разрядов элемента)

В поразрядной сортировке элементы не сравниваются между собой. Вместо этого поочерёдно рассматриваются разряды элементов (для стандартных типов Си роль разряда играет 1 байт) в порядке возрастания значимости (для восходящей сортировки). Сначала подсчитывается количество элементов с одинаковыми значениями в самом младшем разряде. По их количеству строится массив начальных позиций для элементов с одинаковым значением разряда в порядке возрастания. Исходный массив перезаписывается по порядку от начала до конца: каждому элементу по его значению текущего разряда ставится в соответствие позиция, на которую он должен встать, затем позиция для этого значения сдвигается на 1 элемент вправо. По итогу получается массив, отсортированный по первому разряду. Те же действия выполняются для всех последующих разрядов. При этом для элементов с одинаковыми разрядами на свои позиции сначала будут вставать те, у которых предыдущий разряд меньше.

# Руководство пользователя

После запуска, программа попросит пользователя ввести размер массива, программа сама создаёт случайный массив c элементами в диапазоне от -1000 до 1000 и выводит его

Далее на экране монитора появится выбор сортировки:

1 – Bubble Sort;

2 – Quick Sort;

3 – Merge Sort;

4 – Radix Sort.

В качестве результата на экране дисплея появится:

* количество тактов процесса, в ходе выполнения сортировки
* сообщение о корректности сортировки (если сортировка провалилась, то программа выведет номер элемента, начиная с 0 с которого начинается несоответствие).

Также по желанию пользователя можно вывести и сам массив.

По завершению пользователю даётся выбор: выйти из программы или продолжить работу.

# Описание программной реализации

1. Сортировка «пузырьком»

Функция представляет из себя для вложенных цикла. Перед началом задаются левая *left* и правая *right* границы несортированного массива. Индекс *i* проходит от первого элемента до предпоследнего, сравнивая текущий со следующим и, если надо, меняет их местами и запоминает позицию последней замены *control*. Таким образом локальные максимумы «поднимаются вверх» по массиву. Значение *control* показывает, что все элементы правее него уже заняли свои места, поэтому в *right* присваивается *control.* Действия при обратном проходе симметричны.

|  |  |
| --- | --- |
|  | BUBBLE(A, n)[[1]](#footnote-1) |
| 1 | left = 0, right = n-1; |
| 2 | while left ≠ right |
| 3 | for (i = left; i < right; i++) |
| 4 | if A[i] > A[i+1] |
| 5 | swap(A[i] , A[i+1]); |
| 6 | control = i; |
| 7 | right = control; |
| 8 | for (i = right; i > left; i--) |
| 9 | if A[i] < A[i - 1] |
| 10 | swap(A[i] , A[i - 1]); |
| 11 | control = i; |
| 12 | left = control; |

1. Сортировка Хоара (см. п. 2 в разделе Литература)

Программная реализация выполнена рекурсивно. На вход подаётся указатель на сортируемый массив, и границы части массива: левая – *start* (начальное значение – 0) и правая – *finish* (начальное значение – n - 1). С помощью функции *PARTITION* поданный на вход массив разбивается на две части: из массива выбирается случайным образом ключевой элемент *a*, два индекса двигаются с разных концов массива (*i* – слева, *j* – справа) до тех пор, пока *i* не встретит элемент ≥ *a*, а *j* – ≤ *a*. Если *i* и *j* ещё не прошли мимо друг друга, то меняются местами элементы, на которые они указывают, и сами сдвигаются на 1 шаг. В конечном счёте *j* окажется слева от *i* при этом слева от *i* будут только элементы ≤ *a*, справа от *j* – только ≥ *a*, а между *i* и *j* – либо не будет элементов, либо будет один элемент равный ключевому. После окончания разбиения вызываются две функции QUICK для границ *s*, *j* и *i*, *f* соответственно, если границы не указывают на массив из одного элемента или не имеют смысла.

|  |  |
| --- | --- |
|  | QUICK(A, start, finish) |
| 1 | partition(A, &i, &j); |
| 2 | if (j - start) |
| 3 | QUICK(A, start, j); |
| 4 | if (finish - i) > 0 |
| 5 | QUICK(A, i, finish); |

|  |  |
| --- | --- |
|  | PARTITION(A, \*i, \*j) |
| 1 | double a = arr[\*i + rand() % (\*j - \*i + 1)]; |
| 2 | while (\*i <= \*j) |
| 3 | while (A[\*i] < a) |
| 4 | (\*s)++; |
| 5 | while (A[\*j] > a) |
| 6 | (\*f)--; |
| 7 | if (\*i <= \*j) |
| 8 | swap(A[\*i], A[\*j]); |
| 9 | (\*i)++; |
| 10 | (\*j)--; |

1. Сортировка слиянием

В программе реализован итеративный алгоритм, начинающийся со слияния единичных массивов (*step* = 1). В теле цикла происходит поочерёдное слияние (функция *MERGE*(…)) соседних пар частей массива длины *step* и запись полученных частей во вспомогательны массив *tmp\_A* на позицию, соответствующую началу первой части в паре. В конце массива, при этом, может быть пара частей разных размеров или одна часть без пары. В этом случае для соответствующих частей вместо длины *step* заносится длина того, что осталось. Когда все пары слились, меняются значениями указатели на исходный и вспомогательный массивы и изменяется *flag* так, что если число замен чётное, то *flag* = 0, иначе *flag* = 1. Это нужно, чтобы в последнем случае занести отсортированный массив в исходную память (расположенную по адресу *tmp\_A*)

|  |  |
| --- | --- |
|  | MERGE\_SORT(A, n) |
| 1 | for (step = 1; step < n; step \*= 2) |
| 2 | for (i = 0; i < n; i += 2 \* step) |
| 3 | MERGE(A + i, A + i + step, tmp\_A + i, min(step, n - i), max(0, min(step, n - step - i))); |
| 4 | swap(A, tmp\_A); |
| 6 | flag = 1 - flag; |
| 5 | if (flag) |
| 7 | equalize\_arr(A, tmp\_A, n); |

Для слияния массива два индекса *i* и *j* проходят по своим слияемым массивам, и заносят в новый массив меньший элемент из тех, на который они указывают. Если один из массивов заканчивается, то остаток второго заносится без сравнения.

|  |  |
| --- | --- |
|  | MERGE(A1, A2, A3, size1, size2) |
| 1 | for (k = 0, i = 0, j = 0; k < n1 + n2; k++) |
| 2 | if (i == n1) |
| 3 | m3[k] = m2[j++]; |
| 4 | else if (j == n2) |
| 5 | m3[k] = m1[i++]; |
| 6 | else |
| 7 | if ((m1[i]) < m2[j]) |
| 8 | m3[k] = m1[i++]; |
| 9 | else m3[k] = m2[j++]; |

1. Поразрядная сортировка (Radix sort)

Стандартные типы в Си хранятся в виде инверсированной последовательности байтов, то есть младшие байты идут раньше. В первых байтах записана мантисса, в более старших – порядок со сдвигом, в последнем бите последнего байта – знак. Так что при равных порядках больше по модулю то число, у которого больше мантисса; число с большим порядком больше по модулю независимо от мантиссы. Функция *RADIX*(…), представляет исходный массив в виде массива *Auc* из unsigned char. Теперь можно взять значение разряда (байта) элемента типа double. Начиная с меньшего разряда их значения от каждого элемента передаётся в функцию *FOFFSET*(…), где подсчитывается их количество и заносится в одну из 256 ячеек массива *count*[256], где номер ячейки соответствует значению разряда. Из *count*[0] вычитается 1, далее, начиная с *count*[1], подсчитывается интегральная сумма (прибавляется значение предыдущей ячейки). Таким образом получается массив, содержащий последние позиции, на которые должны встать элементы с соответствующими разрядами. По этим позициям исходный массив переписывается с конца во вспомогательный массив *A2*, при этом, когда очередной элемент встаёт на своё место, значение ячейки, хранящей позицию, соответствующую разряду элемента, уменьшается на 1. Теперь указатели *A* иA2 меняются, новый массив опять представляется в виде *Auc* unsigned char. Те же действия выполняются для всех остальных разрядов. При сначала этом на свои места будут вставать элементы с большим значением в младшем разряде (для этого и нужно осуществлять запись с конца). В конце получится массив (хранящийся по исходному адресу), у которого вначале отсортированные положительные элементы, а в конце – отрицательные в обратном порядке (сначала меньшие по модулю). Чтобы расставить их в правильном порядке, методом бинарного поиска ищется позиция первого отрицательного элемента, все элементы, начиная с него, но в обратном порядке, записываются в *A2*, после них туда же в прямом порядке записываются все положительные элементы. Массив отсортирован.

|  |  |
| --- | --- |
|  | RADIX(A, n) |
| 1 | Auc = (\*unsigned char)A; |
| 2 | for (i = 0; i < sizeof(double); i++) |
| 3 | foffset(Auc, n, i, count); |
| 4 | for (j = n - 1; j >= 0; j--) |
| 5 | A2[count[Auc[j \* sizeof(double) + i]]--] = A[j]; |
| 6 | swapp(&A2, &A); |
| 7 | Auc = (unsigned char\*)A; |
| 8 | if (A[n - 1] < 0) |
| 9 | i = 0, j = n - 1; |
| 10 | while ((j - i) > 1) |
| 11 | sgnp = (j + i) / 2; |
| 12 | if (A[sgnp] >= +0) |
| 13 | i = sgnp; |
| 14 | else |
| 15 | j = sgnp; |
| 16 | if (A[sgnp] >= +0) |
| 17 | sgnp += 1; |
| 18 | equalize\_arr(A, arr2, sgnp); |
| 19 | for (i = sgnp; i < n; i++) |
| 20 | A2[i] = A[n - i + sgnp - 1]; |
| 21 | equalize\_arr (A2 + sgnp, A, n - sgnp); |
| 22 | equalize\_arr(A2, A + (n - sgnp), sgnp); |

|  |  |
| --- | --- |
|  | FOFFSET(Auc, n, digit, count[256]) |
| 1 | for (i = 0; i < 256; i++) |
| 2 | count[i] = 0; |
| 3 | for (i = 0; i < n; i++) |
| 4 | count[A[digit + i \* sizeof(double)]]++; |
| 5 | count[0] -= 1; |
| 6 | for (i = 1; i < 256; i++) |
| 7 | count[i] += count[i - 1]; |

# Подтверждение корректности

Для подтверждения корректности в программе в программе происходит сравнение работы результатов сортировки с встроенной в язык Си – qsort (будем считать её корректной). Функция проходится по всем значениям массива, отсортированного qsort, и массива, отсортированного одной из четырех сортировок, и сравнивает их значения. Если значения одинаковые, то массив считается отсортированным корректно, иначе – некорректно. Программа выводит соответствующее сообщение о результате сортировки, а в случае провала ещё и номер первого элемента, стоящего не на своём месте.

Проверим корректность сортировок для массива размера . Для наглядности выведем и количество тиков процессора.

Результаты проверки:

* Сортировка «пузырьком»

Количество тактов процессора – 60’613’922’834

Корректность работы – Everything is fine

* Сортировка Хоара

Количество тактов процессора – 40’515’105

Корректность работы – Everything is fine

* Сортировка слиянием

Количество тактов процессора – 29’619’241

Корректность работы – Everything is fine

* Поразрядная сортировка

Количество тактов процессора – 11’268’632

Корректность работы - Everything is fine

# Результаты экспериментов

Чтобы наглядно показать среднюю асимптотику сортировок проведём серию опытов. Для конкретной сортировки дадим несколько массивы различной длины, каждый массив отсортируем 10 раз, для каждого размера выберем минимальное значение тиков (компьютер распределяет ресурсы и на другие процессы, а значит минимальное и будет наиболее точным). Каждое полученное значение разделим на асимптотику данной сортировки, подставив вместо n соответствующий размер массива (а для поразрядной вместо k – 8). По полученным значениям построим график для каждой сортировки, который для достаточно больших значений должен стремиться к ненулевой константе.

* Сортировка «пузырьком»

Для значений в диапазоне от 10’000 до 100’000 с шагом 10’000 значение константы выравнивается при 50’000 и ≈ 5.6 (рис. 1)

Рис. 1. График отношения сортировки «пузырьком»

* Сортировка Хоара

Для значений от 1’000’000 до 10’000’000 с шагом 1’000’000 значение константы ≈ 63 (рис. 2)

Рис. 2. График отношения сортировки Хоара

* Сортировка слиянием

Для значений от 1’000’000 до 10’000’000 с шагом 1’000’000 значение отношения ведёт себя нестабильно, колеблется возле уровня ≈ 52.5 (рис. 3)

Рис. 3. График отношения сортировки слиянием

* Поразрядная сортировка

Для значений от 1e7 до 1e8 константа ≈ 15,8 (рис. 4)

Рис. 4. График отношения поразрядной сортировки

# Заключение

* В отчете написаны принципы работы сортировок, а также конкретная реализация 4 различных сортировок.
* В руководстве для пользователя описано взаимодействие пользователя и программы, а также её возможностей по анализу работы сортировок.
* Была проанализирована работа каждой сортировки, проверка её на корректность и на верность асимптотики, указанной ранее, и поиск её константы для некоторых значений длины массива (пусть и с низкой точностью), с приложением в виде графиков.

# Литература

1. Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. // Алгоритмы: построение и анализ / Под ред. И. В. Красикова. — 3-е изд. — М.: Вильямс, 2013.
2. https://prirai.github.io/code/quick-sort/

# Приложение

void sort\_cocktail\_arrlf(double\* arr, long long int n)

{

long long int i, left = 0, right = n - 1, control = 0;

while (left != right)

{

for (i = left; i < right; i++)

{

if (arr[i] > arr[i + 1])

{

swaplf((arr + i), (arr + i + 1));

control = i;

}

}

right = control;

for (i = right; i > left; i--)

{

if (arr[i] < arr[i - 1])

{

swaplf((arr + i), (arr + i - 1));

control = i;

}

}

left = control;

}

return;

}

void sort\_quick\_arrlf(double\* arr, long long int s, long long int f)

{

long long int i = s, j = f;

{

partition(arr, &i, &j);

if ((j - s) > 0)

sort\_quick\_arrlf(arr, s, j);

if ((f - i) > 0)

sort\_quick\_arrlf(arr, i, f);

}

return;

}

void partition(double\* arr, long long int\* s, long long int\* f)

{

double a = arr[\*s + rand() % (\*f - \*s + 1)];

while (\*s <= \*f)

{

while (arr[\*s] < a)

(\*s)++;

while (arr[\*f] > a)

(\*f)--;

if (\*s <= \*f)

{

swaplf((arr + \*s), (arr + \*f));

(\*s)++;

(\*f)--;

}

}

return;

}

void sort\_merge\_arrlf(double\* arr, long long int n)

{

double\* tmp\_arr;

long long int step, i;

int f = 0;

tmp\_arr = (double\*)malloc(n \* sizeof(double));

if (tmp\_arr == 0)

{

printf("Memory error! SM");

exit(0);

}

for (step = 1; step < n; step \*= 2)

{

for (i = 0; i < n; i += 2 \* step)

merge\_arrlf(arr + i, arr + i + step, tmp\_arr + i, min(step, n - i), max(0, min(step, n - step - i)));

swapp(&arr, &tmp\_arr);

f = 1 - f;

}

if (f)

{

equalize\_arrlf(arr, tmp\_arr, n);

swapp(&arr, &tmp\_arr);

}

free(tmp\_arr);

return;

}

void merge\_arrlf(double\* m1, double\* m2, double\* m3, long long int n1, long long int n2)

{

long long int i, j, k;

for (k = 0, i = 0, j = 0; k < n1 + n2; k++)

{

if (i == n1)

m3[k] = m2[j++];

else if (j == n2)

m3[k] = m1[i++];

else

{

if ((m1[i]) < m2[j])

m3[k] = m1[i++];

else m3[k] = m2[j++];

}

}

return;

}

void sort\_radix\_backward\_LSD\_arrlf(double\* arr, long long int n)

{

unsigned char\* pm = (unsigned char\*)arr;

int count[256], i, j, sgnp;

double\* arr2 = (double\*)malloc(n \* sizeof(double));

if (arr2 == 0)

{

printf("Memory error! SR");

exit(0);

}

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

{

foffset(pm, n, i, count);

for (j = n - 1; j >= 0; j--)

arr2[count[pm[j \* sizeof(double) + i]]--] = arr[j];

swapp(&arr2, &arr);

pm = (unsigned char\*)arr;

}

if (arr[n - 1] < 0)

{

i = 0, j = n - 1;

while ((j - i) > 1)

{

sgnp = (j + i) / 2;

if (arr[sgnp] >= +0)

i = sgnp;

else

j = sgnp;

}

if (arr[sgnp] >= +0)

sgnp += 1;

}

equalize\_arrlf(arr, arr2, sgnp);

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

arr2[i] = arr[n - i + sgnp - 1];

equalize\_arrlf(arr2 + sgnp, arr, n - sgnp);

equalize\_arrlf(arr2, arr + (n - sgnp), sgnp);

free(arr2);

return;

}

void foffset(unsigned char\* arr, long long int n, int digit, int count[256])

{

int i;

for (i = 0; i < 256; i++)

{

count[i] = 0;

}

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

count[arr[digit + i \* sizeof(double)]]++;

count[0] -= 1;

for (i = 1; i < 256; i++)

count[i] += count[i - 1];

}

1. Здесь и далее в границах раздела A – указатель на начало массива. n – размер массива [↑](#footnote-ref-1)