**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра информационных систем**

отчет

**по практической работе №1**

**по дисциплине «Программирование»**

Тема: Типы данных, определяемые пользователем. Структуры

|  |  |  |
| --- | --- | --- |
| Студент(ка) гр. |  | Марданова Д.М. |
| Преподаватель |  | Глущенко А. Г. |

Санкт-Петербург

2022

**Цель работы.**

Необходимо написать программу, которая:

1)    Создает целочисленный массив размерности *N* = 100. Элементы массивы должны принимать случайное значение в диапазоне от -99 до 99.2)    Отсортировать заданный в пункте 1 массив […] сортировкой (от меньшего к большему). Определить время, затраченное на сортировку, используя библиотеку chrono.3)    Найти максимальный и минимальный элемент массива. Подсчитайте время поиска этих элементов в отсортированном массиве и неотсортированном, используя библиотеку chrono.

4)    Выводит среднее значение (если необходимо, число нужно округлить) максимального и минимального значения в отсортированном и неотсортированном. Выводит индексы всех элементов, которые равны этому значению, и их количество. Подсчитайте время поиска.

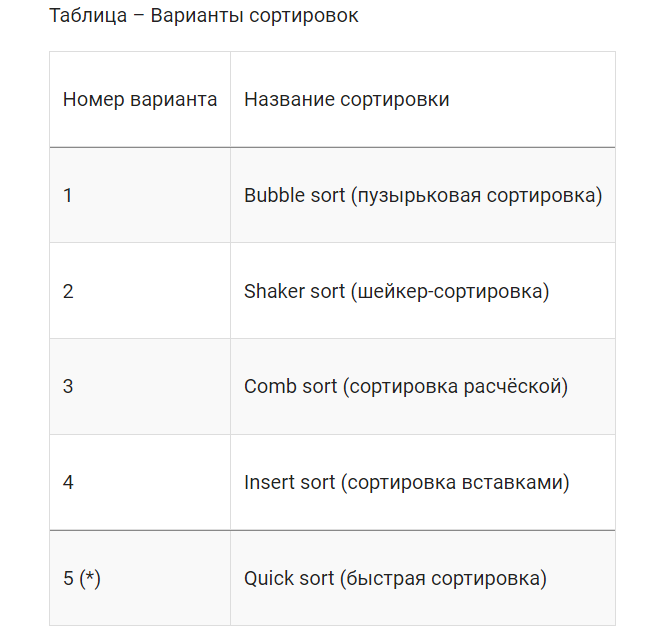
5)    Выводит количество элементов в отсортированном массиве, которые меньше числа *a*, которое инициализируется пользователем.

6)    Выводит количество элементов в отсортированном массиве, которые больше числа *b*, которое инициализируется пользователем.

7)    Выводит информацию о том, есть ли введенное пользователем число в отсортированном массиве. Реализуйте алгоритм бинарного поиска. Сравните скорость его работы с обычным перебором. (\*)

8)     Меняет местами элементы массива, индексы которых вводит польователь. Выведите скорость обмена, используя библиотеку chrono.

Должна присутствовать возможность запуска каждого пункта многократно.



**Основные теоретические положения.**

При использовании простых переменных каждой области памяти для хранения данных соответствует свое имя. Если с группой величин одинакового типа требуется выполнить однообразные действия, им дают одно имя, а различают по порядковому номеру (индексу). Это дает возможность компактно записать множество операций с использованием циклов.

Массив представляет собой индексированную последовательность однотипных элементов с заранее определенным количеством элементов. Наглядно одномерный массив можно представить, как набор пронумерованных ячеек, в каждой из которых содержится определенное значение.

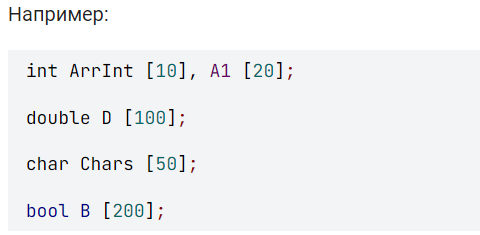
Все массивы можно разделить на две группы: одномерные и многомерные. Описание массива в программе отличается от объявления обычной переменной наличием размерности массива, которая задается в квадратных скобках после имени.

Элементы массива нумеруются с нуля. При описании массива используются те же модификаторы (класс памяти, const и инициализатор), что и для простых переменных.

Аналогом одномерного массива из математики может служить последовательность некоторых элементов с одним индексом: a\_i*ai*​ при  i = 0, 1, 2, … n – одномерный вектор. Каждый элемент такой последовательности представляет собой некоторое значение определенного типа данных.

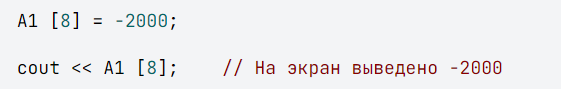
Объявление в программах одномерных массивов выполняется в соответствии со следующим правилом:

**<Базовый тип элементов> <Идентификатор массива> [<Количество элементов>]**



**Значения индексов элементов массивов всегда начинается с 0**. Поэтому максимальное значение индекса элемента в массиве всегда на единицу меньше количества элементов в массиве.

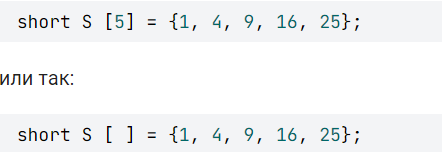
Обращение к определенному элементу массива осуществляется с помощью указания значения индекса этого элемента:



В этом примере, обратившись к элементу массива **A1** с индексом 8, мы, фактически, обратились к его 9-му элементу.

При обращении к конкретному элементу массива этот элемент можно рассматривать как обычную переменную, тип которой соответствует базовому типу элементов массива, и осуществлять со значением этого элемента любые операции, которые характерны для базового типа. Например, поскольку базовым типом массива **A1** является тип данных **int**, с любым элементом этого массива можно выполнять любые операции, которые можно выполнять над значениями типа **int**.

При объявлении массива его можно инициализировать определенными значениями:



Во втором случае мы не указываем количество элементов массива **S**. Автоматически создается массив на 5 элементов в соответствии с инициализирующими значениями.

Количество значений, указанных в фигурных скобках (инициализирующих значений) не должно превышать количества элементов в массиве (в нашем примере - 5).

Значения всех элементов массива в памяти располагаются в непрерывной области одно за другим. Общий объем памяти, выделяемый компилятором для массива, определяется как произведение объема одного элемента массива на количество элементов в массиве и равно:

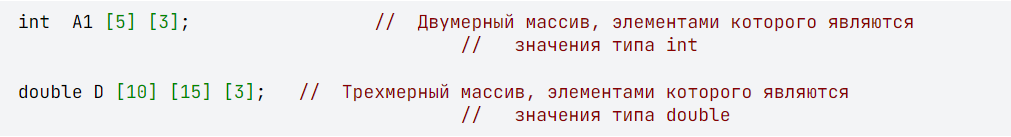
**sizeof( <Базовый тип> )  \* <Количество элементов>**

Для предыдущего примера объем массива **S** будет равен **sizeof( short) \* 5 = 2 \* 5 = 10** байтам.

Поскольку все элементы массивов располагаются в памяти один за другим без разрывов, обращение к элементам массива по их индексам (какой бы длины не был этот массив) осуществляется очень эффективно путем вычисления адреса нужного элемента. Пусть, например, адрес памяти, где начинается массив  **S,**равен **100**, тогда адрес элемента этого массива с индексом **3** будет равен **100 +** **sizeof( short) \* 3 = 100 + 2 \* 3 = 106**. Обращаемся по этому адресу и считываем **2** байта. Это и будет значением элемента с индексом **3** массива **S**.

**В языке C++ не осуществляется проверка выхода за границы массивов**. То есть, вполне корректно (с точки зрения компилятора) будет обращение к элементу массива **S**, индекс которого равен **10**. Это может привести к возникновению весьма серьезных отрицательных последствий. Например, если выполнить присвоение **S[10] = 1000** будут изменены данные, находящиеся за пределами массива, а это может быть значение какой-нибудь другой переменной программы. После этого предсказать поведение программы будет невозможно. Единственный выход – быть предельно внимательным при работе с индексами элементов массивов.

Многомерные массивы определяются аналогично одномерным массивам. Количество элементов по каждому измерению указывается отдельно в квадратных скобках:



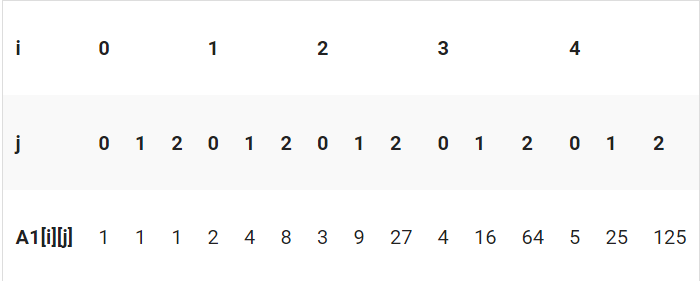
Здесь массив **A1**представляет собой обычную двумерную матрицу из 5-ти строк и 3–х столбцов.

Массив **D** – трехмерный массив, который можно представить как трехмерный параллелограмм из 3-х двумерных матриц.

Общее число элементов в многомерном массиве определяется как произведение количества элементов по каждому измерению. Так, например, массив **D** содержит **10 \* 15 \* 3 = 450**  элементов типа **double**, а объем памяти, требующийся для этого массива, будет равен **450 \* 8 = 3600** байтам.

Массивы с большим, чем 3, количеством измерений используются достаточно редко. Одной из причин этого является быстрый рост объема памяти, необходимой для размещения таких массивов.

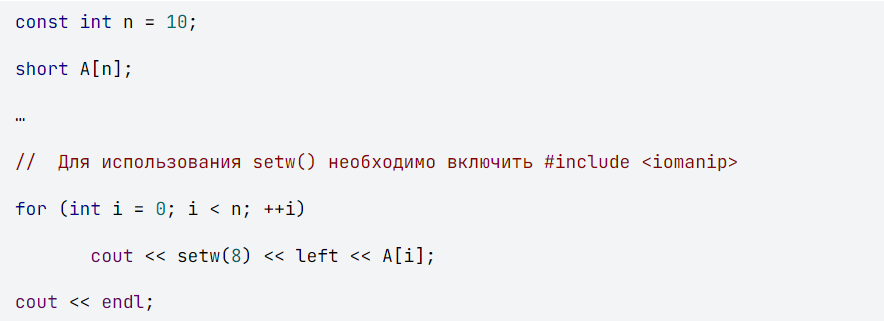
В следующей таблице показана схема размещения элементов массива **A1** в памяти:



Так же как и в одномерном массиве, элементы многомерных массивов располагаются друг за другом в непрерывном участке памяти.

Вывод массивов.

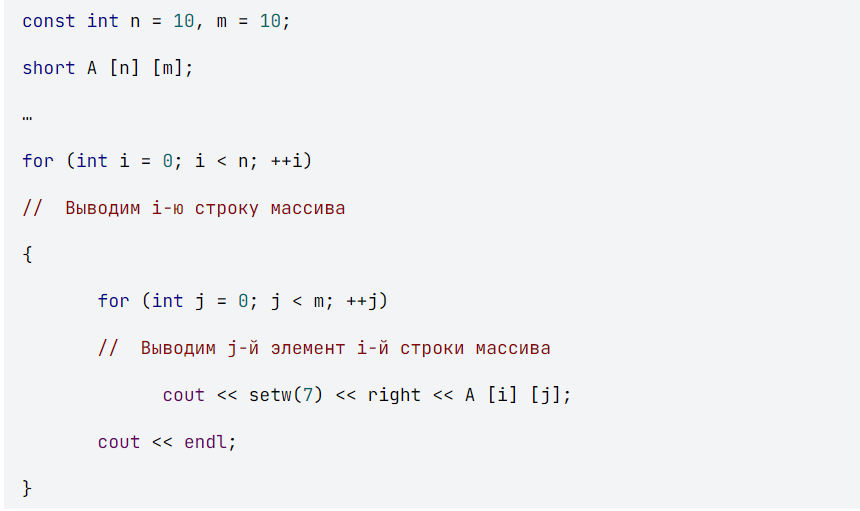
Простейший циклический алгоритм вывода значений элементов некоторого одномерного массива выглядит так:



На каждом шаге этого цикла в поток вывода отправляется очередной **i**-й элемент массива, при этомустанавливается ширина поля вывода, равная 8 позициям, выравнивание по левому краю. После окончания цикла вывода всех **n** элементов массива осуществляется переход на следующую строку экрана.

Обратим внимание на то, что в программах выгоднее задавать размеры массивов через именованные константы (в данном примере – константа **n**), для того чтобы использовать эти же константы для управления работой циклов. При необходимости изменить размеры массива достаточно будет поменять значение этой константы. При этом все циклы, использующие для управления своей работой эту константу, автоматически приспособятся к изменившимся размерам обрабатываемого массива.

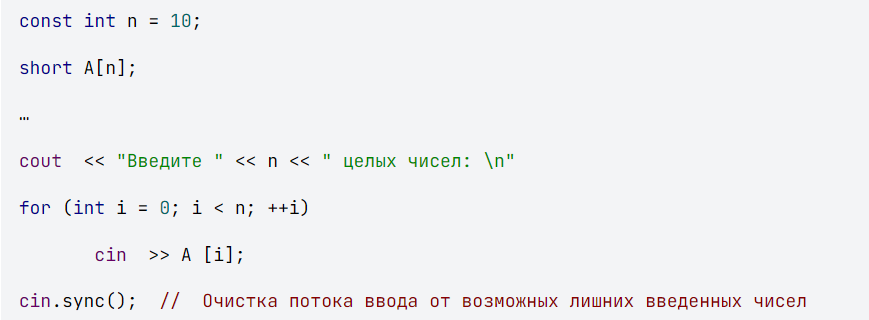
Вывод двумерных массивов, как правило, осуществляется в табличной форме. Реализация такого алгоритма может быть, например, такой:



Здесь используются вложенные циклы. Обратите внимание, что внутренний (вложенный) цикл практически идентичен циклу, реализующему вывод элементов одномерного массива.

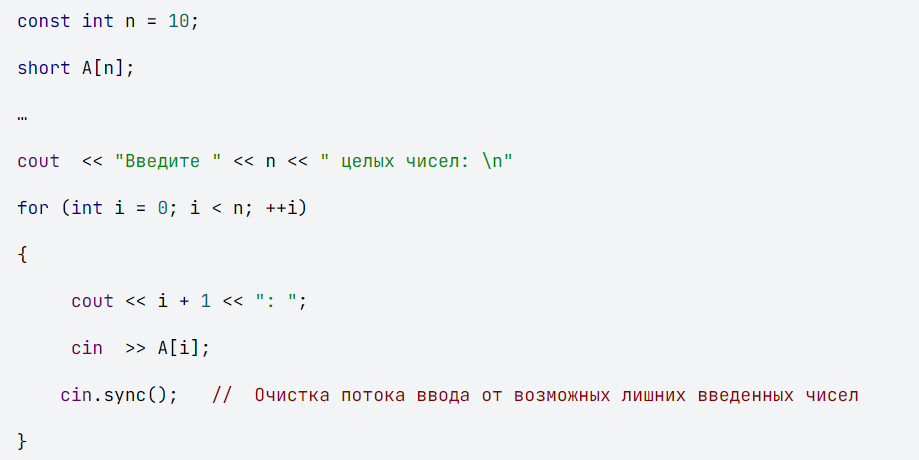
Ввод массивов.

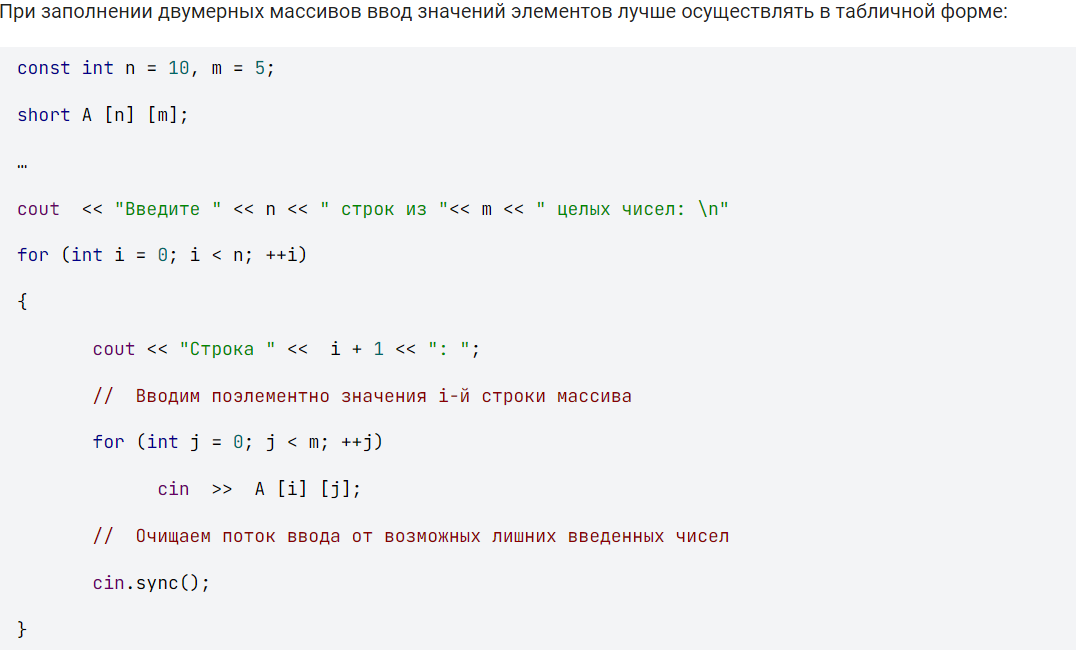
Ввод массивов также осуществляется с помощью циклических алгоритмов.

Простейший алгоритм ввода значений элементов одномерного массива может выглядеть так:

В этом фрагменте ввод значений элементов массива можно осуществлять и в виде строки целых чисел, разделенных пробелами или символами табуляции, и по одному значению с помощью клавиши **Enter**. Обратите внимание на последнюю строку этого фрагмента. Эта инструкция очищает поток ввода от оставшихся в нем данных. Если в потоке ввода после окончания цикла останутся какие-либо данные (например, мы ввели не 10 требуемых значений, а ошибочно 11 чисел), то следующие за циклом инструкции ввода возьмут их из потока, что может привести к неправильной работе программы.

Следующий фрагмент обеспечивает ввод  значений элементов массива с указанием номеров вводящихся элементов:





**Пузырьковая сортировка.**

Сортировка – процесс размещения элементов заданного множества объектов в определенном порядке. Когда элементы отсортированы, их проще найти, производить с ними различные операции. Сортировка напрямую влияет на скорость алгоритма, в котором нужно обратиться к определенному элементу массива.

Простейшая из сортировок – сортировка обменом (пузырьковая сортировка). Вся суть метода заключается в попарном сравнении элементов и последующем обмене. Таким образом, если следующий элемент меньше текущего, то они меняются местами, максимальный элемент массива постепенно смещается в конец массива, а минимальный – в начало. Один полный проход по массиву может гарантировать, что в конце массива находится максимальный элемент.

Затем процесс повторяется до тех пор, пока вся последовательность не будет упорядочена. Важно заметить, что после первого прохода по массиву, уже имеется один упорядоченный элемент, он стоит на своем месте, и менять его не надо. Таким образом на следующем шаге будут сравниваться N-1 элемент.

Очевидно, что хуже всего алгоритм будет работать, когда на вход подается массив, отсортированный в обратную сторону (от большего к меньшему). Быстрее же всего алгоритм работает с уже отсортированным массивом.

Но стандартный алгоритм пузырьковой сортировки предполагает полный циклический проход по массиву. Если изначально подается упорядоченная последовательность, то работа алгоритма все равно продолжиться. Исправить это можно, добавив условие проверки: если на текущей итерации ни один элемент не изменил свой индекс, то работа алгоритма прекращается.

**Шейкер-сортировка.**

Shaker sort – модификация пузырьковой сортировки. Принцип работы этой сортировки аналогичен bubble sort: попарное сравнение элементов и последующий обмен местами. Но имеется существенное отличие. Как только максимальный элемент становится на свое место, алгоритм не начинает новую итерацию с первого элемента, а запускает сортировку в обратную сторону. Алгоритм гарантирует, что после выполнения первой итерации, минимальный и максимальный элемент будут в начале и конце массива соответственно.

Затем процесс повторяется до тех пор, пока массив не будет отсортирован. За счет того, что сортировка работает в обе стороны, массив сортируется на порядок быстрее. Очевидным примером этого был бы случай, когда в начале массива стоит максимальный элемент, а в конце массива – минимальный. Shaker sort справится с этим за 1 итерацию, при условии, что другие элементы стоят на правильном месте.

Кажется, что bubble sort теряет свою эффективность по сравнению с shaker sort. Сортировка проходит в массиве в обоих направлениях, а не только от его начала к концу. Но в работе с большими массивами преимущество шейкер-сортировки уменьшается как раз из-за использования двух циклов.

**Сортировка массива расчёской.**

Очевидный недостаток bubble и shaker sort заключается в том, что элементы переставляются максимум на одну позицию.

Comb sort (сортировка расческой) – ещё одна модификация сортировки пузырьком. Алгоритм был разработан специально для случаев, когда минимальные элементы стоят слишком далеко, или максимальные – слишком близко к началу массива. В сортировке расческой переставляются элементы, стоящие на расстоянии.

Оптимально изначально взять расстояние равным длине массива , а далее уменьшать его на определенный коэффициент, который примерно равен 1.247. Когда расстояние станет равно 1, выполняется обычная сортировка пузырьком.

**Сортировка массива вставками.**

Сортировка вставками (insert sort) – алгоритм сортировки, в котором элементы массива просматриваются по одному, и каждый новый элемент размещается в подходящее место среди ранее упорядоченных элементов.

Общая суть сортировки вставками такова:

1)    Перебираются элементы в неотсортированной части массива.

2)    Каждый элемент вставляется в отсортированную часть массива на то место, где он должен находится.

Сортировка вставками делить массив на 2 части – отсортированную и неотсортированную. С каждым новым элементом отсортированная часть будет увеличиваться, а неотсортированная уменьшаться. Причем найти нужное место для очередного элемента в отсортированном массиве достаточно легко.

Рассмотрим самый простой способ (рис. 3.5). Необходимо пройти массив слева направо и обработать каждый элемент. Слева будет наращиваться отсортированная часть массива, а справа – уменьшаться неотсортированная. В отсортированной части массива ищется точка вставки для очередного элемента. Сам элемент отправляется в буфер, что освобождает место в массиве и позволяет сдвинуть элементы и освободить точку вставки.

**Быстрая сортировка массива.**

Быстрая сортировка (quick sort) – одна из самых быстрых сортировок. Эта сортировка по сути является существенно улучшенной версией алгоритма пузырьковой сортировки.

Общая идея алгоритма состоит в том, что сначала выбирается из массива элемент, который называется опорным. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность. Затем необходимо сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующие друг за другом: меньше опорного, раны опорному и больше опорного. Для меньших и больших значений необходимо выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

**Бинарный поиск.**

Алгоритм бинарного поиска – классический алгоритм поиска в отсортированном массиве, который использует дробление массива на половины. Если элемент, который необходимо найти, присутствует в списке, то бинарный поиск возвращает ту позицию, в которой он был найден.

Рассмотрим простой пример: имеется массив из 100 элементов, упорядоченных по возрастанию от 1 до 100. Было загадано какое-то число, необходимо его назвать. Компьютер имеет три ответа на ваше предположение: верно, число больше, число меньше. Сколько попыток нужно, чтобы ответить правильно? Обычный перебор – наихудшая стратегия. Можно назвать правильный ответ лишь с 100-ой попытки.

Но если начать спрашивать с середины, то ситуация кардинально меняется. Если число больше 50, то необходимо делить правую половину, и следующее предположение – 75, если меньше – 25. Так необходимо продолжать до тех пор, пока не будет названо правильное число. Наибольшее число предположений равняется:

k= ⌈ log\_2 100⌉=7*k*=⌈*log*2​100⌉=7

Этот результат явно лучше простого перебора. Схожий принцип работы имеет алгоритм бинарного поиска.

Бинарный поиск работает только в топ случае, если массив отсортирован. Например, если бы искомое минимальное значение стояло не на своем положенном месте, а на месте максимального элемента, то мы бы откинули его на первой же итерации. Сам алгоритм имеет вид:

1)    Определение значения в середине массива (или иной структуры данных). Полученное значение сравнивается с ключом (значением, которое необходимо найти).

2)    Если ключ меньше значения середины, то необходимо осуществлять поиск в первой половине элементов, иначе – во второй.

3)    Поиск сводится к тому, что вновь определяется значение серединного элемента в выбранной половине и сравнивается с ключом.

4)    Процесс продолжается до тех пор, пока не будет определен элемент, равный значению ключа или не станет пустым интервал для поиска.

Чтобы уменьшить количество шагов поиска, можно сразу смещать границы поиска на элемент, следующий за серединой отрезка.

Предположим, вы ищете слово в словаре с 240 ООО словами. Как вы думаете, сколько попыток вам понадобится в худшем случае?

При простом поиске может потребоваться 240 ООО попыток, если искомое слово находится на самой последней позиции в книге. С каждым шагом бинарного поиска количество слов сокращается вдвое, пока не останется ТОЛЬКО ОДНО СЛОВО.

**Постановка задачи.**

Необходимо написать программу, которая:

1)    Создает целочисленный массив размерности *N* = 100. Элементы массивы должны принимать случайное значение в диапазоне от -99 до 99.2)    Отсортировать заданный в пункте 1 массив […] сортировкой (от меньшего к большему). Определить время, затраченное на сортировку, используя библиотеку chrono.3)    Найти максимальный и минимальный элемент массива. Подсчитайте время поиска этих элементов в отсортированном массиве и неотсортированном, используя библиотеку chrono.

4)    Выводит среднее значение (если необходимо, число нужно округлить) максимального и минимального значения в отсортированном и неотсортированном. Выводит индексы всех элементов, которые равны этому значению, и их количество. Подсчитайте время поиска.

5)    Выводит количество элементов в отсортированном массиве, которые меньше числа *a*, которое инициализируется пользователем.

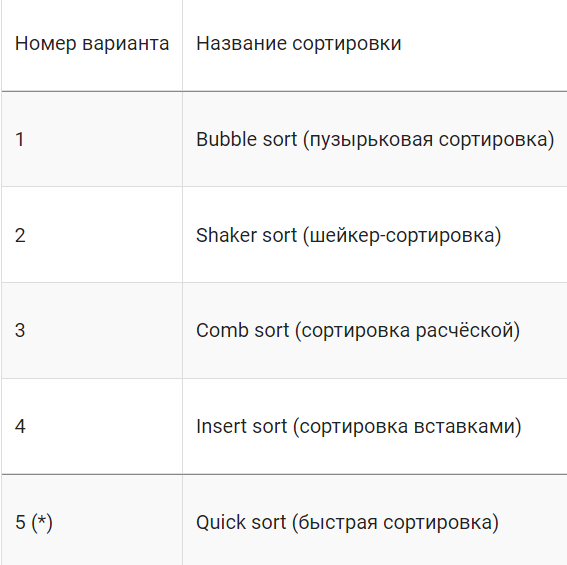
6)    Выводит количество элементов в отсортированном массиве, которые больше числа *b*, которое инициализируется пользователем.

7)    Выводит информацию о том, есть ли введенное пользователем число в отсортированном массиве. Реализуйте алгоритм бинарного поиска. Сравните скорость его работы с обычным перебором. (\*)

8)     Меняет местами элементы массива, индексы которых вводит польователь. Выведите скорость обмена, используя библиотеку chrono.

Должна присутствовать возможность запуска каждого пункта многократно.

Таблица – Варианты сортировок



**Выполнение работы.**

Код программы представлен в приложении.

**Выводы.**

В ходе работы были изучены одномерные статические массивы и методы работы с ними. Изучены алгоритмы сортировки массивов, алгоритм бинарного поиска.

Приложение

рабочий код

#include <chrono>

#include <iostream>

#include <ctime>

#include <cstdlib>

using namespace std;

using namespace chrono;

void quicksort(int\* arr, int endn, int begin)

{

int mid;

int f = begin;

int l = endn;

mid = arr[(f + l) / 2];

while (f < l)

{

while (arr[f] < mid) f++;

while (arr[l] > mid) l--;

if (f <= l)

{

swap(arr[f], arr[l]);

f++;

l--;

}

}

if (begin < l) quicksort(arr, l, begin);

if (f < endn) quicksort(arr, endn, f);

}

int main()

{

srand(time(0));

setlocale(LC\_ALL, "Russian");

auto start = steady\_clock::now();

cout << endl;

auto end = steady\_clock::now();

auto result = duration\_cast<nanoseconds>(end - start);

cout << "Время: " << result.count() << " ns";

cout << "\n\n";

cout << "Входной массив:\n";

const int n = 100;

int a[n], a\_sort[n];

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

{

a[i] = rand() % 199 - 99;

a\_sort[i] = a[i];

cout << a[i] << " ";

}

cout << "\n\nBubble sort";

cout << endl;

start = steady\_clock::now();

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

bool flag = true;

for (int j = 0; j < n - i - 1; j++)

if (a\_sort[j] > a\_sort[j + 1]) {

flag = false;

swap(a\_sort[j], a\_sort[j + 1]);

}

if (flag)

break;

}

end = steady\_clock::now();

result = duration\_cast<nanoseconds>(end - start);

cout << "\nВремя: " << result.count() << " ns";

cout << "\nОтсортированный массив:\n";

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

cout << a\_sort[i] << " ";

a\_sort[i] = a[i];

}

cout << "\n\nShaker sort";

cout << endl;

start = steady\_clock::now();

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

bool flag = true;

for (int j = 0; j < n - 1 - i; j++)

if (a\_sort[j] > a\_sort[j + 1]) {

flag = false;

swap(a\_sort[j], a\_sort[j + 1]);

}

for (int j = n - 2 - i; j > i; j--)

if (a\_sort[j] < a\_sort[j - 1]) {

flag = false;

swap(a\_sort[j], a\_sort[j - 1]);

}

if (flag)

break;

}

end = steady\_clock::now();

result = duration\_cast<nanoseconds>(end - start);

cout << "\nВремя: " << result.count() << " ns";

cout << "\nОтсортированный массив:\n";

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

cout << a\_sort[i] << " ";

a\_sort[i] = a[i];

}

cout << "\n\nComb sort";

cout << endl;

start = steady\_clock::now();

int count = 0;

float k = 1.247, S = n;

while (k < S)

{

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

{

if (a\_sort[i] > a\_sort[int(i + k)])

swap(a\_sort[i], a\_sort[int(i + k)]);

}

S /= k;

}

while (true)

{

for (int i = 0; i < n - 1; i++)

{

if (a\_sort[i] > a\_sort[i + 1])

{

swap(a\_sort[i], a\_sort[i + 1]);

}

else count++;

}

if (count == n - 1)

break;

else

count = 0;

}

end = steady\_clock::now();

result = duration\_cast<nanoseconds>(end - start);

cout << "\nВремя: " << result.count() << " ns";

cout << "\nОтсортированный массив:\n";

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

cout << a\_sort[i] << " ";

a\_sort[i] = a[i];

}

cout << "\n\nInsert sort";

cout << endl;

start = steady\_clock::now();

int x;

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

{

x = a\_sort[i];

int j = i;

while ((j > 0) && (x < a\_sort[j - 1]))

{

a\_sort[j] = a\_sort[j - 1];

j--;

}

a\_sort[j] = x;

}

end = steady\_clock::now();

result = duration\_cast<nanoseconds>(end - start);

cout << "\nВремя: " << result.count() << " ns";

cout << "\nОтсортированный массив:\n";

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

cout << a\_sort[i] << " ";

}

cout << "\n\nМаксимальный и минимальный элементы";

cout << endl;

// ищем max и min для неотсортированного массива

int max = a[0], min = a[0];

start = steady\_clock::now();

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

{

max = a[i] > max ? a[i] : max;

min = a[i] < min ? a[i] : min;

}

end = steady\_clock::now();

result = duration\_cast<nanoseconds>(end - start);

cout << endl;

cout << "\nМаксимальный элемент: " << max;

cout << "\nМинимальный элемент: " << min;

cout << "\nВремя (неотсортированный массив): " << result.count() << " ns";

// ищем max и min для отсортированного массива

start = steady\_clock::now();

int max1 = a\_sort[0], min1 = a\_sort[99];

end = steady\_clock::now();

result = duration\_cast<nanoseconds>(end - start);

cout << "\nВремя (отсортированный массив): " << result.count() << " ns";

// сред значение max и min

float m, m1;

m = float((min + max) / 2);

cout << "\nСреднее значение max и min в неотсортированном массиве: " << m;

m1 = float((min1 + max1) / 2);

cout << "\nСреднее значение max и min в отсортированном массиве: " << m1;

//Выводит индексы всех элементов, которые равны этому значению, и их количество.

cout << "\n\nИндексы элементов, которые равны среднему значению:";

start = steady\_clock::now();

int count3 = 0;

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

if (a[i] == int(m)) {

cout << i << " ";

count3 += 1;

}

end = steady\_clock::now();

result = duration\_cast<nanoseconds>(end - start);

cout << endl << "Количество элементов:" << count3;

cout << "\nВремя в неотсортированном: " << result.count() << " ns";

start = steady\_clock::now();

int count4 = 0;

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

if (a\_sort[i] == int(m)) {

cout << i << " ";

count4 += 1;

}

end = steady\_clock::now();

result = duration\_cast<nanoseconds>(end - start);

cout << endl << "Количество элементов:" << count4;

cout << "\nВремя в неотсортированном: " << result.count() << " ns";

//Выводит количество элементов в отсортированном массиве, которые меньше числа a, которое инициализируется пользователем.

cout << "\n\nВведите число a: ";

int an, count1 = 0;

cin >> an;

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

if (a\_sort[i] < an)

count1 += 1;

cout << "Количество элементов меньше а: " << count1;

//Выводит количество элементов в отсортированном массиве, которые больше числа b, которое инициализируется пользователем.

cout << "\nВведите число b: ";

int b, count2 = 0;

cin >> b;

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

if (a\_sort[i] > b)

count2 += 1;

cout << "Количество элементов больше b: " << count2;

cout << endl;

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

a\_sort[i] = a[i];

//\*

cout << "\n\nQuick sort";

cout << endl;

start = steady\_clock::now();

int endn = n - 1, begin = 0;

quicksort(a\_sort, endn, begin);

end = steady\_clock::now();

result = duration\_cast<nanoseconds>(end - start);

cout << "\nВремя: " << result.count() << " ns";

cout << endl << "Отсортированный массив: ";

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

cout << a\_sort[i] << " ";

//Выводит информацию о том, есть ли введенное пользователем число в отсортированном массиве.

cout << "\n\nБинарный поиск";

cout << "\nВведите число: ";

int key;

cin >> key;

bool flag = false;

int l = 0;

int r = n - 1;

int mid;

start = steady\_clock::now();

while ((l <= r) && (flag != true))

{

mid = (l + r) / 2;

if (a\_sort[mid] == key)

flag = true;

if (a\_sort[mid] > key)

r = mid - 1;

else l = mid + 1;

}

end = steady\_clock::now();

result = duration\_cast<nanoseconds>(end - start);

if (flag)

cout << "\nЧисло есть в массиве";

else

cout << "\nТакого элемента в массиве нет";

cout << "\nВремя: " << result.count() << " ns";

cout << "\n\nОбычный перебор";

cout << "\nВведите число: ";

int key1;

cin >> key1;

bool flag1 = false;

start = steady\_clock::now();

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

if (a[i] == key1)

flag1 = true;

else

flag1 = false;

}

end = steady\_clock::now();

result = duration\_cast<nanoseconds>(end - start);

cout << "\nВремя: " << result.count() << " ns";

if (flag1)

cout << "\nЧисло есть в массиве";

else

cout << "\nТакого элемента в массиве нет";

//Меняет местами элементы массива, индексы которых вводит пользователь.

cout << "\n\nИсходный массив: ";

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

cout << a[i] << " ";

cout << "\nВведите 2 индекса: ";

int c, d;

cin >> c >> d;

cout << "\nИзмененный массив: ";

start = steady\_clock::now();

int swap = a[c];

a[c] = a[d];

a[d] = swap;

end = steady\_clock::now();

result = duration\_cast<nanoseconds>(end - start);

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

cout << a[i] << " ";

}

cout << "\nВремя: " << result.count() << " ns\n"; //Cкорость обмена

system("pause");

return 0;

}