МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий

Кафедра Программной инженерии

Специальность 1-40 01 01 Программное обеспечение информационных технологий

**РЕФЕРАТ**

на тему:

«Анализ видов сортировки данных используя язык С++»

Выполнил:

студент I курса 6 группы

специальности ПОИТ Шимко А.А

Преподаватель:

Белодед Николай Иванович

**Оглавление**

Введение.......................................................................................................................3

Описание задачи..........................................................................................................5

Исходный код...............................................................................................................7

Блок-схемы.................................................................................................................12

Скриншоты работы программы...............................................................................14

Графики зависимости................................................................................................16

Вывод..........................................................................................................................17

**Введение**

**Алгоритм сортировки** — это [алгоритм](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) для упорядочивания элементов в массиве. В случае, когда элемент в массиве имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма.

Формально выходные данные любого алгоритма сортировки должны удовлетворять двум условиям:

1. Вывод осуществляется в монотонном порядке (каждый элемент не меньше/не больше предыдущего элемента, в соответствии с требуемым порядком).
2. Результат представляет собой перестановку (изменение порядка, но с сохранением всех исходных элементов) входных данных.

|  |  |
| --- | --- |
| С самого начала вычислительной техники проблема сортировки привлекала большое внимание исследователей, возможно, из-за сложности ее эффективного решения, несмотря на простую, знакомую формулировку. Среди авторов ранних алгоритмов сортировки примерно в 1951 году была Бетти Холбертон, работавшая над ENIAC и UNIVAC. |  |

Сложность алгоритмов обычно оценивают по времени выполнения или по используемой памяти. В обоих случаях сложность зависит от размеров входных данных: массив из 100 элементов будет обработан быстрее, чем аналогичный из 1000.

**Время** — основной параметр, характеризующий быстродействие алгоритма. Называется также вычислительной сложностью. Для упорядочения важны худшее, среднее и лучшее поведение алгоритма в терминах мощности входного массива.

**Память** — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. Как правило, эти алгоритмы требуют O(log n) памяти, так как работают на основе сравнений.

**O(n) — линейная сложность**

Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в не отсортированном массиве. Нам придётся пройтись по всем n элементам массива, чтобы понять, какой из них максимальный.

**O(log n) — логарифмическая сложность**

Простейший пример — бинарный поиск. Если массив отсортирован, мы можем проверить, есть ли в нём какое-то конкретное значение, методом деления пополам. Проверим средний элемент, если он больше искомого, то отбросим вторую половину массива — там его точно нет. Если же меньше, то наоборот — отбросим начальную половину. И так будем продолжать делить пополам, в итоге проверим log n элементов.

**O(n2) — квадратичная сложность**

Такую сложность имеет, например, алгоритм сортировки вставками. В канонической реализации он представляет из себя два вложенных цикла: один, чтобы проходить по всему массиву, а второй, чтобы находить место очередному элементу в уже отсортированной части. Таким образом, количество операций будет зависеть от размера массива как n \* n, т. е. n2.

Бывают и другие оценки по сложности, но все они основаны на том же принципе.

Также случается, что время работы алгоритма вообще не зависит от размера входных данных. Тогда сложность обозначают как O(1). Например, для определения значения третьего элемента массива не нужно ни запоминать элементы, ни проходить по ним сколько-то раз. Всегда нужно просто дождаться в потоке входных данных третий элемент и это будет результатом, на вычисление которого для любого количества данных нужно одно и то же время.

Аналогично проводят оценку и по памяти, когда это важно. Однако алгоритмы могут использовать значительно больше памяти при увеличении размера входных данных, чем другие, но зато работать быстрее. И наоборот. Это помогает выбирать оптимальные пути решения задач исходя из текущих условий и требований.

**Описание задачи**

Задачей данного реферата является рассмотреть следующие вопросы:

* Зависимость времени от количества элементов.
* Зависимость времени от типа элементов.
* Зависимость времени от диапазона значений элементов.
* Сложность алгоритмов по нотации О большое.

|  |  |  |  |
| --- | --- | --- | --- |
| Сортировка | Описание | Время в худшем случае | Время в среднем |
| Сортировка пузырьком | Проходит по массиву, сравнивает последовательные пары элементов и меняет их местами, если они расположены в неправильном порядке. | O(n^2) | O(n^2) |
| Шейкерная сортировка | Двунаправленный, оптимизированный вариант сортировки пузырьком | O(n^2) | O(n^2) |
| Сортировка расчёской | Модификация сортировки пузырьком, в которой расстояние между сравниваемыми парами значений отлично от 1 | O(n^2) | O(n^2/2^p) |
| Сортировка вставками | Элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов | O(n^2) | O(n^2) |
| Сортировка выбором | Делит входной массив на упорядоченную и неупорядоченную части. Затем последовательно переносит в первую часть наименьшие элементы из второй | O(n^2) | O(n^2) |
| Быстрая сортировка | Выбирается опорный элемент p. Все элементы, меньшие, либо равные p перемещаются, влево от него, а все элементы, большие, либо равные p вправо. Далее алгоритм рекурсивно применяется к каждой из частей | O(n^2) | O(n\*log(n)) |

А также построить графики, отражающие зависимость времени от количество обрабатываемых элементов. К каждому виду прикрепить алгоритм, представленный в виде блок схемы. Также стоит отметить, что массив для сортировки каждый раз будет генерироваться случайный, что не даёт нам возможности говорить о степени его неупорядоченности.

**Исходный код**

#include <iostream>

#include <ctime>

#include <time.h>

using namespace std;

void bubble\_sort(int\* A, int size)//сортировка пузырьком. передается указатель на первый элемент массива и его размер

{

int\* arr = new int[size];// инициализация нового динамического массива

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

{

\*(arr + i) = \*(A + i);

}

for (int i = 0; i < size; i++)// size - 1 необходимо, чтобы не выйти за память, отведенную под массив

{

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

{

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

swap(arr[j], arr[j + 1]);//функция swap() меняет местами значения двух переменных

}

}

cout << "A = ";

for (int i = 0; i < size; i++)//вывод отсортированного массива

{

cout << arr[i] << " ";

}

cout << endl;

delete[] arr; // освобождение памяти

}

void сocktail\_sort(int\* A, int size)//шейкер-сортировка

{

int\* arr = new int[size];// инициализация нового динамического массива

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

{

\*(arr + i) = \*(A + i);

}

bool sort\_or\_not = true;

int right\_border, left\_border = 0;

right\_border = size - 1;

do

{

//следует два пузырька, один сортирует с начала, другой - с конца

// если перестановок не произошло, то массив массив отсортирован, значит, sort\_or\_not = true - завершение do...while цикла

sort\_or\_not = true;

for (int i = left\_border; i < right\_border; i++)

{

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

{

swap(arr[i], arr[i + 1]);

sort\_or\_not = false;

}

}

for (int i = right\_border; i > left\_border; i--)

{

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

{

swap(arr[i], arr[i - 1]);

sort\_or\_not = false;

}

}

++left\_border;

--right\_border;

} while (sort\_or\_not == false);

cout << "B = ";

for (int i = 0; i < size; i++)//вывод отсортированного массива

{

cout << arr[i] << " ";

}

cout << endl;

delete[] arr;

}

void comb\_sort(int\* A, int size)//сортировка расческой

{

double factor = 1.2473309;//фактор уменьшения шага

int step = size - 1;//шаг сортировки

int\* arr = new int[size];// инициализация нового динамического массива

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

{

\*(arr + i) = \*(A + i);

}

while (step >= 1)

{

for (int i = 0; i + step < size; i++)

{

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

{

swap(arr[i], arr[i + step]);

}

}

step /= factor;

}

cout << "C = ";

for (int i = 0; i < size; i++)//вывод отсортированного массива

{

cout << arr[i] << " ";

}

cout << endl;

delete[] arr;

}

void insertion\_sort(int\* A, int size)// сортировка вставками

{

int\* arr = new int[size];// инициализация нового динамического массива

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

{

\*(arr + i) = \*(A + i);

}

int temp, // временная переменная для хранения значения элемента сортируемого массива

item;// переменная, хранящая в себе индекс предыдущего элемента

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

{

temp = arr[i]; // инициализируем временную переменную текущим значением элемента массива

item = i - 1; // запоминаем индекс предыдущего элемента массива

while (item >= 0 && arr[item] > temp) // пока индекс не равен 0 и предыдущий элемент массива больше текущего

{

arr[item + 1] = arr[item]; // перестановка элементов массива

arr[item] = temp;

item--;

}

}

cout << "D = ";

for (int i = 0; i < size; i++)//вывод отсортированного массива

{

cout << arr[i] << " ";

}

cout << endl;

delete[] arr;

}

void selection\_sort(int\* A, int size)// сортировка выбором

{

int\* arr = new int[size];// инициализыция нового динамического массива

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

{

\*(arr + i) = \*(A + i);

}

for (int i = 0; i < size - 1; i++)// size - 1 необходимо, чтобы не выйти за память, отведенную под массив

{

for (int j = i + 1; j < size; j++)

{

if (arr[i] > arr[j])

swap(arr[i], arr[j]);//функция swap() меняет местами значения двух переменных

}

}

cout << "E = ";

for (int i = 0; i < size; i++)//вывод отсортированного массива

{

cout << arr[i] << " ";

}

cout << endl;

delete[] arr; // освобождение памяти

}

void q\_sort(int\* arr, int left, int right)

{

int i = left,//левая граница

j = right;// правая граница

int pivot = arr[(left + right) / 2]; // разрешающий элемент

while (i <= j)

{

while (arr[i] < pivot) i++; // сдвигаем левую границу пока элемент [left] меньше [pivot]

while (arr[j] > pivot) j--;// сдвигаем правую границу пока элемент [right] больше [pivot]

if (i <= j)

{

if (arr[i] > arr[j])

{

swap(arr[i], arr[j]);

}

i++; j--;

}

};

// Рекурсивно вызываем сортировку для левой и правой части массива

if (left < j) q\_sort(arr, left, j);

if (i < right) q\_sort(arr, i, right);

}

int main()

{

setlocale(LC\_ALL, "ru");

srand(time(NULL)); // для того, чтобы не поучался один и то же массив

int size;

clock\_t time\_start, time\_end;

double bubble\_time, insertion\_time, cocktail\_time, comb\_time, selection\_time, qsort\_time;

cout << "Введите количество элементов массива ";

cin >> size;

int\* arr = new int[size];// в отдельную функцию выносить не нужно

cout << "arr = ";

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

{

arr[i] = rand() % 100 + 1;// задает диапазон от 1 до 100

cout << arr[i] << " ";

}

cout << endl;

time\_start = clock();

bubble\_sort(arr, size);

time\_end = clock();

bubble\_time = ((double)time\_end - (double)time\_start) / (double)CLOCKS\_PER\_SEC;//вычисление времени сортировки

time\_start = clock();

insertion\_sort(arr, size);

time\_end = clock();

insertion\_time = ((double)time\_end - (double)time\_start) / (double)CLOCKS\_PER\_SEC;

time\_start = clock();

сocktail\_sort(arr, size);

time\_end = clock();

cocktail\_time = ((double)time\_end - (double)time\_start) / (double)CLOCKS\_PER\_SEC;

time\_start = clock();

comb\_sort(arr, size);

time\_end = clock();

comb\_time = ((double)time\_end - (double)time\_start) / (double)CLOCKS\_PER\_SEC;

time\_start = clock();

selection\_sort(arr, size);

time\_end = clock();

selection\_time = ((double)time\_end - (double)time\_start) / (double)CLOCKS\_PER\_SEC;

time\_start = clock();

q\_sort(arr, 0, size - 1);

time\_end = clock();

qsort\_time = ((double)time\_end - (double)time\_start) / (double)CLOCKS\_PER\_SEC;

cout << "F = ";

for (int i = 0; i < size; i++)//вывод отсортированного массива

{

cout << arr[i] << " ";

}

cout << endl;

delete[] arr; // освобождение памяти

cout << "Время вополнения\nпузырек = " << bubble\_time << endl <<

"Вставки = " << insertion\_time << endl << "Шейкер = " << cocktail\_time << endl <<

"Расческа = " << comb\_time << endl << "Выбор = " << selection\_time <<

endl << "Быстрая = " << qsort\_time;

}

**Блок-схемы**

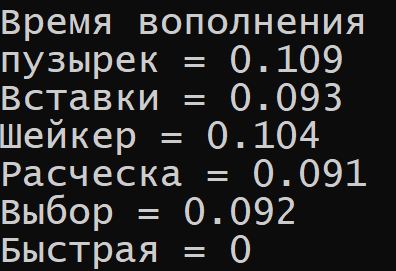
|  |  |  |
| --- | --- | --- |
| **Пузырьковая** | **Шейкерная** | **Расческа** |
|  |  |  |
| **Вставками** | **Выбором** | **Быстрая** |
|  |  |  |

**Скриншот работы программы**

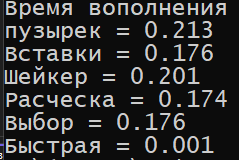
1. Проверка на 1000 элементов:



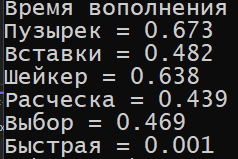
Скриншот после выполнения программы :



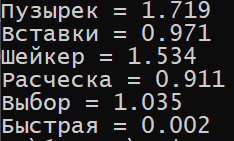
1. Проверка на 2000 элементов:

=

1. Проверка на 5000 элементов:



1. Проверка на 10000 элементов:



Следующим этапом будет составить графики зависимости обработанных элементов от времени исходя из полученных результатов.

**Графики зависимости**

**Вывод**

После анализа графиков выше можно сказать, что тип данных слабо влияет на время сортировки массива. Большую роль играет диапазон значений у массива: чем он больше, тем больше времени требуется на сортировку. Особенно “чувствительным” алгоритмом сортировки к диапазону значений оказался алгоритм сортировки выбором. Он заметно улучшил результаты по времени при уменьшении диапазона.

Самыми неэффективными алгоритмами сортировки оказались пузырьковая и вставкой, самыми эффективными – быстрая и сортировка расческой.

По нотации О большое, все сортировки, кроме быстрой и сортировки расческой, должны были показывать один и тот же результат, но особенность этой нотации в том, что в ней опускаются константные коэффициенты. Однако, несмотря на разные коэффициенты, эти сортировки растут по формуле O(n^2). В этом и заключается основная польза этой нотации: она не показывает, сколько времени будет затрачиваться алгоритмом при n-ом количестве элементов, а показывает скорость роста времени, которое потребуется алгоритму.