МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ

УНИВЕРСИТЕТ»

АЛГОРИТМЫ СОРТИРОВОК МАССИВА

Научная секция «Алгоритмы и структуры данных»

Подшиваленко Диана Игоревна,

студентка I курса

факультета информационных технологий

Научный руководитель:

Шиман Дмитрий Васильевич,

доцент, кандидат технических наук БГТУ

Минск, 2024

**Реферат**

Работа \_\_ с., 2 ч., \_\_ рис., \_\_ табл., \_\_ источников, \_ прил. МАССИВ, АЛГОРИТМ, СОРТИРОВКА, ВЫБОР, СЛОЖНОСТЬ, СКОРОСТЬ, ВСТАВКА, ВРЕМЯ, ОЦЕНКА.

Объектом исследования являются алгоритмы сортировки элементов в массиве.

Цель работы – теоретическое исследование алгоритмов сортировок; применение результатов теоретического исследования для реализации данных алгоритмов на языке C++.

По результатам теоретических исследований, используя язык программирования С++, реализовать алгоритмы сортировок.

Реализованные в С++ алгоритмы помогают понять сущность методов сортировки массивов для последующего выбора наилучшего.

Содержание

**Введение**

Проблема сортировки данных, является одной из наиболее актуальных и сложных для многих областей науки и технологий. Сортировка является одной из основных операций во многих приложениях и системах, и эффективные алгоритмы сортировки могут значительно повысить производительность программы. Каждый алгоритм сортировки обладает своими уникальными особенностями и преимуществами, которые могут быть ценными в разных ситуациях. Разработка алгоритмов не стоит на месте, а продолжает развиваться, и появляются новые, которые стремятся улучшить производительность, приспособиться к новым требованиям и решать специфические задачи.

В данной работе были поставлены цели проанализировать основные алгоритмы сортировки и реализовать их на языке С++.

**1 Аналитический обзор алгоритмов сортировок**

**1.1** **Пузырьковая сортировка**

Сортировка пузырьком — является одним из самых известных алгоритмов сортировки. Его суть заключается в последовательном сравнении значений двух соседних элементов слева направо: если предыдущее больше последующего, то производится обмен элементов местами. При сортировке элементов по убыванию в конец списка должны передвигаться элементы с наименьшим значением.

Данный алгоритм очень хорошо подходит для обучения, так как не сложен для понимания, и общего представления о работе алгоритма сортировки, однако почти не применяется на практике из-за низкой эффективности, так как он медленно работает. Но несмотря на это, на нём основаны многие другие методы сортировок.

Сложность данного алгоритма:

Лучшая временная сложность: O(n)

Средняя временная сложность: O(n2)

Худшая временная сложность: O(n2)

Затраты памяти: O(1)

**1.2** **Сортировка выбором**

В данном алгоритме на каждой итерации цикла осуществляется поиск минимального(максимального) элемента из еще неотсортированных. Далее происходит обмен местами найденного наименьшего(наибольшего) элемента с элементом, который стоит в начале (в конце) еще неотсортированной части.

Алгоритм также прост в реализации и может быть полезен для сортировки небольших массивов.

Сложность данного алгоритма:

Временная сложность: O(n^2)

Затраты памяти: O(n) основной, O(1) вспомогательной.

**1.3** **Сортировка вставками**

Это сортировка, при которой массив постепенно перебирается слева направо. При этом текущий элемент сравнивается со всеми предыдущими элементами и располагается так, чтобы оказаться в нужном месте среди уже отсортированных элементов. Это происходит до того момента, пока не будут пересмотрены все элементы массива.

Алгоритм неплохо показывает себя на небольших наборах данных, прост для понимания и реализации.

Сложность данного алгоритма:

Лучшее время: O(n)

Среднее время: O(n2)

Худшее время: O(n2)

Затраты памяти: O(n) основной, O(1) вспомогательной.

**1.4** **Шейкерная сортировка (сортировка перемешивания)**

Шейкерная сортировка является улучшением пузырьковой сортировки. Отличается от нее тем, что она двунаправленная: алгоритм перемещается не строго слева направо, а сначала слева направо, затем справа налево.

Сложность данного алгоритма:

Лучшее время: O(n)

Среднее время: O(n2)

Худшее время: O(n2)

Затраты памяти: O(1)

**1.5** **Сортировка расческой**

Данная сортировка также является улучшенной версией пузырьковой сортировки. Она получила свое название из-за схожего поведения с расческой для волос. Этот алгоритм сортировки работает путем сравнения и обмена элементов на определенном расстоянии. Первоначальный разрыв нужно выбирать не случайно, а с учётом специальной величины — фактора уменьшения, оптимальное значение которого равно 1,247. Сначала расстояние между элементами будет равняться размеру массива, поделённому на 1,247; на каждом следующем шаге расстояние будет снова делиться на фактор уменьшения — и так до окончания работы алгоритма.

Сложность данного алгоритма:

Лучшее время: O(n\*log(n))

Среднее время: O(n2 / 2p)

Худшее время: O(n2)

Затраты памяти: O(1)

**1.6** **Сортировка Шелла**

Сортировка Шелла является улучшенной версией сортировки вставками, которая разбивает массив на подмассивы и сортирует их независимо друг от друга, а затем соединяет их для окончательной сортировки. В отличие от сортировки вставками, сортировка Шелла сравнивает элементы, которые находятся на определенном расстоянии. Шаг уменьшается по мере продвижения алгоритма, что позволяет более эффективно перемещать элементы и уменьшить количество обменов. Когда шаг становится равен 1, массив просто сортируется вставками.

Сложность данного алгоритма:

Лучшее время: O(n\*log(n))

Среднее время: O(n\*log(n))

Худшее время: O(n2)

Затраты памяти: O(1)

**1.7** **Сортировка слиянием**

В основе алгоритма данной сортировки лежит принцип "разделяй и властвуй". Он разбивает массив на подмассивы, рекурсивно сортирует каждый из них, а затем объединяет их в один отсортированный массив. Базовый случай рекурсии, когда размер массива равен 1.

Сложность данного алгоритма:

Лучшее время: O(n\*log(n))

Среднее время: O(n\*log(n))

Худшее время: O(n\*log(n))

Затраты памяти: O(n)

**1.8** **Быстрая сортировка**

Быстрая сортировка также считается одним из самых эффективных алгоритмов сортировки, который, как и сортировка слиянием, использует принцип "разделяй и властвуй". Он основан на выборе опорного элемента, который разделяет массив на две части – элементы, значение которых меньше опорного, и элементы, значение которых больше опорного. Далее происходит рекурсивная сортировка каждой из этих частей.

Сложность данного алгоритма:

Лучшее время: O(n)

Среднее время: O(n\*log(n))

Худшее время: O(n^2)

Затраты памяти: O(n)

**1.9** **Пирамидальная сортировка**

Пирамидальная сортировка (сортировка кучей) представляет собой сортировку, которая использует такую структуру данных, как «куча». В данной структуре от каждого узла может отходить не более двух ответвлений, и обязательно родительский элемент должен быть больше дочерних.

На первом шагу идет построение кучи из заданного массива. Для построения пирамиды начинаем с последнего узла в массиве и перемещаемся вверх по дереву. Для каждого узла проверяем его дочерние узлы и меняем местами узел с наибольшим значением с текущим узлом, если это необходимо. После построения пирамиды, наибольший элемент находится в корне пирамиды. Извлекаем его, помещаем в конец массива и уменьшаем размер пирамиды на 1. Затем перестраиваем пирамиду, чтобы получить новый наибольший элемент. Повторяем этот шаг до тех пор, пока размер пирамиды не станет равным 1.

Сложность данного алгоритма:

Временная сложность: O(n\*log(n))

Затраты памяти: O(n) или O(1) в зависимости от выбранного способа реализации.

**1.10** **Двусторонняя сортировка или сортировка с двух концов**

Данная сортировка является авторской модификацией сортировки выбором. Ее суть заключается в поиске минимального и максимального элементов на каждой итерации и обмене их местами с крайними элементами неотсортированной части массива. При этом на каждой итерации длина неотсортированной части массива сокращается на 2.

Сложность данного алгоритма:

Временная сложность: O(n^2)

Затраты памяти: O(1)

**2 Сравнительная таблица**

На основании полученных данных была построена сравнительная таблица.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Сортировка** | **Временная сложность** | | | **Затраты памяти** |
|  | **Лучшее** | **Среднее** | **Худшее** |  |
| **Пузырьковая** | **O(n)** | **O(n^2)** | **O(n^2)** | **O(1)** |
| **Выбором** | **O(n^2)** | **O(n^2)** | **O(n^2)** | **O(n)** |
| **Вставками** | **O(n)** | **O(n^2)** | **O(n^2)** | **O(n)** |
| **Шейкерная** | **O(n)** | **O(n^2)** | **O(n^2)** | **O(1)** |
| **Расческой** | **O(n\*log(n))** | **O(n^2 / 2^p)** | **O(n^2)** | **O(1)** |
| **Шелла** | **O(n\*log(n))** | **O(n\*log(n))** | **O(n^2)** | **O(1)** |
| **Слиянием** | **O(n\*log(n))** | **O(n\*log(n))** | **O(n\*log(n))** | **O(n)** |
| **Быстрая** | **O(n)** | **O(n\*log(n))** | **O(n^2)** | **O(n)** |
| **Пирамидальная** | **O(n\*log(n))** | **O(n\*log(n))** | **O(n\*log(n))** | **O(n) или O(1)** |
| **Двусторонняя** | **O(n^2)** | **O(n^2)** | **O(n^2)** | **O(1)** |

**3 Проверка алгоритмов на больших данных**

На данном этапе был проведен замер времени работы на массивах с большим количеством данных. Алгоритмы сортировок были реализованы на языке С++ (Приложение 1). Для замера времени была использована библиотека <chrono> языка С++. Результаты отображены на графиках(Рисунок 1, Рисунок 2, Рисунок 3).

Рисунок 1 – Замеры времени на массиве из 10000 чисел

Рисунок 2 – Замеры времени на массиве из 20000 чисел

Рисунок 3 – Замеры времени на массиве из 30000 чисел

**Приложение 1**

#include <iostream>

#include<Windows.h>

#include <conio.h>

#include <random>

#include <chrono>

#include <vector>

using namespace std;

void bubble\_sort(vector<int> &b, int n) { // пузырьковая сортировка

while (n--) {

bool t = false;

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

if (b[i] > b[i + 1]) {

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

t = true;

}

}

if (!t) {

break;

}

}

}

void insert\_sort(vector<int> &c, int n) { // сортировка вставками

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

int sort = i - 1;

while (sort > -1 and c[sort] > c[sort + 1]) {

swap(c[sort], c[sort + 1]);

sort--;

}

}

}

void choice\_sort(vector<int> &d, int n) { // сортировка выбором

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

int min\_poz = i;

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

if (d[j] < d[min\_poz]) {

min\_poz = j;

}

}

if (min\_poz != i) {

swap(d[i], d[min\_poz]);

}

}

}

void quick\_sort(vector<int> &e, int first, int last) { // быстрая сортировка

int middle;

int l = first, r = last;

middle = e[(l + r) / 2];

while (l < r) {

while (e[l] < middle) l++;

while (e[r] > middle) r--;

if (l <= r) {

swap(e[l], e[r]);

l++;

r--;

}

}

if (first < r) quick\_sort(e, first, r);

if (l < last) quick\_sort(e, l, last);

}

void twoway\_sorting(vector<int> &a, int n) { // двусторонняя сортировка

int i = 0;

while (i <= n - i - 1) {

int min = i;

int max = n - i - 1;

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

if (a[j] < a[min]) {

min = j;

}

if (a[j] > a[max]) {

max = j;

}

}

if (i == max and n - i - 1 == min) {

swap(a[i], a[n - i - 1]);

}

else if (i == max) {

int temp = a[n - i - 1];

a[n - i - 1] = a[max];

a[i] = a[min];

a[min] = temp;

}

else if (n - i - 1 == min) {

int temp = a[i];

a[i] = a[min];

a[n - i - 1] = a[max];

a[max] = temp;

}

else {

swap(a[i], a[min]);

swap(a[n - i - 1], a[max]);

}

i++;

}

}

void sheyker(vector<int> &a, int n) { // шейкерная сортировка

int left = 0;

int right = n - 1;

bool swaped = true;

while (left < right&& swaped) {

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

swaped = false;

if (a[i] > a[i + 1]) {

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

swaped = true;

}

}

right--;

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

if (a[i] < a[i - 1]) {

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

swaped = true;

}

}

left++;

}

}

void comb\_sorting(vector<int> &a, int n) { // сортировка расческой

int step = n;

bool swaped = true;

while (step > 1 || swaped) {

step = (int)(step / 1.247);

if (step < 1) step = 1;

swaped = false;

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

if (a[i] > a[i + step]) {

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

swaped = true;

}

}

}

}

void Shell\_sorting(vector<int> &a, int n) { // сортировка Шелла

for (int i = n / 2; i > 0; i /= 2) {

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

for (int k = j - i; k >= 0 && a[k] > a[k + i]; k -= i) {

swap(a[k], a[k + i]);

}

}

}

}

void merge(vector<int> &a, int left, int middle, int right) { // сортировка слиянием

int lefts = middle - left + 1;

int rights = right - middle;

int\* lefta = new int[lefts];

int\* righta = new int[rights];

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

lefta[i] = a[left + i];

}

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

righta[i] = a[middle + 1 + i];

}

int i = 0, j = 0, k = left;

while (i < lefts && j < rights) {

if (lefta[i] <= righta[j]) {

a[k] = lefta[i];

i++;

}

else {

a[k] = righta[j];

j++;

}

k++;

}

while (i < lefts) {

a[k] = lefta[i];

i++;

k++;

}

while (j < rights) {

a[k] = righta[j];

j++;

k++;

}

}

void mergesort(vector<int> &a, int left, int right) {

if (left < right) {

int middle = left + (right - left) / 2;

mergesort(a, left, middle);

mergesort(a, middle + 1, right);

merge(a, left, middle, right);

}

}

void heapify(vector<int> &a, int size, int root) {// пирамидальная сортировка

int largest = root;

int left = 2 \* root + 1;

int right = 2 \* root + 2;

if (left < size && a[left] > a[largest]) {

largest = left;

}

if (right < size && a[right] > a[largest]) {

largest = right;

}

if (largest != root) {

swap(a[root], a[largest]);

heapify(a, size, largest);

}

}

void heapsort(vector<int> &a, int size) {

for (int i = size / 2 - 1; i >= 0; i--) {

heapify(a, size, i);

}

for (int i = size - 1; i > 0; i--) {

swap(a[0], a[i]);

heapify(a, i, 0);

}

}

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

srand(time(NULL));

int n;

cin >> n;

vector<int> a(n);

vector<int> b(n);

vector<int> c(n);

vector<int> d(n);

vector<int> e(n);

vector<int> f(n);

vector<int> g(n);

vector<int> h(n);

vector<int> l(n);

vector<int> m(n);

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

a[i] = b[i] = c[i] = d[i] = e[i] = f[i] = g[i] = h[i] = l[i] = m[i] = rand() % 100000 + 1;

}

cout << "Исходный массив:\n";

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

cout << a[i] << ' ';

}

cout << '\n';

auto start = chrono::high\_resolution\_clock::now();

bubble\_sort(a, n);

auto end = chrono::high\_resolution\_clock::now();

auto time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Пузырьковая сортировка: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

insert\_sort(b, n);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Сортировка вставками: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

choice\_sort(c, n);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Сортировка выбором: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

quick\_sort(d, 0, n - 1);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Быстрая сортировка: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

twoway\_sorting(e, n);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Двусторонняя сортировка: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

sheyker(f, n);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Шейкерная сортировка: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

comb\_sorting(g, n);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Сортировка расческой: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

Shell\_sorting(h, n);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Сортировка Шелла: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

mergesort(l, 0, n - 1);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Сортировка слиянием: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

heapsort(m, n);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Пирамидальная сортировка: " << time.count() << '\n';

}