МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ

ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение

высшего образования

«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет «Информационные системы и технологии»

Кафедра «Вычислительная техника»

Дисциплина «Алгоритм и структуры данных»

КУРСОВОЙ ПРОЕКТ (РАБОТА)

Тема: «Пузырьковая сортировка»

Выполнил студент: Кириллов Данила Александрович / /

фамилия, инициалы подпись

Курс 2 Группа ИВТАСбд-22

Направление/специальность «Информатика и вычислительная техника»

Руководитель: Горшков Д.А.

Дата сдачи:

« » 2023 г.

Дата защиты:

« » 2023 г.

Оценка:

Ульяновск, 2023

**Оглавление**

[**Пузырьковая сортировка** 3](#_Toc134379275)

[**Теоретические оценки времени исполнения пузырьковой сортировки** 3](#_Toc134379276)

[**Вариации реализации** 4](#_Toc134379277)

[**Варианты рабочей нагрузки при исследовании** 4](#_Toc134379278)

[**Оценивание затрат времени** 5](#_Toc134379279)

[**Характер изменения времени исполнения алгоритма** 5](#_Toc134379280)

[**Рекомендации по реализациям** 8](#_Toc134379281)

# **Пузырьковая сортировка**

Алгоритм пузырьковой сортировки - это один из простейших и наиболее известных алгоритмов сортировки массивов. В его основе лежит идея последовательного прохода по массиву и сравнения каждой пары соседних элементов. Если они находятся в неправильном порядке, то они меняются местами. Этот процесс повторяется до тех пор, пока не будет отсортирован весь массив.

Алгоритм пузырьковой сортировки удобен тем, что он не требует дополнительной памяти для выполнения сортировки. Однако его эффективность снижается на больших массивах данных, когда временные затраты на сортировку становятся слишком высокими.

В автоматизированных системах алгоритм пузырьковой сортировки может использоваться для сортировки большинства типовых задач, где нужно отсортировать небольшой набор данных. Например, сортировка списка контактов или списка товаров в интернет-магазине. Однако для сортировки большого количества данных лучше использовать более эффективные алгоритмы сортировки (например, алгоритм быстрой сортировки или сортировки слиянием).

# **Теоретические оценки времени исполнения пузырьковой сортировки**

Теоретическая оценка времени исполнения пузырьковой сортировки составляет O(n^2), где n - количество элементов, которые нужно отсортировать. Это означает, что время исполнения алгоритма увеличивается пропорционально квадрату количества элементов. Поэтому пузырьковую сортировку не рекомендуется использовать для больших наборов данных, так как она может быть очень медленной.

Это связано с тем, что алгоритм сортировки пузырьком сравнивает каждую пару элементов и меняет их местами, если они находятся в неправильном порядке. Для случайного массива элементов вероятность того, что два элемента будут находиться в неправильном порядке, примерно равна 50%. Таким образом, каждый элемент в среднем нужно сравнить с половиной остальных элементов. Это означает, что общее количество сравнений будет n(n-1)/2, что дает сложность O(n^2).

В то же время, для отсортированного массива (лучший случай) сложность сортировки пузырьком будет O(n), а для обратно отсортированного массива (худший случай) - O(n^2).

# **Вариации реализации**

В ходе исследования алгоритма были использованы следующие вариации реализации:

* Язык программирования: алгоритм был реализован на языке Python и C++.
* Тип данных: целые числа(int), цифры с плавающей точкой(double), строки(string), стурктуры.

# **Варианты рабочей нагрузки при исследовании**

При исследовании данного алгоритма использовалось три варианта рабочей нагрузки с разными входными данными:

* Неотсортированный массив
* Отсортированный массив
* Отсортированный массив в обратном порядке

Для каждого варианта нагрузки было измерено время работы алгоритма. Также была оценена эффективность алгоритма в зависимости от размера массива.

В результате исследования было установлено, что данный алгоритм имеет лучшую эффективность при сортировке отсортированных и частично отсортированных массивов, чем при сортировке неотсортированных и в обратном порядке отсортированных массивов.

Однако, при работе с массивами большого размера, данный алгоритм может иметь значительную задержку, что делает его менее предпочтительным для использования в таких случаях.

Таким образом, выбор этих вариантов рабочей нагрузки обусловлен необходимостью проверки производительности алгоритма в различных условиях, чтобы определить, как он будет работать в реальных приложениях.

# **Оценивание затрат времени**

Для измерения времени в C++ была использована библиотека <chrono>. Для того чтобы понять, за сколько работает пузырьковая сортировка на том или ином массиве, до объявления функции сортировки создается переменная, которой присваивается текущее время: auto start = std::chrono::steady\_clock::now(). Потом вызывается функция сортировки и после создается еще одна переменная, которой также присваивается текущее время, но уже после работы сортировки: auto end = std::chrono::steady\_clock::now(). Далее просто находилась разница между этими переменными и выводилась в консоль.

Для измерения времени выполнения алгоритма на Python была использована библиотека time, в из которой вызывалась функция time.monotonic(). Также, как и на C++, сначала создавалась переменная для time.monotonic(), потом вызывалась функция сортировки, а далее другая переменная для time.monotonic(). В конце также подсчитывалась разница и записывалась в консоль.

# **Характер изменения времени исполнения алгоритма**

Ниже представлены результаты исследования параметров рабочей нагрузки.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

*Рисунок 1-4. Зависимость размера от времени на Python. Случайные входные данные*

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

*Рисунок 5-8. Зависимость размера от времени на Python. Случайные данные, отсортированные в обратном порядке*

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

*Рисунок 8-12. Зависимость размера от времени на Python. Отсортированные входные данные*

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

*Рисунок 13-16. Зависимость размера от времени на C++. Случайные входные данные*

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

*Рисунок 17-20. Зависимость размера от времени на C++. Случайные данные, отсортированные в обратном порядке*

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

*Рисунок 21-24. Зависимость размера от времени на C++. Отсортированные входные данные*

Таким образом, график будет загибаться у оси абсцисс для любых входных данных и будет иметь виды y = x^2.

Временная сложность пузырьковой сортировки может сильно зависеть от количества входных данных, даже если эти данные уже отсортированы в правильном порядке. Это связано с тем, что пузырьковая сортировка имеет квадратичную временную сложность O(n^2), что означает, что время выполнения алгоритма увеличивается пропорционально квадрату числа элементов входных данных. Даже если данные уже отсортированы в правильном порядке, пузырьковая сортировка все равно проходит по каждому элементу массива несколько раз для определения, что все элементы находятся в правильной позиции. Поэтому, для больших объемов данных, временная сложность будет значительно выше, даже если данные уже отсортированы.

Экспериментальные оценки затрат времени исполнения алгоритма близки к теоретическим при обратно отсортированных входных данных, где временная сложность будет равна O(n^2).

# **Рекомендации по реализациям**

Пузырьковую сортировку можно рекомендовать в следующих случаях:

1. Массив не очень большой, т.к. сложность алгоритма O(n^2) и время сортировки будет расти быстро с увеличением количества элементов.

2. Массив уже почти отсортирован, т.к. в этом случае пузырьковая сортировка может работать быстрее, чем другие алгоритмы.

3. Нужна сортировка на месте без использования дополнительной памяти, т.к. пузырьковая сортировка требует только одну дополнительную ячейку памяти для временного хранения значения при перестановке элементов.

Также в ходе исследования выяснилось, что код на C++ быстрее, чем на Python. Это связано с различиями в производительности и оптимизации между языками программирования C++ и Python. C++ - это компилируемый язык программирования, который выполняется значительно быстрее, чем интерпретируемый Python. Кроме того, C++ включает в себя возможности управления памятью и оптимизации кода, что позволяет создавать более эффективный код, а Python - это более гибкий и удобный язык программирования, но его скорость выполнения кода обычно медленнее, что может замедлить выполнение алгоритмов сортировки.

1. Интерпретируемость Python. Python - интерпретируемый язык программирования, что означает, что код Python должен интерпретироваться в реальном времени, когда программа запускается. Это значит, что скорость выполнения на Python будет медленнее, чем на C++, который является компилируемым языком.

2. Работа с памятью. В Python операции с памятью могут вызывать большие накладные расходы, за счет чего производительность снижается. В C++ мы имеем возможность управлять памятью непосредственно, что позволяет более эффективно использовать ее ресурсы.

3. Оптимизация кода в C++. C++ предоставляет мощные оптимизационные инструменты, которые позволяют оптимизировать код для конкретных аппаратных платформ. Это позволяет значительно улучшать производительность кода, в то время как Python не может предложить таких возможностей.

4. Уровень абстракции. Python является высокоуровневым языком программирования и имеет более высокий уровень абстракции, что означает, что он предоставляет более высокий уровень обобщения для решений. C++, с другой стороны, является более низкоуровневым языком, что позволяет разработчикам больше контролировать железо и управлять операциями.

# **Вывод**

В ходе данной курсовой работы было проведено исследование алгоритма пузырьковой сортировки. Реализация происходила на двух языках программирования: C++ и Python. Были проведены замеры времени работы сортировки при различном количестве входных данных, их типе и степени отсортированности.

Таким образом, код на C++ работает быстрее, чем на Python. Большой объем входных данных не очень подходит для данного алгоритма, так как с их увеличением растет временная сложность.

# **Приложение**

Пример реализации на C++

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <chrono>

#include <random>

#include <vector>

#include <string>

using namespace std;

void bubbleSort(int arr[], int n)

{

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

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

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

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

}

}

}

}

void bubbleSortReverse(double arr[], int n) {

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

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

if (arr[j] < arr[j + 1]) {

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

}

}

}

}

int main()

{

auto seed = std::chrono::system\_clock::now().time\_since\_epoch().count();

std::mt19937 generator(seed);

std::uniform\_int\_distribution<int> distribution(1, 5000); std::uniform\_real\_distribution<double> distribution2(1, 5000);

int n = distribution(generator);

int\* arr = new int[n];

cout << "Unsorted Array: ";

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

arr[i] = distribution(generator);

}

cout << endl;

cout << endl;

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

bubbleSort(arr, n);

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

std::chrono::duration<double> diff = end - start;

std::cout << "\nTime taken: " << diff.count() << " seconds\n";

std::cout << n;

return 0;

}

Пример реализации на Python

import datetime  
import random  
import time  
  
  
def bubble\_sort(arr):  
 n = len(arr)  
 # Проход по всем элементам  
 for i in range(n):  
 swapped = False  
 # Последние i элементов уже были отсортированы  
 for j in range(0, n - i - 1):  
 # Если элемент j больше следующего за ним элемента  
 if arr[j] > arr[j + 1]:  
 # Поменять элементы местами  
 arr[j], arr[j + 1] = arr[j + 1], arr[j]  
 swapped = True  
 if not swapped:  
 return arr  
  
 return arr  
  
  
def bubble\_sort\_reverse(arr):  
 n = len(arr)  
 for i in range(n-1):  
 for j in range(n-1-i):  
 if arr[j] < arr[j+1]:  
 arr[j], arr[j+1] = arr[j+1], arr[j]  
 return arr  
  
  
random.seed(time.time())  
n = random.randint(1, 5000)  
arr = [random.randint(1, 5000) for i in range(10)]  
  
print("Unsorted arr: ")  
print(arr)  
print(n)  
  
start = time.monotonic()  
sort = bubble\_sort(arr)  
end = time.monotonic()  
total\_time = end - start  
print("Sorted arr: ")  
print(f"Time taken: {total\_time:.5f} seconds")