ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Сортировки»

Выполнила работу

Луценко Елена

Академическая группа J3112

Принято

Дунаев Максим Владимирович

Санкт-Петербург

2024

**Структура отчёта:**

1. Введение

Цель работы: исследовать эффективность трех различных алгоритмов сортировки: Pancake sort, Heap sort, Comb sort.

Задачи:

1. Реализовать указанные алгоритмы сортировки на языке C++.
2. Сравнить время выполнения каждого алгоритма при различных размерах входных данных.
3. Построить графики зависимости времени выполнения от количества элементов.
4. Проанализировать результаты и оценить эффективность алгоритмов.
5. Теоретическая подготовка
6. Pancake sort — это алгоритм сортировки, который использует операции переворота (flip) для перемещения элементов в правильном порядке.
7. Heap Sort — это алгоритм сортировки, основанный на структуре данных куче (heap), которая представляет собой частично упорядоченное бинарное дерево; алгоритм использует кучу для эффективной сортировки массива.
8. Comb Sort – это улучшенная версия пузырьковой сортировки. Основная идея заключается в том, чтобы постепенно уменьшать "шаг" (интервал) между элементами для сортировки, что ускоряет процесс по сравнению с обычной пузырьковой сортировкой.
9. Реализация

В ходе работы были реализованы три алгоритма сортировки на языке C++. Каждый алгоритм был помещен в отдельную функцию (или функции), а также была реализована функция для чтения данных из файла и замера времени выполнения (код находится в приложениях с комментариями)

1. Экспериментальная часть

В ходе лабораторной работы были протестированы три алгоритма сортировки: Pancake sort, Heap sort, Comb sort. Для каждого алгоритма были собраны данные о времени выполнения при различных объемах входных данных. Теоретически, временная сложность Pancake sort составляет O(n2), пространственная – О(1), в то время как Heap sort и Comb sort имеют временную сложность в среднем случае O(nlogn), а пространственную - О(1) для Comb sort и О(logn) для Heap sort.

Таблица №1 - Подсчёт сложности реализованного алгоритма

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Размер входного набора | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 | 7000 | 8000 |
| Pancake sort | 0.02770 | 0.08685 | 0.19178 | 0.34635 | 0.62188 | 0.86542 | 1.08669 | 1.40259 |
| Heap sort | 0.00061 | 0.00134 | 0.00211 | 0.00298 | 0.00385 | 0.00461 | 0.00661 | 0.00888 |
| Comb sort | 0.00044 | 0.00143 | 0.00167 | 0.00236 | 0.00269 | 0.00315 | 0.00431 | 0.00589 |

График, представляющий визуально удобный формат данных из таблицы №1, представлен на изображениях №1 и №2.

Изображение выглядит как текст, линия, График, диаграмма

Автоматически созданное описание

Изображение №1 - График работы алгоритмов

Изображение выглядит как текст, линия, диаграмма, График

Автоматически созданное описание

Изображение №2 - График работы алгоритмов для Heap Sort и Comb Sort (для более точного отображения)

Согласно теоретическим оценкам Pancake sort имеет временную сложность O(n2) и пространственную – О(1). Результаты эксперимента подтверждают это: время выполнения значительно увеличивается с ростом размера входного массива. Heap sort и Comb sort, имея временную сложность в среднем случае O(nlogn), а пространственную – О(1) для Comb sort и О(logn) для Heap sort, демонстрируют значительно более быстрое выполнение по сравнению с другим алгоритмом, что соответствует теоретической оценке.

Для оценки эффективности разработанных алгоритмов были проведены серии экспериментов с различными объемами входных данных. В соответствии с требованиями лабораторной работы были выбраны наборы данных с 10000 (1е4) и 100000 (1е5) элементов (по 50 для каждого из видов наборов).

Box plot графики для времени работы алгоритмов с числом элементов 1e4 и 1е5 представлены на изображениях № 3-8.

Изображение выглядит как текст, снимок экрана, Шрифт, чек

Автоматически созданное описание

Изображение №3 – box plot графики для времени работы алгоритмов с числом элементов 1е4 (в секундах)

Изображение выглядит как снимок экрана, диаграмма, текст, линия

Автоматически созданное описание

Изображение №4 – box plot график для времени работы алгоритма с числом элементов 1е4 для Pancake sort (в секундах)

Изображение выглядит как диаграмма, снимок экрана, График, линия

Автоматически созданное описание

Изображение №5 – box plot графики для времени работы алгоритмов с числом элементов 1е4 для Heap sort и Comb sort (в миллисекундах)

Изображение выглядит как текст, снимок экрана, диаграмма, число

Автоматически созданное описание

Изображение №6 – box plot графики для времени работы алгоритмов с числом элементов 1е5 (в секундах)

Изображение выглядит как снимок экрана, диаграмма, текст, линия

Автоматически созданное описание

Изображение №7 – box plot график для времени работы алгоритма с числом элементов 1е5 для Pancake sort (в секундах)

Изображение выглядит как диаграмма, снимок экрана, линия, График

Автоматически созданное описание

Изображение №8 – box plot графики для времени работы алгоритмов с числом элементов 1е5 для Heap sort и Comb sort (в секундах)

Боксплоты для каждого алгоритма показывают распределение времени выполнения: Pancake sort имеет наибольшее время выполнения и наибольшую дисперсию среди всех трех алгоритмов. Heap sort имеет время выполнения большее, чем у Comb sort, но значительно меньшее, чем у Pancake sort, а дисперсия у Heap sort в среднем ниже, чем у Comb sort. Выбросы имеются в случае лучших и худших случаев для сортировок.

1. Заключение

В ходе выполнения лабораторной работы были реализованы три алгоритма сортировки: Pancake sort, Heap sort, Comb sort. Цель работы заключалась в сравнении их производительности при различных объемах входных данных. Для достижения этой цели были собраны и проанализированы результаты времени выполнения алгоритмов на различных наборах данных.

Полученные результаты подтвердили теоретические оценки сложности алгоритмов. Comb sort продемонстрировал наилучшие показатели производительности, что соответствует его временной сложности в среднем случае O(nlogn), а пространственной – О(1). Heap sort продемонстрировал показатели немного хуже, что соответствует его временной сложности в среднем случае O(nlogn), а пространственной – О(logn), в то время как Pancake sort показала значительно более высокое время выполнения, соответствующее его временной сложности O(n2) и пространственной сложности О(1). Это подчеркивает важность выбора подходящего алгоритма в зависимости от объема данных и требований к времени выполнения. Так, Pancake sort можно использовать для сортировки массивов с количеством элементов до 1е4 включительно (разница между временем этих трёх сортировок тогда незначительна), а для массивов с большим количеством элементов оптимально использовать Comb sort и Heap sort.

Выбросы имеются в случае лучших и худших случаев для сортировок.

В качестве направлений для будущих исследований можно рассмотреть оптимизацию Pancake sort с точки зрения уменьшения временных затрат.

1. Приложения

ПРИЛОЖЕНИЕ А

Листинг кода файла pancake\_sort.cpp

#include <iostream>  
#include <algorithm>  
#include <chrono>  
#include <iomanip>  
#include <vector>  
  
using namespace std;  
using namespace chrono;  
  
// Функция для переворота подмассива от 0 до i  
void flip(vector<int> &arr, int i) { // временная - O(n), пространственная - O(1)  
 reverse(arr.begin(), arr.begin() + i + 1);  
}  
  
// Функция для нахождения индекса максимального элемента в подмассиве arr[0..n-1]  
int findMax(vector<int> arr, int n) { // временная - O(n), пространственная - O(1)  
 int maxIndex = 0;  
 for (int i = 1; i < n; i++) {  
 if (arr[i] > arr[maxIndex]) {  
 maxIndex = i;  
 }  
 }  
 return maxIndex;  
}  
  
// Основная функция Pancake Sort  
void pancakeSort(vector<int> &arr) {  
 int n = arr.size();  
 for (int curr\_size = n; curr\_size > 1; curr\_size--) { // временная - O(n^2), пространственная - O(1)  
 int maxIndex = findMax(arr, curr\_size);  
 if (maxIndex != curr\_size - 1) {  
 flip(arr, maxIndex); // временная - O(n), пространственная - O(1)  
 flip(arr, curr\_size - 1); // временная - O(n), пространственная - O(1)  
 }  
 }  
}  
  
  
// Функция для вывода массива  
void printArray(vector<int> arr) { // временная - O(n), пространственная - O(1)  
 int n = arr.size();  
 for (int i = 0; i < n; i++) {  
 cout << arr[i] << " ";  
 }  
 cout << endl;  
}  
  
  
int main() {  
 // Лучший случай  
 vector<int> bestCase = **{**1, 2, 3, 4, 5, 6, 7, 8, 9, 10**}**;  
  
 // Худший случай  
 vector<int> worstCase = **{**10, 9, 8, 7, 6, 5, 4, 3, 2, 1**}**;  
  
 // Средний случай  
 vector<int> averageCase = **{**5, 3, 8, 6, 2, 7, 4, 10, 1, 9**}**;  
  
 // Тестирование на лучшем случае  
 cout << "Best Case: ";  
 printArray(bestCase);  
  
 auto start = high\_resolution\_clock::now();  
 pancakeSort(bestCase);  
 auto end = high\_resolution\_clock::now();  
  
 cout << "Sorted Best Case: ";  
 printArray(bestCase);  
  
 auto durationBest = duration\_cast<microseconds>(end - start).count();  
  
 // Тестирование на худшем случае  
 cout << "Worst Case: ";  
 printArray(worstCase);  
  
 start = high\_resolution\_clock::now();  
 pancakeSort(worstCase);  
 end = high\_resolution\_clock::now();  
  
 cout << "Sorted Worst Case: ";  
 printArray(worstCase);  
  
 auto durationWorst = duration\_cast<microseconds>(end - start).count();  
  
 // Тестирование на среднем случае  
 cout << "Average Case: ";  
 printArray(averageCase);  
  
 start = high\_resolution\_clock::now();  
 pancakeSort(averageCase);  
 end = high\_resolution\_clock::now();  
  
 cout << "Sorted Average Case: ";  
 printArray(averageCase);  
  
 auto durationAverage = duration\_cast<microseconds>(end - start).count();  
  
 // Вывод результатов времени выполнения  
 cout << fixed << setprecision(3); // Устанавливаем точность вывода до трех знаков после запятой  
 cout << "Time taken for Best Case: " << durationBest / 1000.0 << " ms" << endl;  
 cout << "Time taken for Worst Case: " << durationWorst / 1000.0 << " ms" << endl;  
 cout << "Time taken for Average Case: " << durationAverage / 1000.0 << " ms" << endl;  
  
  
 return 0;  
}

ПРИЛОЖЕНИЕ В

Листинг кода файла heap\_sort.cpp

#include <iostream>  
#include <chrono>  
#include <iomanip>  
#include <vector>  
  
using namespace std;  
using namespace chrono;  
  
// Функция для обмена двух элементов  
void swap(int &a, int &b) { // временная сложность - O(1), пространственная - O(1)  
 int temp = a;  
 a = b;  
 b = temp;  
}  
  
// Функция для восстановления свойства кучи  
void heapify(vector<int> &arr, int n, int i) { // в среднем случае временная сложность - O(logn), пространственная - O(logn)  
 int largest = i;  
 int left = 2 \* i + 1;  
 int right = 2 \* i + 2;  
 if (left < n and arr[left] > arr[largest]) {  
 largest = left;  
 }  
 if (right < n and arr[right] > arr[largest]) {  
 largest = right;  
 }  
 if (largest != i) {  
 swap(arr[i], arr[largest]);  
 heapify(arr, n, largest);  
 }  
}  
  
// Основная функция для сортировки массива с использованием Heap Sort  
void heapSort(vector<int> &arr) { // в среднем случае временная сложность - O(nlogn), пространственная - O(logn)  
 int n = arr.size();  
 // Строим кучу  
 for (int i = n / 2 - 1; i >= 0; i--) { // в среднем случае временная сложность - O(nlogn), пространственная - O(logn)  
 heapify(arr, n, i);  
 }  
 // Извлекаем элементы из кучи по одному  
 for (int i = n - 1; i > 0; i--) { // в среднем случае временная сложность - O(nlogn), пространственная - O(logn)  
 swap(arr[0], arr[i]);  
 heapify(arr, i, 0);  
 }  
}  
  
// Функция для вывода массива  
void printArray(vector<int> arr) { // временная сложность - O(n), пространственная - O(1)  
 int n = arr.size();  
 for (int i = 0; i < n; i++) {  
 cout << arr[i] << " ";  
 }  
 cout << endl;  
}  
  
  
int main() {  
 // Лучший случай  
 vector<int> bestCase = **{**1, 2, 3, 4, 5, 6, 7, 8, 9, 10**}**;  
  
 // Худший случай  
 vector<int> worstCase = **{**10, 9, 8, 7, 6, 5, 4, 3, 2, 1**}**;  
  
 // Средний случай  
 vector<int> averageCase = **{**5, 3, 8, 6, 2, 7, 4, 10, 1, 9**}**;  
  
 // Тестирование на лучшем случае  
 cout << "Best Case: ";  
 printArray(bestCase);  
  
 auto start = high\_resolution\_clock::now();  
 heapSort(bestCase);  
 auto end = high\_resolution\_clock::now();  
  
 cout << "Sorted Best Case: ";  
 printArray(bestCase);  
  
 auto durationBest = duration\_cast<microseconds>(end - start).count();  
  
 // Тестирование на худшем случае  
 cout << "Worst Case: ";  
 printArray(worstCase);  
  
 start = high\_resolution\_clock::now();  
 heapSort(worstCase);  
 end = high\_resolution\_clock::now();  
  
 cout << "Sorted Worst Case: ";  
 printArray(worstCase);  
  
 auto durationWorst = duration\_cast<microseconds>(end - start).count();  
  
 // Тестирование на среднем случае  
 cout << "Average Case: ";  
 printArray(averageCase);  
  
 start = high\_resolution\_clock::now();  
 heapSort(averageCase);  
 end = high\_resolution\_clock::now();  
  
 cout << "Sorted Average Case: ";  
 printArray(averageCase);  
  
 auto durationAverage = duration\_cast<microseconds>(end - start).count();  
  
 // Вывод результатов времени выполнения  
 cout << fixed << setprecision(3); // Устанавливаем точность вывода до трех знаков после запятой  
 cout << "Time taken for Best Case: " << durationBest / 1000.0 << " ms" << endl;  
 cout << "Time taken for Worst Case: " << durationWorst / 1000.0 << " ms" << endl;  
 cout << "Time taken for Average Case: " << durationAverage / 1000.0 << " ms" << endl;  
  
  
 return 0;  
}

ПРИЛОЖЕНИЕ С

Листинг кода файла comb\_sort.cpp

#include <iostream>  
#include <chrono>  
#include <iomanip>  
#include <vector>  
  
using namespace std;  
using namespace chrono;  
  
  
// Функция для обмена двух элементов  
void swap(int &a, int &b) { // временная сложность - О(1), пространственная - О(1)  
 int temp = a;  
 a = b;  
 b = temp;  
}  
  
// Функция для выполнения сортировки Comb Sort  
void combSort(vector<int> &arr) { // временная сложность в среднем случае - О(nlogn), пространственная - О(1)  
 int n = arr.size();  
 int gap = n;  
 bool swapped = true;  
  
 // Пока шаг больше 1, продолжаем сортировать  
 while (gap > 1 or swapped) { // временная сложность в среднем случае - О(nlogn), пространственная - О(1)  
 gap = gap \* 10 / 13; // Уменьшаем шаг (обычно на 1.3)  
 if (gap < 1) gap = 1;  
  
 swapped = false;  
  
 for (int i = 0; i + gap < n; i++) { // временная сложность - О(n), пространственная - О(1)  
 if (arr[i] > arr[i + gap]) {  
 swap(arr[i], arr[i + gap]);  
 swapped = true;  
 }  
 }  
 }  
}  
  
  
// Функция для вывода массива  
void printArray(vector<int> arr) { // временная сложность - О(n), пространственная - О(1)  
 int n = arr.size();  
 for (int i = 0; i < n; i++) {  
 cout << arr[i] << " ";  
 }  
 cout << endl;  
}  
  
  
int main() {  
 // Лучший случай  
 vector<int> bestCase = **{**1, 2, 3, 4, 5, 6, 7, 8, 9, 10**}**;  
  
 // Худший случай  
 vector<int> worstCase = **{**10, 9, 8, 7, 6, 5, 4, 3, 2, 1**}**;  
  
 // Средний случай  
 vector<int> averageCase = **{**5, 3, 8, 6, 2, 7, 4, 10, 1, 9**}**;  
  
 // Тестирование на лучшем случае  
 cout << "Best Case: ";  
 printArray(bestCase);  
  
 auto start = high\_resolution\_clock::now();  
 combSort(bestCase);  
 auto end = high\_resolution\_clock::now();  
  
 cout << "Sorted Best Case: ";  
 printArray(bestCase);  
  
 auto durationBest = duration\_cast<microseconds>(end - start).count();  
  
 // Тестирование на худшем случае  
 cout << "Worst Case: ";  
 printArray(worstCase);  
  
 start = high\_resolution\_clock::now();  
 combSort(worstCase);  
 end = high\_resolution\_clock::now();  
  
 cout << "Sorted Worst Case: ";  
 printArray(worstCase);  
  
 auto durationWorst = duration\_cast<microseconds>(end - start).count();  
  
 // Тестирование на среднем случае  
 cout << "Average Case: ";  
 printArray(averageCase);  
  
 start = high\_resolution\_clock::now();  
 combSort(averageCase);  
 end = high\_resolution\_clock::now();  
  
 cout << "Sorted Average Case: ";  
 printArray(averageCase);  
  
 auto durationAverage = duration\_cast<microseconds>(end - start).count();  
  
 // Вывод результатов времени выполнения  
 cout << fixed << setprecision(3); // Устанавливаем точность вывода до трех знаков после запятой  
 cout << "Time taken for Best Case: " << durationBest / 1000.0 << " ms" << endl;  
 cout << "Time taken for Worst Case: " << durationWorst / 1000.0 << " ms" << endl;  
 cout << "Time taken for Average Case: " << durationAverage / 1000.0 << " ms" << endl;  
  
  
 return 0;  
}