ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 4

«Алгоритмы сортировки»

Выполнил работу

Воробьев Андрей

Академическая группа №J3110

Принято

Ментор, Вершинин Владислав

Санкт-Петербург

2024

1. Введение

Цель лабораторной работы — изучить и научиться писать алгоритмы сортировки с различными временными и пространственными сложностями.

Задачи – изучить и написать сортировки с разными характеристиками:

* лучшая временная сложность – O(n^2), пространственная сложность – O(1)
* средняя временная сложность до O(n^2), пространственная сложность – до O(n)
* средняя временная сложность O(n\*k), пространственная сложность до O(n\*k).

Необходимо проанализировать все алгоритмы с разными входными данными разных размеров, найти тесты для лучшего, среднего и худшего случаев, построить графики и диаграммы.

1. Реализация

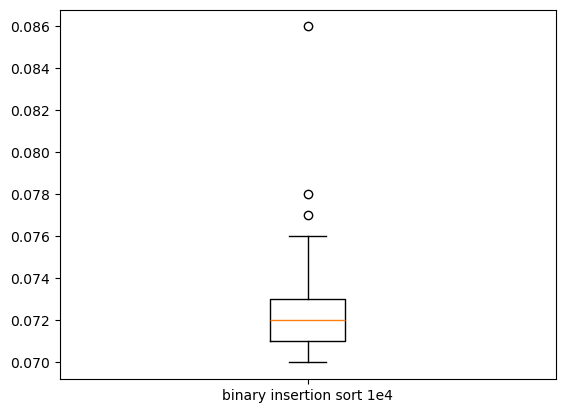
* В интернете я нашёл две сортировки схожие с примерами, но более оптимизированные, последнюю сортировку я взял из предложенных
* Я переписал свою тестирующую систему для более универсальной работы с данными, а также написал генератор массов с заданной длинной.
* Я реализовал все алгоритмы и проверил их на правильность сортировки с помощью тестирующей системы.
* Оптимизация перебора, чтобы немного приблизиться к требуемому времени работы

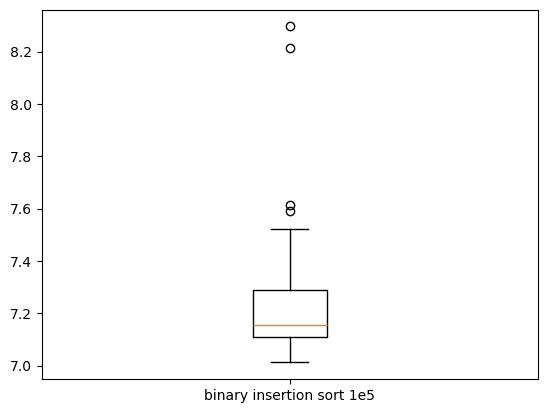
1. Экспериментальная часть
   1. В качестве первого алгоритма сортировки был взят алгоритм

сортировки бинарными вставками. Он является более оптимизированным, чем обычная сортировка вставками, так как совершает log n сравнений, вместо n, то есть работает в среднем за O(n\*log n). Но в худшем случае так же работает за O(n^2), при массиве, построенном в обратном порядке. А в лучшем случае (отсортированный массив) всё равно работает за O(n\*log n), в отличие от O(n). Пространственная сложность – O(1).

Тестирование на массивах от 10^3 до 10^6 с шагом 10^3 занимало слишком большое время, поэтому дойдя до размера 4\*10^5, шаг был изменён на 10^5. Большой ценности дальнейший малый шаг не представлял, так как уже на большой выборке было проверено отсутствие выбросов.

Незначительность выбросов можно увидеть с использованием диаграммы boxplot, представленной на изображении 1 для двух размеров массивов.



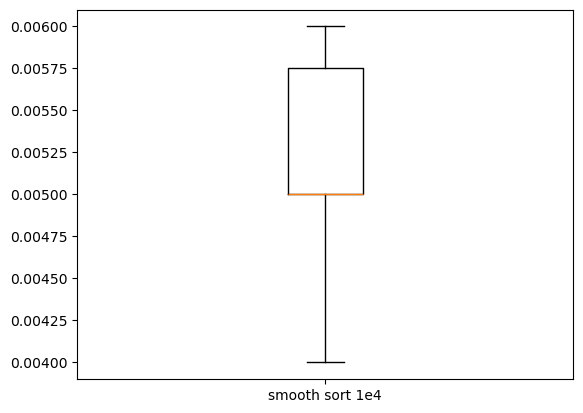


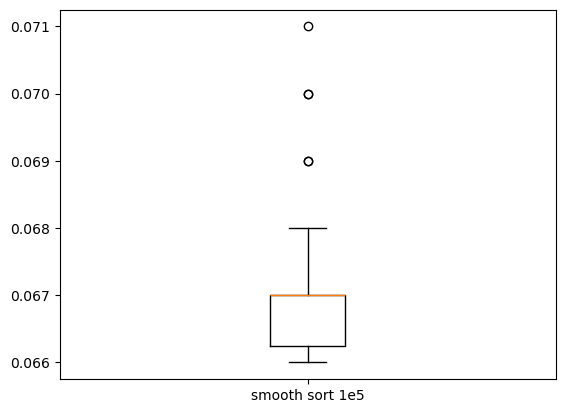
Изображение 1 – Boxplot диаграмма для алгоритма

сортировки бинарными вставками

* 1. В качестве второго алгоритма сортировки был взят алгоритм плавной сортировки. Он является оптимизацией сортировки кучей, так как использует в построении куч числа Леонардо. В худшем случае алгоритм так же работает за O(n\*log n), при массиве, построенном в обратном порядке. Но в среднем его сложность приближается к O(n). Лучшим является также упорядоченный массив и сложность O(n). Пространственная сложность – O(n).

Незначительность выбросов можно увидеть с использованием диаграммы boxplot, представленной на изображении 2 для двух размеров массивов.



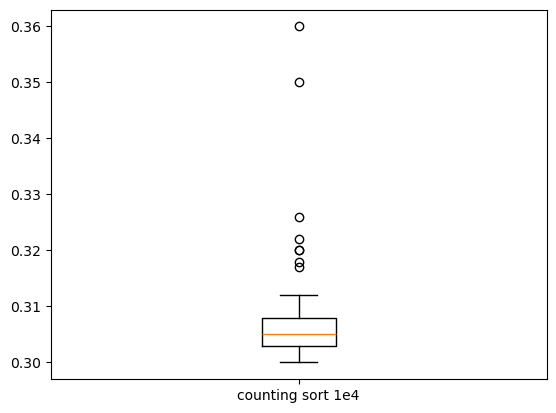


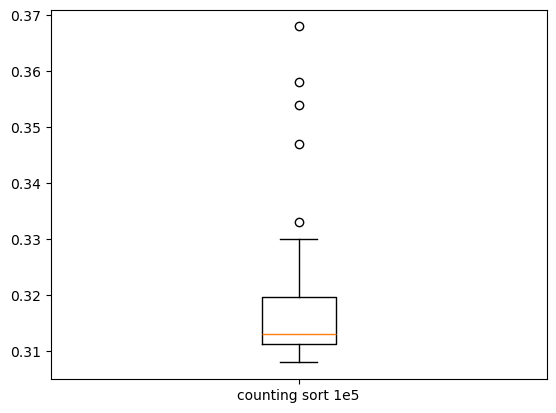
Изображение 2 – Boxplot диаграмма для алгоритма

плавной сортировки

* 1. В качестве третьего алгоритма сортировки был взят алгоритм сортировки подсчётом. Он был приведён в качестве примера и действительно работает за O(n+k), где k – разность между минимальным и максимальным массивом. Сложность алгоритма почти всегда зависит именно от максимальной разницы в данных. Таким образом, в худшем случае - при малых n, но большом разбросе значений, сложность может быть в несоизмеримо больше n. Но в среднем, при небольшом разбросе значений, сложность близка к O(n). Пространственная сложность – O(n+k).

Размеры выбросов (изображение 3) незначительны, так при таких размерах массивы, построенные из случайных чисел, имеют близкие максимальные и минимальные числа.

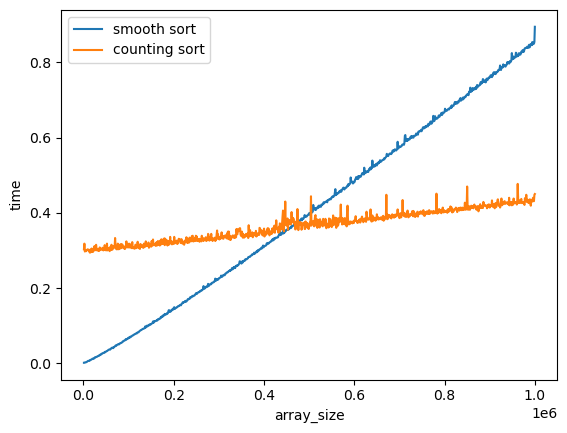
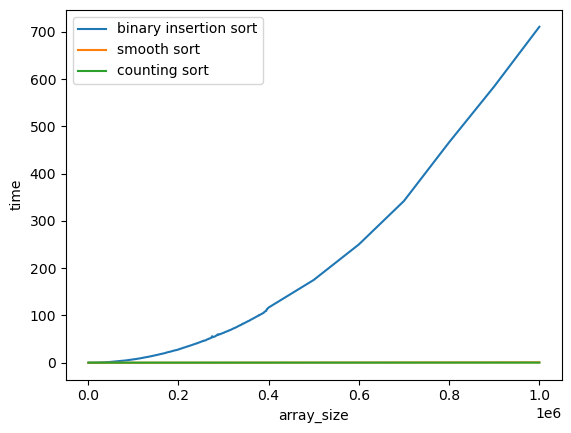




Изображение 3 – Boxplot диаграмма для алгоритма

сортировки подсчётом

* 1. Результаты тестирования алгоритмов на массивах 3 из случайных чисел размера от 10^3 до 10^6 с шагом в 10^3 отображены в графиках на изображениях 4.



Изображение 4 – Линейные диаграммы

работы алгоритмов

По графикам заметно, что сложность алгоритма сортировки бинарными вставками несоизмерима со сложностью других. Время алгоритма возрастает очень стремительно при увеличении n. Сравнение работы более быстрых алгоритмов даёт очень интересный результат. Плавная сортировка имеет очень стабильный линейный рост времени работы с увеличением размера массива. Также число выбросов в её графике очень мало, а выбросы незначительны. Сортировка подсчётом имеет очень стабильную сложность, время работы алгоритма очень медленно растёт при увеличении размера массива. Выбросы достаточно значительны по сравнению с другими сортировками, это связано, что в некоторых случаях числа генерировались с очень большой разницей между максимумом и минимум. Также очень интересно, что эта сортировка изначально имеет достаточно большую числовую константу, но из-за медленного роста – алгоритм становится самым эффективным на больших размерах массива.

1. Заключение

В ходе выполнения работы мною были реализованы алгоритмы сортировки разных типов на языке программирования С++. Благодаря сбору данных о их работы получилось выяснить, что алгоритм сортировки бинарными вставками является очень неэффективным по времени, но пригодится при отсутствии больших объёмов памяти. Алгоритм плавной сортировки является самым универсальным, так как его время работы возрастает практически линейно, и он не зависит от входных данных, потребляя линейное количество памяти. Алгоритм сортировки подсчётом является незаменимым и очень эффективным на данных, которые имеют не большую разность, в таком случае он потребляется O(n) памяти и работает за это же время. Но алгоритм неприменим на данных с большим разбросом, так как в таком случае его время работы и потребление памяти будет хуже, чем все другие алгоритмы, даже для очень маленьких n. Таким образом, каждый алгоритм применим для своего случая.

1. Приложения

ПРИЛОЖЕНИЕ А

Листинг кода файла binary\_insertion\_sort.cpp

#include "binary\_insertion\_sort.h"  
  
size\_t binary\_search(const std::vector<long long> &v, size\_t left, size\_t right, long long key) {  
 if (left >= right)  
 return key >= v[left] ? left + 1 : left;  
  
 size\_t mid = (left + right) / 2;  
  
 if (key == v[mid])  
 return mid + 1;  
 if (key > v[mid])  
 return binary\_search(v, mid + 1, right, key);  
 return binary\_search(v, left, mid == 0 ? mid : mid - 1, key);  
}  
  
void binary\_insertion\_sort(std::vector<long long> &v) {  
 for (size\_t i = 1; i < v.size(); ++i) {  
 *// i - индекс элемента, который̆ нужно вставить  
 // O(n) - время* long long key = v[i];  
 *// вставляемый элемент* size\_t j = i - 1;  
 *// j - индекс последнего элемента отсортированного подмассива* size\_t l = binary\_search(v, 0, j, key);  
 *// loc - индекс на котором должен стоять вставляемый элемент в отсортированном подмассиве  
 // O(log i) - время, i - длина отсортированного подмассива* while (j >= l) {  
 *// сдвигаем элементы большие вставляемого вправо  
 // O(i - l) - время* v[j + 1] = v[j];  
 if (j == 0)  
 break;  
 --j;  
 }  
  
 l == 0 ? v[0] = key : v[j + 1] = key;  
 *// вставляем вставляемый элемент* }  
 *// время:  
 // перебор \* (бинарный поиск + сдвиг)  
 // в отличие от стандартной сортировки:  
 // перебор \* (сравнение + сдвиг)  
  
 // худший случай O(n^2)  
 // средний случай O(n \* log n)  
 // лучший случай O(n \* log n)  
 // память O(1)*}

ПРИЛОЖЕНИЕ Б

Листинг кода файла smooth\_sort.cpp

#include "smooth\_sort.h"  
  
std::vector<size\_t> create\_leonardo\_numbers(const size\_t n) {  
 *// время и память O(1)* std::vector<size\_t> res;  
 res.push\_back(1);  
 res.push\_back(1);  
 while (res.back() < n)  
 res.push\_back(res[res.size() - 1] + res[res.size() - 2] + 1);  
 return res;  
}  
  
  
void balance\_tree(std::vector<long long> &array, std::vector<size\_t> &subtree\_size,  
 const std::vector<size\_t> &leonardo\_nums, const size\_t position) {  
 *// O(log n) - время, просеивание дерева* if (subtree\_size[position] == 0 or subtree\_size[position] == 1)  
 return;  
  
 const size\_t left\_child\_position =  
 position - leonardo\_nums[subtree\_size[position]] + leonardo\_nums[subtree\_size[position] - 1];  
  
 const size\_t right\_child\_position = position - 1;  
  
 if (array[left\_child\_position] > array[right\_child\_position]) {  
 if (array[left\_child\_position] > array[position]) {  
 std::swap(array[position], array[left\_child\_position]);  
 balance\_tree(array, subtree\_size, leonardo\_nums, left\_child\_position);  
 }  
 } else if (array[right\_child\_position] > array[position]) {  
 std::swap(array[position], array[right\_child\_position]);  
 balance\_tree(array, subtree\_size, leonardo\_nums, right\_child\_position);  
 }  
}  
  
  
void sort\_roots(std::vector<long long> &array, std::vector<size\_t> &subtree\_size,  
 const std::vector<size\_t> &leonardo\_nums, size\_t position) {  
 size\_t prev\_root = position - leonardo\_nums[subtree\_size[position]];  
 size\_t left\_child\_position =  
 position + leonardo\_nums[subtree\_size[position] - 1] - leonardo\_nums[subtree\_size[position]];  
 size\_t right\_child\_position = position - 1;  
  
 while (leonardo\_nums[subtree\_size[position]] <= position and array[position] < array[prev\_root] and  
 (subtree\_size[position] < 2 or  
 array[left\_child\_position] < array[prev\_root] and array[right\_child\_position] < array[prev\_root])) {  
 std::swap(array[position], array[prev\_root]);  
 position = prev\_root;  
 left\_child\_position =  
 position + leonardo\_nums[subtree\_size[position] - 1] - leonardo\_nums[subtree\_size[position]];  
 right\_child\_position = position - 1;  
 prev\_root = position - leonardo\_nums[subtree\_size[position]];  
 }  
 *// O(log n) - время, количество корней* balance\_tree(array, subtree\_size, leonardo\_nums, position);  
 *// O(log n) - время, просеивание дерева  
  
 // O(log n) - общее время*}  
  
  
void build\_heap(std::vector<long long> &array, std::vector<size\_t> &subtree\_size,  
 const std::vector<size\_t> &leonardo\_nums) {  
 subtree\_size[0] = 1;  
 subtree\_size[1] = 0;  
 sort\_roots(array, subtree\_size, leonardo\_nums, 1);  
  
 for (size\_t i = 2; i < array.size(); ++i) {  
 std::pair<size\_t, size\_t> prev\_roots = {i - 1, i - leonardo\_nums[subtree\_size[i - 1]] - 1};  
 if (subtree\_size[prev\_roots.first] + 1 == subtree\_size[prev\_roots.second])  
 subtree\_size[i] = subtree\_size[prev\_roots.second] + 1;  
 else if (subtree\_size[i - 1] == 1)  
 subtree\_size[i] = 0;  
 else  
 subtree\_size[i] = 1;  
  
 sort\_roots(array, subtree\_size, leonardo\_nums, i);  
 *// O(log n) - время* }  
 *// O(n \* log n) - общая сложность*}  
  
  
size\_t find\_max\_root\_pos(const std::vector<long long> &array, const std::vector<size\_t> &subtree\_size,  
 const std::vector<size\_t> &leonardo\_nums, size\_t position) {  
 size\_t max\_root\_pos = position;  
 size\_t prev\_root = position - leonardo\_nums[subtree\_size[position]];  
  
 while (leonardo\_nums[subtree\_size[position]] <= position) {  
 if (array[max\_root\_pos] < array[prev\_root])  
 max\_root\_pos = prev\_root;  
  
 position = prev\_root;  
 prev\_root = position - leonardo\_nums[subtree\_size[position]];  
 }  
 *// O(log n) - время, количество корней* return max\_root\_pos;  
}  
  
  
void swap\_with\_max(std::vector<long long> &array, std::vector<size\_t> &subtree\_size,  
 const std::vector<size\_t> &leonardo\_nums, const size\_t position) {  
 const size\_t max\_root\_pos = find\_max\_root\_pos(array, subtree\_size, leonardo\_nums, position);  
 *// O(log n) - время, количество корней* std::swap(array[position], array[max\_root\_pos]);  
 balance\_tree(array, subtree\_size, leonardo\_nums, position);  
 *// O(log n) - время, просеивание дерева* balance\_tree(array, subtree\_size, leonardo\_nums, max\_root\_pos);  
 *// O(log n) - время, просеивание дерева*}  
  
  
void smooth\_sort(std::vector<long long> &array) {  
 if (array.size() <= 1)  
 return;  
  
 const size\_t array\_length = array.size();  
 std::vector<size\_t> leonardo\_nums = create\_leonardo\_numbers(array\_length);  
 *// время и память O(n)* std::vector<size\_t> subtree\_size(array\_length);  
 *// время и память O(n)* build\_heap(array, subtree\_size, leonardo\_nums);  
 *// O(n \* log n) - время* for (int i = array\_length - 2; i >= 0; --i)  
 swap\_with\_max(array, subtree\_size, leonardo\_nums, i);  
 *// O(n \* log n) - время  
  
 // O(n \* log n) - худшее время  
 // O(n) - среднее и лучшее время  
 // O(n) - память*}

ПРИЛОЖЕНИЕ В

Листинг кода файла counting\_sort.cpp

#include "counting\_sort.h"  
  
  
void counting\_sort(std::vector<long long> &input\_vector) {  
 size\_t vector\_size = input\_vector.size();  
 long long max\_element = input\_vector[0];  
 long long min\_element = input\_vector[0];  
  
 for (size\_t i = 0; i < vector\_size; ++i) {  
 max\_element = std::max(max\_element, input\_vector[i]);  
 min\_element = std::min(min\_element, input\_vector[i]);  
 }  
 *// O(n) - время* std::vector<size\_t> count\_vector(max\_element - min\_element + 1, 0);  
 *// O(максимальный элемент - минимальный элемент) - время и память* for (size\_t i = 0; i < vector\_size; ++i)  
 count\_vector[input\_vector[i] - min\_element]++;  
 *// O(n) - время* for (size\_t i = 1; i <= max\_element - min\_element; ++i)  
 count\_vector[i] += count\_vector[i - 1];  
 *// O(максимальный элемент) - время* std::vector<long long> output\_vector(vector\_size);  
 *// O(n) - время и память* for (size\_t i = vector\_size - 1; i >= 0; --i) {  
 output\_vector[count\_vector[input\_vector[i] - min\_element] - 1] = input\_vector[i];  
  
 count\_vector[input\_vector[i] - min\_element]--;  
 if (i == 0)  
 break;  
 }  
 *// O(n) - время* input\_vector = output\_vector;  
  
 *// O(n + максимальный элемент) - время  
 // O(n + максимальный элемент) - память*}