ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Алгоритмы сортировки»

Выполнил работу

Черницын Егор Артёмович

Академическая группа J3114

Принято

Магистр практик Дунаев М. В.

Санкт-Петербург

2024

**ВВЕДЕНИЕ**

Цель работы – научиться выбирать подходящие алгоритмы сортировки исходя из их достоинств и недостатков в тех или иных условиях, научиться анализировать их, понять, как асимптотическая сложность алгоритмов влияет на их практическую применимость.

Задачи:

* Реализовать 3 алгоритма сортировки по заданным критериям на языке C++
* Подсчитать асимптотическую сложность каждого из этих алгоритмов.
* Подсчитать затраты по памяти каждого из этих алгоритмов.
* Написать unit-тесты к программам.
* Отметить на графике зависимость времени выполнения программы от объема входных данных
* Отметить на графике разброс значений времени выполнения каждого из алгоритмов сортировки

**РЕАЛИЗАЦИЯ**

**(код + подсчёт времени и памяти)**

1. **Comb Sort**

****

1. **Heap Sort**

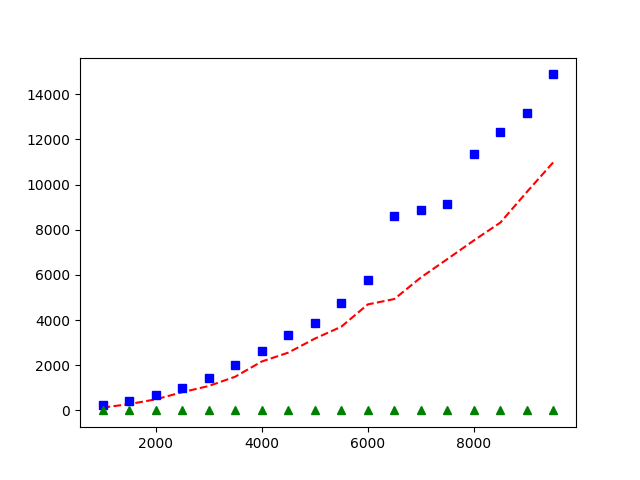
****

1. **Tim Sort**

****

**ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ**

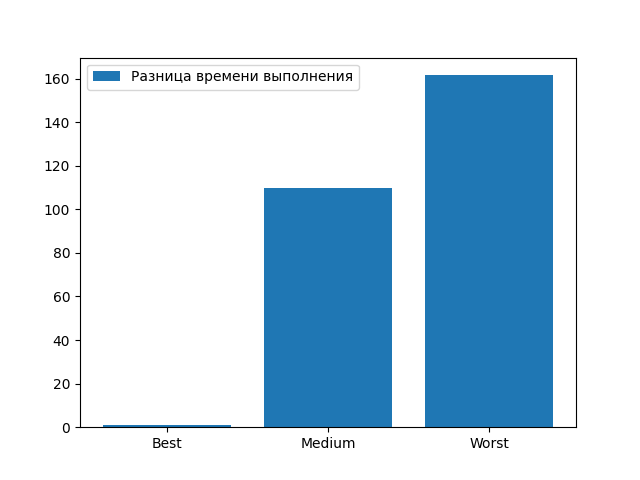
1. **Comb Sort**



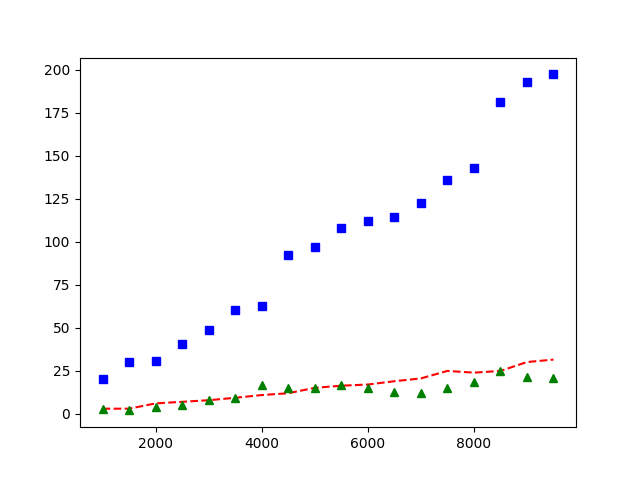
Здесь красная кривая – алгоритм, синяя – n^2, зеленая – n

Unit-тесты для алгоритма:





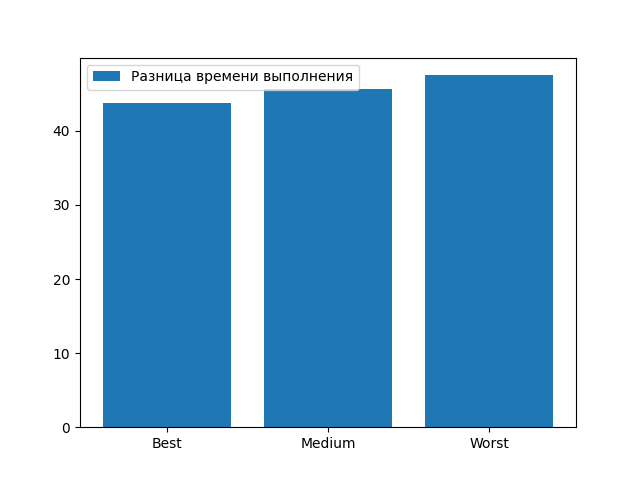
1. **Heap Sort**



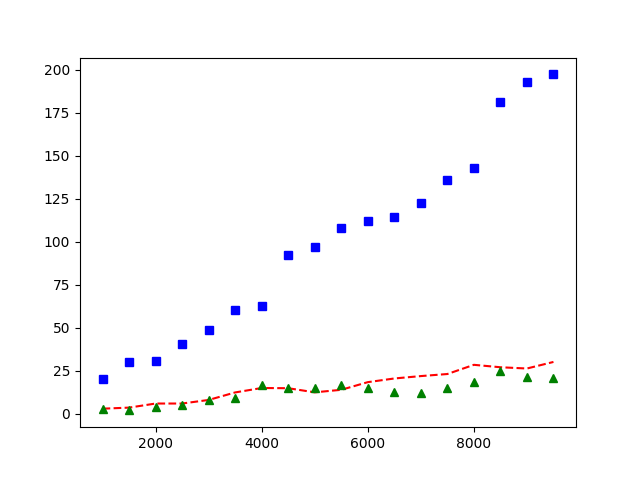
Здесь красная кривая – алгоритм, синяя – n \* (logn)^2, зеленая – n \* logn

Unit-тесты для алгоритма:





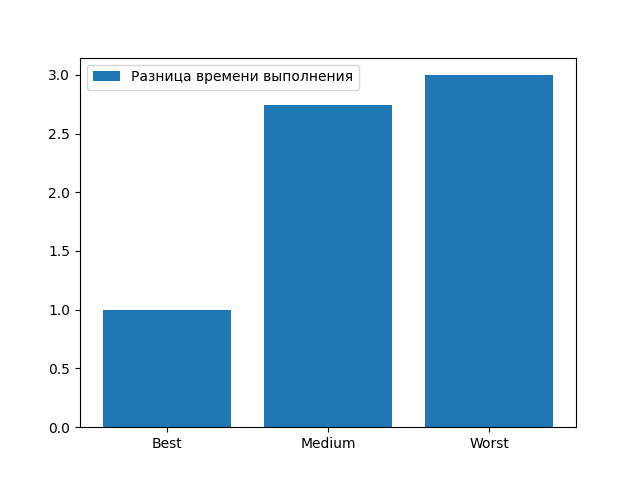
1. **Tim Sort**



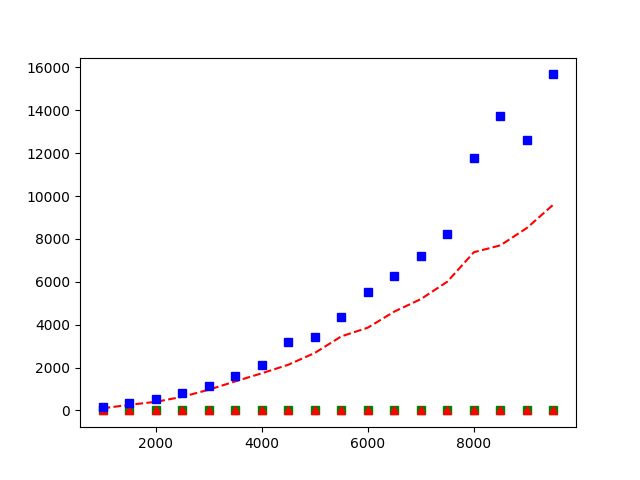
Здесь красная кривая – алгоритм, синяя – n \* (logn)^2, зеленая – n \* logn

Unit-тесты для алгоритма:

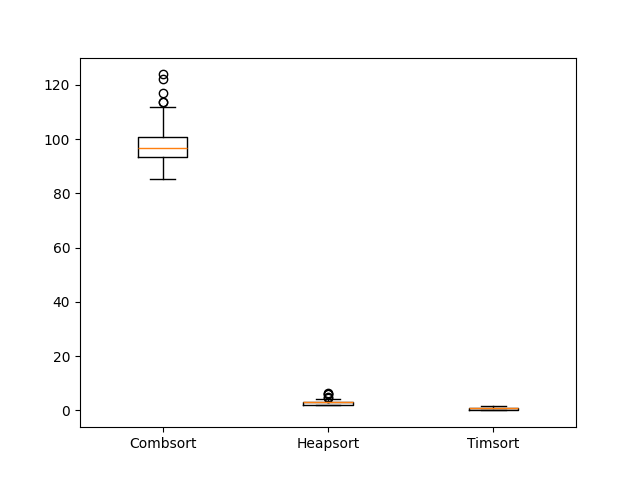
****



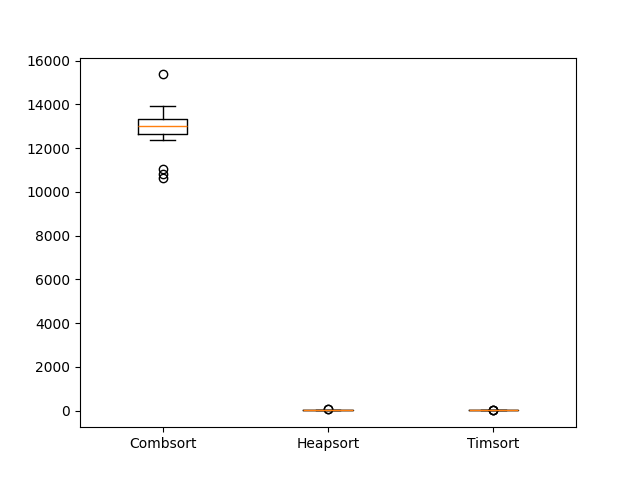
**Общее:**



Здеськрасный прерывистый график – Comb Sort, красный треугольный – это Heap Sort, зеленый квадратный – Tim Sort, синий – n^2



Разброс при 1000 элементов в массиве



Разброс при 10000 элементов в массиве

**ВЫВОД**

**Когда каждая сортировка эффективна?**

1. **Comb Sort:**
   * **Когда эффективна:**  
     Хорошо работает для массивов со случайным распределением элементов, особенно если данные не слишком велики. Подходит для устранения крупных инверсий на ранних этапах. Прост в реализации и не требует дополнительной памяти.
   * **Когда не подходит:**  
     Для больших массивов или почти отсортированных данных уступает адаптивным и более сложным алгоритмам, таким как Timsort.
2. **Heap Sort:**
   * **Когда эффективна:**  
     Идеальна для задач, где требуется гарантированная O(n\*logn) производительность, независимо от структуры данных. Полезна в ограниченных по памяти средах, так как работает без дополнительных массивов.
   * **Когда не подходит:**  
     Если важна стабильность сортировки (Heap Sort не сохраняет порядок равных элементов) или если массив почти отсортирован, она будет менее эффективна, чем Timsort.
3. **Timsort:**
   * **Когда эффективна:**  
     Отлично справляется с реальными данными, которые часто имеют определённую степень упорядоченности. Это делает её предпочтительной для практического использования (например, в Python и Java). Подходит для больших массивов и является стабильной.
   * **Когда не подходит:**  
     Если данные случайны и нет никакой упорядоченности, её преимущества над другими алгоритмами снижаются.

**Как соотносится асимптотика с практикой?**

Мы видим, что heap sort и timsort, имеющие худший случай n \* log n, работают приблизительно одинаково хорошо, быстро и не имеют выбросов. Алгоритм comb sort же, имеющий n^2 худший случай, работает явно медленнее, что видно из графиков, а также имеет выбросы. Так же можно отметить, что в случает comb sort видно, что лучший, средний и худший случаи сильно различаются, в отличие от остальных алгоритмов.

main.cpp

#include <iostream>

#include <vector>

#include <random>

#include <cassert>

using namespace std;

void comb\_sort(int data[], int n) {

// принимаем массив из int размером n \* 4 байт

int step = n; // 4 байта

bool flag = false; // 1 байт

int c = 0; // 4 байта

// раз за разом уменьшаем шаг в 1.25 раз, пока не дойдем до шага в 1 элемент.

// Принцип такой же, как с пузырьком

while (step > 1 or flag) { // как минимум log1.25(n) итераций, как максимум - n (из-за флага)

if (step > 1) {

step = step \* 4 / 5;

}

flag = false;

int i = 0;

while (i + step < n) { // n - step итераций (шаг за шагом приближается к n - 1)

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

flag = true;

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

}

i += 1;

}

}

// имеем всего O(log1.25(n) \* ~(n - 1)) = O(n\*logn) в лучшем случае и O(n^2) в худшем

// затраты по памяти: 4 + 1 + 4 + n \* 4 байт

}

// Процедура для преобразования в двоичную кучу поддерева с корневым узлом i, что является

// индексом в arr[]. n - размер кучи

void heapify(int arr[], int n, int i) // O(n) - доказанная сложность построения кучи из массива.

{

// принимаем массив 4 \* n байт

int largest = i;

// Инициализируем наибольший элемент как корень

int l = 2 \* i + 1; // левый = 2\*i + 1, 4 байта

int r = 2 \* i + 2; // правый = 2\*i + 2, 4 байта

// Если дочерний элемент больше корня

if (l < n && arr[l] > arr[largest])

largest = l;

if (r < n && arr[r] > arr[largest])

largest = r;

if (largest != i)

{

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

heapify(arr, n, largest);

}

// всего по памяти имеем 4 \* n + n / 2 \* 2 \* 4 (каждый элемент из первой половины массива является каким-то корнем, который мы проверяем) = n \* 8 байт

}

// Основная функция, выполняющая пирамидальную сортировку

void heap\_sort(int arr[], int n)

{

// Построение кучи (перегруппируем массив) O(n)

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

heapify(arr, n, i);

}

// Один за другим извлекаем элементы из кучи

// Каждый heapify будет работать за ~log от текущей длины кучи, т.е. <= logn

for (int i = n - 1; i >= 0; i--)

{

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

heapify(arr, i, 0);

}

}

// Имеем цикл n операций, в котором <= logn операций при каждой итерации, и еще цикл из n итераций. В итоге имеем O(n + n\*logn)~O(n\*logn)

int get\_minrun(int n) {

// фактически делим на 2 без остатка

int r = 0;

while (n >= 64) {

r |= (n & 1);

n >>= 1;

}

return n + r;

}

// Сортировка подмассива вставками

void insertion\_sort(vector<int>& arr, int left, int right) {

for (int i = left + 1; i <= right; i++) { // 4 байта

int temp = arr[i]; // 4 байта

int j = i - 1; // 4 байта

while (j >= left && arr[j] > temp) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = temp;

}

// имеем 12 байт в цикле, 8 байт во входнгых переменных и n \* 4 байт из входного массива

}

// Слияние двух отсортированных подмассивов

void merge(vector<int>& arr, int left, int mid, int right) {

// Создаём временные массивы

int len1 = mid - left + 1, len2 = right - mid;

vector<int> leftArr(len1), rightArr(len2); // 24 + 4 \* len1 + 24 + 4 \* len2 байт

for (int i = 0; i < len1; i++) leftArr[i] = arr[left + i]; // 4 байта

for (int i = 0; i < len2; i++) rightArr[i] = arr[mid + 1 + i]; // 4 байта

// Указатели для слияния

int i = 0, j = 0, k = left; // 12 байт

// Слияние массивов

while (i < len1 && j < len2) {

if (leftArr[i] <= rightArr[j]) {

arr[k++] = leftArr[i++];

}

else {

arr[k++] = rightArr[j++];

}

}

// Копируем оставшиеся элементы

while (i < len1) arr[k++] = leftArr[i++];

while (j < len2) arr[k++] = rightArr[j++];

}

// Основной алгоритм TimSort

void timSort(vector<int>& arr) {

int n = arr.size();

int min\_run = get\_minrun(n);

// Сортируем массивы размером minRun с помощью сортировки вставками

// O(n \* min\_run) (pretty ochev)

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

int right = min(i + min\_run - 1, n - 1);

insertion\_sort(arr, i, right);

}

// Слияние отсортированных подмассивов

// Число слияний = log(n/min\_run), само слияние происходит очев за n операций. Итого имеем n \* log(n/min\_run)

for (int size = min\_run; size < n; size \*= 2) {

for (int left = 0; left < n; left += 2 \* size) {

int mid = min(left + size - 1, n - 1);

int right = min(left + 2 \* size - 1, n - 1);

if (mid < right) {

merge(arr, left, mid, right);

}

}

}

// Получили итоговую сложность O(n \* min\_run) + O(n \* log(n / min\_run)) ~ O(n \* logn)

// По памяти же имеем 88 + 4 \* n + 4 \* (len1 + len2) для каждого merge ~ 88 + 4 \* n + 4 \* logn байт

}

void test\_comb() {

int arr[1000] = {};

int correct[1000] = {};

// best

for (int i = 0; i < 1000; ++i) {

arr[i] = i;

correct[i] = i;

}

comb\_sort(arr, 1000);

assert(arr, correct);

// medium

srand((unsigned int)time(NULL));

for (int j = 0; j < 1000; ++j) {

arr[j] = rand();

}

comb\_sort(arr, 1000);

assert(arr, correct);

// worst

for (int k = 0; k < 1000; ++k) {

arr[k] = 999 - k;

}

comb\_sort(arr, 1000);

assert(arr, correct);

}

void test\_heap() {

int arr[1000] = {};

int correct[1000] = {};

// best = medium = worst

for (int i = 0; i < 1000; ++i) {

arr[i] = i;

correct[i] = i;

}

heap\_sort(arr, 1000);

assert(arr, correct);

// best = medium = worst

srand((unsigned int)time(NULL));

for (int j = 0; j < 1000; ++j) {

arr[j] = rand();

}

heap\_sort(arr, 1000);

assert(arr, correct);

// best = medium = worst

for (int k = 0; k < 1000; ++k) {

arr[k] = 999 - k;

}

heap\_sort(arr, 1000);

assert(arr, correct);

}

void test\_tim() {

int arr[1000] = {};

int correct[1000] = {};

// best

for (int i = 0; i < 1000; ++i) {

arr[i] = i;

correct[i] = i;

}

heap\_sort(arr, 1000);

assert(arr, correct);

// medium = worst

srand((unsigned int)time(NULL));

for (int j = 0; j < 1000; ++j) {

arr[j] = rand();

}

heap\_sort(arr, 1000);

assert(arr, correct);

// medium = worst

for (int k = 0; k < 1000; ++k) {

arr[k] = 999 - k;

}

heap\_sort(arr, 1000);

assert(arr, correct);

}

void main()

{

int arr[] = { 5, 6, 7, 12, 14, 13 };

int n = sizeof(arr) / sizeof(arr[0]);

test\_comb();

test\_heap();

test\_tim();

}