

Санкт-Петербургский политехнический университет Петра Великого

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Тема: Сравнение алгоритмов сортировок: Quick sort, Heap sort, Bogosort, Insertion sort и Bubble sort.

Студент гр. 3331506/90001

Клиновицкий А.Д.

Преподаватель

Ананьевский М.С.

Санкт-Петербург

2022

СОДЕРЖАНИЕ

Оглавление

ЗАДАНИЕ	3
1. ПРИНЦИП СОРТИРОВКИ.....	4
1.1 Quick sort	4
1.2 Heap sort	6
1.3 Insertion sort	6
1.4 Bubble sort	7
1.5 Bogosort	8
2 РЕАЛИЗАЦИЯ	9
3. ЗАТРАТЫ ПО ОПЕРАЦИЯМ И ВРЕМЕНИ И ОПЕРАЦИЯМ.....	13
3.1 Экспериментальные данные	13
3.2 Среднеквадратичное отклонение. Время на сортировку в зависимости от количества элементов	15
ВЫВОД	18

ЗАДАНИЕ

Требуется провести сравнение пяти разных алгоритмов сортировок по некоторым признакам:

- Времени
- Количеству смен местами элементов массива

1. ПРИНЦИП СОРТИРОВКИ

1.1 Quick sort

Начать стоит с самого популярного способа сортировки, а именно Quick sort (быстрая сортировка). Принцип его работы прост:

1) Выбирается опорный элемент из массива. От выбора опорного элемента не зависит корректность сортировки, но может сильно зависеть его скорость;

2) Сравниваются все элементы массива и переставляются так, чтобы весь массив был разбит на три части: меньшие, чем опорный элемент (“слева”), равные ему и большие (“справа”). Также иногда делят не на три части, а на две: меньшие, чем опорный элемент и большие или равные, так как это упрощает алгоритм разделения;



3) Пока длина отрезка больше единицы, повторяется рекурсивно вызов для каждого из отрезков.

Пример работы алгоритма показан на рисунке 1.1.1.

Средняя сложность алгоритма $O(n \log n)$, а худшая $O(n^2)$.

4	2	6	5	3	9
---	---	---	---	---	---

4	2	6	5	3	9
↑ L					↑ R

4	2	6	5	3	9
					

4	2	6	5	3	9
		L			R

4	2	6	5	3	9
---	---	---	---	---	---

↑ ↑
L R

Diagram illustrating a queue with elements 4, 2, 3, 5, 6, 9. The element 3 is highlighted in blue and labeled 'L' (Left). The element 6 is highlighted in blue and labeled 'R' (Right). The element 5 is highlighted in yellow.

4	2	3	5	6	9
---	---	---	---	---	---

↑ ↑
L R

4	2	3	5	6	9
---	---	---	---	---	---

↑ ↑
R L

Рисунок 1.1.1 – Принцип работы алгоритма Quick sort.

1.2 Heap sort

Heap sort, также известная как пирамидальная сортировка/сортировка кучей, использует в принципе своей работы бинарно сортирующее дерево, изображенное на рисунке 1.2.1. У такого дерева должны быть выполнены некоторые условия:

- 1) Каждый лист дерева имеет глубину d или $d - 1$, где d – максимальная глубина дерева;
- 2) Значение в любой вершине не меньше (или не больше для другого случая) значений ее потомков.

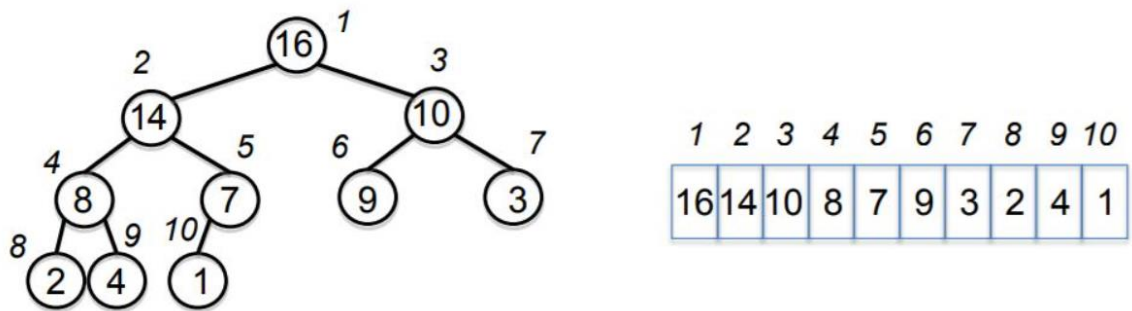


Рисунок 1.2.1 - Бинарное сортирующее дерево

После постройки дерева, чтобы получить отсортированный массив, удаляем элементы из корня по одному и перестраиваем дерево, повторяя процесс, пока в дереве не останется всего один элемент.

Средняя сложность алгоритма $O(n \log n)$, она не зависит от входного массива и всегда такова.

1.3 Insertion sort

Insertion sort, также известная как сортировка вставками, работает на данном принципе:

- 1) Массив делится на две части, отсортированную (“левая”) и нет (“правая”);

2) Берется один элемент из неотсортированной части массива и вставляется в сортированную, в соответствии с необходимым местоположением;

3) Процесс повторяется, до окончания неотсортированной части массива.

Пример работы сортировки вставками показан на рисунке 1.3.1.

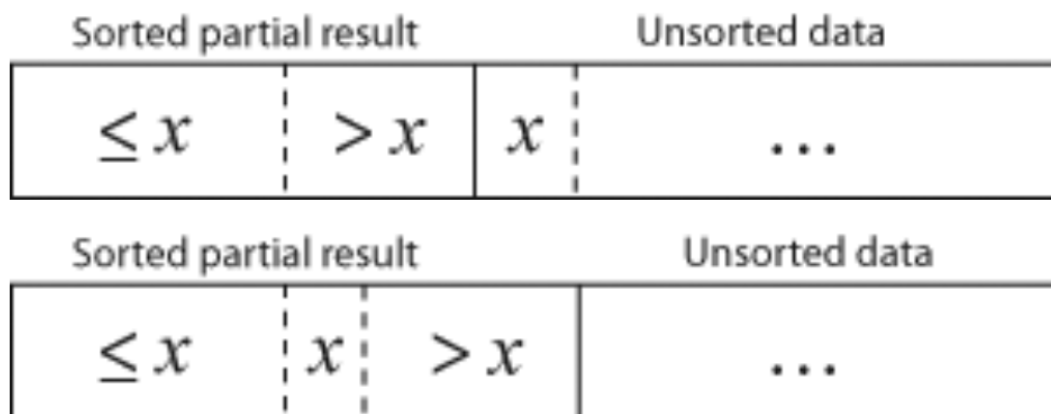


Рисунок 1.3.1 – Принцип работы сортировки вставками

Средняя сложность алгоритма $O(n^2)$, такая же, как и худшая.

1.4 Bubble sort

Bubble sort (Сортировка пузырьком) – метод сортировки массива, путем последовательного сравнения и обмена соседних элементов, если последующий меньше предыдущего. Принцип его работы можно описать так:

1) Проходим массив от первого элемента до последнего, сравнивая их друг с другом, и если предыдущий элемент больше последующего, меняя их;

2) Повторяем этот процесс столько раз, сколько элементов в массиве.

Пример работы алгоритма показан на рисунке 1.4.1.

Данный алгоритм плох в реальном мире и используется как демонстрация для обучения.

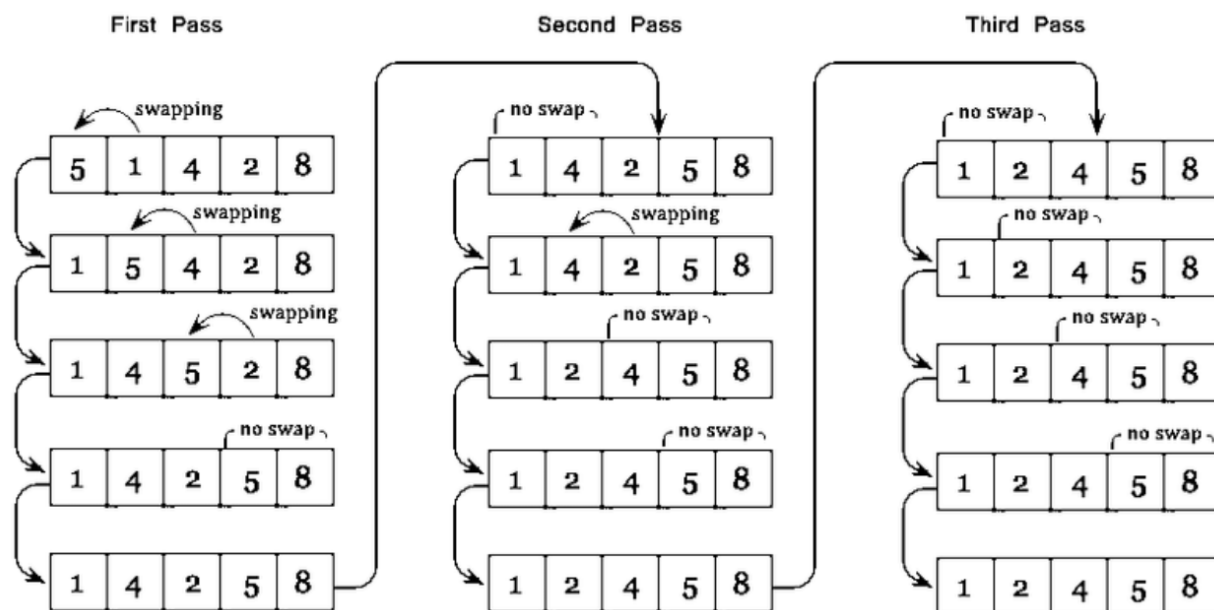


Рисунок 1.4.1 – Принцип работы Bubble sort

Средняя сложность алгоритма $O(n^2)$, такая же, как и худшая.

1.5 Bogosort

Bogosort – самый неэффективный алгоритм сортировки, используемый только в образовательных целях. Принцип работы его прост, пока массив не отсортирован, он случайно меняет элементы массива в нем.

Принцип работы алгоритма показан на рисунке 1.5.1.

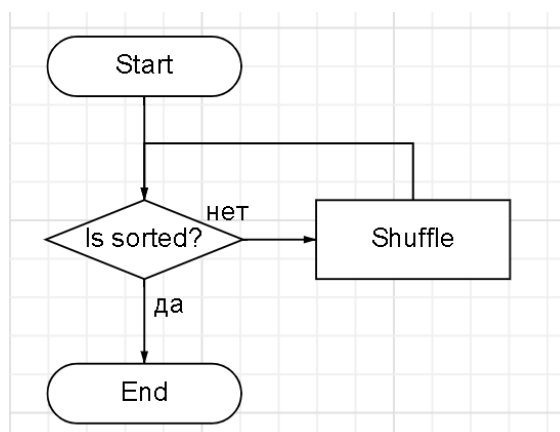


Рисунок 1.5.1 - Принцип работы Bogosort

Вне зависимости от входных данных сложность алгоритма $O(n \cdot n!)$,

2 РЕАЛИЗАЦИЯ

Для наглядности процессов сортировки была написана программа, визуализирующая массив. Пример отсортированного массива и перемешенного показан на рисунках 2.1 и 2.2. Ниже продемонстрированы пять рисунков (2.3, 2.4, 2.5, 2.6, 2.7), соответствующие Quick sort, Heap sort, Insertion sort, Bubble sort и Bogosort для демонстрации процесса сортировки массива из тысячи элементов.

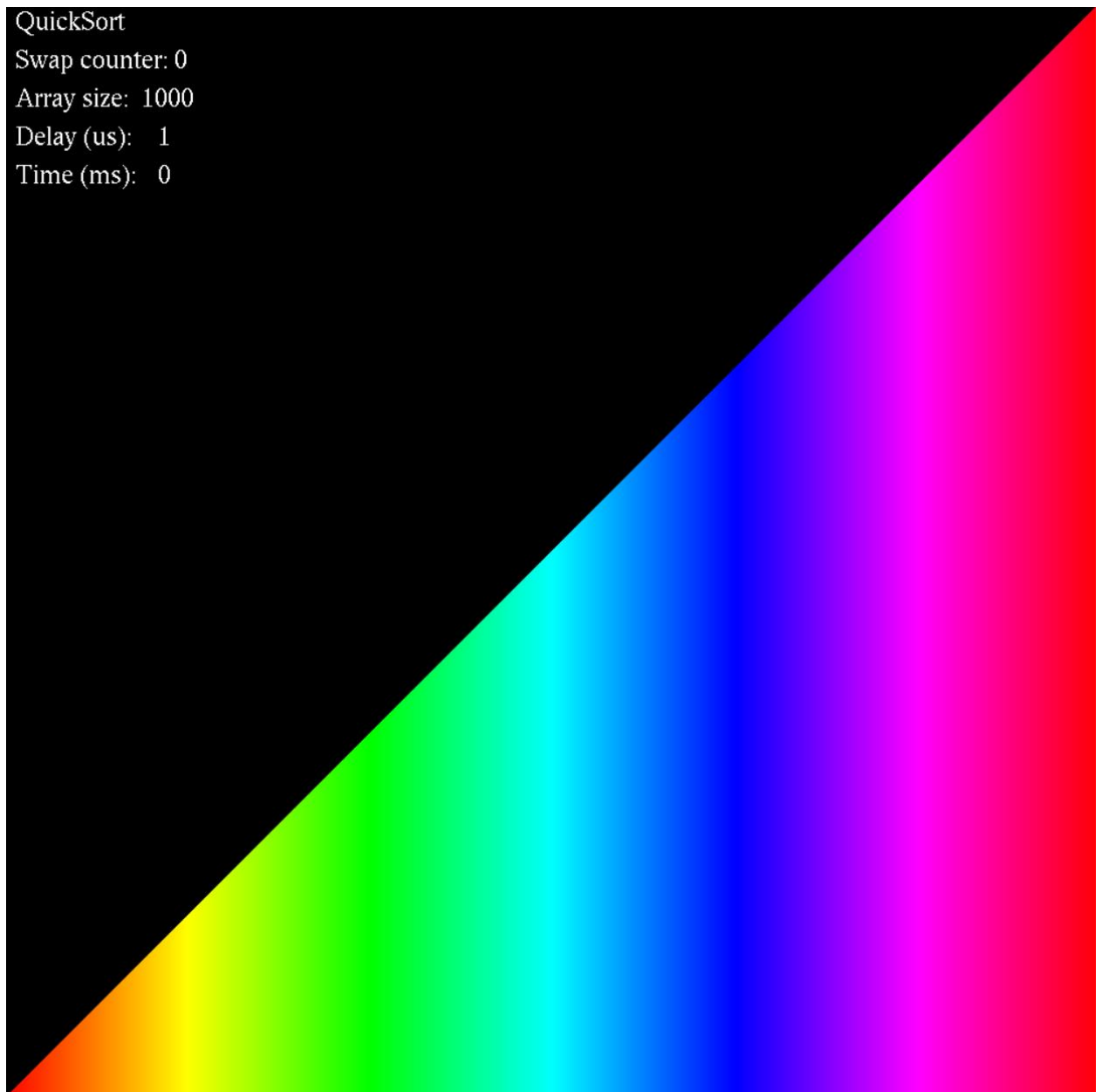


Рисунок 2.1 – Отсортированный массив на тысячу элементов

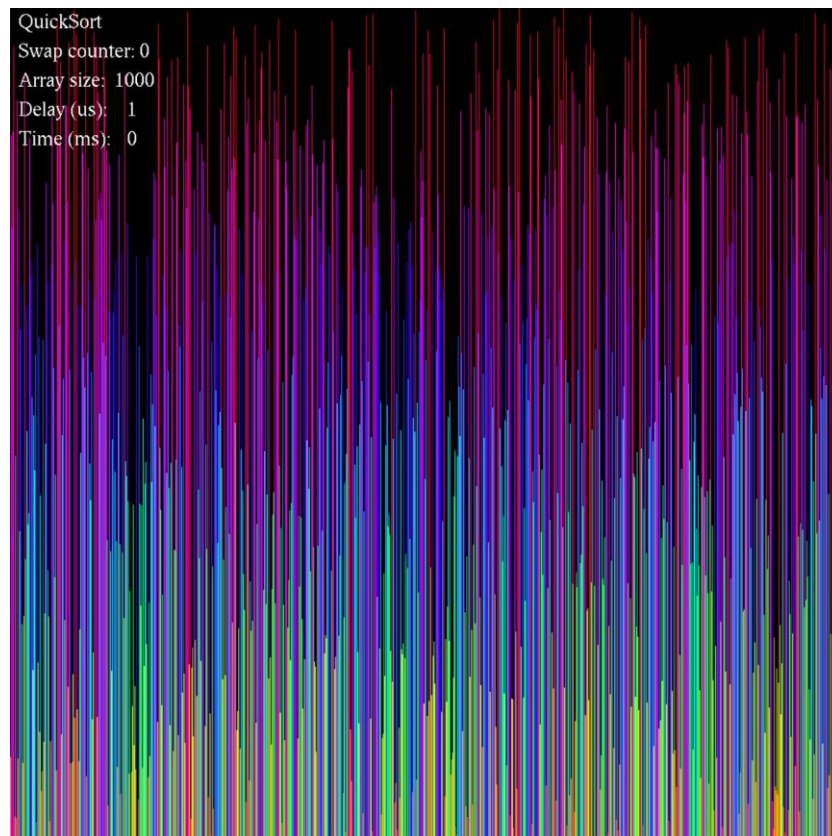


Рисунок 2.2 – Перемешанный массив на тысячу элементов

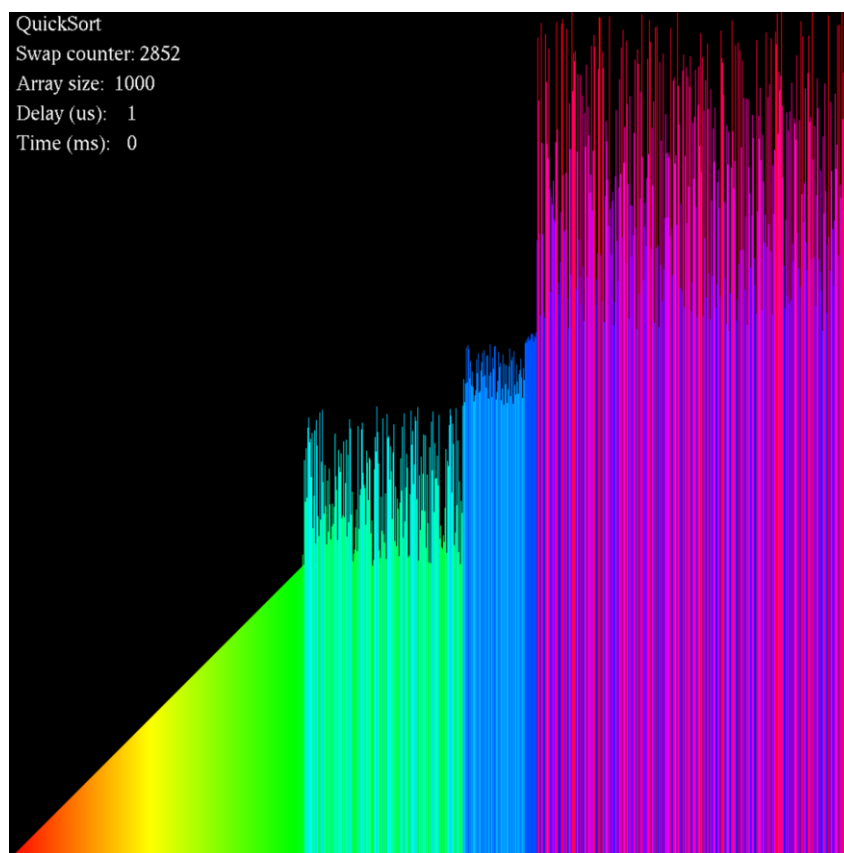


Рисунок 2.3 – Процесс сортировки массива при помощи Quick sort

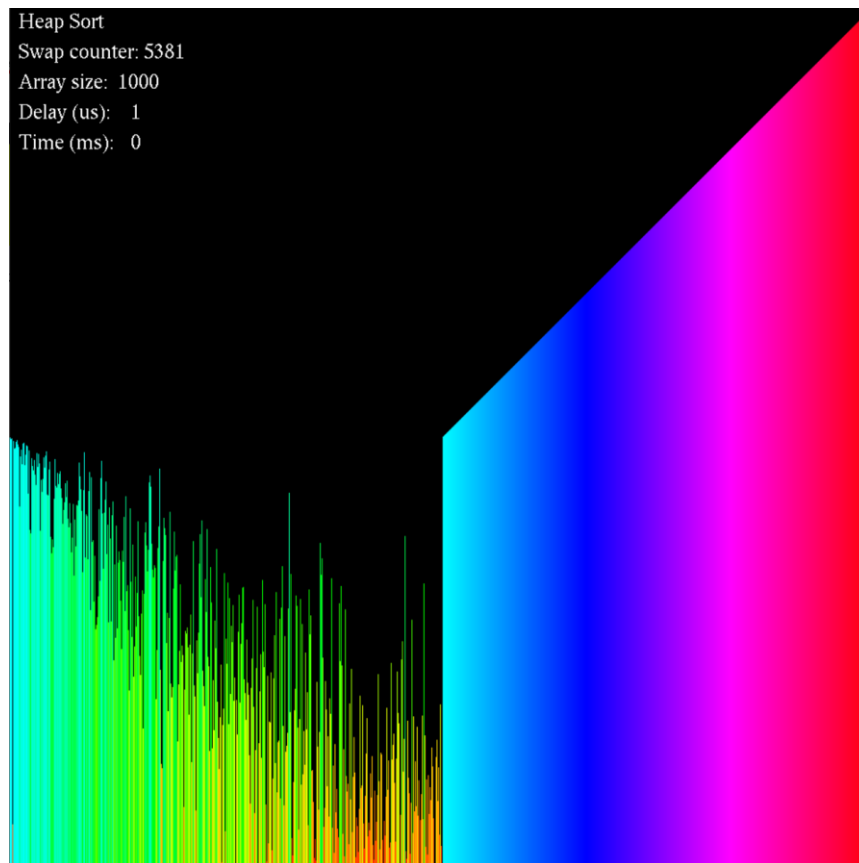


Рисунок 2.4 – Процесс сортировки массива при помощи Heap sort

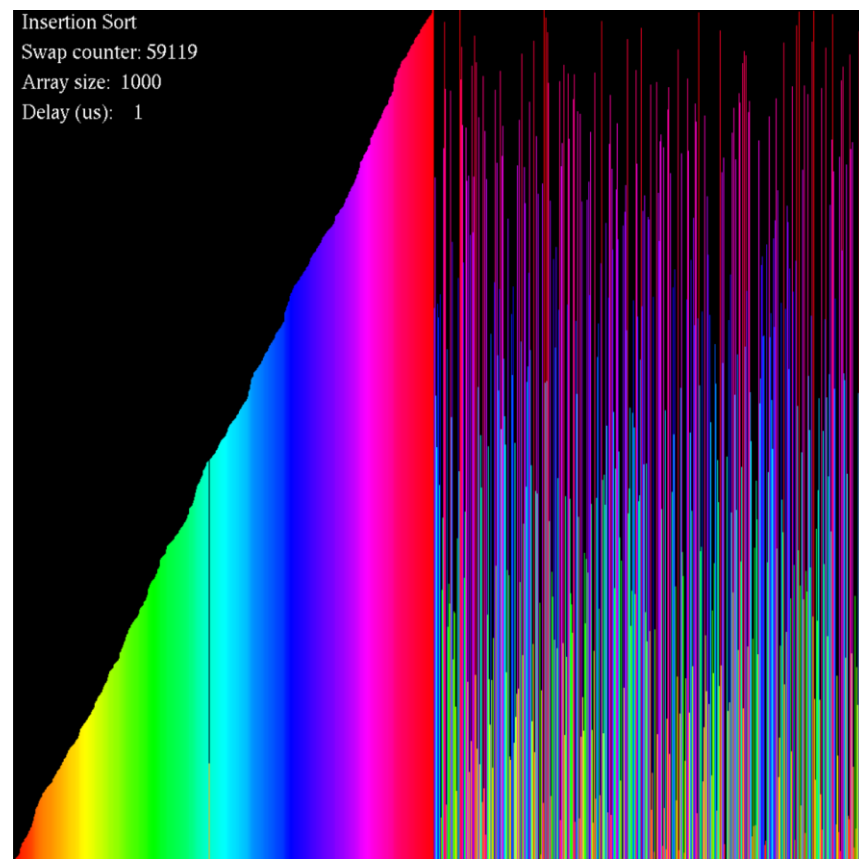


Рисунок 2.5 – Процесс сортировки массива при помощи Insertion sort

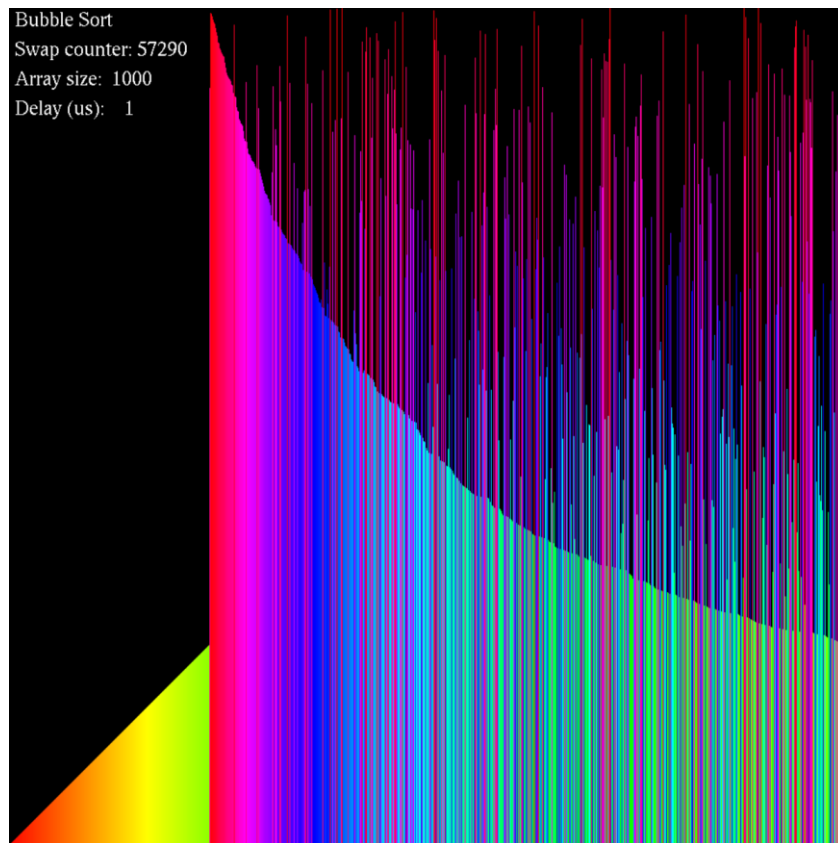


Рисунок 2.6 – Процесс сортировки массива при помощи Bubble sort

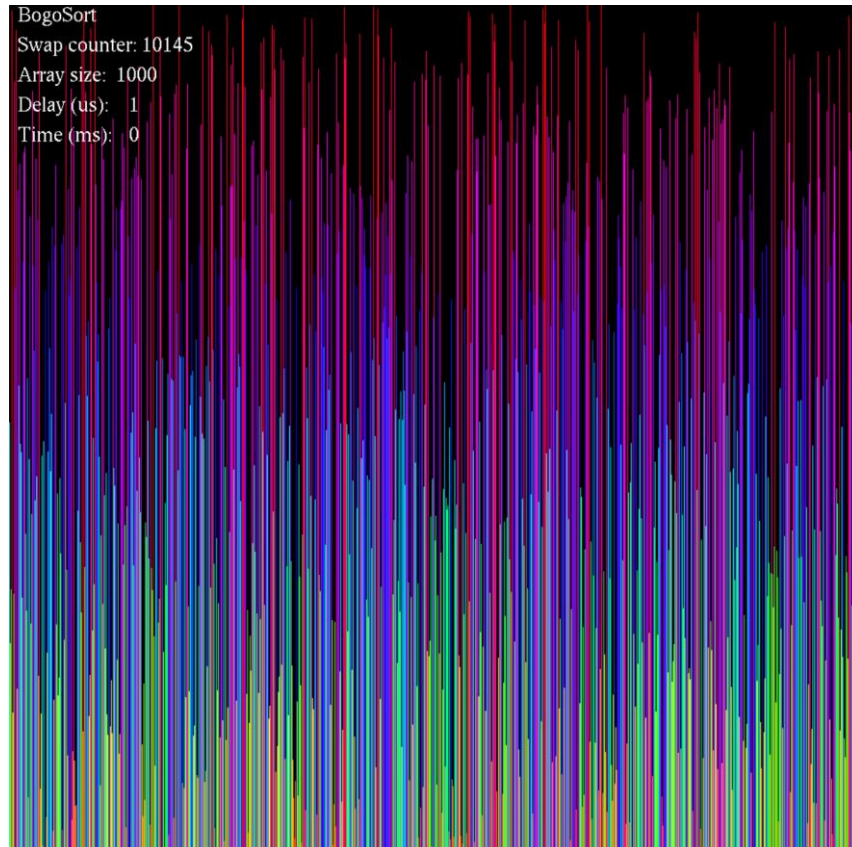


Рисунок 2.7 – Случайный массив Bogosort

3. ЗАТРАТЫ ПО ОПЕРАЦИЯМ И ВРЕМЕНИ И ОПЕРАЦИЯМ

3.1 Экспериментальные данные

Проведем суммарно сто опытов, по пять на каждый из типов сортировок, для массивов размером 5, 10, 100 и 1000 элементов и запишем все в таблицу 1. Задержка перед обменами элементов для 5 элементов – 1024 мкс, а для остальных случаев – 1 мкс. Также, при каждом смене элементов местами, мы считаем событие и заносим в экспериментальную таблицу.

Таблица 1 - Экспериментальные данные

№	N, шт	Quick sort		Heap sort		Insertion sort		Bubble sort		Bogosort	
		Swap, шт	t, мс	Swap, шт	t, мс	Swap, шт	t, мс	Swap, шт	t, мс	Swap, шт	t, мс
1	5	7	15	8	16	3	9	3	8	29	57
2	5	5	12	8	15	2	7	1	4	54	107
3	5	8	18	9	19	7	17	3	9	839	1594
4	5	5	12	10	21	3	9	6	14	3069	5940
5	5	6	13	9	21	7	15	2	8	344	670
1	10	16	2	21	1	24	3	26	2	3,2E+07	-
2	10	18	1	28	2	15	1	25	2	-	-
3	10	18	2	25	1	22	5	29	2	-	-
4	10	18	0	28	1	25	2	15	1	-	-
5	10	21	3	26	1	18	1	24	1	-	-
1	100	460	5	578	7	2471	476	2375	394	-	-
2	100	397	5	561	8	2399	448	2469	461	-	-
3	100	453	6	589	10	2485	485	2336	385	-	-
4	100	399	6	588	7	2183	312	2444	241	-	-
5	100	421	8	584	7	2310	401	2590	521	-	-
1	1000	5793	2218	9059	3249	246479	85660	247505	84888	-	-
2	1000	5672	2065	9100	3298	251560	88715	259800	86372	-	-
3	1000	6388	2414	9106	3392	252936	87488	253672	91818	-	-
4	1000	6725	2464	9079	3401	247804	86179	248081	88610	-	-
5	1000	5245	1907	9112	3128	239657	83630	253783	90636	-	-

Таким образом, Bogosort не удалось отсортировать даже 10 элементов и спустя 4 часа сортировки эксперимент пришлось прервать. Количество смен элементов местами показано на рисунке 3.1.1.

По данным из таблицы 1, посчитаем средние значения и занесем их в таблицу 2.

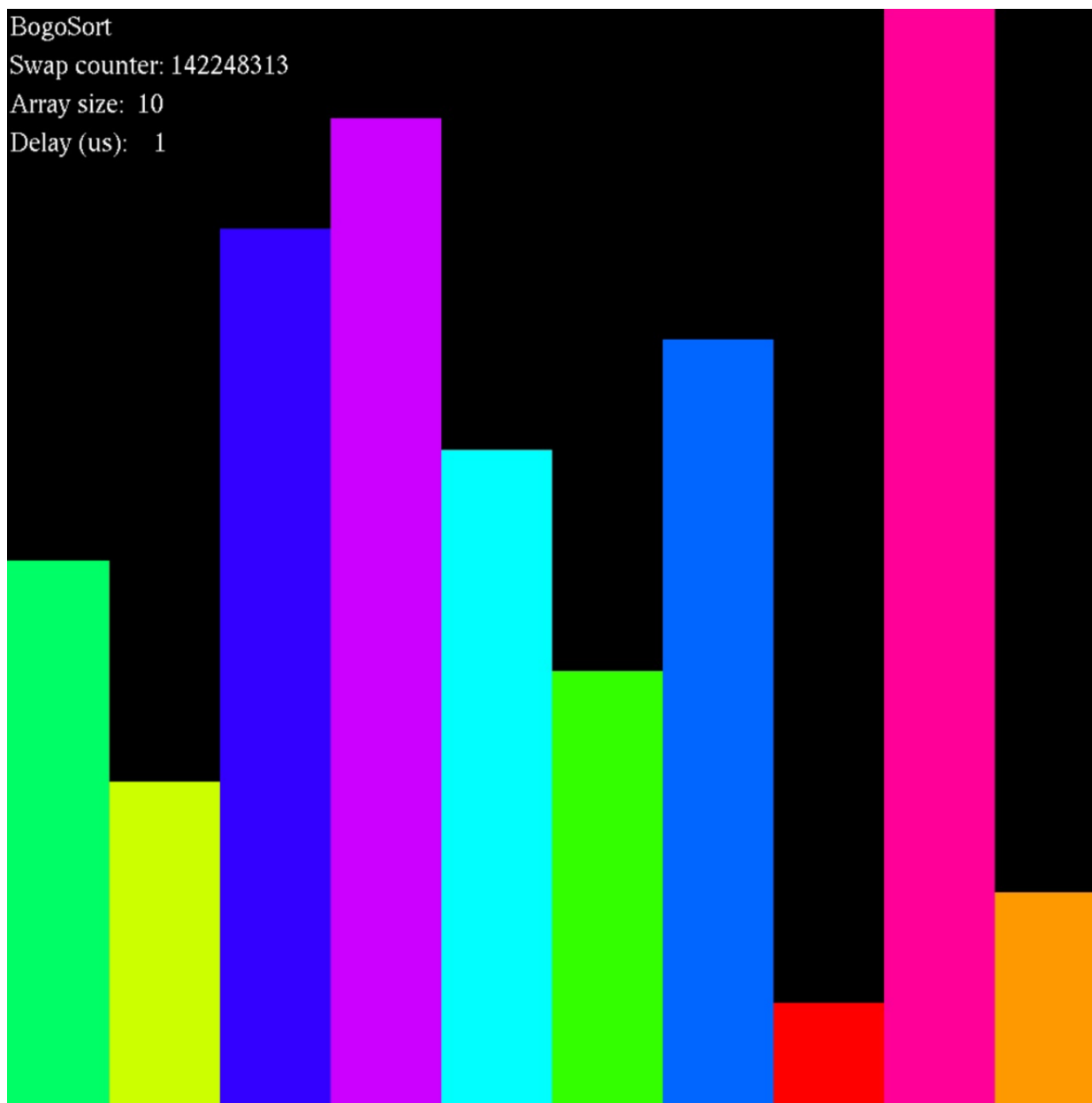


Рисунок 3.1.1 – Прекращение сортировки при помощи Bogosort.

Таблица 2 - Средние показатели

N, шт	Quick sort		Heap sort		Insertion sort		Bubble sort		Bogosort	
	Swap, шт	t, мс	Swap, шт	t, мс	Swap, шт	t, мс	Swap, шт	t, мс	Swap, шт	t, мс
5	6,2	14	8,8	18,4	4,4	11,4	3	8,6	867	1673,6
10	18,2	1,6	25,6	1,2	20,8	2,4	23,8	1,6	-	-
100	426	6	580	7,8	2369,6	424,4	2442,8	400,4	-	-
1000	5964,6	2213,6	9091,2	3293,6	247687	86334,4	252568	88464,8	-	-

Построим график количества обмена элементов в массиве от максимального количества элементов в массиве и изобразим это, на рисунке 3.1.2.

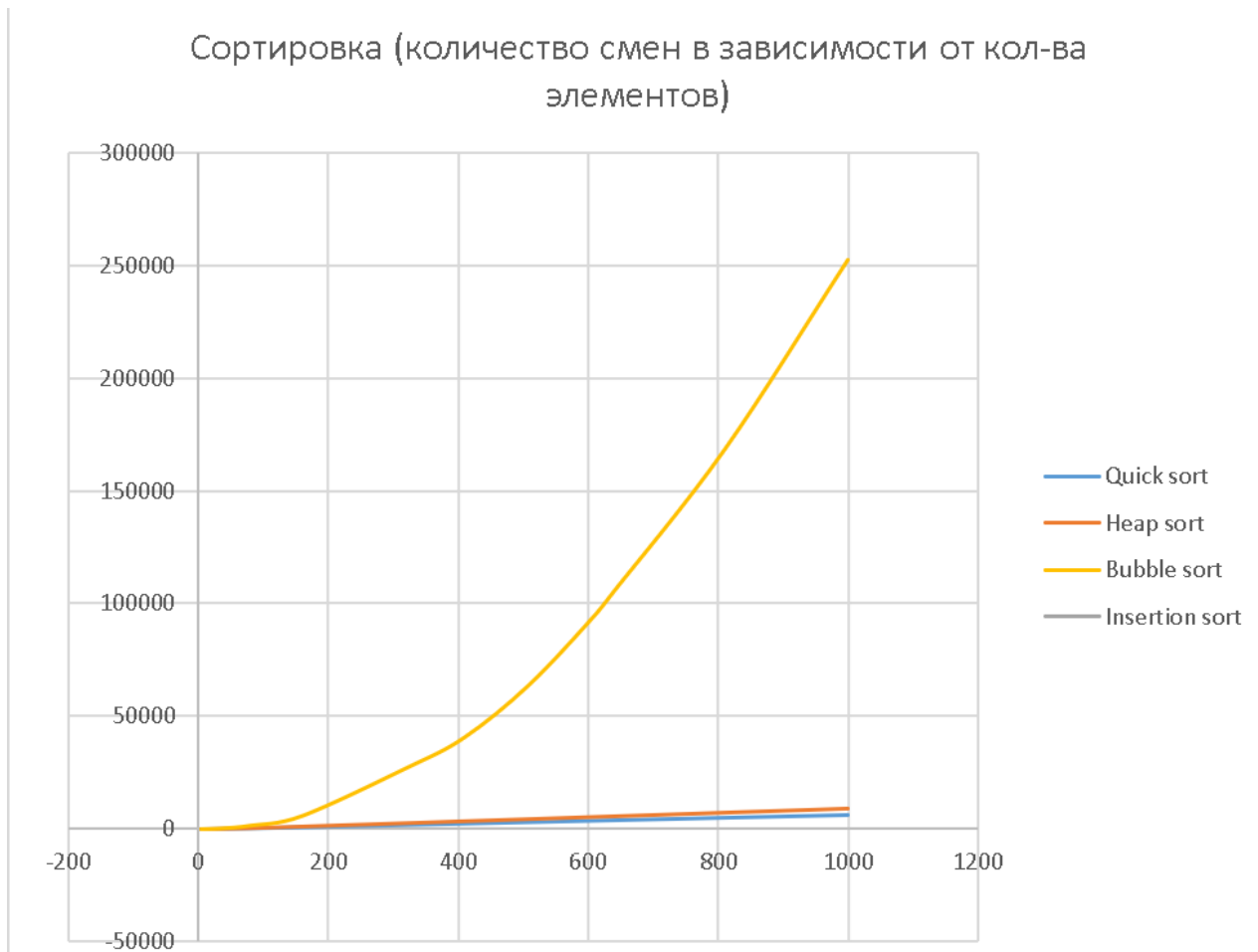


Рисунок 3.1.2 – Зависимость количества обмена переменных в зависимости от количества элементов в массиве

3.2 Среднеквадратичное отклонение. Время на сортировку в зависимости от количества элементов

Составим таблицу 3 среднеквадратичного отклонения для всех экспериментальных данных из таблицы 1.

Таблица 3 - Среднеквадратичное отклонение

№	N, шт	Quick sort		Heap sort		Insertion sort		Bubble sort		Bogosort	
		σ_{swap}	σ_{time}	σ_{swap}	σ_{time}	σ_{swap}	σ_{time}	σ_{swap}	σ_{time}	σ_{swap}	σ_{time}
1	5	0,64	1	0,64	5,76	1,96	5,76	0	0,36	702244	2613396
2	5	1,44	4	0,64	11,56	5,76	19,36	4	21,16	660969	2454236
3	5	3,24	16	0,04	0,36	6,76	31,36	0	0,16	784	6336,16
4	5	1,44	4	1,44	6,76	1,96	5,76	9	29,16	4848804	1,8E+07
5	5	0,04	1	0,04	6,76	6,76	12,96	1	0,36	273529	1007213
1	10	4,84	0,16	21,16	0,04	10,24	0,36	4,84	0,16	-	-
2	10	0,04	0,36	5,76	0,64	33,64	1,96	1,44	0,16	-	-
3	10	0,04	0,16	0,36	0,04	1,44	6,76	27,04	0,16	-	-
4	10	0,04	2,56	5,76	0,04	17,64	0,16	77,44	0,36	-	-
5	10	7,84	1,96	0,16	0,04	7,84	1,96	0,04	0,36	-	-
1	100	1156	1	4	0,64	10282	2662,56	4596,84	40,96	-	-
2	100	841	1	361	0,04	864,36	556,96	686,44	3672,36	-	-
3	100	729	0	81	4,84	13317,2	3672,36	11406,2	237,16	-	-
4	100	729	0	64	0,64	34819,6	12633,8	1,44	25408,4	-	-
5	100	25	4	16	0,64	3552,16	547,56	21667,8	14544,4	-	-
1	1000	29446,6	19,36	1036,84	1989,16	1459747	454815	2,6E+07	1,3E+07	-	-
2	1000	85614,8	22082	77,44	19,36	1,5E+07	5667256	5,2E+07	4379812	-	-
3	1000	179268	40160,2	219,04	9682,56	2,8E+07	1330793	1218374	1,1E+07	-	-
4	1000	578208	62700,2	148,84	11534,8	13642,2	24149,2	2E+07	21083	-	-
5	1000	517824	94003,6	432,64	27423,4	6,4E+07	7313779	1475739	4714109	-	-

Таким образом видно, что Bogosort имеет максимальное среднеквадратичное отклонение даже на 5 элементах.

Теперь занесем время на один обмен в таблицу 4.

Таблица 4 - Время на один обмен

N, шт	Quick sort	Heap sort	Insertion sort	Bubble sort	Bogosort
5	0,442857143	0,47826087	0,385964912	0,348837209	0,518044933
10	11,375	21,33333333	8,666666667	14,875	-
100	71	74,35897436	5,583411876	6,100899101	-
1000	2,694524756	2,760262327	2,86892826	2,85501352	-

Посмотрев на время на один обмен для 1000 элементов, можно получить самые правдивые данные. Как видим, сортировка вставками и пузырьком дольше всего оперируют данными перед обменом, а быстрая сортировка оказалась быстрее всех.

Теперь посмотрим, сколько требуется времени на элемент, который необходимо отсортировать потребуется, в зависимости от типа сортировки и занесем данные в таблицу 5.

Таблица 5 - Время на сортировку (Время/элементы)

N, шт	Quick sort	Heap sort	Insertion sort	Bubble sort	Bogosort
5	2,8	3,68	2,28	1,72	334,72
10	0,16	0,12	0,24	0,16	-
100	0,06	0,078	4,244	4,004	-
1000	2,2136	3,2936	86,3344	88,4648	-

После всех данных, которые мы получили, можем опытным путем проверить временную сложность алгоритмов.

ВЫВОД

В результате проведения работы были разобраны пять разных типов сортировки, проведен их сравнительный анализ по времени и количеству смен элементов, а также экспериментальным способом получена средняя временная сложность алгоритмов.

КОД ПРОГРАММЫ

```
#include <algorithm>           //Swap
#include "GL/freeglut.h"       //Graphic
#include <iostream>             //IO
#include <stdio.h>              //IO
#include <time.h>               //Rand seed, time
#include <unistd.h>             //POSIX API (Sleep)
#include <thread>               //Multithreading
#include <stdlib.h>             //Standart lib

//WIP B-tree
//#define MAX 3
//#define MIN 2
//#define MAX RAND 100
//#define AMOUNT 100

//WIP
//#include "GL/gl.h"
//#include "GL/glut.h"
//#include "GL/glext.h"

int length = 10;
int delay = 10000;
int *arr;
int swapCounter = 0;
int sortIndex = 0;
char sortName[5][15] = {"QuickSort", "Bubble Sort", "BogoSort", "Insertion Sort", "Heap Sort"};
int typeSort = 0;
int startTime;
int endTime;
int sortTime = 0;

int main(int argc, char *argv[]);

void counterMessage(std::string message);

//-----Array-----
void fillArray();

void randomizeArray(int *a, int length);

void swap(int index1, int index2);

void arrayColor(int i, float l);
//^^^^^^^^^^^^^^^^^^^^Array^^^^^^^^^^^^^^^^^^^^

int setUp(int argc, char **argv);

void render();

void displayText(float x, float y, float red, float green, float blue, char *string);

void polygon(float leftX, float rightX, float bottomY, float topY);

void keyboardEvent(unsigned char key, int x, int y);

//-----QuickSort-----
void visualizeQuickSort(int *a, int length);
```

```
void quickSort(int *a, int low, int high);

int partition(int *a, int low, int high);
//^^^^^^^^^^^^^^^^^^^^QuickSort^^^^^^^^^^^^^^^^^^^^

//-----Bubble Sort-----
void visualizeBubbleSort(int *a, int length);
//^^^^^^^^^^^^^^^^^^^^Bubble Sort^^^^^^^^^^^^^^^^^^^^

//-----BogoSort-----
void visualizeBogoSort(int *a, int length);

void shuffle(int *a, int length);

bool isSorted(int *a, int length);
//^^^^^^^^^^^^^^^^^^^^BogoSort^^^^^^^^^^^^^^^^^^^^

//-----Insertion Sort-----
void visualizeInsertionSort(int *a, int length);
//^^^^^^^^^^^^^^^^^^^^Insertion Sort^^^^^^^^^^^^^^^^^^^^

//-----Heap Sort-----
void visualizeHeapSort(int *a, int length);

void heapify(int *a, int length, int i);
//^^^^^^^^^^^^^^^^^^^^Heap Sort^^^^^^^^^^^^^^^^^^^^

void isItRunning();

void (*sort)(int *, int);

/*
//WIP
//-----B-tree-----
void renderBTree(int *pos, struct BTreeNode *myNode);

int bTree();

void insertNode(int val, int pos, struct BTreeNode *node, struct BTreeNode
*child);

void splitNode(int val, int *pval, int pos, struct BTreeNode *node,
struct BTreeNode *child, struct BTreeNode **newNode);

int setValue(int val, int *pval,
struct BTreeNode *node, struct BTreeNode **child);

void insert(int val);

void insert(int val);

void search(int val, int *pos, struct BTreeNode *myNode);

void traversal(struct BTreeNode *myNode);
//^^^^^^^^^^^^^^^^^^^^B-tree^^^^^^^^^^^^^^^^^^^^
*/

int main(int argc, char *argv[]) {
    srand(time(NULL));

    fillArray();
    sort = visualizeQuickSort;
```

```

setUp(argc, argv);

//printf("1"); //DEBUG
free(arr);
//printf("2"); //DEBUG
}

void counterMessage(std::string message) {
    std::cout << "\nSwap count in " << message << ": " << swapCounter <<
    std::endl;
    swapCounter = 0;
}

void isItRunning() {
    MSG msg;
    if (::PeekMessage(&msg, 0, 0, 0, PM_REMOVE)) {
        ::TranslateMessage(&msg);
        ::DispatchMessage(&msg);
    }
}

//-----Array-----
void fillArray() {
    free(arr);
    arr = (int *) malloc(sizeof(int) * length);
    for (int i = 0; i < length; ++i)
        arr[i] = i;
}

void randomizeArray(int *a, int length) {
    for (int i = length - 1; i > 0; --i) {
        std::swap(a[i], a[rand() % (i + 1)]);
    }
}

void swap(int index1, int index2) {
    std::swap(arr[index1], arr[index2]);
    render();
    swapCounter++;
    usleep(delay);
}

void arrayColor(int i, float l) {
    if (arr[i] < (l / 6)) glColor3f(1, 6 * arr[i] / l, 0);
    else if (arr[i] < (l / 3)) glColor3f(1 - ((arr[i] - (l / 6)) * (6 / l)),
1, 0);
    else if (arr[i] < (l / 2)) glColor3f(0, 1, (arr[i] - (l / 3)) * (6 / l));
    else if (arr[i] < (2 * l / 3)) glColor3f(0, 1 - ((arr[i] - (l / 2)) * (6
/ l)), 1);
    else if (arr[i] < (5 * l / 6)) glColor3f((arr[i] - (2 * l / 3)) * (6 /
l), 0, 1);
    else glColor3f(1, 0, 1 - ((arr[i] - (5 * l / 6)) * (6 / l)));
}
//^^^^^^^^^^^^^^^^^^^^Array^^^^^^^^^^^^^^^^^^^^

int setUp(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE);
    glutInitWindowSize(1000, 1000);
    glutInitWindowPosition(50, 50);
    glutCreateWindow("Sort");
}

```

```

    glutDisplayFunc(render);
    glutKeyboardFunc(keyboardEvent);

    //printf("4"); //DEBUG
    //usleep(1000000);
    //printf("5"); //DEBUG

    glutMainLoop();
}

/*
//WIP
void renderBTree(int *pos, struct BTreeNode *myNode) {
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
}
*/

void render() {
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
    float l = (float) length;
    float arrayWidth = 2 / l;
    switch (typeSort % 2) {
        case 0:
            for (int i = 0; i < length; ++i) {
                glBegin(GL_POLYGON);

                float arrayHeight = 2 * (arr[i] + 1) / l;
                float arrayWidthSum = 2 * i / l;

                float leftX = -1 + arrayWidthSum;
                float rightX = leftX + arrayWidth;
                float bottomY = -1;
                float topY = bottomY + arrayHeight;

                arrayColor(i, l);

                polygon(leftX, rightX, bottomY, topY);
                glEnd();
            }
            break;
        case 1:
            for (int i = 0; i < length; ++i) {

                float arrayHeight = 2 * (arr[i] + 1) / l;
                float arrayWidthSum = 2 * i / l;
                float midX = -1 + arrayWidthSum + arrayWidth;
                float midY = -1 + arrayHeight;

                arrayColor(i, l);

                char letter = arr[i] + 'a';

                glRasterPos2f(midX, midY);
                glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, letter);
            }
            break;
    }
}

```

```

displayText(-0.98, 0.95, 1, 1, 1, sortName[sortIndex]);

char charSwapCounter[10] = "";
std::sprintf(charSwapCounter, "%d", swapCounter);
displayText(-0.98, 0.88, 1, 1, 1, "Swap counter:");
displayText(-0.69, 0.88, 1, 1, 1, charSwapCounter);

char charArraySize[10] = "";
std::sprintf(charArraySize, "%d", length);
displayText(-0.98, 0.81, 1, 1, 1, "Array size:");
displayText(-0.75, 0.81, 1, 1, 1, charArraySize);

char charDelay[10] = "";
std::sprintf(charDelay, "%d", delay);
displayText(-0.98, 0.74, 1, 1, 1, "Delay (us):");
displayText(-0.72, 0.74, 1, 1, 1, charDelay);
glFlush();

char charTime[10] = "";
std::sprintf(charDelay, "%d", sortTime);
displayText(-0.98, 0.67, 1, 1, 1, "Time (us):");
displayText(-0.72, 0.67, 1, 1, 1, charDelay);
glFlush();
}

void polygon(float leftX, float rightX, float bottomY, float topY) {
    glVertex2f(leftX, bottomY);
    glVertex2f(rightX, bottomY);
    glVertex2f(rightX, topY);
    glVertex2f(leftX, topY);
}

void displayText(float x, float y, float red, float green, float blue, char
*string) {
    glColor3f(red, green, blue);
    glRasterPos2f(x, y);
    char *c;
    for (c = string; *c != '\0'; c++)
        glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, *c);
}

void keyboardEvent(unsigned char key, int x, int y) {
    //printf("Key:%d\n", key); //DEBUG

    //ESC - Exit
    //s - Sort
    //r - Randomize
    //Num 8 - Add delay (x10)
    //Num 2 - Reduce delay (/10)
    //Num 4 - Reduce size (/10)
    //Num 6 - Increase size (x10)

    switch (key) {
        //General purpose
        case 27:
            free(arr);
            exit(0);
        case 's':
            //std::thread sortThread(sort, arr, length);
            //sortThread.join();
            startTime = clock();
            sort(arr, length);
            endTime = clock();
    }
}

```

```

        sortTime = endTime - startTime;
        break;
    case 'r':
        randomizeArray(arr, length);
        break;

    //NUMPAD
    case 56: //UP - NumPad 8
        if (delay < 40000)
            delay *= 2;
        //printf("1"); //DEBUG
        break;
    case 50: //DOWN - NumPad 2
        if (delay > 1)
            delay /= 2;
        break;
    case 52: //LEFT - NumPad 4
        if (length == 10)
            length = 5;
        else if (length > 10 && length != 26)
            length -= 100;
        fillArray();
        break;
    case 54: //RIGHT - NumPad 6
        if (length == 5)
            length = 10;
        else if (length < 1000 && length != 26)
            length += 100;
        fillArray();
        break;
    case ']':
        typeSort++;
        if (typeSort % 2 == 0) length = 10;
        if (typeSort % 2 == 1) length = 26;
        fillArray();
        break;
    case '[':
        typeSort--;
        if (typeSort % 2 == 0) length = 10;
        if (typeSort % 2 == 1) length = 26;
        fillArray();
        break;

    /*
    //WIP
    case '=':
        startTime = clock();
        bTree();
        endTime = clock();
        sortTime = endTime - startTime;
        printf("%d", sortTime);
        break;
    */

    //Sort
    case 'q':
        sort = visualizeQuickSort;
        sortIndex = 0;
        break;
    case 'b':
        sort = visualizeBubbleSort;
        sortIndex = 1;
        break;
    case 'u':

```



```

        sort = visualizeBogoSort;
        sortIndex = 2;
        break;
    case 'i':
        sort = visualizeInsertionSort;
        sortIndex = 3;
        break;
    case 'h':
        sort = visualizeHeapSort;
        sortIndex = 4;
        break;
}
}

//-----QuickSort-----
void visualizeQuickSort(int *a, int length) {
    quickSort(a, 0, length - 1);
    counterMessage("QuickSort");
}

void quickSort(int *a, int low, int high) {
    isItRunning();
    if (low < high) {
        int pi = partition(a, low, high);

        quickSort(a, low, pi - 1);
        quickSort(a, pi + 1, high);
    }
}

int partition(int *a, int low, int high) {
    int lowIndex = low - 1;
    int pivot = a[high];

    for (int i = low; i < high; ++i) {
        if (a[i] <= pivot) {
            ++lowIndex;
            swap(lowIndex, i);
        }
    }

    ++lowIndex;
    swap(lowIndex, high);

    return lowIndex;
}
//^^^^^^^^^^^^^^^^^^^^^QuickSort^^^^^^^^^^^^^^^^^^^^^

//-----Bubble Sort-----
void visualizeBubbleSort(int *a, int length) {
    for (int i = 0; i < length; i++) {
        isItRunning();
        for (int j = i + 1; j < length; j++)
            if (a[j] < a[i])
                swap(i, j);
    }
    counterMessage("Bubble Sort");
}
//^^^^^^^^^^^^^^^^^^^^^Bubble Sort^^^^^^^^^^^^^^^^^^^^^

//-----BogoSort-----

```

```
void visualizeBogoSort(int *a, int length) {
    while (!isSorted(a, length))
        shuffle(a, length);
    counterMessage("BogoSort");
}

void shuffle(int *a, int length) {
    for (int i = 0; i < length; i++)
        swap(a[i], a[rand() % length]);
}

bool isSorted(int *a, int length) {
    isItRunning();
    for (int i = 0; i < length; i++)
        if (a[i] > a[i + 1])
            return false;
    return true;
}

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^BogoSort^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

//-----Insertion Sort-----
void visualizeInsertionSort(int *a, int length) {
    int key, j;
    for (int i = 1; i < length; i++) {
        isItRunning();
        key = a[i];
        j = i - 1;
        while (j >= 0 && a[j] > key) {
            swap(j + 1, j);
            j = j - 1;
        }
        a[j + 1] = key;
    }
    counterMessage("Insertion Sort");
}

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^Insertion Sort^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

//-----Heap Sort-----
void visualizeHeapSort(int *a, int length) {
    for (int i = length / 2 - 1; i >= 0; i--) {
        isItRunning();
        heapify(a, length, i);
    }
    for (int i = length - 1; i > 0; i--) {
        isItRunning();
        swap(0, i);

        heapify(a, i, 0);
    }
    counterMessage("Heap Sort");
}

void heapify(int *a, int length, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < length && a[left] > a[largest])
        largest = left;

    if (right < length && a[right] > a[largest])
```

```

        largest = right;

    if (largest != i) {
        swap(i, largest);
        heapify(a, length, largest);
    }
}

//^^^^^^^^^^^^^^^^^^^^^^Heap Sort^^^^^^^^^^^^^^^^^^^^^^

/*
//Work in progress
//I want to make a graphical representation of B-tree (In future)
//-----B-tree-----
struct BTreeNode {
    int val[MAX + 1], count;
    struct BTreeNode *link[MAX + 1];
};

struct BTreeNode *root;

int bTree() {
    int ch;

    //Random fill
    for (int i=0; i<=AMOUNT; i++)
        insert(rand()%MAX RAND);

    traversal(root);

    printf("\n");
    search(rand()%MAX RAND, &ch, root);
}

// Create a node
struct BTreeNode *createNode(int val, struct BTreeNode *child) {
    struct BTreeNode *newNode;
    newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
    newNode->val[1] = val;
    newNode->count = 1;
    newNode->link[0] = root;
    newNode->link[1] = child;
    return newNode;
}

// Insert node
void insertNode(int val, int pos, struct BTreeNode *node,
               struct BTreeNode *child) {
    int j = node->count;
    while (j > pos) {
        node->val[j + 1] = node->val[j];
        node->link[j + 1] = node->link[j];
        j--;
    }
    node->val[j + 1] = val;
    node->link[j + 1] = child;
    node->count++;
}

// Split node
void splitNode(int val, int *pval, int pos, struct BTreeNode *node,
              struct BTreeNode *child, struct BTreeNode **newNode) {

```

```

int median, j;

if (pos > MIN)
    median = MIN + 1;
else
    median = MIN;

*newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
j = median + 1;
while (j <= MAX) {
    (*newNode)->val[j - median] = node->val[j];
    (*newNode)->link[j - median] = node->link[j];
    j++;
}
node->count = median;
(*newNode)->count = MAX - median;

if (pos <= MIN) {
    insertNode(val, pos, node, child);
} else {
    insertNode(val, pos - median, *newNode, child);
}
*pval = node->val[node->count];
(*newNode)->link[0] = node->link[node->count];
node->count--;
}

// Set value
int setValue(int val, int *pval,
             struct BTreeNode *node, struct BTreeNode **child) {
    int pos;
    if (!node) {
        *pval = val;
        *child = NULL;
        return 1;
    }

    if (val < node->val[1]) {
        pos = 0;
    } else {
        for (pos = node->count;
             (val < node->val[pos] && pos > 1); pos--)
            ;
        if (val == node->val[pos]) {
            //printf("Do not duplicate numbers\n");
            return 0;
        }
    }

    if (setValue(val, pval, node->link[pos], child)) {
        if (node->count < MAX) {
            insertNode(*pval, pos, node, *child);
        } else {
            splitNode(*pval, pval, pos, node, *child, child);
            return 1;
        }
    }
    return 0;
}

// Insert value
void insert(int val) {
    int flag, i;
    struct BTreeNode *child;

```

```

        flag = setValue(val, &i, root, &child);
        if (flag)
            root = createNode(i, child);
    }

// Search node
void search(int val, int *pos, struct BTreeNode *myNode) {
    if (!myNode) {
        return;
    }

    if (val < myNode->val[1]) {
        *pos = 0;
    } else {
        for (*pos = myNode->count;
            (val < myNode->val[*pos] && *pos > 1); (*pos)--);
        ;
        if (val == myNode->val[*pos]) {
            printf("%d found in row %d", val, *pos);
            return;
        }
    }
    search(val, pos, myNode->link[*pos]);

    return;
}

//Traverse then nodes
void traversal(struct BTreeNode *myNode) {
    int i;
    if (myNode) {
        for (i = 0; i < myNode->count; i++) {
            traversal(myNode->link[i]);
            printf("%d ", myNode->val[i + 1]);
        }
        traversal(myNode->link[i]);
    }
}

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^B-tree^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
*/

```