МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА – Российский технологический университет»

Отчет

Практическая работа №8

Дисциплина Структуры и алгоритмы обработки данных

Тема. Определение эффективного алгоритма сортировки

Выполнил студент Дамарад Д.В.

Фамилия И.О.

Группа ИКБО-13-21

Номер группы

Москва 2022

Задание №1

1. Разработать алгоритм и доказать его работоспособность:

1.1 Постановка задачи: разработать алгоритм сортировки простой вставки одномерного целочисленного массива A[n] и реализовать его функцией.

1.2 Модель решения: Суть его заключается в том, что на каждом шаге алгоритма мы берем один из элементов массива, находим позицию для вставки и вставляем. Стоит отметить, что массив из 1-го элемента считается отсортированным.

1.3 Алгоритм:

```
void insertSort(int* arr, int n) {
   for i ← 1 до array.size {
      for j ← i , пока j>0 и arr[j - 1] > arr[j] {
         int tmp ← arr[j - 1];
         arr[j - 1] ← arr[j];
         arr[j] ← tmp;
      }
   }
}
```

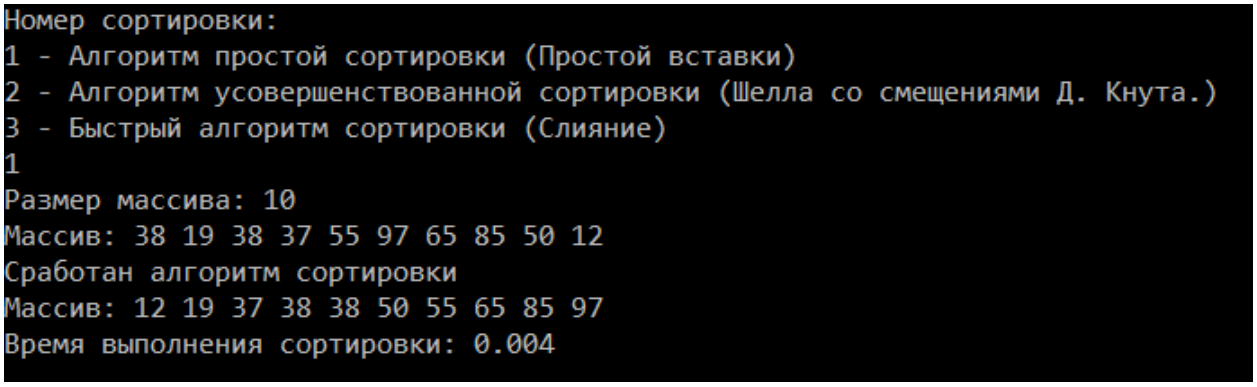
1.4 Определение функции зависимости временной сложности и асимптотической сложности от размера массива: $O(n^2)$

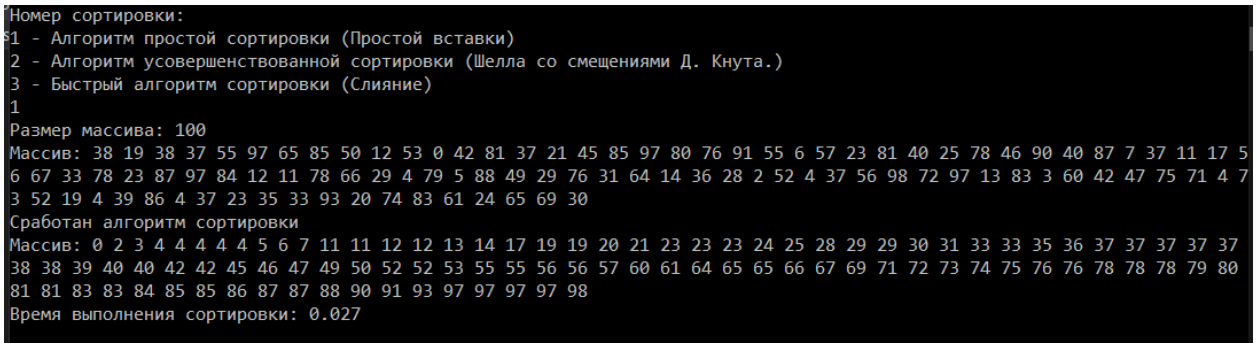
Код:

```
void insertSort(int* arr, int n) {
   for (int i = 1; i < n; i++) {
      for (int j = i; j > 0 && arr[j - 1] > arr[j]; j--) {
         int tmp = arr[j - 1];
         arr[j - 1] = arr[j];
         arr[j] = tmp;
      }
```

}

1.5 Результаты тестирования:





2. Вывод функции роста времени при увеличении размеров массива:

insertSort

Оператор	Количество операторов	
for i ← 1 до array.size	n	C1
for j ← i , пока j>0 и arr[j - 1] > arr[j]	(n+1)*m	C2
int tmp ← arr[j - 1];	n*m	C3
arr[j - 1] ← arr[j];	n*m	C4
arr[j] ← tmp;	n*m	C5

$T(n) = c1 * n + c2 * (n + 1) * m + c3 * n * m + c4 * n * m + c5 * n * m = \boldsymbol{A(n * m) + Bn + C = n^2}$, где $\boldsymbol{n * m = \frac{n^2+n}{2}}$

Наихудший случай = Наилучший случай

3. Результаты прогонов:

n	**T(n) время в сек**	**$T_{эт}$=f(C+M)-функция роста**	**$T_{эп}$=Сф+Мф-количество операций**
100	0.015	10000	2579
1000	0.144	1000000	249882
10000	1.575	100000000	24740187
100000	20.869	10000000000	2473680376
1000000		1000000000000	

3.1 Результаты прогонов при строго убывающем значении:

n	**T(n) время в сек**	**$T_{эт}$=f(C+M)-функция роста**	**$T_{эп}$=Сф+Мф- количество операций**
100	0.016	10000	4901
1000	0.130	1000000	494446
10000	1.629	100000000	49494552
100000	25.856	10000000000	4949957531

1000000		1000000000000	

3.2 Результаты прогонов при строго возрастающем значении:

n	**T(n) время в сек**	**$T_{эт}$=f(C+M)-функция роста**	**$T_{эп}$=Сф+Мф-количество операций**
100	0.013	10000	0
1000	0.127	1000000	0
10000	1.498	100000000	0
100000	12.098	10000000000	0
1000000		1000000000000	

3.3 Исходя из времени и количества операций из таблиц, показывающих рост времени выполнения при наихудшем и наилучшем случае, можно сказать, что алгоритм зависит от входных данных.

4. Порядок роста сложности алгоритма при увеличении размера входных данных – квадратичный.

5. На основе проделанной работы была изучена сортировка простой вставки, а также была проведена оценка ее сложности при разном количестве входных данных в разном порядке.

Задание №2

1. Разработать алгоритм и доказать его работоспособность:

1.2 Постановка задачи: разработать алгоритм усовершенствованной сортировки (Шелла со смещениями Д. Кнута.)

1.3 Модель решения: При сортировке Шелла сначала сравниваются и сортируются между собой значения, стоящие один от другого на некотором расстоянии. После этого процедура повторяется для некоторых меньших значений, а завершается сортировка Шелла упорядочиванием элементов при (то есть обычной сортировкой вставками).

1.4 Алгоритм:

```
void shellSort(int* num, int size)
{
   int increment ← 3;
   while increment > 0
   {
      for i ← 0, i < size
      {
         int j ← i;
         int temp ← num[i];
         while j >= increment and num[j - increment] > temp
         {
            num[j] ← num[j - increment];
            j ← j - increment;
         }
         num[j] ← temp;
      }
      if increment > 1 {
         increment ← increment / 2;
      }
```

```
        else if increment = 1 {
            break;
        }
    }
}
```

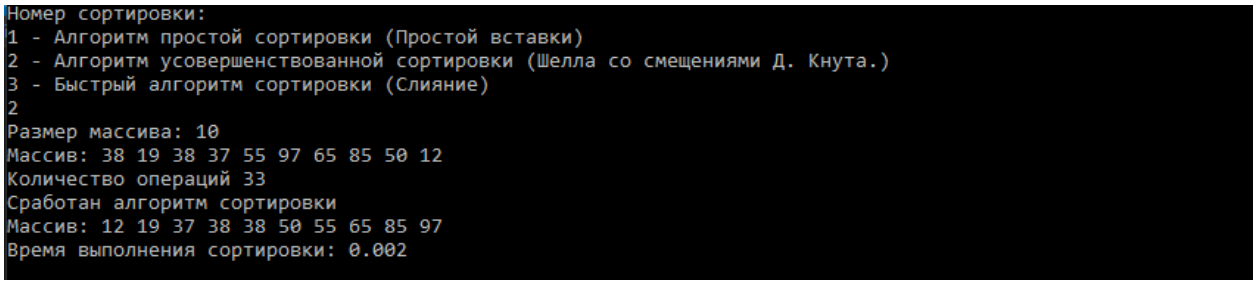
1.5 Определение функции зависимости временной сложности и асимптотической сложности от размера массива: $O(n \log_2 n * \log_2 n)$

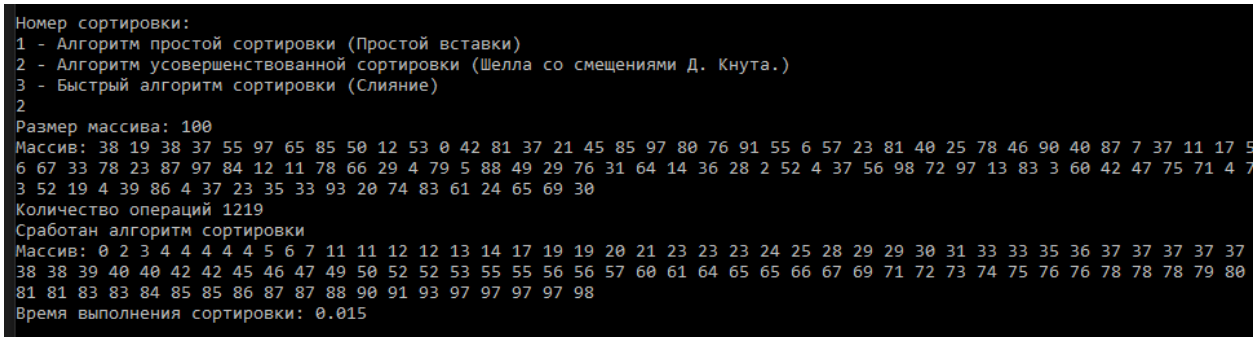
Код:

```
void shellSort(int* num, int size)
{
    int increment = 3;    // начальное приращение сортировки
    while (increment > 0)  // пока существует приращение
    {
        for (int i = 0; i < size; i++)  // для всех элементов массива
        {
            int j = i;          // сохраняем индекс и элемент
            int temp = num[i];
            // просматриваем остальные элементы массива, отстоящие от j-
ого
            // на величину приращения
            while ((j >= increment) && (num[j - increment] > temp))
            {  // пока отстоящий элемент больше текущего
                num[j] = num[j - increment]; // перемещаем его на текущую
позицию
                j = j - increment;       // переходим к следующему отстоящему
элементу
            }
            num[j] = temp; // на выявленное место помещаем сохранённый
элемент
```

```
        }
        if (increment > 1) {      // делим приращение на 2
            increment = increment / 2;
        }
        else if (increment == 1) { // последний проход завершён,
            break;  // выходим из цикла
        }
    }
}
```

1.6 Результаты тестирования:





2 Вывод функции роста времени при увеличении размеров массива:

shellSort

Оператор	Количество операторов	
increment ← 3;	1	C1
while increment > 0	$\log_2 n$	C2
for i ← 0, i < size	$(1 + n)\log_2 n$	C3

int j ← i;	$n * \log_2 n$	C4
int temp ← num[i];	$n * \log_2 n$	C5
while j >= increment and num[j - increment] > temp	$n * \log_2 n * \log_2 n$	C6
num[j] ← num[j - increment];	$n * \log_2 n * \log_2 n$	C7
j ← j - increment	$n * \log_2 n * \log_2 n$	C8
num[j] ← temp;	$n * \log_2 n$	C9
if increment > 1	$n * \log_2 n$	C10
increment←increment/2;	$n * \log_2 n$	C11
else if increment = 1	$n * \log_2 n$	C12
break;	$n * \log_2 n$	C13

$T(n) = \text{c1} * +\text{c2} * \log_2 n + \text{c3} * \log_2 n(n+1) * \text{c4} * n * \log_2 n + \text{c5} * n * \log_2 n + \text{c6} * n * \log_2 n * \log_2 n + \text{c7} * n * \log_2 n * \log_2 n + \text{c8} * n * \log_2 n * \log_2 n + \text{c9} * n * \log_2 n + \text{c10} * n * \log_2 n + \text{c11} * n * \log_2 n + \text{c12} * n * \log_2 n + \text{c13} * n * \log_2 n = \mathbf{An} * (\mathbf{\log_2 n} * \mathbf{\log_2 n}) + \mathbf{Bn} = \mathbf{n} * \mathbf{\log_2 n} * \mathbf{\log_2 n}$

Наихудший случай: $\text{T}(n) = \text{c1} * +\text{c2} * \log_2 n + \text{c3} * \log_2 n(n+1) * \text{c4} * n * \log_2 n + \text{c5} * n * \log_2 n + \text{c6} * n * \log_2 n * \log_2 n + \text{c7} * n * \log_2 n * \log_2 n + \text{c8} * n * \log_2 n * \log_2 n + \text{c9} * n * \log_2 n + \text{c10} * n * \log_2 n + \text{c11} * n * \log_2 n = \mathbf{3n} * \mathbf{\log_2 n} * \mathbf{\log_2 n} + \mathbf{4n} * \mathbf{log_2 n} + \mathbf{3 \log_2 n} + \mathbf{1}$

Наилучший случай: $T(n) = \text{c1} * +\text{c2} * \log_2 n + \text{c3} * \log_2 n(n+1) * \text{c4} * n * \log_2 n + \text{c5} * n * \log_2 n + \text{c9} * n * \log_2 n + \text{c10} * n * \log_2 n + \text{c11} * n * \log_2 n = \mathbf{4n} * \mathbf{log_2 n} + \mathbf{3 \log_2 n} + \mathbf{1}$

3 Результаты прогонов:

n	T(n) время в сек	$T_{эт}$=f(C+M)- функция роста	$T_{эп}$=Сф+Мф- количество операций
100	0.013	4414	1219
1000	0.129	99316	90304
10000	1.333	1765630	544616
100000	14.134	27588000	7069326
1000000			

3.1 Результаты прогонов при строго убывающем значении:

n	T(n) время в сек	$T_{эт}$=f(C+M)- функция роста	$T_{эп}$=Сф+Мф- количество операций
100	0.015	15920	1837
1000	0.150	337844	166834
10000	1.493	5828450	16518191
100000	18.055	89408000	1650185849
1000000			

3.2 Результаты прогонов при строго возрастающем значении:

n	T(n) время в сек	$T_{эт}$=f(C+M)- функция роста	$T_{эп}$=Сф+Мф- количество операций
100	0.016	2678	202

1000	0.149	39894	2002
10000	1.462	531549	20002
100000	12.044	6643910	200002
1000000			

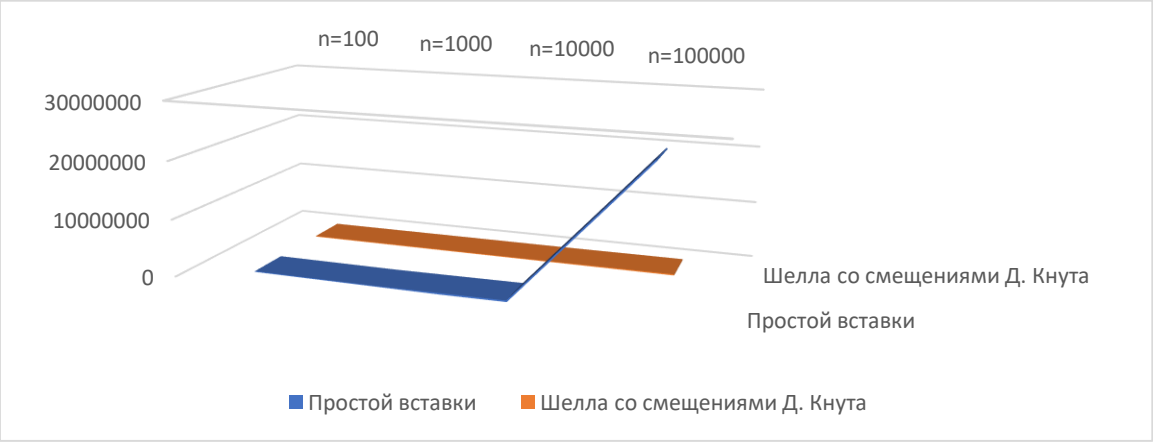
3.3 Исходя из времени и количества операций из таблиц, показывающих рост времени выполнения при наихудшем и наилучшем случае, можно сказать, что алгоритм зависит от входных данных.

4 Порядок роста сложности алгоритма при увеличении размера входных данных – логарифмически-квадратичный.

5 На основе проделанной работы была изучена сортировка Шелла со смещениями Д. Кнута, а также была проведена оценка ее сложности при разном количестве входных данных в разном порядке.

6. По временной и емкостной сложности алгоритм сортировки Шелла со смещениями Д. Кнута превосходит алгоритм сортировки простой вставки.

7. Графики зависимости операций (в сотнях) от числа элементов для сортировки Шелла со смещениями Д. Кнута и сортировки простой вставки:



Задание №3

1. Разработать алгоритм и доказать его работоспособность:

1.2 Постановка задачи: разработать алгоритм сортировки слиянием

1.3 Модель решения: В основе сортировки слиянием лежит принцип «разделяй и властвуй». Список разделяется на равные или практически равные части, каждая из которых сортируется отдельно. После чего уже упорядоченные части сливаются воедино..

1.4 Алгоритм:

```
void Merge(int* A, int first, int last)
{
   int middle, start, final, j;
   int* mas ← new int[last+1];
   middle ← (first + last) / 2;
   start←first;
   final ← middle + 1;
   for j ← first; j <= last;
      if start <= middle and final > last or A[start] < A[final]
      {
         mas[j] ← A[start];
         start++;
      }
      else
      {
         mas[j] ← A[final];
         final++;
      }
   for j ← first; j <= last; {
      A[j] ←mas[j];
   }
   delete[]mas;
```

```
};

void MergeSort(int* A, int first, int last)
{
   if first < last
   {
      MergeSort(A, first, (first + last) / 2);
      MergeSort(A, (first + last) / 2 + 1, last);
      Merge(A, first, last);
   }
}
```

1.5 Определение функции зависимости временной сложности и асимптотической сложности от размера массива: $O(n \log_2 n)$

Код:

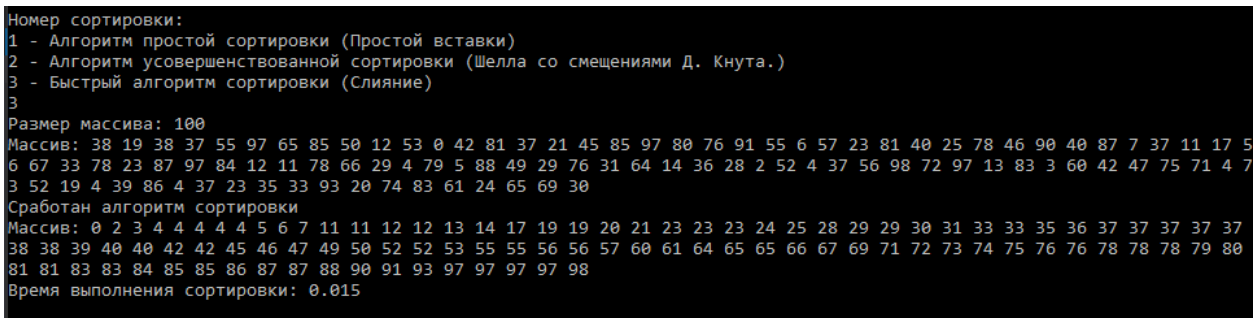
```
void Merge(int* A, int first, int last)
{
   int middle, start, final, j;
   int* mas = new int[last+1];
   middle = (first + last) / 2; //вычисление среднего элемента
   start = first; //начало левой части
   final = middle + 1; //начало правой части
   for (j = first; j <= last; j++) //выполнять от начала до конца
      if ((start <= middle) && ((final > last) || (A[start] < A[final])))
      {
         mas[j] = A[start];
         start++;
      }
      else
      {
```

```
            mas[j] = A[final];
            final++;
        }
    //возвращение результата в список
    for (j = first; j <= last; j++) {
        A[j] = mas[j];
    }
    delete[]mas;
};

//рекурсивная процедура сортировки
void MergeSort(int* A, int first, int last)
{
    long long int counter = 0;
    if (first < last)
    {
        MergeSort(A, first, (first + last) / 2); //сортировка левой части
        MergeSort(A, (first + last) / 2 + 1, last); //сортировка правой части
        Merge(A, first, last); //слияние двух частей
    }
}
```

1.6 Результаты тестирования:

```
Номер сортировки:
1 - Алгоритм простой сортировки (Простой вставки)
2 - Алгоритм усовершенствованной сортировки (Шелла со смещениями Д. Кнута.)
3 - Быстрый алгоритм сортировки (Слияние)
3
Размер массива: 10
Массив: 38 19 38 37 55 97 65 85 50 12
Сработан алгоритм сортировки
Массив: 12 19 37 38 38 50 55 65 85 97
Время выполнения сортировки: 0.002
```



2. Вывод функции роста времени при увеличении размеров массива:

Merge

Оператор	Количество операторов	
int middle, start, final, j;	3	C1
int* mas ← new int[last+1];	1	C2
middle ← (first + last) / 2;	1	C3
start←first;	1	C4
final ← middle + 1;	1	C5
for j ← first; j <= last;	$\log_2 n$	C6
if start <= middle and final > last or A[start] < A[final]	$n * \log_2 n$	C7
mas[j] ← A[start];	$n * \log_2 n$	C8
start++;	$n * \log_2 n$	C9
else	$n * \log_2 n$	C10
mas[j] ← A[final];	$n * \log_2 n$	C11
final++;	$n * \log_2 n$	C12
for j ← first; j <= last;	$\log_2 n$	C13
A[j] ←mas[j];	$\log_2 n$	C14

$T(n) = c1 * 3 + c2 + c3 + c4 + c5 + c6 * \log_2 n + c7 * n * \log_2 n + c8 * n * \log_2 n + c9 * n * \log_2 n + c10 * n * \log_2 n + c11 * n * \log_2 n + c12 * n * \log_2 n + c13 * \log_2 n + c14 * \log_2 n = \mathbf{An} * (\mathbf{\log_2 n}) + \mathbf{B} = \mathbf{n} * \mathbf{\log_2 n}$

Наихудший случай = наилучший случай

MergeSort

Оператор	Количество операторов	
if first < last	1	C1
MergeSort(A, first, (first + last) /2);	$\log_2 n$	C2
MergeSort(A, (first + last) / 2+ 1, last);	$\log_2 n$	C3
Merge(A, first, last);	$\log_2 n$	C4

$T(n) = c1 * 1 + c2 * \log_2 n + c3 * \log_2 n + c4 * \log_2 n = \boldsymbol{A * n \log_2 n + B} = \boldsymbol{n \log_2 n}$

Наихудший случай = наилучший случай

3. Результаты прогонов:

n	**T(n) время в scек**	**$T_{эт}$=f(C+M)- функция роста**	**$T_{эп}$=Сф+Мф- количество операций**
100	0.015	665	1443
1000	0.123	9966	20951
10000	1.492	132878	277231
100000	12.980	1660960	3437855
1000000			

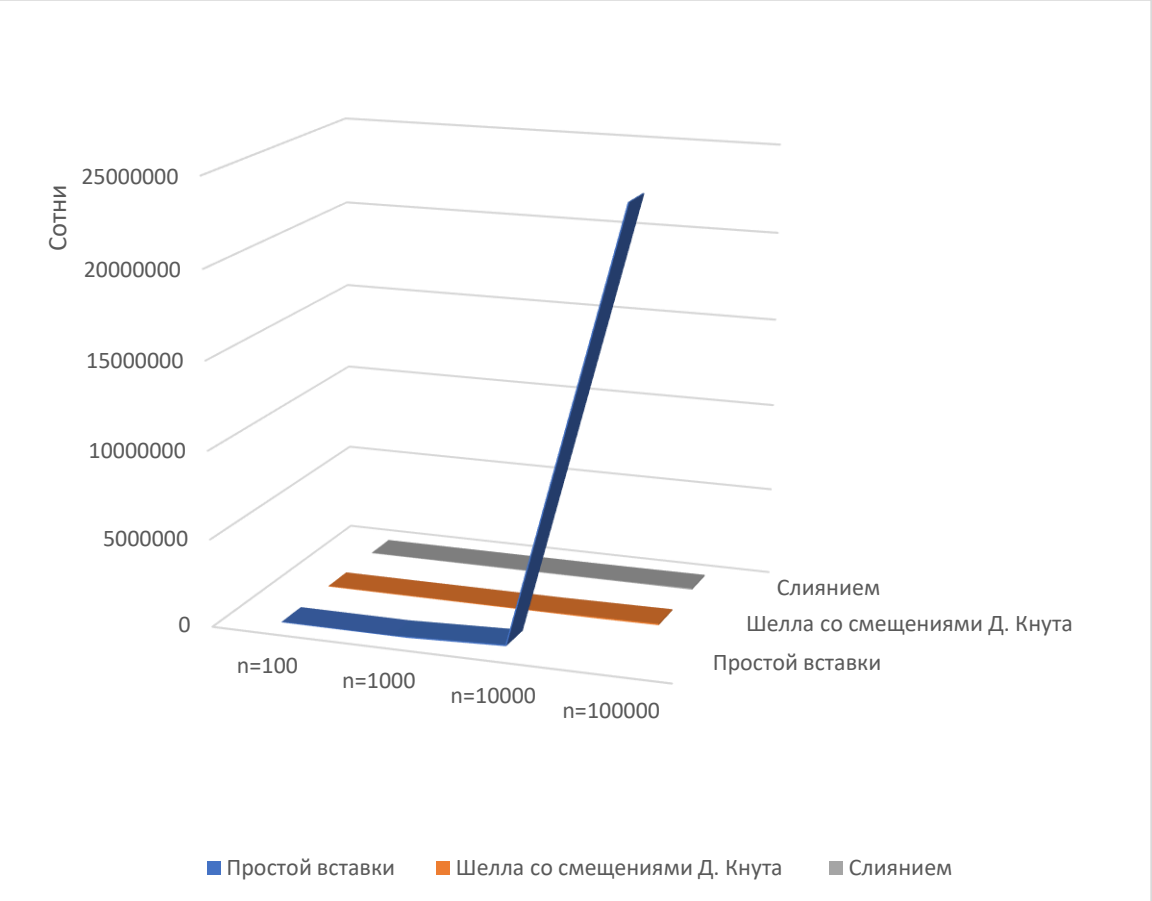
3.1 Результаты прогонов при строго убывающем значении:

n	**T(n) время в мс/сек**	**$T_{эт}$=f(C+M)- функция роста**	**$T_{эп}$=Сф+Мф- количество операций**
100	0.015	1366	1443
1000	0.151	9966	20951
10000	1.484	132878	277231
100000	15.300	1660960	3437855
1000000			

3.2 Результаты прогонов при строго возрастающем значении:

n	T(n) время в мс/сек	$T_{эт}$=f(C+M)- функция роста	$T_{эп}$=Сф+Мф- количество операций
100	0.016	665	1443
1000	0.133	9966	20951
10000	1.247	132878	277231
100000	12.721	1660960	3437855
1000000			

4. График зависимости операций (в сотнях) от количества элементов для сортировки Шелла со смещениями Д. Кнута, сортировки слиянием:



Полный код программы

```
#include <iostream>
```

```
#include <stdlib.h>
#include <ctime>
#include <iomanip>
#include <algorithm>

using namespace std;

void randomFill(int* arr, int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = rand() % 100;
    }
}

void printarr(int* arr, int size) {
    cout << "Массив: ";
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

void sort_po_ubblv(int* arr, int size) {
    for (int i = 0; i < size; i++)
    {
        for (int j = size - 1; j > i; j--)
        {
            if (arr[j] > arr[j - 1])
            {
                swap(arr[j], arr[j - 1]);
            }
        }
    }
}

void insertSort(int* arr, int n) {
    long long int counter = 0;
    for (int i = 1; i < n; i++) {
        for (int j = i; j > 0 && arr[j - 1] > arr[j]; j--) {
            int tmp = arr[j - 1];
            arr[j - 1] = arr[j];
            arr[j] = tmp;
        }
    }
}
```

```
void shellSort(int* num, int size)
{
   long long int counter = 0;
   int increment = 3;    // начальное приращение сортировки
   while (increment > 0)  // пока существует приращение
   {
      for (int i = 0; i < size; i++)  // для всех элементов массива
      {
         int j = i;          // сохраняем индекс и элемент
         int temp = num[i];
         // просматриваем остальные элементы массива, отстоящие от j-ого
         // на величину приращения
         while ((j >= increment) && (num[j - increment] > temp))
         {  // пока отстоящий элемент больше текущего
            num[j] = num[j - increment]; // перемещаем его на текущую позицию
            j = j - increment;       // переходим к следующему отстоящему
элементу
         }
         num[j] = temp; // на выявленное место помещаем сохранённый элемент
      }
      if (increment > 1) {      // делим приращение на 2
         increment = increment / 2;
      }
      else if (increment == 1) { // последний проход завершён,
         break;  // выходим из цикла
      }
   }
}

void Merge(int* A, int first, int last)
{
   int middle, start, final, j;
   int* mas = new int[last+1];
   middle = (first + last) / 2; //вычисление среднего элемента
   start = first; //начало левой части
   final = middle + 1; //начало правой части
   for (j = first; j <= last; j++) //выполнять от начала до конца
      if ((start <= middle) && ((final > last) || (A[start] < A[final])))
      {
         mas[j] = A[start];
         start++;
      }
      else
```

```
        {
            mas[j] = A[final];
            final++;
        }
    //возвращение результата в список
    for (j = first; j <= last; j++) {
        A[j] = mas[j];
    }
    delete[]mas;
};

//рекурсивная процедура сортировки
void MergeSort(int* A, int first, int last)
{
    if (first < last)
    {
        MergeSort(A, first, (first + last) / 2); //сортировка левой части
        MergeSort(A, (first + last) / 2 + 1, last); //сортировка правой части
        Merge(A, first, last); //слияние двух частей
    }
}

int main() {
    srand(time_t(NULL));
    setlocale(LC_ALL, "rus");
    int num; cout << "Номер сортировки: " << endl;
    cout << "1 - Алгоритм простой сортировки (Простой вставки)" << endl;
    cout << "2 - Алгоритм усовершенствованной сортировки (Шелла со
смещениями Д. Кнута.)" << endl;
    cout << "3 - Быстрый алгоритм сортировки (Слияние)" << endl;
    cin >> num;
    switch (num) {
    case 1: {
        int size; cout << "Размер массива: "; cin >> size;
        int* arr = new int[size];
        randomFill(arr, size);
        printarr(arr, size);
        long double start_time = clock();
        insertSort(arr, size);
        cout << "Сработан алгоритм сортировки" << endl;
        printarr(arr, size);
        long double end_time = clock();
        long double search_time = end_time - start_time;
```

```
        cout << "Время выполнения сортировки: " << fixed << setprecision(3) <<
search_time / 1000 << endl;
        delete[]arr;
        break;
    }
    case 2: {
        int size; cout << "Размер массива: "; cin >> size;
        int* arr = new int[size];
        randomFill(arr, size);
        printarr(arr, size);
        long double start_time = clock();
        shellSort(arr, size);
        cout << "Сработан алгоритм сортировки" << endl;
        printarr(arr, size);
        long double end_time = clock();
        long double search_time = end_time - start_time;
        cout << "Время выполнения сортировки: " << fixed << setprecision(3) <<
search_time / 1000 << endl;
        delete[]arr;
        break;
    }
    case 3: {
        int size; cout << "Размер массива: "; cin >> size;
        int* arr = new int[size];
        randomFill(arr, size);
        printarr(arr, size);
        long double start_time = clock();
        MergeSort(arr, 0, size-1);
        cout << "Сработан алгоритм сортировки" << endl;
        printarr(arr, size);
        long double end_time = clock();
        long double search_time = end_time - start_time;
        cout << "Время выполнения сортировки: " << fixed << setprecision(3) <<
search_time / 1000 << endl;
        delete[]arr;
        break;
    }
    default: {
        break;
    }
    }
    return 0;
}
```

Выводы

Наиболее эффективными алгоритмами сортировки являются сортировки Шелла со смещением и слиянием для большого количества данных и небольшого.

Название алгоритма	Асимптотическая сложность алгоритма			
	Наихудший случай	Наилучший случай	Средний случай	Емкостная сложность
Сортировка простой вставки	$O(x^2)$	$\Omega(x^2)$	$\theta(x^2)$	$O(1)$
Сортировка Шелла со смещением Кнута	$O(n\log_2 n)$	$\Omega(n\log_2 n)$	$\theta(n\log_2 n)$	$O(1)$
Сортировка слиянием	$O(n\log_2 n)$	$\Omega(n\log_2 n)$	$\theta(n\log_2 n)$	$O(n)$