

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего
образования
**«Национальный исследовательский Нижегородский государственный
университет им. Н.И. Лобачевского» (ННГУ)**

Институт информационных технологий, математики и механики

Направление подготовки: «Фундаментальная информатика и информационные
технологии»

ОТЧЕТ
по учебной практике

на тему:
«Сравнение алгоритмов сортировки»

Выполнил: студент группы 381606-1
ФИО
Научный руководитель:
ФИО

Нижний Новгород
2016

Содержание

1. Введение	2
2. Постановка задачи	3
2.1. Пузырьковая сортировка	3
2.2. Шейкерная (двунаправленная) сортировка	4
2.3. Сортировка вставками	4
2.4. Сортировка Шелла	5
2.5. Сортировка выбором	6
2.6. Сортировка слиянием	6
3. Структура проекта	7
4. Инструкции пользователю	8
5. Описание эксперимента	9
6. Обсуждение результатов	10
Список литературы	12
Приложение	13

1. Введение

Переразмещение элементов в порядке возрастания или убывания - задача, которая очень часто возникает в программировании. От порядка размещения данных в памяти компьютера зависит не только удобство работы с этими данными, но и скорость выполнения и простота алгоритмов, предназначенных для их обработки.

По оценкам производителей компьютеров в 60-х годах в среднем более четверти машинного времени тратилось на сортировку. Во многих вычислительных системах на нее уходит больше половины машинного времени [1]. Вот некоторые из наиболее распространенных областей применения сортировки:

- 1) Решение задачи группирования, когда нужно собрать вместе все элементы с одинаковыми значениями признака.
- 2) Поиск общих элементов в двух или более массивах.
- 3) Поиск информации по значениям ключей.

При разработке программных продуктов важным этапом становится тестирование, цели которого[2]:

- 1) Продемонстрировать разработчикам и заказчикам, что программа соответствует требованиям
- 2) Выявить ситуации, в которых поведение программы является неправильным, нежелательным или не соответствующим спецификации

2. Постановка задачи

Была поставлена задача провести эксперимент с выявлением лучших качеств алгоритмов сортировки. Алгоритм должен быть эффективным с точки зрения потребления ресурсов процессора и оперативной памяти. Благодаря этому программное обеспечение, требующее отсортированных массивов данных, будет работать быстрее и сможет обрабатывать больший объем информации.

Однако, невозможно выделить самый лучший алгоритм сортировки. Их эффективность и скорость работы сильно зависят от структуры исходных данных. В связи с этим, необходимо провести эксперимент с разными массивами данных, чтобы установить зависимость между структурой информации и скоростью алгоритма сортировки.

Следующие алгоритмы были отобраны для участия в эксперименте как наиболее распространенные:

- 1) Пузырьковая сортировка
- 2) Шейкерная (двунаправленная) сортировка
- 3) Сортировка вставками
- 4) Сортировка Шелла
- 5) Сортировка выбором
- 6) Сортировка слиянием

Ниже будут рассмотрены теоретическая оценка скорости каждого из алгоритмов.

2.1. Пузырьковая сортировка

Попарное сравнение элементов - наиболее очевидное решение проблемы сортировки. Если предположить, что в массиве содержится N элементов и хотя бы один из них занимает свое место в результате однократного просмотра значений, то алгоритм может совершить не более N проходов (Все N понадобятся, если алгоритм изначально отсортирован в обратном порядке). Каждый проход включает в себя N шагов. Отсюда общее время работы - $O(N^2)$. Так как сортировка относится к классу внутренних и не использует дополнительную память, ее затраты составляют $O(1)$. [3]

2.2. Шейкерная (двунаправленная) сортировка

Анализируя метод пузырьковой сортировки, можно отметить два обстоятельства. Во-первых, если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, её можно исключить из рассмотрения. Во-вторых, при движении от конца массива к началу минимальный элемент «всплывает» на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо[4].

Сложность алгоритма имеет порядок $O(N^2)$ для худшего и среднего случая. Но она приближается к $O(N)$ в том случае, если данные уже частично упорядочены. Например, если позиция каждого элемента отличается не более чем на $k \geq 1$ от верной позиции, то алгоритм отработает за $O(kN)$.

Шейкерная сортировка и другие улучшения подробно рассматриваются в книге *Искусство программирования* Дональда Кнута. В частности автор приходит к следующим выводам:

But none of these refinements leads to an algorithm better than straight insertion [that is, insertion sort]; and we already know that straight insertion isn't suitable for large N . [...] In short, the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems.

D. E. Knuth

Но ни одно из этих улучшений не приводит к лучшему алгоритму, чем прямые вставки [да, сортировка вставками], и мы уже знаем, что сортировки вставками не подходят для больших N . [...] В кратце, кажется, что пузырек не за что рекомендовать, за исключением броского названия и факта, что он приводит к некоторым интересным теоретическим проблемам.

Д. Э. Кнут

2.3. Сортировка вставками

Наихудшим случаем является массив, отсортированный в порядке, обратном нужному. При этом каждый новый элемент сравнивается со всеми в отсортированной последовательности. Это означает, что все внутренние циклы состоят из j итераций, то есть $t_j = j$ для всех j . Тогда время работы алгоритма составит[5]:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n j + c_5 \sum_{j=2}^n (j-1) + c_6 \sum_{j=2}^n (j-1) + c_7(n-1)$$

$$T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\frac{n(n-1)}{2} + c_6\frac{n(n-1)}{2} + c_7(n-1) = O(n^2)$$

Время работы является квадратичной функцией от размера входных данных.

Для анализа среднего случая нужно посчитать среднее число сравнений, необходимых для определения положения очередного элемента. При добавлении нового элемента требуется, как минимум, одно сравнение, даже если этот элемент оказался в правильной позиции. i -й добавляемый элемент может занимать одно из $i + 1$ положений. Предполагая случайные входные данные, новый элемент равновероятно может оказаться в любой позиции. Среднее число сравнений для вставки i -го элемента:

$$T_i = \frac{1}{i+1} \left(\sum_{p=1}^i p + i \right) = \frac{1}{i+1} \left(\frac{i(i+1)}{2} + i \right) = \frac{i}{2} + 1 - \frac{1}{i+1}$$

Для оценки среднего времени работы для n элементов нужно просуммировать[5]:

$$T(n) = \sum_{i=1}^{n-1} T_i = \sum_{i=1}^{n-1} \left(\frac{i}{2} + 1 - \frac{1}{i+1} \right) = \sum_{i=1}^{n-1} \frac{i}{2} + \sum_{i=1}^{n-1} 1 - \sum_{i=1}^{n-1} \frac{1}{i+1}$$

$$T(n) \approx \frac{n^2 - n}{4} + (n - 1) - (\ln(n) - 1) = O(n^2)$$

Временная сложность алгоритма — $O(N^2)$. Однако, из-за константных множителей и членов более низкого порядка алгоритм с более высоким порядком роста может выполняться для небольших входных данных быстрее, чем алгоритм с более низким порядком роста.

2.4. Сортировка Шелла

Для алгоритма сортировки, который каждый раз перемещает запись только на одну позицию, среднее время выполнения будет в лучшем случае пропорционально N^2 , потому что в процессе сортировки каждый элемент должен пройти в среднем через $\frac{1}{3}N$ позиций. Поэтому желательно получить метод, существенно превосходящий по скорости метод простых вставок с помощью механизма, позволяющего элементам перемещаться большими скачками, а не короткими шажками[1].

В 1959 году Шелл предложил такой метод. Анализ этого алгоритма — сложная математическая задача, у которой до сих пор нет полного решения[6]. В настоящий момент неизвестно, какая последовательность расстояний даёт наилучший результат, но известно, что расстояния не должны быть кратными друг другу.

Кнут предлагает в качестве последовательно уменьшающихся расстояний использо-

вать одну из следующих последовательностей (приведены в обратном порядке): $1, 4, 13, 40, \dots$, где $h_{i-1} = 3 * h_i + 1$ или $1, 3, 7, 15, 31, \dots$, где $h_{i-1} = 2 * h_i + 1$. В последнем случае математическое исследование показывает, что при сортировке N элементов алгоритмом Шелла затраты пропорциональны $N^{1,2}$.

2.5. Сортировка выбором

Анализ сортировки выбором не сложен, так как сложность не зависит от данных. Поиск минимального элемента требует просмотра N элементов (это $N - 1$ сравнений). Поиск следующего минимального требует просмотра $N - 1$ элементов и так далее. Таким образом, сложность алгоритма можно оценить с помощью следующего выражения:

$$(N - 1) + (N - 2) + \dots + 2 + 1 = \frac{N(N - 1)}{2} = O(N^2)$$

2.6. Сортировка слиянием

При сортировке N элементов сортировка слиянием имеет в худшем и среднем случае сложность $O(N \log N)$. Формула зависимости времени выполнения алгоритма от количества данных следует из его определения[7]:

$$T(N) = 2T\left(\frac{N}{2}\right) + N$$

При этом сортировка слиянием относится к классу внешних и требует дополнительной памяти. Оптимальная реализация требует однократного выделения памяти, затраты которой равны $O(N)$.

3. Структура проекта

Для решения проблемы был разработан проект. Он представляет собой компьютерную программу, написанную на языке высокого уровня C. Программа последовательно применяет различные алгоритмы сортировки к одному и тому же массиву данных и замеряет время работы каждого в секундах.

Проект имеет следующую структуру:

- **report** - каталог, содержащий настоящий отчет в формате L^AT_EX
- **sample** - каталог, содержащий тестовые данные
- **src** - каталог, содержащий исходный код программы
 - **algorithms.c** - библиотека, содержащая алгоритмы сортировки. Её код приводится в Приложении
 - **algorithms.h** - заголовочный файл библиотеки алгоритмов
 - **main.c** - код, отвечающий за интерфейс и основную логику работы программы
 - **utils.c** - некоторые второстепенные функции
 - **utils.h** - заголовочный файл

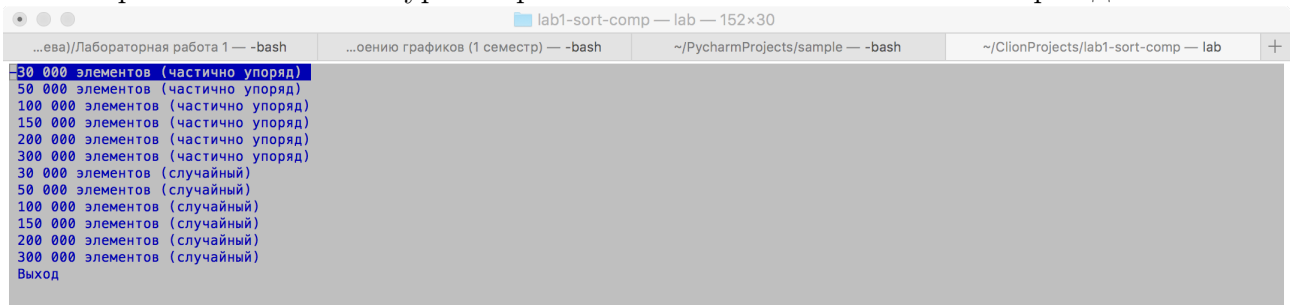
Для сборки проекта в системах macOS и Linux необходимо установить библиотеку ncurses и компилятор gcc и выполнить в корне проекта следующую команду:

```
gcc -o lab src/algorithms.c src/utils.c src/main.c -lncurses -lmenu
```

Полный код проекта доступен по адресу <https://github.com/alexbat98/lab1-sort-comp>

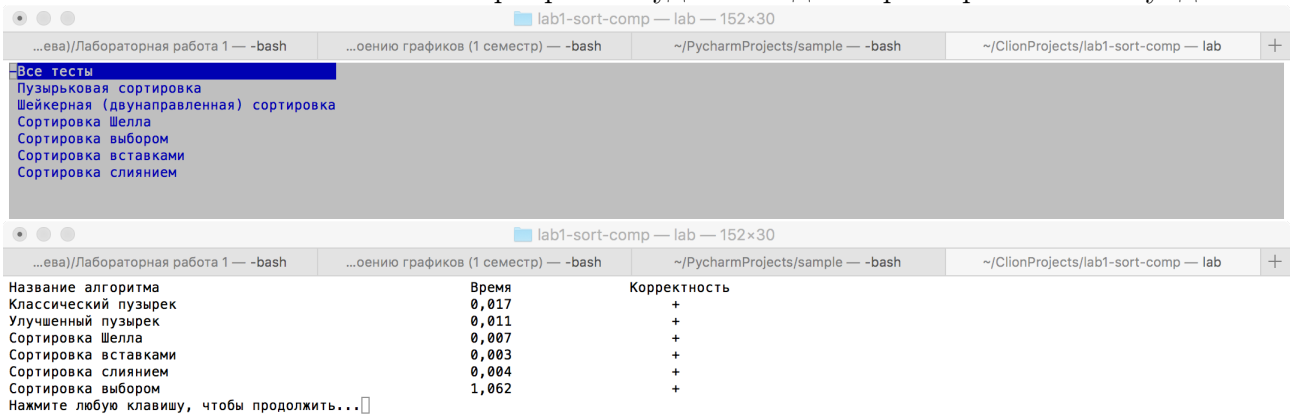
4. Инструкции пользователю

Программа имеет интуитивный псевдографический интерфейс. На первом этапе предлагается стрелками на клавиатуре выбрать 1 из нескольких тестовых наборов данных.



```
lab1-sort-comp — lab — 152×30
...ева)/Лабораторная работа 1 — -bash  ...оению графиков (1 семестр) — -bash  ~/PycharmProjects/sample — -bash  ~/ClionProjects/lab1-sort-comp — lab  +
-30 000 элементов (частично упоряд)
50 000 элементов (частично упоряд)
100 000 элементов (частично упоряд)
150 000 элементов (частично упоряд)
200 000 элементов (частично упоряд)
300 000 элементов (частично упоряд)
30 000 элементов (случайный)
50 000 элементов (случайный)
100 000 элементов (случайный)
150 000 элементов (случайный)
200 000 элементов (случайный)
300 000 элементов (случайный)
Выход
```

Затем необходимо выбрать режим работы. Алгоритмы можно запускать как по одному, так и все вместе. После окончания сортировки будет выведено время работы в секундах.



```
lab1-sort-comp — lab — 152×30
...ева)/Лабораторная работа 1 — -bash  ...оению графиков (1 семестр) — -bash  ~/PycharmProjects/sample — -bash  ~/ClionProjects/lab1-sort-comp — lab  +
-Все тесты
Пузырьковая сортировка
Шейкерная (двунаправленная) сортировка
Сортировка Шелла
Сортировка выбором
Сортировка вставками
Сортировка слиянием

lab1-sort-comp — lab — 152×30
...ева)/Лабораторная работа 1 — -bash  ...оению графиков (1 семестр) — -bash  ~/PycharmProjects/sample — -bash  ~/ClionProjects/lab1-sort-comp — lab  +
Название алгоритма      Время      Корректность
Классический пузырьек    0,017      +
Улучшенный пузырьек      0,011      +
Сортировка Шелла         0,007      +
Сортировка вставками     0,003      +
Сортировка слиянием      0,004      +
Сортировка выбором       1,062      +
Нажмите любую клавишу, чтобы продолжить...
```

Название алгоритма	Время	Корректность
Классический пузырьек	0,017	+
Улучшенный пузырьек	0,011	+
Сортировка Шелла	0,007	+
Сортировка вставками	0,003	+
Сортировка слиянием	0,004	+
Сортировка выбором	1,062	+

5. Описание эксперимента

В программе существует 2 вида наборов данных: частично упорядоченный и абсолютно случайный. Каждый набор содержит не менее 30 000 элементов. Тестирование на меньшем объеме данных не имеет смысла ввиду высокой вычислительной мощности современных компьютеров. Для чистоты эксперимента все тесты проводились на одном и том же компьютере с процессором Intel Core i5 2,6ГГц. Все остальные приложения были закрыты, чтобы не мешать работе тестового.

Частично упорядоченные наборы имеют следующую структуру: первые 100 элементов содержат случайные числа от 1 до 100, следующие 100 от 101 до 200 и так далее. Случайные наборы просто содержат случайные числа из диапазона от 0 до N.

Современные компьютерные системы стараются генерировать случайные числа с нормальным распределением. За счет этого мы сможем добиться наиболее честных условий для работы разных алгоритмов.

6. Обсуждение результатов

В ходе исследования был проведен эксперимент, с помощью которого были выявлены особенности работы алгоритмов сортировки. Результаты представлены в таблицах ниже.

Таблица 1. Частично упорядоченные данные

Алгоритм	30k	50k	100k	150k	200k	300k
Пузырьковая сортировка	0,017	0,027	0,054	0,077	0,099	0,161
Шейкерная сортировка	0,011	0,018	0,038	0,055	0,077	0,122
Сортировка Шелла	0,007	0,012	0,029	0,039	0,052	0,083
Сортировка вставками	0,003	0,006	0,008	0,013	0,015	0,025
Сортировка слиянием	0,004	0,007	0,014	0,022	0,029	0,048
Сортировка выбором	1,064	2,986	11,876	26,528	48,269	107,033

Таблица 2. Случайные данные

Алгоритм	30k	50k	100k	150k	200k	300k
Пузырьковая сортировка	3,154	8,893	35,567	80,754	142,139	319,785
Шейкерная сортировка	2,408	6,749	27,170	62,281	108,503	244,321
Сортировка Шелла	2,132	5,085	23,144	48,592	94,249	209,012
Сортировка вставками	0,632	1,740	6,981	15,498	27,328	62,970
Сортировка слиянием	0,005	0,009	0,018	0,028	0,039	0,062
Сортировка выбором	1,062	2,945	11,754	26,655	47,259	106,825

Как мы можем видеть из представленных данных, на частично упорядоченном массиве данных хорошо показала себя сортировка вставками. Она не требует дополнительной памяти, отрабатывает за минимальное время. Сортировка Шелла хоть и является её улучшением, показала более скромные результаты. Но этот алгоритм недостаточно изучен и допускает правку некоторых параметров, которые могут повлиять на его производительность. Не самое плохое время работы продемонстрировали пузырьковые сортировки, несмотря на то, что их сложность равна $O(N^2)$. А вот сортировка выбором - явный аутсайдер. Она требует больше всего времени.

Но все меняется при случайных данных. Сортировка слиянием показывает отличный результат. Сортировка выбором так же продемонстрировала прежнее время работы. Отсюда можем сделать вывод, что структура данных мало влияет на скорость работы этих двух сортировок. А вот другие сортировки сильно деградировали на неупорядоченном массиве данных. В особенности пузырьковые сортировки.

Таким образом, в большинстве случаев стоит применять сортировку слиянием. Многие языки высокого уровня используют её модификации в качестве алгоритмов сортировки по умолчанию. Но у скорости есть своя цена. Сортировка слиянием требует дополнительной

памяти. Если этот ресурс ограничен, стоит рассмотреть сортировку вставками. На случайных данных она сильно уступает сортировке слиянием по времени, но гораздо более экономично расходует память.

Список литературы

1. *Кнут Д. Э.* Искусство программирования. М. : И.Д. Вильямс, 2012.
2. *Wikipedia* Software testing. URL: https://en.wikipedia.org/wiki/Software_testing.
3. *Стивенс Р.* Алгоритмы. Теория и практическое применение.
Москва : Издательство «Э», 2016.
4. *Wikipedia* Cocktail shaker sort.
URL: https://en.wikipedia.org/wiki/Cocktail_shaker_sort.
5. *Wikipedia* Insertion sort. URL: https://en.wikipedia.org/wiki/Insertion_sort.
6. *Горьков А.* Ещё раз про сортировку. URL: <https://habrahabr.ru/post/104697/>.
7. *Wikipedia* Merge sort. URL: https://en.wikipedia.org/wiki/Merge_sort.

Приложение

```
1 #include <stdlib.h>
2 #include <time.h>
3 #include <stdio.h>
4 #include <string.h>
5 #include "algorithms.h"
6 /**
7  * Классическая пузырьковая сортировка
8  * @param arr Массив значений для сортировки
9  * @param n Длина массива
10  */
11 void bubble_sort(int *arr, int n)
12 {
13     int i, l, hasChanged;
14     l = n - 1;
15     do
16     {
17         hasChanged = 0;
18         for (i = 0; i < l; ++i)
19         {
20             if (arr[i] > arr[i + 1])
21             {
22                 arr[i] ^= arr[i + 1];
23                 arr[i + 1] ^= arr[i];
24                 arr[i] ^= arr[i + 1];
25                 hasChanged = 1;
26             }
27         }
28         l--;
29     } while (hasChanged);
30 }
31 /**
32  * Улучшенная сортировка пузырьком с проходом в обе стороны
33  * @param arr Массив данных
```

```

34  * @param n Количество элементов
35  */
36  void better_bubble_sort(int *arr, int n)
37  {
38      int i, start, finish, hasChanged;
39      start = 0;
40      finish = n - 1;
41      do
42      {
43          hasChanged = 0;
44          for (i = start; i < finish; ++i)
45          {
46              if (arr[i] > arr[i + 1])
47              {
48                  arr[i] ^= arr[i + 1];
49                  arr[i + 1] ^= arr[i];
50                  arr[i] ^= arr[i + 1];
51                  hasChanged = 1;
52              }
53          }
54          --finish;
55          if (hasChanged)
56          {
57              for (i = finish - 1; i >= start; --i)
58              {
59                  if (arr[i] > arr[i + 1])
60                  {
61                      arr[i] ^= arr[i + 1];
62                      arr[i + 1] ^= arr[i];
63                      arr[i] ^= arr[i + 1];
64                      hasChanged = 1;
65                  }
66              }
67              start++;
68          }
69      }

```

```

70     while (hasChanged);
71 }
72 /**
73  * Сортировка Шелла
74  * @param arr Массив данных
75  * @param n Количество элементов
76  */
77 void shell_sort(int *arr, int n)
78 {
79     int i, hasChanged;
80     int d = n;
81     do
82     {
83         d = (d + 1) / 2;
84         hasChanged = 0;
85         for (i = 0; i < n - d; ++i)
86         {
87             if (arr[i] > arr[i + d])
88             {
89                 arr[i] ^= arr[i+d];
90                 arr[i + d] ^= arr[i];
91                 arr[i] ^= arr[i + d];
92                 hasChanged = 1;
93             }
94         }
95     }
96     while (d != 1 || hasChanged);
97 }
98 /**
99  * Поиск минимального элемента в массиве
100  * @param arr Массив
101  * @param n Количество элементов
102  * @return Индекс минимального
103  */
104 int min(int *arr, int n)
105 {

```



```

106     int i, min_idx = 0;
107
108     for (i = 0; i < n; ++i)
109     {
110         if (arr[i] < arr[min_idx])
111         {
112             min_idx = i;
113         }
114     }
115     return min_idx;
116 }
117 /**
118  * Сортировка выбором
119  * @param arr Массив данных
120  * @param n Количество элементов
121  */
122 void selection_sort(int *arr, int n)
123 {
124     int i, j, pos;
125     for (i = 0; i < n - 1; ++i)
126     {
127         pos = i;
128         for (j = i + 1; j < n; ++j)
129         {
130             if (arr[pos] > arr[j])
131             {
132                 pos = j;
133             }
134         }
135         if (pos != i)
136         {
137             arr[i] ^= arr[pos];
138             arr[pos] ^= arr[i];
139             arr[i] ^= arr[pos];
140         }
141     }

```

```

142 }
143 /**
144  * Сортировка вставками
145  * @param arr Массив
146  * @param n Количество элементов
147  */
148 void insertion_sort(int *arr, int n)
149 {
150     int i, j, b;
151     for (i = 0; i < n - 1; ++i)
152     {
153         b = arr[i + 1];
154         j = i;
155         while ((j >= 0) && (b < arr[j]))
156         {
157             arr[j + 1] = arr[j];
158             j--;
159         }
160         arr[j + 1] = b;
161     }
162 }
163 /**
164  * Алгоритм слияния двух упорядоченных массивов
165  * @param first Первая часть
166  * @param nf Размер
167  * @param second Вторая часть
168  * @param ns Размер
169  * @param result Результат
170  * @param k Количество
171  */
172 void merge ( int * first , int nf, int * second , int ns, int * result , int
            k )
173 {
174     int count = 0, i = 0, j = 0;
175     first[nf] = INT_MAX;
176     second[ns] = INT_MAX;

```

```

177     while (count < nf + ns)
178     {
179         if (first[i] < second[j])
180         {
181             result[k + count] = first[i++];
182             count++;
183         } else
184         {
185             result[k + count] = second[j++];
186             count++;
187         }
188     }
189 }
190 /**
191  * Сортировка слиянием
192  * @param arr Исходный массив
193  * @param n Количество элементов
194  */
195 void merge_sort ( int * arr , int n )
196 {
197     int i;
198     int h = 1;
199     int begin;
200     int nf, ns;
201     int *first , *second;
202     first = (int *) calloc (n, sizeof(int));
203     second = (int *) calloc (n / 2 + 1, sizeof(int));
204     while (h < n)
205     {
206         begin = 0;
207         while (begin < n - 1)
208         {
209             nf = 0;
210             for (i = 0; (i < h) && (begin + i < n); i++)
211             {
212                 first[i] = arr[begin + i];

```

```

213         nf++;
214     }
215     ns = 0;
216     for (i = 0; (i < h) && (begin + h + i < n); i++)
217     {
218         second[i] = arr[begin + h + i];
219         ns++;
220     }
221     merge(first , nf, second , ns, arr , begin);
222     begin += 2 * h;
223 }
224 h *= 2;
225 }
226 }

```