МИНОБР НАУКИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА — Российский технологический университет»

РТУ МИРЭА

Отчёт по выполнению практического задания 3
Тема: Определение эффективного алгоритма сортировки
Дисциплина Структуры и алгоритма обработки данных

Выполнил студент Пак С.А.

группа ИКБО-05-20

Москва 2021

СОДЕРЖАНИЕ

Номер варианта: 1

ОТЧЁТ ПО ЗАДАНИЮ 1

1. Задача 1

1. Постановка задачи

Разработать алгоритм простой сортировки, определённой в варианте (алгоритм сортировки пузырьком), реализовать алгоритм.

2. Описание подхода к решению

Суть алгоритма сортировки пузырьком состоит в том, что наибольшие элементы перемещаются в конец массива. Следовательно, можно немного оптимизировать сортировку.

Чтобы оптимизировать сортировку, нужно после каждого перемещения наибольшего элемента в конец массива уменьшать количество итераций внутреннего цикла на единицу.

3. Алгоритм

Алгоритм состоит из повторяющихся проходов по массиву. За каждый проход элементы последовательно попарно сравниваются и, если порядок в паре неверный, выполняется перестановка элементов.

Таким образом, за первый проход наибольший элемент оказывается в конце массива. Поэтому при дальнейших проходах не нужно проверять самые последние элементы.

В коде, приведённом ниже, N — количество элементов в массиве,а a — исходный массив.

Код функции сортировки на псевдокоде:

```
for i ← 0 to N – 1 do
   for j ← 0 to N – i – 1 do
      if (a[j] > a[j + 1]) then
         temp = a[j];
         a[j] = a[j + 1];
         a[j + 1] = temp;
      endif
   od
od
```

4. Определение временной сложности

Для определения временной сложности алгоритма нужно составить таблицу (табл.1):

Таблица 1

Операторы	Время выполнения инструкции	Количество выполнений оператора
`for i ← 0 to N – 1 do`	C_1	N

Продолжение таблицы 1

Операторы	Время выполнения инструкции	Количество выполнений оператора
`for j ← 0 to  N – i – 1 do`	C_2	$\sum_{i=1}^{N-1} t_i$
`if (a[j] > a[j + 1]) then`	C_3	$\sum_{i=1}^{N-1} t_i$
`temp = a[j];`	C_4	$\sum_{i=1}^{N-1} t_i$
`a[j] = a[j + 1];`	C_5	$\sum_{i=1}^{N-1} t_i$
`a[j + 1] = temp;`	C_6	$\sum_{i=1}^{N-1} t_i$
`endif`		
`od`		
`od`		

Таким образом, теоретическая сложность алгортима будет вычисляться по следующей формуле (формула 1):

$$T(n)=C_1\cdot N+C_2\cdot\sum_{i=1}^{N-1} t_i+C_3\cdot\sum_{i=1}^{N-1} t_i+C_4\cdot\sum_{i=1}^{N-1} t_i+C_5\cdot\sum_{i=1}^{N-1} t_i+C_6\cdot\sum_{i=1}^{N-1} t_i= \\ =C_1\cdot N+(C_2+C_3+C_4+C_5+C_6)\sum_{i=1}^{N-1} t_i \quad (1)$$

<u>В худшем случае</u> сумма $\sum_{i=1}^{N-1} t_i=\frac{(1+N-1)\cdot(N-1)}{2}=\frac{N(N-1)}{2}=\frac{N^2-N}{2}$.

Поэтому формула 1 принимает следующий вид (формула 2):

$$T(n)=C_1\cdot N+(C_2+C_3+C_4+C_5+C_6)\frac{N^2-N}{2} \quad (2)$$

<u>В лучшем случае</u> сумма $\sum_{i=1}^{N-1} t_i=\frac{(1+N-1)\cdot(N-1)}{2}=\frac{N(N-1)}{2}=\frac{N^2-N}{2}$.

Следовательно, формула 1 принимает следующий вид (формула 3):

$$T(n)=C_1\cdot N+(C_2+C_3+C_4+C_5+C_6)\frac{N^2-N}{2} \quad (3)$$

То есть, и в худшем, и в лучшем случаях теоретическая сложность одинакова.

Следовательно, сложность в нотации *O* будет иметь следующий вид (формула 4):

$$T(n)=O(C_1 \cdot N+(C_2+C_3+C_4+C_5+C_6)\frac{N^2-N}{2})=O(N^2) \quad (4)$$

5. Код функции сортировки

Код функции реализован на языке программирования TypeScript, который является неким надмножеством языка JavaScript.

```
/**
 * Выполняет сортировку пузырьком для заданного массива array
 * @param array         сортируемый массив
 */
export const bubbleSort = (array: Array<number>): void => {
  const N: number = array.length;

  let comps: number = 0;
  let trans: number = 0;

  for (let i: number = 0; i < N - 1; ++i) {
    for (let j: number = 0; j < N - i - 1; ++j) {
      ++comps;

      if (array[j] > array[j + 1]) {
        ++trans;
        swap(array, j, j + 1);
      }
    }
  }

  console.log(`Сравнений: ${comps}`);
  console.log(`Перемещений: ${trans}`);
}
```

6. Тесты

Ниже представлена сводная таблица результатов для алгоритма сортировки пузырьком (табл.2):

Таблица 2

n	**T**	**f(C+M)**	**$C_ф$+$M_ф$**
100	0,63 с	10000	7210
1000	0,64 с	1000000	745731
10000	0,98 с	100000000	74951916
100000	35,32 с	10000000000	7492650940

2. Задача 2

1. Постановка задачи

Разработать алгоритм ускоренной сортировки, определённой в варианте (алгоритм шейкерной сортировки), реализовать алгоритм.

2. Описание подхода к решению

Алгоритм шейкерной сортировки является некой модификацией алгоритма сортировки пузырьком, причём того же варианта, что показан в задаче 1.

Суть состоит в двойных проходах по массиву. Первый раз слева направо, а во второй раз справа налево. Таким образом за каждый проход по массиву проверяемая граница уменьшается на две единицы. Такие проходы осуществляются до тех пор, пока левая граница строго меньше правой.

3. Алгоритм

1) Создаются две переменные: одна нужна для обозначения левой границы массива; другая — для правой;

2) Пока левая граница строго меньше правой границы нужно выполнить следующие действия:

2.1) Пройтись по всем элементам массива слева направа, рассматривая каждую пару элементов. Если порядок элементов в паре неверен, то осуществить перестановку;

2.2) Пройтись по всем элементам массива справа налево, рассматривая каждую пару элементов. Если порядок элементов в паре неверен, то осуществить перестановку;

3) Увеличить левую границу на единицу, а правую границу уменьшить на единицу.

В коде, приведённом ниже, N — количество элементов в массиве,а a — исходный массив.

Код функции сортировки на псевдокоде:

```
left = 1;
right = N – 1;

do
   for i ← left to right do
      if (a[i – 1] > a[i]) then
         temp = a[i];
         a[i] = a[i – 1];
         a[i – 1] = temp;
      endif
   od

   for j ← right downto left do
      if (a[i – 1] > a[i]) then
```

```
            temp = a[i];
            a[i] = a[i – 1];
            a[i – 1] = temp;
         endif
      od

      left = left + 1;
      right = right – 1;

while (left < right);
```

4. Определение временной сложности

Для определения временной сложности алгоритма нужно составить таблицу (табл.3):

Таблица 3

Операторы	Время выполнения инструкции	Количество выполнений оператора
`left = 1;`	C_1	1
`right = N – 1;`	C_2	1
`do`	C_3	$\frac{N}{2}$
`for i ← left to right do`	C_4	$\sum_{i=1}^{N-1} t_i$
`if (a[i – 1] > a[i]) then`	C_5	$\sum_{i=1}^{N-1} t_i$
`temp = a[i];`	C_6	$\sum_{i=1}^{N-1} t_i$
`a[i] = a[i – 1];`	C_7	$\sum_{i=1}^{N-1} t_i$
`a[i – 1] = temp;`	C_8	$\sum_{i=1}^{N-1} t_i$
`endif`		
`od`		
`for j ← right downto left do`	C_9	$\sum_{i=1}^{N-1} t_i$
`if (a[i – 1] > a[i]) then`	C_{10}	$\sum_{i=1}^{N-1} t_i$
`temp = a[i];`	C_{11}	$\sum_{i=1}^{N-1} t_i$
`a[i – 1] = a[i];`	C_{12}	$\sum_{i=1}^{N-1} t_i$
`a[i] = temp;`	C_{13}	$\sum_{i=1}^{N-1} t_i$

Продолжение таблицы 3

Операторы	Время выполнения инструкции	Количество выполнений оператора
`endif`		
`od`		
`left = left + 1`	C_{14}	$\frac{N}{2}$
`right = right – 1;`	C_{15}	$\frac{N}{2}$
`while (left < right)`		

Таким образом, теоретическая сложность алгортима будет вычисляться по следующей формуле (формула 5):

$$T(n)=2(C_1+C_2)+(C_3+C_{14}+C_{15})\frac{N}{2}+ \\ +(C_4+C_5+C_6+C_7+C_8+C_9+C_{10}+C_{11}+C_{12}+C_{13})\sum_{i=1}^{N-1} t_i \quad (5)$$

Пусть $A=C_1+C_2, B=C_3+C_{14}+C_{15}, C=C_4+C_5+C_6+C_7+...+C_{13}$ какие-то константы. Тогда формула 5 запишется в следующем виде (формула 6):

$$T(n)=2A+B\frac{N}{2}+C\sum_{i=1}^{N-1} t_i \quad (6)$$

<u>В худшем случае</u> сумма $\sum_{i=1}^{N-1} t_i=\frac{N^2-N}{4}$. Поэтому теоретическая сложность преобразуется так (формула 7):

$$T(n)=2A+B\frac{N}{2}+C\frac{N^2-N}{2} \quad (7)$$

<u>В лучшем случае</u> сумма $\sum_{i=1}^{N-1} t_i=\frac{N^2-N}{4}$. Поэтому теоретическая сложность преобразуется так (формула 8):

$$T(n)=2A+B\frac{N}{2}+C\frac{N^2-N}{2} \quad (8)$$

То есть, и в худшем, и в лучшем случаях теоретическая сложность одинакова.

Следовательно, сложность в нотации *O* будет иметь следующий вид (формула 9):

$$T(n)=O(2A+B\frac{N}{2}+C\frac{N^2-N}{2})=O(N^2) \quad (9)$$

5. Код функции сортировки

```
/**
 * Выполняет шейкерную сортировку для заданного массива array
 * @param array         сортируемый массив
 */
export const cocktailSort = (array: Array<number>): void => {
  const N: number = array.length;
  let left: number = 1;
  let right: number = N - 1;
  let comps: number = 0;
  let trans: number = 0;
  do {
    // сначала проход слева направо
    for (let i: number = left; i <= right; ++i) {
      ++comps;
      if (array[i - 1] > array[i]) {
        ++trans;
        swap(array, i - 1, i);
      }
    }
    --right;          // уменьшение правой границы массива
    // потом проход справа налево
    for (let i: number = right; i >= left; --i) {
      ++comps;
      if (array[i - 1] > array[i]) {
        ++trans;
        swap(array, i - 1, i);
      }
    }
    ++left;           // увеличение левой границы массива
  }
  while (left < right);
  console.log(`Сравнений: ${comps}`);
  console.log(`Перемещений: ${trans}`);
}
```

6. Тесты

Ниже представлена сводная таблица результатов для алгоритма шейкерной сортировки (табл.4):

Таблица 4

n	**T**	**f(C+M)**	**$C_ф$+$M_ф$**
100	0,63 с	10000	6503

Продолжение таблицы 4

n	**T**	**f(C+M)**	**$C_ф$+$M_ф$**
1000	0,64 с	1000000	619883
10000	1,06 с	100000000	62344874
100000	29,81 с	10000000000	6266314028

3. Анализ результатов по таблицам 2 и 4

Из таблицы видно, что по времени работы два алгоритма (сортировка пузырьком и шейкерная сортировка) не сильно отличаются до *n = 10000*. После время сортировки массива из 100000 элементов с помощью алгоритма простого обмена стало в приблизительно 1,2 раза больше, чем время сортировки массива с помощью шейкерной сортировки.

К тому же видно, что суммарное количество сравнений и перемещений у алгоритма простого обмена намного больше, чем у шейкерной сортировки, что также влияет на время выполнения сортировки.

Можно предположить, что чем больше элементов содержит массив, тем сильнее будет отличаться время сортировки, причём далеко не в пользу алгоритма простого обмена.

Следовательно, наиболее эффективным алгоритмом из двух представленных является алгоритм шейкерной сортировки.

4. Графики зависимости $C_ф + M_ф$

Ниже представлен график зависимости $C_ф$+$M_ф$ (рис.1) от количества элементов в массиве для двух алгоритмов: алгоритм пузырьковой сортировки (на графике выделен синим цветом); алгоритм шейкерной сортировки (на графике выделен оранжевым цветом).

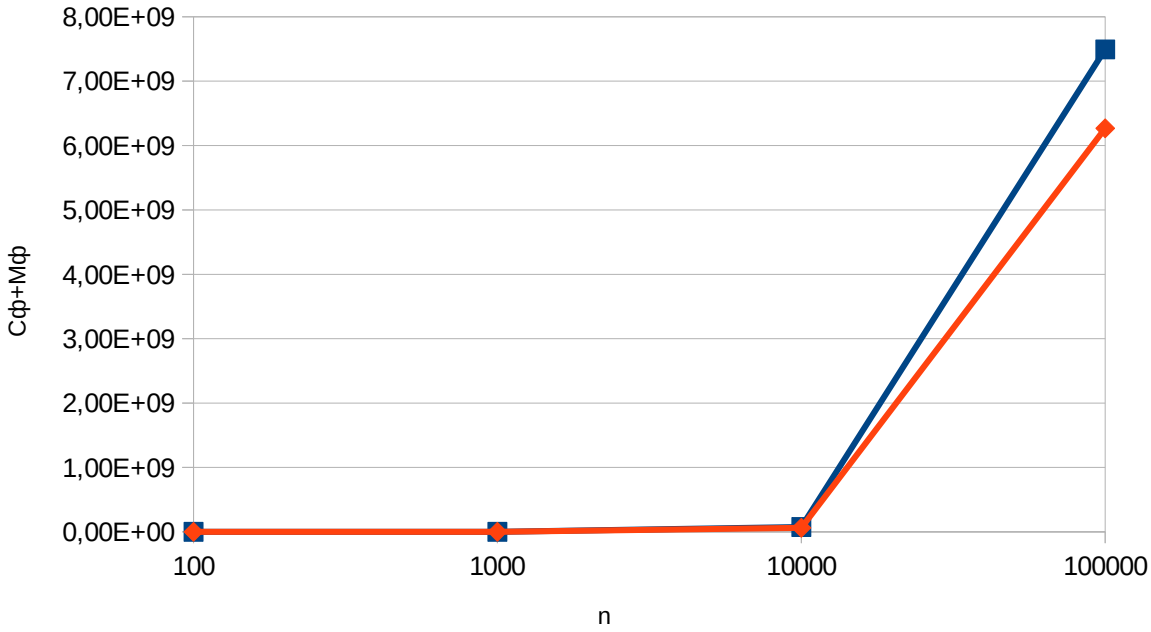

Рис.1 График зависимости $C_ф+M_ф$ ч.1

5. Задача 3

1. Постановка задачи

Разработать алгоритм ускоренной сортировки «Прямое слияние», реализовать алгоритм.

2. Описание подхода к решению

В сортировке слиянием, как и во многих других рекурсивных алгоритмах, используется метод «Разделяй и властвуй». Суть данного метода заключается в следующем: разбить задачу на более мелкие подзадачи, которые легче решаются, и из решений этих подзадач составить решение всей задачи.

То есть, в данном случае задача состоит в сортировке целого массива. Подзадачей можно считать сортировку более мелкого массива (например, массива из одного (двух) элемента(ов)). Затем эти подмассивы можно объединить в один массив.

3. Алгоритм

Для начала стоит определить рекурсивную функцию сортировки слиянием. Пусть *f(a, l, h)* — функция сортировки. *a* — исходный массив, *l* — левая граница массива, *h* — правая граница массива.

Тогда алгоритм будет следующим:

1) Если правая граница массива меньше либо равна левой границе, то закончить исполнение функции;

2) Иначе, рассчитать среднюю границу массива *m*, по формуле $m=\frac{l+h}{2}$;

3) Вызвать функцию *f(a, l, m)*;

4) Вызвать функцию *f(a, m+1, h)*;
5) Объединить подмассивы *a[l...m], a[m+1...h]*.

4. Определение временной сложности

Из источника «Лекция 5 Рекурсивные процессы, реализация и оценка сложности» время работы сортировки слиянием описывается с помощью следующего рекуррентного соотношения (формула 10):

$$T(n)=\begin{cases}\theta(1), & \text{при } n=1,\\ 2T(\frac{n}{2})+\theta(n)\end{cases} \qquad (10)$$

Определим теоретическую сложность в *O*-нотации. Для этого воспользуемся методом подстановки.

Для начала определим верхнюю границу рекуррентного соотношения (формула 11):

$$T(n)=2T(\frac{n}{2})+n \qquad (11)$$

Предположим, что решение имеет вид $T(n)=O(n\log n)$. Докажем, что при подходящем выборе константы $c > 0$ выполняется неравенство $T(n)\le cn\cdot\log n$.

Предположим, что это неравенство справедливо для величины $\frac{n}{2}$, т.е. что выполняется соотношение $T(\frac{n}{2})\le c\cdot\frac{n}{2}\cdot\log\frac{n}{2}$. После подстановки данного выражения в рекуррентное соотношение (формула 11) получаем следующее соотношение (формула 12):

$$T(n)\le 2(c\cdot\frac{n}{2}\cdot\log\frac{n}{2})+n\le cn\cdot\log\frac{n}{2}=cn\cdot\log n-cn\cdot\log 2+n=cn\cdot\log(n)-cn+n=$$
$$=cn\cdot\log n+n(c-1)\le cn\cdot\log n \qquad (12)$$

Следовательно, $T(n)=O(n\log n)$ действительно является решением рекуррентного уравнения.

Таким образом, теоретическая сложность алгоритма сортировки слиянием будет следующей (формула 13):

$$T(n)=O(n\log n) \qquad (13)$$

5. Код функции сортировки

Сама сортировка состоит из двух функций. Первая *merge* — объединяет подмассивы (выполняет слияние), вторая *mergeSort* — сортирует исходный массив.

```
/**
 * Выполняет слияние массива
 * @param array         массив, в котором будет выполняться слияние
 * @param low           левая граница
 * @param middle        средняя граница
 * @param high          правая граница
 */
const merge = (array: Array<number>, low: number, middle: number, high: number):
void => {
  // Слияние array[low...middle] с array[middle+1...high]
  let i: number = low;
  let j: number = middle + 1;

  let extraArray: Array<number> = new Array();

  for (let k: number = low; k <= high; ++k) {
    extraArray[k] = array[k];
  }

  for (let k: number = low; k <= high; ++k) {
    ++mergeComps;
    if (i > middle) {       // элементы из левой половины закончились
      ++mergeTrans;
      array[k] = extraArray[j++];
      continue;
    }

    if (j > high) {       // элементы из правой половины закончились
      ++mergeTrans;
      array[k] = extraArray[i++];
      continue;
    }

    if (extraArray[j] < extraArray[i]) {    // текущий ключ из правой половины
меньше текущего ключа из левой
      ++mergeTrans;
      array[k] = extraArray[j++];
      continue;
    }

    // текущий ключ из левой половины меньше текущего ключа из правой
    array[k] = extraArray[i++];
    ++mergeTrans;
  }
}

/**
 * Выполняет сортировку слиянием для заданного массива array
 * @param array         сортируемый массив
 * @param low           левая граница массива
 * @param high          правая граница массива
```

```
 */
export const mergeSort = (array: Array<number>, low: number, high: number): void
=> {
  if (high <= low) {
    return;
  }

  const middle: number = low + div(high - low, 2);

  mergeSort(array, low, middle);
  mergeSort(array, middle + 1, high);

  merge(array, low, middle, high);
}
```

6. Тесты

Ниже представлена сводная таблица результатов для алгоритма сортировки слиянием (табл.5). Сразу же можно заметить насколько сильно отличаются все три алгоритма по количеству действий за время работы функции:

Таблица 5

n	**T**	**f(C+M)**	**$C_ф$+$M_ф$**
100	0,64 с	664	1344
1000	0,64 с	9966	19952
10000	0,73 с	132877	133616
100000	1,27 с	1660964	3337856

6. Анализ результатов по таблицам 4 и 5

Как видно из таблиц, до *n = 10000* время сортирования почти одинаково. Однако после *n = 10000* сортировка слиянием значительно быстрее справляется с задачей, чем шейкерная сортировка.

Можно предположить, что чем больше будет *n*, тем сильнее будут отличаться результаты. Все из-за того что, зависимости сложности алгоритмов от *n* разные. Шейкерная сортировка зависит от *n* квадратично, поэтому с ростом *n* время будет резко увеличиваться (причём значительно резче, чем у сортировки слиянием).

Следовательно, наиболее эффективным алгоритмом из двух представленных является алгоритм сортировки слиянием. А так как шейкерная сортировка эффективнее пузырьковой сортировки, то сортировка слиянием — самый эффективный алгоритм из всех трёх рассматриваемых.

7. Графики зависимости $C_ф$ + $M_ф$

На рис.2 представлены графики зависимости $C_ф$+$M_ф$ для трёх алгоритмов. Синим цветом выделен график для алгоритма пузырьковой сортировки,

оранжевым цветом выделен график для алгоритма шейкерной сортировки, жёлтым цветом выделен график для алгоритма сортировки слиянием.

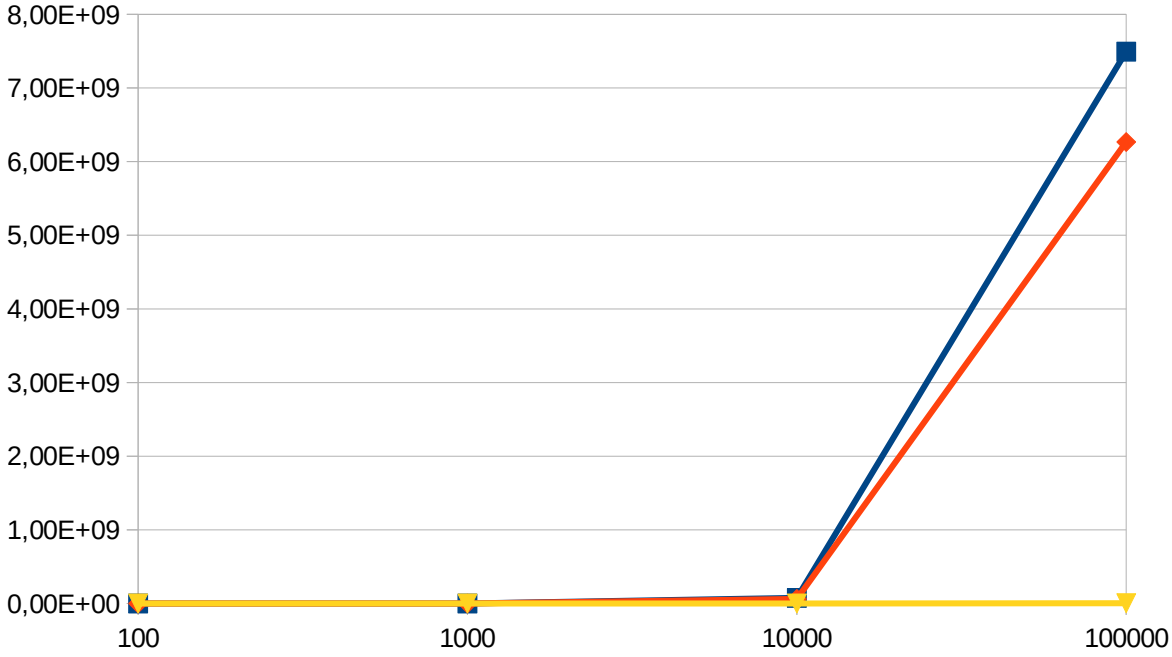

Рис.2 График зависимости $C_ф+M_ф$ ч.2

ОТЧЁТ ПО ЗАДАНИЮ 2

1. Таблицы

Таблицы 6, 7 показывают результаты прогонов программы на массивах, упорядоченных по возрастанию и убыванию соответственно.

Таблица 6

n	**T**	**f(C+M)**	**$C_ф$+$M_ф$**
100	0,62 с	10000	4950
1000	0,64 с	1000000	499500
10000	0,83 с	100000000	49995000
100000	27,91 с	10000000000	4999950000

Таблица 7

n	**T**	**f(C+M)**	**$C_ф$+$M_ф$**
100	0,64 с	10000	9900
1000	0,65 с	1000000	999000
10000	1,02 с	100000000	99990000
100000	42,65 с	10000000000	9999900000

Таблицы 8, 9 показывают результаты прогонов программы на массивах, упорядоченных по возрастанию и убыванию соотвветственно.

Таблица 8

n	**T**	**f(C+M)**	**$C_ф$+$M_ф$**
100	0,63 с	10000	197
1000	0,64 с	1000000	1997
10000	0,63 с	100000000	19997
100000	0,66 с	10000000000	199997

Таблица 9

n	**T**	**f(C+M)**	**$C_ф$+$M_ф$**
100	0,66 с	10000	9900
1000	0,67 с	1000000	999000
10000	1,23 с	100000000	99990000
100000	81,63 с	10000000000	9999900000

Таблицы 10, 11 показывают результаты прогонов программы на массивах, упорядоченных по возрастанию и убыванию соотвветственно.

Таблица 10

n	**T**	**f(C+M)**	**$C_ф$+$M_ф$**
100	0,63 с	664	1344
1000	0,64 с	9966	19952
10000	0,69 с	132877	133616

Продолжение таблицы 10

n	**T**	**f(C+M)**	**$C_ф$+$M_ф$**
100000	1,24 с	1660964	3337856

Таблица 11

n	**T**	**f(C+M)**	**$C_ф$+$M_ф$**
100	0,62 с	664	1344
1000	0,63 с	9966	19952
10000	0,69 с	132877	133616
100000	1,24 с	1660964	3337856

2. Асимптотическая вычислительная сложность

Расчёт вычислительной сложности уже был сделан в п.1, 2, 5 отчёта по заданию 1. Поэтому можно утверждать следующее: все алгоритм, рассматриваемые в данном задании, теоретически, не зависят от случая. Однако самым эффективным среди всех трёх алгоритмов, рассматриваемых в данном задании является алгоритм сортировки слиянием, как уже было упомянуто в п.6 отчёта по заданию 1.

3. Таблица

Согласно формату составления отчёта по заданию 2, нужно заполнить таблицу 12.

Таблица 12

Название алгоритма	**Асимптотическая сложность**			
	Наихудший случай	**Наилучший случай**	**Средний случай**	**Ёмкостная сложность**
Пузырьковая сортировка	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Шейкерная сортировка	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Сортировка слиянием	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

Заметим, что для пузырьковой сортировки и для шейкерной сортировки не нужна дополнительная память, в отличие от сортировки слиянием, поэтому их ёмкостная сложность и равна $O(1)$.

ВЫВОДЫ

В ходе работы получил навыки по анализу вычислительной и ёмкостной сложности алгоритма на массивах, заполненных случайно. Определил наиболее эффективный алгоритм, которым оказался алгоритм сортировки слиянием.