Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития
Кафедра инфокоммуникаций

**ОТЧЕТ**
**ПО ЛАБОРАТОРНОЙ РАБОТЕ №10**
**дисциплины «Алгоритмизация»**
**Вариант___**

Выполнил:
Иващенко Олег Андреевич
2 курс, группа ИВТ-б-о-22-1,
09.03.02 «Информационные и вычислительные машины»,
направленность (профиль)
«Программное обеспечение средств вычислительной техники и автоматизированных систем»

_____
(подпись)

Руководитель практики:
Доцент кафедры инфокоммуникации
Воронкин Роман Александрович

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2023 г.

**Тема**: «Сортировка кучей»

## Порядок выполнения работы

Таблица 1 – Код программы

```
using System;
using System.Diagnostics;

class HeapSort
{
   static void Main()
   {
      // Ввод данных
      Console.WriteLine("[Program] Введите размерность массива:");
      Console.Write(">>> ");
      int N = int.Parse(Console.ReadLine());
      int[] array = new int[N];
      Stopwatch Timer = Stopwatch.StartNew();
      Timer.Start();

      // Заполнение массива
      Random _rnd = new Random();
      for (int i = 0; i < N; i++) array[i] = _rnd.Next(-5 * N, 5 * N);

      // Вывод и сортировка исходного массива
      if (N < 20) for (int i = 0; i < N; i++) Console.WriteLine($"[{i}] {array[i]}");
      HeapSortAlgorithm(array);

      //Вывод отсортированного массива
      Console.WriteLine($"[Program] Время выполнения: {Timer.Elapsed.TotalSeconds} сек.");
      Timer.Stop();
      Console.WriteLine("\n[Program] Отсортированный массив:");
      for (int i = 0; i < N; i++) Console.WriteLine($"[{i}] {array[i]}");
      Console.ReadKey();
   }

   static void HeapSortAlgorithm(int[] array)
   {
      int N = array.Length;

      // Построение кучи (перегруппировка массива)
      for (int i = N / 2 - 1; i >= 0; i--)
         Heapify(array, N, i);

      // Извлечение элементов из кучи
      for (int i = N - 1; i > 0; i--)
      {
         // Перемещаем текущий корень в конец массива
         int temp = array[0];
         array[0] = array[i];
         array[i] = temp;

         // Вызываем процедуру Heapify для уменьшения размера кучи
         Heapify(array, i, 0);
      }
   }

   static void Heapify(int[] arr, int n, int i)
   {
```

```
        int largest = i; // Инициализируем наибольший элемент как корень
        int left = 2 * i + 1; // Левый потомок
        int right = 2 * i + 2; // Правый потомок

        // Если левый потомок больше корня
        if (left < n && arr[left] > arr[largest]) largest = left;

        // Если правый потомок больше корня
        if (right < n && arr[right] > arr[largest]) largest = right;

        // Если самый большой элемент не корень
        if (largest != i)
        {
            int temp = arr[i];
            arr[i] = arr[largest];
            arr[largest] = temp;

            // Рекурсивно вызываем Heapify для поддерева
            Heapify(arr, n, largest);
        }
    }
}
```

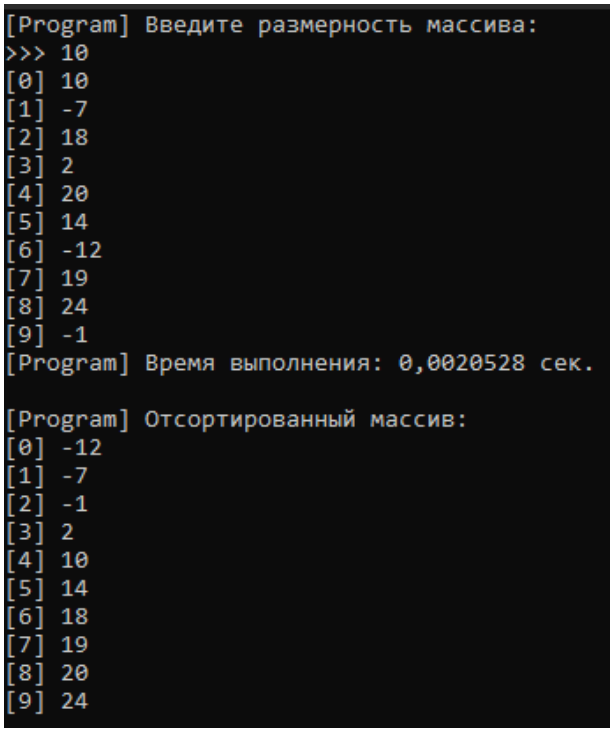


Рисунок 1.1 – Результат выполнения программы

| Случайное заполнение | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Размерность массива M | 100 | 500 | 1 000 | 5 000 | 10 000 | 50 000 | 100 000 | 500 000 | 1 000 000 |
| Время выполнения T, сек | 0,001848 | 0,012936 | 0,001685 | 0,084157 | 0,012545 | 0,101286 | 0,088151 | 1,268258 | 1,83577 |

| Отсортированный массив | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Размерность массива M | 100 | 500 | 1 000 | 5 000 | 10 000 | 50 000 | 100 000 | 500 000 | 1 000 000 |
| Время выполнения T, сек | 0,00925 | 0,018028 | 0,001426 | 0,119637 | 0,031988 | 0,290701 | 0,088726 | 1,144787 | 1,715062 |

| Реверсивный массив | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Размерность массива M | 100 | 500 | 1 000 | 5 000 | 10 000 | 50 000 | 100 000 | 500 000 | 1 000 000 |
| Время выполнения T, сек | 0,005442 | 0,003367 | 0,037479 | 0,014398 | 0,018117 | 0,176576 | 0,180454 | 1,086963 | 1,832588 |

Рисунок 1.2 – Таблица значений для сортировки кучей

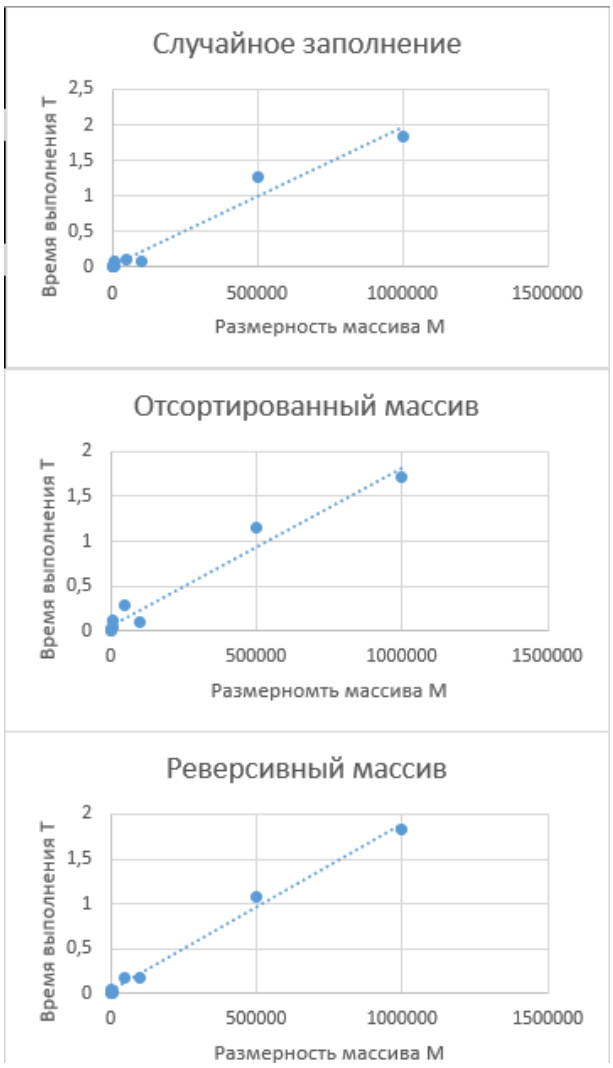


Рисунок 1.3 – Графики отношения времени на количество элементов в массиве

Задание. Даны массивы A[1..n] и B[1..n]. Вывести все n2 сумм вида A[i] + B[j] в возрастном порядке. Наивный способ – создать массив, содержащий все такие суммы, и отсортировать его. Соответствующий алгоритм имеет время работы O(n2 log n) и использует O(n2) памяти. Перевести алгоритм с такие же временем работы, который использует линейную память.

Таблица 2 – Код программы

```
using System;

class Program
{
   static void Main()
   {
      Console.WriteLine("[Program] Введите размерность массивов:");
```

```
        Console.Write(">>> ");
        int N = int.Parse(Console.ReadLine());

        int[] A = new int[N];
        int[] B = new int[N];
        Random _rnd = new Random();

        for (int i = 0; i < N; i++)
        {
            A[i] = _rnd.Next(-500, 500);
            B[i] = _rnd.Next(-500, 500);
        }
        PrintSortedSums(A, B);
        Console.ReadKey();
    }

    static void PrintSortedSums(int[] A, int[] B)
    {
        Array.Sort(A);
        Array.Sort(B);

        int N = A.Length;
        int[] sums = new int[N * N];
        int index = 0;

        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++) sums[index++] = A[i] + B[j];

        Array.Sort(sums);

        Console.WriteLine("[Program] Суммы в возрастающем порядке:");
        foreach (int value in sums) Console.WriteLine(value);
    }
}
```

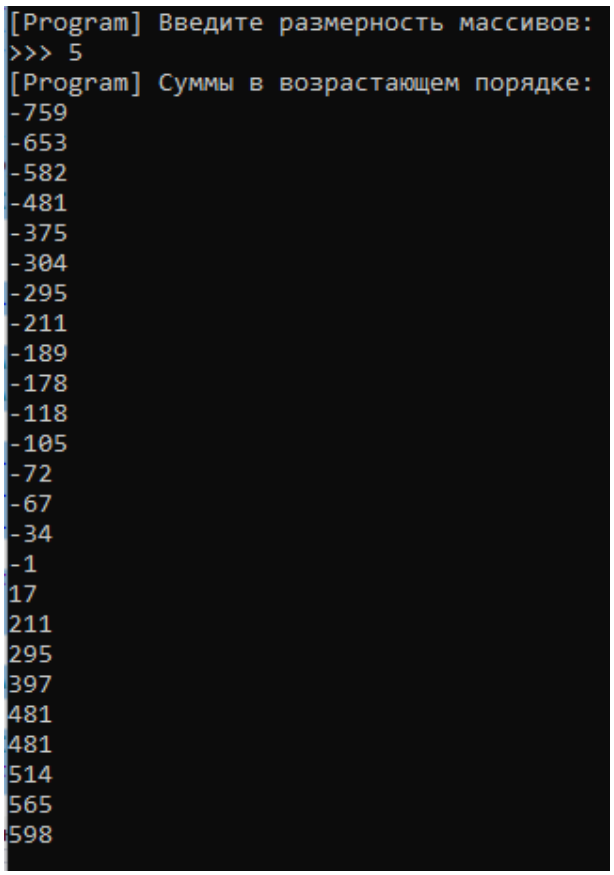


Рисунок 2 – Результат работы программы

**Вывод**: В процессе выполнения практической работы был разобран алгоритм сортировки кучей (Heap Sort), написана программа для сортировки этим алгоритмом. Также он был сравнён с другими алгоритмами, такими как Quick Sort и Merge Sort, из чего можно сделать вывод:

- Heap Sort имеет временную сложность O(n log n) в любом случае. Этот алгоритм не требует дополнительной памяти и устойчив к вариациям входных данных, но работает чуть медленнее по сравнению с некоторым другими алгоритмами сортировки, такими как Quick Sort.
- Quick Sort имеет временную сложность O(n log n) в среднем и лучшем случае и сложность O(n2) в худшем случае (но этого не всегда можно достичь при правильно реализации). Этот алгоритм один из самых быстрых алгоритмов сортировки в среднем случае, не требует дополнительной памяти и сам по себе

использует мало памяти, но неустойчив к вариациям входных данных и может быть неэффективен на некоторых видах данных.

- Merge Sort имеет временную сложность O(n log n) в любом случае. Этот алгоритм стабильный, гарантирует временную сложность в худшем случае и эффективен для больших объёмов данных. Но он использует дополнительную память и на практике может быть медленнее, чем Quick Sort для небольших массивов.

Как итог – если важна эффективность использования памяти и устойчивости к вариациям данных, то Heap Sort может быть хорошим вариантом. Если важна производительность в среднем случае и допускается возможность пожертвовать стабильностью, то стоит использовать Quick Sort. В случае важности стабильности, гарантии временной сложности в худшем случае и не принципиально использование дополнительной памяти, то Merge Sort подходит.