СОРТИРОВКА И ПОИСК

Сортировка

Сортировкой, или упорядочиванием набора элементов называется расположение этих элементов по возрастанию или убыванию согласно определенному линейному отношению порядка, такому, например, как отношение «≤» для целых чисел.

Существуют различные критерии оценки времени выполнения алгоритмов сортировки. Первой и наиболее общей мерой времени выполнения является количество шагов алгоритма, необходимых для упорядочивания n элементов заданного набора. Если размерность набора большая, то перестановка элементов занимает больше времени, чем все другие операции. Поэтому, другой мерой служит количество перестановок элементов, выполненных в ходе алгоритма.

В данном разделе мы рассмотрим сортировку «пузырьком», сортировку вставками, сортировку выбором, сортировку методом Шелла и быструю сортировку, в которых в качестве набора элементов будем рассматривать одномерный массив.

Следует отметить, что реальное количество выполненных действий зависит не только от теоретической сложности алгоритма, но и от структуры сортируемого набора данных. Например, для упорядочивания «случайных» данных лучшие результаты обычно показывает быстрая сортировка, а для случая, когда набор данных уже практически упорядочен, более оптимальной является оптимизированная сортировка «пузырьком».

Метод «пузырька»

Представим, что элементы, подлежащие сортировке, хранятся в массиве, расположенном вертикально. Элементы с малыми значениями более «легкие» и «всплывают» вверх наподобие пузырька. При первом проходе вдоль массива, начиная проход снизу, берется первый элемент массива, и его значение поочередно сравнивается со значениями последующих элементов. Если встречается элемент с большим значением, то эти элементы меняются местами. При встрече элемента с более «легким» значением этот элемент становится эталоном для сравнения, и все последующие элементы сравниваются с ним. В результате элемент с наименьшим значением оказывается в самом верху массива. Во время второго прохода вдоль массива находится элемент со вторым по величине значением, который помещается под элемент, найденный при первом проходе массива, т.е. на вторую сверху позицию, и т.д. Отметим, что во время второго и последующих проходов вдоль массива нет необходимости просматривать элементы, найденные за предыдущие проходы, так как они имеют значения, меньшие, чем у оставшихся элементов массива. Кроме того, если массив содержит n элементов, то потребуется выполнить (n–1) проход вдоль массива. На самом деле: после (n–1) прохода все элементы, кроме последнего, стоят на нужных местах. Но последний элемент, после выполнения (n–1) прохода, имеет наибольшее значение во всем массиве, и, следовательно, тоже стоит на нужном месте.

Рассмотрим следующий массив целочисленных значений: 85, 83, 84, 82, 81, 83, где n=6. Применим к нему алгоритм «пузырька». В таблице, приведенной ниже, показаны 5 проходов вдоль массива. Линии указывают позицию, выше которой элементы массива уже упорядочены.

Начальное положение	**1-й проход**	**2-й проход**	**3-й проход**	**4-й проход**	**5-й проход**
85	81	81	81	81	81
83	85	82	82	82	82
84	83	85	83	83	83
82	84	83	85	83	83
81	82	84	83	85	84
83	83	83	84	84	85

Метод, реализующий рассмотренный алгоритм сортировки, выглядит следующим образом:

```
static void Sort (int[]a)
{
   int temp;
   for(int i = 0; i < a.Length-1; i++)
   {
      for (int j = a.Length-1; j > i; j--)
      {
         if (a[j] < a[j-1])
         {
            temp = a[j];
            a[j] = a[j-1];
            a[j-1] = temp;
         }
      }
   }
}
```

Задание

Измените алгоритм сортировки так, чтобы элементы массива упорядочивались по убыванию.

Алгоритм сортировки методом «пузырька» считается одним из самых неэффективных алгоритмов. Его выполнение потребует (n–1) проходов по массиву, а теоретическое время выполнение данного алгоритма пропорционально n^2.

Следует отметить, что в случае, когда элементы массива уже в значительной степени упорядочены, алгоритм сортировки методом «пузырька» будет выполнять значительное число проходов «вхолостую». Например:

Начальное положение	**1-й проход**	**2-й проход**	**3-й проход**	**4-й проход**	**5-й проход**
82	81	81	81	81	81
81	82	82	82	82	82
83	83	83	83	83	83
83	83	83	83	83	83
85	84	84	84	84	84
84	85	85	85	85	85

В данном случае массив отсортирован уже после первого прохода, но алгоритм выполнит еще 4 прохода и только после этого завершит свою работу.

Специально для задач такого класса алгоритм сортировки методом «пузырька» был оптимизирован. Идея оптимизации заключается в контроле выполнения перестановок во время очередного прохода. Если во время очередного прохода перестановок не было, то массив уже отсортирован, и продолжать работу алгоритма нет необходимости.

Пример кода, реализующего данный алгоритм, приведен ниже:

```
static void Sort(int[] a)
{
   int temp;
   for (int i = 0; i < a.Length - 1; i++)
   {
      bool isSorted = true;
      for (int j = a.Length - 1; j > i; j--)
      {
         if (a[j] < a[j - 1])
         {
            isSorted = false;
            temp = a[j];
            a[j] = a[j - 1];
            a[j - 1] = temp;
         }
      }
      if (isSorted)
         return;
   }
}
```

Реальную сложность данного алгоритма оценить невозможно, не зная ничего о структуре массива. В худшем случае оптимизированный алгоритм «пузырька» потребует столько же проходов и выполнит столько же перестановок, сколько и классический пузырек. Но для случая, когда исходный массив практически отсортирован, данный алгоритм покажет оптимальное время выполнения.

Сортировка вставками

Идея данного алгоритма заключается в том, что на i-ом этапе мы «вставляем» элемент A[i] в нужную позицию среди элементов A[1],A[2],…,A[i–1], которые уже упорядочены. Так, при первом проходе элемент A[1] считается упорядоченным и элемент A[2] будет «вставляться»

относительно него согласно требуемому отношению порядка. На втором проходе упорядочены относительно друг друга будут уже два элемента – A[1] и A[2], а элемент A[3] будет «вставляться» относительно них на нужное место. Всего потребуется выполнить (n–1) проход. Действительно, на последнем, (n–1) проходе элементы A[1],A[2],…,A[n-1] будут упорядочены относительно друг друга, и мы «вставим» последний A[n] элемент на нужное место.

Рассмотрим данный алгоритм для сортировки следующего целочисленного массива: 85, 83, 84, 82, 81, 83. После каждого этапа алгоритма элементы, расположенные выше линии, уже упорядочены, хотя между ними на последующих этапах могут быть вставлены элементы, которые на данном этапе расположены ниже линии.

Начальное положение	**1-й проход**	**2-й проход**	**3-й проход**	**4-й проход**	**5-й проход**
85	83	83	82	81	81
83	85	84	83	82	82
84	84	85	84	83	83
82	82	82	85	84	83
81	81	81	81	85	84
83	83	83	83	83	85

Метод, реализующий рассмотренный алгоритм сортировки, выглядит следующим образом:

```
static void Sort (int[]a)
{
   int temp;
   for( int i = 1; i < a.Length; i++)
   {
      int j = i;
      while (j >= 1 && a[j] < a[j-1])   //1
      {
         temp = a[j];
         a[j] = a[j-1];
         a[j-1] = temp;
         j--;
      }
   }
}
```

Задание

Объясните, для чего в строке 1 используется условие j>=1.

Сортировка посредством выбора

На i-ом этапе сортировки выбирается элемент с наименьшим значением среди элементов A[i],…,A[n] и меняется местами с элементом A[i]. В результате, после i-го этапа все элементы A[1],…,A[i] будут упорядочены. Т.е., на первом этапе ищется минимальный элемент массива и меняется местами с первым элементов. На втором этапе ищется минимальный элемент среди элементов, начиная со второго, и меняется местами со вторым. И так далее. На последнем, (n–1) этапе, ищется минимальный элемент среди A[n–1] и A[n] и меняется местами с A[n–1]. После этого элемент A[n] также будет стоять на своем месте.

Рассмотрим данный алгоритм для сортировки следующего целочисленного массива: 85, 83, 84, 82, 81, 83. Линия в таблице показывает, что элементы, расположенные выше нее, уже упорядочены.

Начальное положение	**1-й проход**	**2-й проход**	**3-й проход**	**4-й проход**	**5-й проход**
85	81	81	81	81	81
83	83	82	82	82	82
84	84	84	83	83	83
82	82	83	84	83	83
81	85	85	85	85	84
83	83	83	83	84	85

Метод, реализующий рассмотренный алгоритм сортировки, выглядит следующим образом:

```
static void Sort (int[]a)
{
   int min;
   int index;
   int i, j;
   for(i = 0; i < a.Length-1; i++)
   {
      index = i;
      min = a[i];
      for (j = i+1; j < a.Length; j++)
      {
         if (a[j] < min)
         {
            min = a[j];
            index = j;
         }
      }
      a[index] = a[i];
      a[i] = min;
   }
}
```


Задание

Измените алгоритм сортировки так, чтобы элементы массива упорядочивались по убыванию.

Замечание

Рассмотренные нами алгоритмы являются простыми алгоритмами сортировки. Время их работы пропорционально n^2 *как в среднем, так и в худшем случае. Если же сравнивать эти алгоритмы с точки зрения количества перестановок, то более предпочтительным оказывается алгоритм выбора. Количество перестановок в этом алгоритме пропорционально* n*, в то время как в первых двух -* n^2*.*

Для больших n *простые алгоритмы сортировки заведомо проигрывают алгоритмам со временем выполнения пропорциональным* $n \log n$*. Значение* n*, начиная с которого быстрые алгоритмы становятся предпочтительней, зависит от многих факторов.*

Для небольших значений n рекомендуется применять простой в реализации алгоритм сортировки Шелла, который имеет временную сложность $O(n^{1,5})$.

Алгоритм сортировки Шелла

Массив A из n элементов упорядочивается следующим образом. На первом шаге упорядочиваются элементы в n/2 парах из двух элементов (A[i],A[n/2+i]) для 1≤i≤n/2; на втором шаге упорядочиваются элементы в n/4 группах из четырех элементов (A[i],A[n/4+i],A[n/2+i],A[3n/4+i]) для 1≤i≤n/4; на третьем шаге упорядочиваются элементы в n/8 группах из восьми элементов и т.д. На последнем шаге упорядочиваются элементы сразу во всем массиве A. На каждом шаге для упорядочивания элементов используется метод сортировки вставками.

Рассмотрим этапы работы алгоритма сортировки Шелла для следующего целочисленного массива: 1, 7, 3, 2, 0, 5, 0, 8.

Начальное положение	**1 шаг**	**2 шаг**	**3 шаг**
1	0	0	0
7	5	2	0
3	0	0	1
2	2	5	2
0	1	1	3
5	7	7	5
0	3	3	7
8	8	8	8

На первом этапе рассматриваются следующие пары: 0-й и 4-й элементы со значением 1 и 0 соответственно, 1-й и 5-й элементы со значением 7 и 5, 2-й и 6-й элементы со значением 3 и 0, 3-й и 7-й элементы со значением 2 и 8. Всего 4 пары по 2 элемента (n=8). В каждой паре упорядочиваем элементы в порядке возрастания методом вставки. В данном случае, в первых трех парах элементы поменяются местами, в четвертой все останется по-прежнему. В результате получим пары (0,1), (5,7), (0,3), (2,8).

На втором этапе рассматриваются четверки: 0-й, 2-й, 4-й и 6-й элементы со значениями 0, 0, 1, 3 соответственно, 1-й, 3-й, 5-й и 7-й элементы со значениями 5, 2, 7, 8. После упорядочивания получим четверки: (0,0,1,3), (2,5,7,8).

На последнем этапе рассматривается весь массив: (0,2,0,5,1,7,3,8). После упорядочивания получаем (0,0,1,2,3,5,7,8) – окончательный результат.

Метод, реализующий рассмотренный алгоритм сортировки, выглядит следующим образом:

```
static void Sort (int[]a)
{
   int temp;
   int i, j, incr = a.Length/2;
   while (incr>0)
   {
      for( i = incr; i<a.Length; i++)
      {
```

```
            j = i - incr;
            while (j>=0)
            {
               if (a[j] > a[j+incr])
               {
                  temp = a[j];
                  a[j] = a[j+incr];
                  a[j+incr] = temp;
                  j = j - incr;
               }
               else
               {
                  j= -1;
               }
            }
         }
         incr = incr/2;
      }
   }
```

Задание

Проверьте, как будет работать алгоритм при сортировке целочисленного массива 85, 83, 84, 82, 81, 83.

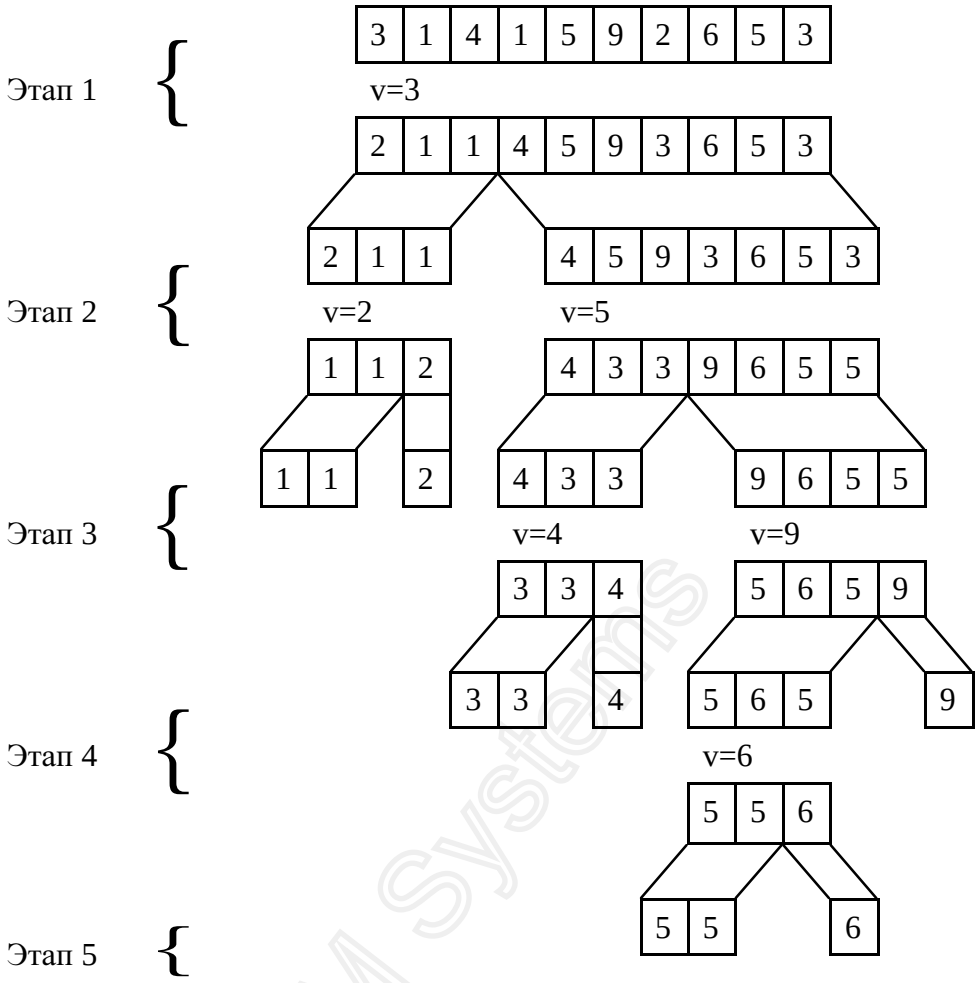
Быстрая сортировка

Теоретическое время выполнения алгоритма быстрой сортировки в среднем пропорционально *n log n*.

В этом алгоритме для сортировки элементов массива A[1],…, A[n] из этих элементов выбирается некоторое значение ключа v в качестве опорного элемента, относительно которого переупорядочиваются элементы массива. Желательно выбрать опорный элемент близким к значению медианы распределения значений ключей так, чтобы опорный элемент разбивал множество ключей на две примерно равные части. Далее элементы массива переставляются так, чтобы для некоторого индекса j все переставленные элементы A[1], …, A[j] имели значения ключей, меньше чем v, а все элементы A[j+1],…, A[n] – значения ключей, больших, или равных v. Затем алгоритм быстрой сортировки рекурсивно применяется к множествам элементов A[1], …, A[j] и A[j+1],…, A[n] для упорядочивания этих множеств по отдельности. Поскольку все значения ключей в первом множестве меньше, чем значения ключей во втором множестве, то исходный массив будет отсортирован правильно. Сортировка завершается тогда, когда отдельные частичные подмножества, на которые разбивается исходный массив в процессе рекурсии, будет содержать элементы с одинаковыми ключами.

Сложность данного алгоритма заключается в том, чтобы найти опорный элемент, т.е., вычислить медиану распределения значений ключей. Поэтому, на практике, в качестве опорного элемента выбираются значение v, которое является наибольшим значение из двух самых левых различных элементов.

Рассмотрим на примере, как будет выполнятся быстрая сортировка последовательности чисел 3, 1, 4, 1, 5, 9, 2, 6, 5, 3.



Задания

1. Проанализируйте, как будет работать алгоритм, если в качестве опорного элемента будет выбираться максимальный элемент в массиве.

2. Проанализируйте, как будет работать алгоритм, если в качестве опорного элемента будет центральный элемент в массиве.

Программная реализация данного алгоритма выглядит следующим образом:

```
class Program
{
   // поиск опорного элемента среди элементов массива a,
   // начиная с позиции i и заканчивая
   // позицией j
   static int FindPivot (int i, int j, int []a)
   {
      int firstKey = a[i];
      for (int k = i + 1; k <= j; k++)
      {
         if (a[k] > firstKey)
         {
            return k;
         }
```

```
            else if (a[k] < firstKey)
            {
               return i;
            }
         }
         //возвращаем тогда, когда различные ключи не найдены
         return -1;
      }
      // выполняет перестановку элементов и возвращает
      // индекс, указывающий на точку разделения фрагмента
      // массива a, где i и j определяют индексы начала и конца
      //фрагмента, на основе опорного элемента pivot
      static int Partition(int i, int j, int pivot, int []a)
      {
         int temp;
         int left = i;
         int right = j;
         do
         {
            temp = a[left];
            a[left] = a[rigth];
            a[rigth] = temp;
            while (a[left] < pivot)
            {
               left++;
            }
            while (a[rigth] >= pivot)
            {
               rigth--;
            }
         }
         while (left <= rigth);
         return left;
      }

      //алгоритм быстрой сортировки
      static void QuickSort(int i, int j, int []a)
      {
         int index = FindPivot(i, j, a);
         // если опорный элемент найден,
         // то рекурсивно упорядочиваем левый и
         // правый фрагменты массива
         if (index != -1)
         {
            int pivot = a[index];
            int k = Partition(i, j, pivot, a);
            QuickSort(i, k-1, a);
            QuickSort(k, j, a);
         }
      }
      static void Print(int []a)
      {
```

```
            foreach(int i in a)
            {
               Console.Write("{0} ",i);
            }
            Console.WriteLine();
         }
         static void Main()
         {
            int []x = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};
            QuickSort(0, x.Length-1, x);
            Print(x);
         }
      }
```

Поиск

Извлечение информации является одной из важнейших прикладных задач: мы указываем имя и запрашиваем список телефонных переговоров, мы указываем номер счета и запрашиваем перечень транзакций, осуществленных по этому счету, и т.д. В этих примерах мы предполагаем, что имеется список записей, каждая из которых состоит из нескольких элементов, которые называют полями. Одно из полей называется ключевым полем, или ключом – по нему и осуществляется поиск. Способ организации записей и выбор метода поиска может оказать существенное влияние на быстроту работы программы, особенно в случае больших объемом информации. В данном разделе мы рассмотрим два наиболее простых алгоритма поиска – последовательный поиск и бинарный поиск.

Далее будем считать, что дан массив, элементами которого являются целые числа. Т.е., запись состоит из одного поля – ключевого. В этом случае задача сводится к поиску в массиве элемента с заданным значением.

Последовательный поиск

Этот алгоритм применяется в том случае, если нет никакой дополнительной информации о данных, среди которых осуществляется поиск. Идея данного алгоритма заключается в том, чтобы просматривать элементы массива с одного конца, например, с начала, и сравнивать значение каждого элемента с искомым до тех пор, пока искомый элемент не будет найден, или мы не дойдем до другого конца массива.

Если поиск даст отрицательный результат, то будет выполнено n операций сравнения – это худшая производительность алгоритма. Наилучшая производительность данного алгоритма будет равна 1 в том случае, если первый же элемент равен искомому.

Метод, реализующий рассмотренный алгоритм поиска, выглядит следующим образом:

```
static int Search(int[] a, int key)
{
   bool found = false;
   int index = 0;
   while (!found && index < a.Length)
   {
      if (a[index] == key)
      {
         found = true;
      }
```

```
            else
            {
               index++;
            }
         }
         if (found)
         {
            return index;
         }
         else
         {
            return -1;
         }
      }
```

Задания

Объясните, в каком случае метод вернет значение -1. Что будет возвращать метод, если поиск увенчался успехом?

Разработайте модификацию метода так, чтобы поиск можно было выполнять не с начала массива, а с заданной позиции.

Двоичный поиск

Алгоритм последовательного поиска работает для любого массива, однако время выполнения его пропорционально количеству элементов массива, что не очень эффективно для «длинных» массивов. Если появляется дополнительная информация о массиве, то возможно сделать поиск более эффективным. Так, например, для отсортированного массива можно использовать метод двоичного поиска, время выполнения которого пропорционально $lg\ n$ (n – количество элементов массива).

Идея данного метода заключается в том, что искомый элемент сравнивается со средним элементом массива, если средний элемент равен искомому, то поиск завершается, иначе поиск продолжается лишь в половине массива – левой, или правой, в зависимости от того, оказался искомый элемент меньше, или больше среднего элемента.

```
      static int BinarySearch(int[] a, int key)
      {
         bool found = false;
         int left = 0;
         int right = a.Length;
         while (!found)
         {
            index = (left+right)/2;
            if (a[index] == key)
            {
               found = true;
            }
            else
            {
               if (a[index]>key)
               {
                  right = index;
               }
```

```
            else
            {
               left = index;
            }
         }
      }
      return index;
   }
```

Задания

1) *Объясните, как поведет себя метод, если искомый элемент будет отсутствовать в массиве.*
2) *Измените метод так, чтобы он корректно работал при любых значениях искомого элемента.*
3) *Разработайте модификацию метода так, чтобы поиск можно было осуществлять на некотором фрагменте массива.*

Оценка реального времени выполнения программ

При решении практических задач необходимо не только оценивать теоретическое время выполнения алгоритма, но и производить измерение реального времени выполнения программы. Сделать это можно разными способами, начиная от написания простейших счетчиков и заканчивая использованием встроенных в Visual Studio версии Team Edition тестов производительности. Мы рассмотрим пример создания собственного счетчика.

Одним из самых простых способов изменения времени является использование класса Stopwatch, объявленного в пространстве имен System.Diagnostics. Объект данного класса имеет два основных метода Start и Stop, соответственно запускающих и останавливающих измерение времени, а также свойство ElapsedMilliseconds, показывающее, сколько времени прошло между вызовами этих методов в миллисекундах.

Ниже приведен пример измерения времени работы функции Sort, сортирующей массив классическим методом «пузырька».

```
using System;
using System.IO;
using System.Diagnostics;
namespace MyProgram
{
   class Program
   {
      static void Sort (int[]a)
      {
         int temp;
         for(int i = 0; i < a.Length-1; i++)
         {
            for (int j = a.Length-1; j > i; j--)
            {
               if (a[j] < a[j-1])
               {
                  temp = a[j];
                  a[j] = a[j-1];
                  a[j-1] = temp;
               }
```

```
                }
            }
        }
        static void Main()
        {
            int[] x = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};
            Stopwatch sw = new Stopwatch();
            sw.Start(); // начинаем измерение
            Sort(x);
            sw.Stop();  // заканчиваем измерение
            //выводим на экран измеренное время сортировки
            Console.WriteLine(sw.ElapsedMilliseconds);
            Console.ReadLine();
        }
    }
}
```

Нужно отметить, что любое измерение реального времени работы приложения необходимо проводить на по возможности «чистой» машине, т.е., машине с минимумом установленных программ и отключенными службами, нагружающими центральный процессор, например, антивирусом. Это делают для того, чтобы свести к минимуму случайное воздействие на результат измерения посторонних процессов. Кроме этого, для получения достоверного результата измерение необходимо повторить, как минимум, несколько десятков раз.

Приложение при этом должно быть скомпилировано в режиме Release и запущено при отключенной отладке (Start without Debugging). Это делается для того, чтобы исключить любой контроль со стороны отладчика, который в состоянии существенно замедлить работу некоторых фрагментов кода.

Также необходимо отметить, что работа с любыми измерителями времени в .NET имеет одну особенность. Дело в том, что согласно концепции компиляции «на лету», методы в больших программах компилируются не в момент запуска программы, а по мере их вызова, что позволяет создать иллюзию быстрого запуска приложения. Из этого следует, что при первом вызове метода Sort может произойти его компиляция, что внесет в работу измерителя существенную погрешность. Поэтому рекомендуется перед реальными измерениями вставить как минимум один вызов функции с тестовыми данными, а само измерение проводить не сразу после запуска программы, а спустя некоторое время, чтобы компилятор успел завершить свою работу.

Задание

Вычислите реальное время выполнения всех рассмотренных алгоритмов сортировок для целочисленных массивов:
1. {1, 2, 3, 6, 4, 5, 7, 8, 9, 8, 10, 11, 10, 12, 13, 14, 15, 16, 16, 15, 17, 18, 19, 20, 21}
2. {21, 13, 19, 1, 6, 7, 18, 5, 16, 8, 2, 20, 9, 12, 4, 17, 14, 10, 15, 3, 10, 15, 11, 16, 8}
и проведите анализ полученных данных.

Примеры использования алгоритмов сортировок и поиска

Пример 1

Дана матрица размерностью n×m, содержащая целые числа. Отсортировать каждый столбец матрицы по убыванию элементов методом вставки.

```
class Program
{
   static void Print(int[,] a)
   {
      for (int i = 0; i < a.GetLength(0); i++)
      {
         for (int j = 0; j < a.GetLength(1); j++)
         {
            Console.Write("{0} ", a[i, j]);
         }
         Console.WriteLine();
      }
   }
   static void Input(out int[,] a)
   {
      Console.Write("n= ");
      int n = int.Parse(Console.ReadLine());
      Console.Write("m= ");
      int m = int.Parse(Console.ReadLine());
      a = new int[n,m];
      for (int i = 0; i < a.GetLength(0); i++)
      {
         for (int j = 0; j < a.GetLength(1); j++)
         {
            Console.Write("a[{0},{1}]= ", i, j);
            a[i,j] = int.Parse(Console.ReadLine());
         }
      }
   }
   static void Sort (int[] a)
   {
      int temp;
      for( int i = 2; i < a.Length; i++)
      {
         int j = i;
         while (j >= 1 && a[j] < a[j-1])
         {
            temp = a[j];
            a[j] = a[j-1];
            a[j-1] = temp;
            j--;
         }
      }
   }
   static void Main()
   {
      int[,] a;
      Input(out a);
      Console.WriteLine("Исходный массив:");
      Print(a);
      //размерность вспомогательного массива равна
      //количеству строк в исходном массиве
      int[] b = new int [a.GetLength(0)];
```

```
            //каждый столбец массива a копируем в массив b,
            //сортируем массив b по убыванию элементов методом
            //выбора, затем копируем элементы массива b
            //в обрабатываемый столбец массива a
            for (int j = 0; j < a.GetLength(1); j++)
            {
               for (int i = 0; i < a.GetLength(0); i++)
               {
                  b[i] = a[i,j];
               }
               Sort(b);
               for (int i = 0; i < a.GetLength(0); i++)
               {
                  a[i,j] = b[i];
               }
            }
            Console.WriteLine("Измененный массив:");
            Print(a);
         }
      }
```

Результат работы программы:

```
Исходный массив:
23  54  65  -9  0
8   96  -4  -7  6
100 12  90  1   2
2   -1  4   -5  -12
Изменённый массив:
100 96  90  1   6
23  54  65  -5  2
8   12  4   -7  0
2   -1  -4  -9  -12
```

Задание

Измените программу так, чтобы была отсортирована каждая строка двумерного массива.

Пример 2

Дана матрица размерностью n×m, содержащая целые числа. Отсортировать каждую строку матрицы по возрастанию элементов, используя алгоритм выбора.

Указания по решению задачи. При решении данной задачи удобно использовать ступенчатый массив.

```
class Program
{
   static void Print(int[][] a)
   {
      for (int i = 0; i < a.Length; i++)
      {
         for (int j = 0; j < a[i].Length; j++)
         {
            Console.Write("{0} ", a[i][j]);
         }
         Console.WriteLine();
```

```
   }
}
static void Input( out int[][]a)
{
   Console.Write("n= ");
   int n = int.Parse(Console.ReadLine());
   Console.Write("m= ");
   int m = int.Parse(Console.ReadLine());
   a = new int[n][];
   for (int i = 0; i < a.Length; i++)
   {
      a[i] = new int[m];
      for (int j = 0; j < a[i].Length; j++)
      {
         Console.Write("a[{0}][{1}]= ", i, j);
         a[i][j] = int.Parse(Console.ReadLine());
      }
   }
}
static void Sort (int[]a)
{
   int min;
   int index;
   int i, j;
   for( i = 0; i < a.Length-1; i++)
   {
      index = i;
      min = a[i];
      for (j = i+1; j < a.Length; j++)
      {
         if (a[j] < min)
         {
            min = a[j];
            index = j;
         }
      }
      a[index] = a[i];
      a[i] = min;
   }
}
static void Main()
{
   int [][]a;
   Input(out a);
   Console.WriteLine("Исходный массив:");
   Print(a);
   //сортируем каждую строку ступенчатого массива
   foreach(int[] x in a)
   {
      Sort(x);
   }
```

```
            Console.WriteLine("Измененный массив:");
            Print(a);
        }
    }
```

Результат работы программы:

```
Исходный массив:
23  54  65  -9  0
8   96  -4  -7  6
100 12  90  1   2
2   -1  4   -5  -12
Изменённый массив:
-9  0   23  54  65
-7  -4  6   8   96
1   2   12  90  100
-12 -5  -1  2   4
```

Пример 3

Известно, что каждая строка целочисленного массива n×n ровно один раз содержит значение x. Определить индекс этого элемента в каждой строке массива до и после упорядочивания элементов строки по возрастанию.

```
class Program
{
    static void Print(int[][] a)
    {
        for (int i = 0; i < a.Length; i++)
        {
            for (int j = 0; j < a[i].Length; j++)
            {
                Console.Write("{0} ", a[i][j]);
            }
            Console.WriteLine();
        }
    }
    static void Input(out int[][]a)
    {
        Console.Write("n= ");
        int n = int.Parse(Console.ReadLine());
        Console.Write("m= ");
        int m = int.Parse(Console.ReadLine());
        a = new int[n][];
        for (int i = 0; i < a.Length; i++)
        {
            a[i] = new int[m];
            for (int j = 0; j < a[i].Length; j++)
            {
                Console.Write("a[{0}][{1}]= ", i, j);
                a[i][j] = int.Parse(Console.ReadLine());
            }
        }
    }
```

```
static int Search(int[] a, int key)
{
   bool found = false;
   int index = 0;
   while ( !found && index < a.Length)
   {
      if (a[index] == key)
      {
         found = true;
      }
      else
      {
         index++;
      }
   }
   if (found)
   {
      return index;
   }
   else
   {
      return -1;
   }
}
static void Sort (int[]a)
{
   int min;
   int index;
   int i, j;
   for(i = 0; i < a.Length - 1; i++)
   {
      index = i;
      min = a[i];
      for (j = i+1; j < a.Length; j++)
      {
         if (a[j] < min)
         {
            min = a[j];
            index = j;
         }
      }
      a[index] = a[i];
      a[i] = min;
   }
}
static int BinarySearch(int[] a, int key)
{
   bool found = false;
   int left = 0;
   int right = a.Length;
   int index = -1;
   while (!found)
   {
```

```
            index = (left + right) / 2;
            if (a[index] == key)
            {
               found = true;
            }
            else
            {
               if (a[index] > key)
               {
                  right = index;
               }
               else
               {
                  left = index;
               }
            }
         }
         return index;
      }
      static void Main()
      {
         int[][] a;
         Input(out a);
         Console.Write("x= ");
         int x = int.Parse(Console.ReadLine());
         Console.WriteLine("Исходный массив:");
         Print(a);
         Console.WriteLine("Данные до сортировки:");
         //в каждой строке массива ищем индекс элемента  x,
         //и сразу сортируем строку
         for(int i = 0;i < a.GetLength(0); i++)
         {
            Console.WriteLine("В строке {0} индекс {1}",
                                    i, Search(a[i], x));
            Sort(a[i]);
         }
         Console.WriteLine("Измененный массив:");
         Print(a);
         Console.WriteLine("Данные после сортировки:");
         //повторно для каждой строки массива ищем индекс элемента x
         for(int i = 0;i < a.GetLength(0); i++)
         {
            Console.WriteLine("В строке {0} индекс {1}",
                                  i, BinarySearch(a[i], x));
         }
      }
   }
```

Результат работы программы:

```
Исходный массив:
1  2  5  3
4  5  2 10
9  0  4  5
```

```
5  2  7  8

X=5

Данные до сортировки:
В строке 0 индекс 2
В строке 1 индекс 1
В строке 2 индекс 3
В строке 3 индекс 0

Измененный массив:
1  2  3  5
2  4  5 10
0  4  5  9
2  5  7  8

Данные после сортировки:
В строке 0 индекс 3
В строке 1 индекс 2
В строке 2 индекс 2
В строке 3 индекс 1
```

Практикум №7

Дана матрица размерностью n×n, содержащая целые числа. Отсортировать:

1) каждую строчку матрицы по убыванию элементов методом «пузырька»;
2) каждую строчку матрицы по убыванию элементов методом выбора;
3) каждую строчку матрицы по убыванию элементов методом вставки;
4) каждый столбец матрицы по возрастанию элементов методом Шелла;
5) каждый столбец матрицы по возрастанию элементов алгоритмом быстрой сортировки;
6) каждый столбец матрицы по возрастанию элементов методом «пузырька»;
7) диагонали матрицы, параллельные главной, по убыванию элементов методом выбора;
8) диагонали матрицы, параллельные главной, по убыванию элементов методом вставки;
9) диагонали матрицы, параллельные главной, по убыванию элементов алгоритмом Шелла;
10) диагонали матрицы, параллельные главной, по убыванию элементов методом быстрой сортировки;
11) диагонали матрицы, параллельные побочной, по возрастанию элементов методом «пузырька»;
12) диагонали матрицы, параллельные побочной, по возрастанию элементов методом выбора;
13) диагонали матрицы, параллельные побочной, по возрастанию элементов методом вставки;
14) диагонали матрицы, параллельные побочной, по возрастанию элементов алгоритмом Шелла;
15) каждый столбец матрицы с номером 2i по убыванию элементов, а с номером 2i+1 по возрастанию элементов методом быстрой сортировки;
16) каждый столбец матрицы с номером 2i по возрастанию элементов, а с номером 2i+1 по убыванию элементов методом «пузырька»;

17) диагонали матрицы, расположенные выше главной, по убыванию элементов, а диагонали матрицы, расположенные ниже главной, по возрастанию элементов методом выбора;
18) диагонали матрицы, расположенные выше главной, по возрастанию элементов, а диагонали матрицы, расположенные ниже главной, по убыванию методом вставки;
19) диагонали матрицы, расположенные выше побочной, по убыванию элементов, а диагонали матрицы, расположенные ниже побочной, по возрастанию элементов алгоритмом Шелла;
20) диагонали матрицы, расположенные выше побочной, по возрастанию элементов, а диагонали матрицы, расположенные ниже побочной, по убыванию элементов методом быстрой сортировки.

Самостоятельная работа №4

Задание 1

Дана последовательность, состоящая из N целых чисел. Отсортировать ее, используя алгоритм:

1) быстрой сортировки;
2) пирамидальной сортировки;
3) поразрядной сортировки;
4) сортировки подсчетом;
5) сортировки слиянием.

Задание 2

Предположим, что необходимо отсортировать список элементов, состоящий из уже упорядоченного списка, который следует за несколькими «случайными» элементами. Какой из рассмотренных в этой главе, или изученных вами самостоятельно методов сортировки наиболее подходит для решения этой задачи?

Задание 3

Алгоритм сортировки называется устойчивым, если он сохраняет исходный порядок следования элементов с одинаковыми значениями ключей. Какие из рассмотренных в этой главе, или изученных вами самостоятельно методов сортировки являются устойчивыми?

Задание 4

Напишите программу нахождения k наименьших элементов в массиве длины n. Каково теоретическое время выполнения этой программы? Для каких значений k эффективней выполнить сначала сортировку всего массива, а затем взять k наименьших элементов, вместо поиска k наименьших элементов в неупорядоченном массиве.

Задание 5

Напишите программу нахождения наиболее часто встречаемого элемента в массиве из n элементов и оцените теоретическое время выполнения этой программы.