

## Алгоритмы поиска в базовых коллекциях (структурах данных)

Исследуем алгоритмы поиска значения в массиве: **линейный** поиск, **бинарный** и (на следующей лекции) поиск в **хеш-таблице**.

Для сравнения эффективности алгоритмов и получения «драматической» разницы в скорости работы рассмотрим практический пример обработки входящих в программу данных.

Искать в коллекции мы будем не числовые значения, а строковые (что усложняет задачу и соответствует практической направленности примера).

У нас имеется класс FileRecord

```
public class FileRecord
{
    Ссылка: 4
    public string Id { get; set; }

    Ссылка: 1
    public string Name { get; set; }

    Ссылка: 1
    public long Size { get; set; }
}
```

Во внешнем файле "records.xml" в формате xml хранится более 100 тысяч таких записей.

Во внешнем файле "filter.txt" в формате txt хранится более 30 тысяч строк с идентификаторами.

Задача – в программе нужно получить список записей FileRecord, в котором будут только те (из 200 тысяч файла records) идентификатор которых присутствует в файле filter.txt

Посмотрим, как можно быстро решить эту задачу.

Функция ReadRecords для чтения записей FileRecord из xml файла.

```
public static List<FileRecord> ReadRecords(string filename)
{
    var records = new List<FileRecord>();
    var xs = new XmlSerializer(typeof(List<FileRecord>));
    var fs = new FileStream(filename, FileMode.Open);
    records = (List<FileRecord>) xs.Deserialize(fs);
    fs.Close();

    return records;
}
```

Загружаем исходные записи и фильтр

```
// читаем все записи
var records = ReadRecords("records.xml");
Console.WriteLine("found = {0} records", records.Count);

// читаем фильтр (массив строк с идентификатором)
var filterIds = File.ReadLines("filter.txt").ToList();
Console.WriteLine("found = {0} ids for filter", filterIds.Count);
```

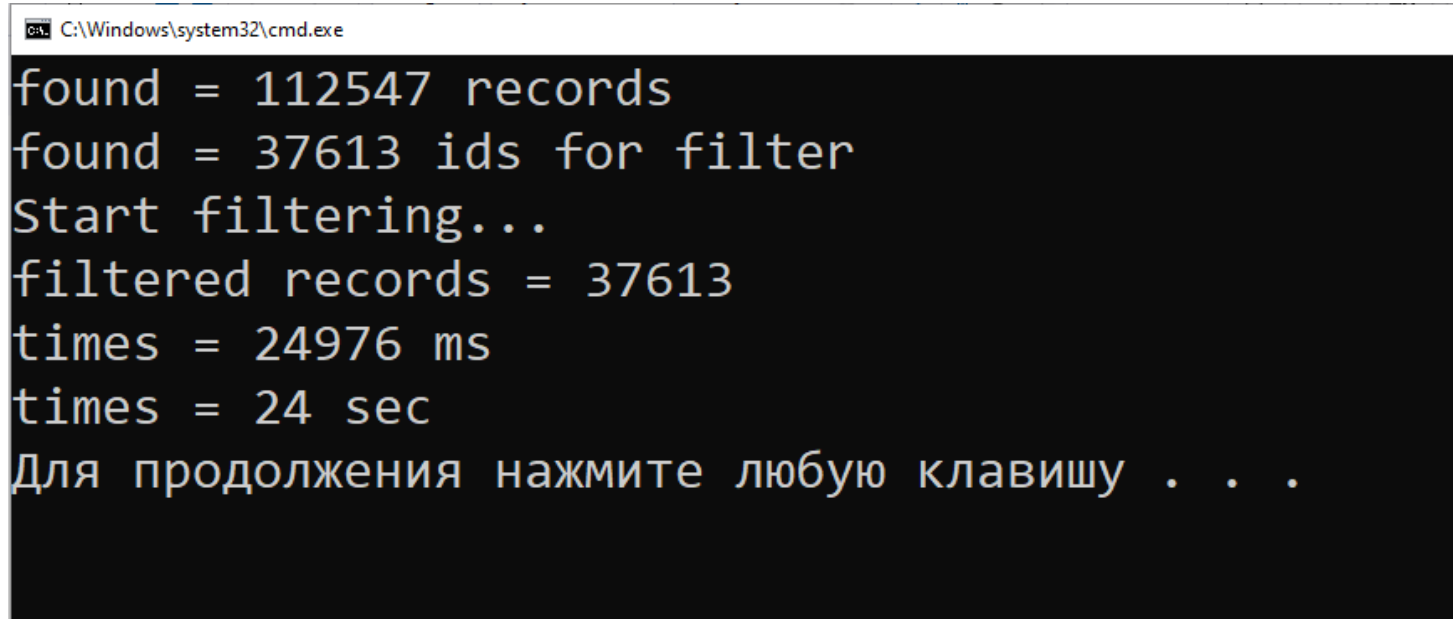
Выполняем фильтрацию с использованием линейного поиска (и замеряем время)

```
// секундомер для измерения времени
var stopwatch = new Stopwatch();
stopwatch.Start();

// фильтруем (ЛИНЕЙНЫЙ ПОИСК) используется в стандартном методе Contains
Console.WriteLine("Start filtering...");
var filteredRecords = records
    .Where(r => filterIds.Contains(r.Id))
    .ToList();
Console.WriteLine("filtered records = {0}", filteredRecords.Count);
stopwatch.Stop();

// выводим затраченные миллисекунды и секунды
Console.WriteLine("times = {0} ms", stopwatch.ElapsedMilliseconds);
Console.WriteLine("times = {0} sec", stopwatch.ElapsedMilliseconds/1000);
```

Результат работы программы.

A screenshot of a Windows command prompt window. The title bar at the top reads "C:\Windows\system32\cmd.exe". The command prompt displays the following text: "found = 112547 records", "found = 37613 ids for filter", "Start filtering...", "filtered records = 37613", "times = 24976 ms", "times = 24 sec", and "Для продолжения нажмите любую клавишу . . .".

```
C:\Windows\system32\cmd.exe  
found = 112547 records  
found = 37613 ids for filter  
Start filtering...  
filtered records = 37613  
times = 24976 ms  
times = 24 sec  
Для продолжения нажмите любую клавишу . . .
```

На выполнение этой простой задачи при использовании линейного поиска было потрачено 24 секунды!!!

Обратите внимание, что измерение времени работы программы не является надёжным инструментом оценки эффективности (на затраченное время могут влиять случайные факторы, связанные с работой операционной системы).

Вам нужно провести исследование эффективности работы линейного и бинарного поиска на этих входных данных.

Доработаем наш класс ArrayDecorator<T>, добавив в него линейный поиск и изменив тип счётчика операций на long (операций очень много!)

```
5  |  /// <summary>
6  |  /// Класс-обёртка для массива с подсчётом операций
7  |  /// </summary>
8  |  /// <typeparam name="T"></typeparam>
9  |  Ссылка: 10
10 |  public class ArrayDecorator<T> where T: IComparable
11 |  {
12 |      private T[] _array;
13 |      protected long _operationCount;
14 |
15 |      Ссылка: 2
16 |      public void AddOperation()
17 |      {
18 |          _operationCount++;
19 |      }
20 |
21 |      // линейный поиск вхождения с подсчётом операций
22 |      Ссылка: 1
23 |      public bool Contains_LinearSearch(T id)
24 |      {
25 |          for (int i = 0; i < _array.Length; i++)
26 |          {
27 |              int compare = _array[i].CompareTo(id);
28 |              if (compare == 0)
29 |                  return true;
30 |          }
31 |
32 |          return false;
33 |      }
34 |
35 |      public int Length
36 |      {
37 |          get {
38 |              return _array.Length;
39 |          }
40 |      }
41 |
42 |      Ссылка: 2
43 |      public long OperationCount
44 |      {
45 |          get {
46 |              return _operationCount;
47 |          }
48 |      }
49 |
50 |      // Индексатор - (на базе Проперти)
51 |      Ссылка: 4
52 |      public T this[int index]
53 |      {
54 |          get
55 |          { // операция чтения
56 |              _operationCount++;
57 |              return _array[index];
58 |          }
59 |          set
60 |          { // операция записи
61 |              _operationCount++;
62 |              _array[index] = value;
63 |          }
64 |      }
65 |  }
```

```
62  ✓
63
64  ✓
65  _array = null;
66  _operationCount = 0;
67  }
68
69  Ссылка: 2
70  public ArrayDecorator(T[] array)
71  {
72      _array = array;
73      _operationCount = 0;
74  }
75
76  Ссылка: 1
77  public T[] GetSource()
78  {
79      return _array;
80  }
81
82  Ссылка: 1
83  public void WrapFor(T[] another)
84  {
85      _array = another;
86      _operationCount = 0;
87  }
88
89  Ссылка: 2
90  public void WrapFor(ArrayDecorator<T> another)
91  {
92      _array = another.GetSource();
93      _operationCount += another.OperationCount;
94  }
95  }
```

И создадим класс-обёртку `StringDecorator` для строки (`string`) с дополнительным функционалом – подсчётом операций (чтения букв и их сравнения) при сравнении строк! Этот Класс будет хранить ссылку на `ArrayDecorator`, для того чтобы считать все операции в одном объекте!

```

5  </summary>
6  /// Наш класс-обёртка для строки, с подсчётом операций
7  /// чтения букв, + содержит ссылку на массив(декоратор),
8  /// который хранит все строки и ведёт общий подсчёт операций
9  /// </summary>
10 Ссылка: 7
11 public class StringDecorator : IComparable
12 {
13     Ссылка: 0
14     public override string ToString()
15     {
16         return _source;
17     }
18
19     Ссылка: 2
20     public int Length{
21         get {return _source.Length;}
22     }
23
24     protected int _operations;
25     protected string _source;
26     protected ArrayDecorator<StringDecorator> _arrayStorage;
27
28     Ссылка: 2
29     public char this[int index]{
30         get
31         {
32             _arrayStorage.AddOperation();
33             return _source[index];
34         }
35     }
36
37     Ссылка: 2
38     public StringDecorator(string source, ArrayDecorator<StringDecorator> array)
39     {
40         _source =source;
41         _arrayStorage = array;
42     }
43 }

```

```

39  ✓      /// <summary>
40  |      /// Наша Функция сравнения строк с подсчётом выполняемых операций
41  |      /// </summary>
42  |      /// <param name="obj">объект (строка) с которым сравниваем</param>
    |      Ссылка: 0
43  ✓      public int CompareTo(object obj)
44  |      {
45  |          var another = (StringDecorator)obj;
46  |          for (int i = 0; i < this.Length; i++)
47  |          {
48  |              if (i > another.Length)
49  |              {
50  |                  return -1;
51  |              }
52  |              char char1 = this[i];
53  |              char char2 = another[i];
54  |              int delta = char1.CompareTo(char2);
55  |
56  |              _arrayStorage.AddOperation(); // учёт операции сравнения
57  |
58  |              if (delta != 0)
59  |                  return delta;
60  |          }
61  |          return 0;
62  |      }
63  |  }
64  |  }
--

```

Подготавливаем все декораторы (для массива строк и для каждой строки) и теперь снова выполняем фильтрацию данных с линейным поиском, но уже с подсчётом всех операций.

```

// подготавливаем все декораторы (для массива строк и для каждой строки)
int filterSize = filterIds.Length/1;
var filterDecorator = new ArrayDecorator<StringDecorator>();
var array1 = filterIds.Take(filterSize).Select(x => new StringDecorator(x, filterDecorator))
    .ToArray();
filterDecorator.WrapFor(array1);

// запускаем фильтрацию
int recordsSize = records.Count/1;
Console.WriteLine("Start filtering...");
var filteredRecords = records.Take(recordsSize)
    .Where(r => filterDecorator.Contains_LinearSearch( new StringDecorator(r.Id, filterDecorator)))
    .ToList();
Console.WriteLine("filtered records = {0}", filteredRecords.Count);
Console.WriteLine("operations = {0}", filterDecorator.OperationCount);

```



Обратите внимание на переменные `filterSize` и `recordSize` – с их помощью (делением на одинаковое число) можно управлять объёмом выборки, над которой выполняется фильтрация.

- 1) Постройте диаграмму (график зависимости) количества операций от объёма выборки (например делением `Size` на 100, 90, 80, 70, ... 1) для алгоритма **линейного поиска**.
- 2) Добавьте в класс `ArrayDecorator` свой (самописный) алгоритм бинарного поиска, отсортируйте массив `filterIds`  
“`var orderedFilterIds = filterIds.OrderBy(x=>x)`”  
перед созданием на его основе массива `array1` элементов `StringDecorator` и после оберните объектом `filterDecorator.Wrap(array1)`.
- 3) Теперь постройте диаграмму (график зависимости) количества операций от объёма выборки (например делением `Size` на 100, 90, 80, 70, ... 1) для алгоритма **бинарного поиска**.

Проведите исследование и выполните сравнительный анализ зависимостей для двух алгоритмов.