Experimental Evaluation of Algorithms

Eleni Mangina

Room B2.05
School of Computer Science and Informatics
University College Dublin, Ireland

Overview

- Performance is a key issue in software engineering.
  - An efficient algorithm can significantly reduce the time taken to complete long running computations.
- The practical impact of an efficient algorithm can be:
  - a significant increase in the number of concurrent users that can be handled by an application, or
  - to make an application practical by reducing the time taken to compute certain information.
- In this part of the course, we will explore how to perform an **experimental analysis** that allows us to compare the performance of two or more algorithms.
  - Later, we will explore more a formal approach to analysing algorithms.

Basic Idea

- We perform an experimental analysis by **timing the execution of each algorithm** under certain well-defined conditions:
  - The inputs for both algorithms must be as similar as possible.
- For array-based algorithms, this means:
  - The same array size
  - The same set of values (possibly not in the same order)
- For searching of arrays, this also means:
  - The same set of queries (target values).
- The basic idea is to evaluate each algorithm under as similar conditions as possible.

Basic Idea

- Unfortunately, because modern operating systems are multi-tasking, we must also deal with variations in the execution environment.
  - Different system tasks may be performed while one of the experiments is running, causing skewing of results.

- To cater for this, we **run the experiment multiple times**:
  - This may be multiple times with the same conditions.
  - It may also require that we run the same experiment multiple times with variations of the same conditions.

- For each run, we generate a **data set** that represents a specific set of conditions.
  - For arrays, we use **multiple data sets of different sizes**.

Basic Idea

- Timings can be generated by using one of the following methods:
  - `System.currentTimeMillis()`: returns a long value that represents the current time in milliseconds since midnight on January 1 1970 (UTC - Coordinated Universal Time).
  - `System.nanoTime()`: returns a long value that represents the current time in nanoseconds relative to some arbitrary time point (possibly the future).

- We can use these methods to get the time before and after the execution of the algorithm and then compute the time taken by taking their difference.
  - I.e. timeTaken = finishTime - startTime

Basic Idea

- When we run the experiment:
  - We generate/use multiple data sets.
  - For each data set, we obtain a set of timings
    - One for each algorithm being evaluated
  - We then plot a graph showing how the timings change as the data sets change.

- For arrays, each data set will represent a specific size of array.
  - Often, we initialise that array with random values.
  - We then plot a graph that maps array size to time taken.

- Lets have a look at an example…

Example Experiment

- Lets compare two algorithms for calculating the prefix averages of integer values in an array.

- A prefix average is calculated relative to a position, `p`, in an array.
  - Formally, the prefix average of position, `p`, in an array of integers, `array` is the average of all the values in the range `array[0], …, array[i]`.

- Our algorithms will compute the prefix average for each position in a given array.
  - It will output an array, `prefix`, of prefix averages, where the value at position, `pos`, holds the prefix average of the input array relative to that position.

First Approach

- We will assume the existence of an integer array, `array`, that is initialised with a random set of values.

- Algorithm:

```
Algorithm PrefixAverage1(A, n):
   Input: An integer array A of size n.
   Output: An array X of size n such that X[j] is the average of A[0], …, A[j].

   Let X be an integer array of size n
   for j=1 to n-1 do
       a ← 0
      for k=1 to j do
          a ← a + A[k]

       X[j] ← a / (j+1)

   return X
```

First Approach

- We will assume the existence of an integer array, `array`, that is initialised with a random set of values.

- Java Code:

```java
public static int[] prefixAverage1(int[] array, int n) {
  int[] prefixAverage = new int[n];
  for (int i=0; i<n; i++) {
      int prefixSum = 0;
      for (int j=0; j<i; j++)
          prefixSum += array[j];
      prefixAverage[i] = prefixSum / (i+1);
  }
  return prefixAverage;
}
```

Second Approach

- We will assume the existence of an integer array, `array`, that is initialised with a random set of values.

- A Better Algorithm:

Algorithm PrefixAverage2(A, n):

Input: An integer array A of size n.

Output: An array X of size n such that X[j] is the average of A[0], …, A[j].

Let A be an array of n numbers.

s $\leftarrow$ 0
for i $\leftarrow$ 0 **to** *n-1* **do**
    s $\leftarrow$ s + X[***i***]
    A[***i***] $\leftarrow$ s/(***i***+ 1)

return array A

Second Approach

- We will assume the existence of an integer array, `array`, that is initialised with a random set of values.

- Java:

```
public static int[] prefixAverage2(int[] array, int n) {
  int[] prefixAverage = new int[n];
  int runningSum = 0;
  for (int i=0; i<n; i++) {
          runningSum += array[i];
          prefixAverage[i] = runningSum / (i+1);
  }
  return prefixAverage;
}
```

Combining Approaches

```
public class Timings {
    public static void main(String[] args) {
        final int N = 100000;
        Random generator = new Random();

        int[] array = new int[N];
        for (int i=0; i<N; i++) array[i] = generator.nextInt();

            // Approach 1:
        prefixAverage1(array, array.length);

        // Approach 2:
           prefixAverage2(array, array.length);
    }
}
```

Adding Timings

```
public class Timings {
    public static void main(String[] args) {
        final int N = 100000;
        Random generator = new Random();

        int[] array = new int[N];
        for (int i=0; i<N; i++) array[i] = generator.nextInt();

        long start = System.currentTimeMillis();
        // Approach 1:
        prefixAverage1(array, array.length);
        long end = System.currentTimeMillis();
        long timing1 = end - start;

        start = System.currentTimeMillis();
        // Approach 2:
        prefixAverage2(array, array.length);
        end = System.currentTimeMillis();
        long timing2 = end - start;
        System.out.println(array.length + “,” + timing1 + “,” + timing2);
    }
}
```

Expected Output

- This program will output one line of code:

  100000,8843,1

- This represents the information that:
  - Algorithm 1 took 8843 milliseconds to calculate the prefix averages for an array of size 100,000
  - Algorithm 2 took 1 millisecond to calculate the prefix averages for an array of size 100,000.

- The format of the output is known as Comma Separated Values (CSV) and we use it because it can be read directly into an Excel Spreadsheet.
  - Each comma is interpreted as a column separator and each newline is interpreted as a row separator.

Extending the Comparison

- Okay, so this gives us a comparison for one value of N (=100,000).

- What we are really interested in is how the algorithms compare over a range of values for N.
  - That is, for array based algorithms, we are interested in exploring how the size of the array affects the performance of the algorithm.

- To achieve this, we extend our program to include a range of values for N.
  - There are two ways of doing this:
    - Use a for loop to specify fixed increments for N in a given range.
    - Use an array that holds the set of values of N over which the algorithms are to be evaluated.

Fixed Increment Ranges

```
public class Timings {
    public static void main(String[] args) {
        Random generator = new Random();

        for (int N = 0; N <= 250000; N+=25000) {
            int[] array = new int[N];
            for (int i=0; i<N; i++) array[i] = generator.nextInt();

            long start = System.currentTimeMillis();
            // Approach 1:
            prefixAverage1(array, array.length);
            long end = System.currentTimeMillis();
            long timing1 = end - start;

            start = System.currentTimeMillis();
            // Approach 2:
            prefixAverage2(array, array.length);
            end = System.currentTimeMillis();
            long timing2 = end - start;
            System.out.println(array.length + “,” + timing1 + “,” + timing2);
        }
    }
}
```

Example Output

```
0,0,0
25000,493,1
50000,1971,0
75000,4655,0
100000,9886,1
125000,12327,1
150000,17682,1
175000,23985,2
200000,31385,2
225000,39735,2
250000,49101,2
```

Example Output

```
0,0,0
25000,493,1
50000,1971,0
75000,4655,0
100000,9886,1
125000,12327,1
150000,17682,1
175000,23985,2
200000,31385,2
225000,39735,2
250000,49101,2
```

But, in the earlier experiment, the timings for N=100,000 were: 100000,8843,1

There is over 1 second difference in the timing of the first algorithm!

Set of Values Ranges

```
public class Timings {
    public static void main(String[] args) {
        Random generator = new Random();
        int[] N = new int[] {0, 10000, 20000, 30000, 50000, 75000, 100000, 200000,
    300000};
        for (int k = 0; k < N.length; k++) {
            int[] array = new int[N[k]];
            for (int i=0; i<array.length; i++) array[i] = generator.nextInt();

            long start = System.currentTimeMillis();
            // Approach 1:
            prefixAverage1(array, array.length);
            long end = System.currentTimeMillis();
            long timing1 = end - start;

            start = System.currentTimeMillis();
            // Approach 2:
            prefixAverage2(array, array.length);
            end = System.currentTimeMillis();
            long timing2 = end - start;
            System.out.println(array.length + “,” + timing1 + “,” + timing2);
        }
    }
}
```

Example Output

```
0,0,0
10000,81,0
20000,315,0
30000,707,1
50000,1968,1
75000,4404,1
100000,7843,2
200000,32256,2
300000,71681,2
```

Example Output

```
0,0,0
10000,81,0
20000,315,0
30000,707,1
50000,1968,1
75000,4404,1
100000,7843,2
200000,32256,2
300000,71681,2
```

And now we get another different set of timings for N=100,000:
100000,8843,1
100000,9886,1

There is over 2 seconds difference in the timings for the first algorithm!

Results Smoothing

- This diversity of timings arises because of the multi-tasking nature of the underlying operating systems.

- Unfortunately, there is no real way to get around this…

- The best we can do is to work out an average timing for each data set.
  - For example, run each algorithm 5 times…

- You can do this with both the approaches we have discussed.
  - As an example, we will do it with the fixed increment approach.

Results Smoothing

```
public class Timings {
    public static void main(String[] args) {
        final int ITERATIONS = 5;
        Random generator = new Random();
        for (int N = 0; N <= 250000; N+=25000) {
            // initialise data set…
            long timing1 = 0;
            for (int it=0; it<ITERATIONS; it++) {
                long start = System.currentTimeMillis();
                prefixAverage1(array, array.length);
                long end = System.currentTimeMillis();
                timing1 += end - start;
            }
            timing1 = timing1 / ITERATIONS;

            long timing2 = 0;
            for (int it=0; it<ITERATIONS; it++) {
                long start = System.currentTimeMillis();
                prefixAverage2(array, array.length);
                long end = System.currentTimeMillis();
                timing2 += end - start;
            }
            timing2 = timing2 / ITERATIONS;
            System.out.println(array.length + “,” + timing1 + “,” + timing2);
        }
    }
}
```

Example Output

```
0,0,0
25000,496,0
50000,1992,0
75000,4660,0
100000,8038,1
125000,12978,1
150000,17647,1
175000,24041,1
200000,31609,1
225000,41067,2
250000,53846,2
```

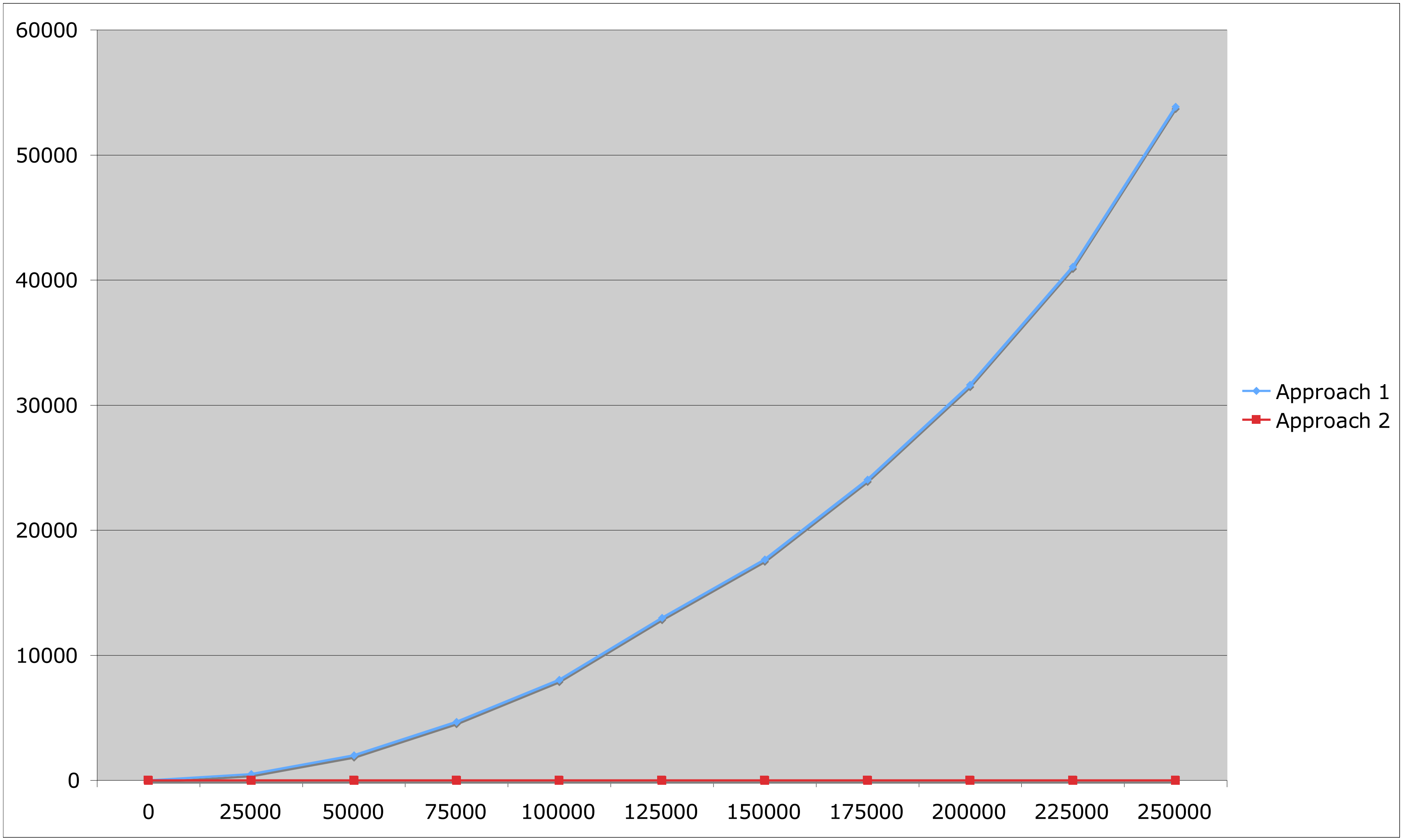
Importing into Excel

- Copy the output data from the IDE output console.
- Create a new text file using a .csv extension, and paste the output data into the empty file.
- Add a new line at the top of the file to include headings:
  - E.g. “Array Size,Approach 1,Approach 2”
- Launch MS Excel and select “Open…”
  - Choose the newly created text file.
- This should load the data into the spreadsheet.

Creating a Chart

- Go to the Insert menu and select “Chart…”
- Select “Line Chart” type and click “Next”
- For the data range, select the values in columns B and C.
- Click on the “Series” tab, and select the values in column A as the “X” axis labels.
- Click on “Finish” and hey presto!

Example Chart



Analyse Results

- As we can see, approach 1 is much slower than approach 2.
- In fact, approach 1 seems to to take polynomial time whereas approach 2 seems to take linear time…
- Anyway, we can see that, in general, approach 2 is clearly better than approach 1!
- Isn’t that useful to know…