Data Structures Coursework 1

100135292

Wed, 6 Dec 2017 13:56

Contents

data-structures-algorithms.pdf **2**

DataStructuresCW1.java **13**

data-structures-algorithms.pdf

 (Included PDF starts on next page.)

Data Structures and Algorithms Coursework 1

100135292

December 5, 2017

1 Writing Pseudo code to find Sub sequences in a 2D array

Algorithm 1 findSeries in a 2D array

Input: $n \times n$ 2D array **T**, Subsequence **S** of length **k**

Output: i, Which denotes the series and index, j, Which denotes the position in the series i.

```
1: lowest ← MAX − VALUE {initialise lowest found to MAX value}
2: result ← 0 {initialise result to 0}
3: index, position ← 0
4: for i ← 1 to n do
5:   for j ← 1 to (n − k) + 1 do
6:     iterator ← j {initialise iterator to j}
7:     square ← 0
8:     for l ← k do
9:       value ← (S[l] − T[i][iterator])
10:      square ← square + (value ∗ value) {to store the squares}
11:      iterator ← iterator + 1
12:    end for
13:    result = √(square)
14:    if result < lowest then
15:      lowest = result
16:      index = i
17:      position = j
18:    end if
19:  end for
20: end for
21: PRINT "LOWEST VALUE IS AT INDEX: index, AND POSITION: position"
22: return lowest
```

Please note

2 Formal Analysis of Algorithm

2.1 Describing the Case

The fundamental operation of the algorithm is:

value (S[l] ← T[i][iterator])

When looking at the algorithm its clear to understand that the best, average, and worst case scenarios are all the same. This is because even though the series may be found immediately at a distance of 0/0, the algorithm will continue to scan throughout the entirety of the 2D array until the end of the array is reached. This logic also applies to the average case scenario, which leads on to the end of the 2d array even if the element is found. This can be described as every case is a worst case scenario as they will all perform the same amount of fundamental operations regardless of the distance found at.

2.2 Run time complexity

The run time complexity is defined by the amount of times the fundamental operation is called while the algorithm is running. we can break this down into the three loops:

$$\sum_{i=1}^{n}(\sum_{j=1}^{(n-k)+1}(\sum_{l=1}^{k}1))$$

Breaking this down we can come up with: $\mathbf{n((n-k)+1(k))}$

This can then be further simplified to: $\mathbf{n}((\mathbf{n}+\mathbf{1})\mathbf{k}-\mathbf{k^2})$

And further still: $\mathbf{kn}(-\mathbf{k}+\mathbf{n}+\mathbf{1})$

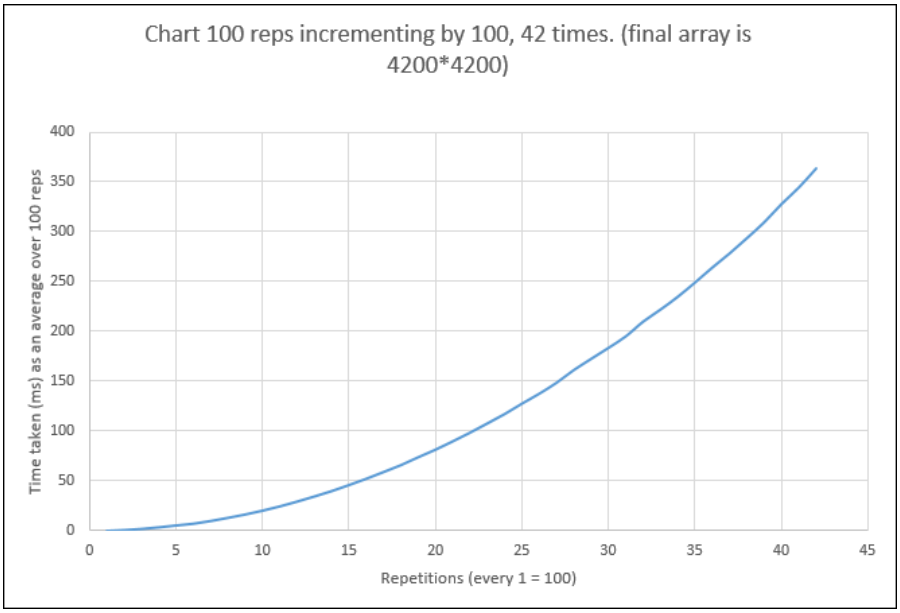
By using this final formula we are able to determine how many fundamental operations will be performed from the size of the array being scanned. In the formula, n represents the n*n array, and k represents the sub-series. (For instance it's possible to see how many times the fundamental operation will be performed on a 7x7 array with sub-series size 3 by simply plugging in the correct numbers to the formula: 3*7(-3+7+1) = 105)

2.3 Order of the Algorithm

The last point to look at when analysing the algorithm is to characterise the function to give the order of the algorithm.
In the pseudo code provided above we are able to note that there are two primary loops that are used to iterate across and down the 2D array of characters. As the two loops are scanning every character in every row in the 2D array, this gives the algorithm a order of:

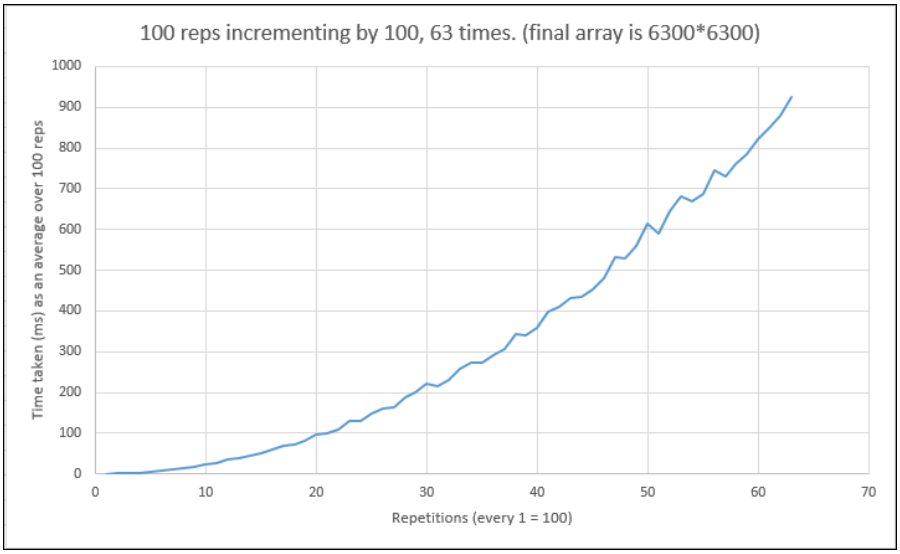
$$O(n^2)$$

This is also further supported by graphs to follow.



At 42 reps(starting at 100 & incrementing by 100 each iteration) there is a smooth increasing curve. This matches the pattern for an algorithm of order $O(n^2)$

There are some slightly noticeable irregularities to the line, stopping it from being a perfectly smooth curve, these will be discussed later.

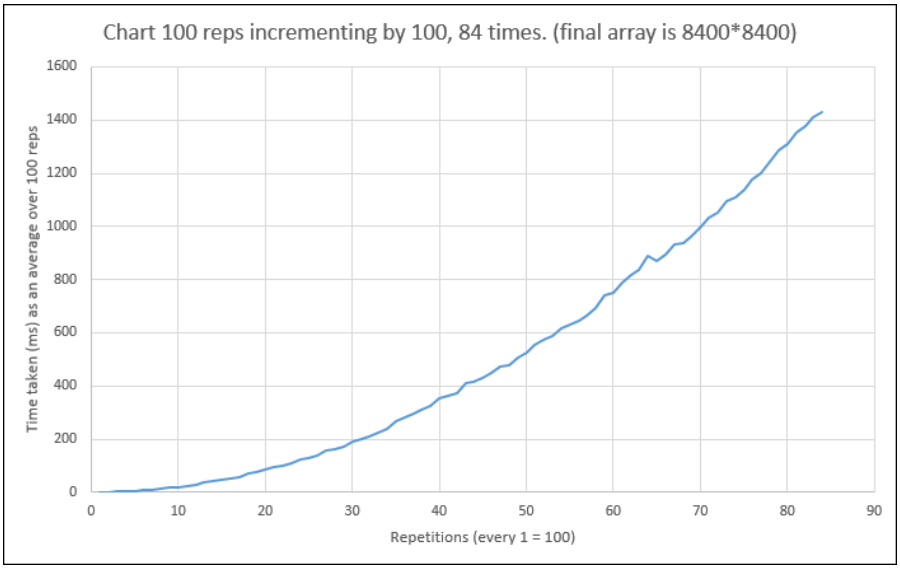


At 63 repetitions the curve has become much more sporadic. There are many different contributing factors that could contribute to the instability of the graph, such as:

- The CPU could be affected by other programs running on the computer which could take up processing power from the CPU that would have been used to work on the algorithm.

- Every CPU is built slightly different with different specifications, so performance may vary on different CPU's, even though they may be the same brand/model. This is a well known phenomenon online, and is often to referred to as 'The Silicone Lottery' as some CPU's posses the ability to be over clocked far beyond their stated maximum.

- Not enough RAM could slow the program down because it would mean as the algorithm runs the calculations would have to pass between physical memory and the CPU cache which can lead to scattering and slowing down the process.



At 84 iterations through (up to a 8400*8400 array) the time taken increases exponentially. the curve in this graph is much smoother than in the graph before, thus enforcing the fact that the 6300*6300 was affected other factors than the efficiency of the algorithm.

3 What is the lower bound?

Proving the lower bounds of an algorithm means to prove the algorithm is using no less of a resource than is required.

The lower bound for all algorithms to solve this problem is that the algorithm finds a perfect match of 0. If a match of 0 is found and the algorithm completes. This would be the lower bound because it would take no more or less time than needed to find the match.

4 Improvements

In the current state of the algorithm, if there is a perfect match of 0 found the algorithm will continue to search the entirety of the 2D array even though a perfect match may have already been found. For instance, if searching for a sequence of 3 and a perfect match of 0 is found in the first position of an array of size 8000*8000 the algorithm will continue to iterate through the rest of the array; using the formula we derived earlier of $kn(-k+n+1)$ we are able to plug in the appropriate values $(3 * 8000(-3 + 8000 + 1))$ and see that a further

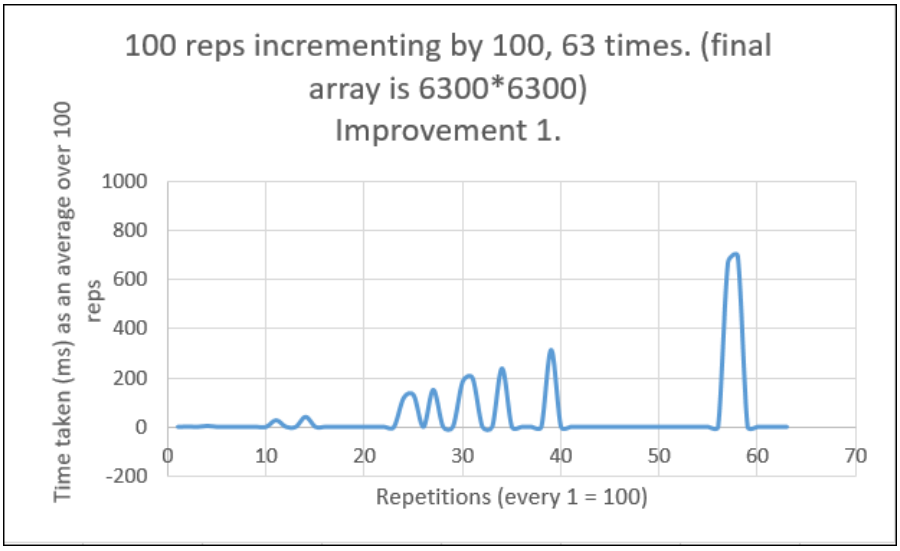
191, 951, 999 calls of the fundamental operation would be done throughout the rest of the array.

Improvements to the algorithm could be made that would increase the constant time factor of the algorithm. One such change that could be effective would be adding an $'IF'$ statement into the code that would break the current loop should there be a match of 0 found. Below is a graph showing the time improvement on the 6300*6300 run with the addition of the $'IF'$ statement.

Algorithm 2 findSeries in a 2D array

Input: $n \times n$ 2D array **T**, Subsequence **S** of length **k**
Output: i, Which denotes the series and index, j, Which denotes the position in the series i.

1: $lowest \leftarrow MAX - VALUE$ {*initialise lowest found to MAX value*}
2: $result \leftarrow 0$ {*initialise result to 0*}
3: $index, position \leftarrow 0$
4: **for** $i \leftarrow 1$ to n **do**
5: **for** $j \leftarrow 1$ to $(n - k) + 1$ **do**
6: $iterator \leftarrow j$ {*initialise iterator to j*}
7: $square \leftarrow 0$
8: **for** $l \leftarrow k$ **do**
9: $value \leftarrow (S[l] - T[i][iterator])$
10: $square \leftarrow square + (value * value)$ {*to store the squares*}
11: $iterator \leftarrow iterator + 1$
12: **end for**
13: result $= \sqrt{(\mathbf{square})}$
14: **if** $result == 0$ **then**
15: BREAK LOOP
16: **end if**
17: **if** $result < lowest$ **then**
18: lowest = result
19: index = i
20: position = j
21: **end if**
22: **end for**
23: **if** $result == 0$ **then**
24: BREAK LOOP
25: **end if**
26: **end for**
27: PRINT "LOWEST VALUE IS AT INDEX: index, AND POSITION: position"
28: **return** *lowest*



This change is shown to improve the speed of the algorithm in comparison to before shown in the graph here. The speed improvement is marginally faster with unusual jumps

A further improvement that can be done is by adding a place holder variable assigned to Double.MAX_VALUE, and recording the total of the K numbers in sequence being squared, then adding a simple $'IF'$ statement that tests if the sum of the K numbers squared is greater than the place holder. If the sum of the K numbers squared is greater than the currently stored place holder then break the loop and move on to the next k numbers. This is done because square rooting a larger number would never result in a lower number than square rooting a smaller number. If the number is smaller, set place holder to the currently lowest square, then continue with the loop.

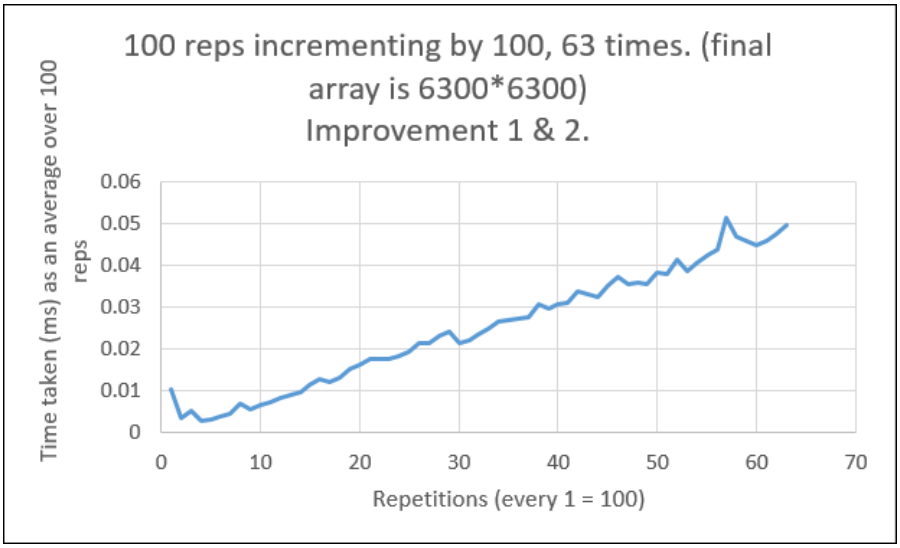
This saves magnitudes of time because the $Math.sqrt(X)$ function is very greedy with time and take up a lot of time by doing the calculation every single time the loop is run. By eliminating the need to do this function every time the loop iterates, we save time on calculations which adds up greatly across the entirety of the array.

Algorithm 3 findSeries in a 2D array FINAL IMPROVEMENT

Input: $n \times n$ 2D array **T**, Subsequence **S** of length **k**

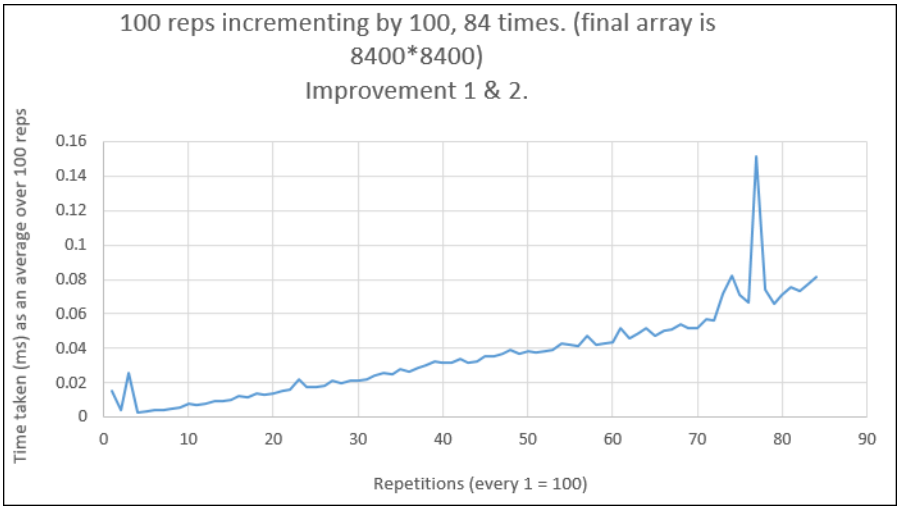
Output: i, Which denotes the series and index, j, Which denotes the position in the series i.

1: $lowest \leftarrow MAX - VALUE$ {*initialise lowest found to MAX value*}
2: $PLACEHOLDER \leftarrow MAX - VALUE$
3: $result \leftarrow 0$ {*initialise result to 0*}
4: $index, position \leftarrow 0$
5: **for** $i \leftarrow 1$ to n **do**
6: **for** $j \leftarrow 1$ to $(n - k) + 1$ **do**
7: $iterator \leftarrow j$ {*initialise iterator to j*}
8: $square \leftarrow 0$
9: **for** $l \leftarrow k$ **do**
10: $value \leftarrow (S[l] - T[i][iterator])$
11: $square \leftarrow square + (value * value)$ {*to store the squares*}
12: $iterator \leftarrow iterator + 1$
13: **end for**
14: **if** square ¿ PLACE HOLDER **then**
15: BREAK LOOP
16: **end if**
17: result $= \sqrt{(\textbf{square})}$
18: **if** $result == 0$ **then**
19: BREAK LOOP
20: **end if**
21: **if** $result < lowest$ **then**
22: lowest = result
23: index = i
24: position = j
25: **end if**
26: **end for**
27: **if** $result == 0$ **then**
28: BREAK LOOP
29: **end if**
30: **end for**
31: PRINT "LOWEST VALUE IS AT INDEX: index, AND POSITION: position"
32: **return** *lowest*



Clearly shown in the graph we can see that the algorithm has become many many magnitudes faster than it was previously. This is achieved by adding in the function that checks for the square of the summed variables that are being scanned in the array. By adding both of the improvements together $(IF(result == 0), BREAKLOOPANDif(square > PLACEHOLDER), BREAKLOOP)$ the algorithm has gone from taking 32 minutes to run, right down to 17 seconds. This is a magnitude of 113 times faster, making the algorithm much faster.

Although the algorithm is still considerably faster, the time taken for each run still increases quite drastically as the size of the array increases over time. This is shown by looking at the graph increasing over time. Also see below the final graph showing the time taken for the same algorithm with the improvements increasing over time.


100 reps incrementing by 100, 84 times. (final array is 8400*8400)
Improvement 1 & 2.
Time taken (ms) as an average over 100 reps
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
10
20
30
40
50
60
70
80
90
Repetitions (every 1 = 100)

DataStructuresCW1.java

```
1  package datastructurescw1;
2
3  import java.text.DecimalFormat;
   import java.util.*;
5
   public class DataStructuresCW1 {
7
       static double[][] arrayFill(double n[][]) {
9          Random p = new Random();
           int min = 1;
11         int max = 10;
           double randomInteger;
13
           for (int i = 0; i < n.length; i++) {
15             for (int j = 0; j < n.length; j++) {
                   n[i][j] = randomInteger = min + p.nextDouble() * max;
17 //                  System.out.print(n[i][j] + "\t");
               }
19 //             System.out.println("\n");
           }
21         return n;
       }
23                     //QUESTION 3 NO IMPROVEMENTS //
       static double findSeries(int subSeq[], double serArray[][]) {
25         double lowest = Double.MAX_VALUE;
           double result = 0;
27         double value;
           double placeHold = Double.MAX_VALUE;
29         int row = 0, pos = 0;
           for (int i = 0; i < serArray.length; i++) {
31             for (int j = 0; j < (serArray.length - subSeq.length) + 1; j++) {
                   int iter = j;
33                 double square = 0;
                   for (int l = 0; l < subSeq.length; l++) {
35                     value = subSeq[l] - serArray[i][iter];
                       square += value * value;
37                     iter++;
                   }
39                 if (square > placeHold) {
                       break;
41                 }
                   placeHold = square;
43                 result = Math.sqrt(square);
   //                System.out.println(result);
45                 if (result < lowest) {
                       lowest = result;
47                     row = i;
                       pos = j;
49                 }
                   if (result == 0) {
51                     break;
                   }
53             }
               if (result == 0) {
55                 break;
               }
57
           }
59 //         System.out.println("Lowest is in row: " + row + " position: " + pos);
           return lowest;
61     }
```

```
                //QUESTION 5 -- METHOD WITH IMPROVEMENTS//
63          static double findSeriesImprov(int subSeq[], double serArray[][]) {
            double lowest = Double.MAX_VALUE;
65          double result = 0;
            double value;
67          double placeHold = Double.MAX_VALUE;
            int row = 0, pos = 0;
69          for (int i = 0; i < serArray.length; i++) {
                for (int j = 0; j < (serArray.length - subSeq.length) + 1; j++) {
71                  int iter = j;
                    double square = 0;
73                  for (int l = 0; l < subSeq.length; l++) {
                        value = subSeq[l] - serArray[i][iter];
75                      square += value * value;
                        iter++;
77                  }
                    result = Math.sqrt(square);
79  //                System.out.println(result);
                    if (result < lowest) {
81                      lowest = result;
                        row = i;
83                      pos = j;
                    }
85              }
            }
87  //          System.out.println("Lowest is in row: " + row + " position: " + pos);
            return lowest;
89      }

91      static int[] subArrayFill(int x[]) {
            Random p = new Random();
93  //          System.out.print("SubArray: ");
            for (int i = 0; i < x.length; i++) {
95              x[i] = p.nextInt(10);
    //              System.out.print(x[i] + " ");
97          }
            return x;
99      }

101     public static void timingExperiment(int x, int n, int reps) {
            int subArray[] = new int[x];
103         double sum = 0;
            double s = 0;
105         double sumSquared = 0;
            double seriesArray[][] = new double[n][n];
107

109         subArrayFill(subArray);
            arrayFill(seriesArray);
111         ////////////////////////////////////////////////////////////////////
            findSeries(subArray, seriesArray);
113         /////////CHANGE TO findSeries1(subArray, seriesArray); for Q5///////

115


117         for (int i = 0; i < reps; i++) {

119             long t1 = System.nanoTime();
                findSeries(subArray, seriesArray);
121             long t2 = System.nanoTime() - t1;

123             sum += (double) t2 / 1000000.0;
                sumSquared += (t2 / 1000000.0) * (t2 / 1000000.0);
```

```
125             }
                double mean = sum / reps;
127             double variance = sumSquared / reps - (mean * mean);
                double stdDev = Math.sqrt(variance);
129             DecimalFormat df = new DecimalFormat("#.####");
                System.out.println(df.format(mean));
131
        }
133
        public static void main(String[] args) {
135
            for (int i = 100; i <= 6400; i += 100) {
137             timingExperiment(3, i, 100);
            }
139


141
        }
143 }
```