CSC2040 Data Structures and Algorithms, and Programming Languages

Practical 9

Wednesday 24 January 2018

Introduction to writing, timing and analysing algorithms

Create a new empty Project called Practical9 on the C:\ drive or on your own drive.

1. Timing parts of your code

To time algorithms in this practical, we will have to slow them down. This is because we can only time to the nearest millisecond, and most of the programs here will run in rather less than a millisecond! We will slow the programs down by including a Sleep operation in the innermost loop. Therefor to time programs you need to do several things:

a) First, you will need to include the library <windows.h>, to enable you to pause your program for one time unit by calling (in the innermost loop):

```
Sleep(1);
```

b) Second, to measure and print the actual time it takes to execute some code, you need to read the system clock at the start, read it at the end, and subtract the two, giving an answer in milliseconds. You can convert this to seconds by dividing by 1000. For example:

```
clock_t  begin = clock();

<<… your code to be timed goes here…>>

clock_t  end = clock();

double elapsed = double(end - begin);
cout << "Time taken with N = " << N << " is " << elapsed <<  “ms = "
<< elapsed / 1000.0 << " s" << endl;
```

Note: you will have to include the library <time.h>

Exercise: Write a function testTiming() which executes and times “Sleep(1)” 1000 times.

2. Timing an array processing loop.

First, write a piece of code which reads a value N from the keyboard, and then creates a vector of integers of this size. Now initialise the vector to contain N random numbers, of range 0 ~ 100000. Test this bit first before going any further. You might want to print out the array’s values. Try typing in different values of N each time you run it.

Now that you have your array set up, write a function, taking the array and N as parameters, to **find the average** of the numbers in the array (the result will be of type double). Check that your answer is correct for a small value of N.

Note: inside your loop, make sure to include a Sleep(1) operation, to slow it down.

Finally, time your function to find the average. Do this for different values of N (say, 100, 200, 300, up to 1000 or 2000). Plot the times as a graph below. Is this what you expected?

3. How many above average?

Write another function to find out how many values in the above array are above average. Before you run it, predict the shape of the graph you expect to get.

4. Binary search of a sorted array

In our lecture, we introduced a method of constructing a balanced binary search tree (BST) from a sorted array to perform binary search. Now you are asked to write a function to perform binary search directly on a sorted array.

a) First, create a vector of size N of random integers. You can use the same arrays as above for this test.

b) Now, sort the vector using the algorithm provided in the library <algorithm>.

c) Write a function

int **binarySearch** (vector<int>& myArray, int first, int last, int value);

which takes the sorted array, and searches the part of the array between first and last for the given *value*. If it is found, return the position in the array. Otherwise return -1. Use a **binary search** to do it as quickly as possible.

For timing you **binarySearch** function, remember to call Sleep(1) every time you do a three-way comparison. You can use recursion if you want.

Run your program for different values of N. Plot the times as a graph below.

Questions: Is this what you expected? What is the time proportional to (in terms of N)? What is the complexity of your algorithm? O ( ) Why?

5. Finding the value which is furthest from any other value

As above, but create an unsorted vector of size N of random integers. Once you have set up your array, the problem is this:

> *Find which number in the array is the most remote: i.e. it is furthest from any other number in the array. Think of the numbers as being islands spaced out.*

For example, if your array contained:

12 20 6 31 40 1 42 19

then the answer should be 31, because the closest number to 31 is 9 away (40). The closest to any of the other numbers is less than this.

Once you have it working (and tested for a small value of N), put a 'Sleep(1)' inside the innermost loop (or inside each loop, if you have separate non-nested loops). Time your algorithm for a range of input values of N and arrays. Again, plot these as a graph below.

Questions: Is this what you expected? What is the time proportional to (in terms of N)? What is the complexity of your algorithm? O ( ) Why?