CSC2059 Data Structures and Algorithms

Practical 7: Introduction to writing, timing and analysing algorithms

Wednesday 24th November 2021

Launch Visual Studio. Create a new Solution Practical7 and name the first project Time1. In project Time1 create a source file test1.cpp.

Project 1. Timing array processing loops

(1) The method used to time 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 include <time.h> and 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;
```

Exercise: in test1.cpp write a function `testTiming(int N)` which executes and times "Sleep(1)" N times. Call the function in the main function with variable N values read from keyboard.

(2) Timing an array processing loop

First, reads a value N from the keyboard, and then creates an array of integers of this size. Now initialise the array 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, `double findAverage(int* pi, int N)`, taking the array (pointed to by pi) 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 to calculate the average, 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?

What is the complexity of the findAverage operation, O(   ), in terms of N?

(3) How many above average

Write another function, `int find_countAverage(int* pi, int N),` that first finds the average of an array, and then finds out how many values in the array are above average (it returns the count). Again, in each loop make sure to include a Sleep(1) operation to slow it down. Before you run it, predict the shape of the graph you expect to get.

What is the complexity of the find_countAverage operation, O(   ), in terms of N?

(4) Finding the value which is furthest from any other value

For the same array, write a function, `int findFurthest(int* pi, int N),` which finds 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. 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?

Project 2. Timing linked list traversals

In Solution Practical7 add a new project Time2. In the project, create a test program test2.cpp, and include the provided ListNode.h and List.h classes. In the provided List.h, we have added two Sleep(1) operations for the convenience of the following timing experiment. The experiment conducts a complexity analysis of the two linked list operations, discussed in Practical 5, Project 3, Part (b).

In test2.cpp, first create a linked list (an object of the List class) of N random numbers, in the range 0 ~ 100, with N being taken from keyboard.

Then, write a function, count1, taking the list and an integer key as parameters, to find the number of occurrences of the key number in the list. The function uses get(int p) to cycle through the list to access the number at each positon p for comparing with the key (see Part 2, Page 5, Practical 5).

Next, write another function, count2, which accomplishes the same task as above but uses set_first and then get_next to cycle through the list to compare each number in the list against the key (see Part 3, Page 5, Practical 5).

Finally, time the two functions for a range of input values of N (say, 10, 20, 30, up to 100 or 200). Plot both functions' times in the same graph below.

Questions: For each of the functions, count1, count2,

- What is the time proportional to (in terms of N)?
- What is the complexity, O ( )? Why?

If you have any questions, ask a demonstrator or a teaching staff face to face in the labs, or online by using MS Teams, during the timetabled practical session time. At any other time outside the timetabled session time, you can send your questions by using the Ticketing System, or by using emails.