CS3052 — Computational Complexity

Practical 1 : Algorithms

Steve Linton sl4@st-andrews.ac.uk

Due: 2021-03-09 (Tue Week 7) 21:00

Key Points

This is **Practical 1**, in which we ask you to implement a number of variants of an algorithm for finding the closest pair of points in two-dimensional space and compare their running times to theoretical complexity estimates

Due date: Tue Week 7 (9 Mar 2021) (MMS is authoritative)

Credit: This practical is worth 45% of the coursework of the module.

Submission: Upload a zipfile with your code, your report in PDF and any supplementary information (data, spreadsheets, etc.) to MMS.

Important Note: The specification of this practical is quite long and quite precise, to allow for automated testing. It is *essential* that your code comply exactly with the specification below and that `stacscheck` is able to build and test your programs on the School systems. Make sure that you have read and understood the full specification before you start work. Submissions that do not meet these conditions will receive a reduced mark.

Purpose

This practical will help practice and develop your skills in:

- implementing divide and conquer algorithms
- implementing geometric algorithms
- designing and running experiments and measuring the run-time of algorithms

Overview and Background

Given an array of points (each given by X and Y coordinates), a fundamental geometrical computation is to find the two of them whose separation is smallest. A "divide-and-conquer" algorithm for this problem was given in lectures and shown to have worst-case running time $O(n(\log n)^2)$ on n points.

In a further lecture some hints were given for improvements, including developing a version with $O(n \log n)$ worst-case running time.

In this practical you will implement a number of versions of this algorithm and conduct experiments to measure the practical running time of your implementation and compare it with theoretical bounds.

Basic Practical Specification

DB in reading this section make sure that you understand the difference between o, O, Θ and ω asymptotic statements.

Task 1 – First implementation

Implement the closest-pairs algorithm *as described in lecture 2 of week 3*, including using `QuickSelect` (which you will also need to implement) to find the initial median. Your implementation should comply with the *program specification* below, to allow for stacscheck checking. You may use any reasonable pivot selection strategy in QuickSelect.

A mark of up to 10 is available for any program solution to this task which is correct, uses the algorithm from the lecture and is completes the stacscheck tests within the timeout.

Task 2 – Analysis

Conduct and report on experiments to assess the asymptotic complexity of your implementation.

A mark of up to 13 is available for experiments which convincingly show your implementation to run in $o(n^2)$ (worst-case) time.

A mark of up to 16 is available for experiments and analysis of experimental results showing clearly that complexity is $\omega(n)$ and consistent with being $\Theta(n(\log n)^2)$.

Task 3 – Further implementations

Implement the two improvements to the algorithm described in Week 3 lecture 3 – to remove the call to QuickSelect and to avoid the need to sort by y at each recursive step. Conduct and report experiments to assess the impact of these two changes.

A mark of up to 18 is available for a good solution to tasks 1–3.

Extension Task 4 – Alternative algorithms

Using the sources referenced in lectures, implement a solution to the problem with $o(n \log n)$ (expected) complexity (either the Fortune-Hopcroft algorithm or one of the randomized algorithms). Conduct and report experiments to assess the performance of your implementation.

This is comparatively difficult, and the sources are old-fashioned and may take some work to understand. The Fortune-Hopcroft paper also only considers the one-dimensional case in detail, so you would need to adjust the algorithm accordingly. You are *strongly* advised not to attempt this task until you have a solution to tasks 1–3 that you are happy with.

Detailed Specification of Program

Your programs should be named `cp1` for task 1, `cp3` for task 3 and `cp4` for task 4. Case is significant and they should be UNIX executables or scripts with no file extensions.

Each program should read from standard input a file with one positive integer N in decimal on the first line (N may be between 2 and $2^{31} - 1$) and then N further lines, each with two floating point values on them (as produced by the C `printf` library function with format string `%lg` and separated by a space). The floating point values will be in the range of standard IEEE double precision.

The program should print to standard output the distance between the two closest points with nine significant figures of accuracy (as it would be printed by the C `printf` function with format string `%.9lg`).

You may, of course, write other programs, or extend the functionality of these programs to gather timing data for your analysis, but when run from the command-line with no arguments they should behave as described.

Additional Details

1. You may use any programming language, provided that it can be run on the school systems, and I can find a specification of it so that I can read your programs.
2. If the input is incorrectly formatted, your program should print "Incorrectly formatted input" followed by a newline to standard output, and exit.
3. Your programs must be contained in a directory called `p01-src` at the top level of your submission. Try not to have any other directories with this name in your submission.
4. Calling `make` in the `p01-src` directory with no arguments must build your programs, resulting in executable files or scripts `cp1`,`cp3` etc. in that same directory.

5. You may use library functions (e.g. for sorting), library data structures or data structures built into your language, but you need to make sure that they do not interfere with the asymptotic complexity of the algorithm.
6. Your programs must not impose arbitrary limits on the size of the problems handled, except for available CPU time and memory.
7. If your stacscheck output differs from what is expected *only* because of *small* differences in floating-point rounding (e.g. differs by 1 in the ninth significant figure) you will not be penalized.

A note on packaging Java or Python programs

Neither Java nor python can normally be compiled to produce standard UNIX executables. For Java, a simple solution is to write a shell script to take the place of the executable. There are many shell tutorials available on line. It will be something like

```
#!/bin/bash
java -cp . cp1
```

although you may need to give a more complex class path and/or specify a package.
The relevant part of the `Makefile` can be something like

```
all: cp1.class cp3.class cp4.class

cp1.class: cp1.java utilities.java
      javac cp1.java utilities.java
```

You should replace `utilities.java` by whatever class, or classes, you need. Watch out for the whitespace rules in makefiles.
For python it is probably enough to put

```
#!/bin/env python3
```

at the start of the Python program, make sure the file name has no extensions and then the Makefile will just need to make the files executable (see `chmod`).
If you are editing the files on Windows, watch out for carriage return characters on the `#!` lines which can cause problems.

Notes on Data Gathering

The design and implementation of your experiments is an important part of this practical, and should be explained in your report. A supporting lecture will be released offering some tips and advice on experimentally assessing the runtimes of programmes, but in brief:

- Make sure you are measuring the CPU time used by the algorithm and not something else (like time to read or create the data).
- As far as you can, make sure that your test data is likely to include the worst case inputs for your algorithm
- Make your experiments repeatable
- Think about an appropriate set of values for n.
- Think about what graphs to plot (or statistical tests to use, if you like, although this is not required) to assess the growth being shown by your data

You are strongly encouraged to automate the process of running and analysing your experimental results as far as possible, and to include that automation as part of your submission.

StacsCheck Tests

An extensive suite of `stacscheck` tests is provided in the `Tests` directory which will test that your programs produce correct results.

Submission

The output of this assignment will be a **report**, which should be in PDF, together with code as noted in the individual tasks. The report should *briefly* discuss your implementation, especially any interesting choices or decisions you had to make, but focus mostly on your experiments and conclusions.

Make a **zip archive** of all of the above, and submit via MMS by the deadline.

Marking

We remind you that it is not enough for your programs to be correct, you have to convince the markers that they are correct.

We are looking for:

- good, understandable code, tested and commented properly, with important design decisions explained in the report;
- good experimental design supported by appropriate automation, and careful and insightful analysis of results

The standard mark descriptors in the School Student Handbook will apply:
`https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors`

Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof):
`https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties`

Good Academic Practice

The University policy on Good Academic Practice applies:
`https://www.st-andrews.ac.uk/students/rules/academicpractice/`

References

Manber "Introduction to Algorithms A Creative Approach" section 8.5 (pp 278-281) describes the $O(n(\log n)^2)$ and $O(n \log n)$ algorithms

Fortune and Hopcroft's paper is at `https://ecommons.cornell.edu/bitstream/handle/1813/7460/78-340.pdf`

Khuller & Matias, "A Simple Randomized Sieve Algorithm for the Closest-Pair Problem" (sections 1 and 2) is a relatively approachable description of a randomized linear time algorithm. `https://www.cs.umd.edu/~samir/grant/cp.pdf`