Four-Sum

Thore Husfeldt and Riko Jacob

Wed Feb 14 10:40:32 2018 +0100, rev. eeof46b

Implement a very simple algorithm in two different languages, run some experiments. Improve the algorithm.

Description

Given a list of N (long) integers, detect if there are four of them that sum to 0.

Exhaustive search. The simplest solution is to use exhaustive search using four nested for loops. Go ahead and implement that in the language that you're comfortable in, Python or Java. There is a code skeleton called `sol/Simple.java` or `sol/simple.py` that you must use. The program expects a list of N integers on standard input, following the number N itself. The directory `data` contains several of such lists for various N.

The program must write `True` to standard output if it finds a solution, and `False` otherwise; your program may write other output to standard error (`stderr`). This is the basis of the tests on codeJudge, and all your programs must at least pass all of `TestsTiny` and `TestsSmall`.

Make it work; and determine the running time on your machine on the input files. There are a handful of inputs for each size. Report the fastest observed time, the slowest, and the average. (Try to be half-way serious about this— *e.g.*, restart the computer before the experiment, so as to make sure no spurious processes are running in the background, don't have a World of Warcraft-server or Bitcoin mining programme running at the same time.) You probably won't be able to handle the larger inputs—don't report experiments that take more than a few minutes. Plot the result using both a standard and a log-log plot.

Exhaustive search in a different language. Do the same as above, but in the other language. This may involve installing compiler or a run-time environment, figuring out where the semicolons or brackets go (or don't go) and other annoyances. Compare the running times of the two implementations of exhaustive search in the different languages. This is the only task you are to implement in both languages.

Faster. Use the idea of the fast three-sum algorithm in [SW, Chapter 1.4] `ThreeSumFast` to write a faster algorithm for the four-sum

problem. Your running time must be close to cubic time, proportional to $N^3 \lg N$. Call it `Faster.java` or `faster.py`. Redo the experiments and report them.

Optional There is a cute, simple, and very satisfying way to solve the problem in time proportional to $N^2 \lg N$ instead. If you can figure that trick out, implement it and hand in that code instead.

Tips and comments

Java types. The integers in the input files are very large, and will not fit into the Java datatypes `int` or `Integer`. Use `Long` instead. Do not worry about overflow.

Python dialect. You can use Python 2 (which may be already installed on your computer) or Python 3 (remember that the library of the course is written in python3).

Course libraries. In this exercise (and this exercise *only*) you are *not* required to use the course libraries, say for input/output. This is because it is too much overhead to ask students in language x to install the libraries for language y. Therefore, this particular exercise is very bare-bones. You are welcome to use the course library if you want to.

Measuring time. On a unix system, the `time` command gives sufficiently good performance estimates, and is better than sitting with a stopwatch. Both Java and Python come with better solutions; the course libraries offer a `Stopwatch` class, and Python has a handy `time` module. You are free to use either. On some modern Java systems, the compiler performs very funky optimisations behind your back, giving wildy divergent measurements for the same code on the same machine. You can handle that by running time code a few times before measuring it. It's a black art.

Experiment design. If you like doing repetitive work, you can run your experiments individually by starting dozens of processes from the command line (or, worse, an IDE) and writing the results into a spreadsheet to compute averages. However, such an approach quickly becomes both tedius and error-prone.

Instead, construct another program that performs all the experiments for you, and writes the results, including a time stamp of the experiment, into another file. There are many schools of thought about how to write that program; you can make it part of your original solution, or you can use a separate programme in a dedicated scripting language. One useful program (or `builtin` in `bash`) is `time`, a simple usage would be

```
time java Simple < ../data/ints-200-2.txt
```

Because of the setup on codeJudge, our programs have debugging output on `stderr`, and there is the need to ignore this when collecting experimental results into a file. The following is an example how this could look like (note that the trailing \ continues the command on the next line):

```
#!/bin/bash

nn="100 200 400 800"
javac Weed.java
if [ ! -d Input ]
then
    mkdir Input
    for n in $nn
    do
        java Weed $n 1 > Input/Weed1_$n.in
    done
fi
rm simple.table
for n in $nn
do
    /usr/bin/time -f "$n %e" bash -c \
                  "java -cp javaSol Simple <Input/Weed1_$n.in > /dev/null 2>&1" \
                  >> simple.table  2>&1
done
```

More advanced solutions (maybe in other languages) could compute averages, or even draw the plots for you. One example (that turned out to be somewhat more complicated) can be found in `../src/experiment.py`.

Learning outcomes

Besides some important secondary skills (input/output, exposing yourself to an unfamiliar programming language, basic experimentation, drawing graphs, computing averages), the main points of this exercise are

- It does *not* make a big difference which language you use.
- It *does* make a big difference which algorithm you use.
- The slowest observed performance is well characterised by the predicted worst-case running time.

- The fastest observed performance contains very little information about general behaviour.