COMP301
Data Structures & Algorithms [C++]

Dr. N. B. Gyan

Central University, Miotso. Ghana

Algorithm Analysis

Algorithm Analysis

- When we run a program on large amounts of input, we must be certain that the program terminates within a reasonable amount of time.
- The amount of running time is almost always independent of the programming language or even the methodology we use (such as procedural versus object-oriented).
- An algorithm is a clearly specified set of instructions a computer follows to solve a problem.

Algorithm Analysis

- Once an algorithm is given for a problem and determined to be correct, the next step is to determine the amount of resources, such as time and space, that the algorithm will require. This step is called algorithm analysis.
- An algorithm that requires several gigabytes of main memory is not useful for most current machines, even if it is completely correct.

Why Algorithm Analysis?

Knowing how to analyse algorithms helps one to know

- how to estimate the time required for an algorithm,
- how to use techniques that drastically reduce the running time of an algorithm,
- how to use a mathematical framework that more rigorously describes the running time of an algorithm.

What Is Algorithm Analysis?

- The amount of time that any algorithm takes to run almost always depends on the amount of input that it must process.
- It is expected, for instance, that sorting 10,000 elements requires more time than sorting 10 elements.

What Is Algorithm Analysis?

- The running time of an algorithm is thus a function of the input size.
- The exact value of the function depends on many factors, such as ...(what do you think?)...the speed of the host machine, the quality of the compiler, and in some cases, the quality of the program.
- For a given program on a given computer, a plot can be made of the running time function on a graph as the following.

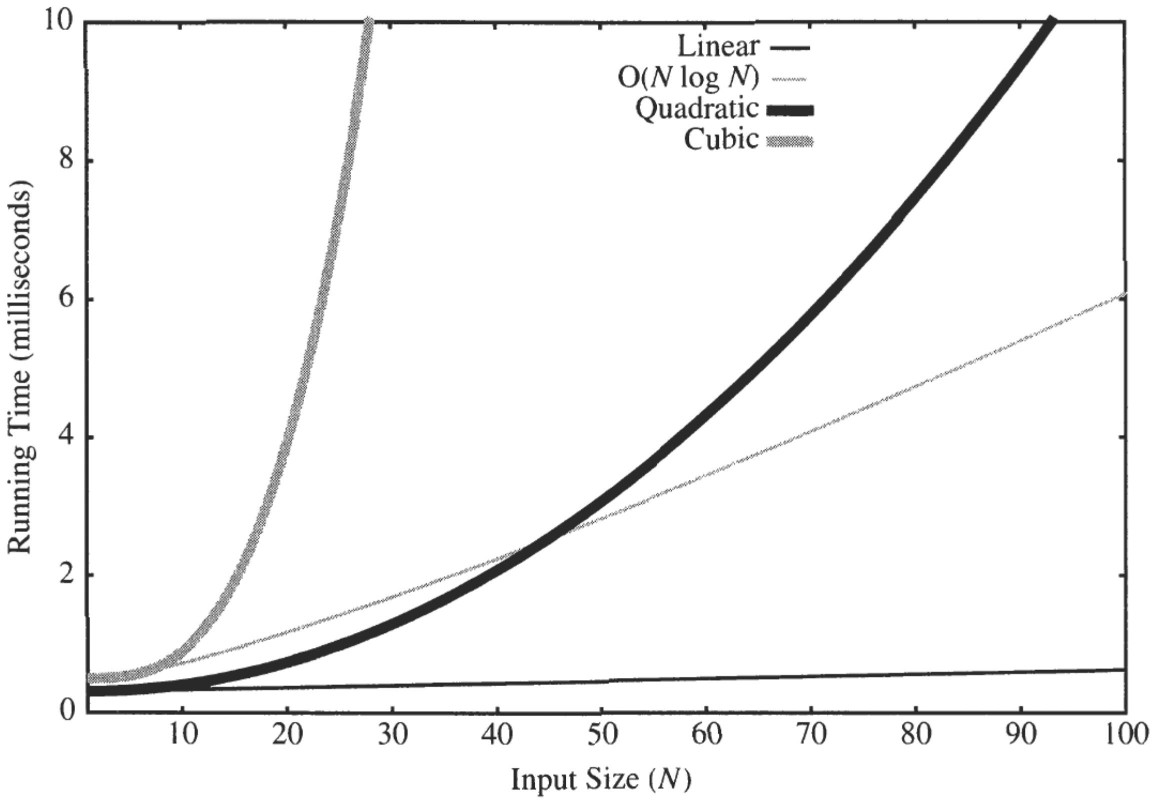

Fig. 3: Different types of algorithm running times

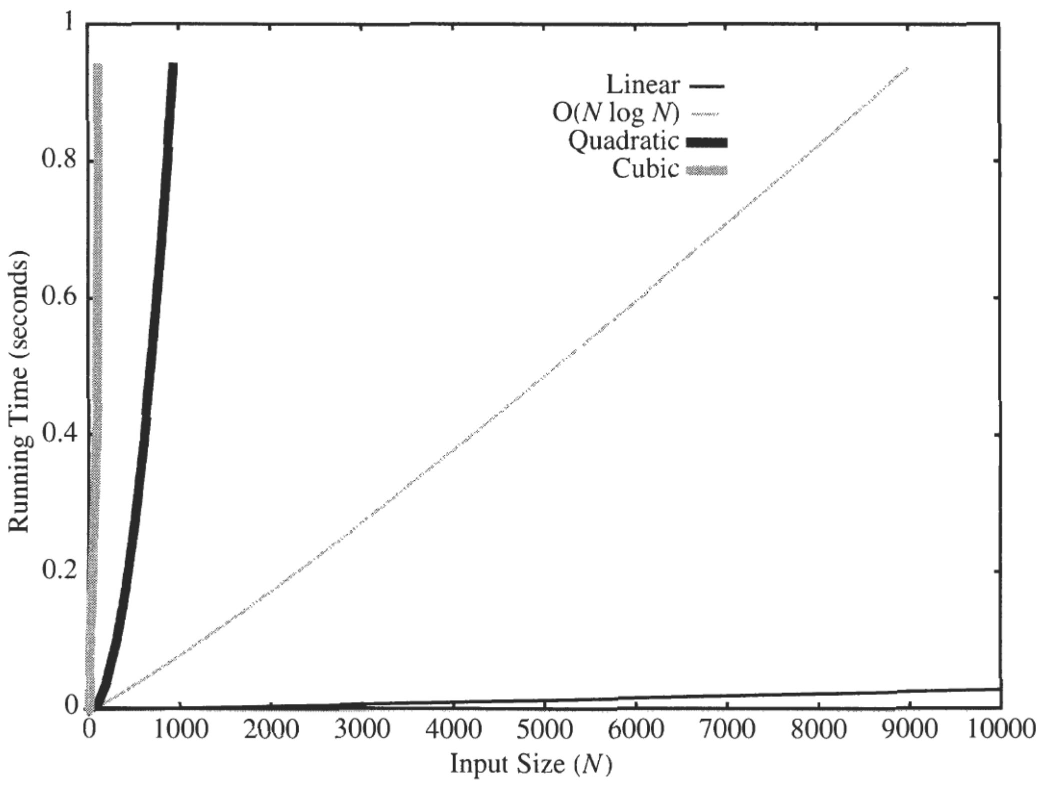

Fig. 4: Different types of algorithm running times (increased *N*)

What Is Algorithm Analysis?

- In Fig. 3 the input size **N** ranges from 1 to 100 items, and the running times range from 0 to 10 milliseconds.
- A quick glance at Fig. 3 and its companion, Fig. 4, suggests that the **linear**, $O(Nlog\ N)$, **quadratic**, and **cubic** curves represent running times in order of decreasing preference.

What Is Algorithm Analysis?

- An example is the problem of downloading a file from the Internet. Suppose that there is an initial $2sec$ delay (to set up a connection), after which the download proceeds at $1.6K/sec$.
- Then if the file is `N` kilobytes, the time to download is described by the formula, $T(N) = N/1.6 + 2$. This is a *linear function*.
- This essentially means that downloading an $80K$ file takes approximately 52 *sec*, whereas downloading a file twice as large ($160K$) takes about 102 *sec*, or roughly twice as long.

What Is Algorithm Analysis?

- This property, in which time essentially is directly proportional to the amount of input, is the signature of a linear algorithm, which is the most efficient algorithm.
- In contrast, as these two graphs above show, some of the nonlinear algorithms lead to large running times. For instance, the linear algorithm is much more efficient than the cubic algorithm.

Benchmark Analysis

Benchmark Analysis?

- At this point, it is important to think more about what we really mean by *computing resources*.
- There are two different ways to look at this: One way is to consider the amount of space or memory an algorithm requires to solve the problem. The amount of space required by a problem solution is typically dictated by the problem instance itself.
- As an alternative to space requirements, we can analyze and compare algorithms based on the amount of time they require to execute. This measure is sometimes referred to as the “execution time” or “running time” of the algorithm.

Benchmark Analysis

- Let's start with calculating the sum a range of integers with a loop. (can you design such a program in `C++`?)
- One way we can measure the execution time for the function `sumIntsOne()` is to do a **benchmark analysis**.
- This means that we will track the actual time required for the program to compute its result.

Benchmark Analysis

- We can benchmark a function by noting the starting time and ending time with respect to the system we are using.
- In `C++`, the `<ctime>` header can be employed. This is shown in `sumIntsOne.cpp` below.

`sumIntsOne.cpp`

Benchmark Analysis

- For $n = 10,000$ the time required for each run, although longer, is very consistent, averaging about 10 times more seconds.
- For n equal to $1,000,000$ the average again turns out to be about 10 times the previous.
- Now consider the following code, which shows a different means of solving the summation problem, `sumIntsTwo()`. iterating.

Benchmark Analysis

This function, `sumIntsTwo( )`, takes advantage of a closed equation:

$$\sum_{i=0}^{n} i = \frac{(n)(n+1)}{2}$$

to compute the sum of the first n integers *without* iterating.

`sumIntsTwo.cpp`

Benchmark Analysis

- There are two important things to notice about this output.
- First, the times recorded above are shorter than any of the previous examples.
- Second, they are very consistent no matter what the value of *n*. It appears that `sumIntsTwo()` is hardly impacted by the number of integers being added.
- The benchmarking really tells us, intuitively, that the iterative solutions seem to be doing more work since some program steps are being repeated.

Benchmark Analysis

- This is likely the reason it is taking longer. Also, the time required for the iterative solution seems to increase as we increase the value of *n*.
- However, there is a problem. If we ran the same function on a different computer or used a different programming language, we would likely get different results.
- It could take even longer to perform `sumIntsTwo()` if the computer were older.

Challenges of Experimental Analysis

Although valuable, there are three major limitations to their use for algorithm analysis:

- Experimental running times of two algorithms are difficult to directly compare unless the experiments are performed in the same hardware and software environments.
- Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important).

Challenges of Experimental Analysis

- An algorithm must be fully implemented in order to execute it to study its running time experimentally.

  (This last requirement is the most serious drawback to the use of experimental studies. At early stages of design, when considering a choice of data structures or algorithms, it would be foolish to spend a significant amount of time implementing an approach that could easily be deemed inferior by a higher-level analysis.)

Big ‘O’ Notation for Algorithm Analysis

- Thus, we need a better way to characterize these algorithms with respect to execution time.
- The benchmark technique (Experimental Analysis) computes the actual time to execute. It does not really provide us with a useful measurement, because it is dependent on a particular machine, program, time of day, compiler, and programming language.

Big ‘O’ Notation for Algorithm Analysis

- Instead, we would like to have a characterization that is independent of the program or computer being used.
- This measure would then be useful for judging the algorithm alone and could be used to compare algorithms across implementations.
- Thus, the **Big ‘O’ Notation** for algorithm analysis.

Exercise

Go online and read about the Big ‘O’ Notation.

See you after the break, God willing 🙏🏿