Side Talk: Cost of Your Programs

Yufei Tao

ITEE
University of Queensland
In the class, we have defined the RAM computation model. In turn, this allowed us to define rigorously algorithms and their cost.

In reality, we do not write programs under the RAM model—instead, we do so in a programming language like C++, JAVA, Python, etc. Thus, how would you measure the cost of your programs? Also, what are the connections to the RAM model?

This side talk will clarify these issues.
Measuring the Cost of Your Program

Rule of Thumb:

Count the number of statements executed.

In other words, to make your program fast, minimize the number of statements that need to be run.
int i = 0;
int n = 100;
int sum = 0;
for (i = 1; i <= n; i ++) {
    sum = sum + i;
}
return sum;

Number of statements executed = $2n + 5$ (in this example, $n = 100$). This is an accurate indication of the cost of your algorithm in the RAM model, up to only a constant factor.

The next slide explains why.
When your program is executed, the computer (the real one where your program is run) assigns a memory cell for each variable. The figure above shows an example—note that the 3 memory cells can be anywhere, and in any order.
A statement like `i = 0` gets automatically translated into a sequence of atomic operations in the RAM model, e.g.:

- set Register 1 to value 0
- set Register 2 to value the address of `i`
- write the contents of Register 1 into memory cell `i`

In this example, the statement `i = 0` entails a cost of 3 in the RAM model.
As another example, statement $\text{sum} = \text{sum} + i$ may get translated into 6 atomic operations:

- set Register 1 to the address of $i$
- load $i$ into Register 2
- set Register 1 to the address of $\text{sum}$
- load $\text{sum}$ into Register 3
- Register 3 $\leftarrow$ Register 2 + Register 3
- write the contents of Register 3 into memory cell $\text{sum}$
Who Does the Translation?

The compiler.

When you are coding, you are writing in a so-called “high level language” where each statement encapsulates multiple atomic operations, so that you do not need to worry about the low-level details such as registers.

When you are done, the compiler does the dirty work by transforming your code into a form that can actually be executed—perhaps you have heard about the name of that “form”: the assembly language.
Take-Away Message 1

The number of statements executed in your program can be regarded as the running time of your program—it differs from the RAM-model cost by at most a constant. For example, the constant is 6, if every statement gets translated to up to 6 atomic operations.
In computer science, we **seldom** care about constants in analyzing the cost of an algorithm. In other words, as far as theory is concerned, a $2n$-statement algorithm is considered **equally fast** as a $10n$-statement algorithm. Although more compelling reasons will be explained later, from this side talk, can you feel why trying hard to optimize the constants may not always pay off?

This is because you do not know how the compiler will do the translation. Actually, this can differ from language to language (C++ vs. Java), version to version (compiler 1.0 vs. 2.0), platform to platform (Windows vs. Linux), and CPU to CPU (Intel vs. AMD). It is possible that one compiler may convert $\text{sum} = \text{sum} + i$ into 6 operations, while another may do so with only 3. Thus, although your program has only $2n$ statements, there is no guarantee that it will have fewer atomic operations than another $10n$-statement program.