CS 240 — Lecture 1

Bartosz Antczak *Instructor: Eric Schost* *January 3, 2017*

1.1 Course Info

The objective of this course is to study efficient methods of storing, accessing, and performing operations on large collections of data. We'll consider various abstract data types (ADTs) and how to implement them efficiently using appropriate data structures.
Algorithms are presented using pseudocode and analyzed using order notation (Big-O notation, $O(n)$). This course is considered more theoretical and focuses on mathematical analysis.

1.2 Initial Dive

1.2.1 Problems (Terminology)

Problem Instance: A particular input for a specified problem
Problem solution: Output (correct answer) for the specified problem instance
Size of problem instance: $Size(I)$ is a positive integer which measures the size of the instance I

Example 1.2.1. *A sorting problem defined with our terminology:*

- *Instance: 14, 3, 1, 10, 20, 5*
- *Solution: 1, 3, 5, 10, 14, 20*
- *Size: number of integers = 6*

1.2.2 Algorithms and Programs

An **algorithm** is a step-by-step process for carrying out a series of computations. By definition, an algorithm successfully solves a problem if, for every instance I of problem Π, A finds (computes) a valid solution for the instance of I in finite time.
For a problem Π, we can have several algorithms.
A **program** a program is an implementation of an algorithm using a specified computer language.
When creating algorithms, we'll be most concerned with the amount of time a program takes to run (running time) and the amount of memory it uses (space). The previous two factors will be determined by $Size(I)$.
There are a few methods of measuring the runtime of a program:

- *Experimental:* Implement it and use a method such as `clock()` to accurately measure the actual running time
- *Theoretical:* A measure of a program's runtime regardless of hardware capabilities. We count the number of primitive operations in an algorithm rather than the actual time itself. This is called **order notation**

Example using big-O notation

Multiplying two matrices a and b into matrix c takes $O(n^3)$ time:

```
for i = 1 ... n
  for j = 1 .. n
  c[i][j] = 0; // n^2 assignments
    for k = 1 .. n
    c[i][j] = c[i][j] + a[i][k] * b[k][j]; // n^3 additions and multiplications
```

1.2.3 Order notation

Disclaimer: $f(n)$ in this course will represent the runtime of a program F.

- O-notation: $f(x) \in O(g(n))$ if $\exists\, c > 0$ and $n_0 > 0$ such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n > n_0$.
  In other words, if $f(x) \in O(g(n))$, then f has a runtime that is less than or equal to $g(n)$
- Ω-notation: $f(x) \in \Omega(g(n))$ if $\exists\, c > 0$ and $n_0 > 0$ such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n > n_0$.
  In other words, if $f(x) \in \Omega(g(n))$, then f has a runtime that is at least $g(n)$
- θ-notation: $f(x) \in \Omega(g(n))$ if c_1, $c_2 > 0$ and $n_0 > 0$ such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n > n_0$.
- o-notation: similar to O-notation, except that $f(n) < c \cdot g(n)$ for **all** constants $c > 0$
- ω-notation: similar to Ω-notation, except that $c \cdot g(n) < f(n)$ for **all** constants $c > 0$

Example 1.2.2. *Given $f(n) = 2n^3 + n^2$ and $g(n) = n^3$, prove that $f(n) \in O(g(n))$.*
Proof: *We see that for all $n \geq 1$*

$$n^2 \leq n^3 \tag{1.1}$$
$$2n^3 \leq 2n^3 \tag{1.2}$$
$$2n^3 + n^2 \leq 3n^3 \tag{1.3}$$
$$f(n) \leq g(n) \tag{1.4}$$
$$f(n) \in O(g(n)) \qquad \text{(By definition)} \tag{1.5}$$