CS 341 — Lecture 2

Bartosz Antczak *Instructor: Bin Ma* *September 12, 2017*

2.1 Model of Computation

We define our *model of computation* as our "computer" that we will run our algorithms on. Some characteristics of this computer that we need to factor in are:

- Multiplication vs. addition (we won't focus on this too much)
- Random vs. sequential access of memory (we will assume to use random access)
- Time difference between accessing memory from disk and main memory (RAM). This is only important when not all the data can fit in RAM
- Time difference between accessing memory from cache and RAM (this is too detailed)
- For-loop vs. a single arithmetic operation (preference on single operation)
- Random number generator (randomized algorithms run faster usually than deterministic ones. We will however focus only on deterministic algorithms in this course)
- Word size: 32-bit vs. 64-bit. It's not a significant difference in terms of algorithm design, but there's more to it than that as this example outlines:

  Example 2.1.1. *Fibonacci numbers*

  A simple algorithm that prints the first n Fibonacci numbers is shown:

```
i = 0, j = 0
for k from 1 to n
    j = i + j
    i = j - 1
```

  The problem here is integer size. At $n = 47$, `j` will take up a size larger than 32 bits. **The word size determines how large of a number we can work with in our algorithm.**

2.1.1 Word-RAM model

A solution to this integer-size problem is the word-RAM model:

- Each memory cell is a word that can hold an integer
- Random access of memory. Accessing a word at the location specified by the value of another word takes constant time
- Basic operations (arithmetic, shifting, logical, comparison) on words take constant time
- All integer values can be placed in one word

2.2 Asymptotic Time Complexity

When we define a $T(n)$ vs. n graph, we assume the relation to be smooth, but that's not always the case.

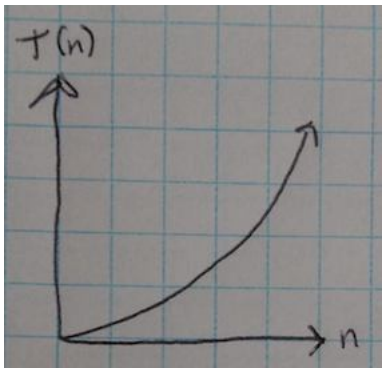

Figure 2.1: A smooth graph

The run time for a particular size of n may vary based on difference kinds of instances. Really, the graph looks like this:

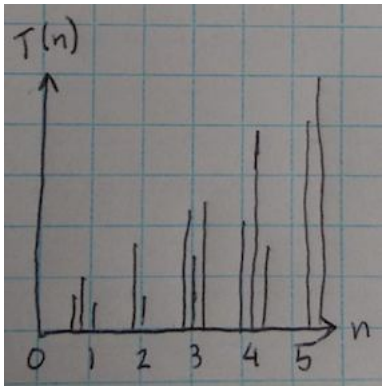

Figure 2.2: A rough graph

Since it's pretty complex, how to we define $T(n)$?

1. We can let $T(n)$ be the *average analysis* of all instances of the input
2. We can also look at the *expected time*, but that's only for randomized algorithms which are not outlined in this course

In this course however, we'll focus on the **worst-case** analysis. This is defined as the *upper-bound* for every instance of the same size.
Now let's define a simple function to describe the trend of the run-time growth based on input size:

2.2.1 Big O

$$T(n) = O(f(n)) \iff \exists\, (c > 0, n_0 > 0) \text{ s.t. } \forall\, n > n_0,\ \ T(n) < c \cdot f(n) \iff \lim_{n \to \infty} \frac{T(n)}{f(n)} \leq c$$

Example 2.2.1. *Runtime functions*

- $T(n) = 5n^2 + 3n + 5 = O(n^2)$
- $T(n) = 10^{100} \cdot n = O(n)$
- $T(n) = \log n = O(n)$, in fact $\forall b\ (\log n) = O(n^b)$

What about $T(n) = (n+1)! = O(n!)$? Is this true? No! The relation is not bounded by a constant:

$$\frac{(n+1)!}{n!} = n$$

Note that n is NOT a constant.

Other Rules

- $f(n) + g(n) = O(\max(f(n), g(n)))$
- $f(n) = O(g(n)),\ g(n) = O(h(n)) \implies f(n) = O(h(n))$

Other Notations

- $T(n) = o(f(n)) \iff \lim_{n\to\infty} \frac{T(n)}{f(n)} = 0$
- $T(n) = \Theta(f(n)) \iff T(n) = O(f(n))$ AND $f(n) = O(T(n))$
- $T(n) = \Omega(f(n)) \iff f(n) = O(T(n))$
- $T(n) = \omega(f(n)) \iff f(n) = o(T(n))$

Sometimes these functions take multiple parameters.

Example 2.2.2. *Graph Theory*

A graph with n vertices and m edges will have a time complexity of $O(m+n)$. The worst case if $O(n^2)$.

Example 2.2.3. *Convex Hull Problem*

Consider a graph where we define the vertices as points on a 2-D plane, and we define an edge as a line connecting two points such that all points sit on only one side of the line. We define a **convex set** as a set of points such that for any two points, the line connecting them intersects only points also found in the set. The problem focuses on finding all of the points that form the edges of the set:

- **Input:** $P_1 = (x, y), \cdots, P_n = (x_n, y_n)$
- **Output:** all of the points that lie on edge

A trivial algorithm (runtime of $O(n^3)$):

```
for each pair of p, q
    check if (p, q) is an edge
```

A faster algorithm is

```
Find one edge (p, q)
output p
while (q != p)
    find r such that (q, r) is the next edge
    output q
    r = q
```

This algorithm has a runtime of $O(n^2)$
Challenge: find a $O(n \log n)$ algorithm for this problem.