CSD101: Introduction to computing and programming (ICP)

Time complexity of algorithms

- We have seen different ways to sort, for example: bubble sort, insertion sort, selection sort, quick sort, merge sort.
- Which is the fastest? Or more generally, how do we estimate the time complexity of algorithms?
- Some observations:
  - The exact time taken clearly depends on the length of the input sequence. More generally length of the input.
  - It also depends on the state of the initial unsorted array.
- To be able to compare the time complexity of algorithms we estimate the time complexity in the **worst case**.
- Also, we measure complexity in terms of the **order** of the fastest growing term and ignore the slower growing and constant terms/factors.
- This way of measuring time complexity is called **big-O** complexity. We will symbolize it by $\mathcal{O}()$. It is called the Landau notation.

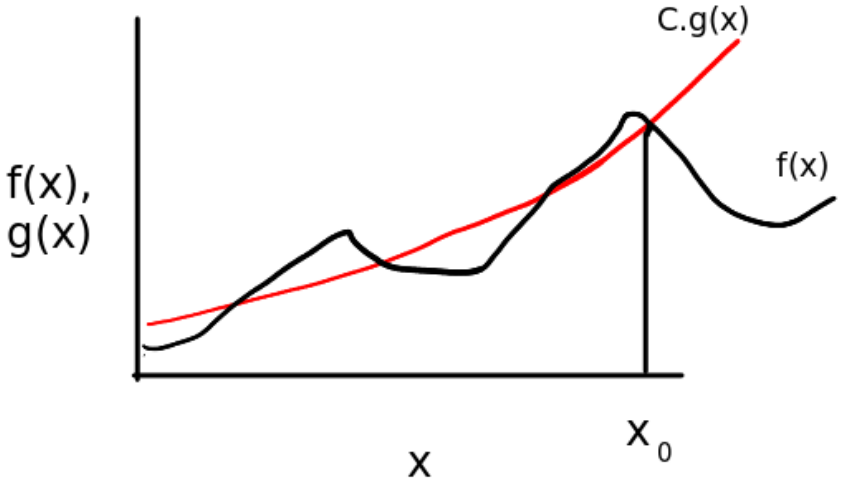
Definition of $\mathcal{O}()$ complexity

Definition 5

Let $f(x)$, $g(x)$ be functions defined on some subset of the real line. Then function $f(x) = \mathcal{O}((g(x))$ if and only if there exist constants C and x_0 such that for all $x > x_0$, $|f(x)| \leq C|g(x)|$.

That is beyond x_0, $Cg(x)$ always dominates $f(x)$. In CS $f(x), g(x) > 0$.

`Note that '=' symbol has a different meaning here.`



This is for large values of x. That is $x \to \infty$. Called asymptotic complexity.

Commonly used $\mathcal{O}()$ functions

Function	**Name**
$\mathcal{O}(1)$	Constant - does not depend on input size
$\mathcal{O}(\log(n))$	Logarithmic
$\mathcal{O}((log(n))^c)$	Poly-logarithmic
$\mathcal{O}(n)$	Linear
$\mathcal{O}(n^2)$	Quadratic
$\mathcal{O}(n^c)$	Polynomial
$\mathcal{O}(c^n)$	Exponential ($c > 1$)

c is a constant and n is size of input. The above are in increasing order.

Examples

- Expression evaluation will typically take constant time (if it does not involve a function call).
- Finding the maximum/minimum element in a sorted sequence of size n can be done in constant or $\mathcal{O}(1)$ time. This is a better example for a constant time algorithm.
- Searching whether an element exists in a sorted sequence of size n can be done in $\mathcal{O}(log_2(n))$ time.
- Finding the maximum or minimum element in a sequence of size n can be done in $\mathcal{O}(\mathrm{n})$ time.
- Bubblesort, selection sort, quicksort take $\mathcal{O}(n^2)$ time.
- Calculating the matrix product $C = A \times B$ where A, B, C are $n \times n$ matrices using the standard formula takes $\mathcal{O}(n^3)$ time.
- Evaluating the nth fibonacci number using the recursive definition will take $\mathcal{O}(2^n)$ time.

How fast functions grow. Intractability.

n	$log_{10}(n)$	n^2	n^4	2^n
10	1	10^2	10^4	$\sim 10^3$
10^2	2	10^4	10^8	$\sim 10^{30}$
10^4	4	10^8	10^{16}	$\sim 10^{3000}$

Already at $n = 100$, 2^n is intractable. So, an algorithm with exponential time complexity is intractable for even relatively small values of n. Very high order polynomials also become intractable for moderate values of n.

Related notation

Definition 6

For real functions $f(x)$, $g(x)$ defined on some part of the real line, $f(x) = o(g(x))$ if and only if for all $C > 0$ there exists x_0 such that $|f(x)| < C|g(x)|$ for all $x > x_0$.
Informally, $f(x)$ grows much more slowly compared to $g(x)$.

Notation	**Definition**
$f(x) = \mathcal{O}(g(x)$	Earlier slide
$f(x) = o(g(x))$	Above
$f(x) = \Omega(g(x))$	$g(x) = \mathcal{O}(f(x))$
$f(x) = \Theta(g(x))$	$f(x) = \mathcal{O}(g(x))$ and $g(x) = \mathcal{O}(f(x))$

$\mathcal{O}()$ complexity and actual time taken

- The $\mathcal{O}()$ complexity is the asymptotic, worst case complexity.
- In practice under different conditions an algorithm with worse theoretical complexity may be faster than one with better theoretically complexity. For example, Quicksort ($\mathcal{O}(n^2)$) works faster than Mergesort ($\mathcal{O}(n\,log(n))$) in many cases - for small to moderate sized sequences.

Pointers I

- The program itself and all the data of a computer program are stored in the memory (RAM) of the computer.
- Computer memory (RAM) can be modelled as a linear sequence of bytes that is typically manipulated in chunks of 2—4—8 bytes. For example, an int typically uses 4 bytes while long uses 8 bytes.
- Each byte (or on some systems a chunk of bytes) has an address which is its position in the linear sequence of bytes that make up the memory.
- Every variable has two items associated with it at all times. A **value** and the location in memory where this value is stored called its **address**.

Pointers II

- To store something into memory we have to give an address and the value to be stored. Most of the time when we use variables in C programs we mean their value. One place where the address is used is in an assignment. In the expression (assume all variables are ints) `a=b+c` the meaning is *store the value obtained by adding the values of b and c in the location for variable a*. So, while `b` and `c` stand for the corresponding values, for `a` the meaning is the corresponding address.
- Pointers provide a way to directly specify and use memory addresses for different types of values.

Pointers III

- To deal with pointers **C** has two operators: * $\equiv$ content-of and & $\equiv$ address-of. The operand for the * operator is an address/pointer and it returns the contents of the address (called de-referencing a pointer). The operand for the & operator is a variable and it returns the starting address where the variable is stored.
- Note that in a **C** program a pointer variable will have a corresponding value (which is an address) and an address where it is stored.
- As a convention in all our code we will add the suffix 'p' to all pointer variables to indicate that they are pointers.