UNIT-1

INTRODUCTION TO ALGORITH & ITS ANALYSIS

Algorithm:

- ⁎ Mis-spelled *logarithm!*.
- ⁎ The first most popular algorithm is the Euclid's algorithm for computing GCD of two numbers.
- ⁎ A well-defined procedure that transfers an input to an output.
- ⁎ Not a program (but often specified like it): An algorithm can often be implemented in several ways.
- ⁎ Knuth's, Art of Computer Programming, vol.1, is a good resource on the history of algorithms!. He says that an algorithm is a finite set of rules that gives a sequence of operations for solving a specific type of problem. Algorithm has five important features:

  Finiteness: must terminate after finite number of steps.

  Definiteness: each step is precisely described.

  Input: algorithm has zero or more inputs.

  Output: has at least one output!.

  Effectiveness: Each operation should be sufficiently basic such that they can be done in finite amount of time using pencil and paper.

– *Design*: The focus of this course is on how to design good algorithms and how to analyze their efficiency. We will study methods/ideas/tricks for developing fast and efficient algorithms.

– *Analysis*: Abstract/mathematical comparison of algorithms (without actually implementing, prototyping and testing them).

- This course will require proving the correctness of algorithms and analyzing the algorithms. Therefore MATH is the main tool. Math is needed in three ways:

  – Formal specification of problem

  – Analysis of correctness

  – Analysis of efficiency (time, memory use,...)

Revise mathematical induction, what is a proof ?, logarithms, sum of series, elementary number theory, permutations, factorials, binomial coefficients, harmonic numbers, Fibonacci numbers and generating functions [Knuth vol 1. or his book Concrete Mathematics is an excellent resource].

- Hopefully the course will show that algorithms matter!

1.2 Model of Computation

- Predict the resources used by the algorithm: running time and the space.
- To analyze the running time we need mathematical model of a computer:

Random-access machine (RAM) model:

  - Memory consists of an infinite array of cells.
  - Each cell can store at most one data item (bit, byte, a record, ..).
  - Any memory cell can be accessed in unit time.
  - Instructions are executed sequentially
  - All basic instructions take unit time:
    - Load/Store
    - Arithmetics (e.g. $+, -, *, /$)
    - Logic (e.g. $>$)

- Running time of an algorithm is the number of RAM instructions it executes.
- RAM model is not realistic, e.g.
  - memory is finite (even though we often imagine it to be infinite when we program)
  - not all memory accesses take the same time (cache, main memory, disk)
  - not all arithmetic operations take the same time (e.g. multiplications are expensive)
  - instruction pipelining
  - other processes
- But RAM model often is enough to give relatively realistic results (if we don't cheat too much).

1.3 Asymptotics

We do not want to compute a detailed expression of the run time of the algorithm, but rather will like to get a feel of what it is like? We will like to see the trend - i.e. how does it increase when the size of the input is increased - is it linear in the size of the input? or quadratic? or exponential? or who knows? The asymptotics essentially capture the rate of growth of the underlying functions describing the run-time. Asymptotic analysis assumes that the input size is large (since we are interested how the running time increases when the problem size grows) and ignores the constant factors (which are usually dependent on the hardware, programming smartness or tricks, compile-time-optimizations).

David Mount suggests the following simple definitions based on limits for functions describing the running time of algorithms. We will describe the formal definitions from [3] later.

Let $f(n)$ and $g(n)$ be two positive functions of n. What does it mean when we say that both f and g grow at roughly the same rate for large n (ignoring the constant factors), i.e.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c,$$

where c is a constant and is neither 0 or ∞. We say that $f(n) \in \theta(g(n))$, i.e. they are asymptotically equivalent. What about $f(n)$ does not grow significantly faster than $g(n)$ or grows significantly faster? Here is the table of definitions from David Mount.

Asymptotic Form	Relationship	Definition
$f(n) \in \Theta(g(n))$	$f(n) \equiv g(n)$	$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
$f(n) \in O(g(n))$	$f(n) \le g(n)$	$0 \le \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
$f(n) \in \Omega(g(n))$	$f(n) \ge g(n)$	$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)}$
$f(n) \in o(g(n))$	$f(n) < g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f(n) \in \omega(g(n))$	$f(n) > g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Example: $T(n) = \sum_{x=1}^{n} x^2 \in \Theta(n^3)$. Why?

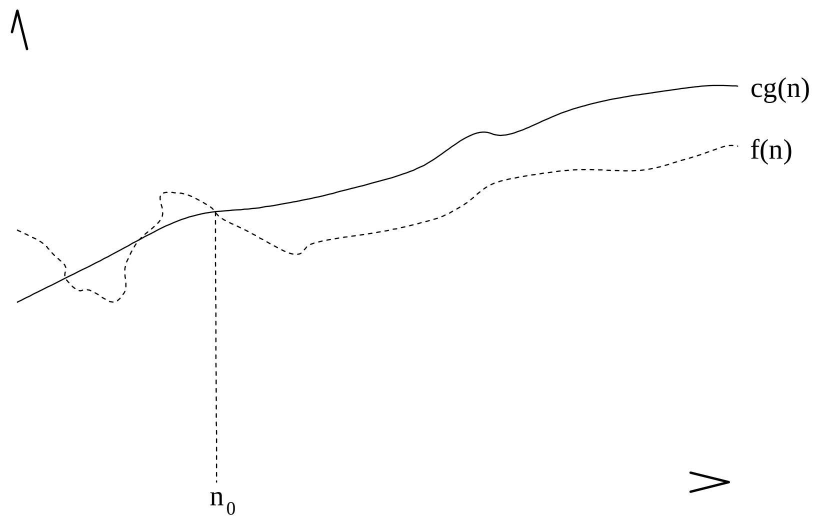
$$\lim_{n\to\infty} \frac{T(n)}{n^3} = \lim_{n\to\infty} \frac{(n^3+3n^2+2n)/6}{n^3} = 1/6,$$

and $0 < 1/6 < \infty$.
Just for fun show that $T(n) \in O(n^4)$ or $T(n) = n^3/3 + O(n^2)$.

1.3.1 O-notation

$O(g(n)) = \{f(n) : \exists\, c, n_0 > 0$ such that $f(n) \le cg(n)\ \forall n \ge n_0\}$

- $O(\cdot)$ is used to asymptotically *upper bound* a function.
- $O(\cdot)$ is used to bound *worst-case* running time.



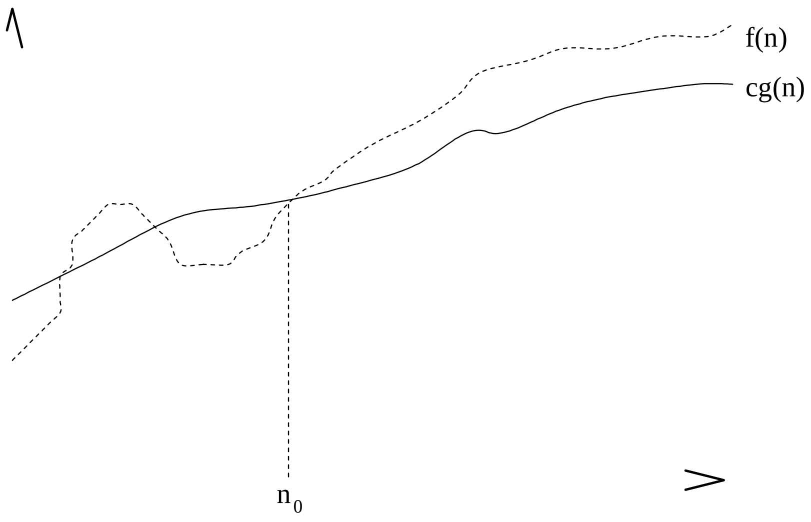
- Examples:
  - $1/3n^2 - 3n \in O(n^2)$ because $1/3n^2 - 3n \le cn^2$ if $c \ge 1/3 - 3/n$ which holds for $c = 1/3$ and $n > 1$.
  - Let $p(n) = \sum_{i=0}^{d} a_i n^i$ be a polynomial of degree d and assume that $a_d > 0$. Then $p(n) \in O(n^k)$, where $k \ge d$ is a constant. What are c and n_0 for this?
- Note:

- When we say –the running time is $O(n^2)$‖ , we mean that the *worst-case* running time is $O(n^2)$ — best case might be better.
- We often abuse the notation:
    - We write $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$!
    - We often use $O(n)$ in equations: e.g. $2n^2 + 3n + 1 = 2n^2 + O(n)$ (meaning that $2n^2 + 3n + 1 = 2n^2 + f(n)$ where $f(n)$ is some function in $O(n)$).
    - We use $O(1)$ to denote a constant.

1.3.2 Ω-notation (big-Omega)

$\Omega(g(n)) = \{f(n) : \exists\, c, n_0 > 0 \text{ such that } cg(n) \le f(n)\ \forall n \ge n_0\}$

- $\Omega(\cdot)$ is used to asymptotically *lower bound* a function.

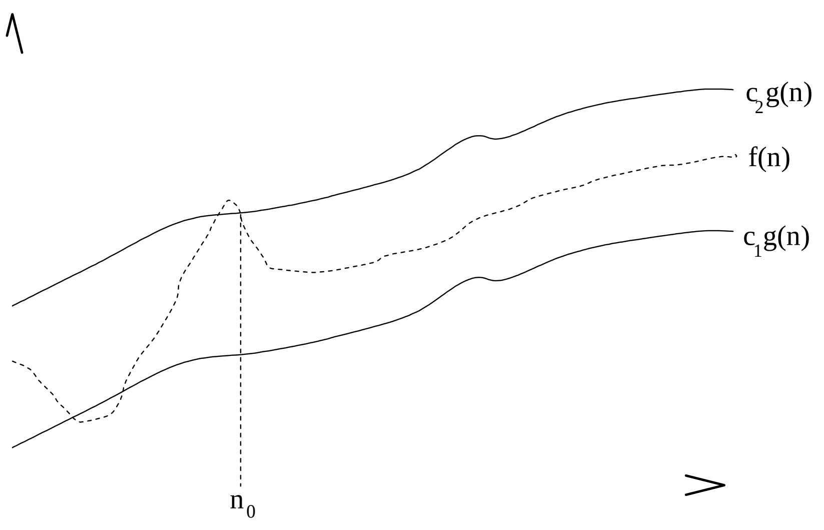


- Examples:
    - $1/3n^2 - 3n = \Omega(n^2)$ because $1/3n^2 - 3n \ge cn^2$ if $c \le 1/3 - 3/n$ which is true if $c = 1/6$ and $n > 18$.
    - Let $p(n) = \sum_{i=0}^{d} a_i n^i$ be a polynomial of degree d and assume that $a_d > 0$. Then $p(n) \in \Omega(n^k)$, where $k \le d$ is a constant. What are c and n_0 for this?
    - Prove or disprove: $g(n) = \Omega(f(n))$ if and only if $f(n) = O(g(n))$.
- Note:
    - When we say –the running time is $\Omega(n^2)$‖ , we mean that the *best case* running time is $\Omega(n^2)$ — the worst case might be worse.

1.3.3 Θ-notation (Big-Theta)

$\Theta(g(n)) = \{f(n) : \exists\, c_1, c_2, n_0 > 0 \text{ such that } c_1 g(n) \le f(n) \le c_2 g(n)\ \forall n \ge n_0\}$

- $\Theta(\cdot)$ is used to asymptotically *tight bound* a function.



$$\boxed{f(n) = \Theta(g(n)) \textit{ if and only if } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))}$$

- Examples:
  - $6n\log n + \sqrt{n}\log^2 n = \Theta(n\log n)$:
    - We need to find n_0, c_1, c_2 such that $c_1 n\log n \le 6n\log n + \sqrt{n}\log^2 n \le c_2 n\log n$ for $n > n_0$. $c_1 n\log n \le 6n\log n + \sqrt{n}\log^2 n \Rightarrow c_1 \le 6 + \frac{\sqrt{n}}{\log n}$. Ok if we choose $c_1 = 6$ and $n_0 = 1$. $6n\log n + \sqrt{n}\log^2 n \le c_2 n\log n \Rightarrow 6 + \frac{\log n}{\sqrt{n}} \le c_2$. Is it ok to choose $c_2 = 7$? Yes, $\log n \le \sqrt{n}$ if $n \ge 2$.
    - So $c_1 = 6$, $c_2 = 7$ and $n_0 = 2$ works.
  - Let $p(n) = \sum_{i=0}^{d} a_i n^i$ be a polynomial of degree d and assume that $a_d > 0$. Then $p(n) \in \Theta(n^k)$, where $k = d$ is a constant.

1.4 How to analyze Recurrences?

There are many ways of solving recurrences. I personally prefer the recursion tree method, since it is visual! Here the recurrence is depicted in a tree, where the nodes of the tree represent the cost incurred at the various levels of the recursion. We illustrate this method using the following recurrence (so called the recurrence used in the Masters method).

Let $a \ge 1, b > 1$ and $c > 0$ be constants and let $T(n)$ be the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k,$$

defined for integer $n \ge 0$. Then

Case 1 $a > b^k$ then $T(n) = \Theta(n^{\log_b a})$.

Case 2 $a = b^k$ then $T(n) = \Theta(n^k \log_b n)$.

Case 3 $a < b^k$ then $T(n) = \Theta(n^k)$.

The proof is fairly simple (of course I would have made some errors on what I have written below). We need to sort of visualize it using the recursion tree.

Level 1: a subproblems are formed, each of size n/b, and the total cost is cn^k.

Level 2: a^2 subproblems are formed, each of size n/b^2, and the total cost is $a * c(n/b)^k$.

Level 3: a^3 subproblems are formed, each of size n/b^3, and the total cost is $a^2 * c(n/b^2)^k$.

…

Level $\log_b n$: $a^{\log_b n}$ subproblems are formed, each of constant size and the total cost is about $a^{\log_b n} c(\frac{n}{b^{\log_b n}})^k$.