Algorithms – I (CS29003/203)

Autumn 2022, IIT Kharagpur

Analysis of Algorithms

Analyzing Algorithms

- Predict how your algorithm performs in practice
- By analyzing several candidate algorithms for a problem we can identify efficient ones
- Criteria
  - Running time
  - Space usage
  - Cache I/O
  - Main memory I/O
  - Lines of codes

Analyzing Running Time

- Random Access Machine (RAM) model
- Every operation including memory access, arithmetic operations etc. takes same amount of time.
- Is it precise?
- Not really. But precise model would be tedious – would yield very little insight into algorithm design and analysis
- However, we should be careful not to abuse it
- We only care about "Order of the cost", i.e., we omit
  - Lower order terms
  - Constants

Asymptotic Analysis

Why Asymptotic Analysis

- Because we only care about how fast a function grows!

- Would you rather have a million rupees one time or one paisa on day one, doubled every day for a month?

- Actually, the second option can get you more than 1 million (in around 27 days itself)

- $1 + 2 + 4 + \cdots + 2^{i-1} = (2^i - 1)$ paisa = 1.342 million (for $i = 27$)

Running Time Analysis of Insertion Sort

- Lets go back to insertion sort and see its running time
- Our expression will evolve from a messy formula that assumes each line of the code takes a constant amount of time

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

t_j denotes the number of times the lines in while loop executes for that value of j.

Running Time Analysis of Insertion Sort

- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$
- Best case: (Array is already sorted) -> $t_j = ?$ $\boxed{1}$
- $T(n) = c_1 n + (c_2 + c_4 + c_5 + c_8)(n-1) = an + b$ -> a linear function of n
- Worst case: (Array is reverse sorted) -> $t_j = ?$
- t_j is maximum each time, i.e., $t_j = j$
- $T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + (c_6 +$

Best/Average/Worst Case Analysis

- We looked at both `best case' (input array was already sorted) and `worst case' (input array was reverse sorted)

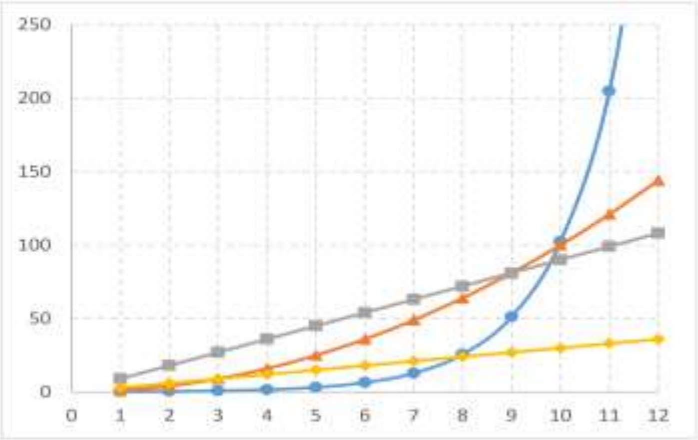
- In this course, we shall usually concentrate on worst-case running time

- Major reasons
  - Gives an upper bound on the running time for any input
  - For some algorithms, worst case occurs fairly often, e.g., searching
  - Average case is often roughly as bad as the worst case.

Order of Growth

- In analyzing running time for `insertion sort', we started with constants c_i to represent the cost of each statement
- Then we observed that they give more detail than we need and we discarded them
- We shall go ahead with more simplifying abstraction: **Rate/Order of Growth**
- For the function $f(n)$ we care when n is large enough. When n is small, $f(n)$ is small anyway
- The constant factors and lower order terms doesn't affect the growth of the function
- One algorithm is more efficient than another if its worst-case running time has a lower order of growth

Order of Growth



$g_2(n) = 0.1 \times 2^n$

$g_1(n) = n^2$

$f_2(n) = 9n$

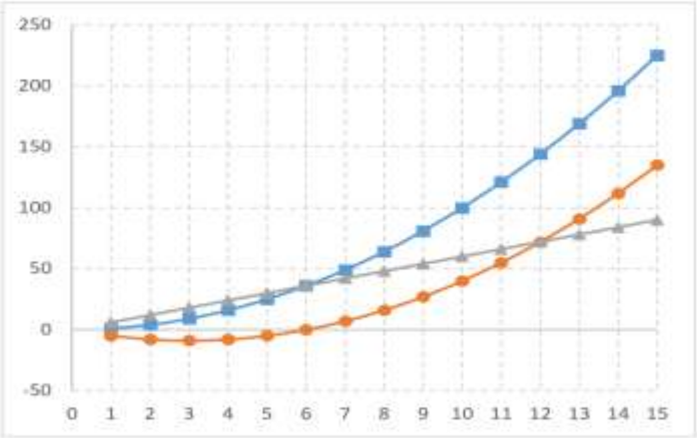
$f_1(n) = 3n$

Omit the constant factors

When n is large enough, $g_1(n)$ will be much larger than $f_1(n)$ or $f_2(n)$

$f_1(n)$ and $f_2(n)$ will have similar growth trend

Order of Growth



$g_1(n) = n^2$

$g_2(n) = n^2 - 6n$

$f(n) = 6n$

Omit the lower-order terms

When n is large enough, $g_1(n)$ or $g_2(n)$ will still be much larger than $f(n)$

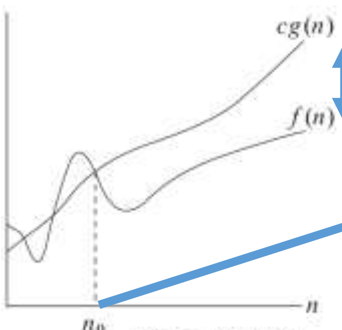
$g_1(n)$ and $g_2(n)$ will have similar growth trend because $-6n$ is much smaller compared to n^2

Asymptotic Notations

- These notations are used to describe the asymptotic running time of an algorithm
- However, asymptotic notations can apply to other functions that have nothing to do whatsoever with algorithms

O (Big-O)

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



$f(n) = O(g(n))$

After some constant n_0

$f(n) \le cg(n)$

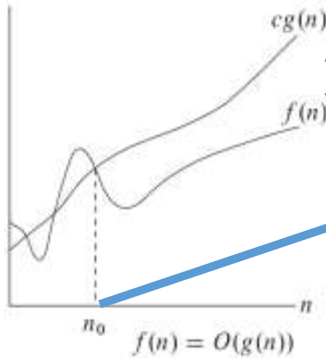
- Asymptotic upper bound
- Note the $\le$: can be of the same order, but can be smaller as well

- $n = O(n^2), n^2 = O(n^2), n^3 \ne O(n^2)$

O (Big-O) [≤]

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

- $f(n) = 3n^2, g(n) = n^2$
- How can we show, $f(n) = O\big(g(n)\big)$
- Let $c = 3, n_0 = 5$
- $cg(n) = 3n^2$, so $f(n) \le cg(n)$
- Similarly, let $c = 10, n_0 = 2$
- $cg(n) = 10n^2$, so $f(n) \le cg(n)$



After some constant n_0

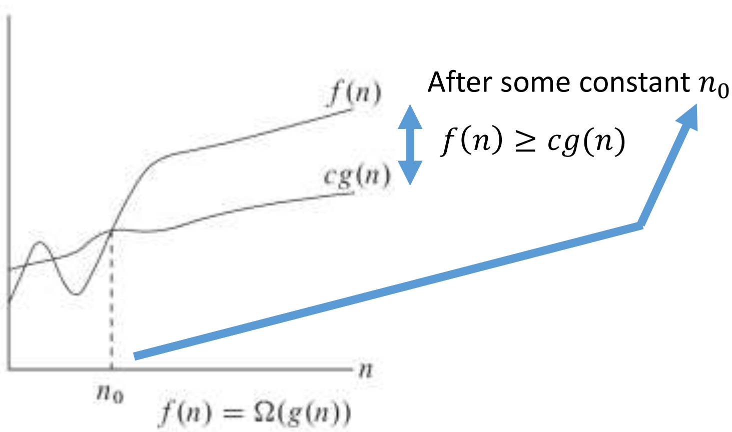
$f(n) \le cg(n)$

Always larger than $n^2 + 2n$

- $f(n) = n^2 + 2n, g(n) = n^3$; How can we show, $f(n) = O\big(g(n)\big)$?
- Let $c = 3, n_0 = 10$; $cg(n) = 3n^3 = n^3 + 2n^3 = n(n^2 + 2n^2)$
- so $f(n) \le cg(n)$

Ω (Big- Ω) [≥]

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

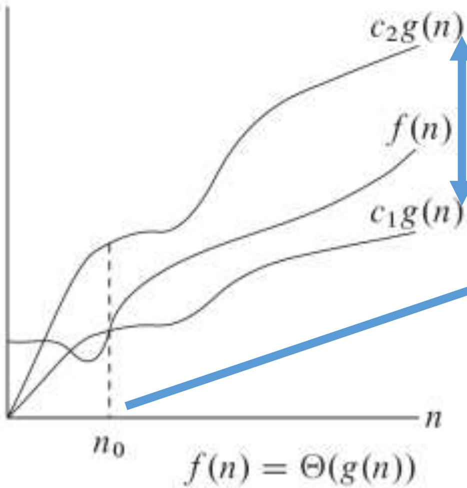


- Asymptotic lower bound
- can be of the same order, but can be larger as well

- $n \ne \Omega(n^2), n^2 = \Omega(n^2), n^3 = \Omega(n^2)$

Θ (Big- theta) [=]

$\Theta(g(n)) = \{f(n) \colon \exists\, c_1, c_2 > 0, n_0 > 0, s.t. 0 \le c_1 g(n) \le f(n) \le c_2 g(n)\ \forall\, n \ge n_0\}$



After some constant n_0 $f(n)$ must be within this range

- Asymptotic tight bound
- Must be of the same order
- $f(n) = \Theta(g(n))$ means $f(n) = O(g(n))$ [≤] and $f(n) = \Omega(g(n))$ [≥], must be =

- $n \neq \Theta(n^2), n^2 = \Theta(n^2), n^3 \neq \Theta(n^2)$

What This also Means

- O($g(n)$): class of functions $f(n)$ that grow <u>no faster</u> than $g(n)$

- **Θ**($g(n)$): class of functions $f(n)$ that grow <u>at same rate</u> as $g(n)$

- **Ω**($g(n)$): class of functions $f(n)$ that grow <u>at least as fast</u> as $g(n)$

Analogy to Real Numbers

functions	Real Numbers
$f(n) = O(g(n))$	$a \leq b$
$f(n) = \Omega(g(n))$	$a \geq b$
$f(n) = \Theta(g(n))$	$a = b$

o (small-o) [<] and ω (small ω)

$$o(g(n)) = \{f(n): for\ any\ c > 0, \exists\ n_0 > 0, such\ that\ 0 \leq f(n) < cg(n) \forall\ n \geq n_0\}$$

- Asymptotic non-tight upper bound
- Note the <: must be smaller
- Equivalently, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

- By analogy,

$$\omega(g(n)) = \{f(n): for\ anyc > 0, \exists n_0 > 0, such\ that\ 0 \leq cg(n) < f(n)\ \forall\ n \geq n_0\}$$

- Asymptotic non-tight lower bound
- Note the >: must be larger
- Equivalently, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Analogy to Real Numbers

functions	Real Numbers
$f(n) = O(g(n))$	$a \leq b$
$f(n) = \Omega(g(n))$	$a \geq b$
$f(n) = \Theta(g(n))$	$a = b$
$f(n) = o(g(n))$	$a < b$
$f(n) = \omega(g(n))$	$a > b$

Time Complexity

- Consider an $\Theta(2^n)$ algorithm running on a computer that can execute 10^8 ops/sec
- For $n = 50$, what amount of time will be required?

Subset Sum Selection Problem

- Given a set $\mathcal{S}$ of integers and a target T, determine if $\mathcal{S}$ has a subset that sums to T exactly.
- $\mathcal{S} = \{1, 2, 5, 9, 10\}$ and $T = 22$?
- What about $T = 23$?

Complexity of Parts to Whole

- Suppose you have 3 sections of an algorithm for which you know the complexities are like this.

$\Theta(n^2)$
$O(n^2)$
$O(n^2)$

- What can you tell about the complexity of the overall algorithm?

Theorem

- If $t_1(n) = O(g_1(n))$ and $t_2(n) = O(g_2(n))$, then $t_1(n) + t_2(n) = O(max\{g_1(n), g_2(n)\})$
- The algorithm's overall efficiency will be determined by the part with a larger order of growth, i.e., its least efficient part.
- For example, $5n^2 + 3nlogn = O(n^2)$

Proof. There exist constants c_1, c_2, n_1, n_2 such that

$$t_1(n) \leq c_1 * g_1(n) \;\forall\, n \geq n_1$$
$$t_2(n) \leq c_2 * g_2(n) \;\forall\, n \geq n_2$$

Define $c_3 = c_1 + c_2$ and $n_3 = \max\{n_1, n_2\}$, then

$$t_1(n) + t_2(n) \leq c_3 * \max\{g_1(n), g_2(n)\} \;\forall\, n \geq n_3$$

Thank You!!