TIME COMPLEXITY ANALYSIS

Data Structures (CS2001) Fall 2022

Abeeda Akram

Time complexity Analysis

Time complexity analysis for an algorithm is *independent* of programming language, and machine.

Objectives of time complexity analysis is:

1. To determine the feasibility of an algorithm by estimating an *upper bound* on the amount of work performed
2. To compare different algorithms before deciding, which one to implement

- **T(n) is usually very complicated so we need an approximation of T(n)….close to T(n).**
- **This measure of efficiency or approximation of T(n) is called ASYMPTOTIC COMPLEXITY or ASYMPTOTIC ALGORITHM ANALYSIS**

Asymptotic Analysis

Is a way of expressing the running time of an algorithm, using idealized units of computational work.

- Time is not in numbers of seconds or any such time unit.
- It represents **number of operations** that are carried out while executing the algorithm.
- Depends on **Input Data**
- Provides the **best**, **average** and the **worst** running times of an algorithm.

Big-O Notation

Is a formal method of expressing the upper bound of an algorithm's running time.

- The longest amount of time, it could possibly take for the algorithm to complete.

For non-negative functions, f(n) and g(n),

if there exists an integer **no** and a constant **c > 0**, such that for all integers n, **$f(n) \leq cg(n)$**, then f(n) is Big O of g(n).

This is denoted as "$f(n) = O(g(n))$".

From Function to Notation

$$f(n) = 2n + 3 \quad \rightarrow \quad O(n)$$
$$h(n) = 2n^2 + 2n + 3 \quad \rightarrow \quad O(n^2)$$

- Constants are ignored
- Terms growing faster, dominate

From Function to Notation

$f(n) = 4n^4 + 2n^2 + 2n + 3 \rightarrow O(n^4)$

$f(n) = 4n^4 + 2^n + 3 \rightarrow ?$

Which term grows rapidly with growing input?

- Constants are ignored
- Terms growing faster, dominate

Example: *Big-O*

Formula: for $n >= n_0$, $f(n) <= cg(n)$

$f(n) = 2n + 3$

$g(n) = (n)$

find c and n_0

n	f (n)	c g(n)	
		c = 2	c = 3
1	5	2	3
2	7	4	6
3	9	6	9
4	11	8	12

Answer : c=3, n_0 = 3

Example: *Big-O*

$f(n) = n^2 + 2n + 5$

$g(n) = (n)$ find c and n_0

Big-Ω omega Notation

Is a formal method of expressing the lower bound of an algorithm's running time.

- The smallest amount of time, it could possibly take for the algorithm to complete.

For non-negative functions, f(n) and g(n),

if there exists an integer **no** and a constant **c > 0,** such that for all integers n, **f(n) ≥ cg(n),** then f(n) is Big Ω of g(n).

This is denoted as "$f(n) = \Omega(g(n))$".

Example: *Big-Ω omega*

Formula: for $n >= n_0$, $f(n) >= cg(n)$

$f(n) = 2n + 3$

$g(n) = (n)$

find c and n_0

Big-θ theta Notation

This is basically saying that the function, f(n) is bounded both from the top and bottom by the same function, g(n).

For non-negative functions, f(n) and g(n), f(n) is theta of g(n) if there exists an integer **no** and a constants **c1 > 0 and c2>0,**

such that $c1g(n) \le f(n) \le c2g(n)$

iff ***f(n) = O(g(n)) and f(n) = Ω(g(n))***.

This is denoted as "f(n) = Θ(g(n))".

Example: *Big-θ theta*

$f(n) = 2n + 3$

$g(n) = (n)$

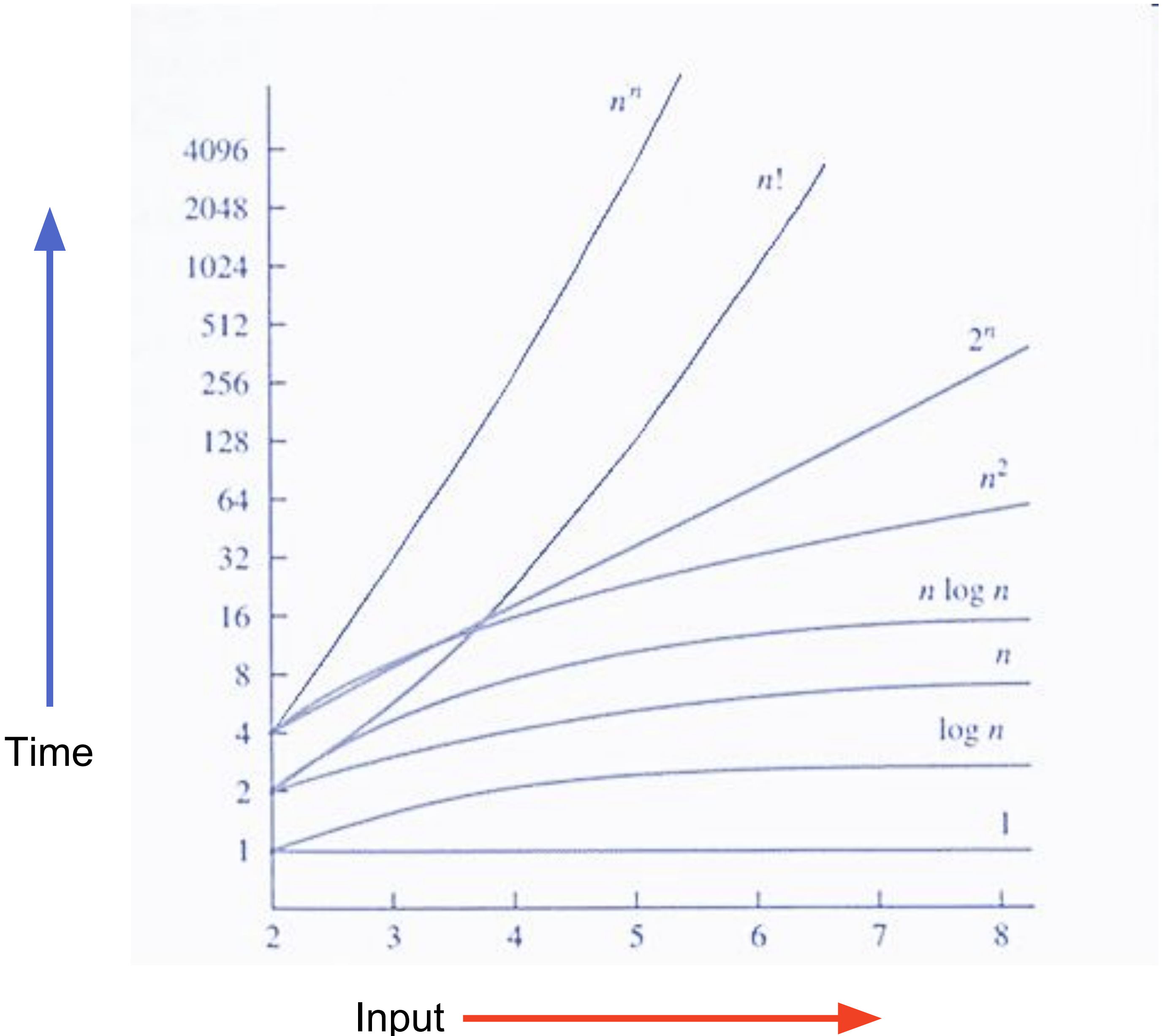
find c and n_0

Common Functions

- $O(1)$ constant / bounded time
- $O(\log_b n) = O(lg n)$ logarithmic time
- $O(n)$ linear
- $O(n \log_b n)$ n log n
- $O(n^2)$ quadratic
- $O(n^3)$ cubic
- $O(n^k)$ polynomial
- $O(2^n)$ exponential

Relative Growth of functions

n	lgn	nlgn	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
32	5	160	1024	32768	4294967296
64	6	384	4096	262144	1.84467E+19
128	7	896	16384	2097152	3.40282E+38
256	8	2048	65536	16777216	1.15792E+77
512	9	4608	262144	134217728	1.3408E+154


Time
4096
2048
1024
512
256
128
64
32
16
8
4
2
1
2
3
4
5
6
7
8
n^n
$n!$
2^n
n^2
$n \log n$
n
$\log n$
1
Input

Properties of Big-O

1. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$
2. If $f(n) = O(g(n))$ and $h(n) = O(g(n))$ then $f(n) + h(n) = O(g(n))$
3. $n^k = O(n^{(k+j)})$ for any positive j
4. If $f(n)$ is polynomial of degree k then $f(n) = \theta(n^k)$
5. if $f1(n) = O(g(n))$ and $f2(n) = O(h(n))$ then $f1(n) + f2(n) = O(\max (g(n), h(n))$
6. $\log^k(n) = O(n)$ for any constant k