Big O crash plan

28/09/2025

6.006 Intro to algorithms MIT, OCW.

Big O cheat sheet.

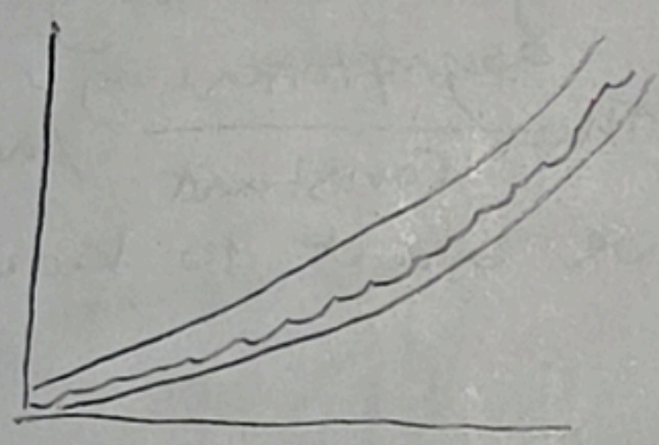
Asymptotic Complexity.

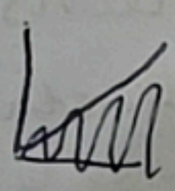
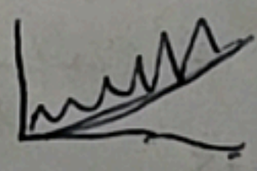
bigocheet sheet.com.

Asymptotic notation, also known as "big-oh" notation uses symbols O, Θ, Ω.

$f_2(x) = \Theta(x^2)$ is misleading because it makes it seem like $\Theta(x^2)$ is a function. $f(x) = \Theta(x^2)$ implies both a lower and upper bound.

The graph shows visual proof that $f_2(x) = \Theta(x^2)$ by showing that its bounded from above by $1.2x^2$ and its bounded below by $0.9x^2$. These two functions differ from the function inside the Θ by a constant factor.



Θ Constrains a function both from above and from below.
O only makes a statement about the upper bound of a function  ① and Ω makes a statement about the functions lower bound  ②.

In ① we cannot say $g(x) = \Theta(x^2)$ because to coefficient of x^2 becomes 0.

In Asymptotic Drowning

In asymptotic notation we can reduce complex functions involving logarithms. Using the following rules.

- $\log(n^{100}) = 100\log(n) = \Theta(\log(n))$ - Constant exponents don't matter.
- $\log_{\log(5)}(n) = \frac{\log(n)}{\log(5)} = \Theta(\log(n))$ - Constant bases don't matter.
  - Don't confuse an exponential inside a logarithm with a logarithm inside an exponential

e.g. $n^{\log(5)}$ cannot be simplified, we can get some loose bounds for it by observing that $\log(5) \approx 2.3219\ldots$, so $n^2 \leq n^{\log(5)} \leq n^3$.

Khan Academy short pieces on Big-O, Big-Ω Big Θ.

- We use BigO notation to ~~asymptomatically~~ asymptotically bound the growth of a running time, to within a constant factors above and below. Sometimes we want to bound from only above.

Worst case running time of a binary search is $O(\log_2 n)$.

- It would be convinient to have a form of asymptotic notation that means "the running time grows at most this much, but it could grow more slowly" ⇒ here we use ~~big-O~~ big-O.

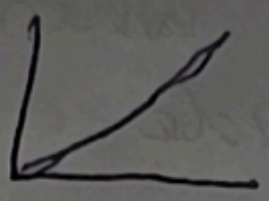
$O(n)$ ← no. inputs

↑ order Complexity.

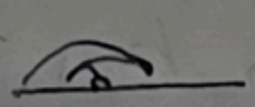
⟶ Time Complexity ↓ takes longer

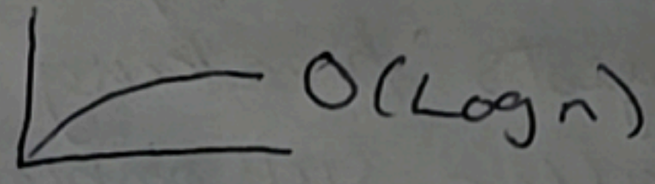
Space Complexity ↓ requires more memory.

Take for example a loop over an array of elements takes 1 unit of time to loop over each element,

↓ this is linear complexity.  but we can do better

② use sorting array, binary search jumping in middle.





This is logarithmic complexity, scales better to large amounts of inputs.

③ loop within loop for example, less efficient. $O(n^2)$ also known as quadratic complexity.

④ Very bad, exponential.

Examples:

$f(n) - (x?)$ → atoms in universe.

$= 10^{80} - \Theta(1)$

$f(n) = (20n)^7 - \Theta(n^7)$

$f(n) = 5^{\log(3)} n^3 + 10^{80} n^2 + \log(3) n^{3.1} + 6006 \ldots =$

$\Theta(n^{3.1})$

↓ eliminate constant factor to get

$f(n) = \log\binom{n}{\frac{n}{2}} = \Theta(n^3) + \Theta(n^2) + \Theta(n^{3.1}) + \Theta(1) \rightarrow \Theta(n^{3.1})$ dominates the sum.

$O(n)$. Use binomial identity to obtain $\binom{n}{n/2} = \frac{n!}{(\frac{n}{2}!)^2}$ = then apply Stirling's approximation and log properties.

Recurrences:

Reccurences show up when trying to analyse the running time of divide-and-conquer algorithms.

Divide-and-conquer (d&c) is a general approach that suggests breaking down big problems into many smaller subproblems to obtain the solution to the bigger problem.

1. Divide Break a problem into smaller sub-problems.
2. Conquer Really small subproblems are easier.
3. profit combine answers to subproblems.

Sample recurrence

Binary search is the canon example of d&c.
- half → half → half.

Divide. compare $A[\frac{n}{2}]$ with x. If the numbers are equal, report success. If $x < A[\frac{n}{2}]$, recurse on $A[1 \ldots \frac{n}{2} - 1]$. otherwise, recurse on $A[\frac{n}{2} + 1 \ldots n]$ (repeat process focusing on)

Conquer: If the array A is empty, x cannot be in it, so I can report failure.

Profit: If I have found x, the problem is obviously solved. However, if I have not found x in the half of the array that I recursed on, I need to convince myself that I'm not missing out by ignoring the other half of the array, and declare that x is not in A, despite the fact that I completely ignored half of its elements.

5

Let $T(n)$ be Time it takes to find x in $A[1....n]$, or give up.

e.g 1. where x does not exist in A.

Assume a guess takes a constant amount of T.

We can write $T(n)$

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

$$T(1) = \Theta(1)$$

any constant is $\Theta(1)$, we could have used more base cases.

Lets aim to guess the closed form formula for $T(n)$

since we're making a guess, ~~we'll~~ we can ~~use~~ re-write recurrence as $\Theta(1)$ as $T(n) = T\left(\frac{n}{2}\right) + c$.

$$T(n) = T\left(\frac{n}{2}\right) + c$$

$$T(n) = T\left(\frac{n}{4}\right) + c + c$$

$$T(n) = T\left(\frac{n}{8}\right) + c + c + c$$

$$T(n) = T\left(\frac{n}{16}\right) + c + c + c + c + c.$$

$$\vdots$$

$$T(n) = T\left(\frac{n}{2^i}\right) + i \times c.$$

using $i = \log(n)$ we get $T(n) = \Theta(\log(n))$

Reccurence Traps.

One Potential pitfall of proof involving big-O is notation hides info on constents involved.

function $f(n) = n$ is $O(1)$

Proof by Induction base case $n=1$.

Clearly $f(1) = 1$ is a constent so we can say $f(1) = O(1)$!

Inductive step assume $f(x) = O(1)$ for all $x < n$.

$$f(n) = n$$
$$= n - 1 + 1$$
$$= f(n-1) + 1$$
$$= O(1) + 1$$
$$= O(1)$$

AAAH problem X.

By definition $f(n)$ is $O(1)$, if only if there exist constent c.

Big O notation hides the fact that the constant increases with every step of the inductive process.

Therefore is not actually constant.

∴ avoid Big o notation in proofs by induction. best to replace with definitions.

$$f(n) = n$$
$$= n - 1 + 1$$
$$= f(n-1) + 1$$
$$< c + 1.$$

CS50

28/09/2025.

Thats it for binary.

focus on HIGHER LEVEL LANGUAGE

example of C.

```
#include <stdio.h>
int main(void)
{
    Printf("hello world\n");
}
```

Today we're going to remove punctuation.

Starting with scratch.mit.edu.

Sprite - character in world.

the purple puzzle pieces in scratch are functions

the yellow are _ _ _ _ . Inputs

Return value, handed back and stored in variable.

Computer scientists generally start counting from 0.

Rest of Lecture 0 was mostly using ~~sprite~~ scratch.