2B: Orders of Growth

CS1101S: Programming Methodology

Martin Henz

August 19, 2016

Orders of Growth

Exponential growth

The first version of `fib` runs in a time that grows exponentially with the argument n.

Linear growth

The second version of `fib` runs in a time (linearly) proportional to the argument n.

What exactly do we mean by this?

Purpose

Rough measure

We are interested in a rough measure of resources used by a computational process.

Abstraction

"Order of growth" is an abstraction technique. We decide to ignore details that we deem irrelevant: the processor speed of the computer, the programming environment, the programming language, or mindor differences in programming style.

Recap: Resources

- Time: How long it takes to run the program?
- Space: How much memory do we need to run the program?

Example: Double the argument of `factorial`

```
factorial(2)
2 * factorial(1)
2 * 1
2

factorial(4)
4 * factorial(3)
4 * (3 * factorial(2))
4 * (3 * (2 * factorial(1)))
4 * (3 * (2 * 1))
4 * (3 * 2)
4 * 6
24
```

Example: First version of `fib`

The number of leaves in the recursion tree?

$fib(n+1)$

Comparision

- `fib(10)` needs to visit *fib(11)=89* leaves
- `fib(20)` needs to visit *fib(21)=10946* leaves
- `fib(100)` needs to visit *fib(101)=573147844013817084101* leaves

“Big Theta”

What are we talking about?

Let n denote the size of the problem, and let $r(n)$ denote the resource needed solving the problem of size n.

Definition

The function r has order of growth $\Theta(g(n))$ if there are positive constants k_1 and k_2 such that $k_1 \cdot g(n) \leq r(n) \leq k_2 \cdot g(n)$ for any sufficiently large value of n.

What does “sufficiently large” mean?

Definition from previous slide

The function r has order of growth $\Theta(g(n))$ if there are positive constants k_1 and k_2 such that $k_1 \cdot g(n) \leq r(n) \leq k_2 \cdot g(n)$ for any sufficiently large value of n.

More formal definition

The function r has order of growth $\Theta(g(n))$ if there are positive constants k_1 and k_2 **and a number** n_0 such that $k_1 \cdot g(n) \leq r(n) \leq k_2 \cdot g(n)$ **for any** $n > n_0$.

"Big Oh"

What are we talking about?

Let n denote the size of the problem, and let $r(n)$ denote the resource needed solving the problem of size n.

Definition

The function r has order of growth $O(g(n))$ if there is a positive constant k such that $r(n) \leq k \cdot g(n)$ for any sufficiently large value of n.

"Big Omega"

What are we talking about?

Let n denote the size of the problem, and let $r(n)$ denote the resource needed solving the problem of size n.

Definition

The function r has order of growth $\Omega(g(n))$ if there is a positive constant k such that $k \cdot g(n) \leq r(n)$ for any sufficiently large value of n.

Do constants matter?

Let's say r has order of growth $\Theta(n^2)$
Does r also have order of growth $\Theta(0.5n^2)$?

Constants don't matter

We can freely choose k, k_1 and k_2

Do minor terms matter?

Let's say r has order of growth $O(n^2)$
Does r also have order of growth $O(n^2 - 40n + 3)$?

Minor terms don't matter

We can adjust n_0, k, k_1 and k_2 such that the minor terms are overruled.

Some common $g(n)$

- 1
- $\log n$
- n
- $n \log n$
- n^2
- n^3
- 2^n

How do we calculate “Big Oh/Theta/Omega”

- Topic of algorithm analysis (CS3230)
- For us:
  - Identify the basic computational steps
  - Try a few small values
  - Extrapolate
  - Watch out for “worst case” scenarios

Some Numbers

n	$log\ n$	$n\ log\ n$	n^2	n^3	2^n
1	0	0	1	1	2
2	0.69	1.38	4	8	4
3	1.098	3.29	9	27	8
10	2.3	23.0	100	1000	1024
20	2.99	59.9	400	8000	10^6
30	3.4	102	900	27000	10^9
100	4.6	460.5	10000	10^6	$1.2 \cdot 10^{30}$
200	5.29	1059.6	40000	$8 \cdot 10^6$	$1.6 \cdot 10^{60}$
300	5.7	1711.13	90000	$27 \cdot 10^6$	$2.03 \cdot 10^{90}$
1000	6.9	6907	10^6	10^9	$1.07 \cdot 10^{301}$
2000	7.6	15201	$4 \cdot 10^6$	$8 \cdot 10^9$	
3000	8	24019	$9 \cdot 10^6$	$27 \cdot 10^9$	
10^6	13.8	$13.8 \cdot 10^6$	10^{12}	10^{18}	

The world's oldest algorithm

Greatest Common Divisor (GCD)

Given two positive integers, find the largest integers that divide both without remainder.

Euclid's original solution

```
function gcd(a, b) {
    return a === b ? a
                   : a > b ? gcd(a - b, b)
                           : gcd(a, b - a);
}
```

The world's oldest algorithm

More modern version

```
function gcd(a, b) {
    return b === 0 ? a
                   : gcd(b, a % b);
}
```

"Pedistrian" Power function

```
function power(b, e) {
    return (e === 0) ? 1 : b * power(b, e - 1);
}
```

Power function: Can we do better?

Example

Calculate 17^6

Simplification

$17^6 = (17 \cdot 17)^{6/2}$

How about

17^7?

Simplification

$17^7 = 17 \cdot 17^{7-1} = 17 \cdot 17^6$ and apply previous trick

Fast Power

```
function fast_power(b, e) {
    if (e === 0) {
        return 1;
    } else if (is_even(e)) {
        return fast_power(b * b, e / 2);
    } else {
        return b * fast_power(b, e - 1);
    }
}
```

Summary

- Big Theta, Big Oh, Big Omega
- The world's first algorithm
- The world's niftiest algorithm?