Exercises

Eleni Mangina

Room B2.05
School of Computer Science and Informatics
University College Dublin, Ireland

“Big-Oh” Notation

Given functions f(n) and g(n), we say that

f(n) is **O(**g(n)**)** if and only if

there are positive constants c and n_0 such that

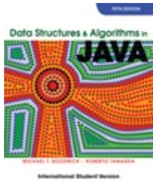
$f(n) \leq c * g(n)$ for all $n \geq n_0$

Example

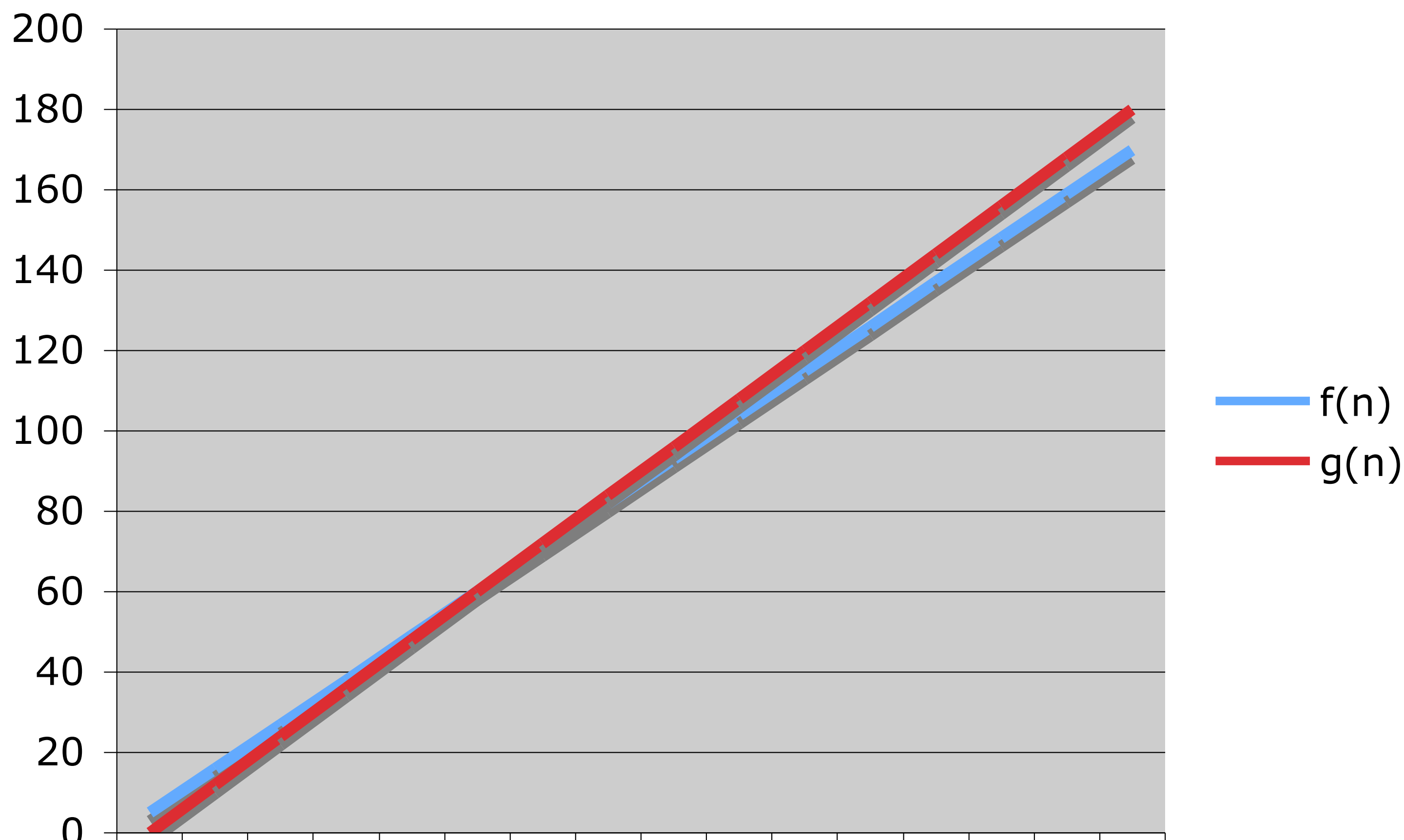
- Consider an algorithm with running time 11n + 5.
  - $f(n) = 11n + 5$
- If we can show that $g(n) = n$ satisfies the constraint:
  - $f(n) \leq c * g(n)$ for all $n \geq n_0$
- Then, we can prove that $11n + 5$ is $O(n)$:
- So, lets consider $c = 12$:

$$\begin{aligned} 11n + 5 &\leq 12n \\ 5 &\leq 12n - 11n \\ 5 &\leq n \end{aligned}$$

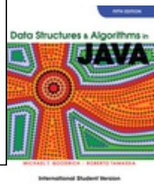
- Therefore, $f(n) \leq 12 * g(n)$ for all $n \geq 5$



Example

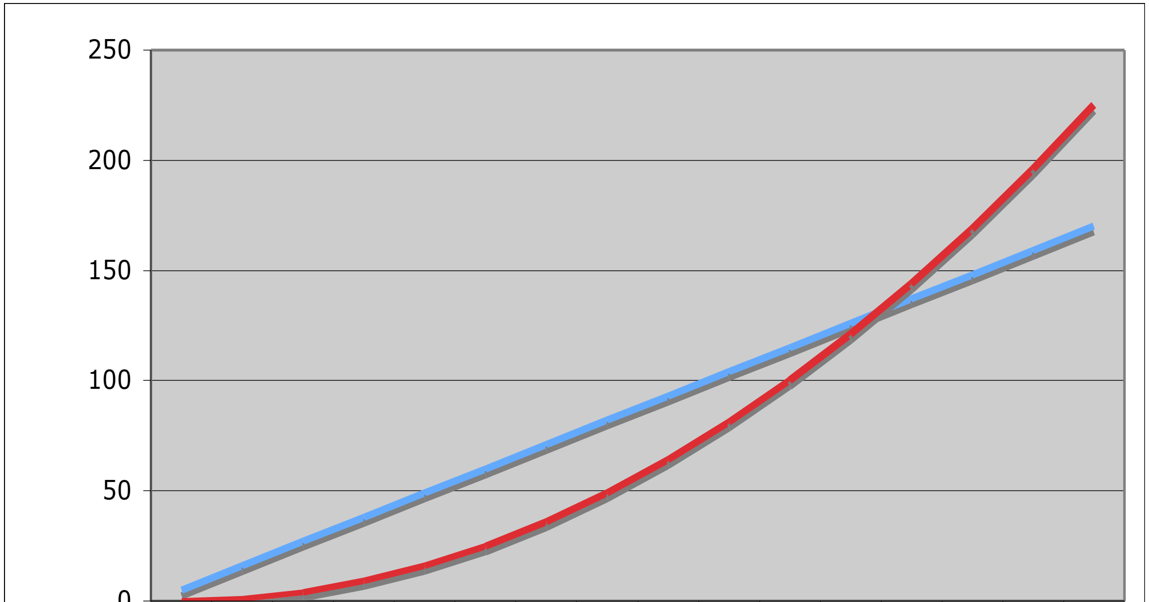


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
f(n)	5	16	27	38	49	60	71	82	93	10	11	12	13	14	15	17
g(n)	0	12	24	36	48	60	72	84	96	10	12	13	14	15	16	18



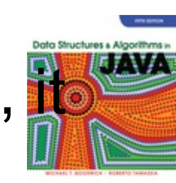
Using “Big-Oh” Notation

- Basically, when we choose g(n), we can choose any function that we want.
  - For example, consider: $f(n) = 11n + 5$, $g(n) = n^2$, $c = 1$, $n_0 = 12$



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
f(n)	5	16	27	38	49	60	71	82	93	104	115	126	137	148	159	170
g(n)	0	1	4	9	16	25	36	49	64	81	100	121	144	169	196	225

- However, while it is correct to say that “$11n + 5$ is $O(n^2)$”, it is not a **good approximation**.

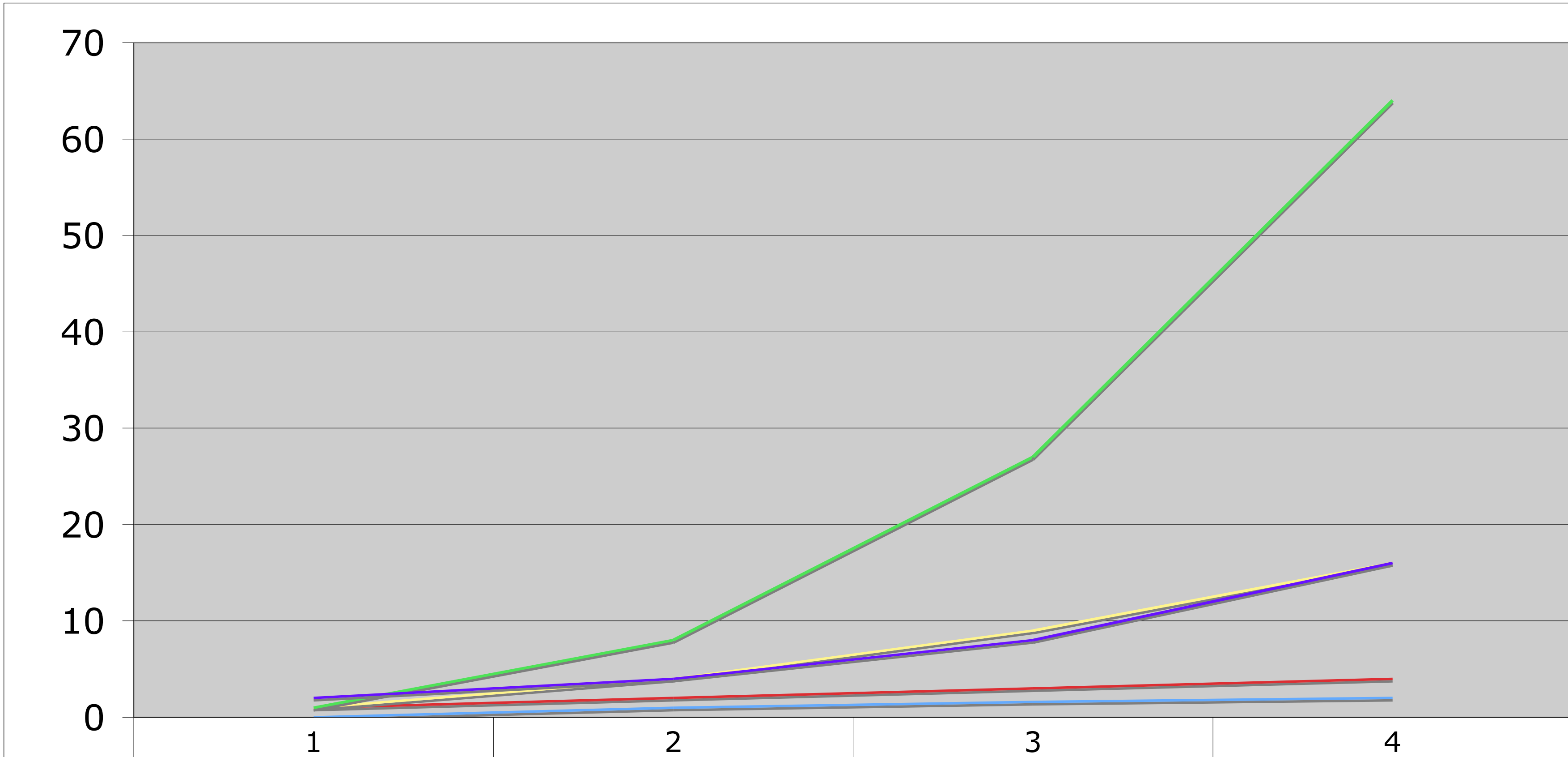


Using "Big-Oh" Notation

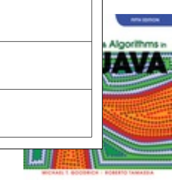
- An approximation is good if it is similar to the actual function, but does not include lower order terms.

- The hierarchy allows us to classify algorithms:
  - fixed: $O(1)$
  - logarithmic: $O(\log n)$
  - linear: $O(n)$
  - quadratic: $O(n^2)$
  - polynomial: $O(n^k)$, $k \geq 1$
  - exponential: $O(a^n)$, $n > 1$

Using “Big-Oh” Notation

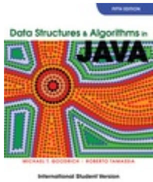


	1	2	3	4
log n	0	1	1.584962501	2
n	1	2	3	4
n2	1	4	9	16
n3	1	8	27	64
2n	2	4	8	16



In Practice

- **Simple Rule**: Drop lower order terms and constant factors:
  - $11n + 5$ is $O(n)$
  - $8n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$
- **Simpler Rule:** Only examine loops / recursion
  - A loop over a fixed range [i-n] is typically O(n)
  - A loop within a loop is typically $O(n^2)$
  - When its not obvious, use induction
- **Caution**! Beware of very large constant factors.
  - An algorithm with running-time 1,000,000 n is still O(n) but might be less efficient on your data set than one with running-time $2n^2$, which is $O(n^2)$.
  - Remember "Big-Oh" only works for values of n greater than or equal to some constant value.



Some exercises:

- *Show that*

if f(n) is O(g(n)) and d(n) is O(h(n)),
then f(n)+d(n) is O(g(n)+h(n))

Recall the definition of the “Big-Oh” Notation: we need constants $c>0$ and $n_0 \geq 1$ such that

$f(n) + d(n) \leq c(g(n) + h(n))$ for every integer $n \geq n_0$. $f(n)$ is $O(g(n))$ means that there exists $c_f > 0$ and an integer $n_{0f} \geq 1$ such that

$f(n) \leq c_f\, g(n)$ for every $n \geq n_{0f}$.

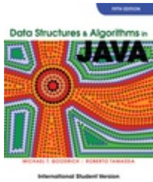
Similarly, $d(n)$ is $O(h(n))$ means that there exists $c_d > 0$ and an integer $n_{0d} \geq 1$ such that

$d(n) \leq c_d\, g(n)$ for every $n \geq n_{0d}$.

Let $n_0 = \max(n_{0f}, n_{0d})$, and $c = \max(c_f, c_d)$.

So $f(n) + d(n) \leq c_f g(n) + c_d h(n) \leq c(g(n) + h(n))$ for $n \geq n_0$.

Therefore $f(n)+d(n)$ is $O(g(n)+h(n))$.



- *Show that*

$$3(n+1)^7 + 2n\log n \text{ is } O(n^7)$$

- $\log n$ is $O(n)$
- $2n\log n$ is $O(2n^2)$
- $3(n+1)^7$ is a polynomial of degree 7, therefore it is $O(n^7)$
- $3(n+1)^7 + 2n\log n$ is $O(n^7 + 2n^2)$ [based on previous proof]
- $3(n+1)^7 + 2n\log n$ is $O(n^7)$

- Algorithm A executes *10nlogn* operations, while algorithm B executes n^2 operations. Determine the minimum integer value n_0 such that A executes fewer operations than B for $n \geq n_0$

Any IDEAS??

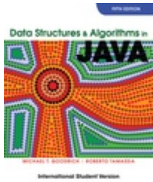
We must find the minimum integer n_0 such that

$10n\log n < n^2$

Since n describes the size of the input data set that the algorithms operate upon, it will always be positive. Since n is positive, we may factor an n out of both sides of the inequality, giving us

$10\log n < n$.

Let us consider the left and right hand side of this inequality. These two functions have one intersection point for $n > 1$, and it is located between $n = 58$ and $n = 59$. Indeed, $10\log 58 = 58.57981 > 58$ and $10\log 59 = 58.82643 < 59$. So for $1 \leq n \leq 58$, $10n\log n \geq n^2$, and for $n \geq 59$, $10n\log n < n^2$. So n_0 we are looking for is 59.



Algorithm Foo (a, n):

Input: two integers, a and n

Output: ?

k ← 0

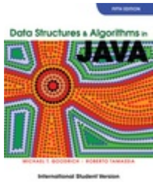
b ← 1

while k < n **do**

k ← k + 1

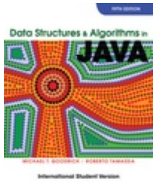
b ← b * a

return b



The algorithm computes a^n. The running time of this algorithm is O(n) because:

- *The initial assignments take constant time*
- *Each iteration of the while loop takes constant time*
- *There are exactly n iterations*



Algorithm Bar (a, n):

Input: two integers, a and n

Output: ?

k ← n

b ← 1

c ← a

while k > 0 **do**

if k mod 2 = 0 **then**

k ← k/2

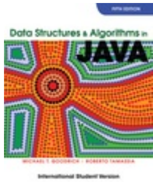
c ← c * c

else

k ← k – 1

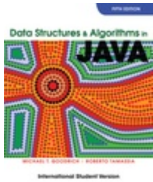
b ← b*c

return b



This algorithm computes a^n. Its running time is O(logn) for the following reasons:

- *The initialization and the **if** statement and its contents take constant time, so we need to figure out how many times the **while** loop gets called. Since k goes down (either gets halved or decremented by one) at each step, and it is equal to n initially, at worst the loop gets executed n times. But we can (and should) do better in our analysis.*
- *Note that if k is even, it gets halved, and if it is odd, it gets decremented, and halved in the next iteration. So at least every second iteration of the **while** loop halves k. One can halve a number n at most [logn] times before it becomes **≤** 1 (each time we halve a number we shift it right by one bit, and a number has [logn] bits). If we decrement a number in between halving it, we still get to halve no more than [logn] times. Since we can only decrement k in between two halving operations (unless n is odd or it is the last iteration) we get to do a decrementing iteration at most [logn] + 2 times. So we can have at most 2[logn] + 2 iterations. This is obviously O(logn).*



- $5n^4 + 3n^3 + 2n^2 + 4n + 1$

$5n^4 + 3n^3 + 2n^2 + 4n + 1$ is $O(n^4)$. Note that $5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq (5+3+2+4+1)n^4 = cn^4$ for $c = 15$, when $n \geq n_0 = 1$

- $5n^2 + 3n\log n + 2n + 5$

$5n^2 + 3n\log n + 2n + 5$ is $O(n^2)$. Note that $5n^2 + 3n\log n + 2n + 5 \leq (5+3+2+5)n^2 = cn^2$ for $c = 15$, when $n \geq n_0 = 2$ (note that $n\log n$ is zero for $n=1$)

- $20n^3 + 10n\log n + 5$

$20n^3 + 10n\log n + 5$ is $O(n^3)$. Note that $20n^3 + 10n\log n + 5 \leq 35n^3$ for $n \geq n_0 = 1$.

- $3\log n + 2$

$3\log n + 2$ is $O(\log n)$. Note that $3\log n + 2 \leq 5\log n$, for $n \geq 2$ (note that $\log n$ is zero for $n=1$. That is why we use $n \geq n_0 = 2$ in this case)

- 2^{n+2}

2^{n+2} is $O(2^n)$. Note that $2^{n+2} = 2^n\, 2^2 = 4 \cdot 2^n$; hence, we can take $c = 4$ and $n_0 = 1$ in this case

- $2n + 100\log n$

$2n + 100\log n$ is $O(n)$. Note that $2n + 100\log n \leq 102n$, for $n \geq n_0 = 2$; hence we take $c = 102$ in this case (1 mark)