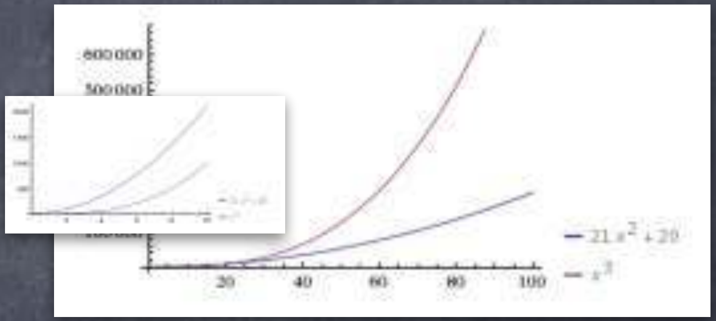
Asymptotics

The Big O



How it scales

- In analysing running time (or memory/power consumption) of an algorithm, we are interested in how it scales as the problem instance grows in "size"
  - Running time on small instances of a problem are often not a serious concern (anyway small)
- Also, exact time/number of steps is less interesting
  - Can differ in different platforms. Not a property of the algorithm alone.
  - Thus "unit of time" (constant factors) typically ignored when analysing the algorithm.

How it scales

- e.g., suppose number of "steps" taken by an algorithm to sort a list of n elements varies between $3n$ and $3n^2+9$ (depending on what the list looks like)
  - If n is doubled, time taken in the worst case could become (roughly) 4 times. If n is tripled, it could become (roughly, in the worst case) 9 times
  - An upper bound that grows "like" n^2
- Typically, interested in easy to interpret guarantees
  - Resource required expressed as a function of input size
  - Upper bounds robust to constant factor speed ups

Upper-bounds: Big O

- T(n) has an upper-bound that grows "like" f(n)
  - **$T(n) = O(f(n))$**

    $\exists c, k > 0, \quad \forall n \geq k, \quad 0 \leq T(n) \leq c \cdot f(n)$

    Unfortunate notation! An alternative used sometimes: $T(n) \in O(f(n))$

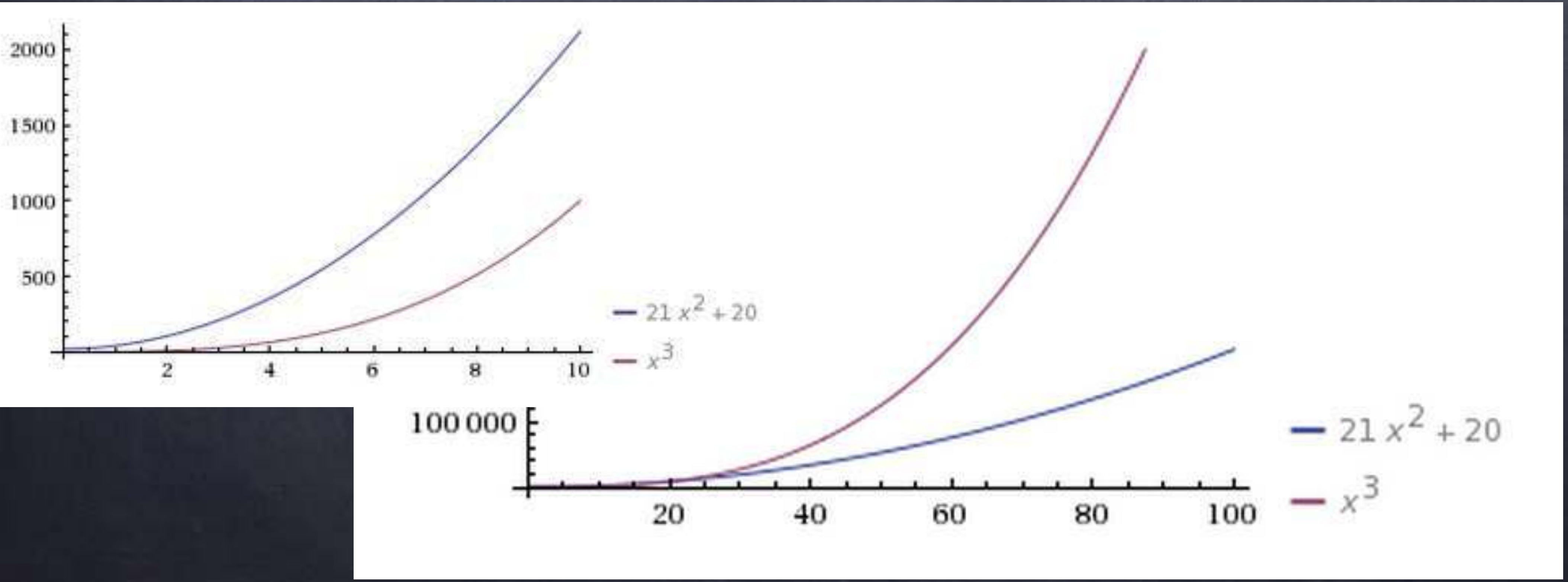
    - Note: we are defining it only for T & f which are eventually non-negative
    - Note: order of quantifiers! c can't depend on n (that is why c is called a constant factor)
- Important: If $T(n)=O(f(n))$, f(n) could be much larger than T(n) (but only a constant factor smaller than T(n))

$T(n) = O(f(n))$

$\exists c, k > 0, \quad \forall n \geq k, \quad 0 \leq T(n) \leq c \cdot f(n)$

Upper-bounds: Big O

- e.g. $T(x) = 21x^2 + 20$
- $T(x) = O(x^3)$



$T(n) = O(f(n))$

$\exists c, k > 0, \ \forall n \geq k, \ 0 \leq T(n) \leq c \cdot f(n)$

Upper-bounds: Big O

- e.g. $T(x) = 21x^2 + 20$
- $T(x) = O(x^3)$
- $T(x) = O(x^2)$ too, since we allow scaling by constants
- But $T(x) \neq O(x)$.
  - $\forall c>0, \forall k>0, \exists x^* \geq k \ \ T(x^*) > c.x^*$

$T(n) = O(f(n))$
$\exists c, k > 0, \ \forall n \geq k, \ 0 \leq T(n) \leq c \cdot f(n)$

Upper-bounds: Big O

- Used in the analysis of running time of algorithms: Worst-case Time(input size) = O(f(input size))
  - e.g. $T(n) = O(n^2)$, $T(n) = O(n \log n)$
- Also used to bound approximation errors
  - e.g., $|\log(n!) - \log(n^n)| = O(n)$
    - A better approximation: $|\log(n!) - \log((n/e)^n)| = O(\log n)$
    - Even better: $|\log(n!) - \log((n/e)^n) - \frac{1}{2} \cdot \log(n)| = O(1)$
- We may also have $T(n) = O(f(n))$, where f is a decreasing function (especially when bounding errors)
  - e.g. $T(n) = O(1/n)$

$T(n) = O(f(n))$

$\exists c, k > 0, \quad \forall n \geq k, \quad 0 \leq T(n) \leq c \cdot f(n)$

Big O: Some Properties

- Suppose $T(n) = O(f(n))$ and $R(n) = O(f(n))$
  - i.e., $\forall n \geq k_T,\ 0 \leq T(n) \leq c_T \cdot f(n)$ and $\forall n \geq k_R,\ 0 \leq R(n) \leq c_R \cdot f(n)$
  - $T(n) + R(n) = O(f(n))$
    - Then, $\forall n \geq \max(k_T, k_R), \quad 0 \leq T(n)+R(n) \leq (c_R+c_T) \cdot f(n)$
  - If eventually ($\forall n \geq k$), $R(n) \leq T(n)$, then $T(n) - R(n) = O(T(n))$
    - $\forall n \geq \max(k, k_R),\ 0 \leq T(n) - R(n) \leq 1 \cdot T(n)$
- If $T(n) = O(g(n))$ and $g(n) = O(f(n))$, then $T(n) = O(f(n))$
    - $\forall n \geq \max(k_T, k_g),\ 0 \leq T(n) \leq c_T \cdot g(n) \leq c_T c_g \cdot f(n)$
- e.g., $7n^2 + 14n + 2 = O(n^2)$ because $7n^2$, $14n$, 2 are all $O(n^2)$
- More generally, if $T(n)$ is upper-bounded by a degree d polynomial with a positive coefficient for n^d, then $T(n) = O(n^d)$

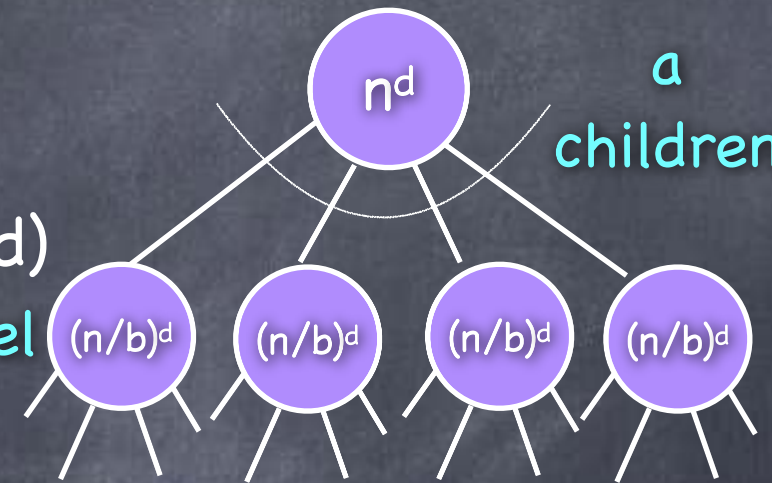
$T(n) = O(f(n))$
$\exists c, k > 0, \forall n \geq k, 0 \leq T(n) \leq c \cdot f(n)$

Some important functions

- $T(n) = O(1)$: $\exists c$ s.t. $T(n) \leq c$ for all ~~sufficiently large~~ n
- $T(n) = O(\log n)$. $T(n)$ grows quite slowly, because log n grows quite slowly (when n doubles, log n grows by 1)
- $T(n) = O(n)$: $T(n)$ is (at most) linear in n
- $T(n) = O(n^2)$: $T(n)$ is (at most) quadratic in n
- $T(n) = O(n^d)$ for some fixed d: $T(n)$ is (at most) polynomial in n
- $T(n) = O(2^{d \cdot n})$ for some fixed d: $T(n)$ is (at most) exponential in n. $T(n)$ could grow very quickly.

A General Solution (a.k.a. "Master Theorem")

- $T(n) = a\,T(n/b) + c \cdot n^d$ (and $T(1)=1$. $a \geq 1, b > 1$ integer, $c>0$, $d \geq 0$ real.)



- Say $n=b^k$ (so only integers encountered)
- #levels $= \log_b n = k$

total at this level $= a \cdot (n/b)^d$

- $T(n) = O(\, n^d \,(\, 1 + (a/b^d) + \ldots + (a/b^d)^k \,)$ total at i^{th} level $= a^i \cdot (n/b^i)^d$
- If $a = b^d$, contribution at each level $= n^d$. $T(n) = O(n^d \cdot \log n)$
- If $a < b^d$: $1 + (a/b^d) + (a/b^d)^2 + \ldots = O(1)$. $T(n) = O(n^d)$
- If $a > b^d$: $(a/b^d)^k[1 + (b^d/a) + (b^d/a)^2 + \ldots\,] = O((a/b^d)^k) = a^k/n^d$
$T(n) = O(a^k) = O(2^{k \cdot \log a}) = O(2^{\log n \cdot \log a/\log b}) = O(n^{\log_b a})$

Tight Bounds: Theta Notation

- If we can give a "tight" upper and lower-bound we use the Theta notation
  - $T(n) = \Theta(f(n))$ if $T(n)=O(f(n))$ and $f(n)=O(T(n))$
  - e.g., $3n^2-n = \Theta(n^2)$
- If $T(n) = \Theta(f(n))$ and $R(n) = \Theta(f(n))$, $T(n) + R(n) = \Theta(f(n))$

$\simeq$ and $\ll$

- Asymptotically equal: $f(n) \simeq g(n)$ if $\lim_{n\to\infty} f(n)/g(n) = 1$
  - i.e., eventually, f(n) and g(n) are equal (up to lower order terms)
  - If $\exists c>0$ s.t. $f(n) \simeq c \cdot g(n)$ then $f(n) = \Theta(g(n))$
    (for f(n) and g(n) which are eventually positive)
- Asymptotically much smaller: $f(n) \ll g(n)$ if $\lim_{n\to\infty} f(n)/g(n) = 0$
  - If $f(n) \ll g(n)$ then $f(n) = O(g(n))$ but $f(n) \neq \Theta(g(n))$
    (for f(n) and g(n) which are eventually positive)
- Note: Not necessary conditions: Θ and O do not require the limit to exist (e.g., $f(n) = n$ for odd n and $2n$ for even n: then $f(n) = \Theta(n)$)