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.
So, interested in how a function scales with its input: behaviour on large values, up to constant factors.

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” $n^2$. 
Upper-bounds: Big O

- T(n) has an upper-bound that grows “like” f(n)

- \( T(n) = O(f(n)) \)

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

- 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))
Big-O

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

- $\forall c>0, \forall k>0, \exists x^* \geq k \quad T(x^*) > c \cdot x^*$
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)

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)
Big O examples

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, \( \forall n \geq \max(k_T, k_R), 0 \leq T(n) + R(n) \leq (c_R + c_T) \cdot f(n) \)

If eventually (\( \forall n \geq k \)), \( R(n) \geq 0 \), then \( T(n) - R(n) = O(T(n)) \)

\( \forall n \geq \max(k, k_R), T(n) - R(n) \leq 1 \cdot T(n) \)

If \( T(n) = O(f(n)) \) and \( f(n) = O(g(n)) \), then \( T(n) = O(g(n)) \)

\( \forall n \geq \max(k_T, k_f), 0 \leq T(n) \leq c_T \cdot f(n) \leq c_T c_f \cdot g(n) \)

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 \( T(n) = O(n^d) \)
Some important functions

- $T(n) = O(1)$: $\exists c \text{ s.t. } T(n) \leq c$ 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.
Below \( n \) denotes the number of nodes in a complete and full 3-ary rooted tree and \( h \) its height. Which of the following is/are true, when considering \( h \) as a function of \( n \), and \( n \) as a function of \( h \)?

1. \( h = O(\log_3 n) \)
2. \( h = O(\log_2 n) \)
3. \( n = O(3^h) \)
4. \( n = O(2^h) \)

A. 1 & 3 only
B. 2 & 4 only
C. 1, 3 & 4 only
D. 1, 2 & 3 only
E. 1, 2, 3 & 4
If we can give a “tight” upper and lower-bound we use the Theta notation

\[ T(n) = \Theta(f(n)) \text{  if  } T(n) = O(f(n)) \text{  and  } f(n) = O(T(n)) \]

e.g., \( 3n^2 - n = \Theta(n^2) \)

If \( T(n) = \Theta(f(n)) \) and \( R(n) = \Theta(f(n)) \), then \( T(n) + R(n) = \Theta(f(n)) \)
Which of the following is/are true?

1. If \( f(x) = O(g(x)) \) and \( g(x) = O(h(x)) \) then \( f(x) = O(h(x)) \)
2. If \( f(x) = O(g(x)) \) and \( h(x) = O(g(x)) \) then \( f(x) = O(h(x)) \)
3. If \( f(x) = \Theta(g(x)) \) and \( h(x) = \Theta(g(x)) \) then \( f(x) = \Theta(h(x)) \)

A. 1 only  
B. 1 & 2 only  
C. 3 only  
D. 1 & 3 only  
E. 1, 2 & 3
Asymptotically equal: $f(n) \approx g(n)$ 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. $f(n) \approx c \cdot g(n)$ 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: $\Theta$ and $O$ do not require the limit to exist (e.g., $f(n) = n$ for odd $n$ and $2n$ for even $n$: then $f(n) = \Theta(n)$)
Analysing Algorithms

- Analyse correctness and running time (or other resources)
  - Latter can be quite complicated

- Behaviour depends on the particular inputs, but we often restrict the analysis to worst-case over all possible inputs of the same “size”
  - Size of a problem is defined in some natural way (e.g., number of elements in a list to be sorted, number of nodes in a graph to be coloured, etc.)
  - Generically, could define as number of bits needed to write down the input
Loops

If an algorithm is “straight-line” without loops or recursion, its running time would be $O(1)$

Need to analyse how many times a loop is taken

e.g. find max among $n$ numbers in an array $L$

```c
findmax(L,n) {
    max = L[1]
    for i = 2 to n {
        if (L[i] > max)
            max = L[i]
    }
    return max
}
```

Time taken by $\text{findmax}(L,n)$
$T(n) = O(n)$
Nested Loops

If an outer-loop is executed $p$ times, and each time an inner-loop is executed $q$ times, the code inside the inner-loop is executed $p \cdot q$ times in all.

More generally, the number of times the inner-loop is taken can be different in different executions of the outer-loop.

e.g.

```plaintext
for i = 1 to n {
    for j = 1 to i {
        tap-fingers()
    }
}
```

what all values of $(i,j)$ are possible when we get here?

- $i=1$: $j=1$.
- $i=2$: $j=1,2$.
- $i=3$: $j=1,2,3$.
- ... $i=n$: $j=1,2,..,n$.

$1 + 2 + 3 + ... + n = \frac{n(n+1)}{2} = O(n^2)$
Loops

\[
i = 1
\text{ while } i \leq n \{
    \text{ for } j = 1 \text{ to } n \{
        \text{ tap-fingers() }
    \}
    i = 2 \times i
\}
\]

\[
i = 1
\text{ while } i \leq n \{
    \text{ for } j = 1 \text{ to } i \{
        \text{ tap-fingers() }
    \}
    i = 2 \times i
\}
\]

\[
i = 1, 2, 4, \ldots, 2^{\log n} \quad (j=1,2,\ldots,n \text{ always})
O(n \log n)
\]

\[
i = 1, 2, 4, \ldots, 2^{\log n} \quad \text{but} \quad j=1,\ldots,i
1 + 2 + 4 + \ldots + 2^{\log n} = O(n)
\]

Number of nodes in a complete & full binary rooted tree with (about) n leaves