Efficient algorithms

When do we call an algorithm efficient?

It should use as few resources as possible:

- Running time
- Space (memory)
- Messages (networking protocols)
- Disk accesses (computations on massive data)
- ...

We focus on running time in this course.

1st definition attempt: When implemented, the algorithm should run as fast as possible on real inputs.

Pros:
- Captures what we really care about

Cons:
- Depends on quality of implementation and speed of machine
- What are "real inputs"?
- Does not capture dependence on input size.

Want:
- Platform independence (hardware, OS, programming language, implementation skills, ...)

  Ideal goal: Pick the best algorithm for all platforms before making the effort to implement it.

- Instance independence (particular properties of the input)

We want a performance measure that succinctly characterizes the algorithm's behaviour on all inputs.

- Dependence on input size

  All algorithms are efficient for small inputs. The slower the running time grows with the input size the larger the inputs we can handle.

Platform independence = model of computation (abstract platform)

A model of computation specifies which operations an algorithm is allowed to use and how expensive each such operation is.

$\Rightarrow$ Analyzing running time = counting operations

Trade-off: Simple model $\Rightarrow$ easy analysis
Detailed model $\Rightarrow$ accurate reflection of true running time

Sophisticated algorithms are almost impossible to analyze using complicated models (limited cognitive capacity). We want a model that is simple enough to allow us to reason about clever algorithms yet is detailed enough to mean something on real machines.

Random Access Machine (RAM) model

Permissible operations

- Addition, subtraction, multiplication, division
- Logical operations: and, or, not
- Comparisons
- Conditionals
- Loops
- Random memory access using integers as address

Each operation takes the same constant amount of time

- Model is certainly simple
- Set of operations rich enough to reflect most real machines
- Uniform cost of operations is (as we will see) a reasonable approximation most of the time

When isn't the approximation good enough?

- often, multiplication and division are more expensive than other operations
- Memory accesses cost more than arithmetic operations
- Caches lead to possibly huge ($\approx 10^6$) differences in the cost of individual memory accesses

Instance independence

Insertion sort		
for $i = 2$ to n	$1 + 4 \cdot n$	
do $x = A[i]$	$3 \cdot (n-1)$	
$j = i - 1$	$3 \cdot (n-1)$	
while $j > 0$ and $A[j] > x$	$6 \cdot (n-1)$	$\sum_{i=2}^{n} (i-1)6$
do $A[j+1] = A[j]$		$\sum_{i=2}^{n} (i-1)7$
$j = j - 1$		
$A[j+1] = x$	$4 \cdot (n-1)$	

Running time on sorted input: $20n - 15$

Running time on reverse sorted input: $6.5n^2 + 13.5n - 15$

How do we unify this into one function of the input size?

Best-case running time = function $T(\cdot)$ s.t. $T(n) =$ min running time over all inputs of size n

Worst-case running time = function $T(\cdot)$ s.t. $T(n) =$ max running time over all inputs of size n

Average-case running time = function $T(\cdot)$ s.t. $T(n) =$ average running time over all inputs of size n
= expected running time assuming each input is equally likely

Best-case running time does not really provide any performance guarantees for arbitrary inputs $\Rightarrow$ poor choice.

Worst-case running time provides the strongest possible guarantee: there is no input for which the algorithm exceeds the predicted running time. Designing an algorithm with good worst-case running time may be substantially harder than designing one with good average-case running time. It depends on the application whether an average-case guarantee is good enough.

Example: Quicksort

Worst-case quicksort is much more complicated and orders of magnitude slower than simple quicksort.

```
Simple QuickSort (A, i, j)
  if j ≤ i then return
  x = A[i]
  k = Partition (A, i, j, x)
  Simple QuickSort (A, i, k)
  Simple QuickSort (A, k+1, j)
```

```
Partition (A, i, j, x)
  l = i-1
  r = j+1
  loop
     do l = l+1 while A[l] < x
     do r = r-1 while A[r] > x
     if l < r then swap A[l] and A[r]
              else return r
```

Worst-case running time:
$\sim n^2$

Best-case running time:
$\sim n \lg n$

Average-case running time:
$\sim n \lg n$
(we'll prove this later)

```
WorstCaseQuickSort(A, i, j)
 if j ≤ i then return
 x = Select(A, i, j, ⌈(j-i+1)/2⌉)
 k = Partition(A, i, j, x)
 WorstCaseQuickSort(A, i, k)
 WorstCaseQuickSort(A, k+1, j)
```

```
Select(A, i, j, k)
 if i = j then return A[i]
 n = ⌊(j-i)/5⌋ + 1
 for h = 0 to n-1 do
    if i+5·h+4 ≤ j then
       InsertionSort(A, i+5·h, i+5·h+4)
       B[h+1] = A[i+5·h+2]
    else B[h+1] = A[i+5·h]
 x = Select(B, 1, n, ⌈n/2⌉)
 ℓ = Partition(A, i, j, x)
 if k ≤ ℓ then return Select(A, i, ℓ, k)
 else return Select(A, ℓ+1, j, k-ℓ)
```

Running time (best, average, worst case): $\sim n \lg n$

In practice much slower than SimpleQuickSort!

How to choose between the two?

If inputs are approximately uniform random permutations, you will never see SimpleQuickSort take more than $n \lg n$ time in your lifetime.

If inputs are nearly sorted most of the time, SimpleQuicksort almost always takes n^2 time. In this case, WorstCaseQuickSort or MergeSort is the better choice.

$\Rightarrow$ Knowledge of the application requirements matters.

Another alternative: Avoid bad behaviour of SimpleQuicksort while keeping it simple by picking a random pivot. (We'll discuss this version later.)

Back to defining efficiency, 2nd attempt: An algorithm is efficient if, at an analytical level, its worst-case performance is better than that of the brute-force method.

Pros:
- Platform-independent ("analytical")
- Instance-independent ("worst-case")
- Captures dependence on input size (why?)
- Captures that we should be intelligent in solving the problem

Cons:
- Beating the brute force method is not a major hurdle
- How do we compare algorithms that both beat the brute force method?

Final definition of efficiency: The algorithm's (worst-case) running time should be polynomial in the input size.

Motivation: Doubling the input size should increase the running time by only a constant factor.

Questions:

- Is n^{100} efficient? No.
- Is $n^{1+0.02 \lg n}$ inefficient? No.

Justification of our definition: Overwhelmingly, polynomial-time algorithms are fast in practice and exponential-time algorithms are not.

Asymptotic running time

Some practical considerations:

- Figuring out exact running times and comparing them is difficult
- Given that the RAM model approximates real machines, can we really say that an algorithm with running time $5n$ is faster than one that takes $6n$ time? No. The types of operations they perform matter.

$\Rightarrow$ Determining constant factors makes our life hard and provides little benefit.

- Almost any algorithm is fast for small inputs. What matters is how the running time increases for larger inputs.

$\Rightarrow$ We prefer algorithm A over algorithm B if $T_A(n) < T_B(n) \ \forall n \geq n_0$, where n_0 is a certain minimum input size.

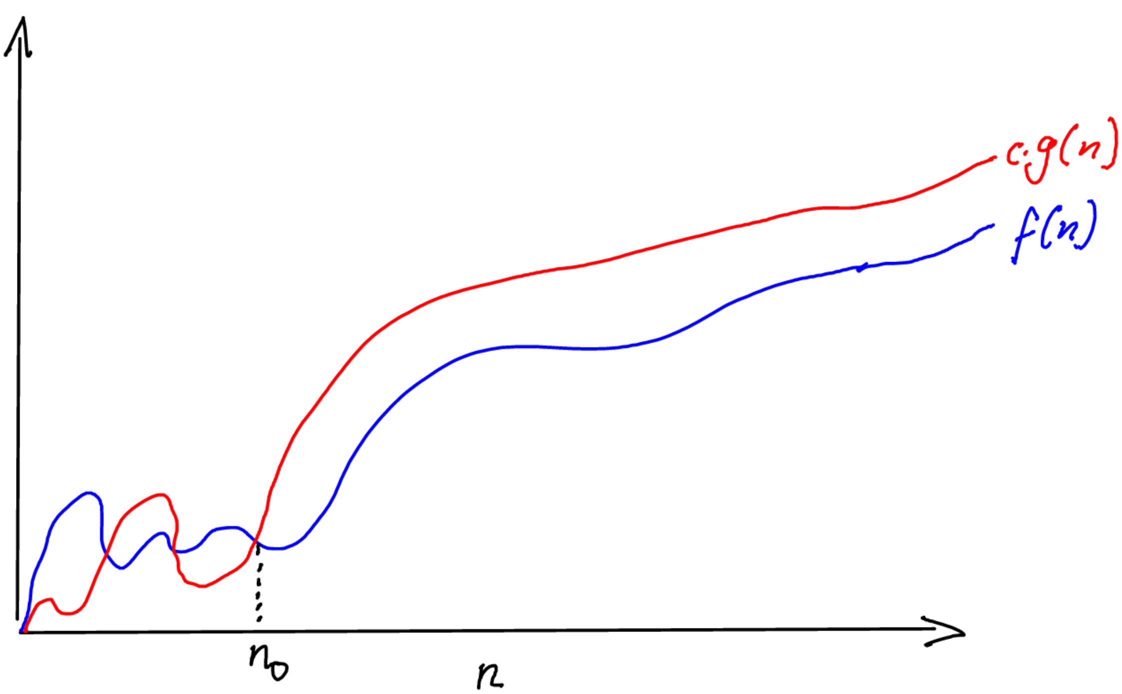
... or, since constants don't matter:

... if $T_A(n) < c \cdot T_B(n) \ \forall n \geq n_0$, for some $c > 0$.

This leads us to the following definitions for comparing the asymptotic growth of functions (running times):

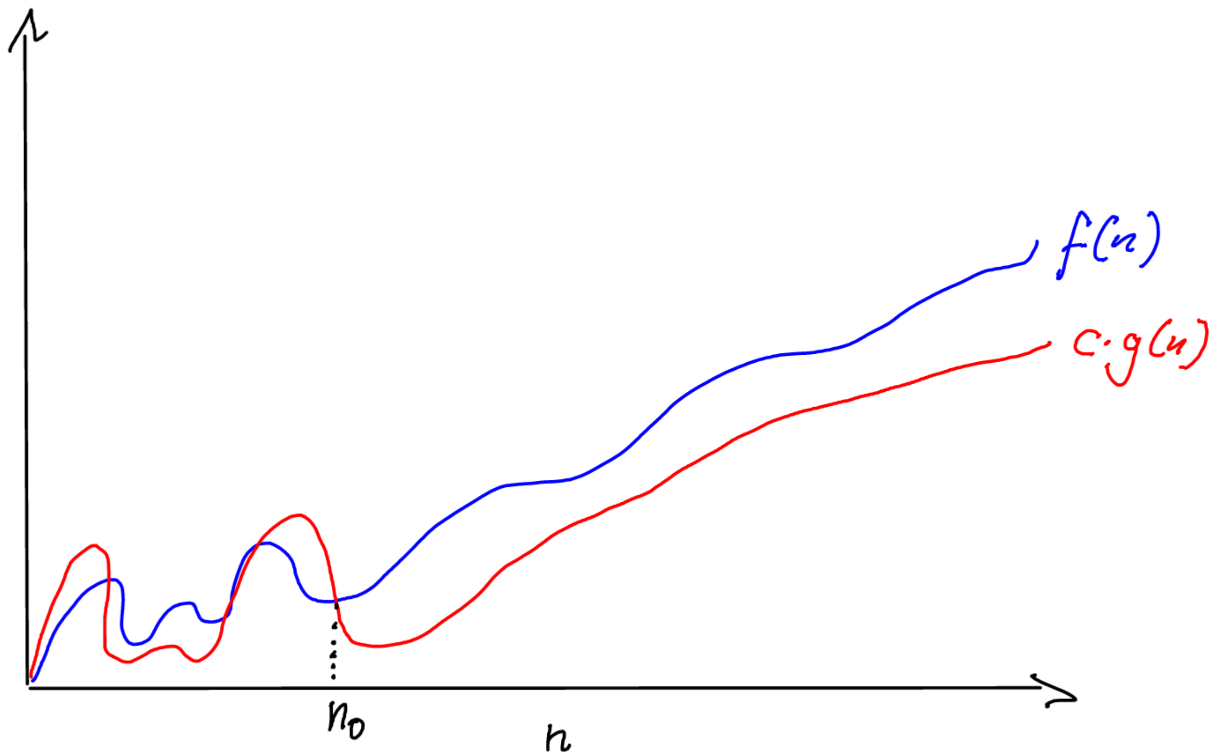
$f(n) \in O(g(n)) \Leftrightarrow \exists c > 0, n_0 \geq 0 \ \forall n \geq n_0 : f(n) \leq c \cdot g(n)$

($O(g(n))$ is the set of functions that grow at most a constant factor faster than $g(n)$. $O(n)$, for example is the set of all linear and sublinear functions.)



$f(n) \in \Omega(g(n)) \Leftrightarrow \exists c>0, n_0 \geq 0 \; \forall n \geq n_0: f(n) \geq c \cdot g(n)$

($\Omega(g(n))$ is the set of functions that grow at most a constant factor slower than $g(n)$. $\Omega(n^2)$ is the set of all quadratic and superquadratic functions.)



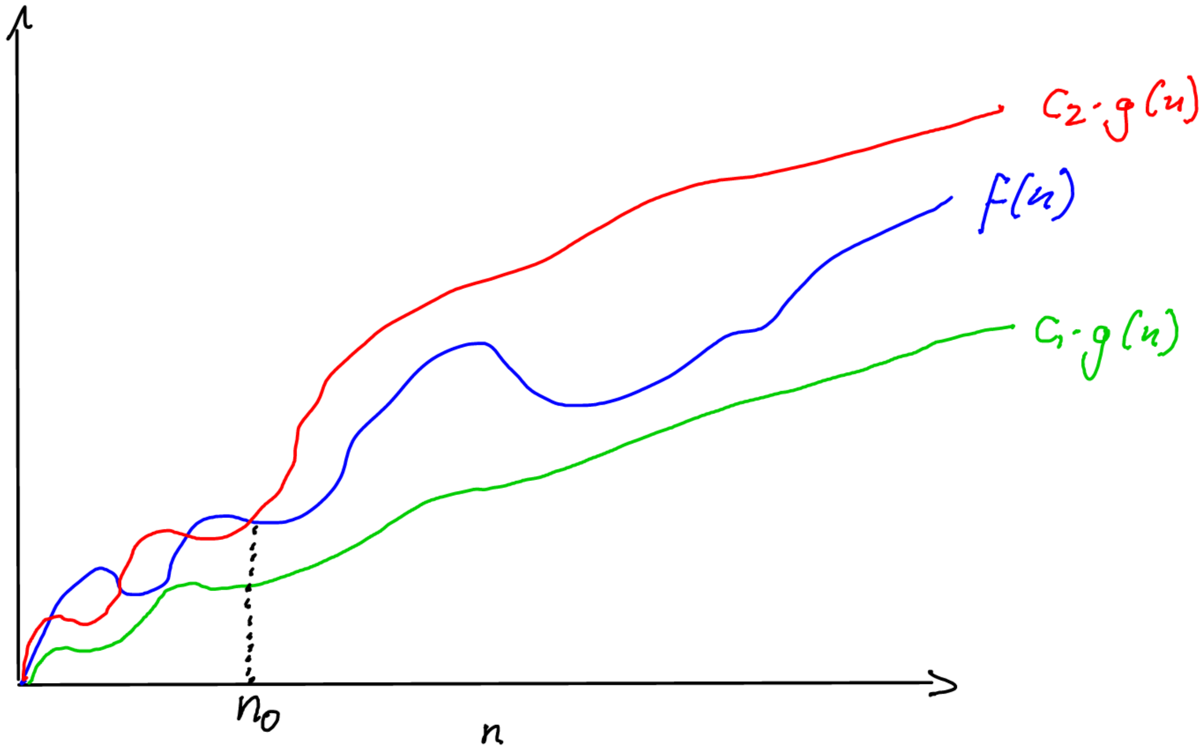
$f(n) \in \Theta(g(n)) \Leftrightarrow \exists c_1>0, c_2>0, n_0 \geq 0 \; \forall n \geq n_0:$

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

$\Leftrightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$

($\Theta(g(n))$ is the set of functions whose asymptotic growth differs from that of $g(n)$ by at most a constant factor. $\Theta(n)$ is the set of all linear functions.)

$$\Rightarrow \Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$



So, if $T_A(n)$ and $T_B(n)$ are running times of algorithms A and B, then

$T_A(n) \in O(T_B(n)) \iff$ A is no worse than B (by more than a constant factor).

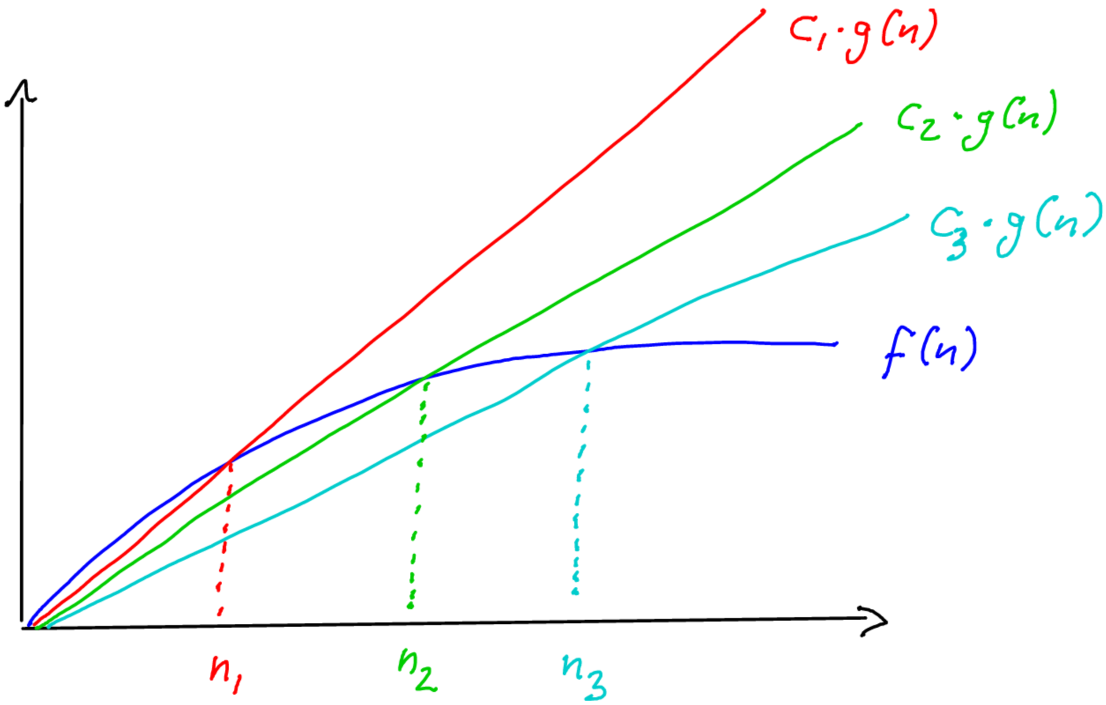
$T_A(n) \in \Omega(T_B(n)) \iff$ A is no better than B (by more than a constant factor).

$T_A(n) \in \Theta(T_B(n)) \iff$ A and B are equally efficient (up to constant factors)

How do we express that A is strictly better than B?

$f(n) \in o(g(n)) \iff \forall c>0\ \exists n_0 \geq 0\ \forall n \geq n_0 : f(n) < c \cdot g(n)$

($o(g(n))$ is the set of functions that grow by more than a constant factor slower than $g(n)$. $o(n)$ is the set of all sublinear functions.)



So, no matter the constant we multiply $g(n)$ with, there always exists an input size beyond which $f(n)$ will "dive below" that scaled version of $g(n)$.

$$f(n) \in \omega(g(n)) \iff \forall c>0\ \exists n_0 \geq 0\ \forall n \geq n_0 : f(n) \geq c \cdot g(n)$$
$$\iff g(n) \in o(f(n))$$

Sanity checks

1. We know linear time is better than $n \lg n$ time, so we should have $f(n) \in o(g(n))$ for $f(n) = 8n + 3\sqrt{n} - 4$ and $g(n) = 2n\lg n - 20n + 8$.

- $\sqrt{n} \leq n \quad \forall n \geq 1 \quad \Rightarrow f(n) \leq 11n \quad \forall n \geq 1$
- $\lg n \geq 20 \quad \forall n \geq 2^{20} \Rightarrow g(n) \geq n \lg n \quad \forall n \geq 2^{20}$
- $c \lg n \geq 11 \quad \forall n \geq 2^{\frac{11}{c}}$

$\Rightarrow \quad f(n) \leq 11n \leq c\, n \lg n \leq c \cdot g(n) \quad \forall n \geq \max(2^{20}, 2^{\frac{11}{c}})$

$\Rightarrow \quad f(n) \in o(g(n))$

2. We know $n \lg n$ time is better than quadratic time, so $f(n) \in o(g(n))$ for $f(n) = 80 n \lg n + 11n$ and $g(n) = n^2 - 40 n \lg n + 8n$

- $\lg n \geq 1 \quad \forall n \geq 2 \Rightarrow f(n) \leq 91 n \lg n \quad \forall n \geq 2$
- $40 \lg n \leq \frac{n}{2} \quad \forall n \geq 1024 \Rightarrow g(n) \geq \frac{1}{2} n^2 \quad \forall n \geq 1024$
- $91 \lg n \leq \frac{c}{2} n \quad \forall n \geq \max\left(e^2, \left(\frac{364}{c}\right)^2\right)$

$$\Rightarrow f(n) \leq 91 n \lg n \leq \frac{c}{2} n^2 \leq c \cdot g(n) \quad \forall n \geq \max\left(1024, \left(\frac{364}{c}\right)^2\right)$$

$$\Rightarrow f(n) \in o(g(n))$$

$$\lg n = \frac{\ln n}{\ln 2} \leq 2 \ln n$$

$$\Rightarrow 91 \lg n \leq \frac{c}{2} n \Leftarrow \ln n \leq \frac{c}{364} n = \frac{n}{a} \text{ for } a = \frac{364}{c}$$

For $n \geq a^2$ and $a \geq e$, we have

$$\ln n = \int_1^n \frac{1}{x} dx = \int_1^{a^2} \frac{1}{x} dx + \int_{a^2}^n \frac{1}{x} dx$$

$$= 2 \ln a + \int_{a^2}^n \frac{1}{x} dx$$

$$\leq a + \int_{a^2}^n \frac{1}{a} dx$$

$$= a + \frac{n}{a} - a$$

$$= \frac{n}{a}$$

A few simple facts

- $f(n) \in O(f(n)) \quad f(n) \in \Omega(f(n)) \quad f(n) \in \Theta(f(n))$
- $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
- $f(n) \in o(g(n)) \Leftrightarrow g(n) \in \omega(f(n))$
- $f(n) \in O(g(n)) \wedge g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$
  (Ditto for Ω, Θ, o, and ω)
- $f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n)) \Leftrightarrow f(n) \in \Theta(g(n))$
- $f_1(n) \in O(g_1(n)) \wedge f_2(n) \in O(g_2(n)) \Rightarrow$
  $f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$
  (Ditto for Ω, Θ, o, and ω)
- $f(n) \in O(g(n)) \Rightarrow f(n) + g(n) \in O(g(n))$

These help a lot:

- $6n\lg n + 20n + 9 \in O(n\lg n)$
  because $20n \in O(n\lg n)$
  $9 \in O(n\lg n)$
  $6n\lg n \in O(n\lg n)$

Asymptotic growth and limits

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \iff f(n) \in o(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0 \iff f(n) \in \Theta(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow a^{f(n)} \in o(a^{g(n)}) \quad \forall a > 1$

  $f(n) \in o(g(n)) \Rightarrow a^{f(n)} \in o(a^{g(n)}) \quad \forall a > 1$

However, $f(n) \in \Theta(g(n)) \not\Rightarrow a^{f(n)} \in \Theta(a^{g(n)})$

Asymptotic notation and algorithm performance

What does it mean when $T_A(n) \in O(T_B(n))$? Is algorithm A faster than algorithm B?

No. All we know is that A is at most a constant factor slower than B.

What does it mean when $T_A(n) \in o(T_B(n))$?

A may initially be slower than B but, as the input size increases, A will start to outperform B and the gap will grow with the input size.

This is true even if we run A on a 1980s style home computer with a 1MHz processor and B on a state-of-the-art Core i7.

Can we ignore constant factors as we do using asymptotic notation?

No. If $T_A(n) \in \Theta(T_B(n))$, we need to determine constant factors to choose between A and B. For this to be meaningful, we need to use more precise models that capture the relative cost of operations and exactly count the number of operations of each type.

Another alternative is to implement both algorithms and compare them experimentally.

Why do we use the RAM model and asymptotic analysis then?

It simplifies the analysis and allows a qualitative grouping of algorithms whose running times differ by at most constant factors. We would always prefer a linear-time algorithm over a quadratic one. If we have different linear-time algorithms for a given problem, these then become candidates for comparison using more detailed analysis techniques or experiments, but we avoid the effort to apply the same detailed analysis to the obviously inferior quadratic-time algorithms.

Back to the Stable Marriage Problem

Lemma: The Gale-Shapley algorithm can be implemented using only arrays and linked lists as data structures so that it runs in $O(n^2)$ time in the worst case.

Sketch:
- Maintain list of unmarried men using a doubly linked list

  ⇒ Testing whether there is an unmarried man, choosing one to make the next proposal, and adding rejected or divorced men to this list are $O(1)$ time operations.

- Every man stores the index of the next woman to propose to and his preference list in an array. Proposing becomes an $O(1)$ time operation.

- Every woman stores an inverted preference list in an array. For each man i, the list stores the rank of i in the woman's preference list. Comparing the ranks of two men now takes constant time, and this is all that's needed to decide whether to accept or reject a proposal.

⇒ Every iteration takes $O(1)$ time. Since there are $\leq n^2$ iterations, the total cost is $O(n^2)$.

The inverted preference list for each woman can be constructed in $O(n)$ time. Constructing n such lists thus takes $O(n^2)$ time.