Algorithms

Computational Efficiency

Hee-Kap Ahn
Graduate School of Artificial Intelligence
Dept. Computer Science and Engineering
Pohang University of Science and Technology (POSTECH)

Textbooks: Algorithms by Jeff Erickson /Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani.

Computational Efficiency

We design an algorithm (and data structures) that runs on a computer.

There can be two or more algorithms that solve the same problem. Among them, which algorithm is better? In other words, which algorithm runs faster?

Assuming that a basic operation ($+$, $-$, $*$,assign,...) takes constant time ($O(1)$),

how many basic operations does the algorithm execute in terms of the input size?

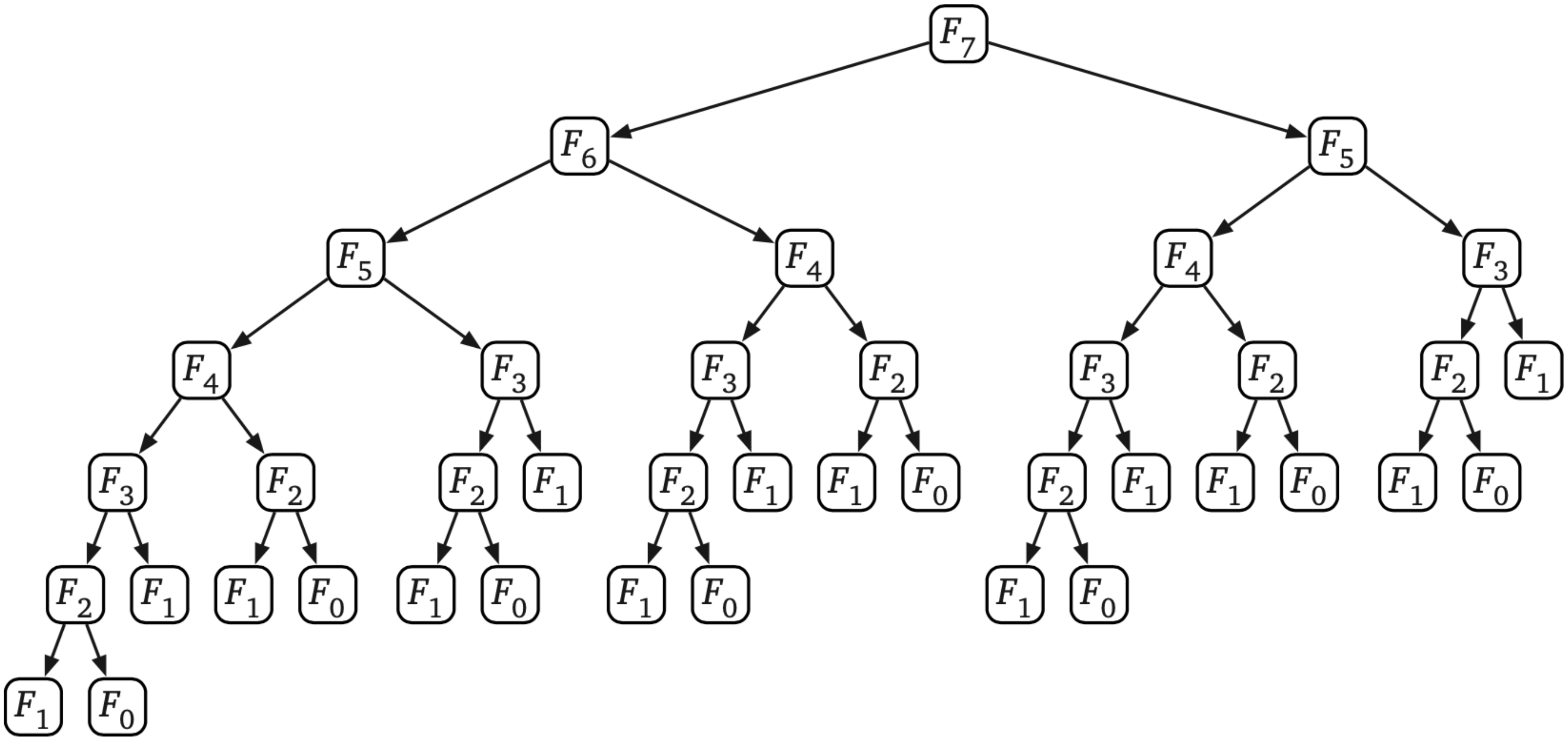
Fibonacci Numbers - FIB1

FIB1(n)
if $n = 0$ **then**
    return 0
else if $n = 1$ **then**
    return 1
return FIB1($n-1$) + FIB1($n-2$)

We always ask three questions.

- Is it correct?
- How much time does it take, as a function $T(n)$ of n?
- Can we do better?

Fibonacci Numbers - FIB1



Recursion tree for computing F_7 using FIB1. by Jeff Erickson

Fibonacci Numbers - FIB1

FIB1(n)
if $n = 0$ **then**
    return 0
else if $n = 1$ **then**
    return 1
return FIB1($n-1$) + FIB1($n-2$)

We always ask three questions.

- Is it correct?
- How much time does it take, as a function $T(n)$ of n?
- Can we do better?

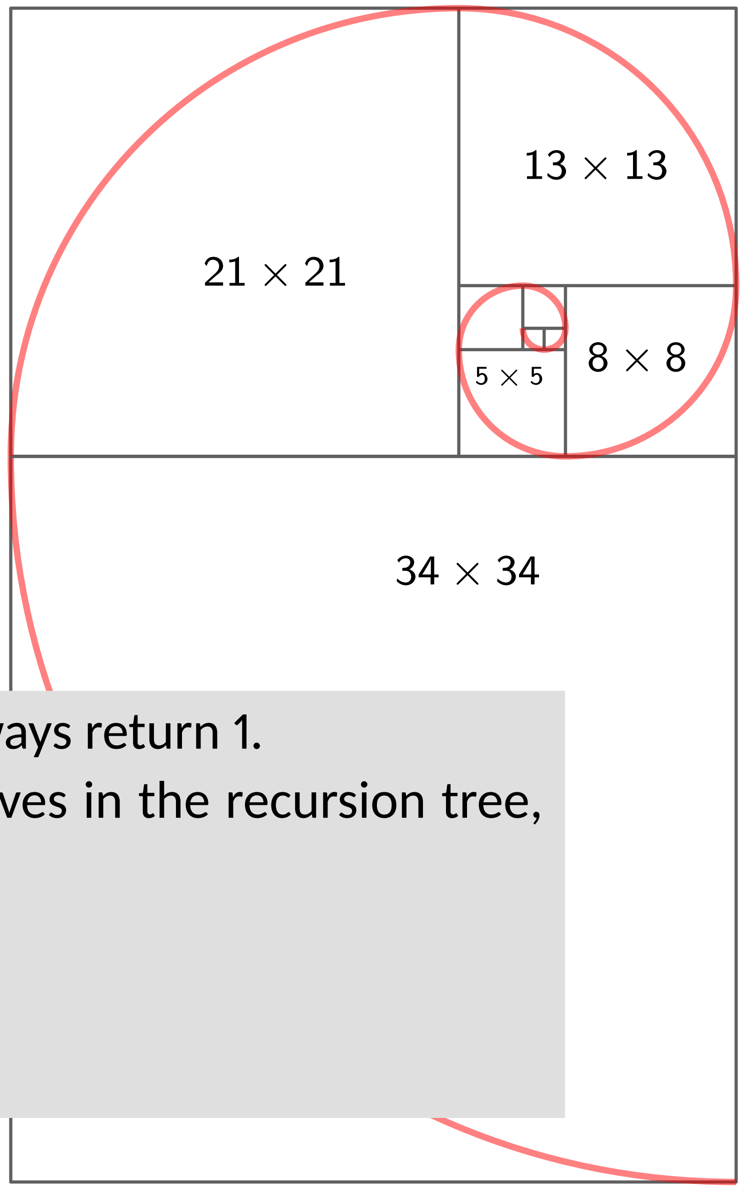
$$T(n) \leqslant 2 \text{ for } n \leqslant 1.$$

$$T(n) = T(n-1) + T(n-2) + O(1) \quad \text{for } n > 1.$$

$\rightarrow T(n) \geqslant F_n$. Thus, $T(n)$ grows as fast as F_n.

Fibonacci Numbers - FIB1

FIB1(n)
if $n = 0$ **then**
    return 0
else if $n = 1$ **then**
    return 1
return FIB1($n - 1$) + FIB1($n - 2$)



- The leaves of the recursion tree of FIB1 will always return 1.
- F_n is the sum of all values returned by the leaves in the recursion tree, which is the number of leaves in the tree.
- Each leaf will take $O(1)$ time to compute.
  $T(n) = F_n \times O(1)$.
- $T(n) \approx \Theta(1.6^n)$. ($\frac{1+\sqrt{5}}{2} = 1.6180339887\ldots$)

Fibonacci Numbers - FIB1

FIB1(n)
if $n = 0$ **then**
  return 0
else if $n = 1$ **then**
  return 1
return FIB1($n - 1$) + FIB1($n - 2$)

#. addition operations.

- FIB1(10) executes $\approx 1.6^{10} \approx 110$. FIB1(20): $\approx 1.6^{20} \approx 12089$.
- FIB1(50): $\approx 1.6^{50} \approx 16,069,380,443$.
- FIB1(100): $\approx 1.6^{100} \approx 258 \times 10^{18}$. Takes 12 days.
  (Fast computer can do 250×10^{12} arithmetic operations a second.
  Or 10^{30} operations in 10^{10} years.)

- FIB1(200):? FIB1(1000):?

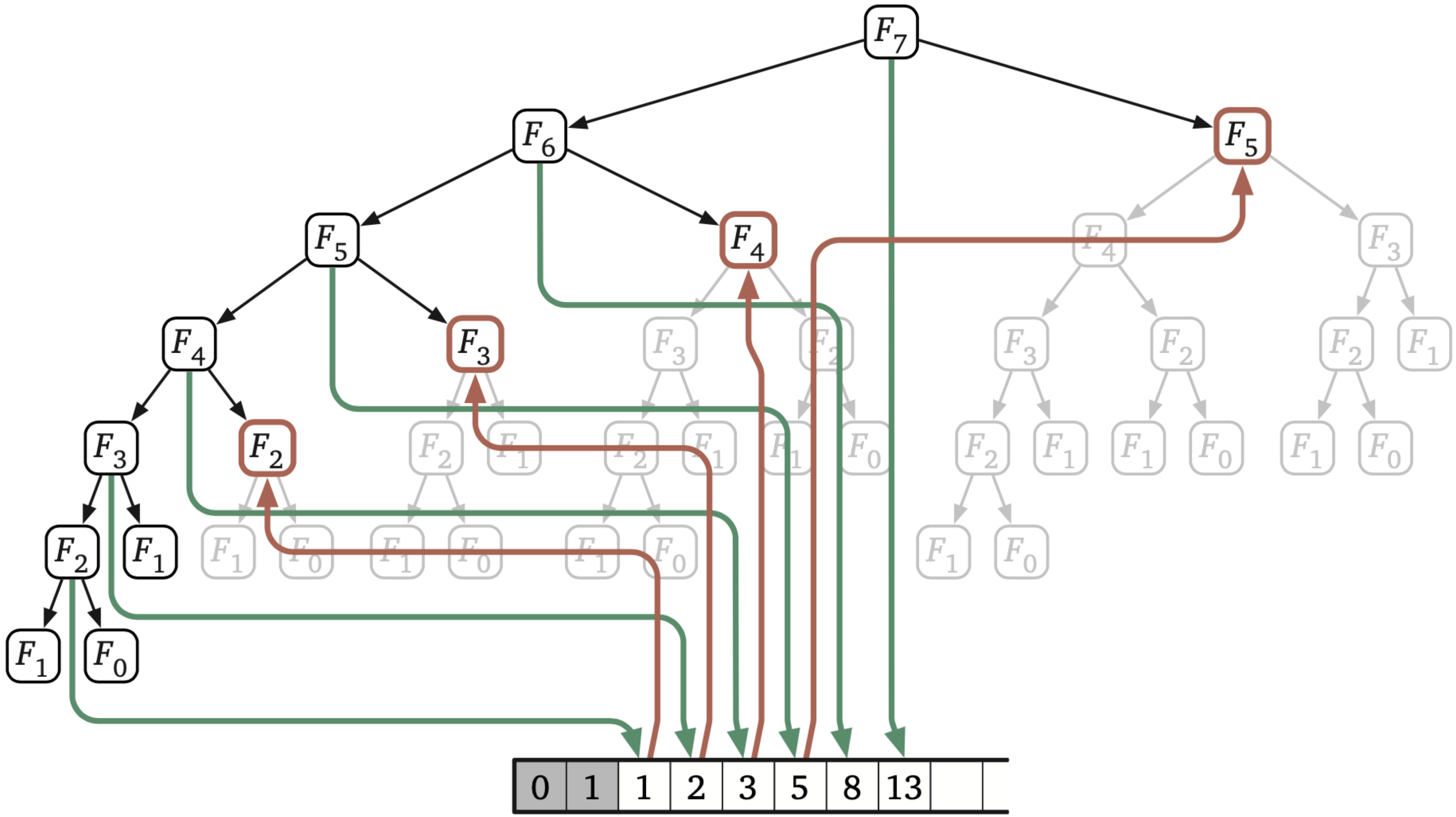
Fibonacci Numbers - FIB2

FIB2(n)
if $n = 0$ **then**
    return 0
create an array $f[0, \ldots, n]$
$f[0] = 0, f[1] = 1$
for $i = 2, \ldots, n$ **do**
    $f[i] = f[i-1] + f[i-2]$
return $f[n]$

We always ask three questions.

- Is it correct?
- How much time does it take, as a function $T(n)$ of n?
- Can we do better?

Fibonacci Numbers - FIB2



Recursion tree for computing F_7 using FIB2 with memoization. by Jeff Erickson

Fibonacci Numbers - FIB2

FIB2(n)
if $n = 0$ **then**
  return 0
create an array $f[0, \dots, n]$
$f[0] = 0, f[1] = 1$
for $i = 2, \dots, n$ **do**
  $f[i] = f[i-1] + f[i-2]$
return $f[n]$

We always ask three questions.

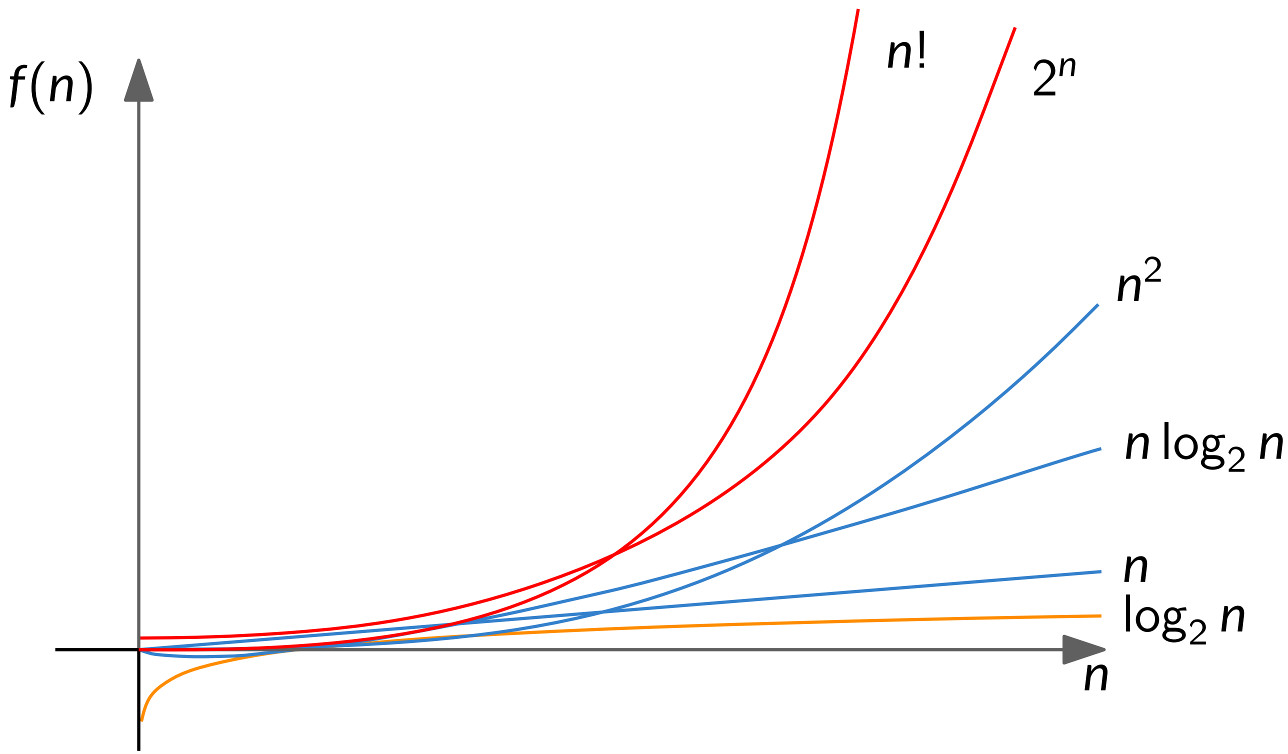
- Is it correct?
- How much time does it take, as a function $T(n)$ of n?
- Can we do better?

FIB2 uses $O(n)$ additions and stores $O(n)$ integers. Thus,

$$T(n) = O(n).$$

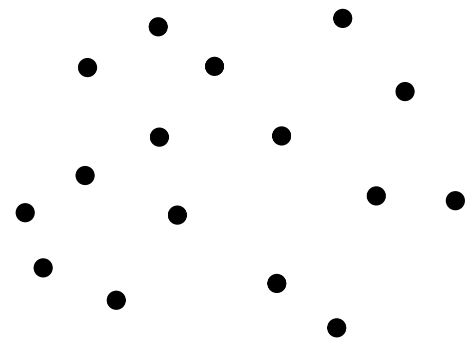
Growth Rates

	n	$n \log n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1s	< 1s	< 1s	< 1s	< 1s	< 1s	seconds
$n = 30$	< 1s	< 1s	< 1s	< 1s	< 1s	minutes	10^{25} yrs
$n = 100$	< 1s	< 1s	< 1s	< 1s	10K yrs	10^{17} yrs	NN
$n = 1,000$	< 1s	< 1s	< 1s	minutes	NN	NN	NN
$n = 1,000,000$	< 1s	seconds	days	30K yrs	NN	NN	NN



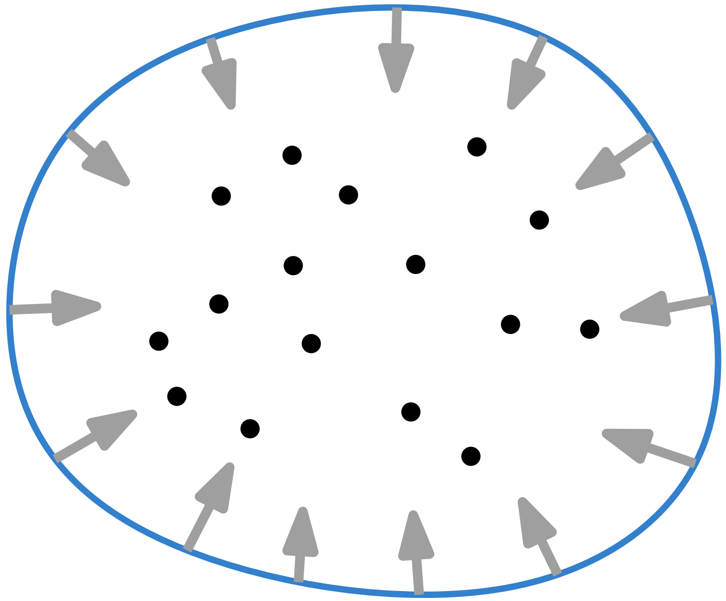
Convex Hulls

The convex hull of a set P of points in the plane is the *smallest* convex set containing P. Equivalently, it is the *largest* convex polygon whose vertices are points in P.



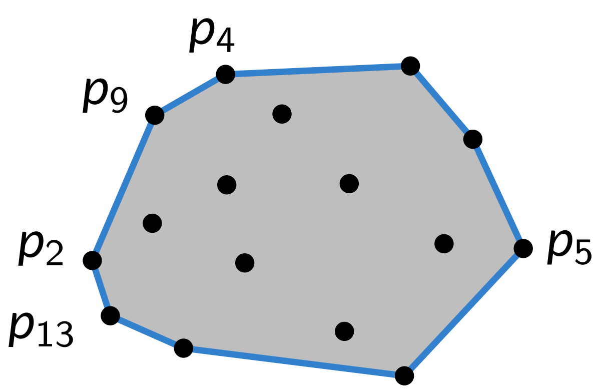
Convex Hulls

The convex hull of a set P of points in the plane is the *smallest* convex set containing P. Equivalently, it is the *largest* convex polygon whose vertices are points in P.



Convex Hulls

The convex hull of a set P of points in the plane is the *smallest* convex set containing P. Equivalently, it is the *largest* convex polygon whose vertices are points in P.



input. a set $P = \{p_1, p_2, \ldots, p_n\}$ of points, where $p_i = (x_i, y_i)$.
output. a representation of the convex hull.

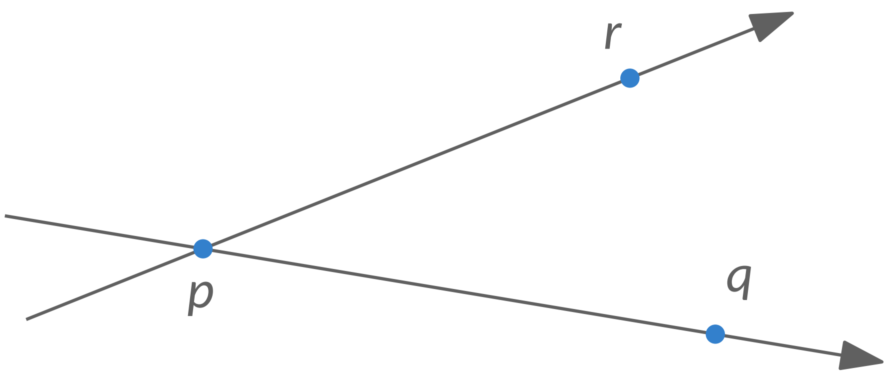
$p_2, p_9, p_4, \ldots, p_5, \ldots, p_{13}$

Convex Hulls

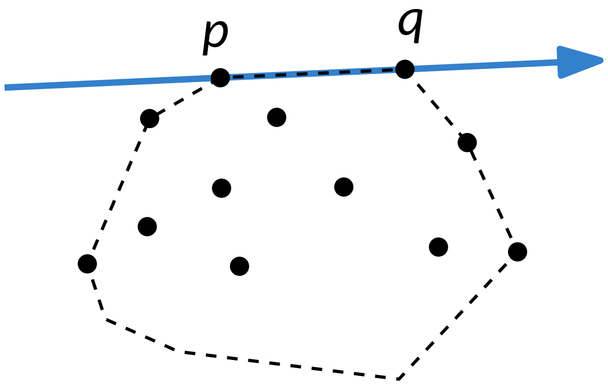
We assume that the points in P are in *general position*, meaning that no three points lie on a common line.

For three points p, q, r, how do we test whether r lies to the left or to the right of the directed line $\vec{pq}$?

$$r \text{ lies to the left of } \vec{pq} \text{ iff } (r_y - p_y)(q_x - p_x) > (q_y - p_y)(r_x - p_x).$$



Brute force. For all ordered pairs of points p and q, check whether the other points lie in the right side of $\overrightarrow{pq}$.

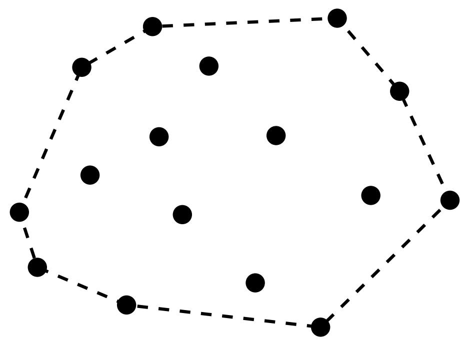


Running time $T(n) = \binom{n}{2} \times O(n) = O(n^3)$

Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

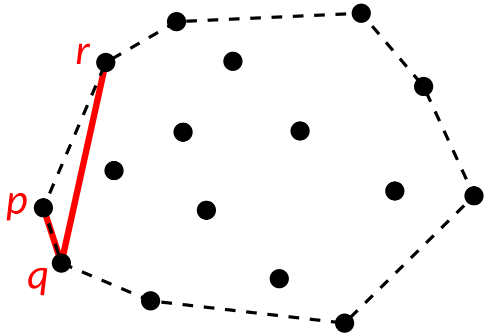
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

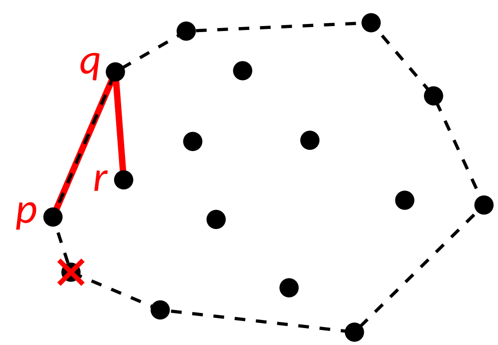
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

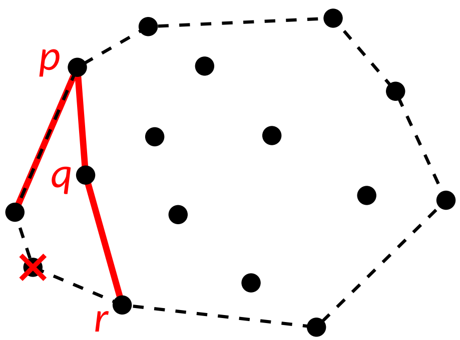
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

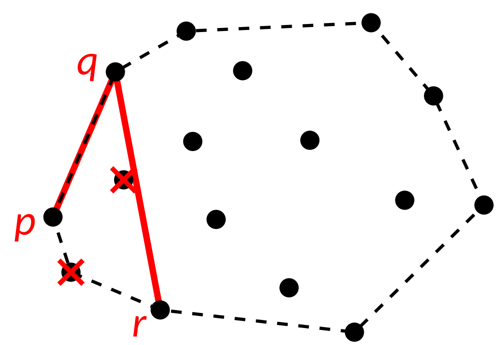
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

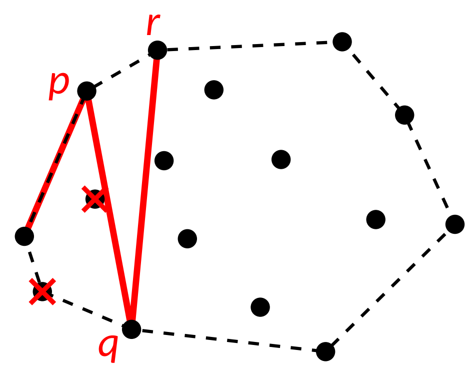
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

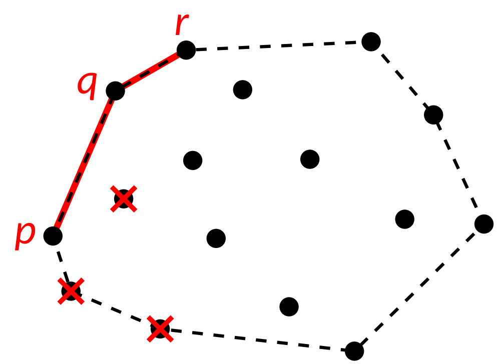
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.

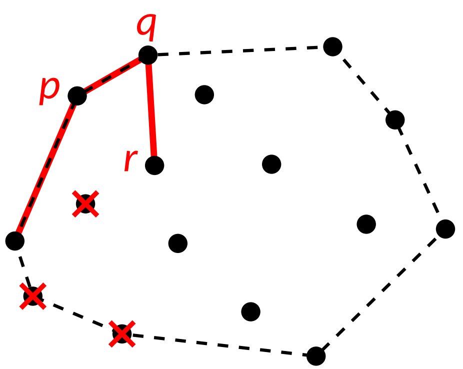


Convex Hulls

Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

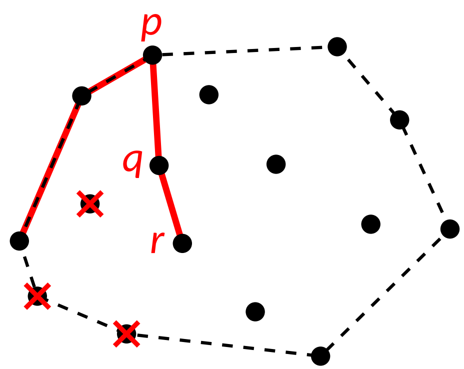
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

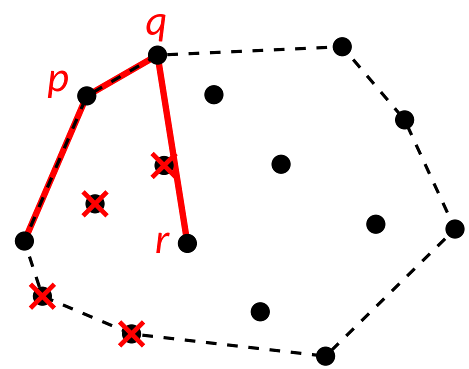
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

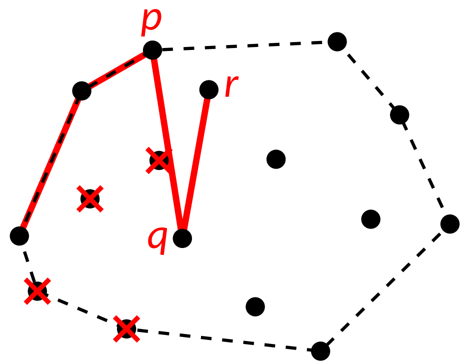
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

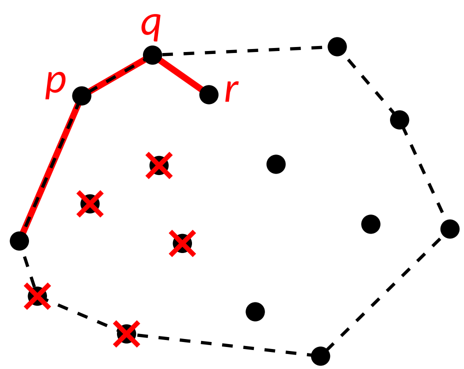
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

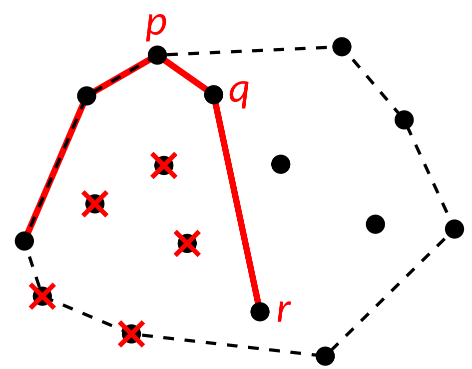
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

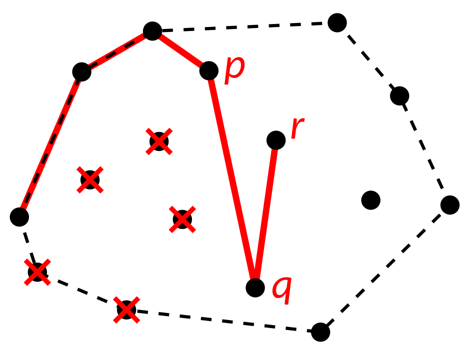
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

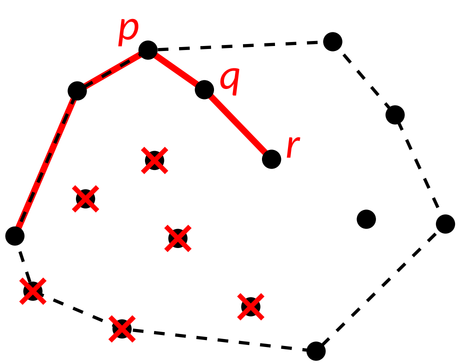
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

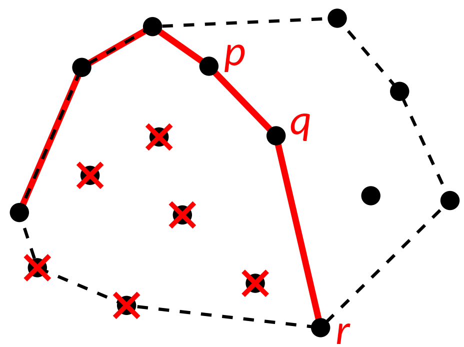
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

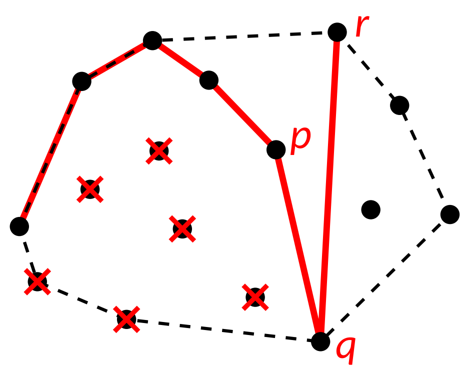
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

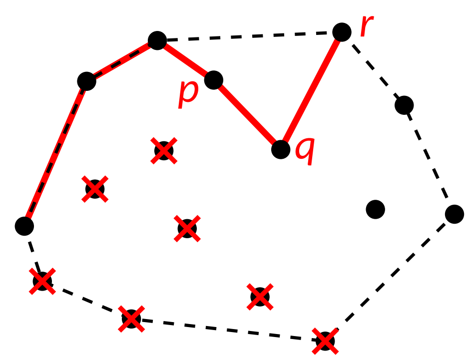
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

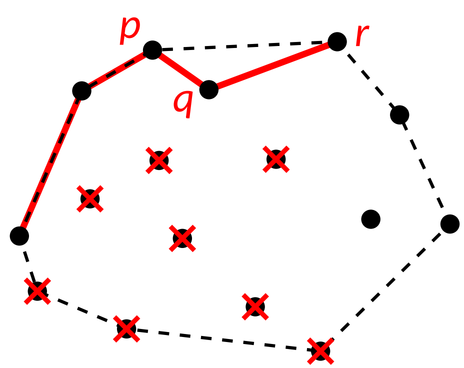
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

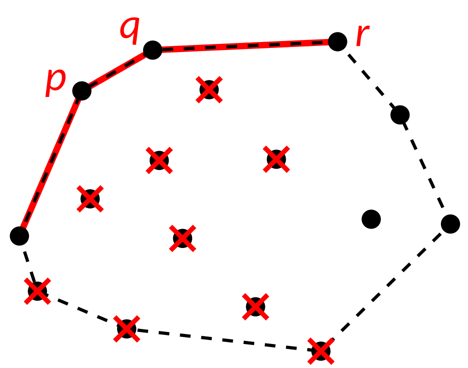
(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



Graham's scan.

- Sort points in P in $O(n \log n)$ time.
- Let p, q, r be the leftmost three points in the sorted list L.
  Repeat the followings.

(1) If r lies to the right of $\vec{pq}$, we move one step forward in L.
(2) Otherwise, remove q from L and move one step backward in L if possible.



- Whenever rule (1) is applied, r advances to the next point in L.
  So it is applied $n-2$ times. $O(n)$ time.
- Whenever rule (2) is applied, a point in L is removed.
  So it is applied $n-h$ times. $O(n)$ time. (h : #. vertices in the convex hull of P.)

$$T(n) = O(n \log n) + O(n) = O(n \log n).$$

O notation

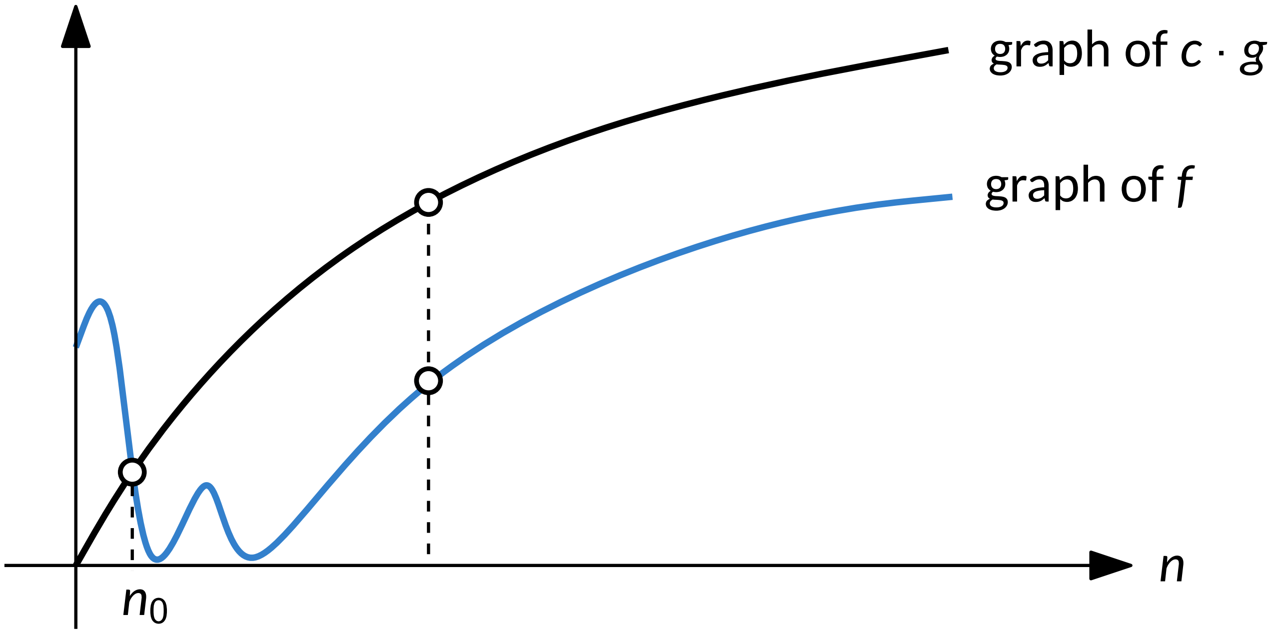
We typically measure the computational efficiency of an algorithm as the number of basic operations it performs as a function of its input length.

The efficiency of an algorithm can be captured by a function T from the set of natural numbers $\mathbb{N}$ to itself such that

> $T(n) =$ the maximum number of basic operations that the algorithm performs on inputs of length n.

O notation

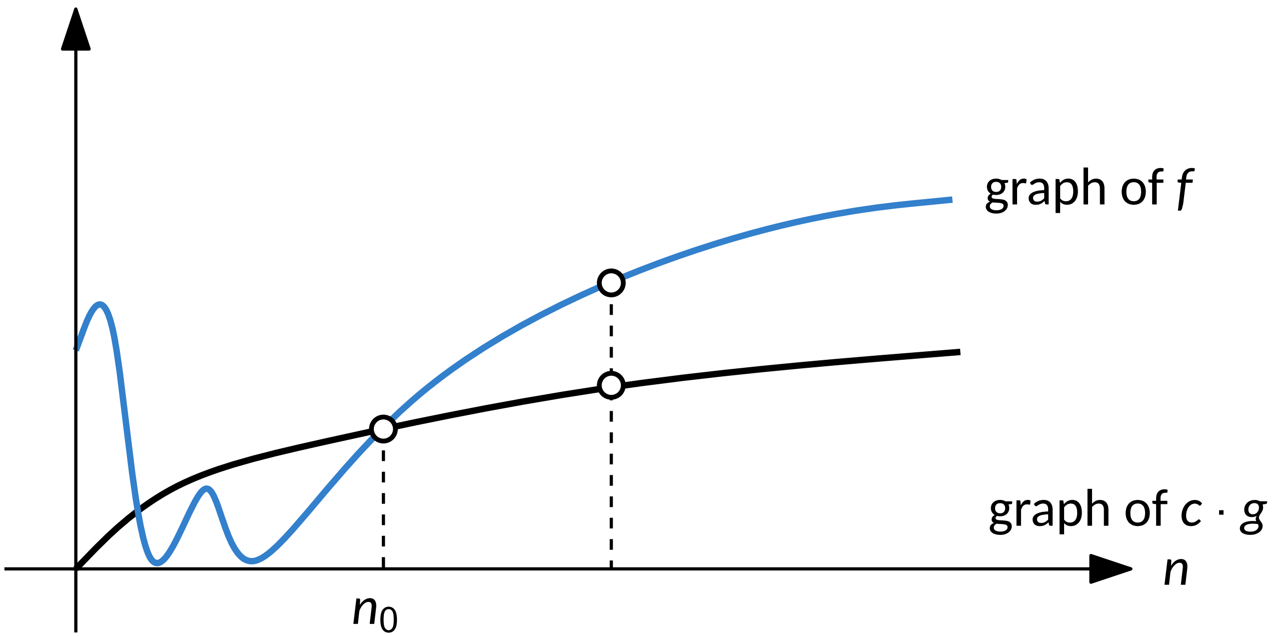
We say that for two functions $f(n)$ and $g(n)$,
$f(n)$ is $O(g(n))$ iff
there exist constants $c > 0$ and $n_0 \geqslant 0$ s.t. for all $n \geqslant n_0$ we have $f(n) \leqslant c \cdot g(n)$.



f grows no faster than g.

Ω notation

We say that for two functions $f(n)$ and $g(n)$,
$f(n)$ is $\Omega(g(n))$ iff
there exist constants $c > 0$ and $n_0 \geqslant 0$ s.t. for all $n \geqslant n_0$ we have $f(n) \geqslant c \cdot g(n)$.

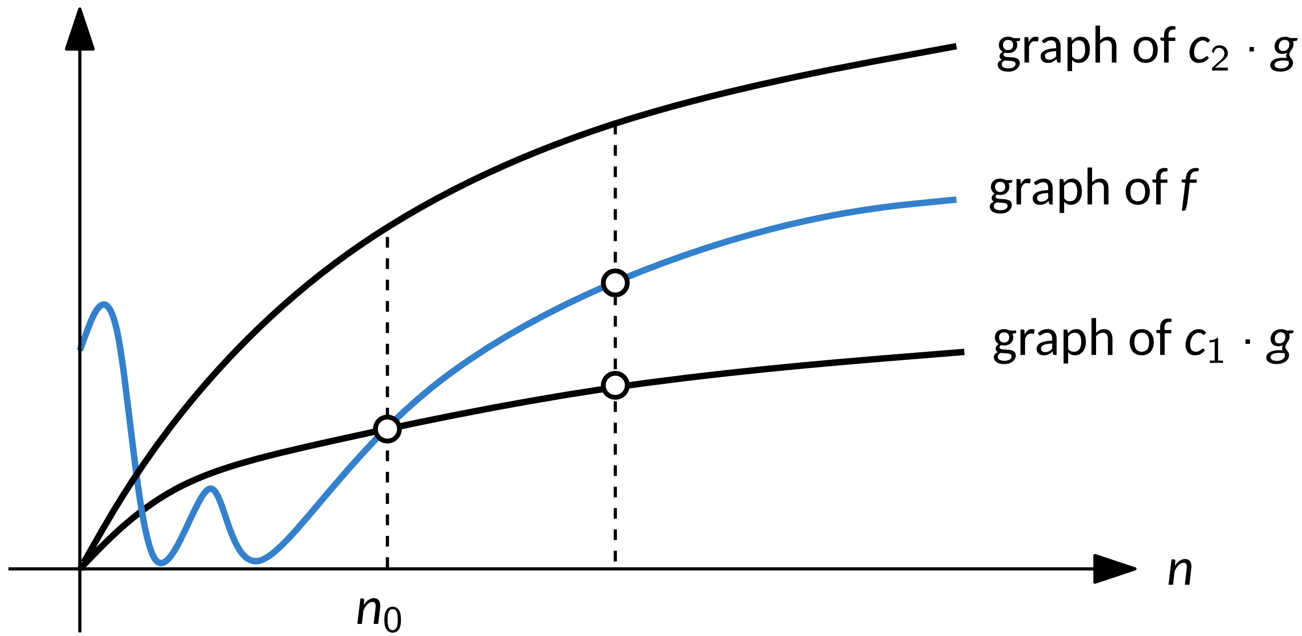


f grows at least as fast as g.

Θ notation

$f(n)$ is $\Theta(g(n))$ iff $f(n)$ is $\Omega(g(n))$ and $O(g(n))$.

There exists an $n_0 \geqslant 0$ and constants $c_1, c_2 > 0$ s.t.
for all $n \geqslant n_0$, $c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n)$.



f grows at the same rate as g.

Asymptotic Bounds

We say that

- $f = o(g)$
  iff for all $c > 0$, there exists an $n_0 > 0$ s.t. $f(n) \leqslant c \cdot g(n)$ for all $n \geqslant n_0$.
  f grows slower than g.

- $f = \omega(g)$ if $g = o(f)$.
  For all $c > 0$, there exists an $n_0 > 0$ s.t. $f(n) \geqslant c \cdot g(n)$ for all $n \geqslant n_0$.
  f grows faster than g.

Properties

Transitivity.

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

Additivity.

- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
- If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.
- If $f = \Theta(h)$ and $g = \Theta(h)$ then $f + g = \Theta(h)$.

Limits. $\lim_{n\to\infty} f(n)/g(n)$ reveals some asymptotic relationship between f and g, provided the limit exists.

- $\lim_{n\to\infty} f(n)/g(n) \neq \infty \implies f = O(g)$.
- $\lim_{n\to\infty} f(n)/g(n) \neq 0 \implies f = \Omega(g)$.
- $\lim_{n\to\infty} f(n)/g(n) \neq 0, \infty \implies f = \Theta(g)$.
- $\lim_{n\to\infty} f(n)/g(n) = 0 \implies f = o(g)$.
- $\lim_{n\to\infty} f(n)/g(n) = \infty \implies f = \omega(g)$.

Common Functions

Polynomials. $a_0 + a_1 n + \cdots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

Polynomial time. Running time is $O(n^d)$ for some constant d independent of the input size n.

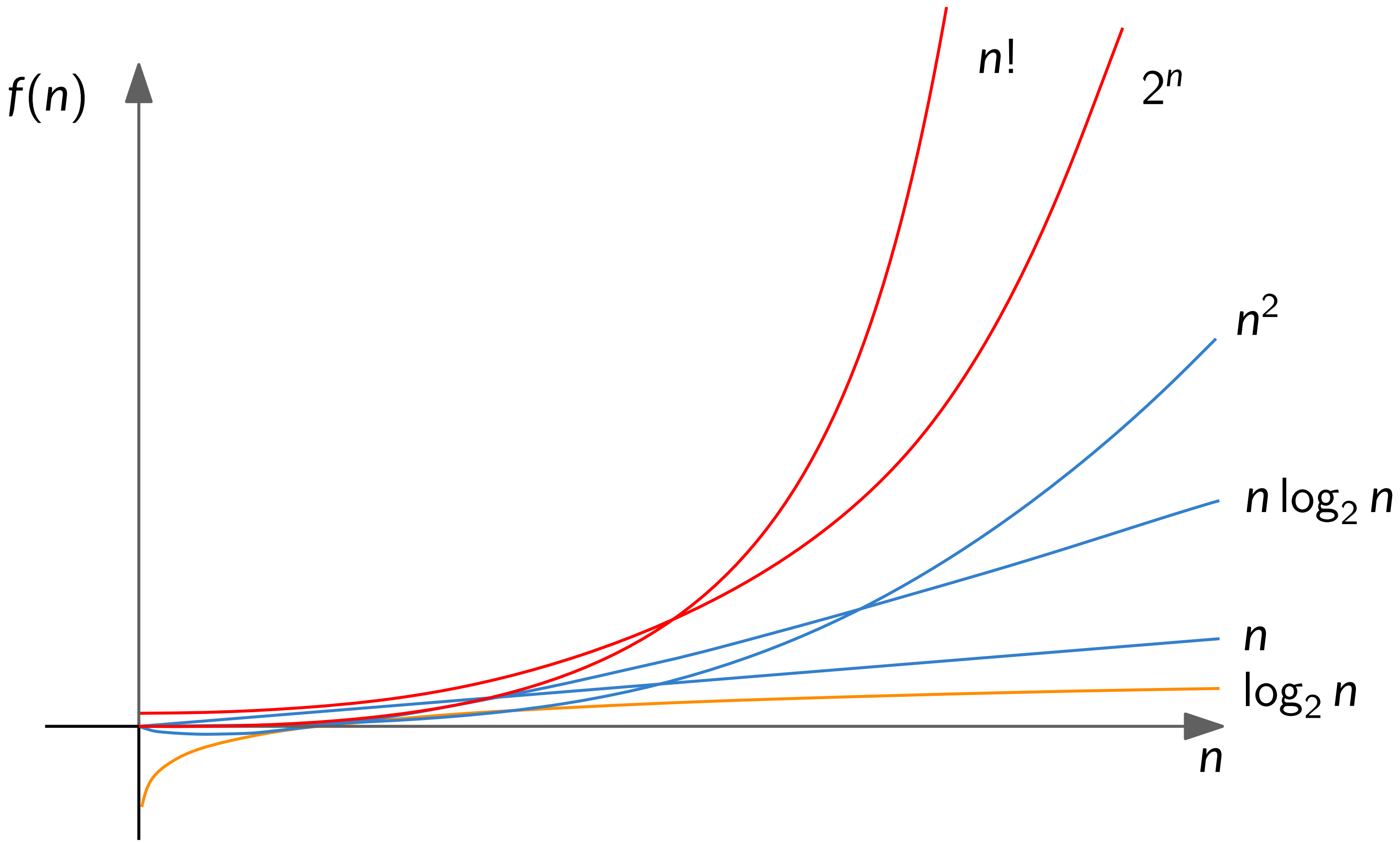
Logarithms. $O(\log_a n) = O(\log_b n)$ for any constants $a, b > 1$.

For every fixed constant $x > 0$, $\log n = O(n^x)$. In other words, every polynomial grows much faster than log.

Exponential. For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$.

every exponential grows faster than every polynomial.

Growth Rates



Properties

Some commonsense rules that help simplify functions:

- multiplicative constants can be omitted :
  $14n^2$ becomes n^2.
- n^a dominates n^b if $a > b$:
  n^2 dominates n.
- any exponential dominates any polynomial :
  3^n dominates n^5 (and 2^n).
- any polynomial dominates any logarithm :
  n dominates $\log^3 n$. n^2 dominates $n \log n$.

Still constants are important!