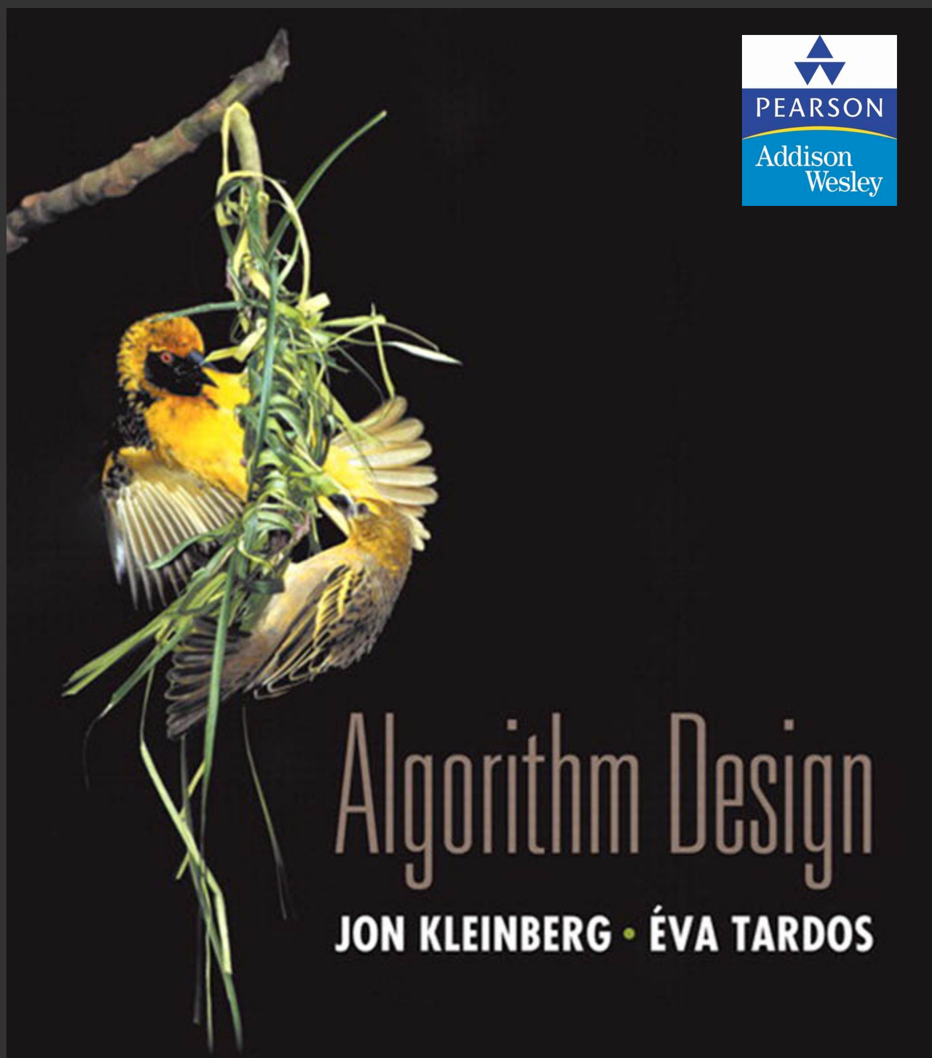
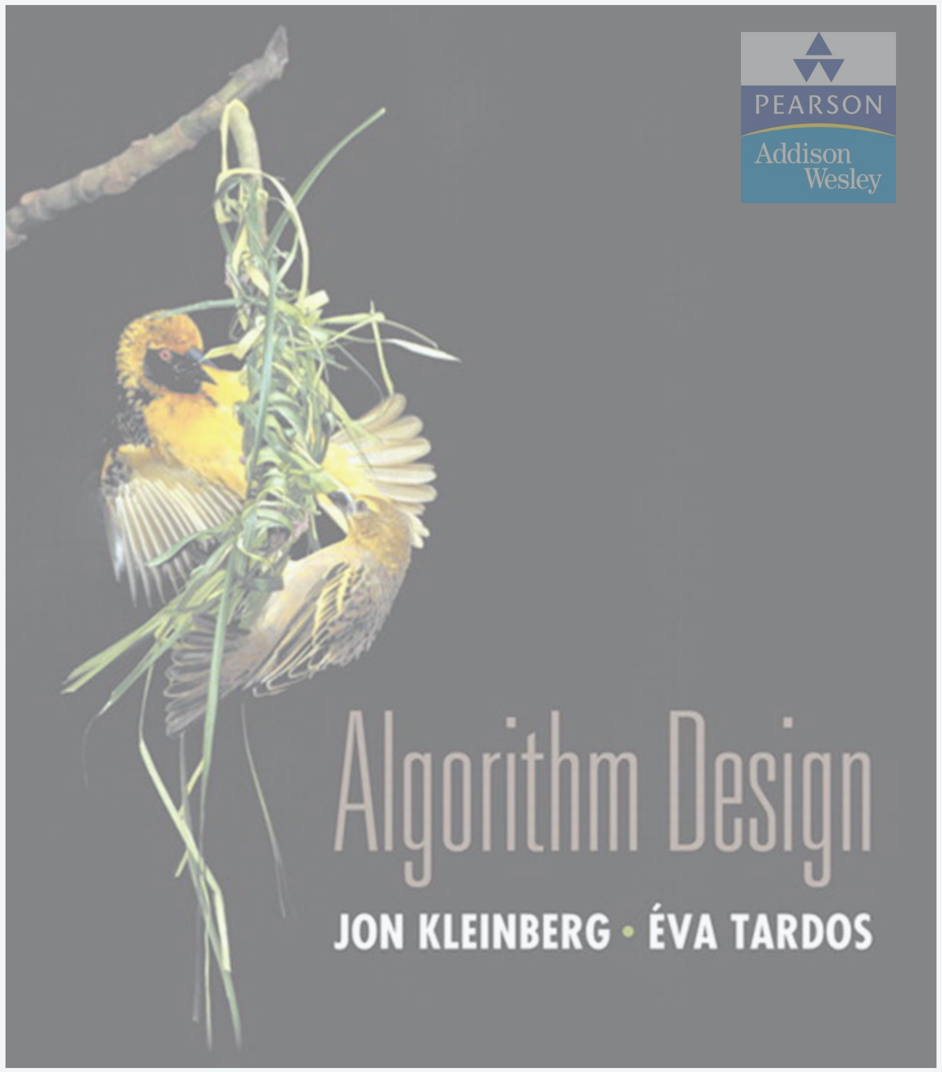


2. Algorithm Analysis

- *computational tractability*
- *asymptotic order of growth*
- *survey of common running times*

Lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos



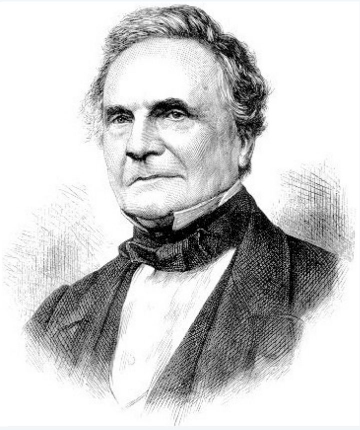
SECTION 2.1

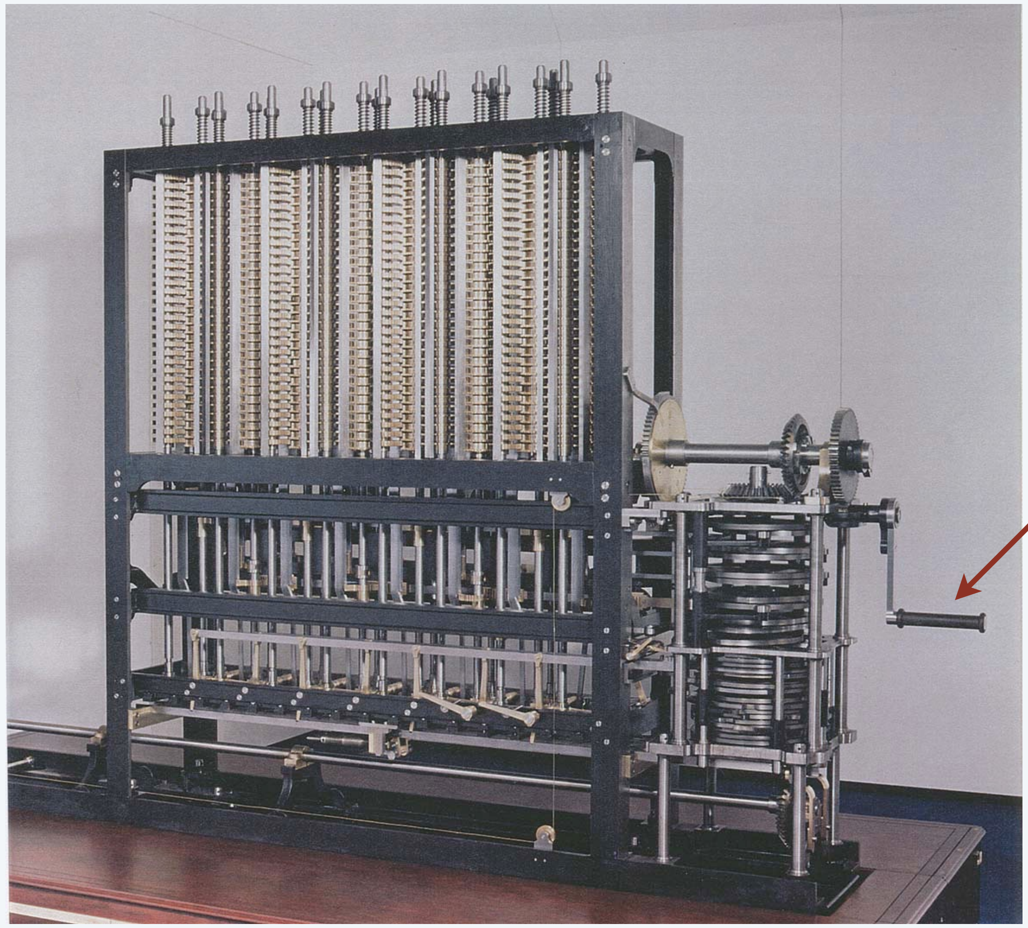
2. ALGORITHM ANALYSIS

- *computational tractability*
- *asymptotic order of growth*
- *survey of common running times*

A strikingly modern thought

" As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time? " — Charles Babbage (1864)





how many times do you have to turn the crank?

Analytic Engine

Models of computation: word RAM

Word RAM.

assume $w \geq \log_2 n$

- Each memory location and input/output cell stores a w-bit integer.
- Primitive operations: arithmetic/logic operations, read/write memory, array indexing, following a pointer, conditional branch, ...

constant-time C-style operations
($w = 64$)

Running time. Number of primitive operations.

Memory. Number of memory cells utilized.

Caveat. At times, need more refined model (e.g., multiplying n-bit integers).

Brute force

Brute force. For many nontrivial problems, there is a natural brute-force search algorithm that checks every possible solution.

- Typically takes 2^n steps (or worse) for inputs of size n.
- Unacceptable in practice.

Ex. Stable matching problem: test all $n!$ perfect matchings for stability.

Polynomial running time

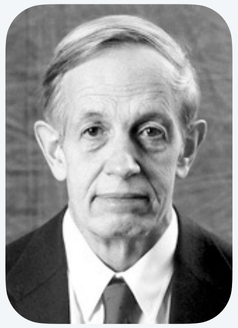
Desirable scaling property. When the input size doubles, the algorithm should slow down by at most some multiplicative constant factor C.

Def. An algorithm is poly-time if the above scaling property holds.

There exist constants a > 0 and b > 0 such that, for every input of size n, the algorithm performs $\leq a\, n^b$ primitive computational steps.

corresponds to $C = 2^b$

von Neumann (1953)

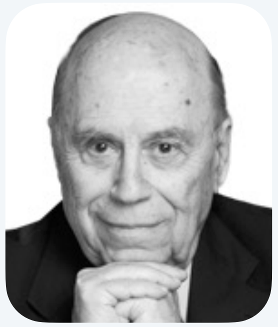


Nash (1955)

Gödel (1956)

Cobham (1964)

Edmonds (1965)



Rabin (1966)

Polynomial running time

We say that an algorithm is efficient if it has a polynomial running time.

Theory. Definition is (relatively) insensitive to model of computation.

Practice. It really works!

- The poly-time algorithms that people develop have both small constants and small exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

Exceptions. Some poly-time algorithms in the wild have galactic constants and/or huge exponents.

n^{120}

> **Map graphs in polynomial time**
>
> Mikkel Thorup*
> Department of Computer Science, University of Copenhagen
> Universitetsparken 1, DK-2100 Copenhagen East, Denmark
> mthorup@diku.dk
>
> **Abstract**
>
> *Chen, Grigni, and Papadimitriou (WADS'97 and STOC'98) have introduced a modified notion of planarity, where two faces are considered adjacent if they share at least one point. The corresponding abstract graphs are called* map graphs. *Chen et.al. raised the question of whether map graphs can be recognized in polynomial time. They showed that the decision problem is in NP and presented a polynomial time algorithm for the special case where we allow at most 4 faces to intersect in any point — if only 3 are allowed to intersect in a point, we get the usual planar graphs.*
>
> *Chen et.al. conjectured that map graphs can be recognized in polynomial time, and in this paper, their conjecture is settled affirmatively.*

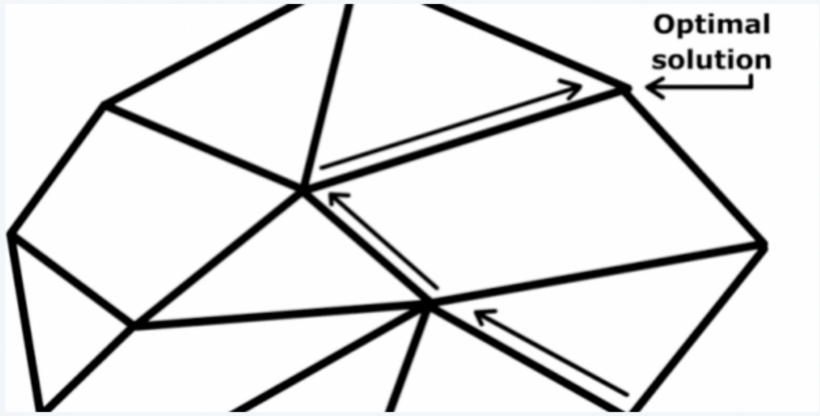
Q. Which would you prefer: $20\,n^{120}$ or $n^{1 + 0.02 \ln n}$?

Worst-case analysis

Worst case. Running time guarantee for any input of size n.

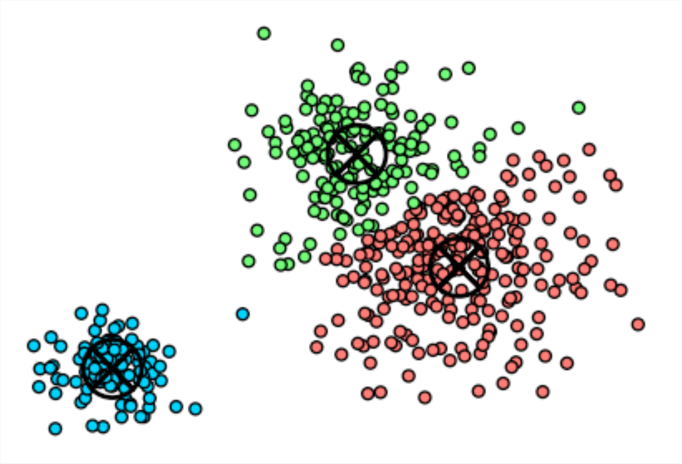
- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Exceptions. Some exponential-time algorithms are used widely in practice because the worst-case instances don't arise.



simplex algorithm

Linux grep



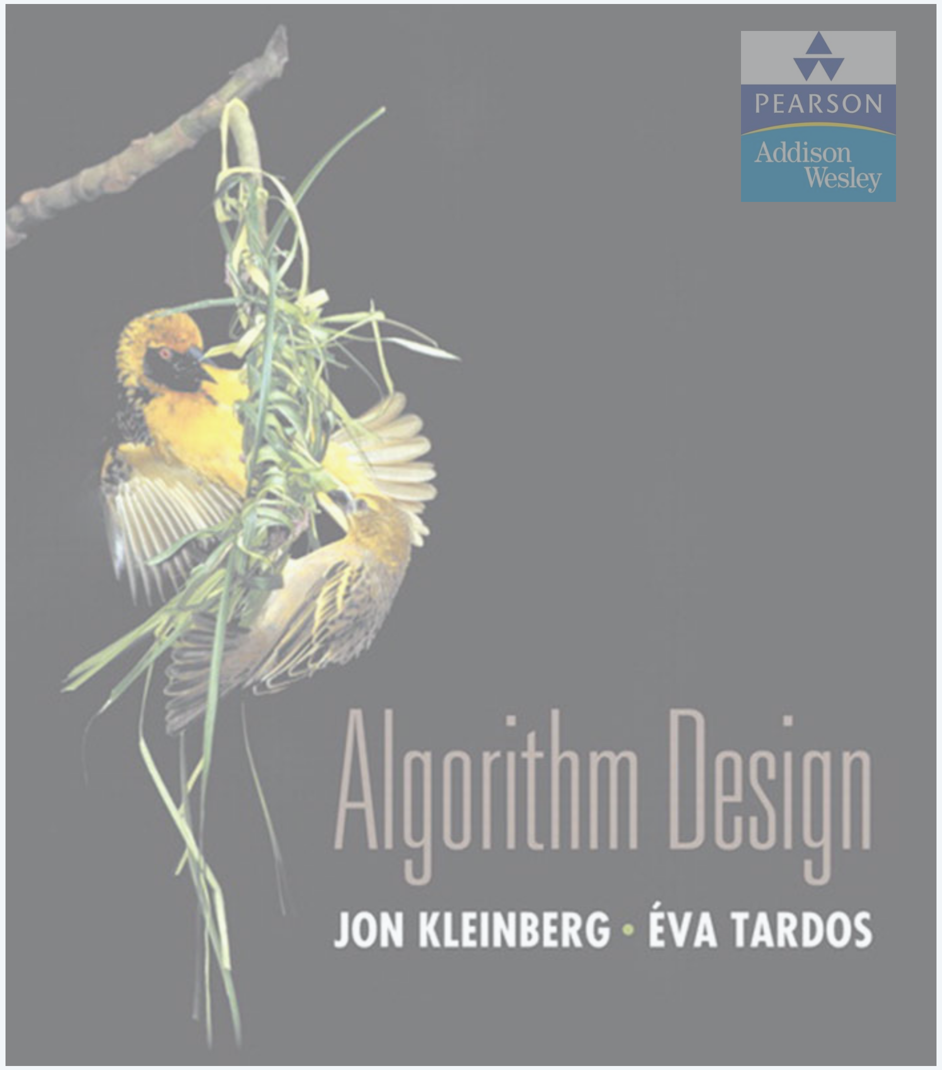
k-means algorithm

Other types of analyses

Probabilistic. Expected running time of a randomized algorithm.
Ex. The expected number of compares to quicksort n elements is $\sim 2n \ln n$.

Amortized. Worst-case running time for any sequence of n operations.
Ex. Starting from an empty stack, any sequence of n push and pop operations takes $O(n)$ primitive computational steps using a resizing array.

Also. Average-case analysis, smoothed analysis, competitive analysis, ...



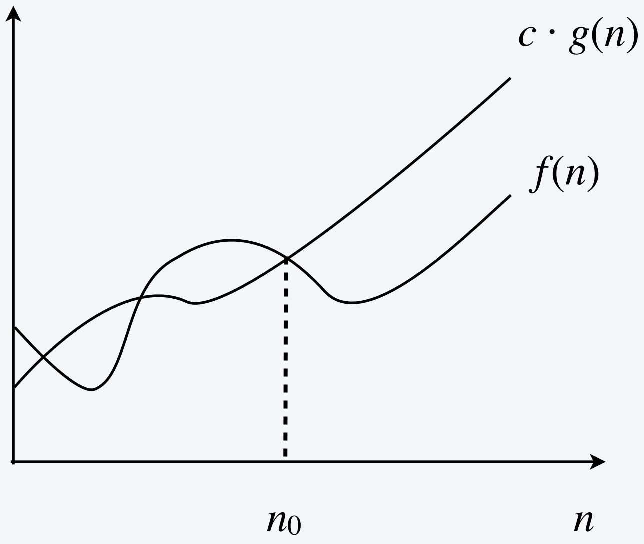
SECTION 2.2

2. ALGORITHM ANALYSIS

- computational tractability
- *asymptotic order of growth*
- survey of common running times

Big O notation

Upper bounds. $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.



Ex. $f(n) = 32n^2 + 17n + 1$.

- $f(n)$ is $O(n^2)$. ← choose $c = 50$, $n_0 = 1$
- $f(n)$ is neither $O(n)$ nor $O(n \log n)$.

Typical usage. Insertion sort makes $O(n^2)$ compares to sort n elements.

Analysis of algorithms: quiz 1

Let $f(n) = 3n^2 + 17\ n \log_2 n + 1000$. Which of the following are true?

A. $f(n)$ is $O(n^2)$.

B. $f(n)$ is $O(n^3)$.

C. Both A and B.

D. Neither A nor B.

Big O notational abuses

One-way "equality." $O(g(n))$ is a set of functions, but computer scientists often write $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$.

Ex. Consider $g_1(n) = 5n^3$ and $g_2(n) = 3n^2$.

- We have $g_1(n) = O(n^3)$ and $g_2(n) = O(n^3)$.
- But, do not conclude $g_1(n) = g_2(n)$.

Domain and codomain. f and g are real-valued functions.

- The domain is typically the natural numbers: $\mathbb{N} \to \mathbb{R}$.
- Sometimes we extend to the reals: $\mathbb{R}_{\geq 0} \to \mathbb{R}$.
- Or restrict to a subset.

input size, recurrence relations

plotting, limits, calculus

Bottom line. OK to abuse notation in this way; not OK to misuse it.

Big O notation: properties

Reflexivity. f is $O(f)$.

Constants. If f is $O(g)$ and $c > 0$, then $c\,f$ is $O(g)$.

Products. If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 f_2$ is $O(g_1 g_2)$.

Pf.

- $\exists\, c_1 > 0$ and $n_1 \geq 0$ such that $0 \leq f_1(n) \leq c_1 \cdot g_1(n)$ for all $n \geq n_1$.
- $\exists\, c_2 > 0$ and $n_2 \geq 0$ such that $0 \leq f_2(n) \leq c_2 \cdot g_2(n)$ for all $n \geq n_2$.
- Then, $0 \leq f_1(n) \cdot f_2(n) \leq \underbrace{c_1 \cdot c_2}_{c} \cdot g_1(n) \cdot g_2(n)$ for all $n \geq \underbrace{\max\{n_1, n_2\}}_{n_0}$. ▪

Sums. If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 + f_2$ is $O(\max\{g_1, g_2\})$.

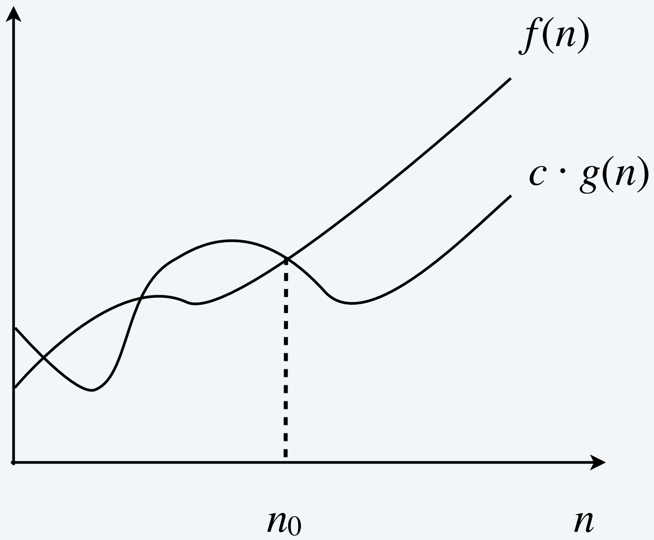
ignore lower-order terms

Transitivity. If f is $O(g)$ and g is $O(h)$, then f is $O(h)$.

Ex. $f(n) = 5n^3 + 3n^2 + n + 1234$ is $O(n^3)$.

Big Omega notation

Lower bounds. $f(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $f(n) \geq c \cdot g(n) \geq 0$ for all $n \geq n_0$.



Ex. $f(n) = 32n^2 + 17n + 1$.

- $f(n)$ is both $\Omega(n^2)$ and $\Omega(n)$. ⟵ choose $c = 32$, $n_0 = 1$
- $f(n)$ is not $\Omega(n^3)$.

Typical usage. Any compare-based sorting algorithm requires $\Omega(n \log n)$ compares in the worst case.

Vacuous statement. Any compare-based sorting algorithm requires at least $O(n \log n)$ compares in the worst case.

Which is an equivalent definition of big Omega notation?

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

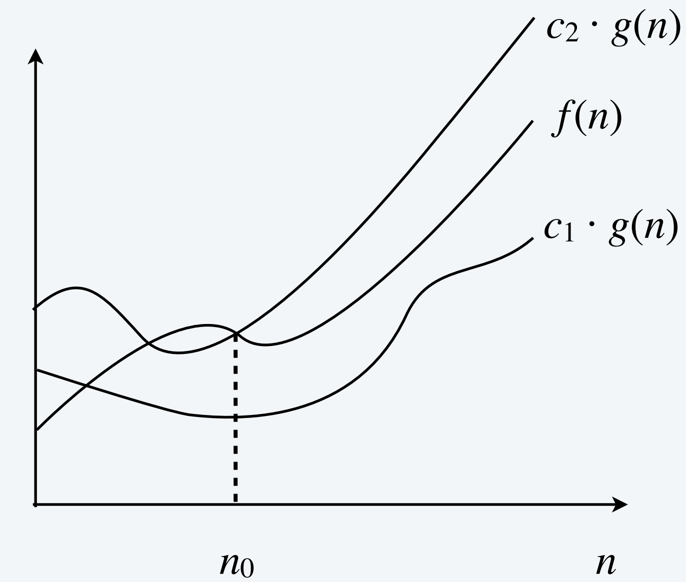
B. $f(n)$ is $\Omega(g(n))$ iff there exists a constant $c > 0$ such that $f(n) \geq c \cdot g(n) \geq 0$ for infinitely many n.

C. Both A and B.

D. Neither A nor B.

Big Theta notation

Tight bounds. $f(n)$ is $\Theta(g(n))$ if there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 \geq 0$ such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$.



Ex. $f(n) = 32n^2 + 17n + 1$.

- $f(n)$ is $\Theta(n^2)$. ← choose $c_1 = 32$, $c_2 = 50$, $n_0 = 1$
- $f(n)$ is neither $\Theta(n)$ nor $\Theta(n^3)$.

Typical usage. Mergesort makes $\Theta(n \log n)$ compares to sort n elements.

between $\frac{1}{2}\, n \log_2 n$ and $n \log_2 n$

Analysis of algorithms: quiz 3

Which is an equivalent definition of big Theta notation?

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

B. $f(n)$ is $\Theta(g(n))$ iff $\lim\limits_{n \to \infty} \dfrac{f(n)}{g(n)} = c$ for some constant $0 < c < \infty$.

C. Both A and B.

D. Neither A nor B.

Asymptotic bounds and limits

Proposition. If $\lim_{n \to \infty} \dfrac{f(n)}{g(n)} = c$ for some constant $0 < c < \infty$ then $f(n)$ is $\Theta(g(n))$.

Pf.

- By definition of the limit, for any $\varepsilon > 0$, there exists n_0 such that

$$c - \epsilon \leq \frac{f(n)}{g(n)} \leq c + \epsilon$$

  for all $n \geq n_0$.
- Choose $\varepsilon = ½\, c > 0$.
- Multiplying by $g(n)$ yields $1/2\, c \cdot g(n) \leq f(n) \leq 3/2\, c \cdot g(n)$ for all $n \geq n_0$.
- Thus, $f(n)$ is $\Theta(g(n))$ by definition, with $c_1 = 1/2\, c$ and $c_2 = 3/2\, c$. ▪

Proposition. If $\lim_{n \to \infty} \dfrac{f(n)}{g(n)} = 0$, then $f(n)$ is $O(g(n))$ but not $\Omega(g(n))$.

Proposition. If $\lim_{n \to \infty} \dfrac{f(n)}{g(n)} = \infty$, then $f(n)$ is $\Omega(g(n))$ but not $O(g(n))$.

Asymptotic bounds for some common functions

Polynomials. Let $f(n) = a_0 + a_1 n + \ldots + a_d n^d$ with $a_d > 0$. Then, $f(n)$ is $\Theta(n^d)$.

Pf.

$$\lim_{n \to \infty} \frac{a_0 + a_1 n + \ldots + a_d n^d}{n^d} = a_d > 0$$

Logarithms. $\log_a n$ is $\Theta(\log_b n)$ for every $a > 1$ and every $b > 1$.

Pf. $$\frac{\log_a n}{\log_b n} = \frac{1}{\log_b a}$$

no need to specify base
(assuming it is a constant)

Logarithms and polynomials. $\log_a n$ is $O(n^d)$ for every $a > 1$ and every $d > 0$.

Pf.

$$\lim_{n \to \infty} \frac{\log_a n}{n^d} = 0$$

Exponentials and polynomials. n^d is $O(r^n)$ for every $r > 1$ and every $d > 0$.

Pf.

$$\lim_{n \to \infty} \frac{n^d}{r^n} = 0$$

Factorials. $n!$ is $2^{\Theta(n \log n)}$.

Pf. Stirling's formula: $n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$

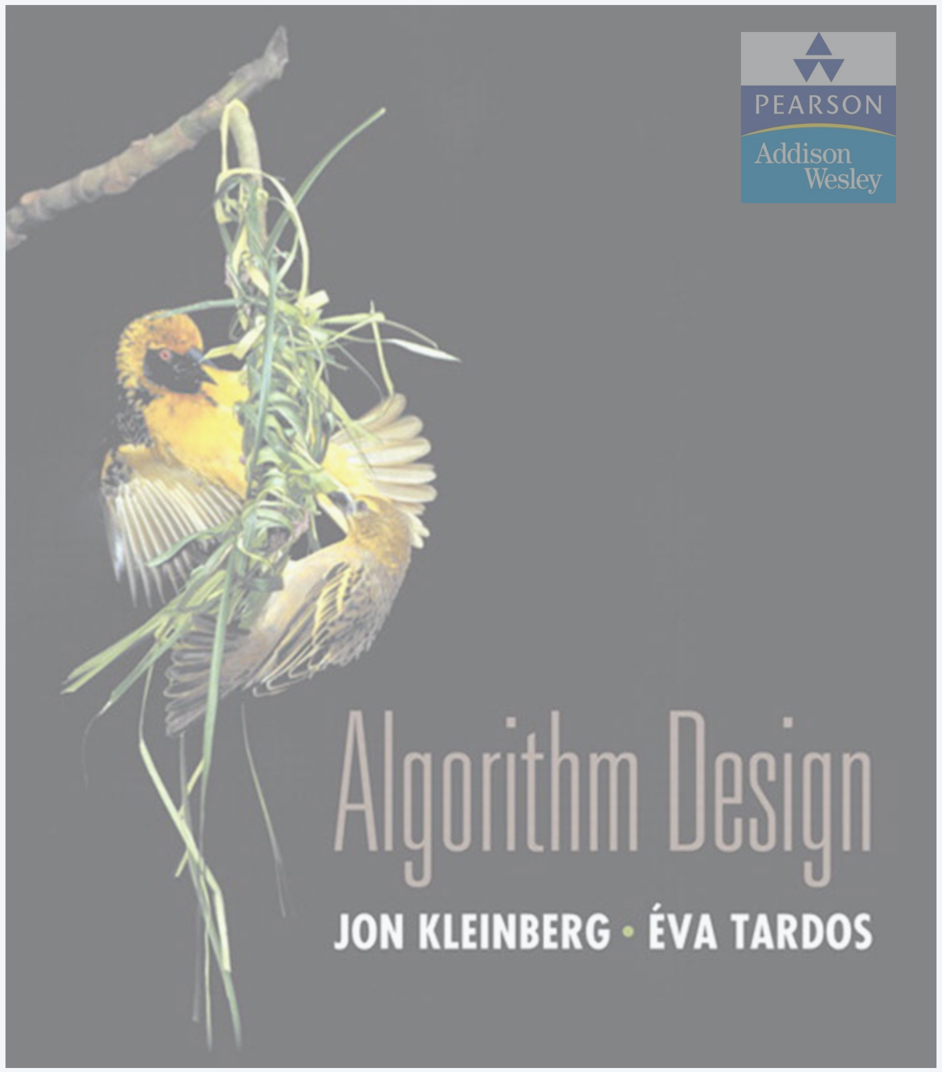
Big O notation with multiple variables

Upper bounds. $f(m, n)$ is $O(g(m, n))$ if there exist constants $c > 0$, $m_0 \geq 0$, and $n_0 \geq 0$ such that $0 \leq f(m, n) \leq c \cdot g(m, n)$ for all $n \geq n_0$ or $m \geq m_0$.

Ex. $f(m, n) = 32mn^2 + 17mn + 32n^3$.

- $f(m, n)$ is both $O(mn^2 + n^3)$ and $O(mn^3)$.
- $f(m, n)$ is $O(n^3)$ if a precondition to the problem implies $m \leq n$.
- $f(m, n)$ is neither $O(n^3)$ nor $O(mn^2)$.

Typical usage. In the worst case, breadth-first search takes $O(m + n)$ time to find a shortest path from s to t in a digraph with n nodes and m edges.



SECTION 2.4

2. ALGORITHM ANALYSIS

- *computational tractability*
- *asymptotic order of growth*
- *survey of common running times*

Constant time

Constant time. Running time is $O(1)$.

bounded by a constant, which does not depend on input size n

Examples.

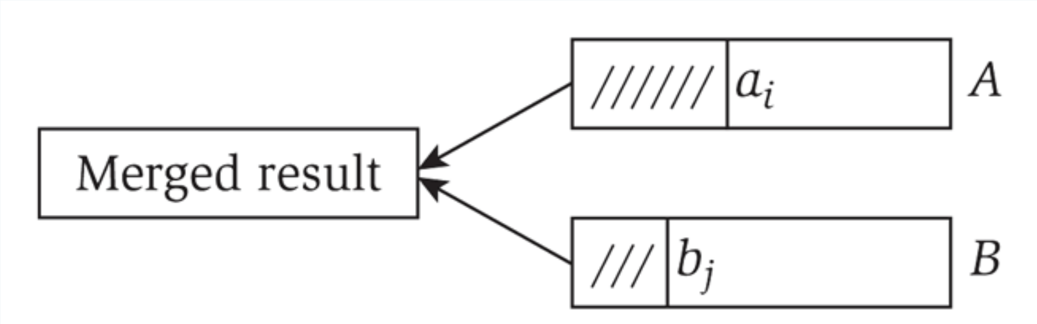
- Conditional branch.
- Arithmetic/logic operation.
- Declare/initialize a variable.
- Follow a link in a linked list.
- Access element i in an array.
- Compare/exchange two elements in an array.
- ...

Linear time

Linear time. Running time is $O(n)$.

Merge two sorted lists. Combine two sorted linked lists $A = a_1, a_2, \ldots, a_n$ and $B = b_1, b_2, \ldots, b_n$ into a sorted whole.

$O(n)$ algorithm. Merge in mergesort.



$i \leftarrow 1;\ j \leftarrow 1.$

WHILE (both lists are nonempty)

  IF ($a_i \le b_j$) append a_i to output list and increment i.

  ELSE append b_j to output list and increment j.

Append remaining elements from nonempty list to output list.

Target Sum

Target-Sum. Given a sorted array of n distinct integers and an integer T, find two that sum to exactly T?

input (sorted)	−20	10	20	30	35	40	60	70	$T = 60$
			↑ i			↑ j			

Target Sum

Target-Sum. Given a sorted array of n distinct integers and an integer T, find two that sum to exactly T?

$O(n^2)$ algorithm. Try all pairs.

$O(n)$ algorithm. Exploit sorted order.

input (sorted)	−20	10	20	30	35	40	60	70	$T = 60$
	↑ i							↑ j	

Invariant. No element to the left of i or right of j in pair that sums to T.

Logarithmic time

Logarithmic time. Running time is $O(\log n)$.

Search in a sorted array. Given a sorted array A of n distinct integers and an integer x, find index of x in array.

$O(\log n)$ algorithm. Binary search.

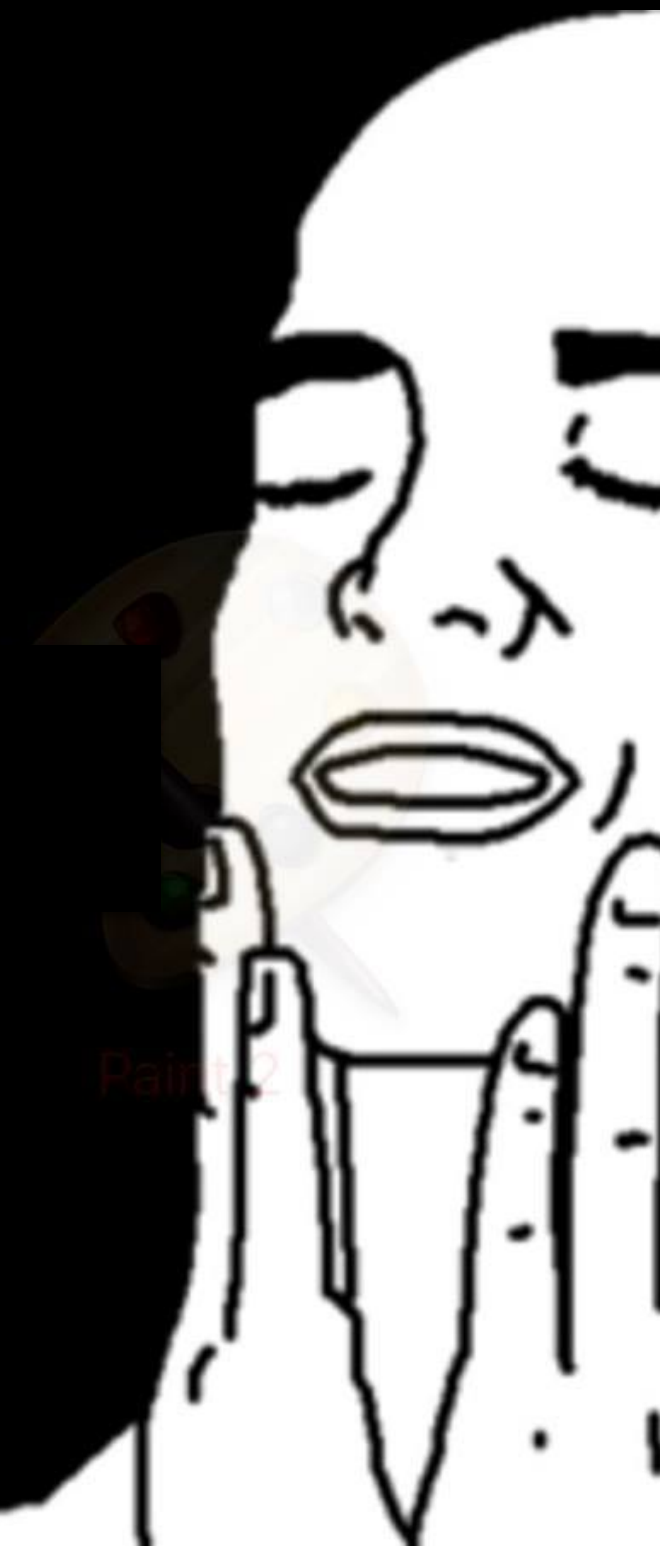
remaining elements

- Invariant: If x is in the array, then x is in $A[lo \mathinner{..} hi]$.
- After k iterations of WHILE loop, $(hi - lo + 1) \le n / 2^k \;\Rightarrow\; k \le 1 + \log_2 n$.

```
lo ← 1; hi ← n.
WHILE (lo ≤ hi)
    mid ← ⌊(lo + hi) / 2⌋.
    IF      (x < A[mid]) hi ← mid − 1.
    ELSE IF (x > A[mid]) lo ← mid + 1.
    ELSE RETURN mid.
RETURN −1.
```

Logarithmic time

$$O(\log n)$$

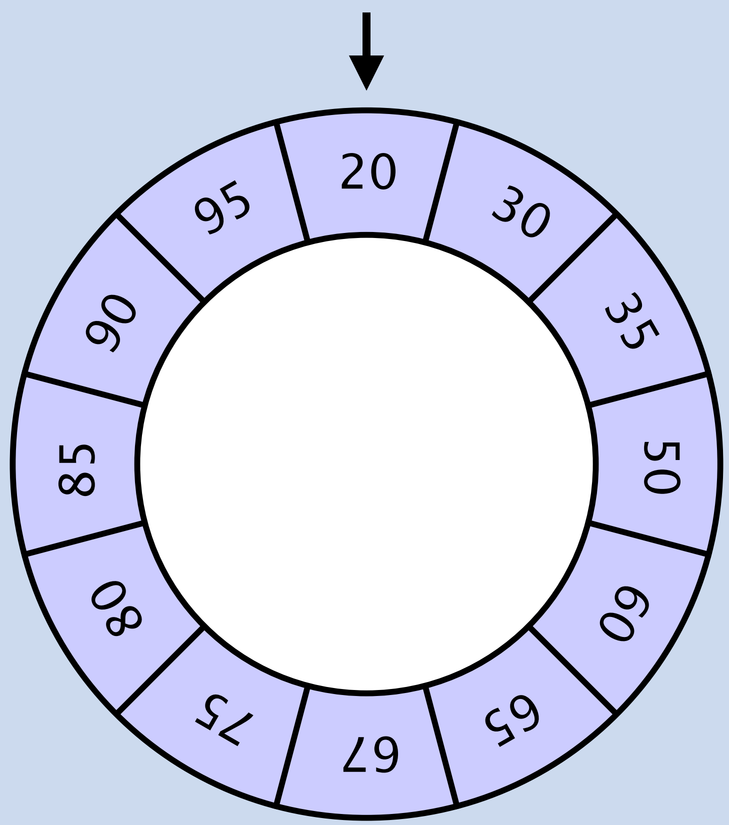


https://www.facebook.com/pg/npcompleteteens

Search in a sorted rotated array

Search-In-Sorted-Rotated-Array. Given a rotated sorted array of n distinct integers and an element x, determine if x is in the array.

sorted circular array



sorted rotated array

80	85	90	95	20	30	35	50	60	65	67	75
1	2	3	4	5	6	7	8	9	10	11	12

Search in a sorted rotated array

SEARCH-IN-SORTED-ROTATED-ARRAY. Given a rotated sorted array of n distinct integers and an element x, determine if x is in the array.

$O(\log n)$ algorithm.

- Find index k of smallest element.
- Binary search for x in either $A[1 \mathinner{..} k{-}1]$ or $A[k \mathinner{..} n]$.

find index of smallest element

$lo \leftarrow 1;\ hi \leftarrow n.$

IF $(A[lo] \le A[hi])$ RETURN 0 ⟵ sorted

WHILE $(lo + 2 \le hi)$ ⟵ at least 3 elements

  $mid \leftarrow \lfloor (lo + hi) / 2 \rfloor.$

  IF $(A[mid] < A[hi])$ $hi \leftarrow mid.$

  ELSE IF $(A[mid] > A[hi])$ $lo \leftarrow mid.$

RETURN hi

loop invariant
$A[lo] > A[hi]$

Linearithmic time

Linearithmic time. Running time is $O(n \log n)$.

Sorting. Given an array of n elements, rearrange them in ascending order.

$O(n \log n)$ algorithm. Mergesort.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Largest Empty interval

Largest-Empty-Interval. Given n timestamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval when no copies of file arrive?

Largest Empty interval

Largest-Empty-Interval. Given n timestamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval when no copies of file arrive?

$O(n \log n)$ algorithm.

- Sort the array a.
- Scan the sorted list in order, identifying the maximum gap between successive timestamps.

Quadratic time

Quadratic time. Running time is $O(n^2)$.

Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest to each other.

$O(n^2)$ algorithm. Enumerate all pairs of points (with $i < j$).

$min \leftarrow \infty$.

FOR $i = 1$ TO n

  FOR $j = i + 1$ TO n

    $d \leftarrow (x_i - x_j)^2 + (y_i - y_j)^2$.

    IF $(d < min)$

      $min \leftarrow d$.

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion. [see §5.4]

Cubic time

Cubic time. Running time is $O(n^3)$.

3-SUM. Given an array of n distinct integers, find three that sum to 0.

$O(n^3)$ algorithm. Enumerate all triples (with $i < j < k$).

```
FOR i = 1 TO n
  FOR j = i + 1 TO n
    FOR k = j + 1 TO n
    IF (a_i + a_j + a_k = 0)
      RETURN (a_i, a_j, a_k).
```

Remark. $\Omega(n^3)$ seems inevitable, but $O(n^2)$ is not hard. [see next slide]

3-Sum

3-Sum. Given an array of n distinct integers, find three that sum to 0.

$O(n^3)$ algorithm. Try all triples.

$O(n^2)$ algorithm.

3-SUM

3-SUM. Given an array of n distinct integers, find three that sum to 0.

$O(n^3)$ algorithm. Try all triples.

$O(n^2)$ algorithm.

- Sort the array a.
- For each integer a_i: solve TARGET-SUM on the array containing all elements except a_i with the target sum $T = -a_i$.

Best-known algorithm. $O(n^2 / (\log n / \log \log n))$.

Conjecture. No $O(n^{2-\varepsilon})$ algorithm for any $\varepsilon > 0$.

Polynomial time

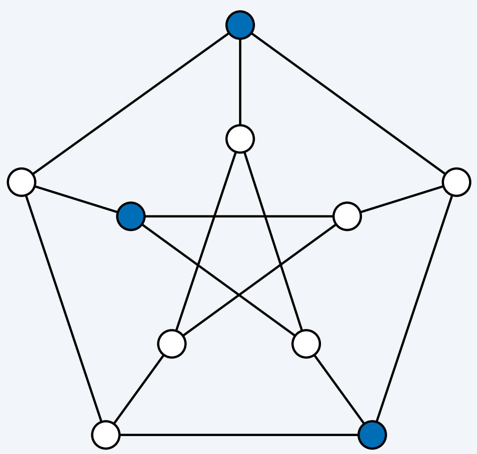
Polynomial time. Running time is $O(n^k)$ for some constant $k > 0$.

Independent set of size k. Given a graph, find k nodes such that no two are joined by an edge.

k is a constant

$O(n^k)$ algorithm. Enumerate all subsets of k nodes.

```
FOREACH subset S of k nodes:
    Check whether S is an independent set.
    IF (S is an independent set)
        RETURN S.
```



independent set of size 3

- Check whether S is an independent set of size k takes $O(k^2)$ time.
- Number of k-element subsets = $\binom{n}{k} = \frac{n(n-1)(n-2)\times\cdots\times(n-k+1)}{k(k-1)(k-2)\times\cdots\times 1} \le \frac{n^k}{k!}$
- $O(k^2 \, n^k / k!) = O(n^k)$.

poly-time for $k = 17$, but not practical

Exponential time

Exponential time. Running time is $O(2^{n^k})$ for some constant $k > 0$.

Independent set. Given a graph, find independent set of max size.

$O(n^2 2^n)$ algorithm. Enumerate all subsets of n elements.

$S^* \leftarrow \varnothing$.

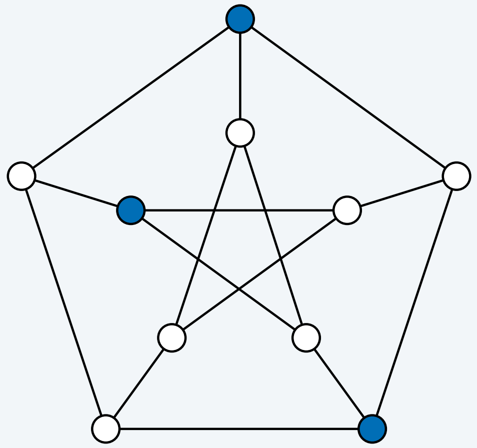
FOREACH subset S of n nodes:

    Check whether S is an independent set.

    IF (S is an independent set and $|S| > |S^*|$)

        $S^* \leftarrow S$.

RETURN S^*.



independent set of max size

Exponential time

Exponential time. Running time is $O(2^{n^k})$ for some constant $k > 0$.

Euclidean TSP. Given n points in the plane, find a tour of minimum length.

$O(n \times n!)$ algorithm. Enumerate all permutations of length n.

$\pi^* \leftarrow \varnothing$.

FOREACH permutation π of n points:

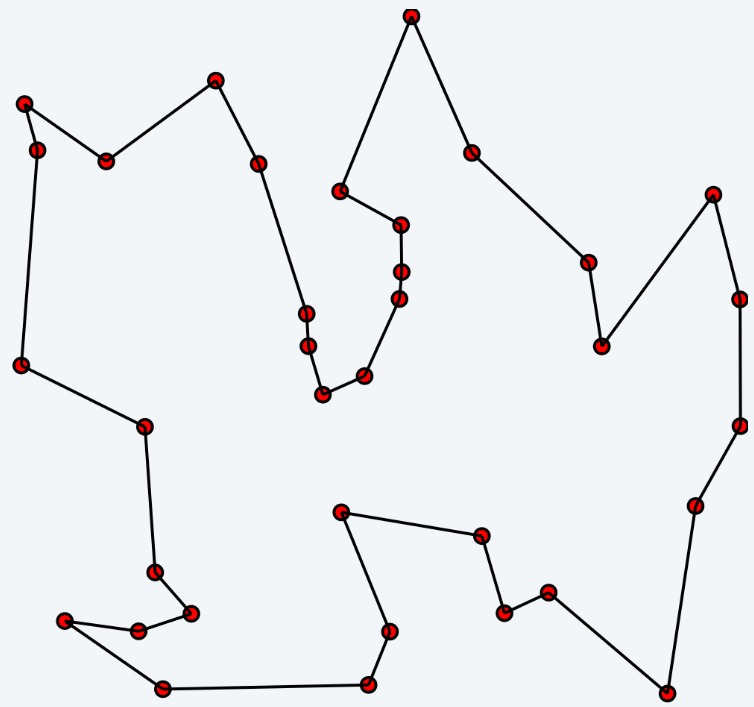
  Compute length of tour corresponding to π.

  IF (length(π) < length(π^*))

    $\pi^* \leftarrow \pi$.

RETURN π^*.

for simplicity, we'll assume Euclidean distances are rounded to nearest integer (to avoid issues with infinite precision)



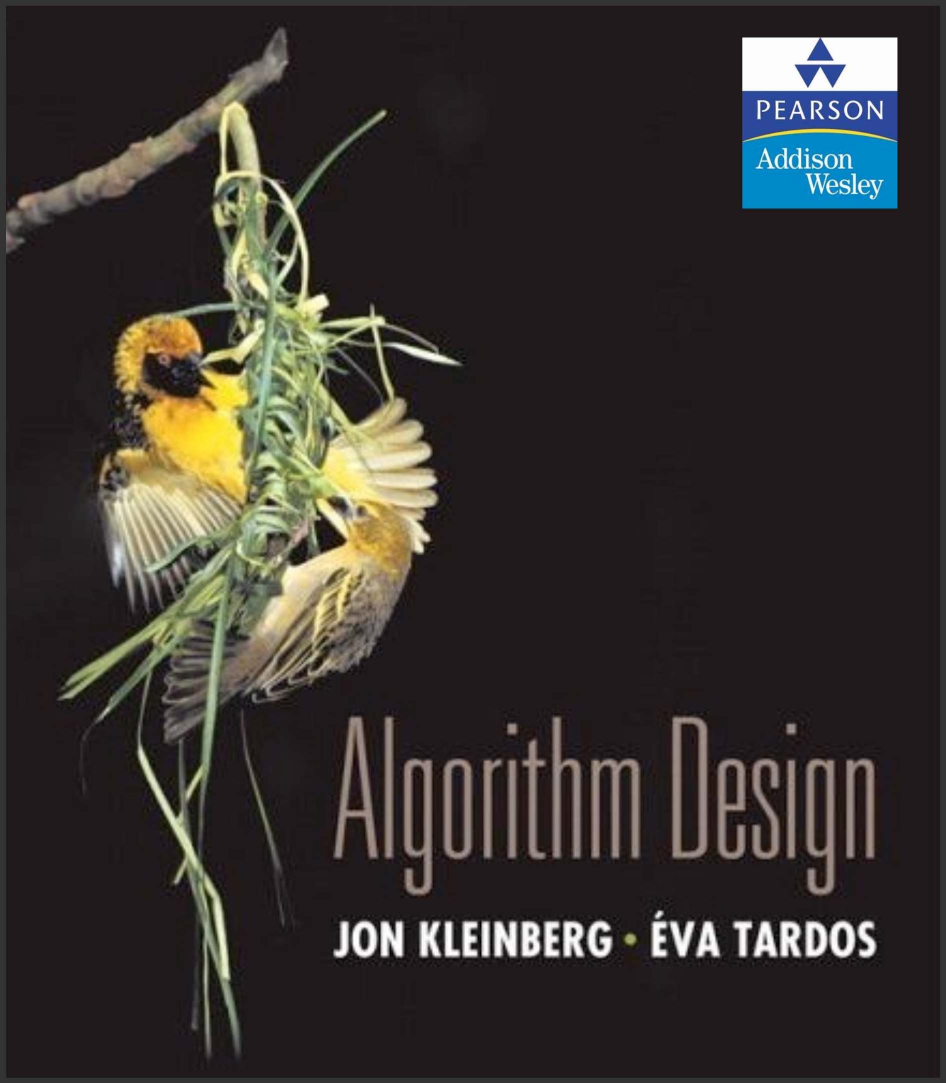
Which is an equivalent definition of exponential time?

A. $O(2^n)$

B. $O(2^{cn})$ for some constant $c > 0$.

C. Both A and B.

D. Neither A nor B.



2. Algorithm Analysis

- *binary search demo*

Lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
↑ *lo*							↑ *mid*							↑ *hi*

Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
↑ *lo*			↑ *mid*			↑ *hi*								

Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				lo	*mid*	*hi*								

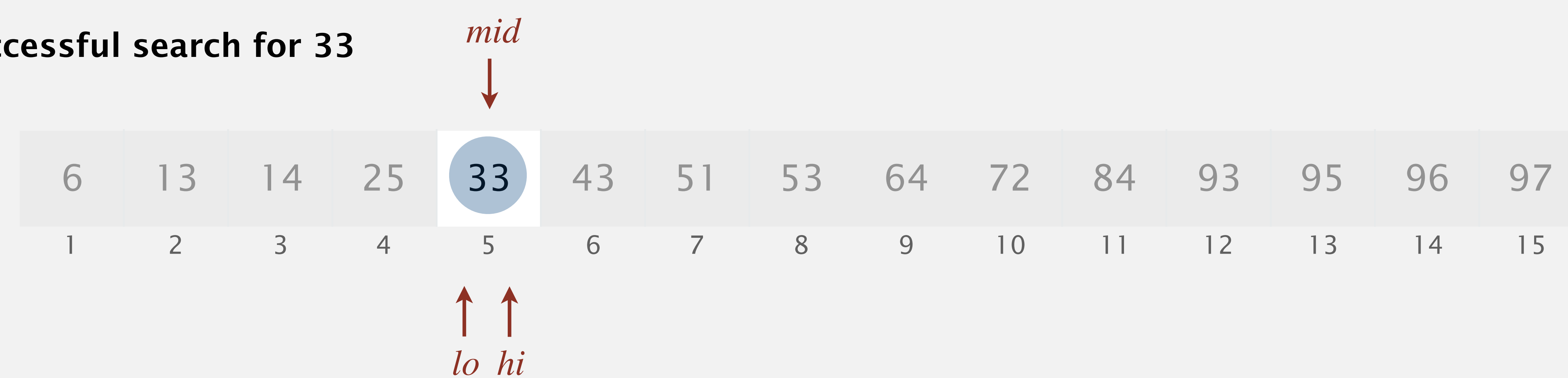
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33



return 5

Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
↑							↑							↑
lo							*mid*							*hi*

Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
lo			*mid*			*hi*								

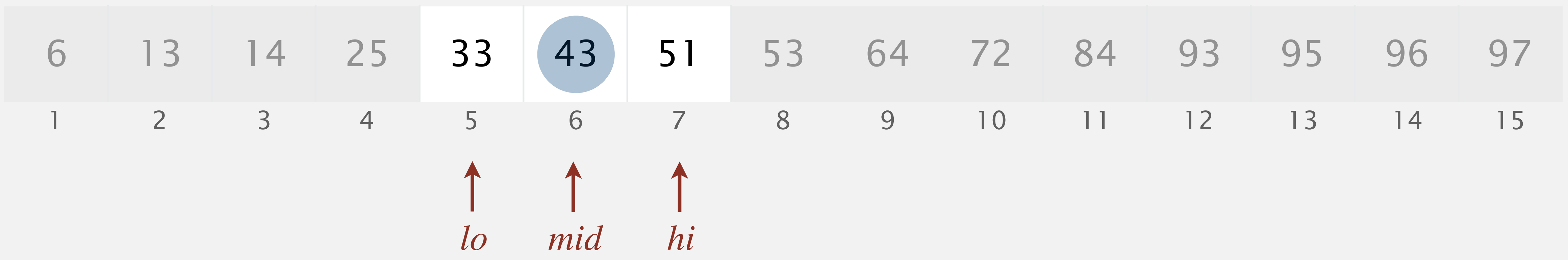
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34



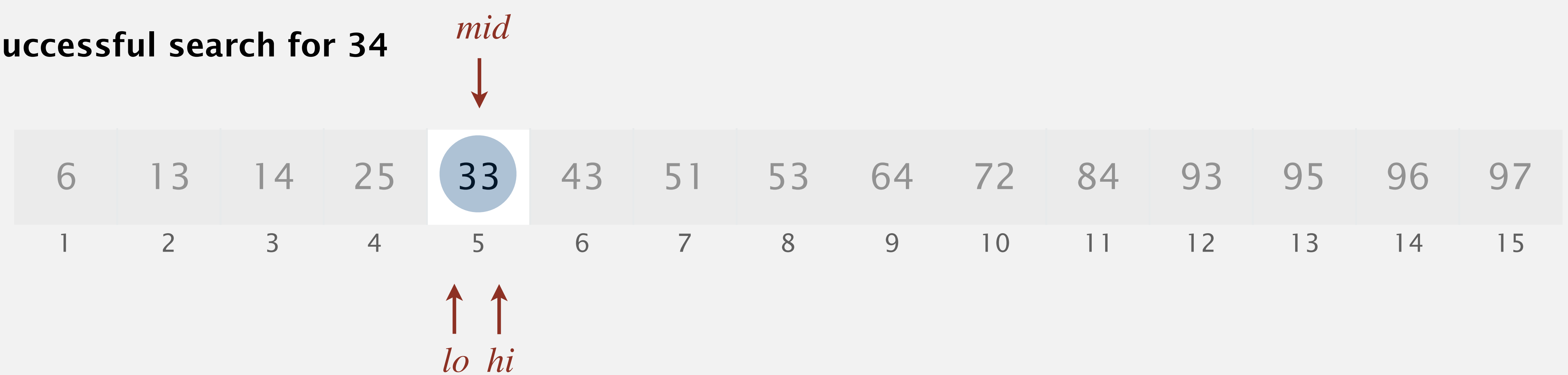
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34



return -1