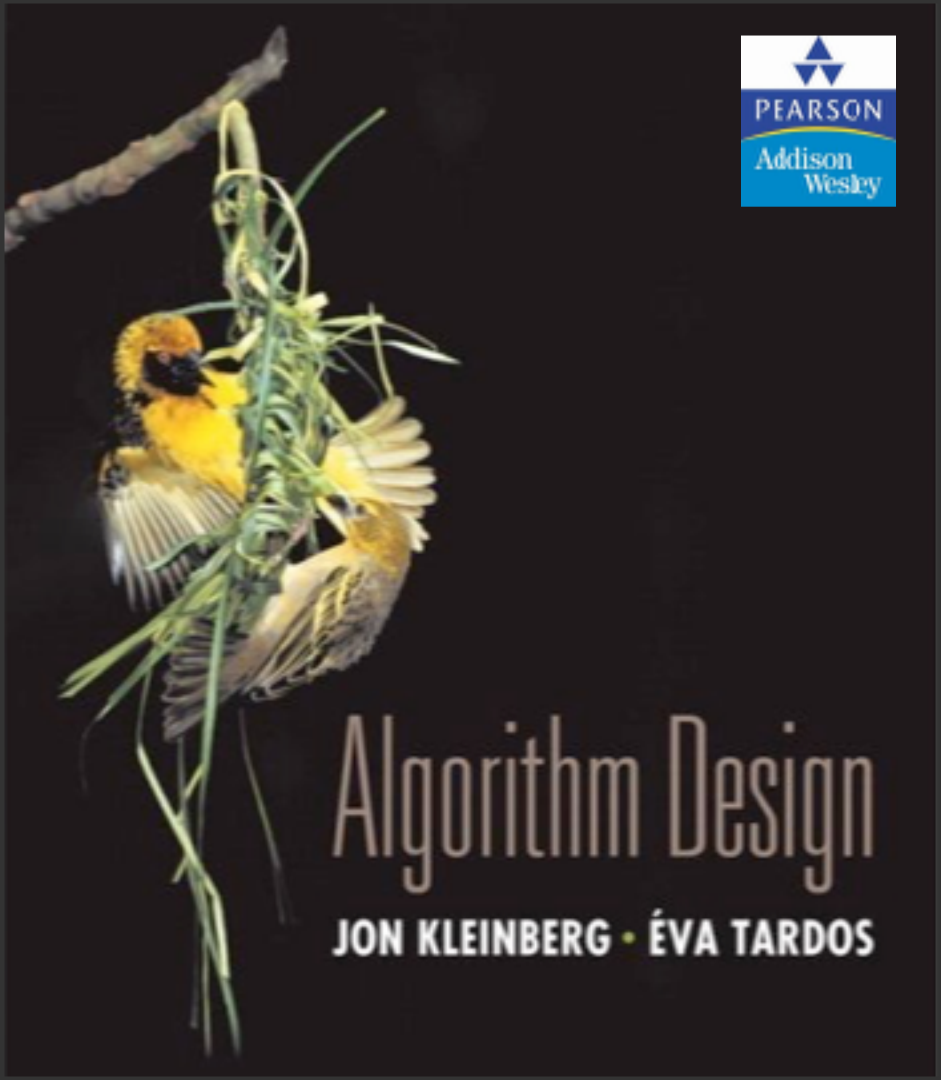
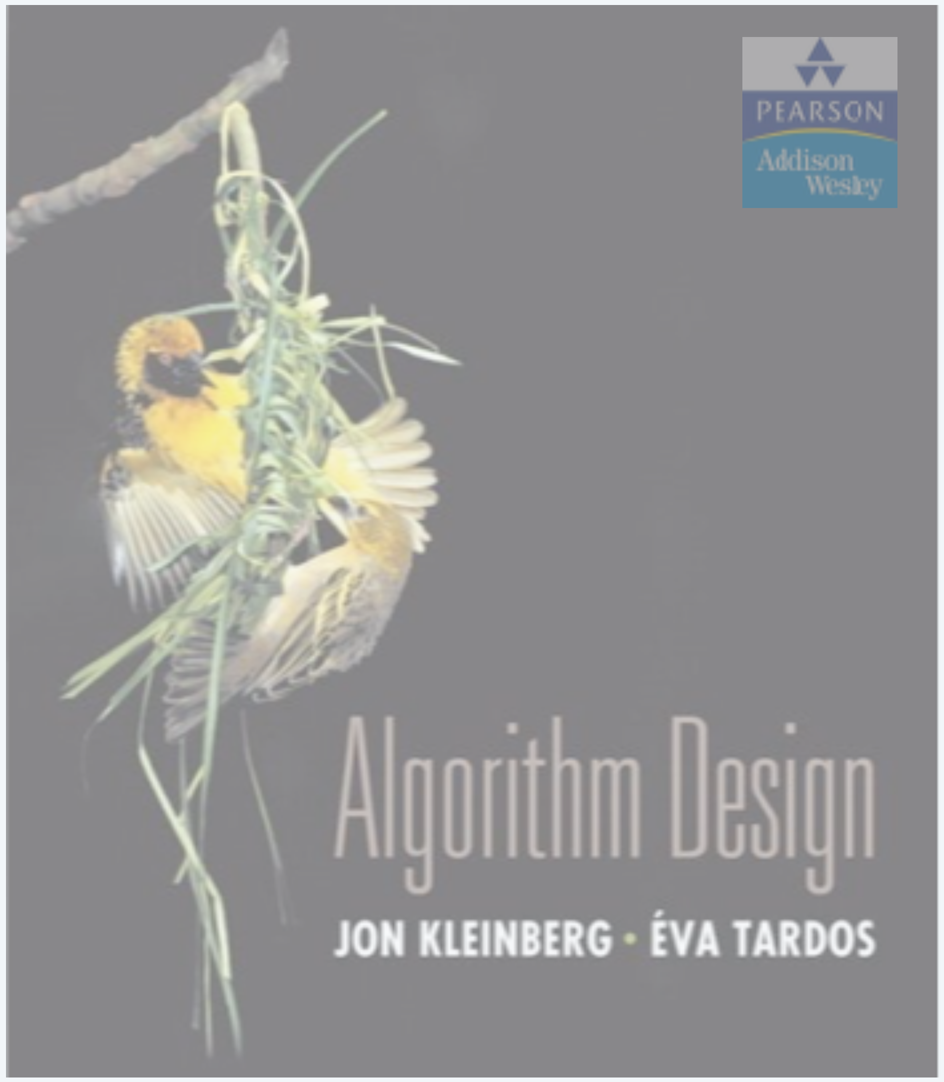


2. Algorithm Analysis

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

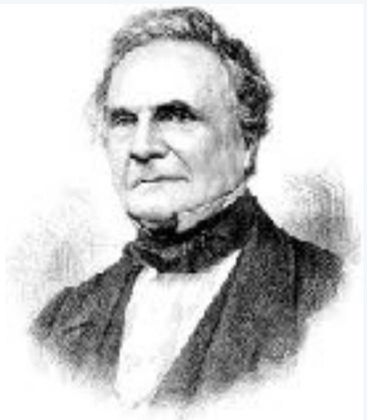


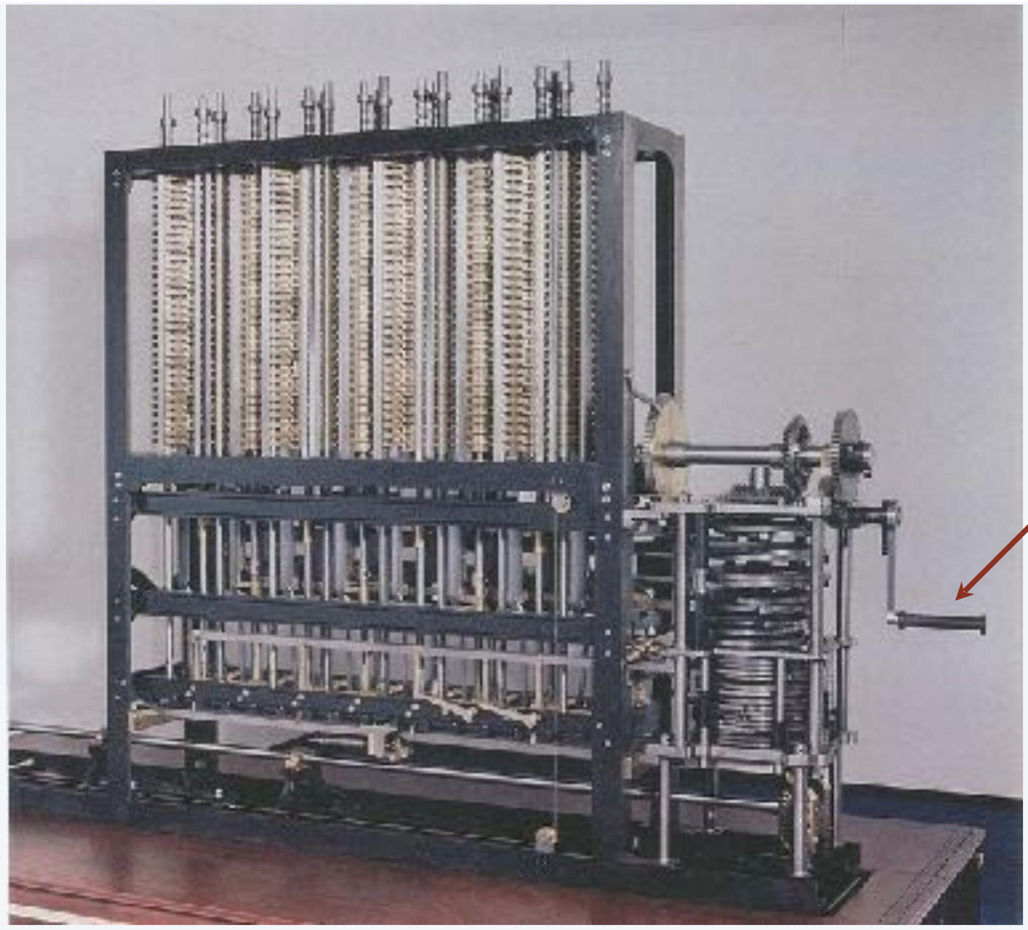
2. Algorithm Analysis

- *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

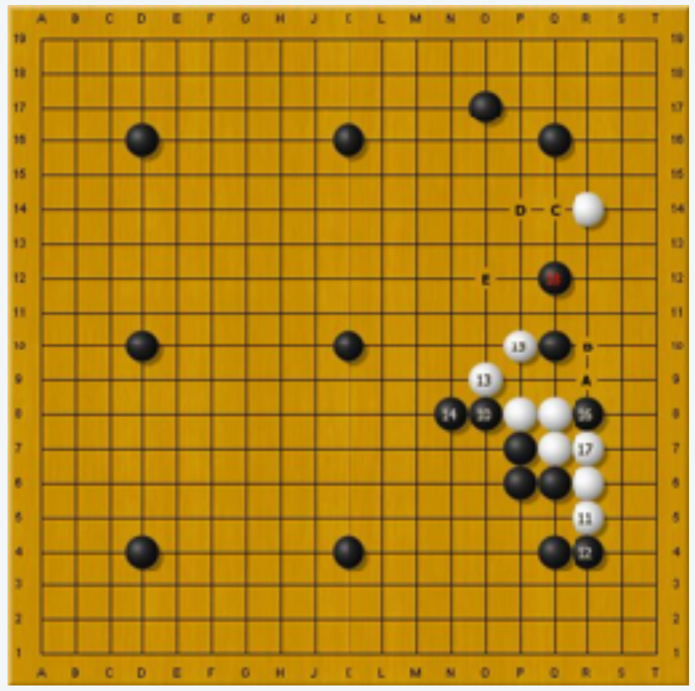
The faster your computing equipment, the more you stand to gain from efficient algorithms

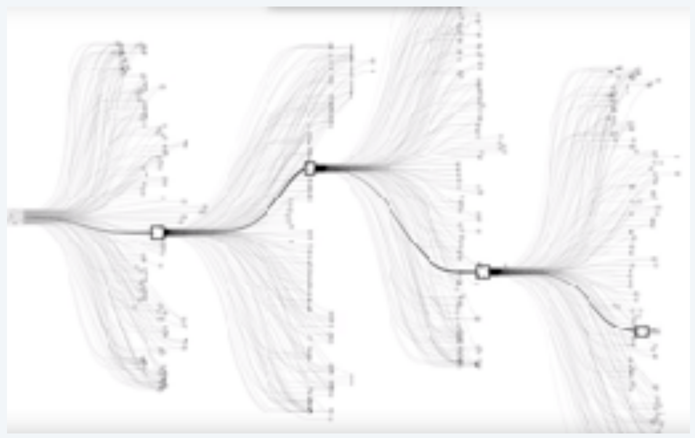
Brute force

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

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

$2^{19\text{x}19}$





Polynomial running time

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

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

There exists constants $c > 0$ and $d > 0$ such that on every input of size n, its running time is bounded by $c\,n^d$ primitive computational steps.

$\leftarrow$ choose $C = 2^d$

von Neumann (1953)

Nash (1955)

Gödel (1956)

Cobham (1964)

Edmonds (1965)

Rabin (1966)

Polynomial running time

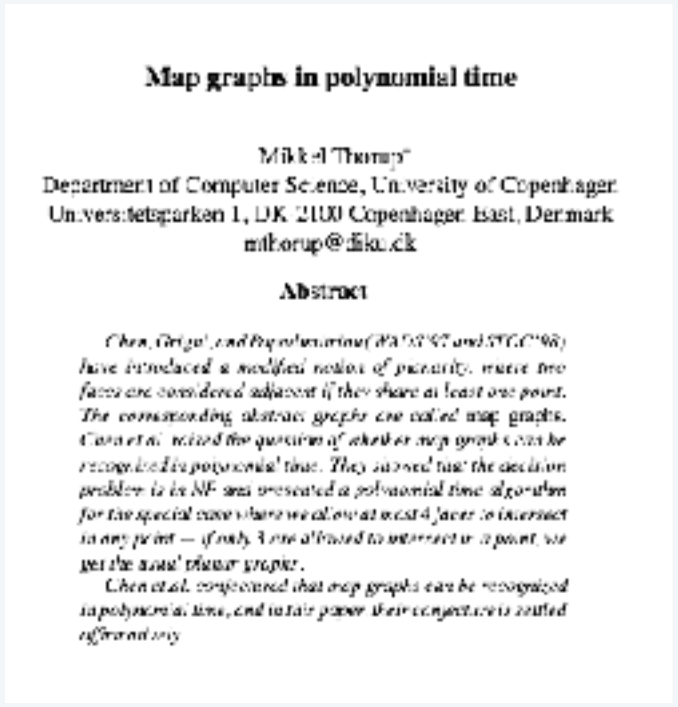
We say that an algorithm is efficient if has a polynomial running time.

Justification. It really works in practice!

- In practice, the poly-time algorithms that people develop have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

Exceptions. Some poly-time algorithms do have high constants and/or exponents, and/or are useless in practice.

Q. Which would you prefer $20\,n^{100}$ vs. $n^{1\,+\,0.02\,\ln n}$?



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.

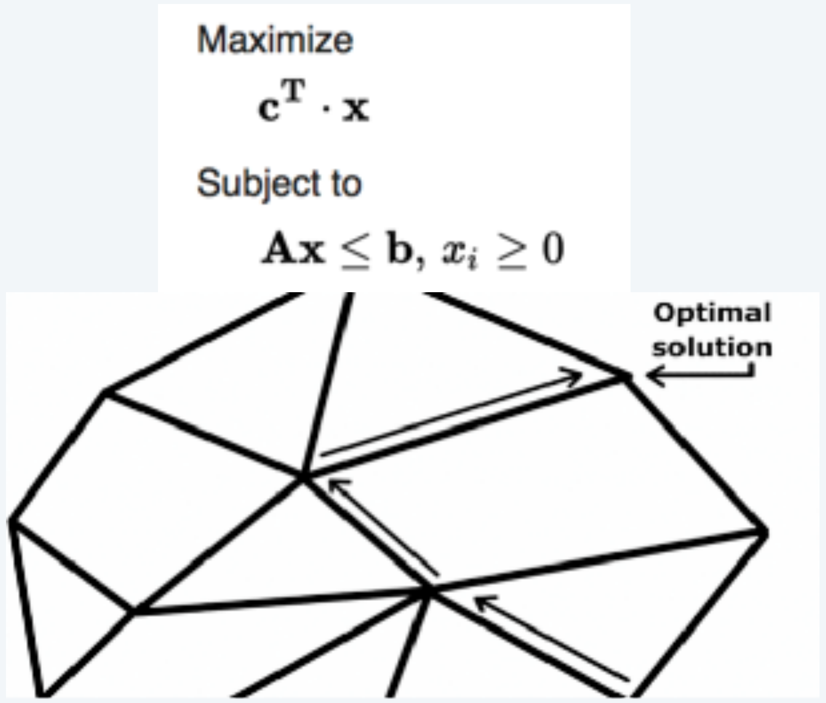
Chen et al. conjectured that map graphs can be recognized in polynomial time, and in this paper their conjecture is settled affirmatively.

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.

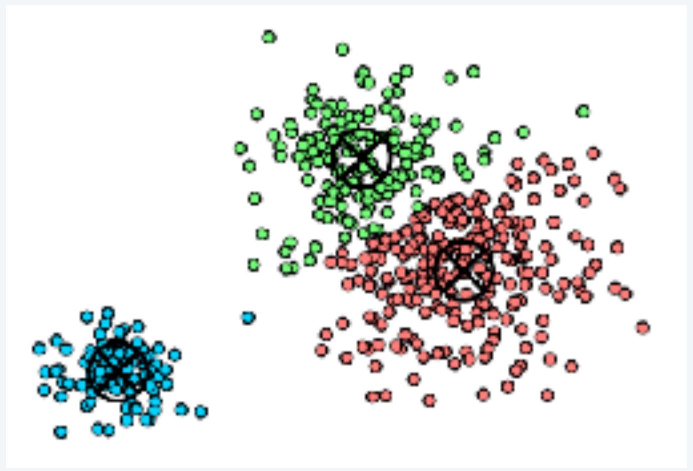
Exceptions. Some exponential-time algorithms are used widely in practice because the worst-case instances seem to be rare.


Maximize
$\mathbf{c}^{\mathbf{T}} \cdot \mathbf{x}$
Subject to
$\mathbf{A}\mathbf{x} \leq \mathbf{b},\ x_i \geq 0$
Optimal solution

simplex algorithm

Global search regular expression(**RE**) and **P**rint out the line

Linux grep



k-means algorithm

Types of analyses

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

Ex. Heapsort requires at most $2\,n \log_2 n$ compares to sort n elements.

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.

Types of analyses

Amortized. Worst-case running time for any sequence of n operations.
Motivation: given a sequence of operations, majority of them are cheap, but some rare might be expensive; thus a standard worst-case analysis might be overly pessimistic.
Basic idea: when expensive operations are particularly rare, their costs can be "spread out" (amortized) to all operations. If the artificial amortized costs are still cheap, we will have a tighter bound of the whole sequence of operations.

Ex: Starting from an empty stack, any sequence of n push and pop operations takes O(n) operations using a resizing array.

Analysis: an arbitrary number of pushes n to an array of size n, push operations take constant time except for the last one which takes O(n) time to perform the size doubling operation. Since there were n operations total we can take the average of this and find that for pushing elements onto the dynamic array takes

$$O(\tfrac{n}{n}) = O(1)$$

Types of analyses

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

Ex. Heapsort requires at most $2\, n \log_2 n$ compares to sort n elements.

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 $O(n)$ operations using a resizing array.

Average-case. Expected running time for a random input of size n.

Ex. The expected number of character compares performed by 3-way radix quicksort on n uniformly random strings is $\sim 2n \ln n$.

Also. Smoothed analysis, hybrid of worst-case and average-case analyses, by measuring the expected performance of algorithms under slight random perturbations of worst-case inputs, e.g., simplex algorithm

competitive analysis, … the performance of an online algorithm (unpredictable sequence of requests, completing each request without being able to see the future) is compared to the performance of an optimal offline algorithm that can view the sequence of requests in advance, e.g., catching, paging

Why it matters

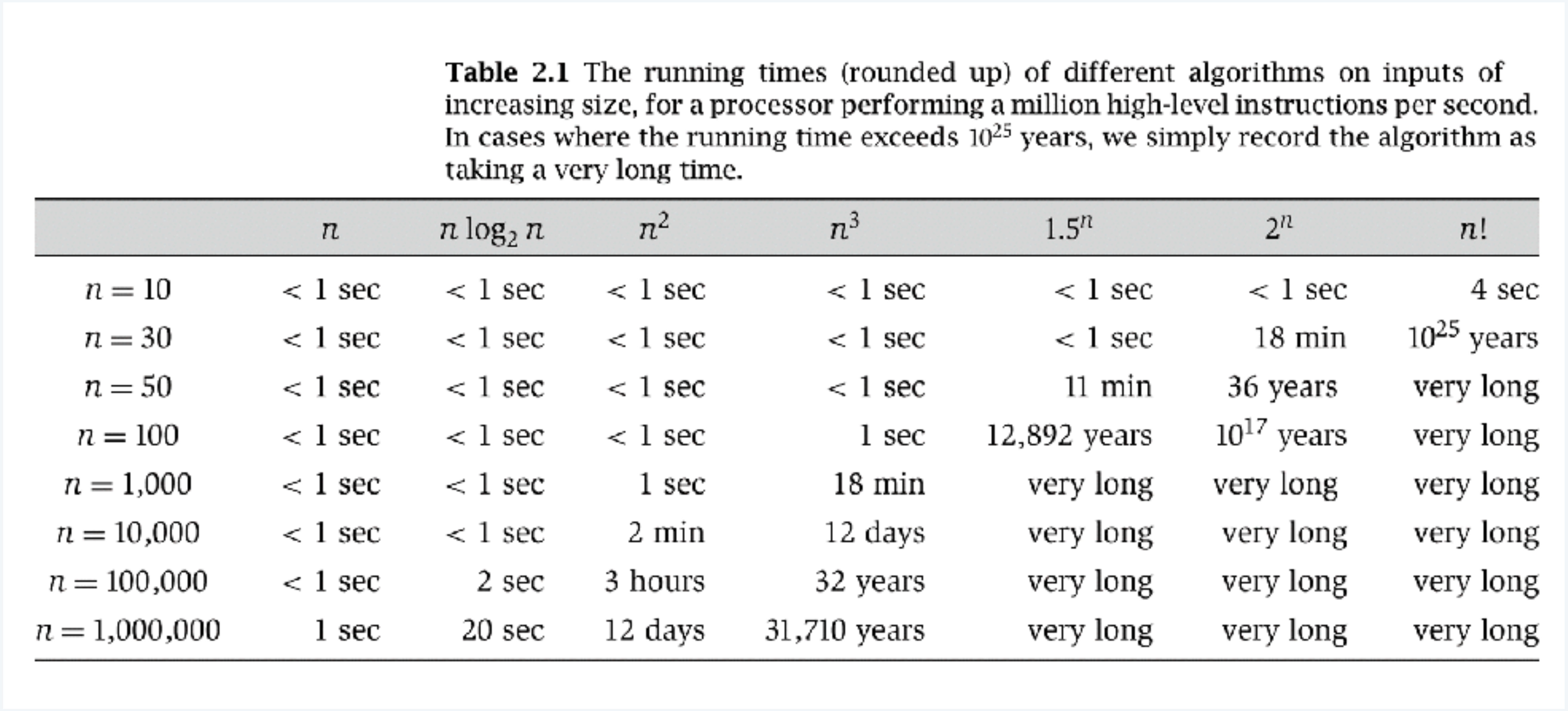
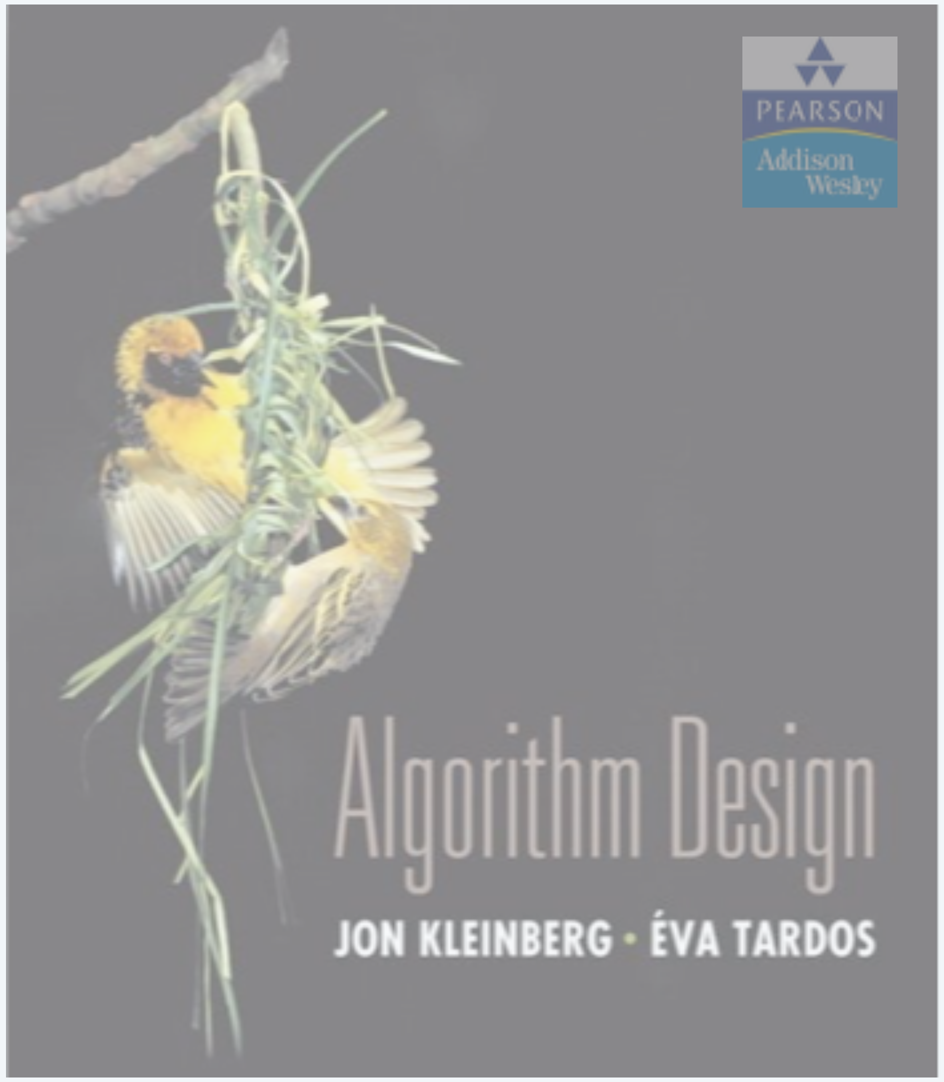


Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	4 sec
$n = 30$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	18 min	10^{25} years
$n = 50$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	11 min	36 years	very long
$n = 100$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	$<$ 1 sec	$<$ 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	$<$ 1 sec	$<$ 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	$<$ 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

The gulf between the growth rates of polynomial and exponential function is enormous.

Therefore, the mathematical formalism and the empirical evidence can line up well in the case of polynomial-time solvability

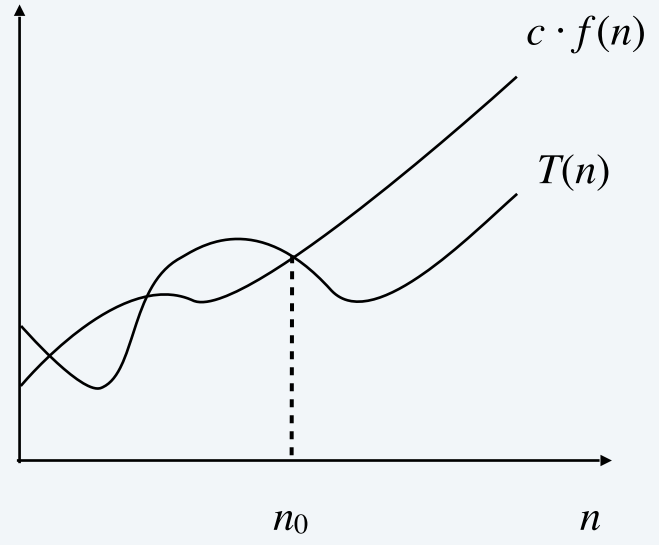


2. Algorithm Analysis

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

Big-Oh notation

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



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

- $T(n)$ is $O(n^2)$. $\longleftarrow$ choose c = 50, n_0 = 1 $\longleftarrow$ Proof
- $T(n)$ is also $O(n^3)$.
- $T(n)$ is neither $O(n)$ nor $O(n \log n)$.

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

Proof:

1+2+3+…+(n-1)= ½ n(n-1) < n^2

Alternate definition. $T(n)$ is $O(f(n))$ if $\limsup_{n \to \infty} \frac{T(n)}{f(n)} < \infty$.

Notational abuses

Equals sign. $O(f(n))$ is a set of functions, but computer scientists often write $T(n) = O(f(n))$ instead of $T(n) \in O(f(n))$.

Ex. Consider $f(n) = 5n^3$ and $g(n) = 3n^2$.

- We have $f(n) = O(n^3) = g(n)$.
- Thus, $f(n) = g(n)$.

Domain. The domain of $f(n)$ is typically the natural numbers $\{ 0, 1, 2, \ldots \}$.

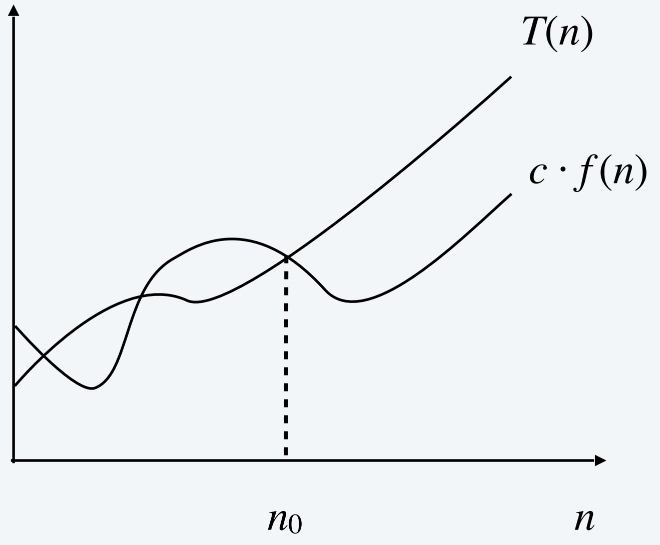
- Sometimes we restrict to a subset of the natural numbers.
  Other times we extend to the reals.

Nonnegative functions. When using big-Oh notation, we assume that the functions involved are (asymptotically) nonnegative.

Bottom line. OK to abuse notation; not OK to misuse it.

Big-Omega notation

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



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

- $T(n)$ is both $\Omega(n^2)$ and $\Omega(n)$. $\longleftarrow$ choose c = 32, n_0 = 1
- $T(n)$ is neither $\Omega(n^3)$ nor $\Omega(n^3 \log n)$.

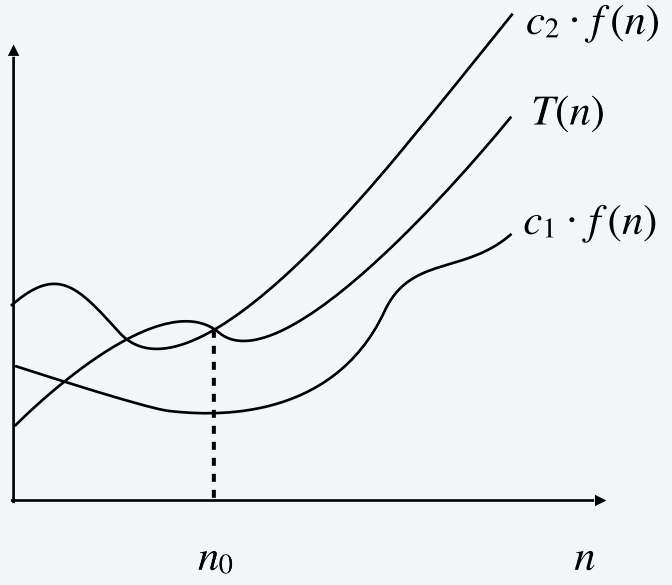
Typical usage. Any compare-based sorting algorithm requires $\Omega(n \log n)$ compares in the worst case.

Sorted order is on of n factorial permutations. If the algorithm always completes after at most f(n) steps, it cannot distinguish more than $2^{f(n)}$ cases because the keys are distinct and each comparison has only two possible outcomes. (Stirling's approximation)

$$2^{f(n)} \geq n! \rightarrow f(n) \geq \log_2(n!). \approx \Omega(n \log_2 n)$$

Big-Theta notation

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



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

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

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

Useful facts

Proposition. If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c > 0$,then $f(n)$ is $\Theta(g(n))$.

Pf. By definition of the limit, there exists n_0 such such that for all $n \geq n_0$

$$\frac{1}{2} c < \frac{f(n)}{g(n)} < 2c$$

- Thus, $f(n) \leq 2\, c\, g(n)$ for all $n \geq n_0$, which implies $f(n)$ is $O(g(n))$.
- Similarly, $f(n) \geq \frac{1}{2}\, c\, g(n)$ for all $n \geq n_0$, which implies $f(n)$ is $\Omega(g(n))$.

Proposition. If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$,then $f(n)$ is $O(g(n))$.

Asymptotic bounds for some common functions

Polynomials. Let $T(n) = a_0 + a_1 n + \ldots + a_d n^d$ with $a_d > 0$. Then, $T(n)$ 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. $\Theta(\log_a n)$ is $\Theta(\log_b n)$ for any constants $a, b > 0$. $\longleftarrow$ no need to specify base (assuming it is a constant)

$\log_a n = 1/\log_b a * \log_b n$

Logarithms and polynomials. For every $d > 0$, $\log n$ is $O(n^d)$.

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

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

For $r > s > 1$, $r^d = \Theta(s^d)$ ✖ $\longleftarrow$ Asymptotically speaking, exponential functions are all different.

Big-Oh notation with multiple variables

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

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

- $T(m, n)$ is both $O(mn^2 + n^3)$ and $O(mn^3)$.
- $T(m, n)$ is neither $O(n^3)$ nor $O(mn^2)$.

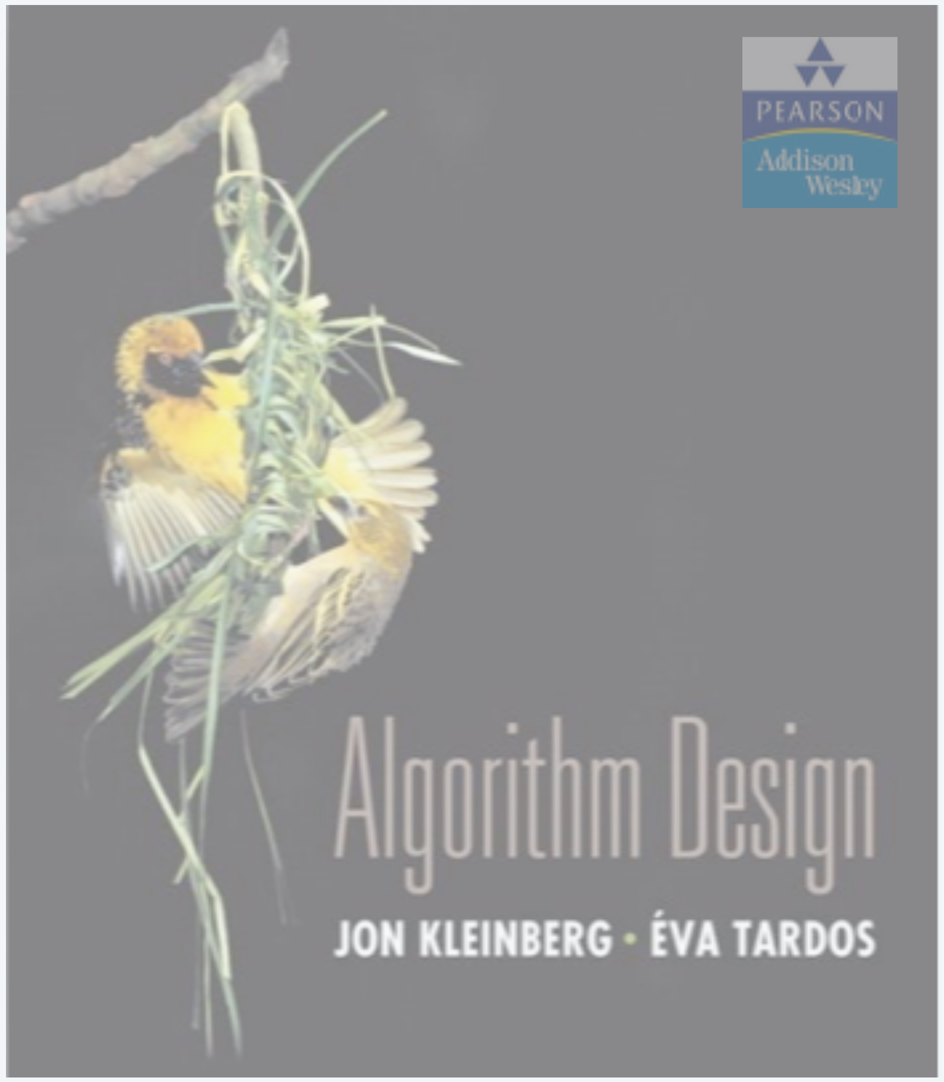
Typical usage. Breadth-first search takes $O(m + n)$ time to find the shortest path from s to t in a digraph.

Equivalent definitions

Limit superior.

- $f(n)$ is $O(g(n))$ iff $\displaystyle \limsup_{n \to \infty} \frac{f(n)}{g(n)} < \infty.$
- $f(n)$ is $\Omega(g(n))$ iff $\displaystyle \limsup_{n \to \infty} \frac{f(n)}{g(n)} > 0.$
- $f(n)$ is $\Theta(g(n))$ iff $\displaystyle 0 < \limsup_{n \to \infty} \frac{f(n)}{g(n)} < \infty.$

- O(*g*(*n*)): class of functions *f*(*n*) that grow no faster than *g*(*n*)
- Θ(*g*(*n*)): class of functions *f*(*n*) that grow at same rate as *g*(*n*)
- Ω(*g*(*n*)): class of functions *f*(*n*) that grow at least as fast as *g*(*n*)



2. Algorithm Analysis

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

Linear time: O(n)

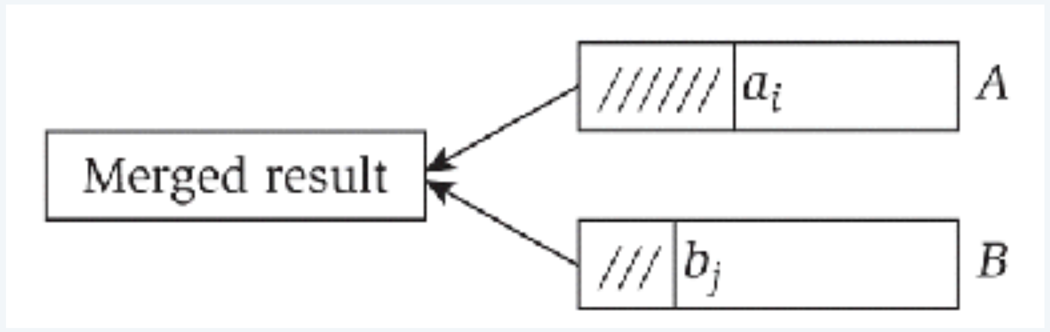
Linear time. Running time is proportional to input size.

Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

```
max ← $a_1$
for i = 2 to n {
   if ($a_i$ > max)
      max = $a_i$
}
```

Linear time: O(n)

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



```
i = 1, j = 1
while (both lists are nonempty) {
   if ($a_i$ < $b_j$) append $a_i$ to output list and increment i
   else           append $b_j$ to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size n takes $O(n)$ time.

Pf. After each compare, the length of output list increases by 1.

Linearithmic time: O(n log n)

O(n log n) time. Arises in divide-and-conquer algorithms.

Sorting. Mergesort and heapsort are sorting algorithms that perform $O(n \log n)$ compares.

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

O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

Quadratic time: $O(n^2)$

Ex. Enumerate all pairs of elements.

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.

$O(n^2)$ solution. Try all pairs of points.

```
min = $(x_1 - x_2)^2 + (y_1 - y_2)^2$
for i = 1 to n {
   for j = i+1 to n {
      d = $(x_i - x_j)^2 + (y_i - y_j)^2$
      if (d < min)
         min = d
   }
}
```

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion. [see Chapter 5]

Cubic time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Given n sets $S_1, \ldots, S_n$ each of which is a subset of $1, 2, \ldots, n$, is there some pair of these which are disjoint?

$O(n^3)$ solution. For each pair of sets, determine if they are disjoint.

```
foreach set $S_i$ {
   foreach other set $S_j$ {
      foreach element p of $S_i$ {
         determine whether p also belongs to $S_j$
      }
      if (no element of $S_i$ belongs to $S_j$)
         report that $S_i$ and $S_j$ are disjoint
   }
}
```

Polynomial time: $O(n^k)$

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

k is a constant

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

```
foreach subset S of k nodes {
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

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

poly-time for k=17,
but not practical

Exponential time

Independent set. Given a graph, what is maximum cardinality of an independent set?

$O(n^2 2^n)$ solution. Enumerate all subsets.

```
S* = ∅
foreach subset S of nodes {
  check whether S in an independent set
  if (S is largest independent set seen so far)
    update S* ← S
  }
}
```

Sublinear time

Search in a sorted array. Given a sorted array A of n numbers, is a given number x in the array?

O(log n) solution. Binary search.

```
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 yes
}
return no
```

Excise

1. Arrange the following functions in ascending order of growth rate.

$F1(n) = 10^n$

$F2(n) = n^{1/3}$

$F3(n) = n^n$

$F4(n) = \log_2 n$

$F5(n) = 2^{\log_2 n}$

2. Assume you have functions f and g such that f(n) is O(g(n)). For each of the following statements, decide whether you think it is true or false and give a proof or counterexample.

(a) $\log_2 f(n)$ is $O(\log_2 g(n))$

(b) $2^{f(n)}$ is $O(2^{g(n)})$

(c) $f(n)^2$ is $O(g(n)^2)$