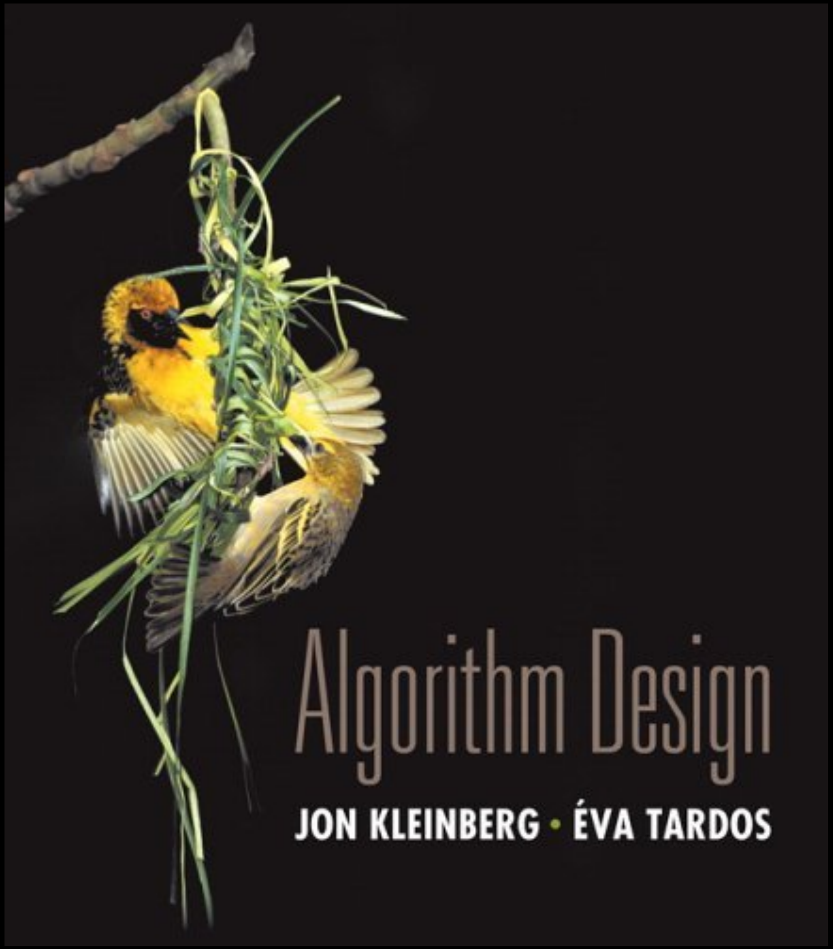


Chapter 2

Basics of
Algorithm Analysis

Algorithm Analysis

- Thinking about how the resource requirements of the algorithms will scale with increasing input size.

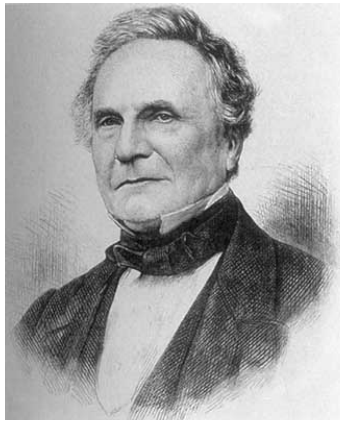
- Resources:
  - time
  - space

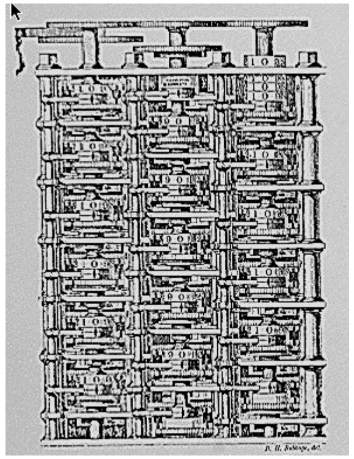
Computational efficiency. Efficiency in running time.

We want algorithms that run quickly!

Computational Tractability

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*


Charles Babbage (1864)


Analytic Engine (schematic)

Worst-Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size N.

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

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size N.

- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.

**1. Designating a law or code of extreme severity. 2. Harsh, rigorous.

Brute-Force Search

Brute force. For many non-trivial 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.

n ! for stable matching
with n men and n women

- Not only too slow to be useful, it is an intellectual cop-out.
- Provides us with absolutely no insight into the structure of the problem.

Proposed definition of efficiency. An algorithm is efficient if it achieves qualitatively better worst-case performance than brute-force search.

Polynomial-Time

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

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

A step. a single assembly-language instruction, one line of a programming language like C…

What happens if the input size increases from N to 2N?

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

Polynomial-Time

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

> There exists constants c > 0 and d > 0 such that on every input of size N, its running time is bounded by cN^d steps.

A step. a single assembly-language instruction, one line of a programming language like C…

What happens if the input size increases from N to 2N?
Answer: runningtime=$c(2N)^{d}=c2^dN^d=O(N^d)$ (because d is const)

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

Worst-Case Polynomial-Time

Def. An algorithm is efficient if its running time is polynomial.

Justification: It really works in practice!

- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always 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 are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

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

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

Asymptotic Order of Growth

- We try to express that an algorithm's worst case running time is at most proportional to some function f(n).
- Function f(n) becomes a bound on the running time of the algorithm.

- Pseudo-code style.
  - counting the number of pseudo-code steps.
  - step. Assigning a value to a variable, looking up an entry in an array, following a pointer, a basic arithmetic operation…

"On any input size n, the algorithm runs for at most $1.62n^2 + 3.5n + 8$ steps."

Do we need such precise bound?

We would like to classify running times at a coarser level of granularity.

- Similarities show up more clearly.

Asymptotic Order of Growth

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

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

Tight bounds. T(n) is $\Theta(f(n))$ if T(n) is both $O(f(n))$ and $\Omega(f(n))$.

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

- T(n) is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
- T(n) is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.

Asymptotically Tight Bounds

How to prove that f(n) is $\Theta(g(n))$?

Answer:

If the following limit exists and is equal to a constant c>0, then f(n) is $\Theta(g(n))$.

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}$$

Or

show that f(n) is both $O(g(n))$ and $\Omega(g(n))$

Notation

Slight abuse of notation. $T(n) = O(f(n))$.

- Asymmetric:
  - $f(n) = 5n^3$; $g(n) = 3n^2$
  - $f(n) = O(n^3) = g(n)$
  - but $f(n) \neq g(n)$.
- Better notation: $T(n) \in O(f(n))$.

Meaningless statement. Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons.

- Statement doesn't "type-check."
- Use Ω for lower bounds.

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 $f + g = \Theta(h)$.

Please see the proofs in the book pp. 39+.

Asymptotic Bounds for Some Common Functions

Polynomials. $a_0 + a_1 n + \ldots + a_d n^d$ 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. (even if d is not an integer.)
(Note that running time is also $\Theta(n^d)$. See book for the proof.)

Logarithms. $O(\log_a n) \underset{\uparrow}{=} O(\log_b n)$ for any constants $a, b > 0$.

can avoid specifying the base

Very slowly growing functions.

Logarithms. For every $x > 0$, $\log n \underset{\uparrow}{=} O(n^x)$.

log grows slower than every polynomial

Exponentials. For every $r > 1$ and every $d > 0$, $n^d \underset{\uparrow}{=} O(r^n)$.

every exponential grows faster than every polynomial

2.4 A Survey of Common Running Times

Linear Time: O(n)

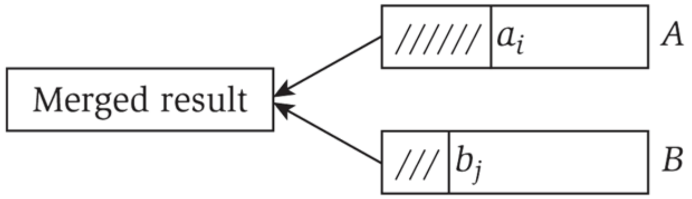
Linear time. Running time is at most a constant factor times the size of the input.

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 ($a_i \le b_j$) append $a_i$ to output list and increment i
   else (a   b ) 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 comparison, the length of output list increases by 1.

O(n log n) Time

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

$\nwarrow$ also referred to as linearithmic time

Sorting. Mergesort and heapsort are sorting algorithms that perform O(n log n) comparisons.

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 of time when no copies of the 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: $O(n^2)$

Quadratic time. 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
   }
}
```

← don't need to take square roots

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, …, n, is there some pair of these which are disjoint?

$O(n^3)$ solution. For each pairs 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: $O(n^k)$ Time

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.

```
for each 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 = $O(k^2)$.
- Number of k element subsets = $\binom{n}{k} = \frac{n\,(n-1)\,(n-2)\cdots(n-k+1)}{k\,(k-1)\,(k-2)\cdots(2)\,(1)} \le \frac{n^k}{k!}$
- $O(k^2\, n^k / k!) = O(n^k)$.

poly-time for k=17,
but not practical

Exponential Time

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

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

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

Total number of subsets: 2^n.

Sublinear Time

Binary search of a sorted list: $O(\log_2 n)$

Data Structure Used May Affect Complexity

Gale Shapley Algorithm (2.3) needs to maintain a dynamically changing set (list of free men).
Need fast ADD, DELETE, SELECT OPERATIONS
Use Priority Queue Data Structure.

Two implementations of priority queues:
1) Using Arrays and Lists: Slower: $O(n^2)$

2) Using Heap: Faster: $O(n\log n)$

Heap

A binary tree with n nodes and of height *h* is **almost complete** iff its nodes correspond to the nodes which are numbered 1 to *n* in the complete binary tree of height *h*.

A **heap** is an *almost complete binary tree* that satisfies the **heap property**:

max-heap: For every node *i* other than the root:

$$A[\text{Parent}(i)] \geq A[i]$$

min-heap: For every node i other than the root:

$$A[\text{Parent}(i)] \leq A[i]$$

Max-Heap

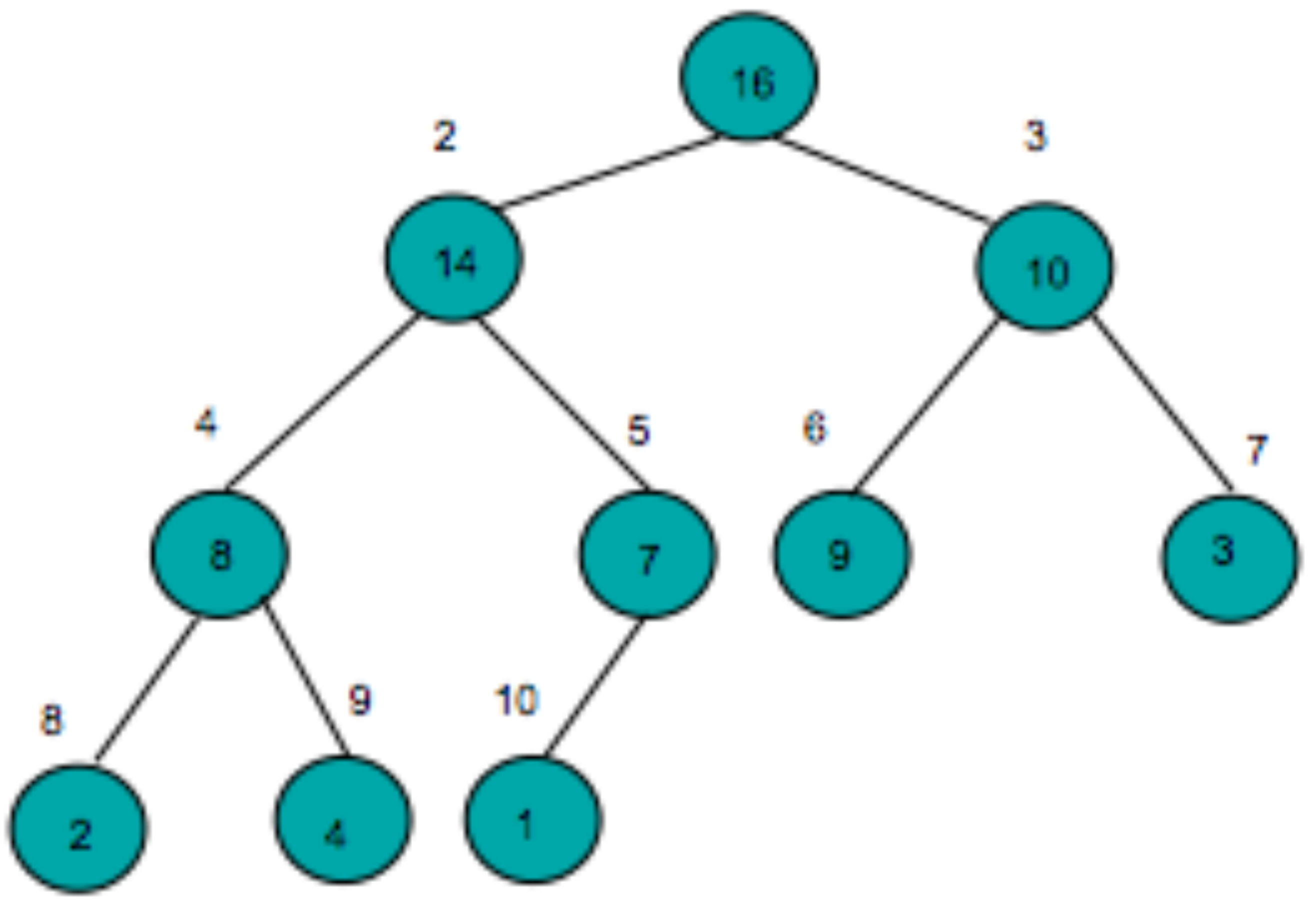
A **max-heap** is an *almost complete binary tree* that satisfies the **heap property**:

For every node i other than the root,

$$A[\text{PARENT}(i)] \geq A[i]$$

What does this mean?

- the value of a node is at most the value of its parent
- the largest element in the heap is stored in the root
- subtrees rooted at a node contain smaller values than the node itself



16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

Priority Queues (from ADS notes)

		Heaps			
Operation	Linked List	Binary	Binomial	Fibonacci *	Relaxed
make-heap	1	1	1	1	1
insert	1	log N	log N	1	1
find-min	N	1	log N	1	1
delete-min	N	log N	log N	log N	log N
union	1	N	log N	1	1
decrease-key	1	log N	log N	1	1
delete	N	log N	log N	log N	log N
is-empty	1	1	1	1	1

Dijkstra/Prim
1 make-heap
|V| insert
|V| delete-min
|E| decrease-key

$O(|V|^2)$

$O(|E| \log |V|)$

$O(|E| + |V| \log |V|)$

Propose-And-Reject Algorithm (Gale Shapley)

Propose-and-reject algorithm. [Gale-Shapley 1962] Intuitive method that guarantees to find a stable matching.

```
Initialize each person to be free.
while (some man is free and hasn't proposed to every woman) {
    Choose such a man m
    w = 1st woman on m's list to whom m has not yet proposed
    if (w is free)
        assign m and w to be engaged
    else if (w prefers m to her fiancé m')
        assign m and w to be engaged, and m' to be free
    else
        w rejects m
}
```

Implementing the Gale Shapley Algorithm

Using Priority Queues (Heap):

There is a need to keep a dynamically changing set (e.g. The set of free men)

We need to add, delete, select elements from the list fast.

Priority queue: Elements have a priority or key, each time you select, you select the item with the highest priority.

A set of elements S

Each element v in S has an associated key value key(v)

Add, delete, search operations in O(logn) time.

A sequence of O(n) priority queue ops can be used to sort n numbers.

An implementation for a priority queue: Heap

Heap order: key(w)<=key(v) where v at node i and w at i's parent

Heap operations:

StartHeap(N), Insert(H,v), FindMin(H), Delete(H,i), ExtractMin(H), ChangeKey(H,v,α)