ALGORITHM BASICS

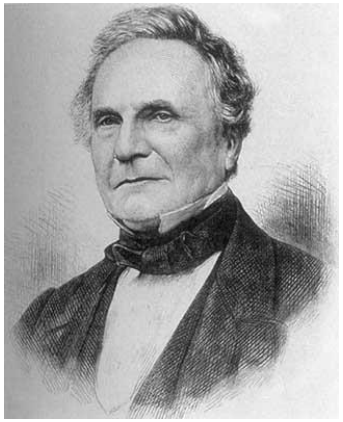
al-Khwārizmī (780-850)

- A Persian mathematician, astronomer and geographer;
- A scholar in the House of Wisdom in Baghdad.
- His book " Arithmetic" on the technique of performing arithmetic with Hindu-Arabic numerals was translated and introduced to the West in the twelfth century;
- Considered the founder of algebra, he presented the first systematic solution of linear and quadratic equations in his book "Algebra";
- The term "algorithm" was derived from the Latinized forms of al-Khwarizmi's name.
- Algorismi (Algoritmi) → Algorism (Algorithm)

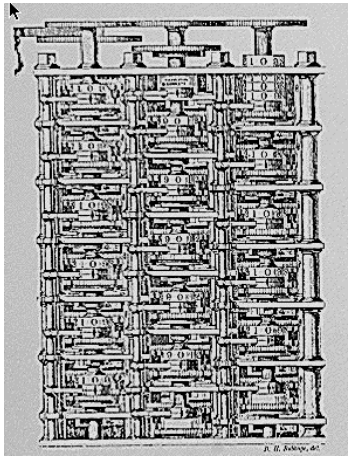
Charles Babbage

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 Babbage1791-1871



Charles Babbage (1864)

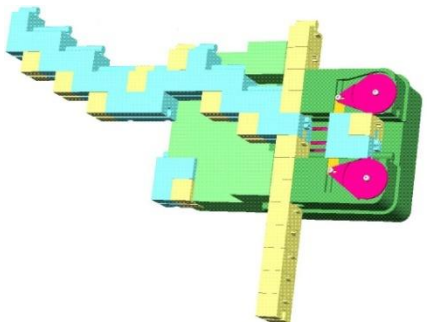


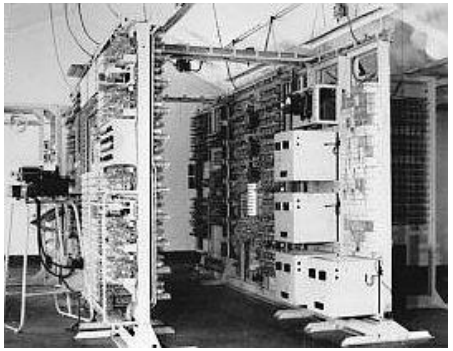
Analytic Engine (schematic)

Alan Turing

- A tape which is divided into cells, one next to the other.
- A head that can read and write symbols on the tape and move left and right.
- A state register that stores the state of the Turing machine
- An action table (or transition function)

Alan Turing (1912-1954)





What is an Algorithm?

An algorithm is made up of instructions (i.e., imperative sentences), input/subject and output/outcome.

1. The instructions must be exact enough to be implemented mechanically;

2. The number of instructions must be finite.

Algorithm Design + Analysis

1. Algorithm Design: describe the algorithm;

2. Algorithm Analysis:

   2.1 Prove the correctness;

   2.2 Analyze the performance, typically, efficiency.

Why Study the Efficiency?

- Given a problem, there are many algorithms to solve it, especially, brute-force search, a general but trivial algorithm.

- We hope the algorithm is as fast as possible. So we need to analyze and compare the algorithms.

How to Study the Efficiency?

How to compare the algorithms with respect to the time efficiency, the space efficiency and so on?!

The general principle of the estimation of the efficiency is the trade-off between precision and operability.

What Affects the Algorithm's Running Time?

- Environment:
  - Pentium v.s. Core;
  - RISC (Reduced Instruction Set Computing) v.s. CISC (Complicated Instruction Set Computing);
  - 32 bits v.s. 64 bits;
  - 2M cache v.s. 8M cache.
  - ... ...
- Input:
  - Small size v.s. large size;
  - Good input v.s. bad input;

Theoretical Methodology

- We consider the arithmetic instructions (i.e., add, subtract, multiply, divide and so on), data movement instructions (i.e., load, store, copy and so on), and control instructions (i.e., subroutine call, return and so on) as elementary computer steps which is machine-independent and cost same and constant time;

- Given a particular input, the running time of an algorithm is the number of elementary computer steps.

Theoretical Methodology

- Describe the running time of the algorithm as a function of the input size.

  Conventionally:

  - For combinatorial problems such as sorting, the input size is the number of items in the input;
  - For numerical problems such as multiplication, the input size is the number of bits in binary to represent the input;
  - For graphical problems, the input size is described with the numbers of vertices and the number of edges in the graph respectively.

- Analyze the worst case (i.e., worst-case analysis).

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.

Polynomial-Time

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

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 $c N^d$ steps.

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

choose $C = 2^d$

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.

simplex method
Unix grep

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

2.2 Asymptotic Order of Growth

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 $\Theta(n^3)$.

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

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.

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

↑ can avoid specifying the base

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

↑ log grows slower than every polynomial

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

↑ every exponential grows faster than every polynomial

2.4 A Survey of Common Running Times

Linear Time: O(n)

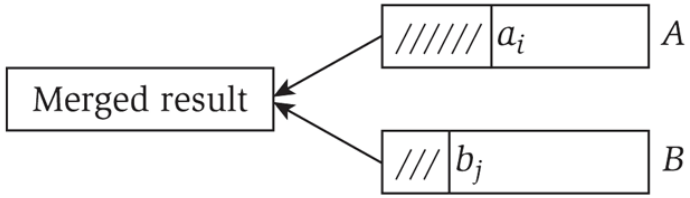
Linear time. Running time is at most a constant factor times the size of the input.

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

```
max ← a1
for i = 2 to n {
    if (ai > max)
        max ← ai
}
```

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 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.

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), …, (x_n, y_n), find the pair that is closest.

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

```
min ← (x1 - x2)^2 + (y1 - y2)^2
for i = 1 to n {
   for j = i+1 to n {
      d ← (xi - xj)^2 + (yi - yj)^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, …, 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 Si {
   foreach other set Sj {
      foreach element p of Si {
         determine whether p also belongs to Sj
      }
      if (no element of Si belongs to Sj)
         report that Si and Sj 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.

```
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
    }
}
```

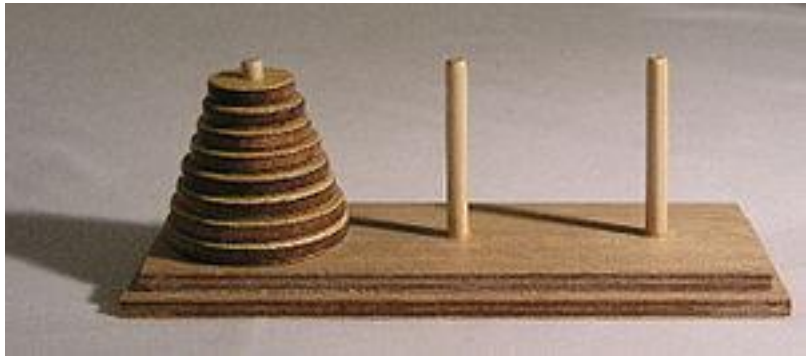
Check whether S is an independent set = $O(k^2)$.

Number of k element subsets = $O(k^2 n^k / k!) = O(n^k)$.

$$\binom{n}{k} = \frac{n\,(n-1)\,(n-2)\cdots(n-k+1)}{k\,(k-1)\,(k-2)\cdots(2)\,(1)} \leq \frac{n^k}{k!}$$

poly-time for k=17, but not practical

Tower of Hanoi



Input: Disks in a neat stack in ascending order of size on one rod, the smallest at the top.

Objective: Move the entire stack to another rod, obeying the following rules:

1. Only one disk may be moved at a time.
2. Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
3. No disk may be placed on top of a smaller disk.

Exponential Time

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

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

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

Leonardo Fibonacci

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases}$$

```
function fib1(n)
if n = 0:   return 0
if n = 1:   return 1
return fib1(n − 1) + fib1(n − 2)
```

```
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i − 1] + f[i − 2]
return f[n]
```

T(n) = T(n - 1) + T(n-2) + 3 for n > 1.

fib1(n) is proportional to $2^{0.694n} \approx (1.6)^n$

fib2 is *linear in* n.