Basics of
Algorithm Analysis

Computational Tractability

"For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing." - *Francis Sullivan*

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*
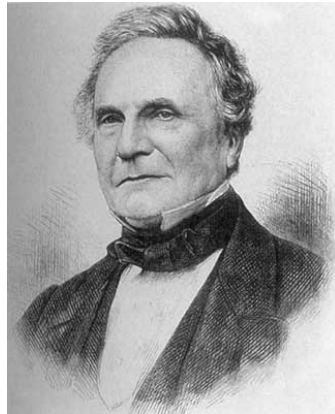
Charles Babbage
(1864)
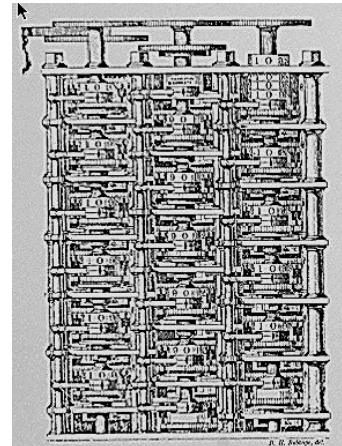
Analytic Engine
(schematic)

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

4

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.

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

Impact of Runtime

	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} yrs
n=50	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 yrs	too much
n=100	< 1 sec	< 1 sec	< 1 sec	1 sec	12892 yrs	10^{17} yrs	too much
n=1000	< 1 sec	< 1 sec	1 sec	18 min	too much	too much	too much
n=10000	< 1 sec	< 1 sec	2 min	12 days	too much	too much	too much
n=100000	< 1 sec	2 sec	3 hrs	32 yrs	too much	too much	too much
n=1000000	1 sec	20 sec	12 days	31710 yrs	too much	too much	too much

Running times of algorithms of varying complexity on inputs of increasing sizes. The assumption is that of a computing system that can perform one million operations per second. > 10^{25} yrs is indicated as "too much".

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 = Ω(g) and g = Ω(h) then f = Ω(h).
- If f = Θ(g) and g = Θ(h) then f = Θ(h).

Additivity.

- If f = O(h) and g = O(h) then f + g = O(h).
- If f = Ω(h) and g = Ω(h) then f + g = Ω(h).
- If f = Θ(h) and g = O(h) then f + g = Θ(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

A Survey of Common Running Times

Linear Time: O(n)

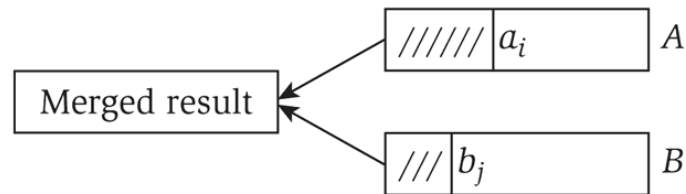Linear time. Running time is at most a constant factor times the size of the input.

Computing the maximum. Compute maximum of n numbers a_1, ..., 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 (a_i ≤ b_j) 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: $O(n^2)$

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        ⟵   don't need to
        if (d < min)                              take square roots
            min ← d
    }
}
```

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion.   ⟵   For later

Cubic Time: $O(n^3)$

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 = $\binom{n}{k} = \dfrac{n\,(n-1)\,(n-2)\,\cdots\,(n-k+1)}{k\,(k-1)\,(k-2)\,\cdots\,(2)\,(1)} \leq \dfrac{n^k}{k!}$
- $O(k^2 \, n^k \,/\, k!) = O(n^k)$.

poly-time for k=17, but not practical

Exponential Time

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

$O(n^2 2^n)$ **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
   }
}
```