2. Algorithm Analysis

2.1 Computational Tractability

Too many solutions → untractable

Polynomial running time: if we have problem to solve find shortest path

2.1.1 Polynomial running time

We want that when the input grows, the algorithm slow down just of a constatime c

> Def. An algorithm is **poly-time** if when the input size doubles, the algorithm should slow down by at most some constant factor C.

An algorithm is efficient if it has a polynomial running time.

2.1.2 Types of analysis

Simple algorithms are not poly-time algorithms.

Worst-case: the maximum considering all input instances, parametrized on n.

Probabilistic: expectation on randomized algorithm, we cannot compute the deterministic value.

Amortized: Worst-case for any sequence of n operations.

Average-case: For random input of size n.

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

2.2.1 Big-Oh

Upper bounds. T(n) is O(f (n)) if there exist constants c > 0 and n0 ≥ 0 such that T(n) ≤ c · f (n) for all n ≥ n0.

Ex. T(n) = 32n2 + 17n + 1.

Typical usage. Insertion sort makes O(n2) compares to sort n elements.
(because n2 is the maximum number of couples)
Alternate definition. T(n) is O(f (n)) if lim sup n→inf T(n)/f(n) < inf.

we assume that the functions involved are (asymptotically) non-negative.

2.2.2 Big-Omega

Lower bounds. T(n) is Ω(f (n)) if there exist constants c > 0 and n0 ≥ 0 such that T(n) ≥ c · f (n) for all n ≥ n0.
Ex. T(n) = 32n2 + 17n + 1.

T(n) is both Ω(n2) and Ω(n).

T(n) is neither Ω(n3) nor Ω(n3 log n).

Any compare-based sorting algorithm requires Ω(n log n) compares in the worst case.

2.2.3 Big-Theta

Tight bounds. T(n) is sandwiched between O and Omega if there exist constants $c1 > 0$, $c2 > 0$, and $n0 \geq 0$ such that $c1 \cdot f(n) \leq T(n) \leq c2 \cdot f(n)$ for all $n \geq n0$.

Ex. $T(n) = 32n2 + 17n + 1$.

- $T(n)$ is $\Theta(n2)$.
- $T(n)$ is neither $\Theta(n)$ nor $\Theta(n3)$.

Proposition. If: lim (n→inf) $f(n)/g(n) = c > 0$, then $f(n)$ is $\Theta(g(n))$.

Polynomials. Let $T(n) = a0 + a1\, n + \ldots + ad\, n^d$ with $ad > 0$. Then, $T(n)$ is $\Theta(n^d)$.

Logarithms. $\Theta(\log a\ n)$ is $\Theta(\log b\ n)$ for any constants $a, b > 0$.

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

Big-Oh for multiple variables: $T(m, n)$ is $O(f(m, n))$ if there exist constants $c > 0$, $m0 \geq 0$, and $n0 \geq 0$ such that $T(m, n) \leq c \cdot f(m, n)$ for all $n \geq n0$ and $m \geq m0$.

2.3 Survey of common running times

2.3.1 Linear time: O(n)

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

E.G.: Merge. Combine two sorted lists A = a1, a2, ..., an with B = b1, b2, ..., bn into sorted whole.

```
i = 1, j = 1
while (both lists are nonempty) {
if (ai ≤ bj) append ai to output list and increment i
else(ai ≤ bj)append bj to output list and increment j
}
append remainder of nonempty list to output list
```

2.3.2 Linearithmic time: O(n log n)

Often in divide-and-conquer (Merge-sort, heap-sort).

2.3.3 Quadratic time: O(n^2)

E.g.: Enumerate all pairs of elements.

O(n2) 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
  }
}
```

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

2.3.4 Cubit time: O(n^3)

E.g.: Enumerate all triples of elements.

```
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
  }
}
```

2.3.5 Polynomial time: O(n^k)

Independent set of size k.

O(n^k) solution. 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)
    report S is an independent set
  }
}
```

2.3.6 Exponential time

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

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

2.3.7 Sublinear time

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