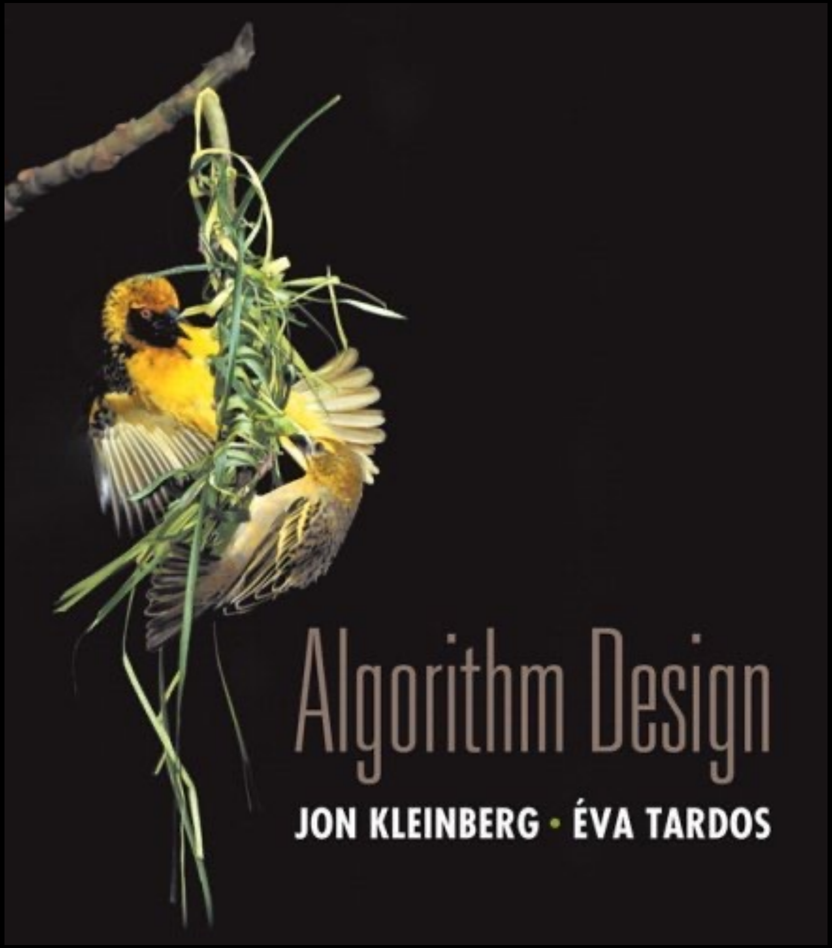


Chapter 2

Basics of
Algorithm Analysis

2.1 Computational Tractability

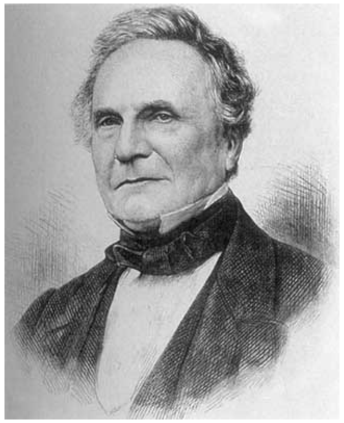
Tractability: 易处理性;
(of a situation or problem)
easy to deal with: *trying to make the mathematics tractable.*

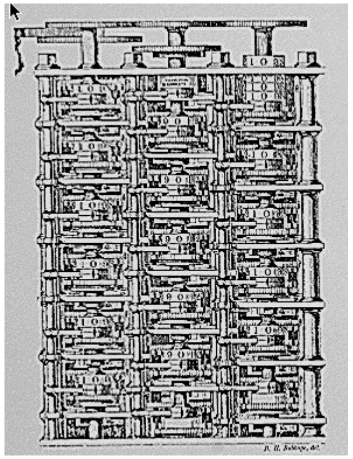
"For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing." - *Francis Sullivan*

Annotated and slightly changed by Yang Xu (徐炀)
Contact: xuyang@sustech.edu.cn
Not for commercial use.

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)

Warm-up Game

Insertion sort VS. merge sort

Insertion sort overview

Input: A sequence of n numbers $\langle a_1, a_2, \cdots, a_n \rangle$
Output: A permutation (reordering) $\langle a'_1, a'_2, \cdots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

Insertion sort
Similar to the way of sorting poker cards:

- Start with an empty hand, and a pile of cards facing down on the table
- Take one card at a time from the table and insert it into the **correct** position in the hand
- To find the correct position, we compare it with each of the cards already in the hand

Pseudocode for insertion sort

```
INSERTION-SORT($A$)
1. for $j = 2$ to $A.length$
2.     $key = A[j]$
3.     // Insert $A[j]$ into the sorted sequence $A[1 .. j-1]$
4.     $i = j - 1$
5.     while $i > 0$ and $A[i] > key$
6.         $A[i + 1] = A[i]$
7.         $i = i - 1$
8.     $A[i + 1] = key$
```

Takes as input an array $A[1..n]$, n numbers to be sorted

Sorts in place: rearranges the numbers within A; no extra memory costed

Merge sort overview

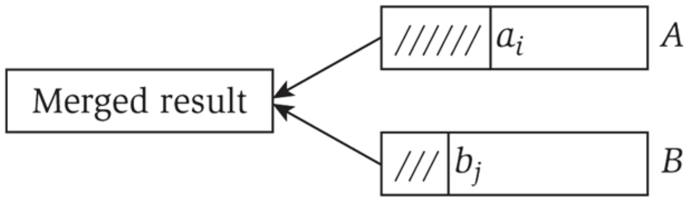
Divide: Divide the n-element sequence to be sorted into *two* subsequences of $n/2$ elements each

Conquer: Sort the two subsequences *recursively* using merge sort

Combine: Merge the two sorted subsequences to produce the sorted answer

The recursion reaches the "bottom" when the sequence to be sorted has length 1, in which case there is no work to be done.

The key is the *merging* of two sorted sequences in "combine" step.



```
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          append $b_j$ to output list and increment j
}
append remainder of nonempty list to output list
```

Design the MERGE procedure

A card-playing analogy:

- Two sorted piles of cards as input, with smallest cards on top
- Wish to merge the two piles into a single sorted output pile
- Strategy: choose the smaller of the two cards on top of the two input piles, and place it onto the output pile
- Repeat this step until one input pile is empty
- Take the remaining input pile and place it onto the output pile

Each basic step takes constant time (just comparing two cards).
Since we perform at most n basic steps, merging takes $\Theta(n)$ time.

Warm-up question

Example:

insertion sort → takes $c_1 n^2$ instructions to sort n items

merge sort → takes $c_2 n \cdot \lg n$ instructions to sort n items

A faster computer (computer A) running insertion sort

- Execute 10 billion (10^{10}) instructions per second
- Low-level programming language, $c_1 = 2$

A slower computer (computer B) running merge sort

- Execute 10 million (10^7) instructions per second
- High-level programming language, $c_2 = 50$

Sort 10 million (10^7) numbers

Warm-up question

Computer A takes:

$$\frac{2 \cdot (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}} = 20{,}000 \text{ second} \approx 5.6 \text{ hours}$$

Computer B takes:

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 1163 \text{ second} \approx 19.4 \text{ minutes}$$

Computer B runs 17 times faster than computer A!
What about sorting 100 million numbers?

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.

这里指问题空间的大小，非G-S算法时间复杂度

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 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.

  Adj. 严苛的

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

Why It Matters

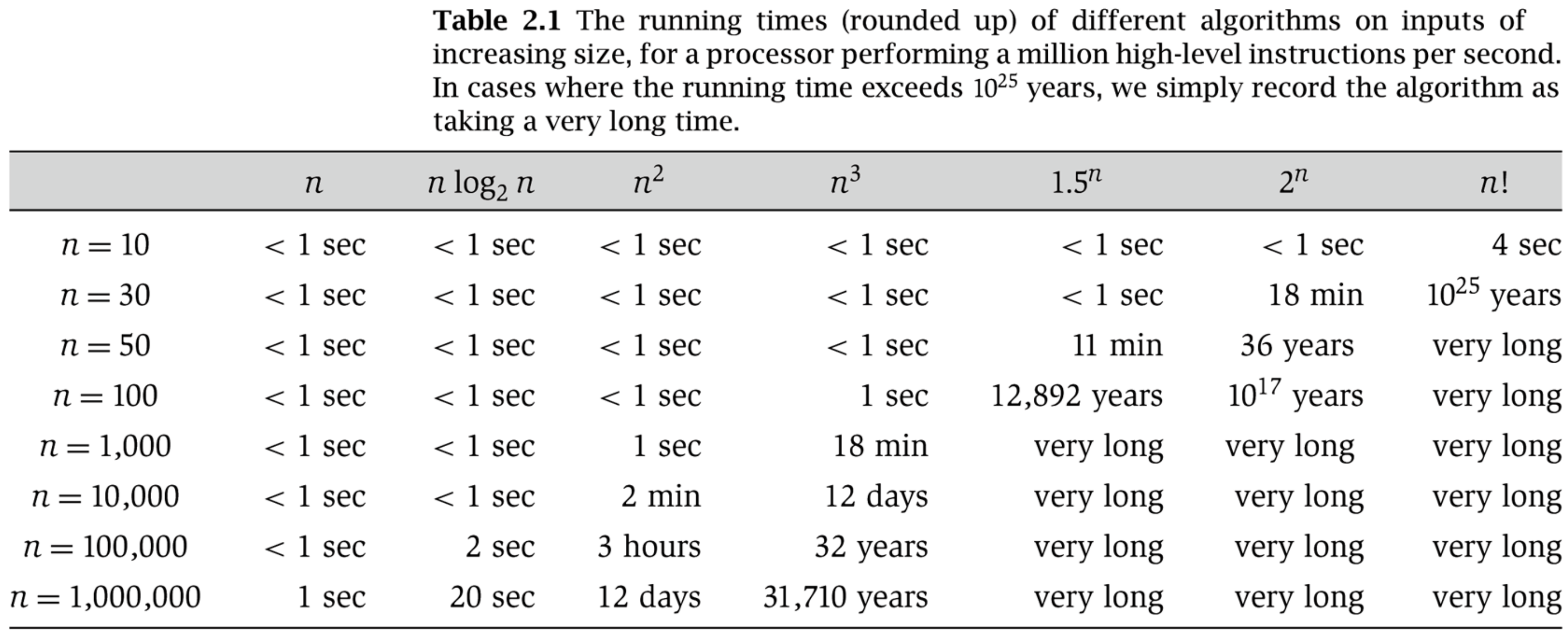
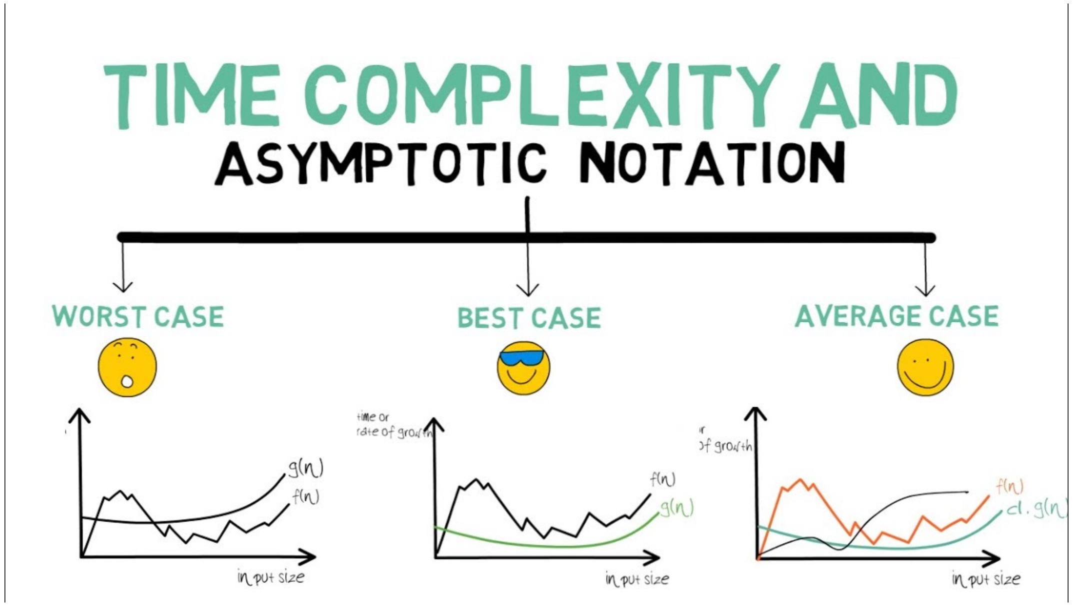


Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	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: approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken)
Source: https://mathworld.wolfram.com/Asymptotic.html



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 $T(n) \leq c \cdot f(n)$.

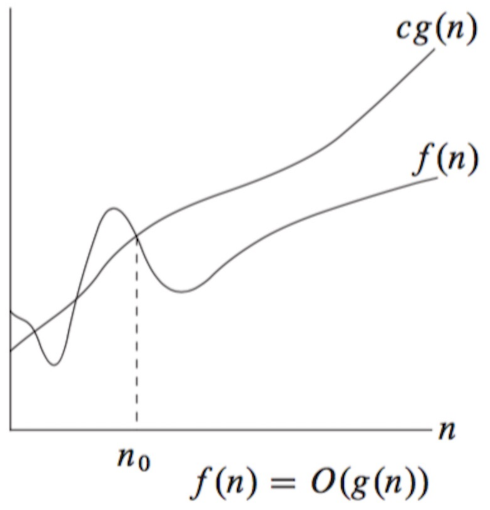
Lower bounds. T(n) is Ω(f(n)) if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

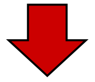
Tight bounds. T(n) is Θ(f(n)) if T(n) is both O(f(n)) and Ω(f(n)).

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

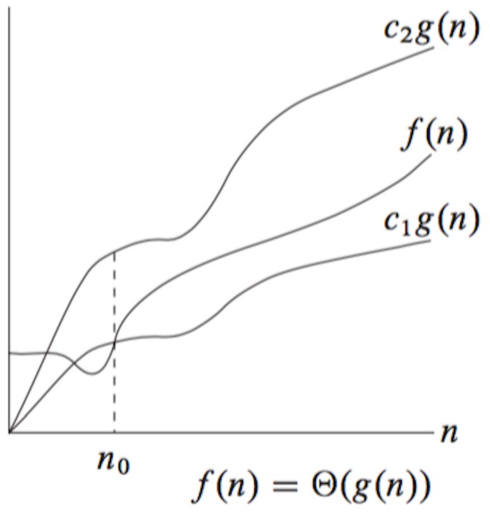
- T(n) is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
- T(n) is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.

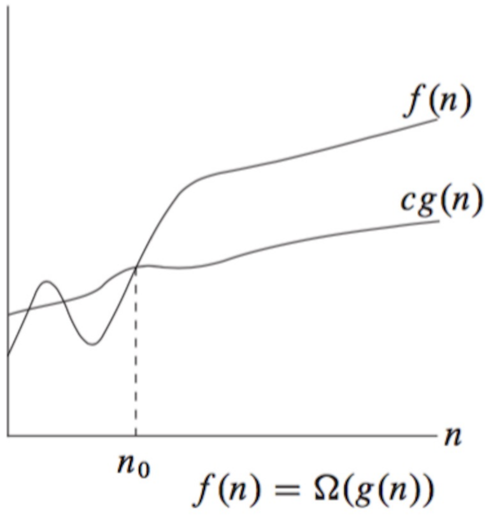
Intuitive understanding of Big O, Omega, Theta





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





Notation

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

- Not transitive:
  - $f(n) = 5n^3$; $g(n) = 3n^2$
  - $f(n) = O(n^3) = g(n)$
  - 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.

Type check: check the correctness of the program before its execution

Properties

Transitivity. 传递性

- 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. 相加性

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

can avoid specifying the base

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

Proving asymptotic bounds
Case 1: Ignore lower order terms

$f(n) = \frac{1}{2}n^2 - 3n = \Theta(n^2)$ → Why?

Need to determine positive constants c_1, c_2, and n_0 such that

$$c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2, \text{ for all } n \ge n_0$$

Dividing by n^2 yields

$$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$$

If we choose any constant $c_2 \ge \frac{1}{2}$, the right inequality always holds

Therefore, by choosing $c_1 = \frac{1}{14}$, $c_2 = \frac{1}{2}$, and $n_0 = 7$, $\frac{1}{2}n^2 - 3n = \Theta(n^2)$ holds

When $n \ge 7$, $\frac{1}{2} - \frac{3}{n} \ge \frac{1}{14}$, thus for any constant $c_1 \le \frac{1}{14}$, the left inequality always holds

Proving asymptotic bounds
Case 2: Higher order functions

To verify that $6n^3 \neq \Theta(n^2)$

Suppose that c_2 and n_0 exist such that $6n^3 \leq c_2 n^2$ holds for all $n \geq n_0$

Dividing by n^2 yields:

$$n \leq \frac{c_2}{6}$$

This is a contradiction because n can be arbitrarily large.

Therefore, $6n^3 \neq \Theta(n^2)$

Proving asymptotic bounds
Case 3: A generic quadratic function

$f(n) = n^2 + 2n + 1$

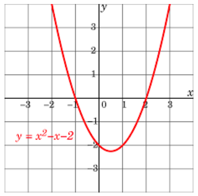
To show $f(n) = \Theta(n^2)$, we need to find c_1, c_2, and n_0 such that

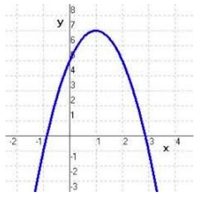
$$c_1 n^2 \leq n^2 + 2n + 1 \leq c_2 n^2, \text{ for all } n \geq n_0$$

The inequalities become:

$\mathbf{0 \leq (1 - c_1)n^2 + 2n + 1}$ and $\mathbf{(1 - c_2)n^2 + 2n + 1 \leq 0}$

Thus, we need to have: $c_1 < 1 < c_2$





And that n be greater than the larger root of the two equations:

$$\begin{cases} (1 - c_2)n^2 + 2n + 1 = 0 \\ (1 - c_1)n^2 + 2n + 1 = 0 \end{cases}$$

c_1 and c_2 can have multiple possible values, as long as $c_1 < 1 < c_2$

if $c_1 = \frac{1}{4}$ and $c_2 = \frac{9}{4}$, then $n_0 = 2$

2.4 A Survey of Common Running Times

From code to time complexity

Insertion sort example

Assume: Each execution of the *i*th line takes c_i time (a constant cost, in *seconds*).
Each line is executed for a certain number of *times* (*integer numbers*)

```
INSERTION-SORT(A)
1.  for j = 2 to A.length
2.          key = A[j]
3.          // Insert A[j] into the sorted sequence A[1 .. j-1]
4.          i = j – 1
5.          while i > 0 and A[i] > key
6.                  A[i + 1] = A[i]
7.                  i = i – 1
8.          A[i + 1] = key
```

	Cost	Number of times
1.	c_1	n
2.	c_2	$n-1$
3.	0	$n-1$
4.	c_4	$n-1$
5.	c_5	?
6.	c_6	?
7.	c_7	?
8.	c_8	$n-1$

$n \rightarrow$ After the last iteration, j = n+1

How many times are the **while** statement executed?

For each j (j = 2, ..., n), the while loop test is executed t_j times, which is *unknown*

Therefore, the total number of times is $\sum_{j=2}^{n} t_j$

From code to time complexity (cont.)

```
INSERTION-SORT(A)
1.  for j = 2 to A.length
2.      key = A[j]
3.      // Insert A[j] into the sorted sequence A[1 .. j-1]
4.      i = j – 1
5.      while i > 0 and A[i] > key
6.          A[i + 1] = A[i]
7.          i = i – 1
8.      A[i + 1] = key
```

Cost	Number of times
c_1	n
c_2	$n-1$
0	$n-1$
c_4	$n-1$
c_5	$\sum_{j=2}^{n} t_j$
c_6	$\sum_{j=2}^{n} (t_j - 1)$
c_7	$\sum_{j=2}^{n} (t_j - 1)$
c_8	$n-1$

Same reason as the for loop

Total running time $T(n)$ => the ***sum*** of running times for each statement executed

A statement that costs c_i to execute and executes n times will contribute $c_i n$ to the total

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Running time of insertion sort: <u>best</u> case

The best case: input array is *already sorted*.

- For any $i < j$, we always have $A[i] \leq A[j]$
- $A[i] \leq key$ always breaks the while loop (Line 5)
- Thus $\boxed{\boldsymbol{t_j = 1}}$ for $j = 2,3,\cdots n$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

$\boldsymbol{T(n) = an + b}$, where a and b are constants

I.e., $T(n)$ is a linear function of n (for the best case)

```
INSERTION-SORT(A)
1.   for j = 2 to A.length
2.       key = A[j]
3.       // Insert A[j] into the sorted sequence A[1 .. j-1]
4.       i = j – 1
5.       while i > 0 and A[i] > key
6.           A[i + 1] = A[i]
7.           i = i – 1
8.       A[i + 1] = key
```

Running time of insertion sort: **worst** case

The worst case is that the input array is in *reverse sorted* order

- The key = $A[j]$ needs to be compared with each element in $A[1\,..\,j-1]$
- Thus, $\boxed{\boldsymbol{t_j = j}}$ for $j = 2,3,\cdots n$

Note that: $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1, \sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

Simplified as $\boldsymbol{an^2 + bn + c}$

$T(n)$ is a quadratic function of n (in the worst case)

Worst-case and average-case analysis

Worst-case running time:

- It gives an upper bound for any input (will never take any longer)
- The worst case occurs fairly often

For insertion sort, the **average case** as bad as the worst case

- Suppose n random numbers as input
- Question: what is t_j(on average)?
- A fair guess: half the elements in $A[1..j-1]$ are less than $A[j]$, and the other half are greater
- Thus, $\boxed{\boldsymbol{t_j} \textbf{ is about } \boldsymbol{j/2}}$
- $T(n) = \sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j/2 = \frac{1}{2}\sum_{j=2}^{n} j$

Summary: Order of growth

For insertion sort: $T(n) = an^2 + bn + c$

It is the order of growth (rate of growth) that really matters

- Keep the highest order term $\boldsymbol{an^2}$
- Ignore the constant coefficient
- $\boldsymbol{T(n) = O(n^2)}$, i.e., $T(n)$ grows as fast as the function n^2

Linear Time: O(n)

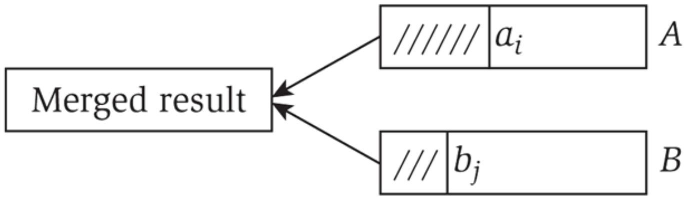
Linear time. Running time is proportional to input size.

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

```
max ← 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 ($a_i \le b_j$) 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.

$\uparrow$ 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: $O(n^k)$

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 = $O(k^2 n^k / k!) = O(n^k)$.

First, the total number of *k*-element subsets in an *n*-element set:

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

Check for each pair of nodes in a set of size k: $O(k^2)$

Exponential Time

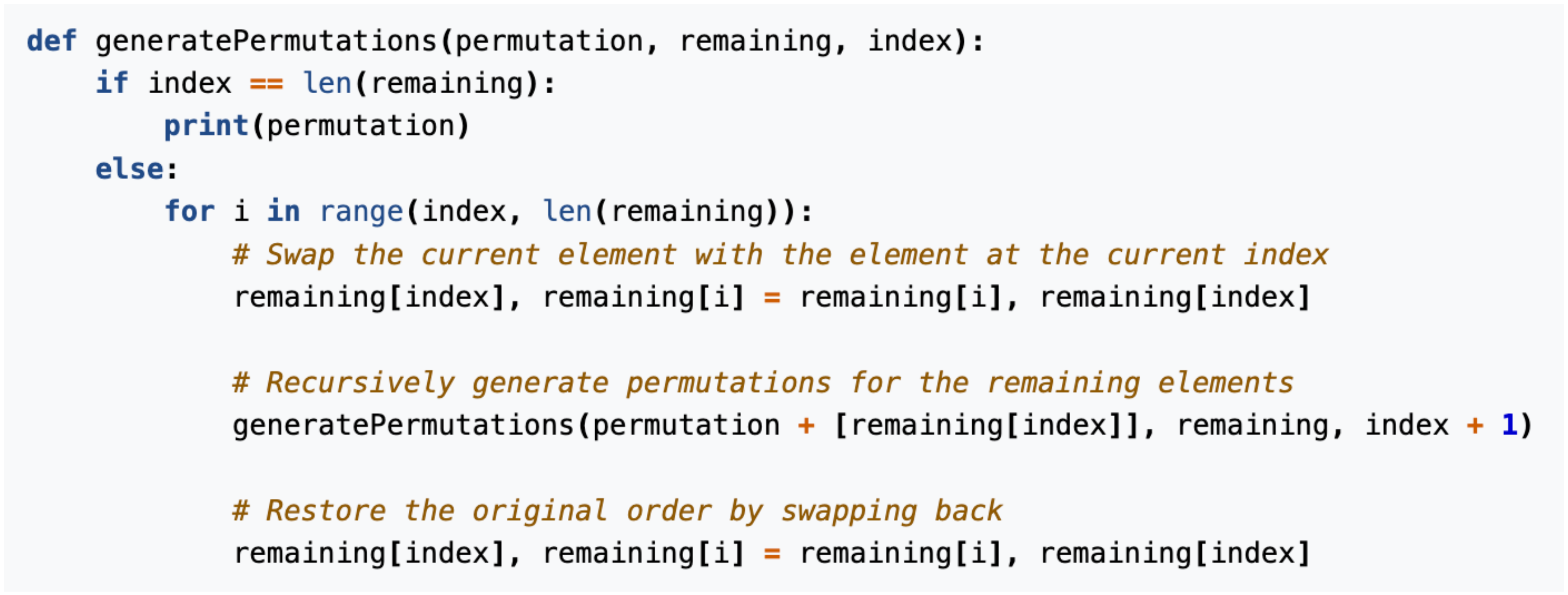
Independent set. Given a graph, what is maximum size of an independent set?

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

Q: What’s the total number of subsets?

Factorial Time: O(n!)



```python
def generatePermutations(permutation, remaining, index):
    if index == len(remaining):
        print(permutation)
    else:
        for i in range(index, len(remaining)):
            # Swap the current element with the element at the current index
            remaining[index], remaining[i] = remaining[i], remaining[index]

            # Recursively generate permutations for the remaining elements
            generatePermutations(permutation + [remaining[index]], remaining, index + 1)

            # Restore the original order by swapping back
            remaining[index], remaining[i] = remaining[i], remaining[index]
```

Source: https://saturncloud.io/blog/example-of-a-factorial-time-algorithm-on