Analysis & Design of Algorithms

Angel Napa

January 1, 2026

1 InsertionSort: analysis and correctness

Consider the following sorting problem

Input: Sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$

Output: Permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ de A such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

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

Figure 1: Cormen, introduction of algorithms

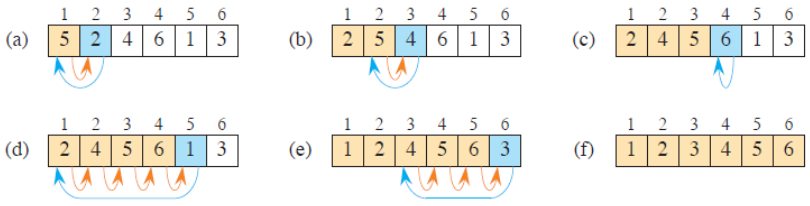

Figure 2: Cormen, Introduction to algorithms

1.1 Correctness analysis

Invariant: At the start of each iteration of lines 1–8, the subarray $A[1 \ldots i-1]$ consists of the elements originally in $A[1 \ldots i-1]$ but in sorted order.

We will prove that this invariant holds.

Inicialization: The invariant is true before the first iteration. This is because for $i = 2$, we have $A[1 \dots i-1] = A[1]$ that consists of the original elements in $A[1]$ but sorted.

Maintenance: Suppose the invariant holds at the beginning of the i-th iteration. This means $A[1 \dots (i-1)]$ is sorted. Because of the while loop (lines 5–7), the element $A[i]$ is inserted in the position $j+1$ such that the subarray $A[1 \dots i]$ is sorted. (Obs: a complete proof should show why $A[i]$ is properly inserted, using another invariant for this while loop, see below).

Termination: At the start of the last iteration, we have $i = n+1$. This means $A[1 \dots i-1] = A[1 \dots n]$ is sorted, concluding the proof.

As mentioned above, a more complete proof should show with another invariant that the loop do what was asked. The invariants that allow us to show this are the following:

Let A' the array A at the beginning of the (lines 5–7). Then, at the start of each iteration of the while loop, we have:

- $key \leq A[j+2 \dots i]$
- $A[j+2 \dots i] = A'[j+1 \dots i-1]$
- $A[1..i] = A'[1..i]$

It is left as an exercise to the reader to prove these invariants. Note that the termination of these invariants implies that $A[1 \dots j] \leq key \leq A[j+2 \dots i]$, and thus, after executing line 8, we obtain that $A[1 \dots i]$ is in fact sorted, as we wanted.

1.2 Execution time analysis

In this course, we will assume the RAM model of computation. This means instructions are executed sequentially (without parallelism). Therefore, atomic operations like addition, subtraction, multiplication, and assignment take constant time. This way each atomic instruction such as addition, subtraction, multiplication, assign, etc., takes constant time.

Let $T(n)$ the total execution time of the algorithm. For each $i = 2 \dots n$, let t_i the number of times the while loop in line 5 is executed for that value of i. For each $k = 1 \dots 8$, let c_k the time it takes to execute the line k. Note that this time is constant.

Then,

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1) \\ &= (c_5 + c_6 + c_7) \sum_{i=2}^{n} t_i + (c_1 + c_2 + c_4 + c_8 - c_6 - c_7)n + (c_7 + c_6 - c_2 - c_4 - c_8) \quad (1) \end{aligned}$$

Note that $t_i \leq i$. Then, by (1),

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

Figure 3: Taken from Cormen, Introduction to Algorithms

$$\begin{aligned} T(n) &\leq (c_5 + c_6 + c_7) \sum_{i=2}^{n} i + (c_1 + c_2 + c_4 + c_8 - c_6 - c_7)n + (c_7 + c_6 - c_2 - c_4 - c_8) \\ &= (c_5 + c_6 + c_7)(\frac{n(n-1)}{2} - 1) + (c_1 + c_2 + c_4 + c_8 - c_6 - c_7)n + (c_7 + c_6 - c_2 - c_4 - c_8) \\ &= \frac{1}{2} n^2 (c_5 + c_6 + c_7) + (c_1 + c_2 + c_4 + c_8 - c_5/2 - 3c_6/2 - 3c_7/2)n + (c_7 - c_5 - c_2 - c_4 - c_8). \end{aligned}$$

Let $a = (c_5 + c_6 + c_7)/2, b = c_1 + c_2 + c_4 + c_8 - c_5/2 - 3c_6/2 - 3c_7/2$ y $c = c_7 - c_5 - c_2 - c_4 - c_8$. The above inequality proves the following theorem:

Theorem 1.1. *There exist some constants a, b and c, with $a > 0$ such that the following holds*

$$T(n) \leq an^2 + bn + c, \quad \forall n \in \mathbb{Z}^+$$

On the other hand, note that $t_i \geq 1$. Hence, by (1),

$$\begin{aligned} T(n) &\geq (c_5 + c_6 + c_7) \sum_{i=2}^{n} 1 + (c_1 + c_2 + c_4 + c_8 - c_6 - c_7)n + (c_7 + c_6 - c_2 - c_4 - c_8) \\ &= (c_1 + c_2 + c_4 + c_8 + c_5)n + (-c_5 - c_2 - c_4 - c_8) \end{aligned}$$

Let $a = (c_1 + c_2 + c_4 + c_8 + c_5)/2$ y $b = (-c_5 - c_2 - c_4 - c_8)$. The above inequality proves the following theorem.

Theorem 1.2. *There exists constants a and b, with $a > 0$, such that*

$$T(n) \geq an + b, \quad \forall n \in \mathbb{Z}^+$$

Note that if $A[1 \ldots n]$ was already ordered in an increased way, then t_i=1 for all i. Also, if $A[1 \ldots n]$ was sorted in a decreased way, we have $t_i = i$. This implies that there exist some inputs such that the above bounds are the best we can find.

1.3 Worst case & average case Analysis

We are interested in the worst case scenario: the longest execution time for any input of size n. This will give us the best upper bound. Usually the average

case is "as bad" as the worst case. But we see some techniques to compute the average case when we study the Quicksort Analysis

Observation 1.1. *Usually the input size depends on the problem. For example, in the sorting problem, the input size is the number of elements to order. If the problem is to multiply two numbers, the input size is the number of bits used to represent those numbers. If the problem receives a graph as an input, the input size is the number of vertices, and the number of edges of such graph.*

2 Order of Growth

It's not necessary having an exact execution time of an algorithm. If the input is large enough, the constants present in the formula become irrelevant to understand how fast or slow is the algorithm, and also the terms with lower order. We are interested in the asymptotic growth of the algorithms, this means, how they behave when n tends to infinity.

2.1 Big-O Notation

Given a function $g(n)$, we define $O(g(n))$ as

$$O(g(n)) = \{f(n) : \text{there exist constants } c, n_0$$
$$\text{such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

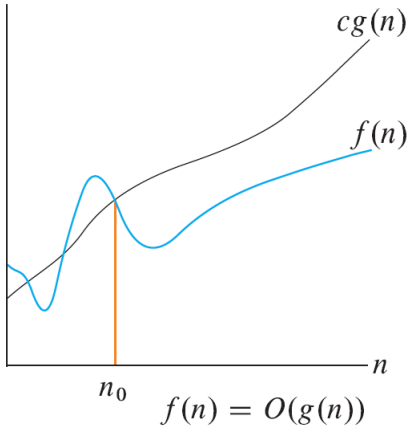

Figure 4: Cormen, Introduction to Algorithms

Example 2.1. *Show that*

$$n^2 + 10n + 2 = O(n^2)$$

(Draft: $n^2 + 10n + 2 \leq cn^2$. We will prove the example using $c = 3$.)

Proof. Note that for $n \geq 10$, we have that $10n \leq n^2$ Then $n^2 + 10n + 2 \leq n^2 + n^2 + n^2 = 3n^2$. Hence, we conclude that $n^2 + 10n + 2 = O(n^2)$. □

Example 2.2. *Show that*

$$\frac{n^2}{2} + 3n = O(n^2)$$

(Draft: $n^2/2 + 3n \leq cn^2$. We will show that, using $c = 1$ we will have that $3n \leq n^2/2$. The example follows for n greater or equal than 6.)

Proof. Note that for $n \geq 6$, we have $6n \leq n^2$. Then $3n \leq n^2/2$.

Thus,

$$0 \leq n^2/2 + 3n \leq n^2/2 + n^2/2 = n^2$$

This proves that $n^2/2 + 3n = O(n^2)$ (because $0 \leq n^2/2 + 3n \leq cn^2$ for $n \geq n_0$, with $n_0 = 6$ and $c = 1$). □

Example 2.3. *Show that $n/100$ is not $O(1)$.*

Proof. Assume by contradiction that $n/100 = O(1)$. This means there exist constants $n_0, c > 0$ such that $\frac{n}{100} \leq c$ for all $n \geq n_0$. Since $n_0 \geq n_0$ we have that $n_0/100 \leq c$ which implies that $n_0 \leq 100c$. Take $n = \lceil 200c + n_0 + 1 \rceil$ and note that $n \geq n0$, but $\frac{n}{100} = \frac{200c}{100} = 2c > c$. This is a contradiction. □

Example 2.4. *Show that $an + b = O(n^2)$ for all $a > 0$.*

(Draft: take $c = a + |b|$ and $n_0 = \max\{1, -b/a\}$.)

Proof. Note that, when $n \geq \max\{1, -b/a\}$, we have $n \geq 1$ and $n \geq -b/a$. This implies

$$an + b \geq a(-b/a) + b \geq 0$$

and

$$an + b \leq an^2 + |b| \leq (a + |b|)n^2$$

This proves

$$0 \leq an + b \leq (a + |b|)n^2$$

for all $n \geq \max\{1, -b/a\}$. □

Observation 2.1. *Big-O is useful to upper bound the execution time of the worst-case scenario of an algorithm, and thus, each case of the algorithm.*

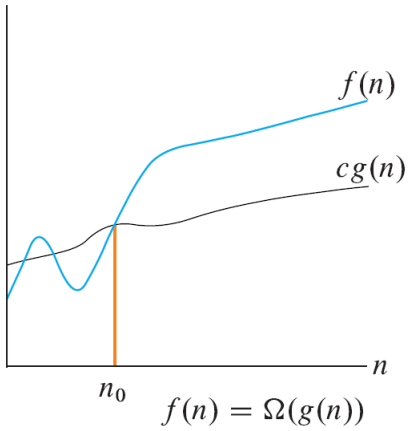

Figure 5: Cormen, Introduction to Algorithms

2.2 Big Ω notation

Given a function $g(n)$, we define $\Omega(g(n))$ as

$$\Omega(g(n)) = \{f(n) : \text{there exist constants } c, n_0$$
$$\text{such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

Observation 2.2. *Ω is used to lower bound the best case scenario of an algorithm, and thus, lower bound each case.*

Observation 2.3. *If $T(n)$ is the execution time function for* InsertionSort *with size n input, then $T(n) = O(n^2)$ y $T(n) = \Omega(n)$.*

2.3 Big-Θ Notation

Given a function $g(n)$, we define $\Theta(g(n))$ as

$$\Theta(g(n)) = \{f(n) : \text{there exist constants } c_1, c_2, n_0$$
$$\text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}.$$

Since $\Theta(g(n))$ is a set, we can also say that $f(n) \in \Theta(g(n))$. We can also say $f(n) = \Theta(g(n))$. And $f(n)$ *is* $\Theta(g(n))$.

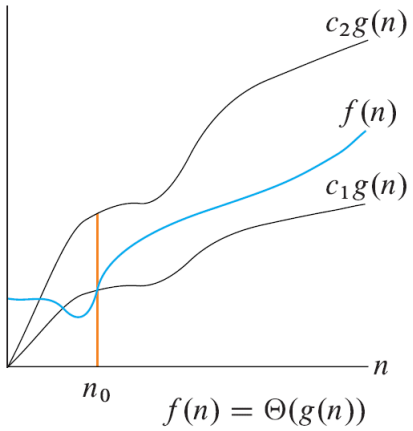

Figure 6: Cormen, Introduction to Algorithms

Theorem 2.1. *$f = \Theta(g(n)$ if and only if $f = O(g(n))$ y $f = \Omega(g(n))$*

Example 2.5. *Show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.*

(Draft: $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$. Then $c_1 \leq \frac{1}{2} - 3/n \leq c_2$. Using $c_2 = 1/2$. When $n = 7$, we have $c_1 \leq 1/2 - 3/7 = 1/14$.)

Proof. Note that $n \geq 7$, we have that

$$\frac{1}{2}n^2 - 3n = n^2(\frac{1}{2} - \frac{3}{n}) \geq \frac{n^2}{14}$$

it also holds that

$$\frac{1}{2}n^2 - 3n \leq \frac{1}{2}n^2$$

Then, for $c_1 = \frac{1}{14}, c_2 = \frac{1}{2}, n_0 = 7$, we have that

$$0 \leq c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

for all $n \geq n_0$. This proves that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. □

Example 2.6. *Show that $6n^3 \neq \Theta(n^2)$.*

Proof. Suppose by contradiction that $6n^3 = \Theta(n^2)$. This means there exist constants $c_1, c_2, n_0 > 0$ such that $0 \leq c_1 n^2 \leq 6n^3 \leq c_2 n^2$. Then, $6n \leq c_2$ for all $n \geq n_0$. This is a contradiction, because c_2 is a constant. □

Exercise 2.1. *$an^2 + bn + c = \Theta(n^2)$ for all $a > 0$.*

Proof. When $n \geq \max\{\frac{2|b|}{a}, 2\sqrt{\frac{|c|}{a}}\}$ we have

$$\frac{|b|}{n} \leq \frac{|b|}{\max\{\frac{2|b|}{a}, 2\sqrt{\frac{|c|}{a}}\}} \leq \frac{|b|}{2|b|/a} = a/2.$$

Then

$$\frac{-a}{2} \leq b/n \leq a/2. \tag{2}$$

Similarly,

$$\frac{|c|}{n^2} \leq \frac{|c|}{\max\{\frac{2|b|}{a}, 2\sqrt{\frac{|c|}{a}}\}} \leq \frac{|c|}{4|c|/a} = a/4.$$

Thus,

$$\frac{-a}{4} \leq c/n^2 \leq a/2. \tag{3}$$

From (1) y (2),

$$\frac{a}{4} \leq a + \frac{b}{n} + \frac{c}{n^2} \leq \frac{7a}{4}.$$

This means

$$0 \leq an^2 \leq an^2 + bn + c \leq \frac{7a}{4}n^2.$$

□

2.4 Little-o notation

Given a function $g(n)$, we define $o(g(n))$ as

$$o(g(n)) = \{f(n) : \text{for each constant } c > 0$$

$$\text{there is a constant } n_c \text{ such that } 0 \leq f(n) < cg(n) \text{ para todo } n \geq n_c\}$$

Example 2.7. $2n = o(n^2)$

(Draft. We want $2n < cn^2$. This means $n > 2/c$. We take $n_0 = 1 + \frac{2}{c}$.)

Proof. let $c > 0$ an arbitrary constant. Note that, for each $n \geq 1 + \frac{2}{c}$, we have $c + 2 \leq nc$. Since n is positive, we also have $nc + 2n \leq cn^2$, thus,$2n < cn^2$. □

Example 2.8. $2n^2 \neq o(n^2)$

Proof. Assume by contradiction that $2n^2 = o(n^2)$. Take $c = 1$, we should have $2n^2 < n^2$, for all n greater or equal to some constant n_0, this is impossible. Thus, we have a contradiction. □

Observation 2.4. *$f(n) = o(g(n))$ if and only if* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

2.5 Little-ω notation

Given a function $g(n)$, we define $\omega(g(n))$ as

$$\omega(g(n)) = \{f(n) : \text{for each constant } c > 0$$
$$\text{there exists a constant } n_c \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_c\}$$

Observation 2.5. *$f(n) = \omega(g(n))$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.*

2.6 Comparing functions

There are obvious comparisons between the different notations

Transitivity

- $f(n) = \Theta(g(n))$, $g(n) = \Theta(h(n))$, imply $f(n) = \Theta(h(n))$
- $f(n) = O(g(n))$, $g(n) = O(h(n))$, imply $f(n) = O(h(n))$
- $f(n) = \Omega(g(n))$, $g(n) = \Omega(h(n))$, imply $f(n) = \Omega(h(n))$
- $f(n) = o(g(n))$, $g(n) = o(h(n))$, imply $f(n) = o(h(n))$
- $f(n) = \omega(g(n))$, $g(n) = \omega(h(n))$, imply $f(n) = \omega(h(n))$

Reflexivity

- $f(n) = \Theta(f(n))$
- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$

Symmetry

- $f(n) = \Theta(g(n))$ implies $g(n) = \Theta(f(n))$

Transpose symmetry

- $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$
- $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$

It is left as an exercise to the reader to prove these equations

Observation 2.6. *There are functions that are not comparable, such as n and $n^{1+\sin n}$.*