

# Chapter 1: Introduction (Role of Algorithms in Computing)

## Algorithms

- An **algorithm** specifies a sequence of computational steps to solve a well-defined computational problem. An algorithm should be precise, correct, and finite.
- A problem specifies the desired input/output relationship.
- Example: The *sorting* problem:
  - Input: a sequence of  $n$  numbers  $\langle a_1, a_2, \dots, a_n \rangle$ .
  - Output: A permutation (reordering)  $\langle a'_1, a'_2, \dots, a'_n \rangle$  of the input sequence such that  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ , that is, monotonically increasing.
  - The numbers to be sorted are known as **keys**. For real data, there is often **satellite data** associated with a key that moves with the key. The key together with the satellite data is referred to as a **record**. Give example of a spreadsheet.
- Algorithms are often specified in **pseudo-code**. Pseudo-code abstracts away the details of actual programming languages so we can focus on the essence of an algorithm. Pseudo-code often ignores aspects of software engineering – such as data abstraction, modularity, and error handling to again focus on the essence of the algorithm.
- To be able to express algorithms using pseudo-code is a higher level skill than expressing them in a specific programming language. This is something we will cultivate this semester, even though we will still do plenty of software engineering and actual coding!
- Here we will present **insertion sort**, which is an efficient algorithm for sorting a small number of elements.
- Show example with a hand of playing cards.
- Show example with the input sequence  $\langle 5, 2, 4, 6, 1, 3 \rangle$ .

```
INSERTION-SORT(A, n)
// Input is A[1]..A[n] or A[1:n]
// Output is A[1]..A[n] or A[1:n], but now sorted
1. for i = 2 to n
2.     key = A[i]
3.     // Insert A[i] into the sorted portion 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.  $A[j + 1] = \text{key}$

**Exercise 2.1-1:** Use INSERTION-SORT to sort  $A = \langle 31, 41, 59, 26, 41, 58 \rangle$ . Show the intermediate steps.

**In-class Exercise:** Use INSERTION-SORT to sort  $A = \langle 1, 2, 3, 4, 5 \rangle$ . Show the intermediate steps.

**In-class Exercise:** Use INSERTION-SORT to sort  $A = \langle 5, 4, 3, 2, 15 \rangle$ . Show the intermediate steps.

What is the best-case for insertion sort? What is the worst-case for insertion sort?

- **Correctness.** We use the **loop invariant** to show the correctness. A **loop invariant** is a property of a program loop that is true before and after each iteration.
- **INSERTION-SORT Loop Invariant:** At the start of each iteration of the for loop of lines 1-8, the subarray  $A[1:i-1]$  consists of elements originally in  $A[1:i-1]$ , but in sorted order.
- Prove that the loop invariant holds at **initialization**, is **maintained** at the start of each iteration, and at **termination** provides with a useful property that helps show that the algorithm is correct.
- Review pseudo-code conventions (pages 21–24 in the textbook)
- **Exercise 2.1-3:** Rewrite INSERTION-SORT pseudo-code to sort into monotonically decreasing order. Answer: Modify Line 5.
- **Exercise 2.1-2:** State loop invariant for the SUM-ARRAY procedure. Use it to prove the correctness of the procedure.
- **Exercise 2.1-4:** Write pseudo-code for linear search and come up with the loop invariant to prove its correctness. [Homework]

## Algorithms as a technology

- To **analyze** an algorithm means to estimate the resources that the algorithm requires to finish. Resources include running time, memory, communication bandwidth, or energy consumption.
- The most useful measure is the running time of an algorithm in terms of the input size  $n$ . We express the running time as a function of  $n$ .

- We assume the Random Access Machine (RAM) model to estimate the costs for the basic steps (instructions) of an algorithm. We assume (based on real hardware) that each instruction takes a constant amount of time (with some assumptions). Browse pages 26–27 on the rationale for this simplifying assumption.
- In most cases, we want to analyze the **worst-case** runtime of an algorithm. Sometimes, we also analyze the **best-case** run time for an algorithm.
- In some particular cases, we are also interested in the **average-case** or **expected** running time of an algorithm. However, the average-case is often as bad as the worst-case.
- Let us analyze INSERTION-SORT as an example. Below the number of statements executed in detail. Note that we can greatly simplify the analysis with techniques that we will learn in the next chapter!
- For the inner while loop, we will use  $t_j$  to represent the number of times the loop statement runs for the  $j$ th iteration of the outer for loop.

INSERTION-SORT(A)	cost	times
1. for i = 2 to n	$c_1$	$n$
2.     key = A[i]	$c_2$	$n - 1$
3.     // Insert A[i] into the sorted subarray A[1:i-1]	0	
4.     j = i - 1	$c_4$	$n - 1$
5.     while j > 0 and A[j] > key	$c_5$	$\sum_{j=2}^n t_j$
6.         A[j + 1] = A[j]	$c_6$	$\sum_{j=2}^n (t_j - 1)$
7.         j = j - 1	$c_7$	$\sum_{j=2}^n (t_j - 1)$
8.     A[j + 1] = key	$c_8$	$n - 1$

- Let  $T(n)$  be the running time of INSERTION-SORT(A) with input size  $n$ . Then the total run time is given by the following equation.

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

- **Best case:** If the input array A is a sorted array already, then  $t_j = 1$  for all  $j$ .

$$\begin{aligned}
T(n) &= c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)
\end{aligned}$$

The above can be expressed as  $an + b$  for constants  $a$  and  $b$ , where  $a = c_1 + c_2 + c_4 + c_5 + c_8$  and  $b = -(c_2 + c_4 + c_5 + c_8)$ .

The running time is thus a **linear** function of  $n$ . We can express that as  $\Theta(n)$  — which is another way of saying that it grows at the rate of  $n$  (more on this notation in the next chapter).

- **Worst case:** If the input array  $A$  is sorted in a reverse order, then  $t_j = j$  for all  $j$ . In the worst-case, the run time can be calculated as follows:

$$\begin{aligned}
T(n) &= c_1n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5 + c_6 + c_7}{2}\right)n^2 + (c_1 + c_2 + c_4 + \frac{c_5 - c_6 - c_7}{2} + c_8)n - (c_2 + c_4 + c_5 + c_8)
\end{aligned}$$

We can express it as  $an^2 + bn + c$ , where  $a = c_5/2 + c_6/2 + c_7/2$ ,  $b = c_1 + c_2 + c_4 + (c_5 - c_6 - c_7)/2 + c_8$ , and  $c = -(c_2 + c_4 + c_5 + c_8)$

The running time is thus a **quadratic** function of  $n$ . We can express that as  $\Theta(n^2)$  — which is another way of saying that it grows at the rate of  $n^2$  (more on this notation in the next chapter).

- **Example code:** Checkout [examples/module1/insertion-sort](#) for a coded up example to play with. Check to see if the runtime is quadratic: e.g.. if we double the input size, the runtime grows by four.
- **Exercise 2.2-1** Express  $n^3/100 - 100n^2 - 100n + 3$  in terms on  $\Theta$  notation.
- **Exercise 2.2-2 Selection Sort**
- **Exercise 2.2-4.** How can we modify almost any algorithm to have a good best-case running time?