Chapter 2: Getting Started

Insertion Sort: Example of an Algorithm

- 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 $\langle a_1, a_2, \ldots, a_n \rangle$.
  - Output: A permutation (reordering) $\langle a_1', a_2', \ldots, a_n' \rangle$ of the input sequence such that $a_1' \leq a_2' \leq \ldots \leq a_n'$, that is , monotonically increasing.
  - The number 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.
  - A key to understanding keys!
    * A **key** is any information that we can use to compare two elements. For example, it could simply be an integer, or be double, or be more complex, like a String, Color, or user-defned class.
    * An **element** may contain just the key and nothing else. Like an array of integers. More commonly, an element has a key and associated data (we call this satellite data). For example, a row in a spreadsheet may have a student id as key and associated information like class, address, email etc. An address object may have a name, house/apt number, a street address, a city name, and a zip code. Here name may be the key and the rest satellite data.
    * When programming, we often consider an object and an element to be the same thing (except for primitive elements such as ints)

- Algorithm 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 languages. 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] = 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 $A = \langle 5, 4, 3, 2, 1 \rangle$. 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. To use a loop invariant, we need to show three things.

  Initialization : It is true prior to the first iteration of the loop.

  Maintenance : If it is true before an iteration of the loop, it remains true before the next iteration.

  Termination : When the loop terminates, the invariant, usually along with the reason the loop terminated, gives us a useful property to prove that the algorithm is correct.

- Using the loop invariant to prove correctness:

  – 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.

  – **Initialization:** At the start of the main loop, $i = 2$. The subarray $A[1 : i - 1]$ contains just the single element $A[1]$. Since it is only one element, is is trivially sorted. Thus the loop invariant is holds prior to the first iteration of the loop.

2

- **Maintenance:** At the start of the ith iteration, the subarray $A[1 : i-1]$ is sorted. The algorithm then shifts $A[i-1], A[i-2]$, and so on by one position to the right until it finds the right position for $A[i]$ such that now $A[1 : i]$ is sorted. In the next iteration i is set to $i+1$ and now the invariant is true for $A[1 : i+1-1] = A[1 : i]$.

  - **Termination:** When the loop terminates, $i = n + 1$. Thus the invariant tells us that the array $A[1 : n + 1 - 1] = A[1 : n]$ is sorted. Hence, the algorithm is correct.

- Loop invariants take some time to get used to. See the end of this chapter for another example.

- 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]

Analyzing Algorithms

- 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 in 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. We can run the code to get an idea but that does not give us a general answer.

- A more general (and much easier once we have had some practice) technique for analyzing is by counting 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_i to represent the number of times the loop statement runs for the ith 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	$\sum_{i=2}^{n} t_i$
6. `A[j + 1] = A[j]`	c_6	$\sum_{i=2}^{n}(t_i - 1)$
7. `j = j - 1`	c_7	$\sum_{i=2}^{n}(t_i - 1)$
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.

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

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

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

  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 $t_i = i$ for all i. In the worst-case, the run time can be calculated as follows:

4

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

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).

– **Experimentation**: Checkout examples/insertion-sort for a coded up example to experiment with. Check to see what happens to the runtime if we double the input size, quadruple the input size, increase by ten times, etc. Does it match our analysis?

Also, try a faster sorting algorithm like merge-sort (we will learn about it in next Module) and see how its runtime compares to Insertion Sort as we increase the input size.

For an input size of 640,000 merge-sort runs over a thousan times faster than insertion sort! For an input size of 100,000,000 merge-sort runs over ten thousand times faster than insertion sort!!

– **Exercise 2.2-1**: Express $n^3/100 - 100n^2 - 100n + 3$ in terms on Θ notation.

– **Exercise 2.2-2**: **Selection Sort** Find the smallest element in A[1:n] and exchange it with A[1]. Then find the second smallest element in A[2:n] and exchange it with A[2]. Continue in this manner for the first n-1 elements of the array. Write pseudo-code for SELECTION-SORT. What loopinvariant does it maintain? Analyze its worst-case running time.

– **Exercise 2.2-4**: How can we modify almost any algorithm to have a good best-case running time?

Answers to selected practice problems

• **Exercise 2.2-1**: $\Theta(n^3)$

• **Exercise 2.2-2**: **Selection Sort**

```
SELECTION-SORT(A)
```

```
1. for i = 1 to  n - 1
2.      smallest = i
3.      for j = i + 1 to n
4.          if A[j] < A[smallest]
5.              smallest = j
6.      swap A[i] and A[smallest]
```

Try to run the SELECTION-SORT for the input $A =< 5, 2, 4, 6, 1, 3, 2, 6 >$.

Running time analysis:

- What is the loop invariant?

  At the start of each iteration of the outer for loop, the subarray $A[1 : i-1]$ consists of the $i - 1$ smallest elements in the array $A[1 : n]$ and this subarray is in sorted order. After the first $n - 1$ elements, the subarray $A[n - 1]$ contains the smallest $n - 1$ elements, sorted, and therefore element $A[n]$ must be the largest element.

- The runtime analysis is similar to the one for insertion sort. The inner loop runs for $n - i$ steps (since $i + 1$ to n contains $n - i$ elements). To get the total run time, we then add that using the following sum for the range of i values in the outer loop.

$$(n - 1)c_2 + \sum_{i=1}^{i=n-1} (n - i)c_1$$

  Here c_1 is a constant that represents the time taken in the inner loop, whereas c_2 is a constant the represents lines 2 and 6. The sum above simplifies to:

$$(1 + 2 + \ldots + n - 1)c_1 + c_2(n - 1)$$

  The sum of the arithmetic series $(1 + 2 + \ldots + n) = n/(n + 1)/2$. Here we replace n by $n - 1$ to get $(n - 1)n/2 = n^2/2 - n/2$. Plugging that into the expression above, we get:

$$c_1 n^2/2 + c_1 n/2 + c_2(n - 1)$$

  The dominant term is n^2. Thus the total running time is on the order of n^2 or quadratic in terms of the input size n.

Additional Loop Invariant example

Another <u>loop invariant</u> example

FIND-MIN(A,n)
// return index of minimum element in $A[1:n]$
1. min = 1
2. for i = 2 to n
3.   if $A[i]$ < min
4.   min = i
5. return min

Loop invariant: The variable <u>min</u> contains the index of the minimum element in $A[1: i-1]$ just before the ith iteration.

<u>Initialization</u>: At the start of the loop $i=2$, so the range for the invariant is $A[1: 2-1] = A[1:1]$. Since we assign min = 1 in line 1 and there is only one element in $A[1:1]$, it must be the minimum

<u>Maintenance</u>: At the start of the ith iteration, <u>min</u> contains the index of the minimum element in $A[1: i-1]$. In lines 3 & 4, we compare min to $A[i]$ and establish a new <u>min</u> if $A[i]$ is smaller. Thus at the start of the $(i+1)$st iteration, <u>min</u> contains the index of minimum element in $A[1: i+1-1] = A[1:i]$, so the invariant holds.

<u>Termination</u>: When the loop terminates, $i = n+1$. So <u>min</u> contains the index of the minimum element in $A[1: n+1-1] = A[1:n]$. So in line 5 we return the correct index.