CLRS for Dummies

Everett

1.22.2025

Contents

Preface

This is a book for dummies who want to read but cannot understand Introduction to Algorithms by CLRS.
Everything is taken directly from that book or paraphrased.
All the details that you are not going to remember even if you read the original book are omitted.
I find the pseudocode really hard to read and that arrays start with 1 instead of 0 really anti-intuitive. So I just write everything in a syntax I came up that I find most easy to read, and I index arrays start with 0.

Chapter 1

The Role of Algorithms

1.1 Definitions

Algorithms

An **algorithm** is a procedure used to solve a computational problem. An algorithm takes some value as **input** and produce some value as **output**.

Example: the **sorting problem**

input : A sequence of n numbers $< a_1, a_2 \ldots a_n >$

Output : The sequence ordered from lowest to highest $< a'_1, a'_2 \ldots a_n >$

Such sequence of inputs is called **instance** of the problem. An algorithm is **correct** and **solve** the problem if for every input instance it ends with the correct output.

Sorting is the only computational problem we can solve with algorithms, every problem we see is some sort of sorting.

Data Structures

A **data structure** is a way to store and organize data that facilitate access and modification.

Complexity

When we talk about the complexity of an algorithm. There are two types.

Time Complexity : How long the algorithm takes to produce the results.

Space Complexity : How much storage space an algorithm needs.

If not specified, complexity usually means time complexity. Complexities are defined by a polynomial function of n, the number of inputs.

Hard Problem

Hard problems are problems that we have no known efficient algorithms yet. They are also called NP-complete problems.

What we learn in this book are all efficient algorithms **efficient algorithms** which has polynomial time complexities.

Chapter 2

Getting Started

2.1 Insertion Sort

Illustration:

Pseudocode:

```
1 def InsertionSort(arr[]):
2         for j=1 to arr.length – 1:
3                 value = arr[j]
4                 i = j – 1
5                 while i >= 0 and arr[i] > value:
6                         arr[i + 1] = arr[i]
7                         i = i – 1
8                 arr[i + 1] = value
9         return arr
```

Explanation:

1. Declare function with one parameter the input array arr.
2. Iterate index j from 0 to the last index (one less than length of the array).
3. Put value of element $arr[j]$ in the variable $value$
4. For each iteration, get the previous index $i = j - 1$.
5. If $i \geq 0$ which means $arr[i]$ is not out of our array, and $arr[i]$ is bigger than $value$
6. Move $arr[i]$ to the right by setting $arr[i + 1] = i$
7. Once we iterated through all the element before $value$ or we reached a value smaller than $value$, put $value$ at $arr[i + 1]$ where it should be in the sorted sub array.
8. Return the array arr that is now sorted

2.2 Loop Invariant

How do we know our algorithm is correct? One way to check is to use the loop Invariant. There are three part of loop invariant.

Initialization: Loop invariant is true before the loop begins.
Maintenance: It will remain true after every single iteration.
Termination: The loop invariant is still true, but it also has to show that the algorithm produces correct output.

Loop Invariant of Insertion Sort

Let's look at the loop invariant of insertion sort as an example.

2.3 Analyzing Algorithms

2.4 Designing Algorithms with Divide and Conquer

Chapter 3

Growth of Functions

Chapter 4

Divide and Conquer

Chapter 5

Probabilistic Analysis and Randomized Algorithms