The Author
Mike Chyson (Li Mingming)

The Big Book of

Algorithms

Theory and practice of algorithms

First created: Dec, 10, 2020
Last modified: Tuesday 15th December, 2020

Dedication

Contents

Chapter 1

Theory

1.1 What is algorithm?

$$input \longrightarrow algorithm \longrightarrow output$$

An algorithm is a sequence of computational that transform the input into the output, it describe a specific computational procedure for achieving the input/output relationship.

1.2 Instance of a problem

An instance of a problem is the input needed to compute a solution to the problem.

1.3 Correct algorithm

For every input instance, a correct algorithm halts out the corrent output.

1.4 Data structure

Data structure is a way to store and organize data in order to faciliate access and modifications. No single data structure works well for all purpose.

1.5 The core technique

Learning an algorithm is to learn the technique of algorithm design and analysis.

1.6 Algorithm efficiency

Computers are not infinitely fast and memory is not free, thus the efficiency of a algorithm matters.

1.7 Algorithms use information

There is information in the problem, the general is, the more information you use, the more efficiency the algorithm is. Data structure is a way to use this information.

1.8 Loop invariant

There can be infinite input cases, so it is hard to say an algorithm is corrent. Loop invariant is a way to guarantee that the algorithm is correct.

There are 3 elements in a loop invariant:

initialization It is true prior to the first iteration of the loop.

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

termination When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

1.9 Analyzing an algorithm

Analyzing an algorithm is to predict the resources consumed. The most important resources is time and space.

In general, the time grows with the size of the input, so it is traditional to describe the running time as the function of the size of its input.

1.9.1 Worst-case analysis

Becuase the behavior of an algorithm may be different for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas. The reason to analyze worst-case running time is as follows:

1. It give an upper bound on the running time.
2. Worst case ocurrs fairly often.
3. The "average case" is often roughly as bad as the worst case.

1.10 Resource model

Before we can analyze an algorithm, we must have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs. However, the focus is algorithm, not the tedious hardware detail. We shall assume a generic one-processor, random-access machine (RAM) model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, with no concurrent operations.

1.11 Growth of functions

Althoght we can sometimes determine the exact running time of an algorithm, the extra procision is not usually worth the effort of computing it. When we look at input sizes large enought to make only the order of growth of the running time relevant, we are studying the **asymptotic efficiency** of algorithms.

1.11.1 Θ-notation

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

Because $\Theta(g(n))$ is a set, we could write "$f(n) \in \Theta(g(n))$" to indicate that $f(n)$ is a memeber of $\Theta(g(n))$. However, we will usually write "$f(n) = \Theta(g(n))$"

to express the same notion.

Θ-notation indicates the function is bounded with a asymptotically tight upper bound and a asymptotically tight lower bound.

1.11.2 O-notation

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

O-notation indicates the function is bounded with a asymptotically tight upper bound.

1.11.3 Ω-notation

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

Ω-notation indicates the function is bounded with a asymptotically tight lower bound.

1.11.4 Theorem

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

1.11.5 o-notation

$$O(g(n)) = \{f(n) : \text{for any positive constant } c \text{ there exist a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$$

or

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

o-notation indicates the function is bounded with a not asymptotically tight upper bound.

1.11.6 ω-notation

$$O(g(n)) = \{f(n) : \text{for any positive constant } c \text{ there exist a constant } n_0 > 0 \text{ such that } 0 \leq\leq cg(n) < f(n) \text{ for all } n \geq n_0\}$$

or

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

ω-nodation indicates the function is bounded with a not asymptotically tight lower bound.

Part I

Data Structure

Chapter 2

Array

2.1 What is an array?

An Array is a collection of items. The items are stored in neighboring (contiguous) memory locations.

For example:

2	1	3	2	1

2.2 Capacity and length

The capacity is the items it can hold at most. It is specified when you create an array.

Length is the number of items it holds currently.

2.3 Operations

Array is a data structure, which means that it stores data in a specific format and supports certain operations on the data it stores.

There are three basic operations in array:

- insert
- delete
- search