

# Visual Analysis of Algorithms

Version 0.0.4

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Algorithm Analysis</b>	<b>3</b>
2.1	Common Complexity Classes . . . . .	3
2.2	Binary Logarithm . . . . .	3
2.3	Amortized Analysis . . . . .	4
<b>3</b>	<b>Data Structures</b>	<b>5</b>
3.1	Heaps . . . . .	5
3.1.1	Applications . . . . .	5
<b>4</b>	<b>Problem Solving Methods</b>	<b>6</b>
4.1	Greedy Algorithms . . . . .	6
4.1.1	Overview . . . . .	6
4.1.2	Dijkstra's Shortest Path Algorithm . . . . .	6
4.2	Recursion . . . . .	6
4.2.1	Fibonacci Sequence . . . . .	7
4.2.2	Binary Exponentiation . . . . .	7
4.3	Dynamic Programming . . . . .	7
<b>5</b>	<b>Appendix</b>	<b>8</b>
5.1	Resources . . . . .	8

# Chapter 1

# Introduction

Algorithms and data structures notes.

# Chapter 2

## Algorithm Analysis

### 2.1 Common Complexity Classes

Name	Notation
Constant	$O(n)$
Logarithmic	$O(\log n)$
Linear	$O(n)$
Linearithmic	$O(n \log n)$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(2^n)$
Factorial	$O(n!)$

### 2.2 Binary Logarithm

Logarithm is the inverse function to exponentiation which means that the logarithm of a number  $n$  to the base  $b$  is the power to which  $b$  must be raised to produce  $n$ . The binary logarithm is the power to which the number 2 must be raised to obtain the value  $n$ .

$$x = \log_2 n \iff 2^x = n$$

- The number of bits in the binary representation of a positive integer  $n$  is the integral part of  $\log_2(n) + 1$ .

- The binary logarithm also frequently appears in the analysis of algorithms because binary logarithms occur in the analysis of algorithms based on two-way branching. If a problem initially has  $n$  choices for its solution, and each iteration of the algorithm reduces the number of choices by a factor of two, then the number of iterations needed to select a single choice is again the integral part of  $\log_2(n)$ .

— [https://en.wikipedia.org/wiki/Binary\\_logarithm](https://en.wikipedia.org/wiki/Binary_logarithm)

## 2.3 Amortized Analysis

The motivation for amortized analysis is that looking at the worst-case run time can be too pessimistic. Instead, amortized analysis averages the running times of operations in a sequence over that sequence. Amortized analysis is a useful tool that complements other techniques such as worst-case and average-case analysis.

Imagine a dynamic list backed by a fixed size array that doubles once capacity is reached and require  $n$  inserts into the new fixed size array.

n	# of inserts
1	1
2	1
3	1
...	1
n	n
...	n + 1

$$\frac{\# \text{ of inserts}}{n} \rightarrow \frac{n+n+1}{n} \rightarrow \frac{2n+1}{n} \rightarrow \frac{2n}{n} \rightarrow 2$$

— [https://en.wikipedia.org/wiki/Amortized\\_analysis](https://en.wikipedia.org/wiki/Amortized_analysis)

# Chapter 3

## Data Structures

### 3.1 Heaps

#### 3.1.1 Applications

- Heapsort
- Priority queue
- K-way merge

# Chapter 4

## Problem Solving Methods

### 4.1 Greedy Algorithms

#### 4.1.1 Overview

A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage. In many problems, a greedy strategy does not produce an optimal solution, but a greedy heuristic can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

— [https://en.wikipedia.org/wiki/Greedy\\_algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm)

#### 4.1.2 Dijkstra's Shortest Path Algorithm

Dijkstra's shortest path algorithm is a search algorithm that finds the shortest path between a vertex and other vertices in a weighted graph.

### 4.2 Recursion

The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions.

— Niklaus Wirth, Algorithms + Data Structures = Programs, 1976<sup>1</sup>

### 4.2.1 Fibonacci Sequence

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 1$$

```
def fib(n: int) -> int:
    if n <= 1:
        return n

    return fib(n-1) + fib(n-2)
```

### 4.2.2 Binary Exponentiation

$$x^n = \begin{cases} x(x^2)^{(n-1)/2}, & \text{if } n \text{ is odd} \\ (x^2)^{n/2}, & \text{if } n \text{ is even} \end{cases}$$

## 4.3 Dynamic Programming

To apply DP, must have the following:

1. Optimal substructure - Problem can be broken into sub-problems and solved recursively.
2. Overlapping sub-problems: Recursive algorithm solves the same sub-problems over and over.

If okay, then we can approach in 2 ways.

1. Top-down - Memoize with cache table.
2. Bottom-up - Solve sub-problems first and use their solutions to solve bigger sub-problems.

---

<sup>1</sup><https://archive.org/details/algorithmsdatast00wirth/page/126>



# Chapter 5

# Appendix

## 5.1 Resources

- LeetCode
- Project Euler
- The Algorithm Design Manual
- Elements of Programming Interviews