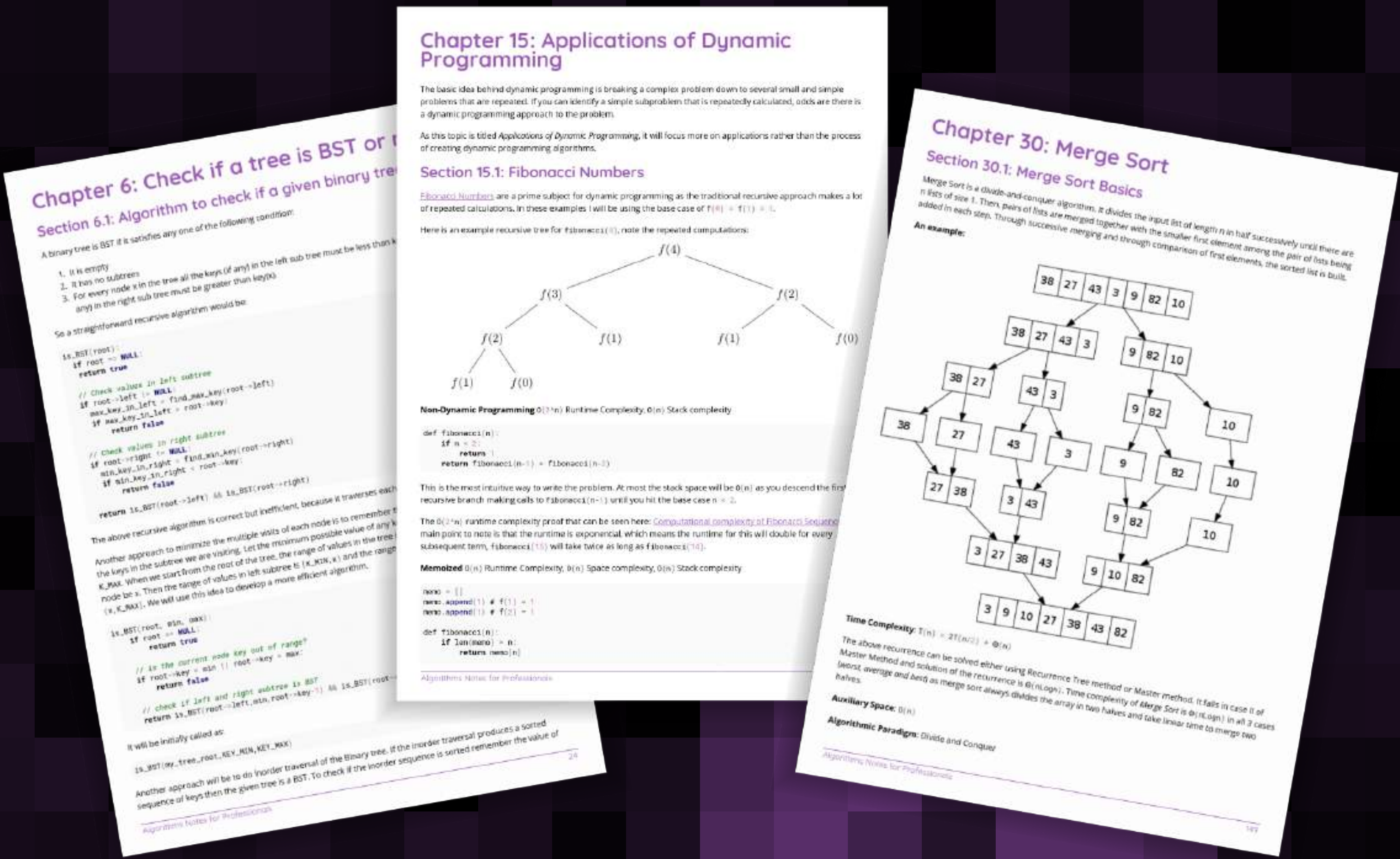
Algorithms

Notes for Professionals



200+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/AlgorithmsBook

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with algorithms

Section 1.1: A sample algorithmic problem

An algorithmic problem is specified by describing the complete set of *instances* it must work on and of its output after running on one of these instances. This distinction, between a problem and an instance of a problem, is fundamental. The algorithmic *problem* known as *sorting* is defined as follows: [Skiena:2008:ADM:1410219]

- Problem: Sorting
- Input: A sequence of *n* keys, `a_1, a_2, ..., a_n`.
- Output: The reordering of the input sequence such that `a'_1 <= a'_2 <= ... <= a'_{n-1} <= a'_n`

An *instance* of sorting might be an array of strings, such as `{ Haskell, Emacs }` or a sequence of numbers such as `{ 154, 245, 1337 }`.

Section 1.2: Getting Started with Simple Fizz Buzz Algorithm in Swift

For those of you that are new to programming in Swift and those of you coming from different programming bases, such as Python or Java, this article should be quite helpful. In this post, we will discuss a simple solution for implementing swift algorithms.

Fizz Buzz

You may have seen Fizz Buzz written as Fizz Buzz, FizzBuzz, or Fizz-Buzz; they're all referring to the same thing. That "thing" is the main topic of discussion today. First, what is FizzBuzz?

This is a common question that comes up in job interviews.

Imagine a series of a number from 1 to 10.

```
1 2 3 4 5 6 7 8 9 10
```

Fizz and Buzz refer to any number that's a multiple of 3 and 5 respectively. In other words, if a number is divisible by 3, it is substituted with fizz; if a number is divisible by 5, it is substituted with buzz. If a number is simultaneously a multiple of 3 AND 5, the number is replaced with "fizz buzz." In essence, it emulates the famous children game "fizz buzz".

To work on this problem, open up Xcode to create a new playground and initialize an array like below:

```
// for example
let number  = [1,2,3,4,5]
// here 3 is fizz and 5 is buzz
```

To find all the fizz and buzz, we must iterate through the array and check which numbers are fizz and which are buzz. To do this, create a for loop to iterate through the array we have initialised:

```
for num in number {
  // Body and calculation goes here
}
```

After this, we can simply use the "if else" condition and module operator in swift ie - % to locate the fizz and buzz

```
for num in number {
  if num % 3 == 0 {
    print("\(num) fizz")
  } else {
    print(num)
  }
}
```

Great! You can go to the debug console in Xcode playground to see the output. You will find that the "fizzes" have been sorted out in your array.

For the Buzz part, we will use the same technique. Let's give it a try before scrolling through the article — you can check your results against this article once you've finished doing this.

```
for num in number {
  if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
    print(num)
  }
}
```

Check the output!

It's rather straight forward — you divided the number by 3, fizz and divided the number by 5, buzz. Now, increase the numbers in the array

```
let number = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

We increased the range of numbers from 1-10 to 1-15 in order to demonstrate the concept of a "fizz buzz." Since 15 is a multiple of both 3 and 5, the number should be replaced with "fizz buzz." Try for yourself and check the answer!

Here is the solution:

```
for num in number {
  if num % 3 == 0 && num % 5 == 0 {
    print("\(num) fizz buzz")
  } else if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
    print(num)
  }
}
```

Wait...it's not over though! The whole purpose of the algorithm is to customize the runtime correctly. Imagine if the range increases from 1-15 to 1-100. The compiler will check each number to determine whether it is divisible by 3 or 5. It would then run through the numbers again to check if the numbers are divisible by 3 and 5. The code would essentially have to run through each number in the array twice — it would have to runs the numbers by 3 first and then run it by 5. To speed up the process, we can simply tell our code to divide the numbers by 15 directly.

Here is the final code:

```
for num in number {
```

```
  if num % 15 == 0 {
    print("\(num) fizz buzz")
  } else if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
    print(num)
  }
}
```

As Simple as that, you can use any language of your choice and get started

Enjoy Coding

Chapter 2: Algorithm Complexity

Section 2.1: Big-Theta notation

Unlike Big-O notation, which represents only upper bound of the running time for some algorithm, Big-Theta is a tight bound; both upper and lower bound. Tight bound is more precise, but also more difficult to compute.

The Big-Theta notation is symmetric: `f(x) = Ө(g(x)) <=> g(x) = Ө(f(x))`

An intuitive way to grasp it is that `f(x) = Ө(g(x))` means that the graphs of f(x) and g(x) grow in the same rate, or that the graphs 'behave' similarly for big enough values of x.

The full mathematical expression of the Big-Theta notation is as follows:
Ө(f(x)) = {g: N0 -> R and c1, c2, n0 > 0, where c1 < abs(g(n) / f(n)), for every n > n0 and abs is the absolute value }

An example

If the algorithm for the input `n` takes `42n^2 + 25n + 4` operations to finish, we say that is `O(n^2)`, but is also `O(n^3)` and `O(n^100)`. However, it is `Ө(n^2)` and it is not `Ө(n^3)`, `Ө(n^4)` etc. Algorithm that is `Ө(f(n))` is also `O(f(n))`, but not vice versa!

Formal mathematical definition

`Ө(g(x))` is a set of functions.

```
Ө(g(x)) = {f(x) such that there exist positive constants c1, c2, N such that 0 <= c1*g(x) <= f(x)
<= c2*g(x) for all x > N}
```

Because `Ө(g(x))` is a set, we could write `f(x) ∈ Ө(g(x))` to indicate that `f(x)` is a member of `Ө(g(x))`. Instead, we will usually write `f(x) = Ө(g(x))` to express the same notion - that's the common way.

Whenever `Ө(g(x))` appears in a formula, we interpret it as standing for some anonymous function that we do not care to name. For example the equation `T(n) = T(n/2) + Ө(n)`, means `T(n) = T(n/2) + f(n)` where `f(n)` is a function in the set `Ө(n)`.

Let `f` and `g` be two functions defined on some subset of the real numbers. We write `f(x) = Ө(g(x))` as `x->infinity` if and only if there are positive constants `K` and `L` and a real number `x0` such that holds:

`K|g(x)| <= f(x) <= L|g(x)|` for all `x >= x0`.

The definition is equal to:

`f(x) = O(g(x)) and f(x) = Ω(g(x))`

A method that uses limits

if `limit(x->infinity) f(x)/g(x) = c ∈ (0,∞)` i.e. the limit exists and it's positive, then `f(x) = Ө(g(x))`

Common Complexity Classes

Name	Notation	n = 10	n = 100
Constant	Ө(1)	1	1
Logarithmic	Ө(log(n))	3	7
Linear	Ө(n)	10	100