Algorithms/Introduction

This book covers techniques for the design and analysis of algorithms. The algorithmic techniques covered include: divide and conquer, backtracking, dynamic programming, greedy algorithms, and hill-climbing.

Any solvable problem generally has at least one algorithm of each of the following types:

1. the obvious way;
2. the methodical way;
3. the clever way; and
4. the miraculous way.

On the first and most basic level, the "obvious" solution might try to exhaustively search for the answer. Intuitively, the obvious solution is the one that comes easily if you're familiar with a programming language and the basic problem solving techniques.

The second level is the methodical level and is the heart of this book: after understanding the material presented here you should be able to methodically turn most obvious algorithms into better performing algorithms.

The third level, the clever level, requires more understanding of the elements involved in the problem and their properties or even a reformulation of the algorithm (e.g., numerical algorithms exploit mathematical properties that are not obvious). A clever algorithm may be hard to understand by being non-obvious that it is correct, or it may be hard to understand that it actually runs faster than what it would seem to require.

The fourth and final level of an algorithmic solution is the miraculous level: this is reserved for the rare cases where a breakthrough results in a highly non-intuitive solution.

Naturally, all of these four levels are relative, and some clever algorithms are covered in this book as well, in addition to the methodical techniques. Let's begin.

Prerequisites

To understand the material presented in this book you need to know a programming language well enough to translate the pseudocode in this book into a working solution. You also need to know the basics about the following data structures: arrays, stacks, queues, linked-lists, trees, heaps (also called priority queues), disjoint sets, and graphs.

Additionally, you should know some basic algorithms like binary search, a sorting algorithm (merge sort, heap sort, insertion sort, or others), and breadth-first or depth-first search.

If you are unfamiliar with any of these prerequisites you should review the material in the *Data Structures* book first.

When is Efficiency Important?

Not every problem requires the most efficient solution available. For our purposes, the term efficient is concerned with the time and/or space needed to perform the task. When either time or space is abundant and cheap, it may not be worth it to pay a programmer to spend a day or so working to make a program faster.

However, here are some cases where efficiency matters:

- When resources are limited, a change in algorithms could create great savings and allow limited machines (like cell phones, embedded systems, and sensor networks) to be stretched to the frontier of possibility.
- When the data is large a more efficient solution can mean the difference between a task finishing in two days versus two weeks. Examples include physics, genetics, web searches, massive online stores, and network traffic analysis.
- Real time applications: the term "real time applications" actually refers to computations that give time guarantees, versus meaning "fast." However, the quality can be increased further by choosing the appropriate algorithm.
- Computationally expensive jobs, like fluid dynamics, partial differential equations, VLSI design, and cryptanalysis can sometimes only be considered when the solution is found efficiently enough.
- When a subroutine is common and frequently used, time spent on a more efficient implementation can result in benefits for every application that uses the subroutine. Examples include sorting, searching, pseudorandom number generation, kernel operations (not to be confused with the operating system kernel), database queries, and graphics.

In short, it's important to save time when you do not have any time to spare.

When is efficiency unimportant? Examples of these cases include prototypes that are used only a few times, cases where the input is small, when simplicity and ease of maintenance is more important, when the area concerned is not the bottle neck, or when there's another process or area in the code that would benefit far more from efficient design and attention to the algorithm(s).

Inventing an Algorithm

Because we assume you have some knowledge of a programming language, let's start with how we translate an idea into an algorithm. Suppose you want to write a function that will take a string as input and output the string in lowercase:

```
// tolower -- translates all alphabetic, uppercase characters in str to lowercase
function tolower(string str): string
```

What first comes to your mind when you think about solving this problem? Perhaps these two considerations crossed your mind:

1. Every character in *str* needs to be looked at
2. A routine for converting a single character to lower case is required

The first point is "obvious" because a character that needs to be converted might appear anywhere in the string. The second point follows from the first because, once we consider each character, we need to do something with it. There are many ways of writing the **tolower** function for characters:

```
function tolower(character c): character
```

There are several ways to implement this function, including:

- look c up in a table—a character indexed array of characters that holds the lowercase version of each character.
- check if c is in the range 'A' ≤ c ≤ 'Z', and then add a numerical offset to it.

These techniques depend upon the character encoding. (As an issue of separation of concerns, perhaps the table solution is stronger because it's clearer you only need to change one part of the code.)

However such a subroutine is implemented, once we have it, the implementation of our original problem comes immediately:

```
// tolower -- translates all alphabetic, uppercase characters in str to lowercase
function tolower(string str): string
  let result := ""
  for-each c in str:
    result.append(tolower(c))
  repeat
  return result
end
```

This code sample is also available in Ada.

The loop is the result of our ability to translate "every character needs to be looked at" into our native programming language. It became obvious that the **tolower** subroutine call should be in the loop's body. The final step required to bring the high-level task into an implementation was deciding how to build the resulting string. Here, we chose to start with the empty string and append characters to the end of it.

Now suppose you want to write a function for comparing two strings that tests if they are equal, ignoring case:

```
// equal-ignore-case -- returns true if s and t are equal, ignoring case
function equal-ignore-case(string s, string t): boolean
```

These ideas might come to mind:

1. Every character in strings s and t will have to be looked at
2. A single loop iterating through both might accomplish this
3. But such a loop should be careful that the strings are of equal length first
4. If the strings aren't the same length, then they cannot be equal because the consideration of ignoring case doesn't affect how long the string is
5. A tolower subroutine for characters can be used again, and only the lowercase versions will be compared

These ideas come from familiarity both with strings and with the looping and conditional constructs in your language. The function you thought of may have looked something like this:

```
// equal-ignore-case -- returns true if s or t are equal, ignoring case
function equal-ignore-case(string s[1..n], string t[1..m]): boolean
  if n != m:
    return false               \if they aren't the same length, they aren't equal\
  fi

  for i := 1 to n:
    if tolower(s[i]) != tolower(t[i]):
      return false
    fi
  repeat
  return true
end
```

This code sample is also available in Ada.

Or, if you thought of the problem in terms of functional decomposition instead of iterations, you might have thought of a function more like this:

```
// equal-ignore-case -- returns true if s or t are equal, ignoring case
function equal-ignore-case(string s, string t): boolean
  return tolower(s).equals(tolower(t))
end
```

Alternatively, you may feel neither of these solutions is efficient enough, and you would prefer an algorithm that only ever made one pass of s or t. The above two implementations each require two-passes: the first version computes the lengths and then compares each character, while the second version computes the lowercase versions of the string and then compares the results to each other. (Note that for a pair of strings, it is also possible to have the length precomputed to avoid the second pass, but that can have its own drawbacks at times.) You could imagine how similar routines can be written to test string equality that not only ignore case, but also ignore accents.

Already you might be getting the spirit of the pseudocode in this book. The pseudocode language is not meant to be a real programming language: it abstracts away details that you would have to contend with in any language. For example, the language doesn't assume generic types or dynamic versus static types: the idea is that it should be clear what is intended and it should not be too hard to convert it to your native language. (However, in doing so, you might have to make some design decisions that limit the implementation to one particular type or form of data.)

There was nothing special about the techniques we used so far to solve these simple string problems: such techniques are perhaps already in your toolbox, and you may have found better or more elegant ways of expressing the solutions in your programming language of choice. In this book, we explore general algorithmic techniques to expand your toolbox even further. Taking a naive algorithm and making it more efficient might not come so immediately, but after understanding the material in this book you should be able to methodically apply different solutions, and, most importantly, you will be able to ask yourself more questions about your programs. Asking questions can be just as important as answering questions, because asking the right question can help you reformulate the problem and think outside of the box.

Understanding an Algorithm

Computer programmers need an excellent ability to reason with multiple-layered abstractions. For example, consider the following code:

```
function foo(integer a):
  if (a / 2) * 2 == a:
     print "The value " a " is even."
  fi
end
```

To understand this example, you need to know that integer division uses truncation and therefore when the if-condition is true then the least-significant bit in a is zero (which means that a must be even). Additionally, the code uses a string printing API and is itself the definition of a function to be used by different modules. Depending on the programming task, you may think on the layer of hardware, on down to the level of processor branch-prediction or the cache.

Often an understanding of binary is crucial, but many modern languages have abstractions far enough away "from the hardware" that these lower-levels are not necessary. Somewhere the abstraction stops: most programmers don't need to think about logic gates, nor is the physics of electronics necessary. Nevertheless, an essential part of programming is multiple-layer thinking.

But stepping away from computer programs toward algorithms requires another layer: mathematics. A program may exploit properties of binary representations. An algorithm can exploit properties of set theory or other mathematical constructs. Just as binary itself is not explicit in a program, the mathematical properties used in an algorithm are not explicit.

Typically, when an algorithm is introduced, a discussion (separate from the code) is needed to explain the mathematics used by the algorithm. For example, to really understand a greedy algorithm (such as Dijkstra's algorithm) you should understand the mathematical properties that show how the greedy strategy is valid for all cases. In a way, you can think of the mathematics as its own kind of subroutine that the algorithm invokes. But this "subroutine" is not present in the code because there's nothing to call. As you read this book try to think about mathematics as an implicit subroutine.

Overview of the Techniques

The techniques this book covers are highlighted in the following overview.

- **Divide and Conquer**: Many problems, particularly when the input is given in an array, can be solved by cutting the problem into smaller pieces (*divide*), solving the smaller parts recursively (*conquer*), and then combining the solutions into a single result. Examples include the merge sort and quicksort algorithms.
- **Randomization**: Increasingly, randomization techniques are important for many applications. This chapter presents some classical algorithms that make use of random numbers.
- **Backtracking**: Almost any problem can be cast in some form as a backtracking algorithm. In backtracking, you consider all possible choices to solve a problem and recursively solve subproblems under the assumption that the choice is taken. The set of recursive calls generates a tree in which each set of choices in the tree is considered consecutively. Consequently, if a solution exists, it will eventually be found.

Backtracking is generally an inefficient, brute-force technique, but there are optimizations that can be performed to reduce both the depth of the tree and the number of branches. The technique is called backtracking because after one leaf of the tree is visited, the algorithm will go back up the call stack (undoing choices that didn't lead to success), and then proceed down some other branch. To be solved with backtracking techniques, a problem needs to have some form of "self-similarity," that is, smaller instances of the problem (after a choice has been made) must resemble the original problem. Usually, problems can be generalized to become self-similar.

- **Dynamic Programming**: Dynamic programming is an optimization technique for backtracking algorithms. When subproblems need to be solved repeatedly (i.e., when there are many duplicate branches in the backtracking algorithm) time can be saved by solving all of the subproblems first (bottom-up, from smallest to largest) and storing the solution to each subproblem in a table. Thus, each subproblem is only visited and solved once instead of repeatedly. The "programming" in this technique's name comes from programming in the sense of writing things down in a table; for example, television programming is making a table of what shows will be broadcast when.
- **Greedy Algorithms**: A greedy algorithm can be useful when enough information is known about possible choices that "the best" choice can be determined without considering all possible choices. Typically, greedy algorithms are not challenging to write, but they are difficult to prove correct.
- **Hill Climbing**: The final technique we explore is hill climbing. The basic idea is to start with a poor solution to a problem, and then repeatedly apply optimizations to that solution until it becomes optimal or meets some other requirement. An important case of hill climbing is network flow. Despite the name, network flow is useful for many problems that describe relationships, so it's not just for computer networks. Many matching problems can be solved using network flow.

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, A

Algorithm and code example

Level 1 (easiest)

1. *Find maximum* With algorithm and several different programming languages
2. *Find minimum* With algorithm and several different programming languages
3. *Find average* With algorithm and several different programming languages
4. *Find mode* With algorithm and several different programming languages
5. *Find total* With algorithm and several different programming languages
6. *Counting* With algorithm and several different programming languages

Level 2

1. *Talking to computer Lv 1* With algorithm and several different programming languages
2. *Sorting-bubble sort* With algorithm and several different programming languages

Level 3

1. *Talking to computer Lv 2* With algorithm and several different programming languages

Level 4

1. *Talking to computer Lv 3* With algorithm and several different programming languages
2. *Find approximate maximum* With algorithm and several different programming languages

Level 5

1. *Quicksort*
