Into Algorithms

Ola Dahl

2024-05-05

Table of contents

Preface

This is a book about algorithms. By algorithms, I mean algorithms in the computer science sense, such as algorithms for traversing a graph, computing prime factors of a number, or finding the best path through a maze.

The book reflects my experience from learning about algorithms. I have practiced, mostly using Kattis, but also participated in on-line courses and purchased, and partly read, on-line articles and books.

I did my first Kattis problem in September 2019. Starting from a hello world style problem, I continued, with problems having difficulty level set to easy.

I was confident that I would succeed. I regarded myself as a programmer, with several years of professional and private experience of software development.

After having tried a few problems with difficulty level set to easy, I moved on to a problem with medium difficulty.

It was a problem about a graph, and about figuring out the shortest path to a goal from different places in the graph, and then deciding to take, for a given node, a path that was not a shortest path.

I made several attempts, and all of them failed.

I had heard of Dijkstra's algorithm, but never implemented it myself. I had also heard of Dynamic Programming, and even practised it during my Master thesis work. I had no clue, however, if these two topics had anything in common.

After many tries, and solving many easier problem in between, I solved the problem. This was in August 2020.

I realized that my confidence about programming was not on par with my skills for solving Kattis problems. I had a long way to go, in order to solve more problems on medium level, and I didn't even dare to try any problems with difficulty level set to hard.

I wanted to learn more. I signed up for, and completed, an excellent course series on Data Structures and Algorithms. It was not for free, but it was a very rewarding investment. The teachers were excellent, the lectures were pedagogical, and the labs were challenging.

I continued with Kattis, and managed to solve also some problems with somewhat higher difficulty level.

I had an ongoing project where I wanted to make books with layers. The idea was, and still is, to have a book with several layers, with common content, and with content specific for each layer.

For a book about programming, it makes sense, I think, to have layers that correspond to programming languages.

Out of this came the decision to write down some of my experiences of working with algorithms in the context of solving Kattis problems, as a layered book.

It is a work in progress, and I will continue to update it, here on GitHub, as time goes on.

It is written using Quarto.

Chapter 1

Introduction

There are many books on algorithms. Why would I try to write yet another one?

One reason is selfishness. When you write, you learn.

Another is that there might be others, in situations similar to mine (software developers that need to sharpen their algorithm skills), that might have some use of what I write.

Chapter 2

Sources of inspiration

I have already mentioned Kattis, which is where I started by journey into algorithms. In the beginning of that journey I discovered Sannemo (2020), which I think contains very good material about competetive programming. It also contains recommended Kattis problems to solve.

Another book about competitive programming is Laaksonen (2020). It contains good material, and it gives a recommendation to use the CSES problem set for practice, while reading the book.

A pair of books, with *The Lower Bound of Programming Contests in the 2020s* as their subtitle, and a web page with information about the books, is Halim and Halim (2018) and Halim and Halim (2020).

Another resource that I would like to mention is Algorithms for Competitive Programming, with descriptions of many algorithms, and code examples.

2.1 Outline

We start of with graphs, in Chapter 4.

Chapter 3

Sets

3.1 Subsets

How can we generate all subsets of a set?

Suppose the set is encoded as a list of numbers.

We can treat the indices of these numbers as another list

$$1, 2, \ldots, n$$

We can generate all subsets of the indices list, using Algorithm 3.1

Algorithm 3.1 Computing all subsets

```
1: procedure SUBSETS(v, k, n)
2:     if k = n + 1 then
3:         PROCESS(v)
4:     end if
5:     v.push_back(k)
6:     SUBSETS(v, k + 1, n)
7:     v.push_back(k)
8:     SUBSETS(v, k + 1, n)
9: end procedure
```

Chapter 4

Graphs

We describe graphs, and some algorithms for graphs.

4.1 Notations and Terminology

A graph is defined by a set of vertices, also referred to as nodes, and a set of edges.

We use the notation V for the set of vertices and the notation E for the set of edges.

A graph g is defined as

$$g = (V, E)$$

An edge is e is defined by a pair of vertices, as

$$e = (v_1, v_2)$$

A directed graph is a graph where the vertices in each edge e are interpreted as a ordered pair (v_1, v_2), so that the edge starts at v_1 and ends at v_2.

An undirected graph is where the vertices in each edge e do not have any defined order. Their role is to indicate to which vertices the edge is connected, without any connotations of direction.

A weighted graph is a graph where each edge e is associated with a numeric value w, referred to as the weight of the edge.

4.2 Graph traversal

We can traverse a graph by starting from a node v_s, and then find a way to visit all nodes that can be reached from v_s.

If the graph is connected, in the sense that all its nodes can be reached from v_s, then the graph traversal will visit all the nodes.

If the graph is not connected, we can, after the traversal is done, pick a node that is not yet visited. We can start a new traversal from that node. When this is done we can check if all nodes are visited. If this is not the case, we can repeat the process, by picking a new node that is not yet visited and start a new traversal from that node.

In the end, we will have visited all nodes.

When doing a graph traversal, we must, at a given node v, determine which node to visit next. Naturally, we must select one of the neighbors of v.

4.2.1 DFS

If we select a neigbor of v, and then select one of the neighbors of the selected neigbor, and proceed like this until we come to a node where where there either are no neigbors to select, or we have already visited all neigbors, we get what is known as depth first search, abbreviated as DFS.

While proceeding, we keep track of nodes where we have selected neighbors, and when we are done with a node, we return to the node from which the node we are done with was selectd. We kan keep track of this using a stack.

If we implement the DFS as a recursive function, the call stack for this function will serve as such a stack.

Pseudocode for a DFS algorithm is shown in Algorithm 4.1 .

Algorithm 4.1 DFS

Require: a graph and a vector *visited*, with all elements initialized to false

```
1: procedure DFS(v, visited)
2:     visited[v] = true
3:     for v_nb in neighbors[v] do
4:         if not visited[v_nb] then
5:             DFS(v_nb, visited)
6:         end if
7:     end for
8: end procedure
```

The algorithm in Algorithm 4.1 performs a DFS traversal from a node v. If we want to traverse all nodes of a graph, we can do this by using the algorithm repeatedly, where each run is started from a node that is not yet visited.

Here is an implementation of DFS in Python.

4.2.2 BFS

If we select and visit all a neigbors of v, and then proceed in the same way for the neighbors, i.e. we visit, for each neigbour, all its neigbours before we proceed, we get what is known as breadth first search, abbreviated as BFS.

While proceeding, we keep track of nodes we have visited, so that we can visit each of these nodes' neigbours when time comes.

We can use a queue to keep track of nodes that we have visited.

Pseudocode for a BFS algorithm is shown in Algorithm 4.2 .

Algorithm 4.2 BFS

```
1:  procedure BFS(v)
2:      visited[v] = true
3:      q.push(v)
4:      while q not empty do
5:          v_cur = q.pop()
6:          for v_nb in neighbors[v_cur] do
7:              if not visited[v_nb] then
8:                  visited[v_nb] = true
9:                  q.push(v_nb)
10:             end if
11:         end for
12:     end while
13: end procedure
```

The algorithm in Algorithm 4.2 performs a breadth-first traversal, starting from a node v. If we want to traverse all nodes of a graph, we do this by using the algorithm repeatedly, where each run is started from a node that is not yet visited.

Starting from a node v, the DFS algorithm visits all nodes that have distance one to v. For each of these nodes, it then visits all nodes that are at a distance two from v.

Suppose we have visited a node v_s, at a distance d from the starting node v. Consider the path th

4.3 Connectedness

4.3.1 Connected components

4.3.2 Strongly connected components

4.3.3 Topological sort

https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/

https://usaco.guide/gold/toposort?lang=cpp

DFS

Kahn

what kind of queue to choose for Kahn?

4.4 Spanning Trees

https://www.cs.cmu.edu/~ckingsf/class/02713-s13/lectures/lec03-othermst.pdf

https://en.wikipedia.org/wiki/Minimum_spanning_tree

4.5 Paths

4.6 Eulerian graphs

4.7 Bipartite graphs

Chapter 5

Paths

https://en.wikipedia.org/wiki/Longest_path_problem

Chapter 6

Structures

6.1 Disjoint sets

Consider n numbers, from 0 up to and including $n-1$.

Let these numbers define n individual sets, with each of the sets having one member.

We might then want to join two of these sets.

For example, joining the set i with the set j gives a new set, consisting of the union of the two sets.

Given a number a, we might want to ask to which set this number belongs?

We can accomplish this by assigning, for each set, a representative of the set.

Initially, all numbers represent their own sets.

After a join, however, we need to select a representative for the resulting set, consisting of the joined sets.

We form trees, with directed links going towards the root of the tree.

Here is an example.

6.2 Fenwick trees

We describe Fenwick trees. Inspiration is mostly taken from Laaksonen (2020).

We start with an integer number v.

We are interested in the bits that are set to 1 in the binary representation of v.

Let's assign indices to the bits, starting with 0 for the rightmost bit.

As an example, the number 52, which is written in binary as

$$0110100$$

has bits 2, 4, and 5 set to 1.

Using the notations $b_1 = 2$, $b_2 = 4$, and $b_3 = 5$, we can write the number 52 as

$$2^{b_1} + 2^{b_2} + 2^{b_3} = 2^2 + 2^4 + 2^5 = 4 + 16 + 32 = 52$$

Generalizing, we can consider a number v with bits

$$b_1, b_2, \ldots, b_k$$

set to 1. This number can be expressed as

$$v = \sum_{i=1}^{k} 2^{b_i} \tag{6.1}$$

Given a number v, define the function $p(v)$ as a function that returns the largest power of two that divdes v.

From Equation 6.1, we see that

$$p(v) = 2^{b_1}$$

Suppose now that we subtract $p(v)$, resulting in a new number $v_1 = v - p(v)$. We can note that $p(v_1) = 2^{b_2}$.

This means that we can compute the terms in Equation 6.1 by iteratively computing the result of the function p and then subtracting the result.

Noting that $p(1) = 0$, we can do this as long as our number is greater or equal to 1.

We can write this as an algorithm, as

Algorithm 6.1 Iterative usage of the function p

```
1: procedure P-ITER(v)
2:     while v ≥ 1 do
3:         v = v − p(v)
4:     end while
5: end procedure
```

As an example, running the algorithm Algorithm 6.1 on the number 52 gives, for the values $p(v)$, the sequence 4, 16, 32.

We now consider the problem of summing values in an array. We assume that the array-indexing starts at the value 1.

Suppse we want to sum the indices from 1 up to and including 52. Noting that $52 = 4 + 16 + 32$, we could make this summation by first summing from from 1 up to and including 32, then from 33 up to and including 48, and finally from 49 up to and including 52.

Given an array a, we could create another array t, where at index v in t, we store the sum

$$t_v = \sum_{i=v-p(v)+1}^{v} a_i \tag{6.2}$$

We refer to the array t as a Fenwick tree.

For our example at hand, using the number 52, we would compute the sum from 1 to 52 as

$$\sum_{i=1}^{52} a_i = \sum_{i=49}^{52} a_i + \sum_{i=33}^{48} a_i + \sum_{i=1}^{32} a_i$$

Using Equation 6.2, we can write this as

$$\sum_{i=1}^{52} a_i = t_{52} + t_{48} + t_{32}$$

We can modify Algorithm 6.1 so that it computes a sum, using the array t. This results in

Algorithm 6.2 Computing the sum in a Fenwick tree

```
1: procedure P-SUM(v, t)
2:     sum = 0
3:     while v ≥ 1 do
4:         sum = sum + t_v
5:         v = v − p(v)
6:     end while
7: end procedure
```

Given an array a, we can compute range sums using Algorithm 6.2 , using a Fenwick tree t.

We might wonder how to update t, if the array a is updated?

We see, from Equation 6.2, which elements a_i that are involved in a given sum. So if an array value at index i is updated, we must update t at all places where the term at index i is included in a sum.

Looking at Algorithm 6.2 , we see that when computing a sum, we visit the places t_v, with v being updated as

$$v = v - p(v)$$

and ending as soon as v becomes 1 or smaller than 1.

So if we update index i in a, we could update t for index i as well as all indices above i, computed as

$$i = i + p(v)$$

This gives an algorithm for update, as

Algorithm 6.3 Updating a Fenwick tree

```
1: procedure P-UPDATE(v, δ, t)
2:     v_max = length(t)
3:     while v ≤ v_max do
4:         t_v = t_v + δ
5:         v = v + p(v)
6:     end while
7: end procedure
```

You might also want to checkout the paper that gave Fenwick trees their name.

Chapter 7

Sequences

https://www.geeksforgeeks.org/count-inversions-of-size-three-in-a-give-array/

https://www.geeksforgeeks.org/inversion-count-in-array-using-bit/

https://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/

Chapter 8

Strings

Chapter 9

Numbers

https://math.stackexchange.com/questions/1125070/counting-the-numbers-with-certain-sum-of-digits

9.1 Prime numbers

Checking that a number is prime

Wheels

Sieves

https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

Factors

Euler's totient function

https://www.geeksforgeeks.org/eulers-totient-function/

9.2 Binary exponentiation

https://cp-algorithms.com/algebra/binary-exp.html

We want to compute

$$a^n$$

Express n in binary, and denotes the bits set to one, with indices starting at zero for the right most bit, as

$$b_1, b_2, \dots, b_k$$

Using this information, the number n can be expressed as

$$n = \sum_{i=1}^{k} 2^{b_i}$$

This means that a^n can be expressed as

$$a^n = a^{\sum_{i=1}^{k} 2^{b_i}} = \prod_{i=1}^{k} a^{2^{b_i}} \tag{9.1}$$

We can create an algorithm for this, as shown in Algorithm 9.1 .

We compute a result in the variable v. We do this by using a factor f, which is updated every iteration of the loop which starts at line 5. The factor f takes on the values $a, a^2, a^4, ...$, and it is updated as long as the value of m is greater than zero.

The update of v at line is done only if the last bit of m is set to one. If this is the case, we update v.
At line 10, we update the value of m by doing a right shift, i.e. a division by two.

Algorithm 9.1 Binary exponentiation

```
1:  procedure BIN-EXP(a, n)
2:      v = 1
3:      f = a
4:      m = n
5:      while m > 0 do
6:          if m&1 = 1 then
7:              v = vf
8:          end if
9:          f = f^2
10:         m = m ≫ 1
11:     end while
12: end procedure
```

Consider the $n + 1$ first Fibonacci numbers, here denoted as $y_0, y_1, \dots, y_n$, and defined as

$$\begin{aligned} y_0 &= 0 \\ y_1 &= 1 \\ y_k &= y_{k-1} + y_{k-2}, \quad k \geq 2 \end{aligned}$$

Introducing the state vector

$$x(k) = \begin{bmatrix} x_1(k) & x_2(k) \end{bmatrix}^T = \begin{bmatrix} y_k & y_{k-1} \end{bmatrix}^T$$

with the initial value

$$x(1) = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$$

and the matrix

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

we can formulate an equation for $x(k+1)$ as

$$x(k+1) = Ax(k)$$

As an example, we can compute $x(n)$ as

$$x(n) = Ax(n-1) = A^2x(n-2) = A^3x(n-3) = ... = A^{n-1}x(1) \tag{9.2}$$

We see that the n-th Fibonacci number y_n can be computed as $x_1(n)$ with $x(n)$ computed from Equation 9.2

$$a = x + 1$$
$$b = x_3 + 95$$

9.3 Coprime

https://math.stackexchange.com/questions/1929485/how-many-coprime-ordered-pairs-are-there-up-to-n

Chapter 10

Optimizations

10.1 Knapsack

We have n items, with values v_i and weights c_i.

For each item, we can choose to use it or not.

We use a binary vector x, with elements x_i to indicate our choices.

We use the notation $x[1:n]$ to indicate a vector x with n components, i.e.

$$\begin{bmatrix} x_1 & x_2 & ... & x_n \end{bmatrix}^T$$

The total capacity of the knapsack is C.

We want to maximize the total value

$$V(n, x) = \sum_{i=1}^{n} v_i x_i$$

This means that we want to compute

$$\max_x V(n, x) = \max_x \sum_{i=1}^{n} v_i x_i$$

subject to the constraint that we don't exceed the capacity of the knapsack. This constraint can be written as

$$\sum_{i=1}^{n} c_i x_i \leq C$$

We also have the constraint that each element can be chosen, or not chosen, which can be written as

$$x_i \in \{0, 1\}, \quad i = 1, ..., n$$

10.1.1 Trying all possibilities

10.1.2 Greedy

10.1.3 Dynamic programming

Consider an optimal solution to the problem

$$\max_{x[1:n]} V(n, x[1:n])$$

We use the notation $V^*(n, x^*[1:n])$ for this solution, i.e.

$$\max_{x[1:n]} V(n, x[1:n]) = V^*(n, x^*[1:n])$$

Suppose that the last element in the optimal solution is zero, i.e. $x_n^* = 0$.

This means that

$$V^*(n, x^*[1:n]) = V(n-1, x^*[1:n-1])$$

and that the optimal solution can be expressed as

$$\max_{x[1:n]} V(n, x[1:n]) = V(n-1, x^*[1:n-1])$$

We might wonder if this is an optimal solution to the problem

$$\max_{x[1:n-1]} V(n, x[1:n-1])$$

Chapter 11

Summary

In summary, this book has no content whatsoever.

```
1 + 1
```

```
[1] 2
```

Chapter 12

Extra

Directed and Undirected

```
# kalle was here

def main():
  print('oh dear, this is Python')
```

Let's make some pseudo code, as

Algorithm 12.1 Quicksort

```
1: procedure QUICKSORT(A, p, r)
2:     if p < r then
3:         q = PARTITION(A, p, r)
4:         QUICKSORT(A, p, q − 1)
5:         QUICKSORT(A, q + 1, r)
6:     end if
7: end procedure
8: procedure PARTITION(A, p, r)
9:     x = A[r]
10:    i = p − 1
11:    for j = p to r − 1 do
12:        if A[j] < x then
13:            i = i + 1
14:            exchange A[i] with A[j]
15:        end if
16:        exchange A[i] with A[r]
17:    end for
18: end procedure
```

Cool, right?

References

Halim, Steven, and Felix Halim. 2018. *Competitive Programming 4 - Book 1.* https://cpbook.net.

———. 2020. *Competitive Programming 4 - Book 2.* https://cpbook.net.

Laaksonen, Antti. 2020. *Guide to Competitive Programming - Learning and Improving Algorithms Through Contests.* 2nd ed. Springer.

Sannemo, Johan. 2020. *Algorithmic Problem Solving.* https://jsannemo.se/aps.pdf.