So you want to be a Competitive Programmer?

Indian National Olympiad in Informatics 2018 — Training Guide

by Aulene De

This guide is meant to build a foundation in Algorithms and Data Structures for the Indian National Olympiad in Informatics, referred to as the INOI, the Indian National Test for entry to the International Olympiad in Informatics. This guide is for you if you are a beginner to competitive programming. Although this is mainly aimed towards the INOI, I'd still give this a whirl if you're still brand new to programming contests.

This guide is definitely not a comprehensive document like Antti Laaksonen's Competitive Programming Handbook. He does a better job explaining concepts than I ever could, and I highly recommend reading his document over mine.

The objective is to ensure that a person learns through hands-on problem-solving rather than giving a lot of theory. There will be a little bit of theory here, but I recommend people consult an Algorithms textbook like CLRS for the more theoretical parts.

I recommend that a person try every problem mentioned here for 45-60 minutes at the minimum. Try to come up with a logic, implement it, find bugs, debug, figure out why that approach didn't work and make sure knowledge gained doesn't go to waste.

This guide isn't supposed to be the one-stop-shop. In fact, it is a track with breadcrumbs. I recommend everyone use Google throughout every point of the guide, looking for problems on certain topics and alternative explanations on the more theoretical parts. I expect everyone to not limit themselves to this one document.

Without much talk, let's begin. I try to embed humour into my guides so that reading doesn't become stale, but I find it hard to convey humour through text. Hope that you have a great time ahead, and good luck!

A Brief Overview of the INOI Syllabi
Bold = Important

1. Complexity Analysis of Algorithms
2. Sorting
3. Searching
4. **Basic Graph Algorithms**
5. **Dynamic Programming**
6. Heaps
7. Permutations
8. Directed Acyclic Graphs (DAGs)
9. Prefix Sums
10. Sliding Window Algorithms

Here's a trade secret. Traditionally, the INOI has two problems — usually a Dynamic Programming problem and a Graph problem. We'll be covering a good portion of the syllabi in this guide.

Both problems are divided into subtasks (for a total score of 200, 100 for each problem). What this means is that, depending on the efficiency of the Algorithm/Data Structure you use, your program could work on a partial score, but not be efficient enough to get a full 100.

In example, if a problem is to be graded out of 100, it could be divided into 20-40-40, and you can use a program that might not be as efficient in order to get a partial score.

But nobody wants a partial score.

"So Aulene-senpai, I really want to go to IOITC, this is super cool I want to get a 200/200 PLEASE HELP ME D:<". Fear not. I shall take you as my padawan. With that, we shall move on to our first encounter with Algorithms.

Getting familiar with Competitive Programming

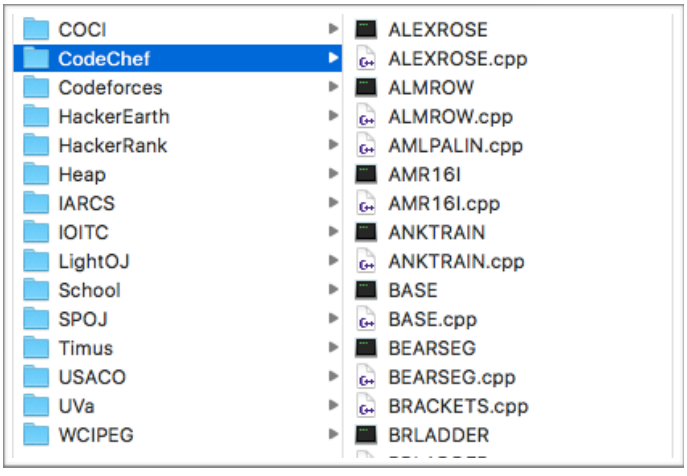
The best way to start is by solving problems. I'll be linking a few problems here to introduce you to the online judge system. Basically, since there is no way to test out your code on every single case, there is a certain amount of 'test data' made against which your code is run. Then, your output is compared with the output of another program which is guaranteed to be correct. If outputs match, we can confidently say that your code/algorithm is correct.

The first step is to make an account on CodeChef, SPOJ and the IARCS Online Judge. A majority of the problems I'll be giving you will be on these websites.

Without much ado, I encourage you to familiarise yourself with CodeChef. Also, most problems I'll tell you to solve will have editorials linked below the problem. Look at them if you feel stuck, but don't look at them before giving ample time to the problem. Whenever you feel that there's nothing more you can do, look at the editorial and get the accepted verdict. Try the following problems. -

- TEST - Do read the Input/Output tab.
- INTEST
- HS08TEST
- FLOW001
- FCTRL2

Soon enough, you could make a nice little folder with all the problems you massacre and brag about it later. :D



Searching and Algorithm Analysis

Task: Given an a array of numbers, check if a number is present in the array. If it is, return the index (location) of the number in that array.

Example
Let our array be {3, 4, 1, 5, 2}. If I ask you the location of the number '3', you return 1 *. If I ask you the index of the number '6', you return 0 i.e false * *.

* - 1-based indexing, meaning that locations of the numbers start from 1 (Location of '3' is 1, location of '4' is 2 .. location of '2' is 5)
* * - In Boolean, 1 corresponds to being 'True' and 0 corresponds to a number being false.

Seems simple enough right? Mhm, it is! Let's take our first look at two Algorithms — Linear Search and Binary Search — see what makes them so different and when to use them.

Linear Search

The idea of Linear Search is simple, we start at the first index and check the number at that index. If we find our number, we say "Found it Aulene-san" and return the index, and if we don't, we say "It's not here ;c" and return false (0).

Note: As we go further along this guide, I will introduce to somewhat-formal notations, so please do read every section 2-3 times in order to grasp all the information you can.

Example

Our array A = {3, 4, 1, 5, 2} and I have asked you the location of the number '1' in A.

We start from index A[1] and check the number there. It's 3, so we go on checking. A[2] is 4, A[3] is... what we're looking for. :D Return that index i.e 3.

What if the number we're looking for is '7'? Well then we check A[1], A[2] .. A[5]. No dice. We return 0.

With that, here's the code for Linear Search.

```
1   int LinearSearch(int A[], int N, int V)
2   {
3       // A[] is our Array, N = Number of Elements in A
4       // V = Number we are looking for
5
6       for(int i = 1; i <= N; i++)
7           if(A[i] == V)
8               return i;
9       return 0;
10  }
```

Alright, brace yourselves.

Now, let's say we have an array consisting of a 100 elements, what is the **worst-case** time that Linear Search will take in order to determine if an element exists in the array or not? Take a minute to think about this and come up with an answer before reading further.

Hint: Think of how many times the for loop will run before returning a value.

The answer is 100. The worst-case is when the element we are looking for is not present in the array, thus the for loop will run a 100 times before saying, "This ain't here, bruh.".

Now, let the number of elements in the array be a variable i.e N. Now, in the worst-case, the for loop will run N times before returning an answer.

Introducing Big-O notation - $O(\ )$

Big-O notation is a way to express the worst-case complexity of an algorithm. You'll be seeing a lot more of it from now on, and the idea will seem complex at first, but it'll get easier in a bit.

With reference to the case in Linear Search — where the number of elements is N — we will express our complexity as $O(N)$. Basically, we're saying that, in worst-case scenario, our algorithm will take N steps.

In simpler terms, the block of code inside the Linear Search for loop runs N times, with each execution being called one operation i.e $O(1)$. So, we say that the complexity of our code is $O(1 \times N)$ or $O(N)$.

Fear not, you'll get familiar with the idea as you solve more problems.

With that, let's move on to...

Binary Search

Binary Search works on the principle that, in a sorted array (an array in ascending/descending order), we need not check every single number in the entire array to check if a number is present like we did in Linear Search.

Let's say we're looking for a number V in an array A consisting of N elements. Let's define 3 variables as follows -

1. Left = 1
2. Right = N
3. Mid = (Left + Right) / 2

We check the element present at A[Mid].

If A[Mid] > V, we can say that all elements from Mid to N (i.e A[Mid], A[Mid+1] ... A[N]) are greater than V, hence we don't need to look for V between Mid and N.

If A[Mid] < V, we can say that all elements from 1 to Mid (i.e A[1], A[2] .. A[N]) are lower than V, hence, we don't need to look for V between 1 and N.

If A[Mid] = V, we have found our element and return the index.

We repeat this process over and over until -
1. A[Mid] = V, which is when we return the index **OR**
2. Left > Right, which means that we could not find the number V in our array and return 0.

Example

Let our array A = {10, 20, 30, 40, 50, 60, 70}, thus number of elements N = 7. Let's say we're looking for the number V = 50.

By our definitions above, Left = 1, Right = 7. This means that Mid = 4.

At A[Mid], we find 40. Since 40 < 50, we can safely say that A[1] upto A[Mid] are also lower than 50. Why? Because our array is **sorted**. (Don't worry, we'll discuss sorting soon)

We update Left to be equal to Mid+1 i.e equal to 5 and repeat the process, making Mid = (5+7)/2 = 6.

At A[Mid], we find 60. Since 60 > 50, we can safely say that A[Mid] upto A[Right] are also greater than 50.

This time, we update Right to be equal to Mid-1 i.e equal to 5 and repeat the process, making Mid = (5+5)/2 = 5.

Check A[5], we see A[5] = 50, which is the number we were looking for. We return the index 5.

What if we were looking for a number not present in the array i.e 55? Well, the process would go just about the same as above, except on the last step, we would return 0 since A[5] =/= 50.

Because I'm a super-nice person, I'll give you the code for Binary Search, but make sure you implement it on your own before reading further. *virtual pinky promise?*

```
1  int BinarySearch(int A[], int N, int V)
2  {
3      int Left = 1, Right = N, Mid;
4
5      while(Left <= Right)
6          {
7              Mid = (Left + Right) / 2;
8
9              if(A[Mid] == V)
10                 return Mid;
11             else if (A[Mid] < V)
12                 Left = Mid+1;
13             else
14                 Right = Mid-1;
15         }
16
17     return 0;
18 }
```

But Aulene-senpai, I already know Linear Search, why do I need to know Binary Search?

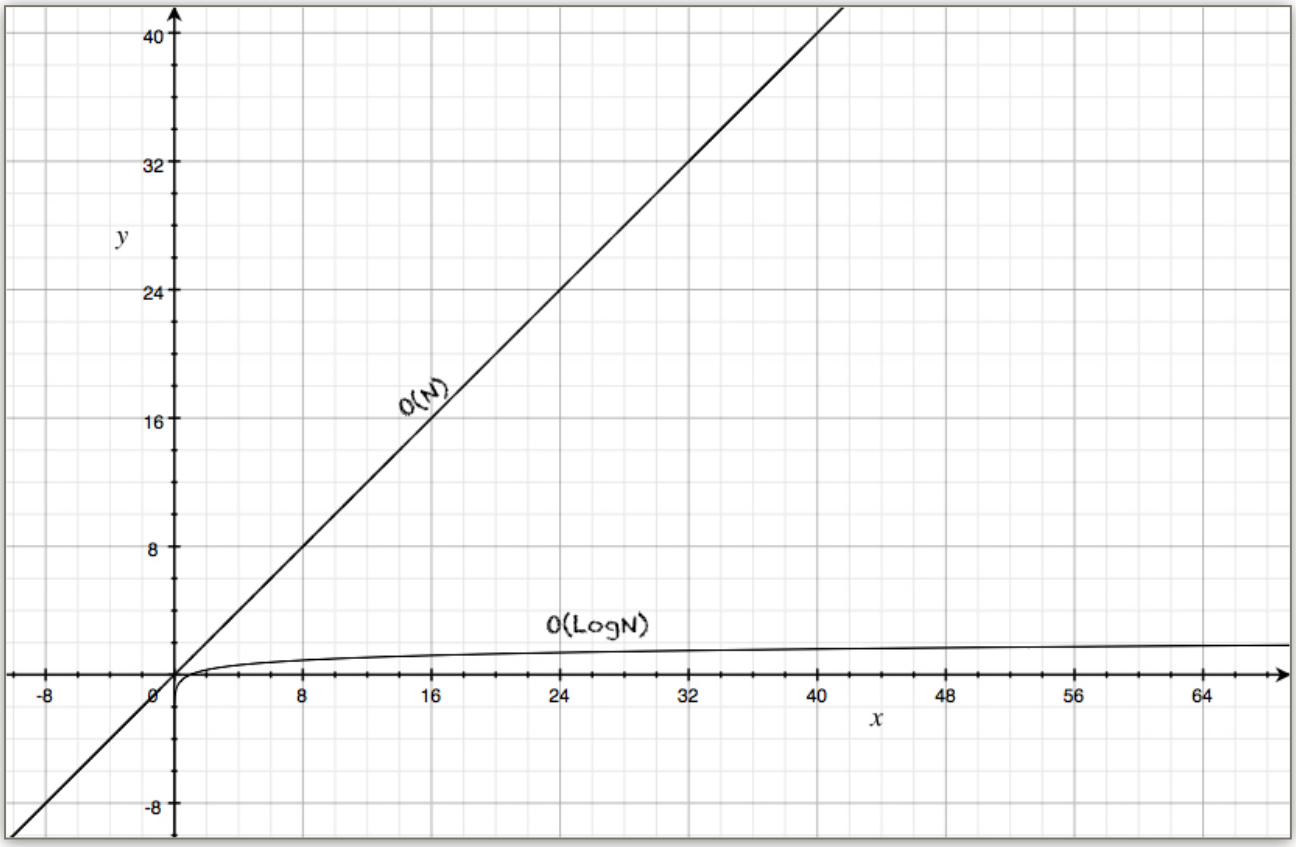
Patience, youngling.

Binary Search v. Linear Search

Remember when we spoke about analysing the worst-case complexity of an algorithm (Big-O Notation)?

Well, it turns out Binary Search is a LOT faster than Linear Search. This is because we're eliminating almost half of our possible indices in one operation, rather than looking through each and every element like we did in Linear Search.

Instead of taking **N** steps like we did in Linear Search, we're taking **LogN** steps. Hence, the complexity of Binary Search is $O(LogN)$, and you can see the difference in this beautiful graph below.



Revisiting Complexity Analysis

Consider the block of code below and try to estimate the complexity our Algorithm.

```
1  for(int i = 0; i < N; i++)
2      cout << BinarySearch(A, N, i) << endl;
3
```

Simply put, since Binary Search takes $O(LogN)$ time to execute and our for loop runs in $O(N)$, we can say that the time complexity of our algorithm is $O(N \bullet LogN)$.

Similarly, the following code has a complexity of $O(N^2 \bullet LogN)$.

```
1  for(int i = 0; i < N; i++)
2      for(int j = i; j < i + N; j++)
3          cout << BinarySearch(A, N, j) << endl;
4
```

Now, before we move on to the next section, I want all of you to solve the following problems in the order below. It's fine if you can't solve a problem, don't look for solutions/editorials immediately. Try to think until you feel exhausted of ideas. After that, get a hot dog and read the editorial, then close the editorial and implement what you learnt until you receive the Accepted verdict.

1. BSEARCH1
2. FORESTGA
3. AGGRCOW

An Introduction to Recursion

“Alright, so recursion is everything, and recursion is recursion. Everything in programming is recursion. I am recursion. YOU are recursion. 42 is recursion and recursion is 42.” - Yours truly

Quotes aside, understanding recursion is key if you want to be a good competitive programmer.

Well sensei, what is recursion?

Recursion is to use small sub-problems in order to solve larger problems.

You’ve all played with Legos, right? These little fun-filled delightful little creatures until you step on them and see your life flashing before your eyes?

Anyway, so Legos start small; you put one block on top of the other until you build the Walt Disney castle :3.

Recursion is similar to Legos. You take a very small problem i.e a 'block' and use it to solve/find the answer to larger problems (i.e A Disneyland made of Lego blocks. I'm sorry I had a sheltered childhood.)

Alright, enough jokes. Recursion is serious business. Understanding recursion would make this entire process a lot simpler, so be prepared to read this section over and over again.

A Simple Series

Let's take a look at Fibonacci numbers. If you don't know what they are, Fibonacci numbers are defined as follows -

$$F[i] = F[i-1] + F[i-2]$$
$$F[0] = 0$$
$$F[1] = 1$$

.. where F[i] denotes the ith Fibonacci number. The Fibonacci series looks like this -

$$0,1,1,2,3,5,8,13,21,34,55,89...$$

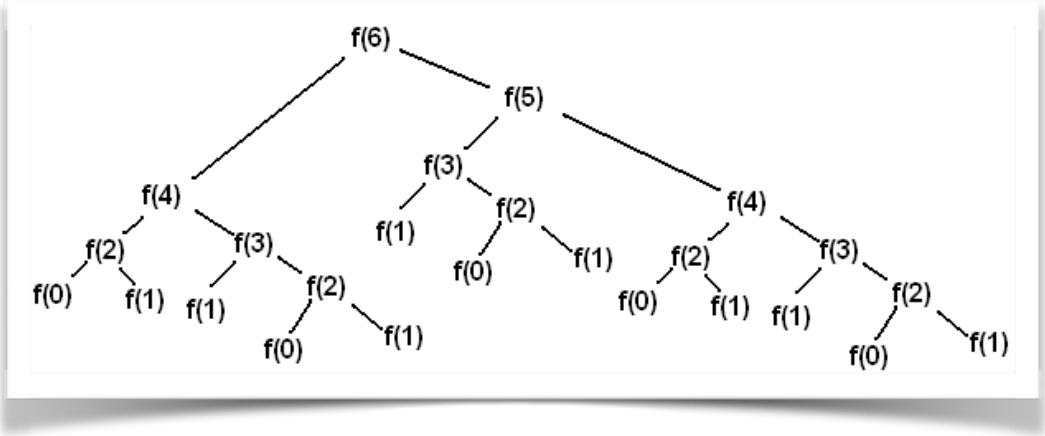
Now, let's look at our first equation. We know the values of the first and second Fibonacci numbers, and we can use them to compute more. We can define a function as follows in order to generate Fibonacci numbers, where N signifies the Nth element in F[].

```
1  int Fibonacci(int N)
2  {
3      if(N == 0)
4          return 0;
5      else if(N == 1)
6          return 1;
7      else
8          return Fibonacci(N-1) + Fibonacci(N-2);
9  }
```

Let's take a look at the code.
If N = 0, return 0.
If N = 1, return 1.

However, since we don't know what the Fibonacci number at the Nth position could be, we tell the computer to call the Fibonacci function again to calculate the Fibonacci number at the (N-2)th and the (N-1)th positions, until we finally reach either N = 0 or N = 2.

Here's a very beautiful and intricately crafted tree in order for you to understand this -



Now, developing recursive thinking is challenging and it won't make sense to you in a day. Don't fret. Try to define the problem in simpler terms and work from there. Recursion is super important, and you'll come to appreciate it's elegance and splendour and all qualities you dream to have in your significant other. ;)

A nice visual way to see the above is linked here.

We'll be using recursive functions when we take a peek at Merge Sort in the next section and recursion is the underlying principle of Dynamic Programming.

Sorting

As defined earlier, sorting is simply putting an array of numbers in ascending order.

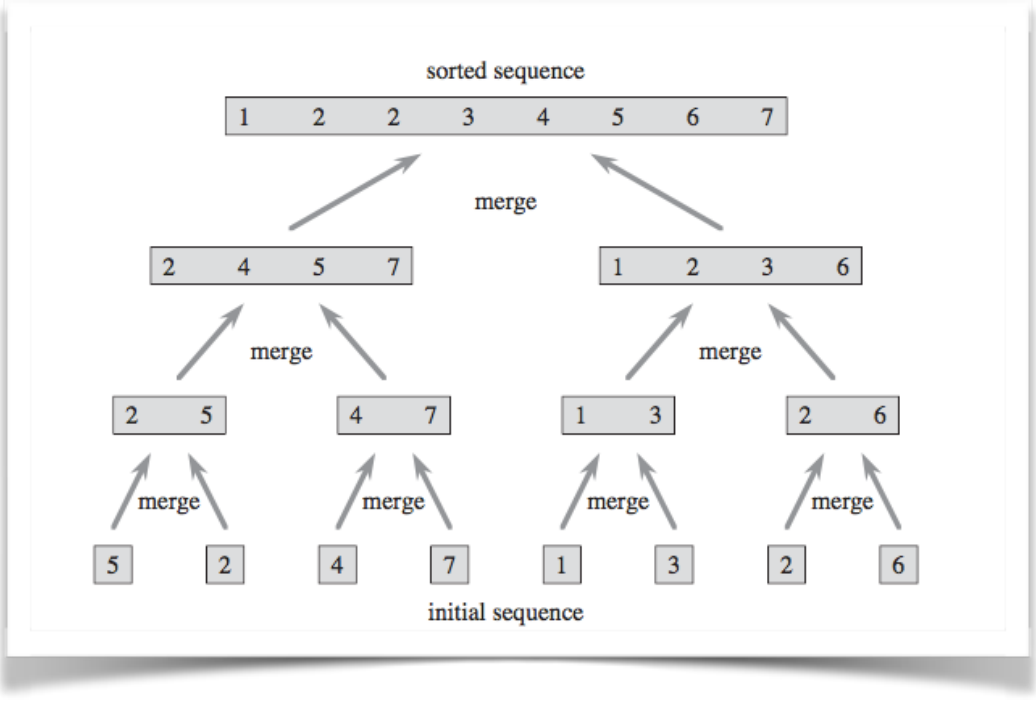
Note: I won't be going in-depth into sorting. This is simply because, most of the time, you'll be using the C++ sort() STL function to accomplish the task.

I'll be discussing Merge Sort here, because it has a very elegant recursive definition and serves as a nice introduction to the Divide and Conquer paradigm. As the note stated, we won't be spending too much time here because I have to get to the g o o d s t u f f.

Merge Sort works in 3 steps -

1. **Divide** an N-element sequence to be sorted into two subsequences of n/2 elements.
2. **Conquer** - sort the two sequences recursively using Merge Sort.
3. **Merge** the two subsequences to produce the sorted answer.

Anyway, this diagram should explain everything. It's straight out of CLRS as well. Again, this section is pretty useless, but really it's only to demonstrate how Merge Sort works.



You can find the code for Merge Sort here and find a visual way to see it in action here. You'll NEVER implement it programming contests, but implement it once and just use the C++ Sort function afterwards.

Alright, time for some problems! Feel stuck? Use Google :)
Do ALL of these. They will strengthen your Searching and Sorting concepts tremendously.

1. TSORT
2. INCPRO4
3. BOOKSHELVES - ZCO (Sorting)
4. VARIATION - ZCO (Sorting)
5. WORMHOLES (Sorting + Binary Search) - ZCO
6. PYRAMID - (Sorting + Binary Search)

Prefix Sums

Alright, so Prefix Sums are super important. Well, Aulene-kun, what are Prefix Sums?

Prefix Sums can be defined as, for the ith element in the in an array, Prefix[i] is the sum of all elements upto and including the nth element.

Formally, we can create a prefix array with the following definitions -

Prefix[1] = A[1]
Prefix[i] = Prefix[i-1] + A[i]

Example
Let's say we have an array A = [1, 3, 5, 6, 10].
Going by our definitions, Prefix = [1, 4, 9, 15, 25].

So why are they important? Well, computing sums from an element i to another element j will result in $O(N)$ worst-case complexity. Let's assume that we have to compute sums K times throughout the course of a problem, our complexity is suddenly $O(N \cdot K)$.

Now that's bad. You know why it's bad? Because we can compute prefix sums **once** in $O(N)$ and find all other required sums in constant time or $O(1)$.

OMG AULENE-SAN, PLEASE TEACH ME YOUR WAYS, WHAT SORCERY, WHAT GRACE, SO MUCH BEAUT, MUCH APPRECI8.

Alright, so think back, how would you use prefix sums to compute sums in constant time?

Let's look at the example above. To reiterate..

Example that I mentioned above but I'm mentioning again because all of you are filthy casuals (haha. jokes. so funny)

Let's say we have an array A = [1, 3, 5, 6, 10].
Going by our definitions, Prefix = [1, 4, 9, 15, 25].

Let's say I have to compute the sum from i = 2 to i = 4. This would be equal to.. 3 + 5 + 6 = 14.

Now, what if I subtract Prefix[1] from Prefix[4]?
This would be equal to 15 - 1 = 14, which is our answer. One less for-loop = very elegant code.

1. CSUMQ
2. JURYMARKS

Anyway, you need to know prefix sums for what's about to follow. It's time for my favourite section, the most important one for the INOI and relatively tough so be prepared for l o t s of problems.

If you felt that the guide was like a low-effort YouTube video up until now, it's because everything upto this point doesn't matter nearly as much as what the following does.

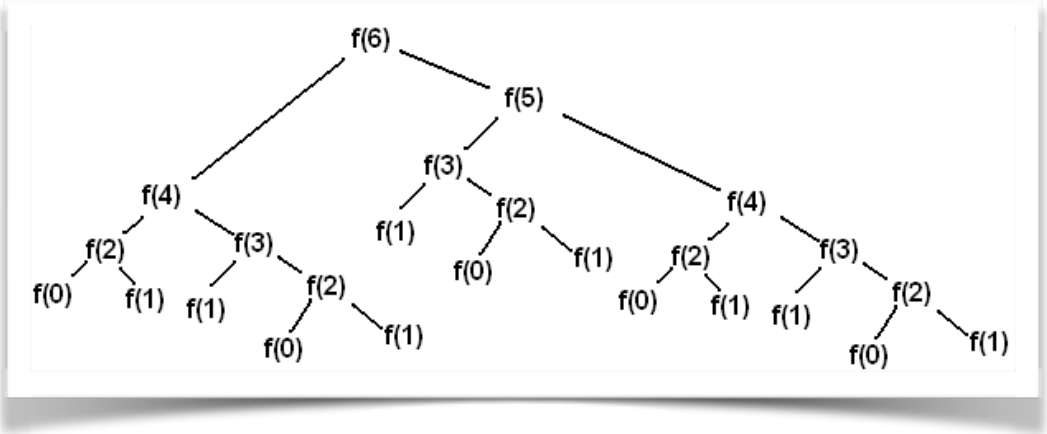
Really, if you want to clear the INOI, these next 2 sections are everything. By everything, I mean..

E V E R Y T H I N G.

Scene 6
Enter: Dynamic Programming

You remember how Recursion works, right? You use small problems and use them to find solutions to bigger problems.

Here's the problem with Recursion. Let's look at the recursive calls for the Fibonacci problem again, shown below.



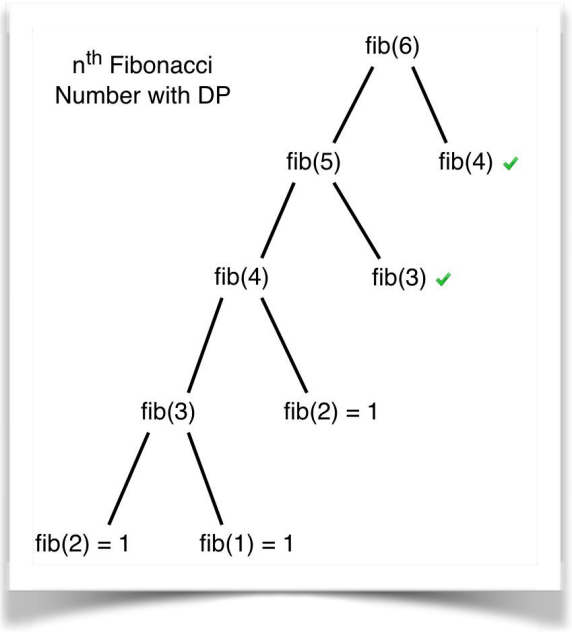
Try to analyse the complexity of the above. Hint: try to think in terms of how many recursive calls are generated at each step!

If you guessed that the complexity would be exponential, congratulations, you're eligible for one of my freshly-baked virtual chocolate chip cookies. To put it formally, if the time to compute the Nth fibonacci number is $T(N)$, then $T(N)=T(N-1)+T(N-2)$.

We can see that, unless we hit F(0) or F(1), we don't return any value. For example, we have to compute F(3) over and over again. Don't you think it would be faster if we somehow 'remembered' the value of F(3)?

Let's take it one step further: what if, every single time we computed a new value for F[], (be it F[3], F[4], F[whatever]), we **stored that value somewhere and returned it as needed**?

Here's how our fibonacci calls look with the optimisation:



With the above optimisation, we have reduced our complexity to $O(N)$, big difference!

Say hello to Dynamic Programming. As I like to call it, Dynamic Programming is 'smart' recursion. Rather than just pursuing recursive calls, we try to store every single unique value we compute and then just return that value.

Now, the best way to learn DP is by doing DP problems. This applies to DP more than any other problem type, because you see, you're a DP problem. You need to solve smaller, easier DP problems in order to be able to solve harder DP problems. Fun, is it not?

So here's what you do. You solve each of these problems in the order that I give you, and you solve them all. Fair warning, you're going to get a lot of problems here on out. Chit-chat's over.

The first thing I want you to do before crossing names off a list like Oliver Queen (Arrow reference, anyone?) is to head over to the following links and read about all sorts of different Dynamic Programming problem types. They're explained beautifully, and do a better job of explaining than my little hands ever could. DON'T read the code presented at ANY of those links.

Here's exactly what I want you to do.

1. Open the problem first.
2. Try to come up with a base case and how you'll build a solution on top of it.
3. Implement your approach. If you can't think of one, proceed to step 4.
4. Read the solution presented at GeeksForGeeks. Again, DON'T look at their code. Analyse complexity. Implement their solution, in your own code.
5. This is the most important one: close the solution, and re-implement.

1. Longest Increasing Subsequence - Problem
2. Longest Common Subsequence - Problem
3. Edit Distance - Problem
4. Grid-Type Min-Cost Path
5. Coin Change
6. 0-1 Knapsack Problem
7. Longest Palindromic Subsequence

Trust me, you do this and the problems that follow, you'll make your DP lives easier by miles.

Give a lot of time to all of the following problems. There's a lot to learn here.

1. CHOOSING CHOCOLATES (ZCO'09, test cases are unavailable though. Still worth trying)
2. SUPW (ZCO'14)
3. IPL (ZCO'13)
4. TRIANGLE (IOI'94)
5. ROUNDTABLE (ZCO'12)
6. BREAK UP (ZCO'15)
7. BYTESM2
8. TILINGS
9. COINS
10. CALVINSGAME (INOI'13)
11. LONGPALIN
12. EDIST
13. BUFFALOES
14. FLOWERS
15. KTREE
16. TETRAHEDRON
17. BRACKETS (INOI'16)
18. POSTOFFICE (IOI'00)
19. TRAINING (INOI'17)
20. OFFICEKEYS
21. STARSKY

With that, I'll conclude the Dynamic Programming section of this guide. This is probably the topic that will require you to challenge yourself the most, so do as many DP problems as you can outside of those mentioned on this guide.

Also, STOP.
If you haven't done so already, I recommend you check out my document on STL. Graphs will require heavy STL implementations, thus it's a good time to know basic data structures and STL now.

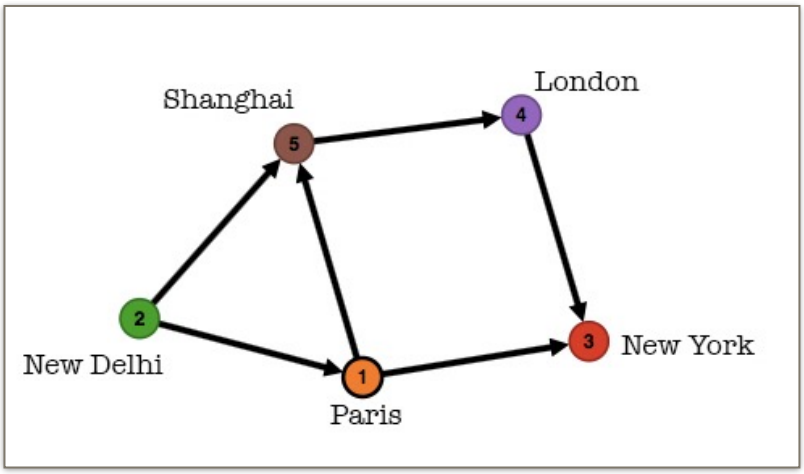
Playing with Graphs by ~~Amit M. Agrawal~~ Austin LegendFlame (I dare you to Google 'Austin LegendFlame')

Let's answer the first question: what are graphs? Well, graphs are basically a set of objects, some of which are linked to each other.

Example
Let's say we have five cities, Paris **(City 1)**, New Delhi (**City 2**), New York (**City 3**), London **(City 4)** and Shanghai **(City 5)**. I have to go to New York **(City 3)**, and I'm currently in New Delhi **(City 2)**. Here's the flight plan for today:-

- New Delhi to Paris **(2 -> 1)**
- New Delhi to Shanghai **(2 -> 5)**
- Paris to New York **(1 -> 3)**
- Paris to Shanghai **(1 -> 5)**
- Shanghai to London **(5 -> 4)**
- London to New York **(4 -> 3)**

We could represent these connections pictorially i.e as a graph.

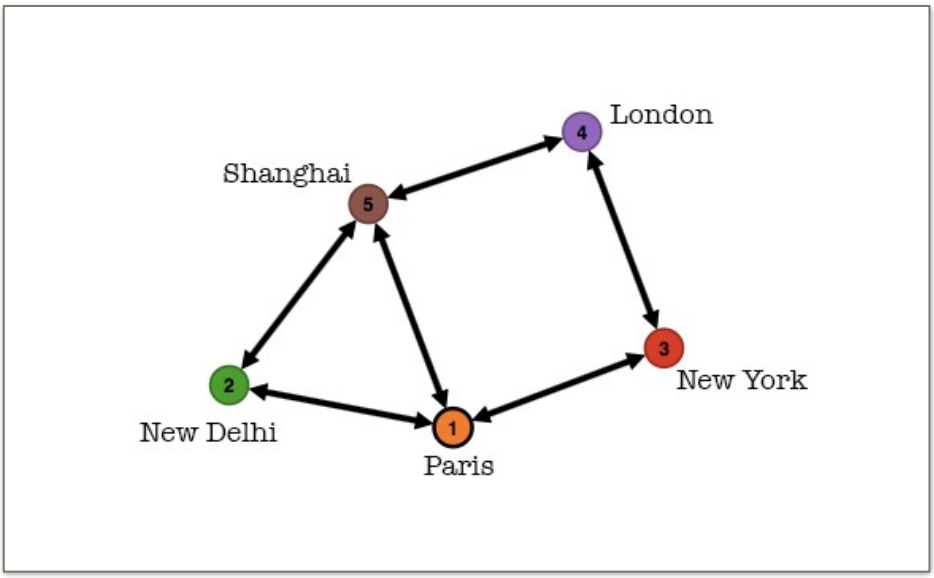


There's a lot of terminology here, so let's break it down one by one.

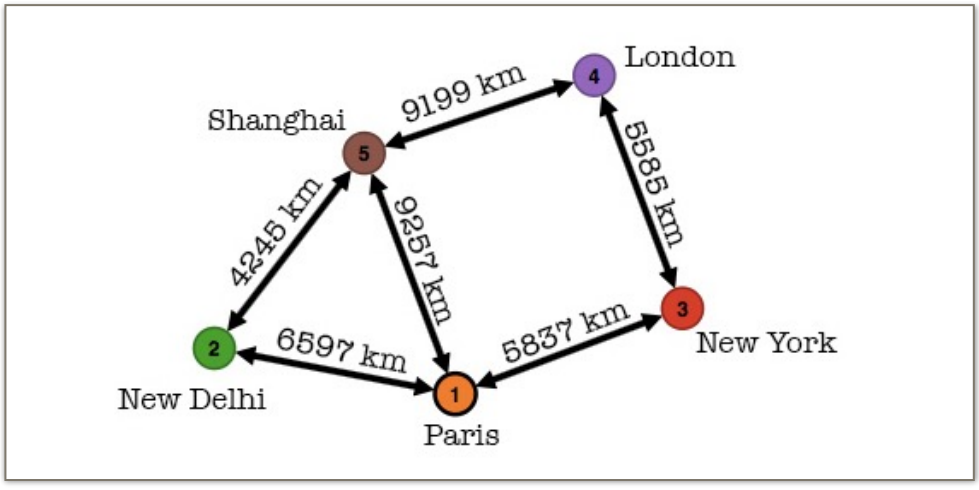
- **Node:** The cities on our graph i.e New York, New Delhi, etc are called nodes. They are the objects on our graph.
- **Edge:** See the connections between cities? They're called edges. They represent a link between two objects, in this case, a flight going from one city to another. You'll encounter two types of edges, Directed and Undirected.

- **Directed Edge:** This implies that there is a one-way connection between two nodes. For example, in our graph, there is an edge from Paris to Shanghai. You cannot use the same edge to travel from Shanghai to Paris.
- **Undirected Edge:** This kind of edge symbolises a two-way connection, meaning that you can use an edge from one node to another in any direction.

Here's how our graph would look like if all the edges were undirected edges.



The above graphs are **unweighted** graphs. This means that traversing any edge has no cost associated with it. However, **weighted** graphs have a cost associated with traversing each edge. In our example example, the edge between two cities could have a weight equal to the distance between them. This would make our graph look something like this -



Now, how do we represent graphs? Glad you asked. Turns out, there are two traditional ways to represent graphs: the Adjacency Matrix and the Adjacency List.

The Adjacency Matrix Representation

This representation is a lot easier to implement. We represent our graph as a 2-dimensional grid, with the rows and the columns equal to the number of nodes in our graph.

If the graph is unweighted, we can simply use graph[x][y] = true to signify that there is an edge between nodes x and y and graph[x][y] = false to signify that there is no edge between x and y.

If the graph is weighted, we initialize all graph[x][y] to be equal to a very high value (say, INT_MAX). Then, we change the value of graph[x][y] to the weight of the edge, given that the edge exists.

Here's how our example would look like in an adjacency matrix representation -

Weighted, Undirected Adjacency Matrix

	1	2	3	4	5
1	∞	6597	5837	∞	9257
2	6597	∞	∞	∞	4245
3	5837	∞	∞	5585	∞
4	∞	∞	5585	∞	9199
5	9257	4245	∞	9199	∞

Unweighted, Undirected Adjacency Matrix

	1	2	3	4	5
1	0	1	1	0	1
2	1	0	0	0	1
3	1	0	0	1	0
4	0	0	1	0	1
5	1	1	0	1	0

The Adjacency List Representation

Aulene-senpai, I already know the Adjacency Matrix representation, why do I need to know the Adjacency List representation?

I'll answer that after class.

In the Adjacency List representation, rather than creating a 2-dimensional grid to store each edge of a graph, we use a vector for each node to store it's neighbours and the weight of the edge between them. If the graph is unweighted, we just store the neighbour.

Thus, for our weighted, undirected graph, the Adjacency List representation will look like this -

Weighted, Undirected Adjacency List (Neighbour, Weight of Edge to Neighbour)

1	2	3	4	5
2, 6597	1, 6597	1, 5837	3, 5585	1, 9257
3, 5837	5, 4245	4, 5585	4, 9199	2, 4245
5, 9257				4, 9199

Why do we need to know the Adjacency List representation AND the Adjacency Matrix representation?

Here's the thing: the Adjacency Matrix representation requires $O(N^2)$ space, so if N >= 10^5, where N is the number of nodes. So when you do try to use an Adjacency Matrix, your computer is going to curse you like a sailor. On the other hand, the Adjacency List representation has an $O(N+E)$ space complexity, where E is the number of edges.

Alright senpai, so if the Adjacency List is s o g o o d, why should I ever use an Adjacency Matrix?

Well, you see, if you ever had to check if an edge exists between two nodes, it's complexity would be $O(N)$ in an Adjacency List, while it would be $O(1)$ in an Adjacency Matrix.

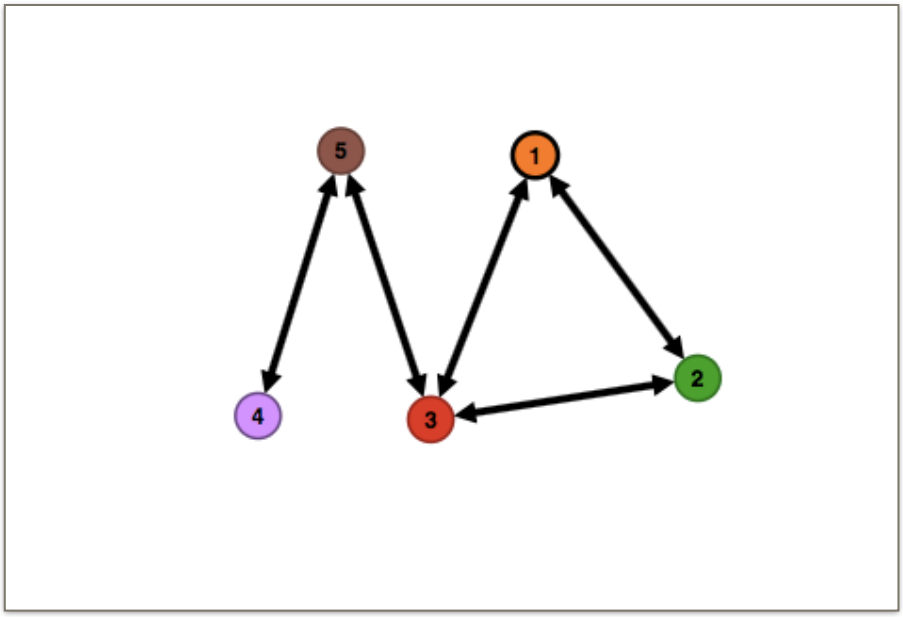
Graph Traversal

Now that you know how to represent a graph, we can move on to how you should explore a graph. What's 'Traversal'? In simple terms, think of nodes as 'cities' and edges as 'roads' that are connecting those cities. We use these roads to go from one city to another. In formal terms, this is exactly what traversal is.

What better way than to start with a problem?

Problem Statement (GREATESC on IARCS)**:**
Given an undirected, unweighted graph comprising of N nodes (1 <= N <= 3500), M edges and two nodes, u and v, calculate the minimum distance to reach v from u. Since the graph is unweighted, assume that all edges have weight = 1.

Example Case:
u = 1, v = 4. The answer for this case will be 3, with the shortest path going from 1 -> 3 -> 5 -> 4.



Spoilers on how to solve this problem follow.

Let's see.. in order to find the shortest path from A to B, we need to explore A's neighbours, correct?

So clearly, if we have to find the minimum distance between node 1 and node 4, we need to traverse every single edge going out from 1, then we need to traverse every edge going out from 1's neighbours, then we need to traverse every edge going out from 1's neighbour's neighbours and so on until we reach node 4.

How would we ever do that?

Depth-First Search (DFS)

How 2 DFS like a PrO (IN THE HOOD) -

1. We create a boolean array visited[], where visited[i] = 1 implies that we have already visited a node and explored it's neighbours, while visited[i] = 0 implies that we haven't explored a node.
2. We'll also create an array distance[], where distance[i] represents the minimum distance of node i from node v. Initially, we set all distance[i] to INF (a very high value, say 1000000).
3. We create an Adjacency Matrix/List and use it to represent our graph.
4. We start our DFS from node 1.
5. We follow the process mentioned above (explore 1's first neighbour in-depth until we return to 1, and then explore the rest of 1's neighbours.)
6. You can find a visual representation of DFS here.

At the end of this 'process', here's how our distance[] array will look like.

1	2	3	4	5
3	3	2	0	1

Also, because I'm really nice, here's the code for the Depth-First Search on an Adjacency Matrix.

```
int dfs(int u)
{
    if(u == v)
        return 0; // if we've reached the final node, we can simply return 0

    if(visited[u])
        return dist[u]; // if node is already visited, we don't need to explore it again

    int i, ans = 1000000; // initializing ans with a high value

    visited[u] = 1; // marking node as visited

    for(i=1; i<=n; i++)
        if(graph[i][u])
            ans = min(ans, 1 + dfs(i)); // travelling to neighbour of u

    return dist[u] = ans; // return computed distance from node v
}
```

Complexity Analysis of Depth-First Search

Adjacency List	Adjacency Matrix
$O(V+E)$	$O(V^2)$

Breadth-First Search (BFS)

BFS is another graph traversal algorithm, like BFS. Fair warning: the following two paragraphs are plagiarised. A 'bit', anyway.

Using DFS, we start the traversal from the root node and explore the search as far as possible from it (aka 'depth-wise'). We keep on exploring a node's neighbours until we hit a leaf node or a node which isn't connected to any non-visited nodes. Exploration of a node is suspended as soon as another unexplored node is found.

However, in BFS, we go in a level-by-level manner. What this means is that rather than going deep into the graph, we completely explore the neighbour's of a single node before moving onto exploring the others.

It's probably better with a visual representation. Since this (as of now, anyway), is a PDF, I can't embed a video file (more importantly, I can't embed in GIF memes... this is a nightmare), so I want you to head over to this VisuAlgo link and try DFS and BFS on the same graph and hit play.
That's about it for BFS. You won't use it much (more like, ever... but it's good to know anyway 👍).

Problems -
1. TASKFORCE
2. GREATESC
3. PPRIME (Hint: Sieve of Eratosthenes)
4. CRITINTS
5. FENCE (INOI'17)
6. PT07Y
7. PT07Z

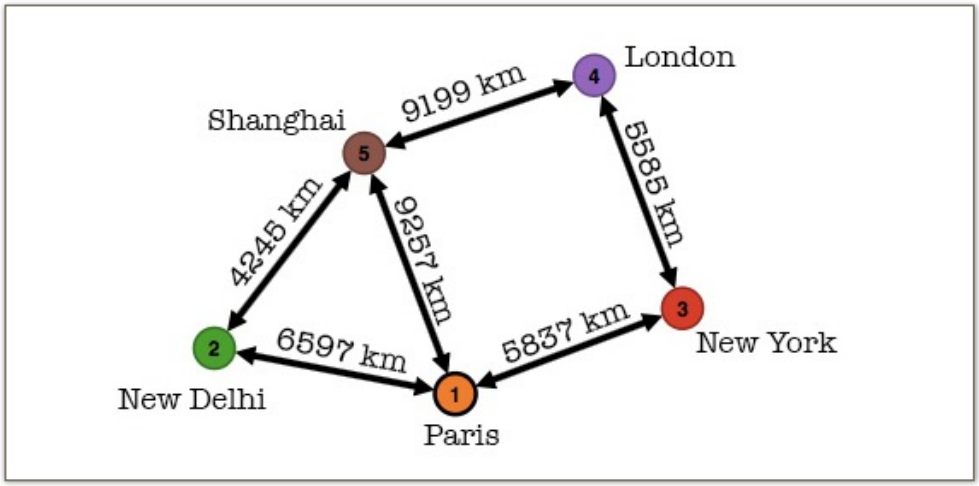
8. PARTY
9. NEWREFORM
10. SPECIES

Shortest Paths

Simply put, in a graph, what is the shortest 'distance' between two nodes U and V, where the distance between two nodes is defined as the sum of edge weights in the path from U to V.

Dijkstra's Algorithm (Single-Source Shortest Paths)

Let's consider our favourite Airport World graph from earlier -



Here's our problem statement: what is the **minimum distance** that I need to travel to get from Paris to any other city? Note that the graph is weighted.

Sample Case:
U = 1, V = 4.
Answer is 11422 km. (Path is 1 -> 3 -> 4)

To formalise the problem a bit more, the problem is asking for single-source shortest paths: basically, given any source node U, you can be asked to find the shortest path to any other node V.

The answer to this problem can be found using Dijkstra's Shortest-Path Algorithm. It works as follows -

1. We select a starting node. In this case, let our starting node be U = 1.
2. We go through all edges outgoing from that node and reduce distances to connected nodes using them.
3. After step 2, we select the node who has the minimum distance from node U, and repeat step 2.
4. Now, we still again an unexplored node with the minimum distance, repeat step 2 and 3 until we have explored all nodes.

The complexity of the above algorithm would be $O(N^2)$. However, we can reduce this by using a max-heap, which would reduce the complexity to $O(NLogN)$. The following is how I implement Dijkstra's.

```
vector< pair<int, int> > g[100007];
priority_queue < pair <int, int> , vector < pair <int, int> > , greater < pair <int, int> > > pq;
int dist[100007];
int n;

int dijkstra(int source, int reach)
{
    int v, d, i;
    vector< pair <int, int> >::iterator it;

    for(i  = 0; i <= n; i++)
        dist[i] = INT_MAX;
    dist[source] = 0;

    pq.p(mp(source, dist[source]));

    while(!pq.empty())
        {
            v = pq.top().first;
            d = pq.top().second;

            pq.pop();

            for(it = g[v].begin(); it != g[v].end(); it++)
                if(dist[it -> first] > d + it -> second)
                    {
                        dist[it -> first] = d + it -> second;
                        pq.p(mp(it -> first, dist[it -> first]));
                    }
        }

    return dist[reach];
}
```

Problems -

1. EZDIJKST (Dijkstra's Sample Problem)
2. SEQUENCELAND
3. WEALTHDISPARITY
4. SHPATH
5. WORDHOP
6. CLIQUED

Epilogue

So... that's about it from what you'll learn from this. Hope you had fun.

This guide was my effort to give back to the community I am very glad to have been a part of over the past year. I started this project exactly on 17th of May, 2017 and finished it on the 2nd of August, 2017 after lots of breaks in writing.

Where do you go from here?

Firstly, give yourself a pat on the back — you've earned it.

You've done more than 50 problems just from this guide, each one harder than the next. Even if you've not, you still have time to do them now.

There's a lot you can learn now - start with Segment Trees. Or try your hand at a couple IOI problems. Whatever floats your boat :P, but do have fun. I also hope that you, with whatever little skill you have, give back to the programming community.

If you have solved all the problems mentioned throughout this manual and have learnt the lost art of Googling, you'll probably be able to handle even harder algorithmic problems and techniques.

I'll leave that to you.

I really do hope this guide helped you in some shape or form.

If you have any critique for this guide or any questions, please email me at aulene@gmail.com, or contact me over Facebook.

See you starside,
Aulene / Aul.
dramatic noise as I press this virtual log out button