3 JANUARY 2019 / #ALGORITHMS

How to Solve the Tower of Hanoi Problem - An Illustrated Algorithm Guide

Dipto Karmakar

of Hanoi problem is. Well, this is a fun puzzle game where the objective is to move an entire stack of disks from the source position to another position. Three simple rules are followed:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack. In other words, a disk can only be moved if it is the uppermost disk on a stack.
3. No larger disk may be placed on top of a smaller disk.

Now, let's try to imagine a scenario. Suppose we have a stack of three disks. Our job is to move this stack from **source A** to **destination C**. How do we do this?

Before we can get there, let's imagine there is an **intermediate point B**.

We can use B as a helper to finish this job. We are now ready to move on. Let's go through each of the steps:

Three disks.

1. Move the first disk from A to C
2. Move the first disk from A to B
3. Move the first disk from C to B
4. Move the first disk from A to C
5. Move the first disk from B to A
6. Move the first disk from B to C
7. Move the first disk from A to C

Boom! We have solved our problem.

Tower of Hanoi for 3 disks. **Wikipedia**

Now, let's try to build the algorithm to solve the problem. Wait, we have a new word here: "**Algorithm**". What is that? Any idea? No problem, let's see.

Photo by bruce mars on Unsplash

What is an algorithm?

An algorithm is one of the most important concepts for a software

development or programming, but for everyone. Algorithms affect us in our everyday life. Let's see how.

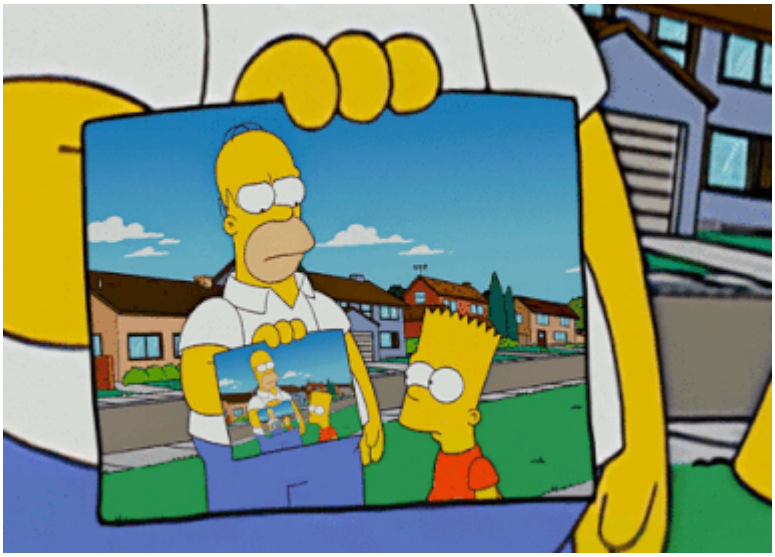
Suppose you work in an office. So every morning you do a series of tasks in a sequence: first you wake up, then you go to the washroom, eat breakfast, get prepared for the office, leave home, then you may take a taxi or bus or start walking towards the office and, after a certain time, you reach your office. You can say all those steps form an **algorithm**.

In simple terms, an algorithm is a set of tasks. I hope you haven't forgotten those steps we did to move three disk stack from A to C. You can also say that those steps are the algorithm to solve the Tower of Hanoi problem.

> *In mathematics and computer science, an algorithm is an unambiguous specification of how to solve a class of problems. Algorithms can perform calculation, data processing and automated reasoning tasks. — Wikipedia*

If you take a look at those steps you can see that we were doing the same task multiple times — moving disks from one stack to another. We can call these steps inside steps **recursion**.

Recursion



Recursion — giphy

Recursion is calling the same action from that action. Just like the above picture.

So there is one rule for doing any recursive work: there must be a condition to stop that action executing. I hope you understand the basics about recursion.

Now, let's try to build a procedure which helps us to solve the Tower of Hanoi problem. We are trying to build the solution using pseudocode. Pseudocode is a method of writing out computer code using the English language.

```
tower(disk, source, intermediate, destination)
{

}
```

an argument. Then we need to pass source, intermediate place, and the destination so that we can understand the map which we will use to complete the job.

Now we need to find a **terminal state**. The terminal state is the state where we are not going to call this function anymore.

```
IF disk is equal 1
```

In our case, this would be our terminal state. Because when there will be one disk in our stack then it is easy to just do that final step and after that our task will be done. Don't worry if it's not clear to you. When we reach the end, this concept will be clearer.

Alright, we have found our terminal state point where we move our disk to the destination like this:

```
move disk from source to destination
```

Now we call our function again by passing these arguments. In that case, we divide the stack of disks in two parts. The largest disk (**nth** disk) is in one part and all other (**n-1**) disks are in the second part. There we call the method two times for -(n-1).

```
tower(disk - 1, source, destination, intermediate)
```

then again we move our disk like this:

```
move disk from source to destination
```

After that we again call our method like this:

```
tower(disk - 1, intermediate, source, destination)
```

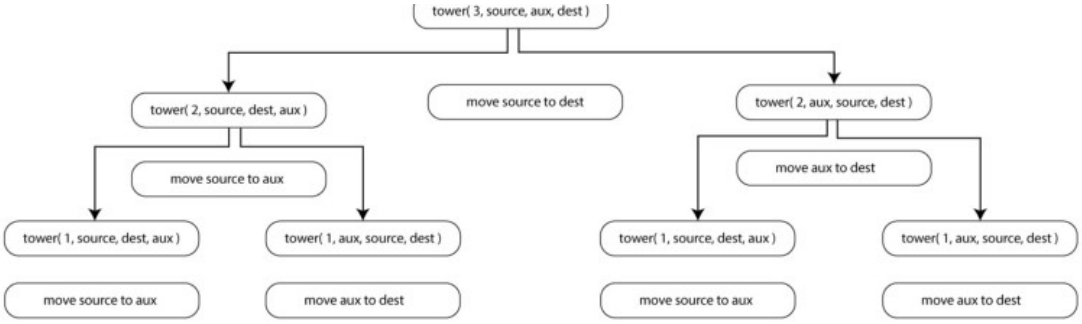
Let's see our full pseudocode:

```
tower(disk, source, inter, dest)

IF disk is equal 1, THEN
      move disk from source to destination
   ELSE
      tower(disk - 1, source, destination, intermediate)   // Step
      move disk from source to destination                 // Step
      tower(disk - 1, intermediate, source, destination)   // Step
   END IF

END
```

This is the tree for three disks:



Tree of tower of hanoi (3 disks)

This is the full code in Ruby:

```
def tower(disk_numbers, source, auxilary, destination)
  if disk_numbers == 1
    puts "#{source} -> #{destination}"
    return
  end
  tower(disk_numbers - 1, source, destination, auxilary)
  puts "#{source} -> #{destination}"
  tower(disk_numbers - 1, auxilary, source, destination)
  nil
end
```

Call `tower(3, 'source','aux','dest')`

Output:

```
source -> dest
```

```
source -> dest
aux -> source
aux -> dest
source -> dest
```

It took seven steps for three disks to reach the destination. We call this a **recursive method**.

Photo by Aron Visuals on Unsplash

calculations

Time complexity

When we run code or an application in our machine it takes time — CPU cycles. But it's not the same for every computer. For example, the processing time for a core i7 and a dual core are not the same. To solve this problem there is a concept used in computer science called **time complexity**.

> Time complexity is a concept in computer science that deals with the quantification of the amount of time taken by a set of code or algorithm to process or run as a function of the amount of input.
>
> In other words, time complexity is essentially efficiency, or how long a program function takes to process a given input. — techopedia

The time complexity of algorithms is most commonly expressed using **big O notation**. It's an asymptotic notation to represent the time complexity.

Now, the **time** required to move **n** disks is **T(n).** There are two recursive calls for (**n-1**). There is one constant time operation to move a disk from source to the destination, let this be **m1**. Therefore:

```
T(n) = 2T(n-1) + m1    ..... eq(1)
```

And

```
T(0) = m2, a constant   ..... eq(2)
From eq (1)
T(1) = 2T(1-1) + m1
     = 2T(0)+m1
     = 2m2 + m1 ..... eq(3) [From eq 2]
T(2) = 2T(2-1) + m1
     = 2T(1) + m1
     = 4m2 + 2m1 + m1 .... eq(4) [From eq(3)]
T(3) = 2T(3-1) + m1
     = 2T(2) + m1
     = 8m2 + 4m1 + 2m1 + m1  [From eq(4)]
```

From these patterns — eq(2) to the last one — we can say that the time complexity of this algorithm is **O(2^n)** or **O(a^n)** where **a** is a constant greater than 1. So it has exponential time complexity. For the single increase in problem size, the time required is double the previous one. This is computationally very expensive. Most of the recursive programs take exponential time, and that is why it is very hard to write them iteratively.

Space complexity

After the explanation of time complexity analysis, I think you can guess now what this is…This is the calculation of space required in ram for running a code or application.

In our case, the space for the parameter for each call is independent of **n**, meaning it is constant. Let it be **J**. When we do the second recursive call, the first one is over. That means that we can reuse the space after finishing the first one. Hence:

```
T(0) = k, [constant] .... eq(2)
T(1) = T(1-1) + k
     = T(0) + k
     = 2K
T(2) = 3k
T(3) = 4k
```

So the space complexity is **O(n)**.

After these analyses, we can see that time complexity of this algorithm is exponential but space complexity is linear.

Conclusion

From this article, I hope you can now understand the **Tower of Hanoi** puzzle and how to solve it. Also, I tried to give you some basic understanding about **algorithms, their importance, recursion, pseudocode, time complexity,** and **space complexity.** If you want to learn these topics in detail, here are some well-known online courses links:

1. Algorithms, Part I
2. Algorithms, Part II
3. The Google course on Udacity
4. Javascript Algorithms And Data Structures Certification (300 hours)

You can visit my data structures and algorithms repo to see my other problems solutions.

I am on GitHub | Twitter | LinkedIn

Donate

Dipto Karmakar

[Full-stack software engineer | Backend Developer | Pythonista] I love to code in python. In my free time, I read books. I enjoy learning and experiencing new skills.

If you read this far, tweet to the author to show them you care.

Tweet a thanks

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started

Continue reading about

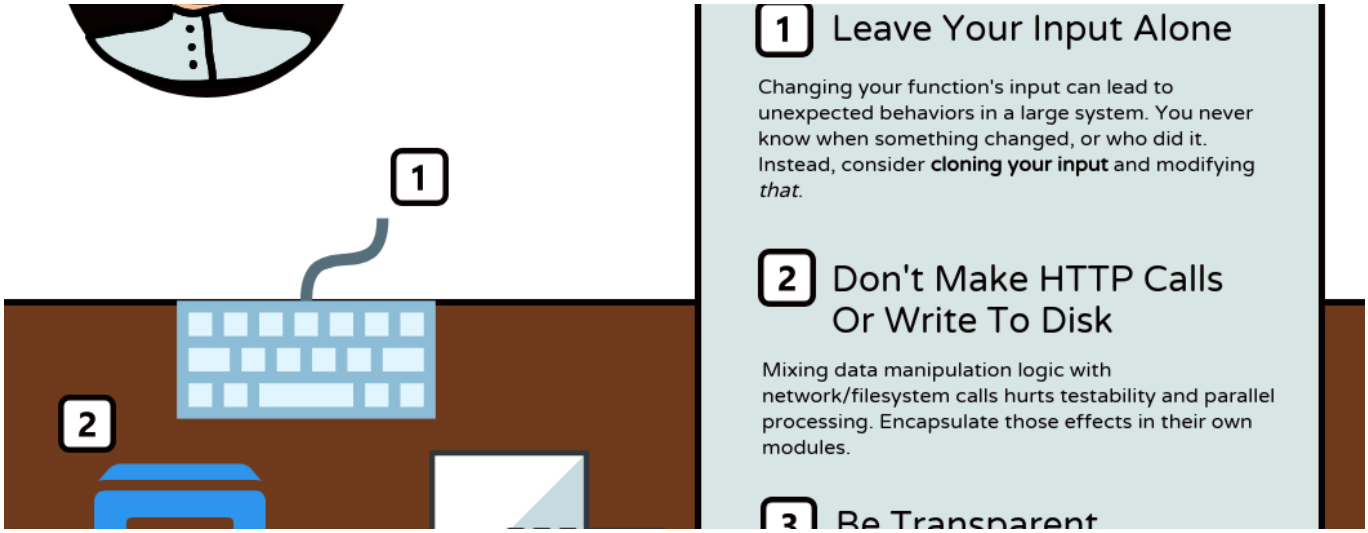
Algorithms

How the Fast Unfolding Algorithm Detects Communities in Large Networks

Dijkstra's Shortest Path Algorithm - A Detailed and Visual Introduction

Python for Finance – Algorithmic Trading Tutorial for Beginners

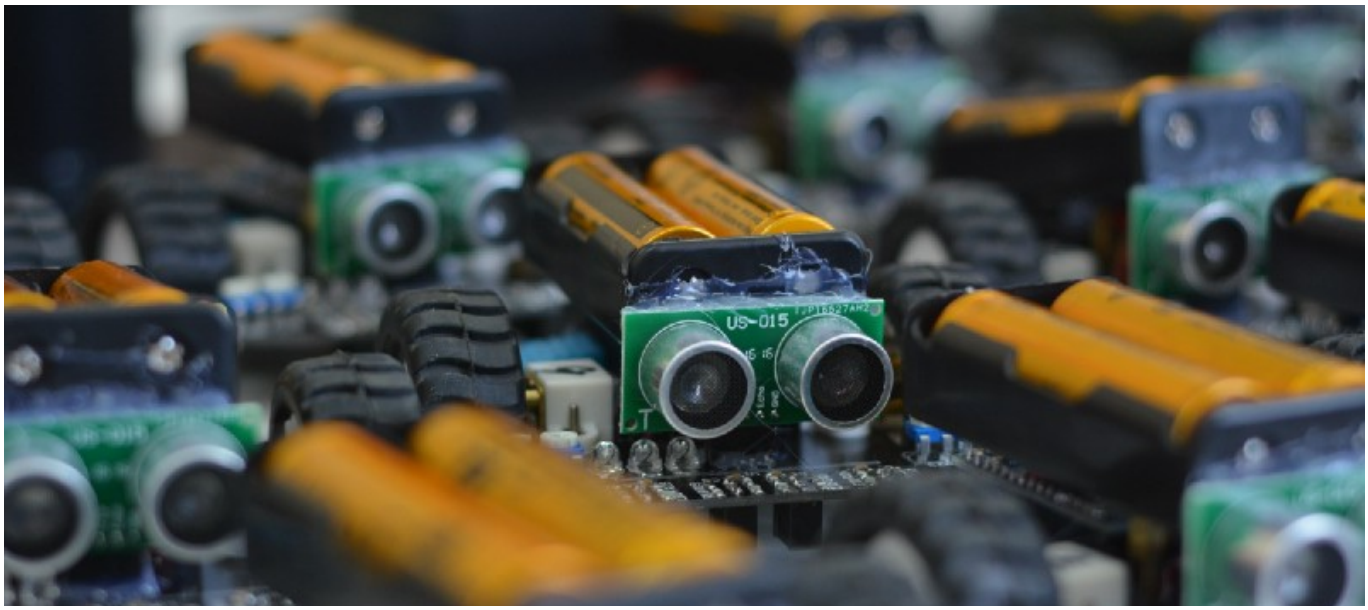
See all 158 posts →



#JAVASCRIPT

What Is a Pure Function in JavaScript?

YAZEED BZADOUGH 2 YEARS AGO



#EDUCATION

How We Can Make Education Accessible Through Community

2 YEARS AGO

Donate

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public. We also have thousands of freeCodeCamp study groups around the world.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

You can make a tax-deductible donation here.

Trending Guides

Git Clone
Agile Methods
Python Main
Callback
Debounce
URL Encode
Blink HTML
Python Tuple
JavaScript Push
Java List
Responsive Web Design
What Is an SVG File?
PDF Password Remover
What Is a PDF?
What Is Python?
UX
Design Thinking
Prime Numbers List
Product Design
Digital Design
Coding Games
SVM
JavaScript forEach
Google BERT
Create Table SQL
What Is TLS?
What Is a LAN?
What is npm?
RSync Examples
Random Forest

Our Nonprofit

About Alumni Network Open Source Shop Support Sponsors Academic Honesty