**Algeasy Text Repository**



**Naveen Arunachalam**

# General Format

Problem

Origin

Uses

Necessary Intuition

The Algorithm

Explanation of Algorithm

Implementation (Python)

Explanation of Implementation (Python)

(Other languages: C++, Java, Matlab)

# Ideas

> “closed form solutions”: 1 + 2 + 3 + 4 … vs n(n+1)/2

> rubiks cube

> towers of Hanoi

> Game of Life

> Sorting algorithms

> API╒s?

> Pi algorithm

> Largest item in a list

> Serial and parallel algorithms

Programming languages

http://www.academia.edu/7824262/Python\_Algorithms\_Mastering\_Basic\_Algorithms\_in\_the\_Python\_Language

# index.html

This website presents a series of articles that intend to teach the ideas behind computer algorithms. My hope is that readers could build off the information presented here to create their own algorithms and implement them in their language of choice. Articles are targeted to the beginner or intermediate student, with a focus on intuition and understanding. Have fun!

# contact.html

Naveen Arun: Site Creator

Naveen Arun is a student who is interested in computers and computer algorithms.

Email: narunach@caltech.edu

# What\_is\_an\_algorithm.html

## Background: What is an algorithm?

At its most basic level, an algorithm is a roadmap for solving a problem. People create algorithms by thinking about ways to approach a problem, such as sorting objects in a row. Then, to allow computers to execute these algorithms, people create programs in various languages, such as Python, C++, and Java. Programs derived from algorithms are the backbone of nearly every computer process, from simple addition to communication via the Internet.

## What makes an algorithm useful?

A useful algorithm solves a general, well-defined problem. For example, let us consider the problem of sorting a list of objects, with the assumption that there is some way of ordering them. There are many instances of this problem - for instance, consider the differences in sorting the list {Albert, Calvin, Bob} versus sorting the list {4, 3, 1, 2}. One involves *strings*, or sequences of characters, while the other involves integers. Nevertheless, the basic idea - sorting a list of objects by some method of ordering - remains constant.

As it turns out, there are many algorithms that solve this problem. One algorithm, which is the most intuitive to humans, is scanning the list, finding the lowest value, and bringing it to the front. Thus, our sorting looks like this:

Step 0: {4, 3, 1, 2} (original list)

Step 1: {1, 4, 3, 2} (move 1 to the front)

Step 2: {1, 2, 4, 3} (move 2 to position after 1)

Step 3: {1, 2, 3, 4} (move 3 to position after 2)

A more efficient algorithm is Merge Sort. We first group the list in sets of twos, then order those sets of twos as sets of fours, and so on. Our sorting looks like this:

Step 0: {4, 3 | 1, 2} (add partition to middle of list)

Step 1: {3, 4 | 1, 2} (sort the contents of each partition)

Step 2: {1, 2, 3, 4} (remove partition and sort the entire list)

Although there are some hidden calculations involved in both examples, one should note that the latter algorithm is good for efficiently sorting large lists, whereas the former can be very inefficient as list sizes grow. The efficiency of Merge Sort comes from the fact that it breaks a large problem into bite-sized pieces, which are then combined into a final product.

Essentially, there are three things that make an algorithm useful:

1. It solves a general, well-defined problem

2. It can be applied to different instances of a problem and still yield a valid solution

3. It can efficiently solve the problem, regardless of scale.

## Algorithms vs. Programs

An algorithm is an idea, whereas a program is a series of commands written out for a computer to execute. Algorithms are what a correctly written program should perform. Consider a simple algorithm for microwaving instant noodles:

Add boiling water

Place in microwave

Remove from microwave

Now consider a computer-style “program” to execute this algorithm:

Obtain (brand name) instant noodles

Obtain water

Boil water

Open lid of instant noodles

Add 3/4 cup boiled water to instant noodles

Open microwave

Close microwave

Set to 60 seconds

Wait 60 seconds

Open microwave

Remove instant noodles

Notice how this example illustrates a few properties of programs and algorithms:

1. An algorithm is an idea that is implemented by a program

2. A program is, by nature, more specific than an algorithm

3. A program is written for a computer, whereas an algorithm is written for humans to understand

By learning and understanding algorithms, humans can create approaches to complex problems in the real world. Then, these approaches can be written in a computer-friendly format in the form of a program, so that computers can carry out the ideas that humans produce.

## How are algorithms implemented?

As mentioned above, programs implement algorithms by converting a problem into machine-executable code. Some popular means of programming and scripting include C++, Python, Java, and Matlab. These languages allow you to type commands according to specific rules, which are then “compiled” into ones and zeroes for computers to understand.

When you write your own algorithms, you may come across an instance where you need to use another algorithm to solve a problem. For example, if you want to sort a two-dimensional array of objects both horizontally and vertically, it may help to break the problem into one-dimensional lists in those directions and sort the lists instead.

Since many algorithms are built on other algorithms, programming languages often let you import or use popular algorithms in your own programs. For instance, in Python, importing the “numpy” package allows operations such as fast list sorting, so that you can save time in writing your own code.

# Algorithms\_and\_complexity.html

Some problems are harder than others. For example, arranging a set of objects in a row is inherently easier than listing out all possible arrangements of objects in a row. In fact, the answer to the first problem is contained in the second problem. (When you list out every arrangement, one of them is going to be the sorted list.) Thus, problems have their own inherent complexity.

Similarly, algorithms created as solutions to problems carry their own inherent complexity. For example, consider two different algorithms for sorting a list:

1. Keep generating random arrangements of the list until the correct one is obtained.
2. Arrange the list as a human would, i.e. scanning the list and bringing lowest values to the front.

Note that the first algorithm is not even guaranteed to terminate - imagine getting the order of a 1000-element list correctly by chance! Nevertheless, for small list sizes, we expect both algorithms to take the same amount of time to complete. For larger list sizes, going with the second option is a no-brainer.

Often, we wish to analyze the complexity of problems and solutions in a numerical way. To do this, we resort to *computational complexity theory*.

* problem size (based on inputs and problem complexity)
* asymptotic behavior
* best, worst, average case complexity

Consider the problem

>Scale of problems

>Computational Complexity

>Big O notation

>Polynomial time

# Big\_O.html

“Big O” notation is a way of quantifying the complexity of an algorithm based on the relationship between input size and completion time. The “O” in “Big O” stands for “order”. Thus, a phrase like “this algorithm is O(n)” would be read as “this algorithm has a Big-O of n”, which translates to “this algorithm is on the order of n”.

But what does the “n” mean? We use “n” to represent the size of an input: it could be number of words in a book to be searched, the number of elements in a list to sort, and so on. The time taken by an order-n algorithm scales linearly with input size – that is, an input three times as large takes three times the amount of time to complete. The time taken by an order-n^2 algorithm scales quadratically with input size – that is, an input size three times as big would result in nine times the time taken.

Perhaps the best way to get a feel for Big O is to walk through a few examples.

## O(1) Algorithms (in other words, O(n^0) algorithms)

Algorithms with a Big O of 1 always take the same time to complete no matter how big the input gets. Consider a function “isEmpty” that operates on strings. Our algorithm works like this:

* Return (output) the value “True” if the string is empty. (i.e. the string is literally ‘’)
* Return the value “False” if the string is non-empty. (i.e. strings like ‘hello’)

Our implementation of this algorithm is as follows:

* Check the first character of the string.
* If the first character is null, return True.
* Otherwise, return False.

Notice how our implementation has nothing to do with the size of the string - we only look at the first character. We could use a string of length 100 or 1000 and the algorithm would take the same time to execute. For this reason, we have an O(n^0) algorithm, since

## O(n) Algorithms

Consider the problem of counting the number of elements in a list. We will use a very simple approach to do this:

* Start at the first element of the list
* Count up from one as you move down to the end of the list.

Our implementation of the algorithm is as follows:

* Define a *counter variable* which starts at 1
* Start at the first element of the list
* Add one to the counter and move down one element of the list until the end of the list is reached.

> O(1)

# P\_vs\_NP.html

Computers take instructions in steps. An algorithm is a series of such steps

Algorithms vs Programs

Order of the steps

Conditionals (if/then/else?)

Errors

Complexity: Understanding Complexity

> Multiple ways to do same thing

Outline:

> ╥Algorithm╙

o basic definition

o etymology

o basic examples

o harder examples