Data Structure & Algorithms

Author: KhoiVN
Github: https://github.com/vnk8071/machine-learning-learning-path

Introduction

How to Solve Problems

Efficiency

We said earlier that this Nanodegree program is about how to write code to **solve problems** and to do so **efficiently**.
In the last section, we looked at some basic aspects of solving problems—but we didn't really think too much about whether our solutions were *efficient*.

Space and **time**

When we refer to the *efficiency* of a program, we aren't just thinking about its speed—we're considering both the **time** it will take to run the program *and* the amount of **space** the program will require in the computer's memory. Often there will be a trade-off between the two, where you can design a program that runs faster by selecting a data structure that takes up more space—or vice versa.

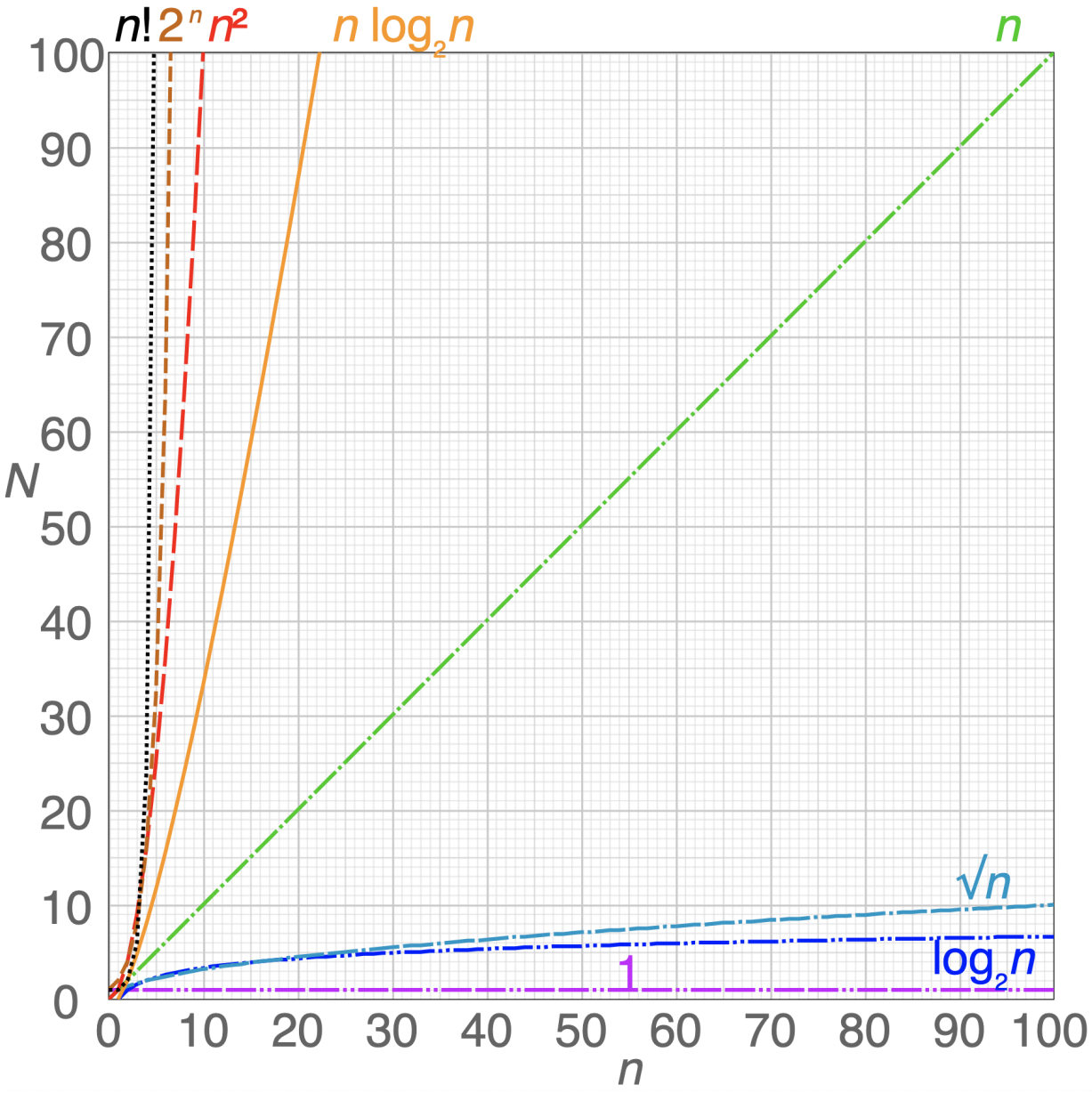
Algorithms

An **algorithm** is essentially *a series of steps for solving a problem*. Usually, an algorithm takes some kind of input (such as an unsorted list) and then produces the desired output (such as a sorted list).
For any given problem, there are usually many different algorithms that will get you to exactly the same end result. But some will be much more efficient than others. To be an effective problem solver, you'll need to develop the ability to look at a problem and identify different algorithms that could be used—and then contrast those algorithms to consider which will be more or less efficient.

But computers are so fast!

Sometimes it seems like computers run programs so quickly that efficiency shouldn't really matter. And in some cases, this is true—one version of a program may take 10 times longer than another, but they both still run so quickly that it has no real impact.

But in other cases, a small difference in how your code is written—or a tiny change in the type of data structure you use—can mean the difference between a program that runs in a fraction of a millisecond and a program that takes hours (or even *years*!) to run.



As the input to an algorithm increases, the time required to run the algorithm may also increase—and different algorithms may increase at different rates.

Reference:

- https://www.bigocheatsheet.com/
- https://wiki.python.org/moin/TimeComplexity

Big O Notation

When describing the efficiency of an algorithm, we could say something like "the run-time of the algorithm increases linearly with the input size". This can get wordy and it also lacks precision. So as an alternative, mathematicians developed a form of notation called **big O notation**. The "O" in the name refers to the *order* of the function or algorithm in question. And that makes sense, because big O notation is used to describe the order—or rate of increase—in the run-time of an algorithm, in terms of the input size (n).

Course 1: Data Structures

Arrays and Linked Lists

Collection

Properties of collections:

- Don't have a particular order (so you can't say "give me the 3rd element in this collection")
- Don't have to have objects of the same type

List

- Have an **order** (so you can say things like "give me the 3rd item in the list")
- Have **no fixed length** (you can add or remove elements)

The distinction between arrays and lists can be a little confusing, especially because of how Python implements the data structure it calls a "list". Below, we'll go over some key points that should make this clearer.

Arrays

An array has some things in common with a list. In both cases:

- There is a collection of items
- The items have an order to them

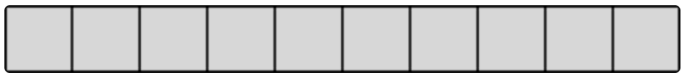
But as Brynn discussed in the video, one of the key differences is that *arrays have indices, while lists do not*.

To understand this, it helps to know how arrays are stored in memory. When an array is created, it is always given some initial size—that is, the number of elements it should be able to hold (and how large each element is). The computer then finds a block of memory and sets aside the space for the array.

Importantly, the space that gets set aside is one, continuous block. That is, all of the elements of the array are *contiguous*, meaning that they are all next to one another in memory.

Another key characteristic of an array is that all of the elements are the same size.

When we represent an array visually, we often draw it as a series of boxes that are all of the same size and all right next to one another:



Because all of the elements are 1) next to one another and 2) the same size, this means that if we know the location of the first element, we can calculate the location of any other element.

For example, if the first element in the array is at memory location `00` and the elements are 24 bytes, then the next element would be at location 00 + 24 = `24`. And the one after that would be at 24 + 24 = `48`, and so on.

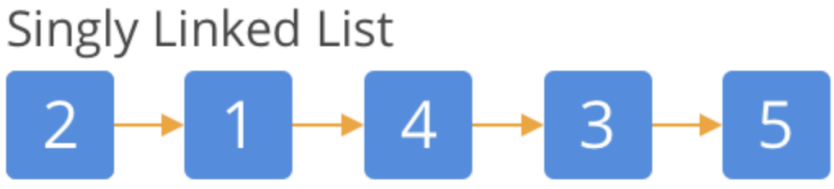
Since we can easily calculate the location of any item in the array, we can assign each item an index and use that index to quickly and directly access the item.

In contrast, the elements of a list may or may not be next to one another in memory! For example, later in this lesson we'll look at *linked lists*, where each list item points to the next list item—but where the items themselves may be scattered in different locations of memory. In this case, knowing the location of the first item in the list does not mean you can simply calculate the location of the other items. This means we cannot use indices to directly access the list items as we would in an array. We'll explore linked lists in more detail shortly.

Linked Lists

1. Singly Linked Lists

In this linked list, each node in the list is connected only to the next node in the list.



This connection is typically implemented by setting the `next` attribute on a node object itself.

2. Doubly Linked Lists

This type of list has connections backwards and forwards through the list.

3. Circular Linked Lists

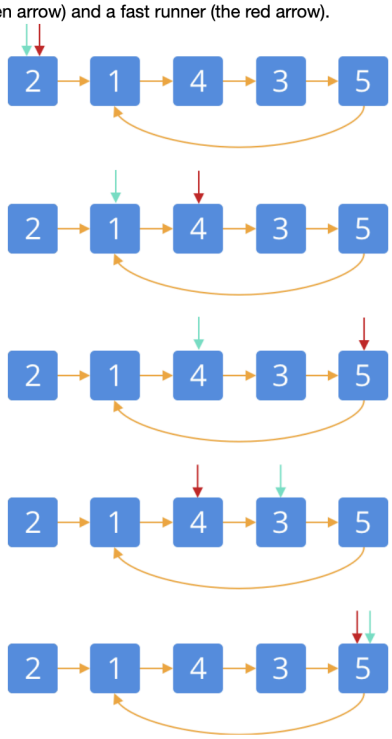
Circular linked lists occur when the chain of nodes links back to itself somewhere. For example `NodeA -> NodeB -> NodeC -> NodeD -> NodeB` is a circular list because `NodeD` points back to `NodeB` creating a loop `NodeB -> NodeC -> NodeD -> NodeB`.

A circular linked list is typically considered pathological because when you try to iterate through it, you'll never find the end. We usually want to detect if there is a loop in our linked lists to avoid these problems. You'll get a chance to implement a solution for detecting loops later in the lesson.

Detecting Loops in Linked Lists

In this notebook, you'll implement a function that detects if a loop exists in a linked list. The way we'll do this is by having two pointers, called "runners", moving through the list at different rates. Typically we have a "slow" runner which moves at one node per step and a "fast" runner that moves at two nodes per step.

If a loop exists in the list, the fast runner will eventually move behind the slow runner as it moves to the beginning of the loop. Eventually it will catch up to the slow runner and both runners will be pointing to the same node at the same time. If this happens then you know there is a loop in the linked list. Below is an example where we have a slow runner (the green arrow) and a fast runner (the red arrow).



Stacks and Queues