Worst case scenarios are rarely the case,

Average vase scenarios are the mostly encountered scenarios

So, which is better: Selection Sort or Insertion Sort? The answer is: well, it

depends. In an average case—where an array is randomly sorted—they perform

similarly. If you have reason to assume you’ll be dealing with data that is

*mostly* sorted, Insertion Sort will be a better choice. If you have reason to

assume you’ll be dealing with data that is mostly sorted in reverse order,

Selection Sort will be faster. If you have no idea what the data will be like,

that’s essentially an average case, and both will be equal.

**HASH TABLE**

Note that hash tables are

called by different names in various programming languages. Other names

include hashes, maps, hash maps, dictionaries, and associative arrays.

Ultimately, a hash table’s efficiency depends on three factors:

• How much data we’re storing in the hash table

• How many cells are available in the hash table

• Which hash function we’re using

A good hash tablem *strikes a balance of avoiding collisions while not consuming lots of memory*

To accomplish this, computer scientists have developed the following rule of

thumb: for every 7 data elements stored in a hash table, it should have 10 cells.

This technique of using a hash table as an “index” comes up frequently in

algorithms that require multiple searches within an array. That is, if your

algorithm will need to keep searching for values inside an array, each search

would itself take up to N steps. By creating a hash table “index” of the array,

we reduce each search to only one step.

\*\*Having a variety of data structures in your programming arsenal also allows you to

create code that is simpler and easier to read.

\*\* Stacks and queues are simply restrictive arrays

**STACKS**

In fact, most computer science literature

refers to the end of the stack as its *top,* and the beginning of the stack

as its *bottom.*

In fact, a stack doesn’t even care about *what* data structure is under the

hood. All it cares about is that there’s a list of data elements that act in a

LIFO way

the stack is

an example of what is known as an *abstract data type*—it’s a kind of data

structure that is a set of theoretical rules that revolve around some other

built-in data structure.

Although a stack is not typically used to store data on a long-term basis, it

can be a great tool to handle temporary data as part of various algorithms.

With a stack, we work with constrained data structure, we can prevent

potential bugs. The linting algorithm, for example, only works if we exclusively

remove items from the top of the stack. If a programmer inadvertently writes

code that removes items from the middle of the array, the algorithm will break

down. By using a stack, we’re forced into only removing items from the top,

as it’s impossible to get the stack to remove any other item.

Undo function – uses stack for popping

Queues

They are applied in queuing printing jobs for example a computer receiving printing requests from different machines on the network. They are also applicable In queuing asynchronous requests and understanding the order in which they are handled.

**RECURSION**

When used correctly,

recursion can be used to solve certain types of tricky problems in

surprisingly simple ways. Sometimes, it even seems like magic. Haha

Reading recursive code

This code can look somewhat confusing at first glance. To walk through the

code to see what it does, here’s the process I recommend:

1. Identify the base case.

2. Walk through the function for the base case.

3. Identify the “next-to-last” case. This is the case just before the base case,

as I’ll demonstrate momentarily.

4. Walk through the function for the “next-to-last” case.

5. Repeat this process by identifying the case before the one you just analyzed,

and walking though the function for that case.

recursion is often a great choice

for an algorithm in which the algorithm needs to dig into an arbitrary number

of levels deep into something.

COMMON SENSE GUIDE

However, we do know that there’s a specific range in which O(N \* M) lies. That

is, if N and M are the same, it’s equivalent to O(N2). And if they’re not the

same, and we arbitrarily assign the smaller number to be M, even if M is as

low as 1, we end up with O(N). In a sense then, O(N \* M) can be construed

as a range between O(N) and O(N2).

Password generator might use an algorithm of O(26)n – Each time we add one element od data, the algorithm doubles in steps

One area in which

recursion shines is where we need to act on a problem that has an arbitrary

number of levels of depth. A second area in which recursion shines is where

it is able to make a calculation based on a subproblem of the problem at hand

computer science literature

refers to the terms bottom up and top down in regard to recursion strategies.

This brings us to the central point of this chapter: recursion shines when

implementing a top-down approach because going top down offers a new

mental strategy for tackling a problem. That is, a recursive top-down approach

allows one to think about a problem in a completely different way

Specifically, when we go top down, we get to mentally “kick the problem down

the road.” We can free our mind from some of the nitty-gritty details we normally

have to think about when going bottom up.

Unnecessary Recursive Calls

In fact, recursion is often the culprit behind

some of the slowest categories of Big O, such as O(2 ^ N).

Max number of array –

One implementation with recursion is of complexity 0 (2 ^ N ) While another also with recursin is O (N), , improvement achieved just by storing the value of a recursive call in avariable rather than having to compute that value again by calling the recursive function.

1. **def max**(array)

*# Base case - if the array has only one element, it is*

*# by definition the greatest number:*

**return** array[0] **if** array.**length** == 1

*# Compare the first element with the greatest element*

*# from the remainder of the array. If the first element*

*# is greater, return it as the greatest number:*

**if** array[0] > max(array[1, array.**length** - 1])

**return** array[0]

*# Otherwise, return the greatest number from the remainder of the array:*

**else**

**return** max(array[1, array.**length** - 1])

**end**

**end**

1. **def max**(array)

**return** array[0] **if** array.**length** == 1

*# Calculate the max of the remainder of the array*

*# and store it inside a variable:*

max\_of\_remainder = max(array[1, array.**length** - 1])

*# Comparison of first number against this variable:*

**if** array[0] > max\_of\_remainder

**return** array[0]

**else**

**return** max\_of\_remainder

**end**

**end**

**\*\*\* Overlapping subproblems arise with recursive Fibonacci algorithm since each sub problem calls the same function that another subproblem calls**

Memoization vs. Bottom-Up

memoization Vs going bottom-up. Is one technique better than the other?

Usually, it depends on the problem and why you’re using recursion in the

first place. If recursion presents an elegant and intuitive solution to a given

problem, you may want to stick with it and use memoization to deal with any

overlapping subproblems. However, if the iterative approach is equally intuitive,

you may want to go with that.

It’s important to point out that even with memoization, recursion does carry

some extra overhead versus iteration. Specifically, with any recursion, the

computer needs to keep track of all the calls in a call stack, which consumes

memory. The memoization itself also requires the use of a hash table, which

will take up additional space on your computer as well.

Generally speaking, going bottom-up is often the better choice unless the

recursive solution is more intuitive. Where recursion is more intuitive, you

can keep the recursion and keep it fast by using memoization.

How does Insertion AND selection sort work again ???