Reffered {

<https://www.linkedin.com/learning/programming-foundations-algorithms/introduction-to-data-structures>

}

Algorithms:

Algorithms are basically processes, or recipes, instructions, whatever you want to call them, that describe how to perform certain tasks. No matter what kind of applications you build, you're bound to come across situations that require the use of one or more algorithms to get the job done

Trying to build a program without understanding algorithms is like trying to build a car without understanding engines. So pop open your favorite text editor and let's get started learning about algorithms

A screenshot of a cell phone

Description automatically generated

A screenshot of a computer

Description automatically generated

An algorithm is a step-by-step procedure or set of instructions used to solve a specific problem.

It involves performing a sequence of operations to transform input data into an output or desired result.

Algorithms have characteristics like space complexity (memory usage) and time complexity (efficiency relative to input size).

They take defined inputs and produce outputs, such as sorting algorithms arranging data into a specific order.

Algorithms can be classified based on criteria like sequential vs. parallel processing, exact vs. approximate solutions, deterministic vs. non-deterministic decision-making.

Common algorithm types:

Searching Algorithms: Used to find specific data within a larger data structure.

Examples: Searching for a substring in a string, finding a file in nested folders.

Sorting Algorithms: Arrange data into a specific order.

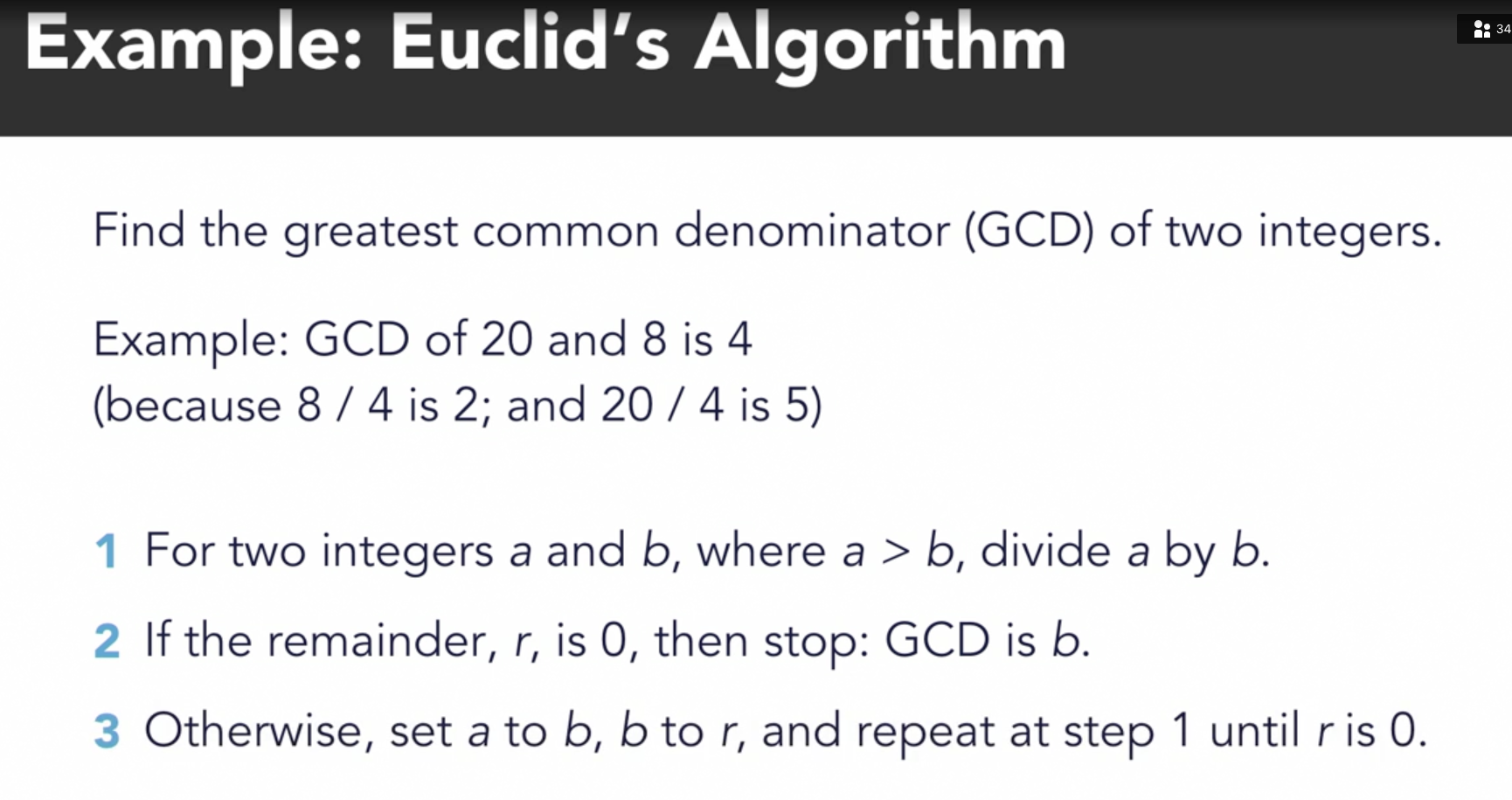
Common task when working with ordered sets of data.

Computational Algorithms: Derive one set of data from another through calculations.

Examples: Checking for prime numbers, converting temperatures between scales.

Collection Algorithms: Manipulate or navigate sets of data stored within a particular structure.

Examples: Counting specific items, filtering unwanted data.



A screenshot of a table

Description automatically generated

# Find the greatest common denominator of two numbers

# using Euclid's algorithm

def gcd(a, b):

while (b != 0):

t = a # set aside the value of a

a = b # set a equal to b

b = t % b # divide t (which is a) by b

return a

# try out the function with a few examples

print(gcd(60, 96)) # should be 12

print(gcd(20, 8)) # should be 4

Measuring Algorithm Performance:

Understanding algorithm performance is crucial for choosing the right algorithm and predicting program behavior. Performance varies with input set size.

Big-O Notation:

Describes algorithm performance concerning input size growth.

Represents the worst-case scenario's time complexity.

The reason the letter O is used is because the growth rate of an algorithm's time complexity is also referred to as the order of operation. It usually describes the worst case scenario of how long it takes to perform a given operation.

Multiple Big-O values can apply to different operations within algorithms or data structures.

Common Big-O Notation Terms:

Constant Time (O(1)):

Operation duration doesn't depend on the data set's size.

Examples: Checking if a number is even or odd, looking up an element in an array by index.

Logarithmic Time (O(log n)):

Typical for operations like finding a value in a sorted array using binary search.

Time grows logarithmically with the data set's size.

Linear Time (O(n)):

Typical for searching in an unsorted array.

Time grows linearly with the data set's size.

Log-Linear Time (O(n log n)):

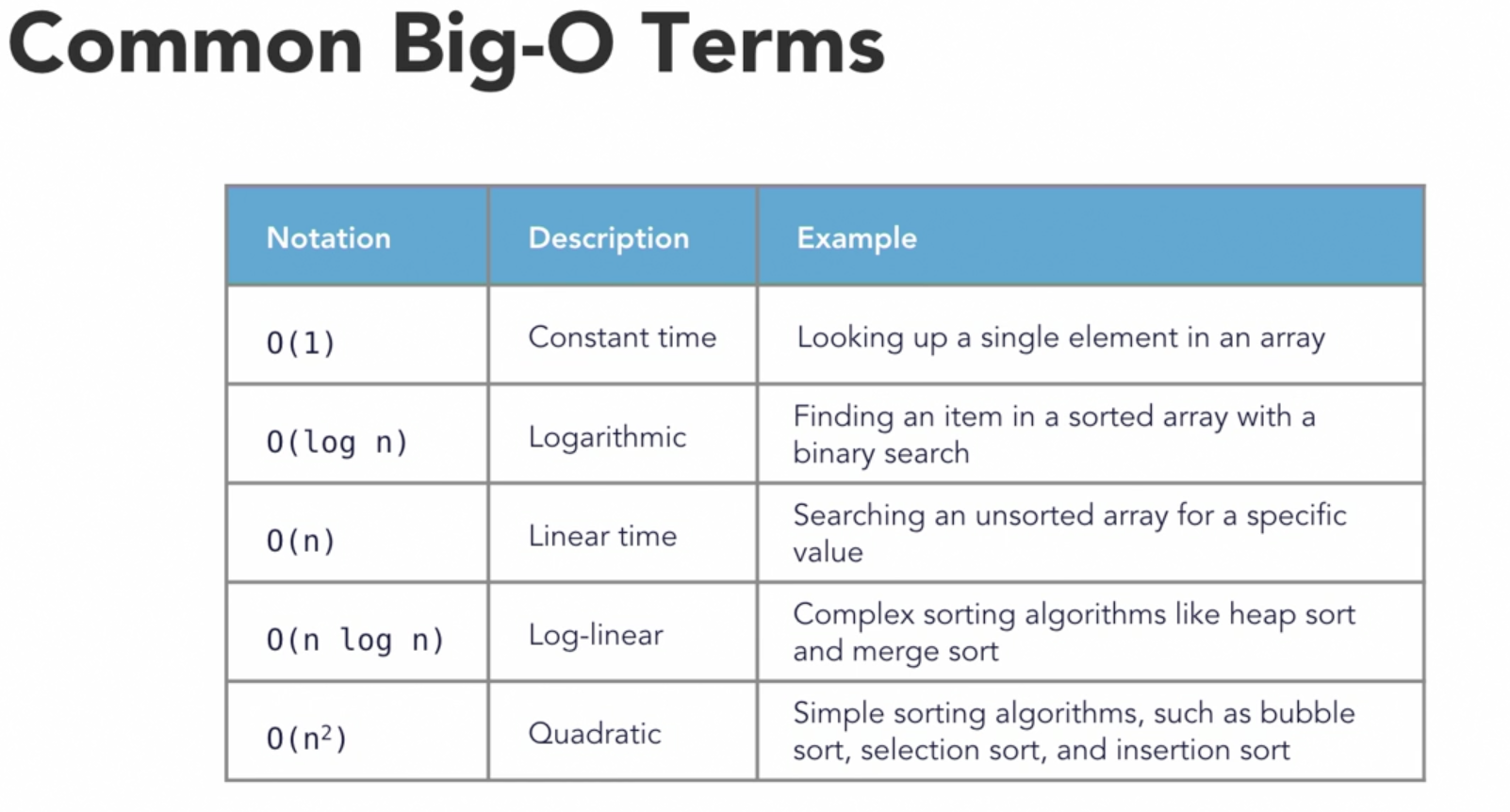
Common in some sorting algorithms like heap sort and merge sort.

Time complexity increases linearly with data set size and logarithmically within each iteration.

Quadratic Time (O(n²)):

Undesirable time complexity where time grows as the square of the data set's size.

Examples include bubble sort and selection sort.



These terms help compare the performance levels of different algorithms and data structures when programming.

Common DataStructures:

Data structures often complement algorithms since most algorithms manipulate data.

Different scenarios require specific data structures to effectively manage and manipulate information.

Data structures organize information for efficient algorithmic operations.

A diagram of data structures

Description automatically generated

Schedulist – array   
Directory structure = Tree

Array Basics:

Arrays are fundamental data structures used in programming.

They consist of a collection of elements, each identified by an index or key value.

One-dimensional arrays are linear sets of values.

Random Access:

Arrays enable direct access to elements using calculated indexes.

Element positions can be computed mathematically, allowing random access.

Zero-Based Indexing:

In most programming languages, array indexes start at zero for the first element.

Multi-Dimensional Arrays:

Arrays can have multiple dimensions, such as two-dimensional arrays.

Accessing elements in multi-dimensional arrays requires specifying two indexes.

Order of Operations:

Calculating an item's index in an array is a constant-time operation.

Inserting or deleting items at the beginning or middle of an array takes linear time complexity (O(n)).

Inserting or deleting items at the end can often be done in constant time.

0 vs 1 based indexing:

Several programming languages use 1-based indexing, which means the index of the first element in an array or list starts at 1 instead of 0. Some of these languages include:

MATLAB

R

Lua

Julia

Fortran

COBOL

Ada

It's important to note that most popular programming languages, such as C, C++, Java, Python, and JavaScript, use 0-based indexing. The choice between 0-based and 1-based indexing is a design decision made by the language creators, and each approach has its advantages and disadvantages.

Advantages of 0-Based Indexing:

Mathematical Consistency: 0-based indexing aligns more closely with mathematical conventions, where the first element is often represented as "element 0." This consistency can make it easier to work with mathematical formulas and algorithms.

Memory Addressing: In many low-level programming languages, such as C and C++, 0-based indexing is used because it aligns with memory addressing and pointer arithmetic. This can lead to more efficient memory management.

Array Bounds: Using 0-based indexing makes it easier to define the bounds of an array. For example, an array with elements at indexes 0 to N-1 has N elements.

Historical Precedence: Many early programming languages, like C and assembly languages, used 0-based indexing. Newer languages often adopt this convention to maintain compatibility with existing code.

Disadvantages of 0-Based Indexing:

Cognitive Load: Some developers find 0-based indexing less intuitive, especially when working with beginners or when dealing with real-world scenarios where elements are naturally counted from 1.

Advantages of 1-Based Indexing:

Human-Friendly: 1-based indexing is often considered more intuitive for humans because it aligns with how people naturally count items. When you say "the first element," it corresponds to index 1.

Historical Precedence: Some early programming languages, like Fortran, used 1-based indexing. This convention is still prevalent in certain domains, such as scientific computing and statistics.

Readability: 1-based indexing can sometimes lead to more readable code, especially when dealing with data that is naturally counted from 1, such as rows and columns in a table.

Disadvantages of 1-Based Indexing:

Mathematical Inconvenience: 1-based indexing can be less convenient when working with mathematical formulas and algorithms that expect 0-based indexing. This can lead to off-by-one errors.

Inefficiency: Some low-level programming tasks, like memory addressing, may become less efficient with 1-based indexing because it doesn't align with the hardware's natural addressing scheme.

Array Bounds: Defining array bounds with 1-based indexing can sometimes be less straightforward. It may require extra calculations to determine the size of an array.

In summary, the choice between 0-based and 1-based indexing depends on factors like historical conventions, language design goals, and the domain in which the language is intended to be used. While 0-based indexing offers some advantages in terms of mathematical consistency and memory management, 1-based indexing is often seen as more human-friendly and intuitive for certain applications. Ultimately, the choice of indexing approach is a trade-off, and different languages make different decisions based on their intended use cases.

Linked List:

A linked list is a linear collection of data elements called nodes, where each node contains some information and a reference to the next node in the list.

The first node in a linked list is known as the head, and the last node points to nothing, indicating the end of the list.

Singly linked lists have one-directional links, with each node pointing to its next neighbor. Doubly linked lists, on the other hand, have links to both the previous and next neighbors.

Linked lists are advantageous for quick insertion and removal of items, as they don't require reorganizing memory like arrays do.

However, linked lists lack constant-time random access to elements, which is a drawback compared to arrays.

To find a specific item in a linked list, you must traverse the list from the head until you reach the desired item, resulting in linear time complexity.

Inserting a new item at the head of a linked list involves setting the new node's next pointer to the current head and updating the head to the new node.

Removing a node from a linked list requires changing the next field of the preceding node to point to the following node, effectively skipping the node to be removed.

# Linked list in py

# the Node class

class Node(object):

def \_\_init\_\_(self, val):

self.val = val

self.next = None

def get\_data(self):

return self.val

def set\_data(self, val):

self.val = val

def get\_next(self):

return self.next

def set\_next(self, next):

self.next = next

# the LinkedList class

class LinkedList(object):

def \_\_init\_\_(self, head=None):

self.head = head

self.count = 0

def get\_count(self):

return self.count

# insert at Head

def insert(self, data):

new\_node = Node(data)

new\_node.set\_next(self.head)

self.head = new\_node

self.count += 1

def find(self, val):

item = self.head

while (item != None):

if item.get\_data() == val:

return item

else:

item = item.get\_next()

return None

def deleteAt(self, idx):

if idx > self.count:

return

if self.head == None:

return

else:

tempIdx = 0

node = self.head

# stop upto before index

while tempIdx < idx-1:

node = node.get\_next()

tempIdx += 1

# copy next to next item pointer

# skip index item link in before index node

node.set\_next(node.get\_next().get\_next())

self.count -= 1

# print all elements in list

def dump\_list(self):

tempnode = self.head

while (tempnode != None):

print("Node: ", tempnode.get\_data())

tempnode = tempnode.get\_next()

# create a linked list and insert some items

itemlist = LinkedList()

itemlist.insert(38)

itemlist.insert(49)

itemlist.insert(13)

itemlist.insert(15)

itemlist.dump\_list()

'''

Node: 15

Node: 13

Node: 49

Node: 38

'''

# exercise the list

print("Item count: ", itemlist.get\_count())#=> 4

print("Finding item: ", itemlist.find(13)) #=> <\_\_main\_\_.Node object at 0x7f9bfae500a0>

print("Finding item: ", itemlist.find(78)) #=> None

# delete an item

itemlist.deleteAt(3)

print("Item count: ", itemlist.get\_count()) #=> 3

print("Finding item: ", itemlist.find(38)) #=> None

itemlist.dump\_list()

'''

Node: 15

Node: 13

Node: 49

'''

Stacks:

Stacks are data structures supporting two primary operations: push and pop.

They are last-in, first-out (LIFO) structures, meaning the last item pushed onto the stack is the first one to be popped.

Pushing an item onto a stack is a constant-time operation.

Popping an item off the stack is also a constant-time operation.

Stacks are used for parsing and evaluating expressions, like reverse Polish notation.

They enable backtracking, as seen in web browsers' back button functionality.

Queues:

Queues are data structures supporting two primary operations: enqueue and dequeue.

They are first-in, first-out (FIFO) structures, meaning the first item added to the queue is the first one to be removed.

Enqueuing and dequeuing items are both constant-time operations.

Queues are used in order processing to fulfill orders in the sequence they are received.

They are employed in message processing to ensure messages are sent in the order they are created, such as in SMS messenger apps.

Implementing a Stack with Constant-Time Push and Pop:

Dynamic Array-Based Stack:

Use a dynamic array (or ArrayList) to store the elements of the stack.

When you push an element onto the stack, add it to the end of the dynamic array (amortized constant time).

When you pop an element from the stack, remove it from the end of the dynamic array (constant time).

Dynamic arrays can automatically resize themselves when needed, so they provide constant-time amortized performance for push and pop operations.

Implementing a Queue with Constant-Time Enqueue and Dequeue:

Doubly Linked List-Based Queue:

Use a doubly linked list to represent the queue.

Maintain two pointers: one pointing to the front (head) of the list and another pointing to the rear (tail) of the list.

When you enqueue an element, add it to the rear of the linked list (constant time), update the tail pointer.

When you dequeue an element, remove it from the front of the linked list (constant time), update the head pointer.

The linked list provides constant-time insertions and removals from both ends (front and rear).

# stack functions in py

# create a new empty stack

stack = []

# push items onto the stack

stack.append(1)

stack.append(2)

stack.append(3)

stack.append(4)

# print the stack contents

print(stack)#=> [1,2,3,4]

# pop an item off the stack

x = stack.pop()

print(x)#=> 4

print(stack)#=> [1,2,3]

# queue functions in py

from collections import deque

# create a new empty deque object that will function as a queue

queue = deque()

# add some items to the queue

queue.append(1)

queue.append(2)

queue.append(3)

queue.append(4)

# print the queue contents

print(queue)#=>deque([1, 2, 3, 4])

# pop an item off the front of the queue (dequeue)

x = queue.popleft()

print(x)#=>1

print(queue)#=>deque([2, 3, 4])

Hash table:

A hash table is a type of associative array that maps keys to their associated values using a hash function.

Ideally, a hash function assigns each key to a unique slot in the hash table.

Collisions can occur when multiple keys map to the same slot, and hash tables must handle these collisions.

Advantages of hash tables include unique key-to-value mapping, speed (especially for large datasets), and the ability to implement algorithms like counters and filters.

Drawbacks include inefficiency for small datasets, lack of ordering, and possible challenges in enumerating entries close to a given key.

In most programming languages, hash table data structures are readily available and handle hashing and collision resolution internally.

Hash tables can be created using dictionary constructors or by adding items progressively.

Attempting to access a non-existent key will result in an error.

Assigning a value to an existing key in the hash table will replace the existing value.

Hash tables can be iterated using language-specific methods like items() in Python.

# hashtable usage in py

# create a hashtable all at once

items1 = dict({"key1": 1, "key2": 2, "key3": "three"})

print(items1) #=> {'key1': 1, 'key2': 2, 'key3': 'three'}

# create a hashtable progressively

items2 = {}

items2["key1"] = 1

items2["key2"] = 2

items2["key3"] = 3

print(items2) #=> {'key1': 1, 'key2': 2, 'key3': 3}

# try to access a nonexistent key

# print(items1["key6"])

'''

Traceback (most recent call last):

File "/home/jdoodle.py", line 18, in <module>

print(items1["key6"])

KeyError: 'key6'

'''

# replace an item

items2["key2"] = "two"

print(items2) #=> {'key1': 1, 'key2': 'two', 'key3': 3}

# iterate the keys and values in the dictionary

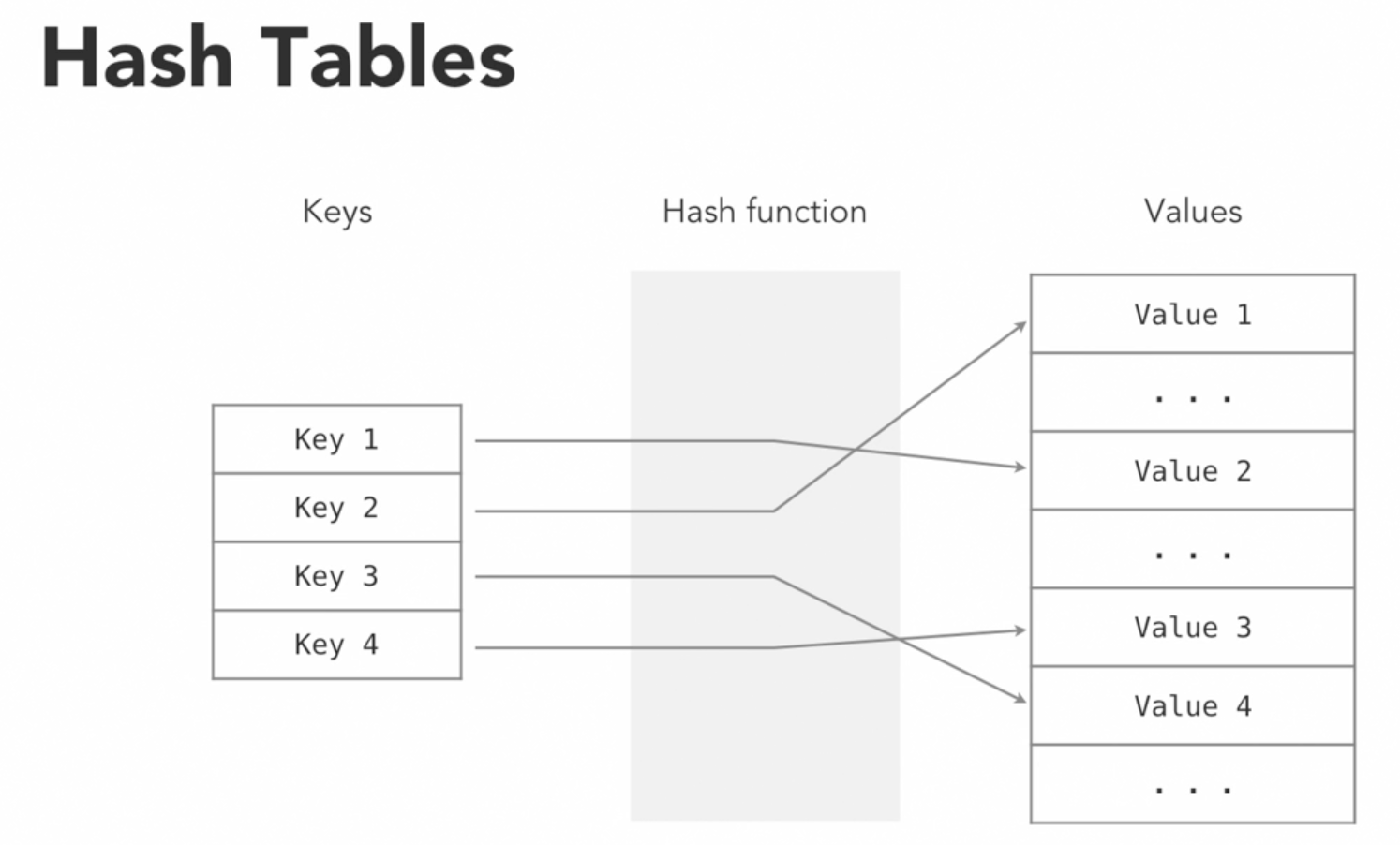
for key, value in items2.items():

print("key: ", key, " value: ", value)

# key: key1 value: 1

# key: key2 value: two

# key: key3 value: 3



Recursion:

Recursion is a programming technique where a function calls itself within its own code.

Recursion is useful in scenarios like searching for files in nested directories.

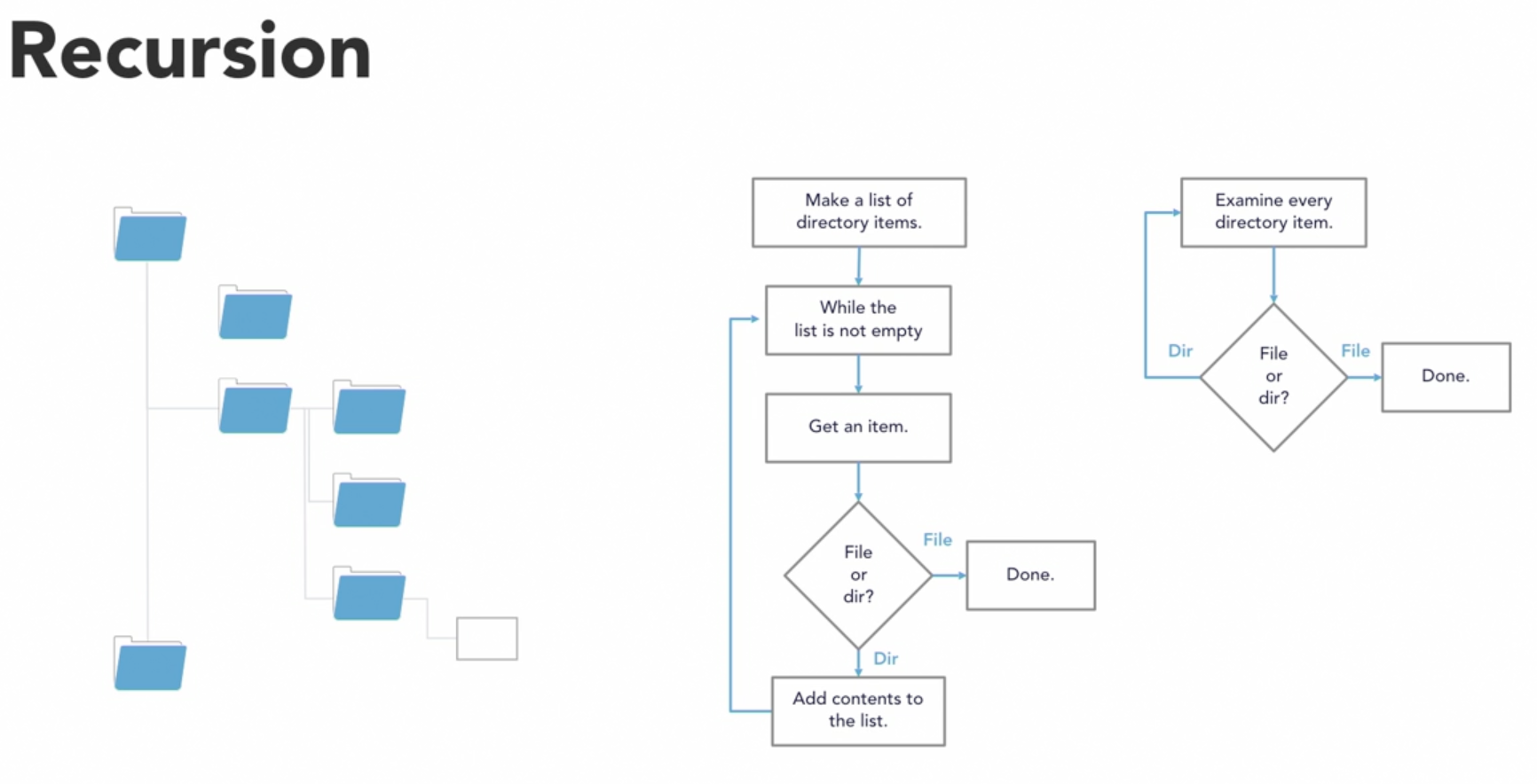
It helps simplify complex problems by breaking them down into smaller, more manageable parts.

It's important to have a breaking condition in a recursive function to ensure it stops calling itself and doesn't create an infinite loop. ( run out of memory to maintain all function calls )

The call stack is a data structure that keeps track of function calls during recursion.

Each recursive call stores its own set of argument values in the call stack.

When the breaking condition is met, the function returns, and all previous function calls return in reverse order, unwinding the call stack.



A table with numbers and symbols

Description automatically generated

*# use recursion to implement a countdown counter in py*

**def** countdown(x):

*# breaking condition*

if x == 0:

print("Done!")

*# exit function*

print('call stack unwounding', x)

return

else:

print(x, "...")

*# recursive coundition with different input*

countdown(x-1)

print('call stack unwounding', x)

countdown(5)

''' =>

5 ...

4 ...

3 ...

2 ...

1 ...

Done!

call stack unwounding 0

call stack unwounding 1

call stack unwounding 2

call stack unwounding 3

call stack unwounding 4

call stack unwounding 5

'''

*# Using recursion to implement power and factorial functions*

*# power - multiplied by the same number to certain times*

*# 2^4 = 2\*2\*2\*2 = 16*

**def** power(num, pwr):

*# breaking condition: if we reach zero, return 1*

if pwr == 0:

*# 1 in product doesn't affect results*

return 1

else:

return num \* power(num, pwr-1)

*# fact - product of all natural numbers upto given number*

*#5! = 5\*4\*3\*2\*1 = 120*

**def** factorial(num):

if (num == 0):

*# 0! = 1 (edge case scenario)*

return 1

else:

return num \* factorial(num-1)

print("{} to the power of {} is {}".format(5, 3, power(5, 3)))

*#=> 5 to the power of 3 is 125*

print("{} to the power of {} is {}".format(1, 5, power(1, 5)))

*#=> 1 to the power of 5 is 1*

print("{}! is {}".format(4, factorial(4)))

*#=> 4! is 24*

print("{}! is {}".format(0, factorial(0)))

*#=> 0! is 1*

Sorting :

Sorting is a common task in programming where you arrange a set of data in a specific order. This is important for both user experience and program efficiency.

Why Sorting Matters:

User-Friendly Display: Sorting helps users make sense of information. For instance, in a real estate app, users might want to see properties sorted by price.

Efficient Processing: Sorting can make programs work faster. For the same real estate app, if a user sets a maximum price, the app can sort data internally to reduce the need for extra information retrieval.

Sorting Algorithms: We'll learn about different ways to sort data, including:

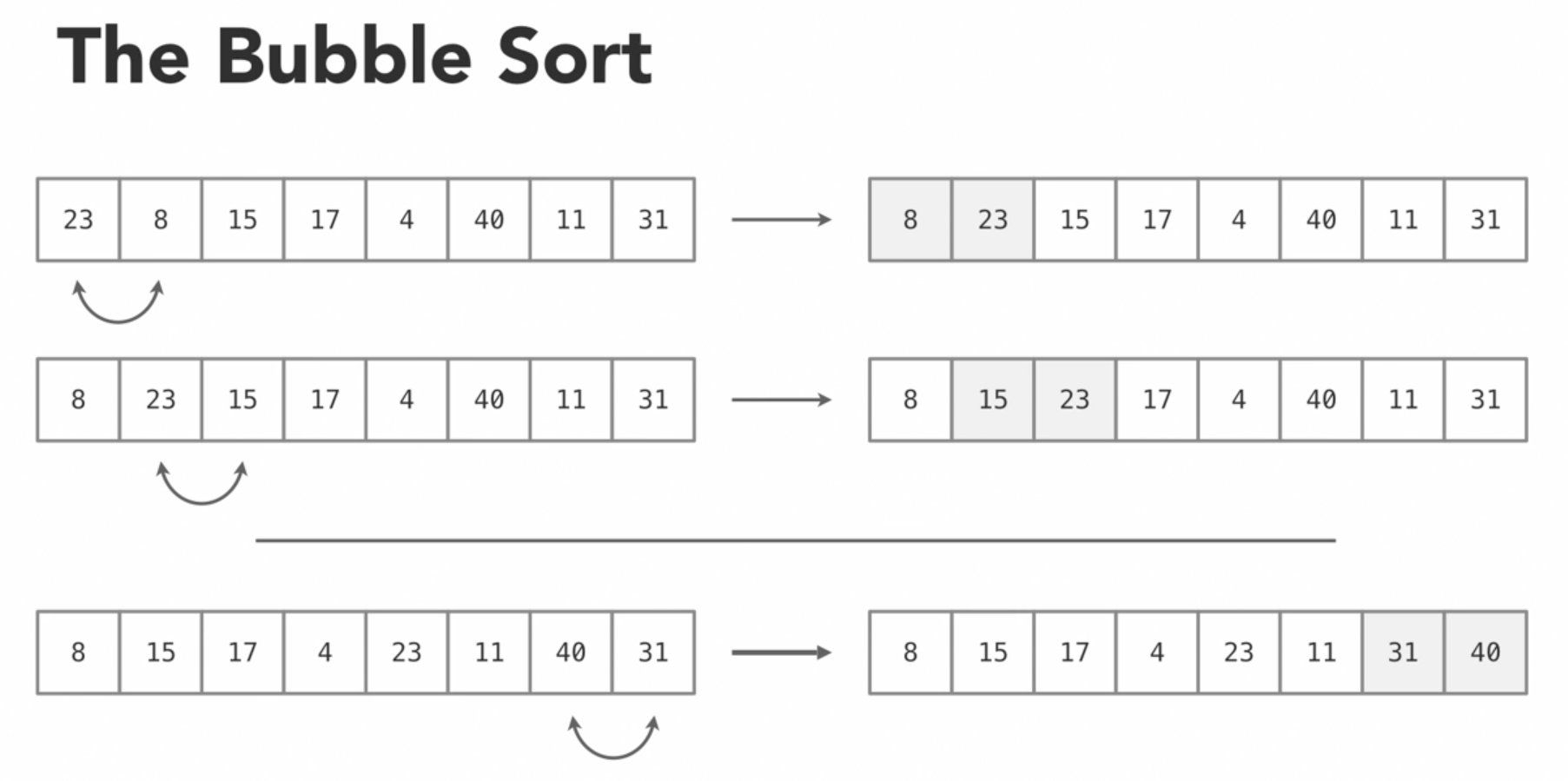
Bubble Sort: A basic algorithm often used for teaching purposes, but not efficient for real-world use.

Merge Sort: Uses recursion to sort data and is more efficient than bubble sort.

Quick Sort: Also uses recursion and is usually faster than merge sort.

These sorting algorithms demonstrate various approaches to solving the sorting problem. While modern programming languages offer built-in sorting functions, understanding these algorithms helps you choose the right one for your needs.

Bubble sort :



Bubble Sort is a basic sorting algorithm. Easy to implement.

It repeatedly compares and swaps adjacent elements if they are in the wrong order.

The largest element "bubbles up" to its correct position in each pass.

It has a time complexity of O(n^2), making it inefficient for large datasets.

Bubble Sort is mainly used for teaching and not recommended for practical use.

Other sorting algorithms like Quick Sort and Merge Sort are more efficient for large datasets.

# Bubble sort algorithm in py

*def* bubbleSort(*dataset*):

# start with the array length and decrement each time

for i in range(len(dataset)-1, 0, -1):

# examine each item pair

for j in range(i):

# swap items if needed

if dataset[j] > dataset[j+1]:

temp = dataset[j]

dataset[j] = dataset[j+1]

dataset[j+1] = temp

print("Current state: ",i, dataset)

*def* main():

list1 = [6, 20, 8, 19, 56, 23, 87, 41, 49, 53]

print("Starting state: ", list1)

bubbleSort(list1)

print("Final state: ", list1)

if \_\_name\_\_ == "\_\_main\_\_":

main()

'''

Starting state: [6, 20, 8, 19, 56, 23, 87, 41, 49, 53]

Current state: 9 [6, 8, 19, 20, 23, 56, 41, 49, 53, 87]

Current state: 8 [6, 8, 19, 20, 23, 41, 49, 53, 56, 87]

Current state: 7 [6, 8, 19, 20, 23, 41, 49, 53, 56, 87]

Current state: 6 [6, 8, 19, 20, 23, 41, 49, 53, 56, 87]

Current state: 5 [6, 8, 19, 20, 23, 41, 49, 53, 56, 87]

Current state: 4 [6, 8, 19, 20, 23, 41, 49, 53, 56, 87]

Current state: 3 [6, 8, 19, 20, 23, 41, 49, 53, 56, 87]

Current state: 2 [6, 8, 19, 20, 23, 41, 49, 53, 56, 87]

Current state: 1 [6, 8, 19, 20, 23, 41, 49, 53, 56, 87]

Final state: [6, 8, 19, 20, 23, 41, 49, 53, 56, 87]

'''

Merge Sort:

Merge Sort is a sorting algorithm that uses a divide-and-conquer approach.

It divides a large dataset into smaller parts and sorts them.

It takes a given set of data and then breaks it down into smaller parts that are easier to work with. It uses recursion to break the data down and then sort the smaller sets of data, gradually working it's way back up to the original dataset.

The key to understanding it is knowing how to merge two sorted arrays into one.

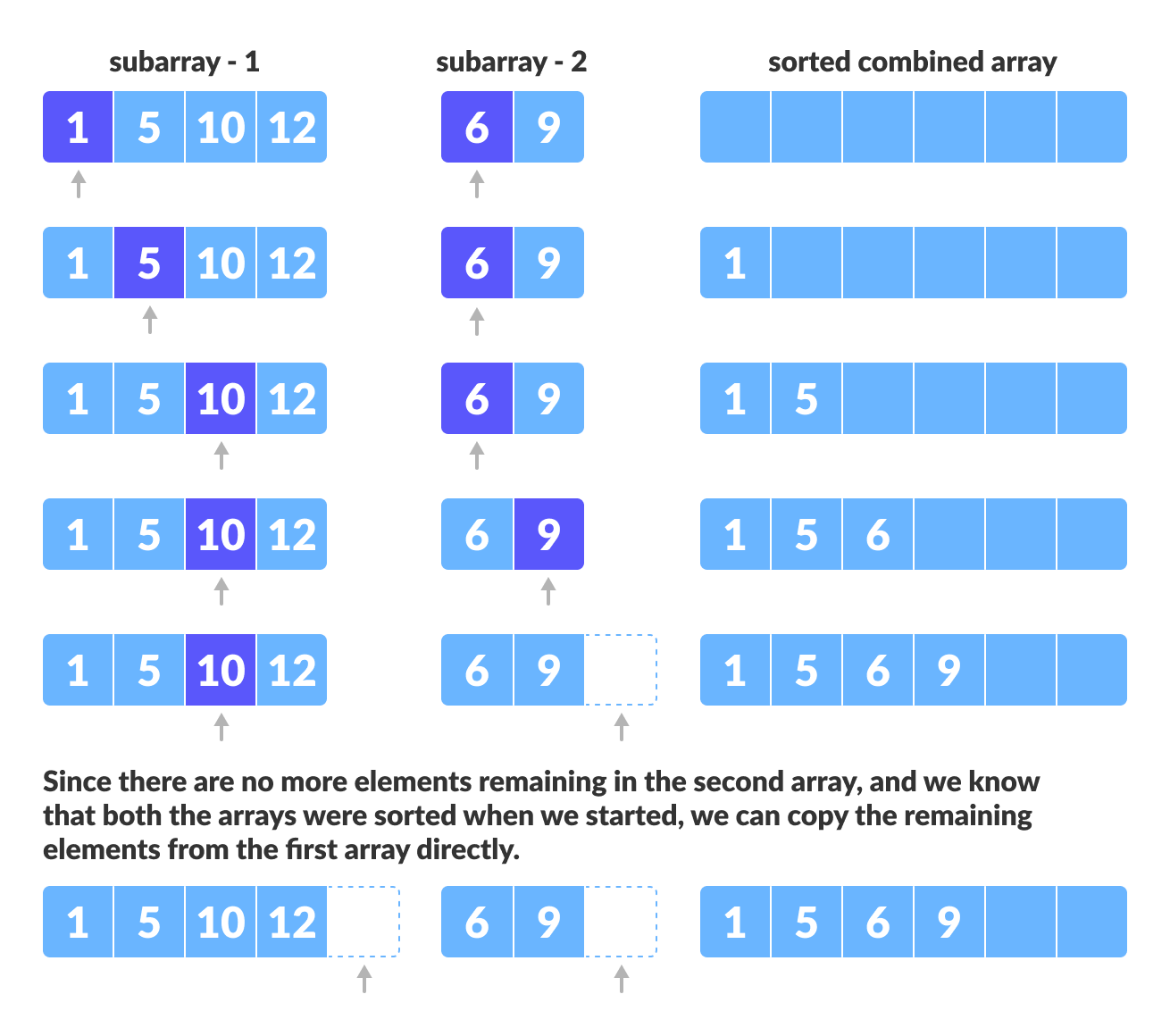
Merging two sorted arrays involves comparing elements and arranging them in sorted order.

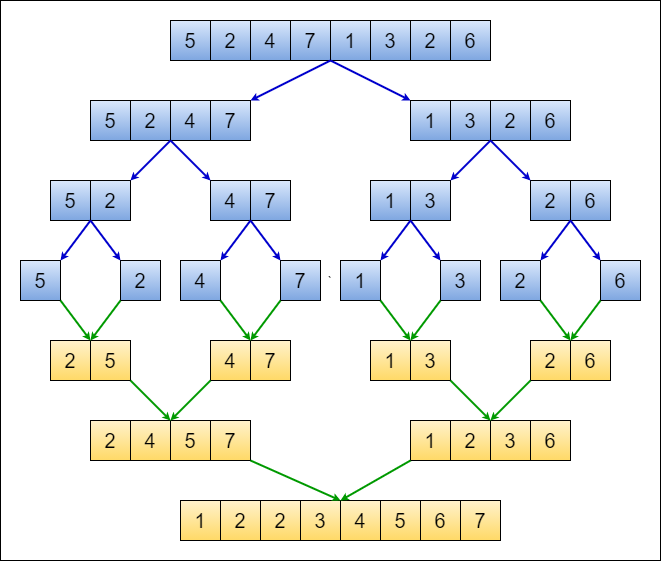
Merge Sort has a good performance profile, operating in O(n log n) time, which is much more efficient than Bubble Sort.

It repeatedly divides the dataset until it's composed of single-element arrays, which are inherently sorted.

It then merges these smaller sorted arrays back together to create a fully sorted dataset.

Merge Sort is efficient and suitable for practical use, especially for large datasets.





# Implement a merge sort with recursion in py

items = [6, 20, 8, 19, 56, 23, 87, 41, 49, 53]

*def* mergesort(*dataset*):

if len(dataset) > 1:

mid = len(dataset) // 2

leftarr = dataset[:mid]

rightarr = dataset[mid:]

# recursively break down the arrays

mergesort(leftarr)

mergesort(rightarr)

# now perform the merging

i=0 # index into the left array

j=0 # index into the right array

k=0 # index into merged array

# while both arrays have content

while i < len(leftarr) and j < len(rightarr):

if leftarr[i] < rightarr[j]:

dataset[k] = leftarr[i]

i += 1

else:

dataset[k] = rightarr[j]

j += 1

k += 1

# if the left array still has values, add them

while i < len(leftarr):

dataset[k] = leftarr[i]

i += 1

k += 1

# if the right array still has values, add them

while j < len(rightarr):

dataset[k] = rightarr[j]

j += 1

k += 1

# test the merge sort with data

print(items) #=> [6, 20, 8, 19, 56, 23, 87, 41, 49, 53]

mergesort(items)

print(items) #=> [6, 8, 19, 20, 23, 41, 49, 53, 56, 87]

Quick sort

Quicksort is another sorting algorithm that, like Merge Sort, uses a divide-and-conquer approach and recursion.

It's generally faster than Merge Sort and sorts data in place, without requiring extra memory.

Quicksort selects a Pivot Point, which is usually the first element in the array.

It then partitions the array into two sections: elements less than the pivot and elements greater than the pivot.

This partitioning process continues recursively until each section only contains one element.

While partitioning, two indexes (lower and upper) are used to navigate through the array.

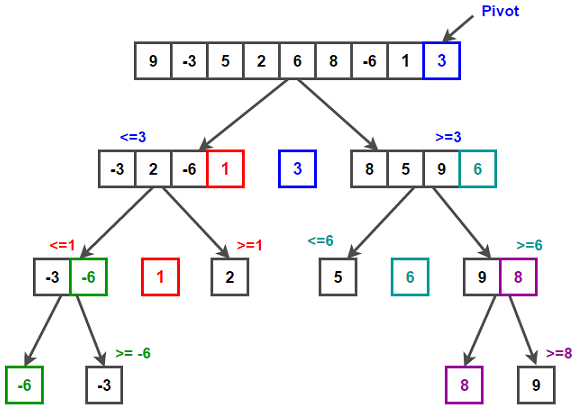
Elements are swapped as needed to place them on the correct side of the pivot.

The pivot itself is moved to its final sorted position.

In the end, the array is sorted.

Quicksort has an average time complexity of O(n log n) (generally), but it can degrade to O(n^2) in the worst case ( already sorted ) .

The choice of pivot and the partitioning process are crucial for its efficiency.



# Implement a quicksort .py

items = [20, 6, 8, 53, 56, 23, 87, 41, 49, 19]

*def* quickSort(*dataset*, *first*, *last*):

if first < last:

# calculate the split point

pivotIdx = partition(dataset, first, last)

# now sort the two partitions

quickSort(dataset, first, pivotIdx-1)

quickSort(dataset, pivotIdx+1, last)

*def* partition(*datavalues*, *first*, *last*):

# choose the first item as the pivot value

pivotvalue = datavalues[first]

# establish the upper and lower indexes

lower = first + 1

upper = last

# start searching for the crossing point

done = False

while not done:

# advance the lower index

while lower <= upper and datavalues[lower] <= pivotvalue:

lower += 1

# advance the upper index

while upper >= lower and datavalues[upper] >= pivotvalue:

upper -= 1

# if the two indexes cross, we have found the split point

if upper < lower:

done = True

else:

# exchange the two values

temp = datavalues[lower]

datavalues[lower] = datavalues[upper]

datavalues[upper] = temp

# when the split point is found, exchange the pivot value

temp = datavalues[first]

datavalues[first] = datavalues[upper]

datavalues[upper] = temp

# return the split point index

return upper

# test the quick sort with data

print(items) #=> [20, 6, 8, 53, 56, 23, 87, 41, 49, 19]

quickSort(items, 0, len(items)-1)

print(items) #=> [6, 8, 19, 20, 23, 41, 49, 53, 56, 87]

Search:

Linear: ( Unordered List Search )

Unordered List Search is about finding an item in a list where the items are not in any specific order.

You have to go through the entire list, item by item, to find what you're looking for.

This method can be inefficient, especially as the list grows, as it has a linear time complexity, which means the time it takes to search increases with the number of items in the list.

If you have a large list, this can result in a lot of comparisons and slow down your program.

There are more efficient ways to search if you know the data is sorted, which is Binary Search…

# Linear search .py

# declare a list of values to operate on

items = [6, 20, 8, 19, 56, 23, 87, 41, 49, 53]

*def* find\_item(*item*, *itemlist*):

for i in range(0, len(itemlist)):

if item == itemlist[i]:

return i

return None

print(find\_item(87, items)) #=> 6

print(find\_item(250, items))#=> None

Binary: ( Ordered List search )

Ordered List Search is a much more efficient way of finding an item in a sorted list compared to an unordered list.

With a sorted list, we can use a method called binary search.

Binary search starts with two indexes, one at the beginning and one at the end of the list.

It calculates the midpoint and checks if the item we're looking for is at that midpoint.

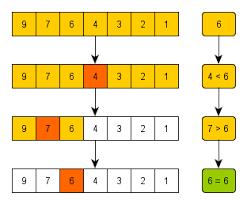
If not, it decides whether to move the upper or lower index based on whether the item is greater or less than the midpoint value.

This process repeats until the item is found, or the indexes cross each other, indicating that the item is not in the list.

Binary search is very efficient; it executes in logarithmic time, meaning that as the list size doubles, you only need a few extra operations.

However, it's important to note that you should consider whether the benefits of keeping the list sorted outweigh the cost of constantly sorting it, especially for large or frequently changing lists.

There's no one-size-fits-all answer, and you'll need to make decisions based on your specific use case and program requirements.



# binary search .py

items = [6, 8, 19, 20, 23, 41, 49, 53, 56, 87]

*def* binarysearch(*item*, *itemlist*):

# get the list size

listsize = len(itemlist) - 1

# start at the two ends of the list

lowerIdx = 0

upperIdx = listsize

while lowerIdx <= upperIdx:

# calculate the middle point

midPt = (lowerIdx + upperIdx)// 2

# if item is found, return the index

if itemlist[midPt] == item:

return midPt

# otherwise get the next midpoint

if item > itemlist[midPt]:

lowerIdx = midPt + 1

else:

upperIdx = midPt - 1

if lowerIdx > upperIdx:

return None

print(binarysearch(23, items)) #=> 4

print(binarysearch(87, items)) #=> 9

print(binarysearch(250, items))#=> None

Is list sorted? :

# Is list sorted .py

items1 = [6, 8, 19, 20, 23, 41, 49, 53, 56, 87]

items2 = [6, 20, 8, 19, 56, 23, 87, 41, 49, 53]

*def* is\_sorted(*itemlist*):

# using the all function

return all(itemlist[i] <= itemlist[i+1] for i in range(len(itemlist)-1))

# using the brute force method

# for i in range(0, len(itemlist)-1):

# if (itemlist[i] > itemlist[i+1]):

# return False

# return True

print(is\_sorted(items1)) #=> True

print(is\_sorted(items2)) #=? False