SEARCHING ALGORITHMS

1. Linear search –sequential search algorithm

Algorithm:

1. Start from first element of the array
2. Compare that element with the target
3. If it equal, return the index
4. Else, move to the next
5. Repeat these steps till the end of the array
6. If no target is located, return -1

Code –

class LinearSearch:

def search(self, arr, target):

for i in range(len(arr)):

if arr[i] == target:

return i

return -1

arr = [3,5,6,2,7]

target = 2

linear\_search = LinearSearch()

index = linear\_search.search(arr, target)

if index != -1:

print(f"Target {target} found at index {index}")

else:

print(f"Target {target} not found in the array")

1. Binary search – repeatatively divides the sorted array into halves

Algorithm –

1. Set the lowerbound as the first element and upperbound as last element of the array.
2. Calculate midpoint using (lb+ub)/2
3. If midpoint = target, return the index of the midpoint
4. If midpoint < target, set lb as midpoint + 1
5. If midpoint > target set ub as midpoint – 1
6. Return -1 if no target value is found

Code –

class BinarySearch:

def search(self, arr, target):

left = 0

right = len(arr) - 1

while left <= right:

mid = (left + right) // 2

if arr[mid] == target:

return mid

elif arr[mid] < target:

left = mid + 1

else:

right = mid - 1

return -1

arr = [1, 3, 5, 7, 9]

target = 5

binary\_search = BinarySearch()

index = binary\_search.search(arr, target)

if index != -1:

print(f"Target {target} found at index {index}")

else:

print(f"Target {target} not found in the array")

1. Breadth First Search –

Algorithm –

1. Create a visited set and a queue
2. Add start vertex to queue
3. Run a loop while queue is not empty, pop a vertex from the queue
4. If the vertex has not been visited, mark it as visited and print it, add its unvisited neighbours to the queue
5. Repeat till queue is empty

Code –

from collections import deque

class Graph:

def \_\_init\_\_(self, graph\_dict=None):

if graph\_dict is None:

graph\_dict = {}

self.graph\_dict = graph\_dict

def add\_edge(self, vertex, edges):

if vertex not in self.graph\_dict:

self.graph\_dict[vertex] = []

self.graph\_dict[vertex].extend(edges)

def bfs(self, start\_vertex):

visited = set()

queue = deque([start\_vertex])

while queue:

vertex = queue.popleft()

if vertex not in visited:

visited.add(vertex)

print(vertex)

queue.extend(self.graph\_dict[vertex] - visited)

graph\_dict = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

graph = Graph(graph\_dict)

graph.bfs('A')

1. Depth First Search –

Algorithm –

1. Create a visited set
2. Call a dfs method with the start vertex and visited set as arguments
3. Mark the current vertex as visited, print it, visit the unvisited vertices recursively
4. Repeat till all vertices are visited

Code –

class Graph:

def \_\_init\_\_(self, graph\_dict=None):

if graph\_dict is None:

graph\_dict = {}

self.graph\_dict = graph\_dict

def add\_edge(self, vertex, edges):

if vertex not in self.graph\_dict:

self.graph\_dict[vertex] = []

self.graph\_dict[vertex].extend(edges)

def dfs(self, start\_vertex):

visited = set()

self.\_dfs(start\_vertex, visited)

def \_dfs(self, vertex, visited):

visited.add(vertex)

print(vertex)

for neighbor in self.graph\_dict[vertex]:

if neighbor not in visited:

self.\_dfs(neighbor, visited)

graph\_dict = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

graph = Graph(graph\_dict)

graph.dfs('A')

SEARCHING ALGORITHMS

1. Bubble sort – sorts by repeatedly swapping the adjacent elements if they are in wrong order.

Algorithm –

1. Run a loop through each element of the unsorted array
2. For each element, run through all elements again, starting from the second element
3. If the current element < previous element, swap
4. Continue till all swaps are done
5. Return sorted array

Code –

class BubbleSort:

def sort(self, arr):

n = len(arr)

for i in range(n):

for j in range(n-i-1):

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

arr[j], arr[j+1] = arr[j+1], arr[j]

return arr

arr = [5, 3, 8, 4, 2]

bubble\_sort = BubbleSort()

sorted\_arr = bubble\_sort.sort(arr)

print(f"Original array: {arr}")

print(f"Sorted array: {sorted\_arr}")

1. Selection sort – finds smallest element and swaps it with the current element

Algorithm –

1. run loop through the elements of the array
2. find the smallest for each element
3. swap it with the smallest element found
4. continue till array is sorted
5. return sorted array

Code –

class SelectionSort:

def sort(self, arr):

n = len(arr)

for i in range(n):

min\_index = i

for j in range(i+1, n):

if arr[j] < arr[min\_index]:

min\_index = j

arr[i], arr[min\_index] = arr[min\_index], arr[i]

return arr

arr = [5, 3, 8, 4, 2]

selection\_sort = SelectionSort()

sorted\_arr = selection\_sort.sort(arr)

print(f"Original array: {arr}")

print(f"Sorted array: {sorted\_arr}")

1. Merge sort – divides array into subarrays, sorts them, merges into a sorted array

Algorithm –

1. divide the array into two halves
2. sort each half
3. merge.
4. repeat till entire array is sorted

Code –

class MergeSort:

def sort(self, arr):

if len(arr) > 1:

mid = len(arr) // 2

left\_arr = arr[:mid]

right\_arr = arr[mid:]

self.sort(left\_arr)

self.sort(right\_arr)

i = j = k = 0

while i < len(left\_arr) and j < len(right\_arr):

if left\_arr[i] < right\_arr[j]:

arr[k] = left\_arr[i]

i += 1

else:

arr[k] = right\_arr[j]

j += 1

k += 1

while i < len(left\_arr):

arr[k] = left\_arr[i]

i += 1

k += 1

while j < len(right\_arr):

arr[k] = right\_arr[j]

j += 1

k += 1

return arr

arr = [5, 3, 8, 4, 2]

merge\_sort = MergeSort()

sorted\_arr = merge\_sort.sort(arr)

print(f"Original array: {arr}")

print(f"Sorted array: {sorted\_arr}")

1. Quick sort – chooses a pivot and partitions the array into subarrays, concatenates them after sorting all

Algorithm –

1. choose a pivot element
2. divide array into 3 parts, elements less than pivot, elements equal to the pivot, elements greater than the pivot
3. sort arrays which have elements greater and lesser than the pivot
4. concatenate the arrays
5. return the sorted array

Code –

class QuickSort:

def sort(self, arr):

if len(arr) <= 1:

return arr

pivot = arr[len(arr) // 2]

left\_arr, right\_arr, equal\_arr = [], [], []

for num in arr:

if num < pivot:

left\_arr.append(num)

elif num > pivot:

right\_arr.append(num)

else:

equal\_arr.append(num)

return self.sort(left\_arr) + equal\_arr + self.sort(right\_arr)

arr = [5, 3, 8, 4, 2]

quick\_sort = QuickSort()

sorted\_arr = quick\_sort.sort(arr)

print(f"Original array: {arr}")

print(f"Sorted array: {sorted\_arr}")

1. Insertion sort – inserts each element into the correct position in the sorted part of the array

Algorithm –

1. run a loop from the second element to the end
2. for each element, compare it to the left element and directly place it in the correct position in the sorted part
3. repeat till array is sorted
4. return sorted array

Code –

class InsertionSort:

def sort(self, arr):

for i in range(1, len(arr)):

key = arr[i]

j = i - 1

while j >= 0 and key < arr[j]:

arr[j + 1] = arr[j]

j -= 1

arr[j + 1] = key

return arr

arr = [5, 3, 8, 4, 2]

insertion\_sort = InsertionSort()

sorted\_arr = insertion\_sort.sort(arr)

print(f"Original array: {arr}")

print(f"Sorted array: {sorted\_arr}")