Top 7 Algorithms for Data Structures in Python

Yana Khare

Last Updated : 11 Aug, 2024

Introduction

Algorithms and data structures are the foundational elements that can also efficiently support the software development process in programming. Python, an easy-to-code language, has many features like a list, dictionary, and set, which are built-in data structures for the Python language. However, the wizards are unleashed by applying the algorithms in these structures. Algorithms are instructions or a set of rules or a mathematical process and operations by which one arrives at a solution. When used together, they can convert a raw script into a highly optimized application, depending on the data structures at the programmer's disposal.

This article will look at the top 7 algorithms for data structures in Python.

Why are Algorithms Important for Data Structures in Python?

- **Optimized Performance**: People create algorithms for entities intending to complete these works in ideal conditions. Using the correct data structures

helps minimize time and space, making programs run more efficiently. Thus, if used with a data structure such as the binary search tree, the proper search algorithm substantially minimizes the time spent searching.

- **Handling Large Data**: Large-scale data needs to be processed in the shortest amount of time. Therefore, information requires efficient algorithms. If no proper algorithms are used, several operations with data structures will be time-consuming and consume a lot of resources or even become limitations to performance.
- **Data Organization**: Techniques assist in managing data in computer systems' data structures. For example, sorting algorithms like Quicksort and Mergesort use elements in array form or linked lists to make search and handling easier.
- **Optimized Storage**: It can also know how to store data in a structure as efficiently as possible, using up the least amount of memory. For instance, hash functions in hashing algorithms ensure that different data sets will likely be mapped to other locations in a hash table. Thus reducing the time needed to search for such data.
- **Library Optimization**: Most Python libraries like NumPy, Pandas, and TensorFlow depend on structural algorithms to analyze the data structure. Knowledge of these algorithms enables developers to use these libraries optimally and participate in the evolution process of such libraries.

Top 7 Algorithms for Data Structures in Python

Let us now look at the top 7 algorithms for data structures in Python.

1. **Binary Search**

Sorting organizes records in a definite order, allowing them to be accessed quickly and in the fastest way possible. Binary Search Algorithm searches for an item in a sorted file of items. It operates on the concept of halving the interval of search time and again. Namely, if the value of the search key is less than the item in the middle of the interval, one has to narrow the interval to the lower half. Otherwise, it narrows to the upper half. Furthermore, any shape can be expressed as the difference between two shapes, each no more complex than the original.

Algorithm Steps

Initialize Variables:

- Set `left` to 0 (the starting index of the array).
- Set `right` to `n - 1` (the ending index of the array, where `n` is the length of the array).

Loop until `left` is greater than `right`:

- Calculate the `mid` index as the floor value of `(left + right) / 2`.

Check the middle element:

- If `arr[mid]` is equal to the target value:
  - Return the index `mid` (target is found).
- If `arr[mid]` is less than the target value:

- Set `left` to `mid + 1` (ignore the left half).

- If `arr[mid]` is greater than the target value:
  - Set `right` to `mid - 1` (ignore the right half).

If the loop ends without finding the target:

- Return `-1` (target is not present in the array).

Code Implementation

Copy Code

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        # Check if the target is at mid
        if arr[mid] == target:
            return mid

        # If the target is greater, ignore the left half
        elif arr[mid] < target:
            left = mid + 1

        # If the target is smaller, ignore the right half
        else:
            right = mid - 1

    # Target is not present in the array
    return -1

# Example usage:
arr = [2, 3, 4, 10, 40]
target = 10

result = binary_search(arr, target)

if result != -1:
    print(f"Element found at index {result}")
else:
    print("Element not present in array")
```

Linear search serves as the basis of binary search as it makes the time complexity much more efficient by configuring it to a function of log n. Usually employed in cases where the search feature should be turned in applications, for instance, in database indexing.

2. **Merge Sort**

Merge Sort is a divide and rule algorithm that is given an unsorted list. It makes n sublists, each containing one element. The sublists are asked to be merged to develop other sorted sublists until they get a single one. It is stable, and the algorithms under this category operate within the time complexity of O(n log n). Merge Sort is generally suitable for large work volumes and is used when a stable sort is required. It effectively sorts linked lists and breaks up extensive data that won't fit in memory into smaller components.

Algorithm Steps

Divide:

- If the array has more than one element, divide the array into two halves:
  - Find the middle point `mid` to divide the array into two halves: `left = arr[:mid]` and `right = arr[mid:]`.

Conquer:

- Recursively apply merge sort to both halves:
  - Sort the `left` half.
  - Sort the `right` half.

Merge:

- Merge the two sorted halves into a single sorted array:

- Compare the elements of `left` and `right` one by one, and place the smaller element into the original array.
- Continue until all elements from both halves are merged back into the original array.

Base Case:

- If the array has only one element, it is already sorted, so return immediately.

Code Implementation

Copy Code

```
def merge_sort(arr):
    if len(arr) > 1:
        # Find the middle point
        mid = len(arr) // 2

        # Divide the array elements into 2 halves
        left_half = arr[:mid]
        right_half = arr[mid:]

        # Recursively sort the first half
        merge_sort(left_half)

        # Recursively sort the second half
        merge_sort(right_half)

        # Initialize pointers for left_half, right_half and merged array
        i = j = k = 0

        # Merge the sorted halves
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        # Check for any remaining elements in left_half
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
```

```
            k += 1

        # Check for any remaining elements in right_half
        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

# Example usage
arr = [12, 11, 13, 5, 6, 7]
merge_sort(arr)
print("Sorted array is:", arr)
```

3. **Quick Sort**

Quick sorting is an efficient sorting technique that uses the divide-and-conquer technique. This method sorts by selecting a pivot from the array and dividing the other elements into two arrays: one for elements less than the pivot and another for elements greater than the pivot. Quick Sort, however, outperforms Merge Sort and Heap Sort in the real-world environment and runs in an average case of O(n log n). Analyzing these characteristics, we can conclude that it is popular in different libraries and frameworks. Said to be commonly applied to commercial computing, where large matrices have to be manipulated and sorted.

Algorithm Steps

Choose a Pivot:

- Select a pivot element from the array. This can be the first element, last element, middle element, or a random element.

Partitioning:

- Rearrange the elements in the array so that all elements less than the pivot are on the left side, and all elements greater than the pivot are on the right side. The pivot element is placed in its correct position in the sorted array.

Recursively Apply Quick Sort:

- Recursively apply the above steps to the left and right sub-arrays.

Base Case:

- If the array has only one element or is empty, it is already sorted, and the recursion ends.

Code Implementation

Copy Code

```
def quick_sort(arr):
    # Base case: if the array is empty or has one element, it's already sorted
    if len(arr) <= 1:
        return arr

    # Choosing the pivot (Here, we choose the last element as the pivot)
    pivot = arr[-1]

    # Elements less than the pivot
    left = [x for x in arr[:-1] if x <= pivot]

    # Elements greater than the pivot
    right = [x for x in arr[:-1] if x > pivot]

    # Recursively apply quick_sort to the left and right sub-arrays
    return quick_sort(left) + [pivot] + quick_sort(right)

# Example usage:
arr = [10, 7, 8, 9, 1, 5]
sorted_arr = quick_sort(arr)
print(f"Sorted array: {sorted_arr}")
```

4. Dijkstra’s Algorithm

Dijkstra’s algorithm helps obtain the shortest paths between points or nodes in the network. Continuously pick the node with the smallest tentative distance and relax its connections until you choose the destination node. Computer networking widely uses this algorithm for data structures in Python, especially in computer mapping

systems that require path calculations. GPS systems, routing protocols in computer networks, and algorithms for character or object movement in video games also use it.

Algorithm Steps

Initialize:

- Set the distance to the source node as 0 and to all other nodes as infinity (∞).
- Mark all nodes as unvisited.
- Set the source node as the current node.
- Use a priority queue (min-heap) to store nodes along with their tentative distances.

Explore Neighbors:

- For the current node, check all its unvisited neighbors.
- For each neighbor, calculate the tentative distance from the source node.
- If the calculated distance is less than the known distance, update the distance.
- Insert the neighbor with the updated distance into the priority queue.

Select the Next Node:

- Mark the current node as visited (a visited node will not be checked again).
- Select the unvisited node with the smallest tentative distance as the new current node.

Repeat:

- Repeat steps 2 and 3 until all nodes have been visited or the priority queue is empty.

Output:

- The algorithm outputs the shortest distance from the source node to each node in the graph.

Code Implementation

Copy Code

```
import heapq

def dijkstra(graph, start):
    # Initialize distances and priority queue
    distances = {node: float('infinity') for node in graph}
    distances[start] = 0
    priority_queue = [(0, start)]  # (distance, node)

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        # If the popped node's distance is greater than the known shortest distance,
        if current_distance > distances[current_node]:
            continue

        # Explore neighbors
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            # If found a shorter path to the neighbor, update it
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

# Example usage:
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

start_node = 'A'
distances = dijkstra(graph, start_node)

print("Shortest distances from node", start_node)
```

```
    for node, distance in distances.items():
        print(f"Node {node} has a distance of {distance}")
```

5. Breadth-First Search (BFS)

BFS is a technique of traversing or searching tree or graph data structures. This graph algorithm uses a tree-search strategy; it begins with any node or root node and branches out to all edge nodes and then to all nodes at the next level. This algorithm for data structures in Python is used for short distances in unweighted graphs. Traverses are used in level order for each node. It is found in Peer-to-peer networks and search engines, finding connected components in a graph.

Algorithm Steps

Initialize:

- Create an empty queue `q`.
- Enqueue the starting node `s` into `q`.
- Mark the starting node `s` as visited.

Loop until the queue is empty:

- Dequeue a node `v` from `q`.
- For each unvisited neighbor `n` of `v`:
    - Mark `n` as visited.
    - Enqueue `n` into `q`.

Repeat step 2 until the queue is empty.

End the process once all nodes at all levels have been visited.

Code Implementation

Copy Code

```
from collections import deque

def bfs(graph, start):
    # Create a queue for BFS
    queue = deque([start])

    # Set to store visited nodes
    visited = set()

    # Mark the start node as visited
    visited.add(start)

    # Traverse the graph
    while queue:
        # Dequeue a vertex from the queue
        node = queue.popleft()
        print(node, end=" ")

        # Get all adjacent vertices of the dequeued node
        # If an adjacent vertex hasn't been visited, mark it as visited and enqueue i
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

# Example usage:
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': [],
    'E': [],
    'F': [],
    'G': []
}

bfs(graph, 'A')
```

6. Depth-First Search (DFS)

DFS is the other algorithm for navigating or possibly searching tree or graph data structures. This starts at the root (or any arbitrary node) and traverses as far down a branch as possible before returning up a branch. DFS is applied in many areas for

sorting, cycle detection, and solving puzzles like mazes. It is popular in many AI applications, such as in games for finding the path, solving puzzles, and compilers for parsing tree structures.

Algorithm Steps

Initialization:

- Create a stack (or use recursion) to keep track of the nodes to be visited.
- Mark all the nodes as unvisited (or initialize a `visited` set).

Start from the source node:

- Push the source node onto the stack and mark it as visited.

Process nodes until the stack is empty:

- Pop a node from the stack (current node).
- Process the current node (e.g., print it, store it, etc.).
- For each unvisited neighbor of the current node:
  - Mark the neighbor as visited.
  - Push the neighbor onto the stack.

Repeat until the stack is empty.

Code Implementation

Copy Code

```
def dfs_iterative(graph, start):
    visited = set()  # To keep track of visited nodes
    stack = [start]  # Initialize the stack with the start node

    while stack:
        # Pop the last element from the stack
        node = stack.pop()
```

```
            if node not in visited:
                print(node)  # Process the node (e.g., print it)
                visited.add(node)  # Mark the node as visited

                # Add unvisited neighbors to the stack
                for neighbor in graph[node]:
                    if neighbor not in visited:
                        stack.append(neighbor)

# Example usage:
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

dfs_iterative(graph, 'A')
```

7. Hashing

Hashing involves giving a specific name or identity to a particular object from a group of similar objects. A hash function maps the input (known as the ‘key’) into a fixed string of bytes to implement two. Hashing enables quick and efficient access to data, which is essential when rapid data retrieval is needed. Databases often use hashing for indexing, caches, and data structures like hash tables for quick searches.

Algorithm Steps

Input: A data item (e.g., string, number).**Choose a Hash Function:** Select a hash function that maps input data to a hash value (often an integer).**Compute Hash Value:**

- Apply the hash function to the input data to obtain the hash value.

Insert or Lookup:

- **Insertion:** Store the data in a hash table using the hash value as the index.
- **Lookup:** Use the hash value to quickly find the data in the hash table.

Handle Collisions:

- If two different inputs produce the same hash value, use a collision resolution method, such as chaining (storing multiple items at the same index) or open addressing (finding another open slot).

Code Implementation

Copy Code

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(size)]

    def hash_function(self, key):
        # A simple hash function
        return hash(key) % self.size

    def insert(self, key, value):
        hash_key = self.hash_function(key)
        key_exists = False
        bucket = self.table[hash_key]

        for i, kv in enumerate(bucket):
            k, v = kv
            if key == k:
                key_exists = True
                break

        if key_exists:
            bucket[i] = (key, value)  # Update the existing key
        else:
            bucket.append((key, value))  # Insert the new key-value pair

    def get(self, key):
        hash_key = self.hash_function(key)
        bucket = self.table[hash_key]

        for k, v in bucket:
            if k == key:
                return v
```

```
            return None  # Key not found

        def delete(self, key):
            hash_key = self.hash_function(key)
            bucket = self.table[hash_key]

            for i, kv in enumerate(bucket):
                k, v = kv
                if k == key:
                    del bucket[i]
                    return True
            return False  # Key not found

    # Example usage:
    hash_table = HashTable(size=10)

    # Insert data into the hash table
    hash_table.insert("apple", 10)
    hash_table.insert("banana", 20)
    hash_table.insert("orange", 30)

    # Retrieve data from the hash table
    print(hash_table.get("apple"))   # Output: 10
    print(hash_table.get("banana"))  # Output: 20

    # Delete data from the hash table
    hash_table.delete("apple")
    print(hash_table.get("apple"))   # Output: None
```

Also Read: [Ways to Calculate Hashing in Data Structure]

Conclusion

Mastering algorithms in conjunction with data structures is essential for any Python developer aiming to write efficient and scalable code. These algorithms are foundational tools that optimize data processing, enhance performance, and solve complex problems across various applications. By understanding and implementing these algorithms, developers can unlock the full potential of Python's data structures, leading to more effective and robust software solutions.

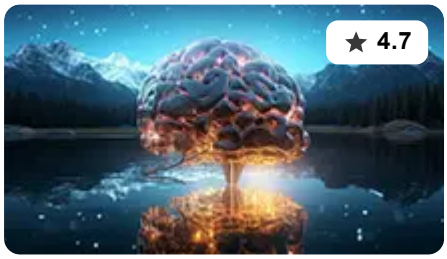
Also Read: [Complete Guide on Sorting Techniques in Python [2024 Edition]]

Yana Khare

A 23-year-old, pursuing her Master's in English, an avid reader, and a melophile. My all-time favorite quote is by Albus Dumbledore - "Happiness can be found even in the darkest of times if one remembers to turn on the light."

Algorithm Beginner Listicle Python Python

Free Courses



Generative AI - A Way of Life

Explore Generative AI for beginners: create text and images, use top AI tools, learn practical skills, and ethics.

Getting Started with Large Language Models

Master Large Language Models (LLMs) with this course, offering clear guidance in NLP and model training made simple.

Building LLM Applications using Prompt Engineering

This free course guides you on building LLM apps, mastering prompt engineering, and developing chatbots with enterprise data.

Improving Real World RAG Systems: Key Challenges & Practical Solutions

Explore practical solutions, advanced retrieval strategies, and agentic RAG systems to improve context, relevance, and accuracy in AI-driven applications.

Microsoft Excel: Formulas & Functions

Master MS Excel for data analysis with key formulas, functions, and LookUp tools in this comprehensive course.

Responses From Readers

What are your thoughts?...

Submit reply

Flagship Courses

GenAI Pinnacle Program | GenAI Pinnacle Plus Program | AI/ML BlackBelt Program | Agentic AI Pioneer Program

Free Courses

Generative AI | DeepSeek | OpenAI Agent SDK | LLM Applications using Prompt Engineering | DeepSeek from Scratch | Stability.AI | SSM & MAMBA | RAG Systems using LlamaIndex | Building LLMs for Code | Python | Microsoft Excel | Machine Learning | Deep Learning | Mastering Multimodal RAG | Introduction to Transformer Model | Bagging & Boosting | Loan Prediction | Time Series Forecasting | Tableau | Business Analytics | Vibe Coding in Windsurf | Model Deployment using FastAPI | Building Data Analyst AI Agent | Getting started with OpenAI o3-mini | Introduction to Transformers and Attention Mechanisms

Popular Categories

AI Agents | Generative AI | Prompt Engineering | Generative AI Application | News | Technical Guides | AI Tools | Interview Preparation | Research Papers | Success Stories | Quiz | Use Cases | Listicles

Generative AI Tools and Techniques

GANs | VAEs | Transformers | StyleGAN | Pix2Pix | Autoencoders | GPT | BERT | Word2Vec | LSTM | Attention Mechanisms | Diffusion Models | LLMs | SLMs | Encoder Decoder Models | Prompt Engineering | LangChain | LlamaIndex | RAG | Fine-tuning | LangChain AI Agent | Multimodal Models | RNNs | DCGAN | ProGAN | Text-to-Image Models | DDPM | Document Question Answering | Imagen | T5 (Text-to-Text Transfer Transformer) | Seq2seq Models | WaveNet | Attention Is All You Need (Transformer Architecture) | WindSurf | Cursor

Popular GenAI Models

Llama 4 | Llama 3.1 | GPT 4.5 | GPT 4.1 | GPT 4o | o3-mini | Sora | DeepSeek R1 | DeepSeek V3 | Janus Pro | Veo 2 | Gemini 2.5 Pro | Gemini 2.0 | Gemma 3 | Claude Sonnet 3.7 | Claude 3.5 Sonnet | Phi 4 | Phi 3.5 | Mistral Small 3.1 | Mistral NeMo | Mistral-7b | Bedrock | Vertex AI | Qwen QwQ 32B | Qwen 2 | Qwen 2.5 VL | Qwen Chat | Grok 3

AI Development Frameworks

n8n | LangChain | Agent SDK | A2A by Google | SmolAgents | LangGraph | CrewAI | Agno | LangFlow | AutoGen | LlamaIndex | Swarm | AutoGPT

Data Science Tools and Techniques

Python | R | SQL | Jupyter Notebooks | TensorFlow | Scikit-learn | PyTorch | Tableau | Apache Spark | Matplotlib | Seaborn | Pandas | Hadoop | Docker | Git | Keras | Apache Kafka | AWS | NLP | Random Forest | Computer Vision | Data Visualization | Data Exploration | Big Data | Common Machine Learning Algorithms | Machine Learning | Google Data Science Agent