pythorn

Release 1.0.0

Robin Singh

Jan 19, 2022

DOCUMENTATION:

CHAPTER

ONE

INTRODUCTION

`Pythorn`: A python module that contains Python-based minimal and clean example implementations of popular data structures and all major algorithms!! Mainly for educational purposes

Features!

A python module written in python having all the major algorithms and claen data structure implementaions. Get the code, time complexities and much more by just importing the required algorithm. A Easiest way to start learning algorithms and data structures without surfing the internet.

1.1 Stack

Info Stack docs

Author Gourav <gouravpatel11072@gmail.com>

Date 2020-11-30 (Mon, 30 Nov 2020)

Description Added stack operations and infix to postfix.

1.1.1 Quick Start Guide

`stack` :- Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

A stack allows access to only one data item: the last item inserted. If you remove this item, you can access the next-to-last item inserted, and so on.

A stack is also a handy aid for algorithms applied to certain complex data structures. In "Binary Trees", we'll see it used to help traverse the nodes of a tree.

Notice how the order of the data is reversed. Because the last item pushed is the first one popped.

commonly implemented with linked lists but can be made from arrays too.

operations of stack: pop(), push(), tos(), isEmpty(), display().

```
# import the required data structure
>>> from pythorn.data_structures.stack import Stack

# creating a stack
>>> a = Stack()

# push elements
```

(continues on next page)

(continued from previous page)

```
>>> a.push(5)
>>> a.push(20)
>>> a.push(13)

# displaying full stack
>>> a.display()
[5, 20, 13]

# top element
>>> a.tos()
13

# poping the element
>>> a.pop()
13
>>> a.isEmpty()
False
```

Example Code for Infix To Postfix

`Infix expression`:The expression of the form a op b. When an operator is in-between every pair of operands.

`Postfix expression`:The expression of the form a b op. When an operator is followed for every pair of operands.

```
# importing Stack and Infix_Postfix
>>> from pythorn.data_structures.stack import Stack
from pythorn.data_structures.stack import Infix_Postfix

# creating a stack
my_stack = Stack()

# My Expression
my_exp = "a+c-*/dsefj-+//jk"

# passing stack and expression to the Infix_Postfix class
infixpostfix = Infix_Postfix(my_exp,my_stack)
infixpostfix.infixToPostfix()
'a c + * d s e f j / - - / j k / +'
```

1.1.2 Stack Programs

Author : Robin Singh

Programs List:

1.Stack

2.Infix To Postfix Conversion Using Stack

3.Integer To Binary Conversion Using Stack

Stack

class pythorn.data_structures.stack.**Stack**
Stack implementation

push(*item*)
Pushes an item into the array

pop()
Pops the top element from the stack, and return -1 if the stack is empty

tos()
returns the top element of the stack, else return -1 if stack is empty

size()
returns the length of the stack, else returns -1 if stack is empty

isEmpty()
checks if the stack is empty or not and returns boolean value

display()
if there is an element in the stack then it prints the full stack ,else returns -1

static get_code()

Returns source code

static time_complexity()

Returns time complexity

Infix To Postfix

class pythorn.data_structures.stack.**Infix_Postfix**(*expression=None*, *stack=None*)
Infix To Postfix conversion

static oper(*char*)
Function of checking whether the given character is an operator or not.

infixToPostfix()
main function for converting infix to postfix using stack.

Returns converted expression

static get_code()

Returns source code

Integer To Binary

Example Code for Integer To Binary

```
# importing Stack and Integer_Binary
>>> from pythorn.data_structures.stack import Stack
>>> from pythorn.data_structures.stack import Integer_Binary

# creating a stack
```

(continues on next page)

(continued from previous page)

```
>>> my_stack = Stack()

# My Number
>>> my_num = 45

# passing my_stack and my_num to the Integer_Binary class
>>> integerbinary = Integer_Binary(my_num,my_stack)
>>> integerbinary.IntegerBinary()
'101101'
```

class pythorn.data_structures.stack.**Integer_Binary**(*Number=None*, *stack=None*)

Integer To Binary Conversion

static get_code()

Returns source code

1.2 Queue

Info Queue Docs

Author Robin Singh <robin25tech@gmail.com>

Date 2020-12-18 (Fri, 18 Dec 2020)

Description Added queue docs with full descriptions

1.2.1 Quick Start Guide

`Queue` :- A queue is a data structure that is some- what like a stack, except that in a queue the first item inserted is the first to be removed (First-In-First-Out, *FIFO*), while in a stack, as we've seen, the last item inserted is the first to be removed (*LIFO*).

too can be implemented with a linked list or an array.

- Queues are a **first in, first out** (FIFO) data structure.
- Made with a doubly linked list that only removes from head and adds to tail.

In programming terms, putting an item in the queue is called an "enqueue" and removing an item from the queue is called "dequeue".

We can implement the queue in any programming language like C, C++, Java, Python or C#, but the specification is pretty much the same.

Basic Operations of Queue

A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations:

- Enqueue: Add an element to the end of the queue
- Dequeue: Remove an element from the front of the queue
- IsEmpty: Check if the queue is empty
- IsFull: Check if the queue is full
- Peek: Get the value of the front of the queue without removing it

Working of Queue Queue operations work as follows:

- two pointers `FRONT` and `REAR`
- `FRONT` track the first element of the queue
- `REAR` track the last elements of the queue
- initially, set value of `FRONT` and `REAR` to -1

Enqueue Operation

- check if the queue is full
- for the first element, set value of `FRONT` to 0
- increase the `REAR` index by 1
- add the new element in the position pointed to by `REAR`

Dequeue Operation

- check if the queue is empty
- return the value pointed by `FRONT`
- increase the `FRONT` index by 1
- for the last element, reset the values of `FRONT` and `REAR` to -1

Limitation of Queue

- As you can see in the image below, after a bit of enqueuing and dequeuing, the size of the queue has been reduced.
- The indexes 0 and 1 can only be used after the queue is reset when all the elements have been dequeued.
- After REAR reaches the last index, if we can store extra elements in the empty spaces (0 and 1), we can make use of the empty spaces. This is implemented by a modified queue called the Circular queue

Applications of Queue Data Structure

- CPU scheduling, Disk Scheduling
- When data is transferred asynchronously between two processes.The queue is used for synchronization. eg: IO - Buffers, pipes, file IO, etc
- Handling of interrupts in real-time systems.
- Call Center phone systems use Queues to hold people calling them in an order

Types of queue `Simple Queue` :- In a simple queue, insertion takes place at the rear and removal occurs at the front. It strictly follows FIFO rule.

`Circular Queue` :- In a circular queue, the last element points to the first element making a circular link.The main advantage of a circular queue over a simple queue is better memory utilization. If the last position is full and the first position is empty then, an element can be inserted in the first position. This action is not possible in a simple queue.

How Circular Queue Works

- Circular Queue works by the process of circular increment i.e. when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.

Circular Queue Operations The circular queue work as follows:

- two pointers `FRONT` and `REAR`
- `FRONT` track the first element of the queue

- `REAR` track the last elements of the queue
- initially, set value of `FRONT` and `REAR` to -1

Enqueue Operation

- check if the queue is full
- for the first element, set value of `FRONT` to 0
- circularly increase the ``REAR index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)
- add the new element in the position pointed to by `REAR`

Dequeue Operation

- check if the queue is empty
- return the value pointed by `FRONT`
- circularly increase the `FRONT` index by 1
- for the last element, reset the values of `FRONT` and `REAR` to -1
- However, the check for full queue has a new additional case:
- Case 1: `FRONT` = 0 && `REAR` == SIZE - 1
- Case 2: `FRONT` = `REAR` + 1
- The second case happens when REAR starts from 0 due to circular increment and when its value is just 1 - less than `FRONT`, the queue is full.

Applications of Circular Queue

- CPU scheduling
- Memory management
- Traffic Management

`Deque (Double Ended Queue)` :- Double Ended Queue is a type of queue in which insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow FIFO rule (First In First Out).

Types of Deque

- Input Restricted Deque
- In this deque, input is restricted at a single end but allows deletion at both the ends.
- Output Restricted Deque
- In this deque, output is restricted at a single end but allows insertion at both the ends.

Operations on a Deque

- Below is the circular array implementation of deque. In a circular array, if the array is full, we start from the beginning.
- But in a linear array implementation, if the array is full, no more elements can be inserted. In each of the operations below, if the array is full, “overflow message” is thrown.
- Before performing the following operations, these steps are followed.
- Take an array (deque) of size n.
- Set two pointers at the first position and set front = -1 and rear = 0.

Insert at the Front This operation adds an element at the front.

- Check the position of front.
- If `front < 1`, reinitialize `front = n-1` (last index)
- Else, decrease `front` by 1
- Add the new key `5` into `array[front]`

Insert at the Rear This operation adds an element to the rear.

- Check if the array is full
- If the deque is full, reinitialize `rear = 0`
- Else, increase `rear` by 1.
- Add the new key `5` into `array[rear]`

Delete from the Front The operation deletes an element from the front.

- Check if the deque is empty
- If the deque is empty (i.e. `front = -1`), deletion cannot be performed (underflow condition).
- If the deque has only one element (i.e. front = rear), set front = -1 and rear = -1.
- Else if front is at the end (i.e. `front = n - 1`), set go to the front `front = 0`.
- Else, `front = front + 1`

Delete from the Rear This operation deletes an element from the rear.

- Check if the deque is empty
- If the deque is empty (i.e. `front = -1`), deletion cannot be performed (underflow condition).
- If the deque has only one element (i.e. `front = rear`), set `front = -1` and `rear = -1`, else follow the steps below.
- If rear is at the front (i.e. `rear = 0`), set go to the `front rear = n - 1`
- Else, `rear = rear - 1`

Check Empty

- This operation checks if the deque is empty. If `front = -1`, the deque is empty.

Check Full

- This operation checks if the deque is full. If `front = 0` and `rear = n - 1` OR `front = rear + 1`, the deque is full.

Applications of Deque Data Structure

- In undo operations on software.
- To store history in browsers.
- For implementing both stacks and queues.

Time Complexity The time complexity of all the above methods and operations is constant i.e. O(1).

Below is the simple example for how to use queue using this package.

```
# import the required data structure
>>> from pythorn.data_structures.queue import Queue

# creating a stack
```

(continues on next page)

(continued from previous page)

```
>>> a = Queue()

# enqueue elements
>>> a.enqueue(5)
>>> a.enqueue(13)
>>> a.enqueue(27)

# display elements
>>> a.display()
[5, 13, 27]

# dequeue elements
>>> a.dequeue()
5
>>> a.dequeue()
13
>>> a.display()
[27]
```

1.2.2 Queue Programs

Author : Robin Singh Programs List: 1.Queue 2.Circular Queue 3.Double Ended Queue

Queue

class pythorn.data_structures.queue.**Queue**(*length=5*)

isEmpty()
checks the queue if its empty or not

enqueue(*data*)
inserts an element into the queue

dequeue()
removes an element from the queue

Size()
returns size of the queue

display()
displays full queue

static get_code()

Returns source code

static time_complexity()

Returns time complexity

Circular Queue

class pythorn.data_structures.queue.**CircularQueue**(*length=5*)

Parameters **length** – pass queue length while making object otherwise default value will be 5

isEmpty()
Checks whether queue is empty or not

isQueuefull()
checks whether queue is full or not

enqueue(*data*)
inserts an element into the queue

dequeue()
removes an element from the queue

display()
displays full queue

static get_code()

Returns source code

static time_complexity()

Returns time complexity

Dequeue

class pythorn.data_structures.queue.**Deque**(*length=5*)

Parameters **length** – pass queue length while making object otherwise default value will be 5

isFull()
checks whether queue is full or not

isEmpty()
Checks whether queue is empty or not

enqueue_start(*element*)
inserts an element at the start of the queue

enqueue_end(*ele*)
inserts an element at the end of the queue

dequeue_start()
deletes an element from the start of the queue

dequeue_end()
deletes an element from the end of the queue

display()
displays full queue

static get_code()

Returns source code

`static time_complexity()`

Returns time complexity

1.3 Linked List

Info Linked List Docs

Author Robin Singh <robin25tech@gmail.com>

Date 2020-12-19 (Fri, 19 Dec 2020)

Description Added Linked list documentaion with time complexity

1.3.1 Quick Start Guide

`Linked List` :- Arrays had certain disadvantages as data storage structures. In an unordered array, searching is slow, whereas in an ordered array, insertion is slow. In both kinds of arrays, deletion is slow. Also, the size of an array can't be changed after it's created.

We'll look at a data storage structure that solves some of these problems: the linked list. Linked lists are probably the second most commonly used general-purpose storage structures after arrays.

In a linked list, each data item is embedded in a link. A link is an object of a class called something like Link. Each Link object contains a reference (usually called next) to the next link in the list.

The LinkList class contains only one data item: a reference to the first link on the list. This reference is called first or``HEAD``. It's the only permanent information the list maintains about the location of any of the links. It finds the other links by following the chain of references from first, using each link's next field.

A linked list data structure includes a series of connected nodes. Here, each node store the data and the address of the next node.You have to start somewhere, so we give the address of the first node a special name called `HEAD`.Also, the last node in the linked list can be identified because its next portion points to `NULL`.

You might have played the game Treasure Hunt, where each clue includes the information about the next clue. That is how the linked list operates.

Linked List	Time Complexity	
Cases	Worst	Average
Search	O(n)	O(n)
Insert	O(1)	O(1)
Deletion	O(1)	O(1)

- **Linked List Applications**
  - Dynamic memory allocation
  - Implemented in stack and queue
  - In undo functionality of softwares
  - Hash tables, Graphs

Basic Operations On Linked List

- Two important points to remember:
  - head points to the first node of the linked list

  - next pointer of the last node is NULL, so if the next current node is NULL, we have reached the end of the linked list.

In all of the examples, we will assume that the linked list has three nodes 1 —>2 —>3 with node structure as below:

- **How to Traverse a Linked List**
  - Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.
  - When temp is `NULL`, we know that we have reached the end of the linked list so we get out of the while loop.
  - **The output of this program will be:** `1 --->2 --->3 --->`
- How to Add Elements to a Linked List

You can add elements to either the beginning, middle or end of the linked list.

- **Add to the beginning**
  - Allocate memory for new node
  - Store data
  - Change next of new node to point to head
  - Change head to point to recently created node
- **Add to the End**
  - Allocate memory for new node
  - Store data
  - Traverse to last node
  - Change next of last node to recently created node
- **Add to the Middle**
  - Allocate memory and store data for new node
  - Traverse to node just before the required position of new node
  - Change next pointers to include new node in between
- How to Delete from a Linked List

You can delete either from the beginning, end or from a particular position.

- **Delete from beginning**
  - Point head to the second node : `head = head->next`
- **Delete from end**
  - Traverse to second last element
  - Change its next pointer to null
- **Delete from middle**
  - Traverse to element before the element to be deleted
  - Change next pointers to exclude the node from the chain

Types of Linked List - There are three common types of Linked List.

- Singly Linked List

- A singly linked list is a type of linked list that is unidirectional, that is, it can be traversed in only one direction from head to the last node (tail).
- Each element in a linked list is called a node. A single node contains data and a pointer to the next node which helps in maintaining the structure of the list.
- **Doubly Linked List**
  - Let's examine another variation on the linked list: the doubly linked list (not to be confused with the double-ended list). What's the advantage of a doubly linked list? A potential problem with ordinary linked lists is that it's difficult to traverse backward along the list. A statement like current=current.next steps conveniently to the next link, but there's no corresponding way to go to the previous link.
  - The doubly linked list provides this capability. It allows you to traverse backward as well as forward through the list. The secret is that each link has two references to other links instead of one. The first is to the next link, as in ordinary lists. The second is to the previous link.
- **Circular Linked List**
  - A circular linked list is a variation of a linked list in which the last element is linked to the first element. This forms a circular loop.
  - A circular linked list can be either singly linked or doubly linked.
    - for singly linked list, next pointer of last item points to the first item
    - In the doubly linked list, `prev` pointer of the first item points to the last item as well.

Below is the sample example code of linked list using this package

```
# import the required data structure
>>> from pythorn.data_structures.linked_list import SinglyList

# creating linked list
>>> a = SinglyList()

# inserting element
>>> a.insert(5)
>>> a.insert(8)
>>> a.insert(16)

# displaying elements
>>> a.display_list()
16 8 5

# delete element
>>> a.delete()
16

# size of the linked list
>>> a.size()
2
>>> a.display_list()
8 5
```

1.3.2 Linked List Programs

Author : Robin Singh

Programs List:

1.Singly Linked List 2.Doubly Linked List 3.Circular Linked List 4.Stack Using Linked List 5.Queue Using Linked List

Singly Linked List

class pythorn.data_structures.linked_list.**SinglyList**

Singly Linked List Implementation

insert(*ele*)

inserts an element at the start of the linked list

delete()

deletes an element from the start

display_list()

displays current linked list

size()

returns size of the linked list

static get_code()

Returns source code

static time_complexity()

Returns time complexity

Doubly Linked List

class pythorn.data_structures.linked_list.**DoublyList**

Doubly Linked List Implementation

insert_start(*ele*)

inserts an element at the start of the linked list

insert_end(*ele*)

inserts an element at the end of the linked list

delete_start()

deletes an element from the start

delete_end()

deletes an element from the end

display_list()

displays current linked list

size()

returns size of the linked list

static get_code()

Returns source code

static time_complexity()

Returns time complexity

CircularList

class pythorn.data_structures.linked_list.**CircularList**

Circular Linked List Implementation

is_Empty()
checks whether list is empty or not

insert_start(*e*)
inserts an element at the start

insert_end(*e*)
inserts an element at the end

insert_position(*e*, *pos*)
inserts an element at the given position

delete()
deletion of an element from the start

display_list()
displays current linked list

static get_code()

Returns source code

static time_complexity()

Returns time complexity

Stack Using Linked List

class pythorn.data_structures.linked_list.**Stack_LinkedList**

Stack Using Linked List

push(*ele*)
for pushing the element into the stack

pop()
for poping out the element

isEmpty()
checks whether stack is empty or not

tos()
tos = top of stack , to find the value of the top most element in the stack

display_stack()
displays full stack

size()
returns size of the stack

static get_code()

Returns source code

static time_complexity()

Returns time complexity

Queue Using Linked List

class pythorn.data_structures.linked_list.**Queue_LinkedList**

Implementation of queue uisng linked list

enqueue(*ele*)
for pushing the element into the queue

dequeue()
for poping out the element

display_queue()
displays full queue

isEmpty()
checks whether queue is empty or not

last_element()
to find the value of the last most element in the queue

size()
returns size of the queue

static get_code()

Returns source code

static time_complexity()

Returns time complexity

1.4 Recursion

1.4.1 Quick Start Guide

`stack` :- Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

A stack allows access to only one data item: the last item inserted. If you remove this item, you can access the next-to-last item inserted, and so on.

A stack is also a handy aid for algorithms applied to certain complex data structures. In "Binary Trees", we'll see it used to help traverse the nodes of a tree.

Notice how the order of the data is reversed. Because the last item pushed is the first one popped.

commonly implemented with linked lists but can be made from arrays too.

operations of stack: pop(), push(), tos(), isEmpty(), display().

1.4.2 Recursion Programs

Author : Robin Singh

Programs List : 1 . Factorial 2 . Fibonacci 3 . Tower Of Hanoi 4 . Binary Search (Recursive)

pythorn.data_structures.recursion.**tower_of_hanoi**(*disk*, *source*, *destination*, *auxiliary*)

Tower Of Hanoi Implementation Using Recursion

Tower of hanoi is a mathematical puzzle.It consists of three poles and a number of disks of different sizes which can slide onto any poles.

The puzzle starts with a disk in a neat stack u ascending order of size in one pole, the smallest at the top making it a conical shape.

The objective of the puzzle is to move all the disks from one pole (Source Pole) to another pole (Destination pole) with the help of the third pole (auxiliary pole)

The puzzle has two rules : 1 . You Can't place a larger disk onto smaller disk or vice-versa 2 . Only one disk can be moved at a time

Moves Reqd to Solve This puzzle is given by : 2^(n)-1

Example : Number Of Dics : 3 Source Pole : A Destination Pole : C Auxiliary Pole : D

Solution : Tower_of_hanoi(3,"A","B","C") Move Disk From Source A To Destination B Move Disk From Source A To Destination C Move Disk From Source B To Destination C Move Disk From Source A To Destination B Move Disk From Source C To Destination A Move Disk From Source C To Destination B Move Disk From Source A To Destination B

pythorn.data_structures.recursion.**binary_search**(*A*, *key*, *low*, *high*)

Implementation of Binary Search Recursive Method

Below function perform sa binary search on a sorted list and returns the index of the item if it;s present else returns false

param list a sorted list

param key key to search from given sorted list

return index of key if its found in the sorted list else returns false

Example : A = [5 9 10 45 65 78 98 102 1045] Key = 11

Binary_Search(A,11,0,len(A))

Solution : Not Present

A = [5 9 10 45 65 78 98 102 1045] Key = 65

Binary_Search(A,65,0,len(A))

Solution :

Element is Present at index : 4

1.5 Searching Algorithms

1.5.1 Quick Start Guide

`Searching Algorithms` :- Searching is also a common and well-studied task. This task can be described formally as follows:

Given a list of values, a function that compares two values and a desired value, find the position of the desired value in the list.

We will look at various algorithms that perform this task:

`linear search`, which simply checks the values in sequence until the desired value is found

Linear search is the simplest searching algorithm that searches for an element in a list in sequential order. We start at one end and check every element until the desired element is not found.

Linear Search Algorithm:

```
 LinearSearch(array, key)
for each item in the array
if item == value
     return its index
```

`binary search`, which requires a sorted input list, and checks for the value in the middle of the list, repeatedly discarding the half of the list which contains values which are definitely either all larger or all smaller than the desired value

Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

- **Binary Search Algorithm can be implemented in two ways which are discussed below.**
  - Iterative Method
  - Recursive Method

The recursive method follows the divide and conquer approach.

- Binary Search Algorithm

  Iteration Method:

```
do until the pointers low and high meet each other.
    mid = (low + high)/2
    if (x == arr[mid])
         return mid
    else if (x > A[mid]) // x is on the right side
         low = mid + 1
    else                          // x is on the left side
         high = mid - 1
```

  Recursive Method:

```
binarySearch(arr, x, low, high)
    if low > high
         return False
    else
         mid = (low + high) / 2
```

(continues on next page)

(continued from previous page)

```
        if x == arr[mid]
             return mid
        else if x < data[mid]        // x is on the right side
             return binarySearch(arr, x, mid + 1, high)
        else                                // x is on the right side
             return binarySearch(arr, x, low, mid - 1)
```

`Jump Search` :- Like binary search, Jump search is a searching algorithm for sorted array.

The basic idea is to check fewer elements by jumping ahead by fixed steps or skipping some elements in the place of searching all elements

For example : suppose we have an array[] of size n and block (to be jumped) size m . then we search at the indexes array[0],array[m],array[2m],.....array[km]and so on.

Once we find the interval (array[km] < X < array[(k+1)m]),we perform a linear search operation from the index km to find the element x.

Example:: let array : [0,2,6,8,10,21,34,66,89,120,124,300,350,500,549,600]

len(array) = 16

key = 66

Assume block size = 4

1 . Jump from index 0 to index 4 2 . jump from index 4 ot index 8 3 . since element at index 8 (89) is greater than the key element (66) so we will jump back to index 4 4 . now from here we will perform linear search from index 4 to index 8 to get our key element 66

`Interpolation Search` :- Interpoaltion search algorithm is a search algorithm that has been inspired by the way humans searc through a telephone book for a particular name,the key value by which book's entries are ordered.

It is an improvement above binary search, in binary search ,we always move to the middle element whereas interpolation search moves to a different element in order to reduce the search space further.

for example : if the value of the key is closer to the last element ,interpolation search is likely to start towards the end side.

`Fibonacci Search` :- Fibonacci search is an efficient search algorithm based on divide and conquer principle that can find an element in the given sorted array with the help of fibonacci series in O(Logn) time complexity.

On Avg , fibonacci search require 4% more comparisons than binary search

There are numerous other searching techniques. Often they rely on the construction of more complex data structures to facilitate repeated searching. Examples of such structures are hash tables (such as Python's dictionaries) and prefix trees. Inexact searches that find elements similar to the one being searched for are also an important topic.

Searching Algorithms	Time Complexity		
Searching	Best	Average	Worst
Binary Search	O(1)	O(Logn)	O(Logn)
Fibonacci Search	O(1)	O(Logn)	O(Logn)
Interpolation Search	O(1)	O(Log Logn)	O(n)
Jump Search	O(1)	O(n)	O(n)
Linear Search	O(1)	O(n)	O(n)

Below is the simple example for how to use queue using this package.

```
# impor required algorithm
>>> from pythorn.data_structures.searching import LinearSearch

# create list
>>> a = [1,5,9,10,78,90,650]

# pass list and key
>>> b = LinearSearch(a,10)

# call the function
>>> b.linear_search()
Key found at index :  3
```

1.5.2 Searching Programs

Author : Robin Singh

Programs List : 1 . Binary Search (Iterative Method) 2 . Fibonacci Search 3 . Interpolation Search 4 . Jump Search 5 . Linear Search

Binary Search

class pythorn.data_structures.searching.**BinarySearch**(*array=None*, *key=None*)

Binary Search Implementation

binary_search()

Implementation of Binary Search Recursive Method

static get_code()

Returns source code

static time_complexity()

Returns time complexity

Fibonacci Search

class pythorn.data_structures.searching.**FibonacciSearch**(*array=None*, *key=None*)

Fibonacci Search Implementation

fibonacci_search()

Returns index of key if its found in the sorted list else returns false

static get_code()

Returns source code

static time_complexity()

Returns time complexity

Interpolation Search

class pythorn.data_structures.searching.**InterpolationSearch**(*array=None*, *key=None*)
Interpolation Search Implementation

interpolation_search()

Returns index of key if its found in the sorted list else returns false

static get_code()

Returns source code

static time_complexity()

Returns time complexity

Jump Search

class pythorn.data_structures.searching.**JumpSearch**(*array=None*, *key=None*)
Jump Search Implementation

jump_search()

Returns index of key if its found in the sorted list else returns false

static get_code()

Returns source code

static time_complexity()

Returns time complexity

Linear Search

class pythorn.data_structures.searching.**LinearSearch**(*array=None*, *key=None*)
Linear Search Implementation

linear_search()

Returns index of key if its found in the list else returns false

static get_code()

Returns source code

static time_complexity()

Returns time complexity

1.6 Sorting Algorithms

1.6.1 Quick Start Guide

`Sorting Algorithms` :- A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

Following are the basic sorting algorithms

- **Bubble Sort**
  - Bubble sort is an algorithm that compares the adjacent elements and swaps their positions if they are not in the intended order. The order can be ascending or descending.
  - **How Bubble Sort Works?**
    - Compare two items.
    - If the one on the left is bigger, swap them.
    - Move one position right.
    - **Linear Search Algorithm::**

      bubbleSort(array)

        for i = 1 to indexOfLastUnsortedElement-1

          if leftElement > rightElement swap leftElement and rightElement

      end bubbleSort
  - **Optimized Bubble Sort**
    - In the simple bubble sort algorithm, all possible comparisons are made even if the array is already sorted which increases the execution time.
    - The code can be optimized by introducing an extra variable `swapped` so After each iteration, if there is no swapping taking place then, there is no need for performing further iterations.
    - In such a case, variable swapped is set false. Thus, we can prevent further iterations.
    - **Optimized Algorithm::**

      bubbleSort(array) swapped = false for i = 1 to indexOfLastUnsortedElement-1

        if leftElement > rightElement swap leftElement and rightElement
          swapped = true

      end bubbleSort
  - **Efficiency**
    - For 10 data items, this is 45 comparisons `(9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1)`.
    - In general, where `N` is the number of items in the array, there are `N-1` comparisons on the first pass, `N-2` on the second, and so on. The formula for the sum of such a series is `(N-1) + (N-2) + (N-3) + ... + 1 = N*(N-1)/2 N*(N-1)/2` is 45 (10*9/2) when `N` is 10.
- **Insertion Sort**
  - Insertion sort works similarly as we sort cards in our hand in a card game.

- We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put at their right place.
- A similar approach is used by insertion sort.
- Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.
- In most cases the insertion sort is the best of the elementary sorts as described above . It still executes in O(N2) time, but it's about twice as fast as the bubble sort and somewhat faster than the selection sort in normal situations. It's also not too complex, although it's slightly more involved than the bubble and selection sorts. It's often used as the final stage of more sophisticated sorts, such as quicksort.
- **How Insertion Sort Works?**
    - The first element in the array is assumed to be sorted. Take the second element and store it separately in `key` variable
    - Compare `key` variable with the first element. If the first element is greater than `key`, then key is placed in front of the first element
    - At this stage first two elements are sorted
    - Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array
    - Place every unsorted element at its correct position till end
    - **Insertion Algorithm::**

      insertionSort(array) mark first element as sorted for each unsorted element X

      store the element X in a variable key for j = lastSortedIndex down to 0

      if current element j > X move sorted element to the right by 1

      break loop and insert X here

      end insertionSort
    - **Efficiency**
        - How many comparisons and copies does this algorithm require? On the first pass, it compares a maximum of one item. On the second pass, it's a maximum of two items, and so on, up to a maximum of N-1 comparisons on the last pass. This is 1 + 2 + 3 + . . . + N-1 = N*(N-1)/2
        - However, because on each pass an average of only half of the maximum number of items are actually compared before the insertion point is found, we can divide by 2, which gives N*(N-1)/4
        - The number of copies is approximately the same as the number of comparisons. However, a copy isn't as time-consuming as a swap, so for random data this algorithm runs twice as fast as the bubble sort and faster than the selection sort.
        - In any case, like the other sort routines, the insertion sort runs in O(N2) time for random data.
        - For data that is already sorted or almost sorted, the insertion sort does much better. When data is in order, the condition in the while loop is never true, so it becomes a simple statement in the outer loop, which executes N-1 times. In

this case the algorithm runs in O(N) time. If the data is almost sorted, insertion sort runs in almost O(N) time, which makes it a simple and efficient way to order a file that is only slightly out of order.

- **Selection Sort**
  - Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.
  - The selection sort improves on the bubble sort by reducing the number of swaps necessary from O(N2) to O(N). Unfortunately, the number of comparisons remains O(N2). However, the selection sort can still offer a significant improvement for large records that must be physically moved around in memory, causing the swap time to be much more important than the comparison time.
  - **How Selection Sort Works?**
    - Set the first element as `minimum`
    - Compare `minimum` with the 2nd element
    - If the 2nd element is smaller than `minimum`, assign the second element as `minimum`
    - Compare `minimum` with the third element
    - Again, if the third element is smaller, then assign `minimum` to the third element otherwise do nothing,The process goes on until the last element
    - After each iteration, `minimum` is placed in the front of the unsorted list
    - For each iteration, indexing starts from the first unsorted element
    - So bascially the list is divided into two parts: the sublist of items already sorted, which is built up from left to right and is found at the beginning, and the sublist of items remaining to be sorted, occupying the remainder of the array
    - **Selection Sort Algorithm::** selectionSort(array, size) repeat (size - 1) times

      set the first unsorted element as the minimum for each of the unsorted elements

      if element < currentMinimum set element as new minimum

      swap minimum with first unsorted position

      end selectionSort
    - **Efficiency**
      - The selection sort performs the same number of comparisons as the bubble sort: N*(N-1)/2. For 10 data items, this is 45 comparisons
      - However, 10 items require fewer than 10 swaps. With 100 items, 4,950 comparisons are required, but fewer than 100 swaps
      - For large values of N, the comparison times will dominate, so we would have to say that the selection sort runs in O(N2) time, just as the bubble sort did

- **Counting Sort**
  - Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array
  - **How Counting Sort Works?**
    - Find out the maximum element (let it be max) from the given array

        * Initialize an array of length `max+1` with all elements 0. This array is used for storing the count of the elements in the array
        * Store the count of each element at their respective index in count array
        * For example: if the count of element 3 is 2 then, 2 is stored in the 3rd position of count array. If element "5" is not present in the array, then 0 is stored in 5th position
        * Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array
        * Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated
        * After placing each element at its correct position, decrease its count by one.

    - **Counting Sort Algorithm::**

        countingSort(array, size) max : find largest element in array initialize count array with all zeros for j <- 0 to size

            find the total count of each unique element and store the count at jth index in count array

            for i <- 1 to max find the cumulative sum and store it in count array itself

            for j <- size down to 1 restore the elements to array decrease count of each element restored by 1

    - **Efficiency**

        * Counting sorts fail when there are large key values (the k in the O(n)). This means that if you have a large variety of key values, counting sort will be slow
        * Radix sort can help solve that problem but it does nothing for other issue
        * Both counting and radix sort are only valid for integer keys
        * While not a terribly serious limitation, it does mean that Radix Sort's value for the number of digits in a key should not be considered constant

- **Merge Sort**

    - mergesort is a much more efficient sorting technique than those we saw in above section, at least in terms of speed. While the bubble, insertion, and selection sorts take `O(N2)` time, the mergesort is `O(N*logN)`
    - For example, if `N` (the number of items to be sorted) is `10,000`, then `N2` is `100,000,000`, while``N*logN`` is only `40,000`
    - If sorting this many items required 40 seconds with the mergesort, it would take almost 28 hours for the insertion sort.
    - Merge Sort uses Divide and Conquer Strategy
    - Using the Divide and Conquer technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.
    - The mergesort is also fairly easy to implement. It's conceptually easier than quicksort and the Shell short.
    - The heart of the mergesort algorithm is the merging of two already-sorted arrays. Merging two sorted arrays `A` and `B` creates a third array, `C`, that contains all the elements of `A` and `B`, also arranged in sorted order.

- Similar to quicksort the list of element which should be sorted is divided into two lists. These lists are sorted independently and then combined. During the combination of the lists the elements are inserted (or merged) on the correct place in the list
- You divide the half into two quarters, sort each of the quarters, and merge them to make a sorted half
- **How Mergesort Works?**
    - Assume the size of the left array is k, the size of the right array is m and the size of the total array is n (=k+m).
    - Create a helper array with the size n
    - Copy the elements of the left array into the left part of the helper array. This is position 0 until k-1.
    - Copy the elements of the right array into the right part of the helper array. This is position k until m-1.
    - Create an index variable i=0; and j=k+1
    - Loop over the left and the right part of the array and copy always the smallest value back into the original array. Once i=k all values have been copied back the original array. The values of the right array are already in place.
- **Efficiency**
    - As we noted, the mergesort runs in O(N*logN) time. There are `24` copies necessary to sort `8` items. `Log28` is `3`, so `8*log28` equals `24`. This shows that, for the case of 8 items, the number of copies is proportional to `N*log2N`
    - In the mergesort algorithm, the number of comparisons is always somewhat less than the number of copies.

- **Quick Sort**
    - Quicksort is an algorithm based on divide and conquer approach in which the array is split into subarrays and these sub-arrays are recursively called to sort the elements.
    - Quicksort is a divide-and-conquer algorithm that involves choosing a pivot value from a data-set and splitting the set into two subsets: a set that contains all values less than the pivot and a set that contains all values greater than or equal to the pivot. The pivot/split process is recursively applied to each subset until there are no more subsets to split. The results are combined to form the final sorted set.
    - The challenge of a quicksort is to determine a reasonable midpoint value for dividing the data into two groups. The efficiency of the algorithm is entirely dependent upon how successfully and accurate the midpoint value is selected
    - Quicksort is undoubtedly the most popular sorting algorithm, and for good reason: In the majority of situations, it's the fastest, operating in `O(N*logN)` time. (This is only true for internal or in-memory sorting; for sorting data in disk files, other algorithms may be better)
    - To understand quicksort, you should be familiar with the partitioning algorithm.
    - **How Quick Sort Works?**
        - If the array contains only one element or zero elements then the array is sorted.
        - If the array contains more then one element then:
        - Select an element from the array. This element is called the "pivot element". For example select the element in the middle of the array.

* All elements which are smaller then the pivot element are placed in one array and all elements which are larger are placed in another array.
* Sort both arrays by recursively applying Quicksort to them.
* Combine the arrays.
* Quicksort can be implemented to sort "in-place". This means that the sorting takes place in the array and that no additional array need to be created.

– **Quick Sort Algorithm::**

quickSort(array, leftmostIndex, rightmostIndex)

if (leftmostIndex < rightmostIndex) pivotIndex = partition(array,leftmostIndex, rightmostIndex) quickSort(array, leftmostIndex, pivotIndex) quickSort(array, pivotIndex + 1, rightmostIndex)

partition(array, leftmostIndex, rightmostIndex) set rightmostIndex as pivotIndex storeIndex = leftmostIndex - 1 for i <- leftmostIndex + 1 to rightmostIndex if element[i] < pivotElement

swap element[i] and element[storeIndex] storeIndex++

swap pivotElement and element[storeIndex+1] return storeIndex + 1

– **Efficiency**

* Quicksort operates in `O(N*logN)` time. This is generally true of the divide-and-conquer algorithms, in which a recursive method divides a range of items into two groups and then calls itself to handle each group.
* In this situation the logarithm actually has a base of `2:` The running time is proportional to `N*log2N`

Time Complexity And Space Complexity

Sorting Algorithms	Time Complexity			Space Complexity
Sorting	Best	Average	Worst	Worst
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(k)
Merge Sort	O(nLogn)	O(nLogn)	O(nLogn)	O(n)
Heap Sort	O(nLogn)	O(nLogn)	O(nLogn)	O(1)
Quick Sort	O(nLogn)	O(nLogn)	O(n^2)	O(nLogn)
ShellSort	O(n)	O((nLogn)^2)	O((nLogn)^2)	O(1)

```
# import required algorithm
>>> from package.pygostructures.data_structures.sorting import BubbleSort

# create array
>>> my_array = [4,1,77,2,3,0,99,100,65]

# pass the array as argument
>>> b = BubbleSort(my_array)

# call the function
```

(continues on next page)

(continued from previous page)

```
>>> b.bubblesort()
Sorted Array :  [0, 1, 2, 3, 4, 65, 77, 99, 1
```

1.6.2 Sorting Algorithms

Author : Robin Singh

Programs List :

1 . Bubble Sort 2 . Counting Sort 3 . Insertion Sort 4 . Merge Sort 5 . Quick Sort 6 . Selection Sort 7 . Shell Sort 8 . Heap Sort

Bubble Sort

class pythorn.data_structures.sorting.**BubbleSort**(*array=None*)

Bubble Sort Implementation

bubblesort()

Returns sorted values

static get_code()

returns source code of the current class

Returns source code

static time_complexity()

returns time complexity of the functions

Returns string

Counting Sort

class pythorn.data_structures.sorting.**CountingSort**(*array=None*)

Counting Sort Implementation

counting_sort()

Returns sorted values

static get_code()

Returns source code

static time_complexity()

Returns time complexity

Insertion Sort

class pythorn.data_structures.sorting.**InsertionSort**(*array=None*)
Insertion Sort Implementation

insertion_sort()

Returns sorted values

static get_code()

Returns source code

static time_complexity()

Returns time complexity

Merge Sort

class pythorn.data_structures.sorting.**MergeSort**(*array*)
Merge Sort Implementation

merge_sort()
function to sort an array using merge sort algorithm
Returns sorted values

static get_code()

Returns source code

static time_complexity()

Returns time complexity

Quick Sort

Example :

```
# import required algorithm
>>> from pythorn.data_structures.sorting import QuickSort

# create array
>>> f = [2,6,9,10,100,20,3,6,9,78,98,2,1500]

# pass array , value zero and len(array)-2 as an argument
>>> b = QuickSort(f,0,len(f)-2)
>>> b.quick_sort()

# call the function
[2, 2, 3, 6, 6, 9, 9, 10, 20, 78, 98, 100, 1500]
```

class pythorn.data_structures.sorting.**QuickSort**(*array=None*, *lower=None*, *upper=None*)
Quick Sort Implementation

static partition(*a*, *lb*, *ub*)
function for partition

quick_sort()

Returns sorted values of the given array

static get_code()

Returns source code

static time_complexity()

Returns time complexity

Selection Sort

class pythorn.data_structures.sorting.**SelectionSort**(*array=None*)
Selection Sort Implementation

selection_sort()

Returns sorted values of array

static get_code()

Returns source code

static time_complexity()

Returns time complexity

Shell Sort

class pythorn.data_structures.sorting.**ShellSort**(*array=None*)
Shell Sort Implementation

shell_sort()

Returns sorted values of array

static get_code()

Returns source code

static time_complexity()

Returns time complexity

HeapSort

class pythorn.data_structures.sorting.**HeapSort**(*array=None*)
Heap Sort Implementation

static heapify(*a*, *n*, *i*)
funtion for converting a binary tree into a Heap data structure

heapSort()

Returns sorted values of array

static get_code()

Returns source code

static time_complexity()

Returns time complexity

1.7 Trees

1.7.1 Quick Start Guide

```
# import required algorithm
>>> from package.pygostructures.data_structures.trees import BinarySearchTree

# creating BST
>>> bst = BinarySearchTree()

# insert value to be inserted
>>> bst.insert(5)
>>> bst.insert(7)
>>> bst.insert(3)
>>> bst.insert(6)
>>> bst.insert(20)
>>> bst.insert(19)
>>> bst.insert(0)
>>> bst.insert(1)
>>> bst.insert(100)
>>> bst.insert(111)

# check if give argument is present in the bst
>>> bst.search(19)
True
>>> bst.search(50)
False

# performs inorder traversal
>>> bst.inorder(bst.root)
0-->1-->3-->5-->6-->7-->19-->20-->100-->111

# performs preorder traversal
>>> bst.preorder(bst.root)
5-->3-->0-->1-->7-->6-->20-->19-->100-->111

# performs postorder
>>> bst.postorder(bst.root)
1-->0-->3-->6-->19-->111-->100-->20-->7-->5
```

1.7.2 Trees Programs

Author : Robin Singh

Binary Search Tree

class pythorn.data_structures.trees.**BinarySearchTree**

Binary Search Tree Funtion

Parameters **root** – root node

insert(*ele*)

inserts a node into the tree

search(*k*)

Function for searching an given element

Returns true if element is present else return false

inorder(*temp*)

here we first traverse to the leftmost node and then print the data and then move to the rightmost child

Parameters **temp** – root node

preorder(*temp*)

here we first print the root node and then traverse towards leftmost node and then to the rightmost node

Parameters **temp** – root node

postorder(*temp*)

here we first traverse to the leftmost node and then to the rightmost node and then print the data

Parameters **temp** – root node

static get_code()

Returns source code

static time_complexity()

Returns time complexity

1.8 Graphs

1.8.1 Quick Start Guide

```
# import required data structures
>>> from pythorn.data_structures.graphs import AdjanceyList

# pass number of nodes
>>> a = AdjanceyList(5)

# add edge with source node and destination node
>>> a.add_edge(0,1)
>>> a.add_edge(0,3)
>>> a.add_edge(4,3)
>>> a.add_edge(2,4)

# printing Adjancey List
>>> a.print_list()
```

(continues on next page)

(continued from previous page)

```
node 1 -> 5
node 2 -> 4
node 3 -> 0 4
node 4 -> 3 2
node 5 -> 5 3
```

1.8.2 Graph Programs

Author : Robin Singh

Programs List : 1 . Adjancey List with Node Class 2 . Adjancey Matrix 3 . Breath First Search 4 . Depth First Search 5 . Topological Sort

Adjancey List

class pythorn.data_structures.graphs.**AdjanceyList**(*number_of_nodes*)

Adjancey list implementation

create_nodes()

function for creating nodes

last_node(*first_node*)

function for getting the last node

add_edge(*node1*, *node2*)

function for adding new edges

Parameters

- **node1** – from vertex (source)
- **node2** – to vertex (destination)

print_adjancey(*node*, *node_No*)

Prints prints adjancey list for a specific node

print_list()

function for printing all the node's adjancey list

static get_code()

Returns source code

static time_complexity()

Returns time complexity

Adjancey Matrix

Example :

```
>>> from pythorn.data_structures.graphs import AdjanceyMatrix
>>> a = AdjanceyMatrix(5)
>>> a.add_edge(1,5)
>>> a.add_edge(1,4)
>>> a.add_edge(2,4)
>>> a.add_edge(3,4)
>>> a.add_edge(3,2)
>>> a.add_edge(3,1)
>>> a.print_matrix()
[0, 0, 1, 1, 1]
[0, 0, 1, 1, 0]
[1, 1, 0, 1, 0]
[1, 1, 1, 0, 0]
[1, 0, 0, 0, 0]
[1, 0, 0, 0, 0]
```

class pythorn.data_structures.graphs.**AdjanceyMatrix**(*no_of_verices*)

Adjancey Matrix Implementation

make_matrix()

function for making matrix of NxN. where n is the number of vertices

add_edge(*node1*, *node2*)

function for adding an edge

Parameters

- **node1** – vertex 1 (source)
- **node2** – vertex 2 (destination)

print_matrix()

function for printing the matrix

static get_code()

Returns source code

static time_complexity()

Returns time complexity

BFS

Example :

```
# import required algorithm
>>> from pythorn.data_structures.graphs import BFS

# create graph
>>> graph={  0: [1, 3,4],
...             1: [2],
...             2: [3],
...             3: [1,4],
```

(continues on next page)

(continued from previous page)

```
...             4: [0,2] }

# pass graph and value 0 as argument
>>> bfs1 = BFS(graph,0)

# call bfs main function
>>> z = bfs1.bfs()

>>> print(z)
[0, 1, 3, 4, 2]
```

class pythorn.data_structures.graphs.**BFS**(*graph*, *start*)

BFS implementation with adjancey list

bfs()

Returns BFS path

static get_code()

Returns source code

static time_complexity()

Returns time complexity

DFS

Example :

```
# import required algorithm
>>> from package.pygostructures.data_structures.graphs import DFS

# create graph
>>> graph1 = {
...             0: [1, 3,4],
...             1: [2],
...             2: [3],
...             3: [1,4],
...             4: [0,2]}

# pass graph and start vertex as argument
>>> dfs1 = DFS(graph1,2)

# call the function
>>> z = dfs1.dfs()

>>> print(z)
[2, 3, 1, 4, 0]
```

class pythorn.data_structures.graphs.**DFS**(*graph*, *start*, *path=[]*)

DFS implementation with adjancey list

dfs()

Returns DFS path

static get_code()

Returns source code

static time_complexity()

Returns time complexity

Topological Sort

Example :

```
# import required algorithm
>>> from package.pygostructures.data_structures.graphs import *

# pass number of vertices as argument
>>> a = TopologicalSort(5)

# adding edge with source and destination vertex
>>> a.add_edge(0,3)
>>> a.add_edge(0,4)
>>> a.add_edge(0,3)
>>> a.add_edge(2,3)
>>> a.add_edge(2,1)
>>> a.add_edge(2,4)


# printing list
>>> a.print_list()
0 Vertex :  ->  3 -> 4 -> 3
3 Vertex :  ->
2 Vertex :  ->  3 -> 1 -> 4

# call main function
>>> a.topological_Sort()
2--> 0--> 3
```

class pythorn.data_structures.graphs.**TopologicalSort**(*vertices*)

Topological Sort Implementation

print_list()

function for printing graph

add_edge(*node1*, *node2*)

Parameters

- **node1** – from vertex (source)
- **node2** – to vertex (destination)

static get_code()

Returns source code

static time_complexity()

Returns time complexity

1.9 Dynamic Programming

1.9.1 Quick Start Guide

1.9.2 Dynamic Programming

Author : Robin Singh

Programs List :

1 . Bellman Ford 2 . Floyd Warshall 3 . Longest Common Subsequence 4 . Coin Change Problem 1 5 . Coin Change Problem 2 6 . Subset Sum

Bellman Ford

Implementation of Bellman Ford Algorithm(Dynamic Programming) :

Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph.

It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights.

Example :

```
# import required algorithm
>>> from pythorn.algorithms.dynamic_programming import BellmanFord

# create graph
>>> graph = {
...          'a':{'b':6,'c':4,'d':5},
...          'b':{'e':-1},
...          'c':{'e':3,'b':-2},
...          'd':{'c':-2,'f':-1},
...          'e':{'f':3},
...          'f':{}
...      }

# pass graph and source node
>>> a = BellmanFord(graph,'a')

>>> a.bellman_ford()
distances from source {'a': 0, 'b': 1, 'c': 3, 'd': 5, 'e': 0, 'f': 3}
predecssor vertices {'a': None, 'b': 'c', 'c': 'd', 'd': 'a', 'e': 'b', 'f': 'e'}
```

class pythorn.algorithms.dynamic_programming.**BellmanFord**(*graph*, *source*)

Implementation of Bellman Ford Algorithm(Dynamic Programming) , Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph. It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights.

bellman_ford()

Prints prints all the calculated distances from source and predecssor vertices

static time_complexity()

Returns time complexity

static get_code()

Returns source code

Floyd Warshall

Implementation of Floyd Warshall's Algorithm:

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph

This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles

Example :

```
# import required algorithm
>>> from pythorn.algorithms.dynamic_programming import FloydWarshall

# initialize a infinity value
>>> INF = 999999999999

# create a graph in matrix form
>>> G = [[0, 9, -4, INF],
...           [6, 0, INF, 2],
...           [INF, 5, 0, INF],
...           [INF, INF, 1, 0]]

# pass graph and no of vertices as an argument
>>> a = FloydWarshall(G,4)
>>> a.floyd_warshall()
0  1  -4  3
6  0  2  2
11  5  0  7
12  6  1  0
```

class pythorn.algorithms.dynamic_programming.**FloydWarshall**(*graph*, *vertex*)

Implementation of Floyd Warshall's Algorithm

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph

This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles

floyd_warshall()

Prints prints final matrix having shortest path between all the pairs of vertices

static time_complexity()

Returns time complexity

static get_code()

Returns source code

Longest Common Subsequence

Implementation Of LCS(Dynamic Approch):

Given two sequences,here we have to find the length of longest subsequence present in both of them

Example

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

Example :

```
# import required algorithm
>>> from pythorn.algorithms.dynamic_programming import LongestCommonSubsequence

# entre string1 and string2
>>> string1 = "ABCDEFABC"
>>> string2 = "CDASCVCE"

# pass both the strings as argument
>>> a = LongestCommonSubsequence(string1,string2)
>>> a.longest_common_subsequence()
Lenth Of Sequence : 4, And Sequence is : CDAC
```

class pythorn.algorithms.dynamic_programming.**LongestCommonSubsequence**(*string1*, *string2*)

Implementation Of LCS(Dynamic Approch)

Given two sequences,here we have to find the length of longest subsequence present in both of them

Example :

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

longest_common_subsequence()

Prints prints longest common subsequence with length of subsequence

static time_complexity()

Returns time complexity

static get_code()

:return:source code

Subset Sum

Implementation of Subset Sum:

Implementation of Subset Sum problem which will return true if at least one sub set exists of the required sum

Example :

```
# import required algorithm
>>> from package.pygostructures.algorithms.dynamic_programming import SubsetSum

# create a array
```

(continues on next page)

(continued from previous page)

```
>>> array = [2,9,7,6,3,4,15,12,32]

# pass array and value
>>> a = SubsetSum(array,14)
>>> a.subset_sum()
Yes,There Exists At least One Sub-Set whose sum of the elements is 14
```

class pythorn.algorithms.dynamic_programming.**SubsetSum**(*array*, *sum*)

Implementation of Subset Sum

Implementation of Subset Sum problem which will return true if at least one sub set exists of the required sum

subset_sum()

Prints prints true if subset exists else prints flase

static time_complexity()

Returns time complexity

static get_code()

Returns source code

Coin Change Problem 1

Coin Exchange:

Implementaion of Number Of Coins Change(Number Of ways to get required Sum)

Example :

```
# import required algorithm
>>> from pythorn.algorithms.dynamic_programming import CoinChange01

# pass amount value
>>> a = CoinChange01(90)
>>> a.coin_change()
Number Of Ways To get Sum 90 = 559
```

class pythorn.algorithms.dynamic_programming.**CoinChange01**(*sum*)

Implementaion of Number Of Coins Change(Number Of ways to get required Sum)

coin_change()

Prints Prints no of ways to get the required sum

static time_complexity()

Returns time complexity

static get_code()

Returns source code

Coin Change Problem 2

Coin Exchange 2:

Implementaion Of minimum number Of coins required to get the sum of given Value

Example :

```
# import required algorithm
>>> from pythorn.algorithms.dynamic_programming import CoinChange02


#Coins denominations are coins = [2, 3, 5, 10]

# pass amount value
>>> a = CoinChange02(50)
>>> a.coin_change()
Minimum Number Of Coins Required to get the sum of 50 = 5 Coins
```

class pythorn.algorithms.dynamic_programming.**CoinChange02**(*sum*)

Implementaion Of minimum number Of coins required to get the sum of given Value

coin_change()

Prints Prints no of coins to get the required sum

static time_complexity()

Returns time complexity

static get_code()

Returns source code

1.10 Greedy Algorithms

1.10.1 Quick Start Guide

Time Complexity

Greedy Algorithms	Time Complexity
Activity Selection	O(n log n) when sorted else O(n)
Dijkstra	O((E+V)Log(V))
Fractional Knapsack	O(nLogn)
Kruskal	O(ElogE) or O(ElogV)
Prims	O(ElogV)
Egyptian Fraction	—
Minimum Coin Exchange	O(N*logN)

1.10.2 Greedy Programs

Author : Robin Singh

Programs List: 1 . Activity Selection Problem 2 . Dijkstra's Algorithm 3 . Egyptian Fraction 4 . Fractional Knapsack 5 . Minimum Coin Exchange Problem 6 . Kruskals Algorithm 7 . Prims Algorithm

Activity Selection

Implementation Of Activity Selection Problem Using Greedy Approch:

> here we have n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time
>
> —Robin Singh

Example :

```
# import required algorithm
>>> from pythorn.algorithms.greedy_algorithm import ActivitySelection

# Start values
>>> start_value = [5,9,10,6,2,3,7]

# finish values
>>> finish_values = [10,9,6,10,1,3,8]

# pass start and finish values as an argument
>>> a = ActivitySelection(start_value,finish_values)
>>> a.activity_selection()
Following Activities Are selected :
Index   Start   Finish
0       5 -->   10
2       10 -->  6
3       6 -->   10
```

class pythorn.algorithms.greedy_algorithm.**ActivitySelection**(*start=None*, *finish=None*)

Implementation Of Activity Selection Problem Using Greedy Approch

here we have n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time

activity_selection()

activity selection main function

Prints Prints a maximum set of activities that can be done by a single person, one at a time

static time_complexity()

Returns time complexity

static get_code()

:return:source code

Dijkstra

Implementation of Dijkstra's Algorithm:

> Dijkstra's algorithm finds the shortest path in a weighted graph containing only positive edge weights from a single source
>
> —Robin Singh

Example :

```
# import required algorithm
>>> from pythorn.algorithms.greedy_algorithm import Dijkstra

# creating graph
>>> graph = {
...          'a':{'b':4,'h':8},
...          'b':{'a':4,'h':11,'c':8},
...          'c':{'b':8,'i':2,'d':7},
...          'd':{'e':9,'c':7,'f':14},
...          'e':{'d':9,'f':10},
...          'f':{'d':14,'e':10,'g':2},
...          'g':{'i':6,'f':2,'h':1},
...          'h':{'a':8,'b':11,'i':7,'g':1},
...          'i':{'c':2,'g':6,'h':7}
...      }

# entre source and destination node
>>> source = 'c'
>>> dest = 'h'

# pass all the 3 parameters as argument
>>> d = Dijkstra(source,dest,graph)
>>> d.dijkstra()
Single Source Shortest Path :['c', 'i', 'h'] --> COST : 9
```

class pythorn.algorithms.greedy_algorithm.**Dijkstra**(*start*, *finish*, *graph*)

Implementation of Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path in a weighted graph containing only positive edge weights from a single source

dijkstra()

Calculates the optimal path from start to end on the graph

Prints prints the optimal path if exists

Fractional Knapsack

Implementation Of Fractional Knapsack:

- Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack
- n Fractional Knapsack, we can break items for maximizing the total value of knapsack
- This problem in which we can break an item is also called the fractional knapsack problem
- An efficient solution is to use Greedy approach. The basic idea of the greedy approach is to calculate

the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with

- the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can
- Which will always be the optimal solution to this problem

—Robin Singh

Example :

```
# import required algorithm
>>> from pythorn.algorithms.greedy_algorithm import FractionalKnapsack

# create a list with profit values
>>> profit = [10,9,7,12,30,16]

# create a list with weight of the items in the same order
>>> weight = [5,7,3,10,6,7]

# capacity of the knapsack
>>> capacity = 15

# weight,profit,capacity as an argument in the same order
>>> a = FractionalKnapsack(weight,profit,capacity)

# call the function
>>> a.fractional_knapsack()
Fractional Knapsack
Profit :30              Weight  : 6             Profit/Weight : 5.0
Profit :7               Weight  : 3             Profit/Weight : 2.3333333333333335
Profit :16              Weight  : 7             Profit/Weight : 2.2857142857142856
Total Profit : 50.714285714285715
```

class pythorn.algorithms.greedy_algorithm.**FractionalKnapsack**(*weight*, *profit*, *capacity*)

Implementation Of Fractional Knapsack

- Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack
- n Fractional Knapsack, we can break items for maximizing the total value of knapsack
- This problem in which we can break an item is also called the fractional knapsack problem
- An efficient solution is to use Greedy approach. The basic idea of the greedy approach is to calculate

the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with

- the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can
- Which will always be the optimal solution to this problem

fractional_knapsack()

Prints prints maximum total value of the knapsack

static time_complexity()

Returns time complexity

static get_code()

Returns source code

Kruskal's Algorithm

Implementation of Kruskal's Algorithm:

> Minimum Cost Spanning Tree of a given connected, undirected and weighted graph
>
> It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost at each step
>
> —Robin Singh

Example :

```
# Here we have to import two classes 1st is Kruskal itself and second is Edge (for␣
↪making source ,destination edge with weight of the edge)

>>> from pythorn.algorithms.greedy_algorithm import Kruskal
>>> from pythorn.algorithms.greedy_algorithm import Edge

# Creating Edge (Source,Destination,Weight)
>>> A = Edge(1, 6, 10)
>>> B = Edge(3, 4, 12)
>>> C = Edge(7, 2, 14)
>>> D = Edge(2, 3, 16)
>>> E = Edge(7, 4, 18)
>>> F = Edge(4, 5, 22)
>>> G = Edge(5, 7, 24)
>>> H = Edge(5, 6, 25)
>>> I = Edge(1, 2, 28)

# declaring number of nodes
>>> num_nodes = 8

# pass number of nodes and list of all the edges as argument
>>> a = Kruskal(num_nodes,[A,B,C,D,E,F,G,H,I])
>>> a.kruskal()
MCST
SOURCE  DESTINATION  WEIGHT
1    -->  6     -->     10
3    -->  4     -->     12
7    -->  2     -->     14
2    -->  3     -->     16
4    -->  5     -->     22
5    -->  6     -->     25

MCST(MINIMUM COST) :  99
```

class pythorn.algorithms.greedy_algorithm.**Kruskal**(*num_nodes=None*, *edgelist=None*)
Implementation of Kruskal's Algorithm,

Minimum Cost Spanning Tree of a given connected, undirected and weighted graph

It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost at each step

kruskal()
main function
Prints prints minimum cost of the given graph

static time_complexity()

Returns time complexity

static get_code()

Returns source code

Prims

Implementation of prims algorithm:

In Prim's Algorithm, a conquered territory (initialized with any start vertex) is chosen in which we keep adding the vertices as we go through the algorithm.

To get the minimum spanning tree, we keep adding vertices to the conquered edges with the greedy paradignm that we select the edge with the minimum weight of all the edges starting the conquered territory and ending at the unconquered territory. The end of the minimum weight edge thus chosen is then added to the conquered territory and removed from the unconquered territory. In such a way, we go on till the conquered territory spans all the vertices of the graph

Example :

class pythorn.algorithms.greedy_algorithm.**Prims**(*source*, *graph*)
Implementation of prims algorithm

In Prim's Algorithm, a conquered territory (initialized with any start vertex) is chosen in which we keep adding the vertices as we go through the algorithm To get the minimum spanning tree, we keep adding vertices to the conquered edges with the greedy paradignm that we select the edge with the minimum weight of all the edges starting the conquered territory and ending at the unconquered territory. The end of the minimum weight edge thus chosen is then added to the conquered territory and removed from the unconquered territory. In such a way, we go on till the conquered territory spans all the vertices of the graph

prims()

Prints prints MCST for the given graph

static time_complexity()

Returns time complexity

static get_code()
:return:source code

Egyptian Fraction

Implementation of Eqyption Fraction:

> Every positive fraction can be represented as sum of unique unit fractions
>
> A fraction is unit fraction if numerator is 1 and denominator is a positive integer, for example 1/3 is a unit fraction
>
> Such a representation is called Egyptian Fraction as it was used by ancient Egyptians

Example :

```
>>> from pythorn.algorithms.greedy_algorithm import EgyptianFraction
>>> a = EgyptianFraction(7,19)

>>> a.egyptian_fraction()

1/3 +  1/29 + 1/1653
```

class pythorn.algorithms.greedy_algorithm.**EgyptianFraction**(*numerator*, *denominator*)
Implementation of Eqyption Fraction

Every positive fraction can be represented as sum of unique unit fractions

A fraction is unit fraction if numerator is 1 and denominator is a positive integer, for example 1/3 is a unit fraction

Such a representation is called Egyptian Fraction as it was used by ancient Egyptians

egyptian_fraction()

Prints prints the value of numnerator/denominator

static time_complexity()

Returns time complexity

static get_code()

Returns source code

Minimum Coin Exchange

Implementation of Minimum Coin Exchange Problem:

> Here we have a value V, if we want to make a change for V Rs, and we have an infinite supply of each of the denominations in Indian currency, i.e., we have an infinite supply of { 1, 2, 5, 10, 20, 50, 100, 500, 2000} valued coins/notes
>
> Then what is the minimum number of coins or notes are needed to make the change

Example :

```
# import required algorithm
>>> from pythorn.algorithms.greedy_algorithm import *

# pass money
>>> a = MinimumCoinExchange(2589)
>>> a.minimum_coin_exchange()
2  2  5  10  20  50  500  2000
```

class pythorn.algorithms.greedy_algorithm.**MinimumCoinExchange**(*money*)
Implementation of Minimum Coin Exchange Problem

Here we have a value V, if we want to make a change for V Rs, and we have an infinite supply of each of the denominations in Indian currency, i.e., we have an infinite supply of { 1, 2, 5, 10, 20, 50, 100, 500, 2000} valued coins/notes

Then what is the minimum number of coins or notes are needed to make the change

minimum_coin_exchange()

Prints prints all denominations change

static time_complexity()

Returns time complexity

static get_code()
:return:source code

1.11 String Matching

1.11.1 Quick Start Guide

1.11.2 String Matching Algorithms

Author : Robin Singh

Programs List :

1 . Knuth Morris Pratt String Matching Algorithm 2 . Naive Method 3 . Rabin Karp String Matching Algorithm

Knuth Morris Pratt

Example :

```
# import required algorithm
>>> from package.pygostructures.algorithms.string_matching import KnuthMorrisPratt

# string to be searched
>>> string1 = "csk"

# main string from which string 1 has to be searched
>>> string2 = "jadhgdajdkcsklsdhajhd"


>>> a = KnuthMorrisPratt(string1,string2)

>>> a.knuth_morris_pratt()
Pattern Found At Index :10
```

class pythorn.algorithms.string_matching.**KnuthMorrisPratt**(*string1*, *string2*)
Implementation Of KMP string Matching Algorithm

static prefix_generator(*pattern*, *m*, *n*)
utility function for generating prefix

knuth_morris_pratt()

Prints prints the index if string is matched

static time_complexity()

Returns time complexity

static get_code()

Returns source code

Naive Method

Example :

```
# import required algorithm
>>> from pythorn.algorithms.string_matching import NaiveMethod

# string to be searched
>>> string1 = "in"

# main string from which string 1 has to be searched
>>> string2 = "djhdjhdinqwert"


>>> a = NaiveMethod(string1,string2)

>>> a.naive_method()
Pattern Found At Index :7
```

class pythorn.algorithms.string_matching.**NaiveMethod**(*string1*, *string2*)

Implementation Of Naive method string Matching Algorithm

naive_method()

Prints prints index if string is matched

static time_complexity()

Returns time complexity

static get_code()

Returns source code

Rabin Karp

Example :

```
# import required algorithm
>>> from pythorn.algorithms.string_matching import RabinKarp

# string to be searched
>>> string1 = "in"

# main string from which string 1 has to be searched
>>> string2 = "djhdjhdinqwert"
```

(continues on next page)

(continued from previous page)

```
>>> a = RabinKarp(string1,string2)

>>> a.rabin_karp()
Pattern Found At Index :7
```

class pythorn.algorithms.string_matching.**RabinKarp**(*string1*, *string2*)

Implementation Of Rabin Karp method string Matching Algorithm

rabin_karp()

Prints prints index if string exists

static time_complexity()

Returns time complexity

static get_code()

:return:source code

CHAPTER

TWO

GETTING STARTED

- For getting started, first download the package using Python package manager

```
pip install pythorn
```

- Or you can download the source code from here, and then just install the package using

```
python setup.py install
```

PYTHON MODULE INDEX

p

INDEX

E

F

G

H

I

J

K