python-algorithms Documentation

Release

Arseniy Antonov

Apr 19, 2018

Contents

python-algorithms project is a collection of algorithms implemented on `Python3.6` You don't need to install these project as a module (via pip) because usually you just need only one algorithm instead of all pack, so just copy and paste the source code. For easy navigation please use links to the source code below.

CHAPTER 1

algorithms.arithmetic

1.1 GCD

Greatest common divisor (gcd) of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers.

gcd (**integer_nums: int*) → int

Function for calculating GCD [greatest common divisor] of N integers

Parameters ***integer_nums** – integer arguments

Returns Greatest common divisor of N positive integers

Examples

```
>>> gcd(54, 24)
6
```

```
>>> gcd(2, 4, 6, 8, 16)
2
```

1.2 LCM

The least common multiple, lowest common multiple, or smallest common multiple of two integers a and b, usually denoted by LCM(a, b), is the smallest positive integer that is divisible by both a and b. [Wikipedia]

lcm (**integer_nums: int*) → int

Private function for calculating LCM [least common multiple] of N integers

Parameters ***integer_nums** – integer arguments

Returns Least common multiple of N positive integers.

Examples

```
>>> lcm(16, 20)
80
```

```
>>> lcm(8, 9, 21)
504
```

CHAPTER 2

algorithms.graphs

get_edges (*graph*)
Function for calculating number of edges

Parameters **graph** – graph representation as Example: {1: {2: 1, 3: 5}, 2: {3: 2}, 4: {1: 2}}

Returns edges of the graph

get_nodes (*graph*)
Function for calculating number of nodes in the graph

Parameters **graph** – graph representation as Example: {1: {2: 1, 3: 5}, 2: {3: 2}, 4: {1: 2}}

Returns nodes of the graph

num_of_edges (*graph*)
Function for calculating number of edges

Parameters **graph** – graph representation as Example: {1: {2: 1, 3: 5}, 2: {3: 2}, 4: {1: 2}}

Returns number of the edges in the graph

num_of_nodes (*graph*)
Function for calculating number of nodes in the graph

Parameters **graph** – graph representation as Example: {1: {2: 1, 3: 5}, 2: {3: 2}, 4: {1: 2}}

Returns number of nodes in the graph

prepare_direct_graph (*edges*)
Function for converting list of edges to dict graph representation

Parameters **edges** – list of tuples, example: [(1,2), (2,3)]

Returns that represent graph

Return type Defaultdict(list)

Examples

```
>>> prepare_direct_graph([(1, 2), (3, 4), (2, 4), (2, 3)])
defaultdict(list, {1: [2], 2: [1, 3, 4], 3: [2, 4], 4: [2, 3]})
```

prepare_undirect_graph(*edges*)

Function for converting list of edges to dict graph representation

Parameters **edges** – list of tuples, example: [(1,2), (2,3)]

Returns that represent graph

Return type Defaultdict(list)

Examples

```
>>> prepare_undirect_graph([(1, 2), (3, 4), (2, 4), (2, 3)])
defaultdict(list, {1: [2], 2: [1, 3, 4], 3: [2, 4], 4: [2, 3]})
```

prepare_weighted_direct_graph(*edges*)

Function for conveting list of edges with weights to dict graph representation :param edges: list of tuples, example [(1, 2, 3), (3, 2, 1)]; [(node_a, node_b, weight)]

Returns

```
>>> prepare_weighted_direct_graph([(1, 2, 1), (4, 1, 2), (2, 3, 2),␣
↪(1, 3, 5)])
defaultdict(dict, {1: {2: 1, 3: 5}, 2: {3: 2}, 4: {1: 2}})
```

prepare_weighted_undirect_graph(*edges*)

Function for conveting list of edges with weights to dict graph representation :param edges: list of tuples, example [(1, 2, 3), (3, 2, 1)]; [(node_a, node_b, weight)]

Returns

```
>>> prepare_weighted_undirect_graph([(1, 2, 1), (4, 1, 2), (2, 3, 2),␣
↪(1, 3, 5) ])
defaultdict(<class 'dict'>, {1: {2: 1, 4: 2, 3: 5}, 2: {1: 1, 3: 2},␣
↪4: {1: 2}, 3: {2: 2, 1: 5}})
```

reverse_graph(*graph*)

Function for reverting direction of the graph

Parameters **graph** – graph representation as Example: [(1, 2), (3, 4), (2, 4), (2, 3)]

Returns reversed graph

Examples

```
>>> reverse_graph({1: [2], 2: [1, 3, 4], 3: [2, 4], 4: [2, 3]})
defaultdict(<class 'list'>, {2: [1, 3, 4], 1: [2], 3: [2, 4], 4: [2, 3]})
```

reverse_weighted_graph(*graph*)

Function for reverting direction of the graph (weights still the same)

Parameters **graph** – graph representation as Example: {1: {2: 1, 3: 5}, 2: {3: 2}, 4: {1: 2}}

Returns reversed graph

Examples

```
>>> reverse_weighted_graph({1: {2: 1, 3: 5}, 2: {3: 2}, 4: {1: 2}})
defaultdict(<class 'dict'>, {2: {1: 1}, 3: {1: 5, 2: 2}, 1: {4: 2}})
```

2.1 BFS

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key') and explores the neighbor nodes first, before moving to the next level neighbours. [Wikipedia]

Worst-case performance: O(V + E) = O(b^d)

Worst-case space complexity O(V) = O(b^d)

https://wikipedia.org/wiki/Breadth-first_search

`bfs_iterative`(*graph*, *root*)
Iterative version of the BFS algorithm

Parameters

- **`graph`** – dict representation of the graph
- **`root`** – start point

Returns in order of visiting vertices

Return type list

Examples:

2.2 DFS

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key') and explores the neighbor nodes first, before moving to the next level neighbours. [Wikipedia]

Worst-case performance: O(V + E) = O(b^d)

Worst-case space complexity O(V) = O(b^d)

https://wikipedia.org/wiki/Breadth-first_search

`bfs_iterative`(*graph*, *root*)
Iterative version of the BFS algorithm

Parameters

- **`graph`** – dict representation of the graph
- **`root`** – start point

Returns in order of visiting vertices

Return type list

Examples:

2.3 Dijkstra

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later [Wikipedia]

Worst-case Performance: O(**|E|+|V|** log **|V|**)

dijkstra (*graph*, *start*, *target*)

Solves shortest path problem using Dijkstra algorithm :param graph: graph representation :param start: start node :param target: target node

Returns distance between start and target nodes

Return type int

Examples

```
>>> graph = prepare_weighted_undirect_graph(
[(1, 2, 7), (1, 3, 9), (1, 6, 14), (6, 3, 2), (6, 5, 9), (3, 2, 10), (3, 4, 11),
(2, 4, 15), (6, 5, 9), (5, 4, 6)])
```

```
>>> dijkstra(graph, 1, 6)
11
```

2.4 Bidirectional Dijkstra

Bidirectional search is a graph search algorithm that finds a shortest path from an initial vertex to a goal vertex in a directed graph. It runs two simultaneous searches: one forward from the initial state , and one backward from the goal, stopping when the two meet in the middle. [Wikipedia]

bidi_dijkstra (*graph*, *start*, *target*)

Calculate shortest path via Dijkstra algorithm, with bidirectional optimization which means that we start from target and start points and swith between them each step

Parameters

- **graph** – graph representation
- **start** – start node
- **target** – target node

Returns lengths of the shortest path between start and target nodes

Return type int

Examples: >>> graph = prepare_weighted_undirect_graph([(1, 2, 7), (1, 3, 9), (1, 6, 14), (6, 3, 2), (6, 5, 9), (3, 2, 10), (3, 4, 11), (2, 4, 15), (6, 5, 9), (5, 4, 6)])

```
>>> dijkstra(graph, 1, 6)
11
```

2.5 Cycle detection (DFS)

2.6 AStar

class AStar(*n*, *adj*, *cost*, *x*, *y*)

Bases: `object`

clear()

potential(*v*, *t*)

query(*s*, *t*)

visit(*q*, *v*, *dist*, *measure*)

readl()

2.7 Bellman-Ford (shortest path with negative arbitrage)

2.8 Kruskal

2.9 Bellman-Ford (with negative cycle detection)

2.10 Bipartite

In the mathematical field of graph theory, a bipartite graph (or bigraph) is a graph whose vertices can be divided into two disjoint and independent sets U and V such that every edge connects a vertex in U to one in V. Vertex sets U and V are usually called the parts of the graph. Equivalently, a bipartite graph is a graph that does not contain any odd-length cycles. [Wikipedia]

bipartite(*graph*)

Function checks if graph is bipartite

Parameters **graph** – graph representation

Returns True if bipartite , False otherwise

Return type bool

2.11 BST check

Function tor check if the tree is correct Binary Search Tree (BST)

check_if_bst(*node*, *mini=-inf*, *maxi=inf*)

Check if the given tree is Binary Search Tree (BST)

Parameters

- **node** – root node of the Tree. *node* arg must have *.left*, *.right* and *.data* variables
- **mini** – min value - should be omitted
- **maxi** – max value - should be omitted

Returns bool - True if it's BST and False if not

Examples

Precondition:

```
>>> class Node:
...     def __init__(self, data):
...         self.data = data
...         self.left = None
...         self.right = None
>>> root = Node(4)
>>> root.left = Node(2)
>>> root.right = Node(6)
>>> root.left.left = Node(1)
>>> root.left.right = Node(3)
>>> root.right.left = Node(5)
>>> root.right.right = Node(7)
```

Example itself:

```
>>> check_if_bst(root)
True
```

2.12 Strongly connected components

In the mathematical theory of directed graphs, a graph is said to be strongly connected or diconnected if every vertex is reachable from every other vertex.

class StronglyConnected(*graph*)
Bases: `object`

strongly_connected_components()
Function finds, and return list of SCC in the graph

Returns SCC

Return type list

Examples

```
>>> graph = prepare_direct_graph([(0, 2), (2, 1), (1, 0), (0, 3), (3, 4)])
>>> StronglyConnected(graph).strongly_connected_components()
[[0, 1, 2], [3], [4]]
```

2.13 Topological sort

In the field of computer science, a topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering. [Wikipedia]

class TopologicalSort(*edges*)
Bases: `object`

sort()

Function do a topological sorting Returns: topologically sorted list of vertices

CHAPTER 3

algorithms.greedy

3.1 Covering segments

Given a set of n segments {[a(0) ,b(0)],[a(1) ,b(1)],...,[a(n)1 ,b(n)1]} with integer coordinates on a line, find the minimum number m of points such that each segment contains at least one point.

covering_segments(*segments: [<class 'tuple'>]*) → list
Function for finding minimum number of points that each segment contains

Parameters **segments** – list of tuples with start and end point coordinates

Returns list of points where segments are crossing

Examples

```
>>> covering_segments([(4, 7), (1, 3), (2, 5), (5, 6)])
[3, 6]
```

3.2 Fractional Knapsack

Given weights and values of n items, we need put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

Complexity: O(n log n)

fractional_knapsack(*capacity: int, items: [<class 'tuple'>]*) → float
Function solves fractional knapsack problem

Parameters

- **capacity** – total capacity of backpack
- **items** – list of tuples [(value, weight),...]

Returns float - maximum value that can be placed in backpack

Examples

```
>>> fractional_knapsack(50, [(60, 10), (100, 20), (120, 30)])
240.0
```

CHAPTER 4

algorithms.search

4.1 Binary Search

In computer science, binary search is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array; if they are unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until it is successful. If the search ends with the remaining half being empty, the target is not in the array.

binary_search (*sorted_array*, *target_element*)

Binary search algorithm

Parameters

- **sorted_array** – list of sorted elements (integers)
- **target_element** – element to find

Returns position of target_element if succes -1 if element is not found

Examples

```
>>> binary_search([1, 2, 3, 4], 5)
-1
```

```
>>> binary_search([x for x in range(10000)], 26)
26
```

4.2 Closest Pair

The closest pair of points problem or closest pair problem is a problem of computational geometry: given n points in metric space, find a pair of points with the smallest distance between them. The closest pair problem for points in the

Euclidean plane[1] was among the first geometric problems that were treated at the origins of the systematic study of the computational complexity of geometric algorithms. [Wikipedia]

brute_force_distance (*points*)

Brute force solution for closest pair problem

Complexity: O(n^2)

Parameters **points** – list of points in the following format [(x1, y1), (x2, y2), … , (xn, yn)]

Returns Minimum distance between points

Examples

```
>>> n_log_n_squared_distance([(2, 3), (12, 30), (40, 50), (5, 1), (12, 10), (3,␣
↪4)])
1.4142135623730951
```

n_log_n_squared_distance (*points*)

O(N (LogN)^2) solution for closest pair problem

Complexity: O(n (log n)^2)

Parameters **points** – list of points in the following format [(x1, y1), (x2, y2), … , (xn, yn)]

Returns Minimum distance between points

Examples

```
>>> n_log_n_squared_distance([(2, 3), (12, 30), (40, 50), (5, 1), (12, 10), (3,␣
↪4)])
1.4142135623730951
```

4.3 Fibonacci

In mathematics, the Fibonacci numbers are the numbers in the following integer sequence, called the Fibonacci sequence, and characterized by the fact that every number after the first two is the sum of the two preceding ones:

fibonacci (*number*)

Recursive implementation of fibonacci function

Parameters **number** – number in fibonacci sequence

Returns fibonacci number

Examples

```
>>> fibonacci_recursive(20)
6765
```

4.4 Fibonacci with modulo

Calculating (n-th Fibonacci number) mod m

fibonacci_modulo(*number*, *modulo*)
Calculating (n-th Fibonacci number) mod m

Parameters

- **number** – fibonacci number
- **modulo** – modulo

Returns (n-th Fibonacci number) mod m

Examples

```
>>> fibonacci_modulo(11527523930876953, 26673)
10552
```

4.5 Rabin-Karp

In computer science, the Rabin–Karp algorithm or Karp–Rabin algorithm is a string searching algorithm created by Richard M. Karp and Michael O. Rabin (1987) that uses hashing to find any one of a set of pattern strings in a text. For text of length n and p patterns of combined length m,

its average and best case running time is O(n+m) in space O(p), but its worst-case time is O(nm).

rabin_karp(*text*, *pattern*)
Rabin-Karp algorithm that finds all occurrences of pattern in text

Parameters

- **text** – text to search in
- **pattern** – pattern to search for

Returns list of position where pattern placed in text

Examples

```
>>> rabin_karp('AABAACAADAABAABA', 'AABA')
[0, 9, 12]
```

```
>>> rabin_karp('aaaaa', 'aaa')
[0, 1, 2]
```

CHAPTER 5

algorithms.sorting

5.1 Merge Sort

In computer science, merge sort (also commonly spelled mergesort) is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. Mergesort is a divide and conquer algorithm that was invented by John von Neumann in 1945.

merge_sort (*array*)

Sort array via merge sort algorithm

Parameters **array** – list of elements to be sorted

Returns Sorted list of elements

Examples

```
>>> merge_sort([1, -10, 21, 3, 5])
[-10, 1, 3, 5, 21]
```

5.2 Quick Sort

Quicksort (sometimes called partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. Developed by Tony Hoare in 1959[1] and published in 1961,[2] it is still a commonly used algorithm for sorting. [Wikipedia]

https://en.wikipedia.org/wiki/Quicksort

Average complexity: O(n log n) Worst case: O(n^2)

quick_sort (*array*, *low*, *high*)

Quick Sort function sorts array

Parameters

- **array** – list of elements
- **low** – at starting point low must be = 0
- **high** – at starting point high must be = len(array)-1

Returns sorted array

Return type list

Examples

```
>>> quick_sort([4,  9,  4,  4,  1,  9,  4,  4,  9,  4,  4,  1,  4], 0, 12)
[1, 1, 4, 4, 4, 4, 4, 4, 4, 4, 9, 9, 9]
```

```
>>> quick_sort([1, 10, 32, 4.], 0, 3)
[1, 4.0, 10, 32]
```

switch (*array*, *a*, *b*)

CHAPTER 6

algorithms.dynamic_programming

6.1 Knapsack

The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit.

optimal_weight(*capacity*, *weights*)

Function calculate optimal_weight for rucksack from given list of weights

Parameters

- **capacity** – max capacity of rucksak
- **weights** – list of weights

Returns Max possible weight that meet <= max capacity

Examples

```
>>> optimal_weight(165, [23, 31, 29, 44, 53, 38, 63, 85, 89, 82])
165
```

CHAPTER 7

algorithms.hash_tables

7.1 Hash Chain

A hash chain is the successive application of a cryptographic hash function to a piece of data. In computer security, a hash chain is a method to produce many one-time keys from a single key or password. [Wikipedia]

class HashChain(*bucket_count*)

Bases: `object`

Class HashChain realisation

Examples

```
>>> hash_chain = HashChain(5)
>>> hash_chain.add("world")
>>> hash_chain.add("HellO")
>>> hash_chain.check(4)
HellO world
>>> hash_chain.find("World")
no
>>> hash_chain.find("world")
yes
>>> hash_chain.delete("world")
>>> hash_chain.check(4)
HellO
>>> hash_chain.delete("HellO")
>>> hash_chain.add("luck")
>>> hash_chain.add("GooD")
>>> hash_chain.check(2)
GooD luck
```

Explanation: The ASCII code of ’w’ is 119, for ’o’ it is 111, for ’r’ it is 114, for ’l’ it is 108, and for ’d’ it is 100. Thus, h(“world”) = 4. It turns out that the hash value of “HellO“ is also 4. We always insert in the beginning of the chain, so after adding "world" and then "HellO" in the same chain index 4, first goes "HellO" and then goes "world". Of course, "World" is not found, and "world" is found, because the strings are case-sensitive, and the codes of 'W' and 'w' are different. After deleting "world", only "HellO" is found in the chain 4. Similarly to "world" and "HellO", after adding "luck" and "GooD" to the same chain 2, first goes "GooD" and then "luck".

add(*data*)
Add data to hash table

Parameters **data** – string data

check(*idx*)
Check hash_chain by index

Parameters **idx** – index in hash

Returns String chain separated by space

delete(*data*)
Delete data from hash table

Parameters **data** – string data

find(*data*)
Find data in hash table

Parameters **data** – string data

Returns True if found, False if not

Python Module Index

a

Index

N

O

P

Q

R

S

T

V