Python Algorithms Documentation

Release 0.2.0

Md. Imrul Hassan

November 30, 2015

Contents

Contents:

CHAPTER 1

Python Algorithms

Python Algorithms contains a collection of useful algorithms written in python. The algorithms include (but not limited to) topics such as searching, sorting, graph, and string theory.

This project is inspired from the textbook Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne and associated book-site http://algs4.cs.princeton.edu/home/. The goal of this book is summarized in the following excerpt from the book-site:

> Our original goal for this book was to cover the 50 algorithms that every programmer should know. We use the word programmer to refer to anyone engaged in trying to accomplish something with the help of a computer, including scientists, engineers, and applications developers, not to mention college students in science, engineering, and computer science.

However, the algorithms for this project are not meant to be a ported version of the algorithms found in the book. Efforts should be made to implement those algorithms from the scratch following Pythonic coding style. Some of the algorithms are well known and the reference for those algorithms should appear in the documentation. While, some of the algorithms are very specific and difficult implement in a different way while maintaining accuracy and efficiency. Such algorithms appear in the scientific literatures and/or books and those should be properly referenced as well.

1.1 Features

- Mainly for educational purposes, but can be useful in certain practical scenarios as well.
- Consequently, built-in algorithms are avoided as much as possible and detailed implementation is done from the scratch.
- Preference is given towards a pythonic style rather than sticking to true OOP style.
- Free software: BSD license.
- Documentation: http://python_algorithms.rtfd.org.

1.2 Algorithms

Here is a list of algorithms divided into packages.

Note: Not all of the algorithms have been fully implemented yet.

1.2.1 Basic

A collection of few basic algorithms that do not fit in other packages. Trivial algorithms and data structures that are built into python are skipped.

- Binary search
- Knuth shuffle
- Stack
- Queue
- Bag
- Union find

  Estimated Release 0.2.0

1.2.2 Searching

Specialized searching algorithms and/or corresponding data structures are included in this package. Although some of the following data structures, such as Hash, are implemented in python, they have been implemented for demonstration purpose. Unless, specific needs arise, the built-in data structures should be preferred in production code.

- BST
- Red black BST
- Hash

  Estimated Release 0.3.0

1.2.3 Sorting

- Insertion
- Selection
- Merge
- Quick
- Quick 3 way
- Shell
- Heap

  Estimated Release 0.4.0

1.2.4 Graph

- Graph
- Directed graph
- BFS
- BFS paths
- DFS

- DFS paths
- Topological

  Estimated Release 0.5.0

1.2.5 String

- LSD
- MSD
- Quick 3 string
- TST
- KMP
- Rabin karp

  Estimated Release 0.6.0

CHAPTER 2

Installation

At the command line:

```
$ easy_install python_algorithms
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv python_algorithms
$ pip install python_algorithms
```

CHAPTER 3

Usage

To use Python Algorithms in a project:

```
import python_algorithms
```

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/mihassan/python_algorithms/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

4.1.4 Write Documentation

Python Algorithms could always use more documentation, whether as part of the official Python Algorithms docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/mihassan/python_algorithms/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *python_algorithms* for local development.

1. Fork the *python_algorithms* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/python_algorithms.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

   ```
   $ mkvirtualenv python_algorithms
   $ cd python_algorithms/
   $ python setup.py develop
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

   Alternatively, use git flow to start a new feature or bugfix branch and work on that branch locally. Once, ready to publish, push the branch to github:

   ```
   $ git flow feature start name-of-feature
   $ echo "Develop the feature on this feature branch"
   $ git flow feature finish
   $ git push
   ```

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

   ```
   $ flake8 python_algorithms tests
   $ python setup.py test
   $ tox
   ```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, and 3.3, and for PyPy. Check https://travis-ci.org/mihassan/python_algorithms/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_python_algorithms
```

CHAPTER 5

Credits

5.1 Development Lead

- Md. Imrul Hassan <mihassan@gmail.com>

5.2 Contributors

None yet. Why not be the first?

CHAPTER 6

History

6.1 0.1.0 (2014-02-14)

- First release on PyPI.

6.2 0.2.0 (2014-04-15)

- **Implemented the following data structures and algorithms:**
  - Binary search
  - Knuth shuffle
  - Stack
  - Queue
  - Bag
  - Union find

CHAPTER 7

API Reference

7.1 python_algorithms package

7.1.1 Subpackages

python_algorithms.basic package

Submodules

python_algorithms.basic.bag module

This module implements a bag or multiset data structure.

A bag or multiset is a generalization of the set data structure which allows repeated or duplicate items to be stored. Items can only be added to the bag and may not be removed. When the items in the bag are iterated there is not restriction on the ordering of the items.

In this module, the implementation of bag is similar to a linked list based stack implementation. In the linked list based implementation, the bag object need to keep track of only the head node. Each node contains an item and a link to the next node.

Note: Python' has a built-in class collections.Counter which is similar to a bag or multiset. instead of adding an item, 1 need to be added with the counter associated with that item and elements return all items (including duplicates) in the bag.

Complexity:

- add – O(1)

class python_algorithms.basic.bag.**Bag**

Bases: object

An implementation of a bag or multiset with linked list.

add(*item*)

Inserts an item to the bag.

isEmpty()

Check if the bag is empty.

Returns: True if the bag is empty. False otherwise.

size
The number of items in the bag.

python_algorithms.basic.binary_search module

This module implements a binary search method.

python_algorithms.basic.binary_search.**search**(*seq*, *val*)
Search location of key in a sorted list.

The method searches the location of a value in a list using binary searching algorithm. If it does not find the value, it return -1.

Args: seq: A sorted list where to search the value. val: A value to search for.

Returns: The location of the value in the sorted list if found. Otherwise returns -1.

python_algorithms.basic.knuth_shuffle module

This module implements a shuffle method using Knuth's algorithm.

python_algorithms.basic.knuth_shuffle.**shuffle**(*seq*)
Shuffle a list randomly using Knuth's algorithm.

The method randomly shuffles a list by iterating over each position and exchanging the element with another random element. The original list is not maintained and will change.

Args: seq: A list to shuffle.

Returns: The original list with all elements shuffled.

python_algorithms.basic.queue module

This module implements a linked list based queue data structure.

A queue is a data structure to hold a collection of items in order ad which supports operations such as the addition of an item (*enqueue*) and removal of an item (*dequeue*) can be performed. The items are always enqueued at the rear end of the queue and dequeued from the front end of the queue. As such, the queue can be also viewed as a First-In-First-Out(*FIFO*) data structure. In a FIFO data structure, the first item added to the structure is the first one to be removed. Apart from those two operations, *peek* operation can also be implemented, returning the value at the front end without removing it.

The particular implementation of queue in this module is based on linked list, as array based queue implementation is already supported in python's list. In the linked list based implementation, the queue object need to keep track of the front and the rear nodes where each node contains an item and a link to the next node.

Note: For most practical purposes, the python's implementation as dequeue suffices as a queue object. Use append method instead of enqueue and popleft method instead of dequeue for queue operations in a list.

Complexity:

- push – O(1)
- pop – O(1)
- peek – O(1)

class python_algorithms.basic.queue.**Queue**
Bases: object

An implementation of a simple queue with linked list.

dequeue()
Remove and return the first item from the queue.

Returns: The first item from the queue.

Raises: IndexError: If the queue is empty.

enqueue(*item*)
Insert an item to the queue.

isEmpty()
Check if the queue is empty.

Returns: True if the queue is empty. False otherwise.

peek()
Return the first item from the queue.

Returns: The first item from the queue.

Raises: IndexError: If the queue is empty.

size
The number of items in the queue.

python_algorithms.basic.stack module

This module implements a linked list based stack data structure.

A stack is a data structure to hold a collection of items in which operations such as the addition of an item (*push*) and removal of an item (*pop*) can be performed. The items are always pushed or popped from the so called *top* of the data structure which is the last item added or first item to be removed. The stack can be also viewed as a Last-In-First-Out(*LIFO*) data structure. In a LIFO data structure, the last item added to the structure must be the first item one to be removed. Apart from those two operations, *peek* operation can also be implemented, returning the value of the top item without removing it.

The particular implementation of stack in this module is based on linked list, as array based stack implementation is already supported in python's list. In the linked list based implementation, the stack object need to keep track of only the head node. Each node contains an item and a link to the next node.

Note: For most practical purposes, the python's list suffices as a stack object. Use append method instead of push and pop method as it is for stack operations in a list.

Complexity:

- push – O(1)
- pop – O(1)
- peek – O(1)

class python_algorithms.basic.stack.**Stack**
Bases: object

An implementation of a simple stack with linked list.

isEmpty()
Check if the stack is empty.

Returns: True if the stack is empty. False otherwise.

peek()
Return the last added item from the stack.

Returns: The last item added to the stack.

Raises: IndexError: If the stack is empty.

pop()
Remove and return the last added item from the stack.

Returns: The last item added to the stack.

Raises: IndexError: If the stack is empty.

push(*item*)
Insert an item to the stack.

size
The number of items in the stack.

python_algorithms.basic.union_find module

This module implements an union find or disjoint set data structure.

An union find data structure can keep track of a set of elements into a number of disjoint (nonoverlapping) subsets. That is why it is also known as the disjoint set data structure. Mainly two useful operations on such a data structure can be performed. A *find* operation determines which subset a particular element is in. This can be used for determining if two elements are in the same subset. An *union* Join two subsets into a single subset.

The complexity of these two operations depend on the particular implementation. It is possible to achieve constant time (O(1)) for any one of those operations while the operation is penalized. A balance between the complexities of these two operations is desirable and achievable following two enhancements:

1. Using union by rank – always attach the smaller tree to the root of the larger tree.
2. Using path compression – flattening the structure of the tree whenever find is used on it.

complexity:

- find – $O(\alpha(N))$ where $\alpha(n)$ is inverse ackerman function.
- union – $O(\alpha(N))$ where $\alpha(n)$ is inverse ackerman function.

class python_algorithms.basic.union_find.**UF**(N)
An implementation of union find data structure. It uses weighted quick union by rank with path compression.

connected(p, q)
Check if the items p and q are on the same set or not.

count()
Return the number of items.

find(p)
Find the set identifier for the item p.

union(p, q)
Combine sets containing p and q into a single set.

Module contents

This module contains few basic algorithms that do not fit in other packages. Trivial algorithms and data structures that are built into python are skipped.

7.1.2 Module contents

The module contains a collection of useful algorithms written in python.

CHAPTER 8

Indices and tables

- *genindex*
- *modindex*
- *search*

p