Algorithms in Python

Release 0.1

O.R.Senthil Kumaran

September 25, 2009

CONTENTS

CHAPTER

ONE

WHAT MAKES PYTHON SUITABLE?

1.1 Study of Algorithms is one of fundamental requirements in CS Education.

1.2 Knowing about Algorithms is going to help Programmers.

1.3 Unfortunately, in using traditional languages like C,C++ a lot of time is spent in understanding the syntax, semantics and program design that the idea of algorithm is overshadowed.

1.4 Well, it can be argued that once a person becomes familiar with language syntax then algorithms using that language is easy. *True*.

1.5 But algorithm learning is language agnostic, that is why _they_ do it in pseudo code.

1.6 But you see, Python is so close to pseudo-code, so close to english and so so natural for anything... well, most natural for learning algorithms.

1.7 Because it is easy to learn, easy to use and very easy to test your implementation and enhance your understanding.

CHAPTER

TWO

ALGORITHM ANALYSIS

2.1 Asymptote

A line which approaches nearer to some curve than assignable distance, but, though infinitely extended, would never meet it. Asymptotes may be straight lines or curves. A rectilinear asymptote may be conceived as a tangent to the curve at an infinite distance. [1913 Webster]

CHAPTER

THREE

NUMBERS, PRIME NUMBERS.

- Does that sound *Bond, James Bond.* Prime numbers are indeed James Bond of Numbers.
- Let us start with a simple algorithm and look at the python code.

3.1 Prime Numbers have exactly two factors 1 and itself.

3.2 Eratosthenes invented a sieve that would drain out composite numbers and give the primes.

```python
def eratosthenes():
    '''Yields the sequence of prime numbers via the Sieve of Eratosthenes.'''
    D = {}
    q = 2
    while True:
        p = D.pop(q, None)
        if p:
            x = p + q
            while x in D:
                x += p
            D[x] = p
        else:
            D[q*q] = q
            yield q
        q += 1
```

Credits: David Eppstein, Alex Martelli

3.3 This is an interesting algorithm. It is maintaining a dictionary of the nearest composite numbers and the smallest prime.

3.4 You may want to print out the variables at diffent point to see it (If you dont get it in the first shot)

CHAPTER

FOUR

TO SORT OR NOT TO SORT.

4.1 Computers spend more amount of time sorting than anything else.

4.2 It makes sense to know the sorting algorithm when doing algorithm intensive work or participating in any coding contest.

4.3 It fun! Almost magical, that same thing done in *slightly* different way produce so many different results.

CHAPTER

FIVE

BOGOSORT

5.1 In-effective sorting algorithm

5.2 Used for Educational purposes

while not InOrder(deck) do Shuffle(deck);

```
from random import shuffle

# Bogo-sort deck in place
while not all(x <= y for x, y in zip(deck, deck[1:])):
    shuffle(deck)
```

5.3 Complexity Analysis

Scenario	*Complexity*
Worst case	O(∞)
Best case	O(n)
Average case	O(n.n!)
Worst case space	O(n)

CHAPTER

SIX

INSERTION SORT

6.1 Start with the second element in the list and insert it at the correct position before it.

```
def insertionsort(A):
    """ Insertion sort in python.
    Start with the second element as the key and compare it with the elements
    preceding it. If you find the elements greater than key, shift the list one
    by one and when you find the element is lesser than key, insert the key at
    that position.
    """
    for j in range(1, len(A)):
        key = A[j]
        i = j -1
        while (i >= 0) and (A[i] > key):
            A[i+1] = A[i]
            i = i -1
        A[i+1] = key
```

6.2 Algorithm Analysis

Scenario	*Complexity*
Worst Case	O(n^2)
Best Case	O(n)
Average Case	O(n^2)
Worst Case Space	O(n) total, O(1) auxillary

CHAPTER

SEVEN

SELECTION SORT

7.1 Select the smallest from the rest and swap

```
def selectionsort(A):
    for i in range(0, len(A)-1):
        min = i
        for j in range(i+1, len(A)):
            if A[j] < A[min]:
                min = j
        if not (i == min):
            A[i], A[min] = A[min], A[i]
```

7.2 Algorithm Analysis

Scenario	*Complexity*
Worst Case	O(n^2)
Best Case	O(nlogn)
Average Case	O(n logn)
Worst Case Space	Varies by implementation

CHAPTER

EIGHT

SHELL SORT

8.1 Is like an insertion sort on an almost sorted list traversing a longer distance.

```python
def shellsort(A):
    inc = int(round(len(A)/2))
    while inc:
        for i in range(inc, len(A)):
            temp = A[i]
            j = i
            while ((j >= inc)  and (A[j-inc] > temp)):
                A[j] = A[j-inc]
                j = j - inc
            A[j] = temp
        inc = int(round(inc/2.2))
```

8.2 Algorithm Analysis

Scenario	*Complexity*
Worst Case	depends on gap seq
Best Case	O(n)
Average Case	depends on gap seq
Worst Case Space	O(n)

CHAPTER

NINE

QUICK SORT

```python
import random

def quicksort(A):
    lesser = []
    greater = []

    if len(A) <= 1:
        return A

    index = random.randint(0,len(A)-1)
    pivot = A.pop(index)

    for x in A:
        if x < pivot:
            lesser.append(x)
        elif x >= pivot:
            greater.append(x)

    lesser = quicksort(lesser)
    greater = quicksort(greater)

    return lesser + [pivot] + greater
```

9.1 Algorithm Analysis

Scenario	*Complexity*
Worst Case	O(n^2)
Best Case	O(n^2)
Average Case	O(n^2)
Worst Case Space	O(n) total, O(1) auxillary

CHAPTER

TEN

MERGE SORT

```python
import math

def merge(left, right):
    result = list()
    while (len(left) > 0) and (len(right) > 0):
        if left[0] <= right[0]:
            result.append(left[0])
            left = left[1:]
        else:
            result.append(right[0])
            right = right[1:]
    if left:
        result.extend(left)
    else:
        result.extend(right)
    return result

def mergesort(m):
    left = list()
    right = list()
    result = list()

    if len(m) <= 1:
        return m
    middle = int(math.ceil(len(m)/2.0))
    for x in range(0,middle):
        left.append(m[x])
    for x in range(middle,len(m)):
        right.append(m[x])

    left = mergesort(left)
    right = mergesort(right)

    left_last_item = left[len(left)-1]
    right_first_item = right[0]

    if left_last_item > right_first_item:
        result = merge(left, right)
    else:
        left.extend(right)
        result = left
    return result
```

10.1 Algorithm Analysis

Scenario	*Complexity*
Worst Case	O(nlogn)
Best Case	O(nlogn)/ O(n)
Average Case	O(n logn)
Worst Case Space	O(n)

CHAPTER

ELEVEN

BUBBLE SORT

“the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems”. Donald Knuth.

```python
def bubblesort(A):
    swapped = True
    while swapped:
        swapped = False
        for i in range(len(A)-1):
            if A[i] > A[i+1]:
                A[i], A[i+1] = A[i+1], A[i]
                swapped = True
```

11.1 Algorithm Analysis

Scenario	*Complexity*
Worst Case	O(n^2)
Best Case	O(n)
Average Case	O(n^2)
Worst Case Space	O(n) total, O(1) auxillary

CHAPTER

TWELVE

BISECT MODULE

12.1 Inserting elements into a list in a sorted order.

12.2 This can be much more efficient than repeatedly sorting a list

12.3 Or Explicitly sorting a large list after it is constructed.

```
"""
The bisect module implements an algorithm for inserting elements into a list in
sorted order. This can be much more efficient than repeatedly sorting a list or
explicitly sorting a large list after it is constructed.
"""

import bisect

def bisectionsort(A):
    resultant = []
    for elem in A:
        pos = bisect.bisect(resultant, elem)
        bisect.bisect(resultant, elem)
    return resultant
```

CHAPTER

THIRTEEN

RUN TIME ANALYSIS

13.1 Let us look at the Running time of various algorithms

13.2 Toy around with the files py31/runtimeanalysis.py py26/runtimeanalysis.py

CHAPTER

FOURTEEN

PYTHON’S INTERNAL SORT

14.1 timsort

14.2 Exploits the patterns in data, Wickedly fast on partially or already sorted list.

14.3 non-recursive adaptive stable natural mergesort / binary insertion sort hybrid

14.4 In a nutshell, the main routine marches over the array once, left to right, alternately identifying the next run, then merging it into the previous runs “intelligently”.

14.5 Java’s sort modified to use timsort.

CHAPTER

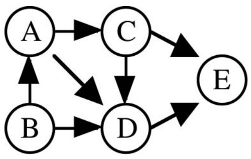
FIFTEEN

REPRESENTING GRAPHS

15.1 Graphs are basically dictionaries with keys as nodes and values as the connected nodes.

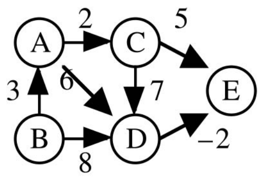
15.2 Directed Graph can also be reprented as class. With common graph operations as methods

15.3 Directed Graph



```
H = {'A': ['C','D'],
     'B': ['A','D'],
     'C': ['D','E'],
     'D': ['E'],
     'E': []
    }
```

15.4 Weighted Graph



```
W = {'A':{'C':2,'D':6},
     'B':{'A':3,'D':8},
     'C':{'E':5,'D':7},
     'D':{'E':-2},
     'E':{}
    }
```

15.5 Graph Class

```
class Graph:
    def __init__(self, g):
        self.g = g

    def V(self):
        return list(self.g.keys())

    def Adj(self, v):
        return list(self.g[v].keys())

    def W(self, v, u):
        return self.g[v][u]
```

CHAPTER

SIXTEEN

PATH FINDING ALGORITHMS

16.1 Good examples for Recursion

```
def find_path(graph, start, end, path=[]):
    path = path + [start]
    if start == end:
        return path
    if start not in graph:
        return None
    for node in graph[start]:
        if node not in path:
            newpath = find_path(graph, node, end, path)
            if newpath: return newpath
    return None

def find_shortest_path(graph, start, end, path=[]):
    path = path + [start]
    if start == end:
        return path
    if start not in graph:
        return None
    shortest = None
    for node in graph[start]:
        if node not in path:
            newpath = find_shortest_path(graph, node, end, path)
            if newpath:
                if not shortest or len(newpath) < len(shortest):
                    shortest = newpath
    return shortest
```

Source: Guido's essay.

CHAPTER

SEVENTEEN

RESOURCES

- Sieve of Eratosthenes at ActiveState
- Algorithm Education in Python - UC, Irvine
- Guido's Graph Essay
- Sorting in Python
- Sorting Algorithm - Wikipedia

CHAPTER

EIGHTEEN

ALGORITHMS RELATED RESOURCES

- itertools module in Python Standard Library
- Easy AI with Python by Raymond Hettinger
- David Eppstein's Python Algorithms and Data Structures

CHAPTER

NINETEEN

THANKS!

- O.R.Senthil Kumaran
- Presentation Slides and Source Files