ANALYSIS OF ALGORITHMS

CSCE 629 – FALL 2016

Aditya Emani
UIN: 524002563

Table of Contents

Matrix Chain Multiplication

Problem Statement

Given a sequence of n matrices A_1, A_2, A_3, ….., A_n with dimensions $p_0,p_1,p_2,\dots p_n$ where, A_i is of dimension p_{i-1} X p_i, determine the order of multiplication that minimizes the cost.

To demonstrate the problem statement, let us consider multiplication of three matrices A_1, A_2, A_3 with rank [2,3], [3,6], [6,2]. The product of A_1, A_2, A_3 can be written mathematically as $(A_1.A_2.\ A_3)$.

As matrix multiplication is associative,

$$(A_1.A_2.A_3) == (A_1.A_2).A_3 == A_1.(A_2.A_3)$$

Cost of two matrices A and B is given by product (**p.q.r**) where p is the number of rows in A, q is the number of columns in A and r is the number of columns in B

Finding cost of $(A_1.A_2).A_3$ & $A_1.(A_2.A_3)$

Cost of $(A_1.A_2).A_3$ = (2*3*6) + (2*6*2) = 36+24 = 60 ->1
Cost of $A_1.(A_2.A_3)$ = (2*3*2) + (3*6*2) = 12+36 = 48 ->2

Clearly from above, the cost of equation 2 is less than the cost of equation 1. Finding the order of matrix multiplication to yield low cost is the problem statement for matrix chain multiplication.

Matrix Chain Multiplication exhibits optimal substructure. So, we use dynamic programming to implement this problem statement. In dynamic programming, we build an optimal solution to the problem by building optimal solution to the sub problems.

Formulation

Let m(i,j) be the minimum number of computations needed to compute $A_{i..j.}$ The optimal cost can be determined by the following formulation below.

$$m(i,j) = 0 \; for \; i = j$$
$$m(i,j) = \min_{i \le k < j} (m(i,k) + \; m(k+1,j), p_{i-1}\,.p_{k}.p_j) \; for \; i < j$$

We use above formulation to solve the matrix chain multiplication problem using dynamic programming.

Implementation Code

Source: program.py

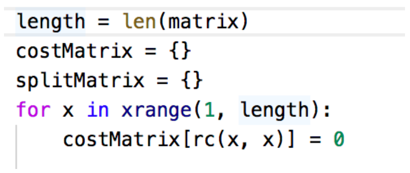
Code for this problem is implemented in python. It has the following three definitions.

Def: Matrix Chain Multiplication

Input: Array named matrix
Output: minimum cost for the matrix multiplication

All the implementation of dynamic programming to solve the problem statement is in this definition. This definition takes an array called matrix as input and implements the algorithm for dynamic programming using this matrix. It then stores the cost for each sub problem in another list called costMatrix. It also stores the order of multiplication in another matrix called splitMatrix. Its implementation is clearly explained below with corresponding code snippets.

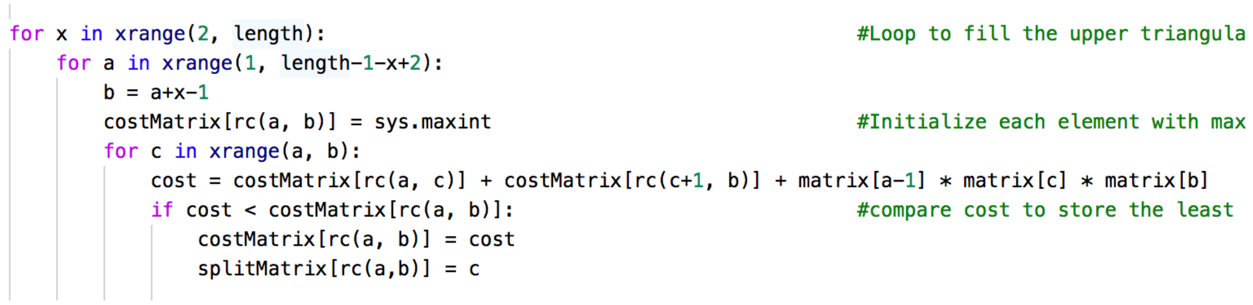
Initialization of Variables



```
length = len(matrix)
costMatrix = {}
splitMatrix = {}
for x in xrange(1, length):
    costMatrix[rc(x, x)] = 0
```

First, we initialize two lists called costMatrix (to store costs of each sub problem) and splitMatrix (to store the order of multiplication) and then initialize all the elements of costMatrix to 0. We save the length of matrix array in variable length.

Implementation



```
for x in xrange(2, length):                                                 #Loop to fill the upper triangula
    for a in xrange(1, length-1-x+2):
        b = a+x-1
        costMatrix[rc(a, b)] = sys.maxint                                   #Initialize each element with max
        for c in xrange(a, b):
            cost = costMatrix[rc(a, c)] + costMatrix[rc(c+1, b)] + matrix[a-1] * matrix[c] * matrix[b]
            if cost < costMatrix[rc(a, b)]:                                 #compare cost to store the least
                costMatrix[rc(a, b)] = cost
                splitMatrix[rc(a,b)] = c
```

Consider a matrix array to be [1,2,3,4]. At the begining of this code, the costMatrix will be as below.

$$costMatrix = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Iterations of code

When x = 2,

- a = 1, b = 2 and c = 1

$$costMatrix = \begin{bmatrix} 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

- a = 2, b = 3 and c = 2

$$costMatrix = \begin{bmatrix} 0 & 6 & 0 & 0 \\ 0 & 0 & 24 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

When, x = 3

- a = 1, b = 3 and c = 1

$$costMatrix = \begin{bmatrix} 0 & 6 & 32 & 0 \\ 0 & 0 & 16 & 24 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

- a = 1, b = 3 and c = 2

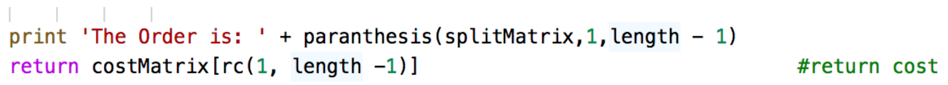
$$costMatrix = \begin{bmatrix} 0 & 6 & 18 & 0 \\ 0 & 0 & 16 & 24 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Note: In this iteration, the cost for [1,3] is calculated in other order and it is observed that the cost is less than that of previous calculation which is presented in red above. So, the value of [1,3] in the costMatrix is replaced with the least cost which is presented in green.

After the final iteration, the value in the first row and (length – 1) i.e., 3rd column is the least cost of matrix multiplication.

This code also keeps track of variable c which is used to check all the possible orders of multiplication in another list called splitMatrix.

Return from definition

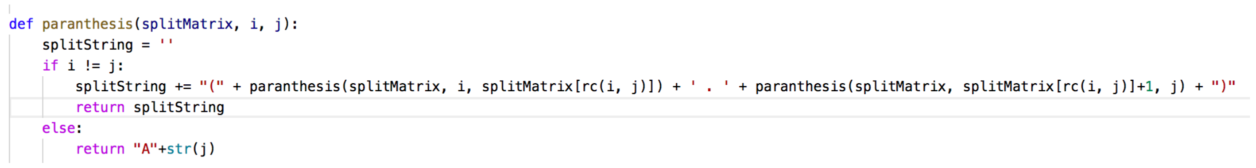


```
print 'The Order is: ' + paranthesis(splitMatrix,1,length - 1)
return costMatrix[rc(1, length -1)]                        #return cost
```

The last two lines of this definition are used to print the order of matrix multiplication to console and return the minimum cost for that order.

Def: Paranthesis

Input: Array named splitMatrix, first index, last index
Output: splitString



```
def paranthesis(splitMatrix, i, j):
    splitString = ''
    if i != j:
        splitString += "(" + paranthesis(splitMatrix, i, splitMatrix[rc(i, j)]) + ' . ' + paranthesis(splitMatrix, splitMatrix[rc(i, j)]+1, j) + ")"
        return splitString
    else:
        return "A"+str(j)
```

This method returns a string called splitString. The method iterates over itself between first index and last index. If the first index is equal to last index, the definition returns a string A by concatenating it with the last index. If the first index is not equal to last index, then, the function recursively calls until they are equal and the output is parenthesized. This definition prints the order of multiplying matrices with least cost.

Def: rc

Input: It takes two integers i and j as inputs
Output: returns a string in the format of (i,j)

This function takes two integers as inputs and returns a string in the format of (i,j). This is used to refer the rows and columns of costMatrix/splitMatrix in the program.

Testing Code

Source: test_program.py

Test code for this problem creates a number of non-trivial inputs and calls the definition matrix_chain_multiplication() with the input parameters. It then assigns the output to another variable called value and outputs the cost to the console. It then checks the correctness of the output using an assert statement. In this test file, 10 different scenarios are used to test the correctness of the problem code in program.py.

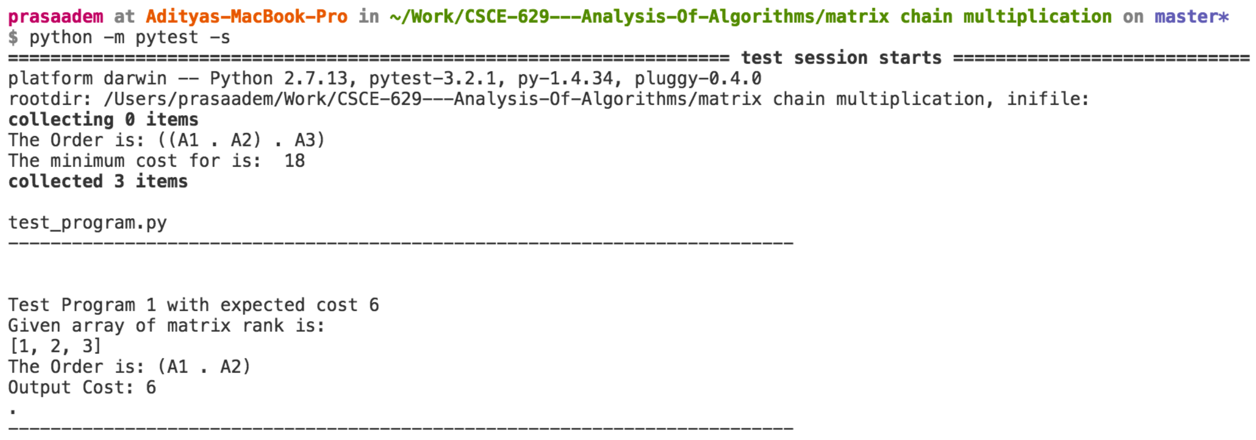
Results:

- All the 10 test scenarios passed successfully.

Example 1

Given input matrix array [1,2,3]

So, the matrices are of rank 1X2 and 2X3. As there are only two matrices, the total minimum cost is: 1*2*3 = 6. The resulted output from the test case is: 6

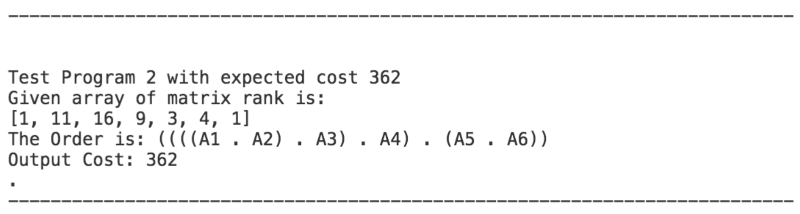


```
prasaadem at Adityas-MacBook-Pro in ~/Work/CSCE-629---Analysis-Of-Algorithms/matrix chain multiplication on master*
$ python -m pytest -s
==================================================================== test session starts ============================
platform darwin -- Python 2.7.13, pytest-3.2.1, py-1.4.34, pluggy-0.4.0
rootdir: /Users/prasaadem/Work/CSCE-629---Analysis-Of-Algorithms/matrix chain multiplication, inifile:
collecting 0 items
The Order is: ((A1 . A2) . A3)
The minimum cost for is:  18
collected 3 items

test_program.py
--------------------------------------------------------------------------


Test Program 1 with expected cost 6
Given array of matrix rank is:
[1, 2, 3]
The Order is: (A1 . A2)
Output Cost: 6
.
--------------------------------------------------------------------------
```

Example 2

Given input matrix array [1, 11, 16, 9, 3, 4, 1]

The expected minimum cost is: 362. The resulted output from the test case is: 362

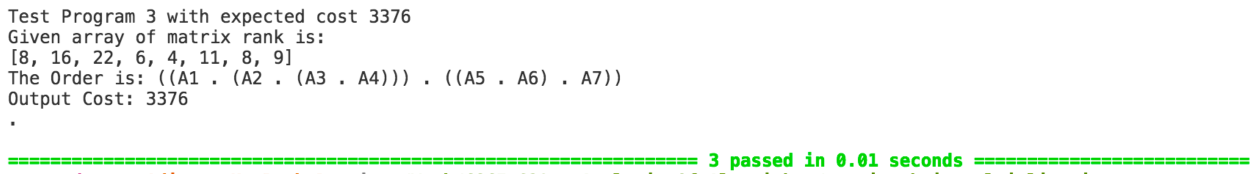


```
--------------------------------------------------------------------------


Test Program 2 with expected cost 362
Given array of matrix rank is:
[1, 11, 16, 9, 3, 4, 1]
The Order is: ((((A1 . A2) . A3) . A4) . (A5 . A6))
Output Cost: 362
.
--------------------------------------------------------------------------
```

Example 3

Given input matrix array [8, 16, 22, 6, 4, 11, 8, 9]

The expected minimum cost is: 3376. The resulted output from the test case is: 3376



```
Test Program 3 with expected cost 3376
Given array of matrix rank is:
[8, 16, 22, 6, 4, 11, 8, 9]
The Order is: ((A1 . (A2 . (A3 . A4))) . ((A5 . A6) . A7))
Output Cost: 3376
.

================================================================ 3 passed in 0.01 seconds ========================
```

Longest Common Subsequence

Problem Statement

In the longest common subsequence problem, we are given with two sequences $X = \langle x_1, x_2, x_3, \ldots\ldots x_m \rangle$ & $Y = \langle y_1, y_2, y_3, \ldots\ldots y_n \rangle$ and wish to find the longest common subsequence of X and Y.

To demonstrate the problem statement, let us consider two strings X = ABACDDEFGA and Y = BCDFAIJO. The longest common subsequence for these two BCDFA and the length of the string is 5.

This problem statement can be resolved by using dynamic programming. In dynamic programming, we build an optimal solution to the problem by building optimal solution to the sub problems.

To find the longest subsequences common to X_i and Y_j, compare the elements x_i and y_j. If they are equal, then the sequence $LCS(X_{i\text{-}1}, Y_{j\text{-}1})$ is extended by that element, x_i. If they are not equal, then the longer of the two sequences, $LCS(X_i, Y_{j\text{-}1})$, and $LCS(X_{i\text{-}1}, Y_j)$, is retained.

Formulation

$$LCS(X_i, Y_j) = \begin{cases} \emptyset \; if \;\; i = 0 \; or \; j = 0 \\ LCS(X_{i-1}, Y_{j-1}) \, x_i \; if \; x_i = y_j \\ \max\left(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)\right) if \; x_i \neq y_j \end{cases}$$

We use above formulation to solve the matrix chain multiplication problem using dynamic programming.

Implementation Code

Source: program.py

Code for this problem is implemented in python. It has the following one definition.

Def: Longest Common Subsequence

Input: Two strings namely string1 and string2
Output: length (length of subsequence) and string (subsequence string)

All the implementation of dynamic programming to solve the problem statement is in this definition. This definition takes two strings string1 and string2 as input and implements the algorithm for dynamic programming using these strings. It then stores the position for each sub problem in another array called position. It also stores the path followed in another array called path. Its implementation is clearly explained below with corresponding code snippets.

Initialization of Variables

```python
def longest_common_subsequence(string1, string2):
  rows = len(string1) + 1
  cols = len(string2) + 1
  position = [[0 for x in xrange(cols)] for y in xrange(rows)]
  path = [[(0, 0) for x in xrange(cols)] for y in xrange(rows)]
```

First, we initialize two arrays called position (to store position) and path (to store the path traversed) and then initialize all the elements of costMatrix to 0. We save the length of each string in rows and cols respectively.

Implementation

```
for i in xrange(1, rows):
  for j in xrange(1, cols):
    if string1[i - 1] == string2[j - 1]:
      position[i][j] = position[i - 1][j - 1] + 1
      path[i][j] = (i - 1, j - 1)
    else:
      if position[i - 1][j] >= position[i][j - 1]:
        position[i][j] = position[i - 1][j]
        path[i][j] = (i - 1, j)
      else:
        position[i][j] = position[i][j - 1]
        path[i][j] = (i, j - 1)
```

Consider two strings “ABABCZ” and “ABCAZZ”. At the begining of this code, both the position and path arrays will be as below.

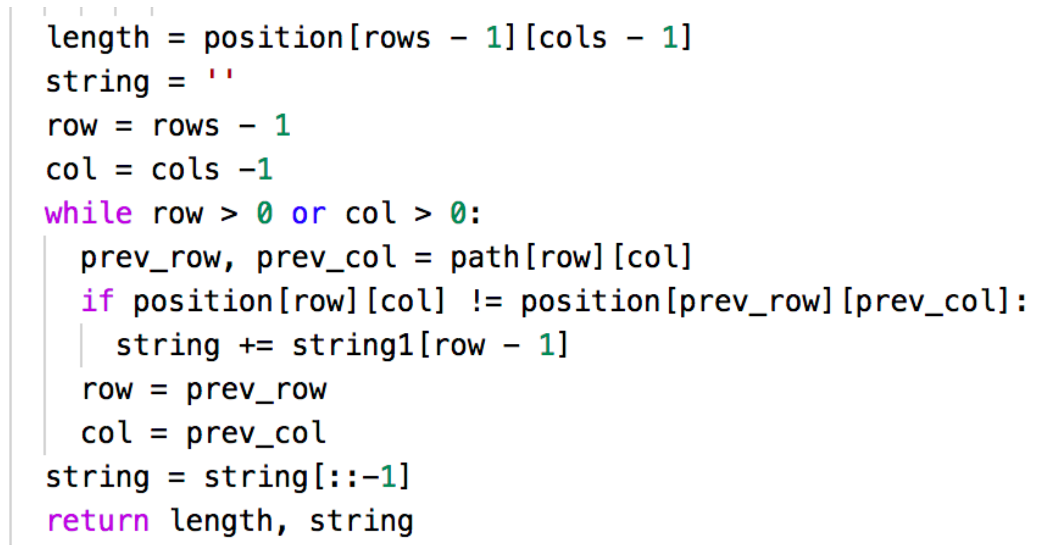
$$position = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Iterations of code

In this section, we iterate variable i from 1 to the length of (string1+1) and check if there is a match with this character to any character in string2. This can be done easily with the help of using another for loop with for string2. If there is a match, increment the value of previous diagonal element by 1 and store it in index (i,j) of position matrix. We then store the previous index in path(i,j). This helps in iteration of matching strings at a later stage to print the subsequence.

If there is no match, then we find the maximum among the two adjacent previous indices and update the values of position and path correspondingly.

Return from definition



```
length = position[rows - 1][cols - 1]
string = ''
row = rows - 1
col = cols -1
while row > 0 or col > 0:
  prev_row, prev_col = path[row][col]
  if position[row][col] != position[prev_row][prev_col]:
    string += string1[row - 1]
  row = prev_row
  col = prev_col
string = string[::-1]
return length, string
```

In this code, we get the length of the subsequence and iterate the path to get the position of the character. The string we get is in reverse as we iterate from the last string. So, we have to reverse the string and return it along with the length.

Testing Code

Source: test_program.py

Test code for this problem creates a number of non-trivial inputs and calls the definition longest_common_subsequence() with the input parameters. It then assigns the output to variables called length and string and outputs them to the console. It then checks the correctness of the output using an assert statement. In this test file, 10 different scenarios are used to test the correctness of the problem code in program.py.

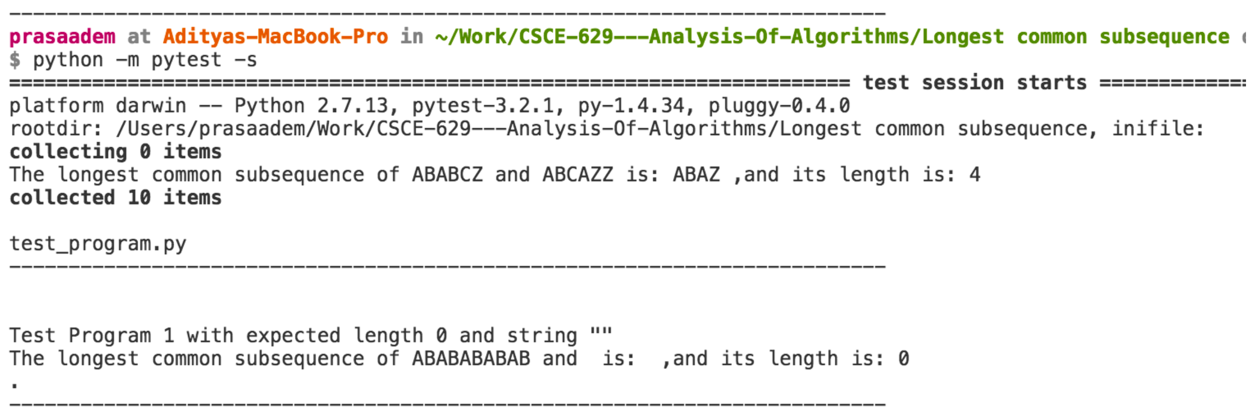
Results:

- All the 10 test scenarios passed successfully.

Example 1

Given input strings ABABABABAB and “”

The longest common subsequence will be “” and its length should be 0. The resulted output from the test case is: 0

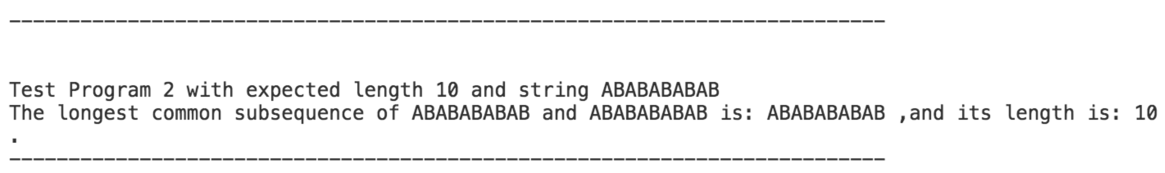


```
--------------------------------------------------------------------------
prasaadem at Adityas-MacBook-Pro in ~/Work/CSCE-629---Analysis-Of-Algorithms/Longest common subsequence 
$ python -m pytest -s
======================================================================= test session starts ============
platform darwin -- Python 2.7.13, pytest-3.2.1, py-1.4.34, pluggy-0.4.0
rootdir: /Users/prasaadem/Work/CSCE-629---Analysis-Of-Algorithms/Longest common subsequence, inifile:
collecting 0 items
The longest common subsequence of ABABCZ and ABCAZZ is: ABAZ ,and its length is: 4
collected 10 items

test_program.py
--------------------------------------------------------------------------


Test Program 1 with expected length 0 and string ""
The longest common subsequence of ABABABABAB and  is:  ,and its length is: 0
.
--------------------------------------------------------------------------
```

Example 2

Given input strings ABABABABAB and ABABABABAB

The longest common subsequence will be ABABABABAB and its length should be 10. The resulted output from the test case is: 10

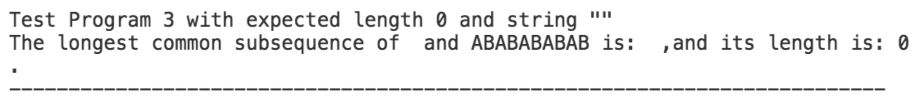


```
--------------------------------------------------------------------------


Test Program 2 with expected length 10 and string ABABABABAB
The longest common subsequence of ABABABABAB and ABABABABAB is: ABABABABAB ,and its length is: 10
.
--------------------------------------------------------------------------
```

Example 3

Given input strings “” and ABABABABAB

The longest common subsequence will be “” and its length should be 0. The resulted output from the test case is: 0



```
Test Program 3 with expected length 0 and string ""
The longest common subsequence of  and ABABABABAB is:  ,and its length is: 0
.
--------------------------------------------------------------------------
```

Single Source Shortest Path

Problem Statement

Find shortest paths from the start vertex to sink vertex nearer than or equal to the end.

Single Source Shortest Path algorithm determines the shortest accumulated total cost to all vertices emanating from a single source vertex. In addition to computing the cost, it records information that allows you to recover the actual shortest path between the source and any vertex in the graph.

To demonstrate the code in action, we use a graph representation such as the following:

```
graph = {'x': {'a': 2, 'b': 1},
            'a': {'x': 3, 'b': 4, 'c':8},
            'b': {'x': 8, 'a': 2, 'd': 2},
            'c': {'a': 2, 'd': 9, 'e': 4},
            'd': {'b': 1, 'c': 1, 'e': 3},
            'e': {'c': 3, 'd': 3}}
```

here, x is the source and e is the sink. We start with the source node x and find its adjacent nodes. Once a node is visited, we mark its distance from previous node and mark it as visited. Then we move to next node. If there is a shorter path from source to already visited node, we update its path and distance. We iterate till all the nodes are visited. Finally, we check for any non-visited nodes and return the path and distance of sink node to console.

Implementation Code

Source: program.py

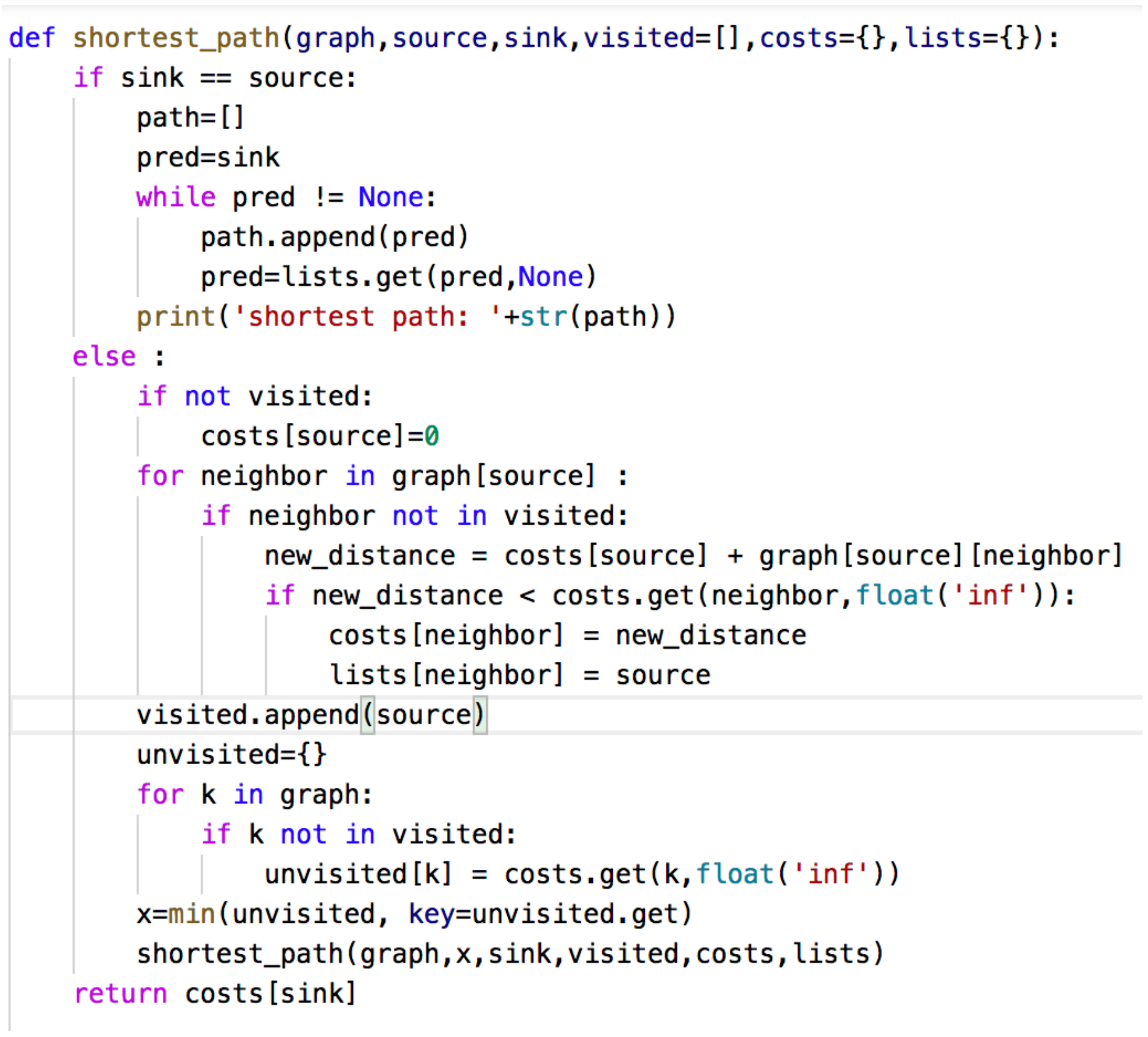
Code for this problem is implemented in python. It has the following one definition.

Def: Shortest Path

Input: graph containing all the links, a source, a sink, an array visited, a dictionary called costs and a list called lists
Output: cost for the sink using costs[sink]

All the implementation of dijksta's algorithm to solve the problem statement is in this definition. This definition takes graph, source and sink as input and implements the algorithm using these. It then starts with source x and find its adjacent nodes. Once a node is visited, we mark its distance from previous node and mark it as visited. Then we move to next node. If there is a shorter path from source to already visited node, we update its path and distance. We iterate till all the nodes are visited. Finally, we check for any non-visited nodes and return the path and distance of sink node to console.

Program Code



```python
def shortest_path(graph,source,sink,visited=[],costs={},lists={}):
    if sink == source:
        path=[]
        pred=sink
        while pred != None:
            path.append(pred)
            pred=lists.get(pred,None)
        print('shortest path: '+str(path))
    else :
        if not visited:
            costs[source]=0
        for neighbor in graph[source] :
            if neighbor not in visited:
                new_distance = costs[source] + graph[source][neighbor]
                if new_distance < costs.get(neighbor,float('inf')):
                    costs[neighbor] = new_distance
                    lists[neighbor] = source
        visited.append(source)
        unvisited={}
        for k in graph:
            if k not in visited:
                unvisited[k] = costs.get(k,float('inf'))
        x=min(unvisited, key=unvisited.get)
        shortest_path(graph,x,sink,visited,costs,lists)
    return costs[sink]
```

To understand the code clearly, first consider the else part of if-else condition. First if there are no elements in visited array, we initialize costs for source to 0. Then we look for neighbors of source in the graph and iterate through them one by one. If the neighbor is not visited, we calculate the distance of the neighbor from source. If the new distance is smaller than the existing cost, we update the new distance in the costs list for that neighbor and also in the lists list. Now we mark the source as visited by adding it to the visited array. Now for every element in graph, we check if this element is visited and then find the cost. A minimum value is stored for that element in x and the shortest path definition is called recursively with all the parameters. Finally, distance to the sink is returned.

To print the path to the sink, we use if condition to check if the sink is equal to the source. If the sink is equal to the source, all the iterations are successful. We then append that path with the predecessor and print the shortest path.

Testing Code

Source: test_program.py

Test code for this problem creates a number of non-trivial inputs and calls the definition shortest_path() with the input parameters. It then assigns the output to variable called cost and outputs to the console. It then checks the correctness of the output using an assert statement. In this test file, 8 different scenarios are used to test the correctness of the problem code in program.py.

Results:

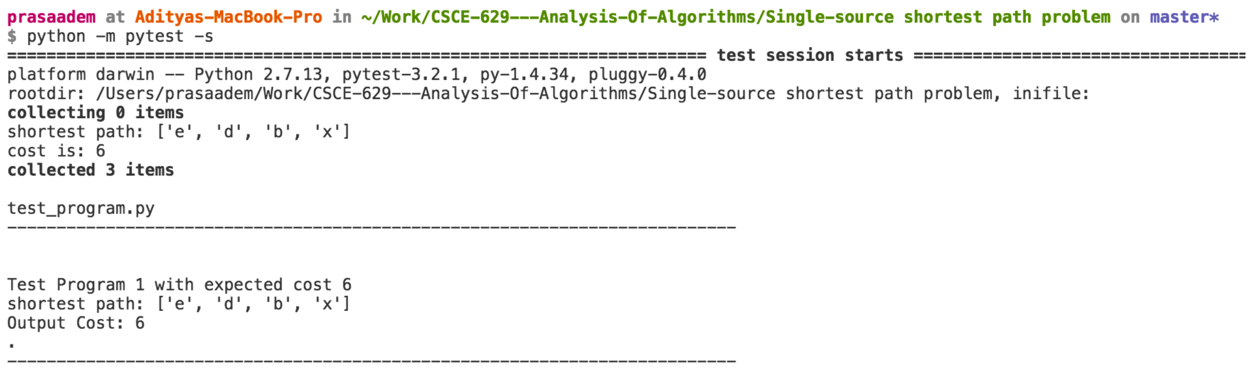
- All the 8 test scenarios passed successfully.

Example 1

Given input graph g = `{'x': {'a': 2, 'b': 1},`

```
'a': {'x': 3, 'b': 4, 'c':8},
'b': {'x': 8, 'a': 2, 'd': 2},
'c': {'a': 2, 'd': 9, 'e': 4},
'd': {'b': 1, 'c': 1, 'e': 3},
'e': {'c': 3, 'd': 3}}
```

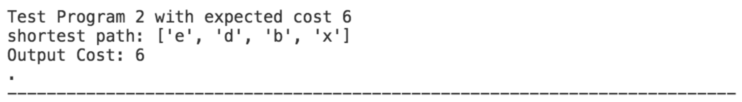
The shortest distance between source x and sink e is: 6



```
prasaadem at Adityas-MacBook-Pro in ~/Work/CSCE-629---Analysis-Of-Algorithms/Single-source shortest path problem on master*
$ python -m pytest -s
======================================================================= test session starts ==================================
platform darwin -- Python 2.7.13, pytest-3.2.1, py-1.4.34, pluggy-0.4.0
rootdir: /Users/prasaadem/Work/CSCE-629---Analysis-Of-Algorithms/Single-source shortest path problem, inifile:
collecting 0 items
shortest path: ['e', 'd', 'b', 'x']
cost is: 6
collected 3 items

test_program.py
--------------------------------------------------------------------------


Test Program 1 with expected cost 6
shortest path: ['e', 'd', 'b', 'x']
Output Cost: 6
.
--------------------------------------------------------------------------
```

Example 2

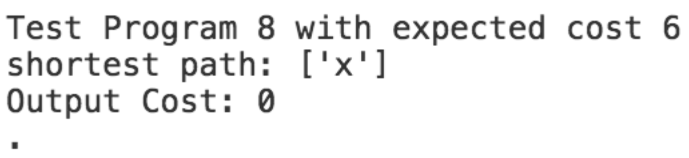
The shortest distance between source x and sink d is: 6



```
Test Program 2 with expected cost 6
shortest path: ['e', 'd', 'b', 'x']
Output Cost: 6
.
--------------------------------------------------------------------------
```

Example 3

The shortest distance between source x and sink x is: 0



```
Test Program 8 with expected cost 6
shortest path: ['x']
Output Cost: 0
.
```

Maximum Flow

Problem Statement

Given a graph with weighted edges, find the maximum flow from source to a sink.

To understand the problem, we need to know a few definitions before solving it.

Vertex: A point or a source
Edge: Path joining two vertices
Flow: Units passing on an edge
Capacity: Maximum units that can pass on that edge
Residual capacity: Total capacity – flow
Augmented Path: A path from source to sink such that every edge in that path has a residual capacity > 0

We use Ford Fulkerson algorithm to solve this problem. In this algorithm, we find the augmented paths from source to sink, find the maximum flow in that path (Usually, the edge with least flow is the maximum flow of that augmented path). We then subtract this flow from all the edges in the augmented path. We then iterate again to find the augmented path and do the same until we cannot find any more augmented paths. Then the maximum flow of the network is given by the sum of flows of each augmented path.

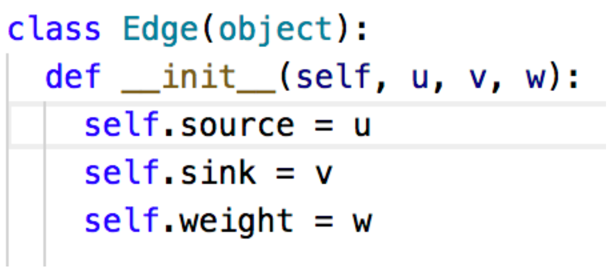
Implementation Code

Source: program.py

Code for this problem is implemented in python. It has the following definitions.

Initialization of Variables and Classes

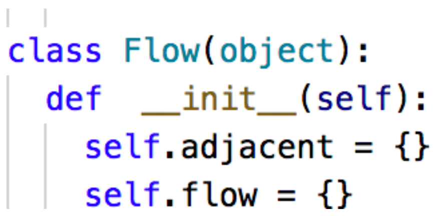
Class: Edge



```python
class Edge(object):
  def __init__(self, u, v, w):
    self.source = u
    self.sink = v
    self.weight = w
```

Edge is a class used to represent an edge in the flow network. It has an initializer method which takes four inputs u, v, w for initializing source, sink and weight of each edge.

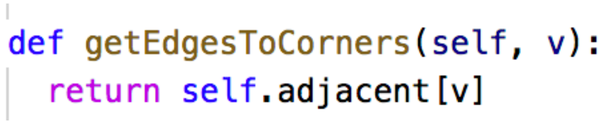
Class: Flow



```python
class Flow(object):
  def  __init__(self):
    self.adjacent = {}
    self.flow = {}
```

Flow is a class used to represent a adjacent edges to vertex in the flow network. It has an initializer method which sets null to flow and adjacent lists.

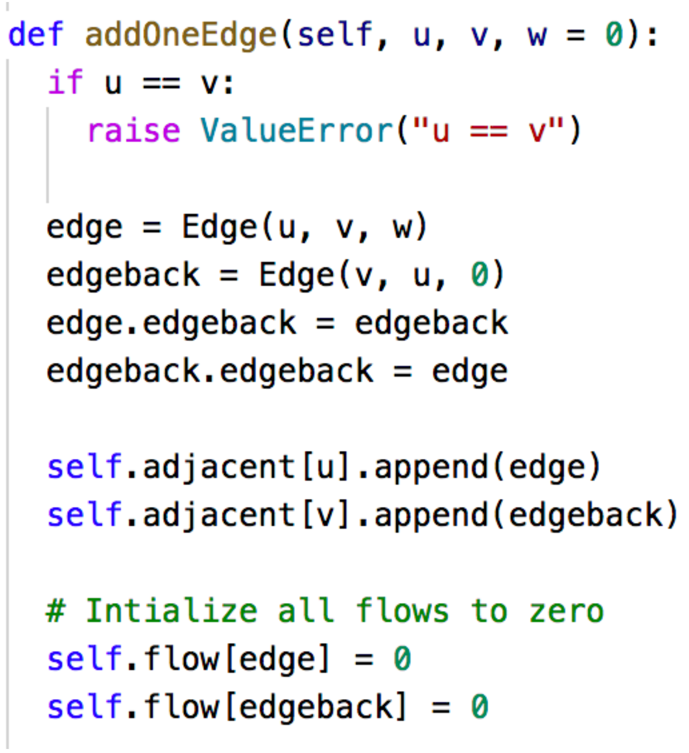
def: getEdgesToCorners



```
def getEdgesToCorners(self, v):
  return self.adjacent[v]
```

This method takes a vertex as an input and returns all the edges which are adjacent to that particular vertex.

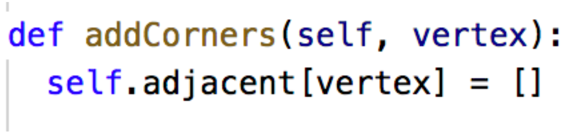
def: addOneEdge



```
def addOneEdge(self, u, v, w = 0):
  if u == v:
    raise ValueError("u == v")

  edge = Edge(u, v, w)
  edgeback = Edge(v, u, 0)
  edge.edgeback = edgeback
  edgeback.edgeback = edge

  self.adjacent[u].append(edge)
  self.adjacent[v].append(edgeback)

  # Intialize all flows to zero
  self.flow[edge] = 0
  self.flow[edgeback] = 0
```

This method takes two vertices and a weight as an input. It then checks if both the vertices are same, if same, raises an error. Then It creates an edge with the input parameters by calling the Edge class initializer. Also it adds the edge to both the vertices in opposite direction.

def: addCorner



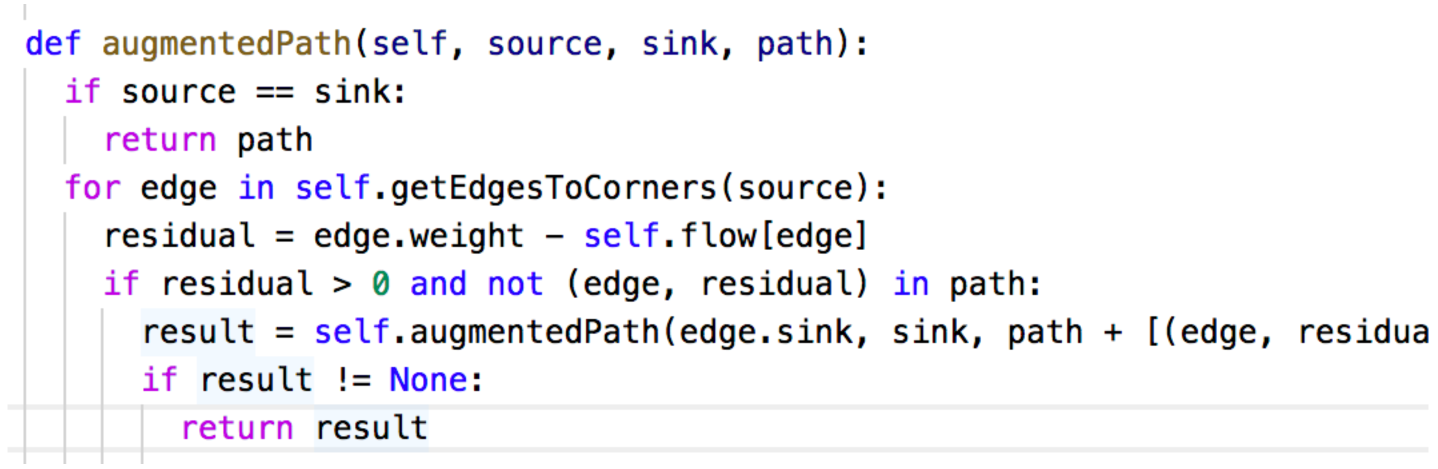
```
def addCorners(self, vertex):
  self.adjacent[vertex] = []
```

This method takes a vertex as an input and adds this vertex to the flow network. It also assignes the adjacent vertices of this vertex to null.

Implementation

def: augmentedPath

Input: A source, a sink and a path
Output: Returns a path to the calling function

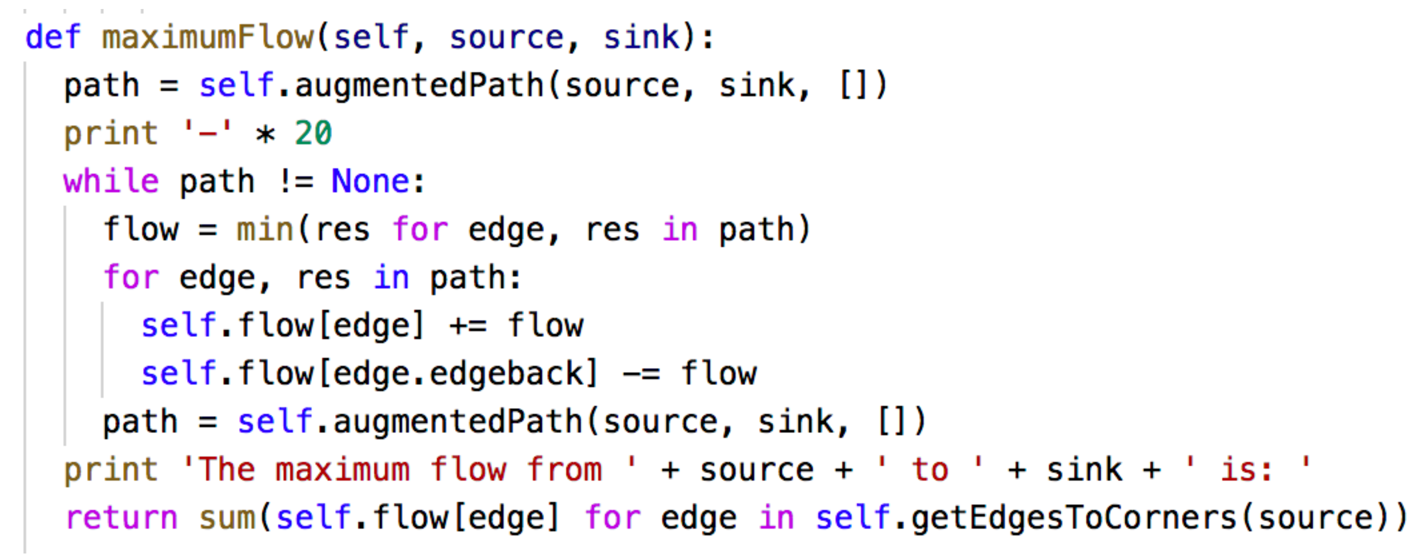


```
def augmentedPath(self, source, sink, path):
    if source == sink:
        return path
    for edge in self.getEdgesToCorners(source):
        residual = edge.weight - self.flow[edge]
        if residual > 0 and not (edge, residual) in path:
            result = self.augmentedPath(edge.sink, sink, path + [(edge, residua
            if result != None:
                return result
```

If the source and sink are same, we cannot find any path, so we return the path without changing it. If the source is different from path, then we get every edge from this vertex and iterate through them. We then check if the residual capacity for that edge is greater than 0 and the edge is not in the path for that residual capacity. If the condition is true, the we iterate through that edge to find the next augmented path. Else, we return the path for that edge as none.

def: maximumFlow

Input: A source, a sink
Output: Returns the maximum flow to the calling function for that source and sink.



```
def maximumFlow(self, source, sink):
    path = self.augmentedPath(source, sink, [])
    print '-' * 20
    while path != None:
        flow = min(res for edge, res in path)
        for edge, res in path:
            self.flow[edge] += flow
            self.flow[edge.edgeback] -= flow
        path = self.augmentedPath(source, sink, [])
    print 'The maximum flow from ' + source + ' to ' + sink + ' is: '
    return sum(self.flow[edge] for edge in self.getEdgesToCorners(source))
```

In this definition, we calculate the augmented path from source to sink and store it in path. If the source and sink are same, then there will not be any path. So, we iterate until the path is none. During this iteration, we calculate the flow for each path as the minimum of residual capacity of an edge in each augmented path. We then subtract this flow from the edge back of each flow and call the augmented path iteration. The maximum flow is sum of all the flows in all the augmented path.

Testing Code

Source: test_program.py

Test code for this problem creates a number of non-trivial inputs and calls the definition maximumFlow() with the input parameters. It then assigns the output to another variable called flow and outputs the flowt to the console. It then checks the correctness of the output using an assert statement. In this test file, 8 different scenarios are used to test the correctness of the problem code in program.py.

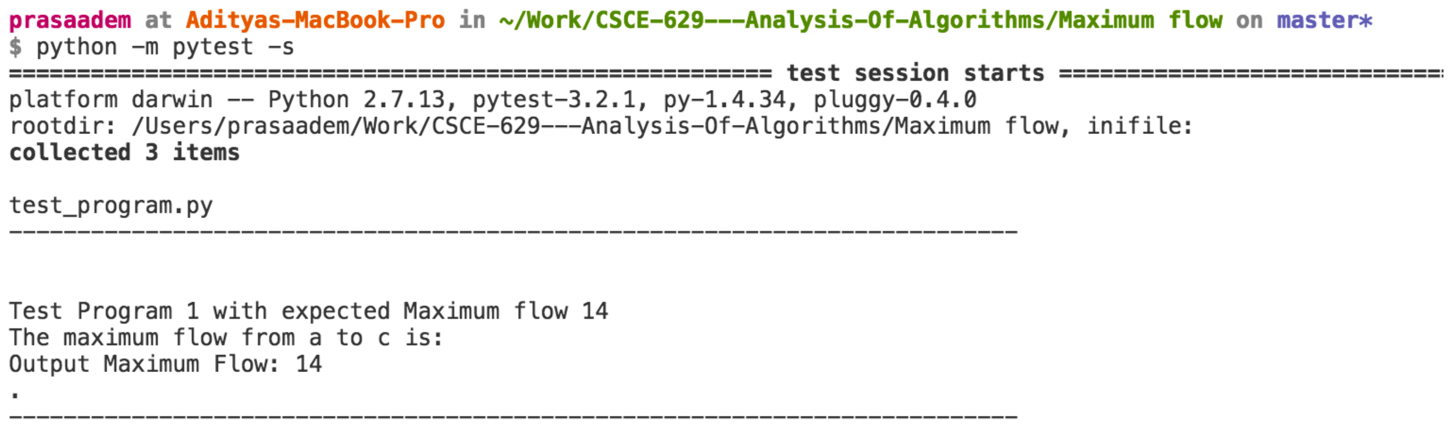
Results:

- All the 8 test scenarios passed successfully.

Example 1

```
g = Flow()
map(g.addCorners, ['a', 'b', 'c', 'd', 'e', 'f'])
g.addOneEdge('a', 'b', 9)
g.addOneEdge('a', 'c', 8)
g.addOneEdge('b', 'c', 6)
g.addOneEdge('b', 'd', 4)
g.addOneEdge('c', 'e', 5)
g.addOneEdge('c', 'e', 2)
g.addOneEdge('d', 'f', 8)
g.addOneEdge('d', 'f', 2)
```

The maximum expected output flow from a to c is 14. Expected flow is 14



```
prasaadem at Adityas-MacBook-Pro in ~/Work/CSCE-629---Analysis-Of-Algorithms/Maximum flow on master*
$ python -m pytest -s
========================================================== test session starts ============================
platform darwin -- Python 2.7.13, pytest-3.2.1, py-1.4.34, pluggy-0.4.0
rootdir: /Users/prasaadem/Work/CSCE-629---Analysis-Of-Algorithms/Maximum flow, inifile:
collected 3 items

test_program.py
-----------------------------------------------------------------------------


Test Program 1 with expected Maximum flow 14
The maximum flow from a to c is:
Output Maximum Flow: 14
.
-----------------------------------------------------------------------------
```

Example 2

```
g = Flow()
map(g.addCorners, ['a', 'b', 'c', 'd', 'e', 'f'])
g.addOneEdge('a', 'b', 9)
g.addOneEdge('a', 'c', 8)
g.addOneEdge('b', 'c', 6)
g.addOneEdge('b', 'd', 4)
g.addOneEdge('c', 'e', 5)
g.addOneEdge('c', 'e', 2)
g.addOneEdge('d', 'f', 8)
g.addOneEdge('d', 'f', 2)
```

The maximum expected output flow from a to b is 9. Expected flow is 9

```
Test Program 2 with expected Maximum flow 9
The maximum flow from a to b is:
Output Maximum Flow: 9
.
----------------------------------------------------------------------
```

Example 3

```
        g = Flow()
        map(g.addCorners, ['a', 'b', 'c', 'd', 'e', 'f'])
        g.addOneEdge('a', 'b', 9)
        g.addOneEdge('a', 'c', 8)
        g.addOneEdge('b', 'c', 6)
        g.addOneEdge('b', 'd', 4)
        g.addOneEdge('c', 'e', 5)
        g.addOneEdge('c', 'e', 2)
        g.addOneEdge('d', 'f', 8)
        g.addOneEdge('d', 'f', 2)
```

The maximum expected output flow from a to d is 4. Expected flow is 44

```
Test Program 3 with expected Maximum flow 4
The maximum flow from a to d is:
Output Maximum Flow: 4
.

======================================================= 3 passed in 0.01 seconds =========================
```

Information

Operating System: Mac OS X High Sierra
Programming Language: Python Version 2.7.13
Testing: Unit testing using pytest

Commands:

- python program.py
- python –m pytest –s (“-s” is for printing print statements to console during testing)

Installation:

- http://www.pyladies.com/blog/Get-Your-Mac-Ready-for-Python-Programming/ (Reference for python)
- https://docs.pytest.org/en/latest/getting-started.html (Reference for pytest)

Details:

- Student Name: Aditya Emani
- Student UIN: 524002563
- Course: CSCE 629 – Analysis of Algorithms
- Semester: Fall 2016
- https://github.com/prasaadem/CSCE-629---Analysis-Of-Algorithms.git