CAMINO COSTE MÍNIMO DIJKSTRA - O(V^2) (O(E log V) con bin heap)

1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3) While *sptSet* doesn't include all vertices
....**a)** Pick a vertex u which is not there in *sptSet* and has minimum distance value.
....**b)** Include u to *sptSet*.
....**c)** Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

```
# Python program for Dijkstra's single
# source shortest path algorithm. The program is
# for adjacency matrix representation of the graph

# Library for INT_MAX
import sys

class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def printSolution(self, dist):
        print "Vertex tDistance from Source"
        for node in range(self.V):
            print node,"t",dist[node]

    # A utility function to find the vertex with

    # minimum distance value, from the set of vertices

    # not yet included in shortest path tree
    def minDistance(self, dist, sptSet):

        # Initilaize minimum distance for next node
        min = sys.maxint

        # Search not nearest vertex not in the
        # shortest path tree
        for v in range(self.V):
            if dist[v] < min and sptSet[v] == False:
                min = dist[v]

            # Pick the minimum distance vertex from
            # the set of vertices not yet processed.
            # u is always equal to src in first iteration
            u = self.minDistance(dist, sptSet)

            # Put the minimum distance vertex in the
            # shotest path tree
            sptSet[u] = True

            # Update dist value of the adjacent vertices
            # of the picked vertex only if the current
            # distance is greater than new distance and
            # the vertex in not in the shotest path tree
            for v in range(self.V):
                if self.graph[u][v] > 0 and sptSet[v] == False and
                   dist[v] > dist[u] + self.graph[u][v]:
                        dist[v] = dist[u] + self.graph[u][v]

        self.printSolution(dist)

# Driver program
g  = Graph(9)
g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0]
```

PRIM MST - O(V^2) (O(E log V) con bin heap)

1) Create a set *mstSet* that keeps track of vertices already included in MST.

2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.

3) While mstSet doesn't include all vertices

....**a)** Pick a vertex *u* which is not there in *mstSet* and has minimum key value.

....**b)** Include *u* to mstSet.

....**c)** Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*.

The idea of using key values is to pick the minimum weight edge from cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertice**.**

KRUSKAL MST - O(ElogE) or O(ElogV)

Un grafo conexo mínimo tiene V-1 lados, donde V = #Vertices, E = #Lados

1. Sort all the edges in non-decreasing order of their weight.

2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Kadane Max Sum Contiguous Subarray - O(N)

```
# Function to find the maximum contiguous subarray
from sys import maxint
def maxSubArraySum(a,size):
    max_so_far = -maxint - 1
    max_ending_here = 0

 #Probamos las secuencias que acaban en i desde 1 hasta n
    for i in range(0, size):
        max_ending_here = max_ending_here + a[i]
        if (max_so_far < max_ending_here):
            max_so_far = max_ending_here

        if max_ending_here < 0:
            max_ending_here = 0
    return max_so_far
```

Largest Common Subsequence NAIVE (LCS) - O(2^N)

```
def lcs(X, Y, m, n):

    if m == 0 or n == 0:
       return 0;
    elif X[m-1] == Y[n-1]:
       return 1 + lcs(X, Y, m-1, n-1);
    else:
       return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
```

Largest Common Subsequence (LCS) - O(mn)

```
def lcs(X , Y):
    # find the length of the strings
    m = len(X)
    n = len(Y)

    # declaring the array for storing the dp values
    L = [[None]*(n+1) for i in xrange(m+1)]

    """Following steps build L[m+1][n+1] in bottom up fashion
    Note: L[i][j] contains length of LCS of X[0..i-1]
    and Y[0..j-1]"""
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0 :
                L[i][j] = 0
            elif X[i-1] == Y[j-1]:
                L[i][j] = L[i-1][j-1]+1
            else:
                L[i][j] = max(L[i-1][j] , L[i][j-1])

    # L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]
    return L[m][n]
#end of function lcs
```

Levenshtein NAIVE - O(3^m)

```
# A Naive recursive Python program to find minimum number
# operations to convert str1 to str2
def editDistance(str1, str2, m , n):

    # If first string is empty, the only option is to
    # insert all characters of second string into first
    if m==0:
         return n

    # If second string is empty, the only option is to
    # remove all characters of first string
    if n==0:
        return m

    # If last characters of two strings are same, nothing
    # much to do. Ignore last characters and get count for
    # remaining strings.
    if str1[m-1]==str2[n-1]:
        return editDistance(str1,str2,m-1,n-1)

    # If last characters are not same, consider all three
    # operations on last character of first string, recursively
    # compute minimum cost for all three operations and take
    # minimum of three values.
    return 1 + min(editDistance(str1, str2, m, n-1),# Insert
                   editDistance(str1, str2, m-1, n),# Remove
                   editDistance(str1, str2, m-1, n-1)      #
Replace
                   )
```

Levenshtein PRO - O(m x n)

```
# A Dynamic Programming based Python program for edit
# distance problem
def editDistDP(str1, str2, m, n):
    # Create a table to store results of subproblems
    dp = [[0 for x in range(n+1)] for x in range(m+1)]

    # Fill d[][] in bottom up manner
    for i in range(m+1):
        for j in range(n+1):

            # If first string is empty, only option is to
            # insert all characters of second string
            if i == 0:
                dp[i][j] = j     # Min. operations = j

            # If second string is empty, only option is to
            # remove all characters of second string
            elif j == 0:
                dp[i][j] = i     # Min. operations = i

            # If last characters are same, ignore last char
            # and recur for remaining string
            elif str1[i-1] == str2[j-1]:
                dp[i][j] = dp[i-1][j-1]

            # If last character are different, consider all
            # possibilities and find minimum
            else:
                dp[i][j] = 1 + min(dp[i][j-1],      # Insert
                                   dp[i-1][j],      # Remove
                                   dp[i-1][j-1])    # Replace

    return dp[m][n]
```

Longest Increasing Subsequence (LIS) - O(n^2)

Secuencia con los elementos ordenados de menor a mayor
0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15 -> 0, 2, 6, 9, 11, 15

```
 Dynamic programming Python implementation of LIS problem

# lis returns length of the longest increasing subsequence
# in arr of size n
def lis(arr):
    n = len(arr)

    # Declare the list (array) for LIS and initialize LIS
    # values for all indexes
    lis = [1]*n

    # Compute optimized LIS values in bottom up manner
    for i in range (1 , n):
        for j in range(0 , i):
            if arr[i] > arr[j] and lis[i]< lis[j] + 1 :
                lis[i] = lis[j]+1

    # Initialize maximum to 0 to get the maximum of all
    # LIS
    maximum = 0

    # Pick maximum of all LIS values
    for i in range(n):
        maximum = max(maximum , lis[i])

    return maximum
# end of lis function
```

NOTA: Tal vez Kadane sobre pesos de 1 daría las posiciones de inicio y final de esto.

Convex Hull - O(n^2)

```
# Python para minimo poligono convexo
# To find orientation of ordered triplet (p, q, r).
# The function returns following values
# 0 --> p, q and r are colinear
# 1 --> Clockwise
# 2 --> Counterclockwise
def orientation(p, q, r):
    val = (q[1] - p[1]) * (r[0] - q[0]) - (q[0] - p[0]) * (r[1] -
q[1])

    if val == 0:
         return 0
    elif val > 0:
        return 1
    else:
        return -1
    return -1

# Prints convex hull of a set of n points.
def convexHull(points, n):
    # There must be at least 3 points
    if (n < 3):
        return

    hull = []

    # Find the leftmost point
    l = 0
    for i in range(n):
        if (points[i][0] < points[l][0]):
            l = i

    #Start from leftmost point, keep moving counterclockwise
    #until reach the start point again.  This loop runs O(h)
    #times where h is number of points in result or output.
    p = l
    #Add current point to result
    hull.append(points[p])

    #Search for a point 'q' such that orientation(p, x,
    #q) is counterclockwise for all points 'x'. The idea
    #is to keep track of last visited most counterclock-
    #wise point in q. If any point 'i' is more counterclock-
    #wise than q, then update q.
    q = (p+1)%n
    for i in range(n):
       #If i is more counterclockwise than current q, then
       #update q
       if (orientation(points[p], points[i], points[q]) == -1):
           q = i;

    #Now q is the most counterclockwise with respect to p
    #Set p as q for next iteration, so that q is added to
    #result 'hull'
    p = q;

    while p!=l:
        hull.append(points[p])
        q = (p+1)%n
        for i in range(n):
           if (orientation(points[p], points[i], points[q]) ==
-1):
               q = i;
        p = q;
      return hull
```

BFS

```
def bfs(graph, start):
  visited, queue = set(), [start]
  while queue:
vertex = queue.pop(0)
if vertex not in visited:
 visited.add(vertex)
 queue.extend(graph[vertex] - visited)
  return visited
bfs(graph, 'A') # {'B', 'C', 'A', 'F', 'D', 'E'}

def bfs_paths(graph, start, goal):
  queue = [(start, [start])]
 while queue:
(vertex, path) = queue.pop(0)
for next in graph[vertex] - set(path):
if next == goal:
            yield path + [next]
else:
queue.append((next, path + [next]))
list(bfs_paths(graph, 'A', 'F')) # [['A', 'C', 'F'], ['A', 'B',
'E', 'F']]
def shortest_path(graph, start, goal):
 try:
      return next(bfs_paths(graph, start, goal))
 except StopIteration:
      return None

shortest_path(graph, 'A', 'F') # ['A', 'C', 'F']
```

DFS

```
def dfs(graph, start):
  visited, stack = set(), [start]
  while stack:
       vertex = stack.pop()
       if vertex not in visited:
 visited.add(vertex)
 stack.extend(graph[vertex] - visited)
  return visited

dfs(graph, 'A') # {'E', 'D', 'F', 'A', 'C', 'B'}

#Returns all paths from start to goal
def dfs_paths(graph, start, goal):
 stack = [(start, [start])]
 while stack:
       (vertex, path) = stack.pop()
       for next in graph[vertex] - set(path):
             if next == goal:
             yield path + [next]
       else:
stack.append((next, path + [next]))

list(dfs_paths(graph, 'A', 'F')) # [['A', 'C', 'F'], ['A', 'B',
'E', 'F']]
```

NOTAS:

1. Yield se usa con generators, que no están siempre en memoria, solo cuando se ejecutan: mygenerator = (x*x for x in range(3))
2. Se pueden usar listas como stacks en python al usar list.append() seguido de list.pop() sin argumentos (pop del último)
3. Análogamente se pueden usar las listas como colas, con list.popleft (=list.pop(0))
4. list.extends() es un append de una colección de elementos

Bellmann-Ford - O(VE)

Given a graph and a source vertex *src* in graph, find shortest paths from *src* to all vertices in the given graph. The graph may contain negative weight edges.

We have discussed Dijkstra's algorithm for this problem. Dijkstra's algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap). *Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford*

```
# The main function that finds shortest distances from src to
    # all other vertices using Bellman-Ford algorithm.  The
function
    # also detects negative weight cycle
    def BellmanFord(self, src):

        # Step 1: Initialize distances from src to all other
vertices
        # as INFINITE
        dist = [float("Inf")] * self.V
        dist[src] = 0


        # Step 2: Relax all edges |V| - 1 times. A simple shortest
        # path from src to any other vertex can have at-most |V| -
1
        # edges
        for i in range(self.V - 1):
            # Update dist value and parent index of the adjacent
vertices of
            # the picked vertex. Consider only those vertices
which are still in
            # queue
            for u, v, w in self.graph:
                if dist[u] != float("Inf") and dist[u] + w <
dist[v]:
                        dist[v] = dist[u] + w
```

```
        # Step 3: check for negative-weight cycles.  The above
step
        # guarantees shortest distances if graph doesn't contain
        # negative weight cycle.  If we get a shorter path, then
there
        # is a cycle.

        for u, v, w in self.graph:
                if dist[u] != float("Inf") and dist[u] + w <
dist[v]:
                        print "Graph contains negative weight
cycle"
                        return

        # print all distance
        self.printArr(dist)
```

is O(VE), which is more than Dijkstra.

Floyd-Warshall (All Pairs Shortest Path) - $O(V^3)$

```
# Python Program for Floyd Warshall Algorithm

# Number of vertices in the graph
V = 4

# Define infinity as the large enough value. This value will be
# used for vertices not connected to each other
INF  = 99999

# Solves all pair shortest path via Floyd Warshall Algorithm
def floydWarshall(graph):

    """ dist[][] will be the output matrix that will finally
        have the shortest distances between every pair of vertices
"""
    """ initializing the solution matrix same as input graph
matrix
    OR we can say that the initial values of shortest distances
    are based on shortest paths considering no
    intermediate vertices """

    dist = map(lambda i : map(lambda j : j , i) , graph)

    """ Add all vertices one by one to the set of intermediate
     vertices.
     ---> Before start of an iteration, we have shortest distances
     between all pairs of vertices such that the shortest
     distances consider only the vertices in the set
    {0, 1, 2, .. k-1} as intermediate vertices.
      ----> After the end of a iteration, vertex no. k is
     added to the set of intermediate vertices and the
    set becomes {0, 1, 2, .. k}
    """
    for k in range(V):

        # pick all vertices as source one by one
        for i in range(V):

            # Pick all vertices as destination for the
            # above picked source
            for j in range(V):

                # If vertex k is on the shortest path from
                # i to j, then update the value of dist[i][j]
                dist[i][j] = min(dist[i][j] ,
                                 dist[i][k]+ dist[k][j]
                                )
    printSolution(dist)
```

NOTA: Se asigna distancia infinito (INF) si no hay camino

Max Flow (Ford-Fulkerson) - O(max_flow*E)

The following is simple idea of Ford-Fulkerson algorithm:
1) Start with initial flow as 0.
2) While there is a augmenting path from source to sink.
Add this path-flow to flow.
3) Return flow.

```
# Returns tne maximum flow from s to t in the given graph
    def FordFulkerson(self, source, sink):

        # This array is filled by BFS and to store path
        parent = [-1]*(self.ROW)

        max_flow = 0 # There is no flow initially

        # Augment the flow while there is path from source to sink
        while self.BFS(source, sink, parent) :

            # Find minimum residual capacity of the edges along
the
            # path filled by BFS. Or we can say find the maximum
flow
            # through the path found.
            path_flow = float("Inf")
            s = sink
            while(s !=  source):
                path_flow = min (path_flow,
self.graph[parent[s]][s])
                s = parent[s]

            # Add path flow to overall flow
            max_flow +=  path_flow

            # update residual capacities of the edges and reverse
edges
            # along the path
            v = sink
            while(v !=  source):
                u = parent[v]
                self.graph[u][v] -= path_flow
                self.graph[v][u] += path_flow
                v = parent[v]

        return max_flow
```

Knuth-Morris-Pratt Algorithm (fast pattern matching)
O(n+m)

```
def KnuthMorrisPratt(text, pattern):

    '''Yields all starting positions of copies of the pattern in the
text.
Calling conventions are similar to string.find, but its arguments
can be
lists or iterators, not just strings, it returns all matches, not
just
the first one, and it does not need the whole text in memory at
once.
Whenever it yields, it will have read the text exactly up to and
including
the match that caused the yield.'''

    # allow indexing into pattern and protect against change during
yield
    pattern = list(pattern)

    # build table of shift amounts
    shifts = [1] * (len(pattern) + 1)
    shift = 1
    for pos in range(len(pattern)):
        while shift <= pos and pattern[pos] != pattern[pos-shift]:
            shift += shifts[pos-shift]
        shifts[pos+1] = shift
```

```
    # do the actual search
    startPos = 0
    matchLen = 0
    for c in text:
        while matchLen == len(pattern) or \
              matchLen >= 0 and pattern[matchLen] != c:
            startPos += shifts[matchLen]
            matchLen -= shifts[matchLen]
        matchLen += 1
        if matchLen == len(pattern):
            yield startPos
```

Rabin-Karp Algorithm (multiple pattern matching)

Es un Algoritmo de búsqueda de subcadenas simple se basa en tratar cada uno de los grupos de m caracteres del texto (siendo m el número de símbolos del patrón) como un índice de una tabla de valores hash (la llamaremos tabla de dispersión), de manera que si la función hash de los m caracteres del texto coincide con la del patrón es posible que hayamos encontrado un acierto. para verificarlo hay que comparar el texto con el patrón, ya que la función hash elegida puede presentar colisiones.

```
# d is the number of characters in input alphabet
d = 256

# pat  -> pattern
# txt  -> text
# q    -> A prime number

def search(pat, txt, q):
    M = len(pat)
    N = len(txt)
    i = 0
    j = 0
    p = 0    # hash value for pattern
    t = 0    # hash value for txt
    h = 1

    # The value of h would be "pow(d, M-1)%q"
    for i in xrange(M-1):
        h = (h*d)%q

    # Calculate the hash value of pattern and first window
    # of text
    for i in xrange(M):
        p = (d*p + ord(pat[i]))%q
        t = (d*t + ord(txt[i]))%q

    # Slide the pattern over text one by one
    for i in xrange(N-M+1):
        # Check the hash values of current window of text and
        # pattern if the hash values match then only check
        # for characters on by one
        if p==t:
            # Check for characters one by one
            for j in xrange(M):
                if txt[i+j] != pat[j]:
                    break

            j+=1
            # if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
            if j==M:
                print "Pattern found at index " + str(i)

        # Calculate hash value for next window of text: Remove
        # leading digit, add trailing digit
        if i < N-M:
            t = (d*(t-ord(txt[i])*h) + ord(txt[i+M]))%q

            # We might get negative values of t, converting it to
            # positive
            if t < 0:
                t = t+q

# Driver program to test the above function
txt = "GEEKS FOR GEEKS"
pat = "GEEK"
q = 101 # A prime number
search(pat,txt,q)
```

Miller-Rabin Primality Test

El test de primalidad de Miller-Rabin es un test de primalidad, es decir, un algoritmo para determinar si un número dado es primo $O(k \log^3 n)$ where k is the number of different values of a that we test

```
def miller_rabin(n, k):
    # The optimal number of rounds (k) for this test is 40
    # for justification

    if n == 2:
        return True
    if n % 2 == 0:
        return False
    r, s = 0, n - 1
    while s % 2 == 0:
        r += 1
        s //= 2
    for _ in xrange(k):
        a = random.randrange(2, n - 1)
        x = pow(a, s, n)
        if x == 1 or x == n - 1:
            continue
        for _ in xrange(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True
```

Prime generator(sieve)

El generador de primos más rápido

```
from itertools import count

def postponed_sieve():                  # postponed sieve, by Will Ness
    yield 2; yield 3; yield 5; yield 7; # original code David Eppstein,
    sieve = {}                          #Alex Martelli, ActiveState
Recipe 2002
    ps = postponed_sieve()             # a separate base Primes Supply:
    p = next(ps) and next(ps)            # (3) a Prime to add to dict
    q = p*p                              # (9) its sQuare
    for c in count(9,2):                 # the Candidate
        if c in sieve:               # c's a multiple of some base prime
            s = sieve.pop(c)         #     i.e. a composite ; or
        elif c < q:
             yield c                  # a prime
             continue
        else:   # (c==q):             # or the next base prime's square:
            s=count(q+2*p,2*p)       #    (9+6, by 6 : 15,21,27,33,...)
            p=next(ps)               #    (5)
            q=p*p                    #    (25)
        for m in s:                  # the next multiple
            if m not in sieve:       # no duplicates
                break
        sieve[m] = s                 # original test entry:
ideone.com/WFv4f
```

GCD and Euler's Totient Function

```
# Function to return gcd of a and b
def gcd(a, b):
    if a == 0:
        return b
    return gcd(b%a, a)

# A simple method to evaluate Euler Totient Function
def phi(n):
    result = 1
    for i in range(2, n):
        if gcd(i, n) == 1:
            result = result + 1
    return result
```

Gauss-Jordan Elimination

Igual algo de aquí es útil, no creo pero sobra espacio así que yolo (:

```
def gauss_jordan(m, eps = 1.0/(10**10)):
  """Puts given matrix (2D array) into the Reduced Row Echelon Form.
     Returns True if successful, False if 'm' is singular.
     NOTE: make sure all the matrix items support fractions! Int matrix
will NOT work!
     Written by Jarno Elonen in April 2005, released into Public Domain"""
  (h, w) = (len(m), len(m[0]))
  for y in range(0,h):
    maxrow = y
    for y2 in range(y+1, h):    # Find max pivot
      if abs(m[y2][y]) > abs(m[maxrow][y]):
        maxrow = y2
    (m[y], m[maxrow]) = (m[maxrow], m[y])
    if abs(m[y][y]) <= eps:     # Singular?
      return False
    for y2 in range(y+1, h):    # Eliminate column y
      c = m[y2][y] / m[y][y]
      for x in range(y, w):
        m[y2][x] -= m[y][x] * c
  for y in range(h-1, 0-1, -1): # Backsubstitute
    c  = m[y][y]
    for y2 in range(0,y):
      for x in range(w-1, y-1, -1):
        m[y2][x] -=  m[y][x] * m[y2][y] / c
    m[y][y] /= c
    for x in range(h, w):       # Normalize row y
      m[y][x] /= c
  return True

def solve(M, b):
  """
  solves M*x = b
  return vector x so that M*x = b
  :param M: a matrix in the form of a list of list
  :param b: a vector in the form of a simple list of scalars
  """
  m2 = [row[:]+[right] for row,right in zip(M,b) ]
  return [row[-1] for row in m2] if gauss_jordan(m2) else None

def inv(M):
  """
  return the inv of the matrix M
  """
  #clone the matrix and append the identity matrix
  # [int(i==j) for j in range_M] is nothing but the i(th row of the
identity matrix
  m2 = [row[:]+[int(i==j) for j in range(len(M) )] for i,row in
enumerate(M) ]
  # extract the appended matrix (kind of m2[m:,...]
  return [row[len(M[0]):] for row in m2] if gauss_jordan(m2) else None
def zeros( s , zero=0):
    """
    return a matrix of size `size`
    :param size: a tuple containing dimensions of the matrix
    :param zero: the value to use to fill the matrix (by default it's zero
)
    """
    return [zeros(s[1:] ) for i in range(s[0] ) ] if not len(s) else zero
```