```python
from itertools import combinations
from collections import defaultdict

def travelling_salesman(G):
    # Construïm la matriu d'adjacència de G. Aquesta matriu té a cada parella de nodes connectats (i,j) el cost de l'aresta.
    adjMatrix = nx.adjacency_matrix(G)
    n = len(G.nodes())

    # Inicialitzem un diccionari de tipus defaultdict. En el cas d'accedir a una clau inexistent, li posarem el valor infinit.
    memo = defaultdict(lambda: float('inf'))

    # Els nodes es diuen 0,1,...,n-1
    # Inserim al diccionari:
    # Clau: (Node, tupla).
    # Valor: El cost inicial de viatjar des del node 0 al node x. El node 0 serà el nostre node inicial sempre.
    for x in range(1, n):
        memo[x , ()] = adjMatrix[0,x]


    # Considerem diferents longituds de camins. Com que considerem el zero com a node inicial,
    # estarem considerant camins de longitud size + 1.
    for size in range(1, n):

        # El node a visitar a continuació pot ser qualsevol
        for k in range(1,n): # iterate over the 1..N nodes as candiates to be visited

            # Considerem tots els ordres possibles de visitar 'size' nodes.
            # Això consisteix en fer (n-1)! / (size!*(n-size-1)!) operacions.
            # S seran els nodes que hem visitat abans de visitar k
            for S in combinations(range(1,n),size):

                # Si el node k està en aquesta combinació, no ens interessa, ja que no el volem visitar dos cops.
                if k in S:
                    continue

                # Per a cada node 'j' que conté S:
                # - Mirem tots els nodes que conté S i que no són j (nodes ja visitats)
                # - Actualitzem el millor cost.
                # -- Pot ser que sigui un cost que ja havíem trobat venint d'una altra branca: memo[k,S]
                # -- Pot ser que haguem trobat un nou millor cost: adjMatrix[j,k] + memo[j,tup]
                for j in S:
                    tup = tuple([ i for i in S if i!=j ])
                    memo[k,S] = min(memo[k,S], adjMatrix[j,k] + memo[j,tup])


    # Només ens falta tornar al node 0. Considerem els diferents camins que hi porten i escollim el mínim.
    value = min([memo[k,tuple([ i for i in range(1,n) if i!=k])] + adjMatrix[k,0] for k in range(1,n)])

    return value
```

```python
# Matriu d'adjacència del graf que volem crear
A = np.array([[0,4,1,9],
              [3,0,6,11],
              [4,1,0,2],
              [6,5,-4,0]])
#A = np.array([[0,3,4,6],[4,0,1,5],[1,6,0,-4],[9,11,2,0]])
#A = np.array([[0,10,15,20],[10,0,35,25],[15,35,0,30],[20,25,30,0]])
G = nx.from_numpy_matrix(A,create_using=nx.DiGraph)

fig = plt.figure(figsize=(8,7))

# Si voleu executar grafs amb més de 4 nodes, podeu canviar el layout a spring_layout
pos=nx.planar_layout(G)
nx.draw(G, pos,with_labels=True,connectionstyle='arc3, rad = 0.08', ax=fig.gca())
labels = nx.get_edge_attributes(G,'weight')
nx.draw_networkx_edge_labels(G,pos,edge_labels=labels,label_pos=0.3)
plt.show()

res = travelling_salesman(G)
res
```

```python
def minimal_paths(mat):
    # Guardem les dimensions de la matriu
    [i, j] = mat.shape

    # Fem la còpia de la matriu (podem editar directament mat,
    # ja que és la matriu de gradients i només caldria sumar el mínim)
    ret = mat

    # Calculem la distància mínima per a cada cel·la de la matriu
    for x in range(1,i):
        for y in range(j):

            # Trobem el mínim dels veins superiors fent ús de la
            # funció superior_neighbors
            minim = np.min(superior_neighbors(mat, [x,y]))

            # Anem creant la matriu D
            ret[x][y] = mat[x][y] + minim

    return ret
```

```python
def find_min_path(mat):
    # Primer agafem les coordenades inicials del camí mínim i les posem al path mínim
    fila_inici = mat.shape[0]-1

    # Utilitzem argmin per trobar la posició on es troba el mínim de l'última fila
    columna_inici = np.argmin(mat[-1], 0)

    min_path = [(fila_inici, columna_inici)]

    # Inicialitzo una llista de veïns que utilitzarem a l'hora d'iterar
    neighbors = []

    # Mirem per cada iteració quin és el camí mínim per la cel·la actual
    fila_actual = fila_inici
    columna_actual = columna_inici

    # El tercer paràmetre del range indica que començarem per la part inferior
    for i in range(len(mat)-2, -1, -1):
        # La fila sempre anirà restant 1
        fila_actual -= 1

        # Guardem a la llista de veïns el que ens retorni el mètode
        # superior_neighbors en el punt actual
        neighbors = superior_neighbors(mat, [fila_actual, columna_actual])

        # Si la llista no està buida, busquem en quina posició de la
        # llista es troba el mínim
        if (len(neighbors) != 0):
            pos_min = np.argmin(neighbors, 0)

        # Si la llista conté tres veïns
        if(len(neighbors) == 3):
            # Si es troba a la posició 0, vol dir que haurem d'anar
            # una columna a l'esquerra
            if(pos_min == 0):
                columna_actual -= 1

            # Si es troba a la posició 2, vol dir que haurem d'anar
            # una columna a la dreta
            elif(pos_min == 2):
                columna_actual += 1

        # Si la llista conté dos veïns
        if(len(neighbors) == 2):
            # Si es troba a la posició 0 i la columna actual és del
            # marge dret, vol dir que haurem d'anar una columna a l'esquerra
            if(pos_min == 0 and columna_actual == mat.shape[1]-1):
                columna_actual -= 1

            # Si es troba a la posició 1 i la columna actual és del marge
            # esquerra, vol dir que haurem d'anar una columna a la dreta
            elif(pos_min == 1 and columna_actual == 0):
                columna_actual += 1

        # Si no compleix cap de les condicions anteriors, haurem de quedar-nos
        # en la mateixa columna

        # Anem afegint al min_path les coordenades del path que estem creant
        min_path.append((fila_actual, columna_actual))

    return min_path
```

```python
def get_gradient_v2(im, i):
    # Transformem la imatge a un sol canal (blanc i negre)
    im_blackwhite = np.dot(im[...,:3], [0.299, 0.587, 0.114])

    # Calculem el gradient usant sobel
    gradient = np.abs(nd.sobel(im_blackwhite))

    # Fem ús de slicing per assignar el valor negatiu als punts de dins del pat...
    gradient[patch[0][0]:patch[1][0]+1, patch[0][1]:patch[0][1]+1+i] = -100

    return gradient
```

```python
def delete_path(im, path):
    # Guardem el shape de la imatge
    shape = im.shape

    # Creem una nova matriu amb la shape de la
    # imatge inicial i de tipus boolean (totes les
    # posicions seran iniciades a True)
    im_new = np.ones((im.shape), dtype = bool)

    # Iterem per a cada coordenada del path i canviem
    # la posició a False (serà el camí que haurem d'eliminar)
    for i, j in path:
        im_new[i, j] = False

    # Canviem la matriu inicial eliminant les posicions en
    # que hi hagi False (és com si fesim una màscara)
    im = im[im_new]

    # Al fer el pas anterior, tenim una matriu flattened,
    # llavors haurem de fer un reshape sabent que tindrem una columna
    # menys al haver eliminat el path
    im = np.reshape(im, (shape[0], shape[1]-1, shape[2]))

    return im
```

```python
def reduce_image(im, N=100):
    # Fem la primera eliminació fora de la iteració per tal de poder mostrar el primer camí que eliminem

    # Obtenim el gradient de la imatge
    G = get_gradient(im)

    # Calculem els camins mínims donada la matriu de gradients
    D = minimal_paths(G)

    # Amb la matriu D trobarem el path mínim que comença des de l'última fila i acaba a la primera
    y = find_min_path(D)

    # Fem la còpia de la matriu i pintem el primer path que elimina
    first_path = im.copy()
    first_path = add_min_path(first_path, y)

    # Creem una nova imatge en la que hem eliminat el path anterior
    updated = delete_path(im, y)

    # Iterem N-1 vegades ja que la primera iteració l'hem fet fora del bucle
    for i in range(N-1):
        # Repetim els passos explicats anteriorment N-1 vegades
        G = get_gradient(updated)
        D = minimal_paths(G)
        y = find_min_path(D)
        updated = delete_path(updated, y)

    # Usem la funció show_row per mostrar les imatges amb els seus títols.
    im_titles = [(im, 'Original'), (first_path, 'Primer camí que elimina'), (updated, 'Imatge resultant')]
    show_row(im_titles)
```

```python
def remove_patch(im, patch):
    # patch_delete serà la imatge còpia sobre la que hi dibuixarem el patch de color vermell
    patch_delete = im.copy()
    patch_delete = add_patch(patch_delete, patch)

    # La primera iteració es fa fora del bucle, els passos són els mateixos que en l'exercici anterior
    # amb la diferència que ara a get_gradient li passem el nº de columna on ens trobem
    G = get_gradient_v2(im, 0)
    D = minimal_paths(G)
    y = find_min_path(D)
    updated = delete_path(im, y)

    # Iterem per la resta de columnes seguint els mateixos passos
    for i in range(patch[1][1] - (patch[0][1])):
        G = get_gradient_v2(updated, i+1)
        D = minimal_paths(G)
        y = find_min_path(D)
        updated = delete_path(updated, y)

    # Usem la funció show_row per mostrar les imatges amb els seus títols.
    im_titles = [(im, 'Original'), (patch_delete, 'Patch a eliminar'), (updated, 'Imatge resultant')]
    show_row(im_titles)
```

```java
public static int coinChange(int[] coins, int amount) {

  if (coins == null) throw new IllegalArgumentException("Coins array is null");
  if (coins.length == 0) throw new IllegalArgumentException("No coin values :/");

  final int N = coins.length;
  // Initialize table and set first row to be infinity
  int[][] dp = new int[N + 1][amount + 1];
  java.util.Arrays.fill(dp[0], INF);
  dp[1][0] = 0;

  // Iterate through all the coins
  for (int i = 1; i <= N; i++) {

    int coinValue = coins[i - 1];
    for (int j = 1; j <= amount; j++) {

      // Consider not selecting this coin
      dp[i][j] = dp[i - 1][j];

      // Try selecting this coin if it's better
      if (j - coinValue >= 0 && dp[i][j - coinValue] + 1 < dp[i][j]) {
        dp[i][j] = dp[i][j - coinValue] + 1;
      }
    }
  }

  // The amount we wanted to make cannot be made :/
  if (dp[N][amount] == INF) return -1;

  // Return the minimum number of coins needed
  return dp[N][amount];
```

```python
def fib_rec(n):
    if (n==0) or (n==1):
        return n
    return fib_rec(n-1)+fib_rec(n-2)
```

```python
# Solució Bottom-up
def fibonacci(n):
    if n < 0:
        print("Incorrect output")
        return 0

    f = [0,1]

    for i in range(2, n+1):
        f.append(f[i-1] + f[i-2])
    return f[n]
```

```python
def fib_top_down(n, dp=None):
    # Inicialització d'un array pa emmagatzemar els valors
    # ja calculats
    if dp is None:
        dp = [0]*(n+1)

    # Casos base
    if (n==0) or (n==1):
        return n

    # Cas en que ja hem calculat el valor prèviament
    if dp[n] != 0:
        return dp[n]

    # Cas on no hem calculat encara el valor previ
    dp[n] = fib_top_down(n-1, dp) + fib_top_down(n-2, dp)

    return dp[n]
```

```python
def fib_bottom_up(n):
    # Emmagatzem en una llista els càlculs previs
    # Guardem els dos primers valors 1 i 1
    # I inicialitzem la resta de valors a zero
    dp = [0, 1] + [0]*(n-1)

    # Anem calculant i afegint el nombre de
    # fibonacci següent a partir dels dos anteriors
    for i in range(2,n+1):
        dp[i] = dp[i-1]+dp[i-2]

    # Retornem el valor demanat
    return dp[n]
```

```python
def fib_bottom_up_v2(n):
    # Inicialitzem dues variables
    #que ens serviran per a calcular
    # el següent valor
    a, b = 0, 1
    if (n==0) or (n==1):
        return n

    # Usem la doble assignació de
    # python per anar calculant el
    # següent valor a partir dels dos anteriors
    for i in range(n-1):
        a, b = b, a+b
    return b
```

```python
def rodCut_top_down(price, n, memo = None):
    if memo is None:
        memo =[-1]*(n+1)

    if n <= 0:
        return 0

    if(memo[n] > 0):
        return memo[n]

    max_val = -1

    for i in range(0, n):
        max_val = max(max_val, price[i] + rodCut_top_down(price, n-i-1))

    memo[n] = max_val
    return memo[n]
```

```python
def rod_cutting_dp_bottom_up(N, prices):
    # Inicialitzem la memòria
    dp = [0]*(N+1)

    # Recorrem total la llista resoldrem tots els subproblemes incrementalment
    # començant pel cas on només tenim una peça i anant augmentant
    for i in range(1,N+1):
        max_val = -float('inf')

        # El màxim d'un subcas serà el màxim entre tots els subcasos anteriors.
        # Amb l'ajuda d'aquest 'for' seleccionem el millor valor considerant:
        # - Venem la peça prices[j]
        # - Prenem el millor valor guardat a la taula per a la peça restant, dp[i-j-1]
        for j in range(i):
            max_val = max(max_val, prices[j] + dp[i-j-1])

        # Assignem el millor valor.
        dp[i] = max_val

    print(dp)
    return dp[N]
```

```python
def rod_cutting_dp_top_down(N, prices, dp=None):
    # Inicialitzem la memòria
    if dp is None:
        dp = [0]*(N+1)

    # Casos base
    if N==0:
        return 0
    if N==1:
        return prices[0]

    # Aprofitem els càlculs previs
    if dp[N]!=0:
        return dp[N]

    # Omplim la memòria amb el càlcul recursiu
    dp[N] = max([prices[N-i-1]+rod_cutting_dp_top_down(i, prices, dp) for i in range(0, N)])

    return dp[N]
```

```python
def rod_cutting_rec(N, prices):
    # Casos base
    if N==0:
        return 0
    if N==1:
        return prices[N-1]

    # Crida recursiva trobant el màxim dels talls
    turn max([prices[N-i-1]+rod_cutting_rec(i, prices) for i in range(0, N)])
```

```python
def knapsack(W, weights, values, n):
    # Inicialitzem la taula de programació dinàmica.
    # Cada cel·la (i,w) ens retornarà el valor màxim que
    # podem obtenir considerant els 'i' primers objectes
    # tals que el seu pes total és inferior o igual a w.
    K = [[0 for x in range(W+1)]for x in range (n+1)]

    # La variable i ens controla els diferents objectes que tenim.
    # L'usem per accedir a la llista de 'values'
    for i in range(n+1):

        # La variable w ens controla el pes total que estem considerant fins
        # al moment
        for w in range(W+1):

            # Si no tenim cap objecte o si el pes màxim que estem considerant
            # és 0, no podem posar cap objecte.
            if(i==0) or (w==0):
                K[i][w] = 0 # També podríem posar un 'continue' doncs ja ho hem
                            # inicialitzat tot a zero.

            # En el cas en que l'objecte a considerar estigui dins el pes permès,
            # mirem si el podem afegir (si maximitza el valor)
            elif weights[i-1] <= w:
                K[i][w] = max(values[i-1] + K[i-1][w-weights[i-1]], K[i-1][w])

            # En cas que l'objecte no estigui dins el pes permès, no modifiquem el
            # valor
            else:
                K[i][w] = K[i-1][w]

    # El cas que ens interessa
    value = K[n][W]

    # Print
    for line in K:
        print(line)
    # end print

    # Llista on guardarem els elements que inserirem a la motxilla
    backtrack_items = []

    # Mida de la motxilla
    max_cap = W

    # Backtracking per veure quins elements hem seleccionat
    for i in range(len(weights), 0, -1):
        if K[i][max_cap] != K[i-1][max_cap]:
            backtrack_items.append(i)
            pos_item = i-1
            max_cap -= weights[pos_item]

    print(backtrack_items)
    return value
```

```python
def lis_dp_bottom_up(seq):
    # Comprovacions inicials
    if (seq is None) or (len(seq) == 0):
        return 0

    n = len(seq)
    length = 0

    # Quan comencem, cada element individual
    # té un LIS de exactament 1, així que cada
    # índex s'inicialitza a 1
    dp = [1] * (n)

    # Iterant l'array d'esquerra a dreta, actualitzem
    # el valor de dp[j] si es compleixen 2 condicions:
    # 1. El valor en i és menor al valor en j
    # 2. Actualitzar el valor de dp[j] a dp[i]+1 és millor
    for i in range(n):
        for j in range(i+1, n):
            if(seq[i] < seq[j]) and (dp[j] < dp[i] + 1):
                dp[j] = dp[i] + 1

        # Track the LIS
        if dp[i] > length:
            length = dp[i]

    return length
```

```python
def lps_dp_v1(seq):n = len(seq)

    # Creem una taula per guardar els resultats
    # dels subproblemes
    dp = [[0 for x in range(n)] for y in range(n)]

    # Strings de len == 1 són palíndroms de len == 1
    for i in range(n):
        dp[i][i] = 1

    # Construïm la taula. Note that the lower diagonal
    # values of table are useless and not filled in the
    # process.
    # cl is length of substring
    for cl in range(2, n+1):
        for i in range(n-cl+1):
            j = i+cl-1

            if seq[i] == seq[j] and cl == 2:
                dp[i][j] = 2
            elif seq[i] == seq[j]:
                dp[i][j] = dp[i+1][j-1] + 2
            else:
                dp[i][j] = max(dp[i][j-1], dp[i+1][j])

    return dp[0][n-1]
```

```python
def levensthein(word_B, word_A, del_cost = 1, ins_cost = 1, sub_cost = 1):
    #Mirem la llargada de la línia
    text_length = len(word_A) + 1

    #Calculem la llargada del patro
    patro_length = len(word_B) + 1

    #Inicialitzem la matriu
    distance_matrix = [[0] * text_length for x in range(patro_length)]
    operation_matrix = [[""] * text_length for j in range(patro_length)]

    sequencia_operacions = []

    for i in range(patro_length):
        distance_matrix[i][0] = i * ins_cost
        operation_matrix[i][0] = "I"
    for j in range(text_length):
        distance_matrix[0][j] = j * del_cost
        operation_matrix[0][j] = "D"
    operation_matrix[0][0] = 'C'

    #Omplim la matriu de distàncies:
    for i in range(1, patro_length):
        for j in range(1, text_length):
            deletion = distance_matrix[i-1][j] + del_cost
            insertion = distance_matrix[i][j-1] + ins_cost
            substitution = distance_matrix[i-1][j-1]

            if word_B[i-1] != word_A[j-1]:
                substitution += sub_cost

            distance_matrix[i][j] = min(insertion, deletion, substitution)

            #Omplim la matriu d'operacions
            if distance_matrix[i][j] == substitution:
                if word_B[i-1] != word_A[j-1]:
                    operation_matrix[i][j] = "S"
                else:
                    operation_matrix[i][j] = "C"

            elif distance_matrix[i][j] == insertion:
                operation_matrix[i][j] = "I"

            else:
                operation_matrix[i][j] = "D"

    #Calculem la distància final i la posició final.
    distancia_minima = min(distance_matrix[patro_length-1])
    posicio_final = distance_matrix[patro_length-1].index(distancia_minima)
```

```python
# solució naïf
def editDistance(str1, str2, m, n):
    # Si el primer string té longitud 0, l'única opció
    # és afegir els caràcters del segon string
    if m == 0:
        return n

    # Si el segon string té longitud 0, l'única opció
    # és afegir els caràcters del primer string
    if n == 0:
        return m

    # Si l'últim caràcter dels dos strings és el mateix
    if str1[m-1] == str2[n-1]:
        return editDistance(str1, str2, m-1, n-1)

    # Si l'últim caràcter dels dos string no és el mateix
    # cridem a la funció recursiva de les diferents accions
    # i afegim la de menor cost+1
    return 1 + min(editDistance(str1, str2, m, n-1), # afegir
                   editDistance(str1, str2, m-1, n), # eliminar
                   editDistance(str1, str2, m-1, n-1)) # reemplaçar
```

```python
def longest_palindrom_subsequence_brute_force(seq, i,j):
    # cas base 1, sequencia d'un únic caràcter
    if i == j:
        return 1

    # cas base 2, sequencia de dos caràcters
    if (seq[i] == seq[j]) and (i + 1 == j):
        return 2

    # Si els dos elements son iguals
    if (seq[i] == seq[j]):
        return longest_palindrom_subsequence_brute_force(seq, i + 1, j - 1) + 2

    # Si son diferents
    return max(longest_palindrom_subsequence_brute_force(seq, i, j - 1),
               longest_palindrom_subsequence_brute_force(seq, i + 1, j))
```

```python
#Càlcul de la posició on comença la subseqüència més semblant i busqueda
# de les operacions realitzades.
x = len(word_B)
y = posicio_final
while ((x != 0) and (y != 0)):
    if operation_matrix[x][y] == "C":
        x, y = x-1, y-1
        sequencia_operacions.append("C")
        print("MAN", word_B[x], " ", word_A[y])
    elif operation_matrix[x][y] == "S":
        x, y = x-1, y-1
        sequencia_operacions.append("S")
        print("SUB", word_B[x], " ", word_A[y])
    elif operation_matrix[x][y] == "I":
        y -= 1
        sequencia_operacions.append("I")
        print("INS", word_B[x])
    else:
        x -= 1
        sequencia_operacions.append("D")
        print("DEL", word_B[x], " ", word_A[y])
posicio_inicial = y
sequencia_operacions = sequencia_operacions[::-1]
return distancia_minima, distance_matrix, sequencia_operacions
```