Contents

1 Setup 2
1.1 Comando de Ejecución 2
1.1.1 run.sh 2

2 Basico 2
2.1 Busqueda Binaria 2
2.1.1 lower_bound.py 2
2.1.2 upper_bound.py 2
2.2 Tabla Aditiva 2
2.2.1 tabla_aditiva.py 2
2.2.2 tabla_aditiva_2D.py 2
2.3 Programación Dinámica 2
2.3.1 sub_set_sum.py 2
2.3.2 cambio_monedas.py 2
2.4 Recurrencias Lineales 2
2.4.1 recurrena_lineal.py 2
2.5 Heap y Heapsort 3
2.5.1 heapsort.py 3
2.5.2 max_heap.py 3

3 Grafos 3
3.1 Leer grafos 3
3.1.1 leer.py 3
3.2 BFS: Busqueda en Anchura 3
3.2.1 bfs.py 3
3.3 Bipartir un grafo 3
3.3.1 bipartir.py 3
3.3.2 bipartir_2.py 3
3.4 Camino Mínimo 3
3.4.1 dijkstra.py 3
3.4.2 floyd_warshall.py 4
3.4.3 bellman_ford.py 4
3.4.4 spfa.py 4
3.5 Union Find 4
3.5.1 Small To Large 4
3.5.2 Path Compression y Union by Size . 4
3.6 MST: Árbol Generador Mínimo 4
3.6.1 kruskal.py 4
3.6.2 prim.py 5
3.7 Componentes Fuertemente Conexas 5
3.7.1 kosaraju_iterativo.py 5
3.7.2 tarjan_iterativo.py 5
3.7.3 grafo_condensado.py 6
3.7.4 2_SAT.py 6
3.8 Components Biconexas, Puentes y Puntos de Articulación 6
3.8.1 componentes_biconexas.py 6
3.9 LCA: Ancestro Común Menor 7
3.9.1 binary_lifting_funcional.py 7
3.9.2 binary_lifting_lca.py 7
3.9.3 lca_sparse_table.py 7

4 Estructuras de Datos 7
4.1 Árbol de Segmentos 7
4.1.1 segment_tree.py 7
4.1.2 segment_tree_lazy_creation.py . . . 8
4.2 Sparse Table 8
4.2.1 sparse_table.py 8

5 Algoritmos 8
5.1 Divide and Conqueer D&C 8
5.1.1 merge_sort.py 8
5.1.2 Teorema Maestro 9
5.2 Técnica de 2 Punteros 9
5.2.1 dos_punteros.py 9
5.2.2 vectores_paralelos.py 9
5.2.3 ventana_deslizante.py 9
5.3 Algoritmo de Mo 9
5.3.1 mo_plantilla.py 9
5.3.2 mo_ejemplo.py 10

6 Matematicas 10
6.1 Números Primos 10
6.1.1 criba_eratostenes.py 10
6.2 Divisores 10
6.2.1 divisores.py 10
6.2.2 divisores_un_numero.py 10
6.3 Divisor Común Mayor 10
6.3.1 euclides.py 10
6.4 Aritmetica Modular 10
6.4.1 aritmetica_modular.py 10
6.5 Combinatoria 10
6.5.1 combinatoria.py 10
6.5.2 Precomputo $O(N)$ 11
6.6 Elementos de Geometría 11
6.6.1 punto.py 11
6.6.2 poligono_convexo.py 11
6.7 Capsula Convexa 11
6.7.1 capsula_convexa.py 11
6.8 Teoría de juegos 12
6.8.1 MEX.py 12
6.9 Identities 12
6.10 Rodrigues Rotation Formula 12
6.11 FFT y NTT 12
6.11.1 FFT.py 12
6.11.2 NTT.py 12

7 Strings 13
7.1 Bordes 13
7.1.1 bordes.py 13
7.2 Función Z 13
7.2.1 funcion_z.py 13
7.3 Manacher (Palindromos) 13
7.3.1 manacher.py 13
7.4 Trie 13
7.4.1 trie.py 13

8 Other 14

9 Tablas y Cotas 14
9.1 Divisores 14
9.2 Factoriales 14

10 Consejos 14
10.1 Debugging 14
10.2 Hitos de prueba 14

University: **Universidad Nacional de La Matanza, DIIT** Team: -ejemplo- **(UNLaM)** Page 1 of 14

Setup

Comando de Ejecución

1.1.1 run.sh

```
cp $1.py $1.print; for x in $1*.in; do echo ARCHIVO: $x; cat $x; echo
    ===; python3 $1.py<$x; echo ===; done | tee -a $1.print

# Uso: ./run.sh nombre_programa
# Notar que no ponemos nombre_programa.py, sino solo nombre programa
# Importante: Los casos de prueba deben estar en el mismo directorio
    que el programa
# Los archivos de entrada deben tener la extensión .in
# Ej: ./run.sh A para ejecutar el programa A.py con los casos de
    prueba A1.in, A2.in, etc.
```

Basico

En esta sección irán los códigos básicos, vistos en la categoría Generales del árbol de correlatividades.

Busqueda Binaria

2.1.1 lower_bound.py

```python
# Devuelve el índice del primer elemento mayor o igual a x
# en un arreglo ordenado
def lower_bound(V, x):
    l, r = -1, len(V)
    while l < r: # V[l] < x <= V[r]
        m = (l + r) // 2
        if V[m] < x:
            l = m
        else:
            r = m
    return r
```

2.1.2 upper_bound.py

```python
# Devuelve el índice del primer elemento mayor a x
# en un arreglo ordenado
def upper_bound(V, x):
    l, r = -1, len(V)
    while l < r: # V[l] <= x < V[r]
        m = (l + r) // 2
        if V[m] <= x:
            l = m
        else:
            r = m
    return r
```

Tabla Aditiva

2.2.1 tabla_aditiva.py

```python
def crear(V):
    n = len(V)
    A = [0] * (n + 1)
    for i in range(n):
        A[i + 1] = A[i] + V[i]
    return A #A[i] = sum(V[:i])

def consulta(A, l, r):
    return A[r] - A[l] #sum(V[l:r])
```

2.2.2 tabla_aditiva_2D.py

```python
# Permite crear y consultar una tabla aditiva para matrices 2D en O(n
    * m) y O(1) respectivamente.
def crear(M): #M: matriz, O(n * m)
    n, m = len(M), len(M[0])
    A = [[0] * (m + 1) for _ in range(n + 1)]
    for i in range(n):
        for j in range(m):
            A[i + 1][j + 1] = A[i + 1][j] + A[i][j + 1] - A[i][j] + M[
                i][j]
    return A #A[i][j] = sum(M[:i][:j])

def consulta(A, l1, r1, l2, r2): #O(1)
    return A[r1][r2] - A[l1][r2] - A[r1][l2] + A[l1][l2] #sum(M[l1:r1)
        [:l2:r2))
```

Programación Dinámica

2.3.1 sub_set_sum.py

```python
# Solución al Problema Sub Set Sum
# Problema: Dados:
#   * Un conjunto de enteros positivos C = {c1, c2, ..., ck}
#   * Un valor V,
# Determinar si es posible sumar exactamente V usando elementos de C.
def sub_set_sum(C, V): #O(n * V)
    n = len(C)
    A = [False] * (V + 1)
    A[0] = True
    for i in range(n):
        for j in range(V, C[i] - 1, -1):
            A[j] |= A[j - C[i]]
    return A
#A[i] = True si es posible sumar exactamente i usando elementos de C
```

2.3.2 cambio_monedas.py

```python
# Solución al problema Cambio de Monedas con DP
# Problema: Dados:
#   * un conjunto de monedas C = {c1, c2, ..., ck}
#   * Un valor V,
# Determinar el mínimo número de monedas de C necesarias para sumar V.
def cambio_monedas(C, V): #O(n * V)
    n = len(C)
    A = [0] + [float('inf')] * V
    for i in range(1, V + 1):
        for j in range(n):
            if i >= C[j]:
                A[i] = min(A[i], A[i - C[j]] + 1)
    return A
# A[i] = mínimo número de monedas de C necesarias para sumar i
```

Recurrencias Lineales

2.4.1 recurrena_lineal.py

```python
# Problema: Dada una recurrencia lineal de la forma
# A[i] = c1 * A[i - 1] + c2 * A[i - 2] + ... + ck * A[i - k]
# con A[0], A[1], ..., A[k - 1] dados, determinar A[n] para n >= k.
# ej: Fibonacci(n) = recurrencia([0,1],[1,1],n)
# IMPORTANTE: no olvidar el modulo

def recurrencia(A, C, n, mod = int(1e9+7)): # O(n * k)
    k = len(C)
    if n < k:
        return A[n]
    A = A + [0] * (n - k + 1)
    for i in range(k, n + 1):
        A[i] = sum(C[j] * A[i - j] for j in range(k)) % mod
    return A[n]

# No lo vimos en clase, pero existe una solución más eficiente en
# O(k^2 * log(n)) usando exponenciación binaria de polinomios.
def recurrencia(A, C, n, mod = int(1e9+7)): # O(k^2 * log(n))
    k = len(C)
    if n < k:
        return A[n]
    A = A + [0] * (n - k + 1)
    def mult(A, B): # Producto de polinomios
        n = len(A)
        C = [0] * n
        for i in range(n):
            for j in range(n):
                C[i] += A[j] * B[i - j]
                C[i] %= mod
        return C
    def exp(A, n): # Potencia rápida de polinomios
        if n == 1:
            return A
        if n % 2 == 0:
            return exp(mult(A, A), n // 2)
        return mult(A, exp(A, n - 1))
    C = [0] * (k * k)
    for i in range(k):
        C[i * k + i] = 1
    C = exp(C, n - k)
    for i in range(k):
        A[n] += C[i] * A[k - i]
        A[n] %= mod
    return A[n]
```

University: Universidad Nacional de La Matanza, DIIT Team: -ejemplo- (UNLaM) Page 2 of 14

Heap y Heapsort

2.5.1 heapsort.py

```
# heap: Estructura de datos que permite mantener un conjunto de
# elementos ordenados y permite insertar y extraer el mínimo
# en O(log n)
# heappush(h, x): Inserta x en el heap h
# heappop(h): Extrae el mínimo del heap h
# h[0] es el mínimo del heap h
# heapsort: Ordena un iterable en O(n log n)
from heapq import heappush, heappop
def heapsort(iterable):
    h = []
    for value in iterable:
        heappush(h, value)
    return [heappop(h) for i in range(len(h))]
```

2.5.2 max_heap.py

```
from heapq import heappush, heappop
def push_inv(h, x):
    heappush(h, -x)

def pop_inv(h):
    return -heappop(h)

def get_inv(h):
    return -h[0]
```

Grafos

Leer grafos

3.1.1 leer.py

```
# Notar que el codigo no cambia si el grafo es ponderado o no
def leer_lista_aristas(m):
    return [
        list(map(lambda x : int(x)-1, input().split()))
        for _ in range(m)
    ]
# ady[u] son los nodos a los que llegan aristas desde u
def leer_lista_adyacencia(n,m):
    # reutilizo código
    aristas = leer_lista_aristas(m)
    ady = [[] for _ in range(n)]
    for arista in aristas:
        u = arista[0]
        v = arista[1]

        # Para grafo ponderado
        ady[u].append([v]+arista[1:])
        ady[v].append([v]+arista[1:]) # no dirigido

        # Para grafo no ponderado
        ady[u].append(v)
        ady[v].append(u) # no dirigido

    return ady

# inc[u] son las aristas incidentes al nodo u
def leer_lista_incidencia(n,m):
    aristas = leer_lista_aristas(m)
    inc = [[] for _ in range(n)]
    for i,arista in enumerate(aristas):
        u = arista[0]
        v = arista[1]
        inc[u].append(i)
        inc[v].append(i) # no dirigido
    return inc, aristas
```

BFS: Busqueda en Anchura

3.2.1 bfs.py

```
# Recorrido de BFS de un grafo
# Recibe la lista de adyacencia y un nodo de origen
# Devuelve la distancia del origen a cada nodo
# inf para nodos inalcanzables
def BFS(inicio : int, ady:list[list[int]])->list[int]:
  N = len(ady)
  dist = [float('inf')]*N
  dist[inicio] = 0
  bolsa, it = [inicio], 0
  while it < len(bolsa):
    nodo = bolsa[it]
    for vecino in ady[nodo]:
      if dist[vecino]>dist[nodo]+1:
        dist[vecino] = dist[nodo]+1
        bolsa.append(vecino)
    it = it+1
  return dist
```

Bipartir un grafo

3.3.1 bipartir.py

```
# Decide si un grafo puede ser bipartito
# Es decir, asignar a cada nodo uno de dos colores
# de tal forma que no haya dos nodos vecinos del mismo color
# Si se puede, retorna True y la lista de colores
# Si no, retorna False y una lista vacía

def Bipartir(ady:list[list[int]])->tuple[bool,list[int]]:
  N = len(ady)
  color = [-1]*N
  for inicio in range(0,N):
    if color[inicio] != -1: continue
    color[inicio] = 0
    bolsa, it = [inicio], 0
    while it < len(bolsa):
      nodo = bolsa[it]
      for vecino in ady[nodo]:
        if color[vecino]==-1:
          color[vecino] = 1-color[nodo]
          bolsa.append(vecino)
        elif color[vecino]==color[nodo]:
          return (False,[])
      it = it+1
  return (True,color)
```

3.3.2 bipartir_2.py

```
# Dado un grafo con aristas con etiquetas 0 y 1
# * Las etiquetas 0 indican que ambos nodos deben tener el mismo color
# * Las etiquetas 1 indican que ambos nodos deben tener colores
    distintos
# Decide si es posible colorear el grafo con dos colores
# de tal forma que se cumplan todas las etiquetas
# Si se puede, retorna True y la lista de colores
# Si no, retorna False y una lista vacía

def Bipartir2(ady: list[tuple[int,int]]) -> tuple[bool, list[int]] :
  # arista es (vecino, peso)
  N = len(ady)
  color = [-1]*N
  for inicio in range(0,N):
    if color[inicio] != -1: continue
    color[inicio] = 0
    bolsa, it = [inicio], 0
    while it < len(bolsa):
      nodo = bolsa[it]
      for vecino, peso in ady[nodo]:
        if color[vecino]==-1:
          color[vecino] = peso ^ color[nodo]
          bolsa.append(vecino)
        elif color[vecino] == color[nodo] ^ peso:
          return (False, [])

      it = it+1
  return (True,color)
```

Camino Mínimo

3.4.1 dijkstra.py

```
import heapq

# Implementación O(M * log N) de Dijkstra con heap
# Es en casi todo caso lo recomendable
# Recibe un nodo de origen y una lista de adyacencia
# Devuelve la distancia mínima de origen a cada nodo
# float('inf') si no es alcanzable
# Funciona tanto para ponderado como para no ponderado
# Recordar que Dijkstra no soporta pesos negativos

def DijkstraHeap(origen : int, G : list[list[tuple[int,int]]]):
  distancias = [float('inf')] * len(G)
  distancias[origen] = 0
  procesados = [False] * len(G)
```

University: **Universidad Nacional de La Matanza, DIIT** Team: -ejemplo- **(UNLaM)** Page 3 of 14

```python
  heap = []
  heapq.heappush(heap, (0, origen))
  while heap:
    dist, nodo = heapq.heappop(heap)
    if procesados[nodo]:
      continue
    procesados[nodo] = True
    for (vecino, distancia) in G[nodo]:
      if distancias[vecino] > distancias[nodo] + distancia:
        distancias[vecino] = distancias[nodo] + distancia
        heapq.heappush(heap, (distancias[vecino], vecino))

  return distancias

# Implementación O(N^2) de Dijkstra
# Solo recomendable en grafos densos donde M ~ N^2
# Recibe y devuelve o mismo que la implementación anterior.
def DijkstraCuadratico(origen : int, G : list[list[tuple[int,int]]]):
  distancias = [float('inf')] * len(G)
  distancias[origen] = 0
  procesados = [False] * len(G)
  for _ in range(len(G)):
    siguiente = -1
    for i in range(len(G)):
      if not procesados[i] and (siguiente == -1 or distancias[i] <
          distancias[siguiente]):
        siguiente = i
    if siguiente == -1:
      break
    procesados[siguiente] = True
    for (vecino, distancia) in G[siguiente]:
      if not procesados[vecino] and distancias[vecino] > distancias[
          siguiente] + distancia:
        distancias[vecino] = distancias[siguiente] + distancia

  return distancias
```

3.4.2 floyd_warshall.py

```python
# Calcula la distancia mínima de cada nodo a cada nodo
# Soporta pesos negativos
# Retorna una matriz de distancias
# O(N^3)
def FloydWarshall(G : list[list[tuple[int,int]]]):
  distancias = [[float('inf')] * len(G) for _ in range(len(G))]
  for u in range(len(G)):
    distancias[u][u] = 0
    for v, w in G[u]:
      distancias[u][v] = w

  for k in range(len(G)):
    for i in range(len(G)):
      for j in range(len(G)):
        distancias[i][j] = min(distancias[i][j], distancias[i][k] +
             distancias[k][j])

  return distancias
```

3.4.3 bellman_ford.py

```python
# Recibe un nodo de origen, una lista de adyacencia y una longitud L
# Calcula para cada nodo y longitud la distancia mínima del origen a
# ese nodo con exactamente esa cantidad de aristas.
# Soporta pesos negativos.
# O((N+M) * L) tiempo, O(N*L) memoria

def BellmanFord(origen : int, G : list[list[tuple[int,int]]], L : int)
    -> list[list[int]]:
  distancias = [ [float('inf')] * len(G) for _ in range(L+1) ]
  distancias[0][origen] = 0
  for l in range(L):
    for u in range(len(G)):
      for v, w in G[u]:
        distancias[l+1][v] = min(distancias[l+1][v], distancias[l][u]
             + w)
  return distancias

# Similar a la anterior pero retorna para cada nodo
# la minima distancia del origen.
# Garantiza que probo al menos todos los caminos de L aristas o menos.
# O((N+M) * L) tiempo pero O(N) memoria
def BellmanFordLigero(origen : int, G : list[list[tuple[int,int]]], L
    : int) -> list[int]:
  distancias = [float('inf')] * len(G)
  distancias[origen] = 0
  for l in range(L):
    for u in range(len(G)):
      for v, w in G[u]:
        distancias[v] = min(distancias[v], distancias[u] + w)
  return distancias
```

3.4.4 spfa.py

```python
# Modificación de BellmanFord
# Calcula la distancia desde el origen a todos los demás nodos
# Soporta pesos negativos
# En el caso promedio: O(N + M)
# En el peor caso: O(N * M)

def SPFA(origen : int, G : list[list[tuple[int,int]]]) -> list[int]:
  distancias = [float('inf')] * len(G)
  distancias[origen] = 0
  cola = [origen]
  i = 0
  en_cola = [False] * len(G)
  while i < len(cola):
    u = cola[i]
    en_cola[u] = False
    for v, w in G[u]:
      if distancias[v] > distancias[u] + w:
        distancias[v] = distancias[u] + w
        if not en_cola[v]:
          cola.append(v)
          en_cola[v] = True
    i += 1
  return distancias
```

Union Find

3.5.1 Small To Large

```python
# Implementa union find utilizando la técnica de
# small to large
# Notar que n se debe definir antes en el código

id = [i for i in range(n)]
# Inicialmente cada nodo esta en su propia componente
cmp = [[i] for i in range(n)]

# Retorna True si se unieron los nodos,
# False si ya estaban en la misma componente
def union(u, v):
    u, v = id[u], id[v]
    if u == v: return False # No se los unio
    if len(cmp[u]) < len(cmp[v]): u, v = v, u
    for x in cmp[v]:
        cmp[u].append(x)
        id[x] = u
    return True
```

3.5.2 Path Compression y Union by Size

```python
# Implementa union find con las optimizaciones
# de path compression y union by size comentadas
# en la clase
# Notar que n se debe definir antes en el código

pad = [i for i in range(n)]
# Inicialmente cada nodo es su propio padre
sz = [1] * n
# tamaño de las componentes

def find(u):
    visto = []
    while u != pad[u]:
        visto.append(u)
        u = pad[u]
    for x in visto:
        pad[x] = u
    return u

# Retorna True si se unieron los nodos,
# False si ya estaban en la misma componente
def union(u, v):
    u, v = find(u), find(v)
    if u == v: return False
    if sz[u] < sz[v]: u, v = v, u
    pad[v] = u
    sz[u] += sz[v]
    return True
```

MST: Árbol Generador Mínimo

3.6.1 kruskal.py

University: **Universidad Nacional de La Matanza, DIIT** Team: -ejemplo- **(UNLaM)** Page 4 of 14

```python
# Dada una lista de aristas, calcula el MST
# MST: Árbol Generador Mínimo
# Notar que es necesario implementar también un union find
# O(M log M)
# Devuelve el costo y la lista de aristas del MST

def Kruskal(g : list[tuple[int,int,int]]):
    # -> tuple[int, list[tuple[int,int,int]]]:
    g.sort(key=lambda x: x[2])
    global n, id, cmp
    n = max([a[0] for a in g] + [a[1] for a in g]) + 1
    id = [i for i in range(n)]
    cmp = [[i] for i in range(n)]
    cost = 0
    mst = []
    for a in g:
        if union(a[0], a[1]):
        # usemos el union-find que nos guste
            cost += a[2]
            mst.append(a)
    return cost, mst
```

3.6.2 prim.py

```python
import heapq
# Dada una lista de aristas, calcula el MST
# MST: Árbol Generador Mínimo
# O(M log M)
# Devuelve el costo y la lista de aristas del MST

def Prim(g : list[tuple[int,int,int]], start : int = 0) :
  # -> tuple[int, list[tuple[int,int,int]]]:
  heap = [(0,-1,start)]
  costo = 0
  mst = []
  n = max([a[0] for a in g] + [a[1] for a in g]) + 1
  adj = [[] for _ in range(n)]
  for a in g:
    adj[a[0]].append((a[1],a[2]))
    adj[a[1]].append((a[0],a[2]))
  used = [False] * n
  while heap:
    w,u,v = heapq.heappop(heap)
    if used[v]: continue
    used[v] = True
    if u != -1:
      costo += w
      mst.append((u,v,w))
    for x in adj[v]:
      if not used[x[0]]:
        heapq.heappush(heap,(x[1],v,x[0]))
  return costo, mst
```

Componentes Fuertemente Conexas

3.7.1 kosaraju_iterativo.py

```python
# Recibe la lista de adyacencia de un grafo dirigido
# Devuelve una lista con el id de la componente
# fuertemente conexa a la que pertenece cada nodo
# O(N+M) tiempo

def Kosaraju(g : list[list[int]]) -> list[int] :
    n = len(g)
    ord = []

    # Ordeno usando simil BFS
    d_in = [0] * n
    for u in range(n):
        for v in g[u]:
            d_in[v] += 1

    visitados = [False] * n
    arista = [0] * n

    # Hago un pseudo-toposort con DFS iterativo
    def dfs(ini):
        pila = [ini]
        while pila:
            u = pila.pop()
            visitados[u] = True

            while arista[u] < len(g[u]):
                v = g[u][arista[u]]
                arista[u] += 1
                if not visitados[v]:
                    pila.append(u)
                    pila.append(v)
                    break
            if arista[u] == len(g[u]):
                ord.append(u)

    for u in range(n):
        if not visitados[u]:
            dfs(u)
    # Transpongo el grafo
    gt = [[] for _ in range(n)]
    for u in range(n):
        for v in g[u]:
            gt[v].append(u)

    # En el transpuesto recorro según el orden inverso de salida de DFS
    cmp = [-1] * n
    cmp_id = 0

    def marcar_componente(u : int):
        pila = [u]
        while pila:
            u = pila.pop()
            if cmp[u] != -1: continue
            cmp[u] = cmp_id
            for v in gt[u]:
                if cmp[v] == -1:
                    pila.append(v)

    # Recorro el grafo
    for u in reversed(ord):
        if cmp[u] == -1:
            marcar_componente(u)
            cmp_id += 1

    return cmp
```

3.7.2 tarjan_iterativo.py

```python
def Tarjan(g : list[list[int]]) -> list[int] :
    n = len(g)
    cmp = [-1] * n
    cmp_id = 0
    tiempo = 0

    entrada = [-1] * n
    min_entrada = [-1] * n
    arista = [0]*n

    def dfs(u):
        nonlocal cmp_id
        nonlocal tiempo

        pila = [u]
        pila_cmp = []
        while pila:
            u = pila[-1]
            pila.pop()
            if entrada[u] == -1:
                entrada[u] = tiempo
                min_entrada[u] = tiempo
                tiempo += 1
                pila_cmp.append(u)

            while arista[u] < len(g[u]):
                v = g[u][arista[u]]
                if entrada[v] == -1:
                    pila.append(u)
                    pila.append(v)
                    break
                elif entrada[v] > entrada[u]:
                    min_entrada[u] = min(min_entrada[u], min_entrada[v])
                elif cmp[v] == -1:
                    min_entrada[u] = min(min_entrada[u], entrada[v])
                arista[u] += 1
            if arista[u] == len(g[u]) and entrada[u] == min_entrada[u]:
                while True:
                    v = pila_cmp.pop()
                    cmp[v] = cmp_id
                    if v == u: break
                cmp_id += 1

    for u in range(n):
        if cmp[u] == -1:
            dfs(u)
    return cmp
```

University: **Universidad Nacional de La Matanza, DIIT** Team: -ejemplo- **(UNLaM)** Page 5 of 14

3.7.3 grafo_condensado.py

```
# Dado un grafo dirigido g, retorna el grafo condensado
# de g y la componente fuertemente conexa de cada nodo
# Recordar: El grafo condensado de G es aquel en el que
# cada componente fuertemente conexa de G es un nodo
# y hay una arista de un nodo U a otro V si en G hay
# una arista de un nodo u en U a un nodo v en V
# Requiere Tarjan o Kosaraju ya implementado
# O(N+M) tiempo

def Condensado(g : list[list[int]]) -> list[list[int]]:
  cmp = Tarjan(g) # Puede ser Kosaraju
  n_cmp = max(cmp)+1
  gc = [[] for _ in range(n_cmp)]
  for u in range(len(g)):
    for v in g[u]:
      if cmp[u] != cmp[v]:
        gc[cmp[u]].append(cmp[v])
  for u in range(n_cmp):
    gc[u] = list(set(gc[u]))
  return (gc, cmp)
```

3.7.4 2_SAT.py

```
# Problema de 2-Satisfactibilidad

# Dada una fórmula en forma normal conjuntiva (CNF)
# con 2 variables por cláusula,
# determinar si existe una asignación de valores a
# las variables que haga verdadera
# a la fórmula.
# La fórmula se representa como una lista de cláusulas,
# donde cada cláusula es una
# tupla de dos elementos. Si el primer elemento de la
# tupla es positivo, se afirma la variable correspondiente.
# Si el segundo elemento de la tupla es positivo, se
# afirma la variable correspondiente. Si el primer elemento
# de la tupla es negativo, se niega la variable correspondiente.
# Si el segundo elemento de la tupla es negativo, se niega la
# variable correspondiente.

# La función retorna una lista de booleanos, donde el i-ésimo
# booleano indica si la variable i debe ser verdadera o falsa.
# Si no existe una  asignación que haga verdadera a la fórmula,
# retorna una lista vacía.
# La función tiene complejidad O(N+M), donde
# N es el número de variables y
# M es el número de cláusulas.
# Ejemplo de uso:
# f = [(1,2),(-1,-2),(1,-2),(-1,2)]
# print(SAT2(2,f)) # [True, True]
# print(SAT2(2,[(1,2),(1,-2),(-1,2),(-1,-2)])) # []
# Necesita tener implementado Condensado y Toposort

def SAT2(n : int, f : list[tuple[int,int]]) -> list[bool]:
  # Formato input: >0 afirmo variable, <0 niego variable
  g = [[] for _ in range(2*n)]

  def neg(x):
    return x+n if x<n else x-n

  # Construyo el grafo de implicancias que modela el problema
  for (p1, p2) in f:
    x1 = p1 - 1 if p1>0 else neg(-p1-1)
    x2 = p2 - 1 if p2>0 else neg(-p2-1)
    g[neg(x1)].append(x2)
    g[neg(x2)].append(x1)

  # Calculo el grafo condensado
  (gc, cmp) = Condensado(g)
  componentes = [[] for _ in range(len(gc))]
  for u in range(2*n):
    componentes[cmp[u]].append(u)

  # Reviso que no haya contradicción
  for i in range(n):
    if cmp[i]==cmp[i+n]:
      return []

  # Asigno valores a las variables
  res = [-1] * n

  orden = Toposort(gc)

  for U in reversed(orden):
    for u in componentes[U]:
      x = u if u<n else neg(u)
      if res[x]==-1:
        res[x] = u<n
  return res
```

Components Biconexas, Puentes y Puntos de Articulación

3.8.1 componentes_biconexas.py

```
# Indentifica los puentes, puntos de articulación y componentes
# biconexas de un grafo no dirigido, recibiendo la lista de incidencia
# g y la lista de aristas ars (cada arista es una tupla de dos nodos)
# Puente: Arista que si elimina aumentan lac antidad de componentes
# conexas del grafo
# Punto de articulación: Nodo que si se elimina aumenta la cantidad
# de componentes conexas del grafo
# Componente biconexa: Subgrafo conexo que no tiene puntos de
# articulación.
# Notar que la división en componentes biconexas es una partición
# de las aristas del grafo (cada arista pertenece a una unica
# componente biconexa) pero no de los nodos, los puntos de
# articulación pertenecen a más de una componente biconexa
# O(N+M) tiempo

def Biconexas(g : list[list[int]], ars : list[tuple[int,int]]):
  # -> tuple[list[int], list[bool], list[bool]] :
  # Primero: Componente biconexa de cada arista
  # Segundo: Para cada nodo, si es punto de articulación
  # Tercero: Para cada arista, si es puente
  n = len(g)
  m = len(ars)

  cmp = [-1] * m
  punto = [0] * n
  puente = [0] * m
  padre = [-1] * n

  llegada = [-1] * n
  min_alcanza = [-1] * n
  tiempo = 0
  pila = []
  indice = [0] * n
  componente = 0

  def DFS(u):
    nonlocal tiempo, componente
    pila_dfs = [u]
    while len(pila_dfs) > 0:
        u = pila_dfs.pop()

        if llegada[u] == -1:
            llegada[u] = tiempo
            min_alcanza[u] = tiempo
            tiempo += 1

        ar = g[u][indice[u]]
        v = ars[ar][0] + ars[ar][1] - u
        if ar != padre[u]:

            if llegada[v] == -1:
                padre[v] = ar
                pila_dfs.append(u)
                pila_dfs.append(v)
                pila.append(ar)
                continue

            if padre[v] == ar:
                if min_alcanza[v] > llegada[u]: puente[ar] = True
                if min_alcanza[v] >= llegada[u]:
                    punto[u] += 1
                    last = pila.pop()
                    while last != ar:
                        cmp[last] = componente
                        last = pila.pop()
                    cmp[ar] = componente
                    componente += 1
                min_alcanza[u] = min(min_alcanza[u], min_alcanza[v])
            elif llegada[v] < llegada[u]:
                pila.append(ar)
                min_alcanza[u] = min(min_alcanza[u], llegada[v])

        indice[u] += 1
        if indice[u] < len(g[u]):
            pila_dfs.append(u)
        continue

  for i in range(n):
    if padre[i] == -1:
      punto[i] -= 1
```

University: **Universidad Nacional de La Matanza, DIIT** Team: –ejemplo– **(UNLaM)** Page 6 of 14

```
    DFS(i)
  punto = [punto[i] > 0 for i in range(n)]
  return cmp, punto, puente
```

LCA: Ancestro Común Menor

3.9.1 binary_lifting_funcional.py

```
# Dado un grafo funcional
# Permite calcular consultas de
# realizar k pasos desde un nodo u
# en O(logN). Notar que si en un árbol
# cada nodo apunta a su padre tenemos un grafo
# funcional.
# O(NlogN) en la inicialización
# O(logN) por consulta

class BinaryLifting:
    def __init__(self, f : list[int]):
        self.n = len(f)
        self.l = self.n.bit_length()
        self.f = [[-1] * self.n for _ in range(self.l)]
        self.f[0] = f
        for i in range(1, self.l):
            for u in range(self.n):
                self.f[i][u] = \
                self.f[i - 1][self.f[i - 1][u]]
    # Obtiene f^k(u)
    def ksig(self, u : int, k : int) -> int:
        for i in range(self.l):
            if k & (1 << i):
                u = self.f[i][u]
        return u
```

3.9.2 binary_lifting_lca.py

```
# Estructura de datos que almacena el Binary
# Lifting de un árbol

class BinaryLifting:

    def __init__(self, g : list[list[int]], \
                 raiz : int = 0, l : int = 0):
        self.n = len(g)
        self.l = max(l,self.n.bit_length())
        self.padres = [[-1] * self.n for _ in range(self.l)]
        self.depth = [0] * self.n
        self.padres[0][raiz] = raiz
        pila = [raiz]
        while pila:
            u = pila[-1]
            pila.pop()
            for v in g[u]:
                if self.padres[0][v] == -1:
                    self.padres[0][v] = u
                    self.depth[v] = self.depth[u] + 1
                    pila.append(v)
        for i in range(1, self.l):
            for u in range(self.n):
                self.padres[i][u] =\
                self.padres[i - 1][self.padres[i - 1][u]]

    def kancestro(self, u : int, k : int) -> int:
        for i in range(self.l):
            if k & (1 << i):
                u = self.padres[i][u]
        return u

    def lca(self, u : int, v : int) -> int:
        if self.depth[u] > self.depth[v]:
            u, v = v, u
        v = self.kancestro(v, self.depth[v] - self.depth[u])
        if u == v:
            return u
        for i in range(self.l - 1, -1, -1):
            if self.padres[i][u] != self.padres[i][v]:
                u = self.padres[i][u]
                v = self.padres[i][v]
        return self.padres[0][u]

    def add_hijo(self, p : int) -> None:
        v = len(self.padres)
        self.padres.append([p] + [-1] * (self.l - 1))
        self.depth.append(self.depth[p] + 1)
        for i in range(1, self.l):
            self.padres[i][v] = \
                self.padres[i - 1][self.padres[i - 1][v]]
```

3.9.3 lca_sparse_table.py

```
# Computa para cada nodo de un árbol
# el tiempo de entrada y la profundidad
# y construye un vector a tal que
# min(a[in[u]:in[v]+1]) es el par ordenado
# (in[c],c), donde c es el LCA de u y v

def generar(g : list[list[int]], raiz : int = 0) \
    -> tuple[list[int], list[int], list[int]]:
    n = len(g)
    in_order = [-1]*n
    depth = [0]*n
    a_vec = []
    arista = [0] * n
    pila = [raiz]
    while pila:
        u = pila[-1]
        pila.pop()
        if in_order[u] == -1:
            in_order[u] = len(a_vec)
        a_vec.append((in_order[u],u))
        if arista[u] < len(g[u]):
            v = g[u][arista[u]]
            arista[u] += 1
            if in_order[v] == -1:
                depth[v] = depth[u] + 1
                pila.append(u)
                pila.append(v)
    return in_order, depth, a_vec

# Usa Sparse Table
class LCA_ST:
    # O(NlogN)
    def __init__(self, g : list[list[int]], raiz : int = 0):
        self.n = len(g)
        self.in_order, self.depth, self.a_vec = generar(g, raiz)
        self.st = st_build(self.a_vec)

    def lca(self, u : int, v : int) -> int:
        l, r = self.in_order[u], self.in_order[v]
        if l > r:
            l, r = r, l
        return st_query(self.st, l, r)[1]
```

Estructuras de Datos

Árbol de Segmentos

4.1.1 segment_tree.py

```
# Árbol de Segmentos
# Se inicializa con un vector V de n valores
# Permite aplicar una operacion op() asociativa a
# un rango [l,r] de V.
# Se deben definir:
# * La operación op(a, b)
# * El valor neutro de la operación
# Complejidad (llamados a op):
# * Construcción: O(n)
# * Consulta: O(log(n))
# * Actualización: O(log(n))

class SegmentTree:
    # Ejemplo de posible operacion
    def Op(self, a, b):
        return a + b
    # Ejemplo del neutro de la operación
    neutro = 0

    def __init__(self, V):
        # La función __init__ nos permite crear un nuevo elemento de
            la clase
        n = len(V)
        self.largo = 1
        # El largo que será representado por el árbol de segmentos
        while self.largo < n:
            self.largo *= 2
            # Tiene que ser mayor o igual a n
        self.st = [neutro for i in range(2 * self.largo)]
        # Crea el árbol inicialmente con el neutro
        for i in range(n):
```

University: **Universidad Nacional de La Matanza, DIIT** Team: –ejemplo– **(UNLaM)** Page 7 of 14

```python
        self.st[self.largo + i] = V[i] # Inicializa las hojas del vector
    for i in range(self.largo - 1, 0, -1):
        self.st[i] = Op(self.st[i * 2], self.st[i * 2 + 1])
        # Inicializa los nodos internos del vector

def Consulta(self, l, r):
    # Consultas iterativas para mejor performance
    # Puede ser la diferencia entre AC y TLE
    l += self.largo
    r += self.largo
    lres = self.neutro
    rres = self.neutro
    while l <= r:
        if l % 2 == 1:
            lres = self.Op(lres, self.st[l])
            l += 1
        if r % 2 == 0:
            rres = self.Op(self.st[r], rres)
            r -= 1
        l //= 2
        r //= 2
    return self.Op(lres, rres)

def Actualizar(self, i, v):
    i += self.largo
    # La posición i en el vector es i + largo en el árbol
    self.st[i] = v # Actualiza el valor en la posición i
    i //= 2 # Accedo al padre de i
    while i >= 1:
        # print(f"Actualizo el nodo {i} accediendo a sus hijos {i*2} e {i*2+1}")
        self.st[i] = Op(self.st[i * 2], self.st[i * 2 + 1])
        #Actualizo el valor del nodo
        i //= 2 # Accedo al padre de i
```

4.1.2 segment_tree_lazy_creation.py

```python
# Árbol de Segmentos
# Se inicializa con un entero n que índica el tamaño del
# dominio. Inicialmente todos los valores son el neutro
# de la operación
# Permite trabajar con un dominio arbitrariamente grande
# Permite aplicar una operacion op() asociativa a
# un rango [l,r] de V.
# Se deben definir:
# * La operación op(a, b)
# * El valor neutro de la operación
# Complejidad (llamados a op):
# * Construcción: O(n)
# * Consulta: O(log(n))
# * Actualización: O(log(n))

class SegmentTreeLazy:
    # Ejemplo de posible operacion
    def Op(self, a, b):
        return a + b
    # Ejemplo del neutro de la operación
    neutro = 0

    def __init__(self, n):
        # La función __init__ nos permite crear un nuevo elemento de la clase
        self.largo = 1
        while self.largo < n:
            self.largo *= 2
        self.st = dict()

    def Consulta(self, lq, rq, nodo = 1, l = 0, r = - 1):
        if r == -1: r = self.largo-1
        # Si r no fue dado, se asume que es el largo del árbol - 1
        if l > rq or r < lq or nodo not in self.st:
            # Si el intervalo [l, r] está completamente fuera de [lq, rq]
            return self.neutro
        if lq <= l and r <= rq:
            # Si el intervalo [l, r] está completamente dentro de [lq, rq]
            return self.st[nodo]
        m = (l + r) // 2
        # Si el intervalo [l, r] está parcialmente dentro de [lq, rq]
        return self.Op(self.Consulta(lq, rq, nodo * 2, l, m), self.Consulta(lq, rq, nodo * 2 + 1, m+1, r))

    def Actualizar(self, i, v):
        i += self.largo # La posición i en el vector es i + largo en el árbol
        self.st[i] = v # Actualiza el valor en la posición i
        i //= 2 # Accedo al padre de i
        while i >= 1:
            self.st[i] = self.Op(self.st.get(i*2,self.neutro), self.st.get(i * 2 + 1,self.neutro))
            #Actualizo el valor del nodo
            i //= 2
            # Accedo al padre de i
```

Sparse Table

4.2.1 sparse_table.py

```python
# La Tabla Sparsa es una estructura de datos que permite una
# inicialización O(NlogN) y consultas en rango:
# * Si la operación es idempotente las operaciones son O(1)
# * Si la operación no es idempotente las operaciones son O(logN)
# (Idempotente: f(a,a)=a, ej mínimo, máximo, and, or)

def operation(a, b):
    return min(a,b)

def next_p2(n: int) -> int:
    return 1 << (n - 1).bit_length()

# Función que recibe un vector y construye su Sparse Table
# O(NlogN)
def st_build(v : list[any]) -> list[list[any]]:
    n = len(v)
    k = n.bit_length()
    st = [[0] * k for _ in range(n)]
    for i in range(n):
        st[i][0] = v[i]
    for j in range(1, k):
        for i in range(n - (1 << j) + 1):
            st[i][j] = operation(st[i][j - 1], st[i + (1 << (j - 1))][j - 1])
    return st

# O(1): Usar si la operación es idempotente (ej: mínimo, máximo, and, or)
def st_query(st : list[list[any]], l : int, r : int) -> any:
    j = r - l
    k = j.bit_length() - 1
    return operation(st[l][k], st[r - (1 << k)][k])

# O(log(n)): Usar si la operación no es idempotente (ej: suma, producto)
def st_query(st : list[list[any]], l : int, r : int) -> any:
    res = None
    for k in range(len(st[0]) - 1, -1, -1):
        if l + (1 << k) <= r:
            if res == None: res = st[l][k]
            else: operation(st[l][k], st_query(st, l + (1 << k), r))
            l += 1 << k
    return res
```

Algoritmos

Divide and Conqueer D&C

5.1.1 merge_sort.py

```python
# Ejemplo de problema resuelto con D&C
# Ordena un vector en O(nlogn)
# Realiza log(n) capas de recursión
def MergeSort(V : list[any]) -> list[any] :
  if len(V) < 2: return V
  m = len(V) // 2
  L = MergeSort(V[:m])
  R = MergeSort(V[m:])
  i,j = 0,0
  for k in range(len(V)):
    if i >= len(L):
      V[k] = R[j]
      j += 1
    elif j >= len(R):
      V[k] = L[i]
      i += 1
    elif L[i] < R[j]:
      V[k] = L[i]
      i += 1
    else:
      V[k] = R[j]
      j += 1
  return V
```

University: **Universidad Nacional de La Matanza, DIIT** Team: -ejemplo- **(UNLaM)** Page 8 of 14

5.1.2 Teorema Maestro

Para analizar la complejidad de los algoritmos de Divide y Vencerás (D&C), existen tres técnicas que nos pueden ser sumamente útiles:

- **Dividir la recursión en capas**: Por ejemplo, en el primer problema podemos observar que en cada capa de la recursión hay una complejidad $O(n)$ y que existen $\log(n)$ capas, porque en cada una, cada subproblema tiene la mitad del tamaño que en la capa anterior.
- **Teorema Maestro**: Si en cada paso un problema de tamaño n se divide en a subproblemas de tamaño n/b y existe un cómputo adicional $O(f(n))$, entonces:
  - Si $f(n) \in O(n^c)$ con $c < \log_b(a)$, entonces $T(n) \in \Theta(n^{\log_b(a)})$. Ejemplo: $T(n) = 8 \times T(n/2) + n^2$, entonces $T(n) \in O(n^3)$.
  - Si $f(n) \in \Theta(n^{\log_b(a)})$, entonces $T(n) \in \Theta(n^{\log_b(a)} \times \log n)$. Ejemplo: $T(n) = 2 \times T(n/2) + n$ (caso de Merge-Sort), entonces $T(n) \in \Theta(n \times \log n)$.
  - Si $f(n) \in \Omega(n^c)$ con $c > \log_b(a)$ y existe $k < 1$ tal que para n suficientemente grande, $a \times f(n/b) \leq k \times f(n)$, entonces $T(n) \in \Theta(f(n))$. Ejemplo: $T(n) = 2 \times T(n/2) + n^2$, entonces $T(n) \in \Theta(n^2)$.
- **Análisis amortizado**: Como en otros algoritmos, puede haber factores que limiten la cantidad de estados de manera ad-hoc. En el segundo problema de ejemplo, se observa que cada estado elimina un elemento, y cada elemento es eliminado por un único estado. Por lo tanto, hay como máximo n estados distintos.

Técnica de 2 Punteros

5.2.1 dos_punteros.py

```python
# La técnica de los dos punteros se utiliza
# para resolver problemas que trabajan con
# el conjunto de subarreglos de un arreglo que
# cumplen una propiedad X tal que si un subarreglo
# cumple la propiedad X, cualquier subarreglo
# que contenga al subarreglo también cumple la
# propiedad X.

# Ejemplo de problema resuelto con dos punteros
# Dado un arreglo de enteros no negativos V y
# un entero k, determinar la cantidad de
# subarreglos de V que suman al menos k.

def DosPunteros(V : list[int],k : int) -> int:
  res, suma = 0, 0
  L = 0
  for R in range(1,len(V)+1):
    suma += V[R-1]
    while R > L and suma >= k:
      suma -= V[L]
      L += 1
    res += R-L
  return res
```

5.2.2 vectores_paralelos.py

```python
# Solución con 2 punteros al problema
# Dados dos vectores $V_A$ de largo $N$ y $V_B$
# de largo $M$ de números enteros, ambos ordenados
# en orden creciente.
# Se desea saber para cada elemento de $V_A$
# cuantos elementos de $V_B$ hay menores o iguales
# a él*
# O(N+M)

def VectoresParalelos(VA : list[int],VB : list[int]) -> list[int]:
  res = [0] * len(VA)
  for i in range(len(VA)):
    if i : res[i] = res[i-1]
    while res[i] < len(VB) and VB[res[i]] <= VA[i]:
      res[i] += 1
  return res
```

5.2.3 ventana_deslizante.py

```python
# Ejemplo del uso de la técnica de ventana deslizante
# para resolver el problema de:
# Dado un arreglo de enteros V y un entero k
# determinar para cada subarreglo de longitud k
# la cantidad de elementos distintos
# O(N * acceso_diccionario)

def Distintos(V : list[int],k : int) -> list[int]:
  res = [0] * (len(V)-k+1)
  histo = dict()
  cantidad = 0
  for i in range(k):
    cantidad += 1 if V[i] not in histo else 0
    histo[V[i]] = histo.get(V[i],0) + 1
  res[0] = cantidad
  for i in range(1, len(V)-k+1):
    j = i+k
    cantidad -= 1 if histo.get(V[i],0) == 1 else 0
    cantidad += 1 if histo.get(V[j-1],0) == 0 else 0
    histo[V[i]] = histo.get(V[i],0) - 1
    histo[V[j-1]] = histo.get(V[j-1],0) + 1
    res[i] = cantidad
  return res
```

Algoritmo de Mo

5.3.1 mo_plantilla.py

```python
# Plantilla para aplicar el algoritmo de Mo a cualquier problema
# Requisitos
# - Se realizán consultas de forma asincronica
# - No hay actualizaciones
# - La función AgregarElemento debe ser implementada
# - La función EliminarElemento debe ser implementada
# - La variable neutro debe ser definida
# CUIDADO: Si la operación no es conmutativa, deben implementar
# versiones por izquierda y derecha de
# AgregarElemento y EliminarElemento
# para evitar errores
# Complejidad: O((N+Q) * sqrt(N) * O(Agregar/Eliminar Elemento))
# En Python es probable que de TLE, en C++ no debería
# Formato del input: [l,r)

def AgregarElemento(actual, elemento):
    # Recomputa la respuesta al agregar un nuevo elemento
    # ejemplo : return actual + elemento

def EliminarElemento(actual, elemento):
    # Recomputa la respuesta al eliminar un elemento
    # ejemplo : return actual - elemento

def Mo(V : list[int], L : list[int], R:list[int]) -> list[int]:
  N, Q = len(V), len(R)
  queries = [(L[i], R[i], i) for i in range(Q)]
  BASE = int(N**0.5)
  vec_res = [0] * Q
  queries.sort(key=lambda x: (x[0]//BASE, x[1]))
  i, j, res = 0, 0, neutro # Cambiar neutro por el valor neutro de la
        operación
  for l, r, idx in queries:
    while i < l:
        res = EliminarElemento(res, V[i])
        i += 1
    while i > l:
        i -= 1
        res = AgregarElemento(res, V[i])
    while j < r:
        res = AgregarElemento(res, V[j])
        j += 1
```

University: Universidad Nacional de La Matanza, DIIT Team: -ejemplo- (UNLaM) Page 9 of 14

```
    while j > r:
        j -= 1
        res = EliminarElemento(res, V[j])
    vec_res[idx] = res
  return vec_res
```

5.3.2 mo_ejemplo.py

```
# Utilizar el algoritmo de Mo para resolver
# el problema de responder consultas de suma
# en rango sobre un vector V de enteros sin
# actualizaciones
# O((N+Q) * sqrt(N))

def SumaEnRango(V : list[int], L : list[int], R:list[int]) -> list[int
    ]:
  N, Q = len(V), len(R)
  queries = [(L[i], R[i], i) for i in range(Q)]
  BASE = int(N**0.5+1)
  res = [0] * Q
  queries.sort(key=lambda x: (x[0]//BASE, x[1]))
  i, j, suma = 0, 0, 0
  for l, r, idx in queries:
    while i < l:
      suma -= V[i]
      i += 1
    while i > l:
      i -= 1
      suma += V[i]
    while j < r:
      suma += V[j]
      j += 1
    while j > r:
      j -= 1
      suma -= V[j]
    res[idx] = suma
  return res
```

Matematicas

Números Primos

6.1.1 criba_eratostenes.py

```
# Calcula la criba de Eratóstenes hasta N
# Para cada número 0 <= i <= N, criba[i] es True
# si i es primo, False en caso contrario
# O(Nlog(log(N)))

def Eratostenes(N:int) -> list[bool]:
    criba = [False] * 2 + [True] * (N - 1)
    # El 0 y el 1 sabemos que no lo son
    for p in range(2, N + 1):
        # Iteramos los números de 2 a N
        if criba[p]: # Si p es primo
            for i in range(p * p, N + 1, p):
            # Recorremos de a saltos de longitud p
                criba[i] = False
    return criba

# para listar primos
primos = Eratostenes(N)
print(list(filter(lambda x: primos[x], range(N+1))))
```

Pensemos que cada número natural (excluido el 0) es un vector infinito de posiciones naturales (incluido el 0). En este caso, $V(N)[i]$ indica el exponente del i-ésimo primo en la factorización del número N. Llamemos $V(N)$ a la representación vectorial de N. Hacer $A \times B$ como números es sumar sus respectivos vectores. Es decir, $V(A \times B) = V(A) + V(B)$.
Análogamente, se tiene que $V\left(\frac{A}{B}\right) = V(A) - V(B)$.
Lo interesante es notar que si un número divide a otro, entonces tiene un exponente menor o igual en cada factor primo. Es decir,

$$A \mid B \iff V(A) \leq V(B)$$

(tomando $\leq$ posición a posición).
También se puede ver que:

$$V(\mathsf{GCD}(A, B)) = \mathsf{min}(V(A), V(B))$$

donde el mínimo se toma posición a posición, y

$$V(\mathsf{LCM}(A, B)) = \mathsf{max}(V(A), V(B))$$

donde el máximo se toma posición a posición.
Esto nos permite ver por qué GCD (Máximo Común Divisor) y LCM (Mínimo Común Múltiplo) tienen propiedades análogas a las de mínimo y máximo.

Divisores

6.2.1 divisores.py

```
# Calcula los divisores de cada número hasta N
# O(Nlog(N))

def Divisores(N:int) -> list[list[int]]:
    divisores = [ [] for _ in range(N + 1) ]
    for i in range(1, N + 1):
        for j in range(i, N + 1, i):
            divisores[j].append(i)
    return divisores
```

6.2.2 divisores_un_numero.py

```
# Obtiene todos los divisores de un número N
# Complejidad: O(sqrt(N))

def DivisoresInd(N :int) -> list[int]:
    divisores = []
    for i in range(1, N):
        if i * i > N: # Es mejor que buscar calcular la raiz cuadrada
             de antes
            break
        if N % i == 0: # Si i es divisor
            divisores.append(i) # Lo añadimos
            if i != N // i: # Si i no es la raiz cuadrada
                divisores.append(N // i) # Añadimos el otro divisor
    return divisores
```

Divisor Común Mayor

6.3.1 euclides.py

```
# Calcula el Divisor Común Mayor de a y b
# O(log(min(a,b)))
# Notar que es una operación:
# - Asociativa
# - Conmutativa
# - Tiene elemento neutro: 0
# - No tiene inverso
# - Idempotente (gcd(a,a) = a)

def gcd(a : int, b: int) -> int:
    while b != 0:
        a, b = b, a % b
    return a
```

Aritmetica Modular

6.4.1 aritmetica_modular.py

```
# Realizar las operaciones con
# los enteros modulo m

def SumaMod(a, b, m):
    return (a+b)%m

def RestaMod(a, b, m):
    return ((a-b)%m+m)%m

def MultMod(a, b, m):
    return (a*b)%m
```

Combinatoria

6.5.1 combinatoria.py

```
# Factorial: n! = n * (n-1) * (n-2) * ... * 1
# Cantidad de formas de ordenar n elementos
# distintos en una fila
# Necesario definir mod
```

University: Universidad Nacional de La Matanza, DIIT Team: –ejemplo– (UNLaM) Page 10 of 14

```
# Usamos memorización para evitar calculos innecesarios
_factorial = [1]
def Factorial(n : int, mod : int) -> int:
    while len(_factorial) <= n:
        _factorial.append(MultMod(_factorial[-1],len(_factorial),mod))
    return _factorial[n]

# Combinatoria: nCr = n! / (r! * (n-r)!)
# Cantidad de subconjuntos de tamaño k
# de un conjunto de tamaño n
# Necesario definir mod

def Combinatoria(n : int, r : int, mod : int) -> int:
    if r > n: return 0
    return MultMod(
        MultMod(Factorial(n),inv(Factorial(r),mod),mod),
        inv(Factorial(n-r),mod),mod)

# Variaciones con Repetición
# Cantidad de formas de elegir r elementos
# de un conjunto de n elementos con repetición
# Necesario definir mod

def VR(n : int, r : int, mod : int) -> int:
    return PotenciaMod(n, r, mod)

# Variaciones sin repetición
# Cantidad de formas de elegir r elementos
# de un conjunto de n elementos sin repetición
# Necesario definir mod

def V(n : int, r : int, mod : int) -> int:
  return MultMod(Factorial(n),inv(Factorial(n-r),mod),mod)

# Permutaciones con repetición
# Cantidad de formas de ordenar un multiconjunto
# con n1, n2, ..., n_k repeticiones de los elementos
# 1, 2, ..., k

def P(ns : list[int], mod : int) -> int:
    n = sum(ns)
    res = Factorial(n,mod)
    for a in ns:
        res = MultMod(res,inv(Factorial(a,mod),mod),mod)
    return res

# Recordar:
# Si tengo X con OX ordenes validos e Y con OY
# ordenes validos, puedo unirlos y si no hay
# restrcciones entre sus elementos
# (X U Y) tiene C(|X|+|Y|,|X|) * OX * OY ordenes
# validos
```

6.5.2 Precomputo $O(N)$

```
# Dado un N y un modulo mod, computa en O(N)
# los factoriales y sus inversos modulo mod
# hasta N inclusive
# idea: https://codeforces.com/blog/entry/83075
# O(N)
def precomputo(N : int, mod : int):
    fact = [1] * (N+1)
    inv = [1] * (N+1)
    inv_fact = [1] * (N+1)
    for i in range(2,N+1):
        fact[i] = (fact[i-1] * i) % mod
        inv[i] = (mod - (mod // i) * inv[mod % i]) % mod
        inv_fact[i] = (inv_fact[i-1] * inv[i]) % mod
    # fact[i] = factorial de i
    # inv[i] = inverso de i
    # inv_fact[i] = inverso del factorial de i
    return fact, inv, inv_fact
```

Elementos de Geometría

6.6.1 punto.py

```
# En esta archivo están las funciones para trabajar
# con puntos/vectores.
# Un vector en R^n  es una lista de n números reales
# Un punto en R^n es un vector en R^n
# Ej: Un punto en R^2 es una lista de 2 números reales
# Ej: Un punto en R^3 es una lista de 3 números reales

# Calcular la norma (tamaño) de un vector en R^n
def norma2(p : list[float]) -> float:
# Retorna el cuadrado de la norma de un vector
    return sum([x**2 for x in p])

# Recordar que sum nos devuelve la suma de todos los elementos de un
    iterable

def norma(p : list[float]) -> float:
# Retorna la norma de un vector
    return norma2(p)**0.5

# Operar con puntos/vectores

def suma_puntos(p1 : list[float], p2 : list[float]) -> list[float]:
    # Retorna la suma de dos puntos
    return [p1[i] + p2[i] for i in range(len(p1))]

def producto_por_escalar(p : list[float], k : float) -> list[float]:
    # Retorna el producto de un punto por un escalar
    return [k * x for x in p]

def resta_puntos(p1 : list[float], p2 : list[float]) -> list[float]:
    # Retorna la resta de dos puntos
    return [p1[i] - p2[i] for i in range(len(p1))]
# Recordar que range(n) nos devuelve un iterable con los números del 0
    al n-1

# Calcular la distancia entre 2 puntos
def distancia(p1 : list[float], p2 : list[float]) -> float:
    # Retorna la distancia entre dos puntos
    return norma(resta_puntos(p1, p2))

def distancia2(p1 : list[float], p2 : list[float]) -> float:
    # Retorna el cuadrado de la distancia entre dos puntos
    return norma2(resta_puntos(p1, p2))

# Calcular el producto punto entre dos vectores
def producto_punto(p1 : list[float], p2 : list[float]) -> float:
    # Retorna el producto punto entre dos puntos
    return sum([p1[i] * p2[i] for i in range(len(p1))])

# Calcular el producto cruz entre dos vectores en R^2
def producto_cruz(p1,p2):
    return p1[0]*p2[1]-p1[1]*p2[0]

# Calcular el producto cruz entre dos vectores en R^3
def producto_cruz3(p1,p2):
    return [p1[1]*p2[2]-p1[2]*p2[1],
            p1[2]*p2[0]-p1[0]*p2[2],
            p1[0]*p2[1]-p1[1]*p2[0]]
```

6.6.2 poligono_convexo.py

```
# Funciones para trabajar con poligonos convexos

# Dados los puntos de un poligono convexo en orden anti-horario
# retorna el area del poligono. (Si están en sentido
# horario el area es negativa)
def Area_Poligono(puntos): # se asumen ordenados
  p = puntos[0]
  return sum(producto_cruz(resta_punto(puntos[i],p),
        resta_punto(puntos[(i+1)%len(puntos)],p))
        for i in range(len(puntos)))/2

# Calcula si un punto está dentro de un poligono convexo
# Asume que los puntos están ordenados en sentido horario
# o anti-horario
# O(N)
def PuntoEnPoligono(p,puntos):
  for i in range(len(puntos)):
    p_i = puntos[i]
    p_ip1 = puntos[(i+1)%len(puntos)]
    area = producto_cruz(resta_punto(p_i,p),resta_punto(p_ip1,p))
    if area<0:
      return False
  return True
```

Capsula Convexa

6.7.1 capsula_convexa.py

```
# Calcula la Capsula Convexa de un conjunto
# de puntos en el plano
# La capsula convexa es el mínimo poligono
# convexo que contiene todos los puntos
# Es mínima en, al menos, los siguientes sentidos
# - Mínima area
# - Mínimo perimetro
# - Está incluida en cualquier otro poligono
#   convexo que contenga todos los puntos

# Esta es una implementación distinta a la vista en clase
# porque es mucho más eficiente y soporta mejor tener 3 o más
# puntos colineales
```

University: **Universidad Nacional de La Matanza, DIIT** Team: -ejemplo- **(UNLaM)** Page 11 of 14

```python
# Complejidad: O(n log n)
# Necesita tener implementadas las funciones de punto.py

def angulo(a, b, c):
    return (a[0] * (b[1] - c[1]) +
            b[0] * (c[1] - a[1]) +
            c[0] * (a[1] - b[1]))


def CapsulaConvexa(puntos):
    puntos = puntos.copy()
    if len(puntos) <= 3:
        return puntos
    puntos.sort()
    cap = []
    # En la comparativa poner >0 para incluir puntos alineados
    # Poner >=0 para excluir puntos alineados
    for p in puntos:
        while len(cap)>1 and angulo(cap[-2],cap[-1],p)>0:
            cap.pop()
        cap.append(p)
    cap.pop()
    puntos.reverse()
    for p in puntos:
        while len(cap)>1 and angulo(cap[-2],cap[-1],p)>0:
            cap.pop()
        cap.append(p)
    return cap

# usar cap, puntos = CapsulaConvexa(ps) para obtener la capsula
    convexa
# y los puntos ordenados.
```

Teoría de juegos

6.8.1 MEX.py

```python
# Calcular el mínimo entero no negativo excluido
# de un iterable.
# Importante porque el número de Grundy de un estado
# de un juego es el MEX de los Grundy de los estados
# a los que se puede llegar.
# O(N)

def MEX(iterable):
  n = len(iterable)
  esta = [False] * (n+1)
  for i in iterable:
    if i <= n:
      esta[i] = True
  mex = 0
  while mex<n and esta[mex]:
    mex += 1
  return mex

# Versión más corta pero menos performante
# por utilizar un set
def MEX_byCopilot(iterable):
  mex = 0
  conjunto = set(iterable)
  while mex in conjunto:
    mex += 1
  return mex
```

Identities

$C_n = \frac{2(2n-1)}{n+1} C_{n-1}$

$C_n = \frac{1}{n+1}\binom{2n}{n}$

$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$

$F_{2n+1} = F_n^2 + F_{n+1}^2$

$F_{2n} = F_{n+1}^2 - F_{n-1}^2$

$\sum_{i=1}^n F_i = F_{n+2} - 1$

$F_{n+i}F_{n+j} - F_n F_{n+i+j} = (-1)^n F_i F_j$

$\sum_{i=0}^n r^i = \frac{r^{n+1}-1}{r-1}$

$\sum_{i=1}^n i^2 = \frac{n\cdot(n+1)\cdot(2n+1)}{6}$

$\sum_{i=1}^n i^3 = \left(\frac{n\cdot(n+1)}{2}\right)^2$

$\sum_{i=1}^n i^4 = \frac{n\cdot(n+1)\cdot(2n+1)\cdot(3n^2+3n-1)}{12}$

$\sum_{i=1}^n i^5 = \left(\frac{n\cdot(n+1)}{2}\right)^2 \cdot \frac{2n^2+2n-1}{3}$

$\sum_{i=1}^n \binom{n-1}{i-1} = 2^{n-1}$

$\sum_{i=1}^n i\cdot\binom{n-1}{i-1} = n\cdot 2^{n-1}$

(Möbius Inv. Formula) Let

$$g(n) = \sum_{d|n} f(d), \text{ then}$$

$$f(n) = \sum_{d|n} g(d)\mu\left(\frac{n}{d}\right)$$

Rodrigues Rotation Formula

Rodrigues rotation formula (rota **v** alrededor de **z** vector unitario, segun un angulo θ:

$$\mathbf{v}_{\text{rot}} = \mathbf{v}\cos\theta + (\mathbf{z}\times\mathbf{v})\sin\theta + \mathbf{z}(\mathbf{z}\cdot\mathbf{v})(1-\cos\theta)$$

Convoluciones Rápidas

6.11.1 FFT.py

```python
import cmath

# FFT function from previous implementation
def fft(a):
    n = len(a)
    if n <= 1:
        return a

    even = fft(a[0::2])
    odd = fft(a[1::2])

    T = [cmath.exp(-2j * cmath.pi * k / n) * odd[k] for k in range(n
        // 2)]

    return [even[k] + T[k] for k in range(n // 2)] + \
           [even[k] - T[k] for k in range(n // 2)]

def ifft(a):
    # Compute the inverse FFT by taking the FFT of the complex
        conjugate,
    # scaling the result, and taking the complex conjugate again.
    n = len(a)
    a_conj = [x.conjugate() for x in a]
    y = fft(a_conj)
    return [(x.conjugate() / n) for x in y]

def convolve(x, y):
    # Length of the result after convolution
    n = len(x) + len(y) - 1
    # Pad x and y with zeros to length n
    x_padded = x + [0] * (n - len(x))
    y_padded = y + [0] * (n - len(y))

    # Compute the FFT of both sequences
    fft_x = fft(x_padded)
    fft_y = fft(y_padded)

    # Point-wise multiplication of the FFTs
    fft_product = [a * b for a, b in zip(fft_x, fft_y)]

    # Compute the inverse FFT to get the convolution result
    result = ifft(fft_product)

    # Since the output may have small imaginary parts due to numerical
        errors, return the real part
    return [round(r.real) for r in result]
```

6.11.2 NTT.py

```python
def ntt(a, n, p, g):
    # Aplica la Transformada Número Teórico (NTT) a la secuencia a
    result = a[:]
    for length in range(1, n, 2):
        w_n = pow(g, (p - 1) // (2 * length), p)
        w = 1
        for start in range(0, n, 2 * length):
            for i in range(length):
                u = result[start + i]
                v = (result[start + i + length] * w) % p
                result[start + i] = (u + v) % p
                result[start + i + length] = (u - v) % p
            w = (w * w_n) % p
    return result

def intt(a, n, p, g):
    # Aplica la Transformada Número Teórico Inversa (INTT)
    n_inv = pow(n, p - 2, p)  # Inversa de n módulo p
```

University: **Universidad Nacional de La Matanza, DIIT** Team: -ejemplo- **(UNLaM)** Page 12 of 14

```python
    g_inv = pow(g, p - 2, p)  # Inversa de g módulo p

    result = ntt(a, n, p, g_inv)
    return [(x * n_inv) % p for x in result]

def next_power_of_2(x):
    # Calcula la siguiente potencia de 2 mayor o igual a x
    return 1 << (x - 1).bit_length()

def convolve_ntt(a, b, p, g):
    # Realiza la convolución usando NTT sin asumir que el tamaño es
        potencia de 2
    n = len(a) + len(b) - 1
    n_padded = next_power_of_2(n)

    # Rellena las secuencias con ceros hasta la siguiente potencia de
        2
    a_padded = a + [0] * (n_padded - len(a))
    b_padded = b + [0] * (n_padded - len(b))

    # Aplica NTT a ambas secuencias
    ntt_a = ntt(a_padded, n_padded, p, g)
    ntt_b = ntt(b_padded, n_padded, p, g)

    # Multiplicación punto a punto
    ntt_c = [(x * y) % p for x, y in zip(ntt_a, ntt_b)]

    # Aplica la NTT inversa
    result = intt(ntt_c, n_padded, p, g)

    # Trunca al tamaño real del resultado de la convolución
    return result[:n]
# Ejemplo de uso
# a = [1, 2, 3]
# b = [4, 5, 6]
# p = 998244353  # Un número primo
# g = 3          # Una raíz primitiva módulo 998244353

#result = convolve_ntt(a, b, p, g)
```

Strings

Bordes

7.1.1 bordes.py

```python
# Calcula el array de bordes de un string
# Un borde es un substring propio que es
# tanto prefijo como sufijo
# bordes[i] = k => s[:k) es el mayor borde de s[:i)
# Complejidad: O(n)

# Notar que podemos obtener las apariciones de un
# string T en un string S calculando
# bordes(T + "#" + S) y contando las apariciones
# de T en los bordes

def bordes(S : str) -> list[int]:
  bordes = [0] * len(S)
  for i in range(1, len(S)):
    # Invariante: bordes[0:i) ya computados
    j = bordes[i - 1]
    while j > 0 and S[i] != S[j]:
      j = bordes[j - 1]
    if S[i] == S[j]:
      j += 1
    bordes[i] = j
  return [0] + bordes
  # para que coincida con la convención

# bordes("abacaba")
# [0, 0, 1, 0, 1, 2, 3, 0]
```

Función Z

7.2.1 funcion_z.py

```python
# Calcula la función z de un string
# La función z de un string S es un arreglo
# de longitud n tal que z[i] es la longitud
# del string más largo que comienza en S[i]
# que es prefijo de S
# Es decir, el Prefijo Común Mayor entre
# S y S[i:]
# Se puede utilizar para encontrar todas las
# ocurrencias de un string T en S
# Calculando z(T + "#" + S) y buscando los
# valores de z iguales a la longitud de T
# Complejidad: O(n)

def array_z(S : str) -> list[int]:
  l, r, n = 0, 0, len(S)
  z = [0]*n
  # z[i] = max k: s[0,k) == s[i,i+k)
  for i in range(1, n):
    # Invariante: s[0,r-l) == s[l,r)
    if i <= r:
      z[i] = min(r - i + 1, z[i - l])
    while i + z[i] < n and S[z[i]] == S[i + z[i]]:
      z[i] += 1
    if i + z[i] - 1 > r:
      l, r = i, i + z[i] - 1
  z[0] = len(S)
  # Por convención puede ser z[0] = 0
  return z

# array_z("xaxbxxax")
# [8, 0, 1, 0, 1, 3, 0, 1]
```

Manacher (Palindromos)

7.3.1 manacher.py

```python
# Dado un string S, la función Manacher(S) devuelve
# dos listas de enteros de longitud n, donde n es la
# longitud de S. La primera lista es impar y la segunda
# es par. La lista impar[i] es la longitud del palíndromo
# más largo con centro en S[i] y la lista par[i] es la
# longitud del palíndromo más largo con centro en el
# espacio entre S[i-1] y S[i].
#
# Es decir, impar[i] es el máximo k tal que S[i-k:i+k]
# es un palindromo y par[i] es el máximo k tal que
# S[i-k:i+k) es un palindromo.
#
# Recordar que un palindromo es una cadena que se lee
# igual de izquierda a derecha que de derecha a izquierda.
#
# O(n).

def Manacher(S : str) -> tuple[list[int],list[int]]:
  n = len(S)
  par, impar = [0]*n, [0]*n
  l, r = 0, -1
  for i in range(n):
    k = 1 if i>r else min(impar[l+r-i],r-i)
    while i+k<n and i-k>=0 and S[i+k]==S[i-k]:
      k+=1
    k -= 1
    impar[i] = k
    if i+k>r: l, r = i-k, i+k

  l,r = 0, -1
  for i in range(n):
    k = 1 if i>r else min(par[l+r-i+1],r-i+1)+1
    while i+k<=n and i-k>=0 and S[i+k-1]==S[i-k]:
      k+=1
    k -= 1
    par[i] = k
    if i+k-1>r: l, r = i-k, i+k-1
  return impar, par

#Ejemplo
#S = "aabbaacaabbaa"
#impar, par = Manacher(S)
#print(impar)
#[0, 0, 0, 0, 0, 0, 6, 0, 0, 0, 0, 0, 0]
#print(par)
#[0, 1, 0, 3, 0, 1, 0, 0, 1, 0, 3, 0, 1]
```

Trie

7.4.1 trie.py

```python
# Implementación de la estructura Trie
# Un Trie es un árbol donde cada nodo tiene
# un diccionario de caracteres a nodos y un contador
# de cuantas veces se ha pasado por ese nodo
# O(|S|) para todas las operaciones


T = [[0, dict()]] # (acumulador, hijos)
# Puede modificarse para guardar metadata adicional

# Agrega la cadena S al trie T
def Agregar(T : list[tuple[int,dict[str,int]]], S : str) -> int:
```

University: **Universidad Nacional de La Matanza, DIIT** Team: -ejemplo- **(UNLaM)** Page 13 of 14

```python
  nodo = 0
  for c in S:
    if c not in T[nodo][1]:
      T[nodo][1][c] = len(T)
      T.append([0, dict()])
    T[nodo][0] += 1
    nodo = T[nodo][1][c]
  T[nodo][0] += 1
  return nodo
# Borra la cadena S del trie T
def Borrar(T : list[tuple[int,dict[str,int]]], S : str) -> int: # Asume que S está representado en T
  nodo = 0
  for c in S:
    T[nodo][0] -= 1
    nodo = T[nodo][1][c]
  T[nodo][0] -= 1
  return nodo
# Busca la cadena S en el trie T
def Buscar(T : list[tuple[int,dict[str,int]]], S : str) -> int:
  nodo = 0
  for c in S:
    if c not in T[nodo][1]:
      return None
    nodo = T[nodo][1][c]
  return nodo
```

Other

Tablas y Cotas

Primos cercanos a 10^n

9941 9949 9967 9973 10007 10009 10037 10039 10061 10067 10069 10079
99961 99971 99989 99991 100003 100019 100043 100049 100057 100069
999959 999961 999979 999983 1000003 1000033 1000037 1000039
9999943 9999971 9999973 9999991 10000019 10000079 10000103 10000121
99999941 99999959 99999971 99999989 100000007 100000037 100000039 100000049
999999893 999999929 999999937 1000000007 1000000009 1000000021 1000000033

Cantidad de primos menores que 10^n

$\pi(10^1)$ = 4 ; $\pi(10^2)$ = 25 ; $\pi(10^3)$ = 168 ; $\pi(10^4)$ = 1229 ; $\pi(10^5)$ = 9592 ; $\pi(10^6)$ = 78.498 ; $\pi(10^7)$ = 664.579 ; $\pi(10^8)$ = 5.761.455 ; $\pi(10^9)$ = 50.847.534 ; $\pi(10^{10})$ = 455.052,511 ; $\pi(10^{11})$ = 4.118.054.813 ; $\pi(10^{12})$ = 37.607.912.018

Divisores

Cantidad de divisores (σ_0) para algunos $n/\neg\exists n' < n, \sigma_0(n') \geqslant \sigma_0(n)$

$\sigma_0(60)$ = 12 ; $\sigma_0(120)$ = 16 ; $\sigma_0(180)$ = 18 ; $\sigma_0(240)$ = 20 ; $\sigma_0(360)$ = 24 ; $\sigma_0(720)$ = 30 ; $\sigma_0(840)$ = 32 ; $\sigma_0(1260)$ = 36 ; $\sigma_0(1680)$ = 40 ; $\sigma_0(10080)$ = 72 ; $\sigma_0(15120)$ = 80 ; $\sigma_0(50400)$ = 108 ; $\sigma_0(83160)$ = 128 ; $\sigma_0(110880)$ = 144 ; $\sigma_0(498960)$ = 200 ; $\sigma_0(554400)$ = 216 ; $\sigma_0(1081080)$ = 256 ; $\sigma_0(1441440)$ = 288 $\sigma_0(4324320)$ = 384 ; $\sigma_0(8648640)$ = 448

Suma de divisores (σ_1) para algunos $n/\neg\exists n' < n, \sigma_1(n') \geqslant \sigma_1(n)$; $\sigma_1(96)$ = 252 ; $\sigma_1(108)$ = 280 ; $\sigma_1(120)$ = 360 ; $\sigma_1(144)$ = 403 ; $\sigma_1(168)$ = 480 ; $\sigma_1(960)$ = 3048 ; $\sigma_1(1008)$ = 3224 ; $\sigma_1(1080)$ = 3600 ; $\sigma_1(1200)$ = 3844 ; $\sigma_1(4620)$ = 16128 ; $\sigma_1(4680)$ = 16380 ; $\sigma_1(5040)$ = 19344 ; $\sigma_1(5760)$ = 19890 ; $\sigma_1(8820)$ = 31122 ; $\sigma_1(9240)$ = 34560 ; $\sigma_1(10080)$ = 39312 ; $\sigma_1(10920)$ = 40320 ; $\sigma_1(32760)$ = 131040 ; $\sigma_1(35280)$ = 137826 ; $\sigma_1(36960)$ = 145152 ; $\sigma_1(37800)$ = 148800 ; $\sigma_1(60480)$ = 243840 ; $\sigma_1(64680)$ = 246240 ; $\sigma_1(65520)$ = 270816 ; $\sigma_1(70560)$ = 280098 ; $\sigma_1(95760)$ = 386880 ; $\sigma_1(98280)$ = 403200 ; $\sigma_1(100800)$ = 409448 ; $\sigma_1(491400)$ = 2083200 ; $\sigma_1(498960)$ = 2160576 ; $\sigma_1(514080)$ = 2177280 ; $\sigma_1(982800)$ = 4305280 ; $\sigma_1(997920)$ = 4390848 ; $\sigma_1(1048320)$ = 4464096 ; $\sigma_1(4979520)$ = 22189440 ; $\sigma_1(4989600)$ = 22686048 ; $\sigma_1(5045040)$ = 23154768 ; $\sigma_1(9896040)$ = 44323200 ; $\sigma_1(9959040)$ = 44553600 ; $\sigma_1(9979200)$ = 45732192

Factoriales

0! = 1	11! = 39.916.800
1! = 1	12! = 479.001.600 ($\in$ int)
2! = 2	13! = 6.227.020.800
3! = 6	14! = 87.178.291.200
4! = 24	15! = 1.307.674.368.000
5! = 120	16! = 20.922.789.888.000
6! = 720	17! = 355.687.428.096.000
7! = 5.040	18! = 6.402.373.705.728.000
8! = 40.320	19! = 121.645.100.408.832.000
9! = 362.880	20! = 2.432.902.008.176.640.000 $\in$ ll
10! = 3.628.800	21! = 51.090.942.171.709.400.000

max signed tint = 9.223.372.036.854.775.807
max unsigned tint = 18.446.744.073.709.551.615

Consejos

Debugging

- ¿Si n = 0 anda? (similar casos borde tipo n=1, n=2, etc)
- ¿Si hay puntos alineados anda?
- ¿Si es vacío anda?
- ¿Si hay multiejes anda?
- ¿Si no tiene aristas anda?
- ¿Si tiene ciclos anda?
- ¿Si tiene un triángulo anda?
- ¿Los arrays son suficientemente grandes? (siempre denle bastante de más por las dudas, pero tampoco se ceben como para que ya no entre en memoria XD)
- ¿Puede dar integer overflow? (SIEMPRE mirar el integer overflow con MUCHO cuidado)
- ¿Podés dividir por cero en algún caso?
- ¿Estás memorizando la recursión bien?
- ¿El caso base está bien hecho y se llega siempre?
- ¿Están bien puestas las cotas iniciales de la binary / inicialización del acumulador máximo/mínimo?
- ¿Estás inicializando bien antes de cada caso?
- ¿Le copiaste el input dos veces en el archivo de entrada (para ver que de igual y bien las dos veces)? [No aplica cuando viene solo una instancia de input]
- ¿Pasa los ejemplos? [No es joda, Leo se quedo afuera de la mundial por esto]

Hitos de prueba

- **45min** todas las columnas de la tabla llena
- **2h** todos conocen todo
- **3h** reunión estratégica
- **4h** reunión estratégica

University: **Universidad Nacional de La Matanza, DIIT** Team: -ejemplo- **(UNLaM)** Page 14 of 14