

# Teoría de algoritmos I 75.29

~ [Maxo](#)

Bienvenidos a mi guía resuelta jeje, tengan en cuenta que son mis resoluciones y no la verdad absoluta, pero les aseguro que les va a servir para tener una idea para encarar los ejercicios. También el formato de cada ejercicio es exactamente el formato perfecto para los exámenes (de ambos profes podberezski y buchwald). Para mas visiten [mis repos de github!](#)

## Indice

- ★ 1. Greedy
- ★ 2. D&C
- ★ 3. Dinamica
- ★ 4. Fuerza bruta y backtracking
- ★ 5. Redes de flujo
- ★ 6. Complejidades

Gracias especiales a mis ayudadores de cursada [Sev](#), [Matex](#) y [Flor](#)

# Greedy

1. possible solution
2. why is it greedy
3. complexity
4. examples
5. code or pseudocode

## exercices

### 1-

Una ruta tiene un conjunto de bifurcaciones para acceder a diferentes pueblos. El listado (ordenado por nombre del pueblo) contiene el número de kilómetros donde está ubicada cada una. Se desea ubicar la menor cantidad de patrullas policiales (en las bifurcaciones) de tal forma que no haya bifurcaciones con vigilancia a más de 50 km. Proponer un algoritmo que lo resuelva.

Ejemplo (ciudad,Bifurcación): (Castelli, 185), (Gral Guido, 249), (Lezama 156), (Maipu, 270), (Sevigne, 194). Si incluimos un patrullero en la bifurcación de Lezama, cubre además de esta a Castelli y Sevigne. Pero no Gral Guido y Maipú. Se necesitaría en ese caso, ubicar otro. Al agregar otro patrullero en Gral Guido, se cubren todas las ciudades restantes. Con 2 móviles policiales en bifurcaciones se cubren todos los accesos a todas las ciudades con distancia menor a 50km.

## Solution

- Ayuda de [Matex](#)

### Algoritmo:

El algoritmo propuesto es un algoritmo en el que primeramente se reordena la lista de ciudades por distancia en lugar de por nombre y luego se recorren evaluando si se debe poner una estacion de la siguiente manera:

- esta ya cubierta por otra estacion mas atras? si no lo esta entonces tendremos que poner una estacion en la ciudad **mas lejana** dentro del radio de 50km
- si no esta cubierta por otra estacion pero esta cubierta por el radio maximo de otra ciudad entonces no coloca estacion ya que cuando se coloque la estacion en la ciudad mas lejana dentro del radio esta tambien estara cubierta
- si se pasa del rango maximo significa que la ciudad mas lejana dentro del rango es la anterior por lo que se pone una estacion ahí y se vuelve a evaluar donde hay que poner la proxima estacion

- si no esta cubierta por una estacion y es la ultima ciudad se pondra una estacion en si misma

### Porque es greedy:

El algoritmo propuesto es un algoritmo greedy ya que se selecciona la opcion mas optima y conveniente en el momento, sea poner estacion en lo mas lejano posible dentro del radio o no hacer nada porque ya esta abarcado por una estacion.

### Complejidad:

La complejidad temporal de este algoritmo es  $O(n \log n)$  siendo  $n$  el largo de la lista de ciudades, ya que es la complejidad de ordenar las ciudades y es la mas alta.

La complejidad espacial del algoritmo es  $O(n)$  ya que se crea una lista de soluciones donde se guarda las ciudades donde se debe poner estaciones, y en el peor caso donde ponemos estaciones en todas las ciudades pq estan todas muy lejos la lista quedaria del mismo tamaño que la lista de ciudades original " $n$ ".

### Ejemplo:



[ (A, 150), (B, 340), (C, 170), (D, 240), (E, 210) ] original

[ (A, 150), (C, 170), (E, 210), (D, 240), (B, 340) ] sorted by distance +

max\_range = ~~150 240 210 290~~  
last\_station = ~~150 170 240 290~~

sol = [A, B, C]

1- max range is -inf so the new max range will be distance + 50 = 200, keep going  
2- covered by max range so keep going  
3- not covered by max range and its not -inf therefore we put a station at the previous city and save it in the solution list, set max range to -inf and keep going  
4- max range is -inf so the new max range will be distance + 50 = 290, keep going  
5- not covered by max range and its not -inf so put a station at the previous city and save it in the solution list, since its the last city and its not covered by the last station we also put a station here and save it to the solution list

In [ ]: # Code

```
def police_stations(cities: list[tuple[str, int]]):
    sorted_cities = sorted(cities, key=lambda city: city[1])
    solution = []

    max_range = float('-inf')
    last_station = float('-inf')

    for index, city in enumerate(sorted_cities):

        if city[1] < max_range or city[1] < last_station + 50:
            continue
```

```

        if max_range == float('-inf'):
            max_range = city[1] + 50
        elif city[1] > max_range:
            solution.append(sorted_cities[index - 1][0])
            last_station = sorted_cities[index - 1][1]
            max_range = float('-inf')

        if city[1] > last_station + 50 and index == len(sorted_cities) - 1:
            solution.append(city[0])

    return solution

print(police_stations([( 'A', 150), ('B', 340), ('C', 170), ('D', 240), ('E', 210)]))
['C', 'D', 'B']

```

## 4-

Para la realización del próximo congreso de "charlas motivacionales para el joven de hoy" se contrató un hotel que cuenta con  $m$  salas de exposición. Existirán " $n$ " oradores. Cada uno solicitó un tiempo de exposición definido por un horario de ingreso y una duración. Los organizadores quieren asignar las salas con un intervalo entre charla y charla de 15 minutos y desean utilizar la menor cantidad de salas posibles. Presentar un algoritmo greedy que resuelve el problema indicando la cantidad de salas a utilizar y la asignación de las charlas. En caso de sobrepasar el máximo de salas disponibles informar. Analice complejidad y optimalidad

### Solution

#### Algoritmo:

El algoritmo que propongo comienza por ordenar todas las charlas de menor a mayor por tiempo de terminación y luego se iterara sobre esa lista ordenada eligiendo para cada momento en que sala se deberá ubicar esa charla bajo el siguiente criterio:

- Si el tiempo de la ultima charla + 15 minutos de descanso intersecciona con la charla que esta siendo analizada entonces no entra en esta sala y vamos a ver si puede entrar en la siguiente sala
- Si el tiempo de la ultima charla + 15 minutos de descanso no intersecciona con la charla analizada entonces dicha charla entra en esta sala y la guardamos

Entonces el numero de salas que usara este algoritmo es exactamente el numero mas alto de intersecciones seguidas que haya y es el minimo numero de salas posible en todas las condiciones, osea que es optimo. si hay  $n$  charlas y todas interseccionan se necesitara un total de  $n$  salas si o si.

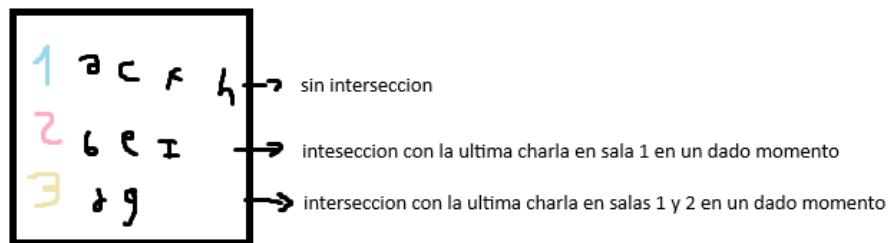
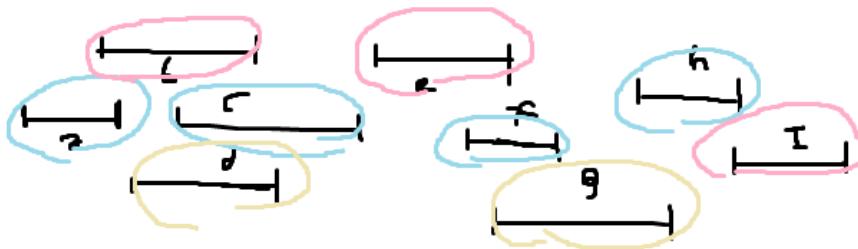
#### Porque es greedy:

Es greedy ya que siempre se toma la charla con tiempo de inicio mas chico que este disponible y en base a checkear intersecciones determinamos en que sala pertenece y la guardamos en la misma.

Complejidad:

La complejidad sera  $O(n\log n + nm)$  siendo  $n$  la cantidad de charlas y  $m$  la cantidad de salas disponibles, el  $n\log n$  es por el ordenamiento inicial y el  $n|m$  es por la logica de seleccion de sala

Ejemplo:



pseudocodigo:

```

def schedule_talks(talks, rooms):

    sorted_talks = sorted(talks, key=lambda talk: talk.time_end) # increasing order O(nLogn)

    rooms_usage = [[] * rooms]
    amount_of_used_rooms = 1

    for talk in sorted_talks: # O(n*m)

        for room in range(rooms):
            if is_intersection(room_usage[room][-1], talk) and room == rooms - 1:
                print("Need more rooms") # INFORMAR
                return 0, None

        if is_intersection(room_usage[room][-1], talk):
            amount_of_used_rooms += 1
            continue
    
```

```
rooms_usage[room].append(talk)
break

return amount_of_used_rooms, rooms_usage
```

## 5-

Una carrera tipo "Ironman" es un triatlón compuesto por 3 instancias: natación (3,86 km de natación), ciclismo (180 km) y carrera a pie (42,2km). Para conocer al ganador se suman los tiempos realizados en cada una de las etapas. Tanto el ciclismo como la carrera a pie se puede realizar en simultáneo con todos los inscriptos. Pero, por una regulación se prohibió que más de 1 persona realice la etapa de nado en el lago en simultáneo. Se conoce el tiempo estimado de cada participante para cada evento. Proponga un orden de salida de tal forma de minimizar el tiempo total de toda la competencia.

### Solution

#### Algoritmo:

Un algoritmo que nos da un orden de salida de los atletas para minimizar la duracion del evento es uno que primeramente ordena los participantes basado en la duracion total estimada de los mismos es decir duracion de natacion + duracion de ciclismo + duracion de correr, en orden decreciente es decir los que mas tardan primero. Esto asegurara que el tiempo sea el minimo como se muestra con inversiones en el ejemplo de abajo.

#### Porque es greedy:

Es greedy ya que vamos a decidir poner primero siempre al mas lento que tengamos

#### Complejidad:

La complejidad temporal es  $O(n\log n)$  por el ordenamiento de participantes

Y la espacial es  $O(1)$  ya que no se crea espacio adicional

#### Ejemplo:

$\left[ (2, 1, 1), (6, 2, 1), (1, 1, 1), (4, 3, 4) \right]$  original  
 $\left[ (6, 2, 1), (4, 3, 4), (2, 1, 1), (1, 1, 1) \right]$  slow swimmers first

if i do fast swimmers first its 16! (+ optimal)  
 but if i do  $\left[ (4, 3, 4), (2, 1, 1), (6, 2, 1), (1, 1, 1) \right]$  its 15 (+ optimal)  
 sorting by total duration dec order  $\left[ (4, 3, 4), (6, 2, 1), (2, 1, 1), (1, 1, 1) \right]$  is also 15 (seems opt)  
 sorting by total duration inc order is 20 (not optimal, worse than before)

winner



A  
C  
D

finish time    B              A C D  
 $(6 + 4 + 2 + 1) + 1 + 1 + 2 = 17$

B starts swimming  
 B finishes swimming and D starts swimming  
 D finishes swimming, A starts swimming and B finished the triathlon  
 A finishes swimming, C starts swimming and D is still cycling  
 C finishes swimming and A and D are running  
 A, C and finally D finish the triathlon  
 FINAL ORDER B, A, C, D and final time 17

In [ ]: # Code

```
def triathlon(competitors: list[tuple[int, int, int]]):
    return sorted(competitors, key=lambda competitor: competitor[0] + competitor[1])

c = [(2, 1, 1), (6, 2, 2), (1, 1, 1), (4, 3, 4)]
print(triathlon(c))
```

$\left[ (4, 3, 4), (6, 2, 2), (2, 1, 1), (1, 1, 1) \right]$

## 6-

Las membresías al club de vino "Varietales de cuyo" son de 4 categorías: "Titanio", "Oro", "Plata" y "Básico". Cada socio tiene una membresía. Por mes envían un pack de degustación que llaman "alfa", "beta", "gamma" y "epsilon". Un socio "Titanio" sólo puede recibir un pack "alfa", Un socio "Oro" puede recibir un pack "alfa" o "beta", un socio "Plata" sólo puede recibir "alfa" o "gamma". Finalmente un "Básico" puede recibir cualquier pack. Diseñar una estrategia greedy para resolver el siguiente problema: Sean Pa, Pb, Pg, Pd los packs disponibles de cada tipo y St, So, Sp, Sb los socios de cada categoría. Informar cómo se puede satisfacer la distribución de packs entre socios (o si no se puede satisfacer).

## Solution

Algoritmo:

El algoritmo propuesto es uno que primeramente asigna todas los packs alpha a los socios de titanio hasta completar las necesarias, luego se procede a los socios de oro y a estos se les da packs beta hasta llenarlos o hasta que se acaben y si se acaban se procedera dandole packs alpha, luego a los socios de plata se les dara primero los packs gamma hasta llenarlos o acabar los packs y si se acaban se dara packs alpha, luego se llega a los socios basicos y se les da los packs que queden hasta completarlos, ya sean alpha, beta, gamma y/o epsilon. Si

en cualquier momento no alcanzan los packs para cubrir un tipo de socio se dara un aviso de que no hay packs suficientes.

### Porque es greedy:

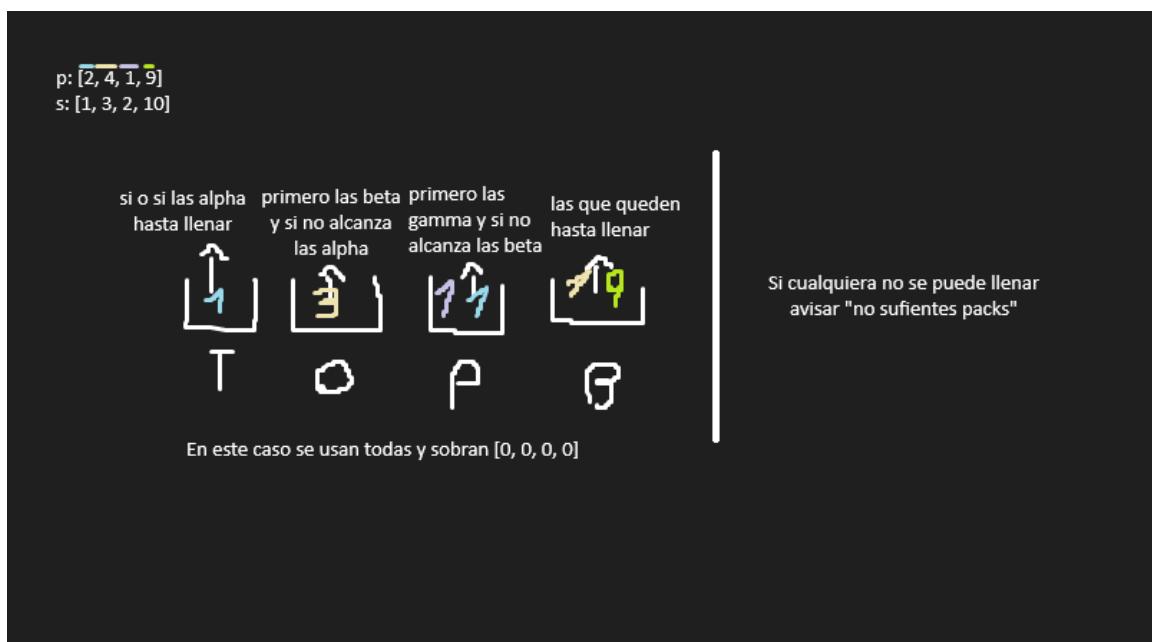
Es greedy porque se le da packs del tipo mas conveniente dependiendo el tipo de socio y que haya disponibilidad en ese instante.

### Complejidad:

La complejidad temporal sera  $O(1)$  ya que el largo de la lista con cantidad de tipos de socios siempre sera 4 y el largo con las cantidades de tipo de packs tambien siempre sera 4.

La complejidad espacial sera  $O(1)$  ya que se la unica memoria adicional que se usara es una lista donde guardar las cantidades de cada pack que deben ser dadas a cada tipo de socio es decir es una lista matriz  $4 \times 4$  y siempre sera del mismo tamaño.

### Ejemplo:



```
In [ ]: # Code

ALPHA = 0
BETA = 1
GAMMA = 2
EPSILON = 3

def distribute_packs(packs: list[int], customers: list[int]):
    solution = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]

    customers_titanium = customers[0]
    customers_gold = customers[1]
    customers_silver = customers[2]
    customers_basic = customers[3]
```

```

# Titanium
if customers_titanium <= packs[ALPHA]:
    while customers_titanium > 0:
        solution[0][ALPHA] += 1
        packs[ALPHA] -= 1
        customers_titanium -= 1
else:
    print('Not enough packs!')
    return

# Gold
while customers_gold > 0 and packs[BETA] > 0:
    solution[1][BETA] += 1
    packs[BETA] -= 1
    customers_gold -= 1
while customers_gold > 0 and packs[ALPHA] > 0:
    solution[1][ALPHA] += 1
    packs[ALPHA] -= 1
    customers_gold -= 1
if customers_gold > 0:
    print('Not enough packs!')
    return

# Silver
while customers_silver > 0 and packs[GAMMA] > 0:
    solution[2][GAMMA] += 1
    packs[GAMMA] -= 1
    customers_silver -= 1
while customers_silver > 0 and packs[ALPHA] > 0:
    solution[2][ALPHA] += 1
    packs[ALPHA] -= 1
    customers_silver -= 1
if customers_silver > 0:
    print('Not enough packs!')
    return

# Basic
inf_loop = False
while customers_basic > 0:
    inf_loop = True
    for i in range(4):
        if packs[i] > 0:
            solution[3][i] += 1
            packs[i] -= 1
            customers_basic -= 1
            inf_loop = False
            break
    if inf_loop:
        print('Not enough packs!')
        return

return solution

print(distribute_packs([2, 4, 1, 9], [1, 3, 2, 10]))

```

```
[[1, 0, 0, 0], [0, 3, 0, 0], [1, 0, 1, 0], [0, 1, 0, 9]]
```

## 7-

El club de amigos de la república Antillense prepara un ágape en sus instalaciones en la que desea invitar a la máxima cantidad de sus "n" socios. Sin embargo por protocolo cada persona invitada debe cumplir un requisito: Sólo puede asistir si conoce a al menos otras 4 personas invitadas. Nos solicita seleccionar el mayor número posible de invitados. Proponga una estrategia greedy óptima para resolver el problema.

### Solution

#### Algoritmo:

El algoritmo que propongo es un algoritmo que primeramente tiene una etapa de setup, crea un diccionario con socio: numero de conocidos, luego relleno este diccionario con la cantidad de gente que conozca cada persona y que tambien este en el diccionario. Luego procedo a ver en el diccionario todos los que tengan menos que 4 conocidos y los pongo en una cola de desinvitados, una vez hecho eso desencolo una persona y disminuyo en 1 la entrada de las personas que este conoció, si alguna de esas personas ahora pasa a tener menos que 4 conocidos lo encolo, despues elimino la persona. Al final las keys del diccionario es la lista de personas a invitar.

#### Porque es greedy:

Es greedy ya que tomo a la primera persona de la cola de desinvitaciones decremento a los conocidos de la persona y si cuando vea que no tienen minimo 4 entonces los encolo para desinvitarlos y no me importa que haya pasado antes o que pase despues.

#### Complejidad:

La complejidad temporal de este algoritmo es  $O(n^2)$  ya que cada persona nueva que vemos le checkeamos los amigos que en el peor caso es todo  $n - 1$  (pq no puede ser su propio amigo)

La complejidad espacial es  $O(n)$  ya que el espacio adicional creado puede ser a lo sumo de tamaño  $n$  (ya sea en el diccionario de posibles invitados y/o en la cola de desinvitaciones)

#### Ejemplo:

```
Partners: { A: [B, D, E, Z], B: [D, E, I, C, X], C: [B, E, C, F, R], D: [B, C, A, I, F], E: [A, B, C, F, I], F: [C, D, E, I], I: [A, B, D, E, F, O] }
```

1 Lets create a dic of possible invitations: amount of known invitees, initializing everyone at 0  
{ A: 0, B: 0, C: 0, D: 0, E: 0, F: 0, I: 0}

2 Fill the dic accordingly  
{ A: 3, B: 4, C: 4, D: 5, E: 5, F: 4, I: 5 } // note that Z, X, R, O weren't counted bc they aren't possible invitations

3 Now check individually if they know at least 4 if not then we put them in a no longer invited queue

4 Now we see all the enqueued people individually and we decrease by 1 everyone who knows them and then delete the entry,  
but if the person we decrease by 1 goes under 4 known invitees then we enqueue them too, and so on

After that process the keys of the used dictionary is the list of people to invite!  
[ B, C, D, E, F, I ]

(not a great example but meh, u get the idea)

CORRECTION i put B in A's list but didn't put A in B's list, so the actual result in this case is [ ], but if I add A to the list then the result would be right

In [ ]: # Code

```
def invites(partners: dict[str, list[str]]):
    from collections import deque
    possible_invites = {}
    uninvites = deque()

    for partner in partners:
        possible_invites[partner] = 0

    for partner in partners: # O(n^2)
        for friend in partners[partner]:
            if friend in possible_invites:
                possible_invites[partner] += 1

    for partner in possible_invites:
        if possible_invites[partner] < 4:
            uninvites.append(partner)

    while uninvites: # worst case O(n^2)
        partner = uninvites.popleft()

        if partner not in possible_invites:
            continue

        for friend in partners[partner]:
            if friend in possible_invites:
                possible_invites[friend] -= 1
                if possible_invites[friend] == 3: # if i == 3 instead of < 4 it goes
                    uninvites.append(friend)
        del possible_invites[partner]

    return possible_invites.keys()

partners = {
    "A": ["B", "D", "E", "Z"],
    "B": ["D", "E", "I", "C", "X"],
    "C": ["B", "E", "C", "F", "R"],
    "D": ["B", "C", "A", "I", "F"],
    "E": ["A", "B", "C", "F", "I"],
    "F": ["C", "D", "E", "I"],
    "I": ["A", "B", "D", "E", "F", "O"]
}
```

```
"B": ["D", "E", "I", "C", "A", "X"],  
"C": ["B", "E", "C", "F", "R"],  
"D": ["B", "C", "A", "I", "F"],  
"E": ["A", "B", "C", "F", "I"],  
"F": ["C", "D", "E", "I"],  
"I": ["A", "B", "D", "E", "F", "O"],  
}  
  
print(invites(partners))
```

```
dict_keys(['B', 'C', 'D', 'E', 'F', 'I'])
```

## 8-

Considere el problema anterior, pero con la adición de una nueva restricción: Los organizadores desean que cada invitado pueda conocer nuevas personas. Por lo que nos solicitan que sólo puede asistir si NO conoce al menos otras 4 personas invitadas. Modifique su propuesta para satisfacer esta nueva solución.

### Solution

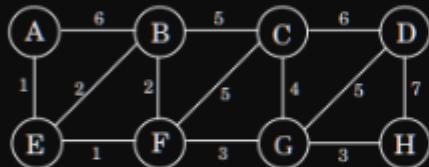
Simplemente la modificación sería que antes de meterlo a la cola por no conocer al menos 4 se checkee si la cantidad de invitados n menos la cantidad de gente que conoce menos si mismo es mayor o igual a 4 en cuyo caso no lo meteríamos a la cola de desinvitaciones.

```
(len(partners) - posibles_invitados[partner] - 1 >= 4)
```

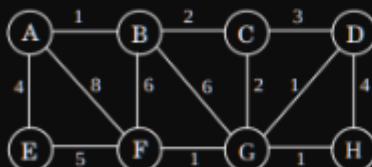
## 9-

Indicar si las siguientes afirmaciones son verdaderas o falsas. Justificar la respuesta.

- a. El algoritmo de Prim genera el mismo resultado que el algoritmo de Kruskal, al calcular el MST de un Grafo.
- b. El algoritmo de Prim aplicado a un Grafo genera siempre el mismo MST
- c. El algoritmo de Kruskal aplicado a un Grafo genera siempre el mismo MST
- d. Si un Grafo posee todas sus aristas distintas, el MST es siempre el mismo sin importar qué algoritmo se utilice
- e. El MST del siguiente Grafo posee 2 MST



- f. El MST del siguiente Grafo es: {AB, AE, BC, CG, GD, GF, GH}



- g. Dado un grafo  $G$  no dirigido con costos en cada arista  $c(e)$ . Si suponemos que  $e^*$  es la arista de menor costo ( $c(e^*) < c(e)$  para cada arista de  $G$  /  $e \neq e^*$ ), entonces podemos afirmar que cualquier MST de  $G$  contendrá a  $e^*$ .

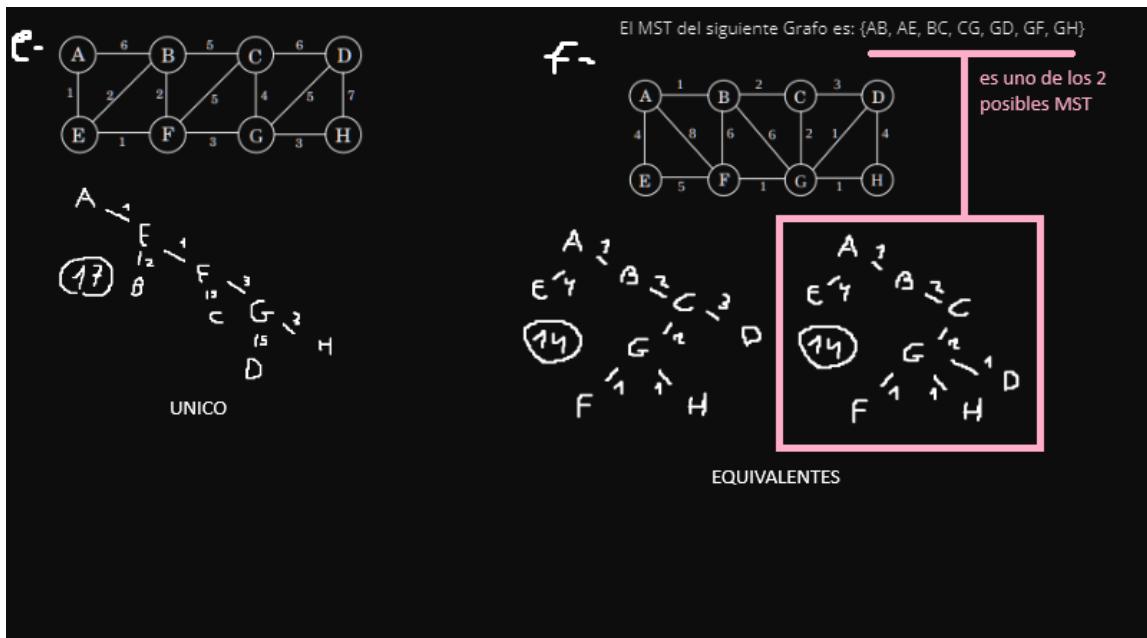
## Solution

a- F, los dos algoritmos generalmente encuentran el mismo MST pero puede pasar que no den el mismo arbol por ejemplo en caso de que haya mas de una arista con el mismo peso y justo cualquiera de las dos sirve para armar un MST

b- F, Prim siempre genera el mismo MST mientras que no haya aristas con mismo peso en ese caso el algoritmo podria llegar a dar diferentes MST

c- F, al igual que en prim krusal siempre da el mismo MST a menos que haya aristas del mismo peso lo que puede llegar a provocar que haya mas de un MST existente

d- V, la afirmacion es correcta



g- F, No necesariamente la arista de menor costo del grafo debe pertenecer al MST

## 10-

Un importante museo nacional tiene "n" piezas de arte en diferentes bóvedas de seguridad (1 pieza por bóveda inicialmente). Se ha programado una exposición especial que requiere que esas n piezas se reúnan para luego ser transportadas al lugar designado. Para la movilización se ha contratado un seguro que cobra un valor cada vez que una pieza corre riesgo de robo (cuando la bóveda que la contiene se abre). Cada pieza fue tasada con un valor determinado. Cuando se realiza un traslado hay 2 bóvedas involucradas. Se abren tanto la bóveda de origen como la de destino. Por lo tanto se debe pagar al seguro los valores de los que están almacenados en ambas bóvedas. En el traslado se pueden seleccionar cualquier cantidad de las piezas que están en las bóvedas involucradas. Deseamos determinar qué trasladados realizar de forma de lograr unificar en una sola bóveda todos las piezas abonando la menor cantidad de plata a la aseguradora. Proponer un algoritmo greedy que solucione el problema (y que sea óptimo).

Ejemplo: Si tengo 4 piezas con sus respectivos valores en 4 bóvedas. A: p1=5 B: p2=4 C: p3=2 D: p4=6 Si comienzo trasladando p1 a la bóveda B. Deberé abrir la bóveda A y B. Se pagará al seguro 5+4=9 por el proceso. Si ahora se desea trasladar p3 a la bóveda B se deberá pagar 2+5+4=11. El proceso continúa hasta que todas las piezas estén unificadas.

### Solution

#### Algoritmo:

El algoritmo optimo propuesto es uno que primeramente empieza haciendo un heapify de la lista de bovedas y le da forma a la misma de un heap de minimos, luego procederemos a desencolar los 2 elementos mas baratos del heap los sumaremos a la cuenta total de gastos

por seguro y los encolamos en el heap como el costo de las dos bovedas juntas, haremos esto hasta que solo quede 1 elemento en el heap que sera la boveda que queda con todos los elementos y ahora tenemos en la variable de gastos por seguro la suma total de los gastos y aseguramos que es la minima posible ya que siempre vamos eligiendo localmente las 2 bovedas mas baratas.

#### Porque es greedy:

Es greedy por lo mismo que se explica arriba el hecho de siempre estar agarrando las 2 bovedas mas baratas disponibles ya que ese seria el optimo local

#### Complejidad:

La complejidad temporal es  $O(n \log n)$ .  $O(n)$  por el heapify +  $O(n \log n)$  el desencolar todos los elementos del heap.

La complejidad espacial es  $O(1)$  ya que el heapify nos permite usar como heap la misma lista que nos pasan por parametro, si no se quiere modificar la misma se puede hacer una copia pero pasaria a ser  $O(n)$ .

#### Ejemplo:

1	3	1	1	1	TOTAL: $2+2+4+7 = \underline{15}$
2	3		1	1	
2	3		2		
4	3				
7					

#### Pseudocodigo:

```
# pa practicar lo hago jej

funcion museo(bovedas):
    heapify de las bovedas
    dinero gastado = 0

    mientras que el largo de bovedas sea > 1:
        boveda1 = desencolar boveda
        boveda2 = desencolar boveda
```

```

dinero gastado += boveda1 + boveda2

encolar (boveda1 + boveda2)

devolver dinero gastado

```

In [ ]: # Code

```

def museum(vaults: list[int]):
    import heapq

    heapq.heapify(vaults)
    min_money_spent = 0

    while len(vaults) > 1:
        vault1 = heapq.heappop(vaults)
        vault2 = heapq.heappop(vaults)
        min_money_spent += vault1 + vault2
        heapq.heappush(vaults, vault1 + vault2)

    return min_money_spent

print('my example:', museum([1, 3, 1, 1, 1]))
print('exercise example:', museum([5, 4, 2, 6]))

```

my example: 15  
 exercise example: 34

## 11-

Un fabricante de perfumes está intentando crear una nueva fragancia. Y desea que la misma sea del menor costo posible. El perfumista le indicó un listado de ingredientes. Por cada uno de ellos determinó una cantidad mínima (puede ser cero) y una máxima que debe contar en la fórmula final. Cada ingrediente tiene asociado un costo por milímetros cúbicos. Sabiendo que en la presentación final es de X milímetros cúbicos. Presentar una solución utilizando metodología greedy que resuelva el problema.

### Solution

#### Algoritmo:

El algoritmo que resuelve esto es simplemente comprar siempre la menor cantidad posible de cada ingrediente, entonces si necesitamos una cantidad no entera de uno compraremos siempre para arriba de tal manera que no falte ingrediente. El costo total se guardara en una variable para ser devuelta.

#### Porque es greedy:

Es greedy porque siempre vamos a comprar la menor cantidad posible de un ingrediente y eso nos lleva al menos costo total posible.

### Complejidad:

La complejidad temporal es O(n) ya que hay que ver todos los ingredientes.

La complejidad espacial es O(1).

### Ejemplo:

ingredients: [(min, max, price per cml)] = [(2, 3, 800), (5, 10, 100), (0.5, 1, 500)]

We will always buy the min amount of each ingredient therefore  
the min cost of the first ingredient is  $2 \times 800$ , for the 2nd ingredient its  $5 \times 100$  and  $\text{ceil}(0.5) \times 500$   
so the total cost in this examples is 2,600

### Pseudocódigo:

```
function perfume_ingredients_min_cost(ingredients_list):  
  
    total cost = 0  
  
    for each ingredient:  
        total cost += ceil(ingredient min amount) * ingredient cost per  
        cml  
  
    return total cost
```

## 12-

El ajedrez se juega con un tablero cuadriculado. La pieza llamada "Rey" puede moverse en cualquiera de los 8 cuadrados aledaños a su posición actual comiendo cualquier otra pieza que esté en ellos. Contamos con un tablero especial de  $n \times m$  cuadrados y una cantidad ilimitada de piezas "Rey". Queremos ubicar la mayor cantidad de reyes sin que estos se puedan comer entre si. Proponer un algoritmo greedy para resolverlo. Brindar complejidad. Justificar la optimalidad de su propuesta.

### Solución

#### Algoritmo:

Primero voy a decir que  $n$  es la cantidad de rows y  $m$  es la cantidad de columns, ahora el algoritmo propuesto es uno que va viendo las rows y dentro de cada una pone un rey en la primera posición que sea posible mientras que no este al lado de otro rey atrás suyo, luego la próxima row será skippeada ya que seguro que no se podrá poner ningún rey ahí porque al poner la máxima cantidad de reyes en la row de arriba esta quedará toda cubierta, se hace esto hasta terminar de analizar todas las rows.

#### Porque es greedy:

Es greedy porque vamos a poner un rey en una posición siempre que podamos, es decir a penas no tenga un rey atrás.

### Complejidad:

La complejidad temporal de este algoritmo es  $O(n*m)$  ya que debemos revisar todo el tablero.

La complejidad temporal es  $O(1)$  ya que no se usara memoria adicional y los reyes se van a poner sobre el tablero existente.

### Ejemplo:

Pongo rey en posicion 1 y  
skipeo una posicion pongo otro  
rey y vuelvo a skippear asi hasta  
terminar una row.

Luego skipeo una row entera  
con la proxima row vuelvo a  
hacer el procedimiento  
anterior

### Pseudocódigo:

```
funcion poner_reyes(tablero):  
  
    cantidad de reyes = 0  
    para cada row_index en range(len(tablero), 2): # de a 2 para skippear  
        una row en el medio  
  
            para cada square_index en range(len(tablero[row_index])):  
                si square_index == 0:  
                    poner rey en tablero[row_index][0]  
                    cantidad de reyes += 1  
                    continue  
  
                si tablero[row_index][square_index-1] no tiene rey:  
                    poner rey en tablero[row_index][square_index]  
                    cantidad de reyes += 1  
  
    return cantidad de reyes
```

18-

Una empresa de telecomunicaciones ganó una licitación para construir un conjunto de " $r$ " líneas de comunicación bidireccional para " $n$ " ciudades. Cada línea de comunicación " $l$ " une

dos ciudades y de generarla le permite obtener una ganancia mensual GI. Puede elegir construir cualquier línea a menos que una nueva línea conecte 2 ciudades previamente conectadas que un camino que pase por 1 o más líneas. Nos solicitan nuestro asesoramiento para determinar qué líneas construir maximizando la ganancia obtenida.

## Solution

### Algoritmo:

Estoy asumiendo que me dan un grafo completo con todas las ciudades ya que si no es completo no habría forma de comparar las ganancias de las líneas.

El algoritmo que propongo es kruskal para buscar el arbol de tendido maximo, esto garantizara que no se creen líneas desde una ciudad a otra si ya habían líneas comunicandolas directa o indirectamente y tambien garantizara que la ganancia sea la maxima posible ya que ordenaran las líneas de mayor a menor por ganancia y se iran seleccionando las mas caras que puedan ser construidas.

### Porque es greedy:

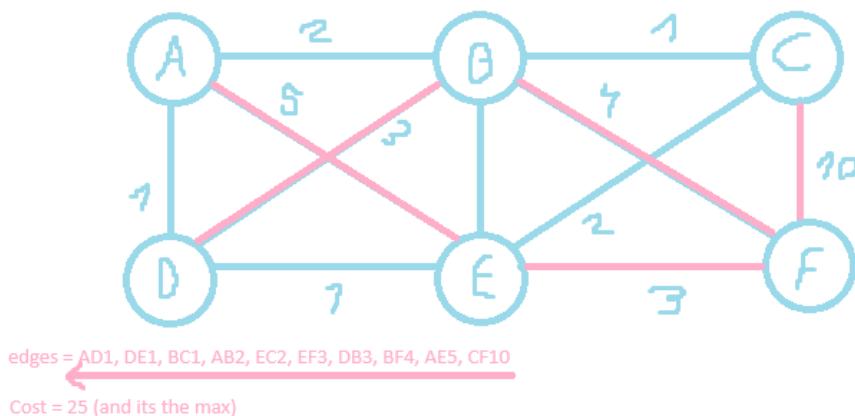
Es greedy ya que siempre vamos a elegir construir las líneas con la mas alta ganancia mientras sea posible.

### Complejidad:

La complejidad temporal del algoritmo sera  $O(n^*logn)$  ya que lo que mas cuesta tiempo en el algoritmo es el ordenado de las aristas, la implementacion de kruskal sera con UnionFind.

La complejidad espacial es  $O(n)$  ya que se creara un arbol con la misma cantidad de nodos n.

### Ejemplo:



### Pseudocódigo:

```

funcion maxima_ganancia(posibles_lineas):

    union_find = crear_union_find(nodos de posibles_lineas)
    arbol_ganancia_maxima = crear_grafo(nodos de posibles_lineas)
    lineas_ordenadas_por_ganancia = ordenar_posibles_lineas de mayor a menor
    por ganancia

        para cada linea en lineas_ordenadas_por_ganancia:
            ciudad1, ciudad2, ganancia = linea # La Linea tiene la informacion
            de quienes une y su ganancia

                si ciudad1 y ciudad2 ya estaban conectadas en union_find:
                    continue

                arbol_ganancia_maxima.agregar_nueva_linea(ciudad1, ciudad2,
                ganancia)
                union_find.unir(ciudad1, ciudad2)

        return arbol_ganancia_maxima

```

## 24-

Contamos con una impresora central en un centro de cómputos del campus universitario. Entre varios departamentos y laboratorios nos solicitan al inicio de cada mes, la impresión de "n" documentos. Cada uno de ellos tiene una duración determinada y cuenta con una fecha de entrega. Si nos pasamos de esta recibimos un apercibimiento proporcional al retraso más largo del mes. Como a la impresora le falta mantenimiento queremos lograr - siempre que sea posible - tiempo entre los trabajos de impresión. Presentar un algoritmo greedy que dada la lista de tareas proponga la fechas de inicio de publicación minimizando el apercibimiento y dando tiempo entre las tareas siempre que sea posible.

### Solution

Algoritmo:

Porque es greedy:

Complejidad:

Ejemplo:

Pseudocodigo:

Skipped 13, 14, 15, 16, 17, 19, 20, 21, 22, 23

# D&C

1. possible solution
2. complexity (master theorem)
3. examples
4. code or pseudocode

exercices

## Teorema maestro

$$**T(n) = AT(n/B) + O(n^C)**$$

$O(n^C)$  \ es el costo de la parte no recursiva\*

Version simplificada:

- $\log_B(A) < C \rightarrow T(n) = O(n^C)$
- $\log_B(A) = C \rightarrow T(n) = O(n^C * \log_B(n))$
- $\log_B(A) > C \rightarrow T(n) = O(n^{\log_B(A)})$

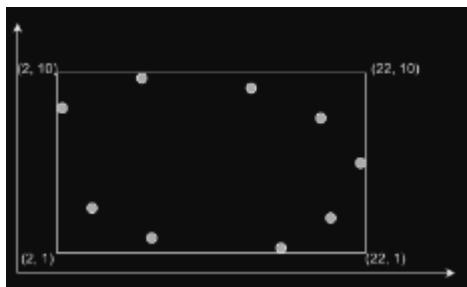
Version general:

- $f(n) = O(n^C), C < \log_B(A) \rightarrow T(n) = \Theta(n^{\log_B(A)})$
- $f(n) = \Theta(n^C * \log^k(n)), C = \log_B(A) \rightarrow T(n) = \Theta(n^C * \log^{k+1}(n))$
- $f(n) = \Omega(n^C), C > \log_B(A) \rightarrow T(n) = \Theta(f(n))$

$\Omega$  (cota inferior),  $O$  (cota superior),  $\Theta$  (cota superior e inferior)

1-

Contamos con una lista ordenada de "n" coordenadas satelitales (latitud-longitud) que conforman un área con forma poligonal convexa. Queremos mostrar ese sector del mapa con el mayor tamaño posible en nuestra pantalla rectangular de la computadora. El programa que muestra el mapa acepta como parámetros 2 coordenadas para construir el rectángulo a mostrar: los correspondientes a los límites inferior izquierdo y superior derecho. Construya un algoritmo eficiente que resuelva el problema con complejidad  $O(\log n)$ . ¿Existe una solución de fuerza bruta que para "n" pequeños sea más eficiente? ¿para qué tamaño de n esto cambia?



## Solution

### Algoritmo:

El algoritmo consiste en crear 3 vectores, un vector A que empieza en el primer punto y va al siguiente, un vector B que empieza en el ultimo punto y va al siguiente y un vector C que corresponde al medio. El algoritmo se centra en poder descartar la mitad del polígono con cada llamada recursiva hasta tener 3 únicamente vectores, cuando eso pase se comparan a mano cual es el maximo con respecto al eje dado y ese seria el caso base. Hay 6 casos totales que pueden pasar con los vectores A, B y C para determinar entre que vectores estara la respuesta, dichos casos se detallan con dibujos abajo. Se hara esto 4 veces, para altura maxima y minima, y para anchura maxima y minima, una vez tengamos esto tendremos la medida de la pantallita.

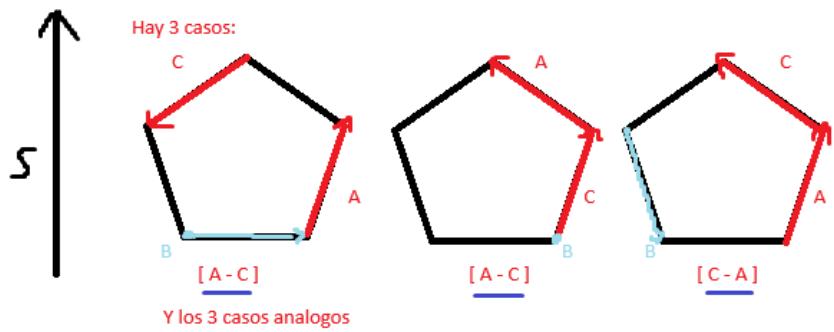
### Complejidad:

$$T(n) = AT(n/B) + O(n^C)$$

- A = 1
- B = 2
- C = 0 (solo comparaciones)

$$\log_2(1) = 0 = C \Rightarrow T(n) = O(n^C * \log_B(n)) = \mathbf{O(\log n)}$$

### Ejemplo:



A es el primer elem  
B es el ultimo elem  
C es el mid

no pseudocode this time cowboy B)

## 2-

Un colaborador del laboratorio de "cálculo automatizado S.A" propone un nuevo método de multiplicación de matrices. Utiliza división y conquista partiendo la matrices en bloques de tamaño  $n/4 \times n/4$ . Y el proceso de combinación de resultados lleva  $\Theta(n^2)$ . Se muestra vago en indicar la cantidad de subproblemas que creará cada paso. Le indican que este dato es fundamental para decidir si proseguir con esa línea de investigación o no. Actualmente utilizan el algoritmo de Strassen con una complejidad de  $O(n \log_2(7))$ . Siendo  $T(n) = aT(n/4) + \Theta(n^2)$ , con  $a$  la información a determinar. ¿Cuál es la cantidad de subproblemas más grande que puede tomar la solución para que sea mejor que su algoritmo actual?

### Solution

```
raw
Complejidad del algoritmo de strassen = O(n^2.8)
```

Sabiendo que la ecuación del teorema maestro es  $T(n) = aT(n/4) + O(n^2)$

```
c = 2
b = 4
a = a
```

```
si log_4(a) < 2 => T(n) = O(n^2) # s1
= 2 => T(n) = O(n^2 * log_4(n)) # no maximiza a
> 2 => T(n) = O(n^log_4(a)) #s2

# s1
```

```

como log_4(a) < 2
    a < 4^2
    a < 16

# s2
si log_4(a) > 2
    a > 4^2
    a > 16
pero log_4(a) < 2.8
    a < 4^2.8
    a < 48.5

```

Entonces el mayor posible a tal que el algoritmo todavía sea mejor que  $O(n^{2.8})$  es 48 ya que llevaría a una complejidad de  $O(n^{2.79})$

### 3-

Se cuenta con un vector de "n" posiciones en el que se encuentran algunos de los primeros "m" números naturales ordenados en forma creciente ( $m \geq n$ ). En el vector no hay números repetidos. Se desea obtener el menor número no incluido. Ejemplo: [1, 2, 3, 4, 5, 8, 9, 11, 12, 13, 14, 20, 22]. Solución: 6. Proponer un algoritmo de tipo división y conquista que resuelva el problema en tiempo inferior a lineal. Expresar su relación de recurrencia y calcular su complejidad temporal.

#### Solution

##### Algoritmo:

El algoritmo propuesto es uno que cuenta con una función recursiva cuyo caso base es que si el largo de la lista pasada por parámetro es de largo 1 entonces ese es el punto crítico donde puede faltar el número buscado atrás o adelante de este, para eso vamos a tener un bool también como parámetro para indicar a qué lado se movió en la recursividad y basandonos en si viene de ir a la izq entonces es +1 y sino -1. Luego se procede a dividir el array a la mitad y elegimos quedarnos con una de las mitades depende de si el elemento del medio está en su correspondiente posición en el array para que los números estén en escalera, osea `arr[0] + medio == arr[medio]` si es verdad vamos a la derecha sino a la izquierda y llamamos recursivamente con esa mitad seleccionada.

##### Complejidad:

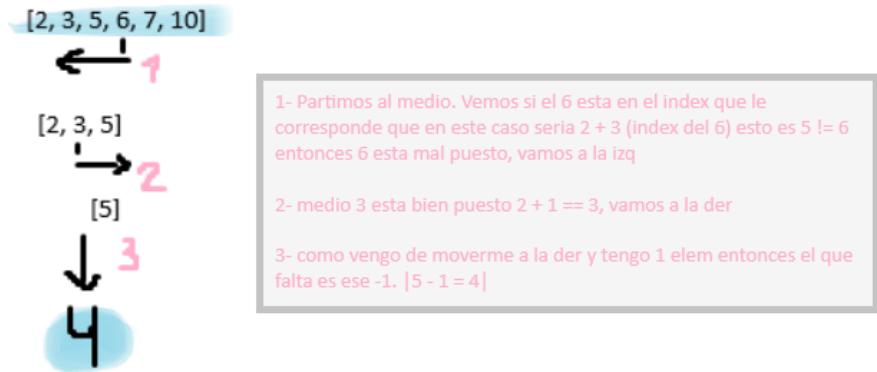
La complejidad temporal la podemos calcular con el teorema maestro con:

$$T(n) = AT(n/B) + O(n^C)$$

- A = 1
- B = 2
- C = 0

$$\log_2(1) = 0 = C \Rightarrow T(n) = O(n^C + \log_B(n)) = O(\log(n))$$

Ejemplo:



```
In [ ]: # Code

def ghost_num(arr):
    if len(arr) == 1:
        return arr[0] + 1
    elif len(arr) == 0:
        return -1

    return _ghost_num(arr, 0, len(arr) - 1, False)

def _ghost_num(arr, start, end, going_left):
    if start == end and going_left:
        return arr[start] + 1
    elif start == end and not going_left:
        return arr[start-1] + 1

    mid = (start + end) // 2

    if arr[mid] == arr[0] + mid:
        return _ghost_num(arr, mid + 1, end, False) # Move right
    else:
        return _ghost_num(arr, start, mid, True) # Move Left

l = [2, 3, 5, 6, 7, 10]
print(ghost_num(l))
```

4

4-

Dado un vector A de "n" números enteros (tanto positivos como negativos) queremos obtener el subvector cuya suma de elementos sea mayor a la suma de cualquier otro

subvector en A. Ejemplo: Array: [-2, -5, 6, -2, -3, 1, 5, -6]. Solución: [6, -2, -3, 1, 5]. Resolver el problema de subarreglo de suma máxima por división y conquista.

## Solution

### Algoritmo:

El algoritmo sera una especie de merge sort pero en lugar de ordenar lo que hara sera checkear si combiene unir los dos subarrays que tiene a la izq y a la der teniendo en cuenta q la union haga que la suma sea mayor a la de solamente quedarse con uno de los subarrays. Primeramente en la funcion recursiva se dara un caso base de que si el array tiene 1 solo elemento lo devuelva, luego se divide el array en 2 y para ambos se llama recursivamente a la funcion de buscar el subarray de suma maxima luego que se tiene los 2 subarrays nos fijamos si la suma de ambas subsumas maximas es mayor a ambas subsumas en cuyo caso debemos crear el nuevo subarray empezando en el inicio de subarray 1 y terminando en el final de subarray 2, luego devuelve la suma del subarray, y los indices de inicio y final del mismo.

### Complejidad:

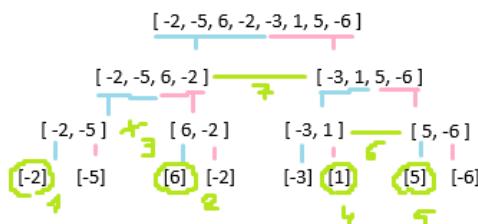
La complejidad temporal sera:

$$T(n) = AT(n/B) + O(n^C)$$

- A = 2
- B = 2
- C = 1 (bc we need to sum all the items in the subarr which in the worst case is the whole arr)

$$\log_2(2) = 1 = C \Rightarrow T(n) = O(n^C * \log_B(n)) = O(n\log n)$$

### Ejemplo:



- 1- Between -2 and -5 -2 is the greatest and -2 -5 is -7 which is not better than -2.
- 2- same as above but with 6 and -2
- 3- now we sum the best on one side and the best on the other side to see if its better, but (-2+6) < 6 so 6 is still the greatest sum starting at index 2 and ending at index 2 (just the 6)
- 4- Between -3 and 1, 1 is greater and (-3+1) < 1 so 1 is the best sum in that step
- 5- same with 5 and -6
- 6- (1+5) > 5 so the new max sum will be from 1 across to 5 which are indexes 5 - 6

7- Now the best on one side is 6 and the best on the other side is 6, their sum is 12 > 6, so we build the final subarray from index 2 (start of the max left arr) to index 6 (end of the max right arr)

FINAL SOL: [ 6, -2, -3, 1, 5 ]

Pseudocódigo:

```
def max_sum_subarray(arr):
    sum, subarr = _max_sum_subarray(arr, 0, len(arr))
    return sum, subarr

def _max_sum_subarray(arr, start, end):
    if start <= end:
        return arr[start]

    mid = (start + end) // 2

    left_max_sum, left_sub_start, left_sub_end = _max_sum_subarray(arr,
start, mid)
    right_max_sum, right_sub_start, right_sub_end = _max_sum_subarray(arr,
mid + 1, end)

    if (left_max_sum + right_max_sum) < left_max_sum:
        return left_max_sum, left_sub_start, left_sub_end # Left is better
    elif (left_max_sum + right_max_sum) < right_max_sum:
        return right_max_sum, right_sub_start, right_sub_end # right is
better

    return sum(arr[left_sub_start:right_sub_end+1]), left_sub_start,
right_sub_end # joining them is better
```

## 5-

Se realiza un torneo con n jugadores en el cual cada jugador juega con todos los otros n-1. El resultado del partido solo permite la victoria o la derrota. Se cuenta con los resultados almacenados en una matriz. Queremos ordenar los jugadores como P1, P2, ..., Pn tal que P1 le gana a P2, P2 le gana a P3, ..., Pn-1 le gana a Pn (La relación "le gana a" no es transitiva). Ejemplo: P1 le gana a P3, P2 le gana a P1 y P3 le gana a P2. Solución: [P1, P3, P2]. Resolver por división y conquista con una complejidad no mayor a O(n log n).

### Solución

#### Algoritmo:

El algoritmo que planteo es un tipo de merge sort que divide el array en 2 y se llama recursivamente para cada parte una vez que tiene las dos partes se hace un merge lineal con el criterio escrito abajo y se retorna. El caso base es cuando queda 1 solo elemento en la recursion y lo devuelve.

Merge: crea un array auxiliar y en este irá poniendo el resultado del merge, el merge continua hasta ver los dos contenidos de los dos arrays izq y der, y se van poniendo en orden todos los jugadores dependiendo quien le gana a quien.

#### Complejidad:

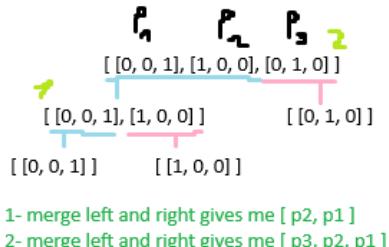
$$T(n) = AT(n/B) + O(n^C)$$

- A = 2
- B = 2
- C = 1 (por el merge)

$$\log_2(2) = 1 = C \Rightarrow T(n) = O(n^C * \log_B(n)) = \mathbf{O(n \log n)}$$

Ejemplo:

	P1	P2	P3
P1	0	0	1
P2	1	0	0
P3	0	1	0



Pseudocódigo:

```

sort(arr, start, end):
    si start >= end:
        devolver arr[start]

    mid = start + end / 2

    izq = sort izq
    der = sort der

    devolver merge(izq, der)

merge(izq, der):
    res = []
    izq_p, der_p = 0

    mientras izq_p < len izq & der_p < len der:
        si izq[izq_p] gana a der[der_p]:
            guardar izq[izq_p] en res
            izq_p += 1

        si der[der_p] gana a izq[izq_p]:
            guardar der[der_p] en res
            der_p += 1
  
```

```
si todavia quedan cosas en izq o der ponerla en res
```

```
devolver res
```

## 6-

A raíz de una nueva regulación industrial un fabricante debe rotular cada lote que produce según un valor numérico que lo caracteriza. Cada lote está conformado por "n" piezas. A cada una de ellas se le realiza una medición de volumen. La regulación considera que el lote es válido si más de la mitad de las piezas tienen el mismo volumen. En ese caso el rótulo deberá ser ese valor. De lo contrario el lote se descarta. Actualmente cuenta con el proceso "A" que consiste en para cada pieza del lote contar cuántas de las restantes tienen el mismo volumen. Si alguna de las piezas corresponde al "elemento mayoritario", lo rotula. De lo contrario lo rechaza. Un consultor informático impulsa una solución (proceso "B") que considera la más eficiente: ordenar las piezas por volumen y con ello luego reducir el tiempo de búsqueda del elemento mayoritario. Nos contratan para construir una solución mejor (proceso "C"). Se pide:

1. Exprese mediante pseudocódigo el proceso "A".
2. Explique si la sugerencia del consultor (proceso "B") realmente puede mejorar el proceso. En caso afirmativo, arme el pseudocódigo que lo ilustre.
3. Proponga el proceso "C" como un algoritmo superador mediante división y conquista. Explíquelo detalladamente y brinde pseudocódigo.

### Solution

#### Proceso A:

```
rotular lote(lote):  
  
    cantidad de elem bien = 0  
    para cada elem en lote:  
        cantidad de elementos con el mismo vol = -1 # para no contarse a  
        si mismo  
  
            para cada otro_elem en lote:  
                si otro_elem y elem tienen mismo vol:  
                    cantidad de elementos con el mismo vol += 1  
  
                si cantidad de elementos con el mismo vol >= largo del lote / 2:  
                    cantidad de elem bien += 1  
  
            si cantidad de elem bien >= largo del lote / 2:  
                rotular  
  
            no rotular
```

#### Proceso B:

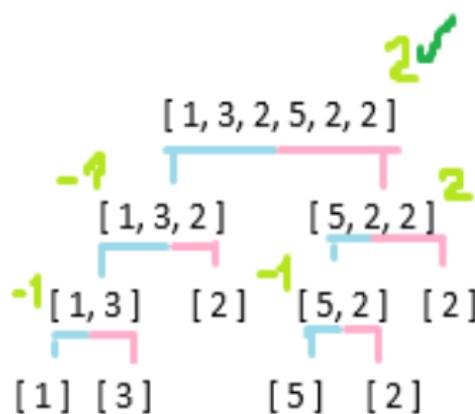
Proceso A es  $O(n^2)$  y el B es  $O(n\log n)$  por el ordenamiento inicial osea que es mas rapido y su pseudocodigo seria:

```
rotular lote(lote):
    lote ordenado = ordenar lote

    si lote ordenado[mitad] == lote ordenado[principio] o lote
    ordenado[mitad] == lote ordenado[final]:
        rotular # ya que la mitad del lote tiene ese volumen
        no rotular
```

### Proceso C:

- Propongo un algoritmo que divida el arr en 2 y llame recursivamente a ambos lados para encontrar el mas repetido en los dos lados, una vez que los tengas si son iguales devuelve ese numero sino cuenta las ocurrencias de los dos en el rango que estamos viendo luego si uno de ellos ocurre la mitad o mas de las veces devolvemos ese numero, si ninguno de los dos cumple eso entonces devuelve -1 que quiere decir que no se rotula. El caso base es cuando se llega a un array de 1 elemento devuelva ese elemento.
- Complejidad:  $T(n) = AT(n/B) + O(n^C)$ 
  - $A = 2$
  - $B = 2$
  - $C = 1$  # por el recuento de repeticiones de un elem en un rango
  - $\log_2(2) = 1 = C \Rightarrow T(n) = O(n^C * \log_B(n)) = O(n\log n)$
- Ejemplo:



```
In [ ]: # Code

def find_majority_in_range(arr, start, end):
    if start == end:
        return arr[start]

    mid = (start + end) // 2

    left_majority = find_majority_in_range(arr, start, mid)
    right_majority = find_majority_in_range(arr, mid + 1, end)

    if left_majority == right_majority:
        return left_majority

    left_count = sum(1 for i in range(start, end + 1) if arr[i] == left_majority)
    right_count = sum(1 for i in range(start, end + 1) if arr[i] == right_majority)

    if left_count >= (end - start + 1) // 2 and left_count > right_count:
        return left_majority

    if right_count >= (end - start + 1) // 2 and right_count > left_count:
        return right_majority

    return -1

arr = [1, 4, 2, 6, 2, 2]
print(find_majority_in_range(arr, 0, len(arr) - 1))
```

2

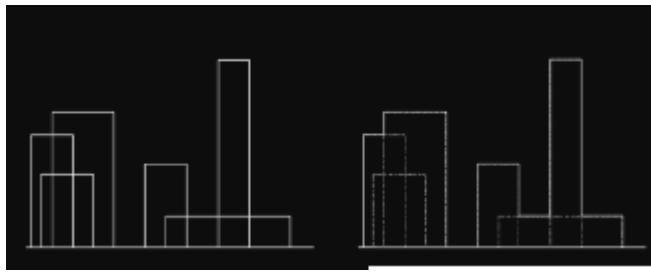
## 7-

Para la elaboración de un juego se desea construir un cielo nocturno de una ciudad donde se vea el contorno de los edificios en el horizonte. Cada edificio "ei" está representado por rectángulos mediante la tripla (izquierda, altura, derecha). Dónde "izquierda" corresponde a la coordenada x menor, "derecha" la coordenada x mayor y altura la coordenada y. Todos los edificios inician en la coordenada 0 de y. Se cuenta con una lista de N edificios que llegan sin un criterio de orden específico. Se desea emitir como resultado el contorno representado como una lista de coordenadas "x" y sus alturas.

Tenga en cuenta el siguiente ejemplo:

- Lista de edificios: (1, 11, 5), (2, 6, 7), (3, 13, 9), (12, 7, 16), (14, 3, 25), (19, 18, 22).
- Contorno: (1, 11), (3, 13), (9, 0), (12, 7), (16, 3), (19, 18), (22, 3), (25, 0).

Presentar un algoritmo utilizando división y conquista que dado el listado de edificios retorna como resultado el contorno de la ciudad.



## Solution

### Algoritmo:

El algoritmo divide en 2 el array y llama recursivamente para las dos mitades hasta que sea de largo 1 y retorne su contorno, luego teniendo el contorno de la derecha y la izq se hace un merge de ambos sin poner los que estan demas en la nueva lista de contornos y luego se devuelve esa lista.

Criterios para el merge:

**Dado un  $c_1 = (x, y)$  y  $c_2 = (u, v)$**

- si  $x < u$ ,  $c_1$  va primero
- si  $x == u \ \& \ y >= v$  mantener  $c_1$  no poner  $c_2$
- si  $y == 0 \ \& \ v == 0$  significa que el elem anterior alarga el contorno entonces si  $x > u$  queda solo  $c_1$  y no  $c_2$

### Complejidad:

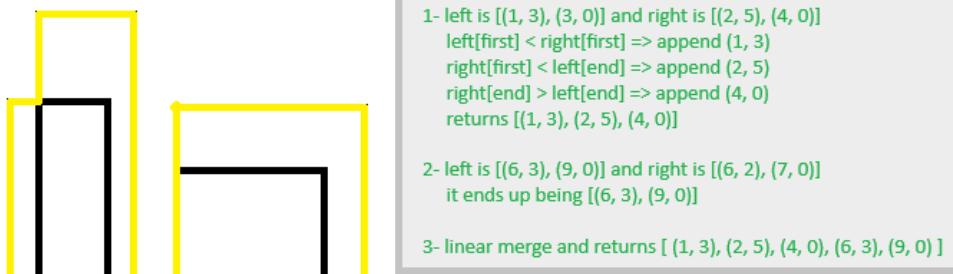
La complejidad del algoritmo es:

$$T(n) = AT(n/B) + O(n^C)$$

- A = 2
- B = 2
- C = 1 (por el merge)

$$\log_2(2) = 1 = C \Rightarrow T(n) = O(n^C * \log_B(n)) = O(n \log n)$$

### Ejemplo:



Pseudocódigo:

```
obtener_contorno(arr, start, end):
    contorno = []

    mitad = (start + end) / 2

    mitad izq = obtener_contorno(arr, start, mitad)
    mitad der = obtener_contorno(arr, mitad end)

    mergear_mitades(mitad izq, mitad der, contorno) # O(n)
    # Mergea con los criterios mencionados arriba

    return contorno
```

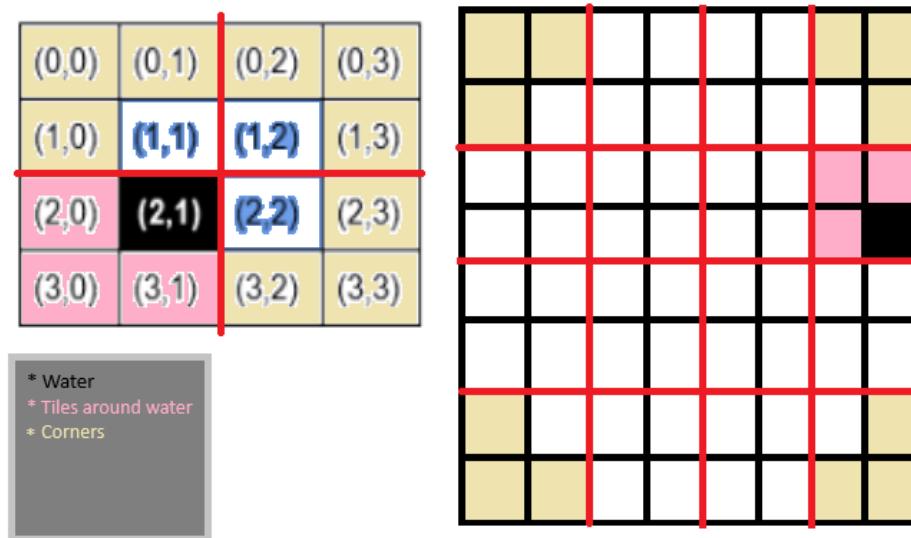
8-

Esta peculiar empresa se dedica a cubrir patios cuadrados de  $n^*n$  metros ("n" es un número entero potencia de 2 y mayor o igual a 2). Cuenta con baldosas especiales que tienen forma en L (como se muestra en celeste en la imagen). Las baldosas no se pueden cortar. Todo patio cuenta con un único sumidero de agua de lluvia. Ocupa 1x1 metro y su ubicación depende del patio (Se muestra en la figura de ejemplo como un cuadrado negro). Nos piden que, dado un patio con un valor "n" y una ubicación del sumidero en una posición x,y desde la punta superior izquierda, determinemos cómo ubicar las baldosas. Presentar un algoritmo que lo resuelva utilizando división y conquista.

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

## Solution

Couldnt think of a solution, ill come back to it... maybe



## 9-

Se realiza un experimento de conductividad de un nuevo material en aleación con otro. Se formaron muestras numeradas de 1 a n. A mayor número, mayor concentración del nuevo material. Además se realizaron "n" mediciones a diferentes temperaturas de conductividad para cada muestra. Los resultados fueron expresados en una matriz M de nxn. Se observa que un mismo material cuanto mayor temperatura tiene mayor conductividad. Además, cuanto mayor concentración a la misma temperatura, también mayor conductividad. En conclusión podemos, al analizar la matriz, ver dos progresiones. Cada fila tiene números ordenados de forma creciente y cada columna tiene números ordenados de forma creciente. Dada la matriz M, los experimentadores quieren encontrar en qué posición se encuentra un determinado número. Proponga una solución utilizando división y conquista.

## Solution

Algoritmo:

El algoritmo que propongo es uno que toma por parametros la matriz, el elemento a buscar, una posicion de inicio y fin, y la posicion de la fila siendo analizada actualmente. Lo que hara este sera para cada fila hacer una busqueda binaria. Habra 2 casos base, uno se fija si inicio < final lo que significa que no encontramos el elemento en la fila y se suma + 1 al parametro de pos de fila actual, y el otro caso base es cuando no encontramos el elemento y estamos en la ultima fila de la matriz en ese caso el elemento no esta en la matriz. Luego busca la mitad y si la mitad es < al elemento nos fijamos hacia la derecha y si es > nos fijamos hacia la izq recursivamente.

**Complejidad:**

La complejidad del algoritmo es  $O(n\log n)$  porque se hacen  $n$  busquedas binarias (en el peor caso) una por cada row y esas busquedas son  $\log n$ . Ta dificil teo maestro aca medio q no c, pero se q esa complejidad deberia estar bien.

**Ejemplo:**

	$C_1$	$C_2$	$C_3$	$C_4$	Find 5	Parameters:
$T_1$	1	2	3	6		curr_row = 0 122
$T_2$	2	3	4	7		start, end, find
$T_3$	3	4	5	8		[ 1, 2, 3, 6 ] Binary search -> nothing curr_row + 1
$T_4$	4	5	6	9		[ 2, 3, 4, 7 ] Binary search -> nothing curr_row + 1
						[ 3, 4, 5, 8 ] Binary search -> pos 2
					ANSWER (2, 2)	

**Pseudocodigo:**

```

buscar_elem(matriz, elem, inicio, fin, fila_actual):

    si inicio >= fin & fila_actual == len(matriz) - 1:
        return no encontrado

    si inicio >= fin:
        fila_actual + 1
        return buscar_elem(matriz, elem, inicio, fin, fila_actual)

    medio = inicio + fin / 2

    si medio > elem:
        return buscar_elem(matriz, elem, inicio, medio, fila_actual) # ir
izq
    
```

```

    si medio < elem:
        return buscar_elem(matriz, elem, medio, fin, fila_actual) # ir der

    return (fila_actual, medio)

```

## 11-

Una agencia gubernamental tiene un conjunto de "n" agentes dispersos por el país. Para una misión urgente requiere utilizar dos de ellos. Cada agente tiene una ubicación (x,y). Se dispone de un helicóptero para buscarlos. Generar una solución por división y conquista que indique cuáles son los 2 agentes más cercanos, cuál es su distancia y dónde debería ir el helicóptero a buscarlo.

### Solution

Algoritmo:

**No me di cuenta y lo hice que devuelva un solo agente en vez de 2 pero no cambia mucho el algoritmo**

El algoritmo que propongo es uno que divide en 2 el array de agentes y llama recursivamente para obtener el mas cercano de la parte izq y de la parte der hasta que se llegue al caso base de que quede solo 1 agente y se devuelve el mismo y su distancia al cuadrado por medio de  $\| \text{agente} - \text{base} \|^2$ , una vez se tiene izq y der se comparan las dist para ver quien es el mas cercano y devuelve el mismo y su dist.

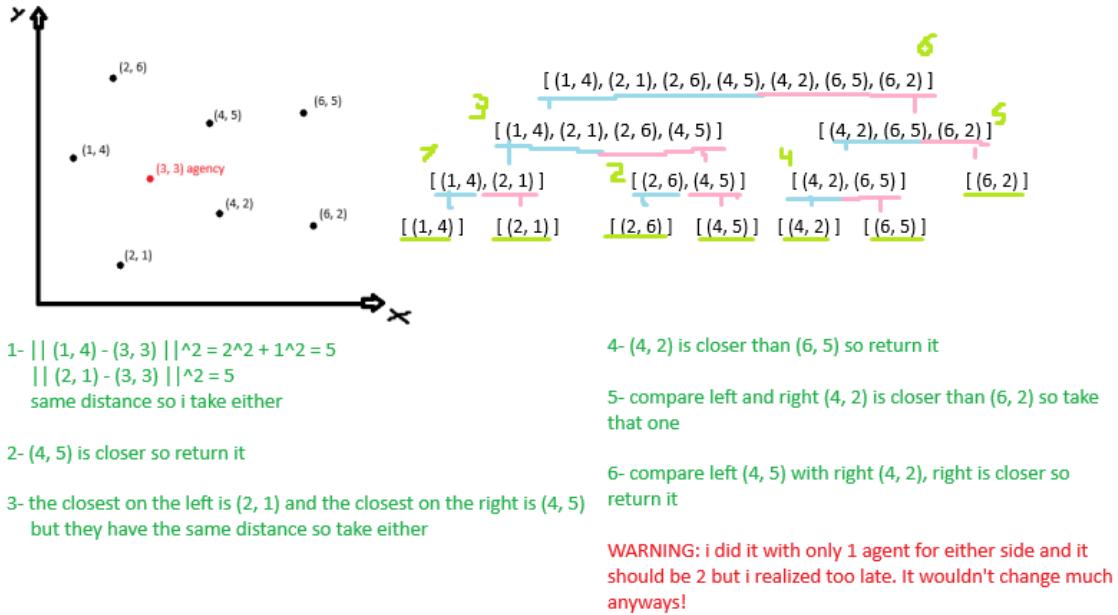
Complejidad:

$$T(n) = AT(n/B) + O(n^C)$$

- A = 2
- B = 2
- C = 0 (ya que lo único que se hace que no sea la recursividad son ifs y calculos de dist)

$$\log_2(2) = 1 > C \Rightarrow T(n) = O(n^{\log_B(A)}) = \mathbf{O(n)}$$

Ejemplo:



pseudocódigo:

```
get_closest(agents, base, start, end):

    if start >= end:
        return agents[start], || agents[start] - base ||

    mid = start + end / 2

    left, l_dist = get_closest left side
    right, r_dist = get closest right side

    return left, l_dist if l_dist >= r_dist else right, r_dist
```

12-

Para determinar si un número es primo existen varios algoritmos propuestos. Entre ellos el test de Fermat. Este es un algoritmo randomizado que opera de la siguiente manera: Dado un número entero "n", seleccionar de forma aleatoria un número entero "a" coprimo a n. Calcular  $a^{n-1}$  módulo n. Si el resultado es diferente a 1, entonces el número "n" es compuesto. La parte central de esta operación es la potenciación. Podríamos algorítmicamente realizarla de la siguiente manera:

```
raw
pot = 1
Desde i=1 a n-1
    pot = pot * a
```

En este caso se realizan  $O(n)$  multiplicaciones. Proponga un método usando división y conquista que resuelva la potenciación con menor complejidad temporal.

## Solution

Algoritmo:

El algoritmo que propongo es uno que divide el calculo a la mitad si el exponente es par y llama recursivamente para calcular esa mitad y devuelve la multiplicacion de esa mitad con ella misma, en caso de que el exponente sea impar se hace -1 y se divide a la mitad y denuevo se llama recursivamente pero esta vez devuelve la multiplicacion de la mitad con ella misma y luego por el n que le sacamos cuando hacemos -1. el caso base es cuando el exponente es 0 y devuelve 1.

Complejidad:

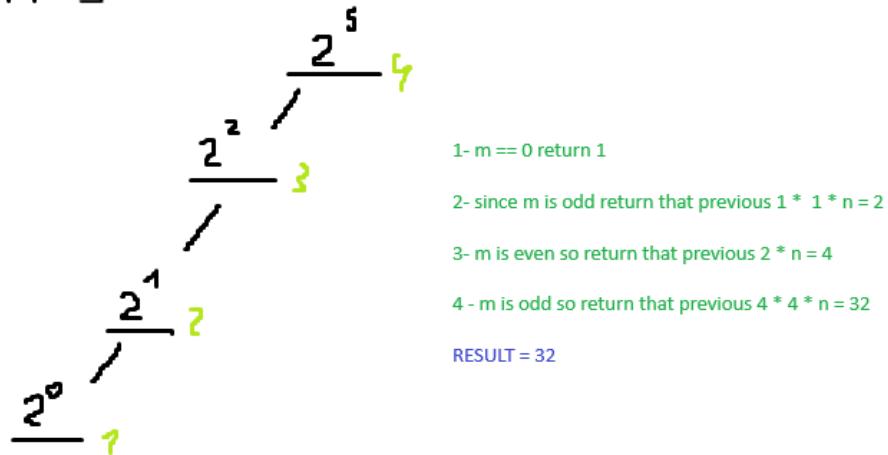
$$T(n) = AT(n/B) = O(n^C)$$

- A = 1
- B = 2
- C = 0

$$\log_2(1) = 0 = C \Rightarrow T(n) = O(\log n)$$

Ejemplo:

$$n = 2, m = 5$$



In [ ]: # Code

```
def power(n, m):  
    if m == 0:  
        return 1  
  
    if m % 2 == 0: # Even  
        half = power(n, m // 2)  
        return half * half  
    else: # Odd  
        half = power(n, (m-1) // 2)
```

```
    return half * half * n  
  
print(power(2, 5))
```

32

## 13-

Dado "L" un listado ordenado de "n" elementos y un elemento "e" determinado. Deseamos conocer la cantidad total de veces que "e" se encuentra en "L". Podemos hacerlo en tiempo  $O(n)$  por fuerza bruta. Presentar una solución utilizando división y conquista que mejore esta complejidad.

**Solution:**

**Algoritmo:**

El algoritmo que propongo es uno que usa busquedas binarias para encontrar la primera posición y para encontrar la última posición en la que se encuentra un dado elemento en la lista, luego se procede a calcular la cantidad con última pos - primera pos + 1, el +1 es para incluir el límite.

**Complejidad:**

La complejidad temporal es  $O(2\log n) = O(\log n)$

**Ejemplo:**

borre el ej sin querer :(

**Pseudocódigo:**

```
obtener repeticiones (elem, array):  
    primera pos = busqueda binaria de la primera pos  
    ultima pos = busqueda binaria de la ultima pos  
  
    si no hay posiciones el elem no esta en el array  
    si esta:  
        reps = ultima pos - primera pos + 1  
  
busqueda binaria (array, elem, inicio, fin, buscando primero):  
  
    si inicio >= fin:  
        devolver array[inicio]  
  
        mid = inicio + fin / 2  
  
        si buscando primero y mid < elem:  
            busqueda binaria derecha  
        si buscando primero y (mid > elem || mid == elem y mid-1 == elem):  
            busqueda binaria izquierda  
        si buscando primero y mid == elem y mid-1 != elem:  
            devolver mid
```

...mismo para si NO buscando primero # buscando ultimo

## 14-

Un conjunto de "n" personas votó de forma anónima entre un conjunto de "o" opciones (con  $o < n$ ). El resultado de la votación lo tenemos en un vector de n posiciones ordenado por opción seleccionada. Queremos determinar cuántos votos tuvo cada una de las opciones. Podemos hacerlo simplemente recorriendo el vector en  $O(n)$ . Sin embargo, utilizando división y conquista se puede lograr en un tiempo inferior. Presentar y analizar una solución utilizando división y conquista que logre lo solicitado.

### Solution

#### Algoritmo:

El algoritmo que propongo es uno que para cada una de las opciones busque la cantidad de veces que esta aparece en el array y aprovechando que el array esta ordenado y todas las personas que votaron a una misma opcion estan seguidas unas de otras vamos a utilizar el algoritmo planteado en el anterior problema. De esta forma se reconstruye cuantos votos tiene cada opcion.

#### Complejidad:

La complejidad temporal es la complejidad temporal del algoritmo anterior utilizado y es  **$O(log n)$**  esto es ya que  $o < n$  y el iterar esas opciones no aporta al análisis de complejidad.

#### Ejemplo:

Candidates:

- Goku
- Naruto
- Ichigo

[ goku, ichigo, ichigo, ichigo, naruto, naruto ]

Goku binary searches:

first pos = 0  
last pos = 0  
votes = last - first + 1 = 1

Naruto binary searches:

first pos = 4  
last pos = 5  
votes = last - first + 1 = 2

Ichigo binary search:

first pos = 1  
last pos = 3  
votes = last - first + 1 = 3

#### Pseudocódigo:

```
cantidad votos (candidatos, votos):  
    votos por candidato = {}
```

```

para cada candidato:
    primera pos = busqueda binaria de cantidato con primera pos = True
# O(Logn)
    ultima pos = busqueda binaria de cantidato con primera pos = False
# O(Logn)

    cantidad = ultima pos - primera pos + 1
    votos por candidato[candidato] = cantidad

devolver votos por candidato

```

## 16-

Dentro de un país existen dos colonias subacuáticas cada una de ellas con "n" habitantes. Cada habitante tiene su documento de identidad único identificado por un número. Para una tarea especial se decidió seleccionar a aquella persona que vive en alguna de las colonias cuyo número de documento corresponda a la mediana de todos los números de documento presentes en ellas. Por una cuestión de protocolo no nos quieren dar los listados completos de documentos. Solo nos responden de cada colonia ante la consulta "Cual es el documento en la posición X de todos los habitantes de la isla ordenados de mayor a menor". Utilizando esto, proponer un algoritmo utilizando división y conquista que resuelva el problema con la menor cantidad posibles de consultas. Analizar complejidad espacial y temporal.

### Solution

the median is the middle number in a set of values when those values are arranged from smallest to largest.

#### Algoritmo:

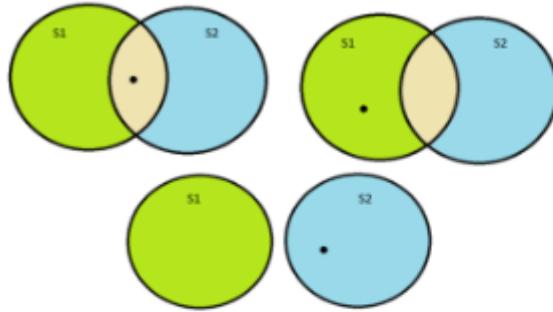
No sé como se supone q tenes q usar div y conquista pq se hace en tiempo constante con un par de calculos.

simplemente cada isla tiene n dnis y los dnis siempre si o si van a ser x, x+1, x+2, x+3 en secuencia así que el mas alto esta atras y el mas bajo adelante, hay que preguntar el mas bajo a las dos islas y el mas alto tambien, luego tomamos el min del los 2 bajos y el max de los dos altos y hacemos  $(\min + \max) / 2$  (division entera) y ya tenemos el median y se hacen un total de 4 llamados a las islas.

#### Complejidad:

- Temporal O(1)
- Espacial O(1)

#### Ejemplo:



$$\frac{s_1}{[0, 1, 2, 3]} \quad \cancel{\frac{s_2}{[0, 1]}}$$

lowest = 0, highest = 3  
median = 1

```
lowest = min(s1[0], s2[0])
highest = max(s1[1], s2[1])
median = lowest + highest / 2 got'em
that works bc ids have no gaps they're always in sequence

4 calls is the min amount of calls
```

In [ ]: # Code

```
def get_median (s1, s2):

    lowest = min(s1[0], s2[0])
    highest = max(s1[-1], s2[-1])

    return (lowest + highest) // 2

print(get_median([1, 3, 5], [2, 4, 6])) # [1, 2, 3*, 4*, 5, 6] (* = median)
```

3

## 17-

Se cuenta con un vector V de "n" elementos. Este vector visto de forma circular está ordenado. Pero no necesariamente en la posición inicial se encuentra el elemento más pequeño. Deseamos conocer la cantidad total de rotaciones que presenta "V". Ejemplo: V = [6, 7, 9, 2, 4, 5] se encuentra rotado en 3 posiciones. Podemos hacerlo en tiempo O(n) por fuerza bruta. Presentar una solución utilizando división y conquista que mejore esta complejidad.

### Solution

Algoritmo:

El algoritmo sera una especie de búsqueda binaria para encontrar la posición del menor elemento ya que esa sera la cantidad de rotaciones a la derecha que hubo, primero buscamos el mid, si es > al primer elemento es la parte rotada y hay que moverse a la derecha, si mid < al primer elemento hay que ir a la izq, si el mid es menor que los dos que lo rodean es el mínimo y devolverlo. el caso base es cuando queda 1 solo elemento devolverlo.

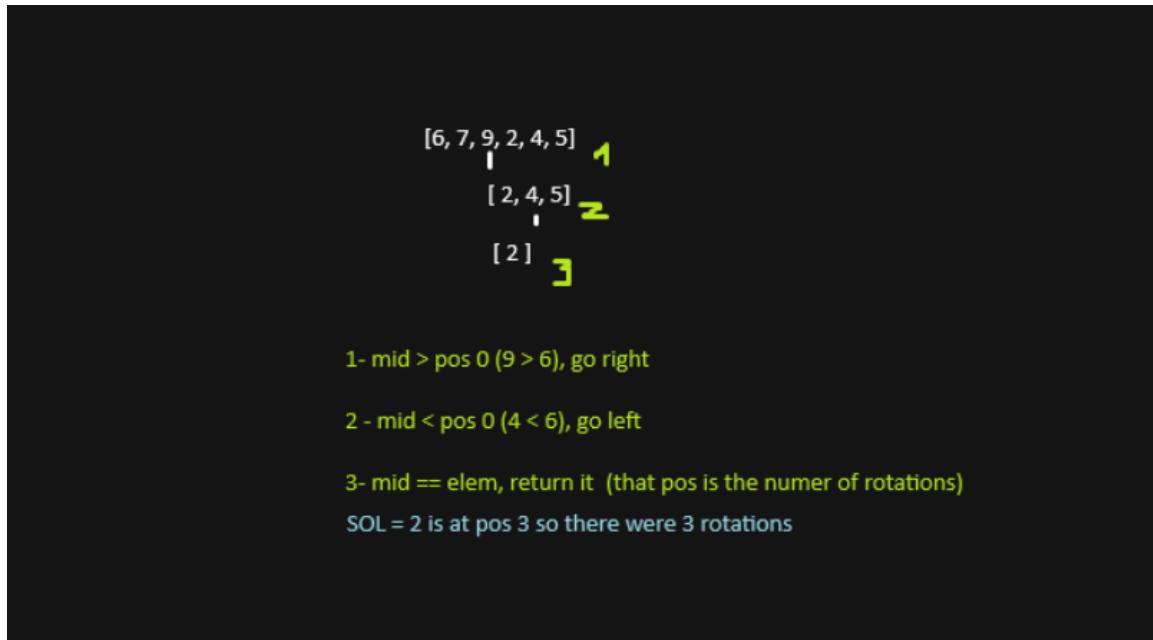
Complejidad:

$$T(n) = AT(n/B) + O(n^C)$$

- A = 1
- B = 2
- C = 0

$\log_2(1) = 0 = C \Rightarrow T(n) = O(n^C * \log_B(n)) = O(\log n)$

Ejemplo:



Pseudocódigo:

```
num_of_rotations(arr, start, end):
    if start >= end:
        return arr[start]

    mid = start + end / 2

    if mid < first elem and mid-1 > mid < mid+1:
        return mid
    if mid > first elem:
        move right
    if mid < first elem:
        move left
```

Skipped 10, 15, 18

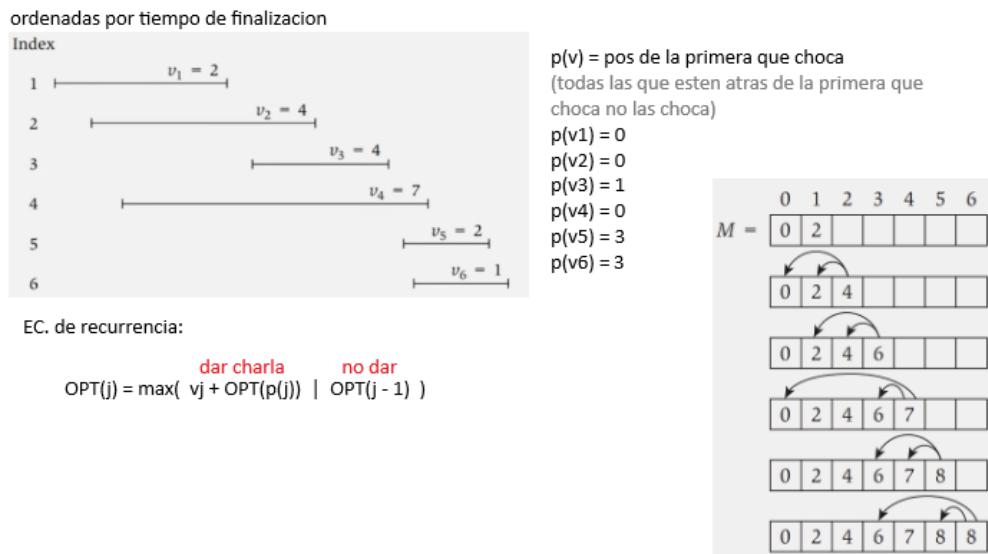
# Dynamic Programming

1. possible solution (con eq de recurrencia)
2. why is it dynamic programming?
3. complexity
4. examples
5. code or pseudocode

exercices

## Examples (buch)

### Scheduling con peso



```
def scheduling_con_peso(charlas):  
    ordenar charlas por tiempo de finalizacion acendiente  
    p = calculador de donde deja de solaparse cada charla  
  
    mem = [0] * (largo charlas + 1)  
    mem[0] = 0  
  
    for j in range(1, largo charlas + 1): # empezamos desde 1 ya que 0 es el caso base!  
        #           dar charla           no dar  
        mem[j] = max(peso de j + mem[p[j]], mem[j-1]) # Ec de recurrencia  
  
    return mem[-1]
```

```

def recup_solucion(mem, p, charlas, j, sol):
    if j == 0:
        return sol

    if peso de j + mem[p[j]] >= mem[j-1]: # se dio la charla j
        sol.append(j)
        return recup_solucion(mem, p, charlas, p[j], sol) # ver desde el
    que vino j
    else: # no se dio charla j
        return recup_solucion(mem, p, charlas, j-1, sol) # ver el de atras
    de j

```

## Los 2 escalones

Dada una escalera, y sabiendo que tenemos la capacidad de subir escalones de a 1 o 2 pasos, encontrar cuántas formas diferentes hay de subir la escalera.

Para una escalera nivel 0: 1 sola forma, quedarme estático = 1 forma

Para una escalera nivel 1: dar 1 paso simple = 1 forma

Para una escalera nivel 2: dar 2 pasos simples, o dar 1 paso doble = 2 formas

Para una escalera nivel n: ?

$$E[N] = E[N-1] + E[N-2]$$

## Juan el vago

Juan es ambicioso pero también algo vago. Dispone de varias ofertas de trabajo diarias, pero no quiere trabajar dos días seguidos. Dado un arreglo con el monto esperado a ganar cada día, determinar por programación dinámica el máximo monto a ganar, sabiendo que no aceptará trabajar dos días seguidos. Por ejemplo, si el arreglo es [100, 20, 30, 70, 50] el monto máximo que podrá ganar serán

180(*trabajarelprimer, tercerryquintodía*). En cambio, si el arreglo fuera [100, 20, 30, 70, 20] 170 (resultado de trabajar el primer y cuarto día).

```

[100, 20, 30, 70, 50]
mem: [ 0, 100, 100, 130, 170, 180 ]
sol: [ day5, day3, day1 ] reverse

```

```

[100, 20, 30, 70, 20]
mem: [ 0, 100, 100, 130, 170, 170 ]
sol: [ day4, day1 ] reverse

```

Recurrence relation:  $\text{OPT}(i) = \max( \text{OPT}(i-2) + i, \text{OPT}(i-1) )$

```

def get_max_income(days):
    mem = [0] * (len(days) + 1)

    mem[0] = 0
    for i in range(1, days+1):
        mem[i] = max(income of day i + mem[i-2], mem[i-1])

    return mem

def get_sol(days, mem):
    sol = []

    d = len(days)-1
    while d > 1:
        if income of day d + mem[i-2] >= mem[i-1]:
            sol.append(day d)
            d -= 2 # go to two days ago
        else:
            d -= 1 # go to one day ago

    if income on day 1 == mem[1]:
        sol.append(day 1)
    else:
        sol.append(0)

    return sol

```

## Problema de la mochila (pseudopolinomial)

no robo	si robo
---------	---------

$$\text{OPT}(n, W) = \max( \text{OPT}(n-1, W), \text{OPT}(n-1, W-P_i) + V_i )$$

## Problema del cambio (pseudopolinomial)

$\text{OPT}(\text{dinero}) = 1 + \min(\text{ para toda moneda } \leq \text{dinero}: \text{OPT}(\text{dinero} - \text{moneda}))$

```
def cant_monedas(sist_monetario, dinero):
    cant[0] = 0
    for i in range(1, dinero+1):
        minimo = i
        for moneda in sist_monetario:
            if moneda > i:
                continue
            cantidad = 1 + cant[i - moneda]
            if cantidad < minimo:
                minimo = cantidad
        cant[i] = minimo
    return cant[dinero]
```

Un algoritmo tiene complejidad **pseudopolinomial** cuando depende del valor numero de una variable y no la cantidad de bits, por ej un for  $i$  in range de 1 a  $n$ , depende del valor numerico de  $n$  y no del largo de los bits que ocupa

## Subset sum

se parece al prob. de la mochila

no usar elem                    usar elem

$\text{OPT}(n, v) = \max(\text{OPT}(n-1, V), \text{OPT}(n-1, V-V_i) + V_i)$

el segundo caso solo si  $V \geq V_i$  ofc

## Ejs

1-

Para una inversión inmobiliaria un grupo inversor desea desarrollar un barrio privado paralelo a la una ruta. Con ese motivo realizaron una evaluación de los diferentes terrenos en un trayecto de la misma. Diferentes inversores participarán, pero a condición de comprar algún terreno en particular. El grupo inversor determinó para cada propiedad su evaluación de ganancia. El mismo surge como la suma de inversiones ofrecida por el terreno menos el costo de compra. Debemos recomendar que terrenos contiguos comprar para que maximicen sus ganancias. Ejemplo:  $S = [-2, 3, -3, 4, -1, 2]$ . La mayor ganancia es de 5, comprando los terrenos de valor [4, -1, 2]. Solucionar el problema mediante un algoritmo de programación dinámica.

## Solution

Algoritmo:

El algoritmo propuesto es uno que va calculando el optimo de cada posicion basandose en el optimo anterior calculado con la siguiente relacion de recurrencia:

$$\text{OPT}(i) = \max(\text{OPT}(i-1) + v[i], v[i])$$

- Siendo la primera opcion la opcion de seguir sumando los elementos de atras mas los nuevos
- Y la segunda opcion la de empezar el subarray desde la posicion i ya que es mas conveniente

Dichos optimos seran recordados en un array del mismo largo de terrenos y el caso base sera el primer elemento del array ya que almenos tendremos esa suma.

Luego se hara el camino en reversa para reconstruir el subarray correspondiente, cuando se esta haciendo esto y la suma de  $\text{OPT}(i-1) + v[i] = \text{mem}[i]$  significaria que se tomo este camino y se lo agrega a la respuesta y si  $v[i] = \text{mem}[i]$  se agrega a la respuesta y termina porque significa que es el comienzo del subarray.

**Porque es PD:**

Es programacion dinamica ya que se estan calculando los problemas mas grandes a partir de los optimos ya calculados de los problemas mas pequeños los cuales estan siendo memoizados en un array para evitar recalculos.

**Complejidad:**

La complejidad temporal de este algoritmo sera **O(n)** ya que solo se recorre el array de terrenos y en cada iteracion solo se hacen comparaciones y calculos constantes.

La complejidad espacial sera **O(n)** tambien porque se crea un array de memoizacion del mismo tamaño del de entrada.

**Ejemplo:**

$$[-2, 3, -3, 4, -1, 2]$$


- $\text{mem} = [-2, 3, 0, 4, 3, 5] \rightarrow \text{ans is } 5$
- $\text{sol} = [2, -1, 4]$  reverse (when  $v[i] = \text{mem}[i]$  add it to the sol and stop)

- caso base:  $\text{arr}[0]$  pq al menos esa es la suma
  - se sigue      se empieza denuevo desde  $i$
  - $\text{OPT}(i) = \max(\text{OPT}(i-1) + v[i], v[i])$

In [ ]: # Code

```

def max_subarray_sum(arr):
    mem = [0] * len(arr)
    mem[0] = arr[0]

    for i in range(1, len(arr)):
        mem[i] = max(mem[i-1] + arr[i], arr[i])

    return mem[-1], mem
def generate_sol(arr, mem):
    sol = []
    for i in range(len(arr)-1, -1, -1):
        if mem[i] == arr[i]:
            sol.append(arr[i])
            break
        elif mem[i-1] + arr[i] == mem[i]:
            sol.append(arr[i])

    return sol

a = [-2, 3, -3, 4, -1, 2]
res, mem = max_subarray_sum(a)
sol = generate_sol(a, mem)
print(res, sol)

```

5 [2, -1, 4]

2-

Dado un grafo dirigido, aciclico  $G(V, E)$  con pesos en sus aristas, y dos vertices "s" y "t", queremos encontrar el camino de mayor peso que exista entre "s" y "t". Resolver mediante programacion dinamica.

## Solution

### Algoritmo:

El algoritmo propuesto es bellman ford pero buscando el camino maximo en lugar del minimo, tambien no hara falta tener en cuenta posibles ciclos ya que el grafo es aciclico.

Primero se crean 2 diccionarios donde se va a hacer la memorizacion, uno de distancias y otro de padres, el nodo S tendra distancia 0 y parente None, y los demas nodos tendran distancia -inf al comenzar, luego se hara un loop por la cantidad de nodos del grafo y dentro de este un loop para todas las aristas del mismo, aca se vera si `dist[nodo de donde sale la arista] + peso arista > dist[nodo al que le llega la arista]` se ser verdadero se actualiza la distancia a la nueva calculada y su parente por el nodo de donde sale la arista.

entonces la relacion de recurrencia sera

```
OPT(i) = max(    OPT(x) + peso arista, para cada x que tiene arista  
apuntando al nodo i      )
```

### Porque es PD:

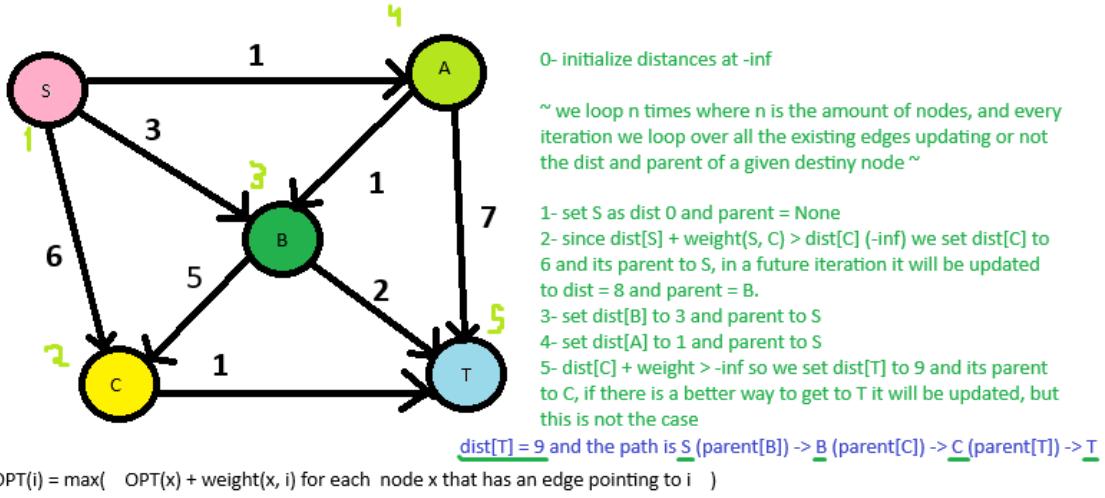
Es programacion dinamica ya que se aprovechan las distancias previamente calculadas de otros nodos para calcular las nuevas distancias de los otros nodos, entonces desde el caso base en el origen con dis = 0 y parente None, se empieza a resolver problemas mas grandes hasta tener la solucion.

### Complejidad:

La complejidad temporal es  $O(V * E)$  siendo V la cantidad de nodos y E las aristas.

La complejidad espacial es  $O(V)$  porque se crean 2 diccionarios de largo de nodos.

### Ejemplo:



Pseudocódigo:

```

def max_path(graph, origin, destination):
    dist = {}
    parents = {}

    for v in graph:
        dist[v] = -inf

    dist[origin] = 0
    parents[origin] = None

    for range(all vertices):
        for edge in all edges:
            from, to, weight = edge
            if dist[from] + weight > dist[to]:
                dist[to] = dist[from] + weight
                parents[to] = from

    return dist[destination], parents

def get_path(parents, destination):
    path = [destination]

    curr_node = destination
    while parents[curr_node] is not None:
        path.append(parents[curr_node])

    return path

```

4-

La organización de una feria internacional tiene que programar diferentes eventos a realizar en su escenario principal. Para ello pueden elegir, en los diferentes días del evento, entre algunos de los siguientes rubros: un cantante, una compañía de danza, un show de variedades o un humorista. Disponen de una oferta de cada tipo para cada día y la posible ganancia por venta de entradas. Existen ciertas restricciones que se aplican. No se pueden repetir 2 días seguidos el mismo rubro. Además por el tiempo de preparación un día después de un cantante solo puede presentarse un humorista. Plantear la resolución mediante programación dinámica.

## Solution

Algoritmo:

Porque es PD:

Complejidad:

Ejemplo:

Pseudocódigo:

## 5-

Una agencia de inteligencia ha conseguido interceptar algunos mensajes encriptados de una agencia rival. Mediante espionaje logran saber algunas cosas para desencriptar los mensajes. Por ejemplo, que para dificultar la tarea de los criptoanalistas los mensajes enviados no contienen espacios. Se ha organizado un grupo de trabajo para generar un algoritmo para quebrar la seguridad. El trabajo se dividió en diferentes partes. A usted le toca, dado un string "desencriptado" y sin espacios, determinar si lo que se lee corresponde a un posible mensaje en idioma castellano. El proceso debe ser rápido dado que se debe utilizar muchas veces. Cuenta con un diccionario de "n" palabras. y con una cadena de texto con el posible mensaje.

Ejemplo: si el diccionario es "peso", "pesado", "oso", "soso", "pesa", "dote", "a", "te", para la cadena de texto "osopesadotepesa". Existe un posible mensaje con las palabras "oso", "pesado", "te", "pesa". Para la cadena de texto "ososoapesadote". No existe un posible mensaje

- Construir una solución que informe si la cadena de entrada es un posible texto utilizando programación dinámica.
- Existe la posibilidad de que una cadena de texto puede corresponder a más de un mensaje. Modifique su solución para que se informen todos los posibles mensajes. Determine el impacto en las complejidades en el nuevo algoritmo.

## Solution

Algoritmo:

El algoritmo es uno que inicialmente crea una matriz donde las filas seran asignadas a un i entre 0 y n (len del codigo) y las columnas asignadas a un j entre 0 y n. de tal manera que si matriz en posicion i,j tendra un booleano que dira si es valido el codigo cuando la ultima palabra es la que empieza en posicion i y termina en posicion j. de esta manera usamos la ecuacion de recurrencia:

$$\text{opt}(i) = \exists \text{ word } (i, j) \& \text{ opt}(j-1)$$

Existe palabra valida entre i - j & existe palabra que termina donde empieza mi nueva palabra

Luego se ve si se llego al final del codigo siendo True en cuyo caso es valido, caso contrario no lo es.

## Porque es PD:

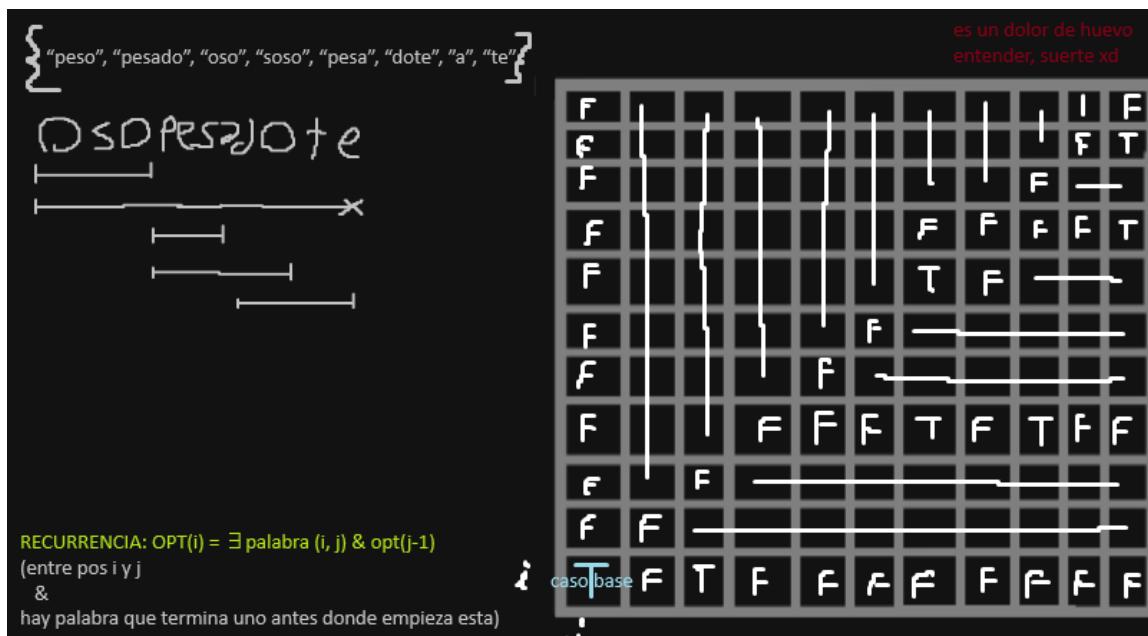
Es programacion dinamica ya que se calcula cada valor de la matriz a partir de un estado actual en conjuncion con informacion previamente calculada en pasos anteriores, en este caso si existe una palabra que termina donde empieza mi nueva palabra. y cada subproblema es cada substring existente con combinaciones de i y j donde ambos van de 0 a n.

**Complejidad:**

La complejidad temporal de este algoritmo es  **$O(n^2)$**  ya que por cada  $i$  de  $0-n$  se ven los  $j$  de  $i-n$ .

La complejidad espacial es tambien  **$O(n^2)$**  por la matriz de optimos.

## Ejemplo:



### Pseudocódigo:

```
def is_valid(code, dic):
    opts = [ [ False ] * len(code) ] * len(code) # n x n where n is the
```

```

Length of the code
    opts[0][0] = True # base case

    for i in range(len(code)):
        for j in range(i, len(code)):
            if (code[i:j] in dic) and (opts -> exists a word that starts
at any i and ends at j-1):
                opts[i][j] = True

    if opts -> theres a True on j = len(code) - 1 for some i: # Means the
entirety of the code is made from words in the dictionary
        return True
    return False

```

## 6-

Una empresa de transporte aéreo cuenta con un avión que cubre una ruta que une dos ciudades. Se especializa en llevar maquinaria pesada. Puede realizar un vuelo diario y trasladar una cantidad de peso determinada. Por cada kilo transportado cobra una tarifa. Por día recibe un pedido de transporte que puede aceptar o rechazar en su totalidad (no se pueden fraccionar). Si rechaza un pedido, lo pierde y no lo puede realizar otro día. Existe una regulación especial que indica que la carga máxima a transportar disminuye según una fórmula por día. Se restablece a su máximo únicamente si el avión pasa por revisión e inspección. Este proceso insume 3 días seguidos. El gerente de la empresa cuenta con una planilla que detalla los envíos solicitados en el próximo mes y la tabla de peso máximo por día pasado desde la última revisión. El avión llegó el día anterior de la revisión. Construir un algoritmo que permita maximizar las ganancias.

### Solution

#### Algoritmo:

El algoritmo propuesto es uno que va calculando los optimos de los n días en función de los n pesos máximos posibles por día. El caso base será cuando el peso es 0 y/o los pedidos son 0. Luego se guardará en una matriz los resultados de realizar unos determinados pedidos teniendo en cuenta cuando sería conveniente recargar el avión, siguiendo la siguiente relación de recurrencia:

$$OPT(i, p) = \max(OPT(i-1, p-1) + V_i \text{ (if possible)}, OPT(i-3, P_{\max}) + V_i)$$

- En el primer caso si es posible toma el valor anterior más el nuevo valor que entra.
- El otro caso es cuando decidimos que es mejor tomar el nuevo valor a cambio de arreglar el avión por los pasados 3 días.

#### Porque es PD:

Es programacion dinamica ya que se usa el precalculo de optimos anteriores para cada calculo de optimo actual, empezando del caso base 0. Se usa memoizacion en forma de matriz nxn.

**Complejidad:**

La complejidad temporal del algoritmo sera **O(n^2)**

La complejidad espacial del algoritmo sera **O(n^2)** por la matriz memo.

**Ejemplo:**

pesos max: [ 100, 70, 50, 20 ]  
pedidos: [ (20, 100), (30, 40), (10, 40), (500, 30) ]

Recurrencia:  
 $OPT(i, p) = \max_{m_{2i}} \begin{cases} OPT(i-1, p-1) + v_i & \text{IF } A_{2i} \\ OPT(i-3, p) + v_i & \text{else} \end{cases}$

$i_4$	0	0	0	0	500
$i_3$	0	20	50	60	60
$i_2$	0	20	50	50	50
$i_1$	0	20	20	20	20
$i_0$	0	0	0	0	0
	$P_0$	$P_1$	$P_2$	$P_3$	$P_4$

**Pseudocodigo:**

# Copyright - Flor atss

```

def max_pedidos(pedidos, pesos):
    memo = matriz nxn # n = Largo de pedidos
    memo[0][0] = 0 # caso base

    para cada i en pedidos:
        para cada p en pesos:
            si capacidad de i < p:
                memo[i][p] = max(memo[i-1][p-1] + valor de i, memo[i-3][p
maximo] + valor de i)
            else:
                memo[i][p] = max(memo[i-1][p-1], memo[i-3][p maximo] +
valor de i)

    devolver memo[-1][-1] # esquina arriba derecha es el OPT

```

Contamos con una carretera de longitud  $M$  km que tiene distribuidos varios carteles publicitarios. Cada cartel " $i$ " está ubicado en un " $k_i$ " kilómetro determinado (pueden ubicarse en cualquier posición o fracción de kilómetro) y a quien lo utiliza le asegura una ganancia " $g_i$ ". Por una regulación no se puede contratar más de 1 cartel a 5km de otros. Queremos determinar qué carteles conviene contratar de tal forma de maximizar la ganancia a obtener.

## Solution

### Algoritmo:

El algoritmo propuesto es el que recorre el array de carteles y va calculando la ganancia maxima de acuerdo a la ecuacion de reccurrencia:

$$OPT(i) = \max(OPT(x) + G_i, OPT(i-1)) \text{ con } x \text{ el anterior compatible a } i$$

- El primer caso es el que conviene sumar el valor del cartel  $i$  con su anterior compatible
- El segundo es cuando conviene mas mantener el optimo calculado hasta el cartel  $i-1$

Los optimos se memoizan en un array del tamaño del array de carteles + 1 ya que se agrega el caso base 0.

Luego se procede a tomar el camino inverso para reconstruir cuales son los carteles a comprar.

### Porque es PD:

Es programacion dinamica ya que se consigue la solucion optima a partir de precalculos de soluciones a problemas mas pequeños anteriores.

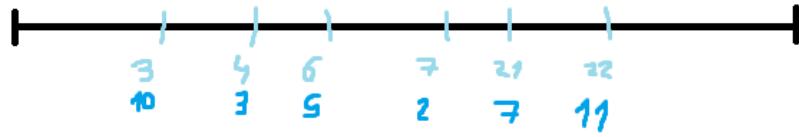
### Complejidad:

La complejidad temporal es **O(n)** por ver todos los carteles.

La complejidad espacial es **O(n)** por el array de memoizacion.

### Ejemplo:

cartels (gi, ki): [ (10, 3), (3, 4), (5, 6), (2, 7), (7, 21), (11, 22) ]



```
mem = [ 0, 10, 17, 21 ]
sol = [ (11, 22), (10, 3) ] reverse
```

$$O_{p+1}(i) = \max \left( O_{p+1}(x) + G_i, O_{p+1}(i-1) \right)$$

X = anterior compatible

Pseudocódigo:

```
def max_ganancia_carteles(carteles):
    mem = [0] * largo carteles + 1

    para cada i en largo carteles + 1:
        mem[i] = max(mem[x] + Gi, mem[i-1]) # donde x es el anterior
    compatible de i

    return mem, mem[-1]

def reconstruir_camino(carteles, mem):
    sol = []

    i = largo carteles
    mientras que i > 0:
        si mem[x] + Gi > mem[i-1]: # x el anterior compatible de i
            sol.append(i)
            i = x
        sino:
            i -= 1

    return sol
```

9-

Un ramal ferroviaria pone en concesión los patios de comida en todas las estaciones. Son en total "n" estaciones. Por cada estación se cuenta con el promedio de facturación de los últimos 5 años. Por normativa antimonopólica existe como límite que ninguna empresa puede explotar 3 o más estaciones contiguas. Pero, no existe una cantidad máxima de estaciones a explotar. Un oferente nos solicita que le digamos cuáles son las estaciones que

le conviene obtener para maximizar sus ganancias. Plantee la solución mediante programación dinámica.

## Solution

### Algoritmo:

El algoritmo propuesto es uno que al checkear si debe comprar un elemento compara que es mejor si agarrar el elemento y su anterior compatible o no agarrar el elemento y seguir como estaba acordemente a la siguiente relación de recurrencia:

```
OPT(i) = max( OPT(x) + Vi, OPT(i-1) ), con x anterior compatible a i  
(puede ser i-1 o i-2)
```

- primer caso se agarra el elemento
- segundo caso se deja el elemento

Se memorizan las respuestas anteriores en un array del largo del array de entrada + 1, donde el primer index es el caso base 0.

### Porque es PD:

Es programacion dinamica ya que se esta usando el optimo calculado hasta el anterior compatible de i y tambien el optimo calculado al inmediatamente anterior a i, para resolver el problema mayor que incluye esos dos problemas mas chicos. Se memoizan todos los calculos de optimos.

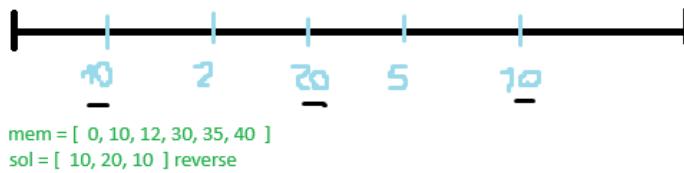
### Complejidad:

La complejidad temporal es **O(n)** ya que se recorre el array.

La complejidad espacial tambien es **O(n)** por el array de memoizacion.

### Ejemplo:

arr: [ 10, 2, 20, 5, 10 ]



Recurrence relation:

$$\text{OPT}(i) = \max_{\substack{\text{take it} \\ \text{leave it}}} (\text{OPT}(x) + V_i, \text{OPT}(i-1))$$

### Pseudocodigo:

# Es muy parecido al pseudocodigo del anterior problema 8.

## 11-

La detección de ciclos negativos tiene una variedad de aplicaciones en varios campos. Por ejemplo en el diseño de circuitos electrónicos VLSI, se requiere aislar los bucles de retroalimentación negativa. Estos corresponden a ciclos de costo negativo en el grafo de ganancia del amplificador del circuito. Tomando como entrada de nuestro problema un grafo ponderado con valores enteros (positivos y/o negativos) dirigido donde un nodo corresponde al punto de partida, queremos conocer si existe al menos un ciclo negativo y en caso afirmativo mostrarlo en pantalla. Proponer una solución al problema que utiliza programación dinámica.

### Solution

#### Algoritmo:

El algoritmo a utilizar es bellman ford para poder mientras se calcula el camino mínimo (lo cual no nos interesa) encontrar un loop negativo, luego a partir de este se reconstruye el camino infinito viendo cada parente de uno de los nodos del loop hasta que se llegue a si mismo denuevo y se vuelva a empezar el loop.

#### Porque es PD:

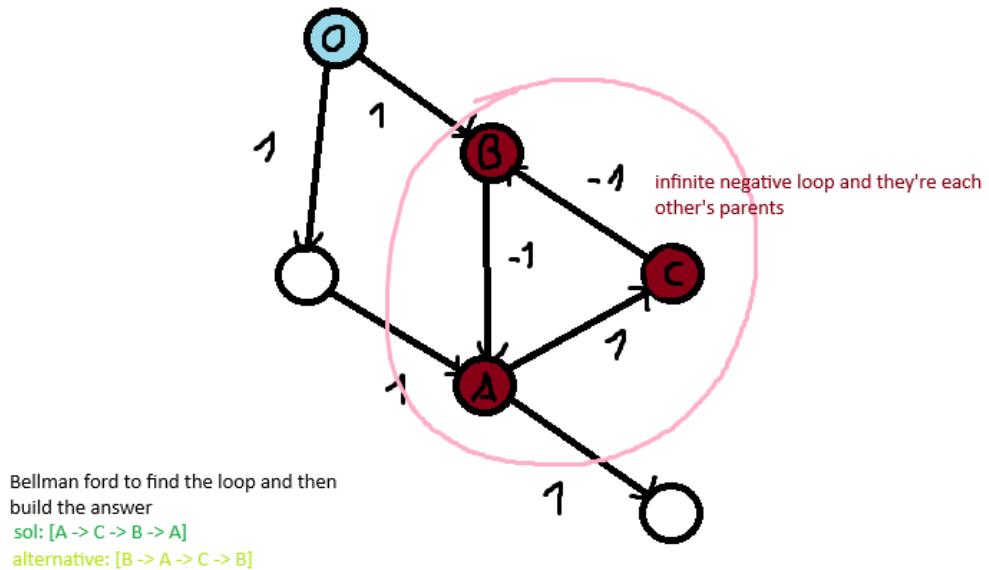
Es programacion dinamica porque usa la distancia calculada previamente y memorizada para resolver los proximos problemas.

#### Complejidad:

Complejidad temporal de bellman ford **O(V \* E)**

Complejidad espacial **O(v)** por los 2 diccionarios de distancias y de padres.

Ejemplo:



Pseudocódigo:

```

def bellman_ford(graph, origin):
    dist = parents = {}
    for v in graph:
        dist[v] = float('inf')
    dist[origin], parents[origin] = 0, None
    edges = get_all_edges(graph)

    for _ in range(len(graph)):
        for origin, dest, weight in edges:
            if dist[origin] + weight < dist[dest]:
                parents[dest] = origin
                dist[dest] = dist[origin] + weight

    node_in_negative_loop = None
    for v, w, weight in edges:
        if dist[v] + weight < dist[w]: # Negative Loop found
            node_in_negative_loop = v

    return build_sol(node_in_negative_loop, parents)

def build_sol(v, parents):

    sol = [v]

    w = parents[v]
    while w != v:
        sol.append(parents[w])
    
```

```
return sol
```

## 12-

Un dueño de un camión de transporte se comprometió a trasladar una carga de una ciudad A a la ciudad B. Para realizar el recorrido puede optar por diferentes caminos que pasan por diferentes ciudades intermedias. Nos acerca un mapa donde para cada tramo que une las diferentes ciudades indica el costo de realizar el mismo. Vemos que algunos costos son positivos (combustible, peaje, etc) y otros negativos o cero (yendo por esos tramos puede ganar unos pesos haciendo algunos encargos "particulares"). Nos solicita que le informemos cuál sería el recorrido óptimo para minimizar el gasto total del viaje. Presentar un algoritmo polinomial utilizando programación dinámica que lo resuelva.

### Solution

Algoritmo:

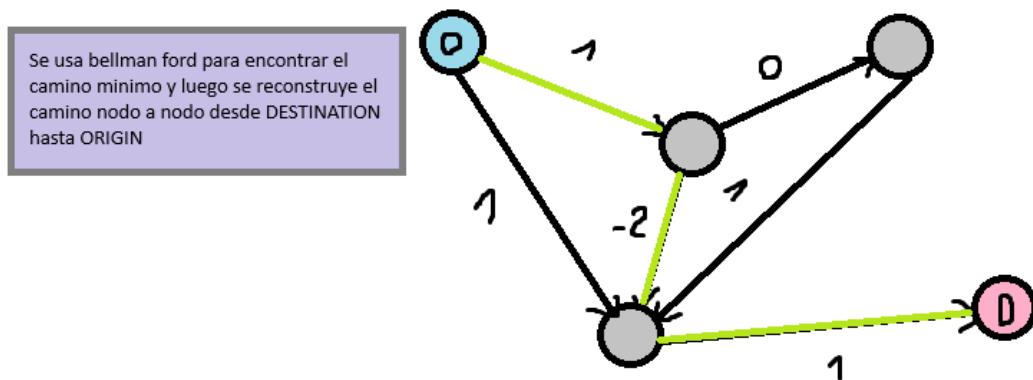
Es literal bellman ford solo que despues se reconstruye el camino ;)

Complejidad:

Temporal  $O(V * E)$

Espacial  $O(V)$

Ejemplo:



Recurrencia	By victor pobde (ta medio raro para mi pero weno atsss)
$\text{minPath}(s', j) = 0$	
$\text{minPath}(n_{i,0}) = +\infty \text{ con } n_i \neq s$	
$\text{minPath}(n_i, j) = \min \left[ \begin{array}{l} \text{minPath}(n_i, j-1) \\ \min \left[ \text{minPath}(n_s, j-1) + w(n_s, n_i) \right] \end{array} \right]$	con $n_s \in \text{pre}(n_i)$

Pseudocódigo:

# Ya fue mostrado previamente en el ejercicio anterior y es muy parecido solo con algunos cambios ya funciona!

## 14-

Para un nuevo satélite a poner en órbita una empresa privada puede optar por incluir diversos sensores a bordo (por ejemplo: variación de temperatura, humedad en tierra, caudal de ríos, etc). Cada uno de ellos tiene un peso "pi" y una ganancia "gi" calculado por su uso durante la vida útil del satélite. Si bien les gustaría incluir todos, el satélite tiene una carga máxima P que puede llevar. Nos piden que generemos un algoritmo (utilizando programación dinámica) para resolver el problema. Indique si su solución es polinomial.

### Solution

#### Algoritmo:

El algoritmo crea una matriz de  $n \times w$  donde se va a memoizar todos los optimos calculados, empezando por el caso base donde el peso es 0 y/o la cantidad de elementos es 0. Luego se itera la lista de 0 a n y por cada iteracion se iteran los pesos de 0 a w (*eso es lo que lo hace pseudopolinomial*) y por cada una de estas iteraciones se usa la relacion de recurrencia:

$$\text{OPT}(i, w) = \max(\text{OPT}(i-1, w-w_i) + v_i, \text{OPT}(i-1, w))$$

- en el primer caso se toma el elemento
- en el segundo se lo deja

#### Porque es PD:

Es programacion dinamica ya que para el calculo de cada optimo se necesita y utiliza calculos de optimos previos y memoizados en la matriz.

#### Complejidad:

Complejidad temporal es **O( $n * 2^m$ )** donde  $m \sim \log_2(w)$  y es **Pseudopolinomial**.

[EXPLICACION ACA](#)

Complejidad espacial es **O( $n * w$ )**

#### Ejemplo:

(valor, peso)  
 sensors: [ (2,1), (5, 4), (1, 3), (4, 2) ]  
 weight = 4

No tiene pq ser cuadrada, es solo un ej

$S_1$	0	2	2	6	6
$S_2$	0	2	2	2	5
$S_3$	0	2	2	2	5
$S_4$	0	2	2	2	2
$S_5$	0	0	0	0	0

Recurrence relation:

take it (mientras q  $w_i - w_j \geq 0$ )

$$DP(i, w) = \max \begin{cases} DP(i-1, w-w_i) + v_i \\ DP(i-1, w) \end{cases}$$

leave it

(problema de la mochila) W es el peso disponible y Wi es el peso del sensor i

Pseudocódigo:

Crear matriz  $n + 1 \times w + 1$  con todos 0

```
Desde i=1 a n # sensors
    Desde w=1 a W # weights
        matriz[i][w] = max(matriz[i-1][w-Wi] + Vi si w-Wi >= 0, matriz[i-1, w])
```

Retornar esquina arriba derecha de la matriz

## 16-

Luego de aprender programación dinámica un estudiante en una charla de café le explica a un amigo que atiende un negocio el algoritmo de cambio mínimo de monedas. Con eso, afirma, podrá optimizar el despacho de mercadería. El comerciante despacha pedidos de cierto producto que viene en empaques prearmados de distintas cantidades o sueltos en unidades. Sin embargo, el amigo le replica que su algoritmo es poco realista. Supone que uno tiene una cantidad ilimitada de empaques de cada presentación. Luego de pensar unos momentos, el estudiante llega a una variante de este problema teniendo en cuenta esta restricción. ¿Podría usted detallar cuál es esta solución?

## Solution

Algoritmo:

El algoritmo propuesto es uno que primero crea una matriz donde irá memoizando todos los optimos, el caso base sera cuando damos 0 packs. Luego tendremos que agarrar los packs que minimicen la cantidad de estos a usar, y al mismo tiempo tenemos que tener en cuenta la posibilidad de quedarnos sin packs de algun tipo. Seguirá la siguiente relación de recurrencia:

```
OPT(pedido, cantidades) = 1 + min(para cada pack <= pedido: OPT(pedido-
pack, cantidades[pack]-1))
```

Al final tendremos la cantidad minima de packs posibles a usar, luego a partir de esto se podra hacer el camino inverso para reconstruir que packs se usaron.

**Porque es PD:**

Es programacion dinamica ya que se utiliza informacion calculada en anteriores pasos para llegar a la solucion optima de el paso actual, y asi hasta llegar al optimo general.

**Complejidad:**

La complejidad sera **O(n \* pedido)** = **O(n \* 2^m)** siendo que n es la cantidad de packs y m = log<sub>2</sub>(pedido) asi que 2<sup>m</sup> = pedido. es complejidad pseudopolinomial.

La complejidad espacial sera **O(n \* pedido)** por la matriz de memoizacion.

**Ejemplo:**

amount of packs: [ 3, 1, 4, 6, 0 ]  
 quantities per pack: [ 1, 3, 5, 7, 9 ]  
 so there're packs with 1, 3, 5 items and so on

ordered quantity: **10**

me dio alta page hacer ejemplo en matriz xd

#### RECURRANCE

$OPT(q, a) = 1 + \min( \text{for each pack } \leq q: OPT(q-\text{pack}, a[\text{pack}]-1) )$

q = quantity

a = amount of each pack so a[pack] is the amount left of that pack

**Pseudocodigo:**

# esta medio asi nomas ngl

```
def cant_packs(packs, cantidades, pedido):
    crear matriz n + 1 x pedido + 1 todo en 0 # con caso base

    for i in range(1, pedido+1):
        minimo = i
        for pack in packs:
            if pack > i:
                continue
            cantidad = 1 + matriz[i - pack][cantidades de pack - 1]
            if cantidad < minimo:
                minimo = cantidad
```

```

else:
    cantidades de pack + 1 # pq no lo uso al final
    matriz[i][cantidades] = minimo
return matriz[-1][-1]

```

## 17-

Se conoce como "Longest increasing subsequences" al problema de, dado un vector de numérico, encontrar la subsecuencia más larga de números (no necesariamente consecutivos) donde cada elemento sea mayor a los anteriores. Ejemplo: En la lista → 2, 1, 4, 2, 3, 9, 4, 6, 5, 4, 7. Podemos ver que la subsecuencia más larga es de longitud 6 y corresponde a la siguiente "1, 2, 3, 4, 6, 7". Resolver el problema mediante programación dinámica.

### Solution

#### Algoritmo:

El algoritmo propuesto es uno que primero consigue el array de anteriores compatibles a cada elemento i, y luego recorre el array aplicando la relación de recurrencia y guardando los cálculos en un array de memoización.

```
OPT(i) = max( OPT(i-1), OPT(x[i]) + 1 )
```

- La primera opción es si queda igual
- La segunda es si el anterior compatible  $x[i] + 1$  es mejor

Para encontrar el camino tomado se aplica igual pero en reversa teniendo en cuenta los seleccionados.

#### Porque es PD:

Es programación dinámica ya que se subdividen los problemas en problemas más chicos cuya solución sirve para resolver los próximos problemas hasta llegar al final.

#### Complejidad:

La complejidad temporal es **O(n)** lo que cueste crear X jeje

La complejidad espacial sera **O(n)** por el array memo

#### Ejemplo:

arr:[ 2, 1, 4, 2, 3, 9, 4, 6, 5, 4, 7 ]  
 mem = [ 0, 1, 1, 2, 2, 3, 4, 4, 5, 5, 5, 6 ]  
 sol = [ 7, 6, 4, 3, 2, 1 ] reverese

**Recurrence relation:**  
 $\text{OPT}(i) = \max( \text{OPT}(i-1), \text{OPT}(x) + 1 )$   
 no mejora    si mejora

Pseudocódigo:

```

def max_len(arr, x):
    memo = [0] * (len(arr) + 1)

    for i in range(1, len(arr)):
        memo[i] = max(memo[i-1], memo[x[i]] + 1)

    return memo, memo[-1]
  
```

18-

Se define el problema 2-Partition de la siguiente manera: Se cuenta con un conjunto de "n" elementos. Cada uno de ellos tiene un valor asociado. Se desea separar los elementos en 2 conjuntos que cumplan con: La suma de los valores de cada conjunto sea igual entre ellos. No se conoce un algoritmo eficiente para resolver este problema. Sin embargo - al igual que otros problemas puede ser resuelto utilizando programación dinámica de forma pseudopolinomial. Presente una solución al problema utilizando dicha metodología.

## Solution

Refs:

- geekforgeeks
- educative
- Neetcode

## Algoritmo:

El algoritmo es uno que primeramente creara una matriz de optimos con booleanos, el caso base sera cuando no hay elementos o cuando la sum // 2 es 0. luego a partir de ahí se vera si existe una forma que con los elementos del array se llegue a sumar la mitad de la suma

total de todos los elementos, porque significaría que la otra mitad hace lo mismo. Para esto primeramente checkeamos que el  $\sum // 2$  sea un entero, sino lo es no hay forma de separar en 2 el array. Si es entero entonces tendremos que iterar el array y para cada iteración iterar nuevamente pero desde 0 hasta la  $\sum // 2$  dentro de esta iteración se aplica la ecuación de recurrencia:

$$\text{OPT}(i, v) = \text{OPT}(i-1, v) \ || \ \text{OPT}(i-1, v-v_i)$$

- en el primer caso significaría que ya existía previamente forma de llegar a esa suma  $v$ .
- en el segundo caso significa que existe forma de que la suma hasta  $i-1$  sea igual al resto que le falta a  $i$  para lograr abarcar todo  $v$ .

Now, if  $\text{OPT}(\text{len(arr)}, \sum(\text{arr}) // 2)$  is True, it means that there exists a subset of elements whose sum is equal to half of the total sum of all elements. This subset, when combined with the remaining elements, will give a total sum equal to the full sum of the array.

#### Porque es PD:

Es programación dinámica ya que se está usando los óptimos previamente calculados para encontrar los óptimos de los problemas más grandes los cuales contienen a los problemas más pequeños, esto se memoiza en una matriz  $n \times \sum // 2$

#### Complejidad:

La complejidad temporal es de  $O(n * \sum // 2) = O(n * \sum)$  donde  $n$  es el largo del array y  $\sum$  la suma de los elementos de este, como eso es un número esta complejidad es pseudopolinomial.

La complejidad espacial será la misma  $O(n * \sum)$  ya que se memoizan los óptimos, hay forma de gastar menos pero idc rn jeje

#### Ejemplo:

arr: [ 1, 3, 5, 2, 3, 2 ]  
 $v = 8$  ( $8 \% 2 = 0$  so there could be a sol)

$i_6$	T	T	T	T	T	T	T	T	T
$i_5$	T	T	F	T	T	T	T	T	T
$i_4$	T	T	F	T	T	T	T	T	T
$i_3$	T	T	F	F	T	T	T	F	T
$i_2$	T	T	F	F	T	F	F	F	F
$i_1$	T	T	F	F	F	F	F	F	F
$i_0$	T	F	F	F	F	F	F	F	F

Recurrence relation      si  $v_i >= 0$   
 $\text{OPT}(i, v) = \text{OPT}(i-1, v) \ || \ \text{OPT}(i-1, v-v_i)$   
ya  $\exists$  suma hasta ahí || existe suma hasta mi anterior que suma el resto que me falta para llegar a  $v$

Pseudocódigo:

```
def findPartition(arr, n):
    Sum = sum(arr)

    if (Sum % 2 != 0):
        return False

    memo = [[False] * ((Sum // 2) + 1)] * (n + 1)

    for i in range(n):
        for v in range(Sum // 2, arr[i] - 1, -1):
            memo[i][v] = memo[i - 1][v] or memo[i - 1][v - arr[i]]

    return memo[-1][-1]
```

19-

Una variante del problema de la mochila corresponde a la posibilidad de incluir una cantidad ilimitada de cada uno de los elementos disponibles. En ese caso, tenemos una mochila de tamaño "k" y un conjunto de "n" elementos con stock ilimitado. Cada elemento tiene un peso y un costo. Queremos seleccionar el subconjunto de elementos que maximice la ganancia de la mochila sin superar su capacidad. Solucione el problema utilizando programación dinámica.

## Solución

Algoritmo:

Porque es PD:

Complejidad:

Ejemplo:

Pseudocódigo:

21-

Recordemos el problema de selección de intervalos. Partimos de un recurso y un conjunto de intervalos. Cada intervalo cuenta con una fecha de inicio y otra de finalización. Queremos maximizar la cantidad de intervalos a seleccionar siempre que los mismos no se superpongan entre sí. Conocemos un algoritmo greedy que los resuelve de forma óptima. Queremos una variante donde la respuesta al problema corresponda a maximizar el tiempo de ocupación del recurso. Proponer un algoritmo utilizando programación dinámica que lo solucione.

## Solución

Algoritmo:

**Porque es PD:**

**Complejidad:**

**Ejemplo:**

**Pseudocodigo:**

## **24-**

El dueño de una cosechadora está teniendo una demanda muy elevada en los próximos 7 meses. Desde "n" campos lo quieren contratar para que preste sus servicios.

Lamentablemente no puede hacer todos los contratos puesto que varios de ellos se superponen en sus tiempos. Cuenta con un listado de los pedidos donde para cada uno de ellos se consigna: fecha de inicio, fecha de finalización, monto a ganar por realizarlo. Su idea es seleccionar la mayor cantidad de trabajos posibles. Mostrarle que esta solución puede no ser la óptima. Proponer una solución utilizando programación dinámica que nos otorgue el resultado óptimo (que trabajos elegir y cuanto se puede ganar).

## **Solution**

**Algoritmo:**

**Porque es PD:**

**Complejidad:**

**Ejemplo:**

**Pseudocodigo:**

**Skipped 3, 4, 7, 10, 13, 15, 20, 22, 23, 25**

# Brute Force & backtracking

1. possible solution
2. clasificacion
3. complexity
4. examples
5. code or pseudocode

exercices

## Examples (buch)

### K coloreo

```
def coloreo(graph, nodes, colors, idx, k):  
    if idx >= len(nodes):  
        return True, colors  
    v = nodes[idx]  
  
    for i in range(k):  
        colors[v] = i  
  
        if not compatible(graph, colors, v):  
            continue  
  
        if coloreo(graph, nodes, colors, idx + 1, k)[0]:  
            return True, colors  
  
    return False, colors  
  
def compatible(graph, colors, v):  
    for w in graph.adj(v):  
        if w in colors and colors[w] == colors[v]:  
            return False  
    return True
```

### Sudoku

```
def sudoku(cant_elem, M):  
  
    cant_elem = sig_pos_a_usar(cant_elem, M)  
  
    if cant_elem >= 9*9:  
        return True  
  
    for num in range(1, 10):
```

```

if puedo_poner(num, cant_elem, M):
    M[fila(cant_elem)][columna(cant_elem)] = num

if sudoku(cant_elem, M):
    return True

M[fila(cant_elem)][columna(cant_elem)] = 0

return False

```

se puede convertir el sudoku en un grafo con cada casilla un vertice y sus adyacentes son todos los de su columna, su fila y su celda. y luego se puede resolver el problema usando **k coloreo** con k = 9.

## Knight tour

```

def knight(paso = 0):
    if completo():
        return True # encontro sol
    x, y = obtener_pos()

    for fil, col in movimientos_knight(x, y):
        if not dentro_de_tablero(fil, col): continue
        if casillero_ya_marcado(fil, col): continue

        mover_a_pos(fil, col, paso)
        if (knight(paso + 1)):
            return True # encontro sol
        volver_a_pos(x, y) # no encontro sol, vuelve atras

    return False

```

Este problema si se modela en un grafo, la solucion es encontrar el camino hamiltoniano.

## Ejs

1-

Contamos con un conjunto de "n" puntos (x,y) en el plano cartesiano. Un par de puntos es el más cercano si la distancia euclidianas entre ellos es menor a la de cualquier otro par.

Resuelva el problema mediante un algoritmo naive que nos informe cuales son los 3 pares de puntos más cercanos.

## Solution

**Algoritmo:**

El algoritmo propuesto es uno que primeramente crea un array de soluciones parciales donde van a estar el par p1, p2 donde p2 es punto mas cercano a p1 para todo p1 en los puntos existentes. El algoritmo recorrera todos los puntos y por cada uno checkeara la

distancia con todos los otros puntos que no son el mismo y el que minimize esta distancia sera el companero de par. Una vez terminadas las iteraciones tendremos la lista de donde simplemente agarramos los 3 que tengan la minima distancia.

(hay que tener cuidado ya que la lista contiene los pares p1, p2 y p2, p1 osea estan repetidos)

### Clasificacion:

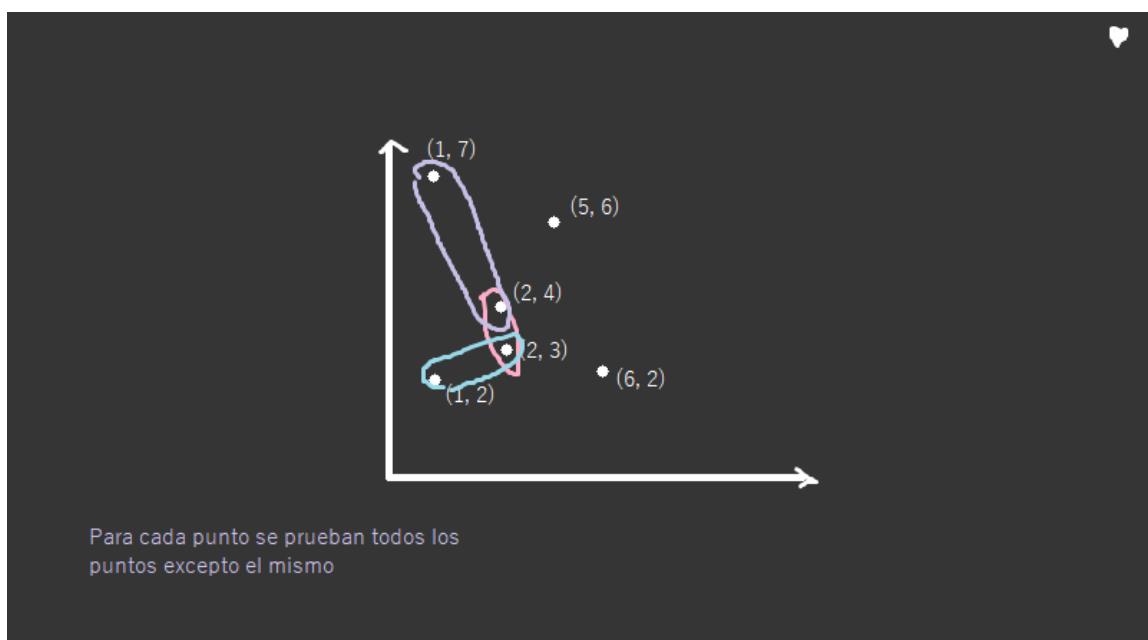
Este algoritmo utiliza fuerza bruta ya que ve todas las posibles opciones ya sean optimas o no, y cuando encuentra una opcion mejor que la que tenia la guarda garantizando que cuando termine la respuesta es si o si la optima.

### Complejidad:

La complejidad temporal es **O(n^2)** siendo n la cantidad de puntos, ya que por cada punto visto vemos denuevo todos los puntos.

La complejidad espacial es **O(n)** ya que se guardan para cada punto i solamente su par pi, px donde px es su mas cercano.

### Ejemplo:



In [ ]: # Code

```
def closest_points(points):
    partial_sols = []

    for i in range(len(points)):
        closest = None
        for j in range(len(points)):
            if i == j:
                continue
            if not closest or distance(points[i], points[j]) < distance(closest[0],
```

```

        closest = (points[i], points[j])
        partial_sols.append(closest)

    return sorted(partial_sols, key=lambda x: distance(x[0], x[1]) )

def distance(pt1, pt2):
    diff = (pt1[0] - pt2[0], pt1[1] - pt2[1])
    return diff[0]**2 + diff[1]**2

pts = [(1, 7), (5, 6), (2, 4), (2, 3), (6, 2), (1, 2)]
print("son todos los puntos ordenados por cercania pq me dio page\nhay que agarrar
son todos los puntos ordenados por cercania pq me dio page
hay que agarrar los 3 mas cercanos sin repetir pares
[((2, 4), (2, 3)), ((2, 3), (2, 4)), ((1, 2), (2, 3)), ((1, 7), (2, 4)), ((5, 6),
(2, 4)), ((6, 2), (5, 6))]
```

### 3-

Se encuentran en un río 3 caníbales y 3 vegetarianos. En la orilla hay un bote que permite pasar a dos personas atravesarlo. Los 6 quieren cruzar al otro lado. Sin embargo existe un peligro para los vegetarianos: si en algún momento en alguna de las márgenes hay más caníbales que vegetarianos estos los atacarán. Tener en cuenta que el bote tiene que ser manejado por alguien para regresar a la orilla. Determinar si es posible establecer un orden de cruces en el que puedan lograr su objetivo conservando la integridad física. Explicar cómo construir el grafo de estados del problema. Determinar cómo explorarlo para conseguir la respuesta al problema. Brinde, si existe, la respuesta al problema.

### Solution

#### Algoritmo:

Un algoritmo para construir el arbol de estados para este problema puede ser uno que simule todos los posibles escenarios empezando desde el principio cuando hay 3c, 3v con bote en isla 1 y 0c, 0v sin bote en isla 2.

Si se llega a un escenario que ya fue probado en esa misma rama del arbol se evita analizar la misma ya que desembocaría en un loop infinito, tambien cuando se detecta una incompatibilidad se abandona ese camino y se vuelve un paso atras para probar los otros.

Cuando las 6 personas esten en la isla 2 se llego a la solucion, pero si termina el algoritmo y eso no se encontro entonces no tiene solucion.

Inspeccionando el ejemplo de *sev* que esta abajo vemos que el arbol siempre termina en Game Over asi se ve que este problema no tendra solucion.

#### Clasificacion:

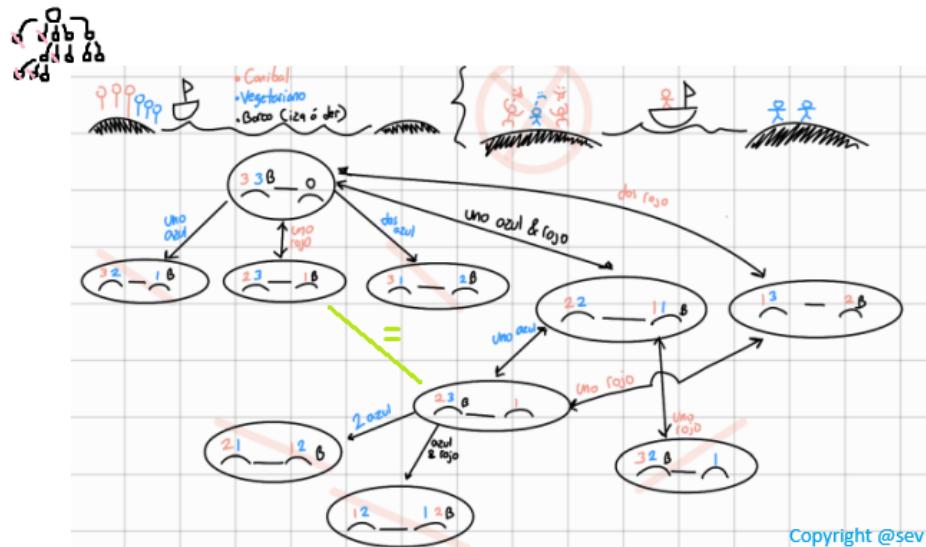
Este algoritmo utilizaria backtracking ya que una vez que un camino es incompatible se poda el arbol de estados y no se inspecciona el camino.

## Complejidad:

La complejidad temporal y espacial sera la misma ya que hay que almacenar y calcular el arbol de estados, y esa complejidad sera **O(n!)** no se pq algo me dice  $n!$  pero fruta

## Ejemplo:

- Copyright - sev (resumen de teoria de algoritmos I)



## Pseudocodigo:

### 4-

Resuelva el problema de las reinas en el tablero de ajedrez mediante backtracking planteado como permutaciones. Brinde el pseudocódigo y determine la cantidad máxima posible de subproblemas a explorar.

## Solution

### Algoritmo:

El algoritmo propuesto es uno que debe poner solo 1 reina por columna, fila y diagonales. Para esto lo que hara es checkear todas las posibles combinaciones de cuando la primera reina esta en cada cuadradito, con la excepcion de que si llegamos a un momento en el cual la solucion ya no es compatible abandonamos ese camino y asi podamos el arbol de estados. Dicho arbol sera que para cada cuadradito tendremos la oportunidad de poner o no poner reina ahí, y el arbol se convierte en un arbol binario de tamaño  $2^n$  como se ve en el ejemplo dado abajo.

Entonces el algoritmo por cada iteracion de cada cuadradito llamara a la version recursiva del algoritmo donde el caso base sera cuando el vertice esta fuera de la matriz y retornara false, luego si ya cubrimos todo el tablero y es compatible retornamos true, de lo contrario

false, tambien si ya no es compatible aunque todavia no se haya cubierto el tablero retornamos false. Luego de los checkeos ubicamos la reina en la posicion actual y llamamos recursivamente para ver si con la reina ahí es solucion si lo es retornamos true, si no lo es desubicamos a la reina puesta y retornamos false para caer uno en la recursividad.

Tambien este problema se puede modelar en un grafo y usar el algoritmo de buscar el **independent set!**

**Clasificacion:**

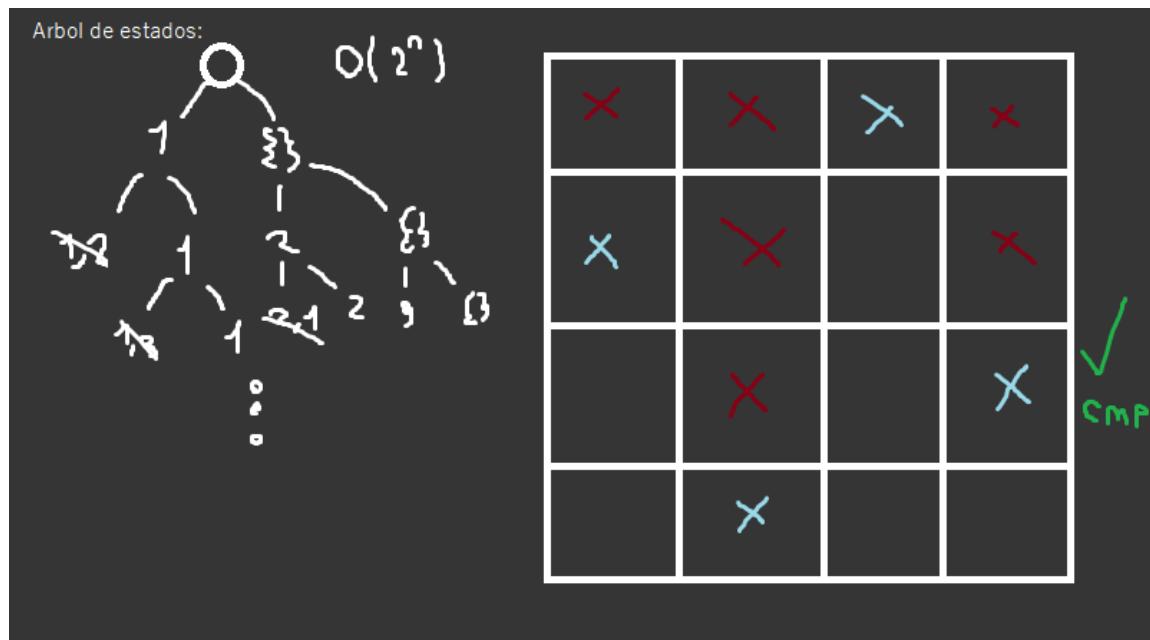
Esto es backtracking ya que con la condición de checkear si es compatible hasta un punto y si no lo es cortamos esa rama estariamos podando el arbol de estados. Osea si no tiene sentido seguir por un camino porque ya esta mal, lo abandonamos y volvemos para atras para probar el otro camino.

**Complejidad:**

La complejidad temporal es  **$O(2^n)$**

La complejidad espacial es  **$O(n)$**  porque el set que se crea contiene como maximo n, y no se crea ninguna otra variable ni nada. *NO estoy seguro de esto ya que el stack puede ser que sea  $2^n$  en espacio pero no lo se, chatgipiri tmb dijo lineal*

**Ejemplo:**



**Pseudocódigo:**

```
def ubicar_reina(tablero):
    ubicados = set()
    for v in tablero: # hacemos la recursiva para cada vertice hasta que encontremos el que anda
        if _ubicar_reina(tablero, v, ubicados):
            return True
```

```

    return False

def _ubicar_reina(tablero, vertice_actual, ubicados):
    if vertice_actual fuera de la matriz:
        return False
    if len(ubicados) == len(tablero):
        return es_compatible(tablero, ubicados)

    if not es_compatible(tablero, ubicados) or los puestos + los que me
quedan < los que tengo que poner:
        # este es el backtracking
        return False

    ubicados.add(vertice_actual) # Lo pongo
    if _ubicar_reina(tablero, proximo vertice, ubicados):
        return True
    ubicados.remove(vertice_actual) # no era por aca, asi que lo saco
    return False

```

## 5-

En un tablero de ajedrez (una cuadrícula de 8x8) se ubica la pieza llamada "caballo" en la esquina superior izquierda. Un caballo tiene una manera peculiar de moverse por el tablero: Dos casillas en dirección horizontal o vertical y después una casilla más en ángulo recto (formando una forma similar a la letra "L"). El caballo se traslada de la casilla inicial a la final sin tocar las intermedias, dado que las "salta". Se quiere determinar si es posible, mover esta pieza de forma sucesiva a través de todas las casillas del tablero, pasando una sola vez por cada una de ellas, y terminando en la casilla inicial. Plantear la solución mediante backtracking.

### Solution

#### Algoritmo:

El algoritmo que resuelve el problema puede ser un algoritmo recursivo que empezando desde el paso 0 va a recrear si es posible el camino desde una dada posición inicial, caso base será si esta completo el tablero, en cuyo caso retorna true, luego se ve uno por uno todos las posiciones a las que se podría mover el caballo desde su posición actual, por cada una de estas checkeamos que siga en el tablero y si esa pos esta ocupada, en cualquiera de estos dos casos continuamos con ver la siguiente posible posición, cuando encontramos una donde se puede poner lo ponemos, luego llamamos recursivamente a esta función y si devuelve true quiere decir que este camino es valido y devuelve true, de lo contrario desmovemos el caballo y al seguimos iterando las demás posiciones hasta terminar, si termina de iterar sale del loop y devuelve false ya que no hubo camino posible.

Tambien se puede modelar este problema como un grafo y la solución sería encontrar su **camino hamiltoniano** es decir el que pasa por todos los nodos una sola vez.

## Clasificación:

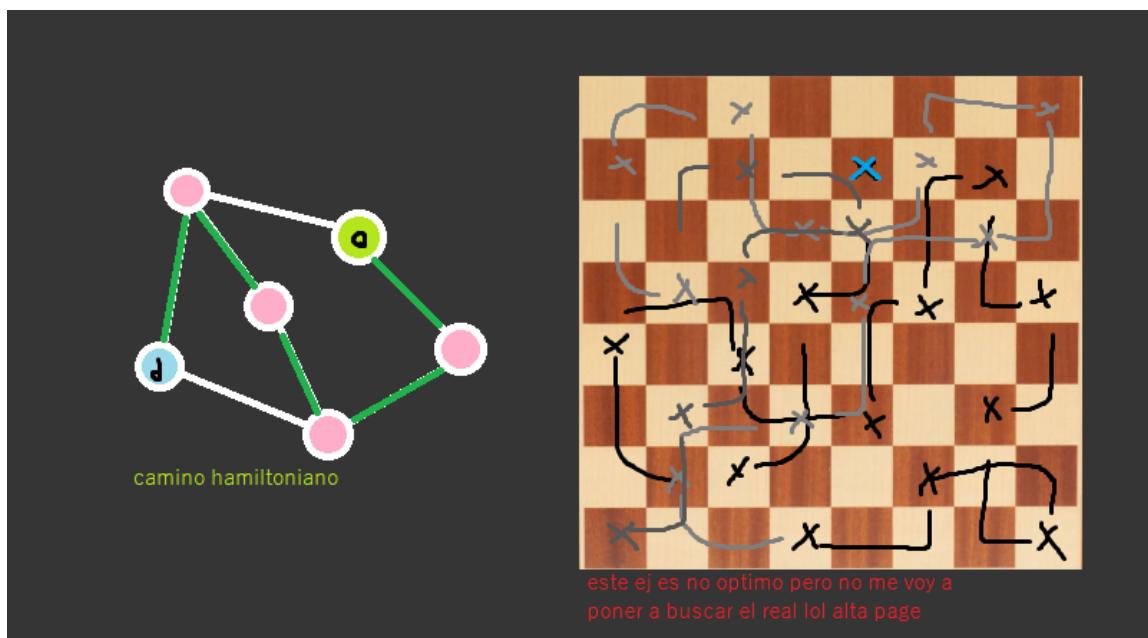
Uso backtracking ya que no se calculan absolutamente todos los caminos, sino que abandona caminos donde detecta que ya paso por ellos y volver a pasar estaria mal.

## Complejidad:

La complejidad temporal de este algoritmo es  $O(2^n)$

La complejidad espacial *creo que* tambien es  $O(2^n)$  ya que por cada llamada recursiva se crea la variable de posicion actual.

## Ejemplo:



## Pseudocódigo:

Ya estaba en los ejemplos pero lo hago denuevo atsssss

```
def camino_de_caballo(paso=0):
    if completo():
        return True

    caballo_pos = caballo_pos()

    for x, y in movimientos_pos():

        if caballo_pos is afuera del tablero:
            continue
        if pos_ocupada(paso):
            continue

        mover caballo a x,y # ver si es por aca
        if camino_de_caballo(paso+1):
            return True
        desmover caballo de x,y # no es, volve para atras
```

```
return False # no hay posible camino
```

## 7-

Se cuenta con "n" trabajos por realizar y la misma cantidad de contratistas para realizarlos. Ellos son capaces de realizar cualquiera de los trabajos aunque una vez que se comprometen a hacer uno, no podrán realizar el resto. Tenemos un presupuesto de cada trabajo por cada contratista. Queremos asignarlos de forma tal de minimizar el costo del trabajo total. Proponer un algoritmo por branch and bound que resuelva el problema de la asignación.

### Solution

#### Algoritmo:

Voy a resolver este ejercicio usando fuerza bruta no branch and bound ya que no lo dieron en *curso verano 2024 buchwald*. No puedo usar backtracking ya que no hay un momento en el que podemos abandonar un camino y podar el arbol de estados.

El algoritmo propuesto es uno que calculara el costo de todos los trabajos al empezar por un dado nodo, y luego el proximo y asi hasta n, luego cada vez que encuentre un camino con costo menor al que se tenia este sera reemplazado.

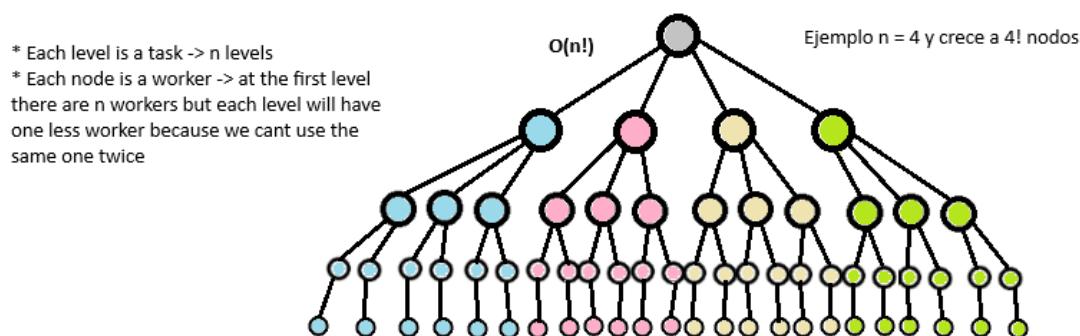
#### Clasificacion:

Es fuerza bruta ya que vamos a calcular todas las posibles escenarios y elegir el de menor costo.

#### Complejidad:

La complejidad temporal de crear el arbol de estados es **O(n!)** y espacial nidea.

#### Ejemplo:



Pseudocodigo:

```
def min_costo(trabajos, trabajadores):
    min = inf
    por cada trabajador:
        ver_min(min, trabajos, trabajadores, trabajo)
    return min

def ver_min(min, suma_parcial, trabajos, trabajadores, trabajo,
trabajador):
    si trabajo es el ultimo y trabajador + suma_parcial < min:
        min = trabajador + suma_parcial
    return
    si trabajo es el ultimo:
        return

    suma_parcial += trabajador
    ver_min(min, suma_parcial, trabajos, trabajadores - trabajador,
trabajo+1, trabajador+1)
    suma_parcial -= trabajador
```

## 8-

Contamos con un conjunto de "n" de equipamientos que se deben repartir entre un conjunto de "m" equipos de desarrollo. Cada equipo solicita un subconjunto de ellas. Puede ocurrir que un mismo equipamiento lo soliciten varios equipos o incluso que un equipamiento no lo solicite nadie. Queremos determinar si es posible seleccionar un subconjunto de equipos de desarrollo entregándoles a todos ellos todo lo que soliciten y al mismo tiempo que ninguno de los equipamientos quede sin repartir. Resolver el problema mediante backtracking.

### Solution

#### Algoritmo:

El algoritmo propuesto sera un algoritmo que por cada equipo que vea tendra la opcion de tomarlo o no tomarlo, en principio se lo toma y se llama recursivamente a la funcion, si esta devuelve false entonces vamos a ver que pasaba si no lo tomabamos, y asi sucesivamente hasta que alguno devuelva true y si es el caso entonces devolvemos true porque encontramos solucion, si terminamos de ver todos los casos y ninguno es solucion retornamos false.

La condicion de poda sera cuando el camino ya no es compatible o cuando se llego al final del camino y no tenemos todos los equipamientos en uso.

#### Clasificacion:

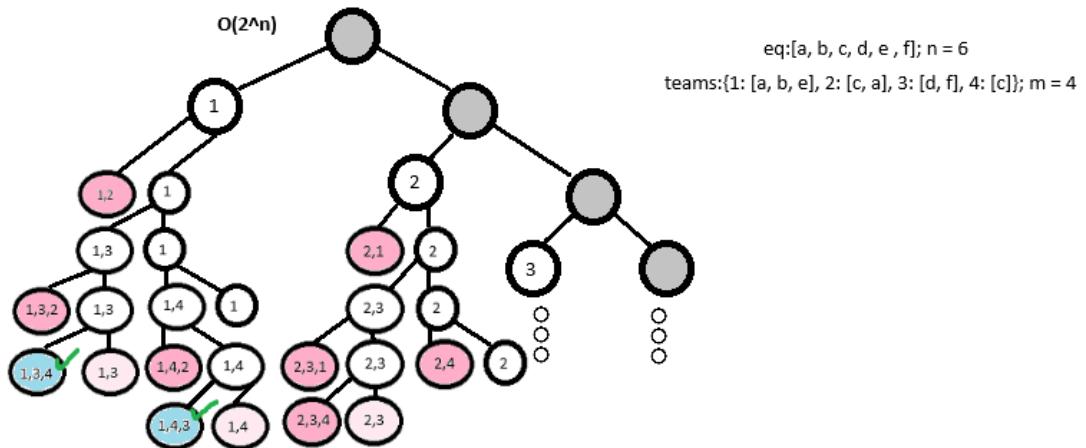
Se usa backtracking ya que se ven todas las opciones posibles pero cuando se descubre que un camino ya no va a ser solucion se abandona y no se lo explora.

Complejidad:

La complejidad temporal sera **O(2^n)**

La complejidad espacial es **O(n)** por el set de seleccionados.

Ejemplo:



Pseudocodigo:

```
def _camino_equipos(equipo, equipamiento, seleccionados):
    if nivel >= len(equipamiento):
        return es_compatible(seleccionados, equipamiento)

    if not es_compatible(seleccionados, equipamiento):
        return False

    seleccionados.add(equipo)
    if _camino_equipos(prox equipo, equipamiento, seleccionados):
        return True
    seleccionados.remove(equipo)

    return False
```

10-

Contamos con un conjunto de "n" actividades entre las que se puede optar por realizar. Cada actividad x tiene una fecha de inicio  $I_x$ , una fecha de finalización  $F_x$  y un valor  $V_x$  que obtenemos por realizarla. Queremos seleccionar el subconjunto de actividades compatibles entre sí que maximice la ganancia a obtener (suma de los valores del subconjunto). Proponer un algoritmo por branch and bound que resuelva el problema.

Solution

**Algoritmo:**

**Clasificacion:**

**Complejidad:**

**Ejemplo:**

**Pseudocodigo:**

## 11-

Se cuenta con "n" servidores especializados en renderización de videos para películas animadas en 3d. Los servidores son exactamente iguales. Además contamos con "m" escenas de películas que se desean procesar. Cada escena tiene una duración determinada. Queremos determinar la asignación de las escenas a los servidores de modo tal de minimizar el tiempo a esperar hasta que la última de las escenas termine de procesarse. Determinar dos metodologías con la que pueda resolver el problema y presente como realizar el proceso.

### Solution

**Algoritmo:**

**Clasificacion:**

**Complejidad:**

**Ejemplo:**

**Pseudocodigo:**

## 12-

Un granjero debe trasladar un lobo, una cabra y una zanahoria a la otra margen de un río. Cuenta con un bote donde solo entra él y un elemento más. El problema es que no puede quedar solo el lobo y la cabra. Dado que la primera se comería a la segunda. De igual manera, tampoco puede dejar solo a la cabra y la zanahoria. La primera no dudaría en comerse a la segunda. ¿Cómo puede hacer para cruzar? Explicar cómo construir el grafo de estados del problema. Determinar cómo explorarlo para conseguir la respuesta al problema. Brinde, si existe, la respuesta al problema.

### Solution

**Algoritmo:**

**Clasificacion:**

**Complejidad:**

**Ejemplo:**

Pseudocodigo:

**Skipped 2, 6, 9**

# Flow network

1. Explanation
2. Drawing
3. Complexity
4. Network creation pseudocode
5. Network function pseudocode

[exercises](#)

## Examples (buch)

### Esquema Ford-Fulkerson

```
def flow(graph, s, t):  
    init flow through every edge to 0  
    while there is an augmenting path in the residual network:  
        increase flow according to that path
```

### Ejemplo codigo Ford-Fulkerson

```
def flow(graph, s, t):  
    flow = {}  
    for v in graph:  
        for w in graph.adj(v):  
            flow[(v, w)] = 0  
  
    residual = copy(graph)  
  
    while (path = obtener_camino(residual, s, t)) is not None:  
        residual_flow = min_weight_of_all_edges(graph, path)  
        for i in range(1, len(path)):  
            if graph.has_edge(path[i-1], path[i]):  
                flow[(path[i-1], path[i])] += residual_flow  
                update_residual_graph(residual, path[i-1], path[i],  
residual_flow)  
            else:  
                flow[(path[i], path[i-1])] -= residual_flow  
                update_residual_graph(residual, path[i], path[i-1],  
residual_flow)  
  
    return flow  
  
def update_residual_graph(graph, u, v, flow):  
    last_weight = graph.weight_of_edge(u, v)
```

```

if last_weight == flow:
    graph.remove_edge(u, v)
else:
    graph.update_edge(u, v, last_weight - flow)

if not graph.has_edge(v, u):
    graph.add_edge(v, u, flow)
else:
    graph.update_edge(v, u, graph.weight_of_edge(v, u) + flow)

```

Complejidad:

La complejidad depende de como buscamos camino, si se usa el algoritmo de **Edmonds-Karp** osea el que usa BFS la complejidad es **O(V\*E^2)**

**O(E\*V)** si es bipartito

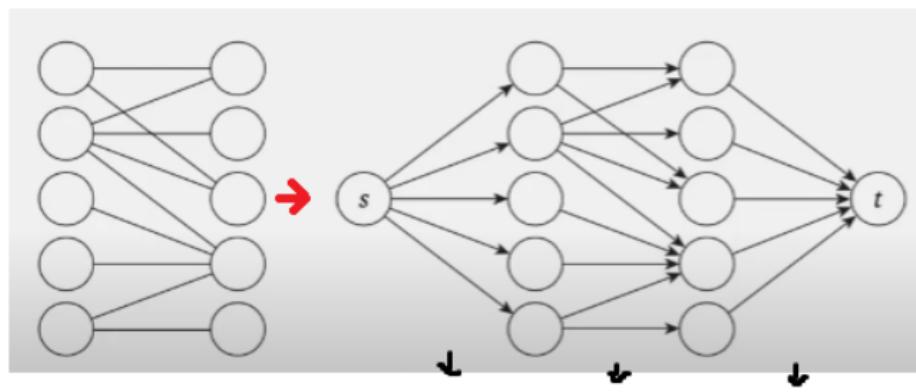
Tambien puede pasar que demore muchisimo o nunca converger **O( $\infty$ )** xd.

## Creacion de grafo residual

- Hay mas de una fuente  $\rightarrow$  superfuente
- Hay mas de un sumidero  $\rightarrow$  supersumidero
- Hay bucles (arista  $v, v$ )  $\rightarrow$  las saco
- Hay ciclos de 2 nodos (aristas  $v, w$  y  $w, v$ )  $\rightarrow$  creo un nodo intermedio imaginario (se forma un triangulo)

## Bipartite Matching

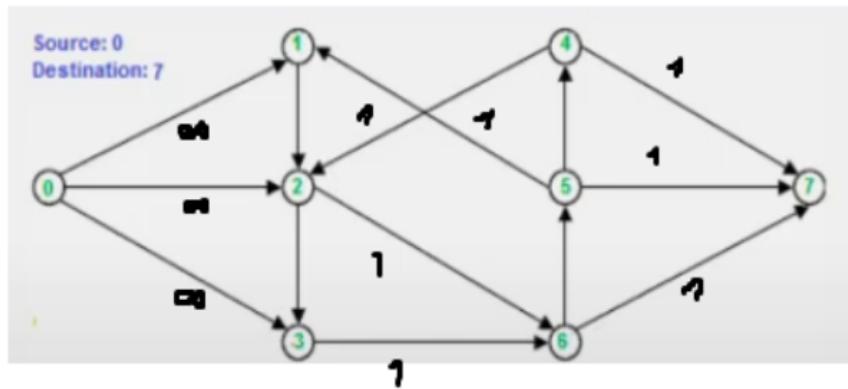
Se hace direccionaldo de una componente a otra componente, se agrega fuente y sumidero, todas las aristas en 1 y despues se aplica ford-fulkerson y donezo



Todos pesos 1 pq no se pueden repetir los vertices

## Disjoint paths

- En dirigido se pone fuente y sumidero con aristas entre fuente y nodos con grado de entrada 0 (sin contar las aristas desde la fuente) el peso va a ser infinito, y el resto de los pesos sera 1, eso garantiza no reutilizar aristas.
- En no dirigido se hace lo mismo con la salvedad de que cuando se hace el BFS encontrando camino se tiene que tener en cuenta que siempre el nivel tiene que ir en aumento, porque si encontramos camino en el que en un momento el nivel disminuye significaría que estamos usando una arista para volver para atrás (y se usaria 2 veces)



## Airline scheduling

Suponer que queremos schedulear cómo los aviones van de un aeropuerto a otro para cumplir horarios y demás. Podemos decir que podemos usar un avión para un segmento/vuelo i y luego para otro j si:

- El destino de i y el origen de j son el mismo.
- Podemos agregar un vuelo desde el destino de i al origen de j con tiempo suficiente.

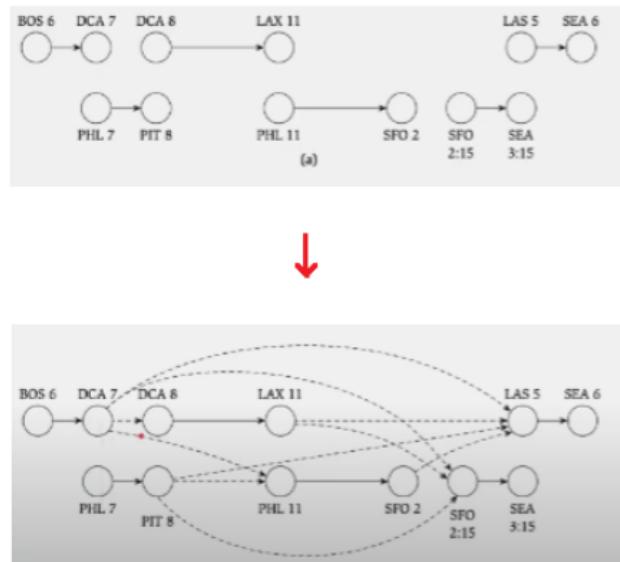
Decimos que el vuelo j es alcanzable desde el vuelo i si es posible usar el avión del vuelo i y después para el vuelo j.

Problema: ¿Podemos cumplir con los m vuelos usando a lo sumo k aviones? -> [VID BUCH](#)

## SOLUCION

1. Nuestras unidades de flujo son literalmente los aviones.
2. Ponemos las aristas de los vuelos que queremos si o si cumplir con cota mínima y capacidad = 1 (para forzar que se usen).

3. Si otro vuelo es alcanzable por las reglas anteriores, ponemos otra arista de capacidad 1.
4. Ponemos una fuente con aristas de capacidad 1 a los orígenes.
5. Ponemos un sumidero con aristas de capacidad 1 desde los destinos.
6. La fuente tiene demanda  $-K$  y el sumidero  $K$  (es decir habrá una superfuente dando flujo  $K$  a nuestra fuente, esto pq sino puede llegar a usar mas de los aviones que tiene la empresa).



\* La fuente esta conectada a todos los orgíenes

\* Todos los destinos estan conectados al sumidero

sólidos: capacidad mínima y máxima 1

punteados: capacidad mínima 0 capacidad máxima 1

## Ejs

### 1-

La sala de guardia de un hospital tiene que tener al menos un médico en todos los feriados y en los fines de semana largos de feriados. Cada profesional indica sus posibilidades: por ejemplo alguien puede estar de guardia en cualquier momento del fin de semana largo del 9 de julio (p. ej. disponibilidad de A para el 9 de julio = (Jueves 9/7, Viernes 10/7, Sábado 11/7, Domingo 12/7)), también puede suceder que alguien pueda sólo en parte (por ejemplo, disponibilidad de B para 9 de julio = (Jueves 9/7, Sábado 11/7, Domingo 12/7)). Aunque los profesionales tengan múltiples posibilidades, a cada uno se lo puede convocar para un solo día (se puede disponer de B sólo en uno de los tres días que indicó). Para ayudar a la sala de guardia a planificar cómo se cubren los feriados durante todo el año debemos resolver el problema de las guardias: Existen  $k$  períodos de feriados (por ejemplo, 9 de julio es un período de jueves 9/7 a domingo 12/7, en 2019 Día del Trabajador fue un período de 1 día: miércoles 1 de mayo, etc.).  $D_j$  es el conjunto de fechas que se incluyen en el período de feriado  $j$ -ésimo. Todos los días feriados son los que resultan de la unión de todos los  $D_j$ . Hay  $n$  médicos y cada médico  $i$  tiene asociado un conjunto  $S_i$  de días feriados en los que puede

trabajar (por ejemplo B tiene asociado los días Jueves 9/7, Sábado 11/7, Domingo 12/7, entre otros).

Proponer un algoritmo polinomial (usando flujo en redes) que toma esta información y devuelve qué profesional se asigna a cada día feriado (o informa que no es posible resolver el problema) sujeto a las restricciones:

- Ningún profesional trabajará más de  $F$  días feriados ( $F$  es un dato), y sólo en días en los que haya informado su disponibilidad.
- A ningún profesional se le asignará más de un feriado dentro de cada período  $D_j$ .

## Solution

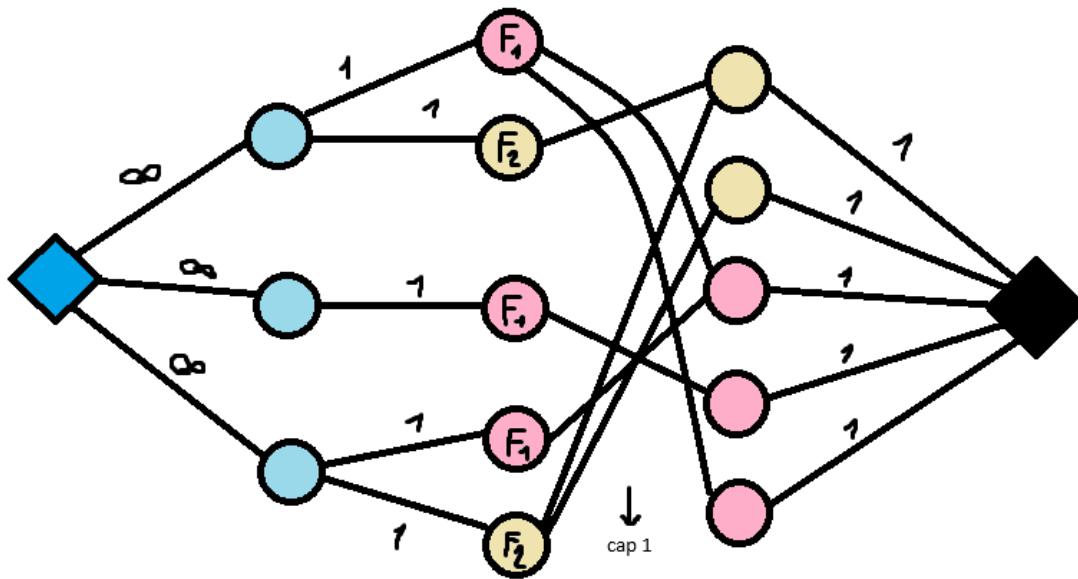
- Con ayuda de [Flor jeje](#)

### Algoritmo:

Se crea la red de flujo donde los doctores estaran conectados a un superfuente que les dara flujo infinito ya que estos pueden trabajar en mas de un feriado, pero como no pueden trabajar 2 veces en el mismo feriado estos estaran conectados a unos nodos intermediarios dependiendo de que feriados pueden trabajar, y cada uno de estos nodos estaran conectados los dias de ese feriado que el doctor puede trabajar, la arista entre doctor feriado tiene capacidad 1 porque el doctor no puede trabajar mas de una vez en el mismo y las aristas entre nodo feriado a dia de ese feriado tambien tendran capacidad 1 ya que si un dia ya esta cubierto no hace falta que otro doctor lo cubra, luego todos esos dias estaran conectados a un supersumidero con aristas de capacidad 1 por la misma razon anterior.

Una vez se tiene esta red de flujo se le calcula el flujo maximo usando un algoritmo como el de ford-fulkerson (o la variante edmond-karp) y si este flujo coincide con el numero de dias de feriados, significa que se logro cubrir todos, de lo contrario hay algun dia que no pudo ser cubierto.

### Ejemplo:



**Complejidad:**

siendo V los nodos de la red de flujo y E las aristas de la misma:

La complejidad temporal es **O(V\*E^2)** por edmond-karp

La complejidad espacial es **O(V+E)** por la creacion del grafo residual que copia el grafo de red de flujo.

**Pseudocodigo:**

```
def cubre_los_dias(red, s, t):
    init flujo de cada arista a 0
    crear grafo residual a partir de la red

    mientras que haya un camino p entre s y t: # p el camino mas corto
        flujo residual = el minimo peso de todas las aristas del camino p
        si la red tiene la arista:
            flujo de la arista += flujo residual
        sino: # se usa arista antiparalela
            flujo de la arista -= flujo residual
        actualizar grafo residual

    devolver flujo == cantidad de feriados
```

**2-**

Definimos el Problema de la Evacuación de la siguiente manera: Se tiene un grafo dirigido  $G = (V, E)$  que describe una red de caminos. Tenemos una colección de nodos  $X \subset V$  que son los nodos poblados (ciudades) y otra colección de nodos  $S \subset V$  que son los nodos de refugio (supondremos que  $X$  y  $S$  son disjuntos). En caso de una emergencia queremos poder

definir un conjunto de rutas de evacuación de los nodos poblados a los refugios. Un conjunto de rutas de evacuación es un conjunto de caminos en G tales que

- cada nodo en X es el origen de un camino
- el último nodo de cada camino es un refugio (está en S)
- los caminos no comparten aristas entre sí. Se pide: dados G, X y S, mostrar cómo se puede decidir en tiempo polinomial si es posible construir un conjunto de rutas de evacuación (usar flujo en redes para eso. Construir la red adecuada).

## Solution

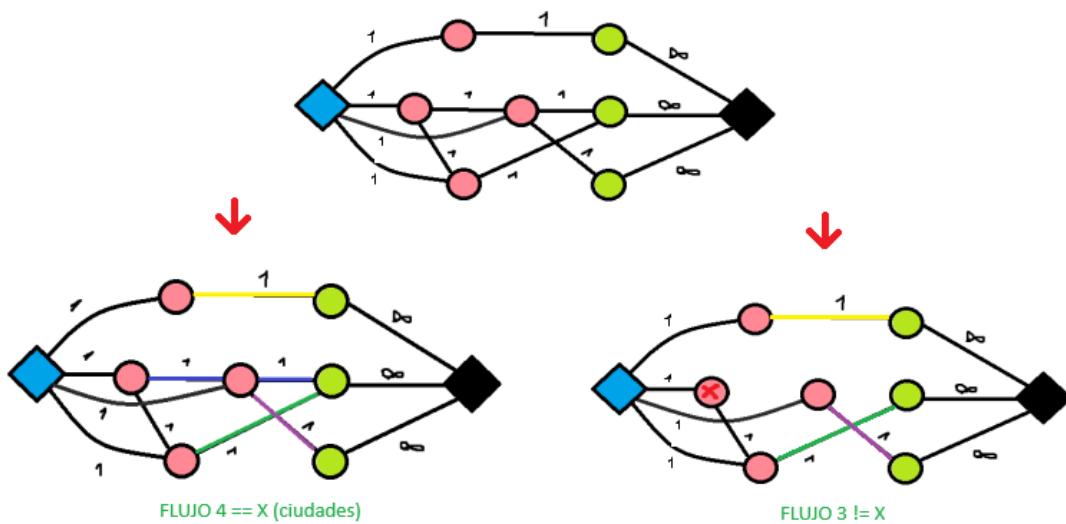
- Con ayuda de [Flor jeje](#)

### Algoritmo:

Se crea una red de flujo donde hay una superfuente que da flujo a todas las ciudades en X con capacidad 1 ya que cada ciudad solamente debería usar un camino y no más, cada ciudad tiene su camino hacia un refugio ya sea que ese camino pase o no por otras ciudades, todas las capacidades intermedias entre aristas de ciudad a ciudad o de ciudad a refugio tendrán capacidad 1, luego todos los refugios serán unidos al supersumidero con capacidad infinita ya que a cada refugio le llegará un número desconocido de ciudades.

Se busca el flujo máximo a través de la red con un algoritmo como ford-fulkerson (usando la variante de edmond-karp) y se ve si el flujo obtenido es igual a la cantidad de ciudades X, si lo es entonces hay camino para evacuar todas las ciudades, de lo contrario significa que hay ciudades que no pueden ser evacuadas sin exceder la capacidad de las rutas.

### Ejemplo:



### Complejidad:

La complejidad temporal será  $O(V^*E^2)$  por edmond-karp

La complejidad espacial sera **O(V+E)** porque se crea una copia del grafo dado para crear el grafo residual.

Pseudocodigo:

```
def se_puede_evacuar(rutas, s, t):
    init flujo para cada arista en 0
    crear grafo residual

    mientras que haya camino de menor longitud entre s y t:
        flujo residual = min(todas las aristas del camino)
        para cada arista en el camino:
            si el grafo rutas tiene la arista:
                flujo de la arista += flujo residual
            sino:
                flujo de la arista -= flujo residual
            actualizar grafo residual

    devolver flujo == X
```

### 3-

Para un evento solidario un conjunto de n personas se ofrecieron a colaborar. En el evento hay m tareas a desarrollar. Cada tarea tiene un cupo máximo de personas que la puede realizar. A su vez cada persona tiene ciertas habilidades que la hacen adecuadas para un subconjunto de tareas. Proponga una solución mediante red de flujos que maximice la cantidad de personas asignadas a las tareas. ¿Hay forma de lograr asegurarnos un piso mínimo de personas en cada tarea? ¿Cómo impacta en la solución presentada en el punto anterior?

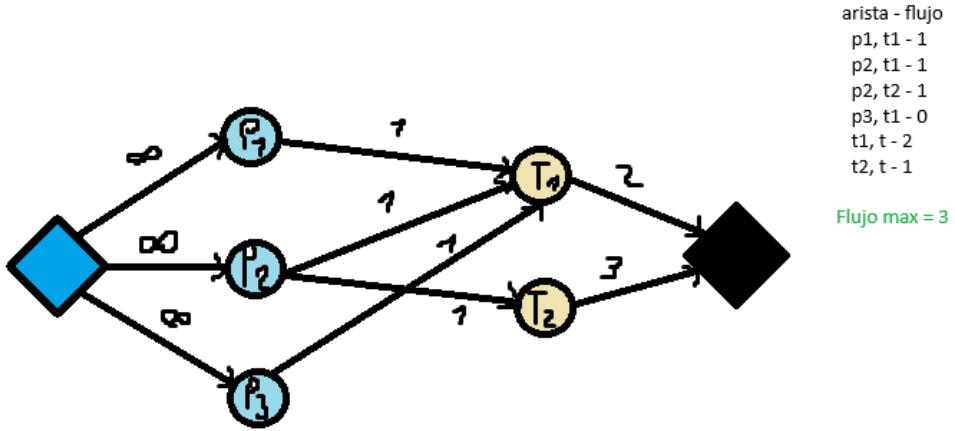
### Solution

Algoritmo:

Se crea la red de flujo como un grafo dirigido donde habra multiples fuentes que representaran a las n personas por lo que se las une con una superfuente que les dara flujo infinito ya que una misma persona podra ser tenida en cuenta para mas de 1 tarea, las m tareas seran los multiples sumideros que tendran que ser conectados a un supersumidero con la capacidad de personas que pueden hacer esa tarea. Luego cada persona tendra una arista hacia las tareas para las que es adecuada y el peso de la misma sera 1, ya que el flujo en este caso representara las personas que hacen tareas, asi que 1 persona da 1 de flujo a una tarea especifica (o mas de una).

Una vez que se tiene esta red de flujo se le calcula el flujo maximo a travez de un algoritmo como ford-fulkerson, usando la variante de edmond-karp y esto nos maximizara las personas asignadas a cada tarea.

Ejemplo:



### Complejidad:

La complejidad temporal sera la de el algoritmo de edmond-karp y es **O(V\*E^2)**

La complejidad espacial sera **O(V+E)** ya que se necesita copiar el grafo original y agregar la superfuente y el supersumidero asi como las aristas correspondientes. Y tambien se guardara el flujo en un diccionario de largo E (aristas del nuevo grafo).

### Pseudocodigo:

```

def create_flow_network(graph):
    flow_network = new Graph with nodes graph.all_nodes()
    create nodes source and sink
    for node in graph:
        if node is person:
            create_edge(source, person, infinity)
            for task in graph.adj(person):
                create_edge(person, task, 1)
        if node is task:
            create_edge(task, sink, task capacity)
    return flow_network

def max_flow(graph, s, t):
    flow = {}
    for v in graph:
        for w in graph.adj(v):
            flow[(v,w)] = 0

    residual = create_residual_graph(graph) # doing everything it means

    while (path = obtener_camino_menor_longitud(residual, s, t)) is not
None:
        residual_flow = min_weight_of_all_edges(graph, path) #
min(path.edges)
    
```

```

        for i in range(1, len(path)):
            if graph.has_edge(path[i-1], path[i]):
                flow[(path[i-1], path[i])] += residual_flow
                update_residual_graph(residual, path[i-1], path[i],
                                      residual_flow)
            else: # using antiparallel edge
                flow[(path[i], path[i-1])] -= residual_flow
                update_residual_graph(residual, path[i], path[i-1],
                                      residual_flow)

    return flow

def update_residual_graph(graph, u, v, flow):
    last_weight = graph.weight_of_edge(u, v)

    if last_weight == flow:
        graph.remove_edge(u, v)
    else:
        graph.update_edge(u, v, last_weight - flow)

    if not graph.has_edge(v, u): # antiparallel edge
        graph.add_edge(v, u, flow)
    else:
        graph.update_edge(v, u, graph.weight_of_edge(v, u) + flow)

```

## 4-

La red de transporte intergaláctico es una de las maravillas del nuevo imperio terráqueo. Cada tramo de rutas galácticas tiene una capacidad infinita de transporte entre ciertos planetas. No obstante, por burocracia - que es algo que no los enorgullece - existen puestos de control en cada planeta que reduce cuantos naves espaciales pueden pasar por día por ella. Por una catástrofe en el planeta X, la tierra debe enviar la mayor cantidad posible de naves de ayuda. Por un arreglo, durante un día los planetas solo procesaran en los puestos de control aquellas naves enviadas para esta misión. Tenemos que determinar cuál es la cantidad máxima de naves que podemos enviar desde la tierra hasta el planeta X.

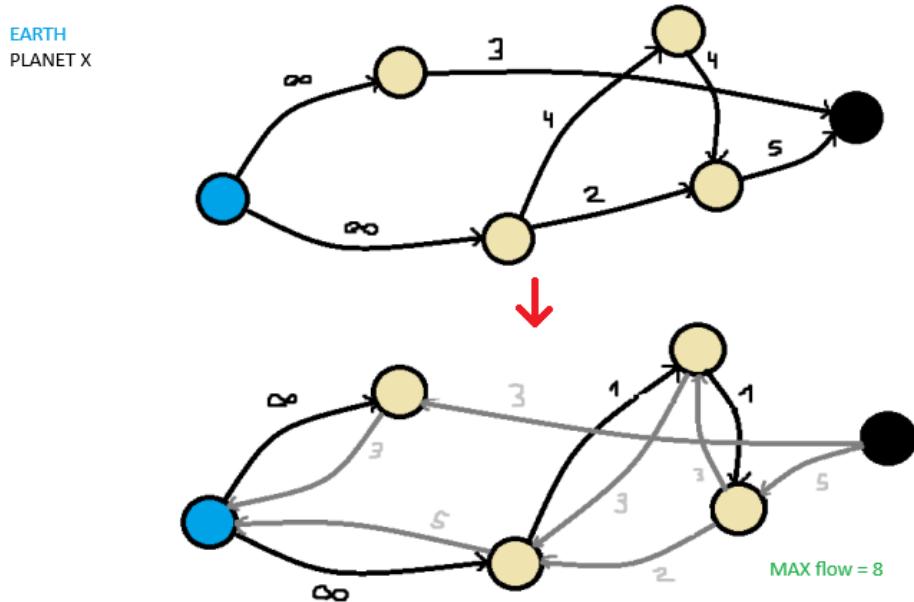
Sugerencia: considerar a este un problema de flujo con capacidad en nodos y no en ejes

## Solution

Algoritmo:

Se crea una red de flujo tal que la tierra es la fuente del flujo y el planeta X el sumidero y en el medio estaran los planetas intermedios entre los mismos. Teniendo en cuenta que en cada planeta hay un puesto de control donde se puede disminuir el flujo de naves, se calculara el flujo maximo a travez de la red utilizando un algoritmo ford-fulkerson buscando los caminos mas cortos primero, para eso se usara bfs (algoritmo edmond-karp). Una vez terminado sabremos hacia donde hay que mandar cada nave para maximizar las naves que llegan al planeta X.

Ejemplo:



Complejidad:

La complejidad temporal es **O(V\*E^2)** por edmond-karp

La complejidad espacial es **O(V+E)** porque se le hace una copia al grafo para crear el residual, y el flujo se guarda en un diccionario de largo E.

Pseudocódigo:

```
def max_ships(galactic_network, earth, planet_x):  
    flow = initialize dict with all galactic_network.edges as keys and 0  
    as value  
  
    residual = create_residual_graph(galactic_network)  
  
    while path = get_shortest_path(residual):  
        residual_flow = min of edges of the path # the weight  
  
        for i in range(1, len(path)):  
            if galactic_network.has_edge(path[i-1], path[i]):  
                flow[(path[i-1], path[i])] += residual_flow  
                update_residual_graph()  
            else:  
                flow[(path[i], path[i-1])] -= residual_flow  
                update_residual_graph()  
  
    return flow
```

5-

La compañía eléctrica de un país nos contrata para que le ayudemos a ver si su red de transporte desde su nueva generadora hidroeléctrica hasta su ciudad capital es robusta. Nos

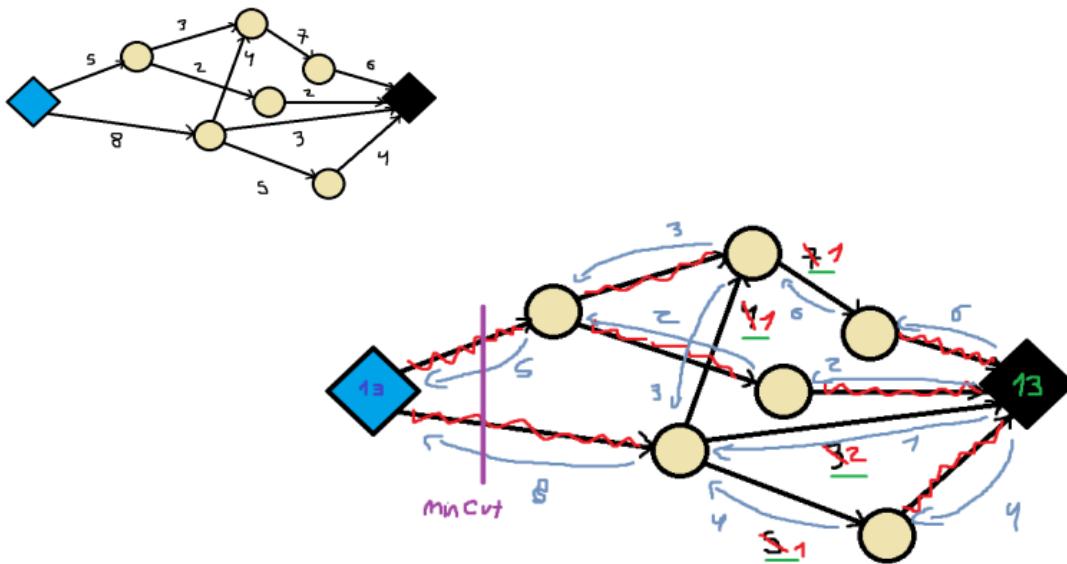
otorgan un plano con la red eléctrica completa: todas las subestaciones de distribución y red de cableados de alta tensión. Lo que quieren que le digamos es: cuantas secciones de su red se pueden interrumpir antes que la ciudad capital deje de recibir la producción de la generadora? (Sugerencia: investigue sobre el Teorema de Menger) Puede informar cual es el subconjunto de ejes cuya falla provoca este problema?

## Solution

### Algoritmo:

Dada la red de flujo de la compañía electrica, al buscar el corte mínimo en la misma a travez de un algoritmo como ford-fulkerson o edmond-karp (buscando primero el flujo maximo, y luego nos fijamos hasta que nodos se puede llegar desde la fuente en el grafo residual, entonces las aristas que salgan de esos ultimos nodos pertenecen al conjunto del corte minimo) obtendriamos cuales son los cables que ocasionarian fallas y los cuales son los primeros que habria que mejorar, tambien si se quiere saber que cables pueden empeorar sin que afecte a la ciudad, estos son los que luego de calcular el flujo maximo todavia tienen capacidad restante.

### Ejemplo:



### Complejidad:

La complejidad es la misma de edmond-karp en caso de usar este y es  **$O(V^*E^2)$**

La complejidad espacial sera  **$O(V + E)$**  ya que se le hace una copia a la red de flujo para crear el grafo residual

### Pseudocodigo:

```
def min_cut(graph, s, t):
    init flow for every edge in graph at 0
    create residual graph
```

```

while there is a path s,t in the residual graph: # with BFS
    residual flow = min weight of every edge in the path
    for edge in path:
        if edge in graph:
            flow of edge += residual flow
        else:
            flow of edge -= residual flow
    update residual graph accordingly

    # Once flow is calculated
    reachable nodes = get all reachable nodes from source in residual
graph # with BFS
    furthest nodes = get furthest nodes from source in reachable nodes

    min cut = set()
    for node in furthest nodes:
        add all graph.adjacent(node) to the min cut

return min cut

```

## 7-

La policía de la ciudad tiene "n" comisarías dispersas por la ciudad. Para un evento deportivo internacional deben asignar la custodia de "m" centros de actividades. Una comisaría y un centro de actividades pueden ser emparejados si y sólo si la distancia entre ellos no es mayor a un valor d. Contamos con la distancia entre todos los centros y las comisarías. Una comisaría sólo puede custodiar un centro. El centro puede ser custodiado por una comisaría. Determinar si es posible la asignación de tal forma que todos los centros estén custodiados. ¿Cómo modificaría la resolución del problema si en lugar de que cada centro de actividades i tenga que ser asignado a una sola comisaría, tenga que ser asignado a mi comisaría? ¿Cómo modificaría la resolución del problema si además hubiera una restricción entre comisarías que implicaría que una comisaría Ni y una Nj no pudieran ser asignadas juntas a un centro Mi? ¿Para qué casos dejaría de ser eficiente la resolución?

### Solution

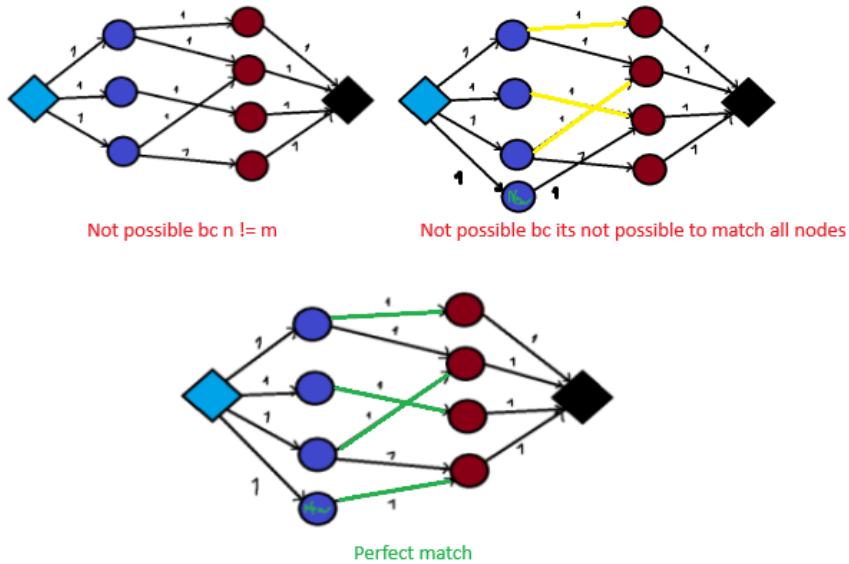
#### Algoritmo:

Primero se crea una red de flujo a partir del grafo donde a todas las estaciones de policia les llegara flujo de una superfuente, estas a su vez estaran conectadas con aristas a solamente los centros que les queda a distancia menor a d, y estos centros estaran conectados a un supersumidero. Las aristas tendran todas peso 1, ya que no queremos que una estacion de policias use mas de una arista porque eso significaria que esta cubriendo mas de un centro simultaneamente.

Una vez hecho esto se puede calcular el flujo maximo y si el flujo == m (y por consiguiente == n) entonces hay un perfect match en la red. Si se quiere saber que aristas se seleccionan

basta con ver las aristas con capacidad al maximo.

Ejemplo:



Complejidad:

La complejidad es la de calcular el flujo maximo (ford-fulkerson o edmond-karp). Temporal  $O(V^*E^2)$  y espacial  $O(V + E)$  por edmond-karp

Pseudocodigo:

*Igual a los flujos que ya hice arriba solo que en el return hay dar flujo\_max == m*

8-

Una compañía minera nos pide que la ayudemos a analizar su nueva explotación. Ha realizado el estudio de suelos de diferentes vetas y porciones del subsuelo. Con estos datos se ha construido una regionalización del mismo. Cada región cuenta con un costo de procesamiento y una ganancia por extracción de metales preciosos. (En algunos casos el costo supera al beneficio). Al ser un procesamiento en profundidad ciertas regiones requieren previamente procesar otras para acceder a ellas. La compañía nos solicita que le ayudemos a maximizar su ganancia, determinando cuales son las regiones que tiene que trabajar. Tener en cuenta que el costo y ganancia de cada región es un valor entero. Para cada región sabemos cuales son aquellas regiones que le preceden. Resolver el problema planteado utilizando una aproximación mediante flujo de redes.

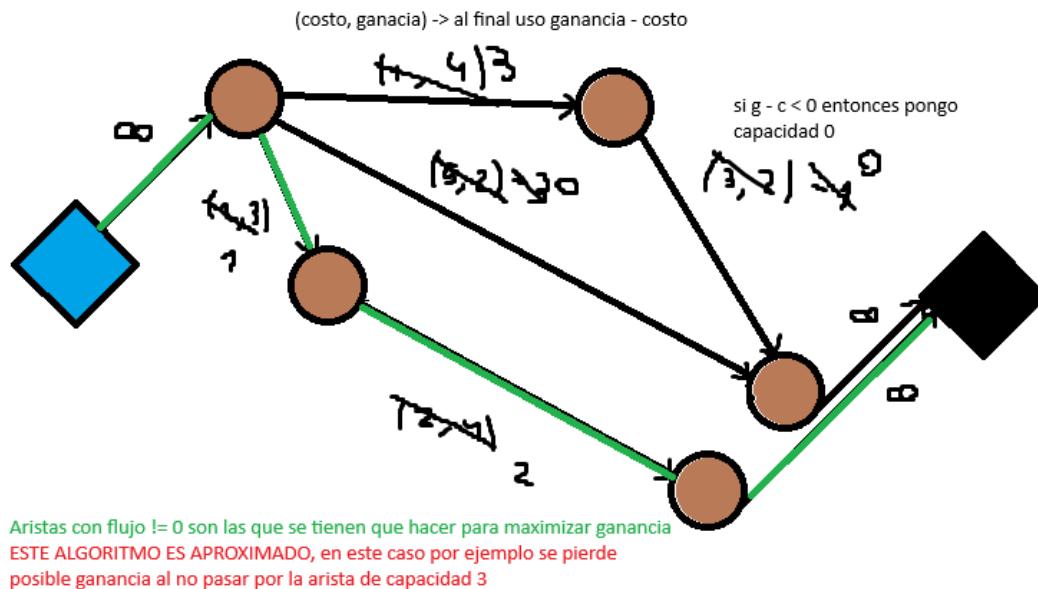
Solution

- Con ayuda de [Flor jeje](#)

Algoritmo:

Se toma la red dada y se usa edmond-karp para calcular flujo maximo, pero se hace un preprocesamiento a la red para que haya una fuente lleno a todos los nodos de grado de entrada 0 y un sumidero al que vayan todos los nodos de grado de salida 0. Luego a las aristas que tendrian como peso (costo, ganancia) las convierto en peso = ganancia - costo, y cuando una es negativa la pongo en 0, como tiene capacidad 0 no podra pasar flujo por ahí, es decir este algoritmo ignorara los caminos en que algun punto tengan costo > ganancia, inclusive aunq este camino de ser recorrido completo daria mas ganancia que costo, es poroso que este algoritmo da una aproximacion y no un resultado optimo. Una vez calculado el flujo maximo, todas las aristas que tengan flujo != 0 pasando por ellas del grafo original, estas son las que se debe atravesar.

Ejemplo:



Complejidad:

La complejidad temporal y espacial seran las mismas que edmond-karp  $O(V^*E^2)$  y  $O(V + E)$  respectivamente

Pseudocodigo:

```
def mine(graph, s, t):
    flow = edmondKarp(graph, s, t) # see previous exercises to see
    pseudocode for this part
    edges = set()
    for e in flow.keys():
        if flow[e] != 0:
            edges.add(e)
    return edges
```

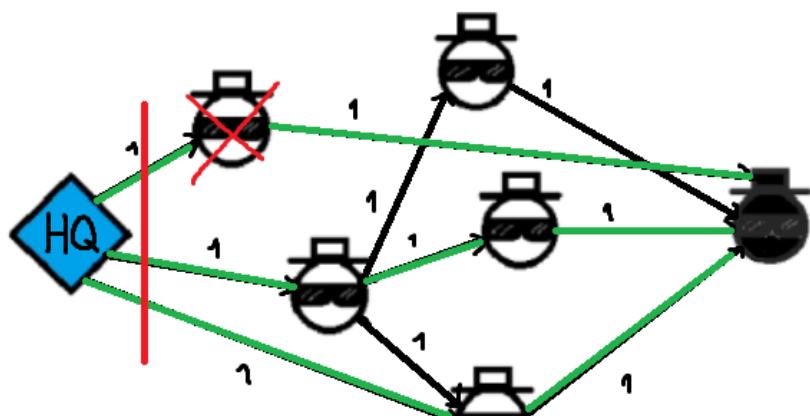
Una red de espías se encuentran diseminados por todo el país. Cada uno de ellos únicamente conoce a un número limitado de sus pares con los que pudo tener contacto dejando un mensaje escrito en una ubicación conocida. Este conocimiento no es recíproco. En caso de una crisis la agencia puede enviar mensajes utilizando esta red desde su base principal a un determinado agente especial. Una cuestión importante es que una vez utilizado un espía para transmitir un mensaje durante el resto de la crisis no se vuelve a utilizar. La agencia desea conocer, dada su red y un agente de destino de sus mensajes. ¿Cuál es la mínima cantidad de espías que un rival podría neutralizar para reducir en un 30% la cantidad de mensajes máximos que puede enviar desde la base al agente? Utilizando redes de flujos dar una solución al problema.

## Solution

### Algoritmo:

Se usa la red dada para calcular el corte mínimo a través de un algoritmo de flujo como ford-fulkerson o edmond-karp, una vez obtenido el corte mínimo vemos el final de cada una de estas aristas de corte en el grafo original, cada una tendrá un espia que de ser asesinado reduciría el flujo. Tenemos que ir simulando asesinatos de espías contando el porcentaje en el que se reduce el flujo siendo 100% si asesinamos a todos los espías.

### Ejemplo:



after calculating the min cut, at the end of each edge we'll have a spy who if killed we would get less flow.  
if we kill the 3 in this case we reduce the flow by 100% BUT if we kill only one we reduce it by 33% → 30% with an error of 3

### Complejidad:

La complejidad temporal y espacial será la misma que edmond-karp

### Pseudocódigo:

```
def espías(grafo, s, t):
    flujo = {}
    for v in grafo:
```

```

        for w in grafo.adj(v):
            flujo[(v, w)] = 0

    residual = crear_grafo_residual(grafo)

    while camino = obtener_camino_menor_longitud(residual, s, t):
        flujo_residual = min(capacidad for arista in camino)

        for i in range(1, len(camino)):
            if grafo.tiene_arista((camino[i-1], camino[i])):
                flujo[(camino[i-1], camino[i])] += flujo_residual
                actualizar_residual(residual, camino[i-1], camino[i],
                flujo_residual)
            else:
                flujo[(camino[i], camino[i-1])] -= flujo_residual
                actualizar_residual(residual, camino[i], camino[i-1],
                flujo_residual)

        furthest_reachable_spies = get_furthest_reachable_nodes(residual, s)
        min_cut_edges = set()
        for v in furthest_reachable_spies:
            for w in grafo.adj(v):
                min_cut_edges.add((v, w))

        espias_asesinados = 0
        while floor(espias_asesinados / len(min_cut_edges))*100 < 30:
            espias_asesinados += 1

    return espias_asesinados # numero de espías a asesinar para disminuir
en 30% el flujo

```

## 11-

En un juego multijugador cooperativo de "p" participantes se muestra una grilla de  $n \times n$  celdas. Inicialmente en una posición al azar se colocan los avatares de los jugadores e igual cantidad de cuevas. Cada "gusano" se desplaza 1 celda por turno ocupando una circundante que esté vacía (como máximo tiene 4: arriba, abajo, izquierda y derecha). A medida que se desplaza crece y por lo tanto sigue ocupando por las que pasó anteriormente. El objetivo del juego es lograr que los "p" jugadores lleven a sus gusanos a las cuevas (es indiferente a cual pero cada uno tiene que estar en una cueva diferente). Nos solicitan que realicemos el pseudocódigo que verifique en un turno determinado si aún es posible que los jugadores ganen. Es decir que todos los gusanos puedan llegar a la cueva.

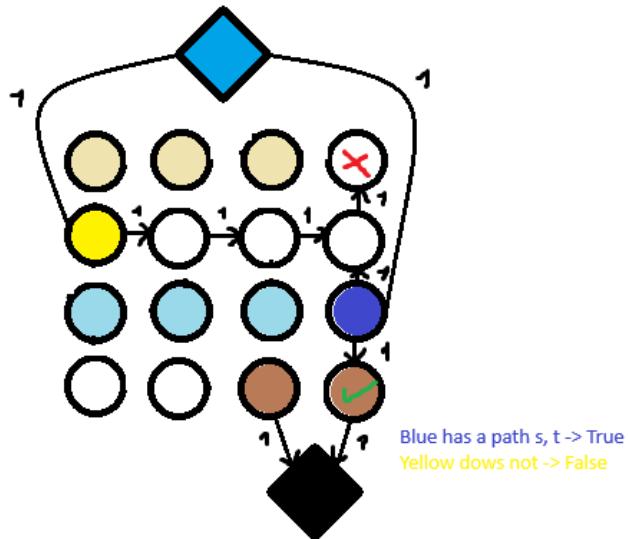
### Solution

#### Algoritmo:

El algoritmo consiste en primeramente crear un grafo donde cada cabeza de serpiente este recibiendo flujo de una fuente, el peso de esa arista sera 1. Tambien cada cabeza estara

conectado con aristas de peso 1 a sus posibles jugadas (wasd) luego ese nodo estara conectado a los posibles lugares a los que se puede ir desde ahí y así sucesivamente. se calcula el flujo y se ve si es igual al numero de jugadores en ese caso es True sino es False.

Ejemplo:



Complejidad:

La complejidad temporal sera  $O(V^*E^2)$  por calcular el flujo con edmond-karp

La complejidad espacial es  $O(V+E)$

Pseudocódigo:

```
def puede_ganar(grafo, s, t):
    return flujo(grafo, s, t) == cant_jugadores
```

13-

Una empresa de autobuses se conformó luego de la fusión de varias compañías menores. Actualmente tienen diferentes rutas que cubrir. Cada una con horario de inicio en una ciudad y finalización en otra. Existe la posibilidad de cubrir con un mismo micro diferentes rutas. Siempre la ruta comienza desde donde parte el micro, pero también puede pasar que el micro tenga tiempo suficiente para trasladarse hasta otro punto y cubrir otra ruta. Cuentan con una flota activa de  $N$  micros. Necesitan saber si les es posible cubrir con ella los requerimientos y si pueden contar con micros de backup ante la necesidad de controles programados de algunos de los móviles. Ayudar a resolver el problema mediante el uso de redes de flujo.

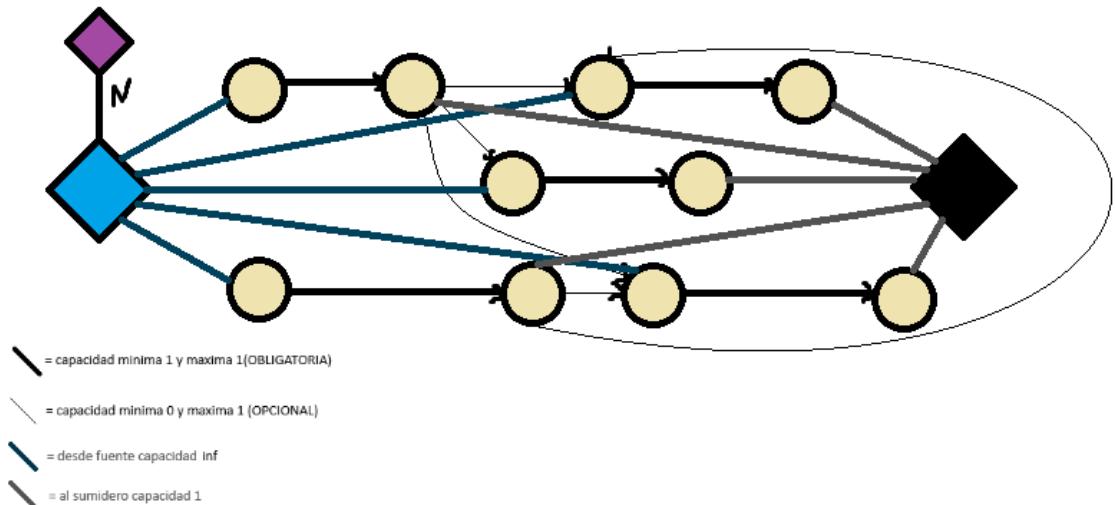
Solution

Algoritmo:

Se crea una red de flujo donde todos los orígenes estarán conectados a una fuente con aristas de capacidad infinita, y todos los destinos estarán conectados al sumidero con aristas de capacidad 1. A su vez entre orígenes y destinos se conectarán con aristas también de capacidad 1, pero se agregarán aristas desde los destinos hacia los orígenes de otros viajes siempre y cuando se pueda usar el mismo micro para ese viaje, es decir que de con el tiempo, estas nuevas aristas tendrán como capacidad mínima 0 (no se usa ese micro para este nuevo viaje) y como capacidad máxima 1 (si se usa).

Al maximizar el flujo por medio de un algoritmo como ford-fulkerson o edmond-karp maximizaremos los micros en uso, pero como la fuente da flujo infinito este podría decírnos que usemos más de los que tenemos, por eso la fuente se conecta a otra fuente que le da flujo N. Así garantizamos que no se pase de los micros existentes.

**Ejemplo:**



**Complejidad:**

La complejidad temporal y espacial será la misma que edmond-karp

**Peudocódigo:**

**Calcular flujo maximo y si este es igual a la cantidad de pares origen-destino entonces se pudo hacer todos los viajes**

**Skipped 6, 9, 12, 14, 15**

# Complexity

1. Is it in NP? + verifier and its pseudocode
2. Is it in NP-Complete? (reductions)
3. Extra additions (np-hard? pspace?)

## exercices

- P = todo problema que se puede resolver en tiempo polinomial.
- NP = todo problema que se puede verificar en tiempo polinomial.
- CO-NP = todo problema de decision cuya falsedad puede ser verificada en tiempo polinomial.
- NP-COMPLETE = todo problema en np tal que: todos los demás problemas np  $\leq$  este problema.
- NP-HARD = todo problema que son al menos tan difíciles como los problemas np-completos.
- PSPACE = todo problema que se puede resolver en una cantidad polinomial de espacio.

## Examples (buch)

### NP-Complete

#### Independent set

- Set de vértices sin compartir aristas entre si

IS  $\leq$  Vertex cover (todos los v no seleccionados por vertex cover es IS)

#### Vertex cover

- Set de vértices que cubra todas las aristas

VC  $\leq$  Independent set (todos los v no seleccionados por independent set es VC)

### SAT

- dado un numero de cláusulas C1, C2, C3... con un numero de variables X1, X2, X3... hay una configuración de variables para que todas las cláusulas sean true

C\_ejemplo = X1 or !X2 or X3

### 3-SAT

- SAT pero con 3 variables por cláusula en lugar de n.

para este problema se crean variables fake Z1, Z2, Z3... cuantas sean necesarias

C\_ejemplo = X1 or !Z1 or Z2

Este problema se ve mejor con triangulitos por cada clausula y los triangulitos tienen aristas entre si juntando posibles conflictos (ej: Z1 y !Z1), a este grafo de triangulitos se le hace Independent set y anda, osea que 3-SAT  $\leq$  IS

## K-Clique

- Determinar si dado un grafo existe un subgrafo completo con k vertices

IS  $\leq$  K-clique:

Definimos que hay un Independent Set de al menos K vértices si hay un clique de al menos K vértices en el grafo complemento (grafo con las aristas que no estan en el grafo original).

## Traveling salesman (viajante)

- The problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Ciclo hamiltoniano  $\leq$  TSP

1. Creamos un Grafo dirigido completo, con "los mismos vértices" que el original.
2. Las aristas del grafo anterior tienen peso 1 cuando existen en el original, y peso 2 si no existen en el original.
3. Existe un ciclo Hamiltoniano en el grafo original, si existe un camino TSP en el nuevo grafo de peso total  $n =$  cantidad de vertices.
4. Si es mayor que n, debe haber utilizado una arista de peso 2  $\rightarrow$  uso una arista que no existe en el original.

## Ciclo hamiltoniano

- ciclo que visita todos los vertices y vuelve al que empezo.

## Camino hamiltoniano

- camino que visita todas las aristas y no termina en el que empezo.

Ciclo H  $\leq$  Camino H

1. Agarramos un vértice al azar y lo reemplazamos por  $v'$  y  $v''$ . Todas las aristas salientes de  $v$  ahora salen de  $v'$ , todas las entrantes entran a  $v''$ .
2. Existe Ciclo Hamiltoniano en el original si existe Camino Hamiltoniano en el nuevo (ese camino necesariamente empieza en  $v'$  y termina en  $v''$ ).

## Two partition

- A partir de un set de numeros se pueden hacer 2 sets de igual largo (miti-miti) tal que los dos tienen la misma suma total.

## **Subset sum**

- Dado un set de numeros hay un subset de estos que suman K.

## **K-Coloreo**

- Decide si se puede pintar un grafo con K colores tal que no hay dos nodos adyacentes con el mismo color.

## **Mochila**

- Se tiene un peso maximo y se quiere agarrar el maximo valor en un array con elementos que tienen valor y peso.

Subset sum <= mochila, subset sum es literal mochila pero que el peso y valor es el mismo numero.

## **Set cover**

- Dado un set Universo y un set de sets Galaxias, encontrar la menor cantidad de sets galaxias cuya union forma todo el universo.

## **Dominating set**

- A dominating set for a graph G is a subset D of its vertices, such that any vertex of G is either in D, or has a neighbor in D.

## **Hitting set**

## **3D Matching**

## **Ejs**

## **2-**

Una agencia de marketing coloca publicidad en la Web. Se han ilusionado con vender publicidad con la siguiente idea, que llamaremos el problema de la Publicidad Estratégica: Un sitio Web se puede modelar como un grafo  $G = (V, E)$ . Las acciones habituales de los usuarios que visitan un sitio se pueden modelar mediante "t" recorridos posibles  $P_1, P_2, \dots, P_t$  (donde cada  $P_i$  es un camino (Finjo demencia que ahi dice arista) dirigido en  $G$ ). Dado un número  $k$ , se quieren elegir a lo sumo  $k$  vértices en  $G$  para poner publicidad, de modo tal que todos los "t" recorridos habituales pasen por al menos uno de esos vértices. Tenemos que mostrarle a esta empresa que su idea no es realizable por el momento ya que el

problema de la Publicidad Estratégica es NP-completo. Sugerencia: relacionarlo con cubrimiento de vértices.

## Solution

- Thankiu [flor](#) atss

### Esta en NP:

Una solucion a este problema puede ser verificada en tiempo polinomial, ya que lo unico que hay que hacer es fijarse la cantidad de aristas de entrada y salida de cada publicidad puesta, y si esta es mayor o igual a t entonces cubre todos los recorridos (o aristas, pq sigo fingiendo demencia que recorrido == camino == arista)

```
def verify(set, graph):  
  
    covered_edges = 0  
    for v in set:  
        covered_edges += v.in_edges() + v.out_edges()  
  
    return covered_edges >= len(graph.edges())
```

La complejidad temporal de esta funcion dependera de como se consigan las aristas de entrada y salida, podria ser **O(n)** siendo n la cantidad de vertices en el set, o **cuadratico** si conseguir dichas aristas es lineal.

### Esta en NP-Complete:

Para demostrar que es NP-Completo voy a reducir el problema vertex cover que es un problema conocido y ya esta probado que es no-completo, a este problema de publicidades, si es posible entonces se confirmara que este pertenece a esa clase.

- Primeramente convertire la instancia de vertex cover a un input valido para este problema, en este caso es casi inmediato, los vertices siguen siendo vertices pero a las aristas las tengo que hacer dirigidas, sin embargo no importa la direccion asi que simplemente las pongo arbitrariamente siguiendo un DFS desde un nodo arbitrario.
- Una vez preparado se usara 1 sola vez a messi para resolver este problema de poner publicidades ya que asumo que el sabe solucionarlo, para esto le paso el nuevo grafo creado y el numero K que es la cantidad de vertices que quiero que tenga el set de respuesta en caso de que sea posible.
- Despues de que messi me da la respuesta le agradezco porque ya tendre la respuesta al vertex cover.

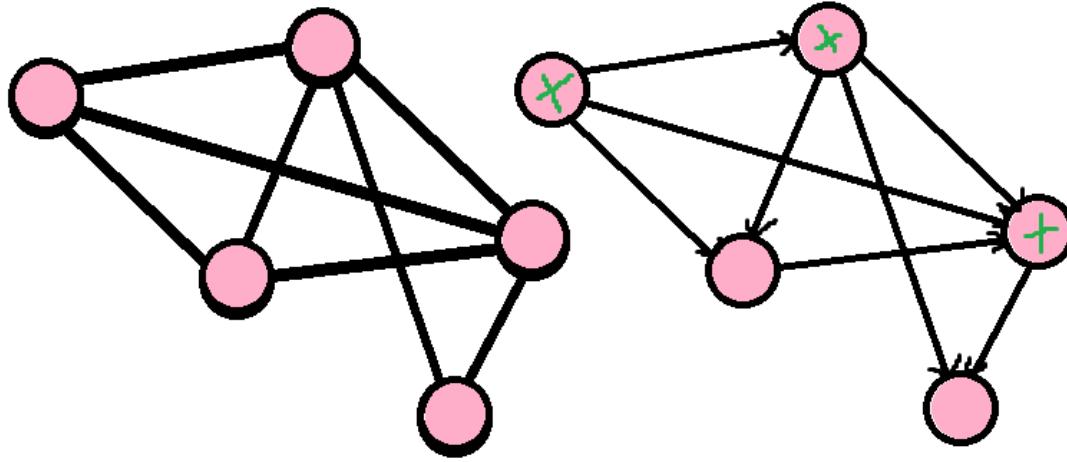
Entonces: Vertex cover  $\leq$  Publicidades (reduccion polinomica)

Por lo tanto: Publicidades **es NP-Complete**

### Ejemplo:

K = 3 -> True  
k = 2 -> False

$$I_{vc} \xrightarrow{T_1} I_p \xrightarrow{\quad} S_p = S_{vc}$$



Grafo para hacerle vertex cover

Grafo adaptado al problema de publicidades

Los nodos con X son el vertex cover K = 3

### 3-

Un almacén registra en una matriz qué productos compra cada uno de sus clientes. Un conjunto de clientes es diverso si cada uno de ellos compra cosas diferentes (tienen intersección vacía con lo que compran los demás). Definimos al problema de los clientes diversos como: Dada una matriz de registro, de tamaño m (clientes) x n (productos), y un número k<=m, ¿existe un subconjunto de tamaño al menos k de los clientes que sea diverso? Probar que el problema es NP-completo. Sugerencia: Reduce polinomialmente conjuntos independientes a clientes diversos.

#### Solution

Esta en NP:

Este problema esta en NP ya que puede ser facilmente verificable su solucion en tiempo polinomial usando el siguiente verificador:

```
def verify_clientes_diversos(sol, k):
    if len(sol) != k: return False
    productos_vistos = set()
    for c in sol:
        for producto in c.productos:
            if producto in productos_visitados: return False
            productos_visitados.add(producto)
    return True
```

Este verificador tiene complejidad temporal **O(n^2)** ya que itera todos los clientes en la sol y por cada uno itera todos sus productos (y es polinomial).

Esta en NP-Completo:

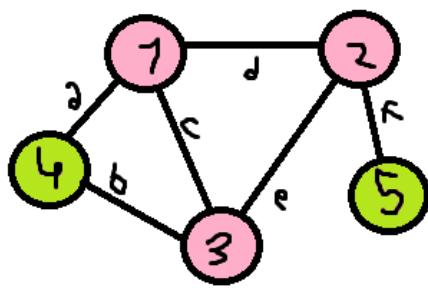
Para demostrar que este problema esta en NP-Completo otro problema NP-Completo se debe poder reducir polinomicamente a el, para esto elijo reducir independent set a este.

- Para poder hacer esto primeramente tenemos que convertir el input de independent set a un input valido para este problema, entonces voy a decir que todo nodo del grafo al que se le quiere calcular el independent set sera un cliente en nuestro problema, y cada arista sera un producto.
- Pasamos la matriz creada y el K de independent set a el resovedor de clientes diversos, este nos dara una respuesta y solo seleccionara los nodos que no tengan ningun producto comun entre si, es decir ninguna arista en comun entre si.
- Luego de usar 1 vez el resovedor ya tendremos un set de los clientes == nodos que conforman el independent set.

Entonces IS  $\leq$  Clientes dispersos (reducción polinomial) y Clientes dispersos **es NP-Completo**

Ejemplo:

$K = 2$   
 $K = 3$  is False



1	a	c	d			
2	d	f	e			
3	b	c	e			
4	a	b				
5	f					

$$I_{IS} \xrightarrow{\tau_1} I_{CD} \xrightarrow{1} S_{CD} = S_{IS}$$

4-

La siguiente es una versión de Conjunto Independiente. Dado un grafo  $G = (V, E)$  y un entero  $k$ , decimos que  $I \subseteq V$  es fuertemente independiente si dados dos vértices  $u$  y  $v$  en  $I$ , la arista  $(v, u)$  no pertenece a  $E$  y además no hay ningún camino de tamaño 2 (con dos aristas) de  $u$  a  $v$ . El problema de Conjuntos Fuertemente Independientes consiste en decidir si  $G$  tiene un conjunto fuertemente independiente de tamaño al menos  $k$ . Probar que el problema de Conjuntos Fuertemente Independientes es NP completo. Utilizar para ello que Conjuntos Independientes es NP completo.

Solution

### Esta en NP:

Esta en NP ya que hay verificador con complejidad polinomial para este problema.

```
def verify_sis(graph, sol, k):
    if len(sol) != k: return False
    for v in sol:
        for w in graph.adj(v):
            if w in sol: return False
            for u in graph.adj(w):
                if u in sol: return False

    return True
```

La complejidad temporal de esta funcion es  $O(V^3)$  siendo v la cantidad de vertices del grafo, ya que estamos iterando for todos los nodos solucion, y por cada uno iteramos por los ady, y por cada uno sus ady, siendo el peor caso cuando sol es del mismo largo que el grafo.

### Esta en NP-Completo:

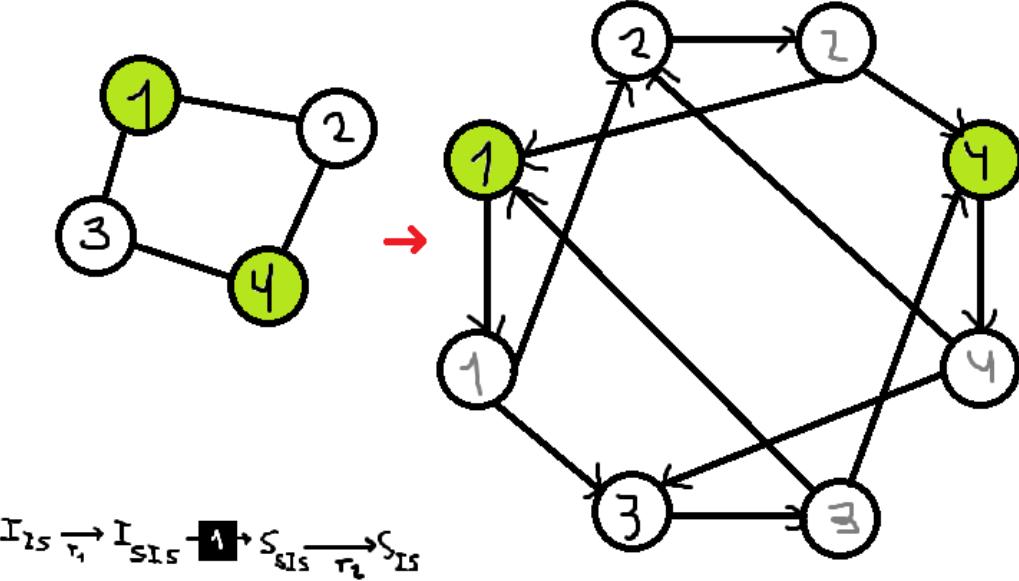
Ahora para demostrar que el problema es NP-completo voy a reducir un problema de esa misma clase a este, el problema a reducir sera Independent Set.

- Primeramente tenemos que transformar el input de Independent set a un input valido para el nuevo problema, para esto vamos a crear un nuevo grafo donde cada nodo en el original seran 2 nodos en el nuevo (digamos que nodo u y nodo u') y cada arista u,v en el grafo original sera una arista dirigida u',v en el nuevo, es decir que las aristas saldran de los nodos prima y entraran a los nodos no prima. Este proceso se puede hacer en tiempo **lineal**
- Ahora que tenemos el nuevo grafo le pasaremos el mismo junto con el mismo K al resovedor del problema de strong independent set y con usarlo solo 1 vez bastara.
- Una vez obtenido el resultado puede contener nodos prima como parte del set, asi que tendremos que convertir cada nodo prima a su contraparte no prima, es decir si tenemos el set {u', v'} se convertiria a {u, v} para nuestra respuesta de independent set. Este proceso sera **lineal** ya que solo hay que ver todos los nodos en la solucion y agregar a un nuevo set su contraparte en caso de ser prima o si mismos si no son prima para matchear el grafo original.

Entonces podemos decir que IS  $\leq$  SIS (reduccion polinomica) y SIS es NP-Completo

### Ejemplo:

K = 2



## 5-

Problema del camino más largo en un grafo general con aristas sin peso: Dados  $G = (V, E)$  un grafo no dirigido y un natural  $k$ , determinar si existe un camino simple en  $G$  de longitud  $\geq k$ . Probar que es un problema NP-completo (Usar el problema del camino hamiltoniano para probarlo).

### Solution

**Para mí** se puede hacer en tiempo polinomial haciendo bfs por cada nodo  $O(V^*(V+E))$ , así que si se pudiera reducir camino hamiltoniano a este entonces  $P = NP$ .

No entiendo porque pero [aca](#) dice (y prueba?) que es np-completo reduciendolo a hamiltonian path.

## 6-

Nos piden que organicemos una jornada de apoyo de estudio para exámenes. Tenemos que poder dar apoyo a "n" materias y hemos recibido currículos de "m" postulantes para ser potenciales ayudantes. Cada ayudante puede ayudar en un determinado subconjunto de materias. Para cada una de las materias hay un subconjunto de postulantes que pueden dar apoyo en ella. La pregunta es: dado un número  $k < m$ , ¿es posible seleccionar a lo sumo "k" ayudantes de modo tal que siempre haya un ayudante que pueda dar consultas en alguna de las n materias? Este problema se llama Contratación Eficiente. Probar que "Contratación Eficiente" es NP-completo. Sugerencia: se puede tratar de usar Cubrimiento de Vértices.

### Solution

Esta en NP:

Este problema pertenece a la clase de complejidades NP ya que existe un verificador polinomial para sus soluciones:

```
def verify_contratacion_eficiente(sol, k):
    if len(sol) != k: return False
    materias_cubiertas = 0
    for ayudante in sol:
        materias_cubiertas += len(ayudante.materias)
    return materias_cubiertas == cantidad_materias
```

Esta verificacion se puede hacer en tiempo **lineal**

Esta en NP-Completo:

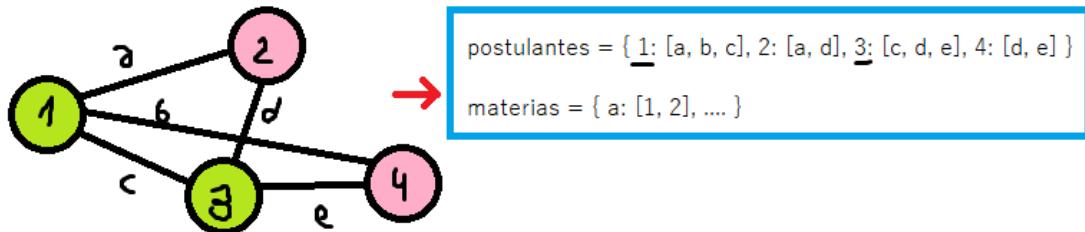
Para poder probar esto reduzco un Np-completo conocido, vertex cover, a este problema

- Se hace una transformacion donde cada nodo de vertex cover sera un ayudante de contratacion eficiente, y cada arista de vertex cover sera una materia. El K queda igual.
- Hecha la transformacion se usa 1 vez el resolvedor magico de este problema.
- La respuesta seran los K ayudantes que cubren todas las materias. lo que equivale a los K nodos que cubren todas las aristas osea el vertex cover.

Entonces VC <= CE (reduccion polinomial) y CE es NP-Completo

Ejemplo:

K = 2



7-

Una de las parejas más ricas del mundo está pasando por un proceso de divorcio. Entre sus bienes cuentan con propiedades, autos, motos, estampillas raras y otros coleccionables. Como no se ponen de acuerdo en la manera de dividirlos, el juez ha dictaminado que un tasador ponga valor a cada bien y luego se haga una partición por valores iguales (el

problema abstracto se conoce como 2-partition) El juez nos pide que elaboremos un algoritmo que en forma eficiente haga este trabajo. Demuestre que la solución pedida es NP-completa. Sugerencia: Pruebe con “subset sum”.

## Solution

- Tuuk clave **flor** y checkeado con sol de victor asi q atuuuuuuus

### Esta en NP:

Si esta en np ya que se puede escribir un verificador que corra en tiempo polinomial de la siguiente manera:

```
def verify_two_partition(sol1, sol2):
    if sum(sol1) != sum(sol2): return False
    return True
```

This verifier checks if the solution is correct or not in **O(n)**

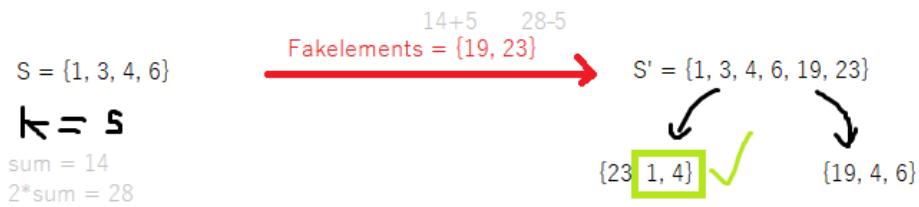
### Esta en NP-Completo:

Para demostrar esto reducir polinomialmente subset sum a two partition

- Primeramente se deberá hacer una transformación de la entrada de SS a una de TP, para esto se siguen los siguientes pasos.
  1. Cada elemento  $i$  queda igual y se pone en el set de TP
  2. Se crea un nuevo elemento igual a  $\text{sum}(SS\_set) + k$
  3. Se crea otro nuevo elemento igual a  $2 * \text{sum}(SS\_set) - k$
- Una vez hecha la transformación se calcula el two partition a ese set.
- Se toma las dos respuestas de two partition y a ambos sets se les saca los fakelements que habían sido creados previamente, ahora que sacamos esos dos la suma total de uno de esos dos sets será igual a  $k$  y ese set será la respuesta de Subset Sum.

Entonces SS  $\leq$  TP (reducción polinómica) y TP es NP-completo thusssss

### Ejemplo:



$$I_{ss} \xrightarrow{\tau_1} I_{rs} \xrightarrow{\text{[Redacted]} \square} S_{rr} \xrightarrow{\tau_2} S_{ss}$$

## 8-

El juego "escaleras y serpientes" consiste en un tablero con celdas numeradas del 1 al n. Cada jugador inicia en la casilla 1 y cada turno arroja un dado de 6 caras para determinar cuántas casillas avanza. En algunas casillas existen escaleras que permiten automáticamente ascender desde su pie hasta la cima. Al caer en ellas el jugador asciende a la casilla donde se apoya la cima de la escalera. En otras casillas existen serpientes que obligan al jugador a descender de su cabeza a cola. Al caer en una casilla donde se encuentra la cabeza de la serpiente el jugador desciende hasta la casilla donde se encuentra la cola de la misma. Diferentes empresas generan diferentes tableros (valor de n y ubicación y cantidad de serpientes y escaleras). El creador de una nueva versión nos muestra su prototipo y quiere que le digamos cuantos turnos durará como mínimo el juego. Se pide:

- Resolver el problema reduciéndolo a un problema de grafos.
- Indicar detalladamente todos los pasos de la reducción.
- Plantee el problema como un problema de decisión. ¿Puede afirmar que el problema pertenece a la clase P? ¿Puede afirmar que pertenece a NP?

## Solution

Este problema se puede reducir polinómicamente al problema de encontrar el camino mas corto en un grafo, para esto se siguen los siguientes pasos:

- Primeramente hay que generar un grafo al cual aplicar el resovedor de dicho problema, para generarla ponemos un nodo por cada casillero en el juego, es decir terminariamos con n nodos, y vamos a poner una arista dirigida desde un dado nodo v hasta cada una de las posibilidades segun el dado, es decir desde u a u+1, u a u+2, ...
- Ahora podremos usar el resovedor de camino mas corto una vez

- Y ahora tenemos en cuenta la cantidad de nodos en el camino y esa sera la minima cantidad de turnos en el problema de escaleras y serpientes

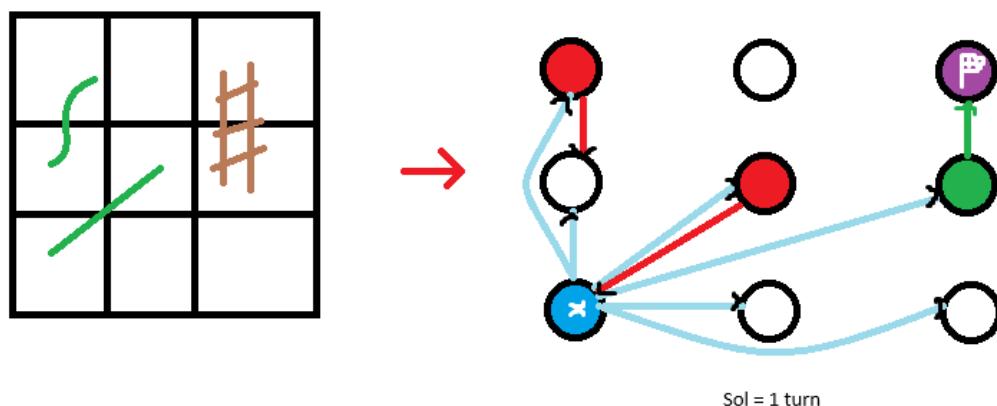
Esta en P:

Si esta en P ya que se puede resolver en tiempo polinomial utilizando el metodo mostrado arriba.

Esta en NP:

Tambien esta en NP porque se existe un verificador polinomial para este problema, y se puede verificar directamente buscando la solucion ya que es polinomial en si.

Ejemplo:



$$I_{es} \xrightarrow{\tau_i} I_{cn} \xrightarrow{\text{[black box]}} S_{cn} = S_{es}$$

9-

Se conoce Bin-Packing al problema de decisión donde se cuenta con "N" elementos de diferentes pesos y con "M" contenedores de cierta capacidad. Queremos saber si es posible acomodar todos los elementos en no más de k contenedores. Se pide demostrar que el problema es NP-Completo. Sugerencia utilizar 2-partition.

### Solution

Esta en NP:

Si esta en NP porque existe un verificador que verifica en tiempo polinomial la correctitud de la respuesta.

```
def verify_bin_packing(elems, cajas, k):
    if len(cajas) > k: return False
    elems_vistos = 0
    for caja in cajas:
```

```

if caja.contenido > caja.capacidad: return False
elems_vistos += caja.contenido
if elems_vistos < len(elems): return False
return True

```

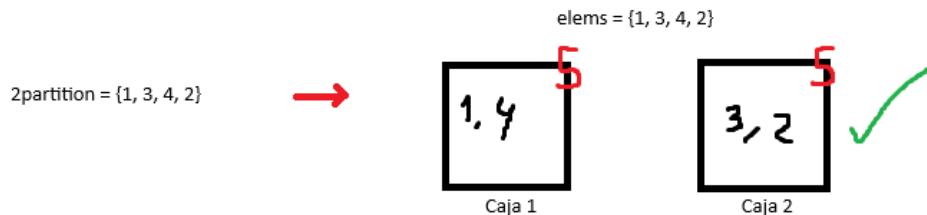
Esta en NP-Completo:

Para probar que este problema pertenece a esta clase de complejidad, voy a reducir un conocido NP-Completo 2 partition a este problema, de ser posible entonces este tambien lo es.

- Primeramente tengo que adaptar la entrada de 2 partition a este problema, el set de numeros quedara igual pero, solo crearemos 2 cajas y ambas tendran la misma capacidad = sum(elements) / 2
- Una vez se tiene el input al problema de Bin-packing se lo usa 1 vez.
- La solucion dada por este problema sera exactamente la necesitada por 2 partition.

Entonces 2P <= BP (reduccion polinomica) y BP esta en NP-Completo

Ejemplo:



$$I_{2P} \xrightarrow{\tau_1} I_{BP} \xrightarrow{1} S_{BP} \xrightarrow{\tau_2} S_{2P}$$

12-

Definimos el problema Subgrafo denso de la siguiente manera: Dado un grafo  $G=(V,E)$  y dos parámetros  $a$  y  $b$ . Existe en  $G$  un subconjunto de " $a$ " vértices con al menos " $b$ " ejes entre ellos. Demostrar que este problema es NP-Completo. Sugerencia: Utilizar el problema del Clique.

## Solution

Esta en NP:

Este problema pertenece a NP porque existe el siguiente verificador polinomial..

```
def verify_subgrafo_denso(graph, sol, a, b):
    if len(sol) != a: return False
    visitados = {}
    for v in sol:
        for w in graph.adj(v):
            if w in sol:
                visitados[v] += 1

    for v, cant in visitados:
        if cant != b: return False
    return True
```

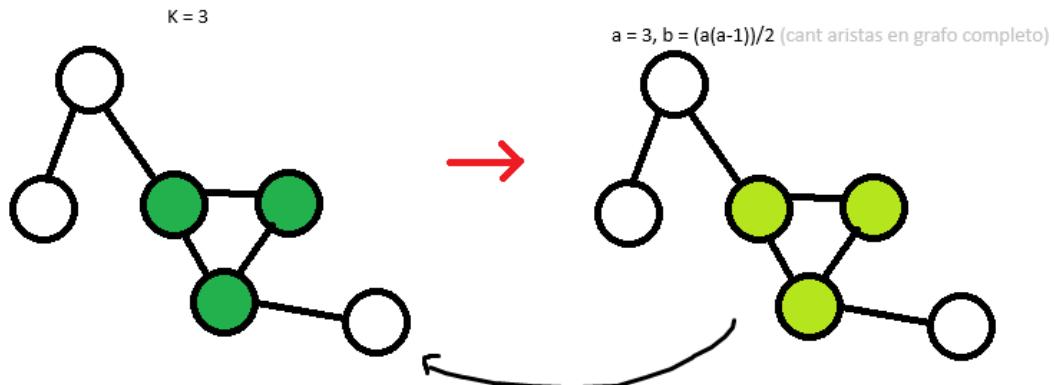
Esta en NP-Completo:

Se puede reducir el problema np-completo conocido llamado k-clique a este problema:

- El grafo de input quedara igual pero al numero 'a' se le asigna el mismo k de k clique y al numero 'b' se le asigna  $k(k-1)/2$  que es exactamente la cantidad de aristas que deveria haber en un grafo completo con k nodos.
- Una vez se tiene la entrada se le pide a messi que porfavor nos ayude a resolver subgrafo denso.
- Ahora que messi nos dio la respuesta esta es exactamente la misma que responde tambien k-clique.

Asi que entonces logramos K-CQ  $\leq$  SD (reduccion polinomica) y SD es NP-Completo

Ejemplo:



$$I_{CQ} \xrightarrow{\pi_1} I_{SD} \xrightarrow{[1]} S_{SD} = S_{CQ}$$

El directorio de una empresa realizará una cena de fin de año. En total son "n" directivos que se deben sentar alrededor de una mesa circular. Lamentablemente existen conflictos entre algunos de ellos que impiden que se sienten uno al lado del otro. Dado una instancia del problema, que incluye los n directivos y un listado donde se ven aquellos pares de directivos que están peleados entre sí, determinar si es posible sentarse en la mesa. Demostrar que el problema es NP-C. Sugerencia: Utilizar ciclo Hamiltoniano.

## Solution

Esta en NP:

Si esta en NP porque hay un verificador que verifica las soluciones a este problema en tiempo polinomial

```
def verify_directivos(directivos, se_caen_mal, sol):
    if len(sol) != len(directivos): return False

    for i in range(1, len(sol)):
        if (sol[i-1], sol[i]) in se_caen_mal: return False
        if i == len(sol) - 1 and (sol[0], sol[i]) in se_caen_mal: return False

    return True
```

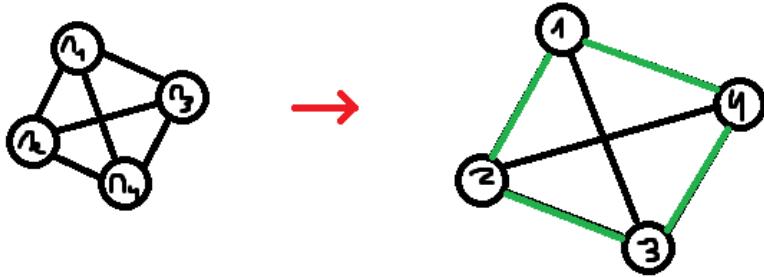
Esta en NP-Completo:

Se prueba que esta en esta clase reduciendo polinomialmente ciclo hamiltoniano a el problema de directivos.

- Dado un grafo al que se le quiere hacer ciclo hamiltoniano, lo convierto tal que nodo = directivo, y cada arista seran los que les cae bien a ese directivo, el resto le cae mal. asi se crea la entrada del problema de directivos.
- Una vez se tiene eso se usa el resovedor 1 vez.
- Este nos dara como tienen que estar sentados los directivos, ahora eso se transforma al camino nodo por nodo y obtenemos el ciclo hamiltoniano.

Entonces CH <= Directivos (reduccion polinomica) y Directivos es NP-Completo athussss

Ejemplo:



\* Nodo = directivo  
 \* aristas en G1 son los que se caen bien en G2, si no hay arista se caen mal  
 1 se sienta al lado de 4  
 4 se sienta al lado de 3  
 3 se sienta al lado de 2  
 2 se sienta al lado de 1

$$I_{CH} \xrightarrow{\tau_1} I_p \xrightarrow{\text{■}} S_p \xrightarrow{\tau_2} S_{CH}$$

## 18-

Una compañía multinacional desea contratar cobertura satelital para sus "n" sedes repartidas por el mundo. Han averiguado entre varias empresas que proveen el servicio pero ninguna de ellas tiene cobertura total. Les gustaría poder contratar a "k" o menos empresas. Pero tienen una condición adicional: al menos una de sus sedes debe tener cobertura de todas las empresas que la ofrecen. Con eso pueden iniciar una certificación de calidad que necesitan. Se pide: Demostrar que el problema es NP-Completo. Sugerencia: Utilizar Set Cover

### Solution

Esta en NP:

Si esta en NP porque se puede verificar en tiempo polinomial

```

def verify_satelites(sedes, empresas, sol, k):
    if len(sol) != k: return False

    sedes_vistas = {} init todas a 0.

    for e in sol:
        for sede in e:
            sedes_vistas[sede] += 1

    hay_central = False
    for sede, veces in sedes_vistas:
        if veces == 0: return False
        if veces == k:
            hay_central = True

    return hay_central
  
```

Esta en NP-Completo:

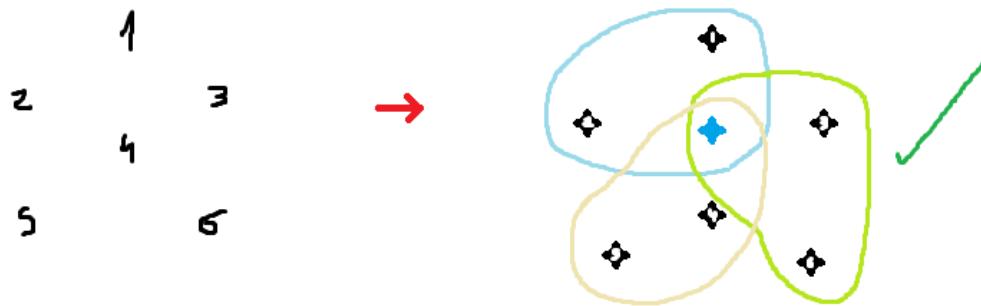
Si esta y se puede probar reduciendo set cover a este.

- Tomamos el set de set cover y ese va a ser equivalente a las sedes de este problema, los sets de cobertura quedaran casi iguales con la excepcion de que a cada uno se le agregara un nodo fantasma (el mismo para todos) que va a actuar como sede central.
- A partir de esta entrada se la da al resovedor de satelites.
- Nos dara los conjuntos que cubren todo, pero tenemos que sacarles el nodo fantasma a cada uno y asi obtenemos la solucion de set cover.

Ejemplo:

$K = 3$

$$U = \{1, 2, 3, 4, 5, 6\}$$
$$S = \{\underline{\{1, 2\}}, \underline{\{3, 6\}}, \underline{\{4, 5\}}, \{1, 2, 3\}, \{2, 4, 5\}\}$$



$$I_{sc} \xrightarrow{\tau_s} I_s \xrightarrow{\square} S_s \xrightarrow{\tau_c} S_{sc}$$

22-

Un departamento dentro de una universidad adquirió "n" proyectores para dar clases durante el cuatrimestre. Envío un formulario a los docentes de las distintas materias para conocer si los necesitaban como complemento para sus clases. Un subconjunto de docentes respondió afirmativamente. Sabiendo que cada materia tiene clases 1 o más veces por semana en un horario establecido. Y sabiendo que los horarios de varias de esas materias se superponen. Nos solicitan determinar si la cantidad comprada alcanza o si se tiene que dejar a docentes sin acceso a esas cuentas. Demostrar que lo solicitado es NP-COMPLETO.

Sugerencia: Tal vez le resulte útil "k" coloreo de grafos.

## Solution

Esta en NP:

Esta en NP porque hay un verificador en tiempo polinomial para ver si es correcta una sol a este problema.

```

def verify_proyectores(profes, sol, k):

    proyectores_usados = set()
    for profe in sol:
        proyectores_usados.add(profe.proyector) if no existe en el set
todavia
        for otro_profe in profe.conflictos:
            if profe.proyector == otro_profe.proyector: return False # Los dos
Lo usan al mismo tiempo

    if len(proyectores_usados) > k: return False
    return True

```

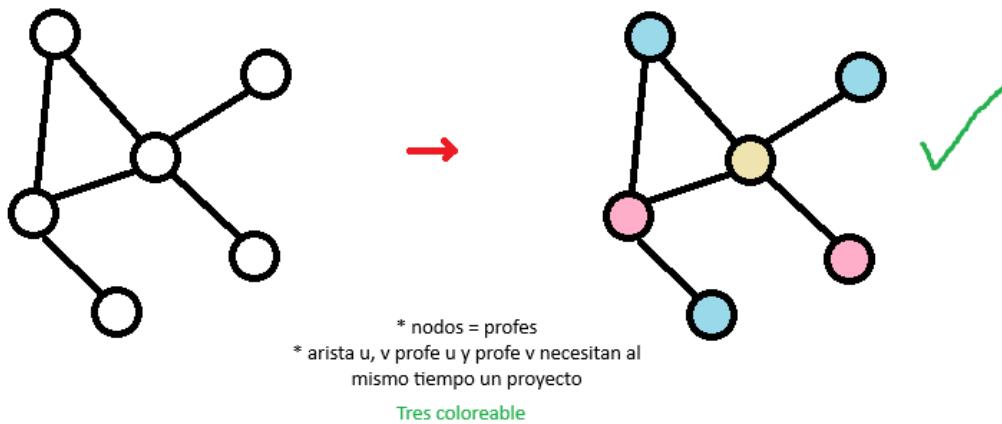
Esta en NP-Completo:

Para probar que es esta clase de complejidad voy a reducir polinomicamente un problema conocido np-completo k-coloreo a este.

- Primero transformamos el input de k-coloreo a proyectores, para esto cada nodo del grafo sera un profe y cada arista significara que esos profes tienen clases al mismo tiempo y no pueden usar el mismo proyector.
- Ahora con esta transformacion mandamos eso y el K al resovedor de proyectores.
- Este nos devolvera una respuesta de los profesores y que proyectores usan, esto lo transformamos a nodos y que colores son.

Ejemplo:

$K = 3$



$$I_{rc} \xrightarrow{\tau_i} I_p \xrightarrow{\text{[redacted]} S_p \xrightarrow{\tau_s} S_{rc}}$$

recommended by Flor: 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 14, 18, 20, 22

- vertex cover: 2, 6
- indep set: 3, 4
- hamiltoniano: 5, 8, 14

- subset sum: 7
- two partition: 9
- sat: 11
- clique: 12
- set cover: 18
- 3d matching: 20
- coloreo: 22

Skipped 1, 10, 11, 13, 15, 16, 17, 19, 20, 21, 23, 24, 25, 26, 27, 28, 29