AG1 – Diseño de algoritmos

03MIAR – Actividad Guiada 1(AG1)

Universidad Internacional de Valencia

Agenda

0. Entradas en el foro
1. Desarrollo de algoritmo con la técnica de **divide y vencerás**(Torres de Hanoi)
2. Desarrollo de algoritmo **voraz** para resolver problemas(devolución de cambio)
3. Desarrollo de algoritmo con la técnica de **vuelta atrás**(backtracking)(N-Reinas)
4. Desarrollo de algoritmo con **Programación dinámica**(paseo por el rio)

Preparar la actividad en Google Colaboratory.

- Abrir un notebook en Google Colaboratory

Actividad Guiada 1 de Algoritmos de Optimizacion

Nombre: XXXXX XXXXXXXX XXXXXX

https://colab.research.google.com/drive/xxxxxxxxxxxxxxxxxxxxxx

https://github.com/mi_usuario/03MAIR---Algoritmos-de-Optimizacion

```
[ ] #Torres de Hanoi - Divide y venceras
    ################################################################################
    #....
```

```
[ ] #Cambio de monedas - Técnica voraz
    ################################################################################
    #....
```

```
[ ] #N Reinas - Vuelta Atrás.
    ################################################################################

    #Verifica que en la solución parcial no hay amenzas entre reinas
    def es_prometedora(SOLUCION,etapa):
      #print(SOLUCION)
      #Si la solución tiene dos valores iguales no es valida => Dos reinas en la misma fila
      for i in range(etapa+1):
```

Universidad Internacional de Valencia

Referencia: https://colab.research.google.com/drive/1KdGsTbI3pbVcxlExfabt7rZqMVuIyGcy?usp=sharing

Preparar la actividad en Google Colaboratory.

- Abrir un notebook en Google Colaboratory
- Renombra el documento python : **Algoritmos - <nombre apellido> - AG1**
- Crear un texto con:

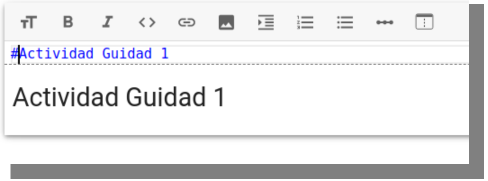
  * AG1- Actividad Guiada 1

  * Nombre Apellidos

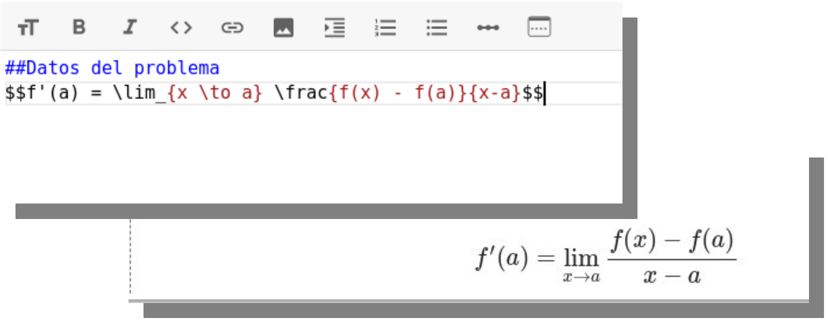
  * Url a la carpeta AG1 de GitHub

Ayuda para texto en Google Colab(Jupyter)

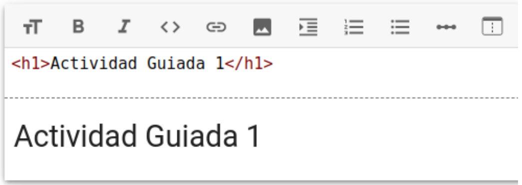
- Markdown:
  https://www.math.ubc.ca/~pwalls/math-python/jupyter/markdown/



- Latex:
  https://www.math.ubc.ca/~pwalls/math-python/jupyter/latex/

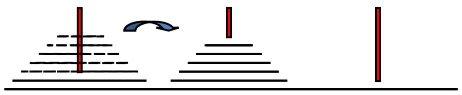


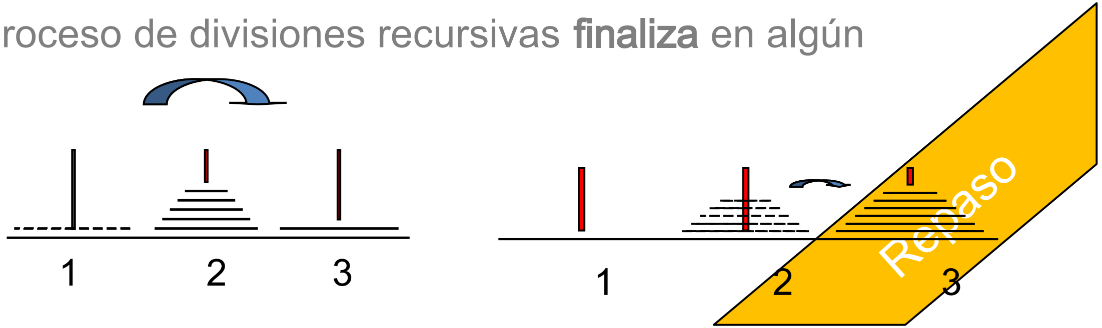
- HTML



Divide y vencerás (I)

- **Características que permiten identificar problemas aplicables:**
  - ✓ El problema puede ser **dividido** en problemas mas pequeños pero de la misma naturaleza que el principal.
  - ✓ Es posible resolver estos sub-problemas de manera recursiva o de otra manera sencilla (**caso simple**).
  - ✓ Es posible **combinar** las soluciones de los sub-problemas para componer la solución al problema principal.
  - ✓ Los sub-problemas son **disjuntos** entre si. No hay solapamiento entre los sub-problemas.
  - ✓ Debemos asegurar que el proceso de divisiones recursivas **finaliza** en algún momento.

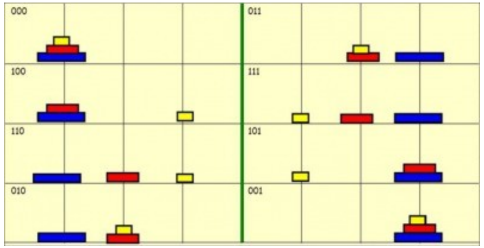




Divide y vencerás (II)

Problema: Torres de Hanoy. Código Python

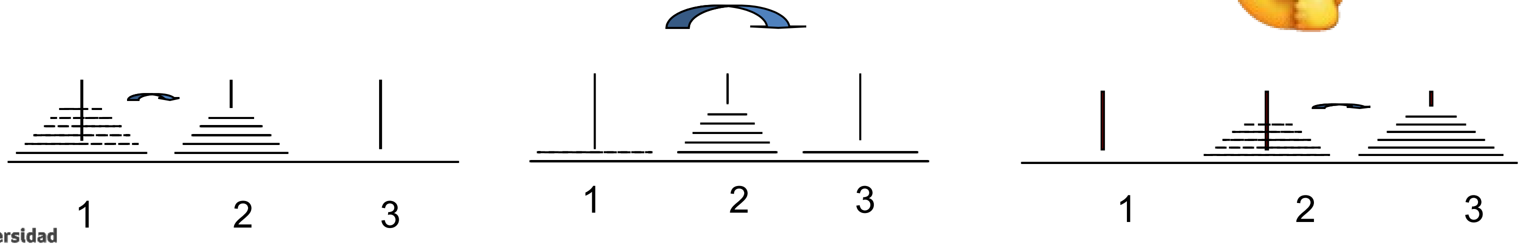
- Solución



Fuente: https://innovacioneducativa.upm.es/pensamientomatematico/node/76

```python
#Torres de Hanoi

def torres_hanoy(N, desde, hasta):
  if  N==1:
    print("LLevar desde " + str(desde) + " hasta " + str(hasta) )
  else:
    torres_hanoy(N-1,desde, 6-desde-hasta )
    print("LLevar desde " + str(desde)  + " hasta " + str(hasta) )
    torres_hanoy(N-1, 6-desde-hasta , hasta )

torres_hanoy(4,1,3)
```

```
LLevar desde 1 hasta 2
LLevar desde 1 hasta 3
LLevar desde 2 hasta 3
LLevar desde 1 hasta 2
LLevar desde 3 hasta 1
LLevar desde 3 hasta 2
LLevar desde 1 hasta 2
LLevar desde 1 hasta 3
LLevar desde 2 hasta 3
LLevar desde 2 hasta 1
LLevar desde 3 hasta 1
LLevar desde 2 hasta 3
LLevar desde 1 hasta 2
LLevar desde 1 hasta 3
LLevar desde 2 hasta 3
```



Para jugar online: http://www.uterra.com/juegos/torre_hanoi.php

Divide y vencerás(III)
Recursividad y calculo del numero de operaciones

```python
#Torres de Hanoi

def torres_hanoy(N, desde, hasta):
  if  N==1:
    print("LLevar desde " + str(desde) + " hasta " + str(hasta) )
  else:
    torres_hanoy(N-1,desde, 6-desde-hasta )
    print("LLevar desde " +  str(desde)  + " hasta " + str(hasta) )
    torres_hanoy(N-1, 6-desde-hasta , hasta )

torres_hanoy(4,1,3)
```

Operaciones

$$T(1) = 1$$

$$T(n) = 1 + 2 \cdot T(N-1) = 1 + 2 \cdot (1 + 2 \cdot T(n-2)) = 1 + 2 + 2^2 + \ldots + 2^n \cdot T(1) = \boxed{2^n - 1}$$

Complejidad

$$O(2^n)$$

Suma de una serie geométrica: $a + ar + ar^2 + ar^3 + \cdots + ar^n = \sum_{k=0}^{n} ar^k = a\left(\frac{1-r^{n+1}}{1-r}\right)$

Técnica Voraz. Algoritmos voraces(I) - (*Greedy algorithms*)

- **Definición:** Basada en la división del problema por etapas, eligiendo en cada etapa una decisión para construir la solución que resulte la más adecuada o ambiciosa sin considerar las consecuencias. Las decisiones descartadas será descartadas para siempre. **Resumen: elegir en cada etapa la decisión óptima.**
- Características que permiten identificar problemas aplicables:
  - ✓ Conjunto de candidatos (elementos seleccionables por etapas)
  - ✓ Solución parcial
  - ✓ **Función de selección** para determinar el mejor candidato en cada etapa.
  - ✓ Función objetivo
  - ✓ Función de factibilidad que asegure que una selección parcial es "prometedora"
  - ✓ Criterio o función que compruebe que una solución parcial ya es una solución final.

Repaso

Técnica Voraz. Algoritmos voraces(II) - (*Greedy algorithms)*

Problema: Cambio de monedas

Buscar las monedas para completar la cantidad con el sistema [25, 10, 5, 1]

- Definimos la función : **cambio_monedas** con dos parámetros: **Cantidad a calcular** y **sistema monetario**
- Inicializamos la variable(lista) **SOLUCION** a cero con tantos valores como tipos de monedas.
- Inicializamos la variable **VALOR_ACUMULADO** para contener el valor acumulado actual
- Recorremos todas las monedas en orden decreciente en valor (voracidad)
  - Calculamos el máximo de monedas posibles en cada iteración:

    monedas = int((CANTIDAD-VALOR_ACULULADO)/SISTEMA[i])
  - Actualizamos: **SOLUCION y VALOR_ACUMULADO**
  - Si llegamos a la cantidad devolvemos la solución:

    if VALOR_ACULULADO == CANTIDAD: return SOLUCION

Técnica Voraz. Algoritmos voraces(III) - (*Greedy algorithms*)

```python
#Cambio de monedas
################################################################################

def cambio_monedas(CANTIDAD,SISTEMA):

  print("SISTEMA:")
  print(SISTEMA)

  SOLUCION = [0 for i in range(len(SISTEMA)) ]          #Inicializamos el array que contendrá la cantidad de monedas de cada valor
  VALOR_ACULULADO = 0                                   #Inicializamos el valor acumulado


  for i in range(len(SISTEMA)):                         #Recorremos el sistema monetario (Conjunto de candidatos)
    monedas = int( (CANTIDAD-VALOR_ACULULADO)/SISTEMA[i]) #Calcula la cantidad de monedas de valor SISTEMA[i] (Función de selección)
    SOLUCION[i] = monedas                               #Añade el numero de monedas a la solución
    VALOR_ACULULADO += monedas * SISTEMA[i]             #Incrementa el valor acumulado (Función de factibilidad)
    if VALOR_ACULULADO == CANTIDAD: return SOLUCION     #finalizamos si ya hemos llegado a la solución(Criterio de solución final)



  return SOLUCION

SISTEMA = [25, 10, 5, 1]
cambio_monedas(27, SISTEMA)
```

```
SISTEMA:
[25, 10, 5, 1]
[1, 0, 0, 2]
```

Técnica Voraz. Algoritmos voraces(IV) - (*Greedy algorithms)*

```
#Cambio de monedas
##########################################################

def cambio_monedas(CANTIDAD,SISTEMA):

  print("SISTEMA:")
  print(SISTEMA)

  SOLUCION = [0 for i in range(len(SISTEMA)) ]
  VALOR_ACULULADO = 0


  for i in range(len(SISTEMA)):
    monedas = int( (CANTIDAD-VALOR_ACULULADO)/SISTEMA[i])
    SOLUCION[i] = monedas
    VALOR_ACULULADO += monedas * SISTEMA[i]
    if VALOR_ACULULADO == CANTIDAD: return SOLUCION



  return SOLUCION
```

- ¿Qué ocurre con otros sistemas monetarios?

```
Sistema_Monetario = [ 11, 5, 1]
```

```
print(cambio_monedas(N=15, SM=Sistema_Monetario))
```

```
[1, 0, 4]
```

- ¡Ojo! No siempre es funciona

- ¿Cuando funciona bien y cuando no?

Algoritmo con la técnica vuelta atrás con Python(I). Backtracking

- **Definición:** Construcción sistemática y por etapas de todas las soluciones posibles que pueden representarse mediante una tupla (x_1, x_2,… X_n) en la que cada componente x_i puede explorarse en la etapa i-esima. A través de un **árbol de expansión** se modela todo el espacio de soluciones donde cada nodo es un valor diferente para cada elemento x_i.
- Características que permiten identificar problemas aplicables:
  - ✓ Problemas discretos en los que las soluciones se componen de elementos que pueden ser relacionados para expresarlos en un árbol de expansión.
  - ✓ Es posible encontrar un criterio para descartar determinadas ramas(ramificación y poda[*]) y evitar un análisis exhaustivo (fuerza bruta)

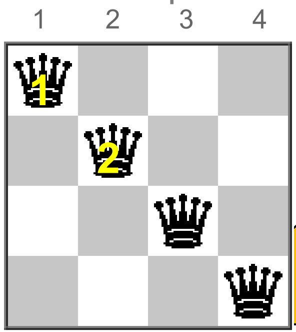
(*) La veremos más adelante asociada a la técnica general de búsqueda en arboles

Repaso

Algoritmo con la técnica vuelta atrás con Python(II). Backtracking

- **Problema: Problema de las 4 reinas**
  - Solución : 4-tuplas (x1, x2, x3, x4) donde el valor de cada elemento es la posición(fila) de una reina en la columna i-esima. P.ej la del dibujo es (1,2,3,4)
  - El árbol de expansión recorrerá todas las posibilidades.
  - Con este modelo, es posible determinar si una solución parcial (rama del árbol) es “prometedora”
  - No puede haber dos reinas en la misma columna. Esta restricción se verifica por el modelo que hemos adoptado
  - Dos reinas estarán en la misma fila si hay dos valores iguales para una solución parcial.

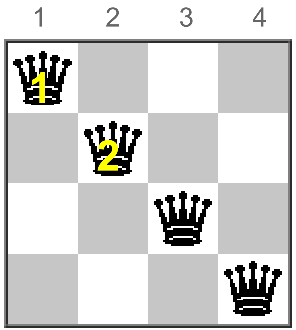
  P.Ej: (1,2,*,*) representa las dos primeras reinas de la imagen (2ª etapa)
  - Dos reinas estará en la misma diagonal si |xi -xj| = |i-j|



Repaso

Algoritmo con la técnica vuelta atrás con Python(III). Backtracking

- **Problema: Problema de las 4 reinas**
  - Usaremos la recursividad (habitual también en la técnica de vuelta atrás) para ir construyendo en cada iteración una etapa de la solución(una nueva reina en la siguiente columna)
  - Definimos una función: **reinas(N, solución , etapa)**
    - N = nº de reinas del tablero NxN
    - solucion = [0,0….0]
    - Etapa = 0



  - ¿Qué hace la función **reinas()** ?

    1º añade una etapa : **SOLUCION[etapa] = i ; para i desde 1 hasta N**

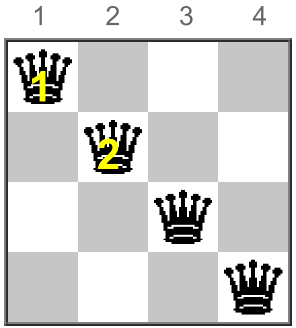
    2º comprueba que es prometedora con la función: **es_prometedora(solucion,etapa)**

    3º si la solución es prometedora y es la última etapa entonces es solución final

Universidad Internacional de Valencia

Algoritmo con la técnica vuelta atrás con Python(IV). Backtracking

- **Problema: Problema de las 4 reinas**
  - ¿Qué realiza la función `es_prometedora(solucion,etapa)` ?
    - Comprueba que no hay reinas en la misma fila o lo que es lo mismo que no hay dos elementos iguales en `SOLUCION`:
      `if SOLUCION.count(SOLUCION[i]) > 1:return False para i=0 hasta etapa`
    - Comprueba que no hay reinas en la misma diagonal o lo que es lo mismo que no se da $|i-j| = |X_i - X_j|$ para todos los i, j de `SOLUCION`:
      `if abs(i-j) == abs(SOLUCION[i]-SOLUCION[j]) : return False`



Algoritmo con la técnica vuelta atrás con Python(V). Backtracking

- **Problema: Problema de las 4 reinas con recursividad**

```python
#Proceso principal de N-Reinas
def reinas(N,solucion=[],etapa=0):
  # N          - Tamaño del tablero
  # solucion  - Solucion parcial
  # etapa     - nº de reinas colocadas en la solución parcial(

  #Inicializa la solución: una lista con ceros
  if len(solucion) == 0:
      solucion=[0 for i in range(N)]

  #Recorremos todas las reinas
  for i in range(1,N+1):
    solucion[etapa] = i

    #print(solucion)
    if es_prometedora(solucion,etapa):
      if etapa == N-1 :
        print("\n\nLa solución es:")
        print(solucion)
        escribe_solucion(solucion)
      else:
        #print("Es prometedora\n###################")
        reinas(N,solucion,etapa+1)
    else:
      #print("NO PROMETEDORA\n###############
      None

    solucion[etapa] = 0
```

Complejidad:

Cálculos complejos(con ecuaciones en recursividad) pero en general Vuelta atrás es Exponencial

```python
def es_prometedora(SOLUCION,etapa):
  #print(SOLUCION)
  #Si la solución tiene dos valores iguales no es valida => Dos reinas en la misma fila
  for i in range(etapa+1):
    #print("El valor " + str(SOLUCION[i]) + " está " +  str(SOLUCION.count(SOLUCION[i])) + " veces")
    if SOLUCION.count(SOLUCION[i]) > 1:       return False

    #Verifica las diagonales
    for j in range(i+1, etapa +1 ):
      #print("Comprobando diagonal de " + str(i) + " y " + str(j))
      if abs(i-j) == abs(SOLUCION[i]-SOLUCION[j]) : return False
  return True
```

$O(n^2)$

Algoritmo con la técnica vuelta atrás con Python(VI). Backtracking

- **Problema: Problema de las N reinas.**

LAVANGUARDIA | Otros Deportes

RESULTADOS

Al Minuto Internacional Política Opinión Vida Deportes Economía Local Gente Cultura Sucesos Temas

Directo Negociación in extremis entre PSOE y Unidas Podemos: todos los detalles de última hora

HASTA AHORA INDESCIFRABLE

Ofrecen un millón de dólares a quien resuelva este problema de ajedrez

- Unos investigadores proponen resolver el 'enigma de las ocho reinas' para tableros de 1000X1000

REDACCIÓN
06/09/2017 14:44

Actualizado a
06/09/2017 15:17

Fuente: https://www.lavanguardia.com/deportes/otros-deportes/20170906/431089708625/ocho-reinas-ajedrez-millon-dolares.html

Programación dinámica (I)

importante

- **Definición:** Es posible dividir el problema en subproblemas más pequeños, guardando las soluciones para ser utilizadas más adelante.
- Características que permiten identificar problemas aplicables:
  - ✓ Es posible almacenar soluciones de los subproblemas para ser reutilizadas.
  - ✓ Debe verificar el **principio de optimalidad** de Bellman: “*en una secuencia optima de decisiones, toda sub-secuencia también es óptima*” (*)
  - ✓ La necesidad de guardar la información acerca de las soluciones parciales unido a la recursividad provoca la necesidad de preocuparnos por la complejidad espacial (cuantos recursos de espacio usaremos)

(*) Ojo! No significa que si tenemos soluciones optimas de los sub-problemas las podamos combinar para el optimo del problema principal.

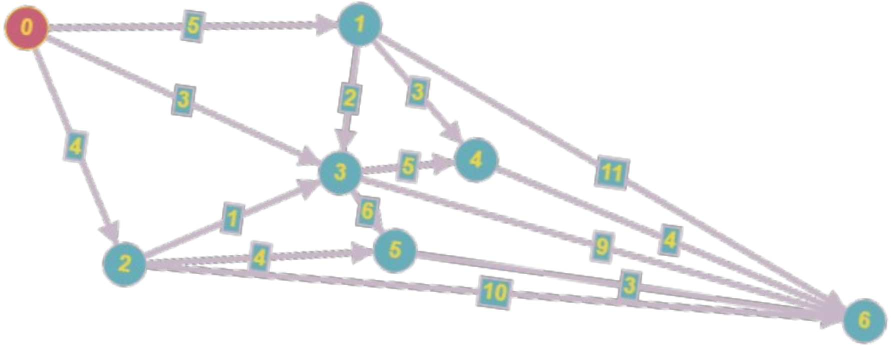
Repaso

Programación dinámica (II)

Problema: Viaje por el rio

- Consideramos una tabla T(i,j) para almacenar todos los precios que nos ofrecen los embarcaderos
- Si no es posible ir desde i a j daremos un valor alto para garantizar que ese trayecto no se va a elegir en la ruta óptima(modelado habitual para restricciones)
- Establecer una tabla intermedia(P(i,j)) para guardar soluciones óptimas parciales para ir desde i a j.

$$P(i,j) = \min \{T(i,j)\ ,\ P(i,k)+T(k,j) \text{ para todo } i<k<=j\ \}$$



Programación dinámica (III)

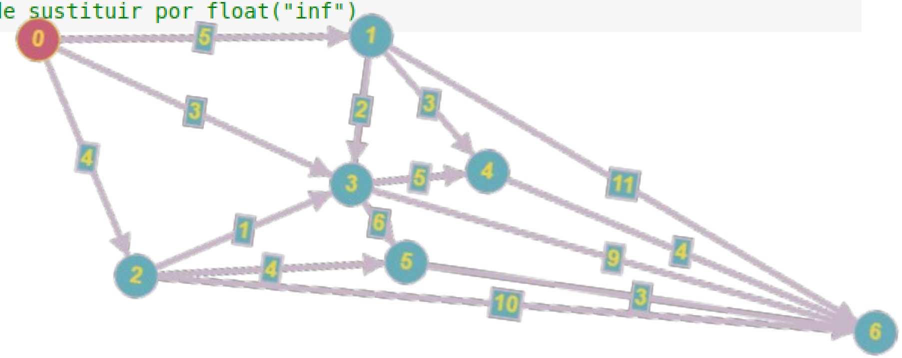
Problema: Viaje por el rio

- Establecemos las tarifas:

```
#Viaje por el rio - Programación dinámica
################################################################################

TARIFAS = [
[0,5,4,3,999,999,999],
[999,0,999,2,3,999,11],
[999,999, 0,1,999,4,10],
[999,999,999, 0,5,6,9],
[999,999, 999,999,0,999,4],
[999,999, 999,999,999,0,3],
[999,999,999,999,999,999,0]
]

#999 se puede sustituir por float("inf")
```

Se puede usar

```
import math
float("inf")
```



Programación dinámica (IV)

```
#####################################################################
def Precios(TARIFAS):
#####################################################################
  #Total de Nodos
  N = len(TARIFAS[0])

  #Inicialización de la tabla de precios
  PRECIOS = [ [9999]*N for i in [9999]*N]
  RUTA = [ [""]*N for i in [""]*N]

  for i in range(N-1):
    for j in range(i+1, N):
      MIN = TARIFAS[i][j]
      RUTA[i][j] = i

      for k in range(i, j):
        if PRECIOS[i][k] + TARIFAS[k][j] < MIN:
            MIN = min(MIN, PRECIOS[i][k] + TARIFAS[k][j] )
            RUTA[i][j] = k
        PRECIOS[i][j] = MIN


  return PRECIOS,RUTA
#####################################################################
```

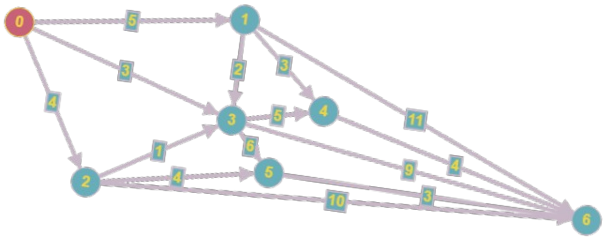
Operaciones

n

n

n

$3 \cdot n^3$



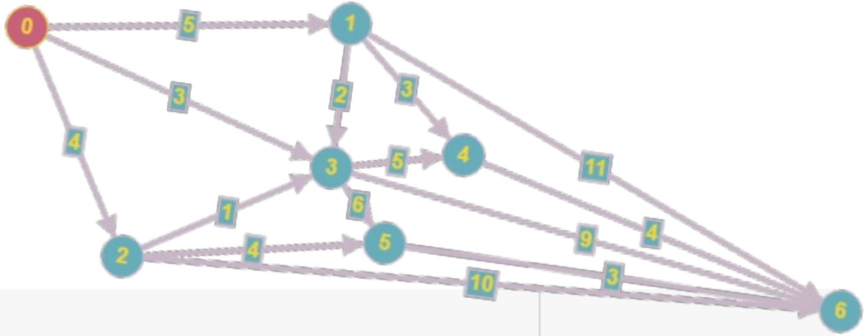

```
[ ]  TARIFAS = [
     [0,5,4,3,999,999,999],
     [999,0,999,2,3,999,11],
     [999,999, 0,1,999,4,10],
     [999,999,999, 0,5,6,9],
     [999,999, 999,999,0,999,4],
     [999,999, 999,999,999,0,3],
     [999,999,999,999,999,999,0]
     ]
```

$$P(i,j) = \min \{T(i,j)\ ,\ P(i,k)+T(k,j) \text{ para todo } i<k<=j \}$$

Programación dinámica (V)

- **RUTA contiene la mejor opción intermedia para ir de un nodo a otro**

```
RUTA
['', 0, 0, 0, 1, 2, 5]
['', '', 1, 1, 1, 3, 4]
['', '', '', 2, 3, 2, 5]
['', '', '', '', 3, 3, 3]
['', '', '', '', '', 4, 4]
['', '', '', '', '', '', 5]
['', '', '', '', '', '', '']
```




```python
def calcular_ruta(RUTA, desde, hasta):
  if desde == hasta:
    #print("Ir a :" + str(desde))
    return desde
  else:
    return str(calcular_ruta(RUTA, desde, RUTA[desde][hasta]) ) +  ',' + str(RUTA[desde][hasta])

print("\nLa ruta es:")
calcular_ruta(RUTA, 0,6)
```

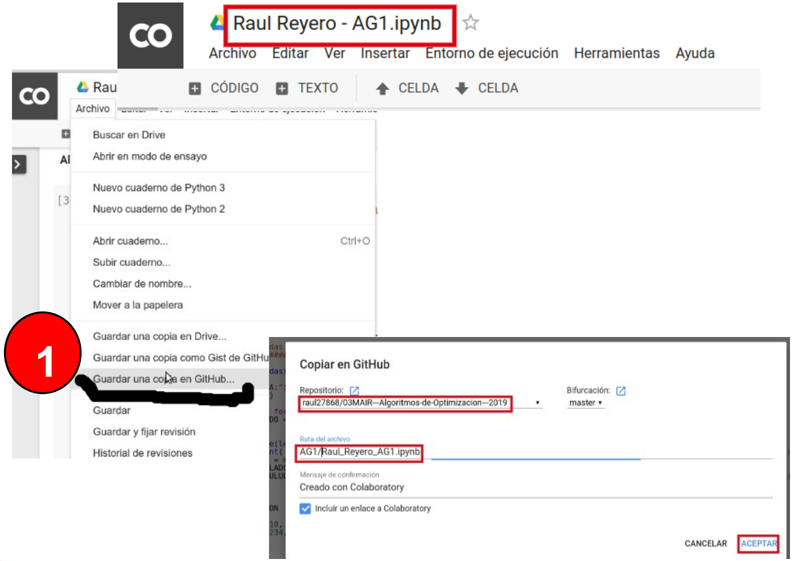
Recursividad

Finalizar la actividad. Grabar, subir a GitHub, Generar pdf (I)

- Guardar en GitHub

  Repositorio: 03MIAR ---Algoritmos de Optimizacion

  Ruta de Archivo con **AG1**



Ojo! Si el repositorio es **privado** no se podrá.

Opciones:

a) Hacerlo publico + guardar + hacerlo privado

b) Guardar el .ipynb manualmente

Finalizar la actividad. Grabar, subir a GitHub, Generar pdf (I)

8/10

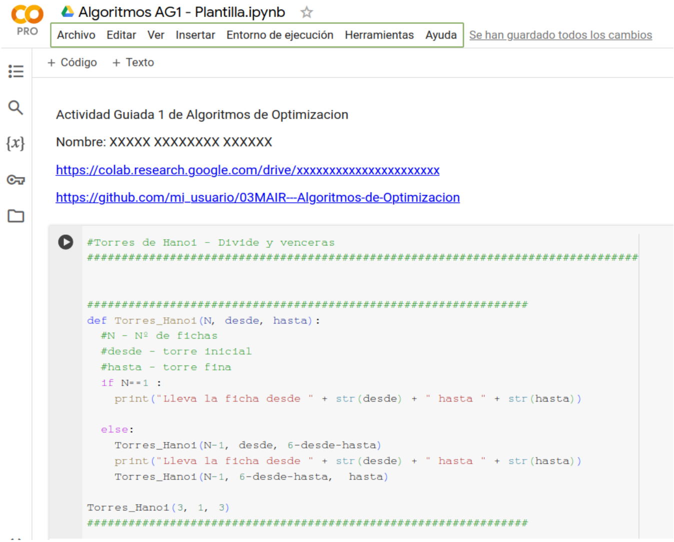
- Descargar pdf y adjuntar el documento generado a la actividad en la plataforma
  - Adjuntar .pdf en la actividad
  - URL GitHub en el texto del mensaje de la actividad

Universidad Internacional de Valencia

Practica individual

8/10

- No he podido seguir la práctica



https://colab.research.google.com/drive/1KdGsTbI3pbVcxlExfabt7rZqMVuIyGcy?usp=sharing

Practica individual

10/10

- **Problema: Encontrar los dos puntos más cercanos**
  - Dado un conjunto de puntos se trata de encontrar los dos puntos más cercanos
  - Guía para aprendizaje:
    - ✓ Suponer en 1D, o sea, una lista de números: [3403, 4537, 9089, 9746, 7259, ….
    - ✓ Primer intento: Fuerza bruta
    - ✓ Calcular la complejidad. ¿Se puede mejorar?
    - ✓ Segundo intento. Aplicar Divide y Vencerás
    - ✓ Calcular la complejidad. ¿Se puede mejorar?
    - ✓ Extender el algoritmo a 2D: [(1122, 6175), (135, 4076), (7296, 2741)…
    - ✓ Extender el algoritmo a 3D.

Practica individual

10/10

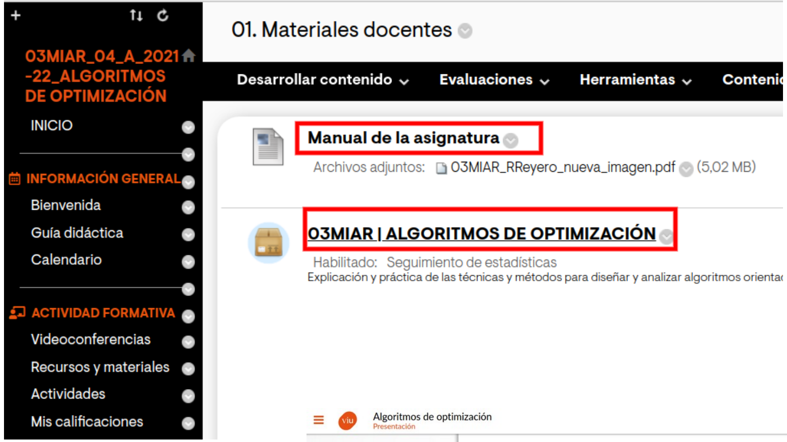
- **Problema: Encontrar los dos puntos más cercanos**
  - Para generar conjuntos de datos aleatorios

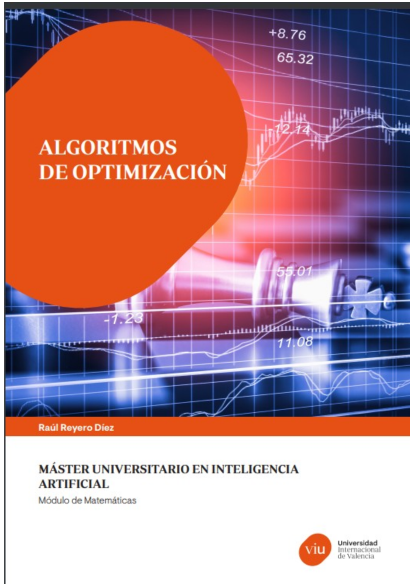
Práctica Indiviudal. Dos puntos más cercanos

```
[ ]  import random

     LISTA_1D = [random.randrange(1,10000) for x in range(1000)]

     LISTA_2D = [(random.randrange(1,10000),random.randrange(1,10000)) for x in range(1000)]
```

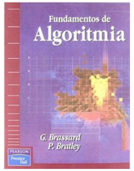
  - Buscar documentación sobre el problema

Manual de la asignatura



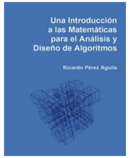


Bibliografía



Fundamentos de algoritmia: Una perspectiva de la ciencia de los computadores
Paul Bratley , Gilles Brassard
ISBN 13: 9788489660007

Introducción al diseño y análisis de algoritmos
R.C.T. Lee,...
ISBN 13: 9789701061244



Una introducción a las matemáticas para el análisis y diseño de algoritmos(*)
Pérez Aguila, R.
ISBN 13: 9781413576474
https://tinyurl.com/yzlt5oed

Técnicas de diseño de algoritmos
Guerequeta, R., y Vallecillo, A. (2000).
http://www.lcc.uma.es/~av/Libro

(*)En la biblioteca de la VIU

¿Preguntas?

Feliz Navidad

Gracias

raul.reyero@campusviu.es

Gracias

raul.reyero@campusviu.es

viu | Universidad Internacional de Valencia