Algoritmos_Michael_Laudrup_Luis_González_AG1

May 28, 2025

0.1 Aspectos adicionales

Se han realizado los siguientes cambios sobre la actividad propuesta base

- Fibonacci: He añadido un apartado denominado Fibonacci 2, el cual, desarrolla el mismo algoritmo planteado pero añadiendo memoria caché, esto es una técnica de programación dinámica (por el uso de memoria) que permitirá ahorrar cálculos duplicados.
- Devolución cambio de monedas: Se aporta una versión que tiene más robustez y menos operaciones aritméticas y comprobaciones con condicionales. Además, se añade una aclaración respecto al algoritmo que explica porque no se obtiene la solución óptima global.
- Se aporta una solución al problema de obtener los puntos con menor distancia dado un listado de puntos en diferentes formatos.

1 Algoritmos - Actividad Guiada 1

Nombre: Michael Laudrup Luis González

URL: https://drive.google.com/file/d/1gJRtteAQEgMNendYXXeXCqWaRaizZiHt/view?usp=sharing

https://github.com/MichaeLaudrup/Master_AI_VIU/tree/main/00_Optimizacion_algoritmos

1.1 Torres de Hanoi con Divide y vencerás

[ ]:
```
def Torres_Hanoi(N, desde, hasta):
  if N == 1:
    print("Lleva la ficha " ,desde , " hasta ", hasta )
  else:
    #Torres_Hanoi(N-1, desde, 6-desde-hasta )
    Torres_Hanoi(N-1, desde, 6-desde-hasta )
    print("Lleva la ficha " ,desde , " hasta ", hasta )
    #Torres_Hanoi(N-1,6-desde-hasta, hasta )
    Torres_Hanoi(N-1, 6-desde-hasta  , hasta )

Torres_Hanoi(3, 1 , 3)
```

```
Lleva la ficha  1  hasta  3
Lleva la ficha  1  hasta  2
Lleva la ficha  3  hasta  2
Lleva la ficha  1  hasta  3
```

```
Lleva la ficha  2  hasta  1
Lleva la ficha  2  hasta  3
Lleva la ficha  1  hasta  3
```

1.2 Fibonacci

[ ]:
```python
#Sucesión_de_Fibonacci
#https://es.wikipedia.org/wiki/Sucesi%C3%B3n_de_Fibonacci
#Calculo del termino n-simo de la sucesión de Fibonacci
def Fibonacci(N:int):
  if N < 2:
    return 1
  else:
    return Fibonacci(N-1)+Fibonacci(N-2)

Fibonacci(5)
```

[ ]: 8

1.3 Fibonacci II (Michael)

En este caso utilizamos programación dinámica para almacenar temporalmente en memoria RAM resultados pre-calculados y así ahorrar llamadas recursivas duplicadas.

[ ]:
```python
def fibonnaci(n, cache = {0: 0, 1:1}):
    # Si el valor existe en cache significa que ha sido previamente calculado
    if(n in cache):
        return cache[n]
    else:
        # Primero se llamara a la llamada recursiva de la izquierda, así,
        # cuando se llame a la de la derecha ya estarán calculados los
        # n-1 valores de fibonacci
        cache[n] = fibonnaci(n-1, cache) + fibonnaci(n-2, cache)
        return cache[n]

print(fibonnaci(10))
```

```
55
```

##Devolución de cambio por técnica voraz

[ ]:
```python
def cambio_monedas(N, SM):
  SOLUCION = [0]*len(SM)   #SOLUCION = [0,0,0,0,..]
  ValorAcumulado = 0

  for i,valor in enumerate(SM):
    monedas =  (N-ValorAcumulado)//valor
    SOLUCION[i] = monedas
```

```
    ValorAcumulado = ValorAcumulado + monedas*valor

    if ValorAcumulado == N:
      return SOLUCION


cambio_monedas(15,[25,10,5,1])
```

[ ]: [0, 1, 1, 0]

1.4 Devolución del cambio de monedas (Michael)

Aquí tenemos una versión que no mejora la anterior a nivel de coste computacional teórico pero si que es más robusta, es decir, no presuponemos que las monedas están ordenadas, sino que damos soporte a sistema monetarios no ordenados y también se reduce el número de operaciones elementales a realizar

[ ]:
```
def get_change(quantity, money_system):
    coin_counter_dic = { }
    coins = sorted(money_system, reverse=True)
    for coin in coins:
        coin_counter_dic[coin] = quantity // coin # optimización local con␣
 ↪esperanza global.
        quantity = quantity % coin
        # Una vez procesamos una moneda la descartamos y continuamos
        # con otros candidatos

    return coin_counter_dic

# Caso similar al propuesto en AG1
case_1 = get_change(15, (25,10,5,1))
print('Caso propuesto profesor', case_1, 'Total monedas', sum(case_1.values()))
case_2 = get_change(6, (1,3,4))
print('Caso donde no encuentra óptimo global', get_change(6, (1,3,4)), 'Total␣
 ↪monedas', sum(case_2.values()))
```

```
Caso propuesto profesor {25: 0, 10: 1, 5: 1, 1: 0} Total monedas 2
Caso donde no encuentra óptimo global {4: 1, 3: 0, 1: 2} Total monedas 3
```

1.4.1 ¿Qué pasaría con sistemas monetarios alternativos?

Como se puede apreciar, en el caso 2, no se nos da la mejor solución que sería 2 monedas de 3 unidades monetarias, sino que se nos devuelve 3 monedas en total en lugar de 2. Esto es una característica de los algoritmos voraces, no piensan a nivel global sino a nivel local. Para mejorar este algoritmo, una de las posibles soluciones sería utilizar programación dinámica y apoyarse en la memoria RAM para construir una solución.

1.5 N-Reinas por técnica de vuelta atrás

[ ]:
```
def escribe(S):
  n = len(S)
  for x in range(n):
    print("")
    for i in range(n):
      if S[i] == x+1:
        print(" X " , end="")
      else:
        print(" - ", end="")


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



def reinas(N, solucion=[], etapa=0):
  if len(solucion) == 0:
      solucion=[0 for i in range(N)]

  for i in range(1, N+1):
    solucion[etapa] = i

    if es_prometedora(solucion, etapa):
      if etapa == N-1:
        print(solucion)
        #escribe(solucion)
        print()
      else:
        reinas(N, solucion, etapa+1)
    else:
      None
```

```
    solucion[etapa] = 0

reinas(8)
```

```
[1, 5, 8, 6, 3, 7, 2, 4]

[1, 6, 8, 3, 7, 4, 2, 5]

[1, 7, 4, 6, 8, 2, 5, 3]

[1, 7, 5, 8, 2, 4, 6, 3]

[2, 4, 6, 8, 3, 1, 7, 5]

[2, 5, 7, 1, 3, 8, 6, 4]

[2, 5, 7, 4, 1, 8, 6, 3]

[2, 6, 1, 7, 4, 8, 3, 5]

[2, 6, 8, 3, 1, 4, 7, 5]

[2, 7, 3, 6, 8, 5, 1, 4]

[2, 7, 5, 8, 1, 4, 6, 3]

[2, 8, 6, 1, 3, 5, 7, 4]

[3, 1, 7, 5, 8, 2, 4, 6]

[3, 5, 2, 8, 1, 7, 4, 6]

[3, 5, 2, 8, 6, 4, 7, 1]

[3, 5, 7, 1, 4, 2, 8, 6]

[3, 5, 8, 4, 1, 7, 2, 6]

[3, 6, 2, 5, 8, 1, 7, 4]

[3, 6, 2, 7, 1, 4, 8, 5]

[3, 6, 2, 7, 5, 1, 8, 4]

[3, 6, 4, 1, 8, 5, 7, 2]
```

[3, 6, 4, 2, 8, 5, 7, 1]

[3, 6, 8, 1, 4, 7, 5, 2]

[3, 6, 8, 1, 5, 7, 2, 4]

[3, 6, 8, 2, 4, 1, 7, 5]

[3, 7, 2, 8, 5, 1, 4, 6]

[3, 7, 2, 8, 6, 4, 1, 5]

[3, 8, 4, 7, 1, 6, 2, 5]

[4, 1, 5, 8, 2, 7, 3, 6]

[4, 1, 5, 8, 6, 3, 7, 2]

[4, 2, 5, 8, 6, 1, 3, 7]

[4, 2, 7, 3, 6, 8, 1, 5]

[4, 2, 7, 3, 6, 8, 5, 1]

[4, 2, 7, 5, 1, 8, 6, 3]

[4, 2, 8, 5, 7, 1, 3, 6]

[4, 2, 8, 6, 1, 3, 5, 7]

[4, 6, 1, 5, 2, 8, 3, 7]

[4, 6, 8, 2, 7, 1, 3, 5]

[4, 6, 8, 3, 1, 7, 5, 2]

[4, 7, 1, 8, 5, 2, 6, 3]

[4, 7, 3, 8, 2, 5, 1, 6]

[4, 7, 5, 2, 6, 1, 3, 8]

[4, 7, 5, 3, 1, 6, 8, 2]

[4, 8, 1, 3, 6, 2, 7, 5]

[4, 8, 1, 5, 7, 2, 6, 3]

[4, 8, 5, 3, 1, 7, 2, 6]

[5, 1, 4, 6, 8, 2, 7, 3]

[5, 1, 8, 4, 2, 7, 3, 6]

[5, 1, 8, 6, 3, 7, 2, 4]

[5, 2, 4, 6, 8, 3, 1, 7]

[5, 2, 4, 7, 3, 8, 6, 1]

[5, 2, 6, 1, 7, 4, 8, 3]

[5, 2, 8, 1, 4, 7, 3, 6]

[5, 3, 1, 6, 8, 2, 4, 7]

[5, 3, 1, 7, 2, 8, 6, 4]

[5, 3, 8, 4, 7, 1, 6, 2]

[5, 7, 1, 3, 8, 6, 4, 2]

[5, 7, 1, 4, 2, 8, 6, 3]

[5, 7, 2, 4, 8, 1, 3, 6]

[5, 7, 2, 6, 3, 1, 4, 8]

[5, 7, 2, 6, 3, 1, 8, 4]

[5, 7, 4, 1, 3, 8, 6, 2]

[5, 8, 4, 1, 3, 6, 2, 7]

[5, 8, 4, 1, 7, 2, 6, 3]

[6, 1, 5, 2, 8, 3, 7, 4]

[6, 2, 7, 1, 3, 5, 8, 4]

[6, 2, 7, 1, 4, 8, 5, 3]

[6, 3, 1, 7, 5, 8, 2, 4]

[6, 3, 1, 8, 4, 2, 7, 5]

[6, 3, 1, 8, 5, 2, 4, 7]

[6, 3, 5, 7, 1, 4, 2, 8]

[6, 3, 5, 8, 1, 4, 2, 7]

[6, 3, 7, 2, 4, 8, 1, 5]

[6, 3, 7, 2, 8, 5, 1, 4]

[6, 3, 7, 4, 1, 8, 2, 5]

[6, 4, 1, 5, 8, 2, 7, 3]

[6, 4, 2, 8, 5, 7, 1, 3]

[6, 4, 7, 1, 3, 5, 2, 8]

[6, 4, 7, 1, 8, 2, 5, 3]

[6, 8, 2, 4, 1, 7, 5, 3]

[7, 1, 3, 8, 6, 4, 2, 5]

[7, 2, 4, 1, 8, 5, 3, 6]

[7, 2, 6, 3, 1, 4, 8, 5]

[7, 3, 1, 6, 8, 5, 2, 4]

[7, 3, 8, 2, 5, 1, 6, 4]

[7, 4, 2, 5, 8, 1, 3, 6]

[7, 4, 2, 8, 6, 1, 3, 5]

[7, 5, 3, 1, 6, 8, 2, 4]

[8, 2, 4, 1, 7, 5, 3, 6]

[8, 2, 5, 3, 1, 7, 4, 6]

[8, 3, 1, 6, 2, 5, 7, 4]

[8, 4, 1, 3, 6, 2, 7, 5]

1.6 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]
]



######################################################################
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(0,N-1):
    RUTA[i][i] = i              #Para ir de i a i se "pasa por i"
    PRECIOS[i][i] = 0           #Para ir de i a i se se paga 0
    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          #Anota que para ir de i a j hay que pasar␣
 ↪por k
        PRECIOS[i][j] = MIN

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


PRECIOS,RUTA = Precios(TARIFAS)
#print(PRECIOS[0][6])

print("PRECIOS")
for i in range(len(TARIFAS)):
```

```
  print(PRECIOS[i])

print("\nRUTA")
for i in range(len(TARIFAS)):
  print(RUTA[i])

#Determinar la ruta con Recursividad
def calcular_ruta(RUTA, desde, hasta):
  if desde == hasta:
    #print("Ir a :" + str(desde))
    return ""
  else:
    return str(calcular_ruta( RUTA, desde, RUTA[desde][hasta])) +  \
                ',' + \
                str(RUTA[desde][hasta] \
              )

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

```
PRECIOS
[0, 5, 4, 3, 8, 8, 11]
[9999, 0, 999, 2, 3, 8, 7]
[9999, 9999, 0, 1, 6, 4, 7]
[9999, 9999, 9999, 0, 5, 6, 9]
[9999, 9999, 9999, 9999, 0, 999, 4]
[9999, 9999, 9999, 9999, 9999, 0, 3]
[9999, 9999, 9999, 9999, 9999, 9999, 9999]

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

La ruta es:
```

```
[ ]: ',0,2,5'
```

1.7 Encontrar los dos puntos más cercanos

1.7.1 Primer intento: Fuerza bruta

En el caso de aplicar fuerza bruta tendríamos el siguiente fragmento de código

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

def brute_force(points):
  best_distance = abs(points[0] - points[1])
  point_A =  {
      "index": 0,
      "value": points[0]
  }

  point_B = {
      "index": 1,
      "value": points[1]
  }

  for idx, point in enumerate(points):
    for idx2, point2 in enumerate(points[idx + 1:]):
       if (abs(point - point2) < best_distance):
         point_A = {
             "index": idx,
             "value": point
         }
         point_B = {
             "index": idx + idx2,
             "value": point2
         }
         best_distance = abs(point - point2)

  return {best_distance}, point_A, point_B


print(brute_force([1,10,20,15,25,19, 5]))
```

```
({1}, {'index': 2, 'value': 20}, {'index': 4, 'value': 19})
```

[ ]:
```python

```

1.7.2 Coste temporal de fuerza bruta

Podemos sacar las siguientes conclusiones:

- El primer bucle for itera sobre cada elemento de la lista points. Si la lista tiene “n” elementos, esta bucle se ejecuta “n” veces.
- El segundo bucle itera sobre una sublista de elementos de tal forma que en cuando idx es cero, iterara sobre n-1 elementos, cuando idx es 1, iterara sobre n-2 elementos, etc. Así evitamos hacer comparaciones previamente hechas (punto A comparada con B y luego comparar punto B con A)

Por lo tanto podemos concluir que el coste temporal por fuerza bruta será de $\mathcal{O}(n^2)$.

Además, en el caso extensible de 2D y 3D tendrá el mismo coste asintótico temporal dado que lo único que cambia es la manera de calcular la distancia, no el algoritmo y la estrategia en si misma.

1.7.3 Caso 1D con estrategia de divide y vencerás

En el siguiente fragmento de código se muestra una estrategia recursiva para resolver el mismo problema

[ ]:
```
import math

def calc_distance(pointA, pointB):
    if isinstance(pointA, int):
        return abs(pointA - pointB)
    elif len(pointA) == 2:
        delta_x = pointB[0] - pointA[0]
        delta_y = pointB[1] - pointA[1]
        return math.sqrt(delta_x**2 + delta_y**2)
    elif len(pointA) == 3:
        delta_x = pointB[0] - pointA[0]
        delta_y = pointB[1] - pointA[1]
        delta_z = pointB[2] - pointA[2]
        return math.sqrt(delta_x**2 + delta_y**2 + delta_z**2)

def trivial_case(points):
    min_dist = float('inf')
    best_pair = (None, None)

    # Como maximo dara 3 vueltas porque a esta funcion solo se llama
    # con listados de casos triviales, por lo que en el coste temporal
    # asintótico no se tendrá en cuenta (las constantes son absorvidas por
    # el termino dominante)
    for i in range(len(points)):
        for j in range(i + 1, len(points)):
            d = calc_distance(points[i], points[j])
            if d < min_dist:
                min_dist = d
                best_pair = (points[i], points[j])

    return {
        "distance": min_dist,
        "A": best_pair[0],
        "B": best_pair[1]
    }


"""
```

```
   Damos por hecho que el listado de puntos que recibimos esta ordenado
   si esto no es así, entonces consideraremos este algoritmo erroneo.
"""
def shorter_distance(points):
    if len(points) <= 3:
        return trivial_case(points)

    mid = len(points) // 2
    border_distance = {
        "distance": calc_distance(points[mid-1], points[mid]),
        "A": points[mid-1],
        "B": points[mid]
    }
    left_best_distance = shorter_distance(points[:mid])
    right_best_distance = shorter_distance(points[mid:])
    # Retornar el mejor entre los dos
    return min([border_distance, left_best_distance, right_best_distance],␣
 ↪key=lambda x: x["distance"])

# Para 1D es importante ordenar primero

list_A= [random.randrange(1,10000) for x in range(1000)]
sorted_list = sorted(list_A)
#sorted_list = [1,5,4,1,2,46,2,12,41,35,6,13,124,126,5,1]
#sorted_list = [(0,0), (1,2), (4,5), (10,11), (1,1)]
#sorted_list = [(0,0,0), (1,1,1), (0,0,1)]
result = shorter_distance(sorted_list)
print(result)
```

```
{'distance': 0, 'A': 322, 'B': 322}
```

Nota: La probabilidad e que caigan duplicados es alta porque el listado propuesto en la diapositivas es muy alta, si se prueba con listados más cortos se obtendrán resultados diferentes a distancia 0.

1.8 Coste asintótico temporal de calculo de distancia mínima en 1D

Sea la recurrencia:

$$T(n) = \begin{cases} c \cdot n^k & \text{si } n < b \\ a \cdot T(\frac{n}{b}) + c \cdot n^k & \text{si } n \geq b \end{cases}$$

Entonces, según el Teorema Maestro:

- Si $a < b^k$, entonces $T(n) = \mathcal{O}(n^k)$
- Si $a = b^k$, entonces $T(n) = \mathcal{O}(n^k \log n)$
- Si $a > b^k$, entonces $T(n) = \mathcal{O}(n^{\log_b a})$

Donde:

- (a): número de subproblemas

- (b): factor por el que se reduce el tamaño en cada llamada
- (k): coste del trabajo fuera de la recursión

Podemos deducir que “b” será igual a 2, porque en cada llamada recursiva el problema se dividirá en dos sub-partes. Además, el número de llamadas recursivas que se invocan en la función son 2 también, por lo que a = 2.

Adicionalmente, “k” lo podemos considerar 1 porque las operaciones fuera de la recursión incluyen hacer un “slicing” de los listados con las instrucciones python “points[:mid]” y “points[mid:]” tiene coste lineal $\mathcal{O}(n)$

De todo esto podremos deducir que al ser $a = b^k (2 = 2)$ el coste temporal será de $\mathcal{O}(n \cdot log(n))$

1.9 Conclusiones finales

Aunque inicialmente se intentó aplicar una solución general para los casos 2D y 3D utilizando la estrategia de divide y vencerás, el algoritmo propuesto solo funciona correctamente cuando no hay conflictos entre puntos cercanos a la frontera. Esto no ocurre en el caso 1D porque la lista está previamente ordenada, lo que permite comparar directamente los puntos vecinos a ambos lados de la división.

Sin embargo, en 2D y 3D el criterio de ordenación se vuelve más complejo. Para resolver correctamente el problema de la frontera en estos casos, es necesario aplicar estrategias más sofisticadas. Aunque el algoritmo calcula correctamente la distancia mínima en cada mitad, falla al no evaluar adecuadamente las posibles distancias mínimas entre puntos cercanos que están en lados opuestos de la división.

En resumen, la estrategia de divide y vencerás resulta eficiente para encontrar la menor distancia entre puntos en una dimensión (1D), gracias a que la ordenación previa permite manejar correctamente los casos frontera. Sin embargo, al extender el algoritmo a espacios de mayor dimensión (2D o 3D), surgen nuevos desafíos: la complejidad en el criterio de ordenación y la gestión de los puntos cercanos a la división central.