Universidad de Alcalá

Escuela Politécnica Superior

Universidad de Alcalá

Algoritmia y Complejidad

Laboratorio – Sesión 3

Divide y vencerás

Laboratorio Jueves 12:00 – 14:00

Grado en Ingeniería Informática – Curso 2018/2019

Eduardo Graván Serrano – 03212337L

Marcos Barranquero Fernández – 51129104N

Ejercicio 4: Robot

Enunciado:

Se quiere programar un robot para poner tapones de corcho a las botellas de una fábrica de reciclado. Se tienen disponibles N botellas y los N corchos que las tapan (N es constante en el problema), pero hay una serie de problemas:

• Las botellas son todas distintas entre sí, igual que los corchos: cada botella solo puede cerrarse con un corcho concreto, y cada corcho solo sirve para una botella concreta.

• El robot está preparado para cerrar botellas, por lo que lo único que sabe hacer es comparar corchos con botellas. El robot puede detectar si un corcho es demasiado pequeño, demasiado grande, o del tamaño justo para cerrar una botella.

• El robot no puede comparar corchos entre sí para "ordenarlos" por grosor, y tampoco puede hacerlo con las botellas.

• El robot tiene espacio disponible y brazos mecánicos para colocar botellas y corchos a su antojo, por ejemplo en distintas posiciones de una mesa, si es necesario.

Diseñar el algoritmo que necesita el robot para taponar las N botellas de manera óptima.

Código

```
def ordenar_corchos(corchos, botellas, indice_pivote):
    """ Dados un array de corchos y botellas (numéricos),
    ordena los corchos de menor a mayor usando como pivotes las
botellas.
    Para cada pivote se hace una llamada al propio método.
    """

    # Si ya se ha pivotado sobre todos los elementos de botellas,
    # se devuelven los corchos ordenados:
    if(indice_pivote >= len(botellas)):
        return corchos
    # Si no, se realiza el algoritmo:
    else:
        # Pivote de botellas
        pivote = botellas[indice_pivote]

        # Pivote en corcho contiene el índice del pivote en corchos
        indice_pivote_en_corchos = corchos.index(pivote)

        # Este índice contiene sobre qué elemento estamos pivotando
en corchos:
        indice_recorrido = 0

        # Recorro todos los corchos:
        while(indice_recorrido < len(corchos)):

            # Si el elemento es mayor y estoy a la izquierda:
```

```
            if(corchos[indice_recorrido] > pivote and
indice_recorrido < indice_pivote_en_corchos):
                valor = corchos[indice_recorrido]
                # Saco la parte de la izquierda
                seccion_izquierda =
corchos[0:indice_pivote_en_corchos+1]
                # Elimino el elemento mayor a la izquierda del
pivote
                seccion_izquierda.remove(valor)
                # Saco la parte derecha
                seccion_derecha =
corchos[indice_pivote_en_corchos+1:]
                # Coloco el elemento mayor a la derecha del pivote
                corchos = seccion_izquierda + [valor] +
seccion_derecha
                # Actualizo el indice del pivote en caso de que
haya cambiado de sitio:
                indice_pivote_en_corchos = corchos.index(pivote)
            # Si el elemento es menor y estoy a la derecha:
            elif(corchos[indice_recorrido] < pivote and
indice_recorrido >= indice_pivote_en_corchos):
                # Valor a reordenar:
                valor = corchos[indice_recorrido]
                # Saco la parte de la izquierda
                seccion_izquierda =
corchos[0:indice_pivote_en_corchos]
                # Saco la parte derecha
                seccion_derecha =
corchos[indice_pivote_en_corchos:]
                # Elimino el elemento menor a la derecha del pivote
                seccion_derecha.remove(valor)
                # Coloco el elemento mayor a la izquierda del
pivote
                corchos = seccion_izquierda + [valor] +
seccion_derecha
                # Actualizo el indice del pivote en caso de que
haya cambiado de sitio:
                indice_pivote_en_corchos = corchos.index(pivote)
            # Si el elemento está correctamente ordenado:
            else:
                # Se avanza al siguiente elemento:
                indice_recorrido += 1
                indice_pivote_en_corchos = corchos.index(pivote)
        return ordenar_corchos(corchos, botellas, indice_pivote+1)
```

```
# Listados de ejemplo

botellas = [1, 4, 6, 8, 3, 7]
corchos = [6, 8, 4, 1, 7, 3]
botellas2 = [números aleatorios que ocupan mucho…]
corchos2 = [números aleatorios que ocupan mucho…


# Programa principal
print("Corchos: ", ordenar_corchos(corchos2, botellas2, 0))
print("Botellas: ", ordenar_corchos(botellas2, corchos2, 0))
```

Explicación:

Consideramos dos arrays de corchos y botellas, ambos con valores numéricos. El objetivo es que queden igualmente ordenados.

Nuestra solución ha resultado ser una adaptación del método de ordenación Quicksort.

A grandes rasgos, lo que se hace es ordenar un vector pivotando sobre otro.

El método está escrito de forma que se ordenan corchos de menor a mayor, pivotando sobre botellas, aunque por la naturaleza numérica de los arrays, el método es reversible.

Por tanto, se ordenarán primero un array respecto al otro, y después el otro sobre el uno, de forma que al finalizar quedan ambos igual ordenados, es decir, los corchos “alineados” con las botellas.

Para ordenar, se siguen los siguientes pasos:

1. Se recibe un índice pivote que irá de 0 a la longitud del array.
2. Se toma el elemento de ese índice pivote como pivote. Por ejemplo, si el índice pivote es 3, se tomará como pivote botellas[3].
3. Una vez tenemos el pivote sobre botellas, ordenaremos sobre corchos.
4. Recorreremos el array de corchos y para cada corcho:
    a. Si está a la izquierda y es mayor del pivote, reordenamos tal que: Array[comienzo:in_pivote+1(sin el elemento)][elemento]Array[pivote+1:final]
    b. Si está a la derecha y es menor que el pivote, reordenamos tal que: Array[comienzo:in_pivote]-[elemento]-Array[in_pivote:final]
    c. Si el elemento está correctamente ordenado, pasamos al siguiente.
5. Aumentamos el índice pivote en 1 (es decir, pivotamos sobre el siguiente elemento en botellas) y llamamos recursivamente a la función para realizar el reordenamiento pivotando sobre el nuevo pivote.

Ejercicio 6: Indiana Croft

Enunciado:

Tras su paso por la Sala de las Baldosas y conseguir la Cuna de la Vida, ahora Indiana Croft se enfrenta a un nuevo desafío antes de poder salir del Templo Maldito. Se encuentra en un puente bajo el que se observa una profunda oscuridad. Afortunadamente, este lugar también aparece en el diario. El puente cruza el llamado Valle de Sombras, que empieza con una pendiente de

bajada (la pendiente no es necesariamente constante) para después de llegar al punto más bajo empezar a subir hasta el otro extremo del puente (de nuevo, no necesariamente con pendiente contante). Justo en la parte inferior del valle hay un río, pero el diario no especifica su ubicación con respecto al puente (por ejemplo, no se sabe si el río está a 53 metros desde el comienzo del puente) ni la distancia en altura (es decir, no se sabe si el río está 228 metros por debajo, por ejemplo). En las pendientes hay afiladísimas rocas. Si Indiana Croft tuviese tiempo, podría encontrar sin problema el punto por donde descolgarse del puente para llegar exactamente al río, ya que tiene un puntero laser para medir alturas que le dice cuántos metros hay desde el puente hasta el suelo en un punto determinado. El problema es que los sacerdotes del templo han averiguado que les han robado la Cuna de la Vida, están persiguiendo a Indiana Croft y le alcanzarán enseguida. El aventurero debe encontrar rápidamente la posición del río para descolgarse y huir seguro. Diseñar el algoritmo que debería usar Indiana Croft para buscar el punto mínimo del valle en las condiciones indicadas. El algoritmo debe ser eficiente: al menos en el mejor caso debe tener un orden logarítmico. Se puede considerar el tiempo que tarda Indiana Croft en desplazarse a lo largo del puente es nulo y que la estimación del punto del río por donde descolgarse puede tener un error de aproximación de ε metros (ε es una constante dada).

Código:

```
def funcion(x):
    # return (((x+10)**2))-20
    # return (((x+50)**2))-200
    return x**2


def valle_minimo_fun(principio, final):
    """ Dado el principio y el final del puente, y
    una funcion auxiliar que define la curva, retorna
    la posición (X) donde se encuentra la profundidad (Y)
    más profunda (la mayor)"""

    # Calculo distancia del espacio a estudiar:
    distancia = final - principio

    # Si es menor que 3, devuelvo el punto de mayor profundidad
    if(distancia < 3):
        # Valor mínimo retornado por la función
        posicion_minimo = principio
        for i in range(principio, final+1):
            if(funcion(i) < funcion(posicion_minimo)):
                posicion_minimo = i
        return posicion_minimo

    # Si no, calculo el tamaño de un tercio:
    tercio = distancia//3

    # Calculo el último elemento del primer tercio:
    elemento1 = funcion(principio + tercio)

    # Calculo el primer elemento del último tercio
    elemento2 = funcion(principio + 2*tercio)

    # Si los dos primeros tercios son más profundos, descarto el
tercero
    if(elemento1 <= elemento2):
        return valle_minimo_fun(principio, final-tercio)

    # Si los dos segundos tercios son más profundos, descarto el
primero
    elif(elemento1 > elemento2):
        return valle_minimo_fun(principio+tercio, final)


print(valle_minimo_fun(-10, 100))
```

Explicación:

Se define un método llamado “función”, que dado un numero por argumento, devuelve el resultado de una función matemática con ese número como parámetro.

El método principal busca calcular el mínimo de esa función en un rango definido por los argumentos. En relación al enunciado, el método función representa el láser de indiana croft, el argumento es la posición en el puente y devuelve la distancia al suelo desde ese punto.

Consideramos que la función es continuamente descendente hacia el punto mínimo, y, desde ahí, continuamente ascendente.

Por tanto, dividiremos el rango en 3 subrangos, y para cada uno de estos subrangos, mediremos la distancia al suelo en los puntos comunes entre ellos. De esta forma, podremos comparar las distancias retornadas en estos puntos. De los tres rangos, nos quedaremos con los dos que comparten un punto cuya distancia al suelo es mayor (es decir, más cercana al mínimo). Y llamaremos al método recursivamente con ese subrango de parámetro.

Hemos tomado 3 como valor de tope, imaginando que el río será de 3 metros. Una vez la distancia entre el principio y el final es menor a un umbral mínimo establecido, se devuelve la coordenada cuyo valor de la función es el mínimo. Esto es, el sitio en el puente desde el cual Indiana debe de saltar.