Universidad Rey Juan Carlos

Prueba de Diseño de Algoritmos – Curso 2016-2017

Diseño y Análisis de Algoritmos

Valor: 35 % de la nota final. Duración: **2 horas y media**

Soluciones

Ejercicio 1 [1 punto]

Dado un laberinto, se desea encontrar un camino que no solo sea una solución, sino que además sea de longitud más corta. Este problema se puede resolver con la técnica de *backtracking*, pero en este ejercicio se pide una solución mediante un algoritmo **voraz**. La solución consiste en aplicar una estrategia de "transforma y vencerás", para poder aplicar uno de los algoritmos voraces vistos en clase.

Se pide describir cómo se puede resolver el problema concreto (cómo se transformaría el problema, y qué algoritmo voraz se aplicaría). No hay que escribir código. La descripción será muy breve: 5 líneas como máximo.

Solución:

Primero se genera un grafo asociado al laberinto. En concreto, tendrá un nodo por cada celda vacía en el laberinto. En cuanto a las aristas, habrá una entre dos nodos si sus respectivas celdas son adyacentes. Finalmente, el camino más corto a través del laberinto es el camino más corto a través del grafo. Por tanto, el problema se puede resuelver aplicando el algoritmo de **Dijkstra**.

Ejercicio 2 [2 puntos]

Considera un número entero no negativo a cuyos dígitos aparecen ordenados orden creciente desde el más significativo hasta el menos (por ejemplo, $a = 245778$). Dado un dígito x, implementa una función que devuelva un nuevo número resultante de insertar x en a, de manera que los dígitos también queden ordenados en orden creciente. Ejemplos:

- $a = 245778$, $x = 0 \longrightarrow 245778$
- $a = 245778$, $x = 1 \longrightarrow 1245778$
- $a = 245778$, $x = 6 \longrightarrow 2456778$
- $a = 245778$, $x = 9 \longrightarrow 2457789$

Posible solución:

```
def inserta_en_entero_creciente(x,a):
    if x>=a%10:  # caso base
        return a*10 + x
    else:  # caso recursivo
        return a%10 + 10*inserta_en_entero_creciente(x,a//10)
```

Ejercicio 3 [3 puntos]

Sea **a** una lista ordenada de n enteros (pueden ser negativos) todos distintos. Se pide diseñar un algoritmo de complejidad $\mathcal{O}(\log n)$ en el peor caso, capaz de encontrar un índice i tal que $a[i] = i$, suponiendo que tal índice i exista (donde $0 \leq i \leq n-1$). El algoritmo devolverá el valor i en caso de que exista, y -1 en caso contrario.

Posible solución:

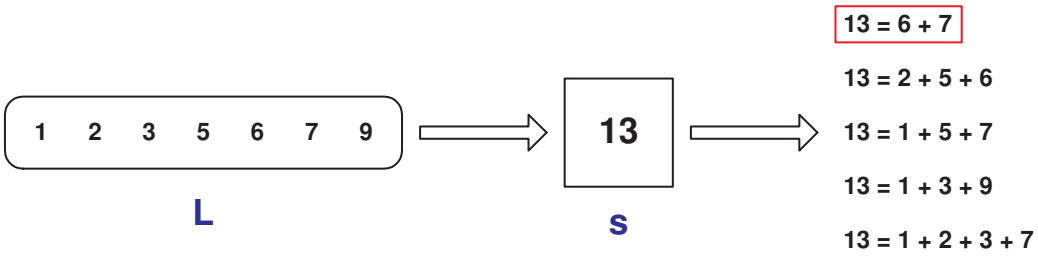
```
def elemento_en_posicion(a, inf, sup):
    mitad = (inf+sup)//2

    if mitad==a[mitad]:  # Caso base 1
        return mitad
    elif inf>=sup:  # Caso base 2
        return -1
    elif mitad<a[mitad]:  # Casos recursivos
        return elemento_en_posicion(a, inf, mitad-1)
    else:
        return elemento_en_posicion(a, mitad+1, sup)
```

Ejercicio 4 [4 puntos]

Se pide implementar un algoritmo basado en la técnica de *backtracking* para resolver el siguiente problema.

Dada una lista L de números enteros no negativos (almacenada en un simple array/lista), y otro número entero no negativo s, se desea hallar un conjunto de números de L tal que su suma sea igual a s, y que además dicho conjunto sea el de menor cardinalidad de entre todos los posibles que cumplan la restricción asociada a la suma. El conjunto se especificará mediante un vector de booleanos de la misma longitud que L.

En el siguiente ejemplo hay varios conjuntos de elementos de L tal que su suma sea igual a 13 (s). El algoritmo devolvería el conjunto $\{6, 7\}$ al ser el de menor cardinalidad (es el que tiene el menor número de elementos):



NOTA: la solución no tiene por qué ser única. Si hay varios conjuntos que cumplan que la suma de sus elementos es s, y además tienen la misma mínima cardinalidad, basta con devolver cualquiera de esos conjuntos.

Posible solución:

```
def busca_subconj_aux(L,s):
    sol_parcial = [False] * (len(L))
    sol_opt = [False] * (len(L))

    card_opt = busca_subconj(0,0,L,s,sol_parcial,sol_opt,len(L)+1,0)

    print(card_opt)
    print(sol_opt)
```

```
def busca_subconj(i,suma_parcial,L,s,sol_parcial,sol_opt,card_opt,card):

    # genera candidatos (esquema para búsqueda de SUBCONJUNTOS)
    for k in range(0,2):

        nueva_suma_parcial = suma_parcial + k*L[i]

        # Comprueba si se puede podar el árbol de recursión
        if nueva_suma_parcial<=s:

            # Actualiza solución parcial
            sol_parcial[i] = (k==1)

            # Incrementa cardinalidad si se incluye el candidato
            if k==1:
                card = card+1

            # Comprueba si se ha alcanzado el valor en s
            if nueva_suma_parcial==s:
                # Actualiza solución y valor óptimos si menor cardinalidad
                if card<card_opt:
                    card_opt = card
                    for j in range(0,len(L)):
                        sol_opt[j] = sol_parcial[j]

            elif i<len(L)-1 and card<card_opt-1:
                card_opt = busca_subconj(i+1,nueva_suma_parcial,L,s,
                    sol_parcial,sol_opt,card_opt,card)

    sol_parcial[i] = False

    return card_opt
```