Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos – Grado en Ingeniería Informática – Móstoles

Curso 2019-2020 – 4 de junio de 2020

Diseño de algoritmos - Soluciones

Ejercicio 1 [**2 puntos**] [Duración: **20 minutos**]

Implementa un algoritmo recursivo que reciba una lista de elementos (se pueden considerar números reales) y genere dos listas. La primera contendrá los elementos que se encuentren en índices pares, mientras que la segunda contendrá los elementos que se encuentran en índices impares. Suponed que el primer índice de una lista es el 0. Por ejemplo, dada la lista $\mathbf{a} = [3, 6, 5, 7, 0, 9, 2]$, el algoritmo generará las listas:

$$\mathbf{a}_{\text{par}} = [3, 5, 0, 2] \qquad \text{y} \qquad \mathbf{a}_{\text{impar}} = [6, 7, 9]$$

Posible solución:

En Python se pueden devolver varias salidas simultáneamente. El siguiente código devuelve las dos listas mediante una "tupla" (solo hay que especificar las salidas entre paréntesis y separadas por comas). Si la lista de entrada es vacía se devuelven dos listas vacías. Si tiene un elemento la lista de índices pares será la propia lista, mientras que la de impares será vacía. En el caso recursivo se extraen las listas de índices pares e impares de toda la lista exceptuando al último elemento (mediante una llamada recursiva). Después, lo único que falta es concatenar el último elemento a la lista de índices pares o impares, dependiendo de la paridad de la longitud de la lista original.

```
def extrae_indices_pares_impares(a):
    n = len(a)
    if n==0:
        return ([],[])
    elif n==1:
        return (a,[])
    else:
        (lista_pares, lista_impares) = extrae_indices_pares_impares(a[:n-1])

        if n%2==0:
            return (lista_pares, lista_impares + [a[n-1]])
        else:
            return (lista_pares + [a[n-1]], lista_impares)
```

Ejercicio 2 [**3 puntos**] [Duración: **30 minutos**]

Implementa un algoritmo basado en la estrategia de **DIVIDE Y VENCERÁS** para evaluar un polinomio P de grado $n-1$:

$$P = c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$$

en un determinado valor real x. Evaluar un polinomio simplemente consiste en determinar qué valor toma el polinomio para un valor de x concreto. Por ejemplo, si $P = 2x^3 - x + 2$, evaluarlo en $x = 1$ sería calcular $P(1) = 2 \cdot (1)^3 - 1 + 2 = 3$.

El polinomio se definirá mediante una lista $\mathbf{c} = [c_0, c_1, \ldots, c_{n-1}]$ de longitud n, que contiene los coeficientes (reales) del polinomio. Además, el algoritmo de divide y vencerás se basará necesariamente en el método descrito en el **Ejercicio 1** para dividir la lista **c** en dos:

$$\mathbf{a}_{\text{par}} = [c_0, c_2, \ldots] \qquad \text{y} \qquad \mathbf{a}_{\text{impar}} = [c_1, c_3, \ldots]$$

Se puede asumir que habéis implementado correctamente el método del Ejercicio 1 (podéis hacer llamadas al método como si estuviese implementado en alguna librería).

Posible solución:

Si el polinomio es una constante **c** tendrá longitud 1, y el resultado siempre será c_0. En caso contrario el polinomio se descompone en dos según los índices pares e impares. Por ejemplo, supongamos que deseamos evaluar $P(x) = 2x^6 + 9x^5 + 0x^4 + 7x^3 + 5x^2 + 6x + 3$ en un valor de x. La lista asociada al polinomio $\mathbf{c} = [3, 6, 5, 7, 0, 9, 2]$ primero se divide en:

$$\mathbf{a}_{\text{pares}} = [3, 5, 0, 2] \qquad \text{y} \qquad \mathbf{a}_{\text{impares}} = [6, 7, 9]$$

Esta descomposición daría lugar a dos subproblemas asociados a la evaluación de los polinomios:

$$P_{\text{pares}}(x) = 2x^3 + 0x^2 + 5x^1 + 3 \qquad \text{y} \qquad P_{\text{impares}}(x) = 9x^2 + 7x^1 + 6$$

Para obtener el algoritmo de divide y vencerás hay que pensar en cómo podemos modificar y combinar las soluciones a estos subproblemas (que obtenemos mediante llamadas recursivas). Para este problema si evaluamos P_{pares} y P_{impares} en x^2 obtendríamos:

$$P_{\text{pares}}(x^2) = 2x^6 + 0x^4 + 5x^2 + 3 \qquad \text{y} \qquad P_{\text{impares}}(x^2) = 9x^4 + 7x^2 + 6$$

cuya suma es casi $P(x)$. Lo único que faltaría sería multiplicar el polinomio de índices impares por x. Por tanto, podemos recuperar $P(x)$ de la siguiente manera, lo cual nos indica la regla recursiva:

$$P(x) = P_{\text{pares}}(x^2) + x \cdot P_{\text{impares}}(x^2).$$

```python
def evalua_polinomio(c, x):
    if len(c)==1:
        return c[0]
    else:
        (cp,ci) = extrae_indices_pares_impares(c)

        return evalua_polinomio(cp, x*x) + x*evalua_polinomio(ci, x*x)
```

Ejercicio 3 [**5 puntos**] [Duración: **50 minutos**]

Implementa un algoritmo basado en la técnica de **BACKTRACKING** para generar permutaciones *con repetición*. Los datos de entrada son una lista $\mathbf{a} = [a_0, a_1, \ldots, a_{m-1}]$ de m elementos distintos, y otra lista $\mathbf{r} = [r_0, r_1, \ldots, r_{m-1}]$, también de longitud m, que indica el número de veces que un elemento de $\mathbf{a}$ debe aparecer en cada permutación. En concreto, r_i indica el número de repeticiones de a_i en una permutación. Esto implica que las permutaciones tendrán $n = r_0 + r_1 + \cdots + r_{m-1}$ elementos.

Por ejemplo, para $\mathbf{a}$ = ['a','b','c'] y $\mathbf{r} = [1, 2, 1]$, generaremos permutaciones de $n = 4$ elementos en las que 'a' y 'c' aparecerán una vez, y 'b' dos veces. En este ejemplo las permutaciones con repetición son:

c b b a
c b a b
c a b b
b c b a
b c a b
b b c a
b b a c
b a c b
b a b c
a c b b
a b c b
a b b c

El método a desarrollar debe imprimir todas las permutaciones con repetición. Además, debe calcular el número P de permutaciones generadas, a medida que las va creando. Podéis verificar si el algoritmo está generando P correctamente ya que:

$$P = \frac{n!}{r_0! \cdot r_1! \cdot \ldots \cdot r_{m-1}!}$$

En el ejemplo, $P = 4!/(1! \cdot 2! \cdot 1!) = 12$.

Se valorará el uso de estructuras de datos para acelerar la comprobación de validez de candidatos/soluciones parciales.

Posible solución:

La solución es muy parecida al algoritmo para generar permutaciones (sin repetición) visto en teoría (transparencias/libro). En ese método se usaba una lista de valores Booleanos que indicaban si un elemento todavía no se había utilizado en la permutación (es decir, si estaba libre). Esa lista también se puede interpretar como una lista binaria, donde un 0 indica que ya no puedes usar un elemento, mientras que un 1 indica que se puede utilizar una vez. Es decir, podemos interpretar que la lista indica el número de veces que puedes usar un elemento. Pues bien, para generar permutaciones con repetición el algoritmo es prácticamente idéntico, pero esa lista podrá tener valores mayores que 1. En concreto, inicialmente contendrá el número de repeticiones de los elementos. Es decir, la lista será precisamente $\mathbf{r}$. Si la vamos modificando a medida que avance el algoritmo la condición de validez es simplemente $r_k > 0$ para el k-ésimo elemento. Además, debemos decrementar r_k si incorporamos el elemento k a la solución parcial, y debemos incrementar r_k al retornar de la llamada recursiva. Por último, en el caso base se imprime una permutación y se devuelve un 1. En el caso recursivo se acumulan los resultados de las llamadas en la variable s, por lo que el método acaba devolviendo el número de permutaciones halladas.

```python
def genera_permutaciones_repeticion(i,sol,r,a):
    n = len(sol)
    m = len(a)

    # Caso base
    if i==n:
        for i in range(0,n):
            print(sol[i],' ',end='')
        print()

        return 1
    else:
        s = 0  # Número de permutaciones halladas

        # Genera candidatos (todos los posibles elementos)
        for k in range(m):

            # Comprueba su validez
            if r[k]>0:

                # Incluye candidato en solución parcial
                sol[i] = a[k]

                # Decrementa el número de apariciones del candidato k
                r[k] = r[k]-1

                # Expande la solución parcial a partir de la posición i+1
                s = s + genera_permutaciones_repeticion(i+1,sol,r,a)

                # Incrementa el número de apariciones del candidato k
                r[k] = r[k]+1

        return s  # Devolvemos el valor acumulado en s

def genera_permutaciones_repeticion_wrapper(a,r):
        n = 0
        for i in range(len(r)):
            n = n + r[i]

        sol = [None] * n
        nsols = genera_permutaciones_repeticion(0,sol,r,a)
        print('Permutaciones con repetición halladas = ',nsols)

        # Comprobación opcional
        m = len(r)
        p = 1
        for i in range(m):
            p = p * factorial(r[i])
        print('Permutaciones con repetición correctas = ',
              int(factorial(n)/p))

# Ejemplo
genera_permutaciones_repeticion_wrapper(['a','b','c'],[1,2,1])
```