Problemas 1

Enunciados

ALGORÍTMICA Y COMPLEJIDAD
Grados en Ing. Informática y Matemáticas
Universidad de Cantabria

Camilo Palazuelos Calderón

Objetivos

1. Resolver problemas de las clases de complejidad P y NP-completo
2. Elegir y aplicar la mejor estrategia algorítmica para cada problema

Duración y fecha de entrega

- Las dos sesiones destinadas a los problemas son las de los días 26/27 de marzo y 9/10 de abril
- Los problemas deben entregarse, a través de la plataforma Moodle, antes del 14 de abril a las 23:59

Qué debéis entregar

- Memoria con respuestas a las preguntas formuladas en este documento

---

Problema 1 [2 PUNTOS]. Dado un array $A[1..n]$ de elementos sumables, el problema del subarray de suma máxima consiste en identificar el subarray $A[i..j]$ tal que la suma de sus elementos sea máxima, donde $1 \le i \le j \le n$. Por ejemplo, en el array de números enteros

$A[1..6]$ =

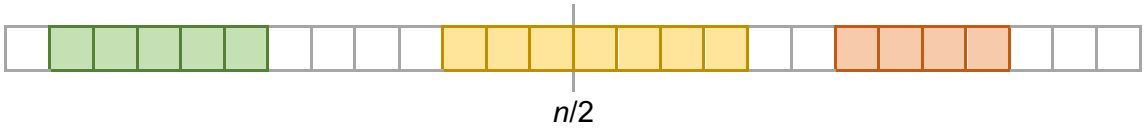
10	-19	18	3	-20	22

,

el subarray de suma máxima es $A[3..6]$, ya que $18 + 3 + (-20) + 22 = 23$ es mayor que cualquier otra suma de elementos contiguos de $A[1..6]$. El coste temporal del algoritmo que resuelve el problema del subarray de suma máxima por fuerza bruta, expresado en cantidad de sumas en términos asintóticos, es $O(n^2)$.

Diseñad un algoritmo que, dado un array $A[1..n]$, resuelva el problema del subarray de suma máxima y cuyo coste temporal sea $\Theta(n \log n)$ en el caso peor. Para ello, el algoritmo deberá dividir el array en dos mitades y hacer dos llamadas recursivas: una con la mitad izquierda y otra con la mitad derecha. El subarray

de suma máxima será el que las llamadas recursivas encuentren en la mitad **izquierda**, en la mitad **derecha** o el que resulte de la fusión de los **extremos** de las mitades:



Problema 2 [3 PUNTOS]. Decimos que *m* es el elemento mayoritario de un array *A*[1..*n*] cuando el número de veces que *m* aparece en *A*[1..*n*] es estrictamente mayor que *n*/2. El algoritmo MAYORITARIO, cuyo diseño se muestra a continuación, devuelve el elemento mayoritario del array *A*[1..*n*] en caso de que exista; si no, devuelve NULL.

```
MAYORITARIO(A[1..n]):
    B[1..n] ← A[1..n]
    m ← CANDIDATO(B[1..n])
    if m ≠ NULL
        m ← COMPROBAR(A[1..n], m)
    return m

COMPROBAR(A[1..n], m):
    k ← 0
    for i ← 1 to n
        if A[i] = m
            k ← k + 1
    if k > ⌊n/2⌋
        return m
    else
        return NULL
```

```
CANDIDATO(A[1..n]):
    if n = 0
        return NULL
    else if n = 1
        return A[n]
    else
        k ← 0
        for j ← 1 to n – 1 by 2
            if A[j] = A[j + 1]
                k ← k + 1
                A[k] ← A[j]
        m ← CANDIDATO(A[1..k])
        if m = NULL and n es impar
            m ← A[n]
        return m
```

a) [2 PUNTOS] Siguiendo la transformación de diseños recursivos a iterativos más adecuada, diseñad un algoritmo CANDIDATO iterativo cuyo coste temporal sea asintóticamente equivalente al de su homónimo recursivo.

b) [1 PUNTO] Sin aplicar el teorema maestro, calculad el coste temporal del algoritmo CANDIDATO en el caso peor. *Pista:* La serie de la secuencia de los recíprocos de las potencias de 2 converge, de ahí que

$$\sum_{i=0}^{\infty} \frac{n}{2^i} = 2n \ .$$

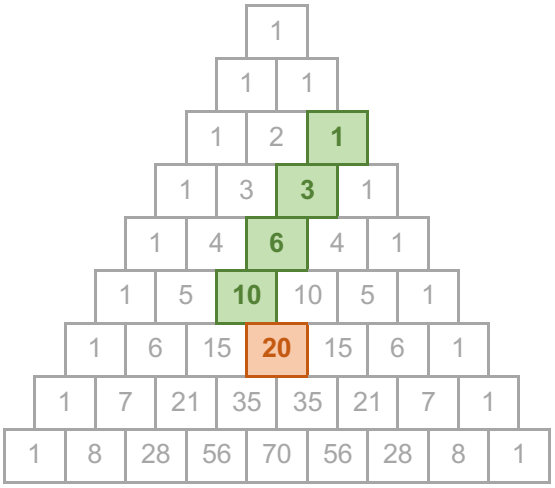
Problema 3 [1 PUNTO]. El ordenamiento chiflado (en inglés, STOOGESORT) es un algoritmo de ordenamiento de arrays excepcionalmente ineficiente. Debe su nombre a Los Tres Chiflados, un grupo de actores cómicos estadounidenses del siglo XX, y su diseño, que se muestra a continuación, refleja por qué.

```
STOOGESORT(A[1..n]):
    if A[1] > A[n]
        A[1] ↔ A[n]
    if n > 2
        m ← ⌊n/3⌋
        STOOGESORT(A[1..n – m])
        STOOGESORT(A[m + 1..n])
        STOOGESORT(A[1..n – m])
```

Calculad el coste temporal del algoritmo STOOGESORT en el caso peor, indicando claramente cómo lo habéis calculado y por qué.

Problema 4 [4 PUNTOS]. A la siguiente identidad se la conoce como la del calcetín de Navidad por la forma que, resaltados en el triángulo de Pascal, presentan los términos del sumatorio y la propia suma:

$$\sum_{i=r}^{n} \binom{i}{r} = \binom{n+1}{r+1} .$$



Dados dos enteros no negativos n y r tales que $n \geq r$, el algoritmo CALCETÍN devuelve el coeficiente binomial $C(n, r)$ en tiempo polinómico con respecto a n y r.

```
CALCETÍN(n, r):
    A[0..n – r, 0..r] ← ((0,…,0),…,(0,…,0))
    return CALCETÍN(A[0..n – r, 0..r], n, r)

CALCETÍN(A[0..ℓ, 0..m], n, r):
    if r = 0 or r = n
        return 1
    else
        if A[n – r, r] = 0
            for i ← r – 1 to n – 1
                A[n – r, r] ← A[n – r, r] + CALCETÍN(A[0..ℓ, 0..m], i, r – 1)
        return A[n – r, r]
```

a) [2 PUNTOS] Diseñad un algoritmo CALCETÍN iterativo cuyo coste temporal sea asintóticamente equivalente al de su homónimo recursivo.

b) [1 PUNTO] Determinad la cantidad de operaciones aritméticas que, en términos asintóticos, realiza el algoritmo CALCETÍN y calculad su coste temporal en los casos $r = 1$ y $r = \lfloor n/2 \rfloor$ (suponed que la complejidad de sumar dos enteros no negativos de d_1 y d_2 dígitos es $O(\max\{d_1, d_2\})$).

c) [1 PUNTO] Determinad la cantidad de elementos de $A[0..n - r, 0..r]$ que, en términos asintóticos, rellena el algoritmo CALCETÍN y calculad su coste espacial en los casos $r = 1$ y $r = \lfloor n/2 \rfloor$ (expresadlo en función del número de dígitos procesados).

Pista para b) y c): El n-ésimo coeficiente binomial de la forma $C(n, \lfloor n/2 \rfloor)$ tiene $O(n)$ dígitos.