Universidad de los Andes Colombia | Departamento de Ingeniería de Sistemas y Computación

Materia: Diseño y Análisis de Algoritmos
Código: 202021963
Nombre: Nicolás Rincón Sánchez

Corrección Examen Parcial #1

Consideraciones Generales

Para el análisis de complejidad, se utiliza la siguiente convención:

- Asignaciones (c_1)
- Comparaciones (c_2)
- Operaciones (c_3)

I. Punto de Programación Dinámica

Suponga que tiene un arreglo de número naturales S de tamaño $[0, n)$. Se le pide determinar si es posible partir el arreglo en dos subconjuntos de tal forma que la suma de los elementos que los componen sea la misma.

Entrada	Salida	Explicación
S=[1, 5, 11, 5]	True	[(1,5,5),(11)]
S=[1,5,3]	False	No hay manera de dividirlo en dos subconjuntos con igual suma
S=[1, 2, 3, 5]	False	No hay manera de dividirlo en dos subconjuntos con igual suma

Problema 1 [20 puntos]
Para el problema de programación dinámica diseñe una ecuación de recurrencia que lo resuelva. No olvide especificar claramente los casos base y explicar el caso recursivo.

Debido a que se debe dividir el arreglo en dos subarreglos con igual suma, esto implica que la suma de cada subarreglo será exactamente la mitad de la suma de todos los elementos del arreglo. De ahí que, como todos los elementos son naturales, entonces si la suma total es impar, automáticamente no se podrá dividir el arreglo.

Sea R igual a la suma de todo el arreglo. Luego, el problema recursivo es igual al del subconjunto suma, pero con un objetivo igual a $R/2$. Se recorre el arreglo desde el último elemento hasta el primero y se revisan dos casos:

- Si se incluye en el subarreglo el elemento i-ésimo (y se reduce la suma faltante)
- Si no se incluye en el subarreglo el elemento i-ésimo

Al hacer esto, estamos creando un subarreglo con suma $R/2$ y, automáticamente, los elementos no incluidos formarán otro subarreglo con suma $R/2$. Definimos la sustitución $B = R/2$, donde $R = (\sum i | 0 \leq i < n : S[i])$.

La ecuación recursiva con los casos base explícitos es la siguiente, tomando los índices del arreglo desde 1 hasta n:

$$subDoble(i,B,S) = \begin{cases} False & 2B \mod 2 \neq 0 \\ False & (B<0) \vee (i=0 \wedge B>0) \\ True & B=0 \\ subDoble(i-1,B,S) & (S[i]>B) \wedge (B>0) \wedge (i>0) \\ subDoble(i-1,B,S) \vee subDoble(i-1,B-S[i],S) & (S[i] \leq B) \wedge (B>0) \wedge (i>0) \end{cases}$$

Problema 2 [15 puntos]
Dibuje para un caso particular el grafo necesidades/tabla (paso de tabulación) donde se muestren las dependencias de los cálculos. **NOTA:** Este punto tiene sentido si y solo si la ecuación de recurrencia es correcta.

Supongamos el caso del primer ejemplo: $S = [1,5,11,5]$.

Calculamos la suma total como $R = 1+5+11+5 = 22$. De esta manera, $B = 22/2 = 11$ y evaluamos la función: $subDoble(4,11,S)$. La tabla construida es la siguiente:

i	B	11	10	9	8	7	6	5	4	3	2	1	0
4	5	True	True	False	False	False	True	True	False	False	False	True	True
3	11	True	False	False	False	False	True	True	False	False	False	True	True
2	5	False	False	False	False	False	True	True	False	False	False	True	True
1	1	False	False	False	False	False	False	False	False	False	False	True	True
0	N.A.	False	False	False	False	False	False	False	False	False	False	False	True

Esta tabla se construye partiendo de la última columna ($B = 0$) y la última fila ($i = 0$) inicializadas, debido a que se toman como casos base. Luego, se construye evaluando la disyunción entre la celda inmediatamente inferior y la celda ubicada a $S[i]$ columnas a la derecha y una fila inferior. Por facilidad, se muestran los valores de los $S[i]$ en la segunda columna. Adicionalmente, se considera que toda celda que caiga por fuera de la tabla mostrada, tendrá un valor False. Así, se llega a la conclusión de que sí es posible dividir el arreglo pues el valor en (4,11) es True.

Siguiendo los caminos posibles, donde un movimiento hacia abajo implica no agregar el elemento $S[i]$ y un movimiento diagonal implica agregarlo, se llega a que los dos caminos posibles para obtener una suma de 11 con el arreglo suministrado es [5,5,1] y [11], que coincide con lo presentado en el enunciado.

Problema 3 [15 puntos]
Desarrollar/implementar un algoritmo de programación dinámica (GCL) para resolver el problema. Indicar su complejidad temporal. **NOTA:** Este punto tiene sentido si y solo si la ecuación de recurrencia y la tabulación es correcta.

```
fun subDoble(S:Array[0,n) of nat) ret existe: bool
var i,j: int
var R: nat
var continuar: bool
var M: Array[0,n+1) x [0,R/2+1) of bool

i, R := 0;                                                    c1
do i<n -> R := R+S[i];                                        n(c2) + n(c1 + c3)
          i := i+1;                                           n(c1 + c3)
od

if R mod 2 != 0 ->  existe, continuar := false;               (c3 + c2) + 2c1
[] R mod 2 = 0 ->  existe, continuar := true;                 (c3 + c2) + 2c1
fi

if continuar = true ->                                        c2
      i := n;                                                 c1
      do i>=0 ->  M[i][0] := true;                            (n + 1)(c2 + c1)
                  i := i-1;                                   (n + 1)(c1 + c3)
      od
```

```
        j := 1;                                                                  c1
        do j<=R/2 ->  M[0][j] := false;                                          (R/2)(c2 + c1)
                      j := j+1;                                                  (R/2)(c1 + c3)
        od

        i := 1;                                                                  c1
        do i<=n ->                                                               n(c2)
             do j<=R/2 ->                                                        n · R/2(c2)
                  if j-S[i-1]<0 ->  M[i][j] := M[i-1][j];                        n · R/2 · (c3 + c2 + c1)
                  [] j-S[i-1]>=0 ->  M[i][j] := M[i-1][j] or M[i-1][j-S[i-1]];   n · R/2 · (2c3 + c2 + c1)
                  fi
                  j := j+1;                                                      R/2 · (c1 + c3)
             od
             i := i+1;                                                           n(c1 + c3)
        od
        existe := M[n][R/2];                                                     c1
 fi

 ret existe;
```

Al calcular la complejidad temporal exacta, se suman todas las complejidades de cada línea, en términos de n y de R:

$$T(n) = 7c_1 + 2c_2 + c_3 + n(3c_1 + 2c_2 + 3c_3) + (n+1)(2c_1 + c_2 + c_3) + \frac{R}{2}(2c_1 + c_2 + c_3) + n \cdot \frac{R}{2}(2c_1 + 2c_2 + 3c_3)$$

Por ende, al estimar la complejidad asintótica con la cota más ajustada será:

$$\mathcal{O}(n \cdot R/2)$$

II. Punto de Diseño de Algoritmos

Un arreglo de números naturales S de tamaño $[0, n)$ **se llama *k-único* si no contiene un par de elementos duplicados a una distancia de *k* posiciones uno del otro** $(k < n)$**, es decir, no existe un** i, j **tal que** $S[i] = S[j] \wedge |j - i| \leq k$**. Diseñe un programa que permita identificar si un arreglo es *k-único*.**

Entrada	Salida
k=3, S=[1, 2, 3, 4, 1, 2, 3, 4]	True
k=3, S=[1, 2, 3, 1, 4, 5]	False
k=3, S=[1, 2, 3, 4, 5]	True
k=3, S=[1, 2, 3, 4, 4]	False

Problema 1 [10 puntos]
Programa en GCL (no tiene que anotar el programa con la pre y post condición, pero si explicarlo).

Para resolver el problema se utilizó un algoritmo basado en dos ciclos. Se recorre cada posición del arreglo y en cada una de las posiciones se realiza un segundo recorrido. En este segundo recorrido, se atraviesan las k posiciones más próximas al elemento del primer ciclo. De esta manera, se revisa si existe algún duplicado del elemento dentro de las siguientes k posiciones. En tal caso, se interrumpe la búsqueda puesto que sólo es necesario un recorrido parcial hasta encontrar algún duplicado para saber que no es *k-único*. Así mismo, el segundo ciclo avanza k posiciones hasta alcanzar el último elemento del arreglo. Si se desborda el contador, se avanzan menos posiciones, de forma que independientemente del valor de k que ingrese por parámetro, el penúltimo elemento revisará sólo el último (su elemento inmediatamente siguiente) y así sucesivamente.

```
fun kUnico(S:Array[0,n) of nat, k:int) ret unico: bool
var i: int
var j: nat

i:= 0;                                                          c1
unico := true;                                                  c1
do i<n ->  j := 1;                                              (n+1)c2 + n·c1
           do j<k and i+j<n and unico = true ->                 n·k(3c2 + c3)
                     if S[i]=S[i+j] ->  unico := false;         n·k(c2 + c1)
                     fi
                     j := j+1;                                  n·k(c1 + c3)
           od
           i := i+1;                                            n(c1 + c3)
od
fi

ret unico;
```

Problema 2 [5 puntos]
Complejidad **exacta** y asintótica en tiempo para el peor caso. Se aconseja poner el costo computacional en cada línea de código para el análisis exacto.

Al calcular la complejidad temporal exacta, se suman todas las complejidades de cada línea, en términos de n y de k:

$$T(n) = 2c_1 + c_2 + n(2c_1 + c_2 + c_3) + n \cdot k(2c_1 + 4c_2 + 2c_3)$$

Por ende, al estimar la complejidad asintótica con la cota más ajustada será:

$$\mathcal{O}(n \cdot k)$$

III. Punto de Diseño de Algoritmos

Un problema común para los compiladores y editores de texto es determinar si los paréntesis (cuadrados/curvos/corchetes) en una cadena están equilibrados y correctamente anidados. Por ejemplo, la cadena "(((){[]}))({[]})()" contiene pares correctamente anidados, mientras que las cadenas ")([](" y "{()}" no. Proporcione un algoritmo que devuelva verdadero si una cadena contiene paréntesis (cuadrados/curvos/corchetes) correctamente anidados y equilibrados, y falso en caso contrario.

Problema 1 [10 puntos]
Programa en GCL (no tiene que anotar el programa con la pre y post condición, pero si explicarlo).

Para resolver el problema de verificar si los paréntesis están correctamente anidados y emparejados, se puede recurrir a una estructura de datos. En particular, es útil el uso de una pila (Stack). Esto debido a que es un TAD que tiene definidas dos operaciones simples cuya complejidad temporal es $\mathcal{O}(1)$. El algoritmo ingresa a la pila todos los paréntesis izquierdos "(","[","{" y cuando lee de la cadena un paréntesis derecho, compara si corresponde a la pareja del paréntesis ubicado en el tope de la pila. Si corresponden, entonces elimina el paréntesis en el tope (pues estarían correctamente emparejados). Si no corresponden, detiene la ejecución porque no estarían correctamente emparejados y retorna Falso. Inicialmente, verifica si la longitud de la cadena es impar. En tal caso, retorna Falso automáticamente porque es imposible que estén correctamente anidados. Lo mismo si al final la pila no está vacía.

```
fun parentesisAnidados(a:Array[0,n) of char) ret correcto: bool
var i: int
var c: char
var s: stack of char
var continuar: bool

i := 0;                                                                    c1
if n mod 2 != 0 ->  correcto, continuar := false;                          (c3 + c2) + 2c1
[] n mod 2 = 0 ->  correcto, continuar := true;                            (c3 + c2) + 2c1
fi

if continuar = true ->                                                     c2
    do i<n and correcto = true ->                                          (n + 1)c2
       if (a[i] = "(") or (a[i] = "[") or (a[i] = "{") ->  push(s, a[i]);  n · (3c2 + k_push)
       [] (a[i] = ")") or (a[i] = "]") or (a[i] = "}") ->  c := top(s);    n/2·(3c2+c1+k_top)

          if (c="(" and a[i]=")") or (c="[" and a[i]="]") or (c="{" and a[i]="}")
               pop(s);                                                     n/2 · (6c2 + k_pop)
          [] else ->  correcto := false;                                   6c2 + c1
          fi

       fi
       i := i+1;                                                           n(c1 + c3)
    od
    if s.size() != 0 ->  correcto := false;                                k_size + c2 + c1
    fi
fi

ret correcto;
```

*Nótese que la operación **top** se estudió en EDA y retorna el primer elemento de la pila sin eliminarlo. Es equivalente a buscar el primer índice de la lista con la que se implemente la pila, por lo que es de tiempo constante. Si la pila es vacía, retorna el carácter nulo. En todo caso, si no fuese permitido su uso, se podría recurrir a un pop, almacenarlo en una variable y luego hacerle push nuevamente para retornar la pila a su estado previo. Como las operaciones son de tiempo constante, sólo agregaría dos pasos en la ejecución, así que la complejidad asintótica no cambiaría de orden.*

Problema 2 [5 puntos]
Complejidad **exacta** y asintótica en tiempo para el peor caso. Se aconseja poner el costo computacional en cada línea de código para el análisis exacto.

En este caso, como se basa en condicionales, se toma la guarda del segundo `if` que tiene la mayor complejidad. Ocurre cuando la cadena tiene longitud par y todos sus elementos son paréntesis de apertura. El algoritmo le hace push a todos los elementos. En otro caso, la segunda guarda se ejecuta $\frac{n}{2}$ veces por mucho, cuando están correctamente emparejados. Lo mismo en su interior, la guarda que hace el pop, pues de lo contrario sólo basta con comparar una vez para ver que no corresponden las parejas y detener la ejecución. De esta manera, la complejidad exacta es:

$$T(n) = 4c_1 + 4c_2 + c_3 + n(4c_2 + k_{push} + c_1 + c_3) + k_{size}$$

Dado que la complejidad de las operaciones sobre la pila son constantes, se consideran como tal y se desprecian al evaluar la complejidad asintótica. Por ende, al estimar la complejidad asintótica con la cota más ajustada será:

$$\mathcal{O}(n)$$

IV. Punto de Ecuaciones Recursivas

Considere la siguiente función recursiva implementada en Python.

```python
def funRecursiva(n):
    """
    funRecursiva: Funcion recursiva de parcial 1
    n: nat
    """

    assert (isinstance(n,int)), "Solo se aceptan numeros enteros"
    assert (n>=0), "Solo se aceptan numeros enteros positivos"

    if n==0:
        return 0
    elif n==1:
        return 2
    elif n==2:
        return 68

    return 128*funRecursiva(n-3)-32*funRecursiva(n-2)-14*funRecursiva(n-1)
```

Problema 1 [20 puntos]
Calcule la complejidad en tiempo usando ecuaciones de recurrencia. Explique en detalle el procedimiento utilizado para que su punto sea válido.

Para resolver la ecuación de recurrencia, primero se expresa la ecuación con todas las evaluaciones recursivas en el lado izquierdo igualadas a una función o a una constante:

$$f(n) + 14f(n-1) + 32f(n-2) - 128f(n-3) = 0$$

Como es una ecuación homogénea, su solución general será igual a su solución homogénea. Obtenemos el polinomio característico y despejamos sus raíces con su respectiva multiplicidad:

$$\lambda^3 + 14\lambda^2 + 32\lambda - 128 = 0 = (\lambda - 2)(\lambda + 8)^2 \qquad \Longrightarrow \qquad \lambda = 2(mult1) \qquad \lambda = -8(mult2)$$

Debido a que las raíces y sus multiplicidades son las encontradas, la solución homogénea es:

$$f(n) = c_1 \cdot 2^n + c_2 \cdot (-8)^n + c_3 \cdot n \cdot (-8)^n$$

Reemplazamos las condiciones iniciales $f(0) = 0$, $f(1) = 2$ y $f(2) = 68$ para formar un sistema de ecuaciones y despejar las constantes expresadas en la forma de la solución general. Así obtenemos el siguiente sistema:

$$\begin{cases} c_1 + c_2 = 0 \\ 2c_1 - 8c_2 - 8c_3 = 2 \\ 4c_1 + 64c_2 + 128c_3 = 68 \end{cases}$$

La solución del sistema es $(1, -1, 1)$, por lo cual escribimos la solución de la recurrencia:

$$f(n) = 2^n - (-8)^n + n(-8)^n$$