Corrección del Examen 1: Diseño y Análisis de Algoritmos

Faiber Alonso Hernández Tavera
Código de estudiante: 202012367
Universidad de los Andes - Colombia
Departamento de Ingeniería de Sistemas y Computación
Semestre 2023-10

7 de abril de 2023

1. Punto 1

1.1. Enunciado

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.

Considere los siguientes casos ejemplo:

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

1.2. Definición de la ecuación de recurrencia

Para definir la ecuación de recurrencia general que puede solucionar el problema sin utilizar por ahora una estructura de datos y programación dinámica, se va a utilizar la función:

$$isPartitionPossible(i, remainingSum)$$

como una función que devuelve True si es posible dividir el subarreglo S[0:i] en dos subconjuntos con igual suma, y False en caso contrario.

Se utiliza la variable i, que corresponde al índice actual en el arreglo S en determinado tiempo y también se usa la variable $remainingSum$, la cual almacena el valor de la suma restante que se quiere alcanzar.

Aclaraciones

Al inicio, remainingSum es la mitad de la suma total de los elementos del arreglo S (es decir, $sum(S)/2$), ya que se están buscando dos subconjuntos cuya suma sea igual y divida la suma total de los elementos en dos partes iguales.

El valor inicial de i es n, donde n es el número de elementos en el arreglo S. Esto se debe a que comenzamos a considerar todos los elementos del arreglo y, en cada paso, decidimos si incluir o no el elemento en el subconjunto, que sería equivalente a restarlo de la variable $remainingSum$.

Por lo tanto, para iniciar el proceso, se va a llamar a la función de la siguiente manera:

$$isPartitionPossible(n, sum(S)//2)$$

donde $sum(S)$ es una función que puede ser definida por aparte y devuelve el valor de la suma de cada uno de los elementos que componen al arreglo S.

Nota: Se tiene en cuenta que si el arreglo es de tamaño 0 o tamaño 1, no es posible dividirlo en dos subarreglos y, por lo tanto, la respuesta será $False$

Construcción de la Ecuación de recurrencia

La ecuación de recurrencia se ve así:

1. **Caso base 1**: Si $remainingSum$ es igual a 0, entonces es posible dividir el subarreglo en dos subconjuntos con igual suma, así que $isPartitionPossible(i, remainingSum) = True$.
2. **Caso base 2**: Si i es igual a 0 y $remainingSum$ no es igual a 0, entonces no es posible dividir el subarreglo en dos subconjuntos con igual suma, así que $isPartitionPossible(i, remainingSum) = False$.
3. **Caso recursivo**: Si $S[i-1] > remainingSum$, entonces el elemento $S[i-1]$ no se puede agregar al subconjunto actual (pues desbordaría la suma que se lleva para un subarray), por lo que $isPartitionPossible(i, remainingSum) = isPartitionPossible(i-1, remainingSum)$. Es decir, se vuelve a llamar de forma recursiva a la función principal pero restando 1 al valor de i, es decir, moviendose sonbre el arreglo hacia la izquierda.
4. **Caso recursivo**: Si $S[i-1] \leq remainingSum$, entonces hay dos posibles opciones: agregar el elemento $S[i-1]$ al subconjunto actual o no agregarlo. Entonces, $isPartitionPossible(i, remainingSum) = isPartitionPossible(i-1, remainingSum)$ or $isPartitionPossible(i-1, remainingSum - S[i-1])$. Esta decisión se toma después de evaluar cada una y revisar si es $True$ alguna de ellas.

Estos casos pueden también verse como una función por partes:

$$isPartitionPossible(i, remainingSum) = \begin{cases} True, & \text{si } remainingSum = 0 \\ False, & \text{si } i = 0 \land remainingSum \neq 0 \\ isPartitionPossible(i-1, remainingSum), & \text{si } S[i-1] > remainingSum \\ isPartitionPossible(i-1, remainingSum) \lor & \\ isPartitionpossible(i-1, remainingSum - S[i-1]), & \text{si se da otro caso.} \end{cases} \tag{1}$$

Esta ecuación incluye los dos casos base identificados y dos casos recursivos, los cuales cuentan con condiciones definidas para ser utilizados dentro del algoritmo.

1.3. Estructura de estados / Paso de tabulación

Ahora, se va a dibujar la estructura de estados para un ejemplo específico. Esta estructura va a permitir esclarecer si es posible o no aplicar Programación Dinámica al problema para resolverlo sin ecuaciones de recurrencia. Este esclarecimiento se logra al confirmar que los resultados intermedios pueden depender de otros en la misma tabla para ser calculados.

En este caso, el enunciado afirma que es posible aplicar Programación Dinámica al problema, por lo tanto, es seguro que esta posibilidad es cierta en este caso.

i\remSum	0	1	2	3	4	5	6	7	8	9	10	11
0	T	F	F	F	F	F	F	F	F	F	F	F
1	T	T	F	F	F	T	F	F	F	F	F	F
2	T	T	F	F	F	T	F	F	F	F	F	T
3	T	T	F	F	F	T	F	F	F	F	F	T
4	T	T	F	F	F	T	T	F	F	F	F	T

Cuadro 1: Matriz de estados para $S : [1, 5, 11, 5]$

Se va a tomar uno de los ejemplos dados en el enunciado, específicamente el que define el arreglo S como: $S : [1, 5, 11, 5]$

Para este caso, una matriz corresponde a la estructura de datos más pertinente para almacenar los estados intermedios de ejecución del algoritmo. El Cuadro 1 permite visualizar esa matriz de estados contruída.

Explicación de la matriz de estados

En la matriz de estados:

1. los índices de las filas corresponden a la variable i, que corresponde a la posición que se está verificando del arreglo. Estos índices van desde 0 hasta el valor de n donde n es el tamaño del arreglo S calculado por una función $size = S.size()$, pues se guarda el índice 0 para el caso base.

2. los índices de las columnas corresponden a la variable $remainingSum$, la cual guarda el valor que hace falta de la suma que se quiere alcanzar con los elementos del arreglo. Las columnas van desde 0 hasta el valor sumaTotal(S)/2, pues se guarda el índice 0 para el caso base.

Nota: sumaTotal(n) se refiere a una función que recibe un arreglo de números n y devuelve la suma de todos sus elementos.

Los elementos de la matriz corresponden a los valores booleanos:

1. T: True

2. F: False

Estos valores booleanos permiten almacenar resultados de ejecuciones intermedias del algoritmo, para que después, cuando se quiera buscar si un arreglo más grande o con elementos distintos puede partirse en subarreglos con igual suma de sus elementos, se puedan utilizar esos resultados y dar una respuesta más rapida a partir de revisar posiciones de la matriz y no hacer llamados recursivos.

Estos valores booleanos con los que se llena matriz, se determinan a partir de la ecuación de recurrencia definida.

¿Cómo se puede determinar el resultado del ejemplo con esta matriz de estados?

Para determinar saber si este arreglo de ejemplo ($S : [1, 5, 11, 5]$) o cualquier otro se pueden dividir en 2 subarreglos cuyos elementos de la misma suma, se puede acudir a la matriz de estados. Esta matriz se puede seguir llenando a partir de más llamados a la ecuación recursiva para más casos donde la variable i y la variable $remainingSum$ son de mayor valor a las del ejemplo.

En el caso de: $S : [1, 5, 11, 5]$

1. La longitud del arreglo: $S.size()$ es igual a 4 (en esta solución, se tiene en cuenta el índice 0 para el tamaño), por tanto, la variable i comienza con el valor 4.

2. La suma total de los elementos del arreglo S es igual a 22, por tanto, la suma faltante ($remainingSum$) se inicializa con el valor $22/2 = 11$

Teniendo estos datos, se revisa la matriz de estados y específicamente su elemento que tiene la fila de índice ($i = 4$) y la columna de índice $remainingSum = 11$

El valor en la posición [4][11] de la matriz es: $True$ y, por tanto, esta es la respuesta para el caso evaluado.

Aplicación en un algoritmo de Programación Dinámica

Esta matriz de estados es de completa utilidad para ahora implementar un algoritmo de programación dinámica que permita utilizar esa estructura de datos con resultados de ejecución intermedios y no tener que depender de recursión para hallar respuestas a este problema.

Se sabe que se puede aplicar programación dinámica porque este paso de Memoización permite ver que, a partir de resultados intermedios (posiciones de la matriz), se pueden determinar otros resultados para valores de i y $remainingSum$ más grandes. Cuando se observa la ecuación recursiva, se puede notar que en los pasos recursivos, se realiza un llamado a la función pero con un valor distinto de i y/o un valor distinto de la variable $remainingSum$.

Esto mismo podría hacerse pero con esta matriz, donde en lugar de volver a llamarse la función, se revise la posición de la matriz con esos valores de i y $remainingSum$ cambiados. Esto permite dar respuestas a casos donde el arreglo tiene más posiciones o distinta suma de sus elementos (problemas más grandes).

A continuación, se va a presentar una implementación definida en GCL para un algoritmo que pueda solucionar este problema utilizando programación dinámica.

1.4. Especificación en GCL y Determinación de la Complejidad

La siguiente es una implementación posible de un algoritmo según la ecuación de recurrencia y matriz de estados definidas que permite solucionar este problema:

```
1
2  fun is_partition_possible(S: array[0:N) of int)
3  ret resultado: bool
4
5  var i, j, target_sum: int;
6  var dp: array[0:N+1) of array[0:SumaTotal/2 + 1) of bool
7
8  N := S.size() "Equivalente a len(S)"
9
10 if N <= 1 →    "Verificacion de que el size del arreglo es mayor a 1"
11     resultado := false
12     ret resultado
13
14 [] N > 1 → total_sum := sum(S)
15
16 fi
17
18 if total_sum mod 2 != 0 →
19     resultado := false     "Suma impar: no es posible hacer particion de"
20                            "igual suma"
21     ret resultado
22 fi
23
```

```
target_sum := total_sum // 2

"(* Inicializar matriz dp *) con valores false"
i := 0
do i <= N →
    j := 0
    do j <= SumaTotal/2 →
        dp[i][j] := false
        j := j + 1
    od;
    i := i + 1
od

"Llenar matriz con (* Casos base *)"

do 0 <= i < N → "(suma restante = 0)"
dp[i][0] := true
od

do 1 <= j < (target_sum+1) →
dp[0][j] := false  "(i = 0, remainingSum != 0)"
od

"Llenar el resto de la matriz 'dp' a partir de resultados"
"intermedios ya calculados y almacenados en la misma"

i := 1
do i < (N+1) →
    remaining_sum := target_sum
    do remaining_sum < (target_sum+1) →
        if S[i - 1] > remaining_sum →
            dp[i][remaining_sum] := dp[i - 1][remaining_sum]

        [] S[i - 1] <= remaining_sum →
            dp[i][remaining_sum] := dp[i - 1][remaining_sum]
            ∨ dp[i - 1][remaining_sum - S[i - 1]]
        fi
    od
od

resultado := dp[N][target_sum]

ret resultado
```

Complejidad del algoritmo construido

La complejidad temporal del algoritmo *is_partition_possible*, implementado con programación dinámica, es $O(n * SumaTotal/2)$, donde n es la cantidad de elementos en el conjunto y SumaTotal es la suma de todos los elementos del arreglo S.

La razón detrás de este valor de complejidad consiste en que el algoritmo se basa principalmente en llenar una tabla (matriz) de tamaño (n x ($SumaTotal/2$)); las demás acciones consisten en asignaciones, comparaciones y operaciones básicas, las cuales tienen complejidad constante y no son realmente influyentes. Llenar esta matriz de estados, que se denota en la implementación como *dp* es la acción que más influye en este algoritmo y, por tanto, su complejidad corresponde a una asíntota superior de la función complejidad exacta del algoritmo. Entonces, la complejidad temporal del algoritmo es proporcional al producto de n y ($SumaTotal/2$):

Complejidad temporal asintótica: $\boldsymbol{O(n * SumaTotal/2)}$

2. Punto 2

2.1. Enunciado

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 $S[i] = S[j]$ y $|j - i| \leq k$. Diseñe un programa que permita identificar si un arreglo es k-único.

Estos son 4 ejemplos de Entradas y salidas esperadas del algoritmo:

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

A continuación, se presenta un algoritmo especificado en GCL (Guarded Command Language) que resuelve el problema planteado:

2.2. Algoritmo especificado en GCL para resolver el problema

```
fun es_k_unico (k: int, S: array [0:N) of int)
ret resultado: bool

var i, j: int
resultado, i := true, 0

do i < N-1 →
    j := i + 1
    do j < N ∧ j <= i + k →
        if S[i] = S[j] →
            resultado := false
        fi;
        j := j + 1
    od;
    i := i + 1
od

ret resultado
```

Precondición: $\{Q : 0 \leq k < N\}$
Postcondición: $\{R : \text{resultado} = \neg(\exists i, j : 0 \leq i < N \wedge 0 \leq j < N \wedge |j - i| \leq k : S[i] = S[j])\}$

2.3. Descripción de la pre y post condición del algoritmo

La Precondición(Q), corresponde a $\{Q : 0 \leq k < N\}$ y consiste en lo siguiente:

- El valor de 'k' debe ser un número entero no negativo y menor que 'N'.
- 'N' es el tamaño del arreglo 'S'.
- Esto garantiza que 'k' sea una distancia válida dentro del rango del arreglo para comparar los elementos y verificar si el arreglo es k-único.

Por otra parte, la postcondición identificada es:

$$\{R : \text{resultado} = \neg(\exists i, j : 0 \leq i < N \wedge 0 \leq j < N \wedge |j - i| \leq k : S[i] = S[j])\}$$

y consiste en lo siguiente:

La postcondición establece que el valor de 'resultado'(un valor booleano) será verdadero (True) si NO existe un par de índices 'i' y 'j' en el rango de 0 a N-1 (inclusive) que cumplan las siguientes condiciones:

1. La distancia entre los índices 'i' y 'j' es menor o igual a 'k', es decir, $|j - i| \leq k$.
2. Los elementos en las posiciones 'i' y 'j' del arreglo 'S' son iguales, es decir, $S[i] = S[j]$.

Si existe al menos un par de índices 'i' y 'j' que cumplen ambas condiciones, entonces el valor de 'resultado' será falso (False), lo que indica que el arreglo no es k-único.

Para ser breve, la postcondición asegura que el valor de 'resultado' refleje correctamente si el arreglo es k-único o no, de acuerdo con la definición del problema y las restricciones dadas.

2.4. Análisis de complejidad

A continuación se presentan los análisis realizados al algoritmo construido en términos de complejidad temporal.

2.4.1. Complejidad exacta

Primero, se va a tomar en consideración de nuevo la especificación del algoritmo en GCL pero esta vez con los elementos del programa enumerados para calcular la complejida exacta:

```
1  fun es_k_unico (k: int, S: array [0:N) of int)
2  ret resultado: bool
3
4  var i, j: int
5  resultado, i := true, 0                    (1)
6
7  do i < N-1 →                               (2)
8      j := i + 1                             (3)
9      do j < N ∧ j <= i + k →                (4)
10         if S[i] = S[j] →                   (5)
11             resultado := false             (6)
12         fi;
13         j := j + 1                         (7)
14     od;
15     i := i + 1                             (8)
16 od
17
18 ret resultado
```

Ahora, teniendo en cuenta las siguientes convenciones del modelo RAM, se va a determinar una ecuación exacta de complejidad del algoritmo:

Valor constante de la complejidad en tiempo que implica:

1. Una asignación: $C1$
2. Una comparación: $C2$
3. Operaciones matemáticas básicas: $C3$

Análisis de cada acción enumerada: Se va a analizar cada linea donde haya una acción por separado, es decir, el costo individual de cada acción dentro del programa.

1. La acción (1) implica dos asignaciones: $2C1$.
2. La acción (2) implica una comparación y repetición do-od que se repiten N veces: $C2N$

3. La acción (3) implica una asignación y una operación básica: $N(C1 + C3)$

4. La acción (4) es la condición del bucle interno, y este bucle interno se ejecuta hasta que j sea igual a N o j sea mayor que i + k. En promedio, el bucle se ejecutará aproximadamente k veces, ya que j puede estar a una distancia máxima k de i (debido a la condición $j \leq i + k$). Además hay dos comparaciones y una operación báscia: $N * k * (2C2 + C3)$

5. La acción (5) es una comparación: $N * k * C2$

6. La acción (6) es una asignación: $N * k * C1$ (en el peor caso)

7. La acción (7) es una asignación y operación básica: $N * k * (C1 + C3)$

8. La acción (8) es una asignación y operación básica: $N * (C1 + C3)$

Ahora, se realiza una suma de esas complejidades individuales de cada acción:

$$T(N) = 2C1+C2*N+N(C1+C3)+N*k*(2C2+C3)+N*k*C2+N*k*C1+N*k*(C1+C3)+N*(C1+C3)$$

y al simplificarse:

$$\mathbf{T(N) = 4C1 * N + (C1 + C2 + C3)N + (2C2 + 3C1 + 4C3)N * k}$$

2.4.2. Complejidad asintótica

La complejidad asintótica en tiempo para el peor caso se puede obtener apartir del analisis del término de mayor crecimiento en la expresión de la complejidad exacta. En este caso, el término de mayor crecimiento es N * k, lo que nos lleva a la siguiente complejidad asintótica en tiempo:

$$\mathbf{O(N * k)}$$

Por lo tanto, la complejidad asintótica en tiempo para el peor caso del algoritmo es O(N * k).

3. Punto 3

3.1. Enunciado

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.

Nota: Entra una cadena de caracteres compuesta únicamente por paréntesis (cuadrados/curvos/corchetes) ()[].

3.2. Algoritmo especificado en GCL para resolver el problema

```
1
2  fun parentesis_balanceados (S: array [0:N) of char)
3  ret resultado: bool
4  var i: int
5  var pila: stack of char
6  {Q: S contiene solo caracteres '(', ')', '[', ']', '{', '}'}
7  resultado, i := true, 0
8
9  do i < N →
10     if S[i] = '(' ∨ S[i] = '[' ∨ S[i] = '{' →
```

```
        push(S[i], pila)
    [] S[i] = ')' ∧ ¬empty(pila) ∧ top(pila) = '(' →
        pop(pila)
    [] S[i] = ']' ∧ ¬empty(pila) ∧ top(pila) = '[' →
        pop(pila)
    [] S[i] = '}' ∧ ¬empty(pila) ∧ top(pila) = '{' →
        pop(pila)
    [] true →
        resultado := false
    fi;
    i := i + 1
od

resultado := resultado ∧ empty(pila)

ret resultado
{R:resultado = la cadena S tiene parentesis correctamente anidados
y equilibrados}
```

3.3. Descripción de la pre y post condición del algoritmo

Precondición

La precondición del algoritmo consiste en que la cadena de entrada S contiene únicamente caracteres de paréntesis (cuadrados/curvos/corchetes), es decir, '(', ')', '[', ']', '', ''. La longitud de la cadena es N ($0 \leq N$).

Entonces, el algoritmo espera una cadena de entrada compuesta exclusivamente por estos caracteres y no considerará otros caracteres en su análisis.

Postcondición

La postcondición del algoritmo consiste en que el resultado (que es un valor booleano: *false o true*) indica si la cadena de entrada S tiene paréntesis correctamente anidados y equilibrados. Si resultado es verdadero (*true*), entonces la cadena S tiene paréntesis correctamente anidados y equilibrados (Porque todos fueron contados y revisados por el algoritmo). Por el contrario, si resultado es falso ($false$), entonces es porque no se cumple con paréntesis anidados y equilibrados en la entrada.

En otras palabras, la postcondición garantiza que el algoritmo determinará correctamente si los paréntesis en la cadena de entrada están anidados y equilibrados de acuerdo con las reglas mencionadas en el enunciado del problema.

3.4. Análisis de complejidad exacta y asintótica

3.4.1. Complejidad exacta

Para la complejidad exacta, se va a analizar cada línea del programa que corresponda a una acción revelante para el orden de crecimiento temporal que podría tener el algoritmo.

```

fun parentesis_balanceados (S: array [0:N) of char)
ret resultado: bool
var i: int
var pila: stack of char
{Q: S contiene solo caracteres '(', ')', '[', ']', '{', '}'}
resultado, i := true, 0                                          (1)

```

```
do i < N →                                                              (2)
    if S[i] = '(' ∨ S[i] = '[' ∨ S[i] = '{' →                          (3)
        push(S[i], pila)
    [] S[i] = ')' ∧ ¬empty(pila) ∧ top(pila) = '(' →                    (4)
        pop(pila)
    [] S[i] = ']' ∧ ¬empty(pila) ∧ top(pila) = '[' →                    (5)
        pop(pila)
    [] S[i] = '}' ∧ ¬empty(pila) ∧ top(pila) = '{' →                    (6)
        pop(pila)
    [] true →                                                           (7)
        resultado := false                                              (8)
    fi;
    i := i + 1                                                          (9)
od

resultado := resultado ∧ empty(pila)                                    (10)

ret resultado
{R:resultado = la cadena S tiene parentesis correctamente anidados
y equilibrados}
```

Se va a analizar cada linea donde haya una acción por separado, es decir, el costo individual de cada acción dentro del programa.

1. La acción (1) implica dos asignaciones: $2C1$.
2. La acción (2) implica una comparación y repetición do-od que se repiten N veces: $C2 * N$
3. Las acciones (3), (4), (5) y (6) realizan 3 comparaciones cada una (si se cuenta a $\neg(empty())$ como otra comparación) y corresponden a guardas cuyas condiciones se evalúan y cuyos cuerpos se ejecutan un número N de veces, teniendo en cuenta que por cada repetición del 'do' solo se ejecuta el cuerpo de una guarda. De acuerdo con esto ,es posible considerar que el costo mayor que tendría la evaluación de cualquier guarda en cada repetición es de: $N * 3C2$
4. La acción (7) es la última guarda que se evalúa como un 'if', corresponde a un true y se utiliza como una condición de "fallback.º de reserva. Cuando se evalúan las condiciones en una estructura if, si ninguna de las otras condiciones se cumple (es decir, el caracter actual no es un paréntesis de apertura, ni coincide con un paréntesis de cierre correspondiente), entonces la guarda [] true se ejecutará.Esta guarda simplemente establece resultado (false), lo que indica que la cadena no está correctamente balanceada y anidada. $N * (C2 + C1)$
5. La acción (7) hace parte de la ejecución de la guarda []true y solo realiza una asignación.
6. La acción (9) es una asignación y una operación básica: $N(C1 + C3)$
7. La acción (10), finalmente, corresponde a una asignación: $C1$

Además, es importante tener en cuenta las complejidades de las operaciones básicas de la pila, las cuales fueron utilizadas teniendo en cuenta su **definición abstracta como estructura de datos**:

Las operaciones de push, pop y empty en una pila tienen generalmente las siguientes complejidades de tiempo en el peor caso:

push: O(1) - La operación de agregar un elemento a la pila es una operación de tiempo constante, ya que solo implica agregar un elemento en la parte superior de la pila.
pop: O(1) - La operación de eliminar un elemento de la pila también es una operación de tiempo constante, ya que solo implica quitar el elemento de la parte superior de la pila.
top O(1) - retornar el elemento que se encuentra en el tope de la pila requiere un solo paso y lo hace en tiempo constante.

empty: O(1) - La operación de verificar si una pila está vacía es una operación de tiempo constante, ya que solo requiere verificar si el tamaño de la pila es cero o no.

Es relevante aclarar que estas complejidades son válidas para pilas implementadas mediante estructuras de datos como arreglos dinámicos o listas enlazadas. Dado que las operaciones top, push, pop y empty tienen complejidades de tiempo constantes, no afectan la complejidad asintótica general del algoritmo de paréntesis balanceados y si se fuesen a añadir en la ecuación de complejidad exacta, lo harían como números constantes, cuya influencia en la complejidad es practicamente nula.

Ahora, se realiza una suma de esas complejidades individuales de cada acción:

$$T(N) = 2C1 + N * C2 + N * 3C2 + N(C2 + C1) + N(C1 + C3) + C1$$

y al simplificarse:

$$= 3C1 + N * (C2 + 3C2 + C2 + C1 + C1 + C3)$$
$$= \mathbf{3C1 + 5NC2 + 2NC1 + NC3}$$

3.4.2. Complejidad asintótica

La complejidad asintótica en tiempo para el peor caso sería dominada por los términos de mayor orden, que en este caso son los términos lineales cuya variable es N:

$$5NC2 + 2NC1 + NC3$$

Dado que estamos interesados en la complejidad en tiempo con respecto a N, y no en los costos específicos constantes C1, C2 y C3, es posible decir que la complejidad asintótica en tiempo para el peor caso es:

$$\mathbf{O(N)}$$

Porque los 3 términos más relevantes que contienen a N están sumándose, no multiplicándose.

Por lo tanto, el algoritmo de paréntesis balanceados tiene una complejidad de tiempo lineal en el peor caso, lo que significa que su tiempo de ejecución aumenta linealmente con el tamaño de la entrada N.

4. Punto 4

4.1. Enunciado

Considere la siguiente función recursiva implementada en Python.

```
1
2  def funRecursiva(n):
3
4  """
5  n: nat
6  """
7  assert (isInstance(n, int)), "Solo se aceptan numeros enteros"
8  assert (n>=0), "Solo se aceptan numeros enteros positivos "
9
10 if n==0:
11         return 0
12
13 elif n==1:
14         return 2
15
16 elif n==2
17         return 68
18
19 return 128*funRecursiva(n-3)-32*funRecursiva(n-2)-14*funRecursiva(n-1)
```

Calcule la complejidad en tiempo usando ecuaciones de recurrencia. Explique en detalle el procedimiento utilizado para que su punto sea válido.

4.2. Calculo de la complejidad temporal

Para estimar la complejidad temporal de este algoritmo, se va a utilizar un método matemático que consiste en determinar las raíces que son soluciones a un polinomio característico obtenido a partir de la ecuación de recurrencia del algoritmo.

Primero, es importante tener en cuenta los casos base de la función recursiva dada:

1. Primer caso base: f(0) = 0
2. Segundo caso base: f(1) = 2
3. Tercer caso base: f(2) = 68

Ahora, la ecuación de recurrencia que aplica para todos los n mayores a 2 ($n > 2$), se puede obtener a partir de la expresión dada en el último retorno de la función de python:

Nota: De ahora en adelante, 'funRecursiva(n)' se va a denotar como f(n).

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

Y si todos los términos que contienen f(n) se agrupan en un solo lado de la igualdad:

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

Se termina por obtener una ecuación homogénea de grado 3.

Ahora, se procede a hallar el polinomio característico de esta ecuación homogénea para posteriormente determinar sus raíces.

El polinomio característico relacionado es:

$$\lambda^3 + 14\lambda^2 + 32\lambda - 128 = 0$$

Una vez se cuenta con el polinomio característico de la ecuación de recurrencia, se procede a solucionarlo, es decir, obtener las raíces solución.

Para este caso, se utilizó el software matemático Symbolab: `https://www.symbolab.com/`

La factorización de este polinomio característico dio como resultado la expresión:

$$(\lambda - 2)(\lambda + 8)^2 = 0$$

Y las raíces solución obtenidas fueron:

1. $\lambda_1 = 2$
2. $\lambda_2 = -8$

Debido a que la multiplicidad de la raíz $\lambda_2 = -8$ es 2, como se puede apreciar en el polinomio característico simplificado, la tercera raíz es:

$$\lambda_3 = -8$$

Ahora, para determinar una expresión que corresponda a todas las soluciones posibles de la ecuación de recurrencia, se va a utilizar el siguiente teorema:

Teorema 2: *Si r es una raíz con multiplicidad m del polinomio característico, entonces para $0 \leq j < m$, $f(n) = n^j * r^n$ son soluciones de la ecuación de recurrencia correspondiente.*

Este teorema hace parte de un conjunto de tres teoremas existentes para resolver Relaciones Lineales de Recurrencia con Coeficientes Constantes. Para la solución de este punto, se ha tomado

como base la teoría correspondiente a la solución de estas ecuaciones que se encuentra en una publicación de la Universidad de Victoria: `https://www.uvic.ca/`, específicamente en el enlace: `https://www.math.uvic.ca/faculty/gmacgill/guide/recurrences.pdf`

La expresión que muestra la estructura de todas las soluciones posibles de la ecuación de recurrencia es la siguiente:

$$\boldsymbol{f(n) = C_1 * 2^n + C_2 * (-8)^n + C_3 * n * (-8)^n}$$

Y finalmente se pueden calcular las constantes C_1, C_2 y C_3 a partir de los casos base conocidos. Para esto es necesario resolver un sistema de ecuaciones de 3 ecuaciones y 3 incógnitas:

Nota: Se considera que C_n es equivalente a Cn

$$f(0) = C1 * 2^0 + C2 * (-8)^0 + C3 * 0 * (-8)^0 = C1 + C2 = 0 \tag{2}$$

$$f(1) = C1 * 2^1 + C2 * (-8)^1 + C3 * 1 * (-8)^1 = 2 \tag{3}$$

$$f(2) = C1 * 2^2 + C2 * (-8)^2 + C3 * 2 * (-8)^2 = 68 \tag{4}$$

Al utilizar un software matemático para resolver este sistema de ecuaciones, los resultados correspondientes a los valores de las constantes C_1, C_2 y C_3 son:

$$C1 = 1 \tag{1}$$

$$C2 = -1 \tag{2}$$

$$C3 = 1 \tag{3}$$

Ahora, con esta información, se tiene una de complejidad temporal para la función: *funRecursiva(n)*, la cual permite tener una idea de la tendencia de su crecimiento en función del tiempo:

$$\boldsymbol{f(n) = 2^n - (-8)^n + n((-8)^n)}$$

En términos generales, la complejidad temporal de una función está dominada por el término de mayor crecimiento que hace parte de ella.

En este caso, el término $n((-8)^n$ crece más rápido que los otros términos en la ecuación, especialmente para valores grandes de n. Por lo tanto, la complejidad temporal de la función funRecursiva(n) estará dominada por el término $n((-8)^n$:

$$\boldsymbol{f(n) \approx n((-8)^n}$$

La complejidad temporal asintótica de la función funRecursiva(n) es, por lo tanto,

$$\boldsymbol{O(n((-8)^n)}$$

Esto significa que el tiempo de ejecución de la función crece exponencialmente con respecto a n, lo que indica que este algoritmo no es eficiente cuando n toma valores muy grandes.

5. Referencia consultada

1. University of Victoria (Ed.). (s.f.). Recurrence Relations,
   Tomado el día 23 de Marzo de 2023 de la página de la Universidad: https://www.uvic.ca/.
   En el recurso: https://www.math.uvic.ca/faculty/gmacgill/guide/recurrences.pdf