Análisis y Diseño de Algoritmos

Tarea 3

Daniel Rojo Mata
danielrojomata@gmail.com

Fecha de Entrega: 8 de septiembre de 2024

Problema 1

Da una variante del algoritmo recursivo de búsqueda binaria que devuelva el índice más pequeño i tal que $A[i]$ es igual al elemento buscado x, o $false$ si x no aparece en A. Tu algoritmo debe tener complejidad de tiempo $O(logn)$ y de espacio $O(1)$.

Solución

BusquedaBinariaModificada

- `Entrada:` Un arreglo A de longitud n y un valor x. El valor x es un carácter que se busca en A.
- `Salida:` El índice más pequeño i tal que $A[i]$ es igual a x, o `false` si x no aparece en A.

```
1 function BusquedaBinariaModificada(A, x):
2     return BuscaIndiceMenor(A, x, 0, longitud(A) - 1, false)
3 end function
```

Algoritmo Anagrama

- `Entrada:` Un arreglo A de longitud n, un valor x, y los índices $inicio$ y fin que definen el rango de búsqueda, así como un valor de entrada 'resultado' inicial.
- `Salida:` El índice más pequeño en A donde se encuentra el valor x, o el valor de 'resultado' si x no está presente en A.

```
1  function BuscaIndiceMenor(A, x, inicio, fin, resultado):
2      if inicio > fin then
3          return resultado
4      end if
5
6      medio = (inicio + fin) // 2
7
8      if A[medio] == x then
9          resultado = medio
10         return BuscaIndiceMenor(A,
11                                 x,
12                                 inicio,
13                                 medio - 1,
14                                 resultado)
15
16     else if A[medio] < x entonces:
17         return BuscaIndiceMenor(A,
18                                 x,
19                                 medio + 1,
20                                 fin,
21                                 resultado)
22
23     else
24         retornar BuscaIndiceMenor(A,
25                                   x,
26                                   inicio,
27                                   medio - 1,
28                                   resultado)
29     end else
30
31 end function
```

El objetivo del algoritmo es encontrar el índice más pequeño i tal que $A[i] = x$. Para lograr esto, el algoritmo realiza una búsqueda binaria modificada de la siguiente manera:

- **División y comparación:** Como en la búsqueda binaria estándar, el algoritmo divide el arreglo en mitades utilizando un índice medio. Si el valor en $A[\text{medio}]$ es igual a x, el algoritmo actualiza el resultado con el valor de medio y continúa buscando en la mitad izquierda del arreglo, ya que es posible que haya una instancia de x en un índice más pequeño.
- **Mitad derecha:** Si $A[\text{medio}] < x$, el algoritmo se enfoca en la mitad derecha del arreglo, ya que el valor x debe estar en esa parte (si es que está en el arreglo).
- **Mitad izquierda:** Si $A[\text{medio}] > x$, el algoritmo se enfoca en la mitad izquierda del arreglo.
- **Condición de terminación:** La búsqueda se detiene cuando el índice de inicio es mayor que el índice de fin. En ese momento, el algoritmo devuelve el valor de `resultado`. Si el valor nunca se encuentra, `resultado` permanece siendo `false`.

Complejidad Temporal:
La complejidad temporal de este algoritmo sigue siendo $O(\log n)$:

- El arreglo se divide a la mitad en cada llamada recursiva, lo que significa que el número total de llamadas es $\log_2(n)$.
- Aunque el algoritmo realiza comparaciones adicionales para encontrar el índice más pequeño, la cantidad de trabajo adicional en cada nivel de recursión es constante, es decir, $O(1)$.

Por lo tanto, el tiempo de ejecución total sigue siendo $O(\log n)$.

Problema 2

Diseña un algoritmo de tiempo $O(n)$ y espacio $O(1)$ que reciba dos cadenas de longitud n (que básicamente son arreglos con caracteres en sus entradas), cada una con una palabra en español, y devuelva `true` si y solo si una es anagrama de la otra.

Solución

Algoritmo Anagrama

- `Entrada:` Dos arreglos A y B, ambos de longitud n, cuyas entradas son caracteres escritos en minúsculas incluida la letra 'ñ'.
- `Salida:` `True` si las palabras formadas en A y B son anagramas.

```
1  conteo_A = {'a': 0, 'b': 0, 'c':0 , ..., 'z':0}
2  conteo_B = {'a': 0, 'b': 0, 'c':0 , ..., 'z':0}
3  abecedario = {1: 'a', 2: 'b', 3: 'c', ..., 27: 'z' }
4
5  for i=1 up to n do
6      conteo_A[A[i]] = conteo_A[A[i]] + 1
7      conteo_B[B[i]] = conteo_B[B[i]] + 1
8  end for
9
10 for i=1 up to 27 do
11     if conteo_A[abecedario[i]] != conteo_B[abecedario[i]]
12         return False
13     end if
14 end for
15
16 return True
```

Corrección del Algoritmo

Para la creación del algoritmo `Anagrama`, se utilizan tres diccionarios. El diccionario `conteo_A` almacena la cantidad de veces que aparece cada letra en el arreglo `A`, siendo la *key* la letra y el *value* la cantidad de apariciones. De manera análoga, el diccionario `conteo_B` cuenta las apariciones de las letras en el arreglo `B`.
A modo de ejemplo, si se tiene el par $(a, 2)$ es porque en la palabra la letra a ha aparecido 2 veces.
El tercer diccionario, llamado `abecedario`, almacena las letras del alfabeto. Las *keys* de este diccionario son valores enteros entre 1 y 27, cubriendo todas las letras del alfabeto, incluida la letra $\tilde{n}$. La asignación se realiza de la siguiente manera: $(1, a), (2, b), (3, c), \ldots, (27, z)$.

Ahora, se procede a argumentar la correctitud del algoritmo.
El algoritmo `Anagrama` verifica si dos arreglos A y B son anagramas si al contar las apariciones de

cada letra en ambos arreglos y luego compararlas estas frecuencias son las mismas.

El primer bucle, `5-8`, recorre los arreglos A y B simultáneamente. Para cada posición i, el algoritmo incrementa el conteo correspondiente en los diccionarios `conteo_A` y `conteo_B`.

Si se pensara en un invariante, éste sería de la forma:

LEMA 1: Después de la iteración `i` en el bucle `5-8`, los diccionarios `conteo_A` y `conteo_B` reflejan el número de apariciones de cada letra en las subcadenas $A[0:i]$ y $B[0:i]$, respectivamente.

- **Inicialización:**

  Antes de comenzar el ciclo, los diccionarios `conteo_A` y `conteo_B` están inicializados con valores de 0 para cada letra del alfabeto. Esto establece el estado inicial correcto para el conteo.

- **Mantenimiento:**

  Durante cada iteración i, el algoritmo incrementa el conteo de la letra en la posición i de los arreglos A y B en ambos diccionarios. Esto asegura que, al finalizar la iteración, los conteos de las letras en `conteo_A` y `conteo_B` reflejan correctamente las letras que han sido procesadas hasta esa iteración.

- **Terminación:**

  Una vez completadas todas las iteraciones, los diccionarios `conteo_A` y `conteo_B` contendrán el conteo de todas las letras en los arreglos A y B. Por lo tanto, el invariante garantiza que al final del ciclo, ambos diccionarios tienen el conteo correcto de cada letra en los arreglos completos.

De esta manera, cada letra en ambos arreglos es contada adecuadamente. Así, el invariante se cumple.

El segundo bucle recorre el diccionario `abecedario`, que contiene todas las letras del alfabeto, y compara el conteo de cada letra en `conteo_A` y `conteo_B`. Si en algún momento los conteos de una letra no coinciden entre ambos diccionarios, el algoritmo retorna `False`, indicando que los arreglos no son anagramas.

Finalmente, si todos los conteos coinciden, el algoritmo retorna `True`, indicando que los arreglos A y B son anagramas. Esto es porque, para que dos arreglos sean anagramas, deben contener las mismas letras con las mismas frecuencias.

Para esta situación se tendría:

LEMA 2: Al finalizar la iteración i del bucle 10-14, el algoritmo ha verificado que los conteos de las letras en `conteo_A` y `conteo_B` para todas las letras desde 1 hasta i coinciden. Esto es, si en alguna iteración se detecta una discrepancia, el algoritmo retorna `False`. Si no se detectan discrepancias, se continúa hasta verificar todas las letras del alfabeto (todos los índices)

- **Inicialización:**

  Para este caso se tiene que $i = 1$, ahora, si pasa que
  `conteo_A[abecedario[1]] != conteo_B[abecedario[1]]` entonces el algoritmo retorna `False`, en caso de que ocurra lo opuesto, se pasa a la siguiente iteración. Esto es, el invariante se cumple, pues ha verificado las coincidencias.

- **Mantenimiento:**

Aquí, $2 \leq i \leq 27$.
Durante la iteración $i+1$, el algoritmo compara el conteo de la letra en la posición $i+1$ en `conteo_A` y `conteo_B`. Si los conteos para la letra $i+1$ coinciden, el invariante se mantiene, ya que el algoritmo ha verificado que los conteos de todas las letras desde 1 hasta $i+1$ son correctos.

Si se detecta una discrepancia, el algoritmo retorna `False`, lo que garantiza que el invariante es mantenido porque se ha encontrado una inconsistencia.

- **Terminación:**

  Cuando el bucle termina, todas las letras del alfabeto desde 1 hasta 27 han sido verificadas. Si el algoritmo no ha retornado `False` durante el bucle, esto indica que los conteos de todas las letras en `conteo_A` y `conteo_B` coinciden para todas las letras del alfabeto.

Al finalizar el bucle, el algoritmo retorna `True` si no se encontraron discrepancias. Esto significa que los conteos de todas las letras coinciden en ambos diccionarios, confirmando que los arreglos `A` y `B` son anagramas.

Observación:

A continuación, se explica la complejidad temporal de algunas operaciones que usan los diccionarios y que en particular se usaron en el algoritmo `Anagrama`.

- **Inserción:**
  - **Operación:** Insertar un nuevo par clave-valor en el diccionario.
  - **Complejidad Temporal Promedio:** $O(1)$
  - **Explicación:** En promedio, la inserción en un diccionario tiene un tiempo constante, $O(1)$, gracias a la implementación basada en tablas hash. La clave se convierte en un índice de tabla hash y el par clave-valor se almacena en esa posición. [1]
- **Búsqueda:**
  - **Operación:** Buscar el valor asociado a una clave en el diccionario.
  - **Complejidad Temporal Promedio:** $O(1)$
  - **Explicación:** La búsqueda en un diccionario se realiza en tiempo constante promedio, $O(1)$. La clave se convierte en un índice mediante una función hash y se accede directamente al valor asociado. [2]
- **Acceso a los Elementos:**
  - **Operación:** Obtener el valor asociado a una clave específica.
  - **Complejidad Temporal Promedio:** $O(1)$
  - **Explicación:** Acceder al valor de una clave específica también tiene un tiempo constante promedio, $O(1)$. La función hash convierte la clave en un índice para acceder directamente al valor almacenado en esa posición.

[1] Sin embargo, en el peor de los casos, debido a colisiones, la complejidad puede ser $O(n)$, pero esto ocurre raramente y la mayoría de las veces se maneja con técnicas de resolución de colisiones que mantienen la eficiencia.
[2] En el peor de los casos, debido a las colisiones, la búsqueda podría ser más lenta, pero esto es poco frecuente.

Problema 3

Diseña un algoritmo de tiempo $O(n)$ y espacio $O(1)$ que calcule el mayor *declive* en un arreglo A de enteros de tamaño n, es decir, la diferencia más grande $A[i] - A[j]$, donde $1 \leq i < j \leq n$. Por ejemplo, si el arreglo es $[19, 12, 13, 11, 20, 14]$ el mayor declive sería 8, entre el primero y el 4o. elemento. Si el último valor en vez de ser 14 fuera 10, entonces el mayor declive sería 10, entre los dos últimos elementos.

Solución

Algoritmo Declive

- `Entrada:` Un arreglo A de enteros de tamaño $n \geq 2$.
- `Salida:` La diferencia más grande $A[i] - A[j]$ donde $1 \leq i < j \leq n$

```
1  valor_max = A[1]
2  declive_max = A[2] - A[1]
3
4  for i = 2 up to n
5      declive = valor_max-A[i]
6      if declive > declive_max
7          declive_max = declive
8      end if
9
10     if A[i] > valor_max
11         valor_max = A[i]
12     end if
13 end for
14
15 return declive_max
```

Corrección del Algoritmo

EL invariante de ciclo para el algoritmo es:

Lema: Al inicio de cada iteración del ciclo con índice i, la variable `declive_max` *contiene el valor máximo de la diferencia $A[k] - A[j]$ para todos los pares de índices k y j donde $1 \leq k < j \leq i$.*

- **Inicialización:**

  Al inicio del ciclo, el valor de `declive_max` representa la diferencia más grande encontrada hasta el momento entre los elementos del arreglo, a saber, antes de la primer iteración, dentro de la variable `declive_max` se almacena el valor `A[2] - A[1]`, que hasta el momento es el valor más grande, pues no se ha iniciado el ciclo y por ende no se ha hecho ninguna comparación, por lo que el invariante se cumple.

- **Mantenimiento:**

Hipótesis de inducción: Supongamos que el invariante de ciclo es verdadero para una iteración i, es decir:

1. `declive_max` contiene el valor máximo de la diferencia $A[k] - A[j]$ para todos los pares de índices k y j donde $1 \leq k < j \leq i$.
2. `valor_max` almacena el valor máximo encontrado en las posiciones anteriores a i.

Paso inductivo:
Dado que el invariante es verdadero en la iteración i, debemos demostrar que también es verdadero en la iteración $i + 1$.

En la iteración $i + 1$, se realizan dos comprobaciones:

1. **Actualizar `declive_max`:**
   - Por hipótesis de inducción, la variable `declive_max` representa la diferencia más grande `A[k] - A[j]` encontrada hasta el momento.
     Se calcula `declive` = `valor_max` $- A[i + 1]$. Si `declive` es mayor que `declive_max`, entonces se actualiza `declive_max`. Pero como hasta el momento `A[k] - A[j]` contenía el valor mayor, actualizar `declive_max` implicó que necesariamente el algoritmo encontró un valor mayor.
   - En otras palabras, se verifica si $A[i + 1]$ es mayor que `valor_max`. Si es así, se actualiza `valor_max` con el valor de $A[i + 1]$.
   - Esto asegura que `valor_max` siempre almacene el valor máximo encontrado en las posiciones hasta $i + 1$, por ayuda de la hipótesis.
   - De manera análoga con `valor_max`.

   Al realizar estas dos actualizaciones, se garantiza que el invariante de ciclo sigue siendo verdadero en la iteración $i + 1$. Es decir:

   a) `declive_max` contiene la diferencia máxima entre los elementos $A[k]$ y $A[j]$ donde $1 \leq k < j \leq i + 1$.

   b) `valor_max` almacena el valor máximo encontrado en las posiciones hasta $i + 1$.

- **Finalización:** Durante cada iteración, el algoritmo actualiza `declive_max` si encuentra una diferencia mayor entre `valor_max` y el valor actual $A[i]$, manteniendo el invariante de que `declive_max` es la mayor diferencia encontrada hasta la posición i.
  Por lo demostrado anteriormente, la variable `declive_max` almacena la diferencia más grande $A[i] - A[j]$, por lo que el algoritmo solo devuelve este valor.

Problema 4

Una *cola de prioridades* almacena pares de valores (k, v), donde k es un entero que se le conoce como *llave*, y v es un valor que tiene a k como su llave. Consideremos una versión estática de la estructura de datos que puede almacenar hasta n pares. Las operaciones que provee son las siguientes:

1. $ConstruyeCola(A, m)$: Dado un arreglo A y un entero $m \leq n$, cuyas entradas $A[1, \ldots, m]$ tienen pares (k, v), construye una cola de prioridades con dichas m parejas, y devuelve la cola construida.
2. $Tamano(C)$: devuelve el número de pares en C.
3. $Mete(C, v, k)$: Si C tiene menos de n elementos, entonces agrega el par (k, v) a C; de otra forma devuelve $false$.
4. $SacaMax(C)$: Si C no es vacía, devuelve algún par (k, v) cuya llave k es máxima entre las llaves en C, y remueve (k, v) de C; de otra forma devuelve $false$.
5. $LeeMax(C)$: Si C no es vacía, devuelve algún par (k, v) cuya llave k es máxima entre las llaves en C, sin remover el par; de otra forma devuelve $false$.

Modifica la estructura de datos *Heap* vista en clase para implementar una cola de prioridades. La complejidad de tiempo de $ConstruyeCola(A, m)$ debe ser $O(n \log n)$, y el resto de operaciones deben tener complejidad de tiempo $O(\log n)$. En tu análisis puedes suponer la corrección de $Heapify$, a menos que hagas una modificación muy grande a la función que requiera demostrar que es correcta.

Solución

ConstruyeCola(A,m)

- `Entrada`: Un arreglo A y un entero $m \leq n$, cuyas entradas $A[1, \ldots, m]$ tienen pares (k, v),
- `Salida`: Una cola de prioridades con dichas m parejas.

```
1  Crear C arreglo de tamanio m
2  C.size = m
3
4  if C.size == 0
5      return ColaVacia
6  else
7      for i=1 up to m do
8          C[i] = A[i]
9      end for
10
11     for i = (m // 2) + 1 down to 1 do
12         Heapify(C, i, m)
13     end for
14 end if-else
15
16 return C
```

Tamanio(C)

- `Entrada`: Un cola de prioridades C.
- `Salida`: El número de pares de C.

```
1  return C.size
```

Mete(C, v, k)

- `Entrada`: Una cola de prioridades C, un valor v, y una llave k.
- `Salida`: La cola de prioridades C con el par (k, v) agregado, o $false$ si la cola ya contiene n elementos.

```
1  if C.size >= n
2      return false
3  end if
4
5  C.size = C.size + 1
6  i = C.size
7  while i > 1 and C[i // 2].key < k do
8      C[i] = C[i // 2]
9      i = i // 2
10 end while
11
12 C[i] = (k, v)
13 return C
```

SacaMax(C)

- `Entrada`: Una cola de prioridades C.
- `Salida`: Un par (k, v) con la llave k máxima en C y la cola de prioridades C con dicho par removido, o $false$ si la cola está vacía.

```
1  if C.size == 0
2      return false
3  end if
4
5  max = C[1]
6  C[1] = C[C.size]
7  C.size = C.size - 1
8  Heapify(C, 1, C.size)
9
10 return max
```

LeeMax(C)

- `Entrada:` Una cola de prioridades C.
- `Salida:` Un par (k, v) con la llave k máxima en C sin removerlo, o $false$ si la cola está vacía.

```
1 if C.size == 0
2     return false
3 end if
4
5 return C[1]
```

Análisis de complejidad

- **ConstruyeCola(A, m)**
  La construcción de la cola de prioridades tiene dos partes principales: copiar los elementos de A a C, lo cual toma $O(m)$, y la aplicación de `Heapify` desde el medio del arreglo hacia la raíz, lo cual toma $O(m \log m)$. Por lo tanto, la complejidad total de esta operación es $O(m \log m)$.

- **Mete(C, v, k):** Insertar un nuevo elemento en la cola de prioridades implica colocar el elemento en la última posición del arreglo y luego hacer que 'suba' hasta su posición correcta, lo que tiene una complejidad de $O(\log n)$.

  Explicación del pseudocódigo

  Este fragmento de código en pseudocódigo implementa la operación `Mete(C, v, k)` para insertar un nuevo par (k, v) en una cola de prioridades basada en un heap máximo. A continuación, se explica paso a paso cómo funciona y por qué es correcto:

  1. **Verificar si la cola está llena:**

     ```
     1     if C.size >= n
     2         return false
     3     end if
     ```

     Este bloque de código verifica si la cola ya contiene n elementos. Si es así, no es posible agregar un nuevo elemento, por lo que la función retorna `false`. Esto asegura que no se exceda la capacidad máxima del heap.

  2. **Aumentar el tamaño de la cola:**

     ```
     1     C.size = C.size + 1
     2     i = C.size
     ```

     Aquí, se incrementa el tamaño de la cola en 1 para reflejar la adición del nuevo elemento. Luego, asignamos la variable i al índice donde colocaremos el nuevo elemento, que es la última posición disponible en el heap.

  3. **Reorganizar el heap (Percolate Up):**

```
1       while i > 1 and C[i // 2].key < k do
2           C[i] = C[i // 2]
3           i = i // 2
4       end while
```

Este es el paso donde se mantiene la propiedad de *max-heap*. El nuevo elemento se inserta en la última posición del heap (en la hoja más a la derecha), pero esto podría violar la propiedad de *max-heap* si la clave k es mayor que la clave de su padre.

El ciclo `while` mueve el nuevo elemento hacia arriba en el heap mientras k sea mayor que la clave de su padre, intercambiando posiciones entre el nodo actual y su padre.

4. **Insertar el nuevo elemento en la posición correcta:**

```
1       C[i] = (k, v)
```

Finalmente, el nuevo par (k, v) se coloca en la posición correcta, que es el lugar donde el ciclo `while` se detuvo.

5. **Retornar la cola actualizada:**

```
1       return C
```

Después de insertar el nuevo elemento y reorganizar el heap, se devuelve la cola de prioridades actualizada.

La inserción de un nuevo elemento en un heap tiene una complejidad de $O(\log n)$, ya que el peor caso requiere comparar y mover elementos a lo largo de la altura del árbol, que es $\log n$ en un heap binario completo.

- **SacaMax(C):** Sacar el elemento máximo requiere mover el último elemento del heap a la raíz y luego aplicar `Heapify` para restaurar la propiedad de heap, lo cual tiene una complejidad de $O(\log n)$.

- **SacaMax(C):**

1. **Sacar el elemento máximo:**
   El elemento máximo en un heap máximo siempre está en la raíz del árbol (índice 1 en la representación por arreglos). Para sacarlo, se elimina el elemento de la raíz, lo que deja un 'hueco' en la posición 1.
2. **Mover el último elemento a la raíz:**
   Para mantener la estructura de árbol binario completo, el último elemento del heap (en la última posición del arreglo) se mueve a la raíz (posición 1). Esto preserva la estructura del árbol binario completo, pero puede violar la propiedad de heap máximo, ya que el nuevo elemento en la raíz podría no ser el máximo entre sus descendientes.
3. **Aplicar `Heapify` para restaurar la propiedad del heap:**
   Para restaurar la propiedad de heap máximo, se debe reorganizar el árbol mediante el proceso de `Heapify`, que se aplica comenzando desde la raíz:
   - **Comparaciones:** `Heapify` compara el elemento en la raíz con sus hijos para asegurarse de que el mayor de los hijos esté en la posición adecuada. Si el elemento en la raíz es menor que alguno de sus hijos, se intercambia con el hijo más grande.

- **Desplazamiento:** Después del intercambio, el proceso de `Heapify` se repite de manera recursiva o iterativa hacia abajo en el árbol (en el subárbol afectado) hasta que el elemento se coloque en una posición donde la propiedad de heap máximo ya no se vea violada.

La altura de un heap binario completo con n elementos es $O(\log n)$, ya que en cada nivel del árbol el número de nodos se duplica. El proceso de `Heapify` se ejecuta a lo largo de la altura del árbol (en el peor caso), lo que implica que la operación de `SacaMax(C)` tiene una complejidad de $O(\log n)$.

- **LeeMax(C):** Leer el elemento máximo sin eliminarlo es una operación que simplemente retorna el primer elemento del heap, lo cual tiene una complejidad de $O(1)$.
- **Tamanio(C):** Obtener el tamaño de la cola es una operación $O(1)$ ya que simplemente se retorna el valor de `C.size`.

Problema 5

Haz lo siguiente:

- Considera la recurrencia $T(n) = 2\,T(n/c) + 5n + \sqrt{n}$, donde c es un entero positivo mayor o igual a 2, y $T(1) = 2$. Prueba que $T(n) \in O(n \log n)$.

  Solución:

  Se resuelve la relación de recurrencia. Para ello se hacen las siguientes evaluaciones:

  0. $T(n)$
     Se encuentra $T(n)$
  1. $T(n) = 2\,T(n/c) + 5n + \sqrt{n}$.
     Se encuentra $T(n/c)$.
  2. $T(n/c) = 2\,T(n/c^2) + 5(n/c) + \sqrt{n/c}$.
     Se encuentra $T(n/c^2)$
  3. $T(n/c^2) = 2\,T(n/c^4) + 5(n/c^2) + \sqrt{n/c^2}$.
     Se encuentra $T(n/c^4)$
  4. $T(n/c^4) = 2\,T(n/c^8) + 5(n/c^4) + \sqrt{n/c^4}$.
     Se encuentra $T(n/c^8)$

     $\vdots$

  Se puede apreciar lo siguiente:

  - En el nivel 0 se tiene una contribución de: $5n + \sqrt{n}$
  - En el nivel 1 se tiene una contribución de: $2\left(5n/c + \sqrt{n/c}\right)$
  - En el nivel 2 se tiene una contribución de: $2^2\left(5\dfrac{n}{c^2} + \sqrt{n/c^2}\right)$
  - En el nivel 3 se tiene una contribución de: $2^3\left(5n/c^3 + \sqrt{n/c^3}\right)$

    $\vdots$

Continuando con esta cadena de contribuciones, en el nivel k se tiene:

$$2^k\left(5n/c^k + \sqrt{n/c^k}\right) \text{ contribuciones}$$

En cada nivel del árbol de recurrencia, se divide el problema original de tamaño n en subproblemas de tamaño $\frac{n}{c}$. Esto se repite hasta que el tamaño del subproblema se reduce a un valor en donde es posible aplicar el caso base.

Para encontrar la altura del árbol, se necesita determinar el nivel k en el cual el tamaño del subproblema alcanza el valor mínimo para que ya no se divida más.
Según la condición base $T(1) = 2$, lo que nos interesa es encontrar el valor de k tal que el subproblema tenga un tamaño igual a 2, ya que es el punto en el que nos acercamos al caso base.

Entonces, el árbol de recurrencia continúa dividiendo el problema hasta que:

$$\frac{n}{c^k} = T(1) = 2$$

Este es el momento en el que la división ya no puede continuar más allá del caso base. En este punto, se ha alcanzado la altura máxima del árbol, y k representa el nivel más profundo, por lo que:

$$n/c^k = 2 \longrightarrow n/2 = c^k \longrightarrow k = \log_c(n/2)$$

Esto es, la altura del árbol es:

$$k = \log_c(n/2) \tag{1}$$

Por lo que la complejidad temporal es la suma de las contribuciones de los niveles, matemáticamente hablando se tiene:

$$\begin{aligned}
T(n) &= \sum_{k=0}^{\log_c\left(\frac{n}{2}\right)} 2^k \left(\frac{5n}{c^k} + \sqrt{\frac{n}{c^k}}\right) \\
&= 5n \sum_{k=0}^{\log_c\left(\frac{n}{2}\right)} \frac{2^k}{c^k} + \sqrt{n} \sum_{k=0}^{\log_c\left(\frac{n}{2}\right)} \frac{2^k}{\sqrt{c^k}} \\
&= 5n \sum_{k=0}^{\log_c\left(\frac{n}{2}\right)} \left(\frac{2}{c}\right)^k + \sqrt{n} \sum_{k=0}^{\log_c\left(\frac{n}{2}\right)} \left(\frac{2}{\sqrt{c}}\right)^k \\
&\leq 5n \sum_{k=0}^{\infty} \left(\frac{2}{c}\right)^k + \sqrt{n} \sum_{k=0}^{\infty} \left(\frac{2}{\sqrt{c}}\right)^k
\end{aligned}$$

Ahora, como $c \geq 2$, entonces se cumple que $\dfrac{2}{c} \leq 1$, por lo que para el sumando izquierdo se usa la serie geométrica:

$$\sum_{k=0}^{\infty} ar^k = \frac{a}{1-r} \quad \text{si} \quad |r| \leq 1 \tag{2}$$

Entonces, para el primer sumando:

$$\sum_{k=0}^{\infty} \left(\frac{2}{c}\right)^k = \frac{1}{1-2/c} = \frac{c}{c-2}$$

Mientras que para el segundo sumando:

$$\sum_{k=0}^{\infty} \left(\frac{2}{\sqrt{c}}\right)^k = \frac{1}{1-2/\sqrt{c}} = \frac{\sqrt{c}}{\sqrt{c}-2}$$

Por lo que se tiene lo siguiente:

$$T(n) \leq 5n \sum_{k=0}^{\infty} \left(\frac{2}{c}\right)^k + \sqrt{n} \sum_{k=0}^{\infty} \left(\frac{2}{\sqrt{c}}\right)^k$$

$$T(n) \leq 5n \frac{c}{c-2} + \sqrt{n} \frac{\sqrt{c}}{\sqrt{c}-2}$$

$$T(n) \leq An + B\sqrt{n}$$

En este momento basta probar que $An + B\sqrt{n} \in O(n \log n)$.

Se demuestra caso por caso, esto es:

- $An \in O(n \log n)$. Lo cual es cierto.
- $B\sqrt{n} \in O(n \log n)$.
  Sea $n_0 = 1$, y sea n tal que $n \geq n_0$, es decir, $n \geq 1$.
  Como $n \geq 1$ entonces es cierto que

  $$n \leq n^2 \longrightarrow \sqrt{n} \leq n$$

  Nuevamente por $n \geq 1$, $\log n \geq 1$ entonces:

  $$\sqrt{n} \leq n \log n$$

  Así, existe $n_0 = 1$ y $c = 1$ tal que $\sqrt{n} \leq n \log n$, por lo que $\sqrt{n} \in O(n \log n)$

En conclusión, se satisface $An + B\sqrt{n} \in O(n \log n)$, lo que implica que necesariamente $T(n) \in O(n \log n)$.
Que era lo que se quería probar.

- Considera la recurrencia $T(n) = T(n-1) + n/3$, y $T(1) = 264$. Prueba que $T(n) \notin O(n \log n)$.

  Solución:

  Se tiene lo siguiente:

- $T(1) = 264$
- $T(2) = T(1) + \dfrac{2}{3}$
- $T(3) = T(1) + \dfrac{2+3}{3}$
- $T(4) = T(1) + \dfrac{2+3+4}{3}$
- $\vdots$
- $T(n-1) = T(1) + \dfrac{2+3+4+\ldots+n-1}{3}$
- $T(n) = T(1) + \dfrac{2+3+4+\ldots+n-1+n}{3}$

Entonces, se cumple que:

$$\begin{aligned} T(n) &= T(1) + \frac{1}{3}\sum_{k=2}^{n} k \\ &= T(1) + \left(\frac{1}{3}\sum_{k=1}^{n} k\right) - \frac{1}{3} \\ &= T(1) + \frac{n(n+1)}{3} - \frac{1}{3} \\ &= 264 - \frac{1}{3} + \frac{n(n+1)}{3} \end{aligned}$$

De esta manera, $T(n) \in O(n^2)$, por lo que es cierto que $T(n) \notin O(n \log n)$.
Para verificar esto último, suponga que no es verdad lo anterior, esto es; suponga que $T(n) \in O(n \log n)$.

Dado lo anterior, entonces, existen constantes $n_0, C > 0$ tal que, para toda $n \geq n_0$, $An^2 \leq Cn \log n$, dividiendo entre n:

$$An \leq C \log n \longrightarrow \frac{An}{\log n} \leq C$$

Entonces, si $n \longrightarrow \infty$ se tiene:

$$\infty \leq C$$

Esto quiere decir que se tendría un valor de C que también fuese infinito, y que acote a la función $T(n)$, pero esto no puede ser posible. Así, el error es suponer que $T(n) \in O(n \log n)$, entonces, por contradicción, es cierto que $T(n) \notin O(n \log n)$, que era lo que se quería probar.

- Demuestra que $\log_a(n^b) + O(\log n) \in O(\log n)$, con $a, b > 1$.

  Solución:

  Por propiedades de los logaritmos:

$$\log_a(n^b) = b \log_a(n)$$

$$b \log_a(n) = b \, \frac{\log(n)}{\log(a)}$$

Tómese $c = \frac{b}{\log(a)}$, por lo que:

$$b \log_a(n) \leq c \log(n)$$

Entonces, $b \log_a(n) \in O(log\ n)$.

Así, como $b \log_a(n) \in O(log\ n)$, en la expresión inicial es cierto que:

$$O(\log n) + O(\log n) \in O(\log n)$$

Por lo tanto, se cumple la afirmación.