Algoritmos y Complejidad INF 221
Listado de ejercicios*

Departamento de Informática - UTFSM - Semestre 2024 - II

1. Encuentre funciones $g(n), g'(n), g''(n)$, tal que $f(n)$ sea $O(g(n)), \Omega(g'(n)), \Theta(g''(n))$, para cada uno de los siguientes casos:

   (a) $f(n) = n^3 + 70n^2 + n$

   (b) $f(n) = 2^n + 5^n$

   (c) $f(n) = \log_2(n) + \sqrt{n}$

   (d) $f(n) = \sum_{i=1}^{n} i$

   (e) $f(n) = \sum_{i=1}^{n} \frac{1}{i}$

Solución:

(a) $f(n) = n^3 + 70n^2 + n$

- $f(n) \in O(n^3)$, con $c = 72$ y $n_0 = 0$
- $f(n) \in \Omega(n^3)$, con $c = 1$ y $n_0 = 0$
- $f(n) \in \Theta(n^3)$, con $c_1 = 1$, $c_2 = 72$ y $n_0 = 0$

(b) $f(n) = 2^n + 5^n$

- $f(n) \in O(5^n)$, con $c = 2$ y $n_0 = 0$
- $f(n) \in \Omega(5^n)$, con $c = 1$ y $n_0 = 0$
- $f(n) \in \Theta(5^n)$, con $c_1 = 1$, $c_2 = 2$ y $n_0 = 0$

(c) $f(n) = \log_2(n) + \sqrt{n}$

- $f(n) \in O(\sqrt{n})$, con $c = 2$ y $n_0 = 0$

*Ejercicios basados en guías de ejercicios desarrolladas por profesores: Roberto Asín y José Fuentes

- $f(n) \in \Omega(\sqrt{n})$, con $c = 1$ y $n_0 = 0$
- $f(n) \in \Theta(\sqrt{n})$, con $c_1 = 1$, $c_2 = 2$ y $n_0 = 0$

(d) $f(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2+n}{2}$

- $f(n) \in O(n^2)$, con $c = 1$ y $n_0 = 0$
- $f(n) \in \Omega(n^2)$, con $c = \frac{1}{2}$ y $n_0 = 0$
- $f(n) \in \Theta(n^2)$, con $c_1 = \frac{1}{2}$, $c_2 = 1$ y $n_0 = 0$

(e) $f(n) = \sum_{i=1}^{n} \frac{1}{i}$

La expresión $\sum_{i=1}^{n} \frac{1}{i}$ es conocido como el n-ésimo número de la serie armónica, H_n. Para encontrar el límite de esta sumatoria, dividimos el rango $[1..n]$ en $\lfloor \lg n \rfloor + 1$ segmentos, para luego encontrar un límite superior a cada segmento. Para $k = 0, 1 \ldots, \lfloor \lg n \rfloor$, el segmento k estará compuesto por los elementos del rango $[\frac{1}{2^i}, \frac{1}{2^{i+1}})$. Eventualmente, el último segmento estará incompleto. De esta manera, tenemos

$$\sum_{i=1}^{n} \frac{1}{i} \leq \sum_{k=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^k-1} \frac{1}{2^k + j} \leq \sum_{k=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^k-1} \frac{1}{2^k} = \sum_{k=0}^{\lfloor \lg n \rfloor} 1 \leq \lg n + 1$$

Por lo tanto, la función $f(n) = \sum_{i=1}^{n} \frac{1}{i}$ está limitada superiormente por $O(\lg n)$.

2. Analice la complejidad temporal para el peor caso del algoritmo PalindromeCheck, que determina si la cadena de caracteres es un palíndromo, es decir, se lee de la misma manera de derecha a izquierda que de izquierda a derecha. Para el caso promedio asuma que la mitad de las veces la cadena no es un palíndromo y las diferencias pueden empezar en cualquier caracter.

```
1  Procedure PalindromeCheck(S)
2      output ← True
3      n ← length(S)
4      h ← ⌊n/2⌋
5      i ← 1
6      while output and i ≤ h do
7          if S[i] ≠ S[n + 1 − i] then return False
8          else i ← i + 1
9      end
10     return True
```

Solución:

- **Peor caso**: En el peor caso, se realizan $\lfloor \frac{n}{2} \rfloor$ comparaciones. Esto ocurre cuando la cadena de caracteres es un palíndromo. Asumiendo que la función LENGTH() y que cada comparación toma tiempo constante,
  PALINDROMECHECK(S)$\in O(n)$
- **Caso promedio**: Para el caso promedio, la mitad de las veces la cadena es un palíndromo. Esto quiere decir, que con probabilidad $p = \frac{1}{2}$, realizamos $\lfloor \frac{n}{2} \rfloor$ comparaciones. La otra mitad de los casos, la cadena no es un palíndromo, por lo que encontraremos un par de símbolos distintos (línea 7 del algoritmo) entre las primeras $\lfloor \frac{n}{2} \rfloor$ comparaciones. Asumiendo una distribución uniforme, encontrar el primer par distinto en la i-ésima comparación tiene probabilidad $\frac{2}{n}$. De esta manera, la posición promedio donde encontraremos el primer par distinto está dada por $\frac{2}{n} \sum_{i=1}^{n/2} i = \frac{2}{n}(\frac{\frac{n}{2}(\frac{n}{2}+1)}{2}) = \frac{1}{n}(\frac{n^2}{4} + \frac{n}{2}) = \frac{n}{4} + \frac{1}{2}$.

  En resumen, en el caso promedio, con la misma probabilidad $p = \frac{1}{2}$ hacemos $\lfloor \frac{n}{2} \rfloor$ o $\frac{n}{4} + \frac{1}{2}$ comparaciones. Por lo tanto, en promedio, hacemos $\frac{1}{2}\lfloor \frac{n}{2} \rfloor + \frac{1}{2}(\frac{n}{4} + \frac{1}{2}) = \frac{n}{4} + \frac{n}{8} + \frac{1}{4} = \frac{3n}{8} + \frac{1}{4}$ comparaciones, lo que corresponde a una complejidad temporal de $O(n)$.

3. El siguiente algoritmo encuentra la representación binaria de un entero decimal positivo. El algoritmo se basa en divisiones sucesivas por 2. Así, para un entero positivo N se calcula:

$$\begin{aligned} N &= 2q_0 + r_0 \\ q_0 &= 2q_1 + r_1 \\ q_1 &= 2q_2 + r_2 \\ &\dots \\ q_{k-1} &= 2q_k + r_k \end{aligned}$$

   donde cada residuo r_i es 0 o 1. El algoritmo se detiene cuando $q_k = 0$: La representación binaria es entonces $N_{(2)} = r_k, r_{k-1}, \dots, r_1, r_0$:

```
1  Procedure RepresentacionBinaria(N)
2      y ← N
3      i ← 0
4      while y ≠ 0 do
5          if esPar(y) then r[i] ← 0
6          else
7              r[i] ← 1
8              y ← y − 1
9          y ← ⌊y/2⌋
10         i ← i + 1
11     end
12     return invertir(r[i − 1.,0])
```

¿Cuál es la complejidad de este algoritmo?

Solución:

- El ciclo while se ejecuta $\lceil \log_2(N) \rceil$ veces. Esto se debe a que la variable y se divide por dos iterativamente hasta que se cumpla $\lceil \frac{y}{2^i} \rceil = 0$. Cuando se cumple ese caso, tenemos que $2^i > y \Rightarrow i > \log_2(y) = \log_2(N)$
- Dentro del ciclo while, se ejecutan $O(1)$ operaciones
- La función invertir() toma tiempo proporcional a la cantidad de veces que se repite el ciclo while, $\lceil \log_2(N) \rceil$.

Por lo tanto, el algorithmo RepresentacionBinaria(N) toma tiempo $O(\log_2 N)$.

4. Dado el siguiente algoritmo:

```
1  Procedure Algo(X)
2      n ← length(X)
3      b ← X[1]
4      for j ← 2 to n do
5          if X[j] < b then b ← X[j]
6      end
7      return b
```

Realice el análisis temporal y espacial para el algoritmo. ¿Qué computa?

Solución: El algoritmo computa el elemento mínimo dentro del arreglo X. El algoritmo toma tiempo $\Theta(n)$ para ejecutarse y consume $O(n \log_2 |U|)$ bits. El consumo

de espacio se desprende del espacio requerido por el arreglo X, donde tenemos n elementos y cada elemento pertenece al dominio $X[i] \in U, 0 \leq i < n$.

5. Existe una algoritmo de búsqueda conocido como *búsqueda doblada*. Este algoritmo realiza una búsqueda de un elemento p sobre un arreglo de elementos *ordenados* y *no repetidos* A, siguiendo estos pasos:

   1. Sea idx la posición dentro del arreglo A que se está observando, siendo inicialmente $idx = 1$ (Para un arreglo que comienza en 1).
   2. Comprobar si $A[idx] = p$. Si la comprobación resulta ser cierta, termina el algoritmo retornando idx. Si $A[idx] < p$, prosigue con el paso 3. En caso contrario, continua con el paso 4.
   3. Redefine idx como $2 \cdot idx$ y va al paso 2.
   4. Realiza una búsqueda binaria en el segmento comprendido entre 2^{idx-1} y 2^{idx}. Si encuentra el valor p en dicho segmento, termina retornando el índice donde está la ocurrencia, en caso contrario, termina retornando un mensaje de error.

   Dado lo anterior, ¿Cuál es la complejidad de dicho algoritmo?

Solución:

El algoritmo encuentra la posición k, tal que $A[k] = p$.

1. En los puntos 2 y 3 se encuentra el primer valor x tal que $A[2^x] \geq p$. Encontrar x toma tiempo $O(x)$.
2. Aplicamos búsqueda binaria (punto 4) sobre el rango $A[2^{x-1}.,2^x]$. Esto implica que, de existir k, $2^{x-1} < k \leq 2^x$. La búsqueda binaria en ese rango toma tiempo $O(\log_2(2^x - 2^{x-1})) = O(\log_2(2^x)) = O(x)$.

Por lo tanto, el algoritmo toma tiempo $O(x)$. Ya que $2^{x-1} < k \leq 2^x$, tenemos que $O(x) = O(\log_2 k)$.

6. Una expresión aritmética construida con los operadores aritméticos binarios '+', '–', '*', '/' y operandos de un único dígito entre 0 y 9, se dice que está en forma posfija si es o bien un sólo operando o dos expresiones en forma posfija una detrás de otra seguidas inmediatamente de un operador. A continuación se muestra una expresión escrita en la notación infija habitual junto con su forma posfija:

   - Forma infija: (3 + 5/9)×(6 – 9)
   - Forma posfija: 3 5 9 / + 6 9 – ×

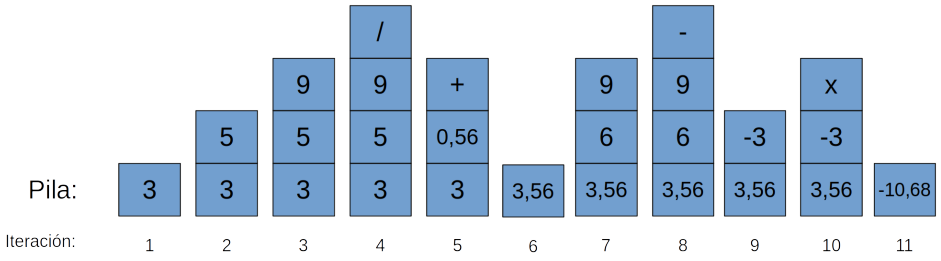
(a) Diseña un algoritmo iterativo que calcule el valor de una expresión dada en forma posfija.

(b) Diseña un algoritmo que dada una expresión en forma infija genere su versión posfija.

Solución:

(a) Una alternativa es diseñar un algoritmo basado en una pila. Nuestro algoritmo comienza recorriendo la expresión en notación posfija de izquierda a derecha. Cada vez que se visita un nuevo elemento (operador u operando), este se agrega a la pila. Si lo que se inserta es un operando, entonces continuamos con el siguiente elemento en la expresión posfija. En cambio, si lo que se inserta es un operador, entonces desapilamos los primeros tres elementos, los que corresponden a un operador y dos operandos. Realizamos la operación matemática correspondiente para luego insertar el resultado en la pila. Aplicamos este procedimiento hasta recorrer toda la expresión. El resultado final quedará como único elemento en la pila.

A continuación, se muestra un seguimiento de la pila usando el ejemplo 3 5 9 / + 6 9 − ×:



(b) Queda como tarea para la/el estudiante.

7. Pruebe que **selection sort** es un algoritmo correcto usando invariantes ¿Por qué se ejecuta sólo sobre $n-1$ elementos en lugar de todos los n elementos a ordenar? Obtenga la complejidad $\Theta(\cdot)$ para su peor y mejor caso.

Solución: La invariante de selection sort indica que al finalizar la i-ésima iteración, las primeras i posiciones contienen los i elementos menores del arreglo, ordenados de menor a mayor. Dado un arreglo a ordenar A, al finalizar la primera iteración, la posición $A[1]$ albergará el menor valor dentro del arreglo (caso base). Asumimos que la invariante se cumple para la iteración $i = k$. Luego, al iniciar la iteración

$i = k + 1$, tenemos que las primeras k posiciones contienen a los k valores menores de A. Luego, entre los restantes valores $A[k + 1...n - 1]$, buscamos el menor valor $A[l], k+1 \leq l < n$. Finalmente, hacemos un intercambio de valores entre las posiciones $k + 1$ y l, terminando la iteración. De esta manera, al terminar la iteración, las primeras $k + 1$ contienen a los $k + 1$ valores menores, lo que cumple la invariante.

Luego de finalizar $n - 1$ iteraciones, el subarreglo $A[1..n - 1]$ contiene a los $n - 1$ valores menores del arreglo. Por lo tanto, $A[n]$ contiene al elemento más grande. Es por esto que sólo se necesitan $n - 1$ iteraciones.

El peor y mejor caso de selection sort es el mismo. Es decir, en cada iteración i, siempre debemos recorrer el subarreglo $A[i..n]$ para buscar el elemento menor. Sin tener información adicional, no hay manera de evitar recorrer ese subarreglo completo. Por lo tanto, en ambos casos, la complejidad es $\Theta(n^2)$.

8. Usando inducción matemática, demuestre que la solución para la siguiente recurrencia es $T(n) = n \lg n$, asumiendo que n es una potencia de 2

$$T(n) = \begin{cases} 2 & \text{if } n = 2 \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

Solución:

- **(Caso base)** Para $n = 2^1$, $T(n) = 2 \lg_2 2 = 2$.
- Asumimos que para $n = 2^k$, $T(n) = n \lg n = 2^k \lg 2^k = 2^k k$.
- Para $n = 2^{k+1}$

$$\begin{aligned} T(n) &= 2T(2^{k+1}/2) + 2^{k+1} \\ &= 2T(2^k) + 2^{k+1} \\ &= 2 \cdot 2^k k + 2^{k+1} \\ &= 2^{k+1}(k + 1) \\ &= 2^{k+1} \lg 2^{k+1} \\ &= n \lg n \end{aligned}$$

Por lo tanto, queda demostrado que $T(n) = n \lg n$ cuando n es una potencia de 2.

9. Se habla de algoritmos de ordenamiento que son estables y no estables. Un algoritmo se dice estable, si dados dos elementos iguales $A[i] = A[j]$, donde $i \leq j$, en el arre-

glo desordenado, estos elementos quedan en el arreglo ordenado en posiciones k y l, respectivamente, donde $k \leq l$. Indique si los siguientes algoritmos son o no estables:

- Selection
- Insertion
- Bubblesort
- Mergesort
- Quicksort

¿Qué tendría que hacer, sin modificar la idea de los algoritmos, para lograr que los algoritmos que no son estables lo sean?

Solución:

- **Selection sort**: La versión clásica de selection sort, basada en arreglos, es inestable. Esto se debe a que en la iteración i del algoritmo, se realiza un intercambio (swap) entre el menor valor en el rango $A[i..n-1]$ y el valor almacenado en $A[i]$. Ese intercambio puede generar un salida no estable. Para hacer selection sort estable, debemos reemplazar la operación de intercambio por una inserción en la posición i del arreglo a ordenar. Insertar un nuevo elemento en un arreglo de largo n toma tiempo $O(n)$, por lo que si queremos mantener la complejidad de selection sort, debemos reemplazar el arreglo por una lista enlazada, en la cual la inserción toma tiempo constante.
- **Insertion sort**: Es estable.
- **Bubblesort sort**: Es estable.
- **Mergesort**: Es estable.
- **Quicksort**: Quicksort no es estable, debido al swap o intercambio de valores que se realiza al separar los elementos que son menores y mayores que el pivote. La operación de intercambio or swap nos permite reutilizar el espacio del arreglo de entrada para hacer el ordenamiento. Una manera de hacer Quicksort estable es usando más espacio, creando una lista para los valores menores al pivote y una lista para los valores mayores al pivote. A medida que se comparan los elementos con el pivote y su posterior inserción a la lista adecuada, es importante mantener el order relativo de esos elementos respecto a su posición en el arreglo de entrada.

10. Use argumentos para rebatir o apoyar lo siguiente: Dado que mergesort usa sudivisión del arreglo para ordenar, alguien ha pensado en lograr una mayor eficiencia si en vez de usar 2 particiones, se usan 4.

Solución: La expresión es incorrecta, ya que ambos casos tienen la misma complejidad algorítmica. Esto se puede apreciar claramente al analizar las recurrencias para el caso de 2 y 4 particiones

2 particiones

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{if } n > 1 \end{cases}$$

4 particiones

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 4T(\frac{n}{4}) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Ambas recursiones cumplen el segundo caso del Teorema Maestro: (2 particiones) $a = 2$, $b = 2$ y $f(n) = \Theta(n)$, cumple que $f(n) = \Theta(n^{\log_2 2})$; y (4 particiones) $a = 4$, $b = 4$ y $f(n) = \Theta(n)$, cumple que $f(n) = \Theta(n^{\log_4 4})$. De esta manera, ambos casos tienen complejidad temporal $\Theta(n \log_2 n)$.

11. Considere los siguientes casos de arreglos y entregue la complejidad de los algoritmos insertion sort, quicksort y mergesort (explique):

    (a) un arreglo ordenado en forma creciente

    (b) un arreglo ordenado en forma decreciente

    (c) un arreglo donde todos lo elementos son iguales

    (d) un arreglo donde la mitad inferior del arreglo tiene los números ordenados que debieran estar en la parte superior del arreglo (ej. 4 5 6 1 2 3).

Solución:

Comentario 1: Algoritmos de ordenamiento que tomen ventaja de rangos ordenados dentro del arreglo de entrada son conocidos como *algoritmos de ordenamiento adaptativos* (adaptive sorting algorithms).

Comentario 2: Mergesort no toma ventaja de rangos de elementos ordenados en la entrada. Su complejidad siempre es $\Theta(n \lg n)$. En el caso de Quicksort, este siempre hará al menos $\Omega(n \lg n)$ comparaciones. Todo depende de la elección del pivote. Si seleccionamos un mal pivote, obtendremos tiempo $O(n^2)$. Si seleccionamos la mediana, obtendremos una complejidad $\Theta(n \lg n)$ (sin contar el costo de encontrar la mediana).

(a) un arreglo ordenado en forma creciente

- **Insertion sort**: $\Theta(n)$, ya que no se realizan ningún swap o intercambio de valores.

- **Quicksort**: Si el pivote es el primer elemento, entonces obtenemos complejijdad $O(n^2)$. Si seleccionamos el elemento de la mitad (mediana), obtenemos tiemmpo $\Theta(n \lg n)$.

(b) un arreglo ordenado en forma decreciente

- **Insertion sort**: Este es el peor caso para insertion sort. Cuando insertamos el i-ésimo elemento, debemos hacer i comparaciones/intercambios. Por lo tanto, obtenemos una complejidad $O(n^2)$.
- **Quicksort**: Similar al punto (a)

(c) un arreglo donde todos lo elementos son iguales

- **Insertion sort**: Similar al punto (a)
- **Quicksort**: Similar al punto (a)

(d) un arreglo donde la mitad inferior del arreglo tiene los números ordenados que debieran estar en la parte superior del arreglo (ej. 4 5 6 1 2 3).

- **Insertion sort**: En este caso, cuando el algoritmo itere sobre la primera mitad, obtendremos una complejidad $O(n)$, ya que se trata de un arreglo ordenado (similar al punto (a)). Por otro lado, cuando el algoritmo itere sobre los elementos de la segunda mitad, todos ellos tendrán que ser comparados contra todos los elementos de la primera mitad, dándonos una costo temporal de $O(\frac{n}{2} \cdot \frac{n}{2}) = O(n^2)$.
- **Quicksort**: Si en la primera llamada recursiva del algoritmo tomamos como pivote al primer elemento, entonces obtendremos al arreglo ya ordenado. No obstante, si mantenemos ese criterio para elegir al pivote en las siguientes llamadas recursivas, obtendremos el peor caso para los dos arreglos ya ordenados, ambos de largo $\frac{n}{2}$. Esto nos da una complejidad total de $O(n^2)$. Situaciones similares ocurren con otros pivotes.

12. Considere el siguiente algoritmo de ordenamiento y realice el seguimiento del proceso de ordenar la siguiente secuencia: 3 1 4 1 5 9 2 6 5 3. Indique cómo se hace el ordenamiento, si el algoritmo es estable y su complejidad para el mejor caso (indique cuál es ese caso).

```
1  Procedure shellSort(A)
2      n ← length(A)
3      h ← ⌊n/2⌋
4      while h > 0 do
5          for i ← h + 1 to n do
6              j ← i − h
7              while j > 0 do
8                  if A[j] > A[j + h] then
9                      intercambiar(A[j], A[j + h])
10                     j ← j − h
11                 else j ← 0
12             end
13         end
14         h ← ⌊h/2⌋
15     end
```

Solución: El ordenamiento y la explicación se dejan como tarea para la/el estudiante. Shellsort pertenece a la familia de los algoritmos no estables. Su mejor caso corresponde a ordenar un arreglo ya ordenado. Tener un arreglo ordenado provoca que la comparación de la línea 8 nunca se cumpla, por lo que el costo computacional de las líneas 6–12 es $\Theta(1)$ en cada iteración del ciclo **for**.

Por lo tanto, la complejidad está dada por la sumatoria $(n-\frac{n}{2})+(n-\frac{n}{4})+(n-\frac{n}{8})+\ldots+1$. El ciclo **while** de la línea 4 se ejecuta $\lceil \lg n \rceil$ veces, por lo que la sumatoria se puede expresar como $n \lg n - n \sum_{i=1}^{\lg n} (\frac{1}{2})^i = n \lg n - n = O(n \lg n)$.

13. Considere el caso de un algoritmo que multiplica matrices $C = A \times B$ de $n \times n$, con elementos en C definidos por $c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$, lo que implica un algoritmo que tiene 3 ciclos (uno para i, otro para j y otra para k).

    Para usar una estrategia de dividir para conquistar en la multiplicación de matrices, uno puede dividir el problema en submatrices a resolver. Considere el caso de crear 4 submatrices de $(n/2) \times (n/2)$ elementos. En este caso, nos queda una:

$$\begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

    Lo que nos da las siguientes submatrices a computar:

$$\begin{aligned} C_{1,1} &= A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1} \\ C_{1,2} &= A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2} \\ C_{2,1} &= A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1} \\ C_{2,2} &= A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2} \end{aligned}$$

Entonces, por cada submatriz $C_{i,j}$ se realiza dos multiplicaciones de submatrices y una suma del resultado de ello. En base a lo explicado, indique el sistema recurrente y la complejidad del algoritmo de multiplicación con la estrategia de dividir para conquistar.

Solución:

$$T(n) = 8T(n/2) + n^2, a = 8, b = 2 \text{ y } f(n) = n^2, n > 1$$

El orden de complejidad de este algoritmo esta dado por $\Theta(n^3)$, en base al primer caso del teorema maestro ($\varepsilon = 1$) y es igual a un algoritmo clásico de multiplicación.

14. Considere la siguiente recurrencia $T(n) = 2T(\lfloor n/2 \rfloor) + n$. ¿Cuál es su complejidad?

Solución:

$$T(n) = 2T(\lfloor n/2 \rfloor) + n, a = 2, b = 2 \text{ y } f(n) = n, n > 1$$

El orden de complejidad de este algoritmo esta dado por $\Theta(n \log_2 n)$, en base al segundo caso del teorema maestro.

15. Considere la siguiente recurrencia $T(n) = 3T(n/4) + cn^2$ ¿Cuál es su complejidad?

Solución:

$$T(n) = 3T(n/4) + cn^2, a = 3, b = 4 \text{ y } f(n) = cn^2, n > 1$$

El orden de complejidad de este algoritmo esta dado por $\Theta(n^2)$, en base al tercer caso del teorema maestro ($\varepsilon = 2 - \log_4 3$ y $c' = 3/16$).

16. Dada una lista ordenada con representación enlazada sencilla, escriba un algoritmo para insertar un elemento, eliminar un elemento y buscar un elemento, luego obtenga la complejidad temporal de los mismos.

Solución: La complejidad de buscar un elemento en una lista ordenada de n elementos tiene complejidad temporal $O(n)$, ya que en el peor caso se debe recorrer la lista completa. Tanto la inserción como la eliminación de un elemento toma tiempo constante una vez que el elemento ha sido identificado, ya que sólo debemos mover

una cantidad constante de punteros o referencias. Por otro lado, para identificar la posición del elemento a insertar/eliminar, debemos realizar una búsqueda. En conclusión, la inserción/eliminación de un elemento tiene complejidad temporal $O(n)$, incluyendo la búsqueda.

El diseño de los algoritmos se deja como tarea para la/el estudiante.

17. Utilizando dos pilas de símbolos diseñe un algoritmo iterativo que decida si una frase es o no palíndroma.

Solución: Asumiendo que la frase de entrada tiene largo n, un algoritmo sería el siguiente

1. Recorrer los primeros $\lfloor \frac{n}{2} \rfloor$ símbolos de izquierda a derecha, insertándolos en una pila A.
2. Recorrer los últimos $\lfloor \frac{n}{2} \rfloor$ símbolos de derecha a izquierda, insertándolos en una segunda pila B.
3. Desapilar símbolos de ambas pilas a la vez. Si los símbolos desapilados son distintos, entonces sabemos que la frase no es palíndroma. Si los símbolos desapilados son iguales, repetimos el proceso, desapilando dos nuevos símbolos, uno de cada pila. Si luego de $\lfloor \frac{n}{2} \rfloor$ iteraciones ambas pilas están vacías, entonces la frase es palíndroma.

18. Represente una matriz bi-dimensional en base a arreglo de una dimensión.

Solución: Sea M una matriz con n filas y m columnas. La matriz M puede ser representada por medio de un arreglo A de $n \times m$ entradas. Acceder a la celda $M[i][j]$ es equivalente a acceder a la entrada $A[i \times m + n]$.

19. Utilizando la idea de árbol binario de búsqueda, escriba un algoritmo de ordenamiento a partir de un arreglo de n elementos desordenados. ¿Cuál es su orden de complejidad?.

Solución: El algoritmo consta de los siguientes pasos

1. Primero, insertamos los n elementos en un árbol binario de búsqueda balanceado (AVL). Para hacer el análisis de este paso, debemos considerar que la altura

del árbol va aumentando a medida que se agregan nuevos elementos. Específicamente, el primer elemento necesita una operación para ser insertado (crear el nodo que lo almacena), los siguientes dos elementos toman dos operaciones (bajar por el árbol de altura uno y crear los nodos), los siguientes 4 elementos toman 3 operaciones, etc. En general, los 2^{i-1} elementos del nivel $i-1$ toman $i2^{i-1}$ operaciones para ser insertados (i operaciones por elemento). De esta manera, asumiendo que obtenemos un árbol binario completo, los n elementos son insertados en $\sum_{i=1}^{\log_2 n} i2^{i-1}$ operaciones. Por lo tanto, la complejidad de este paso está dado por

$$\sum_{i=1}^{\log_2 n} i2^{i-1} \leq \sum_{i=0}^{\log_2 n} i2^i = \frac{2^{\log_2(n)+2}\log_2 n - 2^{\log_2(n)+1}(\log_2(n)+1)+2}{(2-1)^2}$$
$$= 4n\log_2 n - 2n\log_2 n - 2n + 2 = 2n\log_2 n - 2n + 2 = O(n\log_2 n)$$

Donde $\sum_{i=0}^{\log_2 n} i2^i$ es una serie geométrica.

2. El segundo paso consta de un recorrido en pre-order del árbol, lo que toma tiempo $O(n)$.

Por lo tanto, este algoritmo de ordenamiento toma tiempo $O(n\log_2 n+n) = O(n\log_2 n)$.

20. Entregue las cotas superior e inferior para la altura de
    (a) Un árbol binario con n nodos
    (b) Un árbol general con n nodos

Solución: El desarrollo matemático se dejará como tarea para la/el estudiante

(a) Un árbol binario con n nodos tiene una altura acotada por $\Omega(\lg_2 n)$ y $O(n)$. La cota $\Omega(\lg_2 n)$ se puede transformar a $\Theta(\lg_2 n)$ para árboles binarios balanceados. La cota $O(n)$ se alcanza cuando el árbol tiene una forma de una lista de n elementos.

(b) Para el caso de un árbol general con n nodos, podemos acotar tu altura por $\Omega(1)$ y $O(n)$, siendo los casos extremos que $n-1$ nodos sean hijos de la raíz y un árbol con forma de lista enlazada, respectivamente.

21. Demuestre que $f(n) = n\lg n + O(n)$ implica que $f(n) = \Theta(n\lg n)$

Solución: El siguiente desglose muestra los pasos para completar la demostración:

Recordemos las definiciones de las notaciones asintóticas $O(\cdot)$ y $\Theta(\cdot)$.

- $f(n) = O(g(n))$ implica que $\exists\ c, n_0 > 0$, tal que $0 \leq f(n) \leq cg(n)$, $\forall n \geq n_0$.
- $f(n) = \Theta(g(n))$ implica que $\exists\ c_1, c_2, n_0 > 0$, tal que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, $\forall n \geq n_0$.

Para simplificar la explicación, llamaremos $h(n) = O(n)$ al segundo componente de la función $f(n)$ del enunciado. Según la definición de $O(\cdot)$, sabemos que $\exists\ c_h, n_h > 0$, tal que $0 \leq h(n) \leq c_h n$, $\forall n \geq n_h$. De igual manera, si restringimos $n_h \geq 1$ y mantenemos c_h sin cambios, podemos derivar que $h(n) = O(n \lg n)$ (una cota superior menos ajustada).

Con lo anterior, podemos definir $c_1 = 1, c_2 = c_h + 1$ y $n_0 = n_h$ para demostrar que $f(n) = \Theta(n \lg n)$.

22. Asuma que disponemos de una lista de n elementos, donde cada elemento está compuesto por un valor numérico y un color del conjunto {rojo, azul, verde}. La lista está ordenada según los valores numéricos. Diseñe un algoritmo lineal, $O(n)$, para ordenar la lista por color (asuma rojo < azul < verde), tal que todos los elementos que tienen el mismo color se mantienen ordenados por valor numérico. Por ejemplo, dada la lista (1,azul), (3,rojo), (4,azul), (6,verde), (9,rojo) debería resultar (3,rojo), (9,rojo), (1,azul), (4,azul), (6,verde).

Solución: Una posible solución es utilizar counting sort. Utilizaremos L y M para referirnos a las listas de entrada y salida, respectivamente. La solución es la siguiente:

1. Creamos un arreglo de largo 3, el cual nos servirá para contar la cantidad de ocurrencias de los pares rojos, azules y verdes. Llamaremos a este arreglo C e inicialmente $C[rojo] = 0$, $C[azul] = 0$ y $C[verde] = 0$.
2. Luego, recorremos la lista de pares contando la frecuencia de rojos, verdes y azules. Dado un par p, realizaremos la operación $C[p.color] = C[p.color] + 1$
3. A continuación, realizamos una suma de prefijos o frecuencia acumulada de los valores del arreglo C. Ahora C contendrá las posiciones que nos permitirán escribir los pares en su posición final.
4. Finalmente, recorremos la lista de pares L de derecha a izquierda. Dado el i-ésimo par p, realizamos las siguientes operaciones, $M[C[p.color] - 1] = L[i - 1]$ y $C[p.color] = C[p.color] - 1$.

Dado que la lista L se recorre 2 veces, la lista M una vez y el arreglo C es de tamaño constante, el algoritmo tiene complejidad $O(n)$, donde n es el largo de las listas L y M.

23. Revisar los problems del libro guía, *Introduction to Algorithms* (Thomas H. Cormen), Capítulos 2, 3 y 4.