Examen 1: Analisis y Diseño de Algoritmos

Andrey Arguedas Espinoza - 2020426569

October 6, 2022

1. (10 ptos.) Conteste las siguientes preguntas.

    a) Pruebe que para cualquier valor real a>0, $a^n \in O(n!)$.

    b) Pruebe que $(n^5 + 3n^4 + 2n^3 + 5n^2 + n + 1) \in O(n^5)$.

Para demostrarlo realizamos:

$$fn \leq c * g(n) \tag{1}$$

$$n^5 + 3n^4 + 2n^3 + 5n^2 + n + 1 \leq c * n^5 \tag{2}$$

$$n^5 + 3n^5 + 2n^5 + 5n^5 + n^5 + n^5 \leq c * n^5 \tag{3}$$

$$13n^5 \leq c * n^5 \tag{4}$$

Con $\boxed{n = 1}$ $\boxed{c = 13}$

$$13 \leq 13 \tag{5}$$

Por lo tanto queda demostrado que la función pertenece a:

$$O(n^5) \tag{6}$$

2. (10 ptos.) Para el siguiente programa encuentre una función f(n) tal que el programa sea Θ(f(n))

```
begin
input (n)
      for  i:=1 to n do
      ejecuta 5^i pasos
      i := n
      while i >= n/2 do
      begin
      ejecuta i pasos
      i = i -  1
      end
end
```

Para encontrar la función vamos a primero analizar el algoritmo en 2 partes:

- Primera parte:

```
begin
input (n)
      for  i:=1 to n do
      ejecuta 5^i pasos
      i := n
      while i >= n/2 do
      begin
      ejecuta i pasos
      i = i -  1
      end
end
```

En la primera parte vemos que hay un for que ejecuta 5^i pasos en cada iteración desde 1 hasta n, matematicamente esto lo podemos representar como una sumatoria y aplicar la regla de la serie geometrica:

$$\begin{aligned} \sum_{i=1}^{n} 5^i &= \frac{1-5^{n+1}}{1-5} \\ &= \frac{5}{4} * 5^n - 1 \end{aligned} \tag{7}$$

Esta parte del algoritmo la definimos como :

$$\theta(5^n) \tag{8}$$

- Segunda parte:

```
begin
 input (n)
      for  i:=1 to n do
      ejecuta 5^i pasos
      i := n
      while i >= n/2 do
      begin
       ejecuta i pasos
       i = i -  1
      end
end
```

$$\sum_{i=1}^{n/2} n - (i - 1) \; = \; n - [\frac{n-1(n-1+1)}{2}] \tag{9}$$

Finalmente la segunda parte del algoritmo esta en el orden de:

$$\theta(n/2) \tag{10}$$

Finalmente podemos ver que la parte que domina es la primera parte del algoritmo con:

$$\theta(5^n) \tag{11}$$

3. (10 ptos.) Resolver la siguiente ecuación recurrente en forma total:

$$t(n) = \begin{cases} n+1 & n = 0, 1 \\ 3 \cdot t(n-1) - 2 \cdot t(n-2) + 3 \cdot 2^n & \text{si } n > 1 \end{cases}$$

Primero generamos la ecuación característica:

$$t_n - 3t_{n-1} + 2t_{n-2} = 3 * 2^n \tag{12}$$

Del lado izquierdo de la ecuación obtenemos las siguientes raíces:

$$x^2 - 3x + 2 = 0 => (x-2)(x-1) \tag{13}$$

Del lado derecho de la ecuación tenemos que:

$$3 * 2^n \tag{14}$$

$\boxed{b = 2}$ $\boxed{d = 0}$

Por lo que tenemos las siguientes raíces finales: $\boxed{(x-2)^2}$ $\boxed{(x-1)}$

Con esto creamos la nueva ecuacion:

$$t_n = c_1 1^n + c_2 2^n + c_3 n 2^n \tag{15}$$

- Ahora armamos el sistema de ecauciones de 3x3:

$$\begin{aligned} c_1 + c_2 + 0 &= 1 | \text{ n} = 0 \\ c_1 + 2c_2 + 2c_3 &= 2 | \text{ n} = 1 \\ c_1 + 4c_2 + 8c_3 &= 16 | \text{ n} = 2 \end{aligned} \tag{16}$$

Las soluciones del sistema son: $\boxed{c_1 = 12}$ $\boxed{c_2 = -11}$ $\boxed{c_3 = 6}$

$$tn = 12 * 1^n + -11 * 2^n + 6 * n2^n \tag{17}$$

- Finalmente tenemos que $\boxed{n2^n}$ es la parte que domina por lo tanto

$$T(n) \epsilon \theta(n2^n) \tag{18}$$

4. (10 ptos.) Caminando por la playa se encontró un cofre con un tesoro. Dentro del cofre hay n tesoros con pesos $w_1, \ldots, w_n$ y con valores respectivos $v_1, \ldots, v_n$. Desafortunadamente usted solo dispone de una mochila que permite acarrear un peso máximo M. Afortunadamente, usted lleva una mochila que le permite cortar los tesoros si fuera necesario; un tesoro cortado retiene su valor fraccionario (esto es, por ejemplo, tercio del tesoro i pesa w_i / 3 y vale v_i / 3). Usted desea maximizar el valor de los tesoros que pueda acarrear.

   a) Describa un algoritmo voraz con tiempo Θ(n log n) para resolver este problema.

   b) Justificar por qué su algoritmo trabaja correctamente.

El algoritmo que implementaremos consiste en los siguientes pasos:

1- Para cada objeto calcularemos la relación valor/peso Pi/Wi.

2- Ordenamos los objetos descendentemente por el valor Pi/Wi.

3- De mayor a menor valor vamos recorriendo todos los objetos, si lo que pesan es menor al peso actual de la mochila lo insertamos entero, sino solamente la fracción que es posible ingresar.

El pseudocodigo que proponemos es:

```
function knapsack(objects[1...n], W):
    for i= to n do:
        objects[i].pw <— objects[i].profit / objects[i].weight

    objects <— sort(objects) #Suponemos ordenamiento descendente

    weight <- 0
    j <— 0
    result = [1...n]
    while weight < W:
        objects[j]
        if weight + w[j] <= W:
            result[j] <— 1
            weight <— weight + w[j]
        else:
            result[j] <— (W - weight) / w[j]
            weight <— W
    return result
```

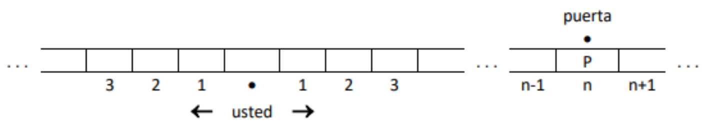
El algoritmo que proponemos es $O(n * logn)$ donde la parte que domina es esl **sort** ya que tanto el calculo de Pi/Wi como el recorrer los elementos solo toma a lo máximo **n** interaciones. Por lo tanto el problema de la mochila es:

$$O(n * logn) \tag{19}$$

El algoritmo trabaja correctamente por que permite las fracciones y siempre que se priorize el valor de valor/peso la respuesta será la correcta.

5. (30 ptos.) Se tiene la siguiente situación:

Usted se encuentra al frente de un muro muy alto que continua hacia el infinito en ambas direcciones, izquierda y derecha. Hay una puerta en el muro pero usted no sabe en qué dirección se encuentra ni a qué distancia está. Está muy oscuro pero usted tiene una linterna muy débil que le permitiría ver la puerta si pasa justo al frente de ella



Dar un algoritmo que le permita encontrar la puerta caminando a lo sumo O(n) pasos, donde n es el número de pasos que usted habría caminado si hubiera sabido en qué dirección se encuentra la puerta. ¿Cuál es la constante multiplicativa de su algoritmo?

Sugerencia: $\sum_{i=0}^{m} 2^i = 2^{m+1} - 1 = 2n - 1$, si $m=\log_2 n$.

Aún si su algoritmo no es lineal, recibirá puntos si encuentra la puerta.

Debido a que la estrucutra sobre la que vamos a trabajar debemos suponer que es infinita, necesitamos una idea que nos permita recorrer tanto hacia la izquierda como a la derecha en tractos. Por lo que haremos una cantidad de 2^i pasos en cada iteración tanto hacia la izquierda como hacia la derecha. En terminos generales la idea sería la siguiente:

- Inicialize i = 0

- Mientras no se haya encontrado la puerta realice los siguientes pasos en orden:

- Inicialize una variable steps = 2^i

- Avanzar 2^i pasos hacia la derecha, si encuentra la puerta detengase, sino:

- Avanzar 2^i pasos hacia la izquierda para devolverse al punto de inicio

- Realice los 2 pasos anteriores ahora en la direccion opuesta (izquierda)

- Incremente i en una unidad y vuelva a empezar el ciclo

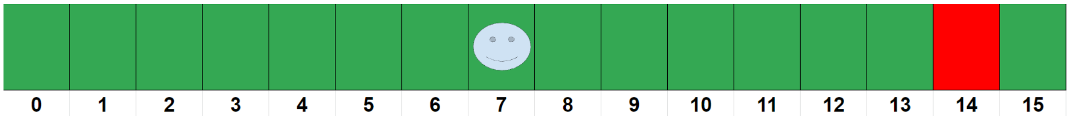
El pseudocodigo de nuestro algoritmo es el siguiente:

```
function findGate(wall, init_pos):
    i <— 0
    direction <— 1 # 1: right, -1:left
    directionsChecked <— 0
    steps <— 2 ** i
    last_pos <— init_pos

    if wall[init_pos] == 1: #Found the door
        return init_pos


    while True: #Door not found yet

        #Go to direction forward
        for idxForward 1...steps:
            last_pos <— init_pos + (direction * idxForward)
            if wall[last_pos] == 1: # Door found
                return last_pos

        #Go to direction backwards, to return to initial position
        for idxBack 1...steps:
            if wall[last_pos + (-direction * idxBack)] == 1:
                return last_pos + (-direction * idxBack) # Door found

        directionsChecked += 1 #Checks a direction visited
        direction *= -1 #Changes the direction for next iteration

        #If right and left have been checked we increment the steps and start
        if directionsChecked == 2:
            i += 1
            steps <— 2 ** i
            directionsChecked <— 0
```

Implementación en Python :

```python
def find_door(wall, initial_pos):
    i = 0
    direction = 1 # 1: right, -1:left
    directionsChecked = 0
    steps = 2 ** i
    last_pos = initial_pos

    if wall[initial_pos] == 1: #Found the door
        return initial_pos


    while True: #Door not found yet

        #Go to direction forward
        for idxR in range(1, steps):
            last_pos = initial_pos + (direction * idxR)
            if wall[last_pos] == 1:
                return last_pos # Door found at right side

        #Go to direction backwards, to return to initial position
        for idxBack in range(1, steps):
            if wall[last_pos + (-direction * idxBack)] == 1:
                return last_pos + (-direction * idxBack)

        directionsChecked += 1
        direction *= -1

        #If right and left have been checked we increment the steps and start again
        if directionsChecked == 2:
            i += 1
            steps = 2 ** i
            directionsChecked = 0
```

Por ejemplo para el siguiente caso nuestro algoritmo se mueve de la siguiente manera:




```
In [30]: runfile('C:/Users/Andrey/TestFindGate.py', wdir='C:/Users/Andrey')
Forward Right 8
Back 7
Forward LEFT 6
Back 7
Forward Right 8
Forward Right 9
Forward Right 10
Back 9
Back 8
Back 7
Forward LEFT 6
Forward LEFT 5
Forward LEFT 4
Back 5
Back 6
Back 7
Forward Right 8
Forward Right 9
Forward Right 10
Forward Right 11
Forward Right 12
Forward Right 13
Forward Right 14
Door found on position: 14
```

Podemos ver los movimientos que realiza y como encuentra la puerta en la posición numero 13 (simplemente como prueba se hizo con un array finito, el algoritmo tambien funcionaría para uno infinito). La constante multiplicativa de este algoritmo sería de **4** porque se estaría caminado 2 veces los pasos por cada dirección.

6. (10 ptos.) Para ilustrar la forma en que se puede utilizar una notación asintótica para ordenar la eficiencia de los algoritmos, utilícense las relaciones "$\subset$" (subconjunto estricto) y "=" para poner los órdenes de las siguentes funciones en una secuencia ascendente. Justifique brevemente sus respuestas.

2^{2n}, $n^{\log n}$, 3^n, 2^{n+1}, n^n, 2^{n+100}

Sugerencia: convertir todas las expresiones a una forma 2^x.

Primero debemos convertir todas las expresiones a una forma de 2^x, sin embargo ya algunas estan en esa forma como lo son: 2^{2n}, 2^{n+1}, 2^{n+100}.

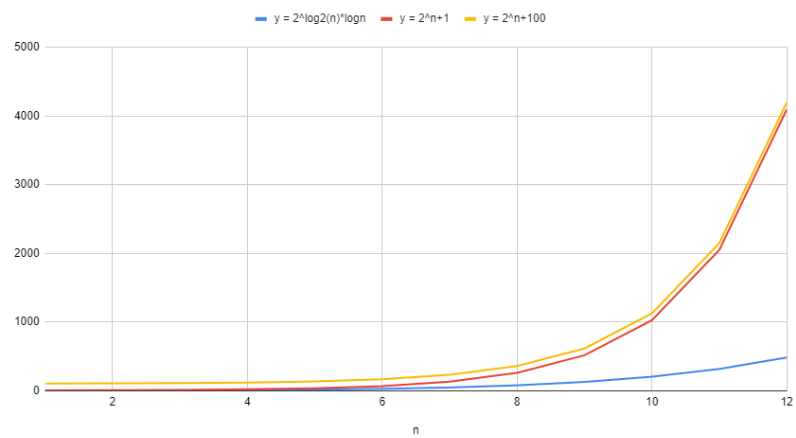
Por lo que se deben convertir las siguientes expresiones n^{logn}, 3^n, n^n, utilizando la propiedad de los logaritmos $a^{loga(b)} = b$.

$$n^{logn} => 2^{log2(n)*log(n)} \tag{20}$$

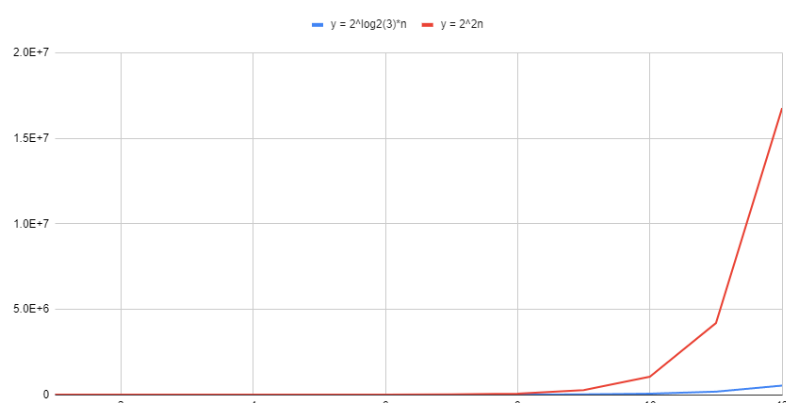
$$n^n => 2^{log2(n)*n} \tag{21}$$

$$3^n => 2^{log2(3)*n} \tag{22}$$

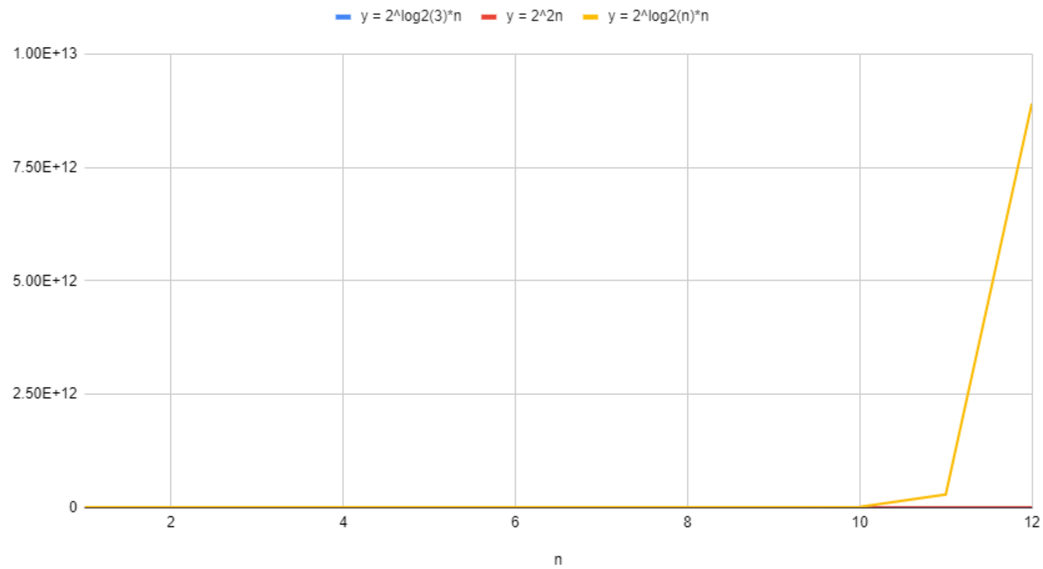
Ya ahora que todos tienen la misma base podemos concentrarnos en los exponentes, además podemos graficar para ver de forma mas visual las acotaciones, en orden ascendente prmero tendriamos a las 3 siguientes funciones:



Seguidamente las mayores siguientes:



Y por ultimo la de mayor orden:



Finalmente las expresiones originales quedan ordenadas ascendentemente de la siguiente manera:

$$2^{log2(n)*log(n)} \subset 2^{n+1} = 2^{n+100} = 2^{log2(3)*n} = 2^{2n} \subset 2^{log2(n)*n} \tag{23}$$

7. (10 ptos.) Resolver exactamente la siguiente recurrencia:

$$t(n) = \begin{cases} 2^n - 1 & n = 0 \\ t(n-1) + n \cdot 3^{n-1} & n \geq 1 \end{cases}$$

Primero generamos la ecuación característica:

$$t_n - t_{n-1} = n * 3^{n-1} \tag{24}$$

Del lado izquierdo de la ecuación obtenemos las siguientes raices:

$$x - 1 = 0 => (x - 1) \tag{25}$$

Del lado derecho de la ecuación tenemos que:

$$n * 3^{n-1} \tag{26}$$

$\boxed{b = 3}$ $\boxed{d = 1}$

Por lo que tenemos las siguientes raices finales: $\boxed{(x-3)^2}$ $\boxed{(x-1)}$

Con esto creamos la nueva ecuacion:

$$t_n = c_1 1^n + c_2 3^n + c_3 n 3^n \tag{27}$$

- Ahora armamos el sistema de ecauciones de 3x3:

$$\begin{aligned} c_1 + c_2 + 0 &= 0 | \text{ n = 0} \\ c_1 + 3c_2 + 3c_3 &= 1 | \text{ n = 1} \\ c_1 + 9c_2 + 18c_3 &= 7 | \text{ n = 2} \end{aligned} \tag{28}$$

Las soluciones del sistema son: $\boxed{c_1 = 1/4}$ $\boxed{c_2 = -1/4}$ $\boxed{c_3 = 1/2}$

$$tn = 1/4 * 1^n + -1/4 * 3^n + 1/2 * n3^n \tag{29}$$

- Finalmente tenemos que $\boxed{n3^n}$ es la parte que domina por lo tanto

$$T(n) \epsilon \theta(n3^n) \tag{30}$$

8. (10 ptos.) Suponga que debe escoger uno de los siguientes tres algoritmos:

i. Algoritmo A resuelve un problema dividiéndolo en cinco subproblemas con la mitad del tamaño del problema original, resolviendo recursivamente cada subproblema y finalmente combinando las soluciones en tiempo lineal.
ii. Algoritmo B resuelve un problema dividiéndolo en cuatro subproblemas con la mitad del tamaño del problema original, resolviendo recursivamente cada subproblema y combinando las soluciones en tiempo cuadrático.
iii. Algoritmo C resuelve un problema de tamaño n dividiéndolo en siete subproblemas con una tercera parte del tamaño del problema original, resolviendo recursivamente cada subproblema y combinando las soluciones en tiempo cuadrático.

¿Cuáles son los tiempos asintóticos de estos algoritmos y cual escogería y por qué?

Recuerde el resultado:

Se tienen enteros $n_0 \geq 1$, $\ell \geq 1$, $b \geq 2$, $k \geq 0$,

se tiene real $c > 0$,

se tiene $T: N \rightarrow R$, asintóticamente no decreciente, tal que

$T(n) = \ell \cdot T(n/b) + c \cdot n^k$, $n \geq n_0$.

Entonces,

$$T(n) = \begin{cases} \Theta(n^k) & \ell < b^k \\ \Theta(n^k \log n) & \ell = b^k \\ \Theta(n^{\log_b \ell}) & \ell > b^k \end{cases}$$

Primero con la descripción de los Algoritmos vamos a convertirlos a expresiones T(n):

Algoritmo A : $5T(n/2) + n$

Algoritmo B : $4T(n/2) + n^2$

Algoritmo C : $7T(n/3) + n^2$

Mediante la utilización del Teorema Maestro tenemos que:

Algoritmo A : $5T(n/2) + n \Rightarrow 5 > 2^1 \Rightarrow \theta(n^{log2(5)})$

Algoritmo B : $4T(n/2) + n^2 \Rightarrow 4 = 2^2 \Rightarrow \theta(n^2 logn)$

Algoritmo C : $7T(n/3) + n^2 \Rightarrow 7 < 3^2 \Rightarrow \theta(n^2)$

Finalmente se escogería el algoritmo C ya que su tiempo asintotico n^2 es el menor de los 3.