Secuencia de paréntesis desbalanceada

Secuencia de paréntesis desbalanceada

Dada una cadena que contiene *N* paréntesis abiertos y *N* cerrados ¿cual es la mínima cantidad de intercambios adyacentes para equilibrarla?

Ejemplo: ( )) ( ) (

1: Intercambiar 3 con 4: ( ) ( )) (

2: Intercambiar 5 con 6: ( ) ( ) ( )

Secuencia de paréntesis desbalanceada

¿Cuál sería el criterio *greedy* en este caso?

Al encontrar un desbalance en un índice *i*, corregirlo “trayendo” hasta ese índice el paréntesis abierto más cercano a la derecha

Y, ¿cómo sería el proceso iterativo?

Repetir el procedimiento a partir del siguiente índice

```
read S
M = |S|
swaps, balance = 0, 0
for i = 0 to M:
   if S_i = '(':
       balance += 1
   else:
       balance -= 1
       if balance < 0:
          j = i + 1
          while j < M and S_j ≠ '(':
             j = j + 1
          swaps += j - i
          balance = 1
          par = S_j
          for k = j to i, step -1:
             S_k = S_k-1
          S_i = par
print swaps
```

¿Cuál es la eficiencia de este algoritmo?

$\boldsymbol{O(N^2)}$

¿Se puede hacer mejor, más simple?

Si, pre-calculando los índices donde se encuentran los paréntesis abiertos

```
read S
M = |S|
pos_open = #arreglo con índices de los ( en S
swaps, p, balance = 0, 0, 0
for i = 0 to M:
   if S_i = '(':
       balance += 1
       p += 1
   else:
       balance -= 1
       if balance < 0:
          swaps += pos_open[p] - i
          S_i, Spos_open_p = Spos_open_p, S_i
          p += 1
          balance = 1
print swaps
```

¿Cuál es la eficiencia de este algoritmo? $\boldsymbol{O(N)}$

Programación de procesos unitarios

Programación de procesos unitarios

Conocido como *Unit Time Scheduling with Deadlines*

Entrada: Un conjunto A = {$a_1, a_2, \ldots, a_N$} de procesos unitarios , es decir, que demoran una (1) unidad de tiempo. Cada proceso a_i tiene un identificador, un tiempo límite de finalización e_i ($1 \leq e_i \leq N$) y una penalización no negativa w_i en que se incurre al no terminar el proceso a_i en el tiempo e_i

Salida: Secuencia de procesos que minimiza la penalización total considerando que solo se puede hacer un proceso al tiempo

Programación de procesos unitarios

Ejemplo:

a_i	A	B	C	D
e_i	3	2	3	1
w_i	25	10	20	15

Soluciones: {D, A, C, B} ó {D, C, A, B}, ambas con una penalización total de 10

En general para *N* procesos, ¿Cuál es la cantidad de posibles soluciones para este problema? $N!$

Programación de procesos unitarios

¿Se podrá resolver mediante una aproximación greedy? ¿Cuál sería el criterio a utilizar?

¿Ideas?

Idea 1: Ordenar los procesos descendentemente por la penalización y ejecutarlos en ese orden

En el ejemplo anterior daría: {A, C, D, B} con una penalización total de 25

Idea 2: Ordenar los procesos descendentemente por la penalización, luego ubicarlos uno por uno según ese orden pero en la posición donde dejen “el mayor margen posible” para los restantes. En caso de empate elegir primero el de menor tiempo límite de finalización

```
read A //N valores con atributos id, end, w
order(A) //Descendentemente por w, ascendentemente por end
for i = 0 to N-1:
     S_i = NULL
q = N-1
for i = 0 to N-1:
     p = A_i.end
     while S_p ≠ NULL:
          p = p-1
     if p ≥ 0:
          S_p = A_i.id
     else:
          while S_q ≠ NULL:
               q = q-1
          S_q = A_i.id
          q = q-1
print S
```

Ejemplo:

id	A	B	C	D	E	F	G
e	4	2	4	3	1	4	6
w	70	60	50	40	30	20	10

S	D	B	C	A	G	F	E

Iteración 1: Tomamos A, con tiempo límite 4 y lo ubicamos en la posición 4

Iteración 2: Tomamos B, con tiempo límite 2 y lo ubicamos en la posición 2

Iteración 3: Tomamos C, con tiempo límite 4 y lo ubicamos en la posición 3

Iteración 4: Tomamos D, con tiempo límite 3 y lo ubicamos en la posición 1

Iteración 5: Tomamos E, con tiempo límite 1 y lo ubicamos en la posición 7

Iteración 6: Tomamos F, con tiempo límite 4 y lo ubicamos en la posición 6

Iteración 7: Tomamos G, con tiempo límite 6 y lo ubicamos en la posición 5

¿Cuál es la eficiencia de este algoritmo? $O(N^2)$

Código de Huffman, parte 1

Código de Huffman

Es usado como mecanismo de compresión de propósito general, es decir sirve para diferentes tipos de archivos y sistemas de codificación. Un sistema de codificación define el “alfabeto” utilizado para representar un determinado elemento.

Archivo	Codificación
Texto plano	ASCII: 7 bits por caracter
	UTF-8: entre 1 y 4 bytes por caracter
Imagen	Grayscale: 1 byte por pixel
	RGBA: 1 byte por canal + canal alfa para transparencia = 4 bytes por pixel
Audio	PCM (Pulse Code Modulation) estándar: 2 bytes por muestra a una frecuencia de 44.1 kHz

En general, para un alfabeto de *N* caracteres, la forma típica de codificación es un código binario de una longitud de $log_2(N)$ bits.

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Fuente: https://es.m.wikipedia.org/wiki/Archivo:ASCII-Table-wide.svg

Ejemplo: ASCII

@: 0111111
U: 1010101

Código de Huffman

Veamos como ejemplo el alfabeto Hawaiiano considerando sus 5 vocales, 7 consonantes, el espacio en blanco, el punto, la coma, y la okina (apóstrofo):

Carácter	**Código**	**Carácter**	**Código**
A	0000	O	1000
E	0001	P	1001
H	0010	U	1010
I	0011	W	1011
K	0100	(espacio)	1100
L	0101	'	1001
M	0110	,	1110
N	0111	.	1111

Pero, si las frecuencias de aparición de los caracteres no es homogénea, es más eficiente usar códigos binarios de longitud variable.

Código de Huffman

Carácter	Frecuencia	Código	Carácter	Frecuencia	Código
A	21%	00	O	14%	110
E	12%	011	P	1,5%	100111
H	1,5%	111010	U	11%	010
I	13%	101	W	1,5%	100110
K	5,5%	11111	(espacio)	6,5%	1000
L	4,5%	11110	'	1%	1110110
M	2,5%	10010	,	0,5%	11101110
N	3,5%	11100	.	0,5%	11101111

Así, para un archivo que contenga por ejemplo 1000 caracteres y que conserve esas mismas frecuencias, el tamaño (sin metadatos) en codificación de longitud fija sería de 4000 bits, mientras que en longitud variable sería de 3560. Es decir, se obtiene una tasa o radio de compresión de 0.89 (ahorro de 11%).

Código de Huffman, parte 2

Código de Huffman

Pongamos un ejemplo más simple que el del alfabeto Hawaiiano simplificado: las cadenas de ADN, las cuales están compuesto por cuatro bases nitrogenadas representadas por las letras A (adenina), T (timina), C (citosina) y G (guanina).

Las frecuencias típicas en el genoma humano pueden variar ligeramente según el cromosoma, pero en términos generales y de manera aproximada, son de 30% para A y T, y de 20% para C y G.

En este caso un posible código binario de longitud fija sería:
A: 00, T: 01, C: 10 y G: 11, mientras que uno de longitud variable que considere las frecuencias sería:
A: 0, T: 1, C: 01, G: 10

Es decir, la tasa de compresión sería $\frac{1*0.3+1*0.3+2*0.2+2*0,2}{2} = \frac{1.4}{2}$

Código de Huffman

Ese código de longitud variable (A: 0, T: 1, C: 01, G: 10) es eficiente (el radio de compresión es menor a 1), pero ¿es efectivo? En otras palabras ¿es fácil de de-codificar?

Pensemos por ejemplo en la cadena 001, ¿a qué caracteres correspondería? (A: 0, T: 1, C: 01, G: 10)

No se podría decir con certeza, por tanto, ¿Es posible definir un sistema de codificación binaria que sea a la vez eficiente y efectivo?

Código de Huffman

Entrada: un alfabeto φ con N caracteres y un arreglo de frecuencias correspondiente $\{f_1, f_2, \ldots, f_N\}$ (valores numéricos no negativos)

Salida: sistema de codificación binario no ambiguo de longitud variable para ese alfabeto.

¿Ideas para evitar la ambigüedad?

Idea: definir solamente códigos (cadenas de bits) que sean “libres de prefijos”, es decir que para todo par de caracteres i, j ninguna de las codificaciones correspondientes sea un prefijo de la otra.

En el ejemplo anterior los códigos no cumplían esta propiedad puesto que 0 es prefijo de 01, así como 1 es prefijo de 10

Código de Huffman

La pregunta clave es: ¿Cómo garantizar esa propiedad?

Volviendo al ejemplo:

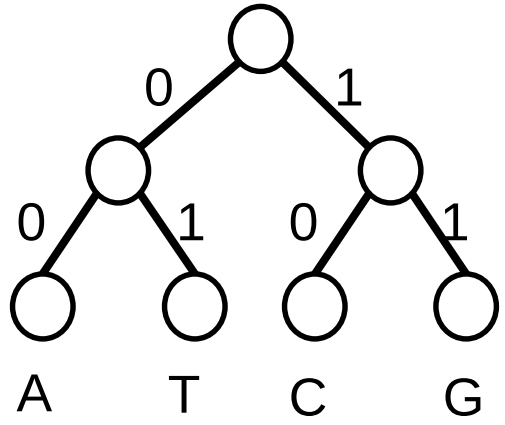
A podría ser 0
T no podría ser 1, pero podría ser 10
C no podría empezar por 0 ni 10, entonces podría ser 110
G podría ser 111

¿Cómo definir un algoritmo general (para cualquier alfabeto) que no solo sea efectivo (libre de prefijos), sino también eficiente (que minimice la cantidad promedio de bits)?

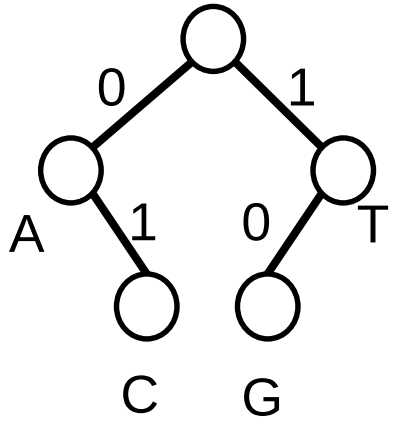
¿Ideas?

Para resolver este problema, Huffman ideó pensar en el sistema de codificación como si fuera un árbol binario.

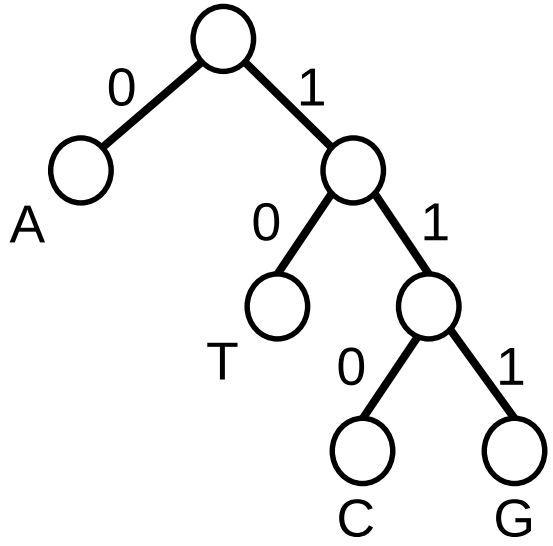
Codificación de longitud fija



Codificación de longitud variable



Otra codificación de longitud variable

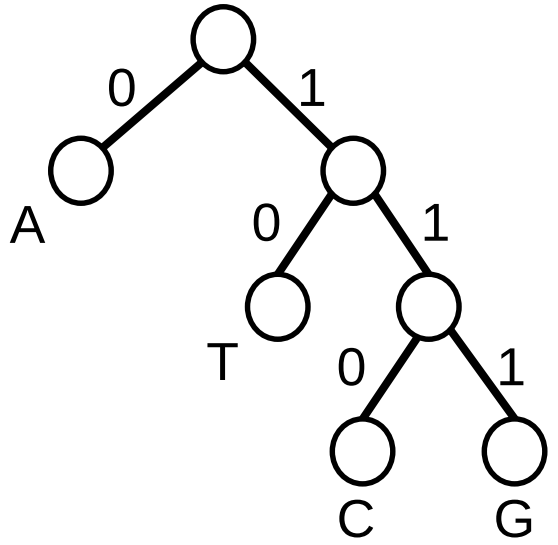


¿En qué se parecen el árbol 1 y 3, los cuales son no ambiguos?

En que en ambos todos los caracteres se encuentran en las hojas del árbol. En otras palabras ningún carácter es ancestro de ningún otro, garantizando así que todos tienen la propiedad de ser “libres de prefijos”

En general, al utilizar esta notación de árbol binario donde los caracteres únicamente están en las hojas, y etiquetando las aristas de los hijos izquierdos con ‘0’ y las de los hijos derechos con ‘1’, tenemos que:

1. Codificar un carácter equivale a acumular las etiquetas de las aristas desde la raíz hasta el carácter;
2. La cantidad de bits necesarios para codificar un carácter es igual al nivel en que se encuentra en el árbol; y
3. Decodificar un código equivale a seguir, repetidamente hasta llegar al final del código, la ruta que va desde la raíz hasta una hoja.



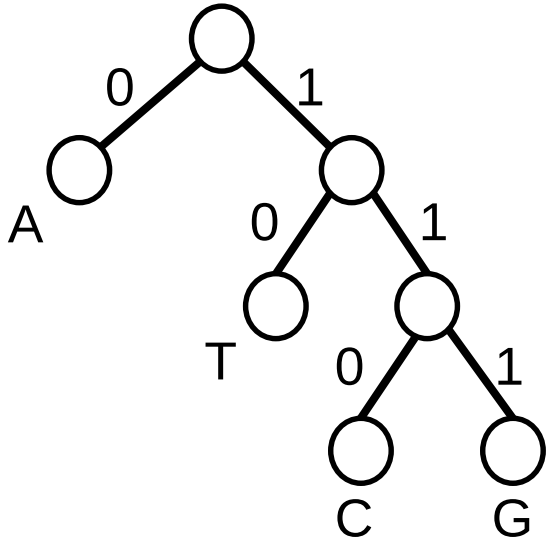
Ejemplo: para decodificar la cadena 100110111 tendríamos:

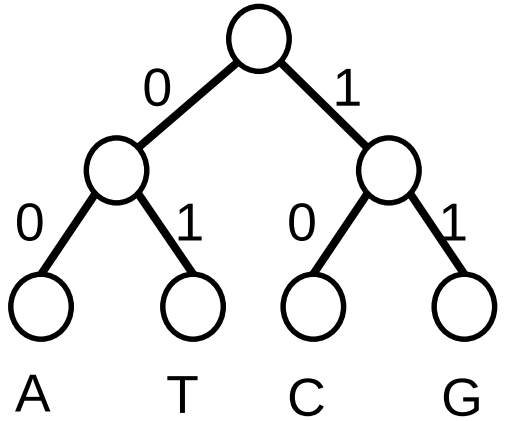
10 = T, 0 = A, 110 = C, 111 = G; es decir, TACG

Código de Huffman

Considerando la notación de árbol, la salida del problema que estamos analizando (sistema de codificación binario no ambiguo de longitud variable para φ) se convierte en:

Minimizar $L(T) = \sum_{i=1}^{N} f_i * (nivel\ de\ i\ en\ T)$ siendo T el árbol binario





L(*T1*) = 0.3*2+0.3*2+0.2*2+0.2*2=**2.0**

L(*T2*) = 0.3*1+0.3*2+0.2*3+0.2*3=**2.1**

Código de Huffman, parte 3

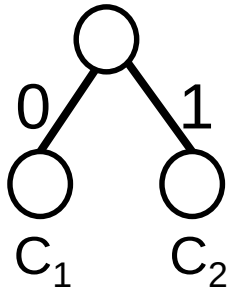
Código de Huffman

¿Cómo construimos el árbol? ¿Se podrá usar un algoritmo voraz?

La idea principal del código de Huffman es realizar combinaciones sucesivas de pares de subárboles, utilizando cada vez un criterio de selección greedy.

¿Qué es combinar? Consideremos dos subárboles que corresponden a caracteres individuales:

Combinar C_1 y C_2:



Código de Huffman

¿Cuál sería entonces el criterio greedy? ¿Cómo seleccionar que par de subarboles es “seguro” combinar en un momento dado?

Considerando que cada que se hace una combinación, los nodos involucrados “heredan” un bit más, los caracteres con mayor frecuencia deberían dejarse para las últimas combinaciones. Por el contrario, los caracteres con menor frecuencia pueden “sacrificarse” en las primeras combinaciones, a sabiendas que quedarán en los últimos niveles del árbol.

Eureka: he allí el criterio greedy que estamos buscando

Solo falta definir un algoritmo que sistemáticamente aplique dicho criterio hasta obtener la solución final

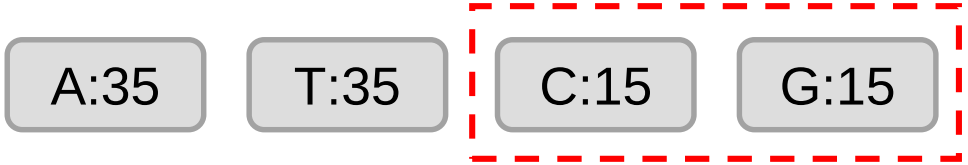
Código de Huffman

```
read A, f //N valores
//Creación de los nodos iniciales donde Q es una cola con prioridad
for i = 0 to N:
     //x es un nodo vacio
     x.char = A_i
     x.freq = f_i
     x.left = NULL
     x.right = NULL
     Q.push(x)
for i = 0 to N-1:
     //x es un nodo vacio
     x.left = Q.pop()
     x.right = Q.pop()
     x.char = x.left.char + x.right.char
     x.freq = x.left.freq + x.right.freq
     Q.push(x)
print Q
```

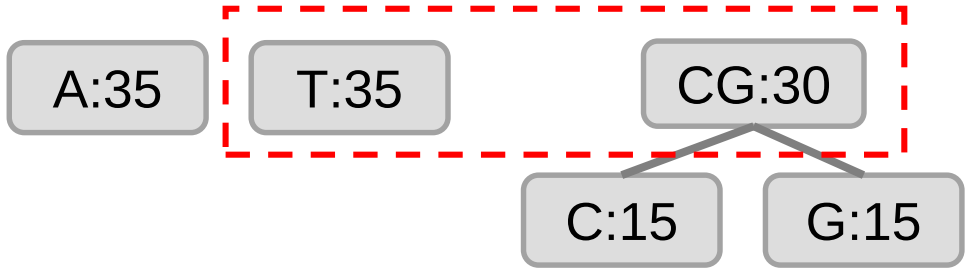
Veamos un ejemplo para el ADN pero con un sesgo AT/GC más pronunciado:

A	T	C	G
35%	35%	15%	15%

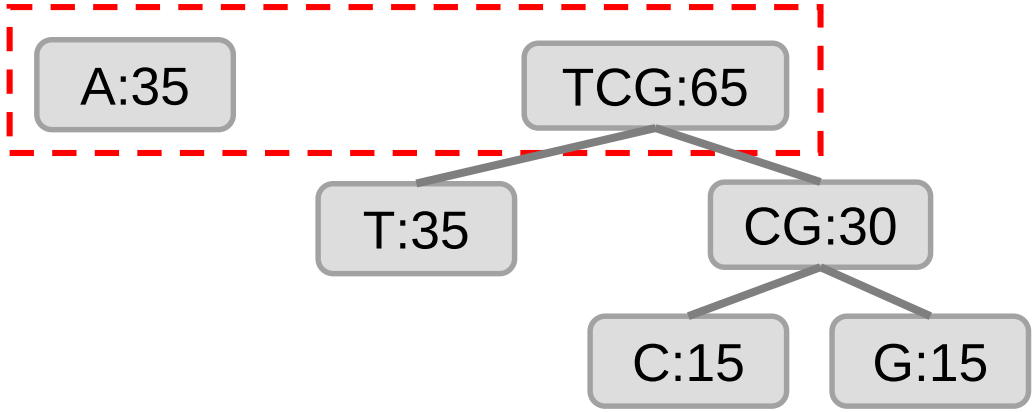
Iteración 0:



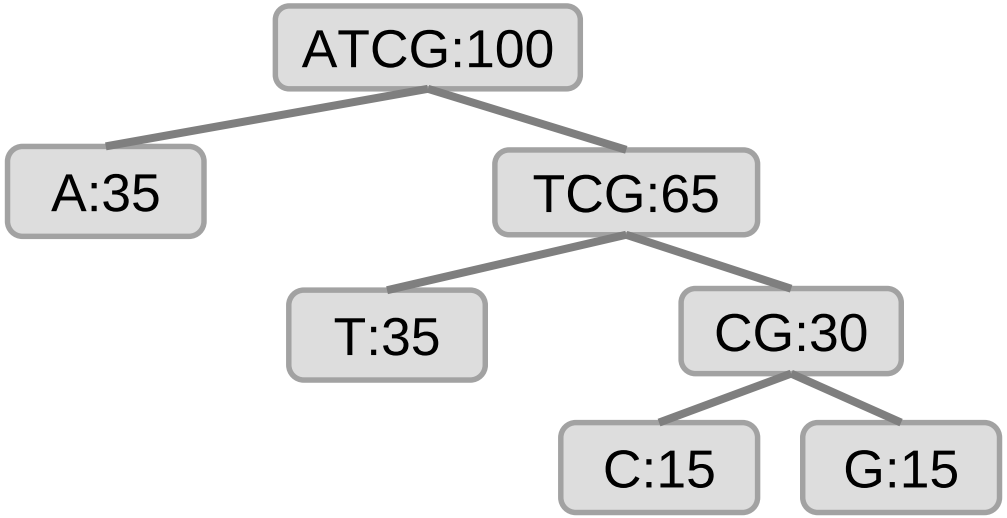
Iteración 1:



Iteración 2:



Iteración 3:



L(T) = ? 0.35*1 + 0.35*2 + 0.15*3 + 0.15*3 = 1.95 vs 2.0 → 2.5%

Código de Huffman

¿Cuál es la complejidad del algoritmo?

```
for i = 0 to N:
     //x es un nodo vacio
     x.char = A_i
     x.freq = f_i
     x.left = NULL
     x.right = NULL
     Q.push(x)
for i = 0 to N-1:
     //x es un nodo vacio
     x.left = Q.pop()
     x.right = Q.pop()
     x.char = x.left.char + x.right.char
     x.freq = x.left.freq + x.right.freq
     Q.push(x)
print Q
```

O(*N*)

O(*1*)

O(log(*N*))

O(*N*)

O(log(*N*))

O(log(*N*))

O(*1*)

O(log(*N*))

$$N * \log(N)) + N * 3\log(N)$$

$$\rightarrow O(N * \log(N))$$

Ejemplo

¿Dado el alfabeto A-G, cuál sería un código de Huffman considerando el siguiente texto?

C C D C F E
C C D C G F
C C C A F E D
A A A F G F

Pista 1: La tasa de compresión es $\frac{63}{75}$

Pista 2:

C	D	E	F	G	A	B
Do	Re	Mi	Fa	Sol	La	Si