PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC1001 — Algoritmos y Sistemas Computacionales — 2023-1

Interrogación 2 - Soluciones

Lunes 5-Junio-2023

Duración: 120 minutos
Responder cada pregunta en una hoja separada.

1. **[12p]** A partir del siguiente conjunto de *strings*:
   $\{\texttt{Jorge}, \texttt{Valeria}, \texttt{ignacio}, \texttt{1203}, \texttt{Catalina}, \texttt{cristian}, \texttt{Francisca}, \texttt{kamilo}\}$

   1.1) **[4p]** Construya una tabla de hash donde la función de hash sea la longitud de cada string, y resolviendo colisiones mediante una lista ligada.

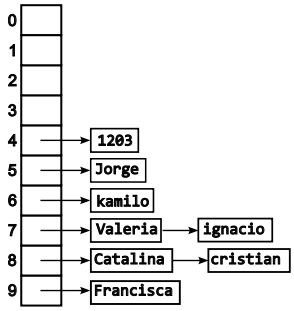
   R. Cada *string* se debe almacenar en un *slot* de acuerdo a su longitud. El *string* más largo tiene 9 caracteres, por lo tanto necesitamos una tabla que contenga al menos la posición 9. Construimos una tabla con 10 *slots*. Para cada *string* se calcula su longitud y se obtiene el siguiente *hash* para cada uno:

   $\{\texttt{Jorge} \to 5, \texttt{Valeria} \to 7, \texttt{ignacio} \to 7, \texttt{1203} \to 4, \texttt{Catalina} \to 8, \texttt{cristian} \to 8, \texttt{Francisca} \to 9, \texttt{kamilo} \to 6\}$

   Los *strings* que coinciden en su *hash* se almacenan en el mismo *slot*, pero en forma de lista ligada. No es importante el orden en que se ingresan a cada lista ligada. La tabla queda construida de la siguiente manera:

   $\{0 \to \{\}, 1 \to \{\}, 2 \to \{\}, 3 \to \{\}, 4 \to \{\texttt{1203}\}, 5 \to \{\texttt{Jorge}\}, 6 \to \{\texttt{kamilo}\}, 7 \to \{\texttt{Valeria}, \texttt{ignacio}\}, 8 \to \{\texttt{Catalina}, \texttt{cristian}\}, 9 \to \{\texttt{Francisca}\}\}$

   De manera gráfica se puede representar así:



   Puntaje (0, 0.5, 1):

   +1 pto: la tabla debe contener al menos 10 posiciones. No puede ir desde 4 hasta 9. Debe empezar en 0.

   +1 pto: calcular *hash* correcto para cada *string* (longitud)

   +1 pto: ubicar cada *string* en su posición correcta

   +1 pto: resolver cada *string* colisión con una lista ligada. No importa el orden dentro de la lista.

1.2) **[4p]** Suponga que tiene un conjunto de N *strings*, donde N puede ser muy grande. Ningún *string* es de largo mayor a K. ¿Cuál es la complejidad, en el peor caso, de buscar un elemento en esta tabla?

R. La existencia de un valor máximo K para el largo de un *string* nos indica que necesitamos $K + 1$ *slots* (una cantidad acotada). Dicho eso, la peor situación es que todos los *strings* sean del mismo largo L y, por lo tanto, todos colisionan en el mismo *slot* L. La búsqueda en este caso se transforma en una búsqueda lineal dentro de una lista de N elementos. La complejidad es $O(N)$.

Es incorrecto decir que es $O(KN)$ o $O((K + 1)N)$ si la argumentación es que hay que buscar en todos los $K + 1$ *slots*, pues al usar una tabla de *hash* la búsqueda se hace solo en 1 *slot*. Por otro lado, se podría argumentar que la comparación de un *string* de largo L en una lista de largo N puede significar comparar $L \times N$ caracteres (puntaje parcial), sin embargo en una búsqueda determinada, L es una constante por lo tanto la complejidad sigue siendo $O(N)$.

Puntaje:

4 pts: complejidad correcta. No es necesario justificar.

2 pts: complejidad incorrecta, pero puede haber una constante junto a N y una justificación razonable de por qué usar esa constante.

0 pto: complejidad incorrecta sin ninguna justificación, o justificación incorrecta.

1.3) **[4p]** Proponga una función de hash para que NO se provoquen colisiones con el caso particular del conjunto de *strings* propuesto (el conjunto de ejemplo).

R. Se pueden proponer diversas funciones, sin embargo una propuesta sencilla para este ejemplo particular puede ser utilizar como *hash* el valor numérico de la primera letra de cada *string*, aprovechando el hecho que cada uno empieza con una letra distinta (recordar que `C` y `c` son letras distintas). De esta manera el valor de *hash* de cada uno es:

$\{\texttt{Jorge} \to \texttt{J}, \texttt{Valeria} \to \texttt{V}, \texttt{ignacio} \to \texttt{i}, \texttt{1203} \to \texttt{1}, \texttt{Catalina} \to \texttt{C}, \texttt{cristian} \to \texttt{c}, \texttt{Francisca} \to \texttt{F}, \texttt{kamilo} \to \texttt{k}\}$

También podría escribirse usando el valor decimal (o bien el hexadecimal) del código ASCII como:

$\{\texttt{Jorge} \to 74, \texttt{Valeria} \to 86, \texttt{ignacio} \to 105, \texttt{1203} \to 49, \texttt{Catalina} \to 67, \texttt{cristian} \to 99, \texttt{Francisca} \to 70, \texttt{kamilo} \to 107\}$

Puntaje:

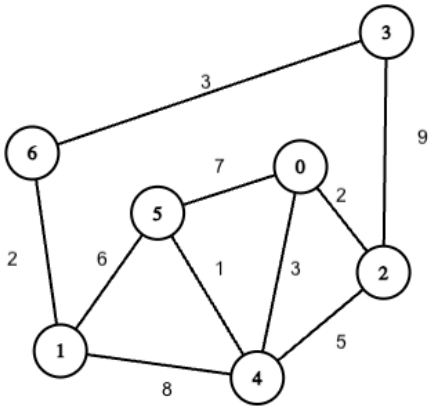
4 pts: describir una función de *hash* que no provoque colisiones. Debe mostrar o dar una explicación correcta, o justificar de alguna manera por qué no provoca colisiones con ese caso (por ejemplo, decir que todos empiezan con letras distintas).

3 pts: función de *hash* correcta, pero no hay una explicación o demostración de que no provoca colisiones, y la explicación no es obvia.

1 pto: función que provoca colisiones con 1 caso

0 pto: función incorrecta. Provoca colisiones con 2 o más casos.

2. **[12p]** A partir del siguiente grafo no dirigido, $G = (V, E)$:



2.1) **[4p]** Obtener un recorrido BFS a partir del nodo 0

R. Existen diversos recorridos BFS posibles dependiendo del orden en que se agregan los nodos que están a una misma distancia del nodo 0. En la siguiente respuesta se muestran entre paréntesis cuadrados ([]) aquellos que son intercambiables entre sí (pero eso no se pedía):

$\{0, [5, 4, 2], [1, 3], 6\}$

Puntaje:

4 pts: un recorrido BFS correcto

3 pts: hay un error menor (un nodo a distancia incorrecta), pero se puede identificar el origen del error a partir de los cálculos

2 pts: un nodo en posición incorrecta, pero sin justificación

0 pto: más de un nodo en posición incorrecta

-1 pto: descuento si no incluye el nodo 0 como primer nodo del recorrido

2.2) **[4p]** Utilice el algoritmo de Dijkstra para obtener las rutas más cortas desde 0 hasta los demás. Debe mostrar el camino desde 0 hacia cada uno de los otros nodos, y el costo de la ruta.

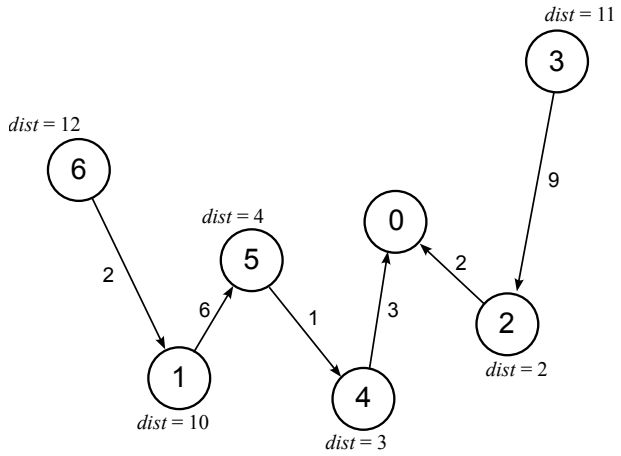
R. La tabla de distancias de cada nodo a partir de 0, y la tabla de predecesores correspondiente quedan de esta manera:

v	$dist[v]$
0	0
1	10
2	2
3	11
4	3
5	4
6	12

v	$pred[v]$
0	–
1	5
2	0
3	2
4	0
5	4
6	1

La tabla de distancias y la tabla de predecesores (juntas) son suficientes para obtener el costo de las ruta más cortas, y el recorrido desde 0 hasta cada nodo.

A partir de ellas se puede ilustrar el sub-grafo de rutas más cortas de la siguiente manera:



Puntaje:

4 pts: Tabla de distancias y de predecesores correcta y completa. Alternativamente puede dibujarse el grafo resultante, que debe indicar las flechas de las rutas y el costo de cada ruta. Alternativamente puede explicitarse la ruta para cada nodo. Ejemplo: Nodo $3 = \{0 \to 2 \to 3\}$, $dist = 11$. No es necesario detallar el procedimiento.

3 pts: Están ambas tablas presentes (o su información), pero hay un error menor en alguna de ellas

2 pts: Una tabla completa y correcta (o su información), y la otra no está presente. Alternativamente, están ambas tablas pero hay errores que pueden entenderse a partir del procedimiento (si lo hay).

1 pto: Solo una tabla con errores importantes.

0 pto: tablas incorrectas.

2.3) **[4p]** Ejecute el siguiente algoritmo sobre el grafo G para construir un nuevo grafo $G' = (V', E')$: (1) tomar el mismo conjunto de nodos $V' = V$; (2) Agregar a E' la arista de E con **menor** peso, y que **no forme un ciclo en** G'. El algoritmo termina cuando no hay más aristas que agregar. Dibuje el resultado e indique su complejidad.

R. El conjunto de nodos G' es el mismo del grafo original: $G' = \{0, 1, 2, 3, 4, 5, 6\}$. Para construir el conjunto de aristas se deben agregar iterativamente aquellas que tienen menor peso, pero cuidando de no formar un ciclo. Las aristas de E, ordenadas por su peso (10 aristas) son:

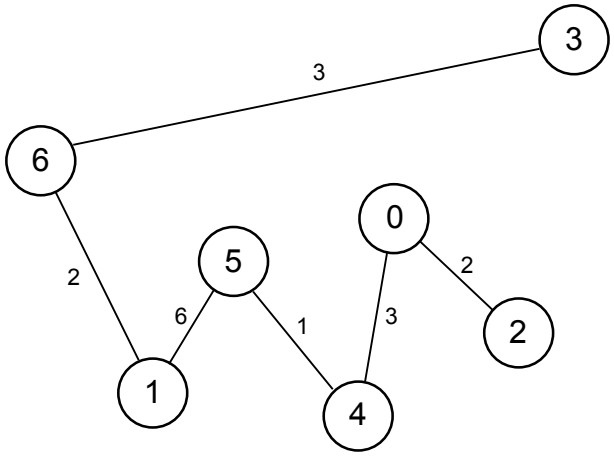
$E = \{(4, 5), (6, 1), (0, 2), (6, 3), (4, 0), (2, 4), (1, 5), (5, 0), (1, 4), (2, 3)\}$

El proceso es el siguiente:

- Agregar arista $(4, 5)$: $E' = \{(4, 5)\}$
- Agregar arista $(6, 1)$: $E' = \{(4, 5), (6, 1)\}$ (podría haber sido $(0, 2)$ primero)
- Agregar arista $(0, 2)$: $E' = \{(4, 5), (6, 1), (0, 2)\}$
- Agregar arista $(6, 3)$: $E' = \{(4, 5), (6, 1), (0, 2), (6, 3)\}$ (podría haber sido $(4, 0)$ primero)
- Agregar arista $(4, 0)$: $E' = \{(4, 5), (6, 1), (0, 2), (6, 3), (4, 0)\}$
- No agregar arista $(2, 4)$, pues formaría un ciclo $0 \to 2 \to 4 \to 0$
- Agregar arista $(1, 5)$: $E' = \{(4, 5), (6, 1), (0, 2), (6, 3), (4, 0), (1, 5)\}$
- No agregar arista $(5, 0)$, pues formaría un ciclo.
- No agregar arista $(1, 4)$, pues formaría un ciclo.
- No agregar arista $(2, 3)$, pues formaría un ciclo.
- No hay más aristas.

$E' = \{(4, 5), (6, 1), (0, 2), (6, 3), (4, 0), (1, 5)\}$

El grafo resultante queda de la siguiente manera:



El algoritmo necesita recorrer una vez todas las aristas, por lo tanto su complejidad es $O(|E|)$.

Puntaje:

3 pts: Las 6 aristas finales agregadas. Pueden mostrarse como lista o gráficamente.

2 pts: Alguna arista faltante o incorrecta

1 pto: Pocas aristas correctas, o hay un ciclo.

+1 pto: complejidad lineal

3. **[13p]** Representaciones y operaciones númericas:

   3.1) **[9p]** Complete la siguiente tabla usando conversiones numéricas (no detalle el procedimiento):

decimal	binario	hexadecimal
1200	**0100 1011 0000**	**4B0**
2048	**1000 0000 0000**	**800**
3072	**1100 0000 0000**	**C00**
33	0010 0001	**21**
90	0101 1010	**5A**
412	0001 1001 1100	**19C**
172	**1010 1100**	AC
231	**1110 0111**	E7
648	**0010 1000 1000**	288

Puntaje:

+0,5 pts cada resultado. No es necesario agregar ceros a la izquierda, ni espacios, pero tampoco es un error.

3.2) **[4p]** Efectúe las siguientes operaciones de números hexadecimales explicitando su procedimiento. Puede hacerlo en notación hexadecimal o binaria, pero **no es válido** convertirlas a decimal (salvo si quiere comprobar un resultado):

a) `0x23` + `0x0E`

R. Pasándolo a binario, esto es:

$$\begin{array}{rl} 0010\ 0011 & = \texttt{0x23} \\ +0000\ 1110 & = \texttt{0x0E} \\ \hline 0011\ 0001 & = \texttt{0x31} \end{array}$$

Como comprobación, la suma en decimal es: $35 + 14 = 49$, que corresponde al hexadecimal `0x31`.

b) `0x79` − `0x64`

R. Para restar `0x64` podemos sumar su versión escritan en complemento de 2. `0x64` es 0110 0100. Al invertir sus bit obtenemos 1001 1011, y al sumar 1 queda 1001 1100. Por lo tanto, el valor −`0x64` escrito en complemento de 2 es `0x9C`.

La suma queda:

$$\begin{array}{rl} 0111\ 1001 & = \texttt{0x79} \\ +1001\ 1100 & = \texttt{0x9C} \\ \hline 10001\ 0101 & = \texttt{0x115} \end{array}$$

, descartando el bit de *carry*, queda:

$$\begin{array}{rl} 0111\ 1001 & = \texttt{0x79} \\ +1001\ 1100 & = \texttt{0x9C} \\ \hline 0001\ 0101 & = \texttt{0x15} \end{array}$$

El resultado es `0x15`, que se interpreta como un número positivo pues su primer bit es 0. Por lo tanto, no es necesario volver a convertirlo usando complemento de 2.

Como comprobación, la operación en decimal es $121 - 100 = 21$, que corresponde a `0x15`.

c) `0x513D` + `0x8`

R. Esta podría hacerse contando 8 valores más en hexadecimal a partir de `0x513D`, que da como resultado `0x5145`.

Al sumar en binario queda:

$$\begin{array}{rl} 0101\ 0001\ 0011\ 1101 & = \texttt{0x513D} \\ +0000\ 0000\ 0000\ 1000 & = \texttt{0x0008} \\ \hline 0101\ 0001\ 0100\ 0101 & = \texttt{0x5145} \end{array}$$

Como comprobación, la suma en decimal es: $20797 + 8 = 20805$, que corresponde a `0x5145`.

d) `0x50FD` − `0x503C`

R. Esto podría resolver calculando la diferencia entre `0xFD` y `0x3C`, que podría deducirse que es `0xC1`. Sabemos que el resultado debe ser positivo pues `0x50FD` es mayor.

La versión en complemento de 2 de `0x503C` se calcula de la siguiente manera: el valor `0x503C` es 0101 0000 0011 1100; al invertir los bit queda 1010 1111 1100 0011, y al sumarle 1 se obtiene el valor 1010 1111 1100 0100 que en hexadecimal es `0xAFC4`.

La suma, descartando el bit de *carry* que se genera, queda:

$$\begin{array}{rl} 0101\ 0000\ 1111\ 1101 & = \texttt{0x50FD} \\ +1010\ 1111\ 1100\ 0100 & = \texttt{0xAFC4} \\ \hline \not{1}0000\ 0000\ 1100\ 0001 & = \texttt{0x00C1} \end{array}$$

Como comprobación, la operación en decimal es: $20733 - 20540 = 193$, que corresponde al hexadecimal `0xC1`.

Puntaje:

1 pto por cada resultado correcto.

0,5 pto si hay algún error menor. No castigar dos veces si se produce un error de arrastre.

0 pts si hay más de un error en el cálculo.

4. **[12p]** Para las siguientes construcciones con celdas lógicas:

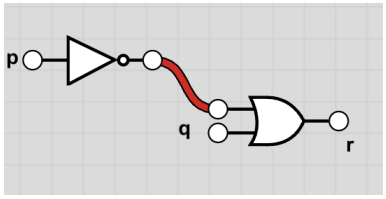
   4.1) **[4p]** Construir una tabla con todas las salidas posibles.

   R: Este circuito digital tiene 2 entradas: p y q por lo tanto hay 4 combinaciones posibles. Se mostrarán las combinaciones con 0 y 1, pero también puede hacerse con V/F, ó con T/F.

p	q	r
0	0	1
0	1	1
1	0	0
1	1	1

, también podría escribirse de manera más explícita como:

p	not p	q	$r =$ not p or q
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1



Bonus (1pto): ¿qué valor lógico representa esto?

R. El valor r es not p or q (también podría escribirse con notación lógica como $\neg p \vee q$, lo que equivale a la operación lógica $p \Rightarrow q$ ("p implica q"). Esta operación solo es falsa cuando p es verdadero y q es falso, y verdadera en todos los otros casos. Notar que $p \Rightarrow q$ es falsa solamente cuando, a partir de algo verdadero (p), se deduce algo falso (q), ya que esto no debe ocurrir. Las otras combinaciones reflejan el hecho que a partir de algo verdadero solo se puede deducir algo verdadero; y que a partir de algo falso se puede deducir cualquier cosa.

Puntaje:

+1 pto: enumerar las 4 combinaciones de p y q

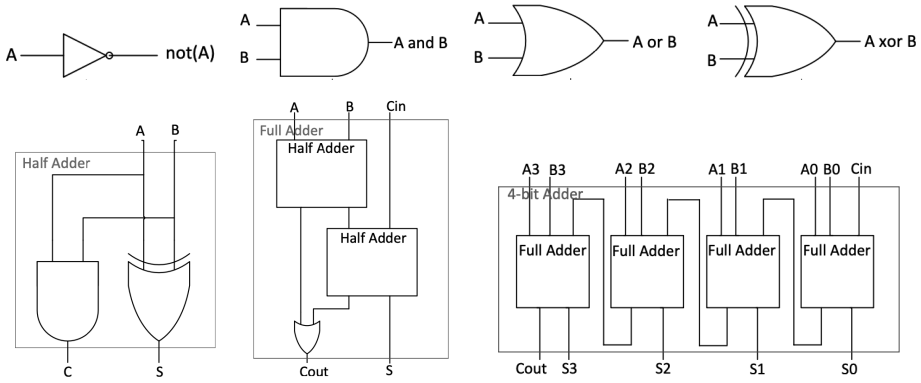
+1 pto: calcular correctamente el valor de la celda *not*. Si no se muestra (no se pedía), pero el resultado final está correcto, otorgar el punto

+1 pto: calcular correctamente el valor de la celda *or*. Si no se muestra (no se pedía), pero el resultado final está correcto, otorgar el punto

+1 pto: mostrar correctamente los valores de r para las 4 combinaciones de p y q

+1 pto (bonus): basta con decir que es la 'implicancia lógica' o similar. Decir que el valor lógico es $\neg p \vee q$ no es suficiente.

4.2) **[8p]** Ejecute la suma de los valores hexadecimales: $0xC$ y $0x6$ usando un full-adder de 4 bit. Muestra la entrada y salida del full-adder. Como referencia se agregan los diagramas vistos en clases.



R. Si asignamos $A = \texttt{0xC} = 0b1100$, y $B = \texttt{0x6} = 0b0110$, entonces los bit de entrada son:

A_3	A_2	A_1	A_0
1	1	0	0

, y para el otro operando:

B_3	B_2	B_1	B_0
0	1	1	0

Por otro lado, el *full-adder* no requiere estar conectado a ningún *full-adder* previo, por lo tanto no hay un *carry* previo que agregar, y $C_{in} = 0$.

El resultado de la suma es $S = \texttt{0xC} + \texttt{0x6} = \texttt{0x12} = 0b10010$. Se produce un bit de *carry* porque el resultado tiene 5 bit y habían entrado 4 desde cada operando. Los bit de salida quedan:

C_{out}	S_3	S_2	S_1	S_0
1	0	0	1	0

Como resultado basta mencionar los valores de cada uno de 9 bit que entran al *full-adder*, y los valores de cada uno de los 5 bit que salen de él. Es posibe mostrarlo como lista, como tabla, o dibujándolo dentro del *full-adder*. Es indiferente si A es `0xC` y B es `0x6` o al revés.

Puntaje

+1 pto: los 4 bit de entrada de A

+1 pto: los 4 bit de entrada de B

+1 pto: determinar que el *carry* de entrada es 0

+1 pto: calcular el resultado correcto de alguna manera (`0x12`). Si solo se muestra el resultado final y está correcto, otorgar este punto.

+3 pto: indicar correctamente los 4 bit de salida de S. Puntaje intermedio si hay errores pero hay desarrollo. Castigar errores de arrastre solo 1 vez.

+1 pto: determinar que el *carry* de salida es 1

5. **[11p]** Una manera de codificar secuencias de dígitos mediante bit es la siguiente: la secuencia empieza y termina con un bit 0; cada dígito d se representa mediante d bit 1 seguidos; y para separar dos dígitos consecutivos se utiliza un bit 0. Por ejemplo, la secuencia 242 se representa como 011011110110. Se empieza con un 0, a continuación 2 bit en 1, luego un bit 0 como separación, a continuación 4 bit en 1, un bit 0 en como separación, 2 bit en 1, y finalmente termina con 0. El dígito 0 se representa con 10 bit 1 seguidos.

   Para que esta secuencia se pueda escribir en Byte, hay que completar con 0's a la izquierda hasta que la cantidad de bit sea un múltiplo de 8. En el ejemplo anterior, que tiene 12 bit, se le agregan 4 bit 0 a la izquierda. El resultado queda 0000011011110110. Esto se puede escribir con 2 Byte.

   Construya un archivo con el siguiente formato:

   - **[3p]** 8 Byte, donde cada Byte contiene el **caracter** ASCII correspondiente a la fecha de hoy en formato `YYYYMMDD`. Atención que son dos caracteres para el mes y dos para el día.

     R. La fecha actual es `20230605`. Se solicita escribir los byte correspondientes a cada **caracter ASCII**, por lo tanto hay que escribir 8 Byte: el Byte que representa el caracter `2`, el Byte que representa el caracter `0`, etc.

Caracter ASCII	`2`	`0`	`2`	`3`	`0`	`6`	`0`	`5`
Byte	`32`	`30`	`32`	`33`	`30`	`36`	`30`	`35`

     Puntaje:

     3 pts: Correcto

     2 pts: Un error leve.

     1 pto: Idea correcta, pero mal ejecutada. Por ejemplo, no agregar los `0`s o no seguir el orden del formato `YYYYMMDD`.

     0 pto: Resultado incorrecto, o bien no utiliza caracteres ASCII, o no usa los 8 Byte.

   - **[1p]** 1 Byte con el valor numérico 201 si usted está en la sala AE201, o el valor 202 si usted está en la sala AE202.

     R. Ahora se pide el **valor numérico** 201 ó 202. Cada uno de estos valores cabe en un Byte, pues un Byte permite almacenar hasta el valor 255. Si se pidieron los caracter ASCII, necesitaríamos 3 Byte (uno por cada caracter).

Valor	201 ó 202
Byte	`C9` ó `CA`

     Puntaje:

     1 pto. Correcto o error muy menor.

     0 pto. Incorrecto. Por ejemplo, ocupar más de 1 byte.

   - **[2p]** 1 Byte con el valor numérico que representa la cantidad de Byte (el valor X) usados en la siguiente parte.

     R. El valor numérico X es 6 (siguiente paso), por lo tanto el Byte que se debe agregar aquí es `06`

   - **[4p]** X Byte con la codificación del número 16251830 de acuerdo a lo explicado.

     Puntaje:

     Puntaje:

     2 pts. Correcto. Si el valor a mostrar viene incorrecto desde el paso siguiente, pero está bien mostrado aquí, considerar correcto.

     1 pto. Correcto o error menor. Por ejemplo, ocupar solo medio Byte (6 en lugar de `06`)

     0 pto. Incorrecto.

     R. La codificación de cada dígito es esa cantidad de 1's seguidos:

Dígito	1	6	2	5	1	8	3	0
Codificación	`1`	`111111`	`11`	`11111`	`1`	`11111111`	`111`	`1111111111`

     Para separar la codificación de cada dígito, se agrega un `0` al inicio, uno al final, y uno entre medio de cada dígito codificado. La codificación completa de 16251830 en bit queda como:

`010111111011011111010111111111011101111111110`

En esa secuencia hay 45 bit. Para llegar a un múltiplo de 8 hay que agregar 3 bit más a la izquierda:

`000`010111111011011111010111111111011101111111110

Para poder representar esta secuencia como Byte, debemos agrupar los bit en grupos de 8.

Bits	0000 1011	1111 0110	1111 1010	1111 1111	0111 0111	1111 1110
Byte	`0B`	`F6`	`FA`	`FF`	`77`	`FE`

Se necesitan 6 Byte (48 bit), por lo tanto el valor X es 6.

Puntaje

4 pts. Correcto.

3 pts. Un error. Ejemplos: falta un cero al inicio, o falta un cero al final, o faltan 0s intermedios, o no se agregan los 0s al inicio.

1 pto. Dos o tres errores distintos.

0 pts. Más de 3 errores. Incorrecto.

- **[1p]** El archivo debe quedar con 16 bytes en cada línea. Si es necesario debe agregar el byte correspondiente al valor numérico 255 al final, y repetirlo hasta completar una línea de 16 byte.

  R. El valor numérico 255 en Byte corresponde a `FF`. Si escribimos los Byte usando 16 Byte por línea, queda:

Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Byte	`32`	`30`	`32`	`33`	`30`	`36`	`30`	`35`	`C9`	`06`	`0B`	`F6`	`FA`	`FF`	`77`	`FE`

  La cantidad de Byte requerida es exactamente 16, por lo tanto no es necesario agregar ningún Byte `FF` al final.

  Si estaba en la sala AE202, el resultado es:

Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Byte	`32`	`30`	`32`	`33`	`30`	`36`	`30`	`35`	`CA`	`06`	`0B`	`F6`	`FA`	`FF`	`77`	`FE`

  Puntaje

  1 pto. Correcto. No agregar nada, o bien agregar consistentemente los `FF` si es que había un error de arrastre previo.

  0 pto. Incorrecto. No agrega `FF` cuando sí había que hacerlo.

Puntaje

No más de 5 pts en total si escribe los Byte en decimal en lugar de hexadecimal.