Pregunta 1

1. (1 pt.) Especifique las funciones de hash1 y hash2 rellenando la tabla de hash. El par es (chip, RUN)
   h1(chip) = chip mod 11
   h2(chip) = chip mod 23
   Los valores de h1 son

k	h1(k)
0	0
22	0
31	9
4	4
15	4
28	6
17	6
88	0
59	4

2. (0,5 pts.) Dibuje la tabla usando colisión por encadenamiento.

Índice	**Valor**
0	$88 \rightarrow 22 \rightarrow 0$
1	
2	
3	
4	$59 \rightarrow 15 \rightarrow 4$
5	
6	$17 \rightarrow 28$
7	
8	
9	31
10	

3. (0,5 pts.) Dibuje la tabla usando colisión por sondeo lineal.

	valor
0	0
1	22
2	88
3	
4	4
5	15
6	28
7	17
8	59
9	31
10	

4. (0,5 pts.) Dibuje la tabla usando colisión por sondeo cuadrático.

Índice	**Valor**
0	0
1	22
2	
3	88
4	4
5	15
6	28
7	17
8	59
9	31
10	

Para las siguientes preguntas, haga uso de su función de hash1 y hash2

5. (0,5 pts.) Dibuje la tabla usando colisión por rehashing. Si el factor de carga es alto Rehashing aumenta el tamaño de la tabla aprox. al doble, en este caso 23 si respondieron hash doble y está bueno, max 0,4pts.

6. (2 pts.) Compare las diferentes técnicas de resolución de colisiones en términos del número de cálculos (no complejidad algorítmica) y la concentración de las llaves para las técnicas elegidas. la probabilidad de colisión es el factor de carga en cada inserción

técnica	cálculos	concentración
encadenamiento	1	bajo en la tabla, mayor en la lista
sondeo lineal	1+1 por colisión	alto
sondeo cuadrático	1+1 por colisión	menor que lineal
rehashing	1+1 por colisión (la mitad)	bajo porque se agranda cuando el factor de carga es alto

7. (1 pt.) Cómo cambia su respuesta anterior si en el bucket caben 2 pares (clave, RUN).
Si caben 2 pares cada bucket soporta el doble de valores, dicho de otra forma la tabla soporta una colisión sin necesidad de resolución

Índice	Valor
0	0
1	-
2	-
3	-
4	4
5	-
6	-
7	31
8	-
9	-
10	-
11	-
12	-
13	-
14	59
15	15
16	-
17	17
18	-
19	-
20	-
21	88
22	22
23	28

2. Backtracking

Considere el siguiente acertijo lógico: En cinco casas alineadas de izquierda a derecha, cada una de un color diferente, viven 5 personas de diferentes nacionalidades, cada una de las cuales prefiere un deporte diferente, una bebida diferente y una mascota diferente. Además, están dados los siguientes hechos:

- La persona inglesa vive en la casa roja.
- La persona española tiene el perro.
- La persona argentina vive en la primera casa de la izquierda.
- Se prefiere el fútbol en la casa amarilla.
- La persona que prefiere el basketball vive en la casa junto a la persona que tiene el gato.
- La persona argentina vive junto a la casa azul.
- La persona que prefiere el voleyball tiene el conejo.
- La persona que prefiere el tenis bebe jugo de naranja.
- La persona chilena bebe té.
- La persona japonesa prefiere el judo.
- Se prefiere el fútbol en la casa junto a la casa donde se tiene el caballo.
- Se bebe café en la casa verde.
- La casa verde está inmediatamente a la derecha (a su derecha) de la casa blanca.
- Se bebe leche en la casa del medio.

La pregunta del acertijo es: **¿Dónde vive la cebra y en qué casa se bebe agua?**. En este contexto se le pide:

a) (1 pt.) Modele este problema como un CSP (Problema de Satisfacción de Restricciones), para esto defina cuáles son las variables, dominios y restricciones del problema.

- **Variables:**
  - `Color[i]`, $1 \leq i \leq 5$: color de la casa i.
  - `Nacionalidad[i]`, $1 \leq i \leq 5$: nacionalidad de la persona que vive en la casa i.
  - `Deporte[i]`, $1 \leq i \leq 5$: deporte preferido por la persona que vive en la casa i.
  - `Bebida[i]`, $1 \leq i \leq 5$: bebida preferida por la persona que vive en la casa i.
  - `Mascota[i]`, $1 \leq i \leq 5$: mascota preferida por la persona que vive en la casa i.
- **Dominios:**
  - `Color` = {*rojo, amarillo, azul, verde, blanco*}.
  - `Nacionalidad` = {*chilena, peruana, inglesa, española, argentina*}.
  - `Deporte` = {*fútbol, tenis, vóleibol, básquetbol, judo*}.
  - `Bebida` = {*té, jugo de naranja, café, leche, agua*}.
  - `Mascota` = {*perro, gato, conejo, caballo, cebra*}.
- **Restricciones:** Las restricciones están dadas por los hechos definidos para el problema. Por ejemplo: "La persona inglesa vive en la casa roja".

b) (2 pts.) Proponga el pseudocódigo de la función `solveCSP()` para resolver el acertijo, indicando los parámetros adecuados para realizar la llamada inicial y obtener la solución al acertijo. Asuma que dispone de una función `verificaCSP()` adecuada para verificar las restricciones del problema.

```
input : mascota buscada, bebida buscada
output: id de la casa donde está la mascota, id de la casa donde está la bebida

solveCSP(mascota, bebida):
0    for i = 0, 24 Casa[i] ← null
1    if Acertijo(Casa, 0):       // resuelve el acertijo, si tiene solución es true
2        for i = 0, 24           // recorre todas las variables
```

```
3            if Casa[i] = mascota
4                m ← (i + 1) / 5 // el ID de la casa correspondiente
5            if Casa[i] = bebida
6                b ← (i + 1) / 4 // el ID de la casa correspondiente
7        return b, m             // retorna los ID de la casa de la bebida y la mascota
8    return -1, -1               // no había solución

input : Arreglo Casa[0...24], // todas las variables de forma consecutiva en el arreglo Casa
        índice 0  i  25
output: true si hay solución, en Casa están los valores de cada variable

Acertijo(Casa, i):
1    if i = 25: return true
2    for v = Dominio(Casa[i])
3        if verificaCSP(Casa, i, v): // asume que tiene acceso a los hechos (restricciones)
4            Casa[i] ← v
5            if Acertijo(Casa, i + 1):
6                return true
7            Casa[i] ← null
8    return false
```

c) (2 pts.) Explique cómo aplicaría en su solución podas y propagación, de modo de beneficiar el desempeño de su solución al utilizar los hechos conocidos del acertijo.

Las podas se pueden aplicar en la línea 3 de `Acertijo()`, por ejemplo, dado que sabemos que "La persona argentina vive en la primera casa de la izquierda" podemos agregar una restricción que rechace cualquier valor de dominio distinto para la nacionalidad de la persona de la `Casa[1]`. De igual forma, es posible aplicar propagación: al asignar un valor de una variable en la línea 4 de `Acertijo()` retiramos ese valor de los dominios de las variables asociadas. Por ejemplo, al asignar *tenis* como deporte, retiramos *tenis* de los dominios de las demás variables por asignar.

d) (1 pt.) Adicionalmente a los hechos indicados en el acertijo, se sabe que habitualmente (la mayor parte de las veces): Los argentinos prefieren el fútbol y el café, los japoneses el judo y los gatos, los chilenos el fútbol y el té, los españoles los perros y el café, los ingleses el té y el tenis. ¿De qué forma utilizaría esta información para mejorar el desempeño de su solución?

Este caso corresponde a la aplicación de heurísticas en `Backtracking`. En este caso, como las variables se eligen secuencialmente, lo que podemos hacer en la línea 2 de `Acertijo()` es escoger el valor del dominio de v a evaluar según las asignaciones ya definidas en `Casa`.

Programación dinámica

Considera la ruta Santiago – Puerto Montt. A lo largo de esta ruta, la autoridad vial ha dispuesto n lugares en los cuales está permitido poner letreros con propaganda. Estos lugares —$x_1, x_2, \ldots, x_n$— están especificados por sus distancias desde Santiago, en km, siendo x_1 el más cercano. Por otra parte, tu negocio contrató a una empresa de marketing que calculó que si pones un letrero en el lugar x_i, entonces vas a recibir una ganancia de r_i (que representa las ventas que va a hacer tu negocio gracias a esa propaganda).

Hay, sin embargo, una única restricción legal que especifica que un mismo negocio no puede poner letreros a 5 km o menos ~~de 5 km~~ de separación entre ellos.

Por lo tanto, la pregunta es, ¿en cuáles lugares —un subconjunto de $x_1, x_2, \ldots, x_n$— te conviene poner los letreros con propaganda de tu negocio, cumpliendo con la restricción anterior, de manera de *maximizar la suma de las ganancias que recibirías*? Por ejemplo, si $n = 4$, $(x_1, x_2, x_3, x_4) = (6, 7, 12, 14)$ y $(r_1, r_2, r_3, r_4) = (5, 6, 5, 1)$, entonces la solución óptima sería poner los letreros en x_1 y x_3 para una ganancia total de 10.

Plantea un algoritmo de programación dinámica para resolver este problema.

a) Considera las dos alternativas de que el lugar x_n esté o no en la solución óptima; ¿cuál es el problema que queda por resolver en cada caso?

Llamemos O a la solución óptima. Para cada lugar x_k consideremos el lugar x_j, $j < k$ (es decir, x_j está más cerca de Santiago que x_k), tal que x_j es el lugar más cercano a x_k que está a una distancia > 5 km de x_k; llamemos $b(k)$ a este lugar.

Así, si x_n está en O, entonces el lugar anterior a x_n que también podría estar en O es $b(n)$; es decir, O sería x_n más (los lugares correspondientes a) **la solución óptima al problema definido por los lugares $x_1, \ldots, b(n)$**.

En cambio, si x_n no está en O, entonces O es igual a **la solución óptima para los lugares $x_1, \ldots, x_{n-1}$**.

b) ¿Cómo se generaliza este razonamiento si consideramos el problema definido sólo por los primeros k lugares: $x_1, x_2, \ldots, x_k$?

Sea O_k la solución óptima para los lugares $x_1, \ldots, x_k$. (Es decir, en a) buscamos O_n.). Generalizando el razonamiento de a), si O_k incluye al lugar x_k, entonces es igual a x_k más $O_{b(k)}$; y si O_k no incluye al lugar x_k, entonces es igual a O_{k-1} .

c) Si $opt(k)$ representa la ganancia de un subconjunto óptimo de lugares entre $x_1, x_2, \ldots, x_k$, ¿cuál es la recurrencia correspondiente? es decir, si $opt(k) = \max\{ \ldots , \ldots \}$, ¿qué va en los …?

$$opt(k) = \max\{ opt(k\text{-}1),\ r_k + opt(b(k)) \}$$

d) Así, finalmente, a partir de la respuesta a c), plantea el algoritmo pedido.

Esta es la versión recursiva más directa del algoritmo, a partir de la recurrencia anterior:

```
opt(j):
        if j = 0:
                return 0
        else:
                return max{ v_j + opt(b(j)) , opt(j-1) }
```

Por supuesto, también son válidas la versión iterativa, y la variante de la versión recursiva en que los valores *opt*(*k*) se van almacenando en una tabla a medida que se van calculando y se sacan de la tabla cada vez que vuelven a aparecer en la recursión.

Grafos

a) Una forma de recorrer (y descubrir) un grafo no direccional $G = (V, E)$ es la siguiente: Primero, elegimos un vértice cualquiera de V y lo ponemos en una cola Q inicialmente vacía; Q es una cola común, del tipo "el primero que entra es el primero que sale". Luego, hacemos lo siguiente: Repetidamente, sacamos el primer vértice de Q, llamémoslo u; recorremos la lista de adyacencias de u y cada vértice que encontramos en la lista lo ponemos en Q.

i) Explica qué se puede hacer para que este algoritmo no "repita" vértices, es decir, para que los vértices que ya han sido descubiertos no vuelvan a ser descubiertos.

Al igual que en BFS, usamos colores. Dos colores es suficiente para hacer la distinción pedida. Podemos usar tres colores si además queremos distinguir los vértices que aún estamos descubriendo.

ii) ¿Cuál es la condición de término del algoritmo?

De acuerdo con el enunciado, cuando la cola Q queda vacía.

iii) ¿Qué información adicional puede extraerse de este algoritmo en términos de la distancia, medida en número de aristas, que hay entre el primer vértice que pusimos en Q y cada uno de los otros vértices que van siendo descubiertos?

Hay que darse cuenta de que un vértice es descubierto a través del menor número posible de aristas desde el primer vértice: es decir, el algoritmo descubre todos los vértices que están a distancia k del primer vértice —distancia medida en número de aristas que separan a ambos vértices— antes de descubrir cualquier vértice que está a distancia k+1.

iv) Describe una versión completa del algoritmo —es decir, incluyendo los tres puntos anteriores— en pseudocódigo similar al usado en clases para describir el algoritmo DFS.

—En la próx. página

```
algoritmoPedido(s):   —s es el vértice (arbitrario) de partida
for each u in V–{s}:
        u.color ← white        —asignación inicial de color a todos los vértices
        u.δ ← ∞                —asignación inicial de distancia desde el primer vértice
s.color ← gray          —acabamos de "descubrir" el vértice de partida
s.δ ← 0                 —… que está a distancia 0 de sí mismo
Q ← cola        —inicializamos una cola Q
Q.enqueue(s)            —…y ponemos s en la cola
while !Q.empty( ):      —condición de término del algoritmo
        u ← Q.dequeue( )      —sacamos el primer vértice de la cola
        for each v in α[u]:   —recorremos la lista de vértices adyacentes a u
                if v.color = white:     —si el vértice adyacente v no ha sido descubierto
                        v.color ← gray          —… lo descubrimos
                        v.δ ← u.δ+1     —…calculamos su distancia al primer vértice
                        Q.enqueue(v)    —…y lo ponemos en la cola
```

b) Queremos saber si estando en una estación del metro, puedo ir a cualquier otra estación usando únicamente el metro. Para esto, representamos la red del metro como un grafo direccional, de la siguiente manera: Cada estación es un vértice, y, si en una misma línea del metro la estación v es la próxima estación a la estación u, entonces (y sólo entonces) incluimos una arista direccional (u, v).

i) Describe un algoritmo eficiente en pseudocódigo que permita responder la pregunta original; tu algoritmo puede hacer uso sólo del algoritmo DFS estudiado en clases.

Si representamos la red del metro como se describe arriba, entonces la respuesta a la pregunta inicial es "sí" si y sólo si el grafo es una sola gran componente fuertemente conexa.

Por lo tanto, el algoritmo que se pide es el algoritmo de componentes fuertemente conexas (CFCs) con la única particularidad de que al terminar hay que ver cuántas CFCs encontró: si encontró exactamente una, entonces el algoritmo imprime "sí", en cualquier otros caso, "no".

Cuando el enunciado dice "*tu algoritmo puede hacer uso sólo del algoritmo DFS estudiado en clases*," quiere decir que hay que escribir el algoritmo con un nivel de detalle similar al usado en clases (que hace uso de DFS), y hay que agregarle la condición final; no se puede poner como respuesta, "*el algoritmo CFC estudiado en clases*."

ii) Justifica que tu algoritmo es eficiente.

El algoritmo básicamente hace dos recorridos BFS del mismo grafo, primero, sobre la versión original del grafo, y luego sobre la versión transpuesta. Luego, es $O(V+E)$, es decir, es lineal en el "tamaño" del grafo: números de vértices y número de aristas.