Ayudantía Repaso I2

2024-1

Dafne Arriagada, Alexander Infanta

Temas a revisar

1. Hashing
2. DFS
3. Backtracking (BT) + Grafos
4. Programación Dinámica (PD)

HASHING

HASHING - Cómo abordar problemas de Hashing? + TIPS

1, Intenta escoger una función de hash que, probando algunos valores, te brinde un distribución más menos uniforme de las keys (siempre vamos a querer disminuir colisiones)

2. Lleva algunas funciones de hash en mente ej: X mod 5

3. Entiende bien el manejo de colisiones.

4. Tips de ayudante

HASHING - EJERCICIO - **P2 I2 2018-2**

PARTE I - HASHING

Queremos multiplicar dos números X y Y, y para verificar si el resultado $Z = X \times Y$ es correcto, aplicamos una función de hash h a los tres números y vemos si

$$h(\, h(X) \times h(Y)\,) = h(Z).$$

Si los números difieren, entonces cometimos un error, pero si son iguales, entonces el resultado es (muy probablemente) correcto. h se define como calcular repetidamente la suma de los dígitos, hasta que quede un solo dígito, en cuyo caso 9 cuenta como 0.

P.ej., si $X = 123456$ y $Y = 98765432$, entonces $Z = 121931851729992$, y $h(X) = h(21) = 3$, $h(Y) = h(44) = 8$, $h(Z) = h(60) = 6$, y efectivamente, $h(3 \times 8) = h(24) = 6 = h(Z)$. Notemos que el dígito 9 no influye en el resultado calculado de h; p.ej., $h(49) = h(13) = 4$.

HASHING - EJERCICIO - **P2 I2 2018-2**

Queremos multiplicar dos números X y Y, y para verificar si el resultado $Z = X \times Y$ es correcto, aplicamos una función de hash h a los tres números y vemos si

$$h(\, h(X) \times h(Y)\,) = h(Z).$$

Si los números difieren, entonces cometimos un error, pero si son iguales, entonces el resultado es (muy probablemente) correcto. h se define como calcular repetidamente la suma de los dígitos, hasta que quede un solo dígito, en cuyo caso 9 cuenta como 0.

P.ej., si $X = 123456$ y $Y = 98765432$, entonces $Z = 121938185172992$, y $h(X) = h(21) = 3$, $h(Y) = h(44) = 8$, $h(Z) = h(60) = 6$, y efectivamente, $h(3 \times 8) = h(24) = 6 = h(Z)$. Notemos que el dígito 9 no influye en el resultado calculado de h; p.ej., $h(49) = h(13) = 4$.

a) Da una expresión matemática para $h(X)$ [**1.5 pts.**]; y **b**) muestra que este método de verificación del resultado de la multiplicación es correcto [**1.5 pts.**].

HASHING - EJERCICIO - **P2 I2 2018-2**

a) Da una expresión matemática para $h(X)$ **[1.5 pts.]**

HASHING - EJERCICIO - **P2 I2 2018-2**

b) muestra que este método de verificación del resultado de la multiplicación es correcto [**1.5 pts.**].

Sabemos que:

$$X = 9 \times Q_1 + R_1$$

$$Y = 9 \times Q_2 + R_2$$

$$X \times Y = Z$$

Y queremos demostrar que:

$$h(h(X) \times h(Y)) = h(Z)$$

HASHING - EJERCICIO - **P2 I2 2018-2**

b) muestra que este método de verificación del resultado de la multiplicación es correcto [**1.5 pts.**].

$$h(h(9 \times Q_1 + R_1) \times h(9 \times Q_2 + R_2) = h(X \times Y)$$
$$h(h(9 \times Q_1 + R_1) \times h(9 \times Q_2 + R_2) = h((9 \times Q_1 + R_1) \times (9 \times Q_2 + R_2))$$

En la expresión de la **izquierda** podemos eliminar todos los términos múltiplos de 9 ya que estamos trabajando en mod 9, obteniendo

$$h(R_1 \times R_2) = h((9 \times Q_1 + R_1) \times (9 \times Q_2 + R_2))$$
$$h(R_1 \times R_2) = h(81 \times Q_1 \times Q_2 + 9 \times Q_1 \times R_2 + 9 \times Q_2 \times R_1 + R_1 \times R_2)$$

*factorizamos por 9.

Bajo el mismo argumento, podemos repetir la estrategia para la expresión de la **derecha**

$$h(R_1 \times R_2) = h(9 \times (9 \times Q_1 \times Q_2 + Q_1 \times R_2 + Q_2) + R_1 \times R_2)$$
$$h(R_1 \times R_2) = h(R_1 \times R_2)$$

qed.

HASHING - EJERCICIO - **P2 I2 2018-2**

PARTE II - TABLAS DE HASH

Para cada uno de los siguientes problemas, responde si es posible resolver el problema eficientemente mediante *hashing*. En caso afirmativo, explica claramente cómo; de lo contrario, sugiere otra forma de resolverlo según lo estudiado en el curso.

a) Considera un sistema de respaldo en que toda la información digital de una empresa tiene que ser respaldada, es decir, copiada, cada cierto tiempo. Una propiedad de estos sistemas es que solo una pequeña fracción de toda la información cambia entre un respaldo y el siguiente. Por lo tanto, en cada respaldo, solo es necesario copiar la información que efectivamente ha cambiado. El desafío es, por supuesto, encontrar lo más que se pueda de la información que no ha cambiado. **[1.5 pts.]**

HASHING - EJERCICIO - **P2 I2 2018-2**

PARTE II - TABLAS DE HASH

a) Considera un sistema de respaldo en que toda la información digital de una empresa tiene que ser respaldada, es decir, copiada, cada cierto tiempo. Una propiedad de estos sistemas es que solo una pequeña fracción de toda la información cambia entre un respaldo y el siguiente. Por lo tanto, en cada respaldo, solo es necesario copiar la información que efectivamente ha cambiado. El desafío es, por supuesto, encontrar lo más que se pueda de la información que no ha cambiado. [**1.5 pts.**]

HASHING - EJERCICIO - **P2 I2 2018-2**

PARTE II - TABLAS DE HASH

b) Dada una lista L de números, queremos encontrar el elemento de L que sea el más cercano a un número dado x. **[0.5 pts.]**

Queda **item c** como desafío!

DEEP FIRST SEARCH

DFS EN 4 MINUTOS

DFS - Cómo abordar problemas DFS? + TIPS

1. No memoricen el algoritmo y sus variantes, mejor es entender (DFS, DFS Visit, qué significa cada color, etc)

2. Perspectiva: Con modificaciones a DFS llegas a Kosaraju (CFC) y orden topológico (TopSort) -> Clave tener esas transiciones en mente.

3. En clases se da énfasis a la estrategia recursiva, pero también puede presentar el caso iterativo (lo veremos ahora)

4. Tips de ayudante

DFS - EJERCICIO - **P3 I2 2018-1**

Se nos entrega la siguiente versión generalizada de DFS (iterativa) para recorrer

```
dfs(graph G, node v)
    stack s
    s.push(v)
    v.state ← discovered
    while not s.empty()
        node u = s.pop()
        for v in G.adjacent[u]
            if v is not discovered
                s.push(v) # agregar vecinos de u para luego visitarlos
                v.state ← discovered
```

Consejos monkis -> luego de este pueden hacer el P1 - 2019 - 1 (.docx parte 2)

DFS - EJERCICIO - **P3 I2 2018-1**

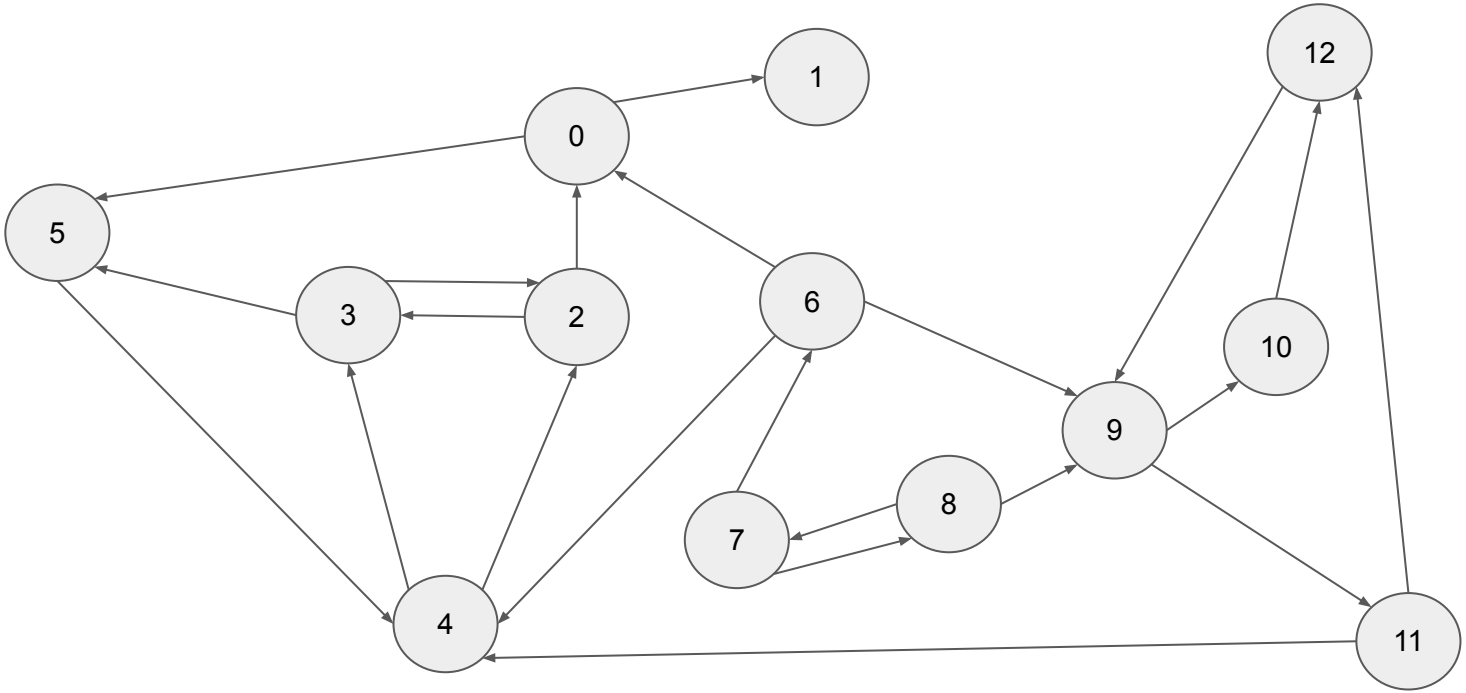
b) Considera el siguiente grafo *G* direccional (las aristas tienen dirección), representado mediante sus listas de adyacencias:

[0]: 5–1	[1]:	[2]: 0–3	[3]: 5–2	[4]: 3–2	[5]: 4	
[6]: 9–4–0	[7]: 6–8	[8]: 7–9	[9]: 11–10	[10]: 12	[11]: 4–12	[12]: 9

Ejecuta el algoritmo DFS anterior a partir del vértice $v = 0$. Muestra el orden en que los vértices van siendo marcados y el contenido del stack *S* cada vez que cambia.

DFS - EJERCICIO - **P3 I2 2018-1**

Ejecuta el algoritmo DFS anterior a partir del vértice $v = 0$. Muestra el orden en que los vértices van siendo marcados y el contenido del stack S cada vez que cambia.



DFS - EJERCICIO - **P3 I2 2018-1**

a) Supongamos que *G* es no direccional (las aristas no tienen dirección). Decimos que *G* es **biconec-tado** si no hay ningún vértice que al ser sacado de *G* desconecta el resto del grafo. Por el contrario, si *G* no es biconectado, entonces los vértices que al ser sacados de *G* desconectan el grafo se conocen co-mo **puntos de articulación**.

Describe un algoritmo eficiente para encontrar todos los puntos de articulación en un grafo conectado. Explica cuál es la complejidad de tu algoritmo.

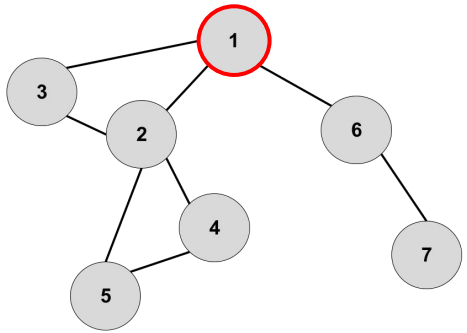


Figura: Ejemplo punto de articulación / corte

DFS - EJERCICIO - **P3 I2 2018-1**

Este problema es recurrente, de hecho fue parte de una tarea pasada. Una buena perspectiva nos brinda el ítem **Indicaciones Generales** de esa pregunta en la tarea

Indicaciones generales

Solo para que lo tengan en cuenta, este problema se puede resolver por fuerza bruta. Esto es, extraer cada nodo y luego contar el número de componentes que genera. La idea es que este no sea el acercamiento que ustedes utilicen, pues para grafos grandes su ejecución será muy costosa. De hecho, será del orden de $\mathcal{O}(V(V+E))$.

¿Cómo nos podemos aproximar a resolver el problema de buena manera entonces? Será **DFS** el algoritmo base que debes usar para esta entrega, gracias a lo cual podremos lograr una complejidad base del orden $\mathcal{O}(V+E)$, bastante mejor que el caso anterior.

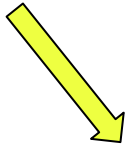
HINT: Entender la idea de las aristas de retorno o BackEdges dentro del contexto de un DFS.

DFS - EJERCICIO - **P3 I2 2018-1**

Buscaremos modificar DFS tal que pueda detectar aristas hacia atrás (BackEdge)

Por qué?

Una vez hemos recorrido un grafo dirigido a través de la estrategia DFS hay 2 posibilidades de que un nodo sea punto de articulación



Raíz del árbol DFS es un nodo de articulación.

Nodo no raíz árbol DFS es un nodo de articulación.

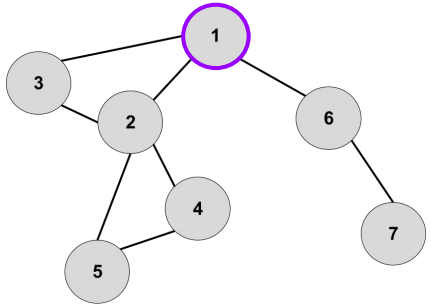
DFS - EJERCICIO - **P3 I2 2018-1**

Raíz del árbol DFS es un nodo de articulación.

Nodo **u** no raíz árbol DFS es un nodo de articulación.

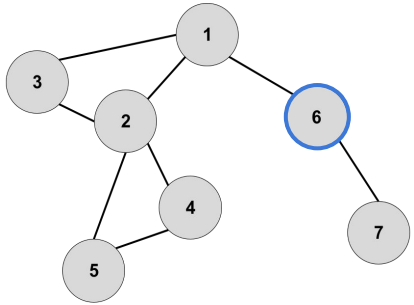
Cuándo ocurre?

Raíz tiene más de un hijo



Ojo que este grafo no es el árbol DFS

Todos los descendientes de **u** no cuentan con un camino al antecesor (padre) de **u**



DFS - EJERCICIO - **P3 I2 2018-1**

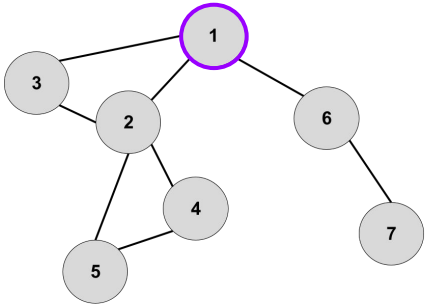
Raíz del árbol DFS es un nodo de articulación.

Raíz tiene más de un hijo

Por qué la estrategia funciona?

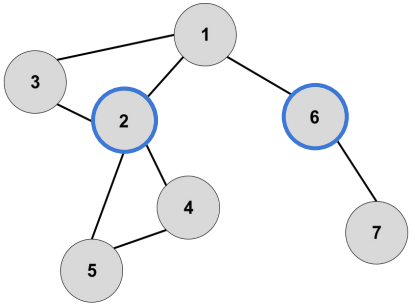
Nodo **u** no raíz árbol DFS es un nodo de articulación.

Todos los descendientes de **u** no cuentan con un camino al antecesor (padre) de **u**



Caso especial: Nodo 4. Tiene más de un hijo, pero removerlo no afecta

Así, la **estrategia completa** es Raíz tiene más de un hijo + su remoción aumenta el número de componentes conexas



Ojo que este grafo no es el árbol DFS

DFS - EJERCICIO - **P3 I2 2018-1**

Raíz del árbol DFS es un nodo de articulación.

Contar vecinos + contar componentes conexas luego de extraerlo

Cómo implementarlo?

Nodo **u** no raíz árbol DFS es un nodo de articulación.

Agregar a nodos el atributo
u.low = min{ u.disc, w.low }

dis => tiempo descubrimiento de u
w => algún nodo descendiente

Punto de articulación serán los que tienen u.low menor que el de sus descendientes directos

DFS - EJERCICIO - **P3 I2 2018-1**

Pongamos más atención a u.low

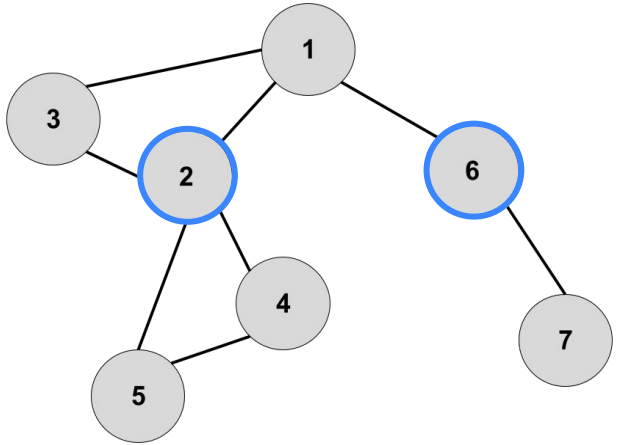
Nodo **u** no raíz árbol DFS es un nodo de articulación.

Agregar a nodos el atributo
u.low = min{ u.disc, w.low }

dis => tiempo descubrimiento de u
w => algún nodo descendiente

Punto de articulación serán los que tienen u.low menor que el de sus descendientes directos (hay que hacer una comparación)

Supongamos que iniciamos DFS desde u = 1



Si para nodo u, u.disc <= w.low, entonces no hay forma de llegar antes a los nodos descendientes de u. Por lo tanto, u es el único punto de entrada a nodos w* (i.e. u es un punto de articulación)

DFS - EJERCICIO - **P3 I2 2018-1**

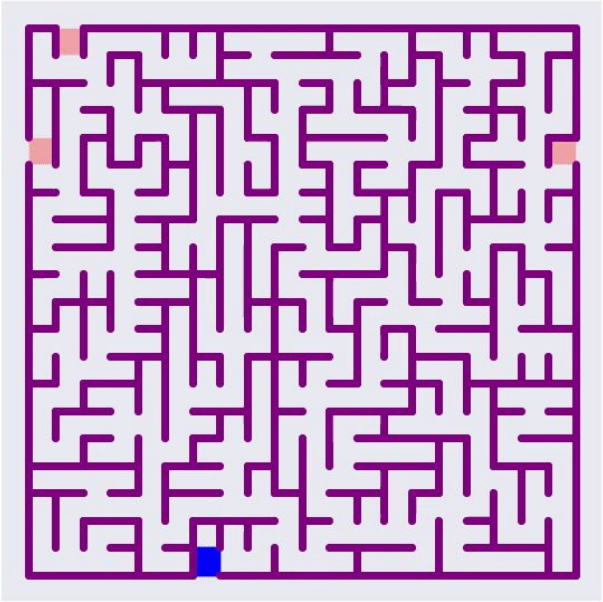
Resumen

1. Ejecutar DFS que registra tiempos de descubrimiento y *low* (paso completo en $O(V+E)$)
2. Revisar cada vértice en el árbol DFS generado y comparar *low* con sus descendientes. Agregar a una lista aquellos vértices que no cumplan la condición buscada (paso completo en $O(V+E)$).
3. Revisar cuántos hijos tiene la raíz del árbol y agregarla si tiene más de uno (paso completo en $O(1)$).

Por lo tanto la complejidad => O(V+E)

BACKTRACKING

Backtracking



Backtracking

6	2	3	7	1	8	9	4	5
4	5	9	2	3	6	1	8	7
8	7	1		9		3	2	6
9		4		7		2	5	1
7	1	8	9	5	2	6	3	4
2	6					7	9	8
5				8	7	4	1	2
1				6		8	7	3
3						5	6	9

¿Por qué Backtracking?

- Problemas de decisión: Búsqueda de una solución factible.
- Problemas de optimización: Búsqueda de la mejor solución.
- Problema de enumeración: Búsqueda de todas las soluciones posibles.

Backtracking

X = Variables

D = Dominio

R = Restricciones

```
is solvable(X, D, R):
    if X = ∅, return true
    x ← alguna variable de X
    for v ∈ D_x:
        if x = v viola R, continue
        x ← v
        if is solvable(X − {x}, D, R):
            return true
        x ← ∅
    return false
```

Backtracking

¿Se puede mejorar este algoritmo?

Backtracking

Hay tres mejoras posibles:

- Podas
- Propagación
- Heurísticas

Poda

Se deducen restricciones a partir de las restricciones o asignaciones anteriores que pueden ser agregadas al problema.

En otras palabras, estamos podando parte del conjunto de caminos a soluciones posibles.

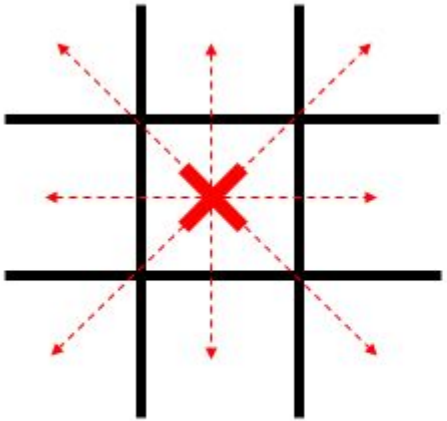
```
is solvable(X, D, R):
    if X = ∅, return true
    x ← alguna variable de X
    for v ∈ D_x:
        if x = v no es válida, continue
        x ← v
        if is solvable(X − {x}, D, R):
            return true
        x ← ∅
    return false
```

Propagación

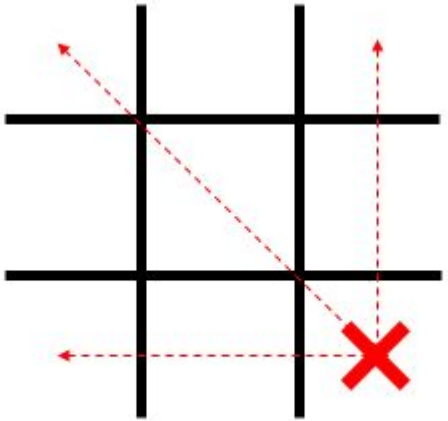
Cuando a la variable se le asigna un valor, se puede propagar esta información para luego poder reducir el dominio de valores de otras variables.

```
is solvable(X, D, R):
    if X = ∅, return true
    x ← alguna variable de X
    for v ∈ D_x:
        if x = v viola R, continue
        x ← v, propagar
        if is solvable(X − {x}, D, R):
            return true
        x ← ∅ , propagar
    return false
```

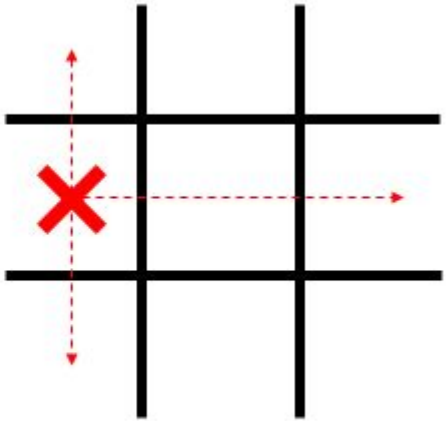
Heurística



4 ways to win the game



3 ways to win the game



2 ways to win the game

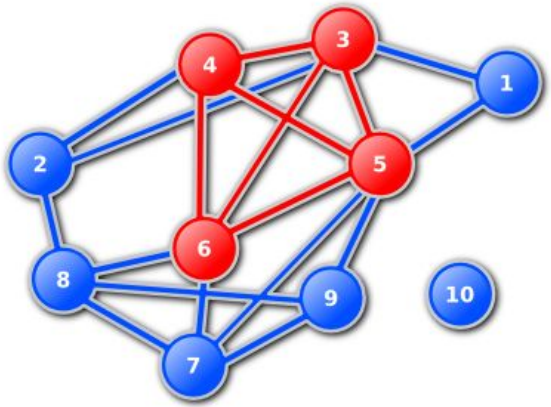
Heurística

Una heurística es una aproximación al mejor criterio para abordar un problema.

```
is solvable(X, D, R):
    if X = ∅, return true
    x ← la mejor variable de X
    for v ∈ D_x, de mejor a peor:
        if x = v viola R, continue
        x ← v
        if is solvable(X − {x}, D, R):
            return true
        x ← ∅
    return false
```

Ejercicio 4 - Backtracking

Sea $G(V, E)$ un grafo no direccional y $C \subseteq V$ un subconjunto de sus vértices. Se dice que C es un *k-clique* si $|C| = k$ y todos vértices de C están conectados con todos los otros vértices de C.



Ejercicio 4 - Backtracking

Sea $G(V, E)$ un grafo no direccional y $C \subseteq V$ un subconjunto de sus vértices. Se dice que C es un *k-clique* si $|C| = k$ y todos vértices de C están conectados con todos los otros vértices de C.

Dado un grafo cualquiera $G(V, E)$ y un número k, queremos determinar si existe un *k-clique* dentro de G. Esto se puede resolver usando *backtracking*.

a) [2pts.] Describe la modelación requerida para aplicar *backtracking* a este problema: explica cuál es el conjunto de **variables**, cuáles son sus **dominios**, y describe en palabras cuáles son las **restricciones** sobre los valores que pueden tomar dichas variables.

Recordar que al ser no direccional, si $(v, v') \in E \rightarrow (v', v) \in E$. A continuación, se plantean dos posibles formas de solucionar el problema:

Ejercicio 4 - Backtracking

Propuesta 1:

Las **variables** son nodos pertenecientes al k-clique.

El **dominio** son los posibles vértices que pueden ir asociados al nodo.

Las **restricciones** son que cada vértice que se escoge del dominio tiene que estar conectado a todos los otros nodos que se han escogido anteriormente.

X = nodos 1, …, k
D = V
R = {(v, v') ϵ E ∀ v' ϵ C}

```
is_solvable(X, D, C):
    si X = {} return True
    x ← alguna variable de X
    para v ϵ D:
        si v no cumple R, continuar
        si is_solvable(X - {x}, D - {v}, C ∪ {x}):
            retornar True
        C = C - {x}
    retornar False
```

Luego, podemos llamar a is_solvable({1, …, k}, D, {}).

Ejercicio 4 - Backtracking

Propuesta 2:

Las **variables** son los vértices de G.

El **dominio** es binario: 1 si el vértice está presente en el k-clique y 0 si no.

Las **restricciones** son que el vértice asignado como presente en el k-clique (con valor 1 en la variable) tiene que estar conectados a todos los otros vértices que han sido asignados como presente en el k-clique, y el número de vértices asignados como presentes en el grafo debe ser igual a k.

$X = V$
$D = \{0, 1\}$
$R = \{(v, v') \in E \ \forall \ v' \in C\}$

```
is_solvable(X, D, C, k):
        si k = 0 return True
        si X = {} return False
        x ← alguna variable de X
        para v ∈ D:
                si v = 1:
                        si v no cumple R, continuar
                        si is_solvable(X - {x}, D, C ∪ {x}, k - 1):
                                retornar True
                        C = C - {x}
                si v = 0:
                        si is_solvable(X - {x}, D, C, k):
                                retornar True

        retornar False
```

Luego, podemos llamar a is_solvable({V, D, {}, k).

Ejercicio 4 - Backtracking

b) [2 pts.] Propón una **poda** y explica la modelación a nivel de código requerida para implementarla eficientemente.

Propuesta 1

Si el vértice que estamos revisando tiene un número de aristas tiene menos de k - 1 aristas que se conectan con él, es imposible que éste pertenezca al k-clique. Para implementarla, podemos preprocesar el grafo, agregándole a cada nodo una variable que determina la cantidad de aristas que tiene hacia otros nodos (Que toma un tiempo O(|E|). Luego, basta acceder a esta variable y si es menor a k - 1, se detiene la ejecución.

Ejercicio 4 - Backtracking

b) [2 pts.] Propón una **poda** y explica la modelación a nivel de código requerida para implementarla eficientemente.

Propuesta 2

Si quedan menos variables a asignar, que el valor de k, podemos terminar esa ejecución ya que es imposible agregar suficientes elementos a C para que pertenezcan al k-clique. Para esto, podemos llevar un contador de cuántas variables nos quedan por asignar, y cuántos elementos llevamos en nuestro k-clique. Si la cantidad de variables que nos queda por asignar es menor a (k - (N° elementos asignados que llevamos en k-clique)), detenemos la ejecución.

Ejercicio 4 - Backtracking

c) [2 pts.] Propón una **heurística para el orden de las variables** y explica la modelación a nivel de código requerida para implementarla eficientemente.

Propuesta 1

Se puede ordenar el dominio en función de la cantidad de aristas que tienen las variables. De esta manera, es más probable que al asignar a la variable su valor, éste pertenezca al k-clique. El procedimiento es similar a la heurística anterior: podemos preprocesar el grafo, agregándole a cada nodo una variable que determina la cantidad de aristas que tiene hacia otros nodos (Que toma un tiempo O(|E|). Luego, basta con ordenar este arreglo de vértices en función de la cantidad de aristas hacia otros nodos, mediante algún algoritmo eficiente de los que se han visto en clases (como MergeSort, QuickSort, etc.), y utilizar esto como el dominio de la función de backtracking

Ejercicio 4 - Backtracking

c) [2 pts.] Propón una **heurística para el orden de las variables** y explica la modelación a nivel de código requerida para implementarla eficientemente.

Propuesta 2

Podemos ordenar los vértices de mayor a menor en función de la cantidad de aristas que poseen. De esta manera, es más probable que éstos sean parte de algún k-clique. Para esto, podemos preprocesar el grafo, agregándole a cada nodo una variable que determina la cantidad de aristas que tiene hacia otros nodos (Que toma un tiempo O(|E|). Luego, basta con ordenar este arreglo de vértices en función de la cantidad de aristas hacia otros nodos, mediante algún algoritmo eficiente de los que se han visto en clases (como MergeSort, QuickSort, etc.)

PROGRAMACIÓN DINÁMICA

Explicación YT

Programación Dinámica (PD)

Idea general: La solución al problema original se puede obtener a partir de la resolución de **subproblemas.**

En los problemas de PD hay que tener claro: (Recordemos el problema de la mochila)

- Cómo definir los **subproblemas:** Mochila con menor o igual capacidad y menos objetos para elegir
- **Casos base:** Casos que se escapan del problema, por ejemplo cuando no quedan objetos para elegir o la mochila excede su capacidad.

Teniendo claro los anterior, se plantea:

- **Ecuación de recurrencia**
- **Estructura de datos** para guardar las soluciones (ej: Matriz[i][j])

Programación Dinámica (PD)

$$DP[i][w] = \max\{DP[i-1][w], \text{valor}[i] + DP[i-1][w - \text{peso}[i]]\}$$

Valor máximo considerando los primeros i objetos y una capacidad w

Subproblema donde no se considera el objeto i

Subproblema donde se considero el objeto i, entonces hay menos capacidad

Formas de implementación:

- **Algoritmo Recursivo:** Parte del problema original y hace llamados recursivos a problemas más pequeños (Mochila con toda su capacidad disponible y todos los objetos para elegir)

- **Algoritmo Iterativo:** Parte de un problema más pequeño y va iterando hasta llegar al problema original (Mochila con capacidad 1, y solo se puede elegir el primer objeto)

Ejercicio PD

I. (3pts) El canal DCC TV tiene un nuevo programa llamado EDD donde los participantes tienen la chance de concursar y ganar dinero en premios. El juego consiste en un triángulo de pelotas, donde cada pelota tiene impreso un número íntegro, tal como se muestra en la figura.

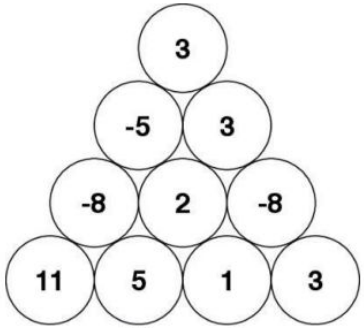


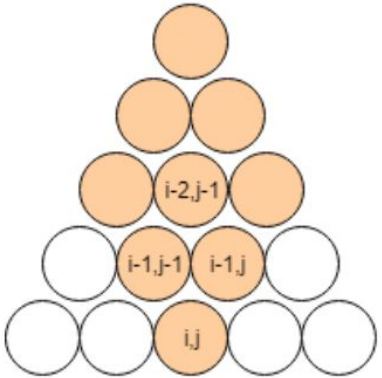
Figura 1

El jugador puede elegir una pelota de la pirámide y sacarla o no sacar ninguna. En caso de que saque una pelota, se quedará con los puntos de esa pelota y **todas las que estén arriba de ella**. En caso de que saque ninguna, se quedará con 0 puntos. En la figura 1, para sacar la pelota 1, se necesitan sacar también las pelotas 2, -8, -5, 3 y 3 por lo que el valor de sacar la pelota 1 sería -4. El presentador está preocupado por el máximo premio que se pueda llevar un concursante, por lo que se te pide ayuda a ti para resolver este problema.

Esta pirámide se representa como arreglo de arreglos M donde M[0] es el nivel superior y M[n] es el nivel de más abajo. Cada nivel M[i] tiene i+1 elementos.

Ejercicio PD

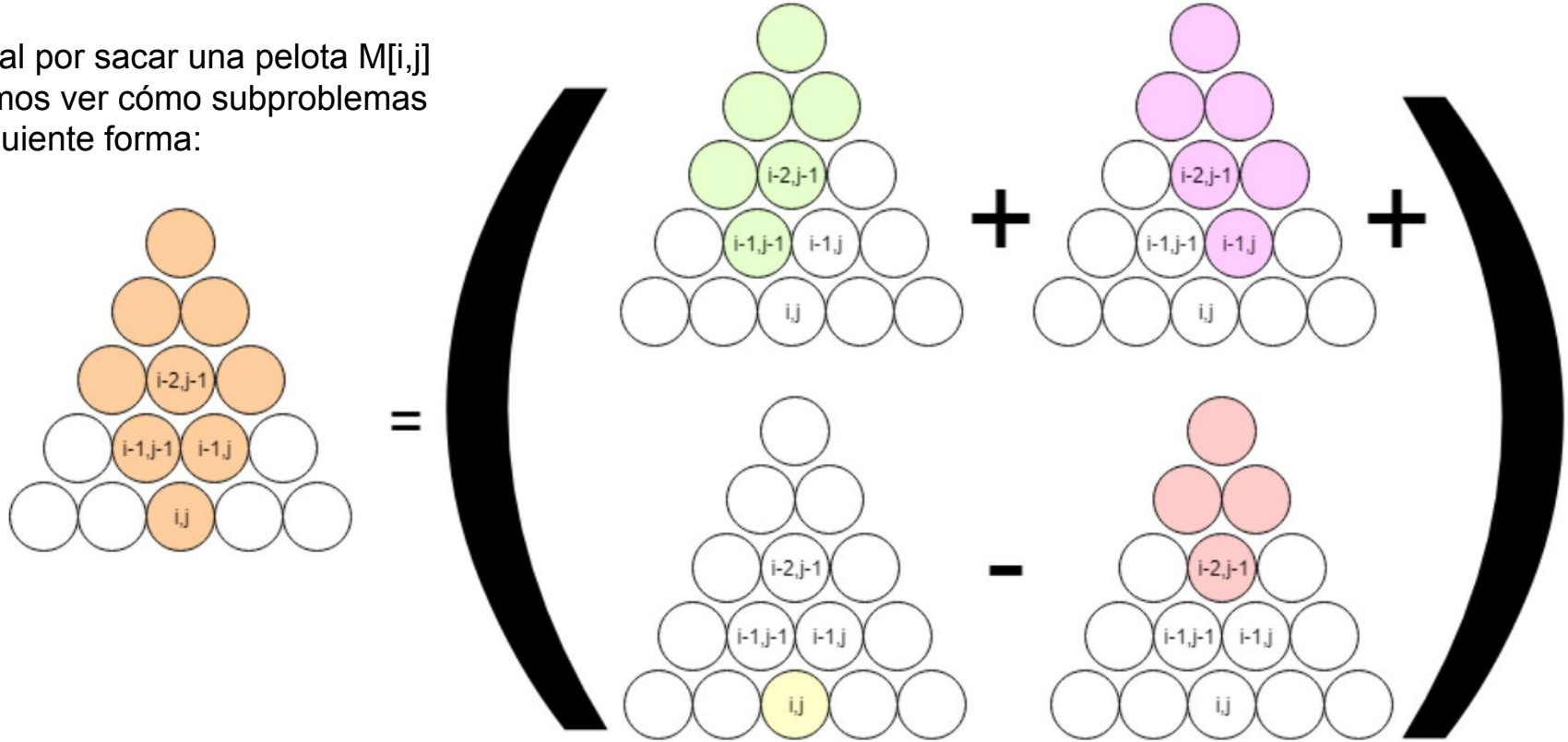
a) Haz la ecuación de recurrencia que permite calcular el premio obtenido de sacar una pelota específica: *P(i, j, M)* donde la pelota elegida es M[i][j].



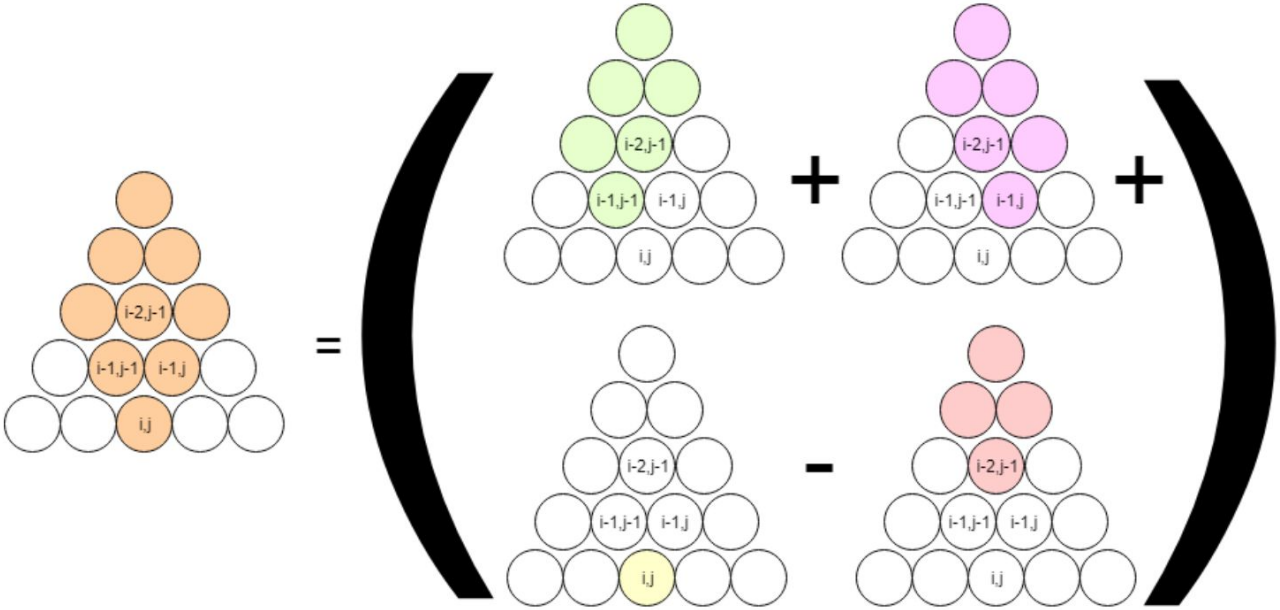
0,0	-	-	-	-
1,0	1,1	-	-	-
2,0	2,1	2,2	-	-
3,0	3,1	3,2	3,3	-
4,0	4,1	4,2	4,3	4,4

Ejercicio PD

Valor total por sacar una pelota M[i,j] lo podemos ver cómo subproblemas de la siguiente forma:

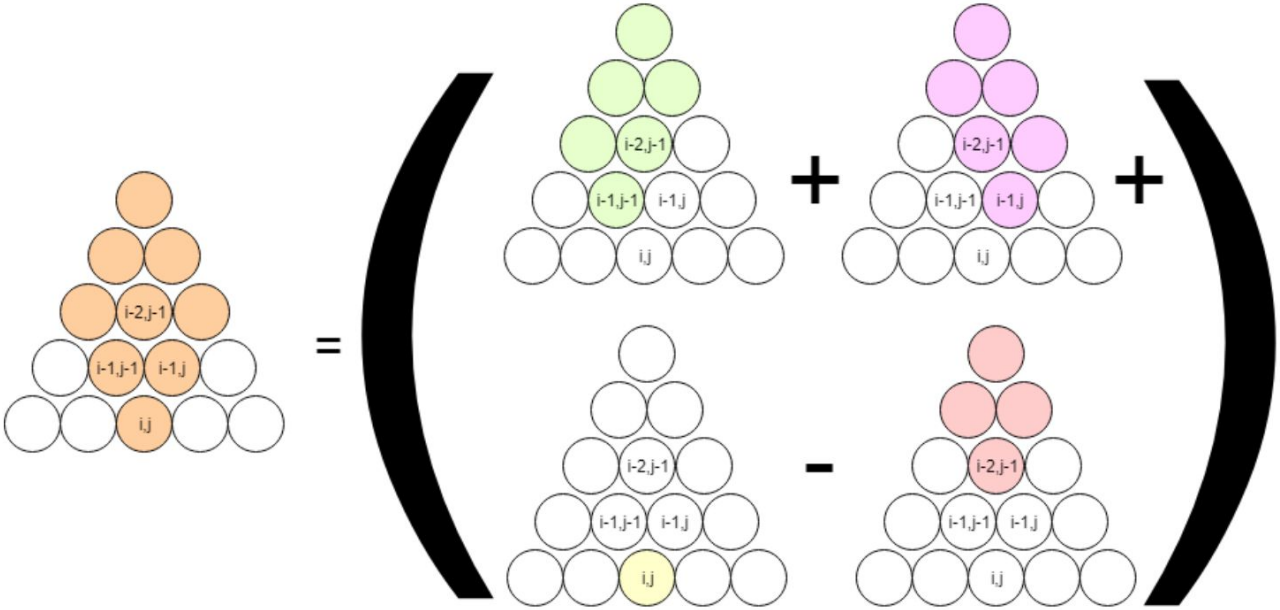


Ejercicio PD



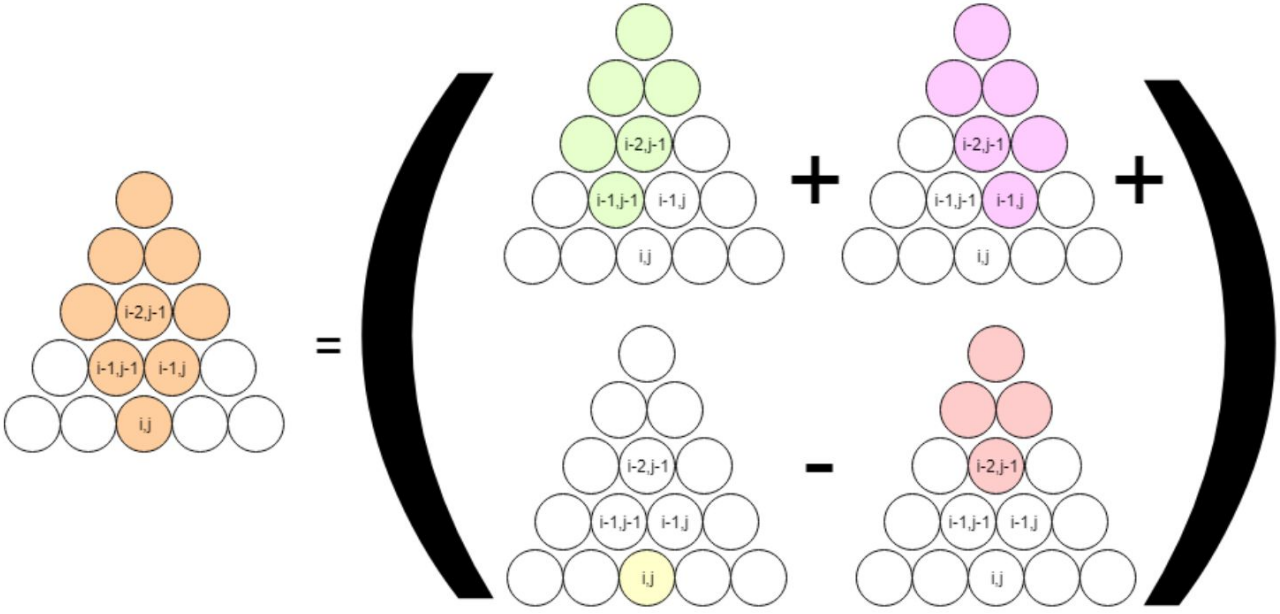
$$p(i,j,M) = \begin{cases} 0 & , \quad i,j < 0 \; o \; j > i \\ M[i][j] + p(i-1,j-1,M) + p(i-1,j,M) - p(i-2,j-1,M), & else \end{cases}$$

Ejercicio PD



$$p(i,j,M) = \begin{cases} 0 & , \quad i,j < 0 \; o \; j > i \\ M[i][j] + p(i-1,j-1,M) + p(i-1,j,M) - p(i-2,j-1,M), & else \end{cases}$$

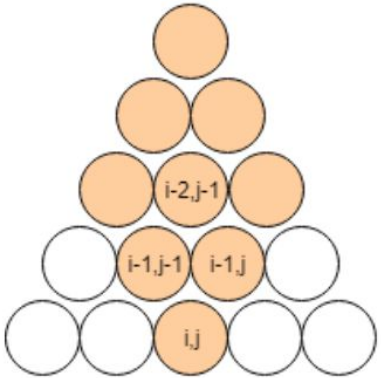
Ejercicio PD



Caso Base

$$p(i,j,M) = \begin{cases} 0, & i,j < 0 \; o \; j > i \\ M[i][j] + p(i-1,j-1,M) + p(i-1,j,M) - p(i-2,j-1,M), & else \end{cases}$$

Ejercicio PD



0,0	-	-	-	-
1,0	1,1	-	-	-
2,0	2,1	2,2	-	-
3,0	3,1	3,2	3,3	-
4,0	4,1	4,2	4,3	4,4

Caso Base

$$p(i,j,M) = \begin{cases} 0, & i,j < 0 \ o \ j > i \\ M[i][j] + p(i-1,j-1,M) + p(i-1,j,M) - p(i-2,j-1,M), & else \end{cases}$$

Ejercicio PD

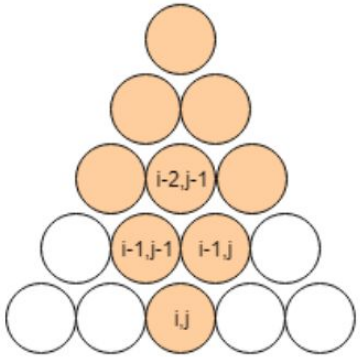
b) Crea una solución iterativa de este problema en la cual se calcula el premio para toda pelota de la pirámide (Versión bottom-up).

Ejercicio PD

b) Crea una solución iterativa de este problema en la cual se calcula el premio para toda pelota de la pirámide (Versión bottom-up).

Respuesta: [1.5 pts] Para hacer una respuesta iterativa para toda la pirámide debo crear una tabla R que contenga las respuestas e implementarlo de la siguiente forma:

```
premios(M):
        R = Tabla de las mismas dimensiones de M
        for i = 0..n:
                for j = 0..i:
                        R[i][j] = M[i][j]
                        if i > 0:
                                if j > 0:
                                        R[i][j] += R[i-1][j-1]
                                if j < i:
                                        R[i][j] +=R[i-1][j]
                                if i - 2 >= 0 and j > 0:
                                        R[i][j] -=R[i-2][j-1]
        return R
```



0,0	-	-	-	-
1,0	1,1	-	-	-
2,0	2,1	2,2	-	-
3,0	3,1	3,2	3,3	-
4,0	4,1	4,2	4,3	4,4

TIPS

- Backtracking: Comúnmente ejercicios de ***aplicación.***
  - Plantear restricciones y dominio del problema
  - Generar función isSolvable y Solve

- Tablas de hash: Comúnmente, preguntas de ***análisis***
  - Dada una ineficiencia en un sistema computacional, detectar el origen del problema.
  - Considerando contexto del problema, proponer solución