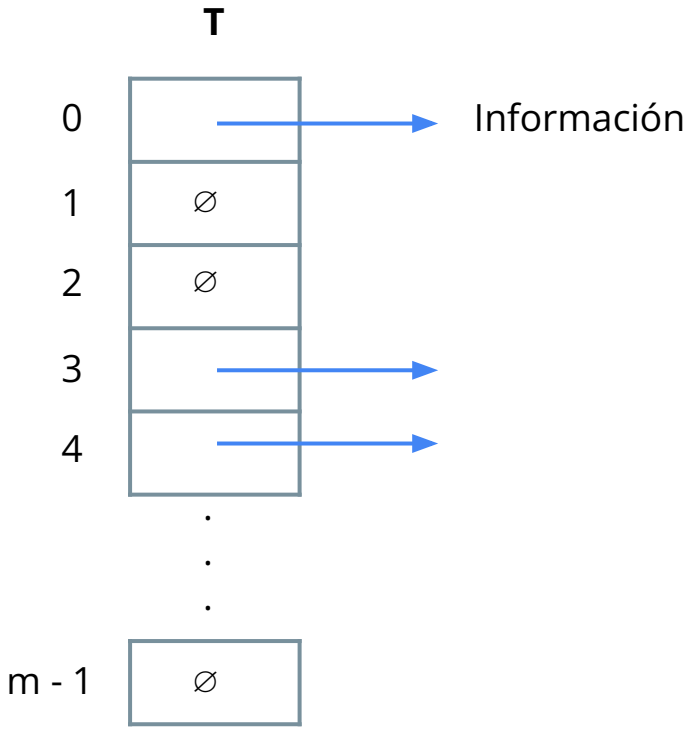
Repaso I2

Hashing - Backtracking - Greedy - DP

Tabla de Hash



T [k] = ∅ **no está**

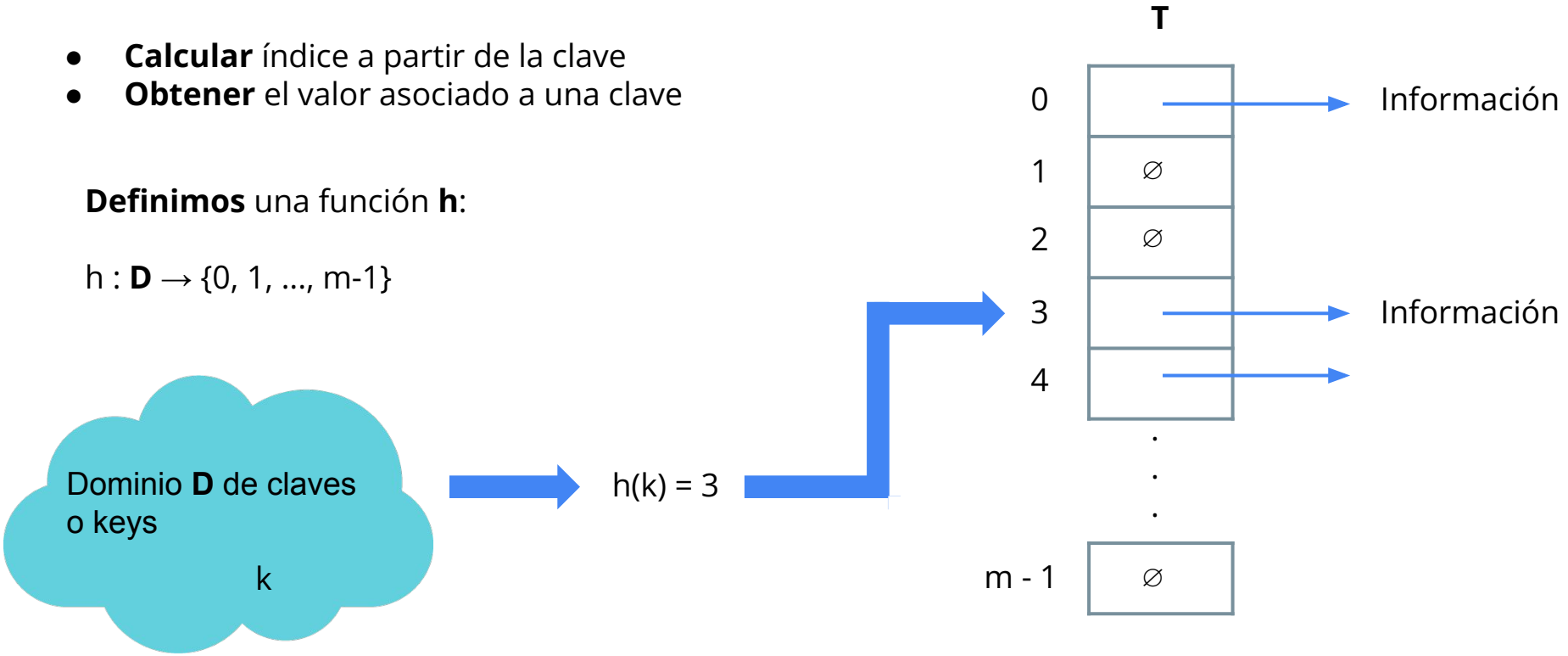
T [k] != ∅ **está**

Función de Hash

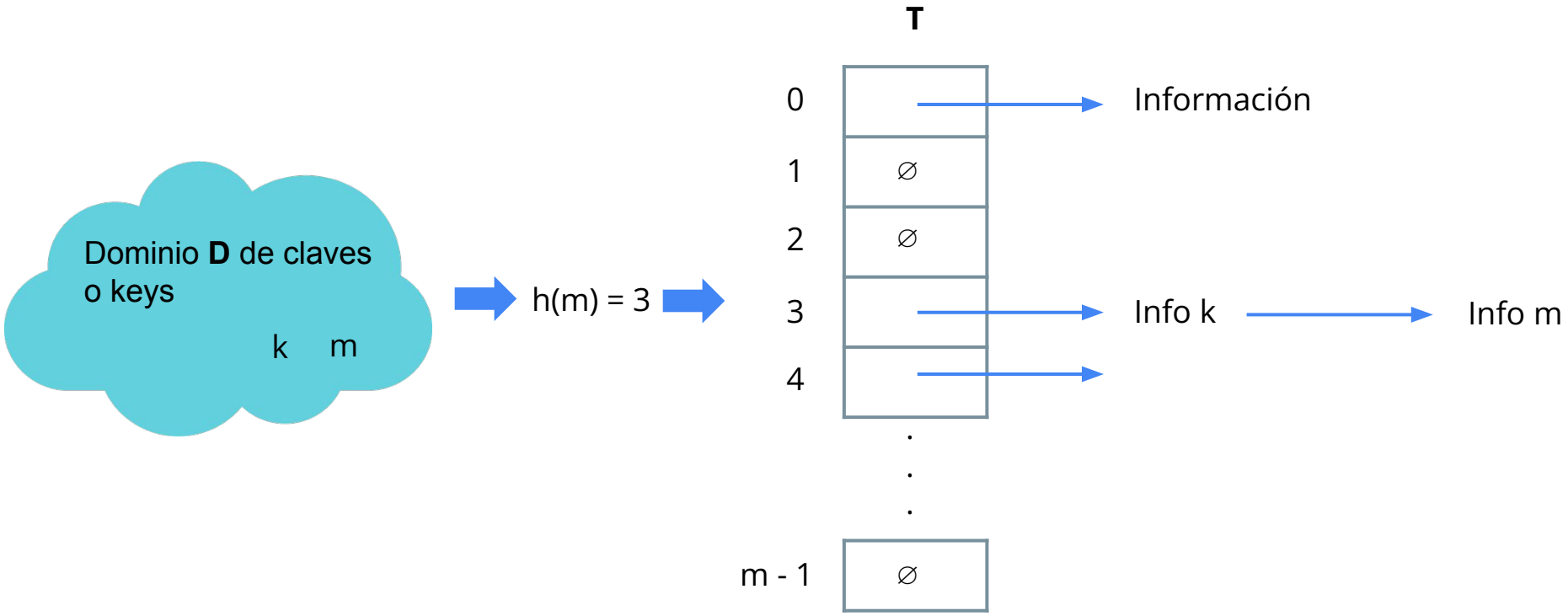
- **Calcular** índice a partir de la clave
- **Obtener** el valor asociado a una clave

Definimos una función **h**:

h : **D** → {0, 1, ..., m-1}



Encadenamiento



Ejercicio 1. Pizzeria M97

La pizzería intergaláctica M87 sirve pizzas de todos los incontables sabores existentes en todos los multiversos, y necesita ayuda para hacer más eficiente la atención de los millones de pedidos que recibe por segundo.

Los pedidos funcionan de la siguiente manera:

- Una persona solicita una pizza del sabor que haya escogido y da su nombre. Su pedido se agrega al sistema.
- Cuando la pizza está lista, se llama por el altavoz a la persona que haya pedido ese sabor hace más tiempo, y, una vez entregada la pizza, se borra ese pedido del sistema.

Explica cómo usar tablas de hash para llevar a cabo este proceso eficientemente. ¿Qué esquema de resolución de colisiones debería usarse y por qué? ¿Qué es lo que se guarda en la tabla?

Ejercicio 1. Pizzeria M97

- Explica cómo usar tablas de hash para llevar a cabo este proceso eficientemente.
- ¿Qué esquema de resolución de colisiones debería usarse y por qué?
- ¿Qué es lo que se guarda en la tabla?

Ejercicio 1. Pizzeria M97

- Explica cómo usar tablas de hash para llevar a cabo este proceso eficientemente.

Ejercicio 1. Pizzeria M97

- Explica cómo usar tablas de hash para llevar a cabo este proceso eficientemente.

- Para eso usamos una **tabla de hash** que nos permita guardar múltiples *values* para un mismo key. En este caso key corresponde al tipo de pizza y un value corresponde al nombre de la persona que lo pidió.

Ejercicio 1. Pizzeria M97

- ¿Qué esquema de resolución de colisiones debería usarse y por qué?

Ejercicio 1. Pizzeria M97

- ¿Qué esquema de resolución de colisiones debería usarse y por qué?
- Los *values* de un mismo key se deben guardar en una Cola (FIFO), de manera que agregar un nuevo *value* o extraer el siguiente sea O(1) y se atienda en orden de llegada. (Si deciden guardarlo en un Heap que ordena por orden de llegada, las operaciones son O(Log (n)).

Ejercicio 1. Pizzeria M97

- ¿Qué esquema de resolución de colisiones debería usarse y por qué?

- Los *values* de un mismo key se deben guardar en una Cola (FIFO), de manera que agregar un nuevo *value* o extraer el siguiente sea O(1) y se atienda en orden de llegada. (Si deciden guardarlo en un Heap que ordena por orden de llegada, las operaciones son O(Log (n)).

- La eliminación del pedido sale automática con el heap o la cola ya que obtener el siguiente elemento lo extrae de la estructura.

Ejercicio 1. Pizzeria M97

- ¿Qué es lo que se guarda en la tabla?

Ejercicio 1. Pizzeria M97

- ¿Qué es lo que se guarda en la tabla?
- En este caso key corresponde al tipo de pizza y un value corresponde al nombre de la persona que lo pidió.

Ejercicio 1. Pizzeria M97

- ¿Qué es lo que se guarda en la tabla?

- En este caso key corresponde al tipo de pizza y un value corresponde al nombre de la persona que lo pidió. Si varias personas pidieron la misma pizza, se guarda como value una lista ligada de los clientes que pidieron ese sabor de pizza.

- Pero como el dominio de las key es infinito, debemos ir despejando las celdas de la tabla cuando una key se queda sin values, ya que no tenemos memoria infinita. Para esto es necesario usar encadenamiento ya que permite eliminar keys de la tabla sin perjudicar el rendimiento de esta. De esta manera, en caso de que varios sabores de pizza sean *hasheados* a la misma celda de la tabla (colisión de sabores), las colas FIFO de los nombres de las personas que pidieron esas pizzas deberán ser encadenadas en una lista doblemente ligada que las contenga

Ejercicio 2. Función de Hash

- Función de hash: Sumar los digitos del numero repetidamente hasta que quede un numero.
- El caso del dígito 9 se tiene que h(9)= 0
- Con esto se espera con esto que el resultado sea probablemente correcto.

Consideración importante: Notar que el hash modulariza por mod 9

Ejercicio 2. Función de Hash

- Queremos multiplicar 2 número X e Y y verificar que se cumpla X*Y= Z. Para verificar esto, utilizamos una función de hash h(), tal que se cumpla que:

$$h(h(X)* h(Y)) = h(Z)$$

Ejercicio 2. Función de Hash

- Queremos multiplicar 2 número X e Y y verificar que se cumpla X*Y= Z. Para verificar esto, utilizamos una función de hash h(), tal que se cumpla que:

$$h(h(X)* h(Y)) = h(Z)$$

Ejercicio 3. Heaps + Hash

Una tienda quiere premiar a sus ***k*** clientes más rentables. Para ello cuenta con la lista de las ***n*** compras de los últimos años, en que cada compra es una tupla de la forma *(ID Cliente, Monto)*. La rentabilidad de un cliente es simplemente la suma de los montos de todas sus compras.
Explica cómo usar tablas de hash y heaps para resolver este problema en tiempo esperado —o promedio- $O(n + nlog(k))$.

El problema se separa en dos partes.

- Calcular la rentabilidad de cada cliente
- Buscar los *k* clientes más rentables

Ejercicio 3. Heaps + Hash

Calcular la rentabilidad de cada cliente

Queremos obtener el set de tuplas *(ID Cliente, Rentabilidad)*, donde la rentabilidad para el ID Cliente ***i*** es la suma de todos los montos de las tuplas de la forma *(i, Monto)*.

Para esto creamos una tabla de hash ***T*** donde se almacenan tuplas *(ID Cliente, Rentabilidad)*. Cada vez que se inserte un ID Cliente *i*:

- Si ya está en la tabla, se suma el monto recién insertado al monto guardado.
- Si no, se guarda en la tabla con el monto recién insertado como monto inicial.

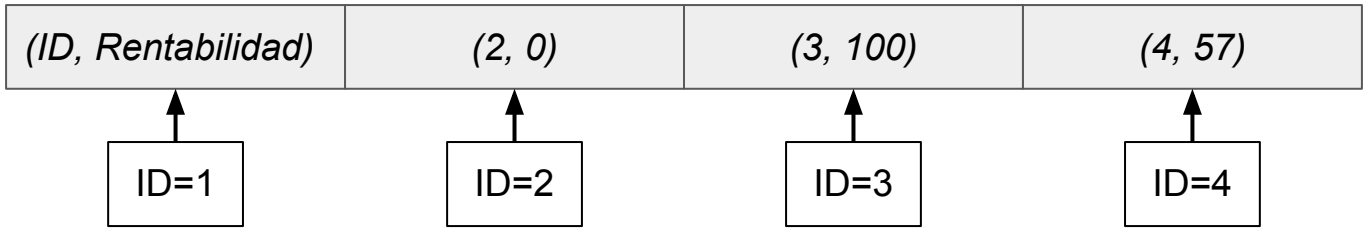
Ejercicio 3. Heaps + Hash

Calcular la rentabilidad de cada cliente

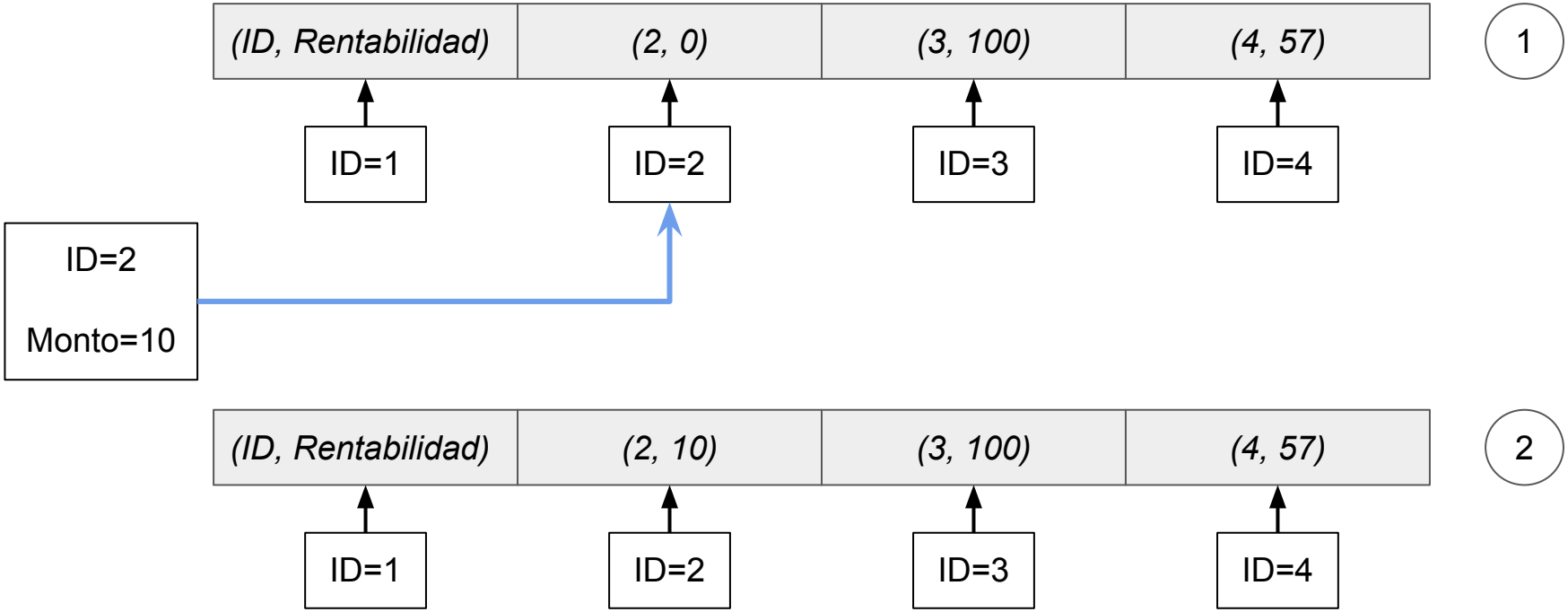
Queremos obtener el set de tuplas *(ID Cliente, Rentabilidad)*, donde la rentabilidad para el ID Cliente ***i*** es la suma de todos los montos de las tuplas de la forma *(i, Monto)*.

Para esto creamos una tabla de hash ***T*** donde se almacenan tuplas *(ID Cliente, Rentabilidad)*. Cada vez que se inserte un ID Cliente :

- Si ya está en la tabla, se suma el monto recién insertado al monto guardado.
- Si no, se guarda en la tabla con el monto recién insertado como monto inicial.



Ejercicio 3. Heaps + Hash



Ejercicio 3. Heaps + Hash

Calcular la rentabilidad de cada cliente

Queremos obtener el set de tuplas *(ID Cliente, Rentabilidad)*, donde la rentabilidad para el ID Cliente ***i*** es la suma de todos los montos de las tuplas de la forma *(i, Monto)*.

Para esto creamos una tabla de hash ***T*** donde se almacenan tuplas *(ID Cliente, Rentabilidad)*. Cada vez que se inserte un ID Cliente *i*:

- Si ya está en la tabla, se suma el monto recién insertado al monto guardado.
- Si no, se guarda en la tabla con el monto recién insertado como monto inicial.

Al hacer esto con todas las *n* tuplas hemos efectivamente encontrado la rentabilidad para cada cliente.

Ejercicio 3. Heaps + Hash

Calcular la rentabilidad de cada cliente

Queremos obtener el set de tuplas *(ID Cliente, Rentabilidad)*, donde la rentabilidad para el ID Cliente ***i*** es la suma de todos los montos de las tuplas de la forma *(i, Monto)*.

Para esto creamos una tabla de hash ***T*** donde se almacenan tuplas *(ID Cliente, Rentabilidad)*. Cada vez que se inserte un ID Cliente *i*:

- Si ya está en la tabla, se suma el monto recién insertado al monto guardado.
- Si no, se guarda en la tabla con el monto recién insertado como monto inicial.

Al hacer esto con todas las *n* tuplas hemos efectivamente encontrado la rentabilidad para cada cliente.

La inserción en esta tabla tiene tiempo esperado ***O(1)*** como se ha visto en clases. Como se realizan ***n*** inserciones, esta parte tiene tiempo esperado ***O(n)***.

Ejercicio 3. Heaps + Hash

Buscar los *k* clientes más rentables

Sea *m* el total de clientes distintos. Creamos un min-heap de tamaño *k* que contendrá los *k* clientes más rentables que hemos visto hasta el momento. De este modo, la raíz del heap es el cliente **menos rentable** de los ***k* más rentables** encontrados hasta el momento.

Iteramos sobre las tuplas en ***T***, insertando en el heap hasta llenarlo, usando la rentabilidad como prioridad. Luego de haber llenado el heap, seguimos iterando sobre *T* .

Ejercicio 3. Heaps + Hash

Buscar los *k* clientes más rentables

Sea *m* el total de clientes distintos. Creamos un min-heap de tamaño *k* que contendrá los *k* clientes más rentables que hemos visto hasta el momento. De este modo, la raíz del heap es el cliente **menos rentable** de los ***k* más rentables** encontrados hasta el momento.

Iteramos sobre las tuplas en ***T***, insertando en el heap hasta llenarlo, usando la rentabilidad como prioridad. Luego de haber llenado el heap, seguimos iterando sobre *T* .

Para cada elemento que veamos que sea mayor a la raíz, lo intercambiamos con esta. Luego hacemos *shift-down* para restaurar la propiedad del heap. Esto mantiene la propiedad descrita en el párrafo anterior.

Ejercicio 3. Heaps + Hash

Buscar los *k* clientes más rentables

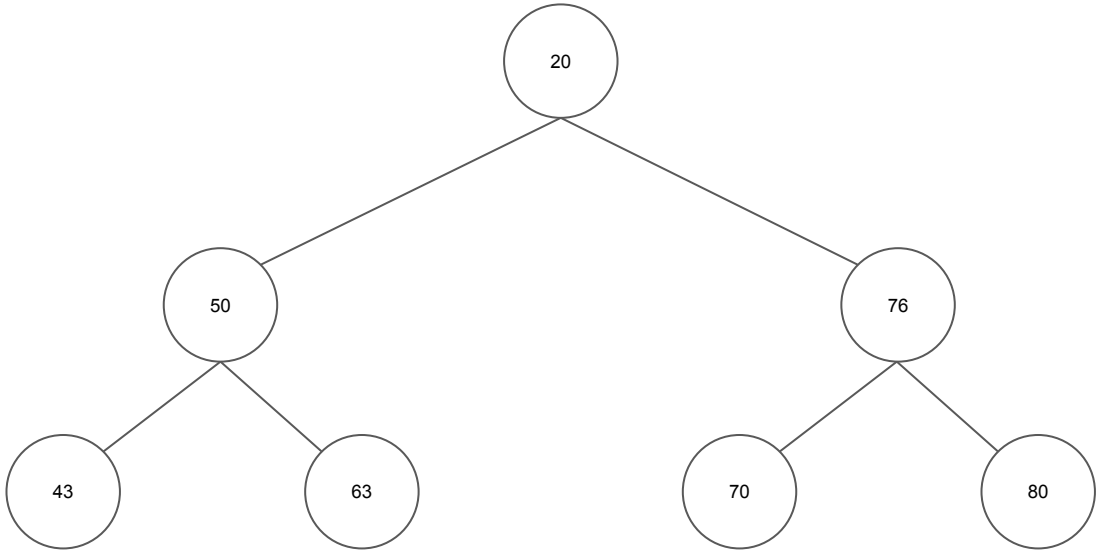
Sea *m* el total de clientes distintos. Creamos un min-heap de tamaño *k* que contendrá los *k* clientes más rentables que hemos visto hasta el momento. De este modo, la raíz del heap es el cliente **menos rentable** de los ***k* más rentables** encontrados hasta el momento.

Iteramos sobre las tuplas en ***T***, insertando en el heap hasta llenarlo, usando la rentabilidad como prioridad. Luego de haber llenado el heap, seguimos iterando sobre *T* .

Para cada elemento que veamos que sea mayor a la raíz, lo intercambiamos con esta. Luego hacemos *shift-down* para restaurar la propiedad del heap. Esto mantiene la propiedad descrita en el párrafo anterior.

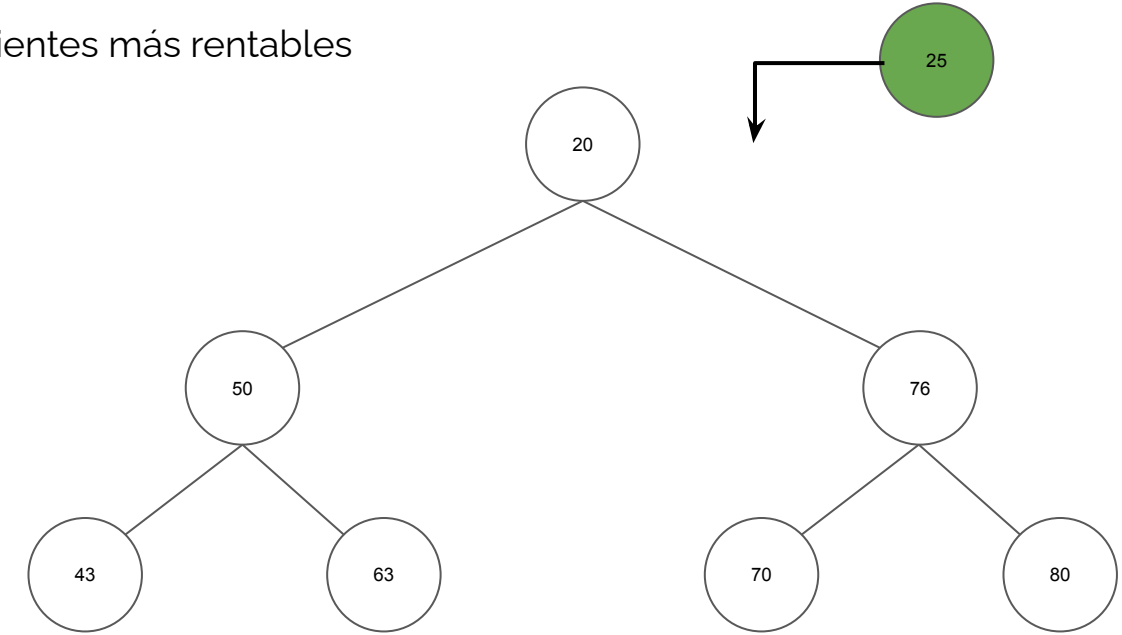
Ejercicio 3. Heaps + Hash

Buscar los *k* clientes más rentables



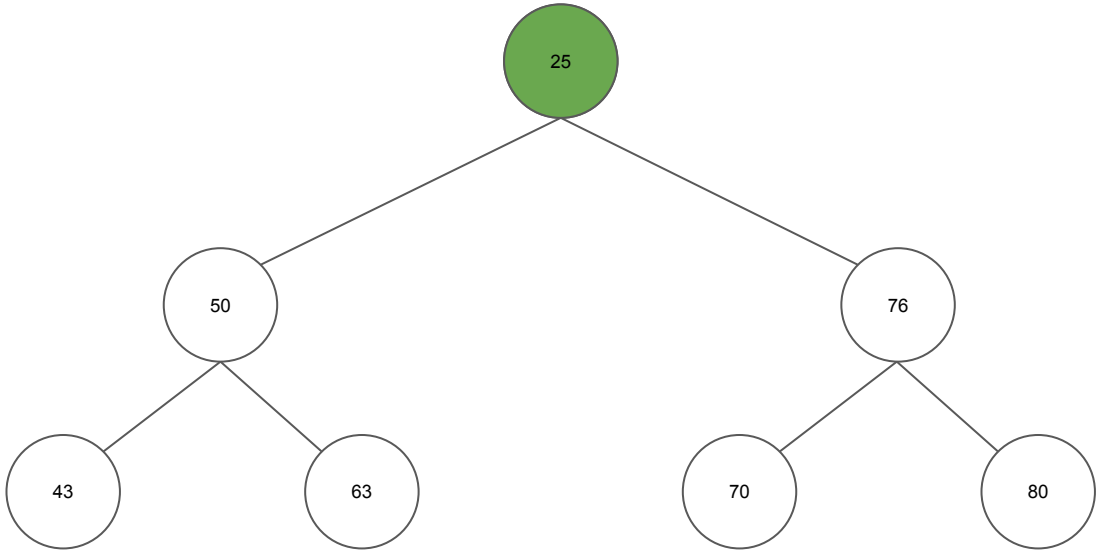
Ejercicio 3. Heaps + Hash

Buscar los *k* clientes más rentables



Ejercicio 3. Heaps + Hash

Buscar los *k* clientes más rentables

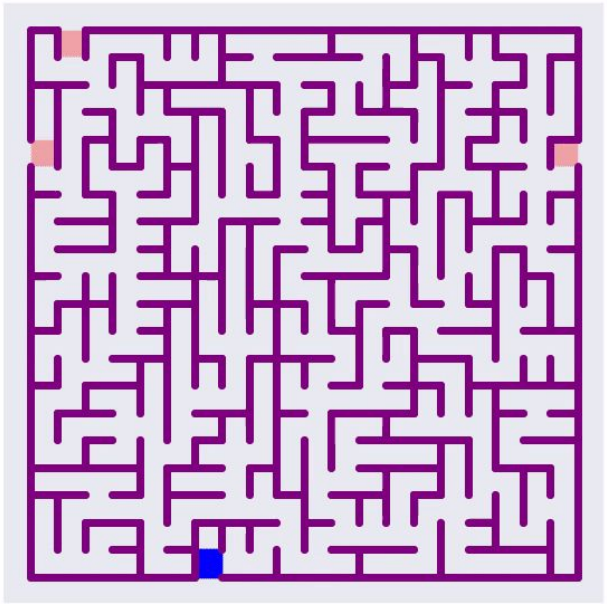


Ejercicio 3. Heaps + Hash

Buscar los *k* clientes más rentables

Cada inserción en el heap toma *O(log k)*. En el peor caso insertamos los *m* elementos en el heap, por lo que esta parte es *O(m log(k))*. Pero como en el peor caso *m=n*, esta parte es *O(n log k)*.

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

```
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.

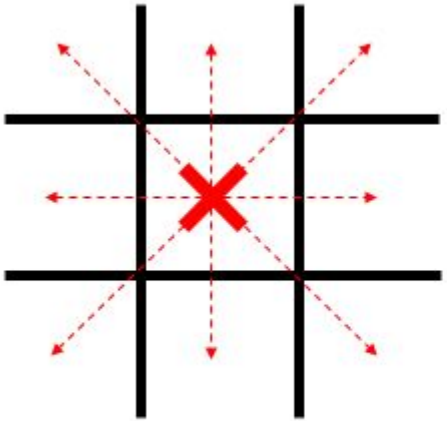
```
is solvable(X, D, R):
    if X = ∅, return true
    x ← alguna variable de X
    for v ∈ D_x:
        if x = v no es válida, continue
        x ← v
        if is solvable(X − {x}, D, R):
            return true
        x ← ∅
    return false
```

Propagación

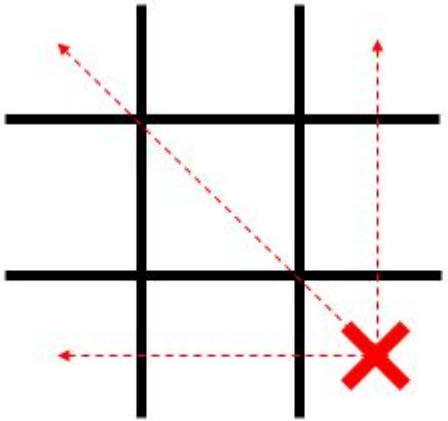
Cuando a variable se le asigna un valor, se puede propagar esta información para luego poder reducir el dominio de valores de otras variables.

```
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
```

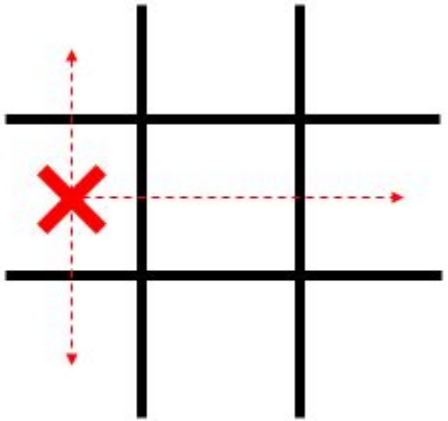
Heurística



4 ways to win the game



3 ways to win the game



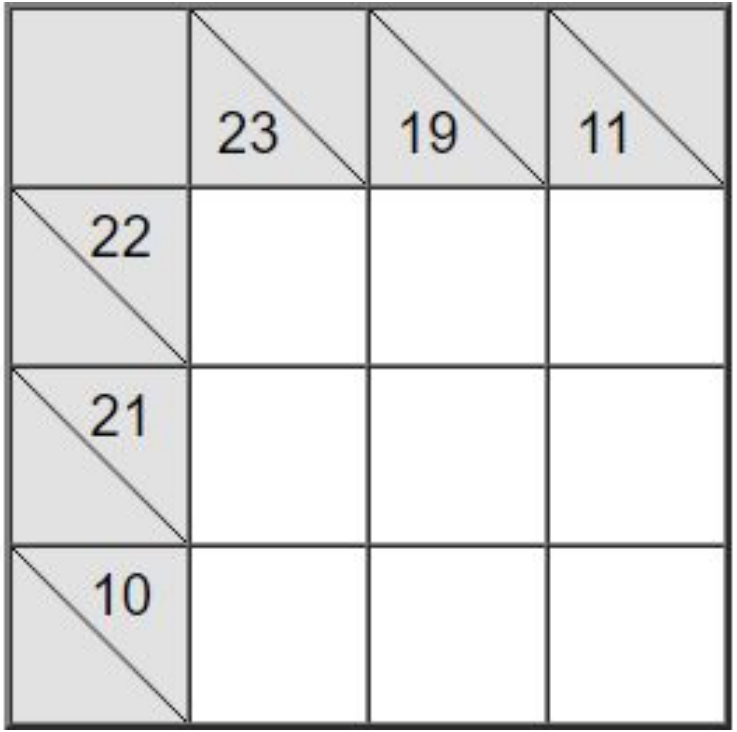
2 ways to win the game

Heurística

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

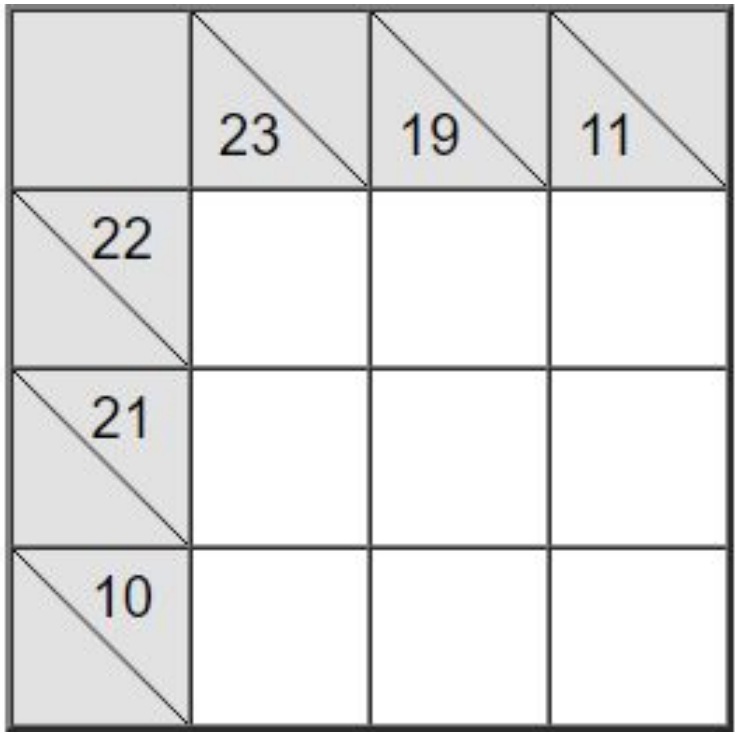
```
is solvable(X, D, R):
    if X = ∅, return true
    x ← la mejor variable de X
    for v ∈ D_x, de mejor a peor:
        if x = v viola R, continue
        x ← v
        if is solvable(X − {x}, D, R):
            return true
        x ← ∅
    return false
```

Ejercicio 1 - Kakuro



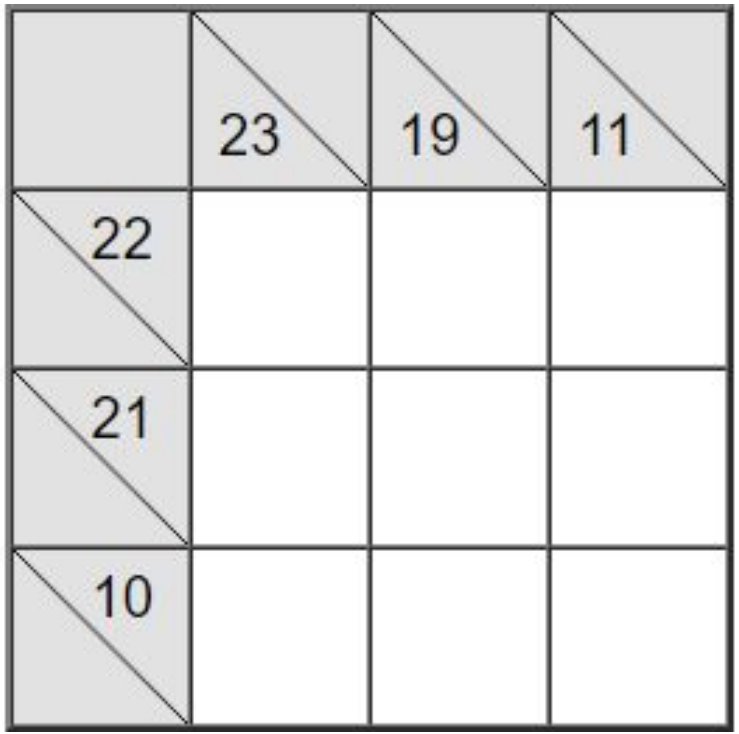
Ejercicio 1 - Kakuro

- Variables?
- Dominio?
- Restricciones?



Ejercicio 1 - Kakuro

- Variables: Valores de celdas
- Dominio: {1, …, MAX}
- Restricciones: Las filas y columnas suman los valores esperados. Cada número es usado una sola vez por suma.



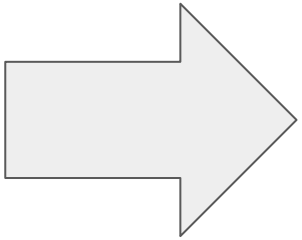
Ejercicio 1 - Kakuro

- Variables: Valores de celdas
- Dominio: {1, …, MAX}
- Restricciones: Las filas y columnas suman los valores esperados. Cada número es usado una sola vez por suma.

	23	19	11
22	9	7	6
21	8	9	4
10	6	3	1

Ejercicio 2 - Corredores: 2021-2 l2

- N corredores.
- Se registra el tiempo del primer corredor en llegar.
- El tiempo del resto de los corredores se registra en función del primer corredor.



- Lista T: De largo N, que registra los tiempos de los corredores, partiendo del tiempo 0.
- Lista D: de largo N(N-1)/2, que registra la diferencia de tiempos entre cada par de corredor. Ordenadas de menor a mayor.

Ejercicio 2 - Corredores: 2021-2 l2

Problema: Se perdió las lista T, y tenemos que volver a generarla a partir de la lista D.

Importante de notar:

- El dominio de las lista son los valores D
- Reconocer la condición de término (el último elemento de D).
- Identificar el largo de la lista T.
- Consideración: valores de D no repetidos
- Tiempos no repetidos.

Ejercicio 2 - Corredores: 2021-2 l2

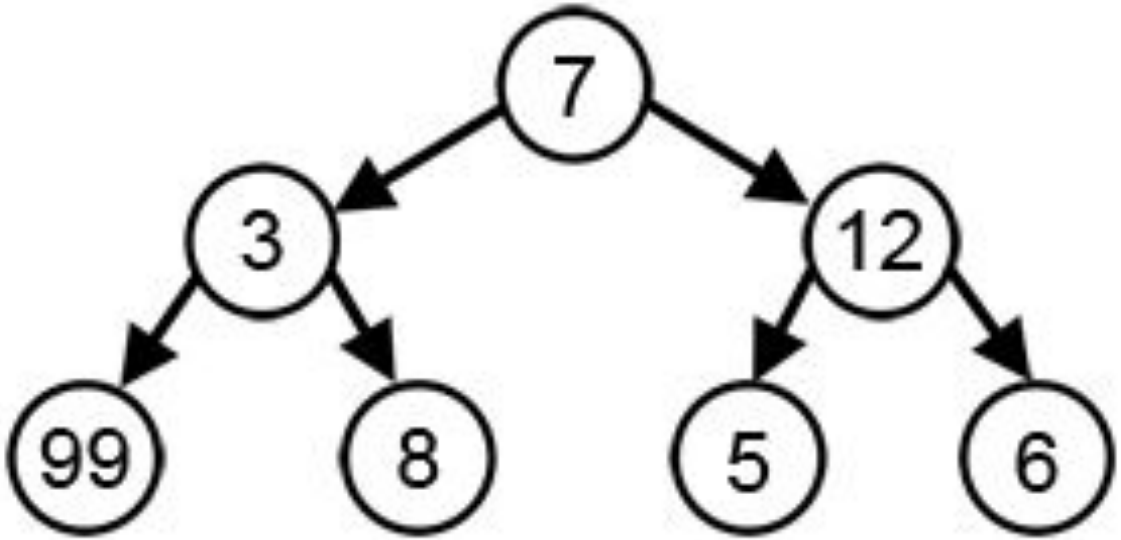
Pseudocódigo:

```
solve(X, D):
    if X[-1] == D[-1]:
        print(X)
        return True
    u= 0
    for value in D:

        if valid_value(value):
            D.pop(u)
            X.append(value)
            if solve(X, D):
                return True
            X.pop()
            D= funcion_insert(D, value, u)
        u+=1
    return False
```

Greedy

Camino más “Costoso”



Greedy

- Se basa en ***tomar decisiones locales óptimas*** en cada paso con la esperanza de alcanzar una solución global óptima.

- Comienza con una solución vacía y agrega gradualmente elementos o toma decisiones que maximicen una métrica o función objetivo en cada paso.

- ***No considera ni revisa las decisiones tomadas anteriormente***, solo se enfoca en la decisión actual basada en la información disponible en ese momento.

- ***No garantiza una solución óptima global y puede quedar atrapado en mínimos locales*** o soluciones parcialmente satisfactorias. (Requiere de un problema con subestructura optima)

Greedy

- Se basa en ***tomar decisiones locales óptimas*** en cada paso con la esperanza de alcanzar una solución global óptima.

- Comienza con una solución vacía y agrega gradualmente elementos o toma decisiones que maximicen una métrica o función objetivo en cada paso.

- ***No considera ni revisa las decisiones tomadas anteriormente***, solo se enfoca en la decisión actual basada en la información disponible en ese momento.

- ***No garantiza una solución óptima global y puede quedar atrapado en mínimos locales*** o soluciones parcialmente satisfactorias. (Requiere de un problema con subestructura optima)

Problema corto

Estamos a final de semestre y quieres alcanzar a realizar la mayor cantidad de actividades antes de que el semestre cierre.

Supongamos que tienes un conjunto de actividades ***A***, cada una con un tiempo de inicio (ti) y un tiempo de finalización (tf).

El objetivo es seleccionar el conjunto máximo de actividades compatibles de manera que ninguna actividad se solape entre sí.

Problema corto

- ¿Por qué este problema es greedy?
- ¿Cómo lo soluciono de forma codiciosa?

Problema corto

Actividad 1: [Inicio: 1, Fin: 4]

Actividad 2: [Inicio: 3, Fin: 5]

Actividad 3: [Inicio: 0, Fin: 6]

Actividad 4: [Inicio: 5, Fin: 7]

Actividad 5: [Inicio: 3, Fin: 8]

Actividad 6: [Inicio: 5, Fin: 9]

Paso 1: Seleccionar la actividad 3, ya que es la que **finaliza primero**.

Paso 2: Descartar la actividad 1, ya que se solapa con la actividad 3.

Paso 3: Seleccionar la actividad 2, ya que es la que **finaliza primero después de la actividad 3**.

Paso 4: Descartar la actividad 4, ya que se solapa con la actividad 2.

Paso 5: Descartar la actividad 5, ya que se solapa con la actividad 2.

Paso 6: Seleccionar la actividad 6, ya que es la que finaliza primero después de la actividad 2.

Problema corto

```
ActivitySelection(A):
    Sort activities in non-decreasing order of finish time
    selectedActivities = [A[0]]
    lastSelectedActivity = A[0]

    for i = 1 to length(A) - 1:
        if A[i].start >= lastSelectedActivity.finish:
            selectedActivities.append(A[i])
            lastSelectedActivity = A[i]

    return selectedActivities
```

DP

- La programación dinámica es un enfoque de resolución de problemas que involucra ***dividir un problema en subproblemas más pequeños*** y resolverlos de manera independiente.

- Se caracteriza por ***almacenar y reutilizar los resultados de los subproblemas para evitar recalcularlos***, lo que mejora la eficiencia del algoritmo.

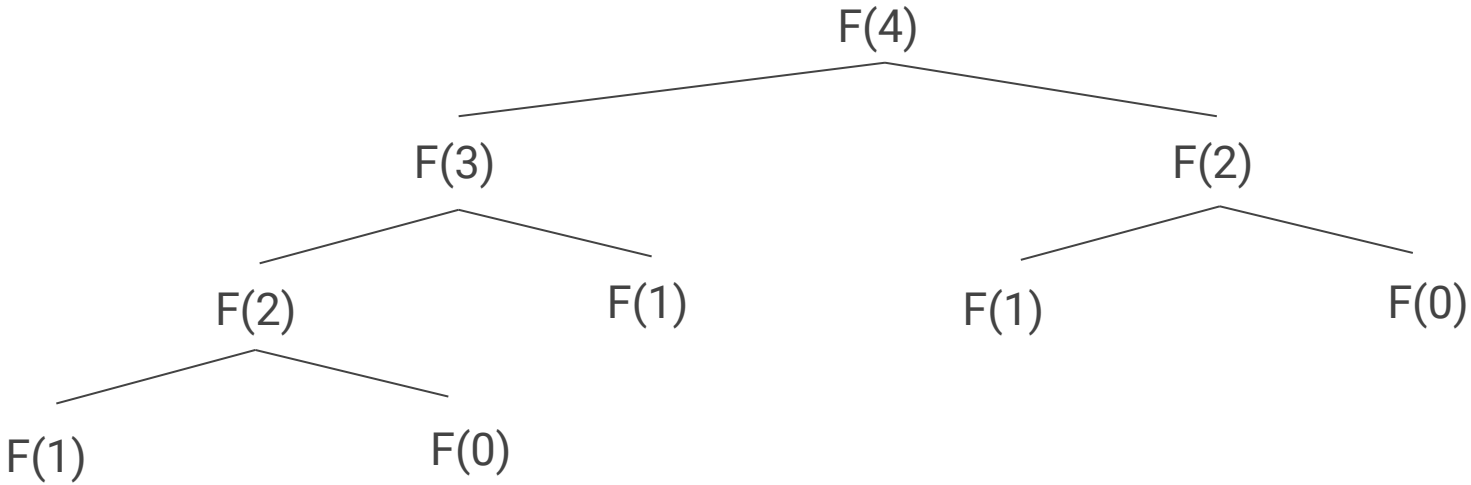
- La programación dinámica busca encontrar la ***solución óptima global combinando las soluciones óptimas de los subproblemas***. Es especialmente útil para problemas con superposición de subproblemas, donde los mismos subproblemas se resuelven repetidamente.

DP

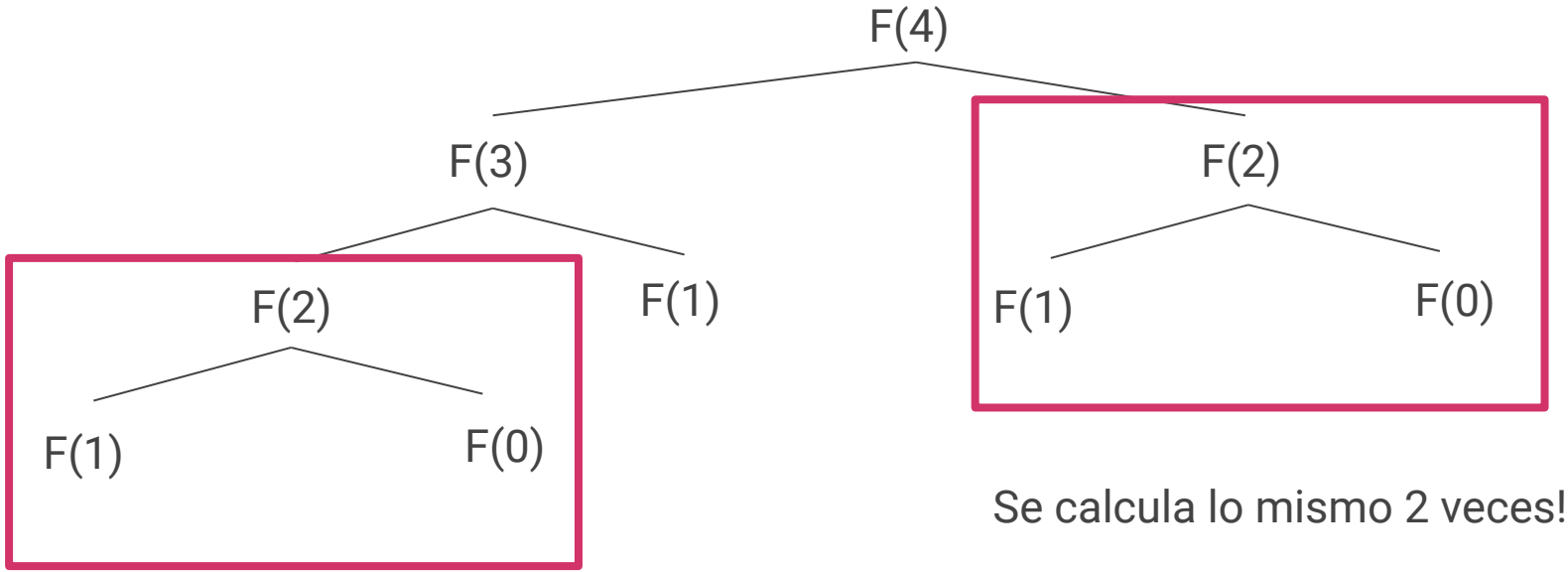
Problema: Calcular el n-ésimo número de Fibonacci.

1. **La subestructura óptima en este problema radica en que el n-ésimo número de Fibonacci se puede calcular utilizando los resultados de los dos números de Fibonacci anteriores (n-1 y n-2).**
2. **Por ejemplo, para calcular el quinto número de Fibonacci (n = 5), necesitamos conocer los resultados de los números de Fibonacci anteriores (n = 4 y n = 3).**
3. **Estos a su vez se calculan utilizando los resultados de los números de Fibonacci aún más anteriores (n = 3, n = 2, n = 2 y n = 1).**
4. **La solución óptima para calcular el quinto número de Fibonacci es combinar las soluciones óptimas para calcular el cuarto y tercer número de Fibonacci.**

DP



DP



Se calcula lo mismo 2 veces!

DP

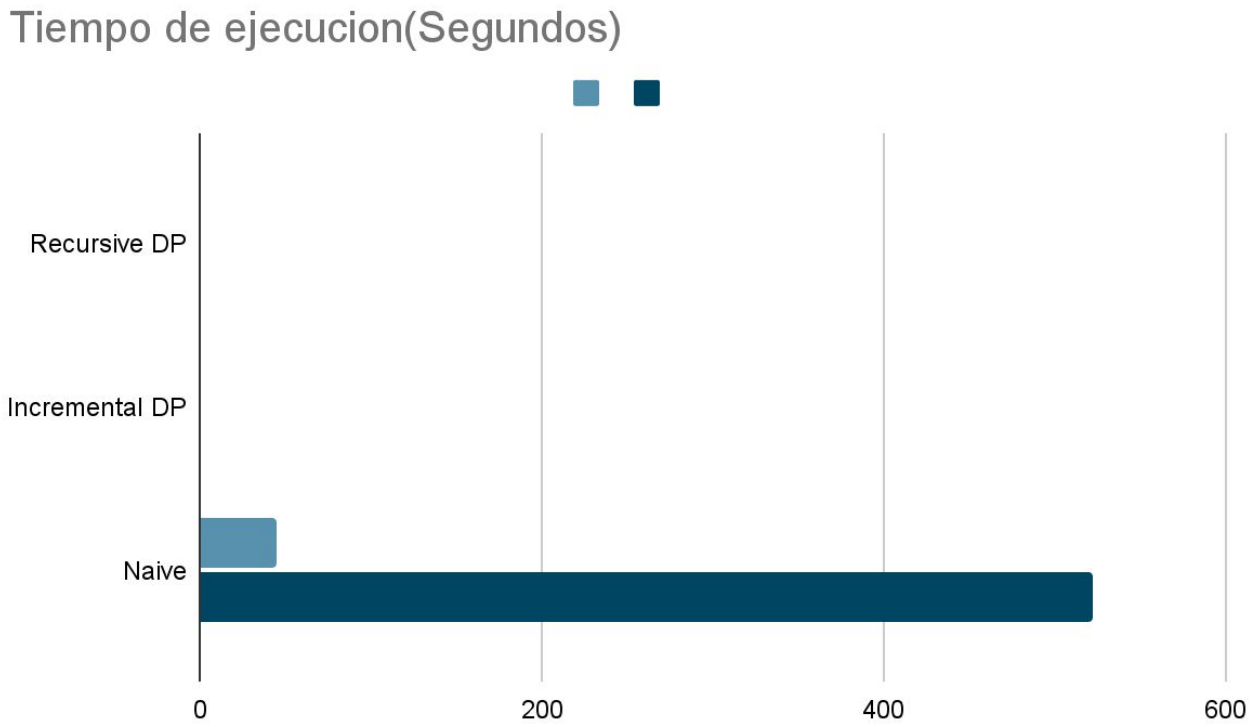
```
Fibonacci(n):
    if n <= 1:
        return n

    fib = [0, 1]

    for i = 2 to n:
        fib.append(fib[i-1] + fib[i-2])

    return fib[n]
```

DP


Tiempo de ejecucion(Segundos)
Recursive DP
Incremental DP
Naive
0
200
400
600

DP

Dado un conjunto de monedas con diferentes valores y un monto objetivo a devolver como vuelto, ¿cómo se puede encontrar la cantidad mínima de monedas necesarias para alcanzar el monto objetivo utilizando programación dinámica?

Objetivo:
Encontrar la cantidad mínima de monedas necesarias para alcanzar el monto objetivo de vuelto utilizando las monedas disponibles.

Restricciones:

- Pueden utilizarse múltiples monedas del mismo valor.
- El monto devuelto debe ser exacto
- No hay límite en la cantidad de monedas utilizadas.

DP

Dado un conjunto de monedas con diferentes valores y un monto objetivo a devolver como vuelto, ¿cómo se puede encontrar la cantidad mínima de monedas necesarias para alcanzar el monto objetivo utilizando programación dinámica?

Objetivo:
Encontrar la cantidad mínima de monedas necesarias para alcanzar el monto objetivo de vuelto utilizando las monedas disponibles.

Restricciones:

- Pueden utilizarse múltiples monedas del mismo valor.
- El monto devuelto debe ser exacto
- No hay límite en la cantidad de monedas utilizadas.

DP

1. Es subestructura optima? ¿Por qué?

El problema del vuelto es de subestructura óptima porque la solución óptima para calcular la cantidad mínima de monedas necesarias para un ***monto objetivo se puede obtener combinando las soluciones óptimas para montos más pequeños,*** es decir, se puede construir a partir de las soluciones óptimas de subproblemas.

DP

```
MinimumChange(coins, amount):
    n = length(coins)
    dp = [infinity] * (amount + 1)
    dp[0] = 0

    for i = 1 to amount:
        for j = 0 to n-1:
            if coins[j] <= i:
                dp[i] = min(dp[i], 1 + dp[i - coins[j]])

    return dp[amount]
```

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

TIPS

- Greedy: Comúnmente, detectar naturaleza del problema, o proveer contraejemplos
  - Dado problema, demostrar que posee estructura subóptima
  - Dada una estrategia/algoritmo, demostrar que no funciona (Contraejemplo), ó que funciona (Inducción)

- DP: Comúnmente, plantear ecuación de recurrencia / algoritmo
  - Dado problema, plantear la ecuación de recurrencia
  - Dada la ecuación de recurrencia plantear el algoritmo que resuelve el problema

- Backtracking
    - I2-2020-2: Determinar si un grafo es K-coloreable
    - C4-2019-2: Definir constraints, y dominio de un problema
    - I1-2017-2: Aplicar y describir algoritmo de backtracking que resuelve el problema

- Hash
    - I2-2020-2: Diseñar una tabla de hash
    - Ex-2019-1: Análisis de problema y propuestas de mejoras
    - I2-2018-2: Análisis de problema, y modelación de tabla de hash

Problemas recomendados para revisar

- Greedy
    - I2-2020-2: Determinar si un grafo es K-coloreable
    - Ex-2018-2: Demostrar correctitud de algoritmo greedy
    - I3-2013-1: Definir subestructura optima y proponer algoritmo

- DP
    - EX-2020-1: Describir ecuación de recurrencia y algoritmo
    - C6-2019-1: Definir recurrencia y diagramar arbol de ejecución
    - Ex-2015-2: Aplicar DP para generar algoritmo en problema mochila

Problemas recomendados para revisar