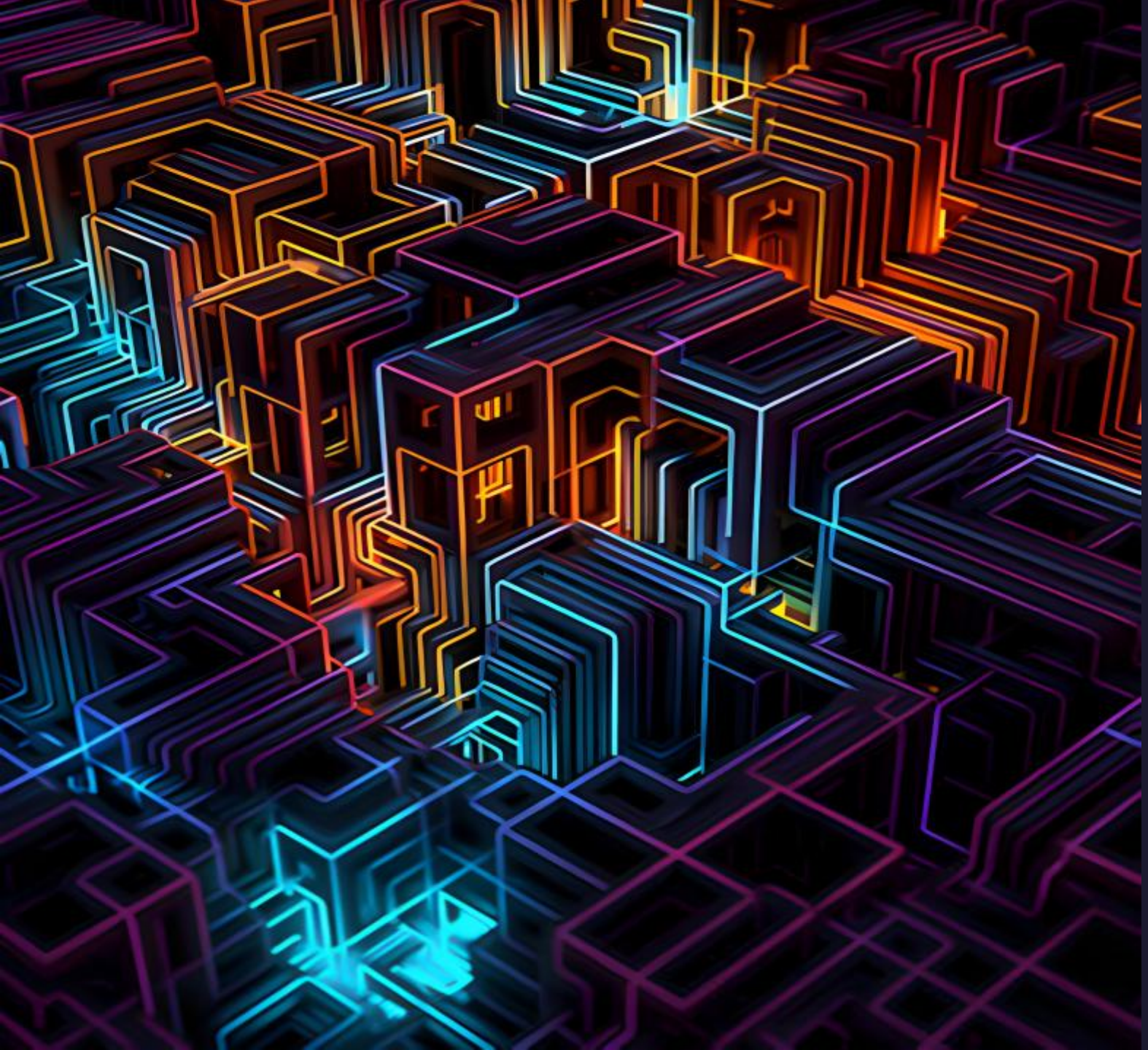


Técnicas de diseño de algoritmos: fuerza bruta

Análisis y diseño de algoritmos avanzados

Dra. Valentina Narváez Terán

Tecnológico de Monterrey

Ejemplo: cruce del puente

Estas cuatro personas deben cruzar un puente oscuro y frágil…

Velocidades:

Ana	1 min	Lola	5 min
Carlos	2 min	Paco	10 min

Reglas:

- Todos inician del mismo lado
- Pueden cruzar máximo dos personas a la vez
- Para cruzar, se necesita llevar la linterna, siempre
- Solo hay una linterna
- Quienes crucen juntos, van a la velocidad del mas lento

¿Cuál es el menor tiempo para que cruce el grupo completo y en que orden lo hacen?

Ejemplo: cruce del puente

Velocidades:

Ana	1 min	Lola	5 min
Carlos	2 min	Paco	10 min

Solución de costo $k = 17$

Cruzan	Costo
Carlos, Ana	2
Ana	1
Paco, Lola	10
Carlos	2
Ana, Carlos	2

Total: 17

Algoritmos de fuerza bruta

Los algoritmos de fuerza bruta computan todas las posibilidades

- Son algoritmos simples y directos
- A menudo, no muy eficientes

- Su simpleza puede ayudar a comprender el problema mejor, para después diseñar un algoritmo mejor

Algunos ejemplos de algoritmos de fuerza bruta que ya conoces

- Búsqueda secuencial
- Selection-sort y bubble-sort

Búsqueda secuencial

Búsqueda secuencial

Recorre los elementos uno a uno

No utiliza, ni requiere espacio extra (in-situ)

```
     ALGORITHM sequentialSearch(A, n, x)
1:   // Computes the index where value x is found in array A, or -1 if x is not in A
2:   // Input: Array A, size of the array n, value x
3:   // Output: Index of x in A, −1 otherwise
4:   index = −1
5:   for i = 0 to n
6:       if x = A[i] then
7:           index = i
8:           return index
9:       end if
10:  end for
11:  return index
```

A	1	6	8	2	4	3	5	9	7	10

$i = 0$ $\qquad$ $i = n - 1$

¿Cuántos pasos toma para $x = 3$?
¿Cuántos toma $x = 10$?

¿Cuál es la complejidad?

Búsqueda secuencial

Búsqueda secuencial

Recorre los elementos uno a uno

No utiliza, ni requiere espacio extra (in-situ)

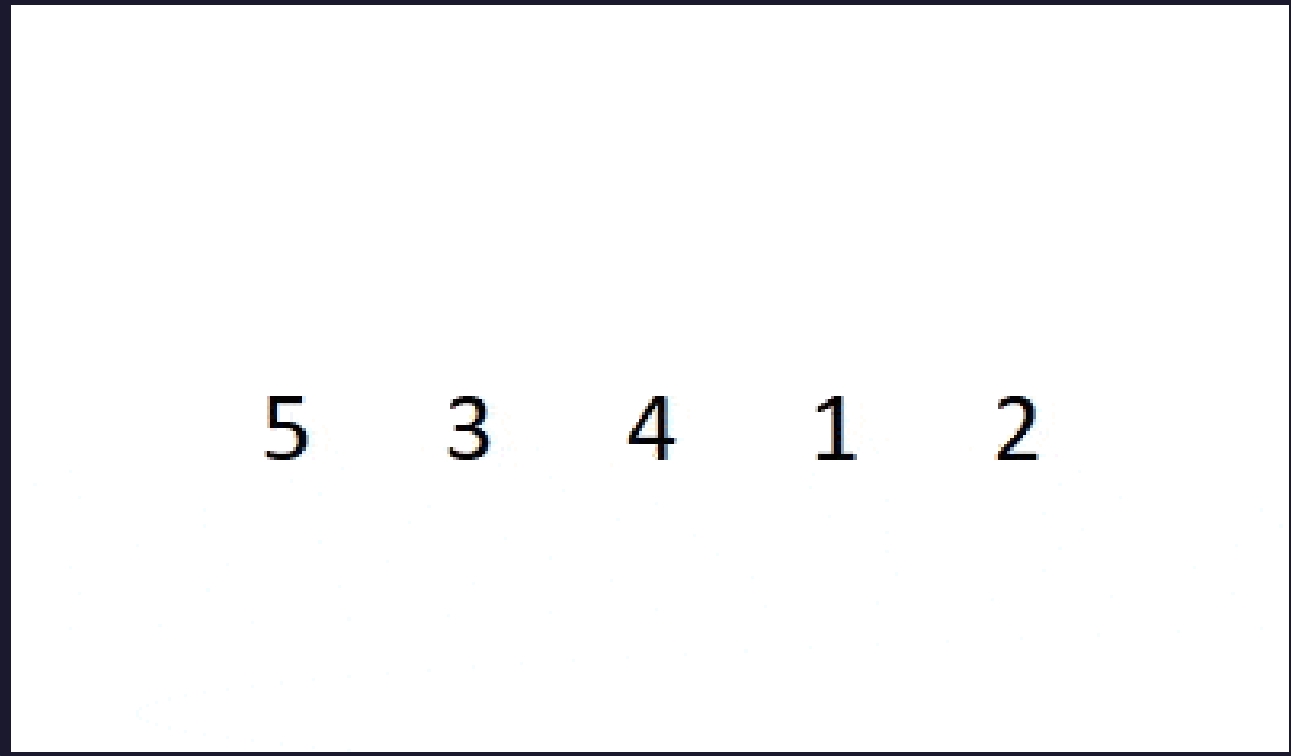
```
    ALGORITHM sequentialSearch(A, n, x)
1:  // Computes the index where value x is found in array A, or -1 if x is not in A
2:  // Input: Array A, size of the array n, value x
3:  // Output: Index of x in A, −1 otherwise
4:  index = −1
5:  for i = 0 to n
6:      if x = A[i] then
7:          index = i
8:          return index
9:      end if
10: end for
11: return index
```

Búsqueda secuencial es $O(n)$:
Máximo, toma n comparaciones encontrar o descartar x en A

Ordenamiento con selection-sort

Selección (Selection sort)

$O(?)$



Encuentra el elemento mínimo en la sección no ordenada, moviéndolo al inicio del arreglo

Lo repite para cada uno de los n elementos

Ordenamiento con selection-sort

Encuentra el elemento mínimo en la sección no ordenada.

Lo repite para cada uno de los n elementos.

```
     ALGORITHM selectionSort (A, n)
1:   // Input: Array A, de tamaño n
2:   // Output: A ordenado
3:
4:   for i = 0 to n − 2 do
5:       // Encuentra el mínimo en la sección desordenada
6:       indexMin = i
7:       for j = i + 1 to n − 1 do
8:           if A[j] < A[indexMin] then
9:               indexMin = j
10:          end if
11:      end for
12:      // Intercambia A[i] con A[indexMin]
13:      aux = A[indexMin]
14:      A[indexMin] = A[i]
15:      A[i] = aux
16:  end for
```

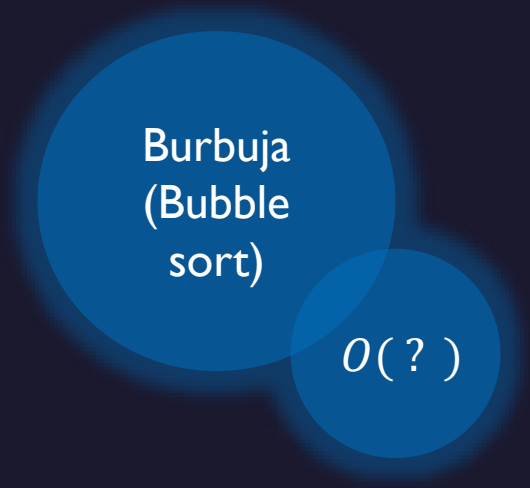
Ordenamiento con selection-sort

Selection-sort es $O(n^2)$

Encuentra n veces el mínimo, y cada vez toma hasta n pasos

```
     ALGORITHM selectionSort (A, n)
1:   // Input: Array A, de tamaño n
2:   // Output: A ordenado
3:
4:   for i = 0 to n − 2 do
5:       // Encuentra el mínimo en la sección desordenada
6:       indexMin = i
7:       for j = i + 1 to n − 1 do
8:           if A[j] < A[indexMin] then
9:               indexMin = j
10:          end if
11:      end for
12:      // Intercambia A[i] con A[indexMin]
13:      aux = A[indexMin]
14:      A[indexMin] = A[i]
15:      A[i] = aux
16:  end for
```

Ordenamiento con bubble-sort

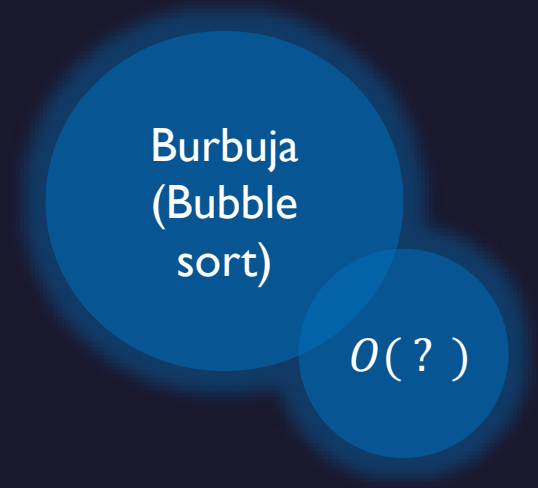


Realiza n ciclos de comparaciones entre cada par de posiciones continuas

Al final del i-esímo ciclo, el i-esímo mayor elemento estará en su lugar correcto

6 5 3 1 8 7 2 4

Ordenamiento con bubble-sort

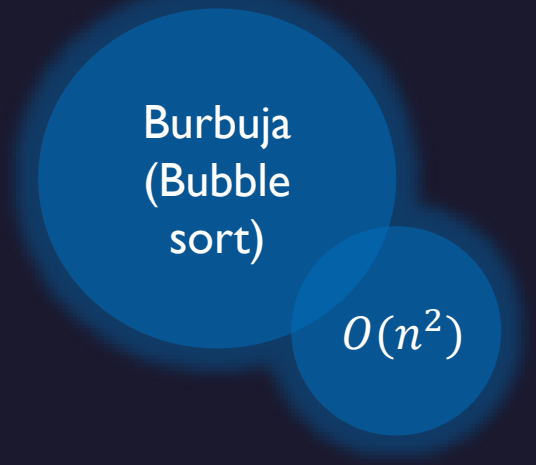


Realiza n ciclos de comparaciones entre cada par de posiciones continuas

Al final del i-esímo ciclo, el i-esímo mayor elemento estará en su lugar correcto

```
     ALGORITHM bubbleSort (A, n)
1:   // Input: Array A, de tamaño n
2:   // Output: A ordenado
3:
4:   for i = 0 to n − 1 do
5:       change = false
6:       for j = 0 to n− i − 1 do
7:           if A[j] > A[j + 1] then
8:               // Intercambio de A[j] con A[j + 1]
9:               aux = A[j + 1]
10:              A[j + 1] = A[j]
11:              A[j] = aux
12:              change = true
13:          end if
14:      end for
15:      if change = false then
16:          break
17:      end if
18:  end for
```

Ordenamiento con bubble-sort



Bubble-sort es $O(n^2)$

Realiza n ciclos entre pares, y cada ciclo toma hasta n comparaciones entre pares

Si en una iteración no se realizo ningún intercambio, el arreglo ya esta ordenado

```
     ALGORITHM bubbleSort (A, n)
1:   // Input: Array A, de tamaño n
2:   // Output: A ordenado
3:
4:   for i = 0 to n − 1 do
5:       change = false
6:       for j = 0 to n− i − 1 do
7:           if A[j] > A[j + 1] then
8:               // Intercambio de A[j] con A[j + 1]
9:               aux = A[j + 1]
10:              A[j + 1] = A[j]
11:              A[j] = aux
12:              change = true
13:          end if
14:      end for
15:      if change = false then
16:          break
17:      end if
18:  end for
```

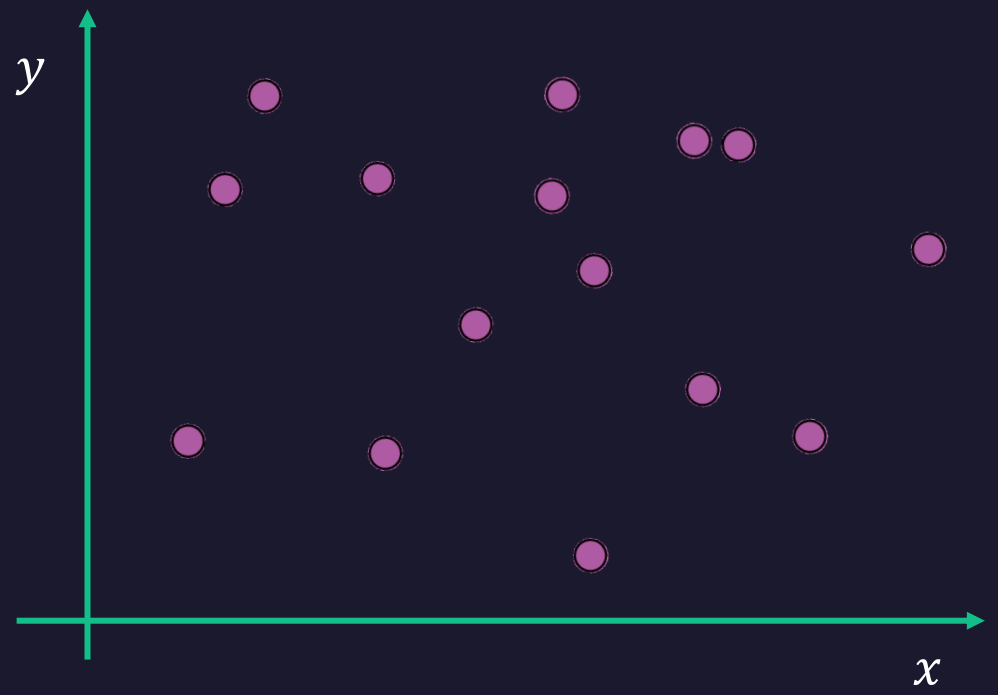
Problema: closest pair

Es un problema de geometría computacional

Sirve para clustering jerárquico en machine learning, donde las muestras mas cercanas se agrupan en un solo cluster

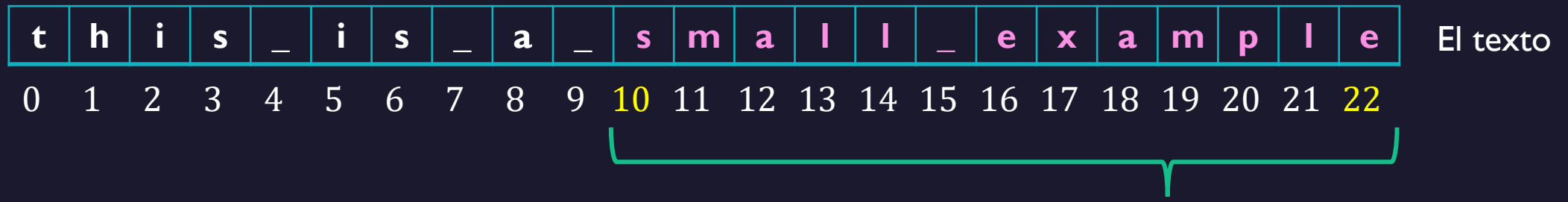
El problema:
Hay n puntos con d dimensiones
¿Cuál es el par de puntos con distancia mas cercana entre si?

¿Cómo se resolvería con un algoritmo de fuerza bruta?
¿Qué complejidad tendría?

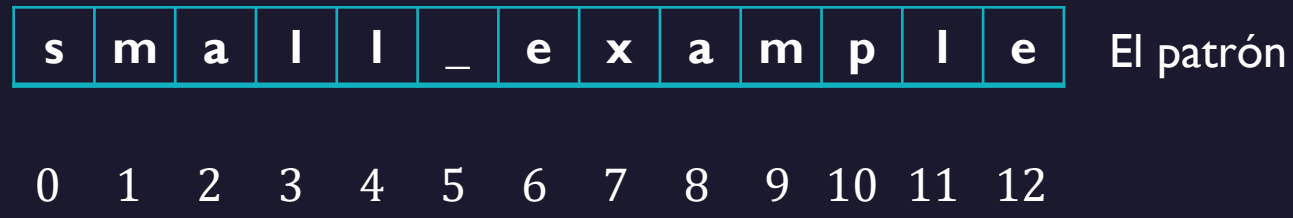


Si $d = 2$, podemos verlos como puntos con coordenadas x, y

Problema: string matching



Dado un texto de tamaño n, y un patrón de tamaño m, encuentra los índices de un substring del texto que contenga el patrón



Para este ejemplo, la salida son los índices 10 y 22

¿Cómo seria el algoritmo de fuerza bruta?
¿Qué complejidad tendría?

Búsqueda exhaustiva

Encontrar la secuencia correcta es un tipo de puzzle ambiental, común en videojuegos

Ejemplo:
Hay 5 monumentos. El cofre del tesoro se desbloquea si los enciendes en el orden correcto

¿Cuántas intentos tendrías que hacer?

Búsqueda exhaustiva

La búsqueda exhaustiva es un algoritmo de fuerza bruta

Consiste en listar exhaustivamente todas las posibles soluciones

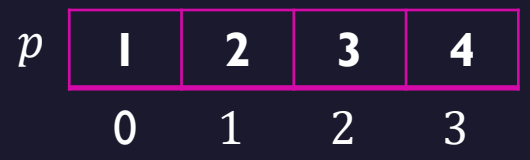
En este ejemplo particular, una búsqueda exhaustiva usaría un algoritmo de generación de permutaciones para probar cada una

1 2 3 4 5
1 2 3 5 4
1 2 4 3 5
...

Las permutaciones crecen factorialmente

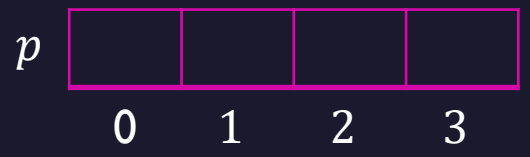
5! = 120

Generación de permutaciones



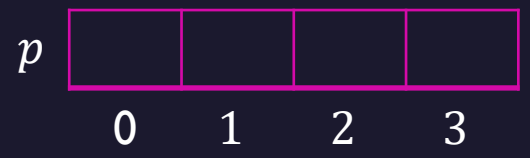
Paso 1: Elige i : el índice mas alto cuyo número sea menor que el número en la casilla siguiente

Elige j: el índice mas alto cuyo número sea mayor que $p[i]$



Paso 2:

Los valores en $p[i]$ y $p[j]$ se intercambian



Paso 3:

Se invierten todos los elementos de $p[i+1]$ hasta $p[n-1]$

Generación de permutaciones

p	1	2	3	4
	0	1	2	3

Paso 1:

¿Cuál es el **mayor índice** i tal que $p[i] < p[i+1]$? Es $i = 2$

¿Cuál es el **mayor índice** j, mayor que i, tal que $p[i] < p[j]$? Es $j = 3$

p	1	2	**4**	**3**
	0	1	2	3

Paso 2:

Los valores en $p[i]$ y $p[j]$ se intercambian

p	1	2	**4**	**3**
	0	1	2	3

Paso 3:

Se invierten todos los elementos de $p[i+1]$ hasta $p[n-1]$
En este ejemplo solo abarca $p[3]$

Permutación nueva

Generación de permutaciones

p	1	2	**4**	**3**
	0	1	2	3

Paso 1:

¿Cuál es el mayor **índice** i tal que $p[i] < p[i+1]$? Es $i = 1$

¿Cuál es el mayor **índice** j, mayor que i, tal que $p[i] < p[j]$? Es $j = 3$

p	1	**3**	**4**	**2**
	0	1	2	3

Paso 2:

Los valores en $p[i]$ y $p[j]$ se intercambian

p	1	**3**	**2**	**4**
	0	1	2	3

Paso 3:

Se invierten todos los elementos de $p[i+1]$ hasta $p[n-1]$

En este ejemplo abarca de $p[2]$ a $p[3]$

Permutación nueva

Generación de permutaciones

ALGORITHM $permutacionesLex(n)$

1: // Computa las permutaciones de n, en orden lexicográfico
2: // **Input:** entero positivo n
3: // **Output:** lista L con las $n!$ permutaciones
4:
5: $p = [1, 2, 3, \ldots, n]$ // Permutación base
6: L = lista vacía
7: Agregar p a L
8:
9: **while** p tenga 2 elementos consecutivos tales que $p[i] < p[i+1]$ **do**
10: i = el mayor índice tal que $p[i] < p[i+1]$
11: j = el mayor índice tal que $p[i] < p[j]$
12: Intercambiar $p[i]$ con $p[j]$
13: Invertir el orden de la sección $p[i+1]$ hasta $p[n-1]$
14: Agrega p a la lista de permutaciones L
15: **end while**
16: **return L**

Lo anterior, pero en pseudocódigo

- Codifica este algoritmo (no otro) en C/C++ o Python
- Haz que imprima las permutaciones
- ¿Cuál es la mayor n que consigue procesar?

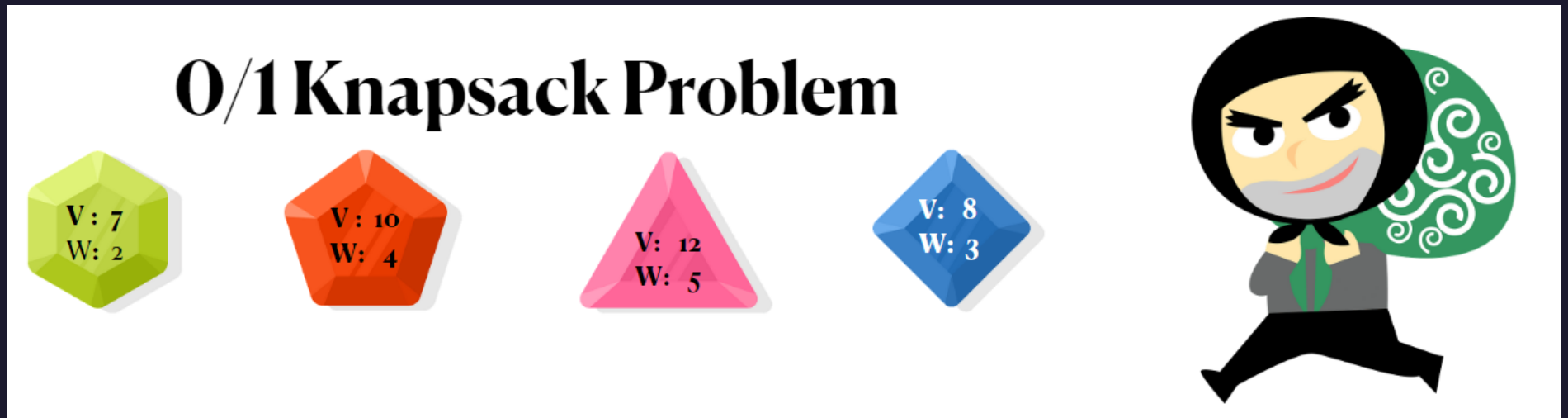
Generación de power sets (conjuntos potencia)

Los números binarios son muy útiles para representar soluciones a algunos problemas combinatorios

Google it:
¿Para que sirve resolver el knapsack?

Knapsack problem

Tienes un bolso y puedes cargar máximo $W = 10kg$
¿Cuáles objetos te llevas para maximizar su valor acumulado total V?



Generación de power sets (conjuntos potencia)

Conjunto potencia (power set):

Es el conjunto de todos los subconjuntos de un conjunto.
Por ejemplo, los subconjuntos de $\{a, b, c\}$ son:

$\emptyset$
$a, b, c,$
$ab, ac, ba,$
abc

Se puede representar en binario

$\{0\ , 1\}$

Si $n = 1$, el conjunto potencia incluye el conjunto vacío y el conjunto con 1 elemento

$\{00\ , 01,$
$11, 10\}$

Si $n = 2$, el conjunto potencia incluye 4 combinaciones

$\{000, 001,$
$011,\ 010,$
$110,\ 111,$
$101,\ 100\}$

Si $n = 3$, hay 8 combinaciones representando los posibles subconjuntos

Generación de power sets (conjuntos potencia)

1) $\{0\,,1\}$ Comienza los subconjuntos de $n = 1$

2) $\{0\,,1\}$
$\{1\,,0\}$ Crea una copia de la lista en orden inverso

3) $\{00\,,01\}$
$\{1\,,0\}$ Agrega un 0 al inicio de cada elemento de la primer lista

4) $\{00\,,01\}$
$\{11\,,10\}$ Agrega un 1 al inicio de cada elemento de la segunda lista

5) $\{00\,,01,$
$11\,,10\}$ Fusiona ambas listas en 1 sola. Estos son los subconjuntos de $n = 2$

Practica repitiendo desde el paso 2 para $n = 3$ y $n = 4$

Generación de subsets

Lo anterior, pero en pseudocódigo

Codifícalo en C/C++ o Python (este algoritmo, NO OTRO)

Haz que imprima los subconjuntos

¿Cuál es la mayor n que consigues procesar?

ALGORITHM $powerSetBin(n)$

```
1:  // Computa recursivamente el conjunto potencia de n elementos
2:  // Input: entero positivo n
3:  // Output: lista con los 2^n subconjuntos como sublistas
4:
5:  // Caso base
6:  if n == 1 then
7:      return [[0], [1]]
8:  else
9:      // Caso recursivo
10:     L1 = powerSetBin(n − 1)
11:     L2 = L1 en reversa
12:     A cada sublista de L1, agregar un 0 al inicio
13:     A cada sublista de L2, agregar un 1 al inicio
14:     return L1 y L2 concatenadas
15: end if
```

Hoy aprendiste

- Algunos algoritmos de fuerza bruta
  - Algoritmos de búsqueda secuencial
  - Ordenamientos por selección y burbuja
  - Búsqueda exhaustiva
  - Generación de todas las permutaciones y subconjuntos

- Problema de optimización clásico: knapsack
- Problema de procesamiento de strings: string matching
- Problema de geometría computacional: closest pair

- Aplicación del conjunto potencia y su codificación binaria

Algoritmos de fuerza bruta: tarea

Implementa algoritmos de fuerza bruta

¿Cuando?
El día de la próxima clase.
Detalles en Canvas, revísalo con atención.

Entrega:
link a Replit/Colab y scripts (ambos, siempre)
Pdf para investigaciones

- En C / C++ / Python
  - Algoritmo de generación de permutaciones en orden lexicográfico
  - Algoritmo de generación de subsets (solo listarlos para una n dada)
  - Además: Investiga un algoritmo de fuerza bruta adicional.
    Descríbelo con pseudocódigo e impleméntalo.