

# Ayudantía: Backtracking II & Greedy



Joaquín Peralta, Amelia González, Alexander Infanta, Elias Ayaach, Paula Grune

# Repaso I1

## 4. Ordenamiento

Hasta que Murphy interfirió, estábamos disfrutando de un buen día custodiando un arreglo ordenado  $A[1..n]$  de números enteros dentro del rango  $[1..n^2]$ . Sin embargo, en un momento de descuido, tomó nuestro arreglo  $A$  y lo cortó en bloques de  $b$  elementos usando una tijera, como si fuera una cinta (donde  $n$  es divisible por  $b$ ). Después, mezcló estos bloques en una bolsa, los sacó uno por uno sin mirar y los unió en ese orden, creando un nuevo arreglo de  $n$  elementos que ya no está necesariamente ordenado.

Por ejemplo, para el arreglo

$$A[1..12] = \langle 3, 5, 7, 8, 9, 10, 12, 15, 17, 20, 21, 23 \rangle$$

y  $b = 3$ , un posible resultado de la acción de Murphy podría ser:

$$\langle 12, 15, 17, 3, 5, 7, 20, 21, 23, 8, 9, 10 \rangle.$$

Ya que nuestra tarea era mantener el arreglo ordenado, debemos restablecer el orden rápidamente para que nuestra jefa no lo note, además usando la menor cantidad de espacio extra posible para no ser detectados. Considere los siguientes casos. Al medir el tiempo de ejecución del algoritmo, considere todas las operaciones realizadas.

# Repaso I1

- a) (2 pts) Si  $b \in O(1)$ , ¿qué algoritmo de ordenación usaría para recuperar el orden de  $A$  de manera que tenga el mínimo tiempo de ejecución posible, mientras usa la mínima cantidad de memoria adicional posible? Indique el tiempo de ejecución y memoria adicional usada por su algoritmo.

# Repaso I1

- a) (2 pts) Si  $b \in O(1)$ , ¿qué algoritmo de ordenación usaría para recuperar el orden de  $A$  de manera que tenga el mínimo tiempo de ejecución posible, mientras usa la mínima cantidad de memoria adicional posible? Indique el tiempo de ejecución y memoria adicional usada por su algoritmo.

**Solución:** Dado que el arreglo está parcialmente desordenado (porque cada bloque de  $b$  elementos se mantiene ordenado), se podría aprovechar para reducir el tiempo necesario para ordenar, comparando sólo los primeros (o últimos) elementos de cada bloque. Sin embargo, para  $b = 1$  lo mejor que pueden hacer es algo que ordene en tiempo  $O(n \lg n)$  [**50 % del puntaje**] y usando espacio  $O(1)$  (es decir, in-place) [**50 % del puntaje**]. Pueden plantear usar **HeapSort** o definir uno propio (mientras que respeten las complejidades de tiempo y espacio mencionadas). Quizás también podrían plantear usar **RadixSort**, pero deberían argumentar que la cantidad de dígitos de cada número del arreglo es  $O(\log n)$ . Estaría mal usar **CountingSort** en este caso, ya que el rango de los números hace que ese algoritmo tenga tiempo de ejecución (y espacio adicional)  $O(n^2)$ .

# Repaso I1

- b) (2 pts) Si  $b = \sqrt{n}$ , diseñe un algoritmo de ordenación para  $A$  cuyo tiempo de ejecución sea estrictamente menor que el de los algoritmos estudiados en clases. Si lo necesita, puede usar algoritmos estudiados en clases como subrutina, indicando claramente cuáles son (y demás detalles que considere necesarios para entender su algoritmo). Su algoritmo puede usar espacio adicional  $O(n)$ .

# Repaso I1

- b) (2 pts) Si  $b = \sqrt{n}$ , diseñe un algoritmo de ordenación para  $A$  cuyo tiempo de ejecución sea estrictamente menor que el de los algoritmos estudiados en clases. Si lo necesita, puede usar algoritmos estudiados en clases como subrutina, indicando claramente cuáles son (y demás detalles que considere necesarios para entender su algoritmo). Su algoritmo puede usar espacio adicional  $O(n)$ .

**Solución:** Aquí se puede lograr tiempo  $O(n)$  definiendo un algoritmo que coloque los primeros (o últimos) elementos de cada bloque en un arreglo separado, ordene esos elementos, y luego copie los bloques de acuerdo a ese orden en una secuencia adicional (que necesita espacio  $O(n)$ ). Es importante que los primeros elementos de cada bloque se ordenen en un arreglo separado, para evitar el tiempo original  $O(n \log n)$ . Un ejemplo de algoritmo es el siguiente:

---

**Algorithm 6:** Ordenar( $A[1..n]$ )

---

```
1 begin
2   Copiar el primer elemento de cada bloque en un arreglo  $B[1..\sqrt{n}]$ , en donde cada elemento
   almacena un elemento de  $A$  y el bloque al que corresponde
3   HeapSort( $B$ )
4   for  $i = 1, \dots, n$  do
5     Copiar el bloque indicado por el elemento  $B[i]$  al final de la secuencia  $C[1..n]$ 
6   Copiar  $C$  en  $A$ 
```

---

# Repaso I1

- b) (2 pts) Si  $b = \sqrt{n}$ , diseñe un algoritmo de ordenación para  $A$  cuyo tiempo de ejecución sea estrictamente menor que el de los algoritmos estudiados en clases. Si lo necesita, puede usar algoritmos estudiados en clases como subrutina, indicando claramente cuáles son (y demás detalles que considere necesarios para entender su algoritmo). Su algoritmo puede usar espacio adicional  $O(n)$ .

---

**Algorithm 6:** Ordenar( $A[1..n]$ )

---

```
1 begin
2   Copiar el primer elemento de cada bloque en un arreglo  $B[1..\sqrt{n}]$ , en donde cada elemento
   almacena un elemento de  $A$  y el bloque al que corresponde
3   HeapSort( $B$ )
4   for  $i = 1, \dots, n$  do
5     Copiar el bloque indicado por el elemento  $B[i]$  al final de la secuencia  $C[1..n]$ 
6     Copiar  $C$  en  $A$ 
```

---

Dado que **HeapSort** se realiza sobre un arreglo de  $\sqrt{n}$  elementos, el tiempo de ejecución es  $O(\sqrt{n} \lg(\sqrt{n}))$ , lo cual es  $O(n)$ . el resto de los pasos también toman tiempo  $O(n)$ , por lo que ese es el tiempo total. En lugar de **HeapSort**, podrían haber usado otro algoritmo y estar bien (por ejemplo, **QuickSort**, **MergeSort**, **SelectionSort**, **InsertionSort**, o **RadixSort**). Nuevamente, no es correcto usar **CountingSort**, por la misma razón de la parte (a).

# Repaso I1

- c) (2 pts) Si  $b = \sqrt{n}$ , diseñe un algoritmo de ordenación para  $A$  cuyo tiempo de ejecución sea estrictamente menor que el de los algoritmos estudiados en clases, y cuyo espacio adicional sea  $O(\sqrt{n})$ . Si lo necesita, puede usar algoritmos estudiados en clases como subrutina, indicando claramente cuáles son y demás detalles que considere necesarios para entender su algoritmo.



# Repaso I1

c) (2 pts) Si  $b = \sqrt{n}$ , diseñe un algoritmo de ordenación para  $A$  cuyo tiempo de ejecución sea estrictamente menor que el de los algoritmos estudiados en clases, y cuyo espacio adicional sea  $O(\sqrt{n})$ . Si lo necesita, puede usar algoritmos estudiados en clases como subrutina, indicando claramente cuáles son y demás detalles que considere necesarios para entender su algoritmo.

**Solución:** A diferencia de la parte anterior, sólo podemos usar espacio  $O(\sqrt{n})$ . La siguiente solución usa incluso menos espacio adicional  $O(1)$ . La idea es usar algo similar a **SelectionSort** con el primer elemento de cada bloque.

---

**Algorithm 7:** SelectionSortBloques( $A[1..n]$ )

---

```
1 begin
2   for  $i = 1..\sqrt{n}$  do
3       min  $\leftarrow i$  // Posición del mínimo
4       for  $j = i + 1..\sqrt{n}$  do
5           if  $A[(\text{min} - 1)\sqrt{n} + 1] > A[(j - 1)\sqrt{n} + 1]$  then
6               min  $\leftarrow j$ 
7       Intercambiar los elementos de los bloques  $i$  y min, elemento a elemento ( $O(1)$  espacio adicional)
```

---

Dado que es **SelectionSort**, el tiempo de ejecución es  $O((\sqrt{n})^2) = O(n)$ . El espacio adicional es  $O(1)$ . Cualquier alternativa que tome tiempo  $O(n)$  y use espacio  $O(\sqrt{n})$  y sea correcta es válida.

# Backtracking 2

Recordando la ayudantía pasada...



# Backtracking 2

- **X**: variables
- **D**: dominios
  - **D<sub>x</sub>**: dominio de **X**
- **R**: restricción
  - c/u para un subconjunto de **X**

*is solvable*( $X, D, R$ ):

*if*  $X = \emptyset$ , *return true*

$x \leftarrow$  alguna variable de  $X$

*for*  $v \in D_x$ :

*if*  $x = v$  viola  $R$ , *continue*

$x \leftarrow v$

*if is solvable*( $X - \{x\}, D, R$ ):

*return true*

$x \leftarrow \emptyset$

*return false*

# Backtracking 2

## Poda

Técnica para reducir el espacio de búsqueda, eliminando ramas del árbol de decisiones que no pueden conducir a una solución válida

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

Por Factibilidad, Dominio y Consistencia, etc.

*is solvable*( $X, D, R$ ):

*if*  $X = \emptyset$ , *return true*

$x \leftarrow$  alguna variable de  $X$

*for*  $v \in D_x$ :

*if*  $x = v$  **no es válida**, *continue*

$x \leftarrow v$

*if is solvable*( $X - \{x\}, D, R$ ):

*return true*

$x \leftarrow \emptyset$

*return false*

(\*) Bajo la idea: Cada posible asignación genera un camino

# Backtracking 2

## Propagación

Cuando a una 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*  $X = \emptyset$ , *return true*

$x \leftarrow$  alguna variable de  $X$

*for*  $v \in D_x$ :

*if*  $x = v$  viola  $R$ , *continue*

$x \leftarrow v$ , **propagar**

*if is solvable*( $X - \{x\}, D, R$ ):

*return true*

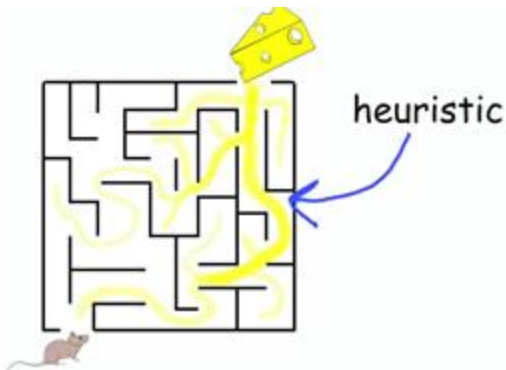
$x \leftarrow \emptyset$ , **propagar**

*return false*

# Backtracking 2

## Heurística

Es una aproximación al mejor criterio para abordar un problema.



*is solvable*( $X, D, R$ ):

*if*  $X = \emptyset$ , *return true*

$x \leftarrow$  la mejor variable de  $X$

*for*  $v \in D_x$ , de mejor a peor:

*if*  $x = v$  viola  $R$ , *continue*

$x \leftarrow v$

*if* *is solvable*( $X - \{x\}, D, R$ ):

*return true*

$x \leftarrow \emptyset$


*return false*

} “lo mejor” respecto a lo que mi heurística determina

# Backtracking 2

Un tablero de *kakuro square* es una grilla de  $N \times M$  donde cada posición tiene una celda que puede ser rellenada por un número natural distinto de 0. Además, cada fila y columna tiene un número natural que indica el valor que debe tener la suma de los números correspondientes a la fila o columna. P.ej.:

	23	15	12
23			
17			
10			



	23	15	12
23	9	8	6
17	8	4	5
10	6	3	1

El problema de rellenar la grilla cumpliendo con las restricciones se puede resolver utilizando la estrategia de backtracking.

- Identifica las variables del problema y sus respectivos dominios.
- Explica con tus palabras cómo se podría revisar en tiempo  $O(1)$  si la asignación de una variable rompe una restricción.

# Backtracking 2

a) Para resolver con backtracking este problema se debe asignar a cada celda un número, por lo que las variables son las celdas [1pto].

En teoría cada celda puede tener cualquier número natural, pero ya que no puede haber un dominio infinito para resolverlo con backtracking es necesario acotar los dominios. Un domino acotado correcto puede ser los números del 1 al  $\text{MAX}(\text{restriction})$ . Ya que se sabe que nunca un número puede ser mayor al valor de su restricción [1pto].

b) Se puede mantener un contador por cada restricción del problema que cuente la suma actual de cada fila o columna. De esta manera se inmediatamente si rompo la restricción cuando el contador supera la restricción [2pts].

**Hasta aquí llegamos la semana pasada!**



# Backtracking 2

a) Para resolver con backtracking este problema se debe asignar a cada celda un número, por lo que las variables son las celdas [1pto].

En teoría cada celda puede tener cualquier número natural, pero ya que no puede haber un dominio infinito para resolverlo con backtracking es necesario acotar los dominios. Un dominio acotado correcto puede ser los números del 1 al  $\text{MAX}(\text{restriction})$ . Ya que se sabe que nunca un número puede ser mayor al valor de su restricción [1pto].

b) Se puede mantener un contador por cada restricción del problema que cuente la suma actual de cada fila o columna. De esta manera se inmediatamente se rompe la restricción cuando el contador supera la restricción [2pts].

c) [2pts]

Podas: Ver que los contadores descritos en b) no superen el máximo antes de tener completa la fila o columna. Ver que la suma de una fila o columna actual más el número de casillas vacías de la fila o

# Backtracking 2

En una ciudad se quiere instalar estaciones de carga para vehículos eléctricos. La ciudad está dividida en una cuadrícula  $M$  de  $n \times n$  celdas y cada celda puede contener una estación de carga o estar vacía. Cada estación de carga  $E_i$  permite cubrir un cuadrante de  $c \times c$  celdas con la estación en su centro y se dispone de una lista  $Z$  de las zonas (celdas) de la ciudad que es prioritario que estén cubiertas por al menos una estación de carga. De igual forma, se dispone de una lista  $S$  que indica todas aquellas celdas de la ciudad en las que no se pueden instalar estaciones de carga por razones de seguridad. Finalmente, por limitaciones de presupuesto se puede instalar un máximo de  $k$  estaciones de carga.

- (a) (1 punto) El problema es claramente un CSP, identifique Variables, Dominios y Restricciones.

# Backtracking 2

En una ciudad se quiere instalar estaciones de carga para vehículos eléctricos. La ciudad está dividida en una cuadrícula  $M$  de  $n \times n$  celdas y cada celda puede contener una estación de carga o estar vacía. Cada estación de carga  $E_i$  permite cubrir un cuadrante de  $c \times c$  celdas con la estación en su centro y se dispone de una lista  $Z$  de las zonas (celdas) de la ciudad que es prioritario que estén cubiertas por al menos una estación de carga. De igual forma, se dispone de una lista  $S$  que indica todas aquellas celdas de la ciudad en las que no se pueden instalar estaciones de carga por razones de seguridad. Finalmente, por limitaciones de presupuesto se puede instalar un máximo de  $k$  estaciones de carga.

(a) (1 punto) El problema es claramente un CSP, identifique Variables, Dominios y Restricciones.

**Variables:** Ubicación de las estaciones  $E_i = E[i] = \text{celda}(x_i, y_i)$  para todo  $i$  en  $1 \leq i \leq k$

**Dominios:** Para cada  $E[i]$  el dominio es  $M[n \times n]$

**Restricciones:**  $E[i] \notin S$  para todo  $i$  en  $1 \leq i \leq k$

# Backtracking 2

En una ciudad se quiere instalar estaciones de carga para vehículos eléctricos. La ciudad está dividida en una cuadrícula  $M$  de  $n \times n$  celdas y cada celda puede contener una estación de carga o estar vacía. Cada estación de carga  $E_i$  permite cubrir un cuadrante de  $c \times c$  celdas con la estación en su centro y se dispone de una lista  $Z$  de las zonas (celdas) de la ciudad que es prioritario que estén cubiertas por al menos una estación de carga. De igual forma, se dispone de una lista  $S$  que indica todas aquellas celdas de la ciudad en las que no se pueden instalar estaciones de carga por razones de seguridad. Finalmente, por limitaciones de presupuesto se puede instalar un máximo de  $k$  estaciones de carga.

- (b) (3 puntos) Diseñe un algoritmo en pseudocódigo que permita ubicar las estaciones de carga para cubrir las zonas  $Z$  de la ciudad, sin ubicarlas en las celdas seguras  $S$ , utilizando a lo más de  $k$  de ellas.

Sea:

```
testSol(Z,E): true si la asignación de estaciones de E cubre todas las celdas de Z
safeCel(celda,S): true si la celda no está en S
```

```
grid(M,Z,S,E,k) // E parte vacío y k en el numero de estaciones
  if testSol(Z,E) // es solución válida
    return true
  elseif k == 0 // no hay estaciones disponibles
    return false
  else
    for celda in M // el dominio completo
      if safeCel(celda,S) // cumple restricciones
        E[k] <- celda // Estación k en esa celda
        if grid(M,Z,S,E,k-1) // una estacion menos para asignar
          return true
        E[k] <- null
    return false
```

# Backtracking 2

En una ciudad se quiere instalar estaciones de carga para vehículos eléctricos. La ciudad está dividida en una cuadrícula  $M$  de  $n \times n$  celdas y cada celda puede contener una estación de carga o estar vacía. Cada estación de carga  $E_i$  permite cubrir un cuadrante de  $c \times c$  celdas con la estación en su centro y se dispone de una lista  $Z$  de las zonas (celdas) de la ciudad que es prioritario que estén cubiertas por al menos una estación de carga. De igual forma, se dispone de una lista  $S$  que indica todas aquellas celdas de la ciudad en las que no se pueden instalar estaciones de carga por razones de seguridad. Finalmente, por limitaciones de presupuesto se puede instalar un máximo de  $k$  estaciones de carga.

- (b) (3 puntos) Diseñe un algoritmo en pseudocódigo que permita ubicar las estaciones de carga para cubrir las zonas  $Z$  de la ciudad, sin ubicarlas en las celdas seguras  $S$ , utilizando a lo más de  $k$  de ellas.

Sea:

```
testSol(Z,E): true si la asignación de estaciones de E cubre todas las celdas de Z  
safeCel(celda,S): true si la celda no está en S
```

```
testSol(Z,E)  
  Aux[n x n] // Matriz auxiliar para la verificacion  
  for celda in E // id que celdas estan cubiertas  
    Aux[(c x c)*celda] <- cubierta // centradas en celda  
  for zona in Z // verificar celdas prioritarias  
    if Aux[zona] <> cubierta  
      return false  
  return true
```

# Backtracking 2

En una ciudad se quiere instalar estaciones de carga para vehículos eléctricos. La ciudad está dividida en una cuadrícula  $M$  de  $n \times n$  celdas y cada celda puede contener una estación de carga o estar vacía. Cada estación de carga  $E_i$  permite cubrir un cuadrante de  $c \times c$  celdas con la estación en su centro y se dispone de una lista  $Z$  de las zonas (celdas) de la ciudad que es prioritario que estén cubiertas por al menos una estación de carga. De igual forma, se dispone de una lista  $S$  que indica todas aquellas celdas de la ciudad en las que no se pueden instalar estaciones de carga por razones de seguridad. Finalmente, por limitaciones de presupuesto se puede instalar un máximo de  $k$  estaciones de carga.

- (b) (3 puntos) Diseñe un algoritmo en pseudocódigo que permita ubicar las estaciones de carga para cubrir las zonas  $Z$  de la ciudad, sin ubicarlas en las celdas seguras  $S$ , utilizando a lo más de  $k$  de ellas.

Sea:

`testSol(Z,E): true si la asignación de estaciones de E cubre todas las celdas de Z`

`safeCel(celda,S): true si la celda no está en S`

```
safeCel(celda,S)
    for s in S
        if s == celda
            return false
    return true
```

## Backtracking 2

- (c) (2 puntos) Un experto aconseja que se minimice la distancia media desde las celdas de  $Z$  a la estación más cercana, siempre utilizando a lo más  $k$  estaciones. Modifique su algoritmo anterior de acuerdo con lo señalado. *Hint: Puede asumir que dispone de las funciones:  $\text{distancia}(E_i, Z_i)$  que le entrega la distancia entre la Zona prioritaria  $Z_i$  y la Estación de carga  $E_i$ , y la función  $\text{masCerca}(Z_i)$  que entrega la estación de carga más cercana a  $Z_i$ .*



# Backtracking 2

- (c) (2 puntos) Un experto aconseja que se minimice la distancia media desde las celdas de  $Z$  a la estación más cercana, siempre utilizando a lo más  $k$  estaciones. Modifique su algoritmo anterior de acuerdo con lo señalado. *Hint: Puede asumir que dispone de las funciones:  $\text{distancia}(E_i, Z_i)$  que le entrega la distancia entre la Zona prioritaria  $Z_i$  y la Estación de carga  $E_i$ , y la función  $\text{masCerca}(Z_i)$  que entrega la estación de carga más cercana a  $Z_i$ .*

Sea:

```
testSol(Z,E): true si la asignación de estaciones de E cubre todas las celdas de Z
safeCel(celda,S): true si la celda no está en S
dMedia(Z,E): distancia media de Z a E mas cercana
```

```
grid(M,Z,S,E,k,dMin) // E parte vacío, k numero de estaciones, dMin +infinito
  if testSol(Z,E) // es solución válida
    d = dMedia(Z,E)
    if d < dMin // es la de distancia minima
      dMin <- d
      minE <- E // guardamos la solucion
    return true
  elseif k == 0 // no hay estaciones disponibles
    return false
  else
    for celda in M // el dominio completo
      if safeCel(celda,S) // cumple restricciones
        E[k] <- celda // Estación k en esa celda
        if grid(M,Z,S,E,k-1) // una estacion menos para asignar
          // E[k] es una solucion, next
        E[k] <- null
    return false
```



# Backtracking 2

- (c) (2 puntos) Un experto aconseja que se minimice la distancia media desde las celdas de  $Z$  a la estación más cercana, siempre utilizando a lo más  $k$  estaciones. Modifique su algoritmo anterior de acuerdo con lo señalado. *Hint: Puede asumir que dispone de las funciones:  $distancia(E_i, Z_i)$  que le entrega la distancia entre la Zona prioritaria  $Z_i$  y la Estación de carga  $E_i$ , y la función  $masCerca(Z_i)$  que entrega la estación de carga más cercana a  $Z_i$ .*

Sea:

```
testSol(Z,E): true si la asignación de estaciones de E cubre todas las celdas de Z  
safeCel(celda,S): true si la celda no está en S  
dMedia(Z,E): distancia media de Z a E mas cercana
```

```
dMedia(Z,E)  
  D <- 0  
  m <- cardinalidad(Z)  
  for i in Z  
    D <- D + distancia(masCerca(i),i)  
  return D / m
```

# Greedy (algoritmos codiciosos)



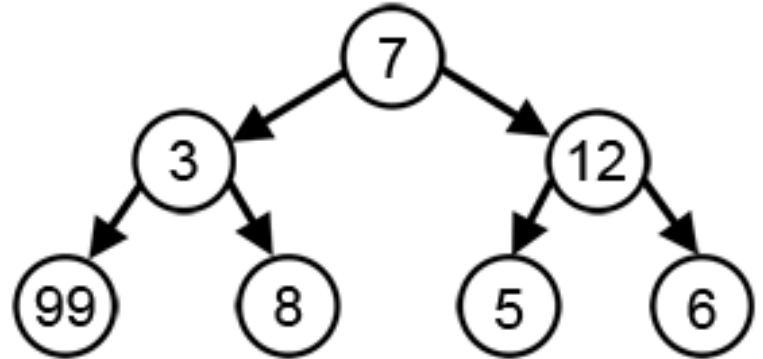
# Greedy

**Diferente a  
Backtracking**

**Varias soluciones  
pueden ser correctas  
dependiendo del  
enfoque**

**Usualmente usado en  
problemas de  
optimización**

**por ejemplo, si queremos encontrar el  
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 o máximos locales*** o soluciones parcialmente satisfactorias. (Requiere de un problema con subestructura óptima)

Para resolver  
bajo  
estrategia  
codiciosa  
necesito



una estrategia ...

...tal que buscar “la mejor solución disponible en el momento” sea un camino prometedor para encontrar una solución óptima global

## I2 2023-2 – Greedy

Para satisfacer la demanda de peluches de Fiu, la mascota de los Juegos Panamericanos y Parapanamericanos, se deben construir tiendas cerca de los recintos deportivos. Consideremos el problema de **minimizar la cantidad de tiendas**, para lo cual, representamos los recintos como puntos  $R = \{r_1, \dots, r_n\}$  en una misma recta, diciendo que un recinto está cubierto si en la recta hay una tienda a lo más a  $L$  kilómetros de él. Por ejemplo, si  $L = 1$  y  $R = \{0, 2\}$  la solución óptima es ubicar una tienda entre ambos recintos



## I2 2023-2 – Greedy

- A) Se propone la siguiente estrategia para escoger dónde colocar tiendas: ***“ubicar la siguiente tienda en la posición que maximiza el número de recintos nuevos cubiertos y que no habían sido cubiertos por tiendas añadidas antes”***. Demuestre que esta estrategia no es óptima en todos los casos.

## I2 2023-2 – Greedy

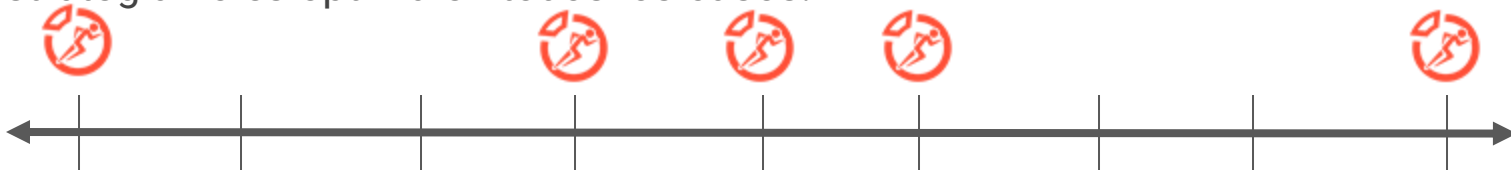
- A) Se propone la siguiente estrategia para escoger dónde colocar tiendas: *“ubicar la siguiente tienda en la posición que maximiza el número de recintos nuevos cubiertos y que no habían sido cubiertos por tiendas añadidas antes”*. Demuestre que esta estrategia no es óptima en todos los casos.

→ Forzaremos a que la estrategia se equivoque



## 12 2023-2 – Greedy

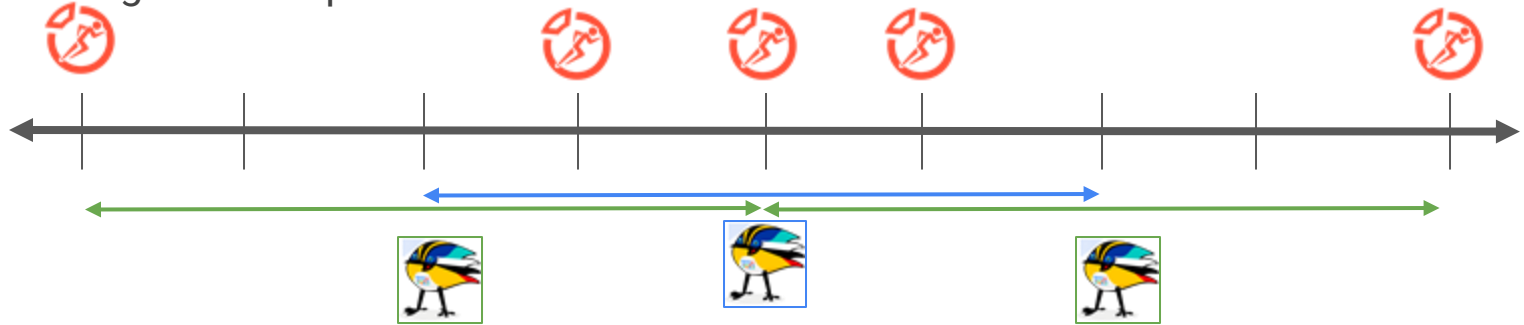
- A) Se propone la siguiente estrategia para escoger dónde colocar tiendas: ***“ubicar la siguiente tienda en la posición que maximiza el número de recintos nuevos cubiertos y que no habían sido cubiertos por tiendas añadidas antes”***. Demuestre que esta estrategia no es óptima en todos los casos.



→ Elegimos  $L = 2$  y “sobrecargamos” el centro para que la estrategia ponga tiendas de más

## 12 2023-2 – Greedy

- A) Se propone la siguiente estrategia para escoger dónde colocar tiendas: “*ubicar la siguiente tienda en la posición que maximiza el número de recintos nuevos cubiertos y que no habían sido cubiertos por tiendas añadidas antes*”. Demuestre que esta estrategia no es óptima en todos los casos.



→ Habrá una tienda extra al centro

## I2 2023-2 – Greedy

B) Considere la siguiente estrategia codiciosa: ***“Con los recintos en orden creciente, si el primer recinto no cubierto es  $r$ , entonces ubicar la siguiente tienda a distancia  $L$  hacia adelante de  $r$ ”***. Proponga el pseudocódigo de un algoritmo codicioso que use esta estrategia para retornar la lista más corta con las ubicaciones de las tiendas que cubren el conjunto de recintos  $R$  con distancia  $L$ . Asuma que  $R$  es un arreglo no necesariamente ordenado.

## I2 2023-2 – Greedy

```
Greedy( $R, L$ ):  
1    $T \leftarrow$  lista vacía  
2    $R \leftarrow$  InsertionSort( $R$ )  
3   Insertar al final de  $T$  el dato  $R[0] + L$   
4    $i \leftarrow 1$   
5   while  $i < n$  :  
6       if  $T.last + L < R[i]$  :  
7           Insertar al final de  $T$  el dato  $R[i] + L$   
8        $i + 1$   
9   return  $T$ 
```

$n \rightarrow$  largo  $R$

## I2 2023-2 – Greedy

C) Si  $R$  contiene  $n$  recintos y se entrega como arreglo no necesariamente ordenado, determine si existe distinción entre mejor y peor caso para su algoritmo de (b) y determine la complejidad de los casos identificados

## I2 2023-2 – Greedy

C) Si  $R$  contiene  $n$  recintos y se entrega como arreglo no necesariamente ordenado, determine si existe distinción entre mejor y peor caso para su algoritmo de (b) y determine la complejidad de los casos identificados

El algoritmo propuesto solo depende de los mejores/peores casos del algoritmo de ordenación empleado. Para el caso específico de InsertionSort, el mejor caso ocurre cuando  $R$  está ordenado y el peor, cuando hay desorden en una cantidad lineal de elementos.

## I2 2023-2 – Greedy

C) Si  $R$  contiene  $n$  recintos y se entrega como arreglo no necesariamente ordenado, determine si existe distinción entre mejor y peor caso para su algoritmo de (b) y determine la complejidad de los casos identificados

El algoritmo propuesto solo depende de los mejores/peores casos del algoritmo de ordenación empleado. Para el caso específico de InsertionSort, el mejor caso ocurre cuando  $R$  está ordenado y el peor, cuando hay desorden en una cantidad lineal de elementos.

La complejidad del loop es  $O(n)$  siempre. Luego, como InsertionSort es  $O(n)$  en el mejor caso, el mejor caso es lineal (  $O(n) + O(n) = O(n)$  ). El peor caso es cuadrático (  $O(n^2) + O(n) = O(n^2)$  ).