Ayudantía 12

Greedy + DP

Definición

Algoritmos Codiciosos

- **Decisiones locales óptimas:** En cada paso se selecciona la opción que parece mejor en ese momento, con la esperanza de alcanzar una solución globalmente óptima.
- **Agregación gradual:** Parte de una solución vacía e incorpora elementos uno a uno según una estrategia de selección.
- **No tiene recuerdos:** No revisa ni modifica decisiones pasadas. Solo mira el estado actual.
- **No garantiza optimalidad global:** Para que funcione, el problema debe tener subestructura óptima. La mayoría de las estrategias producen soluciones subóptimas.

Algoritmos Codiciosos

Pseudocódigo

Las tres funciones clave son:

- Select (qué elegir)
- Feasible (¿es válido?)
- Union (cómo incorporarlo)

Su implementación depende de cada problema.

input : Arreglo de inputs $A[0 \ldots n-1]$, cantidad de inputs n

```
Greedy(A, n):
1    S ← ∅
2    for i = 1 ... n :
3        x ← Select(A)
4        if Feasible(S, x) :
5            S ← Union(S, x)
6    return S
```

Uso

Algoritmos Codiciosos

¿Cuándo usarlo?

Especialmente útil cuando tomar la mejor decisión local en cada paso garantiza la mejor solución global.

Definición

Programación Dinámica

¿Qué es?

Estrategia para resolver problemas dividiéndolos en subproblemas más pequeños, resolviendo cada uno una sola vez y guardando el resultado.

Subproblemas solapados

La programación dinámica busca encontrar la solución óptima global combinando las soluciones óptimas de los subproblemas.

La Estrategia

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

Uso

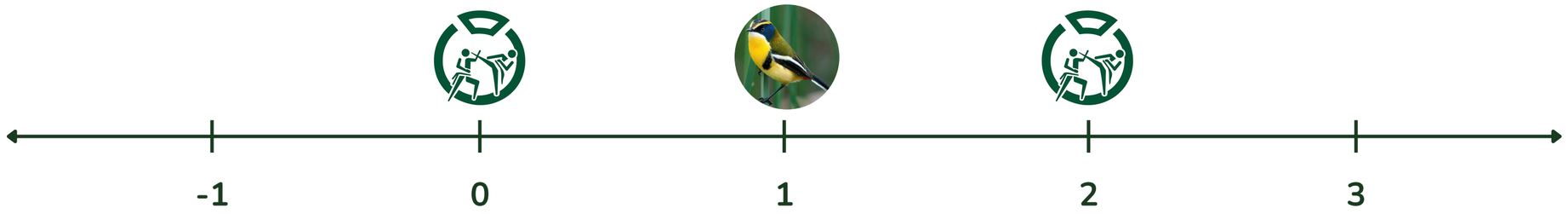
Programación Dinámica

¿Cuándo usarlo?

Especialmente útil para problemas con superposición de subproblemas, donde los mismos subproblemas se resuelven repetidamente.

Ejercicio 1

Para satisfacer la demanda de peluches de Fiu, la mascota de los Juegos Panamericanos y Parapanamericanos, se debe construir tiendas cerca de los recintos deportivos. Consideremos el problema de minimizar la cantidad de tiendas, para lo cual, representamos los recintos como puntos $R = \{r_1, \ldots, r_n\}$ 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:



Pregunta 3 - I2-2023-2

(a) [2 ptos.] 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.

Pregunta 3.a) - I2-2023-2

Pregunta 3.a) - I2-2023-2

Vamos a demostrar por contraejemplo.

La idea es poner una zona muy concentrada de recintos en el centro, donde la estrategia no puede resistir y pone una tienda ahí primero.

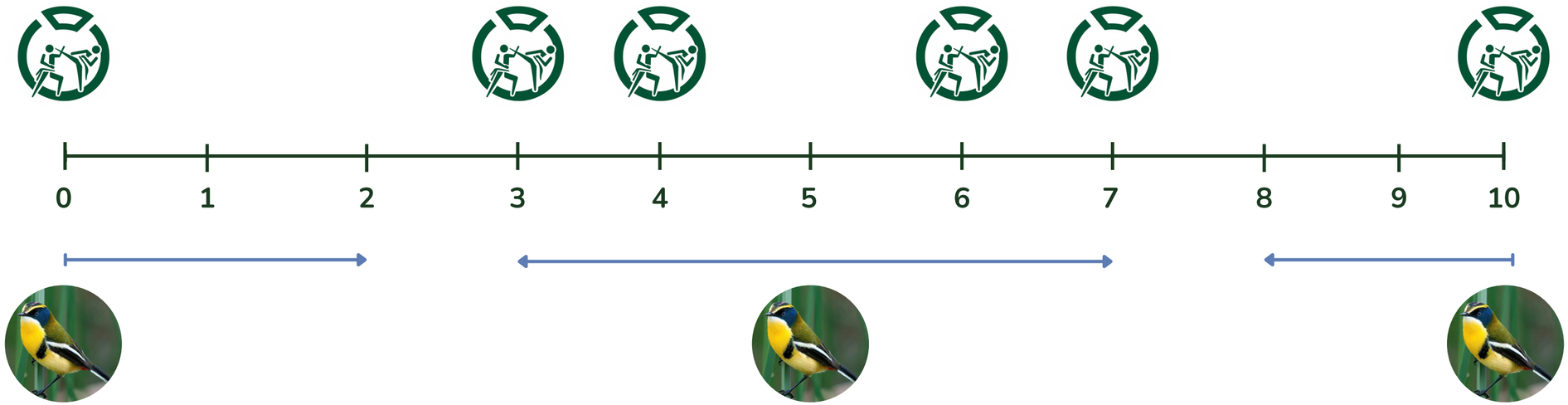
Luego necesita una tienda en cada extremo → total 3 tiendas.

Pero nosotros construimos el ejemplo para que 2 tiendas bien ubicadas cubran todo.

Pregunta 3.a) - I2-2023-2

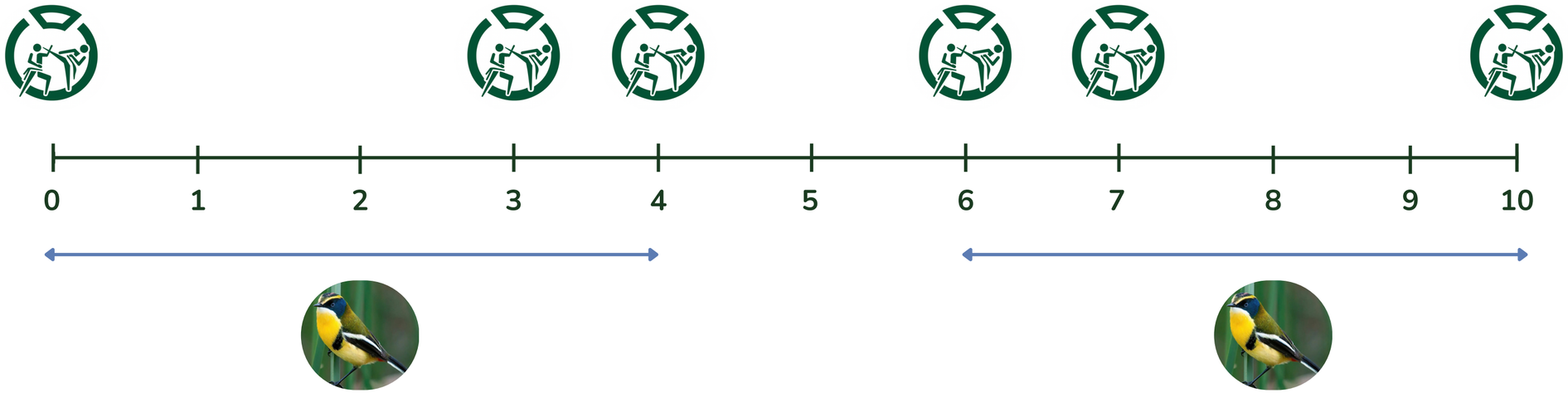
Contraejemplo: L = 2, R = {0, 3, 4, 6, 7, 10}

Usando la estrategia:



Pregunta 3.a) - I2-2023-2

Sin embargo, el óptimo es:



(b) [2 ptos.] 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.

Pregunta 3.b) - I2-2023-2

Pregunta 3.b) - I2-2023-2

"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".

```
Greedy(R, L):
1    T ← lista vacía
2    R ← InsertionSort(R)        ── Ordenamos la secuencia de recintos
3    Insertar al final de T el dato R[0] + L
4    i ← 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                    ├── Avanzar verificando el rango
                                 ┘
```

(c) Si R contiene n recintos y se entrega como arreglo no necesariamente ordenado.

(i) [1 pto.] Determine si existe distinción entre mejor y peor caso para su algoritmo de (b). Justifique.

(ii) [1 pto.] Determine la complejidad de los casos identificados en (i).

Pregunta 3.c) - I2-2023-2

Pregunta 3.c) - I2-2023-2

Sólo depende de los mejores/peores casos del algoritmo de ordenación. 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.

```
Greedy(R, L):
1    T ← lista vacía
2    R ← InsertionSort(R)
3    Insertar al final de T el dato R[0] + L
4    i ← 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
```

La complejidad del loop es O(n) siempre. Luego, como InsertionSort es O(n) en el mejor caso, el mejor caso del algoritmo es lineal O(n) + O(n) = O(n).

El peor caso es cuadrático $O(n^2)$ + O(n) = $O(n^2)$.

Ejercicio 2

Dado un conjunto de enteros positivos $A = \{a_1, \ldots, a_n\}$, el problema `DivSum` consiste en responder si es posible construir dos conjuntos B_1, B_2 disjuntos, no vacíos, tales que $B_1 \cup B_2 = A$ y tales que tienen la misma suma de sus elementos. `DivSum` puede resolverse como problema de decisión, definiendo

$$p(k, S) := \quad \begin{array}{l} 1 \text{ si y solo si es posible construir un conjunto con} \\ \text{elementos de } a_k, \ldots, a_n \text{ con suma exactamente } S. \end{array}$$

y su relación recursiva

$$p(k, S) \;=\; p(k+1, S) \texttt{ OR } p(k+1, S - a_k)$$

donde `OR` es el operador de disyunción que evalúa primero el lado izquierdo. Si dicho lado es 1, entrega 1 sin evaluar el lado derecho. Esto significa que es posible que ciertos llamados de $p(k, S)$ no se realicen.

Pregunta 2 - I2-2023-02

(a) [1 pto.] Identifique los casos base para $p(k, S)$. Defina el valor adecuado de $p(k, S)$ en tales casos.

Pregunta 2.a) - I2-2023-2

Pregunta 2.a) - I2-2023-2

Como nos interesa que la recursión no persista, debemos considerar:

- k > n: representa el caso en que no quedan elementos
- S < 0: representa una suma negativa
- S = 0: representa la condición de éxito (agotamos la suma buscada)

Con esto, los casos se resumen en:

$$p(k, S) = \begin{cases} 0 & k > n \vee S < 0 \\ 1 & S = 0 \end{cases}$$

(b) [2 ptos.] Resuelva el problema `DivSum` para $A = \{1, 4, 3\}$ mediante la definición de $p(k, S)$, usando un llamado inicial adecuado. Muestre el árbol recursivo de llamados que se genera, indicando qué elementos se incluyen/descartan en cada paso y mostrando cuándo la recursión llega a un caso base.

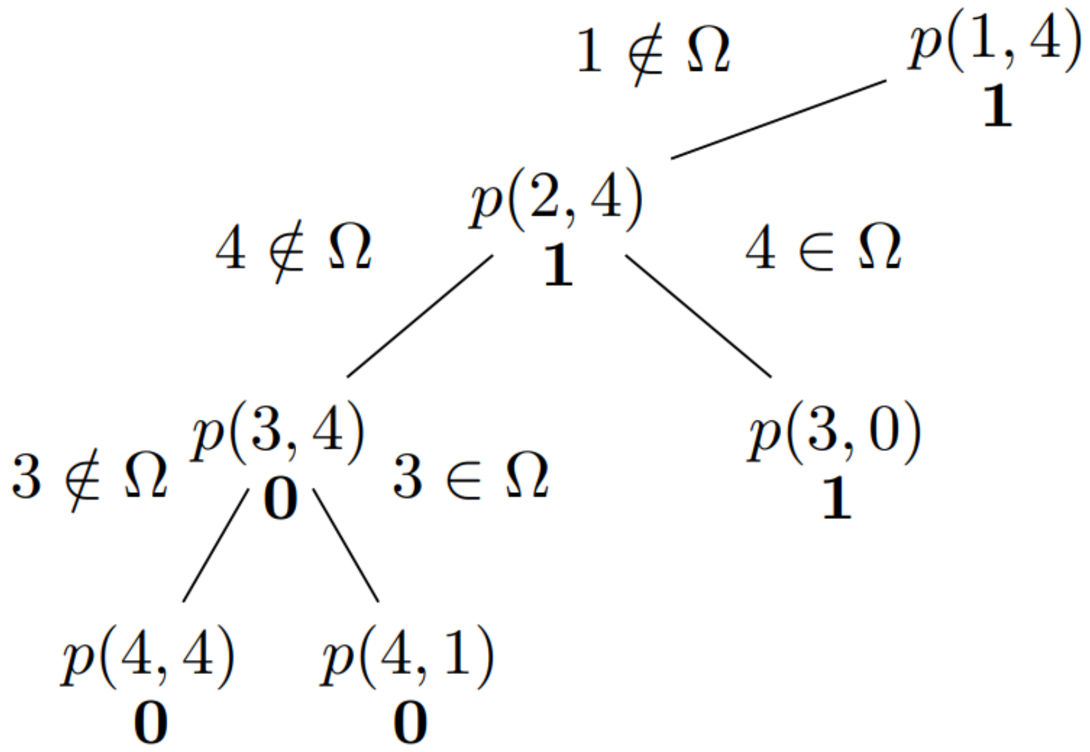
Pregunta 2.b) - I2-2023-2

Pregunta 2.b) - I2-2023-2

El llamado que responde al problema DivSum para A = {1, 4, 3} es p(1, 4), donde el segundo argumento es la mitad de la suma de todos los elementos de A. Si p(1, 4) = 1, significa que es posible construir conjuntos con igual suma.

Pregunta 2.b) - I2-2023-2

Sea Ω uno de los subconjuntos. A continuación mostramos los llamados recursivos que indican cómo construir Ω.



El subárbol derecho no se genera porque el lado izquierdo ya retornó 1.

El árbol muestra que DivSum es positivo para A, con Ω = {4} como uno de los conjuntos válidos.

(c) [3 ptos.] Proponga el pseudocódigo de un algoritmo recursivo que para cualquier conjunto A de enteros positivos, responda el problema `DivSum` para A.

Pregunta 2.c) - I2-2023-2

Pregunta 2.c) - I2-2023-2

Se asume M un arreglo bidimensional con celdas vacías. Primero proponemos el algoritmo de programación dinámica que calcula p(k, S)

```
Dynamic(k, S):
    if k > n ∨ S < 0 :
        return 0
    if S = 0 :
        return 1
    else:
        if M[k][S] ≠ ∅ :
            return M[k][S]
        else:
            M[k][S] ← Dynamic(k + 1, S) OR Dynamic(k + 1, S − a_k)
            return M[k][S]
```

```
DivSum(A):
    S ← SUM(A)
    if S es impar :
        return 0
    return Dynamic(1, S/2)
```

Feedback