Estrategias de resolución de problemas

…

Algoritmos y Estructuras de Datos
2024

UNRaf UNIVERSIDAD NACIONAL DE RAFAELA

Dada una mochila con 25 kg de capacidad,
¿qué combinación de ítems daría el mayor valor?

Item	Item 1	Item 2	Item 3	Item 4	Item 5	Item 6	Item 7
Peso	22	9	10	6	4	2	1
Valor	19	11	9	6	1	0.5	0.1

Soluciones Óptimas

- Un problema puede contener un **gran número de entradas** y, en algunos casos, **restricciones definidas** para su solución.
- En estos casos, para alcanzar la solución al problema es necesario seleccionar un **subconjunto de estas entradas**.
- Cada uno de los subconjuntos que cumplan las restricciones diremos que son **soluciones prometedoras**.
- Una solución prometedora que **maximice o minimice** una **función objetivo** la denominaremos **solución óptima**.

Problema de la mochila (*knapsack problem*)

- Entradas: todos los ítems, sus pesos y valores.
- Restricción: los pesos pueden sumar, a lo sumo, 25.
- Función a maximizar: suma de valores.

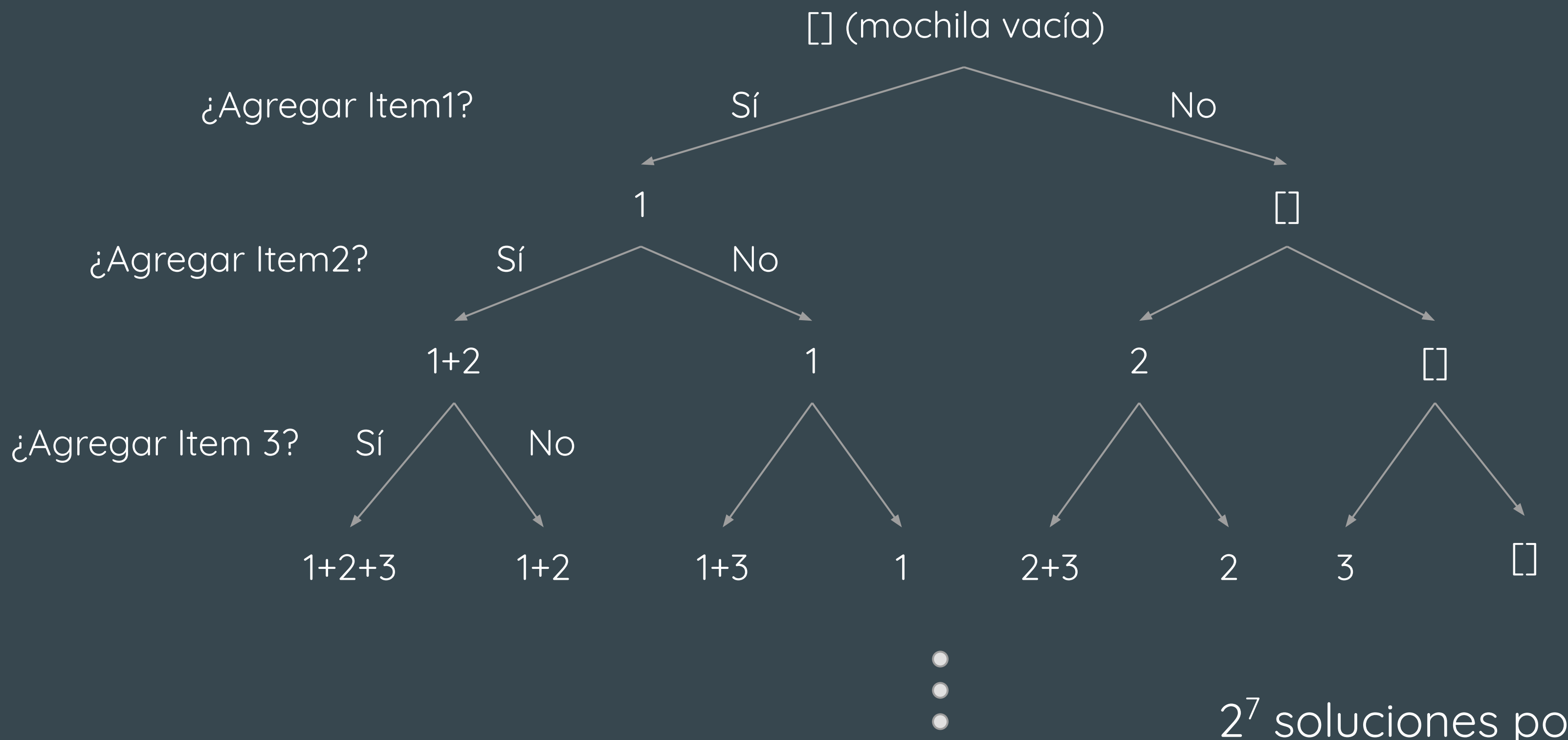
¿Cómo solucionar un problema?

- Fuerza bruta
- Ramificación y poda
- Algoritmos ávidos
- Programación dinámica

Fuerza bruta

- Consiste en evaluar todas los subconjuntos de entradas, ver si cumplen las condiciones dadas y seleccionar la mejor solución.
- Es la estrategia más simple, pero también la de mayor complejidad computacional.
- Muchas de esas esas soluciones no serán prometedoras (no cumplen la restricción del problema).

Calculando todas las soluciones posibles



2^7 soluciones posibles!!

Fuerza bruta: pseudocódigo

```
lista = [Item1, Item2, …]
pesos = [22, 9, …]
valores = [19, 11, …]
peso_max = 25
mejor_solucion = [“”, 0, 0] # Items, peso, valor
solucion_actual = [“”, 0, 0]

Por cada item en lista
  si ultimo_item:
    si solucion_actual mejor mejor_solucion y peso <= limite:
        mejor_solucion = solucion_actual
  sino:
    solucion_incluyendo[0] = solucion_actual[0] + item_actual
    solucion_incluyendo[1] = solucion_actual[1] + peso_actual
    solucion_incluyendo[2] = solucion_actual[2] + valor_actual
    calcular para solucion_incluyendo

    calcular para solucion_actual
```

Fuerza bruta: código

```python
def fb(items, pesos, valores, peso_max, solucion=["", 0, 0],
          solucion_actual = ["", 0, 0], indice = 0):
  if indice == len(items):
    if solucion_actual[2] > solucion[2] and solucion_actual[1] <= peso_max:
        solucion[0] = solucion_actual[0]
        solucion[1] = solucion_actual[1]
        solucion[2] = solucion_actual[2]
  else:
    # Incluyendo el valor
    solucion_incluyendo = ["", 0, 0]
    solucion_incluyendo[0] = solucion_actual[0] + "; " + items[indice]
    solucion_incluyendo[1] = solucion_actual[1] + pesos[indice]
    solucion_incluyendo[2] = solucion_actual[2] + valores[indice]
    fb(items, pesos, valores, peso_max, solucion,
         solucion_incluyendo, indice+1)

    # No incluyendo el valor
    fb(items, pesos, valores, peso_max, solucion, solucion_actual, indice+1)
  return solucion
```

Resultados de la solución por fuerza bruta

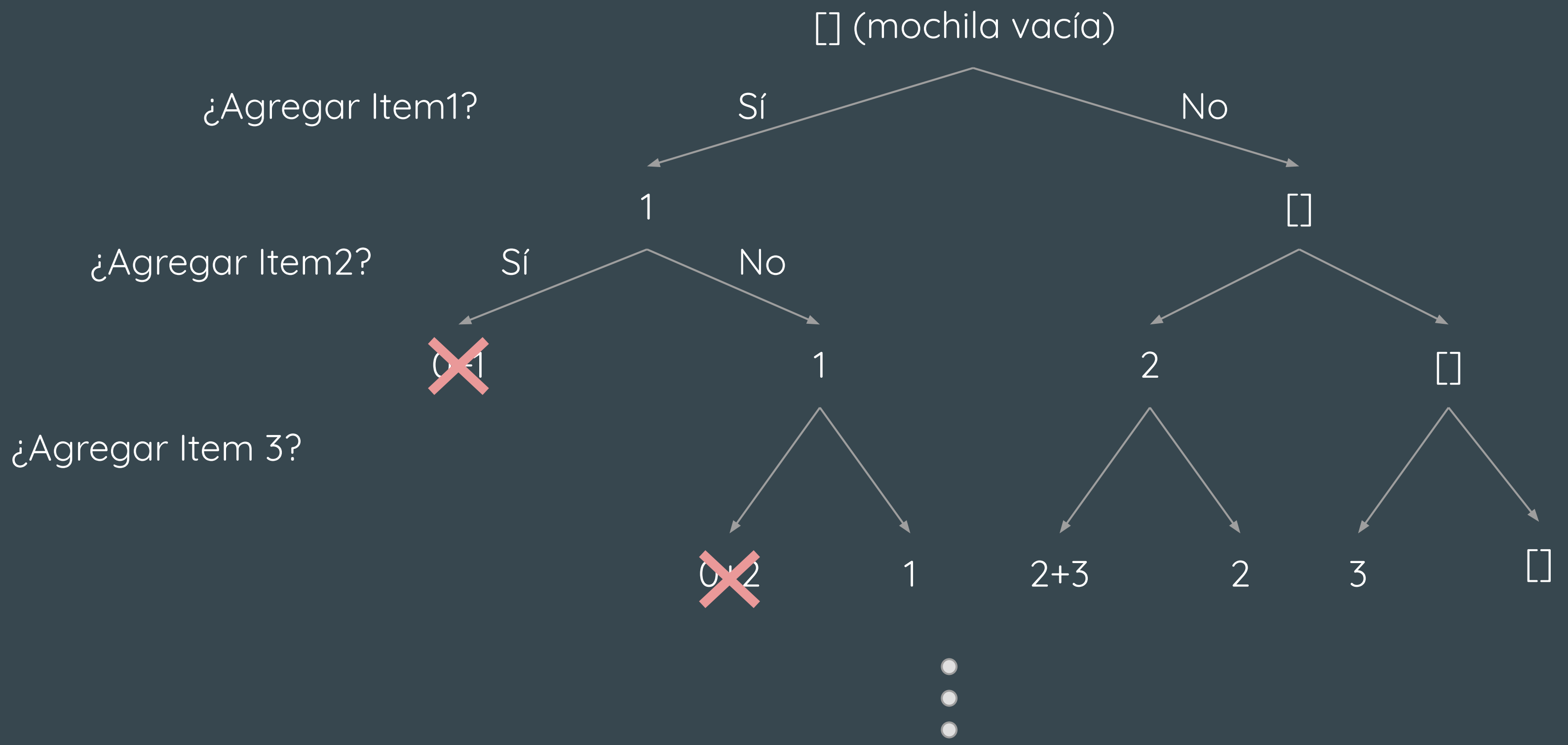
Combinación	Peso	Valor
1+2+3+4+5+6+7	54	46.6
1+2+3+4+5+6	53	46.5
1+2+3+4+5+7	52	46.1
1+2+3+4+5	51	46
1+2+3+4+6+7	50	45.6
1+2+3+4+6	49	45.5

⋮

Ramificación y Poda

- Cuando hay limitaciones para las soluciones, es deseable evitar calcular todas aquellas combinaciones que no representen soluciones prometedoras.
- De esta manera, se reduce el conjunto de soluciones calculadas, y por lo tanto, el número de cálculos totales.
- Este proceso de eliminar ramas no prometedoras del árbol de recursión se conoce como **poda**.

Calculando solo soluciones prometedoras



Poda: pseudocódigo

```
items = [Item1, Item2, …]
pesos = [22, 9, …]
valores = [19, 11, …]
peso_max = 25
mejor_solucion = [“”, 0, 0] # Items, peso, valor
solucion_actual = [“”, 0, 0]

Por cada item_actual en items
  si ultimo_item:
    si solucion_actual mejor mejor_solucion y peso <= limite:
         mejor_solucion = solucion_actual
  sino:
    solucion_incluyendo[0] = solucion_actual[0] + item_actual
    solucion_incluyendo[1] = solucion_actual[1] + peso_actual
    solucion_incluyendo[2] = solucion_actual[2] + valor_actual
    si solucion_incluyendo <= peso_max:
      calcular para solucion_incluyendo

    calcular para solucion_actual
```

Poda: código

```
def poda(items, pesos, valores, peso_max, solucion=["", 0, 0],
          solucion_actual = ["", 0, 0], indice = 0):
  if indice == len(items):
    if solucion_actual[2] > solucion[2]:
          solucion[0] = solucion_actual[0]
          solucion[1] = solucion_actual[1]
          solucion[2] = solucion_actual[2]
  else:
    # Incluyendo el valor
    solucion_incluyendo = ["", 0, 0]
    solucion_incluyendo[0] = solucion_actual[0] + "; " + items[indice]
    solucion_incluyendo[1] = solucion_actual[1] + pesos[indice]
    solucion_incluyendo[2] = solucion_actual[2] + valores[indice]
    if solucion_incluyendo[1] <= peso_max
      poda(items, pesos, valores, peso_max, solucion,
         solucion_incluyendo, indice+1)

     # No incluyendo el valor
     poda(items, pesos, valores, peso_max, solucion, solucion_actual, indice+1)
  return solucion
```

Resultados de la solución usando Poda

Combinación	Peso	Valor
1	22	19
1+6	24	19.5
1+7	23	19.1
2	9	11
2+3	9	20
2+3+5	23	21

⋮

Fuerza bruta vs Poda

- El uso de **fuerza bruta** es la estrategia más **simple** de plantear, pero requiere muchos cálculos (más tiempo y más memoria), ya que evalúa **todo el universo** de soluciones posibles.
- La **poda** de ramas no prometedoras es una estrategia que permite **disminuir la cantidad de cálculos** y acelerar un algoritmo (en este ejemplo, el número de soluciones baja de 128 a 60).

Programación Dinámica

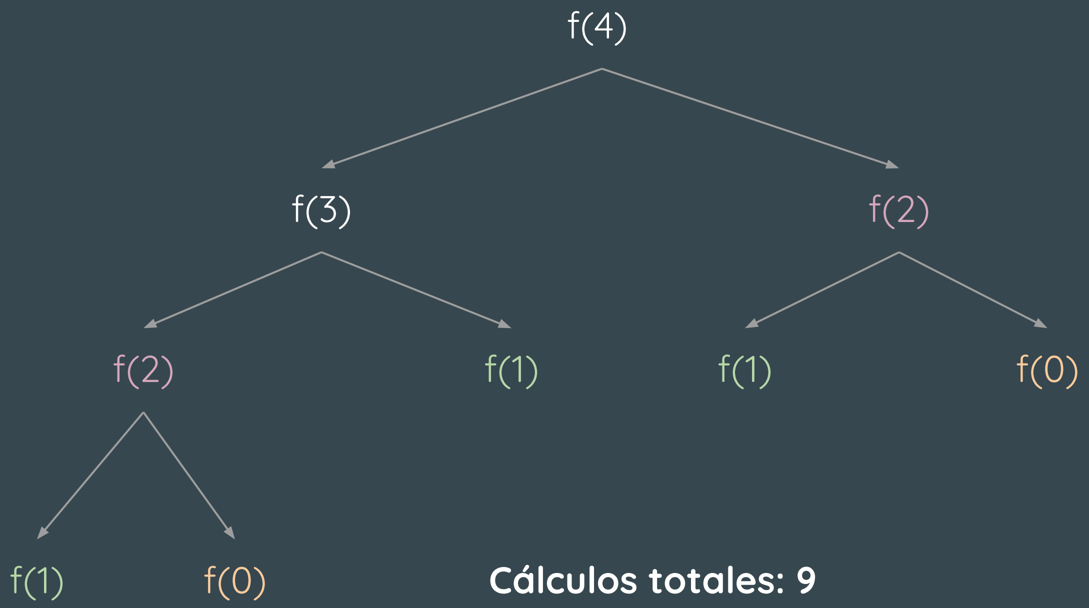
- La **programación dinámica** plantea una alternativa para **optimizar el número de cálculos requeridos** para alcanzar una solución.
- Se comienza con el problema general y se lo divide en subproblemas.
- La programación dinámica se basa en el principio de **memoización**: cada vez que un subproblema es resuelto, los resultados **se almacenan** para poder ser reutilizados más adelante.
- La solución al problema se alcanza al evaluar todas las soluciones a los subproblemas, por lo que **siempre se alcanza la solución óptima**.

Memoización

- Es una técnica de optimización para disminuir el número de cálculos realizados.
- En cada etapa, se consulta si un subproblema ya fue resuelto:
  - En caso que no se conozca el resultado, se procede a calcularlo y almacenarlo en memoria.
  - Si se conoce el resultado, se busca en memoria.
- Se sacrifica espacio en memoria para (teóricamente) disminuir el tiempo de cálculo

Recordando Fibonacci recursivo

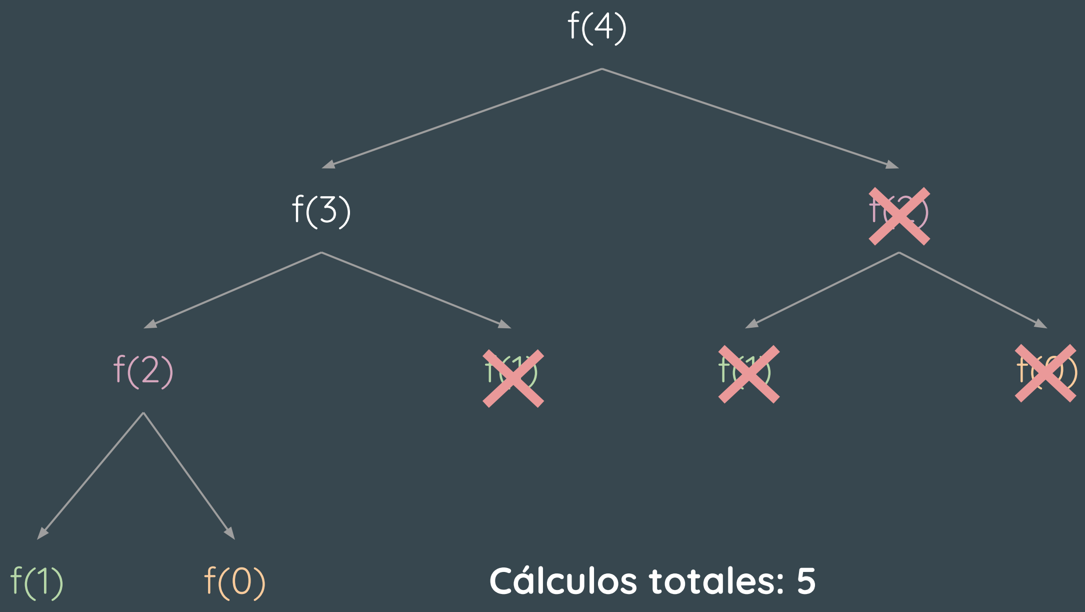
```python
def fibo(n):
  if n == 0:
    return 0
  elif n == 1:
    return 1
  else:
    return fibo(n-1) + fibo(n-2)
```



Cálculos totales: 9

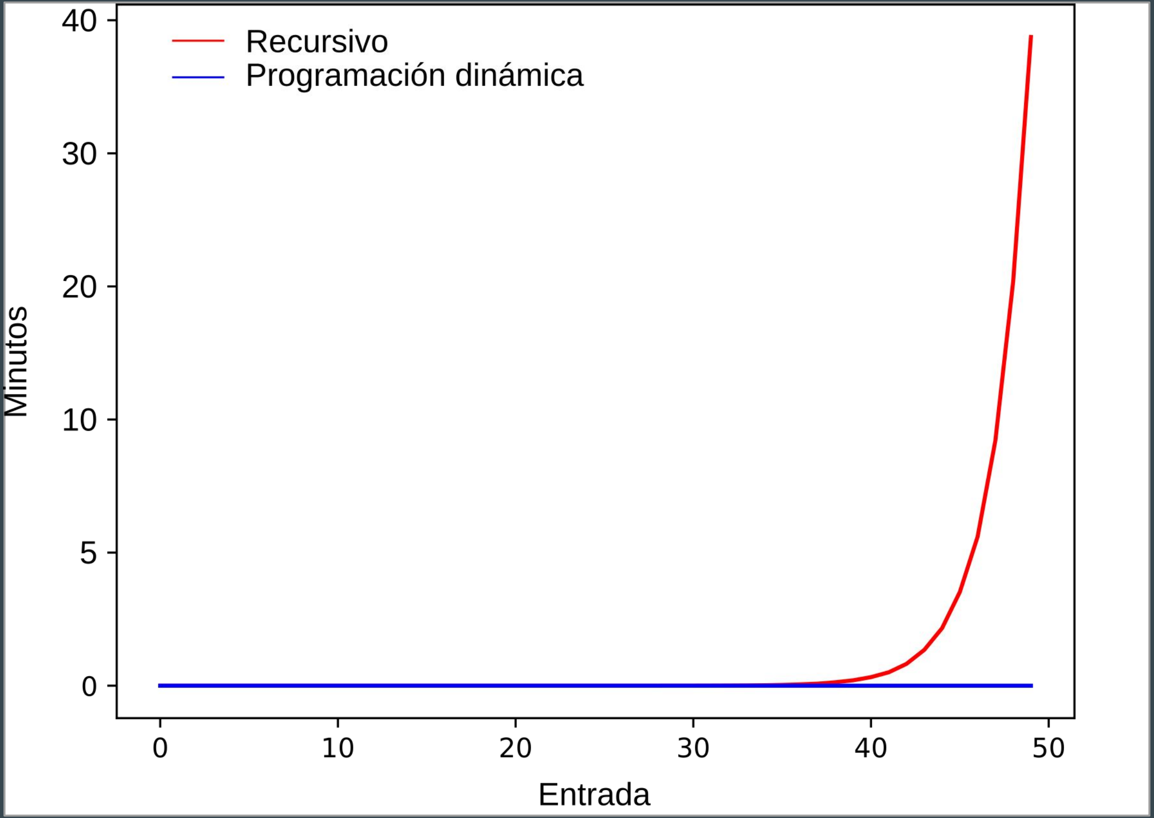
Redefiniendo Fibonacci recursivo

```
def fibo_pd(n, calc = {}):
  if n in calc:
    return calc[n]
  elif n == 1:
    calc[n] = 1
    return 1
  elif n == 0:
    calc[n] = 0
    return 0
  else:
    res = fibo_pd(n-1, calc) +
          fibo_pd(n-2, calc)
    calc[n] = res
    return res
```



Cálculos totales: 5

Fibonacci: recursivo vs programación dinámica



Programación dinámica: problema de la mochila

- La clave de la programación dinámica está en determinar cuál es el mejor subproblema con el cual trabajar.
- En el caso del problema de la mochila, una alternativa sería almacenar la mejor solución que se vaya encontrando para cada peso posible.
- Por ejemplo, si hay 4 kilos restantes, las 2 opciones posibles son "Item5", con valor de 4, e "Item6+Item7", con un valor de 0.6.
- Si ya se sabe que la primer opción es mejor, nunca se considera la segunda.

Programación dinámica: problema de la mochila

Peso	Items	Valor
0	[ ]	0
1	[ ]	0
2	[ ]	0
3	[ ]	0
4	[ ]	0
5	[ ]	0
6	[ ]	0
7	[ ]	0
8	[ ]	0
...	[ ]	0

Item	Peso	Valor
Item7	1	0.1
Item6	2	0.5
Item5	4	1
Item4	6	6
Item3	10	9
Item2	9	11
Item1	22	19

Programación dinámica: problema de la mochila

Peso	Items	Valor
0	[ ]	0
1	[7]	0.1
2	[7]	0.1
3	[7]	0.1
4	[7]	0.1
5	[7]	0.1
6	[7]	0.1
7	[7]	0.1
8	[7]	0.1
…	[7]	0.1

Item	Peso	Valor
Item7	1	0.1
Item6	2	0.5
Item5	4	1
Item4	6	6
Item3	10	9
Item2	9	11
Item1	22	19

Programación dinámica: problema de la mochila

Peso	Items	Valor
0	[ ]	0
1	[7]	0.1
2	[6]	0.5
3	[6, 7]	0.6
4	[6, 7]	0.6
5	[6, 7]	0.6
6	[6, 7]	0.6
7	[6, 7]	0.6
8	[6, 7]	0.6
…	…	…

Item	Peso	Valor
Item7	1	0.1
Item6	2	0.5
Item5	4	1
Item4	6	6
Item3	10	9
Item2	9	11
Item1	22	19

Programación dinámica: problema de la mochila

Peso	Items	Valor
0	[ ]	0
1	[7]	0.1
2	[6]	0.5
3	[6, 7]	0.6
4	[5]	1
5	[5, 7]	1.1
6	[5, 6]	1.5
7	[5, 6, 7]	1.6
8	[5, 6, 7]	1.6
…	…	…

Item	Peso	Valor
Item7	1	0.1
Item6	2	0.5
Item5	4	1
Item4	6	6
Item3	10	9
Item2	9	11
Item1	22	19

Programación dinámica: pseudocódigo

```
items = [Item1, Item2, …]
pesos = [22, 9, …]
valores = [19, 11, …]
peso_maximo = 25
por cada item:
  por cada peso entre pesos[item] y peso_max:
    peso_complementario = peso - peso(item) # Buscamos el valor ya calculado
    si valor_mochila[peso] < valor(item) + valor_mochila(peso_complementario):
       valor_mochila[peso] = valor(item) + valor_mochila(peso_complementario)
    sino, si valor_mochila[peso] < valor_mochila[peso-1]
       valor_mochila[peso] = valor(item)
```

Algoritmos Ávidos (*greedy*)

- Son algoritmos de optimización que, en cada etapa, seleccionan la solución óptima, con la idea de que la suma de estos **óptimos locales** conduzcan a un **óptimo global**.
- Al tomar la decisión, no se consideran consecuencias futuras, sólo se seleccionará el que sea más adecuado en cada etapa.
- La selección de la solución óptima local se basa en **heurísticas**.

Heurísticas

- Son técnicas que emplean un método práctico para lograr una solución rápida a un problema.
- Estas técnicas se apoyan en la experiencias y criterio del desarrollador.
- Permiten **acelerar los procesos** y alcanzar una solución **satisfactoria**, pero **no aseguran encontrar la solución óptima**.
- Por ejemplo: elegir el ítem de mayor valor o el de menor peso.

Componentes necesarios

- Entradas (candidatos)
- Una función de selección que determine el **candidato idóneo** para agregar a la solución.
- Restricción/es.
- Función a maximizar

Algoritmos ávidos: pseudocódigo

```
items = [Item1, Item2, …]
pesos = [22, 9, …]
valores = [19, 11, …]
peso_restante = 25 # Empezamos con la mochila vacía

mientras peso_restante > 0 y haya ítems en lista:
  seleccionar el mejor candidato en base a heuristica
  si peso_candidato <= peso_restante
    agregar candidato a solucion
    disminuir peso restante
  quitar candidato de items, pesos y valores
devolver solucion
```

Algoritmos ávidos: código

```
def mochila_mayor_valor(items, pesos, valores, peso_restante):
  calculados = []
  solucion = [“”, 0 ,0]
  while peso_restante > 0 and len(calculados) < len(items):
    candidato = mayor_valor(valores, calculados)
    if peso_restante >= pesos[candidato]:
      solucion[0] = solucion[0] + “;” + items[candidato]
      solucion[1] += pesos[candidato]
      solucion[2] += valores[candidato]
      peso_restante -= pesos[candidato]
    calculados.append(candidato)
  return solucion
```

Algoritmos ávidos: código

```
def mayor_valor(valores, calculados):
  candidato = None
  valor_candidato = float("-inf")
  for i in range (0, len(valores)):
    if valores[i] > valor_candidato and i not in calculados:
      valor_candidato = valores[i]
      candidato = i
  return candidato
```

Bibliografía

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms. MIT press. Capítulos 15 y 16.
- Guerequeta, R., & Vallecillo, A. (2019). Técnicas de diseño de algoritmos. Segunda edición. Capítulos 4, 5 y 7