Algoritmos Ávidos y Programación Dinámica

...

Algoritmos y Estructuras de Datos
2023

UNRaf UNIVERSIDAD NACIONAL DE RAFAELA

Dada una mochila con 25 kg de capacidad, ¿cuál es el valor máximo que se puede cargar?

Item	Item 0	Item 1	Item 2	Item 3	Item 4	Item 5	Item 6
Peso	22	9	10	6	4	2	1
Valor	19	11	9	6	1	0,5	0,1

¿Cómo solucionar un problema?

1. Utilizar **fuerza bruta**

   En este caso, tenemos 7 elementos, cada uno puede estar o no en la mochila, por lo que el número de soluciones prometedoras es $2^7 = 128$ (crece exponencialmente!)

2. Utilizar **ramificación y poda**
3. Utilizar **algoritmos ávidos**
4. Utilizar **programación dinámica**

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

Pseudocódigo Algoritmo Ávidos

```
while not len(candidatos) != 0 and limite == False:
    x=SeleccionarCandidato(candidatos)
    if EsValida(x,solucion)
        incluir(x,solucion)
    quitar(x, candidatos)
    if LimitadoAlcanzado(solucion, limite):
        limite = True
return solucion
```

Codificando el problema de la mochila (una de las tantas soluciones)

```
def mochila_mayor_valor(peso_restante, pesos, valores):
  calculados = []
  resultados = []
  valor_total = 0
  while peso_restante > 0 and len(calculados) < len(pesos):
    candidato = seleccionar_mayor_valor(pesos, valores, calculados)
    if peso_restante >= pesos[candidato]:
      resultados.append(candidato)
      valor_total += valores[candidato]
      peso_restante -= pesos[candidato]
    calculados.append(candidato)
  return valor_total, resultados
```

Codificando el problema de la mochila (una de las tantas soluciones)

```
def mochila_mayor_valor(peso_restante, pesos, valores):
  calculados = []
  resultados = []
  valor_total = 0
  while peso_restante > 0 and len(calculados) < len(pesos):
    candidato = seleccionar_mayor_valor(pesos, valores, calculados)
    if peso_restante >= pesos[candidato]:
      resultados.append(candidato)
      valor_total += valores[candidato]
      peso_restante -= pesos[candidato]
    calculados.append(candidato)
  return valor_total, resultados
```

¿Se alcanzó el límite?

¿Es válida?

Agregar a solución

Quitar de candidatos

Codificando el problema de la mochila (una de las tantas soluciones)

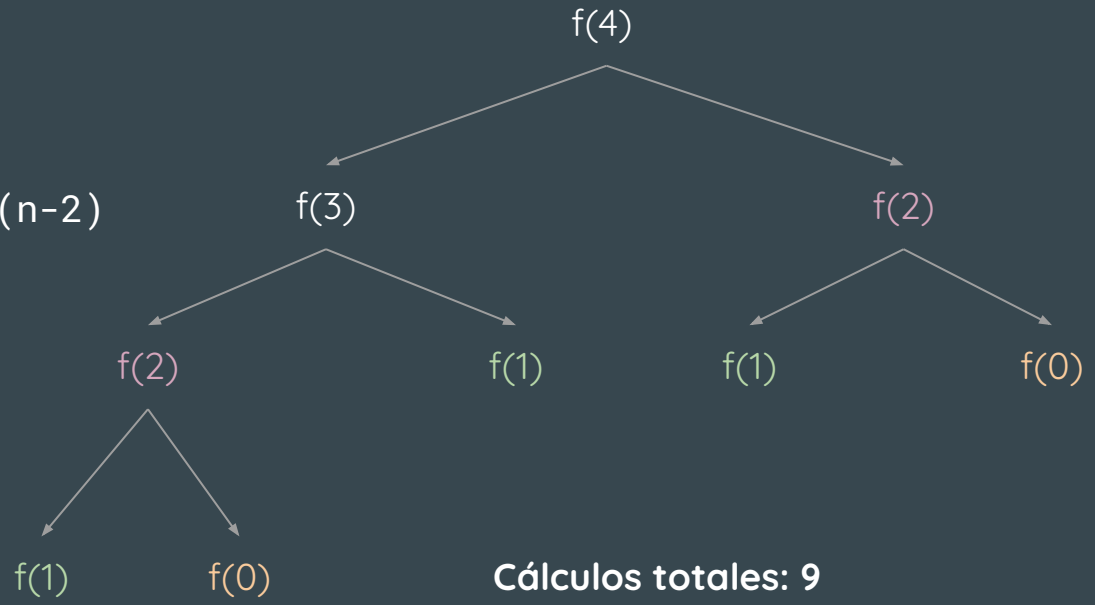
```
def seleccionar_mayor_valor(pesos, valores, calculados):
  candidato = None
  peso_candidato = 0
  valor_candidato = float("-inf")
  for i in range (0, len(pesos)):
    if valores[i] > valor_candidato and i not in calculados:
      peso_candidato = pesos[i]
      valor_candidato = valores[i]
   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

Programación Dinámica

- Las soluciones recursivas suelen ser ineficientes, ya que resuelven el mismo problema en repetidas ocasiones.
- La **programación dinámica** plantea una alternativa para **optimizar el número de cálculos requeridos** para alcanzar una solución.
- Como los algoritmos ávidos, parte del principio de “divide y vencerás”
- La solución al problema se alcanza al evaluar todas las soluciones a los subproblemas, por lo que **siempre se alcanza la solución óptima**.

Recordando Fibonacci recursivo

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



Cálculos totales: 9

Programación Dinámica

- Una forma implementar una solución de programación dinámica es utilizar la estrategia “de arriba hacia abajo” (*top-down*)
- Se comienza con el problema general y se lo divide en subproblemas.
- Cada vez que un subproblema es resuelto, los resultados **se almacenan** para poder ser reutilizados más adelante.
- Este proceso se conoce como **memoización**.

Programación Dinámica

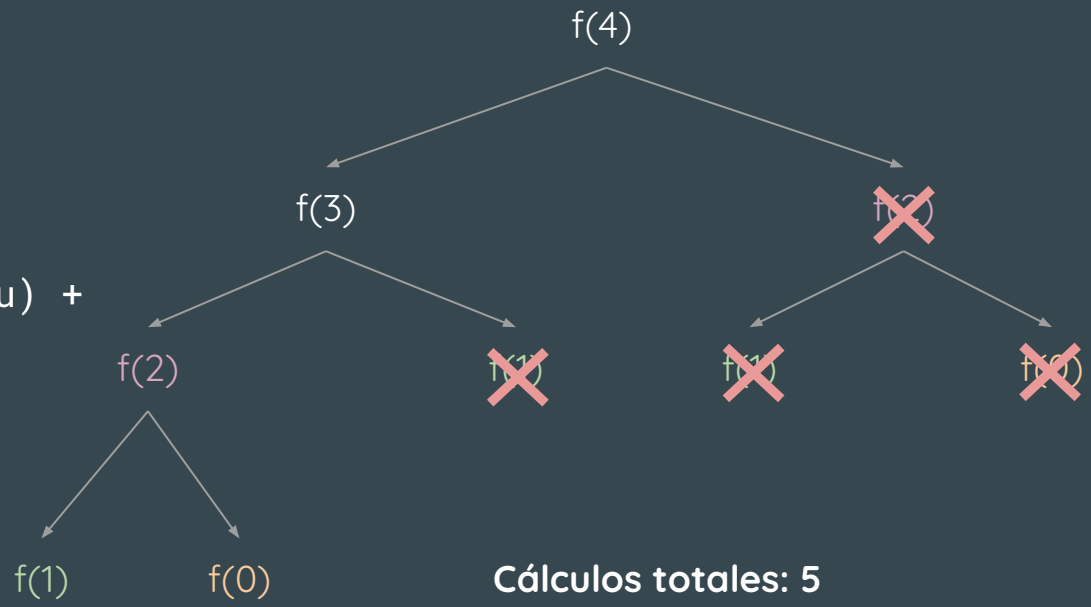
- Una alternativa es el uso de la estrategia “de abajo hacia arriba” (*bottom-up*).
- Para ello, se comienza desde el subproblema más pequeño, resolviendo y guardando los resultados, para ser utilizados más adelante.
- Requiere saber de antemano cuales son los problemas más pequeños y todos los subproblemas que los necesitan.

Memoización

- Es una técnica de optimización para disminuir el número de cálculos realizados.
- En cada etapa, se consulta si un subproblema ya fue resuelto (por ej: si f(2) ya fue calculado).
    - 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

Redefiniendo Fibonacci recursivo

```
def fibo(n, calcu = []):
  if len(calcu) == 0:
    calcu = [-1] * (n+1)
  if calcu[n] != -1:
    return calcu[n]
  elif n > 1:
    result = fibo(n-1, calcu) +
fibo(n-2,calcu)
    calcu[n] = result
    return result
  elif n == 1:
    calcu[n] = 1
    return 1
  else:
    calcu[n] = 0
    return 0
```

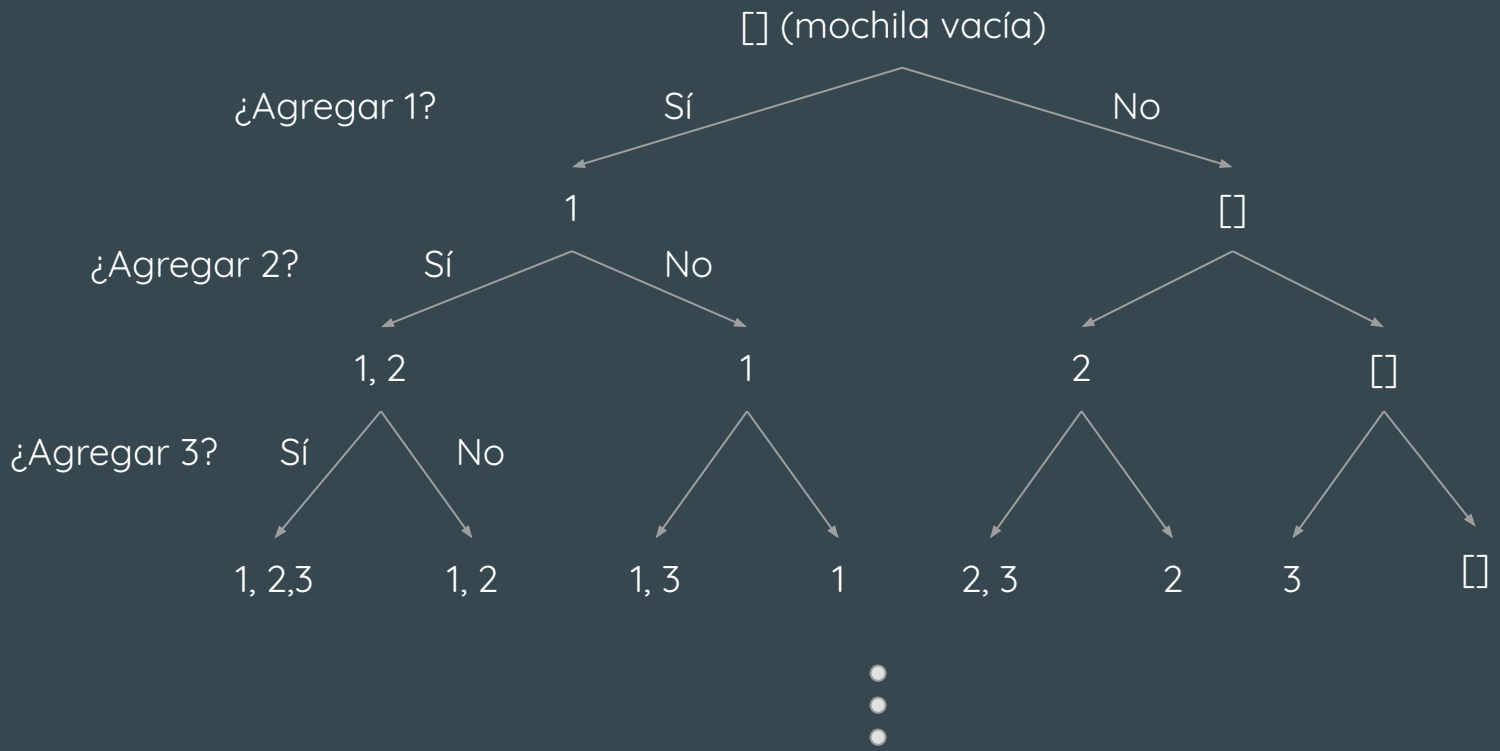


Cálculos totales: 5

Solución recursiva para el problema de la mochila?

- Para cada candidato se puede plantear las preguntas “¿entra en la mochila dado el espacio que tengo?” y “¿cuánto valor tengo al agregar y al no agregar el elemento?”
- En este planteo, el resultado buscado es la suma de valores más alta.
- El valor y peso de cada elemento no varía en cada etapa.
- Lo que sí varía es el peso restante en la mochila (dependiendo si se agregó o no un candidato) y el número de candidatos a evaluar.

Solución recursiva



Pseudocódigo

- Se almacena el lugar restante en la mochila en la variable ***peso_restante*** y el elemento que se está visitando con la variable ***indice***.
- Caso base: si no hay más lugar en la mochila (***peso_restante***) o si no hay más elementos que recorrer (***indice*** >= len(pesos)), devolver 0.
- Si no, si el peso del objeto actual es mayor que el ***peso_restante***, no considerarlo
- Si no, calcular el ***valor*** incorporando el elemento (cambia ***peso_restante***) y el valor si no se incorpora (***peso_restante*** se mantiene). Elegir el mayor

Código

```
def mochila_recursivo(pesos, valores, peso_restante, indice = 0):
  if peso_restante == 0 or len(valores) == indice:
    resultado = 0

  elif pesos[indice] > peso_restante:
    resultado = mochila_recursivo(pesos, valores, peso_restante, indice+1)

  else:
    result_si = valores[indice] + mochila_recursivo(pesos, valores,
peso_restante-pesos[indice], indice+1)
    result_no = mochila_recursivo(pesos, valores, peso_restante, indice+1)
    if result_si > result_no:
      resultado = result_si
    else:
      resultado = result_no
  return resultado
```

Código

```
def mochila_recursivo(pesos, valores, peso_restante, indice = 0):
  if peso_restante == 0 or len(valores) == indice:
    resultado = 0

  elif pesos[indice] > peso_restante:
    resultado = mochila_recursivo(pesos, valores, peso_restante, indice+1)

  else:
    result_si = valores[indice] + mochila_recursivo(pesos, valores,
peso_restante-pesos[indice], indice+1)
    result_no = mochila_recursivo(pesos, valores, peso_restante, indice+1)
    if result_si > result_no:
      resultado = result_si
    else:
      resultado = result_no
  return resultado
```

Caso base

"Si no entra en la mochila"

Suma el valor del candidato al valor total y disminuye el peso restante

Memoizando el problema de la mochila

- Como se vio en el árbol, la solución recursiva considera todas las opciones, agregando o no todos los candidatos.
- En cada llamada, se necesita el elemento a evaluar y el peso restante y se obtiene un valor total de los elementos en la mochila.
- El objetivo de la memoización es evitar calcular combinaciones que ya se hayan calculado

Memoizando el problema de la mochila

- Cuando la mochila solo cuenta con el ítem 2, el peso restante es 25 - 10 = 15.
- El código va a computar la incorporación de todos los elementos que siguen al ítem 2 (3, 4, 5 y 6) con la variable peso_restante=15.
- Cuando la mochila cuenta con los ítem 3 y 4, el peso restante es 25 – 6 - 4 = 15.
- El código va a computar la incorporación de todos los elementos que siguen al ítem 4 (5 y 6) con la variable peso_restante=15.
- Para optimizar un poco esta solución se puede almacenar el resultado (valor) para cada par de candidato y peso_restante ya calculado.

Código

```
def mochila_memoizado(pesos, valores, peso_restante, indice = 0, calc = {}):
  if indice in calc and peso_restante in calc[indice]:
    resultado = calc[indice][peso_restante]

# Muy similar a la función recursiva, pero con un parámetro más

  elif pesos[indice] > peso_restante:
    resultado = mochila_recursivo(pesos, valores, peso_restante, indice+1, calc)

  if indice not in calc:
    calc[indice] = {}
  calc[indice][peso_restante] = resultado

  return resultado
```

Diferencias

Algoritmos Ávidos	Programación Dinámica
Utiliza la técnica de “divide y vencerás”.	Utiliza la técnica de “divide y vencerás”.
Se evalúa un subconjunto de soluciones que se consideran óptimas.	Se evalúan todos los subconjuntos de soluciones.
La optimización se basa en el uso de heurísticas.	La optimización se basa en el almacenamiento de resultados previos.
Se alcanza una solución satisfactoria, pero no siempre óptima.	Se alcanza la solución óptima
En general, requiere menos cálculos.	La optimización depende de cuántos resultados se puedan reutilizar

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 y 5