

Algoritmos y Estructuras de Datos II

Backtracking

18 de mayo de 2016

Clase de hoy

- 1 Repaso
 - Divide y vencerás
 - Algoritmos voraces
- 2 Backtracking
 - Forma general de algoritmos voraces
 - ¿Y cuando no hay un buen criterio de selección?
 - Problema de la moneda
 - Problema de la mochila
 - Camino de costo mínimo entre todo par de vértices
- 3 Conclusiones

Repaso

- cómo vs. qué
- 3 partes
 - 1 análisis de algoritmos
 - 2 tipos de datos
 - 3 técnicas de resolución de problemas
 - divide y vencerás
 - algoritmos voraces
 - [backtracking](#)
 - programación dinámica
 - recorrida de grafos

Divide y vencerás

- Ordenación por intercalación, $\mathcal{O}(n \log n)$.
- Ordenación rápida, $\mathcal{O}(n \log n)$ en la práctica.
- Búsqueda binaria, $\mathcal{O}(\log n)$.
- Exponenciación, $\mathcal{O}(\log n)$ número de multiplicaciones (n es el exponente).
- Multiplicación de grandes números, $\mathcal{O}(n^{\log_2 3})$ donde n es el número de dígitos.

Algoritmos voraces

- Problema de la moneda
 - $\mathcal{O}(n)$ donde n es el número de denominaciones si están ordenadas.
 - no anda para cualquier conjunto de denominaciones
- Problema de la mochila
 - $\mathcal{O}(n)$ donde n es el número de objetos si están ordenados según sus cocientes v_i/w_i .
 - sólo anda para objetos fraccionables
- Problema del árbol generador de costo mínimo
 - Prim es $\mathcal{O}(|V|^2)$ donde V es el conjunto de vértices. Se puede mejorar con implementaciones ingeniosas.
 - Kruskal es $\mathcal{O}(|A| \log |V|)$ donde A es el conjunto de aristas.
 - Boruvka es $\mathcal{O}(|A| \log |V|)$.
- Problema del camino de costo mínimo
 - Dijkstra es $\mathcal{O}(|V|^2)$.

Forma general de algoritmos voraces

```
fun greedy(C) ret S
    {C: conjunto de candidatos, S: solución a construir}
    S := {}
    do S no es solución → c := seleccionar de C
        C := C - {c}
        if S ∪ {c} es factible → S := S ∪ {c} fi
    od
end fun
```

- Ser solución y ser factible no tienen en cuenta optimalidad.
- Optimalidad depende totalmente del criterio de selección.

¿Y cuando no hay un buen criterio de selección?

- A veces no hay un criterio de selección que garantice optimalidad.
- Por ejemplo:
 - Problema de la moneda para conjuntos de denominaciones arbitrarios.
 - Problema de la mochila para objetos no fraccionables.
- En este caso, si se elige un fragmento de solución puede ser necesario “volver hacia atrás” (**backtrack**) sobre esa elección e intentar otro fragmento.
- En la práctica, estamos hablando de considerar todas las selecciones posibles e intentar cada una de ellas para saber cuál de ellas conduce a la solución óptima.

Problema de la moneda

- Sean d_1, d_2, \dots, d_n las denominaciones de las monedas (todas mayores que 0),
- no se asume que estén ordenadas,
- se dispone de una cantidad infinita de monedas de cada denominación,
- se desea pagar un monto k de manera exacta,
- utilizando el **menor número de monedas posibles**.
- Vimos que el algoritmo voraz puede no funcionar para ciertos conjuntos de denominaciones.
- Daremos un algoritmo consistente en considerar todas las combinaciones de monedas posibles.

Simplificación y generalización

- Simplificamos el problema:
 - sólo nos interesa por ahora hallar el menor número de monedas necesario,
 - no nos interesa saber cuáles son esas monedas.
- Generalizamos el problema:
 - Sean $0 \leq i \leq n$ y $0 \leq j \leq k$,
 - definimos $m(i, j) =$ “menor número de monedas necesarias para pagar exactamente el monto j con denominaciones d_1, d_2, \dots, d_i .”
 - La solución del problema original se obtiene calculando $m(n, k)$.

Definiendo $m(i, j)$

Caso $j = 0$

- Recordemos que $m(i, j) =$ “menor número de monedas necesarias para pagar exactamente el monto j con denominaciones d_1, d_2, \dots, d_i .”
- $m(i, 0) = 0$

Definiendo $m(i, j)$

Caso $j > 0$ y $i = 0$

- Recordemos que $m(i, j) =$ “menor número de monedas necesarias para pagar exactamente el monto j con denominaciones d_1, d_2, \dots, d_i .”
- $m(i, 0) = 0$,
- $j > 0 \Rightarrow m(0, j) = \infty$, ya que no hay manera posible de pagar el monto

Definiendo $m(i, j)$

Caso $i > 0$ y $d_i > j > 0$

- Recordemos que $m(i, j) =$ “menor número de monedas necesarias para pagar exactamente el monto j con denominaciones d_1, d_2, \dots, d_i .”
- $m(i, 0) = 0,$
- $j > 0 \Rightarrow m(0, j) = \infty,$
- $i > 0 \wedge d_i > j > 0 \Rightarrow m(i, j) = m(i - 1, j),$ ya que no se pueden usar monedas de denominación d_i , es como si no estuvieran disponibles

Definiendo $m(i, j)$

Caso $i > 0$ y $j \geq d_i$

- Recordemos que $m(i, j) =$ “menor número de monedas necesarias para pagar exactamente el monto j con denominaciones d_1, d_2, \dots, d_i .”
- $m(i, 0) = 0$,
- $j > 0 \Rightarrow m(0, j) = \infty$,
- $i > 0 \wedge d_i > j > 0 \Rightarrow m(i, j) = m(i - 1, j)$,
- si $j \geq d_i$ hay dos posibilidades
 - la solución óptima no usa monedas de denominación d_i
 - $m(i, j) = m(i - 1, j)$
 - la solución óptima usa una o más monedas de denominación d_i
 - $m(i, j) = 1 + m(i, j - d_i)$

Definición recursiva de $m(i, j)$

Conclusión de estas últimas filminas:

$$m(i, j) = \begin{cases} 0 & j = 0 \\ \infty & j > 0 \wedge i = 0 \\ m(i - 1, j) & d_i > j > 0 \wedge i > 0 \\ \min(m(i - 1, j), 1 + m(i, j - d_i)) & j \geq d_i > 0 \wedge i > 0 \end{cases}$$

Decisión que determina esta definición: ¿usamos monedas de denominación d_i o no?

Otras posibles definiciones que usan backtracking

Considerando el número exacto de monedas de denominación d_i

$$m(i, j) = \begin{cases} 0 & j = 0 \\ \infty & j > 0 \wedge i = 0 \\ \min_{q \in \{0, 1, \dots, j \div d_i\}} (q + m(i - 1, j - q * d_i)) & j > 0 \wedge i > 0 \end{cases}$$

Acá estamos considerando la posibilidad de usar 0 monedas ($q = 0$) de denominación d_i , 1 moneda ($q = 1$) de denominación d_i , etc. De todas esas posibilidades se elige la que minimice el número total de monedas.

Decisión que determina esta definición: ¿cuántas monedas de denominación d_i usamos?

Otras posibles definiciones que usan backtracking

Considerando cuál moneda de las disponibles se usa

$$m(i, j) = \begin{cases} 0 & j = 0 \\ 1 + \min_{i' \in \{1, 2, \dots, i\} \wedge d_{i'} \leq j} (m(i', j - d_{i'})) & j > 0 \end{cases}$$

Acá estamos considerando la posibilidad de usar 1 moneda de denominación d_i ($i' = i$), 1 moneda de denominación d_{i-1} ($i' = i - 1$), etc. De todas esas posibilidades se elige la que minimice el número total de monedas. Para evitar cálculos repetidos, se restringe la búsqueda a monedas de índice menor o igual a los ya utilizados.

Decisión que determina esta definición: de las monedas de que disponemos, ¿cuál usamos?

Primera definición recursiva de $m(i, j)$

$$m(i, j) = \begin{cases} 0 & j = 0 \\ \infty & j > 0 \wedge i = 0 \\ m(i - 1, j) & d_i > j > 0 \wedge i > 0 \\ \min(m(i - 1, j), 1 + m(i, j - d_i)) & j \geq d_i > 0 \wedge i > 0 \end{cases}$$

Primera solución, ahora en pseudocódigo

```
fun cambio(d:array[1..n] of nat, i,j: nat) ret r: nat
  if j=0 then r:= 0
  else if i = 0 then r:=  $\infty$ 
    else if d[i] > j then r:= cambio(d,i-1,j)
      else r:= min(cambio(d,i-1,j),1+cambio(d,i,j-d[i]))
    fi
  fi
fi
end fun
```

Segunda definición recursiva de $m(i, j)$

$$m(i, j) = \begin{cases} 0 & j = 0 \\ \infty & j > 0 \wedge i = 0 \\ \min_{q \in \{0, 1, \dots, j \div d_i\}} (q + m(i - 1, j - q * d_i)) & j > 0 \wedge i > 0 \end{cases}$$

Segunda solución, ahora en pseudocódigo

```
fun cambio(d:array[1..n] of nat, i,j: nat) ret r: nat
  if j=0 then r:= 0
  else if i = 0 then r:=  $\infty$ 
    else r:= cambio(d,i-1,j)
      for q:= 1 to j  $\div$  d[i] do
        r:= min(r,q+cambio(d,i-1,j-q*d[i]))
      od
    fi
  fi
end fun
```

Tercera definición recursiva de $m(i, j)$

$$m(i, j) = \begin{cases} 0 & j = 0 \\ 1 + \min_{i' \in \{1, 2, \dots, i\} \wedge d_{i'} \leq j} (m(i', j - d_{i'})) & j > 0 \end{cases}$$

Tercera solución, ahora en pseudocódigo

```
fun cambio(d:array[1..n] of nat, i,j: nat) ret r: nat
  if j=0 then r:= 0
  else r:=  $\infty$ 
    for i':= 1 to i do
      if d[i']  $\leq$  j then r:= min(r,cambio(d,i',j-d[i'])) fi
    od
    r:= r + 1
  fi
end fun
```

Otras posibilidades

- No son éstas las únicas formas de resolver el problema usando backtracking.
- Podríamos definir, por ejemplo,
 - $m(i, j) =$ “menor número de monedas necesarias para pagar exactamente el monto j con denominaciones $d_{i+1}, d_{i+2}, \dots, d_n$.”
- Obtendríamos, entre otras posibles definiciones recursivas,

$$m(i, j) = \begin{cases} 0 & j = 0 \\ \infty & j > 0 \wedge i = n \\ m(i + 1, j) & d_i > j > 0 \wedge i < n \\ \min(m(i + 1, j), 1 + m(i, j - d_i)) & j \geq d_i > 0 \wedge i < n \end{cases}$$

- Para resolver el problema original se calcula $m(0, k)$.

Primera solución, pero ¡Queremos las monedas!

```
fun cambio(d:array[1..n] of nat, i,j: nat) ret r: list of nat
  var r1, r2 : list of nat
  if j=0 then r:= [ ]
  else if i = 0 then r:= una lista infinita
    else if d[i] > j then r:= cambio(d,i-1,j)
    else r1 := cambio(d,i-1,j)
      r2 := cambio(d,i,j-d[i]) < d[i]
      if |r1| ≤ |r2| then r:= r1 else r:= r2 fi
    fi
  fi
end fun
```

|x| es la longitud de x
llamada principal: cambio(d,n,k)

Problema de la mochila

- Tenemos una mochila de capacidad W .
- Tenemos n objetos **no fraccionables** de valor v_1, v_2, \dots, v_n y peso w_1, w_2, \dots, w_n .
- Se quiere encontrar la mejor selección de objetos para llevar en la mochila.
- Por mejor selección se entiende aquélla que totaliza **el mayor valor posible** sin que su peso exceda la capacidad W de la mochila.

Simplificación y generalización

- Simplificamos el problema:
 - sólo nos interesa por ahora hallar el mayor valor posible sin exceder la capacidad de la mochila,
 - no nos interesa saber cuáles son los objetos que alcanzan ese máximo.
- Generalizamos el problema:
 - Sean $0 \leq i \leq n$ y $0 \leq j \leq W$,
 - definimos $m(i, j)$ = “mayor valor alcanzable sin exceder la capacidad j con objetos $1, 2, \dots, i$.”
 - La solución del problema original se obtiene calculando $m(n, W)$.

Definición recursiva de $m(i, j)$

$$m(i, j) = \begin{cases} 0 & j = 0 \\ 0 & j > 0 \wedge i = 0 \\ m(i-1, j) & w_i > j > 0 \wedge i > 0 \\ \max(m(i-1, j), v_i + m(i-1, j - w_i)) & j \geq w_i > 0 \wedge i > 0 \end{cases}$$

Decisión que determina esta definición: ¿colocamos el objeto i en la mochila o no?

Otras posibilidades

- Ofrece las mismas variantes que en el problema de la moneda,
- el pasaje a pseudocódigo es similar,
- la incorporación de información con los objetos que van en la mochila es también parecido.

Problema del camino de costo mínimo

Entre todo par de vértices

- Tenemos un grafo dirigido $G = (V, A)$,
- con costos no negativos en las aristas,
- se quiere encontrar, para cada par de vértices, el camino de menor costo que los une.
- Se asume $V = \{1, \dots, n\}$

Simplificación y generalización

- Simplificamos el problema:
 - sólo nos interesa por ahora hallar el costo de cada uno de los caminos de costo mínimo.
 - no nos interesa saber cuáles son los caminos que alcanzan ese mínimo.
- Generalizamos el problema:
 - Sean $1 \leq i, j \leq n$ y $0 \leq k \leq n$,
 - definimos $m_k(i, j) =$ “menor costo posible para caminos de i a j cuyos vértices intermedios se encuentran en el conjunto $\{1, \dots, k\}$.”
 - La solución del problema original se obtiene calculando $m_n(i, j)$ para el par i (origen) y j (destino) que se desea.

Definición recursiva de $m_k(i, j)$

$$m_k(i, j) = \begin{cases} L[i, j] & k = 0 \\ \min(m_{k-1}(i, j), m_{k-1}(i, k) + m_{k-1}(k, j)) & k \geq 1 \end{cases}$$

donde $L[i, j]$ es el costo de la arista que va de i a j , o infinito si no hay tal arista.

Decisión que determina esta definición: ¿pasamos por el vértice k o no?

Conclusiones

- Hemos visto soluciones a tres problemas.
- En general, muy ineficiente.
- Por ejemplo, para el problema de la moneda, si queremos pagar el monto 90 con nuestros billetes con denominaciones 1, 5 y 10,
 - $m(3,90)$ llama a $m(2,90)$ y $m(3,80)$,
 - $m(2,90)$ llama a $m(1,90)$ y $m(2,85)$,
 - $m(2,85)$ llama a $m(1,85)$ y **$m(2,80)$** ,
 - $m(3,80)$ llama a **$m(2,80)$** y $m(3,70)$.
- Se ve que $m(2,80)$ se calcula 2 veces.
- y muchos otros llamados se repiten, incluso varias veces.
- Esto vuelve los algoritmos exponenciales en el peor caso.