Técnicas de construção de algoritmos

- Força bruta
- Divisão e conquista
- Abordagem *greedy*
- Programação dinâmica

Força bruta

Também conhecida como abordagem ingénua

A solução é calculada da maneira mais directa possível, sem recorrer a qualquer técnica para diminuir o número de operações feitas

Inclui os algoritmos de geração e teste

Pode ser útil:

- Quando a dimensão dos problemas a tratar é pequena
- Para chegar a uma primeira implementação
- Para ajudar a ter confiança noutros algoritmos, cuja correcção é mais difícil de estabelecer

Divisão e conquista

Abordagem

1. Dividir o problema em subproblemas (mais pequenos)
2. Conquistar, resolvendo os subproblemas
3. Combinar as soluções dos subproblemas para obter a solução do problema original

Exemplos

- *Merge sort*
- *Quicksort*

Abordagem *greedy*

Aplica-se, em geral, para resolver problemas de optimização

- Nos problemas de optimização procura-se uma solução que é melhor, de acordo com algum critério
  - O maior lucro
  - O menor custo
  - A que requer menos operações
  - ...

A solução é construída fazendo, em cada momento, a escolha que parece ser a melhor

Nem todos os problemas de optimização podem ser resolvidos através de um algoritmo *greedy*

Também conhecidos como algoritmos gananciosos, ansiosos, gulosos, ...

Programação dinâmica

Método usado na construção de soluções iterativas para problemas cuja solução recursiva tem uma complexidade elevada (exponencial, em geral)

Aplica-se, normalmente, a problemas de optimização

- Um problema de optimização é um problema em que se procura minimizar ou maximizar algum valor associado às suas soluções
- Uma solução com essa característica diz-se óptima
- Pode haver várias soluções óptimas

Venda de varas a retalho

Uma empresa compra varas de aço, corta-as e vende-as aos pedaços

O preço de venda de cada pedaço depende do seu comprimento

Problema

Como cortar uma vara de comprimento n de forma a maximizar o seu valor de venda?

Comprimento i	1	2	3	4	5	6	7	8	9	10
Preço p_i	1	5	7	11	11	17	20	20	24	27

Corte de varas

Caracterização de uma solução óptima (1)

Soluções possíveis, para uma vara de comprimento 10

- Um corte de comprimento 1, mais as soluções para uma vara de comprimento 9
- Um corte de comprimento 2, mais as soluções para uma vara de comprimento 8
- Um corte de comprimento 3, mais as soluções para uma vara de comprimento 7

  ...
- Um corte de comprimento 9, mais as soluções para uma vara de comprimento 1
- Um corte de comprimento 10, mais as soluções para uma uma vara de comprimento 0

Qual a melhor?

Corte de varas

Caracterização de uma solução óptima (2)

Sejam os tamanhos dos cortes possíveis

$$1, 2, \ldots, n$$

com preços

$$p_1, p_2, \ldots, p_n$$

O valor máximo de venda de uma vara de comprimento n é o máximo que se obtém

- fazendo um corte inicial de comprimento $1 \leq i \leq n$, de valor p_i, somado com
- o valor máximo de venda de uma vara de comprimento $n - i$

Corte de varas

Função recursiva

Corte de uma vara de comprimento n

Tamanho dos cortes: $i = 1, \ldots, n$
Preços: $P = (p_1\, p_2\, \ldots\, p_n)$

$v_P(0..n)$: função t.q. $v_P(l)$ é o valor máximo de venda de uma vara de comprimento l, dados os preços P

$$v_P(l) = \begin{cases} 0 & \text{se} \quad l = 0 \\ \max\limits_{1 \le i \le l} \{p_i + v_P(l - i)\} & \text{se} \quad l > 0 \end{cases}$$

Chamada inicial da função
Valor máximo de venda de uma vara completa: $v_P(n)$

Corte de varas

Implementação recursiva

CUT-ROD(p, l)

```
if l = 0 then
  return 0
q <- −∞
for i <- 1 to l do
  q <- max(q, p[i] + CUT-ROD(p, l - i))
return q
```

Argumentos

- p Preços das varas de comprimentos $\{1, 2, \ldots, n\}$
- l Comprimento da vara a cortar

Chamada inicial da função: CUT-ROD(p, n)

Corte de varas

Alguns números

Número de cortes possíveis

$$2^{n-1}$$

Exemplo ($n = 4$)

$4 \quad 1+3 \quad 2+2 \quad 3+1$

$1+1+2 \quad 1+2+1 \quad 2+1+1 \quad 1+1+1+1$

Número de cortes distintos possíveis

$$O\left(\frac{e^{\pi\sqrt{\frac{2n}{3}}}}{4n\sqrt{3}}\right)$$

Exemplo ($n = 4$)

$4 \quad 1+3 \quad 2+2 \quad 1+1+2 \quad 1+1+1+1$

Corte de varas

Implementação recursiva com *memoização*

MEMOIZED-CUT-ROD(p, n)

```
let v[0..n] be a new array
for l <- 0 to n do
    v[l] <- −∞
return MEMOIZED-CUT-ROD-2(p, n, v)
```

MEMOIZED-CUT-ROD-2(p, l, v)

```
if v[l] = −∞ then
  if l = 0 then
    q <- 0
  else
    q <- −∞
    for i <- 1 to l do
      q <- max(q, p[i] + MEMOIZED-CUT-ROD-2(p, l - i, v))
  v[l] <- q
return v[l]
```

Corte de varas

Cálculo iterativo de $v[n]$ (1)

p_i		1	5	7	11	11	17	20	20	24	27

Preenchimento do vector v

	0	1	2	3	4	5	6	7	8	9	10
$v[l]$	0	1	5	7	11	12	17	20	22	25	28

1. Caso base: $v[0] \leftarrow 0$
2. $v[1] \leftarrow \max\{\, p_1 + v[0] \,\} = \max\{\, 1 + 0 \,\}$
3. $v[2] \leftarrow \max\{\, p_1 + v[1], p_2 + v[0] \,\} = \max\{\, 1 + 1, 5 + 0 \,\}$
4. $v[3] \leftarrow \max\{\, p_1 + v[2], p_2 + v[1], p_3 + v[0] \,\} =$
   $= \max\{\, 1 + 5, 5 + 1, 7 + 0 \,\}$

   ...

11. $v[10] \leftarrow \max\{\, p_1 + v[9], p_2 + v[8], \ldots, p_4 + v[6], \ldots, p_{10} + v[0] \,\}$

Corte de varas

Cálculo iterativo de $v[n]$ (2)

BOTTOM-UP-CUT-ROD(p, n)

```
let v[0..n] be a new array
v[0] <- 0
for l <- 1 to n do
  q <- −∞
  for i <- 1 to l do
    q <- max(q, p[i] + v[l - i])
  v[l] <- q
return v[n]
```

Corte de varas

Complexidade

Complexidade de BOTTOM-UP-CUT-ROD($p_1\ p_2\ \ldots\ p_n$)

Ciclo 3–7 é executado n vezes

Ciclo 5–6 é executado l vezes, $l = 1, \ldots, n$

$$1 + 2 + \ldots + n = \sum_{l=1}^{n} l = \frac{n(n+1)}{2} = \Theta(n^2)$$

Todas as operações têm custo constante

Complexidade temporal $\Theta(n^2)$

Complexidade espacial $\Theta(n)$

Corte de varas

Construção da solução (1)

O valor máximo de venda de uma vara é calculado pela função BOTTOM-UP-CUT-ROD

Quais os cortes a fazer para obter esse valor?

Para o preenchimento da posição `l` do vector `v[]`, é escolhido o valor máximo de `p[i] + v[l - i]`

- A inclusão da parcela `p[i]` significa a inclusão de um pedaço de vara de comprimento `i`

Logo, o valor máximo de venda de uma vara de comprimento `l` (vector `c[]`) será obtido:

- Com um pedaço de comprimento `i` e
- Os pedaços que levam ao valor máximo de venda de uma vara de comprimento `l - i`

Corte de varas

Construção da solução (2)

p_i		1	5	7	11	11	17	20	20	24	27
	0	1	2	3	4	5	6	7	8	9	10
$v[l]$	0	1	5	7	11	12	17	20	22	25	28
$c[l]$		1	2	3	4	1	6	7	2	2	4

1. Caso base: $v[0] \leftarrow 0$
2. $v[1] \leftarrow \max\{\, p_1 + v[0] \,\} = \max\{\, 1 + 0 \,\}$, $c[1] \leftarrow 1$
3. $v[2] \leftarrow \max\{\, p_1 + v[1], p_2 + v[0] \,\} = \max\{\, 1 + 1, 5 + 0 \,\}$, $c[2] \leftarrow 2$
4. $v[3] \leftarrow \max\{\, p_1 + v[2], p_2 + v[1], p_3 + v[0] \,\} =$
   $= \max\{\, 1 + 5, 5 + 1, 7 + 0 \,\}$, $c[3] \leftarrow 3$

   ...

11. $v[10] \leftarrow \max\{\, p_1 + v[9], p_2 + v[8], \ldots, p_4 + v[6], \ldots, p_{10} + v[0] \,\}$,
    $c[10] \leftarrow 4$

Corte de varas

Construção da solução (3)

$c[1..n]$: $c[l]$ é o primeiro corte a fazer numa vara de comprimento l

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```
let v[0..n] and c[1..n] be new arrays
v[0] <- 0
for l <- 1 to n do
  q <- −∞
  for i <- 1 to l do
    if q < p[i] + v[l - i] then
      q <- p[i] + v[l - i]
      c[l] <- i                 // corte de tamanho i
  v[l] <- q
return v and c
```

Corte de varas

Resolução completa

PRINT-CUT-ROD-SOLUTION(p, n)

```
(v, c) <- EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
print "The best price is ", v[n]
print "Cuts:"
while n > 0 do
  print c[n]
  n <- n - c[n]
```

Resultado, para a vara de comprimento 10

```
The best price is 28
Cuts:
4
6
```