Técnicas de Projeto de Algoritmos II

Rômulo César Silva

Unioeste

Julho de 2021

unioeste
Universidade Estadual do Oeste do Paraná

Sumário

Projeto por Programação Dinâmica

A técnica de projeto de algoritmos por Programação Dinâmica é indicada principalmente para problemas de **otimização**, ou seja, aqueles em que a solução corresponde a encontrar um **máximo** ou **mínimo** de acordo com critérios estabelecidos no problema.

Observação: o nome *Programação Dinâmica* vem de razões históricas, não muito claras, ligadas à Matemática dos anos 1950. Portanto nada tem a ver como alocação dinâmica de memória em linguagens de programação. Alguns autores sugerem adotar o termo inglês *memoization* que é um termo próximo à palavra memorização, já que a técnica envolve o uso de armazenamento de soluções em tabelas.

Projeto por Programação Dinâmica

Na técnica de Programação Dinâmica:

- ao invés de se solucionar os problemas recursivamente, eles são sequencialmente solucionados e suas soluções são armazenadas em uma tabela
- o "pulo do gato" consiste em resolver os problemas em uma ordem *correta* tal que sempre que a solução de um subproblema é necessária, ela esteja disponível na tabela.
- o problema em questão deve atender ao **princípio da subestrutra ótima**: soluções ótimas de um problema devem incluir subsoluções ótimas de subproblemas
- em geral há **sobreposição de subproblemas**, tal que a resolução usando recursão ocasionaria recálculo de subsoluções

Projeto por Programação Dinâmica

Esquematicamente, um projeto de algortimo usando a técnica de Programação Dinâmica tem os seguintes passos:

1. planejamento da solução como uma sucessão de decisões e verificação que essa sucessão segue o princípio de otimalidade para o problema
2. definição recursiva da solução (geralmente expressa na forma de uma recorrência)
3. cálculo do valor da solução ótima mediante o uso de uma tabela em que se armazenam as soluções dos problemas parciais (subproblemas), permitindo reutilizar os resultados já obtidos
4. construção da solução ótima fazendo uso da informação armazenada na tabela anterior

Projeto por Programação Dinâmica - Fibonacci

Suponha que se deseja calcular o n-ésimo número da sequência de Fibonacci, expresso na recorrência:

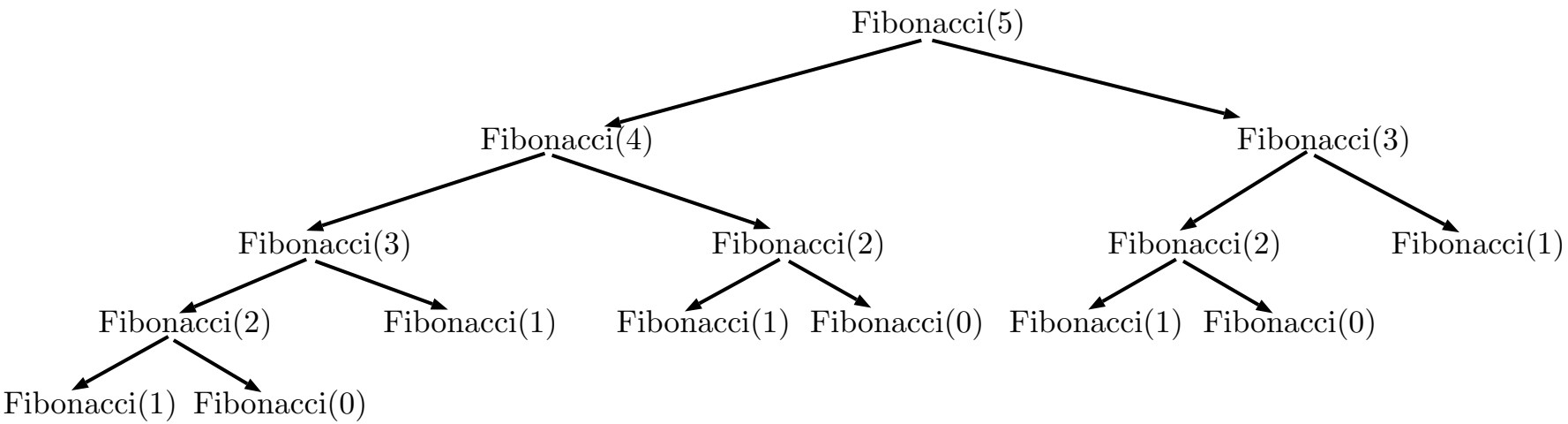
$$f(n) = \begin{cases} 1 & \text{se} \quad n = 0 \quad ou \quad n = 1 \\ f(n-1) + f(n-2) & \text{se} \quad n > 1 \end{cases}$$

que teria como algoritmo recursivo por indução simples:

```
1: function FIBONACCI(n)
2:     ▷ Entrada: inteiro n ≥ 0
3:     ▷ Saída: o n-ésimo número da sequência de Fibonacci
4:     if n ≤ 1 then
5:         return 1
6:     else
7:         return Fibonacci(n − 1) + Fibonacci(n − 2)
8:     end if
9: end function
```

Projeto por Programação Dinâmica - Fibonacci

Observe que a solução por indução simples faz com que sejam resolvidos várias vezes o mesmo subproblema, como por exemplo na figura abaixo que mostra as chamadas recursivas para `Fibonacci(5)`:



Projeto por Programação Dinâmica - Fibonacci

Na técnica de programação dinâmica, a ideia é construir uma tabela que armazene valores já calculados e ao mesmo que o cálculo seja feito de forma *bottom up*, ou seja, calcula-se na seguinte ordem:
`Fibonacci(0), Fibonacci(1), Fibonacci(2), Fibonacci(3), ...`

Projeto por Programação Dinâmica - Fibonacci

Isso conduz ao algoritmo abaixo:

```
 1: function FibIter(n)
 2:     ▷ Entrada: inteiro n ≥ 0
 3:     ▷ Saída: o n-ésimo número da sequência de Fibonacci
 4:     if n ≤ 1 then
 5:         return 1
 6:     else
 7:         Tab[0] ← 1
 8:         Tab[1] ← 1
 9:         for i = 2, ..., n do
10:             Tab[i] ← Tab[i − 1] + Tab[i − 2]
11:         end for
12:         return Tab[n]
13:     end if
14: end function
```

Projeto por Programação Dinâmica - Fibonacci

Para o algoritmo usando indução simples, a complexidade tempo é expressa pela recorrência

$$T(n) = \begin{cases} 1 & \text{se} \quad n = 0 \quad ou \quad n = 1 \\ T(n-1) + T(n-2) & \text{se} \quad n > 1 \end{cases}$$, que

resolvendo pelo método do polinômio característico, obtém-se que $T(n) = (\frac{\sqrt{5}+1}{2\sqrt{5}})[\frac{1+\sqrt{5}}{2}]^n + (\frac{\sqrt{5}-1}{2\sqrt{5}})[\frac{1-\sqrt{5}}{2}]^n$ e logo o algoritmo tem complexidade exponencial.

O algoritmo por programação dinâmica tem claramente complexidade de tempo $\Theta(n)$, portanto, substancialmente melhor que o algoritmo anterior.

Projeto por Programação Dinâmica - Fibonacci

A complexidade de espação do algoritmo por programação dinâmica tambem é $\Theta(n)$, já que é usado um vetor para armazenar as soluções parciais. Isto pode ser reduzido para $\Theta(1)$, trocando o uso do vetor por 3 variáveis auxiliares escalares, conforme abaixo:

```
1:  function FibIter2(n)
2:      ▷ Entrada: inteiro n ≥ 0
3:      ▷ Saída: o n-ésimo número da sequência de Fibonacci
4:      if n ≤ 1 then
5:          return 1
6:      else
7:          aux1 ← 1
8:          aux2 ← 1
9:          for i = 2, ..., n do
10:             soma ← aux1 + aux2
11:             aux2 ← aux1
12:             aux1 ← soma
13:         end for
14:         return soma
15:     end if
16: end function
```

Projeto por Programação Dinâmica - Multiplicação de Cadeia de Matrizes

Problema da Multiplicação de Cadeia de Matrizes

Seja M uma sequência de matrizes a serem multiplicadas:

$$M = A_1 \times A_2 \times ...A_i \times ...A_n$$

- matrizes são sempre multiplicadas aos pares
- a multiplicação de matrizes $A \times B$ somente é possível se o número de colunas de A é igual ao número de linhas de B, isto é, $A_{p \times q}$ e $B_{q \times r}$, resultando em $C_{p \times r}$

Projeto por Programação Dinâmica - Multiplicação de Cadeia de Matrizes

Pela propriedade associativa da multiplicação de matrizes, existem várias ordens possíveis de se efetuar a multiplicação. Dependendo do número de linhas e colunas de cada matriz, e da ordem escolhida para multiplicação, o número de multiplicações e somas escalares pode ser consideravelmente diferente. Considere o exemplo:

$$M = A_1 \times A_2 \times A_3$$

sendo que as dimensões das matrizes são 10×100, 100×5 e 5×50. Existem a seguintes parentizações possíveis com a respectiva quantidade de multiplicações escalares efetuadas:

- $(A_1 \times A_2) \times A_3 \quad \Rightarrow$ 7500 multiplicações
- $A_1 \times (A_2 \times A_3) \quad \Rightarrow$ 75000 multiplicações

Logo multiplicação de matrizes de acordo com a ordem da primeira parentização é 10 vezes mais rápida que a segunda.

Projeto por Programação Dinâmica - Multiplicação de Cadeia de Matrizes

Problema da Multiplicação de Cadeia de Matrizes

Seja M uma sequência de n matrizes a serem multiplicadas:

$$M = A_1 \times A_2 \times ...A_i \times ...A_n$$

em que a matriz A_i tem dimensões $p_{i-1} \times p_i$.

Encontrar uma parentização que minimize o número de multiplicações escalares efetuadas.

Quantas parentizações possíveis existem??

Projeto por Programação Dinâmica - Multiplicação de Cadeia de Matrizes

Seja $P(n)$ o número de parentizações possíveis para multiplicação de uma cadeia de n matrizes.
Desde que a sequência pode ser separada entre a k e a $k+1$ matrizes para $k = 1, 2..., n-1$ e cada parte pode ser parentizada independentemente, obtém-se a recorrência:

$$P(n) = \begin{cases} 1 & \text{se} \quad n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{se} \quad n \geq 2 \end{cases}$$

Projeto por Programação Dinâmica - Multiplicação de Cadeia de Matrizes

$P(n)$ corresponde a sequência dos números de **Catalan** (C), onde $P(n) = C(n-1)$ e

$$C(n) = \frac{1}{n+1}\binom{2n}{n}$$

$$= \Omega(\frac{4^n}{n^{\frac{3}{2}}})$$

E portanto o numero de soluções é exponencial, sendo inviável usar a estratégia de *força bruta* para encontrar a parentização ótima.

Projeto por Programação Dinâmica - Multiplicação de Cadeia de Matrizes

Caracterização do problema para emprego da técnica de programação dinâmica:

- Dada um parentização ótima, exitem 2 pares de parênteses que correspondem ao último par de matrizes (B e C) a ser multiplicado. Isto é, existe k tal que
$M = \underbrace{(A_1 A_2 ... A_k)}_{B} \underbrace{(A_{k+1} A_{k+2} ... A_n)}_{C}$
- Como a parentização de M é ótima, as parentizações para o cálculo de B e C também devem ser ótimas $\Rightarrow$ **princípio da subestrutura ótima** do problema.

Projeto por Programação Dinâmica - Multiplicação de Cadeia de Matrizes

Seja $m[i,j]$ o número mínimo de multiplicações escalares necessário para calcular a multiplicação da cadeia de matrizes $A_i A_{i+1} ... A_j$.
Assim a solução do problema geral é encontrar $m[1,n]$.
$m[i,j]$ pode ser definido da seguinte forma:

- se $i = j$ há apenas uma matriz e o produto da cadeia é a própria matriz A_i. Logo $m[i,i] = 0$ para $i = 1, 2, ..., n$.
- se $i < j$, então $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} \times p_k \times p_j$ onde $i \leq k < j$. Para a solução ser ótima deve-se descobrir k tal que $m[i,j]$ seja mínimo.

Assim $m[i,j]$ pode ser expresso pela recorrência:

$$m[i,j] = \begin{cases} 0 & \text{se} \quad i = j \\ \min\limits_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{se} \quad i < j \end{cases}$$

Projeto por Programação Dinâmica - Multiplicação de Cadeia de Matrizes

O algoritmo recursivo, obtido por indução simples, que permite calcular $m[i,j]$ é dado por:

```
1:  function MinMultMatRec(p[0..n − 1], i, j)
2:      ▷ Entrada: vetor p com as dimensões das matrizes, índices i e j delimitando
    uma subcadeia de matrizes
3:      ▷ Saída: m[i, j] contendo o mínimo de multiplicações para A_i...A_j
4:      if i = j then
5:          return 0
6:      else
7:          m[i, j] ← ∞
8:          aux ← 1
9:          for k = i, ..., j − 1 do
10:             aux ← MinMultMatRec(p, i, k) + MinMultMatRec(p, k + 1, j) +
    p[i − 1] ∗ p[k] ∗ p[j]
11:             if m[i, j] > aux then
12:                 m[i, j] ← aux
13:             end if
14:         end for
15:         return m[i, j]
16:     end if
17: end function
```

Projeto por Programação Dinâmica - Multiplicação de Cadeia de Matrizes

A complexidade de tempo do algoritmo recursivo é dada pela recorrência:

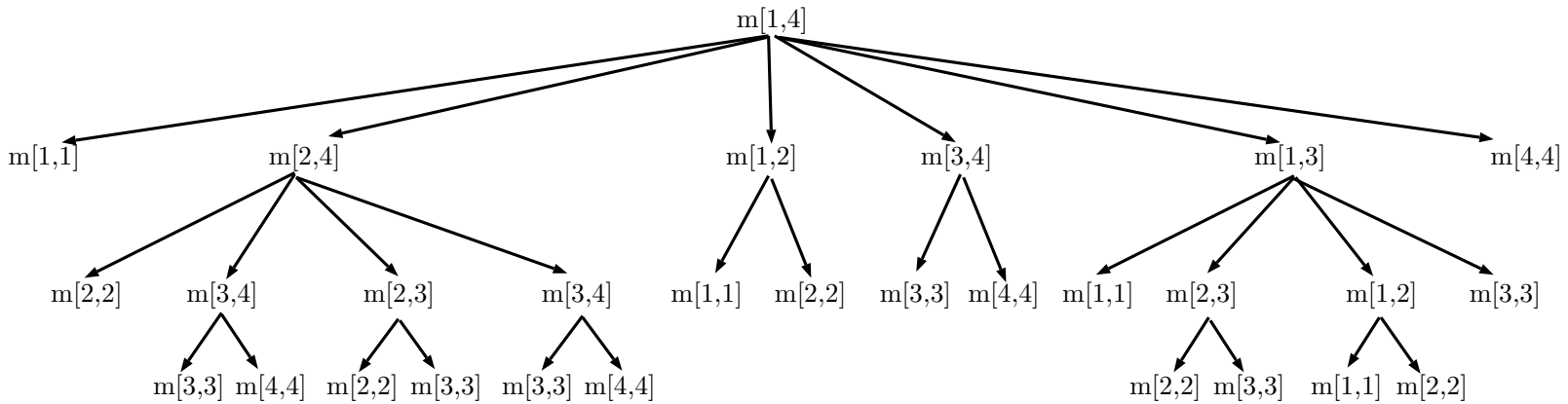
$$T(n) = \begin{cases} 1 & \text{se} \quad n = 1 \\ 1 + \sum_{k=1}^{n} [T(k) + T(n-k) + 1] & \text{se} \quad n > 1 \end{cases}$$

Usando o método da substituição é possível mostrar que $T(n) \geq 2^{n-1}$, ou seja $T(n) \in \Omega(2^n)$, e portanto a complexidade de tempo do algoritmo é exponencial.

A ineficiência do algoritmo ocorre em função da sobreposição de subproblemas, sendo calculado várias vezes o mesmo $m[i, j]$

Projeto por Programação Dinâmica - Multiplicação de Cadeia de Matrizes

A ineficiência do algoritmo ocorre em função da sobreposição de subproblemas, sendo calculado várias vezes o mesmo $m[i, j]$, como pode ser vista na figura abaixo, que mostra as chamadas recursivas para o cálculo de $m[1, 4]$, isto é, para o produto $A_1 A_2 A_3 A_4$:



Projeto por Programação Dinâmica - Multiplicação de Cadeia de Matrizes

Aplicando a técnica de programação dinâmica, os valores da tabela $m[i, j]$ são calculados de tal maneira que não se recalcula subsoluções. Isso implica que o preenchimento da tabela não é feito exatamente da forma convencional que se usa para varrer matrizes.

O algoritmo de programação dinâmica para encontrar o mínimo de multiplicações escalares é mostrado no slide seguinte.

Projeto por Programação Dinâmica - Multiplicação de Cadeia de Matrizes

```
1:  function MinMultMatProgDinamica(p[0..n − 1])
2:      ▷ Entrada: vetor p com as dimensões das matrizes
3:      ▷ Saída: tabela m preenchida
4:      for i ← 1, ..., n do
5:          m[i, i] ← 0
6:      end for
7:      for u ← 1, ..., n − 1 do
8:          for i ← 1, ..., n − u do
9:              j ← i + u
10:             m[i, j] ← ∞
11:             for k ← i, ..., j − 1 do
12:                 q ← m[i, k] + m[k + 1, j] + p_{i−1} * p_k * p_j
13:                 if q < m[i, j] then
14:                     m[i, j] ← q
15:                 else
16:                     m[i, j] ← k
17:                 end if
18:             end for
19:         end for
20:     end for
21:     return m
22: end function
```

Projeto por Programação Dinâmica - Problema Booleano da Mochila

Problema da Mochila

Dada uma mochila de capacidade W (inteiro) e um conjunto de n itens com tamanho w_i (inteiro) e valor v_i associado a cada item i, determinar quais itens devem ser colocados na mochila de modo a **maximizar** o valor total transportado, respeitando sua capacidade.

Projeto por Programação Dinâmica - Problema Booleano da Mochila

Pode-se representar os dados do problema da seguinte maneira:

- uso de um vetor x booleano tal que $x[i] = 1$ se o item i está na solução ótima ou $x[i] = 0$ caso contrário.
- uso de um vetor w tal que $w[i]$ representa o peso do item i
- uso de um vetor v tal que $v[i]$ representa o valor do item i
- pode-se supor que $0 < w[i] \leq W \ \forall i = 1, ..., n$
- o peso de uma mochila corresponde a:

  $x[1] * w[1] + ... + x[n]w[n]$, isto é, $\sum_{i=1}^{n} x[i] * w[i]$
- uma mochila ótima é dada por $max \sum_{i=1}^{n} x[i] * v[i]$ tal que

  $\sum_{i=1}^{n} x[i] * w[i] \leq W$

Projeto por Programação Dinâmica - Problema Booleano da Mochila

Observações:

- este problema é conhecido como a versão binária ou **booleana** do problema da Mochila porque nesta modelagem só se admite que um item esteja inteiramente na mochila ou esteja inteiramente fora da mochila, não admitindo que o item seja dividido em partes
- na versão **fracionária**, é admitido que parte de um item possa entrar na mochila
- na versão **booleana**, o número total de possiblidades a serem testadas é 2^n, já que corresponde à somatória de números binomais representando as combinações $C_{n,0} + C_{n,1} + C_{n,2} + ... + C_{n,n}$, que mostra que o método da **força bruta** é inviável

Projeto por Programação Dinâmica - Problema Booleano da Mochila

Caracterização do problema para uso de programação dinâmica:

- é um problema de otimização, já que deseja-se **maximizar** o valor da mochila
- atende ao princípio da subestrutura ótima:
  - se o item n estiver na solução ótima, o valor desta solução é $v[n]$ mais o valor da solução ótima considerando capacidade $W - w[n]$ e apenas os $n - 1$ primeiros itens
  - se o item n não estiver na solução ótima, o valor desta solução é o valor da solução ótima considerando capacidade W e os $n - 1$ primeiros itens

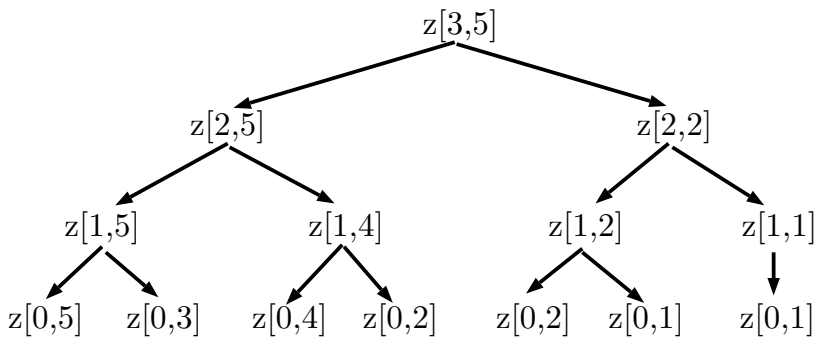
Projeto por Programação Dinâmica - Problema Booleano da Mochila

Pode-se escrever a seguinte recorrência $z[k, d]$ para calcular o valor ótimo da mochila para capacidade d, considerando subconjunto dos k primeiros itens da instância original:

$$z[k,d] = \begin{cases} 0 & \text{se } k = 0 \text{ ou } d = 0 \\ z[k-1, d] & \text{se } w_k > d \\ max\{z[k-1, d],\ z[k-1, d-w_k]\} & \text{se } w_k \leq d \end{cases}$$

Projeto por Programação Dinâmica - Problema Booleano da Mochila

Se usássemos um algoritmo recursivo para calcular os valores da recorrência anterior, ele teria complexidade exponencial, e além disso, calcularia o mesmo valor várias vezes (sobreposição de problemas!), conforme mostra o exemplo abaixo da árvore de recursão considerando vetor $w = \{2, 1, 3\}$ e capacidade $W = 5$.



O algoritmo de programação dinâmica, que evita recálculos, é apresentado no próximo *slide*.

Projeto por Programação Dinâmica - Problema Booleano da Mochila

```
1:  function MochilaBooleana(w[1..n], v[1..n], W)
2:      ▷ Entrada: vetores w de pesos, v de valores e W capacidade total da mochila
3:      ▷ Saída: vetor booleano x
4:      for d ← 0, ..., W do                ▷ inicializa valor da mochila para quantidade de itens igual a zero
5:          m[0, d] ← 0
6:      end for
7:      for k ← 1, ..., n do                ▷ inicializa valor da mochila para capacidade igual a zero
8:          m[k, 0] ← 0
9:      end for
10:     for k ← 1, ..., n do
11:         for d ← 1, ..., W do
12:             z[k, d] ← z[k − 1, d]
13:             if w[k] ≤ d and v[k] + z[k − 1, d − w[k]] > z[k, d] then
14:                 z[k, d] ← v[k] + z[k − 1, d − w[k]]      ▷ inclui o item k na mochila
15:             end if
16:         end for
17:     end for
18:     for k ← n downto 1 do                   ▷ cálculo do vetor x
19:         if z[k, W] = z[k − 1, W] then
20:             x[i] ← 0                        ▷ item k não está na mochila
21:         else
22:             x[i] ← 1                        ▷ item k foi incluído na mochila
23:             W ← W − w[k]
24:         end if
25:     end for
26:     return x
27: end function
```

Projeto por Programação Dinâmica - Problema Booleano da Mochila - exemplo

Considere $w = \{6, 3, 4, 2\}$, $v = \{30, 14, 16, 9\}$ e $W = 10$. Primeiro são preenchidas a $1^{\underline{a}}$ linha e $1^{\underline{a}}$ coluna da tabela $z[k, d]$:

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0										
2	0										
3	0										
4	0										

Projeto por Programação Dinâmica - Problema Booleano da Mochila - exemplo

$w = \{6, 3, 4, 2\}$, $v = \{30, 14, 16, 9\}$ e $W = 10$.
Em seguida, o restante da tabela $z[k, d]$ é preenchida linha a linha:

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0										
3	0										
4	0										

Projeto por Programação Dinâmica - Problema Booleano da Mochila - exemplo

$w = \{6, 3, 4, 2\}$, $v = \{30, 14, 16, 9\}$ e $W = 10$.

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0										
4	0										

Projeto por Programação Dinâmica - Problema Booleano da Mochila - exemplo

$w = \{6, 3, 4, 2\}$, $v = \{30, 14, 16, 9\}$ e $W = 10$.

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0										

Projeto por Programação Dinâmica - Problema Booleano da Mochila - exemplo

$w = \{6, 3, 4, 2\}$, $v = \{30, 14, 16, 9\}$ e $W = 10$.

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	0	14	16	23	30	30	30	44	46

A solução procurada (valor máximo a ser transportado) está em z[4,10].

Projeto por Programação Dinâmica - Problema Booleano da Mochila - exemplo

$w = \{6, 3, 4, 2\}$, $v = \{30, 14, 16, 9\}$ e $W = 10$.

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	0	14	16	23	30	30	30	44	46

Para determinar os itens que maximizam o valor total, compara-se $z[k, d]$ e $z[k-1, d]$ começando por $k = n$. Quando iguais significa que o item k não está na solução ($x_k = 0$), então a solução deve estar em $z[k-1, d]$. Quando diferentes, então k está na solução ($x_k = 1$), e os demais itens correspondem ao valor $z[k-1, d-w_k]$. No exemplo teremos $x_1 = 1, x_2 = 0, x_3 = 1$ e $x_4 = 0$.

Projeto por Programação Dinâmica - Problema Booleano da Mochila

Observações:

- A complexidade do algoritmo é $\Theta(nW)$, o que não é muito eficiente, pois quando W é grande em relação ao número de itens, o algoritmo terá muitas iterações. Por isso, como sua complexidade depende do valor de W, o algoritmo é dito pseudopolinomial.
- este algoritmo funciona apenas para valores inteiros de W

Projeto de Algoritmos Gulosos

Algoritmos Gulosos são indicados para problemas de otimização. Algoritmos para problemas de otimização tipicamente fazem um sequência de passos, com um conjunto de escolhas a cada passo.

Algoritmos Gulosos

- sempre faz a escolha que parece ser a "melhor" no momento, isto é, faz uma escolha ótima localmente na esperança de que esta escolha conduza ao ótimo global
- cada escolha feita nunca é revista, isto é, não existe *backtracking*

Projeto de Algoritmos Gulosos

Os problemas devem apresentar:

- a **propriedade da subestrutura ótima**, assim como na programação dinâmica
- a propriedade da **escolha gulosa**: deve haver alguma propriedade ou argumento válido que garanta que a escolha feita conduz de fato a um ótimo global

O algoritmo guloso tipicamente faz uma escolha de um elemento que irá compor a solução ótima e em seguida um subproblema é resolvido, enquanto na programação dinâmica os subproblemas são resolvidos de maneira ótima antes e depois procede-se à escolha de um escolha para compor a solução ótima.

Projeto de Algoritmos Gulosos - Seleção de Atividades

Problema da Seleção de Atividades

Seja $S = \{a_1, ..., a_n\}$ um conjunto de n atividades a serem realizadas no mesmo local, tal que cada atividade i é realizada no intervalo de tempo $[s_i, f_i)$.

Duas atividades i e j são compatíveis se não há sobreposição entre os intervalos $[s_i, f_i)$ e $[s_j, f_j)$.

O problema da seleção de atividades consiste em encontrar um subconjunto de tamanho **máximo** de atividades mutuamente compatíveis.

Projeto de Algoritmos Gulosos - Seleção de Atividades

Algumas suposições:

- na entrada do problemas, as atividades estão ordenadas pelo tempo de término, isto é, $f_1 \leq f_2 \leq ... \leq f_n$
- s e f são representados como vetores

Projeto de Algoritmos Gulosos - Seleção de Atividades

```
1:  function SelecaoAtividadesGulosa(s[1..n], f[1..n])
2:      ▷ Entrada: vetores s e f, representando os tempos de início e término das
    atividades, estando ordenados por tempo de término
3:      ▷ Saída: conjunto A de atividades
4:      A ← {1}
5:      j ← 1
6:      for i ← 2, ..., n do
7:          if s[i] ≥ f[j] then
8:              A ← A ∪ {i}                          ▷ inclui a atividade i
9:              j ← i
10:         end if
11:     end for
12:     return A
13: end function
```

Complexidade: $\Theta(n)$

Projeto de Algoritmos Gulosos - Seleção de Atividades

Analisando a corretude do algoritmo:

- considerando que as atividades estão ordenadas pelo tempo de término, logo existe uma solução ótima que contém a atividade 1, cujo tempo de término é o menor de todos. Esta é a escolha gulosa do algoritmo
- uma vez que a atividade 1 está em uma solução ótima, então o problema se reduz a encontrar atividades compatíveis com a atividade 1. Isto é, se A é solução ótima para S, então $A' = A - \{1\}$ é solução ótima para $S' = \{i \in S : s[i] \geq f[1]\}$.
- a escolha gulosa garante o seguinte invariante: j é sempre a última atividade inserida no conjunto A. Assim $f[j] = max\{f[k] : k \in A\}$

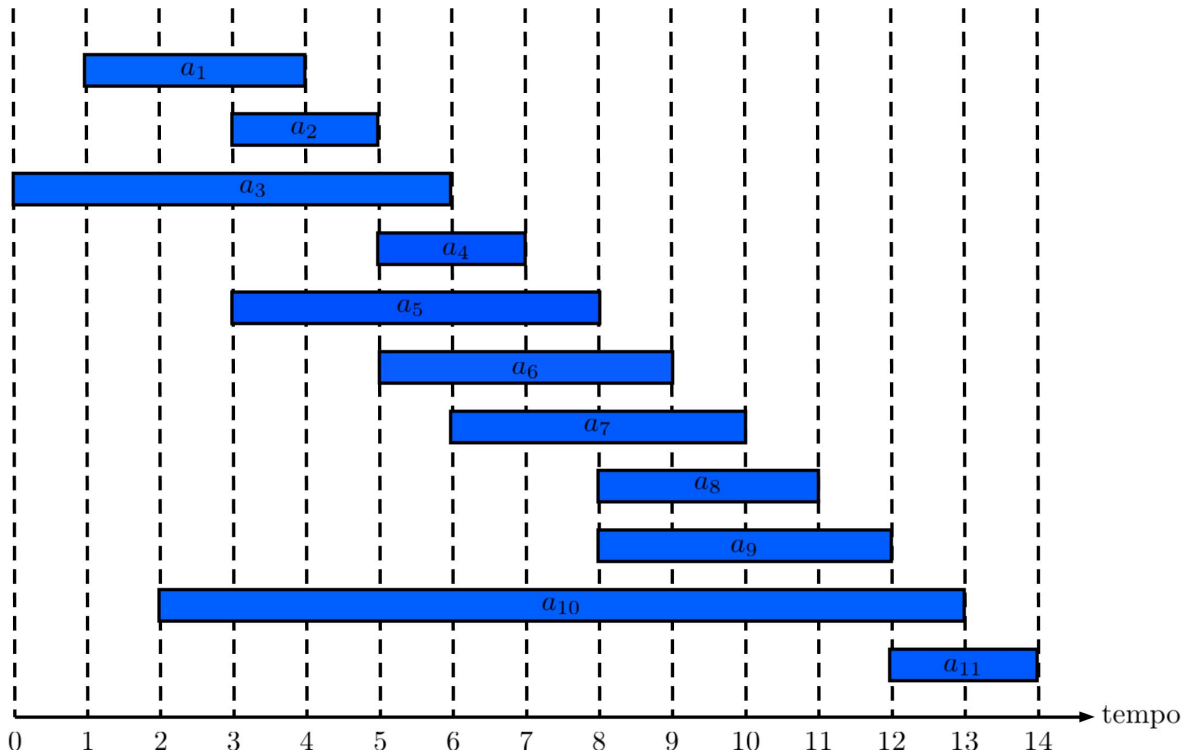
Projeto de Algoritmos Gulosos - Seleção de Atividades - exemplo

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- ex. de compatíveis: $(a_1, a_4), (a_4, a_8)$
- ex. de incompatíveis:$(a_1, a_2), (a_1, a_3)$
- conjuntos máximos de compatíveis: (a_1, a_4, a_8, a_{11}) e (a_2, a_4, a_9, a_{11})

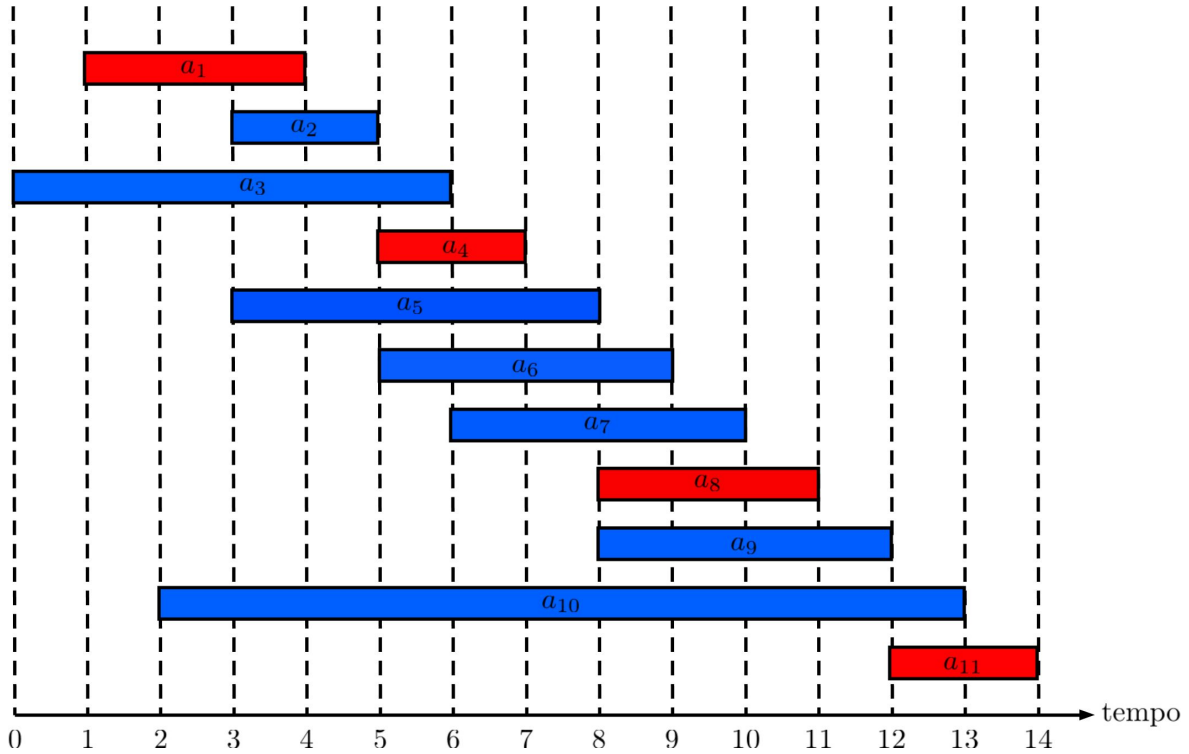
Projeto de Algoritmos Gulosos - Seleção de Atividades - exemplo

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



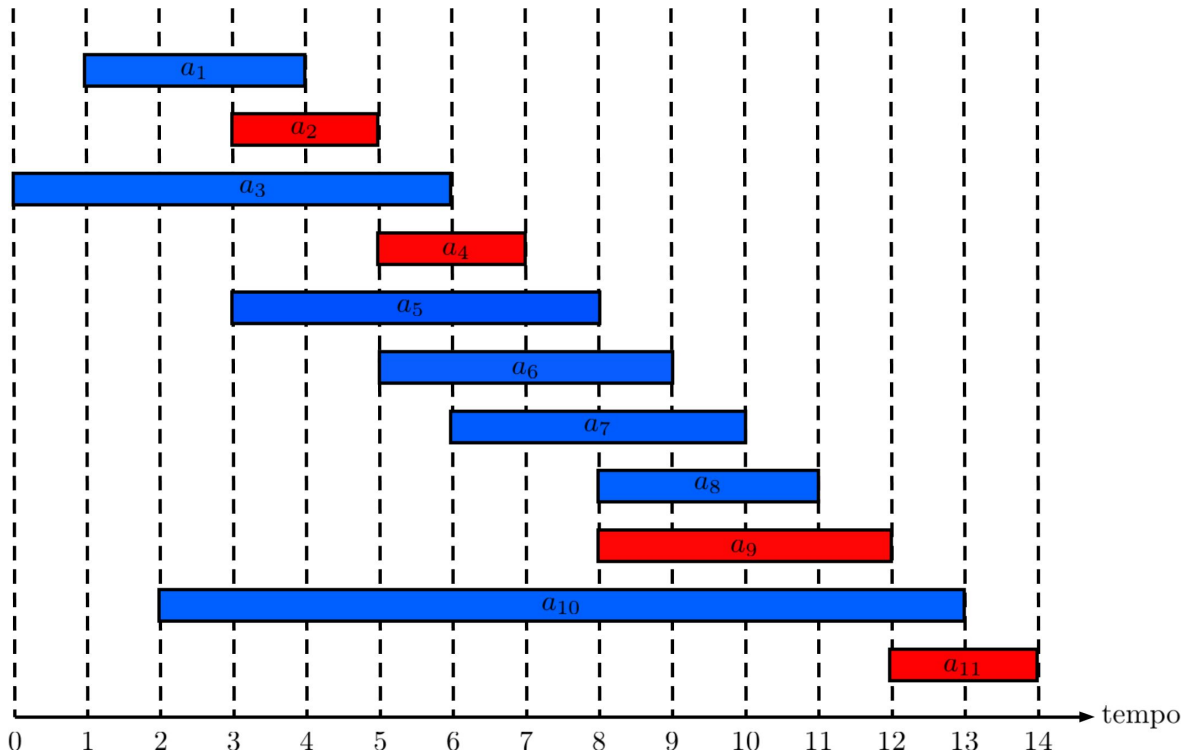
Projeto de Algoritmos Gulosos - Seleção de Atividades - exemplo

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



Projeto de Algoritmos Gulosos - Seleção de Atividades - exemplo

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



Projeto de Algoritmos Gulosos - Algoritmo de Huffman

Considere o problema mais geral de armazenamento e transmissão de arquivos texto via rede de computadores. Como codificar o conteúdo do arquivo?

- **códigos de tamanho fixo**: representar cada caracter do arquivo por uma sequência de *bits*, todas de mesmo tamanho
- **códigos de tamanho variável**: representar cada caracter do arquivo por uma sequência de *bits* de tamanho variável, tal que caracteres mais frequentes no arquivo tenham tamanho menor

Problema da Codificação

Dadas as frequências de ocorrência dos caracteres de um arquivo, determinar as sequências de *bits* (códigos) usadas para representá-los tal que o tamanho do arquivo seja **mínimo**.

Projeto de Algoritmos Gulosos - Algoritmo de Huffman

Exemplo ilustrativo: considere um arquivo texto que contenha caracteres do alfabeto $= \{a, b, c, d, e, f\}$. A tabela abaixo mostra as frequências (em porcentagem) dos caracteres no arquivo, e respectivos códigos, de tamanho fixo igual a 3 e tamanho variável, respectivamente.

	a	b	c	d	e	f
frequência	45	13	12	16	9	5
código (tamanho fixo)	000	001	010	011	100	101
código (tamanho variável)	0	101	100	111	1101	1100

Projeto de Algoritmos Gulosos - Algoritmo de Huffman

Considere um arquivo de 100.000 caracteres.

- usando códigos de tamanho fixo, o tamanho final do arquivo é $3 \times 100.000 = 300.000$ *bits*
- usando códigos de tamanho variável, o tamanho final do arquivo é: $\frac{((45\times1)+(13\times3)+(12\times3)+(16\times3)+(9\times4)+(5\times4))\times100.000}{100}$ $= 224.000$ *bits*
  - ganho de $\approx 25\%$

Como obter uma codificação que gere um arquivo de tamanho **mínimo**?

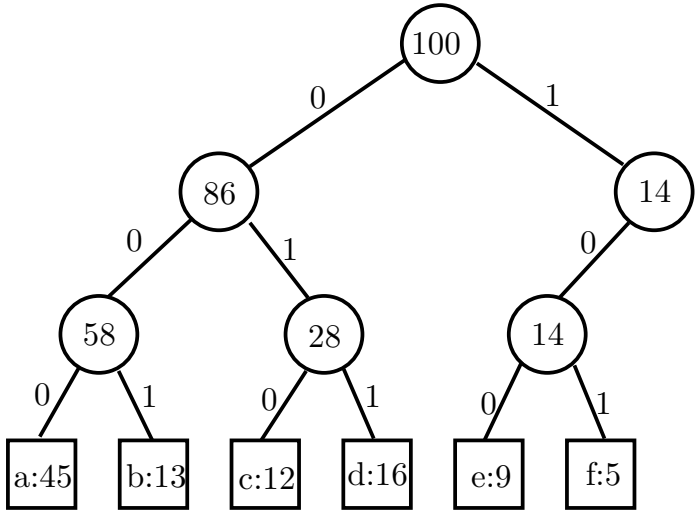
Projeto de Algoritmos Gulosos - Algoritmo de Huffman

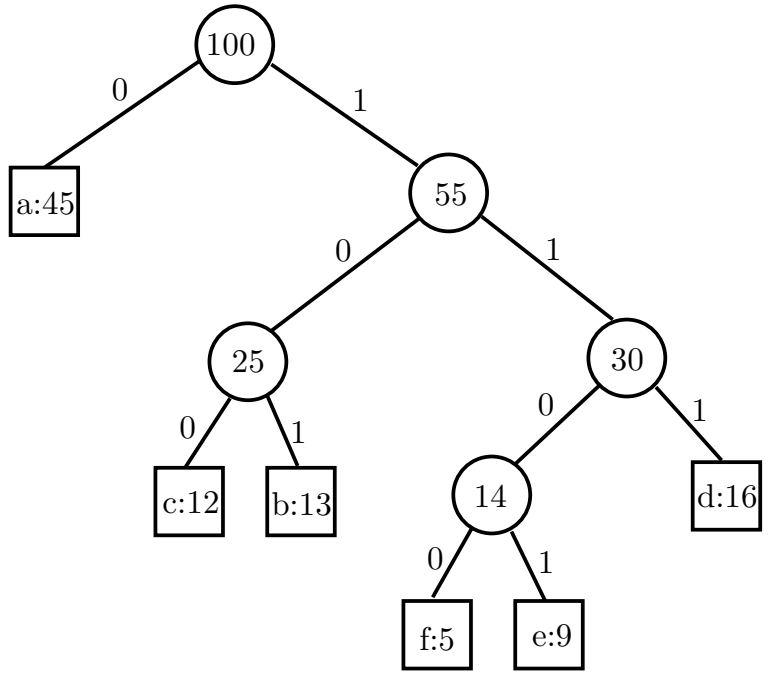
Observações:

- o uso de codificação de tamanho variável exige que os códigos obtidos sejam livres de prefixo, isto é, nenhum código seja prefixo de outro código, para que seja possível a decodificação correta do arquivo. Ou seja, os códigos devem ser tais que não gere ambiguidade na leitura do arquivo

Projeto de Algoritmos Gulosos - Algoritmo de Huffman

- dada uma codificação (tamanho fixo ou variável), ela pode ser representada usando uma árvore binária conforme mostrado nas figuras abaixo.





Projeto de Algoritmos Gulosos - Algoritmo de Huffman

Dada uma árvore de codificação T, o número de *bits* necessários para codificar um arquivo que faz uso do alfabeto C pode calculado por:

$$\sum_{c \in C} f(c) d_T(c)$$

onde:

- $f(c)$: frequência (número de ocorrências) do caracter c no arquivo
- $d_T(c)$: comprimento do código do caracter c na árvore de codificação T

Projeto de Algoritmos Gulosos - Algoritmo de Huffman

Algoritmo de Huffman

O algoritmo de Huffman constrói de baixo para cima uma árvore binária de codificação livre de prefixos, baseado nos seguintes argumentos:

- os caracteres de menor frequência terão códigos maiores enquanto os que mais ocorrem no arquivo devem ter códigos menores visando diminuir o tamanho total do arquivo
- uma árvore de codificação ótima será uma árvore binária cheia (todo nó tem exatamente 0 ou 2 filhos)
- critério de **escolha gulosa**: como a construção se dá a partir das folhas para a raiz, deve-se escolher primeiramente os caracteres de menor frequência, que estarão no nível mais baixo da árvore e portanto terão os maiores códigos.

Projeto de Algoritmos Gulosos - Algoritmo de Huffman

- O algoritmo faz uso de uma fila de prioridade baseada na frequência dos caracteres, e à medida que se constrói a árvore, cada nó gerado pelo agrupamento de nós do nível abaixo tem sua frequência calculada pela soma dos nós agrupados e é inserido na fila. Assim se garante que sempre está se selecionando os nós de menor frequência.
- sendo n o número de caracteres, é necessário fazer o agrupamento de nós $n - 1$ vezes para obtenção da árvore

Projeto de Algoritmos Gulosos - Algoritmo de Huffman

```
1:  function HUFFMAN(C)
2:      ▷ Entrada: conjunto de caracteres C e respectivas frequências f
3:      ▷ Saída: raiz da árvore binária de uma codificação ótima
4:      n ← |C|                                  ▷ número de caracteres do alfabeto
5:      Q ← C                          ▷ Q é fila de prioridade baseada nas frequências
6:      for i ← 1, ..., n − 1 do
7:          z ← newNode()                ▷ z é nó da árvore binária sendo construída
8:          ▷ Retira da fila os 2 nós de menor frequência
9:          x ← ExtraiMinimo(Q)
10:         y ← ExtraiMinimo(Q)
11:         z.esq ← x
12:         z.dir ← y
13:         z.f ← x.f + y.f                  ▷ Calcula a frequência do nó z
14:         insere(Q, z)                     ▷ insere o nó z na fila
15:     end for
16:     return ExtraiMinimo(Q)               ▷ retorna a raiz da árvore
17: end function
```

unioeste
Universidade Estadual do Oeste do Paraná

Projeto de Algoritmos Gulosos - Algoritmo de Huffman

Complexidade:

- linha 5: $O(n)$ se a fila for construída usando uma estrutura de *heap*
- linhas 6-15 (*loop* **for**): é executado $n - 1$ vezes, e como cada operação no *heap* requer $O(lg\ n)$, então o custo do *loop* é $O(n\ lg\ n)$
- complexidade total: $O(n\ lg\ n)$

Resumo

- Existem várias técnicas de projeto de algoritmos
- O que determina a aplicabilidade de uma técnica específica são as características intrínsecas do problema
- A aplicabilidade de uma técnica não garante que o algoritmo obtido seja eficiente
- A técnica de **projeto por indução** é a técnica mais básica. Dependendo da forma como a indução é feita, a técnica recebe um nome específico como a técnica de **divisão e conquista**
- O estudo de diversos problemas e seus algoritmos permitem ao projetista de algoritmo adquirir o "traquejo" no uso das técnicas

Bibliografia I

[Cormen 1997] Cormen,T.; Leiserson, C.; Rivest, R.
Introduction to Algorithms. McGrawHill, New York, 1997.

[Dasgupta 2009] Sanjoy Dasgupta; Christos Papadimitriou; Umesh Vazirani.
Algoritmos. McGrawHill, São Paulo, 2009.

[Feofiloff 2009] Paulo Feofiloff.
Algoritmos em linguagem C. Elsivier, Rio de Janeiro, 2009.

[Manber 1989] Udi Manber.
Introduction to Algorithms: A Creative Approach.
Addison-Wesley, 1989.

[Ziviani 2011] Nivio Ziviani.
Projeto de Algoritmos: com implementação em Java e C++.
Cengage Learning, São Paulo, 2011.