Projeto e Análise de Algoritmos

Técnicas de projeto de algoritmos avançadas

Cid Carvalho de Souza, Cândida Nunes da Silva et al.

Primeiro Semestre de 2017

Programação Dinâmica

Programação Dinâmica: Conceitos Básicos

- Tipicamente o paradigma de programação dinâmica aplica-se a problemas de **otimização**.
- Podemos utilizar programação dinâmica em problemas onde há:
  - **Subestrutura Ótima:** As soluções ótimas do problema incluem soluções ótimas de subproblemas.
  - **Sobreposição de Subproblemas:** O cálculo da solução através de recursão implica no recálculo de subproblemas.

Programação Dinâmica: Conceitos Básicos (Cont.)

- A técnica de programação dinâmica visa evitar o recálculo desnecessário das soluções dos subproblemas.
- Para isso, soluções de subproblemas são armazenadas em tabelas.
- Logo, para que o algoritmo de programação dinâmica seja eficiente, é preciso que o número total de subproblemas que devem ser resolvidos seja pequeno (polinomial no tamanho da entrada).

Multiplicação de Cadeias de Matrizes

Problema: Multiplicação de Matrizes

Calcular o número mínimo de operações de multiplicação (escalar) necessários para computar a matriz M dada por:

$$M = M_1 \times M_2 \times \ldots M_i \ldots \times M_n$$

onde M_i é uma matriz de b_{i-1} linhas e b_i colunas, para todo $i \in \{1, \ldots, n\}$.

- Matrizes são multiplicadas aos pares sempre. Então, é preciso encontrar uma parentização (agrupamento) ótimo para a cadeia de matrizes.
- Para calcular a matriz M' dada por $M_i \times M_{i+1}$ são necessárias $b_{i-1} * b_i * b_{i+1}$ multiplicações entre os elementos de M_i e M_{i+1}.

Multiplicação de Cadeias de Matrizes (Cont.)

- **Exemplo:** Qual é o mínimo de multiplicações escalares necessárias para computar $M = M_1 \times M_2 \times M_3 \times M_4$ com $b = \{200, 2, 30, 20, 5\}$?
- As possibilidades de parentização são:

$$\begin{array}{lcl} M = (M_1 \times (M_2 \times (M_3 \times M_4))) & \rightarrow & 5.300 \text{ multiplicações} \\ M = (M_1 \times ((M_2 \times M_3) \times M_4)) & \rightarrow & 3.400 \text{ multiplicações} \\ M = ((M_1 \times M_2) \times (M_3 \times M_4)) & \rightarrow & 4.500 \text{ multiplicações} \\ M = ((M_1 \times (M_2 \times M_3)) \times M_4) & \rightarrow & 29.200 \text{ multiplicações} \\ M = (((M_1 \times M_2) \times M_3) \times M_4) & \rightarrow & 152.000 \text{ multiplicações} \end{array}$$

- A ordem das multiplicações faz **muita** diferença !

Multiplicação de Cadeias de Matrizes (Cont.)

- Poderíamos calcular o número de multiplicações para todas as possíveis parentizações.
- O número de possíveis parentizações é dado pela recorrência:

$$P(n) = \begin{cases} 1, & n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n > 1, \end{cases}$$

- Mas $P(n) \in \Omega(4^n / n^{\frac{3}{2}})$, a estratégia de força bruta é impraticável !

Multiplicação de Cadeias de Matrizes (Cont.)

- Inicialmente, para todo (i, j) tal que $1 \leq i \leq j \leq n$, vamos definir as seguintes matrizes:

$$M_{i,j} = M_i \times M_{i+1} \times \ldots \times M_j.$$

- Agora, dada uma parentização ótima, existem dois pares de parênteses que identificam o último par de matrizes que serão multiplicadas.
  Ou seja, existe k tal que $M = M_{1,k} \times M_{k+1,n}$.
- Como a parentização de M é ótima, as parentizações no cálculo de $M_{i,k}$ e $M_{k+1,n}$ devem ser ótimas também, caso contrário, seria possível obter uma parentização de M ainda melhor !
- Eis a subestrututra ótima do problema: a parentização ótima de M inclui a parentização ótima de $M_{i,k}$ e $M_{k+1,n}$.

Multiplicação de Cadeias de Matrizes (Cont.)

- De forma geral, se $m[i,j]$ é número mínimo de multiplicações que deve ser efetuado para computar $M_i \times M_{i+1} \times \ldots \times M_j$, então $m[i,j]$ é dado por:

$$m[i,j] := \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + b_{i-1} * b_k * b_j\}.$$

- Podemos então projetar um algoritmo recursivo (indutivo) para resolver o problema.

Multiplicação de Matrizes - Algoritmo Recursivo

MinimoMultiplicacoesRecursivo(b, i, j)

▷ **Entrada:** Vetor b com as dimensões das matrizes e os índices i e j que delimitam o início e término da subcadeia.
▷ **Saída:** O número mínimo de multiplicações escalares necessário para computar a multiplicação da subcadeia. Esse valor é registrado em uma tabela ($m[i,j]$), bem como o índice da divisão em subcadeias ótimas ($s[i,j]$).

1. **se** $i = j$ **então devolva** 0
2. $m[i,j] := \infty$
3. **para** $k := i$ **até** $j-1$ **faça**
4. $\quad q := MinimoMultiplicacoesRecursivo(b,i,k) +$
   $\quad\quad MinimoMultiplicacoesRecursivo(b,k+1,j) +$
   $\quad\quad b[i-1] * b[k] * b[j]$
5. $\quad$ **se** $m[i,j] > q$ **então**
6. $\quad\quad m[i,j] := q$; $s[i,j] := k$
7. **devolva** $m[i,j]$.

Efetuando a Multiplicação Ótima

- É muito fácil efetuar a multiplicação da cadeia de matrizes com o número mínimo de multiplicações escalares usando a tabela s, que registra os índices ótimos de divisão em subcadeias.

MultiplicaMatrizes(M, s, i, j)

▷ **Entrada:** Cadeia de matrizes M, a tabela s e os índices i e j que delimitam a subcadeia a ser multiplicada.
▷ **Saída:** A matriz resultante da multiplicação da subcadeia entre i e j, efetuando o mínimo de multiplicações escalares.

1. **se** $i < j$ **então**
2. $\quad X := MultiplicaMatrizes(M, s, i, s[i,j])$
3. $\quad Y := MultiplicaMatrizes(M, s, s[i,j]+1, j)$
4. $\quad$ **devolva** $Multiplica(X, Y, b[i-1], b[s[i,j]], b[j])$
5. **senão devolva** M_i;

Algoritmo Recursivo - Complexidade

- O número mínimo de operações feita pelo algoritmo recursivo é dada pela recorrência:

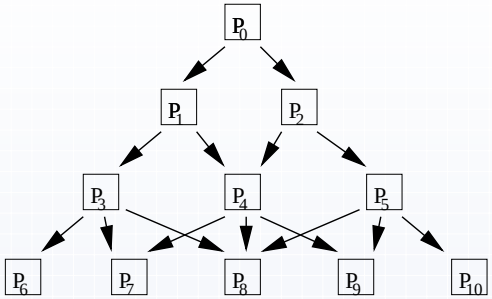
$$T(n) \geq \begin{cases} 1, & n = 1 \\ 1 + \sum_{k=1}^{n-1}[T(k) + T(n-k) + 1] & n > 1, \end{cases}$$

- Portanto, $T(n) \geq 2\sum_{k=1}^{n-1} T(i) + n$, para $n > 1$.
- É possível provar (por substituição) que $T(n) \geq 2^{n-1}$, ou seja, o algoritmo recursivo tem complexidade $\Omega(2^n)$, ainda impraticável !

Algoritmo Recursivo - Complexidade

- A ineficiência do algoritmo recursivo deve-se à sobreposição de subproblemas: o cálculo do mesmo $m[i,j]$ pode ser requerido em vários subproblemas.
- Por exemplo, para $n = 4$, $m[1,2]$, $m[2,3]$ e $m[3,4]$ são computados duas vezes.
- O número total de $m[i,j]$'s calculados é $O(n^2)$ apenas !
- Portanto, podemos obter um algoritmo mais eficiente se evitarmos recálculos de subproblemas.

Um subproblema pode aparecer várias vezes



Divisão e Conquista com Recursão simples:
P_4 resolvido 2 vezes (uma por P_1 e uma por P_2);
P_7 resolvido 3 vezes (uma por P_3 e duas por P_4);
P_8 resolvido 4 vezes (uma por P_3, duas por P_4 e uma por P_5);
P_9 resolvido 3 vezes (uma por P_3 e duas por P_4);

Memorização x Programação Dinâmica

- Existem duas técnicas para evitar o recálculo de subproblemas:
  - **Memorização:** Consiste em manter a estrutura recursiva do algoritmo, registrando em uma tabela o valor ótimo para subproblemas já computados e verificando, antes de cada chamada recursiva, se o subproblema a ser resolvido já foi computado.
  - **Programação Dinâmica:** Consiste em preencher uma tabela que registra o valor ótimo para cada subproblema de forma apropriada, isto é, a computação do valor ótimo de cada subproblema depende somente de subproblemas já previamente computados. Elimina completamente a recursão.

Algoritmo de Memorização

MinimoMultiplicacoesMemorizado(b, n)

1. **para** $i := 1$ **até** n **faça**
2. **para** $j := 1$ **até** n **faça**
3. $m[i,j] := \infty$
4. **devolva** $Memorizacao(b, 1, n)$

Memorizacao(b, i, j)

1. **se** $m[i,j] < \infty$ **então devolva** $m[i,j]$
2. **se** $i = j$ **então** $m[i,j] := 0$
3. **senão**
4. **para** $k := i$ **até** $j - 1$ **faça**
5. $q := Memorizacao(b, i, k) +$
$Memorizacao(b, k+1, j) + b[i-1] * b[k] * b[j]$;
6. **se** $m[i,j] > q$ **então** $m[i,j] := q$; $s[i,j] := k$
7. **devolva** $m[i,j]$

Algoritmo de Programação Dinâmica

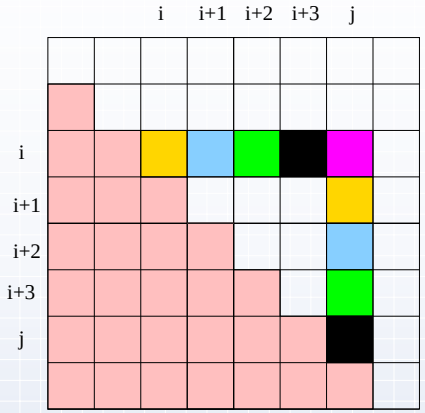
- O uso de programação dinâmica é preferível pois elimina completamente o uso de recursão.
- O algoritmo de programação dinâmica para o problema da multiplicação de matrizes torna-se trivial se computarmos, para valores crescentes de u, o valor ótimo de todas as subcadeias de tamanho u.

Algoritmo de Programação Dinâmica

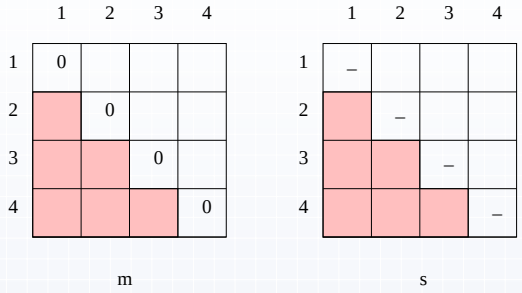
MinimoMultiplicacoes(b)

▷ **Entrada:** Vetor b com as dimensões das matrizes.
▷ **Saída:** As tabelas m e s preenchidas.
1. **para** $i = 1$ **até** n **faça** $m[i,i] := 0$
   ▷ calcula o mínimo de todas sub-cadeias de tamanho $u+1$
2. **para** $u = 1$ **até** $n-1$ **faça**
3. **para** $i = 1$ **até** $n-u$ **faça**
4. $j := i+u$; $m[i,j] := \infty$
5. **para** $k = i$ **até** $j-1$ **faça**
6. $q := m[i,k] + m[k+1,j] + b[i-1] * b[k] * b[j]$
7. **se** $q < m[i,j]$ **então**
8. $m[i,j] := q$; $s[i,j] := k$
9. **devolva** (m,s)

Algoritmo de Programação Dinâmica - Exemplo



Algoritmo de Programação Dinâmica - Exemplo



Algoritmo de Programação Dinâmica - Exemplo

	1	2	3	4
1	0	12000		
2		0	1200	
3			0	3000
4				0

m

	1	2	3	4
1	_	1		
2		_	2	
3			_	3
4				_

s

{ 200, 2, 30, 20, 5 }

Algoritmo de Programação Dinâmica - Exemplo

	1	2	3	4
1	0	12000	9200	
2		0	1200	
3			0	3000
4				0

m

	1	2	3	4
1	_	1	1	
2		_	2	
3			_	3
4				_

s

{ 200, 2, 30, 20, 5 }

b0*b1*b3=200*2*20=8000

b0*b2*b3=200*30*20=120000

Algoritmo de Programação Dinâmica - Exemplo

	1	2	3	4
1	0	12000	9200	
2		0	1200	1400
3			0	3000
4				0

m

	1	2	3	4
1	_	1	1	
2		_	2	3
3			_	3
4				_

s

{ 200, 2, 30, 20, 5 }

b1*b2*b4=2*30*5=300

b1*b3*b4=2*20*5=200

Algoritmo de Programação Dinâmica - Exemplo

	1	2	3	4
1	0	12000	9200	3400
2		0	1200	1400
3			0	3400
4				0

m

	1	2	3	4
1	_	1	1	1
2		_	2	3
3			_	3
4				_

s

{ 200, 2, 30, 20, 5 }

b0*b1*b4=200*2*5=2000

b0*b2*b4=200*30*5=30000

b0*b3*b4=200*20*5=20000

Algoritmo de Programação Dinâmica - Exemplo

	1	2	3	4
1	0	12000	9200	3400
2		0	1200	1400
3			0	3000
4				0

m

	1	2	3	4
1	_	1	1	1
2		_	2	3
3			_	3
4				_

s

M1 ((M2 . M3) . M4)

Algoritmo de Programação Dinâmica - Complexidade

- A complexidade de tempo do algoritmo é dada por:

$$\begin{aligned} T(n) &= \sum_{u=1}^{n-1} \sum_{i=1}^{n-u} \sum_{k=i}^{i+u-1} \Theta(1) \\ &= \sum_{u=1}^{n-1} \sum_{i=1}^{n-u} u\,\Theta(1) \\ &= \sum_{u=1}^{n-1} u(n-u)\,\Theta(1) \\ &= \sum_{u=1}^{n-1} (nu - u^2)\,\Theta(1). \end{aligned}$$

Algoritmo de Programação Dinâmica - Complexidade

- Como

$$\sum_{u=1}^{n-1} nu = n^3/2 - n^2/2$$

e

$$\sum_{u=1}^{n-1} u^2 = n^3/3 - n^2/2 + n/6.$$

Então,

$$T(n) = (n^3/6 - n/6)\,\Theta(1).$$

- A complexidade de tempo do algoritmo é $\Theta(n^3)$.
- A complexidade de espaço é $\Theta(n^2)$, já que é necessário armazenar a matriz com os valores ótimos dos subproblemas.

O Problema Binário da Mochila

O Problema da Mochila

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

- Podemos fazer as seguintes suposições:
  - $\sum_{i=1}^{n} w_i > W$;
  - $0 < w_i \leq W$, para todo $i = 1, \ldots, n$.

O Problema Binário da Mochila

- Podemos formular o problema da mochila com Programação Linear Inteira:
  - Criamos uma variável x_i para cada item: $x_i = 1$ se o item i estiver na solução ótima e $x_i = 0$ caso contrário.
  - A modelagem do problema é simples:

$$\max \sum_{i=1}^{n} c_i x_i \quad (1)$$

$$\sum_{i=1}^{n} w_i x_i \leq W \quad (2)$$

$$x_i \in \{0, 1\} \quad (3)$$

- (1) é a função objetivo e (2-3) o conjunto de restrições.

O Problema Binário da Mochila

- Como podemos projetar um algoritmo para resolver o problema?
- Existem 2^n possíveis subconjuntos de itens: um algoritmo de força bruta é impraticável!
- É um problema de otimização. Será que tem subestrutura ótima?
- Se o item n estiver na solução ótima, o valor desta solução será c_n mais o valor da melhor solução do problema da mochila com capacidade $W - w_n$ considerando-se só os $n - 1$ primeiros itens.
- Se o item n não estiver na solução ótima, o valor ótimo será dado pelo valor da melhor solução do problema da mochila com capacidade W considerando-se só os $n - 1$ primeiros itens.

O Problema Binário da Mochila

- Seja $z[k, d]$ o valor ótimo do problema da mochila considerando-se uma capacidade d para a mochila que contém um subconjunto dos k primeiros itens da instância original.
- A fórmula de recorrência para computar $z[k, d]$ para todo valor de d e k é:

$$z[0, d] = 0$$

$$z[k, 0] = 0$$

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

O Problema Binário da Mochila - Complexidade Recursão

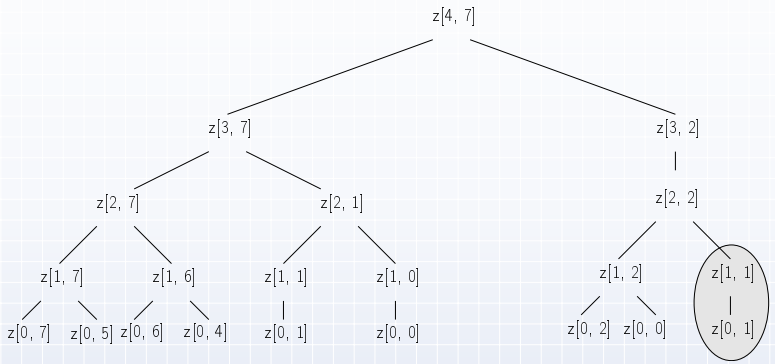
- A complexidade do algoritmo recursivo para este problema no pior caso é dada pela recorrência:

$$T(k, d) = \begin{cases} 1, & k = 0 \text{ ou } d = 0 \\ T(k-1, d) + T(k-1, d-w_k) + 1 & k > 0 \text{ e } d > 0. \end{cases}$$

- Portanto, no pior caso, o algoritmo recursivo tem complexidade $\Omega(2^n)$. É impraticável!
- Mas há sobreposição de subproblemas: o recálculo de subproblemas pode ser evitado!

Mochila - Sobreposição de Subproblemas

- Considere vetor de tamanhos $w = \{2, 1, 6, 5\}$ e capacidade da mochila $W = 7$. A árvore de recursão seria:

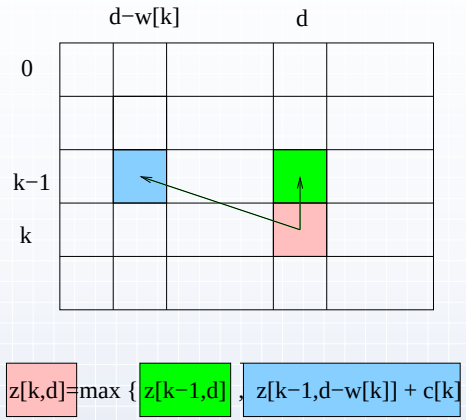


- O subproblema $z[1, 1]$ é computado duas vezes.

Mochila - Programação Dinâmica

- O número total máximo de subproblemas a serem computados é nW.
- Isso porque tanto o tamanho dos itens quanto a capacidade da mochila são inteiros!
- Podemos então usar programação dinâmica para evitar o recálculo de subproblemas.
- Como o cálculo de $z[k, d]$ depende de $z[k-1, d]$ e $z[k-1, d-w_k]$, preenchemos a tabela linha a linha.

Mochila



O Problema Binário da Mochila - Algoritmo

Mochila(c, w, W, n)

▷ **Entrada:** Vetores c e w com valor e tamanho de cada item, capacidade W da mochila e número de itens n.
▷ **Saída:** O valor máximo do total de itens colocados na mochila.

1. **para** $d := 0$ **até** W **faça** $z[0, d] := 0$
2. **para** $k := 1$ **até** n **faça** $z[k, 0] := 0$
3. **para** $k := 1$ **até** n **faça**
4. **para** $d := 1$ **até** W **faça**
5. $z[k, d] := z[k-1, d]$
6. **se** $w_k \leq d$ **e** $c_k + z[k-1, d-w_k] > z[k, d]$ **então**
7. $z[k, d] := c_k + z[k-1, d-w_k]$
8. **devolva** $(z[n, W])$

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0							
2	0							
3	0							
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0							
3	0							
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0							
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Complexidade

- Claramente, a complexidade do algoritmo de programação dinâmica para o problema da mochila é $O(nW)$.
- É um algoritmo pseudo-polinomial: sua complexidade depende do valor de W, parte da entrada do problema.
- O algoritmo não devolve o subconjunto de valor total máximo, apenas o valor máximo.
- É fácil recuperar o subconjunto a partir da tabela z preenchida.

Mochila - Recuperação da Solução

MochilaSolucao(z, n, W)

▷ **Entrada:** Tabela z preenchida, capacidade W da mochila e número de itens n.
▷ **Saída:** O vetor x que indica os itens colocados na mochila.
para $i := 1$ **até** n **faça** $x[i] := 0$
MochilaSolucaoAux(x, z, n, W)
devolva (x)

MochilaSolucaoAux(x, z, k, d)

se $k \neq 0$ **então**
  se $z[k, d] = z[k-1, d]$ **então**
    $x[k] := 0$; *MochilaSolucaoAux*$(x, z, k-1, d)$
  senão

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	**34**

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	**10**	17	17	17	25	32
4	0	7	10	17	17	24	31	**34**

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	**10**	17	17	17	17	17
3	0	7	**10**	17	17	17	25	32
4	0	7	10	17	17	24	31	**34**

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	**10**	10	10	10	10	10
2	0	7	**10**	17	17	17	17	17
3	0	7	**10**	17	17	17	25	32
4	0	7	10	17	17	24	31	**34**

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

x[1] = x[4] = 1 , x[2] = x[3] = 0

Mochila - Complexidade

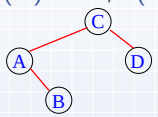
- O algoritmo de recuperação da solução tem complexidade $O(n)$.
- Portanto, a complexidade de tempo e de espaço do algoritmo de programação dinâmica para o problema da mochila é $O(nW)$.
- É possível economizar memória, registrando duas linhas: a que está sendo preenchida e a anterior. Mas isso inviabiliza a recuperação da solução.

Problema da Árvore de Busca Ótima

Problema da Árvore de Busca

Dados elementos $(e_1 < e_2 < \ldots < e_n)$, onde cada item e_i é consultado $f(e_i)$ vezes, construir uma árvore de busca binária, tal que o total de nós consultados é mínimo.

Aplicações: Construção de dicionários estáticos, processadores de texto, verificadores de ortografia.
Exemplo: Considere quatro chaves: $A \leq B \leq C \leq D$
e frequências $f(A) = 45$, $f(B) = 25$, $f(C) = 18$ e $f(D) = 12$.

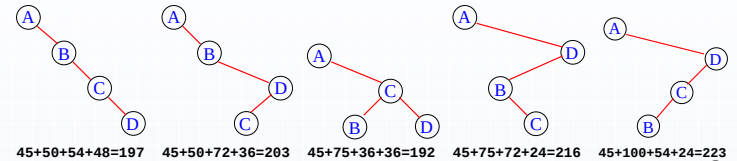


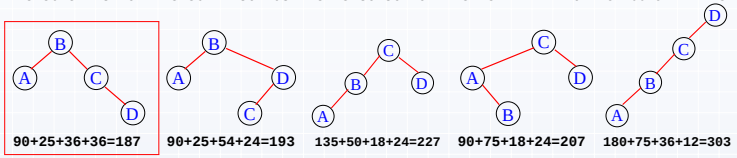
90+75+18+24=207

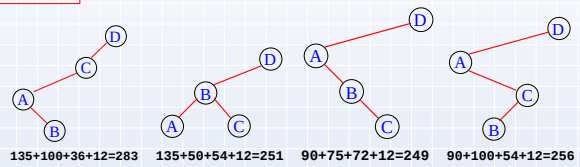
Total de nós acessados nesta árvore = 207
Podemos construir todas as árvores e escolher a melhor?

Listando as árvores

Não! Número de árvores de busca pode ser muito grande!!





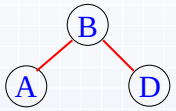


Exercício: Calcule o número de árvores distintas com n nós.

Propriedades da árvore de busca ótima

Definicao Se T é uma árvore binária de busca e v é um vértice, denotamos por $T(v)$ a subárvore enraizada em v contendo todos os vértices abaixo de v.

No exemplo anterior temos $A \leq B \leq C \leq D$.
Pergunta: Em uma árvore de busca T, podemos ter $T(B)$ nesta forma ?

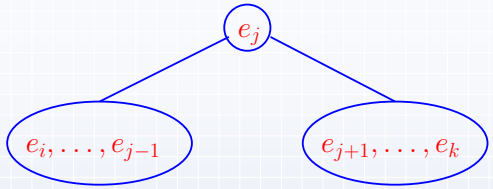


Não: Pois C deveria estar em $T(B)$.
Conclusão: Se T é uma árvore de busca, e v é um vértice de T, então $T(v)$ contém apenas elementos consecutivos.

Propriedades da árvore de busca ótima (Cont.)

Seja T uma árvore de busca, e_j um vértice de T e $T(e_j) = \{e_i, \ldots, e_{j-1}, e_j, e_{j+1}, \ldots, e_k\}$. Então

- no ramo esquerdo deve haver os elementos $e_i, \ldots, e_{j-1}$.
- no ramo direito deve haver os elementos $e_{j+1}, \ldots, e_k$.
- Sub-árvores de $\{e_i, \ldots, e_{j-1}\}$ e $\{e_{j+1}, \ldots, e_k\}$ devem ser ótimas.



- Frequência de acessos à raiz em uma árvore de busca é a soma das frequências dos elementos na árvore
- Qualquer elemento de $\{e_i, \ldots, e_k\}$ é candidato a ser raiz

Definição recursiva da solução ótima

Idéia: Gerar árvores de busca a partir de árvores de busca de tamanhos menores
Estratégia: Bottom-Up

Seja $A(e_i, \ldots, e_k)$:=número de nós acessados em árvore ótima contendo $\{e_i, \ldots, e_k\}$.

Então

- $A(\emptyset) = 0$
- $A(e_i, \ldots, e_k) =$

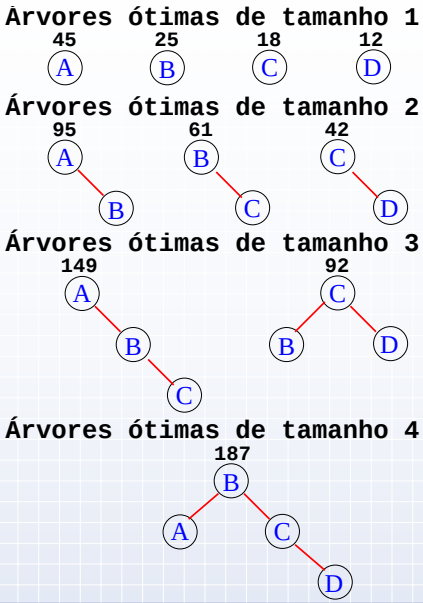
$$\min_{i \leq j \leq k} \left\{ A(e_i, \ldots, e_{j-1}) + A(e_{j+1}, \ldots, e_k) + \sum_{t=i}^{k} f(e_t) \right\}$$

Tabela

Item \ Tam.Seq.	0	1	...	t	...	n
e_1	0	$f(e_1)$	...		...	$M(1,n)$
e_2	0	$f(e_2)$	...		...	
$\vdots$	$\vdots$	$\vdots$	...		...	
e_i	0	$f(e_i)$	...	$M(i,t) := A(e_i, \ldots, e_k)$	...	
$\vdots$	0	$\vdots$	...		...	
$(k=i+t-1)$ e_k	0	$f(e_k)$	...			
$\vdots$	0	$\vdots$	...		...	
e_n	0	$f(e_n)$	...		...	

$A(\emptyset) = 0$
$A(e_i) = f(e_i)$

$$A(e_i, \ldots, e_k) = \min_{i \leq j \leq k} \left\{ A(e_i, \ldots, e_{j-1}) + A(e_{j+1}, \ldots, e_k) + \sum_{t=i}^{k} f(e_t) \right\}$$



Algoritmo

ALGORITMO ÁRVORE-BUSCA$(e_1, \ldots, e_n, f)$

1 para $i \leftarrow 0$ até $n+1$ faça $M(i,0) \leftarrow 0$
2 para $t \leftarrow 1$ até n faça
3 para $i \leftarrow 1$ até $n-t+1$ faça
5 $S \leftarrow f(e_i) + f(e_{i+1}) + \cdots + f(e_{i+t-1})$
7 $M(i,t) \leftarrow \min_{0 \le t' \le t-1} \left\{ M(i,t') + M(i+t'+1, t-t'-1) + S \right\}$

Solução em: $A(1,n)$

Teorema

O algoritmo ÁRVORE-BUSCA *encontra o valor da árvore de busca ótima em tempo* $O(n^3)$.

Exercícios:

- O algoritmo apresentado para resolver o problema da árvore ótima apenas apresenta o valor esperado de consultas de nós para todos os itens. Faça uma implementação do algoritmo de maneira que ele apresente a árvore de busca ótima.

Subcadeia comum máxima

Definição: Subcadeia

Dada uma cadeia $S = a_1 \ldots a_n$, dizemos que $S' = b_1 \ldots b_p$ é uma *subcadeia* de S se existem p índices $i(j)$ satisfazendo:

(a) $i(j) \in \{1, \ldots, n\}$ para todo $j \in \{1, \ldots, p\}$;
(b) $i(j) < i(j+1)$ para todo $j \in \{1, \ldots, p-1\}$;
(c) $b_j = a_{i(j)}$ para todo $j \in \{1, \ldots, p\}$.

- **Exemplo:** $S = ABCDEFG$ e $S' = ADFG$.

Problema da Subcadeia Comum Máxima

Dadas duas cadeias de caracteres X e Y de um alfabeto Σ, determinar a maior subcadeia comum de X e Y

Subcadeia comum máxima (cont.)

- É um problema de otimização. Será que tem subestrutura ótima?
- **Notação:** Seja S uma cadeia de tamanho n. Para todo $i = 1, \ldots, n$, o prefixo de tamanho i de S será denotado por S_i.
- **Exemplo:** Para $S = ABCDEFG$, $S_2 = AB$ e $S_4 = ABCD$.
- **Definição:** $c[i,j]$ é o tamanho da subcadeia comum máxima dos prefixos X_i e Y_j. Logo, se $|X| = m$ e $|Y| = n$, $c[m,n]$ é o valor ótimo.

Subcadeia comum máxima (cont.)

- **Teorema (subestrutura ótima):** Seja $Z = z_1 \ldots z_k$ a subcadeia comum máxima de $X = x_1 \ldots x_m$ e $Y = y_1 \ldots y_n$, denotado por $Z = \texttt{SCM}(X, Y)$.
  1. Se $x_m = y_n$ então $z_k = x_m = y_n$ e $Z_{k-1} = \texttt{SCM}(X_{m-1}, Y_{n-1})$.
  2. Se $x_m \neq y_n$ então $z_k \neq x_m$ implica que $Z = \texttt{SCM}(X_{m-1}, Y)$.
  3. Se $x_m \neq y_n$ então $z_k \neq y_n$ implica que $Z = \texttt{SCM}(X, Y_{n-1})$.
- **Fórmula de Recorrência:**

$$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ c[i-1, j-1] + 1 & \text{se } i,j > 0 \text{ e } x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{se } i,j > 0 \text{ e } x_i \neq y_j \end{cases}$$

Subcadeia comum máxima (cont.)

SCM(X, m, Y, n, c, b)

```
01. para i = 0 até m faça c[i,0] := 0
02. para j = 1 até n faça c[0,j] := 0
03. para i = 1 até m faça
04.     para j = 1 até n faça
05.         se x_i = y_j então
06.             c[i,j] := c[i-1,j-1] + 1 ; b[i,j] := "↖"
07.         senão
08.             se c[i,j-1] > c[i-1,j] então
09.                 c[i,j] := c[i,j-1] ; b[i,j] := "←"
10.             senão
11.                 c[i,j] := c[i-1,j]; b[i,j] := "↑";
12. devolva (c[m,n], b).
```


Subcadeia comum máxima - Exemplo

- **Exemplo:** $X = abcb$ e $Y = bdcab$, $m = 4$ e $n = 5$.

X \ Y		b	d	c	a	b
	0	0	0	0	0	0
a	0	0	0	0	1	1
b	0	1	1	1	1	2
c	0	1	1	2	2	2
b	0	1	1	2	2	3

X \ Y		b	d	c	a	b
	0					
a	1		↑	↑	↑	↖	←
b	2		↖	←	←	↑	↖
c	3		↑	↑	↖	←	↑
b	4		↖	↑	↑	↑	↖

Subcadeia comum máxima - Complexidade

- Claramente, a complexidade do algoritmo é $O(mn)$.
- O algoritmo não encontra a subcadeia comum de tamanho máximo, apenas seu tamanho.
- Com a tabela b preenchida, é fácil encontrar a subcadeia comum máxima.

Subcadeia comum máxima (cont.)

- Para recuperar a solução, basta chamar $Recupera\_MSC(b, X, m, n)$.

Recupera_SCM(b, X, i, j)

```
1. se i = 0 e j = 0 então devolva
2. se b[i, j] = "↖" então
3.    Recupera_MSC(b, X, i − 1, j − 1); imprima x_i
4. senão
5.    se b[i, j] = "↑" então
6.       Recupera_MSC(b, X, i − 1, j)
7.    senão
8.       Recupera_MSC(b, X, i, j − 1)
```


Subcadeia comum máxima - Complexidade

- A determinação da subcadeia comum máxima é feita em tempo $O(m + n)$ no pior caso.
- Portanto, a complexidade de tempo e de espaço do algoritmo de programação dinâmica para o problema da subcadeia comum máxima é $O(mn)$.
- Note que a tabela b é dispensável, podemos economizar memória recuperando a solução a partir da tabela c. Ainda assim, o gasto de memória seria $O(mn)$.
- Caso não haja interesse em determinar a subcadeia comum máxima, mas apenas seu tamanho, é possível reduzir o gasto de memória para $O(\min\{n, m\})$: basta registrar apenas a linha da tabela sendo preenchida e a anterior.

Algoritmos gulosos

Algoritmos Gulosos: Conceitos Básicos

- Tipicamente algoritmos gulosos são utilizados para resolver problemas de **otimização**.
- Uma característica comum dos problemas onde se aplicam algoritmos gulosos é a existência **subestrutura ótima**, semelhante à programação dinâmica!
- Programação dinâmica: tipicamente os subproblemas são resolvidos à otimalidade **antes** de se proceder à escolha de um elemento que irá compor a solução ótima.
- Algoritmo Guloso: primeiramente é feita a escolha de um elemento que irá compor a solução ótima e só **depois** um subproblema é resolvido.

Algoritmos Gulosos: Conceitos Básicos

- Um algoritmo guloso sempre faz a escolha que parece ser a *"melhor"* a cada iteraçãousando um **critério guloso**.
  É uma decisão **localmente** ótima.
- Propriedade da **escolha gulosa**: garante que a cada iteração é tomada uma decisão que irá levar a um ótimo global.
- Em um algoritmo guloso uma escolha que foi feita **nunca é revista**, ou seja, não há qualquer tipo de *backtracking*.
- Em geral é fácil projetar ou descrever um algoritmo guloso. O **difícil** é provar que ele funciona!

Seleção de Atividades

- $S = \{a_1, \ldots, a_n\}$: conjunto de n atividades que podem ser executadas em um mesmo local. Exemplo: palestras em um auditório.
- Para todo $i = 1, \ldots, n$, a atividade a_i começa no instante s_i e termina no instante f_i, com $0 \leq s_i < f_i < \infty$.
  Ou seja, supõe-se que a atividade a_i será executada no intervalo de tempo (semi-aberto) $[s_i, f_i)$.

Definição

As atividades a_i e a_j são ditas **compatíveis** se os intervalos $[s_i, f_i)$ e $[s_j, f_j)$ são disjuntos.

Problema de Seleção de Atividades

Encontre em S um subconjunto de atividades mutuamente compatíveis que tenha tamanho **máximo**.

Seleção de Atividades

- Exemplo:

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

- Pares de atividades incompatíveis: (a_1, a_2), (a_1, a_3)
  Pares de atividades compatíveis: (a_1, a_4), (a_4, a_8)
- Conjunto **maximal** de atividades compatíveis: (a_3, a_9, a_{11}).
- Conjunto **máximo** de atividades compatíveis: (a_1, a_4, a_8, a_{11}).

As atividades estão ordenadas em ordem crescente de instantes de término! Isso será importante mais adiante.

Seleção de Atividades

- Vimos que tanto os algoritmos gulosos quanto aqueles que usam programação dinâmica valem-se da existência da propriedade de subestrutura ótima.
- Inicialmente verificaremos que o problema da seleção de atividades tem esta propriedade e, então, projetaremos um algoritmo por programação dinâmica.
- Em seguida, mostraremos que há uma forma de resolver uma quantidade consideravelmente menor de subproblemas do que é feito na programação dinâmica.
- Isto será garantido por uma **propriedade de escolha gulosa**, a qual dará origem a um algoritmo guloso.
- Este processo auxiliará no entendimento da diferença entre estas duas técnicas de projeto de algoritmos.

Seleção de Atividades

Suponha que $f_1 \leq f_2 \leq \ldots \leq f_n$, ou seja, as atividades estão ordenadas em ordem crescente de instantes de término.

Definição

Denote $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$, ou seja, S_{ij} é o conjunto de atividades que começam depois do término de a_i e terminam antes do início de a_j.

- Atividades artificiais: a_0 com $f_0 = 0$ e a_{n+1} com $s_{n+1} = \infty$
- Tem-se que $S = S_{0,n+1}$ e, com isso, S_{ij} está bem definido para qualquer par (i, j) tal que $0 \leq i, j \leq n + 1$.
- Note que $S_{ij} = \emptyset$ para todo $i \geq j$.
  Por quê?

Seleção de Atividades

- **Subestrutura ótima**: considere o *subproblema* da seleção de atividades definido sobre S_{ij}. Suponha que a_k pertence a uma solução ótima de S_{ij}.

  Como $f_i \leq s_k < f_k \leq s_j$, uma solução ótima para S_{ij} que contenha a_k será composta pelas atividades de uma solução ótima de S_{ik}, pelas atividades de uma solução ótima de S_{kj} e por a_k.
  Por quê?

Seleção de Atividades

- **Definição**: para todo $0 \leq i, j \leq n + 1$, seja $c[i, j]$ o valor ótimo do problema de seleção de atividades para a instância S_{ij}.
- Deste modo, o valor ótimo do problema de seleção de atividades para instância $S = S_{0,n+1}$ é $c[0, n + 1]$.
- **Fórmula de recorrência**:

$$c[i,j] = \begin{cases} 0 & \text{se } S_{ij} = \emptyset \\ \max\limits_{i<k<j : a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{se } S_{ij} \neq \emptyset \end{cases}$$

Agora é fácil escrever o algoritmo de programação dinâmica. (Exercício.)

Seleção de Atividades

Podemos "converter" o algoritmo de programação dinâmica em um algoritmo guloso se notarmos que o primeiro resolve subproblemas desnecessariamente.

Teorema: (escolha gulosa)

Considere o subproblema definido para uma instância não vazia S_{ij}, e seja a_m a atividade de S_{ij} com o menor tempo de término, i.e.:

$$f_m = \min\{f_k : a_k \in S_{ij}\}.$$

Então **(a)** existe uma solução ótima para S_{ij} que contém a_m e **(b)** S_{im} é vazio e o subproblema definido para esta instância é trivial, portanto, a escolha de a_m deixa apenas um dos subproblemas com solução possivelmente não trivial, já que S_{mj} pode não ser vazio.

Seleção de Atividades

Método geral para provar que uma algoritmo guloso funciona

- Mostre que o problema tem subestrutura ótima.
- Mostre que se a foi a primeira escolha do algoritmo, então **existe** alguma solução ótima que contém a. Segue então por indução e pela subestrutura ótima que o algoritmo sempre faz escolhas corretas.

Seleção de Atividades

Vou mostrar que existe uma solução ótima para $S = S_{0,n+1}$ que contém a_m.

Seja A um conjunto de atividades mutuamente compatíveis de tamanho máximo em S_{ij}. Se $a_m \in A$ então nada há a fazer. Suponha então que $a_m \notin A$.

Seja $a_k \in A$ com menor f_k. Seja $A' = A - \{a_k\} \cup \{a_m\}$. Então A' também é conjunto de atividades mutuamente compatíveis de tamanho máximo. (Por quê?)

Este parece ser um truque importante: modificar uma solução ótima "*genérica*" e obter uma solução ótima com a(s) escolha(s) gulosa(s). Tenho que me lembrar disso.

Seleção de Atividades

Usando o teorema anterior, um modo simples de projetar um algoritmo seria o seguinte:

- Suponha que estamos tentando resolver S_{ij}.
- Determine a atividade a_m com menor tempo de término em S_{ij}.
- Resolva o subproblema S_{mj} e junte a_m à solução obtida na recursão. Devolva este conjunto de atividades.

Seleção de Atividades

SelecAtivGulRec(s, f, i, j)

▷ **Entrada:** vetores s e f com instantes de início e término das atividades $a_i, a_{i+1}, \ldots, a_j$, sendo $f_i \leq \ldots \leq f_j$.
▷ **Saída:** conjunto de tamanho máximo de índices de atividades mutuamente compatíveis.

1. $m \leftarrow i + 1$;
   ▷ Busca atividade com menor tempo de término que está em S_{ij}
2. **enquanto** $m < j$ **e** $s_m < f_i$ **faça** $m \leftarrow m + 1$;
3. **se** $m \geq j$ **então devolva** $\emptyset$;
4. **senão**
5. **se** $f_m > s_j$ **então devolva** $\emptyset$; ▷ $a_m \notin S_{ij}$
6. **senão devolva** $\{a_m\} \cup \text{SelecAtivGulRec}(s, f, m, j)$.

Seleção de Atividades

- A chamada inicial será $\textit{SelecAtivGulRec}(s, f, 0, n + 1)$.
- **Complexidade**: $\Theta(n)$.
  Ao longo de todas as chamadas recursivas,cada atividade é examinada exatamente uma vez no laço da linha 2. Em particular, a atividade a_k é examinada na última chamada com $i < k$.
- Como o algoritmo anterior é um caso simples de recursão caudal, é trivial escrever uma versão iterativa do mesmo.

Seleção de Atividades

SelecAtivGulIter(s, f, n)

▷ **Entrada:** vetores s e f com instantes de início e término das n atividades com os instantes de término em ordem crescente.
▷ **Saída:** um conjunto A de tamanho máximo contendo atividades mutuamente compatíveis.

1. $A \leftarrow \{a_1\}$;
2. $i \leftarrow 1$;
3. **para** $m \leftarrow 2$ **até** n **faça**
4. **se** $s_m \geq f_i$ **então**
5. $A \leftarrow A \cup \{a_m\}$;
6. $i \leftarrow m$;
7. **devolva** A.

Seleção de Atividades

- Observe que na linha 3, i é o índice da última atividade colocada em A. Como as atividades estão ordenadas pelo instante de término, tem-se que:

$$f_i = \max\{f_k : a_k \in A\},$$

  ou seja, f_i é sempre o maior instante de término de uma atividade em A.
- Pode-se concluir que o algoritmo faz as mesmas escolhas de *SelecAtivGulRec* e portanto, está correto.
- **Complexidade**: $\Theta(n)$.

Códigos de Huffman

- Códigos de Huffman: técnica de compressão de dados.
- Reduções no tamanho dos arquivos dependem das características dos dados contidos nos mesmos. Valores típicos oscilam entre 20 e 90%.
- Exemplo: arquivo texto contendo 100.000 caracteres no alfabeto $\Sigma = \{a, b, c, d, e, f\}$. As *frequências* de cada caracter no arquivo são indicadas na tabela abaixo.

	a	*b*	*c*	*d*	*e*	*f*
Frequência (em milhares)	45	13	12	16	9	5
Código de tamanho fixo	000	001	010	011	100	101
Código de tamanho variável	0	101	100	111	1101	1100

- Codificação do arquivo: representar cada caracter por uma sequência de *bits*
- Alternativas:
  1. sequências de tamanho fixo.
  2. sequências de tamanho variável.

Códigos de Huffman

- Qual o tamanho (em *bits*) do arquivo comprimido usando os códigos acima ?
- Códigos de tamanho fixo: $3 \times 100.000 = 300.000$
  Códigos de tamanho variável:

$$(\underbrace{45 \times 1}_{a} + \underbrace{13 \times 3}_{b} + \underbrace{12 \times 3}_{c} + \underbrace{16 \times 3}_{d} + \underbrace{9 \times 4}_{e} + \underbrace{5 \times 4}_{f}) \times 1.000 = 224.000$$

Ganho de $\approx 25\%$ em relação à solução anterior.

Problema da Codificação:

Dadas as frequências de ocorrência dos caracteres de um arquivo, encontrar as sequências de *bits* (códigos) para representá-los de modo que o arquivo comprimido tenha tamanho mínimo.

Códigos de Huffman

Definição:

Códigos livres de prefixo são aqueles onde, dados dois caracteres quaisquer i e j representados pela codificação, a sequência de *bits* associada a i **não** é um *prefixo* da sequência associada a j.

Importante:

Pode-se provar que sempre **existe** uma solução ótima do problema da codificação que é dado por um código *livre de prefixo*.

Códigos de Huffman – codificação

O processo de codificação, i.e, de geração do arquivo comprimido é sempre fácil pois reduz-se a concatenar os códigos dos caracteres presentes no arquivo original em sequência.

Exemplo: usando a codificação de tamanho variável do exemplo anterior, o arquivo original dado por abc seria codificado por 0101100.

Códigos de Huffman – decodificação

- A vantagem dos códigos livres de prefixo se torna evidente quando vamos decodificar o arquivo comprimido.
- Como nenhum código é prefixo de outro código, o código que se encontra no início do arquivo comprimido não apresenta ambigüidade. Pode-se simplesmente identificar este código inicial, traduzi-lo de volta ao caracter original e repetir o processo no restante do arquivo comprimido.
- Exemplo: usando a codificação de tamanho variável do exemplo anterior, o arquivo comprimido contendo os *bits* 001011101 divide-se de **forma unívoca** em 0 0 101 1101, ou seja, corresponde ao arquivo original dado por *aabe*.

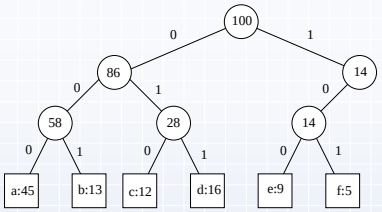
Códigos de Huffman

- Como representar de maneira conveniente uma codificação livre de prefixo de modo a facilitar o processo de decodificação?
- Solução: usar uma árvore binária.
  O filho esquerdo está associado ao *bit* ZERO enquanto o filho direito está associado ao *bit* UM. Nas folhas encontram-se os caracteres presentes no arquivo original.

Códigos de Huffman

Vejamos como ficam as árvores que representam os códigos do exemplo anterior.

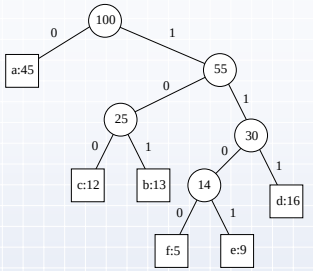
	a	*b*	*c*	*d*	*e*	*f*
Frequência	45	13	12	16	9	5
Código fixo	000	001	010	011	100	101



Códigos de Huffman

Vejamos como ficam as árvores que representam os códigos do exemplo anterior.

	a	*b*	*c*	*d*	*e*	*f*
Frequência	45	13	12	16	9	5
Código variável	0	101	100	111	1101	1100



Códigos de Huffman

- Pode-se mostrar (Exercício!) que uma codificação ótima sempre pode ser representada por uma árvore binária cheia, na qual cada vértice interno tem exatamente dois filhos.
- Então podemos restringir nossa atenção às árvores binárias cheias com $|C|$ folhas e $|C|-1$ vértices internos (Exercício!), onde C é o conjunto de caracteres do alfabeto no qual está escrito o arquivo original.

Códigos de Huffman

Computando o tamanho do arquivo comprimido:

Se T é a árvore que representa a codificação, $d_T(c)$ é a profundidade da folha representado o caracter c e $f(c)$ é a sua frequência, o tamanho do arquivo comprimido será dado por:

$$B(T) = \sum_{c \in C} f(c) d_T(c).$$

Dizemos que $B(T)$ é o **custo** da árvore T.
Isto é exatamente o tamanho do arquivo codificado.

Códigos de Huffman

- **Idéia do algoritmo de Huffman**: Começar com $|C|$ folhas e realizar sequencialmente $|C|-1$ operações de "intercalação" de dois vértices da árvore.
  Cada uma destas intercalações dá origem a um novo vértice interno, que será o pai dos vértices que participaram da intercalação.
- A escolha do par de vértices que dará origem a intercalação em cada passo depende da soma das frequências das folhas das subárvores com raízes nos vértices que ainda não participaram de intercalações.

Algoritmo de Huffman

Huffman(C)

▷ **Entrada:** Conjunto de caracteres C e as frequências f dos caracteres em C.
▷ **Saída:** raiz de uma árvore binária representando uma codificação ótima livre de prefixos.

1. $n \leftarrow |C|$;
   ▷ Q é fila de prioridades dada pelas frequências dos vértices ainda não intercalados
2. $Q \leftarrow C$;
3. **para** $i \leftarrow 1$ **até** $n-1$ **faça**
4. **alocar novo registro** z; ▷ vértice de T
5. $z.esq \leftarrow x \leftarrow$ EXTRAI_MIN(Q);
6. $z.dir \leftarrow y \leftarrow$ EXTRAI_MIN(Q);
7. $z.f \leftarrow x.f + y.f$;
8. INSERE(Q, z);
9. **retorne** EXTRAI_MIN(Q).

Corretude do algoritmo de Huffman

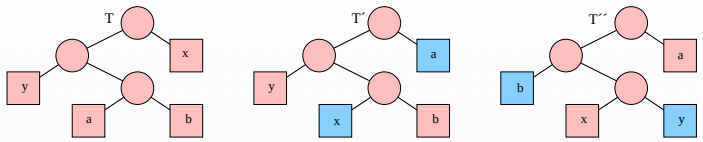
Lema 1: (escolha gulosa)

Seja C um alfabeto onde cada caracter $c \in C$ tem frequência $f[c]$. Sejam x e y dois caracteres em C com as **menores** frequências. Então, existe **um** código ótimo livre de prefixo para C no qual os códigos para x e y tem o mesmo comprimento e diferem apenas no último bit.

Prova do Lema 1:

- Seja T uma árvore **ótima**.
- Sejam a e b duas folhas "irmãs" (i.e. usadas em uma intercalação) mais profundas de T e x e y as folhas de T de menor frequência.
- **Idéia:** a partir de T, obter uma outra árvore **ótima** T' com x e y sendo duas folhas "irmãs".

Corretude do algoritmo de Huffman



$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\ &= f[x]d_T(x) + f[a]d_T(a) - f[x]d_{T'}(x) - f[a]d_{T'}(a) \\ &= f[x]d_T(x) + f[a]d_T(a) - f[x]d_T(a) - f[a]d_T(x) \\ &= (f[a] - f[x])(d_T(a) - d_T(x)) \geq 0 \end{aligned}$$

Assim, $B(T) \geq B(T')$.

Analogamente $B(T') \geq B(T'')$.

Como T é ótima, T'' é ótima e o resultado vale. $\square$

Corretude do algoritmo de Huffman

Lema 2: (subestrutura ótima)

Seja C um alfabeto com frequência $f[c]$ definida para cada caracter $c \in C$. Sejam x e y dois caracteres de C com as menores frequências. Seja C' o alfabeto obtido pela remoção de x e y e pela inclusão de um **novo** caracter z, ou seja, $C' = C \cup \{z\} - \{x, y\}$. As frequências dos caracteres em $C' \cap C$ são as mesmas que em C e $f[z]$ é definida como sendo $f[z] = f[x] + f[y]$.
Seja T' uma árvore binária representado um código ótimo livre de prefixo para C'. Então a árvore binária T obtida de T' substituindo-se o vértice (folha) z pela por um vértice interno tendo x e y como fihos, representa uma código ótimo livre de prefixo para C.

Corretude do algoritmo de Huffman

Prova do Lema 2:

- Comparando os custos de T e T':
  - Se $c \in C - \{x, y\}$, $f[c]d_T(c) = f[c]d_{T'}(c)$.
  - $f[x]d_T(x) + f[y]d_T(y) = (f[x] + f[y])(d_{T'}(z) + 1) = f[z]d_{T'}(z) + (f[x] + f[y])$.
- Logo, $B(T') = B(T) - f[x] - f[y]$.
- Por contradição, suponha que existe T'' tal que $B(T'') < B(T)$.
  Pelo lema anterior, podemos supor que x e y são folhas "irmãs" em T''. Seja T''' a árvore obtida de T'' pela substituição de x e y por uma folha z com frequência $f[z] = f[x] + f[y]$. O custo de T''' é tal que

$$B(T''') = B(T'') - f[x] - f[y] < B(T) - f[x] - f[y] = B(T'),$$

contradizendo a hipótese de que T' é uma árvore ótima para C'. $\square$

Corretude do algoritmo de Huffman

Teorema:

O algoritmo de Huffman constrói um código ótimo (livre de prefixo).

Segue imediatamente dos Lemas 1 e 2.

Passos do projeto de algoritmos gulosos: resumo

1. Formule o problema como um **problema de otimização** no qual uma escolha é feita, restando-nos então resolver um único subproblema a resolver.
2. Provar que existe sempre uma solução ótima do problema que atende à **escolha gulosa**, ou seja, a escolha feita pelo algoritmo guloso é segura.
3. Demonstrar que, uma vez feita a escolha gulosa, o que resta a resolver é um subproblema tal que se combinarmos a resposta ótima deste subproblema com o(s) elemento(s) da escolha gulosa, chega-se à solução ótima do problema original.
   Esta é a parte que requer mais engenhosidade!
   Normalmente a prova começa com uma solução ótima *genérica* e a modificamos até que ela inclua o(s) elemento(s) identificados pela escolha gulosa.