CTC-12

Projeto e Análise de Algoritmos

Carlos Alberto Alonso Sanches

CTC-12

10) Paradigmas de Programação

Divisão-e-Conquista, Método Guloso, Programação Dinâmica

Três paradigmas

- Muitos problemas são resolvidos através de três técnicas de programação:
  - *Divisão-e-Conquista* : subproblemas análogos e disjuntos; resoluções recursivas; combinação entre as subsoluções.
  - *Método Guloso* : solução incremental; otimização de um critério local.
  - *Programação Dinâmica* : subproblemas análogos, mas sobrepostos; resoluções em ordem crescente de tamanho; armazenamento das subsoluções em tabelas.

Divisão-e-Conquista (*Divide-and-Conquer*)

- Esta técnica de programação consiste em três fases:
  - Divisão: o problema é dividido em subproblemas análogos e disjuntos (não há sobreposição de subproblemas).
  - Conquista: por serem análogos, esses subproblemas são resolvidos recursivamente. Quando se tornam suficientemente pequenos, a resolução baseia-se geralmente na "força bruta".
    - O uso da recursão é eficiente porque os subproblemas são disjuntos.
  - Combinação: a partir das subsoluções obtidas, gera-se a solução do problema inicial.
- Exemplos já vistos: Torres de Hanoi, *MergeSort*, *QuickSort*, rede bitônica, exploração em profundidade.

Busca binária

- Busca binária é uma técnica muito conhecida para se encontrar um determinado valor em um vetor já ordenado.
- Também pode ser vista como um exemplo do paradigma da *Divisão-e-Conquista* :
  - Dividir o vetor em duas metades
    - São partes disjuntas
  - Procurar o valor em cada uma delas
    - São problemas análogos ao original
  - Combinar os resultados para dar a resposta final

Algoritmo

Verificação da presença de x no vetor ordenado v[l..r]:

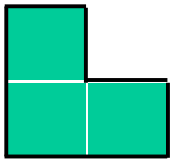
```
bool BinarySearch(l, r, x) {
    if (r < l) return false;
    q = ⌊(l+r)/2⌋;
    if (v[q] == x) return true;
    if (v[q] > x) return BinarySearch(l, q-1, x);
    return BinarySearch(q+1, r, x);
}
```

- Tamanho do vetor: n = r-l+1
- Se n = 1, T(n) = a
- Se n > 1, T(n) = T(⌊n/2⌋) + b

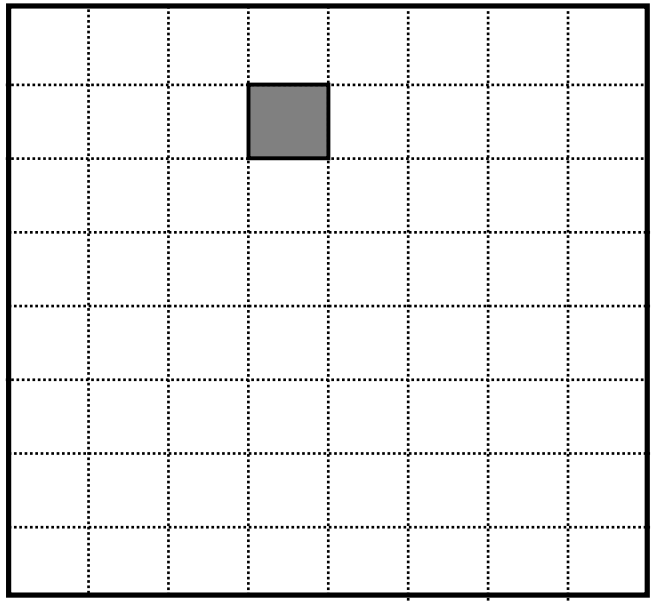
Análise do tempo

- Supondo $n = 2^k$:
  - $T(n) = T(n/2) + b$
  - $T(n) = (T(n/4) + b) + b = T(n/2^2) + 2b$
  - $T(n) = (T(n/8) + b) + 2b = T(n/2^3) + 3b$
  - Generalizando: $T(n) = T(n/2^k) + kb$
- Substituindo $n = 2^k$:
  - $T(n) = T(1) + b.\lg n = a + b.\lg n$
  - Portanto, $T(n) = O(\log n)$
- A generalização para n qualquer não muda a ordem de T(n).

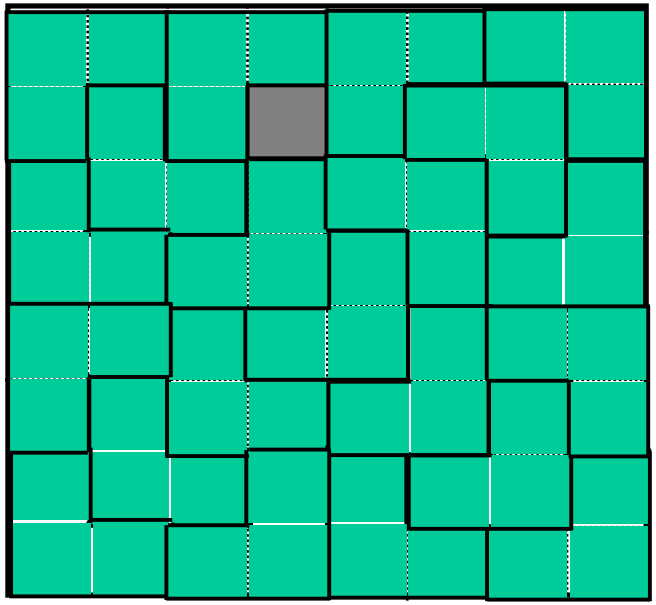
Preenchimento com *treminós*

Um *treminó*: 

Quadro $2^n x 2^n$ com um espaço vago:

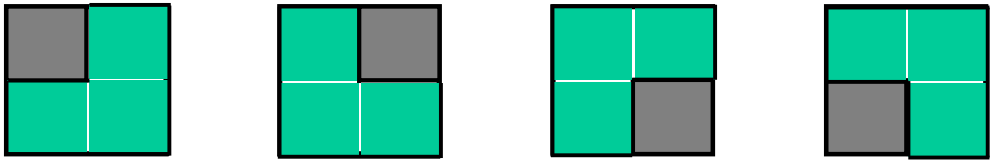


Um possível preenchimento com *treminós*:



O caso trivial

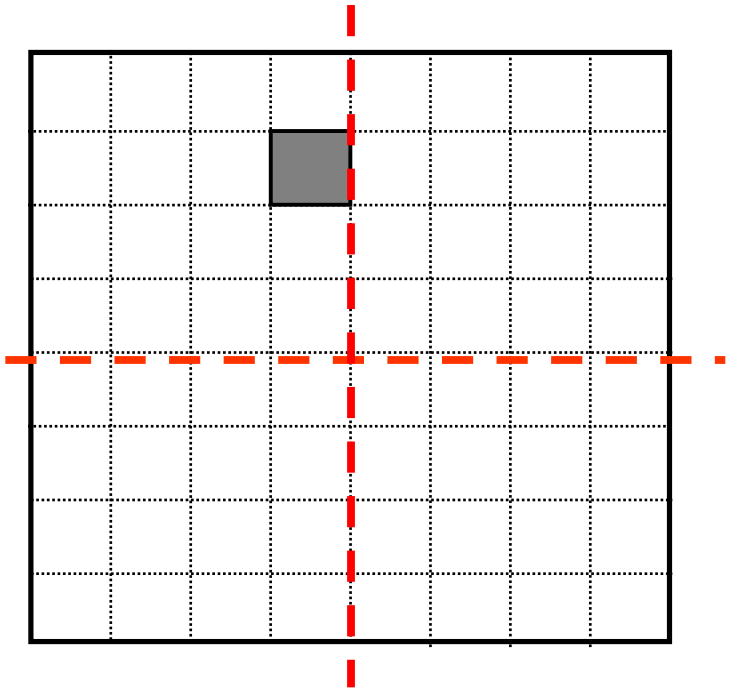
- O caso trivial (n = 1) consiste no preenchimento de um quadro 2x2 com um espaço vago.
- Esse caso admite 4 possíveis soluções:



- A ideia é encontrar um modo de reduzir gradativamente o tamanho do problema original, até se atingir situações correspondentes ao caso trivial...

A divisão do problema

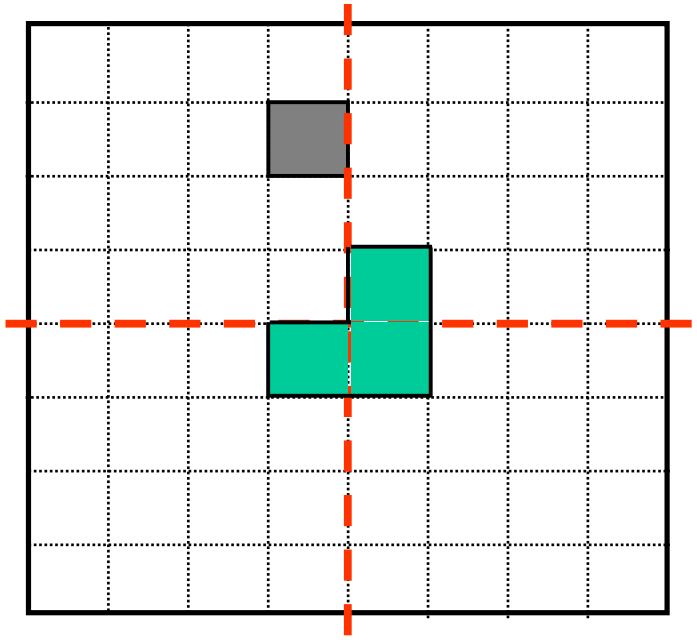
- Podemos dividir o quadro original em outros menores:



- Desse modo, temos 4 quadros de tamanho $2^{n-1}x2^{n-1}$
- No entanto, somente um dos problemas é análogo ao original, pois os demais não têm um espaço vago...

A divisão do problema

- Ideia: inserir um *treminó* no centro do quadro original, de tal forma que cada subproblema fique com um único espaço vago.



- Agora, todos os quadros de tamanho $2^{n-1}x2^{n-1}$ têm um espaço vago.
- Se seguirmos esse processo de divisão até chegarmos aos quadros de tamanho 2x2, todo o quadro original será preenchido.

Algoritmo

- Se o quadro é 2x2, resolva-o trivialmente.
- Caso contrário:
  - Divisão: Divida o quadro em quatro partes iguais; nas três partes sem espaço vago, introduza um espaço em seu canto central, de tal modo que os quatro problemas fiquem semelhantes ao original.
  - Conquista: Resolva recursivamente os quatro subproblemas.
  - Combinação: Junte as subsoluções e acrescente um *treminó* nas posições centrais, cobrindo os três espaços vagos introduzidos.

Multiplicação de inteiros

- Para efetuar a multiplicação de dois números inteiros, separamos os seus dígitos e usamos a tabuada, ou seja, utilizamos um algoritmo de *Divisão-e-Conquista*.
- Por exemplo, sejam os números d_1d_2 e d_3d_4 escritos na base 10, onde d_i são dígitos.
- A multiplicação entre eles pode ser obtida com o cálculo de $d_3.d_1.10^2 + (d_3.d_2 + d_4.d_1).10 + d_4.d_2$.
  - As multiplicações por potência de 10 são feitas através de *shifts*.
  - As multiplicações entre dígitos correspondem à fase da conquista, e as adições e os *shifts* correspondem à fase de combinação.
- Esse processo pode ser generalizado para uma maior quantidade de dígitos, e também para números escritos em notação binária.

Multiplicação de inteiros

- Sejam I e J números com n *bits*, escritos respectivamente como $I_h I_l$ e $J_h J_l$, onde I_h, I_l, J_h e J_l têm n/2 *bits*.
- Portanto, $I = I_h.2^{n/2} + I_l$ e $J = J_h.2^{n/2} + J_l$.
- O cálculo da multiplicação entre esses números seria $I.J = (I_h.2^{n/2} + I_l).(J_h.2^{n/2} + J_l) = I_h.J_h.2^n + (I_h.J_l + I_l.J_h).2^{n/2} + I_l.J_l$.
- Nesse cálculo realizado sobre números de n/2 *bits*, há 4 multiplicações, 3n/2 *shifts* e 3 adições. Como os *shifts* são realizados em tempo constante e as adições em tempo linear, o tempo total do cálculo é $T(n) = 4T(n/2) + \Theta(n)$.
- É fácil verificar que esse algoritmo de *Divisão-e-Conquista* gasta tempo $T(n) = \Theta(n^2)$.

Um algoritmo melhor

- Karatsuba (1960) vislumbrou outra maneira de fazer esse cálculo.
- Vimos que $I.J = I_h.J_h.2^n + (I_h.J_l + I_l.J_h).2^{n/2} + I_l.J_l$
- Sejam $X=I_h.J_h$, $Y=I_l.J_l$ e $Z=(I_h+I_l).(J_h+J_l)-X-Y$
- É fácil verificar que $I.J = X.2^n + Z.2^{n/2} + Y$
- O tempo agora é $T(n) = 3T(n/2) + cn$
- O algoritmo de Karatsuba é considerado o precursor do paradigma da *Divisão-e-Conquista*.

Cálculo da complexidade de tempo

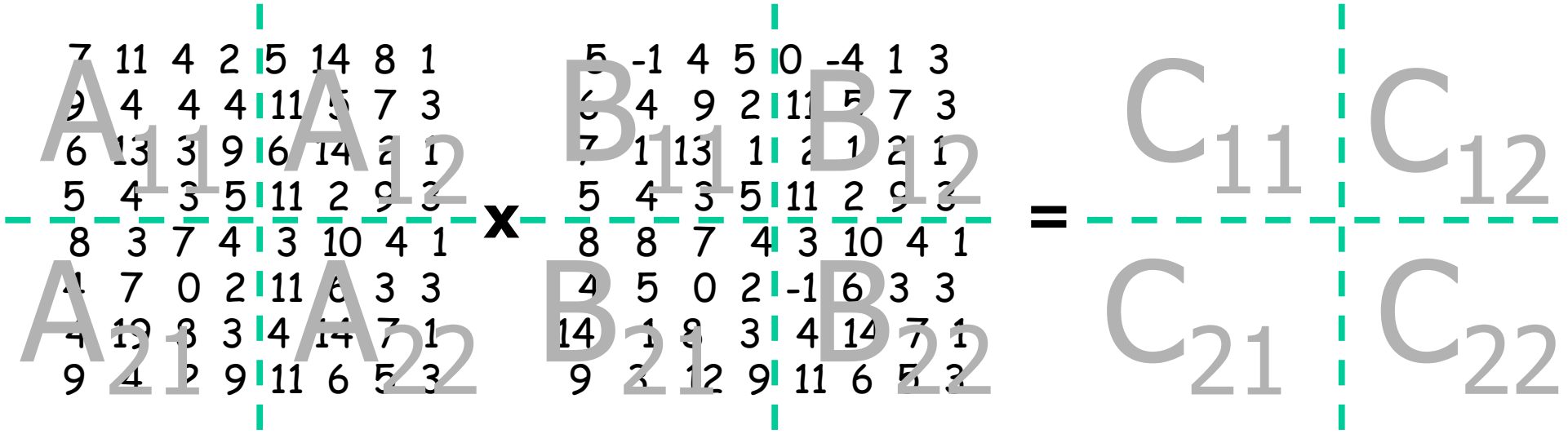
- $T(n) = 3T(n/2) + cn$
- $T(n) = 3(3T(n/2^2) + c(n/2)) + cn$
- $T(n) = 3^2T(n/2^2) + c(n + 3n/2)$
- $T(n) = 3^2(3T(n/2^3) + c(n/2^2)) + c(n + 3n/2)$
- $T(n) = 3^3T(n/2^3) + c(n + 3n/2 + 3^2n/2^2)$
- Generalizando: $T(n) = 3^kT(n/2^k) + cn(1 + 3/2 + \ldots + 3^{k-1}/2^{k-1})$
- $T(n) = 3^kT(n/2^k) + 2cn((3/2)^k-1)$
- Consideremos $n = 2^k$ (ou seja, $k = \lg n$ e $3^k = 3^{\lg n} = n^{\lg 3}$)
- $T(n) = n^{\lg 3}T(1) + 2cn((n^{\lg 3}/n)-1)$
- $T(n) = \Theta(n^{\lg 3}) \approx \Theta(n^{1,585})$

Um exemplo na base 10

- Vamos calcular o produto de dois números na base 10: $I = 5791$ e $J = 234$.
- Podemos vê-los, por exemplo, como $I = I_h.10^2 + I_l$ e $J = J_h.10^2 + J_l$, onde $I_h = 57$, $I_l = 91$, $J_h = 2$ e $J_l = 34$.
- $X = I_h.J_h = 57.2 = 114$
  - É preciso lembrar que as multiplicações são feitas recursivamente
- $Y = I_l.J_l = 91.34 = 3094$
- $Z = (I_h+I_l).(J_h+J_l)-X-Y = (57+91).(2+34)-114-3094 = 2120$
  - Repare que $Z = I_h.J_l + I_l.J_h = 57.34 + 91.2 = 2120$
- $I.J = X.10^4 + Z.10^2 + Y = 1140000 + 212000 + 3094 = 1355094$

Multiplicação de matrizes

- Dadas duas matrizes quadradas A e B de ordem n, o cálculo do produto C = A.B normalmente é feito em tempo $\Theta(n^3)$.
- Também é possível realizar esta multiplicação com um algoritmo de *Divisão-e-Conquista*.
- Exemplo:



$$C_{ij} = A_{i1}.B_{1j} + A_{i2}.B_{2j}$$

Multiplicação por blocos

$C_{11} = A_{11}.B_{11} + A_{12}.B_{21}$ $C_{12} = A_{11}.B_{12} + A_{12}.B_{22}$ 8 multiplicações

$C_{21} = A_{21}.B_{11} + A_{22}.B_{21}$ $C_{22} = A_{21}.B_{12} + A_{22}.B_{22}$ 4 adições

- $T(n) = 8T(n/2) + cn^2$
- $T(n) = 8(8T(n/2^2) + c(n/2)^2) + cn^2 = 8^2T(n/2^2) + cn^2(2 + 1)$
- $T(n) = 8^3T(n/2^3) + cn^2(2^2 + 2^1 + 1)$
- Generalizando: $T(n) = 8^kT(n/2^k) + cn^2(2^{k-1} + \ldots + 2^1 + 1)$
- $T(n) = 2^{3k}T(n/2^k) + cn^2(2^k-1)$
- Para $n = 2^k$: $T(n) = n^3 + cn^2(n-1)$
- $T(n) = \Theta(n^3)$ Equivalente ao algoritmo tradicional (com 3 comandos for)

Algoritmo de Strassen (1969)

- $P = (A_{11}+A_{22}).(B_{11}+B_{22})$
- $Q = (A_{21}+A_{22}).B_{11}$
- $R = A_{11}.(B_{12}-B_{22})$
- $S = A_{22}.(B_{21}-B_{11})$
- $T = (A_{11}+A_{12}).B_{22}$
- $U = (A_{21}-A_{11}).(B_{11}+B_{12})$
- $V = (A_{12}-A_{22}).(B_{21}+B_{22})$

$C_{11} = A_{11}.B_{11} + A_{12}.B_{21}$

$C_{12} = A_{11}.B_{12} + A_{12}.B_{22}$

$C_{21} = A_{21}.B_{11} + A_{22}.B_{21}$

$C_{22} = A_{21}.B_{12} + A_{22}.B_{22}$

$C_{11} = P+S-T+V$

$C_{12} = R+T$

$C_{21} = Q+S$

$C_{22} = P+R-Q+U$

7 multiplicações

18 adições

$$T(n) = 7T(n/2) + cn^2$$

Cálculo da complexidade de tempo

- $T(n) = 7T(n/2) + cn^2$
- $T(n) = 7(7T(n/2^2) + c(n/2)^2) + cn^2 = 7^2T(n/2^2) + cn^2(7/4 + 1)$
- $T(n) = 7^3T(n/2^3) + cn^2(7^2/4^2 + 7/4 + 1)$
- Generalizando: $7^kT(n/2^k) + cn^2((7/4)^{k-1} + \ldots + (7/4)^1 + 1)$
- $T(n) = 7^kT(n/2^k) + cn^2((7/4)^k - 1)/((7/4) - 1)$
- Consideremos $n = 2^k$ (ou seja, $k = \lg n$ e $7^k = 7^{\lg n} = n^{\lg 7}$)
- $T(n) = n^{\lg 7} + cn^2((n^{\lg 7}/n^2) - 1)(4/3)$
- $T(n) = \Theta(n^{\lg 7}) \approx \Theta(n^{2,81})$

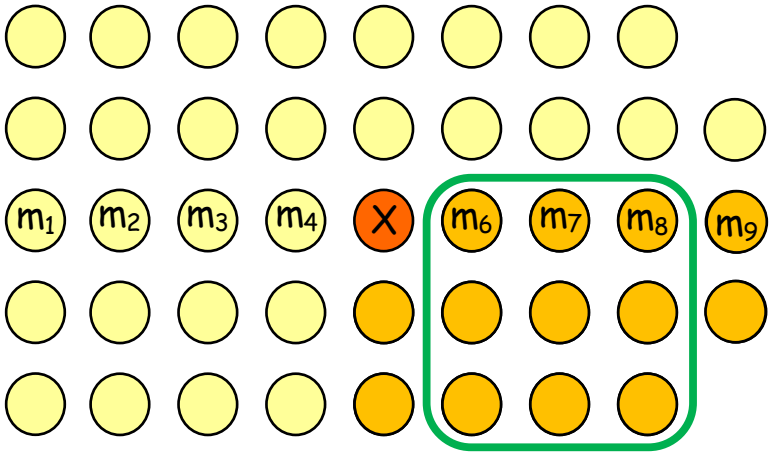
Upper bound é $\Theta(n^{2,371552})$:
algoritmo de Williams et al. (2023)

Seleção do k-ésimo elemento

- Dado um vetor *não ordenado de n elementos distintos*, deseja-se encontrar seu k-ésimo menor elemento, $1 \leq k \leq n$.
- Solução óbvia através da ordenação: $O(n.\log n)$.
- Ideia melhor (Blum, Floyd, Pratt, Rivest e Tarjan, em 1973):
  - Dividir o vetor em partes de tamanho fixo (5 elementos)
  - Encontrar as suas medianas
  - Extrair recursivamente a mediana dessas medianas
  - Particionar o vetor usando essa mediana como pivô
  - Encontrar o k-ésimo elemento na partição adequada

Ideia

- m_i: medianas dos grupos com 5 elementos
- X: mediana dessas medianas (no exemplo abaixo, n = 43)
- Consideremos alguns elementos maiores do que X:



P: partição dentre alguns elementos maiores do que X

$|P| \geq 3.(\lceil \lceil n/5 \rceil /2 \rceil - 2)$

- $|P| \geq 3n/10 - 6$
- Concluímos que há pelo menos $3n/10 - 6$ valores maiores que X
- Portanto, há no máximo $7n/10 + 6$ valores menores que X
- Analogamente, há no máximo $7n/10 + 6$ valores maiores que X

Algoritmo

- Dividir o vetor em grupos de 5 elementos — $\Theta(n)$
- Encontrar as medianas desses grupos — $\lceil n/5 \rceil . \Theta(1)$
- Extrair a mediana X dessas medianas — $T(\lceil n/5 \rceil)$
- Particionar o vetor usando X como pivô — $\Theta(n)$
- m = tamanho da partição com os elementos menores que X
- Busca do k-ésimo: — $T(7n/10 + 6)$
  - Se $k \leq m$, encontrar o k-ésimo elemento da partição dos menores
  - Se $k = m+1$, retornar X
  - Se $k > m+1$, encontrar o $(k-m-1)$-ésimo elemento da outra partição
- Portanto, $T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + \Theta(n)$
- É possível comprovar que $T(n) = \Theta(n)$

Complexidade de tempo

- $T(1) = T(2) = T(3) = T(4) = T(5) = \Theta(1)$
- $T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + a.n,\ n > 5$
- Por indução em n, pode-se provar que $T(n) \leq c.n$
- Base da indução ($n \leq 5$): basta escolher um valor conveniente para c
- Hipótese da indução: supor válido para valores menores que n
- Passo da indução:
  - $T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + a.n$
  - $T(n) \leq c.\lceil n/5 \rceil + c.(7n/10 + 6) + a.n$
  - $T(n) \leq cn/5 + c + 7cn/10 + 6c + a.n$
  - $T(n) \leq 9cn/10 + 7c + a.n$
  - $T(n) \leq cn + (-cn/10 + 7c + a.n)$
  - $T(n) \leq c.n \Leftrightarrow -cn/10 + 7c + a.n \leq 0 \Leftrightarrow c \geq 10a(n/(n-70))$
  - Basta admitir $n > 70$ e escolher c convenientemente
- Portanto, $T(n) = \Theta(n)$: o algoritmo gasta tempo linear

Na verdade, poderia ter sido escolhido qualquer valor maior que 5 (teria tempo linear, mas com desempenho pior)

Método Guloso (*Greedy Algorithms*)

- Características desta técnica:
  - Costuma ser aplicada a problemas de otimização: encontrar soluções viáveis que sejam ótimas de acordo com uma função objetivo.
  - Baseia-se na extensão de soluções parciais: escolhe-se a que propicia o maior ganho imediato (daí o nome de "guloso"; mais propriamente, "ganancioso").
  - Não proporciona necessariamente uma solução ótima: é preciso uma demonstração para cada problema. Por outro lado, basta um contraexemplo para verificar que falha.
  - Quando funciona, origina algoritmos simples e eficientes.
- Exemplos já vistos: Dijkstra, Kruskal e Prim.

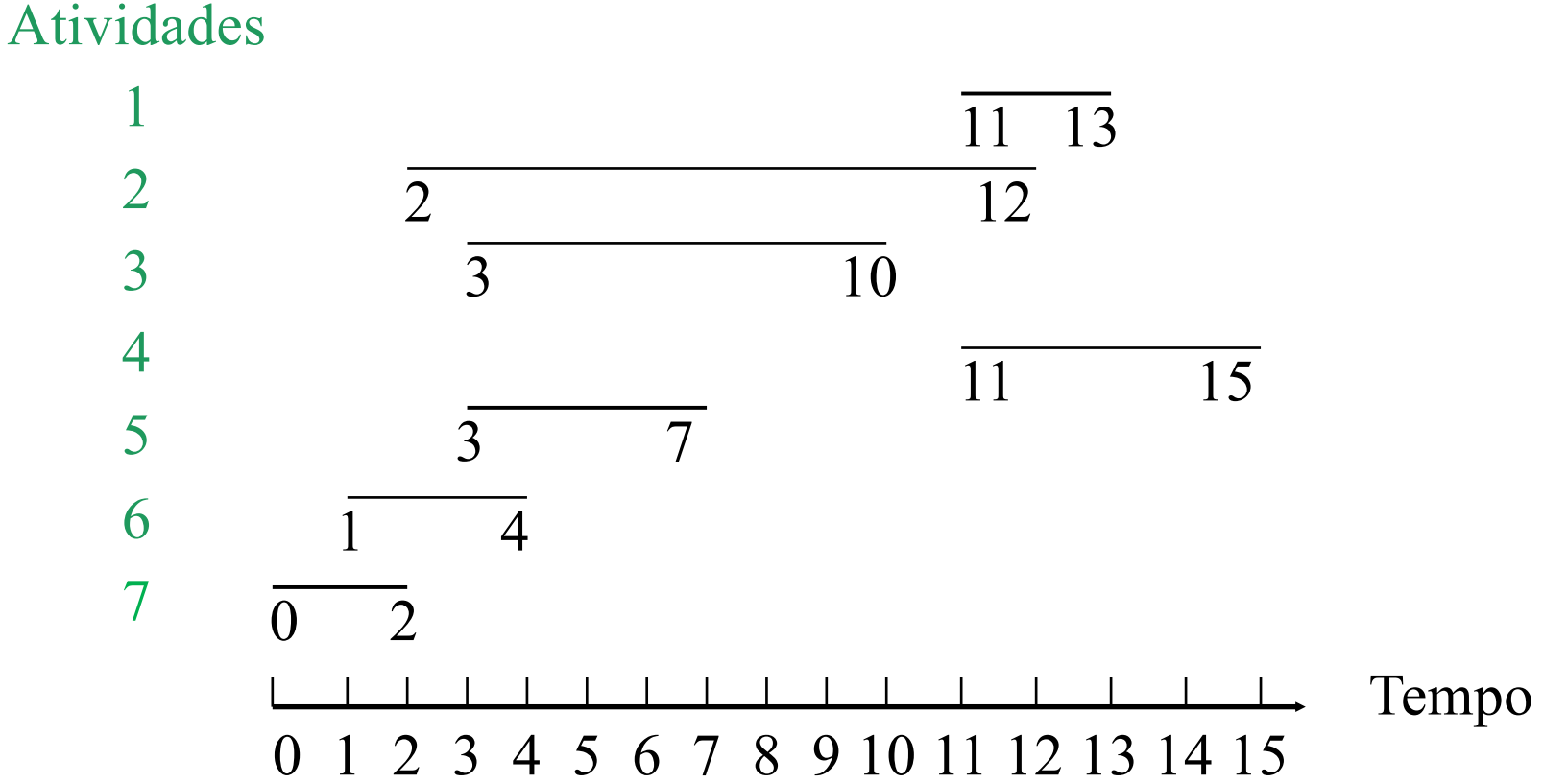
Técnica geral de demonstração

- Há uma técnica geral de demonstração do *Método Guloso* :
  - Expressar o problema original como um problema no qual podemos fazer uma escolha gulosa, restando um subproblema análogo para ser resolvido.
    - Por isso, o *Método Guloso* pode ser entendido como um caso particular da *Divisão-e-Conquista*.
  - Provar que sempre existe uma solução ótima para o problema original que usa a escolha gulosa (ou seja, ela é uma escolha segura). Chamaremos de Parte a.
  - Demonstrar que, se combinarmos a escolha gulosa com a solução ótima do subproblema restante, teremos uma solução ótima do problema original. Chamaremos de Parte b.
  - Nessa situação, a iteração da escolha gulosa gerará garantidamente uma solução ótima para o problema original.

Seleção de atividades

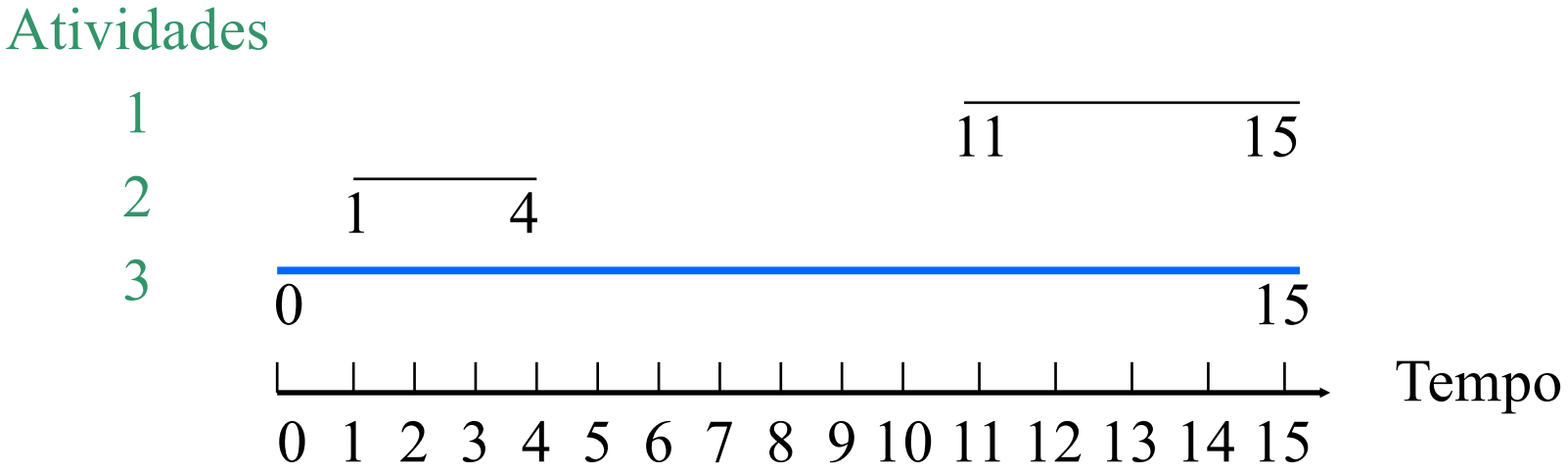
- Seja um conjunto S com n atividades, onde s_i e f_i são respectivamente os tempos de início e de término da atividade i, $1 \leq i \leq n$. Evidentemente, $f_i \geq s_i$.
- Duas atividades i e j são compatíveis entre si quando seus tempos de início e de término não se sobrepõem, ou seja, $s_i \geq f_j$ ou $s_j \geq f_i$.
- Objetivo: encontrar um subconjunto máximo de S com atividades compatíveis.
- Qual seria a escolha gulosa adequada?

Exemplo


Atividades
1
2
3
4
5
6
7
11 13
2 12
3 10
11 15
3 7
1 4
0 2
Tempo
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

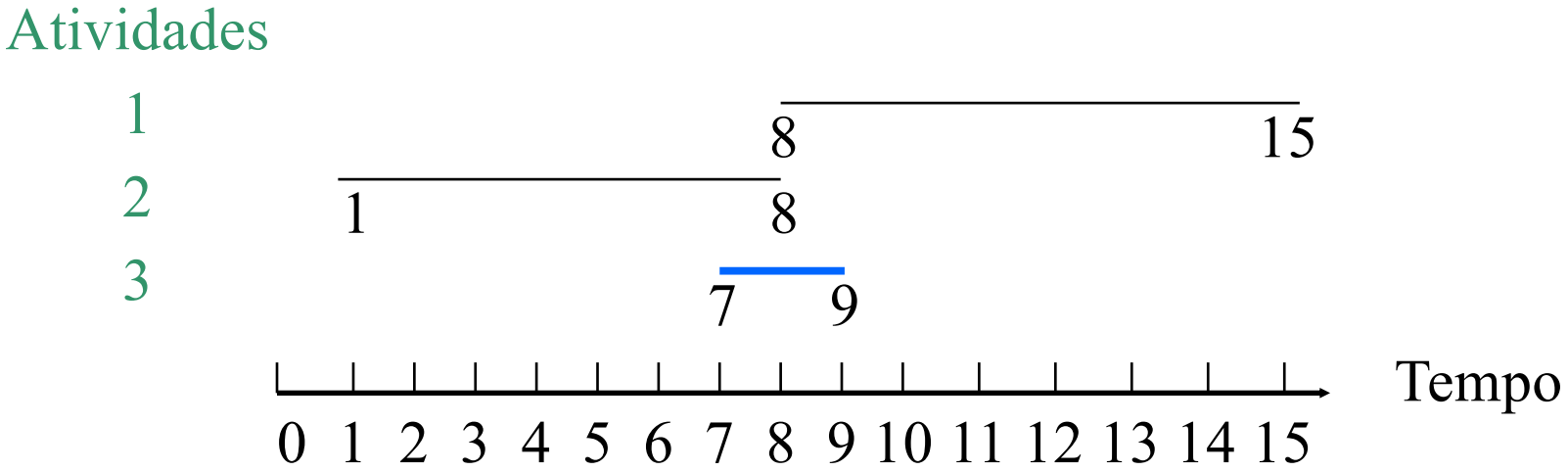
Pelo tempo de início?

- Um contraexemplo:

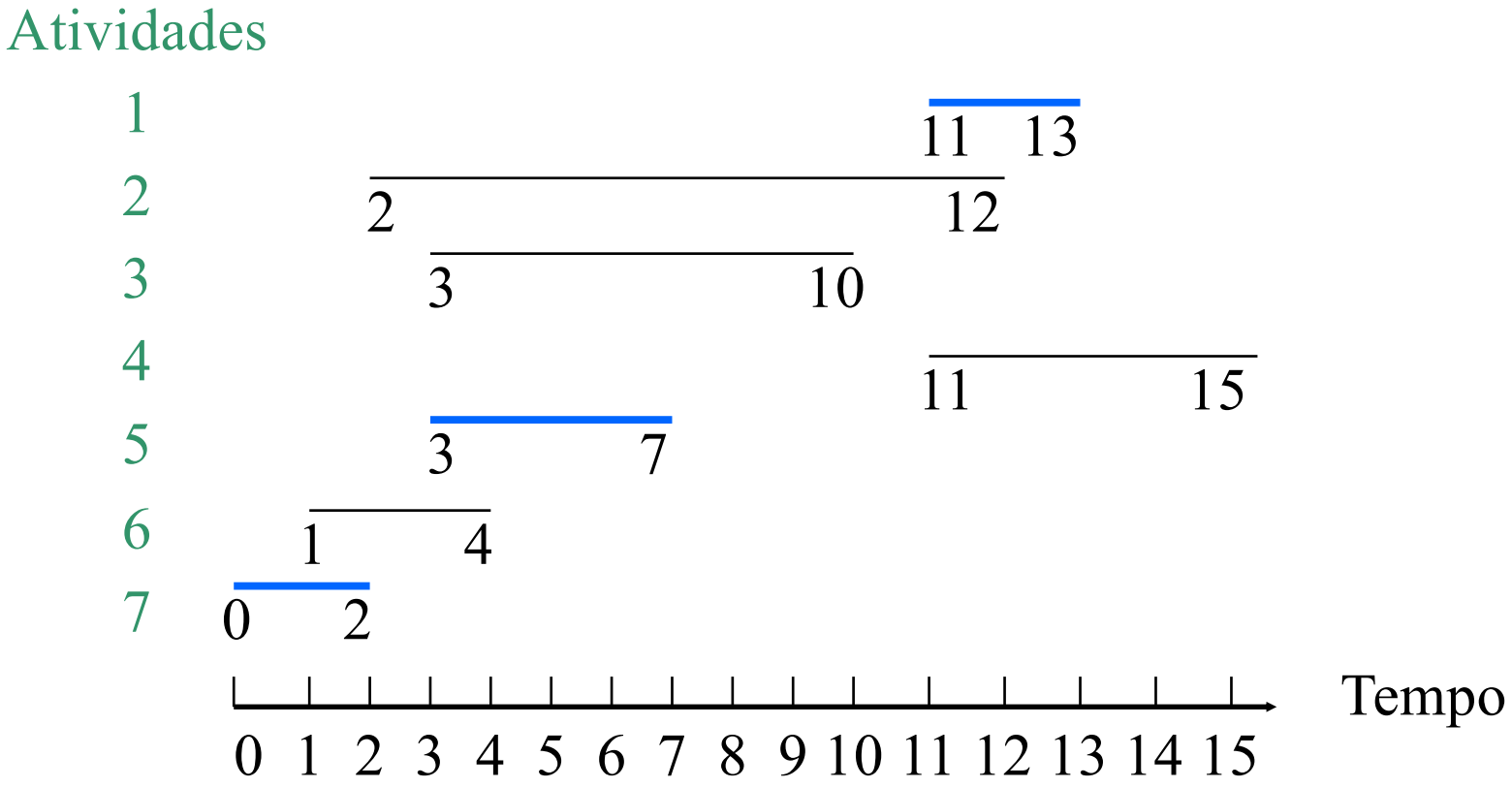


Pela duração mínima?

- Um contraexemplo:



Pelo tempo de término?



Algoritmo

```
ActivitySelection() {
     S = Sort(S, f);  // ordenação de S pelo valor de f
     A = {1}; // índice da atividade que termina primeiro
     j = 1;   // índice da última atividade selecionada
     for (i=2; i<=n; i++)
          if (s_i >= f_j) {
               A = A ∪ {i};
               j = i;
          }
     return A;  // índices das atividades
}
```

Complexidade de tempo: $\Theta(n.\log n)$

Demonstração da Parte a

a) A escolha gulosa está em alguma solução ótima:

- Sejam A essa solução ótima e x a atividade da escolha gulosa. Se $x \in A$, a demonstração está terminada. Se $x \notin A$, mostraremos que $A' = A+\{x\}-\{y\}$ é outra solução ótima que contém a atividade x.
- Seja y a atividade com menor tempo de término em A.
- Como as atividades foram ordenadas pelo tempo de término, então $f_x \leq f_y$. Se $f_x \leq s_y$, poderíamos acrescentar x a A, ou seja, A não seria ótima. Então, $f_x > s_y$. Logo, as atividades x e y se sobrepõem.
- Como $f_x \leq f_y$, podemos trocar a atividade y por x, obtendo uma nova solução $A' = A+\{x\}-\{y\}$, onde $|A'|=|A|$.

Demonstração da Parte b

b) A escolha gulosa e a solução ótima do subproblema restante formam uma solução ótima do problema original:

- Sejam x a atividade da escolha gulosa e S' o subconjunto de atividades que não sobrepõem x, ou seja, $S' = \{ i \mid s_i \geq f_x \}$.
- Seja B uma solução ótima para S'.
  Então, A' = {x}+B é uma solução para o problema original.
- Podemos demonstrar que A' é ótima:
  - Suponhamos que A' não seja ótima.
    Seja A uma solução ótima que contém x (vide Parte a).
    Portanto, |A|>|A'| e |A-{x}| > |A'-{x}| = |B|.
  - No entanto, A-{x} é uma solução para S' com mais elementos que B, ou seja, B não é uma solução ótima.
  - Contradição! Portanto, A' deve ser ótima.

Intercalação ótima de arquivos

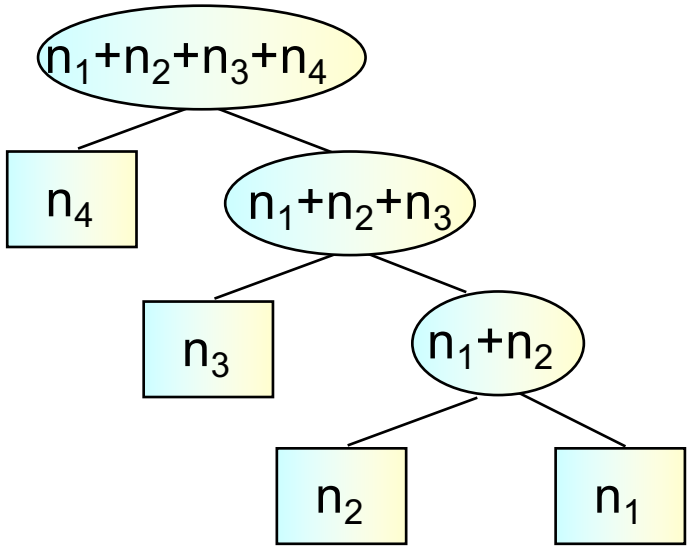
- Este problema consiste em encontrar a melhor sequência de intercalações (*merges*) entre k arquivos ordenados, que serão unidos dois a dois.
- Sabemos que a intercalação de dois arquivos, com m e n registros respectivamente, exige Θ(m+n) operações.
- É fácil constatar que diferentes ordens de intercalação proporcionam diferentes tempos de processamento.
  - Exemplo:

A = 30
B = 20
C = 10
} registros

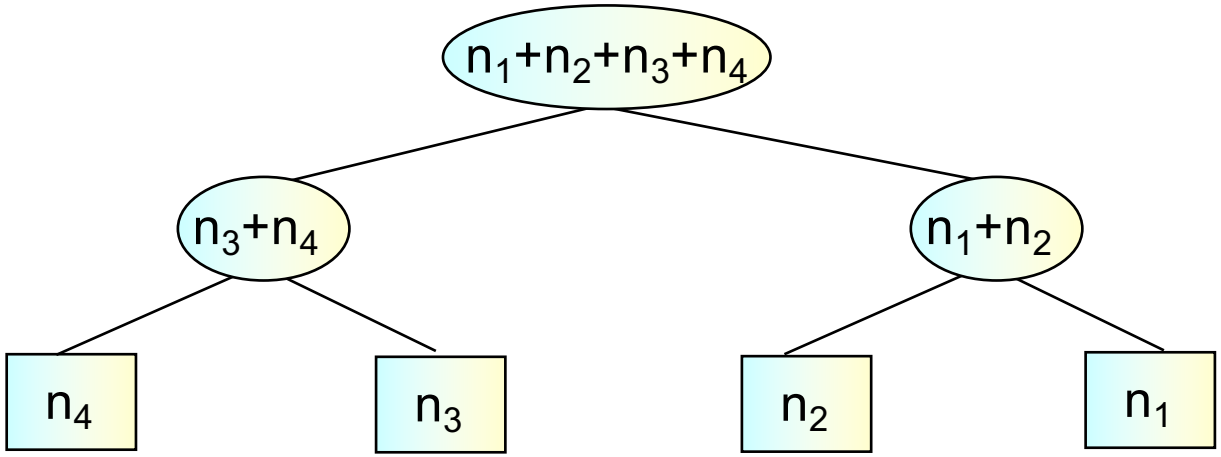
A + B = 50 operações
(A + B) + C = 60 operações
} Total: 110

B + C = 30 operações
(B + C) + A = 60 operações
} Total: 90

Outro exemplo



$$n_1 + n_2$$
$$n_1 + n_2 + n_3$$
$$n_1 + n_2 + n_3 + n_4$$
$$3n_1 + 3n_2 + 2n_3 + n_4$$

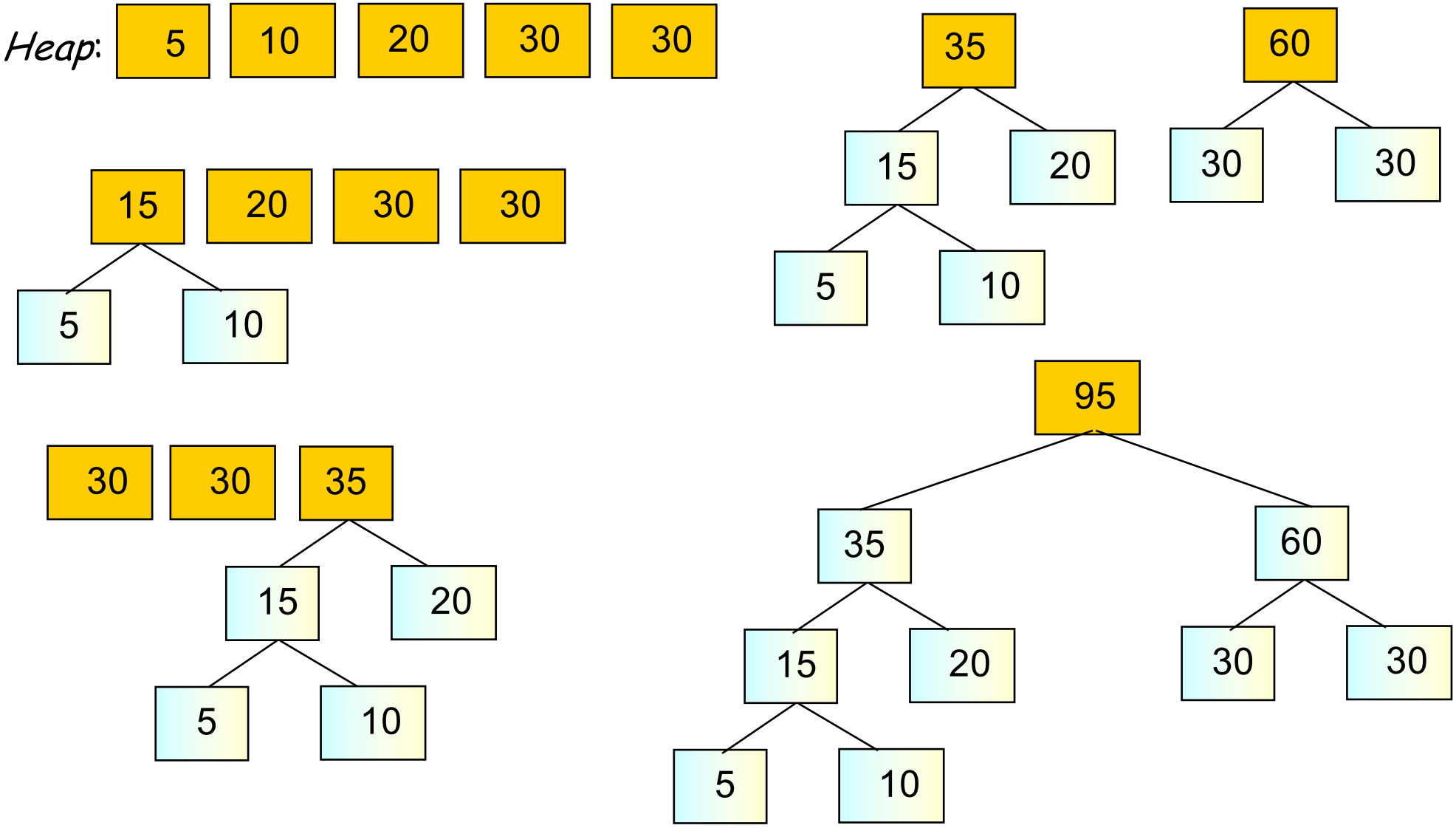


$$n_1 + n_2$$
$$n_3 + n_4$$
$$(n_1 + n_2) + (n_3 + n_4)$$
$$2n_1 + 2n_2 + 2n_3 + 2n_4$$

Cálculo do número de operações

- Arquivos: são as folhas da árvore binária.
- Número de operações: são os nós internos.
- d_i: distância da raiz à folha que representa o i-ésimo arquivo, $1 \leq i \leq k$.
- n_i: número de registros do i-ésimo arquivo, $1 \leq i \leq k$.
- É possível verificar que, na intercalação de todos os k arquivos, o número total de operações será $\Theta(\Sigma n_i.d_i)$. Este é o custo total da árvore de intercalação.
- Ideia gulosa: intercalar antes os menores arquivos. Desse modo, ficarão mais abaixo na árvore, minimizando o número de total de operações.
- Para se encontrar os dois menores arquivos *correntes*, convém utilizar uma estrutura de *heap* de mínimo.

Exemplo com k = 5



Algoritmo

v: vetor com k nós que armazenam no campo `valor` o tamanho de cada arquivo

```
node OptimalMerge(v) {
     h = new HeapMin();
     h.Build(v);
     for (i=1; i<k; i++) {
          no = new TreeNode();
          no.esq = h.ExtractMin();
          no.dir = h.ExtractMin();
          no.valor = no.esq->valor + no.dir->valor;
          h.Insert(no);
     }
     return h.ExtractMin();
}
```

Complexidade de tempo:
$\Theta(k.\log k)$

Para encontrar a melhor sequência de intercalações, não para realizá-las

Demonstração da Parte a

a) A escolha gulosa está em alguma solução ótima:

- Sejam x e y os menores arquivos, onde $n_x \leq n_y$. Seja T_1 a árvore de intercalação ótima, na qual a e b são os arquivos com profundidade máxima. Portanto, sem perda de generalidade, $n_x \leq n_a$ e $n_y \leq n_b$.
- Seja T_2 outra árvore de intercalação, igual a T_1, mas com os nós x e a trocados. Considerando d_i como a distância de n_i à raiz na árvore T_1, a diferença entre os custos de T_1 e de T_2 é:
  $C(T_1) - C(T_2) = n_x.d_x + n_a.d_a - n_x.d_a - n_a.d_x = (n_a - n_x).(d_a - d_x) \geq 0.$
- Como $C(T_1)$ é ótimo, $C(T_2)$ não pode ser menor. Logo, $C(T_1) = C(T_2)$.
- Seja T_3 outra árvore igual a T_2, mas com os nós y e b trocados. Analogamente, $C(T_3) = C(T_2) = C(T_1)$.
- Portanto, a árvore T_3, gerada pela escolha gulosa, é ótima.

Demonstração da Parte b

b) A escolha gulosa e a solução ótima do subproblema restante formam uma solução ótima do problema original:

- Sejam x e y os menores arquivos, e z o arquivo resultante da intercalação entre eles, ou seja, $n_z = n_x + n_y$.
- Seja T_2 uma árvore de intercalação ótima do subproblema restante (considerando o arquivo z ao invés dos arquivos x e y).
- Seja T_1 a árvore obtida a partir de T_2 substituindo-se a folha n_z por um nó interno cujos filhos são n_x e n_y.
- $C(T_1) - C(T_2) = (n_x + n_y)(d_z + 1) - (n_x + n_y)d_z = n_x + n_y$
- $C(T_2) = C(T_1) - n_x - n_y$
- T_1 é uma árvore de intercalação ótima do problema original:
  - Suponha que T_1 não seja ótima. De acordo com a Parte a, seja T_3 a árvore ótima onde n_x e n_y são irmãos. Portanto, $C(T_3) < C(T_1)$.
  - Seja T_4 a árvore T_3 onde o pai de n_x e n_y é trocado pela folha n_z.
  - $C(T_4) = C(T_3) - n_x - n_y < C(T_1) - n_x - n_y = C(T_2)$
  - T_2 não seria uma árvore ótima do subproblema restante: contradição!

Codificação de Huffman (1952)

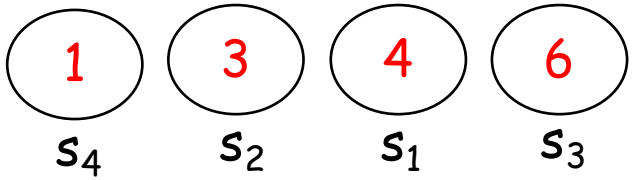
- É uma conhecida técnica de compressão de dados, e a sua implementação baseia-se na mesma ideia anterior.
- O objetivo é atribuir códigos curtos aos caracteres que ocorrem com maior frequência. Haverá compressão desde que os caracteres tenham frequências distintas.
- Cada texto terá uma codificação própria, que não é necessariamente adequada a outros.
- No entanto, como as codificações dos caracteres terão comprimento variável, se uma delas for prefixo de outra, pode haver ambiguidade. Isto deve ser evitado...

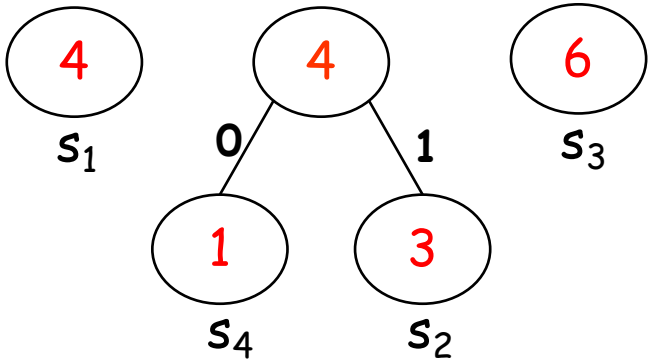
Ideia do algoritmo

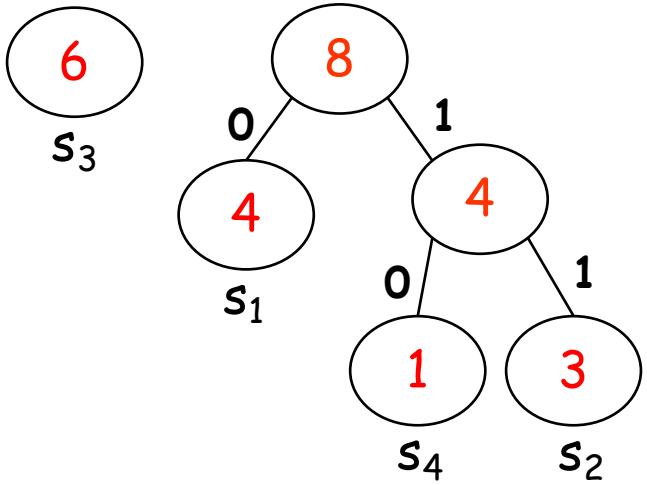
- Calcula-se a frequência de cada caractere.
- Cria-se um nó para cada caractere com frequência não nula.
- Agrupam-se dois a dois os nós com menor frequência, como filhos de um novo nó, cujo valor será a soma das frequências dos filhos.
- Rotulam-se as arestas dessa árvore:
  - 0 para o filho da esquerda;
  - 1 para o filho da direita.
- A codificação de cada caractere será formada pelos rótulos encontrados no percurso da raiz até o seu nó.

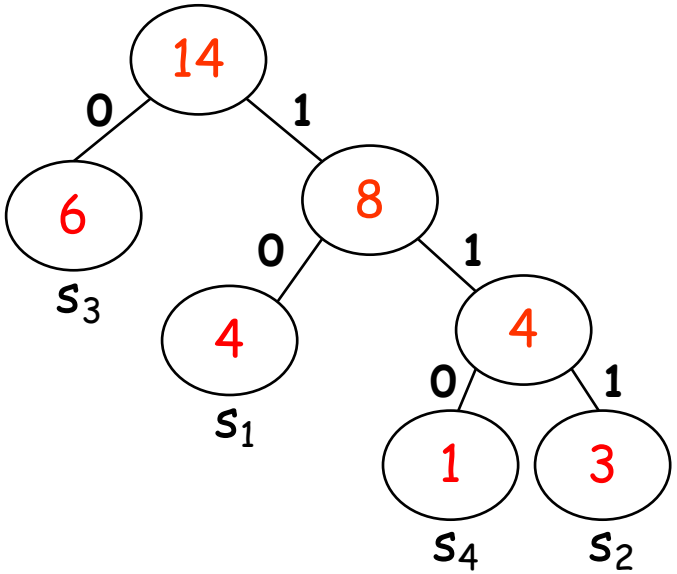
Exemplo de construção

- Sequência: $s_2 s_3 s_1 s_3 s_1 s_4 s_2 s_1 s_3 s_3 s_3 s_3 s_1 s_2$





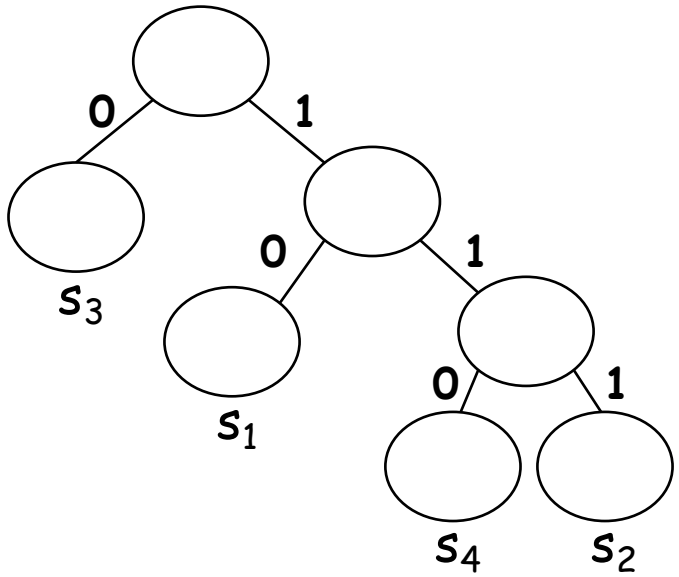




Exemplo de codificação

- Sequência de caracteres: $s_2s_3s_1s_3s_1s_4s_2s_1s_3s_3s_3s_3s_1s_2$

Símbolo	Frequência	Código
s_1	4	10
s_2	3	111
s_3	6	0
s_4	1	110



Sequência codificada: 1110100101101111000010111

Supondo 2 bits por caractere, a taxa de compressão é 2/28 ≈ 7,1 %

Se os caractares tivessem originariamente 8 bits, taxa seria maior

Moedas de troco

- Este simples problema consiste na devolução de um determinado troco utilizando-se uma quantidade mínima de moedas.
- Escolha gulosa: devolver o máximo de moedas de mais alto valor.
- Dado um conjunto de moedas, o método guloso encontra a solução ótima, isto é, o troco com o menor número de moedas?

Algoritmo guloso

```
Troco MakeChange(moedas, quantia) {
    troco = 0;
    while (há moedas diferentes && troco < quantia) {
          Escolher o maior tipo de moeda disponível;
          while (troco < quantia)
                Acrescentar essa moeda ao troco;
          if (troco == quantia) return troco;
          Retirar do troco a última moeda acrescentada;
    }
    return "Não foi possível calcular o troco";
}
```

Funciona?

Encontra a solução ótima?

Um contraexemplo

- Conjunto de moedas: 1, 10, 25 e 50 centavos.
- Troco a ser dado: 30 centavos.
- Escolha gulosa: 25, 1, 1, 1, 1, 1.
- Solução ótima: 10, 10, 10.
- A solução ótima pode ser obtida através de um algoritmo de *Programação Dinâmica*.

Programação Dinâmica

- *Dynamic Programming* surgiu com Bellman nos anos 50, e aplica-se a problemas de otimização.
- Semelhante à *Divisão-e-Conquista*, também consiste na resolução de subproblemas análogos, mas *não disjuntos* (há sobreposição de subproblemas).
- Por esse motivo, todos os subproblemas são resolvidos em ordem incremental de tamanho.
- Esses resultados são armazenadas em uma única tabela.
- Ganha-se tempo:
  - resolvendo-se uma única vez cada subproblema;
  - à custa de um maior espaço de armazenamento.

Exemplo

Cálculo da sequência de Fibonacci com *Programação Dinâmica:*

```
int Fibonacci(n) {
      F[0] = 1;
      F[1] = 1;
      for (i=2; i<=n; i++) {
          F[i] = F[i-1] + F[i-2];
      return F[n];
}
```

Tempo: $\Theta(n)$

Espaço: $\Theta(n)$

A Programação Dinâmica também é chamada de "recursão com tabela"

Moedas de troco

- Poderíamos elaborar também uma resolução *Divisão-e-Conquista* para o problema das moedas de troco:

```
int DCMakeChange(moedas, troco) {
    if (troco == 0) return 0;
    q = troco;  // solução óbvia com somente moedas de 1 centavo
    for (i = 0; i < length(moedas); i++) {
         if (moedas[i] > troco) continue; // essa moeda não serve
         q = min {q, 1 + DCMakeChange(moedas, troco - moedas[i])};
         }
    }
    return q;     // quantidade ótima de moedas para troco
}
```

- Este algoritmo funciona. No entanto, será que é eficiente?
- É fácil constatar que esta tática leva a resolver novamente subproblemas já resolvidos...
- Por isso, é ineficiente: é o que chamamos de sobreposição de subproblemas.

Moedas de troco

- Quando ocorre sobreposição de subproblemas, é mais eficiente resolver todos os subproblemas em ordem incremental de tamanho, armazenando suas subsoluções em um vetor para eventuais consultas posteriores.
- Consideremos um exemplo: troco ótimo de 15 centavos utilizando moedas de 1, 5 e 10 centavos.
- Na simulação abaixo, supomos que o vetor de moedas esteja em ordem crescente, embora isso não seja necessário (nesse caso, seria encontrada outra solução também ótima).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`quant`	0	1	2	3	4	1	2	3	4	5	1	2	3	4	5	2
`ultima`	0	1	1	1	1	5	1	1	1	1	10	1	1	1	1	5

Algoritmo

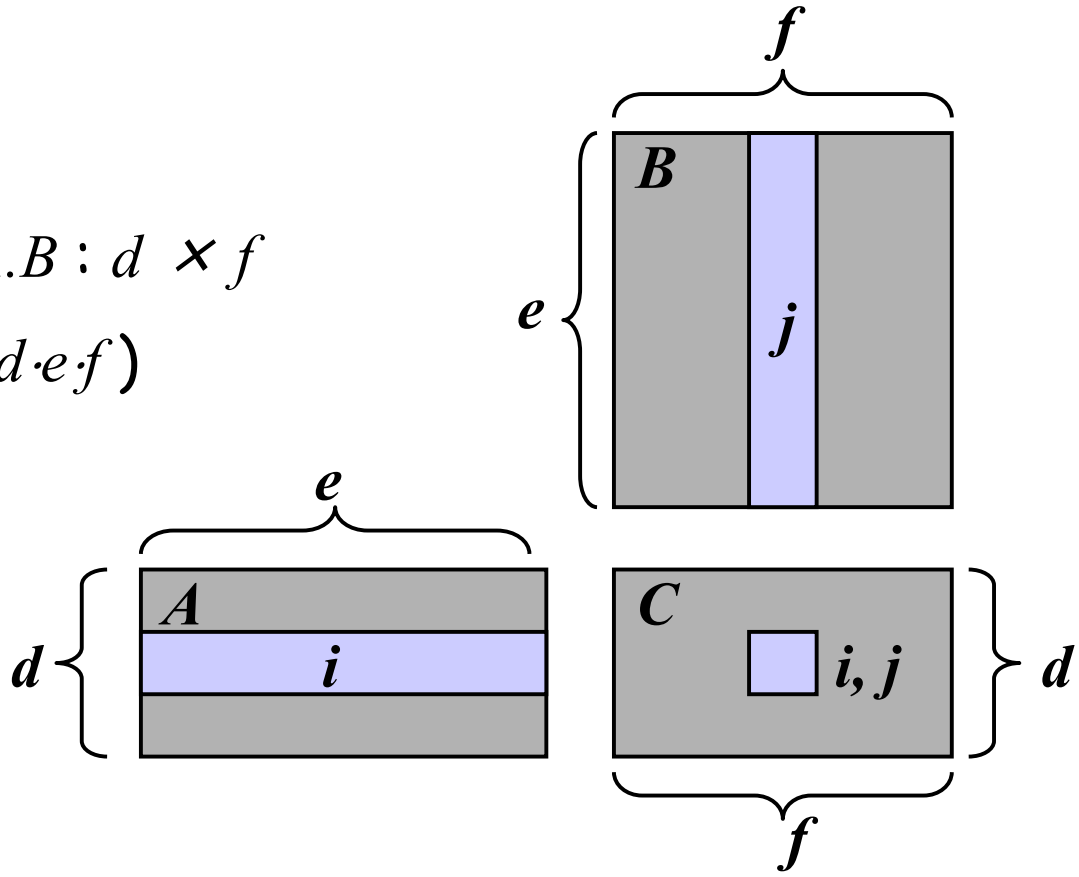
```
int DPMakeChange(moedas, troco) {
    // moedas: vetor de moedas disponíveis (menor é de 1 centavo)
    quant[0] = 0;  // solução ótima para troco de valor 0
    ultima[0] = 0; // última moeda dessa solução
    for (cents = 1; cents <= troco; cents++) {
        quantProv = cents; // solução provisória: todas de 1 centavo
        ultProv = 1;       // última moeda dessa solução
        for (j = 0; j < length(moedas); j++) {
            if (moedas[j] > cents) continue; // essa moeda não serve
            if (quant[cents - moedas[j]] + 1 < quantProv) {
                quantProv = quant[cents - moedas[j]] + 1;
                ultProv = moedas[j];
            }
        }
        quant[cents] = quantProv; // solução para troco == cents
        ultima[cents] = ultProv;  // última moeda dessa solução
    }
    return quant[troco] // quantidade ótima de moedas para troco
}
```

Tempo: $\Theta(n.k)$, onde n é o valor do troco e k o número de moedas

Encadeamento do produto de matrizes

- Multiplicação de duas matrizes:
  - Matriz A: $d \times e$
  - Matriz B: $e \times f$
  - Matriz produto $C = A.B$: $d \times f$
  - Tempo do cálculo: $\Theta(d \cdot e \cdot f)$

$$C[i,j] = \sum_{k=0}^{e-1} A[i,k].B[k,j]$$



Encadeamento do produto de matrizes

- Descrição do problema:
  - O objetivo é calcular $A = A_0.A_1. \ldots .A_{n-1}$
  - A dimensão da matriz A_i é $d_i \times d_{i+1}$
  - Qual a melhor sequência de cálculo?
- Um exemplo:
  - Matriz B é 3×100
  - Matriz C é 100×5
  - Matriz D é 5×5
  - Total de operações de (B.C).D é proporcional a 1500 + 75 = 1575
  - Total de operações de B.(C.D) é proporcional a 2500 + 1500 = 4000

Enumeração das soluções

- Primeira tentativa: "força bruta"
- Possível algoritmo:
  - Tente todas as possíveis maneiras de "parentisar" o produto $A = A_0.A_1. \ldots .A_{n-1}$
  - Calcule o número de operações em cada uma delas
  - Escolha a melhor
- Tempo necessário: exponencial!

Solução gulosa

- Ideia: selecionar as duas matrizes com menores dimensões.
- Um contraexemplo:
  - A: 101×11; B: 11×9; C: 9×100; D: 100×99
  - A solução gulosa é $A.((B.C).D)$, que gasta tempo proporcional a $9900 + 108900 + 109989 = 228789$
  - A solução ótima é $(A.B).(C.D)$, que gasta tempo proporcional a $9999 + 89100 + 89991 = 189090$
- *Método Guloso* não garante solução ótima para este problema, mesmo utilizando outro critério (por exemplo, selecionar as duas matrizes cujo produto tenha o menor número de operações).

Divisão em subproblemas

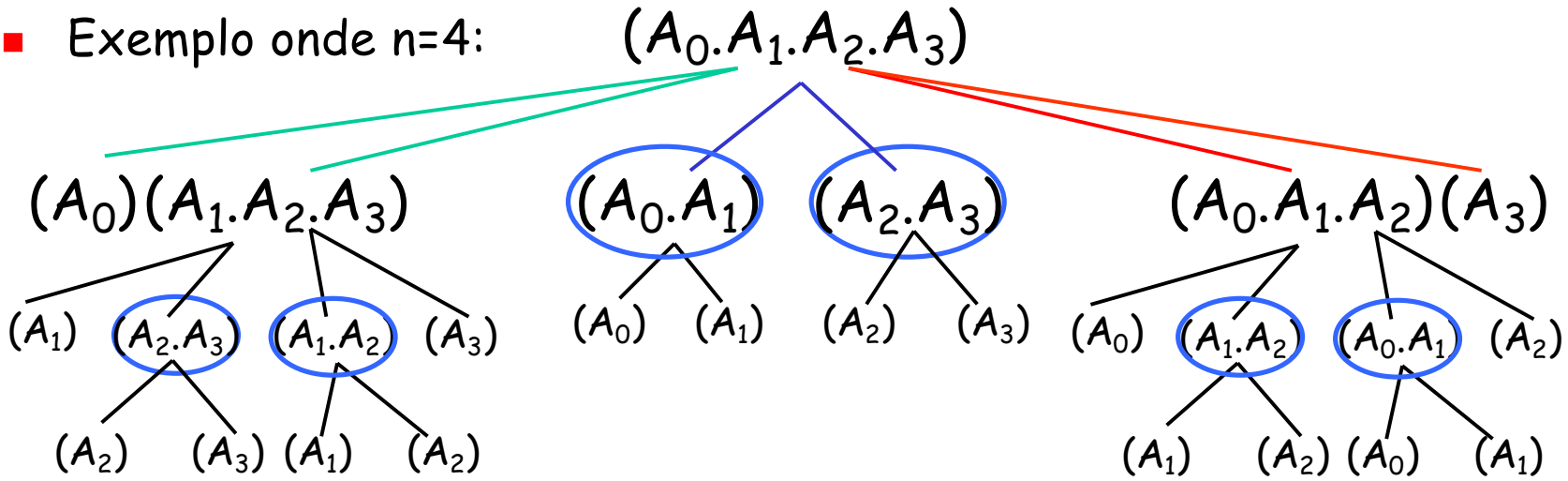
- Definição dos subproblemas:
  - Um subproblema é encontrar a melhor "parentisação" de $A_i.A_{i+1}. \ldots .A_j$.
  - Seja $N_{i,j}$ o número ótimo de operações realizadas neste subproblema.
  - A solução ótima do problema completo será $N_{0,n-1}$.
- A solução ótima do problema original pode ser definida em termos de subsoluções ótimas:
  - Deverá haver uma última multiplicação nessa solução ótima.
  - Suponhamos que seja após o índice k: $(A_0. \ldots .A_k).(A_{k+1}. \ldots .A_{n-1})$.
  - Nesse caso, a solução ótima $N_{0,n-1}$ será a soma de duas subsoluções ótimas $N_{0,k}$ e $N_{k+1,n-1}$, mais o tempo gasto na última multiplicação.
  - Isso seria: $N_{0,k} + N_{k+1,n-1} + d_0.d_{k+1}.d_n$.
  - No entanto, será preciso testar todos os possíveis valores de k...

Equação de cálculo

- Lembrete: cada matriz A_i tem dimensão $d_i \times d_{i+1}$.
- No cálculo de $N_{i,j}$, será preciso considerar todas as possíveis posições para a última multiplicação:

$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

- Repare que esses subproblemas não são disjuntos, mas sobrepostos.
  - Exemplo onde n=4:



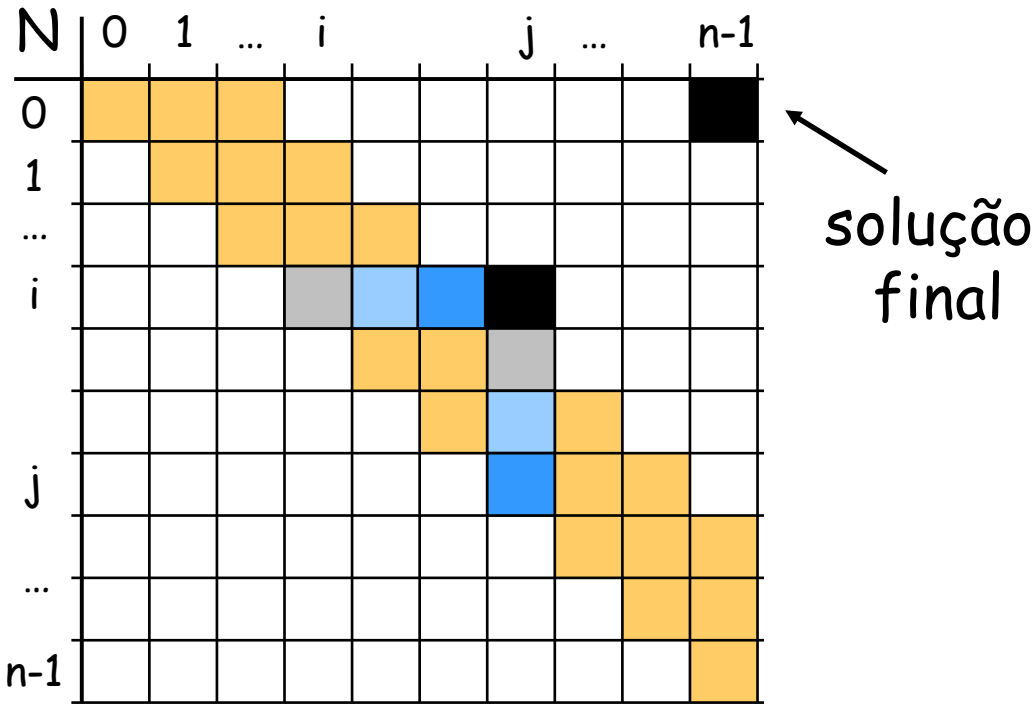
Ideia da *Programação Dinâmica*

- Como há sobreposição de subproblemas, não convém resolvê-los recursivamente (não seria eficiente).
- O melhor método será uma construção *bottom-up* das subsoluções ótimas.
- Começamos pelos subproblemas menores: $N_{i,i}$, $0 \leq i < n$, que tem "tamanho" 1.
- Em seguida, resolveremos os problemas de "tamanho" 2; depois, os de "tamanho" 3; e assim por diante.

Visualização do algoritmo

- $N_{i,j}$ consulta valores prévios na linha i e na coluna j.
- *A construção bottom-up* preenche a matriz N pelas diagonais.
- O cálculo de cada posição $N_{i,j}$ gasta tempo O(n).
- Tempo total é $\Theta(n^3)$.

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$



Exemplo onde n=5

- $A = A_0.A_1.A_2.A_3.A_4$, onde as dimensões são, respectivamente: 5x4, 4x1, 1x3, 3x7 e 7x2.
- Portanto: $d_0=5$, $d_1=4$, $d_2=1$, $d_3=3$, $d_4=7$ e $d_5=2$.

N	0	1	2	3	4
0	0	20	35	76	65
1		0	12	49	43
2			0	21	35
3				0	42
4					0

T	0	1	2	3	4
0		0	1	1	1
1			1	1	1
2				2	3
3					3
4					

Sequência ótima: $((A_0 . A_1).((A_2 . A_3) . A_4))$

Algoritmo

```
MatrixChain() {
    for (i=0; i<n; i++)
       N[i,i] = 0;          // subproblemas triviais
    for (b=1; b<n; b++)     // tamanhos dos subproblemas
       for (i=0; i<n-b; i++) {
          j = i+b;          // novo intervalo para subproblema
          N[i,j] = +∞;      // valor provisório para a solução
          T[i,j] = i;       // valor provisório para o índice
          for (k=i; k<j; k++) {  // equação de cálculo
             x = N[i,k] + N[k+1,j] + d[i].d[k+1].d[j+1];
             if (N[i,j] > x) {
                   N[i,j] = x;
                   T[i,j] = k; // (Ai...Ak).(Ak+1...Aj)
             }
          }
       }
    return N[0,n-1];
}
```

Tempo: $\Theta(n^3)$

Maior subsequência comum (LCS)

- Dadas duas *strings* $X = \langle x_1,\ldots,x_m\rangle$ e $Y = \langle y_1,\ldots,y_n\rangle$, deseja-se encontrar a maior subsequência compartilhada por ambas.
- Não é necessário que os caracteres dessa subsequência estejam em posições contíguas nas palavras originais, mas que seja mantida a ordem entre eles.
- Exemplo: as LCS de X=ababc e Y=abddcb são abc (ababc e abddcb) e abb (ababc e abddcb), ambas de tamanho 3.
- A solução "força bruta" gastaria tempo $\Theta(n.2^m)$: 2^m subsequências de X e n testes com Y.

Ideia

- Seja $X_i = \langle x_1,\ldots,x_i \rangle$ o prefixo formado pelos primeiros i caracteres de X. Idem para $Y_i = \langle y_1,\ldots,y_i \rangle$.
- Seja $Z = \langle z_1,\ldots,z_k \rangle$ uma LCS de X e de Y com tamanho k.
- Se $x_m=y_n$, então $z_k=x_m=y_n$ e Z_{k-1} é uma LCS de X_{m-1} e de Y_{n-1}.
- Se $x_m \neq y_n$, então:
  - ou $z_k \neq x_m$ e Z é uma LCS de X_{m-1} e de Y;
  - ou $z_k \neq y_n$ e Z é uma LCS de X e de Y_{n-1}.
- c[i,j], que é o comprimento da LCS de X_i e de Y_j, será:
  - 0, se i=0 ou j=0;
  - c[i-1,j-1] + 1, se i>0, j>0 e $x_i=y_j$;
  - max {c[i-1,j], c[i,j-1]}, se i>0, j>0 e $x_i \neq y_j$.
- c[m,n] será o comprimento da LCS de X e de Y.

Exemplo

X = ⟨ GGATCGA ⟩ e Y = ⟨ GAATTCAGTTA ⟩

		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1	2	2	2	2
A	0	1	2	2	2	2	2	2	2	2	2	3
T	0	1	2	2	3	3	3	3	3	3	3	3
C	0	1	2	2	3	3	4	4	4	4	4	4
G	0	1	2	2	3	3	4	4	5	5	5	5
A	0	1	2	3	3	3	4	5	5	5	5	6

|LCS| = 6

LCS: ⟨ GATCGA ⟩

Algoritmo

- Inicialmente, calculamos duas matrizes: `c` e `trace`

```
LCS() {
    for (i=0; i<=m; i++) c[i,0] = 0;
    for (j=0; j<=n; j++) c[0,j] = 0;
    for (i=1; i<=m; i++)
        for (j=1; j<=n; j++)
            if (x[i] == y[j]) {
                    c[i,j] = c[i-1,j-1] + 1;
                    trace[i,j] = "↘"; }
            else if (c[i,j-1] > c[i-1,j]) {
                        c[i,j] = c[i,j-1];
                        trace[i,j] = "→"; }
                 else { c[i,j] = c[i-1,j];
                        trace[i,j] = "↓"; }
    return c, trace;
}
```

Tempo: $\Theta(n.m)$

Algoritmo

- Posteriormente, a impressão da LCS pode ser realizada através de um percurso na matriz `trace`

```
PrintLCS(i, j) {
    if (i==0 || j==0) return;
    if (trace[i,j] == "↘") {
          PrintLCS(i-1, j-1);
          print x[i];
          }
    else if (trace[i,j] == "↓") PrintLCS(i-1, j);
          else PrintLCS(i, j-1);
}
```

Chamada inicial: `PrintLCS(m, n)`

Tempo: $\Theta(n+m)$

Problema da mochila

- Dado: um conjunto S de n itens, onde cada item i, $1 \leq i \leq n$, tem:
  - um peso positivo w_i
  - um lucro positivo associado p_i
- Objetivo: escolher os itens de S que proporcionem um lucro máximo, mas cujos pesos não ultrapassem a capacidade c da mochila.
- Quando os itens não podem ser repetidos nem fracionados, este problema é conhecido como *0/1 Knapsack Problem*.
- Neste caso, chamando de T o conjunto de itens selecionados, desejamos:

$$\text{maximizar} \sum_{i \in T} p_i \text{ sujeitos a } \sum_{i \in T} w_i \leq c$$

Equação de otimalidade

- Seja S_k o conjunto dos itens numerados de 1 até k, $1 \leq k \leq n$.
- Definimos B[k,w] como a solução ótima de S_k com peso máximo w, $0 \leq w \leq c$.
- Para calcular B[k,w], temos duas opções:
  - Se $w_k > w$, o item k não poderá entrar nessa solução. Portanto, B[k,w] será igual à solução de S_{k-1} com peso máximo w.
  - Caso contrário, será preciso verificar se vale a pena incluir o item k, utizando soluções ótimas de subproblemas menores.
- Isso permite-nos escrever a seguinte equação de otimalidade:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{, se } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+p_k\} & \text{, caso contrário} \end{cases}$$

Algoritmo

B[k,w]: solução ótima considerando apenas os k primeiros itens e uma mochila de capacidade w.

$$B[k,w] = \begin{cases} B[k-1,w] & \text{, se } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+p_k\} & \text{, caso contrário} \end{cases}$$

```
Knapsack01() {
   for (i=0; i<=c; i++)
      B[0,i] = 0;              // nenhum item é considerado
   for (k=1; k<=n; k++)        // incremento nos itens
      for (i=0; i<=c; i++)     // incremento na capacidade
        if (w[k] > i) B[k,i] = B[k-1,i];
        else B[k,i] = max {B[k-1,i], B[k-1,i-w[k]] + p[k]};
   return B[n,c]
}
```

Tempo: $\Theta(n.c)$

Exemplo com n=4

- $w_1=p_1=7$, $w_2=p_2=7$, $w_3=p_3=2$, $w_4=p_4=3$, $c=11$

B	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	7	7	7	7	7
2	0	0	0	0	0	0	0	7	7	7	7	7
3	0	0	2	2	2	2	2	7	7	9	9	9
4	0	0	2	3	3	5	5	7	7	9	10	10

- O preenchimento ótimo da mochila tem valor 10
- Como encontrar os itens dessa solução?

Vetor de itens

- Através do vetor binário $X = \{x_1, x_2, \ldots, x_n\}$, onde $x_i \in \{0, 1\}$, podemos representar os itens presentes na solução ótima .
- Considere o exemplo anterior, onde a mochila tem capacidade 11, e há 4 itens cujos lucros são iguais aos pesos:

	w
1	7
2	7
3	2
4	3

B	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	7	7	7	7	7
2	0	0	0	0	0	0	0	7	7	7	7	7
3	0	0	2	2	2	2	2	7	7	9	9	9
4	0	0	2	3	3	5	5	7	7	9	10	10

	X
1	1
2	0
3	0
4	1

- A solução ótima é formada pelos itens 1 e 4, com pesos $w_1=7$ e $w_4=3$.

Algoritmo

- Cálculo do vetor X, considerando o caso geral em que cada item i tem peso w[i] e lucro p[i], $1 \leq i \leq n$:

```
VetorX() {
    r = c;              // peso disponível na mochila
    s = B[n,c];         // lucro corrente
    for (i=n; i>0; i--)
        if (B[i-1,r] == s)
            X[i] = 0;  // item i não entrou
        else {
            X[i] = 1;  // item i entrou
            s -= p[i];
            r -= w[i];
        }
}
```

Tempo: $\Theta(n)$

Complexidade de tempo

- O problema da mochila, mesmo com outras variantes, é *NP-Difícil*, ou seja, não se conhece nenhuma resolução de tempo polinomial para ele.
- Embora pareça, $\Theta(n.c)$ não é polinomial. Na verdade, a capacidade c da mochila poderia ser de ordem exponencial em relação ao número de itens da mochila.
- Portanto, neste caso costuma-se dizer que $\Theta(n.c)$ é uma complexidade pseudopolinomial.
- Uma das melhores resoluções para o *0/1 Knapsack Problem* com n itens, sem o uso da *Programação Dinâmica*, é o algoritmo das duas listas, de Horowitz e Sahni, que gasta tempo e espaço $\Theta(2^{n/2})$.

A técnica *memoization*

- Há um modo simples de transformar um algoritmo de *Divisão-e-Conquista* em um "equivalente" do estilo *Programação Dinâmica*.
- Vimos que a *Divisão-e-Conquista* não é eficiente quando os subproblemas são sobrepostos: neste caso, os mesmos subproblemas acabam sendo resolvidos várias vezes...
- Uma forma de evitar esse mau desempenho é armazenar em vetores as subsoluções calculadas e, antes de chamar uma nova recursão, verificar se o caso em questão já foi ou não resolvido.
- Esta técnica é chamada de *memoization*.

Exemplo

Cálculo de Fibonacci com a técnica *memoization:*

```
int Fibonacci(n) {
     m[0] = m[1] = 1;
     for (i=2; i<=n; i++)
          m[i] = -1;  // indicação de que não foi resolvido
     return fib(n);
}
```

Tempo: $\Theta(n)$

Espaço: $\Theta(n)$

```
int fib(n) {
     if (m[n] < 0)   // verifica se já foi resolvido
        m[n] = fib(n-1) + fib(n-2);
     return m[n];
}
```

Comparações

- *Memoization* é uma técnica *top-down*, ao contrário de *Programação Dinâmica (bottom-up)*.
- Vantagens:
  - Pode ser mais simples de codificar e de validar.
  - A complexidade de tempo é a mesma do algoritmo de *Programação Dinâmica.*
  - Se algum subproblema não precisar ser resolvido, o tempo correspondente será economizado.
- Desvantagem:
  - Geralmente, o algoritmo *memoization* é mais lento devido ao *overhead* das chamadas recursivas.

Exercícios

- Simule a execução dos algoritmos de *Programação Dinâmica* abaixo, preenchendo suas tabelas a mão:
  - Moedas de troco
  - Encadeamento do produto de matrizes
  - Maior subsequência comum
  - Problema da mochila
- Escreva a versão *memoization* desses algoritmos.