Projeto e Análise de Algoritmos
Paradigmas de Projeto de Algoritmos

Antonio Alfredo Ferreira Loureiro

loureiro@dcc.ufmg.br

http://www.dcc.ufmg.br/~loureiro

Paradigmas de projeto de algoritmos

- Indução
- Recursividade
- Tentativa e erro
- Divisão e conquista
- Balanceamento
- Programação dinâmica
- Algoritmos gulosos
- Algoritmos aproximados

Princípio da indução matemática (fraca)

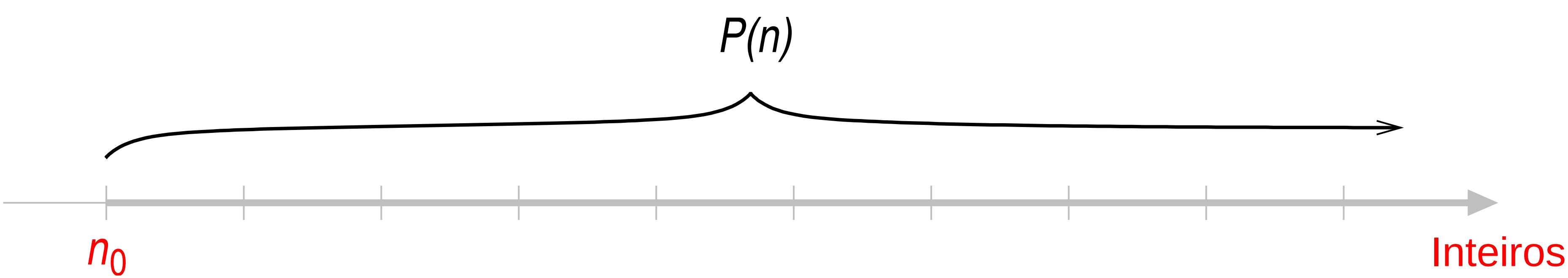
Seja $P(n)$ um predicado definido para os inteiros n, e seja n_0 um inteiro fixo. Suponha que as duas afirmações abaixo sejam verdadeiras:

1. $P(n_0)$ é V.
2. Para todos inteiros $k \geq n_0$,
   se $P(k)$ é V então $P(k+1)$ é V.

➜ Logo, a afirmação

para todos inteiros $n \geq n_0$, $P(n)$

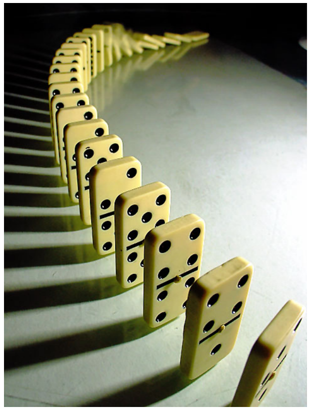
é V.



Princípio da indução matemática

- Técnica aparece pela primeira vez no trabalho do italiano Francesco Maurolico em 1575.
- No século XVII, Pierre de Fermat e Blaise Pascal usam essa técnica em seus trabalhos. Fermat dá o nome de "método do descendente infinito."
- Em 1883, Augustus De Morgan descreve o processo cuidadosamente e dá o nome de indução matemática.

➜ Técnica extremamente importante para a Ciência da Computação.



Para visualizar a idéia da indução matemática, imagine uma coleção de dominós colocados numa seqüência (formação) de tal forma que a queda do primeiro dominó força a queda do segundo, que força a queda do terceiro, e assim sucessivamente, até todos os dominós caírem.

Princípio da indução matemática (fraca)

- A prova de uma afirmação por indução matemática é feita em dois passos:
  1. Passo base: é provado que $P(n_0)$ é V para um dado n_0 específico.
  2. Passo indutivo: é provado que para todos inteiros $k \geq n_0$, se $P(k)$ é V então $P(k+1)$ é V.

     O passo indutivo pode ser escrito formalmente como:

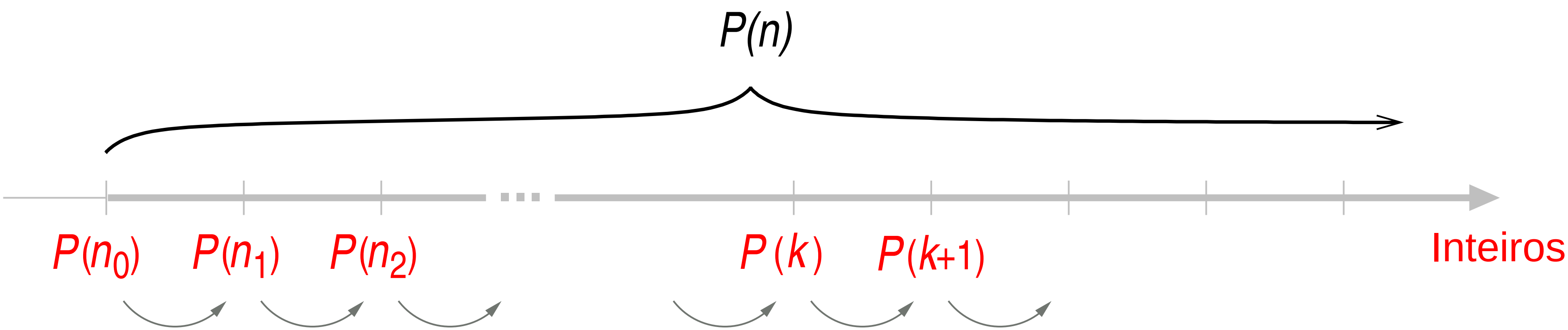
     $$\forall \text{ inteiros } k \geq n_0, \text{ se } P(k) \text{ então } P(k+1)$$

- Para provar o passo indutivo deve-se:
  - supor que $P(k)$ é V, onde k é um elemento específico mas escolhido arbitrariamente de tal forma que seja maior ou igual a n_0.
  - provar que $P(k+1)$ é V.

Princípio da indução matemática (fraca)

- Este princípio pode ser expresso pela seguinte regra de inferência:

$$[P(n_0) \wedge \forall k(P(k) \rightarrow P(k+1))] \rightarrow \forall n P(n).$$



➜ Numa prova por indução matemática ***não é*** assumido que $P(k)$ é verdadeiro para todos os inteiros! É mostrado que se ***for assumido*** que $P(k)$ é verdadeiro, então $P(k+1)$ também é verdadeiro.

Os próximos 10 exemplos ilustram o uso do Princípio da Indução Matemática e estão apresentados aqui para estudo e referência.

Princípio da indução matemática
Exemplo 1

Prove que para todos inteiros $n \geq 1$,

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

Prova (por indução matemática):

1. Passo base: $P(n_0) = P(1)$: Para $n_0 = 1$, $1 = \frac{1(1+1)}{2} = 1$ e a fórmula é verdadeira para $n_0 = 1$.

2. Passo indutivo: se a fórmula é verdadeira para $n = k$ então deve ser verdadeira para $n = k + 1$, i.e., $P(k) \rightarrow P(k+1)$.
   – Suponha que a fórmula seja verdadeira para $n = k$, i.e.,

   $$P(k) : 1 + 2 + \ldots + k = \frac{k(k+1)}{2}$$

   para algum inteiro $k \geq 1$. [hipótese indutiva]

Princípio da indução matemática
Exemplo 1

Deve-se mostrar que

$$P(k+1) : 1 + 2 + \ldots + (k+1) = \frac{(k+1)(k+2)}{2}$$

Sabe-se que

$$\begin{aligned}
1 + 2 + \ldots + k + (k+1) &= \frac{k(k+1)}{2} + (k+1) \\
&= \frac{k(k+1)}{2} + \frac{2(k+1)}{2} \\
&= \frac{k^2 + 3k + 2}{2} \\
&= \frac{(k+1)(k+2)}{2}
\end{aligned}$$

[Isto era o que devia ser provado.]

Princípio da indução matemática
Exemplo 2

Prove que para todos inteiros $n \geq 0$,

$$0 + 1 + 2 + \ldots + n = \frac{n(n+2)}{2}$$ ERRADO!

Prova (por indução matemática):

1. Passo base: $P(n_0) = P(0)$: Para $n_0 = 0$, $0 = \frac{0(0+2)}{2} = 0$ e a fórmula é verdadeira para $n_0 = 0$.

2. Passo indutivo: se a fórmula é verdadeira para $n = k$ então deve ser verdadeira para $n = k + 1$, i.e., $P(k) \rightarrow P(k + 1)$.
   – Suponha que a fórmula seja verdadeira para $n = k$, i.e.,

$$P(k) : 0 + 1 + 2 + \ldots + k = \frac{k(k+2)}{2} = \frac{k^2 + 2k}{2}$$

   para algum inteiro $k \geq 0$. [hipótese indutiva]

Princípio da indução matemática
Exemplo 2

Deve-se mostrar que

$$P(k+1) : 0 + 1 + 2 + \ldots + (k+1) = \frac{(k+1)(k+3)}{2} = \frac{k^2 + 4k + 3}{2}$$

Sabe-se que

$$\begin{aligned} 0 + 1 + 2 + \ldots + k + (k+1) &= \frac{k^2 + 2k}{2} + (k+1) \\ &= \frac{k^2 + 2k + 2(k+1)}{2} \\ &= \frac{k^2 + 4k + 2}{2} \end{aligned}$$

[Assim, não foi possível derivar a conclusão a partir da hipótese. Isto significa que o predicado original é falso.]

Princípio da indução matemática
Exemplo 3

Prove que

$$P(n) : \sum_{i=0}^{n} r^i = \frac{r^{n+1} - 1}{r - 1}$$

para todos inteiros $n \geq 0$ e para todos números reais r, $r \neq 1$.

Prova (por indução matemática):

1. Passo base: $P(n_0) = P(0)$: Para $n_0 = 0$, $r^0 = 1 = \frac{r^{0+1}-1}{r-1} = \frac{r-1}{r-1} = 1$ e a fórmula é verdadeira para $n_0 = 0$.

2. Passo indutivo: se a fórmula é verdadeira para $n = k$ então deve ser verdadeira para $n = k + 1$, i.e., $P(k) \rightarrow P(k + 1)$.

Princípio da indução matemática
Exemplo 3

– $P(k)$: $\sum_{i=0}^{k} r^i = \frac{r^{k+1}-1}{r-1}$, para $k \geq 0$. [hipótese indutiva]

– Deve-se mostrar que $P(k+1)$: $\sum_{i=0}^{k+1} r^i = \frac{r^{k+2}-1}{r-1}$

$$
\begin{aligned}
\sum_{i=0}^{k+1} r^i &= \sum_{i=0}^{k} r^i + r^{k+1} \\
&= \frac{r^{k+1}-1}{r-1} + r^{k+1} \\
&= \frac{r^{k+1}-1}{r-1} + \frac{r^{k+1}(r-1)}{r-1} \\
&= \frac{r^{k+1}-1+r^{k+2}-r^{k+1}}{r-1} \\
&= \frac{r^{k+2}-1}{r-1}
\end{aligned}
$$

Princípio da indução matemática
Exemplo 4

Prove que

$$P(n): 2^{2n} - 1 \text{ é divisível por 3,}$$

para $n \geq 1$.

Prova (por indução matemática):

1. Passo base: $P(n_0) = P(1)$: Para $n_0 = 1$, $2^{2 \cdot 1} - 1 = 3$ que é divisível por 3.

2. Passo indutivo: se a fórmula é verdadeira para $n = k$ então deve ser verdadeira para $n = k + 1$, i.e., $P(k) \rightarrow P(k + 1)$.

Princípio da indução matemática
Exemplo 4

- $P(k)$: $2^{2k} - 1$ é divisível por 3. [hipótese indutiva]
- Deve-se mostrar que $P(k+1)$: $2^{2(k+1)} - 1$ é divisível por 3.

$$
\begin{aligned}
2^{2(k+1)} - 1 &= 2^{2k+2} - 1 \\
&= 2^{2k} \cdot 2^2 - 1 \\
&= 2^{2k} \cdot 4 - 1 \\
&= 2^{2k} \cdot (3+1) - 1 \\
&= 2^{2k} \cdot 3 + (2^{2k} - 1)
\end{aligned}
$$

que é divisível por 3.

Princípio da indução matemática
Exemplo 5

Prove que

$$P(n) : 2^0 + 2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 1,$$

para $n \geq 0$.

Prova (por indução matemática):

1. Passo base: $P(n_0) = P(0)$: Para $n_0 = 2^0 = 1$, $2^1 - 1 = 1$.
2. Passo indutivo: se a fórmula é verdadeira para $n = k$ então deve ser verdadeira para $n = k + 1$, i.e., $P(k) \rightarrow P(k+1)$.

Princípio da indução matemática
Exemplo 5

– $P(k): 2^0 + 2^1 + 2^2 + \ldots + 2^k = 2^{k+1} - 1$, para $k \geq 0$. [hipótese indutiva]

– Deve-se mostrar que $P(k+1): 2^0 + 2^1 + 2^2 + \ldots + 2^{k+1} = 2^{k+2} - 1$

$$\begin{aligned} 2^0 + 2^1 + 2^2 + \ldots + 2^k + 2^{k+1} &= (2^{k+1} - 1) + 2^{k+1} \\ &= 2 \cdot 2^{k+1} - 1 \\ &= 2^{k+2} - 1 \end{aligned}$$

Princípio da indução matemática
Exemplo 6

Prove que

$$P(n) : H_{2^n} \geq 1 + \frac{n}{2},$$

para $n \geq 0$, onde H_j representa o número harmônico, que é definido por:

$$H_j = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{j}.$$

Prova (por indução matemática):

1. Passo base: $P(n_0) = P(0)$:
   Para $n_0 = 0$, temos $H_{2^0} = H_1 = 1 \geq 1 + \frac{0}{2}$.

2. Passo indutivo: se a fórmula é verdadeira para $n = k$ então deve ser verdadeira para $n = k + 1$, i.e., $P(k) \to P(k + 1)$.

Princípio da indução matemática
Exemplo 6

- $P(k) : H_{2^k} \geq 1 + \frac{k}{2}$, para $k \geq 0$. [hipótese indutiva]
- Deve-se mostrar que $P(k+1) : H_{2^{k+1}} \geq 1 + \frac{k+1}{2}$

$$H_{2^{k+1}} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{2^k} + \frac{1}{2^k+1} + \frac{1}{2^k+2} + \ldots + \frac{1}{2^{k+1}}$$

[Definição de número harmônico.]

$$= H_{2^k} + \frac{1}{2^k+1} + \frac{1}{2^k+2} + \ldots + \frac{1}{2^{k+1}}$$

[Definição de número harmônico.]

$$\geq \left(1 + \frac{k}{2}\right) + 2^k \cdot \frac{1}{2^{k+1}}$$

[Hipótese indutiva e existem 2^k termos, cada um pelo menos $1/2^{k+1}$.]

$$\geq \left(1 + \frac{k}{2}\right) + \frac{1}{2}$$

$$\geq 1 + \frac{k+1}{2}.$$

Princípio da indução matemática
Exemplo 7

Seja a seqüência $a_1, a_2, a_3, \ldots$ definida como

$$\begin{aligned} a_1 &= 2 \\ a_k &= 5a_{k-1}, k \geq 2 \end{aligned}$$

Prove que

$$a_n = 2 \cdot 5^{n-1}$$

para $n \geq 1$.

Prova (por indução matemática):

1. Passo base: $P(n_0) = P(1)$: Para $n_0 = 1$, $2 \cdot 5^{1-1} = 2$ e $a_1 = 2$. Logo, a fórmula é válida para $n = 1$.
2. Passo indutivo: se a fórmula é verdadeira para $n = k$ então deve ser verdadeira para $n = k + 1$, i.e., $P(k) \rightarrow P(k + 1)$.

Princípio da indução matemática
Exemplo 7

– $P(k) : a_k = 2 \cdot 5^{k-1}$. [hipótese indutiva]

– Deve-se mostrar que $P(k+1) : a_{k+1} = 2 \cdot 5^{(k+1)-1} = 2 \cdot 5^k$.

$$
\begin{aligned}
a_{k+1} &= 5 \cdot a_{(k+1)-1} \\
&= 5 \cdot a_k \\
&= 5 \cdot (2 \cdot 5^{k-1}) \\
&= 2 \cdot (5 \cdot 5^{k-1}) \\
&= 2 \cdot 5^k
\end{aligned}
$$

Princípio da indução matemática
Exemplo 8

Prove que para todos os inteiros $n \geq 3$

$$P(n)\colon\ 2n + 1 < 2^n$$

Prova (por indução matemática):

1. Passo base: $P(n_0) = P(3)$. Para $n_0 = 3$,

$$2 \cdot 3 + 1 < 2^3.$$

   Logo, a fórmula é válida para $n_0 = 3$.
2. Passo indutivo: se a fórmula é verdadeira para $n = k$ então deve ser verdadeira para $n = k + 1$, i.e., $P(k) \to P(k + 1)$.

Princípio da indução matemática
Exemplo 8

– $P(k)$: $2k+1 < 2^k$, para $k \geq 3$. [hipótese indutiva]

– Deve-se mostrar que $P(k+1)$: $2(k+1)+1 < 2^{k+1}$.

$$\begin{array}{rcl} 2k+2+1 & = & \\ (2k+1)+2 & = & \\ (2k+1)+2 & < & 2^k+2 \\ 2(k+1)+1 & < & 2^k+2 \stackrel{?}{<} 2^{k+1} \end{array}$$

Se puder ser mostrado que $2^k+2 < 2^{k+1}$ então o predicado $P(k+1)$ é verdadeiro.

$$\begin{array}{rcl} 2^k+2 & \stackrel{?}{<} & 2^{k+1} \\ 2 & \stackrel{?}{<} & 2^{k+1}-2^k \\ 2 & \stackrel{?}{<} & 2^k(2-1) \\ 2 & \stackrel{?}{<} & 2^k \\ 1 & < & 2^{k-1}, \text{que é verdade para } k \geq 2. \end{array}$$

Em particular, a inequação ($1 < 2^{k-1}$) é válida para $k \geq 3$. Assim, $P(k+1)$ é V.

Princípio da indução matemática
Exemplo 9

Prove que para todos os inteiros $n \geq 1$

$$P(n)\text{: } n^3 - n \text{ é divisível por 3.}$$

Prova (por indução matemática):

1. Passo base: $P(n_0) = P(1)$. Para $n_0 = 1$,

   $$1^3 - 1 = 0 \text{ é divisível por 3.}$$

   Logo, a fórmula é válida para $n_0 = 3$.
2. Passo indutivo: se a fórmula é verdadeira para $n = k$ então deve ser verdadeira para $n = k + 1$, i.e., $P(k) \rightarrow P(k + 1)$.

Princípio da indução matemática
Exemplo 9

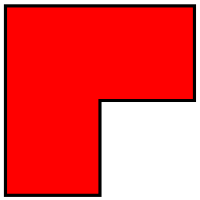
- $P(k)$: $k^3 - k$ é divisível por 3, para $k \geq 1$. [hipótese indutiva]
- Deve-se mostrar que $P(k+1)$: $(k+1)^3 - (k+1)$ é divisível por 3, para $k \geq 1$.

$$
\begin{aligned}
&(k+1)^3 - (k+1) = \\
&(k^3 + 3k^2 + 3k + 1) - (k+1) = \\
&(k^3 - k) + 3(k^2 + k)
\end{aligned}
$$

O primeiro termo é divisível por 3 (hipótese indutiva) e o segundo também. Como a soma de dois números divisíveis por 3 é um número divisível por 3, então o predicado $P(k+1)$ é V.

Princípio da indução matemática
Exemplo 10

Seja o inteiro $n \geq 1$. Mostre que qualquer região quadrada de tamanho $2^n \times 2^n$, com um quadrado removido, a região restante pode ser preenchida com peças no formato L, como mostrado abaixo.



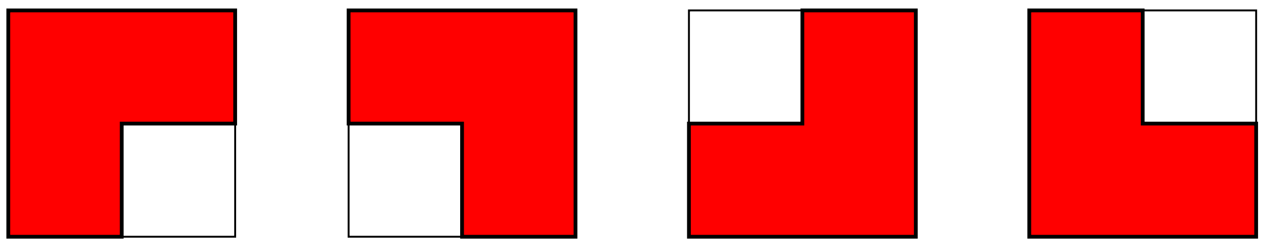
Nota: A peça no formato L é constituída por três quadrados de tamanho 1×1.

Prove que para todos os inteiros $n \geq 1$, $P(n)$: Qualquer região quadrada de tamanho $2^n \times 2^n$, com um quadrado removido, a região restante pode ser preenchida com peças no formato L.

Princípio da indução matemática
Exemplo 10

Prova (por indução matemática):

1. Passo base: $P(n_0) = P(1)$. P(1) é V já que uma região quadrada de tamanho 2×2, com um quadrado removido, a região restante pode se preenchida com peças no formato L, como mostrado abaixo.

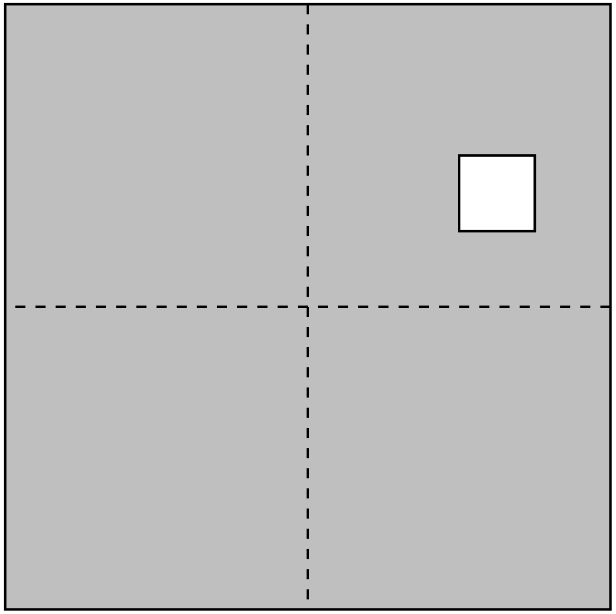


2. Passo indutivo: se a fórmula é verdadeira para $n = k$ então deve ser verdadeira para $n = k + 1$, i.e., $P(k) \rightarrow P(k + 1)$.

Princípio da indução matemática
Exemplo 10

- $P(k)$: Qualquer região quadrada de tamanho $2^k \times 2^k$, com um quadrado removido, a região restante pode ser preenchida com peças no formato L. [hipótese indutiva]
- Deve-se mostrar $P(k+1)$: Qualquer região quadrada de tamanho $2^{k+1} \times 2^{k+1}$, com um quadrado removido, a região restante pode ser preenchida com peças no formato L.

Considere uma região quadrada de tamanho $2^{k+1} \times 2^{k+1}$, com um quadrado removido. Divida essa região em quatro regiões de tamanho $2^k \times 2^k$ como mostrado abaixo.



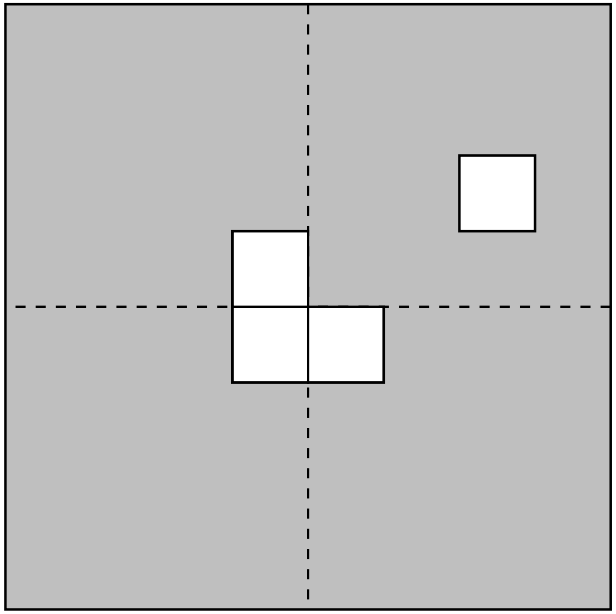
Temos três regiões $2^k \times 2^k$ com nenhum quadrado removido e uma região $2^k \times 2^k$ com um quadrado removido. Ou seja, a região $2^{k+1} \times 2^{k+1}$ possui apenas um quadrado removido.

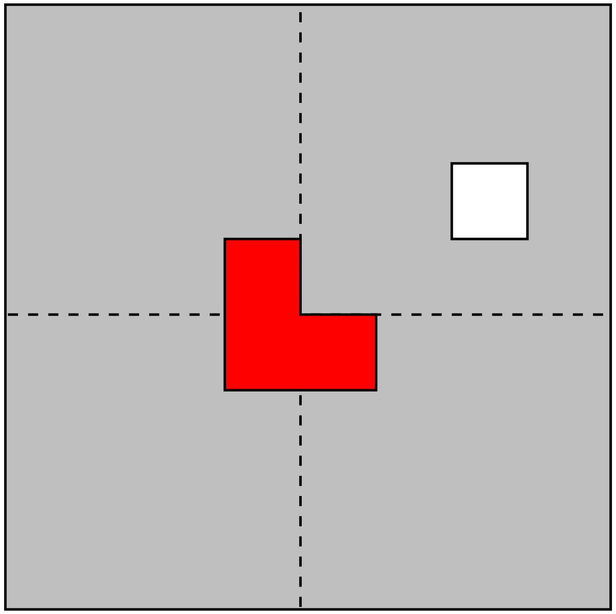
Pela hipótese indutiva, a região $2^k \times 2^k$, com um quadrado removido, pode ser preenchida com peças no formato L. O problema passa a ser como a mesma hipótese indutiva pode ser aplicada às outras três regiões.

Princípio da indução matemática
Exemplo 10

Temporariamente remova um quadrado de cada região $2^k \times 2^k$ que está "completa" como mostrado na figura abaixo à esquerda.

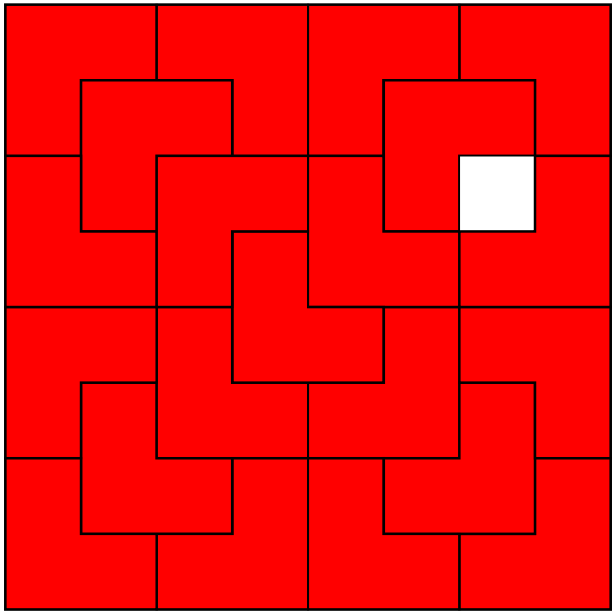
Pela hipótese indutiva cada uma dessas três regiões $2^k \times 2^k$ pode ser preenchida com peças no formato L. No entanto, para resolvermos o problema da peça removida em cada uma dessas três regiões basta colocarmos uma peça L exatamente sobre esses três "buracos" como mostrado na figura abaixo à direita.





Princípio da indução matemática
Exemplo 10

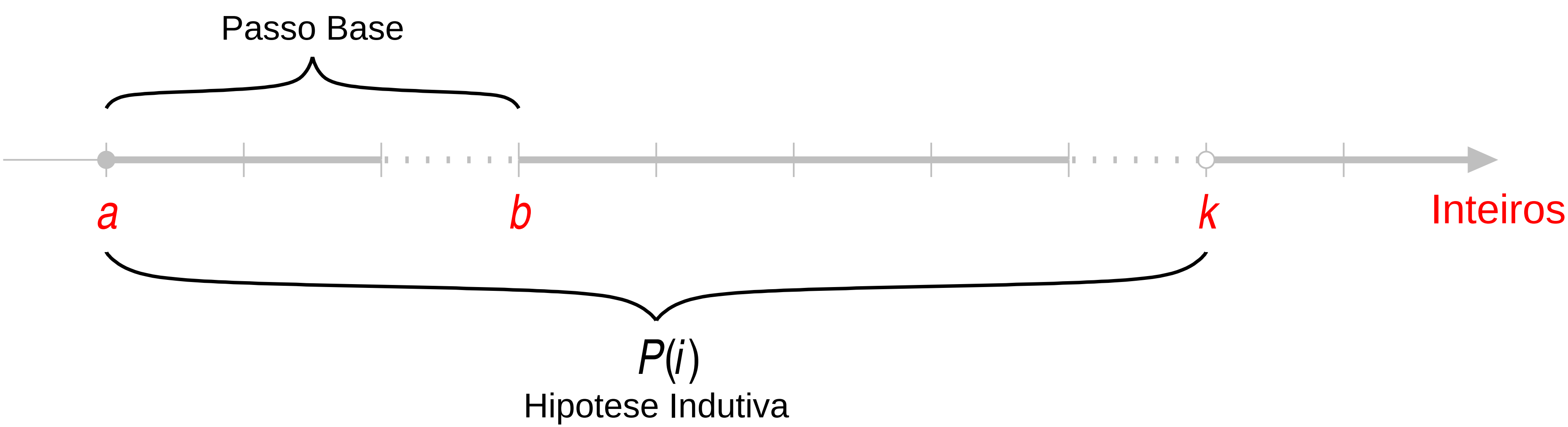
Assim, uma região quadrada de tamanho $2^{k+1} \times 2^{k+1}$, com um quadrado removido, a região restante pode ser preenchida com peças no formato L, como mostrado na figura abaixo.



Princípio da indução matemática (forte)

Seja $P(n)$ um predicado que é definido para inteiros n, e seja a e b inteiros fixos, sendo $a \leq b$. Suponha que as duas afirmações seguintes sejam verdadeiras:

1. $P(a), P(a+1), \ldots, P(b)$ são V. (Passo base)
2. Para qualquer inteiro $k \geq b$,
   se $P(i)$ é V para $a \leq i < k$ então $P(k)$ é V, i.e., $P(i) \rightarrow P(k)$.
   ➜ Logo, a afirmação "para todos inteiros $n \geq a$, $P(n)$" é V. (A suposição que $P(i)$ é V para $a \leq i < k$ é chamada de hipótese indutiva.)



Princípio da indução matemática (forte): Exemplo 11

Seja a sequência $a_1, a_2, a_3, \ldots$ definida como

$$\begin{aligned} a_1 &= 0 \\ a_2 &= 2 \\ a_k &= 3 \cdot a_{\lfloor k/2 \rfloor} + 2, \; k \geq 3 \end{aligned}$$

Prove que a_n é par, para $n \geq 1$.

Prova (por indução matemática):

1. Passo base: Para $n = 1$ e $n = 2$ a propriedade é válida já que $a_1 = 0$ e $a_2 = 2$.

2. Passo indutivo: Vamos supor que a_i é par para todos inteiros i, $1 \leq i < k$. [hipótese indutiva]

Princípio da indução matemática (forte): Exemplo 11

Se a propriedade é válida para $1 \leq i < k$, então é válida para k, ou seja, a_k é par [o que deve ser mostrado].

Pela definição de $a_1, a_2, a_3, \ldots$

$$a_k = 3 \cdot a_{\lfloor k/2 \rfloor} + 2, \;\; k \geq 3$$

O termo $a_{\lfloor k/2 \rfloor}$ é par pela hipótese indutiva já que $k \geq 3$ e $1 \leq \lfloor k/2 \rfloor < k$. Desta forma, $3 \cdot a_{\lfloor k/2 \rfloor}$ é par e $3 \cdot a_{\lfloor k/2 \rfloor} + 2$ também é par, o que mostra que a_k é par.

Indução matemática e algoritmos

- É útil para provar asserções sobre a correção e a eficiência de algoritmos.
- Consiste em inferir uma lei geral a partir de instâncias particulares.
- Seja T um teorema que tenha como parâmetro um número natural n. Para provar que T é válido para todos os valores de n, provamos que:
  1. T é válido para $n = 1$; [PASSO BASE]
  2. Para todo $n > 1$, [PASSO INDUTIVO]
     se T é válido para n,
     então T é válido para $n + 1$.
- Provar a condição 2 é geralmente mais fácil que provar o teorema diretamente (podemos usar a asserção de que T é válido para n).
- As condições 1 e 2 implicam T válido para $n = 2$, o que junto com a condição 2 implica T também válido para $n = 3$, e assim por diante.

Limite superior de equações de recorrência

- A solução de uma equação de recorrência pode ser difícil de ser obtida.
- Nesses casos, pode ser mais fácil tentar advinhar a solução ou obter um limite superior para a ordem de complexidade.
- Advinhar a solução funciona bem quando estamos interessados apenas em um limite superior, ao invés da solução exata.
  ➜ Mostrar que um certo limite existe é mais fácil do que obter o limite.
- Por exemplo:

$$
\begin{aligned}
T(2n) &\leq 2T(n) + 2n - 1, \\
T(2) &= 1,
\end{aligned}
$$

definida para valores de n que são potências de 2.

➜ O objetivo é encontrar um limite superior na notação O, onde o lado direito da desigualdade representa o pior caso.

Indução matemática para resolver equação de recorrência

$$\begin{aligned} T(2) &= 1, \\ T(2n) &\leq 2T(n) + 2n - 1, \end{aligned}$$

definida para valores de n que são potências de 2.

- Procuramos $f(n)$ tal que $T(n) = O(f(n))$, mas fazendo com que $f(n)$ seja o mais próximo possível da solução real para $T(n)$ (limite assintótico firme).
- Vamos considerar o palpite $f(n) = n^2$.
- Queremos provar que

  $$T(n) \leq f(n) = O(f(n))$$

  utilizando indução matemática em n.

Indução matemática para resolver equação de recorrência

Prove que $T(n) \le f(n) = O(f(n))$, para $f(n) = n^2$, sendo

$$\begin{array}{rcl} T(2) & = & 1, \\ T(2n) & \le & 2T(n) + 2n - 1, \end{array}$$

definida para valores de n que são potências de 2.

Prova (por indução matemática):

1. Passo base:
   $T(n_0) = T(2)$: Para $n_0 = 2$, $T(2) = 1 \le f(2) = 4$, e o passo base é V.

2. Passo indutivo: se a recorrência é verdadeira para n então deve ser verdadeira para $2n$, i.e., $T(n) \to T(2n)$ (lembre-se que n é uma potência de 2; conseqüentemente o "número seguinte" a n é $2n$).

   Reescrevendo o passo indutivo temos:

$$\begin{array}{rcl} \text{Predicado}(n) & \to & \text{Predicado}(2n) \\ (T(n) \le f(n)) & \to & (T(2n) \le f(2n)) \end{array}$$

$$\begin{array}{rcll} T(2n) & \le & 2T(n) + 2n - 1 & \text{[Definição da recorrência]} \\ & \le & 2n^2 + 2n - 1 & \text{[Pela hipótese indutiva podemos substituir } T(n)\text{]} \\ & \le & 2n^2 + 2n - 1 \overset{?}{<} (2n)^2 & \text{[A conclusão é verdadeira?]} \\ & \le & 2n^2 + 2n - 1 < 4n^2 & \text{[Sim!]} \end{array}$$

   Essa última inequação é o que queremos provar. Logo, $T(n) = O(n^2)$.

Indução matemática para resolver equação de recorrência

- Vamos tentar um palpite menor, $f(n) = cn$, para alguma constante c.
- Queremos provar que

$$T(n) \leq f(n) = cn = O(f(n))$$

utilizando indução matemática em n.

Indução matemática para resolver equação de recorrência

Prove que $T(n) \leq f(n) = O(f(n))$, para $f(n) = cn$, sendo

$$\begin{aligned} T(2) &= 1, \\ T(2n) &\leq 2T(n) + 2n - 1, \end{aligned}$$

definida para valores de n que são potências de 2.

Prova (por indução matemática):

1. Passo base:
   $T(n_0) = T(2)$: Para $n_0 = 2$, $T(2) = 1 \leq f(2) = 2c$, e o passo base é V.

2. Passo indutivo: se a recorrência é verdadeira para n então deve ser verdadeira para $2n$, i.e., $T(n) \rightarrow T(2n)$.

   Reescrevendo o passo indutivo temos:

$$\begin{aligned} \text{Predicado}(n) &\rightarrow \text{Predicado}(2n) \\ (T(n) \leq f(n)) &\rightarrow (T(2n) \leq f(2n)) \\ (T(n) \leq cn)) &\rightarrow (T(2n) \leq 2cn) \end{aligned}$$

$$\begin{aligned} T(2n) &\leq 2T(n) + 2n - 1 && \text{[Definição da recorrência]} \\ &\leq 2cn + 2n - 1 && \text{[Pela hipótese indutiva podemos substituir } T(n)\text{]} \\ &\leq 2cn + (2n - 1) && \\ &\leq 2cn + 2n - 1 > 2cn && \text{[A conclusão } (T(2n) \leq 2cn) \text{ não é válida]} \end{aligned}$$

Indução matemática para resolver equação de recorrência

Logo:

- a função $f(n) = cn$ cresce mais lentamente que $T(n)$;
- $T(n)$ está entre cn e n^2, mais especifamente;

e $T(n) \not\leq f(n) = cn$.

Indução matemática para resolver equação de recorrência

- Vamos tentar uma função entre n e n^2, como, por exemplo, $f(n) = n \log n$.
- Queremos provar que

$$T(n) \leq f(n) = n \log n = O(f(n))$$

utilizando indução matemática em n.

Indução matemática para resolver equação de recorrência

Prove que $T(n) \leq f(n) = O(f(n))$, para $f(n) = n \log n$, sendo

$$\begin{aligned} T(2) &= 1, \\ T(2n) &\leq 2T(n) + 2n - 1, \end{aligned}$$

definida para valores de n que são potências de 2.

Prova (por indução matemática):

1. Passo base:
   $T(n_0) = T(2)$: Para $n_0 = 2$, $T(2) = 1 \leq f(2) = 2 \log 2$, e o passo base é V.

2. Passo indutivo: se a recorrência é verdadeira para n então deve ser verdadeira para $2n$, i.e., $T(n) \rightarrow T(2n)$.

Reescrevendo o passo indutivo temos:

$$\begin{array}{rcl} \text{Predicado}(n) & \rightarrow & \text{Predicado}(2n) \\ (T(n) \leq f(n)) & \rightarrow & (T(2n) \leq f(2n)) \\ (T(n) \leq n \log n)) & \rightarrow & (T(2n) \leq 2n \log 2n) \end{array}$$

$$\begin{array}{rlll} T(2n) & \leq & 2T(n) + 2n - 1 & \text{[Definição da recorrência]} \\ & \leq & 2n \log n + 2n - 1 & \text{[Podemos substituir } T(n)\text{]} \\ & \leq & 2n \log n + 2n - 1 \overset{?}{<} 2n \log 2n & \text{[A conclusão é verdadeira?]} \\ & \leq & 2n \log n + 2n - 1 < 2n \log n + 2n & \text{[Sim!]} \end{array}$$

Indução matemática para resolver equação de recorrência

- Para o valor de $f(n) = n \log n$, a diferença entre as fórmulas é de apenas 1.

- De fato, $T(n) = n \log n - n + 1$ é a solução exata de

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + n - 1 \\ T(1) &= 0 \end{aligned}$$

que descreve o comportamento do algoritmo de ordenação *Mergesort*.

Indução matemática e algoritmos
Comentários finais

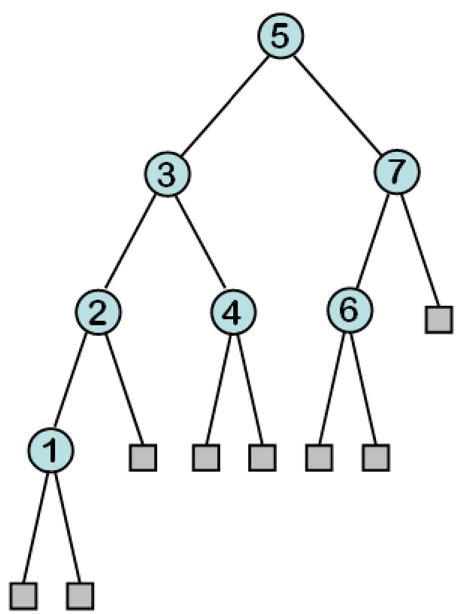
- Indução é uma das técnicas mais poderosas da Matemática que pode ser aplicada para provar asserções sobre a correção e a eficiência de algoritmos.

- No caso de correção de algoritmos, é comum tentarmos identificar invariantes para laços.

- Indução pode ser usada para derivar um limite superior para uma equação de recorrência.

Recursividade

- Um procedimento que chama a si mesmo, direta ou indiretamente, é dito ser **recursivo**.
- Recursividade permite descrever algoritmos de forma mais clara e concisa, especialmente problemas recursivos por natureza ou que utilizam estruturas recursivas.
- Por exemplo, árvore binária de pesquisa:
  - Todos os registros com chaves menores estão na sub-árvore esquerda;
  - Todos os registros com chaves maiores estão na sub-árvore direita.

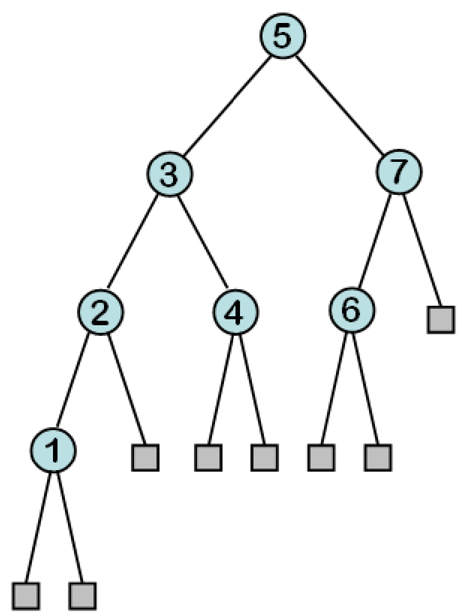


Recursividade

- Algoritmo para percorrer todos os registros em ordem de **caminhamento central**:
  1. Caminha na sub-árvore esquerda na ordem central;
  2. Visita a raiz;
  3. Caminha na sub-árvore direita na ordem central.

- No caminhamento central, os nós são visitados em ordem lexicográfica das chaves.

```
CENTRAL(p)
if p ≠ nil
  then CENTRAL(p↑.esq)
       Visita nó                 ▷ Faz algum processamento
       CENTRAL(p↑.dir)
```



Implementação de recursividade

- Usa-se uma **pilha** para armazenar os dados usados em cada chamada de um procedimento que ainda não terminou.

- Todos os dados não globais vão para a pilha, registrando o estado corrente da computação.

- Quando uma ativação anterior prossegue, os dados da pilha são recuperados.

- No caso do caminhamento central:
  - Para cada chamada recursiva, o valor de p e o endereço de retorno da chamada recursiva são armazenados na pilha.
  - Quando encontra p=nil o procedimento retorna para quem chamou utilizando o endereço de retorno que está no topo da pilha.

Problema de terminação em procedimentos recursivos

- Procedimentos recursivos introduzem a possibilidade de iterações que podem não terminar:
  ➜ Existe a necessidade de considerar o problema de **terminação**.

- É fundamental que a chamada recursiva a um procedimento P esteja sujeita a uma condição B, a qual se torna não-satisfeita em algum momento da computação.

- Esquema para procedimentos recursivos: composição $\mathcal{C}$ de comandos S_i e P.

$$P \equiv \textbf{if } B \textbf{ then } \mathcal{C}[S_i, P]$$

- Para demonstrar que uma repetição termina, define-se uma função $f(x)$, sendo x o conjunto de variáveis do programa, tal que:
  1. $f(x) \leq 0$ implica na condição de terminação;
  2. $f(x)$ é decrementada a cada iteração.

Problema de terminação em procedimentos recursivos

- Uma forma simples de garantir terminação é associar um parâmetro n para P (no caso **por valor**) e chamar P recursivamente com $n - 1$.

- A substituição da condição B por $n > 0$ garante terminação.

$$P \equiv \textbf{if } n > 0 \textbf{ then } \mathcal{P}[S_i, P(n-1)]$$

- É necessário mostrar que o nível mais profundo de recursão é finito, e também possa ser mantido pequeno, pois cada ativação recursiva usa uma parcela de memória para acomodar as variáveis.

Quando não usar recursividade

- Nem todo problema de natureza recursiva deve ser resolvido com um algoritmo recursivo.

- Estes podem ser caracterizados pelo esquema $P \equiv$ **if** B **then** (S, P).

- Tais programas são facilmente transformáveis em uma versão não recursiva $P \equiv (x := x_0;$ **while** B **do** $S)$.

Exemplo de quando não usar recursividade

- Cálculo dos **números de Fibonacci**

$$
\begin{aligned}
f_0 &= 0, \\
f_1 &= 1, \\
f_n &= f_{n-1} + f_{n-2}, \qquad \forall n \geq 2.
\end{aligned}
$$

- Solução:

$$f_n = \frac{1}{\sqrt{5}}[\Phi^n - (-\Phi)^{-n}],$$

onde $\Phi = \frac{\sqrt{5}+1}{2} \approx 1,618$ é a **razão de ouro.**

- O procedimento recursivo (Fibonacci_Rec) obtido diretamente da equação é o seguinte:

```
FIBONACCI_REC(n)
if n < 2
   then FIBONACCI_REC ← n
   else FIBONACCI_REC ← FIBONACCI_REC(n − 1) + FIBONACCI_REC(n − 2)
```

Exemplo de quando não usar recursividade

- O programa é extremamente ineficiente porque recalcula o mesmo valor várias vezes.

- A complexidade de espaço para calcular f_n é $O(\Phi^n)$.

- A complexidade de tempo para calcular f_n, considerando como medida de complexidade o número de adições, é também $O(\Phi^n)$.

Versão iterativa do cálculo de Fibonacci

```
FIBONACCI_ITER(n)
    ▷ Variáveis auxiliares: Aux, k, Fant, F
    Fant ← 0
    F ← 1
    for k ← 2 to n
        do Aux ← F + Fant
           Fant ← F
           F ← Aux
    FIBONACCI_ITER ← F
```

- O programa tem complexidades de tempo $O(n)$ e de espaço $O(1)$.
- Deve-se evitar recursividade quando existe uma solução iterativa.
- Comparação das versões recursiva e iterativa:

n	20	30	50	100
Recursiva	1 s	2 min	21 dias	10^9 anos
Iterativa	1/3 ms	1/2 ms	3/4 ms	1,5 ms

Recursividade na modelagem de problemas
Strings com uma certa propriedade (1)

Seja $\Sigma = \{0, 1\}$. Determine quantos strings existem em $\Sigma^0 \ldots \Sigma^3$ que não contém o padrão 11.

Nota: Σ^i é o conjunto de todos os strings de tamanho i sobre Σ.

Logo, temos que:

Tamanho	Strings
0	ε
1	$0, 1$
2	$00, 01, 10$
3	$000, 001, 010, 100, 101$

Tamanho	# Strings
0	1
1	2
2	3
3	5

Recursividade na modelagem de problemas *Strings com uma certa propriedade* (2)

Quantos elementos existem em Σ^k?

Idéia:

– Suponha que o número de strings $\leq k$ que não contém o padrão $\mathtt{11}$ seja conhecido.

– Use esse fato para determinar o número de strings de tamanho k que não contém $\mathtt{11}$ em função de strings menores que não contém $\mathtt{11}$.

Recursividade na modelagem de problemas
Strings com uma certa propriedade (3)

- Dois casos a considerar em função do símbolo mais à esquerda no string:
  - 0: os $k-1$ símbolos podem ser qualquer seqüência sobre Σ onde 11 não aparece;
  - 1: os dois símbolos mais à esquerda não podem ser 11 e sim 10.
    - ➜ Logo, os $k-2$ símbolos podem ser qualquer seqüência sobre Σ onde 11 não aparece.

- Os dois casos geram dois subconjuntos mutuamente disjuntos, representados pela primeira equação de recorrência abaixo:

$$
\begin{array}{lrcll}
(1) & s_k & = & s_{k-1} + s_{k-2} & \text{Equação de recorrência} \\
(2) & \left\{ \begin{array}{l} s_0 \\ s_1 \end{array} \right. & \begin{array}{c} = \\ = \end{array} & \begin{array}{l} 1 \\ 2 \end{array} & \text{Condições iniciais}
\end{array}
$$

➜ Termos da Série de Fibonacci!

Função definida recursivamente (1)

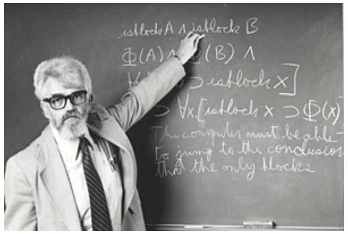
- Uma função é dita ser definida recursivamente se ela refere-se a si mesma.

- Funções recursivas têm um papel fundamental em teoria da computação.

- Exemplo: Função 91 de McCarthy.

$$M(n) = \begin{cases} n - 10 & \text{se } n > 100 \\ M(M(n+11)) & \text{se } n \leq 100 \end{cases}$$

$$\begin{aligned} M(99) &= M(M(110)) \\ &= M(100) \\ &= M(M(111)) \\ &= M(101) \\ &= 91 \end{aligned}$$



A função 91 de McCarthy é uma função recursiva que retorna 91 para todos os inteiros $n \leq 100$ e retorna $n - 10$ para $n > 100$. Essa função foi proposta pelo cientista da computação John McCarthy, ganhador do *ACM Turing Award* de 1971, responsável por cunhar o termo Inteligência Artificial.

Função definida recursivamente (2)
Função de Ackermann

$$
\begin{aligned}
A(0,n) &= n+1\\
A(m,0) &= A(m-1,1)\\
A(m,n) &= A(m-1,A(m,n-1))
\end{aligned}
$$

$$
\begin{aligned}
A(1,2) &= A(0,A(1,1))\\
&= A(0,A(0,A(1,0)))\\
&= A(0,A(0,A(0,1)))\\
&= A(0,A(0,2))\\
&= A(0,3)\\
&= 4
\end{aligned}
$$

Matemático e lógico alemão (1896–1962), principal formulador do desenvolvimento do sistema lógico conhecido como o cálculo de epsilon, originalmente devido a David Hilbert (1862–1943), que se tornaria a base da lógica de Bourbaki e da teoria dos jogos.

Função definida recursivamente (3)
Função de Ackermann

Essa função possui uma taxa de crescimento impressionante:

$$A(4,3) = A(3, 2^{65536} - 3)$$

Função importante em Ciência da Computação que está relacionada com computabilidade.

Função definida recursivamente (4)
Função de Ackermann

A função de Ackermann pode ser representada por uma tabela infinita.

(m,n)	0	1	2	3	4	$A(m,n)$
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$n+2$
2	3	5	7	9	11	$2n+3$
3	5	13	29	61	125	$8 \cdot 2^n - 3$
4	13	65533	$2^{65536} - 3$	$A(3, 2^{65536} - 3)$	$A(3, A(4,3))$	
5	65533	$A(4, 65533)$	$A(4, A(5,1))$	$A(4, A(5,2))$	$A(4, A(5,3))$	
6	$A(5,1)$	$A(5, A(5,1))$	$A(5, A(6,1))$	$A(5, A(6,2))$	$A(5, A(6,3))$	

Os valores da função de Ackermann crescem muito rapidamente:

- $A(4,2)$ é maior que o número de partículas do universo elevado a potência 200.
- $A(5,2)$ não pode ser escrito como uma expansão decimal no universo físico.
- Além da linha 4 e coluna 1, os valores só podem ser expressos usando a própria notação da função.

Função recursiva que não é bem definida

Seja a função $G : \mathbb{Z}^+ \rightarrow \mathbb{Z}$. Para todos inteiros $n \geq 1$:

$$G(n) = \begin{cases} 1 & \text{se } n = 1, \\ 1 + G(\frac{n}{2}) & \text{se } n \text{ é par}, \\ G(3n-1) & \text{se } n \text{ é ímpar e } n > 1. \end{cases}$$

A função G é bem definida? Não!

$$\begin{aligned} G(1) &= 1 \\ G(2) &= 1 + G(1) = 1 + 1 = 2 \\ G(3) &= G(8) = 1 + G(4) = 1 + (1 + G(2)) \\ &= 1 + (1 + 2) = 4 \\ G(4) &= 1 + G(2) = 1 + 2 = 3 \\ G(5) &= G(14) = 1 + G(7) = 1 + G(20) \\ &= 1 + (1 + G(10)) \\ &= 1 + (1 + (1 + G(5))) = 3 + G(5) \end{aligned}$$

Função recursiva que não sabe se é bem definida

Seja a função $H : \mathbb{Z}^+ \rightarrow \mathbb{Z}$. Para todos inteiros $n \geq 1$:

$$H(n) = \begin{cases} 1 & \text{se } n = 1, \\ 1 + H(\frac{n}{2}) & \text{se } n \text{ é par,} \\ H(3n+1) & \text{se } n \text{ é ímpar e } n > 1. \end{cases}$$

A função H é bem definida? Não se sabe!

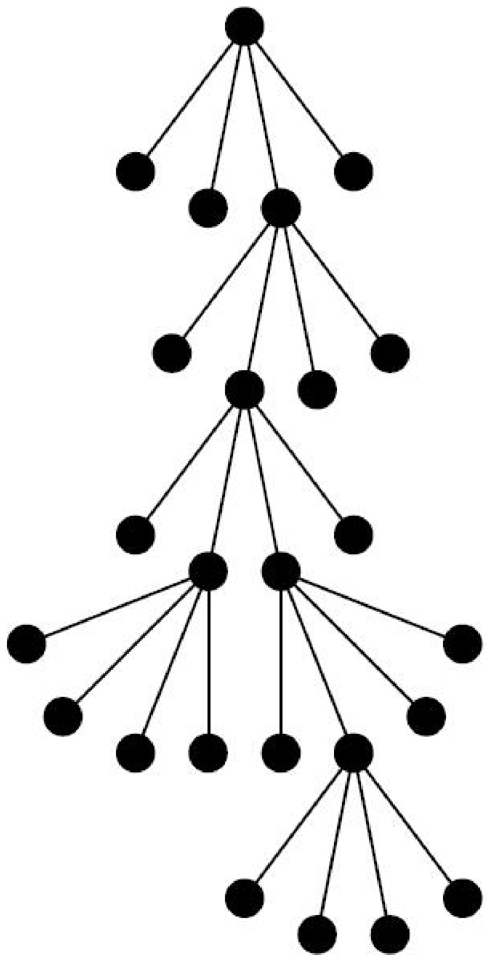
A função é computável para todos inteiros n, $1 \leq n < 10^9$.

Recursividade
Comentários finais

- Técnica bastante adequada para expressar algoritmos que são definidos recursivamente.
- No entanto, deve ser usada com muito cuidado.
- Na maior parte dos casos funciona como uma *técnica conceitual* ao invés de uma *técnica computacional*.
- Algoritmos recursivos são normalmente modelados por uma equação de recorrência.
- Ao se fazer a análise de um algoritmo recursivo, deve-se também analisar o crescimento da pilha.

Algoritmos tentativa e erro (*Backtracking*)

- **Tentativa e erro**: decompor o processo em um número finito de sub-tarefas parciais que devem ser exploradas exaustivamente.
- O processo de tentativa gradualmente constrói e percorre uma árvore de sub-tarefas.
- Algoritmos tentativa e erro não seguem uma regra fixa de computação:
  - Passos em direção à solução final são tentados e registrados.
  - Caso esses passos tomados não levem à solução final, eles podem ser retirados e apagados do registro.



Algoritmos tentativa e erro (*Backtracking*)

- Quando a pesquisa na árvore de soluções cresce rapidamente é necessário usar **algoritmos aproximados** ou **heurísticas** que não garantem a solução ótima mas são rápidas.

- Algoritmos aproximados:
  - Algoritmos usados normalmente para resolver problemas para os quais não se conhece uma solução polinomial.
  - Devem executar em tempo polinomial dentro de limites "prováveis" de qualidade absoluta ou assintótica.

- Heurística:
  - Algoritmo que tem como objetivo fornecer soluções sem um limite formal de qualidade, em geral avaliado empiricamente em termos de complexidade (média) e qualidade das soluções.
  - É projetada para obter ganho computacional ou simplicidade conceitual, possivelmente ao custo de precisão.

Tentativa e erro: Passeio do cavalo

- Tabuleiro com $n \times n$ posições: cavalo se movimenta segundo regras do xadrez.
- Problema: partindo da posição (x_0, y_0), encontrar, se existir, um passeio do cavalo que visita todos os pontos do tabuleiro uma única vez.

Tenta um próximo movimento:

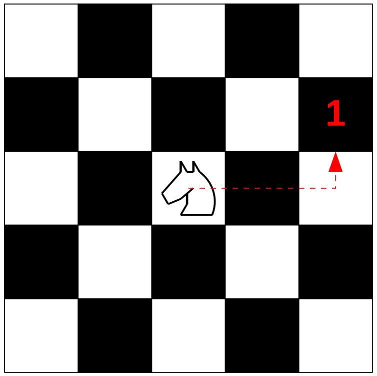
```
TENTA
Inicializa seleção de movimentos
repeat
    Seleciona próximo candidato ao movimento
    if aceitável
      then Registra movimento
            if tabuleiro não está cheio
              then Tenta novo movimento
                    if não é bem sucedido
                      then Apaga registro anterior
until (movimento bem sucedido) ∨ (acabaram-se candidatos ao movimento)
```

Tentativa e erro: Passeio do cavalo

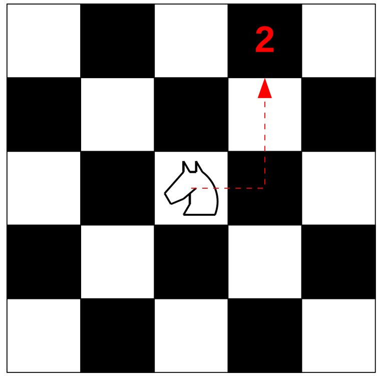
- O tabuleiro pode ser representado por uma matriz $n \times n$.

- A situação de cada posição pode ser representada por um inteiro para recordar o histórico das ocupações:
  - $t[x, y] = 0$, campo $\langle x, y \rangle$ não visitado;
  - $t[x, y] = i$, campo $\langle x, y \rangle$ visitado no i-ésimo movimento, $1 \leq i \leq n^2$.

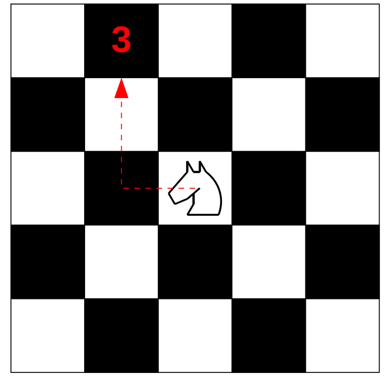
Tentativa e erro: Passeio do cavalo
Regras do xadrez para o movimento do cavalo



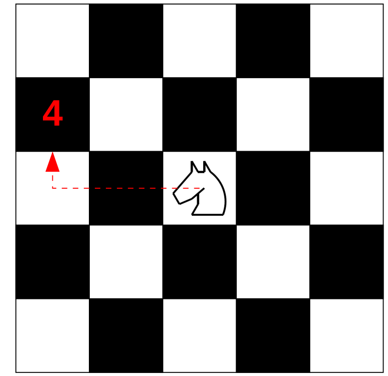
2Dir e 1Cima



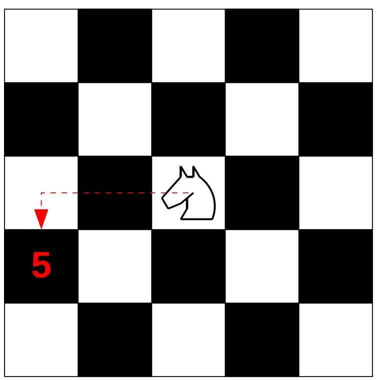
1Dir e 2Cima



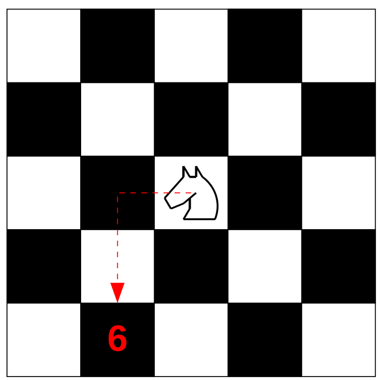
1Esq e 2Cima



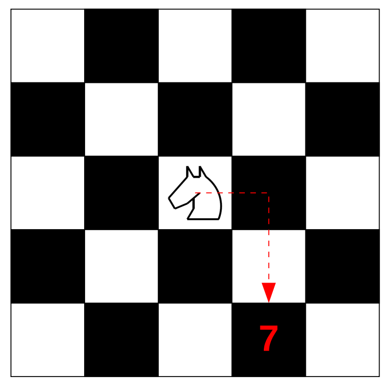
2Esq e 1Cima



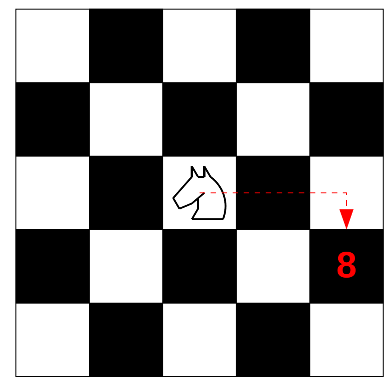
2Esq e 1Baixo



1Esq e 2Baixo



1Dir e 2Baixo



2Dir e 1Baixo

Implementação do passeio do cavalo

PASSEIODOCAVALO(n)

$\triangleright$ Parâmetro: n (tamanho do lado do tabuleiro)
$\triangleright$ Variáveis auxiliares:
i, j $\triangleright$ Contadores
$t[1..n, 1..n]$ $\triangleright$ Tabuleiro de $n \times n$
q $\triangleright$ Indica se achou uma solução
s $\triangleright$ Movimentos identificados por um n°
$h[1..8], v[1..8]$ $\triangleright$ Existem oito movimentos possíveis
$s \leftarrow \{1, 2, 3, 4, 5, 6, 7, 8\}$ $\triangleright$ Conjunto de movimentos
$h[1..8] \leftarrow [2, 1, -1, -2, -2, -1, 1, 2]$ $\triangleright$ Movimentos na horizontal
$v[1..8] \leftarrow [1, 2, 2, 1, -1, -2, -2, -1]$ $\triangleright$ Movimentos na vertical
for $i \leftarrow 1$ **to** n $\triangleright$ Inicializa tabuleiro
  do for $j \leftarrow 1$ **to** n
    do $t[i, j] \leftarrow 0$
$t[1, 1] \leftarrow 1$ $\triangleright$ Escolhe uma casa inicial do tabuleiro
TENTA($2, 1, 1, q$) $\triangleright$ Tenta o passeio usando *backtracking*
if q $\triangleright$ Achou uma solução?
  then print Solução
  else print Não há solução

Implementação do passeio do cavalo

```
Tenta(i, x, y, q)
    ▷ Parâmetros: i (i-ésima casa); x, y (posição no tabuleiro); q (achou solução?)
    ▷ Variáveis auxiliares: xn, yn, m, q1
m ← 0
repeat
    m ← m + 1
    q1 ← false
    xn ← x + h[m]
    yn ← y + v[m]
    if (xn ∈ s) ∧ (yn ∈ s)
      then if t[xn, yn] = 0
               then t[xn, yn] ← i
                    if i < n^2
                      then Tenta(i + 1, xn, yn, q1)
                           if ¬q1
                             then t[xn, yn] ← 0
                      else q1 ← true
until q1 ∨ (m = 8)
q ← q1
```

Algoritmos tentativa e erro (*Backtracking*) Comentários finais

- Técnica usada quando não se sabe exatamente que caminho seguir para encontrar uma solução.

- Não garante a solução ótima.

- Essa técnica pode ser vista ainda como uma variante da recursividade

- Ao se fazer a análise de um algoritmo que usa *backtracking*, deve-se também analisar o crescimento do espaço de soluções.

Divisão e conquista (1)

- Consiste em dividir o problema em partes menores, encontrar soluções para essas partes (supostamente mais fácil), e combiná-las em uma solução global.
  - ➜ Geralmente leva a soluções eficientes e elegantes, principalmente se forem recursivas.

- Basicamente essa técnica consiste das seguintes fases (executadas nesta ordem):
  1. Divisão (particionamento) do problema original em sub-problemas similares ao original mas que são menores em tamanho;
  2. Resolução de cada sub-problema sucessivamente e independentemente (em geral de forma recursiva);
  3. Combinação das soluções individuais em uma solução global para todo o problema.

Divisão e conquista (2)

- Um algoritmo de "divisão e conquista" é normalmente relacionado a uma equação de recorrência que contém termos referentes ao próprio problema.

$$T(n) = aT(\frac{n}{b}) + f(n),$$

  onde a indica o número de sub-problemas gerados, b o tamanho de cada um deles e $f(n)$ o custo para fazer a divisão.

- Paradigma bastante usado em Ciência da Computação em problemas como:
  - Ordenação: Mergesort, Quicksort (Tecnicamente falando, o Quicksort poderia ser chamado de um algoritmo conquista e divisão);
  - Pesquisa: Pesquisa Binária;
  - Algoritmos aritméticos: multiplicação de inteiros, multiplicação de matrizes, FFT (*Fast Fourier Transform*);
  - Algoritmos geométricos: Convex Hull, Par mais próximo;
  - ...

Divisão e conquista: Exemplo 1

- Seja A um vetor de inteiros, $A[1..n]$, $n \geq 1$ que não está ordenado.

- Pede-se:
  - Determine o maior e o menor elementos desse vetor usando divisão e conquista;
  - Determine o custo (número de comparações) para achar esses dois elementos supondo que A possui n elementos.

Divisão e conquista: Exemplo 1

Cada chamada de MaxMin4 atribui às variáveis *Max* e *Min* o maior e o menor elementos em $A[\mathit{Linf}]..A[\mathit{Lsup}]$.

```
MaxMin4(Linf, Lsup, Max, Min)
    ▷ Variáveis auxiliares: Max1, Max2, Min1, Min2, Meio
    if (Lsup − Linf) ≤ 1                                  ▷ Condição da parada recursiva
      then if A[Linf] < A[Lsup]
              then Max ← A[Lsup]
                   Min ← A[Linf]
              else Max ← A[Linf]
                   Min ← A[Lsup]
      else Meio ← ⌊(Linf + Lsup)/2⌋          ▷ Acha o menor e maior elementos de cada partição
           MaxMin4(Linf, Meio, Max1, Min1)
           MaxMin4(Meio+1, Lsup, Max2, Min2)
           if Max1 > Max2
              then Max ← Max1
              else Max ← Max2
           if Min1 < Min2
              then Min ← Min1
              else Min ← Min2
```

Divisão e conquista: Exemplo 1 (Análise)

Seja $f(n)$ o número de comparações entre os elementos de A, que possui n elementos.

$$\begin{array}{ll} f(n) = 1, & \text{para } n \leq 2, \\ f(n) = f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) + 2, & \text{para } n > 2. \end{array}$$

Quando $n = 2^i$ para algum inteiro positivo i, temos que:

$$f(n) = 2f(\tfrac{n}{2}) + 2$$

Divisão e conquista: Exemplo 1 (Análise)

Resolvendo esta equação de recorrência (em função de n e i), temos:

$$\begin{array}{rcl}
f(n) & = & 2f(\frac{n}{2}) + 2 \\
2f(\frac{n}{2}) & = & 2^2 f(\frac{n}{2^2}) + 2^2 \\
2^2 f(\frac{n}{2^2}) & = & 2^3 f(\frac{n}{2^3}) + 2^3 \\
 & \vdots & \\
2^{i-3} f(\frac{n}{2^{i-3}}) & = & 2^{i-2} f(\frac{n}{2^{i-2}}) + 2^{i-2} \\
2^{i-2} f(\frac{n}{2^{i-2}}) & = & 2^{i-1} f(\frac{n}{2^{i-1}}) + 2^{i-1} \\
 & = & 2^{i-1} f(2) + 2^{i-1} \\
 & = & 2^{i-1} + 2^{i-1}
\end{array}$$

$$\begin{array}{rcl}
f(2^i) & = & 2f(2^{i-1}) + 2 \\
2f(2^{i-1}) & = & 2^2 f(2^{i-2}) + 2^2 \\
2^2 f(2^{i-2}) & = & 2^3 f(2^{i-3}) + 2^3 \\
 & \vdots & \\
2^{i-3} f(2^3) & = & 2^{i-2} f(2^2) + 2^{i-2} \\
2^{i-2} f(2^2) & = & 2^{i-1} f(2^1) + 2^{i-1} \\
 & = & 2^{i-1} f(2) + 2^{i-1} \\
 & = & 2^{i-1} + 2^{i-1}
\end{array}$$

Fazendo a expansão desta equação temos:

$$\begin{array}{rcl}
2^{i-2} f(2^2) & = & 2^{i-1} + 2^{i-1} \\
2^{i-3} f(2^3) & = & 2^{i-1} + 2^{i-1} + 2^{i-2} \\
 & \vdots & \\
2^2 f(2^{i-2}) + 2^2 & = & 2^{i-1} + 2^{i-1} + 2^{i-2} + \ldots + 2^3 \\
2f(2^{i-1}) + 2 & = & 2^{i-1} + 2^{i-1} + 2^{i-2} + \ldots + 2^3 + 2^2 \\
f(2^i) & = & 2^{i-1} + 2^{i-1} + 2^{i-2} + \ldots + 2^3 + 2^2 + 2 \\
 & = & 2^{i-1} + \sum_{k=1}^{i-1} 2^k = 2^{i-1} + 2^i - 2 \\
f(n) & = & \frac{n}{2} + n - 2 = \frac{3n}{2} - 2.
\end{array}$$

Logo, $f(n) = 3n/2 - 2$ para o melhor caso, pior caso e caso médio.

Divisão e conquista: Exemplo 1 (Análise)

- Conforme mostrado anteriormente, o algoritmo apresentado neste exemplo é **ótimo**.

- Entretanto, ele pode ser pior do que os já apresentados, pois, a cada chamada recursiva, salva *Linf*, *Lsup*, *Max* e *Min*, além do endereço de retorno da chamada para o procedimento.

- Além disso, uma comparação adicional é necessária a cada chamada recursiva para verificar se *Lsup* $-$ *Linf* ≤ 1 (condição de parada).

- O valor de $n + 1$ deve ser menor do que a metade do maior inteiro que pode ser representado pelo compilador, para não provocar *overflow* na operação *Linf* $+$ *Lsup*.

Divisão e conquista: Exemplo 2

- Motivação:
  - Uma das partes mais importantes da unidade aritmética de um computador é o circuito que soma dois números.
- Pede-se:
  - “Projete” um circuito para somar dois números sem sinal usando divisão e conquista

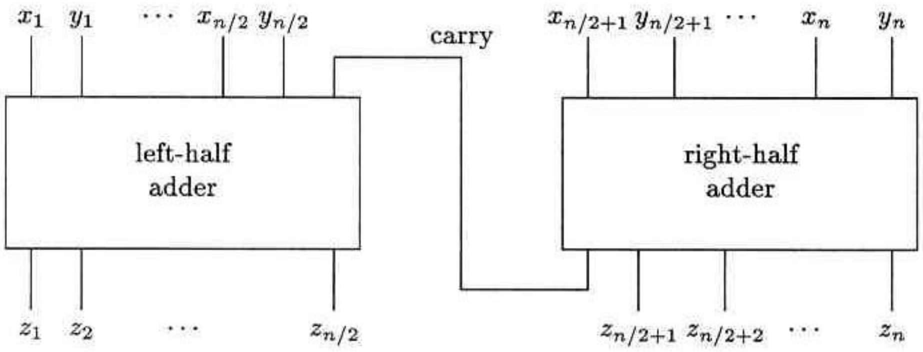
Divisão e conquista: Exemplo 2
Possível solução

- Estratégia para construir um somador de n bits:
  - Usar n somadores de 1-bit.
  - Nesse caso, o atraso (medido pelo caminho mais longo entre a entrada e saída em termos do número de portas lógicas) é $3n$ se for usado o *Ripple-carry Adder*.
  - Exemplo: $n = 32 \Rightarrow$ Atraso = 96.

- Usando a estratégia de divisão e conquista o atraso pode ser menor.

Divisão e conquista: Exemplo 2
Verificando a viabilidade da estratégia DeC

- Dividir os n bits em dois grupos: metade da esquerda e metade da direita.
- Somar cada metade usando circuitos somadores idênticos da metade do tamanho do problema original.
- Questão: A adição da metade da esquerda pode começar antes de terminar a adição da metade da direita?
  ➜ Nessa estratégia não.



Divisão e conquista: Exemplo 2
Verificando a viabilidade da estratégia DeC

- Como começar a computação da esquerda sem conhecer o bit de “vai um” da metade da direita?

- Estratégia:
  - Compute duas somas para a metade da esquerda:
    (a) uma considerando que “vem um” da metade da direita;
    (b) e a outra considerando que não.
  - Uma vez finalizada as somas das duas metades, é possível dizer qual das duas somas da metade da esquerda deve ser utilizada.

Divisão e conquista: Exemplo 2
Estratégia

Sejam as seguintes variáveis para um somador de n bits:

– x_1, x_2, ..., x_n e y_1, y_2, ..., y_n as entradas representando os dois números de n bits a serem somados.

– $s_1, s_2, \ldots, s_n$ a soma de n bits (excluindo o bit de "vai um" mais à esquerda) e considerando que não "veio um" para o bit mais à direita.

– $t_1, t_2, \ldots, t_n$ a soma de n bits (excluindo o bit de "vai um" mais à esquerda) e considerando que "veio um" para o bit mais à direita.

– p, bit propagação de "vai um", que é um 1 se o resultado da soma gera um "vai um" mais à esquerda, assumindo que "veio um" no bit mais à direita.

– g, bit gera "vai um", que é 1 se "vai um" mais à esquerda considerando apenas a soma dos n bits, ou seja, independente se "veio um" no bit mais à direita.

Observe que:

– $g \rightarrow p$, ou seja, se $g = 1$ então $p = 1$.

– No entanto, se $g = 0$ então ainda podemos ter $p = 1$.

Divisão e conquista: Exemplo 2
Calculando os valores desses bits

Duas somas são computadas para a metade da esquerda:

$\swarrow$ **não veio um**

$$
\begin{array}{rrcccc}
 & & x_1 & x_2 & \dots & x_n \\
 & + & y_1 & y_2 & \dots & y_n \\
\hline
\text{Bits } \boxed{\text{p}} \text{ e } \boxed{\text{g}} & \longleftarrow & s_1 & s_2 & \dots & s_n
\end{array}
$$

$\swarrow$ **veio um**

$$
\begin{array}{rrcccc}
 & & x_1 & x_2 & \dots & x_n \\
 & + & y_1 & y_2 & \dots & y_n \\
\hline
\text{Bits } \boxed{\text{p}} \text{ e } \boxed{\text{g}} & \longleftarrow & t_1 & t_2 & \dots & t_n
\end{array}
$$

Divisão e conquista: Exemplo 2
Examinando os valores desses bits quando $n = 1$

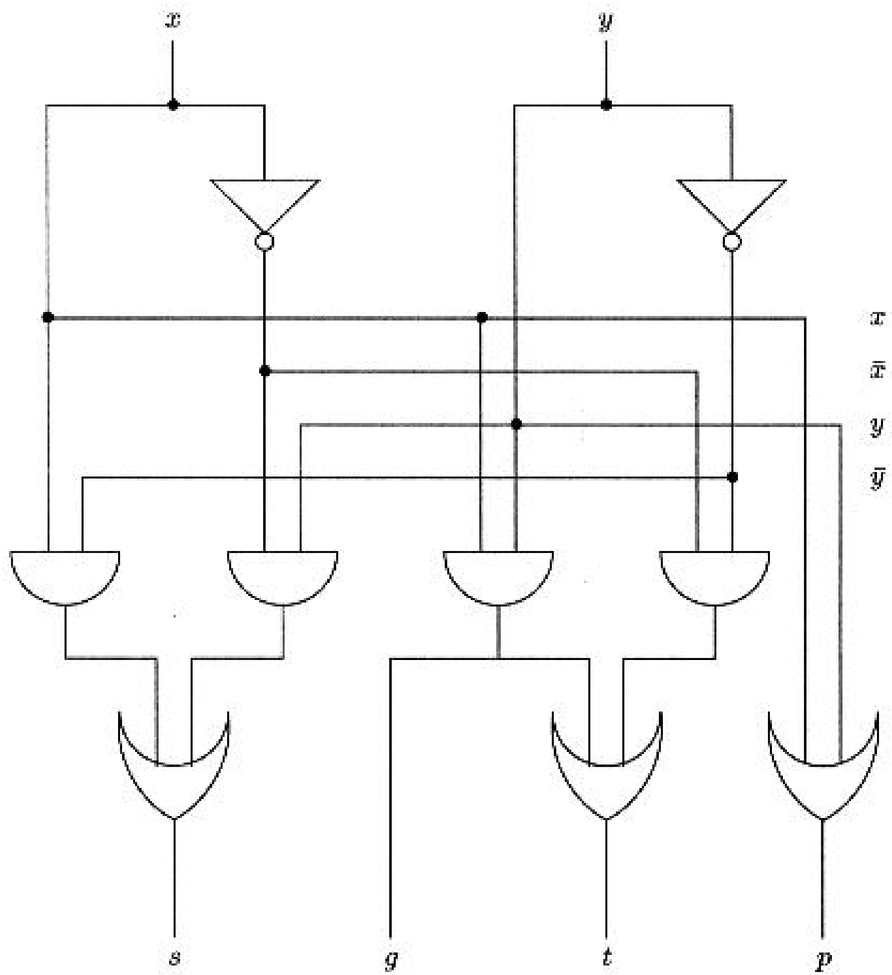
x	y	s	t	p	g
0	0	0	1	0	0
0	1	1	0	1	0
1	0	1	0	1	0
1	1	0	1	1	1

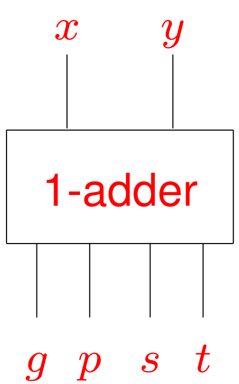
- x e y: entradas a serem somados.
- s: soma de um bit p/ o caso de não “veio um”.
- t: soma de um bit p/ o caso de “veio um”.
- p: bit de propagação de “vai um”, que é um 1 se o resultado da soma gera um “vai um”, p/ o caso de “veio um”.
- g: bit de “vai um”, que é 1 se “vai um” considerando apenas a soma dos n bits.

Expressões correspondentes:

- $s = x\overline{y} + \overline{x}y$
  - A soma s só é 1 quando apenas uma das entradas é 1.
- $t = xy + \overline{x}\,\overline{y}$
  - Assumindo que vem 1, a soma t só será 1 quando as duas entradas forem idênticas.
- $p = x + y$
  - Assumindo que veio 1, também irá 1 quando uma das entradas ou ambas forem 1.
- $g = xy$
  - O bit de vai 1 só será 1 quando as duas entradas forem 1.

Divisão e conquista: Exemplo 2
Somador para o caso $n = 1$





Modelagem:

- Atraso: $D(1) = 3$
- Portas: $G(1) = 9$

Divisão e conquista: Exemplo 2
Idéia para aplicar DeC

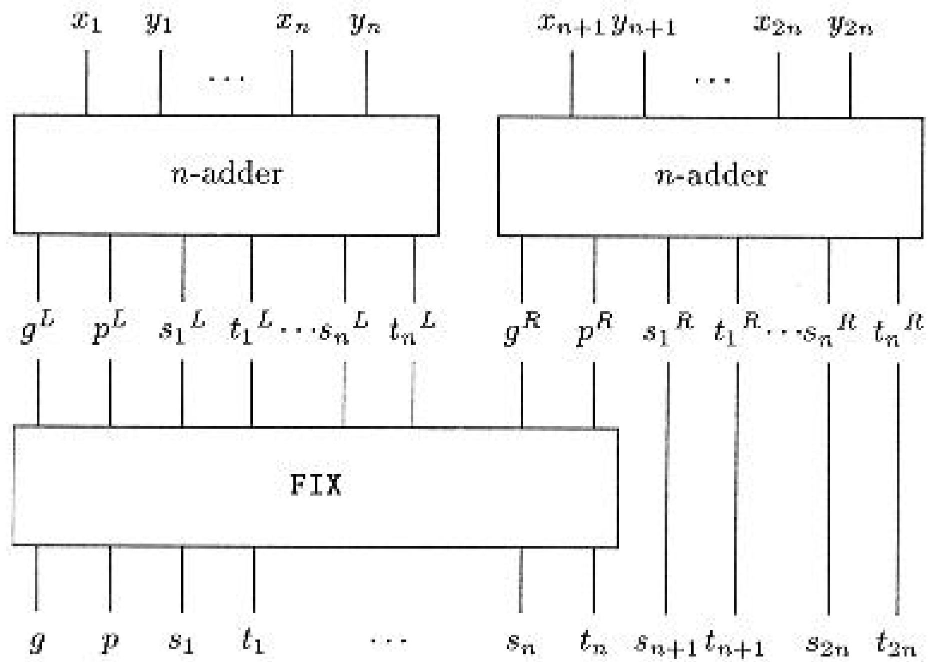
- Idéia:
  – Construir um somador de $2n$ bits usando dois somadores de n bits.

- Computar os bits:
  – propagação de "vai um" (p), e
  – gera "vai um" (g)

  para o somador de $2n$ bits.

- Ajustar a metade da esquerda dos bits s e t para levar em consideração se há um "vai um" para a metade da esquerda vindo da metade da direita.

- Circuito que implementa a idéia:



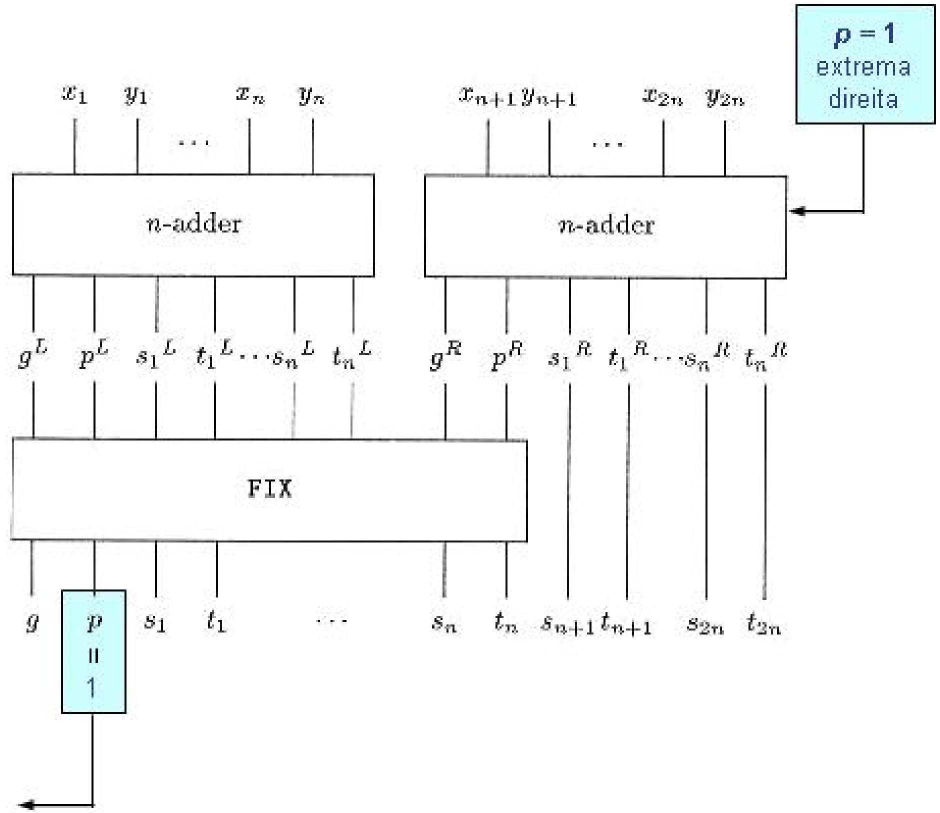
Divisão e conquista: Exemplo 2
Cálculo de p

Suponha que <u>há</u> um "veio um" para o circuito de $2n$ bits (extrema direita). <u>Haverá</u> um "vai um" (extrema esquerda), representado pelo bit de propagação p = 1, se:

- A metade da esquerda gera um "vai um", ou seja, g^L, já que $g^L \rightarrow p^L$.
- As duas metades do somador propagam o "vai um", ou seja, $p^L p^R$. Esta expressão inclui o caso tcblue$p^L g^R$. Como $g^R \rightarrow p^R$, temos que ($p^L p^R$ + $p^L g^R$) $\equiv$ $p^L p^R$.

➜ A expressão para p, bit de propagação de "vai um", é:

$$p = g^L + p^L p^R$$



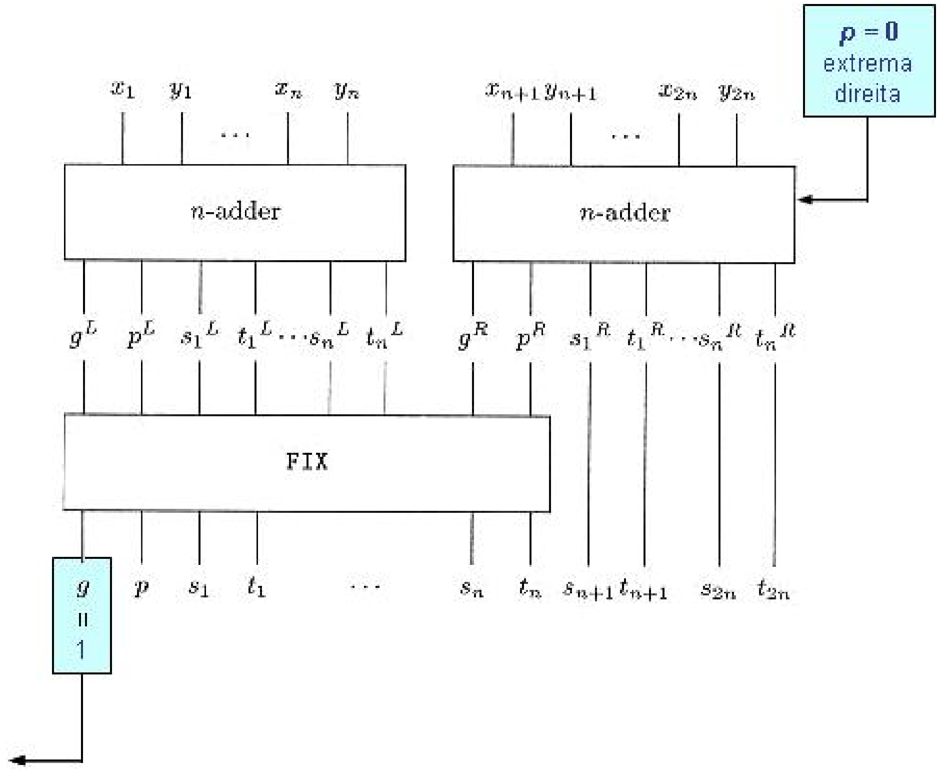
Divisão e conquista: Exemplo 2
Cálculo de g

Suponha que não há um "veio um" para o circuito de $2n$ bits (extrema direita). Haverá um "vai um" (extrema esquerda), ou seja, o bit de gera "vai um" g vale 1 se:

- A metade da esquerda gera um "vai um", ou seja, g^L.
- A metade da direita gera um "vai um" e a metade da esquerda propaga esse bit, ou seja, $p^L g^R$.

➜ A expressão para g, bit gera "vai um", é:

$$g = g^L + p^L g^R$$



Divisão e conquista: Exemplo 2
Cálculo dos bits s e t da direita

- Calculando os bits

  $$s_{n+1}, s_{n+2}, \ldots, s_{2n}$$

  e

  $$t_{n+1}, t_{n+2}, \ldots, t_{2n}$$

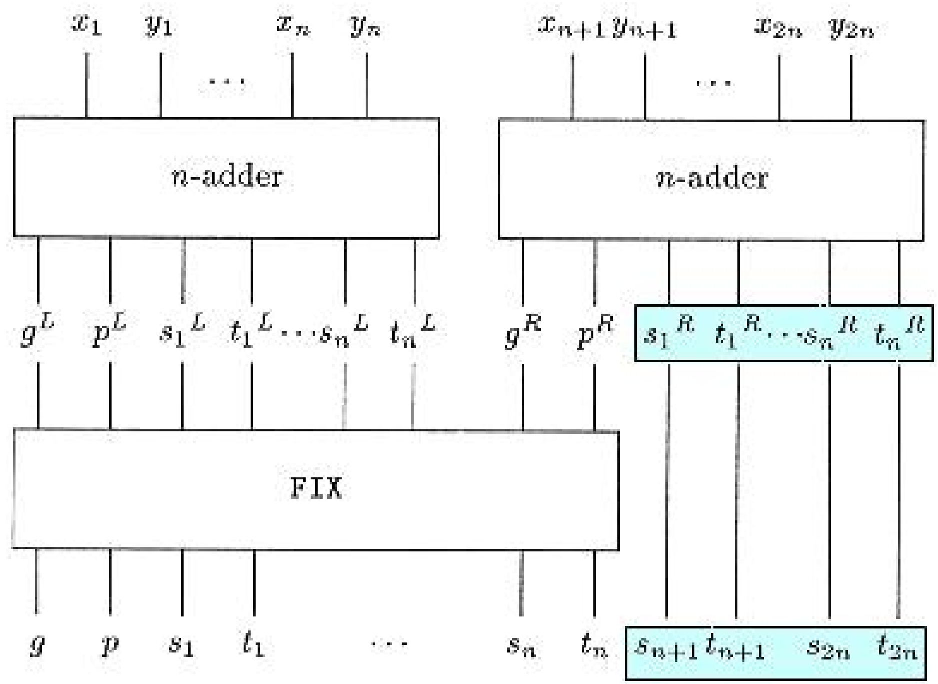
- Bits da direita não são modificados. Assim,

  $$s_{n+i} = s_i^R$$

  $$t_{n+i} = t_i^R$$

  para $i = 1, 2, \ldots, n$.

  Observação: num somador de $2n$ bits, as saídas são identificadas pelos índices $1, 2, \ldots, 2n$ numerados a partir da esquerda. Logo, os índices $n + 1, n + 2, \ldots, 2n$ correspondem à metade da direita.

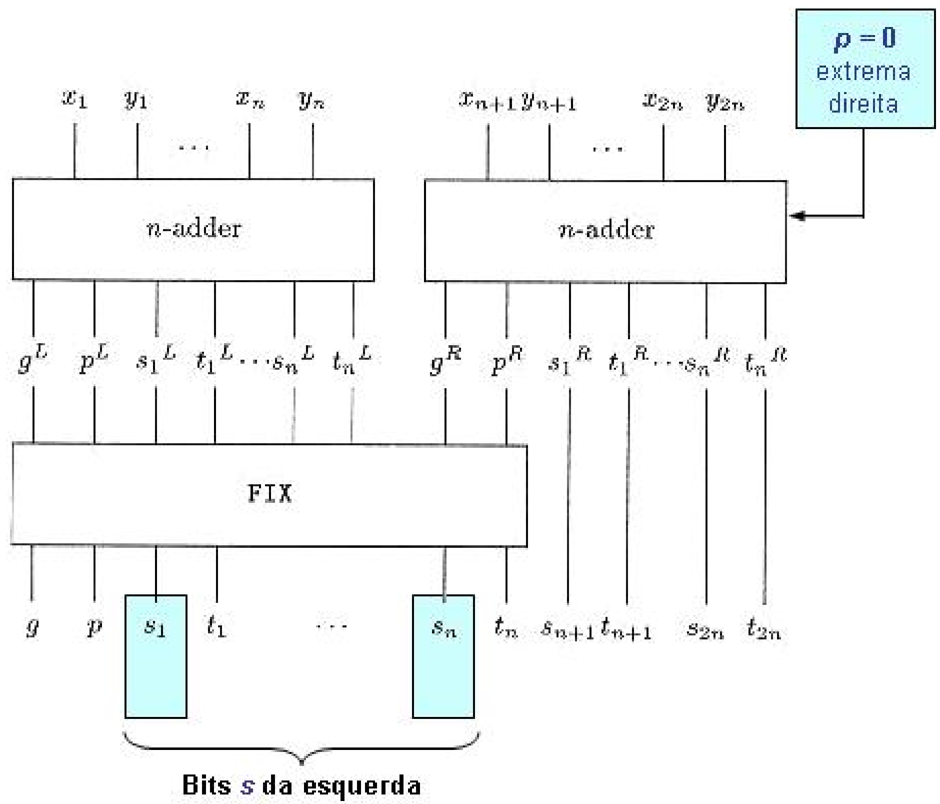


Divisão e conquista: Exemplo 2
Cálculo dos bits s da esquerda

- Suponha que <u>não há</u> um "veio um" (extrema direita) para o circuito de $2n$ bits.
- Neste caso, o "vai um" para a metade da esquerda, se existir, foi gerado pela metade da direita. Assim, se:
  - $g^R = 1 \Rightarrow s_i = t_i^L$
  - $g^R = 0 \Rightarrow s_i = s_i^L$
- A expressão para s_i é:

$$s_i = s_i^L\,\overline{g}^R + t_i^L\,g^R$$

para $i = 1, 2, \ldots, n$.



Divisão e conquista: Exemplo 2
Cálculo dos bits t da esquerda

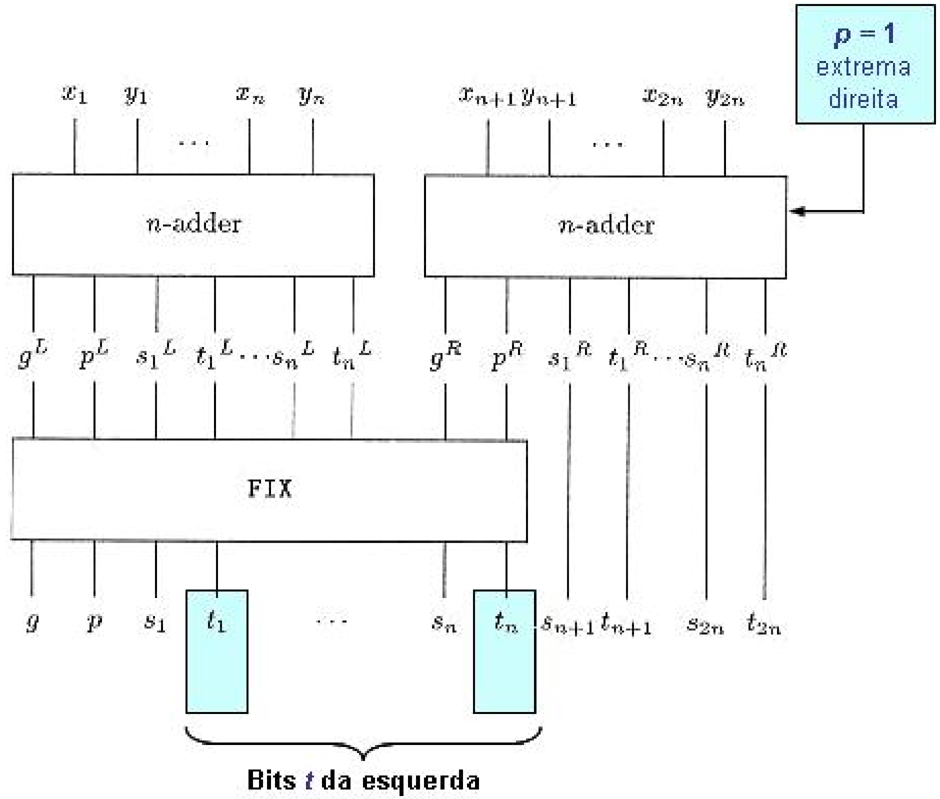
- Suponha que há um "veio um" (extrema direita) para o circuito de $2n$ bits.

- Neste caso, devemos analisar o bit de propagação p. Assim, se:
  - p^R = 1 $\Rightarrow t_i = t_i^L$
  - p^R = 0 $\Rightarrow t_i = s_i^L$

- A expressão para t_i é:

$$t_i = s_i^L\,\overline{p}^R + t_i^L\,p^R$$

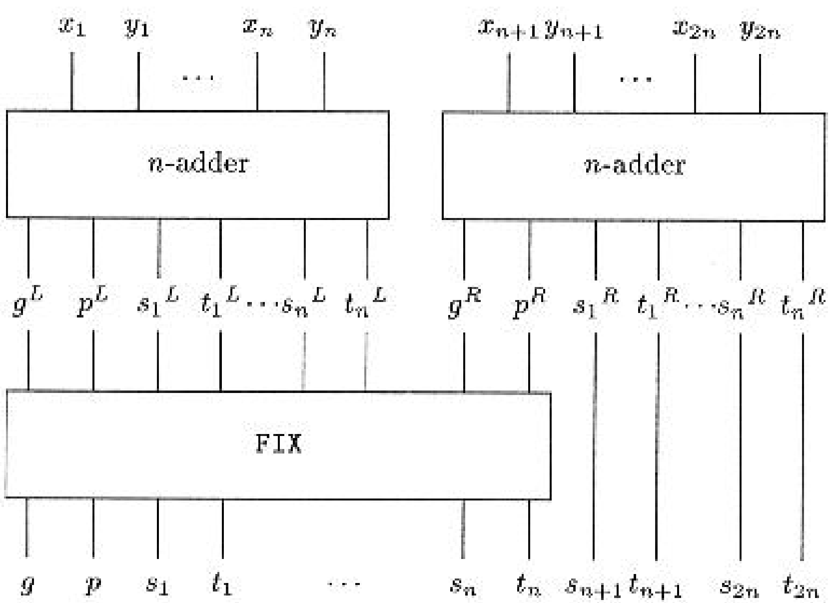
para $i = 1, 2, \ldots, n$.



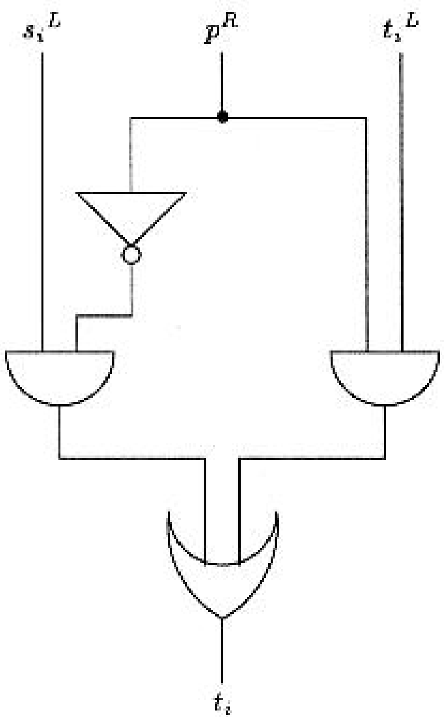
Divisão e conquista: Exemplo 2
Expressões a serem calculadas pelo `FIX`

O módulo `FIX` deve calcular as seguintes expressões:

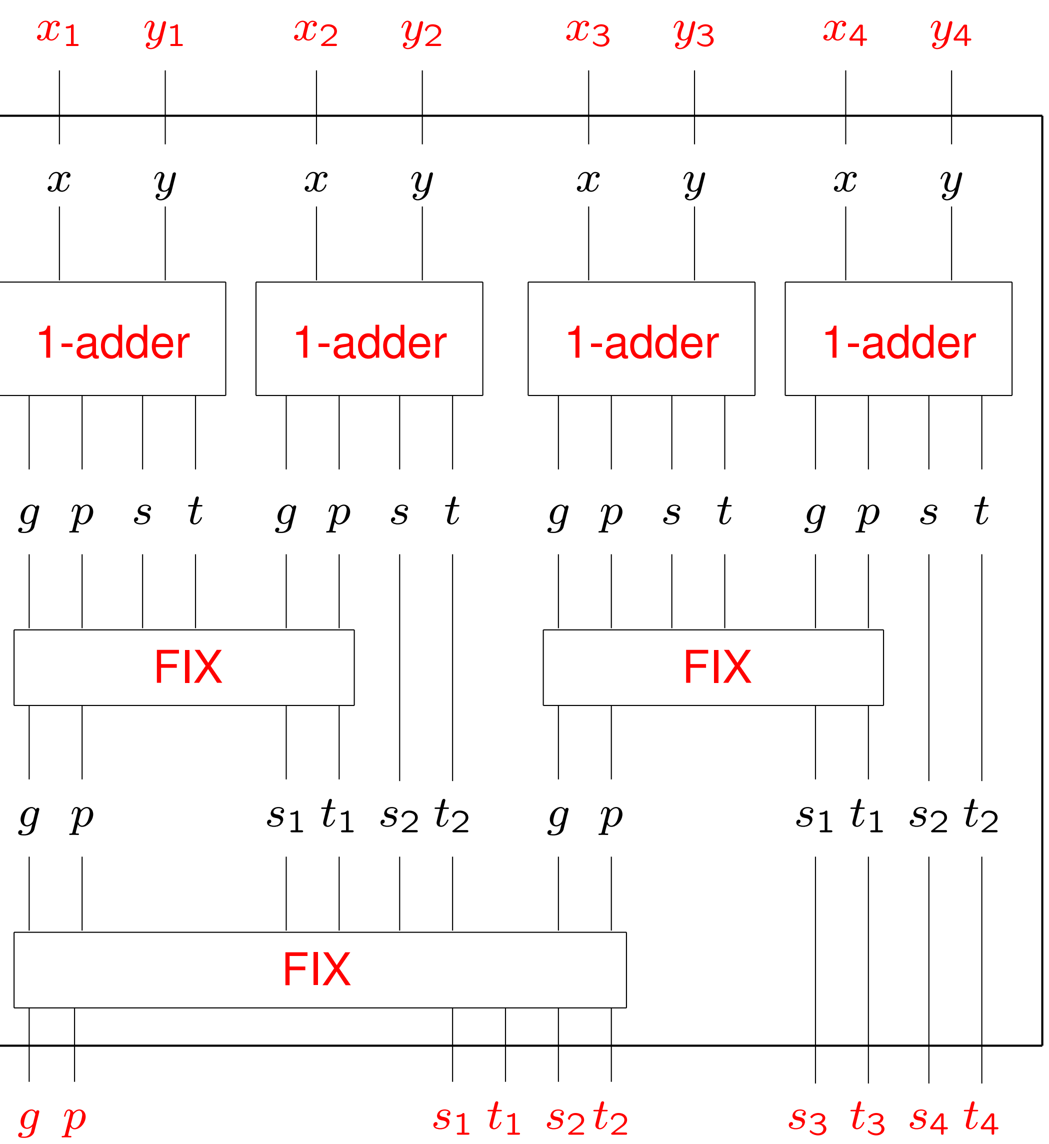
$$
\begin{aligned}
p &= g^L + p^L p^R \\
g &= g^L + p^L g^R \\
s_i &= s_i^L\,\overline{g}^R + t_i^L\,g^R, \qquad i = 1, 2, \ldots, n \\
t_i &= s_i^L\,\overline{p}^R + t_i^L\,p^R, \qquad i = 1, 2, \ldots, n
\end{aligned}
$$



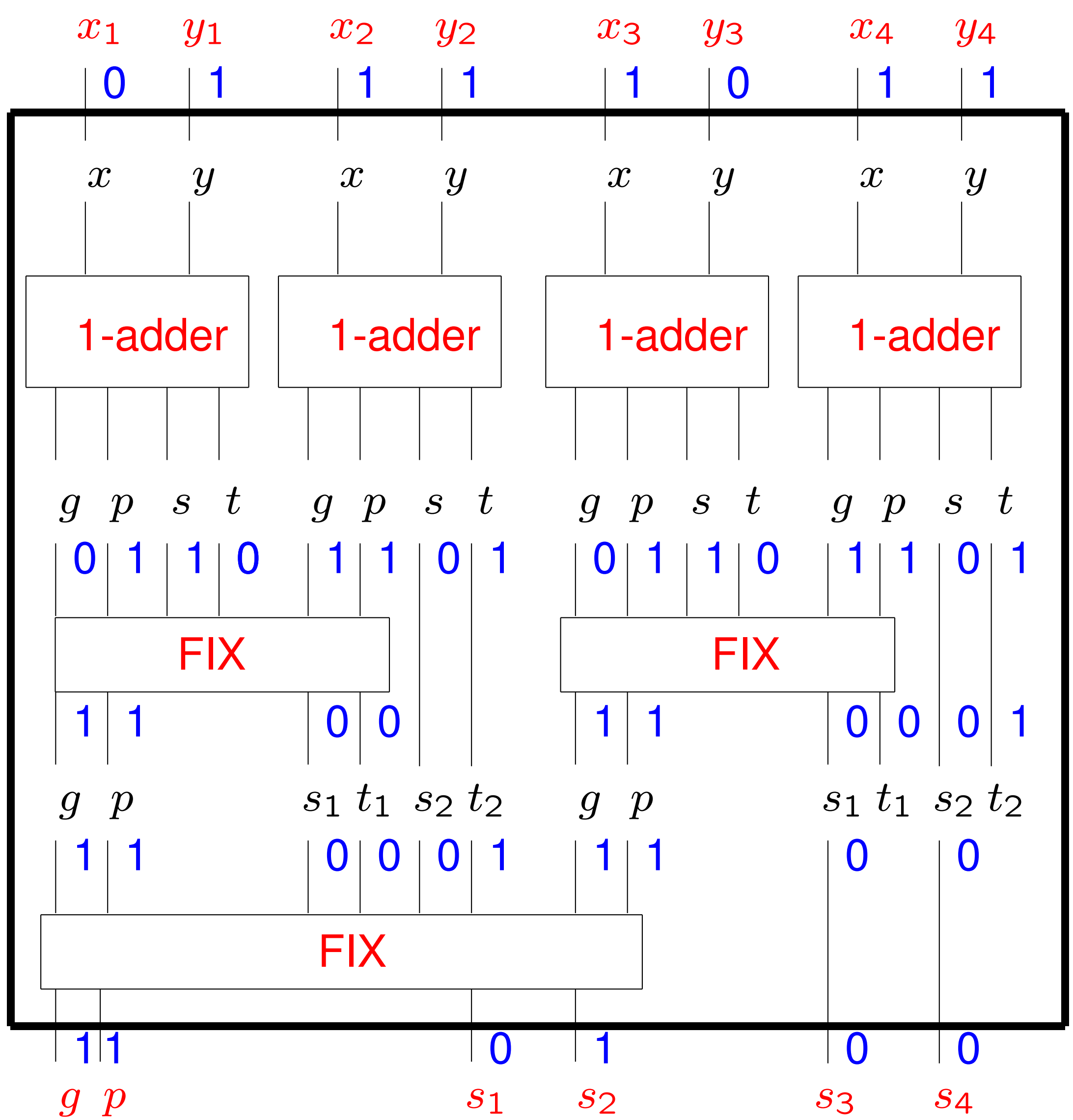
Essas expressões podem ser calculadas por circuitos de no máximo três níveis. O exemplo abaixo é para t_i:



Divisão e conquista: Exemplo 2
Somador para $n = 4$ (Caso genérico)



Divisão e conquista: Exemplo 2
Somador para $n = 4$ (Caso específico)



Divisão e conquista: Exemplo 2
Cálculo do atraso usando DeC

$$\text{Atraso: } \begin{cases} D(1) &= 3 \\ D(2n) &= D(n) + 3 \\ D(n) &= 3(1 + \log n) = O(\log n) \end{cases}$$

Para um somador de 32 bits:

– Divisão e conquista: $3(1 + \log n) = 3(1 + \log 32) = 18$

– *Ripple-carry*: $3n = 96$

Divisão e conquista: Exemplo 2
Comentários sobre este exemplo

- Solução usando divisão e conquista (DeC):
  - Atraso: $O(\log n)$
  - Nº de portas: $O(n \log n)$

- Solução *Ripple-Carry Adder*:
  - Atraso: $O(n)$
  - Nº de portas: $O(n)$

- A solução DeC apresenta um exemplo onde o aumento do espaço (neste caso portas) possibilita uma diminuição no atraso (tempo, neste caso), ou seja, existe um compromisso TEMPO $\times$ ESPAÇO.

- A solução apresentada é um exemplo "não tradicional" da técnica DeC já que o sub-problema da esquerda deve gerar duas soluções, uma vez que seu valor depende da solução do sub-problema da direita, ou seja, os sub-problemas não são independentes.

Divisão-e-conquista: Alguns comentários

- Este paradigma não é aplicado apenas a problemas recursivos.

- Existem pelo menos três cenários onde divisão e conquista é aplicado:
  1. Processar independentemente partes do conjunto de dados.
     – Exemplo: Mergesort.
  2. Eliminar partes do conjunto de dados a serem examinados.
     – Exemplo: Pesquisa binária.
  3. Processar separadamente partes do conjunto de dados mas onde a solução de uma parte influencia no resultado da outra.
     – Exemplo: Somador apresentado.

Balanceamento

- No projeto de algoritmos, é importante procurar sempre manter o **balanceamento** na sub-divisão de um problema em partes menores.

- Divisão e conquista não é a única técnica em que balanceamento é útil.

- Considere o seguinte exemplo de ordenação:

EXEMPLO_DE_ORDENAÇÃO(n)

for $i = 1..n-1$ **do**
    Selecione o menor elemento de $A[i..n]$ e troque-o com o elemento $A[i]$.

  - Inicialmente o menor elemento de $A[1..n]$ é trocado com o elemento $A[1]$.

  - O processo é repetido para as seqüências $n-1$, $n-2$, ..., 2, com os $n-1$, $n-2$, ..., 2 elementos, respectivamente, sendo que em cada passo o menor elemento é trocado com o elemento $A[i]$.

Balanceamento: Análise do exemplo

O algoritmo leva à equação de recorrência:

$$
\begin{aligned}
T(n) &= T(n-1) + n - 1 \\
T(1) &= 0
\end{aligned}
$$

para o número de comparações entre elementos.

Substituindo:

$$
\begin{aligned}
T(n) &= T(n-1) + n - 1 \\
T(n-1) &= T(n-2) + n - 2 \\
&\vdots \\
T(2) &= T(1) + 1
\end{aligned}
$$

e adicionando lado a lado, obtemos:

$$T(n) = T(1) + 1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2}.$$

Logo, o algoritmo é $O(n^2)$.

Balanceamento: Análise do exemplo

- Embora o algoritmo possa ser visto como uma aplicação recursiva de divisão e conquista, ele não é eficiente para valores grandes de n.

- Para obter eficiência assintótica é necessário fazer um **balanceamento**:
  - Dividir o problema original em dois sub-problemas de tamanhos aproximadamente iguais, ao invés de um de tamanho 1 e o outro de tamanho $n - 1$.

- Comentário:
  - A análise da equação de recorrência nos mostra a razão do comportamento quadrático desse algoritmo.
  - É essa equação também que "sugere" como o algoritmo pode ter um desempenho bem melhor, se um balanceamento for usado.

Exemplo de balanceamento: Mergesort

- **Intercalação**:
  - Unir dois arquivos ordenados gerando um terceiro arquivo ordenado (*merge*).
- Colocar no terceiro arquivo o menor elemento entre os menores dos dois arquivos iniciais, desconsiderando este mesmo elemento nos passos posteriores.
- Este processo deve ser repetido até que todos os elementos dos arquivos de entrada sejam escolhidos.

Algoritmo de ordenação: Mergesort

1. Divida recursivamente o vetor a ser ordenado em dois, até obter n vetores de um único elemento.

2. Aplique a intercalação tendo como entrada dois vetores de um elemento, formando um vetor ordenado de dois elementos.

3. Repita este processo formando vetores ordenados cada vez maiores até que todo o vetor esteja ordenado.

Exemplo de balanceamento: Implementação do Mergesort

MERGESORT(A, i, j)
  ▷ Parâmetros: A (vetor); i, j (limites inferior e superior da partição)
  ▷ Variável auxiliar: m (meio da partição)
if $i < j$
  then $m \leftarrow \lfloor \frac{(i+j)}{2} \rfloor$
    MERGESORT(A, i, m)
    MERGESORT($A, m + 1, j$)
    MERGE(A, i, m, j)

Considere n como sendo uma potência de 2.

Merge(A, i, m, j) recebe duas seqüências ordenadas $A[i..m]$ e $A[(m + 1)..j]$ e produz uma outra seqüência ordenada dos elementos de $A[i..m]$ e $A[m + 1..j]$.

Como $A[i..m]$ e $A[m + 1..j]$ estão ordenados, *Merge* requer no máximo $n - 1$ comparações.

Merge seleciona repetidamente o menor dentre os menores elementos restantes em $A[i..m]$ e $A[m + 1..j]$. Caso empate, retira de qualquer uma delas.

Análise do Mergesort

Na contagem de comparações, o comportamento do Mergesort pode ser representado por:

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + n - 1, \\ T(1) &= 0. \end{aligned}$$

No caso dessa equação de recorrência sabemos que o custo é (veja a resolução desta equação na parte de indução ou usando o Teorema Mestre):

$$T(n) = n \log n - n + 1.$$

Logo, o algoritmo é $O(n \log n)$.

Balanceamento:
Alguns comentários

- Para o problema de ordenação, o balanceamento levou a um resultado muito superior:
  - O custo passou de $O(n^2)$ para $O(n \log n)$.

- Balanceamento é uma técnica presente em diferentes aspectos algorítmicos de Ciência da Computação como Sistemas Operacionais.

- Também é uma técnica importante quando o modelo computacional usado é o PRAM (*Parallel Random Access Machine*).

Programação dinâmica

- Programação não está relacionado com um programa de computador.
  - A palavra está relacionada com um método de solução baseado em tabela.

- Programação dinâmica (PD) $\times$ Divisão-e-conquista (DeC):
  - DeC quebra o problema em sub-problemas menores.
  - PD resolve todos os sub-problemas menores mas somente reusa as soluções ótimas.

Programação dinâmica

- Quando

$$\sum \textit{Tamanhos dos sub-problemas} = O(n)$$

é provável que o algoritmo recursivo tenha **complexidade polinomial**.

- Quando a divisão de um problema de tamanho n resulta em

$$n \textit{ Sub-problemas} \times \textit{Tamanho } n - 1 \textit{ cada um}$$

é provável que o algoritmo recursivo tenha **complexidade exponencial**.

- Nesse caso, a técnica de programação dinâmica pode levar a um algoritmo mais eficiente.

- A programação dinâmica calcula a solução para todos os sub-problemas, partindo dos sub-problemas menores para os maiores, armazenando os resultados em uma tabela.

- A vantagem é que uma vez que um sub-problema é resolvido, a resposta é armazenada em uma tabela e nunca mais é recalculado.

Programação dinâmica: Produto de n matrizes

- Seja

$$M = M_1 \times M_2 \times \cdots \times M_n,$$

onde M_i é uma matriz com d_{i-1} linhas e d_i colunas, $2 \leq i \leq n$.

➜ Isto server para dizer apenas que a matriz M_i possui uma quantidade de linhas igual a quantidade de colunas de M_{i-1} (d_{i-1}) e uma quantidade de colunas dada por d_i.

- A ordem da multiplicação pode ter um efeito enorme no número total de operações de adição e multiplicação necessárias para obter M.

- Considere o produto de uma matriz $p \times q$ por outra matriz $q \times r$ cujo algoritmo requer $O(pqr)$ operações.

- Considere o produto

$$M = M_1[10, 20] \times M_2[20, 50] \times M_3[50, 1] \times M_4[1, 100],$$

onde as dimensões de cada matriz aparecem entre colchetes.

Programação dinâmica: Produto de n matrizes

Sejam duas possíveis ordens de avaliação dessa multiplicação:

$$M = M_1[10, 20] \times M_2[20, 50] \times M_3[50, 1] \times M_4[1, 100],$$

$M = M_1 \times (M_2 \times \underbrace{(M_3 \times M_4)}_{50 \times 1 \times 100 = 5\,000 \text{ operações}})$	$M = (M_1 \times \underbrace{(M_2 \times M_3)}_{20 \times 50 \times 1 = 1\,000 \text{ operações}}) \times M_4$
$M = M_1 \times \underbrace{(M_2 \times M_a)}_{20 \times 50 \times 100 = 100\,000 \text{ operações}}$, sendo $M_a[50, 100]$	$M = \underbrace{(M_1 \times M_a)}_{10 \times 20 \times 1 = 200 \text{ operações}} \times M_4$, sendo $M_a[20, 1]$
$M = \underbrace{M_1 \times M_b}_{10 \times 20 \times 100 = 20\,000 \text{ operações}}$, sendo $M_b[20, 100]$	$M = \underbrace{M_b \times M_4}_{10 \times 1 \times 100 = 1\,000 \text{ operações}}$, sendo $M_b[10, 1]$
Total $= 125\,000$ operações	Total $= 2\,200$ operações

- Tentar todas as ordens possíveis para minimizar o número de operações $f(n)$ é exponencial em n, onde $f(n) \geq 2^{n-2}$.
- Usando programação dinâmica é possível obter um algoritmo $O(n^3)$.

Programação dinâmica: Produto de n matrizes

Seja m_{ij} o menor custo para computar

$$M_i \times M_{i+1} \times \cdots \times M_j, \text{ para } 1 \leq i \leq j \leq n.$$

Neste caso,

$$m_{ij} = \begin{cases} 0, & \text{se } i = j, \\ \text{Min}_{i \leq k < j} \ (m_{ik} + m_{k+1,j} + d_{i-1} d_k d_j), & \text{se } j > i. \end{cases}$$

- m_{ik} representa o custo mínimo para calcular $M' = M_i \times M_{i+1} \times \cdots \times M_k$.
- $m_{k+1,j}$ representa o custo mínimo para calcular $M'' = M_{k+1} \times M_{k+2} \times \cdots \times M_j$.
- $d_{i-1} d_k d_j$ representa o custo de multiplicar $M'[d_{i-1}, d_k]$ por $M''[d_k, d_j]$.
- $m_{ij}, \ j > i$ representa o custo mínimo de todos os valores possíveis de k entre i e $j - 1$, da soma dos três termos.

Programação dinâmica: Exemplo

- A solução usando programação dinâmica calcula os valores de m_{ij} na ordem crescente das diferenças nos subscritos.

- O cálculo inicia com m_{ii} para todo i, depois $m_{i,i+1}$ para todo i, depois $m_{i,i+2}$, e assim sucessivamente.

- Desta forma, os valores m_{ik} e $m_{k+1,j}$ estarão disponíveis no momento de calcular m_{ij}.

- Isto acontece porque $j - i$ tem que ser estritamente maior do que ambos os valores de $k - i$ e $j - (k + 1)$ se k estiver no intervalo $i \leq k < j$.

Programação dinâmica: Implementação

```
AvaliaMultMatrizes(n, d[0..n])
    ▷ Parâmetro: n (nº de matrizes); d[0..n] (dimensões das matrizes)
    ▷ Constante e variáveis auxiliares:
      MaxInt = maior inteiro
      i, j, k, h, n, temp
      m[1..n, 1..n]
for i ← 1 to n
    do m[i, i] ← 0
for h ← 1 to n − 1
    do for i ← 1 to n − h
        do j ← i + h
            m[i, j] ← MaxInt
            for k ← i to j − 1 do
                temp ← m[i, k] + m[k + 1, j] + d[i − 1] × d[k] × d[j]
                if temp < m[i, j]
                    then m[i, j] ← temp
print m
```

➜ A execução de AvaliaMultMatrizes obtém o custo mínimo para multiplicar as n matrizes, assumindo que são necessárias pqr operações para multiplicar uma matriz $p \times q$ por outra matriz $q \times r$.

Programação dinâmica: Implementação

A multiplicação de

$$M = M_1[10, 20] \times M_2[20, 50] \times M_3[50, 1] \times M_4[1, 100],$$

sendo

d	10	20	50	1	100
	0	1	2	3	4

produz como resultado

$m_{11} = 0$	$m_{22} = 0$	$m_{33} = 0$	$m_{44} = 0$
$m_{12} = 10.000$ $M_1 \times M_2$	$m_{23} = 1.000$ $M_2 \times M_3$	$m_{34} = 5.000$ $M_3 \times M_4$	
$m_{13} = 1.200$ $M_1 \times (M_2 \times M_3)$	$m_{24} = 3.000$ $(M_2 \times M_3) \times M_4$		
$m_{14} = 2.200$ $(M_1 \times (M_2 \times M_3)) \times M_4$			

Programação dinâmica: Princípio da otimalidade

- A ordem de multiplicação pode ser obtida registrando o valor de k para cada entrada da tabela que resultou no mínimo.

- Essa solução eficiente está baseada no **princípio da otimalidade**:
  - Em uma seqüência ótima de escolhas ou de decisões cada sub-seqüência deve também ser ótima.

- Cada sub-seqüência representa o custo mínimo, assim como m_{ij}, $j > i$.

- Assim, todos os valores da tabela representam escolhas ótimas.

Aplicação do princípio da otimalidade

- O princípio da otimalidade não pode ser aplicado indiscriminadamente.

- Se o princípio não se aplica é provável que não se possa resolver o problema com sucesso por meio de programação dinâmica.
  - Quando, por exemplo, o problema utiliza recursos limitados e o total de recursos usados nas sub-instâncias é maior do que os recursos disponíveis.

- Exemplo do princípio da otimalidade: suponha que o caminho mais curto entre Belo Horizonte e Curitiba passa por Campinas. Logo,
  - o caminho entre BH e Campinas também é o mais curto possível;
  - como também é o caminho entre Campinas e Curitiba;
  - ➜ Logo, o princípio da otimalidade se aplica.

Não aplicação do princípio da otimalidade

Seja o problema de encontrar o caminho mais longo entre duas cidades. Temos que:

- Um caminho simples nunca visita uma mesma cidade duas vezes.
- Se o caminho mais longo entre Belo Horizonte e Curitiba passa por Campinas, isso não significa que o caminho possa ser obtido tomando o caminho simples mais longo entre Belo Horizonte e Campinas e depois o caminho simples mais longo entre Campinas e Curitiba.
  - ➜ Observe que o caminho simples mais longo entre BH e Campinas pode passar por Curitiba!
- Quando os dois caminhos simples são agrupados não existe uma garantia que o caminho resultante também seja simples.
- Logo, o princípio da otimalidade não se aplica.

Quando aplicar PD?

- Problema computacional deve ter uma formulação recursiva.
- Não deve haver ciclos na formulação (usualmente o problema deve ser reduzido a problemas menores).
- Número total de instâncias do problema a ser resolvido deve ser pequeno (n).
- Tempo de execução é $O(n) \times$ tempo para resolver a recursão.
- PD apresenta sub-estrutura ótima:
  - Solução ótima para o problema contém soluções ótimas para os sub-problemas.
- Sobreposição de sub-problemas:
  - Número total de sub-problemas distintos é pequeno comparado com o tempo de execução recursivo.

Algoritmos gulosos

- Aplicado a problemas de otimização.
- Seja o algoritmo para encontrar o caminho mais curto entre dois vértices de um grafo:
  - Escolhe a aresta que parece mais promissora em qualquer instante.
- Assim,
  - ➜ independente do que possa acontecer mais tarde, nunca reconsidera a decisão.
  - ➜ não necessita avaliar alternativas, ou usar procedimentos sofisticados para desfazer decisões tomadas previamente.

Características dos algoritmos gulosos Problema geral

- Dado um conjunto C, determine um sub-conjunto $S \subseteq C$ tal que:
  - S satisfaz uma dada propriedade P, e
  - S é mínimo (ou máximo) em relação a algum critério α.

- O **algoritmo guloso** para resolver o problema geral consiste em um processo iterativo em que S é construído adicionando-se ao mesmo elementos de C um a um.

Características dos algoritmos gulosos

- Para construir a solução ótima existe um conjunto ou lista de candidatos.
- São acumulados um conjunto de candidatos considerados e escolhidos, e o outro de candidatos considerados e rejeitados.
- Existe uma função que verifica se um conjunto particular de candidatos produz uma *solução* (sem considerar otimalidade no momento).
- Outra função verifica se um conjunto de candidatos é *viável* (também sem preocupar com a otimalidade).
- Uma *função de seleção* indica a qualquer momento quais dos candidatos restantes é o mais promissor.
- Uma *função objetivo* fornece o valor da solução encontrada, como o comprimento do caminho construído (não aparece de forma explicita no algoritmo guloso).

Estratégia do algoritmo guloso

GULOSO(C)	$\triangleright$ C é o conjunto de candidatos
1 $S \leftarrow \emptyset$	$\triangleright$ S contém conjunto solução, inicialmente vazio
2 **while** $(C \neq \emptyset \wedge \neg$*Solução*$(S))$	$\triangleright$ $\exists$ candidatos e não achou uma solução?
3 **do** $x \leftarrow$ *Seleciona*(C)	$\triangleright$ Seleciona o próximo candidato
4 $\quad C \leftarrow C - x$	$\triangleright$ Remove esse candidato do conjunto C
5 $\quad$ **if** *Viável*$(S + x)$	$\triangleright$ A solução é viável?
6 $\quad\quad$ **then** $S \leftarrow S + x$;	$\triangleright$ Sim, incorpora o candidato à solução
7 **if** *Solução*(S)	$\triangleright$ Obteve uma solução?
8 $\quad$ **then return**(S)	$\triangleright$ Sim, retorna a solução
9 $\quad$ **else return**$(\emptyset)$	$\triangleright$ Não!

- Inicialmente, o conjunto S de candidatos escolhidos está vazio (linha 1).
- A cada passo, testa se o aumento de S constitui uma solução e ainda existem candidatos a serem avaliados (condição linha 2).
- O melhor candidato restante ainda não tentado é considerado e removido de C (linhas 3 e 4). O critério de escolha é ditado pela função de seleção (linha 3).
- Se o conjunto aumentado de candidatos se torna viável, o candidato é adicionado ao conjunto S de candidatos escolhidos. Caso contrário ;e rejeitado (linhas 5 e 6).
- Ao final do processo, testa se há uma solução (linha 7) que é retornada (linha 8) ou não (linha 9).

Características da implementação de algoritmos gulosos

- Quando funciona corretamente, a primeira solução encontrada é sempre ótima.

- A função de seleção é geralmente relacionada com a função objetivo.

- Se o objetivo é:
  - Maximizar $\Rightarrow$ provavelmente escolherá o candidato restante que proporcione o maior ganho individual.
  - Minimizar $\Rightarrow$ então será escolhido o candidato restante de menor custo.

- O algoritmo nunca muda de idéia:
  - Um candidato escolhido e adicionado à solução passa a fazer parte dessa solução permanentemente.
  - Um candidato excluído do conjunto solução, não é mais reconsiderado.

Árvore geradora mínima

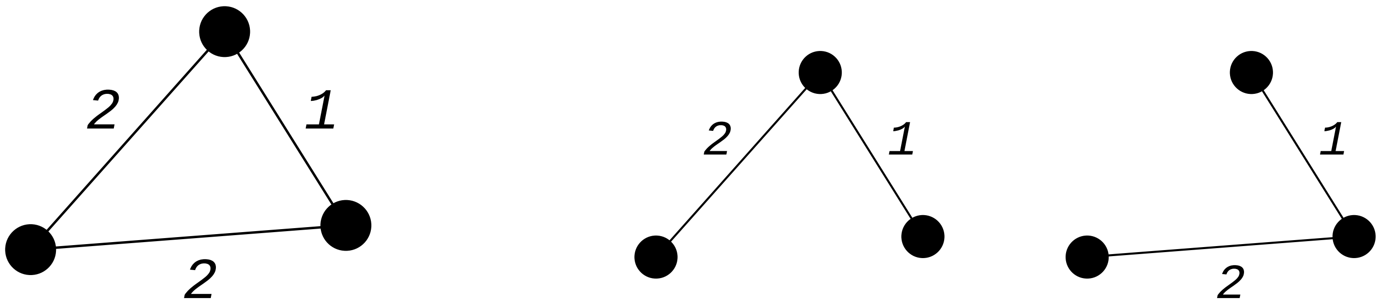
- Definição: Uma árvore geradora de um grafo $G = (V, E)$ é um subgrafo de G que é uma árvore e contém todos os vértices de G. Num grafo com pesos, o peso de um subgrafo é a soma dos pesos das arestas deste subgrafo. Uma *árvore geradora mínima* para um grafo com pesos é uma árvore geradora com peso mínimo.

- Problema: Determinar a *árvore geradora mínima* (em inglês, *minimum spanning tree*), de um grafo com pesos.

- Aplicações: Determinar a maneira mais barata de se conectar um conjunto de terminais, sejam eles cidades, terminais elétricos, computadores, ou fábricas, usando-se, por exemplo, estradas, fios, ou linhas de comunicação

Árvore geradora mínima: Soluções

O exemplo abaixo mostra que um grafo pode ter mais de uma árvore geradora mínima.



Grafo conexo

- Um grafo é *conexo* se, para cada par de vértices v e w, existir um caminho de v para w.

- Observações:
  - Se G não for conexo ele não possui nenhuma árvore geradora, muito menos uma que seja mínima.
  - Neste caso ele teria uma floresta geradora.

- Para simplificar a apresentação do algoritmo para árvore geradora mínima, vamos assumir que G é conexo. É fácil estender o algoritmo (e sua justificativa) para determinar uma floresta de árvores geradoras mínimas.

Árvore geradora mínima
Algoritmo de Dijkstra–Prim (1959/57)

O algoritmo de Dijkstra–Prim começa selecionando um vértice arbitrário, e depois "aumenta" a árvore construída até então escolhendo um novo vértice (e um nova aresta) a cada iteração. Durante a execução do algoritmo, podemos imaginar os vértices divididos em três categorias:

1. *Vértices da árvore*: aqueles que fazem parte da árvore construída até então;
2. *Vértices da borda*: não estão na árvore, mas são adjacentes a algum vértice da árvore;
3. *Vértices não-vistos*: todos os outros.

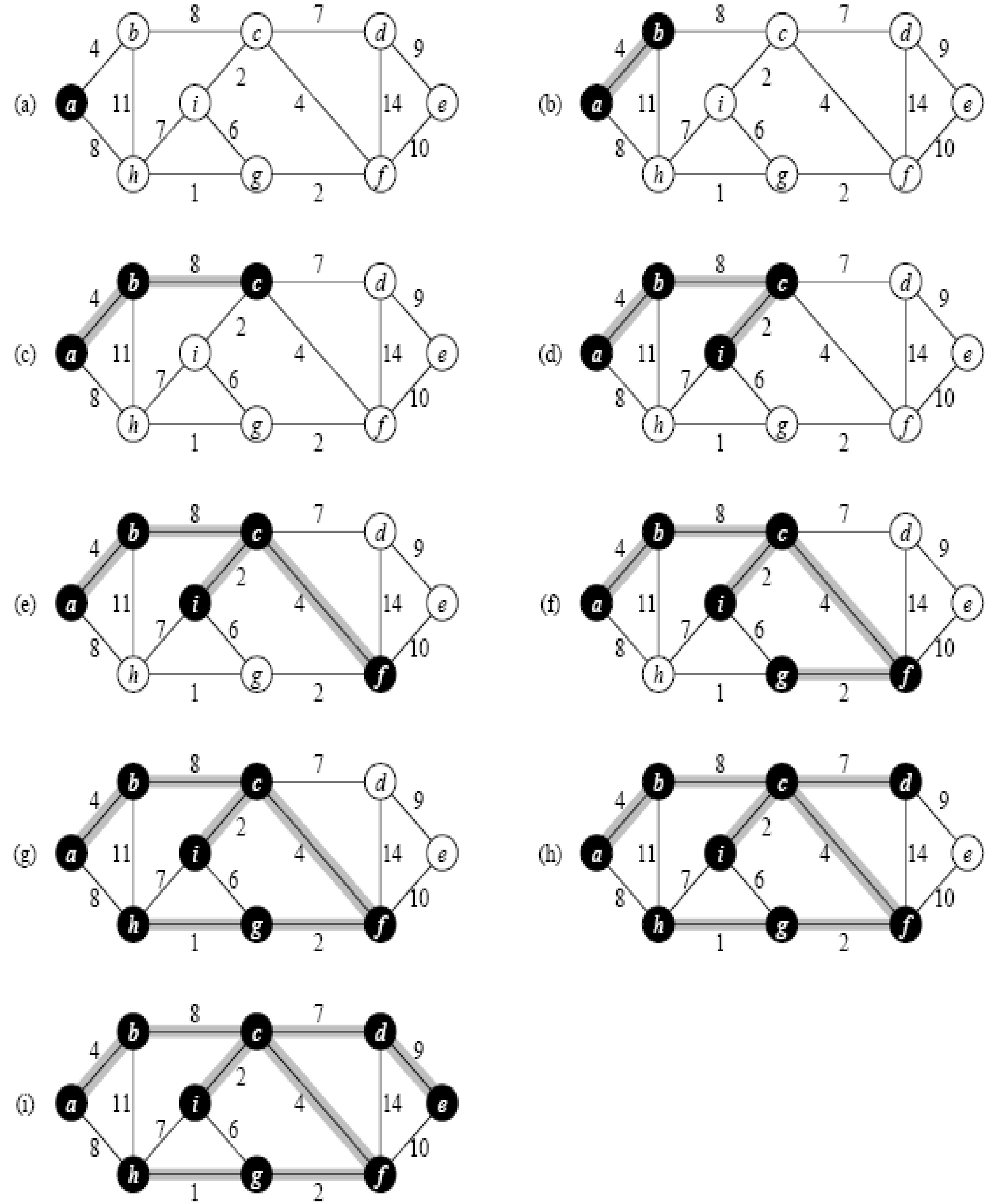
Árvore geradora mínima Algoritmo de Dijkstra–Prim

O principal passo do algoritmo é a seleção de um vértice da borda e de uma aresta incidente a este vértice. O algoritmo de Dijkstra–Prim sempre escolhe uma aresta entre um vértice da árvore e um vértice da borda que tenha peso mínimo. A estrutura geral do algoritmo pode ser descrita do seguinte modo:

```
DIJKSTRA-PRIM(Grafo)
Seleciona um vértice arbitrário para inicializar a árvore;
while existem vértices da borda
   do Seleciona uma aresta de peso mínimo entre um vértice da árvore e um vértice da
      borda;
      Adiciona a aresta selecionada e o vértice da borda à árvore;
```

Algoritmo de Dijkstra–Prim

- Idéia básica:
  - Tomando como vértice inicial A, crie uma fila de prioridades classificada pelos pesos das arestas conectando A.
  - Repita o processo até que todos os vértices tenham sido visitados.



Princípio da técnica de algoritmos gulosos

- A cada passo faça a melhor escolha:
  - Escolha local é ótima.
- Objetivo:
  - ➜ A solução final ser ótima também.
- Sempre funciona?
  - Não. Por exemplo, *0–1 Knapsack Problem*.
- Propriedades de problemas que, em geral, levam ao uso da estratégia gulosa:
  - Propriedade da escolha gulosa.
  - Sub-estrutura ótima.

Propriedade da escolha gulosa

- Solução global ótima pode ser obtida a partir de escolhas locais ótimas.
- Estratégia diferente de programação dinâmica (PD).
- Uma vez feita a escolha, resolve o problema a partir do "estado" em que se encontra.
- Escolha na técnica gulosa depende só do que foi feito e não do que será feito no futuro.
- Progride na forma *top-down*:
  - Através de iterações vai "transformando" uma instância do problema em uma outra menor.

Propriedade da escolha gulosa

- Estratégia da prova que a escolha gulosa leva a uma solução global ótima:
  - Examine a solução global ótima.
  - Mostre que a solução pode ser modificada de tal forma que uma escolha gulosa pode ser aplicada como primeiro passo.
  - Mostre que essa escolha reduz o problema a um similar mas menor.
  - Aplique indução para mostrar que uma escolha gulosa pode ser aplicada a cada passo.

Sub-estrutura ótima

- Um problema exibe sub-estrutura ótima se uma solução ótima para o problema é formada por soluções ótimas para os sub-problemas.

- Técnicas de escolha gulosa e programação dinâmica possuem essa característica.

Técnica gulosa vs. Programação dinâmica

- Possuem sub-estrutura ótima.

- Programação dinâmica:
  - Faz uma escolha a cada passo.
  - Escolha depende das soluções dos sub-problemas.
  - Resolve os problemas *bottom-up*.

- Técnica gulosa:
  - Trabalha na forma *top-down*.

Diferenças das duas técnicas através de um exemplo

- Problema da Mochila (enunciado):
  - Um ladrão acha n itens numa loja.
  - Item i vale v_i unidades (dinheiro, e.g., R$, US$, etc).
  - Item i pesa w_i unidades (kg, etc).
  - v_i e w_i são inteiros.
  - Consegue carregar W unidades no máximo.
  - Deseja carregar a “carga” mais valiosa.

Versões do Problema da Mochila

- Problema da Mochila 0–1 ou (*0–1 Knapsack Problem*):
  - O item i é levado integralmente ou é deixado.

- Problema da Mochila Fracionário:
  - Fração do item i pode ser levada.

Considerações sobre as duas versões

- Possuem a propriedade de sub-estrutura ótima.

- Problema inteiro:
  - Considere uma carga que pesa no máximo W com n itens.
  - Remova o item j da carga (específico mas genérico).
  - Carga restante deve ser a mais valiosa pesando no máximo $W - w_j$ com $n - 1$ itens.

- Problema fracionário:
  - Considere uma carga que pesa no máximo W com n itens.
  - Remova um peso w do item j da carga (específico mas genérico).
  - Carga restante deve ser a mais valiosa pesando no máximo $W - w$ com $n - 1$ itens mais o peso $w_j - w$ do item j.

Considerações sobre as duas versões

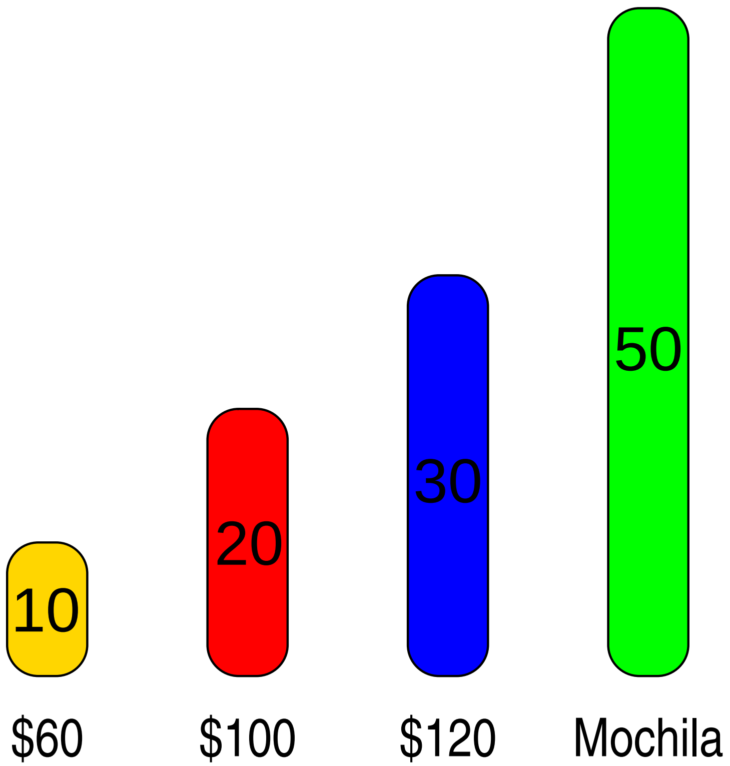
- Problema inteiro:
  ➜ Não é resolvido usando a técnica gulosa.

- Problema fracionário:
  ➜ É resolvido usando a técnica gulosa.

- Estratégia para resolver o problema fracionário:
  - Calcule o valor por unidade de peso $\frac{v_i}{w_i}$ para cada item.
  - Estratégia gulosa é levar tanto quanto possível do item de maior valor por unidade de peso.
  - Repita o processo para o próximo item com esta propriedade até alcançar a carga máxima.

- Complexidade para resolver o problema fracionário:
  - Ordene os itens i $(i = 1 \ldots n)$, pelas frações $\frac{v_i}{w_i}$.
  ➜ $O(n \log n)$.

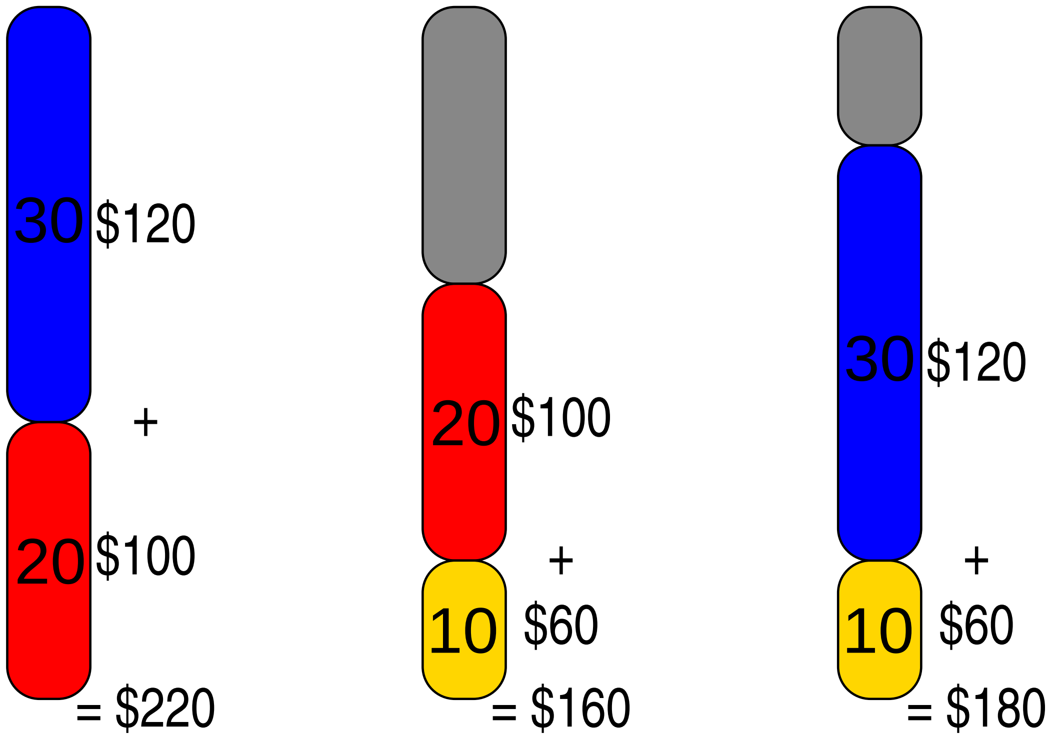
Exemplo: Situação inicial
Problema 0–1



Item	Peso	Valor	V/P
1	10	60	6
2	20	100	5
3	30	120	4

Carga máxima da mochila: 50

Exemplo: Estratégia gulosa
Problema 0 –1



Soluções possíveis:

#	Item (Valor)
1	2 + 3 = 100 + 120 = 220
2	1 + 2 = 60 + 100 = 160
3	1 + 3 = 60 + 120 = 180

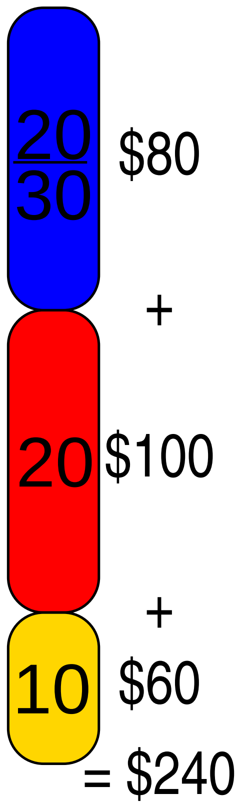
➜ Solução 2 é a gulosa.

Exemplo: Estratégia gulosa Problema 0–1

- Considerações:
  - Levar o item 1 faz com que a mochila fique com espaço vazio
  - Espaço vazio diminui o valor efetivo da relação $\frac{v}{w}$
  - Neste caso deve-se comparar a solução do sub-problema quando:

    Item é incluído na solução $\times$ Item é excluído da solução
  - Passam a existir vários sub-problemas
  - Programação dinâmica passa a ser a técnica adequada

Exemplo: Estratégia gulosa
Problema Fracionário



Item	Peso	Valor	Fração
1	10	60	1
2	20	100	1
3	30	80	2/3

- ➜ Total = 240.
- ➜ Solução ótima!

Algoritmos aproximados

- Problemas que somente possuem algoritmos exponenciais para resolvê-los são considerados "difíceis".

- Problemas considerados intratáveis ou difíceis são muito comuns, tais como:
  – Problema do caixeiro viajante cuja complexidade de tempo é $O(n!)$.

- Diante de um problema difícil é comum remover a exigência de que o algoritmo tenha sempre que obter a solução ótima.

- Neste caso procuramos por algoritmos eficientes que não garantem obter a solução ótima, mas uma que seja a mais próxima possível da solução ótima.

Tipos de algoritmos aproximados

- **Heurística**: é um algoritmo que pode produzir um bom resultado, ou até mesmo obter a solução ótima, mas pode também não produzir solução alguma ou uma solução que está distante da solução ótima.

- **Algoritmo aproximado**: é um algoritmo que gera *soluções aproximadas* dentro de um limite para a razão entre a solução ótima e a produzida pelo algoritmo aproximado (comportamento monitorado sob o ponto de vista da qualidade dos resultados).