Divisão e Conquista

Notas de aula da disciplina IME 04-10823
ALGORITMOS E ESTRUTURAS DE DADOS II

Paulo Eustáquio Duarte Pinto
(pauloedp arroba ime.uerj.br)

abril/2012

Divisão e Conquista

É uma técnica para resolver problemas (construir algoritmos) que
subdivide
o problema em
subproblemas menores, de mesma natureza
e
compõe
a solução desses subproblemas, obtendo uma solução do problema original.

DIVIDIR PARA CONQUISTAR!!!


```
Fatorial(p);
      Se (p = 0) Então
            Retornar 1
      Senão
            Retornar  p.Fatorial(p-1);
Fim;

Fib(p);
      Se (p ≤ 1 ) Então
            Retornar p
      Senão
            Retornar Fib(p-1) + Fib(p-2);
Fim;
```


Visões sobre Recursão:

a) Solução de problemas de trás para frente
enfatiza-se os passos finais da solução, após ter-se resolvido problemas menores. Mas a solução de problemas pequenos ("problemas infantís") tem que ser mostrada.

b) Analogia com a "Indução finita"
Indução Finita: prova-se resultados matemáticos gerais supondo-os válidos para valores inferiores a n e demonstrando que o resultado vale também para n. Além disso, mostra-se que o resultado é correto para casos particulares.

Visões sobre Recursão:

c) Equivalente procedural de Recorrências
Recorrências são maneiras de formular funções para n, utilizando resultados da mesma função para valores menores que n. Além disso uma recorrência deve exibir resultados específicos para determinados valores.

d) Estrutura de um procedimento recursivo
Procedimentos recursivos 'chamam a sí mesmos'. Um procedimento recursivo começa com um 'Se', para separar subproblemas 'infantís' dos demais. O procedimento chama a si mesmo pelo menos uma vez. Sempre há uma chamada externa.

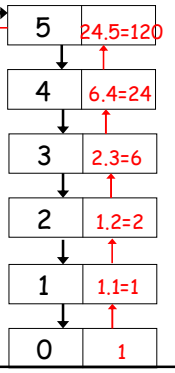
Dinâmica da execução de um procedimento recursivo.

Sempre que uma chamada recursiva é executada, o sistema operacional empilha as variáveis locais e a instrução em execução, desempilhando esses elementos no retorno da chamada.

Chamadas recursivas podem ser expressas através de uma árvore de recursão.

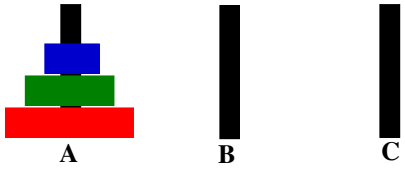
Árvore de recursão para Fatorial

X ← Fatorial(5)



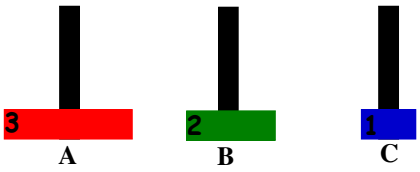

```
Fatorial(p);
        Se (p = 0) Então
                Retornar 1
        Senão
                Retornar p.Fatorial(p-1);
Fim;
```


TORRE DE HANOI



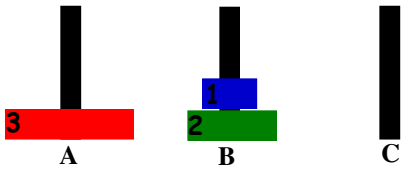
Quer-se levar os n pratos da vareta **A** para a vareta **C**, um a um, usando **B** como auxílio, sem nunca colocar um prato sobre um prato menor.

TORRE DE HANOI - exemplo



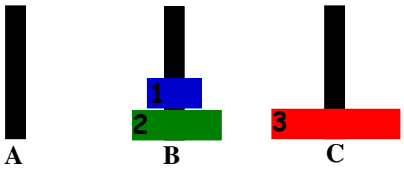
Mover 1 para C
Mover 2 para B

TORRE DE HANOI - exemplo



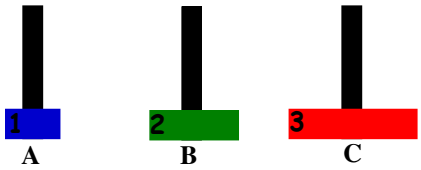
Mover 1 para B

TORRE DE HANOI - exemplo



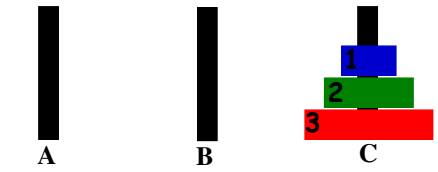
Mover 3 para C

TORRE DE HANOI - exemplo



Mover 1 para A

TORRE DE HANOI - exemplo



Mover 2 para C
Mover 1 para C

Problema resolvido com 7 movimentos!!

TORRE DE HANOI - Formulação 1

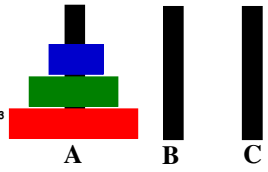
```
Hanoi (n, V1, V2, V3);
      Se (n = 1) Então
            Levar topo de V1 para V3
      Senão
            Hanoi (n-1, V1, V3, V2)
            Mover topo de V1 para V3
            Hanoi (n-1, V2, V1, V3);
Fim;
```


TORRE DE HANOI - Formulação 2

```
Hanoi (n, V1, V2, V3);
      Se (n > 0) Então
            Hanoi (n-1, V1, V3, V2)
            Mover topo de V1 para V3
            Hanoi (n-1, V2, V1, V3);
Fim;
```


TORRE DE HANOI - Árvore de Recursão

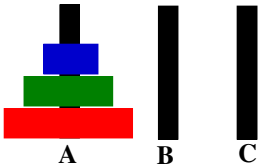
```
Hanoi (n, V1, V2, V3);
(1) Se (n > 0) Então
(2)     Hanoi (n-1, V1, V3, V2)
(3)     Mover topo de V1 para V3
(4)     Hanoi (n-1, V2, V1, V3);
Fim;
```

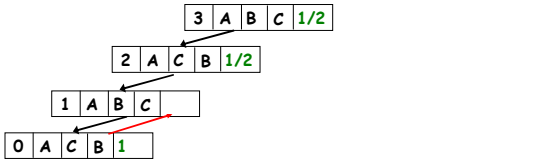


Hanoi (3, A, B, C);

TORRE DE HANOI - Árvore de Recursão

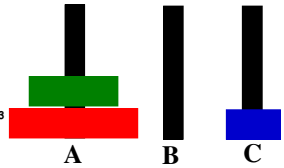
```
Hanoi (n, V1, V2, V3);
(1) Se (n > 0) Então
(2)     Hanoi (n-1, V1, V3, V2)
(3)     Mover topo de V1 para V3
(4)     Hanoi (n-1, V2, V1, V3);
Fim;
```

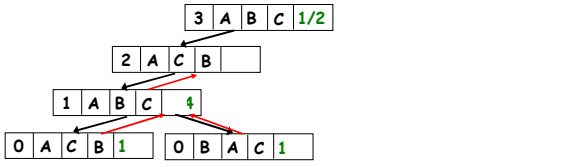




TORRE DE HANOI - Árvore de Recursão

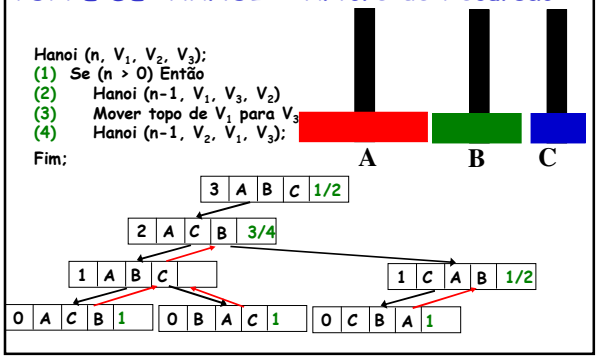
```
Hanoi (n, V1, V2, V3);
(1) Se (n > 0) Então
(2)     Hanoi (n-1, V1, V3, V2)
(3)     Mover topo de V1 para V3
(4)     Hanoi (n-1, V2, V1, V3);
Fim;
```





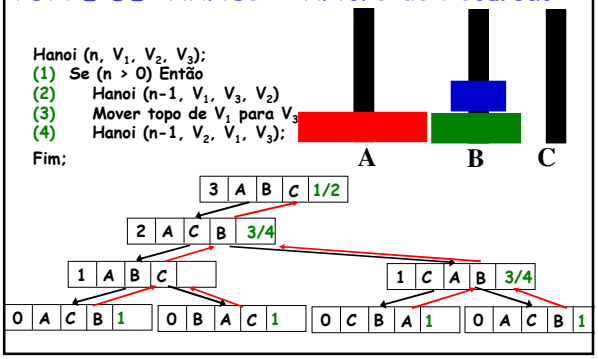
TORRE DE HANOI - Árvore de Recursão

Hanoi (n, V_1, V_2, V_3);
(1) Se (n > 0) Então
(2) Hanoi (n-1, V_1, V_3, V_2)
(3) Mover topo de V_1 para V_3
(4) Hanoi (n-1, V_2, V_1, V_3);
Fim;



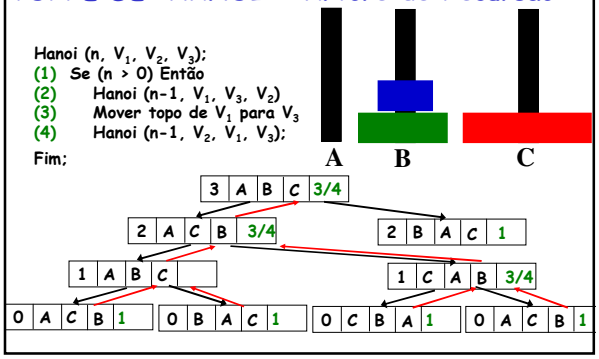
TORRE DE HANOI - Árvore de Recursão

Hanoi (n, V_1, V_2, V_3);
(1) Se (n > 0) Então
(2) Hanoi (n-1, V_1, V_3, V_2)
(3) Mover topo de V_1 para V_3
(4) Hanoi (n-1, V_2, V_1, V_3);
Fim;



TORRE DE HANOI - Árvore de Recursão

Hanoi (n, V_1, V_2, V_3);
(1) Se (n > 0) Então
(2) Hanoi (n-1, V_1, V_3, V_2)
(3) Mover topo de V_1 para V_3
(4) Hanoi (n-1, V_2, V_1, V_3);
Fim;



TORRE DE HANOI - Recorrência

T(n) = número de movimentos para mover n pratos
T(n) = 2*T(n-1) + 1
T(0) = 0 ou T(1) = 1

Solução da recorrência:

$T(n) = 2^n - 1$

Exercício:

Provar, por indução, que a fórmula acima é verdadeira.

TORRE DE HANOI - Recorrência

T(n) = número de movimentos para mover n pratos
T(n) = 2*T(n-1) + 1
T(0) = 0 ou T(1) = 1

Solução da recorrência:

$T(n) = 2^n - 1$

Prova:

a) Verdadeiro para n = 0 e n = 1
b) Se verdadeiro para n-1,
$T(n) = 2*(2^{n-1} - 1) + 1 = 2^n - 1$
então também é verdadeiro para n.

TORRE DE HANOI - Recorrência

T(n) = número de movimentos para mover n pratos
$T(n) = 2^n - 1$

Então o algoritmo é ineficiente?

Resposta:

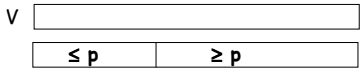
Não. Prova-se que qualquer solução exige um número exponencial de passos. Então o problema é que é "ruim", não o algoritmo.

Quicksort:

Este é um importante método de ordenação, inventado por Hoare, em 1962.

A idéia recursiva básica é a seguinte:

a) Fazer uma partição no vetor, através de trocas, baseada em um pivô p, tal que os elementos da partição esquerda sejam ≤ p, e os da direita, ≥ p.
b) Então basta ordenar, recursivamente, as duas partições geradas.
c) O problema infantil é a partição ter tamanho igual ou inferior a 1, quando nada deve ser feito.



Quicksort:

```
Quicksort (e, d);
      Se (d > e) Então
            Particao (e, d, i, j);
            Quicksort (e, j);
            Quicksort (i, d);
Fim;
```

A idéia é fazer partição no vetor em 2 ou 3 partes. O algoritmo de Partição a ser apresentado, usa como pivô o elemento do meio do vetor e, algumas vezes, particiona o vetor em 3 partes, onde a parte do meio tem tamanho 1 e é igual ao pivô.

Quicksort: exemplo de partição

1	2	3	4	5	6	7	8	9	10	11
P	R	I	M	O	G	E	N	I	T	O

1	2	3	4	5	6	7	8	9	10	11
E	R	I	M	O	G	P	N	I	T	O

Quicksort: exemplo de partição

1	2	3	4	5	6	7	8	9	10	11
P	R	I	M	O	G	E	N	I	T	O
E	R	I	M	O	G	P	N	I	T	O
E	G	I	M	O	R	P	N	I	T	O

Quicksort:

```
Particao (e, d, i, j);
      i ← e;   j ← d;   t ← V[⌊(e+d)/2⌋];
      Enquanto (i ≤ j):
            Enquanto (V[i] < t):   i ← i+1;   Fe;
            Enquanto (V[j] > t):   j ← j-1;   Fe;
            Se (i ≤ j):
                  Troca(i, j); i ← i+1; j ← j-1;
      Fe;
Fim;
```


Quicksort: exemplo de partição

1	2	3	4	5	6	7	8	9	10	11
P	R	I	M	O	G	E	N	I	T	O

Quicksort: exemplo de partição

1	2	3	4	5	6	7	8	9	10	11
E	R	I	M	O	G	P	N	I	T	O

Quicksort: exemplo de partição

1	2	3	4	5	6	7	8	9	10	11
E	G	I	M	O	R	P	N	I	T	O

Quicksort: exemplo de partição

1	2	3	4	5	6	7	8	9	10	11
E	G	I	M	O	R	P	N	I	T	O

Quicksort: exemplos de partição

1	2	3	4	5	6	7	8	9	10	11
E	G	I	M	O	R	P	N	I	T	O

≤ G ≥ G i = 3 j = 2

1	2
E	G

1	2
E	G

=E ≥ E i = 2 j = 0 (!!)

Quicksort

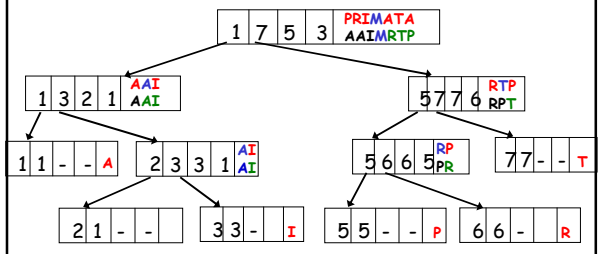
Exercício:

Executar o algoritmo de Partição para o MIXSTRING (10 letras)

Quicksort: exemplo

1	2	3	4	5	6	7
P	R	I	M	A	T	A
A	A	I	M	R	T	P
A	A	I		R	T	P
A				R	P	T
	A	I		R	P	
		I		P	R	
				P		
					R	
						T

Quicksort: exemplo da árvore de recursão



Quicksort

Exercício:

Mostrar a execução do Quicksort para o MIXSTRING (10 letras). Mostrar a árvore de recursão.

Análise da Ordenação pelo Quicksort:

Complexidade:

Pior caso: $O(n^2)$

Melhor caso = caso médio: $O(n \log n)$

Estabilidade (manutenção da ordem relativa de chaves iguais):

Algoritmo não estável

Memória adicional:

Pilha para recursão

Usos especiais:

Melhor algoritmo de ordenação em geral

Máximo e Mínimo.

Problema: Dado um conjunto de números $S = \{s_1, s_2, \ldots s_n\}$, determinar simultâneamente o menor e o maior elementos do conjunto.

Solução ingênua:
Encontrar o mínimo e o máximo, separadamente, utilizando 2*(n-1) comparações.

Pergunta:
É possível encontrar os números procurados fazendo menos que 2*(n-1) comparações?

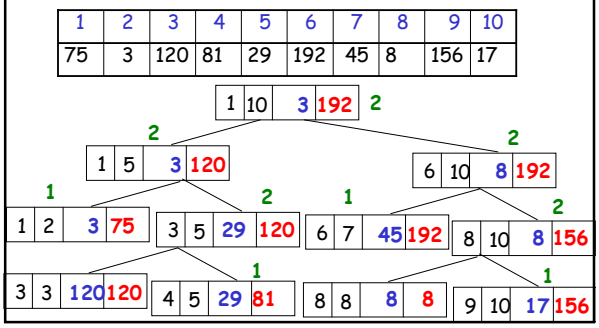
Resposta: Sim, usando um enfoque recursivo.

Máximo e Mínimo - Enfoque Recursivo.

```
Maxmin(S)
    Se (n = 1) Então
        Retornar (s1, s1)
    Senão Se (n = 2) Então
        Se (s1 < s2) Então
            Retornar (s1, s2)
        Senão
            Retornar (s2, s1)
    Senão
        m ← ⌊n/2⌋;
        (a1, b1) ← MaxMin(S1 = {s1,... sm})
        (a2, b2) ← MaxMin(S2 = {sm+1,... sn})
        Retornar (min(a1, a2), max (b1, b2))
Fim;
```


Máximo e Mínimo - Exemplo.

Comparações: 14 (Recursão) x 18 (2 Loops)

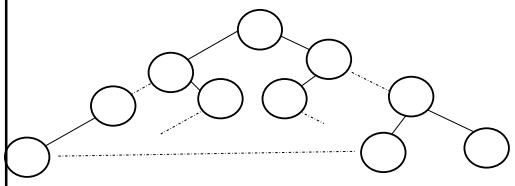


Máximo e Mínimo.

Pergunta:
É possível encontrar os números procurados fazendo menos que 2*(n-1) comparações?

Resposta: Sim, com recursão usa-se 3n/2 - 2.

Máximo e Mínimo. Comparações para n = 2^k



altura da árvore = k
número de nós no último nível = n/2
comparações no último nível = n/2
número de nós intermediários = n – 1 – n/2 = n/2 – 1
número de comparações em nós intermediários = 2(n/2-1)
total de comparações: n/2 + 2n/2 – 2 = 3n/2 - 2

Máximo e Mínimo

Exercício:
Escrever um algoritmo não recursivo para achar máximo e mínimo de um conjunto, fazendo 3n/2 - 2 comparações.

Máximo e Mínimo - Enfoque Não Recursivo.

Idéia: varrer o vetor, comparar os elementos 2 a 2, e co-locar cada elemento do par ou num vetor de candidatos a mínimos ou de candidatos a máximos. Depois obter mínimo e máximo separadamente. Só que a obtenção do mínimo e máximo pode ser feita durante o processo.

```
Maxmin;
        vmin ← V[n];  vmax ← V[n];
        Para i de 1 a ⌊n/2⌋:
            Se (V[2i-1] < V[2i]) Então
                vmin ← min(V[2i-1],vmin);  vmax ← max(V[2i],vmax);
            Senão
                vmin ← min(V[2i],vmin);  vmax ← max(V[2i-1],vmax);
        Fp;
        Retornar (vmin, vmax);
Fim;
```


Cálculo de Combinações

Problema: Às vezes tem-se problemas numéricos no cálculo de combinações, se usar-se a fórmula.
Ex: Calculando-se Comb(50, 3), haveria um estouro numérico no cálculo de 50!

A versão recursiva pode evitar o problema.

Versão 1. Define-se a recorrência:
Comb(n, p) = n, se p = 1
Comb(n, p) = Comb(n, p-1)*(n-p+1)/p , se p > 1.

Cálculo de Combinações

Versão 1. Define-se a recorrência:
Comb(n, p) = n, se p = 1
Comb(n, p) = Comb(n, p-1)*(n-p+1)/p.

Exemplo: Comb(50, 3).

50	3	1225*48/3 = 19600
50	2	50*49/2 = 1225
50	1	50

Cálculo de Combinações

Exercício:

Escrever outra versão recursiva para o cálculo de combinações (Comb(n, p)), diminuindo o problema pelo n.

Divisão e Conquista

Cálculo de Combinações

Exercício:

Escrever outra versão recursiva para o cálculo de combinações (Comb(n, p)), diminuindo o problema pelo n.

Divisão e Conquista

Torneio.

Problema: quer-se montar uma tabela de torneio, com n competidores, onde todos os competidores jogam entre si, em rodadas. Se n for par, deve haver n-1 rodadas e, em cada uma delas, todos os times jogam; se n for ímpar deve haver n rodadas, ficando um time "bye" em cada uma.

Exemplos para n = 6 e n = 5.

r_0	r_1	r_2	r_3	r_4	r_5
1	2	3	4	5	6
2	1	5	3	6	4
3	6	1	2	4	5
4	5	6	1	3	2
5	4	2	6	1	3
6	3	4	5	2	1

r_0	r_1	r_2	r_3	r_4	r_5
1	2	3	4	5	-
2	1	5	3	-	4
3	-	1	2	4	5
4	5	-	1	3	2
5	4	2	-	1	3

Divisão e Conquista

Propriedades da tabela de Torneio com n par.

Considerando a coluna r_0 como numeração dos times 1 a n:

a) cada linha é uma permutação de 1 a n.

b) cada coluna é uma permutação de 1 a n.

c) T[i, j] = p ⇒ T[p, j] = i, pelo emparelhamento.

Exemplos para n = 6 e n = 5.

r_0	r_1	r_2	r_3	r_4	r_5
1	2	3	4	5	6
2	1	5	3	6	4
3	6	1	2	4	5
4	5	6	1	3	2
5	4	2	6	1	3
6	3	4	5	2	1

r_0	r_1	r_2	r_3	r_4	r_5
1	2	3	4	5	-
2	1	5	3	-	4
3	-	1	2	4	5
4	5	-	1	3	2
5	4	2	-	1	3

Divisão e Conquista

Propriedades da tabela de Torneio com n ímpar.

Considera-se mais um time artificial e substituem-se os "bye" por esse time. Passa-se a ter a situação par.

Exemplos para n = 6 e n = 5.

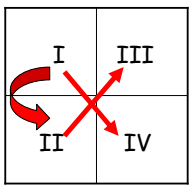
r_0	r_1	r_2	r_3	r_4	r_5
1	2	3	4	5	6
2	1	5	3	6	4
3	6	1	2	4	5
4	5	6	1	3	2
5	4	2	6	1	3
6	3	4	5	2	1

r_0	r_1	r_2	r_3	r_4	r_5
1	2	3	4	5	6
2	1	5	3	6	4
3	6	1	2	4	5
4	5	6	1	3	2
5	4	2	6	1	3
6	3	4	5	2	1

Divisão e Conquista

Torneio para n = 2^k. Exemplo para n = 8

Sempre há as simetrias recursivas:



r_0	r_1	r_2	r_3	r_4	r_5	r_6	r_7
1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

Quadrante I copiado, c/ deslocamento de n/2, para II

Quadrante II copiado para III

Quadrante I copiado para IV

Torneio para n = 2^k. Versão I.

```
Torneio (m);
      Se (m = 1) Então
            T[1,1] ← 1
      Senão
            p ← m/2;   Torneio(p);
            Para i de 1 a p:
                  Para j de 1 a p:
                        T[i+p, j]   ← T[i, j] + p;
                        T[i, j+p]   ← T[i+p, j];
                        T[i+p, j+p] ← T[i, j];
                  Fp;
            Fp;
Fim;
```

Complexidade: $O(n^2)$

Torneio para n = 2^k. Versão I.
Porque o algoritmo está correto?

a) Cada linha é permutação de 1 a n (por construção)

b) Cada coluna é permutação de 1 a n (por construção).

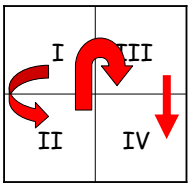
c) resta ver a correção do emparelhamento. Prova por indução em n:

c.1) O emparelhamento está correto para n = 2, pois só há uma rodada. Por inspeção, T[1,2] = 2 e T[2,1] = 1.

c.2) Suponhamos o emparelhamento correto em QI, com número de elementos = p = n/2. Então o emparelhamento também está correto em QII, pois ele é só uma renomeação de QI.
Tomemos um elemento do quadrante III, emparelhado em QIV:
T[i, j+p] = k, 1≤ i,j ≤ p; p = n/2.
Temos que mostrar que T[k, j+p] = i. Isso é verdade, pois:
T[i, j+p] = k = T[i,j] + p. Portanto,
T[k, j+p] = T[T[i,j]+p,j+p] = T[T[i,j],j].
Se T[i,j] = x, Então T[x,j] = i, em QI, por hipótese.
Então T[k, j+p] = T[T[i,j],j] = i.

Torneio para n = 2^k. Versão 2. Exemplo p/ n = 8

Sempre há as simetrias recursivas:



r_0	r_1	r_2	r_3	r_4	r_5	r_6	r_7
1	2	3	4	5	6	7	8
2	1	4	3	6	7	8	5
3	4	1	2	7	8	5	6
4	3	2	1	8	5	6	7
5	6	7	8	1	4	3	2
6	5	8	7	2	1	4	3
7	8	5	6	3	2	1	4
8	7	6	5	4	3	2	1

Quadrante III preenchido c/ Permutação circular da coluna 1 do Quadrante II

Quadrante I copiado, c/ deslocamento de n/2, para II

Quadrante IV com preenchimento forçado

Torneio

Exercício:

Escrever um algoritmo para a Versão 2 do Torneio.

Dica: pense em como gerar permutações circulares para 0 1 ....p-1.

Torneio para n = 2^k. Versão 2.

```
Torneio (m);
      Se (m = 1) Então
            T[1,1] ← 1
      Senão
            p ← m/2;   Torneio(p);
            Para i de 1 a p:
                  Para j de 1 a p:
                        T[i+p, j]   ← T[i, j] + p;
                        T[i, j+p]   ← p+1+(i+j-2) mod p;
                        T[p+1+(i+j-2)mod p, j+p] ← i;
                  Fp;
            Fp;
Fim;
```

Complexidade: $O(n^2)$

Torneio para n = 2^k. Versão II.
Porque o algoritmo está correto?

a) Cada linha é permutação de 1 a n (por construção)

b) Cada coluna é permutação de 1 a n (por construção).

c) resta ver a correção do emparelhamento. Prova por indução em n:

c.1) O emparelhamento está correto para n = 2, pois só há uma rodada. Por inspeção, T[1,2] = 2 e T[2,1] = 1.

c.2) Suponhamos o emparelhamento correto em QI, com número de elementos = p = n/2.
Então o emparelhamento também está correto em QII, pois ele é só uma renomeação de QI.
O emparelhamento em QIII e QIV está correto por construção.

Torneio para n genérico.

Idéia: adaptar a Versão 2 do algoritmo anterior. O passo final da recursão depende da forma de n:

a) n ímpar transformado em par (n+1).

b) n= 4k, usar o esquema da Versão 2.

c) n= 4k+2, tem-se um problema, pois n/2 é ímpar e a montagem do Torneio para n/2 usou n/2 rodadas, implicando um total de n rodadas, o que não é o objetivo. Que fazer então?

R: Adaptar a solução gerada para transformar em n-1 rodadas. É POSSÍVEL !!!

Torneio para n genérico. n = 3

Resolve-se o problema para n = 4 e depois elimina-se a linha 4 e transforma o 4 em "bye"

	r_1	r_2	r_3
1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

Torneio para n genérico. n = 7

Resolve-se o problema para n = 8 e depois elimina-se a linha 8 e transforma o 8 em "bye"

	r_1	r_2	r_3	r_4	r_5	r_6	r_7
1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

Torneio para n genérico. n = 12

Como n é da forma 4k, resolve-se o problema para n = 6 e aplica-se o esquema da Versão 2.

	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	r_{11}
1	2	3	4	5	6	7	8	9	10	11	12
2	1	5	3	6	4	8	9	10	11	12	7
3	6	1	2	4	5	9	10	11	12	7	8
4	5	6	1	3	2	10	11	12	7	8	9
5	4	2	6	1	3	11	12	7	8	9	10
6	3	4	5	2	1	12	7	8	9	10	11
7	8	9	10	11	12	1	6	5	4	3	2
8	7	11	9	12	10	2	1	6	5	4	3
9	12	7	8	10	11	3	2	1	6	5	4
10	11	12	7	9	8	4	3	2	1	6	5
11	10	8	12	7	9	5	4	3	2	1	6
12	9	10	11	8	7	6	5	4	3	2	1

Torneio para n genérico. n = 6

Este é o caso problemático. Tem-se que adaptar a solução.

Preenche-se a tabela como na versão 2, mantendo-se os "bye".

	r_1	r_2	r_3	r_4	r_5	r_6
1	2	3	-	4	5	6
2	1	-	3	5	6	4
3	-	1	2	6	4	5
4	5	6	-	1	3	2
5	4	-	6	2	1	3
6	-	4	5	3	2	1

Torneio para n genérico. n = 6

A ADAPTAÇÃO consiste em eliminar a coluna r_4, movendo cada elemento para a posição de "bye".

	r_1	r_2	r_3	r_4	r_5	r_6
1	2	3	-	4	5	6
2	1	-	3	5	6	4
3	-	1	2	6	4	5
4	5	6	-	1	3	2
5	4	-	6	2	1	3
6	-	4	5	3	2	1

Torneio para n genérico. n = 6

Situação final, com o torneio organizado em 5 rodadas!

	r_1	r_2	r_3	r_4	r_5
1	2	3	4	5	6
2	1	5	3	6	4
3	6	1	2	4	5
4	5	6	1	3	2
5	4	2	6	1	3
6	3	4	5	2	1

Esboço do algoritmo p/ Torneio genérico.

Complexidade: $O(n^2)$

```
Torneio (m);
      Se (m = 1) Então T[1,1] ← 1
      Senão Se (m ímpar)
            Torneio(m+1);    Gera "bye";
      Senão
            p ← m/2;   Torneio(p);
            Copia QI p/ QII, somando p;
            Se (p impar) Então  cq3 ← p+2;  fq3 ← m+1
            Senão               cq3 ← p+1;  fq3 ← m;
            Gera Perm Circular em QIII;
            Preenche QIV forçado;
            Se (p ímpar) Então
                  Move coluna cq3 para pos. "bye";
                  Move p/ esquerda cols (cq3+1) a fq3;
Fim;
```


Torneio

Exercício:

Preencher a tabela de Torneio para n = 10.

Torneio genérico.
Porque o algoritmo está correto?

a) Cada linha é permutação de 1 a n (por construção)

b) Cada coluna é permutação de 1 a n (por construção).

c) o emparelhamento está correto, pois o algoritmo é uma variante da Versão II. A prova é análçoga.

d) a eliminação da coluna cq3 para n ímpar está correta. Vejamos:
Seja o elemento T[i, cq3] = k. k = i+p, por construção.
Portanto T[k, cq3] = i, pois isso é forçado. O "bye" da linha i está em QI, na coluna x. O "bye" da linha k, está em QII na mesma coluna x, pois QII é uma simples renomeação de QI e a linha k é a renomeação da linha i, pois k = i+p.
Portanto, o jogo entre i e k = i+p, pode ser antecipado para a rodada x, a mesma para os dois times.

BALANCEAMENTO

Critério recomendado para gerar recursões eficientes:

Dividir um problema de tamanho n em k subproblemas de tamanho aproximado n/k.

Exemplo: ordenação de um vetor de tamanho n

Mergesort: divide o problema em 2 subproblemas de tamanho n/2.

Bublesort: divide o problema em 2 subproblemas, 1 de tamanho 1 e o outro de tamanho n-1.

BALANCEAMENTO

Mergesort:

Complexidade: $O(n \log n)$

```
Sort(e, d);
      Se (d > e) Então
            m ← ⌊(e+d)/2⌋;   Sort(e, m);   Sort(m+1, d);
            Merge (e, m, d);
Fim;
```

Bubblesort:

Complexidade: $O(n^2)$

```
Sort(d);
      Se (d > 1) Então
            Para j de 2 a d:
                  Se (V[j-1] > V[j]) Então
                        Troca(j-1, j);
            Fp;
            Sort(d-1);
Fim;
```

Análise de algoritmos recursivos

Analisa-se a árvore de recursão, contando o número de chamadas recursivas. Normalmente usa-se uma recorrência para essa determinação.

Analisa-se a complexidade do procedimento recursivo.

Utiliza-se a complexidade do produto das duas quantidades.

Análise de algoritmos recursivos - FAT

```
FAT(n);
      Se (n < 2) Então        Retornar 1
      Senão                   Retornar n.FAT(n-1);
Fim;
```

Número de chamadas recursivas: n

Complexidade do procedimento: $O(1)$

Complexidade do FAT recursivo: $O(n)$

Análise de algoritmos recursivos - FIB

```
FIB(n);
      Se (n < 2) Então Retornar n
      Senão      Retornar FIB(n-1)+FIB(n-2);
Fim;
```

Número de chamadas recursivas: T(n)
T(0) = 1; T(1) = 1;
T(n) = 1+ T(n-1)+T(n-2); $\Rightarrow$ T(n) = 2Fib(n+1)-1;

Complexidade do procedimento: $O(1)$

Complexidade do FIB recursivo: $O(Fib(n))$, que é > $O(2^n)$;

Se um computador fizer um milhão de operações básicas por segundo, levaria mais de 10^8 anos para calcular Fib(100).

MEMORIZAÇÃO

Técnica para fugir da complexidade exponencial, pelo tabelamento de soluções intermediárias.

Exemplo: MOEDAS

Dados m tipos de moedas e seus valores V[1]..V[m], determinar quantas maneiras distintas existem para um troco de valor n.

No Brasil, m = 6 e V = [1, 5, 10, 25, 50,100].

Temos 4 maneiras distintas de dar um troco de 11 centavos:
1+1+1...+1; 1+1+1+1+1+1+5; 1+5+5; 1+10;

MEMORIZAÇÃO - MOEDAS

T(p, n) = número de trocos distintos para n, usando as moedas de tipos 1 a p.
T(p, n) = 0, se n < 0, ou p=0
T(p, 0) = 1;
T(p, n) = T(p, n - V[p]) + T(p-1, n), se n > 0

Temos 4 maneiras de dar um troco de 11 centavos:
1+1+1...+1; 1+1+1+1+1+1+5; 1+5+5; 1+10;

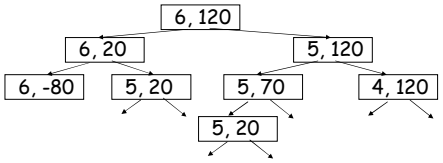
T(6, 11) = T(6,-89)+ T(5, 11) = T(5,-39)+T(4, 11)
= T(4,-14)+ T(3, 11)
= T(3, 1) + T(2,11)
= 1 + 3 = 4

MEMORIZAÇÃO - MOEDAS

T(p, n) = número de trocos distintos para n, usando as moedas de tipos 1 a p.
T(p, n) = 0, se n < 0, ou p=0
T(p, 0) = 1;
T(p, n) = T(p, n - V[p]) + T(p-1, n)

```
Moedas (p, q);
      Se (q < 0) ou (p = 0) Então  Retornar 0
      Senão Se (q = 0) Então Retornar 1
      Senão Retornar Moedas(p,q-V[p]) + Moedas(p-1,q);
Fim;
```

MEMORIZAÇÃO - MOEDAS - n = 120



Árvore de recursão explosiva!!!

MEMORIZAÇÃO - MOEDAS

T(p, n) = 0, se n < 0 ou p = 0;
T(p, 0) = 1;
T(p, n) = T(p, n - V[p]) + T(p-1, n), se n > 0

```
Moedas (p, q);
    Se (q < 0) ou (p = 0) Então  c ← 0
    Senão Se (q = 0) Então  c ← 1
    Senão Se (T[p, q] = -1) Então
                c ← Moedas(p,q-V[p])+Moedas(p-1,q);
                T[p, q] ← c;
          Fs
    Senão c ← T[p, q];
    Retornar c;
Fim;
```


Subproblemas calculados com Memorização

V = {1, 5, 10, 25, 50, 100}

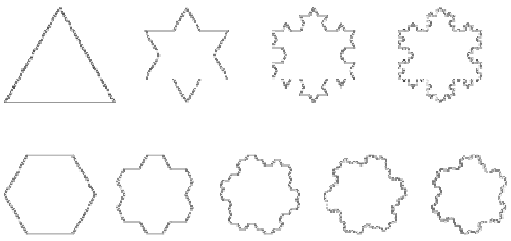
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	-1	-1	-1	-1	2	-1	-1	-1	-1	3	-1	-1	-1	-1	4	-1	-1	-1	-1	5
3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	4	-1	-1	-1	-1	-1	-1	-1	-1	-1	9
4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	9
5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	9
6	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	9

Moedas

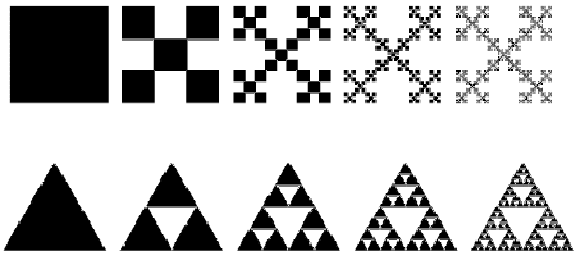
Exercício:

Calcular T(3, 21) a partir da árvore de recursão, com e sem memorização.

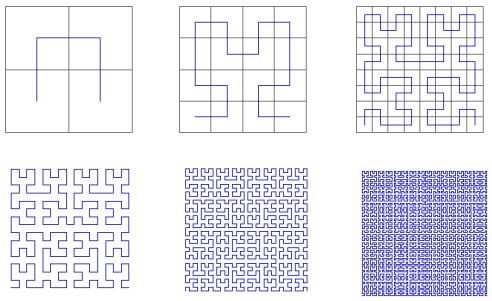
Fractais de Sierpinski



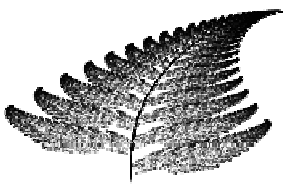
Fractais de Sierpinski

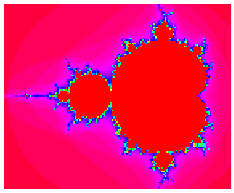


Curvas de Hilbert
(preenchimento do espaço)



Curvas de Mandelbrot





Divisão e Conquista

FIM