Programação 2 _ T11

Estruturas de dados (resumo).

Análise de complexidade de algoritmos. Estratégias de concepção de algoritmos.

Rui Camacho
(slides por Luís Teixeira)
MIEEC 2020/2021

ESTRUTURAS DE DADOS

- **Objetivo**: organizar dados de forma a que sejam úteis e acessíveis de forma fácil/rápida
- **Exemplos**: listas, filas, pilhas, árvores, grafos, heaps, tabelas de dispersão
- Porquê tantas?
  - Estruturas de dados diferentes suportam diferentes tipos de operações (e com diferentes desempenhos)
  - Adequadas para diferentes tarefas
- Como escolher?
  - Quantidade e previsibilidade dos dados
  - Operações pretendidas
  - Limitações (tempo ou espaço)

INSERÇÃO

- Vetor não ordenado: O(1)
- Vetor ordenado: O(N)
- Lista ligada: O(1)
- Lista ligada ordenada: O(N)
- Árvore binária de pesquisa:
  - O(N) pior caso, O(log N) caso médio
- Árvore binária balanceada: O(log N)
- Tabela de dispersão: O(1)

RESUMO

Estrutura	Compexidade temporal (médio / pior caso)				Complexidade espacial (pior caso)
	Acesso	Pesquisa	Inserção	Remoção	
Lista ligada	O(n)	O(n)	O(1)	O(1)	O(n)
Pilha	O(1)	-	O(1)	O(1)	O(n)
Fila	O(1)	-	O(1)	O(1)	O(n)
Heap	O(1)	O(log n)	O(log n)	O(log n)	O(n)
Tabela de dispersão	-	O(1) / O(n)	O(1) / O(n)	O(1) / O(n)	O(n)
Árvore binária de pesquisa	O(log n) / O(n)	O(log n) / O(n)	O(log n) / O(n)	O(log n) / O(n)	O(n)
Árvore AVL	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
Árvore Red-Black	O(log n)	O(log n)	O(log n)	O(log n)	O(n)

ANÁLISE DE COMPLEXIDADE DE ALGORITMOS

COMPLEXIDADE DE ALGORITMOS

- Quando criamos um algoritmo, não basta que ele seja conceptualmente apto a resolver a classe do problema.
- Há algoritmos mais eficientes que outros.
- Importante ter métricas que permitam comparar eficiência e avaliar que recursos vai requerer, nomeadamente em termos de:
  - Tempo de execução
  - Espaço em memória
- Em muitos casos, melhorar um algoritmo em termos dum critério significa piorá-lo em termos do outro.

PROBLEMA DE ORDENAÇÃO

- Problema clássico para ensino de algoritmos
- Ordenar um vetor de dimensão N
- Problema que ocorre em muitas situações reais
- Por exemplo um explorador apresenta os ficheiros ordenados por nome (ou data, ou ..)
- Ou, um jogo de vídeo ordena os objetos 3D presentes na cena usando a distância ao jogador para determinar quais são visíveis (*Visibility Problem*)

ORDENAÇÃO POR SELEÇÃO (EXEMPLO)

- Ordenar um vetor com N elementos (crescente)

(índice *i* varia entre 0 e *N*-1 e trocamos o elemento na posição *i* com o menor entre *i* e *N*)

índice	0	1	2	3	4	5	6	comentário
	4	3	9	6	1	7	0	inicial
i=0	0	3	9	6	1	7	4	troca 0, 4
i=1	0	1	9	6	3	7	4	troca 1, 3
i=2	0	1	3	6	9	7	4	troca 3, 9
i=3	0	1	3	4	9	7	6	troca 6, 4
i=4	0	1	3	4	6	7	9	troca 9, 6
i=5	0	1	3	4	6	7	9	terminado

DESEMPENHO

- Espaço constante -> importa avaliar tempo
- Contabilizar número de acessos ao vetor

- Selection sort
  - $T(n)=n^2+3n-4 \rightarrow n^2$

- Merge Sort
  - $T_m(n)=8n \log n \rightarrow n \log n$

SELECTION VS. MERGE

n	T(n)	$T_m(n)$
2	6	11
3	14	26
4	24	44
5	36	64
6	50	86
7	66	108
8	84	133
9	104	158
10	126	184
11	150	211
12	176	238
13	204	266
14	234	295
15	266	324
16	300	354
17	336	385
18	374	416
19	414	447
20	456	479

n	T(n)	$T_m(n)$
20	456	479
21	500	511
22	546	544
23	594	576
24	644	610
25	696	643
26	750	677
27	806	711
28	864	746
29	924	781
30	986	816

n	T(n)	$T_m(n)$
100	10,296	3,684
1,000	1,002,996	55,262
10,000	100,029,996	736,827
100,000	10,000,299,996	9,210,340
1,000,000	1,000,002,999,996	110,524,084
10,000,000	100,000,029,999,996	1,289,447,652

ANÁLISE DE COMPLEXIDADE DE ALGORITMOS

- Usual **optimizar** o algoritmo em termos de **tempo de execução**, tendo em conta o espaço de memória disponível num sistema computacional alvo.

- Tipicamente, a análise de um algoritmo...
  - Avalia o tempo de execução:
    - No caso médio: $T_{avg}(n)$
    - No pior caso: $T_{worst}(n)$
  - Face ao tamanho dos dados de entrada (***n***)

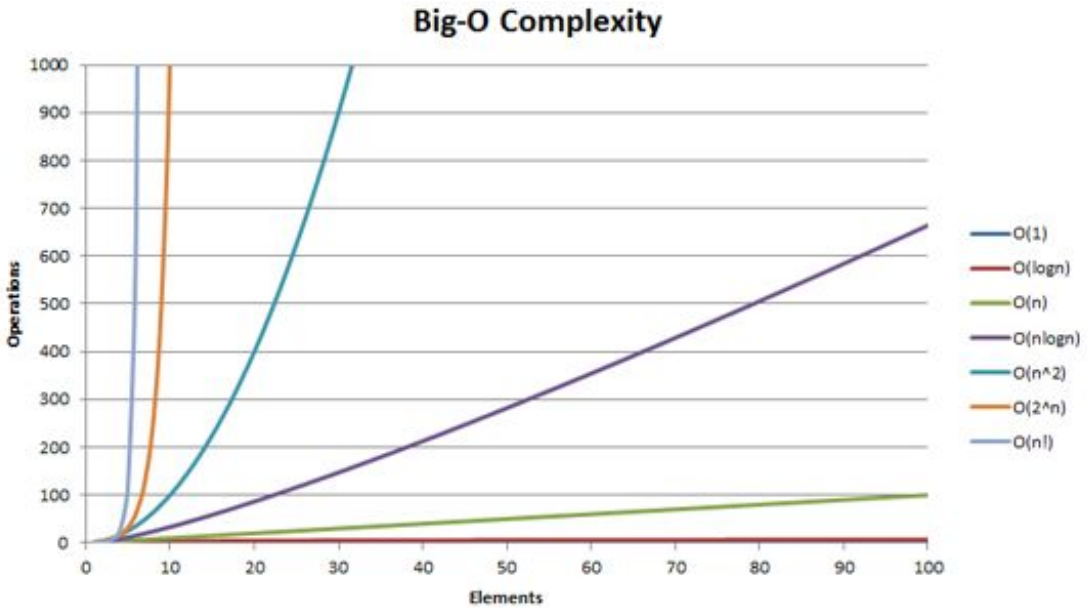
ANÁLISE DE COMPLEXIDADE DE ALGORITMOS

- Tipicamente, quando ***n*** é baixo, os algoritmos requerem poucos recursos.

- Sendo assim, o importante é comparar as funções em termos das suas **taxas de crescimento relativas**.
  - Perceber o que acontece para valores elevados de ***n***.

ANÁLISE DE COMPLEXIDADE DE ALGORITMOS

- Notação "Big-O"
  - Notação utilizada para **classificar e comparar funções** em termos das suas **taxas de crescimento relativas**
  - É usada em análise de complexidade de algoritmos
    - em particular para comparar o tempo de execução



ANÁLISE DE COMPLEXIDADE DE ALGORITMOS

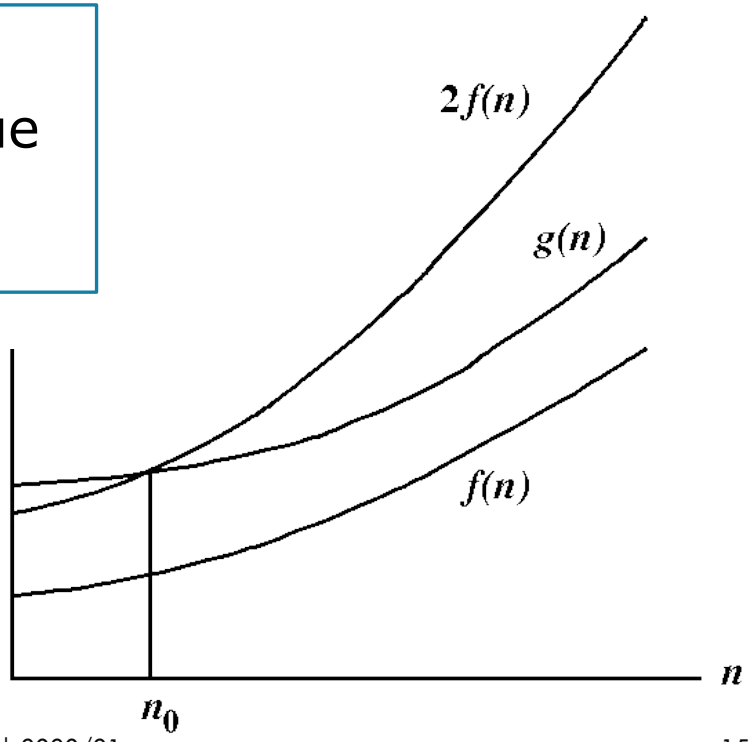
- Definição:

  Diz-se que...

$T(n) = O(f(n))$ se
existirem constantes c e n_0 tais que
$T(n) \leq c \times f(n)$ quando $n \geq n_0$.

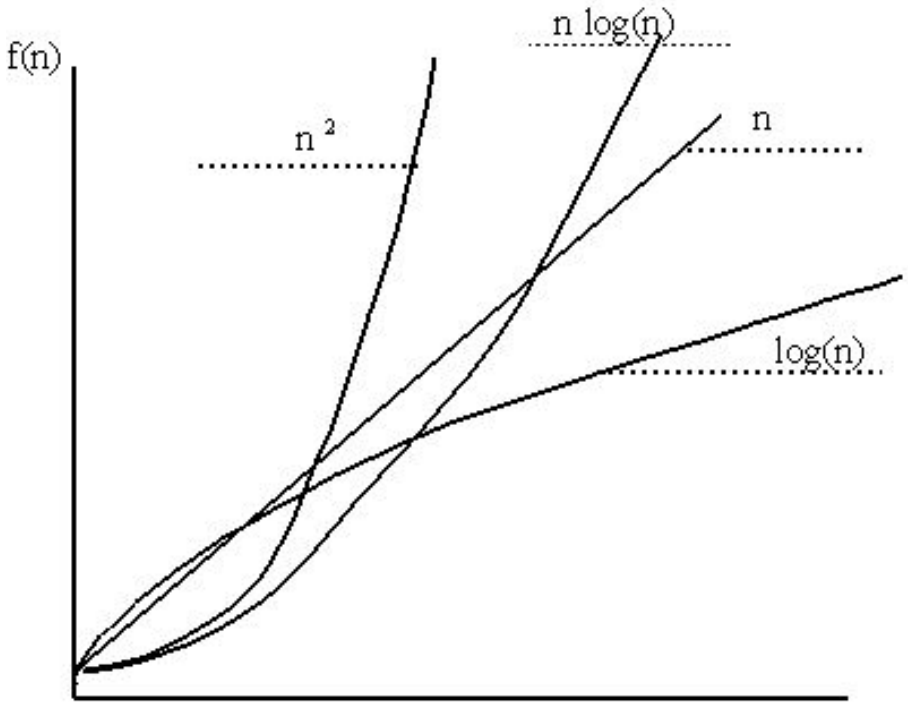
- Nesta figura...

  g(n) = ***O(***f(n)***)***



TAXAS DE CRESCIMENTO TÍPICAS

Função	Nome
	(crescem mais lentamente)
C	Constante
$\log n$	Logarítmica
$\text{Log}^2 n$	
n	Linear
$n \log n$	
n^2	Quadrática
n^3	Cúbica
2^n	Exponencial
	(crescem mais rapidamente)



TAXAS DE CRESCIMENTO TÍPICAS

n	$O(1)$	$O(\log_2 n)$	$O(n)$	$O(n \log_2 n)$	$O(n^2)$
10^2	1 μsec	1 μsec	1 μsec	1 μsec	1 μsec
10^3	1 μsec	1.5 μsec	10 μsec	15 μsec	100 μsec
10^4	1 μsec	2 μsec	100 μsec	200 μsec	10 msec
10^5	1 μsec	2.5 μsec	1 msec	2.5 msec	1 sec
10^6	1 μsec	3 μsec	10 msec	30 msec	1.7 min
10^7	1 μsec	3.5 μsec	100 msec	350 msec	2.8 hr
10^8	1 μsec	4 μsec	1 sec	4 sec	11.7 d

n	$O(n^2)$	$O(2^n)$
100	1 μsec	1 μsec
110	1.2 μsec	1 msec
120	1.4 μsec	1 sec
130	1.7 μsec	18 min
140	2.0 μsec	13 d
150	2.3 μsec	37 yr
160	2.6 μsec	37,000 yr

PROPRIEDADES IMPORTANTES

Se tivermos:

- $T_1(n) = O(f(n))$
- $T_2(n) = O(g(n))$

...então:

- $T_1(n) + T_2(n) = \mathbf{max(}\ O(f(n))\ \mathbf{,}\ O(g(n))\ \mathbf{)}$
- $T_1(n) * T_2(n) = \mathbf{O(}\ f(n)*g(n)\ \mathbf{)}$

ANÁLISE DE COMPLEXIDADE DE ALGORITMOS

- Para **simplificação** da análise, considera-se que:
  - As instruções são executadas sequencialmente
  - Qualquer instrução requer 1 unidade de tempo
  - Não existe limite de memória

- **Não se consideram**:
  - Coeficientes constantes
  - Termos de baixo grau

EXEMPLO: CÁLCULO DE $\sum_{i=1}^{n} i^3$

```
unsigned int sum( int n )              // Linha:     Tempo*:
{
    unsigned int i, partial_sum;  // 1        0
    partial_sum = 0;              // 2        1
    for( i=1; i<=n; i++ )         // 3        2n+2
        partial_sum += i*i*i;     // 4        3n
    return partial_sum;               // 5        1
}
```

Sendo o tempo de invocação e retorno da função ignorado:

- o tempo* total é **5n+4**
- pelo que a complexidade desta função é ***O(n)***

*em unidades de tempo

SIMPLIFICAÇÃO DA ANÁLISE

- Obviamente, para algoritmos mais complexos, é impraticável fazer análises tão pormenorizadas como a anterior.
- No entanto, uma vez que usamos a notação "Big-O", existem **simplificações** que não afetam o resultado final.
- Existem, então, **4 regras gerais** que se podem utilizar para tornar a análise mais rápida.

REGRAS GERAIS DE SIMPLIFICAÇÃO

Regra 1

- Ciclos FOR:

O tempo de execução de um ciclo FOR é, no máximo, o tempo de execução dentro do ciclo (incluindo testes) ***vezes*** *o número de iterações.*

REGRAS GERAIS DE SIMPLIFICAÇÃO

Regra 2

- Ciclos FOR embutidos:

*Devem ser analisados de dentro para fora. O tempo de execução total de uma instrução dentro de um grupo de ciclos embutidos é o da instrução **vezes** o produto dos tamanhos de todos os ciclos que a contêm.*

O seguinte exemplo tem tempo de execução $O(n^2)$:

```
for( i=0; i<n; i++ )
    for( j=0; j<n; j++ )
        k++;
```

REGRAS GERAIS DE SIMPLIFICAÇÃO

Regra 3

- Instruções consecutivas:

Devem simplesmente ser adicionadas, o que significa que basta considerar a de grau maior.

O seguinte exemplo inclui tempo de execução $O(n)$ seguido de $O(n^2)$, pelo que simplifica para $\boldsymbol{O(n^2)}$:

```
for( i=0; i<n; i++)
    a[i] = 0;
for( i=0; i<n; i++ )
    for( j=0; j<n; j++ )
        a[i] += a[j] + i + j;
```

REGRAS GERAIS DE SIMPLIFICAÇÃO

Regra 4

```
if( cond )
    ... // bloco S1: O(n)
else
    ... // bloco S2: O(n^2)
```

- Fluxo condicional IF/ELSE:

Deve-se considerar o tempo de execução do teste (`cond`), mais o tempo de execução do bloco de instruções alternativo com maior tempo de execução (`S2`).

REGRAS GERAIS DE SIMPLIFICAÇÃO

- Analisar de **dentro para fora** é a abordagem mais adequada para a grande maioria dos casos.
  - Na presença de invocação de funções, deve-se obviamente analisar estas primeiro.
  - A análise de funções recursivas que "mascaram" simples ciclos FOR são tipicamente triviais.

Exemplo *O(n)*:

```
unsigned int factorial( unsigned int n )
{
    if(n<=1)   return 1;
    else       return (n * factorial(n-1));
}
```

ESTRATÉGIAS DE CONCEPÇÃO DE ALGORITMOS

Com base em slides da autoria
da Professora Maria Cristina Ribeiro

DIVISÃO E CONQUISTA

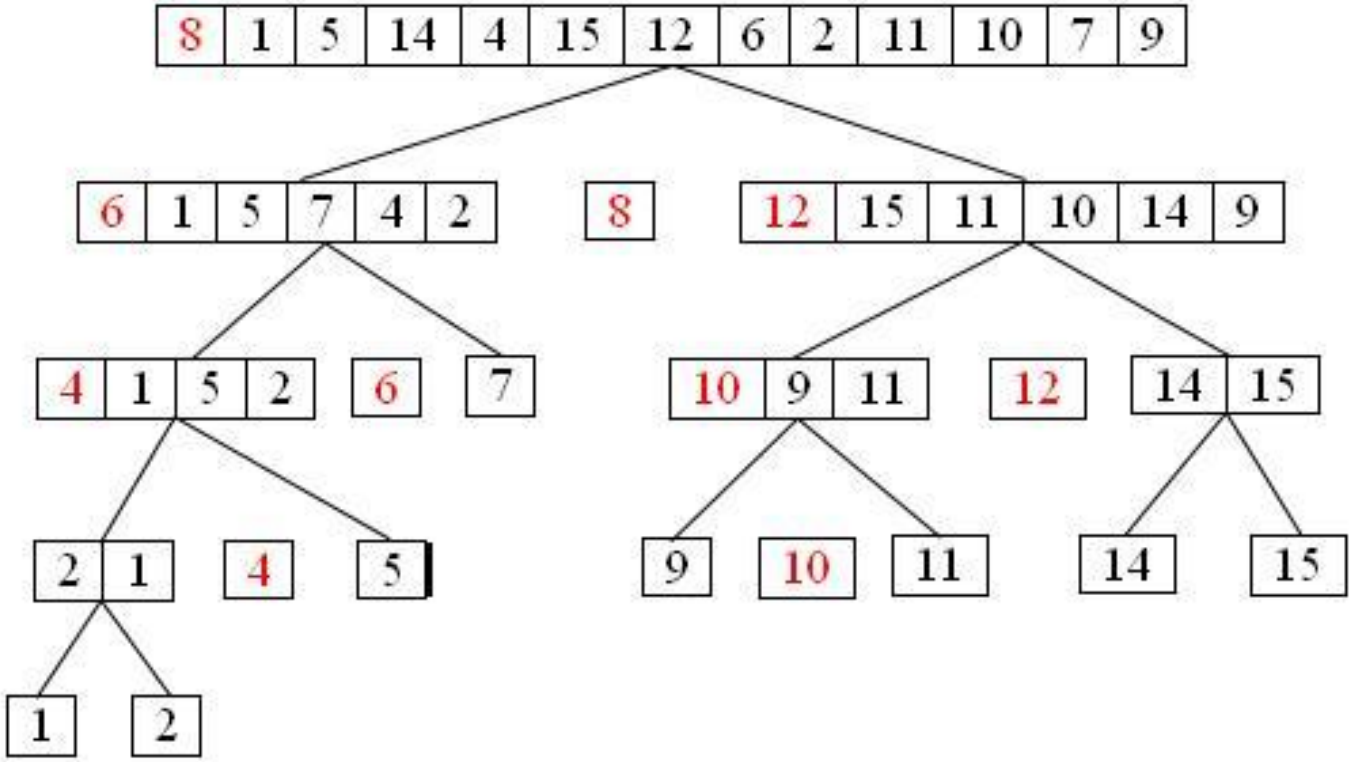
- Divisão:
  resolver recursivamente problemas mais pequenos
  (até ao caso base)
- Conquista:
  solução do problema original é formada com as soluções dos subproblemas
  - Há divisão quando o algoritmo tem pelo menos 2 chamadas recursivas no corpo
  - Subproblemas devem ser disjuntos

DIVISÃO E CONQUISTA

- Exemplos de algoritmos:
  - Travessia de árvores em tempo linear:
    - processar árvore esquerda
    - visitar nó
    - processar árvore direita
  - Aplicação em algoritmos de ordenação:
    - **mergesort**: ordenar 2 subsequências e juntá-las
    - **quicksort**: ordenar elementos menores e maiores que *pivot*, concatenar

DIVISÃO E CONQUISTA: QUICKSORT

A representação gráfica do processo de ordenação do QuickSort revela a sua estratégia de Divisão e Conquista:



DIVISÃO E CONQUISTA: QUICKSORT

```
void quickSortIter(int v[], int left, int right) {
    int i, j, tamanho = right-left+1;
    if(tamanho<2) // com tamanho 0 ou 1 esta ordenado
        return;
    else {
        int pos = rand()%tamanho;  // determina pivot
        swap(&v[pos], &v[right]);  // coloca pivot no fim
        i = left;   j = right-1;   // passo de partição
        while(1) {
            while (v[i] < v[right]) i++;
            while (v[right] < v[j]) j--;
            if (i < j) swap(&v[i], &v[j]);
            else break;
        }
        swap(&v[i], &v[right]);  // repoe pivot
        quickSortIter(v, left, i-1);
        quickSortIter(v, i+1, right);
 }
```

DIVISÃO E CONQUISTA: QUICKSORT (MELHORADO)

```
void quickSortIter(int v[], int left, int right) {
    int i, j;
    if (right-left+1 <= 20)    // se vetor pequeno
        ordenacaoInsercao(v, left, right);
    else {
        int x = median3(v, left, right);  // x é o pivot
        i = left;   j = right-1;   // passo de partição
        while(1) {
            while (v[i] < x) i++;
            while (x < v[j]) j--;
            if (i < j) swap(&v[i], &v[j]);
            else break;
        }
        swap(&v[i], &v[right]);  // repoe pivot
        quickSortIter(v, left, i-1);
        quickSortIter(v, i+1, right);
    }
}
```

```
/* escolha do pivot */
int median3(int v[], int left, int right)
{
    int center = (left+right) /2;
    if (v[center] < v[left])
        swap(&v[left], &v[center]);
    if (v[right] < v[left])
        swap(&v[left], &v[right]);
    if (v[right] < v[center])
        swap(&v[center], &v[right]);
    /* coloca pivot na posicao right */
    swap(&v[center], &v[right]);
    return v[right];
}
```

DIVISÃO E CONQUISTA *VS.* PROGRAMAÇÃO DINÂMICA

- **Divisão e conquista**:
  - problema é partido em subproblemas que se resolvem separadamente;
  - solução obtida por combinação das soluções
- **Programação dinâmica**:
  - resolvem-se os problemas de pequena dimensão e guardam-se as soluções;
  - solução de um problema é obtida combinando as de problemas de menor dimensão
- Divisão e conquista é **top-down**
- Programação dinâmica é **bottom-up**

PROGRAMAÇÃO DINÂMICA

- Abordagem usual em Investigação Operacional
  - "Programação" é aqui usada com o sentido de "formular restrições ao problema que tornam um método aplicável"

- Quando é aplicável a programação dinâmica:

  Estratégia óptima para resolver um problema continua a ser óptima quando este é subproblema de um problema de maior dimensão

PROGRAMAÇÃO DINÂMICA: FIBONACCI

- Problemas expressos recursivamente
  - (podem sempre ser formulados iterativamente)

- Exemplo**:**
  - Números de Fibonacci:

    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, …

    Ou seja:

$$F(n) = \begin{cases} 0, & \text{se } n = 0; \\ 1, & \text{se } n = 1; \\ F(n-1) + F(n-2) & \text{outros casos.} \end{cases}$$

PROGRAMAÇÃO DINÂMICA: FIBONACCI

```
/* Fibonacci RECURSIVO */

long int fib(int n) {
   if(n<=1)
      return n;
   else
      return fib(n-1) + fib(n-2);
}
```

Nota:
n <= 92 (limite de representação)

```
/* Fibonacci ITERATIVO */

long int fibonacci(int n) {
   int i;;
   long int nextToLast = 0;
   long int last = 1, answer = 1;

   if(n<=1)
      return n;

   for(i=2; i<=n; i++) {
      answer = last + nextToLast;
      nextToLast = last;
      last = answer;
   }
   return answer;
}
```

PROGRAMAÇÃO DINÂMICA: FIBONACCI

- Formulação recursiva: algoritmo exponencial
- Formulação iterativa: algoritmo linear

Problema na formulação recursiva:

repetição de chamadas iguais