Recursividade e Análise de Complexidade

Estrutura de Dados

Universidade Federal de Juiz de Fora
Departamento de Ciência da Computação

Conteúdo

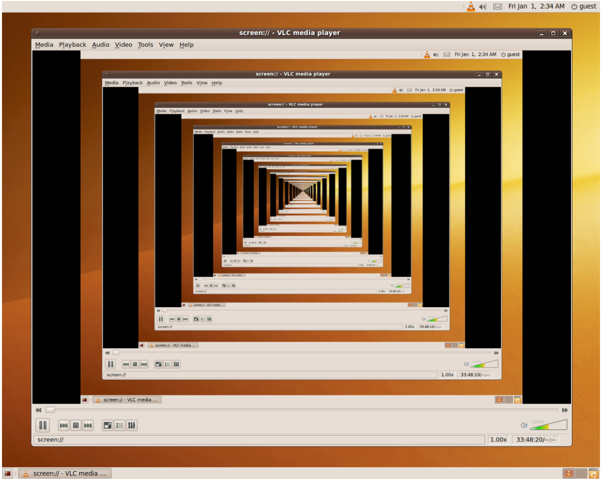
Recursividade

Recursividade

- Recursão é uma abordagem de solução de problemas que pode ser utilizada para gerar soluções simples a certos tipos de problemas que seriam difíceis de resolver de outra maneira.
- Em um algoritmo recursivo, o problema original é dividido e um ou mais versões simples de si mesmo.
- P. ex., se a solução do problema original envolve n itens, o pensamento recursivo deve dividi-lo em 2 problemas: um envolvendo $n-1$ itens e um outro envolvendo um único item.
- O problema com $n-1$ itens poderia ser dividido novamente em um envolvendo $n-2$ itens e um outro envolvendo apenas um único item, e assim por diante.
- Se a solução de todos os problemas com um item é trivial, podemos construir a solução do problema original a partir das soluções dos problemas mais simples.

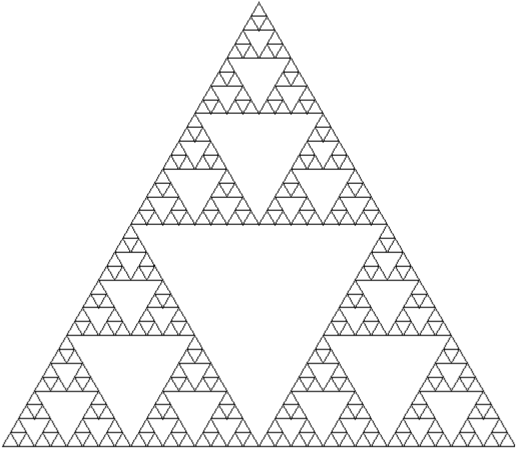
Recursividade

- Em imagens...



Recursividade

- Em imagens...



Recursividade

Algoritmo recursivo genérico

- Considere que n seja um inteiro que representa, de alguma forma, o tamanho de um determinado problema que deseja-se resolver utilizando um algoritmo recursivo.
- Descrição geral de um algoritmo recursivo:

1. Se o problema puder ser resolvido diretamente para o valor atual de n
   - **Resolva-o**
2. Senão
   - Aplique **recursivamente** o algoritmo a um ou mais problemas (iguais ao original), porém envolvendo valores menores de n.
3. **Combine** as soluções dos problemas menores para obter a solução do problema original.

Recursividade

Algoritmo recursivo genérico

- No `se` tem-se um teste para o que é chamado de **caso base**: o valor de n para o problema é suficientemente pequeno e, portanto, o problema pode ser resolvido facilmente.
- No `senão`, tem-se o **passo recursivo**, porque aqui aplica-se o algoritmo recursivamente, isto é, para resolver o mesmo problema, mas para uma instância menor (valor de n menor).
- Sempre que ocorrer uma divisão, revisita-se o caso base para cada novo problema a fim de verificar se esse é um caso base ou um caso recursivo.

Recursividade

- Uma solução recursiva tem as seguintes características:
  - Deve-se conhecer a solução direta para o caso base (para um valor pequeno de n)
  - Um problema de um dado tamanho (digamos, n) pode ser dividido em uma ou mais versão menores do mesmo problema (caso recursivo).
- Será visto, daqui em diante, exemplos de soluções recursivas em C++ para alguns problemas comuns de programação.
  - Fatorial
  - Sequência de Fibonacci
  - Máximo divisor comum
  - etc
- Note que, em implementações recursivas, as funções que resolvem um determinado problema devem chamar a si mesmo.

Recursividade

Fatorial

- **Definição:** o fatorial de um inteiro não-negativo n, representado por $n!$, é o produto de todos os inteiros positivos menores ou iguais a n.
- Adota-se por definição que $0! = 1$.
- Exemplos:
  - $0! = 1$
  - $1! = 1$
  - $2! = 2 \times 1$
  - $3! = 3 \times 2 \times 1$
  - $4! = 4 \times 3 \times 2 \times 1$
  - $5! = 5 \times 4 \times 3 \times 2 \times 1$
- Note que
  - $5! = 5 \times 4!$

Recursividade

Fatorial

- Implementação **iterativa** (não-recursiva)

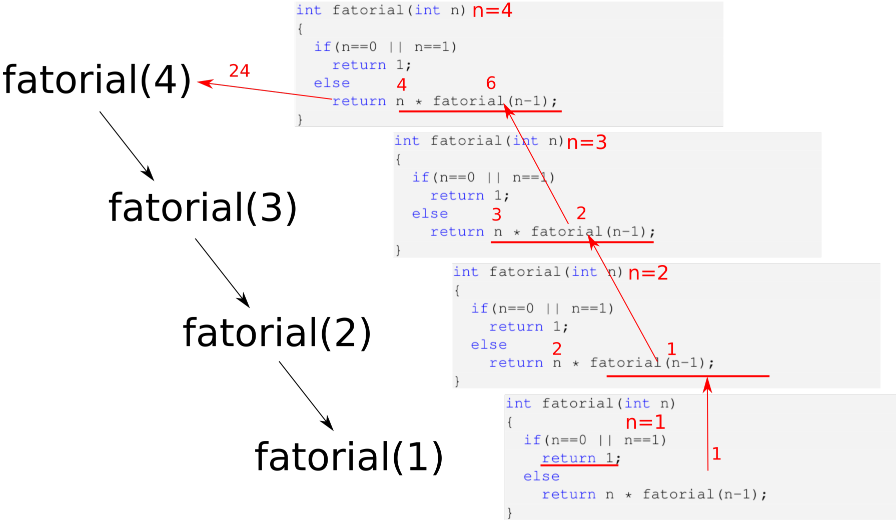
```
int fatorial(int n)
{
  int fat = 1;
  for(int i=1; i<=n; i++)
    fat = fat * i;
  return fat;
}
```

- Implementação **recurvisa**

```
int fatorial(int n)
{
  if(n==0 || n==1)
    return 1;
  else
    return n * fatorial(n-1);
}
```

Recursividade

Fatorial



- Desenvolver uma função recursiva para calcular x^n.
- Para simplificar, inicialmente, considere que $n \geq 0$.
- Definição recursiva do problema:

$$x^n = \begin{cases} 1, & \text{se } n = 0 \\ x \times x^{n-1}, & \text{se } n > 0 \end{cases}$$

Recursividade

Potência

- Implementação iterativa

```
float pot(float x, int n)
{
  float r=1.0;
  for(int i=1; i<=n; i++)
    r = r * x;
  return r;
}
```

- Implementação **recursiva**

```
float pot(float x, int n)
{
  if(n==0)
    return 1.0;
  else
    return x * pot(x,n-1);
}
```

Recursividade

Potência

- Se n pode ser negativo, então

$$x^n = \begin{cases} 1, & \text{se } n = 0 \\ \frac{1}{x^n}, & \text{se } n < 0 \\ x \times x^{n-1}, & \text{se } n > 0 \end{cases}$$

- E assim

```
float pot(float x, int n)
{
  if(n==0)
    return 1.0;
  else if(n < 0)
    return 1.0/pot(x,-n);
  else
    return x * pot(x,n-1);
}
```

Recursividade

Potência

- Pode-se ainda pensar na seguinte melhoria dessa função:

$$x^n = \begin{cases} 1, & \text{se } n = 0 \\ \frac{1}{x^n}, & \text{se } n < 0 \\ (x^2)^{\lfloor n/2 \rfloor}, & \text{se } n \text{ é par} \\ x(x^2)^{\lfloor n/2 \rfloor}, & \text{se } n \text{ é ímpar} \end{cases}$$

```
float pot(float x, int n)
{
  if(n==0)
    return 1.0;
  else if(n < 0)
    return 1.0/pot(x,-n);
  else if(n%2==0)
    return pot(x*x,n/2);
  else
    return x*pot(x*x,n/2);
}
```

Recursividade

MDC

- Calcular o máximo divisor comum (m.d.c.) entre m e n.
- Definição recursiva:

$$mdc(m,n) = \begin{cases} mdc(n,m), & \text{se } M < n \\ n, & \text{se } n \text{ é divisor de } m \\ mdc(n, m \bmod n), & \text{caso contrário} \end{cases}$$

```c
int mdc(int m, int n)
{
  if(m < n)
    return mdc(n, m);
  else if(m%n == 0) // caso base - conhecido
    return n;
  else              // passo recursivo
    return mdc(n, m%n);
}
```

- A sequência de Fibonacci é uma sequência de números inteiros, que começa com os números 0 e 1, na qual cada termo subsequente corresponde à soma dos dois anteriores.

$$0, 1, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{5}, \mathbf{8}, \mathbf{13}, \mathbf{21}, \mathbf{34}, \mathbf{55}, \mathbf{89}, \mathbf{144}, \mathbf{233}, \mathbf{377}, \mathbf{610}, \mathbf{987}, \ldots$$

- Seja $F_1 = 1$ o primeiro número. Se F_n é o n-ésimo número da sequência de Fibonacci, então, este pode ser definido como:
$$F_n = F_{n-1} + F_{n-2}$$

- Implementação **recursiva**

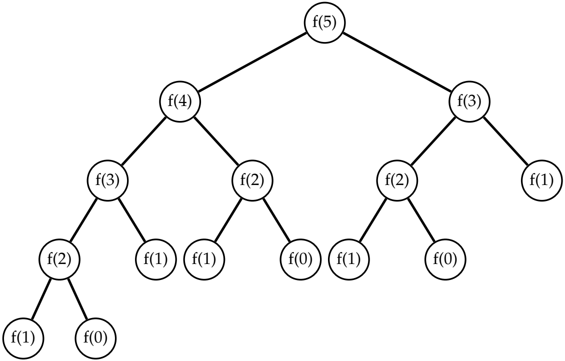
```
int fib(int n)
{
  if(n == 0 || n == 1)
    return n;
  else
    return fib(n-1) + fib(n-2);
}
```

- Note que no `else` dessa implementação recursiva, a função `fib` é chamada duas vezes.

Recursividade

Fibonacci

- Essa solução é muito ineficiente.
- Exemplo para $n = 5$ (foi usado f(n) no lugar de fib(n)):



- O problema com essa solução é que a função é chamada várias vezes para o mesmo valor (parâmetro). Ex: $F(2)$ é calculado 3 vezes.
- Para valores n maiores, a situação piora.

Recursividade

Fibonacci

- Uma outra implementação **recursiva**:

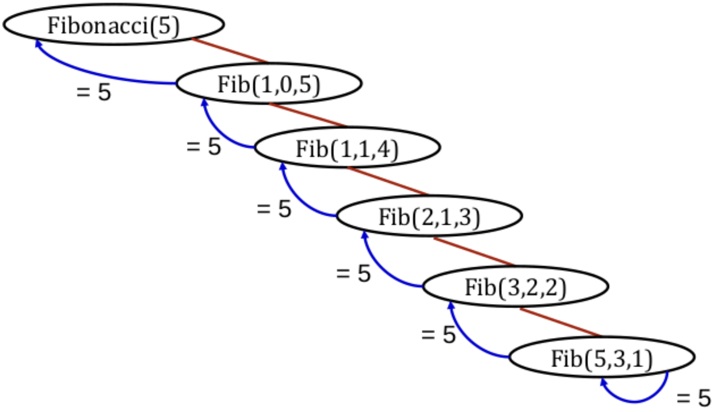
```
int fib(int atual, int anterior, int n)
{
  if(n == 1)
    return atual;
  else
    return fib(atual + anterior, atual, n-1);
}
```

- Para começar a execução deve-se usar a seguinte função

```
int fibonacci(int n)
{
  return fib(1, 0, n);
}
```

Recursividade
Fibonacci

- Essa versão é mais eficiente. Veja o exemplo para $n = 5$.



Recursividade

Intervalo

- Desenvolver uma função recursiva para imprimir todos os números inteiros no intervalo fechado de *a* até *b*.

```
void seq(int a, int b)
{
  if(a <= b)
  {
    cout << a << " ";
    seq(a+1, b);
  }
}
```

- Exemplo de como a função deve ser chamada

```
seq(0, 10)
```

- Saída

```
0 1 2 3 4 5 6 7 8 9 10
```

- Qual o resultado de inverter as seguintes instruções?

```
void seq(int a, int b)
{
  if(a <= b)
  {
    seq(a+1, b);
    cout << a << " ";
  }
}
```

Recursividade

Número primo

- Um número primo só é divisível por 1 e por ele mesmo.

```
bool ehPrimo(int p)
{
  return auxPrimo(p, 2);
}

bool auxPrimo(int p, int i)
{
  if (i==p)
    return true;

  if (p%i == 0)
    return false;

  return auxPrimo(p, i+1);
}
```

- Desenvolver uma função recursiva para calcular e retornar o maior valor de um vetor `v` com n números reais.
- Caso base: vetor com apenas 1 elemento, que é o maior.
- Passo recursivo: o maior elemento do vetor é ou o último elemento ou o maior elemento dentre os $n-1$ primeiros elementos do vetor.
- Isto é:

$$\max(v, n) = \begin{cases} v[0], & \text{se } n = 1 \\ v[n-1], & \text{se } v[n-1] > \max(v, n-1) \\ \max(v, n-1), & \text{caso contrário} \end{cases}$$

- Implementação **recursiva**

```
float max(float vet[], int n)
{
  if (n == 1)                  // caso base
    return vet[0];

  float x = max(vet, n-1); // passo recursivo

  if(x > vet[n-1])
    return x;
  else
    return vet[n-1] ;
}
```

Recursividade

Maior

- Outra implementação
- Função empacotadora

```
float max(float vet[], int n)
{
  return maxR(vet, 0, n);
}
```

- Função recursiva

```
float maxR(float vet[], int i, int n)
{
  if(i == n-1)
    return vet[i];

  float x = maxR(vet, i+1, n);

  return (x > vet[i]) ? x : vet[i];
}
```

Recursividade

Pares

- Desenvolver uma função recursiva para calcular e retornar a quantidade de valores pares de um vetor `v` com n números inteiros.

```
int pares(int vet[], int n)
{
  if(n == 1) // caso base
    if (vet[0] % 2 == 0)
      return 1;

  int x = pares(vet, n-1) ; // passo recursivo

  if (vet[n-1] % 2 == 0)
    return x+1;
  else
    return x;
}
```

Exercícios

1. Desenvolver uma função recursiva que recebe um número inteiro n e retorna o valor do somatório: $n + (n-1) + (n-2) + \ldots + 2 + 1$.
2. Desenvolver uma função recursiva que recebe um vetor de reais e seu tamanho n, calcular e retornar o menor valor do vetor.
3. Desenvolver uma função recursiva que recebe um vetor de reais e seu tamanho n, calcular e retornar a soma de todos os seus valores.
4. Desenvolver uma função recursiva para calcular e retornar a quantidade de valores pares de um vetor v com n números inteiros.
5. Desenvolver uma função recursiva para calcular e retornar uma *string* de caracteres contendo '0' e '1' correspondente à versão binária de um número inteiro positivo dado.

Introdução à Análise de Complexidade

Análise de Complexidade

- Algoritmos demandam tempo de execução e recursos (memória, espaço em disco, dispositivos externos, etc).
- Analisar a alocação de recursos que um certo algoritmo demanda é importante na escolha de soluções mais rápidas ou que ocupem menos espaço de memória, por exemplo.
- Bons programadores se preocupam em implementar algoritmos que demandam o mínimo de recursos e executem no menor tempo possível.
- Embora um programa possa ser analisado sob vários aspectos, destaca-se a seguir a análise relativa ao seu desempenho, especialmente, em relação a medida do seu tempo de computação.

Análise de Complexidade

- A análise de complexidade de algoritmos é uma ferramenta que permite estudar como um algoritmo se comporta quando os dados de entrada aumentam.
- Se o algoritmo é alimentado com uma outra entrada, como o algoritmo se comporta?
- Se o algoritmo leva 1 segundo para executar para uma entrada de tamanho 1000, como ele irá se comportar se o tamanho da entrada for duplicado?

Análise de Complexidade

- Seja um programa com n instruções. Então, o tempo total de execução do programa T é dado por

$$T = \sum_{i=1}^{n} (t_i \, n_i)$$

- onde:
  - t_i é o tempo de execução da instrução i
  - n_i é o número de vezes que a instrução i é executada
- Entretanto, como o tempo de execução da instrução i é sempre de difícil obtenção, avalia-se o tempo total considerando somente o número de vezes que a instrução é executada.
- Esse número é chamado de contagem de frequência ou simplesmente **frequência da instrução** i.

Análise de Complexidade

- Exemplos de determinação da frequência f de uma instrução.
- Comando que pertence a uma sequência simples: tem frequência $f = 1$.

```
x = x + 1;
```

- Se essa instrução pertencer a uma estrutura de repetição

```
for(i=1; i<=n; i++)
{
  // ...
  x = x + 1;
  // ...
}
```

- Nesse caso a instrução tem frequência:

$$f = \sum_{i=1}^{n} 1 = n$$

Análise de Complexidade

- Se a estrutura do exemplo anterior pertencer a outra estrutura de repetição:

```
for(j=1; j<=n; j++)
{
  // ...
  for(i=1; i<=n; i++)
  {
    // ...
    x = x + 1;
    // ...
  }
  // ...
}
```

- Nesse caso a instrução tem frequência:

$$f = \sum_{j=1}^{n} \left(\sum_{i=1}^{n} 1 \right) = \sum_{j=1}^{n} n = n^2$$

Análise de Complexidade

- A maior frequência encontrada em um programa é chamada de **ordem de grandeza** de crescimento de tempo do programa.
- A ordem de grandeza de um algoritmo é o principal parâmetro para análise do desempenho de sua execução.
- Seja N um parâmetro que caracteriza o tamanho de um problema.
- As ordens de grandeza mais comuns nos algoritmos são:
  - $O(1) \rightarrow$ constante
  - $O(\log_2 N) \rightarrow$ logaritmo
  - $O(N) \rightarrow$ linear
  - $O(N \log_2 N)$
  - $O(N^2) \rightarrow$ quadrática
  - $O(N^3) \rightarrow$ cúbica
  - $O(2^N) \rightarrow$ exponencial
  - $O(N!) \rightarrow$ fatorial

Análise de Complexidade

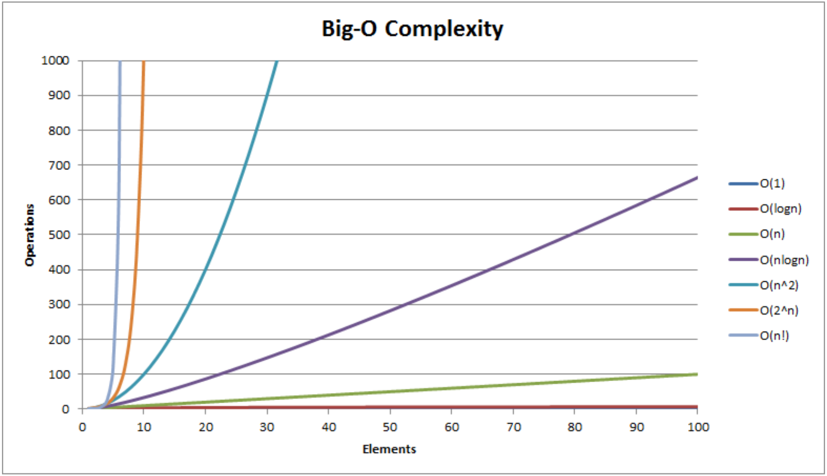


Figura: Ordens de grandeza.

Análise de Complexidade

- Exemplo: Analisar a solução iterativa de um algoritmo que leia um valor inteiro N, calcule e imprima o seu fatorial. Se N for negativo, imprimir uma mensagem de erro.

```
int n, c, fat = 1;
cout << "Digite n" << endl;
cin >> n;
if(n >= 0)
{
  for(c = 1; c<=n; c++)
    fat = fat * c;
  cout << "Fatorial = " << fat << endl;
}
else
  cout << "Valor negativo" << endl;
```

- O algoritmo será estudado para os seguintes casos:
  - $n < 0, n = 0, n = 1$ e $n > 1$

Análise de Complexidade

- Caso $n < 0$

```
int n, c, fat = 1;
cout << "Digite n" << endl;                    // 1
cin >> n;                                      // 1
if(n >= 0)                                     // 1
{
  for(c = 1; c<=n; c++)
    fat = fat * c;
  cout << "Fatorial = " << fat << endl;
}
else
  cout << "Valor negativo" << endl;            // 1
```

- Soma das frequências $= 4$

Análise de Complexidade

- Caso $n = 0$

```
int n, c, fat = 1;
cout << "Digite n" << endl;                    // 1
cin >> n;                                      // 1
if(n >= 0)                                     // 1
{
  for(c = 1; c<=n; c++)                        // 1
    fat = fat * c;
  cout << "Fatorial = " << fat << endl;        // 1
}
else
  cout << "Valor negativo" << endl;
```

- Soma das frequências $= 5$

Análise de Complexidade

- Caso $n = 1$

```
int n, c, fat = 1;
cout << "Digite n" << endl;                         // 1
cin >> n;                                           // 1
if(n >= 0)                                          // 1
{
  for(c = 1; c<=n; c++)                             // 2
    fat = fat * c;                                  // 1
  cout << "Fatorial = " << fat << endl;             // 1
}
else
  cout << "Valor negativo" << endl;
```

- Soma das frequências $= 7$

Análise de Complexidade

- Caso $n = 1$

```
int n, c, fat = 1;
cout << "Digite n" << endl;                   // 1
cin >> n;                                     // 1
if(n >= 0)                                    // 1
{
  for(c = 1; c<=n; c++)                       // n+1
    fat = fat * c;                            // n
  cout << "Fatorial = " << fat << endl;       // 1
}
else
  cout << "Valor negativo" << endl;
```

- Soma das frequências $= 2n + 5$

- Para valores suficientemente pequenos de n, qualquer algoritmo custa pouco para ser executado, mesmo os ineficientes.
- Nesse caso, a escolha de um algoritmo não é um problema crítico.
- É importante analisar algoritmos para grandes valores de n.
- Portanto, estuda-se o **comportamento assintótico** das funções de complexidade de um programa, isto é, o comportamento para grandes valores de n.

Análise de Complexidade

- De volta ao exemplo anterior do fatorial.
- No caso mais geral $n > 1$, a soma das frequências é de $2n + 5$.
- Como deseja-se estudar o comportamento apenas para n grande, pode-se desprezar as constantes e os termos de menor ordem.
- Assim, conclui-se que o programa possui complexidade linear, isto é, $O(n)$.

Análise de Complexidade

Exemplo - Algoritmo 1

- Analisar o tempo de processamento de um programa para calcular o seguinte somatório (série geométrica):

$$S = \sum_{i=0}^{n} x^i$$

```
float soma(int x, int n)
{
  int soma = 0;                   // 1
  for(int i=0; i<=n; i++)         // n+2
  {
    int prod = 1;                 // n+1
    for(int j=0; j<i; j++)
      prod = prod*x;              // ∑_{i=0}^{n} i
    soma = soma + prod;           // n+1
  }
  return soma;                    // 1
}
```

Análise de Complexidade

Exemplo - Algoritmo 1

- O tempo de processamento $T(n)$ desse programa é obtido somando-se a execução de todas instruções listadas anteriormente.

$$T(n) = 1 + (n+2) + (n+1) + (\sum_{i=0}^{n} i) + (n+1) + 1$$

$$T(n) = 6 + 3n + \sum_{i=0}^{n} i = 6 + 3n + \frac{n(n+1)}{2}$$

$$T(n) = \frac{n^2}{2} + \frac{7n}{2} + 6$$

- E assim, conclui-se que esse algoritmo é $O(n^2)$.

- Pode-se utilizar um algoritmo conhecido como **algoritmo de Horner** para realizar esse cálculo.

$$
\begin{aligned}
S = \sum_{i=0}^{n} x^i &= 1 + x + x^2 + x^3 + \ldots + x^n \\
&= 1 + x(1 + x + x^2 + \ldots + x^{n-1}) \\
&= 1 + x(1 + x(1 + x + \ldots + x^{n-2})) \\
&= \ldots \\
&= 1 + x(1 + x(1 + x(1 + \ldots (1 + x(1 + x))) \ldots))
\end{aligned}
$$

- Algoritmo de Horner

```
float somaHorner(int x, int n)
{
  int i, soma = 0;                  // 1
  for(i=0; i<=n; i++)               // n+2
  {
    soma = soma*x + 1;              // n+1
  }
  return soma;                      // 1
}
```

- O tempo de processamento é $T(n) = 2n + 5$
- Portanto, esse algoritmo é $O(n)$.

Análise de Complexidade

Exemplo - Algoritmo 3

- Fórmula fechada

$$S = \sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1}$$

- Para calcular a potência, usa-se `pot(int x, int n)` que possui complexidade $T(n) = \log_2 n + 2$

```
float soma(int x, int n)
{
  return (pot(x,n+1)-1)/(x-1);
}
```

- Portanto, esse algoritmo é $O(\log_2 n)$.

Análise de Complexidade

Exemplo

- Comparação dos três algoritmos
  - Algoritmo 1: $T(n) = \frac{n^2}{2} + \frac{7n}{2} + 6 \quad \Rightarrow \quad O(n^2)$
  - Algoritmo 2: $T(n) = 2n + 5 \quad \Rightarrow \quad O(n)$
  - Algoritmo 3: $T(n) = \log_2 n + 2 \quad \Rightarrow \quad O(\log_2 n)$

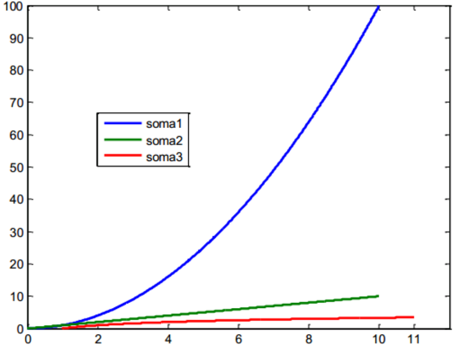


Figura: Comparação dos algoritmos.

Análise de Complexidade

- Tabela com tempos de execução.

FUNÇÃO DE COMPLEXIDADE	n (tamanho do problema)		
	20	40	60
n	0.0002 s	0.0004 s	0.0006 s
$n \log_2 n$	0.0009 s	0.0021 s	0.0035 s
n^2	0.0040 s	0.0160 s	0.0360 s
n^3	0.0800 s	0.6400 s	2.1600 s
2^n	10.0000 s	27 dias	3660 séculos
3^n	580 minutos	38550 séculos	$1.3*10^{14}$ séculos

- Algumas ordens de grandeza de complexidade tornam proibitivo a aplicabilidade do algoritmo, devendo ser usado apenas quando não se conheça solução de menor complexidade.

1. Qual a complexidade do algoritmo abaixo?

```
int maior(int n, int v[])
{
  int m = v[0];
  for (int i=1; i<n; i++ )
  {
    if( v[i] >= m ) {
      m = v[i];
    }
  }
  return m;
}
```

Análise de Complexidade

Exercícios

2. Qual a complexidade das funções `f`, `g` e `h`?

```
int f(int n){
  int i, soma=0;
  for(i=1; i<=n; ++i)
    soma += 1;
  return soma;
}

int g(int n){
  int i, soma=0;
  for(i=1; i<=n; ++i)
    soma += i + f(i);
  return soma;
}

int h(int n){
  return f(n) + g(n);
}
```

3. Apresente ao menos dois algoritmos para calcular x^n e discuta a complexidade da sua solução de cada algoritmo.

Análise de Complexidade

- Lembre-se da solução recursiva discutida na aula anterior.

```
float exp_rec(float x, int n)
{
  if(n < 0)
    return exp_rec(1.0/x, -n);
  else if(n == 0)
    return 1.0;
  else if(n == 1)
    return x;
  else if(n % 2 == 0)
    return exp_rec(x*x, n/2);
  else
    return x * exp_rec(x*x, (n-1)/2);
}
```

Análise de Complexidade

```
float exp_sq(float x, int n)
{
  if(n < 0) {
    x = 1.0/x;
    n = -n;
  }
  if(n == 0) return 1.0;
  float y = 1.0;
  while(n > 1) {
    if(n % 2 == 0){
      x = x*x;
      n = n/2;
    } else {
      y = x*y;
      x = x*x;
      n = (n-1)/2;
    }
  }
  return x*y;
}
```