Análise de algoritmos

Profª. Rose Yuri Shimizu

Eficácia x Eficiência

```c
int p1(int n)
{
    if (n == 1 || n == 2)
        return 1;
    else
        return p1(n - 1) + p1(n - 2);
}

```

```c
int p2(int n)
{
    int i, f_ant = 1, f_ant_ant = 1, f_atual = 1;
    for (i=3; i<=n; i=i+1)
    {
        f_atual = f_ant + f_ant_ant;
        f_ant_ant = f_ant;
        f_ant = f_atual;
    }
    return f_atual;
}

```

Eficácia x Eficiência

```
int p1(int n)
{
    if (n == 1 || n == 2) // <- p1(5) verifica
        return 1;
    else
        return p1(n - 1) + p1(n - 2);
}
```

```
        p1(5)
          |
    (n==1||n==2)?
```

```
     stack

|                |
|                |
|                |
|                |
|    p1(5)       |
```

Eficácia x Eficiência

```
int p1(int n)
{
    if (n == 1 || n == 2)
        return 1;
    else
        return p1(n - 1) + p1(n - 2);
        //    ___________ p1(5) chama p1(4)
}

```

```
          _______ p1(5)
         /
      p1(4)

```

```
|                                  |
|                                  |
|                                  |
|    p1(4)                         |
|    p1(5) -> p1(4)                |

```

Eficácia x Eficiência

```
int p1(int n)
{
    if (n == 1 || n == 2) //<- p1(4) verifica
        return 1;
    else
        return p1(n - 1) + p1(n - 2);
}

```

```
        _______ p1(5)
       /
    p1(4)
      |
(n==1||n==2)?

```

```
|                           |
|                           |
|                           |
|   p1(4)                   |
|   p1(5) -> p1(4)          |

```

Eficácia x Eficiência

```
    int p1(int n)
    {
        if (n == 1 || n == 2)
            return 1;
        else
            return p1(n - 1) + p1(n - 2);
            //    ___________ p1(4) chama (p1(3)
    }

```

```
           _______ p1(5)
         /
       p1(4)
      /
    p1(3)

```

```
|                                  |
|                                  |
|    p1(3)                         |
|    p1(4) -> p1(3)                |
|    p1(5) -> p1(4)                |

```

Eficácia x Eficiência

```
    int p1(int n)
    {
        if (n == 1 || n == 2) //<- p1(3) verifica
            return 1;
        else
            return p1(n - 1) + p1(n - 2);
    }

```

```
          _______ p1(5)
         /
       p1(4)
       /
    p1(3)
      |
(n==1||n==2)?

```

```
|                                      |
|                                      |
|    p1(3)                             |
|    p1(4) -> p1(3)                    |
|    p1(5) -> p1(4)                    |

```

Eficácia x Eficiência

```
    int p1(int n)
    {
        if (n == 1 || n == 2)
            return 1;
        else
            return p1(n - 1) + p1(n - 2);
            //      __________ p1(3) chama p1(2)
    }

```

```
            _______ p1(5)
           /
         p1(4)
        /
      p1(3)
     /
p1(2)

```

```
|                                   |
|     p1(2)                         |
|     p1(3) -> p1(2)                |
|     p1(4) -> p1(3)                |
|     p1(5) -> p1(4)                |

```

Eficácia x Eficiência

```
    int p1(int n)
    {
        if (n == 1 || n == 2) //<- p1(2) verifica
            return 1; //<- p1(2) devolve
        else
            return p1(n - 1) + p1(n - 2);
    }

```

```
          _______ p1(5)
         /
       p1(4)
      /
    p1(3)
    /
p1(2)
  |
(n==1||n==2)?

```

```
|                              |
|    p1(2) ->    1             |
|    p1(3) -> p1(2)            |
|    p1(4) -> p1(3)            |
|    p1(5) -> p1(4)            |

```

Eficácia x Eficiência

```
    int p1(int n)
    {
        if (n == 1 || n == 2)
            return 1;
        else
            return p1(n - 1) + p1(n - 2);
            //                  ———————— p1(3) chama p1(1)
    }

```

```
                ________ p1(5)
              /
          p1(4)
         /
      p1(3)
     /     \
    1       p1(1)

```

```
|                                   |
|    p1(1)                          |
|    p1(3) -> 1 + p1(1)             |
|    p1(4) -> p1(3)                 |
|    p1(5) -> p1(4)                 |

```

Eficácia x Eficiência

```
int p1(int n)
{
    if (n == 1 || n == 2) //<- p1(1) verifica
        return 1; //<- p1(1) devolve
    else
        return p1(n - 1) + p1(n - 2);
}

```

```
          _______ p1(5)
         /
      p1(4)
     /
   p1(3)
  /     \
 1       p1(1)
          |
   (n==1||n==2)?

```

```
|                                |
|    p1(1) ->          1         |
|    p1(3) ->     1  +  p1(1)    |
|    p1(4) -> p1(3)              |
|    p1(5) -> p1(4)              |

```

Eficácia x Eficiência

```c
int p1(int n)
{
    if (n == 1 || n == 2)
        return 1;
    else
        return p1(n - 1) + p1(n - 2); //<- p1(3) devolve
}

```

```
          _______ p1(5)
         /
      p1(4)
     /
  p1(3)
 /     \
1       1

```

```
|                                   |
|                                   |
|    p1(3) ->    1   +     1        |
|    p1(4) -> p1(3)                 |
|    p1(5) -> p1(4)                 |

```

Eficácia x Eficiência

```
    int p1(int n)
    {
        if (n == 1 || n == 2)
            return 1;
        else
            return p1(n - 1) + p1(n - 2);
            //                     __________ p1(4) chama p1(2)
    }

```

```
           _______ p1(5)
          /
       p1(4)
      /      \
     2       p1(2)

```

```
|                                    |
|                                    |
|    p1(2)                           |
|    p1(4) ->    2  +   p1(2)        |
|    p1(5) -> p1(4)                  |

```

Eficácia x Eficiência

```
int p1(int n)
{
    if (n == 1 || n == 2) //<- p1(2) verifica
        return 1; //<- p1(2) devolve
    else
        return p1(n - 1) + p1(n - 2);
}
```

```
          _______ p1(5)
         /
       p1(4)
      /     \
     2      p1(2)
             |
      (n==1||n==2)?
```

```
|                                   |
|                                   |
|   p1(2) ->             1          |
|   p1(4) ->        2  + p1(2)      |
|   p1(5) -> p1(4)                  |
```

Eficácia x Eficiência

```
    int p1(int n)
    {
        if (n == 1 || n == 2)
            return 1;
        else
            return p1(n - 1) + p1(n - 2); //<- p1(4) devolve
    }

```

```
            _______ p1(5)
           /
       p1(4)
      /     \
     2       1

```

```
|                                           |
|                                           |
|                                           |
|    p1(4) ->    2  +    1                  |
|    p1(5) -> p1(4)                         |

```

Eficácia x Eficiência

```
    int p1(int n)
    {
        if (n == 1 || n == 2)
            return 1;
        else
            return p1(n - 1) + p1(n - 2);
            //                 ———————— p1(5) chama p1(3)
    }
```

```
        _______ p1(5) _______
       /                     \
       3                      p1(3)
```

```
|                                  |
|                                  |
|                                  |
|    p1(3)                         |
|    p1(5) ->    3  +  p1(3)       |
```

Eficácia x Eficiência

```
    int p1(int n)
    {
        if (n == 1 || n == 2) //<- p1(3) verifica
            return 1;
        else
            return p1(n - 1) + p1(n - 2);
    }

```

```
     _______ p1(5) _______
    /                     \
    3                      p1(3)
                             |
                      (n==1||n==2)?

```

```
|                                  |
|                                  |
|                                  |
|    p1(3)                         |
|    p1(5) ->    3  +  p1(3)       |

```

Eficácia x Eficiência

```
int p1(int n)
{
    if (n == 1 || n == 2)
        return 1;
    else
        return p1(n - 1) + p1(n - 2);
        //    ______  p1(3) chama p1(2)
}

```

```
   _______ p1(5) _______
  /                     \
 3                      p1(3)
                        /
                     p1(2)

```

```
|                             |
|                             |
|    p1(2)                    |
|    p1(3) -> p1(2)           |
|    p1(5) ->   3  +  p1(3)   |

```

Eficácia x Eficiência

```
    int p1(int n)
    {
        if (n == 1 || n == 2) //<- p1(2) verifica
            return 1; //<- p1(2) devolve
        else
            return p1(n - 1) + p1(n - 2);
    }

```

```
    _______ p1(5) _______
   /                     \
   3                      p1(3)
                         /
                      p1(2)
                        |
                  (n==1||n==2)?

```

```
|                                  |
|                                  |
|    p1(2) ->    1                 |
|    p1(3) -> p1(2)                |
|    p1(5) ->    3  +   p1(3)      |

```

Eficácia x Eficiência

```
int p1(int n)
{
    if (n == 1 || n == 2)
        return 1;
    else
        return p1(n - 1) + p1(n - 2);
        //                 --------- p1(3) chama p1(1)
}

```

```
   _______ p1(5) _______
  /                      \
 3                      p1(3)
                       /     \
                      1      p1(1)

```

```
|                                   |
|                                   |
|    p1(1)                          |
|    p1(3) ->    1   +   p1(1)      |
|    p1(5) ->    3   +   p1(3)      |

```

Eficácia x Eficiência

```
int p1(int n)
{
    if (n == 1 || n == 2) //<- p1(1) verifica
        return 1; //<- p1(1) devolve
    else
        return p1(n - 1) + p1(n - 2);
}

```

```
      _______ p1(5) _______
     /                     \
    3                      p1(3)
                          /     \
                         1      p1(1)
                                  |
                          (n==1||n==2)?

```

```
|                                  |
|                                  |
|   p1(1) ->              1        |
|   p1(3) ->     1   +  p1(1)      |
|   p1(5) ->     3   +  p1(3)      |

```

Eficácia x Eficiência

```
    int p1(int n)
    {
        if (n == 1 || n == 2)
            return 1;
        else
            return p1(n - 1) + p1(n - 2); //<- p1(3) devolve
    }

```

```
      ______ p1(5) ______
     /                   \
   3                      p1(3)
                         /     \
                        1       1

```

```
|                                    |
|                                    |
|                                    |
|    p1(3) ->    1  +     1          |
|    p1(5) ->    3  +   p1(3)        |

```

Eficácia x Eficiência

```
int p1(int n)
{
    if (n == 1 || n == 2)
        return 1;
    else
        return p1(n - 1) + p1(n - 2); //<- p1(5) devolve
}

```

```
     ______ p1(5) ______
    /                   \
   3                     2

```

```
|                                |
|                                |
|                                |
|                                |
|    p1(5) ->   3  +    2        |

```

Eficácia x Eficiência

```c
#include <stdio.h>

int p1(int n)
{
    if (n == 1 || n == 2)
        return 1;
    else
        return p1(n - 1) + p1(n - 2);
}

int main(){

    printf("%d\n", p1(5)); //5

    return 0;
}

```

Eficácia x Eficiência

```
              ______ p1(5) ______
             /                   \
          p1(4)                 p1(3)
         /     \               /     \
     p1(3)     p1(2)       p1(2)     p1(1)
     /   \       |           |         |
  p1(2) p1(1)    1           1         1
   |      |
   1      1
```

Eficácia x Eficiência

```
int p2(int n) //<- chamada da funcao
{
    int i, f_ant = 1, f_ant_ant = 1, f_atual = 1;
    for (i=3; i<=n; i=i+1)
    {
        f_atual = f_ant + f_ant_ant;
        f_ant_ant = f_ant;
        f_ant = f_atual; //atual é o anterior do próximo
    }
    return f_atual;
}

p2(5)


```

Eficácia x Eficiência

```
int p2(int n)
{
    int i, f_ant = 1, f_ant_ant = 1, f_atual = 1; //<-
    for (i=3; i<=n; i=i+1)
    {
        f_atual = f_ant + f_ant_ant;
        f_ant_ant = f_ant;
        f_ant = f_atual;
    }
    return f_atual;
}

p2(5)
  |____ f_atual=?, f_ant=1, f_ant_ant=1
```

Eficácia x Eficiência

```
int p2(int n)
{
    int i, f_ant = 1, f_ant_ant = 1, f_atual = 1;
    for (i=3; i<=n; i=i+1) //<-
    {
        f_atual = f_ant + f_ant_ant;
        f_ant_ant = f_ant;
        f_ant = f_atual;
    }
    return f_atual;
}

p2(5)
  |____ f_atual=?, f_ant=1, f_ant_ant=1
  |____ for


```

Eficácia x Eficiência

```
int p2(int n)
{
    int i, f_ant = 1, f_ant_ant = 1, f_atual = 1;
    for (i=3; i<=n; i=i+1)
    {
        f_atual = f_ant + f_ant_ant; //<-
        f_ant_ant = f_ant;
        f_ant = f_atual;
    }
    return f_atual;
}

p2(5)
  |____ f_atual=?, f_ant=1, f2=1
  |____ for
     |   |____ i=3 -> f_atual = 1+1 = 2, f_ant_ant = 1, f_ant = 2
     |_______________________________________________________________|

```

Eficácia x Eficiência

```
int p2(int n)
{
    int i, f_ant = 1, f_ant_ant = 1, f_atual = 1;
    for (i=3; i<=n; i=i+1)
    {
        f_atual = f_ant + f_ant_ant;
        f_ant_ant = f_ant;
        f_ant = f_atual;
    }
    return f_atual;
}

p2(5)
  |____ f_atual=?, f_ant_ant=1, f_ant=1
  |____ for
     |   |____ i=3 -> f_atual = 1+1 = 2, f_ant_ant = 1, f_ant = 2
     |   |____ i=4 -> f_atual = 1+2 = 3, f_ant_ant = 2, f_ant = 3
     |_________________________________________________________|


```

Eficácia x Eficiência

```
int p2(int n)
{
    int i, f_ant = 1, f_ant_ant = 1, f_atual = 1;
    for (i=3; i<=n; i=i+1)
    {
        f_atual = f_ant + f_ant_ant;
        f_ant_ant = f_ant;
        f_ant = f_atual;
    }
    return f_atual;
}

p2(5)
  |_____ f_atual=?, f1=1, f2=1
  |_____ for
      |    |_____ i=3 -> f_atual = 1+1 = 2, f_ant_ant = 1, f_ant = 2
      |    |_____ i=4 -> f_atual = 1+2 = 3, f_ant_ant = 2, f_ant = 3
      |    |_____ i=5 -> f_atual = 2+3 = 5, f_ant_ant = 3, f_ant = 5
      |_____________________________________________________________|
```

Eficácia x Eficiência

```
int p2(int n)
{
    int i, f_ant = 1, f_ant_ant = 1, f_atual = 1;
    for (i=3; i<=n; i=i+1)
    {
        f_atual = f_ant + f_ant_ant;
        f_ant_ant = f_ant;
        f_ant = f_atual;
    }
    return f_atual;
}

p2(5)
   |_____ f_atual=?, f1=1, f2=1
   |_____ for
   |       |_____ i=3 -> f_atual = 1+1 = 2, f_ant_ant = 1, f_ant = 2
   |       |_____ i=4 -> f_atual = 1+2 = 3, f_ant_ant = 2, f_ant = 3
   |       |_____ i=5 -> f_atual = 2+3 = 5, f_ant_ant = 3, f_ant = 5
   |_____ return 5
```

Eficácia x Eficiência

Eficácia

Faz o que deveria fazer

Eficiência

Faz bem o que deveria fazer

Como calcular a eficiência do algoritmo?

Tempo real de máquina como medida?

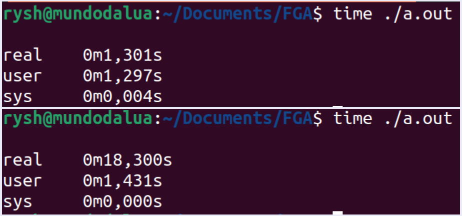


Figura: Máquina com *load* baixo e alto, respectivamente

- real: tempo total para execução (contando todos os processos em execução)
- user: tempo exclusivo do processo executado
- sys: tempo que do sistema dedicado a execução do processo
- Problema: são dependentes de fatores como a linguagem, hardware e/ou processos em execução
- Precisamos de uma medida independente da máquina

Como calcular a eficiência do algoritmo?

Tempo real de máquina como medida?

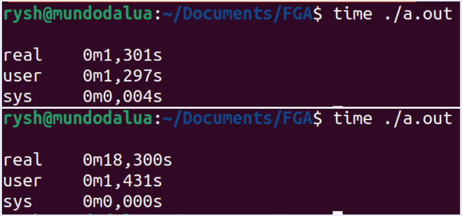


Figura: Máquina com *load* baixo e alto, respectivamente

- real: tempo total para execução (contando todos os processos em execução)
- user: tempo exclusivo do processo executado
- sys: tempo que do sistema dedicado a execução do processo

- Problema: são dependentes de fatores como a linguagem, hardware e/ou processos em execução
- Precisamos de uma medida independente da máquina

Como calcular a eficiência do algoritmo?

Tempo real de máquina como medida?

```
rysh@mundodalua:~/Documents/FGA$ time ./a.out

real    0m1,301s
user    0m1,297s
sys     0m0,004s
```

```
rysh@mundodalua:~/Documents/FGA$ time ./a.out

real    0m18,300s
user    0m1,431s
sys     0m0,000s
```

Figura: Máquina com *load* baixo e alto, respectivamente

- real: tempo total para execução (contando todos os processos em execução)
- user: tempo exclusivo do processo executado
- sys: tempo que do sistema dedicado a execução do processo

- Problema: são dependentes de fatores como a linguagem, hardware e/ou processos em execução
- Precisamos de uma medida independente da máquina

Como calcular a eficiência do algoritmo?

Contar quantas instruções são executadas?

- Analisar somente as operações **relevantes**
- Observando a **tendência do comportamento** a medida que a **entrada cresce**
- Fazendo o **cálculo aproximado** dos custos das operações

Como calcular a eficiência do algoritmo?

Contar quantas instruções são executadas?

- Analisar somente as operações **relevantes**
- Observando a **tendência do comportamento** a medida que a **entrada** cresce
- Fazendo o **cálculo aproximado** dos custos das operações

Como calcular a eficiência do algoritmo?

Contar quantas instruções são executadas?

- Analisar somente as operações **relevantes**
- Observando a **tendência do comportamento** a medida que a **entrada cresce**
- Fazendo o **cálculo aproximado** dos custos das operações

Como calcular a eficiência do algoritmo?

Contar quantas instruções são executadas?

- Analisar somente as operações **relevantes**
- Observando a **tendência do comportamento** a medida que a **entrada cresce**
- Fazendo o **cálculo aproximado** dos custos das operações
  - Definindo a complexidade dos algoritmos
  - Complexidade de um algoritmo particular
  - Complexidade de uma classe de algoritmos

Como calcular a eficiência do algoritmo?

Contar quantas instruções são executadas?

- Analisar somente as operações **relevantes**
- Observando a **tendência do comportamento** a medida que a **entrada cresce**
- Fazendo o **cálculo aproximado** dos custos das operações
  - Definindo a **complexidade dos algoritmos**
  - Complexidade de um algoritmo particular
  - Complexidade de uma classe de algoritmos

Como calcular a eficiência do algoritmo?

Contar quantas instruções são executadas?

- Analisar somente as operações **relevantes**
- Observando a **tendência do comportamento** a medida que a **entrada cresce**
- Fazendo o **cálculo aproximado** dos custos das operações
  - Definindo a **complexidade dos algoritmos**
  - **Complexidade de um algoritmo particular**
    - Busca-se o custo de um algoritmo para resolver um problema específico
    - Podemos observar quantas repetições cada trecho executa e quanta memória é gasta
  - **Complexidade de uma classe de algoritmos**

Como calcular a eficiência do algoritmo?

Contar quantas instruções são executadas?

- Analisar somente as operações **relevantes**
- Observando a **tendência do comportamento** a medida que a **entrada cresce**
- Fazendo o **cálculo aproximado** dos custos das operações
  - Definindo a **complexidade dos algoritmos**
  - **Complexidade de um algoritmo particular**
    - Busca-se o custo de um algoritmo para resolver um problema específico
    - Podemos observar quantas repetições cada trecho executa e quanta memória é gasta
  - Complexidade de uma classe de algoritmos

Como calcular a eficiência do algoritmo?

Contar quantas instruções são executadas?

- Analisar somente as operações **relevantes**
- Observando a **tendência do comportamento** a medida que a **entrada cresce**
- Fazendo o **cálculo aproximado** dos custos das operações
  - Definindo a **complexidade dos algoritmos**
  - **Complexidade de um algoritmo particular**
    - Busca-se o custo de um algoritmo para resolver um problema específico
    - Podemos observar quantas repetições cada trecho executa e quanta memória é gasta
  - Complexidade de uma classe de algoritmos

Como calcular a eficiência do algoritmo?

Contar quantas instruções são executadas?

- Analisar somente as operações **relevantes**
- Observando a **tendência do comportamento** a medida que a **entrada cresce**
- Fazendo o **cálculo aproximado** dos custos das operações
  - Definindo a **complexidade dos algoritmos**
  - **Complexidade de um algoritmo particular**
    - Busca-se o custo de um algoritmo para resolver um problema específico
    - Podemos observar quantas repetições cada trecho executa e quanta memória é gasta
  - **Complexidade de uma classe de algoritmos**
    - Busca-se o menor custo para resolver um **problema particular**
    - Analisa-se uma **família de algoritmos** que resolvem um problema específico
    - Exemplo: nos algoritmos de ordenação, qual o número mínimo possível de comparações para ordenar n números

Como calcular a eficiência do algoritmo?

Contar quantas instruções são executadas?

- Analisar somente as operações **relevantes**
- Observando a **tendência do comportamento** a medida que a **entrada cresce**
- Fazendo o **cálculo aproximado** dos custos das operações
  - Definindo a **complexidade dos algoritmos**
  - **Complexidade de um algoritmo particular**
    - Busca-se o custo de um algoritmo para resolver um problema específico
    - Podemos observar quantas repetições cada trecho executa e quanta memória é gasta
  - **Complexidade de uma classe de algoritmos**
    - Busca-se o menor custo para resolver um **problema particular**
    - Analisa-se uma **família de algoritmos** que resolvem um problema específico
    - Exemplo: nos algoritmos de ordenação, qual o número mínimo possível de comparações para ordenar n números

Como calcular a eficiência do algoritmo?

Contar quantas instruções são executadas?

- Analisar somente as operações **relevantes**
- Observando a **tendência do comportamento** a medida que a **entrada cresce**
- Fazendo o **cálculo aproximado** dos custos das operações
  - Definindo a **complexidade dos algoritmos**
  - **Complexidade de um algoritmo particular**
    - Busca-se o custo de um algoritmo para resolver um problema específico
    - Podemos observar quantas repetições cada trecho executa e quanta memória é gasta
  - **Complexidade de uma classe de algoritmos**
    - Busca-se o menor custo para resolver um **problema particular**
    - Analisa-se uma **família de algoritmos** que resolvem um problema específico

      Exemplo: nos algoritmos de ordenação, qual o número mínimo possível de comparações para ordenar n números

Como calcular a eficiência do algoritmo?

Contar quantas instruções são executadas?

- Analisar somente as operações **relevantes**
- Observando a **tendência do comportamento** a medida que a **entrada cresce**
- Fazendo o **cálculo aproximado** dos custos das operações
  - Definindo a **complexidade dos algoritmos**
  - **Complexidade de um algoritmo particular**
    - Busca-se o custo de um algoritmo para resolver um problema específico
    - Podemos observar quantas repetições cada trecho executa e quanta memória é gasta
  - **Complexidade de uma classe de algoritmos**
    - Busca-se o menor custo para resolver um **problema particular**
    - Analisa-se uma **família de algoritmos** que resolvem um problema específico
    - Exemplo: nos algoritmos de ordenação, qual o número mínimo possível de comparações para ordenar n números

Complexidade ou Função de Custo *f(n)*

Analisamos

Conta-se as operações mais relevantes considerando, também, as instâncias do problemas.

Tamanho da instância do problema *n*

- Problemas em ordenação de vetores: tamanho do vetor
- Problemas de pesquisa em memória: número de registros
- Busca em texto: número de caracteres ou padrão de busca
- etc.

Cenários (dependentes da entrada)

- **Melhor caso**: menor tempo de execução
- **Caso médio**: média dos tempos de execução
- **Pior caso**: maior tempo de execução

Complexidade ou Função de Custo *f(n)*

Analisamos

Conta-se as operações mais relevantes considerando, também, as instâncias do problemas.

Tamanho da instância do problema *n*

- Problemas em ordenação de vetores: tamanho do vetor
- Problemas de pesquisa em memória: número de registros
- Busca em texto: número de caracteres ou padrão de busca
- etc.

Cenários (dependentes da entrada)

- **Melhor caso**: menor tempo de execução
- **Caso médio**: média dos tempos de execução
- **Pior caso**: maior tempo de execução

Complexidade ou Função de Custo *f(n)*

Analisamos

Conta-se as operações mais relevantes considerando, também, as instâncias do problemas.

Tamanho da instância do problema *n*

- Problemas em ordenação de vetores: tamanho do vetor
- Problemas de pesquisa em memória: número de registros
- Busca em texto: número de caracteres ou padrão de busca
- etc.

Cenários (dependentes da entrada)

- **Melhor caso**: menor tempo de execução
- **Caso médio**: média dos tempos de execução
- **Pior caso**: maior tempo de execução

Complexidade ou Função de Custo *f(n)* - Exemplo

Busca sequencial em vetor

1. **Caso 1:**

```
int v[] = {23, 22, 98, 49, 21, 5, 3, 456, 16, 83, 50, 97};
int x = 23;
procura(x, v);
```

- Melhor caso $f(n) = 1$: procurado é o primeiro consultado

2. **Caso 2:**

```
int v[] = {23, 22, 98, 49, 21, 5, 3, 456, 16, 83, 50, 97};
int x = 97;
procura(x, v);
```

- Pior caso $f(n) = n$: procurado é o último consultado

3. **Caso 3:**

```
int v[] = {23, 22, 98, 49, 21, 5, 3,456, 16, 83, 50, 97};
int x = 49;
procura(x, v);
```

- Caso médio $f(n) = (n+1)/2$: examina cerca de metade dos registros

Complexidade ou Função de Custo *f(n)* - Exemplo

Busca sequencial em vetor

1. **Caso 1:**

```c
    int v[] = {23, 22, 98, 49, 21, 5, 3, 456, 16, 83, 50, 97};
    int x = 23;
    procura(x, v);
```

   - **Melhor caso $f(n) = 1$: procurado é o primeiro consultado**

2. **Caso 2:**

```c
    int v[] = {23, 22, 98, 49, 21, 5, 3, 456, 16, 83, 50, 97};
    int x = 97;
    procura(x, v);
```

   - **Pior caso $f(n) = n$: procurado é o último consultado**

3. **Caso 3:**

```c
    int v[] = {23, 22, 98, 49, 21, 5, 3,456, 16, 83, 50, 97};
    int x = 49;
    procura(x, v);
```

   - **Caso médio $f(n) = (n + 1)/2$: examina cerca de metade dos registros**

Complexidade ou Função de Custo *f(n)* - Exemplo

Busca sequencial em vetor

1. **Melhor caso** $f(n) = 1$: procurado é o primeiro consultado
2. **Pior caso** $f(n) = n$: procurado é o último consultado
3. **Caso médio** $f(n) = (n+1)/2$: examina aproximadamente metade

```
    int v[] = {23, 22, 98, 49, 21, 5, 3,456, 16, 83, 50, 97};
    int x = 49;
    procura(x, v);
```

- p_i a probabilidade de encontrar o elemento na posição i
- Todos tem o mesmo $p_i = 1/n$, $1 \leq i \leq n$
- f(n) = soma do número de comparações x probabilidade

$$
\begin{aligned}
f(n) &= 1(\frac{1}{n}) + 2(\frac{1}{n}) + ... + n(\frac{1}{n}) \\
&= \frac{1}{n}(1 + 2 + ... + n) \\
&= \frac{1}{n}(\frac{n(n+1)}{2}) = \frac{(n+1)}{2}
\end{aligned}
$$

Valores comuns da função de custo: $f(n) = 1$

Complexidade constante (tempo constante)

- Independem do tamanho de n
- As instruções são executadas um número fixo de vezes
  - Atribuições
  - Comparações (relacionais)
  - Operações aritmética
  - Acessos a memória
  - Comando de decisão

```

void f1(){
    int i = 19;
    if(i==0){
        i++;
    }
}

```

Valores comuns da função de custo $f(n) = n$

Complexidade linear

- Realiza-se um pequeno trabalho sobre cada elemento da entrada
- n entradas, n saídas
- Anel ou laço

```
int pesquisa(int x, int n, int v[])
{
    for (int i=0; i<n && v[i]!=x; i=i+1);
    return i;
}
```

Valores comuns da função de custo $f(n) = n$

Complexidade linear

- Realiza-se um pequeno trabalho sobre cada elemento da entrada
- n entradas, n saídas
- Anel ou laço

```
int pesquisa(int x, int n, int v[])
{
    for (int i=0; i<n && v[i]!=x; i=i+1);
    return i;
}
```

Valores comuns da função de custo $f(n) = n$

Complexidade linear

- Realiza-se um pequeno trabalho sobre cada elemento da entrada
- n entradas, n saídas
- Anel ou laço
    - (Tempos comandos internos + avaliação da condição) x número de iterações

```
int pesquisa(int x, int n, int v[])
{
    for (int i=0; i<n && v[i]!=x; i=i+1);
    return i;
}
```

Valores comuns da função de custo $f(n) = n$

Complexidade linear

- Realiza-se um pequeno trabalho sobre cada elemento da entrada
- n entradas, n saídas
- Anel ou laço
  - (Tempos comandos internos + avaliação da condição) x número de iterações

```

int pesquisa(int x, int n, int v[])
{
    for (int i=0; i<n && v[i]!=x; i=i+1);
    return i;
}

```

Valores comuns da função de custo $f(n) = n$

Complexidade linear

```c
//fatorial iterativo
//f(n) = 3*n + 1
int fat(int n) {
    int f = 1;
    while(n > 0)
        f *= n--;
    return f;
}
```

```c
//fatorial recursivo
int fat(int n){
    if(n==0) return 1;
    return n * fat(n-1);
}
```

$$
\begin{aligned}
f(n) &= f(n-1) + 3 \\
&= f(n-2) + 3 + 3 \\
&= f(n-i) + 3 * i \\
&= f(0) + 3 * n = 1 + 3 * n
\end{aligned}
$$

Valores comuns da função de custo $f(n) = n^2$

Complexidade quadrática

- Caracterizam-se pelo processamento dos dados em pares, muitas vezes com vários aninhamentos
- Se n dobra, o tempo quadruplica
- Úteis para problemas pequenos

```
//versao quadradica != ordenacao seja quadratica (algoritmo de
    quicksort)
void ordenacao_insercao(int n, int v[]){
    int i, j, x;

    for (i = 1; i < n; i++) {
        x = v[i];
        for (j = i-1; j >=0 && v[j] > x; j--)
            v[j+1] = v[j];
        v[j+1] = x;
    }
}

```

Valores comuns da função de custo $f(n) = n^2$

Complexidade quadrática

$$
\begin{aligned}
f(n) &= f(n-1) + n - 1 \\
&= f(n-2) + (n-1) - 1 + n - 1 \\
&= f(n-2) + (n-1) + n - 2 \\
&= f(n-3) + (n-2) - 1 + (n-1) + n - 2 \\
&= f(n-3) + (n-2) + (n-1) + n - 3 \\
&= f(n-i) + (n-i+1) + (n-i+2) + \ldots + (n-2) + (n-1) + n - i \\
&= \ldots \\
&= f(0) + 1 + 2 + \ldots + (n-2) + (n-1) + n - n \\
&= \frac{(1 + (n-1)) * n}{2} \\
&= \frac{1}{2} * (n^2)
\end{aligned}
$$

Valores comuns da função de custo $f(n) = n^3$

Complexidade cúbica

- Eficientes apenas para pequenos problemas.

```
//versao cubica != multiplicacao seja cubica (algoritmo de
    Strassen)
void multiplica_matrizes(int A[3][2], int B[2][3], int C[3][3])
{
    for (int l = 0; l < 3; l++) {
        for (int c = 0; c < 3; c++) {
            C[l][c] = 0;
            for (int i = 0; i < 2; i++) {
                C[l][c] += A[l][i] * B[i][c];
            }
        }
    }
}

```

Valores comuns da função de custo $f(n) = 2^n$

Complexidade exponencial

- Resultantes de problemas resolvidos por força bruta (verificar todas as possibilidades)
- Quando n é 20, o tempo é cerca de 1 milhão
- Exemplo: enumerar as linhas de uma tabela verdade
- Complexidade fatorial - $f(n) = n!$: pior que a exponencial
- Exemplo: ???

Valores comuns da função de custo $f(n) = 2^n$

Complexidade exponencial

```c
int p1(int n){
    if (n == 0) return 0;
    else if (n == 1) return 1;
    else return p1(n - 1) + p1(n - 2);
}
```

```

2^0                         _____ p1(5) _____
                           /                 \
2^1                     p1(4)                 p1(3)
                       /     \               /     \
2^2              p1(3)        p1(2)       p1(2)     p1(1)
                /    \       /    \      /    \       |
2^3         p1(2)   p1(1) p1(1) p1(0) p1(1) p1(0)     1
            /   \     |     |     |     |     |
2^4     p1(1) p1(0)   1     1     1     1     1
          |     |
          1     1

```

Valores comuns da função de custo $f(n) = 2^n$

Complexidade exponencial

$$
\begin{aligned}
f(0) &= 1 \\
f(1) &= f(2) = 1 \\
f(n) &= 1 * f(n-1) + 1 * f(n-2) = f(n-2) + f(n-3) + f(n-2) \\
&= 2 * f(n-2) + 1 * f(n-3) = 2 * f(n-3) + 2 * f(n-4) + f(n-3) \\
&= 3 * f(n-3) + 2 * f(n-4) = 3 * f(n-4) + 3 * f(n-5) + 2 * f(n-4) \\
&= 5 * f(n-4) + 3 * f(n-5) = 5 * f(n-5) + 5 * f(n-6) + 3 * f(n-5) \\
&= 8 * f(n-5) + 5 * f(n-6) \\
&= \ldots \\
&= Fib(i+1) * f(n-i) + Fib(i-1) * f(n-i-1) \\
&= Fib(n) * f(1) + Fib(n-2) * f(0)
\end{aligned}
$$

- Repete-se Fibonacci vezes: valor que cresce exponencialmente
- Fórmula fechada: $Fib(n) = \frac{1}{\sqrt{5}} * [(\frac{(1+\sqrt{5})}{2})^n - (\frac{(1-\sqrt{5})}{2})^n]$
- $f(n) \approx r^n$, sendo $r = \frac{(1+\sqrt{5})}{2}$

Valores comuns da função de custo: $f(n) = \log n$

Complexidade logarítmica

- Função logarítmica é a inversa da função exponencial
- Um pouco mais lento a medida que n cresce
- Tempo típico de algoritmos que divide o problema em problemas menores
- Não importa a base de log pois a grandeza do resultado não tem alterações significativas

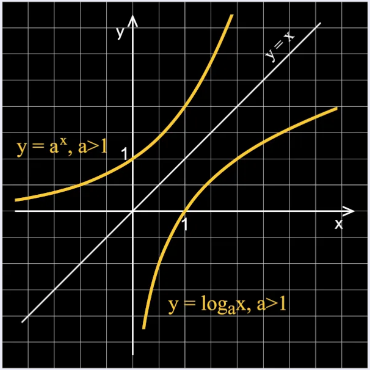
```
int        int    int     int     int
    int

    if                return
    if             return
    else  if
        return
    else
        return
```

Valores comuns da função de custo: $f(n) = \log n$

Complexidade logarítmica

- Função logarítmica é a inversa da função exponencial



- Um pouco mais lento a medida que n cresce
- Tempo típico de algoritmos que divide o problema em problemas menores
- Não importa a base de log pois a grandeza do resultado não tem alterações significativas

Valores comuns da função de custo: $f(n) = \log n$

Complexidade logarítmica

- Função logarítmica é a inversa da função exponencial
- Um pouco mais lento a medida que n cresce
- Tempo típico de algoritmos que divide o problema em problemas menores
- Não importa a base de log pois a grandeza do resultado não tem alterações significativas

```
int            int    int      int      int
    int

    if                  return
    if                return
    else  if
        return
    else
        return
```

Valores comuns da função de custo: $f(n) = \log n$

Complexidade logarítmica

- Função logarítmica é a inversa da função exponencial
- Um pouco mais lento a medida que n cresce
- Tempo típico de algoritmos que divide o problema em problemas menores
- Não importa a base de log pois a grandeza do resultado não tem alterações significativas

```
int            int    int      int      int
    int

    if                return
    if              return
    else  if
        return
    else
        return
```

Valores comuns da função de custo: $f(n) = \log n$

Complexidade logarítmica

- Função logarítmica é a inversa da função exponencial
- Um pouco mais lento a medida que n cresce
- Tempo típico de algoritmos que divide o problema em problemas menores
- Não importa a base de log pois a grandeza do resultado não tem alterações significativas
  - Sendo $n = 1000$, $\log_2 n \approx 10$ ($\log_{10} n = 3$)
  - Sendo $n = 1000000$, $\log_2 n \approx 20$ ($\log_{10} n = 6$)

```
//vetor ordenado
int pesquisa (int x, int v[], int esq, int dir){
    int meio = (esq + dir)/2;

    if (v[meio] == x) return meio;
    if (esq >= dir) return -1;
    else if (v[meio] < x)
        return pesquisa(x, v, meio+1, dir);
    else
        return pesquisa(x, v, esq, meio-1);
}
```

Valores comuns da função de custo: $f(n) = \log n$

Complexidade logarítmica

- Função logarítmica é a inversa da função exponencial
- Um pouco mais lento a medida que n cresce
- Tempo típico de algoritmos que divide o problema em problemas menores
- Não importa a base de log pois a grandeza do resultado não tem alterações significativas
  - Sendo $n = 1000$, $\log_2 n \approx 10$ ($\log_{10} n = 3$)
  - Sendo $n = 1000000$, $\log_2 n \approx 20$ ($\log_{10} n = 6$)

```
//vetor ordenado
int pesquisa (int x, int v[], int esq, int dir){
    int meio = (esq + dir)/2;

    if (v[meio] == x) return meio;
    if (esq >= dir) return -1;
    else if (v[meio] < x)
        return pesquisa(x, v, meio+1, dir);
    else
        return pesquisa(x, v, esq, meio-1);
}
```

Valores comuns da função de custo: $f(n) = \log n$

Complexidade logarítmica

- Função logarítmica é a inversa da função exponencial
- Um pouco mais lento a medida que n cresce
- Tempo típico de algoritmos que divide o problema em problemas menores
- Não importa a base de log pois a grandeza do resultado não tem alterações significativas
  - Sendo $n = 1000$, $\log_2 n \approx 10$ ($\log_{10} n = 3$)
  - Sendo $n = 1000000$, $\log_2 n \approx 20$ ($\log_{10} n = 6$)

```
//vetor ordenado
int pesquisa (int x, int v[], int esq, int dir){
    int meio = (esq + dir)/2;

    if (v[meio] == x) return meio;
    if (esq >= dir) return -1;
    else if (v[meio] < x)
        return pesquisa(x, v, meio+1, dir);
    else
        return pesquisa(x, v, esq, meio-1);
}
```

Valores comuns da função de custo: $f(n) = \log n$

Complexidade logarítmica

```
//vetor ordenado
int pesquisa (int x, int v[], int esq, int dir){
    int meio = (esq + dir)/2;

    if (v[meio] == x) return meio;
    if (esq >= dir) return -1;
    else if (v[meio] < x)
        return pesquisa(x, v, meio+1, dir);
    else
        return pesquisa(x, v, esq, meio-1);
}
```

$$
\begin{aligned}
f(n) &= f(n/2) + 1 \\
&= f(n/4) + 2 \\
&= f(n/8) + 3 \\
&= f(n/2^k) + k, 2^k = n : log_2 2^k = log_2 n : klog_2 2 = log_2 n : k = log_2 n \\
&= f(1) + log_2 n
\end{aligned}
$$

Valores comuns da função de custo $f(n) = n \log n$

Complexidade “linearítmica(?)”

- Caracterizam-se por resolver um problema quebrando em **problemas menores**, resolvendo cada um deles independentemente e **depois juntando as soluções**, localmente resolvidos, gerando um nova solução.
- Divisão e conquista

```
void intercala (int p, int q, int r, int v[]) {...}
void mergesort (int p, int r, int v[]){
    if (p < r-1) {
        int q = (p + r)/2;
        mergesort (p, q, v);
        mergesort (q, r, v);
        intercala (p, q, r, v);
    }
}
```

Valores comuns da função de custo $f(n) = n \log n$

Complexidade "linearítmica(?)"

- Caracterizam-se por resolver um problema quebrando em **problemas menores**, resolvendo cada um deles independentemente e **depois juntando as soluções**, localmente resolvidos, gerando um nova solução.
- Divisão e conquista

```
void intercala (int p, int q, int r, int v[]) {...}
void mergesort (int p, int r, int v[]){
    if (p < r-1) {
        int q = (p + r)/2;
        mergesort (p, q, v);
        mergesort (q, r, v);
        intercala (p, q, r, v);
    }
}
```

Valores comuns da função de custo $f(n) = n \log n$

Complexidade "linearítmica(?)"

- Caracterizam-se por resolver um problema quebrando em **problemas menores**, resolvendo cada um deles independentemente e **depois juntando as soluções**, localmente resolvidos, gerando um nova solução.
- Divisão e conquista

```
void intercala (int p, int q, int r, int v[]) {...}
void mergesort (int p, int r, int v[]){
    if (p < r-1) {
        int q = (p + r)/2;
        mergesort (p, q, v);
        mergesort (q, r, v);
        intercala (p, q, r, v);
    }
}
```

$$\begin{aligned} f(n) &= f(n/2) + f(n/2) + n \\ &= 2 * f(n/2) + n \\ &= 2^i * f(n/2^i) + i * n : i = log_2 n \\ &= n * f(n/n) + log_2 n * n = n * f(1) + n * log_2 n \end{aligned}$$

Como calcular formalmente?
Análise Assintótica

- É uma medição formal (matematicamente consistente) de se **calcular aproximadamente** a eficiência de algoritmos
- Descreve o **crescimento de funções**
- A ideia é achar uma **função** $g(n)$ que represente algum **limite** de $f(n)$
- E como representamos esse comportamento assintótico?

Como calcular formalmente?
Análise Assintótica

- É uma medição formal (matematicamente consistente) de se **calcular aproximadamente** a eficiência de algoritmos
- Descreve o **crescimento de funções**
- A ideia é achar uma **função** $g(n)$ que represente algum **limite** de $f(n)$
- E como representamos esse comportamento assintótico?

Como calcular formalmente?
Análise Assintótica

- É uma medição formal (matematicamente consistente) de se **calcular aproximadamente** a eficiência de algoritmos
- Descreve o **crescimento de funções**
- A ideia é achar uma **função** $g(n)$ que represente algum **limite** de $f(n)$
- E como representamos esse comportamento assintótico?

Como calcular formalmente?
Análise Assintótica

- É uma medição formal (matematicamente consistente) de se **calcular aproximadamente** a eficiência de algoritmos
- Descreve o **crescimento de funções**
- A ideia é achar uma **função** $g(n)$ que represente algum **limite** de $f(n)$
- E como representamos esse comportamento assintótico?

Notação O

- Para representar a relação assintótica surgiram diversas notações
- A mais utilizada é a **notação O**
- Para $f(n) = n^2 + 2n + 1$, sua complexidade em $O(n^2)$
- A relação assintótica entre duas funções distintas f(n) e g(n) é:
- Exemplo: busca sequencial

Notação O

- Para representar a relação assintótica surgiram diversas notações
- A mais utilizada é a **notação O**
- Para $f(n) = n^2 + 2n + 1$, sua complexidade em $O(n^2)$
- A relação assintótica entre duas funções distintas f(n) e g(n) é:
- Exemplo: busca sequencial

Notação O

- Para representar a relação assintótica surgiram diversas notações
- A mais utilizada é a **notação O**
- Para $f(n) = n^2 + 2n + 1$, sua complexidade em $O(n^2)$
- A relação assintótica entre duas funções distintas f(n) e g(n) é:
- Exemplo: busca sequencial

Notação O

- Para representar a relação assintótica surgiram diversas notações
- A mais utilizada é a **notação O**
- Para $f(n) = n^2 + 2n + 1$, sua complexidade em $O(n^2)$
- A relação assintótica entre duas funções distintas f(n) e g(n) é:
  - **Comparação da tendência de crescimento de $f(n)$ e $g(n)$**
  - Para poder concluir que $f(n) = O(g(n))$
  - Informalmente: $f(n)$ cresce, no máximo, tão rapidamente quanto $g(n)$
  - $g(n)$ é o **limite superior** para a taxa de crescimento de $f(n)$
  - Diz-se que $g(n)$ **domina assintoticamente** $f(n)$
- Exemplo: busca sequencial

Notação O

- Para representar a relação assintótica surgiram diversas notações
- A mais utilizada é a **notação O**
- Para $f(n) = n^2 + 2n + 1$, sua complexidade em $O(n^2)$
- A relação assintótica entre duas funções distintas f(n) e g(n) é:
  - **Comparação da tendência de crescimento de $f(n)$ e $g(n)$**
  - Para poder concluir que $f(n) = O(g(n))$
  - Informalmente: $f(n)$ cresce, no máximo, tão rapidamente quanto $g(n)$
  - $g(n)$ é o **limite superior** para a taxa de crescimento de $f(n)$
  - Diz-se que $g(n)$ **domina assintoticamente** $f(n)$
- Exemplo: busca sequencial

Notação O

- Para representar a relação assintótica surgiram diversas notações
- A mais utilizada é a **notação O**
- Para $f(n) = n^2 + 2n + 1$, sua complexidade em $O(n^2)$
- A relação assintótica entre duas funções distintas f(n) e g(n) é:
  - **Comparação da tendência de crescimento de $f(n)$ e $g(n)$**
  - Para poder concluir que $f(n) = O(g(n))$
  - Informalmente: $f(n)$ cresce, no máximo, tão rapidamente quanto $g(n)$
  - $g(n)$ é o **limite superior** para a taxa de crescimento de $f(n)$
  - Diz-se que $g(n)$ **domina assintoticamente** $f(n)$
- Exemplo: busca sequencial

Notação O

- Para representar a relação assintótica surgiram diversas notações
- A mais utilizada é a **notação O**
- Para $f(n) = n^2 + 2n + 1$, sua complexidade em $O(n^2)$
- A relação assintótica entre duas funções distintas f(n) e g(n) é:
  - **Comparação da tendência de crescimento de $f(n)$ e $g(n)$**
  - Para poder concluir que $f(n) = O(g(n))$
  - Informalmente: $f(n)$ cresce, no máximo, tão rapidamente quanto $g(n)$
  - $g(n)$ é o **limite superior** para a taxa de crescimento de $f(n)$
  - Diz-se que $g(n)$ **domina assintoticamente** $f(n)$
- Exemplo: busca sequencial

Notação O

- Para representar a relação assintótica surgiram diversas notações
- A mais utilizada é a **notação O**
- Para $f(n) = n^2 + 2n + 1$, sua complexidade em $O(n^2)$
- A relação assintótica entre duas funções distintas f(n) e g(n) é:
  - **Comparação da tendência de crescimento de** $f(n)$ **e** $g(n)$
  - Para poder concluir que $f(n) = O(g(n))$
  - Informalmente: $f(n)$ cresce, no máximo, tão rapidamente quanto $g(n)$
  - $g(n)$ é o **limite superior** para a taxa de crescimento de $f(n)$
  - Diz-se que $g(n)$ **domina assintoticamente** $f(n)$
- Exemplo: busca sequencial

Notação O

- Para representar a relação assintótica surgiram diversas notações
- A mais utilizada é a **notação O**
- Para $f(n) = n^2 + 2n + 1$, sua complexidade em $O(n^2)$
- A relação assintótica entre duas funções distintas f(n) e g(n) é:
  - **Comparação da tendência de crescimento de** $f(n)$ **e** $g(n)$
  - Para poder concluir que $f(n) = O(g(n))$
  - Informalmente: $f(n)$ cresce, no máximo, tão rapidamente quanto $g(n)$
  - $g(n)$ é o **limite superior** para a taxa de crescimento de $f(n)$
  - Diz-se que $g(n)$ **domina assintoticamente** $f(n)$
- Exemplo: busca sequencial

Notação O

- Para representar a relação assintótica surgiram diversas notações
- A mais utilizada é a **notação O**
- Para $f(n) = n^2 + 2n + 1$, sua complexidade em $O(n^2)$
- A relação assintótica entre duas funções distintas f(n) e g(n) é:
  - **Comparação da tendência de crescimento de $f(n)$ e $g(n)$**
  - Para poder concluir que $f(n) = O(g(n))$
  - Informalmente: $f(n)$ cresce, no máximo, tão rapidamente quanto $g(n)$
  - $g(n)$ é o **limite superior** para a taxa de crescimento de $f(n)$
  - Diz-se que $g(n)$ **domina assintoticamente** $f(n)$
- Exemplo: busca sequencial
  - $O(n)$, custo cresce, no máximo, conforme n cresce

Notação O

- Para representar a relação assintótica surgiram diversas notações
- A mais utilizada é a **notação O**
- Para $f(n) = n^2 + 2n + 1$, sua complexidade em $O(n^2)$
- A relação assintótica entre duas funções distintas f(n) e g(n) é:
  - **Comparação da tendência de crescimento de** $f(n)$ **e** $g(n)$
  - Para poder concluir que $f(n) = O(g(n))$
  - Informalmente: $f(n)$ cresce, no máximo, tão rapidamente quanto $g(n)$
  - $g(n)$ é o **limite superior** para a taxa de crescimento de $f(n)$
  - Diz-se que $g(n)$ **domina assintoticamente** $f(n)$
- Exemplo: busca sequencial
  - $O(n)$, custo cresce, no máximo, conforme n cresce

Notação O

- Supondo em um programa
  - Com tempo de execução $f(n) = 4n^2 + 4n + 1$
  - A medida que n aumenta, o termo quadrático começa a dominar
  - Para n muito grandes, diminui-se o impacto da constante que multiplica o termo quadrático
  - Assim, temos que $f(n) = O(n^2)$

Notação O

- Supondo em um programa
  - Com tempo de execução $f(n) = 4n^2 + 4n + 1$
  - A medida que n aumenta, o termo quadrático começa a dominar
  - Para n muito grandes, diminui-se o impacto da constante que multiplica o termo quadrático
  - Assim, temos que $f(n) = O(n^2)$

Notação O

- Supondo em um programa
  - Com tempo de execução $f(n) = 4n^2 + 4n + 1$
  - A medida que n aumenta, o termo quadrático começa a dominar
  - Para n muito grandes, diminui-se o impacto da constante que multiplica o termo quadrático
  - Assim, temos que $f(n) = O(n^2)$

Notação O

- Supondo em um programa
  - Com tempo de execução $f(n) = 4n^2 + 4n + 1$
  - A medida que n aumenta, o termo quadrático começa a dominar
  - Para n muito grandes, diminui-se o impacto da constante que multiplica o termo quadrático
  - Assim, temos que $f(n) = O(n^2)$

Notação O - observações

- A dominação assintótica revela a **equivalência** entre os algoritmos
  - Sendo F e G algoritmos da mesma classe
  - Com $f(n) = 3.g(n)$, mesmo F sendo 3 vezes mais lento que G
  - Possuem a mesma complexidade $O(f(n)) = O(g(n))$
  - Nestes casos, o comportamento assintótico não é indicado na comparação dos algoritmos F e G
    - Pois são avaliados pela comparação das funções (tendência)
    - Ignorando as constantes de proporcionalidade
  - **Mesma tendência de crescimento**

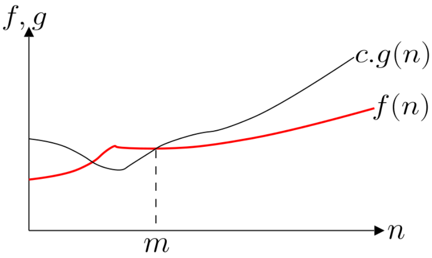
Notação O - observações

- Outro aspecto a ser considerado é o **tamanho do problema** a ser executado
  - **Uma complexidade $O(n)$ em geral representa um programa mais eficiente que um $O(n^2)$**
  - Porém dependendo do valor de n, um algoritmo $O(n^2)$ poder ser mais indicado do que o $O(n)$
  - Por exemplo, com $f(n) = 100.n$ e $g(n) = 2.n^2$
    - ★ Problemas com $n < 50$
    - ★ $g(n) = 2.n^2$ é mais eficiente do que um $f(n) = 100.n$

- Ressalta-se que **algoritmos de complexidade polinomial** e os de **complexidade exponencial** tem significativa distinção quando o tamanho de n cresce:
  - Um **problema** é considerado **bem resolvido** quando existe um **algoritmo polinomial** para resolvê-lo
  - Polinomial: $O(n^2)$, $O(n^3)$, $O(n)$, $O(\log n)$, $O(n \log n)$
  - Exponencial: $f(n) = 2^n$, $f(n) = n!$

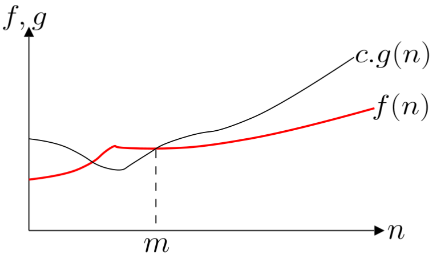
Notação O

- Formalmente, define-se:
  - Uma função $f(n) = O(g(n))$
  - Se $f(n) \leq c.g(n)$, para algum c positivo e para todo n suficiente grande, ou seja,



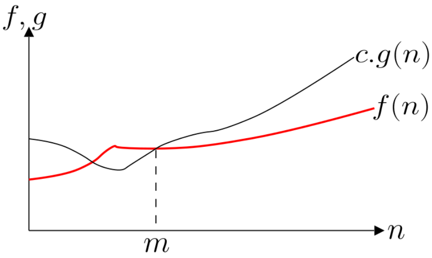
Notação O

- Formalmente, define-se:
  - Uma função $f(n) = O(g(n))$
  - Se $f(n) \leq c.g(n)$, para algum c positivo e para todo n suficiente grande, ou seja,
    - Existem duas constantes positivas c e m
    - Tais que, $f(n) \leq c.g(n)$
    - Para todo $n \geq m$ (ponto inicial do tendência para o comportamento)



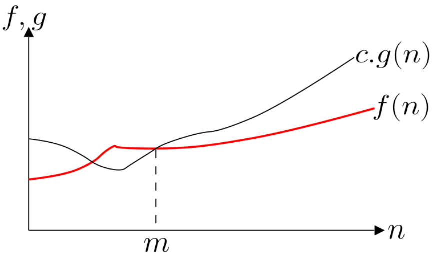
Notação O

- Formalmente, define-se:
  - Uma função $f(n) = O(g(n))$
  - Se $f(n) \leq c.g(n)$, para algum c positivo e para todo n suficiente grande, ou seja,
    - ★ Existem duas constantes positivas c e m
    - ★ Tais que, $f(n) \leq c.g(n)$
    - ★ Para todo $n \geq m$ (ponto inicial do tendência para o comportamento)



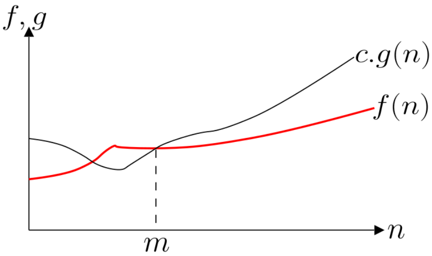
Notação O

- Formalmente, define-se:
  - Uma função $f(n) = O(g(n))$
  - Se $f(n) \leq c.g(n)$, para algum c positivo e para todo n suficiente grande, ou seja,
    - Existem duas constantes positivas c e m
    - Tais que, $f(n) \leq c.g(n)$
    - Para todo $n \geq m$ (ponto inicial do tendência para o comportamento)



Notação O

- Formalmente, define-se:
  - Uma função $f(n) = O(g(n))$
  - Se $f(n) \leq c.g(n)$, para algum c positivo e para todo n suficiente grande, ou seja,
    - ⋆ Existem duas constantes positivas c e m
    - ⋆ Tais que, $f(n) \leq c.g(n)$
    - ⋆ Para todo $n \geq m$ (ponto inicial do tendência para o comportamento)



Notação O - exemplo

- Com tempo de execução $f(n) = (n+1)^2$
  - Temos que $f(n) = O(n^2)$
  - Existem as constantes $m = 1$ e $c = 4$
  - E para todo $n \geq 1$, temos a relação $n^2 + 2n + 1 \leq 4.n^2$
- Com o tempo de execução $f(n) = 2n^2 + 4$
  - Temos que $f(n) = O(n^2)$
  - Pois $2n^2 + 4 \leq 3n^2$ para $n \geq 2 (c = 3, m = 2)$

Notação O - equivalência na tendência ao infinito

- Temos que:
  - $f(n) = O(g(n))$ se $\dfrac{\lim_{n\to\infty} f(n)}{\lim_{n\to\infty} g(n)}$ for constante
  - **Se o fator de proporcionalidade entre $f(n)$ e $g(n)$, for constante**
  - **Se são diretamente proporcionais**
- Demonstração:
  - $f(n) = a_i n^i + a_{i-1} n^{i-1} + \ldots + a_1 n^1 + a_0 n^0$
  - $f(n) = O(n^i)$

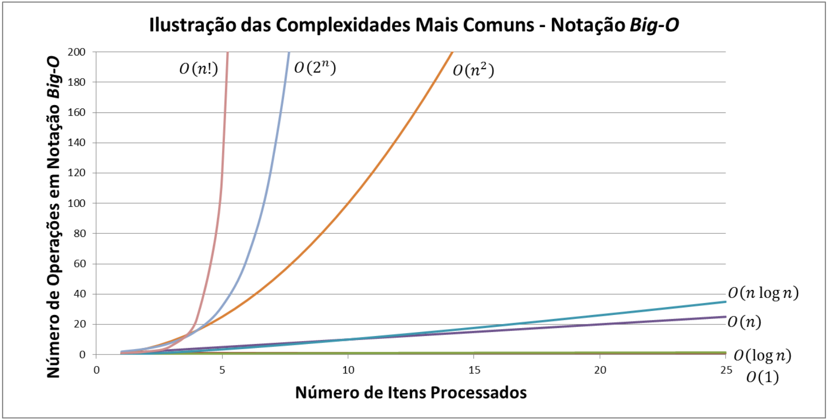
    Como:

$$\begin{aligned}
\lim_{n\to\infty} f(n) &= \lim_{n\to\infty} (a_i n^i + a_{i-1} n^{i-1} + \ldots + a_1 n^1 + a_0 n^0) \\
&= \lim_{n\to\infty} n^i (a_i + \frac{a_{i-1}}{n} + \ldots + \frac{a_1}{n^{i-1}} + \frac{a_0}{n^i}) \\
&= \lim_{n\to\infty} a_i n^i
\end{aligned}$$

    De forma análoga para $g(n) = n^i$, temos:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{a_i n^i}{n^i} = \lim_{n\to\infty} a_i = a_i$$

Notação O - equivalência na tendência ao infinito



Outras Notações

- Ω (omega)
  - Representa o **limite inferior** para função $f(n)$
  - $f(n)$ cresce, no mínimo, tão lento quanto $g(n)$
  - $f(n)$ é $\Omega(g(n))$
    - ★ Se existirem duas constantes c e m
    - ★ Tais que $|f(n)| \geq c.|g(n)|$
    - ★ Para todo $n \geq m$
- θ (theta)
  - Função $f(n)$ é **limitada superiormente e inferiormente** à $g(n)$
  - $f(n)$ cresce tão rápido quanto $g(n)$
  - Com uma diferença de apenas uma constante, ou seja
    - ★ $0 \leq c1.|g(n)| \leq f(n) \leq c2.|g(n)|$
  - Notação para representar maior precisão

- Similares as notações O e Ω
  - Notações 'o' e ω (omegazinho)
    - $f(n) = o(g(n))$: $f(n)$ cresce mais lentamente que $g(n)$
    - $f(n) = \omega(g(n))$: $f(n)$ cresce mais rapidamente que $g(n)$
  - A diferença entre o O e 'o' é que
    - O 'o' condiciona $0 \leq f(n) \leq c.g(n)$ para todo $c > 0$
    - O O condiciona $0 \leq f(n) \leq c.g(n)$ para algum $c > 0$
  - A diferença entre o Ω e ω é que
    - O ω condiciona o $0 \leq c.g(n) \leq f(n)$ para todo $c > 0$
    - O Ω condiciona o $0 \leq c.g(n) \leq f(n)$ para algum $c > 0$