

Análise de algoritmos

Prof^a. Rose Yuri Shimizu

Eficácia x Eficiência

```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2)
4          return 1;
5      else
6          return p1(n - 1) + p1(n - 2);
7  }
```

```
1  int p2(int n)
2  {
3      int i, f_ant = 1, f_ant_ant = 1, f_atual = 1;
4      for (i=3; i<=n; i=i+1)
5      {
6          f_atual = f_ant + f_ant_ant;
7          f_ant_ant = f_ant;
8          f_ant = f_atual;
9      }
10     return f_atual;
11 }
12
```

Eficácia x Eficiência

```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2) // <- p1(5) verifica
4          return 1;
5      else
6          return p1(n - 1) + p1(n - 2);
7  }
8
```

```
1      p1(5)
2      |
3      (n==1||n==2)?
4
5
6
7
8
```

```
1      stack
2
3      |
4      |
5      |
6      |
7      p1(5)
8      |
```

Eficácia x Eficiência

```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2)
4          return 1;
5      else
6          return p1(n - 1) + p1(n - 2);
7          //      _____ p1(5) chama p1(4)
8  }
```

```
1      _____ p1(5)
2      /
3  p1(4)
```

```
1  |
2  |
3  |
4  |  p1(4)
5  |  p1(5) -> p1(4)
6  |
```

Eficácia x Eficiência

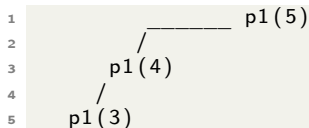
```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2) //<- p1(4) verifica
4          return 1;
5      else
6          return p1(n - 1) + p1(n - 2);
7  }
8
```

```
1          p1(5)
2      /
3  p1(4)
4  |
5 (n==1||n==2)?
6
```

```
1  |
2  |
3  |
4  |  p1(4)
5  |  p1(5) -> p1(4)
6  |
```

Eficácia x Eficiência

```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2)
4          return 1;
5      else
6          return p1(n - 1) + p1(n - 2);
7          //      _____ p1(4) chama (p1(3))
8  }
```



A call stack diagram showing the sequence of function calls. It consists of a vertical line on the left and a vertical line on the right. Between them, the following text is displayed: p1(3), p1(4) -> p1(3), and p1(5) -> p1(4). The diagram is shown on a background with line numbers 1 to 6 on the left.

Eficácia x Eficiência

```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2) //<- p1(3) verifica
4          return 1;
5      else
6          return p1(n - 1) + p1(n - 2);
7  }
8
```

```
1          p1(5)
2          /
3      p1(4)
4      /
5  p1(3)
6      |
7  (n==1||n==2)?
8
```

```
1  |
2  |
3  |  p1(3)
4  |  p1(4) -> p1(3)
5  |  p1(5) -> p1(4)
6  |
```

Eficácia x Eficiência

```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2)
4          return 1;
5      else
6          return p1(n - 1) + p1(n - 2);
7          //      ————— p1(3) chama p1(2)
8  }
```

```
1          /———— p1(5)
2         /
3        /
4       /
5      /
6     /
7    /
8   /
9  /
10 /
11 /
12 /
13 /
14 /
15 /
16 /
17 /
18 /
19 /
20 /
21 /
22 /
23 /
24 /
25 /
26 /
27 /
28 /
29 /
30 /
31 /
32 /
33 /
34 /
35 /
36 /
37 /
38 /
39 /
40 /
41 /
42 /
43 /
44 /
45 /
46 /
47 /
48 /
49 /
50 /
51 /
52 /
53 /
54 /
55 /
56 /
57 /
58 /
59 /
60 /
61 /
62 /
63 /
64 /
65 /
66 /
67 /
68 /
69 /
70 /
71 /
72 /
73 /
74 /
75 /
76 /
77 /
78 /
79 /
80 /
81 /
82 /
83 /
84 /
85 /
86 /
87 /
88 /
89 /
90 /
91 /
92 /
93 /
94 /
95 /
96 /
97 /
98 /
99 /
100 /
```

```
1  |
2  |    p1(2)
3  |    p1(3) -> p1(2)
4  |    p1(4) -> p1(3)
5  |    p1(5) -> p1(4)
6  |
```


Eficácia x Eficiência

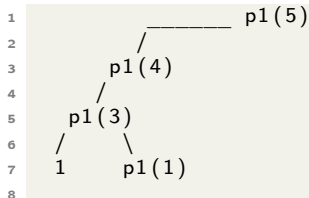
```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2) //<- p1(2) verifica
4          return 1; //<- p1(2) devolve
5      else
6          return p1(n - 1) + p1(n - 2);
7  }
8
```

```
1          p1(5)
2          /
3      p1(4)
4      /
5  p1(3)
6  /
7  p1(2)
8  |
9  (n==1||n==2)?
10
```

```
1  |
2  |  p1(2) -> 1
3  |  p1(3) -> p1(2)
4  |  p1(4) -> p1(3)
5  |  p1(5) -> p1(4)
6  |
```

Eficácia x Eficiência

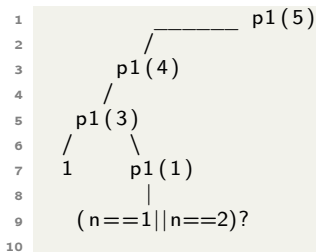
```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2)
4          return 1;
5      else
6          return p1(n - 1) + p1(n - 2);
7          //          _____ p1(3) chama p1(1)
8  }
```



```
1  |
2  |   p1(1)
3  |   p1(3) -> 1 + p1(1)
4  |   p1(4) -> p1(3)
5  |   p1(5) -> p1(4)
6  |
```

Eficácia x Eficiência

```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2) //<- p1(1) verifica
4          return 1; //<- p1(1) devolve
5      else
6          return p1(n - 1) + p1(n - 2);
7  }
8
```



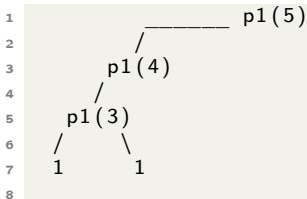
```
1  |
2  |   p1(1) ->      1
3  |   p1(3) ->    1 + p1(1)
4  |   p1(4) -> p1(3)
5  |   p1(5) -> p1(4)
6  |
```

Eficácia x Eficiência

```

1  int p1(int n)
2  {
3      if (n == 1 || n == 2)
4          return 1;
5      else
6          return p1(n - 1) + p1(n - 2); //← p1(3) devolve
7  }
8

```



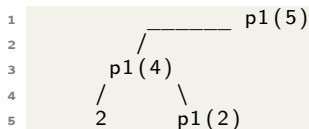
```

1 |
2 |
3 | p1(3) -> 1 + 1
4 | p1(4) -> p1(3)
5 | p1(5) -> p1(4)
6 |

```

Eficácia x Eficiência

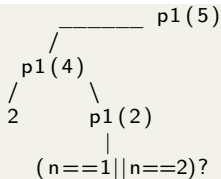
```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2)
4          return 1;
5      else
6          return p1(n - 1) + p1(n - 2);
7          //          _____ p1(4) chama p1(2)
8  }
```



```
1  |
2  |
3  |  p1(2)
4  |  p1(4) -> 2 + p1(2)
5  |  p1(5) -> p1(4)
6  |
```

Eficácia x Eficiência

```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2) //<- p1(2) verifica
4          return 1; //<- p1(2) devolve
5      else
6          return p1(n - 1) + p1(n - 2);
7  }
```



```
1 |
2 |
3 | p1(2) -> 1
4 | p1(4) -> 2 + p1(2)
5 | p1(5) -> p1(4)
6 |
```

Eficácia x Eficiência

```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2)
4          return 1;
5      else
6          return p1(n - 1) + p1(n - 2); //← p1(4) devolve
7  }
8
```

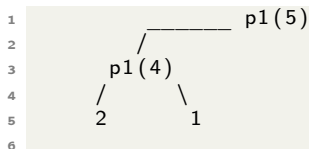


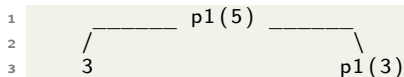
Diagram illustrating the return values for the recursive calls:

```
graph TD
    p1_4[p1(4)] --> 2
    p1_4 --> 1
    p1_5[p1(5)] --> p1_4
```

The diagram shows the return values for the recursive calls. $p1(4)$ returns 2 and 1, and $p1(5)$ returns $p1(4)$.

Eficácia x Eficiência

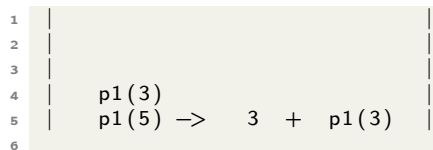
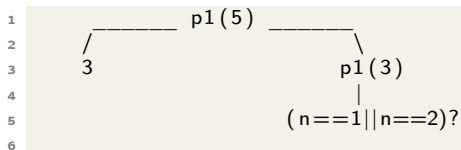
```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2)
4          return 1;
5      else
6          return p1(n - 1) + p1(n - 2);
7          //          _____ p1(5) chama p1(3)
8  }
```



A call stack diagram showing the sequence of function calls. The stack contains p1(3) at the bottom and p1(5) above it. An arrow points from p1(5) to the expression 3 + p1(3), indicating the return value calculation. The diagram is shown on a background with line numbers 1 to 6.

Eficácia x Eficiência

```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2) //<- p1(3) verifica
4          return 1;
5      else
6          return p1(n - 1) + p1(n - 2);
7  }
8
```

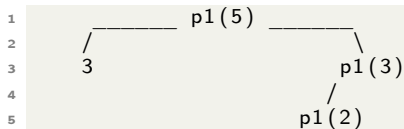


Eficácia x Eficiência

```

1  int p1(int n)
2  {
3      if (n == 1 || n == 2)
4          return 1;
5      else
6          return p1(n - 1) + p1(n - 2);
7          //          _____ p1(3) chama p1(2)
8  }

```



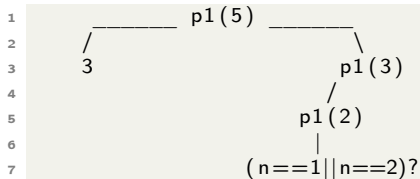
```

1
2
3      p1(2)
4      p1(3) -> p1(2)
5      p1(5) -> 3 + p1(3)

```

Eficácia x Eficiência

```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2) //<- p1(2) verifica
4          return 1; //<- p1(2) devolve
5      else
6          return p1(n - 1) + p1(n - 2);
7  }
8
```



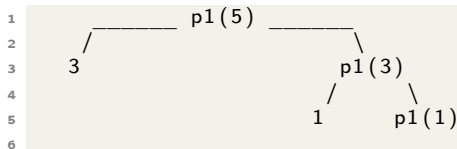
```

1  |
2  |
3  | p1(2) -> 1
4  | p1(3) -> p1(2)
5  | p1(5) -> 3 + p1(3)
6  |

```

Eficácia x Eficiência

```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2)
4          return 1;
5      else
6          return p1(n - 1) + p1(n - 2);
7          //          _____ p1(3) chama p1(1)
8  }
```



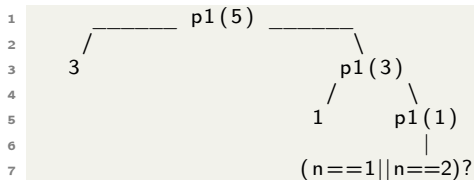
1	
2	
3	p1(1)
4	p1(3) → 1 + p1(1)
5	p1(5) → 3 + p1(3)
6	

Eficácia x Eficiência

```

1  int p1(int n)
2  {
3      if (n == 1 || n == 2) //<- p1(1) verifica
4          return 1; //<- p1(1) devolve
5      else
6          return p1(n - 1) + p1(n - 2);
7  }
8

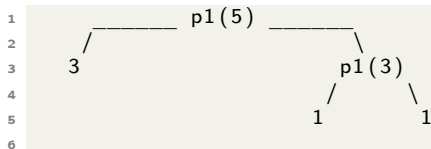
```



1				
2				
3		$p1(1) \rightarrow$		1
4		$p1(3) \rightarrow$	1 +	$p1(1)$
5		$p1(5) \rightarrow$	3 +	$p1(3)$

Eficácia x Eficiência

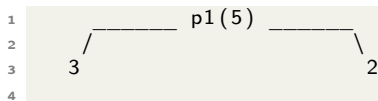
```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2)
4          return 1;
5      else
6          return p1(n - 1) + p1(n - 2); //← p1(3) devolve
7  }
8
```



```
1  |
2  |
3  |
4  |  p1(3) ->  1  +  1
5  |  p1(5) ->  3  +  p1(3)
6  |
```

Eficácia x Eficiência

```
1  int p1(int n)
2  {
3      if (n == 1 || n == 2)
4          return 1;
5      else
6          return p1(n - 1) + p1(n - 2); //<- p1(5) devolve
7  }
8
```



A diagram showing the return value of the function `p1(5)`. It consists of a vertical line on the left and a vertical line on the right. Between these lines, the text `p1(5) -> 3 + 2` is displayed. The diagram is structured as follows:

```

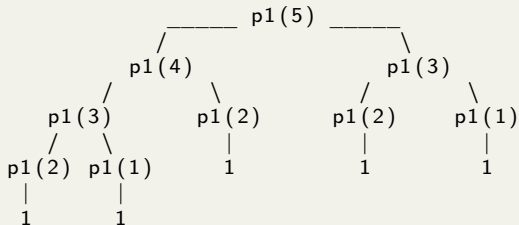
1  |
2  |
3  |
4  |
5  |  p1(5) ->  3  +  2
6  |

```

Eficácia x Eficiência

```
1  #include <stdio.h>
2
3  int p1(int n)
4  {
5      if (n == 1 || n == 2)
6          return 1;
7      else
8          return p1(n - 1) + p1(n - 2);
9  }
10
11 int main(){
12
13     printf("%d\n", p1(5)); //5
14
15     return 0;
16 }
17
```


Eficácia x Eficiência



Eficácia x Eficiência

```
1  int p2(int n) //← chamada da funcao
2  {
3      int i, f_ant = 1, f_ant_ant = 1, f_atual = 1;
4      for (i=3; i<=n; i=i+1)
5      {
6          f_atual = f_ant + f_ant_ant;
7          f_ant_ant = f_ant;
8          f_ant = f_atual; //atual é o anterior do próximo
9      }
10     return f_atual;
11 }
12
13 p2(5)
14
15
```

Eficácia x Eficiência

```
1  int p2(int n)
2  {
3      int i, f_ant = 1, f_ant_ant = 1, f_atual = 1; //<-
4      for (i=3; i<=n; i=i+1)
5      {
6          f_atual = f_ant + f_ant_ant;
7          f_ant_ant = f_ant;
8          f_ant = f_atual;
9      }
10     return f_atual;
11 }
12
13 p2(5)
14 |_____ f_atual=?, f_ant=1, f_ant_ant=1
15
16
```

Eficácia x Eficiência

```
1 int p2(int n)
2 {
3     int i, f_ant = 1, f_ant_ant = 1, f_atual = 1;
4     for (i=3; i<=n; i=i+1) //<-
5     {
6         f_atual = f_ant + f_ant_ant;
7         f_ant_ant = f_ant;
8         f_ant = f_atual;
9     }
10    return f_atual;
11 }
12
13 p2(5)
14 |_____ f_atual=?, f_ant=1, f_ant_ant=1
15 |_____ for
16
17
```

Eficácia x Eficiência

```
1 int p2(int n)
2 {
3     int i, f_ant = 1, f_ant_ant = 1, f_atual = 1;
4     for (i=3; i<=n; i=i+1)
5     {
6         f_atual = f_ant + f_ant_ant; //<-
7         f_ant_ant = f_ant;
8         f_ant = f_atual;
9     }
10    return f_atual;
11 }
12
13 p2(5)
14 |_____ f_atual=?, f_ant=1, f2=1
15 |_____ for
16 |       |_____ i=3 -> f_atual = 1+1 = 2, f_ant_ant = 1, f_ant = 2
17 |_____
18
```

Eficácia x Eficiência

```
1 int p2(int n)
2 {
3     int i, f_ant = 1, f_ant_ant = 1, f_atual = 1;
4     for (i=3; i<=n; i=i+1)
5     {
6         f_atual = f_ant + f_ant_ant;
7         f_ant_ant = f_ant;
8         f_ant = f_atual;
9     }
10    return f_atual;
11 }
```

```
12
13 p2(5)
14 |_____ f_atual=?, f_ant_ant=1, f_ant=1
15 |_____ for
16 |       |_____ i=3 -> f_atual = 1+1 = 2, f_ant_ant = 1, f_ant = 2
17 |       |_____ i=4 -> f_atual = 1+2 = 3, f_ant_ant = 2, f_ant = 3
18 |_____
19
20
```

Eficácia x Eficiência

```
1 int p2(int n)
2 {
3     int i, f_ant = 1, f_ant_ant = 1, f_atual = 1;
4     for (i=3; i<=n; i=i+1)
5     {
6         f_atual = f_ant + f_ant_ant;
7         f_ant_ant = f_ant;
8         f_ant = f_atual;
9     }
10    return f_atual;
11 }
```

```
12
13 p2(5)
14 |_____ f_atual=?, f1=1, f2=1
15 |_____ for
16 |       |_____ i=3 -> f_atual = 1+1 = 2, f_ant_ant = 1, f_ant = 2
17 |       |_____ i=4 -> f_atual = 1+2 = 3, f_ant_ant = 2, f_ant = 3
18 |       |_____ i=5 -> f_atual = 2+3 = 5, f_ant_ant = 3, f_ant = 5
19 |_____
20
21
```

Eficácia x Eficiência

```
1 int p2(int n)
2 {
3     int i, f_ant = 1, f_ant_ant = 1, f_atual = 1;
4     for (i=3; i<=n; i=i+1)
5     {
6         f_atual = f_ant + f_ant_ant;
7         f_ant_ant = f_ant;
8         f_ant = f_atual;
9     }
10    return f_atual;
11 }
```

```
12
13 p2(5)
14 |_____ f_atual=?, f1=1, f2=1
15 |_____ for
16 |_____ i=3 -> f_atual = 1+1 = 2, f_ant_ant = 1, f_ant = 2
17 |_____ i=4 -> f_atual = 1+2 = 3, f_ant_ant = 2, f_ant = 3
18 |_____ i=5 -> f_atual = 2+3 = 5, f_ant_ant = 3, f_ant = 5
19 |_____ return 5
20
21
```


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?

```
rysh@mundodaluá:~/Documents/FGA$ time ./a.out
real    0m1,301s
user    0m1,297s
sys     0m0,004s
rysh@mundodaluá:~/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?

Tempo real de máquina como medida?

```
rysh@mundodaluá:~/Documents/FGA$ time ./a.out  
  
real    0m1,301s  
user    0m1,297s  
sys     0m0,004s  
rysh@mundodaluá:~/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?

Tempo real de máquina como medida?

```
rysh@mundodaluá:~/Documents/FGA$ time ./a.out  
  
real    0m1,301s  
user    0m1,297s  
sys     0m0,004s  
-----  
rysh@mundodaluá:~/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 $f(n)$

Analizamos

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 $f(n)$

Analizamos

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 $f(n)$

Analizamos

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:

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

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

2 Caso 2:

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

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

3 Caso 3:

```
1  int v[] = {23, 22, 98, 49, 21, 5, 3, 456, 16, 83, 50, 97};
2  int x = 49;
3  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:

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

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

2 Caso 2:

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

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

3 Caso 3:

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

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

```
1   int v[] = {23, 22, 98, 49, 21, 5, 3, 456, 16, 83, 50, 97};  
2   int x = 49;  
3   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 \times probabilidade

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

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

```
1
2  void f1 () {
3      int i = 19;
4      if ( i==0){
5          i++;
6      }
7  }
```

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

```
1
2  int pesquisa(int x, int n, int v[])
3  {
4      for (int i=0; i<n && v[i]!=x; i=i+1);
5      return i;
6  }
```


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

```
1  
2 int pesquisa(int x, int n, int v[])  
3 {  
4     for (int i=0; i<n && v[i]!=x; i=i+1);  
5     return i;  
6 }  
7
```

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

```
1
2  int pesquisa(int x, int n, int v[])
3  {
4      for (int i=0; i<n && v[i]!=x; i=i+1);
5      return i;
6  }
7
```

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) \times número de iterações

```
1
2  int pesquisa(int x, int n, int v[])
3  {
4      for (int i=0; i<n && v[i]!=x; i=i+1);
5      return i;
6  }
7
```

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

Complexidade linear

```
1 //fatorial iterativo
2 //f(n) = 3*n + 1
3 int fat(int n) {
4     int f = 1;
5     while(n > 0)
6         f *= n--;
7     return f;
8 }
```

```
1 //fatorial recursivo
2 int fat(int n){
3     if(n==0) return 1;
4     return n * fat(n-1);
5 }
```

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

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

```
1 //versao quadratica != ordenacao seja quadratica (algoritmo de
  quicksort)
2 void ordenacao_insercao(int n, int v[]){
3     int i, j, x;
4
5     for (i = 1; i < n; i++) {
6         x = v[i];
7         for (j = i-1; j >=0 && v[j] > x; j--)
8             v[j+1] = v[j];
9         v[j+1] = x;
10    }
11 }
12
```

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

Complexidade quadrática

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

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

Complexidade cúbica

- Eficientes apenas para pequenos problemas.

```
1 //versao cubica != multiplicacao seja cubica (algoritmo de
   Strassen)
2 void multiplica_matrizes(int A[3][2], int B[2][3], int C[3][3])
3 {
4     for (int l = 0; l < 3; l++) {
5         for (int c = 0; c < 3; c++) {
6             C[l][c] = 0;
7             for (int i = 0; i < 2; i++) {
8                 C[l][c] += A[l][i] * B[i][c];
9             }
10        }
11    }
12 }
13
```

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

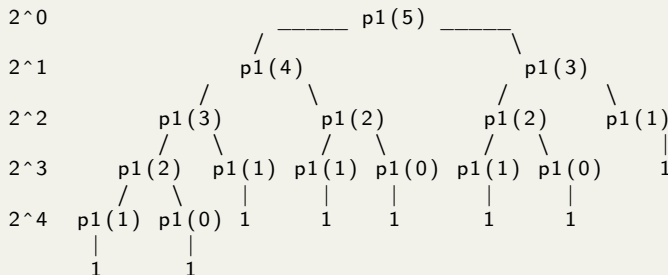
Complexidade exponencial

- 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

```
1  int p1(int n){  
2      if (n == 0) return 0;  
3      else if (n == 1) return 1;  
4      else return p1(n - 1) + p1(n - 2);  
5  }
```



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

Complexidade exponencial

$$f(0) = 1$$

$$f(1) = f(2) = 1$$

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

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

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

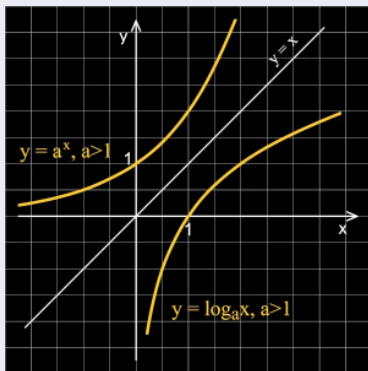
Complexidade logarítmica

- 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

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

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

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 $n = 1000000$, $\log_2 n \approx 20$ ($\log_{10} n = 6$)

```
1 //vetor ordenado
2 int pesquisa (int x, int v[], int esq, int dir){
3     int meio = (esq + dir)/2;
4
5     if (v[meio] == x) return meio;
6     if (esq >= dir) return -1;
7     else if (v[meio] < x)
8         return pesquisa(x, v, meio+1, dir);
9     else
10        return pesquisa(x, v, esq, meio-1);
11 }
```

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

Complexidade logarítmica

```
1 //vetor ordenado
2 int pesquisa (int x, int v[], int esq, int dir){
3     int meio = (esq + dir)/2;
4
5     if (v[meio] == x) return meio;
6     if (esq >= dir) return -1;
7     else if (v[meio] < x)
8         return pesquisa(x, v, meio+1, dir);
9     else
10        return pesquisa(x, v, esq, meio-1);
11 }
```

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

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

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

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

```
1 void intercala (int p, int q, int r, int v[]) { ... }
2 void mergesort (int p, int r, int v[]) {
3     if (p < r-1) {
4         int q = (p + r)/2;
5         mergesort (p, q, v);
6         mergesort (q, r, v);
7         intercala (p, q, r, v);
8     }
9 }
```

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

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

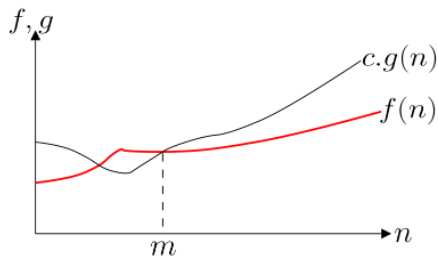
- Outro aspecto a ser considerado é o **tamanho do problema** a ser executado
 - ▶ Uma complexidade $O(n)$ em geral representa um programa mais eficiente que um $O(n^2)$
 - ▶ Porém dependendo do valor de n , um algoritmo $O(n^2)$ poder ser mais indicado do que o $O(n)$
 - ▶ Por exemplo, com $f(n) = 100.n$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

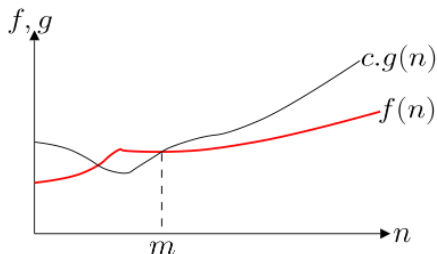
- ▶ Se $f(n) \leq c.g(n)$, para algum c positivo e para todo n suficiente grande, ou seja,



Notação O

- Formalmente, define-se:

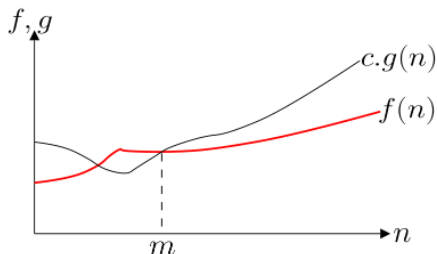
- ▶ Uma função $f(n) = O(g(n))$
- ▶ Se $f(n) \leq c.g(n)$, para algum c positivo e para todo n suficiente grande, ou seja,
 - * Existem duas constantes positivas c e m
 - * Tais que, $f(n) \leq c.g(n)$
 - * Para todo $n \geq m$ (ponto inicial do tendência para o comportamento)



Notação O

- Formalmente, define-se:

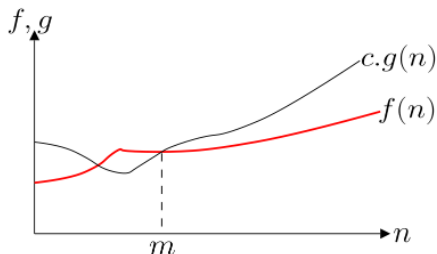
- ▶ Uma função $f(n) = O(g(n))$
- ▶ Se $f(n) \leq c \cdot g(n)$, para algum c positivo e para todo n suficiente grande, ou seja,
 - ★ Existem duas constantes positivas c e m
 - ★ Tais que, $f(n) \leq c \cdot g(n)$
 - ★ Para todo $n \geq m$ (ponto inicial do tendência para o comportamento)



Notação O

- Formalmente, define-se:

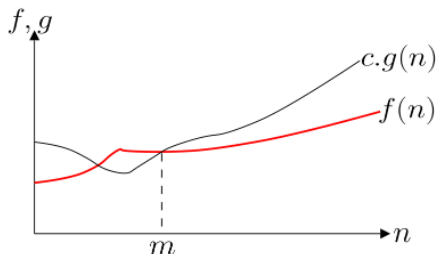
- ▶ Uma função $f(n) = O(g(n))$
- ▶ Se $f(n) \leq c.g(n)$, para algum c positivo e para todo n suficiente grande, ou seja,
 - ★ Existem duas constantes positivas c e m
 - ★ Tais que, $f(n) \leq c.g(n)$
 - ★ Para todo $n \geq m$ (ponto inicial do tendência para o comportamento)



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 $n^2 + 2n + 1 \leq 4 \cdot n^2$
- 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 $\frac{\lim_{n \rightarrow \infty} f(n)}{\lim_{n \rightarrow \infty} g(n)}$ for constante
- ▶ Se o fator de proporcionalidade entre $f(n)$ e $g(n)$, for constante
- ▶ Se são diretamente proporcionais

- Demonstração:

- ▶ $f(n) = a_i n^i + a_{i-1} n^{i-1} + \dots + a_1 n^1 + a_0 n^0$
- ▶ $f(n) = O(n^i)$

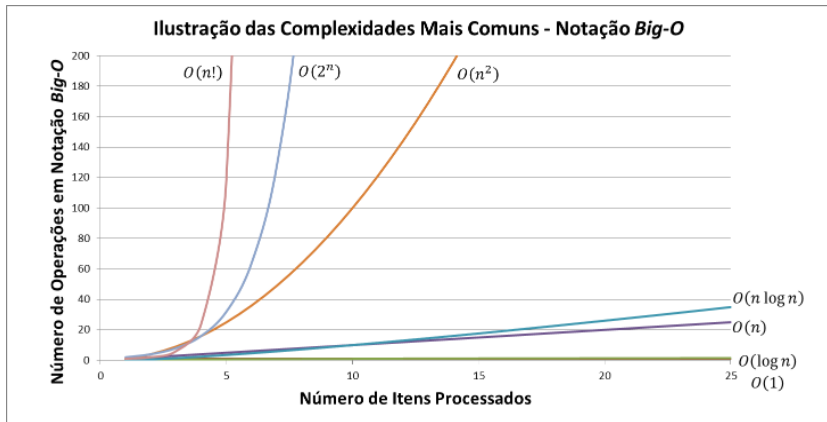
Como:

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

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

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

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 $|f(n)| \geq c \cdot |g(n)|$
 - ★ 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
 - ★ $0 \leq c1 \cdot |g(n)| \leq f(n) \leq c2 \cdot |g(n)|$
- ▶ 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 $c > 0$