



BC1424

Algoritmos e Estruturas de Dados I

Aula 03

Custos de um algoritmo e funções de complexidade

Prof. Jesús P. Mena-Chalco
jesus.mena@ufabc.edu.br

1Q-2015



Custo de um algoritmo e funções de complexidade

Estrutura de dados

- Estrutura de dados e algoritmos estão intimamente ligados:
 - Não se pode estudar ED **sem considerar os algoritmos** associados a elas;
 - Assim como **a escolha dos algoritmos** (em geral) depende da representação e da ED.

Medida do tempo de execução de um programa

- Algoritmos são encontrados em todas as áreas de Computação.
- O projeto de algoritmos é influenciado pelo estudo de seus comportamentos.
- Os algoritmos podem ser **estudados** considerando, entre outros, dois aspectos:
 - Tempo de execução.
 - Espaço ocupado (quantidade de memória).

(1) Análise de um algoritmo particular

- **Qual é o custo de usar um dado algoritmo para resolver um problema específico?**
- Características que devem ser investigadas:
 - Tempo de execução.
 - Quantidade de memória.

(2) Análise de uma **classe** de algoritmos

- Qual é o algoritmo de menos custo possível para resolver um problema particular?
- **Toda uma família de algoritmos é investigada.**
- Procura-se identificar um que seja o **melhor possível.**
- Colocam-se **limites** para a complexidade computacional dos algoritmos pertencentes à classe.

Custo de um algoritmo

- Se conseguirmos **determinar o menor custo possível** para resolver problemas de uma dada classe, então teremos a **medida da dificuldade inerente para resolver o problema.**

Custo de um algoritmo

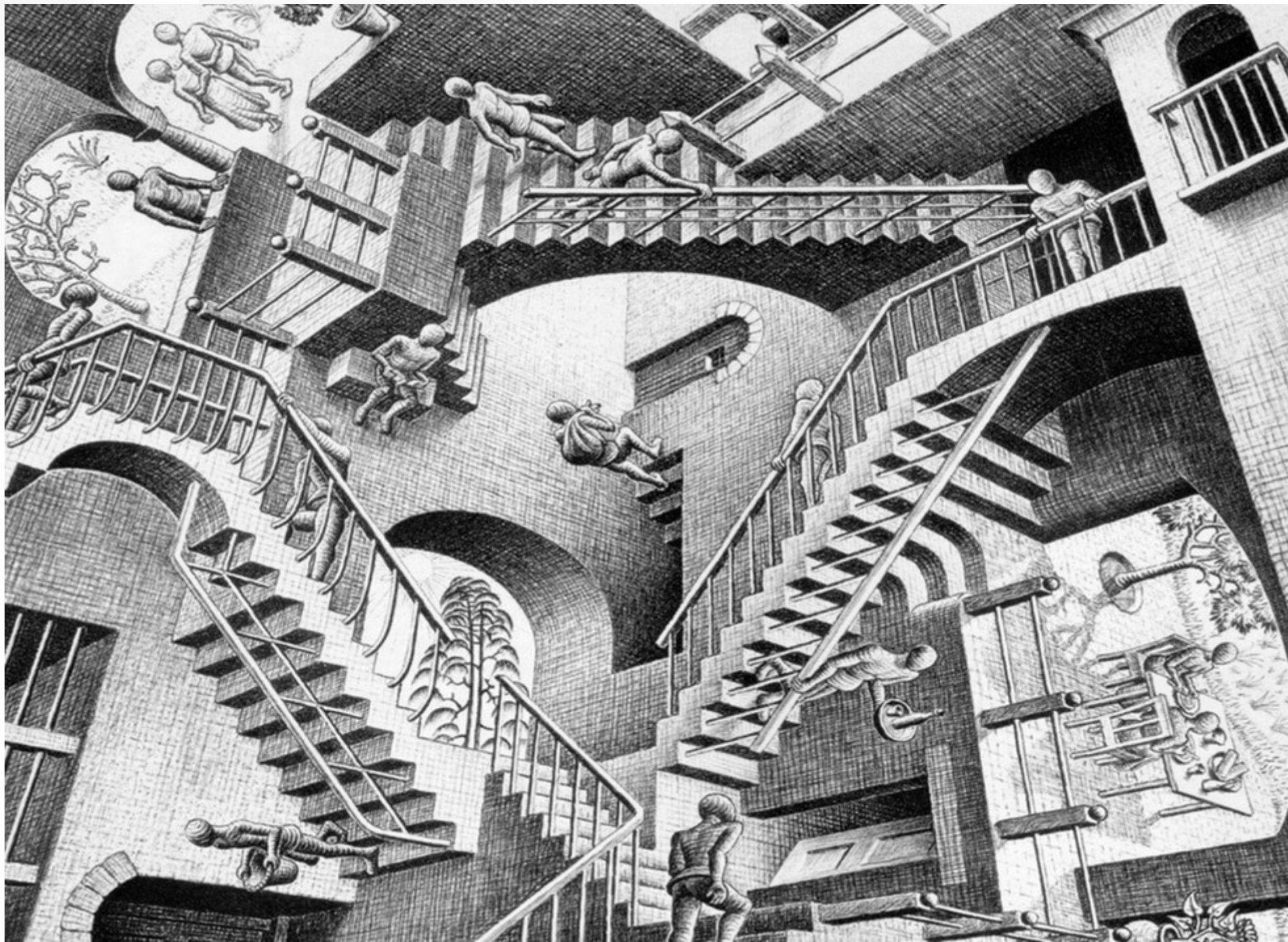
- Se conseguirmos **determinar o menor custo possível** para resolver problemas de uma dada classe, então teremos a **medida da dificuldade inerente para resolver o problema.**
- Quando um algoritmo é igual ao menor custo possível, o **algoritmo é ótimo** para a medida de custo considerada.

Custo de um algoritmo

- Se conseguirmos **determinar o menor custo possível** para resolver problemas de uma dada classe, então teremos a **medida da dificuldade inerente para resolver o problema.**
- Quando um algoritmo é igual ao menor custo possível, o **algoritmo é ótimo** para a medida de custo considerada.
- Podem existir **vários algoritmos** para resolver um mesmo problema.

Custo de um algoritmo

- Se conseguirmos **determinar o menor custo possível** para resolver problemas de uma dada classe, então teremos a **medida da dificuldade inerente para resolver o problema.**
- Quando um algoritmo é igual ao menor custo possível, o **algoritmo é ótimo** para a medida de custo considerada.
- Podem existir **vários algoritmos** para resolver um mesmo problema.
 - Se a mesma medida de custo é aplicada a diferentes algoritmos então é possível **compará-los** e escolher o mais adequado.





Medida de custo pela execução de um programa em uma plataforma real

(1) Medida de custo pela execução de um programa em uma plataforma real

- Tais medidas são bastante inadequadas e os resultados jamais devem ser generalizados:
 - Os resultados são **dependentes do compilador** que pode favorecer algumas construções em detrimento de outras;
 - Os resultados **dependem de hardware**;
 - Quanto grandes quantidades de memória são utilizadas, as medidas de tempo podem depender deste aspecto.

(1) Medida de custo pela execução de um programa em uma plataforma real

- Tais medidas são bastante inadequadas e os resultados jamais devem ser generalizados:
 - Os resultados são **dependentes do compilador** que pode favorecer algumas construções em detrimento de outras;
 - Os resultados **dependem de hardware**;
 - Quanto grandes quantidades de memória são utilizadas, as medidas de tempo podem depender deste aspecto.
- Apesar disso, há argumentos a favor de se obterem medidas reais de tempo:
 - Exemplo: Quando há vários algoritmos distintos para resolver o problema;
 - Assim, são considerados tanto os custos reais das operações como os custos não aparentes, tais como alocação de memória, indexação, carga, dentre outros.



Medida de custo por meio de um modelo matemático

(2) Medida de custo por meio de um modelo matemático

```
int F1(int a, int b) {  
    int i, t1, t2;  
  
    t1 = a;  
    t2 = b;  
  
    a = t2;  
    b = t1;  
  
    for (i=a; i<b; i++)  
        // ...  
}
```

```
int F2(int a, int b) {  
    int i, t;  
  
    t = a;  
  
    a = b;  
    b = t;  
  
    for (i=a; i<b; i++)  
        // ...  
}
```


(2) Medida de custo por meio de um modelo matemático

- Usa um **modelo matemático** baseado em um **computador idealizado**.
- Deve ser especificado o **conjunto de operações e seus custos de execuções**.
- É mais usual ignorar o custo de algumas das operações e **considerar apenas as mais significantes**.
 - Em algoritmos de ordenação:
Consideramos o **conjunto de comparações** entre os elementos do conjunto a ser ordenado e ignoramos as operações aritméticas, de atribuição e manipulação de índices, caso existam.

Função de complexidade

Função de complexidade

- Para medir o custo de execução de um algoritmo, é comum **definir uma função de custo ou função de complexidade f .**

Função de complexidade

- Para medir o custo de execução de um algoritmo, é comum **definir uma função de custo ou função de complexidade f** .
- **Função de complexidade de tempo:**
 $f(n)$ mede o tempo necessário para executar um algoritmo para um problema de tamanho n .
- **Função de complexidade de espaço:**
 $f(n)$ mede a memória necessária para executar um algoritmo para um problema de tamanho n .

Função de complexidade

- Para medir o custo de execução de um algoritmo, é comum **definir uma função de custo ou função de complexidade f** .
- **Função de complexidade de tempo:**
 $f(n)$ mede o tempo necessário para executar um algoritmo para um problema de tamanho n .
- **Função de complexidade de espaço:**
 $f(n)$ mede a memória necessária para executar um algoritmo para um problema de tamanho n .

Utilizaremos f para denotar uma função de complexidade de tempo daqui para frente.
Na realidade, f não representa tempo diretamente, mas **o número de vezes que determinada operação (considerada relevante) é realizada**.

Exemplo: Maior elemento

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros $A[0\dots n-1]$, para $n \geq 1$

```
int maiorElemento(int A[], int n) {  
    int i, max;  
    max = A[0];  
  
    for (i=1; i<n, i++) {  
        if (max < A[i])  
            max = A[i];  
    }  
  
    return max;  
}
```

Exemplo: Maior elemento

```
1  #include "stdio.h"
2
3  int main()
4  {
5      int A[10] = {6,7,8,9,0,1,2,3,4,5};
6      int max = A[0];
7
8      for(int i=1; i<10; i++)
9      {
10         if (max < A[i])
11             max = A[i];
12     }
13
14     printf("valor maximo: %d", max);
15 }
```

Exemplo: Maior elemento

```
1  #include "stdio.h"
2
3  int main()
4  {
5      int A[10] = {6,7,8,9,0,1,2,3,4,5};
6      int max = A[0];
7
8      for(int i=1; i<10; i++)
9      {
10         if (max < A[i])
11             max = A[i];
12     }
13
14     printf("valor maximo: %d", max);
15 }
```

- Seja f uma função de complexidade tal que $f(n)$ é o número de comparações entre os elementos de A .

Logo: $f(n) = n - 1$ para $n \geq 1$

Exemplo: Maior elemento

```
1  #include "stdio.h"
2
3  int main()
4  {
5      int A[10];
6      » A[0] = 6;
7      » A[1] = 7;
8      » A[2] = 8;
9      » A[3] = 9;
10     » A[4] = 0;
11     » A[5] = 1;
12     » A[6] = 2;
13     » A[7] = 3;
14     » A[8] = 4;
15     » A[9] = 5;
16
17     int max = A[0];
18
19     for(int i=1; i<10; i++)
20     {
21         » if (max < A[i])
22         »     max = A[i];
23     }
24
25     printf("valor maximo: %d", max);
26 }
```

$$f(n) = n - 1 \text{ para } n \geq 1$$

Tamanho da entrada de dados

- A medida do custo de execução de um algoritmo **depende principalmente do tamanho de entrada dos dados.**
- É comum considerar o tempo de execução de um programa como uma função do tamanho de entrada.

Tamanho da entrada de dados

- A medida do custo de execução de um algoritmo **depende principalmente do tamanho de entrada dos dados.**
- É comum considerar o tempo de execução de um programa como uma função do tamanho de entrada.
- → No caso da **função para determinar o máximo**, o custo é uniforme (**$n-1$**) sobre todos os problemas de tamanho **n** .
- → Já para um algoritmos de ordenação isso não ocorre: se os dados de entrada estiverem quase ordenados, então o algoritmo pode ter que trabalhar menos.

Melhor caso, pior caso e caso médio

- **Melhor caso:**

Menor tempo de execução sobre todas as entradas de tamanho n .

- **Pior caso:**

Maior tempo de execução sobre todas as entradas de tamanho n .

- **Caso médio (caso esperado):**

Média dos tempos de execução de todas as entradas de tamanho n .

Aqui supoe-se uma distribuição de probabilidades sobre o conjunto de entradas de tamanho n .

Exemplo: Busca de um registro

- Considere o problema de acessar os **registros** de um arquivo (cada registro tem chave única).
- **O problema:**
Dada uma chave qualquer, localize o registro que contenha esta chave
→ Considere o algoritmo de busca sequencial.

Exemplo: Busca de um registro

```
int main()
{
    int A[10] = {6,7,8,9,0,1,2,3,4,5};
    int chave, n;
    n = sizeof(A)/sizeof(A[0]);

    printf("\nIdentificar posicao da chave: ");
    scanf("%d", &chave);
    printf("\nA chave esta na posicao: %d", buscaChave(chave, A, n));
}
```

Exemplo: Busca de um registro

```
int buscaChave(int chave , int A[], int n) {  
    int i;  
  
    for(i=0; i<n; i++) {  
        if (chave == A[i])  
            return i;  
    }  
    return -1;  
}
```

Exemplo: Busca de um registro

- Seja f uma função de complexidade tal que $f(n)$ é o número de registros consultados.

- Melhor caso: $f(n) = 1$

Quando o elemento procurado é o primeiro consultado

Exemplo: Busca de um registro

- Seja f uma função de complexidade tal que $f(n)$ é o número de registros consultados.

- Melhor caso: $f(n) = 1$

Quando o elemento procurado é o primeiro consultado

- Pior caso: $f(n) = n$

Quando o elemento procurado é o último consultado

Exemplo: Busca de um registro

- Seja f uma função de complexidade tal que $f(n)$ é o número de registros consultados.

- Melhor caso: $f(n) = 1$

Quando o elemento procurado é o primeiro consultado

- Pior caso: $f(n) = n$

Quando o elemento procurado é o último consultado

- Caso médio: $f(n) = \frac{n+1}{2}$

Exemplo: Busca de um registro (caso médio)

- Consideremos que toda pesquisa recupera um elemento.
- Para recuperar o i -ésimo elemento são necessárias i comparações.

Exemplo: Busca de um registro (caso médio)

- Consideremos que toda pesquisa recupera um elemento.
- Para recuperar o i -ésimo elemento são necessárias i comparações.
- Seja p_i a probabilidade de que o i -ésimo elemento seja procurado:

$$f(n) = 1 \times p_1 + 2 \times p_2 + 3 \times p_3 + \cdots + n \times p_n$$

Exemplo: Busca de um registro (caso médio)

- Consideremos que toda pesquisa recupera um elemento.
- Para recuperar o i -ésimo elemento são necessárias i comparações.
- Seja p_i a probabilidade de que o i -ésimo elemento seja procurado:

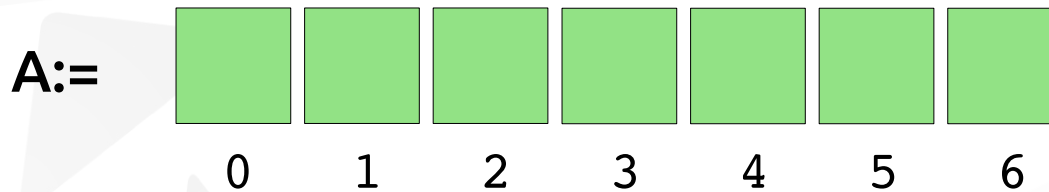
$$f(n) = 1 \times p_1 + 2 \times p_2 + 3 \times p_3 + \cdots + n \times p_n$$

- Se cada elemento tiver a mesma probabilidade de ser escolhido que todos os outros, então

$$f(n) = \frac{1}{n}(1 + 2 + 3 + \cdots + n) = \frac{n+1}{2}$$

Maior e Menor elementos

- Consideremos diferentes versões para o maior e o menor elemento de um vetor de n inteiros, para $n \geq 1$.



Maior e Menor elementos (versão 1)

```
void maxmin1(int A[], int n) {  
    int i, max, min;  
  
    max = A[0];  
    min = A[0];  
  
    for(i=1; i<n; i++) {  
        if (max < A[i])  
            max = A[i];  
        if (min > A[i])  
            min = A[i];  
    }  
    printf("\nmax: %d\nmin: %d", max, min);  
}
```

Identifique a função de complexidade $f(n)$ para o vetor A de n elementos:

- Melhor caso:
- Pior caso:
- Caso médio:

Maior e Menor elementos (versão 1)

```
void maxmin1(int A[], int n) {  
    int i, max, min;  
  
    max = A[0];  
    min = A[0];  
  
    for(i=1; i<n; i++) {  
        if (max < A[i])  
            max = A[i];  
        if (min > A[i])  
            min = A[i];  
    }  
    printf("\nmax: %d\nmin: %d", max, min);  
}
```

Identifique a função de complexidade $f(n)$ para o vetor A de n elementos:

- Melhor caso:
 - Pior caso:
 - Caso médio:
- } $f(n) = 2(n - 1)$

Maior e Menor elementos (versão 2)

```
void maxmin2(int A[], int n) {  
    int i, max, min;  
    max = A[0];  
    min = A[0];  
  
    for(i=1; i<n; i++) {  
        if (max < A[i])  
            max = A[i];  
        else  
            if (min > A[i])  
                min = A[i];  
    }  
    printf("\nmax: %d\nmin: %d", max, min);  
}
```

Identifique a função de complexidade $f(n)$ para o vetor A de n elementos:

- Melhor caso:
- Pior caso:
- Caso médio:

Maior e Menor elementos (versão 2)

```
void maxmin2(int A[], int n) {  
    int i, max, min;  
    max = A[0];  
    min = A[0];  
  
    for(i=1; i<n; i++) {  
        if (max < A[i])  
            max = A[i];  
        else  
            if (min > A[i])  
                min = A[i];  
    }  
    printf("\nmax: %d\nmin: %d", max, min);  
}
```

Identifique a função de complexidade $f(n)$ para o vetor A de n elementos:

- Melhor caso: $f(n) = (n - 1)$ *Quando os elementos estão em ordem crescente.*
- Pior caso:
- Caso médio:

Maior e Menor elementos (versão 2)

```
void maxmin2(int A[], int n) {  
    int i, max, min;  
    max = A[0];  
    min = A[0];  
  
    for(i=1; i<n; i++) {  
        if (max < A[i])  
            max = A[i];  
        else  
            if (min > A[i])  
                min = A[i];  
    }  
    printf("\nmax: %d\nmin: %d", max, min);  
}
```

Identifique a função de complexidade $f(n)$ para o vetor A de n elementos:

- Melhor caso: $f(n) = (n - 1)$ *Quando os elementos estão em ordem crescente.*
- Pior caso: $f(n) = 2(n - 1)$ *Quando os elementos estão em ordem decrescente.*
- Caso médio:

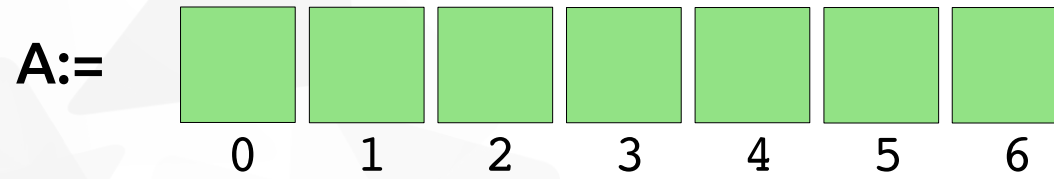
Maior e Menor elementos (versão 2)

```
void maxmin2(int A[], int n) {  
    int i, max, min;  
    max = A[0];  
    min = A[0];  
  
    for(i=1; i<n; i++) {  
        if (max < A[i])  
            max = A[i];  
        else  
            if (min > A[i])  
                min = A[i];  
    }  
    printf("\nmax: %d\nmin: %d", max, min);  
}
```

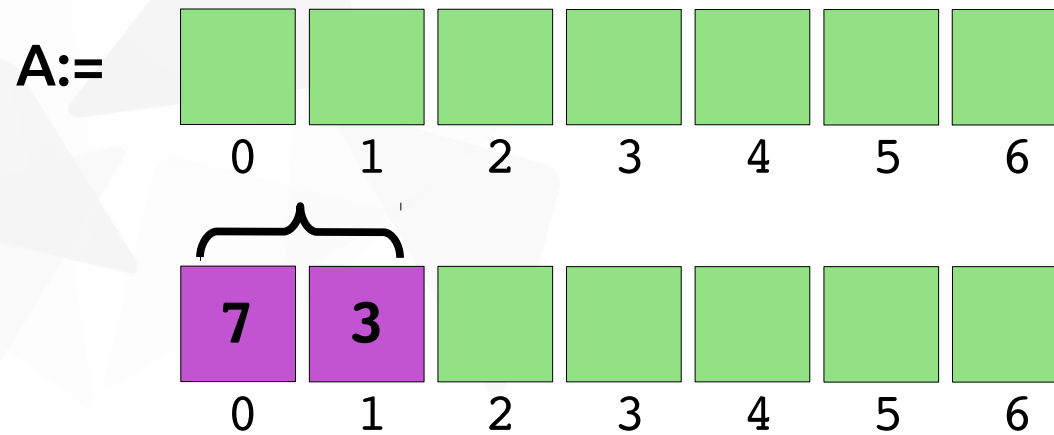
Identifique a função de complexidade $f(n)$ para o vetor A de n elementos:

- Melhor caso: $f(n) = (n - 1)$ *Quando os elementos estão em ordem crescente.*
- Pior caso: $f(n) = 2(n - 1)$ *Quando os elementos estão em ordem decrescente.*
- Caso médio: $f(n) = (n - 1) + \left(\frac{n-1}{2}\right) = \frac{3n}{2} - \frac{3}{2}$
Quando metade das vezes $\max \geq A[i]$

Maior e Menor elementos (versão 3)

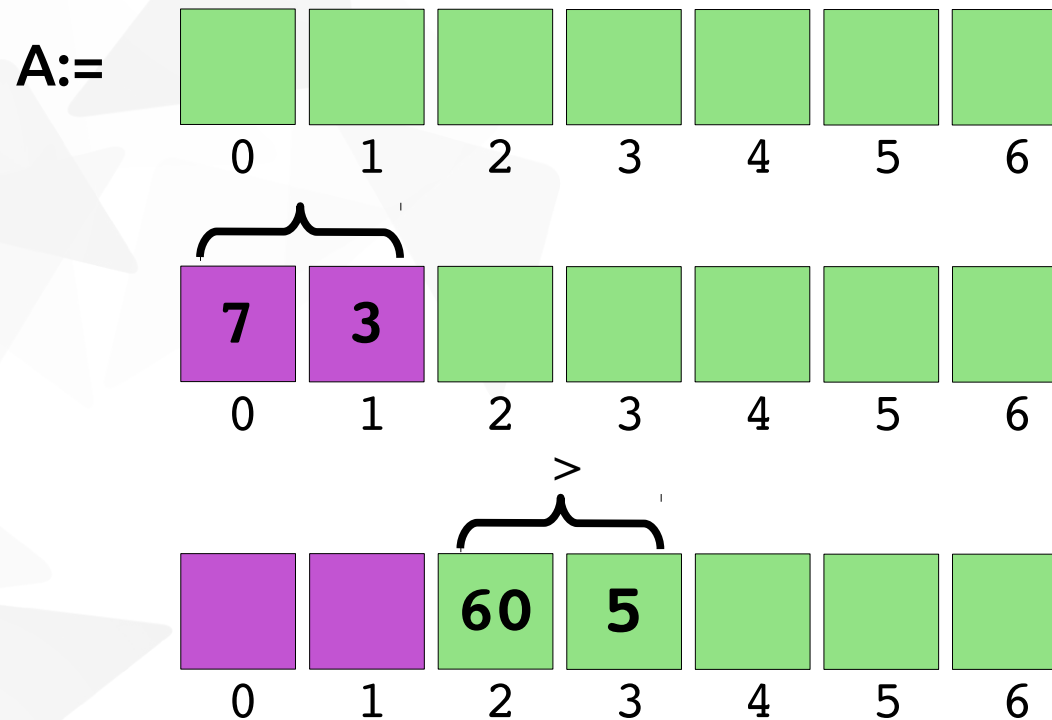


Maior e Menor elementos (versão 3)



Min = 3
Max = 7

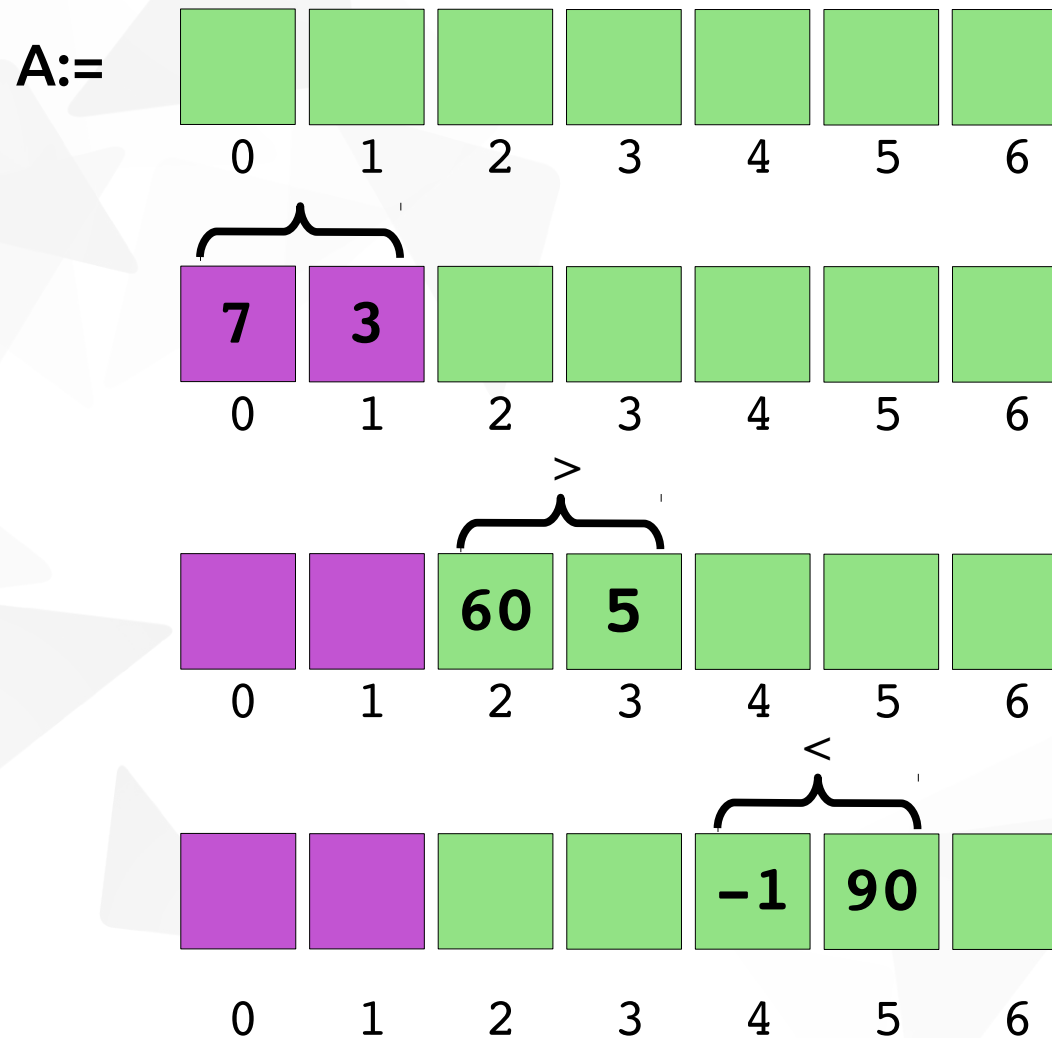
Maior e Menor elementos (versão 3)



Min = 3
Max = 7

Min = 3
Max = 60

Maior e Menor elementos (versão 3)



Min = 3
Max = 7

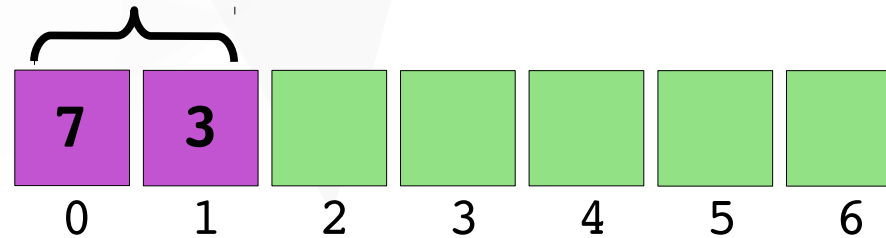
Min = 3
Max = 60

Min = -1
Max = 90

Maior e Menor elementos (versão 3)

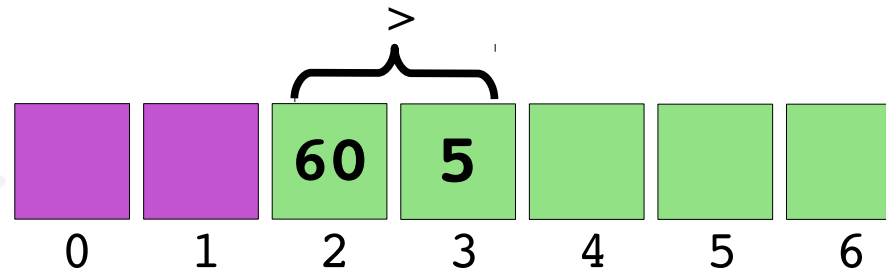
A:=

0	1	2	3	4	5	6



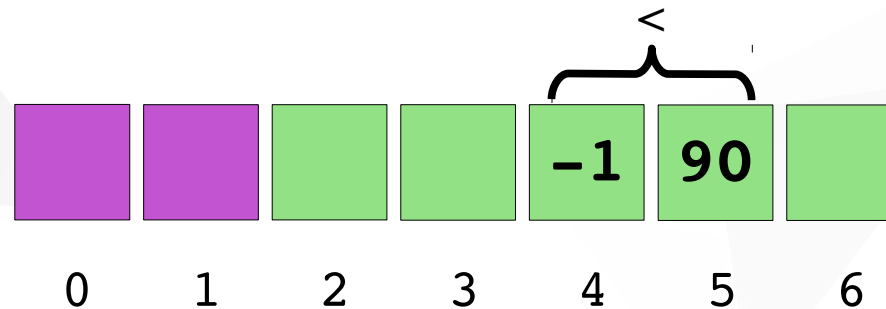
7	3					
0	1	2	3	4	5	6

Min = 3
Max = 7



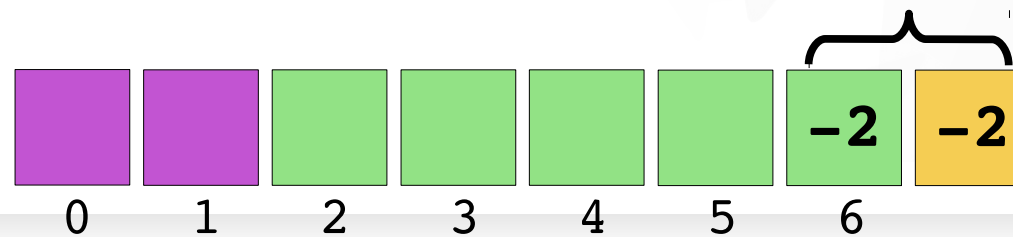
		60	5			
0	1	2	3	4	5	6

Min = 3
Max = 60



				-1	90	
0	1	2	3	4	5	6

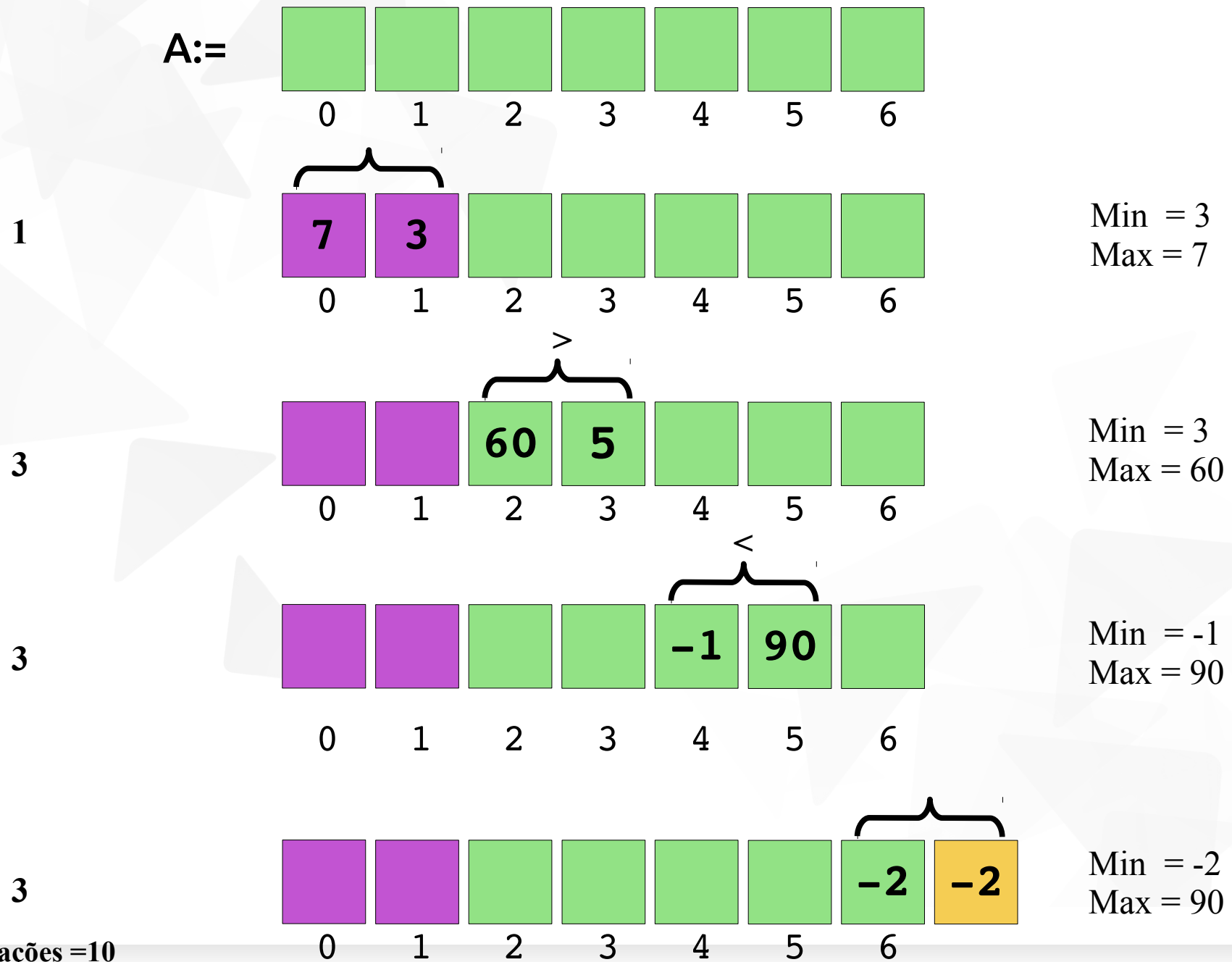
Min = -1
Max = 90



						-2	-2
0	1	2	3	4	5	6	7

Min = -2
Max = 90

Maior e Menor elementos (versão 3)



Versão 3

```
void maxmin3(int A[], int n) {
    int i, max, min;

    if (n%2==1) A[n] = A[n-1];

    if (A[0]>A[1]) {
        max = A[0];
        min = A[1];
    }
    else {
        min = A[0];
        max = A[1];
    }

    for(i=2; i<n; i+=2) {
        if (A[i]>A[i+1]) {
            if (A[i] > max) max = A[i];
            if (A[i+1] < min) min = A[i+1];
        }
        else {
            if (A[i+1] > max) max = A[i+1];
            if (A[i] < min) min = A[i];
        }
    }
    printf("\nmax: %d\nmin: %d", max, min);
}
```

Identifique a função de complexidade $f(n)$ para o vetor A de n elementos:

- Melhor caso
- Pior caso
- Caso médio

Versão 3

```
void maxmin3(int A[], int n) {  
    int i, max, min;
```

```
    if (n%2==1) A[n] = A[n-1];
```

```
    if (A[0]>A[1]) {  
        max = A[0];  
        min = A[1];
```

1 comparação

```
    }  
    else {  
        min = A[0];  
        max = A[1];  
    }
```

```
    for(i=2; i<n; i+=2) {
```

(n-2)/2 comparações

```
        if (A[i]>A[i+1]) {
```

```
            if (A[i] > max) max = A[i];  
            if (A[i+1] < min) min = A[i+1];
```

```
        }
```

```
        else {
```

```
            if (A[i+1] > max) max = A[i+1];  
            if (A[i] < min) min = A[i];
```

```
        }
```

```
    }
```

```
    printf("\nmax: %d\nmin: %d", max, min);
```

```
}
```

Identifique a função de complexidade **$f(n)$** para o vetor A de **n** elementos:

- Melhor caso
- Pior caso
- Caso médio

(n-2)/2 + (n-2)/2 comparações

```
void maxmin3(int A[], int n) {
    int i, max, min;
```

```
    if (n%2==1) A[n] = A[n-1];
```

```
    if (A[0]>A[1]) {
        max = A[0];
        min = A[1];
```

1 comparação

```
    }
    else {
        min = A[0];
        max = A[1];
    }
```

```
    for(i=2; i<n; i+=2) {
```

(n-2)/2 comparações

```
        if (A[i]>A[i+1]) {
```

```
            if (A[i] > max) max = A[i];
            if (A[i+1] < min) min = A[i+1];
```

```
        }
```

```
        else {
```

```
            if (A[i+1] > max) max = A[i+1];
            if (A[i] < min) min = A[i];
```

```
        }
```

```
    }
    printf("\nmax: %d\nmin: %d", max, min);
```

```
}
```

(n-2)/2 + (n-2)/2 comparações

Identifique a função de complexidade **$f(n)$** para o vetor A de **n** elementos:

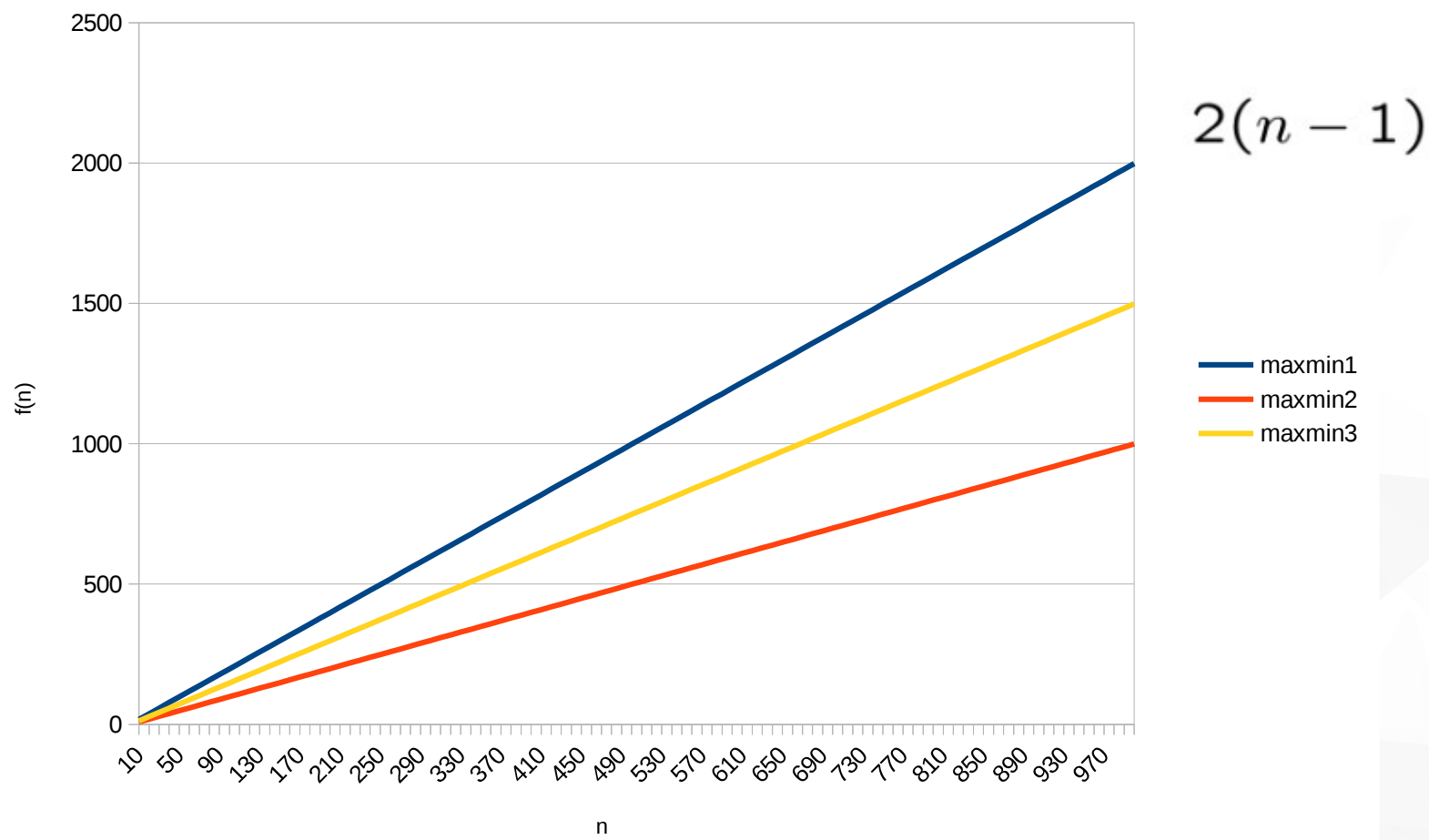
- Melhor caso
- Pior caso
- Caso médio

$$f(n) = \frac{3n}{2} - 2$$

Maior e Menor elementos

Os três algoritmos	$f(n)$		
	Melhor caso	Pior caso	Caso médio
MaxMin1	$2(n - 1)$	$2(n - 1)$	$2(n - 1)$
MaxMin2	$n - 1$	$2(n - 1)$	$3n/2 - 3/2$
MaxMin3	$3n/2 - 2$	$3n/2 - 2$	$3n/2 - 2$

Maior e Menor elementos



Melhor caso

Funções de complexidade

Os três algoritmos	$f(n)$		
	Melhor caso	Pior caso	Caso médio
MaxMin1	$2(n - 1)$	$2(n - 1)$	$2(n - 1)$
MaxMin2	$n - 1$	$2(n - 1)$	$3n/2 - 3/2$
MaxMin3	$3n/2 - 2$	$3n/2 - 2$	$3n/2 - 2$

- Não existe algoritmo que identifique o maior e o menor elemento de um vetor de n elementos com uma função menor a:

$$f(n) = \left\lceil \frac{3n}{2} \right\rceil - 2$$

Comportamento assintótico de funções

- A análise de algoritmos é realizada **para valores grandes de n** .
- Estudaremos o comportamento assintótico das **funções de custo**.
- O comportamento assintótico de **$f(n)$** representa o limite do comportamento de custo, quando **n** cresce.

Dominação assintótica

Definição:

Uma função $f(n)$ **domina assintoticamente** uma outra função $g(n)$ se existem duas constantes positivas c e n_0 tais que, para $n \geq n_0$, temos:

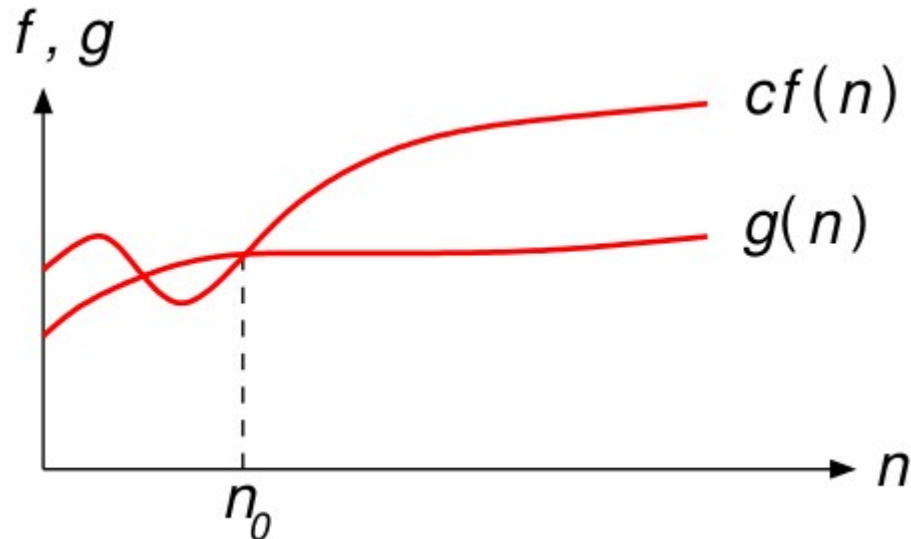
$$|g(n)| \leq c|f(n)|$$

Dominação assintótica

Definição:

Uma função $f(n)$ **domina assintoticamente** uma outra função $g(n)$ se existem duas constantes positivas c e n_0 tais que, para $n \geq n_0$, temos:

$$|g(n)| \leq c|f(n)|$$



Dominação assintótica

Exemplo:

Sejam $g(n) = (n + 1)^2$
 $f(n) = n^2$

Ambas as funções dominam assintoticamente uma da outra, já que:

$$|(n + 1)^2| \leq 4|n^2|$$

para $n \geq 1$

$$|n^2| \leq |(n + 1)^2|$$

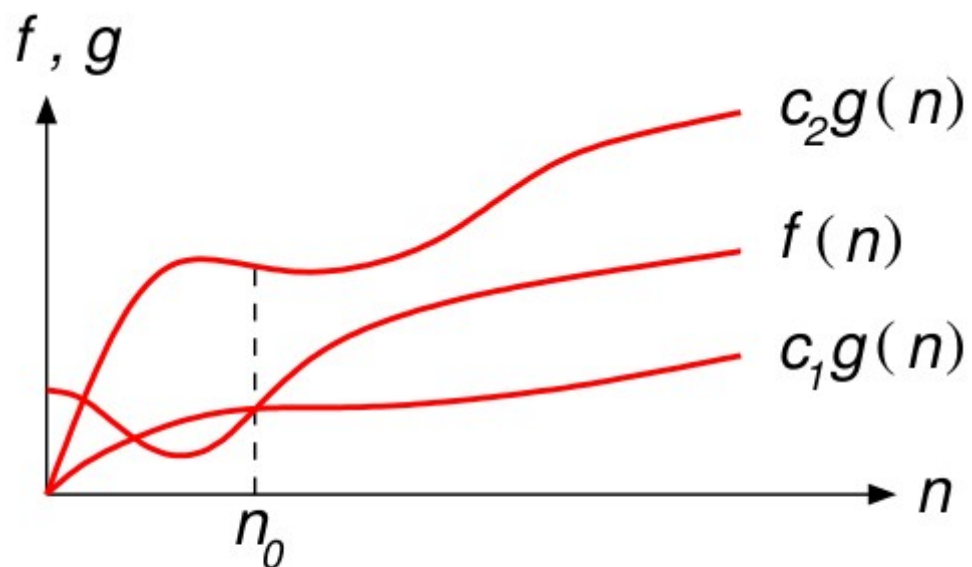
para $n \geq 0$

Notação assintótica de funções

Existem 3 notações assintóticas de funções:

- Notação Θ
- Notação O (*'O grande'*)
- Notação Ω

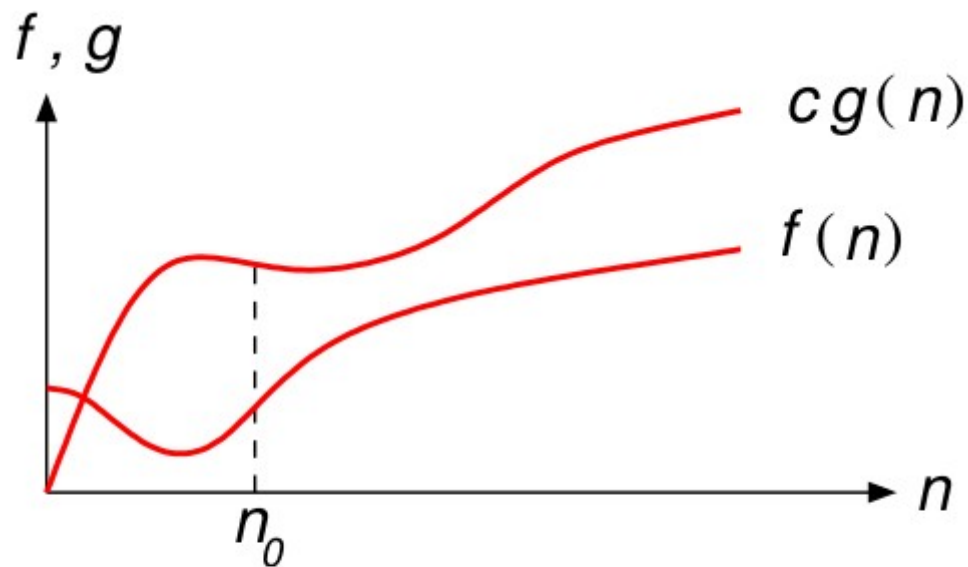
Notação Θ



$$f(n) = \Theta(g(n))$$

$g(n)$ é um limite assintótico firme de $f(n)$

Notação O ('*O grande*')



$$f(n) = O(g(n))$$

f(n) é da ordem no máximo g(n)

O é usada para expressar o tempo de execução de um algoritmo no **pior caso**, **está se definindo também o limite (superior)** do tempo de execução desse algoritmo **para todas as entradas**.

Notação O (' O grande')

$$f(n) = O(f(n))$$

$$c \times O(f(n)) = O(f(n)) \quad c = \text{constante}$$

$$O(f(n)) + O(f(n)) = O(f(n))$$

$$O(O(f(n))) = O(f(n))$$

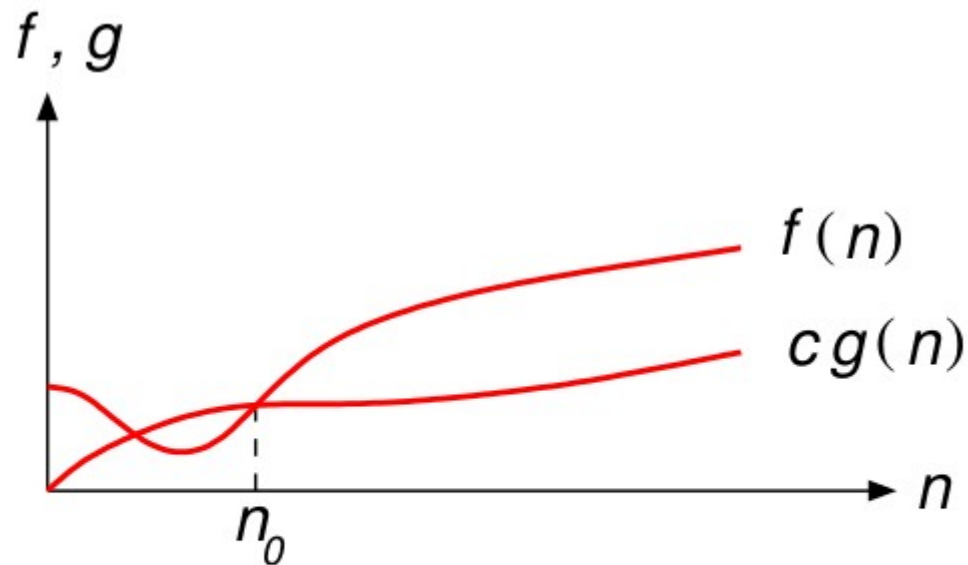
$$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$

$$O(f(n))O(g(n)) = O(f(n)g(n))$$

$$f(n)O(g(n)) = O(f(n)g(n))$$

Operações entre conjuntos de funções

Notação Ω



$$f(n) = \Omega(g(n))$$

Omega: Define um limite inferior para a função, por um fator constante.

$g(n)$ é um limite assintoticamente inferior

Teorema

Para quaisquer funções $f(n)$ e $g(n)$,

$$f(n) = \Theta(g(n))$$

se e somente se,

$$f(n) = O(g(n)), \text{ e}$$

$$f(n) = \Omega(g(n))$$

Comparação de programas

- Podemos avaliar programas **comparando as funções de complexidade, negligenciando as constantes de proporcionalidade.**

Um programa com tempo de execução $O(n)$ é melhor do que outro com tempo $O(n^2)$

Comparação de programas

Programa 1

Programa 2

$$O(n)$$

$$O(n^2)$$

Exemplo:

O programa1 leva $100n$ vezes para ser executado.

O programa2 leva $2n^2$ vezes para ser executa.

Qual dos dois é o melhor?

Depende do tamanho do problema.

Comparação de programas

Programa 1

Programa 2

$$O(n)$$

$$O(n^2)$$

Exemplo:

O programa1 leva $100n$ vezes para ser executado.

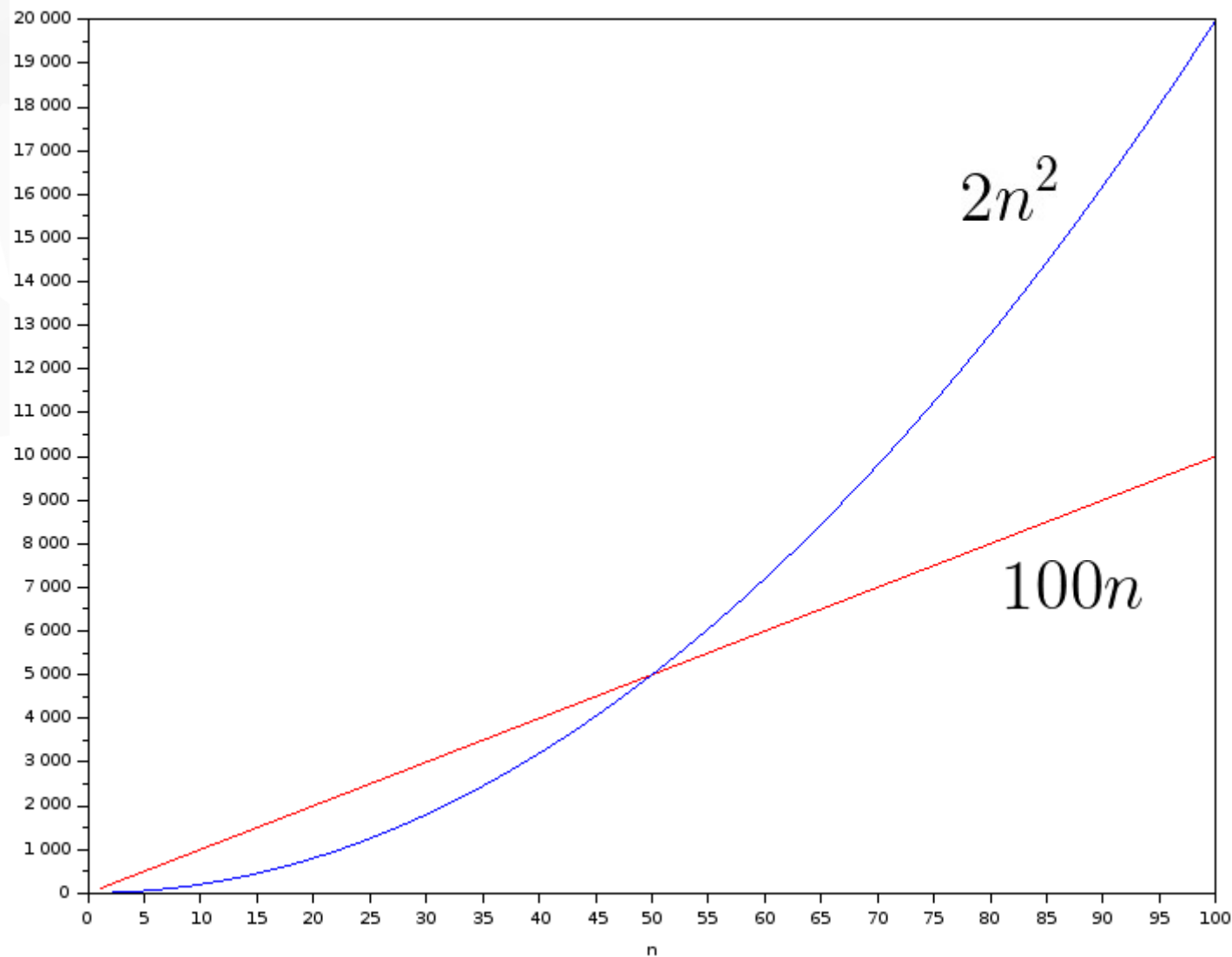
O programa2 leva $2n^2$ vezes para ser executa.

Qual dos dois é o melhor?

Depende do tamanho do problema.

- Para $n < 50$, o programa 2 é melhor
- Para $n > 50$, o programa 1 é melhor

Comparação de programas



Comparação de funções de complexidade

Função de custo	Tamanho n					
	10	20	30	40	50	60
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0.35 s	0,0036 s
n^3	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0.316 s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 s
3^n	0,059 s	58 min	6,5 anos	3855 s	10^8 s	10^{13} s

Hierarquias de funções

A seguinte herarquia de funções pode ser definida do ponto de vista assintótico:

$$1 \prec \log \log n \prec \log n \prec n^\epsilon \prec n^c \prec n^{\log n} \prec c^n \prec n^n \prec c^{c^n}$$

onde c e ϵ são constantes arbitrárias com $0 < \epsilon < 1 < c$