Aula 20
Exercícios de custos de um algoritmo

Prof. Jesús P. Mena-Chalco

3Q-2017

Estudo de algoritmos

- 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).

Atividade em aula: Ordem de crescimento

```
int G1 (int N) {
    int n, i, sum=0;
    for (n=N; n>0; n/=2)
        for (i=0; i<n; i++)
            sum++;
    return sum;
}
```

```
int G2 (int N) {
    int i, j, sum=0;
    for (i=1; i<=N; i*=2)
        for(j=0; j<i; j++)
            sum++;
    return sum;
}
```

```
int G3 (int N) {
    int i, j, sum=0;
    for (i=1; i<=N; i*=2)
        for (j=0; j<N; j++)
            sum++;
    return sum;
}
```

Atividade em aula: Ordem de crescimento

```
int G1 (int N) {
    int n, i, sum=0;
    for (n=N; n>0; n/=2)
        for (i=0; i<n; i++)
            sum++;
    return sum;
}
```

Linear

G1(N) <= 2N-1

```
int G2 (int N) {
    int i, j, sum=0;
    for (i=1; i<=N; i*=2)
        for(j=0; j<i; j++)
            sum++;
    return sum;
}
```

Linear

G2(N) <= 2N-1

```
int G3 (int N) {
    int i, j, sum=0;
    for (i=1; i<=N; i*=2)
        for (j=0; j<N; j++)
            sum++;
    return sum;
}
```

Linearithmic

G3(N) <= N(lg(N)+1)

G1

```c
#include "stdio.h"

int G1(int N) {
    int n, i, sum = 0;

    for (n=N; n>0; n/=2) {
        for (i=0; i<n; i++) {
            sum++;
        }
    }

    return sum;
}


int main(void) {
    printf("%d\n", G1(64) );
    printf("%d\n", G1(1024) );
    printf("%d\n", G1(1000000) );

    return 0;
}
```

Linear

`G1(N) <= 2N-1`

```
127
2047
1999993
```

Geometric series:

$$\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}, \quad c \neq 1, \quad \sum_{i=0}^{\infty} c^i = \frac{1}{1-c}, \quad \sum_{i=1}^{\infty} c^i = \frac{c}{1-c}, \quad c < 1,$$

G2

```c
#include "stdio.h"

int G2(int N) {
    int i, j, sum = 0;

    for (i=1; i<=N; i*=2) {
        for (j=0; j<i; j++) {
            sum++;
        }
    }

    return sum;
}


int main(void) {
    printf("%d\n", G2(64) );
    printf("%d\n", G2(1024) );
    printf("%d\n", G2(1000000) );

    return 0;
}
```

Linear

G2(N) <= 2N-1

```
127
2047
1048575
```

G3

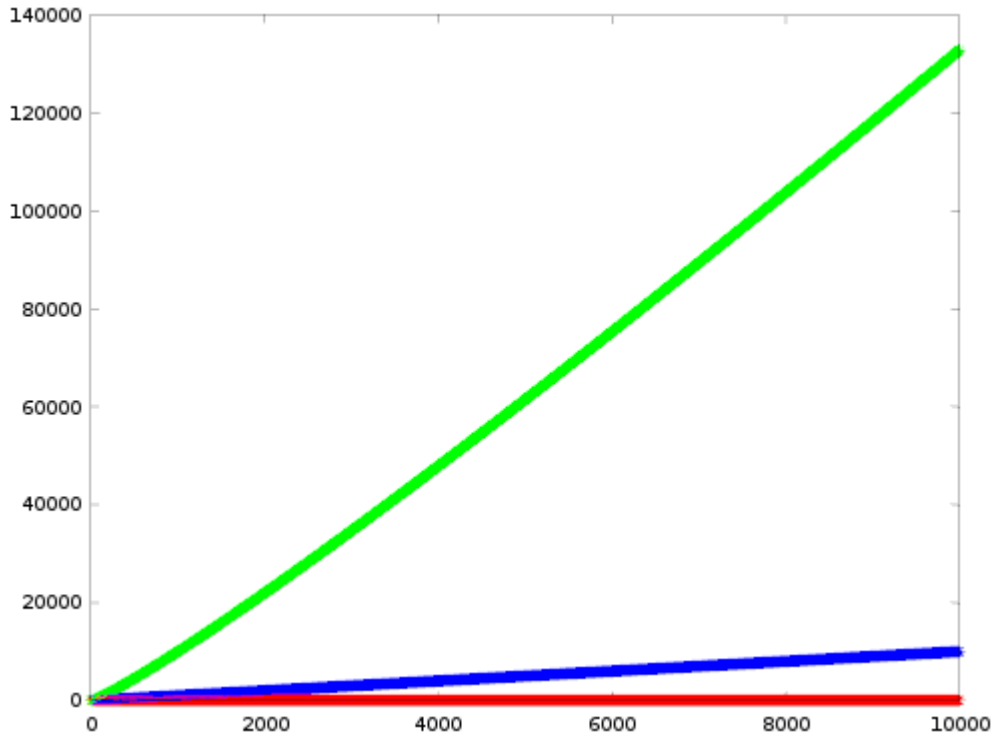
```c
#include "stdio.h"

int G3(int N) {
    int i, j, sum = 0;

    for (i=1; i<=N; i*=2) {
        for (j=0; j<N; j++) {
            sum++;
        }
    }

    return sum;
}


int main(void) {
    printf("%d\n", G3(64) );
    printf("%d\n", G3(1024) );
    printf("%d\n", G3(1000000) );

    return 0;
}
```

Linearithmic

G3(N) <= N(lg(N)+1)

```
448
11264
20000000
```

log(N) , N , N*log(N)

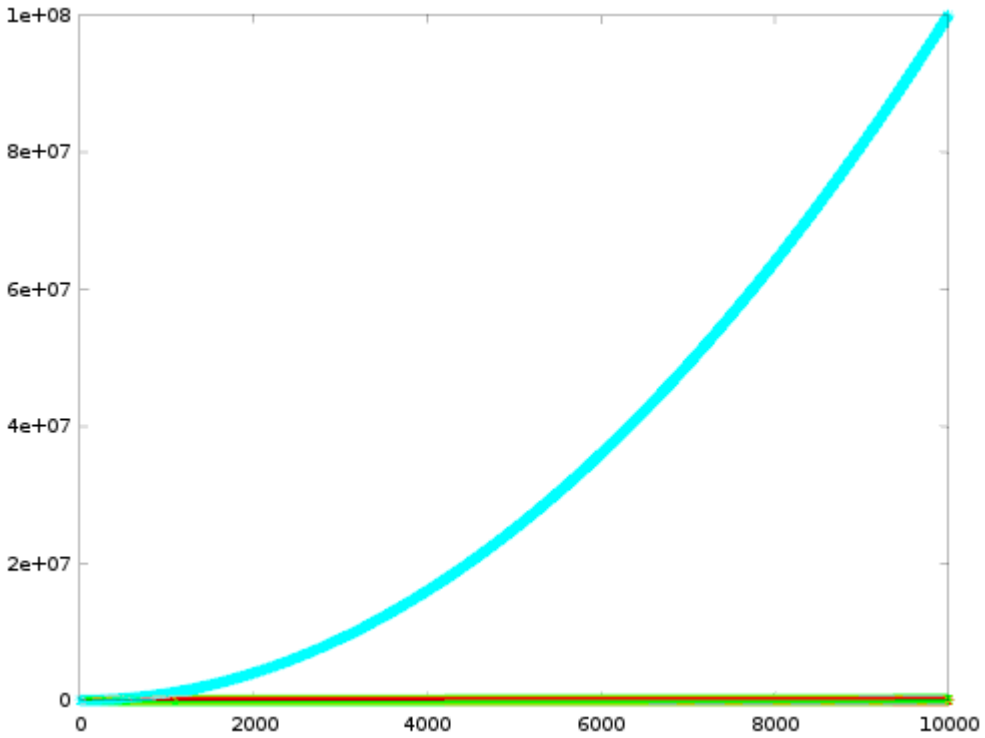


n*log(n)

n

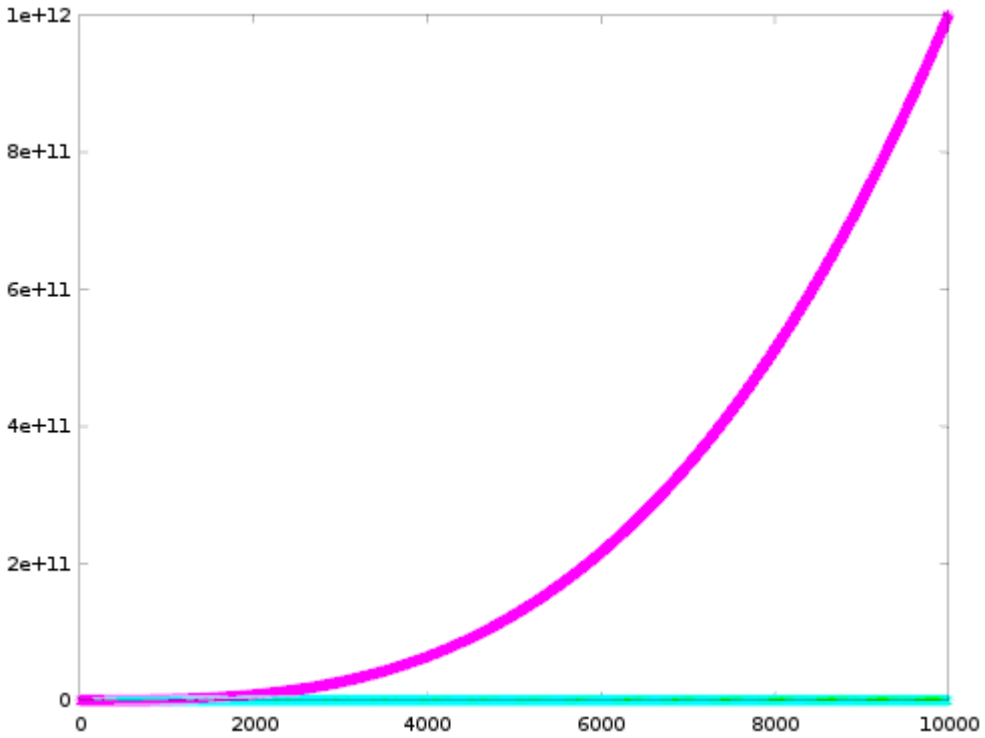
log(n)

log(N) , N , N*log(N), N²



n^2

log(N) , N , N*log(N), N², N³



n^3

description	order of growth	typical code framework	description	example
constant	1	`a = b + c;`	*statement*	*add two numbers*
logarithmic	$\log N$		*divide in half*	*binary search*

linear	N	`double max = a[0];`
		`for (int i = 1; i < N; i++)`
		`  if (a[i] > max) max = a[i];`
			loop	*find the maximum*

linearithmic	$N \log N$		*divide and conquer*	*mergesort*

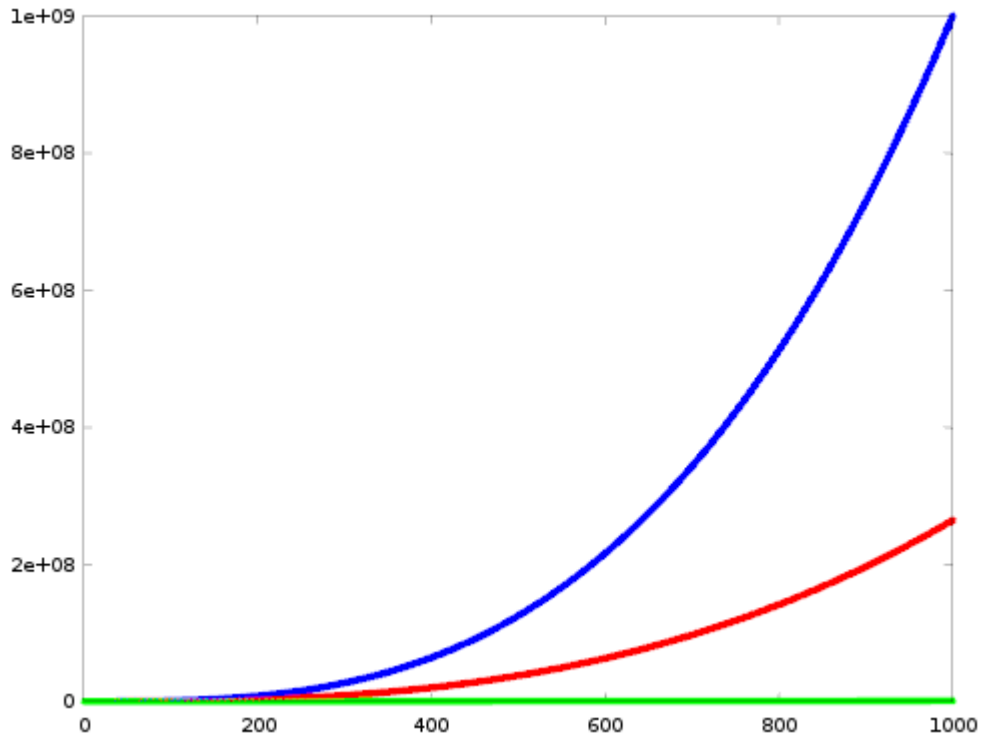
quadratic	N^2	`for (int i = 0; i < N; i++)`
		`  for (int j = i+1; j < N; j++)`
		`  if (a[i] + a[j] == 0)`
		`  cnt++;`
			double loop	*check all pairs*

cubic	N^3	`for (int i = 0; i < N; i++)`
		`  for (int j = i+1; j < N; j++)`
		`  for (int k = j+1; k < N; k++)`
		`  if (a[i] + a[j] + a[k] == 0)`
		`  cnt++;`
			triple loop	*check all triples*

exponential	2^N		*exhasutive search*	*check all subsets*

(*) Fonte: http://algs4.cs.princeton.edu/14analysis/

N^2, $N^{2.8074}$, N^3



n^3

$n^{2.8074}$

n^2

Strassen algorithm

From Wikipedia, the free encyclopedia

Not to be confused with the Schönhage–Strassen algorithm for multiplication of polynomials.

In linear algebra, the **Strassen algorithm**, named after Volker Strassen, is an algorithm for matrix multiplication. It is faster than the standard matrix multiplication algorithm and is useful in practice for large matrices, but would be slower than the fastest known algorithms for extremely large matrices.

Strassen's algorithm works for any ring, such as plus/multiply, but not all semirings, such as min/plus or boolean algebra, where the naive algorithm still works, and so called combinatorial matrix multiplication.

Contents [hide]

$$O([7+o(1)]^n) = O(N^{\log_2 7+o(1)}) \approx O(N^{2.8074}) .$$

History [edit]

Volker Strassen first published this algorithm in 1969 and proved that the n^3 general matrix multiplication algorithm wasn't optimal. The **Strassen algorithm** is only slightly better, but its publication resulted in much more research about matrix multiplication that led to faster approaches, such as the Coppersmith-Winograd algorithm.

Coppersmith–Winograd algorithm

From Wikipedia, the free encyclopedia
(Redirected from Coppersmith-Winograd algorithm)

In linear algebra, the **Coppersmith–Winograd algorithm**, named after Don Coppersmith and Shmuel Winograd, was the asymptotically fastest known matrix multiplication algorithm until 2010. It can multiply two $n \times n$ matrices in $O(n^{2.375477})$ time [1] (see Big O notation). This is an improvement over the naïve $O(n^3)$ time algorithm and the $O(n^{2.807355})$ time Strassen algorithm. Algorithms with better asymptotic running time than the Strassen algorithm are rarely used in practice, because the large constant factors in their running times make them impractical.[2] It is possible to improve the exponent further; however, the exponent must be at least 2 (because an $n \times n$ matrix has n^2 values, and all of them have to be read at least once to calculate the exact result).

In 2010, Andrew Stothers gave an improvement to the algorithm, $O(n^{2.374})$.[3][4] In 2011, Virginia Williams combined a mathematical short-cut from Stothers' paper with her own insights and automated optimization on computers, improving the bound to $O(n^{2.3728642})$.[5] In 2014, François Le Gall simplified the methods of Williams and obtained an improved bound of $O(n^{2.3728639})$.[6]

The Coppersmith–Winograd algorithm is frequently used as a building block in other algorithms to prove theoretical time bounds. However, unlike the Strassen algorithm, it is not used in practice because it only provides an advantage for matrices so large that they cannot be processed by modern hardware.[7]

Henry Cohn, Robert Kleinberg, Balázs Szegedy and Chris Umans have re-derived the Coppersmith–Winograd algorithm using a group-theoretic construction. They also showed that either of two different conjectures would imply that the optimal exponent of matrix multiplication is 2, as has long been suspected. However, they were not able to formulate a specific solution leading to a better running-time than Coppersmith-Winograd at the time.[8]

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 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...n-1], para n>=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

```c
#include "stdio.h"

int main()
{
	int A[10] = {6,7,8,9,0,1,2,3,4,5};
	int max = A[0];

	for(int i=1; i<10; i++)
	{
		if (max < A[i])
			max = A[i];
	}

	printf("valor maximo: %d", 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 }
```

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

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

Exemplo: Maior elemento

```c
#include "stdio.h"

int main()
{
        int A[10];
        A[0] = 6;
        A[1] = 7;
        A[2] = 8;
        A[3] = 9;
        A[4] = 0;
        A[5] = 1;
        A[6] = 2;
        A[7] = 3;
        A[8] = 4;
        A[9] = 5;

        int max = A[0];

        for(int i=1; i<10; i++)
        {
                if (max < A[i])
                        max = A[i];
        }

        printf("valor maximo: %d", max);
}
```

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

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 unifome (**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 elemento

```
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
  - 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:

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

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:

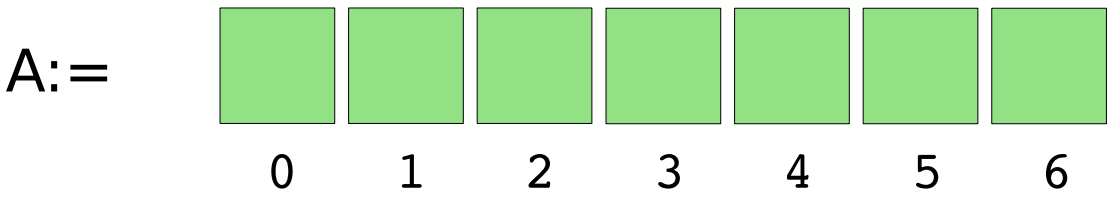
$$f(n) = 1 \times p_1 + 2 \times p_2 + 3 \times p_3 + \cdots + n \times p_n$$

  - 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>=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:

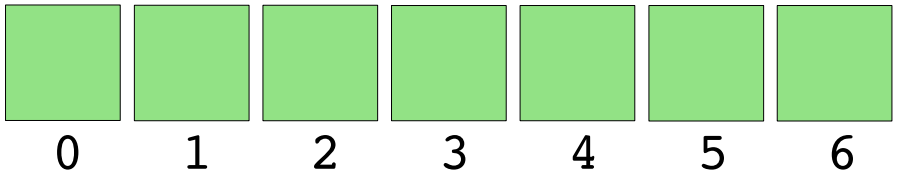
- Melhor caso: $f(n) = (n-1)$ *Quando os elementos estão em ordem crescente.*
- Pior caso: $f(n) = 2(n-1)$ *Quando os elementos estão em ordem decrescente.*
- Caso médio: $f(n) = (n-1) + (\frac{n-1}{2}) = \frac{3n}{2} - \frac{3}{2}$

Quando metade das vezes max>=A[i]

Maior e Menor elementos (versão 3)

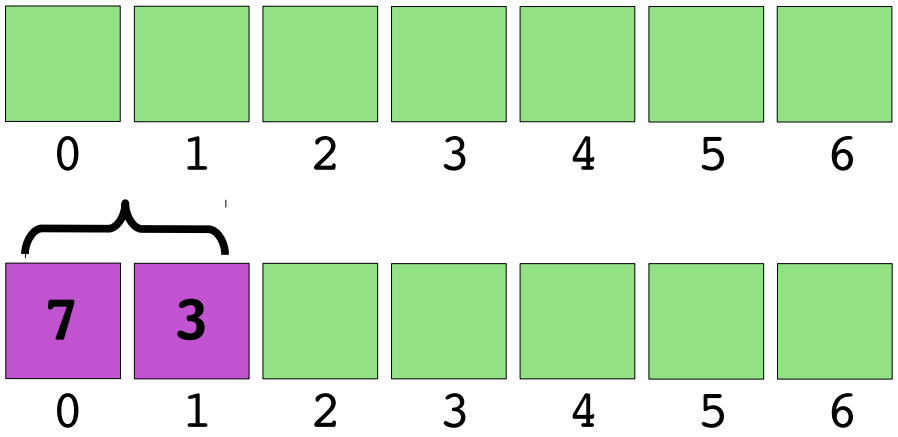
A:

=



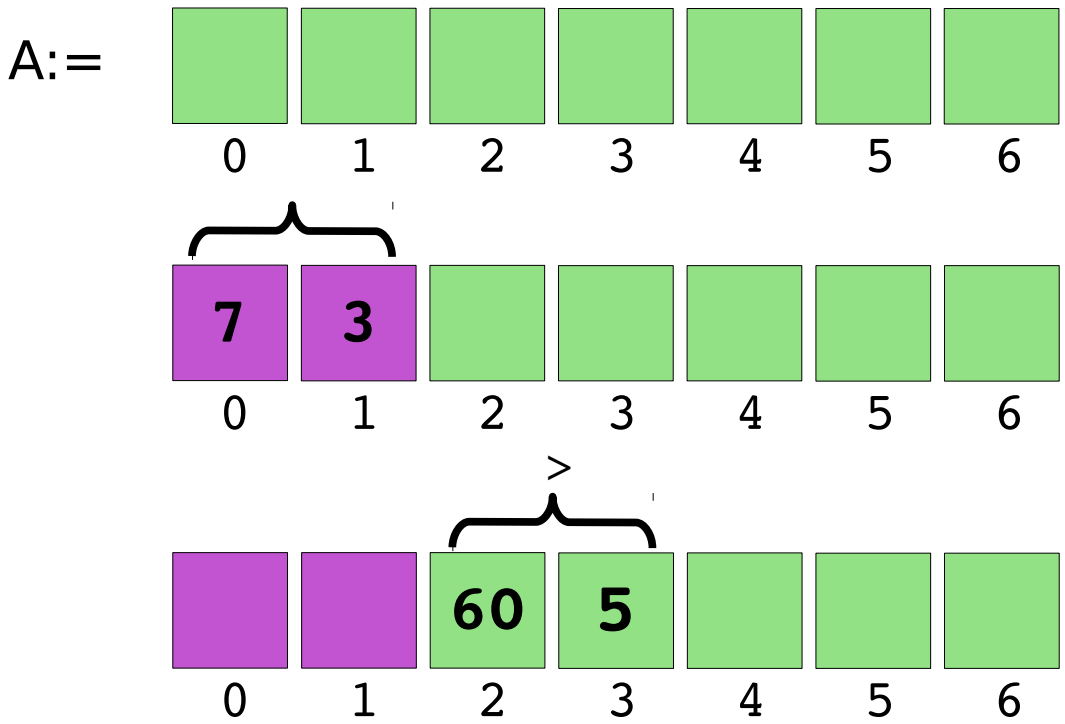
Maior e Menor elementos (versão 3)

A:
=

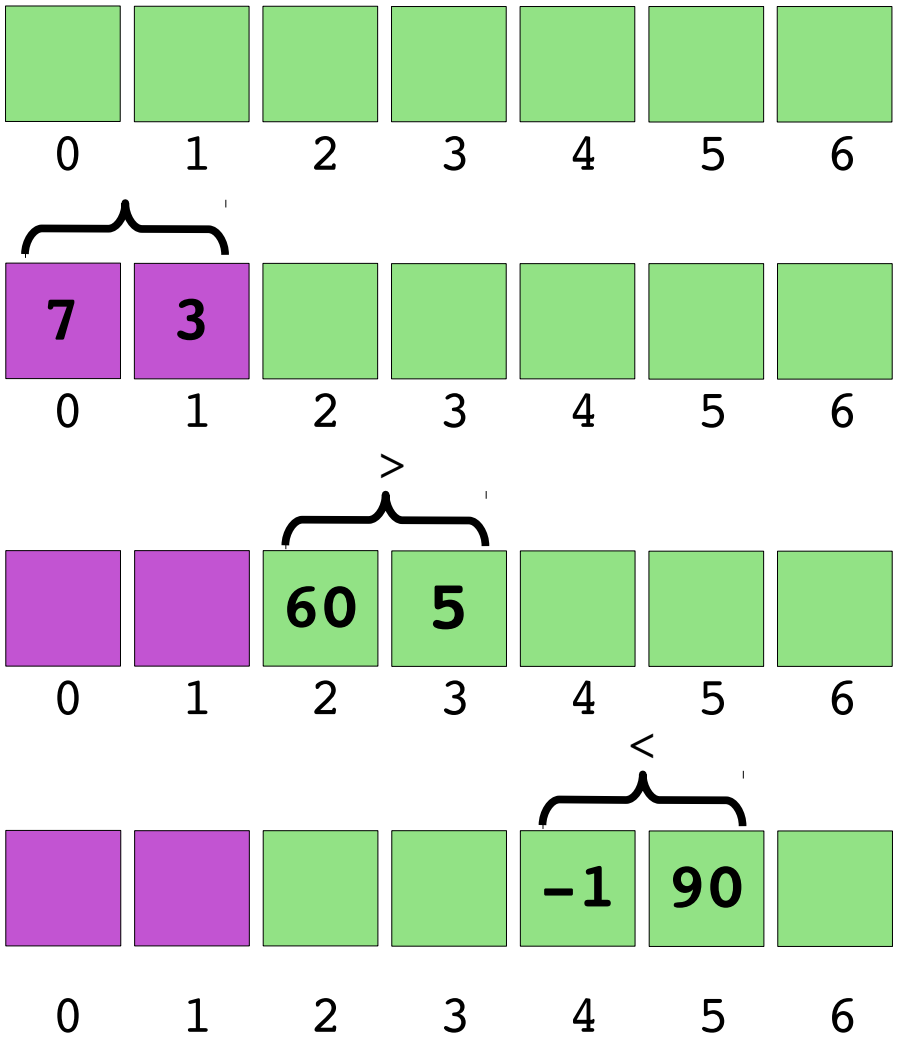


Min = 3
Max = 7

Maior e Menor elementos (versão 3)



Maior e Menor elementos (versão 3)

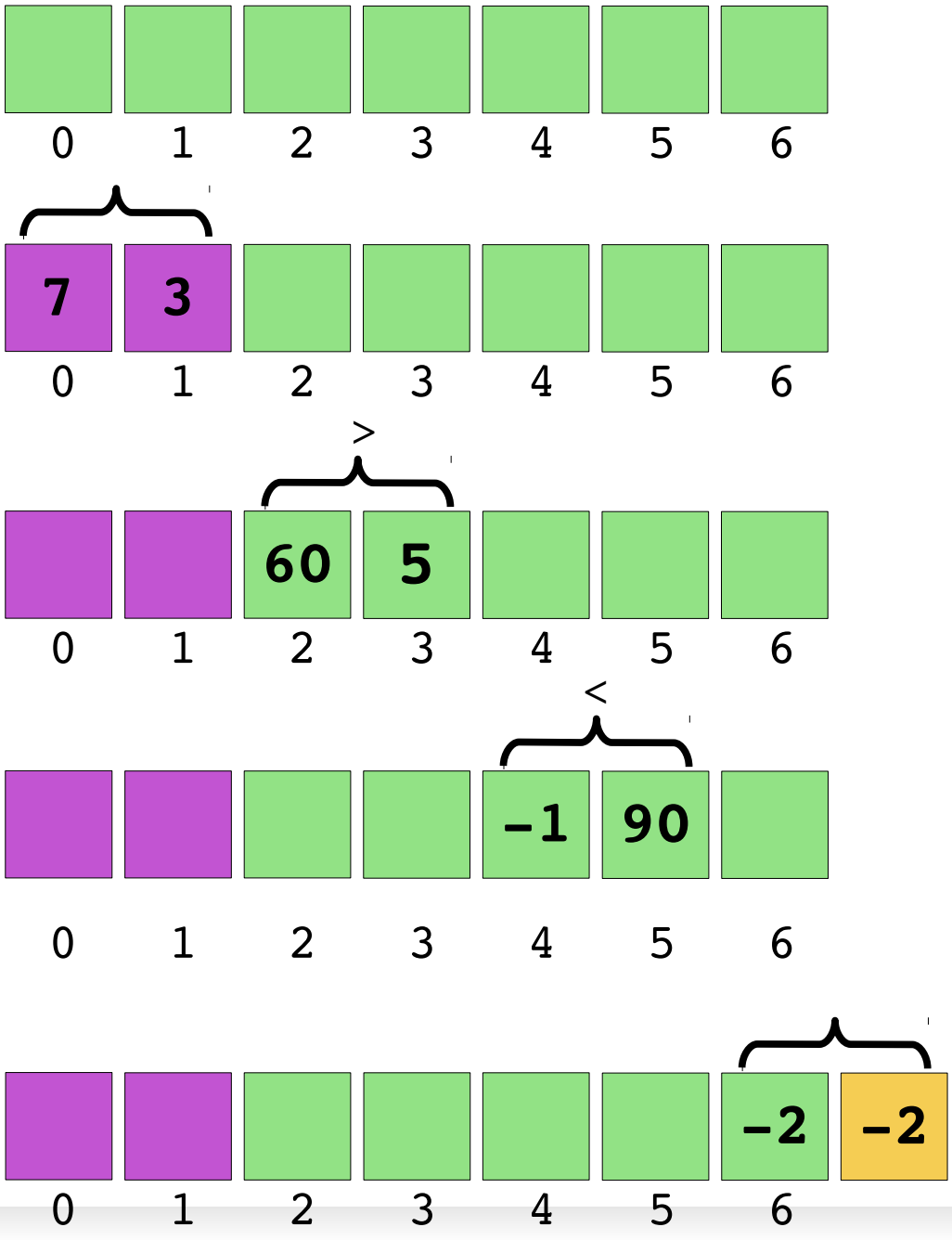


Min = 3
Max = 7

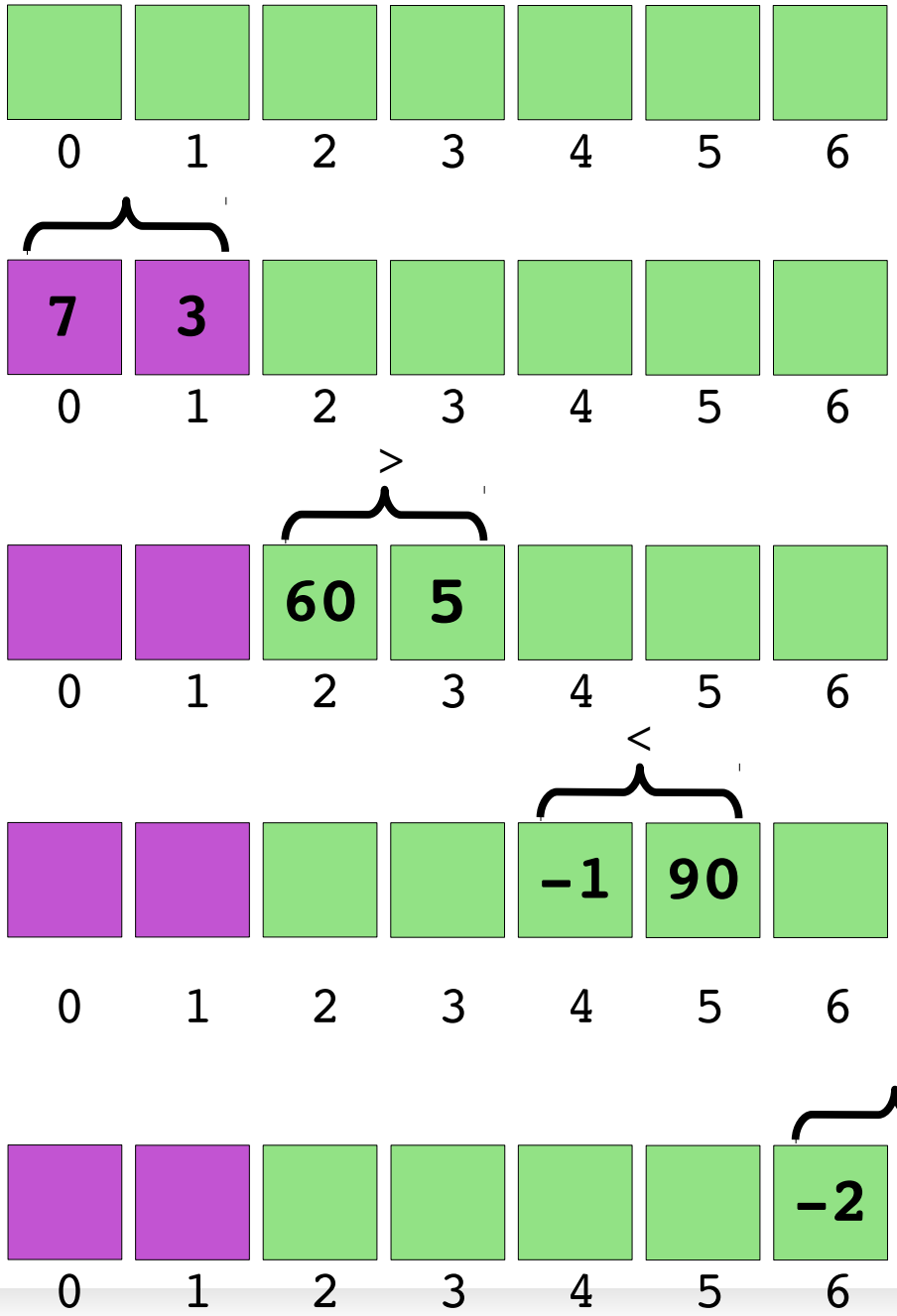
Min = 3
Max = 60

Min = -1
Max = 90

Maior e Menor elementos (versão 3)



Maior e Menor elementos (versão 3)



Comparações =10

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];
    }
    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);
}
```

1 comparação

(n-2)/2comparações

(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

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);
}
```

1 comparação

(n-2)/2comparações

(n-2)/2 + (n-2/2) comparações

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

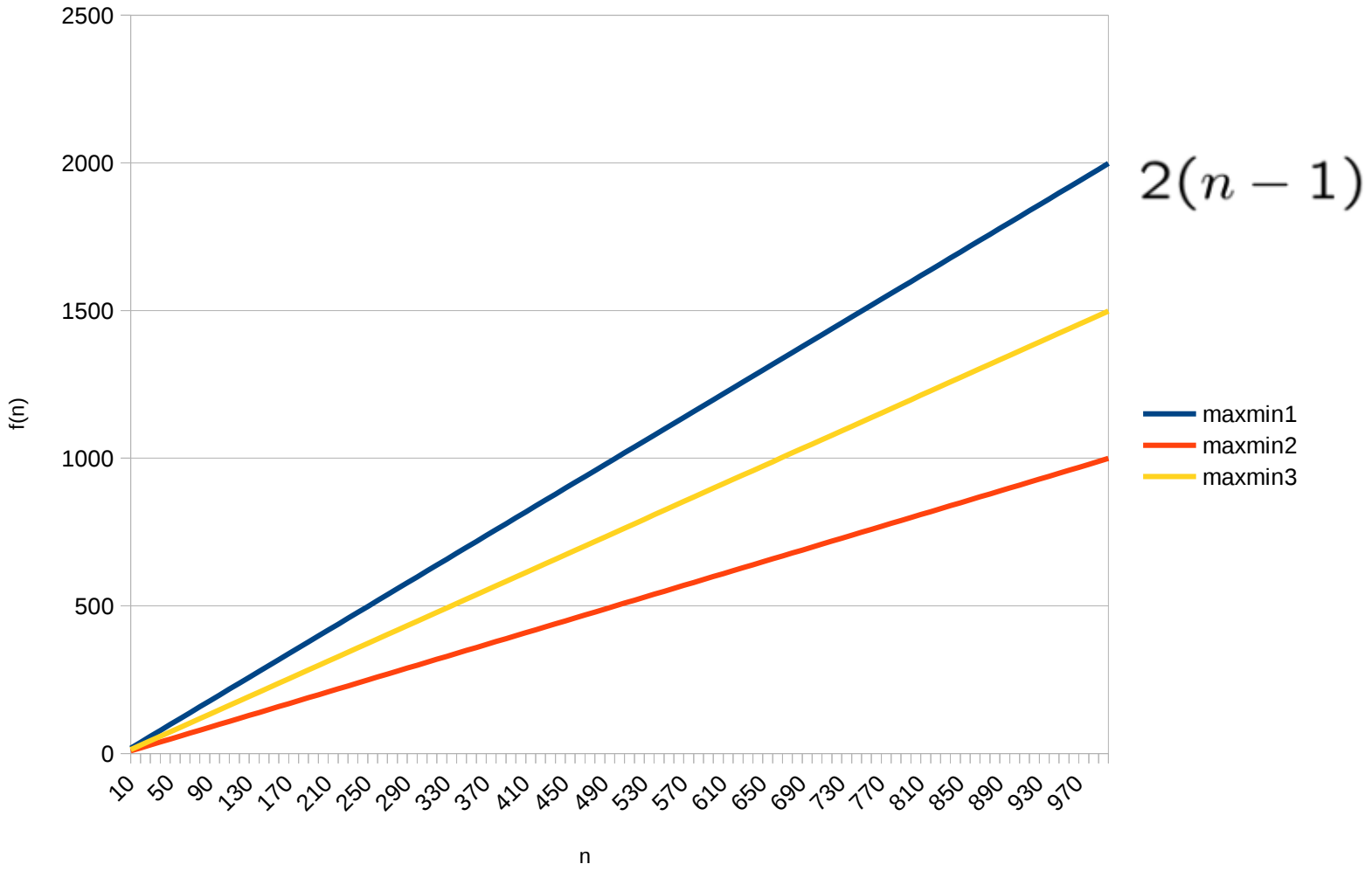
- Melhor caso
- Pior caso
- Caso médio

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

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:

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

Comportamento assintótico de funções

Comportamento assintótico de funções

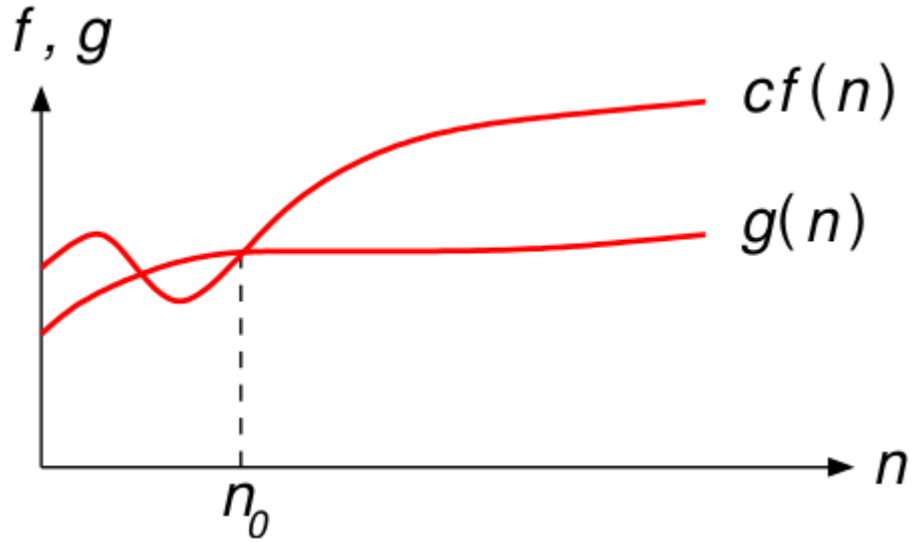
- A análise de algoritmos é realizada p**ara valores grandes de n**.
- Estudamos o comportamento assintótico das **funções de custo**.
- O comportamento assintótico de **f(n)** representa o limite do comportamento de custo, quando **n** cresce.

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:

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

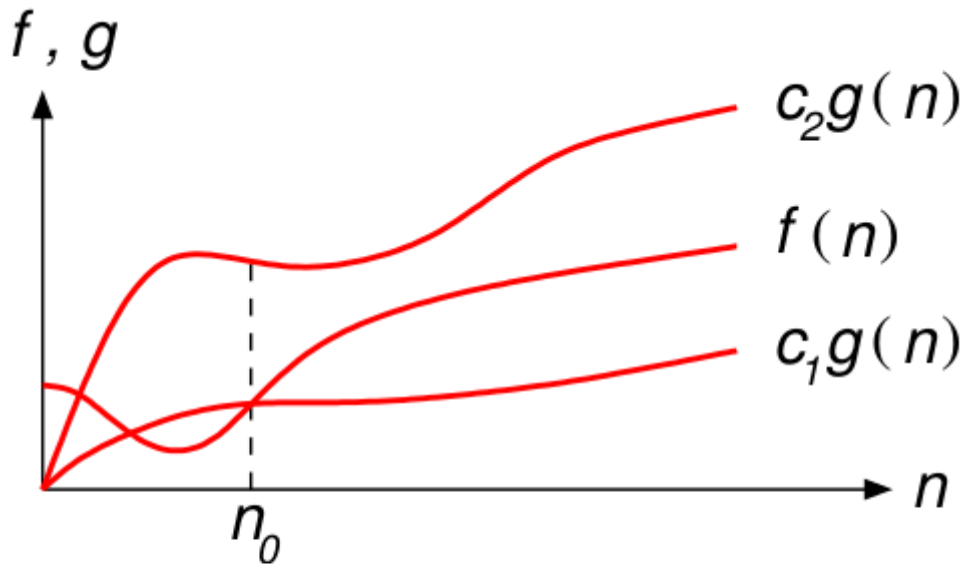


Notação assintótica de funções

Existem 3 notações assintóticas de funções:

- Notação Θ
- Notação O $(\text{`}O \ grande\text{'})$
- Notação Ω

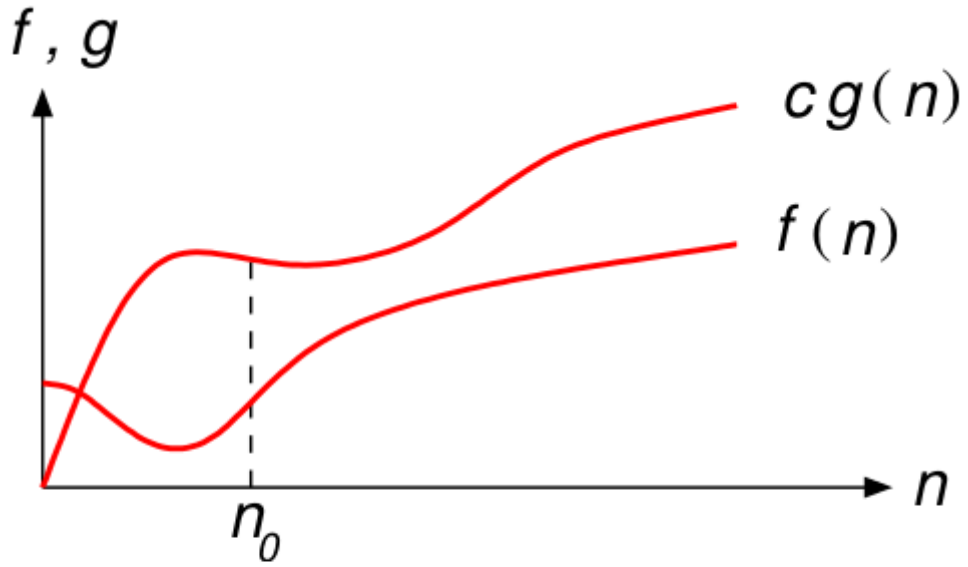
Notação Θ



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

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)

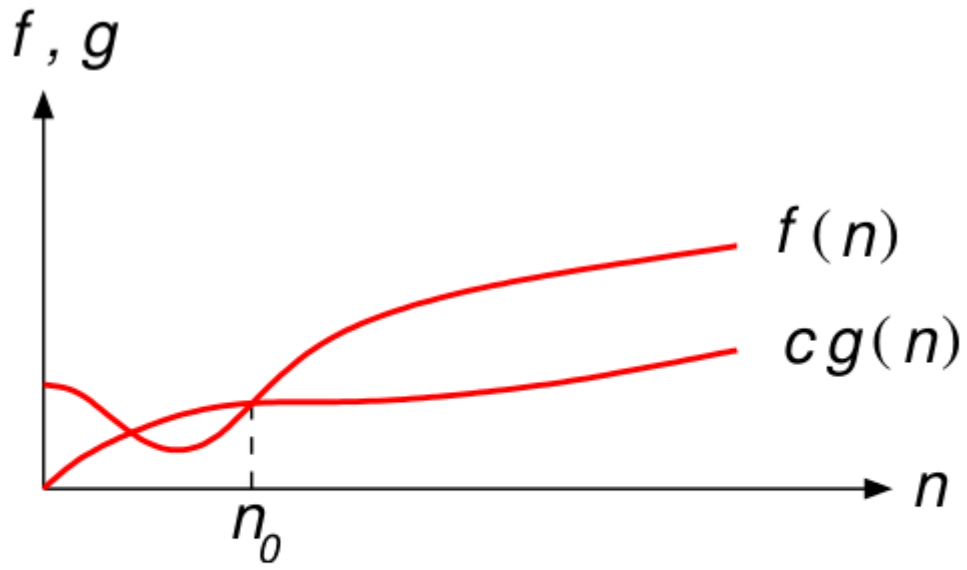
O é usada para expressar o tempo de execução de um algoritmo no **pior caso**, está se definindo também o limite (superior) do tempo de execução desse algoritmo **para todas as entradas**.

Notação O (‘O grande’)

$$
\begin{aligned}
f(n) &= O(f(n)) \\
c \times O(f(n)) &= O(f(n)) \quad c = constante \\
O(f(n)) + O(f(n)) &= O(f(n)) \\
O(O(f(n)) &= O(f(n)) \\
O(f(n)) + O(g(n)) &= O(max(f(n), g(n))) \\
O(f(n))O(g(n)) &= O(f(n)g(n)) \\
f(n)O(g(n)) &= O(f(n)g(n))
\end{aligned}
$$

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,

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

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