# Aula 20
# Exercícios de custos de um algoritmo

Prof. João Henrique Kleinschmidt
Material elaborado pelo prof. Jesús P. Mena-Chalco
3Q-2018

# 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
```

# 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

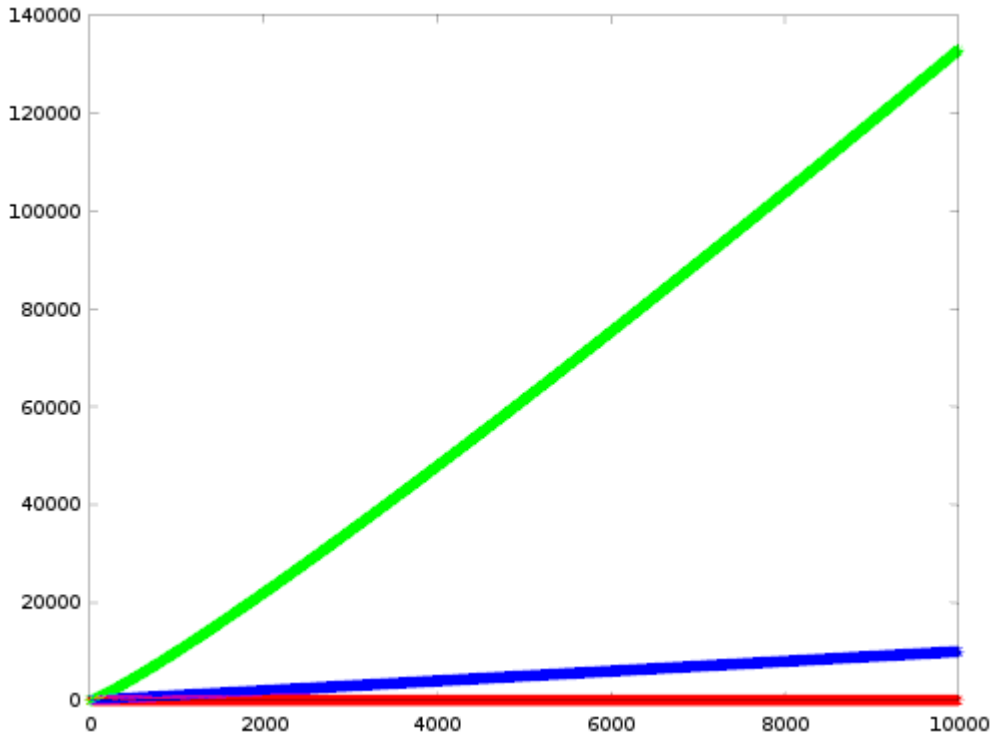
```c
#include "stdio.h"

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)

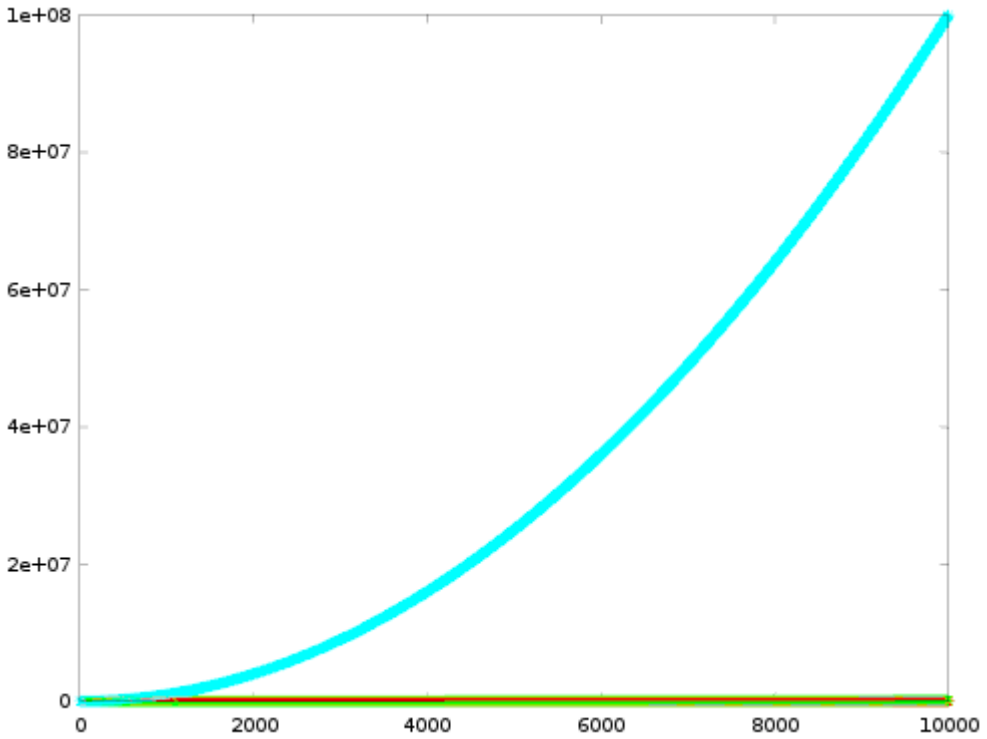



n*log(n)

n

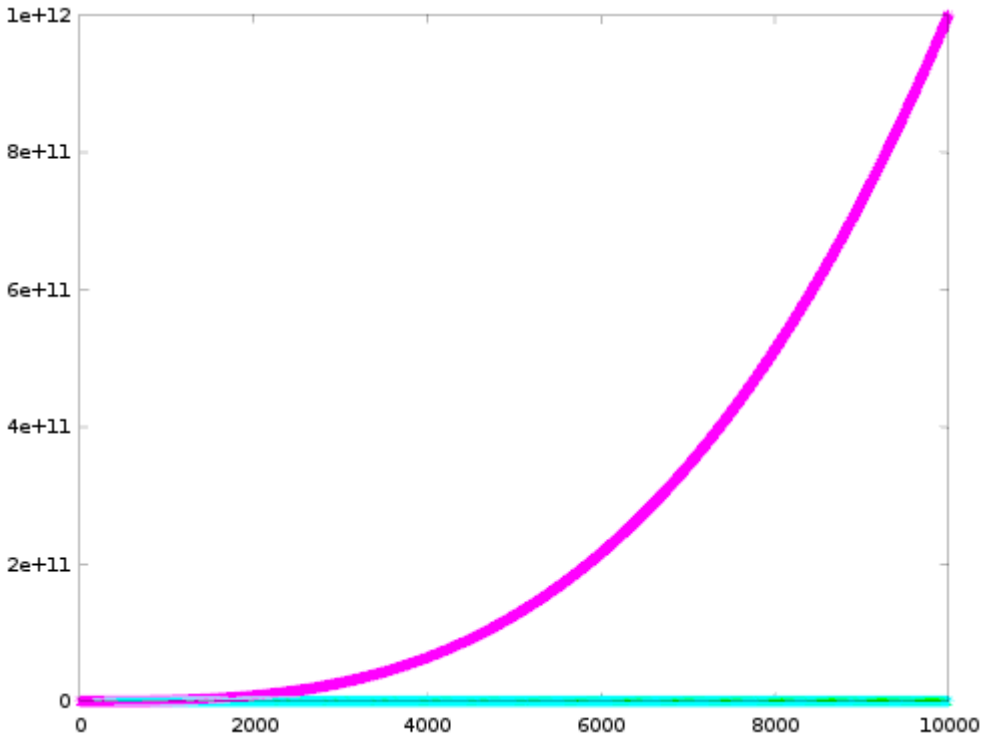
log(n)

# log(N) , N , N*log(N), N²




$n^2$

# log(N) , N , N*log(N), $N^2$, $N^3$

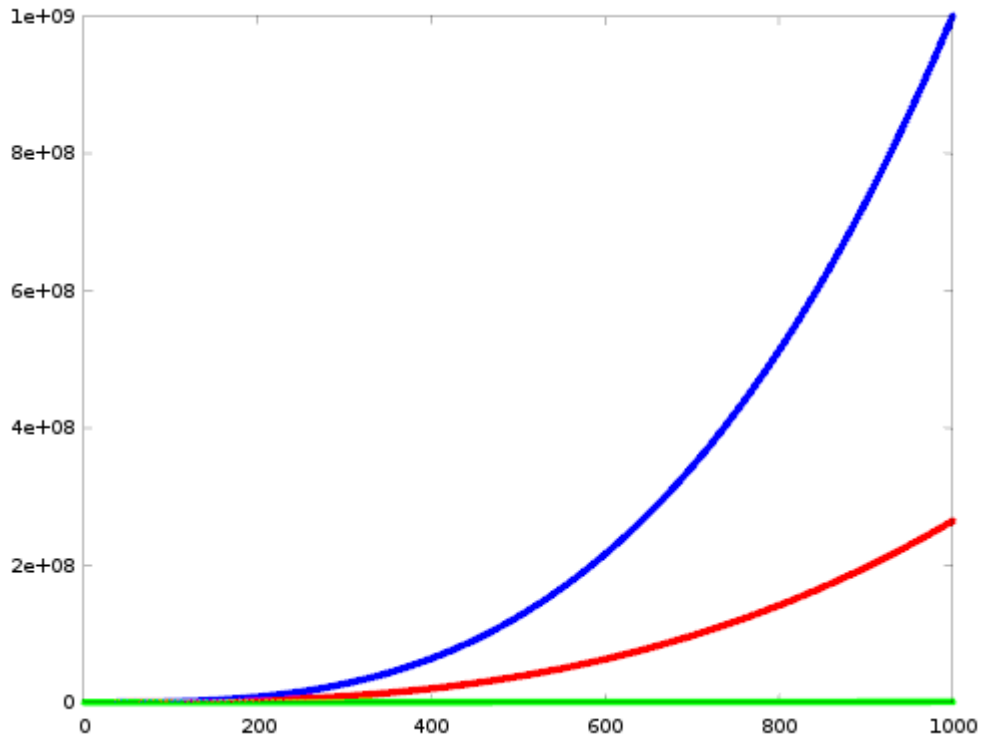



$n^3$

| 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];`<br>`for (int i = 1; i < N; i++)`<br>`  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++)`<br>`  for (int j = i+1; j < N; j++)`<br>`  if (a[i] + a[j] == 0)`<br>`  cnt++;` | *double loop* | *check all pairs* |
| *cubic* | $N^3$ | `for (int i = 0; i < N; i++)`<br>`  for (int j = i+1; j < N; j++)`<br>`  for (int k = j+1; k < N; k++)`<br>`  if (a[i] + a[j] + a[k] == 0)`<br>`  cnt++;` | *triple loop* | *check all triples* |
| *exponential* | $2^N$ | | *exhasutive search* | *check all subsets* |

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

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

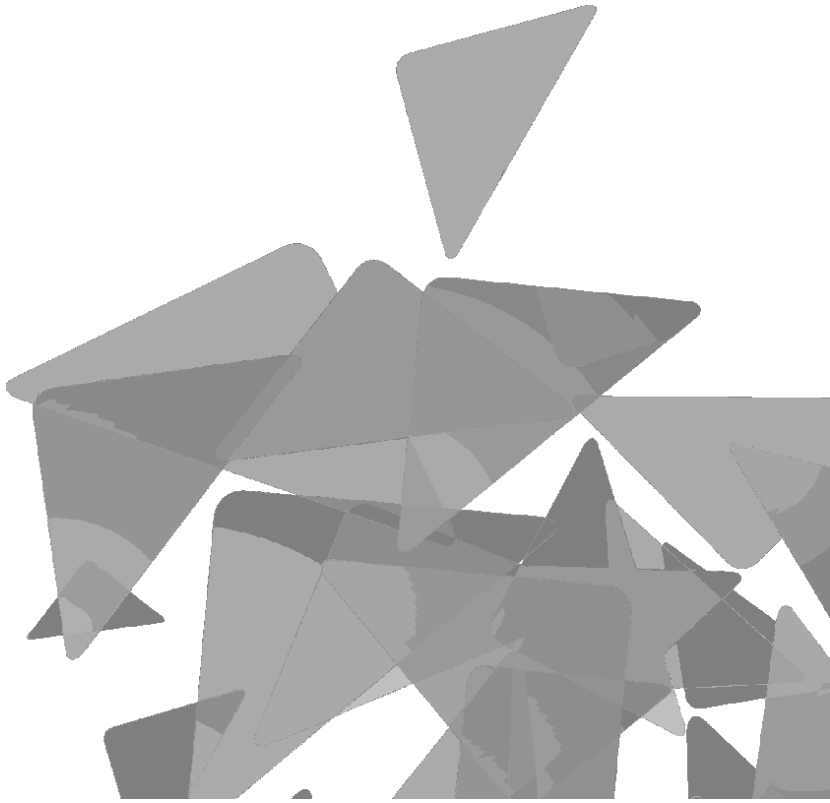



$n^3$

$n^{2.8074}$

$n^2$

# 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***.

- 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.

- → 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 supõe-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.

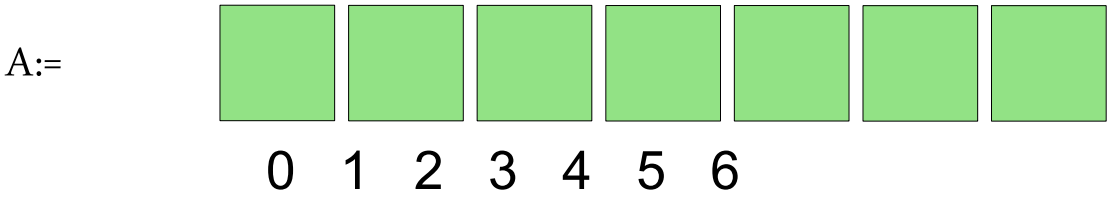
  - Melhor caso: $f(n) = 1$ — Quando o elemento procurado é o primeiro consultado
  - Pior caso: $f(n) = n$ — Quando o elemento procurado é o último consultado
  - Caso médio: $f(n) = \frac{n+1}{2}$

# Exemplo: Busca de um registro (caso médio)

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

# 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)

```c
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]*