Provas de Correção de Algoritmos e Análise de Complexidade Assintótica

Ana Paula Tomás

Desenho e Análise de Algoritmos 2018/19

Setembro 2018

Exemplo: Posição do máximo

Problema: Posição do máximo

Escreva uma função $\textsc{PosMax}(v, k, n)$ para obter o índice da primeira ocorrência do elemento máximo no segmento $v[k], v[k+1], \ldots, v[n]$, do vetor v. Admita que $k \leq n$ e que o segmento está dentro dos limites de v.

Resposta 1

```
PosMax(v, k, n)
    pmax ← k;
    Para i ← k + 1 até n fazer
        Se v[i] > v[pmax] então
            pmax ← i;
    retorna pmax;
```

Resposta 2

```
PosMax(v, k, n)
    pmax ← k;
    i ← k + 1;
    Enquanto i ≤ n fazer
        Se v[i] > v[pmax] então
            pmax ← i;
        i ← i + 1;
    retorna pmax;
```

Exemplo: Posição do máximo

Problema: Posição do máximo

Escreva uma função $\textsc{PosMax}(v, k, n)$ para obter o índice da primeira ocorrência do elemento máximo no segmento $v[k], v[k+1], \ldots, v[n]$, do vetor v. Admita que $k \leq n$ e que o segmento está dentro dos limites de v.

Resposta 1

$\textsc{PosMax}(v, k, n)$
- $pmax \leftarrow k$;
- Para $i \leftarrow k+1$ até n fazer
  - Se $v[i] > v[pmax]$ então
    - $pmax \leftarrow i$;
- retorna $pmax$;

Resposta 2

$\textsc{PosMax}(v, k, n)$
- $pmax \leftarrow k$;
- $i \leftarrow k+1$;
- Enquanto $i \leq n$ fazer
  - Se $v[i] > v[pmax]$ então
    - $pmax \leftarrow i$;
  - $i \leftarrow i+1$;
- retorna $pmax$;

Exemplo: Posição do máximo

Problema: Posição do máximo

Escreva uma função $\textsc{PosMax}(v, k, n)$ para obter o índice da primeira ocorrência do elemento máximo no segmento $v[k], v[k+1], \ldots, v[n]$, do vetor v. Admita que $k \leq n$ e que o segmento está dentro dos limites de v.

Resposta 1

$\textsc{PosMax}(v, k, n)$
- $pmax \leftarrow k$;
- Para $i \leftarrow k+1$ até n fazer
  - Se $v[i] > v[pmax]$ então
    - $pmax \leftarrow i$;
- retorna $pmax$;

Resposta 2

$\textsc{PosMax}(v, k, n)$
- $pmax \leftarrow k$;
- $i \leftarrow k+1$;
- Enquanto $i \leq n$ fazer
  - Se $v[i] > v[pmax]$ então
    - $pmax \leftarrow i$;
  - $i \leftarrow i+1$;
- retorna $pmax$;

Exemplo: Posição do máximo

Problema: Posição do máximo

Escreva uma função $\textsc{PosMax}(v, k, n)$ para obter o índice da primeira ocorrência do elemento máximo no segmento $v[k], v[k+1], \ldots, v[n]$, do vetor v. Admita que $k \leq n$ e que o segmento está dentro dos limites de v.

Resposta 2 (equivalente a Resp 1 segundo a semântica dos ciclos **Para** e **Enquanto**)

```
PosMax(v, k, n)
    pmax ← k;
    i ← k + 1;
    Enquanto i ≤ n fazer
        Se v[i] > v[pmax] então
            pmax ← i;
        i ← i + 1;
    retorna pmax;
```

Exemplo: Posição do máximo

```
PosMax(v, k, n)
1.    pmax ← k;
2.    i ← k + 1;
3.    Enquanto i ≤ n fazer
4.       Se v[i] > v[pmax] então
5.          pmax ← i;
6.       i ← i + 1;
7.    retorna pmax;
```

Invariante de ciclo:

Quando a condição de paragem do ciclo está a ser testada para um dado valor de i (na linha 3), o valor de pmax é o índice da primeira ocorrência do máximo de $v[k], v[k+1], \ldots, v[i-1]$, *sendo isto verdade para todo valor de i tal que* $k+1 \leq i \leq n+1$.

Exemplo: Posição do máximo

POSMAX(v, k, n)

```
1.  pmax ← k;
2.  i ← k + 1;
3.  Enquanto i ≤ n fazer
4.     Se v[i] > v[pmax] então
5.        pmax ← i;
6.     i ← i + 1;
7.  retorna pmax;
```

Invariante de ciclo:

Quando a condição de paragem do ciclo está a ser testada para um dado valor de i (na linha 3), o valor de $pmax$ é o índice da primeira ocorrência do máximo de $v[k], v[k+1], \ldots, v[i-1]$, sendo isto verdade para todo valor de i tal que $k+1 \leq i \leq n+1$.

Exemplo: Posição do máximo

POSMAX(v, k, n)

1. $pmax \leftarrow k$;
2. $i \leftarrow k + 1$;
3. Enquanto $i \leq n$ fazer
4. Se $v[i] > v[pmax]$ então
5. $pmax \leftarrow i$;
6. $i \leftarrow i + 1$;
7. retorna $pmax$;

Esqueleto da prova do invariante (por indução matemática):

Se provarmos (i) e (ii), concluimos (pelo Princípio de Indução) que o invariante se verifica em todas as iterações do ciclo.

- (i) **Caso de base:** o invariante verifica-se no início do ciclo.
- (ii) **Hereditariedade:** o invariante é preservado em cada iteração do ciclo (se verifica numa iteração então verifica-se na seguinte).

Exemplo: Posição do máximo

POSMAX(v, k, n)

1. $pmax \leftarrow k$;
2. $i \leftarrow k + 1$;
3. Enquanto $i \leq n$ fazer
4. Se $v[i] > v[pmax]$ então
5. $pmax \leftarrow i$;
6. $i \leftarrow i + 1$;
7. retorna $pmax$;

Esqueleto da prova do invariante (por indução matemática):

Se provarmos (i) e (ii), concluimos (pelo Princípio de Indução) que o invariante se verifica em todas as iterações do ciclo.

- (i) **Caso de base:** o invariante verifica-se no início do ciclo.
- (ii) **Hereditariedade:** o invariante é preservado em cada iteração do ciclo (se verifica numa iteração então verifica-se na seguinte).

Exemplo: Posição do máximo

```
POSMAX(v, k, n)
    pmax ← k;
    i ← k + 1;
    Enquanto i ≤ n fazer
        Se v[i] > v[pmax] então
            pmax ← i;
        i ← i + 1;
    retorna pmax;
```

Esqueleto da prova do invariante (por indução matemática):

Se provarmos (i) e (ii), concluimos (pelo Princípio de Indução) que o invariante se verifica em todas as iterações do ciclo.

- (i) **Caso de base:** o invariante verifica-se no início do ciclo.
- (ii) **Hereditariedade:** o invariante é preservado em cada iteração do ciclo (se verifica numa iteração então verifica-se na seguinte).

Exemplo: Posição do máximo

POSMAX(v, k, n)

```
pmax ← k;
i ← k + 1;
Enquanto i ≤ n fazer
    Se v[i] > v[pmax] então
        pmax ← i;
    i ← i + 1;
retorna pmax;
```

Esqueleto da prova do invariante (por indução matemática):

Se provarmos (i) e (ii), concluimos (pelo Princípio de Indução) que o invariante se verifica em todas as iterações do ciclo.

- (i) **Caso de base:** o invariante verifica-se no início do ciclo.
- (ii) **Hereditariedade:** o invariante é preservado em cada iteração do ciclo (se verifica numa iteração então verifica-se na seguinte).

Exemplo: Posição do máximo

POSMAX(v, k, n)

```
1. pmax ← k;
2. i ← k + 1;
3. Enquanto i ≤ n fazer
4.     Se v[i] > v[pmax] então
5.         pmax ← i;
6.     i ← i + 1;
7. retorna pmax;
```

Propriedade (**invariante de ciclo**): Quando a condição de paragem do ciclo está a ser testada para um dado valor de i (na linha 3), o valor de $pmax$ é o índice da primeira ocorrência do máximo de $v[k], v[k+1], \ldots, v[i-1]$, sendo tal verdade para todo valor de i tal que $k+1 \leq i \leq n+1$.

Prova (por indução):

- (i) Caso de base:

  No ínicio do ciclo, $pmax = k$ e $i = k+1$, sendo verdade que $pmax$ guarda o índice da primeira ocorrência do máximo da sequência $v[k], v[k+1], \ldots, v[i-1]$, já que esta sequência se reduz a $v[k]$.

Exemplo: Posição do máximo

Prova (por indução):

- (ii) Hereditariedade:
  - Suponhamos, como hipótese de indução, que a condição se verifica para $i = i_0$, isto é, $v[pmax] = \max(v[k], \ldots, v[i_0 - 1])$, e que $i_0 < n$. Então, será efetuada uma nova **iteração** (execução do bloco 4–6).
  - Pela hipótese de indução sobre o estado de *pmax* e *i*, concluimos que quando executamos o bloco 4-5, o valor de *pmax* será:
    - preservado se $v[i_0] \leq v[pmax]$.
    - alterado para i_0 se $v[i_0] > v[pmax]$.

    Portanto, se *pmax* tinha o indíce da primeira ocorrência do máximo de $v[k], \ldots, v[i_0 - 1]$ então passará a ter o indíce da primeira ocorrência do máximo de $v[k], \ldots, v[i_0 - 1], v[i_0]$.
  - Na instrução 6, o valor de i passa a ser $i_0 + 1$, e a seguir a condição de paragem do ciclo (linha 3) volta a ser testada. Do que vimos acima, concluimos que a **condição enunciada sobre o estado das variáveis se verifica para $i = i_0 + 1$ se se verificar para $i = i_0$.**

Exemplo: Posição do máximo

Prova (por indução):

- (ii) Hereditariedade:
  - Suponhamos, como hipótese de indução, que a condição se verifica para $i = i_0$, isto é, $v[pmax] = \max(v[k], \ldots, v[i_0 - 1])$, e que $i_0 < n$. Então, será efetuada uma nova **iteração** (execução do bloco 4–6).
  - Pela hipótese de indução sobre o estado de *pmax* e *i*, concluimos que quando executamos o bloco 4-5, o valor de *pmax* será:
    - preservado se $v[i_0] \leq v[pmax]$.
    - alterado para i_0 se $v[i_0] > v[pmax]$.

    Portanto, se *pmax* tinha o indíce da primeira ocorrência do máximo de $v[k], \ldots, v[i_0 - 1]$ então passará a ter o indíce da primeira ocorrência do máximo de $v[k], \ldots, v[i_0 - 1], v[i_0]$.
  - Na instrução 6, o valor de *i* passa a ser $i_0 + 1$, e a seguir a condição de paragem do ciclo (linha 3) volta a ser testada. Do que vimos acima, concluimos que a **condição enunciada sobre o estado das variáveis se verifica para $i = i_0 + 1$ se se verificar para $i = i_0$**.

Exemplo: Posição do máximo

Prova (por indução):

- (ii) Hereditariedade:
  - Suponhamos, como hipótese de indução, que a condição se verifica para $i = i_0$, isto é, $v[pmax] = \max(v[k], \ldots, v[i_0 - 1])$, e que $i_0 < n$. Então, será efetuada uma nova **iteração** (execução do bloco 4–6).
  - Pela hipótese de indução sobre o estado de *pmax* e *i*, concluimos que quando executamos o bloco 4-5, o valor de *pmax* será:
    - preservado se $v[i_0] \leq v[pmax]$.
    - alterado para i_0 se $v[i_0] > v[pmax]$.

    Portanto, se *pmax* tinha o indíce da primeira ocorrência do máximo de $v[k], \ldots, v[i_0 - 1]$ então passará a ter o indíce da primeira ocorrência do máximo de $v[k], \ldots, v[i_0 - 1], v[i_0]$.
  - Na instrução 6, o valor de i passa a ser $i_0 + 1$, e a seguir a condição de paragem do ciclo (linha 3) volta a ser testada. Do que vimos acima, concluimos que a **condição enunciada sobre o estado das variáveis se verifica para $i = i_0 + 1$ se se verificar para $i = i_0$.**

Exemplo: Posição do máximo

Prova (por indução):

- (ii) Hereditariedade:
    - Suponhamos, como hipótese de indução, que a condição se verifica para $i = i_0$, isto é, $v[pmax] = \max(v[k], \ldots, v[i_0 - 1])$, e que $i_0 < n$. Então, será efetuada uma nova **iteração** (execução do bloco 4–6).
    - Pela hipótese de indução sobre o estado de *pmax* e *i*, concluimos que quando executamos o bloco 4-5, o valor de *pmax* será:
        - preservado se $v[i_0] \leq v[pmax]$.
        - alterado para i_0 se $v[i_0] > v[pmax]$.

      Portanto, se *pmax* tinha o indíce da primeira ocorrência do máximo de $v[k], \ldots, v[i_0 - 1]$ então passará a ter o indíce da primeira ocorrência do máximo de $v[k], \ldots, v[i_0 - 1], v[i_0]$.
    - Na instrução 6, o valor de i passa a ser $i_0 + 1$, e a seguir a condição de paragem do ciclo (linha 3) volta a ser testada. Do que vimos acima, concluimos que a **condição enunciada sobre o estado das variáveis se verifica para $i = i_0 + 1$ se se verificar para $i = i_0$.**

Exemplo: Posição do máximo

Prova (por indução):

- (ii) Hereditariedade:
  - Suponhamos, como hipótese de indução, que a condição se verifica para $i = i_0$, isto é, $v[pmax] = \max(v[k], \ldots, v[i_0 - 1])$, e que $i_0 < n$. Então, será efetuada uma nova **iteração** (execução do bloco 4–6).
  - Pela hipótese de indução sobre o estado de *pmax* e *i*, concluimos que quando executamos o bloco 4-5, o valor de *pmax* será:
    - preservado se $v[i_0] \leq v[pmax]$.
    - alterado para i_0 se $v[i_0] > v[pmax]$.

    Portanto, se *pmax* tinha o indíce da primeira ocorrência do máximo de $v[k], \ldots, v[i_0 - 1]$ então passará a ter o indíce da primeira ocorrência do máximo de $v[k], \ldots, v[i_0 - 1], v[i_0]$.
  - Na instrução 6, o valor de i passa a ser $i_0 + 1$, e a seguir a condição de paragem do ciclo (linha 3) volta a ser testada. Do que vimos acima, concluimos que a **condição enunciada sobre o estado das variáveis se verifica para** $i = i_0 + 1$ **se se verificar para** $i = i_0$.

Exemplo: Posição do máximo

Prova (por indução):

- (ii) Hereditariedade:
  - Suponhamos, como hipótese de indução, que a condição se verifica para $i = i_0$, isto é, $v[pmax] = \max(v[k], \ldots, v[i_0 - 1])$, e que $i_0 < n$. Então, será efetuada uma nova **iteração** (execução do bloco 4–6).
  - Pela hipótese de indução sobre o estado de *pmax* e *i*, concluimos que quando executamos o bloco 4-5, o valor de *pmax* será:
    - preservado se $v[i_0] \leq v[pmax]$.
    - alterado para i_0 se $v[i_0] > v[pmax]$.

    Portanto, se *pmax* tinha o indíce da primeira ocorrência do máximo de $v[k], \ldots, v[i_0 - 1]$ então passará a ter o indíce da primeira ocorrência do máximo de $v[k], \ldots, v[i_0 - 1], v[i_0]$.
  - Na instrução 6, o valor de *i* passa a ser $i_0 + 1$, e a seguir a condição de paragem do ciclo (linha 3) volta a ser testada. Do que vimos acima, concluimos que a **condição enunciada sobre o estado das variáveis se verifica para $i = i_0 + 1$ se se verificar para $i = i_0$.**

Exemplo: Posição do máximo

Conclusão

- Pelo princípio de indução, podemos concluir que a propriedade se verifica para todo o valor de $i \geq k+1$. Isto é, na linha 3, *pmax* tem sempre índice da primeira ocorrência do máximo de $v[k], \ldots, v[i-1]$.

Como se pode concluir agora que o programa está correto?

- O ciclo termina quando $i = n+1$ e a seguir executa instrução 7. Para $i = n+1$, **a propriedade diz que o valor de** *pmax* **é o índice da primeira ocorrência do máximo de** $v[k], v[k+1], \ldots, v[n]$.
- A seguir, executa a instrução 7: sai da função e retorna o valor *pmax*. Portanto, o valor que a função retorna está correto.

Exemplo: Posição do máximo

Conclusão

- Pelo princípio de indução, podemos concluir que a propriedade se verifica para todo o valor de $i \geq k+1$. Isto é, na linha 3, *pmax* tem sempre índice da primeira ocorrência do máximo de $v[k], \ldots, v[i-1]$.

Como se pode concluir agora que o programa está correto?

- O ciclo termina quando $i = n+1$ e a seguir executa instrução 7. Para $i = n+1$, **a propriedade diz que o valor de** *pmax* **é o índice da primeira ocorrência do máximo de** $v[k], v[k+1], \ldots, v[n]$.
- A seguir, executa a instrução 7: sai da função e retorna o valor *pmax*. Portanto, o valor que a função retorna está correto.

Exemplo: Posição do máximo

Conclusão

- Pelo princípio de indução, podemos concluir que a propriedade se verifica para todo o valor de $i \geq k+1$. Isto é, na linha 3, *pmax* tem sempre índice da primeira ocorrência do máximo de $v[k], \ldots, v[i-1]$.

Como se pode concluir agora que o programa está correto?

- O ciclo termina quando $i = n+1$ e a seguir executa instrução 7. Para $i = n+1$, **a propriedade diz que o valor de** *pmax* **é o índice da primeira ocorrência do máximo de** $v[k], v[k+1], \ldots, v[n]$.
- A seguir, executa a instrução 7: sai da função e retorna o valor *pmax*. Portanto, o valor que a função retorna está correto.

Exemplo: Posição do máximo

Conclusão

- Pelo princípio de indução, podemos concluir que a propriedade se verifica para todo o valor de $i \geq k+1$. Isto é, na linha 3, *pmax* tem sempre índice da primeira ocorrência do máximo de $v[k], \ldots, v[i-1]$.

Como se pode concluir agora que o programa está correto?

- O ciclo termina quando $i = n+1$ e a seguir executa instrução 7. Para $i = n+1$, **a propriedade diz que o valor de** *pmax* **é o índice da primeira ocorrência do máximo de** $v[k], v[k+1], \ldots, v[n]$.
- A seguir, executa a instrução 7: sai da função e retorna o valor *pmax*. Portanto, o valor que a função retorna está correto.

Exemplo: Posição do máximo

Conclusão

- Pelo princípio de indução, podemos concluir que a propriedade se verifica para todo o valor de $i \geq k+1$. Isto é, na linha 3, *pmax* tem sempre índice da primeira ocorrência do máximo de $v[k], \ldots, v[i-1]$.

Como se pode concluir agora que o programa está correto?

- O ciclo termina quando $i = n+1$ e a seguir executa instrução 7. Para $i = n+1$, **a propriedade diz que o valor de** *pmax* **é o índice da primeira ocorrência do máximo de** $v[k], v[k+1], \ldots, v[n]$.
- A seguir, executa a instrução 7: sai da função e retorna o valor *pmax*. Portanto, o valor que a função retorna está correto.

Exemplo: complexidade temporal de Posmax

Posmax(v, k, n)	Tempos de execução
1. $pmax \leftarrow k$;	c_1: atribuir valor de variável simples a outra
2. $i \leftarrow k+1$;	c_2: avaliar expressão e atribuir valor a variável
3. Enquanto $i \leq n$ fazer	c_3: testar condição e transferir controlo
4. Se $v[i] > v[pmax]$ então	c_4: aceder, testar e transferir controlo
5. $pmax \leftarrow i$;	c_1: atribuir valor de variável simples a outra
6. $i \leftarrow i+1$;	c_6: incrementar valor de variável
7. retorna $pmax$;	c_7: retornar da função com valor de variável

- Executa as instruções 1, 2 e 7 apenas uma vez. Executa teste na linha 3 para $i = k+1, \ldots, n, n+1$, ou seja, $(n+1)-(k+1)+1 = n-k+1$ vezes.
- Executa a instrução 4 para $i = k+1, \ldots, n$, ou seja $n-(k+1)+1 = n-k$ vezes. Análogo para a instrução 6.
- Quantas vezes executa a instrução 5? Depende da instância.
  - No **melhor caso**, executa zero vezes. Ocorre se $v[k]$ é estritamente maior do que os restantes elementos.
  - No **pior caso**, tem-se $v[k] < v[k+1] < \ldots < v[n-1] < v[n]$ e executa $n-k$ vezes.

Exemplo: complexidade temporal de POSMAX

	POSMAX(v, k, n)	**Tempos de execução**
1.	$pmax \leftarrow k$;	c_1: atribuir valor de variável simples a outra
2.	$i \leftarrow k+1$;	c_2: avaliar expressão e atribuir valor a variável
3.	Enquanto $i \leq n$ fazer	c_3: testar condição e transferir controlo
4.	Se $v[i] > v[pmax]$ então	c_4: aceder, testar e transferir controlo
5.	$pmax \leftarrow i$;	c_1: atribuir valor de variável simples a outra
6.	$i \leftarrow i+1$;	c_6: incrementar valor de variável
7.	retorna $pmax$;	c_7: retornar da função com valor de variável

- Executa as instruções 1, 2 e 7 apenas uma vez. Executa teste na linha 3 para $i = k+1, \ldots, n, n+1$, ou seja, $(n+1)-(k+1)+1 = n-k+1$ vezes.
- Executa a instrução 4 para $i = k+1, \ldots, n$, ou seja $n-(k+1)+1 = n-k$ vezes. Análogo para a instrução 6.
- Quantas vezes executa a instrução 5? Depende da instância.
  - No **melhor caso**, executa zero vezes. Ocorre se $v[k]$ é estritamente maior do que os restantes elementos.
  - No **pior caso**, tem-se $v[k] < v[k+1] < \ldots < v[n-1] < v[n]$ e executa $n-k$ vezes.

Exemplo: complexidade temporal de POSMAX

POSMAX(v, k, n)	Tempos de execução
1. $pmax \leftarrow k$;	c_1: atribuir valor de variável simples a outra
2. $i \leftarrow k+1$;	c_2: avaliar expressão e atribuir valor a variável
3. Enquanto $i \leq n$ fazer	c_3: testar condição e transferir controlo
4. Se $v[i] > v[pmax]$ então	c_4: aceder, testar e transferir controlo
5. $pmax \leftarrow i$;	c_1: atribuir valor de variável simples a outra
6. $i \leftarrow i+1$;	c_6: incrementar valor de variável
7. retorna $pmax$;	c_7: retornar da função com valor de variável

- Executa as instruções 1, 2 e 7 apenas uma vez. Executa teste na linha 3 para $i = k+1, \ldots, n, n+1$, ou seja, $(n+1)-(k+1)+1 = n-k+1$ vezes.
- Executa a instrução 4 para $i = k+1, \ldots, n$, ou seja $n-(k+1)+1 = n-k$ vezes. Análogo para a instrução 6.
- Quantas vezes executa a instrução 5? Depende da instância.
  - No **melhor caso**, executa zero vezes. Ocorre se $v[k]$ é estritamente maior do que os restantes elementos.
  - No **pior caso**, tem-se $v[k] < v[k+1] < \ldots < v[n-1] < v[n]$ e executa $n-k$ vezes.

Exemplo: complexidade temporal de POSMAX

POSMAX(v, k, n)	Tempos de execução
1. $pmax \leftarrow k$;	c_1: atribuir valor de variável simples a outra
2. $i \leftarrow k+1$;	c_2: avaliar expressão e atribuir valor a variável
3. Enquanto $i \leq n$ fazer	c_3: testar condição e transferir controlo
4. Se $v[i] > v[pmax]$ então	c_4: aceder, testar e transferir controlo
5. $pmax \leftarrow i$;	c_1: atribuir valor de variável simples a outra
6. $i \leftarrow i+1$;	c_6: incrementar valor de variável
7. retorna $pmax$;	c_7: retornar da função com valor de variável

- Executa as instruções 1, 2 e 7 apenas uma vez. Executa teste na linha 3 para $i = k+1, \ldots, n, n+1$, ou seja, $(n+1)-(k+1)+1 = n-k+1$ vezes.
- Executa a instrução 4 para $i = k+1, \ldots, n$, ou seja $n-(k+1)+1 = n-k$ vezes. Análogo para a instrução 6.
- Quantas vezes executa a instrução 5? Depende da instância.
  - No **melhor caso**, executa zero vezes. Ocorre se $v[k]$ é estritamente maior do que os restantes elementos.
  - No **pior caso**, tem-se $v[k] < v[k+1] < \ldots < v[n-1] < v[n]$ e executa $n-k$ vezes.

Exemplo: complexidade temporal de POSMAX

POSMAX(v, k, n)

```
1. pmax ← k;
2. i ← k + 1;
3. Enquanto i ≤ n fazer
4.     Se v[i] > v[pmax] então
5.         pmax ← i;
6.     i ← i + 1;
7. retorna pmax;
```

Tempos de execução

- c_1: atribuir valor de variável simples a outra
- c_2: avaliar expressão e atribuir valor a variável
- c_3: testar condição e transferir controlo
- c_4: aceder, testar e transferir controlo
- c_1: atribuir valor de variável simples a outra
- c_6: incrementar valor de variável
- c_7: retornar da função com valor de variável

- Executa as instruções 1, 2 e 7 apenas uma vez. Executa teste na linha 3 para $i = k+1, \ldots, n, n+1$, ou seja, $(n+1)-(k+1)+1 = n-k+1$ vezes.
- Executa a instrução 4 para $i = k+1, \ldots, n$, ou seja $n-(k+1)+1 = n-k$ vezes. Análogo para a instrução 6.
- Quantas vezes executa a instrução 5? Depende da instância.
    - No **melhor caso**, executa zero vezes. Ocorre se $v[k]$ é estritamente maior do que os restantes elementos.
    - No **pior caso**, tem-se $v[k] < v[k+1] < \ldots < v[n-1] < v[n]$ e executa $n-k$ vezes.

Exemplo: complexidade temporal de POSMAX

POSMAX(v, k, n)	**Tempos de execução**
1. $pmax \leftarrow k$;	c_1: atribuir valor de variável simples a outra
2. $i \leftarrow k+1$;	c_2: avaliar expressão e atribuir valor a variável
3. Enquanto $i \leq n$ fazer	c_3: testar condição e transferir controlo
4. Se $v[i] > v[pmax]$ então	c_4: aceder, testar e transferir controlo
5. $pmax \leftarrow i$;	c_1: atribuir valor de variável simples a outra
6. $i \leftarrow i+1$;	c_6: incrementar valor de variável
7. retorna $pmax$;	c_7: retornar da função com valor de variável

- Executa as instruções 1, 2 e 7 apenas uma vez. Executa teste na linha 3 para $i = k+1, \ldots, n, n+1$, ou seja, $(n+1)-(k+1)+1 = n-k+1$ vezes.
- Executa a instrução 4 para $i = k+1, \ldots, n$, ou seja $n-(k+1)+1 = n-k$ vezes. Análogo para a instrução 6.
- Quantas vezes executa a instrução 5? Depende da instância.
  - No **melhor caso**, executa zero vezes. Ocorre se $v[k]$ é estritamente maior do que os restantes elementos.
  - No **pior caso**, tem-se $v[k] < v[k+1] < \ldots < v[n-1] < v[n]$ e executa $n-k$ vezes.

Exemplo: complexidade temporal de POSMAX

POSMAX(v, k, n)	Tempos de execução
1. $pmax \leftarrow k$;	c_1: atribuir valor de variável simples a outra
2. $i \leftarrow k + 1$;	c_2: avaliar expressão e atribuir valor a variável
3. Enquanto $i \leq n$ fazer	c_3: testar condição e transferir controlo
4. Se $v[i] > v[pmax]$ então	c_4: aceder, testar e transferir controlo
5. $pmax \leftarrow i$;	c_1: atribuir valor de variável simples a outra
6. $i \leftarrow i + 1$;	c_6: incrementar valor de variável
7. retorna $pmax$;	c_7: retornar da função com valor de variável

- Executa as instruções 1, 2 e 7 apenas uma vez. Executa teste na linha 3 para $i = k+1, \ldots, n, n+1$, ou seja, $(n+1) - (k+1) + 1 = n - k + 1$ vezes.
- Executa a instrução 4 para $i = k+1, \ldots, n$, ou seja $n - (k+1) + 1 = n - k$ vezes. Análogo para a instrução 6.
- Quantas vezes executa a instrução 5? Depende da instância.
  - No **melhor caso**, executa zero vezes. Ocorre se $v[k]$ é estritamente maior do que os restantes elementos.
  - No **pior caso**, tem-se $v[k] < v[k+1] < \ldots < v[n-1] < v[n]$ e executa $n - k$ vezes.

Exemplo: complexidade temporal de PosMax

PosMax(v, k, n)

1. $pmax \leftarrow k$;
2. $i \leftarrow k + 1$;
3. Enquanto $i \leq n$ fazer
4. Se $v[i] > v[pmax]$ então
5. $pmax \leftarrow i$;
6. $i \leftarrow i + 1$;
7. retorna $pmax$;

Tempos de execução

c_1: atribuir valor de variável simples a outra
c_2: avaliar expressão e atribuir valor a variável
c_3: testar condição e transferir controlo
c_4: aceder, testar e transferir controlo
c_1: atribuir valor de variável simples a outra
c_6: incrementar valor de variável
c_7: retornar da função com valor de variável

- No melhor caso, $T(n, k) = c_1 + c_2 + c_7 + (n - k + 1)c_3 + (n - k)(c_4 + c_6)$.
- No pior caso, $T(n, k) = c_1 + c_2 + c_7 + (n - k + 1)c_3 + (n - k)(c_4 + c_6 + c_1)$.

Para todas as instâncias, tem-se

$$\underbrace{c_1 + c_2 + c_7 + c_3 + (c_3 + c_4 + c_6)(n - k)}_{\text{melhor caso}} \leq T(n, k) \leq \underbrace{c_1 + c_2 + c_7 + c_3 + (c_3 + c_4 + c_6 + c_1)(n - k)}_{\text{pior caso}}$$

Exemplo: complexidade temporal de Posmax

```
Posmax(v, k, n)
    pmax ← k;
    i ← k + 1;
    Enquanto i ≤ n fazer
        Se v[i] > v[pmax] então
            pmax ← i;
        i ← i + 1;
    retorna pmax;
```

Tempos de execução

c_1: atribuir valor de variável simples a outra
c_2: avaliar expressão e atribuir valor a variável
c_3: testar condição e transferir controlo
c_4: aceder, testar e transferir controlo
c_1: atribuir valor de variável simples a outra
c_6: incrementar valor de variável
c_7: retornar da função com valor de variável

- No melhor caso, $T(n,k) = c_1 + c_2 + c_7 + (n-k+1)c_3 + (n-k)(c_4 + c_6)$.
- No pior caso, $T(n,k) = c_1 + c_2 + c_7 + (n-k+1)c_3 + (n-k)(c_4 + c_6 + c_1)$.

Para todas as instâncias, tem-se

$$\underbrace{c_1 + c_2 + c_7 + c_3 + (c_3 + c_4 + c_6)(n-k)}_{\text{melhor caso}} \leq T(n,k) \leq \underbrace{c_1 + c_2 + c_7 + c_3 + (c_3 + c_4 + c_6 + c_1)(n-k)}_{\text{pior caso}}$$

Exemplo: complexidade temporal de POSMAX

POSMAX(v, k, n)

```
pmax ← k;
i ← k + 1;
Enquanto i ≤ n fazer
    Se v[i] > v[pmax] então
        pmax ← i;
    i ← i + 1;
retorna pmax;
```

Tempos de execução

c_1: atribuir valor de variável simples a outra
c_2: avaliar expressão e atribuir valor a variável
c_3: testar condição e transferir controlo
c_4: aceder, testar e transferir controlo
c_1: atribuir valor de variável simples a outra
c_6: incrementar valor de variável
c_7: retornar da função com valor de variável

- No melhor caso, $T(n,k) = c_1 + c_2 + c_7 + (n-k+1)c_3 + (n-k)(c_4 + c_6)$.
- No pior caso, $T(n,k) = c_1 + c_2 + c_7 + (n-k+1)c_3 + (n-k)(c_4 + c_6 + c_1)$.

Para todas as instâncias, tem-se

$$\underbrace{c_1 + c_2 + c_7 + c_3 + (c_3 + c_4 + c_6)(n-k)}_{\text{melhor caso}} \leq T(n,k) \leq \underbrace{c_1 + c_2 + c_7 + c_3 + (c_3 + c_4 + c_6 + c_1)(n-k)}_{\text{pior caso}}$$

Exemplo: complexidade temporal de Posmax

Posmax(v, k, n)	**Tempos de execução**
$pmax \leftarrow k$;	c_1: atribuir valor de variável simples a outra
$i \leftarrow k+1$;	c_2: avaliar expressão e atribuir valor a variável
Enquanto $i \leq n$ fazer	c_3: testar condição e transferir controlo
Se $v[i] > v[pmax]$ então	c_4: aceder, testar e transferir controlo
$pmax \leftarrow i$;	c_1: atribuir valor de variável simples a outra
$i \leftarrow i+1$;	c_6: incrementar valor de variável
retorna $pmax$;	c_7: retornar da função com valor de variável

$$\underbrace{c_1 + c_2 + c_7 + c_3 + (c_3 + c_4 + c_6)(n-k)}_{\text{melhor caso}} \leq T(n,k) \leq \underbrace{c_1 + c_2 + c_7 + c_3 + (c_3 + c_4 + c_6 + c_1)(n-k)}_{\text{pior caso}}$$

Tomando $a = \min_t c_t$ e $b = \max_t c_t$, concluimos que

$$4a + 3a(n-k) \leq T(n,k) \leq 4b + 4b(n-k).$$

Existem constantes $c', c'' \in \mathbb{R}^+$ *tais que* $c'(n-k+1) \leq T(n,k) \leq c''(n-k+1)$. Por exemplo, $c' = 3a$ e $c'' = 4b$. Conclui-se que $T(n,k) \in \Theta(n-k+1)$, ou seja, $T(n,k)$ é **linear** no número de elementos do segmento de v a analisar.

Definição das ordens de grandeza O, Θ e Ω

$O(g(n))$, $\Omega(g(n))$, e $\Theta(g(n))$ designam **conjuntos de funções**: todas as funções $f : \mathbb{N} \to \mathbb{R}_0^+$ que se relacionam com uma dada função $g : \mathbb{N} \to \mathbb{R}_0^+$ da forma indicada na definição correspondente.

$$O(g(n)) = \{f(n) \mid \text{existem } c > 0 \text{ e } n_0 > 0 \text{ tais que } f(n) \leq cg(n), \text{ para todo } n \geq n_0\}$$

$$\Omega(g(n)) = \{f(n) \mid \text{existem } c > 0 \text{ e } n_0 > 0 \text{ tais que } f(n) \geq cg(n), \text{ para todo } n \geq n_0\}$$

$$\Theta(g(n)) = \{f(n) \mid \text{existem } c_1 > 0,\ c_2 > 0 \text{ e } n_0 > 0 \text{ tais que } c_1 g(n) \leq f(n) \leq c_2 g(n), \text{ para todo } n \geq n_0\}$$

- $f(n) \in O(g(n))$ sse $f(n)$ é majorada por $cg(n)$ para alguma constante $c \in \mathbb{R}^+$, a partir de uma certa ordem definida por n_0, com $n_0 \in \mathbb{Z}^+$ fixo.
- $f(n) \in \Omega(g(n))$ sse $f(n)$ é minorada por $cg(n)$ para alguma constante $c \in \mathbb{R}^+$, a partir de uma certa ordem definida por n_0, com $n_0 \in \mathbb{Z}^+$ fixo.
- $f(n) \in \Theta(g(n))$ sse $f(n)$ é majorada por $c_2 g(n)$ e minorada por $c_1 g(n)$ para algum $c_1 \in \mathbb{R}^+$ e algum $c_2 \in \mathbb{R}^+ > 0$, a partir de uma certa ordem definida por n_0, com $n_0 \in \mathbb{Z}^+$ fixo.

Exemplo: complexidade temporal de Posmax

Posmax(v, k, n)	Tempos de execução
$pmax \leftarrow k$;	c_1: atribuir valor de variável simples a outra
$i \leftarrow k+1$;	c_2: avaliar expressão e atribuir valor a variável
Enquanto $i \leq n$ fazer	c_3: testar condição e transferir controlo
Se $v[i] > v[pmax]$ então	c_4: aceder, testar e transferir controlo
$pmax \leftarrow i$;	c_1: atribuir valor de variável simples a outra
$i \leftarrow i+1$;	c_6: incrementar valor de variável
retorna $pmax$;	c_7: retornar da função com valor de variável

Sendo $N = n - k + 1$ o número de elementos do vetor no segmento a analisar vimos que:

- existia $c' \in \mathbb{R}^+$ tal que $T(N) \geq c'N$, para todo $N \geq 1$, ou seja, que $T(N) \in \Omega(N)$.
- existia $c'' \in \mathbb{R}^+$ tal que $T(N) \leq c''N$, para todo $N \geq 1$, ou seja, que $T(N) \in O(N)$.
- existiam $c', c'' \in \mathbb{R}^+$ tais que $c'N \leq T(N) \leq c''T(N)$, para todo $N \geq 1$, ou seja, que $T(N) \in \Theta(N)$.

Aqui, $c' = 3\min_t c_t$, $c'' = 4\max_t c_t$, e $n_0 = 1$.

Exemplo: complexidade temporal de POSMAX

```
POSMAX(v, k, n)
    pmax ← k;
    i ← k + 1;
    Enquanto i ≤ n fazer
        Se v[i] > v[pmax] então
            pmax ← i;
        i ← i + 1;
    retorna pmax;
```

Tempos de execução

c_1: atribuir valor de variável simples a outra
c_2: avaliar expressão e atribuir valor a variável
c_3: testar condição e transferir controlo
c_4: aceder, testar e transferir controlo
c_1: atribuir valor de variável simples a outra
c_6: incrementar valor de variável
c_7: retornar da função com valor de variável

Sendo $N = n - k + 1$ o número de elementos do vetor no segmento a analisar vimos que:

- existia $c' \in \mathbb{R}^+$ tal que $T(N) \geq c'N$, para todo $N \geq 1$, ou seja, que $T(N) \in \Omega(N)$.
- existia $c'' \in \mathbb{R}^+$ tal que $T(N) \leq c''N$, para todo $N \geq 1$, ou seja, que $T(N) \in O(N)$.
- existiam $c', c'' \in \mathbb{R}^+$ tais que $c'N \leq T(N) \leq c''T(N)$, para todo $N \geq 1$, ou seja, que $T(N) \in \Theta(N)$.

Aqui, $c' = 3 \min_t c_t$, $c'' = 4 \max_t c_t$, e $n_0 = 1$.

Exemplo: complexidade temporal de POSMAX

```
POSMAX(v, k, n)
    pmax ← k;
    i ← k + 1;
    Enquanto i ≤ n fazer
        Se v[i] > v[pmax] então
            pmax ← i;
        i ← i + 1;
    retorna pmax;
```

Tempos de execução

c_1: atribuir valor de variável simples a outra
c_2: avaliar expressão e atribuir valor a variável
c_3: testar condição e transferir controlo
c_4: aceder, testar e transferir controlo
c_1: atribuir valor de variável simples a outra
c_6: incrementar valor de variável
c_7: retornar da função com valor de variável

Sendo $N = n - k + 1$ o número de elementos do vetor no segmento a analisar vimos que:

- existia $c' \in \mathbb{R}^+$ tal que $T(N) \geq c'N$, para todo $N \geq 1$, ou seja, que $T(N) \in \Omega(N)$.
- existia $c'' \in \mathbb{R}^+$ tal que $T(N) \leq c''N$, para todo $N \geq 1$, ou seja, que $T(N) \in O(N)$.
- existiam $c', c'' \in \mathbb{R}^+$ tais que $c'N \leq T(N) \leq c''T(N)$, para todo $N \geq 1$, ou seja, que $T(N) \in \Theta(N)$.

Aqui, $c' = 3 \min_t c_t$, $c'' = 4 \max_t c_t$, e $n_0 = 1$.

Exemplos: Ordens de grandeza

- $f(n) = 7n + 15000$, com $n \in \mathbb{N}$
  - $f(n) \in \Omega(n)$ porque $f(n) \geq 7n$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$ e $c = 7$.
  - $f(n) \in O(n)$ porque $f(n) \leq 15007n$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$ e $c = 15007$.
  - $f(n) \in \Theta(n)$ porque $7n \leq f(n) \leq 15007n$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$, $c_1 = 7$ e $c_2 = 15007$.

Exemplos: Ordens de grandeza

- $f(n) = 7n + 15000$, com $n \in \mathbb{N}$
  - $f(n) \in \Omega(n)$ porque $f(n) \geq 7n$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$ e $c = 7$.
  - $f(n) \in O(n)$ porque $f(n) \leq 15007n$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$ e $c = 15007$.
  - $f(n) \in \Theta(n)$ porque $7n \leq f(n) \leq 15007n$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$, $c_1 = 7$ e $c_2 = 15007$.

Exemplos: Ordens de grandeza

- $f(n) = 7n + 15000$, com $n \in \mathbb{N}$
  - $f(n) \in \Omega(n)$ porque $f(n) \geq 7n$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$ e $c = 7$.
  - $f(n) \in O(n)$ porque $f(n) \leq 15007n$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$ e $c = 15007$.
  - $f(n) \in \Theta(n)$ porque $7n \leq f(n) \leq 15007n$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$, $c_1 = 7$ e $c_2 = 15007$.

Exemplos: Ordens de grandeza

- $f(n) = \frac{1}{2}n^2 + 100n + 7$, com $n \in \mathbb{N}$
  - $f(n) \in \Omega(n^2)$ porque $f(n) \geq \frac{1}{2}n^2$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$ e $c = \frac{1}{2}$, por exemplo.
  - $f(n) \in O(n^2)$ pois $f(n) \leq n^2 + 100n^2 + 7n^2 = 108n^2$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$ e $c = 108$, por exemplo.
  - $f(n) \in \Theta(n^2)$ porque $\frac{1}{2}n^2 \leq f(n) \leq 108n^2$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$, $c_1 = \frac{1}{2}$ e $c_2 = 108$, por exemplo.

Exemplos: Ordens de grandeza

- $f(n) = \frac{1}{2}n^2 + 100n + 7$, com $n \in \mathbb{N}$
  - $f(n) \in \Omega(n^2)$ porque $f(n) \geq \frac{1}{2}n^2$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$ e $c = \frac{1}{2}$, por exemplo.
  - $f(n) \in O(n^2)$ pois $f(n) \leq n^2 + 100n^2 + 7n^2 = 108n^2$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$ e $c = 108$, por exemplo.
  - $f(n) \in \Theta(n^2)$ porque $\frac{1}{2}n^2 \leq f(n) \leq 108n^2$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$, $c_1 = \frac{1}{2}$ e $c_2 = 108$, por exemplo.

Exemplos: Ordens de grandeza

- $f(n) = \frac{1}{2}n^2 + 100n + 7$, com $n \in \mathbb{N}$
  - $f(n) \in \Omega(n^2)$ porque $f(n) \geq \frac{1}{2}n^2$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$ e $c = \frac{1}{2}$, por exemplo.
  - $f(n) \in O(n^2)$ pois $f(n) \leq n^2 + 100n^2 + 7n^2 = 108n^2$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$ e $c = 108$, por exemplo.
  - $f(n) \in \Theta(n^2)$ porque $\frac{1}{2}n^2 \leq f(n) \leq 108n^2$, para todo $n \geq 1$.
    Para a definição, $n_0 = 1$, $c_1 = \frac{1}{2}$ e $c_2 = 108$, por exemplo.

Exemplos: Ordens de grandeza

- Sejam $f(n) = \frac{1}{5} n \log_2(n) + 10n + 3$, $g(n) = 200 n \log_2(n) + 50n$, e $h(n) = n \log_2 n$, com $n \geq 1$

  - $f(n) \in \Omega(g(n))$ pois $f(n) \geq \frac{1}{1000} g(n)$, para $n \geq 1$. Para a definição, $n_0 = 1$ e $c = \frac{1}{1000}$, por exemplo.

  - $f(n) \in O(h(n))$ porque $f(n) \leq 14 h(n)$, para todo $n \geq 2$. Para a definição, $n_0 = 2$ e $c = 14$, por exemplo.

  - $h(n) \in \Theta(f(n))$ porque $\frac{1}{100} f(n) \leq h(n) \leq 5 f(n)$, para todo $n \geq 2$. Para a definição, $n_0 = 2$, $c_1 = \frac{1}{100}$ e $c_2 = 5$, por exemplo.

    Notar que se $n \geq 2$, tem-se $\frac{1}{100} f(n) = \frac{1}{500} n \log_2(n) + \frac{1}{10} n + \frac{3}{100} \leq \frac{5}{10} n \log_2 n \leq n \log_2 n$.

  - $h(n) \in \Theta(g(n))$ porque $\frac{1}{400} g(n) \leq h(n) \leq g(n)$, para todo $n \geq 2$. Para a definição, $n_0 = 2$, $c_1 = \frac{1}{400}$ e $c_2 = 1$, por exemplo.

Exemplos: Ordens de grandeza

- Sejam $f(n) = \frac{1}{5} n \log_2(n) + 10n + 3$, $g(n) = 200n \log_2(n) + 50n$, e $h(n) = n \log_2 n$, com $n \geq 1$

  - $f(n) \in \Omega(g(n))$ pois $f(n) \geq \frac{1}{1000} g(n)$, para $n \geq 1$. Para a definição, $n_0 = 1$ e $c = \frac{1}{1000}$, por exemplo.

  - $f(n) \in O(h(n))$ porque $f(n) \leq 14h(n)$, para todo $n \geq 2$. Para a definição, $n_0 = 2$ e $c = 14$, por exemplo.

  - $h(n) \in \Theta(f(n))$ porque $\frac{1}{100} f(n) \leq h(n) \leq 5f(n)$, para todo $n \geq 2$. Para a definição, $n_0 = 2$, $c_1 = \frac{1}{100}$ e $c_2 = 5$, por exemplo.

    Notar que se $n \geq 2$, tem-se $\frac{1}{100} f(n) = \frac{1}{500} n \log_2(n) + \frac{1}{10} n + \frac{3}{100} \leq \frac{5}{10} n \log_2 n \leq n \log_2 n$.

  - $h(n) \in \Theta(g(n))$ porque $\frac{1}{400} g(n) \leq h(n) \leq g(n)$, para todo $n \geq 2$. Para a definição, $n_0 = 2$, $c_1 = \frac{1}{400}$ e $c_2 = 1$, por exemplo.

Exemplos: Ordens de grandeza

- Sejam $f(n) = \frac{1}{5} n \log_2(n) + 10n + 3$, $g(n) = 200n \log_2(n) + 50n$, e $h(n) = n \log_2 n$, com $n \geq 1$

  - $f(n) \in \Omega(g(n))$ pois $f(n) \geq \frac{1}{1000} g(n)$, para $n \geq 1$.
    Para a definição, $n_0 = 1$ e $c = \frac{1}{1000}$, por exemplo.

  - $f(n) \in O(h(n))$ porque $f(n) \leq 14h(n)$, para todo $n \geq 2$.
    Para a definição, $n_0 = 2$ e $c = 14$, por exemplo.

  - $h(n) \in \Theta(f(n))$ porque $\frac{1}{100} f(n) \leq h(n) \leq 5f(n)$, para todo $n \geq 2$.
    Para a definição, $n_0 = 2$, $c_1 = \frac{1}{100}$ e $c_2 = 5$, por exemplo.

    Notar que se $n \geq 2$, tem-se $\frac{1}{100} f(n) = \frac{1}{500} n \log_2(n) + \frac{1}{10} n + \frac{3}{100} \leq \frac{5}{10} n \log_2 n \leq n \log_2 n$.

  - $h(n) \in \Theta(g(n))$ porque $\frac{1}{400} g(n) \leq h(n) \leq g(n)$, para todo $n \geq 2$.
    Para a definição, $n_0 = 2$, $c_1 = \frac{1}{400}$ e $c_2 = 1$, por exemplo.

Exemplos: Ordens de grandeza

- Sejam $f(n) = \frac{1}{5} n \log_2(n) + 10n + 3$, $g(n) = 200n \log_2(n) + 50n$, e $h(n) = n \log_2 n$, com $n \geq 1$
  - $f(n) \in \Omega(g(n))$ pois $f(n) \geq \frac{1}{1000} g(n)$, para $n \geq 1$. Para a definição, $n_0 = 1$ e $c = \frac{1}{1000}$, por exemplo.
  - $f(n) \in O(h(n))$ porque $f(n) \leq 14h(n)$, para todo $n \geq 2$. Para a definição, $n_0 = 2$ e $c = 14$, por exemplo.
  - $h(n) \in \Theta(f(n))$ porque $\frac{1}{100} f(n) \leq h(n) \leq 5f(n)$, para todo $n \geq 2$. Para a definição, $n_0 = 2$, $c_1 = \frac{1}{100}$ e $c_2 = 5$, por exemplo.

    Notar que se $n \geq 2$, tem-se $\frac{1}{100} f(n) = \frac{1}{500} n \log_2(n) + \frac{1}{10} n + \frac{3}{100} \leq \frac{5}{10} n \log_2 n \leq n \log_2 n$.
  - $h(n) \in \Theta(g(n))$ porque $\frac{1}{400} g(n) \leq h(n) \leq g(n)$, para todo $n \geq 2$. Para a definição, $n_0 = 2$, $c_1 = \frac{1}{400}$ e $c_2 = 1$, por exemplo.

Revisão: Ordenação por seleção (selection sort)

Problema

Ordenar as primeiras n posições do vetor v por **ordem decrescente**, supondo que são indexadas a partir de 1. Aplicar *ordenação por seleção*.

SelectionSort(v, n)
  Para cada $k \leftarrow 1$ até $n-1$ fazer
    $j \leftarrow$ PosMax(v, k, n);
    Se $j \neq k$ então
      $aux \leftarrow v[k]$;
      $v[k] \leftarrow v[j]$;
      $v[j] \leftarrow aux$;

Invariante de ciclo:
Para $i \geq 1$, imediatamente após a i-ésima iteração do ciclo "Para" (e incremento de k), o valor da variável k é $i+1$, o vetor v contém exatamente os mesmos elementos que tinha antes da entrada no ciclo, embora possam estar em posições distintas, $v[1] \geq v[2] \geq \ldots \geq v[i]$ e, se $i < n$ então $v[i] \geq \max(v[i+1], \ldots, v[n])$.

Exercício: prove esta propriedade. Assuma que PosMax retorna o índice da primeira ocorrência do máximo de $v[k], \ldots, v[n]$.

Revisão: Ordenação por seleção (selection sort)

Problema

Ordenar as primeiras n posições do vetor v por **ordem decrescente**, supondo que são indexadas a partir de 1. Aplicar *ordenação por seleção*.

$\textsc{SelectionSort}(v, n)$
Para cada $k \leftarrow 1$ até $n-1$ fazer
    $j \leftarrow \textsc{PosMax}(v, k, n)$;
    Se $j \neq k$ então
        $aux \leftarrow v[k]$;
        $v[k] \leftarrow v[j]$;
        $v[j] \leftarrow aux$;

Invariante de ciclo:

Para $i \geq 1$, imediatamente após a i-ésima iteração do ciclo "Para" (e incremento de k), o valor da variável k é $i+1$, o vetor v contém exatamente os mesmos elementos que tinha antes da entrada no ciclo, embora possam estar em posições distintas, $v[1] \geq v[2] \geq \ldots \geq v[i]$ e, se $i < n$ então $v[i] \geq \max(v[i+1], \ldots, v[n])$.

Exercício: prove esta propriedade. Assuma que $\textsc{PosMax}$ retorna o índice da primeira ocorrência do máximo de $v[k], \ldots, v[n]$.

Revisão: Ordenação por seleção (selection sort)

Problema

Ordenar as primeiras n posições do vetor v por **ordem decrescente**, supondo que são indexadas a partir de 1. Aplicar *ordenação por seleção*.

$\textsc{SelectionSort}(v, n)$
Para cada $k \leftarrow 1$ até $n-1$ fazer
  $j \leftarrow \textsc{PosMax}(v, k, n)$;
  Se $j \neq k$ então
    $aux \leftarrow v[k]$;
    $v[k] \leftarrow v[j]$;
    $v[j] \leftarrow aux$;

Invariante de ciclo:

Para $i \geq 1$, imediatamente após a i-ésima iteração do ciclo "Para" (e incremento de k), o valor da variável k é $i+1$, o vetor v contém exatamente os mesmos elementos que tinha antes da entrada no ciclo, embora possam estar em posições distintas, $v[1] \geq v[2] \geq \ldots \geq v[i]$ e, se $i < n$ então $v[i] \geq \max(v[i+1], \ldots, v[n])$.

Exercício: prove esta propriedade. Assuma que $\textsc{PosMax}$ retorna o índice da primeira ocorrência do máximo de $v[k], \ldots, v[n]$.

Revisão: Ordenação por seleção (selection sort)

Problema

Ordenar as primeiras n posições do vetor v por **ordem decrescente**, supondo que são indexadas a partir de 1. Aplicar *ordenação por seleção*.

$\textsc{SelectionSort}(v, n)$
Para cada $k \leftarrow 1$ até $n-1$ fazer
  $j \leftarrow \textsc{PosMax}(v, k, n)$;
  Se $j \neq k$ então
    $aux \leftarrow v[k]$;
    $v[k] \leftarrow v[j]$;
    $v[j] \leftarrow aux$;

Invariante de ciclo:

Para $i \geq 1$, imediatamente após a i-ésima iteração do ciclo "Para" (e incremento de k), o valor da variável k é $i+1$, o vetor v contém exatamente os mesmos elementos que tinha antes da entrada no ciclo, embora possam estar em posições distintas, $v[1] \geq v[2] \geq \ldots \geq v[i]$ e, se $i < n$ então $v[i] \geq \max(v[i+1], \ldots, v[n])$.

Exercício: prove esta propriedade. Assuma que $\textsc{PosMax}$ retorna o índice da primeira ocorrência do máximo de $v[k], \ldots, v[n]$.

Revisão: Ordenação por seleção (selection sort)

Resolução do exercício (prova do invariante de ciclo por indução):
(sabendo já que $\text{POSMAX}(v,k,n)$ retorna o índice de $\max(v[k],\ldots,v[n])$).

$\text{SELECTIONSORT}(v,n)$
- Para cada $k \leftarrow 1$ até $n-1$ fazer
  - $j \leftarrow \text{POSMAX}(v,k,n)$;
  - Se $j \neq k$ então
    - $aux \leftarrow v[k]$;
    - $v[k] \leftarrow v[j]$;
    - $v[j] \leftarrow aux$;

Para $i \geq 1$, após a i-ésima iteração do ciclo "Para" (e incremento de k), o valor de k é $i+1$, o vetor v contém os elementos que tinha antes da entrada no ciclo, possivelmente noutras posições, $v[1] \geq v[2] \geq \ldots \geq v[i]$ e, se $i < n$ então $v[i] \geq \max(v[i+1],\ldots,v[n])$.

Caso de base:

Na iteração 1, $k=1$. Após a execução de $\text{POSMAX}(v,1,n)$, a variável j tem o índice da primeira ocorrência de $\max(v[1],\ldots,v[n])$. A seguir, se $j \neq 1$, troca $v[1]$ com $v[j]$. Logo, $v[1] \geq \max(v[2],\ldots,v[n])$ após a iteração 1 e k será 2.

Hereditariedade:

Vamos provar **se** a condição é verdadeira no fim da iteração $i-1$ **então** é verdadeira no fim da iteração i, para todo $2 \leq i \leq n-1$.

Revisão: Ordenação por seleção (selection sort)

Resolução do exercício (prova do invariante de ciclo por indução):

(sabendo já que $\mathrm{POSMAX}(v, k, n)$ retorna o índice de $\max(v[k], \ldots, v[n])$).

```
SelectionSort(v, n)
  Para cada k ← 1 até n − 1 fazer
    j ← PosMax(v, k, n);
    Se j ≠ k então
      aux ← v[k];
      v[k] ← v[j];
      v[j] ← aux;
```

Para $i \geq 1$, após a i-ésima iteração do ciclo "Para" (e incremento de k), o valor de k é $i+1$, o vetor v contém os elementos que tinha antes da entrada no ciclo, possivelmente noutras posições, $v[1] \geq v[2] \geq \ldots \geq v[i]$ e, se $i < n$ então $v[i] \geq \max(v[i+1], \ldots, v[n])$.

Caso de base:

Na iteração 1, $k = 1$. Após a execução de $\mathrm{POSMAX}(v, 1, n)$, a variável j tem o índice da primeira ocorrência de $\max(v[1], \ldots, v[n])$. A seguir, se $j \neq 1$, troca $v[1]$ com $v[j]$. Logo, $v[1] \geq \max(v[2], \ldots, v[n])$ após a iteração 1 e k será 2.

Hereditariedade:

Vamos provar $\boxed{\textbf{se}}$ a condição é verdadeira no fim da iteração $i-1$ $\boxed{\textbf{então}}$ é verdadeira no fim da iteração i, para todo $2 \leq i \leq n-1$.

Revisão: Ordenação por seleção (selection sort)

Resolução do exercício (prova do invariante de ciclo por indução):

(sabendo já que $\textsc{PosMax}(v, k, n)$ retorna o índice de $\max(v[k], \ldots, v[n])$).

```
SelectionSort(v, n)
  Para cada k ← 1 até n − 1 fazer
    j ← PosMax(v, k, n);
    Se j ≠ k então
      aux ← v[k];
      v[k] ← v[j];
      v[j] ← aux;
```

Para $i \geq 1$, após a i-ésima iteração do ciclo "Para" (e incremento de k), o valor de k é $i+1$, o vetor v contém os elementos que tinha antes da entrada no ciclo, possivelmente noutras posições, $v[1] \geq v[2] \geq \ldots \geq v[i]$ e, se $i < n$ então $v[i] \geq \max(v[i+1], \ldots, v[n])$.

Caso de base:

Na iteração 1, $k = 1$. Após a execução de $\textsc{PosMax}(v, 1, n)$, a variável j tem o índice da primeira ocorrência de $\max(v[1], \ldots, v[n])$. A seguir, se $j \neq 1$, troca $v[1]$ com $v[j]$. Logo, $v[1] \geq \max(v[2], \ldots, v[n])$ após a iteração 1 e k será 2.

Hereditariedade:

Vamos provar **se** a condição é verdadeira no fim da iteração $i-1$ **então** é verdadeira no fim da iteração i, para todo $2 \leq i \leq n-1$.

Revisão: Ordenação por seleção (selection sort)

```
SelectionSort(v, n)
  Para cada k ← 1 até n − 1 fazer
    j ← PosMax(v, k, n);
    Se j ≠ k então
      aux ← v[k];
      v[k] ← v[j];
      v[j] ← aux;
```

Para $i \geq 1$, após a i-ésima iteração do ciclo "Para", $k = i + 1$, v tem os elementos que tinha antes do ciclo, talvez noutras posições, $v[1] \geq v[2] \geq \ldots \geq v[i]$ e, se $i < n$ então $v[i] \geq \max(v[i+1], \ldots, v[n])$.

Hereditariedade:

Se a condição é verdadeira no fim da iteração $i - 1$ **então** é verdadeira no fim da iteração i, para todo $2 \leq i \leq n - 1$, porque:.

- Por hipótese, no fim da iteração $i - 1$ (ínicio da iteração i), a variável k tem $k = (i - 1) + 1 = i$. Sendo $i \leq n - 1$, executará o bloco do ciclo novamente. Atribui a j o índice da posição de $\max(v[i], \ldots, v[n])$ e se $j \neq k = i$, então troca $v[i]$ com $v[j]$. Logo, após esse bloco, $v[i] \geq \max(v[i+1], \ldots, v[n])$, e a troca preserva os elementos de v (ainda que possa alterar posições).
- Por hipótese, $v[1] \geq v[2] \geq \ldots \geq v[i-1]$ e $v[i-1] \geq \max(v[i], \ldots, v[n])$. Logo, para o novo $v[i]$, também se tem $v[i-1] \geq v[i]$.
- Assim, após a iteração i, tem-se $v[1] \geq v[2] \geq \ldots \geq v[i-1] \geq v[i]$ e $v[i] \geq \max(v[i+1], \ldots, v[n])$, e k tem valor $i + 1$.

Revisão: Ordenação por seleção (selection sort)

```
SelectionSort(v, n)
  Para cada k ← 1 até n − 1 fazer
    j ← PosMax(v, k, n);
    Se j ≠ k então
      aux ← v[k];
      v[k] ← v[j];
      v[j] ← aux;
```

Para $i \geq 1$, após a i-ésima iteração do ciclo "Para", $k = i + 1$, v tem os elementos que tinha antes do ciclo, talvez noutras posições, $v[1] \geq v[2] \geq \ldots \geq v[i]$ e, se $i < n$ então $v[i] \geq \max(v[i+1], \ldots, v[n])$.

Hereditariedade:

Se a condição é verdadeira no fim da iteração $i - 1$ **então** é verdadeira no fim da iteração i, para todo $2 \leq i \leq n - 1$, porque:.

- Por hipótese, no fim da iteração $i - 1$ (ínicio da iteração i), a variável k tem $k = (i - 1) + 1 = i$. Sendo $i \leq n - 1$, executará o bloco do ciclo novamente. Atribui a j o índice da posição de $\max(v[i], \ldots, v[n])$ e se $j \neq k = i$, então troca $v[i]$ com $v[j]$. Logo, após esse bloco, $v[i] \geq \max(v[i+1], \ldots, v[n])$, e a troca preserva os elementos de v (ainda que possa alterar posições).
- Por hipótese, $v[1] \geq v[2] \geq \ldots \geq v[i-1]$ e $v[i-1] \geq \max(v[i], \ldots, v[n])$. Logo, para o novo $v[i]$, também se tem $v[i-1] \geq v[i]$.
- Assim, após a iteração i, tem-se $v[1] \geq v[2] \geq \ldots \geq v[i-1] \geq v[i]$ e $v[i] \geq \max(v[i+1], \ldots, v[n])$, e k tem valor $i + 1$.

Revisão: Ordenação por seleção (selection sort)

```
SelectionSort(v, n)
  Para cada k ← 1 até n − 1 fazer
    j ← PosMax(v, k, n);
    Se j ≠ k então
      aux ← v[k];
      v[k] ← v[j];
      v[j] ← aux;
```

Para $i \geq 1$, após a i-ésima iteração do ciclo "Para", $k = i + 1$, v tem os elementos que tinha antes do ciclo, talvez noutras posições, $v[1] \geq v[2] \geq \ldots \geq v[i]$ e, se $i < n$ então $v[i] \geq \max(v[i+1], \ldots, v[n])$.

Hereditariedade:

Se a condição é verdadeira no fim da iteração $i - 1$ **então** é verdadeira no fim da iteração i, para todo $2 \leq i \leq n - 1$, porque:.

- Por hipótese, no fim da iteração $i - 1$ (ínicio da iteração i), a variável k tem $k = (i-1) + 1 = i$. Sendo $i \leq n - 1$, executará o bloco do ciclo novamente. Atribui a j o índice da posição de $\max(v[i], \ldots, v[n])$ e se $j \neq k = i$, então troca $v[i]$ com $v[j]$. Logo, após esse bloco, $v[i] \geq \max(v[i+1], \ldots, v[n])$, e a troca preserva os elementos de v (ainda que possa alterar posições).
- Por hipótese, $v[1] \geq v[2] \geq \ldots \geq v[i-1]$ e $v[i-1] \geq \max(v[i], \ldots, v[n])$. Logo, para o novo $v[i]$, também se tem $v[i-1] \geq v[i]$.
- Assim, após a iteração i, tem-se $v[1] \geq v[2] \geq \ldots \geq v[i-1] \geq v[i]$ e $v[i] \geq \max(v[i+1], \ldots, v[n])$, e k tem valor $i + 1$.

Revisão: Ordenação por seleção (selection sort)

```
SelectionSort(v, n)
  Para cada k ← 1 até n − 1 fazer
    j ← PosMax(v, k, n);
    Se j ≠ k então
      aux ← v[k];
      v[k] ← v[j];
      v[j] ← aux;
```

Para $i \geq 1$, após a i-ésima iteração do ciclo "Para", $k = i + 1$, v tem os elementos que tinha antes do ciclo, talvez noutras posições, $v[1] \geq v[2] \geq \ldots \geq v[i]$ e, se $i < n$ então $v[i] \geq \max(v[i+1], \ldots, v[n])$.

Hereditariedade:

Se a condição é verdadeira no fim da iteração $i - 1$ **então** é verdadeira no fim da iteração i, para todo $2 \leq i \leq n - 1$, porque:.

- Por hipótese, no fim da iteração $i - 1$ (ínicio da iteração i), a variável k tem $k = (i - 1) + 1 = i$. Sendo $i \leq n - 1$, executará o bloco do ciclo novamente. Atribui a j o índice da posição de $\max(v[i], \ldots, v[n])$ e se $j \neq k = i$, então troca $v[i]$ com $v[j]$. Logo, após esse bloco, $v[i] \geq \max(v[i+1], \ldots, v[n])$, e a troca preserva os elementos de v (ainda que possa alterar posições).
- Por hipótese, $v[1] \geq v[2] \geq \ldots \geq v[i-1]$ e $v[i-1] \geq \max(v[i], \ldots, v[n])$. Logo, para o novo $v[i]$, também se tem $v[i-1] \geq v[i]$.
- Assim, após a iteração i, tem-se $v[1] \geq v[2] \geq \ldots \geq v[i-1] \geq v[i]$ e $v[i] \geq \max(v[i+1], \ldots, v[n])$, e k tem valor $i + 1$.

Revisão: Ordenação por seleção (selection sort)

```
SelectionSort(v, n)
  Para cada k ← 1 até n − 1 fazer
    j ← PosMax(v, k, n);
    Se j ≠ k então
      aux ← v[k];
      v[k] ← v[j];
      v[j] ← aux;
```

Para $i \geq 1$, após a i-ésima iteração do ciclo "Para", $k = i + 1$, v tem os elementos que tinha antes do ciclo, talvez noutras posições, $v[1] \geq v[2] \geq \ldots \geq v[i]$ e, se $i < n$ então $v[i] \geq \max(v[i+1], \ldots, v[n])$.

Conclusão:

- A propriedade enunciada é válida no final de cada iteração do ciclo "Para".

Conclusão da prova de correção:

- O ciclo termina quando $k = n$, ou seja, após a iteração $n - 1$.
- De acordo com o invariante demonstrado, nesse momento, $v[1] \geq v[2] \geq \ldots \geq v[n-1]$ e $v[n-1] \geq \max(v[n], \ldots, v[n]) = v[n]$, tendo v exatamente os mesmos elementos que tinha antes da entrada no ciclo mas, possivelmente, por outra ordem.
- Logo, quando o ciclo termina, tem-se $v[1] \geq v[2] \geq \ldots \geq v[n-1] \geq v[n]$, tendo sido corretamente ordenado.

  (cqd)

Revisão: Ordenação por seleção (selection sort)

```
SelectionSort(v, n)
  Para cada k ← 1 até n − 1 fazer
    j ← PosMax(v, k, n);
    Se j ≠ k então
      aux ← v[k];
      v[k] ← v[j];
      v[j] ← aux;
```

Para $i \geq 1$, após a i-ésima iteração do ciclo "Para", $k = i + 1$, v tem os elementos que tinha antes do ciclo, talvez noutras posições, $v[1] \geq v[2] \geq \ldots \geq v[i]$ e, se $i < n$ então $v[i] \geq \max(v[i+1], \ldots, v[n])$.

Conclusão:

- A propriedade enunciada é válida no final de cada iteração do ciclo "Para".

Conclusão da prova de correção:

- O ciclo termina quando $k = n$, ou seja, após a iteração $n - 1$.
- De acordo com o invariante demonstrado, nesse momento, $v[1] \geq v[2] \geq \ldots \geq v[n-1]$ e $v[n-1] \geq \max(v[n], \ldots, v[n]) = v[n]$, tendo v exatamente os mesmos elementos que tinha antes da entrada no ciclo mas, possivelmente, por outra ordem.
- Logo, quando o ciclo termina, tem-se $v[1] \geq v[2] \geq \ldots \geq v[n-1] \geq v[n]$, tendo sido corretamente ordenado.

  (cqd)

Revisão: Ordenação por seleção (selection sort)

```
SelectionSort(v, n)
  Para cada k ← 1 até n − 1 fazer
     j ← PosMax(v, k, n);
     Se j ≠ k então
       aux ← v[k];
       v[k] ← v[j];
       v[j] ← aux;
```

Para $i \geq 1$, após a i-ésima iteração do ciclo "Para", $k = i + 1$, v tem os elementos que tinha antes do ciclo, talvez noutras posições, $v[1] \geq v[2] \geq \ldots \geq v[i]$ e, se $i < n$ então $v[i] \geq \max(v[i+1], \ldots, v[n])$.

Conclusão:

- A propriedade enunciada é válida no final de cada iteração do ciclo "Para".

Conclusão da prova de correção:

- O ciclo termina quando $k = n$, ou seja, após a iteração $n - 1$.
- De acordo com o invariante demonstrado, nesse momento, $v[1] \geq v[2] \geq \ldots \geq v[n-1]$ e $v[n-1] \geq \max(v[n], \ldots, v[n]) = v[n]$, tendo v exatamente os mesmos elementos que tinha antes da entrada no ciclo mas, possivelmente, por outra ordem.
- Logo, quando o ciclo termina, tem-se $v[1] \geq v[2] \geq \ldots \geq v[n-1] \geq v[n]$, tendo sido corretamente ordenado.

  (cqd)

Revisão: Ordenação por seleção (selection sort)

SelectionSort(v, n)
  Para cada $k \leftarrow 1$ até $n-1$ fazer
    $j \leftarrow$ PosMax(v, k, n);
    Se $j \neq k$ então
      $aux \leftarrow v[k]$;
      $v[k] \leftarrow v[j]$;
      $v[j] \leftarrow aux$;

Para $i \geq 1$, após a i-ésima iteração do ciclo "Para", $k = i+1$, v tem os elementos que tinha antes do ciclo, talvez noutras posições, $v[1] \geq v[2] \geq \ldots \geq v[i]$ e, se $i < n$ então $v[i] \geq \max(v[i+1], \ldots, v[n])$.

Conclusão:

- A propriedade enunciada é válida no final de cada iteração do ciclo "Para".

Conclusão da prova de correção:

- O ciclo termina quando $k = n$, ou seja, após a iteração $n-1$.
- De acordo com o invariante demonstrado, nesse momento, $v[1] \geq v[2] \geq \ldots \geq v[n-1]$ e $v[n-1] \geq \max(v[n], \ldots, v[n]) = v[n]$, tendo v exatamente os mesmos elementos que tinha antes da entrada no ciclo mas, possivelmente, por outra ordem.
- Logo, quando o ciclo termina, tem-se $v[1] \geq v[2] \geq \ldots \geq v[n-1] \geq v[n]$, tendo sido corretamente ordenado.

  (cqd)

Análise de complexidade de SelectionSort

SelectionSort(v, n)

1.	Para cada $k \leftarrow 1$ até $n-1$ fazer	inicialização, testes e transf. controlo, incrementos de k
2.	$\quad j \leftarrow$ PosMax(v, k, n);	chamadas da função, execução da função e atribuições do valor
3.	$\quad$ Se $j \neq k$ então	testes e transferências de controlo
4.	$\quad\quad aux \leftarrow v[k]$;	
5.	$\quad\quad v[k] \leftarrow v[j]$;	trocas dos valores? (no pior caso e melhor caso)
6.	$\quad\quad v[j] \leftarrow aux$;	
7.	**retornar**	sempre constante; podemos omitir

- (Linha 1) Inicialização de k, testes de fim de ciclo e transferências de controlo, incrementos de k: $t_1 + t_2 n + t_3(n-1)$
- (Linha 2) chamadas de PosMax, execução, e atribuição do valor a j

$$t_4(n-1) + \sum_{k=1}^{n-1} T(k,n) + t_5(n-1)$$

  - melhor caso - v estritamente decrescente: $T(k,n) \geq c'(n-k+1)$
  - pior caso para PosMax isolada - v estritamente crescente: $T(k,n) \leq c''(n-k+1)$

Análise de complexidade de SelectionSort

SelectionSort(v, n)

1.	Para cada $k \leftarrow 1$ até $n-1$ fazer	inicialização, testes e transf. controlo, incrementos de k
2.	$j \leftarrow$ PosMax(v,k,n);	chamadas da função, execução da função e atribuições do valor
3.	Se $j \neq k$ então	testes e transferências de controlo
4.	$aux \leftarrow v[k]$;	
5.	$v[k] \leftarrow v[j]$;	trocas dos valores? (no pior caso e melhor caso)
6.	$v[j] \leftarrow aux$;	
7.	**retornar**	sempre constante; podemos omitir

- (Linha 1) Inicialização de k, testes de fim de ciclo e transferências de controlo, incrementos de k: $t_1 + t_2 n + t_3(n-1)$
- (Linha 2) chamadas de PosMax, execução, e atribuição do valor a j

$$t_4(n-1) + \sum_{k=1}^{n-1} T(k,n) + t_5(n-1)$$

  - melhor caso - v estritamente decrescente: $T(k,n) \geq c'(n-k+1)$
  - pior caso para PosMax isolada - v estritamente crescente: $T(k,n) \leq c''(n-k+1)$

Análise de complexidade de SelectionSort

SelectionSort(v, n)

1.	Para cada $k \leftarrow 1$ até $n-1$ fazer	inicialização, testes e transf. controlo, incrementos de k
2.	$j \leftarrow$ PosMax(v, k, n);	chamadas da função, execução da função e atribuições do valor
3.	Se $j \neq k$ então	testes e transferências de controlo
4.	$aux \leftarrow v[k]$;	
5.	$v[k] \leftarrow v[j]$;	trocas dos valores? (no pior caso e melhor caso)
6.	$v[j] \leftarrow aux$;	
7.	**retornar**	sempre constante; podemos omitir

- (Linha 3) testes da condição e transferências de controlo: $t_6(n-1)$
- (Linhas 4-6) acessos aos valores e atribuições (para troca de $v[k]$ com $v[j]$)
  - melhor caso (v por ordem estritamente decrescente): 0
  - pior caso para este bloco (sempre troca): $v_n > v_1 > \ldots > v_{n-1}$

$$t_7(n-1)$$

- (Linha 7) retorno: t_8.

Análise de complexidade de SelectionSort

SelectionSort(v, n)

1.	Para cada $k \leftarrow 1$ até $n-1$ fazer	inicialização, testes e transf. controlo, incrementos de k
2.	$\quad j \leftarrow$ PosMax(v, k, n);	chamadas da função, execução da função e atribuições do valor
3.	$\quad$ Se $j \neq k$ então	testes e transferências de controlo
4.	$\quad\quad aux \leftarrow v[k]$;	
5.	$\quad\quad v[k] \leftarrow v[j]$;	trocas dos valores? (no pior caso e melhor caso)
6.	$\quad\quad v[j] \leftarrow aux$;	
7.	**retornar**	sempre constante; podemos omitir

- (Linha 3) testes da condição e transferências de controlo: $t_6(n-1)$
- (Linhas 4-6) acessos aos valores e atribuições (para troca de $v[k]$ com $v[j]$)
  - melhor caso (v por ordem estritamente decrescente): 0
  - pior caso para este bloco (sempre troca): $v_n > v_1 > \ldots > v_{n-1}$

$$t_7(n-1)$$

- (Linha 7) retorno: t_8.

Análise de complexidade de SelectionSort

SelectionSort(v, n)

1.	Para cada $k \leftarrow 1$ até $n-1$ fazer	inicialização, testes e transf. controlo, incrementos de k
2.	$j \leftarrow$ PosMax(v, k, n);	chamadas da função, execução da função e atribuições do valor
3.	Se $j \neq k$ então	testes e transferências de controlo
4.	$aux \leftarrow v[k]$;	
5.	$v[k] \leftarrow v[j]$;	trocas dos valores? (no pior caso e melhor caso)
6.	$v[j] \leftarrow aux$;	
7.	**retornar**	sempre constante; podemos omitir

- (Linha 3) testes da condição e transferências de controlo: $t_6(n-1)$
- (Linhas 4-6) acessos aos valores e atribuições (para troca de $v[k]$ com $v[j]$)
  - melhor caso (v por ordem estritamente decrescente): 0
  - pior caso para este bloco (sempre troca): $v_n > v_1 > \ldots > v_{n-1}$

$$t_7(n-1)$$

- (Linha 7) retorno: t_8.

Complexidade Assintótica de SelectionSort: $\Theta(n^2)$

- Melhor caso – v por ordem estritamente decrescente

$$T(n) \geq t_1 + t_2 n + (t_3 + t_4)(n-1) + \sum_{k=1}^{n-1}(c'(n-k+1)) + (t_5 + t_6)(n-1) + t_8$$

Para α igual ao mínimo de todos os t_i's e c' tem-se:

$$T(n) \geq \alpha \left(1 + n + 2(n-1) + \sum_{k=1}^{n-1}(n-k+1) + 2(n-1) + 1\right)$$

- Pior caso (majoração do tempo de execução):

$$T(n) \leq t_1 + t_2 n + (t_3 + t_4)(n-1) + \sum_{k=1}^{n-1}(c''(n-k+1)) + (t_5 + t_6)(n-1) + t_8 + t_7(n-1)$$

Para β igual ao máximo e todos os t_i's e c'' tem-se:

$$T(n) \leq \beta \left(1 + n + 2(n-1) + \sum_{k=1}^{n-1}(n-k+1) + 3(n-1) + 1\right)$$

- **Tempo** $\boxed{T(n) \in \Theta(n^2)}$ **Memória:** $\boxed{\text{“in-place” – } O(1) \text{ além dos dados (total } \Theta(n))}$

Complexidade Assintótica de SelectionSort: $\Theta(n^2)$

- Melhor caso – v por ordem estritamente decrescente

$$T(n) \geq t_1 + t_2 n + (t_3 + t_4)(n-1) + \sum_{k=1}^{n-1}(c'(n-k+1)) + (t_5 + t_6)(n-1) + t_8$$

Para α igual ao mínimo de todos os t_i's e c' tem-se:

$$T(n) \geq \alpha \left(1 + n + 2(n-1) + \sum_{k=1}^{n-1}(n-k+1) + 2(n-1) + 1\right)$$

- Pior caso (majoração do tempo de execução):

$$T(n) \leq t_1 + t_2 n + (t_3 + t_4)(n-1) + \sum_{k=1}^{n-1}(c''(n-k+1)) + (t_5 + t_6)(n-1) + t_8 + t_7(n-1)$$

Para β igual ao máximo e todos os t_i's e c'' tem-se:

$$T(n) \leq \beta \left(1 + n + 2(n-1) + \sum_{k=1}^{n-1}(n-k+1) + 3(n-1) + 1\right)$$

- **Tempo** $\boxed{T(n) \in \Theta(n^2)}$ **Memória:** $\boxed{\text{“in-place” – } O(1) \text{ além dos dados (total } \Theta(n))}$

Complexidade Assintótica de SelectionSort: $\Theta(n^2)$

- Melhor caso – v por ordem estritamente decrescente

$$T(n) \geq t_1 + t_2 n + (t_3 + t_4)(n-1) + \sum_{k=1}^{n-1} (c'(n-k+1)) + (t_5 + t_6)(n-1) + t_8$$

Para α igual ao mínimo de todos os t_i's e c' tem-se:

$$T(n) \geq \alpha \left(1 + n + 2(n-1) + \sum_{k=1}^{n-1} (n-k+1) + 2(n-1) + 1 \right)$$

- Pior caso (majoração do tempo de execução):

$$T(n) \leq t_1 + t_2 n + (t_3 + t_4)(n-1) + \sum_{k=1}^{n-1} (c''(n-k+1)) + (t_5 + t_6)(n-1) + t_8 + t_7(n-1)$$

Para β igual ao máximo e todos os t_i's e c'' tem-se:

$$T(n) \leq \beta \left(1 + n + 2(n-1) + \sum_{k=1}^{n-1} (n-k+1) + 3(n-1) + 1 \right)$$

- **Tempo** $\boxed{T(n) \in \Theta(n^2)}$ **Memória:** "in-place" – $O(1)$ além dos dados (total $\Theta(n)$)

Complexidade Assintótica de SelectionSort: $\Theta(n^2)$

- Melhor caso – v por ordem estritamente decrescente

$$T(n) \geq t_1 + t_2 n + (t_3 + t_4)(n-1) + \sum_{k=1}^{n-1}(c'(n-k+1)) + (t_5 + t_6)(n-1) + t_8$$

Para α igual ao mínimo de todos os t_i's e c' tem-se:

$$T(n) \geq \alpha \left(1 + n + 2(n-1) + \sum_{k=1}^{n-1}(n-k+1) + 2(n-1) + 1\right)$$

- Pior caso (majoração do tempo de execução):

$$T(n) \leq t_1 + t_2 n + (t_3 + t_4)(n-1) + \sum_{k=1}^{n-1}(c''(n-k+1)) + (t_5 + t_6)(n-1) + t_8 + t_7(n-1)$$

Para β igual ao máximo e todos os t_i's e c'' tem-se:

$$T(n) \leq \beta \left(1 + n + 2(n-1) + \sum_{k=1}^{n-1}(n-k+1) + 3(n-1) + 1\right)$$

- **Tempo** $T(n) \in \Theta(n^2)$ **Memória:** "in-place" – $O(1)$ além dos dados (total $\Theta(n)$)

Complexidade Assintótica de SelectionSort: $\Theta(n^2)$

- Melhor caso – v por ordem estritamente decrescente

$$T(n) \geq t_1 + t_2 n + (t_3 + t_4)(n-1) + \sum_{k=1}^{n-1} (c'(n-k+1)) + (t_5 + t_6)(n-1) + t_8$$

Para α igual ao mínimo de todos os t_i's e c' tem-se:

$$T(n) \geq \alpha \left(1 + n + 2(n-1) + \sum_{k=1}^{n-1} (n-k+1) + 2(n-1) + 1 \right)$$

- Pior caso (majoração do tempo de execução):

$$T(n) \leq t_1 + t_2 n + (t_3 + t_4)(n-1) + \sum_{k=1}^{n-1} (c''(n-k+1)) + (t_5 + t_6)(n-1) + t_8 + t_7(n-1)$$

Para β igual ao máximo e todos os t_i's e c'' tem-se:

$$T(n) \leq \beta \left(1 + n + 2(n-1) + \sum_{k=1}^{n-1} (n-k+1) + 3(n-1) + 1 \right)$$

- **Tempo** $\boxed{T(n) \in \Theta(n^2)}$ **Memória:** $\boxed{\text{“in-place” – } O(1) \text{ além dos dados (total } \Theta(n))}$

Complexidade Assintótica de SelectionSort: $\Theta(n^2)$

De facto, observando que

$$\sum_{k=1}^{n-1}(n-k+1) = n + (n-1) + \cdots + 2 = \sum_{j=1}^{n} j - 1 = \frac{n(n+1)}{2} - 1$$

sendo

$$T(n) \geq \alpha(1 + n + 2(n-1) + \sum_{k=1}^{n-1}(n-k+1) + 2(n-1) + 1)$$

$$T(n) \leq \beta(1 + n + 2(n-1) + \sum_{k=1}^{n-1}(n-k+1) + 3(n-1) + 1)$$

ou seja

$$T(n) \geq \alpha\left(\frac{1}{2}n^2 + \frac{11}{2}n - 3\right) \geq \frac{\alpha}{2}n^2, \text{ para } n \geq 1, \qquad \therefore T(n) \in \Omega(n^2)$$

e

$$T(n) \leq \beta\left(\frac{1}{2}n^2 + \frac{13}{2}n - 4\right) \leq (7\beta)n^2, \text{ para } n \geq 1, \qquad \therefore T(n) \in O(n^2)$$

concluimos que $\boxed{T(n) \in \Theta(n^2)}$

Complexidade Assintótica de SelectionSort: $\Theta(n^2)$

De facto, observando que

$$\sum_{k=1}^{n-1}(n-k+1) = n + (n-1) + \cdots + 2 = \sum_{j=1}^{n} j - 1 = \frac{n(n+1)}{2} - 1$$

sendo

$$T(n) \geq \alpha(1 + n + 2(n-1) + \sum_{k=1}^{n-1}(n-k+1) + 2(n-1) + 1)$$

$$T(n) \leq \beta(1 + n + 2(n-1) + \sum_{k=1}^{n-1}(n-k+1) + 3(n-1) + 1)$$

ou seja

$$T(n) \geq \alpha\left(\frac{1}{2}n^2 + \frac{11}{2}n - 3\right) \geq \frac{\alpha}{2}n^2, \text{ para } n \geq 1, \qquad \therefore T(n) \in \Omega(n^2)$$

e

$$T(n) \leq \beta\left(\frac{1}{2}n^2 + \frac{13}{2}n - 4\right) \leq (7\beta)n^2, \text{ para } n \geq 1, \qquad \therefore T(n) \in O(n^2)$$

concluimos que $\boxed{T(n) \in \Theta(n^2)}$

Complexidade Assintótica de SelectionSort: $\Theta(n^2)$

De facto, observando que

$$\sum_{k=1}^{n-1}(n-k+1) = n + (n-1) + \cdots + 2 = \sum_{j=1}^{n} j - 1 = \frac{n(n+1)}{2} - 1$$

sendo

$$T(n) \geq \alpha(1 + n + 2(n-1) + \sum_{k=1}^{n-1}(n-k+1) + 2(n-1) + 1)$$

$$T(n) \leq \beta(1 + n + 2(n-1) + \sum_{k=1}^{n-1}(n-k+1) + 3(n-1) + 1)$$

ou seja

$$T(n) \geq \alpha\left(\frac{1}{2}n^2 + \frac{11}{2}n - 3\right) \geq \frac{\alpha}{2}n^2, \text{para } n \geq 1, \qquad \therefore\ T(n) \in \Omega(n^2)$$

e

$$T(n) \leq \beta\left(\frac{1}{2}n^2 + \frac{13}{2}n - 4\right) \leq (7\beta)n^2, \text{para } n \geq 1, \qquad \therefore\ T(n) \in O(n^2)$$

concluimos que $\boxed{T(n) \in \Theta(n^2)}$

Exemplo 2 – Ordenação por inserção (insertion sort)

Problema: Ordenar $v[1], \ldots, v[n]$ por **ordem decrescente**, dados v e n.

```
InsertionSort(v, n)
  k ← 2;
  Enquanto k ≤ n fazer
      x ← v[k];  j ← k − 1;
      Enquanto (j ≥ 1 ∧ v[j] < x) fazer
          v[j + 1] ← v[j];   j ← j − 1;
      v[j + 1] ← x;   k ← k + 1;
```

- **Abordagem incremental**: se $v^0[1], \ldots, v^0[n]$ é o estado inicial de v então, após a iteração i, em $v[1], \ldots, v[i+1]$ terá $v^0[1], \ldots, v^0[i+1]$ mas ordenado.
- **Invariante de ciclo:** à entrada da i-ésima iteração do ciclo "Enquanto $k \leq n$", as primeiras i posições de v estão ordenadas e têm os valores de $v^0[1], \ldots, v^0[i]$ (por ordem decrescente) e as posições $v[i+1], \ldots, v[n]$ mantêm os valores originais (e não foram analisadas). O valor de k é $i+1$.

Exemplo 2 – Ordenação por inserção (insertion sort)

Problema: Ordenar $v[1], \ldots, v[n]$ por **ordem decrescente**, dados v e n.

$\text{INSERTIONSORT}(v, n)$

```
k ← 2;
Enquanto k ≤ n fazer
    x ← v[k];  j ← k − 1;
    Enquanto (j ≥ 1 ∧ v[j] < x) fazer
        v[j + 1] ← v[j];  j ← j − 1;
    v[j + 1] ← x;  k ← k + 1;
```

- **Abordagem incremental**: se $v^0[1], \ldots, v^0[n]$ é o estado inicial de v então, após a iteração i, em $v[1], \ldots, v[i+1]$ terá $v^0[1], \ldots, v^0[i+1]$ mas ordenado.
- **Invariante de ciclo:** à entrada da i-ésima iteração do ciclo "Enquanto $k \leq n$", as primeiras i posições de v estão ordenadas e têm os valores de $v^0[1], \ldots, v^0[i]$ (por ordem decrescente) e as posições $v[i+1], \ldots, v[n]$ mantêm os valores originais (e não foram analisadas). O valor de k é $i+1$.

Exemplo 2 – Ordenação por inserção (insertion sort)

Problema: Ordenar $v[1], \ldots, v[n]$ por **ordem decrescente**, dados v e n.

```
InsertionSort(v, n)
  k ← 2;
  Enquanto k ≤ n fazer
      x ← v[k];  j ← k − 1;
      Enquanto (j ≥ 1 ∧ v[j] < x) fazer
            v[j + 1] ← v[j];   j ← j − 1;
      v[j + 1] ← x;   k ← k + 1;
```

- **Abordagem incremental**: se $v^0[1], \ldots, v^0[n]$ é o estado inicial de v então, após a iteração i, em $v[1], \ldots, v[i+1]$ terá $v^0[1], \ldots, v^0[i+1]$ mas ordenado.
- **Invariante de ciclo:** à entrada da i-ésima iteração do ciclo "Enquanto $k \leq n$", as primeiras i posições de v estão ordenadas e têm os valores de $v^0[1], \ldots, v^0[i]$ (por ordem decrescente) e as posições $v[i+1], \ldots, v[n]$ mantêm os valores originais (e não foram analisadas). O valor de k é $i + 1$.

Ordenação por inserção (insertion sort)

Para mostrar esse invariante pode-se começar por analisar o segundo ciclo e provar que:

- **Invariante do ciclo "Enquanto ($j \geq 1 \wedge v[j] < x$)" para k fixo:**

  Supondo que $v^k[1], \ldots, v^k[n]$ é o estado inicial de v à entrada do ciclo (4–5) e que $v^k[1] \geq v^k[2] \geq \ldots \geq v^k[k-1]$, então os valores de k, x e de $v[k+1], \ldots, v[n]$ não são alterados no ciclo, sendo $x = v^k[k]$, e quando se vai testar a condição do ciclo pela m-ésima vez, tem-se:

  - o valor de j é $k-m$ e a posição $v[j+1]$ está "livre" (i.e., o seu valor está em x se $m=1$ ou já foi copiado para $v[j+2]$ se $m>1$)
  - $v[p] = v^k[p-1] > x$, para $j+2 \leq p \leq k$,
  - $v[p] = v^k[p]$, para $1 \leq p \leq j$

$v^k[1]$	$v^k[2]$	$\cdots$	$v^k[j]$	****	$v^k[j+1]$	$\cdots$	$v^k[k-1]$

```
InsertionSort(v, n)
1   k ← 2;
2   Enquanto k ≤ n fazer
3       x ← v[k];  j ← k − 1;
4       Enquanto (j ≥ 1 ∧ v[j] < x) fazer
5           v[j + 1] ← v[j];   j ← j − 1;
6       v[j + 1] ← x;   k ← k + 1;
```

Ordenação por inserção (insertion sort)

Para mostrar esse invariante pode-se começar por analisar o segundo ciclo e provar que:

- **Invariante do ciclo "Enquanto ($j \geq 1 \wedge v[j] < x$)" para k fixo:**
  Supondo que $v^k[1], \ldots, v^k[n]$ é o estado inicial de v à entrada do ciclo (4–5) e que $v^k[1] \geq v^k[2] \geq \ldots \geq v^k[k-1]$, então os valores de k, x e de $v[k+1], \ldots, v[n]$ não são alterados no ciclo, sendo $x = v^k[k]$, e quando se vai testar a condição do ciclo pela m-ésima vez, tem-se:
  - o valor de j é $k - m$ e a posição $v[j+1]$ está "livre" (i.e., o seu valor está em x se $m = 1$ ou já foi copiado para $v[j+2]$ se $m > 1$)
  - $v[p] = v^k[p-1] > x$, para $j + 2 \leq p \leq k$,
  - $v[p] = v^k[p]$, para $1 \leq p \leq j$

$v^k[1]$	$v^k[2]$	$\cdots$	$v^k[j]$	****	$v^k[j+1]$	$\cdots$	$v^k[k-1]$

```
InsertionSort(v, n)
1  k ← 2;
2  Enquanto k ≤ n fazer
3      x ← v[k];  j ← k − 1;
4      Enquanto (j ≥ 1 ∧ v[j] < x) fazer
5          v[j + 1] ← v[j];   j ← j − 1;
6      v[j + 1] ← x;   k ← k + 1;
```

Ordenação por inserção (insertion sort)

Para mostrar esse invariante pode-se começar por analisar o segundo ciclo e provar que:

- **Invariante do ciclo "Enquanto ($j \geq 1 \wedge v[j] < x$)" para k fixo:**
  Supondo que $v^k[1], \ldots, v^k[n]$ é o estado inicial de v à entrada do ciclo (4–5) e que $v^k[1] \geq v^k[2] \geq \ldots \geq v^k[k-1]$, então os valores de k, x e de $v[k+1], \ldots, v[n]$ não são alterados no ciclo, sendo $x = v^k[k]$, e quando se vai testar a condição do ciclo pela m-ésima vez, tem-se:
  - o valor de j é $k - m$ e a posição $v[j+1]$ está "livre" (i.e., o seu valor está em x se $m = 1$ ou já foi copiado para $v[j+2]$ se $m > 1$)
  - $v[p] = v^k[p-1] > x$, para $j + 2 \leq p \leq k$,
  - $v[p] = v^k[p]$, para $1 \leq p \leq j$

$v^k[1]$	$v^k[2]$	$\cdots$	$v^k[j]$	****	$v^k[j+1]$	$\cdots$	$v^k[k-1]$

```
InsertionSort(v, n)
1  k ← 2;
2  Enquanto k ≤ n fazer
3     x ← v[k];  j ← k − 1;
4     Enquanto (j ≥ 1 ∧ v[j] < x) fazer
5         v[j + 1] ← v[j];   j ← j − 1;
6     v[j + 1] ← x;   k ← k + 1;
```

Ordenação por inserção (insertion sort)

Para mostrar esse invariante pode-se começar por analisar o segundo ciclo e provar que:

- **Invariante do ciclo "Enquanto ($j \geq 1 \wedge v[j] < x$)" para k fixo:**

  Supondo que $v^k[1], \ldots, v^k[n]$ é o estado inicial de v à entrada do ciclo (4–5) e que $v^k[1] \geq v^k[2] \geq \ldots \geq v^k[k-1]$, então os valores de k, x e de $v[k+1], \ldots, v[n]$ não são alterados no ciclo, sendo $x = v^k[k]$, e quando se vai testar a condição do ciclo pela m-ésima vez, tem-se:

  - o valor de j é $k - m$ e a posição $v[j+1]$ está "livre" (i.e., o seu valor está em x se $m = 1$ ou já foi copiado para $v[j+2]$ se $m > 1$)
  - $v[p] = v^k[p-1] > x$, para $j + 2 \leq p \leq k$,
  - $v[p] = v^k[p]$, para $1 \leq p \leq j$

$v^k[1]$	$v^k[2]$	$\cdots$	$v^k[j]$	****	$v^k[j+1]$	$\cdots$	$v^k[k-1]$

```
InsertionSort(v, n)
1  k ← 2;
2  Enquanto k ≤ n fazer
3      x ← v[k];  j ← k − 1;
4      Enquanto (j ≥ 1 ∧ v[j] < x) fazer
5          v[j + 1] ← v[j];   j ← j − 1;
6      v[j + 1] ← x;   k ← k + 1;
```

Ordenação por inserção (insertion sort)

Para mostrar esse invariante pode-se começar por analisar o segundo ciclo e provar que:

- **Invariante do ciclo "Enquanto ($j \geq 1 \wedge v[j] < x$)" para k fixo:**
  Supondo que $v^k[1], \ldots, v^k[n]$ é o estado inicial de v à entrada do ciclo (4–5) e que $v^k[1] \geq v^k[2] \geq \ldots \geq v^k[k-1]$, então os valores de k, x e de $v[k+1], \ldots, v[n]$ não são alterados no ciclo, sendo $x = v^k[k]$, e quando se vai testar a condição do ciclo pela m-ésima vez, tem-se:
  - o valor de j é $k - m$ e a posição $v[j+1]$ está "livre" (i.e., o seu valor está em x se $m = 1$ ou já foi copiado para $v[j+2]$ se $m > 1$)
  - $v[p] = v^k[p-1] > x$, para $j + 2 \leq p \leq k$,
  - $v[p] = v^k[p]$, para $1 \leq p \leq j$

$v^k[1]$	$v^k[2]$	$\cdots$	$v^k[j]$	****	$v^k[j+1]$	$\cdots$	$v^k[k-1]$

```
InsertionSort(v, n)
1  k ← 2;
2  Enquanto k ≤ n fazer
3     x ← v[k];  j ← k − 1;
4     Enquanto (j ≥ 1 ∧ v[j] < x) fazer
5          v[j + 1] ← v[j];   j ← j − 1;
6     v[j + 1] ← x;   k ← k + 1;
```

Ordenação por inserção (insertion sort)

Para mostrar esse invariante pode-se começar por analisar o segundo ciclo e provar que:

- **Invariante do ciclo "Enquanto ($j \geq 1 \wedge v[j] < x$)" para k fixo:** Supondo que $v^k[1], \ldots, v^k[n]$ é o estado inicial de v à entrada do ciclo (4–5) e que $v^k[1] \geq v^k[2] \geq \ldots \geq v^k[k-1]$, então os valores de k, x e de $v[k+1], \ldots, v[n]$ não são alterados no ciclo, sendo $x = v^k[k]$, e quando se vai testar a condição do ciclo pela m-ésima vez, tem-se:
  - o valor de j é $k - m$ e a posição $v[j+1]$ está "livre" (i.e., o seu valor está em x se $m = 1$ ou já foi copiado para $v[j+2]$ se $m > 1$)
  - $v[p] = v^k[p-1] > x$, para $j + 2 \leq p \leq k$,
  - $v[p] = v^k[p]$, para $1 \leq p \leq j$

$v^k[1]$	$v^k[2]$	$\cdots$	$v^k[j]$	****	$v^k[j+1]$	$\cdots$	$v^k[k-1]$

- **Conclusão:** Se, após o ciclo 4–5, inserir x na posição $j+1$ (linha 6), terá $v[1] \geq \cdots \geq v[j] \geq v[j+1] > v[j+2] \geq \cdots \geq v[k]$ (antes de incrementar k) e tais posições contêm os valores $v^0[1], \ldots, v^0[k-1], v^0[k]$ ordenados se à entrada do ciclo 3–6, as $k-1$ primeiras posições de v contiverem os $v^0[1], \ldots, v^0[k-1]$ ordenados.

Ordenação por inserção (insertion sort)

Para mostrar esse invariante pode-se começar por analisar o segundo ciclo e provar que:

- **Invariante do ciclo "Enquanto ($j \geq 1 \wedge v[j] < x$)" para k fixo:**
  Supondo que $v^k[1], \ldots, v^k[n]$ é o estado inicial de v à entrada do ciclo (4–5) e que $v^k[1] \geq v^k[2] \geq \ldots \geq v^k[k-1]$, então os valores de k, x e de $v[k+1], \ldots, v[n]$ não são alterados no ciclo, sendo $x = v^k[k]$, e quando se vai testar a condição do ciclo pela m-ésima vez, tem-se:
  - o valor de j é $k - m$ e a posição $v[j+1]$ está "livre" (i.e., o seu valor está em x se $m = 1$ ou já foi copiado para $v[j+2]$ se $m > 1$)
  - $v[p] = v^k[p-1] > x$, para $j+2 \leq p \leq k$,
  - $v[p] = v^k[p]$, para $1 \leq p \leq j$

$v^k[1]$	$v^k[2]$	$\cdots$	$v^k[j]$	****	$v^k[j+1]$	$\cdots$	$v^k[k-1]$

- **Conclusão:** Se, após o ciclo 4–5, inserir x na posição $j+1$ (linha 6), terá $v[1] \geq \cdots \geq v[j] \geq v[j+1] > v[j+2] \geq \cdots \geq v[k]$ (antes de incrementar k) e tais posições contêm os valores $v^0[1], \ldots, v^0[k-1], v^0[k]$ ordenados se à entrada do ciclo 3–6, as $k-1$ primeiras posições de v contiverem os $v^0[1], \ldots, v^0[k-1]$ ordenados.

Ordenação por inserção (insertion sort)

Complexidade temporal assintótica
Caraterização da ordem de grandeza de $T(n)$.

```
InsertionSort(v, n)
  k ← 2;                                   c1: atribuição de valor a variável
  Enquanto k ≤ n fazer                     c2: teste e transferências de controlo
      x ← v[k];  j ← k − 1;                c3: acesso à memória, cálculos, atribuição
      Enquanto (j ≥ 1 ∧ v[j] < x) fazer    c4: teste e transferência de controlo
          v[j + 1] ← v[j];   j ← j − 1;    c5: acesso à memória, cálculos, atribuição
      v[j + 1] ← x;   k ← k + 1;           c6: acesso à memória, cálculos, atribuição
```

c_1, c_2, c_3, c_4, c_5 constantes positivas.

$T(n)$ **no pior caso** e $T(n)$ **no melhor caso**

- O que carateriza as instâncias no pior caso e no melhor caso?
- Qual é a expressão de $T(n)$ em cada um desses casos?

Ordenação por inserção (insertion sort)

Complexidade temporal assintótica
Caraterização da ordem de grandeza de $T(n)$.

$\textsc{InsertionSort}(v, n)$	
$k \leftarrow 2;$	c_1: atribuição de valor a variável
Enquanto $k \leq n$ fazer	c_2: teste e transferências de controlo
$x \leftarrow v[k]; \;\; j \leftarrow k-1;$	c_3: acesso à memória, cálculos, atribuição
Enquanto $(j \geq 1 \wedge v[j] < x)$ fazer	c_4: teste e transferência de controlo
$v[j+1] \leftarrow v[j]; \;\; j \leftarrow j-1;$	c_5: acesso à memória, cálculos, atribuição
$v[j+1] \leftarrow x; \;\; k \leftarrow k+1;$	c_6: acesso à memória, cálculos, atribuição

c_1, c_2, c_3, c_4, c_5 constantes positivas.

$T(n)$ no pior caso e $T(n)$ no melhor caso

- O que carateriza as instâncias no pior caso e no melhor caso?
- Qual é a expressão de $T(n)$ em cada um desses casos?

Ordenação por inserção (insertion sort)

Complexidade temporal assintótica
Caraterização da ordem de grandeza de $T(n)$.

$\textsc{InsertionSort}(v, n)$

$k \leftarrow 2;$	c_1: atribuição de valor a variável
Enquanto $k \leq n$ fazer	c_2: teste e transferências de controlo
$\quad x \leftarrow v[k];\ \ j \leftarrow k-1;$	c_3: acesso à memória, cálculos, atribuição
$\quad$ Enquanto $(j \geq 1 \wedge v[j] < x)$ fazer	c_4: teste e transferência de controlo
$\qquad v[j+1] \leftarrow v[j];\ \ j \leftarrow j-1;$	c_5: acesso à memória, cálculos, atribuição
$\quad v[j+1] \leftarrow x;\ \ k \leftarrow k+1;$	c_6: acesso à memória, cálculos, atribuição

c_1, c_2, c_3, c_4, c_5 constantes positivas.

$T(n)$ no pior caso e $T(n)$ no melhor caso

- O que carateriza as instâncias no pior caso e no melhor caso?
- Qual é a expressão de $T(n)$ em cada um desses casos?

Ordenação por inserção (insertion sort)

- Pior caso: v está ordenado por ordem crescente e os valores são todos distintos, ou seja, $v^0[1] < v^0[2] < \cdots < v^0[n]$.

  Para k fixo, a condição de paragem do ciclo "enquanto" (linha 4) é testada k vezes e a instrução 5 é executada $k-1$ vezes.

$$T(n) \leq c_1 + c_2 n + (c_3 + c_6)(n-1) + c_4 \sum_{k=2}^{n} k + c_5 \sum_{k=2}^{n} (k-1)$$

- Melhor caso: v está ordenado por ordem decrescente, $v^0[1] \geq \cdots \geq v^0[n]$.

  Para k fixo, a condição de paragem do ciclo "enquanto" (linha 4) é testada **uma vez** e a instrução 5 é executada **zero** vezes.

$$T(n) \geq c_1 + c_2 n + (c_3 + c_6)(n-1) + c_4 \sum_{k=2}^{n} 1 + c_5 \sum_{k=2}^{n} 0$$

Para a expressão no pior caso, notar que: $\sum_{k=2}^{n}(k-1) = 1 + 2 + \cdots + (n-1) = \sum_{k=1}^{n-1} k$.

Ordenação por inserção (insertion sort)

- Pior caso: v está ordenado por ordem crescente e os valores são todos distintos, ou seja, $v^0[1] < v^0[2] < \cdots < v^0[n]$.

  Para k fixo, a condição de paragem do ciclo "enquanto" (linha 4) é testada k vezes e a instrução 5 é executada $k - 1$ vezes.

$$T(n) \leq c_1 + c_2 n + (c_3 + c_6)(n-1) + c_4 \sum_{k=2}^{n} k + c_5 \sum_{k=2}^{n} (k-1)$$

- Melhor caso: v está ordenado por ordem decrescente, $v^0[1] \geq \cdots \geq v^0[n]$.

  Para k fixo, a condição de paragem do ciclo "enquanto" (linha 4) é testada **uma vez** e a instrução 5 é executada **zero** vezes.

$$T(n) \geq c_1 + c_2 n + (c_3 + c_6)(n-1) + c_4 \sum_{k=2}^{n} 1 + c_5 \sum_{k=2}^{n} 0$$

Para a expressão no pior caso, notar que: $\sum_{k=2}^{n}(k-1) = 1 + 2 + \cdots + (n-1) = \sum_{k=1}^{n-1} k$.

Ordenação por inserção (insertion sort)

- Pior caso: v está ordenado por ordem crescente e os valores são todos distintos, ou seja, $v^0[1] < v^0[2] < \cdots < v^0[n]$.

  Para k fixo, a condição de paragem do ciclo "enquanto" (linha 4) é testada k vezes e a instrução 5 é executada $k-1$ vezes.

$$T(n) \leq c_1 + c_2 n + (c_3 + c_6)(n-1) + c_4 \sum_{k=2}^{n} k + c_5 \sum_{k=2}^{n}(k-1)$$

- Melhor caso: v está ordenado por ordem decrescente, $v^0[1] \geq \cdots \geq v^0[n]$.

  Para k fixo, a condição de paragem do ciclo "enquanto" (linha 4) é testada **uma vez** e a instrução 5 é executada **zero** vezes.

$$T(n) \geq c_1 + c_2 n + (c_3 + c_6)(n-1) + c_4 \sum_{k=2}^{n} 1 + c_5 \sum_{k=2}^{n} 0$$

Para a expressão no pior caso, notar que: $\sum_{k=2}^{n}(k-1) = 1 + 2 + \cdots + (n-1) = \sum_{k=1}^{n-1} k$.

Ordenação por inserção (insertion sort)

- Pior caso: v está ordenado por ordem crescente e os valores são todos distintos, ou seja, $v^0[1] < v^0[2] < \cdots < v^0[n]$.

  Para k fixo, a condição de paragem do ciclo "enquanto" (linha 4) é testada k vezes e a instrução 5 é executada $k-1$ vezes.

$$T(n) \leq c_1 + c_2 n + (c_3 + c_6)(n-1) + c_4 \sum_{k=2}^{n} k + c_5 \sum_{k=2}^{n} (k-1)$$

- Melhor caso: v está ordenado por ordem decrescente, $v^0[1] \geq \cdots \geq v^0[n]$.

  Para k fixo, a condição de paragem do ciclo "enquanto" (linha 4) é testada **uma vez** e a instrução 5 é executada **zero** vezes.

$$T(n) \geq c_1 + c_2 n + (c_3 + c_6)(n-1) + c_4 \sum_{k=2}^{n} 1 + c_5 \sum_{k=2}^{n} 0$$

Para a expressão no pior caso, notar que: $\sum_{k=2}^{n}(k-1) = 1 + 2 + \cdots + (n-1) = \sum_{k=1}^{n-1} k$.

Ordenação por inserção (insertion sort)

- Pior caso: v está ordenado por ordem crescente e os valores são todos distintos, ou seja, $v^0[1] < v^0[2] < \cdots < v^0[n]$.

  Para k fixo, a condição de paragem do ciclo "enquanto" (linha 4) é testada k vezes e a instrução 5 é executada $k-1$ vezes.

$$T(n) \leq c_1 + c_2 n + (c_3 + c_6)(n-1) + c_4 \sum_{k=2}^{n} k + c_5 \sum_{k=2}^{n} (k-1)$$

- Melhor caso: v está ordenado por ordem decrescente, $v^0[1] \geq \cdots \geq v^0[n]$.

  Para k fixo, a condição de paragem do ciclo "enquanto" (linha 4) é testada **uma vez** e a instrução 5 é executada **zero** vezes.

$$T(n) \geq c_1 + c_2 n + (c_3 + c_6)(n-1) + c_4 \sum_{k=2}^{n} 1 + c_5 \sum_{k=2}^{n} 0$$

Para a expressão no pior caso, notar que: $\sum_{k=2}^{n}(k-1) = 1 + 2 + \cdots + (n-1) = \sum_{k=1}^{n-1} k$.

Ordenação por inserção (insertion sort)

Recordar que $\sum_{k=a}^{b} k = \frac{b+a}{2}(b-a+1)$ e $\sum_{k=a}^{b} 1 = (b-a+1)$, se $b \geq a$. Caso contrário, o valor é 0.

$$c_1+c_2n+(c_3+c_6+c_4)(n-1) \leq T(n) \leq c_1+c_2n+(c_3+c_6)(n-1)+c_4\frac{n+2}{2}(n-1)+c_5\frac{n}{2}(n-1)$$

Tomando $c' = \min(c_1, c_2, \ldots, c_6)$ e $c'' = \max(c_1, c_2, \ldots, c_6)$, podemos escrever

$$4c'n - 2c' \leq T(n) \leq c'' + c''n + 2c''(n-1) + c''(n+1)(n-1)$$

Ou seja,

$$4c'n - 2c' \leq T(n) \leq c''n^2 + 3c''n - 2c''$$

e concluir que, para $n \geq 1$, se tem $2c'n \leq T(n) \leq 4c''n^2$ pois

$$4c'n - 2c'n \leq 4c'n - 2c' \leq T(n)$$

$$T(n) \leq c''n^2 + 3c''n - 2c'' \leq c''n^2 + 3c''n \leq c''n^2 + 3c''n^2 \leq 4c''n^2$$

Concluimos que o tempo de execução do algoritmo InsertionSort para instâncias de tamanho n satisfaz: $T(n) \in \Omega(n)$ e $T(n) \in O(n^2)$. Das expressões de $T(n)$, concluimos também que, no pior caso, $T(n) \in \Theta(n^2)$. No melhor caso, $T(n) \in \Theta(n)$.

Ordenação por inserção (insertion sort)

Recordar que $\sum_{k=a}^{b} k = \frac{b+a}{2}(b-a+1)$ e $\sum_{k=a}^{b} 1 = (b-a+1)$, se $b \geq a$. Caso contrário, o valor é 0.

$$c_1+c_2n+(c_3+c_6+c_4)(n-1) \leq T(n) \leq c_1+c_2n+(c_3+c_6)(n-1)+c_4\frac{n+2}{2}(n-1)+c_5\frac{n}{2}(n-1)$$

Tomando $c' = \min(c_1, c_2, \ldots, c_6)$ e $c'' = \max(c_1, c_2, \ldots, c_6)$, podemos escrever

$$4c'n - 2c' \leq T(n) \leq c'' + c''n + 2c''(n-1) + c''(n+1)(n-1)$$

Ou seja,

$$4c'n - 2c' \leq T(n) \leq c''n^2 + 3c''n - 2c''$$

e concluir que, para $n \geq 1$, se tem $2c'n \leq T(n) \leq 4c''n^2$ pois

$$4c'n - 2c'n \leq 4c'n - 2c' \leq T(n)$$

$$T(n) \leq c''n^2 + 3c''n - 2c'' \leq c''n^2 + 3c''n \leq c''n^2 + 3c''n^2 \leq 4c''n^2$$

Concluimos que o tempo de execução do algoritmo InsertionSort para instâncias de tamanho n satisfaz: $T(n) \in \Omega(n)$ e $T(n) \in O(n^2)$. Das expressões de $T(n)$, concluimos também que, no pior caso, $T(n) \in \Theta(n^2)$. No melhor caso, $T(n) \in \Theta(n)$.

Ordenação por inserção (insertion sort)

Recordar que $\sum_{k=a}^{b} k = \frac{b+a}{2}(b-a+1)$ e $\sum_{k=a}^{b} 1 = (b-a+1)$, se $b \geq a$. Caso contrário, o valor é 0.

$$c_1+c_2n+(c_3+c_6+c_4)(n-1) \leq T(n) \leq c_1+c_2n+(c_3+c_6)(n-1)+c_4\frac{n+2}{2}(n-1)+c_5\frac{n}{2}(n-1)$$

Tomando $c' = \min(c_1, c_2, \ldots, c_6)$ e $c'' = \max(c_1, c_2, \ldots, c_6)$, podemos escrever

$$4c'n - 2c' \leq T(n) \leq c'' + c''n + 2c''(n-1) + c''(n+1)(n-1)$$

Ou seja,

$$4c'n - 2c' \leq T(n) \leq c''n^2 + 3c''n - 2c''$$

e concluir que, para $n \geq 1$, se tem $2c'n \leq T(n) \leq 4c''n^2$ pois

$$4c'n - 2c'n \leq 4c'n - 2c' \leq T(n)$$

$$T(n) \leq c''n^2 + 3c''n - 2c'' \leq c''n^2 + 3c''n \leq c''n^2 + 3c''n^2 \leq 4c''n^2$$

Concluimos que o tempo de execução do algoritmo InsertionSort para instâncias de tamanho n satisfaz: $T(n) \in \Omega(n)$ e $T(n) \in O(n^2)$. Das expressões de $T(n)$, concluimos também que, no pior caso, $T(n) \in \Theta(n^2)$. No melhor caso, $T(n) \in \Theta(n)$.

Exemplos: Pesquisa num vetor ordenado

Problema: supondo que v está ordenado por ordem estritamente decrescente, obter uma posição de x no segmento $v[a], v[a+1], \ldots v[b]$, com $0 \leq a \leq b < n$. Retorna -1 se x não ocorrer.

PesquisaLinear(v, a, b, x)
Enquanto ($a \leq b \wedge v[a] > x$) fazer
    $a \leftarrow a + 1$;
Se ($a \leq b \wedge v[a] = x$) então
    retorna a;
retorna -1;

PesquisaBinaria(v, a, b, x)
Enquanto ($a \leq b$) fazer
    $m \leftarrow \lfloor (a + b)/2 \rfloor$;
    Se ($v[m] = x$) então
        retorna m;
    Se ($v[m] > x$) então
        $a \leftarrow m + 1$;
    senão $b \leftarrow m - 1$;
retorna -1;

- Provas de correção: invariantes de ciclo que permitem mostrar a correção do algoritmo? Como o permitem?
- Complexidade temporal: no melhor caso, no pior caso, e para qualquer instância?

Exemplos: Pesquisa num vetor ordenado

Problema: supondo que v está ordenado por ordem estritamente decrescente, obter uma posição de x no segmento $v[a], v[a+1], \ldots v[b]$, com $0 \leq a \leq b < n$. Retorna -1 se x não ocorrer.

PesquisaLinear(v, a, b, x)
- Enquanto ($a \leq b \wedge v[a] > x$) fazer
  - $a \leftarrow a + 1$;
- Se ($a \leq b \wedge v[a] = x$) então
  - retorna a;
- retorna -1;

PesquisaBinaria(v, a, b, x)
- Enquanto ($a \leq b$) fazer
  - $m \leftarrow \lfloor (a+b)/2 \rfloor$;
  - Se ($v[m] = x$) então
    - retorna m;
  - Se ($v[m] > x$) então
    - $a \leftarrow m + 1$;
  - senão $b \leftarrow m - 1$;
- retorna -1;

- Provas de correção: invariantes de ciclo que permitem mostrar a correção do algoritmo? Como o permitem?
- Complexidade temporal: no melhor caso, no pior caso, e para qualquer instância?

Exemplos: Pesquisa num vetor ordenado

Assume $v[a] > v[a+1] > \ldots > v[b]$, com $0 \leq a \leq b < n$.

```
PesquisaLinear(v, a, b, x)
  Enquanto (a ≤ b ∧ v[a] > x) fazer
      a ← a + 1;
  Se (a ≤ b ∧ v[a] = x) então
      retorna a;
  retorna −1;
```

```
PesquisaBinaria(v, a, b, x)
  Enquanto (a ≤ b) fazer
      m ← ⌊(a + b)/2⌋;
      Se (v[m] = x) então
          retorna m;
      Se (v[m] > x) então
          a ← m + 1;
      senão b ← m − 1;
  retorna −1;
```

Invariantes de Ciclo

Sejam a_k e b_k os valores de a e b quando a condição de ciclo é testada pela k-ésima vez, e seja $I_k = [a_k, b_k]$ o intervalo de inteiros correspondente, com $k \geq 1$.

- PesquisaLinear : Quando testa a condição pela k-ésima vez, $a_k = a + (k-1)$, $b_k = b$, o índice da posição de x não é inferior a a_k, e ainda não analisou $v[a+(k-1)], v[a+k], \ldots, v[b]$ e que $v[a+(k-2)] > x$ (se $k \geq 2$)
- PesquisaBinaria Quando testa a condição pela k-ésima vez, se x ocorrer na secção $[a, b]$ que se pretendia analisar, então o índice da posição de x tem que estar em $I_k = [a_k, b_k]$, e se $k \geq 2$ então $|I_k| < |I_{k-1}|$.

Exemplos: Pesquisa num vetor ordenado

Assume $v[a] > v[a+1] > \ldots > v[b]$, com $0 \leq a \leq b < n$.

```
PesquisaLinear(v, a, b, x)
  Enquanto (a ≤ b ∧ v[a] > x) fazer
      a ← a + 1;
  Se (a ≤ b ∧ v[a] = x) então
      retorna a;
  retorna −1;
```

```
PesquisaBinaria(v, a, b, x)
  Enquanto (a ≤ b) fazer
      m ← ⌊(a + b)/2⌋;
      Se (v[m] = x) então
          retorna m;
      Se (v[m] > x) então
          a ← m + 1;
      senão b ← m − 1;
  retorna −1;
```

Invariantes de Ciclo

Sejam a_k e b_k os valores de a e b quando a condição de ciclo é testada pela k-ésima vez, e seja $I_k = [a_k, b_k]$ o intervalo de inteiros correspondente, com $k \geq 1$.

- PesquisaLinear : Quando testa a condição pela k-ésima vez, $a_k = a + (k-1)$, $b_k = b$, o índice da posição de x não é inferior a a_k, e ainda não analisou $v[a+(k-1)], v[a+k], \ldots, v[b]$ e que $v[a+(k-2)] > x$ (se $k \geq 2$)
- PesquisaBinaria Quando testa a condição pela k-ésima vez, se x ocorrer na secção $[a, b]$ que se pretendia analisar, então o índice da posição de x tem que estar em $I_k = [a_k, b_k]$, e se $k \geq 2$ então $|I_k| < |I_{k-1}|$.

Exemplos: Pesquisa num vetor ordenado

Assume $v[a] > v[a+1] > \ldots > v[b]$, com $0 \leq a \leq b < n$.

```
PesquisaLinear(v, a, b, x)
  Enquanto (a ≤ b ∧ v[a] > x) fazer
      a ← a + 1;
  Se (a ≤ b ∧ v[a] = x) então
      retorna a;
  retorna −1;
```

```
PesquisaBinaria(v, a, b, x)
  Enquanto (a ≤ b) fazer
      m ← ⌊(a + b)/2⌋;
      Se (v[m] = x) então
          retorna m;
      Se (v[m] > x) então
          a ← m + 1;
      senão b ← m − 1;
  retorna −1;
```

Invariantes de Ciclo

Sejam a_k e b_k os valores de a e b quando a condição de ciclo é testada pela k-ésima vez, e seja $I_k = [a_k, b_k]$ o intervalo de inteiros correspondente, com $k \geq 1$.

- PesquisaLinear : Quando testa a condição pela k-ésima vez, $a_k = a + (k-1)$, $b_k = b$, o índice da posição de x não é inferior a a_k, e ainda não analisou $v[a+(k-1)], v[a+k], \ldots, v[b]$ e que $v[a+(k-2)] > x$ (se $k \geq 2$)
- PesquisaBinaria Quando testa a condição pela k-ésima vez, se x ocorrer na secção $[a, b]$ que se pretendia analisar, então o índice da posição de x tem que estar em $I_k = [a_k, b_k]$, e se $k \geq 2$ então $|I_k| < |I_{k-1}|$.

Exemplos: Pesquisa num vetor ordenado

Assume $v[a] > v[a+1] > \ldots > v[b]$, com $0 \leq a \leq b < n$.

```
PesquisaLinear(v, a, b, x)
  Enquanto (a ≤ b ∧ v[a] > x) fazer
      a ← a + 1;
  Se (a ≤ b ∧ v[a] = x) então
      retorna a;
  retorna −1;
```

```
PesquisaBinaria(v, a, b, x)
  Enquanto (a ≤ b) fazer
      m ← ⌊(a + b)/2⌋;
      Se (v[m] = x) então
          retorna m;
      Se (v[m] > x) então
          a ← m + 1;
      senão b ← m − 1;
  retorna −1;
```

Invariantes de Ciclo

Sejam a_k e b_k os valores de a e b quando a condição de ciclo é testada pela k-ésima vez, e seja $I_k = [a_k, b_k]$ o intervalo de inteiros correspondente, com $k \geq 1$.

- PesquisaLinear : Quando testa a condição pela k-ésima vez, $a_k = a + (k-1)$, $b_k = b$, o índice da posição de x não é inferior a a_k, e ainda não analisou $v[a+(k-1)], v[a+k], \ldots, v[b]$ e que $v[a+(k-2)] > x$ (se $k \geq 2$)
- PesquisaBinaria Quando testa a condição pela k-ésima vez, se x ocorrer na secção $[a, b]$ que se pretendia analisar, então o índice da posição de x tem que estar em $I_k = [a_k, b_k]$, e se $k \geq 2$ então $|I_k| < |I_{k-1}|$.

Exemplos: Pesquisa binária (*binary search*)

```
PesquisaBinaria(v, a, b, x)
  Enquanto (a ≤ b) fazer
     m ← ⌊(a + b)/2⌋;
     Se (v[m] = x) então
       retorna m;
     Se (v[m] > x) então
       a ← m + 1;
     senão b ← m − 1;
  retorna −1;
```

```
PesquisaBinariaRec(v, a, b, x)
  Se (a > b) então
     retorna −1;
  m ← ⌊(a + b)/2⌋;
  Se (v[m] = x) então
     retorna m;
  Se (v[m] > x) então
     retorna PesquisaBinariaRec(v, m + 1, b, x);
  retorna PesquisaBinariaRec(v, a, m − 1, x);
```

Prova de correção de PesquisaBinariaRec (por indução forte sobre o número de elementos do intervalo)

- Caso de base: x não se encontra no segmento definido por $[a, b]$ se $\#[a, b] = 0$, isto é se $a > b$. A função retorna corretamente -1.
- Hereditariedade: Assumimos, como hipótese, que para $\#[a, b]$ fixo, sendo a e b quaisquer $0 \leq a \leq b < n$, a função retorna o valor correto para todos os $[a', b']$, com $\#[a', b'] < \#[a, b]$, com $0 \leq a' \leq b' < n$ ou $a' > b'$. Vamos mostrar que então também dá o valor correto para $[a, b]$.

Exemplos: Pesquisa binária (*binary search*)

```
PesquisaBinaria(v, a, b, x)
  Enquanto (a ≤ b) fazer
    m ← ⌊(a + b)/2⌋;
    Se (v[m] = x) então
      retorna m;
    Se (v[m] > x) então
      a ← m + 1;
    senão b ← m − 1;
  retorna −1;
```

```
PesquisaBinariaRec(v, a, b, x)
  Se (a > b) então
    retorna −1;
  m ← ⌊(a + b)/2⌋;
  Se (v[m] = x) então
    retorna m;
  Se (v[m] > x) então
    retorna PesquisaBinariaRec(v, m + 1, b, x);
  retorna PesquisaBinariaRec(v, a, m − 1, x);
```

Prova de correção de PesquisaBinariaRec (por indução forte sobre o número de elementos do intervalo)

- Caso de base: x não se encontra no segmento definido por $[a, b]$ se $\#[a, b] = 0$, isto é se $a > b$. A função retorna corretamente -1.
- Hereditariedade: Assumimos, como hipótese, que para $\#[a, b]$ fixo, sendo a e b quaisquer $0 \leq a \leq b < n$, a função retorna o valor correto para todos os $[a', b']$, com $\#[a', b'] < \#[a, b]$, com $0 \leq a' \leq b' < n$ ou $a' > b'$. Vamos mostrar que então também dá o valor correto para $[a, b]$.

Exemplos: Pesquisa binária (*binary search*)

```
PesquisaBinaria(v, a, b, x)
  Enquanto (a ≤ b) fazer
     m ← ⌊(a + b)/2⌋;
     Se (v[m] = x) então
        retorna m;
     Se (v[m] > x) então
        a ← m + 1;
     senão b ← m − 1;
  retorna −1;
```

```
PesquisaBinariaRec(v, a, b, x)
  Se (a > b) então
     retorna −1;
  m ← ⌊(a + b)/2⌋;
  Se (v[m] = x) então
     retorna m;
  Se (v[m] > x) então
     retorna PesquisaBinariaRec(v, m + 1, b, x);
  retorna PesquisaBinariaRec(v, a, m − 1, x);
```

Prova de correção de PesquisaBinariaRec (por indução forte sobre o número de elementos do intervalo)

- Caso de base: x não se encontra no segmento definido por $[a, b]$ se $\#[a, b] = 0$, isto é se $a > b$. A função retorna corretamente -1.
- Hereditariedade: Assumimos, como hipótese, que para $\#[a, b]$ fixo, sendo a e b quaisquer $0 \leq a \leq b < n$, a função retorna o valor correto para todos os $[a', b']$, com $\#[a', b'] < \#[a, b]$, com $0 \leq a' \leq b' < n$ ou $a' > b'$. Vamos mostrar que então também dá o valor correto para $[a, b]$.

Exemplos: Pesquisa binária (*binary search*)

```
PesquisaBinaria(v, a, b, x)
  Enquanto (a ≤ b) fazer
     m ← ⌊(a + b)/2⌋;
     Se (v[m] = x) então
        retorna m;
     Se (v[m] > x) então
        a ← m + 1;
     senão b ← m − 1;
  retorna −1;
```

```
PesquisaBinariaRec(v, a, b, x)
  Se (a > b) então
     retorna −1;
  m ← ⌊(a + b)/2⌋;
  Se (v[m] = x) então
     retorna m;
  Se (v[m] > x) então
     retorna PesquisaBinariaRec(v, m + 1, b, x);
  retorna PesquisaBinariaRec(v, a, m − 1, x);
```

Prova de correção de PesquisaBinariaRec (por indução forte sobre o número de elementos do intervalo)

- Caso de base: x não se encontra no segmento definido por $[a, b]$ se $\#[a, b] = 0$, isto é se $a > b$. A função retorna corretamente -1.
- Hereditariedade: Assumimos, como hipótese, que para $\#[a, b]$ fixo, sendo a e b quaisquer $0 \leq a \leq b < n$, a função retorna o valor correto para todos os $[a', b']$, com $\#[a', b'] < \#[a, b]$, com $0 \leq a' \leq b' < n$ ou $a' > b'$. Vamos mostrar que então também dá o valor correto para $[a, b]$.

Exemplos: Pesquisa binária (*binary search*)

```
PesquisaBinaria(v, a, b, x)
  Enquanto (a ≤ b) fazer
     m ← ⌊(a + b)/2⌋;
     Se (v[m] = x) então
       retorna m;
     Se (v[m] > x) então
       a ← m + 1;
     senão b ← m − 1;
  retorna −1;
```

```
PesquisaBinariaRec(v, a, b, x)
  Se (a > b) então
     retorna −1;
  m ← ⌊(a + b)/2⌋;
  Se (v[m] = x) então
     retorna m;
  Se (v[m] > x) então
     retorna PesquisaBinariaRec(v, m + 1, b, x);
  retorna PesquisaBinariaRec(v, a, m − 1, x);
```

Prova de correção de PesquisaBinariaRec (por indução forte sobre o número de elementos do intervalo)

- Caso de base: x não se encontra no segmento definido por $[a, b]$ se $\#[a, b] = 0$, isto é se $a > b$. A função retorna corretamente -1.
- Hereditariedade: Assumimos, como hipótese, que para $\#[a, b]$ fixo, sendo a e b quaisquer $0 \leq a \leq b < n$, a função retorna o valor correto para todos os $[a', b']$, com $\#[a', b'] < \#[a, b]$, com $0 \leq a' \leq b' < n$ ou $a' > b'$. Vamos mostrar que então também dá o valor correto para $[a, b]$.

Exemplos: Pesquisa binária (*binary search*)

```
PesquisaBinaria(v, a, b, x)
  Enquanto (a ≤ b) fazer
     m ← ⌊(a + b)/2⌋;
     Se (v[m] = x) então
       retorna m;
     Se (v[m] > x) então
       a ← m + 1;
     senão b ← m − 1;
  retorna −1;
```

```
PesquisaBinariaRec(v, a, b, x)
  Se (a > b) então
     retorna −1;
  m ← ⌊(a + b)/2⌋;
  Se (v[m] = x) então
     retorna m;
  Se (v[m] > x) então
     retorna PesquisaBinariaRec(v, m + 1, b, x);
  retorna PesquisaBinariaRec(v, a, m − 1, x);
```

Prova de correção de PesquisaBinariaRec (por indução forte sobre o número de elementos do intervalo)

- Caso de base: x não se encontra no segmento definido por $[a, b]$ se $\#[a, b] = 0$, isto é se $a > b$. A função retorna corretamente -1.
- Hereditariedade: Assumimos, como hipótese, que para $\#[a, b]$ fixo, sendo a e b quaisquer $0 \leq a \leq b < n$, a função retorna o valor correto para todos os $[a', b']$, com $\#[a', b'] < \#[a, b]$, com $0 \leq a' \leq b' < n$ ou $a' > b'$. Vamos mostrar que então também dá o valor correto para $[a, b]$.

Exemplos: Pesquisa binária (*binary search*)

```
PesquisaBinariaRec(v, a, b, x)
1. | Se (a > b) então
2. |    retorna −1;
3. | m ← ⌊(a + b)/2⌋;
4. | Se (v[m] = x) então
5. |    retorna m;
6. | Se (v[m] > x) então
7. |    retorna PesquisaBinariaRec(v, m + 1, b, x);
8. | retorna PesquisaBinariaRec(v, a, m − 1, x);
```

Hereditariedade:

Assumimos, como hipótese, que para $\#[a, b]$ fixo, sendo a e b quaisquer $0 \leq a \leq b < n$, a função retorna o valor correto para todos os $[a', b']$, com $\#[a', b'] < \#[a, b]$, com $0 \leq a' \leq b' < n$ ou $a' > b'$.

- Como $a \leq b$, executa as linhas 1, 3 e 4. Como m fica com $\lfloor (a+b)/2 \rfloor$, e se tem $m \in [a, b]$, é correto retornar m na linha 5, se $v[m] = x$.
- Se $v[m] \neq x$, executa 6, e se $v[m] > x$, sabemos que x só pode estar no segmento $[m+1, b]$ (por v estar por ordem decrescente). Nesse caso, a chamada $\text{PesquisaBinariaRec}(v, m+1, b, x)$, em que $\#[m+1, b] < \#[a, b]$, dará a resposta correta (por hipótese).
- Se na linha 6 se tem $v[m] < x$, passa à linha 8. Como x teria de ocorrer no segmento $[a, m-1]$ (se ocorrer), $\text{PesquisaBinariaRec}(v, m+1, b, x)$, em que $\#[a, m-1] < \#[a, b]$, dará a resposta correta (por hipótese).

Exemplos: Pesquisa binária (*binary search*)

```
PesquisaBinariaRec(v, a, b, x)
1.  Se (a > b) então
2.     retorna −1;
3.  m ← ⌊(a + b)/2⌋;
4.  Se (v[m] = x) então
5.     retorna m;
6.  Se (v[m] > x) então
7.     retorna PesquisaBinariaRec(v, m + 1, b, x);
8.  retorna PesquisaBinariaRec(v, a, m − 1, x);
```

Hereditariedade:

Assumimos, como hipótese, que para $\#[a, b]$ fixo, sendo a e b quaisquer $0 \leq a \leq b < n$, a função retorna o valor correto para todos os $[a', b']$, com $\#[a', b'] < \#[a, b]$, com $0 \leq a' \leq b' < n$ ou $a' > b'$.

- Como $a \leq b$, executa as linhas 1, 3 e 4. Como m fica com $\lfloor (a+b)/2 \rfloor$, e se tem $m \in [a, b]$, é correto retornar m na linha 5, se $v[m] = x$.
- Se $v[m] \neq x$, executa 6, e se $v[m] > x$, sabemos que x só pode estar no segmento $[m+1, b]$ (por v estar por ordem decrescente). Nesse caso, a chamada PesquisaBinariaRec$(v, m+1, b, x)$, em que $\#[m+1, b] < \#[a, b]$, dará a resposta correta (por hipótese).
- Se na linha 6 se tem $v[m] < x$, passa à linha 8. Como x teria de ocorrer no segmento $[a, m-1]$ (se ocorrer), PesquisaBinariaRec$(v, m+1, b, x)$, em que $\#[a, m-1] < \#[a, b]$, dará a resposta correta (por hipótese).

Exemplos: Pesquisa binária (*binary search*)

```
PesquisaBinariaRec(v, a, b, x)
1.  Se (a > b) então
2.      retorna −1;
3.  m ← ⌊(a + b)/2⌋;
4.  Se (v[m] = x) então
5.      retorna m;
6.  Se (v[m] > x) então
7.      retorna PesquisaBinariaRec(v, m + 1, b, x);
8.  retorna PesquisaBinariaRec(v, a, m − 1, x);
```

Hereditariedade:

Assumimos, como hipótese, que para $\#[a, b]$ fixo, sendo a e b quaisquer $0 \leq a \leq b < n$, a função retorna o valor correto para todos os $[a', b']$, com $\#[a', b'] < \#[a, b]$, com $0 \leq a' \leq b' < n$ ou $a' > b'$.

- Como $a \leq b$, executa as linhas 1, 3 e 4. Como m fica com $\lfloor (a+b)/2 \rfloor$, e se tem $m \in [a, b]$, é correto retornar m na linha 5, se $v[m] = x$.
- Se $v[m] \neq x$, executa 6, e se $v[m] > x$, sabemos que x só pode estar no segmento $[m + 1, b]$ (por v estar por ordem decrescente). Nesse caso, a chamada PesquisaBinariaRec$(v, m + 1, b, x)$, em que $\#[m + 1, b] < \#[a, b]$, dará a resposta correta (por hipótese).
- Se na linha 6 se tem $v[m] < x$, passa à linha 8. Como x teria de ocorrer no segmento $[a, m - 1]$ (se ocorrer), PesquisaBinariaRec$(v, m + 1, b, x)$, em que $\#[a, m - 1] < \#[a, b]$, dará a resposta correta (por hipótese).

Exemplos: Pesquisa binária (*binary search*)

```
PesquisaBinariaRec(v, a, b, x)
1.  Se (a > b) então
2.      retorna −1;
3.  m ← ⌊(a + b)/2⌋;
4.  Se (v[m] = x) então
5.      retorna m;
6.  Se (v[m] > x) então
7.      retorna PesquisaBinariaRec(v, m + 1, b, x);
8.  retorna PesquisaBinariaRec(v, a, m − 1, x);
```

Hereditariedade:

Assumimos, como hipótese, que para $\#[a, b]$ fixo, sendo a e b quaisquer $0 \leq a \leq b < n$, a função retorna o valor correto para todos os $[a', b']$, com $\#[a', b'] < \#[a, b]$, com $0 \leq a' \leq b' < n$ ou $a' > b'$.

- Como $a \leq b$, executa as linhas 1, 3 e 4. Como m fica com $\lfloor (a+b)/2 \rfloor$, e se tem $m \in [a, b]$, é correto retornar m na linha 5, se $v[m] = x$.
- Se $v[m] \neq x$, executa 6, e se $v[m] > x$, sabemos que x só pode estar no segmento $[m+1, b]$ (por v estar por ordem decrescente). Nesse caso, a chamada PesquisaBinariaRec$(v, m+1, b, x)$, em que $\#[m+1, b] < \#[a, b]$, dará a resposta correta (por hipótese).
- Se na linha 6 se tem $v[m] < x$, passa à linha 8. Como x teria de ocorrer no segmento $[a, m-1]$ (se ocorrer), PesquisaBinariaRec$(v, m+1, b, x)$, em que $\#[a, m-1] < \#[a, b]$, dará a resposta correta (por hipótese).

Exemplos: Pesquisa binária (*binary search*)

```
PesquisaBinariaRec(v, a, b, x)
1. | Se (a > b) então
2. |    retorna −1;
3. | m ← ⌊(a + b)/2⌋;
4. | Se (v[m] = x) então
5. |    retorna m;
6. | Se (v[m] > x) então
7. |    retorna PesquisaBinariaRec(v, m + 1, b, x);
8. | retorna PesquisaBinariaRec(v, a, m − 1, x);
```

Hereditariedade:

Assumimos, como hipótese, que para $\#[a, b]$ fixo, sendo a e b quaisquer $0 \leq a \leq b < n$, a função retorna o valor correto para todos os $[a', b']$, com $\#[a', b'] < \#[a, b]$, com $0 \leq a' \leq b' < n$ ou $a' > b'$.

- Como $a \leq b$, executa as linhas 1, 3 e 4. Como m fica com $\lfloor (a+b)/2 \rfloor$, e se tem $m \in [a, b]$, é correto retornar m na linha 5, se $v[m] = x$.
- Se $v[m] \neq x$, executa 6, e se $v[m] > x$, sabemos que x só pode estar no segmento $[m+1, b]$ (por v estar por ordem decrescente). Nesse caso, a chamada $\textsc{PesquisaBinariaRec}(v, m+1, b, x)$, em que $\#[m+1, b] < \#[a, b]$, dará a resposta correta (por hipótese).
- Se na linha 6 se tem $v[m] < x$, passa à linha 8. Como x teria de ocorrer no segmento $[a, m-1]$ (se ocorrer), $\textsc{PesquisaBinariaRec}(v, m+1, b, x)$, em que $\#[a, m-1] < \#[a, b]$, dará a resposta correta (por hipótese).

Exemplos: Pesquisa binária (*binary search*)

$\textsc{ProcuraBinaria}(v, n, x)$	*Caso* $x \notin v[\cdot]$
$a \leftarrow 0;$	c_1
$b \leftarrow n-1;$	c_2
Enquanto $(a \leq b)$ fazer	$c_3 \times (k_1 + k_2 + 1)$
$m \leftarrow \lfloor (a+b)/2 \rfloor;$	$c_4 \times (k_1 + k_2)$
Se $(v[m] = x)$ então	$c_5 \times (k_1 + k_2)$
retorna m;	$c_6 \times 0$
Se $(v[m] > x)$ então	$c_5 \times (k_1 + k_2)$
$a \leftarrow m+1;$	$c_6 k_1$
senão	
$b \leftarrow m-1;$	$c_6 k_2$
retorna -1;	c_7

- Neste caso $|I_1| = n$ e, para $k \geq 2$, quando testa a condição de paragem do ciclo enquanto pela k-ésima vez, tem-se $|I_k| \leq \frac{1}{2}|I_{k-1}|$.
- Logo, $|I_k| \leq \frac{n}{2^{k-1}}$. Se $k-1 > \log_2(n)$ então $|I_k| = 0$, i..e,, $a > b$ e o ciclo pára.
- Cada iteração tem complexidade $\Theta(1)$. No pior caso., a função é $\Theta(\log_2(n))$. No melhor caso, se $x = v[m]$ na primeira iteração, a complexidade é $\Theta(1)$.
- Conclusão: a complexidade de $\textsc{ProcuraBinaria}(v, n, x)$ é $O(\log_2(n))$.

Exemplos: Merge sort

$\textsc{MergeSort}(v, a, b)$
- Se $(a < b)$ então
  - $m \leftarrow \lfloor (a+b)/2 \rfloor$;
  - $\textsc{MergeSort}(v, a, m)$;
  - $\textsc{MergeSort}(v, m+1, b)$;
  - $\textsc{Merge}(v, a, m, b)$;

$\textsc{Merge}(v, a, m, b)$
- $i \leftarrow a$; $j \leftarrow m+1$; $k \leftarrow 0$;
- Enquanto $(i \leq m \wedge j \leq b)$ fazer
  - Se $v[i] < v[j]$ então $\{aux[k] \leftarrow v[i];\ i \leftarrow i+1;\}$
  - senão $\{aux[k] \leftarrow v[j];\ j \leftarrow j+1;\}$
  - $k \leftarrow k+1$;
- Enquanto $(i \leq m)$ fazer
  - $aux[k] \leftarrow v[i]$;
  - $i \leftarrow i+1$; $k \leftarrow k+1$;
- Para $i \leftarrow 0$ até $k-1$ fazer
  - $v[i+a] \leftarrow aux[i]$;

Propriedade importante para a correção:
Os dois últimos ciclos de $\textsc{Merge}(v, a, m, b)$, não referem as posições a partir de j até b, porque se existirem, já estão corretamente colocadas em v.

Complexidade temporal: sendo $N = b - a + 1$ o número de elementos a ordenar

- $\textsc{Merge}(v, a, m, b)$ tem complexidade $O(N)$
- $\textsc{MergeSort}(v, a, b)$ tem complexidade $\Theta(N \log_2 N)$.

Exemplos: Merge sort

$\textsc{MergeSort}(v, a, b)$
- Se $(a < b)$ então
  - $m \leftarrow \lfloor (a+b)/2 \rfloor$;
  - $\textsc{MergeSort}(v, a, m)$;
  - $\textsc{MergeSort}(v, m+1, b)$;
  - $\textsc{Merge}(v, a, m, b)$;

$\textsc{Merge}(v, a, m, b)$
- $i \leftarrow a$; $j \leftarrow m+1$; $k \leftarrow 0$;
- Enquanto $(i \leq m \wedge j \leq b)$ fazer
  - Se $v[i] < v[j]$ então $\{aux[k] \leftarrow v[i];\ i \leftarrow i+1;\}$
  - senão $\{aux[k] \leftarrow v[j];\ j \leftarrow j+1;\}$
  - $k \leftarrow k+1$;
- Enquanto $(i \leq m)$ fazer
  - $aux[k] \leftarrow v[i]$;
  - $i \leftarrow i+1$; $k \leftarrow k+1$;
- Para $i \leftarrow 0$ até $k-1$ fazer
  - $v[i+a] \leftarrow aux[i]$;

Propriedade importante para a correção:
Os dois últimos ciclos de $\textsc{Merge}(v, a, m, b)$, não referem as posições a partir de j até b, porque se existirem, já estão corretamente colocadas em v.

Complexidade temporal: sendo $N = b - a + 1$ o número de elementos a ordenar

- $\textsc{Merge}(v, a, m, b)$ tem complexidade $O(N)$
- $\textsc{MergeSort}(v, a, b)$ tem complexidade $\Theta(N \log_2 N)$.

Exemplos: Merge sort

$\text{MergeSort}(v, a, b)$
  Se $(a < b)$ então
    $m \leftarrow \lfloor (a+b)/2 \rfloor$;
    $\text{MergeSort}(v, a, m)$;
    $\text{MergeSort}(v, m+1, b)$;
    $\text{Merge}(v, a, m, b)$;

$\text{Merge}(v, a, m, b)$
  $i \leftarrow a$; $j \leftarrow m+1$; $k \leftarrow 0$;
  Enquanto $(i \leq m \land j \leq b)$ fazer
    Se $v[i] < v[j]$ então $\{aux[k] \leftarrow v[i];\ i \leftarrow i+1;\}$
    senão $\{aux[k] \leftarrow v[j];\ j \leftarrow j+1;\}$
    $k \leftarrow k+1$;
  Enquanto $(i \leq m)$ fazer
    $aux[k] \leftarrow v[i]$;
    $i \leftarrow i+1$; $k \leftarrow k+1$;
  Para $i \leftarrow 0$ até $k-1$ fazer
    $v[i+a] \leftarrow aux[i]$;

Propriedade importante para a correção:
Os dois últimos ciclos de $\text{Merge}(v, a, m, b)$, não referem as posições a partir de j até b, porque se existirem, já estão corretamente colocadas em v.

Complexidade temporal: sendo $N = b - a + 1$ o número de elementos a ordenar

- $\text{Merge}(v, a, m, b)$ tem complexidade $O(N)$
- $\text{MergeSort}(v, a, b)$ tem complexidade $\Theta(N \log_2 N)$.

Exemplos: Merge sort

$\textsc{MergeSort}(v, a, b)$
- Se $(a < b)$ então
  - $m \leftarrow \lfloor (a+b)/2 \rfloor$;
  - $\textsc{MergeSort}(v, a, m)$;
  - $\textsc{MergeSort}(v, m+1, b)$;
  - $\textsc{Merge}(v, a, m, b)$;

$\textsc{Merge}(v, a, m, b)$
- $i \leftarrow a;\quad j \leftarrow m+1;\quad k \leftarrow 0;$
- Enquanto $(i \leq m \wedge j \leq b)$ fazer
  - Se $v[i] < v[j]$ então $\{aux[k] \leftarrow v[i];\ \ i \leftarrow i+1;\}$
  - senão $\{aux[k] \leftarrow v[j];\ \ j \leftarrow j+1;\}$
  - $k \leftarrow k+1;$
- Enquanto $(i \leq m)$ fazer
  - $aux[k] \leftarrow v[i];$
  - $i \leftarrow i+1;\ \ k \leftarrow k+1;$
- Para $i \leftarrow 0$ até $k-1$ fazer
  - $v[i+a] \leftarrow aux[i];$

Propriedade importante para a correção:
Os dois últimos ciclos de $\textsc{Merge}(v, a, m, b)$, não referem as posições a partir de j até b, porque se existirem, já estão corretamente colocadas em v.

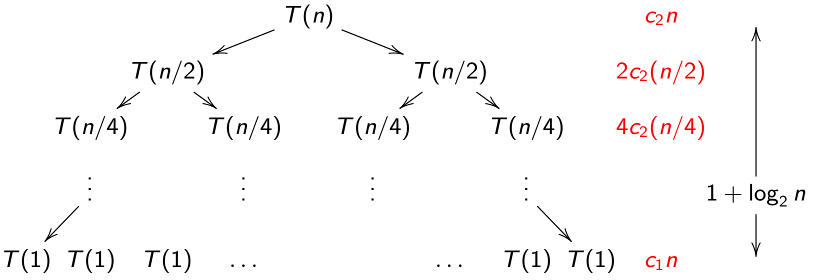
Complexidade temporal: sendo $N = b - a + 1$ o número de elementos a ordenar

- $\textsc{Merge}(v, a, m, b)$ tem complexidade $O(N)$
- $\textsc{MergeSort}(v, a, b)$ tem complexidade $\Theta(N \log_2 N)$.

Exemplos: Merge sort

A complexidade temporal de MergeSort pode ser definida pela recorrência $T(1) = c_1$ e $T(n) \leq 2T(\lceil n/2 \rceil) + c_2 n$, para $n \geq 2$.

- Se n é potência de 2, o número de níveis da árvore de recursão é $1 + \log_2 n$ e a complexidade de cada nível é $\Theta(n)$. Logo, $T(n) \in \Theta(n \log_2 n)$.

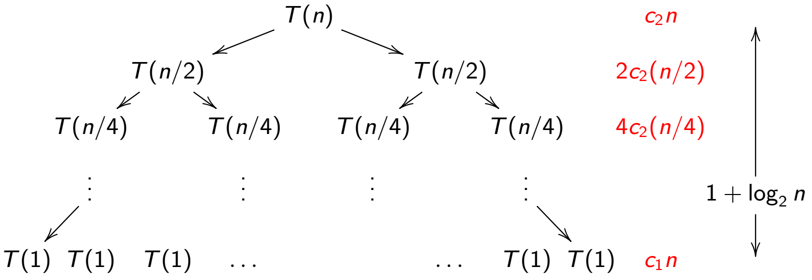


Algoritmos baseados em comparação (como *mergesort*, *insertion sort*, *selection sort*, *quicksort*, *heapsort*) requerem $\Omega(n \log_2 n)$, no pior caso. Apenas *Mergesort* e *heapsort* são assintoticamente ótimos (e *heapsort* é "in-place").

Exemplos: Merge sort

A complexidade temporal de MERGESORT pode ser definida pela recorrência $T(1) = c_1$ e $T(n) \leq 2T(\lceil n/2 \rceil) + c_2 n$, para $n \geq 2$.

- Se n é potência de 2, o número de níveis da árvore de recursão é $1 + \log_2 n$ e a complexidade de cada nível é $\Theta(n)$. Logo, $T(n) \in \Theta(n \log_2 n)$.



Algoritmos baseados em comparação (como *mergesort*, *insertion sort*, *selection sort*, *quicksort*, *heapsort*) requerem $\Omega(n \log_2 n)$, no pior caso. Apenas *Mergesort* e *heapsort* são assintoticamente <u>ótimos</u> (e *heapsort* é "in-place").

Par de pontos a distância mínima numa reta e no plano

Problema: Determinar um par de pontos a distância mínima num conjunto P de $n > 1$ pontos, para $P \subset \mathbb{R}$ ou $P \subset \mathbb{R}^2$.

- **Resolve-se trivialmente em $\Theta(n^2)$, por força bruta.**

ParMaisProximo_Trivial(p, n, par)
 $distmin \leftarrow$ QDist$(p, 0, 1)$; $par[0] \leftarrow 0$; $par[1] \leftarrow 1$;
 Para $i \leftarrow 0$ até $n-2$ fazer
  Para $j \leftarrow i+1$ até $n-1$ fazer
   $aux \leftarrow$ QDist(p, i, j);
   Se $(aux < distmin)$ então
    $distmin \leftarrow aux$; $par[0] \leftarrow i$; $par[1] \leftarrow j$;

Em $\mathbb{R}^2$, deve evitar o cálculo de raízes quadradas: QDist(p, i, j) retorna $(x_i - x_j)^2 + (y_i - y_j)^2$, o **quadrado da distância euclideana entre p_i e p_j**.

- **Para pontos numa reta, o par mais próximo pode ser obtido em $O(n \log_2 n)$.**

Os pontos a distância mínima ocorrem em posições consecutivas na reta.

Par de pontos a distância mínima numa reta e no plano

Problema: Determinar um par de pontos a distância mínima num conjunto P de $n > 1$ pontos, para $P \subset \mathbb{R}$ ou $P \subset \mathbb{R}^2$.

- **Resolve-se trivialmente em** $\Theta(n^2)$, por **força bruta**.

ParMaisProximo_Trivial(p, n, par)
  $distmin \leftarrow$ QDist($p, 0, 1$); $par[0] \leftarrow 0$; $par[1] \leftarrow 1$;
  Para $i \leftarrow 0$ até $n-2$ fazer
    Para $j \leftarrow i+1$ até $n-1$ fazer
      $aux \leftarrow$ QDist(p, i, j);
      Se ($aux < distmin$) então
        $distmin \leftarrow aux$; $par[0] \leftarrow i$; $par[1] \leftarrow j$;

Em $\mathbb{R}^2$, deve evitar o cálculo de raízes quadradas: QDist(p, i, j) retorna $(x_i - x_j)^2 + (y_i - y_j)^2$, o **quadrado da distância euclideana entre** p_i **e** p_j.

- **Para pontos numa reta, o par mais próximo pode ser obtido em** $O(n \log_2 n)$.

Os pontos a distância mínima ocorrem em posições consecutivas na reta.

Par de pontos a distância mínima numa reta e no plano

Problema: Determinar um par de pontos a distância mínima num conjunto P de $n > 1$ pontos, para $P \subset \mathbb{R}$ ou $P \subset \mathbb{R}^2$.

- **Resolve-se trivialmente em $\Theta(n^2)$**, por **força bruta**.

```
ParMaisProximo_Trivial(p, n, par)
| distmin ← QDist(p, 0, 1);  par[0] ← 0;  par[1] ← 1;
| Para i ← 0 até n − 2 fazer
|    Para j ← i + 1 até n − 1 fazer
|       aux ← QDist(p, i, j);
|       Se (aux < distmin) então
|          distmin ← aux;  par[0] ← i;  par[1] ← j;
```

Em $\mathbb{R}^2$, deve evitar o cálculo de raízes quadradas: $\textsc{QDist}(p, i, j)$ retorna $(x_i - x_j)^2 + (y_i - y_j)^2$, o **quadrado da distância euclideana entre p_i e p_j.**

- **Para pontos numa reta, o par mais próximo pode ser obtido em $O(n \log_2 n)$.**

Os pontos a distância mínima ocorrem em posições consecutivas na reta.

Par de pontos a distância mínima numa reta e no plano

Problema: Determinar um par de pontos a distância mínima num conjunto P de $n > 1$ pontos, para $P \subset \mathbb{R}$ ou $P \subset \mathbb{R}^2$.

- **Resolve-se trivialmente em $\Theta(n^2)$**, por **força bruta**.

```
ParMaisProximo_Trivial(p, n, par)
  distmin ← QDist(p, 0, 1);  par[0] ← 0;  par[1] ← 1;
  Para i ← 0 até n − 2 fazer
     Para j ← i + 1 até n − 1 fazer
        aux ← QDist(p, i, j);
        Se (aux < distmin) então
           distmin ← aux;  par[0] ← i;  par[1] ← j;
```

Em $\mathbb{R}^2$, deve evitar o cálculo de raízes quadradas: $\mathrm{QDist}(p, i, j)$ retorna $(x_i - x_j)^2 + (y_i - y_j)^2$, o **quadrado da distância euclideana entre p_i e p_j.**

- **Para pontos numa reta, o par mais próximo pode ser obtido em $O(n \log_2 n)$.**

Os pontos a distância mínima ocorrem em posições consecutivas na reta.

Par de pontos a distância mínima numa reta e no plano

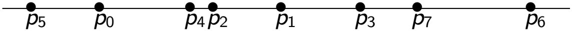
Problema: Determinar um par de pontos a distância mínima num conjunto P de $n > 1$ pontos, para $P \subset \mathbb{R}$ ou $P \subset \mathbb{R}^2$.

- **Resolve-se trivialmente em** $\Theta(n^2)$, por **força bruta**.

```
ParMaisProximo_Trivial(p, n, par)
  distmin ← QDist(p, 0, 1);  par[0] ← 0;  par[1] ← 1;
  Para i ← 0 até n − 2 fazer
     Para j ← i + 1 até n − 1 fazer
        aux ← QDist(p, i, j);
        Se (aux < distmin) então
           distmin ← aux;  par[0] ← i;  par[1] ← j;
```

Em $\mathbb{R}^2$, deve evitar o cálculo de raízes quadradas: $\mathrm{QDist}(p, i, j)$ retorna $(x_i - x_j)^2 + (y_i - y_j)^2$, o **quadrado da distância euclideana entre** p_i **e** p_j.

- **Para pontos numa** **reta**, **o par mais próximo pode ser obtido em** $O(n \log_2 n)$.



Os pontos a distância mínima ocorrem em posições consecutivas na reta.

Par de pontos a distância mínima em $\mathbb{R}$ em $O(n \log_2 n)$

$\textsc{ParMaisProximo\_Recta}(p, n, par)$
- $\textsc{Ordenar}(p, n, pos)$;
- $distmin \leftarrow \textsc{QDist}(p, pos[0], pos[1])$;
- $par[0] \leftarrow pos[0]$; $par[1] \leftarrow pos[1]$;
- Para $i \leftarrow 2$ até $n - 1$ fazer
  - $aux \leftarrow \textsc{QDist}(p, pos[i-1], pos[i])$;
  - Se $(aux < distmin)$ então
    - $distmin \leftarrow aux$;
    - $par[0] \leftarrow pos[i-1]$; $par[1] \leftarrow pos[i]$;

- $\textsc{Ordenar}(p, n, pos)$ deve ter complexidade $O(n \log_2 n)$ e não deve alterar as posições dos elementos de p mas produzir o vetor pos tal que $pos[i]$ indica o índice do elemento que estaria na posição i na sequência ordenada. Por exemplo, para $p = [5, -4, 7, 3, 9, -1, 13, 18]$, obteria $pos = [1, 5, 3, 0, 2, 4, 6, 7]$.
- Não pode ser estendido para $P \subset \mathbb{R}^2$, mas podemos usar um método alternativo – **método de Shamos** (1975) – baseado em "divisão e conquista".

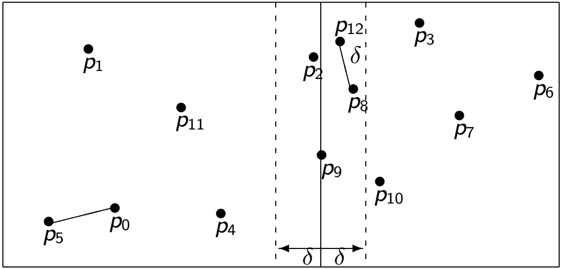
Par de pontos a distância mínima em $\mathbb{R}$ em $O(n \log_2 n)$

ParMaisProximo_Recta(p, n, par)
  Ordenar(p, n, pos);
  $distmin \leftarrow$ QDist($p, pos[0], pos[1]$);
  $par[0] \leftarrow pos[0]$; $par[1] \leftarrow pos[1]$;
  Para $i \leftarrow 2$ até $n-1$ fazer
    $aux \leftarrow$ QDist($p, pos[i-1], pos[i]$);
    Se ($aux < distmin$) então
      $distmin \leftarrow aux$;
      $par[0] \leftarrow pos[i-1]$; $par[1] \leftarrow pos[i]$;

- Ordenar(p, n, pos) deve ter complexidade $O(n \log_2 n)$ e não deve alterar as posições dos elementos de p mas produzir o vetor pos tal que $pos[i]$ indica o índice do elemento que estaria na posição i na sequência ordenada. Por exemplo, para $p = [5, -4, 7, 3, 9, -1, 13, 18]$, obteria $pos = [1, 5, 3, 0, 2, 4, 6, 7]$.
- Não pode ser estendido para $P \subset \mathbb{R}^2$, mas podemos usar um método alternativo – **método de Shamos** (1975) – baseado em "divisão e conquista".

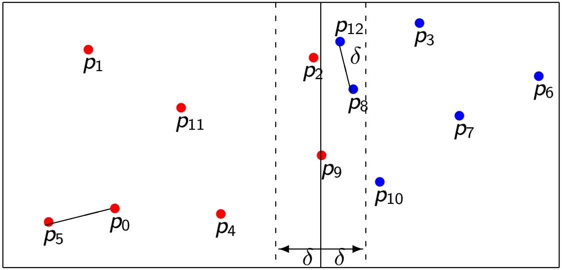
Método de Shamos - Par a distância mínima no plano

Explora propriedades geométricas



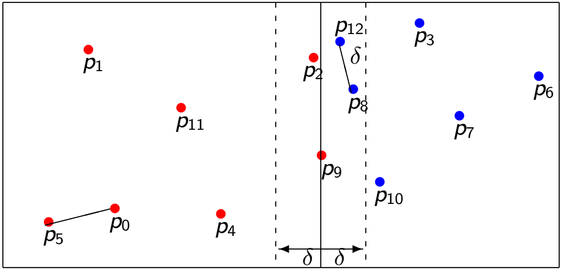
- Para facilitar a descrição, assumimos $\mathcal{P}$ em *posição geral* o que, neste caso, significa que não há dois pontos com a mesma abcissa nem com a mesma ordenada.
- Numa fase de pré-processamento, ordena os pontos por ordem crescente de abcissa e por ordem decrescente de ordenada
- A ordenação deve ser realizada em $O(2n\log_2 n) = O(n\log_2 n)$ e será efetuada apenas uma vez, antes de iniciar o processo recursivo.

Método de Shamos - Par a distância mínima no plano



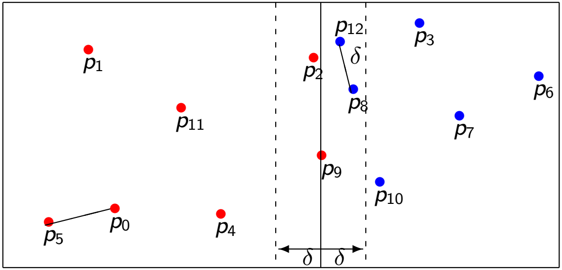
- **Parte $\mathcal{P}$ em dois conjuntos similares** $\mathcal{P}_1$ e $\mathcal{P}_2$, usando a mediana das abcissas (reta $x = x_m$ que parte $\mathcal{P}$ ao meio aproximadamente)
- Cada conjunto fica com $n/2$ pontos, se n for par, e com $\lceil n/2 \rceil$ ou $\lfloor n/2 \rfloor$, se for ímpar. x_m é a abcissa do ponto mais à direita em $\mathcal{P}_1$ (no exemplo, p_9).
- **Recursivamente, obtém o par mais próximo em** $\mathcal{P}_1$ **e em** $\mathcal{P}_2$. Sejam δ_1 e δ_2 as distâncias mínimas encontradas para $\mathcal{P}_1$ e $\mathcal{P}_2$.
- **Resta comparar pontos de** $\mathcal{P}_1$ **com** $\mathcal{P}_2$. Seja $\delta = \min(\delta_1, \delta_2)$. Analisa apenas a **faixa** $]x_m - \delta, x_m + \delta[$. Fora desta faixa, a distância seria pelo menos δ. Filtra $\mathcal{P}$ para obter os pontos que estão na faixa.

Método de Shamos - Par a distância mínima no plano



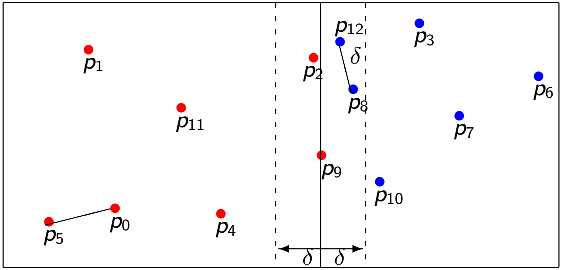
- **Parte $\mathcal{P}$ em dois conjuntos similares** $\mathcal{P}_1$ e $\mathcal{P}_2$, usando a mediana das abcissas (reta $x = x_m$ que parte $\mathcal{P}$ ao meio aproximadamente)
- Cada conjunto fica com $n/2$ pontos, se n for par, e com $\lceil n/2 \rceil$ ou $\lfloor n/2 \rfloor$, se for ímpar. x_m é a abcissa do ponto mais à direita em $\mathcal{P}_1$ (no exemplo, p_9).
- **Recursivamente, obtém o par mais próximo em** $\mathcal{P}_1$ **e em** $\mathcal{P}_2$. Sejam δ_1 e δ_2 as distâncias mínimas encontradas para $\mathcal{P}_1$ e $\mathcal{P}_2$.
- **Resta comparar pontos de** $\mathcal{P}_1$ **com** $\mathcal{P}_2$. Seja $\delta = \min(\delta_1, \delta_2)$. Analisa apenas a **faixa** $]x_m - \delta, x_m + \delta[$. Fora desta faixa, a distância seria pelo menos δ. Filtra $\mathcal{P}$ para obter os pontos que estão na faixa.

Método de Shamos - Par a distância mínima no plano



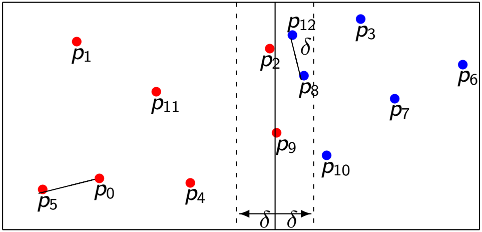
- **Parte $\mathcal{P}$ em dois conjuntos similares** $\mathcal{P}_1$ e $\mathcal{P}_2$, usando a mediana das abcissas (reta $x = x_m$ que parte $\mathcal{P}$ ao meio aproximadamente)
- Cada conjunto fica com $n/2$ pontos, se n for par, e com $\lceil n/2 \rceil$ ou $\lfloor n/2 \rfloor$, se for ímpar. x_m é a abcissa do ponto mais à direita em $\mathcal{P}_1$ (no exemplo, p_9).
- **Recursivamente, obtém o par mais próximo em $\mathcal{P}_1$ e em $\mathcal{P}_2$.** Sejam δ_1 e δ_2 as distâncias mínimas encontradas para $\mathcal{P}_1$ e $\mathcal{P}_2$.
- **Resta comparar pontos de $\mathcal{P}_1$ com $\mathcal{P}_2$.** Seja $\delta = \min(\delta_1, \delta_2)$. Analisa apenas a **faixa** $]x_m - \delta, x_m + \delta[$. Fora desta faixa, a distância seria pelo menos δ. Filtra $\mathcal{P}$ para obter os pontos que estão na faixa.

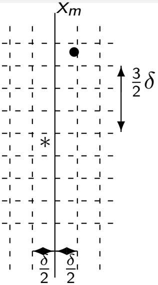
Método de Shamos - Par a distância mínima no plano



- **Parte $\mathcal{P}$ em dois conjuntos similares** $\mathcal{P}_1$ e $\mathcal{P}_2$, usando a mediana das abcissas (reta $x = x_m$ que parte $\mathcal{P}$ ao meio aproximadamente)
- Cada conjunto fica com $n/2$ pontos, se n for par, e com $\lceil n/2 \rceil$ ou $\lfloor n/2 \rfloor$, se for ímpar. x_m é a abcissa do ponto mais à direita em $\mathcal{P}_1$ (no exemplo, p_9).
- **Recursivamente, obtém o par mais próximo em $\mathcal{P}_1$ e em $\mathcal{P}_2$.** Sejam δ_1 e δ_2 as distâncias mínimas encontradas para $\mathcal{P}_1$ e $\mathcal{P}_2$.
- **Resta comparar pontos de $\mathcal{P}_1$ com $\mathcal{P}_2$.** Seja $\delta = \min(\delta_1, \delta_2)$. Analisa apenas a **faixa** $]x_m - \delta, x_m + \delta[$. Fora desta faixa, a distância seria pelo menos δ. Filtra $\mathcal{P}$ para obter os pontos que estão na faixa.

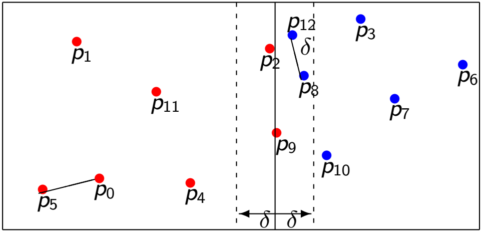
Método de Shamos - Par a distância mínima no plano

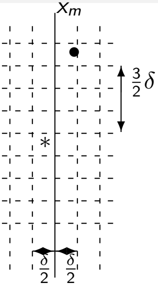




- A fase de combinação realiza-se em $O(n)$: se tomarmos os pontos na faixa por ordem decrescente de ordenada, **basta comparar cada ponto com K pontos que o seguem**, sendo K uma constante segura. $K = 15$ serve, mas podia ser menor.
- Para $K > 15$, a distância é superior a δ. Porquê? **Cada quadrado $\frac{\delta}{2} \times \frac{\delta}{2}$ tem no máximo um ponto da faixa**. Se fosse mais do que um, a distância não excedia o comprimento da diagonal, $\frac{\sqrt{2}}{2}\delta$, que é $< \delta$. Absurdo! pois cada quadrado está contido ou em $\mathcal{P}_1$ ou em $\mathcal{P}_2$. Pontos $\bullet$ e $\star$ que diferem em 15 posições ou mais no vetor ordenado estão a distância maior do que $\frac{3}{2}\delta$ (contar as células...).

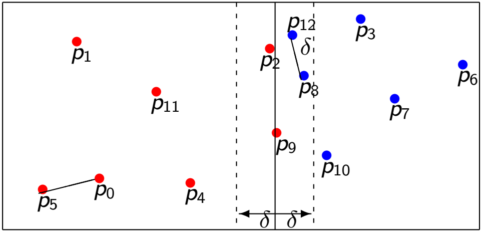
Método de Shamos - Par a distância mínima no plano

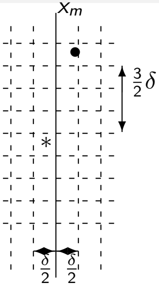




- A fase de combinação realiza-se em $O(n)$: se tomarmos os pontos na faixa por ordem decrescente de ordenada, **basta comparar cada ponto com K pontos que o seguem**, sendo K uma constante segura. $K = 15$ serve, mas podia ser menor.
- Para $K > 15$, a distância é superior a δ. Porquê? **Cada quadrado $\frac{\delta}{2} \times \frac{\delta}{2}$ tem no máximo um ponto da faixa**. Se fosse mais do que um, a distância não excedia o comprimento da diagonal, $\frac{\sqrt{2}}{2}\delta$, que é $< \delta$. Absurdo! pois cada quadrado está contido ou em $\mathcal{P}_1$ ou em $\mathcal{P}_2$. Pontos $\bullet$ e $\star$ que diferem em 15 posições ou mais no vetor ordenado estão a distância maior do que $\frac{3}{2}\delta$ (contar as células...).

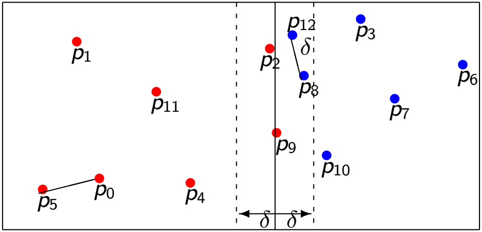
Método de Shamos - Par a distância mínima no plano

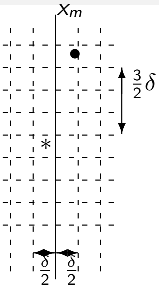




- A fase de combinação realiza-se em $O(n)$: se tomarmos os pontos na faixa por ordem decrescente de ordenada, **basta comparar cada ponto com K pontos que o seguem**, sendo K uma constante segura. $K = 15$ serve, mas podia ser menor.
- Para $K > 15$, a distância é superior a δ. Porquê? **Cada quadrado $\frac{\delta}{2} \times \frac{\delta}{2}$ tem no máximo um ponto da faixa**. Se fosse mais do que um, a distância não excedia o comprimento da diagonal, $\frac{\sqrt{2}}{2}\delta$, que é $< \delta$. Absurdo! pois cada quadrado está contido ou em $\mathcal{P}_1$ ou em $\mathcal{P}_2$. Pontos $\bullet$ e $\star$ que diferem em 15 posições ou mais no vetor ordenado estão a distância maior do que $\frac{3}{2}\delta$ (contar as células. . .).

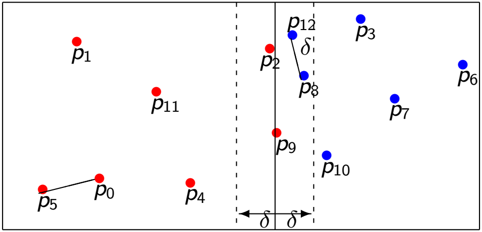
Método de Shamos - Par a distância mínima no plano

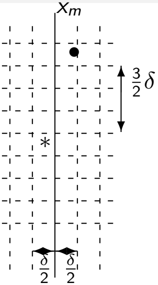




- A fase de combinação realiza-se em $O(n)$: se tomarmos os pontos na faixa por ordem decrescente de ordenada, **basta comparar cada ponto com K pontos que o seguem**, sendo K uma constante segura. $K = 15$ serve, mas podia ser menor.
- Para $K > 15$, a distância é superior a δ. Porquê? **Cada quadrado $\frac{\delta}{2} \times \frac{\delta}{2}$ tem no máximo um ponto da faixa**. Se fosse mais do que um, a distância não excedia o comprimento da diagonal, $\frac{\sqrt{2}}{2}\delta$, que é $< \delta$. Absurdo! pois cada quadrado está contido ou em $\mathcal{P}_1$ ou em $\mathcal{P}_2$. Pontos $\bullet$ e $\star$ que diferem em 15 posições ou mais no vetor ordenado estão a distância maior do que $\frac{3}{2}\delta$ (contar as células...).

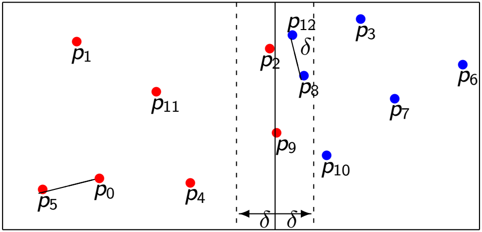
Método de Shamos - Par a distância mínima no plano

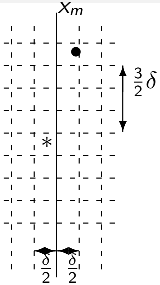




- A fase de combinação realiza-se em $O(n)$: se tomarmos os pontos na faixa por ordem decrescente de ordenada, **basta comparar cada ponto com K pontos que o seguem**, sendo K uma constante segura. $K = 15$ serve, mas podia ser menor.
- Para $K > 15$, a distância é superior a δ. Porquê? **Cada quadrado $\frac{\delta}{2} \times \frac{\delta}{2}$ tem no máximo um ponto da faixa**. Se fosse mais do que um, a distância não excedia o comprimento da diagonal, $\frac{\sqrt{2}}{2}\delta$, que é $< \delta$. Absurdo! pois cada quadrado está contido ou em $\mathcal{P}_1$ ou em $\mathcal{P}_2$. Pontos $\bullet$ e $\star$ que diferem em 15 posições ou mais no vetor ordenado estão a distância maior do que $\frac{3}{2}\delta$ (contar as células. . .).

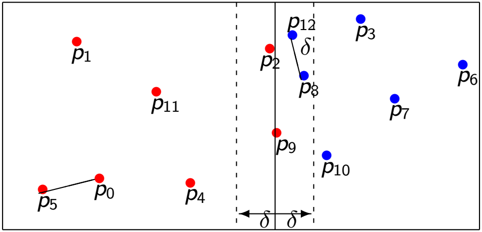
Método de Shamos - Par a distância mínima no plano

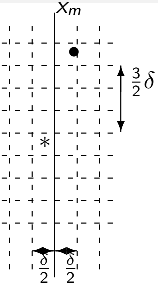




- A fase de combinação realiza-se em $O(n)$: se tomarmos os pontos na faixa por ordem decrescente de ordenada, **basta comparar cada ponto com K pontos que o seguem**, sendo K uma constante segura. $K = 15$ serve, mas podia ser menor.
- Para $K > 15$, a distância é superior a δ. Porquê? **Cada quadrado $\frac{\delta}{2} \times \frac{\delta}{2}$ tem no máximo um ponto da faixa**. Se fosse mais do que um, a distância não excedia o comprimento da diagonal, $\frac{\sqrt{2}}{2}\delta$, que é $< \delta$. Absurdo! pois cada quadrado está contido ou em $\mathcal{P}_1$ ou em $\mathcal{P}_2$. Pontos $\bullet$ e $\star$ que diferem em 15 posições ou mais no vetor ordenado estão a distância maior do que $\frac{3}{2}\delta$ (contar as células...).

Método de Shamos - Par a distância mínima no plano





- A fase de combinação realiza-se em $O(n)$: se tomarmos os pontos na faixa por ordem decrescente de ordenada, **basta comparar cada ponto com K pontos que o seguem**, sendo K uma constante segura. $K = 15$ serve, mas podia ser menor.
- Para $K > 15$, a distância é superior a δ. Porquê? **Cada quadrado $\frac{\delta}{2} \times \frac{\delta}{2}$ tem no máximo um ponto da faixa**. Se fosse mais do que um, a distância não excedia o comprimento da diagonal, $\frac{\sqrt{2}}{2}\delta$, que é $< \delta$. Absurdo! pois cada quadrado está contido ou em $\mathcal{P}_1$ ou em $\mathcal{P}_2$. Pontos $\bullet$ e $\star$ que diferem em 15 posições ou mais no vetor ordenado estão a distância maior do que $\frac{3}{2}\delta$ (contar as células...).

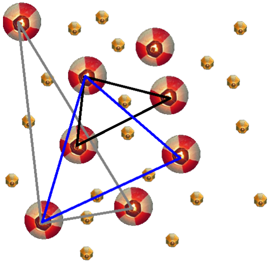
J - Saint John Festival (SWERC 2015)

Problem

Given two sets of points $\mathcal{A}$ and $\mathcal{B}$ in the **plane**, how many points in $\mathcal{B}$ are in the interior or on the boundary of triangles defined by any 3 points in $\mathcal{A}$.

Classification

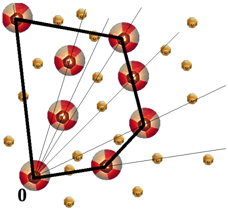
- Categories: Geometry
  - **Convex hull**
  - Point in **convex** polygon
- Difficulty: Medium



J - Saint John Festival (SWERC 2015)

Sample Solution

Background (from Charatheodory's Theorem): The union of all triangles having vertices in $\mathcal{A}$ is the **convex hull** $\mathcal{CH}(\mathcal{A})$ of $\mathcal{A}$.

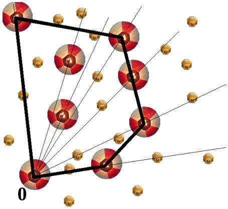


1. Find $\mathcal{CH}(\mathcal{A})$ in $\mathcal{O}(L \log L)$, e.g., by Graham scan, with $L = |\mathcal{A}|$.
   - Do not compute polar angles.
   - Make use of primitive operations based on cross product and inner product: left-turn, right-turn; handle collinearities.
2. Then, for each $p \in \mathcal{B}$, check if $p \in \mathcal{CH}(\mathcal{A})$ efficiently.

J - Saint John Festival (SWERC 2015)

Sample Solution

Background (from Charatheodory's Theorem): The union of all triangles having vertices in $\mathcal{A}$ is the **convex hull** $\mathcal{CH}(\mathcal{A})$ of $\mathcal{A}$.

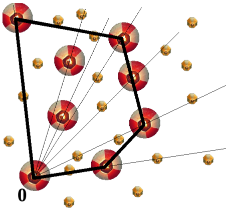


1. Find $\mathcal{CH}(\mathcal{A})$ in $\mathcal{O}(L \log L)$, e.g., by Graham scan, with $L = |\mathcal{A}|$.
   - Do not compute polar angles.
   - Make use of primitive operations based on cross product and inner product: left-turn, right-turn; handle collinearities.
2. Then, for each $p \in \mathcal{B}$, check if $p \in \mathcal{CH}(\mathcal{A})$ efficiently.

J - Saint John Festival (SWERC 2015)

Sample Solution

Background (from Charatheodory's Theorem): The union of all triangles having vertices in $\mathcal{A}$ is the **convex hull** $\mathcal{CH}(\mathcal{A})$ of $\mathcal{A}$.

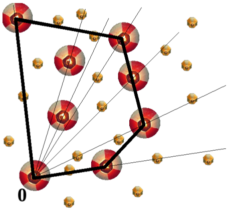


1. Find $\mathcal{CH}(\mathcal{A})$ in $\mathcal{O}(L \log L)$, e.g., by Graham scan, with $L = |\mathcal{A}|$.
   - Do not compute polar angles.
   - Make use of primitive operations based on cross product and inner product: left-turn, right-turn; handle collinearities.
2. Then, for each $p \in \mathcal{B}$, check if $p \in \mathcal{CH}(\mathcal{A})$ efficiently.

J - Saint John Festival (SWERC 2015)

Sample Solution

Background (from Charatheodory's Theorem): The union of all triangles having vertices in $\mathcal{A}$ is the **convex hull** $\mathcal{CH}(\mathcal{A})$ of $\mathcal{A}$.



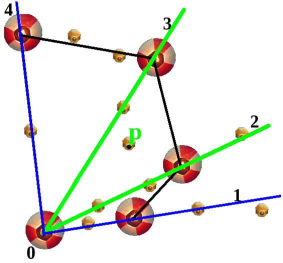
1. Find $\mathcal{CH}(\mathcal{A})$ in $\mathcal{O}(L \log L)$, e.g., by Graham scan, with $L = |\mathcal{A}|$.
    - Do not compute polar angles.
    - Make use of primitive operations based on cross product and inner product: left-turn, right-turn; handle collinearities.
2. Then, for each $p \in \mathcal{B}$, check if $p \in \mathcal{CH}(\mathcal{A})$ efficiently.

J - Saint John Festival (SWERC 2015)

Sample Solution

For each $p \in \mathcal{B}$, check if $p \in \mathcal{CH}(\mathcal{A})$ in $\mathcal{O}(\log h)$, where h is the number of vertices of $\mathcal{CH}(\mathcal{A})$.

- $\mathcal{CH}(\mathcal{A})$ partitioned into wedges with apex p_0 (the lowest vertex);
- The wedges are already sorted. **Binary Search** for finding p.



$p \in \text{wedge}(0, 2, 3)$
$(0, 2, p)$ L-turn
$(0, 3, p)$ R-turn
$(2, 3, p)$ L-turn

- $\mathcal{CH}(\mathcal{A})$ is a **convex** polygon.
- Overall time complexity: $\mathcal{O}(L \log L + S \log h)$ or $\mathcal{O}((L + S) \log L)$.
- Robust: L-turn; R-turn; collinear; point in line segment.
- (p, q, r) is a left-turn (right-turn) if $\vec{pq} \times \vec{pr}$ positive (negative)...

Convex hull de n pontos no plano

Definição – Conjunto convexo

Um conjunto S é **convexo** se quaisquer que sejam os pontos $p, q \in S$, todos os pontos do segmento de recta $[p, q]$ estão em S.

Definição – Invólucro convexo (*convex hull*)

Seja P um conjunto de n pontos no plano. O **invólucro convexo** $\mathcal{CH}(\mathcal{P})$ de P é o *menor* conjunto convexo que contém P (menor significa que não contém nenhum outro polígono convexo que contenha P).

No exemplo, a fronteira de $\mathcal{CH}(\mathcal{P})$ é dada por $v_1, v_2, v_3, v_4, v_5, v_1$, se for percorrida no sentido anti-horário (*CCW*, em Inglês).

r está à esquerda de $\overrightarrow{pq}$

Produto escalar: $\vec{u} \cdot \vec{v} = x_1 y_1 + x_2 y_2$

Produto vetorial:

$$\vec{u} \times \vec{v} = \begin{vmatrix} x_1 & y_1 & 0 \\ x_2 & y_2 & 0 \\ \vec{i} & \vec{j} & \vec{k} \end{vmatrix} = (x_1 y_2 - y_1 x_2)\vec{k}$$

Convex hull de n pontos no plano

Definição – Conjunto convexo

Um conjunto S é **convexo** se quaisquer que sejam os pontos $p, q \in S$, todos os pontos do segmento de recta $[p, q]$ estão em S.

Definição – Invólucro convexo (*convex hull*)

Seja P um conjunto de n pontos no plano. O **invólucro convexo** $\mathcal{CH}(\mathcal{P})$ de P é o *menor* conjunto convexo que contém P (menor significa que não contém nenhum outro polígono convexo que contenha P).

No exemplo, a fronteira de $\mathcal{CH}(\mathcal{P})$ é dada por $v_1, v_2, v_3, v_4, v_5, v_1$, se for percorrida no sentido anti-horário (*CCW*, em Inglês).

r está à esquerda de $\overrightarrow{pq}$

Produto escalar: $\vec{u} \cdot \vec{v} = x_1 y_1 + x_2 y_2$

Produto vetorial:

$$\vec{u} \times \vec{v} = \begin{vmatrix} x_1 & y_1 & 0 \\ x_2 & y_2 & 0 \\ \vec{i} & \vec{j} & \vec{k} \end{vmatrix} = (x_1 y_2 - y_1 x_2)\vec{k}$$

Convex hull de n pontos no plano

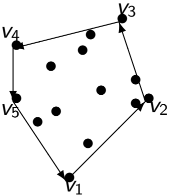
Definição – Conjunto convexo

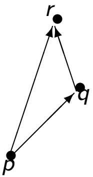
Um conjunto S é **convexo** se quaisquer que sejam os pontos $p, q \in S$, todos os pontos do segmento de recta $[p, q]$ estão em S.

Definição – Invólucro convexo (*convex hull*)

Seja P um conjunto de n pontos no plano. O **invólucro convexo** $\mathcal{CH}(\mathcal{P})$ de P é o *menor* conjunto convexo que contém P (menor significa que não contém nenhum outro polígono convexo que contenha P).

No exemplo, a fronteira de $\mathcal{CH}(\mathcal{P})$ é dada por $v_1, v_2, v_3, v_4, v_5, v_1$, se for percorrida no sentido anti-horário (CCW, em Inglês).





r está à esquerda de $\overrightarrow{pq}$

Produto escalar: $\vec{u} \cdot \vec{v} = x_1 y_1 + x_2 y_2$

Produto vetorial:

$$\vec{u} \times \vec{v} = \begin{vmatrix} x_1 & y_1 & 0 \\ x_2 & y_2 & 0 \\ \vec{i} & \vec{j} & \vec{k} \end{vmatrix} = (x_1 y_2 - y_1 x_2)\vec{k}$$

Convex hull de n pontos no plano – Algoritmo de Graham

Teste de viragem

Dados três pontos p, q e r no plano, não colineares, o sinal da componente não nula do produto vetorial $\overrightarrow{pq} \times \overrightarrow{pr}$ indica se (p, q, r) define uma **viragem à esquerda** ou **a direita**.

Se o referencial $(\mathcal{O}, \vec{i}, \vec{j}, \vec{k})$ tem orientação positiva e é ortonormado, como o canónico, se o sinal for positivo, a viragem é à esquerda e, se for negativo, é à direita.

$$\begin{vmatrix} x_p & y_p & 1 \\ x_q & y_q & 1 \\ x_r & y_r & 1 \end{vmatrix} = \begin{vmatrix} x_p & y_p & 1 \\ x_q - x_p & y_q - y_p & 0 \\ x_r - x_p & y_r - y_p & 0 \end{vmatrix} = (x_q - x_p)(y_r - y_p) - (y_q - y_p)(x_r - x_p)$$

O algoritmo de Graham calcula $\mathcal{CH}(\mathcal{P})$ em $O(n \log_2 n)$. A sua correção baseia-se na propriedade seguinte:

Um polígono é convexo se e só se quando se percorre a sua fronteira no sentido CCW, se vira à esquerda em cada vértice.

Convex hull de n pontos no plano – Algoritmo de Graham

Teste de viragem

Dados três pontos p, q e r no plano, não colineares, o sinal da componente não nula do produto vetorial $\overrightarrow{pq} \times \overrightarrow{pr}$ indica se (p, q, r) define uma **viragem à esquerda** ou **a direita**. Se o referencial $(\mathcal{O}, \vec{i}, \vec{j}, \vec{k})$ tem orientação positiva e é ortonormado, como o canónico, se o sinal for positivo, a viragem é à esquerda e, se for negativo, é à direita.

$$\begin{vmatrix} x_p & y_p & 1 \\ x_q & y_q & 1 \\ x_r & y_r & 1 \end{vmatrix} = \begin{vmatrix} x_p & y_p & 1 \\ x_q - x_p & y_q - y_p & 0 \\ x_r - x_p & y_r - y_p & 0 \end{vmatrix} = (x_q - x_p)(y_r - y_p) - (y_q - y_p)(x_r - x_p)$$

O algoritmo de Graham calcula $\mathcal{CH}(\mathcal{P})$ em $O(n \log_2 n)$. A sua correção baseia-se na propriedade seguinte:

Um polígono é convexo se e só se quando se percorre a sua fronteira no sentido CCW, se vira à esquerda em cada vértice.

Convex hull de n pontos no plano – Algoritmo de Graham

Teste de viragem

Dados três pontos p, q e r no plano, não colineares, o sinal da componente não nula do produto vetorial $\overrightarrow{pq} \times \overrightarrow{pr}$ indica se (p, q, r) define uma **viragem à esquerda** ou **a direita**. Se o referencial $(\mathcal{O}, \vec{i}, \vec{j}, \vec{k})$ tem orientação positiva e é ortonormado, como o canónico, se o sinal for positivo, a viragem é à esquerda e, se for negativo, é à direita.

$$\begin{vmatrix} x_p & y_p & 1 \\ x_q & y_q & 1 \\ x_r & y_r & 1 \end{vmatrix} = \begin{vmatrix} x_p & y_p & 1 \\ x_q - x_p & y_q - y_p & 0 \\ x_r - x_p & y_r - y_p & 0 \end{vmatrix} = (x_q - x_p)(y_r - y_p) - (y_q - y_p)(x_r - x_p)$$

O algoritmo de Graham calcula $\mathcal{CH}(\mathcal{P})$ em $O(n \log_2 n)$. A sua correção baseia-se na propriedade seguinte:

Um polígono é convexo se e só se quando se percorre a sua fronteira no sentido CCW, se vira à esquerda em cada vértice.

Algoritmo de Graham (*Graham scan*)

Para facilitar, supomos P **em posição geral:** não existem três pontos colineares nem dois com a mesma ordenada ou abcissa.

Graham-Scan($\mathcal{P}$)

1. Seja p_1 o ponto de $\mathcal{P}$ com menor ordenada
2. Seja $\{p_2, \ldots, p_n\}$ o conjunto $\mathcal{P} \setminus \{p_1\}$ ordenado em CCW em torno de p_1 (i.e., por ordem crescente de ângulo polar)
3. Criar uma pilha S e inserir p_1, p_2, p_3 em S (p_3 fica no topo)
4. Para $i \leftarrow 4$ até n fazer
   /* sendo w o topo atual da pilha e w^- o elemento abaixo dele */
5.   Enquanto (w^-, w, p_i) não for viragem à esquerda fazer
6.   Retirar w de S
7.   Colocar p_i em S
8. retornar S

Complexidade temporal do bloco 3–7: $\Theta(n)$

$n - 3 \leq$ número total de testes de viragem $< 2n$. Cada p_i entra uma vez em S e, se sair, não volta a ser considerado. Quando um teste de viragem falha, retira o topo da pilha. **Análise de complexidade amortizada.**

Complexidade do algoritmo de Graham: $O(n \log n)$, se usar *mergesort* no passo 2.

Algoritmo de Graham (*Graham scan*)

Para facilitar, supomos P **em posição geral:** não existem três pontos colineares nem dois com a mesma ordenada ou abcissa.

GRAHAM-SCAN($\mathcal{P}$)

1. Seja p_1 o ponto de $\mathcal{P}$ com menor ordenada
2. Seja $\{p_2, \ldots, p_n\}$ o conjunto $\mathcal{P} \setminus \{p_1\}$ ordenado em CCW em torno de p_1 (i.e., por ordem crescente de ângulo polar)
3. Criar uma pilha S e inserir p_1, p_2, p_3 em S (p_3 fica no topo)
4. Para $i \leftarrow 4$ até n fazer
   /* sendo w o topo atual da pilha e w^- o elemento abaixo dele */
5. Enquanto (w^-, w, p_i) não for viragem à esquerda fazer
6. Retirar w de S
7. Colocar p_i em S
8. retornar S

Complexidade temporal do bloco 3–7: $\Theta(n)$

$n - 3 \leq$ número total de testes de viragem $< 2n$. Cada p_i entra uma vez em S e, se sair, não volta a ser considerado. Quando um teste de viragem falha, retira o topo da pilha. **Análise de complexidade amortizada.**

Complexidade do algoritmo de Graham: $O(n \log n)$, se usar *mergesort* no passo 2.

Algoritmo de Graham (*Graham scan*)

Para facilitar, supomos P **em posição geral:** não existem três pontos colineares nem dois com a mesma ordenada ou abcissa.

GRAHAM-SCAN($\mathcal{P}$)

1. Seja p_1 o ponto de $\mathcal{P}$ com menor ordenada
2. Seja $\{p_2, \ldots, p_n\}$ o conjunto $\mathcal{P} \setminus \{p_1\}$ ordenado em CCW em torno de p_1 (i.e., por ordem crescente de ângulo polar)
3. Criar uma pilha S e inserir p_1, p_2, p_3 em S (p_3 fica no topo)
4. Para $i \leftarrow 4$ até n fazer
   /* sendo w o topo atual da pilha e w^- o elemento abaixo dele */
5.   Enquanto (w^-, w, p_i) não for viragem à esquerda fazer
6.   Retirar w de S
7.   Colocar p_i em S
8. retornar S

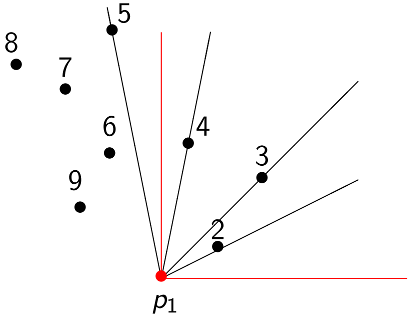
Complexidade temporal do bloco 3–7: $\Theta(n)$

$n - 3 \leq$ número total de testes de viragem $< 2n$. Cada p_i entra uma vez em S e, se sair, não volta a ser considerado. Quando um teste de viragem falha, retira o topo da pilha. **Análise de complexidade amortizada.**

Complexidade do algoritmo de Graham: $O(n \log n)$, se usar *mergesort* no passo 2.

Ordenação dos pontos por ordem crescente de ângulo polar

Para ordenar os pontos por ordem de ângulo polar crescente relativamente a p_1, não é preciso calcular os ângulos polares explicitamente para depois os comparar.



Para decidir se p ficará antes ou depois de q basta notar que:

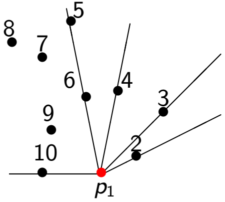
– Se (p_1, p, q) definir uma viragem à esquerda, q tem ângulo polar maior do que p.

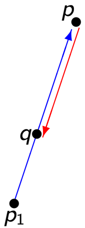
– Se (p_1, p, q) definir uma viragem à direita, q tem ângulo polar menor do que p.

Assim, evitamos os erros numéricos inerentes ao cálculo de ângulos.

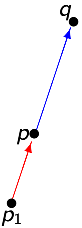
E, se os pontos não estão em posição geral?

- Se há mais do que um ponto com ordenada mínima, p_1 será o que tem a abcissa maior entre esses pontos.
- Se há três pontos sobre um mesmo raio com origem em p_1, então $\overrightarrow{p_1p} \times \overrightarrow{p_1q} = \vec{0}$.
  - Colocar primeiro o ponto que estiver mais afastado de p_1.
  - Se o *produto escalar* $\overrightarrow{p_1p} \cdot \overrightarrow{pq}$ for negativo, p é o mais afastado. Se for positivo, q é o mais afastado.
  - Se $\overrightarrow{p_1p} \times \overrightarrow{p_1q} = \vec{0}$, a viragem é à esquerda se $\overrightarrow{p_1p} \cdot \overrightarrow{pq} < 0$.





$\overrightarrow{p_1p} \cdot \overrightarrow{pq} < 0$



$\overrightarrow{p_1p} \cdot \overrightarrow{pq} > 0$