Algoritmos de Ordenação

1. Ordenação por seleção

Algorithm 1 selection_sort(**ref** A, p, r)

Require: A representa vetor; p e r são índices do vetor com $p < r$
Ensure: $A[p \dots r]$ está ordenado
  if $r > p$ **then**
    $minidx \leftarrow p$
    for $i \leftarrow p + 1$ **to** r **do**
      if $A[i] < A[minidx]$ **then**
        $minidx \leftarrow i$
      end if
    end for
    $swap(A[p], A[minidx])$
    $selection\_sort(A, p + 1, r)$
  end if

Algorithm 2 selection_sort2(**ref** A, p, r)

Require: A representa vetor; p e r são índices do vetor com $p < r$
Ensure: $A[p \dots r]$ está ordenado
  if $r > p$ **then**
    $maxidx \leftarrow p$
    for $i \leftarrow p + 1$ **to** r **do**
      if $A[i] > A[maxidx]$ **then**
        $maxidx \leftarrow i$
      end if
    end for
    $swap(A[r], A[maxidx])$
    $selection\_sort(A, p, r - 1)$
  end if

---

2. Ordenação por bolha

Algorithm 3 buble_sort(**ref** A, p, r)

Require: A representa vetor; p e r são índices do vetor com $p < r$
Ensure: $A[p \ldots r]$ está ordenado

```
if r > p then
    for i ← r downto p+1 do
        if A[i − 1] > A[i] then
            swap(A[i], A[i − 1])
        end if
    end for
    buble_sort(A, p + 1, r)
end if
```

3. Ordenação por inserção

Algorithm 4 insert(**ref** A, p)

Require: $A[0 \ldots p-1]$ é um vetor ordenado ; p é um índice do vetor
Ensure: $A[0 \ldots p]$ está ordenado

```
x ← A[p]
i ← p − 1
while i > 0 and A[i] > x do
    A[i + 1] ← A[i]
    i ← i − 1
end while
A[i + 1] ← x
```

Algorithm 5 insertion_sort(**ref** A, p, r)

Require: A é um vetor ; p e r são índices do vetor com $p < r$
Ensure: $A[p \ldots r]$ está ordenado

```
if p < r then
    insert(insertion_sort(A, p, r − 1), r)
end if
```

4. No algoritmo mergesort, um vetor $A[start \ldots end]$ de tamanho maior que 1 é ordenado da seguinte maneira:

   - O vetor é dividido em duas metades:
     - $A[start \ldots \lfloor (start + end)/2 \rfloor]$
     - $A[\lceil (start + end)/2 \rceil \ldots end]$

- Cada metade é resolvida utilizando o próprio algoritmo mergesort.
- Em seguida, os dois vetores ordenados são combinados para produzir um único vetor ordenado.

O algoritmo do mergesort é descrito em pseudocódigo pelo Algoritmo 6

Algorithm 6 $mergesort(A, start, end)$

Require: Um vetor $A[start \dots end]$ com elementos que podem ser comparados e dois inteiros $start$ e end
Ensure: O subvetor $A[start \dots end]$ ordenado em ordem não decrescente.
  if $q - p > 1$ **then**
    $mid \leftarrow \lfloor (p + q)/2 \rfloor$
    $mergesort(A, start, mid)$
    $mergesort(A, mid + 1, end)$
    $merge(A, start, mid, end)$
  end if

O processo de entrelaçamento dos dois vetores ordenado pode ser descrito da seguinte maneira: dois índices são inicializados no ínicio de cada vetor ordenado. Os elementos apontados pelos índices são comparados e o menor deles é adicionado no novo vetor, o índice do menor elemento é incrementado. Essa operação é realizada até que um dos vetores acabe seus elementos. Em seguida, os elementos do vetor não vazio são copiados para o novo vetor. No final, o novo vetor é copiado para o vetor orginal. O algoritmo de merge dos dois vetores ordenados está descrito em pseudocódigo no Algoritmo 9.

O vetor é modificado durante a execução do algoritmo de merge, vamos mostrar o processo de ordenação mostrando todas as transformações realizadas pelo algoritmo merge considerando a seguinte entrada $[1, 2, 1, 3, 5, 3, 2, 1]$:

Função	Entrada	Vetor resultado
merge	[1], [2]	[1,2]
merge	[1] ,[3]	[1,3]
merge	[1,2], [1,3]	[1,1,2,3]
merge	[5] ,[3]	[3,5]
merge	[2] ,[1]	[1,2]
merge	[3,5], [1,2]	[1,2,3,5]
merge	[1,1,2,3] ,[1,2,3,5]	[1,1,1,2,2,3,3,5]

Algorithm 7 $merge(start, mid, end)$

Require: Um vetor $A[start \ldots end]$ com elementos que podem ser comparados e três inteiros $start$, mid e end

Ensure: O subvetor $A[start \ldots end]$ ordenado em ordem não decrescente.

$start1 \leftarrow start$
$start2 \leftarrow mid + 1$
$start3 \leftarrow 0$
Seja B um vetor de tamanho $end - start + 1$
while start1 ¡= mid **and** start2 ¡= end **do**
    if A[start1] ¡ A[start2] **then**
        $B[start3] \leftarrow A[start1]$; $start1 \leftarrow start1 + 1$
    else
        $B[start3] \leftarrow A[start2]$; $start2 \leftarrow start2 + 1$
    end if
    $start3 \leftarrow start3 + 1$
end while
while start1 ¡= mid **do**
    $B[start3] \leftarrow A[start1]$; $start1 \leftarrow start1 + 1$; $start3 \leftarrow start3 + 1$
end while
while start2 ¡= mid **do**
    $B[start3] \leftarrow A[start2]$; $start2 \leftarrow start2 + 1$; $start3 \leftarrow start3 + 1$
end while
$k \leftarrow 0$
for $i \leftarrow start$ **to** end **do**
    $A[i] \leftarrow B[k]$; $k \leftarrow k + 1$
end for

Muitos problemas podem ser resolvidos facilmente quando recebemos um vetor já ordenado. No próximo exemplo, adaptaremos o algoritmo de mergesort para executar uma tarefa que pode ser realizada de maneira bastante rápida quando consideramos o vetor ordenado.

5. MergeCount

   Dado um vetor não ordenado $A[0 \ldots n-1]$, devolva a quantidade elementos repetidos no vetor. Por exemplo,

   - Dado A = 1,2,1,3,5,3,2,1, temos 4 repetições.
   - Dado A = 1,2,1,3,5,3,2,1,2, temos 5 repetições.

   No algoritmo mergecount, um vetor $A[start \ldots end]$ de tamanho maior que 1 é ordenado da seguinte maneira:

- O vetor é dividido em duas metades:
  - $A[start \dots \lfloor (start + end)/2 \rfloor]$
  - $A[\lceil (start + end)/2 \rceil \dots end]$
- Em cada metade, contaremos a quantidade de repetições de elementos em cada metade.
- Em seguida, contaremos a quantidade de repetições de elementos que aparecem em metades distintas.

Algorithm 8 Mergecount

Require: Um vetor $A[start \dots end]$ com elementos que podem ser comparados e dois inteiros $start$ e end

Ensure: O subvetor $A[start \dots end]$ ordenado em ordem não decrescente e devolve a quantidade de repetições no subvetor $A[start \dots end]$.

```
if q − p > 1 then
    mid ← ⌊(p + q)/2⌋
    p1 ← mergecount(A, start, mid)
    p2 ← mergecount(A, mid + 1, end)
    p3 ← merge(A, start, mid, end)
    return p1 + p2 + p3
else
    return 0
end if
```

Agora, precisamos adaptar o algoritmo de merge para contar a quantidade de elementos repetidos entre vetores ordenados.

Todas as repetições são contadas pelo algoritmo de merge, vamos mostrar o resultado de todas as contagem realizada pelo algoritmo merge considerando a seguinte entrada $[1, 2, 1, 3, 5, 3, 2, 1]$:

Função	Entrada	Número de repetições
merge	[1], [2]	0
merge	[1] ,[3]	0
merge	[1,2], [1,3]	1
merge	[5] ,[3]	0
merge	[2] ,[1]	0
merge	[3,5], [1,2]	0
merge	[1,1,2,3] ,[1,2,3,5]	3

O total de repetições calculada pelo Algoritmo é 4.

Algorithm 9 Merge

Require: Um vetor $A[start \dots end]$ com elementos que podem ser comparados e três inteiros $start$, mid e end

Ensure: O subvetor $A[start \dots end]$ ordenado em ordem não decrescente.

$start1 \leftarrow start$
$start2 \leftarrow mid + 1$
$start3 \leftarrow 0$
Seja B um vetor de tamanho $end - start + 1$
$cont \leftarrow 0$
while start1 ¡= mid **and** start2 ¡= end **do**
    if A[start1] ¡ A[start2] **then**
        $B[start3] \leftarrow A[start1]$; $start1 \leftarrow start1 + 1$
    else
        if A[start1] == A[start2] **then**
            $cont \leftarrow cont + 1$
        end if
        $B[start3] \leftarrow A[start2]$; $start2 \leftarrow start2 + 1$
    end if
    $start3 \leftarrow start3 + 1$
end while
while start1 ¡= mid **do**
    $B[start3] \leftarrow A[start1]$; $start1 \leftarrow start1 + 1$; $start3 \leftarrow start3 + 1$
end while
while start2 ¡= mid **do**
    $B[start3] \leftarrow A[start2]$; $start2 \leftarrow start2 + 1$; $start3 \leftarrow start3 + 1$
end while
$k \leftarrow 0$
for $i \leftarrow start$ **to** end **do**
    $A[i] \leftarrow B[k]$; $k \leftarrow k + 1$
end for
return $cont$

6. QuickSort

   Dado um vetor não ordenado $A[0 \dots n-1]$, ordene o vetor utilizando o algoritmo de quicksort.

   O algoritmo de quicksort é um outro exemplo de algoritmo de divisão e conquista. O algoritmo do quicksort pode ser descrito da seguinte maneira:

   - Utilizando um elemento particular do vetor chamado de pivô,o vetor é particionado em duas partes: os elementos menores que o pivô e os elementos maiores que o pivô.

- Cada parte é ordenado de maneira recursiva.

O pseudocódigo do algoritmo do quicksort é o seguinte:

Algorithm 10 Quicksort

Require: Um vetor $A[start \ldots end]$ com elementos que podem ser comparados e dois inteiros $start$ e end representando índices do vetor A.
Ensure: O subvetor $A[start \ldots end]$ ordenado em ordem não decrescente.
  if $q - p > 1$ **then**
    $j \leftarrow particiona(A, start, end)$
    $quicksort(A, start, j - 1)$
    $quicksort(A, j + 1, end)$
  end if

Algorithm 11 particiona(A, start, end)

Require: Um vetor $A[start \ldots end]$ com elementos que podem ser comparados e dois inteiros $start$ e end representando índices do vetor A.
Ensure: devolve o índice j tal que $A[i] \leq piv\hat{o} \forall i \leq j$ e $A[i] \geq piv\hat{o} \forall i \geq j$
  $j \leftarrow start$
  $pivo \leftarrow A[end]$
  for $k \leftarrow start$ **to** end-1 **do**
    if $A[k] \leq pivo$ **then**
      $A[k] \leftrightarrow A[j]$
      $j \leftarrow j + 1$
    end if
  end for
  $A[j] \leftrightarrow A[end]$
  return j

---

7. QuickCount

   Dado um vetor não ordenado $A[0 \ldots n - 1]$, devolva a quantidade elementos repetidos no vetor adaptando o algoritmo do quicksort. Por exemplo,

   - Dado A = 1,2,1,3,5,3,2,1, temos 4 repetições.

- Dado A = 1,2,1,3,5,3,2,1,2, temos 5 repetições.

O adaptação do algoritmo do quicksort pode ser descrito da seguinte maneira:

- Utilizando um elemento particular do vetor chamado de pivô,o vetor é particionado em três partes: os elementos menores que o pivô, os elementos iguais ao pivô e os elementos maiores ou iguais que o pivô.
- Cada parte é ordenado de maneira recursiva e devolve a quantidade de elementos repetidos em cada parte.

Algorithm 12 Quickcount

Require: Um vetor $A[start \dots end]$ com elementos que podem ser comparados e dois inteiros $start$ e end representando índices do vetor A.

Ensure: O subvetor $A[start \dots end]$ ordenado em ordem não decrescente.

if $end - start > 1$ **then**
  $particiona2(A, start, end, start1, end1)$
  $p1 \leftarrow quicksort(A, start, start1 - 1)$
  $p2 \leftarrow quicksort(A, end1 + 1, end)$
  return p1 + p2 + end1 - start1
else
  return 0
end if

O algoritmo particiona2 chama o algoritmo particiona para separa em duas partes. Em seguida, particiona a segunda parte em mais duas partes. O ínicio e o fim da parte central é devolvida pela função particiona2 através das variáveis start1 e end1.

Todas as repetições são contadas pelo algoritmo particiona2, vamos mostrar o resultado de todas as contagem realizada pelo algoritmo particiona2 considerando a seguinte entrada $[1, 2, 1, 3, 5, 3, 2, 1]$:

Entrada	Pivô	Particões	Número de repeticões
$[1, 2, 1, 3, 5, 3, 2, 1]$	1	$[], [1, 1, 1], [3, 5, 3, 2, 2]$	2
$[3, 5, 3, 2, 2]$	2	$[], [2, 2], [3, 5, 3]$	1
$[3, 5, 3]$	5	$[3, 3], [5], []$	0
$[3, 3]$	3	$[], [3, 3], []$	1

Algorithm 13 particiona2(A, start, end, start1, end1)

Require: Um vetor $A[start \dots end]$ com elementos que podem ser comparados e dois inteiros $start$ e end representando índices do vetor A.
Ensure: devolve o índice j tal que $A[i] \leq piv\hat{o}\forall i \leq j$ e $A[i] \geq pivo\forall i \geq j$

$start1 \leftarrow particiona(A, start, end)$
$j \leftarrow start1$
$pivo \leftarrow A[j]$
for $k \leftarrow start1$ **to** end **do**
  if $A[k] == pivo$ **then**
    $A[k] \leftrightarrow A[j]$
    $j \leftarrow j + 1$
  end if
end for
$end1 \leftarrow j$

O total de repetições calculada pelo Algoritmo é 4.