Vectores

Mestrado Integrado em Engenharia Informática FCT UNL
http://ctp.di.fct.unl.pt/miei/ip/

Corpo Docente 2020/2021
António Ravara, Artur Miguel Dias, Bernardo Toninho,
Ema Vieira, Inês Fernandes, Margarida Mamede,
Miguel Monteiro, Rui Nóbrega

Operações em Vectores
(Inserção ordenada)

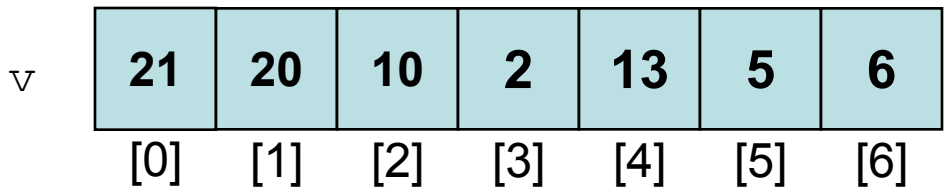
```
public void insertSort(type e) {
    int pos=-1;
    int i=0;
    if (isFull()) resize();
    while (i<count && pos==-1)
        if (v[i] >= e)
          pos=i;
        else
          i++;
    if (pos == -1) pos=count;
    insertAt(e,pos);
}
```

Cada vez que se insere, procura-se a posição correcta para o elemento de forma a preservar a ordem dos elementos no vector.

Procura-se o primeiro elemento com um valor superior ou igual a `e`. A inserção tem de ser feita nessa posição.

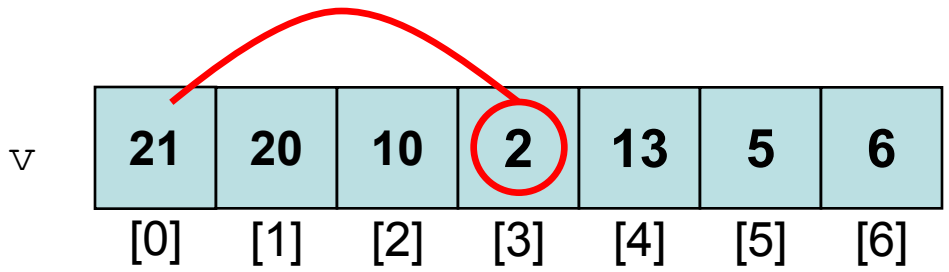
Operações em Vectores
(Ordenação)

- Um dos algoritmos de ordenação mais simples é o algoritmo de *Bubble Sort*.
- Suponha que se pretende ordenar o seguinte vector:



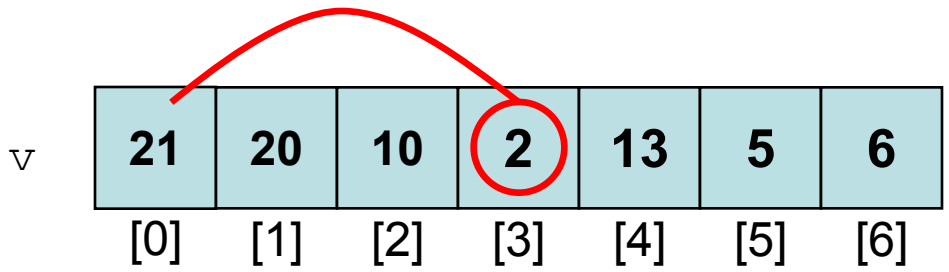
Operações em Vectores
(Ordenação)

- Vamos começar por tentar empurrar o valor mais baixo para a posição mais à esquerda...
- Mas como?



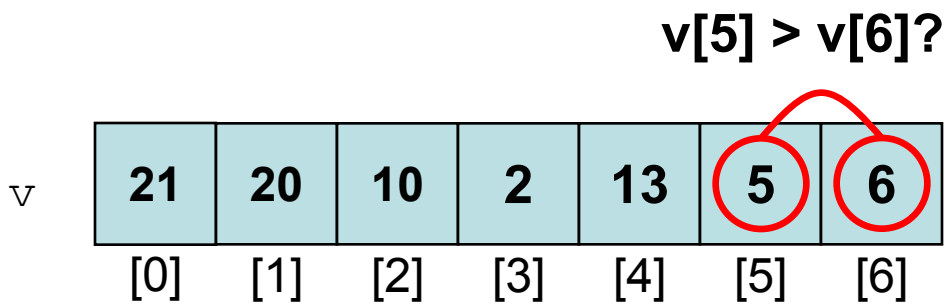
Operações em Vectores
(Ordenação)

Varrendo o vector da direita para a esquerda.



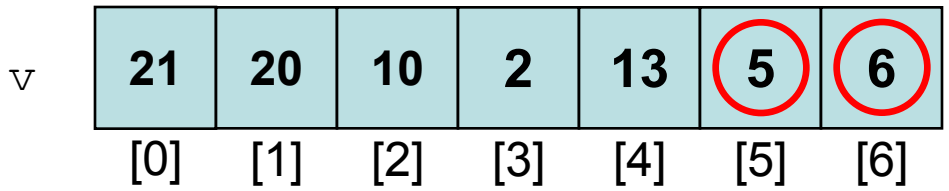
Operações em Vectores
(Ordenação)

- O algoritmo *Bubble Sort* resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos e se estiverem fora de ordem trocam de posição



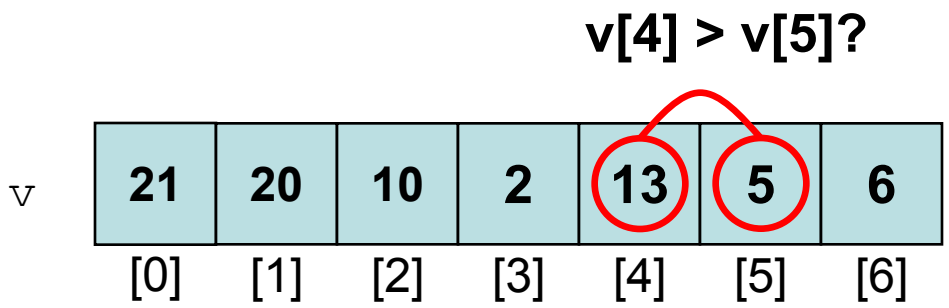
Operações em Vectores
(Ordenação)

- O algoritmo *Bubble Sort* resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos e se estiverem fora de ordem trocam de posição



Operações em Vectores
(Ordenação)

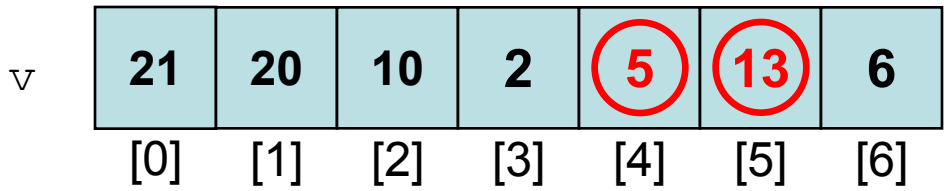
- O algoritmo *Bubble Sort* resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda



Operações em Vectores
(Ordenação)

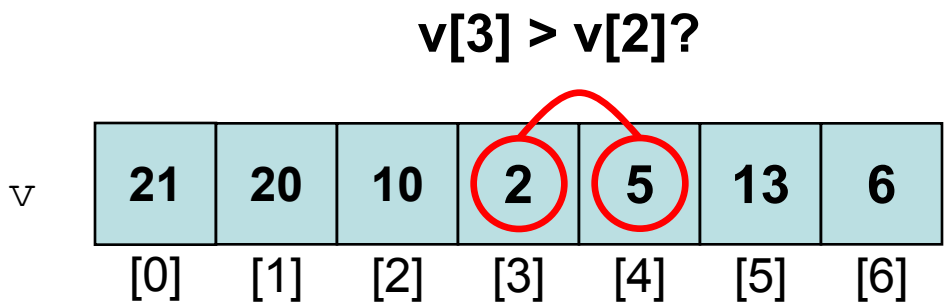
varrendo o vector da direita para a esquerda.

- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda



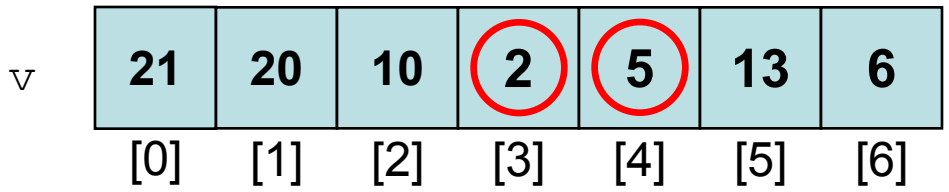
Operações em Vectores
(Ordenação)

- O algoritmo *Bubble Sort* resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda



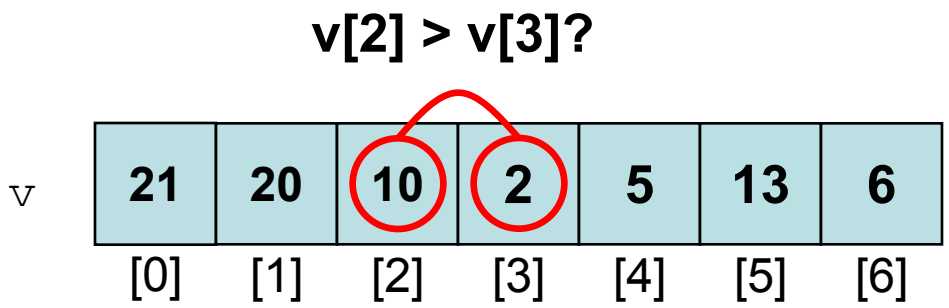
Operações em Vectores
(Ordenação)

- O algoritmo *Bubble Sort* resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda



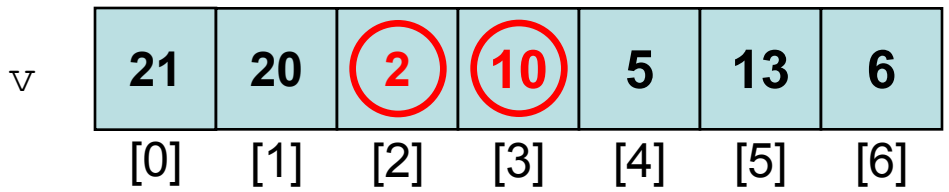
Operações em Vectores
(Ordenação)

- O algoritmo *Bubble Sort* resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda



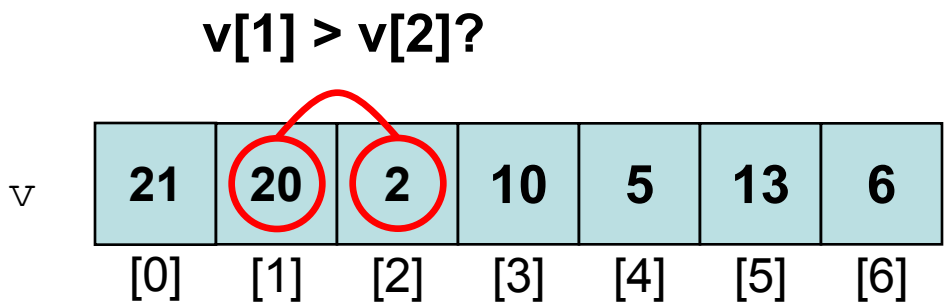
Operações em Vectores
(Ordenação)

- O algoritmo *Bubble Sort* resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda



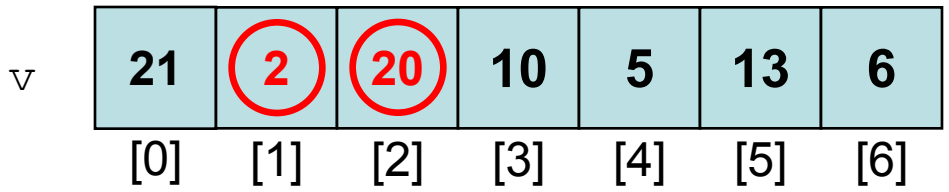
Operações em Vectores
(Ordenação)

- O algoritmo *Bubble Sort* resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda



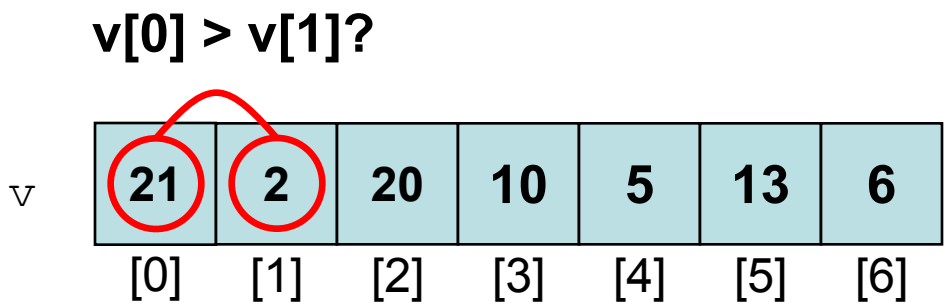
Operações em Vectores
(Ordenação)

- O algoritmo *Bubble Sort* resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda



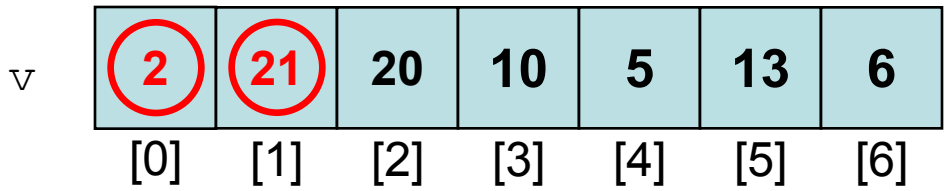
Operações em Vectores
(Ordenação)

- O algoritmo *Bubble Sort* resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda



Operações em Vectores
(Ordenação)

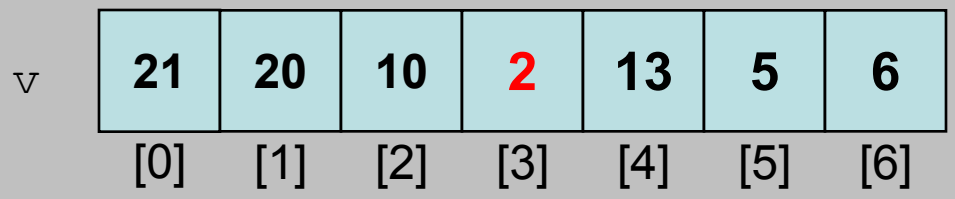
- O algoritmo *Bubble Sort* resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda



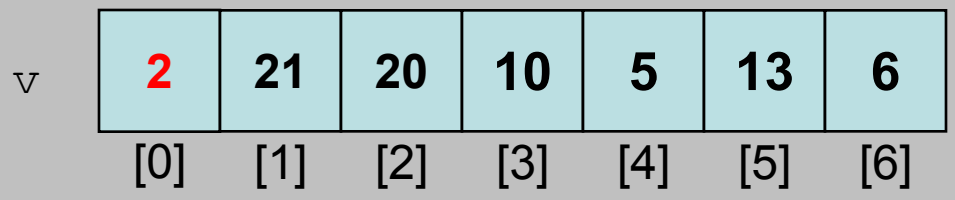
Operações em Vectores
(Ordenação)

- No final do primeiro varrimento temos o menor elemento encostado à esquerda, no seu devido lugar…

Antes

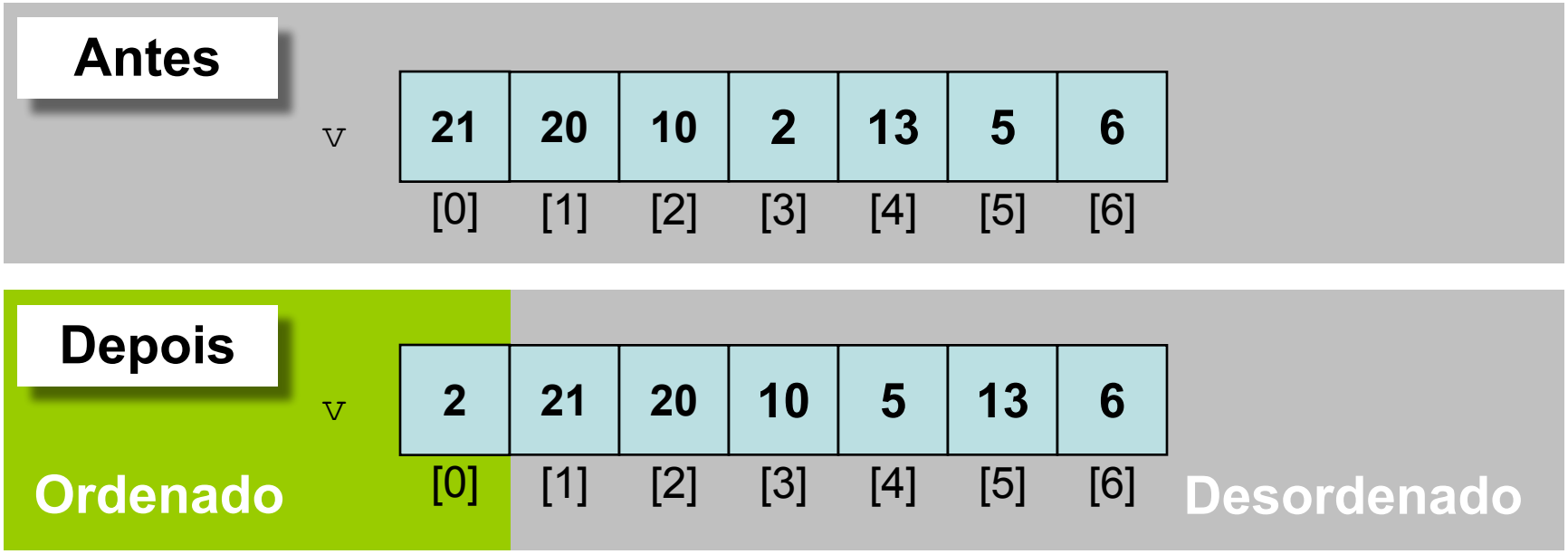


Depois



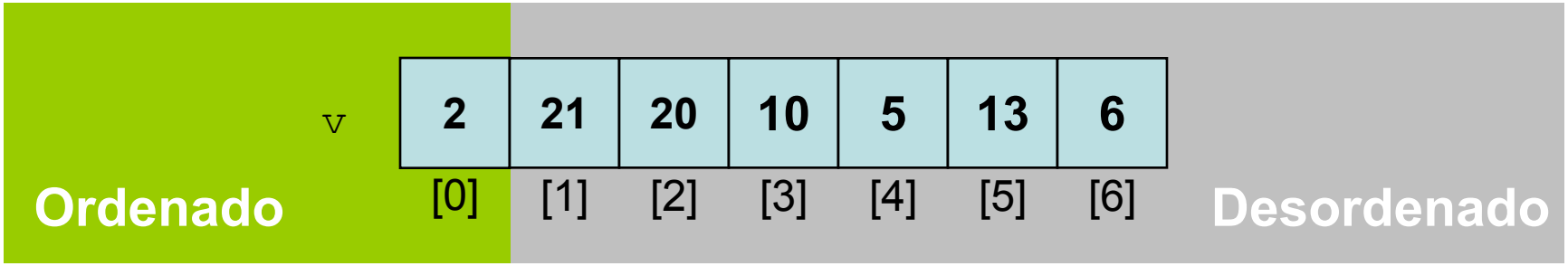
Operações em Vectores
(Ordenação)

- E estamos mais perto da solução pois já temos uma parte do vector ordenado…
- Neste caso até temos mais elementos na sua posição correcta mas foi por mero acaso ☺



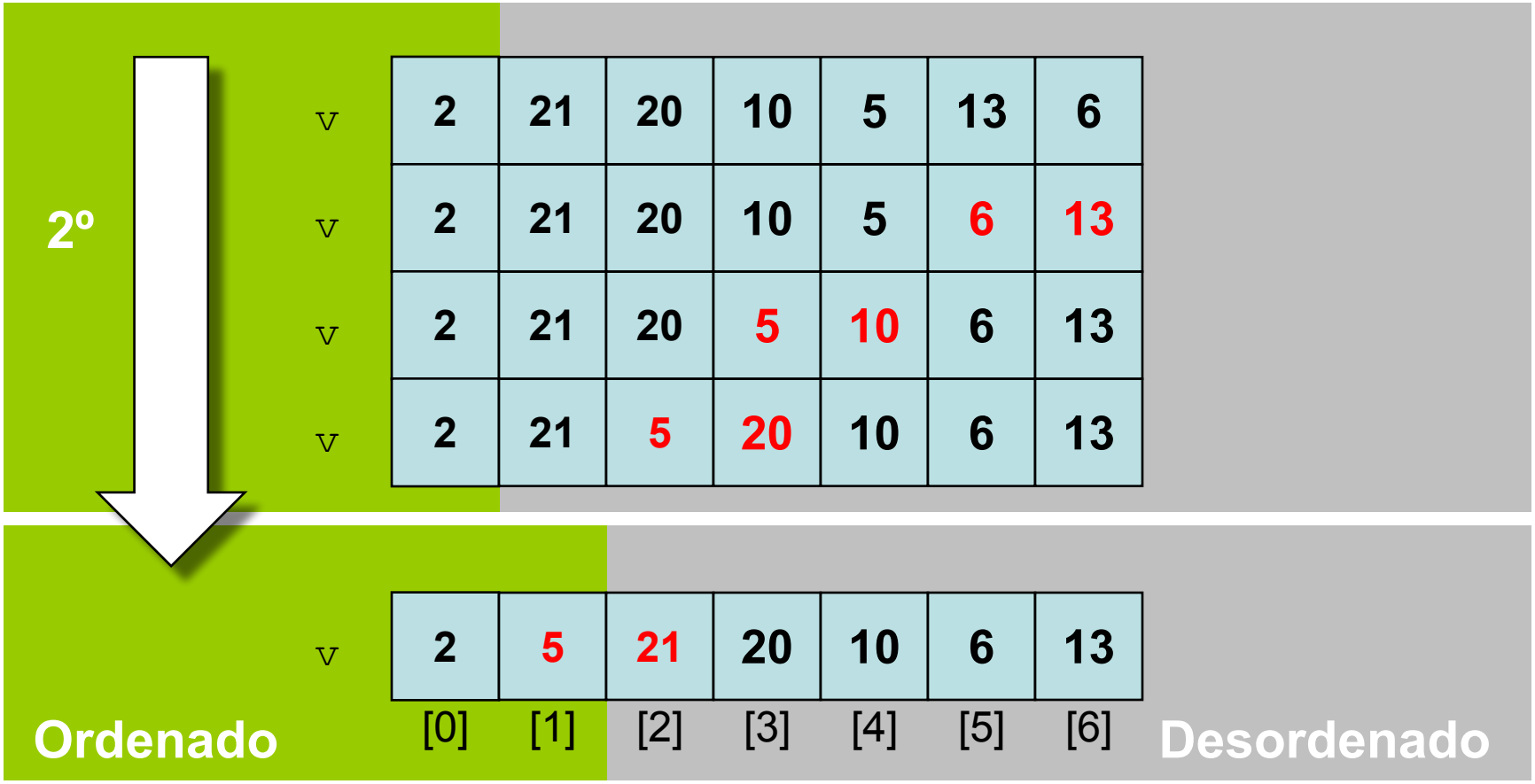
Operações em Vectores
(Ordenação)

- Como fazemos para ordenar mais um elemento?
  - Basta repetir o processo mas esquecendo agora o primeiro elemento e actuando como se o vector começasse no índice 1!



Operações em Vectores
(Ordenação)

- Durante o 2º varrimento o vector passará pelos seguintes estados:

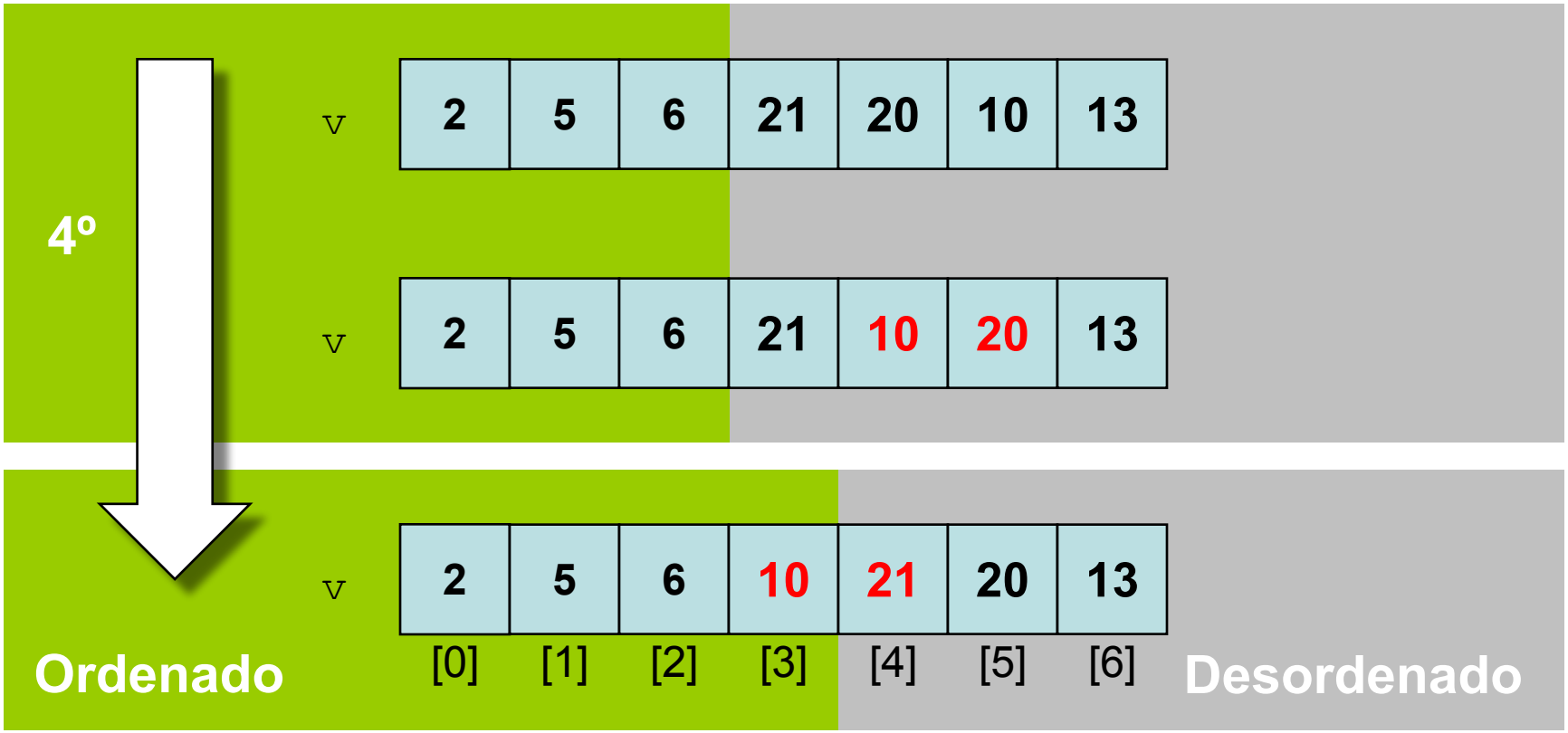


Operações em Vectores
(Ordenação)

- Durante o 3º varrimento o vector passará pelos seguintes estados:

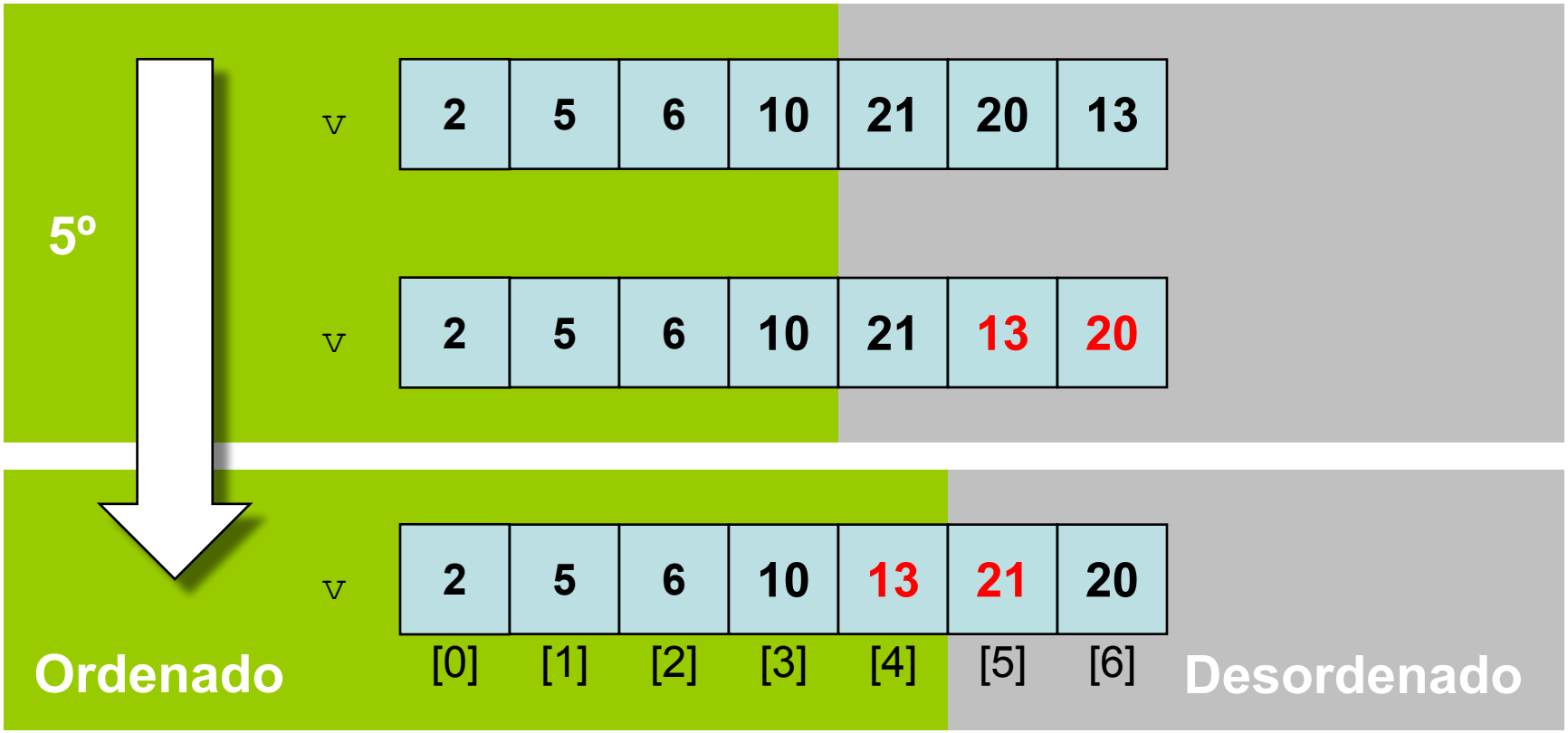
Operações em Vectores
(Ordenação)

- Durante o 4º varrimento o vector passará pelos seguintes estados:



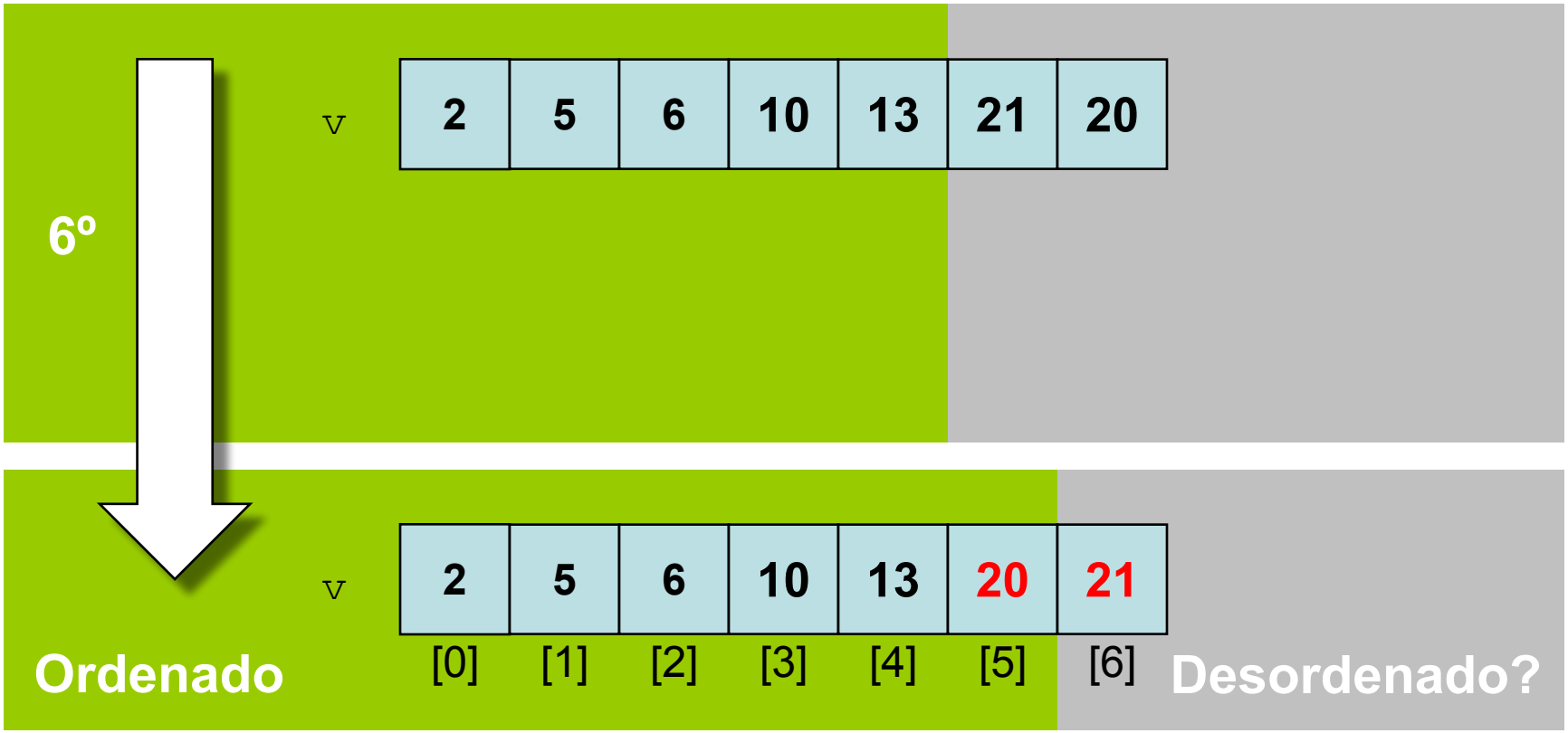
Operações em Vectores
(Ordenação)

- Durante o 5º varrimento o vector passará pelos seguintes estados:



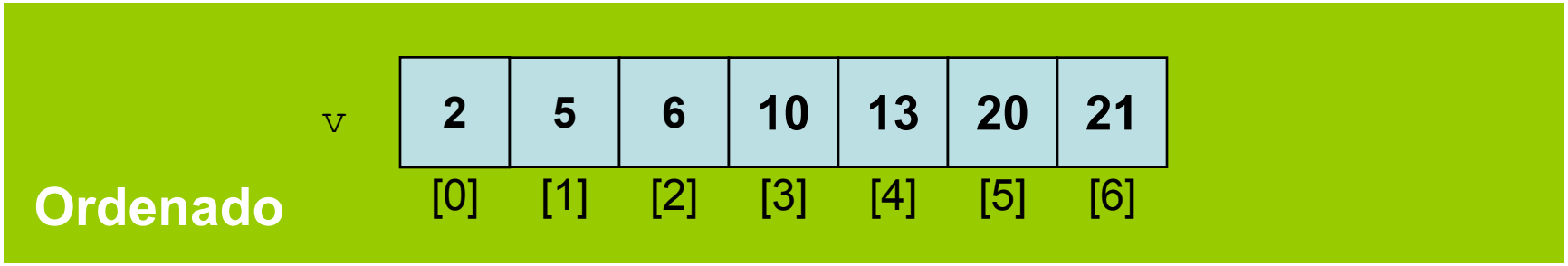
Operações em Vectores
(Ordenação)

- Durante o 6º varrimento o vector passará pelos seguintes estados:



Operações em Vectores
(Ordenação)

- Após o 6º varrimento, um vector de comprimento 7 estará ordenado!
- Para ordenar um vector de comprimento `count` serão necessários `count-1` varrimentos!



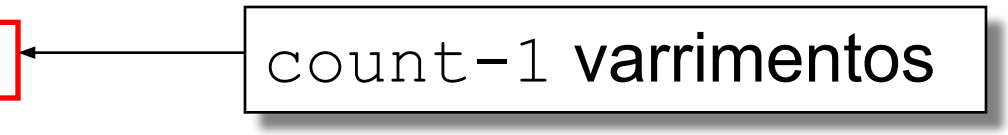
Operações em Vectores
(Ordenação)

- Para ordenar um vector de comprimento `count` são necessários `count-1` varrimentos!

```
public void bubbleSort() {
   for(int i=1; i<count; i++)

        ...
        // Falta programar aqui cada varrimento...
        ...


}
```



Operações em Vectores
(Ordenação)

- Para ordenar um vector de comprimento `count` são necessários `count-1` varrimentos!

```
public void bubbleSort() {
  for(int i=1; i<count; i++)
    for(int j=count-1; j>=?; j--){
      if(v[j-1] > v[j]) {

        ...

      }
    }
}
```

Cada varrimento inicia-se no final do vector

Em cada passo comparam-se dois elementos entre si.

Operações em Vectores
(Ordenação)

- Para ordenar um vector de comprimento `count` são necessários `count-1` varrimentos!

```
public void bubbleSort() {
  for(int i=1; i<count; i++)
    for(int j=count-1; j>=?; j--){
      if(v[j-1] > v[j]) {

        ...

      }
    }
}
```

No 1º (i=1) varrimento, `j` deverá descer até ao valor 1, de modo a comparar v[0] com v[1].

No 2º (i=2) varrimento, `j` deverá descer até ao valor 2, de modo a comparar v[1] com v[2].

Operações em Vectores
(Ordenação)

- Para ordenar um vector de comprimento `count` são necessários `count-1` varrimentos!

```
public void bubbleSort() {
  for(int i=1; i<count; i++)
    for(int j=count-1; j>=i; j--){
      if(v[j-1] > v[j]) {

        ...

      }
    }
}
```

Em cada varrimento, `j` deverá descer até ao valor de `i`!!!

Operações em Vectores
(Ordenação)

- Para ordenar um vector de comprimento `count` são necessários `count-1` varrimentos!

```
public void bubbleSort() {
  for(int i=1; i<count; i++)
    for(int j=count-1; j>=i; j--){
      if(v[j-1] > v[j]) {
        type tmp = v[j-1];
        v[j-1] = v[j];
        v[j] = tmp;
      }
    }
}
```

Cada vez que se encontram 2 elementos fora de ordem, trocam-se de posição…

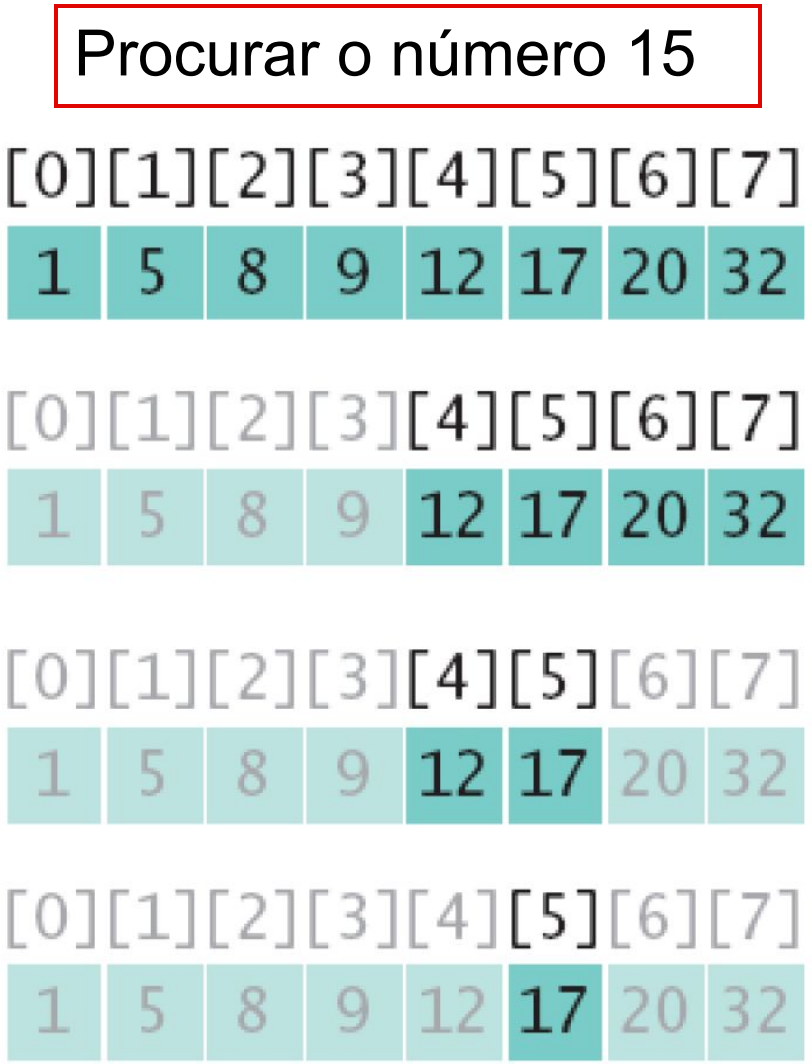
Operações em Vectores
(Ordenação)

- Eis o algoritmo de ordenação *Bubble Sort*:

```
public void bubbleSort() {
  for(int i=1; i<count; i++)
    for(int j=count-1; j>=i; j--){
      if(v[j-1] > v[j]) {
        type tmp = v[j-1];
        v[j-1] = v[j];
        v[j] = tmp;
      }
    }
}
```

Pesquisa Binária

- Localiza um valor num vector ordenado, da seguinte forma:
  - Determina se o valor está na primeira ou na segunda metade do vector
  - Repete a pesquisa para uma das metades



Operações em Vectores
(Pesquisa Binária)

```
public boolean binarySearch(type e) {

   return ...;
}
```

O elemento a procurar (`e`), do tipo *`type`*, é passado como parâmetro.

Operações em Vectores
(Pesquisa Binária)

```
public boolean binarySearch(type e) {
   boolean found = false;

   return found;
}
```

Durante a pesquisa usaremos uma variável booleana (`found`) para indicar se já se encontrou o elemento a pesquisar.

No final, a variável deverá conter o resultado da pesquisa.

Operações em Vectores
(Pesquisa Binária)

```
public boolean binarySearch(type e) {
   boolean found = false;
   int low=0, high=count-1;
```

As variáveis `low` e `high` delimitam a parte do vector onde a busca incide.

No início, todo o vector deverá ser pesquisado:
v[0]…v[count-1]

```
   return found;
}
```

Operações em Vectores
(Pesquisa Binária)

```
public boolean binarySearch(type e) {
   boolean found = false;
   int low=0, high=count-1;

   while(!found ...) {









   }
   return found;
}
```

A pesquisa deverá prosseguir enquanto `found` valer `false`.

O mesmo será dizer: "enquanto não encontrado", traduzindo a linha de código para linguagem natural! ☺

Operações em Vectores
(Pesquisa Binária)

```
public boolean binarySearch(type e) {
  boolean found = false;
  int low=0, high=count-1;

  while(!found && ...) {









  }
  return found;
}
```

Mas, no caso do elemento a pesquisar não estar contido no vector, a variável `found` nunca valerá `true`, pelo que será necessário terminar a busca com base noutra condição…

Operações em Vectores
(Pesquisa Binária)

```
public boolean binarySearch(type e) {
   boolean found = false;
   int low=0, high=count-1;

   while(!found && low <= high) {









   }
   return found;
}
```

No caso do elemento a pesquisar não pertencer ao vector, o intervalo de pesquisa deverá, no final, ser vazio: não há mais nenhuma metade onde procurar!

Operações em Vectores
(Pesquisa Binária)

```
public boolean binarySearch(type e) {
   boolean found = false;
   int low=0, high=count-1;

   while(!found && low <= high) {
      int mid = (low + high)/2;









   }
   return found;
}
```

Em cada etapa, começa-se por determinar o índice do elemento central do intervalo de busca.

Se `mid` representar o índice, `v[mid]` representa o valor central do vector.

Operações em Vectores
(Pesquisa Binária)

```
public boolean binarySearch(type e) {
  boolean found = false;
  int low=0, high=count-1;

  while(!found && low <= high) {
    int mid = (low+high)/2;
    if(v[mid]==e) found = true;
    else ...




  }
  return found;
}
```

Tenta-se a sorte! ☺

Poderá dar-se o caso do elemento a pesquisar estar, precisamente, nessa posição. Nesse caso assinalamos o sucesso da pesquisa.

Operações em Vectores
(Pesquisa Binária)

```
public boolean binarySearch(type e) {
  boolean found = false;
  int low=0, high=count-1;

  while(!found && low <= high) {
    int mid = (low+high)/2;
    if(v[mid]==e) found = true;
    else ...

  }
  return found;
}
```

Caso contrário é preciso verificar em qual das metades se deverá pesquisar o valor `e`.

Operações em Vectores
(Pesquisa Binária)

```
public boolean binarySearch(type e) {
  boolean found = false;
  int low=0, high=count-1;

  while(!found && low <= high) {
    int mid = (low+high)/2;
    if(v[mid]==e) found = true;
    else if(e < v[mid])
      ...
    else // e > v[mid]
      ...
  }
  return found;
}
```

Se e<v[mid] o elemento apenas poderá estar na primeira metade do vector: [low]..[mid-1]

Operações em Vectores
(Pesquisa Binária)

```
public boolean binarySearch(type e) {
  boolean found = false;
  int low=0, high=count-1;

  while(!found && low <= high) {
    int mid = (low+high)/2;
    if(v[mid]==e) found = true;
    else if(e < v[mid])
      ...
    else // e > v[mid]
      ...
  }
  return found;
}
```

Se e>v[mid] o elemento apenas poderá estar na segunda metade do vector: [mid+1]..[high]

Operações em Vectores
(Pesquisa Binária)

```
public boolean binarySearch(type e) {
  boolean found = false;
  int low=0, high=count-1;

  while(!found && low <= high) {
    int mid = (low+high)/2;
    if(v[mid]==e) found = true;
    else if(e < v[mid])
      high = mid-1;
    else // e > v[mid]
      low = mid+1;
  }
  return found;
}
```

Basta ajustar os limites da pesquisa em cada um dos casos e prosseguir…

Operações em Vectores
(Pesquisa Binária)

```
public boolean binarySearch(type e) {
  boolean found = false;
  int low=0, high=count-1;

  while(!found && low <= high) {
    int mid = (low+high)/2;
    if(v[mid]==e) found = true;
    else if(e < v[mid])
      high = mid-1;
    else // e > v[mid]
      low = mid+1;
  }
  return found;
}
```

Razões para querer ordenar vectores

Eis duas razões importantes para querer ordenar vectores:

- Em algumas situações, interessa apresentar ao utilizador dados ordenados para estes ficarem mais fáceis de entender.
- Ainda mais importante é a aceleração das pesquisas, algo absolutamente essencial no caso de vectores grandes.

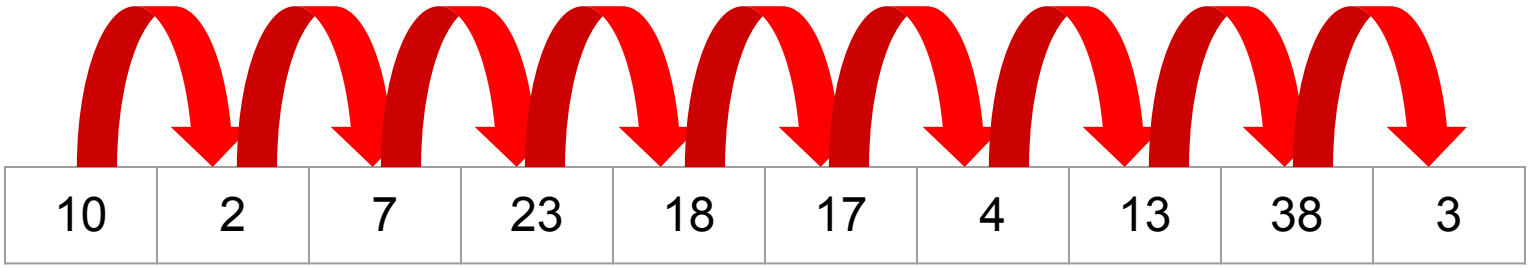
Razões para querer ordenar vectores

É dramática a aceleração das pesquisas que a ordenação proporciona! Considere um vector contendo **1 milhão** de elementos:

- **Se o vetor não estiver ordenado**, somos obrigados a usar pesquisa linear e o número de comparações efetuadas será, **em média, quinhentos mil.**
  - No pior dos casos, quando temos de verificar todas as posições do vector para concluir que o valor que procuramos não existe, temos de **comparar todos os elementos do vector com esse valo**r
- **Se o vector estiver ordenado**, então podemos usar pesquisa binária e o número de comparações efetuadas será **sempre inferior a 24**.
  - Em cada iteração do ciclo, **metade** da zona relevante do vector, que ainda se mantém em análise, é descartada, ou seja a dimensão do vector em análise é dividida por 2

Pesquisa sequencial - vector não ordenado

À procura do número 27



Considere um vector contendo **1 milhão** de elementos:

- **Se o vetor não estiver ordenado**, somos obrigados a usar pesquisa linear e o número de comparações efetuadas será, **em média, quinhentos mil.**
  - No pior dos casos, quando temos de verificar todas as posições do vector para concluir que o valor que procuramos não existe, temos de **comparar todos os elementos do vector com esse valor**

Pesquisa binária - vector ordenado

Considere um vector contendo **1 milhão** de elementos:

- **Se o vector estiver ordenado**, então podemos usar pesquisa binária e o número de comparações efetuadas será **sempre inferior a 24**.

Em cada iteração do ciclo, **metade** da zona relevante do vector, que ainda se mantém em análise, é descartada, ou seja a dimensão do vector em análise é dividida por 2

1 milhão = 1000000 = $10^6 < 16^6 = (2^4)^6 = 2^{24}$

Procurar o número 15