Agenda de Contactos

(Ordenação e Pesquisa)

Agenda de Contactos

- Alterar o nosso interpretador de comandos, de modo a que o comando LC apresente a listagem de todos os contactos por ordem alfabética do nome do contacto.
  - Que alterações são necessárias?
    - Solução A: É necessário alterar o nosso iterador, de modo a ordenar os contactos;
    - Solução B: O vector de contactos está sempre ordenado por ordem alfabética do nome.

Agenda de Contactos
Solução A

- Operações a alterar (classe ContactBook):
  - Para percorrer a informação referente a todos os contactos tinhamos as seguintes operações.

`public void` `initializeIterator()`

Método que inicia a iteração sobre os contactos da agenda

`public` `boolean hasNext()`

Método que indica se existe um contacto seguinte.

Quando se chega ao fim da travessia, devolve false, caso contrário devolve true

`public` `Contact next()`

Método que devolve a informação referente ao contacto corrente

  - Como queremos a listagem por ordem alfabética de nome, é necessário ordenar os contactos no momento de início da iteração (método `initializeIterator`)

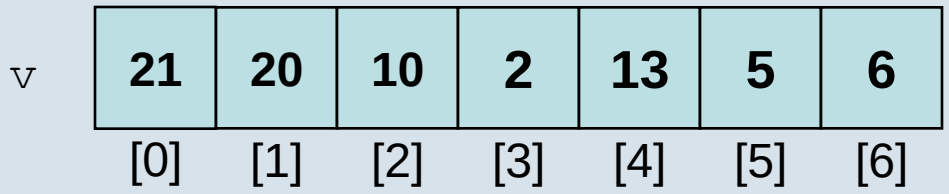
Agenda de Contactos
Solução A

```
public class ContactBook{
    private Contact[] contacts; // vector de contactos
    private int counter; // número de contactos no vector
    private int currentContact; // posição do contacto corrente
    ...
    public void initializeIterator(){
        if (counter > 0){
            this.sortContacts();
            currentContact = 0;
        }
        else currentContact = -1
    }
    ...
}
```

Odenação dos contactos

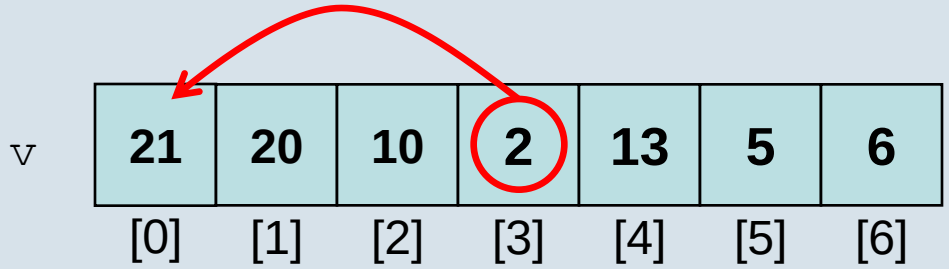
Operações em Vectores
(Ordenação)

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



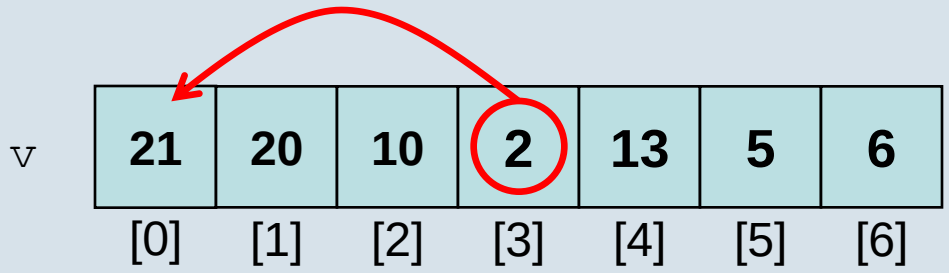
Operações em Vectores
(Ordenação)

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



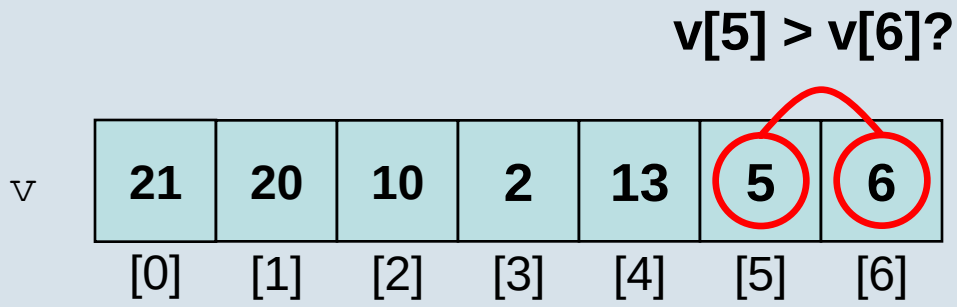
Operações em Vectores
(Ordenação)

- O algoritmo Bubble Sort resolve este problema varrendo o vector da direita para a esquerda.



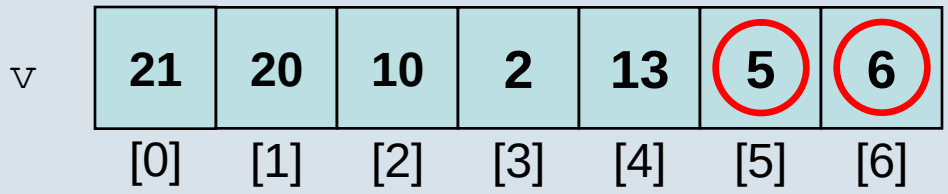
Operações em Vectores
(Ordenação)

- 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



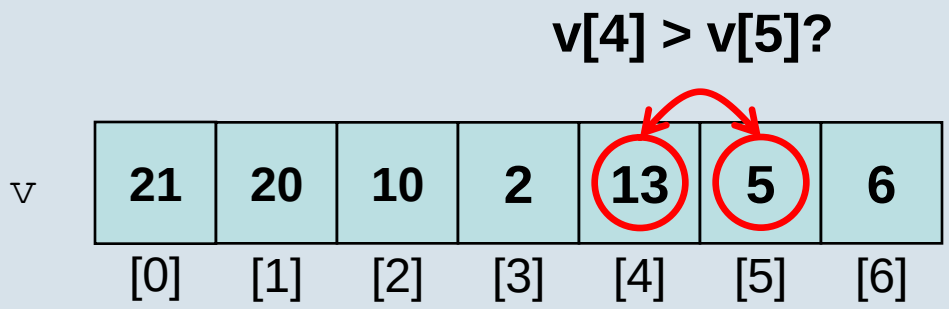
Operações em Vectores
(Ordenação)

- 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



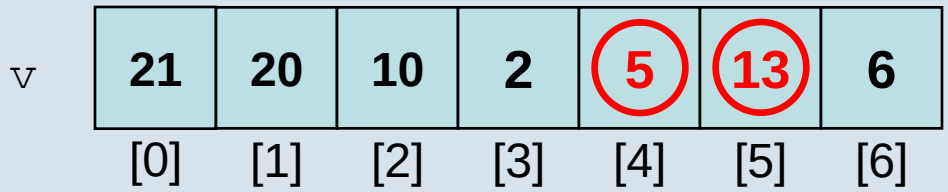
Operações em Vectores
(Ordenação)

- 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



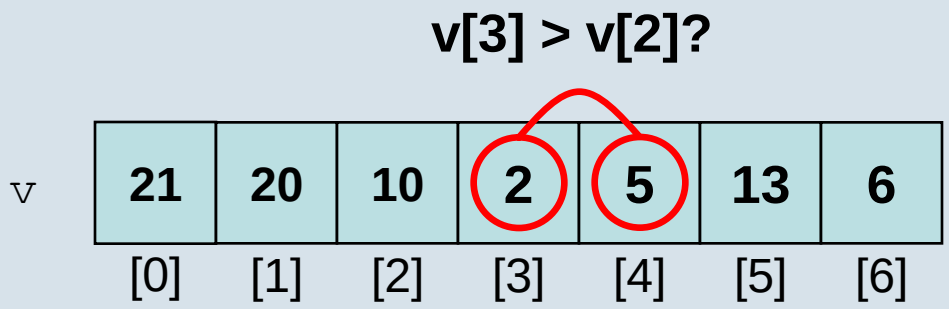
Operações em Vectores
(Ordenação)

- 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



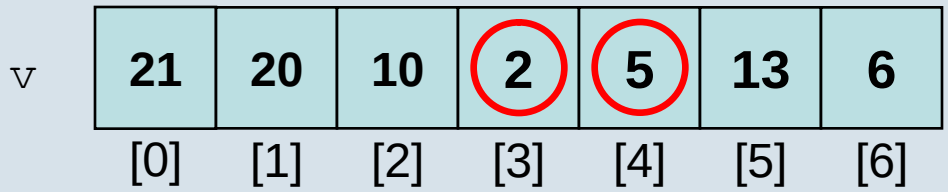
Operações em Vectores
(Ordenação)

- 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



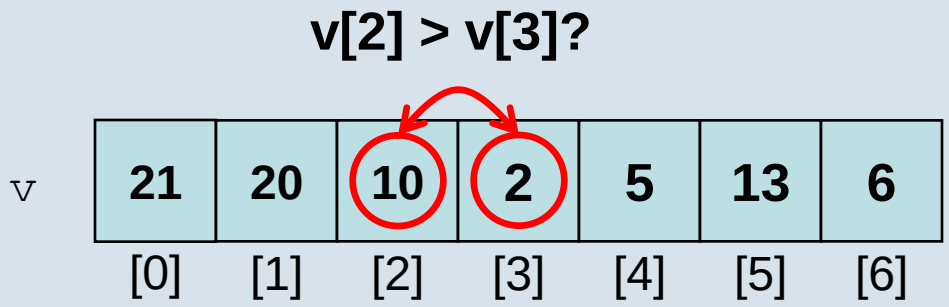
Operações em Vectores
(Ordenação)

- 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



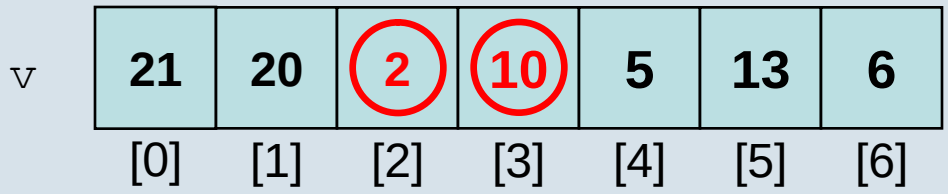
Operações em Vectores
(Ordenação)

- 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



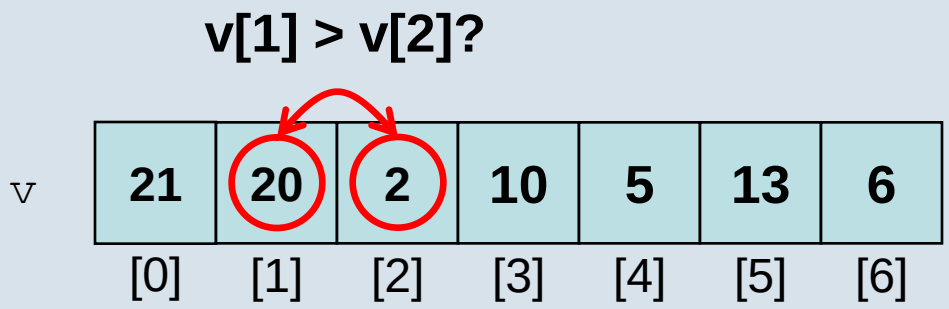
Operações em Vectores
(Ordenação)

- 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



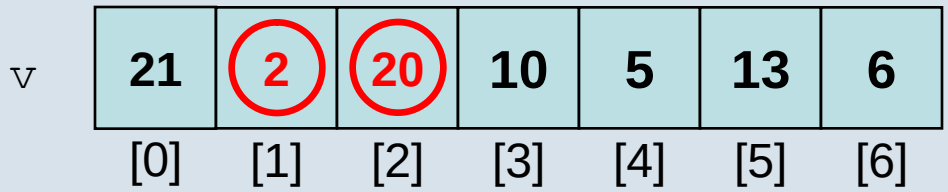
Operações em Vectores
(Ordenação)

- 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



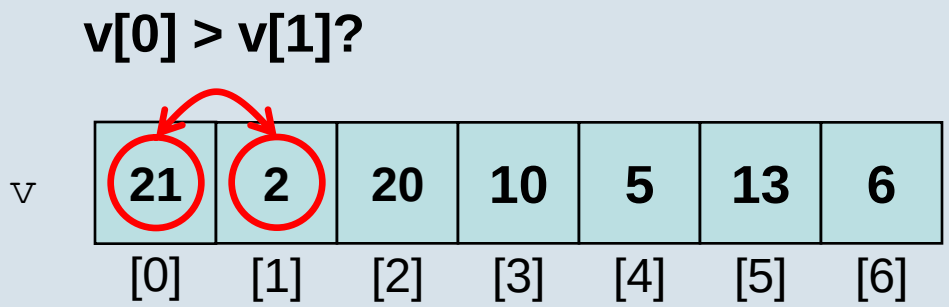
Operações em Vectores
(Ordenação)

- 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



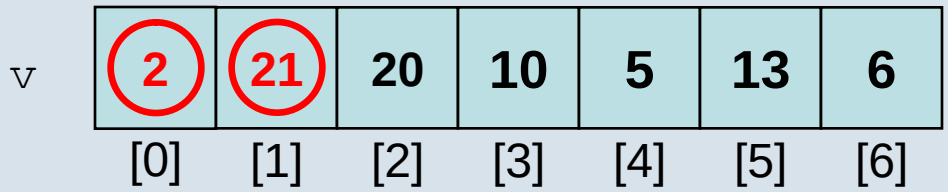
Operações em Vectores
(Ordenação)

- 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)

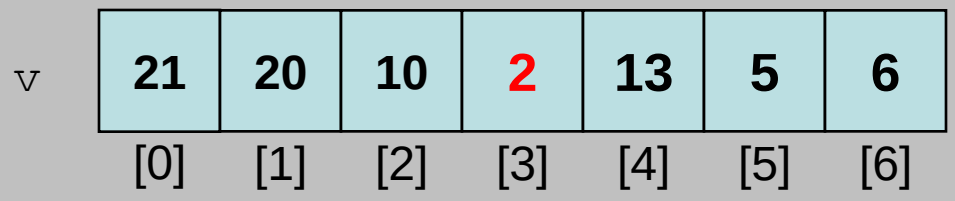
- O algoritmo Bubble Sort resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda



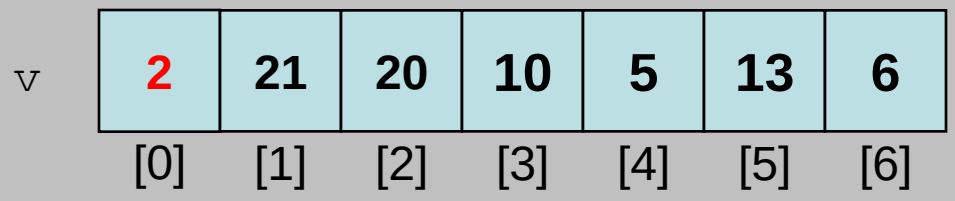
Operações em Vectores
(Ordenação)

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

Antes

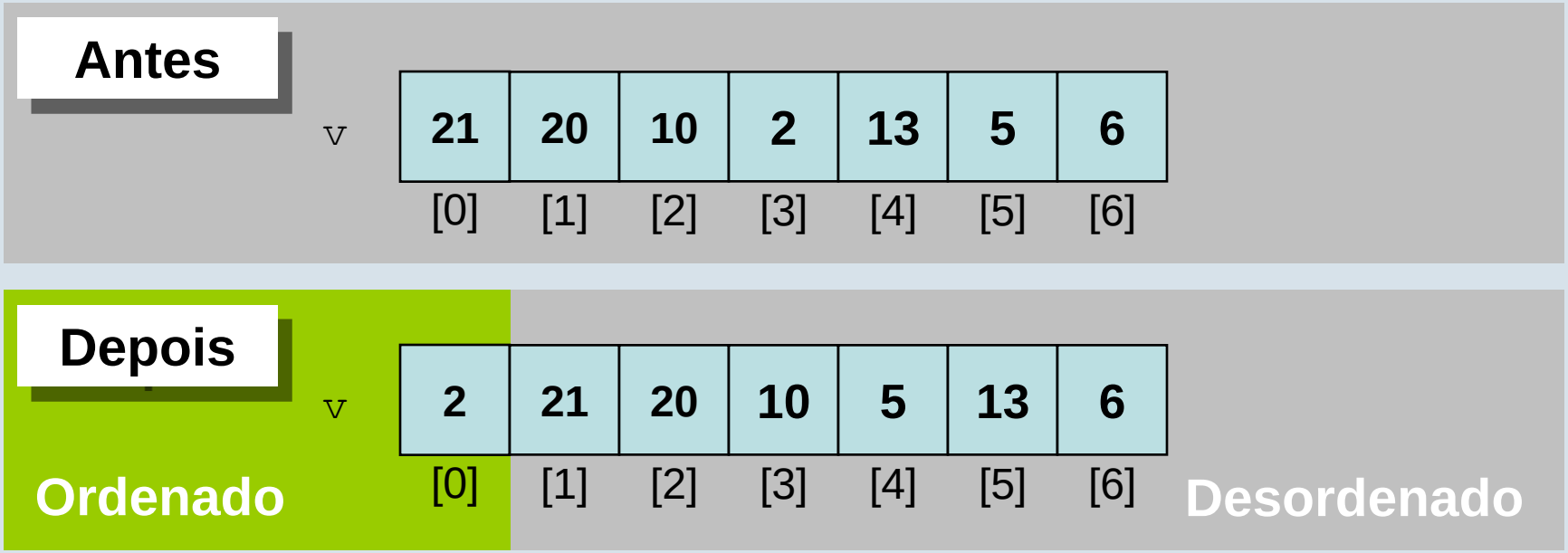


Depois



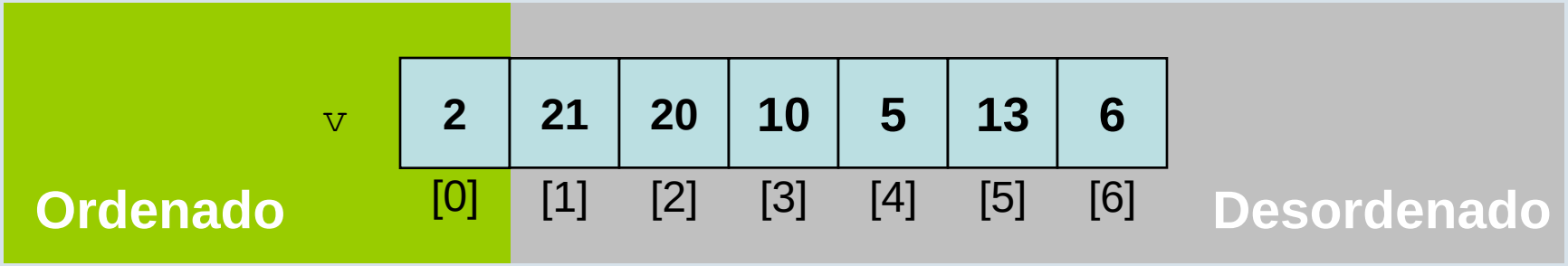
Operações em Vectores
(Ordenação)

- 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)

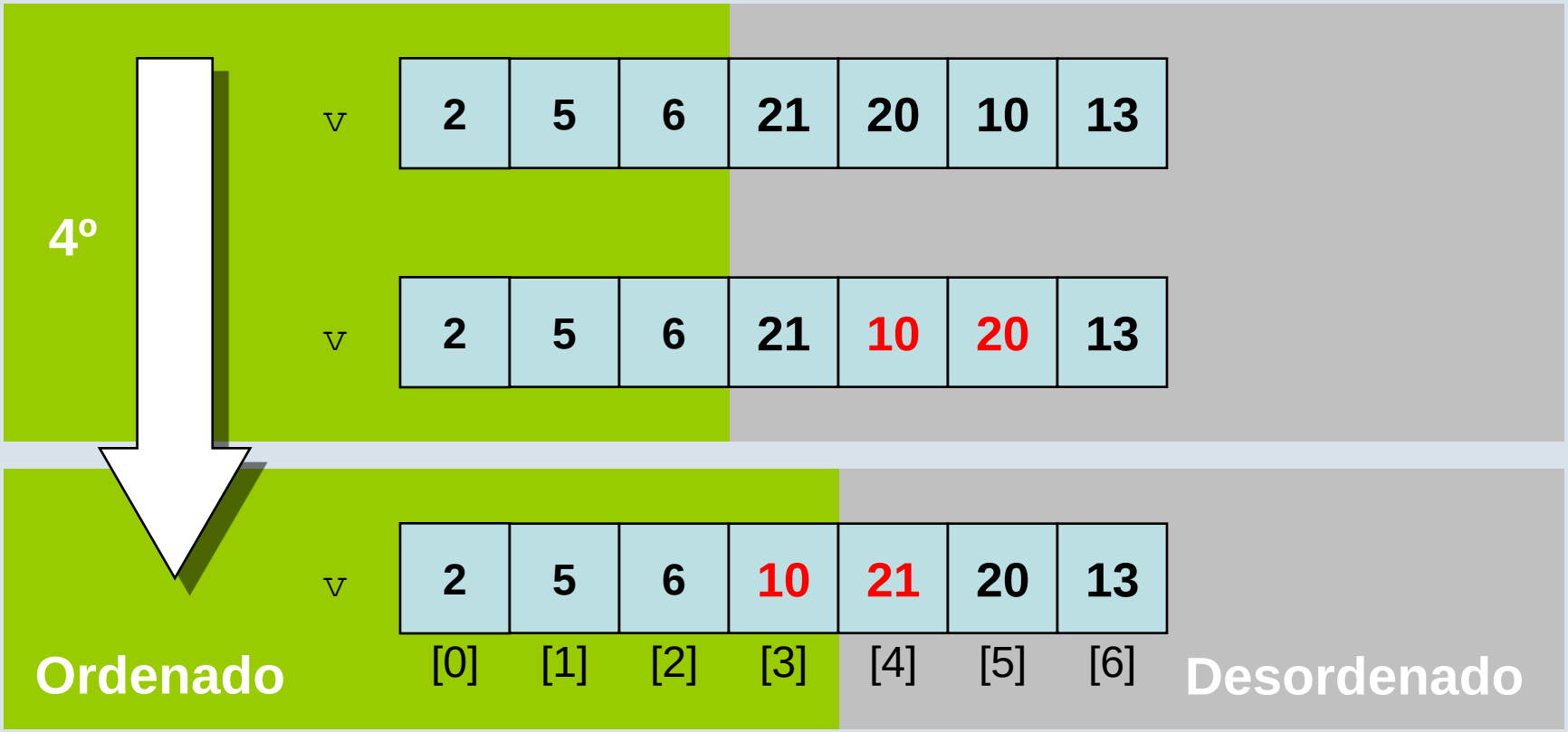
- Durante o 2º varrimento o vector passará pelos seguintes estados:

Operações em Vectores
(Ordenação)

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

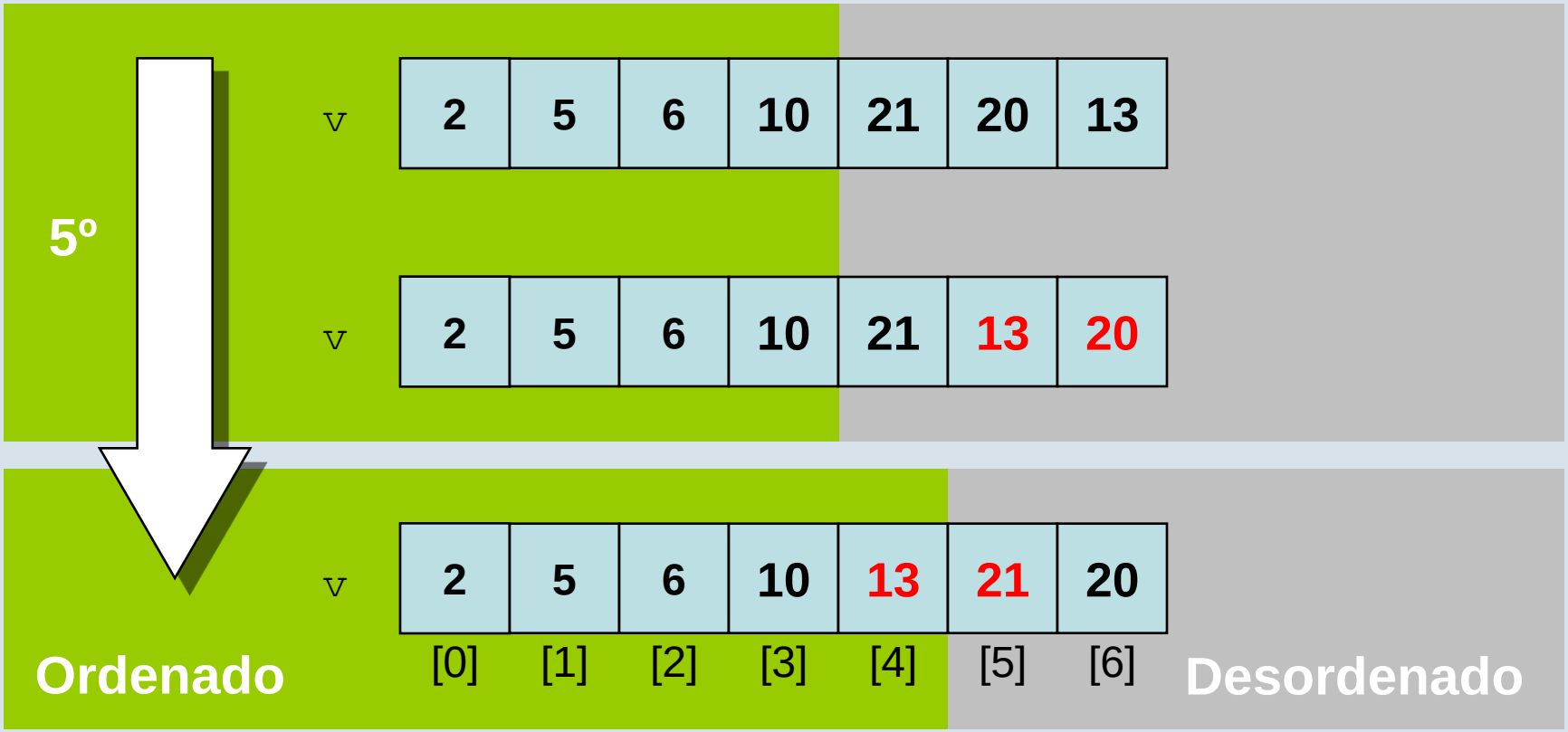
Operações em Vectores
(Ordenação)

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



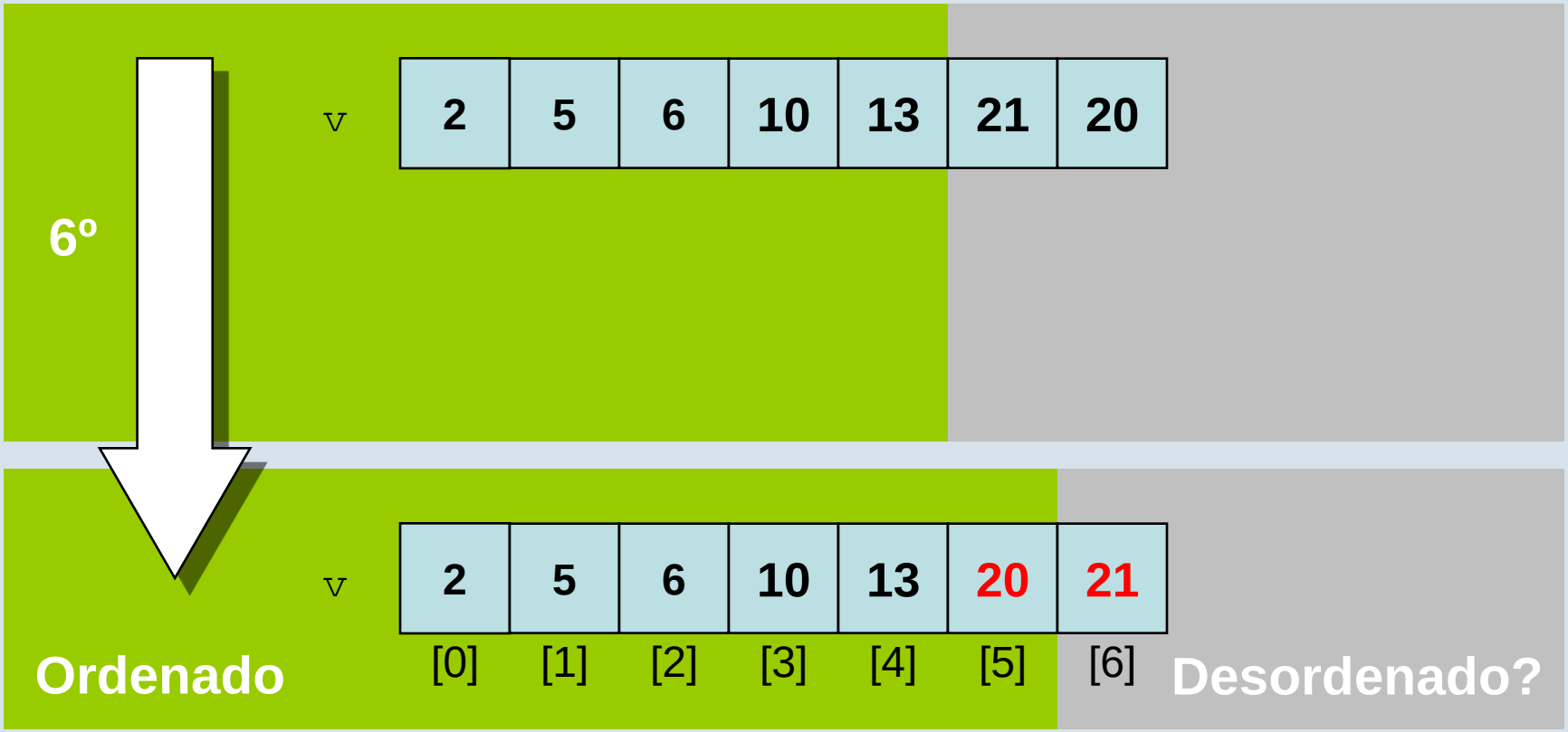
Operações em Vectores
(Ordenação)

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



Operações em Vectores
(Ordenação)

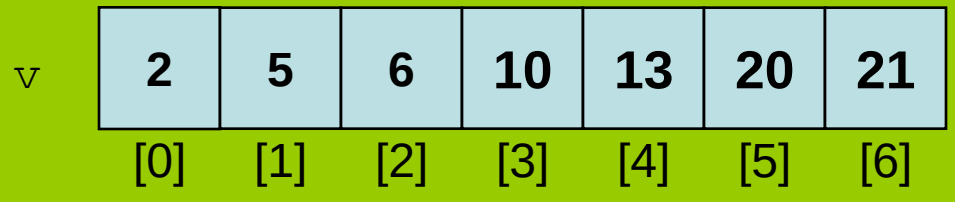
- Durante o 6º varrimento o vector passará pelos seguintes estados:



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!

Ordenado



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...
      ...
}
```

`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++)
     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;
      }
}
```

Agenda de Contactos
(Ordenação)

- Eis o algoritmo de ordenação Bubble Sort para um vector de objectos (agenda):

```
public class ContactBook{
   ...
   private Contact[] contacts; // vector de contactos
   private int counter; // número de contactos no vector
   ...
   private  void  sortContats() {
      for(int i=1; i<counter; i++)
         for(int j=counter-1; j>=i; j--)
            if(contacts[j-1].compareTo(contacts[j]) > 0) {
                     Contact tmp = contacts[j-1];
                     contacts[j-1] = contacts[j];
                     contacts[j] = tmp;
            }
   }
   ...
}
```

Agenda de Contactos
(Ordenação)

- Eis o algoritmo de ordenação Bubble Sort para um vector de objectos (agenda):

```
public class ContactBook{
   ...
   private Contact[] contacts; // vector de contactos
   private int counter; // número de contactos no vector
   ...
   private  void sortContats() {
      for(int i=1; i<counter; i++)
         for(int j=counter-1; j>=i; j--)
            if(contacts[j-1].compareTo(contacts[j]) > 0) {
                        Contact tmp = contacts[j-1];
                        contacts[j-1] = contacts[j];
                        contacts[j] = tmp;
            }
   }
   ...
}
```

Comparação de dois objectos da classe Contact (método `compareTo`)

Objectos
(Função de comparação)

- Em relação aos operadores relacionais (<,<=, >, >=), os mesmos não se podem aplicar a objectos, pelo que será necessário usar uma função (método) para o efeito.
- A convenção normalmente usada na linguagem Java consiste em programar um método de comparação…

Objectos
(Função de comparação)

- Método de comparação entre dois objectos de uma mesma classe *type*:

```
public int compareTo(type other)
```

Valor a retornar pelo método:

>0	se `this` for considerado "maior" que `other`
<0	se `this` for considerado "menor" que `other`
0	restantes situações (`this` igual a `other`)

Método compareTo na classe `Contact`

- A comparação entre dois contactos é realizada com base no nome do contacto (ordenação alfabética).

```
public class Contact {
   private String name;
   private int phone;
   private String email;

   ...
   public int compareTo(Contact other) {
     return name.compareTo(other.getName());
   }
}
```

Consultar a classe `String` para obter informação sobre o método `compareTo`

Agenda de Contactos

- Alterar o nosso interpretador de comandos, de modo a que o comando LC apresente a listagem de todos os contactos por ordem alfabética do nome do contacto.
  - Que alterações são necessárias?
    - Solução A: É necessário alterar o nosso iterador, de modo a ordenar os contactos; ✔
    - Solução B: O vector de contactos está sempre ordenado por ordem alfabética do nome.

Agenda de Contactos
Solução B

- Operações a alterar (classe ContactBook):
  - O método que insere um novo contacto, deve fazer uma inserção de modo a manter a ordenação desejada.

```
public void addContact(String name, int phone, String email)
```

  - O método que procura um dado contacto por nome deve ser alterado de modo a fazer uma pesquisa mais “inteligente”, já que os contactos estão ordenados.

```
private int searchIndex(String name)
```

  - Caso, no método que remove um dado contacto tenham optado por colocar na posição desse contacto o último contacto da agenda. Então devem alterar o método de modo a manter a ordenação do vector.

```
public void deleteContact(String name)
```

Método addContact

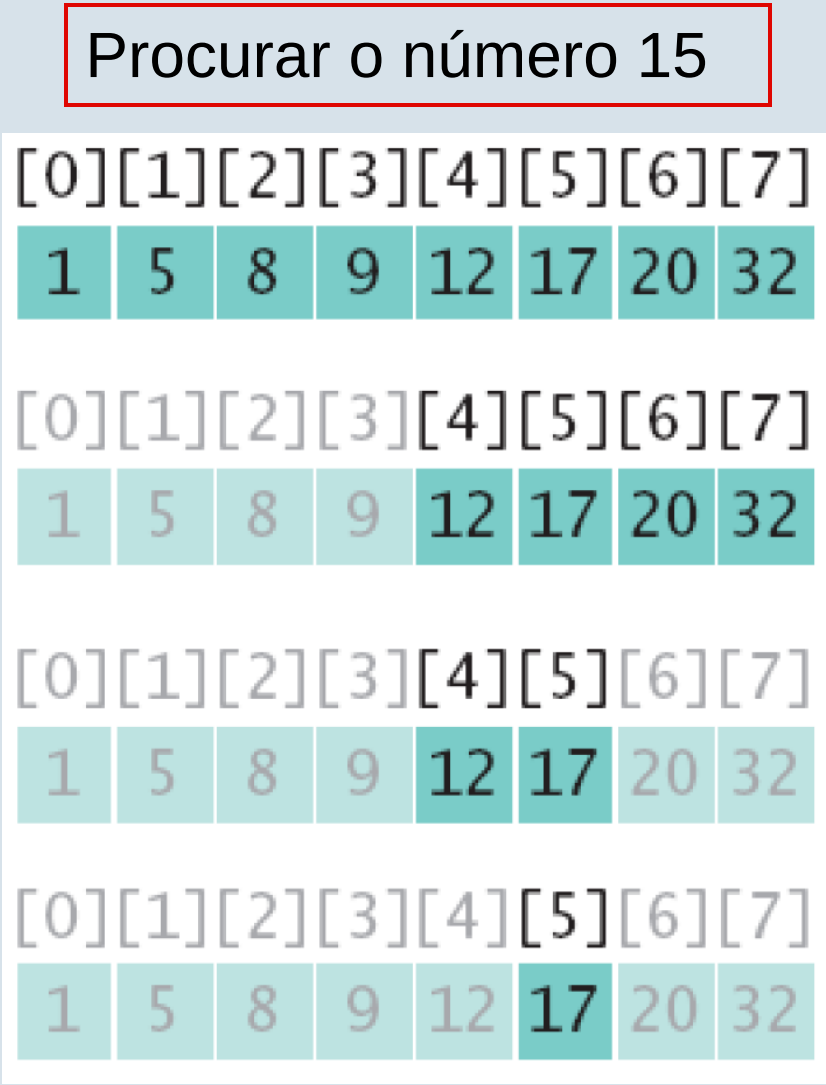
```
public class ContactBook{
    public void addContact(String name, int phone, String email){
      int pos = 0, comp =-1;
      while ((pos < counter) && (comp < 0) {
          comp = contacts[pos].getName().compareTo(name);
          if (comp < 0) pos++;
      }
      if (comp != 0) { //não foi encontrado, e' um novo contacto
          if (counter == contacts.length) { // vector cheio
                //código associado a fazer crescer o vector
          }
          for(int i = counter; i > pos; i--)
                contacts[i] = contacts[i-1];
          contacts[pos] = new Contact(name,phone,email);
          counter++;
      }
    }
```

Método searchIndex

```
public class ContactBook{
    ...
    private Contact[] contacts; // vector de Contactos
    private int counter; // número de contactos no vector
    private int currentContact; // posição do contacto corrente num varrimento
    ...
    private int searchIndex(String name){
      // pesquisa binária, já que o vector está ordenado por nome
    }
    ...
}
```

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 int binarySearch(type e) {

}
```

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

Operações em Vectores
(Pesquisa Binária)

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




}
```

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

Operações em Vectores
(Pesquisa Binária)

```
public int 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]

Operações em Vectores
(Pesquisa Binária)

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

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

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

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









  }

}
```

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

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




  }

}
```

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 int 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 ...




  }

}
```

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 int 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 ...

  }

}
```

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

Operações em Vectores
(Pesquisa Binária)

```
public int 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]
      ...
  }

}
```

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

Operações em Vectores
(Pesquisa Binária)

```
public int 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]
      ...
  }

}
```

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

Operações em Vectores
(Pesquisa Binária)

```
public int 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;
  }

}
```

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

Operações em Vectores
(Pesquisa Binária)

```
public int 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;
  }
  if (found) return mid;
   return -1;
}
```

Método searchIndex

```
public class ContactBook{
    private Contact[] contacts; // vector de Contactos
    private int counter; // número de contactos no vector
    ...
    private int searchIndex(String name){
      boolean found=false;
      int low=0, high = counter-1, mid, comp;
      while ( !found && low <= high ) {
              mid = (low+high)/2;
              comp = name.compareTo(contacts[mid].getName());
              if(comp == 0) found = true;
              else if(comp < 0)
                     high = mid-1;
                   else // comp > 0
                     low = mid+1;
      }
      if (found) return mid;
      return -1;
    }
    ...
}
```

Método deleteContact

- Caso, no método que remove um dado contacto tenham optado por colocar na posição desse contacto o último contacto da agenda. Então devem alterar o método de modo a manter a ordenação do vector.

```
public class ContactBook{
    private Contact[] contacts; // vector de contactos
    private int counter; // número de contactos no vector
    ...
    public void deleteContact(String name){
     int index = searchIndex(name);
     if (index != -1) { // contacto existente
          int i=index;
          while (i < counter-1) {
               contacts[i] = contacts[i+1];
               i++;
          }
          counter--;
    }
...
```

Agenda de Contactos

- Alterar o nosso interpretador de comandos, de modo a que o comando LC apresente a listagem de todos os contactos por ordem alfabética do nome do contacto.
  - Que alterações são necessárias?
    - Solução A: É necessário alterar o nosso iterador, de modo a ordenar os contactos; ✔
    - Solução B: O vector de contactos está sempre ordenado por ordem alfabética do nome. ✔