Almost all really new ideas have a certain aspect of foolishness when they are first produced.
Alfred North Whitehead

Estruturas de Dados

Ficha 2 - Pesquisa binária e variações

Engenharia Informática

3º ano, 1º semestre

Algoritmos de pesquisa

- A forma como se pesquisa uma lista ou tabela pode ter um impacto muito significativo no tempo que se demora
- Quanto menor for a complexidade do algoritmo, menor o tempo de execução
- Pesquisar uma lista que não tem qualquer ordem, implica um algoritmo no mínimo linear
- No entanto, se for uma lista ordenada, é possível estratégias mais eficientes, designadamente logarítmicas:
  - Uma só comparação elimina metade do espaço de pesquisa

Pesquisa binária

- Uma só comparação elimina metade do espaço de pesquisa
  - O algoritmo de pesquisa pode ser recursivo ou iterativo
  - O recursivo terá maior complexidade espacial, uma vez que a cada chamada precisa de mais memória

Ficha 2 – Método para criar uma tabela com números inteiros

```
static int[] criaArrayCom(
     int valor, // Número que vai existir na tabela
     int dimensao, // Dimensão da tabela
     boolean diferentes){  // Se for true, a tabela não terá repetidos
  int m[]=new int[dimensao];
  if(diferentes){ // Preencher de forma sistemática p/ garantir diferentes
      for(int i=0;i<dimensao;i++)  m[i]=i*10;
      if((valor%10!=0)||(0>valor)||(valor>(dimensao-1)*10))
        m[0]=valor; // Garantir que valor existe na tabela
   }else{ // Preencher com aleatórios
      Random r=new Random();
      int gama=(Math.abs(valor)<10)?10:Math.abs(valor);
      for(int i=0;i<dimensao;i++)  m[i]=r.nextInt(gama*4)-gama*2;
      m[0]=valor; // Garantir que valor existe na tabela
   }
   Arrays.sort(m);  // Ordenar a tabela
   return m;
}
```

1 – Construa um método que efetua uma **pesquisa binária** de forma **recursiva**, devolvendo **true** ou **false** conforme o valor seja encontrado ou não.

Ficha 2 – Exercício 1

3	7	12	15
0	1	2	3

```
import java.util.Arrays;

public class Main{

public static boolean binRecursivo(int []tab,int chave){

//  Colocar o que falta aqui.
// Pode-se usar Arrays.copyOfRange() para copiar a parte relevante da tabela

    return binRecursivo(newt,chave); //,start,end);
}

    public static void main(String[] args) {
        int[] ar = {1,3,4,6,8};
        boolean res = binRecursivo(ar, 1);

        if(res==true)   System.out.println("Existe");
        eles System.out.println("Não existe");
    }
}
```

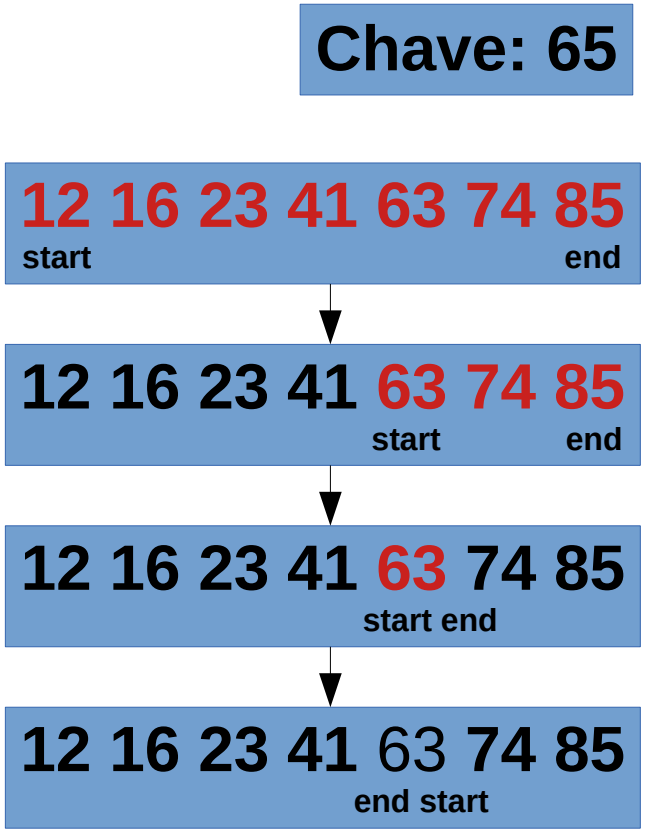
Ficha 2 – Exercício 1

3	7	12	15
0	1	2	3

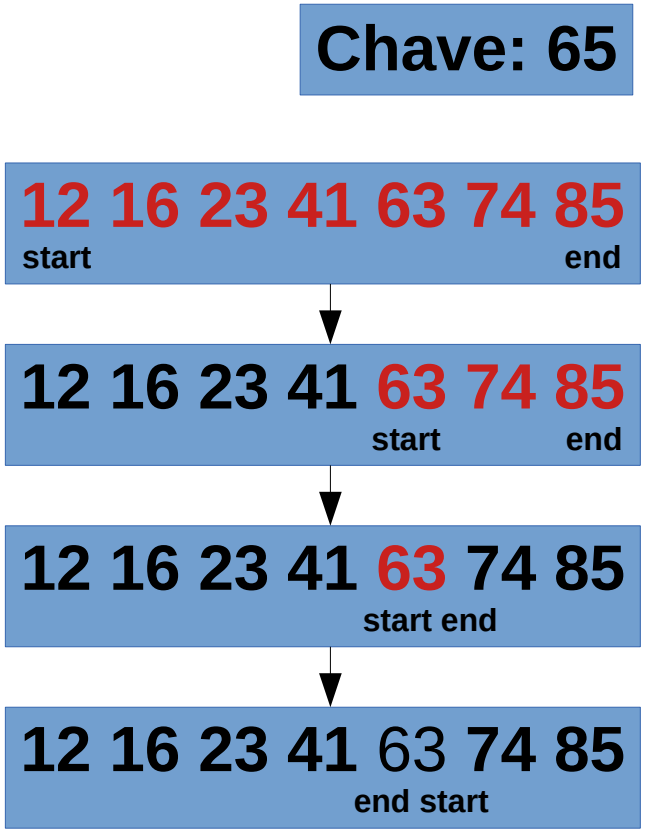
```
import java.util.Arrays;

public class Main{

public static boolean binRecursivo(int []tab,int chave){
    int start = 0, end=tab.length,  meio = (start+end)/2;

    if(end==0) {  System.out.println("Tabela vazia");  return false;   }

    if(tab[meio]==chave) return true;
    if(tab[meio]>chave)  end = meio - 1 ;  else start = meio + 1;

    if(end<tab.length) end++; // copyOfRange vai até end-1
    int[] newt=Arrays.copyOfRange(tab, start, end); // Com cópia de dados

    return binRecursivo(newt,chave); //,start,end);
}

    public static void main(String[] args) {
        int[] ar = {1,3,4,6,8};
        boolean res = binRecursivo(ar, 1);

        if(res==true)   System.out.println("Existe");
        eles System.out.println("Não existe");
    }
}
```

1 – Pesquisa binária recursiva:

- O algoritmo de pesquisa binária com uma comparação reduz para **metade** o espaço de pesquisa
  - É de complexidade **temporal** logarítmica O(log(n)), logaritmo base 2
- Se for implementado de forma recursiva, a cada iteração a função de pesquisa duplica-se, abrindo uma nova instância
  - Complexidade espacial logaritmica também

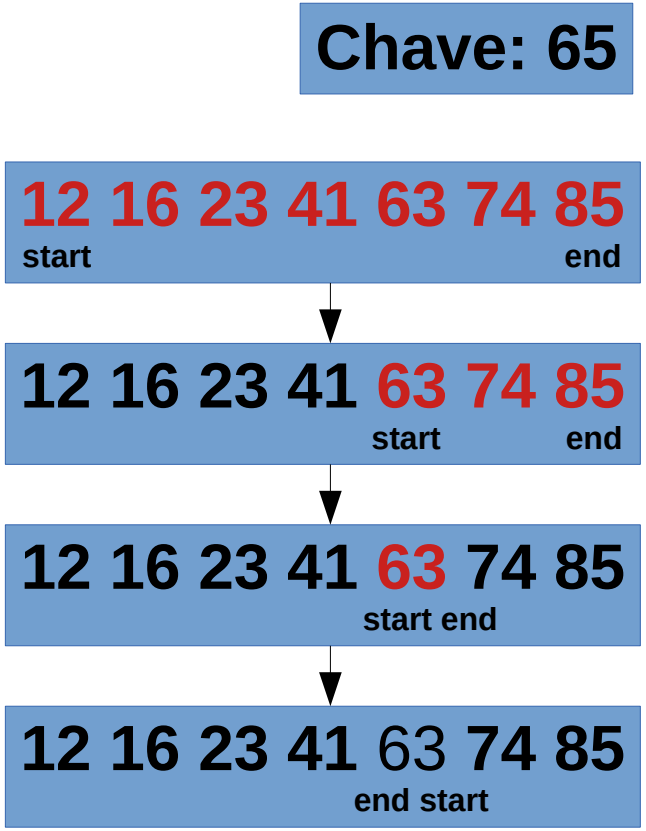


2 – Construa um método que efectua uma pesquisa binária de forma i**terativa**, devolvendo **`true`** ou **`false`** conforme o valor seja encontrado ou não.



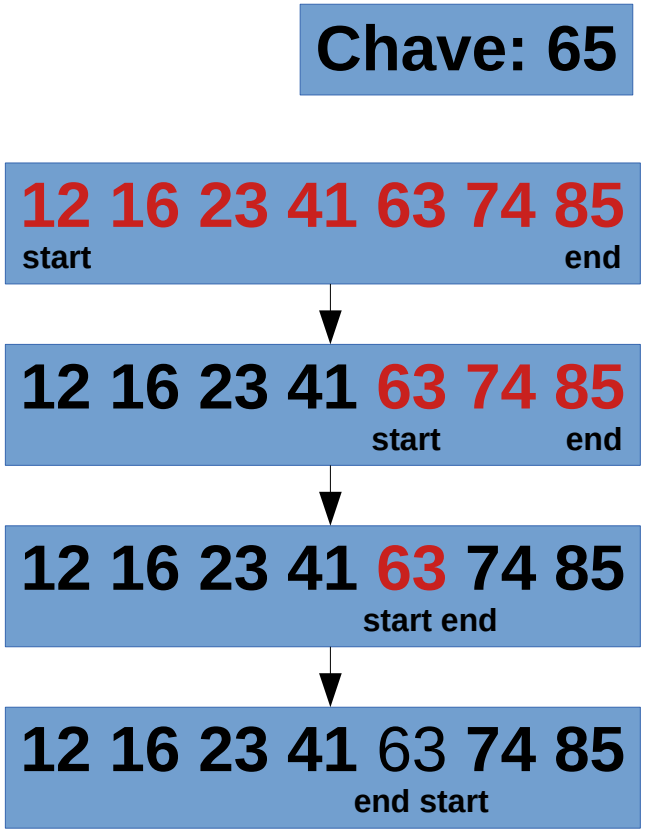
2 – Pesquisa binária iterativa

```
boolean binIterativo(int []tab,int chave){
 int linf=0, lsup=tab.length-1;
 int meio=lsup/2;

    do{
      if(tab[meio]==chave) return true;
      if(tab[meio]<chave) linf=meio+1;
          else lsup=meio-1;
      meio = (linf+lsup)/2;
    }while(linf<lsup);

    if(tab[meio]==chave) return true;
    return false;
}
```



2 – Pesquisa binária iterativa:

- A implementação iterativa (não recursiva) é mais eficiente, pois não expande o número de métodos em memória
  - Devem ser evitadas soluções recursivas, sempre que há alternativas práticas
    - No caso do processamento de árvores, por exemplo, usam-se soluções recursivas porque outro tipo de algoritmos seria muito complicado



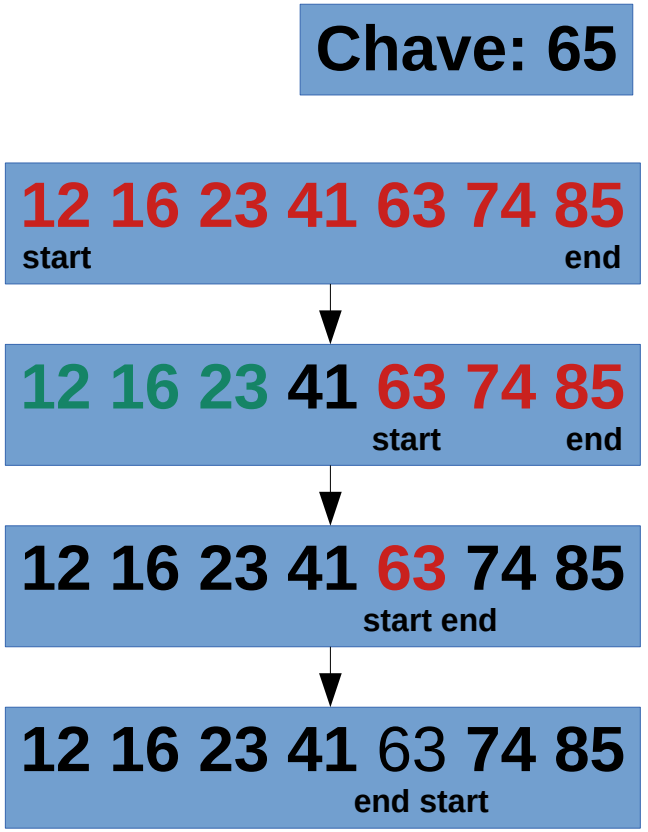
Exercício 3

3 - Construa um método que efetua uma pesquisa binária. Este método deve devolver a **posição** em que o valor procurado se encontra, ou então -1 caso este não esteja no array indicado. Teste o método, conforme o indicado nos exercícios anteriores.

Exercício 3

3 - Construa um método que efetua uma pesquisa binária.

```
public static int binIndice(int []tab,int chave){
    int linf=0,lsup = tab.length-1;
    int meio = tab.length/2;

    do{
      if(tab[meio]==chave) return meio;
      if(tab[meio]<chave)
          linf=meio+1;
      else
          lsup=meio-1;
      meio = (linf+lsup)/2;
    }while(linf<lsup);

    if(tab[meio]==chave) return meio;
    return -1;
}
```



Exercício 4

4 - Construa um método que efetua uma pesquisa binária. Este método deve devolver a **posição** em que o valor procurado se encontra, ou então um valor negativo (-X) caso este não esteja no array indicado. O valor de |X+1| deve indicar uma posição em que o valor procurado poderia ser inserido para preservar a ordem. Teste o método, conforme o indicado nos exercícios anteriores.

3	7	12	15
0	1	2	3

Exemplo para Array: {3, 7, 12, 15}

Chave=15 resultado= 3

Chave=3 resultado= 0

Chave=1 resultado= -1 (Deveria ser inserido na posição 0)

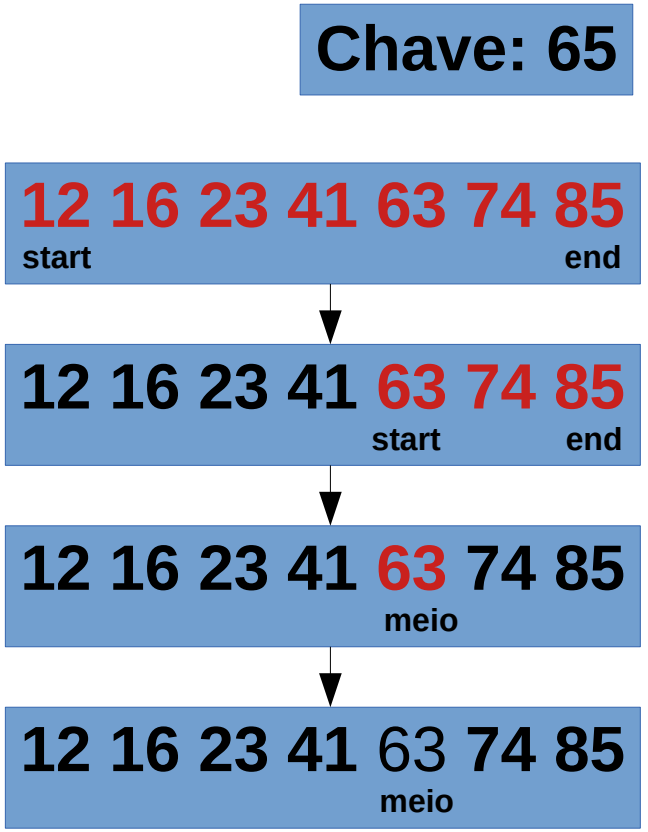
Chave=4 resultado= -2 (o valor deveria ser inserido na posição 1)

Chave=10 resultado= -3 (o valor deveria ser inserido na posição 2)

3	7	~~12~~ 10	12	15
0	1	2	~~2~~ 3	~~3~~ 4

Exercícios

```
 public static int binCmp(int []tab,int chave){
    int linf=0,lsup = tab.length-1,
    int meio = tab.length/2;

    do{
      if(tab[meio]==chave) return meio;
      if(tab[meio]<chave)
          linf=meio+1;
      else
          lsup=meio-1;
      meio = (linf+lsup)/2;
    }while(linf<lsup);

    if(tab[meio]==chave) return meio;

// if(  tab[meio]<chave) meio +=2; // Chave para a direita
//     else meio +=1;   // Chave para a esquerda
// return -meio;

    if(tab[meio]<chave) return -meio-2; // Chave deve ir para direita do meio
    return -meio-1; // Chave deve ir para a esquerda do meio
}
```



5 - Construa um método que recebe um array ordenado de inteiros, todos diferentes, e um valor, e devolve a **percentagem** de valores do array que são menores do que o valor indicado. O algoritmo deve ser de complexidade logarítmica.

Exemplo:

Array: {3,7,12,15}

Valor =15 resultado= 0.75

Valor =14 resultado= 0.75

Valor =3 resultado= 0.0

Valor =1 resultado= 0.0

Valor =100 resultado= 1.0

3	7	12	15
0	1	2	3

Exercícios

5 - Construa um método que recebe um array ordenado de inteiros, todos diferentes, e um valor, e devolve a **percentagem** de valores do array que são menores do que o valor indicado.

Exemplo de solução, de ordem de **complexidade logarítmica**:

```
public static double percentagem(int m[], int valor){
  int pos=binCmp(m,valor); // Método da alínea anterior

  //se o valor existe
  if(pos>=0) return (double)pos/m.length;

  //se o valor não existe
  int posInsert=-pos-1;
  //o valor na pos de inserção é menor e deve contar
  return posInsert/(double)m.length;
}
```