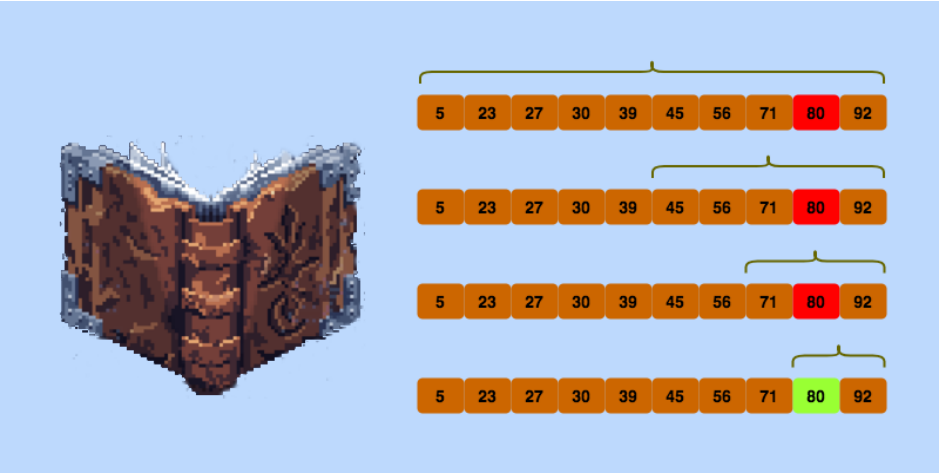
Capítulo

6 Pesquisa



Pesquisa - Search

Apresentaremos e discutiremos diferentes estratégias para efetuarmos a pesquisa (busca) de um elemento específico em um conjunto de dados. Esta operação é muito importante, pois é encontrada com muita freqüência em diversas aplicações. Apresentaremos os métodos de pesquisa seqüencial e binária sobre a estrutura de dados vetor.

Varredura – Pesquisa Sequencial

É a forma mais simples e primitiva de pesquisa em um arquivo seqüencial. Este procedimento consiste em varrer o arquivo até o fim e só parar quando achar a chave, ou quando a chave de pesquisa for maior que a próxima chave do arquivo ou ainda quando for fim de arquivo.

A ***varredura*** é uma técnica que consiste basicamente de se varrer o arquivo seqüencialmente do início até o seu final. A pesquisa será terminada quando o registro for achado ou quando detectar o final do arquivo. No caso dos arquivos seqüenciais, ordenados por uma determinada chave, a pesquisa também termina quando a chave de pesquisa for menor que a chave corrente do arquivo.

Considere a Figura 6.1 (a) que corresponde um vetor ordenado seqüencialmente. Na pesquisa da chave 20, deve-se varrer o vetor até a posição cuja célula contém a chave igual a 20. Logo, os endereços visitados são: 0, 1 e 2. A Figura 6.1 (b) ilustra esta seqüência.

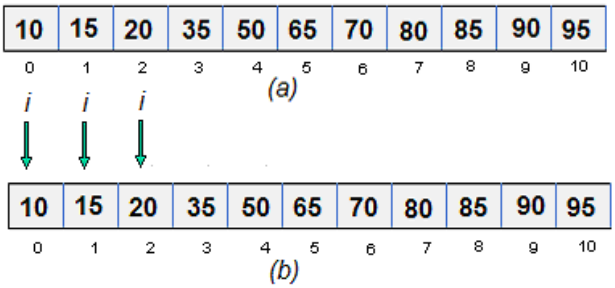


Figura 6.1 – Pesquisa do elemento 20.

Para a pesquisa da chave 40, deve-se varrer o arquivo do 1º registro (endereço 0) e parar no 5º registro (endereço 4) pois, se a chave de pesquisa é menor que a chave corrente do vetor, e como o vetor está ordenado, pode-se concluir que a chave de pesquisa não se encontra no vetor. A Figura 6.2 (a), ilustra este procedimento. Para a pesquisa da chave 100, deve-se varrer o arquivo do 1º registro (endereço 0) e parar no final do vetor (endereço 12) . A Figura 6.2 (b), ilustra este procedimento.

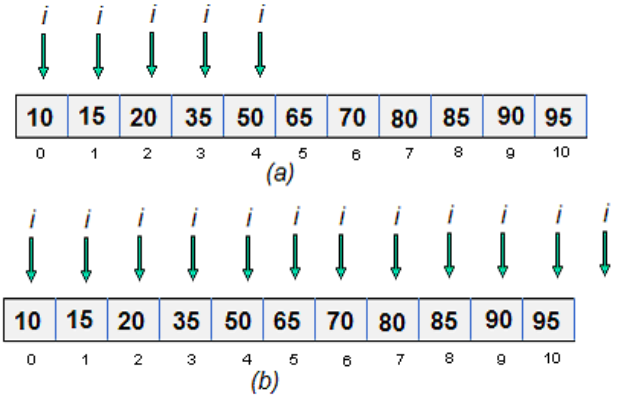


Figura 6.2 – (a) Pesquisa do elemento 40. (b) Pesquisa do elemento 100

O fragmento de código 6.1 ilustra a pesquisa de uma chave em um vetor de forma iterativa.

```
global NOT_FOUND
NOT_FOUND=-1

# iterative scan search
def scan(alist,key):
  N = len(alist)
  for i in range(0, N):
      if key == alist[i]:
        return i
      elif key < alist[i]:
        return NOT_FOUND
  return NOT_FOUND
```

Fragmento de Código 6.1 – Pesquisa iterativa de uma determinada chave em um vetor.

O fragmento de código 6.2 ilustra a pesquisa de uma chave em um vetor de forma recursiva.

```
def recursiveScan(alist,key):
  def bodyRecursiveScan(alist,key,i):
    N = len(alist)
    if i==N:
      return NOT_FOUND
    elif key == alist[i]:
      return i
    elif key < alist[i]:
      return NOT_FOUND
    return bodyRecursiveScan(alist,key,i+1)
  return bodyRecursiveScan(alist,key,0)
```

Fragmento de Código 5.2 – Pesquisa iterativa de uma determinada chave em um vetor.

Pesquisa Binária

A idéia básica deste método está em visitar o registro do meio verificando se a chave de pesquisa é maior, menor ou igual à chave do arquivo. Caso a chave seja igual, pare e retorne o endereço da célula.

A Figura 6.3 (a) ilustra este procedimento supondo a chave de pesquisa igual a 65, neste caso a função retornará o endereço 5. Caso a chave de pesquisa for maior que a chave do vetor, por exemplo 85, conclui-se que a chave deve estar no subvetor à direita da célula do meio. Neste caso o ponteiro low receberá o valor mid+1, e a rotina é ativada novamente. A figura 6.3 (b) ilustra este caso. Caso a chave de pesquisa for menor que a chave do vetor, conclui-se que a chave deve estar no subvetor à esquerda da célula do meio. Neste caso o ponteiro high receberá o valor mid-1 e a rotina é ativada novamente. A figura 6.3 (c) ilustra este caso.

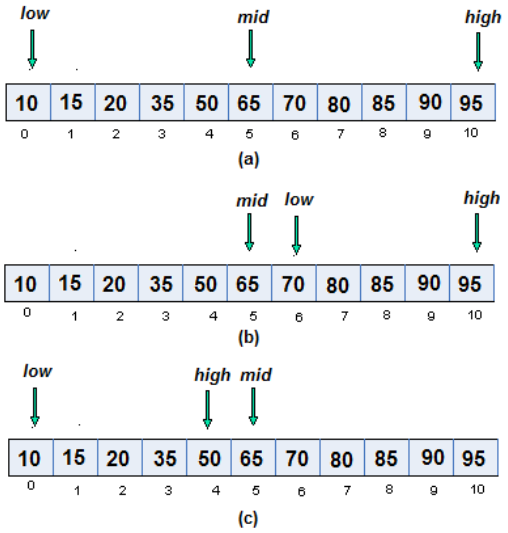


Figura 6.3 – (a) Posicionamento na célula de endereço mid (b) chave maior que a chave do endereço mid (c) chave menor que a chave do endereço mid .

É fácil de concluir que a pesquisa termina quando a chave de pesquisa for igual à chave armazenada no vetor, ou caso o ponteiro low > high, sendo que neste caso a chave não existe no vetor.

Procedimento de pesquisa binária.

1. Início = 0, fim = tamanho do vetor
2. Se inicio > fim parar o procedimento com insucesso.
3. Visitar a célula do meio onde meio = quociente inteiro da divisão de início + fim por 2.
4. Se a chave de pesquisa for igual à chave da célula do meio parar o procedimento com sucesso e retornar o endereço da célula do meio.
5. Se a chave de pesquisa for menor que a chave do meio, ignorar todas as células posteriores á célula do meio, inclusive esta (basta fazer fim = meio - 1), e voltar ao passo 2.
6. Se a chave de pesquisa for maior que a chave do meio, ignorar todas as células anteriores à célula do meio, inclusive esta (basta fazer início = meio +1), e voltar ao passo 2.

O fragmento de código 6.3 ilustra a pesquisa binária de uma chave em um vetor de forma iterativa.

```
def iterativeBinarySearch(alist,key):
  N = len(alist)
  low = 0
  high = N-1
  mid=(low+high) //2
  while(low<=high):
    if key == alist[mid]:
      return mid
    elif key < alist[mid]:
      high=mid-1
    else:
      low=mid+1
    mid=(low+high) //2
  return NOT_FOUND
```

Fragmento de Código 6.3 – Pesquisa binária iterativa de uma determinada chave em um vetor.

O fragmento de código 6.4 ilustra a pesquisa binária de uma chave em um vetor de forma recursiva.

```
def recursiveBinarySearch(alist,key):
    return bodyRecursiveBinarySearch(alist,key,0,len(alist)-1)

def bodyRecursiveBinarySearch(alist,key,low,high):
    mid=(low+high) //2
    if low>high:
      return NOT_FOUND
    if key == alist[mid]:
      return mid
    elif key < alist[mid]:
      return bodyRecursiveBinarySearch(alist,key,low,mid-1)
    else:
      return bodyRecursiveBinarySearch(alist,key,mid+1,high)
    return NOT_FOUND
```

Fragmento de Código 6.4 – Pesquisa binária recursiva de uma determinada chave em um vetor.

Pesquisa por Interpolação

Variante da pesquisa binária em que escolhe o elemento a verificar de acordo com a interpolação. Por exemplo, se o valor da chave visitada estiver mais próximo do valor da chave do início do vetor, damos um “peso” maior ao lado do início, caso contrário damos um “peso”maior ao lado do fim. Um exemplo prático do uso da interpolação é a pesquisa de uma palavra no dicionário.

A idéia deste método é tornar com célula a ser visitada a média ponderada no intervalo [low, high], com pesos valor da chave do ponteiro low e valor da chave do ponteiro high. O cálculo do endereço a ser visitado é dado pela expressão:

$$mid = low + (abs((high - low) * (key - min))/(max - min))$$

Onde:

low : ponteiro correspondente ao início do bloco
high: ponteiro correspondente ao fim do bloco
key : chave a ser pesquisada
max: valor da chave que está na célula high
min: valor da chave que está na célula low

OBS: PROBLEMA As chaves devem ser do tipo inteiro pois se fosse do tipo string teríamos que ter uma função que convertesse em inteiro de forma que as essas chaves tenham valores inteiros sequenciais.

A Figura 6.4 ilustra este procedimento. A letra (a) ilustra a pesquisa da chave 20 e a figura (b) ilustra a pesquisa da chave 80.

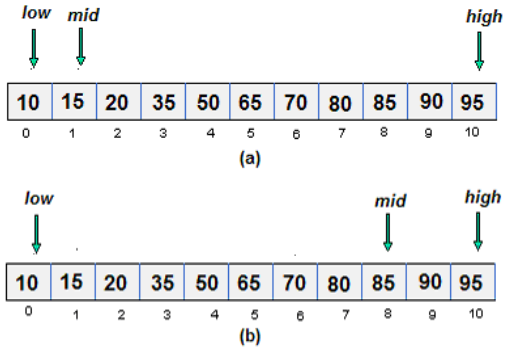


Figura 6.4 - Arquivo Sequencial para aplicação da pesquisa por interpolação.

O fragmento de código 6.5 ilustra a pesquisa por interpolação de uma chave em um vetor de forma iterativa.

```
def iterativeInterpolationSearch(alist,key):
    N = len(alist)
    low=0
    high=N-1
    while(low<=high):
        min = alist[low]
        max = alist[high]
        mid = ( low + abs((high-low ) * (key-min))// (max-min))
        if(max==min):
          return NOT_FOUND
        if(mid>high or mid<low):
          return NOT_FOUND
        if( alist[ mid ]== key ):
          return mid
        elif( alist[ mid ] < key ):
          low= mid + 1
        else :
          high=mid - 1
    return NOT_FOUND
```

Fragmento de Código 6.5 – Pesquisa por interpolação iterativa de uma determinada chave em um vetor.

O fragmento de código 6.6 ilustra a pesquisa por interpolação de uma chave em um vetor de forma recursiva.

```
def recursiveInterpolationSearch(alist,key):
  def bodyRecursiveInterpolationSearch(alist,key,low,high):
    if(low<=high):
        min = alist[low]
        max = alist[high]
        mid = ( low + abs((high-low ) * (key-min))// (max-min))
        if(max==min):
          return NOT_FOUND
        if(mid>high or mid<low):
          return NOT_FOUND
        if( alist[ mid ] == key ):
          return mid
        if( alist[ mid ] < key ):
          return bodyRecursiveInterpolationSearch(alist,key,mid+1,high)
        else :
          return bodyRecursiveInterpolationSearch(alist,key,low,mid-1)
    return NOT_FOUND
```

Fragmento de Código 6.6 – Pesquisa por interpolação recursiva de uma determinada chave em um vetor.

Pesquisa Blocada

A ideia desse método está em dividir o vetor em blocos de tamanho constante, e pesquisar a maior chave de cada bloco. Se a chave de pesquisa for menor que a maior chave do bloco, pesquisar a chave no bloco correspondente utilizando qualquer método de pesquisa. Caso, a chave de pesquisa seja maior que a maior chave do bloco, visitar o próximo bloco e aplicar o procedimento anterior. O algoritmo, sinalizando não existe, quando a chave de pesquisa for menor que a chave do bloco e essa chave não estiver no bloco ou, quando se acabaram os blocos de pesquisa.

Procedimento

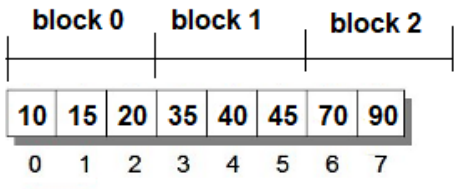
O vetor é dividido em blocos de tamanho fixo, e a pesquisa é feita em duas partes:

i. Determina-se qual é o possível bloco que contém a chave

visitar a última célula do bloco
Se a chave de pesquisa for maior que a chave da última célula do bloco, ir para o próximo bloco, caso contrário, esse é o possível bloco

ii. Executa-se uma varredura dentro do possível bloco

Por exemplo, considere o vetor a seguir ordenado a seguir, com os blocos lógicos de tamanho 3.



Em seguida, visita-se a última célula de cada bloco e escolhe-se da seguinte forma:

$$i = block\_size * (bn + 1) - 1$$

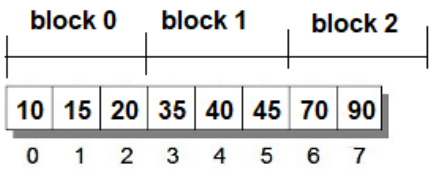
Onde

$block\_size$: tamanho do bloco
bn : número do bloco

Se a chave de pesquisa for menor ou igual à chave da última célula do bloco, executa-se a varredura dentro desse bloco

Se a chave de pesquisa for maior que a chave da última célula do bloco pular para o próximo bloco.

Exemplo 1) Fazer a pesquisa da chave 40



Primeiro endereço visitado (bloco 0) :

$$i = block\_size * (bn + 1) - 1$$
$$i = 3 * (0 + 1) - 1 = 2 \rightarrow (array[2] = 20)$$

Como 40 > 20, ir para o próximo bloco:

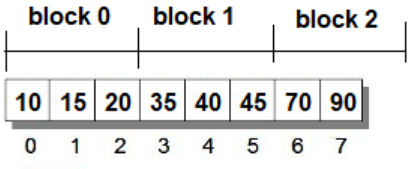
Segundo endereço visitado (bloco 1):

$$i = block\_size * (bn + 1) - 1$$
$$i = 3 * (1 + 1) - 1 = 5 \rightarrow (array[5] = 45)$$

Como 40 <45, fazer uma varredura no bloco 1

Endereços visitados na varredura do bloco: 3,4
O procedimento para com sucesso e, retorna 4
Endereços visitados : 2,5, 3, 4 para com sucesso e. retorna 4

Exemplo 2) Fazer a pesquisa da chave 80



Primeiro endereço visitado:

$$i = block\_size * (bn + 1) - 1$$
$$i = 3 * (0 + 1) - 1 = 2 \rightarrow (array[2] = 20)$$

Como 80 > 20, ir para o próximo bloco:

Segundo endereço visitado:

$$i = block\_size * (bn + 1) - 1$$
$$i = 3 * (1 + 1) - 1 = 5 \rightarrow (array[5] = 45)$$

Como 80 > 45, ir para o próximo bloco:

Terceiro endereço visitado:

$$i = block\_size * (bn + 1) - 1$$
$$i = 3 * (2 + 1) - 1 = 8$$

Como $i \geq len(array)$, pois esse bloco não está completo, recalcula-se o valor de i para receber o endereço da última célula

$$i = len(array) - 1 = 7 \rightarrow (array[7] = 90)$$

Como 80 < 90, fazer a varredura dentro do bloco

Endereços visitados na varredura do bloco: 6,7 para com insucesso e retorna -1 (NOT_FOUND)

Endereços visitados : 2,5, 7,6,7 para com insucesso e retorna -1

O fragmento de código 6.7 ilustra a pesquisa blocada de uma chave em um vetor de forma iterativa.

```
def iterativeBlockedSearch(alist,key):
  N = len(alist)
  block_size= int(math.sqrt(N))  # tamanho lógico do bloco
  for bn in range(0,N // block_size + 1):  # pesquisa do bloco
    # bn : número do bloco: 0,1,2, ....
    # inicializa i com o índice da primeira célula do bloco
    i=block_size*(bn+1)-1
    if(i >= N): # verifica se índice é maior índice da última célula
      i = N - 1 # retorna i para a posição final do vetor
    if key > alist[i]: # chave maior que a última chave do bloco
        if i == N - 1 :
          return NOT_FOUND
        else:
          continue
    else:
        # percorre o bloco que pode conter a chave
        return blockScan(key,alist,bn*block_size,i+1)
  return NOT_FOUND
```

Fragmento de Código 6.7 – Pesquisa blocada iterativa de uma determinada chave em um vetor.

Pesquisa Blocada utilizando pesquisa binária

A ideia desse método está em pesquisar o possível bloco utilizando pesquisa binária, e depois de achado o possível bloco que contém a chave é feita novamente uma pesquisa binária dentro do bloco.

Se a chave de pesquisa for menor que a primeira chave do bloco mid, o bloco high receberá mid – 1 e, então repete-se o processo. A figura 6.5 ilustra esse processo.

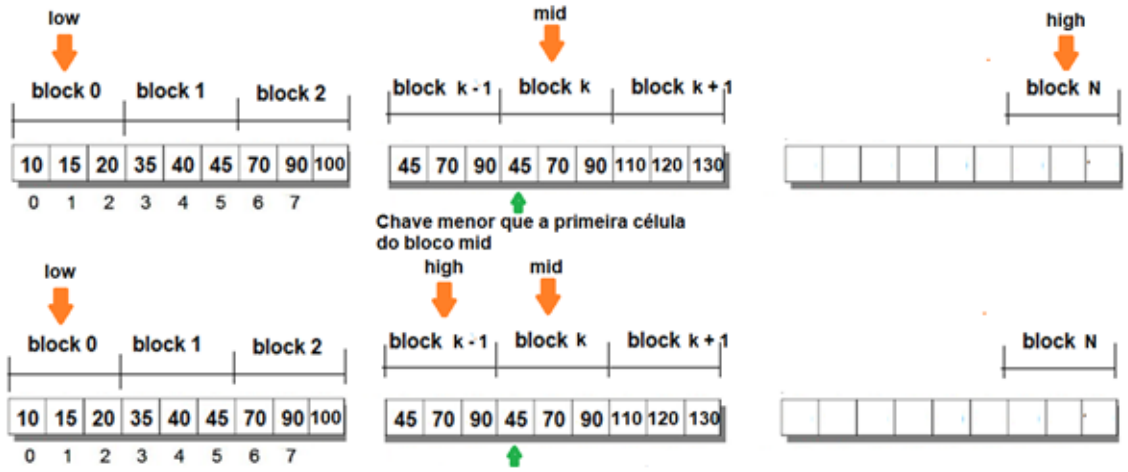


Figura 6.5 - Pesquisa blocada utilizando pesquisa binária. Chave de pesquisa menor que a primeira célula do bloco mid

Se a chave de pesquisa for menor que a última chave do bloco mid, o bloco low receberá mid + 1 e, então repete-se o processo. A figura 6.6 ilustra esse processo.

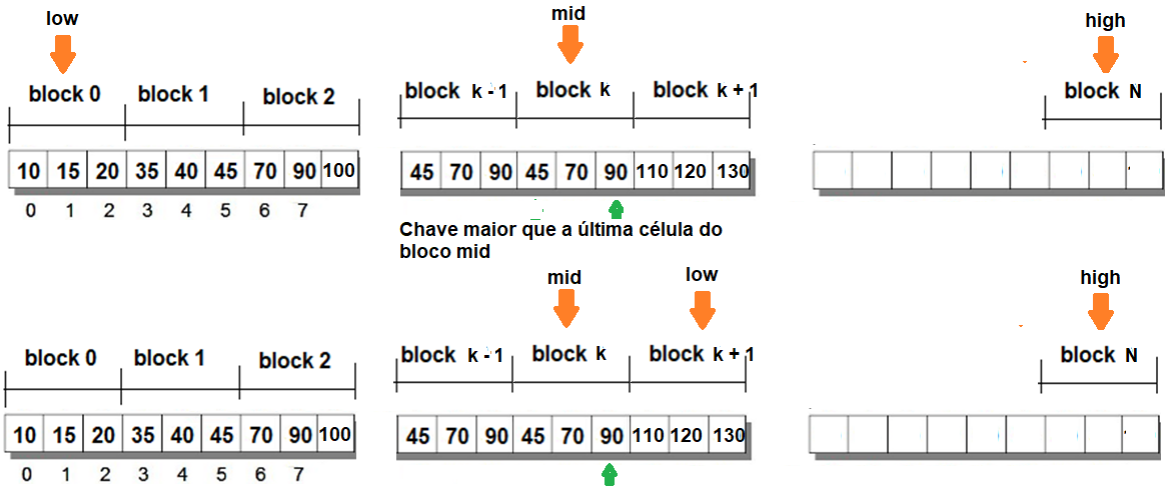


Figura 6.6 - Pesquisa blocada utilizando pesquisa binária. Chave de pesquisa maior que a última célula do bloco mid

No caso contrário, executa-se uma pesquisa binária dentro do bloco mid, ou seja, a chave de pesquisa maior que a primeira célula do bloco mid e, menor que a última célula do bloco mid A figura 6.7 ilustra que o bloco a ser pesquisado é o bloco mid.

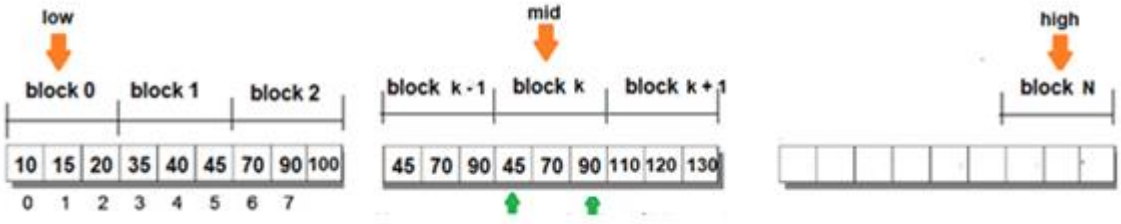


Figura 6.7 - Pesquisa blocada utilizando pesquisa binária. Chave de pesquisa maior que a primeira célula do bloco mid e, menor que a última célula do bloco mid

O fragmento de código 6.8 ilustra a procedimento auxiliar, blockScan, que faz a varredura dentro do bloco escolhido.

```
def blockScan(key,alist,min,max):
    for i in range(min,max):
      if key == alist[i]:
        return i
      if key < alist[i]:
        return NOT_FOUND
    return NOT_FOUND
```

Fragmento de Código 6.8 – Procedimento auxiliar que faz a varredura dentro do bloco escolhido.

O fragmento de código 6.9 ilustra a pesquisa blocada de forma recursiva. Para determinação do possível bloco que contém a chave, é feita uma pesquisa binária de forma recursiva e, depois de determinarmos o bloco, faz-se uma pesquisa binária recursiva dentro do bloco.

```
def blockedBinarySearch(arr, key):
    # recursive bodyblockedBinarySearch
    def bodyBlockedBinarySearch(alist, block_size, low, high, key):
        mid=(low+high) // 2
        if low > high:
            return NOT_FOUND
        if key < alist[block_size*mid]:
            return bodyBlockedBinarySearch(alist,block_size,low,mid-1,key)
        elif key > alist[block_size*(mid+1)-1]:
            return bodyBlockedBinarySearch(alist,block_size,mid+1,high,key)
        else:
            pos = bodyRecursiveBinarySearch(alist,key,block_size*mid,block_size*(mid+1))
            if pos == -1:
                return NOT_FOUND
            else:
                return pos # pos+block_size*mid
        return NOT_FOUND
    block_size= int(math.sqrt(len(arr)))    # tamanho lógico do bloco
    return bodyBlockedBinarySearch(arr,block_size,0, len(arr) // block_size, key)
```

Fragmento de Código 6.9 – Pesquisa blocada recursiva de uma determinada chave em um vetor.