Ordenação e Busca
(IED-001)

Prof. Dr. Silvio do Lago Pereira

Departamento de Tecnologia da Informação

Faculdade de Tecnologia de São Paulo

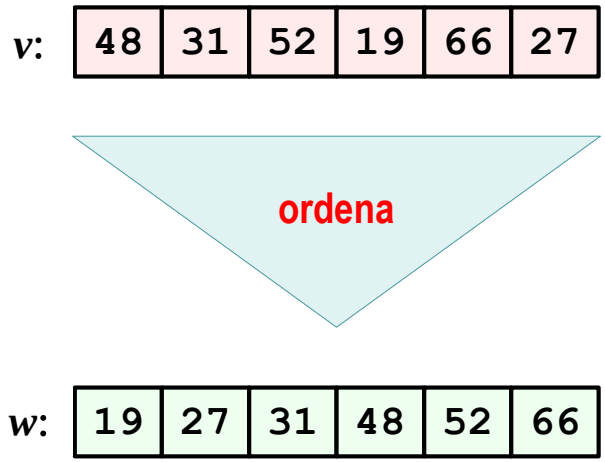
Ordenação

Ordenação

é a operação que **reorganiza** os itens de um vetor para que eles fiquem em ordem **crescente**.

Dado um vetor $v[n]$, sendo $n \geq 1$, a **ordenação** encontra uma permutação w de v tal que:

- $w_0 \leq w_1 \leq \ldots \leq w_{n-1}$



Alternativamente, a ordenação pode ter que encontrar uma permutação **decrescente** de v!

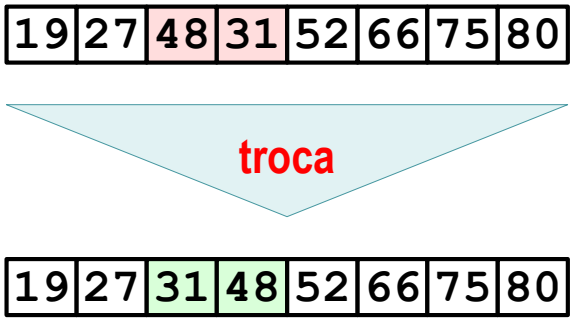
Ordenação

Troca

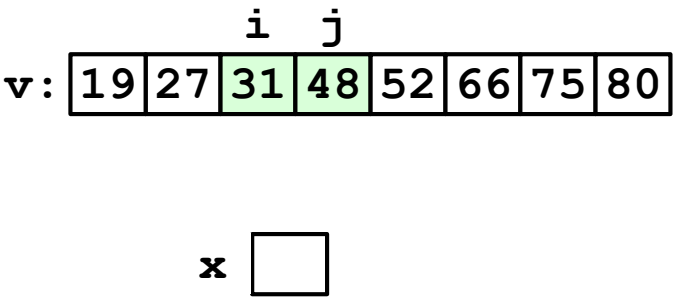
é a operação que **troca** os valores de duas posições quaisquer de um vetor.

Estratégia:

- Copie o valor da posição ***i*** para ***x***.
- Armazene na posição ***i*** o valor da posição ***j***.
- Armazene na posição ***j*** o valor de ***x***.



Funcionamento:




```
void troca(int v[], int i, int j) {
   int x = v[i];
   v[i] = v[j];
   v[j] = x;
}
```

A operação **troca** consome tempo constante (isto é, independe do tamanho do vetor)!

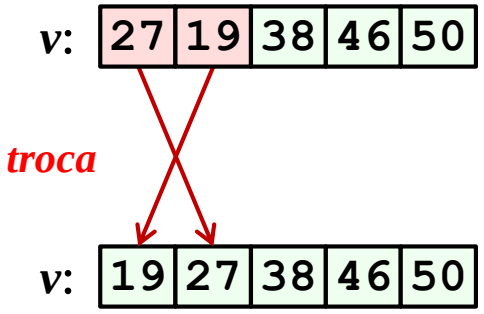
Ordenação por trocas (*bubble sort*)

é um algoritmo de ordenação cuja operação básica é a **troca**.

Estratégia:

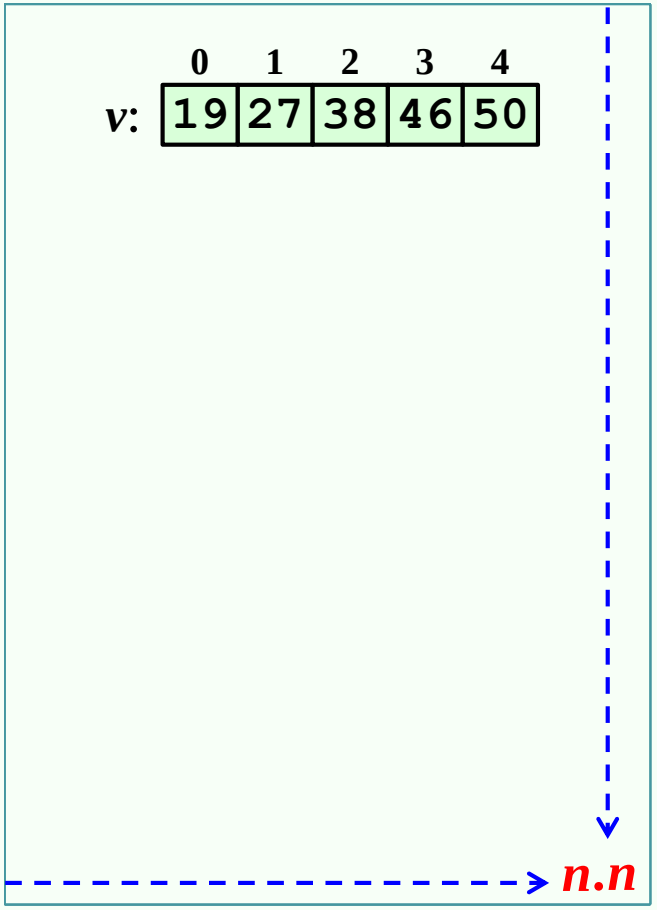
Para ordenar um vetor $v[n]$, usando ordenação por trocas:

- Encontre em v um par de itens consecutivos v_j e v_{j+1}, tal que $v_j > v_{j+1}$.
- Troque esses itens de posição, de modo que cada um passe a ocupar a posição do outro.
- Repita o procedimento, enquanto for possível.



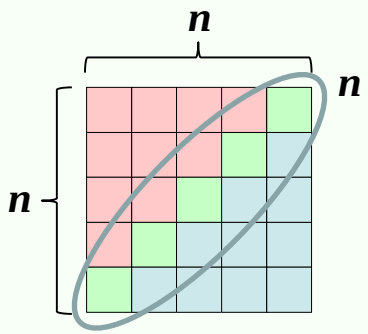
Para garantir a correta ordenação, o algoritmo deve fazer todas as comparações possíveis!

Exemplo 1. Simulação do *bubble sort*



Para um vetor v com n itens distintos:

- No total são feitas $(n–1) + (n–2) + (n–3) + \ldots + 2 + 1 = (n^2–n)/2$ comparações (cada uma delas podendo resultar numa troca).
- No pior caso, o algoritmo ***bubble sort*** consome tempo proporcional a n^2.



Ordenação

Exemplo 2. Implementação do *bubble sort*

```
void bsort(int v[], int n) {
    for(int i=1; i<n; i++)
        for(int j=0; j<n-i; j++)
            if( v[j]>v[j+1] )
                troca(v,j,j+1);
}
```

Exercício 1. Teste da função `bsort()`

Crie a função **`exibe()`**, complete e execute o programa a seguir.

```
#include <stdio.h>
...
int main(void) {
    int v[10] = {83,31,91,46,27,20,96,25,96,80};
    bsort(v,10);
    exibe(v,10);
    return 0;
}
```

Ordenação

Exercício 2. A função `empurra()`

Crie a função recursiva **`empurra(v,u)`**, que "empurra" um item **máximo** do vetor **`v`** para a posição **`u`** de **`v`**, possivelmente alterando a ordem dos demais itens do vetor. Por exemplo, o código abaixo deve produzir a saída indicada a seguir:

```
int v[9] = {51,82,38,99,75,19,69,46,27};
empurra(v,8);
exibe(v,9);
```

Saída: `{51,38,82,75,19,69,46,27,99}`

Exercício 3. Versão recursiva de *bubble sort*

Crie a função recursiva **`bsr(v,n)`**, que usa a função **`empurra()`** e a estratégia do ***bubble sort***, para organizar os **`n`** itens do vetor **`v`** em ordem **crescente**.

```
int v[9] = {51,82,38,99,75,19,69,46,27};
bsr(v,9);
exibe(v,9);
```

Saída: `{19,27,38,46,51,69,75,82,99}`

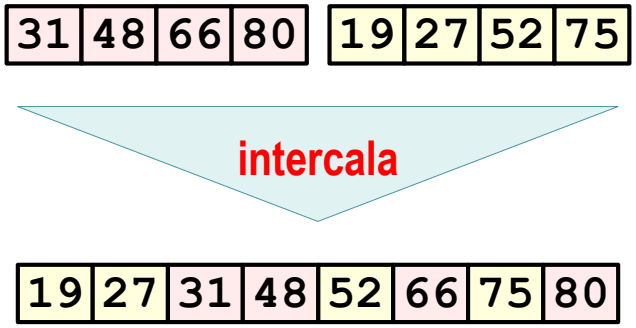
Ordenação por intercalação

Intercalação (*merge*)

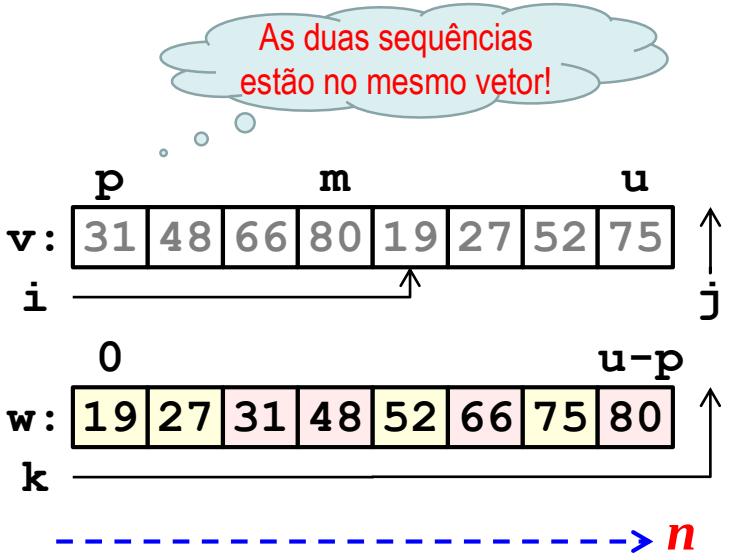
é a operação que **combina** duas sequências ordenadas numa única sequência ordenada.

Estratégia:

- Enquanto nenhuma sequência estiver vazia:
  - Compare o 1º item de uma ao 1º item da outra.
  - Copie o menor deles para a sequência final.
- Copie os itens restantes para a sequência final.



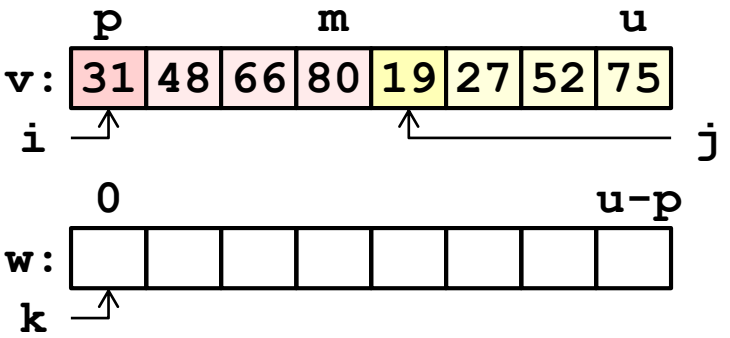
Funcionamento:



A operação **intercala** consome tempo proporcional a n.

Ordenação por intercalação

Exemplo 3. A função `intercala()`




```
void intercala(int v[], int p, int m, int u) {
   int *w = malloc((u-p+1)*sizeof(int));
   int i=p, j=m+1, k=0;
   while( i<=m && j<=u )
      w[k++] = (v[i]<v[j]) ? v[i++] : v[j++];
   while( i<=m ) w[k++] = v[i++];
   while( j<=u ) w[k++] = v[j++];
   for(k=0; k<=u-p; k++) v[p+k] = w[k];
   free(w);
}
```

Ordenação

Exercício 4. Teste da função `intercala()`

Complete o programa a seguir, execute-o e analise os resultados.

```
#include <stdio.h>
...
int main(void) {
    int v[8] = {31,48,60,80,19,27,52,75};
    intercala(v,0,3,7);
    exibe(v,8);
    int w[9] = {10,82,27,38,41,53,60,75,99};
    intercala(w,0,1,8);
    exibe(w,9);
    return 0;
}
```

Note que a operação **intercala** funciona até para intercalar partes de tamanhos diferentes!

Ordenação por intercalação

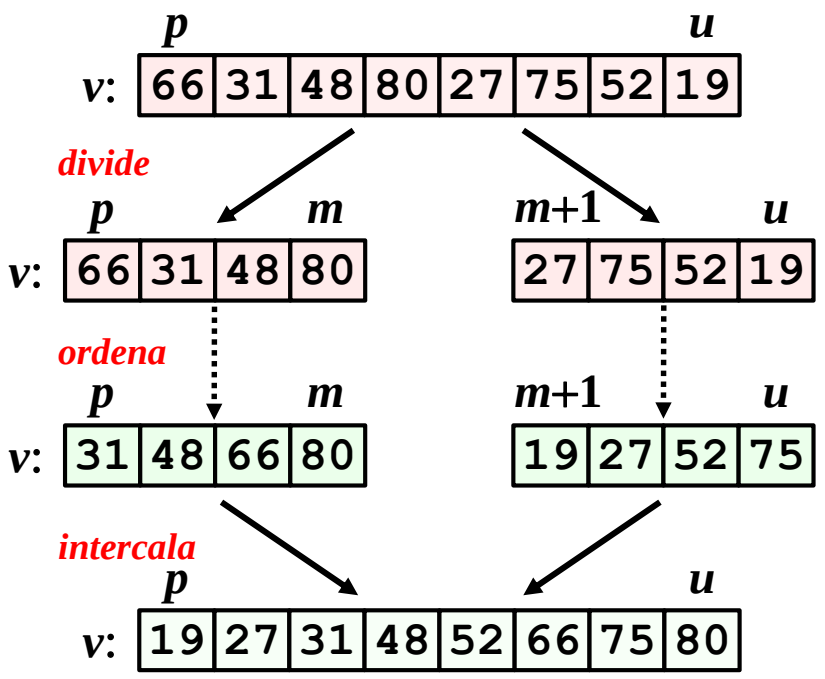
Ordenação por intercalação (*merge sort*)

é um algoritmo de ordenação cuja operação básica é a **intercalação**.

Estratégia:

Para ordenar uma sequência $v[p:u+1]$, com pelo menos dois itens, usando ordenação por intercalação:

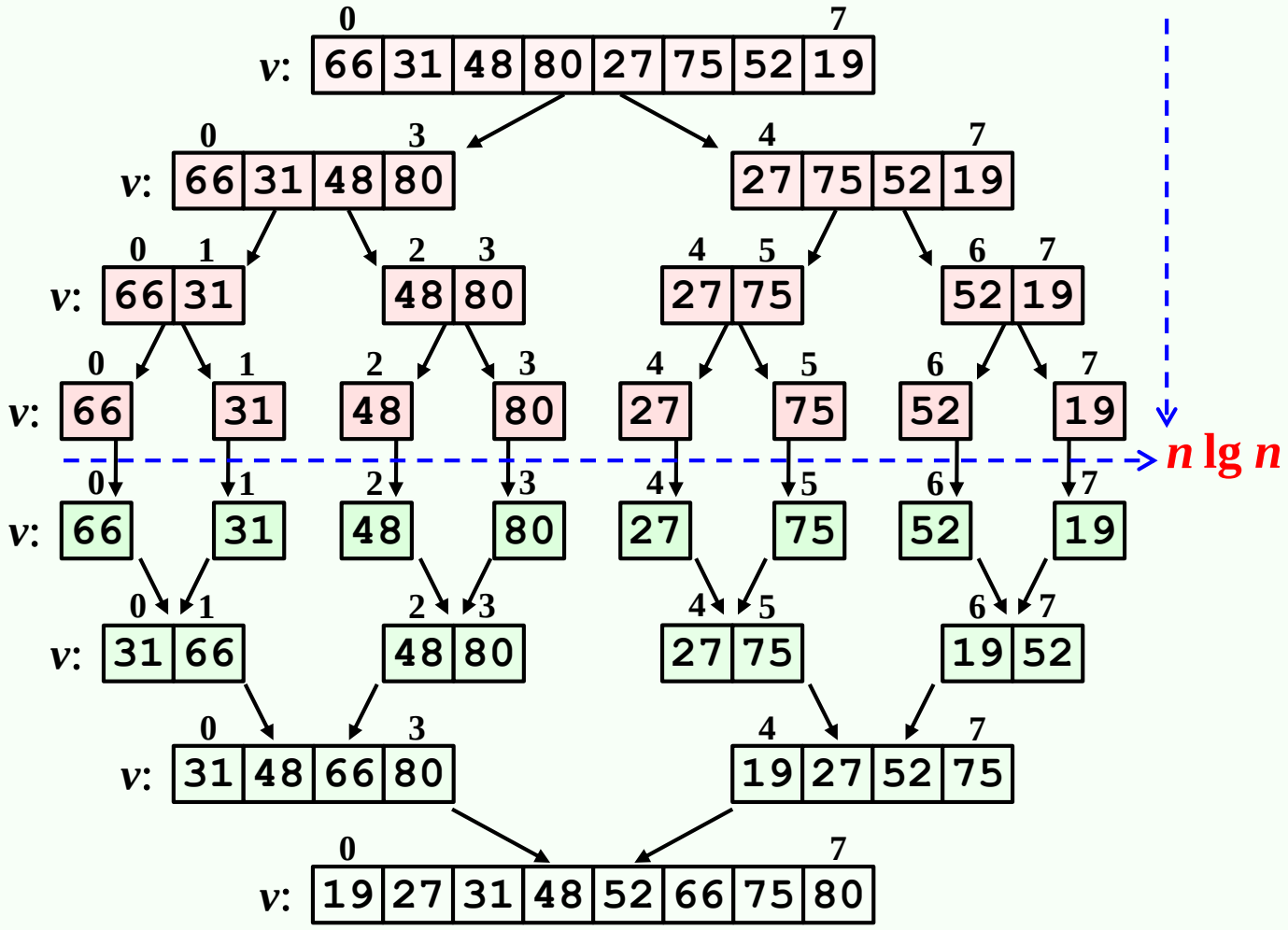
- Divida v em duas partes com aproximadamente o mesmo tamanho ($v[p:m+1]$ e $v[m+1:u+1]$).
- Ordene *recursivamente* cada uma das partes.
- Intercale as duas partes já ordenadas.



O algoritmo de ordenação por intercalação é baseado do princípio da "**divisão e conquista**"!

Ordenação por intercalação

Exemplo 4. Simulação do *merge sort*

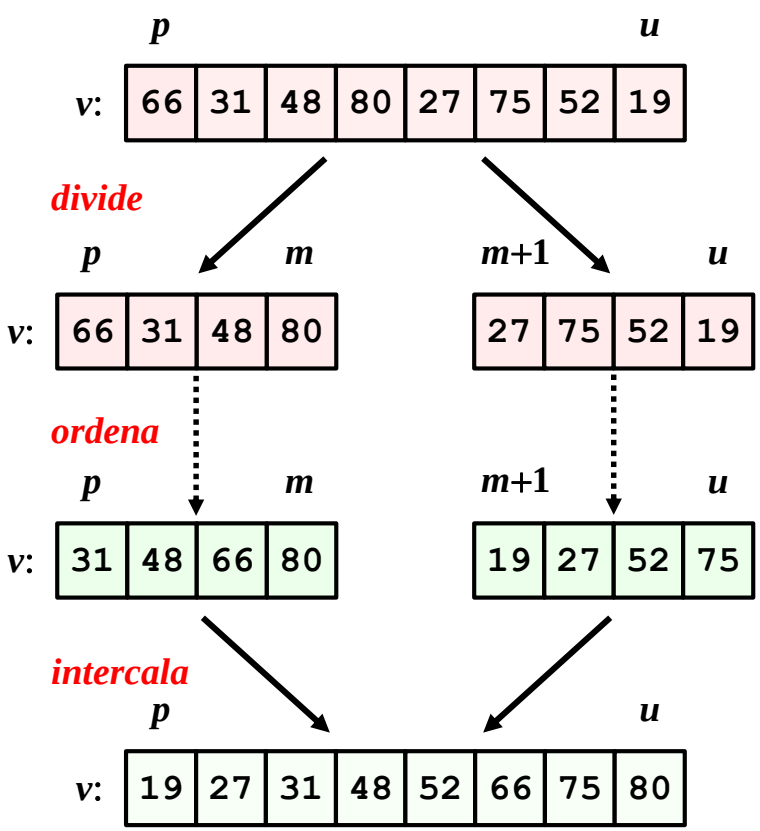


Ordenação

Exemplo 5. Implementação do *merge sort*

```
void ms(int v[], int p, int u) {
    if( p==u ) return;
    int m = (p+u)/2;
    ms(v,p,m);
    ms(v,m+1,u);
    intercala(v,p,m,u);
}

void msort(int v[], int n) {
    ms(v,0,n-1);
}
```



A função **msort()** é um "*wrapper*" para a função **ms()**!

Ordenação

Exercício 5. Teste da função `msort()`

Complete e execute o programa a seguir.

```
#include <stdio.h>
...
int main(void) {
    int v[10] = {83,31,91,46,27,20,96,25,96,80};
    msort(v,10);
    exibe(v,10);
    return 0;
}
```

Exercício 6. Preenchimento aleatório

Faça um programa para testar o funcionamento da função a seguir, que preenche um vetor **v** com **n** inteiros aleatórios, gerados a partir da semente **s**, escolhidos no intervalo [0,999].

```
void preenche(int v[], int n, int s) {
    srand(s); // definida em stdlib.h
    for(int i=0; i<n; i++) v[i] = rand()%1000;
}
```

Ordenação

Exercício 7. Comparação entre `bsort()` e `msort()`

Complete o programa a seguir, execute-o e analise os resultados.

```
...
int main(void) {
   int v[1e5];
   double t, b, m;
   puts("      n bsort msort");
   for(int n=1e4; n<=1e5; n+=1e4) {
      preenche(v,n,1);
      t = clock();                        // definida em time.h
      bsort(v,n);
      b = (clock()-t)/CLOCKS_PER_SEC;   // tempo do bsort
      preenche(v,n,1);
      t = clock();
      msort(v,n);
      m = (clock()-t)/CLOCKS_PER_SEC;   // tempo do bsort
      printf("%6d %5.1f %5.1f\n",n,b,m);
   }
   return 0;
}
```

1e5 é o mesmo que 1×10^5.

Ordenação

Exercício 8. Desempenho de `msort()` para vetores muito grandes

Complete o programa a seguir, execute-o e analise os resultados.

```
...
int main(void) {
   // precisamos usar malloc para criar vetores muito grandes!
   int *v = malloc(1e8*sizeof(int));
   puts("         n msort");
   for(int n=1e7; n<=1e8; n+=1e7) {
      preenche(v,n,1);
      double t = clock();
      msort(v,n);
      double m = (clock()-t)/CLOCKS_PER_SEC;
      printf("%9d %5.1f\n",n,m);
   }
   free(v);
   return 0;
}
```

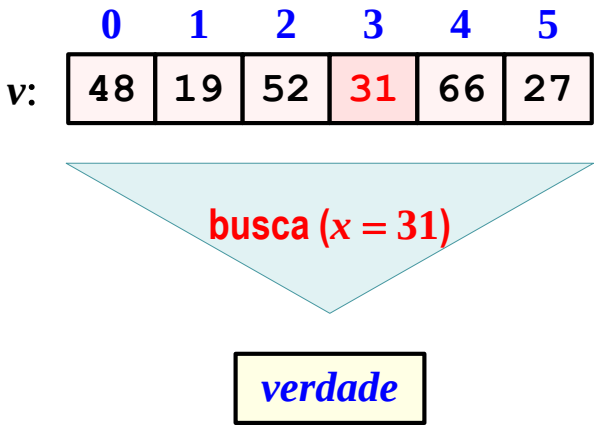
Em aplicações que precisam ordenar vetores bem pequenos, **`bsort()`** pode ser preferível!

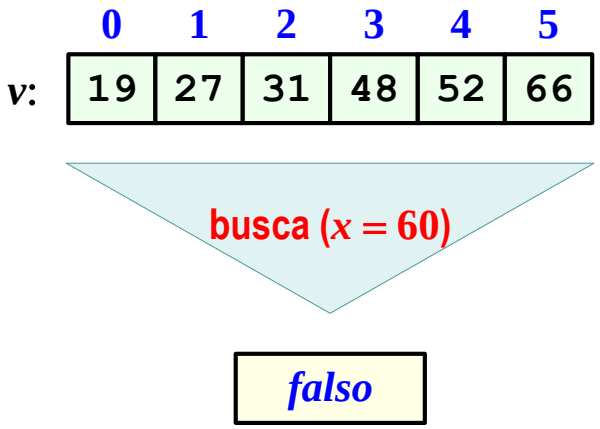
Busca

é a operação que verifica se um item **pertence** a um vetor.

Dados um item x e um vetor v, com n itens, a **busca** resulta em **verdade** se, e só se, existir um índice i tal que:

- $0 \leq i \leq n-1$ e $x = v[i]$.





A forma como os itens estão organizados no vetor pode influenciar no tempo de busca!

Busca

Busca linear

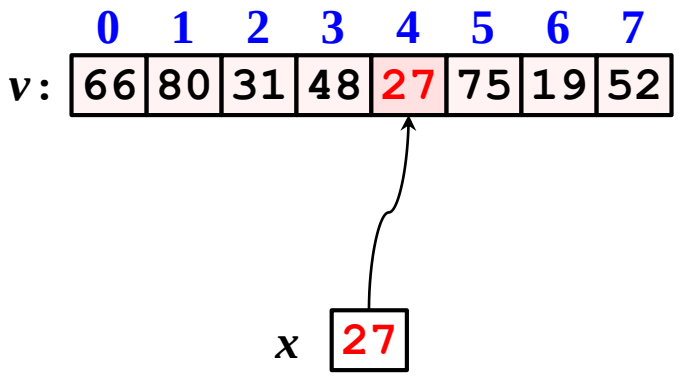
não supõe que o vetor onde é feita a busca está ordenado.

Estratégia:

Para determinar se um item x pertence a um vetor $v[0..n-1]$, usando busca linear:

- Para i variando de 0 até $n-1$, faça:
  - Se x for igual a $v[i]$, devolva **verdade**.
- Devolva **falso**.

Funcionamento:



No pior caso, a **busca linear** consome tempo proporcional a n.

Busca

Exemplo 6. Implementação da busca linear

```
int lsearch(int x, int v[], int n) {
   for(int i=0; i<n; i++)
      if( x == v[i] )
         return 1;
   return 0;
}
```

Exercício 9. Teste da função `lsearch()`

Complete e execute o programa a seguir:

```
#include <stdio.h>
...
int main(void) {
   int v[8] = {66,80,31,48,27,75,19,52};
   printf("27: %d\n", lsearch(27,v,8));
   printf("51: %d\n", lsearch(51,v,8));
   return 0;
}
```

Busca

Busca binária

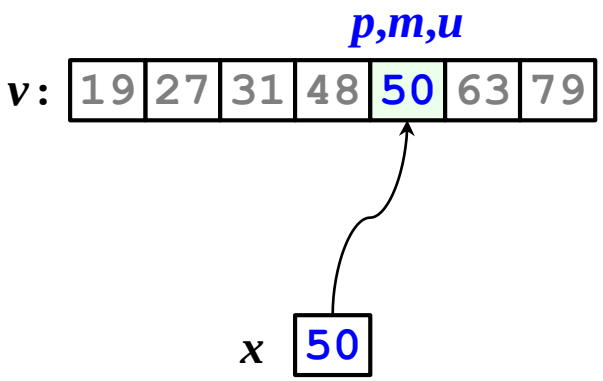
supõe que o vetor onde é feita a busca está **ordenado**.

Estratégia:

Para determinar se um item x pertence a um vetor crescente $v[0..n-1]$, usando busca binária:

- Faça $p = 0$.
- Faça $u = n-1$.
- Enquanto $p \leq u$:
  - Faça $m = \lfloor (p+u)/2 \rfloor$.
  - Se $x = v[m]$, devolva **verdade**.
  - Se $x < v[m]$, faça $u = m-1$.
  - Senão, faça $p = m+1$.
- Devolva **falso**.

Funcionamento:



No pior caso, a **busca binária** consome tempo proporcional a $\lg n$.

Busca

Exemplo 7. Implementação da busca binária

```
int bsearch(int x, int v[], int n) {
    int p = 0;
    int u = n-1;

    while( p<=u ) {
        int m = (p+u)/2;
        if( x==v[m] ) return 1;
        if( x<v[m] ) u = m-1;
        else p = m+1;
    }

    return 0;
}
```

Note que, nesta função, a expressão `(p+u)/2` sempre produz um resultado inteiro!

Busca

Exercício 10. Teste da função `bsearch()`

Complete e execute o programa a seguir:

```
#include <stdio.h>
...
int main(void) {
    int v[8] = {19,27,31,48,52,66,75,80};
    printf("27: %d\n", bsearch(27,v,8));
    printf("51: %d\n", bsearch(51,v,8));
    return 0;
}
```

Exercício 11. Versão recursiva de busca linear

Crie a função recursiva **`rlsearch(x,v,n)`**, que faz uma busca linear para determinar se o item **`x`** está no vetor **`v`**, que tem **`n`** itens.

Exercício 12. Versão recursiva de busca binária

Crie a função recursiva **`rbsearch(x,v,p,u)`**, que faz uma busca binária para determinar se o item **`x`** está no vetor crescente **`v`**, indexado de **`p`** até **`u`**.

Fim