Busca

Busca

- Dada uma lista (Vetor), como buscar um elemento nesta lista?
- Duas maneiras:
  - Busca sequencial: percorre-se a lista, elemento por elemento, até achar, ou não;
  - Busca binária: dado que a lista está ordenada, divide-a percorrendo a metade da esquerda ou a metade da direita.

Busca Sequencial

- É fácil de ser implementada:

```
função Busca(Lista L, int x, int n)
{
        int i ← 1;
        enquanto ( L[i].Chave != x && i <= n ) faça
                i ← i + 1;
        fimenquanto
        se ( i <= n ) então
                retorne i;
        senão
                retorne -1;
        fimse
}
```

Busca Binária

- Se a lista onde o dado deve ser buscado já está ordenada, podemos usar um método superior ao da busca sequencial;
- Este método utiliza a abordagem de dividir e conquistar;
- Primeiro verifica o elemento central. Se o elemento é maior que a Chave, o método testa a segunda metade. Caso contrário, testa a primeira metade.

Busca Binária

- Por exemplo, buscar o elemento 4 na lista: (1, 2, 3, 4, 5, 6, 7, 8, 9);
- Calculamos o meio da lista e comparamos o 4 com o elemento do meio;
- Como 5 > 4, realizamos a busca apenas na primeira metade (1, 2, 3, 4, 5);
- Novo elemento do meio: 3;
- Como 4 > 3, usamos a segunda metade (4, 5).

Busca Binária

- O número de comparações, no pior caso, é: $O(log_2 n)$;
- No melhor caso, exatamente $O(1)$.

Busca Binária

```
funcao Busca_Bin(Lista L, int x, int inic, int fim)
{
        int busca_bin = -1;
        int meio;
        enquanto ( inic <= fim ) faca
        {
                meio = (int)piso((inic + fim) / 2);
                se ( L [meio].Chave = x ) entao
                {
                        busca_bin = meio;
                        inic = fim + 1;
                }
                senao
                {
                        se ( x > L [meio].Chave) entao
                                inic = meio + 1
                        senao
                                fim = meio – 1;
                }
        }
        retorne busca_bin;
}
```

Busca Binária

- Desenvolver o algoritmo recursivo para a Busca Binária.

Algoritmos de Ordenação

- Entrada:
  - Uma sequência de n números (a_1, a_2, ..., a_n).
- Saída:
  - Uma permutação (reordenação) (a'_1, a'_2, ..., a'_n) da sequência de entrada, tal que $a'_1 <= a'_2 <= ... <= a'_n$.

Algoritmos de Ordenação

- Exemplo:
  - Dada uma sequência de entrada como (31, 41, 59, 26, 41, 58), um algoritmo de ordenação retorna como saída uma sequência (26, 31, 41, 41, 58, 59);
  - Ordenação é uma operação fundamental na área Computacional;
  - O melhor algoritmo de ordenação – entre outros fatores – depende do número de itens a serem ordenados.

Algoritmos de Ordenação

- Dois tipos de algoritmos:
  - Comparação e Troca;
  - Divisão e conquista.

Algoritmos de Ordenação

- Comparação e Troca:
  - Dados dois elementos, digamos a e b, e um aux;

```
se a > b então
    aux <- a;
    a <- b;
    b <- aux;
```

Algoritmos de Ordenação

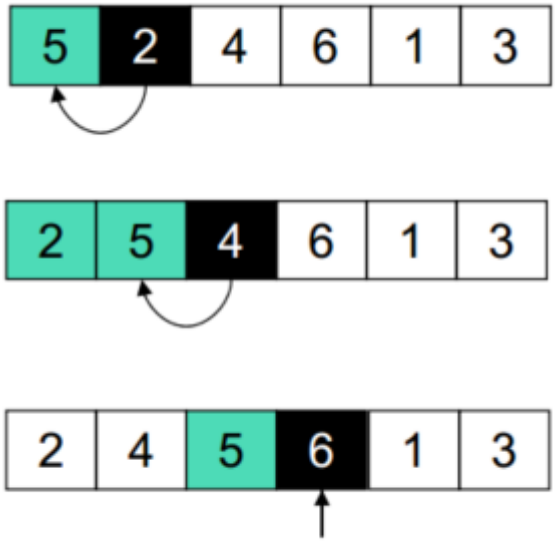
- Divisão e conquista:
  - Dividir: o problema em um determinado número de subproblemas;
  - Conquistar: os subproblemas, resolvendo-os recursivamente. Porém, se os tamanhos dos subproblemas forem pequenos o bastante, resolvemos de maneira direta;
  - Combinar: as soluções dadas aos subproblemas, a fim de formar a solução para o problema original.

Ordenação por Inserção (*Insertion Sort*)

- A ordenação por inserção é bastante simples e eficiente para uma quantidade pequena de números;
- Ideia: semelhante à ordenação de cartas de baralho na mão de um jogador:
  - A mão esquerda começa vazia e a mão direita insere uma carta de cada vez na posição correta. Ao final, quando todas as cartas foram inseridas, elas estarão ordenadas;
  - Para encontrar, durante a inserção, a posição correta para um valor, compara-se este valor um-a-um com as cartas da mão esquerda, até encontrarmos a posição correta.

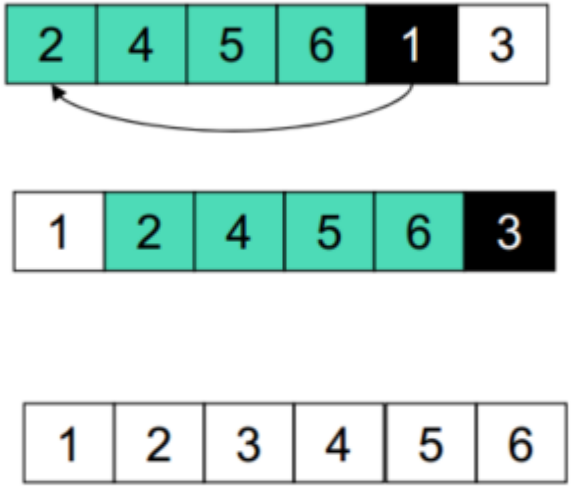
Ordenação por Inserção (*Insertion Sort*)

- Exemplo de funcionamento do algoritmo para a sequência (5,2,4,6,1,3):



Ordenação por Inserção (*Insertion Sort*)

- Exemplo de funcionamento do algoritmo para a sequência (5,2,4,6,1,3):



Ordenação por Inserção (*Insertion Sort*)

```
procedimento InsertionSort(A)
início
    para j de 2 até n faça
        chave ← A[j];
        // Inserir A[j] na posição correta.
        para i de j até i > 1 e A[i-1] > chave passo - 1 faça
            A[i] = A[i-1];
        fimpara
        A[i] ← chave;
    fimpara
fim
```

Ordenação por Inserção (*Insertion Sort*) – Implementação

- Como nós podemos implementar o algoritmo de ordenação *InsertionSort* ?
- Qual estrutura podemos utilizar ?
  - Cada elemento pode ser representado por uma chave;
  - Imaginem que cada elemento pode armazenar dados sobre um produto, como poderíamos estruturar isto?
  - Claro, usando *classes* em *Java*.

Ordenação por Inserção (*Insertion Sort*) – Implementação

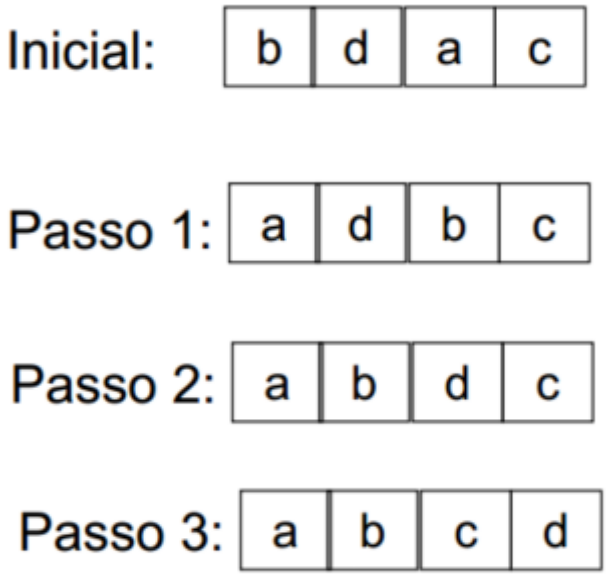
- Complexidade:
  - É fácil ver que, no pior caso, o algoritmo gasta um tempo na ordem $O(n^2)$.
  - Por que?

Ordenação por Seleção (*Selection Sort*)

- Ideia (para *n* elementos):
  - Seleciona o elemento de menor valor e troca-o pelo primeiro elemento da lista;
  - Para os n – 1 elementos restantes, seleciona o elemento de menor chave e este é trocado pelo segundo elemento da lista;
  - E assim por diante;
  - As trocas continuam até os dois últimos elementos.

Ordenação por Seleção (*Selection Sort*)

- Exemplo:



Ordenação por Seleção (*Selection Sort*)

```
procedimento SelectionSort(A)
{
        int i, j, min, aux;
        para i de 1 até n – 1 faça
                min ← i;
                para j de i+1 até n faça
                        se( A[j] < A[min] ) então
                                min ← j;
                fimpara
                se( min != i ) então
                        Troca(A[i], A[min]);
        fimpara
}
```

Ordenação por Seleção (*Selection Sort*) - Implementação

- Como nós podemos implementar o algoritmo de ordenação *SelectionSort* ?
- Qual estrutura podemos utilizar ?
  - Mesma situação do *InsertionSort*.

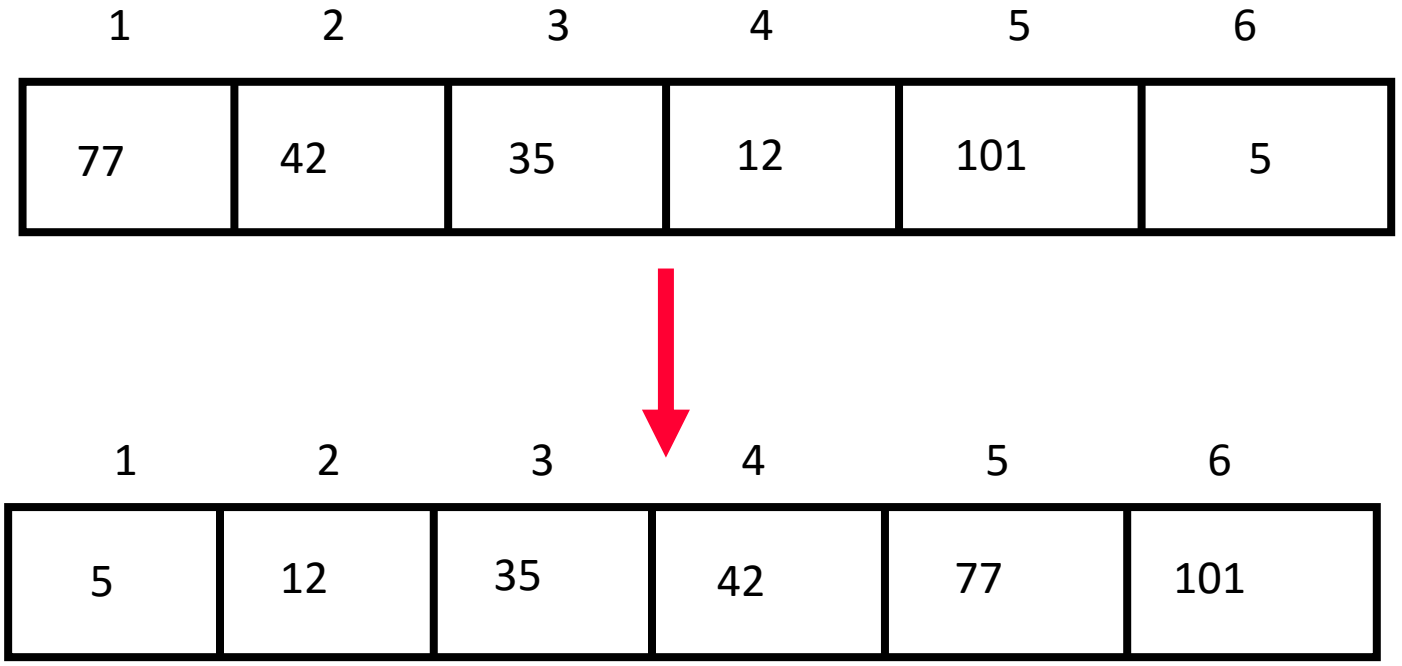
Ordenação por Seleção (*Selection Sort*) - Implementação

- Complexidade:
  - O laço mais externo é executado $n - 1$ vezes;
  - O laço interno é executado n – 1 vezes, depois n – 2 vezes, n – 3 vezes, e assim por diante;
  - Portanto, novamente a complexidade chega a $O(n^2)$.

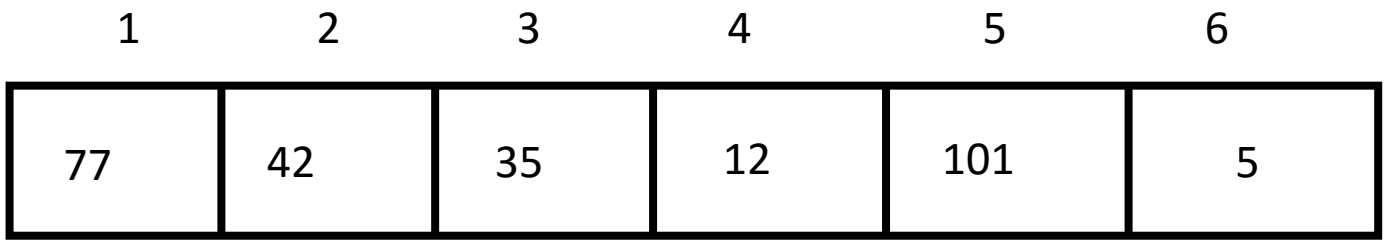
Ordenação *Bubble Sort*

- Método de ordenação mais conhecido;
  - Ordenação por trocas.
- Não tem boa eficiência;
- Ideia (*n* elementos):
  - Realiza n – 1 comparações e, se necessário, trocam-se dois elementos adjacentes;
  - Quantas vezes? Aproximadamente *n* vezes.

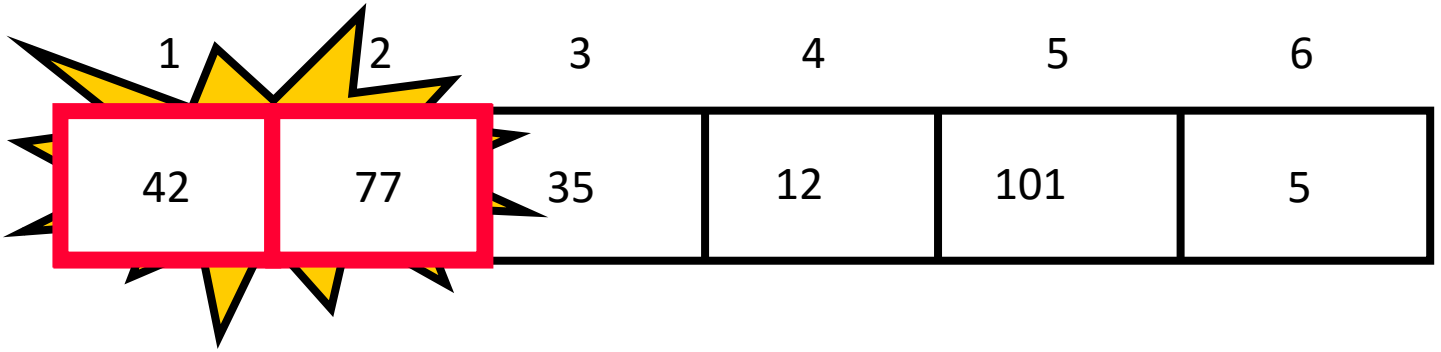
Ordenação *Bubble Sort*



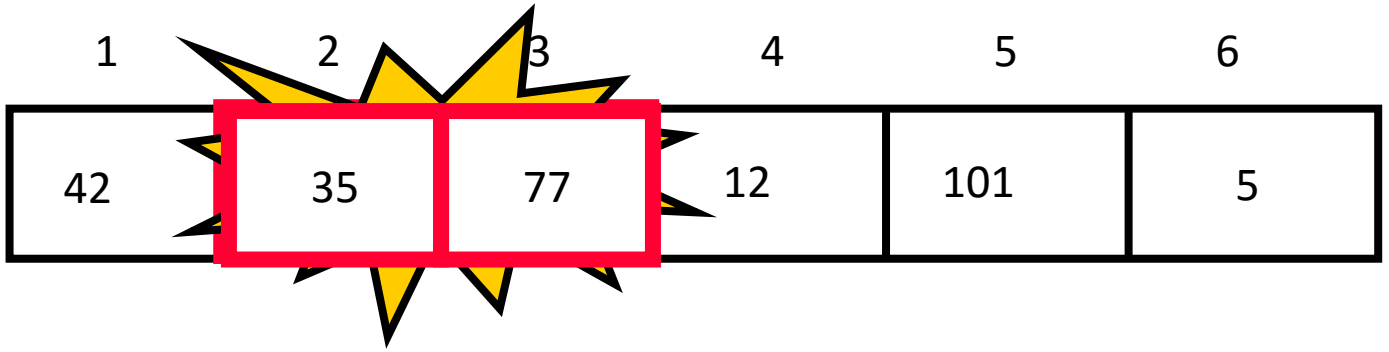
Ordenação *Bubble Sort*



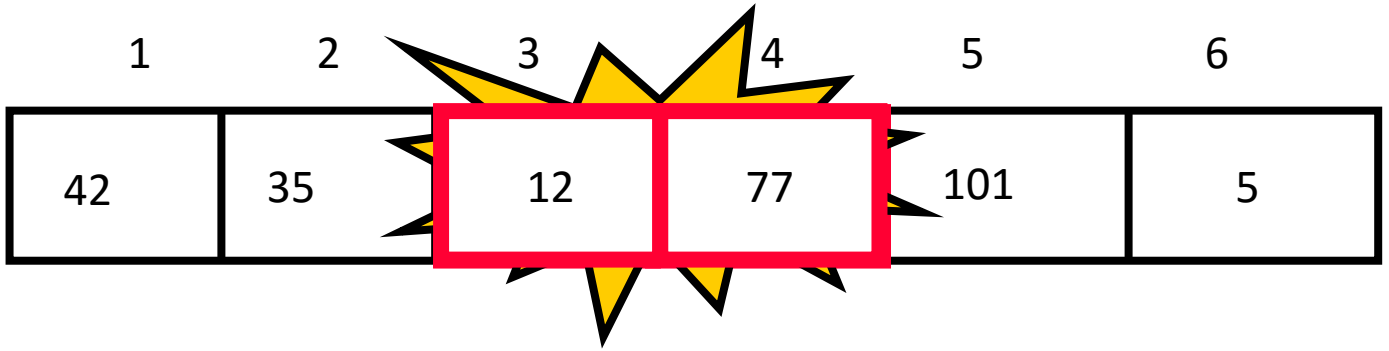
Ordenação *Bubble Sort*



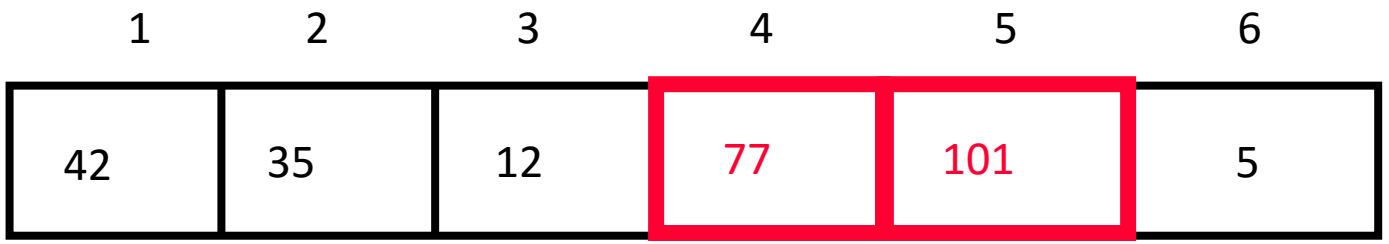
Ordenação *Bubble Sort*



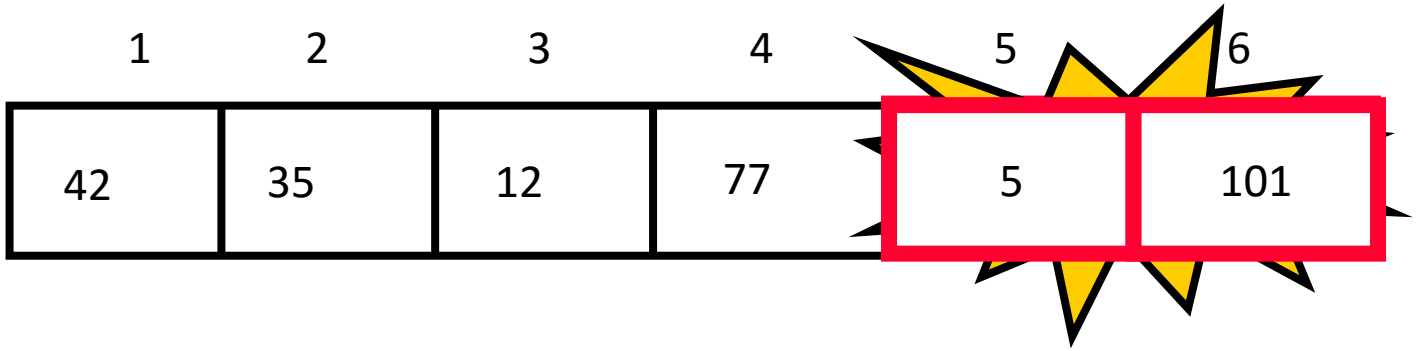
Ordenação *Bubble Sort*



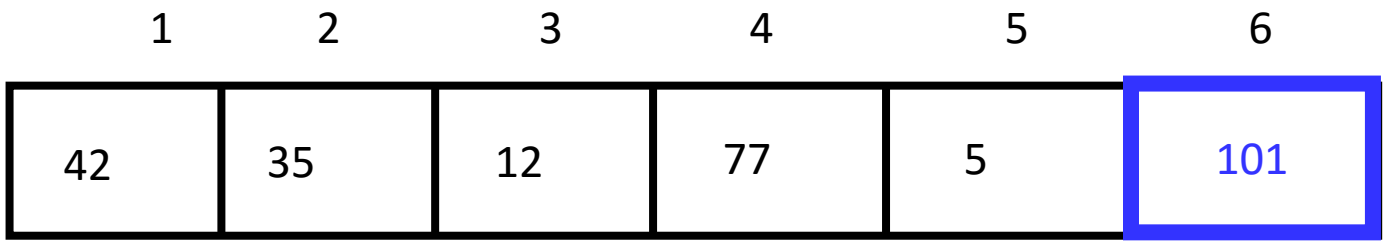
Ordenação *Bubble Sort*



Ordenação *Bubble Sort*



Ordenação *Bubble Sort*



Ordenação *Bubble Sort* - Implementação

```
procedimento BubbleSort(A)
{
    int i, j;
    para i de 1 até n - 1 faça
    {
        para j de 1 até n - 1 faça
        {
            se A[j] > A[j + 1] então
                Troca(A[j], A[j + 1]);
        }
    }
}
```

Ordenação *Bubble Sort* - Implementação

- Como nós podemos implementar o algoritmo de ordenação *BubbleSort* ?
- Qual estrutura podemos utilizar ?
  - Mesma situação do *InsertionSort*.
- Complexidade:
  - Complexidade chega a $O(n^2)$;
  - Por que? Verifiquem !!!