

N_EST DAD_A3 – Texto de Apoio

Site: [EAD Mackenzie](#)

Tema: ESTRUTURA DE DADOS {TURMA 03A} 2023/1

Livro: N_EST DAD_A3 – Texto de Apoio

Impresso por: FELIPE BALDIM GUERRA .

Data: quarta, 26 abr 2023, 00:52

Índice

ALGORITMOS DE ORDENAÇÃO

Algumas considerações...

Algoritmos de ordenação mais conhecidos

Ordenação por trocas – Bubble Sort

Ordenação por inserção – Insertion Sort

Ordenação por seleção – Selection Sort

ALGORITMOS DE BUSCA

Busca sequencial

Busca binária

Busca Sequencial versus Busca Binária

ALGORITMOS DE ORDENAÇÃO

Um algoritmo de ordenação coloca os elementos de uma lista em determinada ordem. A ordenação pode ser crescente ou decrescente e pode seguir uma das regras abaixo:

- ordem numérica; ou
- ordem lexicográfica (como as palavras de um dicionário).

Um método de ordenação eficiente é importante para otimizar o uso de outros algoritmos, como busca e mesclagem (merge) as quais requerem, como entrada, listas ordenadas. A ordenação é útil também para facilitar a recuperação dos dados pelas pessoas.

Em computação:

Ordenar (ou seja, rearranjar) os elementos de um vetor $v[0..n-1]$ de tal modo que eles fiquem em ordem crescente, ou seja, para que tenhamos

$$v[0] \leq v[1] \leq \dots \leq v[n-1]$$

Algumas considerações...

- Computadores gastam mais tempo ordenando do que fazendo qualquer outra coisa.
- A ordenação aparece em muitos problemas na prática e é o problema mais estudado em ciência da computação.
- A ordenação pode ser **interna**, quando o conjunto de dados a ser ordenado é armazenado inteiramente na memória principal do computador ou **externa**, quando o conjunto de dados não cabe na memória principal e precisa ser armazenado em um disco, por exemplo.
- Se os itens com chaves iguais permanecerem na mesma ordem (não posição!) que estavam nos dados iniciais após a ordenação, então, dizemos que o método é **estável**.
- Notação utilizada nos algoritmos:
 - Os algoritmos trabalham sobre os registros de um arquivo, sobre vetores de tipos primitivos ou ainda sobre vetores de objetos.
 - Independente da estrutura a ser ordenada, ela deve ter uma ou mais chaves que controlarão a ordenação. Por exemplo: ordem crescente de nome, ordem decrescente de salário etc.
- A escolha de um algoritmo de ordenação deve considerar o tempo gasto pela ordenação.
- Sendo **n** o número registros/dados a serem ordenados, as instruções relevantes, na questão de “contagem” do tempo são:
 - Número de comparações $C(n)$ entre chaves.
 - Número de movimentações $M(n)$ de itens do arquivo.

Algoritmos de ordenação mais conhecidos

- Ordenação pelo método das trocas (Bubble Sort).
- Ordenação por inserção (Insertion Sort).
- Ordenação por seleção (Selection Sort).
- Ordenação por intercalação (Merge Sort).
- Ordenação por heaps (Heap Sort).
- Ordenação rápida (Quick Sort).

Neste primeiro contato com ordenação, veremos apenas os métodos **Bubble Sort**, **Insertion Sort** e **Selection Sort**.

Ordenação por trocas – Bubble Sort

O Bubble Sort data da década de 1950 e, apesar de ser antigo, é o método mais simples e mais conhecido. Porém, é também o menos eficiente (realiza muitos passos para completar a classificação). Nesse método, a chave de maior/menor valor é colocada no final/início do vetor a cada varredura efetuada.

Considere o vetor abaixo. Iremos ordená-lo de forma crescente (do menor para o maior):

1ª Varredura:

45	67	12	34	25	39
----	----	----	----	----	----

São comparados os elementos 0 e 1 do vetor (45 e 67) e são trocados de posição caso necessário. Nesse caso, não será feita a troca.

45	67	12	34	25	39
----	----	----	----	----	----

O foco, então, passa para os elementos 1 e 2 do vetor (67 e 12). Nesse caso, eles serão trocados de posição.

45	12	67	34	25	39
----	----	----	----	----	----

O foco, então, passa para os elementos 2 e 3 do vetor (67 e 34). Nesse caso, eles serão trocados de posição.

45	12	34	67	25	39
----	----	----	----	----	----

O foco, então, passa para os elementos 3 e 4 do vetor (67 e 25). Nesse caso, eles serão trocados de posição.

45	12	34	25	67	39
----	----	----	----	----	----

O foco, então, passa para os elementos 4 e 5 do vetor (67 e 39). Nesse caso, eles serão trocados de posição.

45	12	34	25	39	67
----	----	----	----	----	----

O vetor, então, fica com o número 67 em sua posição definitiva. Ou seja, ele é o maior valor do vetor!

Aqui, termina a primeira varredura. Repare que ainda não temos o vetor ordenado. Esse processo se repetirá até que todas as chaves ocupem suas posições finais.

2ª Varredura

45	12	34	25	39	67
----	----	----	----	----	----

São comparados os elementos 0 e 1 do vetor (45 e 12) e são trocados de posição caso necessário. Nesse caso, será feita a troca.

12	45	34	25	39	67
----	----	----	----	----	----

São comparados os elementos 1 e 2 do vetor (45 e 34). Nesse caso, eles serão trocados de posição.

12	34	45	25	39	67
----	----	----	----	----	----

São comparados os elementos 2 e 3 do vetor (45 e 25). Nesse caso, eles serão trocados de posição.

12	34	25	45	39	67
----	----	----	----	----	----

São comparados os elementos 3 e 4 do vetor (45 e 39). Nesse caso, eles serão trocados de posição.

12	34	25	39	45	67
----	----	----	----	----	----

O vetor, então, fica com o número 45 em sua posição definitiva.

Aqui, termina a segunda varredura.

3ª Varredura:

12	34	25	39	45	67
----	----	----	----	----	----

São comparados os elementos 0 e 1 do vetor (12 e 34) e são trocados de posição caso necessário. Nesse caso, não será feita a troca.

12	34	25	39	45	67
----	----	----	----	----	----

São comparados os elementos 1 e 2 do vetor (34 e 25). Nesse caso, eles serão trocados de posição.

12	25	34	39	45	67
----	----	----	----	----	----

São comparados os elementos 2 e 3 do vetor (34 e 39). Nesse caso, eles não são trocados de posição.

12	25	34	39	45	67
----	----	----	----	----	----

O vetor, então, fica com o número 39 em sua posição definitiva.

Aqui, termina a terceira varredura.

4ª Varredura:

12	25	34	39	45	67
----	----	----	----	----	----

São comparados os elementos 0 e 1 do vetor (12 e 25) e são trocados de posição caso necessário. Nesse caso, não será feita a troca.

12	25	34	39	45	67
----	----	----	----	----	----

São comparados os elementos 1 e 2 do vetor (25 e 34). Nesse caso, não será feita a troca.

12	25	34	39	45	67
----	----	----	----	----	----

O vetor, então, fica com o número 34 em sua posição definitiva.

Aqui, termina a quarta varredura.

5ª Varredura:

12	25	34	39	45	67
----	----	----	----	----	----

São comparados os elementos 0 e 1 do vetor (12 e 25) e são trocados de posição caso necessário. Nesse caso, não será feita a troca.

12	25	34	39	45	67
----	----	----	----	----	----

O vetor, então, fica com os números 12 e 25 em suas posições definitivas.

Aqui, termina a quinta e última varredura.

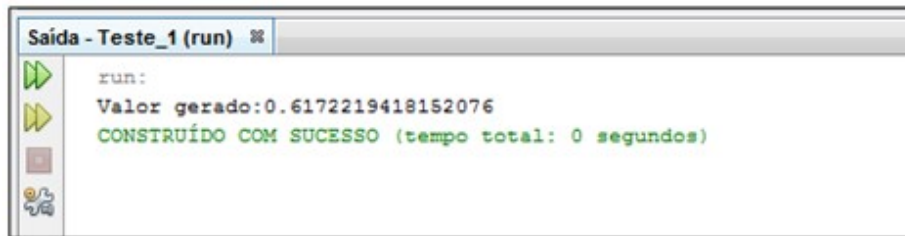
IMPLEMENTAÇÃO

A implementação desse método é bastante simples, não exigindo nenhum grande artifício. O programa é controlado por um looping, que é executado enquanto ocorrerem trocas durante uma varredura.

Neste exemplo, criaremos um vetor com 50 números inteiros que serão “sorteados” aleatoriamente pelo Java. Usaremos, para isso, a função matemática `Math.random()`. Veja como funciona:

```
double x = Math.random();  
System.out.println("Valor gerado:" + x);
```

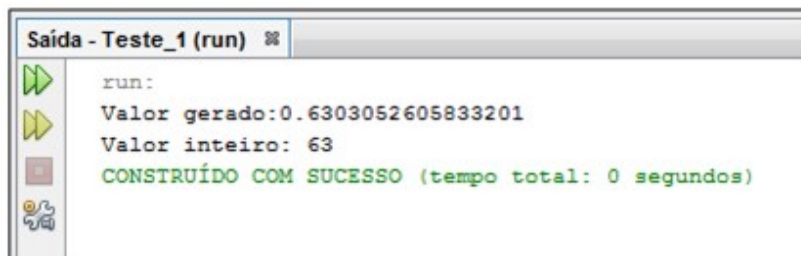
Repare que o comando `Math.random()` gera um número do tipo `double` (real) entre 0 e 1. Se executarmos esse trecho de código, a saída gerada poderia ser (a cada vez que executamos o programa, um número diferente é gerado):



```
Saída - Teste_1 (run) ✖  
run:  
Valor gerado:0.6172219418152076  
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Como nós queremos números inteiros, basta multiplicarmos o número gerado (`double`) por um valor inteiro. O resultado será sempre em função do número que você usou na multiplicação:

```
double x = Math.random();  
System.out.println("Valor gerado:" + x);  
int y = (int) (100 * x);  
System.out.println("Valor inteiro: " + y);
```



```
Saída - Teste_1 (run) ✖  
run:  
Valor gerado:0.6303052605833201  
Valor inteiro: 63  
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Vamos ao código do Bubble Sort:

```
public class NewClass {  
  
    public static void main(String[] args) {  
        int[] lista = new int[50];  
        sorteia(lista);  
        bubbleSort(lista);  
        exhibe(lista);  
    }  
  
    public static void sorteia(int[] lista) {  
        // sorteia valores para o vetor e os exibe na tela  
        System.out.println("Vetor antes da ordenação");  
        for (int i = 0; i < lista.length; i++) {  
            lista[i] = (int) (100 * Math.random());  
            System.out.print(lista[i] + " ");  
        }  
    }  
}
```

```
public static void bubbleSort(int[] lista) {  
    int temp;  
    // inicia o Bubble Sort  
    for (int i = 0; i < lista.length; i++) {  
        for (int j = 1; j < lista.length - i; j++) {  
            if (lista[j - 1] > lista[j]) {  
                //troca os elementos  
                temp = lista[j - 1];  
                lista[j - 1] = lista[j];  
                lista[j] = temp;  
            }  
        }  
    }  
}
```

```
public static void bubbleSort(int[] lista) {  
    int temp;  
    // inicia o Bubble Sort  
    for (int i = 0; i < lista.length; i++) {  
        for (int j = 1; j < lista.length - i; j++) {  
            if (lista[j - 1] > lista[j]) {  
                //troca os elementos  
                temp = lista[j - 1];  
                lista[j - 1] = lista[j];  
                lista[j] = temp;  
            }  
        }  
    }  
}
```

```
public static void exibe(int[] lista) {  
    // exibe vetor ordenado  
    System.out.println("\nVetor após a ordenação");  
    for (int i = 0; i < lista.length; i++) {  
        System.out.print(lista[i] + " ");  
    }  
}
```

Este algoritmo tem complexidade QUADRÁTICA, ou seja, $O(n^2)$. Se n dobra, o número de operações quadruplica.

Ordenação por inserção – Insertion Sort

A classificação por inserção é caracterizada pelo princípio no qual os n dados a serem ordenados são divididos em dois segmentos: um já ordenado e outro a ser ordenado.

Inicialmente, o primeiro segmento é formado apenas pelo primeiro elemento (já ordenado). O segundo segmento é formado pelos restantes $n-1$ elementos. A partir daí, o processo se desenvolve em $n-1$ iterações, sendo que, em cada uma delas, um elemento do segmento não ordenado é transferido para o segmento já ordenado, sendo inserido em sua posição correta.

15	27	8	42	30	18	9
Segmento ordenado	Segmento não ordenado					

A partir daí, “pega-se” o primeiro elemento do segmento não-ordenado e localiza-se sua posição correta no segmento ordenado, da direita para a esquerda.

Cada comparação realizada entre o elemento a ser inserido e os que estão no segmento ordenado pode gerar duas possibilidades: caso o elemento a ser inserido seja maior que todos que estão no segmento já ordenado, a inserção corresponde a deixá-lo na posição que já ocupava.

Após a inserção, a divisa entre os dois segmentos é deslocada uma posição para a direita, indicando que o segmento ordenado ganhou mais um elemento e o desordenado perdeu um. Este processo é repetido até que todos os elementos do segmento desordenado tenham sido inseridos no primeiro.

15	27	8	42	30	18	9
----	----	---	----	----	----	---

O número 27 é comparado com 15. Como ele é maior, ele é transferido para o segmento ordenado à direita do 15.

15	27		8	42	30	18	9
O número 8 é comparado com 27 e 15. Como ele é menor que os dois, ele é transferido para o segmento ordenado à esquerda do 15.							

8	15	27		42	30	18	9
O número 42 é comparado com 27. Como ele é maior que todos, ele é transferido para o segmento ordenado à direita do 27.							

8	15	27	42		30	18	9
O número 30 é comparado com 42 e 27. Ele é maior que 27 e menor que 42, portanto, será transferido para o segmento ordenado à direita do 27.							

8	15	27	30	42		18	9
O número 18 é comparado com 42, 30, 27 e 15. Ele é maior que 15 e menor que 27, portanto, será transferido para o segmento ordenado à direita do 15.							

8	15	18	27	30	42		9
---	----	----	----	----	----	--	---

O número 9 é comparado com 42, 30, 27, 18, 15 e 8. Ele é maior que 8 e menor que 15, portanto, será transferido para o segmento ordenado à direita do 8.

8	9	15	18	27	30	42
O vetor encontra-se totalmente ordenado.						

IMPLEMENTAÇÃO

O segmento já ordenado é percorrido da direita para a esquerda, até que seja encontrada uma chave menor ou igual àquela que está sendo inserida, ou até que o segmento termine. Enquanto nenhuma das condições ocorrer, as chaves comparadas vão sendo deslocadas uma posição para a direita. Na hipótese da chave a ser inserida ser maior que todas as chaves do segmento ordenado, ela permanece em seu local original; caso contrário, é inserida na posição deixada vaga pelos deslocamentos, avançando-se, a seguir, a fronteira entre os dois segmentos. O processo todo é completado em n-1 iterações.

```
public class NewClass {  
  
    public static void main(String [] args){  
        int [] lista = new int[50];  
        geraVetor(lista);  
        insertionSort(lista);  
        exhibeVetor(lista);  
    }  
  
    static void geraVetor (int[] lista) {  
        System.out.println("Vetor antes da ordenação");  
        for (int i=0;i<lista.length;i++){  
            lista[i]=(int) (100*Math.random());  
            System.out.print(lista[i] + " ");  
        }  
    }  
}
```



```

static void insertionSort (int[] lista) {
    int j, aux;
    for (int i=1; i<lista.length; i++) {
        aux=lista[i];
        for (j=i-1; j>=0 && aux < lista[j]; j--)
            lista[j+1]=lista[j];
        lista[j+1]=aux;
    }
}

static void exibeVetor (int[] lista) {
    System.out.println("\nVetor após ordenação");
    for (int i=0; i<lista.length; i++){
        System.out.print(lista[i] + " ");
    }
}
}

```

Esse algoritmo tem complexidade QUADRÁTICA, ou seja, $O(n^2)$. Se n dobra, o número de operações quadruplica.

Ordenação por seleção – Selection Sort

Este método caracteriza-se por procurar, a cada iteração, a chave de menor (ou maior) valor do vetor e colocá-la em sua posição definitiva correta, seja no início ou no final, por troca com a chave que ocupava aquela posição. O vetor a ser classificado fica, então, reduzido de um elemento. O mesmo processo é repetido para o resto do vetor, até que este fique reduzido a um elemento, quando, então, a classificação estará concluída.

A busca pela menor chave é feita por pesquisa sequencial. Uma vez encontrada, esta é trocada com a chave que ocupa a posição inicial do vetor. O processo de seleção é repetido para o restante do vetor.

15	29	38	6	55	1	62	13	Na primeira varredura, a chave de menor valor é selecionada e colocada na posição inicial do vetor. Chave selecionada=1 é movida para a posição do número 15.
----	----	----	---	----	---	----	----	--

1	29	38	6	55	15	62	13	Na segunda varredura, a chave de menor valor é selecionada e colocada na posição definitiva. Chave selecionada=6 é movida para a posição do número 29.
---	----	----	---	----	----	----	----	---

1	6	38	29	55	15	62	13	Na terceira varredura, a chave de menor valor é selecionada e colocada na posição definitiva. Chave selecionada=13 é movida para a posição do número 38.
---	---	----	----	----	----	----	----	---

1	6	13	29	55	15	62	38	Na quarta varredura, a chave de menor valor é selecionada e colocada na posição definitiva. Chave selecionada=15 é movida para a posição do número 29.
---	---	----	----	----	----	----	----	---

1	6	13	15	55	29	62	38	Na quinta varredura, a chave de menor valor é selecionada e colocada na posição definitiva. Chave selecionada=29 é movida para a posição do número 55.
---	---	----	----	----	----	----	----	---

1	6	13	15	29	55	62	38	Na sexta varredura, a chave de menor valor é selecionada e colocada na posição definitiva. Chave selecionada=38 é movida para a posição do número 55.
---	---	----	----	----	----	----	----	--

1	6	13	15	29	38	62	55	Na sétima e última varredura, a chave de menor valor é selecionada e colocada na posição definitiva. Chave selecionada=55 é movida para a posição do número 62.
---	---	----	----	----	----	----	----	--

1	6	13	15	29	38	55	62	Note que o vetor já está ordenado após a sétima varredura.
---	---	----	----	----	----	----	----	--

IMPLEMENTAÇÃO

Para esse método, precisamos selecionar o menor elemento do vetor e trocá-lo pelo primeiro. Essas operações serão repetidas, considerando apenas os n-1 elementos que ainda não estão selecionados.

```
public class NewClass {  
    public static void main(String [] args){  
        int [] lista = new int[50];  
        geraVetor(lista);  
        selectionSort(lista);  
        exhibeVetor(lista);  
    }  
  
    static void geraVetor (int[] lista) {  
        System.out.println("Vetor antes da ordenação");  
        for (int i=0;i<lista.length;i++){  
            lista[i]=(int) (100*Math.random());  
            System.out.print(lista[i] + " ");  
        }  
    }  
}
```

```

static void selectionSort(int[] lista){
    int j, min, aux;
    for (int i=0;i<lista.length-1;i++) {
        min=i;
        for (j=i+1;j<lista.length;j++)
            if (lista[j]<lista[min])
                min=j;
        if (i!=min){
            aux=lista[i];
            lista[i]=lista[min];
            lista[min]=aux;
        }
    }
}

static void exibeVetor (int[] lista) {
    System.out.println("\nVetor após ordenação");
    for (int i=0;i<lista.length;i++){
        System.out.print(lista[i] + " ");
    }
}
}

```

Esse algoritmo tem complexidade QUADRÁTICA, ou seja, $O(n^2)$. Se n dobra, o número de operações quadruplica.

ALGORITMOS DE BUSCA

Existem basicamente dois modelos de algoritmos de busca: a busca sequencial e a busca binária. Devemos conhecer o funcionamento, discutir e analisar os algoritmos de busca antes de utilizá-los para uma aplicação específica.

Busca sequencial

Nós já utilizamos o algoritmo de busca sequencial na aula sobre Listas Lineares. Dentro da classe Vetor, fizemos um método chamado search:

```
public int search(String nomeBusca) {  
    for (int i = 0; i < qtde; i++) {  
        if (vetor[i].nome.equals(nomeBusca)) {  
            return i;  
        }  
    }  
    return -1;  
}
```

Basicamente, estamos procurando um determinado elemento em uma lista. Podemos não saber se o elemento PROCURADO existe na lista antes da pesquisa. A Busca Sequencial funciona da mesma forma, independentemente da estrutura de dados utilizada (vetor, arrayList etc.).

Normalmente, não temos garantia sobre a ordem dos elementos na lista, se (por exemplo) as inserções aconteceram sob o controle de um usuário. Em tais circunstâncias, uma busca sequencial a partir da posição 0 é tão eficiente quanto qualquer outro método, para encontrar um valor. Assim, a busca começa no primeiro elemento da lista e continua até que o item seja encontrado ou toda a lista tenha sido percorrida.

Dado um vetor V e um elemento E, o **problema da busca** consiste em **verificar se E pertence a V** e, normalmente, quer se saber, também, qual é **primeira posição em que o elemento E aparece no vetor**.

0	1	2	3	4	5	6
8	7	5	6	7	-3	4

Na busca sequencial, parte-se da primeira posição do vetor e, a cada elemento consultado, verificamos se ele é o procurado: caso afirmativo, devolve-se o índice. Em caso negativo, isto é, o elemento não está no vetor, devolve-se -1.

A sequência abaixo ilustra a busca pelo elemento 6 no vetor V:

0	1	2	3	4	5	6
8	7	5	6	7	-3	4
?						

0	1	2	3	4	5	6
8	7	5	6	7	-3	4
	?					

0	1	2	3	4	5	6
8	7	5	6	7	-3	4
		?				

0	1	2	3	4	5	6
8	7	5	6	7	-3	4
			?			

Pela simulação da busca sequencial acima, concluímos que o elemento 6 está na posição 3 do vetor.

Analizando esse algoritmo

As instruções do loop “for” são repetidas várias vezes. Para cada iteração do loop, o item de pesquisa é comparado com um elemento da lista. O loop termina quando o item de busca é encontrado na lista ou quando o elemento não é encontrado na lista. Para efeito de cálculo, contaremos o número de comparações realizadas.

- Nós queremos determinar o número de comparações feitas pelo algoritmo de Busca Sequencial, quando a lista é percorrida para encontrar determinado valor.
- Se o item não estiver na lista, então, n comparações são realizadas (todos os itens da lista devem ser verificados) antes do retorno indicando que não existe o elemento.
- MELHOR CASO – o valor procurado é o primeiro – 1 comparação – $O(1)$
- PIOR CASO – o valor procurado é o último valor da lista – n comparações – $O(n)$
- CASO MÉDIO – $(1+n)/2$ – $O(n)$

O que acontece se tivermos de buscar uma palavra em um vetor de 100.000.000.000 de elementos? A Busca Sequencial consumirá MUITO tempo...

Busca binária

Quando **o vetor está ordenado**, podemos utilizar uma técnica de divisão-e-conquista chamada busca binária:

- Acessamos o elemento central do vetor.
- Se o elemento procurado for o elemento do meio, pare. Se o elemento procurado for menor que o elemento do meio, repetimos o processo para os elementos menores que o elemento do meio; caso contrário, repetimos o processo nos elementos maiores que o elemento do meio.

Vamos localizar o número 34 na lista abaixo:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	23	34	38	41	44	49	51	53	57	61	67	71	77	98

MEIO

↓

O processo se repete com a primeira metade da lista. Vamos localizar o valor 34 na lista abaixo:



0	1	2	3	4	5	6	7
12	23	34	38	41	44	49	51

MEIO

↓

O processo se repete... Vamos localizar o valor 34 na lista abaixo:

MEIO
↓

0	1	2	3	4
12	23	34	38	41

**ELEMENTO ENCONTRADO COM
TRÊS COMPARAÇÕES!**

IMPLEMENTAÇÃO

```
int pesquisaBinaria(int X, int numero[]) {  
    int inf, sup, meio;  
    inf = 0;  
    sup = numero.length - 1;  
    while (inf <= sup) {  
        meio = (inf + sup) / 2;  
        if (X == numero[meio]) {  
            return meio;  
        } else if (X < numero[meio]) {  
            sup = meio - 1;  
        } else {  
            inf = meio + 1;  
        }  
    }  
    return -1;  
    /* não encontrado */  
}
```

Analizando esse algoritmo

A cada passo do algoritmo, o tamanho do vetor é reduzido aproximadamente pela metade. No pior caso, quando o elemento não está no vetor, a redução do vetor vai até o tamanho 1. A partir desse ponto, $low > high$ e o algoritmo termina. Para se atingir o tamanho 1, devemos calcular a quantidade de passos k para chegar a tal tamanho:

$$n, n/2, n/4, \dots, n/2^k = 1$$

$$2^k = n \Rightarrow k = \log_2 n \text{ passos}$$

Como em cada passo as operações de comparação e divisão pela metade podem ser feitas em tempo $O(1)$, a complexidade total da busca binária é $O(\log_2 n)$. Normalmente, omite-se a base 2 e é comum dizermos que este algoritmo é **$O(\log n)$** .

Para n muito grande, $\log n$ é bem menor que n . Assim, um algoritmo $O(\log n)$ é mais eficiente que um $O(n)$. Isso nos permite concluir que, para vetores ordenados, a **busca binária [$O(\log n)$] é mais eficiente que a busca sequencial [$O(n)$]**.

Busca Sequencial versus Busca Binária

- O benefício de busca binária sobre busca sequencial torna-se significativo para as listas com mais de 100 elementos. Para listas menores, a busca sequencial pode ser mais rápida.
- A regra geral é que, para listas grandes, a busca binária é muito mais rápida do que a busca sequencial, mas não vale a pena para listas pequenas.
- Note que a pesquisa binária não é apropriada para estruturas de lista ligada (sem acesso aleatório para o termo central).