ORDENAÇÃO E BUSCA

Em computação, freqüentemente, armazenamos dados que, mais tarde, precisam ser recuperados. Como veremos, a eficiência na busca de informações depende, essencialmente, da ordem em que esses dados são guardados.

Silvio Lago

1. MÉTODOS DE ORDENAÇÃO

Seja $v[1..n]$ um vetor cujos n itens aparecem numa ordem aleatória. A *ordenação* consiste em se determinar uma permutação dos itens em v tal que tenhamos $v[1] \le v[2] \le v[3] \le \ldots \le v[n]$.

1.1. ORDENAÇÃO POR TROCAS

Talvez a estratégia mais simples para ordenar um vetor seja comparar pares de itens consecutivos e permutá-los, caso estejam fora de ordem. Se o vetor for assim processado, sistematicamente, da esquerda para a direita, um item máximo será deslocado para sua última posição.

Exemplo 1.1. Veja essa estratégia aplicada a um vetor de números:

$v[1]$	$v[2]$	$v[3]$	$v[4]$	$v[5]$	$v[6]$	$v[7]$	$v[8]$	$v[9]$
71	**63**	46	80	39	92	55	14	27
63	**71**	**46**	80	39	92	55	14	27
63	46	**71**	**80**	39	92	55	14	27
63	46	71	**80**	**39**	92	55	14	27
63	46	71	39	**80**	**92**	55	14	27
63	46	71	39	80	**92**	**55**	14	27
63	46	71	39	80	55	**92**	**14**	27
63	46	71	39	80	55	14	**92**	**27**
63	46	71	39	80	55	14	27	**92**

À medida em que o vetor vai sendo processado, cada número vai sendo deslocado para a direita, até que seja encontrado outro maior. Evidentemente, no final do processo, um valor máximo estará na última posição do vetor. ☑

Em cada fase desse método, um item de maior valor é deslocado para sua posição definitiva no vetor ordenado. Então, se um vetor tem n itens, após a primeira fase haverá $n-1$ itens a ordenar. Usando a mesma estratégia, após a segunda fase, teremos $n-2$ itens, depois $n-3$ e assim sucessivamente até que reste um único item.

Se considerarmos os números maiores como mais pesados e os menores como mais leves, veremos que, durante a ordenação por esse método, os números mais pesados "descem" rapidamente para o "fundo" do vetor, enquanto que os números mais leves "sobem" lentamente para a "superfície". Os números pesados descem como pedras e os leves sobem como bolhas[1] de ar.

Embora seja um dos métodos de ordenação menos eficientes que existem, o método da bolha é geralmente o primeiro algoritmo de ordenação que todo mundo aprende. A sua popularidade deve-se, principalmente, à sua simplicidade e facilidade de codificação. Métodos mais eficientes, em geral, são também mais complicados de entender e de codificar.

Exemplo 1.2. Veja agora a ordenação completa de um vetor pelo método da bolha. A variável *i* indica a fase da ordenação e *j* indica a posição do par de itens consecutivos que serão comparados e, eventualmente, permutados.

fase	***i***	***j***	***v*[1]**	***v*[2]**	***v*[3]**	***v*[4]**	***v*[5]**
1º	1	1	**46**	**39**	55	14	27
		2	39	**46**	**55**	14	27
		3	39	46	**55**	**14**	27
		4	39	46	14	**55**	**27**
			39	46	14	27	**55**
2º	2	1	**39**	**46**	14	27	**55**
		2	39	**46**	**14**	27	**55**
		3	39	14	**46**	**27**	**55**
			39	14	27	**46**	**55**
3º	3	1	**39**	**14**	27	**46**	**55**
		2	14	**39**	**27**	**46**	**55**
			14	27	**39**	**46**	**55**
4º	4	1	**14**	**27**	**39**	**46**	**55**
			14	**27**	**39**	**46**	**55**

Note que na última fase o vetor já está ordenado mas, mesmo assim, uma última comparação precisa ser feita para que isso seja confirmado. ☑

Observando o exemplo acima, podemos constatar que para ordenar *n* itens bastam apenas *n*–1 fases e que, numa determinada fase *i*, são realizadas *n*–*i* comparações.

[1] *Daí esse método ser conhecido como método da bolha ou bubble sort.*

Exemplo 1.3. Ordenação por trocas: *bubble sort*.

```
...
const max = 5;
type vetor = array[1..max] of integer;
...
procedure bsort(var v:vetor);
var i, j, x : integer;
begin
   for i:=1 to max-1 do
      for j:=1 to max-i do
         if v[j]>v[j+1] then
            begin
               x := v[j];
               v[j] := v[j+1];
               v[j+1] := x;
            end;
end;
```
☑

Análise da ordenação por trocas

Analisando a rotina *bsort()*, verificamos que o número total de comparações realizadas é $(n-1)+(n-2)+(n-3)+\ldots+2+1 = \frac{1}{2}(n^2-n)$, ou seja, a sua complexidade[2] de tempo é $O(n^2)$. Como a cada comparação corresponde uma troca em potencial, no pior caso, isto é, quando o vetor estiver em ordem decrescente, serão realizadas no máximo $\frac{1}{2}(n^2-n)$ trocas.

Exercício 1.1. Simule a execução do procedimento *bsort()*, no estilo do exemplo 1.2, para ordenar o vetor $v=\langle 92, 80, 71, 63, 55, 41, 39, 27, 14\rangle$.

Exercício 1.2. Adapte o procedimento *bsort()* de modo que o vetor seja ordenado de forma decrescente.

1.2. ORDENAÇÃO POR SELEÇÃO

A estratégia básica desse método é, em cada fase, selecionar um menor item ainda não ordenado e permutá-lo com aquele que ocupa a sua posição na seqüência ordenada. Mais precisamente, isso pode ser descrito assim: para ordenar uma seqüência $\langle a_i, a_{i+1}, \ldots, a_n\rangle$, selecione uma valor k tal que $a_k = min\{a_i, a_{i+1}, \ldots, a_n\}$, permute os elementos a_i e a_k e, se $i+1<n$, repita o procedimento para ordenar a subseqüência $\langle a_{i+1}, \ldots, a_n\rangle$.

---

[2] *A complexidade determina a eficiência do método; quanto maior for a taxa de crescimento da função, menos eficiente é o método.*

Exemplo 1.4. Ordenação da seqüência ⟨46, 55, 59, 14, 38, 27⟩ usando seleção.

Fase	***i***	***k***	a_1	a_2	a_3	a_4	a_5	a_6
1º	1	4	**46**	55	59	**14**	38	27
2º	2	6	**14**	**55**	59	46	38	**27**
3º	3	5	**14**	**27**	**59**	46	**38**	55
4º	4	4	**14**	**27**	**38**	**46**	59	55
5º	5	6	**14**	**27**	**38**	**46**	**59**	**55**
			14	**27**	**38**	**46**	**55**	59

Note que, em cada fase, um valor apropriado para k é escolhido e os itens a_i e a_k são permutados. Particularmente na 4º fase, como os valores de i e k são iguais, a permutação não seria necessária. ☑

O que ainda não está muito claro é como o valor k é escolhido em cada fase. Para isso, vamos codificar a função *selmin*(v, i, n), que devolve a posição de um item mínimo dentro da seqüência $\langle v_i, v_{i+1}, ..., v_n \rangle$. A estratégia adotada supõe que o item v_i é o mínimo e então o compara com v_{i+1}, v_{i+2}, ... até que não haja mais itens ou então que um item menor que v_k seja encontrado. Nesse caso, v_k passa a ser o mínimo e o processo continua analogamente.

Exemplo 1.5. Selecionando um item mínimo numa seqüência.

k	***j***	v_1	v_2	v_3	v_4	v_5	v_6
1	1	**46**	**55**	59	14	38	27
1	2	**46**	55	**59**	14	38	27
1	3	**46**	55	59	**14**	38	27
4	4	46	55	59	**14**	**38**	27
4	5	46	55	59	**14**	38	**27**
4	–	46	55	59	**14**	38	27

Inicialmente, o item v_1 é assumido como o mínimo (k=1). Então v_1 é comparado aos demais itens da seqüência até que v_4 é encontrado. Como $v_4 < v_1$, o item v_4 passa a ser o mínimo (k=4) e o processo continua analogamente. ☑

Exemplo 1.6. Selecionando um item mínimo.

```
function selmin(var v:vetor; i:integer) : integer;
var j, k : integer;
begin
   k:=i;
   for j:=i+1 to max do
      if v[k]>v[j] then k:=j;
   selmin := k;
end;
```
☑

Exemplo 1.7. Ordenação por seleção: *selection sort*.

```
procedure ssort(var v:vetor);
var i, k, x : integer;
begin
   for i:=1 to max-1 do
      begin
         k := selmin(v,i);
         x := v[i];
         v[i] := v[k];
         v[k] := x;
      end;
end;
```
☑

Análise da ordenação por seleção

Analisando a rotina *ssort()*, constatamos que ela realiza o mesmo número de comparações que a rotina *bsort()*, ou seja, $(n-1)+(n-2)+\ \ldots+2+1 = \frac{1}{2}(n^2-n)$. Logo, a sua complexidade de tempo também é $O(n^2)$. Entretanto, como a subseqüência a ordenar diminui de um item a cada troca feita, temos que o número máximo de trocas[3] na *ssort()* é $n-1$; um número consideravelmente menor do que aquele encontrado para a rotina *bsort()*. Podemos dizer que o número de trocas na *bsort()* é $O(n^2)$, enquanto na *ssort()* é $O(n)$.

Exercício 1.3. Simule a execução da rotina *ssort()*, conforme no exemplo 1.4, para ordenar o vetor $L=\langle 82, 50, 71, 63, 85, 43, 39, 97, 14\rangle$.

Exercício 1.4. Codifique a rotina *ssort()* sem usar a função *selmin()*, ou seja, embutindo a lógica dessa função diretamente no código da *ssort()*.

Exercício 1.5. Adapte a rotina *ssort()* para ordenar um vetor de *strings*.

1.3. ORDENAÇÃO POR INSERÇÃO

Ao ordenar uma seqüência $\mathcal{V}=\langle a_1, a_2, ..., a_n\rangle$, esse método considera uma subseqüência ordenada $\mathcal{V}_o=\langle a_1, ..., a_{i-1}\rangle$ e outra desordenada $\mathcal{V}_d=\langle a_i, ..., a_n\rangle$. Então, em cada fase, um item é removido de $\mathcal{V}_d$ e inserido em sua posição correta dentro de $\mathcal{V}_o$. À medida em que o processo se desenvolve, a subseqüência desordenada vai diminuindo, enquanto a ordenada vai aumentando.

[3] *As trocas de índices realizadas em selmin() não estão sendo consideradas.*

Exemplo 1.8. Ordenação de um vetor v usando inserção.

i	v_1	v_2	v_3	v_4	v_5	v_6
1	**34**	**17**	68	29	50	47
2	**17**	**34**	**68**	29	50	47
3	**17**	**34**	**68**	**29**	50	47
4	**17**	**29**	**34**	**68**	**50**	47
5	**17**	**29**	**34**	**50**	**68**	**47**
	17	**29**	**34**	**47**	**50**	**68**

Inicialmente, a parte ordenada[4] é ⟨34⟩ e a desordenada é ⟨17, 68, 29, 50, 47⟩. Então, o item 17 é removido da parte desordenada e a posição liberada fica disponível no final da parte ordenada. Em seguida, o item 17 é comparado ao 34 que, sendo maior, é deslocado para a direita. Como não há mais itens, o 17 é inserido na posição liberada pelo item 34. Ao final dessa fase, a parte ordenada tem um item a mais e a desordenada, um item a menos... ☑

Exemplo 1.9. Ordenação por inserção: *insertion sort*.

```
procedure isort(var v:vetor);
var i, j, x : integer;
begin
   for i:=2 to max do
      begin
         x := v[i];
         j:=i-1;
         while (j>=1) and (x<v[j]) do
           begin
              j := j-1;
              v[j+1] := v[j];
           end;
         v[j+1] := x;
      end;
end;
```
☑

Análise da ordenação por inserção

A análise da rotina *isort()* mostra que o número de comparações realizadas é no máximo $\frac{1}{2}(n^2-n)$ e que, portanto, sua complexidade de tempo é $O(n^2)$. Entretanto, ao contrário do que acontece com os métodos de trocas e de sele-

[4] *Evidentemente, qualquer seqüência contendo um único item está trivialmente ordenada.*

ção, para as quais esse número é fixo, no método da inserção o número de comparações varia de $n-1$, quando a seqüência é crescente, a $\frac{1}{2}(n^2-n)$, quando a seqüência é decrescente. Isso quer dizer que, no caso médio, o método da inserção pode ser um pouco mais eficiente que os outros dois.

Exercício 1.6. Simule a execução da rotina *isort()*, preenchendo uma tabela como aquela apresentada no exemplo 1.8, para ordenar a seqüência $\langle 82, 50, 71, 63, 85, 43, 39, 97, 14\rangle$.

2. MÉTODOS DE BUSCA

A *busca* é o processo em que se determina se um particular elemento x é membro de uma determinado vetor $\mathcal{V}$. Dizemos que a busca tem *sucesso* se $x \in \mathcal{V}$ e que *fracassa* em caso contrário.

2.1. BUSCA LINEAR

A forma mais simples de se consultar um vetor em busca de um item particular é, a partir do seu início, ir examinando cada um de seus itens até que o item desejado seja encontrado ou então que seu final seja atingido.

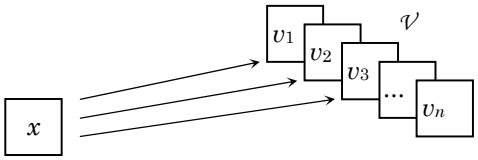

Figura 1 – *O método de busca linear*

Como os itens do vetor são examinados linearmente, em seqüência, esse método é denominado *busca linear* ou *busca seqüencial*. Para exemplificar seu funcionamento, vamos implementar uma função que determina se um certo número x consta de um vetor de números inteiros $\mathcal{V} = \langle v_1, v_2, v_3, ..., v_n\rangle$.

$$pertence(x,\mathcal{V}) = \begin{cases} verdade\ se\ x \in \mathcal{V} \\ falso\ se\ x \notin \mathcal{V} \end{cases}$$

Exemplo 2.1. Busca linear (*linear search*) num vetor de inteiros.

```
function pertence(x:integer; var v:vetor) : boolean;
var i : integer;
begin
   for i:=1 to max do
      if x=v[i] then
         begin
           pertence := true;
           exit;
         end;
   pertence := false;
end;
```

☑

Análise da busca linear

A vantagem da busca linear é que ela sempre funciona, independentemente do vetor estar ou não ordenado. A desvantagem é que ela é geralmente muito lenta; pois, para encontrar um determinado item x, a busca linear precisa examinar todos os itens que precedem x no vetor. Para se ter uma idéia, se cada item do vetor fosse procurado exatamente uma vez, o total de comparações realizadas seria $1+2+3+...+n=\frac{1}{2}(n^2+n)$. Isso significa que, para encontrar um item, a busca linear realiza em média $\frac{1}{2}(n+1)$ comparações, ou seja, examina aproximadamente a metade dos elementos armazenados no vetor.

No pior caso, quando o item procurado não consta do vetor, a busca linear precisa examinar todos os elementos armazenados no vetor para chegar a essa conclusão. Dizemos então que o tempo gasto por esse algoritmo é da ordem de n, isto é, $O(n)$. Isso significa, por exemplo, que se o tamanho do vetor é dobrado a busca linear fica aproximadamente duas vezes mais lenta.

Exercício 2.1. Codifique a função definida a seguir:

$$posição(x,V) = \begin{cases} i & se\ \exists\ i\ tal\ que\ x = V[i] \\ -1 & caso\ contrário \end{cases}$$

Exercício 2.2. Dados a lista de convidados de uma festa e o nome de uma pessoa, determinar se essa pessoa é ou não convidada da festa. Codifique um programa completo para resolver esse problema. Crie um procedimento para fazer a entrada da lista de convidados e adapte a função *pertence()*, definida anteriormente, para verificar se o nome consta ou não da lista.

2.2. BUSCA BINÁRIA

Se não sabemos nada a respeito da ordem em que os itens aparecem no vetor, o melhor que podemos fazer é uma busca linear. Entretanto, se os itens aparecem ordenados[5], podemos usar um método de busca muito mais eficiente. Esse método é semelhante àquele que usamos quando procuramos uma palavra num dicionário: primeiro abrimos o dicionário numa página aproximadamente no meio; se tivermos sorte de encontrar a palavra nessa página, ótimo; senão, verificamos se a palavra procurada ocorre antes ou depois da página em que abrimos e então continuamos, mais ou menos do mesmo jeito, procurando a palavra na primeira ou na segunda metade do dicionário...

Como a cada comparação realizada o espaço de busca reduz-se aproximadamente à metade, esse método é denominado *busca binária*.

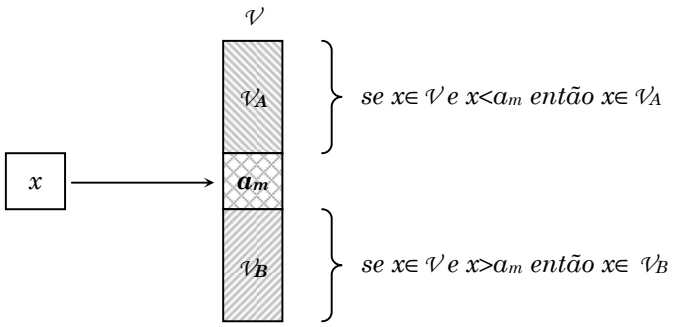

Figura 2 – *O método de busca binária*

Seja $\mathcal{V} = \mathcal{V}_A \circ \langle a_m \rangle \circ \mathcal{V}_B$ um vetor[6] tal que a_m esteja armazenado aproximadamente no meio dele. Então temos três possibilidades:

- $x = a_m$: nesse caso, o problema está resolvido;
- $x < a_m$: então x deverá ser procurado na primeira metade; e
- $x > a_m$: então x deverá ser procurado na segunda metade.

Caso a busca tenha que continuar, podemos proceder exatamente da mesma maneira: verificamos o item existente no meio da metade escolhida e se ele ainda não for aquele que procuramos, continuamos procurando no meio do quarto escolhido, depois no meio do oitavo e assim por diante até que o item procurado seja encontrado ou que não haja mais itens a examinar.

---

[5] *Quando nada for dito em contrário, 'ordenado' quer dizer 'ordenado de forma ascendente'*

[6] *O operador* $\circ$ *indica concatenação de seqüências.*

Exemplo 2.2. Vamos simular o funcionamento do algoritmo de busca binária para determinar se o item $x=63$ pertence ao vetor $V=\langle 14, 27, 39, 46, 55, 63, 71, 80, 92\rangle$. Para indicar o intervalo em que será feita a busca, usaremos dois índices, i e f, e para indicar a posição central desse intervalo, o índice m.

Passo	x	i	f	m	$V[1]$	$V[2]$	$V[3]$	$V[4]$	$V[5]$	$V[6]$	$V[7]$	$V[8]$	$V[9]$
1º	63	1	9	5	14	27	39	46	**55**	63	71	80	92
2º	63	6	9	7						63	**71**	80	92
3º	63	6	6	6						**63**			

No primeiro passo o intervalo de busca compreende todo o vetor, desde a posição 1 até a posição 9, e o item central ocupa a posição 5. Como $x>V[5]$, a busca deve continuar na segunda metade do vetor. Para indicar isso, atualizamos o índice i com o valor 6 e repetimos o procedimento. No segundo passo, o intervalo de busca foi reduzido aos itens que ocupam as posições de 6 a 9 e o meio[7] do vetor é a posição 7. Agora $x<V[7]$ e, portanto, a busca deve continuar na primeira metade; então o índice f é atualizado com o valor 6. No terceiro passo, resta um único item no intervalo de busca e o meio do vetor é a posição 6. Finalmente, temos $x=V[6]$ e a busca termina com sucesso. ☑

Como em cada passo o intervalo de busca reduz-se aproximadamente à metade, é evidente que o processo de busca binária sempre termina, mesmo que o item procurado não conste do vetor. Nesse caso, porém, o processo somente termina quando não há mais itens a examinar, ou seja, quando o intervalo de busca fica vazio. Como o início e o final do intervalo são representados pelos índices i e f, o intervalo estará vazio se e somente se tivermos $i>f$.

Exemplo 2.3. Seja $x=40$ e $L=\langle 14, 27, 39, 46, 55, 63, 71, 80, 92\rangle$. A tabela a seguir mostra o que acontece quando o item procurado não consta do vetor.

Passo	x	i	f	m	$L[1]$	$L[2]$	$L[3]$	$L[4]$	$L[5]$	$L[6]$	$L[7]$	$L[8]$	$L[9]$
1º	40	1	9	5	14	27	39	46	**55**	63	71	80	92
2º	40	1	4	1	14	**27**	39	46					
3º	40	3	4	3			**39**	46					
4º	40	4	4	4				46					
5º	40	**4**	**3**	?									

Note que, durante a busca, o valor que indica o início do intervalo fica maior que aquele que indica o seu final (veja 5º passo na tabela acima). Isso significa que não há mais itens a considerar e que, portanto, o item procurado não consta do vetor. Nesse caso, a busca deve terminar com fracasso. ☑

[7] *O meio do vetor é dado pelo quociente inteiro da divisão* $(i+f)/2$*, isto é, a média é truncada.*

Exemplo 2.4. Busca binária (*binary search*) num vetor de inteiros.

```
fiunction bb(x:integer; var L:vetor) : boolean;
var i, f, m : integer;
begin
   i := 1;
   f := max;
   while i<=f do
      begin
         m := (i+f) div 2;
         if x = L[m] then
            begin
               bb := true;
               exit;
            end;
         if  x<L[m] then f := m-1
         else            i := m+1;
      end;
   bb := false;
end;
```

☑

Análise da busca binária

Ao contrário da busca linear, a busca binária somente funciona corretamente se o vetor estiver ordenado. Isso pode ser uma desvantagem. Entretanto, à medida em que o tamanho do vetor aumenta, o número de comparações feitas pelo algoritmo de busca binária tende a ser muito menor que aquele feito pela busca linear. Então, se o vetor é muito grande, e a busca é uma operação muito requisitada, esse aumento de eficiência pode compensar o fato de termos que ordenar o vetor antes de usar a pesquisa binária.

Exemplo 2.5. Quantos itens são examinados pelo algoritmo de busca binária faz, no máximo, para encontrar um item num vetor com 5000 itens?

Seja n o número de itens no vetor. Se n é ímpar, o item do meio divide o vetor em duas partes iguais de tamanho $(n-1)/2$. Se n é par, o vetor é dividido em uma parte com $n/2-1$ itens e outra com $n/2$ itens[8]. Sendo assim, à medida em que o algoritmo executa, o número de itens vai reduzindo do seguinte modo:

$5000 \Rightarrow 2500 \Rightarrow 1250 \Rightarrow 625 \Rightarrow 312 \Rightarrow 156 \Rightarrow 78 \Rightarrow 39 \Rightarrow 19 \Rightarrow 9 \Rightarrow 4 \Rightarrow 2 \Rightarrow 1 \Rightarrow 0$

Na seqüência acima, cada redução implica numa comparação. Logo, são realizadas no máximo 13 comparações. ☑

[8] *Como queremos o número máximo de comparações, vamos considerar sempre a parte maior.*

Seja $C(n)$ o número máximo de comparações realizadas pelo algoritmo de busca binária num vetor de n elementos. Claramente, $C(1) = 1$. Suponha $n>1$. Como a cada comparação realizada o número de elementos cai para a metade, temos que $C(n) = 1 + C(n/2)$. Iterando essa recorrência temos:

$$
\begin{aligned}
C(n) &= 1 + C(n/2) \\
&= 1 + 1 + C(n/4) \\
&= 1 + 1 + 1 + C(n/8) \\
&\ldots \\
&= k + C(n/2^k)
\end{aligned}
$$

Supondo $n=2^k$, para algum inteiro k, temos que $C(n/2^k) = 1$ e $k = lg\, n$ e, portanto, $C(n) \approx lg\, n + 1$. Mais precisamente, temos que[9] $C(n) = \lfloor lg\, n \rfloor + 1$. Assim, podemos dizer que o algoritmo de busca binária tem complexidade $O(lg\, n)$.

Temos a seguir a relação entre o tamanho n do vetor e número de itens examinados pelos algoritmos de busca linear $C_L(n)$ e binária $C_B(n)$.

n	$C_L(n)$	$C_B(n)$
1	1	1
2	2	2
4	4	3
8	8	4
16	16	5
32	32	6
64	64	7
128	128	8
256	256	9
1024	1024	10
2048	2048	11
4096	4096	12
8192	8192	13
16384	16384	14
32768	32768	15
65536	65536	16
131072	131072	17
262144	262144	18
524288	524288	19
1048576	1048576	20

Figura 3 – *Comparações na busca linear versus na binária*

[9] *Essa fórmula, em que a função $\lfloor x \rfloor$ denota o maior inteiro menor ou igual a x, pode ser provada por indução finita.*

Exercício 2.3. Simule o funcionamento do algoritmo de busca binária para determinar se os itens 33, 50, 77, 90 e 99 constam do vetor $L = \langle 10, 16, 27, 31, 33, 37, 41, 49, 53, 57, 68, 69, 72, 77, 84, 89, 95, 99 \rangle$.

Exercício 2.4. Faça as alterações necessárias para que o algoritmo de busca binária funcione com vetores ordenados de forma decrescente.