

Algumas ferramentas do sistema Linux



O sistema GNU/Linux tem uma rica e poderosa caixa de ferramentas. Trata-se de uma coleção de programas (muitos escritos em C) que executam tarefas corriqueiras. Alguns desses programas são implementações de algoritmos e estruturas de dados discutidos [neste sítio](#).

astyle

Faz indentação correta do [arquivo-fonte](#) de um programa C. Veja o sítio [astyle](#).

bc

Basic calculator: arbitrary precision calculator language. Faz cálculos com números inteiros arbitrariamente grandes. Veja documentação na [Wikipedia](#), ou em [man7.org](#).

```
bc
```

chardet

Universal character encoding detector: procura determinar, heuristicamente, o esquema de codificação (= *encoding*) de um arquivo.

```
chardet nomes-de-arquivos
```

cmp

Semelhante ao diff. Veja documentação na [Wikipedia](#), ou em [man7.org](#).

```
cmp file1 file2
```

diff

The UNIX file-difference utility. Compara as linhas de dois [arquivos de texto](#). Mostra as linhas do primeiro que não estão no segundo e vice-versa. Veja documentação na [Wikipedia](#), ou em [man7.org](#).

```
diff -b -B file1 file2
```

enscript

Converte [arquivos de texto](#) em arquivos no formato HTML, no formato PostScript, ou outros. Exemplo:

```
enscript --highlight=c --color --language=html \
--output=isort.html isort.c
```

converte o arquivo [isort.c](#) no arquivo [isort.html](#). Veja documentação no [sítio oficial](#), ou na [Wikipedia](#).

file

Tenta determinar o tipo de um arquivo (texto, executável, binário, etc.) e qual o esquema de codificação (= *encoding*) do arquivo (ASCII, ISO-LATIN-1, UTF-8, etc.). O comando

```
file -i *
```

faz um relatório sobre o conteúdo do diretório corrente. Veja manual no [man7.org](#).

fmt

Um formatador de texto simples. Veja documentação na [Wikipedia](#), ou em [man7.org](#).

```
fmt -w65 <nome_do_arquivo>
```

gcc

GNU Compiler Collection: [compilador de programas em C](#). Veja [manual \(versão 6.3\) no gnu.org](#) ou [man7.org](#). Veja também o verbete [GNU Compiler Collection](#) na Wikipedia.

```
gcc -std=c99 -Wall arquivo1.c arquivo2.c
```

Veja [tutorial](#) no Cprogramming.com ou esse outro [tutorial](#).

gdb

[GNU debugger](#) (depurador de programas = caçador de erros de programas). Veja documentação no [sítio oficial](#), ou na [Wikipedia](#), ou em [man7.org](#). Veja também a folhinha [GDB Quick Reference](#).

```
gdb
```

Para aprender a usar o GDB leia os artigos [Debugging with GDB](#) e [Debugging Segmentation Faults and Pointer Problems](#) no sítio Cprogramming.com. Veja também o [tutorial](#) da disciplina CSE141 na Universidade da Califórnia.

(De acordo com Andrew Singer, “The art of debugging is figuring out what you really told your program to do rather than what you thought you told it to do.”)

gprof

GNU profiler: dá estatísticas sobre a execução de um programa (quantas vezes cada função foi chamada, quanto tempo cada função consumiu, etc.). Veja documentação na [Wikipedia](#), ou em [man7.org](#).

```
gprof
```

grep

Procura um padrão (*pattern*) em todos os arquivos de um diretório. Veja documentação na [Wikipedia](#), ou em [man7.org](#).

```
grep -r -s -i -I -e 'pattern' ./
```

hexdump

Filtro que exibe o conteúdo de arquivo no formato especificado pelo usuário. Veja também [od](#).

iconv

(O nome é uma abreviatura de *international conversion*). Muda o [esquema de codificação](#) de um [arquivo de texto](#) (por exemplo, de ISO-LATIN-1 para UTF-8).

```
iconv -f iso-8859-1 -t utf-8 in.txt -o out.txt
```

(A opção `-f` significa *from* e a opção `-t` significa *to*.) Veja documentação no [sítio oficial](#), ou na [Wikipedia](#), ou em [man7.org](#).

ispell

Verificador ortográfico (spelling checker). Veja documentação na [Wikipedia](#).

```
ispell -t files
```

less

Exibe o conteúdo de um arquivo no terminal. Veja documentação na [Wikipedia](#), ou em [man7.org](#).

```
less arquivo
```

make

Automatiza o processo de compilação de um programa. Veja a seção [Make e Makefile](#) no capítulo *Como organizar e compilar um programa*. Veja documentação na [Wikipedia](#), ou em [man7.org](#).

```
make
```

Veja também o verbete [Makefile](#) na Wikipedia.

od

Octal dump: exibe todos os bytes de um arquivo, digamos `xxx`. Diga

```
od -t u1 -A d xxx
```

para exibir todos os bytes em notação decimal. Diga

```
od -t o1 -A d xxx
```

para exibir os bytes em notação [octal](#). Diga

```
od -t c -A d xxx
```

para exibir os caracteres ASCII representados por cada byte. Veja documentação na [Wikipedia](#), ou em [man7.org](#). Veja também [hexdump](#).

pr

Prepara [arquivos de texto](#) para impressão. Veja documentação na [Wikipedia](#), ou em [man7.org](#)

```
pr -3 -t
```

sort

Ordena [lexicograficamente](#) as linhas de um [arquivo de texto](#). O funcionamento é afetado pela [variável de ambiente](#) LC_COLLATE. Supondo que o arquivo de texto esteja em português e use [codificação UTF-8](#), convém adotar LC_COLLATE="pt_BR.UTF-8". Veja documentação na [Wikipedia](#) ou em [man7.org](#).

```
LC_COLLATE=pt_BR.UTF-8 sort nome_do_arquivo
```

Para ordenar lexicograficamente byte-a-byte (ignorando os caracteres que alguns blocos de bytes podem eventualmente representar), diga

```
LC_COLLATE=C sort nome_do_arquivo
```

valgrind

Defeitos na administração da memória, caça a segmentation faults, [vazamentos de memória](#), e estatísticas de execução. Veja documentação no [sítio da ferramenta](#), ou na [Wikipedia](#), ou em [man7.org](#).

```
valgrind --leak-check=yes xxxx
```

Veja o artigo [Using Valgrind to Find Memory Leaks and Invalid Memory Use](#) no sítio Cprogramming.com.

wc

Word count: conta o número de linhas, de palavras e de bytes de um [arquivo ASCII](#). Se o seu arquivo é xxx.txt, basta dizer

```
wc xxx.txt
```

Veja a versão do [programa wc](#) escrito por Donald Knuth e Silvio Levi. Eis o [documento original](#), antes de sua conversão automática em código C. Isso foi escrito por Knuth e Levi para ilustrar o [sistema CWEB](#) de “programação letreada”.

Veja documentação na [Wikipedia](#), ou em [man7.org](#).

Atualizado em 2018-07-20

<https://www.ime.usp.br/~pf/algoritmos/>

Paulo Feofiloff

[DCC-IME-USP](#)



Leiaute

*"Let us change our traditional attitude
to the construction of programs.
Instead of imagining that our main task is
to instruct a computer what to do,
let us imagine that our main task is to explain to human beings
what we want a computer to do."*

— Donald E. Knuth, *Literate Programming*

*"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand."*

— Martin Fowler,
Refactoring: Improving the Design of Existing Code

Programas precisam ser entendidos não só por computadores, mas também (e principalmente) por seres humanos. Por isso, o leiaute (= *layout*) do [código-fonte](#) de um programa, ou seja, a disposição do código na folha de papel e na tela do terminal, é tão importante. Os dois elementos principais do leiaute são

- a *indentação*, que é o recuo das linhas em relação à margem esquerda da página, e
- os *espaços* entre as palavras (e demais símbolos) de uma linha.

O leiaute de um programa deve seguir os mesmos princípios que o leiaute de qualquer outro tipo de texto. É fácil habituar-se a produzir um bom leiaute. Com um pouco de prática, os dedos do programador, dançando sobre o teclado, passarão a fazer a coisa certa de maneira autônoma, deixando a mente do programador livre para cuidar de assuntos mais importantes.

Um bom leiaute

Eis uma amostra do melhor leiaute que conheço. Ele está de acordo com as recomendações de muitos gurus e programadores experientes:

```
int func (int n, int v[])
{
    int i = 0;
    while (i < n) {
        if (v[i] != 0)  i = i + 1;
        else {
            for (int j = i + 1; j < n; ++j)
                v[j-1] = v[j];
            n = n - 1;
        }
    }
    return n;
}
```

Veja como é fácil conferir o casamento de “{” com “}”. Também é fácil remover linhas e inserir novas linhas com o auxílio de um [editor de texto](#), quando isso for necessário.

Tenha compaixão por seus leitores e *não use fonte de espaçamento variável* (a [fonte](#) de texto normal) para escrever programas. O resultado é ruim pois muitos caracteres (como “i”, “-”, espaço, etc.) são estreitos demais:

```
int func (int n, int v[])
```

```

{
    int i = 0;
    while (i < n) {
        if (v[i] != 0) i = i + 1;
        else {
            for (int j = i + 1; j < n; ++j)
                v[j-1] = v[j];
            n = n - 1;
        }
    }
    return n;
}

```

Prefira sempre uma fonte monoespaçada (= *monospace*), como fazem todos os exemplos de código deste sítio.

Um leiaute compacto

O leiaute abaixo ocupa menos espaço vertical que o anterior. Ele é bom para escrever programas com lápis e papel (mas não tão bom para fazer alterações com o auxílio de um editor de texto). Indentação correta é essencial nesse leiaute.

```

int func (int n, int v[]) {
    int i = 0;
    while (i < n) {
        if (v[i] != 0) i = i + 1;
        else {
            for (int j = i + 1; j < n; ++j)
                v[j-1] = v[j];
            n = n - 1; }
    }
    return n; }

```

Mau exemplo

O que você acha do leiaute no seguinte exemplo?

```

int func ( int n,int v[] ){
    int i=0;
    while(i < n){
        if (v[i] !=0) i=i +1;
        else
        {
            for(int j=i+1;j<n;++j )
                v[j-1]=v[j];
            n =n- 1;
        }
    }
    return n;
}

```

Este leiaute é *péssimo* porque deixa espaços onde não deve, suprime os espaços onde eles são necessários, e é *inconsistente* (ou seja, escreve as coisas uma vez de um jeito e outra vez de outro jeito). Lembre-se: os espaços no código são tão importantes quanto as pausas na música!

Infelizmente, [alguns bons programadores e autores](#) cometem esse tipo de atrocidade gratuita. Espero que você não queira imitar esses maus exemplos.

Sugestões sobre leiaute

Quer uma sugestão? Use as mesmas regras de leiaute que qualquer texto (conto, artigo, reportagem, etc.) num livro, revista, jornal ou blog:

- use um espaço para separar cada palavra da palavra seguinte (os símbolos `=`, `<=`, `while`, `if`, `for` etc. contam como palavras);
- deixe um espaço depois, mas não antes, de cada sinal de pontuação;
- deixe um espaço depois, mas não antes, de parêntese e colchete direito;
- deixe um espaço antes, mas não depois, de parêntese e colchete esquerdo.

A expressão “`while(j < n)`” tem o mesmo sabor desagradável de “enquantoj for menor que n”. Portanto,

- jamais escreva `while(j < n)` no lugar de `while (j < n) ;`
- jamais escreva `else{` no lugar de `else {`
- não escreva `for (i=1;i<n;++i)` no lugar de `for (i = 1; i < n; ++i)`;
- etc.

Há três exceções notórias às regras acima: escreva

- `x->prox` e não `x -> prox`,
- `x[i]` e não `x [i]`,
- `x++` e não `x ++`.

Além disso, é usual não deixar espaços antes nem depois dos operadores de multiplicação e divisão. Assim, é usual escrever `x*y` e `x/y` em lugar de `x * y` e `x / y` respectivamente.

Sugiro deixar um espaço entre o nome de uma função e o parêntese esquerdo seguinte, ainda que isso contrarie a notação tradicional da matemática. Por exemplo, sugiro escrever

funcao (arg1, arg2)

em vez de `funcao(arg1, arg2)`, pois o primeiro leiaute é mais legível que o segundo.

Exercícios

1. Qual das duas linhas abaixo tem leiaute mais civilizado?

para j variando de 1 até n de 1 em 1, faça
paraj variando del até n de lem1,faça

2. Qual das duas linhas abaixo tem leiaute mais decente?

`for(j=0;j<n;++j){`
`for (j = 0; j < n; ++j) {`

3. Qual das duas linhas abaixo tem melhor leiaute?

`for (j = 0; j < n; ++j) {`
`for(j = 0; j < n; ++j) {`

4. Qual das duas linhas abaixo tem leiaute mais civilizado?

`for (j = 0;j < n; ++j){`
`for (j = 0; j < n; ++j) {`

5. Reescreva o trecho de programa abaixo com bom leiaute.

```
int func(int n,int v[]){int i,j;i=0;while(i<n){  
if(v[i]!=0) ++i;else{for(j=i+1;j<n;++j)  
v[j-1]=v[j];--n;}}return n;}
```

6. Reescreva a seguinte linha de código com leiaute decente. Depois, escreva uma versão mais simples da linha.

```
for(i = 23; i --> 0;)
```

7. Corrija os erros de leiaute no texto abaixo.

A computação não é a ciência dos computadores, da mesma forma que a astronomia não é a ciência dos telescópios .

8. Corrija os erros de leiaute do texto abaixo.

Em 1959 e nas décadas seguintes nenhum programador Cobol poderia imaginar que os programas de computador que estava criando ainda estariam em operação no fim do século. Poucos se lembram hoje que há menos de quinze anos os primeiros PCs possuíam apenas 64Kbytes de memória. Como o custo dos dispositivos de armazenamento era alto e a quantidade de memória disponível era pequena, usavam-se muitos truques para economizar esse recurso. Um deles foi o uso de dois dígitos para representar o ano: armazenava-se (por exemplo) 85 em vez de 1985. Com a chegada do ano 2000, essa codificação "econômica" transformou-se em um erro em potencial .

SOLUÇÃO:

Em 1959 e nas décadas seguintes nenhum programador Cobol poderia imaginar que os programas de computador que estava criando ainda estariam em operação no fim do século. Poucos se lembram hoje que há menos de quinze anos os primeiros PCs possuíam apenas 64K bytes de memória. Como o custo dos dispositivos de armazenamento era alto e a quantidade de memória disponível era pequena, usavam-se muitos truques para economizar esse recurso. Um deles foi o uso de dois dígitos para representar o ano: armazenava-se (por exemplo) 85 em vez de 1985. Com a chegada do ano 2000, essa codificação "econômica" transformou-se em um erro em potencial.

9. Reescreva o trecho de programa abaixo usando leiaute decente.

```
esq= 0; dir=N-1;
i=(esq+dir)/2; /*indice do "meio"de r[]*/
while(esq <= dir && r[i] != x){
    if(r[i]<x) esq = i+1;
    else dir = i-1; /* novo indice do "meio"de r[] */
    i= (esq + dir)/2;
}
```

SOLUÇÃO:

```
esq = 0; dir = N - 1;
i = (esq + dir)/2; /* índice do "meio" de r[] */
while (esq <= dir && r[i] != x) {
    if (r[i] < x) esq = i + 1;
    else dir = i - 1; /* novo índice do "meio" de r[] */
    i = (esq + dir)/2;
}
```

Recomendações finais

- Não use linhas longas, nem no código, nem nos comentários. Linhas longas são difíceis de ler e em geral não cabem numa folha de papel impressa. Sugiro nunca passar de setenta-e-tantos caracteres por linha. (A propósito, veja [Eighty Column Rule](#).)
- Não use tabulação (tecla [tab](#)) no [código-fonte](#) do seu programa. Para fazer a indentação de uma linha, use o número apropriado de espaços. (A propósito, veja [Tabs Are Evil](#).)
- Se usar tabulação (apesar da recomendação anterior), use-a apenas e tão somente *no início* da linha. Seja sistemático: se uma linha começa com [tab](#) então *todas* as linhas devem começar com um ou mais [tabs](#) (seguidos de um caractere diferente de espaço).
- Alguns [editores de texto](#) trocam, automaticamente, longas sequências de espaços por

tabulação. A substituição é imperceptível na tela do terminal, mas tende a bagunçar o leiaute do programa impresso. Configure o seu editor de texto de modo a desligar essa substituição automática de espaços por tabs.

- Evite caracteres supérfluos (exceto espaços em branco) no código. Em particular, evite os parênteses que as [regras de precedência entre operadores](#) tornam redundantes.
- Não há nada melhor que um bom layout feito manualmente. Mas numa emergência você pode recorrer a algum aplicativo, como o [astyle](#) ou um [IDE \(Integrated Development Environment\)](#), para corrigir os defeitos de layout do seu programa.

Veja [“How important is formatting code?” \(by Steve Baker\)](#) no Quora.

Atualizado em 2018-03-27

<https://www.ime.usp.br/~pf/algoritmos/>

Paulo Feofiloff

[DCC-IME-USP](#)



Documentação

"Programming is best regarded as the process of creating works of literature, which are meant to be read."
— Donald E. Knuth, *Literate Programming*

"Computer science is no more about computers than astronomy is about telescopes."
— Edsger W. Dijkstra

*"Comments are, at best, a necessary evil.
The proper use of comments is to compensate
for our failure to express ourselves in code [...]
Truth can only be found in one place: the code."*
— Robert C. Martin, *Clean Code*

*"Use definite, specific, concrete language."
"Write with nouns and verbs."
"Put the emphatic words at the end."
"Omit needless words."*
— W. Strunk, Jr. & E.B. White, *The Elements of Style*

Há quem diga que *documentar* um programa é o mesmo que escrever muitos comentários de mistura com o código. Isso está errado! Uma boa documentação não suja o código com comentários. Uma boa documentação limita-se a

explicar *o que* cada função do programa faz.

Uma boa documentação não perde tempo tentando explicar *como* a função faz o que faz, porque o leitor interessado nessa questão pode ler o código.

A distinção entre *o que* e *como* é a mesma que existe entre a interface ([arquivo .h](#)) e a implementação (arquivo .c) de uma biblioteca C. A seguinte analogia pode tornar mais clara a diferença. Uma empresa de entregas promete apanhar o seu pacote em São Paulo e entregá-lo em Manaus. Isso é *o que* a empresa faz. *Como* o serviço será feito — se o transporte será terrestre, aéreo ou marítimo, por exemplo — é assunto interno da empresa.

Em suma, a documentação de uma função é um *minimanual* que dá instruções completas sobre o uso correto da função. (Nesse sentido, o conceito de documentação se confunde com o conceito de [API](#).) Esse minimanual deve dizer o que a função recebe e o que devolve. Em seguida, deve dizer, de maneira muito precisa, que efeitos a função produz, ou seja, qual a relação entre o que a função recebe e o que devolve.

Uma documentação correta é uma questão de honestidade intelectual, pois coloca nas mãos do leitor e do usuário a real possibilidade de *detectar os erros* que o autor possa ter cometido ao escrever o código.

Exemplo

Em [um dos capítulos deste sítio](#) há uma função que encontra o valor de um elemento máximo de um vetor. Vamos repetir aqui o código daquela função juntamente com uma documentação perfeita:

```
// A seguinte função recebe um número n >= 1
// e um vetor v e devolve o valor de um
```

```
// elemento máximo de v[0..n-1].  
  
int max (int n, int v[]) {  
    int x = v[0];  
    for (int j = 1; j < n; j += 1)  
        if (x < v[j])  
            x = v[j];  
    return x;  
}
```

Veja como a documentação é simples e *precisa*. A documentação diz *o que* a função faz mas não perde tempo tentando explicar *como* a função faz o que faz (por exemplo, se a função é recursiva ou iterativa, se percorre o vetor da esquerda para a direita ou vice-versa). Observe também que não há comentários inúteis (como “o índice *j* vai percorrer o vetor”) sujando o código da função.

Eis alguns exemplos de má documentação. Dizer apenas que a função

```
devolve o valor de um elemento máximo  
de um vetor
```

é *indecentemente vago*, pois nem sequer menciona os parâmetros (*n* e *v*) da função! Dizer que a função

```
devolve o valor de um elemento máximo  
do vetor v
```

é um pouquinho melhor, mas ainda *muito vago*: o leitor fica sem saber qual o papel do parâmetro *n*. Dizer que a função

```
devolve o valor de um elemento máximo  
de um vetor v que tem n elementos
```

é melhor, mas ainda está vago: não se sabe se o vetor é *v[0..n-1]* ou *v[1..n]*. Dizer que a função

```
devolve o valor de um elemento máximo  
de v[0..n-1]
```

já está quase bom, mas sonega a informação de que a função só faz sentido e funciona corretamente se *n* ≥ 1.

Outro exemplo

Em [um dos capítulos deste sítio](#) há uma função que decide se um número *x* é igual a algum elemento de um vetor *v*. Repetimos aqui o código daquela função juntamente com uma documentação perfeita:

```
// Recebe um número x, um vetor v,  
// e um índice n ≥ 0. Devolve 1 se  
// x está em v[0..n-1] e devolve 0  
// em caso contrário.  
  
int busca (int x, int n, int v[]) {  
    int j = 0;  
    while (j < n && v[j] != x)  
        j += 1;  
    if (j < n) return 1;  
    else return 0;  
}
```

Observe como a documentação diz, de maneira precisa e completa, *o que* a função faz. Veja como a documentação não perde tempo tentando explicar *como* a função faz o serviço.

Para contrastar, eis alguns exemplos de má documentação. Dizer

```
decide se x está em v[0..n-1]
```

é um pouco vago, pois o leitor precisa adivinhar o que a função devolve. Dizer

```
decide se x está no vetor v
```

é muito vago, pois não explica o papel do parâmetro n. Dizer

```
decide se um número está em um vetor
```

é absurdamente vago, pois nem sequer menciona os parâmetros da função!

Mais um exemplo

Eis uma função acompanhada de documentação perfeita:

```
// Recebe x, v e n >= 0 e devolve j
// tal que 0 <= j < n e v[j] == x.
// Se tal j não existe, devolve n.

int onde (int x, int v[], int n) {
    int j = 0;
    while (j < n && v[j] != x)
        j += 1;
    return j;
}
```

Exercícios 1

1. Considere a seguinte documentação de uma função: “Esta função recebe números inteiros p, q, r, s e devolve a média aritmética de p, q, r.” O que há de errado?
2. Considere a seguinte documentação de uma função: “Esta função recebe números inteiros p, q, r tais que p <= q <= r e devolve a média aritmética de p, q, r.” O que há de errado?

Observações

1. A documentação das funções de um programa tem o importante papel de separar as responsabilidades do programador das do usuário. Cabe ao programador dizer, na documentação, quais os valores válidos de cada parâmetro da função; cabe ao usuário a tarefa de verificar, antes de invocar a função, se os valores dos argumentos são válidos. Com isso, o programador *fica dispensado de verificar a validade dos argumentos* e pode dedicar toda a sua atenção à solução do problema que a função deve resolver.
2. Nos demais capítulos deste sítio, por conveniência tipográfica, a documentação de muitas funções não foi integrada ao código (na forma de *// comentário*) mas escrita no texto que precede o código.
3. Existem excelentes ferramentas para integrar código com documentação. Veja, por exemplo, o sistema [CWEB](#) de D.E. Knuth. Para ilustrar, preparei dois programas em CWEB: [mdp](#) e [isort](#).

Invariante de iterações

Há uma situação em que comentários misturados com código são úteis. O corpo de muitas funções consiste em um processo [iterativo](#) (controlado por um `for` ou um `while`). Nesses casos, depois de dizer *o que* a função faz, você pode enriquecer a documentação dizendo quais os [invariante](#)s do processo iterativo. Um *invariante* é uma *relação entre os valores das variáveis* que

vale no início de cada iteração

e não se altera de uma iteração para outra. Essas relações [invariante](#)s explicam o funcionamento do processo iterativo e permitem *provar*, por indução, que ele tem o efeito desejado.

EXEMPLO 1. A função `max` calcula o valor de um elemento máximo de $v[0..n-1]$. O comentário embutido no código dá o invariante do processo iterativo:

```
int max (int n, int v[]) {
    int x = v[0];
    for (int j = 1; j < n; ++j)
        // neste ponto, x é um
        // elemento máximo de v[0..j-1]
        if (x < v[j])
            x = v[j];
    return x;
}
```

EXEMPLO 2. Digamos que um segmento $v[i..j]$ de um vetor $v[0..n-1]$ é *constante* se todos os seus elementos têm o mesmo valor. A função `scmax` abaixo recebe um vetor $v[0..n-1]$, com $n > 0$, e devolve o comprimento de um segmento constante máximo. O comentário embutido no código dá o invariante do processo iterativo:

```
int scmax (int v[], int n) {
    int i = 0, max = 0;
    while /*A*/ i < n) {
        // no ponto A,
        // 1. max é o comprimento de um segmento
        //     constante máximo de v[0..i-1] e
        // 2. i == 0 ou v[i-1] != v[i] ou i == n
        int j = i+1;
        while (j < n && v[j] == v[i]) ++j;
        if (max < j-i) max = j-i;
        i = j;
    }
    return max;
}
```

[Carlos A. Estombelo-Montesco encontrou um erro na versão anterior do código.]

Invariante s são essenciais para entender *por que* uma função ou um [algoritmo](#) estão corretos. Vários exemplos de prova de correção de algoritmos baseada em invariantes aparecem nos capítulos dedicadas à [busca binária](#), à [ordenação](#), ao [mergesort](#), ao [heapsort](#), e ao [quicksort](#).

Exercícios 2

- Um programador inexperiente afirma que a seguinte proposição é um invariante da função `max` acima: “ x é o maior elemento da parte do vetor v vista até agora.” Critique essa afirmação.
- A seguinte documentação da função `scmax` está correta?

```
// a função recebe um vetor crescente
// v[0..n-1] com n >= 1 e devolve o
// comprimento de um segmento constante
// máximo do vetor
```

Como organizar, compilar e depurar um programa C

Este capítulo é um resumo de informações práticas sobre a organização, a compilação, a correção de defeitos (= *debugging*), e a execução um programa em linguagem C. As informações valem para o sistema operacional GNU/Linux. Algo semelhante vale nos sistemas MAC e Windows (mas *não use Windows*, por favor!)

Como organizar um programa C

Todo programa em linguagem C é um conjunto de funções, uma das quais é `main`. A execução do programa consiste na execução de `main`, que tipicamente invoca outras funções do conjunto. Em um programa bem organizado, a função `main` tem caráter apenas *gerencial*: ela cuida da entrada de dados, da saída de resultados, e da chamada de outras funções (que fazem o trabalho realmente importante).

O conjunto de funções que constitui um programa reside em um ou mais arquivos, conhecidos como *arquivos-fonte* (= *source files*). Cada arquivo-fonte é um *módulo* do programa. O nome de cada módulo termina com “`.c`”. O módulo *principal* é o que contém a função `main`. Nos exemplos a seguir, vamos supor que o programa reside em dois módulos,

`ppp.c` e `qqq.c`,

sendo `ppp.c` o módulo principal. Um bom programador tem a sensibilidade de espalhar as funções pelos módulos de uma maneira lógica, colocando num mesmo módulo as funções encarregadas de uma mesma parte do problema que o programa deve resolver.

Para cada módulo, exceto o principal, é importante escrever um *arquivo-interface* (= *header file*), contendo as `macros`, as declarações das eventuais `variáveis globais` e os protótipos das funções a que o módulo principal pode ter acesso. O nome da interface acompanha o nome do correspondente módulo: se o módulo é `qqq.c`, a interface é `qqq.h`. Assim, se `ppp.c` é o módulo principal, o programa completo consistirá nos arquivos

`ppp.c` , `qqq.c` e `qqq.h`.

Como compilar um programa C

Antes que o programa possa ser executado, ele deve ser *compilado*. A compilação transformará o conjunto de arquivos-fonte em um arquivo *executável*, também conhecido como *binário*.

Preparação

O primeiro passo é simples mas importante: abra um terminal, crie um diretório de trabalho, e vá para esse diretório:

```
~$ mkdir meu-diretorio-de-trabalho
```

```
~$ cd meu-diretorio-de-trabalho  
~$
```

Coloque todos os arquivos do seu programa no diretório de trabalho. Confira o conteúdo do diretório de trabalho:

```
~$ ls -1  
ppp.c  
qqq.c  
qqq.h  
~$
```

Compilação

Para compilar os arquivos do seu programa e transformá-los em um arquivo executável `xxx`, basta invocar o [compilador gcc](#). Por exemplo,

```
~$ gcc ppp.c qqq.c -o xxx  
~$
```

A primeira fase da compilação é um [pré-processamento](#), que cuida de todas as linhas de código que começam com “#”. A segunda fase, compila os arquivos pré-processados.

O compilador emite uma lista dos erros porventura presentes nos arquivos-fonte. Corrija os erros com auxílio de um [editor de texto](#) e compile o programa novamente.

Warnings de compilação

Além de detetar erros que impedem a compilação, o compilador gcc pode emitir advertências (= *warnings*) sobre imperfeições do seu programa. Para forçar o gcc a fazer isso, use a [opção -std=c99](#) (para escolher o padrão C99 da linguagem C) e a opção adicional [-Wall](#):

```
~$ gcc -std=c99 -Wall ppp.c qqq.c -o xxx
```

(Sugiro também usar as opções [-Wextra](#), [-Wno-unused-result](#), [-Wpedantic](#) e [-O0](#).) Se o programa usa a biblioteca `math`, ou seja, se tem um `#include <math.h>`, acrescente um `-lm` ao final da linha de comando.

Os warnings devem ser levados muito a sério. Corrija as causas das advertências e compile o programa novamente. (Felizmente, a longa lista de opções e nomes de arquivos não precisa ser redigitada: basta usar a tecla ↑.)

O ciclo compilar-corrigir-compilar deve ser repetido até que não haja mais *nenhum warning*. Só depois disso, o programa `xxx` estará pronto para ser executado.

Como executar um programa compilado

Agora que a compilação foi bem-sucedida, o programa contido no arquivo `xxx` pode ser executado:

```
~$ ./xxx
```

(O prefixo “`./`” é importante para indicar que se trata do arquivo `xxx` que está no seu diretório de trabalho e não de um eventual homônimo de `xxx` que está em algum outro diretório.)

Se a função `main` do seu programa tiver parâmetros, digite os correspondentes argumentos na [linha de comando](#). Por exemplo, se o seu programa tiver três argumentos, digite

```
~$ ./xxx arg1 arg2 arg3
```

Tipicamente, um ou mais desses argumentos são nomes de arquivos de dados que o programa xxx deve processar. Recomendo colocar todos esses arquivos de dados no seu diretório de trabalho antes de executar o programa.

Make e Makefile

Para automatizar um pouco a compilação do seu programa, reduzindo assim a digitação enfadonha de longas linhas de comando, use o utilitário [make](#). No caso do exemplo discutido acima, coloque um [arquivo Makefile muito simples](#) no seu diretório de trabalho e diga

```
~$ make xxx
```

para compilar o programa.

Depuração (debugging) e GDB

[The art of debugging is figuring out what you really told your program to do rather than what you thought you told it to do.](#)
— Andrew Singer

[Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.](#)
— Brian W. Kernighan

Finalmente, seu programa passou pelo compilador sem erros e sem warnings! Infelizmente isso não significa que o programa está livre de *bugs* (= defeitos = erros).

Rode o seu programa com dados de teste. As primeiras tentativas podem terminar abruptamente em um *crash*, como um *segmentation fault* (tentativa de acessar uma posição de memória que está fora dos limites alocados para a execução do seu programa), por exemplo. Para encontrar o bug que causou o crash, examine os arquivos-fonte do programa e os resultados da tentativa de execução. Use seu espírito de detetive. Se encontrar o bug, corrija-o e recompile o programa.

Se a tentativa de encontrar o bug manualmente não tiver sucesso, use o poderoso caça-bugs [GDB Debugger](#). (Antes, é preciso recompilar o programa com a opção -g do gcc.)

(Veja [Julia's drawings: How I got better at debugging](#).)

Vazamento de memória

Um bug (= defeito = erro) particularmente desagradável é o vazamento de memória (= *memory leak*): o programa funciona muito bem quando a quantidade de dados é pequena mas esgota a memória disponível e termina num crash quando a quantidade de dados é grande. (Isso pode ser constatado, por meio do *system monitor* ou *task manager*, observando o gráfico da quantidade de memória em uso.) O vazamento de memória acontece tipicamente quando o programador esquece de [desalocar](#) a memória alocada por [malloc](#).

A melhor maneira de descobrir onde está o vazamento é examinar os arquivos-fonte do programa com muita atenção. Se essa análise manual não for suficiente, você pode recorrer ao

Bytes, números e caracteres

```
01000010 01111001 01101000 01100101 01110011 00101100 00100000
01101110 01110101 01101101 01100101 01110010 01101111 01110011
00100000 01100101 00100000 01100011 01100001 01110010 01100001
01100011 01110100 01100101 01110010 01100101 01110011 00001010
```

A memória de qualquer computador é uma [sequência](#) de bytes. Cada *byte* é uma sequência de 8 bits (dígitos binários) e portanto tem $2^8 = 256$ possíveis valores:

```
00000000
00000001
00000010
:
11111110
11111111
```

Em geral, trabalhamos com blocos de s bytes consecutivos, podendo s valer 1, 2, 4 ou até 8. Cada bloco de s bytes tem $8s$ bits e portanto pode assumir 2^{8s} valores.

Um bloco de bytes pode ser interpretado de três maneiras diferentes:

- como um número natural (0, 1, 2, 3, ...),
- como um número inteiro (... , -2, -1, 0, +1, +2, ...) ou
- como um caractere (A, B, C, ... , a, b, c, ...).

Esta página discute essas três interpretações.

Números naturais e notação binária

Toda sequência de bits pode ser vista como um número natural em *notação binária*: o número natural é a soma das potências de 2 que correspondem aos bits 1. Por exemplo, a sequência 1101 representa o número $2^3 + 2^2 + 2^0$, igual a 13. A sequência 1111 representa $2^3 + 2^2 + 2^1 + 2^0$, igual a 15.

Portanto, toda sequência de s bytes — ou seja, $8s$ bits — representa um número natural no intervalo (fechado)

$$0 \dots 2^{8s}-1.$$

Se $s = 1$, por exemplo, o intervalo vai de 0 a 2^8-1 , isto é, de 0 a 255. Se $s = 2$, o intervalo vai até $2^{16}-1$, isto é, até 65535. Se $s = 4$, o intervalo vai até $2^{32}-1$, isto é, até 4294967295.

Exemplo. Para que o exemplo caiba na página, considere sequências de *apenas* 4 bits. Uma tal sequência representa, em [notação binária](#), um número no intervalo $0 \dots 2^4-1$:

bits	número
0000	0
0001	1
0010	2
0011	3
0100	4

0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Convém imaginar que a tabela é cíclica — ou seja, que o fim da tabela é emendado com o começo — e que as operações aritméticas entre números são executadas módulo 2^4 , seguindo o ciclo. Por exemplo, o resultado da adição de 1 a 15 é 0, e o resultado da adição de 7 a 10 é 1.

Exercícios 1

1. Mostre que todo número natural pode ser escrito em notação binária.
2. Mostre que $2^k + 2^{k-1} + \dots + 2^1 + 2^0 = 2^{k+1}-1$, qualquer que seja o número natural k .
3. Escreva os números 2^8 , 2^8-1 , 2^{16} , $2^{16}-1$, 2^{32} e $2^{32}-1$ em [notação hexadecimal](#).

Números inteiros e notação complemento-de-dois

Toda sequência de s bytes — ou seja, $8s$ bits — pode ser interpretada como um número inteiro no intervalo (fechado)

$$-2^{8s-1} \dots 2^{8s-1}-1.$$

Se $s = 1$, por exemplo, esse intervalo vai de -2^7 a 2^7-1 , isto é, de -128 a 127 . Se $s = 2$, o intervalo vai de -2^{15} a $2^{15}-1$, isto é, de -32768 a 32767 . Se $s = 4$, o intervalo vai de -2^{31} a $2^{31}-1$, isto é, de -2147483648 a 2147483647 .

Como determinar o inteiro que uma dada sequência de $8s$ bits representa? Comece por interpretar a sequência como um número em [notação binária](#). Digamos que esse número é n . Se a sequência começa com um bit 0, ela representa o inteiro positivo n . Se a sequência começa com um bit 1, ela representa o inteiro [estritamente negativo](#) $n - 2^{8s}$. Essa maneira de representar inteiros é conhecida como *notação complemento-de-dois* (= [two's complement](#)).

Exemplo. Para que o exemplo caiba na página, considere sequências de *apenas* 4 bits. Uma tal sequência representa um número inteiro no intervalo $-2^3 \dots 2^3-1$:

bits	inteiro
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	-8
1001	-7
1010	-6
1011	-5

1100	-4
1101	-3
1110	-2
1111	-1

Convém imaginar que a tabela é cíclica (o fim é emendado com o começo) e que as operações aritméticas entre os números são executadas módulo 2^4 , seguindo o ciclo. Por exemplo, o resultado da adição de 7 e 1 é -8. Analogamente, o resultado da adição de 5 e -14 é 7.

Exercícios 2

1. NOTAÇÃO COMPLEMENTO-DE-DOIS. Mostramos acima como a notação complemento-de-dois transforma em um inteiro negativo qualquer sequência de s bytes que começa com um bit 1. Agora considere a operação inversa. Dado um inteiro n no intervalo $-2^{8s-1} \dots -1$, mostre que a sequência de s bytes que representa n em notação complemento-de-dois, é a mesma sequência de bytes que representa $n + 2^{8s}$ em notação binária.
2. NOTAÇÃO COMPLEMENTO-DE-DOIS. Mostramos acima como a notação complemento-de-dois transforma em um inteiro negativo qualquer sequência de s bytes que começa com um bit 1. Agora considere a operação inversa. Dado um inteiro n no intervalo $-2^{8s-1} \dots -1$, mostre que basta tomar a sequência de bits que representa o valor absoluto de n em notação binária, depois inverter todos os bits (ou seja, trocar 0 por 1 e 1 por 0), e finalmente somar 1, em binário, ao resultado.
3. ALTERNATIVA PARA COMPLEMENTO-DE-DOIS. Suponha, como fizemos no exemplo acima, que dispomos de apenas 4 bits para representar números inteiros. Agora adote a seguinte maneira de interpretar uma sequência de 4 bits. Seja n o número inteiro positivo que os 3 últimos bits representam em notação binária. Se o primeiro bit é 0, então a sequência toda representa o número positivo n . Se o primeiro bit é 1, então a sequência toda representa o número negativo $-n$. (Por exemplo, a sequência 1101 representa -5.) Discuta as desvantagens dessa representação de números inteiros.
4. Escreva os números 2^7 , 2^7-1 , 2^{15} , $2^{15}-1$, 2^{31} e $2^{31}-1$ em [notação hexadecimal](#).

Caracteres e a tabela ASCII

Um [byte](#) pode ser interpretado como um caractere (letra, dígito, sinal de pontuação, etc.). Essa interpretação tem por base a [tabela ASCII](#), uma tabela arbitrária mas universalmente aceita que associa um caractere a cada byte. ([ASCII](#) é a sigla do American Standard Code for Information Interchange.) Por razões históricas, a tabela ASCII usa apenas bytes cujo primeiro bit é 0 e portanto, tem apenas 128 linhas. Eis uma pequena amostra da tabela:

byte	caractere
00111111	?
01000000	@
01000001	A
01000010	B
01000011	C
01100001	a
01100010	b
01100011	c
01111110	~

O conjunto de caracteres coberto pela tabela é conhecido como *alfabeto ASCII*. A parte principal desse alfabeto consiste nos seguintes caracteres:

```
! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9
: ; < = > ? @
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

```
[ \ ] ^ ~
a b c d e f g h i j k l m n o p q r s t u v w x y z
{ | } ~
```

(O primeiro caractere da lista é um espaço.) É cômodo usar atalhos verbais ao falar de caracteres ASCII. Por exemplo, em vez de dizer “o caractere A” podemos dizer “o caractere 65”, pois o byte que corresponde a A na tabela ASCII vale 65 em [notação binária](#).

Além dos noventa e cinco caracteres “normais”, o alfabeto ASCII contém trinta e três caracteres “especiais”, conhecidos como *caracteres de controle*. Esses caracteres não são símbolos tipográficos como os outros e por isso precisam ser indicados por uma notação especial: uma barra invertida seguida de um dígito ou letra. Eis os caracteres de controle mais comuns:

byte	caractere	
00000000	\0	<i>caractere nulo (null)</i>
00001001	\t	<i>tabulação horizontal (tab)</i>
00001010	\n	<i>fim de linha (newline)</i>
00001011	\v	<i>tabulação vertical</i>
00001100	\f	<i>fim de página (new page)</i>
00001101	\r	<i>carriage return</i>

O caractere \0 é usado para marcar o fim de uma [string](#) e não ocupa espaço algum ao ser exibido; o caractere \n marca o fim de uma linha de texto e produz uma mudança de linha ao ser exibido; o caractere \f marca o fim de uma página; e assim por diante. Embora o espaço não seja um caractere de controle, podemos usar a notação \ (barra invertida seguida de um espaço) para torná-lo mais visível.

Os caracteres \ , \t, \n, \v, \f e \r são conhecidos como *brancos* (= *white-spaces*). Muitas funções das [bibliotecas padrão](#) tratam todos os brancos como se fossem espaços.

O alfabeto ASCII não contém letras com [sinais diacríticos](#), como é, Á, õ, ç, etc. Cada uma dessas e muitas outras letras é representada por 2 ou mais bytes num esquema de codificação conhecido como UTF-8. Trataremos disso na página [Unicode e UTF-8](#).

Exercícios 3

- Quais são os bytes que representam os caracteres O, o, 0 e \0 ?
- Escreva os bytes 01000001, 01000010 e 01000011 em [notação hexadecimal](#).
- Escreva, em notação decimal, a sequência de bytes que representa o texto “Um byte tem 8 bits.”.
- Considere os bytes que representam os inteiros 65 67 72 79 85 33 10 51 50 32 43 32 52 51 32 61 32 55 53 em notação binária. Qual a sequência de caracteres representada pela sequência de bytes?
- Familiarize-se com os programas [od](#) e [hexdump](#) (os nomes são abreviaturas de *octal dump* e *hexadecimal dump*). Esses utilitários exibem, byte-a-byte, o conteúdo de qualquer arquivo dado.

Perguntas e respostas

- PERGUNTA: O alfabeto ASCII usa apenas bytes cujo primeiro bit é 0. Por que não usar os bytes cujo primeiro bit é 1 para representar letras com diacríticos, como á, é, ç, etc.?

RESPOSTA: A [tabela ISO-LATIN-1](#) faz exatamente isso! Infelizmente, ela caiu em desuso.

Os tipos int e char

$-2147483648 \dots 2147483647$

$-128 \dots 127$

Na linguagem C, os números naturais ($0, 1, 2, 3, \dots$) são conhecidos como “inteiros sem sinal”, enquanto os números inteiros ($\dots, -2, -1, 0, +1, +2, \dots$) são conhecidos como “inteiros com sinal”. Inteiros sem sinal são implementados pelos [tipos-de-dados](#) `unsigned char` e `unsigned int`. Inteiros com sinal são implementados pelos tipos-de-dados `char` e `int`. Para declarar uma variável `u` do primeiro tipo, uma variável `n` do segundo, uma `c` do terceiro e uma `i` do quarto, basta dizer

```
unsigned char u;
unsigned int n;
char c;
int i;
```

(No lugar de “`unsigned int`” podemos escrever, simplesmente, “`unsigned`”.) Existem ainda os tipos `short int` e `long int` e as correspondentes versões com prefixo “`unsigned`”, mas esses tipos serão usados apenas marginalmente neste sítio.

O tipo `unsigned char`

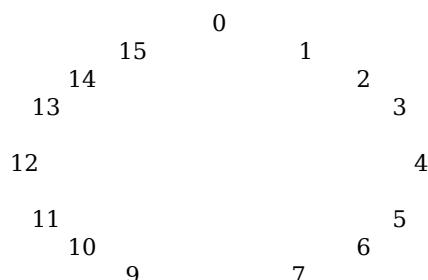
Um `unsigned char` é um inteiro-sem-sinal no intervalo $0 \dots 2^8 - 1$, ou seja, $0 \dots 255$. Cada `unsigned char` é implementado em 1 [byte](#), usando [notação binária](#).

Os inteiros fora do intervalo $0 \dots 255$ são *reduzidos módulo 2^8* , ou seja, representados pelo resto da divisão por 256. Em outras palavras, todo inteiro N sem sinal é representado pelo `unsigned char` u tal que a diferença $N - u$ é um múltiplo de 256.

Há quem goste de usar “`byte`” como apelido de “`unsigned char`”. Para fazer isso, basta introduzir a definição

```
typedef unsigned char byte;
```

Exemplo. Para que o exemplo seja mais didático, suporemos que cada byte tem *apenas 4 bits*. Nessas condições, o valor de todo `unsigned char` pertence ao intervalo $0 \dots 2^4 - 1$. Esse intervalo pode ser representado por um círculo para sugerir a redução módulo 2^4 :



Por exemplo, 16 é representado por 0 (pois $16 = 1 \times 2^4 + 0$), 17 é representado por 1 (pois $17 = 1 \times 2^4 + 1$) e 36 é representado por 4 (pois $36 = 2 \times 2^4 + 4$).

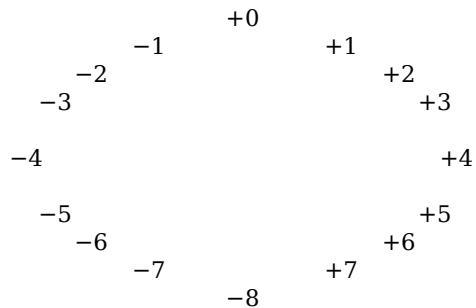
Caracteres. Números do tipo `unsigned char` são usados principalmente para [representar caracteres](#). A [tabela ASCII](#) converte o byte que representa o `unsigned char` em um [caractere ASCII](#). Note que [apenas os números naturais menores que 128](#) representam caracteres.

O tipo `char`

Um `char` é um inteiro-com-sinal no intervalo $-2^7 \dots 2^7 - 1$, ou seja, $-128 \dots 127$. Cada `char` é implementado em 1 [byte](#) usando [notação complemento-de-dois](#). (Portanto, entre 0 e 127, um `char` e um `unsigned char` são representados pelo mesmo byte.)

Inteiros fora do intervalo $-128 \dots 127$ são *reduzidos módulo 2^8* , ou seja, todo inteiro N com sinal é representado pelo `char` c tal que a diferença $N - c$ é um múltiplo inteiro (positivo ou negativo) de 256.

Exemplo. Para que o exemplo seja mais didático, suporemos que todo byte tem *apenas 4 bits*. Nessas condições, o valor de todo `char` pertence ao intervalo $-8 \dots 7$. Esse intervalo pode ser representado por um círculo para sugerir a redução módulo 2^4 :



Por exemplo, 8 módulo 2^4 é -8 (pois $8 = 1 \times 2^4 - 8$), 9 módulo 2^4 é -7 (pois $9 = 1 \times 2^4 - 7$) e -30 módulo 2^4 é 2 (pois $-30 = -2 \times 2^4 + 2$).

Caracteres. Números do tipo `char` são usados principalmente para [representar caracteres](#): o byte que representa o número é convertido em um [caractere ASCII](#) pela [tabela ASCII](#). [Apenas os números entre 0 e 127 representam caracteres](#). Nesse intervalo, um `char` e um `unsigned char` são implementados pelo mesmo byte e portanto representam o mesmo caractere.

Exercícios 1

1. Escreva, em notação binária, o menor e o maior valor que um `unsigned char` pode ter. Repita o exercício com `char`.
2. Escreva, em [notação hexadecimal](#), o menor e o maior valor que um `unsigned char` pode ter. Repita o exercício com `char`.
3. Qual o `unsigned char` que representa 1000? Qual o `char` que representa 1000? Qual o `char` que representa -1000 ?

4. Escreva e teste um programa que exiba na tela os [caracteres](#) representados pelos chars 32 a 127. (Veja [impressão em formato %c](#) na página *Entrada e saída*.) Exiba dez caracteres por linha.
5. Escreva e teste um programa que tente exibir na tela os [caracteres](#) representados pelos bytes cujo primeiro bit é 1. (Veja [impressão em formato %c](#) na página *Entrada e saída*.)
6. Escreva um programa que receba dois caracteres do [alfabeto ASCII](#) digitados pelo usuário no terminal e diga se o primeiro vem antes ou depois do segundo na tabela ASCII. (Veja [leitura em formato %c](#) na página *Entrada e saída*.)
7. BIBLIOTECA CTYPE. Familiarize-se com as funções da [biblioteca ctype](#). A função `isspace`, por exemplo, decide se um dado [caractere ASCII](#) é um [branco](#). A função `isalpha` decide se um dado caractere ASCII é uma letra.

O tipo `unsigned int`

Um `unsigned int` é um inteiro-sem-sinal no intervalo $0 \dots 2^{8s}-1$. Cada `unsigned int` é implementado em s [bytes](#) consecutivos, usando [notação binária](#). O valor de s depende do computador, mas em geral é 4. Em cada computador, o valor de s é dado pela expressão `sizeof (unsigned int)` e o número $2^{8s}-1$ está registrado na constante `UINT_MAX`, definida na [interface limits.h](#).

Inteiros maiores que `UINT_MAX` são *reduzidos módulo `UINT_MAX+1`*, ou seja, todo inteiro N é representado pelo `unsigned int` n tal que a diferença $N - n$ é um múltiplo de `UINT_MAX+1`.

Desse ponto em diante, os exemplos supõem que $s = 4$. Assim, `UINT_MAX` vale $2^{32}-1$, ou seja, 4294967295.

Aritmética `unsigned int`. As operações de adição, subtração e multiplicação entre números do tipo `unsigned int` estão sujeitas a *transbordamento* (= [overflow](#)), pois o resultado exato de uma operação pode fugir do intervalo $0 \dots \text{UINT_MAX}$. (Em geral isso não ocorre pois trabalhamos com números pequenos.) Overflows não são tratados como algo excepcional e o resultado exato de cada operação é tacitamente reduzido módulo `UINT_MAX+1`. Por exemplo:

```
unsigned int n, m, x;
n = 4000000000;
m = 300000000;
x = n + m; // overflow
// x vale 5032704
```

A operação de divisão entre `unsigned int`s não está sujeita a overflow, mas o quociente é truncado: a expressão $9/2$, por exemplo, tem valor $\lfloor 9/2 \rfloor$, ou seja, o [piso](#) de $9/2$.

O tipo `int`

Um `int` é um inteiro-com-sinal no intervalo $-2^{8s-1} \dots 2^{8s-1}-1$. Cada `int` é implementado em s [bytes](#) consecutivos, usando [notação complemento-de-dois](#). O valor de s depende do computador, mas em geral é 4. Em cada computador, o valor de s é dado pela expressão `sizeof (int)`, igual a `sizeof (unsigned int)`. Os números -2^{8s-1} e $2^{8s-1}-1$ estão registrados nas constantes `INT_MIN` e `INT_MAX` respectivamente, ambas definidas na [interface limits.h](#). Como seria de se esperar, `INT_MAX - INT_MIN = UINT_MAX`.

Inteiros fora do intervalo `INT_MIN..INT_MAX` são reduzidos módulo `UINT_MAX+1`, ou seja, todo inteiro N com sinal é representado pelo `int` i tal que a diferença $N - i$ é um múltiplo (positivo ou negativo) de `UINT_MAX+1`.

Desse ponto em diante, suporemos que $s = 4$. Assim, `INT_MIN` vale -2^{31} , ou seja, -2147483648 , e `INT_MAX` vale $2^{31}-1$, ou seja, 2147483647 .

Aritmética int. As operações de adição, subtração e multiplicação entre números do tipo `int` estão sujeitas a *transbordamento* (= [overflow](#)), pois o resultado exato de uma operação pode fugir do intervalo `INT_MIN..INT_MAX`. (Em geral isso não ocorre pois trabalhamos com números próximos de zero.) Overflows não são tratados como algo excepcional e o resultado exato de cada operação é tacitamente reduzido módulo `UINT_MAX+1`. Por exemplo:

```
int i, j, x;
i = 2147483000;
j = 2147483000;
x = i + j; // overflow
// x vale -1296
```

A atribuição de um `unsigned int` a um `int` também poderia resultar em overflow e portanto é feita módulo `UINT_MAX+1`. Por exemplo:

```
int i;
unsigned int n;
n = 2147483700;
i = n; // overflow
// i vale -2147483596
```

A operação de divisão entre `ints` não está sujeita a overflow, mas o quociente é truncado: a expressão $9/2$, por exemplo, tem valor $\lfloor 9/2 \rfloor$, ou seja, o [piso](#) de $9/2$. Mas no caso de números estritamente negativos, o resultado da divisão é arredondado em direção ao zero: a expressão $-9/2$ tem valor $-\lfloor 9/2 \rfloor$, não $-\lfloor -9/2 \rfloor$.

Exercícios 2

1. `SIZEOF`. Compile e execute o seguinte programa:

```
int main (void) {
    printf ("sizeof (unsigned): %lu\n",
            sizeof (unsigned));
    printf ("UINT_MAX: %u\n",
            UINT_MAX);
    printf ("sizeof (int) = %lu\n",
            sizeof (int));
    printf ("INT_MIN: %d\nINT_MAX: %d\n",
            INT_MIN, INT_MAX);
    return EXIT_SUCCESS; }
```

2. Escreva os números `UINT_MAX`, `INT_MIN` e `INT_MAX` em [notação hexadecimal](#).
3. Suponha que precisamos contar o número de ocorrências de um certo fenômeno num computador em que `sizeof (unsigned)` vale 2. Sabemos de antemão que o fenômeno não ocorre mais que 65535 vezes. Podemos usar uma variável do tipo `unsigned` para contar as ocorrências do fenômeno? Que fazer se o fenômeno puder ocorrer mais que 65535 vezes? Proponha uma solução que use [listas encadeadas](#) para representar um contador em base 100.
4. Suponha que `sizeof (unsigned)` vale 2 no seu computador. Qual o valor de $60000 + 30000$ em aritmética `unsigned int`? Qual o valor de $30000 + 15000$ em aritmética `int`? Qual o valor de 60000×11 em aritmética `unsigned int`? Qual o valor de 30000×2 em aritmética `int`?
5. ★ DETECÇÃO DE OVERFLOW. Escreva uma função [booleana](#) que receba `unsigned ints` n e m e decida (`true` ou `false`) se a adição exata de n e m produz overflow. Repita o exercício supondo que n e m são do tipo `int`.
6. △ ATÉ ONDE? Escreva um programa que receba um `unsigned int` n e imprima as potências n^2, n^3, n^4, n^5 , etc. O programa só deve parar quando não for capaz de armazenar a próxima potência em uma variável do tipo `unsigned int`.

Constantes

O código de quase todo programa em C contém [constantes inteiras](#). Muitos programas também têm [constantes-caractere](#). Por exemplo:

```
a = 999;  
c = 'a';
```

As constantes inteiras (como 999 no exemplo) são tratadas como se fossem do [tipo int](#) e devem ter valor entre INT_MIN e INT_MAX.

As constantes-caractere (como 'a' no exemplo acima) são embrulhadas em aspas para que não sejam confundidas com nomes de variáveis. Elas estão restritas ao [alfabeto ASCII](#) e portanto expressões como 'Ã' e 'ç' não são válidas. O valor de uma constante-caractere é dado pela [tabela ASCII](#) (assim, 'a' é o mesmo que 97).

Constantes-caractere são do [tipo int](#) e não do [tipo char](#), como seria de se esperar. Mas é claro que quando a constante é atribuída a uma variável do tipo char, ela é convertida em um byte (por exemplo, 'a' é convertida em 01100001).

A título de ilustração, o seguinte fragmento de programa exibe as vinte e seis letras maiúsculas do alfabeto ASCII. Note que essas letras ocupam posições consecutivas na tabela ASCII e estão em ordem alfabética. Algo análogo acontece com as letras minúsculas.

```
char c;  
for (c = 'A'; c <= 'Z'; ++c)  
    printf ("%c ", c);
```

Exercícios 3

1. Qual a diferença entre '0', 'o', '0', '\0' e 0?
2. Qual o efeito do seguinte fragmento de programa?

```
for (int i = 1; i <= 26; ++i)  
    printf ("%c\n", 'a' + i - 1);
```

Expressões aritméticas que misturam tipos

Operações aritméticas entre operandos do tipo char e/ou unsigned char *não* são executadas módulo 2⁸, como poderíamos imaginar. Nessas operações, todos os operandos são previamente convertidos (“promovidos”) ao tipo int e a operação é executada em [aritmética int](#).

Por exemplo, se as variáveis u e v são do tipo unsigned char e têm valor 255 e 2 respectivamente, a expressão u + v é do tipo int e tem valor 257. (É claro, entretanto, que a atribuição de u + v a uma variável do tipo unsigned char é feita módulo 2⁸.)

Considerações análogas valem para expressões aritméticas sobre operandos de tipos mistos, como int, char e unsigned char. Por exemplo, se a variável c é do tipo char e tem valor 127, a expressão c + 2 é do tipo int e tem valor 129 (a constante 2 é do tipo int por definição).

Exercícios 4

1. Qual o efeito do seguinte fragmento de código?

```
unsigned char u, v, w;  
u = 255; v = 2;  
printf ("%d", u + v);
```

```
w = u + v;
printf ("%d", w);
```

2. Qual o efeito dos dois fragmentos de código a seguir?

```
unsigned char u;
for (u = 0; u < 256; ++u)
    printf (".");

char c;
for (c = 0; c < 128; ++c)
    printf (".");
```

Tipos inteiros grandes

Algumas aplicações envolvem números inteiros que não cabem em um `int`. Para atender essas aplicações, a linguagem C tem o tipo-de-dados `long int`, que ocupa mais bytes que o tipo `int`. No meu computador, um `long int` ocupa 8 bytes, ou seja, 64 bits. (Mas veja a página [Is there any need of "long" data type in C and C++?](#) em [GeeksforGeeks](#).)

Perguntas e respostas

- PERGUNTA: Em lugar de `int`, não deveríamos usar os novos tipos-de-dados `int8_t`, `int16_t`, `int32_t`, etc., definidos na interface [`stdint.h`](#)? (A mesma pergunta vale para `unsigned int` versus `uint8_t`, `uint16_t`, etc.)

RESPOSTA: O tipo-de-dados `int32_t`, por exemplo, ocupa exatamente 4 bytes qualquer que seja o valor de `sizeof (int)`. Os demais tipos são definidos de maneira análoga. Para algumas aplicações, o uso desses tipos melhora a portabilidade dos programas. Entretanto, estas notas estão mais preocupado com os algoritmos que com os detalhes de suas implementações. Por isso, continuo usado `int` e `unsigned` neste sítio. (Veja [resposta de Matt Whiting](#) à pergunta “Why does the C library have their own Int and other datatypes?” no Quora.)

Veja os verbetes [C syntax](#), [C data types](#) e [Integer \(computer science\)](#) na Wikipedia.

Veja [Type Conversion in C](#) em [GeeksforGeeks](#).

Atualizado em 2018-07-20

<https://www.ime.usp.br/~pf/algoritmos/>

Paulo Feofiloff

[DCC-IME-USP](#)

Strings e cadeias de caracteres



Grande parte dos dados no mundo todo tem a forma de texto. Em outras palavras, grande parte dos dados está em cadeias de caracteres. Muitas dessas cadeias estão armazenadas como strings em arquivos digitais. Este capítulo discute os conceitos de *cadeia de caracteres* e *string* e a relação entre esses conceitos.

Strings

Na linguagem C, uma *string* é um [vetor](#) de [bytes](#) em que o byte nulo `00000000` é interpretado como uma sentinela que marca o fim da parte relevante do vetor. Eis um pequeno exemplo de string:

```
01000001 01000010 01000011 00000000 01000100
```

(os símbolos 0 e 1 representam bits). Nesse exemplo, apenas os 4 primeiros bytes constituem a string propriamente dita.

O *comprimento* (= *length*) de uma string é o seu número de bytes, sem contar o byte nulo final. (Assim, a string acima tem comprimento 3.) Uma string é *vazia* se tem comprimento zero.

Cada byte de uma string é tratado como um [char](#), e portanto uma string é um vetor de chars. Entretanto, para evitar confusão entre “char” e “caractere”, podemos adotar byte como sinônimo de char:

```
typedef char byte;
```

O *endereço* de uma string é o [endereço](#) do seu primeiro byte. Em discussões informais, é usual confundir uma string com o seu endereço. Assim, a expressão “considere a string `str`” deve ser entendida como “considere a string cujo endereço é `str`”. O seguinte exemplo constrói uma string `str` byte-a-byte:

```
byte *str;
str = malloc (10 * sizeof (byte));
str[0] = 65;
str[1] = 66;
str[2] = 67;
str[3] = 0;
str[4] = 68;
```

Depois da execução desse fragmento de programa, o vetor `str[0..3]` contém a string `01000001 01000010 01000011 00000000`. A porção `str[4..9]` do vetor é ignorada.

Poderíamos tratar strings como um novo [tipo-de-dados](#). Para isso, bastaria introduzir o [typedef](#)

abaixo e passar a escrever “string” no lugar de “byte *”.

```
typedef byte *string;
```

Exercícios 1

1. Escreva uma função [booleana](#) que decida se duas strings de mesmo comprimento diferem em exatamente um byte.
2. Escreva uma função que receba um byte c e devolva uma string cujo único byte é c .
3. VAZIO VERSUS NULL. Suponha que str é uma string. Qual a diferença entre as afirmações “ str é uma string vazia” e “ str é NULL”?
4. PEDAÇO DE STRING. Escreva uma função que receba uma string str e inteiros [positivos](#) i e j e devolva uma string com o mesmo conteúdo que o [segmento](#) $str[i..j]$. Escreva duas versões: na primeira, sua função não deve alocar novo espaço e pode alterar a string str que recebeu; na segunda, sua função deve devolver uma cópia do segmento $str[i..j]$ e não pode alterar a string str que recebeu.
5. Escreva uma função [booleana](#) que receba strings s e t e decida se s é um [segmento](#) de t . Escreva um programa que use a função para contar o número de ocorrências de uma string s em uma string t .

Cadeias de caracteres

Um *texto*, ou *cadeia de caracteres*, é uma sequência de símbolos tipográficos (letras, dígitos, sinais de pontuação, etc.). Esta página, por exemplo, consiste em uma longa cadeia de caracteres. Eis um exemplo muito menor:

Função C99

(o sétimo caractere da cadeia é um espaço). O *comprimento* de uma cadeia de caracteres é o número de caracteres da cadeia. A cadeia do exemplo tem comprimento 10.

Strings. Em arquivos digitais e na memória do computador, cadeias de caracteres são representadas por [strings](#). Para definir essa representação, é preciso convencionar uma correspondência entre caracteres e bytes. A correspondência mais antiga e mais básica é a [tabela ASCII](#), que associa a cada caractere do [alfabeto ASCII](#) um byte entre 00000000 e 01111111, ou seja, entre 0 e 127. Por exemplo, a cadeia de caracteres ABC é representada pela string 65 66 67 0. Qualquer string cujos bytes pertencem ao intervalo 0..127 é chamada *string ASCII*.

Caracteres que não pertencem ao alfabeto ASCII (como á, ã, é, ç, etc.) são representados por dois ou mais bytes consecutivos. Como [estamos supondo](#) que nosso ambiente de programação usa [codificação UTF-8](#), cada caractere é representado por até quatro bytes consecutivos e o primeiro byte de cada caractere não-ASCII é maior que 127. (Veja a tabela [UTF-8 encoding table and Unicode characters](#).) Por exemplo, a cadeia de caracteres Função C99 é representada pela string 70 117 110 195 167 195 163 111 32 67 57 0.

O código UTF-8 foi [construído de tal maneira](#) que o [caractere \0](#) (que pertence ao alfabeto ASCII) é o único cujo código contém um byte nulo. Assim, o primeiro byte nulo da string indica o fim da string bem como o fim da cadeia de caracteres que a string representa.

Exercícios 2

1. Suponha que `str` é uma string que representa uma cadeia de caracteres *C*. Dê uma cota inferior e uma cota superior para o comprimento de `str` em função do comprimento de *C*. Repita o exercício supondo que `str` é uma string ASCII.
2. [Sedgewick 3.57] PALÍNDROMOS. Escreva uma função `booleana` que decida se uma `string ASCII` dada é um palíndromo (ou seja, se o inverso da string é igual a ela). Escreva um programa para testar a função.
3. [Sedgewick 3.56] Escreva uma função que receba uma `string ASCII` e exiba uma tabela com o número de ocorrências de cada um dos caracteres do alfabeto ASCII na string. Escreva um programa para testar a função.
4. [Sedgewick 3.60] Escreva uma função que receba uma `string ASCII` e substitua cada `segmento` de dois ou mais `espaços` por um só espaço. Refaça o exercício depois de remover a restrição a strings ASCII.
5. ★ Escreva uma função `booleana` que decida se uma dada string é `ASCII` ou não. Suponha codificação UTF-8.
6. ★ NÚMERO DE CARACTERES. Escreva uma função que receba uma string e calcule o `comprimento da cadeia de caracteres` que a string representa em código UTF-8. (Dica: Estude a [estrutura do código UTF-8](#).) [\[Solução\]](#)

Leitura e escrita de cadeias

Para obter a cadeia de caracteres que uma dada string representa, é preciso decodificar a string. No caso de `strings ASCII`, a decodificação é simples: basta consultar a [tabela ASCII](#). Em geral, entretanto, a decodificação é mais complexa. Felizmente, a função `printf` com a especificação de formato `%s` cuida da decodificação. Por exemplo, o fragmento de programa

```
byte str[100] = {97,195,167,195,163,111,0};
printf ("%s\n", str);
```

exibe a cadeia de caracteres

ação

Uma alternativa a `printf` é a função `fputs` (o nome é uma abreviatura de *file put string*) da [biblioteca stdio](#). Essa função grava uma string num arquivo. Por exemplo,

```
fputs (str, arquivo);
```

Se o segundo argumento for `stdout`, a string `str` exibida na tela do terminal depois de ser convertida em uma cadeia de caracteres.

Leitura. A especificação de formato `%s` também pode ser usada com a função de leitura `scanf`. Por exemplo, o fragmento de programa abaixo lê uma cadeia de caracteres do teclado, converte a cadeia numa string (acrescentando um byte nulo ao final) e armazena a string no vetor `str`:

```
byte str[100];
scanf ("%s", str);
```

A função `scanf` interrompe a leitura ao encontrar o primeiro caractere `branco` (espaço, tabulação, fim de linha, etc.). Para ler uma cadeia de caracteres que contém brancos, use a função `fgets` (o nome é uma abreviatura de *file get string*) da biblioteca `stdio`. Se o último argumento de `fgets` for `stdin`, a função lê uma cadeia de caracteres do teclado até o fim da linha (tecla ↵), converte a cadeia numa string, e armazena a string no endereço indicado pelo primeiro argumento:

```
byte str[100];
fgets (str, 100, stdin);
```

O segundo argumento de `fgets` é uma proteção contra linhas muito longas, mais longas que o

espaço reservado para o primeiro argumento. No exemplo acima, se o número de bytes da cadeia de caracteres a ser lida for maior que 99, somente os 99 primeiros bytes serão armazenados em `str`. (Se algum dos caracteres for representado por dois ou mais bytes, o número de *caracteres* lidos pode ser menor que 99.)

Cadeias constantes

Para especificar uma cadeia [constante](#), basta embrulhar a cadeia num par de aspas duplas retas. (Cadeias constantes aparecem com frequência em programas C; basta lembrar o primeiro argumento das funções [printf](#) e [scanf](#).)

Quando uma cadeia constante é atribuída a uma string, ela é convertida numa sequência de bytes e um byte nulo é acrescentado ao final. Por exemplo,

```
byte *str;  
str = "ABC";
```

produz efeito semelhante ao que aparece num dos exemplos [acima](#). Os caracteres de uma cadeia constante não precisam pertencer ao [alfabeto ASCII](#); podemos, por exemplo, dizer

```
str = "Função";
```

Nesse caso, supondo codificação UTF-8, a string `str` terá 9 bytes: os três primeiros caracteres ocupam 1 byte cada, os dois caracteres seguintes [ocupam 2 bytes cada](#), o último caractere ocupa 1 byte, e há um byte nulo no fim.

Exemplo. A seguinte função conta o número de vogais em uma [string ASCII](#):

```
int contaVogais (byte s[]) {  
    byte *vogais;  
    vogais = "aeiouAEIOU";  
    int numVogais = 0;  
    for (int i = 0; s[i] != '\0'; ++i) {  
        byte ch = s[i];  
        for (int j = 0; vogais[j] != '\0'; ++j) {  
            if (vogais[j] == ch) {  
                numVogais += 1;  
                break;  
            }  
        }  
    }  
    return numVogais;  
}
```

Exercícios 3

1. Qual o efeito dos dois fragmentos de programa a seguir? O que há de errado com o segundo?

```
byte *s;  
s = "ABC";  
printf ("%s\n", s);  
  
byte s[20];  
s = "ABC";  
printf ("%s\n", s);
```

2. Qual a diferença entre "A" e 'A'?
3. Qual a diferença entre "mno" e "m\no"? Qual a diferença entre "MNOP" e "MNØP"? Qual a diferença entre "MN\ØP" e "MNØP"?
4. Reescreva a função `contaVogais` depois de remover a restrição a strings ASCII. Suponha que a

string representa um texto em português e conte as letras á, ã, É, etc. como vogais.

5. ★ REMOVE ACENTOS. Escreva uma função que remova todos os sinais diacríticos de uma cadeia de caracteres dada, trocando Á por A, ã por a, ç por c, e assim por diante. A cadeia de caracteres dada é representada por uma string em código UTF-8. Você pode supor que apenas os caracteres usados em português aparecem na cadeia, e portanto cada caractere ocupa no máximo 2 bytes. O resultado da função deve ser uma string ASCII.

Veja minha página [Manipulação de strings](#).

Veja os verbetes [Null-terminated string](#) e [C string handling](#) na Wikipedia.

Veja a página [GNU libunistring](#): bibliotecas de manipulação de strings UTF-8.

Veja o artigo [On Character Strings](#) de Tim Bray.

Veja [What's difference between char s\[\] and char *s in C?](#) em [GeeksforGeeks](#).

Atualizado em 2018-07-20

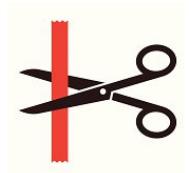
<https://www.ime.usp.br/~pf/algoritmos/>

Paulo Feofiloff

DCC-IME-USP



Manipulação de strings



Este capítulo trata da conversão de [strings](#) em números e das operações básicas sobre strings implementadas na biblioteca `string`.

Strings que representam números

Uma [cadeia de caracteres decimal](#) é qualquer sequências de dígitos decimais (caracteres 0 a 9) possivelmente precedida dos caracteres + ou -. Uma *string decimal* é qualquer [string ASCII](#) que representa uma cadeia de caracteres decimal. Como converter uma string decimal no correspondente número do [tipo int](#)?

A [função atoi](#) (o nome é uma abreviatura de *alphanumeric to int*) da biblioteca `stdlib` recebe uma string decimal e devolve o correspondente `int`. Infelizmente, a função não verifica se a string representa um `int` válido (em particular, não verifica se a string é longa demais) e assim abre as portas para o seu mau uso, especialmente com [argumentos na linha de comando](#). Diante isso, é melhor usar da função `strtol`, embora ela seja mais complexa.

A [função strtol](#) (o nome é uma abreviatura de *string to long*) da biblioteca `stdlib` converte uma string decimal no correspondente [long int](#). A função aceita não apenas strings decimais mas também strings que representam números em outras bases; o terceiro argumento especifica a base. Por exemplo,

```
strtol ("-9876", NULL, 10)
```

devolve o número `-9876`. (Vamos ignorar, por ora, o significado do segundo argumento.)

Exercícios 1

1. A-TO-I. Escreva uma função que imite comportamento de `atoi`. Sua função deve verificar se a string dada é válida. Se a string for longa demais, descarte os *últimos* dígitos.)
2. I-TO-A. Escreva uma função `itoa` (o nome é uma abreviatura de *int to alphanumeric*) que receba um inteiro `n` e devolva uma string decimal que represente `n`. Por exemplo, o inteiro `-123` deve ser convertido na string `"-123"`.
3. DE STRING BINÁRIA PARA INT. Escreva uma função que receba uma string ASCII de 0s e 1s, interprete essa string como um número natural em [notação binária](#) e devolva o correspondente `int`. Por exemplo, a string `"1111011"` deve ser convertida no inteiro `123`. (Se a string for longa demais, descarte os *últimos* dígitos.)
4. DE INT PARA STRING BINÁRIA. Escreva uma função que receba um número natural `n` e devolva a

string ASCII de 0s e 1s que represente n em [notação binária](#). Por exemplo, o número 123 deve ser convertido na string "1111011".

5. Escreva uma função para converter uma string decimal na correspondente string [hexadecimal](#). Para testar a função, escreva um programa que receba uma cadeia de caracteres decimal pela linha de comando e imprima a correspondente cadeia hexadecimal. Repita o exercício para converter hexadecimal em decimal, decimal em [octal](#), octal em decimal, etc.

Ordem lexicográfica

Um byte b é *menor* que um byte c se o número representado por b em [notação binária](#) é menor que o número representado por c . Por exemplo, 01000001 é menor que 01011010. No caso de bytes cujo primeiro bit é 0, essa relação de ordem entre bytes é consistente com a ordem usual ('A' < 'B' < ... < 'Z') das letras no alfabeto ASCII (conforme a [tabela ASCII](#)).

Agora que definimos a relação de ordem entre bytes, podemos tratar da relação de ordem entre strings. Dizemos que uma string s é *lexicograficamente menor* que uma string t se o primeiro byte de s que difere do correspondente byte de t é menor que o byte de t . Mais precisamente, para decidir se s é lexicograficamente menor que t basta fazer o seguinte. Procure a primeira posição, digamos k , em que as duas strings diferem. Se $s[k] < t[k]$ então s é lexicograficamente menor que t . Se $s[k] > t[k]$ então t é lexicograficamente menor que s . Se k não está definido então s e t são iguais ou uma é [prefixo](#) próprio da outra; no segundo caso, a string mais curta é lexicograficamente menor que a mais longa.

Uma lista de strings está *em ordem lexicográfica* se cada string da lista é lexicograficamente menor que todas as seguintes. (Há quem diga “ordem alfabética” em lugar de “ordem lexicográfica”, mas isso não está correto.) No caso de [strings ASCII](#), a ordem lexicográfica é análoga à ordem em que palavras aparecem em um dicionário (embora coloque todas as letras maiúsculas antes de todas as minúsculas). Por exemplo, a seguinte lista de strings está em ordem lexicográfica:

```
a  
ab  
abc  
abcd  
abd  
ac  
acd  
ad  
b  
bc  
bcd  
bd  
c  
cd  
d
```

Exercícios 2

1. Escreva uma função que receba duas strings digamos s e t , e decida se s é lexicograficamente menor que t .
2. O seguinte fragmento de programa pretende decidir se “abacate” é lexicograficamente menor que “banana”. O que está errado?

```
char *a, *b;  
a = "abacate"; b = "banana";  
if (a < b)  
    printf ("%s é menor que %s\n", a, b);  
else  
    printf ("%s é maior que %s\n", a, b);
```

3. O seguinte fragmento de programa pretende decidir se “abacate” é lexicograficamente menor que “amora”. O que está errado?

```
char *a, *b;
a = "abacate"; b = "amora";
if (*a < *b)
    printf ("%s é menor que %s\n", a, b);
else
    printf ("%s é maior que %s\n", a, b);
```

A biblioteca string

A [biblioteca string](#) (não confunda com a biblioteca obsoleta `strings`) contém várias funções de manipulação de strings. Segue uma descrição das funções mais importantes da biblioteca.

String length. A função `strlen` recebe uma string e devolve o seu [comprimento](#). Essa função poderia ser escrita assim:

```
unsigned int strlen (char *s) {
    int k;
    for (k = 0; s[k] != '\0'; ++k)
        ;
    return k;
}
```

String copy. A função `strcpy` recebe duas strings e copia a segunda (inclusive o byte nulo final) para o espaço ocupado pela primeira. O conteúdo original da primeira string é perdido. Não invoque a função se o comprimento da primeira string for menor que o da segunda. (*Buffer overflow* é uma das mais comuns origens de bugs de segurança!) Essa função poderia ser escrita assim:

```
void strcpy (char *s, char *t) {
    int i;
    for (i = 0; t[i] != '\0'; ++i)
        s[i] = t[i];
    s[i] = '\0';
}
```

(Na verdade, a função `strcpy` não é do tipo `void`: ela devolve o seu primeiro argumento. Isso é útil em algumas situações, mas em geral o usuário só está interessado no *efeito* da função e não no que ela devolve.)

A função `strcpy` pode ser usada da seguinte maneira para obter um efeito semelhante ao do [exemplo no início do capítulo Strings e cadeias de caracteres](#):

```
char s[10];
strcpy (s, "ABC");
```

String compare. A função `strcmp` recebe duas strings e compara as duas [lexicograficamente](#). Ela devolve um número estritamente negativo se a primeira string for lexicograficamente menor que a segunda, devolve 0 se as duas strings são iguais e devolve um número estritamente positivo se a primeira string for lexicograficamente maior que a segunda. Essa função poderia ser escrita assim:

```
int strcmp (char *s, char *t) {
    int i;
    for (i = 0; s[i] == t[i]; ++i)
        if (s[i] == '\0') return 0;
    unsigned char si = s[i], ti = t[i];
    return si - ti;
}
```

(Vale lembrar que o valor da expressão `si - ti` é calculado em [aritmética int](#) e não [módulo 256](#).)

Se as strings `s` e `t` são [ASCII](#), elas representam cadeias de [caracteres ASCII](#). Nesse caso, a ordem lexicográfica entre as strings coincide com a ordem em que as cadeias apareceriam em um dicionário (pois a ordem `... 0 .. 9 .. A B .. Z .. a b .. z ...` em que os caracteres aparecem na [tabela ASCII](#) é consistente com a ordem alfabética usual). Por exemplo, a seguinte sequência de palavras está em ordem lexicográfica:

```
abacaXi  
abacate  
abacates  
abacaxi
```

Se as strings não são ASCII (ou seja, se têm bytes maiores que 127), a situação é mais complexa. [Nesse caso](#), cada string representa uma cadeia de caracteres em [código UTF-8](#), e portanto cada caractere é representado por 1, 2, 3 ou 4 bytes consecutivos. Felizmente, o código UTF-8 [foi construído de tal maneira que](#) \triangleq a ordem lexicográfica entre as strings coincide com a ordem em que as cadeias apareceriam em um dicionário se os caracteres fossem colocados em ordem crescente dos seus [números Unicode](#), ou seja, `... 0 .. 9 .. A B C .. Z .. a b c .. z .. À Á .. Ç È .. à á .. ç è ..` (veja a ordenação na página [UTF-8 encoding table and Unicode characters](#)). Por exemplo, a seguinte sequência de palavras está em ordem lexicográfica:

```
abocanhar  
abrir  
abóbora  
academia  
acordar  
acácia  
adaptar  
aço  
açucarado  
ação
```

(Essa ordem lexicográfica parece estranha pois deixa as letras acentuadas longe das correspondentes letras sem acentos. Para corrigir isso, veja a função `strcoll` num dos exercícios [abaixo](#).)

Exercícios 3

1. Qual a diferença entre as expressões “`strcpy (s, t)`” e “`s = t`”?

2. Qual a diferença entre as expressões “`if (strcmp (s, t) < 0)`” e “`if (s < t)`”?

3. Discuta as diferenças entre os três fragmentos de programa a seguir:

```
char a[8], b[8];  
strcpy (a, "abacate"); strcpy (b, "banana");  
  
char *a = malloc (8), *b = malloc (8);  
strcpy (a, "abacate"); strcpy (b, "banana");  
  
char *a, *b;  
a = "abacate"; b = "banana";
```

4. Estude a documentação da função [strncpy](#) da biblioteca `string`.

5. Escreva uma função que receba uma string `s` e devolva uma cópia de `s`. (Compare sua função com a função `strdup` da [biblioteca string](#). Qual a diferença entre essa função e `strcpy`?)

6. Escreva uma função que receba uma string ASCII `s` e um caractere ASCII `c` e devolva o índice da primeira posição em `s` que é igual a `c`. (Compare sua função com a função `strchr` da [biblioteca string](#). Agora faça uma versão mais geral da função para procurar `c` a partir de uma dada posição `i`.)

7. NÚMERO DE CARACTERES. Escreva uma função que receba uma string e calcule o [comprimento da](#)

[cadeia de caracteres](#) que a string representa em [código UTF-8](#).

8. Escreva uma função que decida se um vetor $vs[0..n-1]$ de strings está em ordem lexicográfica.
9. ★ Suponha dado um vetor $vs[0..n-1]$ de strings, em ordem lexicográfica, e uma string s . Problema: se vs não contém uma string igual a s , inserir s no vetor de modo a manter a ordem lexicográfica. Escreva uma função que resolva o problema e devolva 1 se a inserção tiver sido feita e 0 em caso contrário.
10. △ Suponha que strings s e t representam, em código UTF-8, as cadeias de caracteres S e T respectivamente. Prove que a ordem lexicográfica entre s e t (dada por `strcmp(s, t)`) coincide com a ordem em que as cadeias S e T estariam num dicionário se o conjunto de caracteres estivesse em ordem crescente de [números Unicode](#).
11. ★ ORDEM LEXICOGRÁFICA EM PORTUGUÊS. A ordem lexicográfica calculada pela função `strcmp` compara duas strings byte-a-byte, ignorando as características do eventual [esquema de codificação](#). Se as strings [representam cadeias de caracteres](#) em código UTF-8, a ordem lexicográfica é [consistente com a ordem crescente dos números Unicode dos caracteres](#). Infelizmente, essa não é a ordem usual de um dicionário de português, pois as letras maiúsculas ficam longe das correspondentes minúsculas e longe das correspondentes letras com diacríticos (por exemplo, A, a e á ficam longe umas das outras). Para contornar esse inconveniente, use a variante `strcoll` de `strcmp`. △ A ordem lexicográfica calculada por `strcoll` (o nome da função é uma abreviatura de *string collate*) usa a ordenação dos caracteres determinada pela [variável de ambiente](#) `LC_COLLATE` do seu sistema. Se o valor de `LC_COLLATE` é `pt_BR.UTF-8`, por exemplo, a ordem dos caracteres é ... a A á Á à À â Â .. b B c C ç Ç d D .. z Z ... Use a função `strcoll` para escrever um programa que ordene lexicograficamente uma sequência de palavras em português codificada em UTF-8. As palavras são digitadas no terminal e a lista ordenada deve ser exibida na tela, uma por linha. Faça testes. Se o valor de `LC_COLLATE` no seu sistema não for `pt_BR.UTF-8`, você pode fazer algo assim:

```
#include <locale.h>
void main (void) {
    setlocale (LC_COLLATE, "pt_BR.UTF-8");
    ... código da ordenação ...
}
```

(Também pode mudar o valor de `LC_COLLATE`, temporariamente, antes de invocar o seu programa:
~\$ `LC_COLLATE="pt_BR.UTF-8" ./meuprograma.`)

Veja a [biblioteca strlib](#) de [Eric Roberts](#).

Veja a página [GNU libunistring](#): bibliotecas de manipulação de strings UTF-8.

Atualizado em 2018-07-20

<https://www.ime.usp.br/~pf/algoritmos/>

Paulo Feofiloff

[DCC-IME-USP](#)



Entrada e saída



Este capítulo descreve, superficialmente, as funções “de entrada” (= *input*) e “de saída” (= *output*) mais importantes da linguagem C. Todas estão na biblioteca stdio. Para ter acesso a essa biblioteca, seu programa deve incluir a [interface](#) da biblioteca por meio de

```
#include <stdio.h>
```

Teclado e tela do terminal

A função **printf** (abreviatura de *print formatted*) exibe na tela do terminal uma lista “formatada” de [números](#), [caracteres](#), [strings](#), etc. O primeiro argumento da função é uma [string](#) que [especifica o formato da impressão](#).

A função **scanf** (abreviatura de *scan formatted*) recebe do teclado uma lista de números, caracteres, strings, etc. O primeiro argumento da função é uma string que [especifica o formato da lista](#). Os demais argumentos são os [endereços](#) das variáveis onde os valores recebidos devem ser armazenados. A função trata todos os [brancos](#) como se fossem espaços (caracteres ' '). Veja um exemplo:

```
#include <stdio.h>
#include <stdlib.h>

int main (void) {
    int a, b;
    scanf ("%d %d", &a, &b);
    double media;
    media = (a + b)/2.0;
    printf ("A média de %d e %d é %f\n", a, b, media);
    return EXIT_SUCCESS;
}
```

Se o nome do programa for `average`, veremos o seguinte resultado na tela (o computador escreve em **vermelho**):

```
~$ ./average
222 333
A média de 222 e 333 é 277.500000
~$
```

Para exibir o [caractere ASCII](#) representado por um [char](#) ou [unsigned char](#), use a função `printf` com especificação de formato `%c`. Por exemplo, o fragmento de programa

```
char a = 65; // ou a = 'A'  
printf ("%d %c", a, a);
```

exibe

```
65 A
```

Para ler um caractere ASCII do teclado, use a função `scanf` com especificação de formato `%c`. A função lê o primeiro caractere não-[branco](#) (os brancos são desprezados), converte o caractere lido no [correspondente byte](#), e armazena o byte em `a`:

```
char a;  
scanf ("%c", a);
```

Para ler uma [cadeia de caracteres](#) do teclado, use `scanf` com especificação de formato `%s`. A função interrompe a leitura ao encontrar o primeiro caractere [branco](#). Por exemplo, o fragmento de programa abaixo lê uma cadeia de caracteres, converte a cadeia numa [string](#) e armazena a string no vetor `str`:

```
char str[100];  
scanf ("%s", str);
```

Arquivos

Um *arquivo* (= [file](#)) é uma sequência de [bytes](#) que reside na memória “lenta” do computador (um disco magnético, por exemplo). Os bytes de um arquivo não têm [endereços](#) individuais. Assim, o acesso ao arquivo é estritamente sequencial: para chegar ao 5º byte é preciso passar pelo 1º, 2º, 3º e 4º bytes.

Para que um programa possa manipular um arquivo, é preciso associá-lo a uma variável do tipo [FILE](#) (esse tipo está definido na interface [stdio.h](#)). A operação de associação é conhecida como *abertura* do arquivo e é executada pela função `fopen` (= *file open*). O primeiro argumento da função é o nome do arquivo e o segundo argumento é “r” ou “w” para indicar se o arquivo deve ser aberto “para leitura” (= *read*) ou “para escrita” (= *write*). A função `fopen` devolve o endereço de um [FILE](#) (ou [NULL](#), se não encontrar o arquivo especificado). Depois de usar o arquivo, é preciso *fechá-lo* com a função `fclose` (= *file close*).

Digamos, por exemplo, que o [arquivo de texto](#) `dados.txt` contém uma sequência de números inteiros (em notação decimal) separados por [brancos](#). O programa abaixo calcula a média desses números. Para ler o arquivo, o programa usa a função `fscanf` (o nome é uma abreviatura de *file scanf*), que generaliza a função `scanf`:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <stdbool.h>  
  
int main (void) {  
    FILE *entrada;  
    entrada = fopen ("dados.txt", "r");  
    if (entrada == NULL) {  
        printf ("\nNão encontrei o arquivo!\n");  
        exit (EXIT_FAILURE);  
    }  
    double soma = 0.0;  
    int n = 0;  
    while (true) {  
        int x;  
        int k = fscanf (entrada, "%d", &x);  
        if (k != 1) break;  
        soma += x;  
        n += 1;  
    }  
}
```

```

fclose (entrada);
printf ("A média dos números é %f\n", soma/n);
return EXIT_SUCCESS;
}

```

A função [fscanf](#), tal como a função `scanf`, devolve o número de objetos efetivamente lidos. Se não houver um próximo número a ser lido, devolve 0. O programa acima usa isso para detectar o fim do arquivo. (O programa do exemplo supõe que o arquivo contém pelo menos um número.)

Stdin e stdout. O “arquivo” padrão de entrada (= *standard input*) é o teclado. Ele está permanente aberto e é representado pela constante `stdin`. Portanto `fscanf (stdin, ...)` equivale a `scanf (...)`.

Algo análogo acontece com as funções `printf`, `fprintf` e o “arquivo” padrão de saída `stdout`, que corresponde à tela do terminal.

As funções `putc` e `getc`

A função mais básica de saída — mais básica que `fprintf` — é [putc](#) (o nome é uma abreviatura de *put character*). A função recebe um [byte](#) e grava-o no arquivo especificado. (Muitas vezes, o byte tem valor entre 0 e 127 e portanto representa um [caractere ASCII](#).) Se `c` é um `char` e `f` aponta um arquivo então `putc (c, f)` grava `c` no arquivo `f`. Por exemplo, `putc ('#', stdout)` exibe o caractere # na tela.

A correspondente função de entrada é [getc](#) (o nome é uma abreviatura de *get character*). Cada chamada da função lê um byte do arquivo especificado. (Muitas vezes, o byte lido representa um [caractere ASCII](#).) Por exemplo, `getc (stdin)` lê o próximo byte do teclado e devolve o byte lido.

As expressões [putchar \(x\)](#) e [getchar \(\)](#) são abreviaturas de `putc (x, stdout)` e `getc (stdin)` respectivamente.

Exemplo. O programa abaixo lê do teclado uma sequência de bytes que termina com `\n` (por exemplo, uma linha de [caracteres ASCII](#)), armazena essa linha em um vetor, e exibe os correspondentes caracteres na tela. O programa supõe que a sequência tem no máximo 100 bytes (contando com o `\n` final):

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main (void) {
    char linha[100];
    int n = 0;
    while (true) {
        linha[n] = getchar ();
        if (linha[n] == '\n') break;
        n = n + 1;
    }
    for (int i = 0; i <= n; i += 1)
        putchar (linha[i]);
    return EXIT_SUCCESS;
}

```

Outro exemplo. O programa abaixo deveria ler o primeiro byte do arquivo `dados.txt` e exibir o correspondente caractere (supõe-se que seja um [caractere ASCII](#)) na tela:

```

#include <stdio.h>
#include <stdlib.h>

```

```

int main (void) {
    FILE *entrada;
    entrada = fopen ("dados.txt", "r");
    if (entrada == NULL) exit \(EXIT\_FAILURE\);
    char c; // erro!
    c = getc (stdin);
    fclose (entrada);
    putc (c, stdout);
    return EXIT_SUCCESS;
}

```

O programa ter um defeito a ser discutido na próxima seção.

Que tipo de objeto getc devolve?

Que acontece se getc tenta ler o próximo byte de um arquivo que já se esgotou? Seria necessário que getc devolvesse algum tipo de “byte inválido”; mas todos os 256 bytes são válidos!

Para resolver esse impasse, getc *devolve um inteiro*, não um byte. O conjunto de valores int contém todos os valores char e unsigned char e mais alguns. Assim, se o arquivo tiver se esgotado, getc pode devolver um inteiro diferente de qualquer char. Mais especificamente,

1. se houver um próximo byte no arquivo, getc lê o byte como se fosse [unsigned char](#), converte-o em um int positivo, e devolve o resultado;
2. se o arquivo não tiver mais bytes, getc devolve um int estritamente negativo.

Para ser mais exato, se o arquivo não tiver mais bytes, getc devolve a constante *EOF* (o nome é uma abreviatura de *end of file*), que está definida na interface [stdio.h](#) e vale -1 na maioria dos computadores.

Resumindo, a função getc devolve um elemento de um *superconjunto* do conjunto em que estamos realmente interessados. Assim, a resposta de getc é sempre do mesmo tipo (um int), até em situações excepcionais. Esse truque é uma importante lição de projeto (*design*)!

Exemplo. O seguinte fragmento de código exibe o próximo byte do arquivo a menos que estejamos no fim do arquivo:

```

int c;
c = getc (entrada);
if (c != EOF)
    putc (c, stdout);
else
    printf ("\n0 arquivo terminou!");

```

(A propósito, se o arquivo de entrada for `stdin`, o fim do arquivo é produzido pela combinação de teclas [Ctrl D](#), que gera o [byte 4](#).)

Exercícios 1

1. Um programador propõe redefinir a função getc de modo que ela devolva um char (e não um int) e a expressão `getc (entrada)` tenha o seguinte efeito: se não houver um próximo byte no arquivo `entrada`, a função devolve o [caractere \a](#) e exibe a mensagem “arquivo acabou” na tela. Critique essa proposta.
2. Escreva um programa completo que faça uma cópia byte-a-byte do arquivo cujo nome é digitado pelo usuário. [[Solução](#).]

3. Escreva um programa que remova os comentários (do tipo `/*...*/` e do tipo `//...`) do arquivo-fonte de um programa C. O resultado deve ser gravado em um novo arquivo-fonte.

Argumentos na linha de comando

A execução de qualquer programa C consiste na execução da função `main` (que em geral invoca outras funções). A função `main` admite dois parâmetros, que chamaremos `numargs` e `arg`. O segundo parâmetro é um vetor e o primeiro é o número de elementos do vetor.

```
int main (int numargs, char *arg[]) {  
    . . .  
}
```

Cada elemento de `arg` é uma [string](#). Essas strings são digitadas pelo usuário ao invocar o programa. A primeira string digitada é o nome do programa e passará a ser o valor de `arg[0]`. As strings seguintes são conhecidas como *argumentos na linha de comando* (= *command-line arguments*) e passarão a ser os valores de `arg[1]`, `arg[2]`, ..., `arg[numargs-1]`. (O usuário digita um ou mais espaços para separar um argumento do anterior.) O valor de `numargs` é definido implicitamente pelo número de strings digitadas.

No seguinte exemplo, o nome do programa é `prog`. Se digitarmos a linha de comando

```
~$ ./prog -a bb ccc 2222
```

`numargs` assumirá o valor 5 e `arg[0]` a `arg[4]` serão as strings "prog", "-a", "bb", "ccc" e "2222" respectivamente.

Exemplo. O seguinte programa calcula a média dos números inteiros digitados na linha de comando. Cada um dos números digitados deve estar no intervalo [INT_MIN..INT_MAX](#).

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main (int numargs, char *arg[]) {  
    int soma = 0;  
    for (int i = 1; i < numargs; ++i)  
        soma += strtol (arg[i], NULL, 10);  
    int n = numargs - 1;  
    printf ("média = %.2f\n", \(double\) soma / n);  
    return EXIT_SUCCESS;  
}
```

Se o nome do programa é `prog`, veremos a seguinte interação na tela:

```
~$ ./prog +22 33 -11 +44  
média = 22.00  
~$
```

Outro exemplo. O seguinte programa exibe uma tabela de conversão de [graus Celsius em graus Fahrenheit](#) ou vice-versa. O usuário especifica a direção da conversão, bem como o início e o fim da tabela.

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
// Programa temperatura  
// -----  
// Para obter uma tabela de conversão de graus  
// Celsius em graus Fahrenheit, digite  
//      ./temperatura c-f 10 40  
//
```

```

// A primeira coluna começará com 10 graus
// Celsius e andará em passos de 1 grau até 40
// graus Celsius. A segunda coluna trará a
// correspondente temperatura em graus
// Fahrenheit. Troque "c-f" por "f-c" para
// obter a tabela de conversão de graus
// Fahrenheit em graus Celsius.

int main (int numargs, char *arg[]) {
    if (numargs != 4) {
        printf ("Número de argumentos errado.\n");
        return EXIT_FAILURE;
    }
    int inf = strtol (arg[2], NULL, 10);
    int sup = strtol (arg[3], NULL, 10);
    // arg[2] e arg[3] devem representar inteiros
    // no intervalo INT_MIN..INT_MAX
    if (strcmp (arg[1], "c-f") == 0) {
        printf ("Celsius Fahrenheit\n");
        for (int c = inf; c <= sup; c += 1)
            printf ("%7d %10.2f\n", c, 9.0/5.0*c + 32);
        return EXIT_SUCCESS;
    }
    if (strcmp (arg[1], "f-c") == 0) {
        printf ("Fahrenheit Celsius\n");
        for (int f = inf; f <= sup; f += 1)
            printf ("%10d %8.2f\n", f, 5.0*(f-32)/9.0);
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}

```

Redirecionamento de entrada/saída. Se um argumento na linha de comando for o nome de um arquivo precedido pelo caractere “<” então esse arquivo passa a fazer o papel da [entrada padrão](#). Analogamente, se um argumento na linha de comando for o nome de um arquivo precedido pelo caractere “>” então esse arquivo passa a fazer o papel da [saída padrão](#). Por exemplo, se o arquivo `in.txt` no diretório corrente contém

222 333

e `average` é o nome do [programa no primeiro exemplo deste capítulo](#), então a invocação

```
~$ ./average < in.txt > out.txt
```

produzirá um arquivo `out.txt` com o seguinte conteúdo:

A média de 222 e 333 é 277.500000

Exercícios 2

1. Modifique o programa do [exemplo acima](#) de modo a interromper a execução se `strtol` devolver um número que não cabe em um `int`.
2. Escreva e teste um programa que exiba uma tabela com os valores de todos os argumentos digitados pelo usuário na linha de comando (e pare em seguida).
3. WORD COUNT. Escreva um programa que conte o número de ocorrências de cada caractere em um [arquivo de texto](#) cujos bytes representam [caracteres ASCII](#). O programa deve receber o nome do arquivo pela linha de comando e exibir uma tabela com o número de ocorrências de cada caractere. (Para ganhar inspiração, analise o comportamento do [utilitário wc](#).)

Perguntas e respostas

- PERGUNTA: Quando meu arquivo é exibido na tela, aparece um `\M` no fim de cada linha. Por que?
RESPOSTA: Provavelmente o arquivo foi gerado no sistema Windows e está sendo exibido no sistema Linux. No sistema Linux, o fim de uma linha é indicado pelo [caractere \n](#). Já no sistema Windows, o fim de uma linha é indicado pelo par de caracteres `\r\n`, e o [caractere \r](#) aparece na tela como `\M`.

Essa diferença entre os dois sistemas é uma fonte de dor de cabeça para quem precisa processar no Linux arquivos que vieram do Windows: é preciso lidar com os caracteres \r que aparecem no fim das linhas.

Veja o verbete [C_file input/output](#) na Wikipedia

Atualizado em 2018-05-20

<https://www.ime.usp.br/~pf/algoritmos/>

Paulo Feofiloff

[DCC-IME-USP](#)



Registros e structs

Um *registro* (= *record*) é um “pacote” de variáveis, possivelmente de tipos diferentes. Cada variável é um *campo* do registro. Na linguagem C, registros são conhecidos como **structs** (o nome é uma abreviatura de *structure*).

Definição e manipulação de structs

O seguinte exemplo declara um registro `x` com três campos que pode ser usado para armazenar datas:

```
struct {
    int dia;
    int mes;
    int ano;
} x;
```

É uma boa ideia dar um nome à classe de todos os registros de um mesmo tipo. No nosso exemplo, acho que `dma` é um nome apropriado:

```
struct dma {
    int dia;
    int mes;
    int ano;
};
struct dma x; // um registro x do tipo dma
struct dma y; // um registro y do tipo dma
```

Para se referir a um campo de um registro, basta escrever o nome do registro e o nome do campo separados por um ponto:

```
x.dia = 31;
x.mes = 12;
x.ano = 2018;
```

Registros podem ser tratados como um novo [tipo-de-dados](#). Depois da seguinte definição, por exemplo, poderemos passar a dizer “data” no lugar de “`struct dma`”:

```
typedef struct dma data;
data x, y;
```

Exemplo. A seguinte função calcula a data de fim de um evento ao receber a data de início do evento e a duração do evento em dias.

```
data fimEvento (data inicio, int duracao) {
    data fim;
    . .
    .
    .
    fim.dia = ...
    fim.mes = ...
    fim.ano = ...
    return fim;
}
```

O código foi omitido porque é um tanto enfadonho, uma vez que deve levar em conta o número de dias em diferentes meses e os anos bissextos. Eis como a função `fimEvento` poderia ser usada:

```
int main (void) {
    data a, b;
    scanf ("%d %d %d", &a.dia, &a.mes, &a.ano);
    // &a.dia significa &(a.dia)
    int dura;
    scanf ("%d", &dura);
    b = fimEvento (a, dura);
    printf ("%d %d %d\n", b.dia, b.mes, b.ano);
    return EXIT_SUCCESS;
}
```

Exercícios 1

1. Complete o código da função `fimEvento` acima.
2. Escreva uma função que receba duas structs do tipo `dma`, cada uma representando uma data válida, e devolva o número de dias que decorreram entre as duas datas.
3. Escreva uma função que receba um número inteiro que representa um intervalo de tempo medido em minutos e devolva o número equivalente de horas e minutos (por exemplo, 131 minutos equivalem a 2 horas e 11 minutos). Use uma struct como a seguinte:

```
struct hm {
    int horas;
    int minutos;
};
```

Structs e ponteiros

Cada registro tem um [endereço](#) na memória do computador. (Você pode imaginar que o endereço de um registro é o endereço de seu primeiro campo.) É muito comum usar um [ponteiro](#) para armazenar o endereço de um registro. Dizemos que um tal ponteiro *aponta* para o registro. Por exemplo,

```
data *p;           // p é um ponteiro para registros dma
data x;
p = &x;           // agora p aponta para x
(*p).dia = 31; // mesmo efeito que x.dia = 31
```

(Cuidado! Graças [às regras de precedência](#), a expressão `*p.dia` equivale a `*(p.dia)` e tem significado muito diferente de `(*p).dia`.)

A expressão `p->mes` é uma abreviatura muito útil da expressão `(*p).mes`:

```
p->dia = 31; // mesmo efeito que (*p).dia = 31
```

Exercícios 2

1. Defina um registro empregado para armazenar os dados (nome, sobrenome, data de nascimento, CPF, data de admissão, salário) de um colaborador de sua empresa. Defina um vetor de empregados para armazenar todos os colaboradores de sua empresa.
2. Um *racional* é qualquer número da forma p/q , sendo p inteiro e q inteiro não nulo. É conveniente representar um racional por um registro:

```
typedef struct {
    int p, q;
} racional;
```

Vamos convencionar que o campo q de todo racional é estritamente positivo e que o máximo divisor comum dos campos p e q é 1. Escreva funções

- reduz, que receba inteiros a e b e devolva o racional que representa a/b ;
- neg, que receba um racional x e devolva o racional $-x$;
- soma, que receba racionais x e y e devolva o racional que representa a soma de x e y ;
- produ, que receba racionais x e y e devolva o racional que representa o produto de x por y ;
- quoci, que receba racionais x e y e devolva o racional que representa o quociente de x por y .

Atualizado em 2018-05-25

<https://www.ime.usp.br/~pf/algoritmos/>

Paulo Feofiloff

[*DCC-IME-USP*](#)



Endereços e ponteiros



Os conceitos de *endereço* e *ponteiro* são fundamentais em qualquer linguagem de programação. Em C, esses conceitos são explícitos; em algumas outras linguagens eles são ocultos (e representados pelo conceito mais abstrato de *referência*). Dominar o conceito de ponteiro exige algum esforço e uma boa dose de prática.

Endereços

A memória RAM (= *random access memory*) de qualquer computador é uma [sequência](#) de [bytes](#). A posição (0, 1, 2, 3, etc.) que um byte ocupa na sequência é o *endereço* (= *address*) do byte. (É como o endereço de uma casa em uma longa rua que tem casas de um lado só.) Se *e* é o endereço de um byte então *e+1* é o endereço do byte seguinte.

Cada variável de um programa ocupa um certo número de bytes consecutivos na memória do computador. Uma variável do tipo [char](#) ocupa 1 byte. Uma variável do tipo [int](#) ocupa 4 bytes e um [double](#) ocupa 8 bytes em muitos computadores. O número exato de bytes de uma variável é dado pelo operador [sizeof](#). A expressão `sizeof (char)`, por exemplo, vale 1 em todos os computadores e a expressão `sizeof (int)` vale 4 em muitos computadores.

Cada variável (em particular, cada registro e cada vetor) na memória tem um *endereço*. Na maioria dos computadores, o endereço de uma variável é o endereço do seu primeiro byte. Por exemplo, depois das declarações

```
char c;
int i;
struct {
    int x, y;
} ponto;
int v[4];
```

as variáveis poderiam ter os seguintes endereços (o exemplo é fictício):

c	89421
i	89422
ponto	89426
v[0]	89434
v[1]	89438
v[2]	89442

O endereço de uma variável é dado pelo operador `&`. Assim, se *i* é uma variável então `&i` é o seu endereço. (Não confunda esse uso de “&” com o operador lógico *and*, que se escreve “`&&`” em C.) No exemplo acima, `&i` vale 89422 e `&v[3]` vale 89446.

Outro exemplo: o segundo argumento da função de biblioteca [scanf](#) é o endereço da variável que deve receber o valor lido do teclado:

```
int i;
```

```
scanf ("%d", &i);
```

Exercícios 1

1. TAMANHOS. Compile e execute o seguinte programa:

```
int main (void) {
    typedef struct {
        int dia, mes, ano;
    } data;
    printf ("sizeof (data) = %d\n",
            sizeof (data));
    return EXIT_SUCCESS;
}
```

2. Compile e execute o seguinte programa. (O [cast](#) (`long int`) é necessário para que `&i` possa ser impresso com [especificação de formato](#) `%ld`. É mais comum imprimir endereços em [notação hexadecimal](#), usando a especificação de formato `%p`, que exige o cast (`void *`).)

```
int main (void) {
    int i = 1234;
    printf (" i = %d\n", i);
    printf ("&i = %ld\n", (long int) &i);
    printf ("&i = %p\n", (void *) &i);
    return EXIT_SUCCESS;
}
```

Ponteiros

Um *ponteiro* (= apontador = *pointer*) é um tipo especial de variável que armazena um endereço. Um ponteiro pode ter o valor

NULL

que é um endereço “inválido”. A [macro](#) NULL está definida na [interface stdlib.h](#) e seu valor é 0 (zero) na maioria dos computadores.

Se um ponteiro `p` armazena o endereço de uma variável `i`, podemos dizer “`p` aponta para `i`” ou “`p` é o endereço de `i`”. (Em termos um pouco mais abstratos, diz-se que `p` é uma *referência* à variável `i`.) Se um ponteiro `p` tem valor diferente de NULL então

`*p`

é o *valor* da variável apontada por `p`. (Não confunda esse operador “`*`” com o operador de multiplicação!) Por exemplo, se `i` é uma variável e `p` vale `&i` então dizer “`*p`” é o mesmo que dizer “`i`”.

A seguinte figura dá um exemplo. Do lado esquerdo, um ponteiro `p`, armazenado no endereço 60001, contém o endereço de um inteiro. O lado direito dá uma representação esquemática da situação:



Tipos de ponteiros. Há vários tipos de ponteiros: ponteiros para bytes, ponteiros para inteiros, ponteiros para ponteiros para inteiros, ponteiros para [registros](#), etc. O computador precisa saber de que tipo de ponteiro você está falando. Para declarar um ponteiro `p` para um inteiro, escreva

```
int *p;
```

(Há quem prefira escrever [int* p.](#)) Para declarar um ponteiro p para um registro reg, diga

```
struct reg *p;
```

Um ponteiro r para um ponteiro que apontará um inteiro (como no caso de uma [matriz de inteiros](#)) é declarado assim:

```
int **r;
```

Exemplos. Suponha que a, b e c são variáveis inteiras e veja um jeito bobo de fazer “c = a+b”:

```
int *p; // p é um ponteiro para um inteiro
int *q;
p = &a; // o valor de p é o endereço de a
q = &b; // q aponta para b
c = *p + *q;
```

Outro exemplo bobo:

```
int *p;
int **r; // ponteiro para ponteiro para inteiro
p = &a; // p aponta para a
r = &p; // r aponta para p e *r aponta para a
c = **r + b;
```

Exercícios 2

1. Compile e execute o seguinte programa. (Veja um dos exercícios [acima](#).)

```
int main (void) {
    int i; int *p;
    i = 1234; p = &i;
    printf ("*p = %d\n", *p);
    printf (" p = %ld\n", (long int) p);
    printf (" p = %p\n", (void *) p);
    printf ("&p = %p\n", (void *) &p);
    return EXIT_SUCCESS;
}
```

Aplicação

Suponha que precisamos de uma função que troque os valores de duas variáveis inteiras, digamos i e j. É claro que a função

```
void troca (int i, int j) {
    int temp;
    temp = i; i = j; j = temp;
}
```

não produz o efeito desejado, pois [recebe apenas os valores das variáveis](#) e não as variáveis propriamente ditas. A função recebe “cópias” das variáveis e troca os valores dessas cópias, enquanto as variáveis originais permanecem inalteradas. Para obter o efeito desejado, é preciso passar à função os *endereços* das variáveis:

```
void troca (int *p, int *q)
{
    int temp;
    temp = *p; *p = *q; *q = temp;
```

```
}
```

Para aplicar essa função às variáveis i e j basta dizer `troca (&i, &j);` ou então

```
int *p, *q;
p = &i; q = &j;
troca (p, q);
```

Exercícios 3

1. Verifique que a troca de valores de variáveis discutida acima poderia ser obtida por meio de uma macro do [pré-processador](#):

```
#define troca (X, Y) {int t = X; X = Y; Y = t;}
. . .
troca (i, j);
```

2. Por que o código abaixo está errado?

```
void troca (int *i, int *j) {
    int *temp;
    *temp = *i; *i = *j; *j = *temp;
}
```

3. Um ponteiro pode ser usado para dizer a uma função onde ela deve depositar o resultado de seus cálculos. Escreva uma função hm que converta minutos em horas-e-minutos. A função recebe um inteiro mnts e os endereços de duas variáveis inteiras, digamos h e m, e atribui valores a essas variáveis de modo que m seja menor que 60 e que $60*h + m$ seja igual a mnts. Escreva também uma função main que use a função hm.

4. Escreva uma função mm que receba um vetor inteiro $v[0..n-1]$ e os endereços de duas variáveis inteiras, digamos min e max, e deposite nessas variáveis o valor de um elemento mínimo e o valor de um elemento máximo do vetor. Escreva também uma função main que use a função mm.

Aritmética de endereços

Os elementos de qualquer [vetor](#) são armazenados em bytes consecutivos na memória do computador. Se cada elemento do vetor ocupa k bytes, a diferença entre os endereços de dois elementos consecutivos é k . Mas o [compilador](#) ajusta os detalhes internos de modo a criar a ilusão de que a diferença entre os endereços de dois elementos consecutivos é 1, qualquer que seja o valor de k . Por exemplo, depois da declaração

```
int *v;
v = malloc (100 * sizeof (int));
```

o endereço do primeiro elemento do vetor é v, o endereço do segundo elemento é v+1, o endereço do terceiro elemento é v+2, etc. Se i é uma variável do tipo int então

$v + i$

é o endereço do $(i+1)$ -ésimo elemento do vetor. As expressões $v + i$ e $\&v[i]$ têm exatamente o mesmo valor e portanto as atribuições

```
*(v+i) = 789;
v[i] = 789;
```

têm o mesmo efeito. Analogamente, qualquer dos dois fragmentos de código abaixo pode ser usado para preencher o vetor v:

```
for (i = 0; i < 100; ++i) scanf ("%d", &v[i]);
for (i = 0; i < 100; ++i) scanf ("%d", v + i);
```

Todas essas considerações também valem se o vetor for alocado estaticamente (ou seja, antes

que o programa comece a ser executado) por uma declaração como

```
int v[100];
```

mas nesse caso `v` é uma espécie de “ponteiro constante”, cujo valor não pode ser alterado.

Exercícios 4

1. Suponha que os elementos de um vetor `v` são do tipo `int` e cada `int` ocupa 4 bytes no seu computador. Se o endereço de `v[0]` é 55000, qual o valor da expressão `v + 3`?
2. Suponha que `i` é uma variável inteira e `v` um vetor de inteiros. Descreva, em português, a sequência de operações que deve ser executada para calcular o valor da expressão `&v[i + 9]`.
3. Suponha que `v` é um vetor. Descreva a diferença conceitual entre as expressões `v[3]` e `v + 3`.
4. O que faz a seguinte função?

```
void imprime (char *v, int n) {  
    char *c;  
    for (c = v; c < v + n; c++)  
        printf ("%c", *c);  
}
```

5. O programa abaixo produziu a seguinte resposta, que achei surpreendente:

```
x: 111  
v[0]: 999
```

Os valores de `x` e `v[0]` não deveriam ser iguais?

```
void func1 (int x) {  
    x = 9 * x;  
}  
  
void func2 (int v[]) {  
    v[0] = 9 * v[0];  
}  
  
int main (void) {  
    int x, v[2];  
    x = 111;  
    func1 (x); printf ("x: %d\n", x);  
    v[0] = 111;  
    func2 (v); printf ("v[0]: %d\n", v[0]);  
    return EXIT_SUCCESS;  
}
```

6. O seguinte fragmento de código pretende decidir se “abacate” vem antes ou depois de “uva” no dicionário. (Veja a seção [Cadeias constantes](#) do capítulo *Strings*.) O que está errado?

```
char *a, *b;  
a = "abacate"; b = "uva";  
if (a < b)  
    printf ("%s vem antes de %s\n", a, b);  
else  
    printf ("%s vem depois de %s\n", a, b);
```

Perguntas e respostas

- PERGUNTA: É verdade que devemos atribuir um valor a um ponteiro tão logo o ponteiro é declarado?

RESPOSTA: Há quem recomende inicializar todos os ponteiros imediatamente, ou seja, escrever `int *p = NULL;` em vez de simplesmente `int *p;;`. Isso poderia ajudar a encontrar eventuais imperfeições no seu programa e poderia [proteger contra a ação de hackers](#). Este sítio não segue essa recomendação para não cansar o leitor com detalhes repetitivos. (Além disso, é deselegante atribuir um valor a uma variável sem que isso seja logicamente necessário...)

Alocação dinâmica de memória



As declarações abaixo alocam espaço na memória para algumas variáveis. A alocação é *estática* (nada a ver com a palavra-chave `static`), ou seja, acontece antes que o programa comece a ser executado:

```
char c;
int i;
int v[10];
```

Em muitas aplicações, a quantidade de memória a alocar só se torna conhecida *durante a execução* do programa. Para lidar com essa situação é preciso recorrer à alocação *dinâmica* de memória. A alocação dinâmica é administrada pelas funções `malloc`, `realloc` e `free`, que estão na biblioteca `stdlib`. Para usar essa biblioteca, inclua a correspondente [interface](#) no seu programa:

```
#include <stdlib.h>
```

A função malloc

A função `malloc` (o nome é uma abreviatura de *memory allocation*) aloca espaço para um bloco de [bytes](#) consecutivos na memória RAM (= *random access memory*) do computador e devolve o [endereço](#) desse bloco. O número de bytes é especificado no argumento da função. No seguinte fragmento de código, `malloc` aloca 1 byte:

```
char *ptr;
ptr = malloc (1);
scanf ("%c", ptr);
```

O endereço devolvido por `malloc` é do tipo genérico `void *`. O programador armazena esse endereço num [ponteiro](#) de tipo apropriado. No exemplo acima, o endereço é armazenado no ponteiro `ptr`, que é do tipo ponteiro-para-char. A transformação do ponteiro genérico em ponteiro-para-char é automática; não é necessário escrever `ptr = (char *) malloc (1);`.

A fim de alocar espaço para um objeto que precisa de mais que 1 byte, convém usar o operador [sizeof](#), que diz quantos bytes o objeto em questão tem:

```
typedef struct {
    int dia, mes, ano;
} data;
data *d;
d = malloc (sizeof (data));
d->dia = 31; d->mes = 12; d->ano = 2016;
```

Note que `sizeof` não é uma função mas um operador, tal como `return`, por exemplo. Os

parênteses na expressão `sizeof (data)` são necessários porque `data` é um tipo-de-dados (os parênteses são análogos aos do [casting](#)). O operador `sizeof` também pode ser aplicado diretamente a uma variável: se `var` é uma variável então `sizeof var` é o número de bytes ocupado por `var`. Poderíamos ter escrito `d = malloc (sizeof *d)` no exemplo acima.

Exercícios 1

1. Escreva uma função que receba um byte `c` (que pode representar um [caractere ASCII](#), por exemplo) e transforme-o em uma [string](#), ou seja, devolva uma string de comprimento 1 tendo `c` como único elemento.
2. Discuta, passo a passo, o efeito do seguinte fragmento de código:

```
int *v;
v = malloc (10 * sizeof (int));
```

3. Discuta o efeito do seguinte fragmento de código:

```
x = malloc (10 * sizeof *x);
```

A função free

As variáveis alocadas [estaticamente](#) dentro de uma função, também conhecidas como variáveis *automáticas* ou *locais*, desaparecem assim que a execução da função termina. Já as variáveis alocadas dinamicamente continuam a existir mesmo depois que a execução da função termina. Se for necessário liberar a memória ocupada por essas variáveis, é preciso recorrer à função `free`.

A função `free` desaloca a porção de memória alocada por `malloc`. A instrução `free (ptr)` avisa ao sistema que o bloco de bytes apontado por `ptr` está disponível para reciclagem. A próxima invocação de `malloc` poderá tomar posse desses bytes.

Não aplique a função `free` a uma *parte* de um bloco de bytes alocado por `malloc` (ou [realloc](#)). Aplique `free` apenas ao bloco todo.

Exercícios 2

1. O que há de errado com o seguinte fragmento de código?

```
int *v;
v = malloc (100 * sizeof (int));
v[0] = 999;
free (v+1);
```

2. A seguinte função promete devolver um vetor com os 3 primeiros números primos maiores que 1000. Onde está o erro?

```
int *primos (void) {
    int v[3];
    v[0] = 1009; v[1] = 1013; v[2] = 1019;
    return v; }
```

3. A seguinte função promete acrescentar uma [célula-cabeça](#) à [lista encadeada](#) `lst` e devolver o endereço da nova lista. Onde está o erro?

```
celula *acrescentaCabeca (celula *lst) {
    celula cabeca;
    cabeca.prox = lst;
    return &cabeca; }
```

4. Discuta, passo a passo, o efeito do seguinte fragmento de código:

```

int *p, *q;
p = malloc (sizeof (int));
*p = 123;
q = malloc (sizeof (int));
*q = *p;
q = p;
free (p);
free (q); // má ideia...
q = NULL; // boa ideia

```

Vetores e matrizes

Eis como um [vetor](#) (= *array*) com n elementos inteiros pode ser alocado (e depois desalocado) durante a execução de um programa:

```

int *v;
int n;
scanf ("%d", &n);
v = malloc (n * sizeof (int));
for (int i = 0; i < n; ++i)
    scanf ("%d", &v[i]);
.
.
.
free (v);

```

(A propósito, veja a [observação sobre endereços e vetores](#) no capítulo *Endereços e ponteiros*.) Do ponto de vista conceitual, mas apenas desse ponto de vista, a instrução

```
v = malloc (100 * sizeof (int));
```

tem efeito análogo ao da alocação estática

```
int v[100];
```

Matrizes. Matrizes bidimensionais são implementadas como vetores de vetores. Uma matriz com m linhas e n colunas é um vetor de m elementos cada um dos quais é um vetor de n elementos. O seguinte fragmento de código faz a alocação dinâmica de uma tal matriz:

```

int **M;
M = malloc (m * sizeof (int *));
for (int i = 0; i < m; ++i)
    M[i] = malloc (n * sizeof (int));

```

Portanto, $M[i][j]$ é o elemento de M que está no cruzamento da linha i com a coluna j .

Exercícios 3

1. Escreva uma função que desalogue a matriz M alocada acima. Quais devem ser os parâmetros da função?

Redimensionamento e a função `realloc`

Às vezes é necessário alterar, durante a execução do programa, o tamanho de um bloco de bytes que foi alocado por `malloc`. Isso acontece, por exemplo, durante a leitura de um [arquivo](#) que se revela maior que o esperado. Nesse caso, podemos recorrer à função `realloc` para *redimensionar* o bloco de bytes.

A [função `realloc`](#) recebe o endereço de um bloco previamente alocado por `malloc` (ou por

`realloc`) e o número de bytes que o bloco redimensionado deve ter. A função aloca o novo bloco, copia para ele o conteúdo do bloco original, e devolve o endereço do novo bloco.

Se o novo bloco for uma extensão do bloco original, seu endereço é o mesmo do original (e o conteúdo do original não precisa ser copiado para o novo). Caso contrário, `realloc` copia o conteúdo do bloco original para o novo e libera o bloco original (invocando `free`). A propósito, o tamanho do novo bloco pode ser *menor* que o do bloco original.

Suponha, por exemplo, que alocamos um vetor de 1000 inteiros e depois decidimos que precisamos de duas vezes mais espaço. Veja um caso concreto:

```
int *v;
v = malloc (1000 * sizeof (int));
for (int i = 0; i < 990; i++)
    scanf ("%d", &v[i]);
v = realloc (v, 2000 * sizeof (int));
for (int i = 990; i < 2000; i++)
    scanf ("%d", &v[i]);
```

Nesse exemplo, poderíamos usar a seguinte implementação *ad hoc* de `realloc`:

```
int *realloc (int *v, unsigned int N) {
    int *novo = malloc (N);
    for (int i = 0; i < 1000; i++)
        novo[i] = v[i];
    free (v);
    return novo;
}
```

É claro que a implementação de `realloc` na biblioteca `stdlib` é mais geral e mais eficiente.

Exercícios 4

1. Suponha dado um [arquivo de texto](#) que contém uma sequência de números inteiros. O comprimento da sequência é desconhecido. Escreva uma função que imprima esses números em ordem inversa (o último, depois o penúltimo, etc.). É claro que você terá que ler todos os números e armazená-los na memória. A dificuldade está em alocar espaço para uma quantidade de números que só será conhecida quando chegarmos ao fim do arquivo.

A memória é finita

Se a memória do computador já estiver toda ocupada, `malloc` não consegue alocar mais espaço e devolve `NULL`. Convém verificar essa possibilidade antes de prosseguir:

```
ptr = malloc (sizeof (data));
if (ptr == NULL) {
    printf ("Socorro! malloc devolveu NULL!\n");
    exit (EXIT_FAILURE);
}
```

A digitação frequente e repetida desse teste é cansativa. Por isso, usaremos, neste sítio, a seguinte função-embalagem (= *wrapper function*) de `malloc`:

```
void *mallocc (size_t nbytes) {
    void *ptr;
    ptr = malloc (nbytes);
    if (ptr == NULL) {
        printf ("Socorro! malloc devolveu NULL!\n");
        exit (EXIT_FAILURE);
    }
```

```
    return ptr;
}
```

O parâmetro de `malloc` é do tipo `size_t` (abbreviatura de *size data type*); em muitos computadores, `size_t` é [o mesmo que `unsigned int`](#).

Da mesma forma, podemos preparar uma função-embalagem `reallocc` para cuidar das situações em que `realloc` devolve `NULL`.

Perguntas e respostas

- PERGUNTA: É verdade que não convém alocar blocos de poucos bytes, ou seja, invocar `malloc` com argumento muito pequeno?

RESPOSTA: Cada invocação de `malloc` aloca um bloco de bytes maior que o solicitado; os bytes adicionais são usados para guardar informações administrativas sobre o bloco (essas informações permitem que o bloco seja corretamente desalocado, mais tarde, pela função `free`). Assim, não é eficiente alocar blocos muito pequenos repetidamente; é melhor alocar um bloco grande e dele retirar pequenas porções na medida do necessário. A boa notícia é que o programador não precisa fazer isso pessoalmente pois `malloc` implementa essa política “por baixo do pano” sem que o programador perceba.

- PERGUNTA: Posso alocar um vetor estaticamente com número não-constante de elementos? Por exemplo, posso dizer

```
int v[n];
```

se o valor de `n` só se torna conhecido durante a execução do programa?

RESPOSTA: Não é uma boa ideia.

- PERGUNTA: É verdade que devemos atribuir `NULL` a cada ponteiro que se tornou inútil ou desnecessário?

RESPOSTA: Sim. Convém não deixar ponteiros “soltos” (= *dangling pointers*) no seu programa, pois isso pode dificultar a depuração do programa e [pode ser explorado por hackers para atacar o seu computador](#). Portanto, depois de cada `free` (`ptr`), atribua `NULL` a `ptr`:

```
free (ptr);
ptr = NULL;
```

(Atribuir um valor a uma variável que não será mais usada é decididamente deselegante, mas não há como lidar com hackers de maneira elegante...) Para não cansar o leitor com detalhes repetitivos, este sítio ignora essa recomendação de segurança.

- PERGUNTA: É verdade que deveríamos usar a função `calloc` no lugar de `malloc`?

RESPOSTA: Talvez. A chamada `calloc` (`a, b`) aloca um bloco de $a \cdot b$ bytes e *atribui valor zero a todos esses bytes*. Veja o artigo [Why does calloc exist?](#) de Nathaniel J. Smith (2016).

[Julia's drawings: Memory allocation](#).

Veja [Memory Layout of C Programs](#) em [GeeksforGeeks](#).

[Valgrind](#): ferramenta para encontrar [vazamentos de memória](#) (= *memory leaks*) e *segmentation faults* no seu programa.

Atualizado em 2018-05-29

<https://www.ime.usp.br/~pf/algoritmos/>

Paulo Feofiloff

[DCC-IME-USP](#)



Leitura de linhas de texto

Para tornar a leitura de um [arquivo](#) mais confortável, convém adotar um processo iterativo em que cada iteração tem dois passos:

1. o primeiro passo lê uma linha do arquivo e converte a linha em uma [string](#);
2. o segundo passo extrai da string as informações desejadas.

O segundo passo pode ser executado pela função [scanf](#) da biblioteca stdio (veja, por exemplo, a função [getInteger](#) [abaixo](#)). O primeiro passo pode ser executado pela função [fgets](#) da biblioteca stdio. Este capítulo oferece uma alternativa caseira mais poderosa que fgets.

Arquivos de texto

Grande parte de nossos dados é armazenada em [arquivos de texto](#), ou seja, arquivos cujos bytes representam [caracteres](#). Suporemos que todos os nossos arquivos usam codificação UTF-8 e portanto cada caractere é representado por 1, 2, 3, ou 4 bytes consecutivos. (Convém lembrar que [certos bytes não fazem parte do código de nenhum caractere](#); é o caso dos bytes no intervalo 248 .. 255, por exemplo.) Se cada caractere do arquivo é representado por apenas 1 byte — ou seja, se todos os caracteres são [ASCII](#) — temos um *arquivo ASCII*.

Em geral, um [arquivo de texto](#) tem várias *linhas*, sendo o fim de cada linha sinalizado pelo [byte \n](#). Diremos que cada linha do arquivo é uma *linha de texto*. (A propósito, o último byte de todo arquivo de texto bem educado [deve ser um \n](#); além disso, nenhuma linha deve ter um espaço antes de \n.)

Leitura de uma linha de texto

[Eric Roberts](#) escreveu uma biblioteca [simpio](#) (veja a interface `simpio.h` e a implementação `simpio.c`) cujas funções tornam confortável a leitura de uma linha de texto. A principal função da biblioteca é `ReadLine`, que substitui, com vantagem, a função `fgets` (mas talvez não seja tão rápida quanto aquela).

Segue uma versão ligeiramente modificada da função `ReadLine` de Roberts. A função lê uma linha de comprimento arbitrário de um [arquivo de texto](#). Para dar conta da tarefa, a função faz redimensionamentos periódicos do vetor que recebe a linha de texto.

```
#include <stdlib.h>
#include <string.h>

// A função readLine lê uma linha (a partir da posição
// corrente) do arquivo de texto infile e devolve uma
// string com o mesmo conteúdo da linha. O byte \n que
// sinaliza o fim da linha não é armazenado como parte
// da string. A função devolve NULL se a posição
// corrente do arquivo estiver no fim do arquivo. Uso
// típico da função:
//           s = readLine (infile);
// (Esta função é uma adaptação da função ReadLine da
```

```

// biblioteca simples de Eric Roberts.)

char *readLine (FILE *infile)
{
    int n = 0, size = 128, ch;
    char *line;
    line = malloc (size + 1);
    while ((ch = getc (infile)) != '\n' && ch != EOF) {
        if (n == size) {
            size *= 2;
            line = realloc (line, size + 1);
        }
        line[n++] = ch;
    }
    if (n == 0 && ch == EOF) {
        free (line);
        return NULL;
    }
    line[n] = '\0';
    line = realloc (line, n + 1);
    return line;
}

```

A função transfere bytes do arquivo `infile` para o vetor `line` alocado dinamicamente. Toda vez que o vetor fica cheio, seu tamanho é dobrado.

A função `readLine` supõe, implicitamente, que a linha lida não contém [bytes nulos](#) (uma hipótese perfeitamente razoável).

Exercícios 1

1. Como `readLine` se comporta se porventura encontrar um byte nulo (`\0`) antes de `\n`?
2. A documentação de `readLine` não explica o que acontece [se o arquivo `infile` não tiver um `\n`](#) (entre a posição corrente e o fim do arquivo). Preencha essa lacuna.
3. Analise a seguinte variante de `readLine`, mais próxima da `ReadLine` de Roberts. A função [`strncpy`](#) que aparece no código é uma variante da função [`strcpy`](#): com argumentos `nline`, `line`, e `n`, ela copia os `n` primeiros bytes de `line` para `nline`.

```

int n = 0, size = 128, ch;
char *line, *nline;
line = malloc (size + 1);
while ((ch = getc (infile)) != '\n' && ch != EOF) {
    if (n == size) {
        size *= 2;
        nline = malloc (size + 1);
        strncpy (nline, line, n);
        free (line);
        line = nline;
    }
    line[n++] = ch;
}
if (n == 0 && ch == EOF) {
    free (line);
    return NULL;
}
line[n] = '\0';
nline = malloc (n + 1);
strcpy (nline, line);
free (line);
return nline;

```

4. No exercício [anterior](#), o que acontece se trocarmos `strncpy (nline, line, n)` por `strcpy (nline, line)`?
5. Compare a documentação de `readLine` com a da função [`fgets`](#) da biblioteca [`stdio`](#).

Leitura de um número inteiro

A função GetInteger da [biblioteca simpio de Eric Roberts](#) usa a função ReadLine para extrair um número inteiro de uma linha de texto digitada no teclado. Segue uma versão ligeiramente modificada da função:

```
// Lê uma linha de texto do teclado, extrai dela um
// número inteiro e devolve esse inteiro. Se a linha
// tiver algum byte não branco antes ou depois do
// inteiro a função dá ao usuário a oportunidade de
// tentar de novo. Uso típico: i = getInteger ();
// (Esta função é uma adaptação da função GetInteger
// da biblioteca simpio de Eric Roberts.)

int getInteger (void) {
    while (1) {
        char *line = readLine (stdin);
        int value;
        char ch;
        switch (sscanf (line, " %d %c", &value, &ch)) {
            case 1:
                free (line);
                return value;
            case 2:
                printf ("Byte inesperado: '%c'\n", ch);
            default:
                printf ("Tente de novo\n");
        }
        free (line);
    }
}
```

A função lê uma linha de texto e em seguida usa [sscanf](#) para extrair dela um inteiro. Ler uma linha completa é essencial para que a função possa se recobrar de um eventual erro de digitação do usuário; caso contrário, os bytes depois da ocorrência do erro permaneceriam no buffer de entrada e confundiriam as operações de entrada subsequentes.

A função sscanf da biblioteca [stdio](#) é como a função [fscanf](#), exceto que opera sobre uma string em lugar de um arquivo. A função recebe uma string line e tenta extrair dela os objetos especificados pelo formato dado no segundo argumento. A função devolve o número de objetos que conseguiu extrair com sucesso de line.

Exercícios 2

1. Estude as funções da biblioteca da [biblioteca simpio](#) de Eric Roberts.
2. Estude a documentação da função sscanf da biblioteca padrão stdio.

Números aleatórios



Sequências de números aleatórios (ou seja, imprevisíveis) são úteis em muitas aplicações. São úteis, em particular, para gerar dados de teste de programas. Números verdadeiramente aleatórios são muito difíceis de obter (veja random.org); por isso, devemos nos contentar com números *pseudoaleatórios*, gerados por algoritmos. Para simplificar a linguagem, omitiremos o “pseudo” no que segue.

A função rand

A função `rand` (o nome é uma abreviatura de *random*) da biblioteca `stdlib` gera números inteiros aleatórios. Cada invocação da função produz um inteiro aleatório no intervalo fechado `0 .. RAND_MAX`. A macro `RAND_MAX` está definida na interface `stdlib.h` e é menor ou igual a `INT_MAX`. (Para aplicações pouco exigentes, podemos supor que os números gerados por `rand` são mais ou menos uniformemente distribuídos no intervalo `0 .. RAND_MAX`, ou seja, que cada número do intervalo tem mais ou menos a mesma probabilidade de ser gerado.)

Exercícios 1

1. [Roberts] O seguinte programa promete simular uma jogada de um dado de 6 faces. Qual o defeito do programa?

```
int RolaDado (void) {
    int r = rand ();
    if (r < RAND_MAX/6) return 1;
    else if (r < 2 * RAND_MAX/6) return 2;
    else if (r < 3 * RAND_MAX/6) return 3;
    else if (r < 4 * RAND_MAX/6) return 4;
    else if (r < 5 * RAND_MAX/6) return 5;
    else return 6;
}
```

2. [Roberts] O seguinte programa promete simular uma jogada de um dado de 6 faces. Qual o defeito do programa?

```
int RolaDado (void) {
    int r = rand ();
    if (r < RAND_MAX/6) return 1;
    else if (r < RAND_MAX/6 * 2) return 2;
    else if (r < RAND_MAX/6 * 3) return 3;
```

```

    else if (r < RAND_MAX/6 * 4) return 4;
    else if (r < RAND_MAX/6 * 5) return 5;
    else return 6;
}

```

3. [Roberts] SUTIL. O seguinte programa promete simular uma jogada de uma moeda. Qual o defeito do programa?

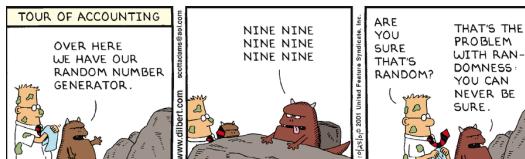
```

char *RolaMoeda (void) {
    int r;
    r = rand () % 2;
    if (r == 1) return "cara";
    else return "coroa";
}

```

Inteiros aleatórios

Como obter um número inteiro aleatório num dado intervalo [0..N-1](#)? A primeira ideia é usar a expressão `rand() % N` (que dá o resto da divisão de `rand()` por `N`). Essa ideia seria razoável se `rand` produzisse números verdadeiramente aleatórios. Como os números produzidos por `rand` são apenas *pseudoaleatórios*, os *últimos dígitos* de cada número podem não ser aleatórios, e assim o resto da divisão por `N` pode não ser aleatório (poderia ser sempre ímpar, por exemplo).



[Eric Roberts](#) escreveu uma pequena [biblioteca random](#) que procura contornar a dificuldade apontada no parágrafo anterior e fornecer inteiros razoavelmente aleatórios no intervalo `low..high`. Eis uma das funções da biblioteca:

```

// A função randomInteger devolve um inteiro
// aleatório entre low e high inclusive,
// ou seja, no intervalo fechado low..high.
// Vamos supor que low <= high e que
// high - low <= RAND_MAX. (O código foi copiado
// da biblioteca random de Eric Roberts.)

int randomInteger (int low, int high)
{
    double d;
    d = (double) rand () / ((double) RAND_MAX + 1);
    int k = d * (high - low + 1);
    return low + k;
}

```

Primeiro, a função transforma o inteiro produzido por `rand` em um [número real](#) `d` no intervalo semi-aberto $[0, 1]$. Depois, transforma `d` em um número inteiro `k` no intervalo $0..high-low$. Finalmente, transforma `k` em um inteiro no intervalo `low..high`.

Para aplicações pouco exigentes, podemos supor que os números produzidos por `randomInteger` são distribuídos mais ou menos uniformemente no intervalo `low..high`, especialmente se `high - low` for muito menor que `RAND_MAX`.

Exercícios 2

- No código de `randomInteger`, por que não escrever `(RAND_MAX+1)` no lugar de `((double)RAND_MAX+1)`? Por que não escrever `RAND_MAX` no lugar de `((double)RAND_MAX+1)`?

2. Analise o código para verificar que o inteiro produzido por `randomInteger (0, RAND_MAX)` é igual ao produzido por `rand ()`.
3. Prove que o número `k` que `randomInteger` devolve é tal que `low ≤ k ≤ high`. Prove também que `randomInteger` devolve `low` quando `rand` gera 0. Prove ainda que `randomInteger` devolve `high` quando `rand` gera `RAND_MAX`.
4. Use a função `randomInteger` para simular o rolar de um dado de 6 faces.

Semente

A função `rand` tem uma memória interna que armazena o inteiro, digamos `r`, produzido pela execução anterior da função. A cada nova execução, a função usa `r` para calcular um novo inteiro aleatório. (O inteiro calculado passa a ser o novo valor de `r`.)

Onde tudo isso começa? O inteiro `r` que corresponde à *primeira* invocação de `rand` é conhecido como *semente* (= *seed*). Dada a semente, a sequência de inteiros produzida por `rand` está *completamente determinada*.

O programador pode especificar a semente por meio da função `srand` da biblioteca `stdlib`, que recebe a semente (um `unsigned int`) como argumento. Se o programador não especificar a semente, o sistema adota o valor 0. A seguinte função da biblioteca de Roberts usa o relógio para especificar a semente:

```
#include <time.h>

// A função randomize inicializa o gerador
// de números aleatórios de modo que os
// resultados das invocações de randomInteger
// sejam imprevisíveis.

void randomize (void)
{
    srand (time (NULL));
}
```

Exercícios 3

1. Escreva um programa que receba, pela [linha de comando](#), dois inteiros *positivos* `n` e `s` e imprima os números produzidos por `n` invocações da função `rand` com semente `s`.
2. Escreva um programa que receba, pela [linha de comando](#), os inteiros *positivos* `n`, `l`, `h` e `s` e imprima os números produzidos por `n` invocações da função `randomInteger` com argumentos `l` e `h` e semente `s`.

Permutações aleatórias

Uma *permutação* de um vetor é qualquer rearranjo dos elementos do vetor; cada elemento do vetor muda de posição ou permanece onde está.

A seguinte função faz uma permutação aleatória de um vetor `v[0..n-1]`. Se os elementos do vetor forem distintos entre si, todas as $n!$ permutações são igualmente prováveis.

```
// Faz uma permutação aleatória dos elementos
// do vetor v[0..n-1].

void permutacaoAleatoria (int v[], int n) {
```

```

int r, k, t;
for (k = n-1; k > 0; k--) {
    r = randomInteger (0, k); // 0 <= r <= k
    t = v[k], v[k] = v[r], v[r] = t;
}

```

Veja a [animação](#) produzida por [Mike Bostock](#). (Dica de Yoshiharu Kohayakawa.)

Exercícios 4

1. Escreva um programa que receba, pela [linha de comando](#), os inteiros [positivos](#) m e n e imprima m permutações aleatórias sucessivas do vetor $v[0 .. n-1]$ definido por $v[i] = i$.
2. Escreva um programa que receba, pela [linha de comando](#), os inteiros [positivos](#) $m, n, v_0, \dots, v_{n-1}$ e imprima m permutações aleatórias sucessivas do vetor (v_0, \dots, v_{n-1}) .

Veja verbetes [Random number generation](#) e [Mersenne twister](#) na Wikipedia.

Atualizado em 2018-07-26

<https://www.ime.usp.br/~pf/algoritmos/>

Paulo Feofiloff

[DCC-IME-USP](#)



Precedência entre operadores em C

Na tabela abaixo, os operadores estão em ordem decrescente de prioridade: os operadores da primeira linha são executados em primeiro lugar e os operadores da última são executados por último.

Os operadores da segunda linha são *unários* (um só operando); todos os demais são *binários* (dois operandos). A coluna direita indica a regra de associação para os operadores da linha: e-d significa “da esquerda para a direita” e d-e significa “da direita para a esquerda”.

()	[]	-> .	e-d
-	++	--	d-e
!	&	*	d-e
~	(type)	sizeof	d-e
*	/	%	e-d
+	-		e-d
<<	>>		e-d
<	<=	>=	e-d
>			e-d
==	!=		e-d
&			e-d
^			e-d
			e-d
&&			e-d
			e-d
?	:		d-e
=	op=		d-e
,			e-d

Exemplos:

expressão	interpretação
&x[i]	&(x[i])
*p.dia	*(p.dia)
a[i].b[j]	((a[i]).b)[j]
h->e->d	(h->e)->d
&h->e	&(h->e)
*x++	*(x++)

Tabela ASCII

A tabela [ASCII](#) (American Standard Code for Information Interchange) dá os caracteres (coluna *c* da tabela) [associados aos números 0 a 127](#) ([números Unicode](#) U+0000 a U+007F). O conjunto de todos os caracteres da tabela constitui o *alfabeto ASCII*.

	binário	Unicode	<i>c</i>	observação
0	00000000	U+0000	\0	<i>byte nulo</i>
1	00000001	U+0001		<i>não usamos</i>
2	00000010	U+0002		<i>não usamos</i>
3	00000011	U+0003		<i>não usamos</i>
4	00000100	U+0004		<i>não usamos</i>
5	00000101	U+0005		<i>não usamos</i>
6	00000110	U+0006		<i>não usamos</i>
7	00000111	U+0007	\a	<i>apito</i>
8	00001000	U+0008	\b	<i>backspace</i>
9	00001001	U+0009	\t	<i>tabulação</i>
10	00001010	U+000A	\n	<i>fim de linha</i>
11	00001011	U+000B	\v	<i>tab vertical</i>
12	00001100	U+000C	\f	<i>fim de página</i>
13	00001101	U+000D	\r	<i>carriage return</i>
14	00001110	U+000E		<i>não usamos</i>
15	00001111	U+000F		<i>não usamos</i>
16	00010000	U+0010		<i>não usamos</i>
17	00010001	U+0011		<i>não usamos</i>
18	00010010	U+0012		<i>não usamos</i>
19	00010011	U+0013		<i>não usamos</i>
20	00010100	U+0014		<i>não usamos</i>
21	00010101	U+0015		<i>não usamos</i>
22	00010110	U+0016		<i>não usamos</i>
23	00010111	U+0017		<i>não usamos</i>
24	00011000	U+0018		<i>não usamos</i>
25	00011001	U+0019		<i>não usamos</i>
26	00011010	U+001A		<i>não usamos</i>
27	00011011	U+001B		<i>não usamos</i>
28	00011100	U+001C		<i>não usamos</i>
29	00011101	U+001D		<i>não usamos</i>
30	00011110	U+001E		<i>não usamos</i>
31	00011111	U+001F		<i>não usamos</i>
32	00100000	U+0020		<i>espaço</i>
33	00100001	U+0021	!	
34	00100010	U+0022	"	<i>aspas</i>
35	00100011	U+0023	#	
36	00100100	U+0024	\$	
37	00100101	U+0025	%	
38	00100110	U+0026	&	
39	00100111	U+0027	'	<i>apóstrofe</i>

40	00101000	U+0028	(
41	00101001	U+0029)
42	00101010	U+002A	*
43	00101011	U+002B	+
44	00101100	U+002C	,
45	00101101	U+002D	- <i>vírgula</i>
46	00101110	U+002E	.
			<i>ponto</i>
47	00101111	U+002F	/
48	00110000	U+0030	0
49	00110001	U+0031	1
50	00110010	U+0032	2
51	00110011	U+0033	3
52	00110100	U+0034	4
53	00110101	U+0035	5
54	00110110	U+0036	6
55	00110111	U+0037	7
56	00111000	U+0038	8
57	00111001	U+0039	9
58	00111010	U+003A	:
59	00111011	U+003B	;
60	00111100	U+003C	<
61	00111101	U+003D	=
62	00111110	U+003E	>
63	00111111	U+003F	?
64	01000000	U+0040	@
65	01000001	U+0041	A
66	01000010	U+0042	B
67	01000011	U+0043	C
68	01000100	U+0044	D
69	01000101	U+0045	E
70	01000110	U+0046	F
71	01000111	U+0047	G
72	01001000	U+0048	H
73	01001001	U+0049	I
74	01001010	U+004A	J
75	01001011	U+004B	K
76	01001100	U+004C	L
77	01001101	U+004D	M
78	01001110	U+004E	N
79	01001111	U+004F	O
80	01010000	U+0050	P
81	01010001	U+0051	Q
82	01010010	U+0052	R
83	01010011	U+0053	S
84	01010100	U+0054	T
85	01010101	U+0055	U
86	01010110	U+0056	V
87	01010111	U+0057	W
88	01011000	U+0058	X
89	01011001	U+0059	Y
90	01011010	U+005A	Z
91	01011011	U+005B	[
92	01011100	U+005C	\
93	01011101	U+005D]

94	01011110	U+005E	^	
95	01011111	U+005F	-	<i>underscore</i>
96	01100000	U+0060	`	<i>apóstrofe esq.</i>
97	01100001	U+0061	a	
98	01100010	U+0062	b	
99	01100011	U+0063	c	
100	01100100	U+0064	d	
101	01100101	U+0065	e	
102	01100110	U+0066	f	
103	01100111	U+0067	g	
104	01101000	U+0068	h	
105	01101001	U+0069	i	
106	01101010	U+006A	j	
107	01101011	U+006B	k	
108	01101100	U+006C	l	
109	01101101	U+006D	m	
110	01101110	U+006E	n	
111	01101111	U+006F	o	
112	01110000	U+0070	p	
113	01110001	U+0071	q	
114	01110010	U+0072	r	
115	01110011	U+0073	s	
116	01110100	U+0074	t	
117	01110101	U+0075	u	
118	01110110	U+0076	v	
119	01110111	U+0077	w	
120	01111000	U+0078	x	
121	01111001	U+0079	y	
122	01111010	U+007A	z	
123	01111011	U+007B	{	
124	01111100	U+007C		
125	01111101	U+007D	}	
126	01111110	U+007E	~	
127	01111111	U+007F		<i>delete</i>

binário Unicode c observação

(O prolongamento [ISO-LATIN-1](#) dessa tabela caiu em desuso com a popularização do [código UTF-8](#).)

Veja a [Caracteres e a tabela ASCII](#) na página *Bytes, números e caracteres*

Atualizado em 2017-05-16

<https://www.ime.usp.br/~pf/algoritmos/>

Paulo Feofiloff

[DCC-IME-USP](#)



Unicode e UTF-8

!	?	@	+	-	*	/	=	<	>
0	1	2	3	4	5	6	7	8	9
A	E	I	O	U	C	...			
Á	Â	Ã	É	Ó	Ô	Õ	Ú	Ç	...
≡	≠	≤	≥						
Γ	Δ	Π	Σ	Ω					
									⋮

Esta página faz uma rápida introdução aos conceitos de [caractere](#), [Unicode](#), [esquema de codificação](#), e [UTF-8](#). Para uma introdução mais completa ao assunto veja os artigos indicados no fim da página.

Caracteres

Um *caractere* é um símbolo tipográfico usado para escrever texto em alguma língua. (Embora imperfeita, essa definição é suficiente para nossas necessidades.) Eis alguns exemplos de caracteres:

! " - 9 A B a b ~ À ã ç é ÿ Σ α — “

O número de caracteres usados pelas diferentes línguas do mundo é muito grande. O português usa apenas 127 caracteres e o inglês fica satisfeito com 94 desses. Mas não podemos nos limitar a essas duas línguas porque no mundo globalizado de hoje estamos expostos a muitas outras línguas, às vezes várias numa mesma sentença.

Para começar a organizar essa [Babel](#), é preciso dar *nomes* a todos os caracteres. O [consórcio Unicode](#) atribuiu *nomes numéricos* (conhecidos como [code points](#)) a mais de 1 milhão de caracteres. Segue uma minúscula amostra da lista de caracteres e seus números:

número	caractere
Unicode	
33	!
34	"
45	-
57	9
65	A
66	B
97	a
98	b
126	~
192	À
227	ã
231	ç
233	é
255	ÿ
931	Σ
945	α
8212	—

Nessa amostra, os nomes numéricos dos caracteres estão escritos em notação decimal. Em geral, entretanto, esses números são escritos em notação [hexadecimal](#). Além disso, é usual acrescentar o prefixo “U+” a cada número:

número Unicode	caractere
U+0021	!
U+0022	"
U+002D	-
U+0039	9
U+0041	A
U+0042	B
U+0061	a
U+0062	b
U+007E	~
U+00C0	À
U+00E3	ã
U+00E7	ç
U+00E9	é
U+00FF	ÿ
U+03A3	Σ
U+03B1	α
U+2014	—
U+201C	“

A lista completa de caracteres e seus números Unicode pode ser vista na página [List of Unicode characters](#) da Wikipedia ou na página [Unicode / Character reference](#) do Wikibooks.

O conjunto de todos os caracteres da lista Unicode pode ser chamado *alfabeto Unicode* e cada caractere desse alfabeto pode ser chamado *caractere Unicode*. (Se a pretensão do projeto Unicode for justificada, *todos* os caracteres de *todas* as línguas do mundo são caracteres Unicode.)

É cômodo usar atalhos verbais óbvios ao falar de caracteres. Por exemplo, em vez de dizer “o caractere A” podemos dizer

- “o caractere cujo nome é U+0041”, ou
- “o caractere cujo número é U+0041” ou, mais simplesmente,
- “o caractere U+0041”.

Caracteres ASCII

Os primeiros 128 caracteres da lista Unicode são os mais usados. Esse conjunto de caracteres vai de U+0000 a U+007F e é conhecido como [alfabeto ASCII](#). Os elementos desse alfabeto serão chamados *caracteres ASCII*. O alfabeto ASCII contém letras, dígitos decimais, sinais de pontuação, e alguns caracteres especiais. A lista dos 128 caracteres ASCII e seus números Unicode está registrada na [Tabela ASCII](#).

Infelizmente o alfabeto ASCII não é suficiente para escrever texto em português, pois não contém letras com [sinais diacríticos](#).

Esquemas de codificação

Um *esquema de codificação* (= [character encoding](#)) é uma tabela que associa uma sequência

de [bytes](#) com cada número Unicode, e portanto com cada caractere Unicode. Em geral, omitimos “esquema” e dizemos apenas “codificação” ou “código”.

A sequência de bytes associada com um caractere é o *código* do caractere. Esse código representa o caractere na memória do computador e em arquivos digitais.

Código ASCII

O código ASCII é muito simples: o número Unicode de cada caractere é escrito em [notação binária](#). Esse código é usado apenas para o [alfabeto ASCII](#). Como o alfabeto tem apenas 128 caracteres, o código ASCII necessita de apenas 1 byte por caractere e o primeiro bit desse byte é 0. Segue uma amostra da tabela de códigos:

número Unicode	caractere	código ASCII	hexadecimal
U+0021	!	00100001	0x21
U+0022	"	00100010	0x22
U+002D	-	00101101	0x2D
U+0039	9	00100111	0x39
U+0041	A	01000001	0x41
U+0042	B	01000010	0x42
U+0061	a	01100001	0x61
U+0062	b	01100010	0x62
U+007E	~	01111110	0x7E

A última coluna traz o código ASCII escrito em notação hexadecimal.

(Por que não aproveitar *todos* os 8 bits de um byte? Com isso, poderíamos codificar 128 caracteres adicionais além dos 128 do alfabeto ASCII. O [código ISO-LATIN-1](#) faz exatamente isso, mas caiu em desuso.)

Código UTF-8

O [alfabeto Unicode](#) tem mais de 1 milhão caracteres. Portanto, o código de cada caractere precisaria de pelo menos 3 bytes se usássemos notação binária. Usar um número fixo de bytes por caractere não seria eficiente, já que 1 byte é suficiente para codificar os caracteres mais comuns. A solução é recorrer a um código *multibyte*, que emprega um número *variável* de bytes por caractere: alguns caracteres usam 1 byte, outros usam 2 bytes, e assim por diante.

O código multibyte mais usado é conhecido como [UTF-8](#). Ele associa uma sequência de 1 a 4 bytes (8 a 32 bits) com cada caractere Unicode. Os primeiros 128 caracteres usam o velho e bom [código ASCII](#) de 1 byte por caractere. Os demais caracteres têm um código mais complexo. Veja uma minúscula amostra:

número Unicode	caractere	código UTF-8	hexadecimal
U+0021	!	00100001	0x21
U+0022	"	00100010	0x22
U+002D	-	00101101	0x2D
U+0039	9	00100111	0x39
U+0041	A	01000001	0x41
U+0042	B	01000010	0x42
U+0061	a	01100001	0x61
U+0062	b	01100010	0x62

U+007E	~	01111110	0x7E
U+00C0	À	11000011 01000000	0xC380
U+00E3	ã	11000011 10100011	0xC3A3
U+00E7	ç	11000011 10100111	0xC3A7
U+00E9	é	11000011 10101001	0xC3A9
U+00FF	ÿ	11000011 10111111	0xC3BF
U+03A3	Σ	11001110 10100011	0xCEA3
U+03B1	α	11001110 10110001	0xCEB1
U+2014	—	11100010 10000000 10010100	0xE28094
U+201C	“	11100010 10000000 10011100	0xE2809C

(A última coluna traz o código UTF-8 escrito em notação hexadecimal.) A lista dos códigos UTF-8 de todos os caracteres Unicode pode ser vista na página [UTF-8 encoding table and Unicode characters](#) ou na página [Unicode / Character reference](#) do Wikibooks. Por exemplo, a cadeia de caracteres “ação” é representada em UTF-8 pela seguinte sequência de bytes:

0x61	0xC3	0xA7	0xC3	0xA3	0x6F
a	ç	ã	o		

Todas as letras com [sinais diacríticos](#) usadas em português são representados em UTF-8 por apenas 2 bytes, o primeiro dos quais é 0xC3 (195 em notação decimal).

Decodificação. Como o número de bytes por caractere não é fixo, a decodificação de uma sequência de bytes que representa um texto não é fácil. Como saber onde termina o código de um caractere e começa o código do caractere seguinte? O esquema de codificação UTF-8 foi construído de modo que [os primeiros bits do código de um caractere dizem quantos bytes o código ocupa](#). Assim, se o primeiro bit é 0, e portanto o valor do primeiro byte é menor que 128, então esse é o único byte do caractere. Se o valor do primeiro byte pertence ao intervalo 192 .. 223 então o código do caractere tem dois bytes. E assim por diante.

A linguagem de programação C não prescreve nenhum esquema de codificação específico. Mas o código mais usado é UTF-8, tanto para entrada e saída quanto para a representação interna de caracteres. O [presente sítio](#) supõe que todos os arquivos (programas e dados) do leitor usam [código UTF-8](#) (ou o subconjunto ASCII de UTF-8).

Exercícios 1

1. A seguinte sequência de bytes está em notação decimal. Que [cadeia de caracteres](#) esses bytes representam em código UTF-8?

67 195 179 100 105 103 111

2. A seguinte sequência de bytes está em notação hexadecimal. Que [cadeia de caracteres](#) esses bytes representam em código UTF-8?

0x41 0x74 0x65 0x6E 0xC3 0xA7 0xC3 0xA3 0x6F 0x21

3. Consulte a tabela [UTF-8 encoding table and Unicode characters](#) (ou outra semelhante) para verificar que todas as letras com [sinais diacríticos](#) usadas em português são representados em UTF-8 por apenas 2 bytes, o primeiro dos quais é 0xC3 (195 em notação decimal).

4. Escreva uma função que receba um arquivo que contém texto em código UTF-8 e decida se cada byte do arquivo representa um único caractere (ou seja, se o alfabeto do arquivo é [ASCII](#)).

Como o meu arquivo está codificado?

Todo [arquivo de texto](#) é uma sequência de bytes. Não há como saber, com certeza, qual o

esquema de codificação usado por um dado arquivo. Portanto, não há como saber, com segurança, que [cadeia de caracteres](#) o arquivo representa. O autor do arquivo precisa informar, fora do arquivo, qual código usou.

Há utilitários (como o [file](#), por exemplo) que examinam um arquivo e tentam adivinhar, com algum grau de confiança, o seu esquema de codificação.

Se souber qual o esquema de codificação usado por seu arquivo, você pode usar o aplicativo [iconv](#) para mudar a codificação (convertendo, por exemplo, um arquivo codificado em [ISO-LATIN-1](#) em um arquivo equivalente codificado em [UTF-8](#)).

Exercícios 2

1. Familiarize-se com os programas [od](#) e [hexdump](#). Esses utilitários imprimem a sequência de bytes de um arquivo sem convertê-los em caracteres.
2. A função [isalpha](#) da [biblioteca ctype](#) só reconhece letras do [alfabeto ASCII](#) (sem [sinais diacríticos](#), portanto). Escreva uma extensão de [isalpha](#) que reconheça letras com diacríticos. Sua função deve receber uma [string](#) que seja a [codificação UTF-8](#) de um caractere e decidir se o código representa uma letra (com sinal diacrítico ou não).

Locales

O comportamento de qualquer programa depende do *ambiente* criado pelo sistema operacional. Esse ambiente é definido pelas *variáveis de ambiente* (= *environment variables*) do sistema. As variáveis especificam a língua e o [esquema de codificação](#) padrão. Digite “`locale`” no terminal para obter o valor de todas as variáveis de ambiente:

```
~$ locale
LANG=pt_US.UTF-8
LANGUAGE=en_US:en_GB:en
LC_CTYPE=pt_US.UTF-8
LC_NUMERIC=en_US.UTF-8
LC_TIME=pt_US.UTF-8
LC_COLLATE=pt_US.UTF-8
LC_MONETARY=pt_BR.UTF-8
LC_MESSAGES=pt_BR.UTF-8
LC_MEASUREMENT=pt_BR.UTF-8
LC_ALL=
```

A variável de ambiente mais relevante para o comportamento de `printf`, `scanf`, etc. é `LC_CTYPE`. Suporemos que `LC_CTYPE` vale `pt_BR.UTF-8` ou `en_US.UTF-8` no seu sistema.

O [presente sítio](#) supõe que o leitor usa o sistema operacional Linux com *locale* `UTF-8`.

Suporemos também que todos os programas do leitor que manipulam texto invocam a [função setlocale](#) para importar a variável de ambiente `LC_CTYPE` e outras variáveis relevantes.

Você também pode mudar o valor de variáveis de ambiente temporariamente antes de invocar o seu programa. Por exemplo,

```
~$ LC_COLLATE=pt_BR.UTF-8 ./meuprograma
```

ou `LC_COLLATE=C`, ou `LC_ALL=C`, ou...

Exemplo. Considere o seguinte programa:

```
#include <locale.h>
```

```

int main (void) {
    setlocale (LC_ALL, ""); // importa as variáveis de ambiente
    setlocale (LC_CTYPE, "pt_BR.UTF-8"); // por via das dúvidas
    char buffer[100];
    scanf ("%s", buffer);
    printf ("%s\n", buffer);
    return EXIT_SUCCESS;
}

```

Suponha que o usuário invoca o programa de modo que um arquivo `entra.txt` seja [usado no papel de stdin](#). Se o arquivo `entra.txt` contiver

áéíóú

e estiver codificado em UTF-8, o programa imprimirá

áéíóú

Note que a primeira linha do arquivo `entra.txt` tem 5 letras acentuadas (seguidas de um `\n`) e portanto tem 11 bytes: 2 bytes para cada letra e mais 1 byte para `\n`. A [string](#) armazenada no vetor `buffer` tem 11 bytes: os 10 bytes das letras acentuadas mais o byte nulo `\0` que marca o fim da string.

[*The Absolute Minimum Every Software Developer Must Know About Unicode and Character Sets*](#), por Joel Spolsky

[*On the Goodness of Unicode*](#), [*Characters vs. Bytes*](#), [*On Character Strings*](#), [*Programming Languages and Text*](#), por Tim Bray

[*Unicode in C and C++: What You Can Do About It Today*](#), por Jeff Bezanson

Veja a página [*GNU libunistring*](#): bibliotecas de manipulação de strings UTF-8.

Atualizado em 2018-04-27

<https://www.ime.usp.br/~pf/algoritmos/>

Paulo Feofiloff

[*DCC-IME-USP*](#)



Pré-processamento

A [compilação](#) de qualquer programa C começa com um *pré-processamento* que cuida das linhas do programa que começam com "#". Esse pré-processamento é realizado por um módulo do compilador conhecido como *pré-processador*, ou *pré-compilador*.

A ideia de um programa que tem dois “níveis lógicos”, com uma sintaxe para uma parte das linhas de código e outra sintaxe para as demais linhas, pode parecer deselegante. Mas do ponto de vista prático, ela é útil e poderosa.

Este sítio usa apenas duas “diretivas” (= *directives*) de pré-processamento: `#define` e `#include`.

A diretiva define

A diretiva `define` define uma abreviatura — conhecida como *macro* — para um trecho de código. Por exemplo, a linha

```
#define MAX 1000
```

define “MAX” como abreviatura de “1000”. A sequência de caracteres que começa depois de “MAX” e vai △ até o fim da linha é o significado da abreviatura, ou seja, o valor da macro. (Se você escrever um “;” no fim da linha, esse caractere fará parte da macro!)

Exemplo 1

O programa

```
#define MAX 1000

int main (void) {
    int v[MAX];
    for (int i = 0; i < MAX; ++i) {
        . .
    }
    return EXIT_SUCCESS;
}
```

é transformado pelo pré-processador em

```
int main (void) {
    int v[1000];
    for (int i = 0; i < 1000; ++i) {
        . .
    }
    return EXIT_SUCCESS;
}
```

Exemplo 2

Tal como uma função, uma macro pode ter parâmetros. Assim, o trecho de programa

```
#define troca (A, B) {int t = A; A = B; B = t;}
```

```

for (int i = 0; i < 100; ++i) {
    int k = i;
    for (int j = i+1; j <= 100; ++j)
        if (a[j] < a[k]) k = j;
    troca (a[i], a[k]);
}

```

é transformado pelo pré-processador em

```

for (int i = 0; i < 100; ++i) {
    int k = i;
    for (int j = i+1; j <= 100; ++j)
        if (a[j] < a[k]) k = j;
    {int t = a[i]; a[i] = a[k]; a[k] = t;}
}

```

A diretiva include

A diretiva `include` acrescenta o conteúdo de um arquivo especificado seu programa.

Exemplo 1

Suponha que um arquivo `aaa.txt` no diretório corrente tem o seguinte conteúdo:

```

int GLOB = 8; // variável global
int func (int e); // protótipo de função

```

Então o programa

```

#include "aaa.txt"

int main (void) {
    while (GLOB <= 64) {
        int y = func (5);
        printf ("%d\n", y);
        GLOB *= 2;
    }
    return EXIT_SUCCESS;
}

int func (int i) { // calcula GLOB^i
    . .
}

```

será transformado pelo pré-processador em

```

int GLOB = 8;
int func (int e);

int main (void) {
    while (GLOB <= 64) {
        int y = func (5);
        printf ("%d\n", y);
        GLOB *= 2;
    }
    return EXIT_SUCCESS;
}

int func (int i) { // calcula GLOB^i
    . .
}

```

Exemplo 2

Se o nome do arquivo a ser incluído estiver embrulhado em “<” e “>” (e não em aspas duplas), o pré-processador irá procurá-lo num diretório apropriado do sistema (usualmente no diretório `/usr/include/`) e não no diretório corrente:

```
#include <aaa.txt>

int main (void) {
    . .
}
```

Isso é usado, em geral, para incluir [interfaces de bibliotecas padrão](#) no seu programa.

Exemplo 3

O arquivo a ser incluído pode conter outros `#include` e `#define`. Nesse caso, a expansão das diretivas é recursiva. Suponha, por exemplo, que o arquivo `aaa.txt` no diretório corrente consiste no seguinte:

```
#define PI 3.14159
#include <math.h>
```

Então o programa

```
#include "aaa.txt"

int main (void) {
    double y = sin (PI/4);
    printf ("%f\n", y);
    return EXIT_SUCCESS;
}
```

será transformado pelo pré-processador em

```
. .
.

int main (void) {
    double y = sin (3.14159/4);
    printf ("%f\n", y);
    return EXIT_SUCCESS;
}
```

onde as linhas “. . .” estão no lugar do conteúdo do arquivo `math.h`.

A versão pré-processada de um programa

O resultado do pré-processamento de um programa é uma versão “limpa” do programa, sem diretivas de pré-processamento. O programador não precisa colocar as mãos nessa versão pré-processada, pois ela será automaticamente submetida ao módulo “nobre” do compilador. Entretanto, se tiver curiosidade, o programador pode examinar a versão pré-processada: a linha de comando

```
~$ gcc -E xxx.c > yyy.c
```

(note a opção `-E` do [compilador gcc](#)) submete o programa original `xxx.c` ao compilador e grava a versão pré-processada do programa no arquivo `yyy.c`.

Logaritmos

Este capítulo é um pequeno exercício de cálculo de logaritmos (mais precisamente, logaritmos truncados). O exercício é relevante porque a análise do consumo de tempo de muitos algoritmos envolve a função logarítmica. O cálculo de logaritmos serve de pretexto para exercitar o conceito de [piso](#) de um número real e o conceito de [invariante](#) de um algoritmo iterativo.

Aproveitaremos também a ocasião para exibir um programa completo em C, com bom [leiaute](#), boa [documentação](#), e boa [organização](#).

Piso do logaritmo na base 2

O logaritmo na base 2 de um número N é o expoente a que 2 deve ser elevado para produzir N . O logaritmo na base 2 de N é denotado por [log N](#). É claro que $\log N$ só está definido se N é [estritamente positivo](#).

[Problema](#): dado um inteiro estritamente positivo N , calcular o [piso](#) de $\log N$.

O piso de $\log N$, usualmente denotado por $\lfloor \log N \rfloor$, é aproximadamente igual ao número de bits na representação binária de N . A seguinte tabela mostra alguns valores de $\log N$ (aproximados até a terceira casa decimal) bem como os correspondentes valores de $\lfloor \log N \rfloor$:

N	$\log N$	$\lfloor \log N \rfloor$
10	3.322	3
100	6.644	6
1000	9.966	9

Eis uma função que resolve nosso problema:

```
// A função lg recebe um inteiro N > 0
// e devolve o piso de log N, ou seja,
// o único inteiro i tal que
// 2^i <= N < 2^(i+1).

int lg (int N)
{
    int i, n;
    i = 0;
    n = 1;
    while (n <= N/2) {
        n = 2 * n;
        i += 1;
    }
    return i;
}
```

O código poderia também ser escrito com divisões sucessivas no lugar das multiplicações sucessivas:

```
int lg (int N)
{
    int i = 0;
```

```

int n = N;
while (n > 1) {
    n = n / 2;
    i += 1;
}
return i;
}

```

Como a expressão `n / 2` só envolve objetos do tipo `int`, a operação de divisão é [inteira](#). Assim, o valor da expressão é $\lfloor n/2 \rfloor$.

Exercícios 1

1. Critique a seguinte versão da função `lg`. Ela usa as funções `log10` e `floor` da [biblioteca math](#):

```

#include <math.h>
int lg (int N) {
    double x;
    x = log10 (N) / log10 (2);
    return floor (x);
}

```

2. ★ Mostre que código abaixo corre o risco de produzir resultados errados em virtude de [overflow aritmético](#).

```

int lg (int N) {
    int i = 0, n = 1;
    while (n <= N) {
        n = 2 * n;
        i += 1;
    }
    return i-1;
}

```

3. Verifique que a seguinte versão alternativa da função `lg` está correta:

```

int i = 0;
for (int n = 2; n <= N; n = 2*n)
    i += 1;
return i;

```

4. Critique a seguinte versão alternativa da função `lg`:

```

int achou = 0, i = 0, n = 1;
while (!achou) {
    n *= 2;
    i += 1;
    if (n > N) achou = 1;
}
return i - 1;

```

5. Critique a seguinte versão alternativa da função `lg`:

```

if (N == 1) return 0;
if (N == 2) return 1;
int i = 2;
int n = 4;
while (n <= N) {
    n = 2 * n;
    i += 1;
}
return i - 1;

```

6. EFICIÊNCIA. Critique a seguinte versão alternativa da função `lg`. Ela calcula, explicitamente, a maior potência de 2 que não passa de `N`.

```

int potencia (int i) {
    int p = 1;
    for (int j = 1; j <= i; ++j) p = 2 * p;
    return p;
}
int lg (int N) {
    for (int i = 0; potencia (i) <= N; ++i) {}
    return i - 1;
}

```

- }
7. LOGARITMOS NA BASE 10. Escreva uma função que calcule o piso do logaritmo na base 10 de um número.

Análise da função lg

Considere o processo iterativo da [primeira versão da função lg](#) (aquele das multiplicações sucessivas por 2). Digamos que o início de cada iteração fica imediatamente antes do teste " $n \leq N/2$ ". Então as seguintes relações entre as variáveis n , i e N valem no início de cada iteração:

$$n = 2^i \quad \text{e} \quad n \leq N.$$

(Verifique!) Essas relações são, portanto, [invariantes](#). A validade das invariantes no início da última iteração garante que o algoritmo está correto. De fato, quando $n > N/2$

temos $N/2 < n \leq N$,
onde $n \leq N < 2n$,
onde $2^i \leq N < 2^{i+1}$,
onde $i \leq \log N < i+1$,

e [portanto](#) $i = \lfloor \log N \rfloor$. Assim, ao devolver i , a função `lg` cumpre o que prometeu.

Exercícios 2

1. Prove os dois invariantes da [primeira versão da função lg](#).
2. Quais os invariantes da [segunda versão da função lg](#) (aquele das divisões sucessivas por 2)? Use os invariantes para mostrar que essa versão está correta.
3. Escreva uma versão [recursiva](#) da função `lg`.
4. O que faz a função abaixo? Escreva o invariante que explica a função.

```
int tlg (int N) {
    int i = 0, n = 1;
    while (n < N) {
        n *= 2;
        i += 1;
    }
    return i;
}
```

5. O que é o *teto* de um número real r ? Se R é um número estritamente positivo, qual a relação entre o teto de $\log_{10} R$ e R ? Escreva uma função recursiva que receba um número inteiro estritamente positivo N e devolva o teto de $\log_{10} N$.

Um programa completo

A função `lg` que discutimos acima pode ser usada para imprimir uma tabela do piso do logaritmo na base 2. Segue um pequeno programa C que imprime uma tal tabela. Dados quaisquer números naturais B e K , o programa imprime o valor de $\lg(B^i)$ para i variando de 1 a K .

Este exercício é um pretexto para exibir um programa completo com bom [leiaute](#), boa [documentação](#) e [organização](#), bem como para ilustrar o uso de [bibliotecas de funções](#) e a [entrada de dados pela linha de comando](#).

```

// Programa: pisolog
// Autor: PF
// Data: 2017-06-29
//
// Este programa imprime uma tabela dos valores do
// piso de log N para N = B^1, B^2, ..., B^K.
// Como de hábito, log é o logaritmo na base 2.
// Os valores de B e K são especificados na linha
// de comando: para executar o programa com B = 10
// e K = 6, por exemplo, diga
//
//      ./pisolog 10 6
//
// Vamos supor que B e K são inteiros, com B >= 2
// e K >= 1.
///////////////////////////////

```

// Protótipos de funções.
// O programa usa as funções externas printf (da
// biblioteca stdio) e strtol (da biblioteca
// stdlib).

```

#include <stdio.h>
#include <stdlib.h>
int lg (int);

// Comunicação com o usuário.
// Os dois argumentos na linha de comando, B e K,
// devem ser menores que INT_MAX (veja a interface
// limits.h).

int main (int numargs, char *arg[])
{
    int B = strtol (arg[1], NULL, 10);
    int K = strtol (arg[2], NULL, 10);
    int N = 1;
    printf ("\n          N  lg(N)\n\n");
    for (int i = 1; i <= K; ++i) {
        N = B * N;
        printf ("%lld %d\n", N, lg (N));
    }
    return EXIT_SUCCESS;
}

// A função lg recebe um inteiro N > 0
// e devolve o piso de log N.

int lg (int N)
{
    int i = 0, n = N;
    while (n > 1) {
        i += 1;
        n /= 2;
    }
    return i;
}

// Exemplo com B = 10 e K = 6:
// $ ./pisolog 10 6
//
//          N  lg(N)
//
//          10    3
//          100   6
//          1000  9
//          10000 13
//          100000 16
//          1000000 19
// $

```

Se você [compilar](#) o programa e depois digitar `./pisolog 2 30` no terminal, receberá como resposta a tabela

N	lg(N)
2	1
4	2
8	3
16	4
32	5
64	6

128	7
256	8
512	9
1024	10
2048	11
4096	12
8192	13
16384	14
32768	15
65536	16
131072	17
262144	18
524288	19
1048576	20
2097152	21
4194304	22
8388608	23
16777216	24
33554432	25
67108864	26
134217728	27
268435456	28
536870912	29
1073741824	30

(Executar o programa com B igual a 2 é um bom teste porque é fácil ver se a resposta está correta.)

Exercícios 3

1. ★ Como se sabe, $\log B^i = i \log B$ para quaisquer B e i . Poderíamos pensar em reescrever o programa `pisolog` da seguinte maneira: calcule $b = \lg(B)$ e depois imprima $b, 2*b, \dots, K*b$. Isso produziria uma tabela correta?
2. ★ Ao calcular as potências de B em `main`, o programa corre o risco de produzir resultados errados em virtude de um *overflow aritmético*. (Isso acontece, por exemplo, se B vale 2 e K vale 31.) Modifique o código de `main` de modo que a execução seja interrompida antes que ocorra um overflow.
3. Se R é um número estritamente positivo, qual a relação entre o piso de $\log_{10} R$ e R ? Escreva uma função recursiva que receba um número inteiro estritamente positivo m e devolva o piso de $\log_{10} m$.

Atualizado em 2018-08-22

<https://www.ime.usp.br/~pf/algoritmos/>

Paulo Feofiloff

DCC-IME-USP



Vetores

“ . . . there is a huge difference between working programs and good programs. A good program not only works, but is easy to read and maintain.”

— P. A. Darnell, Ph. E. Margolis,
Software Engineering in C

“Programs must be written for people to read, and incidentally for machines to execute.”

— H. Abelson, G. Sussman,
The Structure and Interpretation of Computer Programs

“A Computação anda sobre três pernas: a correção, a eficiência e a elegância.”

— prof. [Imre Simon](#)

Um *vetor*, ou *arranjo* (= *array*), é uma estrutura de dados que armazena uma sequência de objetos, todos do mesmo tipo, em posições consecutivas da memória RAM (= *random access memory*) do computador. Essa estrutura permite acesso *aleatório*: qualquer elemento do vetor pode ser alcançado diretamente, sem passar pelos elementos anteriores (o décimo elemento, por exemplo, pode ser alcançado sem que seja necessário passar antes pelo primeiro, segundo, etc., nono elementos).

Como procurar um determinado objeto em um vetor? Como inserir um novo objeto no vetor? Como remover um elemento do vetor? Esses problemas serão usados como pretexto para exibir exemplos de algoritmos e para ilustrar os conceitos de correção, eficiência e elegância de código. Os três problemas — busca, inserção e remoção — reaparecerão, em outros contextos, em muitas páginas deste sítio.

Imagine que queremos armazenar n números inteiros num vetor v . O espaço ocupado pelo vetor na memória pode ter sido reservado pela declaração

```
int v[N];
```

sendo N uma constante (possivelmente uma macro definida por um `#define`). Se os números forem armazenados nas posições $0, 1, \dots, n-1$, diremos que

$v[0..n-1]$ é um *vetor de inteiros*.

É claro que devemos ter $0 \leq n \leq N$. Se $n == 0$, o vetor $v[0..n-1]$ está *vazio*. Se $n == N$, o vetor está *cheio*.

Exercícios 1

1. [Sedgewick 3.11] Diga (sem usar o computador) qual o conteúdo do vetor v depois das seguintes instruções.

```
int v[99];
for (i = 0; i < 99; ++i) v[i] = 98 - i;
for (i = 0; i < 99; ++i) v[i] = v[v[i]];
```

2. INVERSÃO. Suponha que um vetor $v[1..n]$ contém uma permutação de $1..n$. Escreva uma função que inverta essa permutação: se $v[i] == j$ no vetor original, queremos ter $v[j] == i$ no fim da execução da função.

Busca

Dado um inteiro x e um vetor $v[0..n-1]$ de inteiros, o [problema](#) da busca ($= search problem$) consiste em encontrar x em v , ou seja, *encontrar um índice k tal que $v[k] == x$.*

0	0	n-1
443	222 555 111 333 444 555 666 888 777 888 999	
x		v

É preciso começar com uma *decisão de projeto*: que fazer com as [instâncias](#) do problema que não têm solução? Ou seja, que fazer se x não está no vetor? Adotaremos a convenção de devolver -1 nesses casos; essa convenção é apropriada pois -1 não pertence ao conjunto $0..n-1$ de índices “válidos”. Para implementar essa convenção, convém varrer o vetor do fim para o começo:

```
// A função recebe x, n >= 0 e v e devolve
// um índice k em 0..n-1 tal que x == v[k].
// Se tal k não existe, devolve -1.

int
busca (int x, int n, int v[])
{
    int k;
    k = n-1;
    while (k >= 0 && v[k] != x)
        k -= 1;
    return k;
}
```

Observe que a função está correta mesmo quando $n == 0$. Observe como a função é [eficiente](#) e [elegante](#): não perde tempo à toa, não tem instruções nem variáveis desnecessárias, e não trata de casos especiais em separado.

Um código igualmente correto, eficiente e elegante pode ser escrito com um `for` no lugar do `while`:

```
for (int k = n-1; k >= 0; --k)
    if (v[k] == x) return k;
return -1;
```

Maus exemplos. Para fazer contraste com a função busca acima, eis algumas soluções deselegantes, ineficientes ou incorretas. A primeira, muito popular, usa uma [variável booleana](#) “de passagem”, ou “de sinalização”:

```
int achou = 0, k = n-1;
while (k >= 0 && achou == 0) { // deselegante
    if (v[k] == x) achou = 1; // deselegante
    else k -= 1;
}
return k;
```

A segunda, também bastante popular, trata do vetor vazio em separado, sem necessidade:

```
if (n == 0) return -1; // deselegante
else {
    int k = n-1;
    while (k >= 0 && v[k] != x) k -= 1;
    return k;
}
```

A próxima solução é ineficiente (pois continua calculando depois de ter encontrado uma solução) e deselegante (pois inicializa uma variável desnecessariamente):

```
int k = 0; // deselegante
int sol = -1;
for (k = n-1; k >= 0; --k) // ineficiente
    if (v[k] == x) sol = k; // deselegante
```

```
return sol;
```

Algumas “soluções” parecem razoáveis à primeira vista, mas estão *erradas*. No seguinte exemplo, a última iteração comete o erro de examinar $v[-1]$:

```
int k = n-1;
while (v[k] != x && k >= 0) // errado!
    k -= 1;
return k;
```

(As comparações deveriam ter sido feitas [na ordem correta](#): $k \geq 0 \&& v[k] \neq x$.)

Solução com sentinelas. Podemos tomar uma decisão de projeto diferente e devolver n , em lugar de -1 , se x não estiver em $v[0..n-1]$. Nesse caso, é melhor percorrer o vetor do começo para o fim:

```
int k = 0;
while (k < n && v[k] != x)
    k += 1;
return k;
```

A solução ficaria ainda melhor se usássemos uma *sentinela* para evitar as repetidas comparações de k com n :

```
v[n] = x; // sentinel
int k = 0;
while (v[k] != x)
    k += 1;
return k;
```

(Estamos supondo aqui que o vetor não está [cheio](#) e portanto há espaço para a sentinela.)

Exercícios 2

1. Qual o [invariante](#) do processo [iterativo](#) na função busca? [\[Solução\]](#)
2. Suponha que o vetor $v[0..n-1]$ tem dois ou mais elementos iguais a x . Qual deles será apontado pela função [busca](#)?
3. Quantas comparações entre x e elementos do vetor v a função [busca](#) faz?
4. Critique a seguinte versão da função busca:

```
int k = 0;
while (k < n && v[k] != x) k += 1;
if (v[k] == x) return k;
else return -1;
```

5. Critique a seguinte versão da função busca:

```
int sol;
for (int k = 0; k < n; ++k) {
    if (v[k] == x) sol = k;
    else sol = -1;
}
return sol;
```

6. VERSÃO BOOLEANA. Escreva uma função que receba x , v e n e devolva 1 se x está em $v[0..n-1]$ e 0 em caso contrário.
7. MÁXIMO. A função abaixo promete encontrar o valor de um elemento máximo de $v[0..n-1]$. A função cumpre a promessa?

```
int maxi (int n, int v[]) {
    int m = v[0];
    for (int j = 1; j < n; ++j)
        if (v[j-1] < v[j]) m = v[j];
    return m;
}
```

8. MÁXIMO. A seguinte função promete calcular o valor de um elemento máximo de um vetor $v[0..n-1]$. O problema faz sentido quando n vale 0? A função está correta?

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

Faz sentido trocar “ $x = v[0]$ ” por “ $x = 0$ ”, como fazem alguns programadores descuidados? Faz sentido trocar “ $x = v[0]$ ” por “ $x = \text{INT_MIN}$ ”? Faz sentido trocar “ $x < v[j]$ ” por “ $x \leq v[j]$ ”? [\[Solução parcial\]](#)

9. ★ SEGMENTO MÁXIMO DE ZEROS. Escreva uma função que calcule o comprimento do mais longo segmento de zeros (ou “carreira” de zeros) em um vetor de números inteiros. Procure examinar o menor número possível de elementos do vetor.

Busca recursiva

Eis uma solução recursiva (veja a página [Recursão e algoritmos recursivos](#)) do [problema da busca](#):

```
// Recebe x, n >= 0 e v e devolve k
// tal que 0 <= k < n e v[k] == x.
// Se tal k não existe, devolve -1

int busca_r (int x, int n, int v[]) {
    if (n == 0) return -1;
    if (x == v[n-1]) return n-1;
    return busca_r (x, n-1, v);
}
```

Como isso funciona? O número de elementos relevantes de v é n . Se n é 0 então $v[0..n-1]$ é vazio e portanto x não está em $v[0..n-1]$. Agora suponha que $n > 0$; nesse caso, x está em $v[0..n-1]$ se e somente se

x é igual a $v[n-1]$ ou x está em $v[0..n-2]$.

Resumindo: o código reduz a instância que busca x em $v[0..n-1]$ à instância que busca x em $v[0..n-2]$.

Mau exemplo. A seguinte alternativa para a função `busca_r` é muito deselegante. Ela coloca o caso $n == 1$ na base da recursão e com isso complica as coisas sem necessidade. (Além disso, só funciona se $n \geq 1$.)

```
int feio (int x, int n, int v[]) {
    if (n == 1) { // deselegante
        if (x == v[0]) return 0; // deselegante
        else return -1;
    }
    if (x == v[n-1]) return n-1;
    return feio (x, n-1, v);
}
```

Exercícios 3

- Suponha que o vetor $v[0..n-1]$ tem dois ou mais elementos iguais a x . Qual deles será apontado pela função `busca_r`?
- Quantas comparações entre x e elementos do vetor v a função `busca_r` faz?
- Critique a seguinte variante da função `busca_r`:

```

int busc (int x, int n, int v[]) {
    if (n == 0) return -1;
    int k = busc (x, n-1, v);
    if (k != -1) return k;
    if (x == v[n-1]) return n-1;
    return -1;
}

```

4. Critique a seguinte variante da função busca_r:

```

int busc (int x, int n, int v[]) {
    if (v[n-1] == x) return n-1;
    else return busc (x, n-1, v);
}

```

5. Critique a seguinte variante da função busca_r:

```

int busc (int x, int n, int v[]) {
    if (v[n-1] == x || n == 0) return n-1;
    else return busc (x, n-1, v);
}

```

6. Escreva uma função recursiva que receba um inteiro x , um vetor v e índices i e m e devolva um índice k no conjunto $i..m-1$ tal que $v[k] == x$; se tal k não existe, devolva $i-1$.

Remoção

Digamos que quero [remover](#) o elemento de índice k do vetor $v[0..n-1]$. Para isso, basta deslocar o segmento $v[k+1..n-1]$ do vetor para as posições $k..n-2$. Por exemplo, o resultado da remoção do elemento de índice 3 do vetor $0\ 11\ 22\ 33\ 44\ 55$ é o vetor $0\ 11\ 22\ 44\ 55$. É claro que o problema faz sentido se e somente se $0 \leq k < n$.

A seguinte [função](#) resolve o problema e devolve o valor do elemento removido:

```

// Esta função recebe um vetor v[0..n-1]
// e um índice k tal que 0 <= k < n.
// Ela devolve v[k] e remove esse
// elemento do vetor.

int
remove (int k, int n, int v[])
{
    int x = v[k];
    for (int j = k+1; j < n; ++j)
        v[j-1] = v[j];
    return x;
}

```

Note como tudo funciona bem até mesmo quando k vale $n-1$ e quando k vale 0 . (Mas é claro que a remoção haverá de consumir tanto mais tempo quanto menor for k .)

Como usar a função? Para remover o elemento de índice 51 de $v[0..n-1]$ (estou supondo que $51 < n$), por exemplo, basta dizer

```

x = remove (51, n, v);
n -= 1; // atualiza o valor de n

```

Versão recursiva. É um bom exercício escrever uma versão [recursiva](#) de remove. O tamanho de uma instância do problema é medido pela diferença $n-k$ e a instância é considerada pequena se $n-k$ vale 1. Portanto, a base da recursão é o caso em que k vale $n-1$.

```

// Esta função recebe um vetor v[0..n-1]
// e um índice k tal que 0 <= k < n.
// Ela devolve v[k] e remove esse
// elemento do vetor.

int remove_r (int k, int n, int v[]) {

```

```

int x = v[k];
if (k < n-1) {
    int y = remove_r (k+1, n, v);
    v[k] = y;
}
return x;
}

```

[Carlos A. Estombelo-Montesco mostrou que uma versão anterior dessa função estava errada.]

Exercícios 4

1. Que acontece se trocarmos “ $v[j-1] = v[j]$ ” por “ $v[j] = v[j+1]$ ” no código da função remove?

2. Critique as seguintes versões de remove:

```

for (int j = n-1; j > k; --j)
    v[j-1] = v[j];

for (int j = k+1; j < n; ++j)
    v[j-1] = v[j];
v[n-1] = 0;

if (k < n - 1)
    for (int j = k+1; j < n; ++j)
        v[j-1] = v[j];

```

3. Escreva uma versão da função remove que cuide de decrementar o valor de n depois da remoção.
(Sugestão: Passe o [endereço](#) da variável n à função remove.)

4. Discuta a seguinte versão de remove_r:

```

int remove_r2 (int k, int n, int v[]) {
    int x = v[k];
    if (k < n-1) {
        remove_r2 (k, n-1, v);
        v[n-2] = v[n-1];
    }
    return x;
}

```

5. Rediscuta o problema da remoção sob condições mais gerais: Suponha que a parte relevante do vetor é $v[i..m-1]$; para remove $v[k]$, puxe $v[k+1..m-1]$ para a esquerda ou empurre $v[i..k-1]$ para a direita, dependendo de qual das alternativas for mais rápida. (E não esqueça de atualizar os valores de i e m .)

Inserção

Dado um vetor de números $v[0..n-1]$, queremos inserir um novo número x entre os elementos de índices $k-1$ e k . Isso faz sentido não só quando $1 \leq k \leq n-1$ como também quando k vale 0 (insere no início) e quando k vale n (insere no fim). Em suma, faz sentido para qualquer k no conjunto $0..n$.

É claro que você só deve inserir x se tiver certeza de que o vetor não está [cheio](#); caso contrário, teremos um *transbordamento* (= *overflow*). Portanto, certifique-se de que $n+1 \leq N$ antes de chamar a função.

```

// Esta função insere x entre as
// posições k-1 e k do vetor v[0..n-1]
// supondo que 0 <= k <= n.

void
insere (int k, int x, int n, int v[])
{
    for (int j = n; j > k; --j)
        v[j] = v[j-1];
}

```

```

    v[k] = x;
}

```

Note como `insere` funciona bem até mesmo quando quero inserir no início do vetor e quando quero inserir no fim!

Para inserir um novo elemento com valor 999 entre as posições 50 e 51 (estou supondo que $51 \leq n$) basta dizer

```

insere(51, 999, n, v);
n++; // atualiza n

```

Versão recursiva. É um bom exercício escrever uma versão [recursiva](#) de `insere`:

```

// Esta função insere x entre as
// posições k-1 e k do vetor v[0..n-1]
// supondo que 0 <= k <= n.

void insere_r(int k, int x, int n, int v[]) {
    if (k == n) v[n] = x;
    else {
        v[n] = v[n-1];
        insere_r(k, x, n - 1, v);
    }
}

```

Exercícios 5

1. Escreva uma função que insira x entre as posições k e $k+1$ de um vetor $v[0..n-1]$. Escreva também uma boa [documentação](#) da função.
2. Escreva uma versão da função `insere` que cuide de incrementar o valor de n depois da inserção.
(Sugestão: Passe o [endereço](#) da variável n à função `insere`.)
3. Discuta a seguinte versão de `insere_r`:

```

int insere_r2(int k, int x, int n, int v[]) {
    if (k == n) {
        v[n] = x;
        return n + 1;
    }
    else {
        int y = v[k];
        v[k] = x;
        return insere_r2(k+1, y, n, v);
    }
}

```

4. Rediscuta o problema da inserção sob condições mais gerais: Suponha que a parte relevante do vetor v é $v[i..m-1]$; para inserir x entre $v[k-1]$ e $v[k]$ você tem duas opções: empurrar $v[k..m-1]$ para a direita ou puxar $v[i..k-1]$ para a esquerda; escolha a opção mais rápida. (E não esqueça de atualizar os valores de i e m .)

Busca e remoção

Considere agora uma combinação dos problemas de busca e remoção. Suponha que queremos *remover todos os elementos nulos* da sequência $v[0], \dots, v[n-1]$. É claro que o problema faz sentido com qualquer $n \geq 0$ (o caso em que n vale 0 é trivial). Exemplo: Se

```

v vale 11 22 0 0 33 0 44

```

a função deve transformar v em 11 22 33 44. Embora o enunciado do problema não peça isso explicitamente, vamos exigir que a função devolva o número de elementos do vetor depois da remoção dos zeros.

Eis uma solução elegante e eficiente do problema:

```
// Esta função elimina todos os
// elementos nulos de v[0..n-1].
// Supõe apenas que n >= 0. A função
// deixa o resultado em v[0..i-1]
// e devolve i.

int
tirazeros (int n, int v[])
{
    int i = 0;
    for (int j = 0; j < n; ++j)
        if (v[j] != 0)
            v[i++] = v[j];
    return i;
}
```

(A instrução “`v[i++] = v[j];`” tem o mesmo efeito que “`v[i] = v[j]; i += 1;`”.) No início de cada iteração temos o seguinte invariante: `v[0..i-1]` é o vetor que resulta da remoção dos zeros de `v[0..j-1]`; é claro que $i \leq j$.

0	i	j	n
999 999 999 999 999 000 999 000 999 000 999 000 999	versão sem zeros do v[0..j-1] original	lixo lixo lixo	

Note como a coisa funciona bem até mesmo quando n vale 0. Também funciona bem quando o vetor dado não tem zero algum. Também funciona bem quando o vetor dado só tem zeros.

Para remover os elementos nulos de um vetor `v[0..n-1]`, basta dizer

```
n = tirazeros (n, v);
```

Maus exemplos. Eis uma versão pior, que desperdiça espaço (pois usa um vetor auxiliar de n elementos):

```
int vv[1000], i = 0;
for (int j = 0; j < n; ++j)
    if (v[j] != 0) vv[i++] = v[j];
for (int j = 0; j < i; ++j)
    v[j] = vv[j];
return i;
```

Eis uma versão que está simplesmente errada (por que?):

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

Segue mais uma versão deselegante e ineficiente. Esta versão é ineficiente porque a instrução “`v[j] = v[j+1]`” é repetida um número excessivo de vezes, uma vez que não copia `v[j+1]` para o seu lugar definitivo:

```
int i = 0;
while (i < n) {
    if (v[i] != 0) i += 1;
    else {
        for (int j = i; j+1 < n; ++j)
            v[j] = v[j+1];
        --n;
    }
}
return n;
```

Para melhor sentir a ineficiência desta versão, considere o seguinte exemplo. Suponha que n é

100 e que todos os elementos de v exceto $v[99]$ são nulos. A versão acima copia elementos de v de um lugar para outro 4950 vezes enquanto a [primeira versão de tirazeros](#) faz isso uma só vez!

Segue uma versão ainda mais deselegante e ineficiente. A alteração do valor de i dentro do `for` controlado por i é uma péssima maneira de obter o efeito desejado. Além disso, a variável j é inicializada desnecessariamente e a condição “ $i < n-1$ ” no `if` é supérflua.

```
int j = 0;
for (int i = 0; i < n; ++i)
    if (i < n-1 && v[i] == 0) {
        for (j = i; j+1 < n; ++j)
            v[j] = v[j+1];
        --n;
        --i;
    }
return n;
```

A versão seguinte não é mais eficiente que a anterior, mas pelo menos não é tão torta:

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

Versão recursiva. Para terminar, eis uma versão [recursiva](#) de `tirazeros`. Note como a instrução “ $v[m] = v[n-1]$ ” coloca $v[n-1]$ no seu lugar definitivo.

```
// A função tirazeros_r elimina todos
// os elementos nulos de  $v[0..n-1]$ .
// A função deixa o resultado em
//  $v[0..i-1]$  e devolve  $i$ .

int tirazeros_r (int n, int v[]) {
    if (n == 0) return 0;
    int m = tirazeros_r (n - 1, v);
    if (v[n-1] == 0) return m;
    v[m] = v[n-1];
    return m + 1;
}
```

Segue uma versão *torta* e *ineficiente*. Ao contrário da versão anterior, ela não coloca cada elemento do vetor no seu lugar definitivo de uma só vez. (Note que há *duas* chamadas recursivas da função e que as duas chamadas têm forma diferente. Mau sinal!)

```
// Recebe  $0 \leq k \leq n$  e elimina os zeros
// de  $v[k..n-1]$ . O vetor sem os zeros terá
// a forma  $v[k..m-1]$ . A função devolve o
// valor de  $m$ .

int ineficiente (int k, int n, int v[]) {
    if (k == n) return n;
    if (v[k] != 0)
        return ineficiente (k+1, n, v);
    for (int i = k; i < n-1; ++i) v[i] = v[i+1];
    return ineficiente (k, n-1, v);
}
```

Às vezes a versão recursiva de uma função exige parâmetros adicionais, como aconteceu aqui com o parâmetro k . É fundamental explicar ao usuário, como fizemos acima, *qual o papel do parâmetro adicional*. (Mas é claro que nem a melhor das explicações pode tornar boa uma função malconcebida.)

Exercícios 6

1. Critique a seguinte função. Ela promete eliminar os zeros de $v[0..n-1]$, deixar o resultado em $v[0..m-1]$ e devolver m .

```
int tira0 (int n, int v[]) {
    int z = 0;
    for (int i = 0; i < n; ++i) {
        if (v[i] == 0) z += 1;
        v[i-z] = v[i];
    }
    return n - z;
}
```

2. Critique a seguinte função. Ela promete eliminar os zeros de $v[0..n-1]$, deixar o resultado em $v[0..m-1]$ e devolver m .

```
int tira0 (int n, int v[]) {
    int z = 0;
    for (int i = 0; i < n - z; ++i) {
        if (v[i] == 0) {
            z += 1;
            for (int k = i; k < n - z; ++k)
                v[k] = v[k+1];
            --i;
        }
    }
    return n - z;
}
```

3. Escreva uma função que remova de $v[i..m-1]$ todos os elementos que têm um dado valor y .

4. ★ Escreva uma função recursiva que apague todos os $\#$ de um vetor $c[0..n-1]$ de [caracteres ASCII](#). Exemplo: Se n vale 7 e o vetor contém $a\ b\ c\ \#\ \#\ d\ \#$ então o resultado deve ser $a\ b\ c\ d$.

5. ★ Escreva uma função que receba duas [strings ASCII](#), str e del , e apague de str todos os bytes que estão em del . Por exemplo, se str é "aaa*\$-bbb#ccc*" e del é "\$#*", o estado final de str deve ser "aaa-bbbccc". Procure escrever uma função que seja eficiente. Sua função não deve alocar nenhum vetor novo.

6. PEQUENA APLICAÇÃO. Escreva um programa para administrar uma coleção de números digitados pelo usuário. A coleção pode conter mais de uma cópia de um mesmo número. O usuário pode inserir novos números na coleção e remover números que já estão lá. A coleção é armazenada em ordem crescente. Se o usuário digitar

i 222

(seguido de ↵) o número 222 é inserido na coleção. Se digitar

r 333

o número 333 é removido da coleção (se esse número não estiver na coleção, o comando é ignorado). Depois de cada inserção ou remoção, o programa deve exibir a coleção. Se o usuário digitar qualquer outro caractere que não 'i' ou 'r', a execução do programa termina. [\[Solução\]](#)

Exercícios 7: desafios

- PROBLEMA DE JOSEPHUS. Imagine uma roda de n pessoas. Suponha que as pessoas estão numeradas de 1 a n no sentido horário. Começando com a pessoa de número 1, percorra a roda no sentido horário e elimine cada m -ésima pessoa enquanto a roda tiver duas ou mais pessoas. Qual o número do sobrevivente?
- SEGMENTO CONSTANTE. Digamos que um [segmento](#) $v[i..j]$ de um vetor $v[0..n-1]$ é *constante* se todos os seus elementos têm o mesmo valor. Escreva uma função simples e eficiente que calcule o comprimento de um segmento constante de comprimento máximo de um vetor $v[0..n-1]$.
- SUBSEQUÊNCIA. Um [subvetor](#) de um vetor v é o que sobra depois que alguns dos elementos de v são apagados. (Por exemplo, 12 13 10 3 é um subvetor de 11 12 13 11 10 9 7 3 3 mas não de 11 12 10 11 13 9 7 3 3.) Escreva uma função eficiente que decida se $x[0..m-1]$ é subvetor de $v[0..n-1]$.
- SUBSEQUÊNCIA CRESCENTE MÁXIMA. Um [subvetor](#) de um vetor v é o que sobra depois que alguns dos elementos de v são apagados. Escreva uma função eficiente que determine um subvetor [crescente](#) de comprimento máximo de um vetor $v[0..n-1]$.
- SEGMENTO DE SOMA MÁXIMA. A *soma* de um vetor $v[i..k]$ é o número $v[i] + v[i+1] + \dots + v[k]$. A *altura* de um vetor $v[1..n]$ é a soma de um [segmento](#) não vazio de $v[1..n]$ que tenha soma

máxima. Em outras palavras, a altura de $v[1..n]$ é a maior soma da forma $v[i] + v[i+1] + \dots + v[k]$ com $1 \leq i \leq k \leq n$. (Exemplo: Um dos segmentos do vetor 20 -30 15 -10 30 -20 -30 30 tem soma $15 - 10 + 30 = 35$. Existe algum segmento com soma maior que 35?) Escreva uma função eficiente que calcule a altura de um vetor $v[1..n]$ de números inteiros. Use o algoritmo mais eficiente que puder.

6. SOMA DE DOIS ELEMENTOS DISTANTES. Dado um vetor $v[0..n-1]$ de números e um inteiro d tal que $0 \leq d \leq n-1$ encontrar o maior número da forma $v[i] + v[j]$ com $j - i \geq d$.

Perguntas e respostas

- PERGUNTA: Uma das funções acima usa a expressão `while (k >= 0 && v[k] != x)`. Eu não deveria escrever `while ((k >= 0) && (v[k] != x))`?
- RESPOSTA: Não. Os parênteses adicionais são supérfluos porque os operadores `>=` e `!=` têm precedência sobre `&&`.
- PERGUNTA: Por que deixar um espaço entre o nome de uma função e a lista de seus argumentos?
- RESPOSTA: Eu prefiro “`funcao (arg1, arg2)`” a “`funcao(arg1, arg2)`” porque na segunda alternativa o nome da função fica encavalado com o primeiro argumento, dificultando a leitura. Veja o capítulo [Leiaute](#).

Atualizado em 2018-08-25

<https://www.ime.usp.br/~pf/algoritmos/>

Paulo Feofiloff
[DCC-IME-USP](#)



Recursão e algoritmos recursivos

*“Ao tentar resolver o problema,
encontrei obstáculos dentro de obstáculos.
Por isso, adotei uma solução recursiva.”*
— aluno S.Y.

*“To understand recursion,
we must first understand recursion.”*
— anônimo

*“Para fazer um procedimento recursivo
é preciso ter fé.”*
— prof. [Siang Wun Song](#)

Muitos problemas têm a seguinte propriedade: cada [instância](#) do problema contém uma instância menor do mesmo problema. Dizemos que esses problemas têm *estrutura recursiva*. Para resolver uma instância de um problema desse tipo, podemos aplicar o seguinte método:

- se a instância em questão for pequena,
resolva-a diretamente (use força bruta se necessário);
- senão
 reduza-a a uma instância menor do mesmo problema,
 aplique o método à instância menor,
 volte à instância original.

O programador só precisa mostrar como obter uma solução da instância original a partir de uma solução da instância menor; o computador faz o resto. A aplicação desse método produz um algoritmo *recursivo*.

Um exemplo

Para ilustrar o conceito de algoritmo recursivo, considere o seguinte [problema](#): *Determinar o valor de um elemento máximo de um vetor $v[0..n-1]$.* (O problema já foi mencionado [num dos exercícios](#) na página *Vetores*.)

Observe que o problema só faz sentido se o vetor não é vazio, ou seja, se $n \geq 1$. Eis uma função recursiva que resolve o problema:

```
// Ao receber v e n >= 1, a função devolve
// o valor de um elemento máximo do
// vetor v[0..n-1].
//
int
maximo (int n, int v[])
{
    if (n == 1)
        return v[0];
    else {
        int x;
        x = maximo (n-1, v);
        // x é o máximo de v[0..n-2]
        if (x > v[n-1]) return x;
        else return v[n-1];
    }
}
```

}

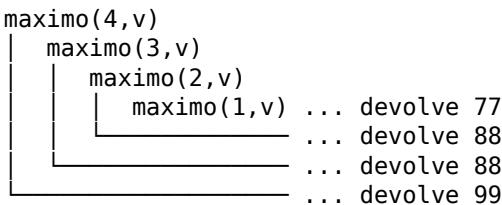
A análise da [correção](#) da função tem a forma de uma prova por indução. Para começar, considere o caso em que n vale 1. Nesse caso, a resposta que a função devolve, $v[0]$, está correta pois $v[0]$ é o único elemento relevante do vetor. Agora considere o caso em que n é maior que 1. Nesse caso, o vetor tem duas partes não vazias:

$v[0..n-2]$ e $v[n-1]$.

Podemos supor que a função resolve corretamente a instância $v[0..n-2]$ do problema, pois essa instância é menor que a original. Portanto, depois da linha “ $x = \maximo(n-1, v)$ ”, podemos supor que x é um máximo correto de $v[0..n-2]$. Logo, a solução da instância original é o maior dos números x e $v[n-1]$. E é justamente esse o número que a função calcula e devolve. Isso conclui a prova de que a função `maximo` está correta.

Para que uma função recursiva seja compreensível, é essencial que o autor diga [o que a função faz](#). Isso deve ser dito de maneira explícita e completa, como fiz acima no comentário que precede o código da função.

Como o computador executa uma função recursiva? Embora relevante e importante, essa pergunta será ignorada por enquanto. (Mas você pode ver o apêndice [A pilha de execução de um programa](#) do capítulo *Pilhas*.) Vamos nos limitar a mostra um exemplo de execução. Se $v[0..3]$ é o vetor 77 88 66 99 então teremos a seguinte sequência de chamadas da função:



Desempenho. Algumas pessoas acreditam que funções recursivas são inherentemente lentas, mas isso não passa de lenda. Talvez a lenda tenha origem em certos usos descuidados da recursão, como em [alguns](#) dos exercícios [abaixo](#). (Nem tudo são flores, entretanto: o espaço de memória que uma função recursiva consome para “[rascunho](#)” pode ser grande.)

Exercícios 1

1. Que tipos de problemas podem ser resolvidos por um algoritmo recursivo?
2. Considere a função `maximo` acima. Faz sentido trocar “`return v[0]`” por “`return 0`”? Faz sentido trocar “`return v[0]`” por “`return INT_MIN`”? Faz sentido trocar “`x > v[n-1]`” por “`x >= v[n-1]`”?
3. Considere a função `maximo` acima. Se o vetor $v[0..n-1]$ tiver dois os mais elementos máximos, qual deles a função devolve?
4. Considere a função `maximo` acima. Quantas comparações envolvendo elementos do vetor sua função faz no pior caso?
5. Verifique que a seguinte maneira de escrever a função `maximo` é exatamente equivalente à dada [acima](#):

```
int maximo (int n, int v[]) {
    int x;
    if (n == 1) x = v[0];
    else {
        x = maximo (n-1, v);
        if (x < v[n-1]) x = v[n-1];
    }
    return x;
}
```

6. Verifique que a seguinte maneira de escrever a função `maximo` é exatamente equivalente à dada

[acima](#):

```
int maximo (int n, int v[]) {
    if (n == 1) return v[0];
    int x = maximo (n-1, v);
    if (x > v[n-1]) return x;
    else return v[n-1];
}
```

7. Critique a seguinte função recursiva que promete encontrar o valor de um elemento máximo de $v[0..n-1]$:

```
int maxim1 (int n, int v[]) {
    if (n == 1) return v[0];
    if (n == 2) {
        if (v[0] < v[1]) return v[1];
        else return v[0];
    }
    int x = maxim1 (n-1, v);
    if (x < v[n-1]) return v[n-1];
    else return x;
}
```

8. Critique a seguinte função recursiva que promete encontrar o valor de um elemento máximo de $v[0..n-1]$:

```
int maxim2 (int n, int v[]) {
    if (n == 1) return v[0];
    if (maxim2 (n-1, v) < v[n-1])
        return v[n-1];
    else
        return maxim2 (n-1, v);
}
```

9. PROGRAMA DE TESTE. Escreva um pequeno programa para testar a função recursiva `maximo` dada [acima](#). O seu programa deve gerar um vetor [aleatório](#) e encontrar um elemento máximo desse vetor. Acrescente ao seu programa uma função que *confira a resposta* dada por `maximo`. [Veja uma [solução](#)]

10. Escreva uma função recursiva `maxmin` que calcule o valor de um elemento máximo e o valor de um elemento mínimo de um vetor $v[0..n-1]$. Quantas comparações envolvendo os elementos do vetor sua função faz?

Outra solução

A função `maximo` discutida [acima](#) reduz a instância $v[0..n-1]$ do problema à instância $v[0..n-2]$. Podemos escrever uma função que reduza $v[0..n-1]$ a $v[1..n-1]$, ou seja, “empurre pra direita” o início do vetor:

```
// Ao receber v e n >= 1, esta função devolve
// o valor de um elemento máximo do vetor
// v[0..n-1].
int
maximo2 (int n, int v[])
{
    return max (0, n, v);
}
```

A função `maximo2` é apenas uma “embalagem” ou “invólucro” (= *wrapper-function*); o serviço propriamente dito é executado pela função recursiva `max`:

```
// Recebe v e i < n e devolve o valor de
// um elemento máximo do vetor v[i..n-1].
int
max (int i, int n, int v[])
{
    if (i == n-1) return v[i];
    else {
```

```

        int x;
        x = max (i + 1, n, v);
        if (x > v[i]) return x;
        else return v[i];
    }
}

```

A função `max` resolve um problema mais geral que o original (veja o parâmetro `i`). A necessidade de generalizar o problema original ocorre com frequência no projeto de algoritmos recursivos.

A título de curiosidade, eis outra maneira, talvez surpreendente, de aplicar recursão ao segmento `v[1..n-1]`. Ela usa [aritmética de endereços](#):

```

int maximo2b (int n, int v[]) {
    if (n == 1) return v[0];
    int x = maximo2b (n - 1, v + 1);
    if (x > v[0]) return x;
    return v[0];
}

```

Exercícios 2

1. Considere a função `maximo2` acima. Se o vetor `v[0..n-1]` tiver dois os mais elementos máximos, qual deles a função devolve? E se trocarmos `if (x > v[i])` por `if (x >= v[i])`?
2. A seguinte função recursiva pretende encontrar o valor de um elemento máximo de `v[p..u]`, supondo $p \leq u$. (O índice `p` aponta o primeiro elemento do vetor e o índice `u` aponta o último.) A função está correta? Suponha que `p` e `u` valem 0 e 6 respectivamente e mostre a sequência, devidamente indentada, de chamadas de `maxx`.

```

int maxx (int p, int u, int v[]) {
    if (p == u) return v[u];
    else {
        int m, x, y;
        m = (p + u)/2; // p ≤ m < u
        x = maxx (p, m, v);
        y = maxx (m+1, u, v);
        if (x >= y) return x;
        else return y;
    }
}

```

Exercícios 3

1. Escreva uma função recursiva que calcule a soma dos elementos [positivos](#) do vetor de inteiros `v[0..n-1]`. O problema faz sentido quando `n` é igual a 0? Quanto deve valer a soma nesse caso?
2. Critique a documentação do seguinte código:

```

// Recebe um vetor de tamanho n e devolve
// a soma dos elementos do vetor.
int soma (int v[], int n) {
    return sss (v, n+1);
}
int sss (int v[], int n) {
    if (n == 1) return 0;
    return sss (v, n-1) + v[n-1];
}

```

3. Diga *o que* a função abaixo faz. Para quais valores dos parâmetros sua resposta está correta?

```

int ttt (int x[], int n) {
    if (n == 0) return 0;
    if (x[n] > 100) return x[n] + ttt (x, n-1);
    else return ttt (x, n-1);
}

```

- }
4. Escreva uma função recursiva que calcule o produto dos elementos estritamente positivos de um vetor de inteiros $v[0..n-1]$. O problema faz sentido quando todos os elementos do vetor são nulos? O problema faz sentido quando n vale 0? Quanto deve valer o produto nesses casos?
 5. Escreva uma função recursiva que calcule a soma dos dígitos decimais de um inteiro estritamente positivo n . A soma dos dígitos de 132, por exemplo, é 6.
 6. Escreva uma função recursiva que calcule o piso do logaritmo de N na base 2. (Veja uma [versão não-recursiva do exercício](#).)

Exercícios 4

1. Qual o valor de $X(4)$ se X é dada pelo seguinte código? [Veja uma [solução](#).]

```
int X (int n) {
    if (n == 1 || n == 2) return n;
    else return X (n-1) + n * X (n-2);
}
```

2. Qual é o valor de $f(1, 10)$? Escreva uma função equivalente que seja mais simples.

```
double f (double x, double y) {
    if (x >= y) return (x + y)/2;
    else return f (f (x+2, y-1), f (x+1, y-2));
}
```

3. Qual o resultado da execução do seguinte programa?

```
int ff (int n) {
    if (n == 1) return 1;
    if (n % 2 == 0) return ff (n/2);
    return ff ((n-1)/2) + ff ((n+1)/2);
}
int main (void) {
    printf ("%d", ff(7));
    return EXIT_SUCCESS;
}
```

4. Execute $\text{fusc}(7, 0)$. Mostre a sequência, devidamente indentada, das chamadas a fusc .

```
int fusc (int n, int profund) {
    for (int i = 0; i < profund; ++i)
        printf ("    ");
    printf ("fusc(%d,%d)\n", n, profund);
    if (n = 1)
        return 1;
    if (n % 2 == 0)
        return fusc (n/2, profund+1);
    return fusc ((n-1)/2, profund+1) +
        fusc ((n+1)/2, profund+1);
}
```

5. IMPORTANTE. Critique a seguinte função recursiva:

```
int XX (int n) {
    if (n == 0) return 0;
    else return XX (n/3+1) + n;
}
```

6. FIBONACCI. A função de Fibonacci é definida assim: $F(0) = 0$, $F(1) = 1$ e $F(n) = F(n-1) + F(n-2)$ para $n > 1$. Descreva a função F em linguagem C. Faça uma versão recursiva e uma [iterativa](#).

7. Seja F a versão recursiva da [função de Fibonacci](#). O cálculo do valor da expressão $F(3)$ provocará a seguinte sequência de invocações da função:

```
F(3)
  F(2)
    F(1)
      F(0)
        F(1)
```

Qual a sequência de invocações da função durante o cálculo de $F(5)$?

8. EUCLIDES. A seguinte função calcula o maior divisor comum dos inteiros estritamente positivos m e n . Escreva uma função recursiva equivalente.

```
int Euclides (int m, int n) {
    int r;
    do {
        r = m % n;
        m = n;
        n = r;
    } while (r != 0);
    return m;
}
```

9. EXPONENCIAÇÃO. Escreva uma função recursiva eficiente que receba inteiros estritamente positivos k e n e calcule k^n . (Suponha que k^n cabe em um [int](#).) Quantas multiplicações sua função executa aproximadamente?

10. RÉGUA INGLESA [Sedgewick 5.8, Roberts 5.11] Escreva uma [função](#) recursiva que imprima uma *régua de ordem* n no intervalo $[0..2^n]$. O “traço” no ponto médio da régua deve ter comprimento n , os traços nos pontos médios dos subintervalos superior e inferior devem ter comprimento $n-1$, e assim por diante. A figura mostra uma régua de ordem 4.



Perguntas e respostas

- PERGUNTA: Num dos exercícios acima aparece a expressão `if (n == 1 || n == 2)`. Eu não deveria escrever `if ((n == 1) || (n == 2))`?
RESPOSTA: Não. Os parênteses adicionais são redundantes porque o operador `==` tem [precedência](#) sobre `||`.

Veja o verbete [Recursion](#) na Wikipedia

Veja bons exemplos de recursão no capítulo sobre [algoritmos de enumeração](#)

Atualizado em 2018-05-21

<https://www.ime.usp.br/~pf/algoritmos/>

Paulo Feofiloff

[DCC-IME-USP](#)



Listas encadeadas

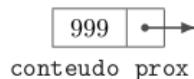


Uma lista encadeada é uma representação de uma [sequência](#) de objetos, todos do mesmo tipo, na memória RAM (= *random access memory*) do computador. Cada elemento da sequência é armazenado em uma célula da lista: o primeiro elemento na primeira célula, o segundo na segunda, e assim por diante.

Estrutura de uma lista encadeada

Uma *lista encadeada* (= *linked list* = lista ligada) é uma sequência de *células*; cada célula contém um objeto (todos os objetos são do mesmo tipo) e o [endereço](#) da célula seguinte. Neste capítulo, suporemos que os objetos armazenados nas células são do tipo `int`. Cada célula é um [registro](#) que pode ser definido assim:

```
struct reg {
    int      conteudo;
    struct reg *prox;
};
```



É conveniente tratar as células como um novo [tipo-de-dados](#) e atribuir um nome a esse novo tipo:

```
typedef struct reg celula; // célula
```

Uma célula `c` e um [ponteiro](#) `p` para uma célula podem ser declarados assim:

```
celula c;
celula*p;
```

Se `c` é uma célula então `c.conteudo` é o conteúdo da célula e `c.prox` é o endereço da próxima célula. Se `p` é o endereço de uma célula, então `p->conteudo` é o conteúdo da célula e `p->prox` é o endereço da próxima célula. Se `p` é o endereço da *última* célula da lista então `p->prox` vale `NULL`.



(A figura pode dar a falsa impressão de que as células da lista ocupam posições consecutivas na memória. Na realidade, as células estão tipicamente espalhadas pela memória de maneira

imprevisível.)

Exercícios 1

1. DECLARAÇÃO ALTERNATIVA. Verifique que a declaração de células pode também ser escrita assim:

```
typedef struct reg celula;
struct reg {
    int conteudo;
    celula *prox;
};
```

2. ★ DECLARAÇÃO ALTERNATIVA. Verifique que a declaração de células pode também ser escrita assim:

```
typedef struct reg {
    int conteudo;
    struct reg *prox;
} celula;
```

3. TAMANHO DE CÉLULA. Compile e execute o seguinte programa:

```
int main (void) {
    printf ("sizeof (celula) = %d\n",
    sizeof (celula));
    return EXIT_SUCCESS;
}
```

Endereço de uma lista encadeada

O *endereço* de uma lista encadeada é o endereço de sua primeira célula. Se `le` é o endereço de uma lista encadeada, convém dizer simplesmente que

le é uma lista encadeada.

(Não confunda “`le`” com “`1e`”.) A lista está *vazia* (ou seja, não tem célula alguma) se e somente se `le == NULL`.

Listas são animais eminentemente [recursivos](#). Para tornar isso evidente, basta fazer a seguinte observação: se `le` é uma lista não vazia então `le->prox` também é uma lista. Muitos algoritmos sobre listas encadeadas ficam mais simples quando escritos em estilo recursivo.

EXEMPLO. A seguinte função recursiva imprime o conteúdo de uma lista encadeada `le`:

```
void imprime (celula *le) {
    if (le != NULL) {
        printf ("%d\n", le->conteudo);
        imprime (le->prox);
    }
}
```

E aqui está a versão iterativa da mesma função:

```
void imprime (celula *le) {
    celula *p;
    for (p = le; p != NULL; p = p->prox)
        printf ("%d\n", p->conteudo);
}
```

Exercícios 2

1. Escreva uma função que *conte* o número de células de uma lista encadeada. Faça duas versões: uma iterativa e uma recursiva.
2. ALTURA. A *altura* de uma célula *c* em uma lista encadeada é a distância entre *c* e o fim da lista. Mais precisamente, a altura de *c* é o número de passos do caminho que leva de *c* até a última célula da lista. Escreva uma função que calcule a altura de uma dada célula.
3. PROFUNDIDADE. A *profundidade* de uma célula *c* em uma lista encadeada é o número de passos do único caminho que vai da primeira célula da lista até *c*. Escreva uma função que calcule a profundidade de uma dada célula.

Busca em uma lista encadeada

Veja como é fácil verificar se um objeto *x* pertence a uma lista encadeada, ou seja, se é igual ao conteúdo de alguma célula da lista:

```
// Esta função recebe um inteiro x e uma
// lista encadeada le de inteiros e devolve
// o endereço de uma celula que contém x.
// Se tal celula não existe, devolve NULL.

celula *
busca (int x, celula *le)
{
    celula *p;
    p = le;
    while (p != NULL && p->conteudo != x)
        p = p->prox;
    return p;
}
```

Que beleza! Nada de [variáveis booleanas “de sinalização”](#)! Além disso, a função tem comportamento correto mesmo que a lista esteja vazia.

Eis uma versão recursiva da mesma função:

```
celula *busca_r (int x, celula *le)
{
    if (le == NULL)  return NULL;
    if (le->conteudo == x)  return le;
    return busca_r (x, le->prox);
}
```

Exercícios 3

1. A função abaixo promete ter o mesmo comportamento da função busca acima. Critique o código.

```
celula *busca (int x, celula *le) {
    celula *p = le;
    int achou = 0;
    while (p != NULL && !achou) {
        if (p->conteudo == x) achou = 1;
        p = p->prox;
    }
    if (achou) return p;
    else return NULL;
}
```

2. Critique o código da seguinte variante da função busca.

```
celula *busca (int x, celula *le) {
    celula *p = le;
    while (p != NULL && p->conteudo != x)
        p = p->prox;
    if (p != NULL)  return p;
    else printf ("x não está na lista\n");
```

- ```

 }

3. Escreva uma função que verifique se uma lista encadeada que contém números inteiros está em ordem crescente.

4. Escreva uma função que faça uma busca em uma lista encadeada crescente. Faça versões recursiva e iterativa.

5. Escreva uma função que encontre uma célula com conteúdo mínimo. Faça duas versões: uma iterativa e uma recursiva.

6. Escreva uma função que verifique se duas listas encadeadas são iguais, ou melhor, se têm o mesmo conteúdo. Faça duas versões: uma iterativa e uma recursiva.

7. PONTO MÉDIO. Escreva uma função que receba uma lista encadeada e devolva o endereço de uma célula que esteja o mais próximo possível do meio da lista. Faça isso sem contar explicitamente o número de células da lista.

```

## Cabeça de lista

Às vezes convém tratar a primeira célula de uma lista encadeada como um mero “marcador de início” e ignorar o conteúdo da célula. Nesse caso, dizemos que a primeira célula é a *cabeça* (= *head cell = dummy cell*) da lista encadeada.

Uma lista encadeada *le* com cabeça vazia se e somente se *le->prox == NULL*. Para criar uma lista encadeada vazia com cabeça, basta dizer

```

celula *le;
le = malloc (sizeof (celula));
le->prox = NULL;

```

Para imprimir o conteúdo de uma lista encadeada *le* com cabeça, faça

```

void imprima (celula *le) {
 celula *p;
 for (p = le->prox; p != NULL; p = p->prox)
 printf ("%d\n", p->conteudo);
}

```

## Exercícios 4

1. Escreva versões das funções *busca* e *busca\_r* para listas encadeadas com cabeça.
2. Escreva uma função que verifique se uma lista encadeada com cabeça está em ordem crescente. (Suponha que as células contêm números inteiros.)

## Inserção em uma lista encadeada

Considere o problema de inserir uma nova célula em uma lista encadeada. Suponha que quero inserir a nova célula entre a posição apontada por *p* e a posição seguinte. (É claro que isso só faz sentido se *p* é diferente de *NULL*.)

```

// Esta função insere uma nova celula
// em uma lista encadeada. A nova celula
// tem conteudo x e é inserida entre a
// celula p e a celula seguinte.
// (Supõe-se que p != NULL.)
void

```

```

insere (int x, celula *p)
{
 celula *nova;
 nova = malloc (sizeof (celula));
 nova->conteudo = x;
 nova->prox = p->prox;
 p->prox = nova;
}

```

Simples e rápido! Não é preciso movimentar células para “abrir espaço” para um nova célula, como fizemos para [inserir um novo elemento em um vetor](#). Basta mudar os valores de alguns ponteiros. Observe que a função comporta-se corretamente mesmo quando quero inserir no *fim* da lista, isto é, quando *p->prox == NULL*. Se a lista tem cabeça, a função pode ser usada para inserir no *início* da lista: basta que *p* aponte para a célula-cabeça. Mas no caso de lista *sem* cabeça a função não é capaz de inserir antes da primeira célula.

O tempo que a função *insere* consome *não depende* do ponto da lista em que é feita a inserção: tanto faz inserir uma nova célula na parte inicial da lista quanto na parte final. Isso é bem diferente do que ocorre com a inserção em um vetor.

## Exercícios 5

1. Por que a seguinte versão da função *insere* não funciona?

```

void insere (int x, celula *p) {
 celula nova;
 nova.conteudo = x;
 nova.prox = p->prox;
 p->prox = &nova;
}

```

2. Escreva uma função que insira uma nova célula em uma lista encadeada *sem* cabeça. (Será preciso tomar algumas decisões de projeto antes de começar a programar.)
3. Escreva uma função que faça uma *cópia* de uma lista encadeada. Faça duas versões da função: uma iterativa e uma recursiva.
4. Escreva uma função que *concatene* duas listas encadeadas (isto é, “engate” a segunda no fim da primeira). Faça duas versões: uma iterativa e uma recursiva.
5. Escreva uma função que insira uma nova célula com conteúdo *x* *imediatamente depois* da *k*-ésima célula de uma lista encadeada. Faça duas versões: uma iterativa e uma recursiva.
6. Escreva uma função que *troque de posição* duas células de uma mesma lista encadeada.
7. Escreva uma função que *inverta* a ordem das células de uma lista encadeada (a primeira passa a ser a última, a segunda passa a ser a penúltima etc.). Faça isso sem usar espaço auxiliar, apenas alterando ponteiros. Dê duas soluções: uma iterativa e uma recursiva.
8. ALOCAÇÃO DE CÉLULAS. É uma boa ideia alocar as células de uma lista encadeada uma-a-uma? (Veja observação sobre [alocação de pequenos blocos de bytes](#) no capítulo *Alocação dinâmica de memória*.) Proponha alternativas.

## Remoção em uma lista encadeada

Considere o problema de [remover](#) uma certa célula de uma lista encadeada. Como especificar a célula em questão? A ideia mais óvia é apontar para a célula que quero remover. Mas é fácil perceber que essa ideia não é boa; é melhor apontar para a célula *anterior* à que quero remover. (Infelizmente, não é possível remover a *primeira* célula usando essa convenção.)

```

// Esta função recebe o endereço p de uma
// célula de uma lista encadeada e remove
// da lista a célula p->prox. A função supõe

```

```
// que p != NULL e p->prox != NULL.

void
remove (celula *p)
{
 celula *lixo;
 lixo = p->prox;
 p->prox = lixo->prox;
 free (lixo);
}
```

Veja que maravilha! Não é preciso copiar informações de um lugar para outro, como fizemos para [remover um elemento de um vetor](#): basta mudar o valor de um ponteiro. A função consome sempre o mesmo tempo, quer a célula a ser removida esteja perto do início da lista, quer esteja perto do fim.

Note também que a função de remoção não precisa conhecer o endereço da lista, ou seja, não precisa saber onde a lista começa.

## Exercícios 6

1. Critique a seguinte versão da função remove:

```
void remove (celula *p, celula *le) {
 celula *lixo;
 lixo = p->prox;
 if (lixo->prox == NULL) p->prox = NULL;
 else p->prox = lixo->prox;
 free (lixo);
}
```

2. Suponha que queremos remover a primeira célula de uma lista encadeada le não vazia. Critique o seguinte fragmento de código:

```
celula **p;
p = ≤
le = le->prox;
free (*p);
```

3. Invente um jeito de remover uma célula de uma lista encadeada *sem* cabeça. (Será preciso tomar algumas decisões de projeto antes de começar a programar.)
4. Escreva uma função que *desaloque* todas as células de uma lista encadeada (ou seja, aplique a função free a todas as células). Estamos supondo que cada célula da lista foi originalmente alocado por malloc. Faça duas versões: uma iterativa e uma recursiva.
5. PROBLEMA DE JOSEPHUS. Imagine uma roda de n pessoas numeradas de 1 a n no sentido horário. Começando com a pessoa de número 1, percorra a roda no sentido horário e elimine cada m-ésima pessoa enquanto a roda tiver duas ou mais pessoas. Qual o número do sobrevivente?

## Exercícios 7

1. Escreva uma função que copie o conteúdo de um vetor para uma lista encadeada preservando a ordem dos elementos. Faça duas versões: uma iterativa e uma recursiva.
2. Escreva uma função que copie o conteúdo de uma lista encadeada para um vetor preservando a ordem dos elementos. Faça duas versões: uma iterativa e uma recursiva.
3. UNIÃO. Digamos que uma *lesc* é uma lista encadeada sem cabeça que contém uma sequência estritamente crescente de números inteiros. (Portanto, uma lescl representa um *conjunto* de números.) Escreva uma função que faça a *união* de duas lescls produzindo uma nova lescl. A lescl resultante deve ser construída com as células das duas lescls dadas.
4. LISTAS ENCADEADAS SEM PONTEIROS. Implemente uma lista encadeada sem usar endereços e ponteiros. Use dois vetores paralelos: um vetor conteudo[0..N-1] e um vetor prox[0..N-1]. Para

cada  $i$  no conjunto  $0..N-1$ , o par  $(\text{conteudo}[i], \text{prox}[i])$  representa uma célula da lista. A célula seguinte é  $(\text{conteudo}[j], \text{prox}[j])$ , sendo  $j = \text{prox}[i]$ . Escreva funções de busca, inserção e remoção para essa representação.

5. Esta questão trata de listas encadeadas que contêm [strings ASCII](#) (cada célula contém uma string). Escreva uma função que verifique se uma lista desse tipo está [em ordem lexicográfica](#). As células são do seguinte tipo:

```
typedef struct reg {
 char *str; struct reg *prox;
} celula;
```

6. ★ CONTAGEM DE PALAVRAS. Digamos que um *texto* é um vetor de [bytes](#), todos com valor entre 32 e 126. (Cada um desses bytes representa um [caractere ASCII](#).) Digamos que uma *palavra* é um segmento maximal de texto que consiste apenas de letras. Escreva uma função que receba um texto e imprima uma relação de todas as palavras que ocorrem no texto juntamente com o número de ocorrências de cada palavra. Use uma lista encadeada para armazenar as palavras.

## Busca e remove

Dada uma lista encadeada  $le$  de inteiros e um inteiro  $y$ , queremos remover da lista a primeira célula que contiver  $y$ . Se tal célula não existir, não é preciso fazer nada. Para simplificar, vamos supor que a lista tem cabeça; assim, não será preciso mudar o endereço da lista, mesmo que a célula inicial contenha  $y$ .

```
// Esta função recebe uma lista encadeada le
// com cabeça e remove da lista a primeira
// célula que contiver y, se tal célula existir.

void
busca_e_remove (int y, celula *le)
{
 celula *p, *q;
 p = le;
 q = le->prox;
 while (q != NULL && q->conteudo != y) {
 p = q;
 q = q->prox;
 }
 if (q != NULL) {
 p->prox = q->prox;
 free (q);
 }
}
```

Para provar que o código está correto, é preciso verificar o seguinte invariante: no início de cada iteração (imediatamente antes do teste “ $q \neq \text{NULL}$ ”), tem-se

```
q == p->prox
```

ou seja,  $q$  está um passo à frente de  $p$ .

## Exercícios 8

1. Escreva uma função busca-e-remove para listas encadeadas *sem* cabeça.
2. Escreva uma função para remover de uma lista encadeada *todas* as células que contêm  $y$ .
3. Escreva uma função que remova a  $k$ -ésima célula de uma lista encadeada *sem* cabeça. Faça duas versões: uma iterativa e uma recursiva.

## Busca e insere

Suponha dada uma lista encadeada `le`, com cabeça. Queremos inserir na lista uma nova célula com conteúdo `x` imediatamente *antes* da primeira célula que contém `y`.

```
// Esta função recebe uma lista encadeada le
// com cabeça e insere na lista uma nova celula
// imediatamente antes da primeira que contém y.
// Se nenhuma celula contém y, insere a nova
// celula no fim da lista. O conteudo da nova
// celula é x.

void
busca_e_insere (int x, int y, celula *le)
{
 celula *p, *q, *nova;
 nova = malloc (sizeof (celula));
 nova->conteudo = x;
 p = le;
 q = le->prox;
 while (q != NULL && q->conteudo != y) {
 p = q;
 q = q->prox;
 }
 nova->prox = q;
 p->prox = nova;
}
```

## Exercícios 9

1. Escreva uma função busca-e-insere para listas encadeadas *sem* cabeça.

## Outros tipos de listas

Uma vez entendidas as listas encadeadas básicas, você pode inventar muitos outros tipos de listas encadeadas.

Por exemplo, você pode construir uma lista encadeada *circular*, em que a última célula aponta para a primeira. O endereço de uma tal lista é o endereço de qualquer uma de suas células. Você pode também ter uma lista *duplamente encadeada*: cada célula contém o endereço da célula anterior e o endereço da célula seguinte.

Pense nas seguintes questões, apropriadas para qualquer tipo de lista encadeada. Convém ter uma célula-cabeça e/ou uma célula-rabo? Em que condições a lista está vazia? Como remover a célula apontada por `p`? Idem para a célula seguinte à apontada por `p`? Idem para a célula anterior à apontada por `p`? Como inserir uma nova célula entre a célula apontada por `p` e a anterior? Idem entre `p` e a seguinte?

## Exercícios 10

1. Descreva, em linguagem C, a estrutura de uma célula de uma lista duplamente encadeada.
2. Escreva uma função que remova de uma lista duplamente encadeada a célula apontada por `p`. Que dados sua função recebe? Que coisa devolve?
3. Escreva uma função que insira uma nova célula com conteúdo `x` em uma lista duplamente

encadeada logo após a célula apontada por  $p$ . Que dados sua função recebe? Que coisa devolve?

---

Veja o verbete [Linked list](#) na Wikipedia

---

Atualizado em 2018-08-25

<https://www.ime.usp.br/~pf/algoritmos/>

*Paulo Feofiloff*

[DCC-IME-USP](#)



# Filas



Uma fila é uma estrutura de dados dinâmica que admite [remoção](#) de elementos e [inserção](#) de novos objetos. Mais especificamente, uma *fila* (= *queue*) é uma estrutura sujeita à seguinte regra de operação: sempre que houver uma remoção,

o elemento removido é o que está na estrutura há *mais tempo*.

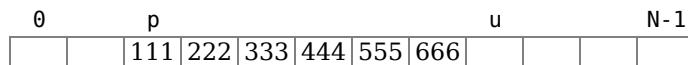
Em outras palavras, o primeiro objeto inserido na fila é também o primeiro a ser removido. Essa política é conhecida pela sigla FIFO (= *First-In-First-Out*).

## Implementação em um vetor

Suponha que nossa fila mora em um vetor `fila[0..N-1]`. (A natureza dos elementos do vetor é irrelevante: eles podem ser inteiros, bytes, ponteiros, etc.) Digamos que a parte do vetor ocupada pela fila é

`fila[p..u-1]` .

O primeiro elemento da fila está na posição `p` e o último na posição `u-1`. A fila está *vazia* se `p == u` e *cheia* se `u == N`. A figura mostra uma fila que contém os números 111, 222, ..., 666:



Para *tirar*, ou *remover* (= [delete](#) = *de-queue*), um elemento da fila basta fazer

```
x = fila[p++];
```

Isso [equivale](#) ao par de instruções “`x = fila[p]; p += 1;`”, nesta ordem. É claro que você só deve fazer isso se tiver certeza de que a fila não está vazia. Para *colocar*, ou *inserir* (= [insert](#) = *enqueue*), um objeto `y` na fila basta fazer

```
fila[u++] = y;
```

Isso equivale ao par de instruções “`fila[u] = y; u += 1;`”, nesta ordem. Note como esse código funciona corretamente mesmo quando a fila está vazia. É claro que você só deve inserir um objeto na fila se ela não estiver cheia; caso contrário, a fila *transborda* (ou seja, ocorre um *overflow*).

Para ajudar o leitor humano, podemos embalar as operações de remoção e inserção em duas pequenas funções. Se os objetos da fila forem números inteiros, podemos escrever

```
int tiradafila (void) {
 return fila[p++];
}
```

```

void colocanafila (int y) {
 fila[u++] = y;
}

```

Estamos supondo aqui que as variáveis `fila`, `p`, `u` e `N` são [globais](#), isto é, foram definidas fora do código das funções. (Para completar o pacote, precisaríamos de mais três funções: uma que crie uma fila, uma que verifique se a fila está vazia e uma que verifique se a fila está cheia; veja exercício [abaixo](#).)

## Exercícios 1

1. MÓDULO DE IMPLEMENTAÇÃO DE FILA (VERSÃO 1). Escreva um [módulo](#) `filadeints.c` que implemente uma fila de números inteiros em um vetor alocado estaticamente. O módulo deve conter as funções `criafila`, `colocanafila`, `tiradafila`, `filavazia` e `filacheia`. O vetor que abriga a fila bem como os índices que indicam o início e o fim da fila devem ser variáveis globais do módulo. Escreva também uma [interface](#) `filadeints.h` para o módulo. [\[Solução\]](#)
2. Escreva uma função que devolva o comprimento (ou seja, o número de elementos) da fila.
3. Suponha que, diferentemente da convenção adotada acima, a parte do vetor ocupada pela fila é `fila[p..u]`. Escreva o código das funções `colocanafila`, `tiradafila`, `filavazia` e `filacheia`.
4. Suponha que, diferentemente da convenção adotada acima, a parte do vetor ocupada pela fila é `fila[0..u]`. Escreva o código das funções `colocanafila`, `tiradafila`, `filavazia` e `filacheia`.

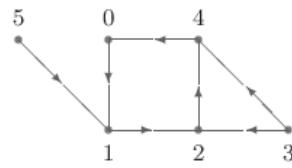
## Aplicação: distâncias

A ideia de fila aparece naturalmente no cálculo de [distâncias em um grafo](#). (Uma versão simplificada do problema é a [varredura por níveis](#) de uma [árvore binária](#).) Imagine  $N$  cidades numeradas de 0 a  $N-1$  e interligadas por estradas de mão única. As ligações entre as cidades são representadas por uma matriz  $A$  da seguinte maneira:

$A[i][j]$  vale 1 se existe estrada de  $i$  para  $j$

e vale 0 em caso contrário. Suponha que a matriz tem zeros na diagonal, embora isso seja irrelevante. Segue um exemplo em que  $N$  vale 6:

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 |
| 3 | 0 | 0 | 1 | 0 | 1 |
| 4 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 1 | 0 | 0 | 0 |



A [distância](#) de uma cidade  $c$  a uma cidade  $j$  é o menor número de estradas que precisamos percorrer para ir de  $c$  a  $j$ . (A distância de  $c$  a  $j$  é, em geral, diferente da distância de  $j$  a  $c$ .) Nossa [problema](#): dada uma cidade  $c$ ,

determinar a distância de  $c$  a cada uma das demais cidades.

As distâncias podem ser armazenadas em um vetor `dist`: a distância de  $c$  a  $j$  é `dist[j]`. É preciso tomar uma decisão de projeto para cuidar do caso em que é impossível ir de  $c$  a  $j$ . Poderíamos dizer que `dist[j]` é infinito nesse caso; mas é mais limpo e prático dizer que `dist[j]` vale  $N$ , pois nenhuma distância “real” pode ser maior que  $N-1$ . Se tomarmos  $c$  igual a 3 no exemplo acima, teremos

|         |   |   |   |   |   |   |
|---------|---|---|---|---|---|---|
| i       | 0 | 1 | 2 | 3 | 4 | 5 |
| dist[i] | 2 | 3 | 1 | 0 | 1 | 6 |

Eis a ideia de um algoritmo que usa uma *fila de cidades ativas* para resolver nosso problema. Uma cidade  $i$  é *ativa* se  $\text{dist}[i]$  já foi calculada mas as estradas que começam em  $i$  ainda não foram todas exploradas. Em cada [iteração](#), o algoritmo

tira da fila uma cidade  $i$  e coloca na fila todas as cidades vizinhas a  $i$  cujas distâncias ainda não foram calculadas.

Segue o código que implementa a ideia. (Veja antes um [rascunho em pseudocódigo](#).) Para simplificar, as variáveis `fila`, `p`, `u` e `dist` são [globais](#), ou seja, são definidas fora do código das funções.

```
#define N 100
int fila[N], int p, u;
int dist[N];

void criafila (void) {
 p = u = 0;
}

int filavazia (void) {
 return p >= u;
}

int tiradafila (void) {
 return fila[p++];
}

void colocanafila (int y) {
 fila[u++] = y;
}

// Esta função recebe uma matriz A
// que representa as interligações entre
// cidades 0..N-1 e preenche o vetor dist
// de modo que dist[i] seja a distância
// da cidade c à cidade i, para cada i.

void distancias (int A[][N], int c) {
 for (int j = 0; j < N; ++j) dist[j] = N;
 dist[c] = 0;
 criafila ();
 colocanafila (c);

 while (! filavazia ()) {
 int i = tiradafila ();
 for (int j = 0; j < N; ++j)
 if (A[i][j] == 1 && dist[j] >= N) {
 dist[j] = dist[i] + 1;
 colocanafila (j);
 }
 }
}
```

(Poderíamos operar a fila diretamente, sem invocar as funções de manipulação da fila. O resultado [seria mais curto e compacto](#), mas um pouco menos legível.)

## Exercícios 2

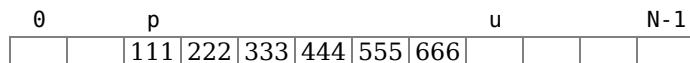
1. TAMANHO DA FILA. O espaço alocado para o vetor `fila` é suficiente? A instrução `colocanafila (j)` não provocará o transbordamento da fila?

2. ALOCAÇÃO DINÂMICA. Escreva uma versão da função `distancias` que tenha o número de cidades como um dos parâmetros. (Portanto, `N` não é definido por um `#define`.)
3. ☆ O ALGORITMO ESTÁ CORRETO. Prove que a função `distancias` está correta, ou seja, que calcula as distâncias corretas a partir de `c`. [[Dicas](#)]
4. Faça uma versão da função `distancias` que devolva a distância de uma cidade a a outra `b`.
5. Imagine um tabuleiro quadriculado com  $m \times n$  casas dispostas em  $m$  linhas e  $n$  colunas. Algumas casas estão livres e outras estão bloqueadas. As casas livres são marcadas com “-” e as bloqueadas com “#”. Há um robô na casa  $(1,1)$ , que é livre. O robô só pode andar de uma casa livre para outra. Em cada passo, só pode andar para a casa que está “ao norte”, “a leste”, “ao sul” ou “a oeste”. Ajude o robô a encontrar a saída, que está na posição  $(m,n)$ . (Sugestão: Faça uma moldura de casas bloqueadas.)

## Implementação circular

Na implementação que adotamos acima, a fila “anda para a direita” dentro do vetor que a abriga. Isso pode tornar difícil prever o valor que o [parâmetro `N`](#) deve ter para evitar o transbordamento da fila. Uma implementação “circular” pode ajudar a adiar o transbordamento, como mostraremos a seguir.

Suponha que os elementos da fila estão dispostos no vetor `fila[0..N-1]` de uma das seguintes maneiras: `fila[p..u-1]` ou `fila[p..N-1] fila[0..u-1]`.



Teremos sempre  $0 \leq p < N$  e  $0 \leq u < N$ , mas não podemos supor que  $p \leq u$ . A fila está

- *vazia* se  $u == p$  e
- *cheia* se  $u+1 == p$  ou  $u+1 == N$  e  $p == 0$  (ou seja, se  $(u+1) \% N == p$ ).

A posição anterior a  $p$  ficará sempre desocupada para que possamos distinguir uma fila cheia de uma vazia. Com essas convenções, a remoção de um elemento da fila pode ser escrita assim:

```
int tiradafila (void) {
 int x = fila[p++];
 if (p == N) p = 0;
 return x;
}
```

(desde que a fila não esteja vazia). A inserção de um objeto  $y$  na fila pode ser escrita assim:

```
void colocanafila (int y) {
 fila[u++] = y;
 if (u == N) u = 0;
}
```

(desde que a fila não esteja cheia).

## Exercícios 3

1. Imagine uma implementação circular de fila em um vetor `fila[0..9]` que contém

16 17 18 19 20 11 12 13 14 15

Suponha que o primeiro elemento da fila está na posição de índice 5 e o último está na posição de índice 4. Essa fila está cheia?

2. Considere a implementação circular de uma fila em um vetor. Escreva o código das funções `colocanafila`, `tiradafila`, `filavazia` e `filacheia`. Escreva uma função que devolva o comprimento (ou seja, o número de elementos) da fila.
3. MÓDULO DE IMPLEMENTAÇÃO DE FILA (VERSÃO 2). Escreva um [módulo](#) `filadeints.c` que faça uma implementação circular de uma fila de números inteiros em um vetor. O módulo deve conter as funções `criafila`, `colocanafila`, `tiradafila`, `filavazia` e `filacheia`. O vetor e as variáveis que indicam o início e o fim da fila devem ser globais no módulo. Escreva também uma [interface](#) `filadeints.h` para o módulo. (Inspire-se num dos [exercícios acima](#).)

## Implementação em vetor com redimensionamento

Nem sempre é possível prever a quantidade de espaço que deve ser reservada para a fila de modo a evitar transbordamentos. Se o vetor que abriga a fila foi alocado [dinamicamente](#) (com a função `malloc`), é possível resolver essa dificuldade [redimensionando](#) o vetor: toda vez que a fila ficar cheia, aloque um vetor maior e transfira a fila para esse novo vetor. Para evitar redimensionamentos frequentes, convém que o novo vetor seja pelo menos duas vezes maior que o original.

Eis um exemplo para o caso em que a fila contém números inteiros (e as variáveis `fila`, `p`, `u` e `N` são [globais](#)):

```
void redimensiona (void) {
 N *= 2;
 fila = realloc (fila, N * sizeof (int));
}
```

Uma versão [ad hoc](#) poderia ser escrita assim sem usar `realloc`:

```
void redimensiona (void) {
 N *= 2;
 int *novo;
 novo = malloc (N * sizeof (int));
 for (int i = p; i < u; i++)
 novo[i] = fila[i];
 free (fila);
 fila = novo;
}
```

Melhor ainda seria transferir `fila[p..u-1]` para `novo[0..u-p-1]` e reajustar as variáveis `p` e `u` de acordo.

## Exercícios 4

1. MÓDULO DE IMPLEMENTAÇÃO DE FILA (VERSÃO 3). Escreva um [módulo](#) `filadeints.c` que implemente uma fila de números inteiros num vetor com redimensionamento. O módulo deve conter as funções `criafila`, `colocanafila`, `tiradafila`, `filavazia`, `liberafila`. (Nessa versão, a função `filacheia` não faz sentido.) Trate os parâmetros da fila como variáveis globais do módulo. Escreva também uma [interface](#) `filadeints.h` para o módulo. [[Solução](#)]
2. MÓDULO DE IMPLEMENTAÇÃO DE FILA (VERSÃO 4). Escreva um módulo `filadechars.c` que implemente uma fila de [bytes](#). Veja o [exercício anterior](#).

# Fila implementada em uma lista encadeada

Como administrar uma fila armazenada em uma [lista encadeada](#)? Digamos que as células da lista são do tipo *celula*:

```
typedef struct reg {
 int conteudo;
 struct reg *prox;
} celula;
```

É preciso tomar algumas decisões de projeto sobre como a fila vai morar na lista. Vamos supor que nossa lista encadeada é *circular*: a última célula aponta para a primeira. Vamos supor também que a lista tem uma [célula-cabeça](#); essa célula não é removida nem mesmo se a fila ficar vazia. O primeiro elemento da fila fica na *segunda* célula e o último elemento fica na célula *anterior à cabeça*.

Um ponteiro *fi* aponta a célula-cabeça. A fila está *vazia* se *fi->prox == fi*. Uma fila vazia pode ser criada e inicializada assim:

```
celula *fi;
fi = malloc (sizeof (celula));
fi->prox = fi;
```

Podemos agora definir as funções de manipulação da fila. A remoção é fácil:

```
// Tira um elemento da fila fi e devolve
// o conteúdo do elemento removido.
// Supõe que a fila não está vazia.

int tiradafila (celula *fi) {
 celula *p;
 p = fi->prox; // o primeiro da fila
 int x = p->conteudo;
 fi->prox = p->prox;
 free (p);
 return x;
}
```

A inserção usa um truque sujo: armazena o novo elemento na célula-cabeça original e cria uma nova célula-cabeça:

```
// Coloca um novo elemento com conteúdo y
// na fila fi. Devolve o endereço da
// cabeça da fila resultante.

celula *colocanafila (int y, celula *fi) {
 celula *nova;
 nova = malloc (sizeof (celula));
 nova->prox = fi->prox;
 fi->prox = nova;
 fi->conteudo = y;
 return nova;
}
```

## Exercícios 5

1. Implemente uma fila em uma lista encadeada circular *sem* célula-cabeça. (Basta manter o endereço *u* da última célula; a primeira célula será apontada por *u->prox*. A lista encadeada estará vazia se e somente se *u == NULL*.)
2. Implemente uma fila em uma lista encadeada não circular com célula-cabeça. Será preciso manter o endereço *c* da célula-cabeça e o endereço *u* da última célula.
3. Implemente uma fila em uma lista encadeada não circular sem célula-cabeça. Será preciso manter

um ponteiro `p` para a primeira célula e um ponteiro `u` para a última.

4. MÓDULO DE IMPLEMENTAÇÃO DE FILA (VERSÃO 5). Escreva um módulo `filadeints.c` que implemente uma fila de números inteiros numa lista encadeada (escolha lista circular ou não circular, com ou sem cabeça). O módulo deve conter as funções `criafila`, `colocanafila`, `tiradafila`, `filavazia`, `liberafila`. Trate os parâmetros da fila como variáveis globais do módulo. Escreva também uma interface `filadeints.h` para o módulo. (Inspire-se num dos exercícios acima.)
5. LISTA DUPLAMENTE ENCADEADA. Implemente uma fila em uma lista duplamente encadeada sem célula-cabeça. Use um ponteiro `p` para a primeira célula e um ponteiro `u` para a última.
6. DEQUE. Uma *fila dupla* (= *deque*, pronuncia-se *deck*) permite inserção e remoção em qualquer das duas extremidades da fila. Implemente uma fila dupla (em um vetor ou uma lista encadeada) e escreva as funções de manipulação da estrutura.

---

Veja o verbete [Queue](#) na Wikipedia.

Atualizado em 2018-08-29

<https://www.ime.usp.br/~pf/algoritmos/>

Paulo Feofiloff

[DCC-IME-USP](#)



# Pilhas



Uma pilha é uma estrutura de dados que admite [remoção](#) de elementos e [inserção](#) de novos objetos. Mais especificamente, uma *pilha* (= *stack*) é uma estrutura sujeita à seguinte regra de operação: sempre que houver uma remoção,

o elemento removido é o que está na estrutura há *menos tempo*.

Em outras palavras, o primeiro objeto a ser inserido na pilha é o último a ser removido. Essa política é conhecida pela sigla LIFO (= *Last-In-First-Out*).

## Implementação em um vetor

Suponha que nossa pilha está armazenada em um vetor `pilha[0..N-1]`. (A natureza dos elementos do vetor é irrelevante: eles podem ser inteiros, bytes, ponteiros, etc.) Digamos que a parte do vetor ocupada pela pilha é

`pilha[0..t-1]`.

O índice `t` indica a primeira posição vaga da pilha e `t-1` é o índice do *topo* da pilha. A pilha está *vazia* se `t` vale 0 e *cheia* se `t` vale `N`. No exemplo da figura, os caracteres A, B, ..., H foram inseridos na pilha nessa ordem:

| 0 |   |   |   |   |   |   |   |  | t |  |  |  | N-1 |
|---|---|---|---|---|---|---|---|--|---|--|--|--|-----|
| A | B | C | D | E | F | G | H |  |   |  |  |  |     |

Para *remover*, ou *tirar*, um elemento da pilha — essa operação é conhecida como *desempilhar* (= *to pop*) — faça

```
x = pilha[--t];
```

Isso [equivale](#) ao par de instruções “`t -= 1; x = pilha[t];`”, nessa ordem. É claro que você só deve desempilhar se tiver certeza de que a pilha não está vazia.

Para *inserir*, ou *colocar*, um objeto `y` na pilha — a operação é conhecida como *empilhar* (= *to push*) — faça

```
pilha[t++] = y;
```

Isso equivale ao par de instruções “`pilha[t] = y; t += 1;`”, nessa ordem. Antes de empilhar, certifique-se de que a pilha não está cheia, para evitar um *transbordamento* (= *overflow*).

Para facilitar a leitura do código, é conveniente embalar essas operações em duas pequenas

funções. Se os objetos com que estamos lidando são do [tipo char](#), podemos escrever

```
char desempilha (void) {
 return pilha[--t];
}

void empilha (char y) {
 pilha[t++] = y;
}
```

Estamos supondo aqui que as variáveis `pilha` e `t` são [globais](#), ou seja, foram definidas fora do código das funções. (Para completar o pacote, precisaríamos de mais três funções: uma que crie uma pilha, uma que verifique se a pilha está vazia e uma que verifique se a pilha está cheia. Veja exercício [abaixo](#).)

## Exercícios 1

1. MÓDULO DE IMPLEMENTAÇÃO DE PILHA (VERSÃO 1). Escreva um [módulo](#) `pilhadechars.c` que implemente uma pilha de [caracteres ASCII](#). O módulo deve conter as funções `criapilha`, `empilha`, `desempilha`, `pilhavazia`, `pilhacheia`. Trate os parâmetros da pilha (o vetor `pilha` e o índice `t`) como variáveis globais do módulo. Escreva também uma [interface](#) `pilhadechars.h` para o módulo. [\[Solução\]](#)
2. Suponha que, diferentemente da convenção adotada no texto, a parte do vetor ocupada pela pilha é `pilha[1..t]`. Escreva o código das funções `empilha`, `desempilha`, `pilhavazia` e `pilhacheia`.
3. Escreva um algoritmo que use uma pilha para inverter a ordem das letras de cada palavra de uma [string ASCII](#), preservando a ordem das palavras. Por exemplo, para a string `ESTE EXERCICIO E MUITO FACIL` o resultado deve ser `ETSE OICICREXE E OTIUM LICAF`.
4. PERMUTAÇÕES PRODUZIDAS PELO DESEMPLHAR. Suponha que objetos 1,2,3,4 são colocados, nessa ordem, numa pilha inicialmente vazia. Depois de empilhar um objeto, você pode tirar zero ou mais elementos da pilha. Cada elemento desempilhado é impresso numa folha de papel. Por exemplo, a sequência de operações

`empilha 1, empilha 2, desempilha, empilha 3, desempilha, desempilha, empilha 4,`  
`desempilha,`

produz a impressão da sequência 2,3,1,4. Quais das 24 [permutações](#) de 1,2,3,4 podem ser obtidas dessa maneira?

5. [\[Sedgewick\]](#) O fragmento de programa abaixo manipula uma pilha de objetos do [tipo char](#). (A função `espio` devolve uma cópia do topo da pilha, mas não tira esse elemento da pilha.) Diga, em português, *o que* o fragmento faz. Escreva um fragmento de código equivalente que seja bem mais curto e mais simples.

```
if (pilhavazia ()) empilha ('B');
else {
 if (espio () != 'A') empilha ('B');
 else {
 while (!pilhavazia () && espio () == 'A')
 desempilha ();
 empilha ('B'); } }
```

## Aplicação: parênteses e colchetes

Considere o [problema](#) de decidir se uma dada [sequência](#) de parênteses e colchetes está bem-formada (ou seja, parênteses e colchetes são fechados na ordem inversa àquela em que foram abertos). Por exemplo, a sequência

( ( ) [ ( ) ] )

está bem-formada, enquanto ( [ ) ] está malformada. Suponha que a sequência de

parênteses e colchetes está armazenada em uma [string ASCII](#) s. (Como é hábito em C, o último caractere da string é `\0`.)

Usaremos uma pilha para resolver o problema. O algoritmo é [simples](#): examine a string da esquerda para a direita e empilhe os parênteses e colchetes esquerdos à espera de que apareçam os correspondentes parênteses e colchetes direitos.

Para simplificar, as variáveis pilha e t serão [globais](#). Suporemos também que o tamanho N do vetor que abriga a pilha é maior que o tamanho da string e portanto a pilha jamais transborda.

```
#define N 100
char pilha[N];
int t;

// Esta função devolve 1 se a string ASCII s
// contém uma sequência bem-formada de
// parênteses e colchetes e devolve 0 se
// a sequência é malformada.

int bemFormada (char s[])
{
 criapilha ();
 for (int i = 0; s[i] != '\0'; ++i) {
 char c;
 switch (s[i]) {
 case ')': if (pilhavazia ()) return 0;
 c = desempilha ();
 if (c != '(') return 0;
 break;
 case ']': if (pilhavazia ()) return 0;
 c = desempilha ();
 if (c != '[') return 0;
 break;
 default: empilha (s[i]);
 }
 }
 return pilhavazia ();
}

void criapilha (void) {
 t = 0;
}

void empilha (char y) {
 pilha[t++] = y;
}

char desempilha (void) {
 return pilha[--t];
}

int pilhavazia (void) {
 return t <= 0;
}
```

(Poderíamos operar a pilha diretamente, sem invocar as funções de manipulação da pilha. O resultado seria [bem mais curto e compacto](#), mas um pouco menos legível.)

## Exercícios 2

1. Dê uma definição formal de *sequência bem-formada* de parênteses e colchetes. Sugestão: definição recursiva.
2. A função bemFormada funciona corretamente se a sequência s tem apenas dois elementos? apenas

um? nenhum?

3. Considere a função `bemFormada`. Mostre que no início de cada iteração `s` está bem-formada se e somente se a sequência `pilha[0..t-1] s[i...]` estiver bem-formada.
4. Escreva uma versão da função `bemFormada` que aloque a pilha dinamicamente.
5. Digamos que nosso alfabeto contém apenas as letras `a`, `b` e `c`. Considere o seguinte conjunto de strings: `c`, `aca`, `bcb`, `abcba`, `bacab`, `aacaa`, `bbcbb`, ... Qualquer string desse conjunto tem a forma `WcM`, sendo `W` uma sequência de letras que só contém `a` e `b` e `M` o inverso de `W` (ou seja, `M` é `W` lido de trás para frente). Escreva um programa que decida se uma string `X` pertence ou não ao nosso conjunto, ou seja, decida se `X` é da forma `WcM`.

## Outra aplicação: notação polonesa

Na notação usual de expressões aritméticas, os operadores são escritos *entre* os operandos; por isso, a notação é chamada *infixa*. Na notação *posfixa*, ou *polonesa*, os operadores são escritos *depois* dos operandos. Eis alguns exemplos de expressões infixas e correspondentes expressões posfixas:

| infixa                      | posfixa                   |
|-----------------------------|---------------------------|
| $(A+B*C)$                   | $ABC*+$                   |
| $(A*(B+C)/D-E)$             | $ABC+*D/E-$               |
| $(A+B*(C-D*(E-F)-G*H)-I*3)$ | $ABCDEF-* -GH*- * +I3* -$ |
| $(A+B*C/D*E-F)$             | $ABC*D/E*+F-$             |
| $(A+B+C*D-E*F*G)$           | $AB+CD*+EF*G*-$           |
| $(A+(B-(C+(D-(E+F))))))$    | $ABCDEF+-+ -$             |
| $(A*(B+(C*(D+(E*(F+G))))))$ | $ABCDEF+*+*+*$            |

Note que os operandos (`A`, `B`, `C`, etc.) aparecem na mesma ordem na expressão infixada e na correspondente expressão posfixada. Note também que a notação posfixa *dispensa parênteses e regras de precedência* entre operadores (como a precedência de `*` sobre `+` por exemplo), que são indispensáveis na notação infixada.

Nosso [problema](#): traduzir para notação posfixada a expressão infixada armazenada em uma string `inf`. Para simplificar nossa vida, vamos supor que

- a expressão `inf` é válida e contém apenas [letras ASCII](#), parênteses, e os símbolos das quatro operações aritméticas,
- todas as operações (em particular `-` e `+`) têm *dois* operandos,
- os nomes das variáveis têm apenas uma letra cada,
- a expressão `inf` está embrulhada em um par de parênteses (ou seja, o primeiro caractere é `'('` e os dois últimos são `')'` e `\0`).

O algoritmo lê a expressão `inf` caractere-a-caractere e usa uma pilha para fazer a tradução. Todo parêntese esquerdo é colocado na pilha. Ao encontrar um parêntese direito, o algoritmo desempilha tudo até encontrar um parêntese esquerdo, que também é desempilhado. Ao encontrar um `+` ou um `-`, o algoritmo desempilha tudo até encontrar um parêntese esquerdo, que não é desempilhado. Ao encontrar um `*` ou um `/`, o algoritmo desempilha tudo até encontrar um parêntese esquerdo ou um `+` ou um `-`. Constantes e variáveis são transferidos diretamente de `inf` para a expressão posfixada. (Veja um [rascunho em pseudocódigo](#).)

As variáveis `pilha` e `t` são [globais](#). Vamos supor que o tamanho `N` da pilha é maior que o tamanho da string `inf`, e portanto não precisamos nos preocupar com pilha cheia. Como a expressão `inf` está embrulhada em parênteses, não precisamos nos preocupar com pilha vazia.

```
#define N 100
char pilha[N];
int t;

// Esta função recebe uma expressão infixada inf
```

```

// e devolve a correspondente expressão posfixa.

char *infixaParaPosfixa (char *inf) {
 int n = strlen (inf);
 char *posf;
 posf = malloc ((n+1) * sizeof (char));
 criapilha ();
 empilha (inf[0]); // empilha '('

 int j = 0;
 for (int i = 1; inf[i] != '\0'; ++i) {
 switch (inf[i]) {
 char x;
 case '(': empilha (inf[i]);
 break;
 case ')': x = desempilha ();
 while (x != '(') {
 posf[j++] = x;
 x = desempilha ();
 }
 break;
 case '+':
 case '-': x = desempilha ();
 while (x != '(') {
 posf[j++] = x;
 x = desempilha ();
 }
 empilha (x);
 empilha (inf[i]);
 break;
 case '*':
 case '/': x = desempilha ();
 while (x != '(' && x != '+' && x != '-') {
 posf[j++] = x;
 x = desempilha ();
 }
 empilha (x);
 empilha (inf[i]);
 break;
 default: posf[j++] = inf[i];
 }
 }
 posf[j] = '\0';
 return posf;
}

```

(Poderíamos operar a pilha diretamente, sem invocar as funções de manipulação da pilha. O resultado seria [mais curto e compacto](#), mas um pouco menos legível.)

Veja o resultado da aplicação da função `infixaParaPosfixa` à expressão infixada `(A*(B*C+D))`. A tabela registra os valores das variáveis no início de cada iteração:

| <u>inf[0..i-1]</u> | <u>pilha[0..t-1]</u> | <u>posf[0..j-1]</u> |
|--------------------|----------------------|---------------------|
| (                  | (                    |                     |
| (A                 | (                    | A                   |
| (A*                | (*                   | A                   |
| (A*(               | (*(                  | A                   |
| (A*(B              | (*(                  | AB                  |
| (A*(B*             | (*(*                 | AB                  |
| (A*(B*C            | (*(*                 | ABC                 |
| (A*(B*C+           | (*(+                 | ABC*                |
| (A*(B*C+D          | (*(+                 | ABC*D               |
| (A*(B*C+D)         | (*                   | ABC*D+              |
|                    |                      | ABC*D+*             |

## Exercícios 3

1. Use a função `infixaParaPosfixa` para converter a expressão infixa  $(A+B)*D+E/(F+A*D)+C$  na expressão posfixa equivalente.
2. Na função `infixaParaPosfixa`, que tamanho a pilha pode atingir no pior caso, em função de  $n$ ? Em outras palavras, qual o valor máximo da variável  $t$  no pior caso? Que acontece se o número de parênteses esquerdos na expressão for limitado (menor que 6, por exemplo)?
3. Reescreva a função `infixaParaPosfixa` sem supor que a expressão infixa está embrulhada em um par de parênteses.
4. Reescreva a função `infixaParaPosfixa` e as funções de manipulação da pilha de modo que o vetor pilha seja alocado [dinamicamente](#).
5. Reescreva a função `infixaParaPosfixa` supondo que a expressão pode ter *colchetes* além de parênteses.
6. Reescreva a função `infixaParaPosfixa` supondo que a expressão pode não ser válida.
7. VALOR DE EXPRESSÃO POLONESA. Suponha que `posf` é uma [string ASCII](#) não vazia que armazena uma expressão aritmética em notação posfixa. Suponha que `posf` contém somente os operadores `+`, `-`, `*` e `/` (todos exigem *dois* operandos). Suponha também que a expressão não tem constantes e que todos os nomes de variáveis na expressão consistem em uma única letra maiúscula. Suponha ainda que temos um vetor `valor` que dá os valores das variáveis (todos inteiros):  
`valor[0]` é o valor da variável A,  
`valor[1]` é o valor da variável B, etc.

Escreva uma função que calcule o valor da expressão `posf`. Cuidado com divisões por zero!

## Implementação em vetor com redimensionamento

Nem sempre é possível prever a quantidade de espaço que deve ser reservada para abrigar a pilha de modo a evitar transbordamentos. Podemos, então, redimensionar o vetor toda vez que a pilha ficar cheia (como [já fizemos com a implementação de fila](#)).

## Exercícios 4

1. MÓDULO DE IMPLEMENTAÇÃO DE PILHA (VERSÃO 2). Escreva um [módulo](#) `pilhadechars.c` que implemente uma pilha de caracteres ASCII num vetor com [redimensionamento](#). O módulo deve conter as funções `criapilha`, `empilha`, `desempilha`, `pilhavazia` e `liberapilha`. Trate os parâmetros da pilha como variáveis globais do módulo. Escreva também uma [interface](#) `pilhadechars.h` para o módulo.

## Pilha implementada em uma lista encadeada

Como implementar uma pilha de [caracteres ASCII](#) em uma [lista encadeada](#)? Digamos que as células da lista são do tipo `celula`:

```
typedef struct reg {
 char conteudo;
 struct reg *prox;
} celula;
```

Decisões de projeto: Nossa lista terá uma célula-cabeça (e portanto a primeira célula da lista não faz parte da pilha). Uma variável [global](#) `pi` apontará a cabeça da lista:

```
celula *pi;
```

O topo da pilha ficará na *segunda* célula e não na última (por quê?). As funções de criação e manipulação da pilha podem então ser escritas assim:

```
void criapilha (void) {
 pi = malloc (sizeof (celula)); // cabeça
 pi->prox = NULL;
}

void empilha (char y) {
 celula *nova;
 nova = malloc (sizeof (celula));
 nova->conteudo = y;
 nova->prox = pi->prox;
 pi->prox = nova;
}

char desempilha (void) {
 celula *p;
 p = pi->prox;
 char x = p->conteudo;
 pi->prox = p->prox;
 free (p);
 return x;
}
```

(Como de hábito, a função `desempilha` não deve ser invocada se a pilha estiver vazia.)

## Exercícios 5

1. Implemente um pilha em uma lista encadeada *sem* célula-cabeça. A pilha será dada pelo endereço da primeira célula da lista (que é o topo da pilha).
2. Reescreva as funções [bemFormada](#) e [infixaParaPosfixa](#) armazenando a pilha em uma lista encadeada.

## Apêndice: A pilha de execução de um programa

Todo programa C consiste em uma ou mais funções (sendo `main` a primeira função a ser executada). Para administrar as invocações das funções, o computador usa uma *pilha de execução*. (Veja o verbete [Call stack](#) na Wikipedia.) A operação pode ser descrita conceitualmente da seguinte maneira:

Ao encontrar a invocação de uma função, o computador cria um novo “espaço de trabalho”, que contém todos os parâmetros e todas as variáveis locais da função. Esse espaço de trabalho é colocado na pilha de execução (por cima do espaço de trabalho que invocou a função) e a execução da função começa (confinada ao seu espaço de trabalho). Quando a execução da função termina, o seu espaço de trabalho é removido da pilha e descartado. O espaço de trabalho que estiver agora no topo da pilha é reativado e a execução é retomada do ponto em que havia sido interrompida. (Veja [Julia's drawings: What's the stack?](#))

Considere o seguinte exemplo:

```
int G (int a, int b) {
 int x;
 x = a + b;
 return x;
}

int F (int i, j, k) {
 int x;
 x = /*2*/ G (i, j) /*3*/;
```

```

 x = x + k;
 return x;
 }

 int main (void) {
 int i, j, k, y;
 i = 111; j = 222; k = 444;
 y = /*1*/ F (i, j, k) /*4*/;
 printf ("%d\n", y);
 return EXIT_SUCCESS;
 }

```

A execução do programa prossegue da seguinte maneira:

- Um espaço de trabalho é criado para a função `main` e colocado na pilha de execução. O espaço contém as variáveis locais `i`, `j`, `k` e `y`. A execução de `main` começa.
- No ponto 1, a execução de `main` é temporariamente interrompida e um espaço de trabalho para a função `F` é colocado na pilha. Esse espaço contém os parâmetros `i`, `j`, `k` da função (com valores 111, 222 e 444 respectivamente) e a variável local `x`. Começa então a execução de `F`.
- No ponto 2, a execução de `F` é interrompida e um espaço de trabalho para a função `G` é colocado na pilha. Esse espaço contém os parâmetros `a` e `b` da função (com valores 111 e 222 respectivamente) e a variável local `x`. Em seguida, começa a execução de `G`.
- Quando a execução de `G` termina, a função devolve 333. O espaço de trabalho de `G` é removido da pilha e descartado. O espaço de trabalho de `F` (que agora está no topo da pilha de execução) é reativado e a execução é retomada no ponto 3. A primeira instrução executada é “`x = 333;`”.
- Quando a execução de `F` termina, a função devolve 777. O espaço de trabalho de `F` é removido da pilha e descartado. O espaço de trabalho de `main` (que agora está no topo da pilha) é reativado e a execução é retomada no ponto 4. A primeira instrução executada é “`y = 777;`”.

No nosso exemplo, `F` e `G` são funções distintas. Mas tudo funcionaria da mesma maneira se `F` e `G` fossem idênticas, ou seja, se `F` fosse uma função recursiva.

## Exercícios 6

1. Escreva uma função iterativa que simule o comportamento da seguinte função recursiva. Use uma pilha.

```

int TTT (int x[], int n) {
 if (n == 0) return 0;
 if (x[n] > 0) return x[n] + TTT (x, n-1);
 else return TTT (x, n-1);
}

```

2. PILHA DE EXECUÇÃO DE PROGRAMAS. (Este exercício não só simula o funcionamento da pilha de execução como também o processamento das diretivas `#include` e `#define` do pré-processador do compilador C.) Escreva um programa que receba um [arquivo de texto ASCII](#) e grave outro arquivo de texto ASCII como especificado a seguir. Cada linha do arquivo de entrada contém palavras separadas por espaços; as palavras que começam com `#` são *especiais* e as outras são *normais*. O arquivo de saída conterá as palavras normais, em uma só linha, numa certa ordem. Suponha, por exemplo, que o arquivo de entrada contém as seguintes linhas (os números no início das linhas servem apenas de referência e não fazem parte do arquivo):

```

0 #4 aaa #2
1 bbb
2 CC #4 DDD #1 ee
3 FF #2 #4
4 GG hhh

```

Então o arquivo de saída, deverá conter

```
GG hhh aaa CC GG hhh DDD bbb ee
```

Como o exemplo sugere, as palavras especias são substituídas pela linha do arquivo de entrada cujo número é dado depois de "#", e isso deve ser feito *recursivamente*. Para tornar o exercício mais interessante, não use funções recursivas.

---

Veja o verbete [Stack \(data structure\)](#) na Wikipedia.

---

Atualizado em 2018-08-30

<https://www.ime.usp.br/~pf/algoritmos/>

*Paulo Feofiloff*

[\*DCC-IME-USP\*](#)



# Busca em vetor ordenado

*"Binary search is to algorithms what a wheel is to mechanics:  
It is simple, elegant, and immensely important."*

— Udi Manber, *Introduction to Algorithms*

*"Binary search is a notoriously tricky algorithm to program correctly.  
It took seventeen years after its invention until the first correct version of binary search was published!"*

— Steven Skiena, *The Algorithm Design Manual*

Este capítulo estuda o seguinte [problema](#) de busca: encontrar um dado número em um vetor ordenado de números. Mais especificamente, dado um inteiro  $x$  e um vetor [crescente](#)  $v[0..n-1]$  de inteiros,

encontrar um índice  $m$  tal que  $v[m] == x$ .

É claro que nem toda [instância](#) desse problema tem solução. Considere, por exemplo, as instâncias em que  $v[i-1] < x < v[i]$  para algum  $i$ .

Comecemos por adotar uma formulação mais geral e mais interessante do problema: determinar o lugar no vetor onde  $x$  *deveria* estar. Mais precisamente, dado  $x$  e um vetor crescente  $v[0..n-1]$ , queremos *encontrar um índice j tal que*

$$v[j-1] < x \leq v[j].$$

Para obter uma solução do problema original, basta comparar  $x$  com  $v[j]$ .

Para tirar o máximo proveito dessa formulação do problema, devemos permitir que a solução  $j$  assuma os valores  $0$  e  $n$ . Nesses dois casos, a expressão  $v[j-1] < x \leq v[j]$  deve ser interpretada com bom senso: se  $j == 0$ , a expressão se reduz a  $x \leq v[0]$ , pois  $v[-1]$  não está definido; se  $j == n$ , a expressão se reduz a  $v[n-1] < x$ , pois  $v[n]$  não está definido. Tudo se passa como se nosso vetor tivesse um componente imaginário  $v[-1]$  com valor  $-\infty$  e um componente imaginário  $v[n]$  com valor  $+\infty$ .

No exemplo a seguir, se  $x$  vale 555, a solução  $j$  do problema é 4; se  $x$  vale 800, a solução é 8; e se  $x$  vale 1000, a solução é 13.

| $0$ | $n-1$ |     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 111 | 222   | 333 | 444 | 555 | 555 | 666 | 777 | 888 | 888 | 888 | 999 | 999 | 999 |

Vamos examinar dois algoritmos para o problema: um óbvio mas lento e outro sofisticado e muito mais rápido.

## Exercícios 1

1. Escreva uma função que decida se um vetor  $v[0..n-1]$  está em ordem crescente. Depois, critique

o código a seguir.

```
int verifica (int v[], int n) {
 int anterior = v[0], sim = 1;
 for (int i = 1; i < n && sim; ++i) {
 if (anterior > v[i]) sim = 0;
 anterior = v[i];
 }
 return sim; }
```

2. Critique a seguinte formulação do problema de busca: dado  $x$  e um vetor crescente  $v[0..n-1]$ , encontrar um índice  $j$  tal que  $v[j-1] \leq x \leq v[j]$ . Critique a formulação construída em torno da expressão  $v[j-1] < x < v[j]$ .

## Busca sequencial

Comecemos com um algoritmo óbvio, que examina um a um todos os elementos do vetor. Eis uma implementação do algoritmo:

```
// Esta função recebe um inteiro x e um vetor
// crescente v[0..n-1] e devolve um índice j
// em 0..n tal que v[j-1] < x <= v[j].
int
buscaSequencial (int x, int n, int v[]) {
 int j = 0;
 while (j < n && v[j] < x)
 ++j;
 return j;
}
```

Quantas iterações a função faz? Ou melhor, quantas comparações faz entre  $x$  e elementos de  $v$ ? No pior caso,  $x$  é comparado com cada elemento do vetor e portanto o número total de comparações é  $n$ . Assim, por exemplo, se o tamanho do vetor for multiplicado por 100, o número de comparações também será multiplicado por 100.

O consumo de tempo da função é proporcional ao número de comparações que envolvem  $x$ , e portanto proporcional a  $n$  no pior caso.

É possível resolver o problema em menos tempo? É possível resolver o problema sem comparar  $x$  com cada um dos elementos do vetor? A resposta é afirmativa, como veremos a seguir.

## Exercícios 2

1. INVARIANTES. Quais são os invariantes do processo iterativo na função buscaSequencial? Use os invariantes para mostrar que a função está correta. [Solução]

2. Discuta a seguinte versão da função buscaSequencial:

```
int buscaSeq2 (int x, int n, int v[]) {
 int j;
 for (j = 0; j < n; ++j)
 if (x <= v[j]) break;
 return j; }
```

3. Critique a seguinte versão da função buscaSequencial:

```
int buscaSeq3 (int x, int n, int v[]) {
 int j = 0;
 while (v[j] < x && j < n) ++j;
 return j; }
```

4. Discuta a seguinte versão recursiva da função buscaSequencial:

```
int buscaSeq4 (int x, int n, int v[]) {
```

```

if (n == 0) return 0;
if (x > v[n-1]) return n;
return buscaSeq4 (x, n-1, v); }

```

5. Escreva uma versão da função `buscaSequencial` que satisfaça uma especificação diferente da nossa: dado  $x$  e um vetor crescente  $v[0..n-1]$ , encontrar um índice  $j$  tal que  $v[j] \leq x < v[j+1]$ .

## Busca binária

Existe um algoritmo muito mais rápido que a busca sequencial. Ele é análogo ao método que se usa para encontrar um nome em uma lista telefônica antiga, aquela que parece um livro. É claro que a ideia só funciona porque o vetor está *ordenado*.

```

// Esta função recebe um inteiro x
// e um vetor crescente v[0..n-1]
// e devolve um índice j em 0..n
// tal que v[j-1] < x <= v[j].

int
buscaBinaria (int x, int n, int v[]) {
 int e = -1, d = n; // atenção!
 while (e < d-1) {
 int m = (e + d)/2;
 if (v[m] < x) e = m;
 else d = m;
 }
 return d;
}

```

Simples e limpo! Os nomes das variáveis não foram escolhidos ao acaso:  $e$  lembra “esquerda”,  $m$  lembra “meio” e  $d$  lembra “direita”. O resultado da divisão por 2 na expressão  $(e+d)/2$  é automaticamente [truncado](#), pois as variáveis são do tipo `int`. Por exemplo, se  $e$  vale 6 e  $d$  vale 9, a expressão  $(e+d)/2$  vale 7.

| 0   |     | $e$ |     | $d$ |     | $n-1$                       |
|-----|-----|-----|-----|-----|-----|-----------------------------|
| 111 | 222 | 333 | 444 | 555 | 555 | 666 777 888 888 888 999 999 |

A ideia da busca binária ( $= binary search$ ) é um verdadeiro [ovo de Colombo](#). Essa ideia é o ponto de partida de algoritmos eficientes para muitos problemas.

## Exercícios 3

1. Para evitar valores de índices fora do intervalo  $0..n-1$ , poderíamos trocar a linha “ $e = -1; d = n$ ” da função `buscaBinaria` pelas três linhas a seguir. Discuta essa variante do código.

```

if (v[n-1] < x) return n;
if (x <= v[0]) return 0;
// agora v[0] < x <= v[n-1]
e = 0; d = n-1;

```

2. Responda as seguintes perguntas sobre a função `buscaBinaria`. Que acontece se “`while (e < d-1)`” for trocado por “`while (e < d)`”? ou por “`while (e <= d-1)`”? Que acontece se trocarmos “ $(e+d)/2$ ” por “ $(e+d-1)/2$ ” ou por “ $(e+d+1)/2$ ” ou por “ $(d-e)/2$ ”? Que acontece se “`if (v[m] < x)`” for trocado por “`if (v[m] <= x)`”? Que acontece se “ $e = m$ ” for trocado por “ $e = m+1$ ” ou por “ $e = m-1$ ”? Que acontece se “ $d = m$ ” for trocado por “ $d = m+1$ ” ou por “ $d = m-1$ ”?

3. Suponha que  $n = 9$  e  $v[i] = i$  para cada  $i$ . Execute `buscaBinaria` com vários valores de  $x$ . Repita o exercício com  $n = 15$  e vários valores de  $x$ . Repita o exercício com  $n = 14$  e  $x = 9$ .

4. Execute a função buscaBinaria com  $n = 16$ . Quais os possíveis valores de  $m$  na primeira iteração? Quais os possíveis valores de  $m$  na segunda iteração? Na terceira? Na quarta?
5. Na função buscaBinaria, é verdade que  $m$  está em  $0..n-1$  sempre que a instrução `if (v[m] < x)` é executada? (Note que  $e$  e  $d$  podem não estar em  $0..n-1$ .)
6. Confira a validade da seguinte afirmação: quando  $n+1$  é uma potência de 2, a expressão  $(e + d)$  é divisível por 2, quaisquer que sejam  $v$  e  $x$ .

## A função buscaBinaria está correta?

Para entender a função buscaBinaria, basta [verificar](#) que no início de cada repetição do `while`, imediatamente antes da comparação de  $e$  com  $d-1$ , vale a relação

$$v[e] < x \leq v[d].$$

Essa relação é, portanto, [invariante](#). (O algoritmo foi *construído* a partir desse invariante!) Note a semelhança entre o invariante e o objetivo  $v[j-1] < x \leq v[j]$  que estamos perseguindo. O invariante vale, em particular, no início da *primeira* iteração: basta imaginar que  $v[-1]$  vale  $-\infty$  e  $v[n]$  vale  $+\infty$ .

No início de cada iteração, em virtude do invariante, temos  $v[e] < v[d]$  e portanto  $e < d$ , uma vez que o vetor é crescente. No início da última iteração, temos  $e \geq d-1$  e portanto  $e = d-1$ . A relação invariante garante agora que, ao devolver  $d$ , a função está se comportando como prometeu!

Em cada iteração temos  $e < m < d$  (por quê?). Logo, tanto  $d - m$  quanto  $m - e$  são estritamente menores que  $d - e$ . Portanto, a sequência de valores da expressão  $d - e$  é [estritamente decrescente](#). É por isso que a execução do algoritmo termina, mais cedo ou mais tarde.

## Exercícios 4

1. INVARIANTE. Mostre que no início de cada iteração de buscaBinaria (ou seja, imediatamente antes da comparação de  $e$  com  $d-1$ ) tem-se  $v[e] < x \leq v[d]$ .
2. VARIANTES DO ALGORITMO. Escreva uma versão da função buscaBinaria que tenha o seguinte invariante: no início de cada iteração,  $v[e-1] < x \leq v[d]$ . Repita com  $v[e-1] < x \leq v[d+1]$ . Repita com  $v[e] < x \leq v[d+1]$ .
3. VETORES GRANDES. Se o número de elementos do vetor estiver próximo de [INT\\_MAX](#), o código da busca binária pode descarrilar, em virtude de um [overflow aritmético](#), ao calcular  $m = (e + d)/2$ . Como evitar isso?

## Desempenho da busca binária

Quantas iterações a função buscaBinaria executa ao processar um vetor com  $n$  elementos? No início da primeira iteração,  $d - e$  vale aproximadamente  $n$ . No início da segunda, vale aproximadamente  $n/2$ . No início da terceira, aproximadamente  $n/4$ . No início da  $k+1$ -ésima, aproximadamente  $n/2^k$ . Quando  $k$  passar de  $\log n$ , o valor da expressão  $n/2^k$  fica menor que 1 e a execução do algoritmo para. (Veja o exercício de cálculo de  $\log n$  em [outra página](#).) Logo, o número de iterações é aproximadamente

$$\log n$$

tanto no pior caso quanto no melhor. O número aproximado de comparações de  $x$  com elementos do vetor também é  $\log n$ . Veja alguns exemplos quando  $n$  é potência de 2:

|          |    |      |       |         |          |     |
|----------|----|------|-------|---------|----------|-----|
| $n$      | 32 | 1024 | 32768 | 1048576 | 33554432 | ... |
| $\log n$ | 5  | 10   | 15    | 20      | 25       | ... |

O valor de  $\log n$  cresce muito devagar pois  $\log$  transforma multiplicações em adições. Por exemplo, se a busca em um vetor de tamanho  $n$  exige  $C$  comparações, então a busca em um vetor de tamanho  $2n$  exigirá apenas  $C+1$  comparações, a busca em um vetor de tamanho  $8n$  exigirá apenas  $C+3$  comparações, e a busca em um vetor de tamanho  $100n$  exigirá menos que  $C+7$  comparações.

O *consumo de tempo* da função `buscaBinaria` é proporcional ao número de comparações de  $x$  com elementos de  $v$ . O consumo de tempo é, portanto, proporcional a  $\log n$ .

## Exercícios 5

- Suponha que o vetor  $v$  tem 511 elementos e que  $x$  não está no vetor. Quantas vezes, exatamente, a função `buscaBinaria` comparará  $x$  com um elemento do vetor? Suponha agora que o vetor  $v$  tem 50000 elementos e que  $x$  não está no vetor. Quantas vezes, aproximadamente, a função `buscaBinaria` comparará  $x$  com um elemento do vetor?
- Se preciso de  $t$  segundos para fazer uma busca binária em um vetor com  $n$  elementos, de quanto tempo preciso para fazer uma busca em  $n^2$  elementos?
- NÚMERO EXATO DE ITERAÇÕES. Mostre que, no pior caso, a função `buscaBinaria` faz exatamente  $1 + \lfloor \log(n) \rfloor$  comparações de  $x$  com elementos do vetor, sendo `lg(n)` o *piso* de  $\log n$ .

## Exercícios 6

Implementações de busca binária exigem cuidado e atenção aos detalhes. É muito fácil escrever uma implementação que dá respostas erradas ou “entra em loop”. Os exercícios abaixo discutem versões alternativas da função `buscaBinaria`, diferentes da que examinamos acima. Todas procuram encontrar  $x$  em  $v[0..n-1]$ . Todas produzem (ou deveriam produzir) um índice  $j$  em  $0..n$  tal que  $v[j-1] < x \leq v[j]$ .

- Mostre que a seguinte versão de `buscaBinaria` está correta. Ela é quase tão bonita quanto a versão discutida acima.

```
e = 0; d = n;
while (e < d) { // v[e-1] < x <= v[d]
 m = (e + d)/2;
 if (v[m] < x) e = m+1;
 else d = m;
} // e == d
return d;
```

- Mostre que a seguinte versão de `buscaBinaria` está correta. Acho que ela é um pouco mais feia que as versões anteriores.

```
e = 0; d = n-1;
while (e <= d) { // v[e-1] < x <= v[d+1]
 m = (e + d)/2;
 if (v[m] < x) e = m+1;
 else d = m-1; } // e == d+1
return d+1;
```

- A seguinte versão de `buscaBinaria` está correta?

```
e = -1; d = n-1;
while (e < d) {
 m = (e + d)/2;
 if (v[m] < x) e = m;
 else d = m-1; }
return d+1;
```

- A seguinte versão de `buscaBinaria` está correta?

```
e = -1; d = n-1;
while (e < d) {
```

```

m = (e + d + 1)/2;
if (v[m] < x) e = m;
else d = m-1;
return d+1;

```

## Exercícios 7

1. Escreva uma implementação de busca binária que receba um inteiro  $x$  e um vetor crescente  $v[0..n-1]$  e devolva  $j$  tal que  $v[j-1] \leq x < v[j]$ . Quais os possíveis valores de  $j$ ?
2. Escreva uma implementação de busca binária que procure  $x$  num vetor crescente  $v[0..n]$ . Escreva uma implementação da busca binária que procure  $x$  em  $v[1..n]$ .
3. Escreva uma implementação de busca binária que procure  $x$  num vetor *decrescente*  $v[0..n-1]$ .
4. Modifique o código da função buscaBinaria de modo a resolver o problema de busca enunciado no [início deste capítulo](#).

## Versão recursiva da busca binária

Para formular uma versão recursiva da busca binária é preciso generalizar ligeiramente o problema, trocando  $v[0..n-1]$  por  $v[a..b]$ . A ponte entre a formulação básica e a generalizada é uma “função-embalagem” (= *wrapper-function*) buscaBinaria2 que apenas repassa o serviço para uma função recursiva bb.

```

// Esta função recebe um vetor crescente
// v[0..n-1] e um inteiro x e devolve um
// índice j em 0..n tal que
// v[j-1] < x <= v[j].

int
buscaBinaria2 (int x, int n, int v[]) {
 return bb (x, -1, n, v);
}

// Recebe um vetor crescente v[e+1..d-1] e
// um inteiro x tal que v[e] < x <= v[d] e
// devolve um índice j em e+1..d tal que
// v[j-1] < x <= v[j].

static int
bb (int x, int e, int d, int v[]) {
 if (e == d-1) return d;
 else {
 int m = (e + d)/2;
 if (v[m] < x)
 return bb (x, m, d, v);
 else
 return bb (x, e, m, v);
 }
}

```

(A palavra-chave static está aí apenas para indicar que a função bb tem caráter auxiliar, e não deve ser invocada diretamente pelo usuário do algoritmo de busca binária.)

Qual a profundidade da recursão na função bb? Ou seja, quantas vezes bb chama a si mesma?  
Resposta: cerca de  $\log n$  vezes.

## Exercícios 8

1. CORREÇÃO. Prove que a função recursiva bb está correta. Observe que a condição  $v[e] < x \leq v[d]$  garante que  $e \leq d-1$ . Observe também que cada invocação de bb satisfaz as condições estabelecidas na documentação da função.

2. Discuta a seguinte versão alternativa da função buscabinaria2:

```
int buscaBinaria3 (int x, int n, int v[]) {
 if (v[n-1] < x) return n;
 if (x <= v[0]) return 0;
 return bb (x, 0, n-1, v); }
```

3. Discuta a seguinte versão alternativa da função bb. Escreva a documentação da função.

```
int bb2 (int x, int e, int d, int v[]) {
 if (e > d) return d+1;
 else {
 int m = (e + d)/2;
 if (v[m] < x)
 return bb2 (x, m+1, d, v);
 else
 return bb2 (x, e, m-1, v); } }
```

## Exercícios 9

1. TWO SUM. Escreva um algoritmo eficiente para o seguinte problema: dados um vetor crescente  $v[0..n-1]$  de números inteiros e um número inteiro  $t$ , encontrar dois índices distintos  $i$  e  $j$  tais que  $v[i] + v[j] = t$ .
2. Escreva uma função que receba um vetor inteiro *estritamente* crescente  $v[0..n-1]$  e devolva um índice  $i$  entre  $0$  e  $n-1$  tal que  $v[i] == i$  ou, se tal  $i$  não existe, devolve  $-1$ . A função não deve fazer mais que  $\log n$  comparações envolvendo elementos de  $v$ .
3. Escreva uma função eficiente que receba inteiros estritamente positivos  $k$  e  $n$  e calcule  $k^n$ . Quantas multiplicações sua função executa?
4. VETOR DE STRINGS. Suponha que  $v[0..n-1]$  é um vetor de [strings ASCII](#) em [ordem lexicográfica](#). Escreva uma função que receba uma string  $x$  e devolva um índice  $j$  tal que a  $v[j-1] < x \leq v[j]$ , onde “ $<$ ” e “ $\leq$ ” se referem à ordem lexicográfica.
5. VETOR DE STRUCTS. Suponha que cada elemento do vetor  $v[0..n-1]$  é uma [struct](#) com dois campos: o nome de um aluno e o número do aluno. Suponha que o vetor está em ordem crescente de números. Escreva uma função de busca binária que receba o número de um aluno e devolva o seu nome. Se o número não está no vetor, a função deve devolver a [string](#) vazia.
6. INTERVALOS DISJUNTOS. Suponha dado um conjunto de intervalos fechados disjuntos entre si. Suponha que os extremos dos intervalos são números inteiros. Escreva um programa que receba um inteiro como argumento e determine o intervalo a que o inteiro pertence. Por exemplo, se os intervalos são  $1643..2033$ ,  $5532..7643$ ,  $8999..10332$ ,  $5666653..5669321$ , o número  $9122$  pertence ao terceiro intervalo e  $8122$  não pertence a intervalo algum.
7. Familiarize-se com a função `bsearch` da biblioteca [stdlib](#).

---

Veja o verbete [Binary search algorithm](#) na Wikipedia.

---

Veja também os capítulos 4 e 5 do “[Programming Pearls](#)”.

Atualizado em 2018-09-15

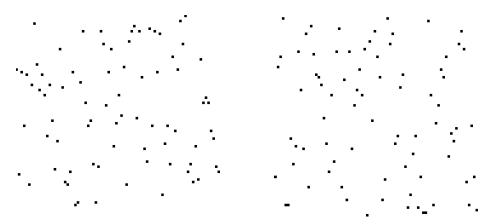
<https://www.ime.usp.br/~pf/algoritmos/>

*Paulo Feofiloff*

[DCC-IME-USP](#)



# Ordenação: algoritmos elementares



“Primeiro, coloque os números em ordem crescente.  
Depois decidiremos o que fazer.”  
— estratégia algorítmica

Este capítulo trata do seguinte [problema](#) fundamental: *Permutar (ou seja, rearranjar) os elementos de um vetor  $v[0..n-1]$  de tal modo que eles fiquem em ordem crescente*, isto é, de tal forma que tenhamos  $v[0] \leq v[1] \leq \dots \leq v[n-1]$ .

| 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 111 | 999 | 222 | 999 | 333 | 888 | 444 | 777 | 555 | 666 | 555 |

|     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 111 | 222 | 333 | 444 | 555 | 555 | 666 | 777 | 888 | 999 | 999 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

Discutiremos aqui dois algoritmos simples para o problema. Outros capítulos descreverão algoritmos mais sofisticados e bem mais rápidos.

Embora o problema tenha sido apresentado em termos da ordenação de um vetor, ele faz sentido para qualquer estrutura linear, como uma [lista encadeada](#), por exemplo.

## Exercício 1

1. IMPORTANTE. Escreva uma função que verifique se um vetor  $v[0..n-1]$  está em ordem crescente. (Este exercício põe em prática a estratégia de *escrever os testes* antes de inventar algoritmos para o problema.)
2. Critique o código da seguinte função, que promete decidir se o vetor  $v[0..n-1]$  está em ordem crescente.

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

3. Critique o código da seguinte função, que promete decidir se o vetor  $v[0..n-1]$  está em ordem crescente.

```
int verifica (int v[], int n) {
 int sim;
 for (int i = 1; i < n; i++)
 if (v[i-1] <= v[i]) sim = 1;
 else {
 sim = 0;
```

```

 break; }
return sim; }

```

4. Escreva uma função que rearranje um vetor  $v[0..n-1]$  de modo que ele fique em ordem estritamente crescente.
5. Escreva uma função que receba um inteiro  $x$  e um vetor  $v[0..n-1]$  de inteiros em ordem crescente e insira  $x$  no vetor de modo a manter a ordem crescente.

## Ordenação por inserção

Eis um algoritmo de ordenação muito popular, frequentemente usado para colocar em ordem um baralho de cartas (veja página [aula da Khan Academy](#)):

```

// Esta função rearranja o vetor v[0..n-1]
// em ordem crescente.

void
insercao (int n, int v[])
{
 for (int j = 1; j < n; ++j) {
 int x = v[j];
 int i;
 for (i = j-1; i >= 0 && v[i] > x; --i)
 v[i+1] = v[i];
 v[i+1] = x;
 }
}

```

(Compare o `for` interno com o [algoritmo de inserção discutido no capítulo Vetores](#).) Para entender o funcionamento do algoritmo, basta observar que no início de cada repetição do `for` externo, imediatamente antes da comparação “ $j < n$ ”,

1. o vetor  $v[0..n-1]$  é uma permutação do vetor original e
2. o vetor  $v[0..j-1]$  está em ordem crescente.

Essas condições [invariantes](#) são trivialmente verdadeiras no início da primeira iteração, quando  $j$  vale 1. No início da última iteração,  $j$  vale  $n$  e portanto o vetor  $v[0..n-1]$  está em ordem, como desejado. (Note que a última iteração é abortada logo no início, pois a condição “ $j < n$ ” é falsa.)

| 0   | crescente | $j-1$ | $j$ | $n-1$                       |
|-----|-----------|-------|-----|-----------------------------|
| 444 | 555       | 555   | 666 | 777 222 999 222 999 222 999 |

**Desempenho do algoritmo.** Quantas vezes a função `insercao` compara  $x$  com um elemento do vetor? Mais precisamente, quantas vezes, no máximo, a função `insercao` executa a comparação “ $v[i] > x$ ”? Esse número está diretamente relacionado com a sequência de valores de  $i$  ao longo da execução da função. Para cada valor de  $j$ , a variável  $i$  assume, no [pior caso](#), os valores  $j-1, \dots, 0$ . A seguinte tabela mostra esses valores explicitamente:

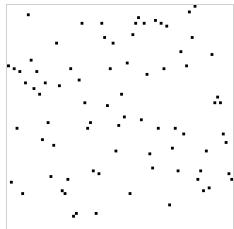
| j        | i                           |          |
|----------|-----------------------------|----------|
| 1        | 0                           | 1        |
| 2        | 1 0                         | 2        |
| 3        | 2 1 0                       | 3        |
| $\vdots$ | $\vdots$                    | $\vdots$ |
| $n-1$    | $n-2 \ n-3 \ \dots \ 1 \ 0$ | $n-1$    |

A terceira coluna da tabela dá o número de diferentes valores de  $i$  na linha. Portanto, o número de execuções da comparação “ $v[i] > x$ ” é, no pior caso, igual à soma da terceira coluna. Essa soma é  $n(n-1)/2$ , ou seja, um pouco menos que a metade de

$$n^2.$$

Agora considere o *tempo* que a função `insercao` consome para fazer o serviço. Esse tempo é *proporcional ao número de execuções da comparação* “ $v[i] > x$ ” (pense!). Logo, o consumo de tempo da função cresce, no pior caso, como o quadrado do tamanho do vetor. Se um vetor de tamanho  $N$  consome  $T$  segundos então um vetor de tamanho  $2N$  consumirá  $4T$  segundos e um vetor de tamamanho  $10N$  consumirá  $100T$  segundos!

Essa análise mostra que o algoritmo de inserção é lento demais para ordenar vetores grandes. Graças à sua simplicidade, entretanto, o algoritmo é muito útil no caso de vetores pequenos. Além disso, no [melhor caso](#) (por exemplo, se o vetor já está “quase ordenados”), o desempenho do algoritmo é muito bom: o número de comparações não passa de  $n$  e portanto o consumo de tempo é proporcional a  $n$ .



**Animações.** A animação à esquerda (copiada da [Wikimedia](#)) mostra a ordenação por inserção de um vetor  $v[0..79]$  de números positivos. Cada elemento  $v[i]$  do vetor é representado pelo ponto de coordenadas  $(i, v[i])$ . Há várias outras animações interessantes na rede WWW:

- [Insertion sort](#), na Wikipedia.
- [Animação de algoritmos de ordenação](#), de Nicholas André Pinho de Oliveira.
- [Insert-sort com dança folclórica Romena](#), Universidade Sapientia (Romênia).

## Exercícios 2

1. Escreva uma versão *recursiva* do algoritmo de ordenação por inserção.
2. Na função `insercao`, troque a comparação “ $v[i] > x$ ” por “ $v[i] \geq x$ ”. A nova função continua correta?
3. Que acontece se trocarmos “`for (j = 1)`” por “`for (j = 0)`” no código da função `insercao`? Que acontece se trocarmos “`v[i+1] = x`” por “`v[i] = x`”?
4. Critique a seguinte implementação do algoritmo de ordenação por inserção:

```
for (int j = 1; j < n; ++j) {
 int x = v[j];
 for (int i = j-1; i >= 0 && v[i] > x; --i) {
 v[i+1] = v[i];
 v[i] = x; } }
```

5. Critique a seguinte implementação do algoritmo de ordenação por inserção:

```
int i, j, x;
for (j = 1; j < n; ++j) {
 for (i = j-1; i >= 0 && v[i] > v[i+1]; --i) {
 x = v[i]; v[i] = v[i+1]; v[i+1] = x; } }
```

6. A seguinte implementação do algoritmo de ordenação por inserção está correta?

```
for (int j = 1; j < n; ++j) {
 int x = v[j];
 int i = j - 1;
 while (i >= 0 && v[i] > x) {
 v[i+1] = v[i];
 --i;
 }
 v[i+1] = x; }
```

7. IMPORTANTE. Escreva uma versão do algoritmo de inserção que rearranje em ordem crescente um vetor  $v[p..r]$  e tenha o seguinte invariante: no início de cada iteração, o vetor  $v[k+1..r]$  é crescente.
8. BUSCA BINÁRIA. O papel do `for` interno na função `insercao` é encontrar o ponto onde  $v[j]$  deve ser inserido em  $v[0..j-1]$ . Considere fazer isso com uma [busca binária](#). Analise o resultado.
9. PIOR CASO. Descreva e analise uma [instância](#) de pior caso para o algoritmo de inserção, ou seja, um vetor  $v[0..n-1]$  que leva o algoritmo a executar o maior número possível de comparações.

10. MELHOR CASO. Descreva e analise uma instância de melhor caso para o algoritmo de inserção, ou seja, um vetor  $v[0..n-1]$  que leva o algoritmo a executar o menor número possível de comparações.
11. MOVIMENTAÇÃO DE DADOS. Quantas vezes, no pior caso, o algoritmo de inserção copia um elemento do vetor de um lugar para outro? Quantas vezes isso ocorre no melhor caso?
12. TESTES COM VETOR ALEATÓRIO. Escreva um programa que teste, experimentalmente, a correção de sua implementação do algoritmo de inserção. Use [permutações aleatórias de 1..n](#) para os testes. Compare o resultado da ordenação com  $1..n$ . Aproveite a ocasião para cronometrar o algoritmo de ordenação (use a função `clock` da [biblioteca time](#)).
13. Escreva uma função com protótipo `insert_sort (int v[], int n)` que rearranje um vetor  $v[1..n]$  (note os índices!) em ordem crescente. (Basta invocar `insercao` da maneira correta.)
14. ORDEM DECRESCENTE. Escreva uma função que permuta os elementos de um vetor  $v[0..n-1]$  de modo que eles fiquem em ordem decrescente. Inspire-se no algoritmo de ordenação por inserção.

## Ordenação por seleção

Esta seção trata de outro algoritmo de ordenação bem conhecido. Ele usa a seguinte estratégia: seleciona o menor elemento do vetor, depois o segundo menor, depois o terceiro menor, e assim por diante:

```
// Esta função rearranja o vetor v[0..n-1]
// em ordem crescente.

void
selecao (int n, int v[])
{
 for (int i = 0; i < n-1; ++i) {
 int min = i;
 for (int j = i+1; j < n; ++j)
 if (v[j] < v[min]) min = j;
 int x = v[i]; v[i] = v[min]; v[min] = x;
 }
}
```

Para entender por que o algoritmo está correto, basta observar que no início de cada repetição do `for` externo, imediatamente antes da comparação “ $i < n-1$ ”, valem os seguintes invariantes:

1. o vetor  $v[0..n-1]$  é uma permutação do vetor original,
2.  $v[0..i-1]$  está em ordem crescente e
3.  $v[i-1] \leq v[i..n-1]$ .

A tradução do terceiro invariante para linguagem humana é a seguinte:  $v[0..i-1]$  contém todos os elementos “pequenos” do vetor original e  $v[i..n-1]$  contém todos os elementos “grandes”.

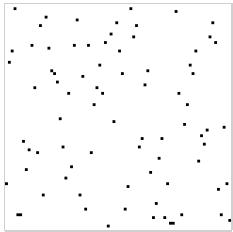
| 0        | crescente | $i-1$ | $i$ | $n-1$                       |
|----------|-----------|-------|-----|-----------------------------|
| 110      | 120       | 120   | 130 | 140 999 666 999 666 999 666 |
| pequenos |           |       |     | grandes                     |
|          |           |       |     |                             |

Os três invariantes garantem que no início de cada iteração  $v[0], \dots, v[i-1]$  já estão em suas posições definitivas. No início da última iteração, o vetor está em ordem crescente, como desejado.

**Desempenho do algoritmo.** Tal como o de ordenação por inserção, o algoritmo de ordenação por seleção faz cerca de  $n^2/2$  comparações entre elementos do vetor no [pior caso](#).

Diferentemente do algoritmo de inserção, entretanto, o algoritmo de seleção também faz cerca

de  $n^2/2$  comparações no [melhor caso](#) (por exemplo, quando o vetor já está “quase ordenado”). Assim, o consumo de tempo do algoritmo é sempre proporcional a  $n^2$ . Em vista dessa análise (e de outras observações que faremos nos exercícios) o algoritmo de inserção é preferível ao de seleção.



**Animações.** A animação à esquerda (copiada da [Wikipedia](#)) mostra a ordenação por seleção de um vetor  $v[0..79]$  de números positivos. Cada elemento  $v[i]$  do vetor é representado pelo ponto de coordenadas  $(i, v[i])$ . Há várias outras animações interessantes na rede WWW:

- [Selection sort](#), na Wikipedia.
- [Animação de algoritmos de ordenação](#), de Nicholas André Pinho de Oliveira.
- [Select-sort com dança cigana](#), Universidade Sapientia (Romênia).

## Exercícios 3

1. Escreva uma versão *recursiva* do algoritmo de ordenação por seleção.
2. Na função [selecao](#), que acontece se trocarmos “`for (i = 0)`” por “`for (i = 1)`”? Que acontece se trocarmos “`for (i = 0; i < n-1)`” por “`for (i = 0; i < n)`”?
3. Na função [selecao](#), troque a comparação “`v[j] < v[min]`” por “`v[j] <= v[min]`”. A nova função continua correta?
4. PIOR CASO. Descreva e analise uma [instância](#) de pior caso para o algoritmo de seleção, ou seja, um vetor  $v[0..n-1]$  que leva o algoritmo a executar o maior número possível de comparações.
5. MELHOR CASO. Descreva e analise uma instância de melhor caso para o algoritmo de seleção, ou seja, um vetor  $v[0..n-1]$  que leva o algoritmo a executar o menor número possível de comparações.
6. MOVIMENTAÇÃO DE DADOS. Quantas vezes, no pior caso, o algoritmo de seleção copia um elemento do vetor de um lugar para outro? Quantas vezes isso ocorre no melhor caso?
7. ORDENAÇÃO DECRESCENTE. Escreva uma função que permuta os elementos de um vetor inteiro  $v[0..n-1]$  de modo que eles fiquem em ordem decrescente. Inspire-se no algoritmo de seleção.

## Exercícios 4

1. ORDENAÇÃO DE STRINGS. Escreva uma função que coloque em [ordem lexicográfica](#) um vetor de [strings ASCII](#). Use o algoritmo de inserção.
2. ORDENAÇÃO DE ARQUIVO DIGITAL. Escreva uma função que rearranje as linhas de um [arquivo ASCII](#) em [ordem lexicográfica](#). Trate o arquivo como um vetor de [bytes](#). Compare com o utilitário [sort](#).
3. ORDENAÇÃO DE STRUCTS. Suponha que cada elemento de um vetor é um registro com dois campos: um é um inteiro e outro uma [string ASCII](#):

```
struct registro {int aa; char *bb};
```

Escreva uma função que rearranje o vetor de modo que os campos `aa` fiquem em ordem crescente. Escreva outra função que rearranje o vetor de modo que os campos `bb` fiquem em [ordem lexicográfica](#).

4. DESAFIO. Invente um algoritmo de ordenação que seja *mais rápido* que o de inserção e o de seleção.
5. EMBARALHAMENTO ALEATÓRIO? Considere a seguinte antítese do problema de ordenação: fazer um embaralhamento aleatório dos elementos de um vetor  $v[0..n-1]$ , ou seja, rearranjar os elementos do vetor de modo que todas as [permutações](#) sejam igualmente prováveis. Discuta e critique a seguinte solução (ela foi [usada na distribuição européia do Windows 7 da Microsoft](#)):

```

void insercao_aleatoria (int n, int v[]) {
 for (int j = 1; j < n; ++j) {
 int x = v[j];
 int i;
 for (i = j-1; i >= 0 ; --i) {
 if (rand() > RAND_MAX/2)
 v[i+1] = v[i];
 else break;
 }
 v[i+1] = x;
 }
}

```

6. ORDENAÇÃO DE LISTA ENCADEADA. Escreva uma função que ordene uma [lista encadeada](#). Inspire-se no algoritmo de inserção para vetores. (Sua função precisa devolver alguma coisa?).
7. ORDENAÇÃO DE LISTA ENCADEADA. Escreva um função para ordenar uma lista encadeada. Imita o algoritmo de seleção para vetores. (Sua função precisa devolver alguma coisa?).

## Exercícios 5: aplicações

Muitos problemas podem ser reduzidos à ordenação de um vetor, ou seja, podem ser resolvidos com o auxílio de um algoritmo de ordenação (não necessariamente um dos algoritmos discutidos neste capítulo).

1. ANAGRAMAS. Uma palavra é *anagrama* de outra se a [sequência](#) de letras de uma é permutação da sequência de letras da outra. Por exemplo, “aberto” é anagrama de “rebato”. Digamos que duas palavras são *equivalentes* se uma é anagrama da outra. Uma *classe de equivalência* de palavras é um conjunto de palavras duas a duas equivalentes. Escreva um programa que receba um [arquivo ASCII](#) contendo palavras (uma por linha) e extraia desse arquivo uma classe de equivalência máxima. Aplique o seu programa a um [arquivo que contém todas as palavras do português brasileiro com diacríticos removidos](#). (O arquivo é grande; portanto, o seu programa deverá ser muito eficiente.)
2. VALORES DISTINTOS. Dado um vetor  $v[0..n-1]$  de números inteiros, determinar quantos números distintos há no vetor (ou seja, determinar o tamanho do *conjunto* de elementos do vetor).
3. MEDIANA. Seja  $v[0..n-1]$  um vetor de números inteiros, todos diferentes entre si. A *mediana* do vetor é um elemento do vetor que seja maior que metade dos elementos do vetor e menor que (a outra) metade dos elementos. Mais precisamente, a mediana de  $v[0..n-1]$  é um número  $m$  que seja igual a algum elemento do vetor e estritamente maior que exatamente  $\lfloor (n-1)/2 \rfloor$  elementos do vetor. (Não confunda mediana com média. Por exemplo, a média de 1 2 99 é 51, enquanto a mediana é 2.) Escreva um algoritmo que encontre a mediana de um vetor  $v[0..n-1]$  de números diferentes entre si.
4. ESCALONAMENTO DE TAREFAS. Suponha que  $n$  tarefas devem ser processadas em um único processador. Dadas as durações  $t_1, \dots, t_n$  das tarefas, em que ordem elas devem ser processadas para minimizar o *tempo médio* de conclusão de uma tarefa?

## Estabilidade da ordenação

Um algoritmo de ordenação é *estável* (= *stable*) se não altera a posição relativa dos elementos que têm o mesmo valor. Digamos, por exemplo, que um vetor de [números do tipo double](#) é ordenado com base na parte inteira dos números, ignorando a parte fracionária. Se o algoritmo de ordenação for estável, os números que têm a mesma parte inteira continuarão na mesma ordem em que estavam originalmente. Na seguinte figura, a primeira linha mostra o vetor original e a segunda, o vetor ordenado:

|      |      |      |      |      |      |      |      |
|------|------|------|------|------|------|------|------|
| 44.0 | 55.1 | 55.2 | 66.0 | 22.9 | 11.0 | 22.5 | 33.0 |
| 11.0 | 22.9 | 22.5 | 33.0 | 44.0 | 55.1 | 55.2 | 66.0 |

Eis outro exemplo. Digamos que cada elemento do vetor é um *struct* com dois campos: o primeiro contém o nome de uma pessoa e o segundo contém o ano de nascimento da pessoa. Suponha que o vetor original tem dois “João da Silva”, primeiro o que nasceu em 1990 e depois o que nasceu em 1995. Se o vetor for ordenado por um algoritmo estável com base no primeiro

campo, os dois “João da Silva” continuarão na mesma ordem relativa: primeiro o de 1990 e depois o de 1995.

O algoritmo de inserção é estável? O algoritmo de seleção é estável?

## Exercícios 6

1. O algoritmo de ordenação por inserção é estável?
2. O algoritmo de ordenação por seleção é estável?
3. Na função [insercao](#), troque a comparação “ $v[i] > x$ ” por “ $v[i] \geq x$ ”. A função modificada faz uma ordenação estável?

---

Veja os verbetes [Sorting algorithm](#), [Insertion sort](#) e [Selection sort](#) na Wikipedia.

---

Veja o capítulo 11 do [Programming Pearls](#) de Jon Bentley.

Atualizado em 2018-09-24

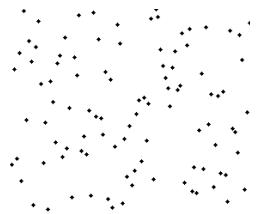
<https://www.ime.usp.br/~pf/algoritmos/>

*Paulo Feofiloff*

[DCC-IME-USP](#)



# Mergesort: ordenação por intercalação



Nosso problema: Rearranjar um vetor  $v[0 \dots n-1]$  de tal modo que ele fique em ordem crescente, ou seja, de tal modo que tenhamos  $v[0] \leq \dots \leq v[n-1]$ .

Já analisamos [alguns algoritmos simples](#) para esse problema que consomem tempo quadrático, ou seja, tempo proporcional a  $n^2$ . Vamos examinar agora um algoritmo mais sofisticado e muito mais rápido.

## Intercalação de vetores ordenados

Antes de resolver nosso problema principal é preciso resolver o seguinte *problema da intercalação* (= *merge*): dados vetores crescentes  $v[p \dots q-1]$  e  $v[q \dots r-1]$ , rearranjar  $v[p \dots r-1]$  em ordem crescente.

| p   |     |     |     |     |     |     |  | q-1 | q   | r-1 |     |
|-----|-----|-----|-----|-----|-----|-----|--|-----|-----|-----|-----|
| 111 | 333 | 555 | 555 | 777 | 999 | 999 |  | 222 | 444 | 777 | 888 |

É fácil resolver o problema em tempo proporcional ao quadrado do tamanho do vetor  $v[p \dots r-1]$ : basta ignorar o estado ordenado das duas “metades” e ordenar o vetor  $v[p \dots r-1]$ . Mas é possível fazer algo bem melhor. Para isso, será preciso usar uma “área de trabalho”, digamos  $w[0 \dots r-p-1]$ , do mesmo tipo e mesmo tamanho que o vetor  $v[p \dots r-1]$ .

```
// A função recebe vetores crescentes v[p..q-1]
// e v[q..r-1] e rearranja v[p..r-1] em ordem
// crescente.
```

```
static void
intercala1 (int p, int q, int r, int v[])
{
 int *w; // 1
 w = malloc ((r-p) * sizeof (int)); // 2
 int i = p, j = q; // 3
 int k = 0; // 4

 while (i < q && j < r) { // 5
 if (v[i] <= v[j]) w[k++] = v[i++]; // 6
 else w[k++] = v[j++]; // 7
 } // 8
 while (i < q) w[k++] = v[i++]; // 9
 while (j < r) w[k++] = v[j++]; // 10
 for (i = p; i < r; ++i) v[i] = w[i-p]; // 11
 free (w); // 12
```

}

(A palavra-chave `static` está aí apenas para indicar que a função `intercalal` tem caráter auxiliar, e não deve ser invocada diretamente pelo usuário final do algoritmo de ordenação.)

**Desempenho.** A função `intercalal` consiste essencialmente em *movimentar* elementos do vetor `v` de um lugar para outro (primeiro de `v` para `w` e depois de `w` para `v`). A função executa

$$2n$$

dessas movimentações, sendo `n` o tamanho do vetor `v[p..r-1]`. O tempo que `intercalal` consome é proporcional ao número de movimentações. Portanto, o consumo de tempo da função é proporcional a `n`. Assim, o algoritmo é [linear](#).

## Exercícios 1

1. Escreva uma função que receba vetores disjuntos `x[0..m-1]` e `y[0..n-1]`, ambos em ordem crescente, e produza um vetor `z[0..m+n-1]` que contenha o resultado da intercalação dos dois vetores dados. Escreva duas versões da função: uma iterativa e uma recursiva.
2. A função `intercalal` está correta quando `p` é igual a `q`, ou seja, quando o vetor `v[p..q-1]` está vazio? E quando o vetor `v[q..r-1]` está vazio?
3. Troque as linhas 9 a 11 da função `intercalal` pelas duas linhas a seguir. A função continua correta?

```
while (i < q) w[k++] = v[i++];
for (i = p; i < j; ++i) v[i] = w[i-p];
```

4. Troque o bloco de linhas 5 a 8 da função `intercalal` pelas linhas abaixo. Critique o efeito da troca.

```
while (i < q && j < r) {
 if (v[i] <= v[j]) w[k++] = v[i++];
 if (v[i] > v[j]) w[k++] = v[j++]; }
```

5. Na função `intercalal`, troque o bloco de linhas 3 a 10 pelas linhas abaixo. A função continua correta?

```
i = p; j = q;
for (k = 0; k < r-p; ++k) {
 if (j >= r || (i < q && v[i] <= v[j]))
 w[k] = v[i++];
 else
 w[k] = v[j++]; }
```

6. Na função `intercalal`, troque o bloco de linhas 5 a 10 pelas linhas abaixo. A função continua correta?

```
while (k < r-p) {
 while (i < q && v[i] <= v[j])
 w[k++] = v[i++];
 while (j < r && v[j] <= v[i])
 w[k++] = v[j++]; }
```

7. INVARIANTES. Quais são os [invariantes](#) do primeiro `while` na função `intercalal`?
8. Mostre que o consumo de tempo da função `intercalal` pode não ser proporcional ao número de comparações entre elementos do vetor.
9. Critique a seguinte função de intercalação. Ela insere cada elemento de `v[q..r-1]` em `v[p..q-1]` como o [algoritmo de inserção](#). (Observe que a função faz a intercalação *in loco*, ou seja, sem usar vetor auxiliar.)

```
while (q < r) {
 int x = v[q], int i;
 for (i = q-1; i >= p && v[i] > x; --i)
 v[i+1] = v[i];
 v[i+1] = x;
 q++; }
```

10. A seguinte solução do problema da intercalação está correta? Quais os invariantes do `while`?

(Observe que a função faz a intercalação *in loco*, ou seja, sem usar vetor auxiliar.) Qual o consumo de tempo?

```
int i, k, x;
i = p;
while (i < q && q < r) {
 if (v[i] >= v[q]) {
 x = v[q];
 for (k = q - 1; k >= i; --k)
 v[k+1] = v[k];
 v[i] = x;
 ++q;
 }
 ++i;
}
```

11. DESAFIO: INTERCALAÇÃO IN LOCO. Invente um função de intercalação tão eficiente quanto `intercalal` que resolva o problema *in loco*, ou seja, sem usar um vetor auxiliar.
12. Um algoritmo de intercalação é *estável* se não altera a posição relativa de elementos iguais. A função `intercalal` discutida acima é estável? E se a comparação “`v[i] <= v[j]`” for trocada por “`v[i] < v[j]`”?
13. INTERCALAÇÃO DE LISTAS ENCADEADAS. Digamos (para efeito deste exercício) que uma LEC é uma [lista encadeada](#) (sem cabeça) que contém uma sequência crescente de números inteiros. Escreva uma função que intercale duas LECs, produzindo assim uma terceira. Sua função não deve alocar novas células na memória, mas reaproveitar as células das duas listas dadas.
14. UNIÃO DE LISTAS ENCADEADAS. Digamos que uma LEC é uma [lista encadeada](#) (sem cabeça) que contém uma sequência *estritamente* crescente de números inteiros. (Portanto, uma LEC representa um *conjunto* de números.) Escreva uma função que faça a *união* de duas LECs. A lista resultante deve ser uma LEC e deve ser construída com as células das duas listas dadas.

## Intercalação com sentinelas

[Sedgewick](#) escreve o algoritmo de intercalação de uma maneira mais interessante. Primeiro, copia o vetor `v[p..q-1]` para o espaço de trabalho `w[0..q-p-1]`; depois, copia `v[q..r-1]` para o espaço `w[q-p..r-p-1]` *em ordem inversa*. Com isso, a metade esquerda de `w` serve de sentinela para a metade direita durante o processo de intercalação, e vice-versa. Assim, a intercalação de `w[0..q-p-1]` com `w[q-p..r-p-1]` pode ser feita com base na comparação `w[i] <= w[j]` sem que seja preciso verificar, a cada iteração, as condições `i < q-p` e `j ≥ q-p`.

```
// Esta função recebe vetores crescentes
// v[p..q-1] e v[q..r-1] e rearranja
// v[p..r-1] em ordem crescente.

static void
intercala2 (int p, int q, int r, int v[])
{
 int i, j, *w;
 w = malloc ((r-p) * sizeof (int));

 for (i = p; i < q; ++i) w[i-p] = v[i];
 for (j = q; j < r; ++j) w[r-p+q-j-1] = v[j];
 i = 0; j = r-p-1;
 for (int k = p; k < r; ++k)
 if (w[i] <= w[j]) v[k] = w[i++];
 else v[k] = w[j--];
 free (w);
}
```

Tal como a versão anterior, esta consome tempo proporcional ao tamanho do vetor `v[p..r-1]`.

## Exercícios 2

1. Discuta o seguinte código alternativo de `intercala2`:

```

for (i = 0, k = p; k < q; ++i, ++k)
 w[i] = v[k];
for (j = r-p-1, k = q; k < r; --j, ++k)
 w[j] = v[k];
i = 0; j = r-p-1;
for (k = p; k < r; ++k)
 if (w[i] <= w[j]) v[k] = w[i++];
 else v[k] = w[j--];

```

2. [Sedgewick 8.6] Mostre que a função `intercala2` discutida acima não é estável. Que modificações é preciso introduzir para que ela se torne estável?

## Algoritmo Mergesort

Agora podemos usar qualquer das funções `intercala` discutidas acima para escrever um algoritmo de ordenação, um algoritmo rápido que rearranje qualquer vetor  $v[p..r-1]$  em ordem crescente.

O algoritmo é recursivo. A base da recursão é o caso  $p \geq r-1$ ; nesse caso, o vetor tem no máximo 1 elemento e portanto não é preciso fazer nada.

```

// A função mergesort rearranja o vetor
// v[p..r-1] em ordem crescente.

void
mergesort (int p, int r, int v[])
{
 if (p < r-1) {
 int q = (p + r)/2;
 mergesort (p, q, v);
 mergesort (q, r, v);
 intercala (p, q, r, v);
 }
}

```

(O resultado da divisão por 2 na expressão  $(p+r)/2$  é automaticamente truncado. Por exemplo,  $(3+6)/2$  vale 4.) Se  $p < r-1$ , o código de `mergesort` reduz a instância  $v[p..r-1]$  do problema ao par de instâncias  $v[p..q-1]$  e  $v[q..r-1]$ . Essas duas instâncias são estritamente menores que a instância original, uma vez que  $p < q < r$  graças à condição  $p < r-1$ . Assim, por hipótese de indução, o vetor  $v[p..q-1]$  estará em ordem crescente no fim da linha 3 e o vetor  $v[q..r-1]$  estará em ordem crescente no fim da linha 4. Portanto, no fim da linha 5, de acordo com a documentação da função `intercala`, o vetor  $v[p..r-1]$  estará em ordem crescente, como prometeu a documentação de `mergesort`.

| 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 111 | 999 | 222 | 999 | 333 | 888 | 444 | 777 | 555 | 666 | 555 |
| 111 | 999 | 222 | 999 | 333 | 888 | 444 | 777 | 555 | 666 | 555 |
| 111 | 999 | 222 | 999 | 333 | 888 | 444 | 777 | 555 | 666 | 555 |
| 111 | 999 | 222 | 333 | 999 | 444 | 777 | 888 | 555 | 555 | 666 |
| 111 | 222 | 333 | 999 | 999 | 444 | 555 | 555 | 666 | 777 | 888 |
| 111 | 222 | 333 | 444 | 555 | 555 | 666 | 777 | 888 | 999 | 999 |

Para rearranjar em ordem crescente um vetor  $v[0..n-1]$ , como quer a formulação original do

[problema](#), basta executar `mergesort (θ, n, v)`.

## Exercícios 3

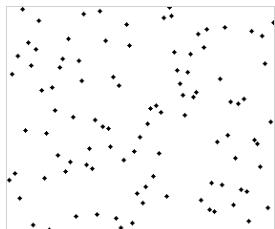
1. Mostre que  $p < q < r$  no fim da linha 2 de `mergesort`.
2. Que acontece se trocarmos  $"(p+r)/2"$  por  $"(p+r-1)/2"$  no código da função `mergesort`? Que acontece se trocarmos  $"(p+r)/2"$  por  $"(p+r+1)/2"$ ?
3. Submeta um vetor indexado por  $1..4$  à função `mergesort`. Teremos a seguinte sequência de invocações da função: (observe a indentação):

```
mergesort (1,5,v)
 mergesort (1,3,v)
 mergesort (1,2,v)
 mergesort (2,3,v)
 mergesort (3,5,v)
 mergesort (3,4,v)
 mergesort (4,5,v)
```

Repita o exercício com um vetor indexado por  $1..5$ .

4. PEGADINHA. Quais são os invariantes da função `mergesort`?
5. A função `mergesort` é [estável](#)?
6. OVERFLOW. Se o tamanho do vetor estiver próximo de [INT\\_MAX](#), a execução da função `mergesort` pode descarrilar na linha  $"q = (p + r)/2;"$  em virtude de um [overflow](#) aritmético. Como evitar isso?
7. TESTES COM VETOR ALEATÓRIO. Escreva um programa que teste, experimentalmente, a correção de sua implementação do algoritmo Mergesort. Use [permutações aleatórias de  \$1..n\$](#)  para os testes. Compare o resultado da ordenação com  $1..n$ .

## Animações do Mergesort



A animação à esquerda (copiada da [Wikipedia](#)), mostra a ordenação de um vetor  $v[0..99]$  que contém uma permutação aleatória de  $0..99$ . Cada elemento  $v[i]$  é representado pelo ponto de coordenadas  $(i, v[i])$ . Há várias outras animações na rede WWW:

- [Comparison Sorting Algorithms](#), de David Galles (University of San Francisco).
- [Sorting Algorithms Animations](#).
- Animação de [15 algoritmos de ordenação](#), de Timo Bingmann, no YouTube.
- [Animação de algoritmos de ordenação](#), de Nicholas André Pinho de Oliveira.
- [Merge-sort com dança folclórica da Transilvânia](#), Universidade Sapientia (Romênia).

## Exercícios 4

1. A seguinte função promete rearranjar  $v[p..r-1]$  em ordem crescente. A função está correta?

```
void mergesort1 (int p, int r, int v[]) {
 if (p < r-1) {
 int q = (p + r) / 2;
 mergesort1 (p, q, v);
 mergesort1 (q, r, v);
 intercala (p, q+1, r, v); } }
```

2. A seguinte função promete rearranjar  $v[p..r-1]$  em ordem crescente. A função está correta?

```

void mergesort2 (int p, int r, int v[]) {
 if (p < r) {
 int q = (p + r) / 2;
 mergesort2 (p, q, v);
 mergesort2 (q, r, v);
 intercala (p, q, r, v); } }

```

3. Esta função está correta? Ela promete rearranjar  $v[p..r-1]$  em ordem crescente.

```

void mergesort3 (int p, int r, int v[]) {
 if (p < r-1) {
 int q = (p + r - 1) / 2;
 mergesort3 (p, q, v);
 mergesort3 (q, r, v);
 intercala (p, q, r, v); } }

```

4. Esta função rearranja  $v[p..r-1]$  em ordem crescente? E se trocarmos “ $(p+r)/2$ ” por “ $(p+r+1)/2$ ”?

```

void mergesort4 (int p, int r, int v[]) {
 if (p < r-1) {
 int q = (p + r) / 2;
 mergesort4 (p, q-1, v);
 mergesort4 (q-1, r, v);
 intercala (p, q-1, r, v); } }

```

5. Esta função rearranja  $v[p..r-1]$  em ordem crescente?

```

void mergesort5 (int p, int r, int v[]) {
 if (p < r-1) {
 q = r - 2;
 mergesort5 (p, q, v);
 if (v[r-2] > v[r-1]) {
 int t = v[r-2];
 v[r-2] = v[r-1];
 v[r-1] = t; }
 intercala (p, q, r, v); } }

```

6. Esta função rearranja  $v[p..r-1]$  em ordem crescente?

```

void mergesort6 (int p, int r, int v[]) {
 if (p < r-1) {
 q = r - 1;
 mergesort6 (p, q, v);
 intercala (p, q, r, v); } }

```

7. Suponha que sua biblioteca tem uma função  $mrg(p, q, r, v)$  que recebe um vetor  $v$  tal que  $v[p..q]$  e  $v[q+1..r-1]$  são crescentes e rearranja o vetor de modo que  $v[p..r-1]$  fique crescente. Use  $mrg$  para implementar o algoritmo Mergesort.

8. Suponha que sua biblioteca tem uma função  $interc(v, p, q, r)$  que recebe um vetor  $v$  tal que  $v[p..q-1]$  e  $v[q..r-1]$  estão em ordem crescente e rearranja o vetor de modo que  $v[p..r-1]$  fique em ordem crescente. (Qual o menor valor de  $q$  que  $interc$  deve aceitar? Qual o maior valor?) Use  $interc$  para escrever uma função  $mrgsort(v, p, r)$  que rearranje um vetor  $v[p..r]$  em ordem crescente.

## Desempenho do algoritmo Mergesort

Aplique a função `mergesort` a um vetor  $v[0..n-1]$ . O tamanho do vetor é reduzido à metade a cada passo da recursão. Na primeira “rodada”, a instância original do problema é reduzida a duas menores:  $v[0..n/2-1]$  e  $v[n/2..n-1]$ . Na segunda “rodada”, temos quatro instâncias:

$v[0..n/4-1]$ ,  $v[n/4..n/2-1]$ ,  $v[n/2..3n/4-1]$  e  $v[3n/4..n-1]$ .

E assim por diante, até que, na última “rodada”, cada instância tem no máximo 1 elemento. O número total de “rodadas” é aproximadamente  $\log n$  (portanto também aproximadamente  $\lg(n)$ ).

Em cada “rodada”, a função `intercala` executa  $2n$  movimentações de elementos do vetor  $v[0..n-1]$ . Assim, o número total de movimentações para ordenar  $v[0..n-1]$  é aproximadamente

$$2n \log n .$$

É fácil constatar que o consumo de *tempo* da função mergesort é proporcional ao número total de movimentações, e portanto proporcional a

$$n \log n .$$

Diz-se que o algoritmo é [linearítmico](#). O número  $n \log n$  cresce muito mais devagar que  $n^2$  e apenas um pouco mais rapidamente que  $n$ . Assim, se um vetor de tamanho  $N$  exige  $T$  unidades de tempo, um vetor de tamanho  $2N$  exigirá menos que  $2.2 T$  unidades de tempo, desde que  $N$  seja maior que  $2^{10}$ . Da mesma forma, um vetor de tamanho  $4N$  exigirá menos que  $4.4 T$  unidades de tempo, desde que  $N$  seja maior que  $2^{20}$ .

O consumo de tempo da função mergesort, proporcional a  $n \log n$ , é muito menor que o dos [algoritmos elementares](#), que consomem tempo proporcional a  $n^2$ . Entretanto, o fator de proporcionalidade é maior no caso do mergesort, pois o código é mais complexo. Assim, mergesort só se torna realmente mais rápido quando  $n$  é suficientemente grande.

## Exercícios 5

1. Como o consumo de tempo do seguinte fragmento de código varia com  $n$ ?

```
int c = 1;
for (int i = 0; i < n; i *= 2)
 for (int j = 1; j < n; ++j)
 c += 1;
```

2. CRONOMETRAGEM. Escreva um programa que cronometre sua implementação do Mergesort (use a função `clock` da [biblioteca time](#)). Divida os tempos por  $n \log n$  para comparar com as previsões teóricas.
3. VERSÃO “TRUNCADA”. Escreva uma versão do algoritmo Mergesort com *cutoff* para vetores pequenos: quando o vetor a ser ordenado tiver menos que  $M$  elementos, a ordenação passa a ser feita pelo [algoritmo de inserção](#). O valor de  $M$  pode ficar entre 10 e 20. (Esse truque é usado na prática porque o algoritmo de inserção é mais rápido que o Mergesort puro quando o vetor é pequeno. O fenômeno é muito comum: algoritmos sofisticados são tipicamente mais lentos que algoritmos simplórios quando o volume de dados é pequeno.)
4. ALOCAÇÃO/DESALOCAÇÃO EXCESSIVOS. A função mergesort acima chama as funções `malloc` e `free` muitas vezes (as chamadas acontecem dentro de `intercala`). Escreva uma versão da função que contenha o código da função de intercalação e chame `malloc` uma só vez.
5. DESAFIO: MERGESORT IN LOCO. Invente uma implementação de Mergesort que façam o serviço *in loco*, ou seja, dispensem o uso de um vetor auxiliar.
6. ORDEM DECRESCENTE. Escreva uma versão recursiva do algoritmo Mergesort que rearranje um vetor  $v[p..r-1]$  em ordem decrescente. Sua função deve conter o código da intercalação (que deve começar com  $v[p..q-1]$  e  $v[q..r-1]$  decrescentes e terminar com  $v[p..r-1]$  decrescente).
7. A seguinte função recursiva pretende encontrar o valor de um elemento máximo do vetor  $v[p..r]$ , não necessariamente ordenado. É claro que o problema só faz sentido se  $p \leq r$ .

```
int max (int p, int r, int v[]) {
 if (p == r) return v[r];
 else {
 int q = (p + r)/2;
 int x = max (p, q, v);
 int y = max (q+1, r, v);
 if (x >= y) return x;
 else return y; } }
```

A função está correta? Ela é mais rápida que a função iterativa óbvia? Quantas vezes a função chama a si mesma? Suponha que  $p$  e  $r$  valem 0 e 6 respectivamente e mostre a sequência, devidamente indentada, das chamadas de `max`.

## Versão iterativa do Mergesort

A versão iterativa do algoritmo Mergesort rearranja um vetor  $v[0..n-1]$  em ordem crescente. A ideia é muito simples. A cada iteração, intercalamos dois blocos consecutivos de  $b$  elementos cada: o primeiro bloco com o segundo, o terceiro com o quarto, etc. A variável  $b$  assume os valores  $1, 2, 4, 8, \dots$ .

```
// Esta função rearranja o vetor v[0..n-1]
// em ordem crescente.

void
mergesort_i (int n, int v[])
{
 int b = 1;
 while (b < n) {
 int p = 0;
 while (p + b < n) {
 int r = p + 2*b;
 if (r > n) r = n;
 intercala (p, p+b, r, v);
 p = p + 2*b;
 }
 b = 2*b;
 }
}
```

A figura ilustra a iteração em que  $b$  vale 2:

| 0   | p   | p+b | p+2b | n-1                         |
|-----|-----|-----|------|-----------------------------|
| 111 | 999 | 222 | 999  | 333 888 444 777 555 666 555 |

## Animações da versão iterativa do Mergesort

- [Animação do Mergesort](#) produzida por [Mike Bostock](#) (veja também [uma variante](#), uma [visualização estática](#) e uma [outra forma de visualização](#)). [Link sugerido por Yoshiharu Kohayakawa.]
- YouTube video: [Algorithms Lesson 3: Merge Sort](#).

## Exercícios 6

1. INVARIANTES. Quais são os invariantes do `while` externo na função `mergesort_i`? E os invariantes do `while` interno?
2. SEGMENTOS CRESCENTES. A função `mergesort_i` começa por quebrar o vetor original em segmentos de comprimento 1. Por que não começar com segmentos crescentes maximais? Exemplo: os segmentos crescentes maximais do vetor 1 2 3 0 2 4 6 4 5 6 7 8 9 são 1 2 3, 0 2 4 6 e 4 5 6 7 8 9. Explore esta ideia.

## Exercícios 7

1. LISTAS ENCADEADAS. Escreva uma versão do algoritmo Mergesort que rearranje uma lista encadeada em ordem crescente. Sua função não deve alocar novas células na memória. Faça duas versões: uma recursiva e uma iterativa.
2. NÚMERO DE INVERSÕES. O *número de inversões* de um vetor  $v[0..n-1]$  é o número de pares

ordenados  $(i, j)$  tais que  $0 \leq i < j < n$  e  $v[i] > v[j]$ . Escreva uma função que calcule o número de inversões de um vetor dado. O consumo de tempo de sua função deve ser proporcional a  $n \log n$  no pior caso.

3. DISTÂNCIA TAU DE KENDALL. Suponha dadas duas [permutações](#), digamos  $x[0..n-1]$  e  $y[0..n-1]$ , de um mesmo conjunto de números. A *distância tau* entre  $x$  e  $y$  é o número de pares de elementos do conjunto que estão em ordem diferente em  $x$  e  $y$ , ou seja, a cardinalidade do conjunto  $X - Y$  onde  $X$  é o conjunto de todos os pares  $(x[i], x[j])$  tais que  $i < j$  e  $Y$  é o conjunto de todos os pares  $(y[i], y[j])$  tais que  $i < j$ . (A definição não é assimétrica pois os conjuntos  $X - Y$  e  $X - Y$  têm a mesma cardinalidade.) Escreva uma função eficiente que calcule a distância tau entre  $x$  e  $y$ .

---

Veja o verbete [Merge sort](#) na Wikipedia.

---

Veja [Mergesort no Cprogramming.com](#).

---

Veja o algoritmo [Timsort](#), baseado em ideias semelhantes às do Mergesort.

Atualizado em 2018-09-20

<https://www.ime.usp.br/~pf/algoritmos/>

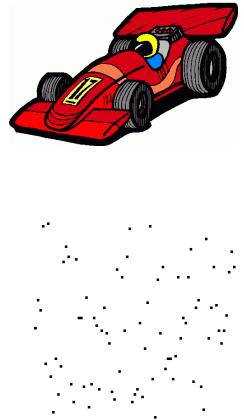
*Paulo Feofiloff*

[DCC-IME-USP](#)



[Se desejar, você pode ver a [versão anterior desta página](#).]

# Quicksort



O problema da ordenação consiste em rearranjar um vetor  $v[0..n-1]$  em ordem crescente, ou seja, [permutar](#) os elementos do vetor de modo que tenhamos  $v[0] \leq v[1] \leq \dots \leq v[n-1]$ . O algoritmo Quicksort, [inventado por C.A.R. Hoare](#) em 1962, resolve o problema.

Em algumas raras [instâncias](#), o Quicksort pode ser tão lento quanto os [algoritmos elementares](#), mas em geral é muito mais rápido. Mais precisamente, o algoritmo é [linearítmico](#) em média e [quadrático](#) no pior caso.

Usaremos duas abreviaturas ao discutir o algoritmo. A expressão “ $v[i..k] \leq x$ ” será usada como abreviatura de “ $v[j] \leq x$  para todo  $j$  no conjunto de índices  $i..k$ ”. A expressão “ $v[i..k] \leq v[p..r]$ ” será interpretada como “ $v[j] \leq v[q]$  para todo  $j$  no conjunto  $i..k$  e todo  $q$  no conjunto [p..r](#)”.

## O problema da separação

O núcleo do algoritmo Quicksort é o seguinte problema da separação (= *partition subproblem*), que formularemos de maneira propositalmente vaga:

rearranjar um vetor  $v[p..r]$  de modo que todos os elementos pequenos fiquem na parte esquerda do vetor e todos os elementos grandes fiquem na parte direita.

O ponto de partida para a solução desse problema é a escolha de um *pivô* (ou *fulcro*), digamos  $c$ . Os elementos do vetor que forem maiores que  $c$  serão considerados grandes e os demais serão considerados pequenos. É importante escolher  $c$  de tal modo que as duas partes do vetor rearranjado sejam estritamente menores que o vetor todo. A dificuldade está em resolver o problema da separação rapidamente sem usar um vetor auxiliar (como “espaço de trabalho”).

O problema da separação admite muitas formulações concretas. Eis uma primeira: rearranjar  $v[p..r]$  de modo que tenhamos  $v[p..j] \leq v[j+1..r]$  para algum  $j$  em  $p..r-1$ . (Nessa formulação, o pivô não é explícito.) Eis uma segunda formulação: rearranjar  $v[p..r]$  de modo que

$$v[p..j-1] \leq v[j] < v[j+1..r]$$

para algum  $j$  em  $p..r$ . (Aqui,  $v[j]$  é o pivô.) Exemplo:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 666 | 222 | 111 | 777 | 555 | 444 | 555 | 777 | 999 | 888 |
|     |     |     |     |     |     |     |     |     |     |

$j$

Neste capítulo, usaremos a segunda formulação do problema da separação.

## Exercícios 1

1. POSITIVOS E NEGATIVOS. Escreva uma função que rearranje um vetor  $v[p..r]$  de números inteiros de modo que tenhamos  $v[p..j-1] \leq 0$  e  $v[j..r] > 0$  para algum  $j$  em  $p..r+1$ . Faz sentido exigir que  $j$  esteja em  $p..r$ ? Procure escrever uma função *rápida* que não use vetor auxiliar. Repita o exercício depois de trocar “ $v[j..r] > 0$ ” por “ $v[j..r] \geq 0$ ”. Faz sentido exigir que  $v[j]$  seja 0?
2. LISTA. Suponha dada uma [lista encadeada](#) que armazena números inteiros. Escreva uma função que transforme a lista em duas: a primeira contendo os números pares e a segunda contendo os ímpares.
3. Critique a seguinte tentativa de resolver o problema da separação:

```
int sep (int v[], int p, int r) {
 int j = r;
 for (int i = r-1; i >= p; i--)
 if (v[i] > v[r]) {
 int t = v[r]; v[r] = v[i]; v[i] = t;
 j = i; }
 return j; }
```

4. Critique a seguinte solução do problema da separação:

```
int sep (int v[], int p, int r) {
 int w[1000], i = p, j = r, c = v[r];
 for (int k = p; k < r; ++k)
 if (v[k] <= c) w[i++] = v[k];
 else w[j--] = v[k];
 // agora i == j
 w[j] = c;
 for (int k = p; k <= r; ++k) v[k] = w[k];
 return j; }
```

5. Um programador inexperiente afirma que a seguinte função resolve a primeira formulação do problema da separação, ou seja, rearranja qualquer vetor  $v[p..r]$  (com  $p < r$ ) de tal forma que  $v[p..i] \leq v[i+1..r]$  para algum  $i$  em  $p..r-1$ .

```
int sep (int v[], int p, int r) {
 int i = p, j = r;
 int q = (p + r)/2;
 do {
 while (v[i] < v[q]) ++i;
 while (v[j] > v[q]) --j;
 if (i <= j) {
 int t = v[i], v[i] = v[j], v[j] = t;
 ++i, --j; }
 } while (i <= j);
 return i; }
```

Mostre um exemplo em que essa função não dá o resultado prometido. E se trocarmos “return  $i$ ” por “return  $i-1$ ”? É possível fazer algumas poucas correções de modo que a função dê o resultado esperado?

6. ★ Digamos que um vetor  $v[p..r]$  está *separado* se existe  $j$  em  $p..r$  tal que  $v[p..j-1] \leq v[j] < v[j+1..r]$ . Escreva um algoritmo que decida se  $v[p..r]$  está separado. Em caso afirmativo, o seu algoritmo deve devolver o índice  $j$ .

# O algoritmo da separação

Eis como o [livro de Cormen, Leiserson, Rivest e Stein](#) implementa o algoritmo da separação. (A implementação é atribuída a N. Lomuto.) A função resolve a [segunda formulação](#) do problema da separação:

```
// Recebe vetor v[p..r] com p <= r. Rearrange
// os elementos do vetor e devolve j em p..r
// tal que v[p..j-1] <= v[j] < v[j+1..r].

static int
separa (int v[], int p, int r) {
 int c = v[r]; // pivô
 int t, j = p;
 for (int k = p; /*A*/ k < r; ++k)
 if (v[k] <= c) {
 t = v[j], v[j] = v[k], v[k] = t;
 ++j;
 }
 t = v[j], v[j] = v[r], v[r] = t;
 return j;
}
```

(A palavra-chave `static` está aí apenas para indicar que `separa` é uma função auxiliar e não deve ser invocada diretamente pelo usuário final do Quicksort.)

Para mostrar que a função `separa` está correta, basta verificar que no início de cada iteração — ou seja, a cada passagem pelo ponto A — temos a seguinte configuração:

| p   | j   | k   | r   |
|-----|-----|-----|-----|
| ≤ c | ≤ c | ≤ c | > c |

(Note que podemos ter  $j == p$  bem como  $k == j$ , ou seja, o primeiro bem como o segundo segmentos podem estar vazios.) Mais precisamente, valem os seguintes [invariante](#)s: no início de cada iteração,

1.  $p \leq j \leq k$ ,
2.  $v[p..j-1] \leq c$ ,
3.  $v[j..k-1] > c$  e
4.  $v[p..r]$  é uma permutação do vetor original,

Na última passagem pelo ponto A, teremos  $k == r$  e portanto a seguinte configuração:

| p   | j   | k   |
|-----|-----|-----|
| ≤ c | ≤ c | ≤ c |

Assim, na fim da execução da função, teremos  $p \leq j \leq r$  e  $v[p..j-1] \leq v[j] < v[j+1..r]$ , como prometido.

| p   | j   | r   |
|-----|-----|-----|
| ≤ c | ≤ c | ≤ c |

[Gostaríamos](#)  $j$  que ficasse a meio caminho entre  $p$  e  $r$ , mas devemos estar preparados para aceitar os casos extremos em que  $j$  vale  $p$  ou  $r$ .

**Desempenho.** Quanto tempo a função `separa` consome? O consumo de tempo é proporcional ao número de comparações entre elementos do vetor. Não é difícil perceber que esse número é proporcional ao tamanho  $r - p + 1$  do vetor. A função é, portanto, [linear](#).

## Exercícios 2

1. Aplique a operação separa ( $v$ , 0, 15) ao seguinte vetor: 33 22 55 33 44 22 99 66 55 11 88 77 33 88 66 66.
2. CASO EXTREMO. Qual o resultado da função separa quando  $p == r$ ?
3. CASOS EXTREMOS. Qual o resultado da função separa quando os elementos de  $v[p..r]$  são todos iguais? E quando  $v[p..r]$  é crescente? E quando  $v[p..r]$  é decrescente? E quando cada elemento de  $v[p..r]$  tem um de dois valores possíveis?
4. INVARIANTES. Prove que os invariantes de separa enunciados [acima](#) estão de fato corretos.
5. VERSÃO RECURSIVA. Escreva uma versão recursiva da função separa.
6. PRIMEIRO ELEMENTO COMO PIVÔ. Reescreva o código de separa de modo que  $v[p]$  seja usado como de pivô.
7. A função separa produz um rearranjo [estável](#) do vetor, ou seja, preserva a ordem relativa de elementos de mesmo valor?
8. IMPLEMENTAÇÃO GRIES DA SEPARAÇÃO. Analise e discuta a seguinte implementação do algoritmo de separação. (Essa implementação aparece no livro *The Science of Programming* de D. Gries.) Mostre que essa implementação é equivalente à função separa [de Lomuto](#). [[Solução](#)]

```
int separa_Gries (int v[], int p, int r) {
 int c = v[p], i = p+1, j = r;
 while (true) {
 while (i <= r && v[i] <= c) ++i;
 while (c < v[j]) --j;
 if (i >= j) break;
 int t = v[i], v[i] = v[j], v[j] = t;
 ++i; --j; }
 v[p] = v[j], v[j] = c;
 return j; }
```

9. Critique a seguinte variante da [função separa\\_Gries](#):

```
int sep (int v[], int p, int r) {
 int c = v[p], i = p+1, j = r;
 while (i <= j) {
 if (v[i] <= c) ++i;
 else {
 int t = v[i], v[i] = v[j], v[j] = t;
 --j; } }
 v[p] = v[j], v[j] = c;
 return j; }
```

10. Critique a seguinte solução do problema da separação (extraída do [livro de E. Roberts](#)).

```
int separa_R (int v[], int p, int r) {
 int c = v[p], i = p+1, j = r;
 while (1) {
 while (i < j && c < v[j]) --j;
 while (i < j && v[i] <= c) ++i;
 if (i == j) break;
 int t = v[i], v[i] = v[j], v[j] = t; }
 if (v[j] > c) return p;
 v[p] = v[j], v[j] = c;
 return j; }
```

11. DESAFIO. Escreva uma solução do problema da separação que devolva um índice  $j$  tal que  $(r-p)/10 \leq j-p \leq 9(r-p)/10$ .
12. Um vetor é *binário* se cada um de seus elementos vale 0 ou 1. Escreva uma função que rearranke um vetor binário  $v[0..n-1]$  em ordem crescente e consuma tempo proporcional ao tamanho do vetor.

## Quicksort básico

Agora que resolvemos o problema da separação, podemos cuidar do Quicksort propriamente dito. O algoritmo usa a estratégia da [divisão e conquista](#) e tem a aparência de um [Mergesort](#)

“ao contrário”:

```
// Esta função rearranja qualquer vetor
// v[p..r] em ordem crescente.

void
quicksort (int v[], int p, int r)
{
 if (p < r) {
 int j = separa (v, p, r); // 2
 quicksort (v, p, j-1); // 3
 quicksort (v, j+1, r); // 4
 }
}
```

Se  $p \geq r$  (essa é a base recursão) não é preciso fazer nada. Se  $p < r$ , o código reduz a instância  $v[p..r]$  do problema ao par de instâncias  $v[p..j-1]$  e  $v[j+1..r]$ . Como  $p \leq j \leq r$ , essas duas instâncias são estritamente menores que a instância original. Assim, por hipótese de indução,  $v[p..j-1]$  estará em ordem crescente no fim da linha 3 e  $v[j+1..r]$  estará em ordem crescente no fim da linha 4. Como  $v[p..j-1] \leq v[j] < v[j+1..r]$  graças à função separa, o vetor todo estará em ordem crescente no fim da linha 4, como promete a documentação da função.

Podemos eliminarmos a segunda chamada recursiva (linha 4) e trocar o `if` por um `while`. Isso produz uma função equivalente ao quicksort acima:

```
void qcksrt (int v[], int p, int r) {
 while (p < r) {
 int j = separa (v, p, r);
 qcksrt (v, p, j-1);
 p = j + 1;
 }
}
```

## Exercícios 3

1. Que acontece se trocarmos “ $p < r$ ” por “ $p != r$ ” na linha 1 do quicksort? Que acontece se trocarmos “ $j-1$ ” por “ $j$ ” na linha 3 do código? Que acontece se trocarmos “ $j+1$ ” por “ $j$ ” na linha 4?
2. PEGADINHA. Quais são os invariantes da função quicksort?
3. Submeta o vetor 77 55 33 99 indexado por 1..4 à função quicksort. Teremos a seguinte sequência de chamadas da função (observe a indentação):

```
quicksort (v,1,4)
 quicksort (v,1,2)
 quicksort (v,1,0)
 quicksort (v,2,2)
 quicksort (v,4,4)
```

Repita o exercício com o vetor 55 44 22 11 66 33 indexado por 1..6.

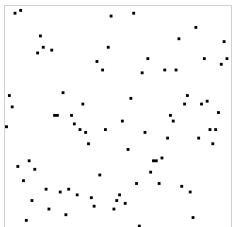
4. Escreva uma função com protótipo `quick_sort (int v[], int n)` que rearranje um vetor  $v[0..n-1]$  (atenção aos índices!) em ordem crescente. (Basta invocar quicksort da maneira correta.)
5. A função quicksort produz uma ordenação estável do vetor?
6. Discuta a seguinte implementação do algoritmo Quicksort (supondo  $p < r$ ):

```
void qsrt (int v[], int p, int r) {
 int j = separa (v, p, r);
 if (p < j-1) qsrt (v, p, j-1);
 if (j+1 < r) qsrt (v, j+1, r);
}
```

7. ★ FORMULAÇÃO ALTERNATIVA DA SEPARAÇÃO. Suponha dada uma função que resolve a primeira das formulações do problema da separação dadas acima (ou seja, rearranja  $v[p..r]$  e devolve  $j$  tal que  $v[p..j] \leq v[j+1..r]$ .) Escreva uma versão do algoritmo Quicksort que use essa função de separação. Que restrições devem ser impostas sobre  $j$ ?

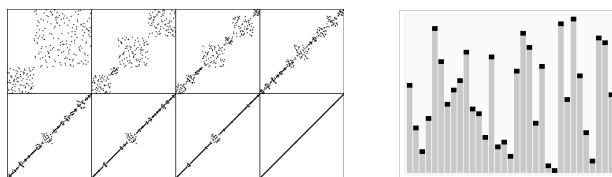
8. ★ VERSÃO ITERATIVA. Escreva uma versão não recursiva do algoritmo Quicksort. [[Solução](#).]
9. ★ QUICKSORT ALEATORIZADO. Escreva uma versão aleatorizada (= *randomized*) do algoritmo Quicksort.
10. TESTES COM VETOR ALEATÓRIO. Escreva um programa que teste, experimentalmente, a correção de sua implementação do algoritmo Quicksort. Use permutações aleatórias de  $1..n$  para os testes. Compare o resultado da ordenação com  $1..n$ .

## Animações do Quicksort



A animação à esquerda (copiada da [Simon Waldherr / Golang Sorting Visualization](#)) mostra a aplicação do Quicksort a um vetor  $v[0..79]$  de números positivos. Cada elemento  $v[i]$  do vetor é representado pelo ponto de coordenadas  $(i, v[i])$ .

A primeira das figuras abaixo (copiada da [Wikimedia](#)) mostra a aplicação do Quicksort a uma permutação  $v[0..249]$  do conjunto  $0..249$ . Cada elemento do vetor é representado por um ponto de coordenadas  $(i, v[i])$ . A figura mostra o estado do vetor em 8 estágios da ordenação. A segunda figura (copiada da [Wikipedia](#)) mostra a aplicação do Quicksort a uma permutação  $v[0..32]$  do conjunto  $0..32$ .



Veja outras animações de algoritmos de ordenação:

- [Comparison Sorting Algorithms](#), de David Galles (University of San Francisco).
- Animação de [15 algoritmos de ordenação](#), de Timo Bingmann, no YouTube.
- Animação do [Quicksort](#), de C. Parsons, no YouTube.
- [Animação do Quicksort](#) produzida por [Mike Bostock](#) (veja também uma [visualização estática](#), uma [variante](#), e uma [outra forma de visualização](#)). [Link sugerido por Yoshiharu Kohayakawa.]
- [Sorting Algorithms Animations](#) da Toptal.
- [Animação de algoritmos de ordenação](#), de Nicholas André Pinho de Oliveira.
- [Quick-sort com dança folclórica húngara](#), Universidade Sapientia (Romênia).

## Desempenho do Quicksort básico

Não é difícil perceber que o consumo de tempo da função `quicksort` é proporcional ao número de comparações entre elementos do vetor. Se o índice  $j$  devolvido por `separa` estiver [sempre mais ou menos a meio caminho](#) entre  $p$  e  $r$ , o número de comparações será aproximadamente  $n \log n$ , sendo  $n$  o tamanho  $r-p+1$  do vetor. Por outro lado, se o vetor já estiver ordenado ou quase ordenado, o número de comparações será aproximadamente

$$n^2.$$

Portanto, o pior caso do quicksort não é melhor que o dos [algoritmos elementares](#). Felizmente, o pior caso é muito raro: *em média*, o consumo de tempo do quicksort é proporcional a

$$n \log n.$$

(Veja detalhes na página [Análise do Quicksort](#) do meu sítio *Análise de Algoritmos*.) Portanto, o algoritmo é [linearítmico](#) em média e [quadrático](#) no pior caso.

## Exercícios 4

1. CRONOMETRAGEM. Escreva um programa que cronometre sua implementação do Quicksort (use a função `clock` da [biblioteca time](#)). Divilde os tempos por  $n \log n$  para comparar com as previsões teóricas.
2. VERSÃO “TRUNCADA”. Escreva uma versão do algoritmo Quicksort com *cutoff* para vetores pequenos: quando o vetor a ser ordenado tiver menos que  $M$  elementos, a ordenação passa a ser feita pelo [algoritmo de inserção](#). O valor de  $M$  pode ficar entre 10 e 20. (Esse truque é usado na prática porque o algoritmo de inserção é mais rápido que o Quicksort puro quando o vetor é pequeno. O fenômeno é muito comum: algoritmos sofisticados são tipicamente mais lentos que algoritmos simplórios quando o volume de dados é pequeno.)
3. A seguinte função [contribuição de Pedro Rey] promete rearranjar o vetor  $v[p..r]$  em ordem crescente. Mostre que a função está correta. Estime o consumo de tempo da função.

```
void psort (int v[], int p, int r) {
 if (p >= r) return;
 if (v[p] > v[r]) {
 int t = v[p], v[p] = v[r], v[r] = t;
 }
 psort (v, p, r-1);
 psort (v, p+1, r); }
```

4. ★ OUTRAS VERSÕES DO QUICKSORT. Procure outras versões do algoritmo Quicksort nos livros e na rede WWW. Discuta as diferenças em relação ao código acima. Qual a forma do [problema da separação](#) resolvido por cada versão? [[Solução](#)]

## Altura da pilha de execução do Quicksort

Na [versão básica do Quicksort](#), o código cuida imediatamente do segmento  $v[p..j-1]$  e trata do segmento  $v[j+1..r]$  somente depois que  $v[p..j-1]$  estiver ordenado. Dependendo do valor de  $j$  nas sucessivas chamadas da função, a [pilha de execução](#) pode crescer muito, atingindo altura igual ao tamanho do vetor. (Isso acontece, por exemplo, se o vetor estiver em ordem decrescente.) Esse fenômeno pode esgotar o espaço de memória. Para controlar o crescimento da pilha de execução é preciso tomar duas providências:

1. cuidar primeiro do  $\Delta$  menor dos segmentos  $v[p..j-1]$  e  $v[j+1..r]$  e
2. eliminar a segunda chamada recursiva da função, [trocando o if por um while](#).

Se adotarmos essas providências, o tamanho do segmento que está no topo da pilha de execução será menor que a metade do tamanho do segmento que está logo abaixo na pilha. De modo mais geral, o segmento que está em qualquer das posições da pilha de execução será menor que metade do segmento que está imediatamente abaixo. Assim, se a função for aplicada a um vetor com  $n$  elementos, a altura da pilha não passará de  $\log n$ .

```
// A função rearranja o vetor v[p..r]
// em ordem crescente.

void
quickSort (int v[], int p, int r)
{
 while (p < r) {
 int j = separa (v, p, r);
 if (j - p < r - j) {
 quickSort (v, p, j-1);
 p = j + 1;
 } else {
 quickSort (v, j+1, r);
 r = j - 1;
 }
 }
}
```

```
 }
}
}
```

## Exercícios 5

1. Suponha que a seguinte função é aplicada a um vetor com  $n$  elementos. Mostre que a altura da pilha de execução pode passar de  $\log n$ .

```
void quicks (int v[], int p, int r) {
 if (p < r) {
 int j = separa (v, p, r);
 if (j - p < r - j) {
 quicks (v, p, j-1);
 quicks (v, j+1, r); }
 else {
 quicks (v, j+1, r);
 quicks (v, p, j-1); } } }
```

2. Familiarize-se com a função [qsort](#) da biblioteca stdlib.

---

Veja o verbete [Quicksort](#) na Wikipedia.

---

Veja minha página [Análise do Quicksort](#).

---

Veja capítulo 11 do [Programming Pearls](#) de Jon Bentley.

---

Veja a página [Quicksort](#) no Cprogramming.com.

Atualizado em 2018-10-03

<https://www.ime.usp.br/~pf/algoritmos/>

*Paulo Feofiloff*

[DCC-IME-USP](#)



# Heapsort



Este capítulo examina um importante e sofisticado algoritmo de ordenação de vetores. O algoritmo que discutiremos

rearranja os elementos de um vetor  $v[1 \dots n]$  de modo que eles fiquem em ordem crescente,

ou seja, de modo que tenhamos  $v[1] \leq v[2] \leq \dots \leq v[n]$ . O algoritmo, conhecido como Heapsort, foi [descoberto por J.W.J. Williams](#) em 1964. O Heapsort [linearítmico](#), mesmo no pior caso.

Suporemos que os índices do vetor são  $1 \dots n$  e não  $0 \dots n-1$  (como é usual em C) pois essa convenção torna o código um pouco mais simples.

## Vetores e árvores binárias

Antes de começar a discutir o Heapsort, precisamos aprender a enxergar a árvore binária que está escondida em qualquer vetor. O conjunto de índices de qualquer vetor  $v[1 \dots m]$  pode ser encarado como uma árvore binária da seguinte maneira:

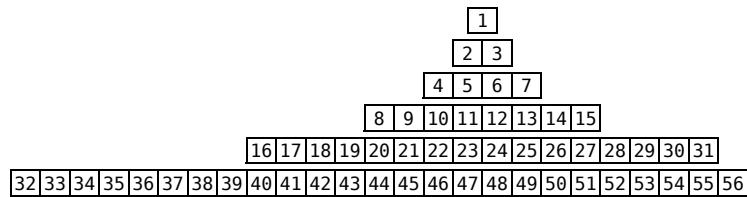
- o índice 1 é a *raiz* da árvore;
- o *pai* de qualquer índice  $f$  é  $f/2$  (é claro que 1 não tem pai);
- o *filho esquerdo* de um índice  $p$  é  $2p$  (esse filho só existe se  $2p \leq m$ );
- o *filho direito* de  $p$  é  $2p+1$  (esse filho só existe se  $2p+1 \leq m$ ).

Aqui, como no resto deste capítulo, vamos convencionar que as expressões que figuram como índices de um vetor são sempre *inteiros*. Assim, uma expressão da forma  $f/2$  significa  $\lfloor f/2 \rfloor$ , ou seja, o [piso](#) de  $f/2$ .

|     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10  | 11  | 12  | 13  | 14  |
| 999 | 999 | 999 | 999 | 999 | 999 | 999 | 999 | 999 | 999 | 999 | 999 | 999 | 999 |

Para tornar a árvore binária mais evidente, podemos desenhar o vetor em *camadas*, de tal

modo que cada filho fique na camada seguinte à do pai. A figura abaixo é um tal desenho do vetor  $v[1..56]$ ; os números nas caixas são os índices  $i$  e não os valores  $v[i]$ . Observe que cada camada, exceto talvez a última, tem duas vezes mais elementos que a camada anterior. Com isso, o número de camadas de um vetor  $v[1..m]$  é exatamente  $1 + \lfloor \lg(m) \rfloor$ , sendo  $\lfloor \lg(m) \rfloor$  o piso de  $\log m$ .



## Exercícios 1

1. Seja  $m$  um número da forma  $2^k - 1$ . Mostre que mais da metade dos elementos de qualquer vetor  $v[1..m]$  está na última camada.

## Heap

O segredo do algoritmo Heapsort é uma estrutura de dados, conhecida como [heap](#), que enxerga o vetor como uma árvores binária. Há dois sabores da estrutura: max-heap e min-heap; trataremos aqui apenas do primeiro, omitindo o prefixo “max”.

Um *heap* (= monte) é um vetor em que o valor de todo pai é maior ou igual ao valor de cada um de seus dois filhos. Mais precisamente, um vetor  $v[1..m]$  é um *heap* se

$$v[f/2] \geq v[f]$$

para  $f = 2, \dots, m$ . (Como já dissemos acima, a expressão  $f/2$  deve ser entendida como  $\lfloor f/2 \rfloor$ .)

|     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10  | 11  | 12  | 13  | 14  |
| 999 | 888 | 777 | 555 | 666 | 777 | 555 | 222 | 333 | 444 | 111 | 333 | 666 | 333 |

Para entender o algoritmo Heapsort, será preciso tratar, às vezes, de heaps “com defeitos”: diremos que um vetor  $v[1..m]$  é um heap *exceto talvez pelo índice k* se a desigualdade  $v[f/2] \geq v[f]$  vale para todo  $f$  diferente de  $k$ .

## Exercícios 2

1. O vetor 161 41 101 141 71 91 31 21 81 17 16 é um heap?
2. Mostre que todo vetor decrescente é um heap. Mostre que a recíproca não é verdadeira.
3. Escreva uma função que decida se um vetor  $v[1..m]$  é ou não um heap.
4. ★ Mostre que  $v[1..m]$  é um heap se e somente se cada índice  $p$  tem as seguintes propriedades:
  - (a)  $v[p] \geq v[2p]$  se  $2p \leq m$  e
  - (b)  $v[p] \geq v[2p+1]$  se  $2p+1 \leq m$ .
5. Suponha que  $v[1..m]$  é um heap e  $p$  é um índice menor que  $m/2$ . É verdade que  $v[2p] \geq v[2p+1]$ ? É verdade que  $v[2p] \leq v[2p+1]$ ?
6. Suponha que  $v[1..m]$  é um heap e sejam  $i < j$  dois índices tais que  $v[i] < v[j]$ . Se  $v[i]$  e  $v[j]$

forem intercambiados,  $v[1..m]$  continuará sendo um heap? Repita o exercício sob a hipótese  $v[i] > v[j]$ .

7. ★ Suponha que  $v[1..m]$  é um heap e que os elementos do vetor são diferentes entre si. É claro que o maior elemento do vetor está no índice 1. Onde pode estar o segundo maior elemento? Onde pode estar o terceiro maior elemento? É verdade que o terceiro maior elemento é filho do segundo maior elemento?

## Construção de um heap

É fácil rearranjar os elementos um vetor de inteiros  $v[1..m]$  para que ele se torne um heap. A ideia é repetir o seguinte processo enquanto o valor de um filho for maior que o de seu pai: troque os valores de pai e filho e “suba” um passo em direção à raiz. Mais precisamente, enquanto  $v[f/2] < v[f]$ , faça troca ( $v[f/2], v[f]$ ) e em seguida  $f = f/2$ . A operação de troca é definida assim:

```
#define troca (A, B) {int t = A; A = B; B = t;}
```

Eis o código completo:

```
// Rearranja um vetor v[1..m] de modo a
// transformá-lo em heap.

static void
constroiHeap (int m, int v[])
{
 for (int k = 1; k < m; ++k) {
 // v[1..k] é um heap
 int f = k+1;
 while (f > 1 && v[f/2] < v[f]) { // 5
 troca (v[f/2], v[f]); // 6
 f /= 2; // 7
 }
 }
}
```

(A palavra-chave `static` está aí apenas para indicar que `constroiHeap` é uma função auxiliar que não deve ser invocada diretamente pelo usuário final do Heapsort.)

No início de cada iteração controlada pelo `for`, o vetor  $v[1..k]$  é um heap. No curso da iteração,  $v[k+1]$  é incorporado ao heap. Para isso,  $v[k+1]$  “sobe” em direção à raiz até encontrar sua posição correta no heap.

Em cada iteração do bloco de linhas 6-7, o índice  $f$  pula de uma camada do vetor para a camada anterior. Portanto, esse bloco de linhas é repetido no máximo  $\lg(k)$  vezes para cada  $k$  fixo. Segue daí que o número total de comparações entre elementos do vetor (todas acontecem na linha 5 do código) não passa de

$$m \lg(m).$$

(Como veremos [adiante](#), é possível ser mais eficiente e transformar  $v[1..m]$  em heap com apenas  $m$  comparações.)

## Exercícios 3

1. IMPORTANTE. Critique a seguinte ideia: para transformar um vetor em heap, basta rearranjá-lo em ordem decrescente (usando o algoritmo Mergesort ou o algoritmo Quicksort, por exemplo).
2. Prove que a função `constroiHeap` está correta. Comece por estabelecer os [invariante](#)s do processo iterativo nas linhas 5 a 7.

3. Discuta a seguinte versão da função `constroiHeap`:

```
for (int k = 1; k < m; ++k) {
 int f = k+1, x = v[k+1];
 while (f > 1 && v[f/2] < x) {
 v[f] = v[f/2];
 f /= 2;
 }
 v[f] = x;
}
```

## A função peneira

O coração de muitos algoritmos que manipulam heaps é uma função que, ao contrário de [constroiHeap](#), “desce” em direção à base da árvore. Essa função, que chamaremos peneira, recebe um vetor qualquer  $v[1..m]$  e

faz  $v[1]$  “descer” até sua posição correta,

pulando de uma camada para a seguinte. Como isso é feito? Se  $v[1] \geq v[2]$  e  $v[1] \geq v[3]$ , não é preciso fazer nada. Se  $v[1] < v[2]$  e  $v[2] \geq v[3]$ , basta trocar  $v[1]$  com  $v[2]$  e depois fazer  $v[2]$  “descer” para sua posição correta. Os dois outros casos são semelhantes. (Faça um [rascunho](#) do algoritmo em pseudocódigo.) No exemplo a seguir, cada linha retrata o estado do vetor no início de uma iteração:

|    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 85 | 99 | 98 | 97 | 96 | 95 | 94 | 93 | 92 | 91 | 90 | 89 | 97 | 86 |
| 99 | 85 | 98 | 97 | 96 | 95 | 94 | 93 | 92 | 91 | 90 | 89 | 97 | 86 |
| 99 | 97 | 98 | 85 | 96 | 95 | 94 | 93 | 92 | 91 | 90 | 89 | 97 | 86 |
| 99 | 97 | 98 | 93 | 96 | 95 | 94 | 85 | 92 | 91 | 90 | 89 | 97 | 86 |

Segue o código da função. Cada iteração começa com um índice  $p$  e escolhe o filho  $f$  de  $p$  que tem maior valor:

```
static void
peneira (int m, int v[]) {
 int f = 2;
 while (f <= m) {
 if (f < m && v[f] < v[f+1]) ++f;
 // f é o filho mais valioso de f/2
 if (v[f/2] >= v[f]) break;
 troca (v[f/2], v[f]);
 f *= 2;
 }
}
```

A função será aplicada a vetores que são heaps [exceto talvez](#) por um ou dois índices. A função pode, portanto, ser documentada assim:

```
// Recebe um vetor v[1..m] que é um heap
// exceto talvez pelos índices 2 e 3 e
// rearranja o vetor de modo a
// transformá-lo em heap.
```

A seguinte versão é um pouco melhor, porque faz menos movimentações de elementos do vetor (e menos divisões de  $f$  por 2):

```
static void
peneira (int m, int v[])
{
 int p = 1, f = 2, x = v[1];
```

```

 while (f <= m) {
 if (f < m && v[f] < v[f+1]) ++f;
 if (x >= v[f]) break;
 v[p] = v[f];
 p = f, f = 2*p;
 }
 v[p] = x;
 }
}

```

**Desempenho.** A função peneira é muito rápida. Ela faz no máximo  $\lg(m)$  iterações, uma vez que o vetor tem  $1 + \lg(m)$  camadas. Cada iteração envolve duas comparações entre elementos do vetor e portanto o número total de comparações não passa de

$$2 \lg(m).$$

O consumo de *tempo* é proporcional ao número de comparações e portanto proporcional a  $\log m$  no pior caso.

## Exercícios 4

1. Por que a seguinte implementação de peneira não funciona?

```

int p = 1, f = 2;
while (f <= m) {
 if (v[p] < v[f]) {
 troca (v[p], v[f]);
 p = f;
 f = 2*p;
 } else {
 if (f < m && v[p] < v[f+1]) {
 troca (v[p], v[f+1]);
 p = f+1;
 f = 2*p;
 } else break;
 }
}

```

2. O seguinte código alternativo da função peneira está correto?

```

for (int f = 2; f <= m; f *= 2) {
 if (f < m && v[f] < v[f+1]) ++f;
 int p = f/2;
 if (v[p] >= v[f]) break;
 troca (v[p], v[f]);
}

```

3. Discuta a seguinte variante do código de peneira:

```

int x = v[1], f = 2;
while (f <= m) {
 if (f < m && v[f] < v[f+1]) ++f;
 if (x >= v[f]) break;
 v[f/2] = v[f];
 f *= 2;
}
v[f/2] = x;

```

4. Escreva uma versão recursiva da função peneira.

5. PENEIRA GENERALIZADA. Suponha que um vetor  $v[1..m]$  é um heap exceto talvez pelos índices  $2p$  e  $2p+1$ . Escreva uma função peneira\_2 que receba  $v[1..m]$  e  $p$  e transforme o vetor em heap.

6. ★ CONSTRUÇÃO RÁPIDA DE UM HEAP. Mostre que a seguinte função tem o mesmo efeito que constroiHeap\_acima, ou seja, transforma qualquer vetor  $v[1..m]$  em heap:

```

void constroiHeap_2 (int m, int v[]) {
 for (int p = m/2; p >= 1; --p)
 peneira_2 (p, m, v);
}

```

(Veja peneira\_2 no exercício anterior.) Mostre que constroiHeap\_2 faz no máximo  $(m \lg(m))/2$  comparações entre elementos do vetor. Refine sua análise para mostrar que, na verdade, a função faz no máximo  $m$  comparações.

7. O seguinte fragmento de código transforma qualquer vetor  $v[1..m]$  em heap?

```
for (int p = 1; p <= m/2; ++p)
 peneira_2 (p, m, v);
```

8. FILA DE PRIORIDADES. Uma *fila de prioridades* (= *priority queue*) é um conjunto de objetos (números, por exemplo) sujeito a duas operações: (1) remoção de um elemento de valor máximo e (2) inserção de um novo elemento. Se o conjunto for mantido num heap, essas duas operações podem ser realizadas de maneira muito rápida. Implemente as duas operações de modo que cada uma consuma tempo proporcional a  $\log n$  no pior caso, sendo  $n$  o número de objetos no conjunto.

## O algoritmo Heapsort

Não é difícil reunir o que dissemos acima para obter um algoritmo que rearanja um vetor  $v[1..n]$  em ordem crescente. O algoritmo tem duas fases: a primeira transforma o vetor em heap e a segunda rearanja o heap em ordem crescente. (Comece fazendo um [rascunho](#) do algoritmo.)

```
// Rearranja os elementos do vetor v[1..n]
// de modo que fiquem em ordem crescente.

void
heapsort (int n, int v[])
{
 constroiHeap (n, v);
 for (int m = n; m >= 2; --m) {
 troca (v[1], v[m]);
 peneira (m-1, v);
 }
}
```

No início de cada iteração do `for {}` valem as seguintes propriedades ([invariante](#)s):

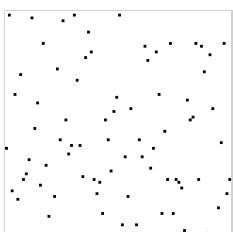
- o vetor  $v[1..m]$  é um heap,
- $v[1..m] \leq v[m+1..n]$ ,
- $v[m+1..n]$  está em ordem crescente e
- $v[1..n]$  é uma permutação do vetor original.

A expressão “ $v[1..m] \leq v[m+1..n]$ ” é uma abreviatura de “todos os elementos de  $v[1..m]$  são menores ou iguais a todos os elementos de  $v[m+1..n]$ ”.

| 1                  | heap |     |     |     |     |     |                   | m   | crescente |     |     |     | n   |
|--------------------|------|-----|-----|-----|-----|-----|-------------------|-----|-----------|-----|-----|-----|-----|
| 888                | 777  | 666 | 555 | 444 | 333 | 222 | 111               | 000 | 999       | 999 | 999 | 999 | 999 |
| elementos pequenos |      |     |     |     |     |     | elementos grandes |     |           |     |     |     |     |

Segue daí que  $v[1..n]$  estará em ordem crescente quando  $m$  for igual a 1. Portanto, o algoritmo está correto.

## Animações do Heapsort



A animação à esquerda (copiada de [Simon Waldherr / Golang Sorting Visualization](#)) mostra a aplicação do Heapsort a um vetor  $v[0..79]$  de números positivos. Cada elemento  $v[i]$  do vetor é representado pelo ponto de coordenadas  $(i, v[i])$ . (Por algum motivo, a animação não executa as duas últimas iterações.)

Veja outras animações de algoritmos de ordenação:

- [Heapsort](#), na Wikipedia.

- [Heapsort animation](#), de David Galles (University of San Francisco).
- Animação de [15 algoritmos de ordenação](#), de Timo Bingmann, no YouTube.
- [Sorting Algorithms Animations](#) da Toptal.
- [Animação de algoritmos de ordenação](#), de Nicholas André Pinho de Oliveira.

## Exercícios 5

1. Use a função `heapsort` para ordenar o vetor `16 15 14 13 12 11 10 9 8 7 6 5 4`. Mostre o estado do vetor no início de cada iteração.
2. Descreva o estado do vetor no início de uma iteração genérica do algoritmo Heapsort.
3. A função `heapsort` produz um rearranjo [estável](#) do vetor, ou seja, preserva a ordem relativa de elementos de mesmo valor?
4. Escreva uma função com protótipo `heap_sort (int *v, int n)` que rearranje um vetor `v[0..n-1]` (note os índices) em ordem crescente. (Basta invocar `heapsort` da maneira correta.)
5. TESTES COM VETOR ALEATÓRIO. Escreva um programa que teste, experimentalmente, a correção de sua implementação do algoritmo Heapsort. Use [permutações aleatórias de 1..n](#) para os testes e compare o vetor ordenado com `1..n`.
6. MIN-HEAP. Escreva uma função que rearranje um vetor `v[1..n]` em ordem *decrescente*. Sugestão: Adapte a definição de heap e reescreva a função peneira.
7. Imita um heap por meio de um conjunto de células interligadas com [ponteiros](#). Cada célula terá quatro campos: um valor e três ponteiros, um apontando o pai da célula, outro apontando o filho direito, e outro apontando o filho esquerdo. Escreva uma versão apropriada da função peneira. (Veja o capítulo [Árvores binárias](#).)

## Desempenho do Heapsort

Quantas comparações entre elementos do vetor a função `heapsort` executa? A função auxiliar `constroiHeap` faz [n lg\(n\) comparações no máximo](#). Em seguida, a função peneira é chamada cerca de  $n$  vezes e cada uma dessas chamadas faz [2 lg\(n\) comparações no máximo](#). Logo, o número total de comparações não passa de

$$3 n \lg(n) .$$

Quanto ao consumo de *tempo* do `heapsort`, ele é proporcional ao número de comparações entre elementos do vetor, e portanto proporcional a  $n \log n$  no pior caso. (Entretanto, a constante de proporcionalidade é maior que a do [Mergesort](#) e a do [Quicksort](#).)

## Exercícios 6

1. CRONOMETRAGEM. Escreva um programa que cronometre sua implementação do Heapsort (use a função `clock` da [biblioteca time](#)). Divida os tempos por  $n \log n$  para comparar com a previsão teórica.

Veja o capítulo 14 do “[Programming Pearls](#)” de Jon Bentley.

Veja [Heapsort no Cprogramming.com](#).

Atualizado em 2018-09-25

<https://www.ime.usp.br/~pf/algoritmos/>

*Paulo Feofiloff*

[\*DCC-IME-USP\*](#)

# Ordenação digital

"A good algorithm is like a sharp knife:  
 it does what it is supposed to do  
 with a minimum amount of applied effort.  
 Using the wrong algorithm to solve a problem  
 is like trying to cut a steak with a screwdriver:  
 you may eventually get a digestible result,  
 but you will expend considerably more effort  
 than necessary,  
 and the result is unlikely to be aesthetically pleasing."  
 — Th. Cormen, Ch. Leiserson, R. Rivest,  
*Introduction to Algorithms*

Muitos algoritmos de [ordenação](#) têm por base a *comparação* entre elementos do vetor. Este capítulo trata de um algoritmo de ordenação baseado na *contagem* dos elementos do vetor. Esse algoritmo, por sua vez, é a mola mestra de um método de ordenação para [strings](#) curtas, conhecido como *radix sort*. (O capítulo foi adaptado da seção 5.1 do [livro de Sedgewick e Wayne](#).)

## Vetores de inteiros pequenos

Suponha que queremos rearranjar em ordem crescente um vetor  $v[0..n-1]$  cujos elementos são números inteiros no intervalo

$$0 .. R-1$$

sendo  $R$  é um número pequeno. Esse intervalo é o *universo* do vetor. Depois de ordenado, o vetor terá uma carreira de elementos iguais a 0, seguida de uma carreira de elementos iguais a 1, etc., e finalmente uma carreira de elementos iguais a  $R-1$ . A figura mostra um vetor sobre o universo  $0..4$  antes e depois da ordenação:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 3 | 4 | 1 | 3 | 0 | 3 | 1 | 2 | 2 | 1 | 2 | 4 | 3 | 4 | 4 | 2 | 3 | 4 |
| 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 |   |

## Exercícios 1

1. VETOR DE BITS. Escreva uma função que rearranje em ordem crescente um vetor  $v[0..n-1]$  cujos elementos são 0s e 1s.
2. DNA. Escreva uma função que rearranje em ordem crescente um vetor cujos elementos são os caracteres A, C, G e T do código genético. Comece por transformar o conjunto A C G T no intervalo numérico  $0..3$ .

## Ordenação por contagem

Para ordenar o vetor  $v[0..n-1]$ , poderíamos usar um algoritmo baseados em comparação, como o de [inserção](#), o [Mergesort](#), o [Quicksort](#), ou o [Heapsort](#). Mas podemos fazer algo diferente, uma vez que o universo do vetor é conhecido. Para começar, precisamos do conceito de frequência.

A *frequência* de um elemento  $r$  de  $0..R-1$  é o número de ocorrências de  $r$  no vetor  $v[0..n-1]$ . Esse número será denotado por  $f[r]$ . No vetor do exemplo [acima](#), que tem universo  $0..4$ , as frequências são dadas pela seguinte tabela  $f[0..4]$ :

|     |   |   |   |   |   |
|-----|---|---|---|---|---|
|     | 0 | 1 | 2 | 3 | 4 |
| $f$ | 1 | 3 | 5 | 6 | 5 |

A *frequência dos predecessores* de um elemento  $r$  de  $0..R$  (note que acrescentamos  $R$  ao universo) é a soma  $f[0] + \dots + f[r-1]$ , denotada por  $fp[r]$  (observe que a soma não inclui  $f[r]$ ). É claro que  $fp[0]$  é nulo. No vetor do exemplo acima, a frequência dos predecessores é dada pela seguinte tabela  $fp[0..5]$ :

|      |   |   |   |   |    |    |
|------|---|---|---|---|----|----|
|      | 0 | 1 | 2 | 3 | 4  | 5  |
| $fp$ | 0 | 1 | 4 | 9 | 15 | 20 |

Observe que  $fp[r]$  é o índice a partir do qual deve começar, no vetor ordenado, a carreira de elementos iguais a  $r$ . No exemplo acima, a carreira de 1's deve começar na posição 1, a carreira de 2's deve começar na posição 4, a carreira de 3's deve começar na posição 9, etc.

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 3 | 3  | 3  | 3  | 3  | 3  | 4  | 4  | 4  | 4  | 4  |

Portanto, o vetor  $v[0..n-1]$  pode ser ordenado em duas etapas: primeiro, a tabela  $fp$  é usada para copiar os elementos de  $v[0..n-1]$  para as posições “corretas” de um vetor auxiliar  $aux[0..n-1]$ ; na segunda, o vetor  $aux$  é copiado de volta para o espaço ocupado por  $v$ . O seguinte código implementa as duas etapas do algoritmo:

```
// Rearranja v[0..n-1] em ordem crescente
// supondo que os elementos do vetor
// pertencem ao universo 0..R-1.

void
countingsort (int *v, int n, int R)
{
 int r;
 int *fp, *aux;
 fp = malloc ((R+1) * sizeof (int));
 aux = malloc (n * sizeof (int));

 for (r = 0; r <= R; ++r)
 fp[r] = 0;
 for (int i = 0; i < n; ++i) {
 r = v[i];
 fp[r+1] += 1;
 }
 // agora fp[r] é a frequência de r-1
 for (r = 1; r <= R; ++r)
 fp[r] += fp[r-1];
 // agora fp[r] é a freq dos predecessores de r
 // logo, a carreira de elementos iguais a r
 // deve começar no índice fp[r]
 for (int i = 0; i < n; ++i) {
 r = v[i];
 aux[fp[r]] = v[i];
 fp[r]++;
 }
 // aux[0..n-1] está em ordem crescente
 for (int i = 0; i < n; ++i)
 v[i] = aux[i];
}
```

```

 free (fp);
 free (aux);
}

```

A linha `*` incrementa `fp[r]` e assim prepara o terreno para posicionar o próximo elemento do vetor que tenha valor igual a `r`.

**Consumo de tempo.** A função `countingsort` consome  $n + R$  unidades de tempo. Se  $R$  é pequeno (da ordem de  $n$  no pior caso), isso é melhor que o consumo de tempo  $n \log n$  de algoritmos como o Mergesort, o Quicksort, e o Heapsort.

**Estabilidade.** O algoritmo que a função `countingsort` implementa é [estável](#). Essa propriedade é a base da aplicação da função à ordenação “digital” de vetores de strings a ser examinada na próxima seção.

## Exercícios 2

1. Verifique que a função `countingsort` está correta.
2. Critique a seguinte versão simplificada da função `countingsort`:

```

int *f = malloc (R * sizeof (int));
for (int r = 0; r < R; ++r) f[r] = 0;
for (int i = 0; i < n; ++i) f[v[i]] += 1;
int i = 0;
for (int r = 0; r < R; ++r)
 for (int k = 0; k < f[r]; ++k)
 v[i++] = r;
free (f);

```

[[Solução](#)]

3. VETOR DE STRUCTS. Imagine que o vetor `v[0..n-1]` representa o cadastro de funcionários de uma empresa. Cada elemento do vetor é uma struct com dois campos: um campo `num`, que dá o número do funcionário, e um campo `nome`, que dá o nome do funcionário. Suponha que cada número de funcionário é um inteiro com três dígitos decimais. Adapte o código de `countingsort` para ordenar `v[0..n-1]` pelo campo `num`.
4. VETORES DE LETRAS. Adapte a função `countingsort` para ordenar um vetor cujos elementos pertencem ao conjunto de caracteres A..Z.
5. ★ Mostre que a função `countingsort` é [estável](#).

## Ordenação digital de strings de mesmo comprimento

Suponha dado um vetor `v[0..n-1]` de [strings](#), todas do mesmo comprimento, e considere o problema de

rearranjar o vetor em [ordem lexicográfica](#).

(Você pode imaginar que todas as strings [são ASCII](#), mas isso não é essencial.) No contexto desse problema, os elementos (bytes) das strings são tradicionalmente chamados *dígitos*, mesmo que não pertençam no conjunto 0..9.

**Exemplo.** Suponha que as strings são placas de licenciamento de automóveis. (Poderíamos nos restringir aos quarenta e três caracteres que vão de '0' a 'Z'.) Segue um exemplo com 10 placas de 7 dígitos cada:

```
CDA7891
FAJ4021
DOG1125
BAT7271
GIZ1234
BAT7328
BIG8733
CAT9955
```

Para colocar esse vetor em ordem lexicográfica, podemos usar [ordenação por contagem](#) repetidas vezes: primeiro, para ordenar pelo último dígito, depois para ordenar pelo penúltimo dígito, e assim por diante:

```
CDA7891 EAD3312 FAJ4021 DOG1125 CDA7891 EAD3312 BAT7271
FAJ4021 FAJ4021 DOG1125 GIZ1234 EAD3312 FAJ4021 BAT7328
BAT7271 FON1723 GIZ1234 FON1723 DOG1125 BAT7271 BIG8733
EAD3312 DOG1125 BAT7271 EAD3312 BIG8733 BAT7328 CAT9955
FON1723 BAT7328 EAD3312 FAJ4021 FAJ4021 CAT9955 CDA7891
BIG8733 BIG8733 BAT7328 BAT7271 FON1723 CDA7891 DOG1125
GIZ1234 GIZ1234 FON1723 BAT7328 BAT7271 BIG8733 EAD3312
DOG1125 CAT9955 BIG8733 CDA7891 BAT7328 GIZ1234 FAJ4021
CAT9955 BAT7271 CDA7891 BIG8733 CAT9955 DOG1125 FON1723
BAT7328 CDA7891 CAT9955 CAT9955 GIZ1234 FON1723 GIZ1234
```

Como a ordenação por contagem é estável, o vetor de strings acaba ficando em ordem lexicográfica. Observe que é essencial aplicar a contagem “da direita para a esquerda”, ou seja, começando pelo último dígito.

**Algoritmo.** O código a seguir implementa o algoritmo sugerido no exemplo. O parâmetro  $W$  representa o comprimento comum de todas as strings (7 no exemplo acima) e o parâmetro  $R$  (menor que 256) especifica o tamanho do universo de [dígitos](#):

```
typedef unsigned char byte;

// Rearranja em ordem lexicográfica um vetor v[0..n-1]
// de strings. Cada v[i] é uma string v[i][0..W-1]
// cujos elementos pertencem ao conjunto 0..R-1.

void ordenacaoDigital (byte **v, int n, int W, int R)
{
 int *fp;
 byte **aux;
 fp = malloc ((R+1) * sizeof (int));
 aux = malloc (n * sizeof (byte *));

 for (int d = W-1; d >= 0; --d) {
 int r;
 for (r = 0; r <= R; ++r)
 fp[r] = 0;
 for (int i = 0; i < n; ++i) {
 r = v[i][d];
 fp[r+1] += 1;
 }
 for (r = 1; r <= R; ++r)
 fp[r] += fp[r-1];
 for (int i = 0; i < n; ++i) {
 r = v[i][d];
 aux[fp[r]] = v[i];
 fp[r]++;
 }
 for (int i = 0; i < n; ++i)
 v[i] = aux[i];
 }

 free (fp);
 free (aux);
}
```

O código ignora o byte nulo, \0, que [marca o fim de cada string](#) e portanto pode ser aplicado a qualquer vetor de vetores de bytes (desde que o valor de cada byte esteja no intervalo 0 .. R-1).

Esse algoritmo de ordenação é *digital* porque ordena as strings dígito-a-dígito. O algoritmo também é conhecido como

- ordenação digital da direita para a esquerda,
- ordenação digital a partir do dígito menos significativo, e
- *LSD (least significant digit) radix sort.*

**Consumo de tempo.** A função `ordenacaoDigital` consome  $W(n+R)$  unidades de tempo. Se  $R$  é pequeno (da ordem de  $n$  na pior hipótese), o consumo de tempo é proporcional a  $Wn$ , ou seja, ao número total de dígitos na entrada.

## Exercícios 3

1. Na função `ordenacaoDigital`, diga o que acontece se trocarmos a linha `for (d = W-1; d >= 0; --d)` por `for (d = 0; d < W; ++d)`.
2. [Sedgewick-Wayne 5.1.2] Aplique o algoritmo de ordenação digital ao vetor de strings  
`no is th ti fo al go pe to co to th ai of th pa`
3. [Sedgewick-Wayne 5.1.9] Escreva uma versão de `ordenacaoDigital` para vetores de strings ASCII de comprimento variável.
4. [Sedgewick-Wayne 5.1.20] Escreva uma função que receba inteiros  $n$  e  $W$  e produza um vetor de  $n$  strings aleatórias com  $W$  caracteres ASCII cada. (Essa função gera dados de teste para `ordenacaoDigital`.)
5. ★ EXPERIMENTO. Compare experimentalmente o desempenho das funções `ordenacaoDigital` e `heapsort`. Para fazer a comparação, gere vetores aleatórios com  $n$  strings, cada string com  $W$  dígitos decimais. Faça testes para  $n$  igual a 1 mil, 2 mil, 4 mil, 8 mil, ..., 512 mil e  $W$  igual a 1, 2, 4, 8.
6. [Sedgewick 10.36] Implemente o algoritmo de ordenação digital para listas encadeadas.

---

YouTube vídeo: [Radix sort on the playground](#).

---

Atualizado em 2018-08-17

<https://www.ime.usp.br/~pf/algoritmos/>

*Paulo Feofiloff*

[DCC-IME-USP](#)



# Hashing

*hash* = picadinho  
*to hash* = picar, fazer picadinho, misturar  
["Hashing is used extensively in applications and deserves recognition as one of the cleverer inventions of computer science."](#)  
 — E.S. Roberts, [Programming Abstractions in C](#)

Este capítulo usa um pequeno problema de contagem como pretexto para introduzir a estrutura de dados conhecida como *tabela de dispersão* ou *hash table*. Essa estrutura é responsável por acelerar muitos algoritmos que envolvem consultas, inserções e remoções de uma tabela de dados.

## Um problema de contagem

Suponha dado um fluxo de números inteiros [positivos](#) na entrada padrão [stdin](#). Os números serão chamados *chaves*. As chaves estão em ordem arbitrária e há muitas chaves repetidas. Considere agora o [problema](#) de

calcular o número de ocorrências de cada chave.

Por exemplo, o número de ocorrências de cada chave no fluxo 4998 9886 1933 1435 9886 1435 9886 7233 4998 7233 1435 1435 1004 é dado pela seguinte tabela (na primeira linha temos as diferentes chaves e na segunda o número de ocorrências de cada chave):

|      |      |      |      |      |      |
|------|------|------|------|------|------|
| 1004 | 1435 | 1933 | 4998 | 7233 | 9886 |
| 1    | 4    | 1    | 2    | 2    | 3    |

Nosso problema de contagem tem um requisito adicional importante: a contagem deve ser feita de maneira incremental, ou seja, *on-line*. Cada chave recebida do fluxo de entrada deve ser imediatamente contabilizada, de modo que tenhamos, a cada passo, as contagens referentes à *parte do fluxo vista até o momento*.

O desempenho de qualquer algoritmo para o problema será medido pelo tempo consumido para contabilizar *uma* chave. Idealmente, gostaríamos que esse tempo fosse constante, ou seja, não dependesse do número de chaves já lidas (nem do número de chaves *distintas* já lidas). Mas seremos obrigados a nos contentar com algo menos que o ideal.

**Quatro soluções.** Discutiremos quatro diferentes algoritmos para o problema. Os dois primeiros são muito simples. Os dois seguintes, bem mais eficientes na prática, usam a técnica de hashing. Embora simples, os dois primeiros algoritmos constituem uma importante introdução aos dois outros.

Denotaremos por  $N$  o *comprimento* do fluxo de entrada, ou seja, o número total de chaves no fluxo. O valor de  $N$  pode ser muito grande (milhões ou até bilhões), mas o número de chaves distintas é tipicamente bem menor.

Suporemos que todas as chaves são menores que um número  $R$ . No exemplo acima,  $R$  vale 10000. O conjunto  $0..R-1$  será chamado *universo das chaves*. Em geral, nem todas as chaves

do universo estão presentes no fluxo de entrada.

## Exercícios 1

1. Critique a seguinte proposta de algoritmo para o problema de contagem: armazene o fluxo de chaves num vetor, ordene o vetor, e depois percorra o vetor ordenado contando o número de ocorrências de cada chave.

## Algoritmo 1: endereçamento direto

Começamos com um algoritmo conhecido como *endereçamento direto*. Embora muito simples, esse algoritmo contém a semente da técnica de hashing.

Suponhamos que as chaves são do tipo int e que R chaves cabem confortavelmente na memória RAM do computador. Podemos então usar uma tabela tb[0..R-1] para registrar os números de ocorrências.

```
int *tb;
tb = malloc (R * sizeof (int));
```

O algoritmo de endereçamento direto inicializa o vetor tb com zeros e repete a seguinte rotina: lê uma chave ch do fluxo de entrada e contabiliza a chave executando a seguinte função:

```
void contabiliza (int ch) {
 tb[ch]++;
}
```

Depois de cada execução dessa função, para cada c em 0..R-1, o valor de tb[c] será o número de ocorrências de c na parte do fluxo lida até o momento.

**Desempenho.** Cada invocação de contabiliza consome tempo constante, ou seja, independente do tamanho R do universo e do número de chaves já lidas.

Esse algoritmo é muito rápido, mas só é prático se R for pequeno e conhecido explicitamente de antemão. Mesmo nesse caso, o algoritmo pode desperdiçar muito espaço. Por exemplo, se R vale 10 mil e o fluxo contém apenas mil chaves distintas, 90% do vetor tb ficará ocioso.

## Exercícios 2

1. TESTES. Escreva e teste um programa que resolva o problema de contagem e imprima um relatório com as seguintes informações: o comprimento N do fluxo de entrada, o número de chaves distintas, a chave mais frequente, e o número de ocorrências dessa chave. Use como fluxo de entrada os arquivos randInt1K.txt, randInt10K.txt, randInt100K.txt, randInt1M.txt, que contêm mil, 10 mil, 100 mil e 1 milhão de chaves aleatórias, todas entre 0 e 9999. Use a ideia do endereçamento direto. Cronometre o seu programa (use a função clock da biblioteca time). Os resultados estão de acordo com as previsões teóricas?

## Algoritmo 2: lista encadeada

Nosso segundo algoritmo armazena a contagem das chaves numa lista encadeada. As células da lista podem ter a seguinte estrutura:

```

typedef struct reg celula;
struct reg {
 int chave, ocorr;
 celula *prox;
};

```

Se  $p$  é o [endereço](#) de uma célula então  $p->ocorr$  é o número de ocorrências da chave  $p->chave$ . Se  $p$  e  $q$  são endereços de duas células diferentes então  $p->chave$  é diferente de  $q->chave$ . A lista encadeada será apontada pela variável global  $tb$ :

```
celula *tb;
```

O algoritmo de contagem inicializa  $tb$  com `NULL` e repete a seguinte rotina: lê uma chave  $ch$  do fluxo de entrada e invoca a seguinte função para contabilizar a chave:

```

void contabiliza (int ch) {
 celula *p;
 p = tb;
 while (p != NULL && p->chave != ch)
 p = p->prox;
 if (p != NULL)
 p->ocorr += 1;
 else {
 p = malloc (sizeof (celula));
 p->chave = ch;
 p->ocorr = 1;
 p->prox = tb;
 tb = p;
 }
}

```

**Desempenho.** No pior caso, cada execução de `contabiliza` consome tempo proporcional ao número de chaves distintas já lidas. Portanto, a execução de `contabiliza` pode ficar cada vez mais lenta à medida que o fluxo de entrada é lido. Se todas as chaves forem distintas, as últimas execuções de `contabiliza` podem chegar a consumir tempo proporcional a  $N$ .

Mesmo no caso médio, típico de aplicações práticas, o desempenho de `contabiliza` não é bom pois consome tempo proporcional à metade do número de chaves distintas já lidas.

Por outro lado, essa solução do problema de contagem não desperdiça espaço, pois o número de células é igual ao número de chaves distintas no fluxo de entrada.

## Exercícios 3

1. LISTA EM ORDEM CRESCENTE. Reescreva a função `contabiliza` acima de modo que as chaves sejam armazenadas na lista encadeada em ordem crescente. Estime o desempenho. Vale a pena manter a lista em ordem crescente?
2. TESTES. Repita os testes sugeridos num dos exercícios acima, desta vez usando uma lista encadeada para armazenar as contagens.
3. VETOR DE CÉLULAS. Refaça a discussão da seção anterior usando um *vetor* de células no lugar de uma lista encadeada. [Redimensione](#) o vetor à medida que o número de chaves distintas aumenta. Qual o consumo de tempo dessa versão? Agora mantenha o vetor em ordem crescente de chaves e refaça a análise.

## Tabelas de dispersão e funções de espalhamento

O próximos algoritmos haverão de combinar as boas qualidades dos dois algoritmos anteriores. Antes porém, precisamos introduzir o conceito de *hashing*. Começamos por estabelecer a

terminologia e descrever as ideias de maneira abstrata; implementações concretas serão dadas nas seções seguintes.

Vamos supor que a contagem das chaves é registrada num vetor  $tb[0..M-1]$ . O valor de  $M$  e a natureza exata dos elementos do vetor ficarão indefinidos por enquanto. Mas você deve imaginar que, de alguma forma, cada elemento de  $tb$  registra o número de ocorrências de alguma chave. O vetor  $tb$  será chamado *tabela de dispersão* ( $= hash table$ ). O tamanho  $M$  da tabela é usualmente menor que o tamanho  $R$  do universo de chaves. Assim, um elemento típico do vetor  $tb$  deverá cuidar de duas ou mais chaves.

Para determinar a posição no vetor  $tb$  que corresponde a uma chave  $ch$ , é preciso converter  $ch$  em um índice entre  $0$  e  $M-1$ . Qualquer função que faça a conversão, levando o universo  $0..R-1$  das chaves no conjunto  $0..M-1$  de índices, é chamada *função de espalhamento* ( $= hash function$ ). Neste capítulo, indicaremos por

$\text{hash}(ch, M)$

a invocação de uma função de espalhamento para uma chave  $ch$ . O número  $\text{hash}(ch, M)$  será chamado *código hash* ( $= hash code$ ) da chave  $ch$ . Uma função de espalhamento muito popular leva cada chave  $ch$  em  $ch \% M$ , ou seja, no resto da divisão de  $ch$  por  $M$ . Se  $M$  vale 100, por exemplo, então  $ch \% M$  consiste nos dois últimos dígitos decimais de  $ch$ .

Se a função de espalhamento levar duas chaves no mesmo índice, teremos uma *colisão*. Se  $M$  é menor que  $R$  — e mais ainda se  $M$  é menor que o número de chaves distintas — as colisões são inevitáveis. Se  $M$  vale 100, por exemplo, a função resto-da-divisão-por- $M$  faz colidir todas as chaves que têm os mesmos dois últimos dígitos decimais.

Uma boa função de espalhamento deve espalhar bem as chaves pelo conjunto  $0..M-1$  de índices. Uma função que leva toda chave num índice par, por exemplo, não é boa. Uma função que só depende de alguns poucos dígitos da chave também não é boa. Infelizmente, não existe uma função de espalhamento que seja boa para todos os conjuntos de chaves extraídos do universo  $0..R-1$ . Para começar a enfrentar essa dificuldade,

recomenda-se [que o parâmetro  \$M\$  seja um número primo](#),

pois isso tende a reduzir o número de colisões. Veja, por exemplo, o conjunto de chaves na primeira coluna da seguinte tabela (copiada do livro de Sedgewick e Wayne) e considere o resto da divisão de cada chave por 100 (um não-primo) e o resto da divisão por 97 (um primo). Observe que o número de colisões é maior no primeiro caso:

| ch  | ch%100 | ch%97 |
|-----|--------|-------|
| 212 | 12     | 18    |
| 618 | 18     | 36    |
| 302 | 2      | 11    |
| 940 | 40     | 67    |
| 702 | 2      | 23    |
| 704 | 4      | 25    |
| 612 | 12     | 30    |
| 606 | 6      | 24    |
| 772 | 72     | 93    |
| 510 | 10     | 25    |
| 423 | 23     | 35    |
| 650 | 50     | 68    |
| 317 | 17     | 26    |
| 907 | 7      | 34    |
| 507 | 7      | 22    |
| 304 | 4      | 13    |
| 714 | 14     | 35    |
| 857 | 57     | 81    |
| 801 | 1      | 25    |
| 900 | 0      | 27    |
| 413 | 13     | 25    |
| 701 | 1      | 22    |
| 418 | 18     | 30    |
| 601 | 1      | 19    |

Em geral, encontrar uma boa função de espalhamento é mais uma arte que uma ciência...

Agora que cuidamos de espalhar as chaves pelo intervalo  $0..M-1$ , precisamos inventar um meio de *resolver as colisões*, ou seja, de armazenar todas as chaves que a função de espalhamento

leva numa mesma posição da tabela de dispersão. As seções seguintes descrevem duas maneiras de fazer isso.

## Exercícios 4

1. POR QUE MÓDULO PRIMO? Suponha que  $ch$  e  $M$  são divisíveis por um inteiro  $k$ . Mostre que  $ch \% M$  também será divisível por  $k$ . (Este exercício dá uma pequena indicação das vantagens de usar um número primo como valor de  $M$ .)
2. Seja  $d$  o número  $[R/M]$ , isto é, [teto](#) de  $R/M$ . Considere a função de espalhamento que associa a cada chave  $ch$  o piso de  $ch/d$  (ou seja, o resultado da [divisão inteira](#) de  $ch$  por  $d$ ). Por exemplo, se  $R$  é  $10^5$  e  $M$  é  $10^2$  então  $d$  vale  $10^3$  e portanto  $ch/d$  é dado pelos dois primeiros dígitos decimais de  $ch$ . Discuta a qualidade dessa função de espalhamento.
3. PARADOXO DO ANIVERSÁRIO. Aplique a função de espalhamento resto-da-divisão-por- $M$  a uma sequência de chaves aleatórias. Depois de quantas chaves acontece a primeira colisão? Faça experimentos, com diversos valores de  $M$ , para determinar o momento da primeira colisão. (De acordo com a teoria das probabilidades clássica, a primeira colisão acontece depois de aproximadamente  $\sqrt{pM/2}$  chaves, sendo  $p$  o número  $\pi$ , igual a  $3.14159\dots$ )

## Algoritmo 3: hashing com encadeamento

A técnica de hashing tem dois ingredientes: uma função de espalhamento e um mecanismo de resolução de colisões. A seção anterior discutiu o primeiro ingrediente; esta seção cuida do segundo.

As colisões na [tabela de dispersão](#) podem ser resolvidas por meio de listas encadeadas: todas as chaves que têm um mesmo código hash são armazenadas numa lista encadeada. As células das listas encadeadas são iguais às do [algoritmo 2](#):

```
typedef struct reg celula;
struct reg {
 int chave, ocorr;
 celula *prox;
};
```

Podemos supor então que os elementos da tabela de dispersão  $tb[0..M-1]$  são ponteiros para listas encadeadas:

```
celula **tb;
tb = malloc (M * sizeof (celula *));
```

Para cada índice  $h$ , a lista encadeada  $tb[h]$  conterá todas as chaves que têm código hash  $h$ .

O algoritmo de contagem resultante é conhecido como *hashing com encadeamento* e pode ser visto como uma combinação dos algoritmos [1](#) e [2](#) discutidos acima. O algoritmo inicializa todos os elementos do vetor  $tb$  com `NULL` e repete a seguinte rotina: lê uma chave  $ch$  do fluxo de entrada e executa a seguinte função:

```
void contabiliza (int ch) {
 int h = hash (ch, M);
 celula *p = tb[h];
 while (p != NULL && p->chave != ch)
 p = p->prox;
 if (p != NULL)
 p->ocorr += 1;
 else {
 p = malloc (sizeof (celula));
 p->chave = ch;
 p->ocorr = 1;
```

```

 p->prox = tb[h];
 tb[h] = p;
}
}

```

**Desempenho.** No pior caso, a função de espalhamento hash leva todas as chaves na mesma posição da tabela de dispersão e portanto todas as chaves ficam na mesma lista encadeada. Nesse caso, o desempenho não é melhor que o do [algoritmo 2](#): cada execução de contabiliza consome tempo proporcional ao número de chaves já lidas do fluxo de entrada.

No caso médio, típico de aplicações práticas, o desempenho de contabiliza é muito melhor. Se a função hash espalhar bem as chaves pelo conjunto  $0..M-1$ , todas as listas encadeadas terão aproximadamente o mesmo comprimento e então podemos esperar que o consumo de tempo de contabiliza seja limitado por algo proporcional a

$$n / M,$$

onde  $n$  é o número de chaves lidas até o momento. Se  $M$  for 997, por exemplo, podemos esperar que a função seja cerca de 1000 vezes mais rápida que aquela do algoritmo 2. É claro que devemos procurar escolher um valor de  $M$  que seja grande o suficiente para que as  $M$  listas sejam curtas (digamos algumas dezenas de elementos) mas não tão grande que muitas das listas fiquem vazias.

## Exercícios 5

1. Resolva o problema da contagem para o fluxo de chaves 17 21 19 4 26 30 37 usando hashing com encadeamento. A tabela de dispersão deve ter tamanho 13 e a função de espalhamento deve ser o resto da divisão da chave por  $M$ . Faça uma figura do estado final da tabela de dispersão.
2. Suponha que o comprimento  $N$  do fluxo de entrada é aproximadamente 50000. Escolha um bom valor para o tamanho  $M$  da tabela de dispersão.
3. TESTES. Repita os testes sugeridos num dos exercícios [acima](#), desta vez usando uma tabela de dispersão com colisões resolvidas por encadeamento. Experimente diferentes valores (primos e não-primos) para o tamanho  $M$  da tabela de dispersão. Calcule também a média e o desvio padrão dos comprimentos das listas encadeadas.

## Algoritmo 4: hashing com sondagem linear

Esta seção descreve uma segunda maneira de [resolver as colisões](#) na [tabela de dispersão](#): todas as chaves que colidem são armazenadas *na própria tabela*.

Os elementos da tabela de dispersão  $tb[0..M-1]$  são células que têm apenas os campos `chave` e `ocorr`:

```

typedef struct reg celula;
struct reg {
 int chave, ocorr;
};

celula *tb;
tb = malloc (M * sizeof (celula));

```

Durante a contagem, algumas das células da tabela  $tb$  estarão *vagas* enquanto outras estarão *ocupadas*. As células vagas terão `chave` igual a -1. Nas células ocupadas, a chave estará em  $0..R-1$  e `ocorr` será o correspondente número de ocorrências. Se uma célula  $tb[h]$  está vaga, podemos garantir que nenhuma chave na parte já lida do fluxo de entrada tem [código hash](#) igual a  $h$ . Mas se  $tb[h]$  está ocupada, não podemos concluir que o código hash de  $tb[h].chave$  é igual a  $h$ .

Cada chave  $ch$  do fluxo de entrada será contabilizada da seguinte maneira. Seja  $h$  o código hash de  $ch$ . Se a célula  $tb[h]$  estiver vaga ou tiver chave igual a  $ch$ , o conteúdo da célula é atualizado. Senão, algoritmo *procura a próxima célula* de  $tb$  que esteja vaga ou tenha chave igual a  $ch$ .

A implementação dessas ideias leva ao algoritmo de *hashing com sondagem linear*. O algoritmo começa por inicializar todas as células da tabela  $tb$  fazendo  $chave = -1$  e  $ocorr = 0$ . Depois, repete a seguinte rotina: lê uma chave  $ch$  e invoca a seguinte função:

```
void contabiliza (int ch) {
 int c, sonda, h;
 h = hash (ch, M);
 for (sonda = 0; sonda < M; sonda++) {
 c = tb[h].chave;
 if (c == -1 || c == ch) break; /* *
 h = (h + 1) % M;
 }
 if (sonda >= M)
 exit (EXIT_FAILURE);
 if (c == -1)
 tb[h].chave = ch;
 tb[h].ocorr++;
}
```

A função faz  $M$  tentativas — conhecidas como *sondagens* — de encontrar uma célula “boa” (na linha \* do código). A busca fracassa somente se a tabela  $tb$  estiver cheia. Nesse caso, a execução da função é abortada. (Teria sido melhor [redimensionar](#) a tabela  $tb$  e continuar trabalhando.)

Suponha, por exemplo, que o tamanho  $M$  da tabela de dispersão é 10 (adotamos um número não-primo para tornar o exemplo mais simples). Suponha também que  $hash(ch, M)$  é definido como  $ch \% M$ . Se o fluxo de entrada é

333 336 1333 333 7777 446 556 999

então o estado final da tabela de dispersão  $tb[0..9]$  será o seguinte:

| chave | ocorr |
|-------|-------|
| 999   | 1     |
| -1    | 0     |
| -1    | 0     |
| 333   | 2     |
| 1333  | 1     |
| -1    | 0     |
| 336   | 1     |
| 7777  | 1     |
| 446   | 1     |
| 556   | 1     |

**Desempenho.** No pior caso, a função de espalhamento  $hash$  leva todas as chaves na mesma posição da tabela de dispersão e portanto as chaves ocupam células consecutivas da tabela. Nesse caso, o desempenho não é melhor que o do [algoritmo 2](#): cada execução de `contabiliza` consome tempo proporcional ao número de chaves já lidas do fluxo de entrada.

No caso médio, típico de aplicações práticas, o desempenho de `contabiliza` é muito melhor. △ Se mais da metade das células da tabela de dispersão estiver vaga (como acontece [se usarmos redimensionamento](#)) e a função  $hash$  espalhar bem as chaves pelo conjunto  $0..M-1$ , uma execução da função `contabiliza` não precisará fazer mais que algumas poucas sondagens para encontrar uma célula “boa”. Assim, o consumo de tempo de uma execução de `contabiliza` será praticamente independente do número de chaves já lidas.

## Exercícios 6

1. Resolva o problema da contagem para o fluxo de chaves 17 21 19 4 26 30 37 usando hashing com

sondagem linear. A tabela de dispersão deve ter tamanho 13 e a função de espalhamento deve ser o resto da divisão da chave por 13. Faça uma figura do estado final da tabela de dispersão.

2. PROVA DE CORREÇÃO. Considere a função contabiliza do algoritmo de hashing com sondagem linear dada [acima](#). Suponha que temos  $c == -1$  numa certa iteração. Prove que não existe célula  $tb[h]$  tal que  $tb[h].chave == ch$ .
3. ★ REDIMENSIONAMENTO DA TABELA DE DISPERSÃO. A execução da função contabiliza dada [acima](#) é abortada se a tabela de dispersão estiver cheia. Escreva uma versão melhor, que [redimensione](#) a tabela escolhendo um novo valor de  $M$  que seja aproximadamente o dobro do anterior, alocando uma nova tabela  $tb$ , e reinserindo todas as chaves na nova tabela.
4. TESTES. Repita os testes sugeridos num dos exercícios [acima](#), desta vez usando uma tabela de dispersão com colisões resolvidas por sondagem linear. Experimente diferentes valores (primos e não-primos) para o tamanho  $M$  da tabela. Calcule também o número máximo de sondagens que contabiliza faz para encontrar a posição desejada na tabela de dispersão.

## Exercícios 7

1. FILTRO DE REPETIDOS. Escreva um programa que leia um [arquivo de texto](#) que contém (as representações decimais de) números inteiros [positivos](#), elimine os números repetidos, e grave uma versão do arquivo sem os repetidos. Não altere a ordem relativa dos números. O seu programa deve ter caráter de *filtro*, ou seja, deve gravar o resultado à medida que o arquivo de entrada for sendo lido.
2. Encontre o primeiro número não-repetido em um longo [arquivo de texto](#) que contém números inteiros.

## Hashing de strings

Em muitas aplicações, as chaves são [strings](#) (especialmente [strings ASCII](#)) e não números. Como construir uma tabela de dispersão nesse caso? Poderíamos, por exemplo, adotar uma tabela de dispersão de tamanho 256 e usar a função de espalhamento que leva cada string no valor numérico do seu primeiro byte. (Todas as strings que começam com 'a', por exemplo, seriam levadas para a posição 97 da tabela.) Mas essa ideia não espalharia bem o conjunto de chaves pelo intervalo 0..255.

Uma maneira geral de lidar com chaves que são strings envolve dois passos: o primeiro converte a string em um número inteiro; o segundo submete esse número a uma função de espalhamento. Uma conversão óbvia consiste em tratar cada string como um número inteiro na base 256. A string "abcd", por exemplo, é convertida no número  $97 \times 256^3 + 98 \times 256^2 + 99 \times 256^1 + 100 \times 256^0$ , igual a 1633837924. Esse tipo de conversão pode facilmente produzir números maiores que [INT\\_MAX](#), e assim levar a um [overflow](#) aritmético. Se os cálculos forem feitos com variáveis [int](#), o resultado poderá ser estritamente negativo, o que seria desastroso. Por isso, faremos os cálculos com variáveis [unsigned](#), garantindo assim que o resultado fique entre 0 e [UINT\\_MAX](#) mesmo que haja overflow:

```
typedef char *string;

unsigned convert (string s) {
 unsigned h = 0;
 for (int i = 0; s[i] != '\0'; i++)
 h = h * 256 + s[i];
 return h;
}
```

Para submeter o resultado da conversão à função de espalhamento basta fazer `hash(convert(s), M)`. Se adotarmos a função resto-da-divisão-por-M, basta fazer `convert(s) % M`.

Uma boa alternativa é fazer a conversão e o cálculo da função de espalhamento em um só movimento. O resultado pode não ser idêntico a `convert(s) % M`, mas cumpre o papel de espalhar as chaves pelo conjunto  $0..M-1$ :

```
int string_hash (string s, int M) {
 unsigned h = 0;
 for (int i = 0; s[i] != '\0'; i++)
 h = (h * 256 + s[i]) % M;
 return h;
}
```

Se a string é "abcd" e  $M$  é 101, por exemplo, o índice calculado por `string_hash` é 11:

```
(0 * 256 + 97) % 101 = 97
(97 * 256 + 98) % 101 = 84
(84 * 256 + 99) % 101 = 90
(90 * 256 + 100) % 101 = 11
```

O uso da base 256 não é obrigatório; podemos usar uma outra base qualquer. Por razões não muito óbvias, convém que a base seja um número primo, como 31 por exemplo.

## Exercícios 8

1. Suponha que as chaves são strings e  $M$  vale 256. Considere a função de espalhamento que associa a cada chave o seu primeiro byte. Discuta a qualidade dessa função de espalhamento.
2. Mostre que se trocarmos "256" por "1" na função `convert`, todas as [permutações](#) da string  $s$  colidirão.
3. Na função `convert`, por que não trocar as duas linhas do meio pelas seguintes?

```
unsigned h = s[0];
for (i = 1; s[i] != '\0'; i++)
```

---

Veja minhas [notas de aulas sobre hashing](#) baseadas no livro de Sedgewick e Wayne e [minhas notas](#) baseadas no livro de Sedgewick.

---

Veja os verbetes [Hash table](#) e [Hash function](#) na Wikipedia.

---

Veja o artigo [Hash function](#) no MathWorld da Wolfram.

Atualizado em 2018-08-14

<https://www.ime.usp.br/~pf/algoritmos/>

*Paulo Feofiloff*

[DCC-IME-USP](#)



# Busca de palavras em um texto

“É como procurar agulha num palheiro.”  
— dito popular

Quantas vezes a palavra “algoritmo” aparece neste capítulo?

Procurar uma palavra num texto é uma atividade corriqueira. No contexto computacional, esse [problema](#) é conhecido como *substring searching* ou *string matching* e pode ser formulado assim:

Encontrar as ocorrências de um dado vetor  $a$  em um dado vetor  $b$ .

Suporemos que os elementos dos vetores são [bytes](#) (embora o problema faça sentido para vetores de qualquer outro tipo). Mas não vamos supor que os vetores são [strings](#), ou seja, que terminam com um byte nulo.

Em muitas aplicações, os elementos de  $a$  e  $b$  representam [caracteres ASCII](#) e assim cada byte pertence ao conjunto [0...127](#)). Em outras aplicações,  $a$  e  $b$  representam sequências de [caracteres Unicode](#) em [código UTF-8](#) (portanto, cada caractere é representado por 1, 2, 3 ou 4 bytes consecutivos). Em geral, entretanto, não há restrições sobre os elementos de  $a$  e  $b$ : cada um pode assumir qualquer valor entre 0 e 255.

Diremos que o vetor  $a$  é uma *palavra* (mesmo que não represente uma palavra em português ou inglês no sentido usual) e o vetor  $b$  é um *texto*. O problema consiste, então, em encontrar as ocorrências de uma palavra em um texto.

A seguinte figura sugere uma aplicação ao processamento de código genético. Nesse caso, os elementos dos vetores são as letras A, C, G e T. A figura destaca uma ocorrência da palavra TACTA no texto GTAG...TAG:

G T A G T A T A T A T A T A C T A C T A G T A G  
T A C T A

Outro exemplo: procurar por um determinado vírus de software num arquivo digital também é, essencialmente, um problema de busca de uma palavra (o vírus) num texto (o arquivo).

## Terminologia e decisões de projeto

Já tomamos a decisão de tratar apenas de vetores de bytes. Tomaremos outra decisão de projeto: a palavra e o texto serão indexados a partir de 1 (e não a partir de 0) a fim de simplificar ligeiramente as expressões envolvendo segmentos desses vetores. Finalmente, vamos nos restringir à versão simplificada do problema que pede apenas um *número* como resposta:

Encontrar o número de ocorrências de uma palavra  $a[1..m]$  em um texto  $b[1..n]$ .

Se  $m > n$ , o número de ocorrências de  $a$  em  $b$  é zero. Para garantir que o número de ocorrências seja finito, suporemos sempre que  $m \geq 1$ .

Convém adotar a seguinte terminologia ao tratar do problema:

- um vetor  $a[1..m]$  *casa com* um vetor  $b[j..k]$  se  $a[1] = b[j], a[2] = b[j+1], \dots, a[m] = b[k]$ ; é claro que devemos ter  $m-1 = k-j$ ;
- um *sufixo* de um vetor  $b[1..k]$  é qualquer *segmento* terminal do vetor, ou seja, qualquer segmento da forma  $b[j..k]$ , com último índice igual a  $k$ ;
- a expressão “ $a[1..m]$  casa com um sufixo de  $b[1..k]$ ” será usada como abreviatura de “ $a[1..m]$  casa com  $b[k-m+1..k]$ ”;
- um vetor  $a[1..m]$  *ocorre em*  $b[1..n]$  se existe  $k$  tal que  $a[1..m]$  casa com um sufixo de  $b[1..k]$ .

Note que duas ocorrência de  $a$  em  $b$  podem se sobrepor. Por exemplo, as duas ocorrências de BABA em XBABABAX se sobrepõem.

**Exemplos.** Nos exemplos a seguir, o sinal  $\downarrow$  indica as posições  $k$  em que o vetor  $a$  (linha inferior) casa com um sufixo do vetor  $b[1..k]$  (linha superior):

|                                                      |   |   |
|------------------------------------------------------|---|---|
| U m v e t o r    a    o c o r r e    e m    b    s e | ↓ |   |
| o c o r r e    e                                     |   |   |
| 3 1 3 1 4 3 1 4 1 3 1 4 1 5 9 3 1 4 1 5 9 2 6 3 1 4  | ↓ | ↓ |
| 3 1 4 1 5 9                                          |   |   |
| G T A G T A T A T A T A T A C T A C T A G T A G      | ↓ | ↓ |
| T A C T A                                            |   |   |

**Direções de varredura.** Qualquer algoritmo que procure uma palavra num texto deverá executar uma varredura (= *scan*) do texto. Para procurar as ocorrências de uma palavra  $a$  num texto  $b$ , poderíamos varrer  $b$  da esquerda para a direita ou da direita para a esquerda. As duas alternativas são equivalentes, mas vamos adotar sempre a primeira: comparar  $a$  com  $b[1..m]$ , depois com  $b[2..m+1]$ , e assim por diante.

Para um  $k$  fixo, a comparação elemento-a-elemento de  $a[1..m]$  com um sufixo de  $b[1..k]$  poderia ser feita da esquerda para a direita ou da direita para a esquerda. Em geral, as duas alternativas são equivalentes, mas um dos algoritmos que veremos adiante exige que a comparação seja feita na direção *contrária* à da varredura do texto. Por isso, a comparação elemento-a-elemento será sempre feita da direita para a esquerda: primeiro  $a[m]$  com  $b[k]$ , depois  $a[m-1]$  com  $b[k-1]$ , e assim por diante.

## Exercícios 1

1. Quantas vezes a palavra AAA ocorre no texto AAAA ?
2. Quais são os bytes que representam os caracteres A, C, G e T?
3. ★ Faça uma figura semelhante às exibidas acima para mostrar uma ocorrência do vetor ação no vetor notação binária. Cada pequena “caixa” da figura deve representar um byte, não um caractere.
4. Discuta (vagamente, em termos gerais) a seguinte afirmação: “qualquer algoritmo para a versão simplificada do problema pode ser modificado de modo a resolver a versão mais geral”.

# Algoritmo inocente

A seguinte função resolve o problema (da busca de uma palavra em um texto) da maneira mais óbvia e direta. Pacientemente, ela tenta casar a com  $b[1..m]$ , depois com  $b[2..m+1]$ , e assim por diante:

```
typedef unsigned char byte;

// Recebe vetores a[1..m] e b[1..n],
// com m >= 1 e n >= 0, e devolve
// o número de ocorrências de a em b.

int
inocente (byte a[], int m, byte b[], int n)
{
 int ocorrs = 0;
 for (int k = m; k <= n; ++k) {
 // a[1..m] casa com b[k-m+1..k]?
 int i = m, j = k;
 while (i >= 1 && a[i] == b[j])
 --i, --j;
 if (i < 1) ++ocorrs;
 }
 return ocorrs;
}
```

Podemos imaginar que, a cada iteração, a palavra a desliza para a direita ao longo do texto b, como no seguinte exemplo:

```
X C B A B X C B A A X B C B A B X
B C B A
B C B A
B C B A
B C B A
B C B A
etc.
```

No pior caso, a função `inocente` compara cada elemento de a com cada elemento de b e portanto consome tempo proporcional a

$m \cdot n$ .

Será possível resolver o problema sem comparar cada elemento de a com cada elemento de b?

## Exercícios 2

1. O algoritmo inocente funciona corretamente quando  $m > n$ ? Que acontece [se tentarmos executar a função com argumento m igual a 0?](#)
2. Dê um exemplo em que o algoritmo inocente faz o maior número possível de comparações entre elementos de a e b. Qual é esse número *exatamente*?
3. SOLUÇÃO DO PROBLEMA ORIGINAL. Modifique a função `inocente` de modo que ela resolva a [versão original](#) do problema da busca de palavra em texto: para cada ocorrência de a em b, imprima o índice j tal que  $a[1..m]$  casa com  $b[j..m+j-1]$ .
4. Familiarize-se com a função `strstr` da [biblioteca string](#) que localiza a primeira ocorrência de uma string em outra. Procure descobrir o algoritmo que `strstr` implementa.
5. ★ ESPAÇOS CONSECUTIVOS [Sedgewick 3.62] Escreva uma função que receba um vetor de bytes  $b[1..n]$  e um inteiro  $m$  e devolva a posição da primeira ocorrência de  $m$  [espaços](#) (bytes 32) consecutivos em b. (Você pode imaginar que os elementos de b representam [caracteres ASCII](#),

embora isso seja irrelevante.) Procure examinar o menor número possível de elementos de b. Escreva um programa para testar sua função.

## Primeiro algoritmo de Boyer-Moore

Um *alfabeto* de uma [instância do problema](#) é qualquer conjunto de bytes que contém todos os elementos dos vetores a e b. [É claro que](#) toda instância admite 0..255 como alfabeto. Mas algumas instâncias podem ter um alfabeto menor, como 0..127 no caso de caracteres ASCII, ou como A C G T no caso das aplicações à genética.

[R.S. Boyer e J.S. Moore](#) tiveram a engenhosa ideia de usar uma tabela auxiliar indexada pelo alfabeto para acelerar o [algoritmo inocente](#). Suponha que já comparamos a[1..m] com um sufixo de b[1..k]. Agora, podemos *saltar algumas iterações* do algoritmo inocente e passar a comparar a[1..m] com um sufixo de

b[1..k+d]

para algum d positivo. O valor de d é escolhido de tal modo que a posição k+1 de b fique emparelhada com a última ocorrência (contando da esquerda para a direita) do byte b[k+1] em a[1..m]. No exemplo abaixo, marcamos com | as posições que fazem o papel de k+d. No caso em que há casamento, a marca | foi substituída por ↓ :

|         |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X       | C | B | A | B | X | C | B | A | A | X | B | C | B | A | B | X |
|         |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| B       | C | B | A |   |   |   |   |   |   |   |   |   |   |   |   |   |
| B C B A |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| B       | C | B | A |   |   |   |   |   |   |   |   |   |   |   |   |   |
| B C B A |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| B       | C | B | A |   |   |   |   |   |   |   |   |   |   |   |   |   |
| B C B A |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| B       | C | B | A |   |   |   |   |   |   |   |   |   |   |   |   |   |

Para implementar a ideia, basta pré-processar a palavra a de modo a lembrar, para cada “letra” f do alfabeto, △ a posição da *última ocorrência* (contando da esquerda para direita) de f em a. Essa posição será denotada por ult[f]. Se o alfabeto do exemplo anterior é o conjunto dos 128 caracteres ASCII, teremos

|         |     |    |   |   |   |   |   |   |   |   |   |     |     |
|---------|-----|----|---|---|---|---|---|---|---|---|---|-----|-----|
| f       | ... | => | ? | @ | A | B | C | D | E | F | G | ... |     |
| ult[f]  | ... | 0  | 0 | 0 | 0 | 4 | 3 | 2 | 0 | 0 | 0 | 0   | ... |
| 1 2 3 4 |     |    |   |   |   |   |   |   |   |   |   |     |     |
| B C B A |     |    |   |   |   |   |   |   |   |   |   |     |     |

Segue uma implementação do algoritmo. O primeiro processo iterativo faz o pré-processamento da palavra e o segundo cuida de contar as ocorrências da palavra no texto.

```
typedef unsigned char byte;

// Recebe vetores a[1..m] e b[1..n] de
// bytes, com m >= 1 e n >= 0, e
// devolve o número de ocorrências
// de a em b.

int
boyermoore1 (byte a[], int m,
 byte b[], int n)
{
 int ult[256]; // o alfabeto é 0..255

 // pré-processamento da palavra a
```

```

for (int f = 0; f < 256; ++f) ult[f] = 0;
for (int i = 1; i <= m; ++i) ult[a[i]] = i;

// busca da palavra a no texto b
int ocorrs = 0;
int k = m;
while (k <= n) {
 // a[1..m] casa com b[k-m+1..k]?
 int i = m, j = k;
 while (i >= 1 && a[i] == b[j])
 --i, --j;
 if (i < 1) ++ocorrs;
 if (k == n) k += 1;
 else k += m - ult[b[k+1]] + 1;
}
return ocorrs;
}

```

Esse é o primeiro algoritmo de Boyer-Moore.

## Exercícios 3

1. Responda rápido: o seguinte fragmento de código funciona corretamente?

```
for (byte f = 0; f < 256; ++f) ult[f] = 0;
```

2. Verifique que depois do pré-processamento temos  $1 \leq \text{ult}[f] \leq f$  para cada  $f$ .

3. ★ Que acontece na linha “ $k += m - \text{ult}[b[k+1]] + 1$ ” se o byte  $b[k+1]$  não ocorre em  $a[1..m]$ ? Que acontece se  $b[k+1]$  for igual a  $a[m]$ ?

4. Prove que a fase de pré-processamento na função boyermoore1 preenche corretamente a tabela `ult`.

5. A seguinte variante do código da busca da palavra a no texto b está correta?

```

ocorrs = 0; k = m;
while (k <= n) {
 i = m, j = k;
 while (i >= 1 && a[i] == b[j]) --i, --j;
 if (i < 1) ++ocorrs;
 kk = k+1;
 while (kk <= n && ult[b[kk]] == 0) ++kk;
 if (kk > n) break;
 k += m - ult[b[kk]] + kk - k;
}
return ocorrs;
}

```

6. Podemos eliminar o “`if (k == n) k += 1`” se introduzirmos uma setinela apropriada na posição  $b[n+1]$ . Escreva essa variante de `boyermoore1`.

7. ★ Mostre que a seguinte variante da função `boyermoore1` está correta:

```

int ult[256];
for (int i = 0; i < 256; ++i) ult[i] = 0;
for (int i = 1; i < m; ++i) ult[a[i]] = i;
int ocorrs = 0, k = m;
while (k <= n) {
 int i = m, j = k;
 while (i >= 1 && a[i] == b[j])
 --i, --j;
 if (i < 1) ++ocorrs;
 k += m - ult[b[k]];
}
return ocorrs;
}

```

## Segundo algoritmo de Boyer-Moore

O segundo algoritmo de Boyer-Moore, ao contrário do primeiro, não precisa conhecer o alfabeto de  $a$  e  $b$  explicitamente. Em compensação, é essencial que compare a palavra com o texto [da direita para a esquerda](#): primeiro  $a[m]$  com  $b[k]$ , depois  $a[m-1]$  com  $b[k-1]$ , e assim por diante.

Considere o seguinte exemplo. Suponha que já descobrimos que  $a[h..m]$  casa com um sufixo de  $b[1..k]$ . Suponha também  $\Delta$  que  $a[h..m]$  *não* casa

com  $a[h-1..m-1]$ , nem com  $a[h-2..m-2]$ , nem com  $a[h-3..m-3]$ .

(Note que estamos comparando  $a[h..m]$  com segmentos do próprio vetor  $a[1..m]!$ )

|                 |   |   |
|-----------------|---|---|
| 1               | h | m |
| & C B A * C B A |   |   |
| & C B A * C B A |   |   |

É fácil deduzir então, sem fazer quaisquer comparações adicionais, que  $a[1..m]$  não casa com um sufixo

de  $b[1..k+1]$ , nem de  $b[1..k+2]$ , nem de  $b[1..k+3]$ .

Portanto, no próximo passo, devemos tentar casar  $a[1..m]$  com um sufixo de  $b[1..k+4]$ . Em particular, devemos tentar casar  $a[h-4..m-4]$  com  $b[k-m+h..k]$ . Mas isso é o mesmo que verificar se  $a[h-4..m-4]$  casa com  $a[h..m]$ , uma vez que, por hipótese,  $b[k-m+h..k]$  casa com  $a[h..m]$ .

Para completar o exemplo acima, suponha que  $a[1..m]$  de fato casa com um sufixo de  $b[1..k+4]$ . Se  $h-4 \geq 1$  então  $a[h..m]$  *casa com um sufixo de*  $a[1..m-4]$ .

|                 |               |           |  |  |  |  |
|-----------------|---------------|-----------|--|--|--|--|
| k               |               |           |  |  |  |  |
| - - - - -       | C B A * C B A | - - - - - |  |  |  |  |
| 1               | h             | m         |  |  |  |  |
| & C B A * C B A |               |           |  |  |  |  |
| & C B A * C B A |               |           |  |  |  |  |

Por outro lado, se  $h-4 < 1$  então  $a[1..m-4]$  *casa com um sufixo de*  $a[h..m]$ .

|             |               |           |  |  |  |  |
|-------------|---------------|-----------|--|--|--|--|
| k           |               |           |  |  |  |  |
| - - - - -   | C B A * C B A | - - - - - |  |  |  |  |
| 1           | h             | m         |  |  |  |  |
| B A * C B A |               |           |  |  |  |  |
| B A * C B A |               |           |  |  |  |  |

Esse exemplo sugere que a implementação da ideia deve começar com o seguinte pré-processamento da palavra  $a$ : para cada  $h$  em  $1..m$ , calcule o maior  $k$  em  $1..m-1$  tal que  $\Delta$

- $a[h..m]$  casa com um sufixo de  $a[1..k]$  ou
- $a[1..k]$  casa com um sufixo de  $a[h..m]$ .

Na falta de um termo melhor, diremos que esse valor máximo de  $k$  é o *pulo*, ou *jump*, de  $h$ . Eis alguns exemplos de palavras com a correspondente tabela jump:

|             |                     |
|-------------|---------------------|
| 1 2 3 4 5 6 | h 6 5 4 3 2 1       |
| C A A B A A | jump[h] 5 3 0 0 0 0 |

|                 |                         |
|-----------------|-------------------------|
| 1 2 3 4 5 6 7 8 | h 8 7 6 5 4 3 2 1       |
| B A - B A * B A | jump[h] 5 5 2 2 2 2 2 2 |

|                         |                           |
|-------------------------|---------------------------|
| 1 2 3 4 5 6 7 8 9 10 11 | h 11 10 9 8 7 6 5 4 3 2 1 |
|-------------------------|---------------------------|

```
B A - B A * B A * B A jump[h] 8 8888222222
```

Depois dessa preparação, podemos examinar a implementação do segundo algoritmo de Boyer-Moore:

```
typedef unsigned char byte;

// Recebe uma palavra a[1..m] com 1 <= m e
// um texto b[1..n]. Devolve o número de
// ocorrências de a em b.

int
boyermoore2 (byte a[], int m,
 byte b[], int n)
{
 int *jump = malloc ((m+1) * sizeof (int));
 // usaremos jump[1..m]

 // pré-processamento da palavra a
 int h = m, k = m-1;
 while (h >= 1 && k >= 1) {
 int i = m, j = k;
 while (i >= h && j >= 1)
 if (a[i] == a[j]) --i, --j;
 else i = m, j = --k;
 jump[h--] = k;
 }
 while (h >= 1)
 jump[h--] = k;

 // busca da palavra a no texto b
 int ocorrs = 0;
 k = m;
 while (k <= n) {
 int i = m, j = k;
 while (i >= 1 && a[i] == b[j])
 --i, --j;
 if (i < 1) ++ocorrs;
 if (i == m) k += 1;
 else k += m - jump[i+1];
 }
 return ocorrs;
}
```

Segue uma versão mais compacta e eficiente do pré-processamento:

```
// pré-processamento da palavra a
h = m, k = m-1;
i = m, j = k;
while (h >= 1) {
 while (i >= h && j >= 1)
 if (a[i] == a[j]) --i, --j;
 else i = m, j = --k;
 jump[h--] = k;
}
```

## Exercícios 4

1. Mostre que a fase de pré-processamento na função boyermoore2 preenche corretamente a tabela jump. Mostre que essa fase consome  $m^2$  unidades de tempo no pior caso.
2. Poderíamos eliminar o "if ( $i == m$ )  $k += 1$ " se introduzíssemos uma setinela apropriada na posição  $\text{jump}[m+1]$ . Escreva essa variante de boyermoore2.
3. Mostre que a seguinte variante do pré-processamento está correta:

```

i = m;
j = k = m-1;
for (h = m; h >= 1; --h) {
 while (i >= h && j >= 1)
 if (a[i] == a[j]) --i, --j;
 else i = m, j = --k;
 jump[h] = k;
}

```

4. Mostre que a seguinte variante do pré-processamento está correta:

```

k = m-1;
r = 0;
for (h = m; h >= 1; --h) {
 while (m-r >= h && k-r >= 1)
 if (a[m-r] == a[k-r]) ++r;
 else r = 0, k--;
 jump[h] = k;
}

```

5. Use a função boyermoore2 para contar o número de ocorrências da palavra ABCBCCABC no texto ABCCBAABCABCBCABC.

## Terceiro algoritmo de Boyer-Moore

O terceiro algoritmo de Boyer-Moore é uma fusão dos dois anteriores. A cada passo, o algoritmo escolhe o maior dos dois deslocamentos: aquele ditado pela [tabela `ult`](#) e aquele dado pela [tabela `jump`](#). (Esse é o *algoritmo de Boyer-Moore* propriamente dito. A distinção que fizemos acima entre primeiro e segundo algoritmos é apenas didática.)

O pré-processamento consome  $m^2$  unidades de tempo. Infelizmente, a fase de busca consome  $m n$  unidades de tempo no pior caso, tal como no algoritmo inocente.

Mas o pior caso é tão raro que no caso médio, típico de aplicações práticas, o terceiro algoritmo de Boyer-Moore consome tempo proporcional a  $n$  e independente de  $m$ . Ou seja, em média, cada elemento do texto é comparado com apenas alguns poucos elementos da palavra, qualquer que seja o comprimento da palavra.

A definição da tabela `jump` pode ser aperfeiçoada de tal maneira que, mesmo no pior caso, a fase de busca do terceiro algoritmo [consuma apenas  \$6n\$  unidades de tempo](#).

## Exercícios 5

1. Implemente o terceiro algoritmo de Boyer-Moore.
2. ★ TESTES. Compare, experimentalmente, o desempenho do terceiro algoritmo de Boyer-Moore com o do [algoritmo inocente](#). Invente pares palavra/texto interessantes para fazer os testes. Para fazer os testes, você pode usar o arquivo br-utf8.txt da “[Lista de todas as palavras do português brasileiro](#)” e o livro “[Quincas Borba](#)” de Machado de Assis. Também pode usar uma lista de palavras em inglês e o livro “[A Tale of Two Cities](#)” de Charles Dickens.
3. ★ DESAFIO. Investigue as alterações que devem ser feitas na definição da tabela `jump` para que o terceiro algoritmo de Boyer-Moore faça no máximo  $6n$  comparações entre elementos da palavra e do texto na fase de busca.
4. Escreva uma função que receba vetores  $b[1..n]$ ,  $a[1..m]$  e  $x[1..p]$  e substitua por  $x$  cada ocorrência de  $a$  em  $b$ .
5. BUSCA COM CURINGA. Suponha que o caractere # tem um significado especial dentro da palavra: ele representa 0 ou mais ocorrências de qualquer outro caractere (ou seja, # é um curinga ou *wildcard*). Exemplos:
  - a palavra A#B#C casa com qualquer segmento do texto que comece com A, termine com C e tenha um B em algum lugar entre A e C;
  - a palavra x#[#]#=#; casa com  $x[i] = x[i-1] + 1$ ; e também com

`x2[i]=x[i-1]+1; y= z; .`

Escreva uma função que busque uma palavra em um texto interpretando o caractere # como sugerido acima.

6. ★ Familiarize-se com o poderoso utilitário [grep](#), que faz busca de palavras em texto.

---

Veja o verbete [String searching algorithm](#) na Wikipedia.

---

Veja o sítio [Exact String Matching Algorithms](#), com animação de algoritmos de busca de palavras em texto.

---

Veja também a bibliografia [Pattern Matching Pointers](#).

Atualizado em 2018-08-13

<https://www.ime.usp.br/~pf/algoritmos/>

*Paulo Feofiloff*

[\*DCC-IME-USP\*](#)



# Árvores binárias

As árvores da computação têm a curiosa tendência de crescer para baixo...

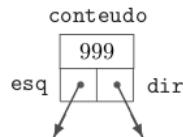


Uma árvore binária é uma estrutura de dados mais geral que uma [lista encadeada](#). Este capítulo introduz algumas operações básicas sobre árvores binárias. O capítulo seguinte, [Árvores binárias de busca](#), trata de uma aplicação fundamental.

## Nós e filhos

Uma árvore binária (= *binary tree*) é um conjunto de [registros](#) que satisfaz certas condições. As condições não serão dadas explicitamente, mas elas ficarão implicitamente claras no contexto. Os registros serão chamados *nós* (poderiam também ser chamados *células*). Cada nó tem um [endereço](#). Suporemos por enquanto que cada nó tem apenas três campos: um número inteiro e dois [ponteiros para nós](#). Os nós podem, então, ser definidos assim:

```
typedef struct reg {
 int conteudo; // conteúdo
 noh *esq;
 noh *dir;
} noh; // nó
```

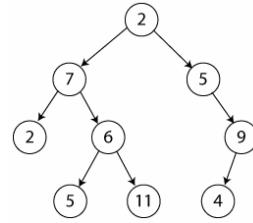


O campo *conteúdo* é a “carga útil” do nó; os dois outros campos servem apenas para dar estrutura à árvore. O campo *esq* de cada nó contém `NULL` ou o endereço de outro nó. Uma hipótese análoga vale para o campo *dir*.

Se o campo *esq* de um nó *P* é o endereço de um nó *E*, diremos que *E* é o *filho esquerdo* de *P*. Analogamente, se *P.dir* é igual a `&D`, diremos que *D* é o *filho direito* de *P*. Se um nó *F* é filho (esquerdo ou direito) de *P*, diremos que *P* é o *pai* de *F*. Uma *folha* (= *leaf*) é um nó que não tem filho algum.

É muito conveniente confundir, verbalmente, cada nó com seu endereço. Assim, se *x* é um ponteiro para um nó (ou seja, se *x* é do tipo `*noh`), dizemos “considere o nó *x*” em lugar de “considere o nó cujo endereço é *x*”.

A figura abaixo mostra dois exemplos de árvores binárias. Do lado esquerdo, temos uma curiosa árvore binária na natureza. Do lado direito, uma representação esquemática de uma árvore binária cujos nós contêm os números 2, 7, 5, etc.



## Exercícios 1

- Dê uma lista das [condições](#) que um conjunto de nós deve satisfazer para ser considerado uma árvore binária.

## Árvores e subárvores

Suponha que  $x$  é um nó. Um *descendente* de  $x$  é qualquer nó que possa ser alcançado pela iteração das instruções  $x = x->\text{esq}$  e  $x = x->\text{dir}$  em qualquer ordem. (É claro que essas instruções só podem ser iteradas enquanto  $x$  for diferente de NULL. Estamos supondo que NULL é de fato atingido mais cedo ou mais tarde.)

Um nó  $x$  juntamente com todos os seus descendentes é uma *árvore binária*. Dizemos que  $x$  é *raiz* (= *root*) da árvore. Se  $x$  tiver um pai, essa árvore é *subárvore* de alguma árvore maior. Se  $x$  é NULL, a árvore é *vazia*.

Para qualquer nó  $x$ , o nó  $x->\text{esq}$  é a raiz da *subárvore esquerda* de  $x$  e  $x->\text{dir}$  é a raiz da *subárvore direita* de  $x$ .

## Endereço de uma árvore e definição recursiva

O *endereço* de uma árvore binária é o endereço de sua raiz. É conveniente confundir, verbalmente, árvores com seus endereços: dizemos “considere a árvore  $r$ ” em lugar de “considere a árvore cuja raiz tem endereço  $r$ ”. Essa convenção sugere a introdução do nome alternativo *arvore* para o tipo-de-dados ponteiro-para-nó:

```
typedef noh *arvore; // árvore
```

A convenção permite formular a definição de árvore binária de maneira recursiva: um ponteiro-para-nó  $r$  é uma árvore binária se

- $r$  é NULL ou
- $r->\text{esq}$  e  $r->\text{dir}$  são árvores binárias.

Muitos algoritmos sobre árvores ficam mais simples quando escritos em estilo recursivo.

## Exercícios 2

- Árvores binárias têm uma relação muito íntima com certas [sequências bem-formadas de parênteses](#). Discuta essa relação.
- ★ EXPRESSÕES ARITMÉTICAS. Árvores binárias podem ser usadas, de maneira muito natural, para

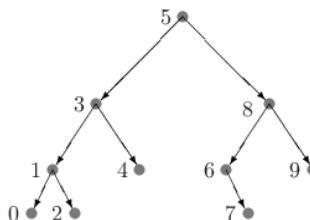
representar expressões aritméticas (como  $((a+b)*c-d)/(e-f)+g$ , por exemplo). Discuta os detalhes.

## Varredura esquerda-raiz-direita

Uma árvore binária pode ser percorrida de muitas maneiras diferentes. Uma maneira particularmente importante é a esquerda-raiz-direita, ou e-r-d, também conhecida como *inorder traversal*, ou varredura infixa, ou varredura central. A *varredura e-r-d* visita

1. a subárvore esquerda da raiz, em ordem e-r-d,
2. a raiz,
3. a subárvore direita da raiz, em ordem e-r-d,

nessa ordem. Na seguinte figura, os nós estão numerados na ordem em que são visitados pela varredura e-r-d.



Eis uma função recursiva que faz a varredura e-r-d de uma árvore binária r:

```
// Recebe a raiz r de uma árvore binária e
// imprime os conteúdos dos seus nós
// em ordem e-r-d.

void erd (arvore r) {
 if (r != NULL) {
 erd (r->esq);
 printf ("%d\n", r->conteudo);
 erd (r->dir);
 }
}
```

É um excelente exercício escrever uma versão iterativa dessa função. A versão abaixo usa uma [pilha](#) de nós. A sequência de nós na pilha é uma espécie de “roteiro” daquilo que ainda precisa ser feito: cada nó x na pilha é um lembrete de que ainda precisamos imprimir o conteúdo de x e o conteúdo da subárvore direita de x. O elemento do topo da pilha pode ser um NULL, mas os demais elementos são diferentes de NULL.

```
// Recebe a raiz r de uma árvore binária e
// imprime os conteúdos dos seus nós
// em ordem e-r-d.

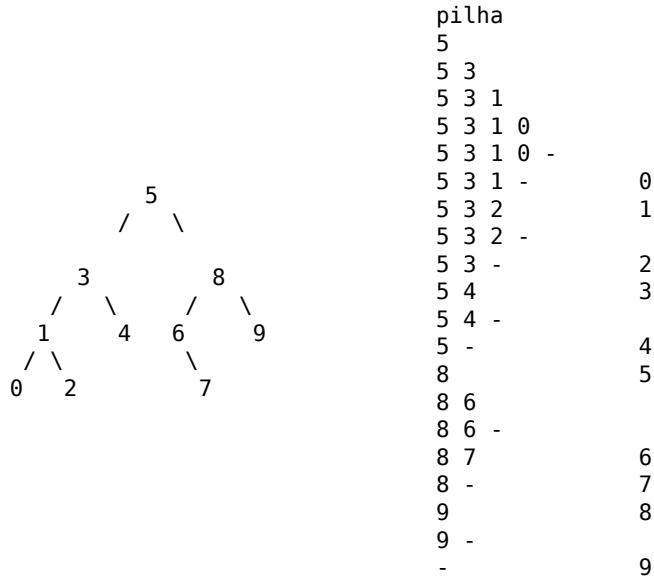
void erd_i (arvore r) {
 criapilha (); // pilha de nós
 empilha (r);
 while (true) {
 x = desempilha ();
 if (x != NULL) {
 empilha (x);
 empilha (x->esq);
 }
 else {
 if (pilhavazia ()) break;
 x = desempilha ();
 printf ("%d\n", x->conteudo);
 empilha (x->dir);
 }
 }
}
```

```

 }
 liberapilha ();
}

```

(O código ficaria ligeiramente mais simples — mas um pouco menos legível — se manipulasse a pilha diretamente.) A figura abaixo dá um exemplo de execução da função `erd_i`. Cada linha da tabela resume o estado das coisas no início de uma iteração: à esquerda está a pilha e à direita os nós impressos. O valor `NULL` é indicado por `-`.



Os exercícios abaixo discutem duas outras ordens de varredura de uma árvore binária: a *varredura r-e-d*, que percorre a árvore em ordem raiz-esquerda-direita, e a *varredura e-d-r*, que percorre a árvore em ordem esquerda-direita-raiz. A primeira também é conhecida como *preorder traversal*, ou varredura em pré-ordem, ou varredura prefixa. A segunda também é conhecida como *postorder traversal*, ou varredura em pós-ordem, ou varredura posfixa. (A propósito, veja [abaixo](#) o exercício sobre notações infixas, posfixas e prefixas.)

## Exercícios 3

1. Considere a função `erd_i`. É verdade que a sequência de nós na pilha é um caminho que começa em algum nó e segue os ponteiros esquerdo e direito em alguma ordem?
2. Verifique que o código abaixo é equivalente ao da função `erd_i`. O código usa uma pilha de nós, todos diferentes de `NULL`, e mais um nó `x` (que é candidato a entrar na pilha mas pode ser `NULL`).

```

void erd_i (arvore r) {
 noh *x = r;
 criapilha ();
 while (true) {
 while (x != NULL) {
 empilha (x);
 x = x->esq;
 }
 if (pilhavazia ()) break;
 x = desempilha ();
 printf ("%d\n", x->conteudo);
 x = x->dir;
 }
 liberapilha ();
}

```

3. NÚMERO DE NÓS. Escreva uma função que calcule o número de nós de uma árvore binária.
4. FOLHAS. Escreva uma função que imprima, em ordem e-r-d, os conteúdos das *folhas* de uma árvore binária.
5. Dada uma árvore binária, encontrar um nó da árvore cujo conteúdo tenha um dado valor `val`.
6. VARREDURA R-E-D. Escreva uma função que faça a varredura r-e-d (varredura prefixa) de uma árvore

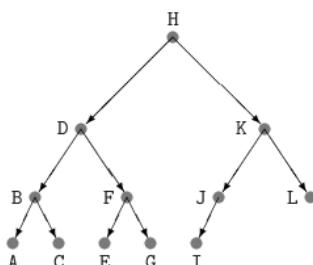
binária. (A varredura r-e-d também é conhecida como “busca em profundidade” ou *depth-first search*.)

7. VARREDURA E-D-R. Escreva uma função que faça varredura e-d-r (varredura posfixa) de uma árvore binária.
8. NOTAÇÕES INFIXA, POSFIXA E PREFIXA. Mostre que a varredura e-r-d da [árvore de uma expressão aritmética](#) corresponde exatamente à [representação infixa](#) da expressão. Mostre que a varredura e-d-r da árvore de uma expressão aritmética corresponde exatamente à representação da expressão em [notação posfixa](#). Mostre que a varredura r-e-d da árvore de uma expressão aritmética corresponde exatamente à notação prefixa.

## Altura e profundidade

A *altura de um nó*  $x$  em uma árvore binária é a distância entre  $x$  e o seu descendente mais afastado. Mais precisamente, a altura de  $x$  é o número de passos no mais longo caminho que leva de  $x$  até uma folha. Os caminhos a que essa definição se refere são os obtido pela iteração das instruções  $x = x->esq$  e  $x = x->dir$ , em qualquer ordem.

A *altura (= height)*  $x$  de uma árvore é a altura da raiz da árvore. Uma árvore com um único nó tem altura 0. A árvore da figura tem altura 3.



Veja como a altura de uma árvore com raiz  $r$  pode ser calculada:

```

// Devolve a altura da árvore binária
// cuja raiz é r.

int altura (arvore r) {
 if (r == NULL)
 return -1; // altura da árvore vazia
 else {
 int he = altura (r->esq);
 int hd = altura (r->dir);
 if (he < hd) return hd + 1;
 else return he + 1;
 }
}

```

Qual a relação entre a altura, digamos  $h$ , e o número de nós de uma árvore binária? Se a árvore tem  $n$  nós então

$$n-1 \geq h \geq \lg(n),$$

onde [lg\(n\)](#) denota  $\lfloor \log n \rfloor$ , ou seja, o [piso](#) de [log n](#). Uma árvore binária de altura  $n-1$  é um “tronco sem galhos”: cada nó tem no máximo um filho. No outro extremo, uma árvore de altura  $\lg(n)$  é “quase completa”: todos os “níveis” estão lotados exceto talvez o último.

| $n$ | $\lg(n)$ |
|-----|----------|
| 4   | 2        |
| 5   | 2        |
| 6   | 2        |
| 10  | 3        |
| 64  | 6        |
| 100 | 6        |
| 128 | 7        |

|         |    |
|---------|----|
| 1000    | 9  |
| 1024    | 10 |
| 1000000 | 19 |

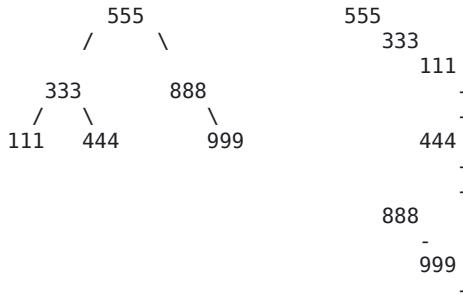
Uma árvore binária é *balanceada* (ou *equilibrada*) se, em cada um de seus nós, as [subárvores](#) esquerda e direita tiverem aproximadamente a mesma altura. Uma árvore binária balanceada com  $n$  nós tem altura próxima de  $\log n$ . (A árvore do exemplo [acima](#) é balanceada). Convém trabalhar com árvores balanceadas sempre que possível. Mas isso não é fácil se a árvore aumenta e diminui ao longo da execução do seu programa.

**Profundidade.** A *profundidade* (= *depth*) de um nó  $s$  em uma árvore binária com raiz  $r$  é a distância de  $r$  a  $s$ . Mais precisamente, a profundidade de  $s$  é o comprimento do único caminho que vai de  $r$  até  $s$ . Por exemplo, a profundidade de  $r$  é 0 e a profundidade de  $r->esq$  é 1.

Uma árvore é balanceada se todas as suas folhas têm aproximadamente a mesma profundidade.

## Exercícios 4

1. Desenhe uma árvore binária que com 17 nós que tenha a menor altura possível. Repita com a maior altura possível.
2. Escreva uma função iterativa para calcular a altura de uma árvore binária.
3. Escreva uma função que imprima o conteúdo de cada nó de uma árvore binária. Faça uma indentação (recuo de margem) proporcional à profundidade do nó. Segue um exemplo de árvore e sua representação (os caracteres '-' representam NULL):



4. VARREDURA POR NÍVEIS. A *varredura por níveis* (= *level-traversal*), ou [busca em largura](#) (= *breadth-first search*), de uma árvore binária visita todos os nós à profundidade 0, depois todos os nós à profundidade 1, depois todos os nós à profundidade 2, e assim por diante. Para cada profundidade, os nós são visitados "da esquerda para a direita". Escreva uma função que imprima os conteúdos dos nós de uma árvore na ordem da varredura por níveis. (Dica: Use uma [fila](#) de nós. Veja o [cálculo de distâncias em um grafo](#).)
5. Digamos que  $h$  é a altura e  $p$  é a profundidade de um nó  $x$  em uma árvore binária. É verdade que  $h + p$  é igual à altura da árvore?
6. Escreva uma função que determine a profundidade de um nó dado.
7. Escreva uma função que decida se uma dada árvore binária é quase completa.
8. DE HEAP PARA ÁRVORE. Escreva uma função que transforme um [heap](#)  $v[1..n]$  em uma árvore binária (quase completa).
9. DE ÁRVORE PARA HEAP. Uma árvore binária é *hierárquica* se  $x->conteudo \geq x->esq->conteudo$  e  $x->conteudo \geq x->dir->conteudo$  para todo nó  $x$ , desde que os filhos existam. É possível transformar qualquer árvore hierárquica quase completa em [heap](#)?
10. HIERARQUIZAÇÃO. Uma árvore binária é *esquerdista* se, para todo nó  $x$ ,  $x->esq == \text{NULL}$  implica em  $x->dir == \text{NULL}$ . Uma árvore binária é *hierárquica* se  $x->conteudo \geq x->esq->conteudo$  e  $x->conteudo \geq x->dir->conteudo$  para todo nó  $x$ , desde que os filhos existam. Escreva uma função que movimente os campos *conteudo* de uma árvore esquerdista de modo a torná-la hierárquica. (Sugestão: Imitie a função que transforma um vetor em [heap](#). Em particular, imite a função [peneira](#).)

11. ÁRVORE AVL. Uma árvore é balanceada *no sentido AVL* se, para cada nó  $x$ , as alturas das subárvore que têm raízes  $x->esq$  e  $x->dir$  diferem de no máximo uma unidade. Escreva uma função que decida se uma dada árvore é balanceada no sentido AVL. Procure escrever sua função de modo que ela visite cada nó no máximo uma vez.

## Nós com campo pai

Em algumas aplicações (veja a seção seguinte) é conveniente ter acesso direto ao pai de qualquer nó. Para isso, é preciso acrescentar um campo *pai* a cada nó:

```
typedef struct reg {
 int conteudo;
 struct reg *pai;
 struct reg *esq, *dir;
} noh;
```

É um bom exercício escrever uma função que preencha o campo *pai* de todos os nós de uma árvore binária.

## Exercícios 5

1. Escreva uma função que preencha corretamente todos os campos *pai* de uma árvore binária.
2. FILA DE PRIORIDADES. Uma árvore binária quase completa com campo *pai* é *hierárquica* se  $x->conteudo \geq x->esq->conteudo$  e  $x->conteudo \geq x->dir->conteudo$  para todo nó  $x$  (desde que os filhos existam). Escreva uma implementação de [fila de prioridades](#) baseada em árvore hierárquica. A implementação deve ter funções para criar uma fila vazia, retirar um elemento máximo da fila, e inserir um elemento na fila. (Dica: veja a implementação de [fila de prioridades baseada em heap](#).)

## Primeiro e último nós

Considere o seguinte [problema](#) sobre uma árvore binária: encontrar o endereço do *primeiro* nó da árvore na ordem e-r-d. É claro que o problema só faz sentido se a árvore não é vazia. Eis uma função que resolve o problema:

```
// Recebe uma árvore binária não vazia r
// e devolve o primeiro nó da árvore
// na ordem e-r-d.

noh *primeiro (arvore r) {
 while (r->esq != NULL)
 r = r->esq;
 return r;
}
```

Não é difícil fazer uma função análoga que encontre o *último* nó na ordem e-r-d.

## Nó seguinte e anterior (sucessor e predecessor)

Digamos que  $x$  é o endereço de um nó de uma árvore binária. Nossa problema: calcular o endereço do nó seguinte na ordem e-r-d.

Para resolver o problema, é necessário que os nós tenham um [campo \*pai\*](#). A função a seguir resolve o problema. É claro que a função só deve ser chamada com  $x$  diferente de NULL. A função

devolve o endereço do nó seguinte a x ou devolve NULL se x é o último nó.

```
// Recebe o endereço de um nó x. Devolve o endereço
// do nó seguinte na ordem e-r-d.
// A função supõe que x != NULL.

noh *seguinte (noh *x) {
 if (x->dir != NULL) {
 noh *y = x->dir;
 while (y->esq != NULL) y = y->esq;
 return y; // A
 }
 while (x->pai != NULL && x->pai->dir == x) // B
 x = x->pai; // B
 return x->pai;
}
```

(Note que a função não precisa saber onde está a raiz da árvore.) Na linha A, y é o endereço do primeiro nó, na ordem e-r-d, da subárvore cuja raiz é x->dir. As linhas B fazem com que x suba na árvore enquanto for filho direito de alguém.

## Exercícios 6

1. Escreva uma versão recursiva da função primeiro.
2. Escreva uma função que encontre o último nó na ordem e-r-d.
3. Escreva uma função que receba o endereço de um nó x de uma árvore binária e encontre o endereço do nó anterior a x na ordem e-r-d.
4. Escreva uma função que faça varredura e-r-d usando as funções primeiro e seguinte.

---

Atualizado em 2018-08-07

<https://www.ime.usp.br/~pf/algoritmos/>

*Paulo Feofiloff*  
[DCC-IME-USP](#)



# Árvores binárias de busca

Assim como [árvores binárias](#) generalizam a ideia de [listas encadeadas](#), árvores binárias de busca (= *binary search trees = BSTs*) generalizam a ideia de listas encadeadas crescentes.

## Definição

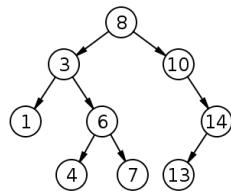
Considere uma [árvore binária](#) cujos nós têm um campo chave de um tipo (como `int`, `char`, `string`, etc.) que admite comparação. Neste capítulo, suparemos que as chaves são do tipo `int`. Suparemos também que os nós da árvore têm a seguinte [estrutura](#):

```
typedef struct reg {
 int chave;
 int conteudo;
 struct reg *esq, *dir;
} noh; // nó
```

Uma árvore binária deste tipo é *de busca* se [cada nó](#) p tem a seguinte propriedade:  $\Delta$  a chave de p é (1) maior ou igual à chave de cada nó da [subárvore](#) esquerda de p e (2) menor ou igual à chave de cada nó da subárvore direita de p. Em outras palavras, se p é um nó qualquer então

$$e->\text{chave} \leq p->\text{chave} \leq d->\text{chave}$$

para todo nó e na subárvore esquerda de p e todo nó d na subárvore direita de p. Eis outra maneira, talvez mais clara, de descrever a propriedade que define árvores de busca: *a ordem e-r-d das chaves é crescente*.



## Exercícios 1

1. ★ Suponha que  $x->\text{esq}->\text{chave} \leq x->\text{chave}$  para cada nó x dotado de filho esquerdo e  $x->\text{chave} \leq x->\text{dir}->\text{chave}$  para cada nó x dotado de filho direito. Essa árvore é de busca?
2. ★ IMPORTANTE! Escreva uma função que decida se uma dada árvore binária é ou não é de busca.

## Busca

Considere o seguinte [problema](#): Dada uma árvore de busca r e um número k, encontrar um nó de r cuja chave seja k. A seguinte função recursiva resolve o problema:

```

// Recebe uma árvore de busca r e
// um número k. Devolve um nó
// cuja chave é k; se tal nó não existe,
// devolve NULL.

noh *
busca (arvore r, int k) {
 if (r == NULL || r->chave == k)
 return r;
 if (r->chave > k)
 return busca (r->esq, k);
 else
 return busca (r->dir, k);
}

```

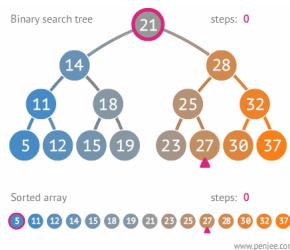
(O tipo-de-dados arvore é [igual a um ponteiro-para-nó](#).) Eis uma versão iterativa da mesma função:

```

while (r != NULL && r->chave != k) {
 if (r->chave > k)
 r = r->esq;
 else
 r = r->dir;
}
return r;

```

No pior caso, a busca consome tempo proporcional à [altura](#) da árvore. Se a árvore for [balanceada](#), o consumo será proporcional a  $\log n$ , sendo  $n$  o número de nós. Esse tempo é da mesma ordem que a [busca binária num vetor ordenado](#).



## Exercícios 2

1. Escreva uma função `min` que encontre uma chave mínima em uma árvore de busca. Escreva uma função `max` que encontre uma chave máxima.
2. Suponha que as chaves de nossa árvore de busca são distintas duas a duas. Escreva uma função que receba uma chave  $k$  e devolva a chave seguinte na ordem crescente.
3. Escreva uma função que transforme um vetor crescente em uma árvore binária de busca que seja balanceada.
4. Escreva uma função que transforme uma árvore de busca em um vetor crescente.
5. Há uma relação muito íntima entre árvores de busca e o [algoritmo de busca binária](#) num vetor crescente. Qual é, exatamente, essa relação?

## Inserção

Considere o problema de inserir um novo nó, com chave  $k$ , em uma árvore de busca. É claro que a árvore resultante deve também ser de busca. O novo nó será uma [folha](#) da árvore resultante:

```

 noh *novo;
 novo = malloc (sizeof (noh));
 novo->chave = k;
 novo->esq = novo->dir = NULL;

```

É claro que a função que resolve o problema deve devolver a raiz da nova árvore:

```

// A função recebe uma árvore de busca r
// e uma folha avulsa novo e insere a folha
// na árvore de modo que a árvore continue
// sendo de busca. A função devolve a raiz
// da árvore resultante.

arvore
insere (arvore r, noh *novo) {
 if (r == NULL) return novo;
 if (r->chave > novo->chave)
 r->esq = insere (r->esq, novo);
 else
 r->dir = insere (r->dir, novo);
 return r;
}

```

(A raiz da árvore resultante é a mesma da árvore original a menos que a árvore original seja vazia.)

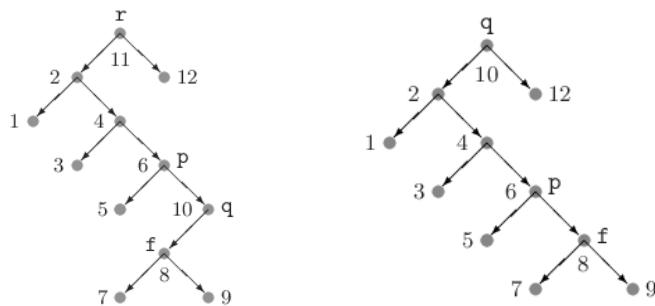
## Exercícios 3

1. Que acontece se a função `insere` acima for invocada com `r` igual a `NULL`?
2. ★ Escreva uma versão não recursiva de `insere`.
3. Suponha que as chaves 50 30 70 20 40 60 80 15 25 35 45 36 são inseridas, nesta ordem, numa árvore de busca inicialmente vazia. Desenhe a árvore que resulta.
4. Suponha que os nós de uma árvore binária de busca têm um campo pai. Escreva um função que insira um novo nó na árvore. Faça duas versões: uma recursiva e uma iterativa.

## Remoção

Problema: Remover um nó de uma árvore de busca de tal forma que a árvore continue sendo de busca.

Comecemos tratando do caso em que o nó a ser removido é a raiz da árvore. Se a raiz não tem um dos filhos, basta que o outro filho assuma o papel de raiz. Senão, faça com que o [nó anterior](#) à raiz na ordem e-r-d assuma o papel de raiz.



A figura ilustra o antes-e-depois da remoção do nó `r`. O nó anterior a `r` na ordem e-r-d é `q` (os

nós estão numerados em ordem e-r-d). O nó q é colocado no lugar de r, os filhos de r passam a ser filhos de q, e f passa a ser filho (direito) de p.

```

// Recebe uma árvore não vazia r.
// Remove a raiz da árvore e rearranja
// a árvore de modo que ela continue sendo
// de busca. Devolve o endereço da
// nova raiz.

arvore
removeraiz (arvore r) {
 noh *p, *q;
 if (r->esq == NULL) {
 q = r->dir;
 free (r);
 return q;
 }
 p = r; q = r->esq;
 while (q->dir != NULL) {
 p = q; q = q->dir;
 }
 // q é nó anterior a r na ordem e-r-d
 // p é pai de q
 if (p != r) {
 p->dir = q->esq;
 q->esq = r->esq;
 }
 q->dir = r->dir;
 free (r);
 return q;
}

```

Suponha agora que queremos remover um nó que não é a raiz da árvore. Para remover o filho esquerdo de um nó x faça

```
x->esq = removeraiz (x->esq);
```

e para remover o filho direito de x faça

```
x->dir = removeraiz (x->dir);
```

## Exercícios 4

1. Suponha que as chaves 50 30 70 20 40 60 80 15 25 35 45 36 são inseridas, nesta ordem, numa árvore de busca inicialmente vazia. Desenhe a árvore que resulta. Em seguida remova o nó que contém 30.
2. Escreva uma versão recursiva da função que remove um nó de uma árvore de busca.
3. Suponha que os nós de uma árvore binária de busca têm um campo pai. Escreva um função que remova um nó da árvore. Faça duas versões: uma recursiva e uma iterativa.

## Exercícios 5

1. Considere árvores binárias de busca cujos nós têm a estrutura indicada abaixo. Escreva uma função que receba a raiz de uma tal árvore e o endereço de um nó x e devolva o endereço do pai de x.

```

typedef struct reg {
 int chave;
 int conteudo;
 struct reg *esq, *dir;
} noh;

```

(Se  $x$  é a raiz da árvore, a função deve devolver NULL. Se  $x$  não pertence à árvore, a função também deve devolver NULL.) (Dê uma solução melhor que as respostas à pergunta [How can we find the parent of a node in a given binary tree if it does not have a pointer for the parent?](#) no Quora.)

## Desempenho dos algoritmos

Quanto tempo gastam os algoritmos de busca, inserção e remoção discutidos acima? É claro que esse tempo é limitado pelo número de nós, digamos  $n$ , da árvore (pois nenhum nó é visitado mais que 3 vezes). Mas esta é uma resposta muito grosseira. Eis uma resposta mais fina: no pior caso, qualquer dos algoritmos acima

gasta uma quantidade de tempo proporcional à [altura](#) da árvore.

Conclusão: interessa trabalhar sempre com árvores [balanceadas](#), ou seja, árvores que têm altura próxima a  $\log n$ , isto é, árvores em que todas as folhas têm aproximadamente a mesma [profundidade](#).

Infelizmente não é fácil fazer isso. A altura da árvore é sujeita a chuvas e trovoadas. É muito fácil construir um exemplo em que uma árvore começa com altura próxima de  $\log n$  mas depois de algumas inserções azaradas fica com altura muito mais próxima de  $n-1$  (claro que o valor de  $n$  muda ao longo desse processo).

Os algoritmos de inserção e remoção descritos acima *não* produzem árvores balanceadas. Se a função `insere` for repetidamente aplicada a uma árvore balanceada, o resultado pode ser uma árvore bastante desbalanceada. Algo análogo pode acontecer depois de uma sequência de chamadas da função `remove`.

Para enfrentar essa situação é preciso inventar algoritmos bem mais sofisticados e complexos de inserção e remoção; esses algoritmos fazem um rebalanceamento da árvore após cada inserção e cada remoção. Vou apenas citar dois termos técnicos pitorescos: há um pacote de algoritmos de inserção e remoção que produz “[árvores AVL](#)”; há um outro pacote que produz “[árvores rubro-negras](#)”.

---

Veja minha página [Árvores binárias de busca](#) baseada no [livro de Sedgewick e Wayne](#) (código Java).

---

Atualizado em 2018-08-02

<https://www.ime.usp.br/~pf/algoritmos/>

*Paulo Feofiloff*

*DCC-IME-USP*



# Algoritmos de enumeração

Enumere as estrelas de nossa galáxia.  
— um antigo problema

*"Often it appears that there is no better way to solve a problem than to try all possible solutions. This approach, called *exhaustive search*, is almost always slow, but sometimes it is better than nothing."*  
— Ian Parberry, *Problems on Algorithms*

Para resolver certos problemas computacionais é necessário *enumerar* — ou seja, fazer uma lista de — todos os objetos de um determinado tipo (por exemplo, todas as [árvores binárias de busca](#) com chaves entre 1 e 999). O número de objetos é tipicamente muito grande, e portanto sua enumeração consome muito tempo.

Os algoritmos de enumeração não são complexos, mas têm suas sutilezas. As versões recursivas são particularmente úteis e interessantes.

Os problemas de enumeração estão relacionados com expressões como *backtracking*, *busca exaustiva*, *força bruta*, e *branch-and-bound*.

## Sequências

Os objetos principais deste capítulo são [sequências](#) de números inteiros, como 9 51 61 4 8 2 18 97 13 22 1 3 por exemplo. Essas sequências serão representadas por [vetores](#). Assim, uma sequência como  $s_1 s_2 s_3 \dots s_k$  será denotada simplesmente por

$$s[1 \dots k].$$

Por conveniência, as sequências e vetores serão indexados a partir de 1 e não a partir de 0, como é mais comum em C.

Usaremos  $\Delta$  a abreviatura especial  $1 \dots n$  para a sequência  $1 2 3 \dots n$ . De modo mais geral,  $m \dots n$  será nossa abreviatura para  $m m+1 m+2 \dots n$ . Se  $m > n$ , a sequência é vazia.

Sequências são comparadas [lexicograficamente](#). Uma sequência  $r[1 \dots j]$  é *lexicograficamente menor* que outra  $s[1 \dots k]$  se

1. existe  $i$  tal que  $r[1 \dots i-1] = s[1 \dots i-1]$  e  $r[i] < s[i]$  ou
2.  $j < k$  e  $r[1 \dots j] = s[1 \dots j]$ .

Nas mesmas condições,  $s[1 \dots k]$  é *lexicograficamente maior* que  $r[1 \dots j]$ . Por exemplo, 4 1 3 2 é lexicograficamente maior que 4 1 2 5 6 9 e lexicograficamente menor que 4 1 3 2 5 6.

Num dado conjunto de sequências, uma sequência  $r$  é a *predecessora imediata* de outra, digamos  $s$ , se for a maior das sequências menores que  $s$  na ordem lexicográfica. O conceito de *sucessora imediata* de uma sequência é definido de maneira análoga.

## Exercícios 1

1. Escreva uma função [booleana](#) que decida se uma sequência  $r[1..j]$  é lexicograficamente menor que uma sequência  $s[1..k]$ .

## Enumeração de subsequências

Uma [subsequência](#) é o que sobra quando alguns termos de uma sequência são apagados. Mais precisamente, uma *subsequência* de  $a[1..n]$  é qualquer sequência  $s[1..k]$  tal que  $s[1] = a[i_1]$ ,  $s[2] = a[i_2]$ , ...,  $s[k] = a[i_k]$  para alguma sequência  $1 \leq i_1 < i_2 < \dots < i_k \leq n$  de índices. Em particular,  $\text{subs}[1..k]$  é uma subsequência de  $1..n$  se

$$1 \leq \text{subs}[1] < \text{subs}[2] < \dots < \text{subs}[k] \leq n.$$

Por exemplo, 2 3 5 8 é uma subsequência de 1..8.

Nosso [problema](#): Enumerar as subsequências de  $1..n$ , ou seja, fazer uma lista em que cada subsequência apareça uma e uma só vez.

O tamanho da lista cresce explosivamente com  $n$  pois o número de subsequências de  $1..n$  dobra toda vez que  $n$  aumenta em uma unidade.

A ordem em que as subsequências de  $1..n$  são enumeradas não é muito importante, mas é natural usar a ordem lexicográfica. Veja, por exemplo, a enumeração das subsequências não vazias de  $1..4$  em ordem lexicográfica:

```
1
1 2
1 2 3
1 2 3 4
1 2 4
1 3
1 3 4
1 4
2
2 3
2 3 4
2 4
3
3 4
4
```

Para transformar uma subsequência  $\text{subs}[1..k]$  de  $1..n$  na sua [sucessora imediata](#), basta fazer o seguinte:

```
if ($\text{subs}[k] < n$) {
 $\text{subs}[k+1] = \text{subs}[k] + 1$;
 ++k ;
} else {
 $\text{subs}[k-1] += 1$;
 --k ;
}
```

Esse cálculo só não está correto em dois casos: (1) se o valor original de  $k$  é 0 e (2) se o valor original de  $\text{subs}[k]$  é  $n$  e  $k$  vale 1. No primeiro caso, a sucessora imediata é definida por  $\text{subs}[k=1] = 1$ . No segundo, *não existe* sucessora alguma. A propósito, a operação dentro do bloco `else {}` é conhecida como *backtrack* ou “volta de ré”.

## Exercícios 2

- Mostre que o número de subsequências de  $1..n$  dobra toda vez que  $n$  aumenta de uma unidade. Deduza daí que o número de subsequências de  $1..n$  é  $2^n$ .
- Escreva uma função que receba uma subsequência  $\text{subs}[1..k]$  de  $1..n$  e imprima sua sucessora imediata.

## Subsequências em ordem lexicográfica

A discussão da seção anterior leva a um algoritmo iterativo que enumera as subsequências não vazias de  $1..n$ :

```
// Recebe n >= 1 e imprime todas as
// subsequências não vazias de 1..n,
// em ordem lexicográfica.

void
ssLex (int n)
{
 int *subs, k;
 subs = malloc ((n+1) * sizeof (int));
 subs[k=0] = 0;

 while (true) {
 if (subs[k] < n) {
 subs[k+1] = subs[k] + 1;
 ++k;
 } else {
 subs[k-1] += 1;
 --k;
 }
 if (k == 0) break;
 imprime (subs, k);
 }
 free (subs);
}
```

(A instrução `imprime (subs, k)` não faz mais que imprimir `subs[1..k]`.) Cada iteração começa com uma subsequência `subs[1..k]` de  $1..n$ . A primeira iteração começa com a subsequência vazia. Cada iteração gera a [sucessora imediata](#) de `subs[1..k]`. Se `subs[1..k]` não tiver sucessora, o processo termina.

A “sentinela” `subs[0]` cuida da primeira iteração (que começa com  $k$  igual a 0) e da última iteração (que começa com `subs[k]` igual a  $n$  e  $k$  igual a 1).

O vetor `subs[1..k]` comporta-se como uma [pilha](#), com topo `subs[k]`. É curioso como o código de `ssLex` se assemelha à [versão iterativa da varredura e-r-d de uma árvore binária](#).

## Exercícios 3

- Que acontece se `ssLex` for executada com parâmetro  $n$  igual a 0? Verifique, cuidadosamente, a correção do código da função nos casos em que  $n$  vale 1 e 2.
- No código de `ssLex`, a variável  $k$  pode crescer mas nunca é comparada com  $n$ . Isso não é um erro?
- Analise a seguinte versão alternativa de `ssLex`.

```
subs[0] = 0; subs[k=1] = 1;
while (k >= 1) {
 imprime (subs, k);
 if (subs[k] < n) {
 subs[k+1] = subs[k] + 1; ++k;
 } else {
 subs[k-1] += 1; --k; } }
```

4. Analise a seguinte versão alternativa de ssLex:

```
subs[k=1] = 1;
imprime (subs, 1);
while (subs[1] < n) {
 if (subs[k] < n) {
 subs[k+1] = subs[k] + 1; ++k; }
 else {
 subs[k-1] += 1; --k; }
imprime (subs, k); }
```

5. ★ Escreva uma versão da função ssLex que trate o vetor  $\text{subs}[0..k]$  explicitamente como uma [pilha](#). Use as [funções criapilha, empilha, desempilha, pilhavazia e liberapilha](#) para manipular a pilha. Suponha que existe uma função `tamanhodapilha` que devolve o número de elementos da pilha e uma função `imprimepilha` que imprime o conteúdo da pilha a partir do seu segundo elemento.

6. Modifique a função ssLex para que ela produza todas as subsequências não vazias de  $0..n-1$ .
7. SUBSET SUM. Suponha que você emitiu cheques nos valores  $v[1], \dots, v[n]$  ao longo de um mês. No fim do mês, o banco informa que um total  $t$  foi debitado de sua conta. Quais dos cheques foram descontados? Por exemplo, se  $v$  é  $[61, 62, 63, 64]$  e  $t = 125$  então só há duas possibilidades: ou foram debitados os cheques 1 e 4 ou foram debitados os cheques 2 e 3. Essa é uma instância do seguinte *problema da soma de subconjunto*, mais conhecido como *subset sum*: Dado um número  $t$  e um vetor  $v[1..n]$  de números (não necessariamente todos positivos), encontrar todas as subsequências  $s[1..k]$  de  $1..n$  tais que  $v[s[1]] + \dots + v[s[k]] = t$ . Escreva uma função que resolva o problema.
8. TWO SUM. Escreva um algoritmo eficiente para o seguinte problema: dados um vetor  $v[1..n]$  de números inteiros e um número inteiro  $t$ , decidir se existem dois índices distintos  $i$  e  $j$  tais que  $v[i] + v[j] = t$ . [O problema é superficialmente semelhante ao [anterior](#), mas admite um algoritmo muito eficiente.] Parte 2: Suponha que os elementos do vetor são diferentes entre si e calcule *todos* os pares  $i, j$  de índices distintos tais que  $v[i] + v[j] = t$ .

## Subsequências: versão recursiva

A versão recursiva de ssLex é muito interessante. A interface com o usuário fica a cargo da seguinte função-embalagem (= *wrapper-function*):

```
// Recebe n >= 1 e imprime todas as
// subsequências não vazias de 1..n,
// em ordem lexicográfica.

void
ssLexR (int n)
{
 int *subs;
 subs = malloc ((n+1) * sizeof (int));
 ssLexComPrefixo (subs, 0, 1, n);
 free (subs);
}
```

O serviço pesado é todo executado pela função recursiva `ssLexComPrefixo`, cujo terceiro parâmetro é um índice  $m$  entre 1 e  $n+1$ :

```
static void
ssLexComPrefixo (int subs[], int k, int m, int n)
{
 if (m <= n) {
 // caso 1: inclui m
 subs[k+1] = m;
 imprime (subs, k+1);
 ssLexComPrefixo (subs, k+1, m+1, n);
 // caso 2: não inclui m
 ssLexComPrefixo (subs, k, m+1, n);
 }
}
```

O que faz, exatamente, a função `ssLexComPrefixo`? Como explicar o funcionamento de `ssLexComPrefixo`? Eis a resposta:

```
// A função ssLexComPrefixo recebe m <= n+1
// e uma subsequência subs[1..k] de 1..m-1
// e imprime, em ordem lexicográfica,
// todas as subsequências não vazias de m..n
// cada uma precedida do prefixo subs[1..k].
```

Em outras palavras, imprime todas as sequências da forma  $x[1..k..kk]$  tais que  $x[1..k] = \text{subs}[1..k]$  e  $x[k+1..kk]$  é uma subsequência não vazia de  $m..n$ . (É tentador começar o código da função com “`if (m > n) imprime (subs, k);`” mas isso não produziria ordem lexicográfica. Veja a ordem lexicográfica especial abaixo.)

Suponha, por exemplo, que  $\text{subs}[1] = 2$ ,  $\text{subs}[2] = 4$ , e  $n = 9$ . Então a chamada `ssLexComPrefixo (subs, 2, 7, n)` imprime a lista

```
2 4 7
2 4 7 8
2 4 7 8 9
2 4 7 9
2 4 8
2 4 8 9
2 4 9
```

A primeira linha é produzida por `imprime (subs, 3)`; as três linhas seguintes são produzidas por `ssLexComPrefixo (subs, 3, 8, n)`; e as demais, por `ssLexComPrefixo (subs, 2, 8, n)`.

Portanto, a chamada `ssLexComPrefixo (subs, 0, 1, n)` no código de `ssLexR` faz exatamente o que queremos: imprime todas as subsequências não vazias de  $1..n$  (com prefixo vazio).

A função `ssLexR` tem uma séria desvantagem em relação à sua versão iterativa: a pilha de execução da versão recursiva consome bem mais espaço na memória que a versão iterativa (que só precisa de espaço para armazenar `subs[0..n+1]`).

## Exercícios 4

1. IMPORTANTE. Verifique, cuidadosamente, a correção do código de `ssLexR` nos casos em que  $n$  vale 0, 1 e 2.
2. SUBSEQUÊNCIAS DE CADEIAS DE CARACTERES. Escreva uma função que imprima todas as subsequências não vazias de uma cadeia de caracteres ASCII dada. Por exemplo, as subsequências não vazias de ABC são
  - A
  - B
  - C
  - AB
  - AC
  - BC
  - ABC
3. O *vetor característico* de uma subsequência  $\text{subs}[1..k]$  de  $1..n$  é a sequência  $b[1..n]$  de bits que indica quais dos elementos  $1..n$  estão em  $\text{subs}[1..k]$  (ou seja,  $b[i]$  vale 1 se  $i$  está na subsequência e vale 0 em caso contrário). Para  $n = 5$ , por exemplo, o vetor característico de 2 3 5 é 01101. Escreva uma função que calcule o vetor característico de uma subsequência  $\text{subs}[1..k]$  de  $1..n$ . Escreva uma função que imprima todas as subsequências de  $1..n$  da seguinte maneira: gera todas as sequências de bits  $b[1..n]$  em ordem lexicográfica e, para cada sequência de bits, imprime a correspondente subsequência  $\text{subs}[1..k]$  de  $1..n$ . (Sugestão: imagine que um vetor de bits representa um número em notação binária e gere os vetores de bits em ordem numérica crescente.) Sua função imprime as subsequências em ordem lexicográfica?
4. Verifique que a ordem lexicográfica no conjunto de todas as subsequências de  $1..n$  é uma *ordem total*, ou seja, para quaisquer subsequências  $r$  e  $s$  tem-se
  1.  $r \leq s$  ou  $s \leq r$ ,
  2. se  $r \leq s$  e  $s \leq r$  então  $r = s$ ,
  3. se  $r \leq s$  e  $s \leq t$  então  $r \leq t$ ,onde “ $\leq$ ” denota a relação “igual ou lexicograficamente menor”.
5. ★ ORDEM LEXICOGRÁFICA ESPECIAL. A ordem lexicográfica *especial* dá preferência às subsequências

mais longas: uma sequência  $r[1..j]$  é menor que outra  $s[1..k]$  na ordem lexicográfica especial se (1) existe  $i$  tal que  $r[1..i-1]$  é igual  $s[1..i-1]$  e  $r[i] < s[i]$  ou (2)  $j > k$  e  $r[1..k]$  é igual  $s[1..k]$ . Veja, por exemplo, a enumeração de  $1..3$  em ordem lexicográfica especial:

```
1 2 3
1 2
1 3
1
2 3
2
3
```

Escreva uma função que enumere as subsequências de  $1..n$  em ordem lexicográfica especial. Faça uma versão iterativa e outra recursiva. Os algoritmos são um pouco mais simples que os da ordem lexicográfica usual. [[Solução](#).]

6. ORDEM MILITAR. As subsequências de  $1..n$  podem ser colocadas em *ordem militar*. A figura ilustra a ordem no caso em que  $n$  vale 4. Analise essa ordem. Escreva funções iterativa e recursiva que gerem todas as subsequências de  $1..n$  em ordem militar.

```
1
2
3
4
1 2
1 3
1 4
2 3
2 4
3 4
1 2 3
1 2 4
1 3 4
2 3 4
1 2 3 4
```

7. BIPARTIÇÃO EQUILIBRADA. Dois irmãos receberam uma herança que consiste em  $n$  objetos, cada objeto  $i$  tendo um valor  $v[i]$ . É possível dividir a herança  $v[1..n]$  em duas partes de mesmo valor? Essa é uma instância do problema da bipartição equilibrada: Dado um vetor  $v[1..n]$  de números (não necessariamente inteiros positivos), encontrar uma subsequência  $s[1..k]$  de  $1..n$  tal que  $v[s[1]] + \dots + v[s[k]] = v[t[1]] + \dots + v[t[m]]$ , sendo  $t[1..m]$  o complemento de  $s[1..k]$  em  $1..n$ . Escreva uma função que resolva o problema.
8. COMBINAÇÕES. Escreva uma função que imprima todas as subsequências de  $1..n$  que têm exatamente  $k$  elementos. (Isso corresponde, exatamente, aos subconjuntos de  $\{1, 2, \dots, n\}$  que têm exatamente  $k$  elementos.)

## Enumeração de permutações

Uma [permutação](#) da sequência  $1..n$  é qualquer rearranjo dos termos dessa sequência. Em outras palavras, uma permutação de  $1..n$  é qualquer sequência  $p[1..n]$  em que cada elemento de  $1..n$  aparece uma e uma só vez. É fácil verificar que há exatamente  $n!$  permutações de  $1..n$ . Por exemplo, as 24 permutações de  $1 2 3 4$  são

```
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 2 3
1 4 3 2
2 1 3 4
2 1 4 3
2 3 1 4
2 3 4 1
2 4 1 3
2 4 3 1
3 1 2 4
3 1 4 2
```

```

3 2 1 4
3 2 4 1
3 4 1 2
3 4 2 1
4 1 2 3
4 1 3 2
4 2 1 3
4 2 3 1
4 3 1 2
4 3 2 1

```

(Aqui, as permutações estão em ordem lexicográfica. Em particular, a primeira permutação é crescente e a última é decrescente.)

Nosso problema: Enumerar as permutações de  $1 \dots n$ , ou seja, produzir uma lista em que cada permutação de  $1 \dots n$  apareça uma e uma só vez.

A ordem em que as permutações são enumeradas não é muito importante, mas é natural dar preferência à [ordem lexicográfica](#).

**Preliminares.** A ideia é gerar cada permutação a partir de sua  [predecessora imediata](#). Considere, por exemplo as permutações de  $1 \dots 4$ . Para obter a  [sucessora imediata](#) da permutação  $3 4 1 2$ , podemos apagar o último termo, somar 1 ao penúltimo, e adotar o único valor possível para o último termo:

```

3 4 1 2
3 4 2 -
3 4 2 1

```

Em geral, pode ser necessário repetir esse processo. Por exemplo, para obter a sucessora imediata de  $2 1 4 2$ , não basta apagar o último termo, pois o valor seguinte do penúltimo termo não pertence ao universo  $1 \dots 4$ . Considere então apagar os *dois* últimos termos e somar 1 ao antepenúltimo. Isso ainda não resolve o impasse, pois já temos um 2 entre os primeiros termos. Mas somar mais 1 ao antepenúltimo termo torna-o diferente dos anteriores e assim resolve o impasse. Por fim, basta preencher as posições que ficaram vagas com os valores que ainda faltam, 1 e 4, nessa ordem:

```

2 1 4 3
2 1 5 - não, pois 5 não pertence ao universo
2 2 - - não, porque já temos 2 à esquerda
2 3 - - sim! todos diferentes!
2 3 1 4 completar em ordem crescente

```

Para dar uma descrição geral do processo, diremos que uma *subpermutação* de  $1 \dots n$  é qualquer permutação de uma *subsequência* de  $1 \dots n$ . Em outras palavras, uma subpermutação de  $1 \dots n$  é qualquer vetor  $\text{perm}[1 \dots k]$  cujos elementos pertencem a  $1 \dots n$  e são diferentes entre si. Agora, o algoritmo pode ser organizado de modo que cada iteração comece com

- uma subpermutação  $\text{perm}[1 \dots k]$  e
- um candidato  $m$  para a posição  $\text{perm}[k+1]$ .

Cada iteração incorpora  $m$  (aumentando  $k$ ) ou desiste de  $m$ , reduz  $k$ , e escolhe um novo  $m$ . (Esse movimento de volta, que desfaz o último elemento de  $\text{perm}[1 \dots k]$  e prepara o valor seguinte de  $m$ , é conhecido como *backtrack*.) Ao cabo de algumas iterações, teremos a sucessora imediata da permutação inicial ou constataremos que não existe sucessora, pois a permutação inicial já é a última.

Segue um esboço do código que realiza a tarefa. Se  $k == n$ , já temos uma permutação completa e portanto o único movimento possível é um backtrack:

```

m = perm[k] + 1;
--k;

```

(Esse processo de fazer-e-desfazer lembra minha avó tricotando um sueter de lã por tentativa-

e-erro!) Se  $k < n$ , é preciso verificar se  $m$  já não está em  $\text{perm}[1..k]$ :

```
while (m <= n && !pode (perm, k, m))
 m++;
if (m <= n) { // pode
 perm[k+1] = m;
 m = 1;
 ++k;
} else {
 m = perm[k] + 1;
 --k;
}
```

A função [booleana](#) pode com argumentos ( $\text{perm}$ ,  $k$ ,  $m$ ) decide se  $m$  é diferente dos elementos de  $\text{perm}[1..k]$ .

**Algoritmo.** O esboço que acabamos de fazer leva à seguinte função de enumeração de permutações:

```
// Imprime, em ordem lexicográfica,
// todas as permutações de 1..n.
//
void
permutacoes (int n)
{
 int *perm = malloc ((n+1) * sizeof (int));
 int k;
 for (k = 1; k <= n; k++) perm[k] = k;
 k = n;

 while (true) {
 int m;
 if (k >= n) {
 imprime (perm, n);
 m = n+1; // para forçar backtracking
 }
 while (m <= n && !pode (perm, k, m))
 m++;
 if (m <= n) { // pode
 perm[k+1] = m;
 m = 1;
 ++k;
 } else { // backtrack
 if (k < 1) break;
 m = perm[k] + 1;
 --k;
 }
 }
 free (perm);
}

// Recebe subpermutação $\text{perm}[1..k]$ e decide
// se $\text{perm}[1..k]m$ é uma subpermutação.
//
static bool pode (int *perm, int k, int m) {
 for (int i = 1; i <= k; ++i)
 if (m == perm[i])
 return false;
 return true;
}
```

(Eu reconheço que esse código é deselegante e difícil de entender. Tente escrever um código melhor!) Não é necessário começar a primeira iteração com a permutação  $1..n$ . Podemos começar com a subpermutação vazia: basta trocar “`for (k = 1; k <= n; k++) perm[k] = k; k = n;`” por “`k = 0; m = 1;`”.

Para entender o algoritmo, é preciso descobrir seus invariantes. Não convém formular os

invariantes no início do processo iterativo mas sim imediatamente depois que uma permutação é impressa, ou seja, depois do bloco “`if (k >= n) { ... }`”. A cada passagem por esse ponto, temos um índice  $k$  que pertence a  $0..n$ , um número  $m$  que pertence a  $1..n+1$ , e uma subpermutação  $\text{perm}[1..k]$  de  $1..n$  com a seguinte propriedade:

todas as permutações lexicograficamente menores que  $\text{perm}[1..k]m$  já foram impressas.

(A expressão  $\text{perm}[1..k]m$  representa a sequência  $x[1..k+1]$  tal que  $x[1..k]$  é igual a  $\text{perm}[1..k]$  e  $x[k+1] == m$ . Essa sequência não é, necessariamente, uma subpermutação.)

A função `permutacoes` só pode ser usada para valores muito modestos de  $n$ , uma vez que o número de permutações cresce assustadoramente à medida que  $n$  aumenta.

## Exercícios 5

1. Prove que há exatamente  $n!$  permutações de  $1..n$ .
2. ★ IMPORTANTE. Verifique, cuidadosamente, a correção do código de `permutacoes` nos casos em que  $n$  vale 0, 1 e 2.
3. Escreva uma função que receba uma permutação  $\text{perm}[1..n]$  de  $1..n$  e imprima sua [sucessora imediata](#).
4. VERSÃO RECURSIVA. Analise a seguinte versão da função `permutacoes`, que enumera as permutações  $1..n$  em ordem lexicográfica. O que faz, exatamente, a função recursiva `permsComPrefixo`?

```
void permutacoes (int n) {
 int *perm;
 perm = malloc ((n+1) * sizeof (int));
 permsComPrefixo (perm, 0, n);
 free (perm);
}
static void permsComPrefixo (int* sp, int i, int n) {
 if (i >= n) imprime (sp, n);
 else {
 for (int m = 1; m <= n; m++) {
 if (pode (sp, i, m)) {
 sp[i+1] = m;
 comprefixo (sp, i+1, n); } } }
}
static bool pode (int *sp, int i, int m) {
 for (int k = 1; k <= i; k++)
 if (m == sp[k]) return false;
 return true;
}
```

5. UM ALGORITMO CLÁSSICO. Analise o seguinte algoritmo clássico de enumeração de permutações de  $1..n$ . O que faz, exatamente, a função recursiva `permsComPrefixo`? As permutações são impressas em ordem lexicográfica?

```
void permutacoes (int n) {
 int *perm;
 perm = malloc ((n+1) * sizeof (int));
 for (int i = 1; i <= n; i++) perm[i] = i;
 permsComPrefixo (perm, 0, n);
 free (perm);
}
static void permsComPrefixo (int *perm, int i, int n) {
 if (i >= n-1)
 imprime (perm, n);
 else {
 for (int j = i+1; j <= n; j++) {
 troca (perm, i+1, j);
 permsComPrefixo (perm, i+1, n);
 troca (perm, i+1, j); } }
}
static void troca (int *perm, int k, int j) {
 int t = perm[k]; perm[k] = perm[j]; perm[j] = t;
}
```

6. PERMUTAÇÃO DE CADEIAS DE CARACTERES. Escreva uma função que imprima todas as permutações dos bytes de uma [cadeia de caracteres ASCII](#) dada. Por exemplo, ao receber a cadeia ABC, a função

deve imprimir

ABC ACB BAC BCA CBA CAB

7. PROBLEMA DAS RAINHAS. É possível colocar 8 rainhas do jogo de xadrez sobre o tabuleiro de modo que nenhuma das rainhas possa atacar outra? Escreva uma função que enumere todas as maneiras de dispor  $n$  rainhas num tabuleiro generalizado  $n \times n$ . Uma disposição das rainhas pode ser representada por um vetor  $perm[1 \dots n]$  tal que a rainha da linha  $i$  fica na coluna  $perm[i]$ . (Sugestão: Adapte o código da função [permutacoes](#).)

8. PASSEIO DO CAVALO. Suponha dado um tabuleiro de xadrez  $n$ -por- $n$ . Determine se é possível que um cavalo do jogo de xadrez parta da posição  $(1,1)$  do tabuleiro e complete um passeio por todas as  $n \times n$  posições do tabuleiro em  $n \times n - 1$  passos válidos. Por exemplo, para um tabuleiro 5-por-5 uma solução do problema é

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 6  | 15 | 10 | 21 |
| 14 | 9  | 20 | 5  | 16 |
| 19 | 2  | 7  | 22 | 11 |
| 8  | 13 | 24 | 17 | 4  |
| 25 | 18 | 3  | 12 | 23 |

(Sugestão: Numere as casas do tabuleiro da maneira óbvia: a primeira linha da esquerda para a direita, depois a segunda linha, etc. Agora examine todas as permutações de  $1 \ 2 \ \dots \ n^2$ . Para cada permutação, verifique se ela representa um passeio válido.)

9. PERMUTAÇÃO CIRCULAR. Uma sequência  $s[1 \dots n]$  é *permutação circular* de outra  $t[1 \dots n]$  se existe  $k$  tal que  $s[k \dots n]$  é igual a  $t[1 \dots n-k+1]$  e  $s[1 \dots k-1]$  é igual a  $t[n-k+2 \dots n]$ . Escreva uma função que decida se  $s[1 \dots n]$  é permutação circular de  $t[1 \dots n]$ .
10. PERMUTAÇÃO ALEATÓRIA. Escreva uma função que produza uma permutação [aleatória](#) de  $1 \dots n$ . A função deve produzir, com igual probabilidade, qualquer uma das  $n!$  permutações de  $1 \dots n$ .
11. DISTÂNCIA TAU DE KENDALL. Como medir a distância entre duas permutações? Veja [exercício no capítulo do algoritmo Mergesort](#).
12. DESARRANJOS. Um *desarranjo* (= *derangement*) da sequência  $1 \dots n$  é qualquer permutação dessa sequência que muda *todos* os termos de posição. Em outras palavras, um desarranjo de  $1 \dots n$  é qualquer permutação  $p[1 \dots n]$  de  $1 \dots n$  tal que  $p[i]$  é diferente de  $i$  para todo  $i$ . Por exemplo, os 9 desarranjos de  $1 \ 2 \ 3 \ 4$  são  $2 \ 1 \ 4 \ 3$ ,  $2 \ 3 \ 4 \ 1$ ,  $2 \ 4 \ 1 \ 3$ ,  $3 \ 1 \ 4 \ 2$ ,  $3 \ 4 \ 1 \ 2$ ,  $3 \ 4 \ 2 \ 1$ ,  $4 \ 1 \ 2 \ 3$ ,  $4 \ 3 \ 1 \ 2$ ,  $4 \ 3 \ 2 \ 1$ . Escreva uma função que imprima, exatamente uma vez, cada desarranjo de  $1 \dots n$ .
13. PARTIÇÕES. Escreva uma função que imprima uma lista de todas as [partições](#) do conjunto  $\{1, 2, \dots, n\}$  em  $m$  blocos não vazios. Você pode representar uma tal partição por um vetor  $w[1 \dots n]$  com valores em  $1 \dots m$  dotado da seguinte propriedade: para cada  $i$  em  $1 \dots m$  existe  $j$  tal que  $w[j] = i$ .

---

Atualizado em 2018-08-19

<https://www.ime.usp.br/~pf/algoritmos/>

Paulo Feofiloff

[DCC-IME-USP](#)

