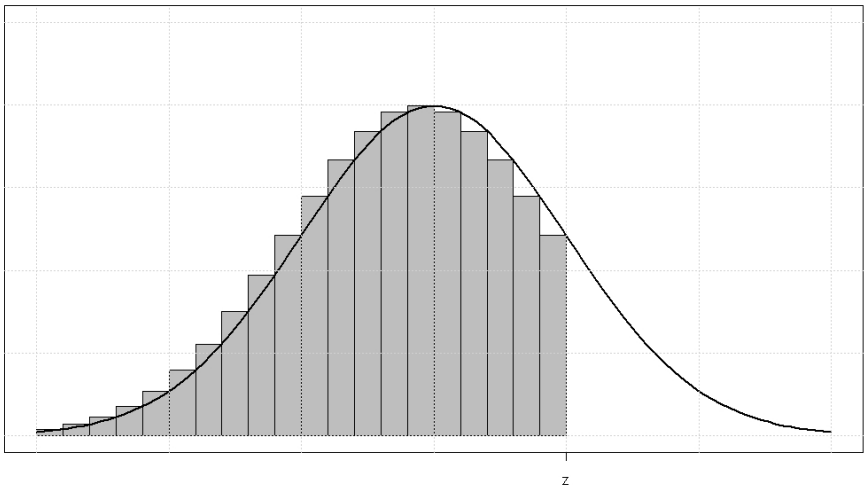
Apostila de Programação Estatística



Jessica Kubrusly

Departamento de Estatística
Instituto de Matemática e Estatística
Universidade Federal Fluminenes

Agosto 2016

Prefácio

Esta apostila foi desenvolvia com o objetivo de se tornar um roteiro de aula para a disciplina Programação Estatística, oferecida aos alunos do curso de Graduação em Estatística pelo Departamento de Estatística da Universidade Federal Fluminense - UFF.

A disciplina de Programação Estatística é constituída por uma aula teórica e duas práticas por semana. Cada capítulo desta apostila aborda o conteúdo trabalhado em cada semana. Além disso a apostila é separada em 3 partes. Ao final de cada parte os alunos são submetidos a uma prova teórica e uma prova prática. O foco da disciplina é praticar técnicas de programação usando a linguagem de programação R e tendo como plano de fundo conceitos de estatística descritiva, álgebra linear básica e cálculo.

O aluno matriculado nessa disciplina já foi aprovado em uma disciplina introdutória sobre programação de computadores e por isso este não é o seu primeiro contato com o assunto. Ao final desta disciplina espera-se que o aluno tenha não só aprimorado sua habilidade computacional como também reforçado conceitos de estatística e cálculo já vistos em outras disciplinas.

Para maiores informações sobre a linguagem de programação R entre no site oficial do projeto: `http://www.r-project.org/`. Lá é possível obter informações sobre como baixar e instalar o R em seu computador. Além disso, manuais introdutórios são disponibilizados gratuitamente.

Sumário

Parte I

Conceitos Básicos de Programação

Semana 1: Objetos e Classes

Toda variável dentro da linguagem R é interpretada como um objeto que pertence a uma determinada classe. Para saber qual a classe de um certo objeto podemos usar o comando `class(obj)`, onde `obj` é o nome do objeto para o qual queremos saber a classe.

Existem diversos tipos de classes. Algumas serão vistas ao longo deste curso e outras apenas em disciplinas futuras. Por exemplo, podemos citar alguns tipos básicos que serão vistos na aula de hoje: `"numeric"`, `"logical"`, `"character"`, `"matrix"` e `"list"`.

1.1 Números

Os objetos numéricos fazem parte da classe `"numeric"` e tratam-se dos números rais. Para criar um objeto desse tipo podemos usar, por exemplo, o comando `x<-3.4`. Para verificar a sua classe basta digitar `class(x)` no prompt do R. A resposta será `"numeric"`.

Para os objetos desse tipo já estão definidos alguns operadores (`+, -, *, /`) além de função conhecidas, como por exemplo `sqrt()`, `log()`, `exp()`, `abs()`, entre outras. Para saber mais detalhes sobre essas funções ou sobre a classe `"numeric"` use o *help* do R através do comando `help(log)` ou `help(numeric)`, por exemplo.

1.2 Textos

Os objetos do tipo textos fazem parte da classe `"character"` e é dessa forma que o R guarda letras ou frases. Dentro do R os textos sempre vão aparecer entre aspas (""). Para criar um objeto desse tipo podemos usar, por exemplo, os comandos `ch1<-"a"`, `ch2<-"abc"` e `ch3<-"1"`. Os três objetos criados são da classe `"character"`, para checar isso basta digitar `class(ch1)`, `class(ch2)` e `class(ch3)`. Veja que `class(ch3)` não é um número e sim um texto, ou seja, é o número 1 armazenado como texto.

Um comando bastante útil para a classe `"character"` é o `paste()`. Ele recebe como argumento objetos do tipo `"character"` e retorna um único objeto do tipo `"character"`, que é definido pela concatenação dos objetos passados na entrada. Quando chamamos o comando `paste()` podemos indicar como os objetos serão separados usando a opção `sep`. Se o campo `sep` não for definido ele será considerado como espaço.

Aqui estão alguns exemplos de utilização do comando `paste()`. Se digitarmos `ch4 <- paste(ch1,ch2,"a",sep=".")`, `ch4` é um objeto do tipo `"character"` definido por `"a.abc.a"`. Se mudarmos o valor de `sep` a resposta muda complentamente: `ch5 <- paste(ch1,ch2,"a",sep="")`, `ch5` é um objeto do tipo `"character"` definido por `"aabca"`. Se não definirmos o `sep` os textos serão concatenados por espaços, como já foi coment-dao: `ch6 <- paste(ch1,ch2,"a")`, `ch6` é um objeto do tipo `"character"` definido por `"a abc a"`.

Para saber mais sobre a classe `"character"` consulte o *help* do R através do comando `help(character)`.

1.3 Lógicos

Os objetos do tipo lógico fazem parte da classe `"logical"` e podem ser de dois tipos: `TRUE` ou `FALSE`. Eles também podem ser simplesmente representados pelas letras `T` ou `F`. Para criar um objeto desse tipo podemos usar, por exemplo, o comando `v<-TRUE` ou `v<-T`. Para checar a sua classe basta digitar `class(v)` no prompt do R. A resposta será `"logical"`.

Dentro do R já estão definidos os operadores lógicos E ($\cap$), representado pelo comando `&&`, e OU ($\cup$), representado pelo comando `||`. A tabela verdade a seguir indica a resposta para cada um desses dois operadores.

A	B	A&&B	A||B
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE

Também podemos realizar a negação usando o comando `!`. Por exemplo, se `a <- TRUE`, `a!` retorna `FALSE`; se `a <- FALSE`, `a!` retorna `TRUE`.

Se quisermos testar se dois objetos são iguais (ou diferentes), podemos usar os comandos `==` ou `!=` e a resposta será um objeto da classe `"logic"`. Por exemplo, se digitarmos `a<-1; b<-1; c<-2`, podemos testar se os objetos são iguais ou não. Digitando `a==a`, `a==b` ou `a!=c` termos `TRUE` como resposta. Digitando `a==c`, `a!=b` ou `b==c` termos `FALSE` como resposta.

Para saber mais sobre a classe `"logical"` consulte o *help* do R através do comando `help(logical)`.

1.4 Array

Dentro da linguagem R um array é uma sequencia de objetos do mesmo tipo. Por exemplo, podemos ter array de números reais, isto é uma sequência de objetos do tipo `"numeric"`, array de textos, isto é uma sequência de objetos do tipo `"character"`, array de "verdadeiro" ou "falso", isto é uma sequência de objetos do tipo `"logic"`, ou array de qualquer outra classe dentro do R.

Existem várias maneiras de criarmos um array, a mais simples delas é usando o comando `c()`, por exemplo, `a<-c(1,2,3)`, `b<-c("a", "aa")` ou `c<-c(T,T,F,F)`. Nesse caso `a` é um array de números, `b` um array de texto e `c` um array de "verdadeiro" ou "falso".

No R array não é uma classe, como os demais itens citados acima. Se você usar o comando `class()` nos objetos `a` , `b` e `c` as respostas serão: `"numeric"`, `"character"` e `"logic"`, respectivamente. Isso ocorre pois o R trata qualquer objeto como um array. No caso de ser um objeto simples, e não uma sequência de objetos, ele interpreta como se fosse um array de tamanho 1, já as sequências de objetos são arrays de tamanhos maiores que 1.

Um comando muito importante que pode ser usado em arrays é o `length()`, que retorna o tamanho do array. Por exemplo, se digitarmos `length(a)` a resposta será o número 3. Ou seja, se digitarmos `n<-length(a)`, `n` é um objeto do tipo `"numeric"` que guarda o número 3.

Para acessarmos uma posição de um array usamos o comando `[ ]`. Por exemplo, para o array `a` definido acima, `a[1]` retorna a primeira posição do array, que é o número 1, `a[2]` retorna a segunda posição do array, que é o número 2, e assim por diante. Se acessarmos uma posição que não tenha sido definida, como por exemplo `a[5]`, a resposta será `NA`, que representa o termo em inglês "Not Available".

Existem outras maneiras de criarmos arrays dentro do R. Uma delas é usando o próprio `[ ]` para alocarmos uma posição específica. Por exemplo, podemos trocar a primeira posição do array `a` usando o comando `a[1]<-12`. Depois disso `a` passa a ser a sequência com os números 12, 2 e 3. Podemos inclusive alocar posições vazias do array, se fizermos `a[4]<-0` o array `a` passa a ser uma array de tamanho 4 com os números 12, 2, 3 e 0. Podemos ainda fazer `a[6]<-6` e nesse caso o array `a` tem tamanho 6 e é definido pela sequência 12, 2, 3, 0, NA e 6.

Temos ainda diversas outras maneiras de criarmos arrays dentro do R. A tabela a seguir apresenta mais algumas:

Comandos	Descrição
`y <- c(1:10)`	`y` é um array de tamanho 10 com os números de 1 até 10. Ou seja, $y = (1, 2, 3, 45, 6, 7, 8, 9, 10)$.
`z <- seq(1,20,by=2)`	`z` é um array definido pela sequência começando em 1, terminando em 20 e crescendo de 2 em 2. Ou seja, $z = (1, 3, 5, 7, 9, 11, 13, 15, 17, 19)$.
`w <- rep(0, times=5)`	`w` é um array com 5 números 0. Ou seja, $w = (0, 0, 0, 0, 0)$
`v<-c(1,2); v<-c(v,4)`	`v` é um array de tamanho 3 com os números $1, 2, 4$. Nesse exemplo o comando `c()` foi usando para concatenar o vetor v inicialmente definido com o elemento 4.

Como pode ser vistos nos exemplos citados, dentro do R podemos acrescentar elementos dentro um array sem problema algum. Isso não necessariamente acontece com outras linguagens de programação. Nesse caso ainda tempos a opção de criarmos um array vazio, usando o comando `v <- NULL`, e acrescentar suas posições uma a uma: `v[1] <- 1`, `v[2] <- 5` ou `v <- c(v,11)`. Se soubermos qual o tipo de objeto que será alocado dentro do array podemos inicia-lo vazio com o seguinte comando: `v<-numeric()`, no caso de um array de objetos da classe `"numeric"`, `v<-character()`, no caso de um array de objetos da classe `"character"`, e assim por diante.

1.5 Matrizes

Dentro da linguagem R existe uma classe pré definida chamada `"matrix"` que guarda objetos do mesmo tipo em forma de matriz, ou seja, guarda os objetos por linhas e colunas.

Para criar um novo objeto do tipo `"matrix"` podemos usar o comando

`matrix(data=, nrow=, ncol=, byrow=, dimnames=)`.

Ao lado do sinal de = devemos colocar as informações da matriz que está sendo criada:

`data=`	array com os objetos que serão alocado dentro da matriz;
`ncol=`	número de colunas da matriz;
`nrow=`	número de linhas da matriz;
`byrow=`	`TRUE` ou `FALSE` que indica se os objetos serão alocados por linha ou por coluna;
`dimnames =`	lista[1] com os nomes para as linhas e colunas da matriz.

Se alguma das informações não for preenchida será considerado o valor padrão para cada entrada, que nesse caso é: `data=NA`, `ncol=1`, `nrow=1` e `dimnames = NULL`.

Vejamos alguns exemplos de como usar este comando e criar matrizes. Considere os seguintes comandos digitados no prompt do R:

```
> mdat1 <- matrix(c(1,2,3,11,12,13), nrow = 2, ncol=3, byrow=TRUE)
> mdat1
      [,1]  [,2]  [,3]
 [1,]    1     2     3
 [2,]   11    12    13
```

Veja que `mdat1` é um objeto da classe `"matrix"`. Para verificar basta digitar:

```
> class(mdat1)
[1] "matrix"
```

Veja que se mudarmos o número de linhas ou colunas a matriz criada é totalmente diferente.

```
> mdat2 <- matrix(c(1,2,3,11,12,13), nrow = 3, ncol=2, byrow=TRUE)
> mdat2
      [,1]  [,2]
 [1,]    1     2
 [2,]    3    11
 [3,]   12    13
```

Veja agora como ficaria a matriz se ela fosse alocada por colunas (`byrow=FALSE`).

```
> mdat3 <- matrix(c(1,2,3,11,12,13), nrow = 3, ncol=2, byrow=FALSE)
> mdat3
      [,1]  [,2]
 [1,]    1    11
 [2,]    2    12
 [3,]    3    13
```

Quando criamos um objeto da classe `"matrix"`, o número de linhas e de colunas tem que ser determinado logo de início e não pode ser alterado depois.

As posições de uma matriz são acessadas usando o comando `[ , ]`. Por exemplo, se digitarmos `mdat3[2,1]` a resposta sera 2, que é o elemento guardado na segunda linha e primeira coluna. Com esse mesmo comando podemos "pegar" linhas ou colunas inteiras de uma matriz: se digitarmos `mdat3[1,]` a resposta será um array com os elementos da linha 1; se digitarmos `mdat3[,2]` a resposta será um array com os elementos da coluna 2.

Se quisermos o número de linhas de uma matriz podemos usar o comando `nrow()`, se digitarmos `nrow(mdat3)` a resposta será 3. Para saber o número de colunas de uma matriz use, de forma análoga, o comando `ncol()`.

[1] A classe lista será vista na próxima seção.

Para saber mais sobre a classe `"matrix"` consulte o *help* do R através do comando `help(matrix)`.

1.6 Listas

Na linguagem R a classe que guarda os objetos em forma de lista é denominada `"list"`. Esse objeto guarda uma sequência de objetos e a sua principal diferença para os arrays é que a lista pode guardar objetos de tipos diferentes.

Para criarmos uma lista vamos usar o comando `list()`. Se simplesmente digitarmos `l1 <- list()`, a `l1` será uma lista vazia. Se dentro dos parênteses colocarmos uma sequencia de objetos, a lista criada guardará os objetos digitados. Por exemplo: `l2 <- list(1,"a",TRUE)`, `l2` é uma lista com 3 objetos; ou `l3 <- list(c(1,2,3),c(1,1,1,1), TRUE,c("A","a"))`, `l3` é uma lista com 4 objetos. Podemos até criar uma lista de listas, por exemplo `l4 <- list(l1,l2,l3)`.

Enquanto usamos `[ ]` para acessar uma posição de um array, para acessar as posições de uma lista usaremos o comando `[[ ]]`. Dessa forma, `l3[[2]]` deve retornar o array `c(1,1,1,1)`, que está alocado na segunda posição da lista `l3`.

Para saber quantos objetos estão guardados dentro de uma lista também podemos usar o comando `length()`. Dessa forma `length(l1)` deve retornar 0, `length(l2)` deve retornar 3, `length(l3)` deve retornar 4 e `length(l4)` deve retornar 3.

Pergunta: Se digitarmos `length(l3[[2]])`, qual a resposta retornada?

Da mesma forma que os arrays, novas posições de uma lista podem ser alocadas. No caso das listas usaremos o comando `[[ ]]` para isso. Além de alocar novas posições ele também serve para modificar posições já existentes. Então, se digitarmos `l2[[4]]<- c(1,2,3)` a lista `l2` passa a guardar na posição 4 um array de tamanho 3 com os números 1,2 e 3.

Para saber mais sobre a classe `"list"` consulte o *help* do R através do comando `help(list)`.

Exercícios - 1ª Semana

1.1 Defina `x`, `y` e `z` como sendo objetos do tipo `"numeric"` que guardam os valores 0, -1 e $\frac{3}{2}$, respectivamente. Faça no prompt do R as contas a seguir e verifique se o resultado está como o esperado.

(a) $a = x + y + z$, $b = yz$ e $c = \frac{z}{y}$;

(b) z^2, z^3, z^x e z^y;

(c) $\sqrt{a}$, $\sqrt{x}$ e $\sqrt{y}$;

(d) $\sqrt[3]{b}$, $\sqrt[4]{-\frac{1}{c}}$ e $\sqrt[3]{z^2}$;

(e) $|x|$, $|y|$ e $|z|$;

(f) $\exp(x)$, $\exp(y)$ e $\exp(c)$;

(g) $\ln(x)$, $\ln(a)$ e $\ln(b)$;

(h) $\sqrt{\pi}$ e e^{-x}.

1.2 Defina `ch1`, `ch2` e `ch3` como sendo objetos do tipo `"character"` que guardam os textos "a", "b" e "c", respectivamente.

(a) Usando a função `paste` a partir de `ch1`, `ch2` e `ch3` crie um quarto objeto da classe `"character"`, `ch4`, definido por "a.b.c".

(b) Usando a função `paste` a partir de `ch1`, `ch2` e `ch3` crie um quinto objeto da classe `"character"`, `ch5`, definido por "abc".

(c) Usando o comando `==` verifique se `ch4` e `ch5` são iguais ou diferentes.

(d) Usando o comando `!=` verifique se `ch4` e `ch5` são iguais ou diferentes.

1.3 O operador `%%` fornece o resto da divisão entre dois números, por exemplo, `15%%4` fornece o resto da divisão de 15 por 4, que é 3. Esse comando será bastante usado durante o curso. Faça os itens a seguir primeiros no papel e depois verifique a resposta usando o R.

(a) Qual a resposta para `18%%5`, `-5%%2`, `15%%5` e `8.3%%3`?

(b) Como podemos usar o operador `%%` para testar se um número é par? Faça o teste no prompt do R e use também os operadores `==` ou `!=` de forma que a resposta seja `TRUE` se o número for par e `FALSE` caso contrários.

(c) Como podemos usar o operador `%%` para testar se um número é inteiro? Faça o teste no prompt do R e use também os operadores `==` ou `!=` de forma que a resposta seja `TRUE` se o número for inteiro e `FALSE` caso contrários.

(d) Como podemos usar o operador `%%` para testar se um número é natural, isto é, inteiro e positivo? Faça o teste no prompt do R e use também os operadores `==`, `!=`, `&&` ou `||` de forma que a resposta seja `TRUE` se o número for natural e `FALSE` caso contrários.

1.4 Digite no prompt do R:

```
> a<-seq(1:10); b<-seq(1,20,by=2); c<-seq(20,1,by=-2)
```

Usando os operadores `+,-,*,/` e também `==,!=,<,>` faça o que se pede nos itens a seguir.

(a) Crie um array `x` onde cada posição de `x` é dada pela subtração entre as respectivas posições de `b` e `c`.

(b) Crie um array `y` onde cada posição de `y` é o dobro de cada posição de `a`.

(c) Crie um array `z` onde cada posição de `z` é um objeto da classe `"logic"`. A posição i de `z` vai guardar `TRUE` se `a[i]` for igual a `b[i]` e `FALSE` caso contrário.

(d) Crie um array `w` onde cada posição de `w` é um objeto da classe `"logic"`. A posição i de `w` vai guardar `TRUE` se `c[i]` for maior que `b[i]` e `FALSE` caso contrário.

1.5 No R já exitem alguns objetos pré-definidos que são chamados de constantes. Como exemplo temos a constante `pi`, já usada no exercício (1), e os arrays `letters` e `LETTERS`: sequências com as letras minúsculas e maiúsculas do alfabeto.

(a) Primeiro digite `letters` e `LETTERS` para como são exatamente esses objetos.

(b) Qual a classe dos objetos `letters` e `LETTERS`? Primeiro tente responder sem usar o comando `class` e depois verifique a sua resposta usando tal comando.

(c) Sem contar, como podemos encontrar o tamanho dos arrays `letters` e `LETTERS`? Qual o tamanho deles?

(d) Se digitarmos `a<-c(LETTERS,letters)` qual a classe do objeto `a`, qual o seu tamanho e como é este objeto. Tente responder sem digitar e depois use o prompt para verificar a sua resposta.

(e) Se digitarmos `b<-paste(LETTERS,letters)` qual a classe do objeto `b`, qual o seu tamanho e como é este objeto. Tente responder sem digitar e depois use o prompt para verificar a sua resposta.

1.6 Crie as seguintes matrizes no R:

(a)

```
     [,1] [,2]
[1,]    1  101
[2,]    2  102
[3,]    3  103
[4,]    4  104
```

(b)

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]  101  102  103  104
```

(c)

```
     [,1] [,2]
[1,]    0    0
[2,]    0    0
[3,]    0    0
```

(d)

```
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1
```

Faça os itens (c) e (d) sem usar um array cheio de 0's ou 1's.

1.7 Digite no prompt do R o seguinte comando:

```
> A<-matrix(c(1,2,3,4,5,6,7,8,9,10,11,12),4,3)
```

Sem contar, usando os comandos da classe `"matrix"` que fornece as dimensões de uma matriz, encontre o número de linhas e o número de colunas de `A`.

1.8 (a) Crie uma lista chamada `minha_lista` com 4 elementos. O primeiro elemento é o seu **nome**, o segundo sua **idade**, o terceiro um array que guarda suas **medidas** (altura e peso, nessa ordem) e o quarto elemento é outro array que guarda `TRUE` para as respostas afirmativas e `FALSE` para as respostas negativas das seguintes

perguntas: (i) Você já estagiou? ; (ii) Você já participou de algum projeto como voluntário? ; (iii) Você tem interesse em assuntos relacionados ao meio ambiente?.

(b) A partir do objeto `minha_lista` criado acesse o seu nome.

(c) A partir do objeto `minha_lista` criado acesse a sua idade.

(d) A partir do objeto `minha_lista` criado acesse a sua altura.

(e) A partir do objeto `minha_lista` criado acesse o seu peso.

(f) A partir do objeto `minha_lista` criado acesse a resposta para a pergunta "Você tem interesse em assuntos relacionados ao meio ambiente?".

1.9 (a) Refaça o item (a) do exercícios anterior agora com os dados de um amigo ou dados fictícios. Chame essa nova lista de `lista_2`.

(b) Crie agora outra lista com 2 objetos, vamos chamá-la de `dados_alunos`. O primeiro objeto dessa lista é a lista criada no exercício anterior, `minha_lista`, e o segundo objeto é a lista criada no primeiro item desse exercício, `lista_2`.

(c) A partir do objeto `dados_alunos` criado acesse o seu nome.

(d) A partir do objeto `dados_alunos` criado acesse o nome do seu amigo.

(e) A partir do objeto `dados_alunos` criado acesse a sua altura.

(f) A partir do objeto `dados_alunos` criado acesse a resposta do seu amigo para a pergunta "Você já estagiou?".

1.10 Qual a diferença entre os objeto `obj1`, `obj2` e `obj3` definidos a seguir?

```
> obj1 <- list(1,2,3)
> obj2 <- list(c(1,2,3))
> obj3 <- c(1,2,3)
```

Semana 2: Controle de Fluxo

Os controles de fluxo são operações definidas em todas as linguagens de programação, como por exemplo C, C++, Java, Fortran, Pascal, etc. Como não podia deixar de ser, tais operações também estão definidas dentro da linguagem R.

Cada linguagem de programação tem a sua própria sintaxe, isto é, sua própria regra de como essas operações devem ser usadas. Veremos nessa aula a sintaxe e mais detalhes sobre os controles de fluxo para a linguagem R.

2.1 if/else

Sintaxe:

```
if( afirmação ) {
        tarefas 1
} else {
        tarefas 2
}
```

Descrição:

Realiza as tarefas 1 se a afirmação for verdadeira e realiza as tarefas 2 se a afirmação for falsa. O comando else é opicional. A afirmação tem que ser um objeto da classe `"logical"`.

Vejamos alguns exemplos de como usar o controle de fluxo `if/else`. Primeiro vamos escrever um código que imprimi um texto que varia de acordo com o valor guardado em uma variável `x`. Se `x` recebe o valor 2, veja qual texto será impresso.

```
> x <- 2
> if( x < 5 ){
+     print(paste(x,"e menor que",5))
+ } else {
+     print(paste(x,"e maior ou igual a",5))
+ }
[1] "2 e menor que 5"
```

O comando `print` é responsável por imprimir na tela e o comando `paste` por concatenar textos e criar um único objeto do tipo `character`. Para mais detalhes digite no prompt do R `help(print)` e `help(paste)`.

Se a variável `x` receber um valor maior que 5 o texto impresso é outro.

```
> x <- 7
> if( x < 5 ){
+     print(paste(x,"e menor que",5))
+ } else {
+     print(paste(x,"e maior ou igual a",5))
+ }
[1] "7 e maior ou igual a 5"
```

Considere agora a seguinte sequência de comandos:

```
> x <- 3
> if (x > 2){
+     y <- 2*x
+ } else {
+     y <- 3*x
+ }
```

Qual o valor da variável `y` ao final desse código? Respondeu certo quem disse 6. O controle de fluxo `if/else` será usado na maioria das vezes dentro de funções, como veremos na próxima semana.

2.2 for

Sintaxe:

```
for(i in a:b){
        tarefas
}
```

Descrição:

`i` = nome da variável do *loop* ou laço.
`a` e `b` = valores inteiros.
No início do *loop* a variável `i` recebe o valor `a` e é incrementada (ou decrementada, se `a`>`b`) até chegar no valor `b`, que é o valor final da variável `i`.
As tarefas descritas dentro das chaves são executadas para cada valor de `i` entre `a` e `b`.

Veja a seguir um exemplo de como usar o `for`.

```
> y<-0
> for(i in 1:10){
+     y<-y+1
+ }
```

Ao final do comando qual o valor guardado na variável `y`? Antes de responder vamos tentar entender cada passo.

Veja que `y` começa com o valor 0. Quando `i`=1, `y` é incrementado de 1 unidade e passa a guardar o valor 0+1=1. Quando `i`=2, `y` é incrementado de mais 1 unidade e passa a guardar o valor 1+1=2. Assim por diante até que temos `i`=10, quando `y` recebe seu último incremente e passa a guardar o valor 10.

Veja outro exemplo.

```
> x<-3
> for(var in 2:5){
+     x<-x+var
+ }
```

E agora, ao final do comando qual o valor guardado na variável `x`? Vamos novamente entender cada passo que está sendo realizado.

Veja que `x` começa guardando o valor 3. Na primeira iteração do `for` a variável `var` assume o valor 2 e dessa forma o valor de `x` é atualizado para 3+2=5. Na segunda iteração `var` assume o valor 3 e assim o valor de `x` é atualizado para 5+3=8. Na iteração seguinte

`var` assume o valor 4 e `x` é atualizado para 8+4=12. Na última iteração `var` assume o valor 5 e então o valor de `x` recebe a sua última atualização: `x` = 12+5=17. Sendo assim seu valor final igual a 17.

Para terminar de falar do `for` vale a pena apresentar um exemplo onde dois *loops* são combinados, um dentro do outro. Suponha que temos uma matriz 5×5 cheia de zeros e queremos preencher cada posição dessa matriz com o número 1. O código a seguir executa essa tarefa.

```
> M <- matrix(0,ncol=5,nrow=5)
> for(i in 1:5){
+   for(j in 1:5){
+     M[i,j] <- 1
+   }
+ }
```

Ao final das linhas de comando descritas acima a matriz `M`, que foi iniciada como uma matriz nula, passa a ser uma matriz com todas as posições iguais a 1. Veja que para isso foi preciso usar dois comandos `for`, um para controlar o índice das linhas e o outro para controlar o índice das colunas. Quando um `for` é usado dentro do outro é preciso tomar cuidado, pois a variável de um deve ser diferente da variável do outro. Para o exemplo acima usamos as variáveis `i` e `j`.

E como ficaria o código se quiséssemos preencher cada posição de uma matriz com o número que indica a sua linha? Isso pode ser feito como no exemplo abaixo.

```
> M <- matrix(0,ncol=5,nrow=5)
> for(i in 1:5){
+   for(j in 1:5){
+     M[i,j] <- i
+   }
+ }
```

Veja que a única diferença com relação ao último código é que agora em vez da posição (i, j) da matriz receber o número 1 ela irá receber o número i. Implemente o exemplo no R e veja como fica a matriz `M` ao final dessa sequência de comandos.

2.3 while

Sintaxe:

```
while (afirmação){
        tarefas
}
```

Descrição:

Realiza as tarefas descritas dentro das chaves enquanto a afirmação for verdadeira. A afirmação tem que ser um objeto da classe `"logical"`.

Um detalhe importante para o controle de fluxo `while` é que as tarefas devem modificar a afirmação de forma a garantir que em algum momento a afirmação vire falsa, se não vamos entrar em *loop* infinito. Veja mais detalhes nos exemplos a seguir.

A sequência de comandos a seguir usa o controle de fluxo `while` para criar um vetor com os números de 1 até 100.

```
> vet <- 1
> while(length(vet) < 100){
+     i <- length(vet)
+     vet[i+1] <- i+1
+ }
```

Veja que o tamanho do vetor cresce em cada iteração, dessa forma sabemos que em algum momento `length(vet) < 100` será igual a `FALSE` e o *loop* termina.

E como podemos usar o `while` para criar um vetor com os números pares entre 1 e 100? Veja duas alternativas nas sequências de comandos a seguir.

Alternativa 1:

```
> vet <- 2
> while(length(vet) < 50){
+     i <- length(vet)
+     vet[i+1] <- 2*(i+1)
+ }
```

Alternativa 2:

```
> vet <- 2
> i   <- 1
> while(i < 50){
+     i <- i + 1
+     vet[i] <- 2*i
+ }
```

Você tem alguma outra sugestão?

2.4 repeat/break

Sintaxe:

```
repeat{
        tarefas
        if(afirmação)
                break
}
```

Descrição:

Repete as tarefas dentro das chaves indefinidamente. Por isso é preciso usar o `if` e o `break` para garantir a parada. Assim que a afirmação dentro do `if` for verdadeira, o `break` é executado e o *loop* interrompido.

Este controle de fluxo é muito semelhante ao `while`. Qualquer comando implementado com `repeat` pode ser reescrito usando o `while` e vice-versa. Por exemplo, também podemos usar o `repeat/break` para criar um vetor com os 100 primeiros inteiros, veja o código a seguir.

```
> y <- 1
> i <- 1
> repeat{
+     i <- i + 1
+     y[i] <- i
+     if(i==100)
+         break
+ }
```

O comando `break` pode ser aplicado dentro de qualquer *loop*, não necessariamente precisa ser dentro do `repeat`. Sempre a sua tarefa será interromper o *loop* quando alguma condição for satisfeita. Veja um exemplo do `break` usado para interromper um `for`.

```
> x <- 100
> for(i in 1:10){
+     x <- x+x*i
+     if(x>1000) break
+ }
```

Quais os valores das variáveis `x` e `i` ao final desse código? Vamos tentar acompanhar o passo-a-passo feito pelo computador para responder essa pergunta.

Quando o `for` começa a variável `x` guarda o valor 100 e a variável `i` o valor 1. Na primeira linha de comando dentro do `for` `x` passa a assumir o valor $100 + 100 \times 1 = 200$. Em seguida a condição `x>1000` é testada e como ela é falsa o processo não é interrompido. Na segunda iteração do `for` a variável `x` começa com o valor 200 e `i` com o valor 2. Logo na primeira linha o valor de `x` é atualizado para $200 + 200 \times 2 = 600$. Novamente, como a condição testada no `if` não é falsa o processo não é interrompido. No início da terceira iteração temos `x`=600 e `i`=3. Quando o valor de `x` é atualizado essa variável passa a guardar o valor $600 + 600 \times 3 = 2400$. É nesse momento que a condição testada passa a ser verdadeira, `x`>1000, e então o processo é interrompido.

Logo, ao final desse `for` a variável `x` guarda o valor 2400 e a variável `i` guarda o valor 3.

Exercícios - 2ª Semana

2.1 Primeiro guarde nas variáveis `a`, `b` e `c` o tamanho dos lados de um triangulo qualquer. Em seguida implemente um código no R que imprime na tela uma mensagem informando se o triângulo em questão é equilátero, isósceles ou escaleno. Teste o código implementado para diferentes valores de `a`, `b` e `c`.

2.2 Para cada item a seguir implemente um código no R para encontrar o que se pede. Não use os comandos `seq` ou algo parecido. Dica: Comece com um vetor nulo e use o(s) controle(s) de fluxo que achar adequado para preenchê-lo.

(a) A sequência com os 100 primeiros múltiplos de 3.

(b) A sequência com todos os múltiplos de 3 menores que 100.

(c) A sequência com os 100 primeiros números ímpares.

2.3 Usando os controles de fluxo vistos em sala de aula, faça o que se pede. Dica: a partir do segundo item vai ser preciso usar dois *loops*, um dentro do outro.

(a) Primeiro crie uma matriz 10×10 nula. Em seguida, usando um *loop*, preencha toda a sua primeira linha com o número 1.

(b) Comece novamente com uma matriz 10×10 nula. Preencha cada uma de suas linhas com o número que indica a linha em questão. Por exemplo, a primeira linha deve ser preenchido com 1, a segunda com 2 e assim por diante, até a décima linha que deve ser preenchida com 10.

(c) Agora comece com uma matriz 100×100 nula e implemente um *loop* que preenche cada coluna com o número correspondente da coluna, isto é, a primeira coluna com o número 1 e assim por diante.

(d) Crie uma matriz 100×100 tal que as posições em linhas pares recebem o número 2 e as posições em linhas ímpares o número 1.

2.4 Comece cada item a seguir com uma matriz 100×100 nula e não use o comando `seq` ou um vetor pronto.

(a) Crie uma matriz diagonal 100×100 cujos elementos da diagonal principal são os números de 1 até 100 em ordem crescente.

(b) Crie uma matriz diagonal 100×100 cujos elementos da diagonal principal são os números de 1 até 100 em ordem decrescente.

2.5 Usando os *loops* vistos em sala de aula crie as listas definidas em cada item a seguir.

(a) L1 é uma lista com 10 posições tal que cada posição `i` dessa lista guarda o número `i`.

(b) L2 é uma lista com 10 posições tal que cada posição `i` dessa lista guarda um vetor de tamanho `i` com todas as posições iguais a 1.

(c) L3 é uma lista com 10 posições tal que cada posição `i` dessa lista guarda um vetor com os 10 primeiros múltiplos de `i`.

(d) L4 é uma lista com 10 posições tal que cada posição `i` dessa lista guarda um vetor com os `i` primeiros múltiplos de 2.

(e) L5 é uma lista com 10 posições tal que cada posição `i` dessa lista guarda a matriz identidade de tamanho `i`×`i`.

2.6 Usando as listas L1 e L3 do exercício 2.5, faça o que se pede.

(a) Encontre o valor da soma de todos os números guardados em L1.

(b) Encontre o vetor definido pela soma de todos os vetores guardados em L3.

2.7 Usando a lista L4 do exercício 2.5, faça o que se pede.

(a) Crie um vetor `soma` tal que a sua posição `i` guarda a soma dos elementos do vetor alocado na posição `i` da lista L4.

(b) Crie um vetor `v` de `character` tal que a sua posição `i` guarda o objeto `soma[i]` concatenado com `"é um múltiplo de 5"` se a o valor da posição `i` do vetor `soma` for um múltiplo de 5. Caso contrário guarde na posição `i` de `v` o objeto `soma[i]` concatenado com `"não é um múltiplo de 5"`. Para concatenar textos use o comando `paste`.

(c) A partir do vetor `soma` ou do vetor `v` criados nos itens anteriores conte o número de vetores da lista L4 tais que a sua soma é um número múltiplos de 5. Não é para você visualizar `soma` ou `v` e contar, e sim para usar um *loop* e um contador para realizar essa conta.

2.8 Uma progressão aritmética (p.a.) é uma sequência numérica em que cada termo, a partir do segundo, é igual à soma do termo anterior com uma constante r. O número r é chamado de razão. O primeiro termo da sequência será chamado de x_0.

(a) Faça um código em R que determine os 100 primeiros termos da progressão aritmética cuja termo inicial é $x_0 = 2$ e a razão é $r = 3$. Vamos chamar o vetor com os elementos dessa sequência de y.

Depois que y estiver construído,

(b) Faça um código que determine a soma dos 35 primeiros termos dessa sequência. Compare o valor encontrado com o valor fornecido pela fórmula da soma de uma p.a.. Você lembra dessa fórmula?

(c) Faça um código que conte um número de elementos em y múltiplos de 4 (lembre do comando `%%` visto na semana passada, que fornece o resto da divisão).

(d) Faça um código que conte um número de elementos em y múltiplos de 4 e múltiplos de 5 simultaneamente.

(e) Faça um código que conte um número de elementos em y múltiplos de 4 ou múltiplos de 5.

(f) Vamos agora criar uma nova sequência x a partir da sequência y da seguinte maneira: cada termo par da sequencia y será mantido e os termos ímpares serão substituídos por 0. Faça um código que gere a sequência x assim definida.

2.9 A famosa sequência de Fibonacci é definida da seguinte maneira: os dois primeiros elementos são iguais a $[1, 1]$ e a partir do terceiro elemento cada termo da sequência é definido como a soma dos dois termos anteriores. Por exemplo, o terceiro termo é $2\ (= 1 + 1)$, o quarto termo é $3\ (= 1 + 2)$, o quinto termo é $5\ (= 2 + 3)$ e assim por diante.

(a) Faça um código em R para encontrar os 12 primeiros números da sequência de Fibonacci.

(b) Faça um código em R para encontrar todos os números da sequência de Fibonacci menores que 300.

(c) Faça um código em R que determine o número de termos da sequência de Fibonacci menores que 1.000.000. Veja que nesse caso você não precisa (e nem deve!) guardar os termos da sequência, apenas precisa contar o número de elementos.

Semana 3: Funções e o Conceito de Variável Local

Nesse semana vamos ver como podemos criar funções dentro do R para tornar nosso código mais dinâmico e prático. Além disso será apresentado o conceito de variável local, que é de grande importância em qualquer linguagem de programação.

3.1 Funções

Uma função dentro dentro de uma linguagem de programação é definida pelos seguintes itens:

- nome da função;
- argumentos (entrada);
- sequência de comandos (corpo);
- retorno (saída).

Tanto os argumentos quanto a saída podem ser nulos, ou vazios, mas em geral eles são definidos.

Depois que uma função é definida para executar a sua sequência de comandos basta chamá-la pelo nome, passando os argumentos de entrada caso estes não sejam nulos. Veremos alguns exemplos ao longo da aula.

Para definir uma função no R deve ser usada a seguinte sintaxe.

Sintaxe:

```
nome_da_funcao <- function("argumentos"){
        "corpo da função"
        return("retorno")
}
```

Descrição:

Argumentos: Define as variáveis cujos valores serão atribuídos pelo usuário quando a função for chamada.

Corpo da Função: Contém os cálculos e tarefas executadas pela função.

Retorno: Indica qual o valor que a função retorna como saída.

Vejamos alguns exemplos. Primeiro vamos construir uma função que retorna o maior entre dois valores passados como argumentos. Essa função já existe pronta no R e se chama `max`, o que vamos fazer é entender como ela funciona criando a nossa própria função.

```
> maior <- function(a,b){
+     if(a>b)
+         return(a)
+     else
+         return(b)
+ }
```

Depois da função definida e compilada podemos chamá-la sem ter que digitar todo o código novamente. Veja o que acontece quando a função é chamada no prompt do R.

```
> maior(3,2)
[1] 3
> maior(-1,4)
[1] 4
> maior(10,10)
[1] 10
```

Também podemos optar por guardar a saída da função em uma variável, o que pode ser bastante útil.

```
> x<-maior(-5,-3)
```

Qual o valor de `x` depois desse comando?

E se quiséssemos encontrar o maior entre 3 números? Temos duas alternativas. A primeira é usar a função acima composta com ela mesma.

```
> maior(1,maior(2,5))
[1] 5
> maior(10,maior(6,-1))
[1] 10
> maior(-3,maior(0,1))
[1] 1
```

Outra alternativa é criar uma função com três argumentos própria para isso.

```
> maior_de_3 <- function(a,b,c){
+     if(a>b && a>c){
+         return(a)
+     } else {
+         if(b>c)
+             return(b)
+         else
+             return(c)
+     }
+ }
> y <- maior_de_3(2,15,6)
> y
[1] 15
```

Vamos agora fazer uma função que recebe como argumento um número natural `n` e retorna um vetor com os n primeiros múltiplos de 3. Na semana passada, na aula prática, fizemos isso para $n = 100$, agora estamos fazendo para qualquer n, quem vai escolher o seu valor é o usuário quando ele chamar a função.

Temos muitas maneiras de fazer isso, veja duas alternativas usando o `for`.

```
> multiplos_3 <- function(n){
+     vet <- NULL
+     for(i in 1:n){
+         vet[i] <- 3*i
+     }
+     return(vet)
+ }
```

Outra possibilidade seria:

```
> multiplos_3 <- function(n){
+     vet <- 3
+     for(i in 2:n){
+         vet[i] <- vet[i-1] + 3
+     }
+     return(vet)
+ }
```

As duas funções funcionam de forma igual, são formas diferente de encontrar os múltiplos de 3. Vejamos a saída quando uma dessas funções é chamada.

```
> multiplos_3(10)
 [1]  3  6  9 12 15 18 21 24 27 30
> multiplos_3(15)
 [1]  3  6  9 12 15 18 21 24 27 30 33 36 39 42 45
```

E se colocarmos como argumento um número que não seja natural, o que acontece? Ele funciona de forma não previsível, quando não dá erro. Veja:

```
> multiplos_3(1.5)
[1] 3 6
> multiplos_3(-1)
Error in vet[i] <- vet[i - 1] + 3 : replacement has length zero
```

Por isso é muito importante verificar se o usuário passou os argumentos de forma correta. Para esse exemplo precisamos apenas checar se o valor de `n` passado pelo usuário é realmente um natural. E como podemos fazer isso?

Uma alternativa é verificar se o valor de `n` é positivo e se é múltiplo de 1. Caso contrário passamos uma mensagem de erro para o usuário e interrompemos o processo a partir do comando `stop`.

```
> multiplos_3 <- function(n){
+     if((n<=0)||(n%%1 != 0)){
+         stop("n tem que ser um natural")
+     }
```

```
+     vet <- NULL
+     for(i in 1:n){
+         vet[i] <- 3*i
+     }
+     return(vet)
+ }
> multiplos_3(-1)
Error in multiplos_3(-1) : n tem que ser um natural
> multiplos_3(1.5)
Error in multiplos_3(1.5) : n tem que ser um natural
```

3.2 Variáveis Locais

As variáveis locais são aquelas usadas somente no corpo da função. Para garantirmos que essa variável realmente não está assumindo um valor pré definido é preciso que ela seja iniciada dentro do corpo da função.

Ainda trabalhando com a função que retorna os múltiplos de 3, veja como as variáveis locais devem ser iniciadas dentro da própria função.

Forma Correta:

```
> multiplos_3 <- function(n){
+     vet <- NULL
+     for(i in 1:n){
+         vet[i] <- 3*i
+     }
+     return(vet)
+ }
```

Forma Errada:

```
> multiplos_3 <- function(n){
+     for(i in 1:n){
+         vet[i] <- 3*i
+     }
+     return(vet)
+ }
```

No exemplo acima a variável local em questão é o objeto `vet`, que é um array da classe `"numeric"`. Repare que na forma correta ele é iniciado como nulo ao início da função. Mas de que forma isso pode influenciar no desempenho da função?

Veja como a forma correta e a forma errada se comportam de maneira diferente. Se digitarmos

```
> vet <- c(1:20)
> y <- multiplos_3(10)
```

a variável `y` receberá valores diferentes no caso da função `multiplos_3` ter sido implementada na forma correta ou na forma errada. Se a implementação foi correta temos a seguinte saída:

```
> y
 [1]  3  6  9 12 15 18 21 24 27 30
```

Como o esperado, `y` será um vetor com 10 posições e em cada posição um múltiplo de 3. Porém, se a implementação foi na forma errada a saída será:

```
> y
 [1]  3  6  9 12 15 18 21 24 27 30 11 12 13 14 15 16 17 18 19 20
```

A variável `y` retornada pela função será um vetor com 20 posições onde as 10 primeiras possuem múltiplos de 3 e as 10 últimas o valor do índice da posição.

Tanto para a implementação correta quanto para a implementação errada a variável `vet` permanece a mesma após as duas linhas de comando. Faça o teste no R e verifique!

> ⚠ Dentro de uma função podemos ter dois tipos de variáveis: aquelas passadas como argumento e as variáveis locais. As variáveis passadas como argumento **não** podem ser iniciadas dentro do corpo da função, nesse caso perderíamos o valor informado pelo usuário. Já as variáveis locais, aquelas que não foram passadas como argumento, **devem** ser iniciadas dentro da função.

Veja mais um exemplo de função antes da seção de exercícios.

Suponha agora que queremos criar uma função que em vez de retornar os `n` primeiros múltiplos de 3 passe a retornar os `n` primeiros múltiplos de `m`. Veja como esta função pode ser implementada.

```
> multiplos <- function(n,m){
+     if((n<=0)||(n%%1 != 0)){
+         stop("n tem que ser um natural")
+     }
+     if((m<=0)||(m%%1 != 0)){
+         stop("m tem que ser um natural")
+     }
+     vet <- NULL
+     for(i in 1:n){
+         vet[i] <- m*i
+     }
+     return(vet)
+ }
```

Mas essa não é a única maneira de implementar tal função. Podemos fazer isso de várias forma diferete. Veja outra opção.

```
> multiplos <- function(n,m){
+     if((n<=0)||(n%%1 != 0)){
+         stop("n tem que ser um natural")
+     }
+     if((m<=0)||(m%%1 != 0)){
+         stop("m tem que ser um natural")
+     }
+     vet <- m
+     for(i in 2:n){
+         vet[i] <- vet[i-1] + m
+     }
+     return(vet)
+ }
```

Exercícios - 3ª Semana

3.1 (a) Implemente uma função que recebe como argumento dois números reais e retorna o menor entre eles.

(b) Implemente uma função que recebe como argumento três números reais e retorna o menor entre eles.

OBS: Não use a função `min` pronta no R.

3.2 Implemente uma função que recebe como argumento o tamanho de cada lado de um triângulo e retorna um texto informando se o triângulo é equilátero, isósceles ou escaleno. Antes de fazer o exercício pense:

- Quantos argumentos a sua função vai receber?
- Quais são os valores aceitáveis para esses argumentos?
- Qual o tipo de objeto que a sua função deve retornar?

3.3 Implemente uma função que recebe como argumento um vetor de números reais e retorna a quantidade de elementos positivos desse vetor. Não se esqueça de inciar todas as variáveis locais usadas em sua função. OBS: Depois que a sua função estiver pronta invente vetores para serem passados como argumento de forma a verificar se a função está funcionando como o esperado. Por exemplo, use a função para contar o número de elementos positivos em $v = (1.0, 3.2, -2.1, 10.6, 0.0, -1.7, -0.5)$.

item Implemente uma função que recebe como argumento um vetor de números reais $v|$ e um número real α e retorna o número de elementos do vetor v menores que α.

3.4 Para cada item a seguir faça o que se pede. Não se esqueça de fazer as verificações necessárias para garantir que o usuário passe os argumentos de forma correta.

(a) Implemente uma função que recebe como argumento as variáveis n e m e retorna um vetor que guarda os n primeiros múltiplos de m.

(b) Implemente uma função que recebe como argumento as variáveis m e K e retorna um vetor com os múltiplos de m menores que K.

(c) Implemente uma função que recebe como argumento as variáveis m e K e retorna a quantidade de múltiplos de m menores que K.

(d) Classifique cada variável que aparece dentro das funções implementadas nesse exercício como "variável local" ou "argumento da função". Todas as variáveis locais foram iniciadas dentro do corpo da função?

3.5 Para cada item a seguir faça o que se pede. Não se esqueça de fazer as verificações necessárias para garantir que o usuário passe os argumentos de forma correta.

(a) Implemente uma função que recebe como entrada um número natural n e retorna uma matriz $n \times n$ tal que as posições em linhas pares recebem o número 2 e as posições em linhas ímpares o número 1.

(b) Implemente uma função que recebe como entrada um numero natural n e retorna uma matriz $n \times n$ tal que a coluna i dessa matriz guarda o elemento i. Por exemplo, a primeira coluna deve ser preenchida com 1, a segunda com 2 e assim por diante, até a n-ésima coluna que deve ser preenchida com o número n.

(c) Implemente uma função que recebe como entrada um numero natural n e retorna uma matriz diagonal $n \times n$ tal que na diagonal principal aparecem os valores de 1 até n. Por exemplo, a posição $(1, 1)$ deve ser preenchido com 1, a posição $(2, 2)$ com 2 e assim por diante. As demais posições devem ser nulas, uma vez que a matriz de saída é diagonal.

3.6 Para cada item a seguir faça o que se pede. Não se esqueça de fazer as verificações necessárias para garantir que o usuário passe os argumentos de forma correta.

(a) Implemente uma função que recebe como entrada um vetor de número reais v e retorna uma matriz diagonal com os elementos de v guardados na diagonal principal.

(b) Implemente uma função que recebe como entrada um vetor de número reais v e retorna uma matriz quadrada cujas colunas são iguais ao vetor v.

(c) Implemente uma função que recebe como entrada um vetor de número reais v e retorna uma matriz quadrada cujas linhas são iguais ao vetor v.

3.7 Para cada item a seguir faça o que se pede. Não se esqueça de fazer as verificações necessárias para garantir que o usuário passe os argumentos de forma correta.

(a) Implemente uma função que recebe como argumento o valor inicial x_0 e retorna os 10 primeiros termos de uma p.a. cuja razão é 3.

(b) Implemente uma função que recebe como argumento o valor inicial x_0, a razão r e retorna um vetor com os 10 primeiros termos dessa p.a.

(c) Implemente uma função que recebe como argumentos o valor inicial x_0, a razão r, um inteiro n e retorna um vetor com os n primeiros termos de uma p.a. Vamos chamar essa função de `pa`.

(d) Usando a função `pa` implementada no item anterior, implemente outra função que recebe como argumento o valor inicial x_0, a razão r, um inteiro n e retorna a soma dos n primeiros termos de uma p.a. Para isso, chame a função `pa` dentro da nova função que está sendo implementada neste item.

(e) Classifique cada variável que aparece dentro das funções implementadas nos itens 3.7c e 3.7d como "variável local" ou "argumento da função". Todas as variáveis locais foram iniciadas dentro do corpo da função?

3.8 Implemente uma função que:

(a) recebe como argumento a variável n e retorna um vetor com os n primeiros termos da sequência de Fibonacci.

(b) recebe como argumento a variável K e retorna um vetor com os todos os termos da sequência de Fibonacci menores que K.

(c) recebe como argumento a variável K e retorna o número de termos da sequência de Fibonacci menores que K.

3.9 Suponha que a seguinte função foi implementada no R com o intuito de gerar um vetor com os naturais de 1 até `n`, este passado como argumento pelo usuário.

```
> f <- function(n){
+     for(i in 1:n){
+         v[i] <- i
+     }
+     return(v)
+ }
```

Após `f` ter sido definida ela foi chamada a partir da seguinte sequência de comandos:

```
> v <- c(0,0,0,0,0)
> vet <- f(3)
```

(a) Sem usar o computador, qual o valor de `v` e qual o valor de `vet` ao final dessa sequência de comandos?

(b) A saída da função `f`, guardada no vetor `vet`, é como o esperado? Caso negativo, qual mudança você faria em `f` para que essa função passe a funcionar corretamente?

3.10 Uma progressão geométrica (p.g.) é uma sequência numérica em que cada termo, a partir do segundo, é igual ao produto do termo anterior por uma constante q. Esta constante q é chamada razão da progressão geométrica. O primeiro termo da sequência será chamado de x_0.

(a) Implemente uma função que recebe como argumento as variáveis x_0, q e n e retorna uma vetor com os n primeiros termos de uma p.g. cujo termo inicial x_0 e a razão é q.

(b) Implemente uma função que recebe como argumento as variáveis x_0, q e m e retorna a soma dos m primeiros termos de uma p.g. cujo termo inicial x_0 e a razão é q.

(c) Usando as funções implementadas, encontre a soma dos 10 primeiros termos da p.g. de razão $\frac{1}{2}$ e valor inicial $\frac{1}{2}$.

(d) Usando as funções implementadas, encontre a soma dos 30 primeiros termos da p.g. de razão $\frac{1}{2}$ e valor inicial $\frac{1}{2}$.

(e) Você acredita que essa soma converge, ou seja, você acha que

$$\frac{1}{2} + \frac{1}{2}^2 + \ldots \frac{1}{2}^n = \sum_{i=1}^{n} \frac{1}{2}^i \xrightarrow[n\to\infty]{} a \in \mathbb{R}?$$

(f) Como você poderia demonstrar a ocorrência ou não dessa convergência?

OBS: Use o comando `options(digits=22)` para usar o número máximos de casas decimais que o R aceita, 22.

Semana 4: Algoritmos para Cálculos Estatísticos

A partir dessa semana vamos nos concentrar na análise e implementação de algoritmos e por isso será evitado a apresentação de códigos prontos na linguagem R. Em vez disso serão apresentados e discutidos alguns pseudo-códigos (passo-a-passo), como costuma aparecer na maioria dos livros. As aulas práticas serão dedicadas à implementação dos algoritmos a partir dos pseudo-códigos apresentados em sala de aula.

Nesse capítulo vamos discutir como podemos encontrar os valores para algumas estatísticas a partir de um conjunto de dados guardados em um *array*. Por falta de tempo nem todas as seções serão vistas em detalhes na aula teórica, algumas ficaram para leitura em casa. A ideia principal é rever alguns conceitos e pensar como podemos "ensinar" o computador a encontrá-los.

A maioria das funções que serão implementadas nessa seção já estão prontas no R. Mais uma vez não vamos usar tais funções prontas e sim parar para pensar como elas foram implementadas e escrever nosso próprio código.

4.1 Máximo

Definição 4.1.1 *Seja A um conjunto de dados. Dizemos que $a \in A$ é o máximo desse conjunto se $a \geq r \ \forall \ r \in A$.*

Queremos escrever um pseudo-código que recebe como entrada um *array* de dados e retorna o seu máximo de acordo com a definição acima. Na semana anterior foi implementando uma função que recebe dois ou três números e retorna o maior entre eles. O que está sendo pedido agora é o caso geral: encontrar o máximo entre todos os elementos guardados em um vetor.

A ideia principal do algoritmo apresentado a seguir é percorrer um vetor guardando o maior elemento encontrado até o momento. O algoritmo termina quando o vetor já foi percorrido por completo. Veja como isso pode ser feito no passo-a-passo a seguir.

Entrada: v = array com os dados.
Saída: valor máximo em v.

1 Defina n = tamanho do vetor v;
2 Faça $max = v_1$;
3 Inicie $i = 2$;
4 Se $v_i > max, max = v_i$;
5 Incremente i: $i = i + 1$;
6 Se $i \leq n$, volta para a linha 4;
7 Retorne max.

Na linha 2 a variável *max* guarda o primeiro elemento do vetor, pois no início do algoritmo esse é o único valor observado, logo o máximo até o momento. Na linha 4 acontece a troca do valor guardado em *max*, caso o novo valor observado seja maior que o máximo até o momento.

Veja que o passo-a-passo descreve um *loop*: as linhas 4 - 6 se repetem várias vezes. Como podemos reproduzir isso em uma linguagem de programação? A resposta é: usando um dos controles de fluxo para *loop* vistos no Capítulo 2.

A escolha do controle de fluxo depende do algoritmo. No caso deste temos a variável i que começa assumindo o valor 2 e é incrementada até atingir o valor n, ou seja, conhecemos o valor inicial e final da variável i. Dessa forma parece que o *for* é uma opção bastante razoável, onde i sera a variável definida dentro dele.

4.2 Mínimo

Definição 4.2.1 *Seja A um conjunto de dados. Dizemos que $a \in A$ é o mínimo desse conjunto se $a \leq r \ \forall \ r \in A$.*

Queremos agora escrever um pseudo-código que recebe como entrada um *array* de dados e retorna o seu mínimo, de acordo com a definição acima.

A ideia principal do algoritmo proposto é percorrer um vetor guardando o menor elemento encontrado até o momento. O algoritmo termina quando o vetor já foi percorrido por completo. Repare que esse algoritmo é análogo ao apresentado para o máximo, por isso a elaboração do pseudo-código (passo-a-passo) será deixada como exercício.

4.3 Média

Definição 4.3.1 *Seja $A = \{a_1, a_2, \ldots, a_n\}$ um conjunto finito de dados e n o seu número de elementos. A média amostral desse conjunto é definido por:*

$$média \ = \frac{a_1 + a_2 + \ldots + a_n}{n} = \frac{\sum_{i=1}^{n} a_i}{n}.$$

Queremos agora escrever um pseudo-código que recebe como entrada um *array* de dados e retorna a sua média de acordo com a definição acima. A ideia principal será percorrer o vetor v somando seus elementos uma a um. Quando v já tiver sido percorrido por completo, basta dividir a soma final pelo número total de elementos que encontraremos a média.

Entrada: v = array com os dados.
Saída: a média amostral dos valores de v.

1 Defina n = tamanho do vetor v;
2 Inicie $soma = 0$;
3 Inicie $i = 1$;

4 Incremente a variável *soma*: $soma = soma + v_i$;
5 Incremente a posição a ser somada: $i = i + 1$;
6 Se $i \leq n$, volta para a linha 4;
7 Faça $media = soma/n$;
8 Retorne *media*.

Repare que temos novamente a ocorrência de um *loop* dentro do algoritmo, basta notar a repetição das linhas 4 - 6. Então quando esse passo-a-passo for implementado em uma linguagem de programação sera preciso escolher um controle de fluxo para realizar esse *loop*.

4.4 Mediana

Definição 4.4.1 *Seja $A = \{a_1, a_2, \ldots, a_n\}$ um conjunto finito de dados e n o seu número de elementos. Considere a notação de estatística de ordem tal que $A = \{a_{(1)}, a_{(2)}, \ldots, a_{(n)}\}$ e $a_{(1)} \leq a_{(2)} \leq \ldots \leq a_{(n)}$. A mediana desse conjunto é definida informalmente como o ponto que separa o conjunto ordenado em duas partes de mesmo tamanho. Sua definição formal encontra-se a seguir.*

$$mediana = \begin{cases} a_{(\frac{n+1}{2})} & , \text{ se } n \text{ é ímpar;} \\ (a_{(\frac{n}{2})} + a_{(\frac{n}{2}+1)})/2 & , \text{ se } n \text{ é par;} \end{cases}$$

Queremos agora escrever um pseudo-código que recebe como entrada um *array* de dados e retorna a sua mediana de acordo com a definição acima.

Entrada: v = array com os dados.
Saída: a mediana dos valores de v.
1 Defina n = tamanho do vetor v;
2 Defina $\widetilde{v}$ como o vetor v ordenado;
3 Se n é ímpar, faça $mediana = \widetilde{v}_{\frac{n+1}{2}}$;
4 Se n é par, faça $mediana = (\widetilde{v}_{\frac{n}{2}} + \widetilde{v}_{\frac{n}{2}+1})/2$;
5 Retorne *mediana*.

Veja que nesse algoritmo não temos a ocorrência de *loops*.

Na linha 2 do passo-a-passo acima o vetor v é ordenado. Ainda não aprendemos como ordenar vetores, isso será visto somente no Capítulo 8. Por isso, pro enquanto, usaremos o comando `sort` do R para ordenar vetores.

4.5 Quartis

Definição 4.5.1 *Seja $A = \{a_1, a_2, \ldots, a_n\}$ um conjunto finito de dados e n o seu número de elementos. Considere a notação de estatística de ordem tal que $A = \{a_{(1)}, a_{(2)}, \ldots, a_{(n)}\}$*

e $a_{(1)} \leq a_{(2)} \leq \ldots \leq a_{(n)}$. *Os quartis desse conjunto é definido informalmente como os três pontos que separam o conjunto ordenado em quatro partes de mesmo tamanho.*

Existem alguns métodos para encontrar os quartis de um conjunto de dados. Vejamos a seguir dois desses métodos.

Método 1

1. Encontre a mediana do conjunto de dados e divida o conjunto ordenado em dois subconjuntos. Não inclua a mediana em nenhum deles.
2. O primeiro quartil é a mediana do subconjunto com os menores valores.
3. O segundo quartil é a mediana do conjunto original.
4. O terceiro quartil é a mediana do subconjunto com os maiores valores.

Método 2

1. Encontre a mediana do conjunto de dados e divida o conjunto ordenado em dois subconjuntos. Caso a mediana seja um elemento do conjunto original ela deve ser incluída em ambos os subconjuntos.
2. O primeiro quartil é a mediana do subconjunto com os menores valores.
3. O segundo quartil é a mediana do conjunto original.
4. O terceiro quartil é a mediana do subconjunto com os maiores valores.

Queremos escrever um pseudo-código que recebe como entrada um *array* de dados e retorna cada um dos seus quartis. A ideia será simplesmente seguir um dos métodos acima e recorrer ao cálculo da mediana, que já é conhecido. A seguir isso será feito considerando Método 1. Faça como exercício o pseudo-código considerando o Método 2.

Entrada: v = array com os dados.
Saída: quartis do conjunto de dados v.

1 Defina n = tamanho do vetor v;
2 Defina $\widetilde{v}$ como o vetor v ordenado;
3 Se n for par, seja $k = n/2$ e $j = k + 1$;
4 Se n for ímpar, seja $k = (n-1)/2$ e $j = k + 2$;
5 Defina v_1 como sendo o vetor $\widetilde{v}$ das posições de 1 até k;
6 Defina v_2 como sendo o vetor $\widetilde{v}$ das posições de j até n;
7 Defina q_1 = mediana de v_1;
8 Defina q_2 = mediana de v;
9 Defina q_3 = mediana de v_2;
10 Retorne o vetor (q_1, q_2, q_3).

Veja que nas linhas 3 e 4 são definidas as posições de $\widetilde{v}$ onde termina v_1 e começa v_2, considerando o Método 1. Essas linhas devem mudar para o Método 2.

4.6 Frequências

Quando temos um vetor de dados pode ser interessante conhecer a frequência com que cada valor aparece dentro desse vetor. Para isso podemos pensar em dois tipos de frequência: absoluta e acumulada.

Definição 4.6.1 *Seja $A = \{a_1, \ldots, a_n\}$ um finito conjunto de dados. A frequência absoluta de um valor x é definida pelo número de vezes que esse valor aparece no conjunto A.*

Definição 4.6.2 *Seja $A = \{a_1, \ldots, a_n\}$ um conjunto de dados com n elementos. A frequência absoluta de um valor x é definida pela razão entre o número de vezes que esse valor aparece no conjunto A e n.*

Queremos escrever um pseudo-código que recebe como entrada um *array* de dados e retorne uma matriz de duas linhas. Na primeira linha deve aparecer os diferentes valores que aparecem no *array* de entrada. Na segunda linha deve aparecer a frequência absoluta de cada valor.

A ideia principal desse pseudo-código é "varrer" o *array* de entrada "lendo" todos os valores guardados nele. Toda vez que um valor novo é encontrado ele deve ser guardado em um *array* e a sua frequência absoluta iniciada com o valor 1. Se um valor é encontrado mais de uma vez, apenas é preciso incrementar a sua frequência absoluta. Veja o passo-a-passo a seguir.

Entrada: v = array com os dados.
Saída: matriz M com as frequências absolutas.

1 Defina n = tamanho do vetor v;
2 Inicie dois *arrays*: val e $freq$;
3 Faça $val_1 = v_1$ e $freq_1 = 1$;
4 Inicie $i = 2$;
5 Se existe o valor v_i dentro do *array* val, vá para a linha 7;
6 Caso contrário, vá para a linha 8;
7 Seja k a posição do valor v_i em val, incremente $freq_k$ e vá pra linha 9;
8 Seja j o tamanho do *array* val, faça val_{j+1} receber v_i e $freq_{j+1}$ receber 1;
9 Incremente i: $i = i + 1$;
10 Se $i \leq n$, volte para a linha 5;
11 Crie uma matriz M cuja primeira linha contém os elementos de val e a segunda linha os
12 elementos de $freq$;
13 Retorne M.

Veja que na linha 5 é feito a busca pelo elemento v_i dentro do *array* val. Ficará mais simples o código se você criar uma função própria para isso. Ou seja, defina uma função a parte que recebe como entrada um *array* e um valor e retorna `TRUE` se existe o valor dentro do *array* ou `FALSE` caso não exista.

Se o interesse for na frequência relativa basta dividir os elementos do vetor $freq$ por n antes de criar a matriz de saída.

4.7 Moda

Definição 4.7.1 *Seja $A = \{a_1, a_2, \ldots, a_n\}$ um conjunto finito de dados. A moda desse conjunto é definida pelo valor mais frequente dentro do conjunto de dados.*

Queremos escrever um pseudo-código que recebe como entrada um *array* de dados e retorna a sua moda.

A ideia principal é procurar o valor com maior frequência. Como já sabemos encontrar as frequências de cada valor (Seção 4.6) o algoritmo fica bem simples: basta procurar onde está o maior valor na segunda linha da matriz de frequências.

Entrada: v = array com os dados.
Saída: a moda de v.

1 Seja M a matriz de frequências criada a partir de v;
2 Seja j a coluna referente ao maior elemento da linha 2 de M;
3 Retorne $M_{1,j}$;

A parte difícil é encontrar a posição do maior elemento da linha 2 de M. Mas isso pode ser feito em uma função a parte para simplificar o código.

4.8 Amplitude Total

Definição 4.8.1 *Seja A um conjunto de dados, max o seu máximo e min o seu mínimo, como definidos anteriormente. A amplitude total de A é definida pela diferença $max - min$.*

Queremos agora escrever um pseudo-código que recebe como entrada um *array* de dados e retorna a sua amplitude total. Veja que podemos considerar que já sabemos encontrar o máximo e o mínimo. Faça você mesmo esse pseudo-código como exercício.

4.9 Distância Interquartílica

Definição 4.9.1 *Seja A um conjunto de dados, q_1 o primeiro quartil e q_3 o terceiro quartil. A distância interquartílica de A é definida por $q_3 - q_1$.*

Queremos agora escrever um pseudo-código que recebe como entrada um *array* de dados e retorna a sua distância interquartílica. Faça você mesmo esse pseudo-código considerando que você já sabe encontrar o primeiro e terceiro quartil.

4.10 Variância Amostral

Definição 4.10.1 *Seja $A = \{a_1, a_2, \ldots, a_n\}$ um conjunto finito de dados e n o seu número de elementos. Seja m a média amostral de A. A variância amostral desse conjunto*

é definido por:

$$S^2 = \frac{\sum_{i=1}^{n}(a_i - m)^2}{n-1}.$$

Queremos escrever um pseudo-código que recebe como entrada um *array* de dados e retorna a sua variância amostral. A ideia principal é primeiro encontrar a média amostral e em seguida percorrer o *array* de entrada calculando a variância amostral.

Entrada: v = array com os dados.
Saída: variância amostral dos valores de v.

1 Defina n = tamanho do vetor v;
2 Defina m = média amostral de v;
3 Inicie $soma = 0$;
4 Inicie $i = 1$;
5 Incremente a variável $soma$: $soma = soma + (v_i - m)^2$;
6 Incremente i: $i = i + 1$;
7 Se $i \leq n$, volta para a linha 5;
8 Faça $s_2 = soma/(n-1)$;
9 Retorne s_2.

4.11 Desvio Médio

Definição 4.11.1 *Seja $A = \{a_1, \ldots, a_n\}$ um conjunto de dados e n o seu número de elementos. Seja m a média amostral de A. O desvio médio é definido por:*

$$dm = \frac{\sum_{i=1}^{n}|a_i - m|}{n}.$$

Queremos agora escrever um pseudo-código que recebe como entrada um *array* de dados e retorna o seu desvio médio. Esse pseudo-código é bem parecido com o da variância amostral. Faça você mesmo como exercício.

4.12 Covariância Amostral

Definição 4.12.1 *Seja $A = \{a_1, a_2, \ldots, a_n\}$ e $B = \{b_1, b_2, \ldots, b_n\}$ dois conjuntos de dados pareados e n o seu número de elementos. Seja m_A a média amostral de A e m_B a média amostral de B. A covariância amostral entre os conjuntos A e B é definida por:*

$$cov_{A,B} = \frac{\sum_{i=1}^{n}(a_i - m_A)(b_i - m_B)}{n-1}.$$

Queremos agora escrever um pseudo-código que recebe como entrada dois *arrays* de dados e retorna a covariância amostral entre eles.

Entradas: v = array com os dados; w = array com os dados.
Saída: covariância amostral entre os valores em v e w.

1 Defina n = tamanho do vetor v;
2 Defina k = tamanho do vetor w;
3 Se $n \neq k$ retorna erro;
4 Defina m_v = média amostral de v;
5 Defina m_w = média amostral de w;
6 Inicie $soma = 0$;
7 Inicie $i = 1$;
8 Incremente a variável $soma$: $soma = soma + (v_i - m_v)(w_i - m_w)$;
9 Incremente i: $i = i + 1$;
10 Se $i \leq n$, volta para a linha 8;
11 Faça $cov = soma/(n-1)$;
12 Retorne cov.

Exercícios - 4ª Semana

Para todos os exercícios a seguir não se esqueça de:

- Verificar sempre se as entradas passadas pelo usuário são viáveis para os cálculos das funções.
- Inventar várias entradas para as funções implementadas a fim de verificar se elas estão funcionando corretamente.
- Sempre que possível chamar as funções já implementadas dentro de uma nova função. Assim você simplifica bastante seu código.

4.1 Faça o que se pede sem usar as funções `max` e `which.max` do R.

(a) No computador implemente o algoritmo visto em sala de aula que recebe como entrada um vetor v e retorna o seu valor máximo.

(b) Em seu caderno escreva um pseudo-código para o algoritmo que recebe como entrada um vetor v e retorna a posição onde se encontra o máximo. Nesse item não é para usar o computador.

(c) Agora, novamente no computador, implemente uma nova função que executa o pseudo-código elaborado no item 1b. Não use o mesmo nome da função implementada no item 1a.

4.2 Faça o que se pede sem usar as funções `min` e `which.min` do R.

(a) Primeiro escreva em seu caderno um pseudo-código para o algoritmo que recebe como entrada um vetor v e retorna o valor mínimo guardado em v. Nesse item não é para usar o computador.

(b) Agora no computador implemente uma função que executa o pseudo-código elaborado do item 2a.

(c) De volta ao caderno escreva um pseudo-código para o algoritmo que recebe como entrada um vetor v e retorna a posição onde se encontra o mínimo. Nesse item não é para usar o computador.

(d) Novamente no computador implemente no uma nova função que executa o pseudo-código elaborado no item 2c. Não use o mesmo nome da função implementada no item 2b.

4.3 Faça o que se pede sem usar as funções `mean` e `sum` do R. No computador implemente o algoritmo visto em sala de aula que recebe como entrada um vetor v e retorna a média amostral dos valores em v.

OBS: Se quiser use a função `mean` para comparação.

4.4 Faça o que se pede sem usar a função `median` do R. Para ordenar o vetor de entrada use a função `sort` do R. No computador implemente o algoritmo visto em sala de aula que recebe como entrada um vetor v e retorna a mediana de v.

OBS: Se quiser use a função `median` para comparação.

4.5 Faça o que se pede sem usar a função `quantile` do R. Para ordenar o vetor de entrada use a função `sort` do R.

(a) No computador implemente o pseudo-código visto em sala de aula, baseado no Método 1, que recebe como entrada um vetor v e retorna os três quartis.

(b) No caderno escreva um pseudo-código que calcula os três quartis a partir do Método 2.

(c) No computador implemente o pseudo-código escrito no item acima.

4.6 Faça o que se pede sem usar a função `table` ou algo parecido.

(a) No computador implemente o algoritmo visto em sala de aula que recebe como entrada um vetor v e retorna uma matriz cuja primeira linha guarda os diferentes valores de v e a segunda as frequências absolutas com que esses valores aparecem no vetor de dados.

(b) Repita o item 6a gerando agora as frequências relativas. Não use o mesmo nome da função implementada no item 6a.

(c) Faça agora uma outra função que recebe como argumento além do vetor v um objeto to tipo `"logic"`, vamos chamá-lo de `rel`. Se $rel = TRUE$ a matriz de saída deve guardar as frequências relativas, caso contrário ela guarda a frequência absoluta.

4.7 Usando as funções implementadas em 6, implemente o algoritmo visto em sala de aula que recebe como entrada um vetor de dados e retorna a moda desse vetor.

4.8 (a) Primeiro escreva no caderno um pseudo-código para o algoritmo que recebe como entrada um vetor v e retorna a sua Amplitude Total. Não é para usar o computador ainda. Considere que você já sabe calcular os valores mínimo e máximo do seu vetor de entrada.

(b) Agora no computador implemente o pseudo-código feito do item 8a usando as funções implementadas nos exercícios 1 e 2.

4.9 (a) Primeiro escreva no caderno um pseudo-código para o algoritmo que recebe como entrada um vetor v e retorna a sua Distância Interquartílica. Não é para usar o computador ainda. Considere que você já sabe calcular os quartis de v.

(b) Agora no computador implemente o passo-a-passo feito do item 9a usando as funções implementadas no exercício 5.

4.10 Faça o que se pede sem usar as funções `sd`, `mean` e `sum` do R. No computador implemente o algoritmo visto em sala de aula que recebe como entrada um vetor v e retorna a sua variância amostral. Para simplificar use a função implementada no exercício 3.

OBS: Se quiser use a função `sd` para comparação.

4.11 (a) Primeiro escreva no caderno um pseudo-código para o algoritmo que recebe como entrada um vetor v e retorna o seu Desvio Médio. Não é para usar o computador ainda. Considere que você já sabe calcular a média de v.

(b) Agora no computador implemente o pseudo-código feito do item 11a.

4.12 Faça o que se pede sem usar as funções `mean` e `sum` do R.

(a) Implemente o algoritmo visto em sala de aula que recebe como entrada dois vetores v e w e retorna a covariância amostral entre eles. Para simplificar use a função implementada no exercício 3.

(b) Desafio: vamos generalizar o item 12a. Implemente uma função que recebe como entrada uma matriz de dados. Considere que cada coluna dessa matriz representa um vetor de dados. A função implementada deve retornar a matriz de covariância amostral entre os vetores coluna dessa matriz.

Lembre-se que a matriz de covariância é definida pela matriz cuja posição (i, j) guarda a covariância da variável i com a j.

Semana 5: Algoritmos para Cálculos Matriciais

A motivação dese capítulo é implementar funções que realizam cálculos matriciais. Assim como no capítulo anterior, na aula teórica será visto os pseudo-códigos e estes serão implementados durantes as aulas práticas, através dos exercícios propostos ao final do capítulo.

Primeiro serão apresentadas algumas operações entre vetores. Em seguida será visto operações também envolvendo matrizes.

5.1 Multiplicação de vetor por escalar

Definição 5.1.1 *Sejam $v = (v_1, v_2, \ldots, v_n) \in \mathbb{R}^n$ e $\alpha \in \mathbb{R}$. A multiplicação do vetor v pelo escalar α é o vetor $w = \alpha v$ definido por $w = (\alpha v_1, \alpha v_2, \ldots, \alpha v_n) \in \mathbb{R}^n$.*

Queremos escrever um pseudo-código que recebe como entrada um vetor v e um escalar α e retorna o vetor definido por $w = \alpha v$. Para isso basta percorrer o vetor v multiplicando cada posição pelo escalar α. Veja como isso pode ser feito no pseudo-código a seguir.

Entradas: v = vetor de números reais; α = número real
Saída: o vetor $w = \alpha v$

1 n = tamanho do vetor v;
2 Inicie o vetor w como nulo;
3 Inicie $i = 1$;
4 Faça $w_i = \alpha v_i$;
5 Incremente $i = i + 1$;
6 Se $i \leq n$, volte para a linha 4;
7 Retorne w;

5.2 Soma entre vetores

Definição 5.2.1 *Sejam $v = (v_1, v_2, \ldots, v_n) \in \mathbb{R}^n$ e $u = (u_1, u_2, \ldots, u_n) \in \mathbb{R}^n$. A soma entre os vetores v e u é o vetor $w = v + u$ definido por $w = (v_1 + u_1, v_2 + u_2, \ldots, v_n + u_n) \in \mathbb{R}^n$.*

Queremos escrever um pseudo-código que recebe como entrada dois vetores v e u e retorna o vetor definido pela soma entre eles. Veja que para essa operação ser realizada é

preciso que ambos os vetores tenham mesma dimensão. Para encontrar a soma entre os dois vetores de mesma dimensão basta somar cada posição uma a uma. Veja o pseudo-código a seguir.

Entradas: v e u, vetores que serão somados.
Saída: o vetor $w = v + u$.

1 n = tamanho do vetor v;
2 m = tamanho do vetor u;
3 Se $n \neq m$, retorne uma mensagem de erro e FIM;
4 Inicie o vetor w como nulo;
5 Inicie $i = 1$;
6 Faça $w_i = v_i + u_i$;
7 Incremente $i = i + 1$;
8 Se $i \leq n$, volte para a linha 6;
9 Retorna w.

5.3 Subtração entre vetores

Definição 5.3.1 *Sejam $v = (v_1, v_2, \ldots, v_n) \in \mathbb{R}^n$ e $u = (u_1, u_2, \ldots, u_n) \in \mathbb{R}^n$. A substração do vetor v por u é o vetor $w = v - u$ definido por $w = (v_1 - u_1, v_2 - u_2, \ldots, v_n - u_n) \in \mathbb{R}^n$.*

O pseudo-código que recebe como entrada dois vetores e retorna o vetor definido pela subtração entre eles pode ser feito de duas maneiras. A primeira alternativa é fazer um pseudo-código análogo ao definido para a soma entre dois vetores. A segunda alternativa é combinar a multiplicação de um vetor por um escalar com a soma entre dois vetores: primeiro multiplica-se o vetor u pelo escalar -1 e em seguida soma o resultado com o vetor v.

Faça você mesmo esse pseudo-código para a subtração entre dois vetores da maneira que achar melhor.

5.4 Produto interno

Definição 5.4.1 *Sejam $v = (v_1, v_2, \ldots, v_n) \in \mathbb{R}^n$ e $u = (u_1, u_2, \ldots, u_n) \in \mathbb{R}^n$. O produto interno entre v e u é o número real definido por:*
$< v, u >= v_1u_1 + v_2u_2 + \ldots + v_nu_n = \sum_{i=1}^{n} v_iu_i$.

Queremos escrever o pseudo-código que recebe como entrada dois vetores v e u e retorna o produto interno entre eles. Veja que essa operação só é possível se ambos os vetores tiverem mesma dimensão. A ideia é percorrer as posições dos vetores, multiplicando posição a posição e guardando a soma em uma variável local. Veja o pseudo-código a seguir.

Entrada: v e u, vetores para os quais queremos o produto interno
Saída: o produto interno entre v e u: $< v, u >$.

1 Defina n = tamanho do vetor v;
2 Defina m = tamanho do vetor w;
3 Se $n \neq m$, retorne uma mensagem de erro e FIM;
4 Inicie $p = 0$;
5 Inicie $i = 1$;
6 Incremente $p = p + v_i w_i$;
7 Incremente $i = i + 1$;
8 Se $i \leq n$, volte para a linha 6;
9 Retorna p.

5.5 Multiplicação de matriz por escalar

Definição 5.5.1 *Sejam A uma matriz de dimensão $n \times m$ e $\alpha \in \mathbb{R}^n$ um escalar. A multiplicação de A pelo escalar α é a matriz $M = \alpha A$ de dimensão $n \times m$ onde cada posição (i, j) é definida por $M_{i,j} = \alpha A_{i,j}$.*

Queremos escrever um pseudo-código que recebe como entrada uma matriz A e um escalar α e retorna a matriz definida por $M = \alpha A$. Para isso basta percorrer a matriz A multiplicando cada posição pelo escalar α. Mas lembre-se que para percorrer uma matriz é preciso usar um *loop* dentro do outro. Veja como isso pode ser feito no pseudo-código a seguir.

Entradas: A = matriz de números reais; α = número real
Saída: a matriz $M = \alpha A$

1 n = número de linhas da matriz A;
2 m = número de colunas da matriz A;
3 Inicie uma matriz M de dimensão $n \times m$;
4 Inicie $i = 1$;
5 Inicie $j = 1$;
6 Faça $M_{i,j} = \alpha A_{i,j}$;
7 Incremente $j = j + 1$;
8 Se $j \leq m$, volte para a linha 6;
9 Incremente $i = i + 1$;
10 Se $i \leq n$, volte para a linha 5;
11 Retorne M;

Reparou que temos dois *loops*? O primeiro está entre as linhas 6 e 8 e o outro entre as linhas 5 e 10. Como isso pode ser implementado em uma linguagem de programação, em particular na linguagem R?

5.6 Soma entre matrizes

Definição 5.6.1 *Sejam A e B duas matrizes de dimensão $n \times m$. A soma das matrizes A e B é a matriz $M = A + B$ de dimensão $n \times m$ onde cada posição (i, j) é definida por $M_{i,j} = A_{i,j} + B_{i,j}$.*

Queremos escrever o pseudo-código que recebe como entrada duas matrizes A e B e retorna a matriz definida pela soma delas. Novamente como teremos que percorrer as linhas e colunas das matrizes será preciso usar dois *loops*, um dentro de outro. Veja no pseudo-código a seguir como isso pode ser feito.

Entrada: A e B, matrizes que serão somadas
Saída: matriz $M = A + B$.

1 Defina $n =$ número de linhas de A;
2 Defina $m =$ número de colunas de A;
3 Defina $l =$ número de linhas de B;
4 Defina $c =$ número de colunas de B;
5 Se $n \neq l$, retorne uma mensagem de erro e FIM.
6 Se $m \neq c$, retorne uma mensagem de erro e FIM.
7 Crie uma matriz M de dimensão $n \times m$;
8 Inicie $i = 1$;
9 Inicie $j = 1$;
10 Preencha a posição (i, j) de M: $M_{i,j} = A_{i,j} + B_{i,j}$;
11 Incremente $j = j + 1$;
12 Se $j \leq m$, volta para a linha 10;
13 Incremente $i = i + 1$;
14 Se $i \leq n$, volta para a linha 9;
15 Retorna M.

5.7 Subtração entre Matrizes

Definição 5.7.1 *Sejam A e B duas matrizes de dimensão $n \times m$. A subtração da matriz A pela matriz B é a matriz $M = A - B$ de dimensão $n \times m$ onde cada posição (i, j) é definida por $M_{i,j} = A_{i,j} - B_{i,j}$.*

Assim como a subtração de vetores, o pseudo-código que recebe como entrada duas matrizes e retorna a matriz definida pela subtração entre elas pode ser feito de duas maneiras. A primeira alternativa é fazer um pseudo-código análogo ao definido para a soma entre matrizes. A segunda alternativa é combinar a multiplicação de uma matriz por um escalar com a soma entre duas matrizes: primeiro multiplica-se a matriz B pelo escalar -1 e em seguida soma o resultado com a matriz A.

Faça você mesmo o pseudo-código para a subtração entre duas matrizes da maneira que achar melhor.

5.8 Transposição de Matrizes

Definição 5.8.1 *Seja A uma matriz $n \times m$. A transposta da matriz A é a matriz A^T de dimensão $m \times n$ onde cada posição (i, j) é definida por $A^T_{i,j} = A_{j,i}$.*

Queremos escrever um pseudo-código que recebe como entrada uma matriz A e retorna a sua transposta. Veja que só precisamos percorrer as linhas e colunas da matriz e guardar na posição (i, j) o elemento guardando na posição (j, i) da matriz de entrada. O pseudo-código a seguir realiza essa tarefa.

Entrada: A, matriz a ser transposta
Saída: matriz A^T

1 Defina n = número de linhas da matriz A;
2 Defina m = número de colunas da matriz A;
3 Crie uma matriz M com dimensão $m \times n$;
4 Inicie $i = 1$;
5 Inicie $j = 1$;
6 Preencha a posição (i, j) da matriz M: $M_{i,j} = A_{j,i}$;
7 Incremente $j = j + 1$;
8 Se $j \leq n$, volta para a linha 6;
9 Incremente $i = i + 1$;
10 Se $i \leq m$, volta para a linha 5;
11 Retorna M.

5.9 Multiplicação entre matriz e vetor

Definição 5.9.1 *Seja A uma matriz $n \times m$ e $v \in \mathbb{R}^m$ um vetor. Considere $a_i \in \mathbb{R}^m$ o vetor formado pela i-ésima linha da matriz A. O produto entre a matriz A e o vetor v é o vetor $w = Av \in \mathbb{R}^n$ tal que cada posição é definida por $w_i =< a_i, v >$.*

A partir da definição acima vamos escrever um pseudo-código para uma função que recebe como entrada uma matriz A e um vetor v e retorna o vetor definido pelo produto entre eles. Vamos considerar que já sabemos calcular o produto interno entre dois vetores. Então basta percorrer as colunas da matriz e calcular o produto interno entre os vetores colunas e o vetor v. Veja que as dimensões de A e v devem ser testadas para verificar se a multiplicação realmente pode ser realizada: a conta só é possível quando o número de colunas da matriz é igual ao número de elementos no vetor. O pseudo-código a seguir mostra como isso pode ser feito.

Entrada: uma matrizes A e um vetor v.
Saída: vetor $w = Av$.

1 Defina n = número de linhas de A;
2 Defina m = número de colunas de A;
3 Defina k = a dimensão do vetor v;

4 Se $k \neq m$, retorne uma mensagem de erro e FIM.
5 Inicie um vetor w como nulo;
6 Inicie $i = 1$;
7 Crie a_i como o vetor definido pela linha i da matriz A;
8 Faça $w_i =< a_i, v >$;
9 Incremente $i = i + 1$;
10 Se $i \leq n$, volta para a linha 7;
11 Retorne w.

Dica: Na linguagem R se o objeto `A` é uma matriz então o objeto `ai = A[,i]` é um *array* de `"numeric"` definido pela i-ésima coluna de `A`.

5.10 Multiplicação entre matrizes

Definição 5.10.1 *Seja A uma matriz $n \times m$ e B uma matriz $m \times k$. Considere $a_i \in \mathbb{R}^m$ o vetor formado pela i-ésima linha da matriz A e $b_j \in \mathbb{R}^m$ o vetor formado pela j-ésima coluna da matriz B. O produto entre a matriz A e a matriz B é a matriz $M = AB$ de dimensão $n \times k$ onde cada posição é definida por $M_{i,j} =< a_i, b_j >$.*

A partir da definição acima vamos escrever um pseudo-código para uma função que recebe como entrada duas matrizes A e B e retorna outra matriz definida pelo produto entre elas. Vamos considerar que já sabemos calcular o produto interno entre dois vetores. Veja que as dimensões de A e B devem ser testadas para verificar se a multiplicação realmente pode ser realizada: só podemos multiplicar A por B se o número de colunas de A for igual ao número de colunas de B.

Entrada: A e B, matrizes que serão multiplicadas
Saída: matriz $M = AB$.
1 Defina $n =$ número de linhas de A;
2 Defina $m =$ número de colunas de A;
3 Defina $l =$ número de linhas de B;
4 Defina $k =$ número de colunas de B;
5 Se $m \neq l$, retorne uma mensagem de erro e FIM.
6 Crie uma matriz M de dimensão $n \times k$;
7 Inicie $i = 1$;
8 Inicie $j = 1$;
9 Crie a_i como o vetor definido pela linha i da matriz A;
10 Crie b_j como o vetor definido pela coluna j da matriz B;
11 Faça $M_{i,j} =< a_i, b_j >$;
12 Incremente $j = j + 1$;
13 Se $j \leq k$, volta para a linha 10;
14 Incremente $i = i + 1$;
15 Se $i \leq n$, volta para a linha 8;
16 Retorne M.

Exercícios - 5ª Semana

5.1 Implemente uma função que recebe como entrada um vetor v e um escalar α e retorna o vetor αv.

5.2 Implemente uma função que recebe como entrada dois vetores v e u e retorna outro vetor definido pela soma entre eles.

5.3 (a) No caderno escreva um pseudo-código para o algoritmo que recebe como entrada dois vetores v e u e retorna outro vetor definido pela subtração do primeiro pelo segundo.

(b) Agora no computador implemente o pseudo-código elaborado no item acima.

5.4 Implemente uma função que recebe como entrada dois vetores v e u e retorna o produto interno entre eles.

5.5 Implemente uma função que recebe como entrada dois vetores e retorna TRUE caso eles forem ortogonais e FALSE caso contrário.

Dica: dá para saber se dois vetores são ortogonais a partir do produto interno entre eles.

5.6 Implemente uma função que recebe como entrada uma matriz A e um escalar α e retorna a matriz αA.

5.7 Implemente uma função que recebe como entrada duas matrizes A e B e retorna outra matriz definida pela soma entre elas.

5.8 (a) No caderno escreva um pseudo-código para o algoritmo que recebe como entrada duas matrizes A e B e retorna outra matriz definida pela subtração da primeira pela segunda.

(b) Agora no computador implemente o pseudo-código elaborado no item acima.

5.9 Implemente uma função que recebe como entrada uma matriz A e retorna a sua transposta.

5.10 (a) No caderno escreva um pseudo-código para o algoritmo que recebe como entrada uma matriz A e retorna $TRUE$ se essa matriz for simétrica e $FALSE$ caso contrário. Lembre-se: uma matriz simétrica é uma matriz quadrada tal que $A_{i,j} = A_{j,i}$.

(b) Agora no computador implemente o pseudo-código elaborado no item acima.

5.11 Implemente uma função que recebe como entrada uma matriz A e um vetor v e retorna o vetor definido por Av.

5.12 Implemente uma função que recebe como entrada duas matrizes A e B e retorna a matriz definida pelo produto entre A e B.

5.13 Considere $\alpha = 4$, $\beta = -3$, $v_1 = (2, -3, -1, 5, 0, -2)$, $v_2 = (3, 4, -1, 0, 1, 1)$, $v_3 = (1, 2, 3, 4, 5)$, $v_4 = (0, 1, 1)$, $M_1 = \begin{pmatrix} 1 & 3 & 2 \\ -1 & 0 & 1 \end{pmatrix}$, $M_2 = \begin{pmatrix} 0 & -5 & 3 \\ -1 & 1 & -1 \\ 1 & 4 & 0 \end{pmatrix}$, $M_3 = \begin{pmatrix} 3 & 1 \\ -2 & 10 \\ 3 & -1 \end{pmatrix}$, $M_4 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$ e $M_5 = \begin{pmatrix} 3 & 1 & 0 & 1 \\ 1 & 1 & 3 & 2 \\ 0 & 3 & -5 & 0 \\ 1 & 2 & 0 & 0 \end{pmatrix}$.

Usando as funções que você implementou nos exercícios anteriores faça as contas que se pede. Tente cada item usando apenas uma linha de comando no R.

(a) αv_3

(b) $v_1 + v_2$

(c) $v_3 - v_1$

(d) $< v_1, v_2 >$

(e) $< \alpha v_1, v_2 - v_1 >$

(f) $< v_1 + v_2, v_1 - v_2 >$

(g) Os vetores v_1 e v_2 acima são perpendiculares (ortogonais)?

(h) βM_1

(i) M_1^T

(j) Verifique se as matrizes M_1, M_4 e M_5 são simétricas.

(k) $M_1 v_4$

(l) $M_2 v_4 + v_4$

(m) $M_1 M_2$

(n) $M_2 M_1$

(o) $M_3^T M_2$

(p) $(M_1 M_3) + M_4$

(q) $M_1 M_2 M_3$

(r) $M_1 M_2 M_3 - M_4$

Parte II

Recursão

Semana 6: Algoritmos Recursivos

Na ciência da computação um algoritmo é dito recursivo quando ele chama a si próprio com entradas mais simples. De forma semelhante, dizemos que uma função é implementada de forma recursiva quando ela chama a si própria com argumentos mais simples. Vejamos alguns exemplos de algoritmos e funções implementadas de forma recursiva no R nas seções a seguir.

Como será mostrado nesse capítulo, para todo algoritmo recursivo existe outro que executa a mesma tarefa de forma não recursiva. A vantagem em usar os algoritmos recursivos é a maior simplicidade e clareza no código (confira!). A desvantagem é que em geral tais algoritmos consomem muita memória, devido ao uso intensivo de pilha.

6.1 Fatorial

O exemplo mais clássico em recursão é o cálculo do fatorial de n. Podemos fazer isso de forma recursiva ou não. Vejamos primeiro o algoritmo sem recursão.

Entrada: n
Saída: $n!$

1 Se n não for inteiro positivo, retorne erro.
2 Inicie $saida = 1$;
3 Se $n = 0$, retorne $saida$;
4 Inicie $i = 1$;
5 Faça $saida = saida * i$;
6 Incremente $i = i + 1$;
7 Se $i \leq n$, volte para a linha 5;
8 Retorne $saida$.

No caso do algoritmo recursivo é importante definir o seu nome para que fique clara a chamada recursiva. Veja agora o algoritmo recursivo.

Entrada: n
Saída: $n!$
Nome: Fatorial

1 Se n não for inteiro positivo, retorne erro.
2 Se $n = 0$, retorne 1;
3 Retorne $n \times$ Fatorial$(n - 1)$.

Veja que a chamada recursiva aparece na linha 3. A função Fatorial, que está sendo definida para uma entrada n, é chamada para a entrada $n-1$. Ou seja, o algoritmo chama a si próprio com entrada simplificada. Dessa forma garantimos que em algum momento a função Fatorial será chamada para $n = 0$, argumento para o qual temos resposta imediata.

Vamos comparar agora as respectivas implementações no R.

Sem recursão:

```
> fatorial_1 <- function(n){
+   if(n%%1 != 0 || n<0)
+     stop("ERRO")
+   saida = 1
+   for(i in 1:n){
+     saida = saida*i
+   }
+   return(saida)
+ }
```

Com recursão:

```
> fatorial_2 <- function(n){
+   if(n%%1 != 0 || n<0)
+     stop("ERRO")
+   if(n==0)
+     return(1)
+   return(n*fatorial_2(n-1))
+ }
```

A função implementada à esquerda é aquela semelhante às que já foram feitas nas aulas práticas. Já na segunda aparece a recursão.

6.2 Vetores

Também podemos usar a recursão para trabalhar com vetores. O exemplo a seguir mostra como encontrar o maior elemento em um vetor usando ou não recursão. Veja primeiro o pseudo-código sem considerar recursão, como já fizemos anteriormente.

Entrada: v
Saída: Maior elemento de v

1 Defina $n =$ tamanho do vetor v;
2 Se $n = 1$ retorne v_1;
3 Inicie $max = v_1$;
4 Inicie $i = 2$;
5 Se $v_i > max$ faça $max = v_i$; ;
6 Incremente $i = i + 1$;
7 Se $i \leq n$, volte para a linha 5;
8 Retorne max.

Agora veja o algoritmo recursivo.

Entrada: v
Saída: Maior elemento de v
Nome: Max

1 Defina n = tamanho do vetor v;
2 Se $n = 1$, retorne v_1;
3 Defina $w = (v_2, v_3, ..., v_n)$;
4 Defina $maxw$ =Max(w);
5 Se $v_1 > maxw$, retorne v_1;
6 Retorne $maxw$.

Veja que a chamada recursiva aparece na linha 4. Nesse exemplo o argumento foi simplificado uma vez que vetor passado como argumento tem uma dimensão a menos. Assim garantimos que em algum momento a função sera chamada para um vetor de tamanho 1, entrada para a qual temos uma resposta imediata.

6.3 Sequências Definidas a partir de Equações de Diferenças

Uma sequência de números reais $\{x_n\}_{n=1}^{\infty}$ é definida a partir de uma equação de diferenças quando o seu n-ésimo termo é escrito em função de termos anteriores. Ou seja, quando existe f tal que $x_n = f(n, x_{n-1}, x_{n-2}, \ldots)$.

Um exemplo de equação de diferenças é a sequência de Fibonacci, que já foi vista em capítulos anteriores. Esse exemplo será revisto na seção 6.3.3. Antes disso veremos alguns outros exemplos.

Encontrar a solução de uma equação de diferenças consiste em determinar o n-ésimo termo da sequência de forma direta, sem depender dos termos anteriores, dependendo apenas de n. Para isso existem muitas técnicas, mas aprender essas técnicas não é o foco do nosso curso.

Por exemplo, seja $\{x_n\}_{n=1}^{\infty}$ uma sequência definida por:

$$x_0 = 1 \text{ e } x_n = 2x_{n-1} + 1.$$

Com essas informações podemos calcular todos os termos de forma iterativa:

$$\begin{aligned} x_0 &= 1, \\ x_1 &= 2 \times 1 + 1 = 3, \\ x_2 &= 2 \times 3 + 1 = 7, \\ x_3 &= 2 \times 7 + 1 = 15, \ldots \end{aligned}$$

A solução dessa equação de diferenças é $x_n = 2^n - 1$ (verifique!). Mas nesse curso não vamos aprender como chegar nessa solução, nosso trabalho será montar a equação e resolver o problema de forma iterativa a partir de uma função implementada no R. Faremos isso usando e não usando recursão.

Vejamos a seguir alguns problemas que utilizam equações de diferenças em sua modelagem.

6.3.1 Padrões geométricos

Veja o padrão geométrico a seguir e pense em uma equação de diferenças que expresse o número de bolinhas em cada grupo. Ou seja, considere x_n o número de bolinhas no grupo e pense em como podemos escrever x_n em função de x_{n-1} e n.

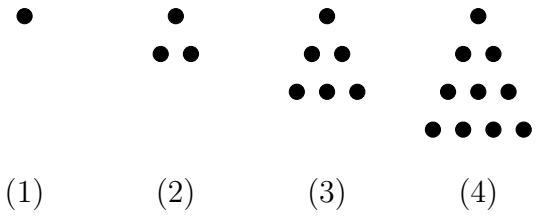


Figura 6.1: Padrão Geométrico Triangular

Pela figura temos $x_1 = 1$, $x_2 = 3$, $x_3 = 6$ e $x_4 = 10$. Você identificou algum padrão nessa formação? Veja que $x_n = x_{n-1} + n$.

Agora vamos escrever um pseudo-código que recebe como argumento um número n e retorna o número de bolinhas no n-ésimo grupo, ou seja, x_n. Primeiro de forma não recursiva.

Entrada: n
Saída: x_n

1 Se n não for inteiro positivo, retorne erro.
2 Se $n = 1$ retorne 1;
3 Defina $x = 1$;
4 Faça $i = 2$;
5 Faça $x = x + i$;
6 Incremente $i = i + 1$;
7 Se $i \leq n$, volte para a linha 5;
8 Retorne x.

Agora o pseudo-código usando recursão.

Entrada: n
Saída: x_n
Nome: NumBolinhas

1 Se n não for inteiro positivo, retorne erro.
2 Se $n = 1$ retorne 1;
3 Retorne NumBolinhas$(n - 1) + n$

Veja que novamente a chamada recursiva usou um argumento simplificado: trocou n por $n - 1$.

6.3.2 Matemática Financeira

Em Matemática Financeira muitos problemas que envolvem juros, como investimentos ou dívidas, podem ser expressos em termos de sequências de diferenças. Veja dois exemplos a seguir.

Suponha que você investiu R$ 1.000,00 em um fundo de investimento que paga $0,7\%$ de rendimento ao mês. Considere x_n como sendo o total acumulado após n meses de

investimento. Primeiro veja que o total acumulado após n meses depende diretamente do total acumulado após $n-1$ meses:

$$x_0 = 1.000,00$$
$$x_n = x_{n-1} + 0,007 \times x_{n-1} = 1,007 \times x_{n-1}$$

Essa é uma equação de diferenças homogênea de primeira ordem com coeficiente constante, ou seja, ela é do tipo: $x_n = cx_{n-1}$ com $c \in \mathbb{R}$. Toda equação desse tipo tem uma solução fácil: $x_n = c^n x_0$ (verifique!). Mas não é isso que estamos buscando. Nesse capítulo queremos treinar a implementação de algoritmos usando recursão. Vamos então escrever um código que recebe como entrada n e retorna x_n, onde cada x_i é calculado de forma iterativa, como se não soubéssemos a solução da equação de diferenças.

Veja primeiro o pseudo-código sem usar recursão.

Entrada: n
Saída: x_n

1 Se n não for inteiro não negativo, retorne erro.
2 Se $n = 0$ retorne 1.000;
3 Defina $x = 1.000$;
4 Faça $i = 1$;
5 Faça $x = 1,007 \times x$;
6 Incremente $i = i + 1$;
7 Se $i \leq n$, volte para a linha 5;
8 Retorne x.

Agora o usando recursão.

Entrada: n
Saída: x_n
Nome: Investimento

1 Se n não for inteiro não negativo, retorne erro.
2 Se $n = 0$ retorne 1.000;
3 Retorne $1,007 \times$Investimento$(n-1)$

Vejamos agora outro exemplo envolvendo juros: um financiamento.

Suponha que você vai pegar emprestado R$1.000,00 no banco e deverá pagar esse valor corrigido por um juros composto de 1,5% ao mês. Ou seja, a cada mês sua dívida cresce 1,5%. Suponha que você está disposto a pagar parcelas mensais de R$ 200,00.

Considere y_n como sendo sua dívida após n meses do início do financiamento. Primeiro veja que a dívida após n meses depende diretamente da dívida após $n-1$ meses:

$$y_0 = 1.000,00$$
$$y_n = y_{n-1} + 0,015 \times y_{n-1} - 200 = 1,015 \times y_{n-1} - 200$$

Com essas informações podemos escrever um pseudo-código que fornece a dívida após n meses. Só temos que tomar cuidado que essa sequência após um número finito de passos

passa a ser negativa, caso contrário a dívida não seria paga nunca. Nesse caso, apesar de $y_n < 0$ a real dívida não existe mais, então a função deve retornar 0. Veja primeiro o pseudo-código sem usar recursão.

Entrada: n
Saída: y_n

1 Se n não for inteiro não negativo, retorne erro.
2 Se $n = 0$ retorne 1.000;
3 Defina $y = 1.000$;
4 Faça $i = 1$;
5 Faça $y = 1,015 \times y - 200$;
6 Incremente $i = i + 1$;
7 Se $i \leq n$, volte para a linha 5;
8 Se $y < 0$, retorne 0. Senão, retorne y.

Agora o usando recursão.

Entrada: n
Saída: y_n
Nome: Dívida

1 Se n não for inteiro não negativo, retorne erro.
2 Se $n = 0$ retorne 1.000;
3 Faça $d = 1,015 \times \text{Dívida}(n-1) - 200$;
4 Se $d < 0$, retorne 0. Senão, retorne d.

Outra informação que pode ser de interessante é o tempo que a dívida vai demorar para ser paga. Como podemos achar isso? Pense em um pseudo-código que recebe como entrada o valor das parcelas fixas que serão pagas por mês e retorna o número de meses que a pessoa demora para pagar a dívida de R$1.000,00 considerando um juros composto de 1,5% ao mês.

6.3.3 Fibonacci

A sequência de Fibonacci é um caso particular de equações de diferenças de segunda ordem, isto é, uma equação em que x_n depende não só de x_{n-1} como também de x_{n-2}. Lembrando, uma sequencia de Fibonacci é definida por:

$$\begin{aligned} F_1 &= 1, \\ F_2 &= 1 \\ F_n &= F_{n-1} + F_{n-2}. \end{aligned}$$

Vamos escrever um pseudo-código que recebe como entrada n e retorna o n-ésimo termo da sequência de Fibonacci (F_n). Primeiro sem recursão.

Entrada: n
Saída: F_n

1 Se $n = 1$ ou $n = 2$, retorne 1
2 Defina $F1 = 1$ e $F2 = 1$
3 Inicie $i = 3$
4 Faça $F = F1 + F2$
5 Faça $F1 = F$, $F2 = F1$
6 Incremente $i = i + 1$
7 Se $i \leq n$, volte para a linha 4
8 Retorne F

Agora com recursão.

Entrada: n
Saída: F_n
Nome: Fibonacci

1 Se $n = 1$ ou $n = 2$, retorne 1
2 Retorne Fibonacci$(n-1)$ + Fibonacci$(n-2)$

Veja que nesse exemplo, por se tratar de uma equação de diferenças de segunda ordem, a chamada recursiva é feita duas vezes: Fibonacci$(n-1)$ e Fibonacci$(n-2)$. Em ambas o argumento foi simplificado. Como temos dois casos básicos, $n = 1$ e $n = 2$, garantimos que para qualquer natural n sempre teremos solução.

Exercícios - 6ª Semana

6.1 Implemente de forma recursiva uma função que recebe como entrada um número natural n e retorna $n!$. Não esqueça de verificar se o argumento passado como entrada é realmente um número natural.

6.2 Implemente de forma recursiva uma função que recebe como entrada um vetor v e retorna o valor máximo desse vetor.

6.3 (a) No caderno escreva um pseudo-código recursivo para o algoritmo que recebe como entrada um vetor v e retorna a soma dos elementos desse vetor.

(b) Agora no computador implemente o pseudo-código elaborado acima.

6.4 (a) No caderno escreva um pseudo-código recursivo para o algoritmo que recebe como entrada um vetor v e retorna a posição onde se encontra o valor máximo desse vetor. Dica: em vez de definir $w = (v_2, v_3, \ldots, v_n)$ defina $w = (v_1, v_2, \ldots, v_{n-1})$. Mas cuidado que isso muda um pouco a forma de pensar.

(b) Agora no computador implemente o pseudo-código elaborado acima.

6.5 Considere o seguinte padrão geométrico.

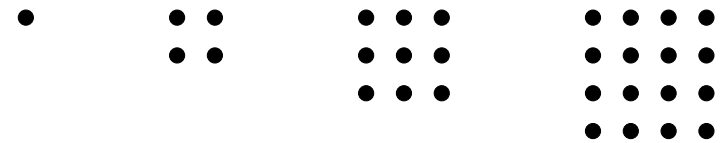


Figura 6.2: Padrão Geométrico Quadrado

Defina x_n como o número de bolinhas no grupo n.

(a) Encontre uma fórmula recursiva para x_n, ou seja, escreva x_n como função de x_{n-1}.

(b) Usando a fórmula encontrada, no caderno escreva um pseudo-código recursivo para o algoritmo que recebe como entrada um número natural n e retorna o número de bolinhas no grupo n.

(c) Agora no computador implemente o pseudo-código elaborado acima.

6.6 Suponha que você vá investir R$ $500,00$ na poupança e que esta rende $7,5\%$ ao ano.

(a) Calcule na mão o quanto de dinheiro você teria no banco depois de 1, 2 e 3 anos de investimento.

(b) Tente achar uma equação que relacione o total de dinheiro acumulado em n anos de investimento com o total de dinheiro acumulado em $n-1$ anos.

(c) Usando a equação encontrada, no caderno escreva um pseudo-código recursivo para o algoritmo que recebe como entrada um número natural n e retorna o dinheiro acumulado em n anos nesse investimento.

(d) Agora no computador implemente o pseudo-código elaborado acima.

6.7 Vamos generalizar o exercício anterior.

(a) Seja I o valor investido em uma aplicação de rentabilidade $j\%$ ao ano. Implemente uma função que recebe como entrada I, j e n e retorna o total acumulado nessa aplicação após n anos.

(b) Use a função implementada para descobrir quanto de dinheiro teríamos a mais se investíssemos R$ 1.000,00 durante 2 anos em um fundo que rendesse 10% ao ano em vez de $7,5\%$.

6.8 Suponha que você vai fazer um financiamento de R$ 1.200,00 e vai pagar juros compostos de 2% ao mês. Considere que você pode pagar R$150,00 por mês.

(a) Calcule na mão o valor da sua dívida depois de 1, 2 e 3 meses.

(b) Tente achar uma equação que relacione a sua dívida no mês n com a sua dívida no mês $n-1$.

(c) Usando a equação encontrada escreva no caderno um pseudo-código recursivo para o algoritmo que recebe como entrada um número natural n e retorna a sua dívida após n meses do início do financiamento. Não se esqueça de considerar o caso em que a dívida foi paga, nesse caso você deve retornar 0.

(d) Agora no computador implemente o pseudo-código elaborado acima.

6.9 Vamos generalizar o exercício anterior.

(a) Seja F o valor financiado a juros compostos de $j\%$ ao mês. Considere K o valor das parcelas fixas que serão pagas todo mês. Implemente uma função recursiva que recebe como entrada F, j, K e n e retorna a dívida existente após n meses desde o início do financiamento.

(b) Use a função implementada acima para comparar o valor da sua dívida de R$ 1.200,00 após 10 meses nos seguintes casos: parcela mensal de R$ 150,00 e parcela mensal de R$ 120,00. Considere os mesmos 2% de juros compostos ao mês.

6.10 Vamos fazer outro exercício que considera um financiamento, mas agora estamos interessados na quantidade de meses para se pagar a dívida. Suponha que você vai fazer um financiamento de F rais e vai pagar juros compostos de $j\%$ ao mês. Considere que você pode pagar K reais por mês.

(a) No caderno escreva um pseudo-código recursivo para o algoritmo que recebe como entrada F, j e K e retorna o número de meses que você vai demorar para pagar a sua dívida.
Dica: A simplificação na chamada recursiva ocorre na entrada F.

(b) Agora no computador implemente o pseudo-código elaborado acima.

(c) Use a função implementada acima para comparar o número de meses até a quitação da dívida de R$ 1.200,00 nos seguintes casos: parcela mensal de R$ 150,00; parcela mensal de R$ 120,00 e parcela mensal de R$ 200,00. Considere os mesmos 2% de juros compostos ao mês.

6.11 Implemente uma função recursiva que recebe como entrada um número natural n e retorna o n-ésimo termo da sequência de Fibonacci.

6.12 Considere a seguinte sequência definida a partir de uma equação de diferenças de segunda ordem.

$$y_n = 2y_{n-1} + y_{n-2} + n \quad \text{com} \quad y_1 = 0 \text{ e } y_2 = 0$$

(a) No caderno escreva um pseudo-código recursivo para o algoritmo que recebe como entrada um número natural n e retorna o valor de y_n.

(b) Agora no computador implemente o pseudo-código elaborado acima.

Semana 7: Algoritmos Recursivos (continuação)

Nessa semana veremos mais alguns exemplos de algoritmos recursivos. Diferente do capítulo anterior, nesse será apenas apresentado o algoritmos recursivo. Mas vale lembrar que sempre existe um outro algoritmo não recursivo que realiza a mesma tarefa.

7.1 Séries

Seja $\{x_i\}_{i=0}^{\infty}$ uma sequência de número reais. Defina outra sequência $\{S_n\}_{n=0}^{\infty}$ como a soma dos n primeiros termos dessa sequência: $S_n = \sum_{i=0}^{n} x_i$. A sequência $\{S_n\}_{n=0}^{\infty}$ é chamada de série.

Nessa seção vamos usar a recursão para encontrar os termos de uma séria $S_n = \sum_{i=0}^{n} x_i$ dada a sequência $\{x_i\}$. Ou seja, para uma dada sequência $\{x_i\}$ e um natural n estamos interessados em encontrar a soma dos n primeiros termos dessa sequência.

Como primeiro exemplo considere $x_i = i$ e $S_n = \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} i$. Veja que para esse exemplo $S_n = 1 + 2 + \ldots + n$. Vamos escrever um algoritmo que recebe como entrada n e retorna o n-ésimo termo dessa série, isto é, S_n.

Entrada: n
Saída: $S_n = \sum_{i=0}^{n} i$
Nome: Serie1

1 Se n não for inteiro não negativo, retorne erro.
2 Se $n = 0$, retorne 0;
3 Retorne n+ Série1$(n-1)$.

Veja que a chamada recursiva usa como argumento $n - 1$. Dessa forma garantimos a simplificação do argumento e que em algum momento chegaremos ao caso mais simples e o algoritmo termina.

Vejamos agora outro exemplo. Considere $x_i = \frac{1}{i!}$ e defina $S_n = \sum_{i=0}^{n} x_i = \sum_{i=0}^{n} \frac{1}{i!}$. Podemos criar um algoritmo recursivo e usar o computador para encontrar qualquer termo n dessa série, como mostra o pseudo-código a seguir.

Entrada: n
Saída: $S_n = \sum_{i=1}^{n} \frac{1}{i}$;
Nome: Série2

```
1 Se n não for inteiro não negativo, retorne erro.
2 Se n = 0, retorne 1;
3 Retorne Série2(n − 1) + 1/n!.
```

Veja que no código acima consideram que já sabemos calcular $n!$. Mas é verdade, já fizemos isso na seção 6.1 acima. Então quando formos implementa esse último código no computador vamos chamar a função feita anteriormente que calcula $n!$ também de forma recursiva.

Já vimos em algumas disciplinas que a sequencia S_n definida acima converge para o número irracional $e = 2.718282\ldots$, ou seja, $S_n \xrightarrow[n\to\infty]{} e$. Se ainda não vimos veremos em breve. Então para n razoavelmente grande esperamos que S_n esteja próximo de $e = 2.718282\ldots$, como indica o limite. Com essa ideia podemos usar o algoritmo acima para encontrar uma aproximação para o número irracional e, basta chamar a função com valores grandes de n.

7.2 Maior Divisor Comum

Um algoritmo recursivo bem famoso é o Algoritmo de Euclides, conhecido desde a obra *Elements* de Euclides, por volta de 300 a.c.. Este algoritmo calcula o maior divisor comum entre dois números sem precisar fatorá-los. Lembrando, o maior divisor comum (MDC) entre dois números inteiros é o maior número inteiro que divide ambos sem deixar resto.

O algoritmo de Euclides é baseado no princípio de que o MDC não muda se o maior número for subtraído do menor. Por exemplo, o MDC entre 252 e 105 é 21. Veja que $252 = 21 \times 12$ e $105 = 21 \times 5$. Também é verdade que o MDC entre 252 - 105 = 147 e 105 é 21. Veja que $147 = 252 - 105 = 21 \times 12 - 21 \times 5 = 21 \times 7$.

A partir dessa ideia podemos afirmar que:

- $MDC(n, m) = MDC(m, n)$;
- se $n = 0$, $MDC(n, m) = m$;
- se $0 < n \leq m$, $MDC(n, m) = MDC(n, m - n) = MDC(m, m - n)$.

Podemos então criar uma função recursiva que encontra o MDC entre dois números inteiros quaisquer.

$$MDC(n, m) = \begin{cases} m, & \text{se } n = 0; \\ n, & \text{se } m = 0; \\ MDC(n, m - n) & \text{se } n \leq m; \\ MDC(m, n - m) & \text{se } m < n; \end{cases}$$

A partir da função acima podemos criar um pseudo-código recursivo que recebe como entrada dois números inteiros não negativos n e m e retorna o maior divisor comum entre eles.

Entradas: n e m, inteiros não negativos
Saída: maior divisor comum entre n e m
Nome: MDC

1 Se n ou m não forem inteiros não negativo, retorne erro.
2 Seja $maior$ = maior entre n e m;
3 Seja $menor$ = menor entre n e m;
4 Se $menor = 0$, retorne $maior$;
5 Retorne MDC($menor, maior - menor$) .

Veja que a chamada recursiva ocorre na linha 5 e que ambos os argumentos estão sendo simplificados. Até poderíamos fazer a chamada recursiva simplificando somente um argumento, por exemplo, substituindo a linha 5 por: Retorne MDC($m, maior - menor$) ou Retorne MDC($n, maior - menor$). Mas do jeito que está implementando no passo-a-passo a recursão vai precisar de menos passos para chegar no caso base onde uma das entradas é zero.

Podemos melhorar ainda mais o desempenho do nosso algoritmo. Veja que se m é muito maior que n nossa chamada recursiva sera MDC($n, m - n$) várias vezes. Isso vai terminar quando já tivermos "tirado" todos os n que cabem dentro do m. Ou seja, a última chamada recursiva desse tipo será MDC($n, m\%\%n$)! Então se substituirmos na linha 5 $maior - menor$ por $maior\%\%menor$ (resto da divisão de $maior$ por $menor$) vamos economizar muitos passos da recursão. Nesse caso o pseudo-código mais "enxuto" será:

Entradas: n e m, inteiros não negativos
Saída: maior divisor comum entre n e m
Nome: MDC

1 Se n ou m não forem inteiros não negativo, retorne erro.
2 Seja $maior$ = maior entre n e m;
3 Seja $menor$ = menor entre n e m;
4 Se $menor = 0$, retorne $maior$;
5 Retorne MDC($menor, maior\%\%menor$).

7.3 Torre de Hanoi

O problema ou quebra-cabeça conhecido como Torre de Hanói foi publicado em 1883 pelo matemático francês Edouard Lucas, também conhecido por seus estudos com a série Fibonacci.

O problema Consiste em transferir a torre composta por N discos, como na Figura 7.1, do pino A (origem) para o pino C (destino), utilizando o pino B como auxiliar. Somente um disco pode ser movimentado de cada vez e um disco não pode ser colocado sobre outro disco de menor diâmetro.

Veja que se existe um único dico, $N = 1$, a solução é imediata: mova este disco de A para C. E se existirem N discos, quais os movimentos que devemos fazer? É isso que vamos tentar resolver.

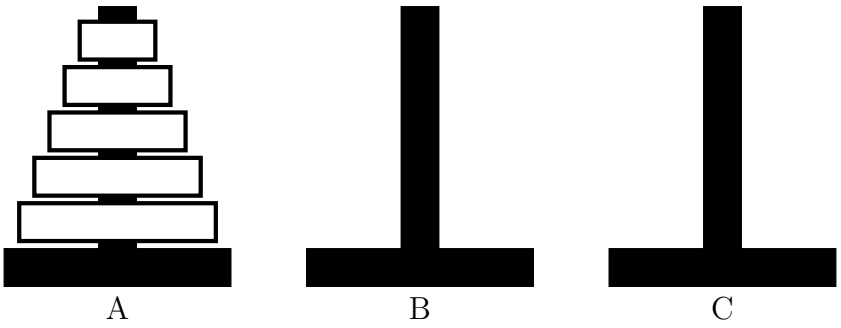
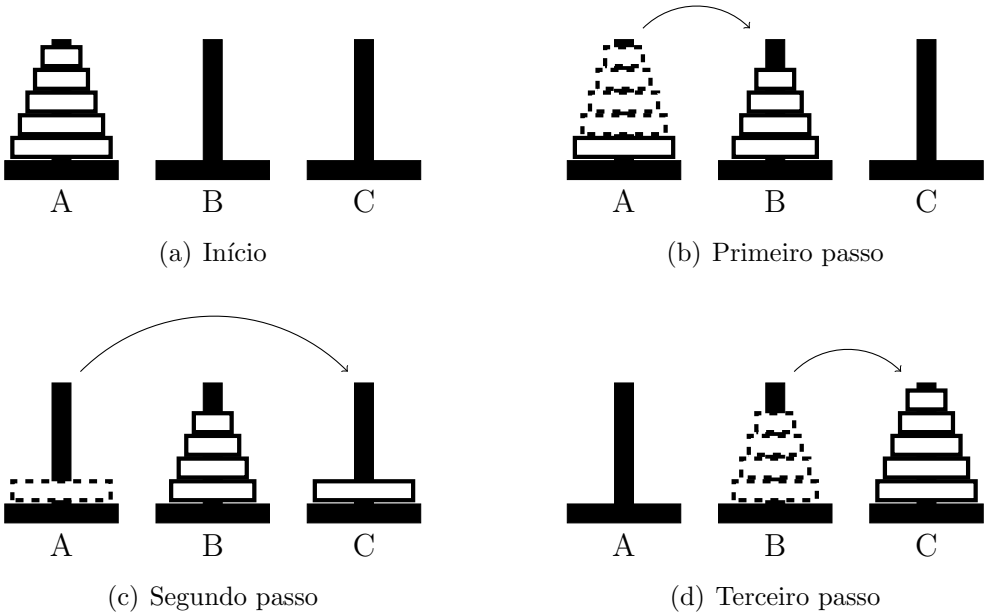


Figura 7.1: Torre de Hanoi

Suponha que sabemos a sequência de movimentos para mover $N-1$ discos de um pino origem para um pino destino. Então para mover os N discos do pino A para o pino C devemos primeiro mover os primeiros $N-1$ discos de A para B, veja figura 7.2(b). Em seguida mova o disco que sobrou no pino A para o pino C, veja figura 7.2(c). Para terminar basta mover novamente os $N-1$ discos, agora do pino B para o pino C, veja figura 7.2(d).



(a) Início (b) Primeiro passo

(c) Segundo passo (d) Terceiro passo

Figura 7.2: Torre de Hanoi - Esquema Recursivo

Dessa forma foi definida uma solução recursiva para o problema:

Se $N = 1$: mova um disco do pino A para o pino C.
Se $N > 1$:
- primeiro transfira $N-1$ discos de A para B (Figura 7.2(b));
- em seguida mova um disco do pino A para o pino C (Figura 7.2(c));
- por último transfira $N-1$ discos de B para C (Figura 7.2(d)).

Baseado nessa ideia vamos escrever o pseudo-código que fornece a sequência de movimentos para transferir N discos do pino A (origem) para o pino C (destino).

Entradas: N,A,C,B (número de discos, nome do pino origem, nome do pino destino e nome do pino auxiliar).
Saída: -
Nome: Hanoi

1 Se N não for inteiro positivo, retorne erro.
2 Se $N = 1$, imprima "mova um disco do pino A para o pino C" e fim.
3 Hanoi(N-1,A,B,C);
4 Imprima "mova um disco do pino A para o pino C"
5 Hanoi(N-1,B,C,A);
6 Fim.

Veja que o pseudo-código nada retorna, ele apenas imprime a sequência de movimentos que deve ser realizado. Veja que como o nome do pino que aparece na mensagem impressa é o argumento da função há a necessidade de concatenar os textos. No R podemos concatenar e imprimir textos usando o comando `cat`. Por exemplo, na linha 2 a impressão da mensagem no R pode ser feita assim: `cat("mova um disco do pino ",A," para o pino ",C)`.

Uma outra possibilidade é fazer com que a saída seja um *array* de objetos do tipo `"character"` tal que cada posição desse *array* guarda uma mensagem com um movimento. Veja como fica o pseudo-código com essa mudança.

Entradas: N,A,C,B (número de discos, nome do pino origem, nome do pino destino e nome do pino auxiliar).
Saída: Sequência de movimentos guardados em um array de textos.
Nome: Hanoi

1 Se N não for inteiro positivo, retorne erro.
2 Se $N = 1$, retorne "mova um disco do pino A para o pino C" e fim.
3 $mov1$ = Hanoi(N-1,A,B,C);
4 $mov2$ = "mova um disco do pino A para o pino C"
5 $mov3$ = Hanoi(N-1,B,C,A);
6 Retorne $(mov1, mov2, mov3)$.

Nesse caso vamos usar o comando `paste` para concatenar os textos, pois nesse caso não queremos imprimí-lo.

Repara que a primeira posição do *array* de saída guarda o texto com o primeiro movimento e a i-ésima posição do *array* de saída guarda o i-ésimo movimento. O tamanho do *array* de saída indica o número de movimentos necessários para mover os N discos.

Exercícios - 7ª Semana

7.1 Implemente de forma recursiva uma função que recebe como entrada um número natural n e retorna a soma de todos os naturais até n, isto é, retorna $S_n = \sum_{i=0}^{n} i = 0 + 1 + 2 + \ldots + n$.

7.2 (a) Implemente uma função que recebe como entrada um número natural n e retorna o n-ésimo termo da série $S_n = \sum_{i=0}^{n} \frac{1}{i!}$.

(b) Teste a função implementada para diferentes valores de n e veja se quando n cresce S_n se aproxima de $e = 2.718282...$

7.3 Considere a seguinte série: $S_n = \sum_{i=0}^{n} 4\frac{(-1)^i}{2i+1}$. Essa séria foi desenvolvida por Leibniz em 1682 e é conhecida para calcular aproximações para o número irracional π, uma vez que $S_n \xrightarrow[n\to\infty]{} \pi$.

(a) No caderno escreva um pseudo-código recursivo para o algoritmo que recebe como entrada um número natural n e retorna S_n de acordo com a fórmula acima.

(b) Agora no computador implemente o pseudo-código elaborado acima.

(c) Teste a função implementada para diferentes valores de n e veja se quando n cresce S_n se aproxima de $\pi = 3.141593...$

7.4 (a) No caderno escreva um pseudo-código recursivo para o algoritmo que recebe como entrada um número natural n e retorna a soma dos n primeiros termos da sequência de Fibonacci. Considere que você sabe encontrar a o termo n da sequência de Fibonacci.

(b) Agora no computador implemente o pseudo-código elaborado acima. Use a função que retorna o n-ésimo termos da sequência de Fibonacci implementada na semana passada.

7.5 Seja $\{x_i\}$ a sequência definida por $x_i = \frac{1}{3^i}$. Defina $S_n = \sum_{i=0}^{n} x_i$.

(a) No caderno escreva um pseudo-código recursivo para o algoritmo que recebe como entrada um número natural n e retorna S_n.

(b) Agora no computador implemente o pseudo-código elaborado acima.

(c) Para $a \leq b$ e $a, b \in \mathbb{N}$ defina $Soma(a, b) = x_a + x_{a+1} + \ldots + x_b$. Implemente no computador uma função que recebe como entrada a e b e retorna $Soma(a, b)$ de forma recursiva, sem chamar a função implementada no item 5b.
Dica: O caso base acontece quando $a = b$, nesse caso qual deve ser a saída de $Soma(a, b)$? E se $a < b$, como será a chamada recursiva?

7.6 Considere a função $f(n) = 2^n$, onde n um número natural. Veja que ela pode ser calculada de forma recursiva:

$$2^0 = 1 \;\; ; \;\; 2^1 = 22^0 \;\; ; \;\; 2^2 = 22^1 \;\; ; \;\; 2^3 = 22^2 \;\; ; \;\; 2^4 = 22^3 \ldots$$

(a) No caderno escreva um pseudo-código recursivo para o algoritmo que recebe como entrada um número natural n e retorna $f(n) = 2^n$.

(b) Agora no computador implemente o pseudo-código elaborado acima.

7.7 Agora vamos generalizar a questão anterior. Considere a função $f(x, n) = 2^n$, onde n um número natural e x um número real. Veja que ela pode ser calculada de forma recursiva:

$$x^0 = 1 \;\; ; \;\; x^1 = xx^0 \;\; ; \;\; x^2 = xx^1 \;\; ; \;\; x^3 = xx^2 \;\; ; \;\; x^4 = xx^3 \ldots$$

(a) No caderno escreva um pseudo-código recursivo para o algoritmo que recebe como entrada um número natural n e um número real x e retorna $f(x, n) = x^n$.

(b) Agora no computador implemente o pseudo-código elaborado acima.

7.8 (a) No caderno escreva um pseudo-código recursivo para o algoritmo que recebe como entrada um *array* de números e retorna o número de elementos nulos nesse *array*.

(b) Agora no computador implemente o pseudo-código elaborado acima.

7.9 (a) No caderno escreva um pseudo-código recursivo para o algoritmo que recebe como entrada um *array* qualquer e retorna outro *array* definido pela ordem inversa do *array* de entrada.

(b) Agora no computador implemente o pseudo-código elaborado acima.

7.10 (a) Implemente de forma recursiva uma função que recebe como entrada dois números inteiros e retorna o maior divisor comum entre eles.

(b) Usando a função acima encontre o maior divisor comum entre os seguintes pares de números: 125 e 325, 2829 e 861, 299 e 217.

7.11 Implemente de forma recursiva uma função que recebe como entrada o número de discos em uma torre de Hanói, o nome do pino definido como origem, o nome do pino definido como destino e o nome do pino que será usado como auxiliar.

(a) Primeiro faça uma função que retorne nada, mas imprimi na tela os movimentos que devem ser feitos a fim de levar os discos do pino origem para o pino destino.

(b) Faça agora uma função que não imprime nada na tela, mas retorna um *array* de `"character"` que guarda em cada posição i o i-ésimo movimento que deve ser feito a fim de levar os discos do pino origem para o pino destino.

(c) Para terminar faça agora uma terceira função para resolver o problema da torre de Hanoi. Nessa o retorno deve ser uma lista com dois *array* de `"character"`: s e c. Tais *arrays* indicam os movimentos da seguinte forma: cada posição i de s guarda o nome do pino que sai o disco no i-ésimo movimento enquanto cada posição i de c guarda o nome do pino que chega o disco no i-ésimo movimento. Dessa forma cada movimento i é definido pelo transporte de um disco do pino s_i para o pino c_i.

(d) Teste as funções implementadas nesta questão em algum site de jogos online que tenha a Torre de Hanoi.

7.12 Implemente uma função recursiva que recebe como entrada um *array* de dados e retorna todos as permutações possíveis com ele. A saída pode ser uma matriz, cujas linhas guardam as permutações, ou uma lista. Quando for testar sua função entre com *arrays* pequenos, de tamanho não muito maior que 10, caso contrário vai demorar muito para rodar. Lembre-se que para um conjunto com n elementos temos $n!$ permutações.

Semana 8: Algoritmos de Ordenação

Nessa semana serão apresentados dois algoritmos de ordenação, isto é, algoritmos que ordem os elementos em um vetor qualquer. Veja que isso nada mais é do que a implementação da função `sort` já pronta no R.

Existem vários algoritmos de ordenação, uns mais eficientes que os outros como veremos na semana que vem. Nessa semana vamos aprender dois desses métodos: O método de Ordenação Bolha (*Bubble Sort*) e o método de Ordenação Rápida (*Quick Sort*).

8.1 Ordenação Bolha (*Bubble Sort*)

A ideia desse método é ordenar o vetor seguindo os seguintes passos:

- Primeiro leve o maior elemento para a última posição, comparando os elementos dois a dois até a última posição;
- Depois repita o processo e levar o segundo maior elemento para a segunda maior posição, comparando os elementos dois a dois até a penúltima posição;
- Esse processo se repete várias vezes até que o vetor esteja todo ordenado.

Exemplo 8.1.1 *Suponha* $v = (37, 33, 48, 12, 92, 25, 86, 57)$ *o vetor de entrada, ou seja, o vetor que queremos ordenar. O primeiro passo é levar o maior elementos para a maior posição fazendo comparações de elementos dois a dois. Vamos indicar por negrito os elementos que estão sendo comparados em cada passo.*

37	**33**	*48*	*12*	*92*	*25*	*86*	*57*	*como 37>33, troca*
33	**37**	**48**	*12*	*92*	*25*	*86*	*57*	*como 37<48, não troca*
33	*37*	**48**	**12**	*92*	*25*	*86*	*57*	*como 48>12, troca*
33	*37*	*12*	**48**	**92**	*25*	*86*	*57*	*como 48<92, não troca*
33	*37*	*12*	*48*	**92**	**25**	*86*	*57*	*como 92>25, troca*
33	*37*	*12*	*48*	*25*	**92**	**86**	*57*	*como 92>86, troca*
33	*37*	*12*	*48*	*25*	*86*	**92**	**57**	*como 92>57, troca*
33	*37*	*12*	*48*	*25*	*86*	*57*	*92*	*fim dessa etapa*

Os elementos que já estiverem em suas devidas posições serão sublinhado.

Veja que para realizar essa etapa foram necessárias 7 comparações.

Agora que o maior elemento já está na última posição o próximo passo é levar o segundo maior elemento até a segunda maior posição. Repare que nesse caso as comparações serão até a penúltima posição, uma vez que a última posição já está com o seu devido elemento.

33	**37**	*12*	*48*	*25*	*86*	*57*	*92*	*não troca*
33	**37**	**12**	*48*	*25*	*86*	*57*	*92*	*troca*
33	*12*	**37**	**48**	*25*	*86*	*57*	*92*	*não troca*
33	*12*	*37*	**48**	**25**	*86*	*57*	*92*	*troca*
33	*12*	*37*	*25*	**48**	**86**	*57*	*92*	*não troca*
33	*12*	*37*	*25*	*48*	**86**	**57**	*92*	*troca*
33	*12*	*37*	*25*	*48*	*57*	*86*	*92*	*fim dessa etapa*

Veja que para realizar essa etapa foram necessárias 6 comparações. Agora o terceiro maior elementos será levado para a terceira maior posição.

33	**12**	*37*	*25*	*48*	*57*	*86*	*92*	*troca*
12	**33**	**37**	*25*	*48*	*57*	*86*	*92*	*não troca*
12	*33*	**37**	**25**	*48*	*57*	*86*	*92*	*troca*
12	*33*	*25*	**37**	**48**	*57*	*86*	*92*	*não troca*
12	*33*	*25*	*37*	**48**	**57**	*86*	*92*	*não troca*
12	*33*	*25*	*37*	*48*	*57*	*86*	*92*	*fim dessa etapa*

Veja que para realizar essa etapa foram necessárias 5 comparações. Agora o quarto maior elementos será levado para a quarta maior posição.

12	**33**	*25*	*37*	*48*	*57*	*86*	*92*	*não troca*
12	**33**	**25**	*37*	*48*	*57*	*86*	*92*	*troca*
12	*25*	**33**	**37**	*48*	*57*	*86*	*92*	*não troca*
12	*25*	*33*	**37**	**48**	*57*	*86*	*92*	*não troca*
12	*25*	*33*	*37*	*48*	*57*	*86*	*92*	*fim dessa etapa*

Veja que para realizar essa etapa foram necessárias 4 comparações.

Nesse momento o vetor já está ordenado, porém o computador não tem como saber disso. Por isso as etapas seguintes serão realizadas, apesar de nenhuma troca ser feita.

12	**25**	*33*	*37*	*48*	*57*	*86*	*92*	*não troca*
12	**25**	**33**	*37*	*48*	*57*	*86*	*92*	*não troca*
12	*25*	**33**	**37**	*48*	*57*	*86*	*92*	*não troca*
12	*25*	*33*	*37*	*48*	*57*	*86*	*92*	*fim de mais uma etapa*
12	**25**	*33*	*37*	*48*	*57*	*86*	*92*	*não troca*
12	**25**	**33**	*37*	*48*	*57*	*86*	*92*	*não troca*
12	*25*	*33*	*37*	*48*	*57*	*86*	*92*	*fim de mais uma etapa*
12	**25**	*33*	*37*	*48*	*57*	*86*	*92*	*não troca*
12	*25*	*33*	*37*	*48*	*57*	*86*	*92*	*fim de mais uma etapa*
12	*25*	*33*	*37*	*48*	*57*	*86*	*92*	*fim de mais uma etapa*

E assim o vetor foi ordenado!

A seguir algumas observações sobre o método e o sobre exemplo acima.

- Em um vetor de tamanho n é preciso de $n-1$ comparações para realizar a primeira etapa, isto é, levar o maior elemento para a maior posição.

- Já para levar o segundo maior elemento para a segunda maior posição são realizadas $n-2$ comparações.
- De forma geral, para levar o j-ésimo maior elemento para a j-ésima maior posição, isto é realizar a etapa j, são realizadas $n-j$ comparações.
- Em cada etapa j serão comparados os elementos das posições i e $i+1$ para $i = 1, \ldots, n-j$.
- Veja que no exemplo acima foram realizadas 7 etapas para ordenar um vetor de tamanho 8. De forma geral, para ordenar um vetor de tamanho n são realizadas $n-1$ etapas.

Veja que serão necessários dois *loops*, um dentro do outro. Um para controlar as etapas e outro para controlar as comparações dentro de cada etapa. Veja a seguir um primeiro pseudo-código não recursivo para o algoritmo apresentado.

Entrada: v
Saída: vetor v arrumado em ordem crescente.
Nome: OrdenaBolha

1 Defina n = tamanho do vetor v;
2 Inicie $j = 1$;
3 Inicie $i = 1$;
4 Se $v_i < v_{i+1}$, troque a posição i com posição $i+1$ no vetor v;
5 Incremente $i = i + 1$
6 Se $i \leq n - j$, volte para a linha 4;
7 Incremente $j = j + 1$
8 Se $j \leq n - 1$, volte para a linha 3;
9 Retorne v.

Veja que a variável j controla as etapas, por isso ela varia de 1 até $n-1$. Já a variável i controla as comparações dentro de cada etapa j, por isso ela varia de 1 até $n-j$.

> ⚠ Na linha 4 o elemento da posição i será trocado com o elemento da posição $i+1$. Para isso é necessário utilizar uma variável temporária, se não um dos elementos do vetor será perdido.

No R a troca dos elementos entre as posições i e $i+1$ pode ser feita da seguinte maneira: `temp <- v[i]; v[i] <- v[i+1]; v[i+1] <- temp`.

O mesmo algoritmo pode ser implementado de forma recursiva.

Entrada: v
Saída: vetor v arrumado em ordem crescente.
Nome: OrdenaBolhaRec

1 Defina n = tamanho do vetor v;
2 Se $n = 1$, retorne v;
3 Inicie $i = 1$;

4 Se $v_i < v_{i+1}$, troque a posição i com posição $i+1$ no vetor v;
5 Incremente $i = i + 1$
6 Se $i \leq n - 1$, volte para a linha 4;
7 Defina $w = (v_1, v_2, ..., v_{n-1})$;
8 Defina wo =OrdenaBolhaRec(w);
9 Retorne $vo = (wo, v_n)$.

Veja que entre as linhas 3 e 6 o que está sendo feito é levar o maior elemento para a maior posição. Dessa forma o vetor w será formado pelos elementos do vetor v a menos do maior deles, que agora está na posição de índice n. Dessa forma wo guarda os elementos de v, a menos do maior deles, já em ordem crescente. Então v em ordem crescente será definido pela concatenação entre os elementos em wo e o maior elementos de v, que está guardado em v_n.

Repare que nos dois algoritmos acima se o vetor ficar ordenado no meio do processo o algoritmo continua mesmo assim, ou seja, ele irá fazer todas as iterações e comparações mesmo que nenhuma troca seja realizada. Para evitar algumas comparações desnecessárias podemos introduzir uma variável que indica se o vetor já está ou não ordenado. O processo será interrompido caso ele esteja.

Mas como saber se um vetor já está ordenado? Se em uma etapa j qualquer não ocorre nenhuma troca isso significa que o vetor já está ordenado.

O pseudo-código a seguir é uma alternativa ao pseudo código não recursivo já apresentado acima. A diferença é que neste foi introduzida uma variável `troca` para evitar algumas trocas desnecessárias.

Entrada: v
Saída: vetor v arrumado em ordem crescente.
Nome: OrdenaBolha2

1 Defina n = tamanho do vetor v;
2 Inicie $j = 1$;
3 Inicie $troca = F$;
4 Inicie $i = 1$;
5 Se $v_i < v_{i+1}$, troque a posição i com posição $i+1$ no vetor v e faça $troca = T$;
6 Incremente $i = i + 1$
7 Se $i \leq n - j$, volte para a linha 5;
8 Incremente $j = j + 1$
9 Se $j \leq n - 1$ e $troca = T$, volte para a linha 3;
10 Retorne v.

Veja que a variável `troca` é um objeto do tipo `"logical"` que guarda `TRUE` ou `FALSE`. Sempre que iniciamos uma etapa essa variável começa com `FALSE`, veja linha 3. Ela será trocada para `TRUE` quando acontecer uma primeira troca. Então se chegarmos na linha 9 com `troca=FALSE` significa que não houve troca alguma e por isso sabemos que o vetor já está ordenado. Nesse caso não devemos continuar as iterações, o processo é interrompido.

A mesma modificação pode ser feita também no algoritmo recursivo.

Entrada: v
Saída: vetor v arrumado em ordem crescente.
Nome: OrdenaBolhaRec2

1 Defina n = tamanho do vetor v;
2 Se $n = 1$, retorne v;
3 Inicie $troca = F$;
4 Inicie $i = 1$;
5 Se $v_i < v_{i+1}$, troque a posição i com posição $i + 1$ no vetor v e faça $troca = T$;
6 Incremente $i = i + 1$
7 Se $i \leq n - 1$, volte para a linha 5;
8 Se $troca = F$, retorne v;
9 Defina $w = (v_1, v_2, ..., v_{n-1})$;
10 Defina wo =OrdenaBolhaRec2(w);
11 Defina $vo = (wo, v_n)$;
12 Retorne vo.

8.2 Ordenação Rápida (*Quick Sort*)

A ideia desse método é ordenar o vetor da seguinte maneira:

- Primeiro o valor guardado na primeira posição, chamado de pivô, será levado para a sua posição correta de forma que os elementos à esquerda são menores que o pivô e os elementos à direita são maiores que o pivô.
- O passo seguinte é repetir o mesmo processo no sub-vetor à esquerda e no sub-vetor à diretia do pivô.

A questão é: como se leva o pivô para a sua posição correta? Essa tarefa será feita da seguinte maneira:

- Primeiro percorra o vetor a partir da posição seguinte a do pivô até encontrar um elemento maior que o pivô. Seja i a posição desse elemento.
- Se não existe elemento maior que o pivô, a posição correta do pivô é a última.
- Caso contrário, percorra o vetor do final para o início até encontrar um elemento menor ou igual ao pivô. Seja j a posição desse elemento.
- Se $i < j$, troque os elementos das posições i e j e recomece a busca por novos i e j a partir das posições trocadas.
- Se $i > j$, a posição correta para o pivô é a posição j.

Vejamos como o mesmo vetor v do Exemplo 8.1.1 será ordenado pelo método de Ordenação Rápida.

Exemplo 8.2.1 *Suponha* $v = (37, 33, 48, 12, 92, 25, 86, 57)$ *o vetor de entrada, ou seja, o vetor que queremos ordenar. O primeiro passo é encontrar a posição correta do pivô, que no caso é o* 25.

37	*33*	*48*	*12*	*92*	*25*	*86*	*57*	*definição do pivô*
37	**33**	*48*	*12*	*92*	*25*	*86*	*57*	*início da busca por i*
37	*33*	**48**	*12*	*92*	*25*	*86*	*57*	*definição de* $i = 3$
37	*33*	*48*	*12*	*92*	*25*	*86*	**57**	*início da busca por j*
37	*33*	*48*	*12*	*92*	*25*	**86**	*57*	
37	*33*	*48*	*12*	*92*	**25**	*86*	*57*	*definição de* $j = 6$
37	*33*	*25*	*12*	*92*	*48*	*86*	*57*	*como* $i < j$*, troca* v_i *com* v_j
37	*33*	*25*	**12**	*92*	*48*	*86*	*57*	*início da busca por i*
37	*33*	*25*	*12*	**92**	*48*	*86*	*57*	*definição de* $i = 5$
37	*33*	*25*	*12*	*92*	*48*	*86*	**57**	*início da busca por j*
37	*33*	*25*	*12*	*92*	*48*	**86**	*57*	
37	*33*	*25*	*12*	*92*	**48**	*86*	*57*	
37	*33*	*25*	*12*	**92**	*48*	*86*	*57*	
37	*33*	*25*	**12**	*92*	*48*	*86*	*57*	*definição de* $j = 4$
12	*33*	*25*	*37*	*92*	*48*	*86*	*57*	*como* $i > j$*, troca o pivô com* v_j

Veja que depois desses passos o pivô está em sua posição correta uma vez que antes dele estão os elementos menores que ele e depois os maiores. Agora o algoritmo é repetido para o sub-vetor à esquerda.

12	*33*	*25*	*37*	*92*	*48*	*86*	*57*	*definição do vetor*
12	**33**	*25*	*37*	*92*	*48*	*86*	*57*	*definição de* $i = 2$
12	*33*	**25**	*37*	*92*	*48*	*86*	*57*	*início da busca por j*
12	**33**	*25*	*37*	*92*	*48*	*86*	*57*	
12	*33*	*25*	*37*	*92*	*48*	*86*	*57*	*definição de* $j = 1$
12	*33*	*25*	*37*	*92*	*48*	*86*	*57*	*como* $i > j$*, troca o pivô com* v_j

Nesse momento os elementos 12 *e* 37 *já estão em suas posições corretas. Agora vamos refazer o processo para o sub-vetor à direita do pivô 12, ou seja, agora vamos trabalhar apenas com as posições de 2 e 3.*

12	**33**	*25*	*37*	*92*	*48*	*86*	*57*	*definição do pivô*
12	**33**	**25**	*37*	*92*	*48*	*86*	*57*	*como não existe elemento maior que o pivô, troque o pivô com* v_n
12	*25*	*33*	*37*	*92*	*48*	*86*	*57*	*definição de* $j = 3$

Veja que depois desses passos o pivô 33 também está em sua posição correta. Agora o algoritmo é repetido para o seu sub-vetor à direita, que tem apenas uma posição, logo já está ordenado. Como não existe sub-vetor a esquerda do pivô 33 temos a seguinte situação nesse momento:

12 *25* *33* *37* *92* *48* *86* *57*

Agora o processo será repetido no sub-vetor à direita do pivô 37. Ou seja, vamos trabalhar entre as posições 5-8.

12	*25*	*33*	*37*	**92**	*48*	*86*	*57*	*definição do pivô*
12	*25*	*33*	*37*	**92**	**48**	*86*	*57*	*início da busca por i*
12	*25*	*33*	*37*	**92**	*48*	**86**	*57*	
12	*25*	*33*	*37*	**92**	*48*	*86*	**57**	
12	*25*	*33*	*37*	**92**	*48*	*86*	**57**	
12	*25*	*33*	*37*	*57*	*48*	*86*	*92*	*como não existe elemento maior que o pivô, troque o pivô com* v_n

Agora o processo se repete no sub-vetor à esquerda do pivô 92.

12	*25*	*33*	*37*	**57**	*48*	*86*	*92*	*definição do pivô*
12	*25*	*33*	*37*	**57**	**48**	*86*	*92*	*início da busca por i*
12	*25*	*33*	*37*	**57**	*48*	**86**	*92*	*definição de $i = 7$*
12	*25*	*33*	*37*	**57**	*48*	**86**	*92*	*início da busca por j*
12	*25*	*33*	*37*	**57**	**48**	*86*	*92*	*definição de $j = 6$*
12	*25*	*33*	*37*	*48*	*57*	*86*	*92*	*como $i > j$, troca o pivô com v_j*

Como tanto o sub-vetor a esquerda quanto o sub-vetor à direta do pivô 57 são de tamanho 1, eles já estão ordenados e o processo termina.

12 *25* *33* *37* *48* *57* *86* *92*

E assim o vetor foi ordenado!

Reparou que a ideia do algoritmo é recursiva? Caso base: se $n = 1$ retorna o próprio vetor de entrada. Se não, primeiro coloque o pivô na posição correta, como explicado acima. Em seguida defina *we* e *wd* como os sub-vetores à esquerda e à direita e aplique a função neles. O vetor ordenado será a concatenação entre *we* ordenado, o pivô e *wd* ordenado.

A implementação de forma não recursiva é bem mais complicada, por isso será apenas apresentada a versão recursiva para o pseudo-código desse algoritmo.

Entrada: v
Saída: vetor v arrumado em ordem crescente.
Nome: OrdenaRapidoRec

1 Defina $n =$ tamanho do vetor v;
2 Se $n = 1$, retorne v;
3 Inicie $i = 2$ e $j = n$;
4 Se $v_1 < v_i$, vá para a linha 8;
5 Incremente $i = i + 1$
6 Se $i \leq n$, volte para a linha 4;
7 Troque a posição de v_1 com v_n e retorne v.
8 Se $v_1 \geq v_j$, vá para a linha 11;
9 Decremente $j = i - 1$
10 Se $j \geq 1$, volte para a linha 8;
11 Se $i < j$, troque as posições v_i com v_j e volte para a linha 5;
12 Troque as posições v_1 com v_j ;
13 Defina $we = (v_1, v_2, ..., v_{j-1})$;
14 Defina $wd = (v_{j+1}, v_2, ..., v_n)$;
15 Defina $weo =$OrdenaRapidoRec(we);
16 Defina $wdo =$OrdenaRapidoRec(wd);
17 Retorne $vo = (weo, v_j, wdo)$.

Exercícios - 8ª Semana

8.1 Implemente uma função que recebe como entrada um vetor v e retorna um vetor com os mesmos elementos de v mas em ordem crescente a partir do algoritmo de Ordenação Bolha visto na última aula teórica.

(a) Faça seguindo a primeira implementação sugerida: não-recursiva e sem a variável "troca".

(b) Faça seguindo a segunda implementação sugerida: recursiva e sem a variável "troca".

(c) Faça seguindo a terceira implementação sugerida: não-recursiva e com a variável "troca".

(d) Faça seguindo a quarta implementação sugerida: recursiva e com a variável "troca".

8.2 Modifique as quarto funções implementadas no exercício 8.1 acima de forma que elas retornem, além do vetor ordenado, o número de comparações realizadas para ordenar o vetor.

8.3 Modifique as quarto funções implementadas no exercício 8.1 acima de forma que elas retornem, além do vetor ordenado, o número de trocas realizadas para ordenar o vetor.

8.4 Usando as funções já implementadas, verifique quantas comparações cada uma delas precisa para ordenar os seguintes vetores:

(a) `v<-c(10,9,8,7,6,5,4,3,2,1)`

(b) `v<-c(1,3,5,5,4,0,-1,2,6,-2)`

(c) `v<-c(2,0,4,6,8,10,12,14,16,18,20)`

(d) `v<-c("fabio","ana","pedro","bruno","bruna","marco")`

OBS: No R é possível comparar palavras com o comando `<`. Por exemplo, digite no cursor o comando `"aaa"<"aba"` e veja a resposta. Dessa forma você também pode passar como entrada para as funções implementadas um *array* de caracteres, como sugerido no item 8.4d.

8.5 Escolha uma das funções não recursivas implementadas no exercício 8.1 e coloque antes de cada troca de posição o comando `plot(v)`, onde no lugar de `v` deve entrar o nome que você deu ao vetor na sua função. Agora digite os seguintes comandos:

```
> v <- c(5,9,4,2,6,10,3,8,1,7)
> w <- novafuncao(v)
> plot(w)
```

onde `novafuncao` é o nome da função modificada nesse exercícios. Veja os gráficos plotados e interprete.

8.6 Refaça a questão 8.1 de forma que o vetor retornado esteja em ordem decrescente em vez de em ordem crescente.

8.7 Implemente uma função que recebe como entrada um vetor v e retorna um vetor com os mesmos elementos de v mas em ordem crescente a partir do algoritmo de Ordenação Rápida visto na última aula teórica.

8.8 Modifique a função implementada no exercício 8.7 acima de forma que ela retorne, além do vetor ordenado, o número de comparações realizadas para ordenar o vetor.

8.9 Modifique a função implementada no exercício 8.7 acima de forma que ela retorne, além do vetor ordenado, o número de trocas realizadas para ordenar o vetor.

8.10 Verifique quantas comparações e quantas trocas são necessárias para ordenar os mesmo vetores do exercício 8.4 a partir da ordenação rápida.

8.11 Refaça a questão 8.7 de forma que o vetor retornado esteja em ordem decrescente em vez de em ordem crescente.

Semana 9: Complexidade de Algoritmos

Nessa última semana da segunda parte do curso será apresentado uma noção introdutória sobre complexidade de algoritmos. A ideia é apresentar o conceito, alguns exemplos e as notações usadas. Um estudo mais detalhado sobre esse tema precisaria de muito mais aulas.

9.1 Noção Introdutória

Analisar a complexidade de um algoritmo consiste em tentar "calcular" o custo para o computador em executar esse algoritmo. Esse cálculo na maioria das vezes não será preciso, mas a partir dele será possível comparar a complexidade entre diferentes algoritmos. Assim espera-se conseguir decidir entre dois algoritmos que realizam a mesma tarefa qual é o mais eficiente, ou seja, qual tem a menor complexidade.

Mas como podemos medir a complexidade de algoritmos? Uma alternativa é medir o tempo de execução, mas esse tempo varia de acordo com as especificações do computador e por isso não seria uma boa medida de comparação. Ao invés disso vamos contar o número de iterações realizadas pelo algoritmo em questão.

Exemplo 9.1.1 *Suponha um algoritmo que recebe como entrada um vetor v e retorna o maior elemento desse vetor, como no pseudo-código abaixo.*

Entrada: v
Saída: *maior elemento de* v.

1 *Defina* n = *tamanho do vetor* v;
2 *Inicie* $max = v_1$;
3 *Inicie* $i = 2$;
4 *Se* $max < v_i$, *faça* $max = v_i$;
5 *Incremente* $i = i + 1$
6 *Se* $i \leq n$, *volte para a linha 4;*
7 *Retorne* max.

É possível calcular o número de iterações realizadas quando a entrada é um vetor de tamanho n? Veja que nesse caso serão feitas $n - 1$ iterações.

Como no exemplo acima na maioria das vezes o número de iterações depende do "tamanho" dos argumentos de entrada, por isso o que vamos tentar encontrar é uma expressão que nos diga o número de operações em função de n.

Em alguns casos fica ainda mais complicado de encontrar esse número de operações. Veja no exemplo 9.1.2 a seguir.

Exemplo 9.1.2 *Suponha um algoritmo que recebe como entrada um vetor v e um número k e busca a posição de k dentro do vetor v como no pseudo-código abaixo.*

Entrada: *v, k*
Saída: *posição de k em v.*

1 *Defina n = tamanho do vetor v;*
2 *Inicie $i = 1$;*
3 *Se $v_i = k$, retorne i;*
4 *Incremente $i = i + 1$*
5 *Se $i \leq n$, volte para a linha 3;*
6 *Imprima: "O elemento não pertence ao vetor";*
7 *Retorne $NULL$.*

E nesse caso, é possível calcular o número de iterações realizadas quando a entrada é um vetor de tamanho n? Para esse algoritmo o número de iterações depende não só de n como também de onde está k. Por exemplo, se k for o primeiro elemento do vetor será feita apenas 1 iteração. Se k for o último ou se não houver elemento em v igual a k serão feitas n iterações. Na média, supondo que k tem chance igual de estar em cada uma das n posições, serão necessárias $\frac{1+2+...+n}{n} = \frac{n+1}{2}$ iterações.

Dessa forma dizemos que no pior caso serão realizadas n iterações, no melhor caso 1 iteração e no caso médio $\frac{n+1}{2}$ operações.

Devido a situação exposta no Exemplo 9.1.2 em geral vamos analisar a complexidade de um algoritmo pensando separadamente no melhor caso, no pior caso e no caso médio. As vezes um algoritmo ganha no pior caso mas perde no caso médio, por isso a importância em separar os casos.

9.2 A Notação O

Como já foi comentado, na maioria das vezes a conta não sera feita de forma exata devido a dificuldade em fazer isso. Como estamos interessados em comparar a complexidade para entradas grandes vamos apenas pensar qual algoritmo será mais eficiente quando $n \to \infty$, onde n é o tamanho da entrada.

Suponha que o número de iterações para realizar um certo algoritmo é definido por $g(n)$. Vamos tentar classificar esse número de operação em algum tipo de ordem. Vejamos algumas complexidades padrões:

- Se $g(n) = c_0$ dizemos que o algoritmo é da ordem $O(1)$;
- Se $g(n) = c_k n^k$ dizemos que o algoritmo é da ordem $O(n^k)$;
- Se $g(n) = c^n$ dizemos que o algoritmo é da ordem $O(c^n)$;
- Se $g(n) = c\log(n)$ dizemos que o algoritmo é da ordem $O(\log(n))$;

- Se $g(n) = n\log(n)$ dizemos que o algoritmo é da ordem $O(n\log(n))$.

Vale a seguinte hierarquia:

$$O(1) < O(\log(n)) < O(n) < O(n\log(n)) < O(n^2) < O(n^3) < \ldots < O(c^n)$$

Além disso, se $g(n) = g_1(n) + g_2(n)$ a ordem do algoritmo será a maior entre as ordem de g_1 e g_2.

9.3 Exemplos Básicos de Complexidade

9.3.1 Complexidade Constante - $O(1)$

Sempre que o número de operações não depender dos argumentos de entrada o algoritmos será de ordem constante, isto é, será $O(1)$. Como exemplo temos o seguinte algoritmo.

Entrada: n
Saída: indica se n é ou não maior que 5.

1 Se $n > 5$, retorne $TRUE$;
2 Retorne $FALSE$.

Veja que nesse caso $g(n) = 1$ e por isso dizemos que o algoritmo é de complexidade constante.

9.3.2 Complexidade Logarítmica- $O(log(n))$

Acontece quando um problema de tamanho n é transformado em um problema de tamanho $\frac{n}{2}$ em cada iteração. Por exemplo, quando buscamos a posição de um elemento em um vetor ordenado. Se o elemento procurado não estiver na posição central desse vetor podemos buscá-lo no sub-vetor à esquerda ou à direita, não é preciso buscar no vetor todo.

9.3.3 Complexidade Linear - $O(n)$

Sempre que o número de operações for proporcional a n o algoritmos será de ordem linear, isto é, será $O(n)$. Como exemplo podemos citar o algoritmo que recebe um vetor e retorna o seu maior elemento (Exemplo 9.1.1). Veja que nesse caso temos $g(n) = n-1$. De forma geral, sempre que tivermos um *loop* simples, isto é um for ou um while, o algoritmo será de ordem $O(n)$. Além disso, se tivermos dois *loops* seguidos (e não um dentro do outro) o algoritmo também sera de ordem linear.

9.3.4 Complexidade $n\log(n)$ - $O(n\log(n))$

Ocorre tipicamente quando um algoritmo divide o problema em dois problemas menores e resolve cada um deles separadamente. Geralmente isso ocorre a partir de um algoritmo recursivo. Como exemplo temos o algoritmo de Ordenação Rápida visto na semana

passada. Veja que no caso da ordenação rápida podemos dizer que $g(n) = 1 + g(k) + g(n-k-1)$ e $g(1) = 1$, com $k < n$ e n indicando o tamanho do vetor de entrada. Sempre que o número de iterações em um algoritmo tiver esse perfil ele será da ordem $O(n\log(n))$.

9.3.5 Complexidade quadrática - $O(n^2)$

Ocorre tipicamente quando temos dois *loops* um dentro do outro. Como exemplo temos a soma de duas matrizes: se as matrizes de entrada forem $n \times n$ o número de iterações será dados por $g(n) = n^2$, pois será preciso percorrer cada posição das matrizes.

Também é $O(n^2)$ o pior caso e o caso médio do algoritmo de Ordenação Bolha. Veja que o melhor caso desse algoritmo é $O(n)$, seria quando o vetor passado como entrada já estivesse ordenado.

9.3.6 Complexidade cúbica - $O(n^3)$

Ocorre tipicamente quando temos três *loops* um dentro do outro. Como exemplo temos a multiplicação de matrizes: se as matrizes de entrada forem $n \times n$ para cada posição das matrizes será feito o produto interno entre os vetores linhas e colunas, para realizar cada produto interno são necessárias n iterações, como isso será feito para cada um das n^2 posições no final serão feitas n^3 iterações.

9.3.7 Complexidade exponencial - $O(c^n)$

Um típico exemplo da complexidade exponencial é o cálculo do n elemento da série de Fibonacci a partir do algoritmo recursivo. Também temos a torre de hanoi. Em geral um algoritmo recursivo que depende de n que faz duas chamas recursiva simplificando para $n-1$ ou $n-2$ é de complexidade exponencial. Repare que a ordenação rápida não é, apesar de ter duas chamas recursivas cada uma delas foi diminuída de forma que a soma das duas ainda é menor que n.

Exercícios - 9ª Semana

9.1 Considere os algoritmos a seguir. Para cada um deles encontre a sua ordem de complexidade. Justifique a sua resposta. Se quiser, implemente os algoritmos para facilitar a sua análise.

a) Algoritmo não recursivo que retorna a soma dos elementos de um vetor de tamanho n passado como entrada.

b) Algoritmo recursivo que retorna a soma dos elementos de um vetor de tamanho n passado como entrada.

c) Algoritmo não recursivo que retorna a soma dos elementos de uma matriz de tamanho $n \times n$ passada como entrada.

d) Algoritmo não recursivo que busca a posição de um elementos passado como entrada em uma matriz de tamanho $n \times n$ também passada como entrada.

e) Algoritmo não recursivo que fornece o produto interno entre dois vetores de tamanho n passados como entrada.

f) Algoritmo não recursivo que fornece o produto entre uma matriz de tamanho $n \times n$ e um vetor de tamanho n passados como entrada.

g) Algoritmo não recursivo que verifica se uma matriz de tamanho $n \times n$ passada como entrada é simétrica ou não.

h) Algoritmo recursivo que retorna todas as permutações de um *array* de tamanho n passado como entrada (ex. 7.12)

9.2 Considere os dois pseudo-códigos a seguir. Ambos recebem como entrada um número natural n e retorna o n-ésimo termo da sequência $x_n = 2x_{n-1} + n$, com $x_0 = 1$.

Entrada: n	**Entrada**: n
Saída: x_n	**Saída**: x_n
Nome: Seq1	**Nome**: Seq2
Se $n = 0$, retorne 1;	Se $n = 0$, retorne 1;
Retorne $2 \times$Seq1$(n-1) + n$	Retorne Seq2$(n-1)$+Seq2$(n-1) + n$

(a) Implemente cada um dos pseudo-códigos acima no R.

(b) Teste as duas implementações para $n = 15$. Percebeu alguma diferença no tempo de execução?

(c) Vamos agora fazer um gráfico dos tempos de execução em função do valor de entrada em cada função. Para isso use os comandos `plot` e `Sys.time()`, este último retorna o instante de tempo em que o comando foi chamada. Analise os gráficos gerados e deduza a ordem de complexidade de cada uma das implementações.
OBS: Veja uma sugestão de código para o gráfico no final desta lista.

(d) O que tem de diferente nas duas implementações para uma ser tão mais demorada que a outra?

9.3 Fibonacci com Recursão:

(a) Refaça a função que retorna o n-ésimo elemento da sequência de Fibonacci usando recursão.

(b) Verifique o tempo de execução para $n = 1, 2, \ldots, 30$, ou seja, o tempo que a função demora para rodar. Para isso crie um vetor de tamanho 30 que guarda o tempo de execução para cada $n = 1, 2, \ldots, 30$. Para isso use o comando `Sys.time()` sugerido anteriormente.

(c) Faça um gráfico de n versus o tempo de execução.

(d) Analisando o gráfico, o que você diria da ordem desse algoritmo?

(e) Esse algoritmo é de ordem $O(2^n)$. Essa informação está de acordo com o gráfico?

(f) Veja que para n um pouco maior que 30 já demora muito para rodar.

OBS: Se fizermos as contas podemos ver que para $n = 100$ o algoritmo já fica impossível de rodar. Considerando que uma operação leva um picosegundo (1×10^{-12} segundos), 2^{100} operações levam $2^{100} \times 10^{-12}$ s $= 30.000.000.000.000$ anos!!!

9.4 Fibonacci sem Recursão:

(a) Implemente o passo-a-passo a seguir e verifique que ele fornece o n-ésimo termo da sequência de Fibonacci sem usar recursão.

---

1 Se $n = 1$ ou $n = 2$ retorne 1.
2 Defina $penultimo = 1$;
3 Defina $ultimo = 1$;
4 Inicie $i = 3$;
5 Defina $atual = ultimo + penultimo$;
6 Atualize $penultimo = ultimo$;
7 Atualize $ultimo = atual$;
8 Incremente $i = i + 1$;
9 Se $i \leq n$, volte para a linha 5;
10 Retorne $atual$.

(b) Analisando o código implementado, qual a ordem desse algoritmo?

(c) Como na questão anterior, gere um vetor com os tempos de execução.

(d) Faça um gráfico de n versus o tempo de execução.

(e) Analisando o gráfico, o que você diria da ordem desse algoritmo?

(f) Para o gráfico ficar mais explicativo, gere um vetor que guarda os tempos de execução para $n = 1, 2001, 4001, 6001, ..., 40001$ e em seguida faça um novo gráfico.

9.5 Vamos agora comparar o tempo de execução para o pior caso entre os dois algoritmos de ordenação vistos na semana passada.

(a) Escolha um dos algoritmos já implementados que ordenam um vetor passado como entrada pelo método de Ordenação Bolha. Veja que para esse método o pior caso é quando o vetor está na ordem inversa, por exemplo $(1, 2, 3, 4, 5)$. Então, com base no código ao final dessa lista de exercícios, dentro do `for` para cada i defina um vetor $v = (i, i - 1, \ldots, 2, 1)$ e rode o método bolha nesse vetor.

Faça ao final o gráfico do tempo de execução em função do tamanho do vetor de entrada. Em vez de subir o tamanho do vetor de entrada de um em um use um `while` e suba de 10 em 10.

(b) Faça o mesmo para a Ordenação Rápida, mas veja que nesse caso o pior caso já é o vetor ordenado. Então o vetor v definido dentro do `for` será $v = (1, 2, \ldots, i)$. Faça ao final o gráfico do tempo de execução em função do tamanho do vetor de entrada.

Sugestão de código para fazer o gráfico do tempo de execução de uma função `f` em função do argumento de entrada, supondo este inteiro.

```
> Temp <- NULL
> for(i in 1:20){
+   ta <- Sys.time()
+   f(i)
+   td <- Sys.time()
+   Temp  <- c(Temp,td - ta)
+ }
> plot(Temp)
```

Parte III

Uma Introdução ao Cálculo Numérico

Semana 10: Aproximação de Funções

Nessa semana vamos aprender como o computador avalia funções como e^x, $\ln(x)$ e $\text{sen}(x)$. A ideia é mostrar que se o computador sabe encontrar valores (aproximados) para $e^{-1,5}$, $\ln(0,5)$ ou $\text{sen}(1)$, nós também podemos encontrar sem a ajuda dele. Até porque, se o computador faz essas contas, faz porque alguém programou para que ele soubesse fazer. E que programa é esse?

10.1 Aproximação para a função $f(x) = e^x$

Resultados de cálculo, que não serão discutidos nesse curso, nos mostram que

$$\lim_{n\to\infty} \sum_{i=0}^{n} \frac{x^i}{i!} = e^x \qquad , \ \forall\ x \in \mathbb{R}. \tag{10.1}$$

Ou seja, para qualquer $x \in \mathbb{R}$, se $n \in \mathbb{N}$ for um número bem grande,

$$\sum_{i=0}^{n} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} \approx e^x.$$

Vale destacar que a convergência é mais rápida para valores de x próximos de zero. Isso significa que, considerando um número fixo de parcelas na equação acima, quanto mais próximo x está de zero, mais perto a soma está de e^x. Mas ela acontece qualquer que seja $x \in \mathbb{R}$, ou seja, mesmo para x distante de zero a soma converge para e^x quando o número de parcelas cresce.

Exemplo 10.1.1 *Vejamos como podemos usar a Equação 10.1 para encontrar uma aproximação para $e^{-1.5}$, sem a ajuda do computador, usando apenas operações de soma e multiplicação.*

Vamos começar encontrando uma aproximação usando $n = 2$:

$$e^{-1.5} = e^{-\frac{3}{2}} \approx 1 + \left(-\frac{3}{2}\right) + \frac{\left(-\frac{3}{2}\right)^2}{2!} = 1 - \frac{3}{2} + \frac{9}{8} = \frac{8 - 12 + 9}{8} = \frac{5}{8} = 0.625$$

E se quisermos uma aproximação mais precisa podemos usar um valor maior de n. Vamos ter que fazer um pouco mais de contas, mas teremos uma aproximação melhor. Por exemplo veja a aproximação para $n = 3$.

$$e^{-1.5} = e^{-\frac{3}{2}} \approx 1 + \left(-\frac{3}{2}\right) + \frac{\left(-\frac{3}{2}\right)^2}{2!} + \frac{\left(-\frac{3}{2}\right)^3}{3!} = \frac{5}{8} - \frac{9}{16} = \frac{10 - 9}{16} = \frac{1}{16} = 0.062$$

Agora para $n = 4$.

$$e^{-1.5} = e^{-\frac{3}{2}} \approx 1 + \left(-\frac{3}{2}\right) + \frac{\left(-\frac{3}{2}\right)^2}{2!} + \frac{\left(-\frac{3}{2}\right)^3}{3!} + \frac{\left(-\frac{3}{2}\right)^4}{4!} = \frac{1}{16} + \frac{27}{128} = \frac{35}{128} \approx 0.2734$$

E para $n = 5$.

$$e^{-1.5} = e^{-\frac{3}{2}} \approx 1 + \left(-\frac{3}{2}\right) + \frac{\left(-\frac{3}{2}\right)^2}{2!} + \frac{\left(-\frac{3}{2}\right)^3}{3!} + \frac{\left(-\frac{3}{2}\right)^4}{4!} + \frac{\left(-\frac{3}{2}\right)^5}{5!} = \frac{35}{128} - \frac{81}{1280} = \frac{269}{1280} \approx 0.2101$$

Se usarmos uma calculadora (computador) vamos encontrar a seguinte aproximação para $e^{-1.5} = 0.223130$. Veja que realmente as aproximações calculadas acima melhoraram quando n cresceu.

Critério de Parada

Se tivermos um computador podemos criar um programa que faça as contas para a gente. Mas como escolher o valor de n? Vamos optar pela seguinte estratégia: usando no minimo $n = 10$, para garantir que já estamos perto do valor limite, vamos calculando aproximações com n cada vez maior até que o incremento entre duas aproximações consecutivas seja relativamente pequeno.

Veja a seguir o pseudocódigo para encontrar uma aproximação para e^x.

Entrada: x e *incremento*.
Saída: uma aproximação para e^x.
Nome: AproxExp.

1 Calcule $aprox = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^{10}}{10!}$;
2 Comece com $i = 11$;
3 Defina $parcela = \frac{x^i}{i!}$;
4 Se $|parcela| < incremento$, retorne $aprox$. Se não, seguir.
5 Faça $aprox = aprox + parcela$;
6 Incremente $i = i + 1$;
7 Volte para a linha 3.

O pseudocódigo acima simplesmente calcula a série definida na Equação 10.1 até que o incremento da série, definido por *parcela*, seja menor que o valor atribuído à variável *incremento*, escolhido pelo usuário e passado como argumento. Garantimos que isso em algum momento acontece uma vez que $\frac{x^n}{n!} \xrightarrow[n\to\infty]{} 0$ para todo $x \in \mathbb{R}$.

A escolha do valor do *incremento* é feita pelo usuário. Geralmente se escolhe valores muito pequenos, como por exemplo, $erro = 10^{-3} = 0.001$ ou até menores.

OBS: Veja que a linha 3 poderia ser alterada para $parcela = parcela\frac{x}{i}$. Isso tornaria o código mais eficiente, pois não seria necessário calcular o fatorial em cada iteração do programa.

10.2 Aproximação para a função $f(x) = \ln(x)$

Resultados de cálculo, que não serão discutidos nesse curso, nos mostram que

$$\lim_{n\to\infty} \sum_{i=i}^{n} (-1)^{i+1} \frac{(x-1)^i}{i} = \ln(x) \qquad , \ \forall\ 0 < x < 2. \tag{10.2}$$

Ou seja, se $0 < x < 2$, se $n \in \mathbb{N}$ for um número bem grande,

$$\sum_{i=i}^{n}(-1)^{i+1}\frac{(x-1)^i}{i} = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \ldots + (-1)^{n+1}\frac{(x-1)^2}{2} \approx \ln(x).$$

Vale destacar que a convergência é mais rápida para valores de x próximos de 1. Isso significa que, considerando um número fixo de parcelas na equação acima, quanto mais próximo x está de 1, mais perto a soma está de $\ln(x)$. Mas ela acontece qualquer que seja $0 < x < 2$, ou seja, a soma converge para $\ln(x)$ quando o número de parcelas cresce.

Exemplo 10.2.1 *Vejamos como encontrar uma aproximação para* $\ln(0.5)$ *sem usar calculadora ou computador, usando apenas as operações de soma e multiplicação. Para isso vamos usar a Equação 10.2, atribuindo algum valor para n. Quanto maior o valor de n, melhor será a aproximação. Veja como fica a aproximação para* $n = 4$.

$$\ln(0.5) = \ln\left(\frac{1}{2}\right) \approx \left(\frac{1}{2}-1\right) - \frac{\left(\frac{1}{2}-1\right)^2}{2} + \frac{\left(\frac{1}{2}-1\right)^3}{3} - \frac{\left(\frac{1}{2}-1\right)^4}{4} = -\frac{1}{2} - \frac{1}{8} - \frac{1}{24} - \frac{1}{64} =$$

$$= \frac{-96-24-8-3}{192} = -\frac{130}{192} \approx -0.677$$

Se usarmos o computador vamos encontrar o seguinte valor: $\ln(0.5) \approx -0.6931$.

Mas se queremos o valor de $\ln(3)$? Nesse caso não podemos usar a Equação 10.2 substituindo x por 3, pois a convergência só acontece para $0 < x < 2$. Mas podemos usar uma propriedade dos logaritmos para resolver o nosso problema e calcular $\ln(3)$. Veja que

$$\ln\left(\frac{a}{b}\right) = \ln(a) - \ln(b)$$

sempre que $b \neq 0$. Então,

$$\ln\left(\frac{1}{x}\right) = \ln(1) - \ln(x) = -\ln(x) \Rightarrow \ln(x) = -\ln\left(\frac{1}{x}\right)$$

Além disso, se $x > 2 \Rightarrow 0 < \frac{1}{x} < \frac{1}{2} < 2$, logo podemos usar a Equação 10.2 para encontrar uma aproximação para $\ln\left(\frac{1}{x}\right)$. Resumindo, se precisamos encontrar $\ln(x)$ para $x > 2$, então vamos primeiro encontrar uma aproximação para $\ln\left(\frac{1}{x}\right)$ e depois multiplicar por -1, que temos então uma aproximação para $\ln(x)$. Veja um exemplo.

Exemplo 10.2.2 *Encontre uma aproximação para* $\ln(3)$ *usando a Equação 10.2 com* $n = 4$.

$$\ln(3) = -\ln\left(\frac{1}{3}\right) \approx -\left(\left(\frac{1}{3}-1\right) - \frac{\left(\frac{1}{3}-1\right)^2}{2} + \frac{\left(\frac{1}{3}-1\right)^3}{3} - \frac{\left(\frac{1}{3}-1\right)^4}{4}\right) =$$

$$= -\left(-\frac{2}{3} - \frac{2}{9} - \frac{8}{81} - \frac{16}{324}\right) = \frac{216+72+32+16}{324} = \frac{336}{324} \approx 1.037$$

Usando a calculadora chegamos em $\ln(3) \approx 1.0986$.

Veja no pseudocódigo a seguir como fica o algoritmo para encontrar uma aproximação para $\ln(x)$. O critério de parada é o mesmo adotado na aproximação da exponencial, o

algoritmo começa com $n = 10$ e depois incrementa n, até encontrar duas aproximações consecutivas tais que a diferença é menor que o valor de *incremento* definido pelo usuário.

Entrada: x e *incremento*.
Saída: uma aproximação para $\ln(x)$.
Nome: AproxLn.

1 Se $x \leq 0$, pare e retorne uma mensagem de erro;
2 Se $x \geq 2$, retorne $-\text{AproxLn}(\frac{1}{x}, erro)$;
3 Calcule $aprox = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \ldots + (-1)^{10+1}\frac{(x-1)^2}{2}$ Comece com $i = 11$;
4 Defina $parcela = \frac{(-1)^{i+1}(x-1)^i}{i}$
5 Se $|parcela| < incremento$, retorne $aprox$. Se não, siga.
6 Faça $aprox = aprox + parcela$;
7 Incremente $i = i + 1$;
8 Volte para a linha 4.

OBS: Veja que a linha 4 poderia ser alterada para $parcela = -parcela\frac{(x-1)(i-1)}{i}$. Isso tornaria o código mais eficiente.

10.3 Aproximação para a função $f(x) = \text{sen}(x)$

Resultados de cálculo, que não serão discutidos nesse curso, nos mostram que

$$\lim_{n\to\infty} \sum_{i=0}^{n} (-1)^i \frac{x^{2i+1}}{(2i+1)!} = \text{sen}(x) \qquad , \ \forall\ x \in \mathbb{R}. \tag{10.3}$$

Ou seja, para qualquer $x \in \mathbb{R}$, se $n \in \mathbb{N}$ for um número bem grande,

$$\sum_{i=0}^{n} (-1)^i \frac{x^{2i+1}}{(2i+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} \approx \text{sen}(x)$$

Vale destacar que a convergência é mais rápida para valores de x próximos de 0. Isso significa que, considerando um número fixo de parcelas na equação acima, quanto mais próximo x está de 0, mais perto a soma está de $\text{sen}(x)$. Mas ela acontece qualquer que seja $x \in \mathbb{R}$, ou seja, a soma sempre converge para $\text{sen}(x)$ quando o número de parcelas cresce.

Atenção: o argumento da função sen é a medida do ângulo em radianos. Dessa forma, sen é uma função periódica com período de tamanho 2π.

Exemplo 10.3.1 *Vejamos como encontrar uma aproximação para* $\text{sen}(1)$ *sem usar calculadora ou computador, usando apenas as operações de soma e multiplicação. Para isso vamos usar a Equação 10.3 com* $n = 4$.

$$\text{sen}(1) \approx 1 - \frac{1^3}{3!} + \frac{1^5}{5!} - \frac{1^7}{7!} + \frac{1^9}{9!} = 1 - \frac{1}{6} + \frac{1}{120} - \frac{1}{5.040} + \frac{1}{362.880} 0,8414710$$

Se usarmos o computador vamos encontrar o seguinte valor: $\text{sen}(1) \approx 0,8414709$. *Veja que com* $n = 5$ *já temos uma aproximação excelente.*

No Exemplo 10.3.1 tivemos uma boa aproximação para sen(1) usando n relativamente baixo, $n = 4$. Isso se deve ao fato de que 1 está razoavelmente próximo de zero. Provavelmente a aproximação não seria tão boa com $n = 4$ para encontrar sen(10). Mas podemos aproveitar a periodicidade da função seno e aumentar a velocidade de convergência mesmo para valores de x longe de zero. Veja um exemplo.

Exemplo 10.3.2 *Usando o computador encontramos* $\text{sen}(10) = -0.5440211$. *Vamos tentar encontrar esse valor na mão. Primeiro a partir da Equação 10.3 usando* $n = 4$ *e* $x = 10$.

$$\begin{aligned}\text{sen}(10) &\approx 10 - \frac{10^3}{3!} + \frac{10^5}{5!} - \frac{10^7}{7!} + \frac{10^9}{9!} \\ &= 10 - \frac{1.000}{6} + \frac{100.000}{120} - \frac{10.000.000}{5.040} + \frac{1.000.000.000}{362.880} \\ &\approx -1307,46.\end{aligned}$$

Veja que chegamos numa aproximação muito ruim, precisaríamos muito mais parcelas para chegar perto do valor correto de sen(10).

Veja que, pela periodicidade da função seno, $\text{sen}(10) = \text{sen}(10 - 2\pi) = \text{sen}(3.716815)$. *E como 3.716815 está mais perto de zero do que 10, provavelmente a aproximação será melhor usando o mesmo valor de n. Vamos às contas.*

$$\begin{aligned}\text{sen}(10) &= \text{sen}(10 - 2\pi) \approx \text{sen}(3.716815) \\ &\approx 3.716815 - \frac{3.716815^3}{3!} + \frac{3.716815^5}{5!} - \frac{3.716815^7}{7!} + \frac{3.716815^9}{9!} \\ &\approx -0,6397173.\end{aligned}$$

Veja como já melhorou! Mas ainda podemos melhorar mais, pois $10 - 2\pi - 2\pi = -2.566371$ *está ainda mais perto de zero.*

$$\begin{aligned}\text{sen}(10) &= \text{sen}(10 - 2\pi - 2\pi) \approx \text{sen}(-2.566371) \\ &\approx -2.566371 - \frac{-2.566371^3}{3!} + \frac{-2.566371^5}{5!} - \frac{-2.566371^7}{7!} + \frac{-2.566371^9}{9!} \\ &\approx -0,5482006.\end{aligned}$$

O Exemplo 10.3.2 nos mostra que aproveitar a periodicidade da função seno e escolher valores próximos de zero para serem aplicados na Equação 10.3 garante uma boa convergência com menos parcelas. Esse mecanismos será aplicado no pseudocódigo a seguir. Veja no pseudocódigo a seguir como fica o algoritmo para encontrar uma aproximação para sen(x).

Entrada: x e *erro*.
Saída: uma aproximação para sen(x).
Nome: AproxSeno.

1 Se $|x| > \pi$, retorne AproxSeno($x - 2\pi$,*erro*);
2 Calcule $aprox = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + (-1)^{10}\frac{x^{2\times 10+1}}{(2\times+1)!}$;
3 Comece com $i = 11$;
4 Defina $parcela = \frac{(-1)^i x^{2i+1}}{(2i+1)!}$;

5 Se $|parcela| < incremento$, retorne $aprox$. Se não, siga.
6 Faça $aprox = aprox + parcela$;
7 Incremente $i = i + 1$;
8 Volte para a linha 4.

Veja que a linha 4 poderia ser alterada para $parcela = -parcela\frac{x^2}{(2i+1)\times 2i}$. Isso tornaria o código mais eficiente.

Exercícios - 10ª Semana

10.1 A partir da Equação 10.1 encontre uma aproximações para e^1 e e^3 usando $n = 4$. Faça as contas na mão.

10.2 a) Implemente uma função que recebe como entrada $x \in \mathbb{R}$ e $n \in \mathbb{N}$ e retorna o polinômio $\sum_{i=0}^{n} \frac{x^i}{i!}$ avaliado em x. Vamos chamar essa função de `pol_exp(x,n)`.

b) Digite o código a seguir para ver como esse polinômio se aproxima da função exponencial conforme seu grau cresce.

```
> plot(exp,-4,4)
> grid()
> segments(x0=0,y0=0,x1=0,y1=150,lty=2)
> curve(pol_exp(x,n=2),add=T,col="violet")
> curve(pol_exp(x,n=3),add=T,col="red")
> curve(pol_exp(x,n=4),add=T,col="blue")
> curve(pol_exp(x,n=5),add=T,col="green")
```

Veja que a aproximação melhora quanto mais perto de 0 for o ponto avaliado ou quando maior for o valor de n.

10.3 a) Implemente o pseudocódigo visto em sala de aula que recebe como entrada $x \in \mathbb{R}$ e um erro e retorna uma aproximação para e^x.

b) Use a função implementada acima e encontre aproximações para e, e^{-1}, e^3, $\sqrt{e}$ e $e^{7.3}$. Compare os resultados com os valores fornecidos pela função `exp` do R.

c) Refaça a função implementada em 3a de forma que além da aproximação para e^x ela também retorne o valor de n usado para realizar essa aproximação. Refaça também os testes sugeridos em 3b.

d) Usando a função implementada no item 3a, isto é, a aproximação para e^x, implemente um outra função que recebe como entrada x e um erro e retorna uma aproximação para $e^{-x^2/2}$.

10.4 a) A partir da Equação 10.2 use $n = 3$ e encontre uma aproximações para $\ln(\frac{1}{10})$, $\ln(\frac{3}{5})$ e $\ln(4)$. Faça as contas na mão.

b) Sem olhar os valores exatos quais das aproximações encontradas no item 4a você acha que é mais precisa, a aproximação para $\ln(\frac{1}{10})$ ou para $\ln(\frac{3}{5})$? Por quê?

10.5 a) Implemente uma função que recebe como entrada $x \in \mathbb{R}$ e $n \in \mathbb{N}$ e retorna o polinômio $\sum_{i=i}^{n} (-1)^{i+1} \frac{(x-1)^i}{i}$ avaliado em x. Vamos chamar essa função de `pol_ln(x,n)`.

b) Digite o código a seguir para ver como esse polinômio se aproxima da função ln conforme seu grau cresce.

```
> plot(log,0,4)
> grid()
> segments(x0=1,y0=-4,x1=1,y1=10,lty=2)
> curve(pol_ln(x,n=2),add=T,col="violet")
> curve(pol_ln(x,n=3),add=T,col="red")
> curve(pol_ln(x,n=4),add=T,col="blue")
> curve(pol_ln(x,n=5),add=T,col="green")
```

Veja que a aproximação melhora quanto mais perto de 1 for o ponto avaliado ou quando maior for o valor de n.

10.6 a) Implemente o pseudocódigo visto em sala de aula que recebe como entrada um número real positivo x e um *erro* retorna uma aproximação para $\ln(x)$.

b) Use a função implementada acima e encontre aproximações para $\ln(0.1)$, $\ln(2)$, $\ln(10)$ e $\ln(3.8)$. Compare os resultados com os valores fornecidos pela função `ln` do R.

c) Implemente agora uma função que retorna o logaritmos de x em qualquer base, ou seja, essa nova função recebe como entrada x, b e *erro* e retorna uma aproximação para $\log_b(x)$. Para isso lembre-se da seguinte propriedade:

$$\log_b(x) = \frac{\log_a(x)}{\log_a(b)} \quad \forall \quad a, b, x \in \mathbb{R}^+.$$

Dica: essa nova função deve chamar a função implementada em 6a e usar a propriedade acima considerando $a = e$.

10.7 a) Implemente o pseudocódigo visto em sala de aula que recebe como entrada um número real positivo x e um *erro* e retorna uma aproximação para $\text{sen}(x)$. Aproveite a periodicidade da função e, a partir de uma chamada recursiva, simplifique o argumento até que ele esteja entre $-\pi$ e π.

b) Use a função implementada acima e encontre aproximações para $\text{sen}(2)$, $\text{sen}(25)$, $\text{sen}(50^o)$ e $\text{sen}\,(\pi/3)$. Compare os resultados com os valores fornecidos pela função `sin` do R.

c) Implemente agora uma função que recebe como entrada x e *erro* e retorna $\cos(x)$. Para isso perceba que a função cosseno é a função seno deslocada de $\pi/2$, ou seja,

$$\cos(x) = \text{sen}(x + \pi/2).$$

Semana 11: Aproximação de Raízes de Funções Reais

Encontrar raízes de funções tem diversas aplicações práticas, como por exemplo, encontrar aproximações para números irracionais ou achar máximo de funções. Nessa semana vamos estudar um método numérico para encontrar aproximações de raízes: o Método da Bisseção. Existem ainda outros que não serão apresentados no curso, como por exemplo, Método de Newton-Raphson, Método do Ponto Fixo e Método da Secante.

11.1 Preliminares

Primeiro vamos lembrar o que é raiz de uma função real.

Definição 11.1.1 *Seja $f : \mathbb{R} \to \mathbb{R}$ uma função real. Dizemos que $x \in \mathbb{R}$ é raiz de f quando $f(x) = 0$.*

O problema de encontrar raízes de uma função real muitas vezes é bem fácil. Sabemos, por exemplo, achar facilmente as raízes das funções $f(x) = x + 3$, $f(x) = x^2 - x + 4$, $f(x) = 1 - e^x$, $f(x) = ln(x + 1)$. Mas dependendo da expressão da função f encontrar suas raízes pode ser uma tarefa muito difícil, como por exemplo para as funções $f(x) = x^3 + 2x^2 - x + 1$, $f(x) = x + ln(x)$, $f(x) = e^x + x^2 - 2$. Nesses casos o problema será resolvido de forma aproximada a partir de métodos numéricos, como veremos nessa semana.

Antes de aplicar um método numérico para encontrar uma raiz pode ser de grande ajuda um estudo prévio da função f, a fim de conhecer melhor a localização da(s) raiz(es) que estamos procurando. Isto é, analisar a função para a qual se busca as raízes e tentar identificar um intervalo $[a, b]$ que contenha a raiz procurada. Esse estudo pode ser a partir do Teorema de Bolano ou por métodos gráficos.

Teorema 11.1.2 *(Teorema de Bolzano) Seja f uma função contínua em um intervalo $[a, b]$, tal que, $f(a)f(b) < 0$ (isto é, o sinal de $f(a)$ e $f(b)$ são diferentes). Então a função f possui pelo menos uma raiz no intervalo $[a, b]$.*

Assim, se f é uma função contínua, para encontrar um intervalo $[a, b]$ em que f possui pelo menos uma raiz basta encontrar a e b tais que $f(a)$ e $f(b)$ tenham sinais diferentes. Esse resultado será de grande utilidade para o Método da Bisseção.

Outra alternativa, que nem sempre é possível, é usar gráficos para encontrar esse intervalo. Nesse caso a vantagem é que muitas vezes podemos saber exatamente quantas raízes existem no intervalo. Veja a seguir dois exemplos em que o gráfico de f é usado para encontrar um intervalo $[a, b]$ em que f possui pelo menos uma raiz.

Exemplo 11.1.3 *Seja* $f(x) = e^x + x^2 - 2$. *Queremos nesse exemplo encontrar intervalos reais que contenham a raiz procurada. Veja que nesse caso é possível afirmar que* $f(x) = g(x) - h(x)$, *com* $g(x) = e^x$ *e* $h(x) = 2 - x^2$. *Logo,* $f(x) = 0$ *se e somente se* $g(x) = h(x)$. *Não sabemos ainda resolver a equação, mas sabemos desenhar o gráfico de* g *e* h. *As raízes de* f *serão as abcissas dos pontos de intercessão da curva de* g *com* h. *Veja o gráfico apresentado na Figura 11.1.*

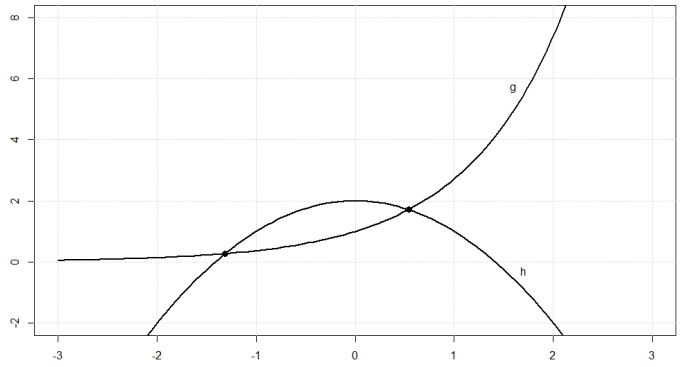


Figura 11.1: Gráficos das curvas $g(x) = e^x$ e $h(x) = 2 - x^2$ (Exemplo 11.1.3).

Logo, podemos afirmar que existe uma raiz de f *no intervalo* $[-2, -1]$ *e outra no intervalo* $[0, 1]$.

Exemplo 11.1.4 *O mesmo pode ser feito para encontrar um intervalo* $[a, b]$ *onde a função* $f(x) = x + ln(x) - 2$ *possui pelo menos uma raiz. Vamos desenhar o gráfico de* $g(x) = ln(x)$ *e* $h(x) = -x + 2$ *e procurar as intercessões das duas curvas. Veja o gráfico apresentado na Figura 11.2. Assim podemos afirmar que a única raiz de* f *está dentro do intervalo* $[1, 2]$.

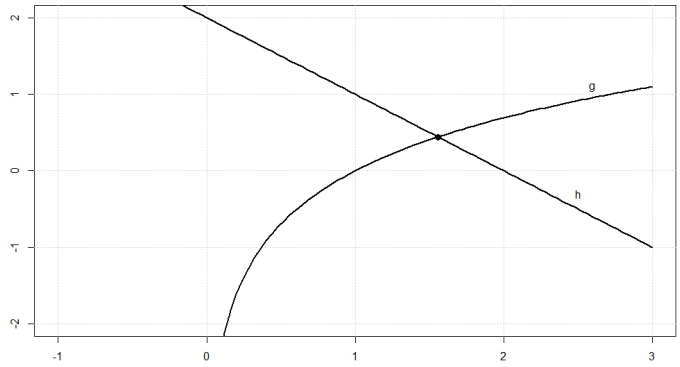


Figura 11.2: Gráficos das curvas $g(x) = \ln(x)$ e $h(x) = -x + 2$ (Exemplo 11.1.4).

11.2 Método da Bisseção

Para que o método da Bisseção seja aplicado é preciso primeiro conhecer um intervalo $[a, b]$ tal que $f(a)$ e $f(b)$ tenham sinais opostos, isto é, tal que $f(a)f(b) < 0$. Dessa forma, de acordo com o Teorema de Bolzano, podemos garantir que existe pelo menos uma raiz nesse intervalo. O que o método vai fazer é retornar uma aproximação para uma dessas raízes.

A ideia do método é dividir o intervalo $[a, b]$ ao meio e, usando o resultado do Teorema de Bolzano, decidir em qual dos dois lados a raiz está. Ou seja, começamos com $[a, b]$ tal que $f(a)f(b) < 0$ e depois de um passo chegamos em dois outros intervalos: $[a, \frac{a+b}{2}]$ e $[\frac{a+b}{2}, b]$. Calculando o valor de $f(\frac{a+b}{2})$ e comparando o seu sinal com os sinais de $f(a)$ e $f(b)$ podemos determinar em qual dos dois subintervalos a raiz está.

Se $f(a)f(\frac{a+b}{2}) < 0$ sabemos que a raiz está no intervalo $[a, \frac{a+b}{2}]$, caso contrário teremos $f(b)f(\frac{a+b}{2}) < 0$ e com isso podemos concluir que a raiz está no intervalo $[\frac{a+b}{2}, b]$. Veja que dessa forma chegamos em um intervalo menor tal que a função f avaliada nos pontos extremos desse intervalo tem sinais opostos. O processo é repetido e em cada iteração chegamos em um intervalo cada vez menor, ou seja, estamos cada vez mais próximos de uma raiz de f.

Exemplo 11.2.1 *Seja $f(x) = x^3 - x - 4$. Nesse exemplo vamos encontrar uma aproximação para uma raiz real de f. Primeiro veja que a função f possui uma raiz no intervalo $[0, 2]$, uma vez que $f(0) = -4$ e $f(2) = 2$. Então vamos realizar 4 iterações do método da bisseção para encontrar na mão uma aproximação para uma raiz de f nesse intervalo.*

Começamos com $a = 0$ e $b = 2$: $f(a) = f(0) = -4 < 0$ e $f(b) = f(2) = 2 > 0$. Definimos então m como sendo o ponto médio entre a e b: $m = \frac{a+b}{2} = 1$. Veja que $f(m) = f(1) = -4$. Como $f(m)$ e $f(b)$ tem sinais opostos podemos afirmar que existe uma raiz dentro do intervalo $[m, b] = [1, 2]$. Veja que chegamos em um intervalo menor que o original.

Vamos fazer agora tudo de nova a partir desse novo intervalo $[1, 2]$. Então na segunda iteração do método temos $a = 1$ e $b = 2$ tais que $f(a) < 0$ e $f(b) > 0$. Definimos então m como sendo o ponto médio entre a e b: $m = \frac{1+2}{2} = \frac{3}{2}$. Veja que $f(m) = \frac{27}{8} - \frac{3}{2} - 4 = \frac{27-24-32}{8} < 0$. Então $f(m)$ e $f(b)$ têm sinais opostos e por isso podemos afirmar que existe uma raiz dentro do intervalo $[m, b] = [\frac{3}{2}, 2]$. E assim reduzimos ainda mais o tamanho do intervalo que contém a raiz que estamos procurando.

Na terceira iteração temos $a = \frac{3}{2}$ e $b = 2$, assim definimos $m = \left(\frac{3}{2} + 2\right)\frac{1}{2} = \frac{7}{4}$. Veja que $f(m) = \frac{343}{64} - \frac{7}{4} - 4 = \frac{343-112-256}{64} < 0$. Então $f(m)$ e $f(b)$ têm sinais opostos e por isso podemos afirmar que existe uma raiz dentro do intervalo $[m, b] = [\frac{7}{4}, 2]$. E assim reduzimos ainda mais o tamanho do intervalo que contém a raiz que estamos procurando.

A quarta e última iteração realizada nesse exercício começa com $a = \frac{7}{4}$ e $b = 2$. Defina $m = \left(\frac{7}{4} + 2\right)\frac{1}{2} = \frac{15}{8}$. Veja que $f(m) = \frac{3375}{512} - \frac{15}{8} - 4 = \frac{3375-960-2048}{512} > 0$. Então $f(a)$ e $f(m)$ têm sinais opostos e por isso podemos afirmar que existe uma raiz dentro do intervalo $[a, m] = [\frac{7}{4}, \frac{15}{8}]$. E assim reduzimos ainda mais o tamanho do intervalo que contém a raiz que estamos procurando.

Veja que ao final de 4 iterações podemos afirmar que existe uma raiz dentro do intervalo $[\frac{7}{4}, \frac{15}{8}] = [1.75, 1.875]$, então com certeza o ponto médio desse intervalo dista da verdadeira raiz no máximo $\left(\frac{15}{8} - \frac{7}{4}\right)\frac{1}{2} = \frac{1}{16} = 0.0625$. Ou seja, $\left(\frac{15}{8} + \frac{7}{4}\right)\frac{1}{2} = \frac{29}{16} = 1.8125$ é uma aproximação para a raiz de f com erro de no máximo 0.0625.

Critério de Parada

Seguindo a ideia apresentada no exemplo podemos fazer com que o computador realize diversas iterações do método até alcançar a precisão desejada. Ou seja o usuário indica um erro ε e quando o algoritmo chegar em um intervalo $[a_n, b_n]$ de diâmetro menor que 2ε, isto é $b_n - a_n < 2\varepsilon$, podemos afirmar que o ponto médio desse intervalo dista menos que ε de uma raiz de f. Então podemos afirmar que o ponto médio é uma aproximação para uma raiz de$f$ com erro menor que ε.

Veja a seguir o pseudo-código de uma função que recebe como entrada a, b, f e ε e retorna uma aproximação com erro menor que ε para uma raiz de f dentro do intervalo $[a, b]$ a partir do Método da Bisseção.

Entrada: a, b, f e ε.
Saída: uma aproximação para uma raiz de f dentro do intervalo $[a, b]$.
Nome: RaizBissecao.

1 Se $f(a)$ e $f(b)$ não têm sinais opostos, pare e retorne erro.
2 Defina $m = \frac{a+b}{2}$, o ponto médio do intervalo $[a, b]$;
3 Se $b - a < 2\varepsilon$, retorne m. Se não, siga.
4 Se $f(a)f(m) < 0$, faça $b = m$ e volte para a linha 2;
5 Caso contrário, faça $a = m$ e volte para a linha 2.

Veja que na linha 4 quando fazemos $b = x$ estamos escolhendo ficar com o intervalo $[a, \frac{a+b}{2}]$ pois é nesse subintervalo que a raiz está. Já na linha 5 a escolha é por ficar com o subintervalo $[\frac{a+b}{2}, b]$, uma vez que $f(a)f(x) > 0$ e isso indica que a e x têm mesmo sinal logo b e x tem sinais opostos.

Dica: Talvez fique mais simples se o algoritmo for implementado usando o `repeat`.

O algoritmo também pode ser implementado de forma recursiva. Veja como no pseudo-código a seguir.

Entrada: a, b, f e ε.
Saída: uma aproximação para uma raiz de f dentro do intervalo $[a, b]$.
Nome: RaizBissecaoRec.

1 Se $f(a)$ e $f(b)$ não têm sinais opostos, pare e retorne erro.
2 Defina $m = \frac{a+b}{2}$, o ponto médio do intervalo $[a, b]$;
3 Se $b - a < 2\varepsilon$, retorne m. Se não, siga.
4 Se $f(a)f(m) < 0$, retorne RaizBissecaoRec(a, m, f, ε);
5 Caso contrário, retorne RaizBissecaoRec(m, b, f, ε);

Algumas vantagens desse método: se conseguimos um intervalo $[a, b]$ tal que $f(a)$ e $f(b)$ têm sinais opostos, temos a garantia de que o método converge para uma raiz de f; ao final do método sabemos que chegamos em uma aproximação com precisão ε, ou seja, o valor x fornecido pelo método é tal que $|x - \alpha| < \varepsilon$, onde α é a rais procurada e desconhecida.

Algumas desvantagens desse método: é necessário conhecer um intervalo $[a, b]$ tal que $f(a)f(b) < 0$, as vezes esse intervalo nem existe; e o processo de convergência não é dos mais rápidos.

11.3 Comentários Gerais

Os métodos citados são usados para encontrar raízes de funções, mas eles também podem ser muito úteis para outros problemas que acabam virando um problema de encontrar raízes.

Por exemplo, se quisermos encontrar uma solução para a expressão $e^x - x^2 = x$ podemos transformar esse problema original para o de encontrar as raízes da função $f(x) = e^x - x^2 - x$. Então um método numérico pode ser aplicado nessa função f.

Outro exemplo que podemos citar é quando queremos encontrar o ponto de máximo (ou de mínimo) de uma função f. Para encontrar o ponto de máximo vamos derivar f e procurar as raízes de f'. Se não for possível de encontrar as raízes na mão podemos procurar aproximações para ela a partir dos métodos vistos nesse capítulo. Mas nesse caso atenção, a função para a qual estamos aplicando o método é f'. Então se formos usar o Método da Bisseção precisamos de um intervalo $[a, b]$ tal que $f'(a)f'(b) < 0$ e se formos usar o Método de Newton-Raphson precisamos encontrar a derivada de f'.

Exercícios - 11ª Semana

11.1 Vamos usar o Método da Bisseção para encontrar a raiz de $f(x) = e^x - \frac{1}{x}$.

a) Primeiro implemente uma função que recebe como entrada os valores a, b e ε e retorna uma aproximação da raiz de $f(x) = e^x - \frac{1}{x}$ com erro menor que ε pelo Método da Bisseção.

b) Para testar a sua função é preciso primeiro escolher valores de a e b tais que $f(a)$ e $f(b)$ tenham sinais opostos. Encontre um intervalo com essa característica. Tente usar o método gráfico.

c) Escolha o valor de ε e encontre uma aproximação para a raiz de f com erro menor que ε.

d) Refaça agora o item 1a usando recursão. Ou seja, implemente uma função recursiva que recebe como entrada os valores a, b e ε e retorna uma aproximação da raiz de $f(x) = e^x - \frac{1}{x}$ com erro menor que ε pelo Método da Bisseção.

e) Vamos ver com mais detalhes como esse método funciona. Para isso modifique uma das funções implementadas (1a ou 1d)de forma que a função implementada retorne um *array* com todos os pontos médios dos intervalos encontrados durante o método.

f) Use os comandos de desenho `plot`, `curve`, `points` para visualizar o passo-a-passo do algoritmo. Guarde a os pontos médios retornado pela última função implementadas em uma variável de nome `saida` e siga a sugestão de código a seguir.

```
> plot(0,xlab="x",ylab="f(x)",xlim=c(0,1),ylim=c(-1,1),type="n")
> grid()
> segments(x0=-1,y0=0,x1=2,y1=0)
> segments(x0=0,y0=-2,x1=0,y1=2)
> curve(exp(x)-1/x,col="blue",add=T)
> for(i in 1:length(saida)){
+ points(saida[i],0,col="red",pch=18,cex.lab=1)
+ text(saida[i],-0.2,i,col="red")
+ }
```

11.2 Vamos usar o Método da Bisseção para encontrar uma aproximação para o número irracional $\sqrt{3}$.

a) Escolha uma função f que tenha uma raiz em $\sqrt{3}$. Não vale $f(x) = x - \sqrt{3}$, pois estamos supondo que não sabemos calcular $\sqrt{3}$. A função deve conter apenas números racionais.

b) Implemente, para a função escolhida, o Método da Bisseção.

c) Escolha valores iniciais de a, b e ε e encontre a aproximação desejada.

11.3 a) Quantas soluções tem a equação $e^x = 2 - x^2$.

b) Use o Método da Bisseção para encontrar todas as soluções.

c) Como você pode verificar se as aproximações encontradas são razoáveis e/ou estão certas? Faça isso.

Semana 12: Derivação Numérica

Nesse capítulo vamos estudar como usar o computador para encontrar uma aproximação para o coeficiente angular da reta tangente à f no ponto x_0, denominada $f'(x_0)$. Para isso vamos supor que sabemos avaliar f em uma vizinhança de x_0, ou pelo menos uma aproximação para isso.

12.1 Métodos Numéricos

12.1.1 Primeiro Método

Já estudamos em cálculo que a derivada de uma função f em x_0 é a inclinação da reta tangente à f em x_0. E para encontrar o valor de $f'(x_0)$ temos que calcular o seguinte limite:

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}.$$

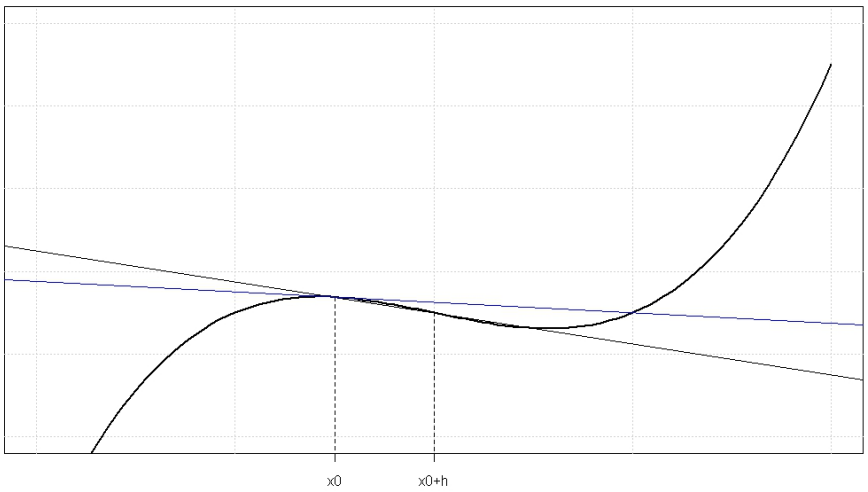


Figura 12.1: Ilustração para a derivada em x_0.

Esse limite já sugere uma aproximação para a derivada de f em x_0. Se escolhemos h bem pequeno podemos afirmar que:

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}.$$

O que estamos fazendo nesse caso é aproximando a inclinação da reta tangente à f no ponto x_0 (reta em azul na Figura 12.1) pela inclinação da reta secante a f que passa pelos pontos $(x_0, f(x_0))$ e $(x_1, f(x_1))$, com $x_1 = x_0 + h$ (reta em preto na Figura 12.1). Veja que quanto menor o valor de h mais próxima está uma reta da outra, logo mais próxima está uma inclinação de uma reta da inclinação da outra.

Podemos usar h positivo ou negativo. Caso $h > 0$ a reta secante em questão passa por $(x_0, f(x_0))$ e $(x_1, f(x_1))$, com $x_1 = x_0 + h$ sendo um ponto à direita de x_0, como mostra a Figura 12.1. Já se $h < 0$ a reta considerada na aproximação passa em $(x_0, f(x_0))$ e $(x_2, f(x_2))$, com $x_2 = x_0 - h$ sendo um ponto à esquerda de x_0.

12.1.2 Segundo Método

Uma outra alternativa para calcular uma aproximação para $f'(x_0)$ é usar a reta secante que passa pelos pontos $(x_0-h, f(x_0-h))$ e $(x_0+h, f(x_0+h))$, nesse caso vamos considerar o seguinte limite:

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0 - h)}{2h}.$$

Se escolhemos h bem pequeno podemos afirmar que:

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0 - h)}{2h}.$$

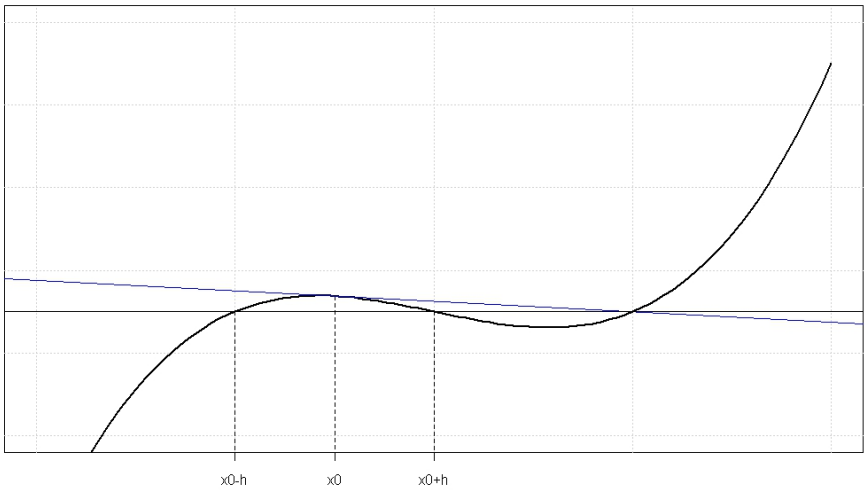


Figura 12.2: Ilustração para a derivada em x_0.

Esse novo método está ilustrado na Figura 12.2. Nesse caso a reta secante usada na aproximação não leva em consideração $(x_0, f(x_0))$ e sim $(x_1, f(x_1))$ e $(x_2, f(x_2))$, com $x_1 = x_0 + h$ e $x_2 = x_0 - h$, pontos à direita e à esquerda de x_0, respectivamente.

12.2 Algoritmo

Vamos ao algoritmo. Para aproximar $f'(x_0)$ vamos começar com um h qualquer, que pode ou não ser informado pelo usuário. A partir desse valor de h calculamos uma primeira

aproximação. Depois diminuímos o valor de h, por exemplo dividindo por 2, e calculamos uma nova aproximação.

Critério de Parada

Realizamos esse procedimento diversas vezes até que a diferença entre aproximações consecutivas seja bem pequena, ou melhor, seja menor que o valor de *incremeto* determinado pelo usuário.

Entrada: x_0, f e *incremento*.
Saída: uma aproximação para $f'(x_0)$.
Nome: DerivadaNumérica

1 Defina $h = 1$;
2 Defina $x_1 = x_0 + h$ e $x_2 = x_0 - h$;
3 Se $x_1 \notin D(f)$ ou $x_2 \notin D(f)$, pare e retorne erro.
4 Calcule $d = \frac{f(x_1) - f(x_2)}{2h}$;
5 Atualize o valor de h: $h = \frac{h}{2}$;
6 Atualize x_1 e x_2: $x_1 = x_0 + h$ e $x_2 = x_0 - h$;
7 Calcule $\tilde{d} = \frac{f(x_1) - f(x_2)}{2h}$;
8 Se $|d - \tilde{d}| < incremento$, retorne $\tilde{d}$;
9 Faça $d = \tilde{d}$ e volte para a linha 5.

Dica: Talvez fique mais simples se o algoritmo for implementado usando o `repeat`.

Veja agora uma possibilidade de realizar o algoritmo de forma recursiva. Nesse caso vai ser bem mais simples de h também for passado como entrada.

Entrada: x_0, f, *incremento* e h.
Saída: uma aproximação para $f'(x_0)$.
Nome: DerivadaNuméricaRec

1 Defina $x_1 = x_0 + h$ e $x_2 = x_0 - h$;
2 Se $x_1 \notin D(f)$ ou $x_2 \notin D(f)$, pare e retorne erro.
3 Calcule $d = \frac{f(x_1) - f(x_2)}{2h}$;
4 Atualize o valor de h: $h = \frac{h}{2}$;
5 Atualize x_1 e x_2: $x_1 = x_0 + h$ e $x_2 = x_0 - h$;
6 Calcule $\tilde{d} = \frac{f(x_1) - f(x_2)}{2h}$;
7 Se $|d - \tilde{d}| < incremento$, retorne $\tilde{d}$;
8 Retorne DerivadaNuméricaRec$(x_0, incremento, h)$.

Exercícios - 12ª Semana

Para os exercícios a seguir considere $f'(x_0)$ = coeficiente angular da reta tangente à f no ponto x_0.

12.1 Em cada item a seguir considere que as únicas informações disponíveis sobre a função f estejam na tabela apresentada. Complete a terceira coluna de cada tabela com aproximações para a derivada no ponto em questão. Faça as contas na mão. Para cada linha use o método mais apropriador de acordo com as informações disponíveis.

a)

x_0	$f(x_0)$	$f'(x_0)$
1.1	9.025013	
1.2	11.02318	
1.3	13.46374	
1.4	16.44465	

b)

x_0	$f(x_0)$	$f'(x_0)$
7.4	-68.31929	
7.6	-71.69824	
7.8	-75.15762	
8.0	-78.69741	

12.2 Compare o valor aproximado encontrado no exercício acima com os valores reais, dado pelas funções a seguir.

a) $f'(x_0) = 2e^{2x_0}$

b) $f'(x_0) = \frac{1}{x_0+2} - 2(x_0 + 1)$

12.3 Vamos agora implementar o método visto em sala de aula. Para isso considere a função $f(x) = \frac{1}{x^2+1}$.

a) Qual o domínio da função f?

b) Implemente uma função que recebe como entrada $x_0 \in Dom(f)$ e o valor de *incremento* e retorna uma aproximação para $f'(x_0)$ a partir do segundo método visto em sala de aula.

c) A partir da função implementada encontre aproximações para $f'(0)$, $f'(-\frac{1}{5})$ e $f'(\frac{1}{3})$.

d) Vamos implementar agora o algoritmo de forma recursiva. Para facilitar considere h um argumento de entrada da sua função. Dessa forma, implemente uma função recursiva que recebe como entrada $x_0 \in Dom(f)$, o valor do *incrementop* e h, e retorna uma aproximação para $f'(x_0)$ a partir do segundo método visto em sala de aula.

e) A partir da função recursiva encontre aproximações para $f'(0)$, $f'(-\frac{1}{5})$ e $f'(\frac{1}{3})$. Quando for chamar a função para encontrar as aproximações use $h = 1$.

12.4 Considere agora a função $f(x) = ln(x^2 + x - 2)$.

a) Qual o domínio da função f?

b) Implemente uma função que recebe como entrada $x_0 \in Dom(f)$ e o valor de *incremento*, e retorna uma aproximação para $f'(x_0)$ a partir do segundo método visto em sala de aula.
Dica: Nesse caso será necessário ter cuidado com: (i) o x_0 informado pelo usuário, se ele não estiver no domínio interrompa o processo e envie uma mensagem de erro; (ii) a escolha de h inicial, atenção para não acontecer de $x_0 - h$ ou $x_0 + h$ não pertencer ao domínio.

c) A partir da função implementada encontre aproximações para $f'(3)$, $f'(-\frac{5}{2})$ e $f'(\frac{4}{3})$.

d) Vamos implementar agora o algoritmo de forma recursiva. Para facilitar considere novamente h um argumento de entrada da sua função. Dessa forma, implemente uma função recursiva que recebe como entrada $x_0 \in Dom(f)$, o valor de *incremento* e h, e retorna uma aproximação para $f'(x_0)$ a partir do segundo método visto em sala de aula.

e) A partir da função recursiva encontre aproximações para $f'(3)$, $f'(-\frac{5}{2})$ e $f'(\frac{4}{3})$.

12.5 Considere agora $f(x) = e^{-x/3}\left(1 + \frac{x}{x^2+1}\right) - 1$.

a) Qual o domínio da função f?

b) Primeiro implemente uma função que recebe como entrada x e retorna $f(x)$. Vamos chamar essa função de `f`.

c) Implemente agora uma função que recebe como entrada x e retorna uma aproximação para $f'(x)$ considerando um erro de 10^{-3}. Vamos chamar esse função de `df`.

d) Nosso objetivo agora é usar o método da bisseção para encontrar os pontos de máximo e mínimo locais de f. Veja que esses pontos são os pontos x_0 tais que $f'(x_0) = 0$. Para isso siga os itens a seguir.

i) Digite `plot(f,xlim=c(-3,5))` e `plot(df,xlim=c(-3,5));abline(h=0)` e, comparando os dois gráficos, veja onde estão os pontos x_0 tais que $f'(x_0) = 0$. Para encontrais aproximações para tais pontos vamos buscar as raízes da função `df`.

ii) Use o método da bisseção para encontrar uma aproximação para o mínimo local de f. A partir dos gráficos escolha valores para a e b de forma a garantir que o método converge para o mínimo local. Faça a sua função de forma que ela chame `df`.

iii) Use o método da bisseção para encontrar uma aproximação para o máximo local de f. A partir dos gráficos escolha valores para a e b de forma a garantir que o método converge para o máximo local. Faça a sua função de forma que ela chame `df`.

iv) Vamos testar se deu certo. Guarde no objeto `xmin` a aproximação para o mínimo local e no objeto `xmax` a aproximação para o máximo local de f. Agora digite a seguinte sequência de comandos e discuta o gráfico gerado.

```
> plot(df,xlim=c(-3,5))
> abline(h=0)
> segments(x0=xmin,y0=2,x1=xmin,y1=-2,lty=2)
> points(xmin,0,pch=19,cex=1.2)
> segments(x0=xmax,y0=1,x1=xmax,y1=-1,lty=2)
> points(xmax,0,pch=19,cex=1.2)
```

Semana 13: Integração Numérica

Nessa semana veremos um método iterativo que, a partir de uma função conhecida f e de valores $a, b \in \mathcal{D}(f)$, encontra um valor aproximado para a área entre a curva de f e o eixo horizontal limitada no intervalo $[a, b]$, representada por cinza na Figura 13.1.

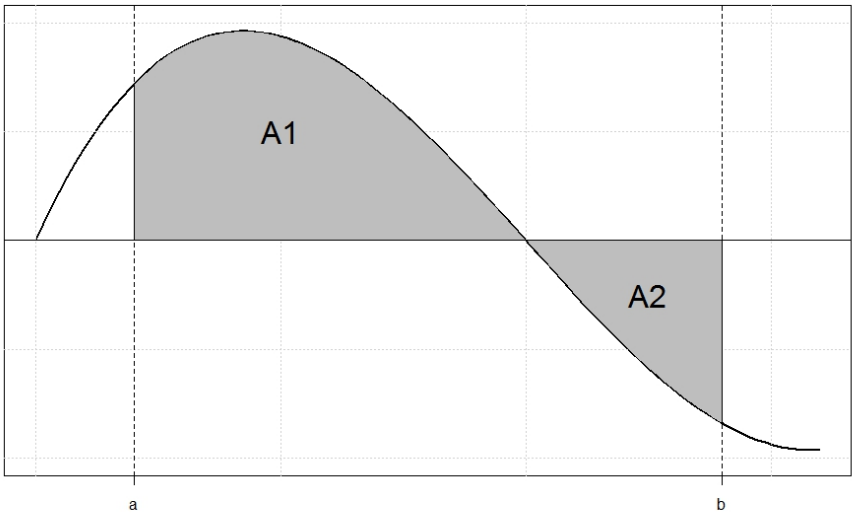


Figura 13.1: Área entre a curva de f e o eixo horizontal, limitada no intervalo $[a, b]$.

Dependendo da função f, essa conta pode ser feita de forma precisa a partir de integrais definidas, como vocês aprenderão em cálculo. Mas algumas vezes isso não é possível, ou seja, não existe como encontrar o valor da área de forma exata. Nesses casos a alternativa é encontrar uma aproximação a partir de métodos numéricos, como veremos nesse curso.

Antes de seguirmos com a ideia por trás do métodos vale destacar que nesse contexto a área entre a curva de f e o eixo horizontal, a qual queremos encontrar, considera área acima do eixo como positiva e área abaixo como negativa. Dessa forma, considerando a Figura 13.1, a área total entre a curva de f e o eixo, denominada A, é

$$A = A1 - A2,$$

onde $A1$ e $A2$ são os valores de área, ilustrados na Figura 13.1, ambos positivos.

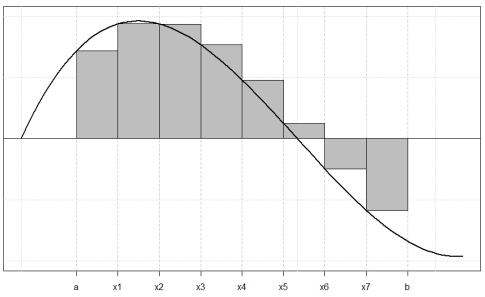
13.1 Aproximação por retângulos

Para fazer a aproximação por retângulos primeiro vamos dividirmos o intervalo $[a, b]$ em n subintervalos de mesmo tamanho. O tamanho de cada subintervalo será

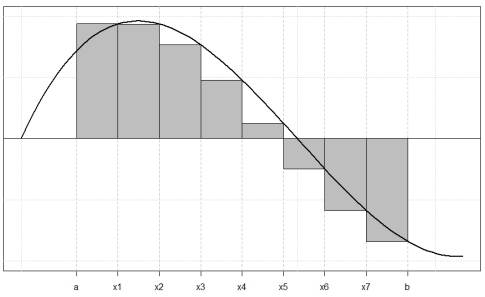
$$\delta = \frac{b-a}{n}$$

e os subintervalos são definidos pelos segmentos $[x_{i-1}, x_i]$, com $i = 1, \ldots, n$ e $x_0 = a$, $x_1 = a + \delta$, $x_2 = x_1 + \delta$, ..., $x_n = b$.

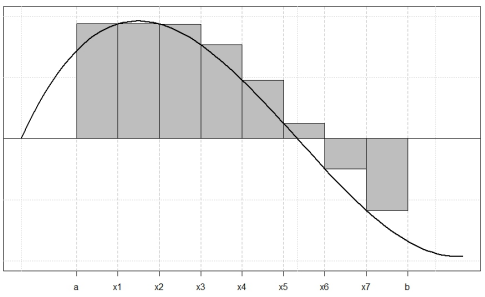
Para cada subintervalo $[x_{i-1}, x_i]$ vamos definir um retângulo cuja base é o próprio subintervalo e altura pode ser definida de diferentes formas, iremos apresentar 4 formas, como ilustra a Figura 13.2. Primeiro considere que a altura é o valor de f avaliado no ponto mais à esquerda desse subintervalo. Nesse caso a altura do retângulo de base $[x_{i-1}, x_i]$ é $f(x_{i-1})$, veja Figura 13.2(a). Também podemos definir a altura do retângulo como o valor de f avaliado no ponto mais à direita desse subintervalo. Nesse caso a altura do retângulo de base $[x_{i-1}, x_i]$ é $f(x_i)$, veja Figura 13.2(b).



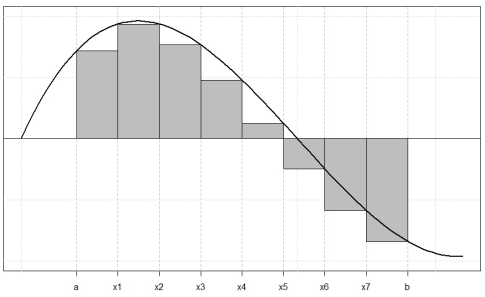
(a) Altura definida pelo ponto à esquerda.



(b) Altura definida pelo ponto à direita.



(c) Altura definida pelo maior valor de f.



(d) Altura definida pelo menor valor de f.

Figura 13.2: Aproximação por retângulos.

Podemos também definir a altura do retângulo de base $[x_{i-1}, x_i]$ pelo maior ou pelo menor valor entre $f(x_{i-1})$ e $f(x_i)$. Nesse caso a altura do retângulo de base $[x_{i-1}, x_i]$ é $\max(f(x_{i-1}), f(x_i))$ (Figura 13.2(c)), no caso de ser o maior valor, ou $\min(f(x_{i-1}), f(x_i))$ (Figura 13.2(d)), no caso de ser o menor valor.

Veja que podemos aproximar a área embaixo da curva de f pela soma das áreas dos n retângulos. Veja também que quanto maior o número retângulos, ou seja, quanto maior n, mais próximo está o valor da soma das áreas dos retângulos e o real valor da área.

Outra observação interessante é que se definirmos como a altura do retângulo de base $[x_{i-1}, x_i]$ o maior valor entre $f(x_{i-1})$ e $f(x_i)$, Figura 13.2(c), então a soma das áreas dos retângulos será sempre maior que o real valor da área. Dessa forma, dizemos que a soma das áreas desses retângulos, denominada $\hat{A}_U$, é um limite superior para o real valor da área A, pois

$$A \leq \hat{A}_U. \tag{13.1}$$

Por outro lado, se definirmos como a altura do retângulo de base $[x_{i-1}, x_i]$ o menor valor entre $f(x_{i-1})$ e $f(x_i)$, Figura 13.2(d), então a soma das áreas dos retângulos será sempre menor que o real valor da área. Dessa forma, dizemos que a soma das áreas desses retângulos, denominada $\hat{A}_L$, é um limite inferior para o real valor da área A, pois

$$\hat{A}_L \leq A. \tag{13.2}$$

13.2 Algoritmo

Como já foi discutido, serão usados os retângulos para aproximar a área entre a curva de f e o eixo horizontal. A ideia é escolher um valor para n, que representa o número de retângulos, e usar a soma das áreas dos retângulos para aproximar a área entre a curva de f e o eixo horizontal. Mas como determinar o número de subintervalos n que irá fornecer uma aproximação relativamente boa? E qual altura de retângulo vamos usar?

Critério de Parada

Veja que quanto mais subdivisões tem o intervalo $[a, b]$, mais precisa será a aproximação. Além disso, sendo $\hat{A}_U$ e $\hat{A}_L$ os valores da soma das áreas dos retângulos com alturas iguais a $\max(f(x_{i-1}), f(x_i))$ e $\min(f(x_{i-1}), f(x_i))$, respectivamente, de acordo com as Equações 13.1 e 13.2,

$$\hat{A}_L \leq A \leq \hat{A}_U.$$

Então o que vamos fazer para encontrar a aproximação é aumentar n, inciando com $n = 100$, até que a diferença entre os valores de $\hat{A}_L$ e $\hat{A}_U$ seja bem pequena, ou melhor, até que essa diferença seja menor que 2ε, onde ε é passado pelo usuário. Quando isso acontecer podemos afirmar que o ponto médio entre $\hat{A}_L$ e $\hat{A}_U$, definido por

$$\frac{\hat{A}_U + \hat{A}_L}{2},$$

é uma aproximação para a área desejada com erro menor que ε.

Entrada: a, b, f e ε.
Saída: uma aproximação para a área entre a curva de f e o eixo horizontal, limitada pelo intervalor $[a, b]$, com erro menor que ε.
Nome: IntegralNumérica

1 Defina $n = 100$;
2 Defina $\delta = \frac{b-a}{n}$;
3 Defina $\hat{A}_U = 0$ e $\hat{A}_L = 0$;
4 Inicie $x = a$;
5 Faça $\hat{A}_U = \hat{A}_U + \delta \max(f(x), f(x + \delta))$;
6 Faça $\hat{A}_L = \hat{A}_L + \delta \min(f(x), f(x + \delta))$;

7 Incremente $x = x + \delta$;
8 Se $x < b$, volte para a linha 5;
9 Se $\hat{A}_U - \hat{A}_L < 2\varepsilon$, retorne $\frac{\hat{A}_U + \hat{A}_L}{2}$;
10 Faça $n = n + 10$ e volte para a linha 2.

Para terminar, vejamos um resultado interessante. Sejam $\hat{A}_{Li} = \delta \min(f(x_{i-1}), f(x_i))$ e $\hat{A}_{Ui} = \delta \max(f(x_{i-1}), f(x_i))$ as áreas dos i-ésimos retângulos abaixo e acima da curva de f, respectivamente. Veja que o valor médio entre essas duas áreas pode ser calculado da seguinte maneira:

$$\begin{aligned}
\frac{\hat{A}_{Li} + \hat{A}_{Ui}}{2} &= \frac{1}{2}\left(\delta \min(f(x_{i-1}), f(x_i)) + \delta \max(f(x_{i-1}), f(x_i))\right) \\
&= \frac{\delta}{2}\left(\min(f(x_{i-1}), f(x_i)) + \max(f(x_{i-1}), f(x_i))\right) \\
&= \frac{\delta}{2}\left(f(x_{i-1}) + f(x_i)\right)
\end{aligned}$$

O valor $\frac{\delta}{2}(f(x_{i-1}) + f(x_i))$ é exatamente a área do trapézio de base $[x_{i-1}, x_i]$ e alturas $f(x_{i-1})$ e $f(x_i)$. Logo, a aproximação para a área dada pela média entre as áreas dos retângulos acima e abaixo da curva de f é igual à aproximação dada pela área do trapézio, ilustrada na Figura 13.3.

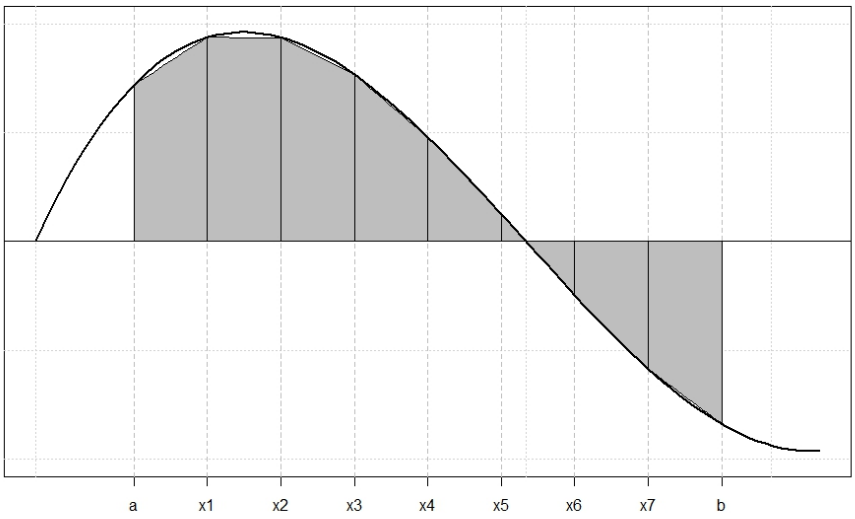


Figura 13.3: Aproximação por trapézios

Veja que para o mesmo valor de n, no caso do exemplo das figuras $n = 8$, a aproximação pela soma das áreas dos trapézio, Figura 13.3, é bem melhor que a aproximação pela somas das áreas dos retângulos, Figura 13.2.

Exercícios - 13ª Semana

13.1 Primeiro vamos aplicar o método em uma função que sabemos fazer as contas na mão, usando apenas áreas de figuras geométricas. Seja $f(x) = |x - 3| - 2$.

a) Na mão, sem usar o computador, faça um esboço do gráfico de f e hachure a área entre a curva de f e o eixo horizontal, limitada no intervalo [0,5].

b) Ainda na mão, calcule o valor da área hachurada. Lembre-se de considerar sinal positivo para área acima do eixo e sinal negativo para área abaixo dele. Esse é o valor que queremos aproximar.

c) Implemente uma função que recebe como entrada o valor n e retorna uma aproximação para a área hachurada dada pela a soma das áreas dos n retângulos. Escolha como quiser a definição da altura dos retângulos: valor da função no ponto à esquerda; valor da função no ponto à direita, maior valor da função; menor valor da função.
OBS: Nessa questão ainda não é para fazer o algoritmo visto em sala de aula, que retorna a aproximação dada pela média ente a soma das áreas dos retângulos acima e abaixo da curva de f.

d) Usando a função implementadas no item acima, encontre uma aproximação para a área hachurada com

- 50 retângulos.
- 100 retângulos.
- 150 retângulos.

e compare os resultados das 3 aproximações obtidas com o valor exato para a área encontrado no item b.

13.2 Implemente o algoritmo visto em sala de aula que recebe como entrada uma função f, o erro ε e valores $a, b \in \mathcal{D}(f)$ e retorna uma aproximação para a área entre a curva de f e o eixo horizontal, limitada no intervalo $[a, b]$, com erro menor que ε. A sua implementação será testada nas questões a seguir.

13.3 Agora vamos aplicar a implementação feita no exercício 13.2 para encontrar aproximações de áreas que só podemos achar o valor exato com recursos de cálculo.

a) Seja $f(x) = x^2 - x - 1$. Encontre uma aproximação para a área entre a curva de f e o eixo horizontal, limitada no intervalo $[-1, 1]$, com erro menor que $0,005$. Depois de encontrada a aproximação compare o resultado obtido com o valor exato, que é $-4/3$.

b) Seja $g(x) = xe^{x^2}$. Encontre uma aproximação para a área entre a curva de g e o eixo horizontal, limitada no intervalo $[0, 2]$, com erro menor que $0,01$. Depois de encontrada a aproximação compare o resultado obtido com o valor exato, que é $(e^4 - 1)/2$.
OBS: esse item pode levar tempo para rodar, no meu computador ele demorou mais de 1 minuto para terminar o método. Tenha paciência..

c) Seja $h(x) = \ln(x - 1)$. Encontre uma aproximação para a área entre a curva de h e o eixo horizontal, limitada no intervalo $[-2, -1]$, com erro menor que $0,0001$. Depois de encontrada a aproximação compare o resultado obtido com o valor exato, que é $\ln(1/2)$.

13.4 Modifique a função implementada no item 13.2 de forma que ela passe a retornar uma lista com dois objetos. O primeiro objeto da lista é uma aproximação para a área entre a curva de f e o eixo horizontal, limitada no intervalo $[a, b]$, com erro menor que ε. O segundo objeto da lista é o número de retângulos usados para conseguir a aproximação, isto é, o valor final de n.

13.5 Usando a função implementada no item 13.4, encontre o número de retângulos usados em cada uma das aproximações encontradas no item 13.3. Veja que o número de retângulos não depende somente do erro ε, mas também da função para a qual o método está sendo aplicado.