

# MAC 113 – Introdução à Ciência da Computação

Aula 28

---

Nelson Lago

1º/2025



**Previously on MAC113...**

Programar envolve

- 1 **Compreender um problema em termos computacionais**
- 2 **Definir como esse problema pode ser solucionado (*algoritmo*)**
  - ▶ Sequência **finita** de passos **bem definidos** baseados em um conjunto limitado (**vocabulário**) de operações possíveis
    - » *O algoritmo é abstrato (como a planta de um prédio ou uma receita de bolo)*
- 3 **Implementar o algoritmo em uma linguagem de programação**
  - ▶ Gerando um *programa* que pode ser *executado* para solucionar o problema
- 4 **Testar o programa**

Para ser útil, um programa geralmente

❶ **Obtém dados**

❷ **“Faz alguma coisa” com esses dados**

- ▶ Gerando um resultado

❸ **“Faz alguma coisa” com esse resultado**

- ▶ Mostra para o usuário

- ▶ **Utiliza como dado para fazer outra coisa**

**Mas como criar o algoritmo “certo”?**

- 1 Usando ou adaptando um algoritmo que já existe
- 2 Adaptando ideias de outros contextos
- 3 Conhecendo técnicas comuns (ou seja, estudo e prática)



Doctor Strange (2016)  
dir. Scott Derrickson

# Busca linear

Escreva uma função que recebe um vector de números `vec` e um número `n` e informa se o número está ou não no vector (sem usar `%in%`)

```
pertence <- function(vec, n) {  
  for (val in vec) {  
    if (val == n) { return (TRUE) }  
  }  
  return (FALSE)  
}
```

# Busca linear

Escreva uma função que recebe um vector **ordenado** de números `vec` e um número `n` e informa se o número está ou não no vector

```
pertence <- function(vec, n) {  
  for (val in vec) {  
    if (val > n) { return (FALSE) }  
    else if (val == n) { return (TRUE) }  
  }  
  return (FALSE)  
}
```

**Mas é assim que procuramos uma palavra  
em um dicionário?!?!**

Quando procuramos em um dicionário:

- **Abrimos o dicionário “em algum lugar perto do meio”**
  - ▶ lh, é antes → abre “em algum lugar do meio” entre o começo e a página atual
  - ▶ lh, é depois → abre “em algum lugar do meio” entre a página atual e o fim
- **No caso do dicionário, sabemos o mínimo (“a”) e o máximo (“z”) da lista, então não abrimos “no meio”, mas tentamos “chutar” uma página próxima**
- **Numa lista de números genérica, não temos essa “dica”, então o melhor é olhar o “meio” mesmo**

Vamos procurar pelo número 23

Vamos procurar pelo número 23

1	3	7	14	21	22	23	26	27	30	31
---	---	---	----	----	----	----	----	----	----	----

Vamos procurar pelo número 23

1	3	7	14	21	22	23	26	27	30	31
---	---	---	----	----	----	----	----	----	----	----



Vamos procurar pelo número 23

1	3	7	14	21	22	23	26	27	30	31
---	---	---	----	----	----	----	----	----	----	----

Vamos procurar pelo número 23

1	3	7	14	21	22	23	26	27	30	31
---	---	---	----	----	----	----	----	----	----	----



Vamos procurar pelo número 23

1	3	7	14	21	22	23	26	27	30	31
---	---	---	----	----	----	----	----	----	----	----

Vamos procurar pelo número 23

1	3	7	14	21	22	23	26	27	30	31
---	---	---	----	----	----	----	----	----	----	----



Vamos procurar pelo número 23

1	3	7	14	21	22	23	26	27	30	31
---	---	---	----	----	----	----	----	----	----	----

# Busca binária

Escreva uma função que recebe um vector **ordenado** de números `vec` e um número `n` e informa se o número está ou não no vector

```
pertence <- function(vec, n) {  
  i <- 1  
  j <- length(vec)  
  while (i <= j) {  
    meio <- (i + j) %/% 2  
    if (n == vec[meio]) { return (TRUE) }  
    else if (n > vec[meio]) { i <- meio + 1 }  
    else { j <- meio - 1 }  
  }  
  return (FALSE)  
}
```

# Busca binária

Escreva uma função que recebe um vector **ordenado** de números `vec` e um número `n` e informa se o número está ou não no vector

```
pertence <- function(vec, n) {  
  while (length(vec) > 1) {  
    meio <- length(vec) %% 2  
    if (n == vec[meio]) { return (TRUE) }  
    if (n < vec[meio]) { vec <- vec[seq_len(meio - 1)] }  
    else { vec <- vec[-seq_len(meio)] }  
  }  
  if (length(vec) == 1) { return (n == vec[1]) }  
  return (FALSE)  
}
```

## Exercício — adivinhando números 3/4

Escreva um programa que tenta adivinhar um número de 1 a 10 escolhido pelo usuário: o usuário responde “<”, “>” ou “=” para indicar se o número correto é maior, menor ou igual ao número gerado pelo computador (repetições até atingir um resultado)

```
library(glue)
main <- function() {
  cat("Escolha um número de 1 a 10, vou tentar adivinhá-lo!", "\n")
  resposta <- "x" # qualquer coisa diferente de "="
  while (resposta != "=") {
    chute <- sample(1:10, 1)
    resposta <- readline(glue('Será {chute}? responda com "<", ">" ou "=" '))
  }
  cat("Aêh, sou vidente!", "\n")
}
main()
```

## Exercício — adivinhando números 4/4

Melhore o programa anterior, fazendo uso das informações de maior/menor fornecidas pelo usuário

```
cat("Escolha um número de 1 a 10, vou tentar adivinhá-lo!", "\n")
resposta <- "x" # qualquer coisa diferente de "="
menor <- 1
maior <- 10
while (resposta != "=") {
  chute <- sample(menor:maior, 1)
  resposta <- readline(glue('Será {chute}??; responda com "<", ">" ou "=" '))
  if (resposta == ">") {
    menor <- chute + 1
  }
  if (resposta == "<") {
    maior <- chute - 1
  }
}
cat("Aêh, sou vidente!", "\n")
```

# Divisão e conquista

# Divisão e conquista

Você sabe somar mais de dois números de uma vez?

$$1 + 2 + 3 + 4 = (1 + 2) + 3 + 4 = 3 + 3 + 4 = (3 + 3) + 4 = 6 + 4 = 10$$

Imagine que o computador só é capaz de fazer uma soma de cada vez; faça uma função que soma uma lista de números

```
soma <- function(v) {  
  s <- 0  
  for (n in v) { s <- s + n }  
  return (s)  
}
```

**Pega um por um e soma**

(como no exemplo  $1 + 2 + 3 + 4$  acima)

# Divisão e conquista

Mas podemos pensar um pouco diferente:

```
somatório <- function(l) {  
  if (length(l) == 0) { return (0) }  
  if (length(l) == 1) { return (l[1]) }  
  if (length(l) == 2) { return (l[1] + l[2]) }  
  if (length(l) >= 3) { return (l[1] + somatório(l[2:length(l)])) }  
}
```

$1 + 2 + 3 + 4 = 1 + (2 + 3 + 4) = 1 + (2 + (3 + 4)) = 1 + (2 + 7) = 1 + 9 = 10$

**A soma de n números é a soma do primeiro com a soma do restante**



# Divisão e conquista

```
somatório <- function(l) {  
  if (length(l) == 0) { return (0) }  
  resto <- seq_len(length(l) - 1) + 1  
  return (l[1] + somatório(l[resto]))  
}
```

```
somatório <- function (l) {  
  resto <- seq_len(length(l) - 1) + 1  
  return (l[1] + somatório(l[resto]))  
}
```

- **Não dá para fazer chamadas recursivas para sempre!**

- ▶ É preciso haver ao menos um caso em que **não** fazemos a chamada recursiva
- ▶ É preciso garantir que **sempre** vamos chegar nesse caso

# Busca binária – divisão e conquista

```
pertence <- function(vec, n) {  
  i <- 1  
  j <- length(vec)  
  while (i <= j) {  
    meio <- (i + j) %% 2  
    if (n == vec[meio]) { return (TRUE) }  
    else if (n > vec[meio]) { i <- meio + 1 }  
    else { j <- meio - 1 }  
  }  
  return (FALSE)  
}
```

## Busca binária – divisão e conquista

Escreva uma função que recebe **dois** vectors ordenados de números  $V_a$ ,  $V_b$  e um número  $n$  e informa se o número está ou não em um deles

## Busca binária – divisão e conquista

Escreva uma função que recebe **dois** vectors ordenados de números  $V_a$ ,  $V_b$  e um número  $n$  e informa se o número está ou não em um deles

```
pertences <- function(Va, Vb, n) {  
  
}
```

## Busca binária – divisão e conquista

Escreva uma função que recebe **dois** vectors ordenados de números  $V_a$ ,  $V_b$  e um número  $n$  e informa se o número está ou não em um deles

```
pertences <- function(Va, Vb, n) {  
  return (pertence(Va, n) || pertence(Vb, n))  
}
```

## Busca binária – divisão e conquista

Mas uma lista com mais de um elemento é igual a duas sub-listas!

```
pertence(L, n)
```

↔

```
pertence(L[1:(length(L)%/2)], n) ||  
pertence(L[length(L)%/2 + 1:length(L)], n)
```

# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

```
pertence <- function(vec, n) {  
  i <- 1  
  j <- length(vec)  
  while (i <= j) {  
    meio <- (i + j) %% 2  
    if (n == vec[meio]) { return (TRUE) }  
    else if (n > vec[meio]) { i <- meio + 1 }  
    else { j <- meio - 1 }  
  }  
  return (FALSE)  
}
```

## Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

```
pertence <- function(vec, n) {
```

```
}
```

# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

```
pertence <- function(vec, n) {  
  
  meio <- length(vec) %/% 2  
  return (pertence(vec[1:meio], n) || pertence(vec[(meio + 1):length(vec)], n))  
}
```

# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

```
pertence <- function(vec, n) {  
  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %/% 2  
  return (pertence(vec[1:meio], n) || pertence(vec[(meio + 1):length(vec)], n))  
}
```

# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

```
pertence <- function(vec, n) {  
  if (length(vec) == 0) { return (FALSE) }  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %/% 2  
  return (pertence(vec[1:meio], n) || pertence(vec[(meio + 1):length(vec)], n))  
}
```

# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

```
pertence <- function(vec, n) {  
  if (length(vec) == 0) { return (FALSE) }  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %/% 2  
  return (pertence(vec[1:meio], n) || pertence(vec[(meio + 1):length(vec)], n))  
}
```



# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

```
pertence <- function(vec, n) {  
  if (length(vec) == 0) { return (FALSE) }  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %/% 2  
  return (pertence(vec[1:meio], n) || pertence(vec[(meio + 1):length(vec)], n))  
}
```



# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

```
pertence <- function(vec, n) {  
  if (length(vec) == 0) { return (FALSE) }  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %% 2  
  return (pertence(vec[1:meio], n) || pertence(vec[(meio + 1):length(vec)], n))  
}
```



OU

# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

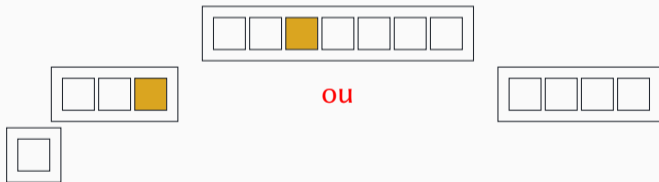
```
pertence <- function(vec, n) {  
  if (length(vec) == 0) { return (FALSE) }  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %/% 2  
  return (pertence(vec[1:meio], n) || pertence(vec[(meio + 1):length(vec)], n))  
}
```



# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

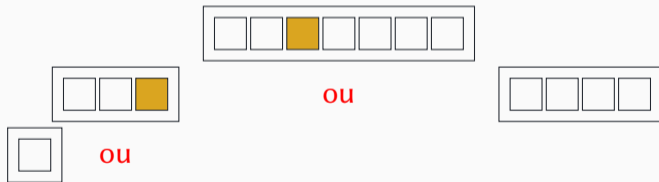
```
pertence <- function(vec, n) {  
  if (length(vec) == 0) { return (FALSE) }  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %/% 2  
  return (pertence(vec[1:meio], n) || pertence(vec[(meio + 1):length(vec)], n))  
}
```



# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

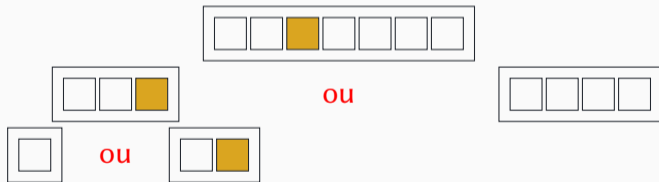
```
pertence <- function(vec, n) {  
  if (length(vec) == 0) { return (FALSE) }  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %/% 2  
  return (pertence(vec[1:meio], n) || pertence(vec[(meio + 1):length(vec)], n))  
}
```



# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

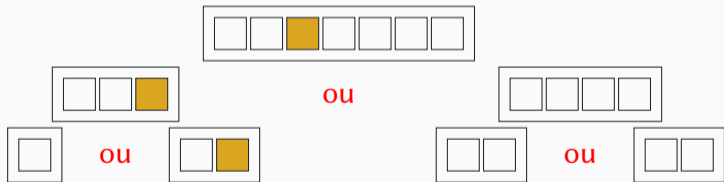
```
pertence <- function(vec, n) {  
  if (length(vec) == 0) { return (FALSE) }  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %/% 2  
  return (pertence(vec[1:meio], n) || pertence(vec[(meio + 1):length(vec)], n))  
}
```



# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

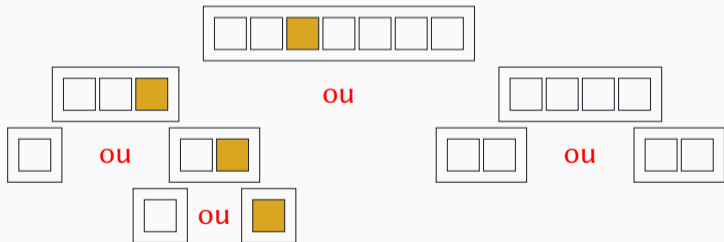
```
pertence <- function(vec, n) {  
  if (length(vec) == 0) { return (FALSE) }  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %% 2  
  return (pertence(vec[1:meio], n) || pertence(vec[(meio + 1):length(vec)], n))  
}
```



# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

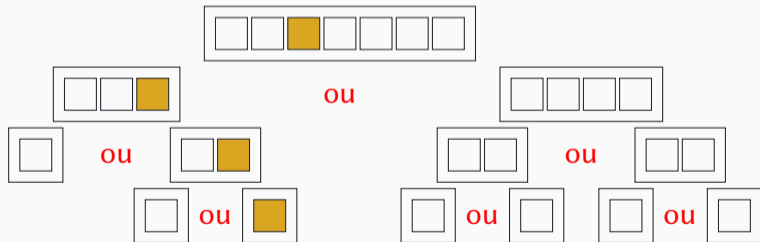
```
pertence <- function(vec, n) {  
  if (length(vec) == 0) { return (FALSE) }  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %/% 2  
  return (pertence(vec[1:meio], n) || pertence(vec[(meio + 1):length(vec)], n))  
}
```



# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

```
pertence <- function(vec, n) {  
  if (length(vec) == 0) { return (FALSE) }  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %/% 2  
  return (pertence(vec[1:meio], n) || pertence(vec[(meio + 1):length(vec)], n))  
}
```



# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

```
pertence <- function(vec, n) {  
  if (length(vec) == 0) { return (FALSE) }  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %/% 2  
  return (pertence(vec[1:meio], n) || pertence(vec[(meio + 1):length(vec)], n))  
}
```

# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

```
pertence <- function(vec, n) {  
  if (length(vec) == 0) { return (FALSE) }  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %/% 2  
  if (vec[meio] > n) { return (pertence(vec[1:meio], n)) }  
  return (pertence(vec[(meio + 1):length(vec)], n))  
}
```

# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

```
pertence <- function(vec, n) {  
  if (length(vec) == 0) { return (FALSE) }  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %/% 2  
  if (vec[meio] > n) { return (pertence(vec[1:meio], n)) }  
  return (pertence(vec[(meio + 1):length(vec)], n))  
}
```



# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

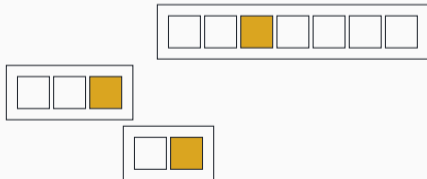
```
pertence <- function(vec, n) {  
  if (length(vec) == 0) { return (FALSE) }  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %% 2  
  if (vec[meio] > n) { return (pertence(vec[1:meio], n)) }  
  return (pertence(vec[(meio + 1):length(vec)], n))  
}
```



# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

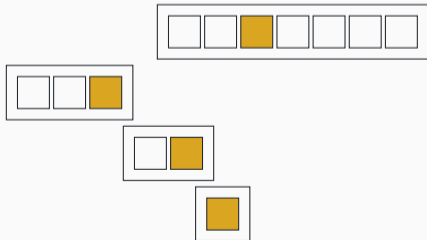
```
pertence <- function(vec, n) {  
  if (length(vec) == 0) { return (FALSE) }  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %/% 2  
  if (vec[meio] > n) { return (pertence(vec[1:meio], n)) }  
  return (pertence(vec[(meio + 1):length(vec)], n))  
}
```



# Busca binária – divisão e conquista

Escreva uma versão recursiva da função `pertence()`.

```
pertence <- function(vec, n) {  
  if (length(vec) == 0) { return (FALSE) }  
  if (length(vec) == 1) { return (vec[1] == n) }  
  meio <- length(vec) %/% 2  
  if (vec[meio] > n) { return (pertence(vec[1:meio], n)) }  
  return (pertence(vec[(meio + 1):length(vec)], n))  
}
```



# Recursão

- Recursão é útil quando você

- **Recursão é útil quando você**
  - ▶ Sabe como resolver o problema quando ele é “pequeno”

- **Recursão é útil quando você**
  - ▶ Sabe como resolver o problema quando ele é “pequeno”
  - ▶ Sabe como dividir o problema “grande” em partes menores

- **Recursão é útil quando você**
  - ▶ Sabe como resolver o problema quando ele é “pequeno”
  - ▶ Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

- **Recursão é útil quando você**
  - ▶ Sabe como resolver o problema quando ele é “pequeno”
  - ▶ Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*
  - ▶ E sabe “re-montar” o problema grande a partir das partes menores

- **Recursão é útil quando você**
  - ▶ Sabe como resolver o problema quando ele é “pequeno”
  - ▶ Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*
  - ▶ E sabe “re-montar” o problema grande a partir das partes menores
- **Recursões são inspiradas nas demonstrações por indução da matemática**

- **Recursão é útil quando você**
  - ▶ Sabe como resolver o problema quando ele é “pequeno”
  - ▶ Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*
  - ▶ E sabe “re-montar” o problema grande a partir das partes menores
- **Recursões são inspiradas nas demonstrações por indução da matemática**
- **É essencial definir quando a recursão para!**

- **Recursão é útil quando você**
  - ▶ Sabe como resolver o problema quando ele é “pequeno”
  - ▶ Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*
  - ▶ E sabe “re-montar” o problema grande a partir das partes menores
- **Recursões são inspiradas nas demonstrações por indução da matemática**
- **É essencial definir quando a recursão para!**
  - ▶ Ou seja, o caso “pequeno” que você já sabe resolver

- **Recursão é útil quando você**
  - ▶ Sabe como resolver o problema quando ele é “pequeno”
  - ▶ Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*
  - ▶ E sabe “re-montar” o problema grande a partir das partes menores
- **Recursões são inspiradas nas demonstrações por indução da matemática**
- **É essencial definir quando a recursão para!**
  - ▶ Ou seja, o caso “pequeno” que você já sabe resolver
    - » *(equivalente à base da recursão)*

- **Recursão é útil quando você**
  - ▶ Sabe como resolver o problema quando ele é “pequeno”
  - ▶ Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*
  - ▶ E sabe “re-montar” o problema grande a partir das partes menores
- **Recursões são inspiradas nas demonstrações por indução da matemática**
- **É essencial definir quando a recursão para!**
  - ▶ Ou seja, o caso “pequeno” que você já sabe resolver
    - » *(equivalente à base da recursão)*
  - ▶ Às vezes há mais de um caso “pequeno”, como no exemplo anterior

# Recursão – mesclando listas

Escreva uma versão recursiva da função `mescla()`

```
mescla <- function(v1, v2) {  
  tudo <- integer(length(v1) + length(v2))  
  i <- 1  
  j <- 1  
  k <- 1  
  while (i <= length(v1) && j <= length(v2)) {  
    if (v1[i] < v2[j]) {  
      tudo[k] <- v1[i]  
      i <- i + 1  
    } else {  
      tudo[k] <- v2[j]  
      j <- j + 1  
    }  
    k <- k + 1  
  }  
}
```

```
while (i <= length(v1)) {  
  tudo[k] <- v1[i]  
  i <- i + 1  
  k <- k + 1  
}  
while (j <= length(v2)) {  
  tudo[k] <- v2[j]  
  j <- j + 1  
  k <- k + 1  
}  
return (tudo)  
}
```

# Recursão – mesclando listas

Escreva uma versão recursiva da função `mescla()`

```
mescla <- function(v1, v2) {
```

```
}
```





# Recursão – mesclando listas

Escreva uma versão recursiva da função `mescla()`

```
mescla <- function(v1, v2) {  
  if (length(v1) == 0) { return (v2) }  
  if (length(v2) == 0) { return (v1) }  
  
}
```

# Recursão – mesclando listas

Escreva uma versão recursiva da função `mescla()`

```
mescla <- function(v1, v2) {  
  if (length(v1) == 0) { return (v2) }  
  if (length(v2) == 0) { return (v1) }  
  
  return (novo)  
}
```

# Recursão – mesclando listas

Escreva uma versão recursiva da função `mescla()`

```
mescla <- function(v1, v2) {  
  if (length(v1) == 0) { return (v2) }  
  if (length(v2) == 0) { return (v1) }  
  if (v1[1] < v2[1]) {  
  
  } else {  
  
  }  
  return (novo)  
}
```

# Recursão – mesclando listas

Escreva uma versão recursiva da função `mescla()`

```
mescla <- function(v1, v2) {  
  if (length(v1) == 0) { return (v2) }  
  if (length(v2) == 0) { return (v1) }  
  if (v1[1] < v2[1]) {  
    novo <-  
  } else {  
  
  }  
  return (novo)  
}
```

# Recursão – mesclando listas

Escreva uma versão recursiva da função `mescla()`

```
mescla <- function(v1, v2) {  
  if (length(v1) == 0) { return (v2) }  
  if (length(v2) == 0) { return (v1) }  
  if (v1[1] < v2[1]) {  
    novo <- c(                )  
  } else {  
  
  }  
  return (novo)  
}
```

# Recursão – mesclando listas

Escreva uma versão recursiva da função `mescla()`

```
mescla <- function(v1, v2) {  
  if (length(v1) == 0) { return (v2) }  
  if (length(v2) == 0) { return (v1) }  
  if (v1[1] < v2[1]) {  
    novo <- c(v1[1]                )  
  } else {  
  
  }  
  return (novo)  
}
```

# Recursão – mesclando listas

Escreva uma versão recursiva da função `mescla()`

```
mescla <- function(v1, v2) {  
  if (length(v1) == 0) { return (v2) }  
  if (length(v2) == 0) { return (v1) }  
  if (v1[1] < v2[1]) {  
    novo <- c(v1[1], mescla(  
  } else {  
  
  }  
  return (novo)  
}
```

# Recursão – mesclando listas

Escreva uma versão recursiva da função `mescla()`

```
mescla <- function(v1, v2) {  
  if (length(v1) == 0) { return (v2) }  
  if (length(v2) == 0) { return (v1) }  
  if (v1[1] < v2[1]) {  
    novo <- c(v1[1], mescla(      v2))  
  } else {  
  
  }  
  return (novo)  
}
```

# Recursão – mesclando listas

Escreva uma versão recursiva da função `mescla()`

```
mescla <- function(v1, v2) {  
  if (length(v1) == 0) { return (v2) }  
  if (length(v2) == 0) { return (v1) }  
  if (v1[1] < v2[1]) {  
    novo <- c(v1[1], mescla(v1[ ], v2))  
  } else {  
  
  }  
  return (novo)  
}
```

# Recursão – mesclando listas

Escreva uma versão recursiva da função `mescla()`

```
mescla <- function(v1, v2) {  
  if (length(v1) == 0) { return (v2) }  
  if (length(v2) == 0) { return (v1) }  
  if (v1[1] < v2[1]) {  
    novo <- c(v1[1], mescla(v1[-1], v2))  
  } else {  
  
  }  
  return (novo)  
}
```

# Recursão – mesclando listas

Escreva uma versão recursiva da função `mescla()`

```
mescla <- function(v1, v2) {  
  if (length(v1) == 0) { return (v2) }  
  if (length(v2) == 0) { return (v1) }  
  if (v1[1] < v2[1]) {  
    novo <- c(v1[1], mescla(v1[-1], v2))  
  } else {  
    novo <- c(v2[1], mescla(v2[-1], v1))  
  }  
  return (novo)  
}
```

- Um algoritmo para ordenar uma lista de inteiros é o *mergesort*

- **Um algoritmo para ordenar uma lista de inteiros é o *mergesort***
  - ▶ (a desvantagem do *mergesort* é que ele precisa criar uma lista auxiliar durante o processamento, ocupando mais memória)

- **Um algoritmo para ordenar uma lista de inteiros é o *mergesort***
  - ▶ (a desvantagem do *mergesort* é que ele precisa criar uma lista auxiliar durante o processamento, ocupando mais memória)
- **Uma lista com zero ou um elementos sempre está ordenada**

- **Um algoritmo para ordenar uma lista de inteiros é o *mergesort***
  - ▶ (a desvantagem do *mergesort* é que ele precisa criar uma lista auxiliar durante o processamento, ocupando mais memória)
- **Uma lista com zero ou um elementos sempre está ordenada**
- **Para ordenar uma lista maior, basta**

- **Um algoritmo para ordenar uma lista de inteiros é o *mergesort***
  - ▶ (a desvantagem do *mergesort* é que ele precisa criar uma lista auxiliar durante o processamento, ocupando mais memória)
- **Uma lista com zero ou um elementos sempre está ordenada**
- **Para ordenar uma lista maior, basta**
  - 1 Dividir a lista na “metade”

- **Um algoritmo para ordenar uma lista de inteiros é o *mergesort***
  - ▶ (a desvantagem do *mergesort* é que ele precisa criar uma lista auxiliar durante o processamento, ocupando mais memória)
- **Uma lista com zero ou um elementos sempre está ordenada**
- **Para ordenar uma lista maior, basta**
  - 1 Dividir a lista na “metade”
  - 2 Ordenar cada uma das partes

- **Um algoritmo para ordenar uma lista de inteiros é o *mergesort***
  - ▶ (a desvantagem do *mergesort* é que ele precisa criar uma lista auxiliar durante o processamento, ocupando mais memória)
- **Uma lista com zero ou um elementos sempre está ordenada**
- **Para ordenar uma lista maior, basta**
  - 1 Dividir a lista na “metade”
  - 2 Ordenar cada uma das partes
  - 3 Juntar as duas partes mantendo a ordem

## Exercício – mergesort

Escreva uma função (recursiva) que implementa o algoritmo *mergesort*

## Exercício – mergesort

Escreva uma função (recursiva) que implementa o algoritmo *mergesort*

## Exercício – mergesort

Escreva uma função (recursiva) que implementa o algoritmo *mergesort*

```
mergesort <- function(vec) {
```

```
}
```



## Exercício – mergesort

Escreva uma função (recursiva) que implementa o algoritmo *mergesort*

```
mergesort <- function(vec) {  
  if (length(vec) <= 1) { return (vec) }  
  meio <- length(vec) %/% 2  
  esqd <- vec[1:meio]  
  drta <- vec[(meio + 1):length(vec)]  
  
}
```

## Exercício – mergesort

Escreva uma função (recursiva) que implementa o algoritmo *mergesort*

```
mergesort <- function(vec) {  
  if (length(vec) <= 1) { return (vec) }  
  meio <- length(vec) %/% 2  
  esqd <- vec[1:meio]  
  drta <- vec[(meio + 1):length(vec)]  
  esqd <- mergesort(esqd)  
  drta <- mergesort(drta)  
}
```

## Exercício – mergesort

Escreva uma função (recursiva) que implementa o algoritmo *mergesort*

```
mergesort <- function(vec) {  
  if (length(vec) <= 1) { return (vec) }  
  meio <- length(vec) %% 2  
  esqd <- vec[1:meio]  
  drta <- vec[(meio + 1):length(vec)]  
  esqd <- mergesort(esqd)  
  drta <- mergesort(drta)  
  return (mescla(esqd, drta))  
}
```

# Exercício – mergesort

Escreva uma função (recursiva) que implementa o algoritmo *mergesort*

```
mergesort <- function(vec) {  
  if (length(vec) <= 1) { return (vec) }  
  meio <- length(vec) %% 2  
  esqd <- vec[1:meio]  
  drta <- vec[(meio + 1):length(vec)]  
  esqd <- mergesort(esqd)  
  drta <- mergesort(drta)  
  return (mescla(esqd, drta))  
}
```



- **Maneiras diferentes de pensar repetições**

- Maneiras diferentes de pensar repetições
- Muitas vezes “dá na mesma”

- **Maneiras diferentes de pensar repetições**
- **Muitas vezes “dá na mesma”**
- **Às vezes, uma é (muito) mais simples de entender que a outra**

# Ordenação

Escreva uma função que recebe um vector de números e informa se ele está ordenado



# Ordenação

Escreva uma função que recebe um vector de números e informa se ele está ordenado

```
ordenado <- function(vec) {  
  
  return (TRUE)  
}
```

# Ordenação

Escreva uma função que recebe um vector de números e informa se ele está ordenado

```
ordenado <- function(vec) {  
  for (i in seq_len(length(vec) - 1)) {  
  
  }  
  return (TRUE)  
}
```

# Ordenação

Escreva uma função que recebe um vector de números e informa se ele está ordenado

```
ordenado <- function(vec) {  
  for (i in seq_len(length(vec) - 1)) {  
    if (vec[i] > vec[i+1]) {  
      return (FALSE)  
    }  
  }  
  return (TRUE)  
}
```

# Ordenação

Escreva uma função que recebe um vector de números e informa se ele está ordenado

```
ordenado <- function(vec) {  
  for (i in seq_len(length(vec) - 1)) {  
    if (vec[i] > vec[i + 1]) {  
      return (FALSE)  
    }  
  }  
  return (TRUE)  
}
```

# Ordenação

Escreva uma função que recebe um vector de números e informa se ele está ordenado

```
ordenado <- function(vec) {  
  for (i in seq_len(length(vec) - 1)) {  
    if (vec[i] > vec[i + 1]) { return (FALSE) }  
  }  
  return (TRUE)  
}
```

# Ordenação

Escreva uma função que recebe um vector de números e informa se ele está ordenado

```
ordenado <- function(vec) {  
  for (i in seq_len(length(vec) - 1)) {  
    if (vec[i] > vec[i + 1]) { return (FALSE) }  
  }  
  return (TRUE)  
}
```

# Ordenação — ideias (bozosort)

E como ordenar uma lista?

## Ordenação — ideias (bozosort)

E como ordenar uma lista?

O computador é rápido, certo? Se embaralharmos a lista várias vezes seguidas, em algum momento a lista vai acabar ordenada.



## Ordenação — ideias (bozosort)

E como ordenar uma lista?

O computador é rápido, certo? Se embaralharmos a lista várias vezes seguidas, em algum momento a lista vai acabar ordenada.

```
vec <- c(7,4,8,3,1,9)
cat("Ordenado:", bozosort(vec), "\n")
```

# Ordenação — ideias (bozosort)

E como ordenar uma lista?

O computador é rápido, certo? Se embaralharmos a lista várias vezes seguidas, em algum momento a lista vai acabar ordenada.

```
bozosort <- function(vec) {  
  
  
  
  
  
  
}  
vec <- c(7,4,8,3,1,9)  
cat("Ordenado:", bozosort(vec), "\n")
```

# Ordenação — ideias (bozosort)

E como ordenar uma lista?

O computador é rápido, certo? Se embaralharmos a lista várias vezes seguidas, em algum momento a lista vai acabar ordenada.

```
bozosort <- function(vec) {  
  
    return(vec)  
}  
vec <- c(7,4,8,3,1,9)  
cat("Ordenado:", bozosort(vec), "\n")
```

# Ordenação — ideias (bozosort)

E como ordenar uma lista?

O computador é rápido, certo? Se embaralharmos a lista várias vezes seguidas, em algum momento a lista vai acabar ordenada.

```
bozosort <- function(vec) {  
  while (! ordenado(vec)) {  
    vec <- sample(vec)  
  }  
  return(vec)  
}  
vec <- c(7,4,8,3,1,9)  
cat("Ordenado:", bozosort(vec), "\n")
```

## Ordenação — ideias (bubblesort)

Se a lista está fora de ordem, existe algum elemento de índice  $i$  tal que  $\text{elemento}[i] > \text{elemento}[i+1]$ . Se “consertarmos” todos esses casos, a lista fica ordenada.



## Ordenação — ideias (bubblesort)

Se a lista está fora de ordem, existe algum elemento de índice  $i$  tal que `elemento[i] > elemento[i+1]`. Se “consertarmos” todos esses casos, a lista fica ordenada.

```
bubblesort <- function(vec) {
```

```
}
```



## Ordenação — ideias (bubblesort)

Se a lista está fora de ordem, existe algum elemento de índice  $i$  tal que  $\text{elemento}[i] > \text{elemento}[i+1]$ . Se “consertarmos” todos esses casos, a lista fica ordenada.

```
bubblesort <- function(vec) {  
  while (! ordenado(vec)) {  
    for (i in seq_len(length(vec) - 1)) {  
  
      }  
    }  
  }  
}
```

## Ordenação — ideias (bubblesort)

Se a lista está fora de ordem, existe algum elemento de índice  $i$  tal que  $\text{elemento}[i] > \text{elemento}[i+1]$ . Se “consertarmos” todos esses casos, a lista fica ordenada.

```
bubblesort <- function(vec) {  
  while (! ordenado(vec)) {  
    for (i in seq_len(length(vec) - 1)) {  
      if (vec[i] > vec[i + 1]) {  
  
        }  
    }  
  }  
}
```

## Ordenação — ideias (bubblesort)

Se a lista está fora de ordem, existe algum elemento de índice  $i$  tal que  $\text{elemento}[i] > \text{elemento}[i+1]$ . Se “consertarmos” todos esses casos, a lista fica ordenada.

```
bubblesort <- function(vec) {  
  while (! ordenado(vec)) {  
    for (i in seq_len(length(vec) - 1)) {  
      if (vec[i] > vec[i + 1]) {  
        tmp <- vec[i]  
        vec[i] <- vec[i + 1]  
        vec[i + 1] <- tmp  
      }  
    }  
  }  
}
```

# Ordenação — ideias (selection sort)

Para ordenar um vector A:

“Encontra o próximo e coloca no fim de uma novo vector”

# Ordenação — ideias (selection sort)

Para ordenar um vector A:

“Encontra o próximo e coloca no fim de uma novo vector”

**1 Cria um vector B vazio**

# Ordenação — ideias (selection sort)

Para ordenar um vector A:

“Encontra o próximo e coloca no fim de uma novo vector”

- 1 Cria um vector B vazio
- 2 Encontra o menor elemento do vector A e move esse elemento para o final do vector B

# Ordenação — ideias (selection sort)

Para ordenar um vector A:

“Encontra o próximo e coloca no fim de uma novo vector”

- 1 Cria um vector B vazio
- 2 Encontra o menor elemento do vector A e move esse elemento para o final do vector B
- 3 Repete até o vector A estar vazio

## Exercício – selection sort

Escreva uma função que recebe um vector, encontra o menor elemento e devolve uma lista com dois itens: o menor elemento encontrado e uma cópia do vector original sem ele



## Exercício – selection sort

Escreva uma função que recebe um vector, encontra o menor elemento e devolve uma lista com dois itens: o menor elemento encontrado e uma cópia do vector original sem ele

```
encontra_menor <- function(vec) {
```

```
}
```



## Exercício – selection sort

Escreva uma função que recebe um vector, encontra o menor elemento e devolve uma lista com dois itens: o menor elemento encontrado e uma cópia do vector original sem ele

```
encontra_menor <- function(vec) {  
  pos <- 1  
  
  return (list(vec[pos], nvec))  
}
```

## Exercício – selection sort

Escreva uma função que recebe um vector, encontra o menor elemento e devolve uma lista com dois itens: o menor elemento encontrado e uma cópia do vector original sem ele

```
encontra_menor <- function(vec) {  
  pos <- 1  
  for (i in seq_len(length(vec))) {  
  
  }  
  
  return (list(vec[pos], nvec))  
}
```

## Exercício – selection sort

Escreva uma função que recebe um vector, encontra o menor elemento e devolve uma lista com dois itens: o menor elemento encontrado e uma cópia do vector original sem ele

```
encontra_menor <- function(vec) {  
  pos <- 1  
  for (i in seq_len(length(vec))) {  
    if (vec[i] < vec[pos]) { pos <- i }  
  }  
  
  return (list(vec[pos], nvec))  
}
```

## Exercício – selection sort

Escreva uma função que recebe um vector, encontra o menor elemento e devolve uma lista com dois itens: o menor elemento encontrado e uma cópia do vector original sem ele

```
encontra_menor <- function(vec) {  
  pos <- 1  
  for (i in seq_len(length(vec))) {  
    if (vec[i] < vec[pos]) { pos <- i }  
  }  
  parte1 <- seq_len(pos - 1)  
  
  return (list(vec[parte1], nvec))  
}
```

## Exercício – selection sort

Escreva uma função que recebe um vector, encontra o menor elemento e devolve uma lista com dois itens: o menor elemento encontrado e uma cópia do vector original sem ele

```
encontra_menor <- function(vec) {  
  pos <- 1  
  for (i in seq_len(length(vec))) {  
    if (vec[i] < vec[pos]) { pos <- i }  
  }  
  parte1 <- seq_len(pos - 1)  
  
  return (list(vec[pos], nvec))  
}
```

## Exercício – selection sort

Escreva uma função que recebe um vector, encontra o menor elemento e devolve uma lista com dois itens: o menor elemento encontrado e uma cópia do vector original sem ele

```
encontra_menor <- function(vec) {  
  pos <- 1  
  for (i in seq_len(length(vec))) {  
    if (vec[i] < vec[pos]) { pos <- i }  
  }  
  parte1 <- seq_len(pos - 1)  
  parte2 <- seq_len(length(vec) - pos)  
  
  return (list(vec[pos], nvec))  
}
```

## Exercício – selection sort

Escreva uma função que recebe um vector, encontra o menor elemento e devolve uma lista com dois itens: o menor elemento encontrado e uma cópia do vector original sem ele

```
encontra_menor <- function(vec) {  
  pos <- 1  
  for (i in seq_len(length(vec))) {  
    if (vec[i] < vec[pos]) { pos <- i }  
  }  
  parte1 <- seq_len(pos - 1)  
  parte2 <- seq_len(length(vec) - pos) + pos  
  
  return (list(vec[pos], nvec))  
}
```

## Exercício – selection sort

Escreva uma função que recebe um vector, encontra o menor elemento e devolve uma lista com dois itens: o menor elemento encontrado e uma cópia do vector original sem ele

```
encontra_menor <- function(vec) {  
  pos <- 1  
  for (i in seq_len(length(vec))) {  
    if (vec[i] < vec[pos]) { pos <- i }  
  }  
  parte1 <- seq_len(pos - 1)  
  parte2 <- seq_len(length(vec) - pos) + pos  
  nvec <-  
  return (list(vec[pos], nvec))  
}
```

## Exercício – selection sort

Escreva uma função que recebe um vector, encontra o menor elemento e devolve uma lista com dois itens: o menor elemento encontrado e uma cópia do vector original sem ele

```
encontra_menor <- function(vec) {  
  pos <- 1  
  for (i in seq_len(length(vec))) {  
    if (vec[i] < vec[pos]) { pos <- i }  
  }  
  parte1 <- seq_len(pos - 1)  
  parte2 <- seq_len(length(vec) - pos) + pos  
  nvec <- c(                )  
  return (list(vec[pos], nvec))  
}
```

## Exercício – selection sort

Escreva uma função que recebe um vector, encontra o menor elemento e devolve uma lista com dois itens: o menor elemento encontrado e uma cópia do vector original sem ele

```
encontra_menor <- function(vec) {  
  pos <- 1  
  for (i in seq_len(length(vec))) {  
    if (vec[i] < vec[pos]) { pos <- i }  
  }  
  parte1 <- seq_len(pos - 1)  
  parte2 <- seq_len(length(vec) - pos) + pos  
  nvec <- c(vec[parte1], vec[parte2])  
  return (list(vec[pos], nvec))  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção usando a função `encontra_menor()`







## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção usando a função `encontra_menor()`

```
selection_sort <- function(vec) {  
  novo <- integer(length(vec))  
  for (i in seq_len(length(novo))) {  
  
  
  
  
  
  
  }  
  return (novo)  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção usando a função `encontra_menor()`

```
selection_sort <- function(vec) {  
  novo <- integer(length(vec))  
  for (i in seq_len(length(novo))) {  
    tmp <- encontra_menor(vec)  
  
  }  
  return (novo)  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção usando a função `encontra_menor()`

```
selection_sort <- function(vec) {  
  novo <- integer(length(vec))  
  for (i in seq_len(length(novo))) {  
    tmp <- encontra_menor(vec)  
    menor <- tmp[[1]]  
  
  }  
  return (novo)  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção usando a função `encontra_menor()`

```
selection_sort <- function(vec) {  
  novo <- integer(length(vec))  
  for (i in seq_len(length(novo))) {  
    tmp <- encontra_menor(vec)  
    menor <- tmp[[1]]  
    novo[i] <- menor  
  }  
  return (novo)  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção usando a função `encontra_menor()`

```
selection_sort <- function(vec) {  
  novo <- integer(length(vec))  
  for (i in seq_len(length(novo))) {  
    tmp <- encontra_menor(vec)  
    menor <- tmp[[1]]  
    novo[i] <- menor  
    vec <- tmp[[2]]  
  }  
  return (novo)  
}
```

- **Mas nem precisa de dois vectors!**

- **Mas nem precisa de dois vectors!**
- **Basta criar uma divisão imaginária no vector**

- **Mas nem precisa de dois vectors!**
- **Basta criar uma divisão imaginária no vector**
  - ▶ O pedaço à esquerda da divisão é o vector “novo”, que está em ordem

- **Mas nem precisa de dois vectors!**
- **Basta criar uma divisão imaginária no vector**
  - ▶ O pedaço à esquerda da divisão é o vector “novo”, que está em ordem
  - ▶ O pedaço à direita é o vector “velho”, que não está em ordem

- **Mas nem precisa de dois vectors!**
- **Basta criar uma divisão imaginária no vector**
  - ▶ O pedaço à esquerda da divisão é o vector “novo”, que está em ordem
  - ▶ O pedaço à direita é o vector “velho”, que não está em ordem
  - ▶ A linha da divisão imaginária vai avançando a cada passo, de maneira que os elementos “saem” do vector desordenado e “entram” no vector ordenado

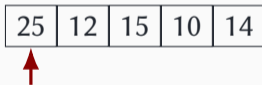
# Selection sort

*Selection sort*: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.

25	12	15	10	14
----	----	----	----	----

# Selection sort

*Selection sort*: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.



# Selection sort

*Selection sort*: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.



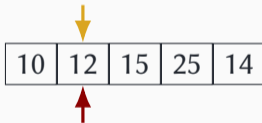
# Selection sort

*Selection sort*: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.



# Selection sort

*Selection sort*: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.



# Selection sort

*Selection sort*: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.



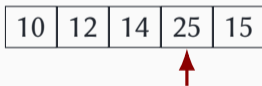
# Selection sort

*Selection sort*: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.



# Selection sort

*Selection sort*: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.



# Selection sort

*Selection sort*: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.



# Selection sort

*Selection sort*: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.



# Selection sort

*Selection sort*: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.



## Exercício – selection sort

Escreva uma função que recebe um vector e devolve o índice do menor elemento desse vector



## Exercício — selection sort

Escreva uma função que recebe um vector e devolve o índice do menor elemento desse vector

```
encontra_menor <- function(vec) {
```

```
}
```

## Exercício — selection sort

Escreva uma função que recebe um vector e devolve o índice do menor elemento desse vector

```
encontra_menor <- function(vec) {  
  pos <- 1  
  
  return (pos)  
}
```

## Exercício — selection sort

Escreva uma função que recebe um vector e devolve o índice do menor elemento desse vector

```
encontra_menor <- function(vec) {  
  pos <- 1  
  for (i in seq_len(length(vec))) {  
    if (vec[i] < vec[pos]) { pos <- i }  
  }  
  return (pos)  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção (“Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”) usando essa nova versão da função `encontra_menor()`

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção (“Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”) usando essa nova versão da função `encontra_menor()`

```
selection_sort <- function(vec) {
```

```
}
```



## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção (“Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”) usando essa nova versão da função `encontra_menor()`

```
selection_sort <- function(vec) {  
  for (i in seq_len(length(vec))) {  
  
  
  
  
  
  
  
  
  
  }  
  return(vec)  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção (“Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”) usando essa nova versão da função `encontra_menor()`

```
selection_sort <- function(vec) {  
  for (i in seq_len(length(vec))) {  
    jafoi <- i  
  
    }  
  return(vec)  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção (“Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”) usando essa nova versão da função `encontra_menor()`

```
selection_sort <- function(vec) {  
  for (i in seq_len(length(vec))) {  
    jafoi <- i - 1  
  
    }  
  return(vec)  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção (“Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”) usando essa nova versão da função `encontra_menor()`

```
selection_sort <- function(vec) {  
  for (i in seq_len(length(vec))) {  
    jafoi <- i - 1  
    falta <- length(vec)  
  
    }  
  return(vec)  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção (“Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”) usando essa nova versão da função `encontra_menor()`

```
selection_sort <- function(vec) {  
  for (i in seq_len(length(vec))) {  
    jafoi <- i - 1  
    falta <- length(vec) - jafoi  
  
    }  
  return(vec)  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção (“Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”) usando essa nova versão da função `encontra_menor()`

```
selection_sort <- function(vec) {  
  for (i in seq_len(length(vec))) {  
    jafoi <- i - 1  
    falta <- length(vec) - jafoi  
    pos <- encontra_menor(vec  
  
    )  
  
  }  
  return(vec)  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção (“Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”) usando essa nova versão da função `encontra_menor()`

```
selection_sort <- function(vec) {  
  for (i in seq_len(length(vec))) {  
    jafoi <- i - 1  
    falta <- length(vec) - jafoi  
    pos <- encontra_menor(vec[  
  
    ])  
  
  }  
  return(vec)  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção (“Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”) usando essa nova versão da função `encontra_menor()`

```
selection_sort <- function(vec) {  
  for (i in seq_len(length(vec))) {  
    jafoi <- i - 1  
    falta <- length(vec) - jafoi  
    pos <- encontra_menor(vec[seq_len(falta)      ])  
  
  }  
  return(vec)  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção (“Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”) usando essa nova versão da função `encontra_menor()`

```
selection_sort <- function(vec) {  
  for (i in seq_len(length(vec))) {  
    jafoi <- i - 1  
    falta <- length(vec) - jafoi  
    pos <- encontra_menor(vec[seq_len(falta) + jafoi])  
  
  }  
  return(vec)  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção (“Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”) usando essa nova versão da função `encontra_menor()`

```
selection_sort <- function(vec) {  
  for (i in seq_len(length(vec))) {  
    jafoi <- i - 1  
    falta <- length(vec) - jafoi  
    pos <- encontra_menor(vec[seq_len(falta) + jafoi])  
    pos <- pos + jafoi  
  
  }  
  return(vec)  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção (“Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”) usando essa nova versão da função `encontra_menor()`

```
selection_sort <- function(vec) {  
  for (i in seq_len(length(vec))) {  
    jafoi <- i - 1  
    falta <- length(vec) - jafoi  
    pos <- encontra_menor(vec[seq_len(falta) + jafoi])  
    pos <- pos + jafoi  
  
    vec[i] <- vec[pos]  
  
  }  
  return(vec)  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção (“Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”) usando essa nova versão da função `encontra_menor()`

```
selection_sort <- function(vec) {  
  for (i in seq_len(length(vec))) {  
    jafoi <- i - 1  
    falta <- length(vec) - jafoi  
    pos <- encontra_menor(vec[seq_len(falta) + jafoi])  
    pos <- pos + jafoi  
    tmp <- vec[i]  
    vec[i] <- vec[pos]  
  
  }  
  return(vec)  
}
```

## Exercício – selection sort

Implemente o algoritmo de ordenação por seleção (“Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”) usando essa nova versão da função `encontra_menor()`

```
selection_sort <- function(vec) {  
  for (i in seq_len(length(vec))) {  
    jafoi <- i - 1  
    falta <- length(vec) - jafoi  
    pos <- encontra_menor(vec[seq_len(falta) + jafoi])  
    pos <- pos + jafoi  
    tmp <- vec[i]  
    vec[i] <- vec[pos]  
    vec[pos] <- tmp  
  }  
  return(vec)  
}
```

# Complexidade computacional

- **Complexidade computacional**

# Complexidade computacional

- **Complexidade computacional**

- ▶ O “tamanho do esforço” que o computador deve fazer para resolver um problema de tamanho  $n$

# Complexidade computacional

- **Complexidade computacional**

- ▶ O “tamanho do esforço” que o computador deve fazer para resolver um problema de tamanho  $n$
- ▶ Geralmente, estamos preocupados com o tempo de execução, mas às vezes o “esforço” é o consumo de memória ou algum outro aspecto

# Complexidade computacional

- **Complexidade computacional**

- ▶ O “tamanho do esforço” que o computador deve fazer para resolver um problema de tamanho  $n$
- ▶ Geralmente, estamos preocupados com o tempo de execução, mas às vezes o “esforço” é o consumo de memória ou algum outro aspecto
- ▶ A ideia intuitiva é “**mais ou menos proporcional a**”

# Complexidade computacional

- **Complexidade computacional**

- ▶ O “tamanho do esforço” que o computador deve fazer para resolver um problema de tamanho  $n$
- ▶ Geralmente, estamos preocupados com o tempo de execução, mas às vezes o “esforço” é o consumo de memória ou algum outro aspecto
- ▶ A ideia intuitiva é “**mais ou menos proporcional a**”

- **Busca linear**

# Complexidade computacional

- **Complexidade computacional**

- ▶ O “tamanho do esforço” que o computador deve fazer para resolver um problema de tamanho  $n$
- ▶ Geralmente, estamos preocupados com o tempo de execução, mas às vezes o “esforço” é o consumo de memória ou algum outro aspecto
- ▶ A ideia intuitiva é “**mais ou menos proporcional a**”

- **Busca linear**

- ▶ O “tamanho do esforço” em termos do tempo de execução é “mais ou menos proporcional” ao tamanho da lista de entrada

# Complexidade computacional

- **Complexidade computacional**

- ▶ O “tamanho do esforço” que o computador deve fazer para resolver um problema de tamanho  $n$
- ▶ Geralmente, estamos preocupados com o tempo de execução, mas às vezes o “esforço” é o consumo de memória ou algum outro aspecto
- ▶ A ideia intuitiva é “**mais ou menos proporcional a**”

- **Busca linear**

- ▶ O “tamanho do esforço” em termos do tempo de execução é “mais ou menos proporcional” ao tamanho da lista de entrada
  - » Dizemos que o algoritmo é  **$O(n)$**

# Complexidade computacional

- **Complexidade computacional**

- ▶ O “tamanho do esforço” que o computador deve fazer para resolver um problema de tamanho  $n$
- ▶ Geralmente, estamos preocupados com o tempo de execução, mas às vezes o “esforço” é o consumo de memória ou algum outro aspecto
- ▶ A ideia intuitiva é “**mais ou menos proporcional a**”

- **Busca linear**

- ▶ O “tamanho do esforço” em termos do tempo de execução é “mais ou menos proporcional” ao tamanho da lista de entrada
  - » Dizemos que o algoritmo é  **$O(n)$**

- **Busca binária**

# Complexidade computacional

- **Complexidade computacional**

- ▶ O “tamanho do esforço” que o computador deve fazer para resolver um problema de tamanho  $n$
- ▶ Geralmente, estamos preocupados com o tempo de execução, mas às vezes o “esforço” é o consumo de memória ou algum outro aspecto
- ▶ A ideia intuitiva é “**mais ou menos proporcional a**”

- **Busca linear**

- ▶ O “tamanho do esforço” em termos do tempo de execução é “mais ou menos proporcional” ao tamanho da lista de entrada
  - » Dizemos que o algoritmo é  **$O(n)$**

- **Busca binária**

- ▶ O “tamanho do esforço” em termos do tempo de execução é “mais ou menos proporcional” ao número de vezes que podemos dividir a lista de entrada pela metade  $\rightarrow \log_2 n$

# Complexidade computacional

- **Complexidade computacional**

- ▶ O “tamanho do esforço” que o computador deve fazer para resolver um problema de tamanho  $n$
- ▶ Geralmente, estamos preocupados com o tempo de execução, mas às vezes o “esforço” é o consumo de memória ou algum outro aspecto
- ▶ A ideia intuitiva é “**mais ou menos proporcional a**”

- **Busca linear**

- ▶ O “tamanho do esforço” em termos do tempo de execução é “mais ou menos proporcional” ao tamanho da lista de entrada
  - » Dizemos que o algoritmo é  **$O(n)$**

- **Busca binária**

- ▶ O “tamanho do esforço” em termos do tempo de execução é “mais ou menos proporcional” ao número de vezes que podemos dividir a lista de entrada pela metade  $\rightarrow \log_2 n$ 
  - » Dizemos que o algoritmo é  **$O(\log(n))$**

# Complexidade computacional

- Em computadores modernos, na maior parte do tempo, a complexidade **não importa**

# Complexidade computacional

- Em computadores modernos, na maior parte do tempo, a complexidade **não importa**
  - ▶ Algoritmos com complexidade “ruim” só se tornam perceptivelmente lentos com valores “grandes” de  $n$

# Complexidade computacional

- Em computadores modernos, na maior parte do tempo, a complexidade **não importa**
  - ▶ Algoritmos com complexidade “ruim” só se tornam perceptivelmente lentos com valores “grandes” de  $n$
  - ▶ Nos problemas complexos que envolvem valores grandes de  $n$ , você provavelmente vai usar uma biblioteca pronta que implementa uma boa solução para o problema

# Complexidade computacional

- Em computadores modernos, na maior parte do tempo, a complexidade **não importa**
  - ▶ Algoritmos com complexidade “ruim” só se tornam perceptivelmente lentos com valores “grandes” de  $n$
  - ▶ Nos problemas complexos que envolvem valores grandes de  $n$ , você provavelmente vai usar uma biblioteca pronta que implementa uma boa solução para o problema
- **MAS!**

# Complexidade computacional

- Em computadores modernos, na maior parte do tempo, a complexidade **não importa**
  - ▶ Algoritmos com complexidade “ruim” só se tornam perceptivelmente lentos com valores “grandes” de  $n$
  - ▶ Nos problemas complexos que envolvem valores grandes de  $n$ , você provavelmente vai usar uma biblioteca pronta que implementa uma boa solução para o problema
- **MAS!**
- Às vezes é preciso levar a complexidade em conta

# Complexidade computacional

- **Em computadores modernos, na maior parte do tempo, a complexidade **não importa****
  - ▶ Algoritmos com complexidade “ruim” só se tornam perceptivelmente lentos com valores “grandes” de  $n$
  - ▶ Nos problemas complexos que envolvem valores grandes de  $n$ , você provavelmente vai usar uma biblioteca pronta que implementa uma boa solução para o problema
- **MAS!**
- **Às vezes é preciso levar a complexidade em conta**
  - ▶ E é bom saber que existem problemas para os quais não se conhecem soluções com complexidade “boa”

**bozosort:**

# Complexidade dos algoritmos de ordenação

bozosort: **NEM IDEIA!** 😂

**bozosort: NEM IDEIA!** 😂

- ▶ Não é determinístico (pode não terminar nunca)

**bozosort: NEM IDEIA!** 😂

- ▶ Não é determinístico (pode não terminar nunca)
- ▶ Eu **chuto** que a **probabilidade** de encontrar uma solução **em um determinado tempo** deve ser proporcional a  $O(n!)$

**bozosort: NEM IDEIA!** 😂

- ▶ Não é determinístico (pode não terminar nunca)
- ▶ Eu **chuto** que a **probabilidade** de encontrar uma solução **em um determinado tempo** deve ser proporcional a  $O(n!)$ 
  - » *Com dez elementos, tende a demorar vários segundos!*

**bubblesort:**

# Complexidade dos algoritmos de ordenação

## **bubblesort:**

- 1 Troca a posição de todos os elementos fora do lugar

# Complexidade dos algoritmos de ordenação

## **bubblesort:**

- 1 Troca a posição de todos os elementos fora do lugar (**proporcional a  $n$** )

# Complexidade dos algoritmos de ordenação

## **bubblesort:**

- 1 Troca a posição de todos os elementos fora do lugar (**proporcional a  $n$** )
- 2 Repete até todos os elementos estarem no lugar certo

# Complexidade dos algoritmos de ordenação

## **bubblesort:**

- 1 Troca a posição de todos os elementos fora do lugar (**proporcional a  $n$** )
- 2 Repete até todos os elementos estarem no lugar certo (**proporcional a  $n$** )

# Complexidade dos algoritmos de ordenação

**bubblesort:  $O(n^2)$**

- 1 Troca a posição de todos os elementos fora do lugar (**proporcional a  $n$** )
- 2 Repete até todos os elementos estarem no lugar certo (**proporcional a  $n$** )

# Complexidade dos algoritmos de ordenação

**bubblesort:  $O(n^2)$**

- 1 Troca a posição de todos os elementos fora do lugar (**proporcional a  $n$** )
- 2 Repete até todos os elementos estarem no lugar certo (**proporcional a  $n$** )

**selection sort:**

# Complexidade dos algoritmos de ordenação

## **bubblesort: $O(n^2)$**

- 1 Troca a posição de todos os elementos fora do lugar (**proporcional a  $n$** )
- 2 Repete até todos os elementos estarem no lugar certo (**proporcional a  $n$** )

## **selection sort:**

- 1 Passa pela lista procurando o menor elemento

# Complexidade dos algoritmos de ordenação

## **bubblesort: $O(n^2)$**

- 1 Troca a posição de todos os elementos fora do lugar (**proporcional a  $n$** )
- 2 Repete até todos os elementos estarem no lugar certo (**proporcional a  $n$** )

## **selection sort:**

- 1 Passa pela lista procurando o menor elemento (**proporcional a  $n$** )

# Complexidade dos algoritmos de ordenação

## **bubblesort: $O(n^2)$**

- 1 Troca a posição de todos os elementos fora do lugar (**proporcional a  $n$** )
- 2 Repete até todos os elementos estarem no lugar certo (**proporcional a  $n$** )

## **selection sort:**

- 1 Passa pela lista procurando o menor elemento (**proporcional a  $n$** )
- 2 Repete para todos os elementos da lista

# Complexidade dos algoritmos de ordenação

## **bubblesort: $O(n^2)$**

- 1 Troca a posição de todos os elementos fora do lugar (**proporcional a  $n$** )
- 2 Repete até todos os elementos estarem no lugar certo (**proporcional a  $n$** )

## **selection sort:**

- 1 Passa pela lista procurando o menor elemento (**proporcional a  $n$** )
- 2 Repete para todos os elementos da lista (**proporcional a  $n$** )

# Complexidade dos algoritmos de ordenação

## **bubblesort: $O(n^2)$**

- 1 Troca a posição de todos os elementos fora do lugar (**proporcional a  $n$** )
- 2 Repete até todos os elementos estarem no lugar certo (**proporcional a  $n$** )

## **selection sort: $O(n^2)$**

- 1 Passa pela lista procurando o menor elemento (**proporcional a  $n$** )
- 2 Repete para todos os elementos da lista (**proporcional a  $n$** )

# Complexidade dos algoritmos de ordenação

## **bubblesort: $O(n^2)$**

- 1 Troca a posição de todos os elementos fora do lugar (**proporcional a  $n$** )
- 2 Repete até todos os elementos estarem no lugar certo (**proporcional a  $n$** )

## **selection sort: $O(n^2)$**

- 1 Passa pela lista procurando o menor elemento (**proporcional a  $n$** )
- 2 Repete para todos os elementos da lista (**proporcional a  $n$** )

## **mergesort:**

# Complexidade dos algoritmos de ordenação

## **bubblesort: $O(n^2)$**

- 1 Troca a posição de todos os elementos fora do lugar (**proporcional a  $n$** )
- 2 Repete até todos os elementos estarem no lugar certo (**proporcional a  $n$** )

## **selection sort: $O(n^2)$**

- 1 Passa pela lista procurando o menor elemento (**proporcional a  $n$** )
- 2 Repete para todos os elementos da lista (**proporcional a  $n$** )

## **mergesort:**

- 1 Divide a lista na metade

# Complexidade dos algoritmos de ordenação

## **bubblesort: $O(n^2)$**

- 1 Troca a posição de todos os elementos fora do lugar (**proporcional a  $n$** )
- 2 Repete até todos os elementos estarem no lugar certo (**proporcional a  $n$** )

## **selection sort: $O(n^2)$**

- 1 Passa pela lista procurando o menor elemento (**proporcional a  $n$** )
- 2 Repete para todos os elementos da lista (**proporcional a  $n$** )

## **mergesort:**

- 1 Divide a lista na metade (**proporcional a  $\log n$** )

# Complexidade dos algoritmos de ordenação

## **bubblesort: $O(n^2)$**

- 1 Troca a posição de todos os elementos fora do lugar (**proporcional a  $n$** )
- 2 Repete até todos os elementos estarem no lugar certo (**proporcional a  $n$** )

## **selection sort: $O(n^2)$**

- 1 Passa pela lista procurando o menor elemento (**proporcional a  $n$** )
- 2 Repete para todos os elementos da lista (**proporcional a  $n$** )

## **mergesort:**

- 1 Divide a lista na metade (**proporcional a  $\log n$** )
- 2 Para cada par de “fatias”, mescla

# Complexidade dos algoritmos de ordenação

## **bubblesort: $O(n^2)$**

- 1 Troca a posição de todos os elementos fora do lugar (**proporcional a  $n$** )
- 2 Repete até todos os elementos estarem no lugar certo (**proporcional a  $n$** )

## **selection sort: $O(n^2)$**

- 1 Passa pela lista procurando o menor elemento (**proporcional a  $n$** )
- 2 Repete para todos os elementos da lista (**proporcional a  $n$** )

## **mergesort:**

- 1 Divide a lista na metade (**proporcional a  $\log n$** )
- 2 Para cada par de “fatias”, mescla (**proporcional a  $n$** )

# Complexidade dos algoritmos de ordenação

## **bubblesort: $O(n^2)$**

- 1 Troca a posição de todos os elementos fora do lugar (**proporcional a  $n$** )
- 2 Repete até todos os elementos estarem no lugar certo (**proporcional a  $n$** )

## **selection sort: $O(n^2)$**

- 1 Passa pela lista procurando o menor elemento (**proporcional a  $n$** )
- 2 Repete para todos os elementos da lista (**proporcional a  $n$** )

## **mergesort: $O(n \times \log(n))$**

- 1 Divide a lista na metade (**proporcional a  $\log n$** )
- 2 Para cada par de “fatias”, mescla (**proporcional a  $n$** )

# Complexidade dos algoritmos de ordenação

- É possível demonstrar que não existe algoritmo de ordenação genérico com complexidade melhor que  $O(n \times \log(n))$

# Complexidade dos algoritmos de ordenação

- **É possível demonstrar que não existe algoritmo de ordenação genérico com complexidade melhor que  $O(n \times \log(n))$** 
  - ▶ Na verdade, isso se aplica a algoritmos baseados em comparações diretas entre os elementos

# Complexidade dos algoritmos de ordenação

- **É possível demonstrar que não existe algoritmo de ordenação genérico com complexidade melhor que  $O(n \times \log(n))$** 
  - ▶ Na verdade, isso se aplica a algoritmos baseados em comparações diretas entre os elementos
  - ▶ Existem alguns algoritmos com complexidade  $O(n)$ , mas eles não fazem comparações diretas entre os elementos; para isso ser possível, eles dependem de limitações específicas, como um número máximo de dígitos em cada número

# Números de ponto flutuante e suas pegadinhas

# Números de ponto flutuante

- Em python (e outras poucas linguagens), números inteiros podem ser tão grandes quanto necessário

# Números de ponto flutuante

- **Em python (e outras poucas linguagens), números inteiros podem ser tão grandes quanto necessário**
  - ▶ (Até o limite da memória disponível)

# Números de ponto flutuante

- **Em python (e outras poucas linguagens), números inteiros podem ser tão grandes quanto necessário**
  - ▶ (Até o limite da memória disponível)
    - » *A menos que você esteja lidando com números **muito** grandes, cada um deles ocupa pouca memória*

# Números de ponto flutuante

- **Em python (e outras poucas linguagens), números inteiros podem ser tão grandes quanto necessário**
  - ▶ (Até o limite da memória disponível)
    - » *A menos que você esteja lidando com números  **muito**  grandes, cada um deles ocupa pouca memória*
- **Mas, na maioria das linguagens (incluindo R), geralmente há um tamanho máximo**

# Números de ponto flutuante

- **Em python (e outras poucas linguagens), números inteiros podem ser tão grandes quanto necessário**
  - ▶ (Até o limite da memória disponível)
    - » *A menos que você esteja lidando com números  **muito**  grandes, cada um deles ocupa pouca memória*
- **Mas, na maioria das linguagens (incluindo R), geralmente há um tamanho máximo**
  - ▶ Em R, números até ~2 bilhões, positivo ou negativo (32 bits)

# Números de ponto flutuante

- **Em python (e outras poucas linguagens), números inteiros podem ser tão grandes quanto necessário**
  - ▶ (Até o limite da memória disponível)
    - » *A menos que você esteja lidando com números **muito** grandes, cada um deles ocupa pouca memória*
- **Mas, na maioria das linguagens (incluindo R), geralmente há um tamanho máximo**
  - ▶ Em R, números até ~2 bilhões, positivo ou negativo (32 bits)
    - » *Pouco! Uma das razões pelas quais inteiros não são muito usados em R*

# Números de ponto flutuante

- **Em python (e outras poucas linguagens), números inteiros podem ser tão grandes quanto necessário**
  - ▶ (Até o limite da memória disponível)
    - » *A menos que você esteja lidando com números **muito** grandes, cada um deles ocupa pouca memória*
- **Mas, na maioria das linguagens (incluindo R), geralmente há um tamanho máximo**
  - ▶ Em R, números até ~2 bilhões, positivo ou negativo (32 bits)
    - » *Pouco! Uma das razões pelas quais inteiros não são muito usados em R*
  - ▶ Em geral, números até ~9 quintilhões, positivo ou negativo (64 bits)

# Números de ponto flutuante

- Mas e números como  $\frac{2}{3}$  ou  $\sqrt{2}$ ?

# Números de ponto flutuante

- Mas e números como  $\frac{2}{3}$  ou  $\sqrt{2}$ ?
  - ▶ Não é possível representá-los com precisão em um sistema com memória finita

# Números de ponto flutuante

- Mas e números como  $\frac{2}{3}$  ou  $\sqrt{2}$ ?
  - ▶ Não é possível representá-los com precisão em um sistema com memória finita
- E números “pequenos”, como  $4.30 \times 10^{-7}$ ?

# Números de ponto flutuante

- Mas e números como  $\frac{2}{3}$  ou  $\sqrt{2}$ ?
  - ▶ Não é possível representá-los com precisão em um sistema com memória finita
- E números “pequenos”, como  $4.30 \times 10^{-7}$ ?
  - ▶ Não são incomuns!

# Números de ponto flutuante

- Mas e números como  $\frac{2}{3}$  ou  $\sqrt{2}$ ?
  - ▶ Não é possível representá-los com precisão em um sistema com memória finita
- E números “pequenos”, como  $4.30 \times 10^{-7}$ ?
  - ▶ Não são incomuns!
- Uma representação ingênua é escolher um valor mínimo, como  $10^{-12}$ , e tratá-lo como “1”

# Números de ponto flutuante

- **Mas e números como  $\frac{2}{3}$  ou  $\sqrt{2}$ ?**
  - ▶ Não é possível representá-los com precisão em um sistema com memória finita
- **E números “pequenos”, como  $4.30 \times 10^{-7}$ ?**
  - ▶ Não são incomuns!
- **Uma representação ingênua é escolher um valor mínimo, como  $10^{-12}$ , e tratá-lo como “1”**
  - ▶ Mas com isso os inteiros “normais” passam a gastar mais memória e/ou o número máximo possível fica reduzido

# Números de ponto flutuante

- **Mas e números como  $\frac{2}{3}$  ou  $\sqrt{2}$ ?**
  - ▶ Não é possível representá-los com precisão em um sistema com memória finita
- **E números “pequenos”, como  $4.30 \times 10^{-7}$ ?**
  - ▶ Não são incomuns!
- **Uma representação ingênua é escolher um valor mínimo, como  $10^{-12}$ , e tratá-lo como “1”**
  - ▶ Mas com isso os inteiros “normais” passam a gastar mais memória e/ou o número máximo possível fica reduzido
  - ▶ Além disso, os números próximos ao mínimo têm pouca precisão

# Números de ponto flutuante

- **Mas e números como  $\frac{2}{3}$  ou  $\sqrt{2}$ ?**
  - ▶ Não é possível representá-los com precisão em um sistema com memória finita
- **E números “pequenos”, como  $4.30 \times 10^{-7}$ ?**
  - ▶ Não são incomuns!
- **Uma representação ingênua é escolher um valor mínimo, como  $10^{-12}$ , e tratá-lo como “1”**
  - ▶ Mas com isso os inteiros “normais” passam a gastar mais memória e/ou o número máximo possível fica reduzido
  - ▶ Além disso, os números próximos ao mínimo têm pouca precisão
  - ▶ O que acontece se precisarmos do número  $10^{-13}$ ?

# Números de ponto flutuante

- **Mas e números como  $\frac{2}{3}$  ou  $\sqrt{2}$ ?**
  - ▶ Não é possível representá-los com precisão em um sistema com memória finita
- **E números “pequenos”, como  $4.30 \times 10^{-7}$ ?**
  - ▶ Não são incomuns!
- **Uma representação ingênua é escolher um valor mínimo, como  $10^{-12}$ , e tratá-lo como “1”**
  - ▶ Mas com isso os inteiros “normais” passam a gastar mais memória e/ou o número máximo possível fica reduzido
  - ▶ Além disso, os números próximos ao mínimo têm pouca precisão
  - ▶ O que acontece se precisarmos do número  $10^{-13}$ ?
    - » *(ele não é tãããão pequeno assim...)*

# Números de ponto flutuante

- **Mas e números como  $\frac{2}{3}$  ou  $\sqrt{2}$ ?**
  - ▶ Não é possível representá-los com precisão em um sistema com memória finita
- **E números “pequenos”, como  $4.30 \times 10^{-7}$ ?**
  - ▶ Não são incomuns!
- **Uma representação ingênua é escolher um valor mínimo, como  $10^{-12}$ , e tratá-lo como “1”**
  - ▶ Mas com isso os inteiros “normais” passam a gastar mais memória e/ou o número máximo possível fica reduzido
  - ▶ Além disso, os números próximos ao mínimo têm pouca precisão
  - ▶ O que acontece se precisarmos do número  $10^{-13}$ ?
    - » *(ele não é tããããõ pequeno assim...)*
- **Essa solução é usada apenas em alguns casos especiais**

# Números de ponto flutuante

- O que fazemos é representar esses números usando a notação científica: **mantissa**  $\times 10^{\text{expoente}}$

# Números de ponto flutuante

- O que fazemos é representar esses números usando a notação científica: **mantissa**  $\times 10^{\text{expoente}}$ 
  - ▶ É possível representar números em uma faixa **muito** maior (em R e na maioria das linguagens, de  $\sim 1.7 \times 10^{-308}$  a  $\sim 1.7 \times 10^{308}$ , positivo ou negativo)

# Números de ponto flutuante

- O que fazemos é representar esses números usando a notação científica: **mantissa**  $\times 10^{\text{expoente}}$ 
  - ▶ É possível representar números em uma faixa **muito** maior (em R e na maioria das linguagens, de  $\sim 1.7 \times 10^{-308}$  a  $\sim 1.7 \times 10^{308}$ , positivo ou negativo)
  - ▶ A precisão dos números é independente de sua grandeza

# Números de ponto flutuante

- O que fazemos é representar esses números usando a notação científica: **mantissa**  $\times 10^{\text{expoente}}$ 
  - ▶ É possível representar números em uma faixa **muito** maior (em R e na maioria das linguagens, de  $\sim 1.7 \times 10^{-308}$  a  $\sim 1.7 \times 10^{308}$ , positivo ou negativo)
  - ▶ A precisão dos números é independente de sua grandeza
    - » *depende apenas do tamanho da mantissa (na maioria das linguagens, cerca de 15 dígitos decimais)*

# Números de ponto flutuante

- O que fazemos é representar esses números usando a notação científica: **mantissa**  $\times 10^{\text{expoente}}$ 
  - ▶ É possível representar números em uma faixa **muito** maior (em R e na maioria das linguagens, de  $\sim 1.7 \times 10^{-308}$  a  $\sim 1.7 \times 10^{308}$ , positivo ou negativo)
  - ▶ A precisão dos números é independente de sua grandeza
    - » *depende apenas do tamanho da mantissa (na maioria das linguagens, cerca de 15 dígitos decimais)*
  - ▶ O consumo de memória é pequeno

# Números de ponto flutuante

- O que fazemos é representar esses números usando a notação científica: **mantissa**  $\times 10^{\text{expoente}}$ 
  - ▶ É possível representar números em uma faixa **muito** maior (em R e na maioria das linguagens, de  $\sim 1.7 \times 10^{-308}$  a  $\sim 1.7 \times 10^{308}$ , positivo ou negativo)
  - ▶ A precisão dos números é independente de sua grandeza
    - » *depende apenas do tamanho da mantissa (na maioria das linguagens, cerca de 15 dígitos decimais)*
  - ▶ O consumo de memória é pequeno
- **MAS**

# Números de ponto flutuante

- O que fazemos é representar esses números usando a notação científica: **mantissa**  $\times 10^{\text{expoente}}$ 
  - ▶ É possível representar números em uma faixa **muito** maior (em R e na maioria das linguagens, de  $\sim 1.7 \times 10^{-308}$  a  $\sim 1.7 \times 10^{308}$ , positivo ou negativo)
  - ▶ A precisão dos números é independente de sua grandeza
    - » *depende apenas do tamanho da mantissa (na maioria das linguagens, cerca de 15 dígitos decimais)*
  - ▶ O consumo de memória é pequeno
- **MAS**
- O limite de precisão pode causar surpresas

# Números de ponto flutuante

- Imagine uma representação em que a mantissa tem até 4 dígitos e o expoente pode variar de  $10^{-4}$  até  $10^5$

- **Imagine uma representação em que a mantissa tem até 4 dígitos e o expoente pode variar de  $10^{-4}$  até  $10^5$** 
  - ▶ O maior número representável seria  $9.999 \times 10^5$ , ou seja, 999.900

# Números de ponto flutuante

- **Imagine uma representação em que a mantissa tem até 4 dígitos e o expoente pode variar de  $10^{-4}$  até  $10^5$** 
  - ▶ O maior número representável seria  $9.999 \times 10^5$ , ou seja, 999.900
- **Não seria possível representar o número 34.567!**

# Números de ponto flutuante

- **Imagine uma representação em que a mantissa tem até 4 dígitos e o expoente pode variar de  $10^{-4}$  até  $10^5$** 
  - ▶ O maior número representável seria  $9.999 \times 10^5$ , ou seja, 999.900
- **Não seria possível representar o número 34.567!**
  - ▶ Seria preciso usar  $3.457 \times 10^4$ , ou seja, 34.570

**E como lidar com isso?**

# Imprecisões com ponto flutuante

- Testes de igualdade

# Imprecisões com ponto flutuante

- **Testes de igualdade**

- ▶ Números que deveriam ser iguais, mas são diferentes por erros de arredondamento, vão causar erros em condicionais (`if`, `while`)

# Imprecisões com ponto flutuante

- **Testes de igualdade**

- ▶ Números que deveriam ser iguais, mas são diferentes por erros de arredondamento, vão causar erros em condicionais (`if`, `while`)

```
if (sqrt(2)**2 == 2){ ... }
```

# Imprecisões com ponto flutuante

- **Testes de igualdade**

- ▶ Números que deveriam ser iguais, mas são diferentes por erros de arredondamento, vão causar erros em condicionais (`if`, `while`)

```
if (sqrt(2) * 2 == 2){ ... }
```

# Imprecisões com ponto flutuante

- **Testes de igualdade**

- ▶ Números que deveriam ser iguais, mas são diferentes por erros de arredondamento, vão causar erros em condicionais (`if`, `while`)



```
if (sqrt(2)**2 == 2){ ... }
```

```
epsilon <- 1e-15 # É preciso escolher um valor "razoável"  
if (abs(sqrt(2)**2 - 2) < epsilon) { ... }
```

- **Multiplicações e divisões não costumam causar problemas**

- **Multiplicações e divisões não costumam causar problemas**
  - ▶ Independentemente da grandeza (expoente), os cálculos são feitos com toda a precisão da mantissa

- **Multiplicações e divisões não costumam causar problemas**

- ▶ Independentemente da grandeza (expoente), os cálculos são feitos com toda a precisão da mantissa

- »  $(1,234 \times 10^8) \times (5,678 \times 10^{-3})$

- **Multiplicações e divisões não costumam causar problemas**

- ▶ Independentemente da grandeza (expoente), os cálculos são feitos com toda a precisão da mantissa

- »  $(1,234 \times 10^8) \times (5,678 \times 10^{-3}) = 1,234 \times 5,678 \times 10^5$

# Imprecisões com ponto flutuante

- Somas e subtrações podem causar problemas

# Imprecisões com ponto flutuante

- **Somas e subtrações podem causar problemas**
  - Somas/subtrações com números de grandezas muito diferentes

# Imprecisões com ponto flutuante

- Somas e subtrações podem causar problemas
  - Somas/subtrações com números de grandezas muito diferentes
    - » `cat(format(2e10 + 4e-10, digits=22), "\n")`

# Imprecisões com ponto flutuante

- Somas e subtrações podem causar problemas
  - Somas/subtrações com números de grandezas muito diferentes
    - » `cat(format(2e10 + 4e-10, digits=22), "\n")`  
2e+10

# Imprecisões com ponto flutuante

- Somas e subtrações podem causar problemas
  - Somas/subtrações com números de grandezas muito diferentes
    - » `cat(format(2e10 + 4e-10, digits=22), "\n")`  
2e+10 🤪

# Imprecisões com ponto flutuante

- Somas e subtrações podem causar problemas

- ▶ Somas/subtrações com números de grandezas muito diferentes

- » `cat(format(2e10 + 4e-10, digits=22), "\n")`

2e+10 🤪

- ▶ Subtrações com números muito próximos

# Imprecisões com ponto flutuante

- Somas e subtrações podem causar problemas

- ▶ Somas/subtrações com números de grandezas muito diferentes

- » `cat(format(2e10 + 4e-10, digits=22), "\n")`

2e+10 🤪

- ▶ Subtrações com números muito próximos

- » *O resultado da diferença vai ser predominantemente composto pelos dígitos menos significativos, onde o erro é proporcionalmente maior*

# Imprecisões com ponto flutuante

- Somas e subtrações podem causar problemas

- ▶ Somas/subtrações com números de grandezas muito diferentes

- » `cat(format(2e10 + 4e-10, digits=22), "\n")`

2e+10 🤪

- ▶ Subtrações com números muito próximos

- » *O resultado da diferença vai ser predominantemente composto pelos dígitos menos significativos, onde o erro é proporcionalmente maior*

- » `cat(format(0.1000000000000001, digits=22), "\n")`

0.100000000000000999950123 (erros no dígito 18)

# Imprecisões com ponto flutuante

- Somas e subtrações podem causar problemas

- ▶ Somas/subtrações com números de grandezas muito diferentes

- » `cat(format(2e10 + 4e-10, digits=22), "\n")`

2e+10 🤪

- ▶ Subtrações com números muito próximos

- » *O resultado da diferença vai ser predominantemente composto pelos dígitos menos significativos, onde o erro é proporcionalmente maior*

`cat(format(0.100000000000001, digits=22), "\n")`

0.1000000000000009999950123 (erros no dígito 18)

`cat(format(0.100000000000001 - 0.1, digits=22), "\n")`

(diferença no dígito 13)

# Imprecisões com ponto flutuante

- Somas e subtrações podem causar problemas

- ▶ Somas/subtrações com números de grandezas muito diferentes

- » `cat(format(2e10 + 4e-10, digits=22), "\n")`

2e+10 🤪

- ▶ Subtrações com números muito próximos

- » *O resultado da diferença vai ser predominantemente composto pelos dígitos menos significativos, onde o erro é proporcionalmente maior*

`cat(format(0.100000000000001, digits=22), "\n")`

0.1000000000000009999950123 (erros no dígito 18)

`cat(format(0.100000000000001 - 0.1, digits=22), "\n")`

(diferença no dígito 13)

9.99894611553031609219e-14

# Imprecisões com ponto flutuante

- Somas e subtrações podem causar problemas

- ▶ Somas/subtrações com números de grandezas muito diferentes

- » `cat(format(2e10 + 4e-10, digits=22), "\n")`

2e+10 🤪

- ▶ Subtrações com números muito próximos

- » *O resultado da diferença vai ser predominantemente composto pelos dígitos menos significativos, onde o erro é proporcionalmente maior*

`cat(format(0.100000000000001, digits=22), "\n")`

0.1000000000000009999950123 (erros no dígito 18)

`cat(format(0.100000000000001 - 0.1, digits=22), "\n")`

(diferença no dígito 13)

9.99894611553031609219e-14 (erro no dígito 4)

# Imprecisões com ponto flutuante

- Somas e subtrações podem causar problemas

- ▶ Somas/subtrações com números de grandezas muito diferentes

- » `cat(format(2e10 + 4e-10, digits=22), "\n")`

2e+10 🤪

- ▶ Subtrações com números muito próximos

- » *O resultado da diferença vai ser predominantemente composto pelos dígitos menos significativos, onde o erro é proporcionalmente maior*

`cat(format(0.100000000000001, digits=22), "\n")`

0.1000000000000009999950123 (erros no dígito 18)

`cat(format(0.100000000000001 - 0.1, digits=22), "\n")`

(diferença no dígito 13)

9.99894611553031609219e-14 (erro no dígito 4) 🤪

# Imprecisões com ponto flutuante

- Somas e subtrações podem causar problemas

- ▶ Somas/subtrações com números de grandezas muito diferentes

- » `cat(format(2e10 + 4e-10, digits=22), "\n")`

2e+10 🤪

- ▶ Subtrações com números muito próximos

- » *O resultado da diferença vai ser predominantemente composto pelos dígitos menos significativos, onde o erro é proporcionalmente maior*

- » `cat(format(0.1000000000000001, digits=22), "\n")`

0.1000000000000009999950123 (erros no dígito 18)

- » `cat(format(0.1000000000000001 - 0.1, digits=22), "\n")`

(diferença no dígito 13)

9.99894611553031609219e-14 (erro no dígito 4) 🤪

- Somatórios podem fazer as imprecisões se combinarem, tornando erros irrelevantes em catastróficos

# Imprecisões com ponto flutuante

- Somas e subtrações podem causar problemas

- ▶ Somas/subtrações com números de grandezas muito diferentes

- » `cat(format(2e10 + 4e-10, digits=22), "\n")`

2e+10 🤪

- ▶ Subtrações com números muito próximos

- » *O resultado da diferença vai ser predominantemente composto pelos dígitos menos significativos, onde o erro é proporcionalmente maior*

`cat(format(0.100000000000001, digits=22), "\n")`

0.100000000000000999950123 (erros no dígito 18)

`cat(format(0.100000000000001 - 0.1, digits=22), "\n")`

(diferença no dígito 13)

9.99894611553031609219e-14 (erro no dígito 4) 🤪

- Somatórios podem fazer as imprecisões se combinarem, tornando erros irrelevantes em catastróficos

- ▶ Para evitar isso, procure processar do “pequeno” para o “grande”

## Exercício — calculando $\pi$ com Gregory-Leibniz

Dado um inteiro  $k \geq 0$ , é possível calcular um valor aproximado de  $\pi$  através da expansão de Gregory-Leibniz:

$$\pi \approx \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} \dots + \frac{(-1)^k 4}{2k+1}$$

Escreva um programa que lê um valor para  $k$  e realiza o cálculo acima.

## Exercício – calculando $\pi$ com Gregory-Leibniz

$$\pi \approx \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} \dots + \frac{(-1)^k 4}{2k+1}$$

```
main <- function() {
  k <- as.integer(readline("Digite o valor de k: "))
  cat("O valor de pi é aproximadamente", acha_pi(k), "\n")
}
acha_pi <- function(k) {
  pi <- 0
  sinal <- 1
  n <- 0
  while (n <= k) {
    pi <- pi + sinal * 4 / (2*n + 1)
    sinal <- sinal * -1
    n <- n + 1
  }
  return(pi)
}
main()
```

## Exercício — calculando $\pi$ com Gregory-Leibniz

$$\pi \approx \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} \dots + \frac{(-1)^k 4}{2k+1}$$

```
main <- function() {
  k <- as.integer(readline("Digite o valor de k: "))
  cat("O valor de pi é aproximadamente", acha_pi(k), "\n")
}
acha_pi <- function(k) {
  pi <- 0
  sinal <- 1
  if (k %% 2 != 0) { sinal <- -1 }
  while (k > 0) {
    pi <- pi + sinal * 4 / (2*k + 1)
    sinal <- sinal * -1
    k <- k - 1
  }
  return(pi)
}
main()
```

**E como lidar com isso?**

**E como lidar com isso?**

**Na maior parte do tempo,  
não é preciso se preocupar**