Programação Funcional

(COMP0393)

Leila M. A. Silva

dcomp
departamento de computação

Programando com Listas

(COMP0393)

Aula 9

dcomp
departamento de computação

Algoritmos de Ordenação

- Vamos ver alguns algoritmos para ordenar elementos de uma lista de inteiros:
  - Seleção
  - Inserção Direta
  - Merge Sort
  - Quick Sort
- A entrada destes algoritmos é uma lista de inteiros em ordem arbitrária. A saída é a mesma lista ordenada em ordem crescente.
- Cada um destes algoritmos tem uma estratégia diferente para realizar a mesma tarefa.
- Alguns empregam indução fraca outros indução forte.

Seleção

- O algoritmo de Seleção ordena uma lista de inteiros através da seguinte estratégia:
  - Se a lista for vazia, não há o que ordenar.
  - Caso contrário, coleta o elemento mínimo da lista.
  - Remove o elemento mínimo da lista original e ordena o restante da lista, recursivamente.
  - Adiciona o mínimo na cabeça da lista restante ordenada.
- Para resolver este problema precisamos de funções para:
  - procurar o elemento mínimo de uma lista, se a lista não for vazia;
  - remover o mínimo da lista.

Seleção

Lista em ordem arbitrária

Procura o mínimo

Remove o mínimo

Ordena o restante da lista

Lista ordenada

Insere o mínimo na cabeça da lista ordenada

Seleção

- Função para calcular o elemento mínimo de uma lista não vazia:
  - Caso base: lista unitária *[x]*. Neste caso o elemento mínimo é o próprio *x*.
  - HI: Sei calcular o menor elemento de uma lista não vazia de *k-1* elementos.
  - Caso geral. Desejo calcular o menor elemento de uma lista não vazia de *k* elementos.
    - Estratégia: Suponha a lista *(x:xs)*.

      Reservo o primeiro elemento *x* .

      A lista *xs* tem *k-1* elementos. Aplico HI em *xs* e obtenho o menor elemento de *xs*, digamos *y*.

      Comparo *y* com *x* para definir quem é o menor da lista original *(x:xs)*.

Seleção

- Função para calcular o elemento mínimo de uma lista não vazia.

```
mini :: Int -> Int -> Int
mini a b = if a<=b then a else b

menor :: [Int] -> Int
menor [x] = x                          --caso base
menor (x:xs) = mini x (menor xs)   --caso geral
```

Seleção

- Função para remover um elemento da lista:
  - Caso base: lista vazia. Neste caso não há elemento para remover e a lista retorna vazia.
  - HI: Sei remover um elemento de listas com *k-1* elementos.
  - Caso geral: Desejo remover um elemento de uma lista de *k* elementos.
    - Estratégia:

      Suponha a lista *(x:xs)* e o elemento *y* que desejo remover.

      Se *y* == *x* retorno a lista *xs*.

      Caso contrário, *y* está dentro da lista *xs*. Mas esta lista tem tamanho *k-1* e por HI eu já sei fazer. Ao aplicar a HI em *xs* a lista retornada terá os elementos de *xs* sem o elemento *y*, que foi removido. Vamos chamar esta lista de *xs'*.

      A solução final será *x:xs'*.

Seleção

- Função para remover um elemento da lista.

```
removeLista :: Int -> [Int] -> [Int]
removeLista _ [] = []                    --caso base
removeLista y (x:xs)                     --caso geral
   | y==x = xs
   | otherwise = x: removeLista y xs
```

- Observe que se a lista tiver mais de um elemento `y` somente a primeira ocorrência dele é removida. Uma solução usando compreensões removeria todas as ocorrências, mas neste problema de ordenação desejamos remover apenas uma ocorrência do elemento.

Seleção

- Podemos estruturar o algoritmo de Seleção por indução fraca como:
  - Caso base: Lista vazia. Neste caso não há elementos a ordenar e a resposta é a lista vazia.
  - HI: Sei ordenar uma lista de *k-1* elementos pelo método de Seleção.
  - Caso geral: Desejo ordenar uma lista de *k* elementos pelo método de Seleção.
    - Estratégia:
      - Suponha uma lista *(x:xs)* com *k* elementos.
      - Identifico o mínimo de *(x:xs)*, digamos *m*.
      - Removo *m* de *(x:xs)* resultando uma lista de tamanho *k-1*, digamos *xs'*.
      - Aplico HI em *xs'* porque ela tem *k-1* elementos e ao fazer isto o retorno é a lista xs' ordenada. Vamos chamar esta lista ordenada de *xs"*.
      - Componho a solução fazendo *m:xs"*.

Seleção

- Agora que temos uma função para calcular o mínimo e outra para remover um elemento podemos usar estas funções para montar o algoritmo de ordenação por seleção.

```
ordSelecao :: [Int] -> [Int]
ordSelecao [] = []                                  --caso base
ordSelecao zs = m:ordSelecao (removeLista m zs)     --caso geral
    where m = menor zs
```

Seleção

```
ordSelecao [4,2,9,8] ↝ 2:ordSelecao (removeLista 2 [4,2,9,8]) ↝... ↝
                     ↝ 2:ordSelecao ([4,9,8])
                     ↝ 2:4: ordSelecao (removeLista 4 [4,9,8]) ↝... ↝
                     ↝ 2:4: ordSelecao ([9,8])
                     ↝ 2:4:8: ordSelecao (removeLista 8 [9,8]) ↝... ↝
                     ↝ 2:4:8: ordSelecao ([9])
                     ↝ 2:4:8:9: ordSelecao (removeLista 9 [9]) ↝... ↝
                     ↝ 2:4:8:9: ordSelecao ([])
                     ↝ 2:4:8:9:[] ↝ 2:4:8:[9] ↝ 2:4:[8,9]
                     ↝ 2:[4,8,9] ↝ [2,4,8,9]
```

Passos das funções auxiliares menor e removeLista foram omitidos

Inserção Direta

- O algoritmo de Inserção Direta ordena uma lista de inteiros através da seguinte estratégia:
  - Se a lista for vazia, não há o que ordenar.
  - Caso contrário, o algoritmo reserva o primeiro elemento e ordena o restante da lista recursivamente.
  - Em seguida, insere o elemento reservado na lista dos demais elementos que foi ordenada, preservando a ordem.
  - Ao final do processo, devolve toda a lista ordenada.

Inserção Direta

Listas em ordem arbitrária

Reserva o primeiro elemento

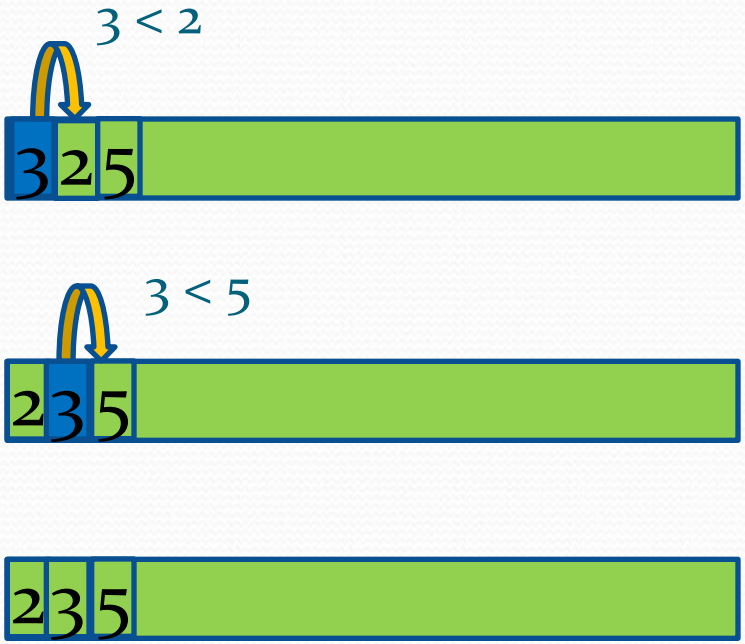
Ordena o restante

Insere em ordem

Lista ordenada

Inserção Direta

- Como coloco o elemento na ordem? Por comparação com a cabeça da lista já ordenada.



- Esta é a parte crucial do algoritmo, é a função principal!!

Inserção Direta

- Função para inserir um elemento em uma lista ordenada.
  - Caso base: lista vazia *[]*. Neste caso, retorno uma lista com o elemento que desejo inserir *[y]*.
  - HI: Sei inserir um elemento *y* em uma lista ordenada de *k-1* elementos.
  - Caso geral. Desejo inserir um elemento *y* em uma lista ordenada de *k* elementos.
    - Estratégia: Suponha o elemento a inserir *y* na lista ordenada (*z:zs*).

      Se *y* <=*z*, insiro na cabeça da lista e retorno *y: (z: zs)*.

      Caso contrário, sei que *z* vem antes de *y* na lista ordenada e reduzo o meu problema a inserir *y* na lista *zs*, que tem *k-1* elementos.

      Aplico HI para inserir *y* em *zs* e obtenho a lista ordenada *zs'*, com o y inserido na posição correta.

      A solução final será *(z:zs')*.

Inserção Direta

- Função para inserir um elemento em uma lista ordenada.

```
insOrd :: Int -> [Int] -> [Int]
insOrd y [] = [y]                          --caso base
insOrd y (z:zs)                            --caso geral
   | y <= z = y : z : zs
   | otherwise = z: insOrd y zs
```

Inserção Direta

- Agora podemos fazer o algoritmo por indução fraca.
  - Caso base: lista vazia, retorna lista vazia, não há o que ordenar.
  - HI : Sei ordenar por inserção uma lista com *k-1* elementos.
  - Caso geral. Desejo ordenar por inserção uma lista de *k* elementos.
    - Estratégia:
      - Suponho a lista a ordenar (x:xs).
      - Reservo o primeiro elemento.
      - Aplico HI na lista *xs* que tem *k-1* elementos, retornando a lista ordenada digamos *xs'*.
      - Para compor a solução final, uso a função de *insOrd* para inserir *x* em *xs'*, que já está ordenada.

Inserção Direta

```
ordInsercao :: [Int] -> [Int]
ordInsercao [] = []                                  --caso base
ordInsercao (x:xs) = insOrd x (ordInsercao xs) -- caso geral
```

Inserção Direta

```
ordInsercao [4,2,9,8]
   ↝ insOrd 4 (ordInsercao [2,9,8])
   ↝ insOrd 4 (insOrd 2 (ordInsercao [9,8]))
   ↝ insOrd 4 (insOrd 2 (insOrd 9 (ordInsercao [8])))
   ↝ insOrd 4 (insOrd 2 (insOrd 9 (insOrd 8 (ordInsercao []))))
   ↝ insOrd 4 (insOrd 2 (insOrd 9 (insOrd 8 [])))
   ↝ insOrd 4 (insOrd 2 (insOrd 9 [8]))
   ↝ insOrd 4 (insOrd 2 (insOrd 9 [8]))
   ↝ insOrd 4 (insOrd 2 (8: insOrd 9 [])
   ↝ insOrd 4 (insOrd 2 (8: [9])
   ↝ insOrd 4 (insOrd 2 [8,9])
   ↝ insOrd 4 (2: 8: [9]) ↝ ... ↝
   ↝ insOrd 4 [2,8,9] ↝
   ↝ 2: insOrd 4 [8,9]
   ↝ 2:4:8:[9] ↝ ... ↝ [2,4,8,9]
```

Merge Sort

- O algoritmo de Merge Sort ordena uma lista de inteiros através da seguinte estratégia:
  - Se a lista for vazia, não há o que ordenar. Se ela for unitária está trivialmente ordenada, retorna a própria lista.
  - Caso contrário, o algoritmo divide a lista original ao meio gerando duas sublistas: *A* e *B*.
  - As sublistas *A* e a *B* são então ordenadas recursivamente.
  - Para compor a solução final, usa-se o procedimento de intercalação de listas ordenadas.

Merge Sort

Lista em ordem arbitrária

Divide a lista ao meio

Sublistas ordenadas

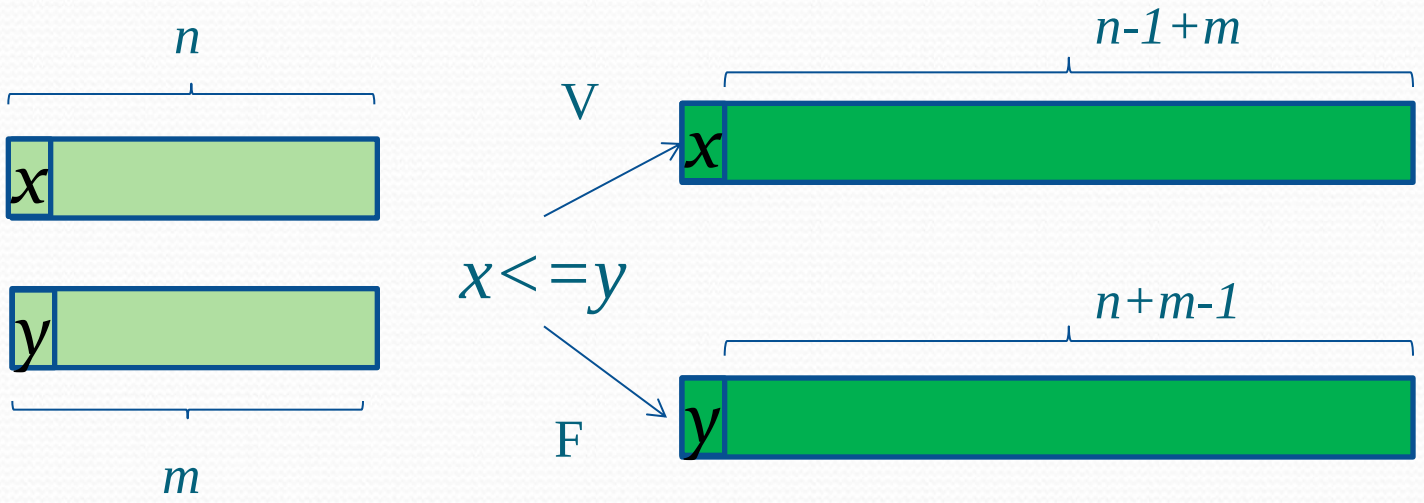
Ordena as duas sublistas

Lista final ordenada

Intercala as duas sublistas

Merge Sort

- Como intercalo listas ordenadas? Comparando a cabeça das duas listas, escolhendo a menor e intercalando o restante.



Merge Sort

- Função para intercalar duas listas ordenadas de tamanhos *m* e *n*.
  - Caso base: pelo menos uma lista é vazia *[]*. Neste caso, retorno a outra lista.
  - HI: Sei intercalar duas listas ordenadas cuja soma dos tamanhos seja *k-1*.
  - Caso geral. Intercalar duas listas ordenadas cuja soma dos tamanho seja *k* elementos.
    - Estratégia:
      - Suponha duas listas ordenadas *(x:xs)* e *(y:ys)*.
      - Se *x<=y* então aplique HI nas listas *xs* e *(y:ys)*, cuja soma dos tamanhos é *k-1*, retornando uma lista ordenada, xys'.
        A solução final será *x:xys'*.
      - Caso contrário, aplique HI nas listas (*x:xs)* e *ys*, cuja soma dos tamanhos é *k-1*, retornando uma lista ordenada, xys".
        A solução final será *y:xys"*.

Merge Sort

- Função intercalar duas listas ordenadas

```
intercala :: [Int] -> [Int] -> [Int]
intercala xs [] = xs                    --casos base
intercala [] ys = ys
intercala (x:xs) (y:ys)                 --caso geral
    | x <= y = x: intercala xs (y:ys)
    | otherwise = y: intercala (x:xs) ys
```

Merge Sort

- Agora podemos fazer o algoritmo por indução forte.
  - Caso base: lista vazia, retorna lista vazia, não há o que ordenar. Lista unitária, retorna a própria lista.
  - HI : Sei ordenar por merge sort uma lista com o$\leq i < k$ elementos
  - Caso geral. Desejo ordenar por merge sort uma lista de *k* elementos
    - Estratégia:
      - Suponho a lista a ordenar (*x:xs*).
      - Divido a lista aproximadamente ao meio, gerando duas sublistas, digamos *us* e *vs*.
      - Aplico HI nas listas *us* e *vs*, que têm < *k* elementos, retornando as listas ordenadas digamos *us'* e *vs'*.
      - Para compor a solução final, uso a função *intercala* para intercalar *us'* e *vs'*.

Merge Sort

```
mergeSort :: [Int] -> [Int]
mergeSort [] = []                                          --caso base
mergeSort [x] = [x]
mergeSort xs = intercala (mergeSort us) (mergeSort vs) -- caso geral
      where meio = (length xs) `div` 2
            us = take meio xs
            vs = drop meio xs
```

Merge Sort

```
mergeSort [4,9,2]
    ↝ intercala (mergeSort [4]) (mergeSort [9,2])
    ↝ intercala [4] (mergeSort [9,2])
    ↝ intercala [4] (intercala (mergeSort [9]) (mergeSort [2]))
    ↝ intercala [4] (intercala [9] (mergeSort [2]))
    ↝ intercala [4] (intercala [9][2])
    ↝ intercala [4] (2: intercala [9][])
    ↝ intercala [4] (2: [9])
    ↝ intercala [4] [2,9]
    ↝ 2: intercala [4] [9]
    ↝ 2: 4: intercala [] [9]
    ↝ 2: 4: [9]
    ↝ 2: [4, 9]
    ↝ [2, 4, 9]
```

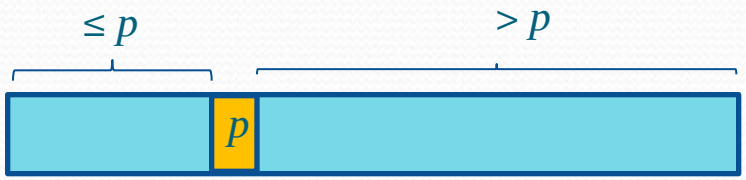
Quick Sort

- O algoritmo de Quick Sort ordena uma lista de inteiros através da seguinte estratégia:
  - Se a lista for vazia, não há o que ordenar.
  - Caso contrário, o algoritmo escolhe um elemento arbitrário para pivô e divide a lista original em três parte:
    - Uma sublista *A* contendo os elementos menores ou iguais ao pivô
    - O pivô
    - Uma sublista *B* contendo os elementos maiores que o pivô.
  - As duas sublistas são então ordenadas recursivamente.
  - Para compor a solução final, concatena-se a *A*, *[pivô]* e *B*.

Quick Sort

Lista em ordem arbitrária

Escolhe o pivô, *p*



Separa os elementos da lista de acordo com o pivô

Sublistas ordenadas

Ordena cada sublista

Lista final ordenada

Quick Sort

- Agora podemos fazer o algoritmo por indução forte.
  - Caso base: lista vazia, retorna lista vazia, não há o que ordenar.
  - HI : Sei ordenar por quick sort uma lista com o$\leq i < k$ elementos
  - Caso geral. Desejo ordenar por quick sort uma lista de k elementos
    - Estratégia:
      - Suponho a lista a ordenar (*x:xs*).
      - Escolho um elemento, por exemplo a cabeça da lista, *x,* para ser o pivô.
      - Divido a lista gerando duas sublistas, *us,* com elementos ≤ *x* e *vs,* com elementos >*x*.
      - Como *x* não faz parte destas sublistas elas possuem <*k* elementos. Aplico HI nas listas *us* e *vs,* retornando as listas ordenadas *us'* e *vs'*.
      - Para compor a solução final, uso a função concateno *us'* , *[x]* e *vs'*.

Quick Sort

```
quickSort :: [Int] -> [Int]
quickSort [] = []                                          --caso base
quickSort (x:xs) = quickSort us ++ [x]++ quickSort vs -- caso geral
      where us = [u| u <-xs, u <= x]
            vs = [v| v <-xs, v > x]
```

```
quickSort :: [Int] -> [Int]
quickSort [] = []                                          --caso base
quickSort (x:xs) = quickSort us ++ (x: quickSort vs)    -- caso geral
      where us = [u| u <-xs, u <= x]
            vs = [v| v <-xs, v > x]
```

Quick Sort

```
quickSort [4,9,2,1]
   ↝ quickSort [2,1] ++ (4: quickSort [9])
   ↝ quickSort [1] ++ (2: quickSort []) ++ (4: quickSort [9])
   ↝ quickSort [] ++ (1: quickSort []) ++ (2: quickSort []) ++ (4:
                                                    quickSort [9])
   ↝ []++(1: quickSort [])++(2: quickSort [])++(4: quickSort [9])
   ↝ []++(1: [])++(2: quickSort [])++(4: quickSort [9])
   ↝ []++ [1]++(2: quickSort [])++(4: quickSort [9])
   ↝ []++ [1]++(2: [])++(4: quickSort [9])
  ↝ []++ [1]++[2]++(4: quickSort [9])
  ↝ []++ [1]++[2]++(4: (quickSort [] ++ (9: quickSort [])))
  ↝ []++ [1]++[2]++(4: ([] ++ (9: quickSort [])))
  ↝ []++ [1]++[2]++(4: ([] ++ (9:[])))
  ↝ []++ [1]++[2]++(4: ([] ++ [9])) ↝ ... ↝[1,2,4,9]
```

Mais exemplos

- Considere a seguinte figura do número 7, em que: preto (1) e branco (.).

1111111

..111..

111....

- Desejo construir funções para:
  - Inverter a cor;
  - Refletir segundo um espelho na horizontal;
  - Refletir segundo um espelho na vertical;
  - Colocar uma do lado da outra.
  - Escalar a figura de um dado valor.

Mais exemplos

- Representação da figura:

```
type LinhaFig = [Char]
type Figura = [LinhaFig]
```

1111111 `[['1','1','1','1','1','1','1'],`

..111.. `['.','.','1','1','1','.','.'],`

111.... `['1','1','1','.','.','.','.']]`

- Inverter a cor da Figura: inverter caracter, inverter linha, inverter a figura

```
inverteCh :: Char -> Char
inverteCh x = if x = '1' then '.' else '1'

inverteCorLin :: LinhaFig-> LinhaFig
inverteCorLin xs = [inverteCh x| x<-xs]

inverteCorFig :: Figura -> Figura
inverteCorFig [] = []
inverteCorFig (xs:xss) = inverteCorLin xs:inverteCorFig xss
```

Mais exemplos

- Refletir na horizontal: linhas são preservadas mas a ordem das linhas é invertida.

1111111
..111..
111....

```
[['1','1','1','1','1','1','1'],
 ['.','.','1','1','1','.','.'],
 ['1','1','1','.','.','.','.']]
```

111....
..111..
1111111

```
refleteH :: Figura -> Figura
refleteH [] = []
refleteH xss = reverse xss
```

Mais exemplos

- Refletir na vertical: linhas são invertidas mas a ordem das linhas é preservada.

1111111
..111..
111....

```
[['1','1','1','1','1','1','1'],
 ['.','.','1','1','1','.','.'],
 ['1','1','1','.','.','.','.']]
```

1111111
..111..
....111

```
refleteV :: Figura -> Figura
refleteV [] = []
refleteV (xs:xss) = reverse xs: refleteV xss
```

Mais exemplos

- Colocar uma do lado da outra.

1111111
..111..
111....

```
[['1','1','1','1','1','1','1'],
 ['.','.','1','1','1','.','.'],
 ['1','1','1','.','.','.','.']]
```

1111111111111
..111....111..
111....111....

```
ladoAlado :: Figura -> Figura
ladoAlado [] = []
ladoAlado (xs:xss) = (xs++xs): ladoAlado xss
```

Mais exemplos

- Escalar a figura de um número *k*: replicar cada caracter da lista *k* vezes e replicar a linha replicada *k* vezes.

1111111 `[['1','1','1','1','1','1','1'],`

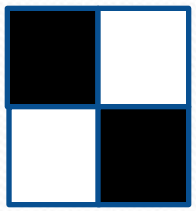
..111.. `['.','.','1','1','1','.','.'],`

111.... `['1','1','1','.','.','.','.']]`

```
111111111111111
111111111111111
....111111....
....111111....
111111........
111111........
```

```
escalaFig :: Int -> Figura -> Figura
escalaFig _ [] = []
escalaFig  k (xs:xss)
   | k <= 0 = []
   |otherwise = replicaLin k (linEsc xs) ++ escalaFig k xss
   where linEsc ys = concat [replicate k ch| ch<-ys]
         replicaLin :: Int -> LinhaFig -> Figura
         replicaLin 0 us = []
         replicaLin i us = us: replicaLin (i-1) us
```

Exercícios Recomendados

- Dados duas listas sem elementos repetidos, faça uma função para checar se as listas são disjuntas.
- Dadas duas listas sem elementos repetidos, faça uma função para gerar uma lista com os elementos comuns das duas listas **sem usar** a função `elem`.
- Estenda o exemplo da figura fazendo funções para:
  - Colocar uma cópia da figura em cima da figura de entrada;
  - Imprimir uma figura com o uso de `putStr;`
  - Considere que o quadrado branco simboliza a figura original e o preto a figura original com cores invertidas
  - Elabore uma função que gere figuras do tipo:
    - A) 
    - B) 
      ...

      A quantidade de quadrados é informada na função
    - C)