ENGENHARIA INFORMÁTICA E DE COMPUTADORES

Algoritmos e Estruturas de Dados

(parte 1 – Técnicas de Algoritmia)

2° Semestre 2022/2023
Instituto Superior de Engenharia de Lisboa
Paula Graça

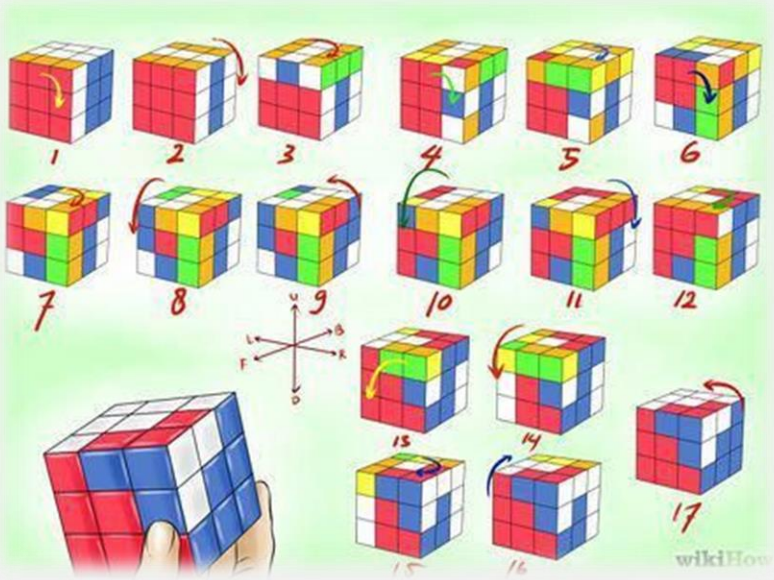
TÉCNICAS DE ALGORITMIA

- Principais técnicas de desenho de algoritmos

  - Força Bruta (*Brute Force*)
  - Programação Dinâmica (*Dynamic Programming*)
  - Técnica Gananciosa (*Greedy*)
  - Dividir para conquistar (*Divide and Conquer*)
  - Recursividade
  - Problemas tipo de ilustração das várias técnicas

FORÇA BRUTA

- A técnica por Força Bruta (*Brute Force*) é a forma mais simples mas mais ineficiente de desenhar um algoritmo
  - Para calcular a solução:
    - Gera todas as combinações possíveis dentro do domínio de resultados para o problema em causa
    - Seleciona de entre todas as combinações geradas, qual a solução do problema



FORÇA BRUTA

- Problema: Ladrão da Mochila
  - Um ladrão entra numa loja com uma mochila de capacidade **P** (peso)
  - Na loja existem objetos de peso $P_1, P_2, ..., P_n$ com valor $V_1, V_2, ..., V_n$
  - Qual o saque mais valioso que o ladrão pode levar na mochila?

FORÇA BRUTA

- Solução através da técnica de força bruta

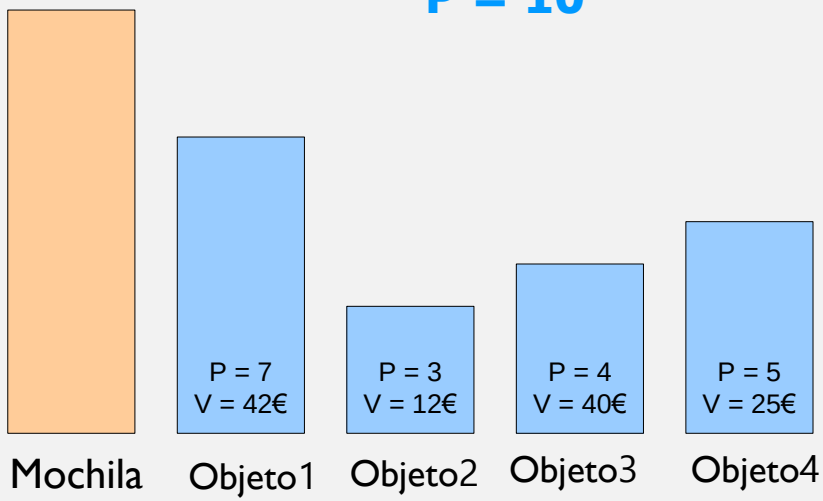
1. São identificadas todas as combinações possíveis dos n objetos (subconjuntos)
2. É calculado o peso total de cada subconjunto, de forma a identificar os que são válidos (cujo peso não ultrapassa a capacidade da mochila)
3. A solução é o subconjunto com maior valor, de entre todas as combinações válidas

FORÇA BRUTA

Sub-conjuntos objetos	Peso total	Valor total
Ø	0	0€
{1}	7	42€
{2}	3	12€
{3}	4	40€
{4}	5	25€
{1,2}	10	54€
{1,3}	11	n/a
{1,4}	12	n/a
{2,3}	7	52€
{2,4}	8	37€
{3,4}	**9**	**65€**
{1,2,3}	14	n/a
{1,2,4}	15	n/a
{1,3,4}	16	n/a
{2,3,4}	12	n/a
{1,2,3,4}	19	n/a

Capacidade da mochila:

P = 10



Solução
Subconjunto de objetos com o saque mais valioso

PROGRAMAÇÃO DINÂMICA

- A Programação Dinâmica (*Dynamic Programming*) é uma técnica para a resolução de problemas complexos
  - Foi inventada pelo matemático Richard Belmann (US), em 1950
  - É do tipo *space-for-time trade off* (troca de tempo por espaço)

1. Parte o problema original em sub-problemas mais pequenos
2. As soluções parciais dos sub-problemas são então calculadas e memorizadas numa tabela (*array* ou estrutura similar), para evitar serem recalculadas
3. A solução do problema original é obtida combinando as soluções dos sub-problemas

PROGRAMAÇÃO DINÂMICA

- A técnica pode ser usada em problemas que podem ser divididos em sub-problemas do mesmo tipo, cujas soluções são reutilizadas no cálculo da solução final

  - Exemplo: **1 + 2 + 3 + 4 + 5 + 6**
  - Pode ser calculado somando o resultado dos três sub-problemas:
    - **1 + 2 = 3**
    - **3 + 4 = 7**
    - **5 + 6 = 11**

  - Solução final:
    - **3 + 7 + 11**

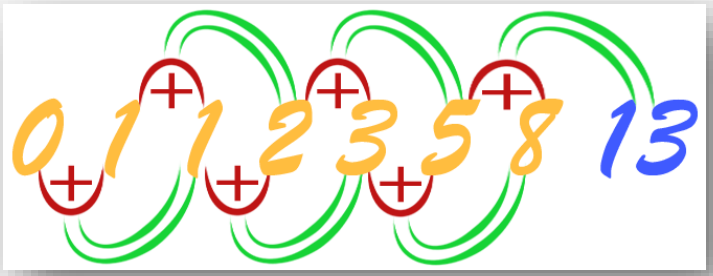
Memória Adicional

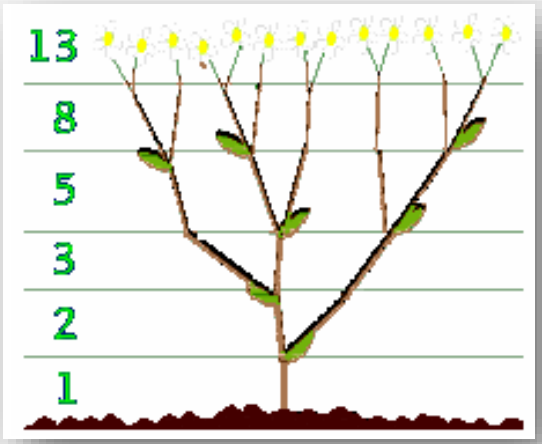
val a = IntArray(3)

for (i in a.indices) a[i] …

PROGRAMAÇÃO DINÂMICA

- Problema: Números de Fibonacci
  - Ilustração da técnica através dos números de Fibonacci, dados pela sequência:





Definição matemática

F(0) = 0, F(1) = 1
F(n) = F(n-1) + F(n-2) para n >= 2

PROGRAMAÇÃO DINÂMICA

- O problema do cálculo de Fibonacci de n
  - **F(n)**
- É expresso em termos do cálculo dos sub-problemas
  - **F(0), F(1), F(2), ... F(n)**
- É necessário um *array* com **n+1** posições para registar as soluções consecutivas dos sub-problemas de F(n)
  - Iniciando com **F(0) = 0** e **F(1) = 1**
  - Usando as regras **F(n) = F(n-1) + F(n-2)**, calcula-se e memoriza-se a solução de cada sub-problema em posições sucessivas do *array*
  - A última posição do *array* contém a solução final. Exemplo para **F(7)**:

PROGRAMAÇÃO DINÂMICA

- Algoritmo para o cálculo dos Números de Fibonacci através da programação dinâmica

```
// Cálculo do n-ésimo número de Fibonacci
fun fibonacci1(n: Int): Int {
    val f = IntArray(n+1)
    f[0] = 0
    f[1] = 1
    for (i in 2..n)
        f[i] = f[i-1] + f[i-2]
    return f[n]
}
```

PROGRAMAÇÃO DINÂMICA

- Problema: Ladrão da Mochila
- Solução através da técnica de programação dinâmica

  - Calcular a melhor combinação para todas as mochilas de capacidade 1 até M (capacidade máxima)

  - Começar por considerar que só se pode usar o objeto 1,
  - depois os objetos 1 e 2,
  - depois os objetos 1, 2 e 3,
  - e finalmente todos os objetos de 1 a N (N = nº de objetos)

PROGRAMAÇÃO DINÂMICA

- Problema: Ladrão da Mochila
- Solução através da técnica de programação dinâmica

Array (A) — (índices colunas)

i/P	0	1	2	3	4	5	6	7	8	9	10
Objecto$_i$ (Pi, Vi) 0	0	0	0	0	0	0	0	0	0	0	0
Objeto1 (7, 42€) 1	0	0	0	0	0	0	0	42€	42€	42€	42€
Objeto2 (3, 12€) 2	0	0	0	12€	12€	12€	12€	42€	42€	42€	54€
Objeto3 (4, 40€) 3	0	0	0	12€	40€	40€	40€	52€	52€	52€	54€
Objeto4 (5, 25€) 4	0	0	0	12€	40€	40€	40€	52€	52€	65€	**65€**

V

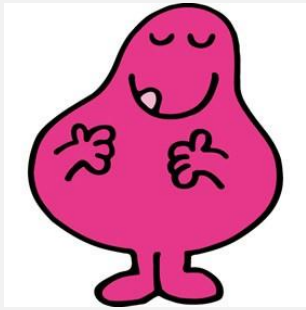
(índices linhas)

```
if Pi <= P // objeto i pode fazer parte da solução
    if Vi + A[i-1][P-Pi] > A[i-1][P]
        A[i][P] = Vi + A[i-1][P-Pi]
    else
        A[i][P] = A[i-1][P]
else // Pi> P
    A[i][P] = A[i-1][P]
```

Solução
Saque mais valioso

TÉCNICA GANANCIOSA

- A técnica Gananciosa (*Greedy*) calcula a melhor solução de um problema através de uma repetição de uma série de passos (escolha de soluções parciais), até obter a solução ótima final



- Em cada passo, a escolha tem que ser
  - **Ótima** – tem que ser a melhor escolha entre todas as possíveis e disponíveis até ao momento
  - **Possível** – não pode violar as restrições do problema (solução impossível)
  - **Irreversível** – uma vez escolhida, não pode ser alterada (voltar atrás) nos passos subsequentes do algoritmo

TÉCNICA GANANCIOSA

- Problema: Troco em Moedas
  - Calcular o troco em moedas de uma dada quantia, no menor número possível de moedas. Em cada passo é escolhida uma moeda
  - Exemplo para um troco = **2.59€**

1. Pode ser escolhida uma moeda de qualquer valor. Contudo, a técnica gananciosa leva à escolha de **2€** pois reduz ao máximo a quantia restante, para **59 cent**
2. Dispomos igualmente de moedas de todos os valores, mas não podem ser escolhidas 2€ e 1€ pois violam as restrições do problema (ultrapassam o valor do troco). A melhor escolha é de **50 cent**, restando **9 cent**
3. Seguindo a mesma técnica, as restantes moedas são: **5 cent**, **2 cent** e **2 cent**

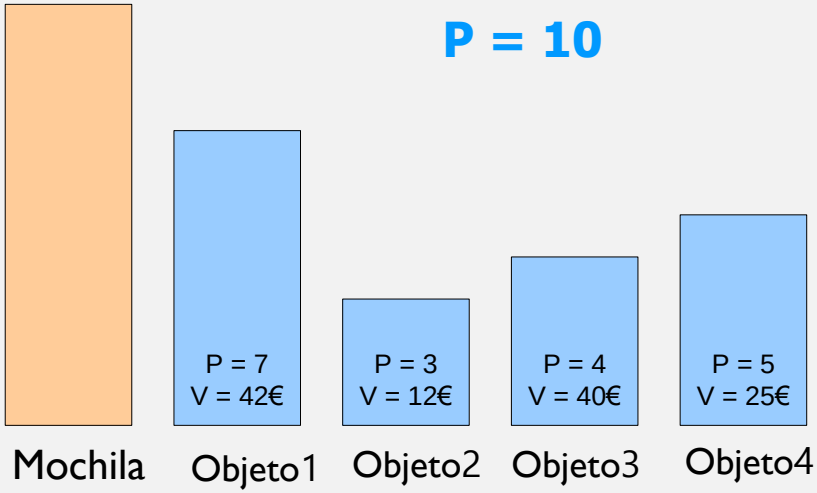
TÉCNICA GANANCIOSA

- Problema: Ladrão da mochila
- Solução através da técnica gananciosa
  1. **Calcular**: a relação valor-peso V_i / P_i, para todos os objetos (para conhecer quais os mais valiosos com menor peso)
  2. **Ordenar**: os objetos por ordem decrescente da relação valor-peso
  3. **Repetir**: até a mochila estar cheia ou não existirem mais objetos:
     - Percorrer os objetos por ordem decrescente de relação-peso
     - Se o objeto cabe na mochila, coloca-se, senão descarta-se
     - Proceder para o próximo objeto

TÉCNICA GANANCIOSA

Capacidade da mochila:

P = 10



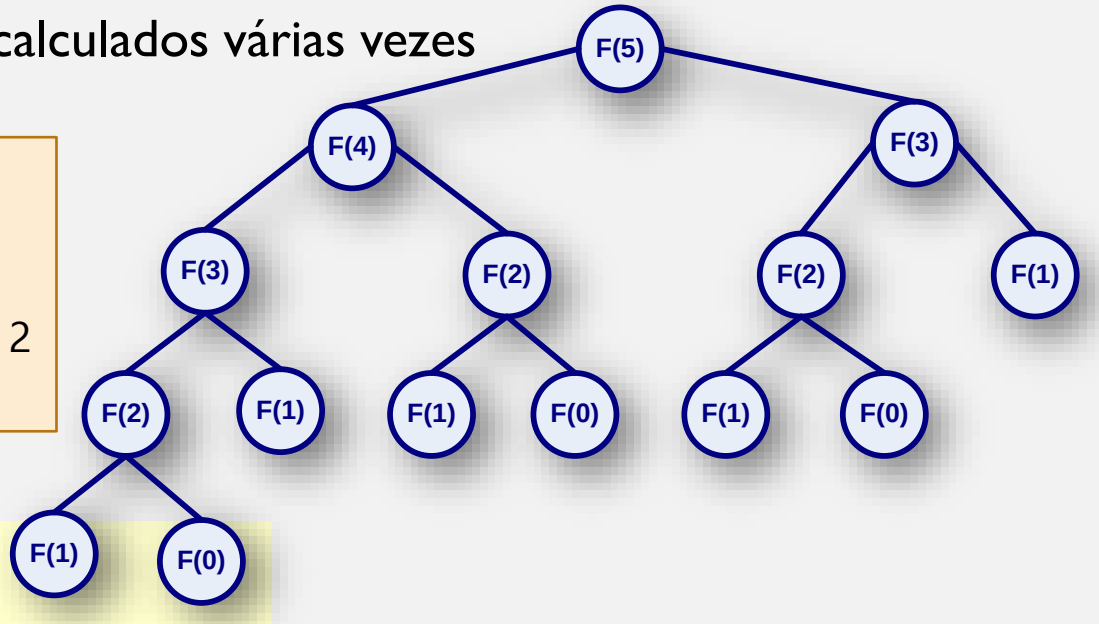
Objeto	Peso (P)	Valor (V)	V/P
1	7	42€	6
2	3	12€	4
3	4	40€	10
4	5	25€	5

Solução
Subconjunto de objetos com o saque mais valioso: 65€

Objeto	Peso (P)	Valor (V)	V/P
3	**4**	**40€**	**10**
1	7	42€	6
4	**5**	**25€**	**5**
2	3	12€	4

RECURSIVIDADE

- Algoritmo para o cálculo dos Números de Fibonacci através da solução naive recursiva
  - É menos eficiente em termos de tempo
  - Os sub-problemas são recalculados várias vezes



Definição matemática:

F(0) = 0, F(1) = 1
F(n) = F(n-1) + F(n-2) para n >= 2

```
// Cálculo do n-ésimo número
// de Fibonacci
fun fibonacci(n: Int): Int =
    if (n <= 1) n
    else fibonacci(n-1) + fibonacci(n - 2)
```

EXEMPLO DE RECURSIVIDADE

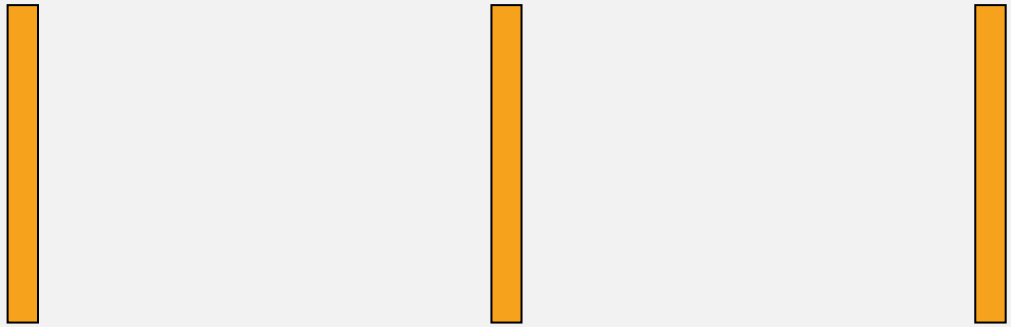
- Problema: Torres de Hanoi
  - O matemático Édouard Lucas inspirou-se numa lenda para construir o jogo das Torres de Hanói em 1883, o qual consiste numa base contendo três pinos. No primeiro são então dispostos alguns discos uns sobre os outros em ordem decrescente de diâmetro. O problema consiste em passar todos os discos de um pino para outro, movendo um disco de cada vez e usando um dos pinos como auxiliar, de maneira que um disco menor fique sempre em cima de outro maior. O número de discos pode variar, sendo três na situação mais simples.

EXEMPLO DE RECURSIVIDADE

- Solução para 3 discos
  - O objetivo é mover todos os discos um pino para a direita de acordo com as seguintes regras:
    - Só se pode mover um disco de cada vez
    - Só se pode colocar um disco num pino vazio ou sobre um disco de diâmetro superior

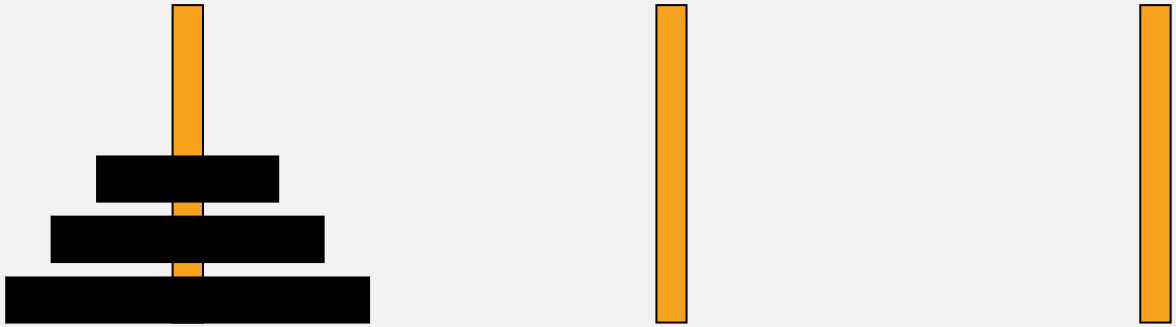
EXEMPLO DE RECURSIVIDADE

- Solução para 3 discos
  - O objetivo é mover todos os discos um pino para a direita de acordo com as seguintes regras:
    - Só se pode mover um disco de cada vez
    - Só se pode colocar um disco num pino vazio ou sobre um disco de diâmetro superior



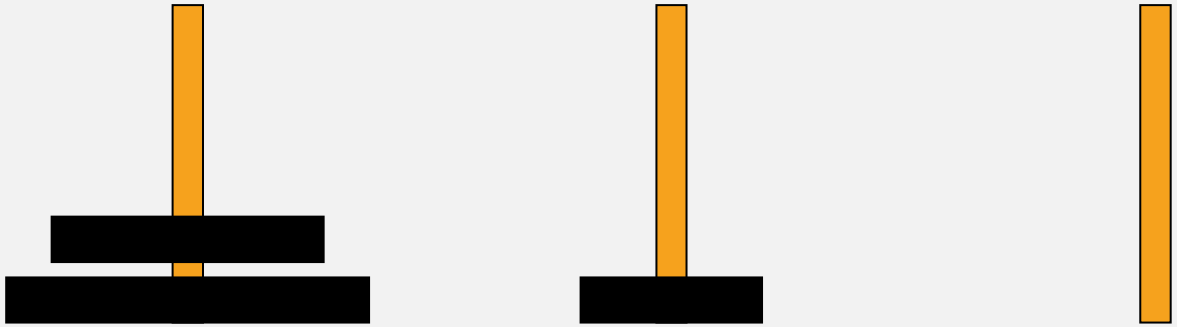
EXEMPLO DE RECURSIVIDADE

- Solução para 3 discos
  - O objetivo é mover todos os discos um pino para a direita de acordo com as seguintes regras:
    - Só se pode mover um disco de cada vez
    - Só se pode colocar um disco num pino vazio ou sobre um disco de diâmetro superior



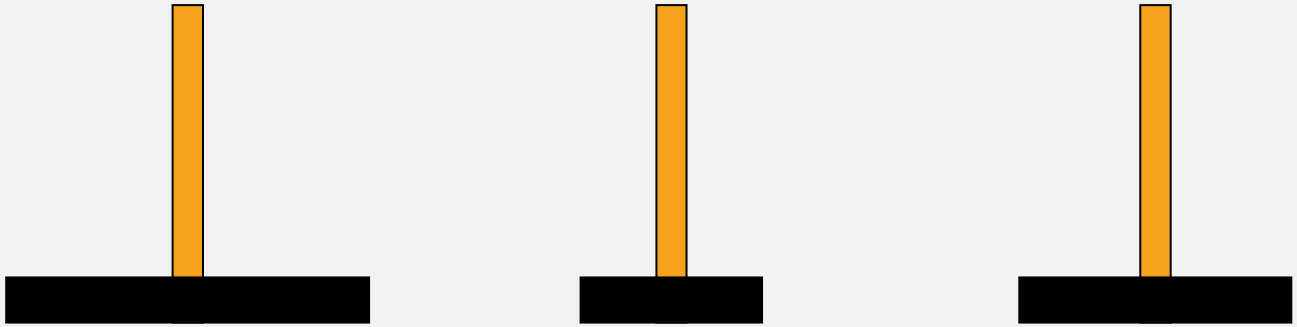
EXEMPLO DE RECURSIVIDADE

- Solução para 3 discos
  - O objetivo é mover todos os discos um pino para a direita de acordo com as seguintes regras:
    - Só se pode mover um disco de cada vez
    - Só se pode colocar um disco num pino vazio ou sobre um disco de diâmetro superior



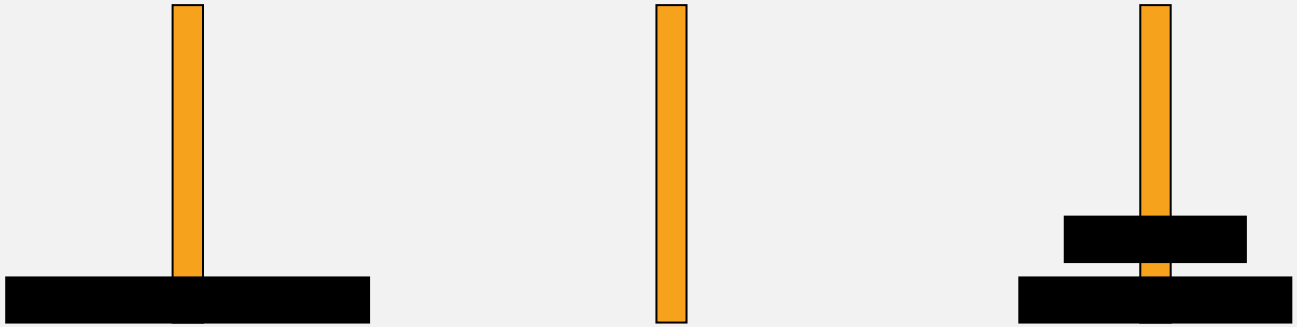
EXEMPLO DE RECURSIVIDADE

- Solução para 3 discos
  - O objetivo é mover todos os discos um pino para a direita de acordo com as seguintes regras:
    - Só se pode mover um disco de cada vez
    - Só se pode colocar um disco num pino vazio ou sobre um disco de diâmetro superior



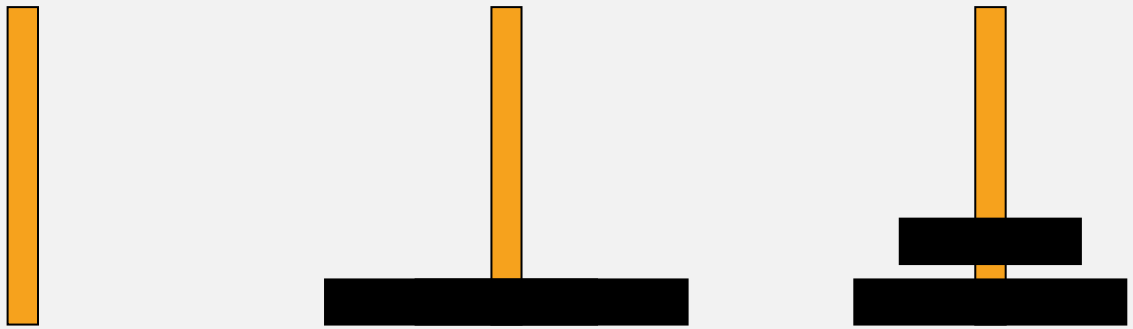
EXEMPLO DE RECURSIVIDADE

- Solução para 3 discos
  - O objetivo é mover todos os discos um pino para a direita de acordo com as seguintes regras:
    - Só se pode mover um disco de cada vez
    - Só se pode colocar um disco num pino vazio ou sobre um disco de diâmetro superior



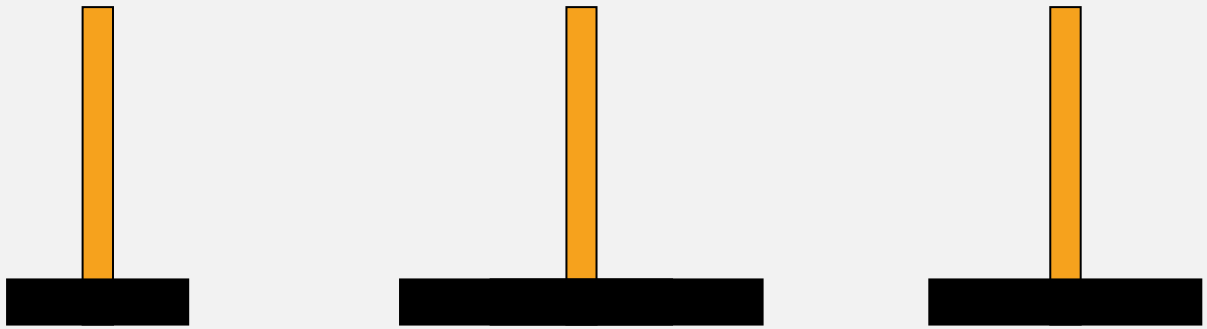
EXEMPLO DE RECURSIVIDADE

- Solução para 3 discos
  - O objetivo é mover todos os discos um pino para a direita de acordo com as seguintes regras:
    - Só se pode mover um disco de cada vez
    - Só se pode colocar um disco num pino vazio ou sobre um disco de diâmetro superior



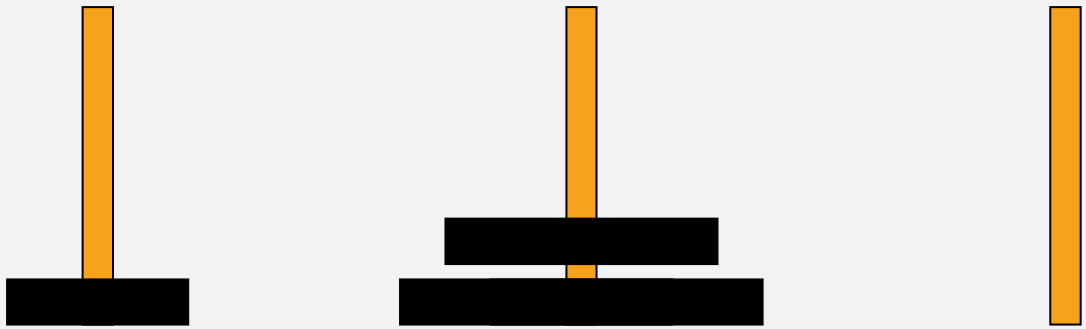
EXEMPLO DE RECURSIVIDADE

- Solução para 3 discos
  - O objetivo é mover todos os discos um pino para a direita de acordo com as seguintes regras:
    - Só se pode mover um disco de cada vez
    - Só se pode colocar um disco num pino vazio ou sobre um disco de diâmetro superior



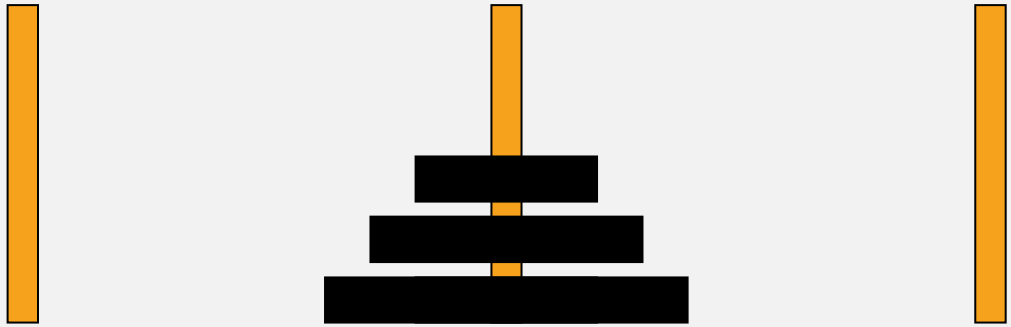
EXEMPLO DE RECURSIVIDADE

- Solução para 3 discos
  - O objetivo é mover todos os discos um pino para a direita de acordo com as seguintes regras:
    - Só se pode mover um disco de cada vez
    - Só se pode colocar um disco num pino vazio ou sobre um disco de diâmetro superior



EXEMPLO DE RECURSIVIDADE

- Solução para 3 discos
  - O objetivo é mover todos os discos um pino para a direita de acordo com as seguintes regras:
    - Só se pode mover um disco de cada vez
    - Só se pode colocar um disco num pino vazio ou sobre um disco de diâmetro superior



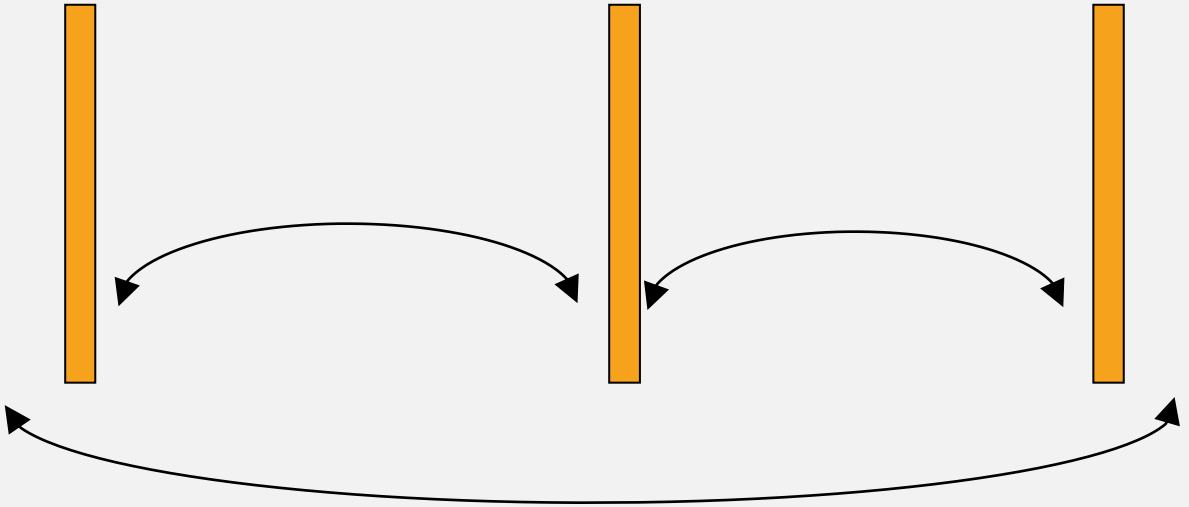
EXEMPLO DE RECURSIVIDADE

- Solução para 5 discos
  - Para mover os 5 discos um pino para a direita
    - Movem-se os 4 discos menores um pino à esquerda
    - Move-se o disco 5 um pino à direita
    - Movem-se os 4 discos menores um pino à esquerda

EXEMPLO DE RECURSIVIDADE

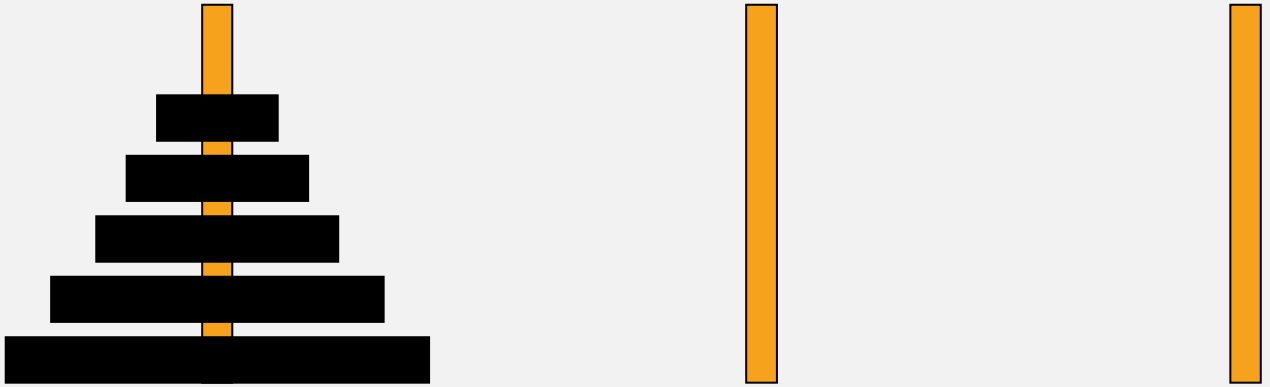
- Solução para 5 discos
  - Para mover os 5 discos um pino para a direita
    - Movem-se os 4 discos menores um pino à esquerda
    - Move-se o disco 5 um pino à direita
    - Movem-se os 4 discos menores um pino à esquerda

Considera-se uma movimentação circular nos pinos para a esquerda ou para a direita



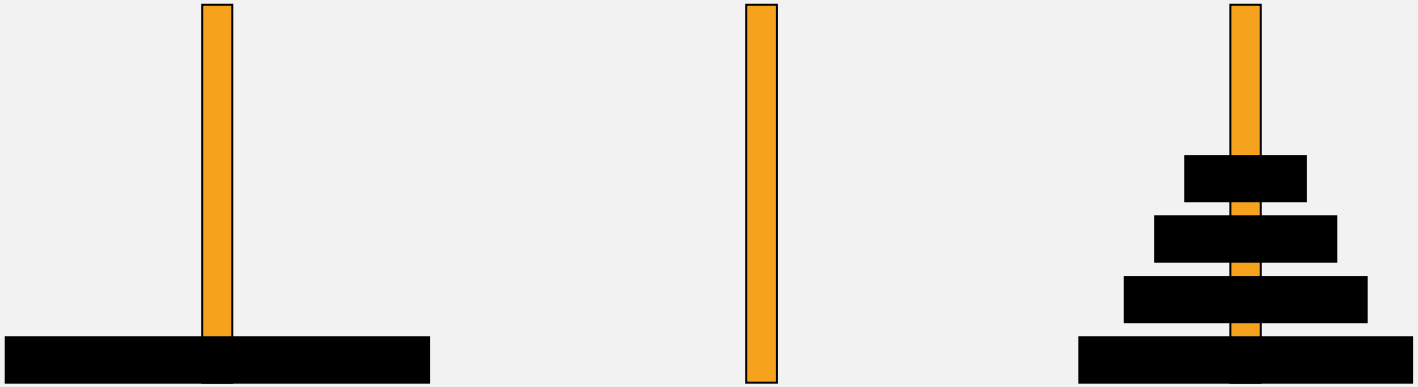
EXEMPLO DE RECURSIVIDADE

- Solução para 5 discos
  - Para mover os 5 discos um pino para a direita
    - Movem-se os 4 discos menores um pino à esquerda
    - Move-se o disco 5 um pino à direita
    - Movem-se os 4 discos menores um pino à esquerda



EXEMPLO DE RECURSIVIDADE

- Solução para 5 discos
  - Para mover os 5 discos um pino para a direita
    - Movem-se os 4 discos menores um pino à esquerda
    - Move-se o disco 5 um pino à direita
    - Movem-se os 4 discos menores um pino à esquerda

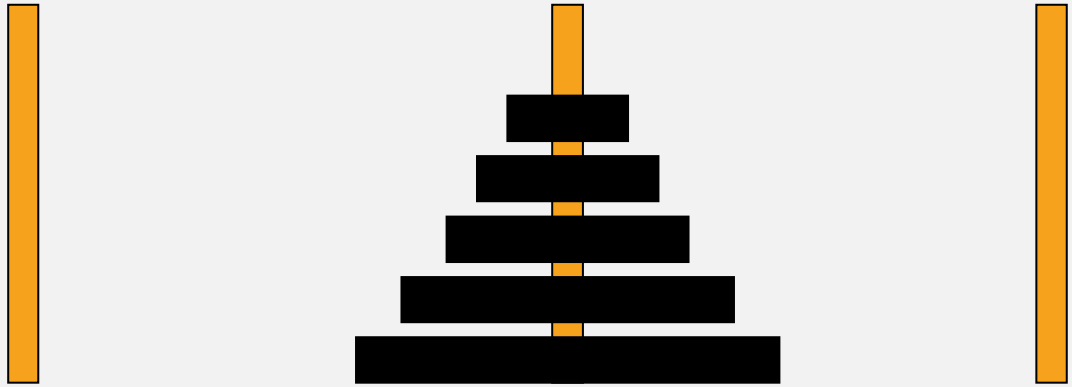


EXEMPLO DE RECURSIVIDADE

- Solução para 5 discos
  - Para mover os 5 discos um pino para a direita
    - Movem-se os 4 discos menores um pino à esquerda
    - Move-se o disco 5 um pino à direita
    - Movem-se os 4 discos menores um pino à esquerda

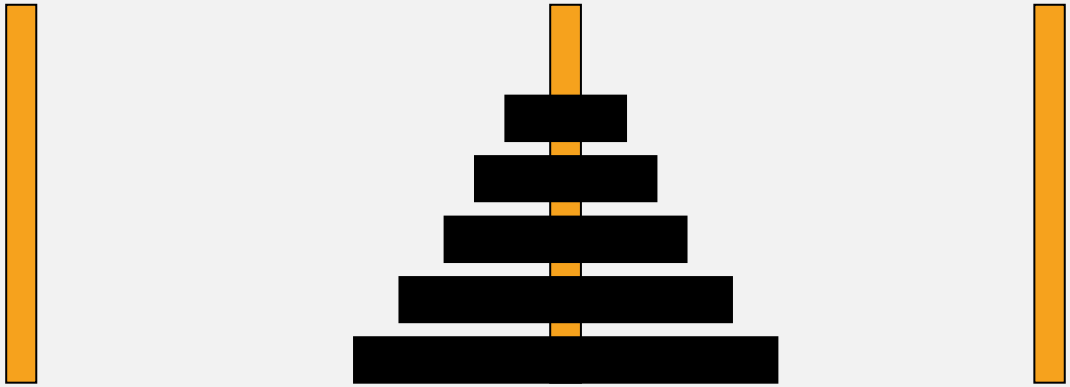
EXEMPLO DE RECURSIVIDADE

- Solução para 5 discos
  - Para mover os 5 discos um pino para a direita
    - Movem-se os 4 discos menores um pino à esquerda
    - Move-se o disco 5 um pino à direita
    - Movem-se os 4 discos menores um pino à esquerda



EXEMPLO DE RECURSIVIDADE

- Solução para 5 discos
  - Para mover os 5 discos um pino para a direita
    - Movem-se os 4 discos menores um pino à esquerda
    - Move-se o disco 5 um pino à direita
    - Movem-se os 4 discos menores um pino à esquerda



    - Mas não se podem mover 4 discos de cada vez. Então …

EXEMPLO DE RECURSIVIDADE

- Para mover os 4 discos um pino para a esquerda
  - Movem-se os 3 discos menores um pino à direita
  - Move-se o disco 4 um pino à esquerda
  - Movem-se os 3 discos menores um pino à direita
  - Mas não se podem mover 3 discos de cada vez. Então …

EXEMPLO DE RECURSIVIDADE

- Para mover os 3 discos um pino para a direita
  - Movem-se os 2 discos menores um pino à esquerda
  - Move-se o disco 3 um pino à direita
  - Movem-se os 2 discos menores um pino à esquerda
  - Mas não se podem mover 2 discos de cada vez. Então …
- … Repete-se até reduzir a 1 disco, o qual pode ser movido

EXEMPLO DE RECURSIVIDADE

- Solução recursiva para N discos
  - Para mover os N discos um pino para a direita
    - Movem-se os N-1 discos menores um pino à esquerda
    - Move-se o disco N um pino à direita
    - Movem-se os N-1 discos menores um pino à esquerda

Caso de menor dimensão

Caso geral

```
fun hanoi(n: Int, d: Int) {
    if (n == 1 ) move (n, +d)
    else {
        hanoi(n - 1, -d)
        move (n, +d)
        hanoi(n - 1, -d)
    }
}
```

n : número de discos
+d : movimento um pino à dta
-d : movimento um pino à esq

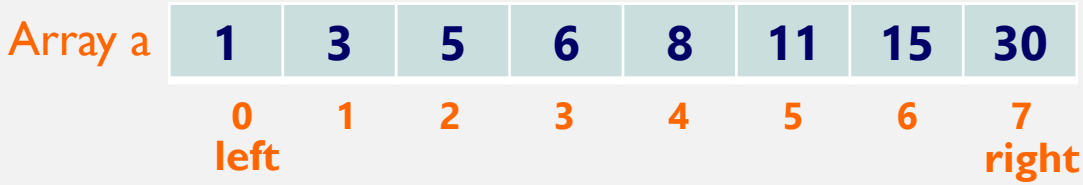
DIVIDIR PARA CONQUISTAR

- A técnica Dividir para Conquistar (*Divide and Conquer*) é provavelmente uma das melhores técnicas conhecidas de desenho de algoritmos
  - Alguns dos algoritmos mais eficientes, são implementações específicas desta estratégia

1. **Divide** o problema em sub-problemas (idealmente com a mesma dimensão), os quais são instâncias mais pequenas do mesmo problema
2. **Conquista** os sub-problemas resolvendo-os recursivamente (tipicamente). Se estes são suficientemente pequenos (casos de menor dimensão), a solução é logo encontrada
3. **Combina** as soluções dos sub-problemas para obter a solução do problema original

PESQUISA

- Problema:
  - Pretende-se pesquisar se existe um dado elemento num array ordenado. Caso exista, a função devolve o seu índice. Senão, devolve -1



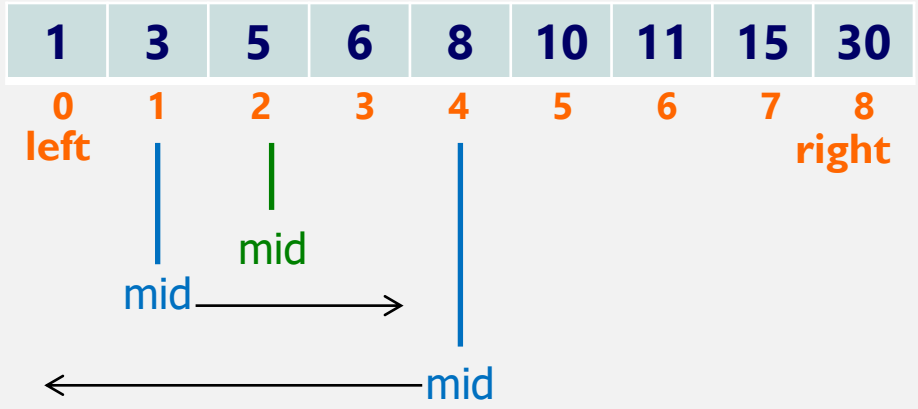
- Solução: força bruta

```
fun linearSearch (a: IntArray, left: Int, right: Int, x: Int): Int {
    for (i in left..right)
        if (x == a[i]) return i;
    return -1;
}
```

PESQUISA BINÁRIA

- Solução: dividir para conquistar (o array tem que estar ordenado)
  - **Divide** o problema particionando o array ao meio sucessivamente, até encontrar o elemento ou o array ficar vazio
  - **Conquista** os sub-problemas verificando em cada instância, se o elemento procurado está na posição central. Caso esteja, a solução é encontrada devolvendo o índice do elemento. Senão, se o elemento procurado for menor que o elemento central, a pesquisa continua na metade esquerda do array, caso contrário continua na metade direita
  - **Combina** (não implica nenhuma operação adicional)

Exemplo:
x = 5



PESQUISA BINÁRIA

```
fun binarySearchRecursive (a: IntArray, left: Int, right: Int, x: Int): Int {
    if (left > right) return -1
    val mid = (left + right) / 2
    if (x == a[mid]) return mid
    else if (x < a[mid]) return binarySearchRecursive(a, left,mid-1, x)
    else return binarySearchRecursive(a, mid+1, right, x)
}
```

- Algoritmo recursivo

- Algoritmo iterativo

```
fun binarySearchIterative (a: IntArray, left: Int, right: Int, x: Int): Int {
    var l = left
    var r = right
    while (l <= r) {
        val mid = (l + r) / 2
        if (x == a[mid]) return mid else if (x < a[mid]) r = mid -1
        else l = mid + 1
    }
    return -1
}
```

DIVIDIR PARA CONQUISTAR

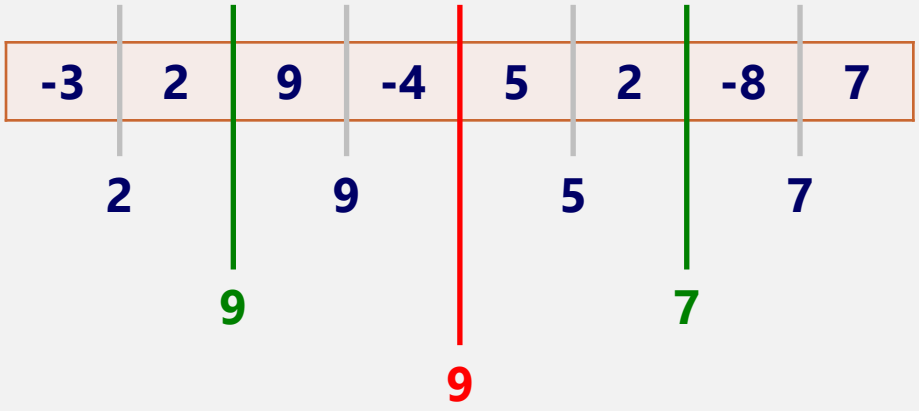
- Problema: Máximo de um Array
  - Dado um array preenchido com números inteiros, encontrar o maior
  - Exemplo: Array a

- Solução: força bruta
  - Pesquisa exaustiva percorrendo e comparando todas as posições do *array* até encontrar o maior valor

```
fun maxOfArray(a: IntArray, left: Int, right: Int): Int {
    var maxV = a[left]
    for (i in left + 1..right)
        if (a[i] > maxV) maxV = a[i]
    return maxV
}
```

DIVIDIR PARA CONQUISTAR

- Solução: dividir para conquistar
  - **Divide** o problema particionando o array ao meio sucessivamente, até que cada metade tenha dimensão 1
  - **Conquista** os sub-problemas calculando recursivamente o maior valor de cada metade. Como estas têm dimensão 1 (caso de menor dimensão), o máximo valor em cada uma é logo encontrado
  - **Combina** os resultados calculando em cada instância o maior valor de ambas as metades



DIVIDIR PARA CONQUISTAR

- O algoritmo particiona o array a[left, ..., right] em duas metades:

  a[left, ..., mid] e a[mid+1, ..., right]

- Até que cada metade tenha dimensão 1 (left = right)
- Calcula o máximo valor de cada metade
- Devolve o maior valor de ambas as metades, que na ultima instância é o máximo de todo o *array*

```
fun maxOfArray(a: IntArray, left: Int, right: Int): Int {
    if (left == right) return a[left]
    else {
        val mid = (left + right) / 2
        val maxL = maxOfArray(a, left, mid)          //máx da metade esq
        val maxR = maxOfArray(a, mid + 1, right) //máx da metade dta
        return if (maxL > maxR) maxL else maxR
    }
}
```

SUBARRAY MÁXIMO

- Problema: SubArray Máximo
  - O problema do *subarray* máximo consiste em encontrar a maior soma de números consecutivos, num *array* unidimensional preenchido com números inteiros (positivos e negativos)
  - Exemplo: resultados do lucro da empresa X Corp.

Ano	1	2	3	4	5	6	7	8
Lucro M€	**2**	**1**	**-4**	**5**	**2**	**-1**	**3**	**-3**

  - Qual o máximo de lucro ganho pela X Corp. em anos consecutivos?
  - Entre o ano 1 e 8

    2 + 1 – 4 + 5 + 2 – 1 + 3 – 3 = 5 M€
  - Entre o ano 2 e 6

    1 – 4 + 5 + 2 – 1 = 3 M€
  - Entre os anos 4 e 7 a X Corp. ganhou:

    5 + 2 – 1 + 3 = 9 M€

SUBARRAY MÁXIMO

- Solução: força bruta
  - Calcula-se a soma de todos os sub-conjuntos possíveis do *array*, comparando e guardando a maior

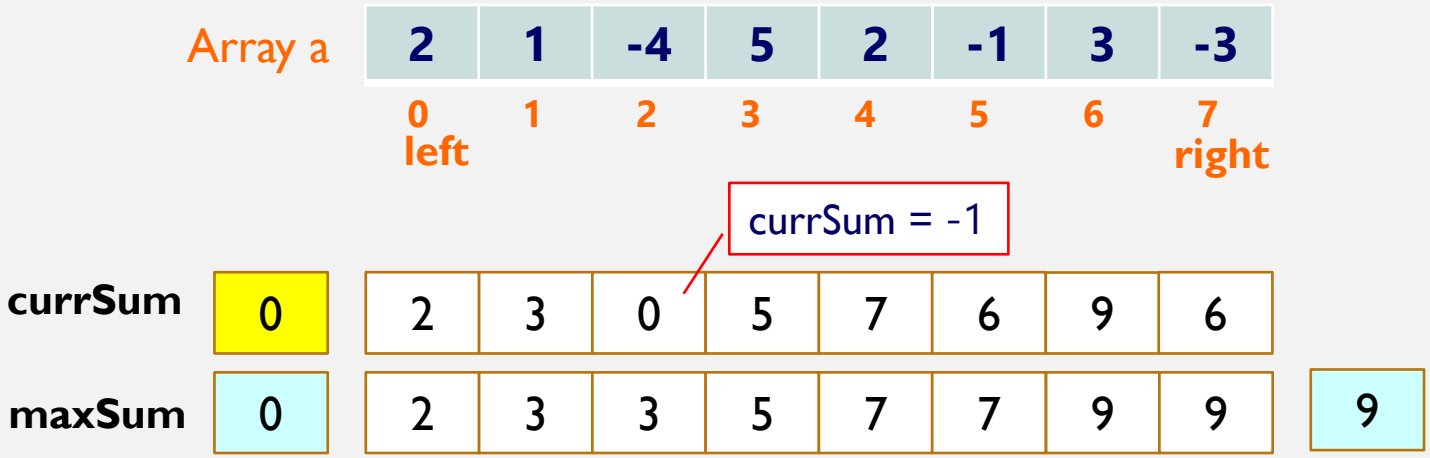
Array a	2	1	-4	5	2	-1	3	-3
	0 left	1	2	3	4	5	6	7 right

```
fun maximumSubArray(a: IntArray, left: Int, right: Int): Int {
    var maxSum = 0 // soma máxima
    for (i in left..right) {
        var currSum = 0 // soma corrente
        for (j in i..right) {
            currSum += a[j]
            if (currSum > maxSum) maxSum = currSum
        }
    }
    return maxSum
}
```

```
i = 0 1 2 3 4 5 6 7
j = 0
j = 0 1
j = 0 1 2
j = 0 1 2 3
j = 0 1 2 3 4
j = 0 1 2 3 4 5
j = 0 1 2 3 4 5 6
j = 0 1 2 3 4 5 6 7
```

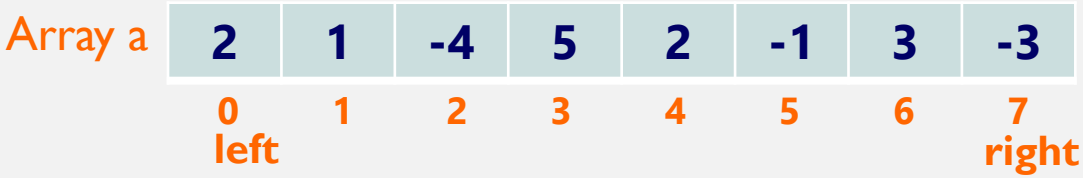
SUBARRAY MÁXIMO

- Solução: programação dinâmica (Algoritmo de Kadane)
  - Consiste em percorrer todo o *array*, calculando em cada posição a soma máxima obtida até à posição corrente
  - Em cada posição, adiciona-se o valor corrente à soma obtida até ao momento
    - Caso a soma corrente dê um valor negativo, recomeça-se com soma = 0
    - Caso contrário, o elemento corrente fica a fazer parte da soma



SUBARRAY MÁXIMO

- Algoritmo de Kadane




```
fun maximumSubArrayKadane(a: IntArray, left: Int, right: Int): Int {
    var maxSum = 0  // soma máxima
    var currSum = 0  // soma corrente
    for (i in left..right) {  // i = 0 1 2 3 4 5 6 7
        currSum = if (currSum + a[i] > 0) currSum + a[i] else 0
        if (currSum > maxSum) maxSum = currSum
    }
    return maxSum
}
```