Unidade II:

Introdução à Análise de Algoritmos

Prof. Max do Val Machado

Instituto de Ciências Exatas e Informática
Curso de Ciência da Computação

Exercício

- Para a aula de análise de complexidade, resolva as equações abaixo:

a) $2^0 =$

b) $2^1 =$

c) $2^2 =$

d) $2^3 =$

e) $2^4 =$

f) $2^5 =$

g) $2^6 =$

h) $2^7 =$

i) $2^8 =$

j) $2^9 =$

k) $2^{10} =$

l) $2^{11} =$

Exercício

- Para a aula de análise de complexidade, resolva as equações abaixo:

a) lg(2048) =
b) lg(1024) =
c) lg(512) =
d) lg(256) =
e) lg(128) =
f) lg(64) =
g) lg(32) =
h) lg(16) =
i) lg(8) =
j) lg(4) =
k) lg(2) =
l) lg(1) =

Nota: lg (n) é a mesma coisa que o logaritmo de n na base dois, ou seja, $\log_2(n)$

Exercício

- Para a aula de análise de complexidade, plote um gráfico com todas as funções abaixo:

a) $f(n) = n$

b) $f(n) = n^2$

c) $f(n) = n^3$

d) $f(n) = sqrt(n)$

e) $f(n) = lg\ (n) = log_2(n)$

f) $f(n) = 3n^2 + 5n - 3$

g) $f(n) = - 3n^2 + 5n – 3$

h) $f(n) = | - n^2 |$

i) $f(n) = 5n^4 + 2n^2$

j) $f(n) = n * lg\ (n)$

Exercício

- Faça um resumo sobre Somatórios. Use LaTeX e siga o modelo de artigos da SBC (sem abstract, resumo e seções) com no máximo duas página

Definições para Algoritmo

Definições para Algoritmo

- Um algoritmo é uma descrição de um padrão de comportamento expresso em termos de um conjunto finito de ações (Dijkstra, 1971)

- Informalmente, um algoritmo corresponde a uma sequência finita de ações executáveis para solucionarmos um determinado problema

- Um algoritmo recebe um ou mais valores como entrada, processa tais valores e gera uma saída

Restrição dos Algoritmos

- Quando propomos um algoritmo para resolver um problema, tal algoritmo tem que ser implementado em um computador:
  - Os computadores possuem restrições quanto à capacidade computacional e a de armazenamento
  - Logo, devemos analisar qual será a complexidade de se implementar tal algoritmo
  - De que adianta um algoritmo que leva séculos para ser executado???

Aspectos para a Análise de Complexidade

- Tempo de execução
- Espaço de memória ocupado

Tipos de Problemas na Análise de Complexidade

- Análise de um algoritmo particular:
  - Dado um problema, qual é o custo de um algoritmo específico?
- Análise de uma classe de algoritmos:
  - Dado um problema, qual é o algoritmo de menor custo para resolvê-lo?

Análise de uma Classe de Algoritmos

- Limite da família de algoritmos: a família toda é analisada com o objetivo de identificarmos o melhor algoritmo

- Toda classe de problemas possui um nível mínimo de dificuldade para ser resolvida. Quando definimos o menor custo para resolvê-la, temos a medida da dificuldade inerente a essa classe

Algoritmo Ótimo

- Algoritmo cujo custo é igual ao menor custo possível

Formas para Medir o Custo de um Algoritmo

Formas para Medir o Custo de um Algoritmo

- Um computador real depende de vários itens tais como:
  - Hardware
  - Arquitetura
  - Sistema Operacional
  - Compilador
  - Linguagem

Formas para Medir o Custo de um Algoritmo

- Modelo matemático para contar o número de operações de cada tipo:
  - Determinamos as operações relevantes e o número de vezes que são executadas
  - Desconsideramos sobrecargas de gerenciamento de memória ou E/S
  - Para a ordenação interna, por exemplo, normalmente, consideramos apenas o número de comparações (e/ou o de movimentações) entre os elementos do *array* e ignoramos as operações aritméticas, de atribuição e manipulações de indices

Formas para Medir o Custo de um Algoritmo

- Modelo matemático para contar o número de operações de cada tipo:
  - Em geral, estamos interessados na complexidade no pior caso
  - Para medir o custo de execução, definimos uma função de complexidade

Observação

- Frequentemente, os alunos se preocupam em fazer otimizações que são indiferentes para o compilador
  - Por exemplo, ele pode gerar o mesmo código objeto para if-else-if e switch-case; for e while; entre outros...
  - Por exemplo, ele pode fazer várias otimizações a serem abordas durante nosso curso de Ciência da Computação

Observação

- Exemplo de Otimização do Compilador

```
for (int i = 0; i < 20 ; i++){
        array[i] = i;
}
```

Qual é a vantagem de cada um dos códigos?

```
array [0] = 0;
array [1] = 1;

■ ■ ■

array [19] = 19;
```

Função de Complexidade

- Função de complexidade de tempo :
  - f(n) é a medida de tempo necessário para executar um algoritmo para um problema de tamanho n
  - Não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada

Função de Complexidade

- Função de complexidade de espaço:
  - f(n) é a medida da quantidade de memória necessária para executar um algoritmo de tamanho n

Como Calcular a Complexidade de um Algoritmo

Como Calcular a Complexidade de um Algoritmo

- Da mesma forma que calculamos o custo de um churrasco:
  - Carne: 400 gramas por pessoa (preço médio do kg R$ 20,00 - picanha, asinha, coraçãozinho ...)
  - Cerveja: 1,2 litros por pessoa (litro R$ 3,80)
  - Refrigerante: 1 litro por pessoa (Garrafa 2 litros R$ 3,50)

Exercício: Monte a função de complexidade (ou custo) do nosso churrasco.

Como Calcular a Complexidade de um Algoritmo

- Da mesma forma que calculamos o custo de um churrasco:
  - Carne: 400 gramas por pessoa (preço médio do kg R$ 20,00 - picanha, asinha, coraçãozinho ...)
  - Cerveja: 1,2 litros por pessoa (litro R$ 3,80)
  - Refrigerante: 1 litro por pessoa (Garrafa 2 litros R$ 3,50)

Exercício: Monte a função de complexidade (ou custo) do nosso churrasco.

$$f(n) = n * \frac{400}{1000} * 20 + n * 1,2 * 3,8 + n * 1 * \frac{3,5}{2}$$

$$= 14,31 * n$$

Como Calcular a Complexidade de um Algoritmo

- Da mesma forma que calculamos o custo de uma viagem:
  - Passagem:
  - Hotel:
  - Saídas:

Calculo de Complexidade

- O custo total de um algoritmo é igual a soma do custo de suas operações

- Operações normalmente com custo 1:
  - Leitura
  - Escrita
  - Atribuição
  - Operações lógica ou aritmética

Calculo de Complexidade

- Operação condicional: custo da condição mais o máximo entre as operações para *true* e as para *false*

```
if ( condição() ){
        listaVerdadeiro();
} else {
        listaFalso();
}
```

Custo:
condição() + máximo(listaVerdadeiro(), listaFalso())

- Operação de repetição: o custo de uma interação vezes o número de interações

```
i = 0;
while ( i < n){
        lista ();
}
```

Custo: lista() * n

Exercício

- Calcule o número de adições que o código abaixo realiza:

```
...
if (a + 5 < b + 3 || c + 1 < d + 3){
        i++;
        ++b;
        a += 3;
} else {
        j++;
}
```

Exercício

- Calcule o número de subtrações que o código abaixo realiza:

```
int i = 0, b = 10;

while (i < 3){
        i++;
        b--;
}
```

Exercício

- Calcule o número de subtrações que o código abaixo realiza:

```
int i = 10;

while (i >= 7){
    i--;
}
```

Exercício

- Calcule o número de subtrações que o código abaixo realiza:

```
int a = 10;

for (int i = 0; i < 3 ; i++){
      for (int j = 0; j < 2 ; j++){
            a--;
      }
}
```

Exercício

- Calcule o número de comparações que o código abaixo realiza:

```
int a = 10;

for (int i = 0; i < 3 ; i++){
      for (int j = 0; j < 2 ; j++){
            a--;
      }
}
```

Exercício

- Calcule o número de subtrações que o código abaixo realiza:

```
int i = 1, b = 10;

while (i > 0){
        b--;
        i = i >> 1;
}

i = 0;

while (i < 15){
        b--;
        i += 2;
}
```

Exercício

- Calcule o número de multiplicações que o código abaixo realiza:

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n - 3; j++)
        a *= 2;
```

Exercício

- Calcule o número de multiplicações que o código abaixo realiza:

```
for (int i = n - 7; i >= 1; i--)
    for (int j = 0; j < n; j++)
        a *= 2;
```

Exercício

- Calcule o número de multiplicações que o código abaixo realiza:

```
for (int i = n - 7; i >= 1; i--)
    for (int j = n - 7; j >= 1; j--)
        a *= 2;
```

Exercício

- Calcule o número de multiplicações que o código abaixo realiza:

```
for (int i = n; i > 0; i /= 2)
    a *= 2;
```

Exercício

- Faça um método que receba um número inteiro n e efetue o número de subtrações pedido em:

a) $3n + 2n^2$

b) $5n + 4n^3$

c) $\lg(n) + n$

d) $2n^3 + 5$

e) $9n^4 + 5n^2 + n/2$

f) $\lg(n) + 5\lg(n)$

Exercício

- Faça um método que receba um número inteiro n e efetue o número de subtrações pedido em:

a) $3n + 2n^2$

b) $5n + 4n^3$

c) $\lg(n) + n$

d) $2n^3 + 5$

e) $9n^4 + 5n^2 + n/2$

f) $\lg(n) + 5\lg(n)$

```
        ...
i = 0;

while (i < n){
      i++;
      a--; b--; c--;
}

for (i = 0;  i < n; i++){
      for (j = 0;  j < n; j++){
            a--; b--;
      }
}
```

Calculo de Complexidade

- Outros laços: sempre consideramos o limite superior
- Métodos: consideramos o custo do método
- Métodos recursivos: utilizamos equações de recorrência (PAA + MD)

Calculo de Complexidade

- Exemplo: Encontrar o menor valor em um *array* de inteiros

```
int min = vet[0];

for (int i = 1; i < n; i++){
    if (min > vet[i]){
        min = vet[i];
    }
}
```

1º) Qual é a operação relevante?

2º) Quantas vezes ela será executada?

3º) O nosso algoritmo é ótimo? Por que?

Calculo de Complexidade

- Exemplo: Encontrar o menor valor em um *array* de inteiros

```
int min = vet[0];

for (int i = 1; i < n; i++){
    if (min > vet[i]){
        min = vet[i];
    }
}
```

1º) Qual é a operação relevante?

R: Comparação entre elementos do array

2º) Quantas vezes ela será executada?

3º) O nosso algoritmo é ótimo? Por que?

Calculo de Complexidade

- Exemplo: Encontrar o menor valor em um *array* de inteiros

```
int min = vet[0];

for (int i = 1; i < n; i++){
    if (min > vet[i]){
        min = vet[i];
    }
}
```

1º) Qual é a operação relevante?

R: Comparação entre elementos do *array*

2º) Quantas vezes ela será executada?

R: Se tivermos n elementos: T(n) = n - 1

3º) O nosso algoritmo é ótimo? Por que?

Calculo de Complexidade

- Exemplo: Encontrar o menor valor em um *array* de inteiros

```
int min = vet[0];

for (int i = 1; i < n; i++){
    if (min > vet[i]){
        min = vet[i];
    }
}
```

1º) Qual é a operação relevante?

R: Comparação entre elementos do array

2º) Quantas vezes ela será executada?

R: Se tivermos n elementos: T(n) = n - 1

3º) O nosso algoritmo é ótimo? Por que?

R: Sim porque temos que testar todos os elementos para garantir nossa resposta.

Calculo de Complexidade

- Cenários possíveis:
  - Melhor caso: menor tempo de execução para todas entradas possíveis de tamanho n
  - Pior caso: maior tempo de execução
  - Caso médio (ou esperado): média dos tempos de execução

Calculo de Complexidade

- Exemplo: Encontrar o menor valor em um *array* de inteiros

```
int min = vet[0];

for (int i = 1; i < n; i++){
    if (min > vet[i]){
        min = vet[i];
    }
}
```

4º) O nosso T(n) = n – 1 é para qual dos três casos?

Calculo de Complexidade

- Exemplo: Pesquisa sequencial em um *array* de inteiros

```
boolean resp = false;

 for (int i = 0; i < n; i++){
       if (vet[i] == x){
            resp = true;
            i = n;
       }
}
```

Calculo de Complexidade

- Exemplo: Pesquisa sequencial em um *array* de inteiros

```
boolean resp = false;

 for (int i = 0; i < n; i++){
       if (vet[i] == x){
            resp = true;
            i = n;
       }
}
```

1º) Qual é a operação relevante?

R: Comparação entre elementos do array

2º) Quantas vezes ela será executada?

R: ???

3º) O nosso algoritmo é ótimo? Por que?

R:

Calculo de Complexidade

- Exemplo: Pesquisa sequencial em um *array* de inteiros

```
boolean resp = false;

 for (int i = 0; i < n; i++){
       if (vet[i] == x){
            resp = true;
            i = n;
       }
}
```

1º) Qual é a operação relevante?

R: Comparação entre elementos do array

2º) Quantas vezes ela será executada?

R: Em qual dos casos?

3º) O nosso algoritmo é ótimo? Por que?

R:

Calculo de Complexidade

- Exemplo: Pesquisa sequencial em um *array* de inteiros

```
boolean resp = false;

 for (int i = 0; i < n; i++){
       if (vet[i] == x){
            resp = true;
            i = n;
       }
}
```

1º) Qual é a operação relevante?

R: Comparação entre elementos do array

2º) Quantas vezes ela será executada?

R: Melhor caso: f(n) = 1
Pior caso: f(n) = n
Caso médio: f(n) = (n + 1) / 2

3º) O nosso algoritmo é ótimo? Por que?

R:

Calculo de Complexidade

- Exemplo: Pesquisa sequencial em um *array* de inteiros

```
boolean resp = false;

 for (int i = 0; i < n; i++){
       if (vet[i] == x){
            resp = true;
            i = n;
       }
}
```

1º) Qual é a operação relevante?

R: Comparação entre elementos do array

2º) Quantas vezes ela será executada?

R: Melhor caso: f(n) = 1
Pior caso: f(n) = n
Caso médio: f(n) = (n + 1) / 2

3º) O nosso algoritmo é ótimo? Por que?

R:

Calculo de Complexidade

- Exemplo: Pesquisa sequencial em um *array* de inteiros

```
boolean resp = false;

 for (int i = 0; i < n; i++){
       if (vet[i] == x){
            resp = true;
            i = n;
       }
}
```

1º) Qual é a operação relevante?

R: Comparação entre elementos do array

2º) Quantas vezes ela será executada?

R: Melhor caso: f(n) = 1
Pior caso: f(n) = n
Caso médio: f(n) = (n + 1) / 2

3º) O nosso algoritmo é ótimo? Por que?

R: Sim porque temos que testar todos os elementos para garantir nossa resposta.

Calculo de Complexidade

E se o *array* estiver ordenado?

Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado

E se o *array* estiver ordenado?

2	3	5	7	9	11	15	17	20	21	30	43	49	70	71	82
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

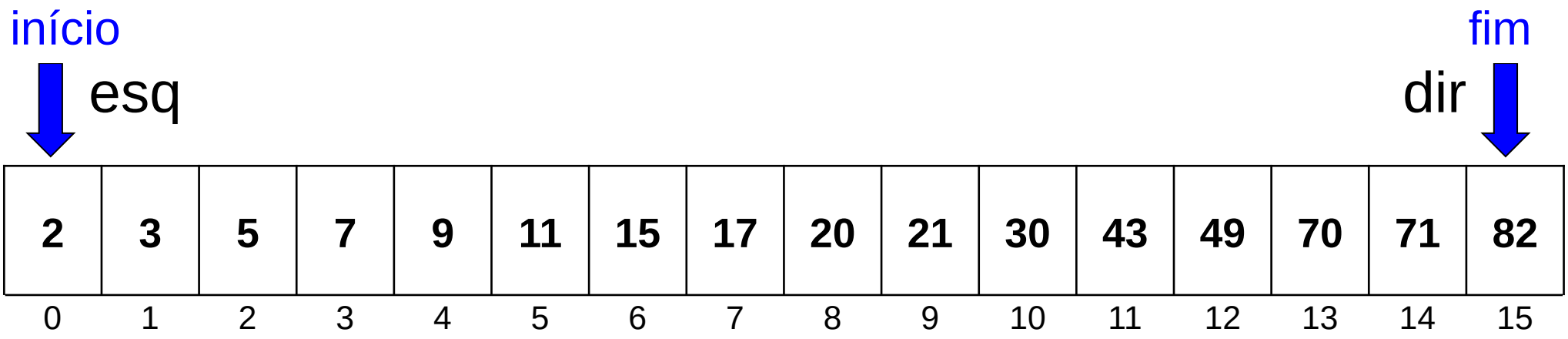
Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

2	3	5	7	9	11	15	17	20	21	30	43	49	70	71	82
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

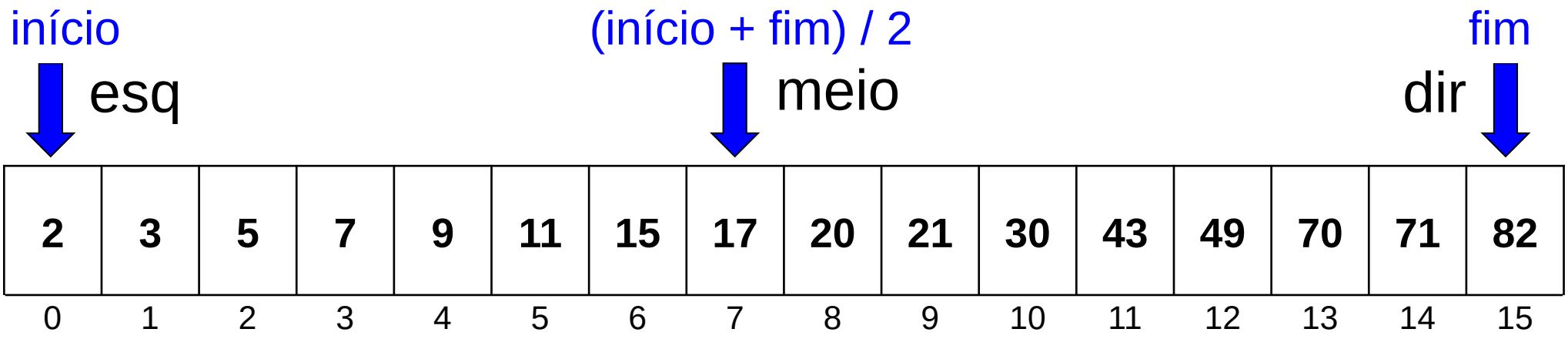
Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**



Calculo de Complexidade

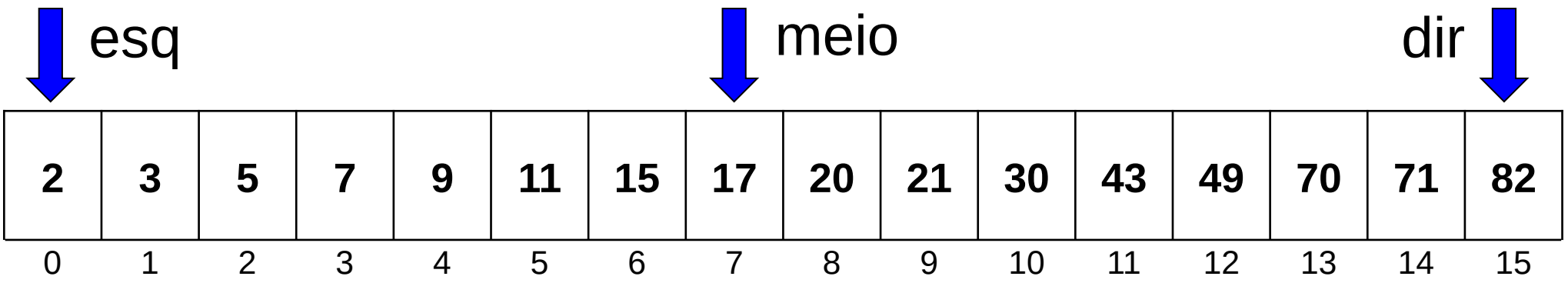
- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**



Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
        meio = (esq + dir) / 2;
        if (x == vet[meio]){
                resp = true;
                esq = n;
        } else if (x > vet[meio]){
                esq = meio + 1;
        } else {
                dir = meio - 1;
}       }
```

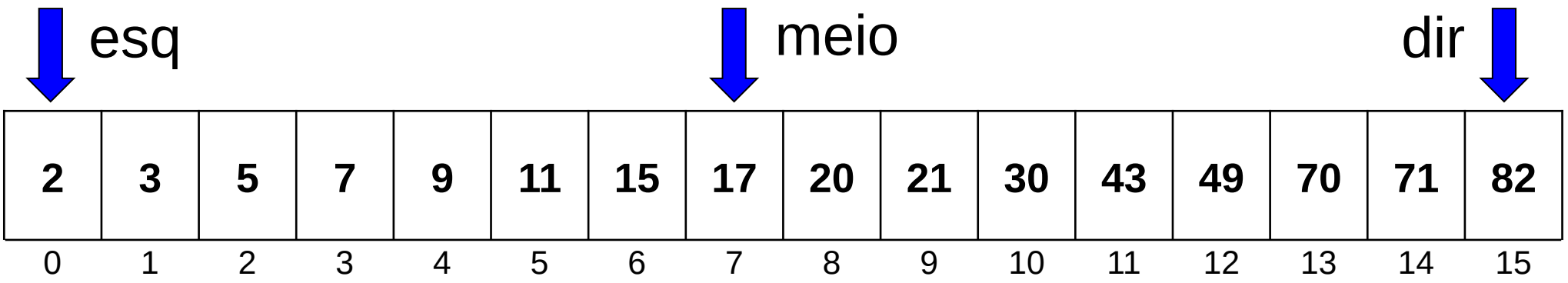


Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

resp | false

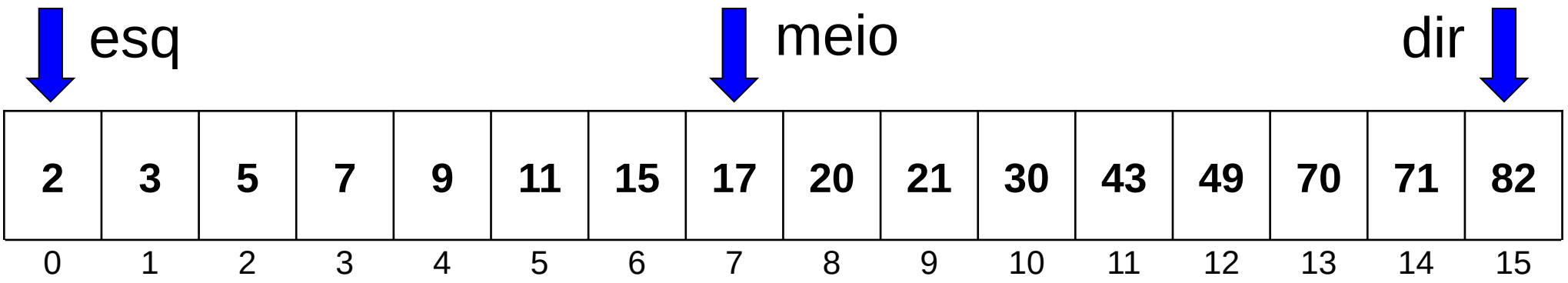


Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

resp | false



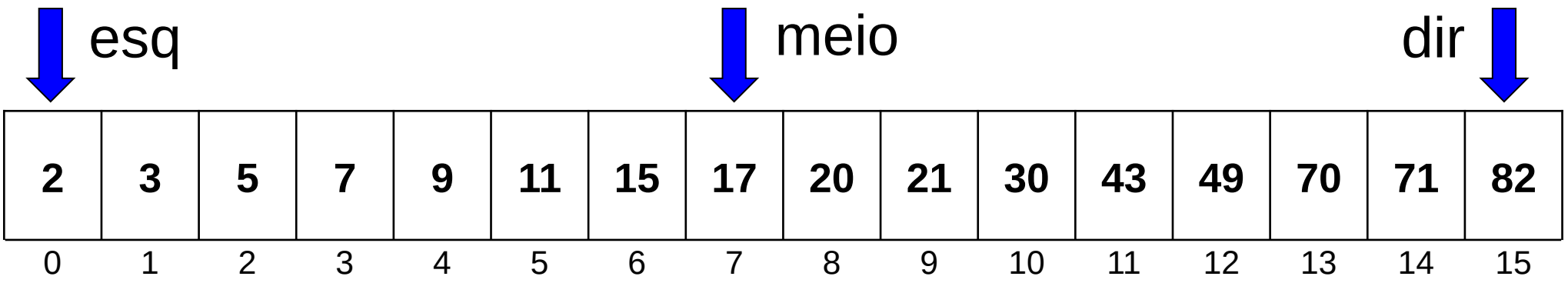
Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

(0 <= 15): true

resp	false



Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

(0 + 15) / 2: 7

resp	false

esq meio dir

2	3	5	7	9	11	15	17	20	21	30	43	49	70	71	82
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

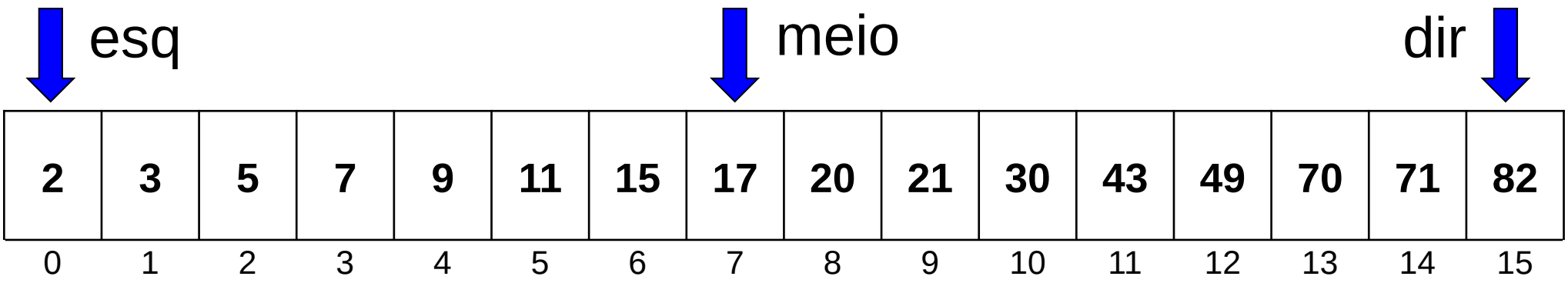
Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

(35 == 17): false

resp | false



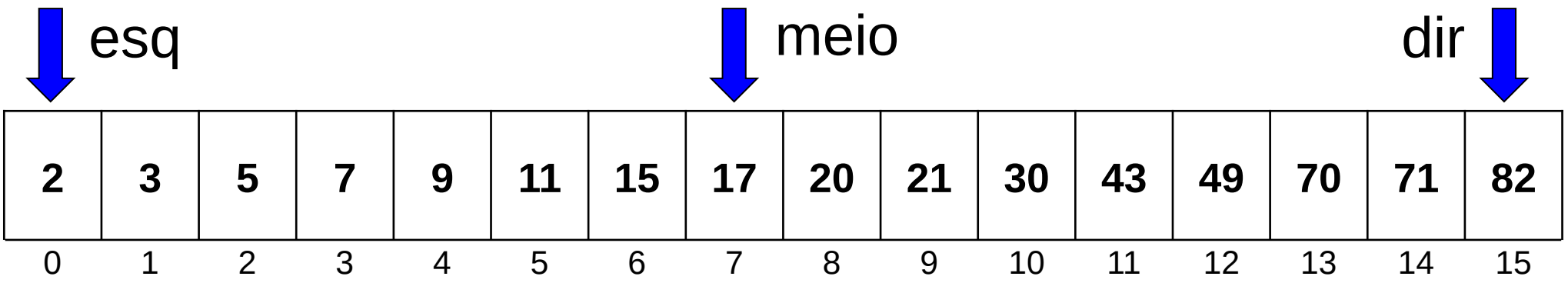
Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

(35 > 17): true

resp	false



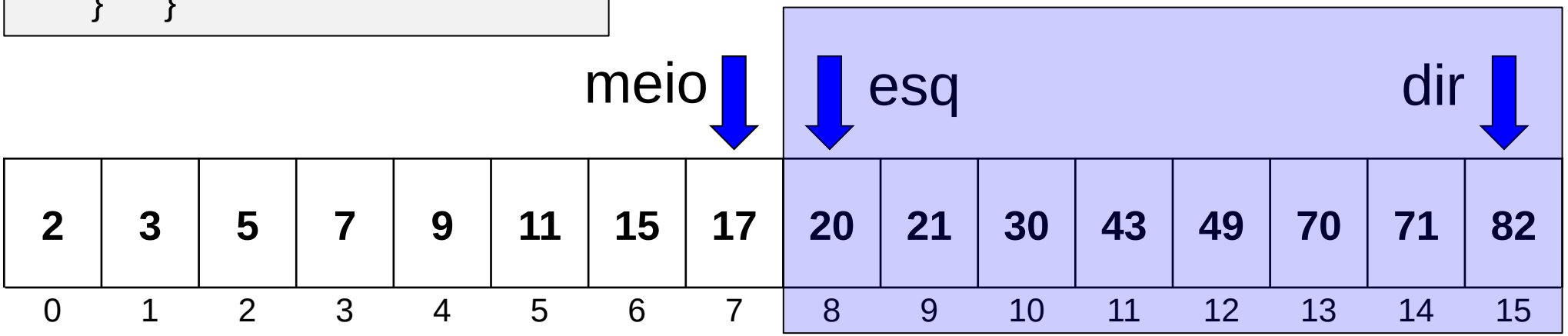
Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

resp | false

Com uma comparação, reduzimos o espaço de busca pela metade



Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

(8 <= 15): true

resp	false

esq ↓ dir ↓

2	3	5	7	9	11	15	17	20	21	30	43	49	70	71	82
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

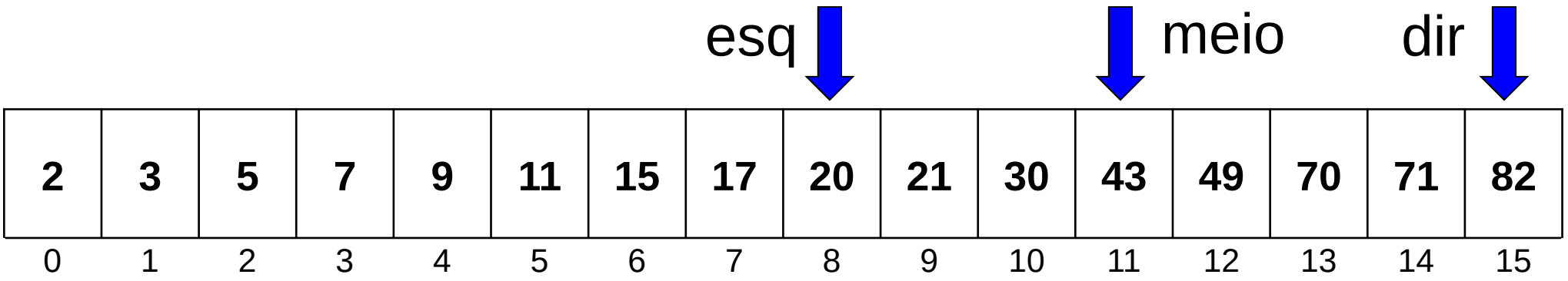
Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

(8 + 15) / 2: 11

resp	false



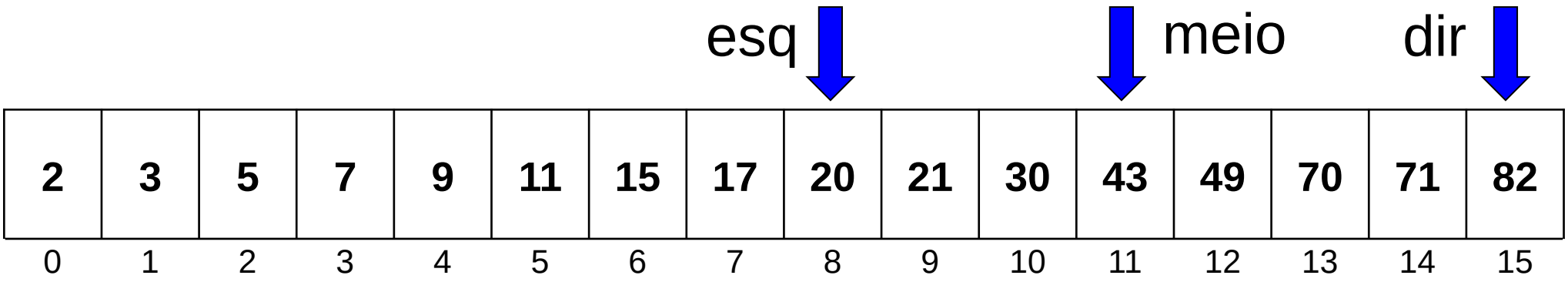
Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

(35 == 43): false

resp | false



Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

(35 > 43): false

resp	false

esq meio dir

2	3	5	7	9	11	15	17	20	21	30	43	49	70	71	82
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

resp	false

esq ↓ ↓ dir

2	3	5	7	9	11	15	17	20	21	30	43	49	70	71	82
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

(8 <= 10): true

resp	false

esq ↓ ↓ dir

2	3	5	7	9	11	15	17	20	21	30	43	49	70	71	82
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

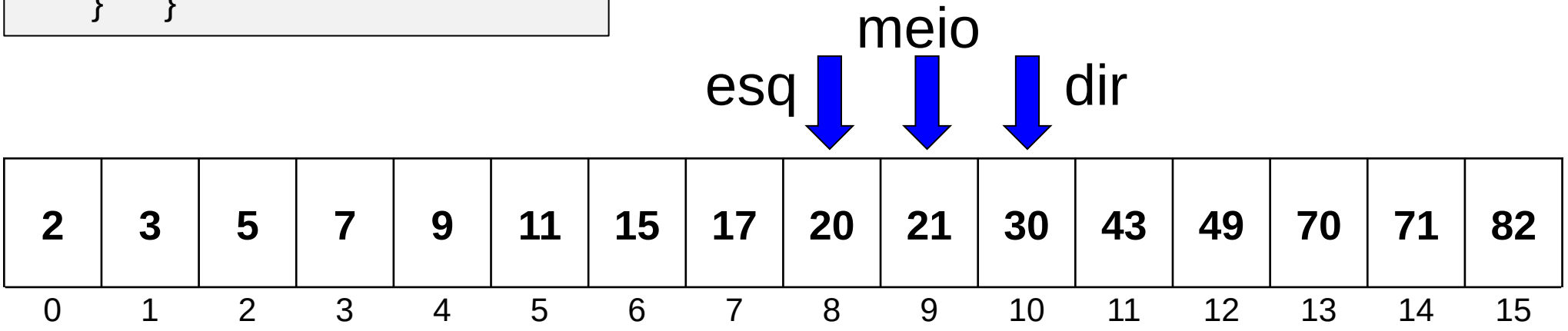
Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

(8 + 10) / 2: 9

resp	false



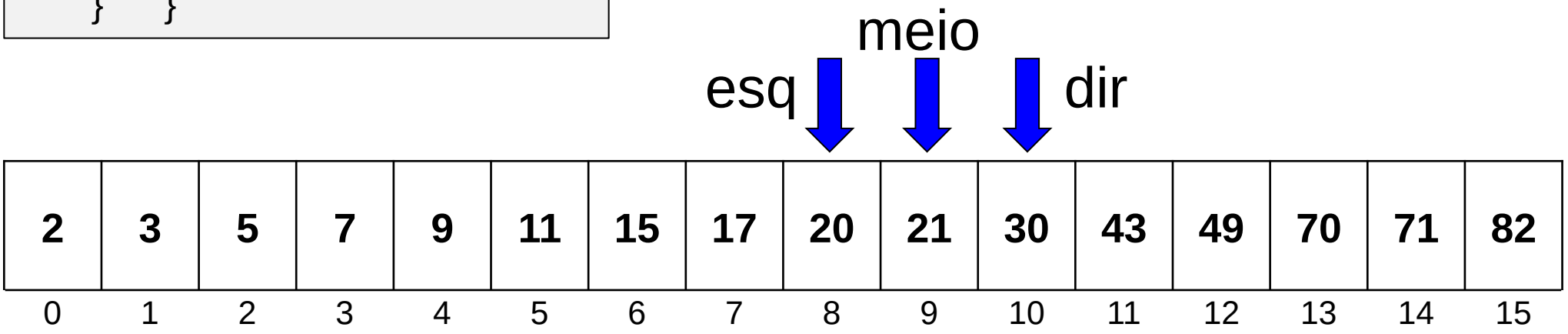
Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

(35 == 21): false

resp	false



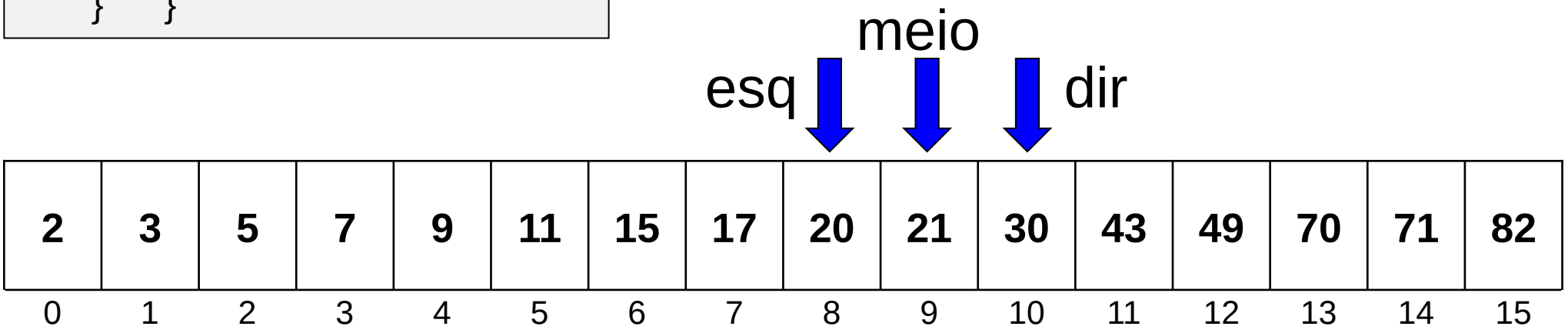
Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

(35 > 21): true

resp	false

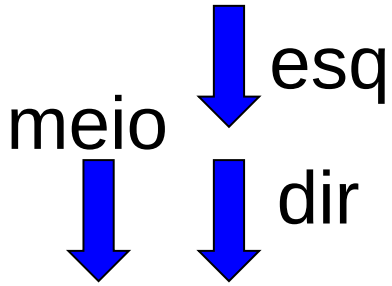


Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

resp	false



2	3	5	7	9	11	15	17	20	21	30	43	49	70	71	82
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

(10 <= 10): true

resp	false

esq

dir

2	3	5	7	9	11	15	17	20	21	30	43	49	70	71	82
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

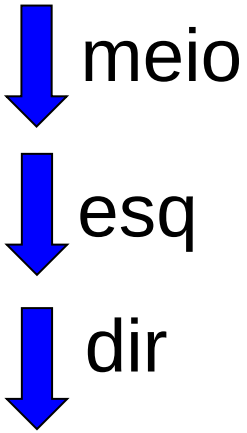
Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

(10 + 10) / 2: 10

resp	false



2	3	5	7	9	11	15	17	20	21	30	43	49	70	71	82
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

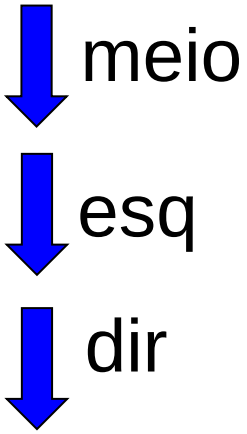
Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

(35 == 30): false

resp | false



2	3	5	7	9	11	15	17	20	21	30	43	49	70	71	82
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

(35 > 30): true

resp | false

meio

esq

dir

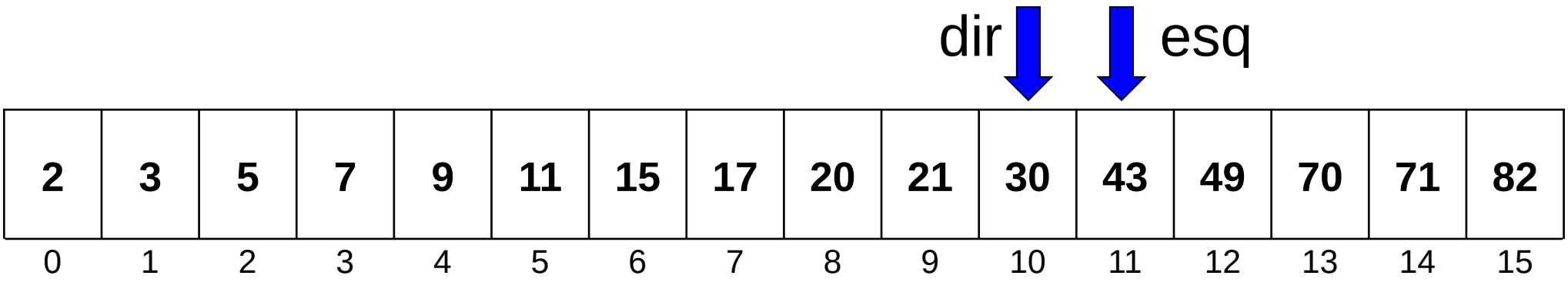
2	3	5	7	9	11	15	17	20	21	30	43	49	70	71	82
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

resp | false

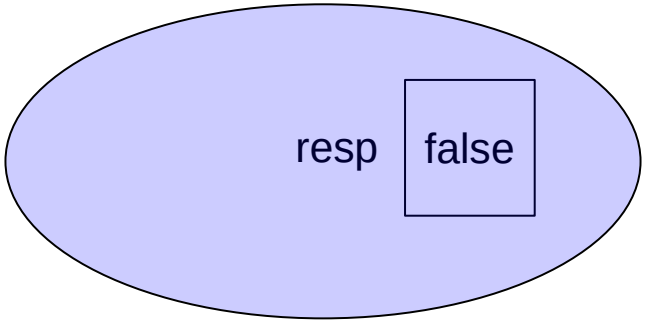


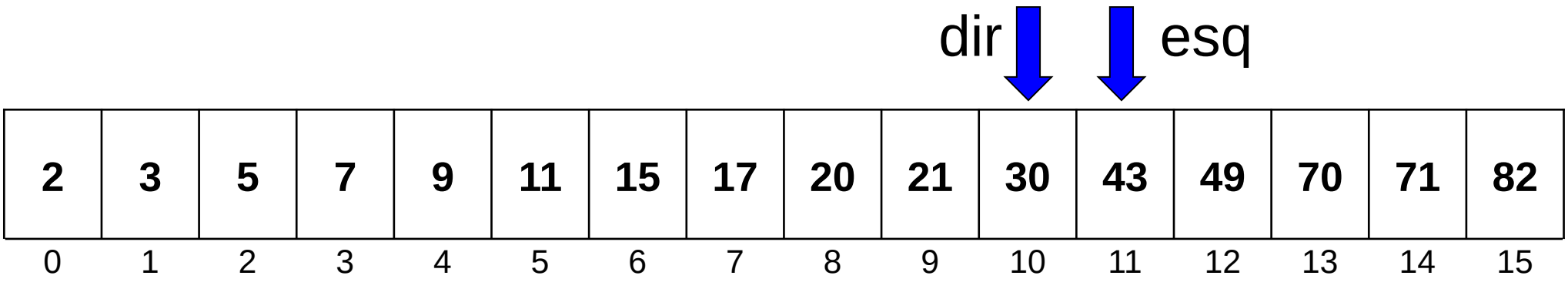
Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado **(ex.: procurar 35)**

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
    meio = (esq + dir) / 2;
    if (x == vet[meio]){
        resp = true;
        esq = n;
    } else if (x > vet[meio]){
        esq = meio + 1;
    } else {
        dir = meio - 1;
}   }
```

(11 <= 10): false





Calculo de Complexidade

- Pesquisa binária em um *array* de inteiros ordenado

```
boolean resp = false;
int dir = n - 1, esq = 0, meio;
while (esq <= dir) {
        meio = (esq + dir) / 2;
        if (x == vet[meio]){
                resp = true;
                esq = n;
        } else if (x > vet[meio]){
                esq = meio + 1;
        } else {
                dir = meio - 1;
}       }
```

1º) Qual é a operação relevante?

R: Comparação entre elementos do *array*.

2º) Quantas vezes ela será executada?

R:Melhor caso: f(n) = 1
Pior caso: f(n) = lg(n)
Caso médio: f(n) = (lg(n) + 1) / 2

Calculo de Complexidade

- Exercício: Encontrar o maior e o menor valores em um *array* de inteiros

Exercício

- Um aluno deve procurar um valor em um *array* de números reais. Ele tem duas alternativas. Primeiro, executar uma pesquisa sequencial. Segundo, ordenar o vetor e, em seguida, aplicar uma pesquisa binaria. O que fazer?

Exercício

- Um aluno deve procurar um valor em um *array* de números reais. Ele tem duas alternativas. Primeiro, executar uma pesquisa sequencial. Segundo, ordenar o vetor e, em seguida, aplicar uma pesquisa binaria. O que fazer?

O aluno deve escolher a primeira opção, pois a pesquisa sequencial tem custo O(n). A segunda opção tem custo O(n x lg n) para ordenar mais O (lg n) para a pesquisa binária

Regras Gerais da Notação *O*

- Consideramos apenas a maior potência
- Ignoramos os coeficientes
- Se um algoritmo é O(f(n)), ele também será O(g(n)) para toda função g(n) tal que “g(n) é maior que f(n)”

Exercício

- Responda as questões abaixo:

a) $3n^2 + 5n + 1$ é $O(n^2)$?

b) $3n^2 + 5n + 1$ é $O(n)$?

c) $3n^2 + 5n + 1$ é $O(n^3)$?

d) $\lg(n)$ é $O(n)$?

e) $n * \lg(n)$ é $O(n)$?

f) $n * \lg(n)$ é $O(n * \lg(n))$?

Operações com a Notação *O*

- f(n) = O(f(n))
- c * O(f(n)) = O(f(n))
- O(f(n)) + O(f(n)) = O(f(n))
- O(O(f(n))) = O(f(n))
- O(f(n)) + O(g(n)) = O(máximo(f(n),g(n)))
- O(f(n)) * O(g(n)) = O(f(n) * g(n))
- f(n) * O(g(n)) = O(f(n) * g(n))

Classe de Algoritmos

- Constante: O(1)
- Logarítmico: O(log n)
- Linear: O(n)
- Quadrático: $O(n^2)$
- Exponencial: $O(2^n)$
- Duplo exponencial: $O(2^{2^n})$

Algoritmos Polinomiais

- Um algoritmo é polinomial se é $O(n^p)$ para algum inteiro p
- Problemas com algoritmos polinomiais são considerados tratáveis
- Problemas para os quais não há algoritmos polinomiais são considerados intratáveis
- Classes de problemas e o problema $P \stackrel{?}{=} NP$

Exercício

- Faça um resumo sobre Teoria da Complexidade, Classes de Problemas P, NP e NP-Completo. Use LaTeX e siga o modelo de artigos da SBC (sem abstract, resumo e seções) com no máximo duas página

Definição da Notação *O*

- Uma função *f(n)* **domina assintoticamente** *g(n)* se existem duas constantes positivas *c* e *m* tais que, para $n \geq m$, temos $|g(n)| \leq c \times |f(n)|$

Definição da Notação *O*

- Uma função *f(n)* **domina assintoticamente** *g(n)* se existem duas constantes positivas *c* e *m* tais que, para $n \geq m$, temos $|g(n)| \leq c \times |f(n)|$

Ou seja, escolha duas constantes quaisquer ...

..., contudo, respeitando as regras acima

É claro o que significa quaisquer?

Definição da Notação *O*

- Uma função *f(n)* **domina assintoticamente** *g(n)* se existem duas constantes positivas *c* e *m* tais que, para $n \geq m$, temos $|g(n)| \leq c \times |f(n)|$

- Nesse caso, dizemos que:

$$g(n) = O(f(n))$$

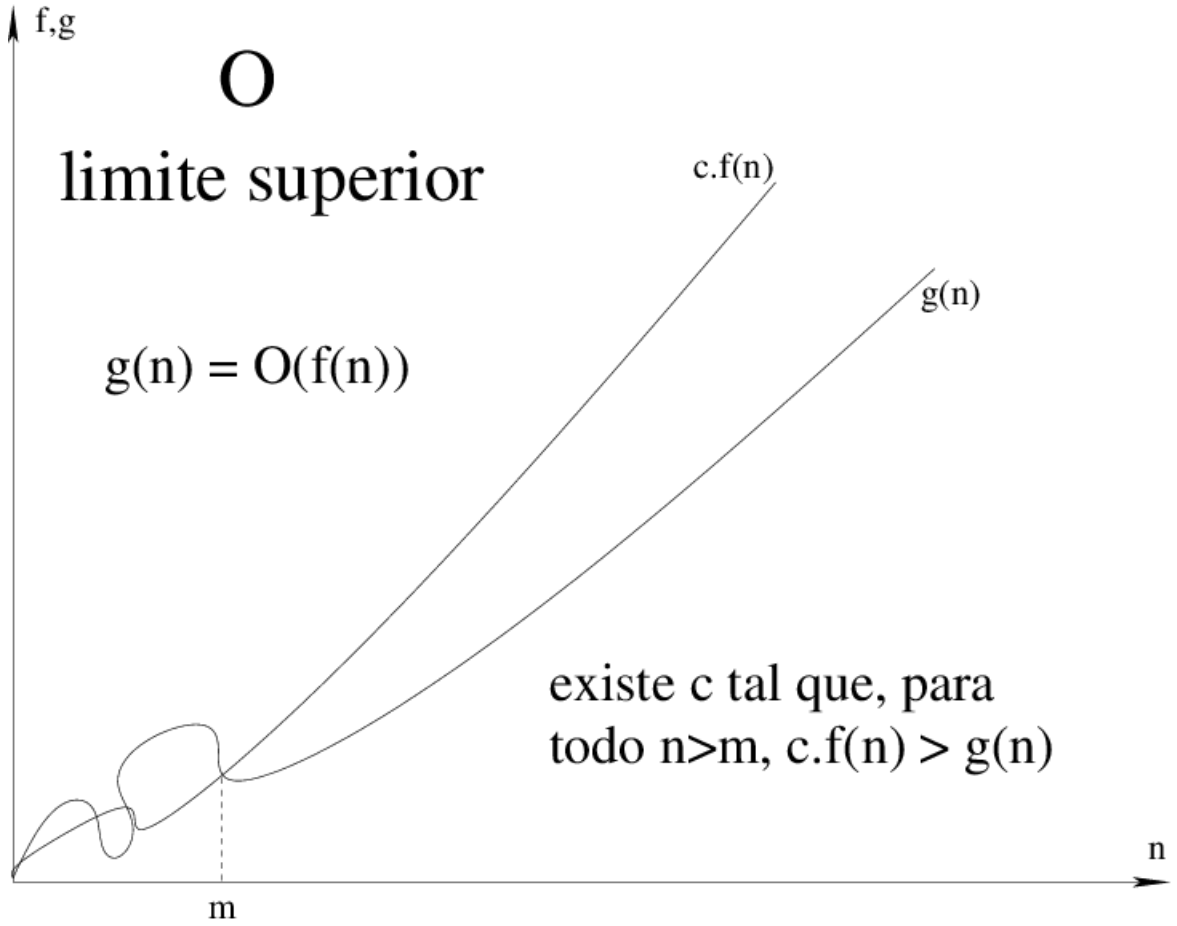
Definição da Notação *O*

- Uma função *f(n)* **domina assintoticamente** *g(n)* se existem duas constantes positivas *c* e *m* tais que, para $n \geq m$, temos $|g(n)| \leq c \times |f(n)|$

- Nesse caso, dizemos que:

$$g(n) = O(f(n))$$

- O comportamento assintótico das funções de custo representa o limite quando n cresce



Exercício

- $3n^2 + 5n + 1$ é $O(n^2)$?
  - Em outras palavras, conseguimos escolher os valores de c e m tal que, para $n \geq m$, temos $| 3n^2 + 5n + 1 | \leq c \times |n^2|$?
  - Para ajudar, pegue o arquivo unidade02_analiseAlgoritmos_grafico.xlsx

Exercício

- $3n^2 + 5n + 1$ é $O(n)$?
  - Em outras palavras, conseguimos escolher os valores de c e m tal que, para $n \geq m$, temos $| 3n^2 + 5n + 1 | \leq c \times |n|$?

Notação *O*

- **Exemplo**: Seja $g(n) = n$ e $f(n) = -n^2$, temos que $|n| \leq |-n^2|$ para todo $n \in N$
  - Fazendo c = 1 e m = 0, a definição é satisfeita
  - Logo, f(n) domina assintoticamente g(n)

$$g(n) = O(f(n))$$

Notação *O*

- **Exemplo**: Seja $g(n) = n$ e $f(n) = -n^2$, temos que $|n| \leq |-n^2|$ para todo $n \in N$
  - Fazendo c = 1 e m = 0, a definição é satisfeita
  - Logo, f(n) domina assintoticamente g(n)

$$g(n) = O(f(n))$$

Atenção! A igualdade não é simétrica.

Notação *O*

- **Exemplo**: Seja $g(n) = 3n^3+2n^2+n$ é $O(n^3)$, pois $3n^3+2n^2+n \leq 6n^3$, n ≥ 0
  - $g(n)$ é também $O(n^4)$, mas isto é mais fraco do que dizer que $g(n)$ é $O(n^3)$.

Exercício

- Apresente a função e a ordem de complexidade para os números de comparação e movimentações de registros para o pior e melhor caso

```
void imprimirMaxMin( int [] array, int n){
    int maximo, minimo;

    if (array[0] > array[1]){
        maximo = array[0];     minimo = array[1];
    } else {
        maximo = array[1];     minimo = array[0];
    }

    for (int i = 2; i < n; i++){
        if (vet[i] > maximo){          maximo = vet[i];
        } else if (vet[i] < minimo){   minimo = vet[i];
}   }   }
```

Exercício

- Apresente a função e a ordem de complexidade para os números de comparação e movimentações de registros para o pior e melhor caso

função de complexidade

	MOV	CMP
PIOR	2 + (n – 2)	1 + 2(n – 2)
MELHOR	2 + (n – 2) x 0	1 + (n – 2)

ordem de complexidade

	MOV	CMP
PIOR	O(n)	O(n)
MELHOR	O(1)	O(n)

Exercício

- Apresente a função e a ordem de complexidade para o número de subtrações para o pior e melhor caso

```
i = 0;

while (i < n) {
        i++;
        a--;
}

if (b > c) {
        i--;
} else {
        i--;
        a--;
}
```

Exercício

- Apresente a função e a ordem de complexidade para o número de subtrações para o pior e melhor caso

```
i = 0;

while (i < n) {
        i-
        a
}

if (b >
        i-
} else
        i--;
        a--;
}
```

função de complexidade / ordem de complexidade

	f(n)	O(f(n))
PIOR	n + 2	O(n)
MELHOR	n +1	O(n)

Exercício

- Apresente a função e a ordem de complexidade para o número de subtrações para o pior e melhor caso

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        a--;
        b--;
    }
    c--;
}
```

Exercício

- Apresente a função e a ordem de complexidade para o número de subtrações para o pior e melhor caso

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

função de complexidade / ordem de complexidade

	f(n)	O(f(n))
TODOS	(2n + 1)n	$O(n^2)$

Exercício

- Apresente a função e a ordem de complexidade para o número de subtrações para o pior e melhor caso

```
for (i = 0; i < n; i++) {
    for (j = 1; j <= n; j *= 2) {
        b--;
    }
}
```

Exercício

- Apresente a função e a ordem de complexidade para o número de subtrações para o pior e melhor caso

```
for (i = 0; i < n; i++) {
    for (j = 1; j <= n; j *= 2) {
```

função de complexidade / ordem de complexidade

	f(n)	O(f(n))
TODOS	(lg(n) + 1) * n = n * lg(n) + n	O(n * lg(n))

Exercício

- Suponha um sistema de monitoramento contendo os métodos telefone, luz, alarme, sensor e câmera, apresente a função e ordem de complexidade para o pior e melhor caso: (a) método alarme; (b) outros métodos.

```
void sistemaMonitoramento() {
    if (telefone() == true && luz() == true){
        alarme(0);
    } else {
        alarme(1);
    }
    for (int i = 2; i < n; i++){
        if (sensor(i- 2) == true){
            alarme (i - 2);
        } else if (camera(i- 2) == true){
            alarme (i - 2 + n);
} } }
```

Exercício

- Apresente um código método, defina duas operações relevantes e apresente a função e a ordem de complexidade para as operações escolhidas para o pior e melhor caso