


Algoritmos de Procura e de Ordenação II

01/10/2025

Sumário

- **Binary Search** – Procura binária num **array ordenado**
- Algoritmos de Ordenação
- **Selection Sort** – Ordenação por seleção
- Exercícios / Tarefas 
- Sugestão de leitura

Binary Search


– Array ordenado

Binary Search – Procura com sucesso

- O valor **2** pertence ao array ?

0	1	2	3	4	5
2	4	6	8	10	12

0	1
2	4



Encontrado !!

Binary Search – Procura com **insucesso**

- O valor **13** pertence ao array ?

0	1	2	3	4	5
2	4	6	8	10	12



3	4	5
8	10	12



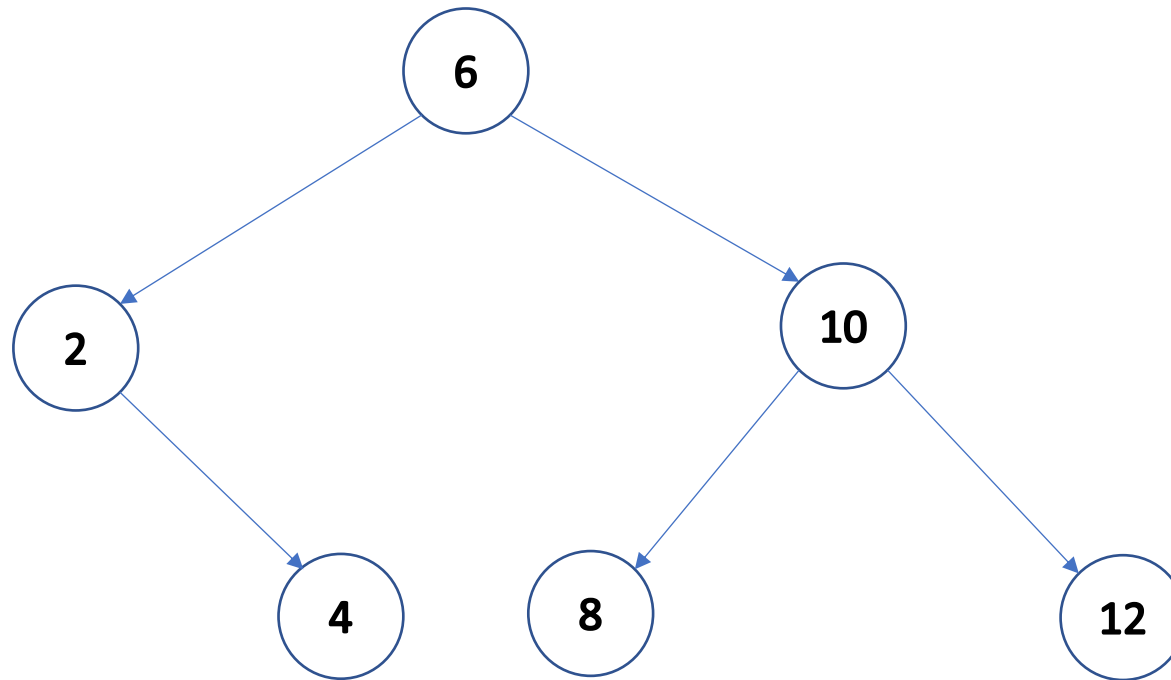
5
12



Não !!

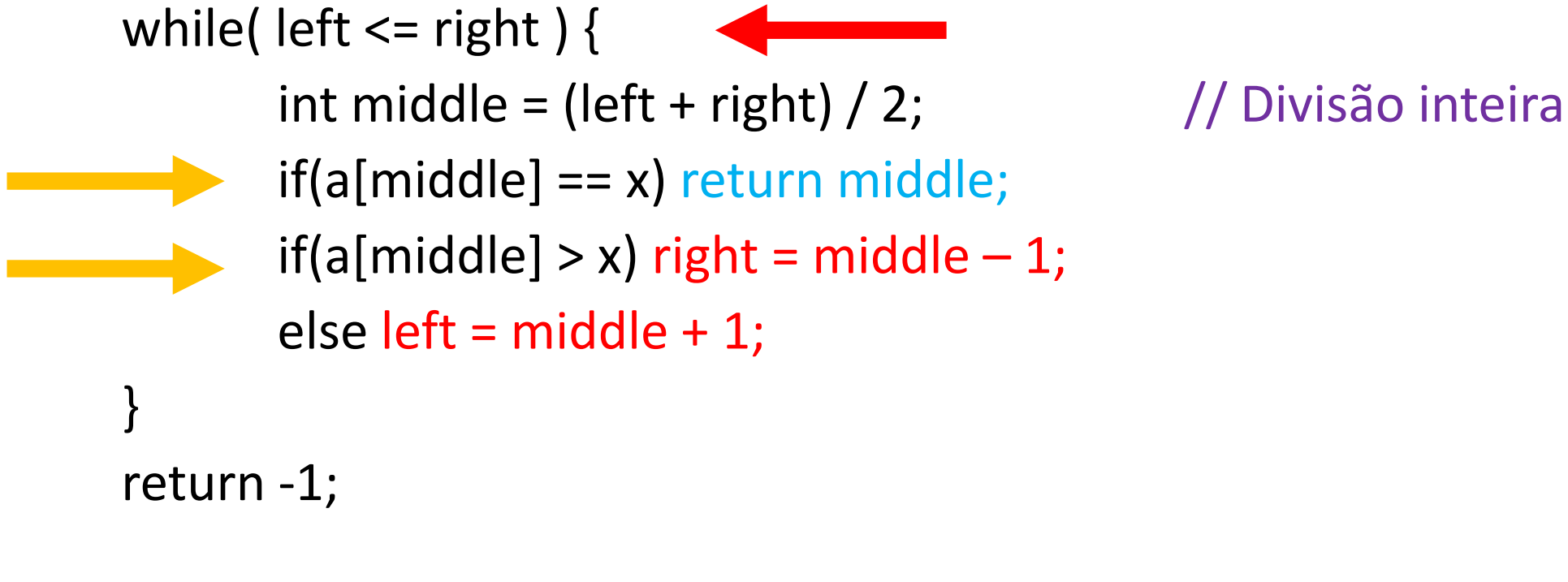
Representação do array como árvore binária

- A **representação gráfica como árvore binária** auxilia a compreensão do funcionamento do algoritmo



Procura binária – Nº de iterações do ciclo ?

```
int binSearch( int a[], int n, int x ) {  
    int left = 0; int right = n - 1;  
    while( left <= right ) {  
        int middle = (left + right) / 2;           // Divisão inteira  
        if(a[middle] == x) return middle;  
        if(a[middle] > x) right = middle - 1;  
        else left = middle + 1;  
    }  
    return -1;  
}
```



Binary Search

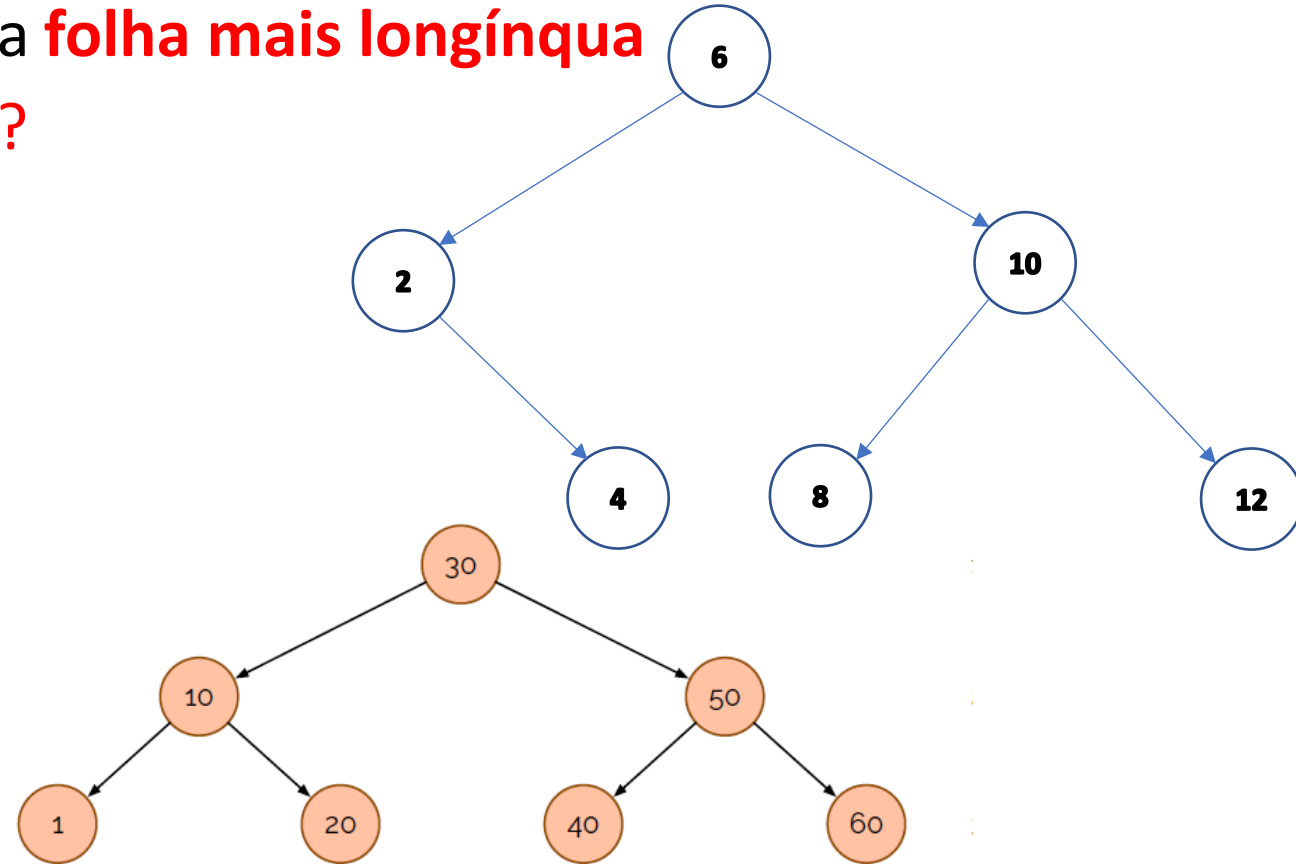
– Melhor Caso e Piores Casos

Procura binária – Melhor Caso

- O 1º elemento consultado é o valor procurado !!
- 1 comparação com o elemento na posição $a[\text{middle}]$
- 1 iteração do ciclo while
- $B(n) = 1$
- Muito pouco habitual...

Pior Caso – Procura com sucesso

- Percorrer a árvore até atingir a/uma **folha mais longínqua**
- Em geral, qual é a **forma** da árvore ?
- E qual é a sua **altura** ?
- Array com **1** elemento : ?
- Array com **2** elementos : ?
- Array com **3** elementos : ?
- Array com **4** elementos : ?
- ...
- **Array com n elementos : ?**



Pior Caso – Instâncias mais simples

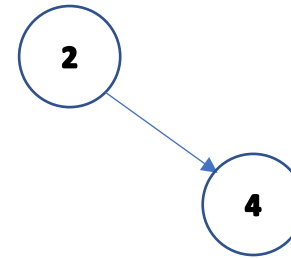
- $n = 1$ 1 iteração

- $n = 2$

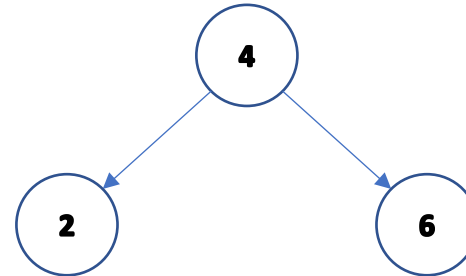
0	1
2	4

- $n = 3$

0	1	2
2	4	6



2 iterações

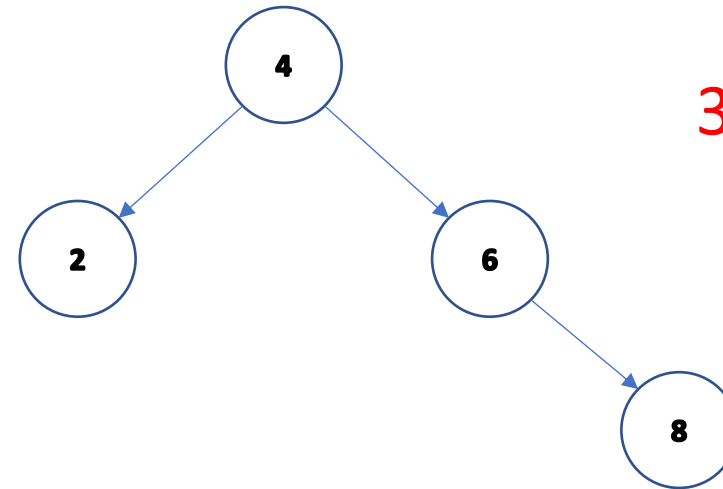


2 iterações

Pior Caso – Instâncias mais simples

- $n = 4$

0	1	2	3
2	4	6	8

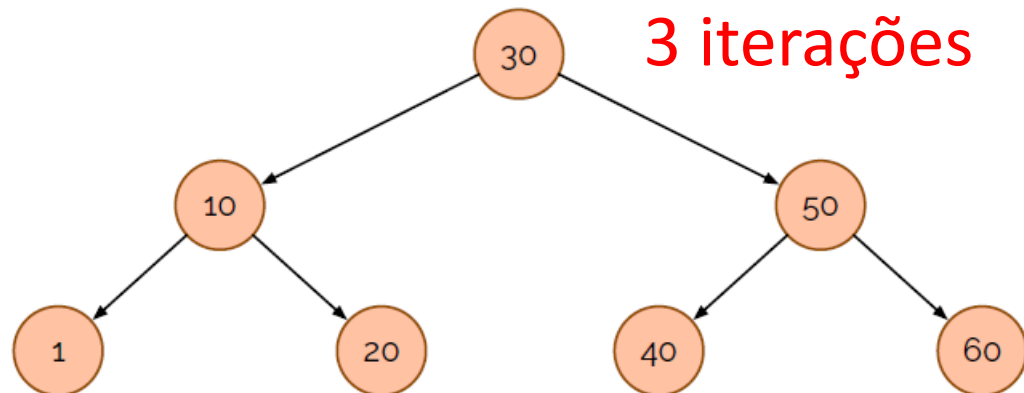


3 iterações

- ...

- $n = 7$

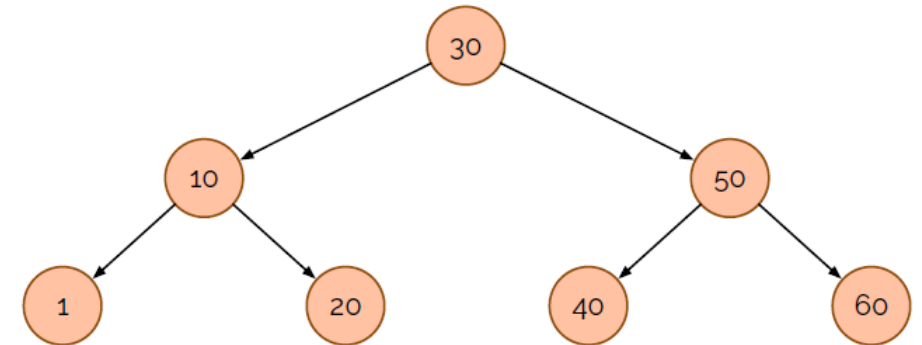
0	1	2	3	4	5	6
1	10	20	30	40	50	60



3 iterações

Pior Caso – Procura com sucesso

- Percorrer a árvore até atingir a/uma **folha mais longínqua**
- Array com 1 elemento : 1 iteração
- Array com **2 elementos : 2 iterações**
- Array com 3 elementos : 2 iterações
- Array com **4 elementos : 3 iterações**
- ...
- Array com 7 elementos : 3 iterações
- Array com **8 elementos : 4 iterações**



Array com n elementos : ?

Pior Caso – Procura com insucesso

- Há alguma diferença se o valor procurado não pertencer ao array ?
- Ou o nº de iterações é o mesmo, sempre que procurarmos um valor que não pertence ao array ?
- No pior caso, o nº de iterações é determinado pela altura da árvore binária associada

$$W(n) = 1 + \lfloor \log_2 n \rfloor = 1 + \text{floor}(\log n)$$

$$W(n) \in O(\log n)$$

Binary Search

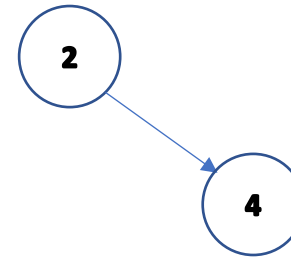
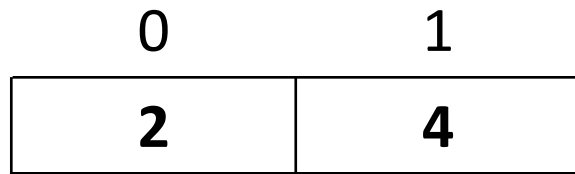
– Caso Médio : Cenário 1

Caso Médio – Procuras com sucesso

- Cenário experimental
 - Procurar uma vez cada um dos valores registados no array
 - Qual o nº médio de iterações realizadas ?
-
- Análise formal
 - Equiprobabilidade
 - Caso particular : array com $2^k - 1$ elementos

Caso Médio – Instâncias mais simples

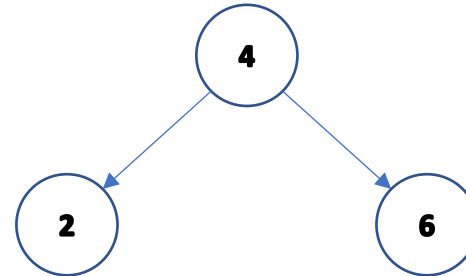
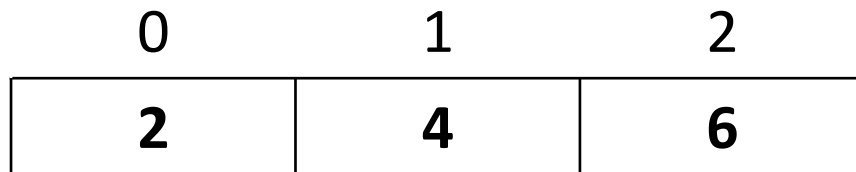
- $n = 2$



$$(1 + 2) / 2 = 1.5$$

1.5 iterações

- $n = 3$



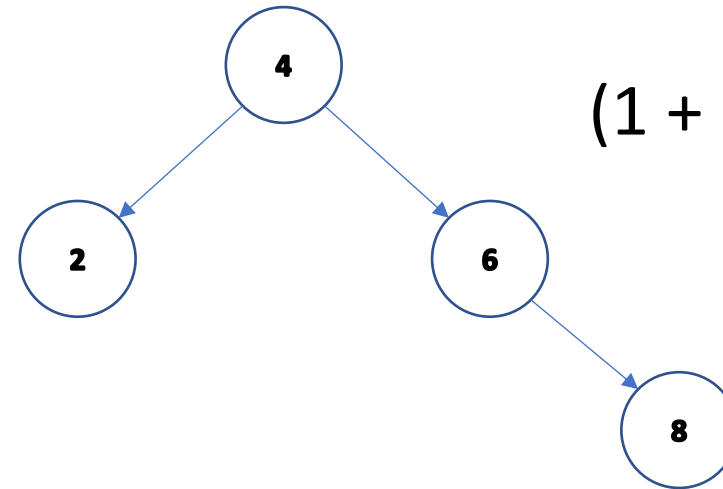
$$(1 + 2 + 2) / 3 = 1.67$$

1.67 iterações

Caso Médio – Instâncias mais simples

- $n = 4$

0	1	2	3
2	4	6	8



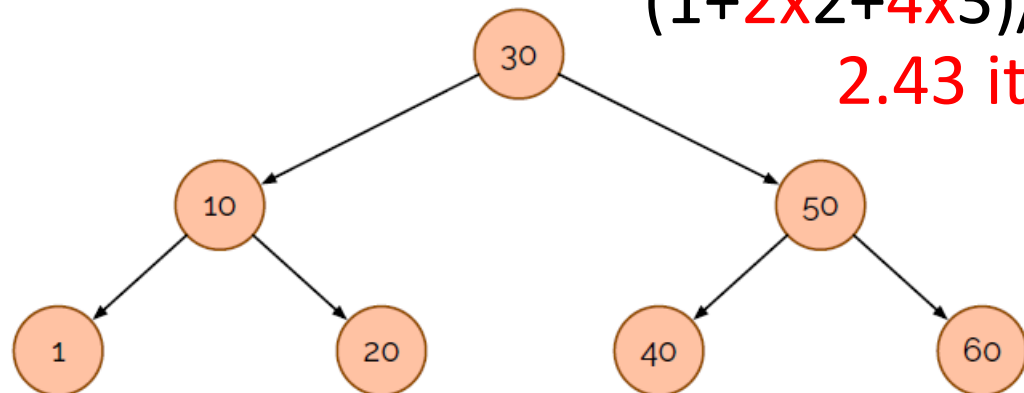
$$(1 + 2 + 2 + 3) / 4 = 2$$

2 iterações

- ...

- $n = 7$

0	1	2	3	4	5	6
1	10	20	30	40	50	60



$$(1 + 2 \times 2 + 4 \times 3) / 7 = 2.43$$

2.43 iterações

Caso Médio – Nº de nós em cada nível da árvore

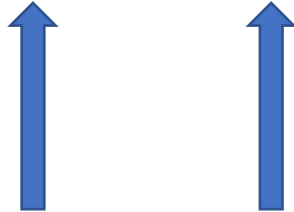
- Valor procurado pertence ao array – Equiprobabilidade
- Caso particular : $n = 2^k - 1$, $k = \log_2(n + 1)$
- Representação em árvore binária : árvore tem **k níveis**

Índice do nível	Nº de nós no nível	Nº de iterações
0	1	1
1	2	2
2	4	3
3	8	4
...

Caso Médio – Sucesso – N° de iterações

- Valor procurado pertence ao array – **Equiprobabilidade**
- Caso particular : $n = 2^k - 1$, $k = \log_2(n + 1)$
- **N° de níveis** da árvore binária = k

$$A(n) = \sum_{i=0}^{k-1} \frac{1}{n} \times 2^i \times (i + 1) = \frac{1}{n} \left[\sum_{i=0}^{k-1} i \times 2^i + \sum_{i=0}^{k-1} 2^i \right]$$



N° de nós no nível i N° de iterações para um nó do nível i

Caso Médio – Sucesso – N^o de iterações

- Expressões auxiliares

$$\sum_{i=0}^{k-1} 2^i = 2^k - 1$$

$$\sum_{i=0}^{k-1} i \times 2^i = 2^k(k - 2) + 2$$

$$A(n) = \frac{1}{n} [2^k(k - 1) + 1] = \frac{k \times 2^k - (2^k - 1)}{n} = \frac{k \times (n + 1)}{n} - 1$$

$$A(n) = k + \frac{k}{n} - 1 = k - \left(1 - \frac{k}{n}\right) = \log_2(n + 1) - \left(1 - \frac{\log_2(n + 1)}{n}\right)$$

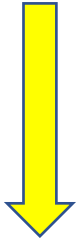
Verificação para as instâncias mais simples

- $n = 1$ 1 iteração
- $n = 3$ 1.67 iterações
- $n = 7$ 2.43 iterações

- $A(1) = \log 2 - 1 + (\log 2)/1 = 1$
- $A(3) = \log 4 - 1 + (\log 4)/3 = 1 + 2/3$
- $A(7) = \log 8 - 1 + (\log 8)/7 = 2 + 3/7$

OK!

Caso Médio **vs** Pior Caso – Nº de iterações

- Caso particular : $n = 2^k - 1$
 - $W(3) = 1 + \text{floor}(\log 3) = 2$
 - $A(3) = \log 4 - 1 + (\log 4)/3 = 1 + 2/3$ **$A(3) = W(3) - 1/3$**
 - $W(7) = 1 + \text{floor}(\log 7) = 3$
 - $A(7) = \log 8 - 1 + (\log 8)/7 = 2 + 3/7$ **$A(7) = W(7) - 4/7$**
 - $W(n) = 1 + \text{floor}(\log n) = \log(n + 1)$
 - $A(n) = \log(n + 1) - 1 + \log(n + 1)/n = W(n) - 1 + \log(n + 1)/n \approx \text{W(n)} - 1$
- 

Binary Search

– Caso Médio : Cenário 2

Caso Médio – $p = Prob(x \in a[0..(n - 1)])$

Casos possíveis		Nº de iterações	Probabilidade
Sucesso 0	É o 1º elemento	?	p / n
Sucesso 1	É o 2º elemento	?	p / n
...
Sucesso $(n - 1)$	É o último elemento	?	p / n
Insucesso 0	Menor do que o 1º	Nº de níveis da árvore	$(1 - p) / (n + 1)$
Insucesso 1	Entre o 1º e o 2º	Nº de níveis da árvore	$(1 - p) / (n + 1)$
...
Insucesso $(n - 1)$	Entre o penúltimo e o último	Nº de níveis da árvore	$(1 - p) / (n + 1)$
Insucesso n	Maior do que o último	Nº de níveis da árvore	$(1 - p) / (n + 1)$

Caso Médio – $p = \text{Prob}(x \in a[0..(n-1)])$

- Caso particular : $n = 2^k - 1$, $k = \log_2(n + 1)$
- Nº de níveis da árvore binária = k

$$A(n) = \left[\frac{p}{n} \times \sum_{i=0}^{k-1} 2^i \times (i + 1) + \frac{1-p}{n+1} \times (n+1) \times k \right]$$

$$A(n) = \frac{p}{n} \times [k \times (n+1) - n] + (1-p) \times k$$

$$A(n) = k - p \times \left(1 - \frac{k}{n}\right) \approx k - p \approx \log_2(n+1) - p$$


Caso Médio – $p = \text{Prob}(x \in a[0..(n-1)])$

$$A(n) = k - p \times \left(1 - \frac{k}{n}\right) \approx \log_2(n+1) - p$$

- Se $p = 1$, então $A(n) \approx \log_2(n+1) - 1$ – Sempre sucesso !
- Se $p = 50\%$, então $A(n) \approx \log_2(n+1) - \frac{1}{2}$
- Se $p = 25\%$, então $A(n) \approx \log_2(n+1) - \frac{1}{4}$
- Se $p = 0$, então $A(n) \approx \log_2(n+1)$ – Sempre insucesso!

Binary Search

– Resumo

Procura Binária – Resumo

- Estratégia “**diminuir-para-reinar**” – “**decrease-and-conquer**”
- Representar as possíveis situações com o auxílio de uma **árvore binária**
- **Pior caso**: percurso da raiz até à **folha mais longínqua**
- $W(n) = 1 + \lfloor \log_2 n \rfloor$ Nº de níveis da árvore binária
- Caso particular : $n = 2^k - 1$, $k = \log_2(n + 1)$
- Procuras sempre com sucesso : $A(n) \approx W(n) - 1$
- Sucesso com **probabilidade p** : $A(n) \approx W(n) - p$ **$O(\log n)$**
- Para um **n qualquer**, obtemos **expressões “semelhantes”**

P. Sequencial vs P. Binária – Comparações

Nº de elementos	Procura Sequencial		Procura Binária	
	A(n)	W(n)	A(n)	W(n)
$2^9 - 1 = 511$	256	512	17	18
$2^{10} - 1 = 1023$	512	1024	19	20
$2^{14} - 1 = 16383$	8192	16384	27	28
$2^{17} - 1 = 131071$	65536	131072	33	34
$2^{20} - 1 = 1048575$	524288	1048576	39	40
	$O(n)$	$O(n)$	$O(\log n)$	$O(\log n)$

- **Cenário** : valores procurados podem não pertencer ao array
- **P. Binária** : nº de comparações $\approx 2 \times$ nº de iterações
- Caso médio : valores aproximados



Algoritmos de Ordenação

Algoritmos de Ordenação

- Para **quê ordenar** os elementos de um conjunto/multi-conjunto ?
- Para **facilitar operações** de procura, reunião, etc.
- Para **simplificar**, vamos considerar **arrays de números inteiros**
- **Em geral**, cada elemento do array é um **registo com uma chave de ordenação**
- **Função auxiliar** para a **comparação de duas chaves**: -1, 0, 1

Critérios de ordem habituais

- Ordem crescente : $a[i-1] < a[i]$

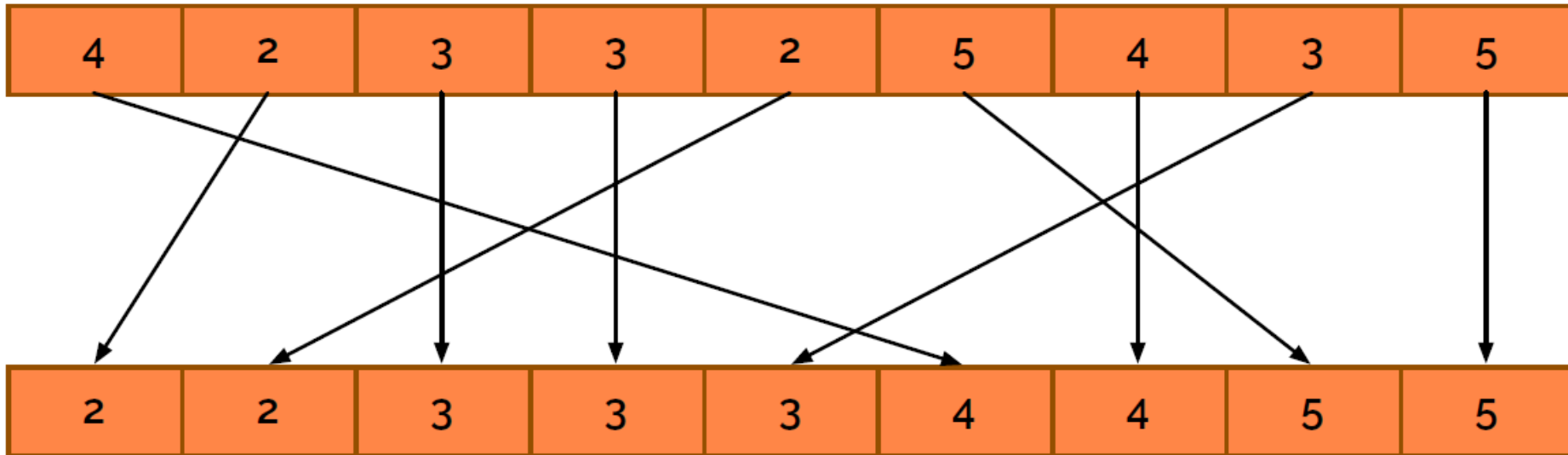
0	1	2	3	4	5
2	4	6	8	10	12

- Ordem não-decrescente : $a[i-1] \leq a[i]$

0	1	2	3	4	5
2	4	4	8	10	10

Algoritmo de ordenação **estável**

- Preservar a **ordem relativa inicial** de elementos com o **mesmo valor**



Arrays auxiliares de indexação

- Evitar a troca de elementos/registos que ocupem muitos bytes

0	1	2	3	4	5
99999	33333	22222	88888	55555	11111

- Trocar apenas os correspondentes **índices num array auxiliar**

0	1	2	3	4	5
0	1	2	3	4	5

5	2	1	4	3	0
---	---	---	---	---	---

Selection Sort

– Ordenação por Seleção

Selection Sort – Estratégia

- Procurar a **última ocorrência do maior** elemento
 - Quantas **comparações** ?
- Colocá-lo na última posição, se necessário, efetuando uma **troca**
- **Repetir** o processo para os restantes elementos
 - Quantas **comparações** ?
- Algoritmo **in-place**
- **Variante:** procurar a primeira ocorrência do menor elemento

Exemplo

0	1	2	3	4
7	2	6	4	3

Exemplo – Encontrar o maior elemento

0	1	2	3	4
7	2	6	4	3
7	2	6	4	3

- 4 comparações

Exemplo – Trocar para a posição final

0	1	2	3	4
7	2	6	4	3
7	2	6	4	3
3	2	6	4	7

- 4 comparações
- 1 troca

Exemplo – Encontrar o maior dos restantes

0	1	2	3	4
7	2	6	4	3

7	2	6	4	3
3	2	6	4	7

- 4 + 3 comparações
- 1 troca

Exemplo – Trocar para a posição final

0	1	2	3	4
7	2	6	4	3

7	2	6	4	3
3	2	6	4	7
3	2	4	6	7

- 4 + 3 comparações
- 1 + 1 trocas

Exemplo – Encontrar o maior dos restantes

0	1	2	3	4
7	2	6	4	3

7	2	6	4	3
3	2	6	4	7
3	2	4	6	7

- 4 + 3 + 2 comparações
- 1 + 1 trocas

Exemplo – Não é necessário trocar !!

0	1	2	3	4
7	2	6	4	3

7	2	6	4	3
3	2	6	4	7
3	2	4	6	7
3	2	4	6	7

- 4 + 3 + 2 comparações
- 1 + 1 + 0 trocas

Exemplo – Encontrar o maior dos restantes

0	1	2	3	4
7	2	6	4	3

7	2	6	4	3
3	2	6	4	7
3	2	4	6	7
3	2	4	6	7

- $4 + 3 + 2 + 1$ comparações
- $1 + 1 + 0$ trocas

Exemplo – Trocar para a posição final



0	1	2	3	4
7	2	6	4	3

7	2	6	4	3
3	2	6	4	7
3	2	4	6	7
3	2	4	6	7
2	3	4	6	7

• 1 + 1 + 0 + 1 trocas

Terminado !!

Selection Sort

```
void selectionSort( int a[], int n ) {  
    for( int k = n - 1; k > 0; k-- ) {  
        int indMax = 0;  
        for( int i = 1; i <= k; i++ ) {  
             if( a[i] >= a[indMax] ) indMax = i;  
        }  
        if( indMax != k ) swap( &a[indMax], &a[k] );   
    }  
}
```

Selection Sort – Nº de Comparações

- Número **fixo** de comparações ! --- Algoritmo “pouco inteligente”
- Mesmo que o array já esteja ordenado, continuamos a comparar !!

$$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

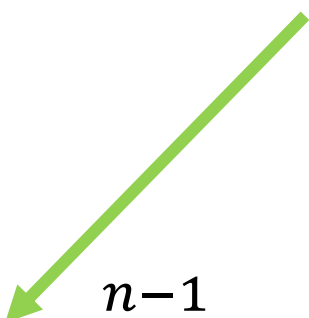
$O(n^2)$

Nº de Trocas – Melhor Caso e Pior Caso

- Melhor Caso ?
- **$Bt(n) = 0$** - Quando ?
- Pior Caso ?
- **$Wt(n) = n - 1$** - Quando ? **$O(n)$**
- Um **array pela ordem inversa** é uma configuração de pior caso ?

Selection Sort – Nº de Trocas – Caso Médio

- $p(I_j)$ é a probabilidade de o elemento $a[j]$ estar na posição correta
- Simplificação : Equiprobabilidade : $p(I_j) = 1 / (j + 1)$
- $(1 - p(I_j))$ é a probabilidade de ser necessária uma troca para o elemento $a[j]$ ficar na posição correta
- $(n - 1)$ iterações

$$A_t(n) = \sum_{j=1}^{n-1} (1 - p(I_j)) \times 1 = \sum_{j=1}^{n-1} 1 - \sum_{j=1}^{n-1} p(I_j)$$


Selection Sort – Nº de Trocas – Caso Médio

$$A_t(n) = \sum_{j=1}^{n-1} (1 - p(I_j)) \times 1 = \sum_{j=1}^{n-1} 1 - \sum_{j=1}^{n-1} p(I_j)$$

$$A_t(n) = n - 1 - \sum_{j=1}^{n-1} \frac{1}{j+1} = n - 1 - \left\{ \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \right\} = n - H_n$$

$$A_t(n) = n - H_n \approx n - \ln n \quad O(n)$$



Exercícios / Tarefas

Exercício 1 – Escolha múltipla

Considere o *array* ordenado de 8 elementos. Usando o algoritmo de **procura binária**:

0	1	2	3	4	5	6	7
1	3	5	7	9	11	13	15

- a) O elemento de valor 3 é encontrado ao fim de 2 tentativas.
- b) O elemento de valor 13 é encontrado ao fim de 3 tentativas.
- c) Ambas estão corretas.
- d) Nenhuma está correta.

Exercício 2 – Escolha múltipla

Considere o *array* ordenado de 15 elementos. Usando o algoritmo de **procura binária**:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	5	7	9	11	13	15	17	19	21	23	25	27	29

- a) O elemento de valor 11 é encontrado ao fim de 3 tentativas.
- b) O elemento de valor 1 é encontrado ao fim de 4 tentativas.
- c) Ao fim 4 tentativas conclui-se que o valor 28 não pertence ao *array*.
- d) Todas estão corretas.

Exercício 3 – Escolha múltipla

Considere o seguinte *array* de 6 elementos que é ordenado, por ordem crescente, usando o algoritmo “*Selectionsort*”.

0	1	2	3	4	5
6	5	4	3	2	1

- a) São efetuadas **15 comparações** entre elementos do *array*.
- b) São efetuadas **3 trocas** entre elementos do *array*.
- c) Ambas estão corretas.
- d) Nenhuma está correta.

Tarefa – Ordenação por Seleção

- Mantendo o objetivo de ordenar os elementos de um array por **ordem não-decrescente**, é possível desenvolver **variações do algoritmo Selection Sort** :
- **Versão 1** : Em cada iteração identifica-se, para o conjunto dos **elementos ainda não ordenados**, a **primeira ocorrência do seu menor elemento** e, se necessário, coloca-se na sua **posição final**
- **Versão 2 (mais complexa)** : Em cada iteração identifica-se, para o conjunto dos **elementos ainda não ordenados**, a **primeira ocorrência do seu menor elemento** e a **última ocorrência do seu maior elemento**, e, se necessário, colocam-se nas suas **posições finais**
- **Desenvolva e teste estes algoritmos**

Sugestão de leitura

Sugestão de leitura

- J. J. McConnell, Analysis of Algorithms, 1st Edition, 2001
 - Capítulo 2: **secção 2.2**