

Análise da Complexidade IV

Joaquim Madeira

23/03/2021

Sumário

- Recap
- Binary Search – Procura binária
- Algoritmos de Ordenação
- Selection Sort – Ordenação por seleção
- Sugestão de leitura

Let's
RECAP

Recapitulação

Procura sequencial – Array não ordenado

```
int search( int a[], int n, int x ) {  
    for( int i=0; i<n; i++ ) {  
        if( a[i] == x ) {  
            return i;  
        }  
    }  
    return -1;  
}
```

Comparações:

$B(n) = 1$

$W(n) = n$

$A(n) = ?$



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

Casos possíveis		Nº de comparações	Probabilidade
l_0	É o 1º elemento	1	p/n
l_1	É o 2º elemento	2	p/n
l_2	É o 3º elemento	3	p/n
...
l_{n-1}	É o último elemento	n	p/n
l_n	Não encontrado !	n	$(1 - p)$

$$A(n, p) = \sum_{i=0}^{n-1} \frac{p}{n} \times (i + 1) + (1 - p) \times n = \frac{p \times (n + 1)}{2} + (1 - p) \times n$$

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

$$A(n, p) = \frac{p \times (n + 1)}{2} + (1 - p) \times n$$

- Se $p = 1$, então $A(n) = (n + 1) / 2 \approx n / 2$
- Se $p = 50\%$, então $A(n) = (n + 1) / 4 + n / 2 \approx 3 \times n / 4$
- Se $p = 25\%$, então $A(n) = (n + 1) / 8 + 3 \times n / 4 \approx 7 \times n / 8$

Procura sequencial – Array ordenado

```
int search( int a[], int n, int x ) {  
    int stop = 0; int i;  
    for( i=0; i<n; i++ ) {  
        if( x <= a[i] ) {  
            stop = 1; break;  
        }  
    }  
    if( stop && x == a[i] ) return i;  
    return -1;  
}
```

Comparações:

$$B(n) = 2$$

$$W(n) = n + 1$$

$$A(n) = ?$$

Ordem da conjunção !!

Caso Médio

Casos possíveis		Nº de comparações	Probabilidade
Sucesso 0	É o 1º elemento	2	$1 / (2n + 1)$
Sucesso 1	É o 2º elemento	3	$1 / (2n + 1)$
...
Sucesso (n - 1)	É o último elemento	n + 1	$1 / (2n + 1)$
Insucesso 0	Menor do que o 1º	2	$1 / (2n + 1)$
Insucesso 1	Entre o 1º e o 2º	3	$1 / (2n + 1)$
...
Insucesso (n - 1)	Entre o penúltimo e o último	n + 1	$1 / (2n + 1)$
Insucesso n	Maior do que o último	n	$1 / (2n + 1)$

- $A(n) \approx \frac{n}{2}$ --- Melhor do que caso idêntico com array não ordenado !!

Binary Search

- Array ordenado

Procura binária

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

- O valor **2** pertence ao array ?
- Qual é o **primeiro elemento** consultado ?

Sim pertence

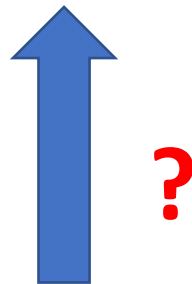
$\text{middle} = (\text{left} + \text{right}) / 2$
depois fazer floor

Procura binária

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

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

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



Procura binária

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

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

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

0	1
2	4

Qual é o **próximo elemento** consultado?

Procura binária

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

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

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

0	1
2	4

Encontrado !!



Procura binária

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

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

Procura binária

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

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



Procura binária

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

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

3	4	5
8	10	12

Procura binária

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

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

3	4	5
8	10	12



Procura binária

- 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



Procura binária

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

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

3	4	5
8	10	12

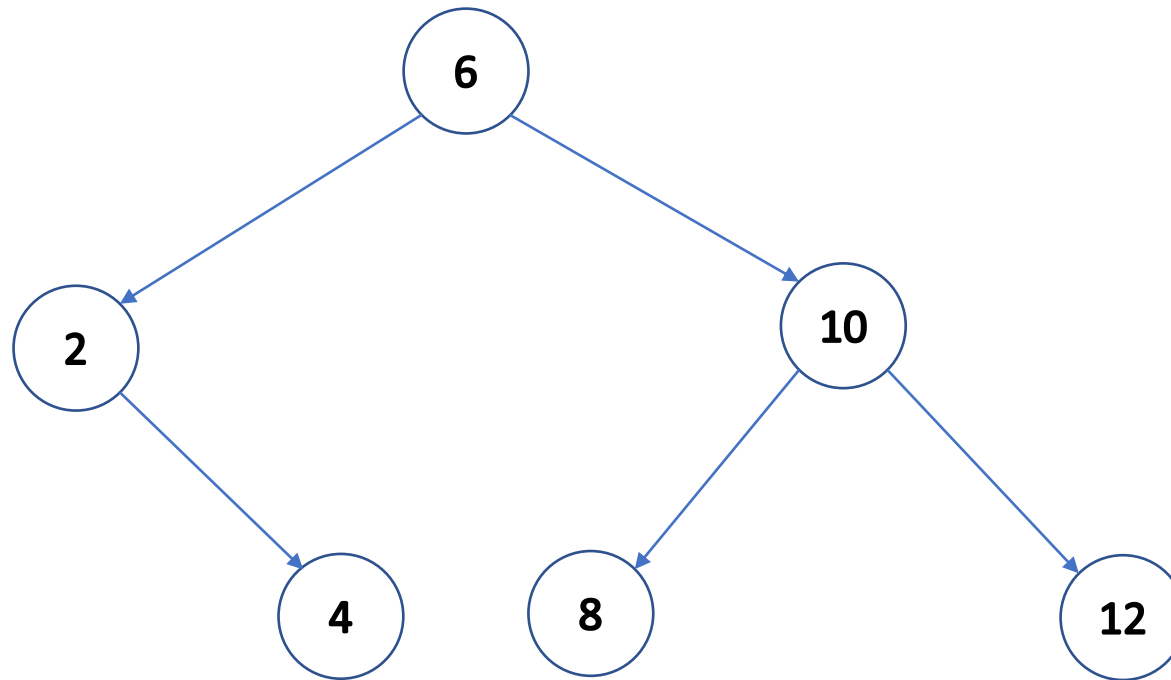
5
12

Não encontrado !!




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;  
}
```

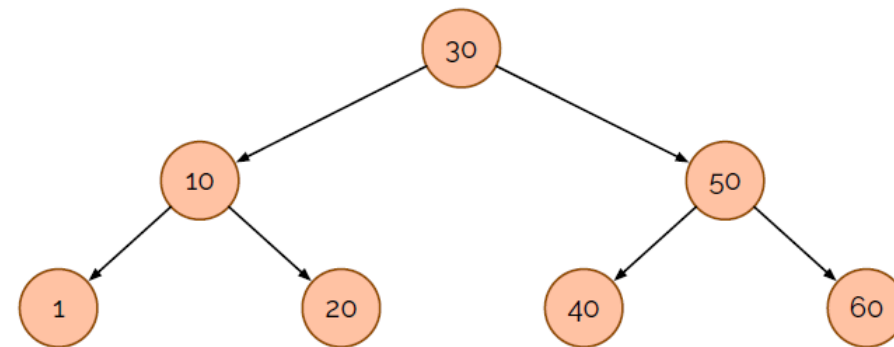
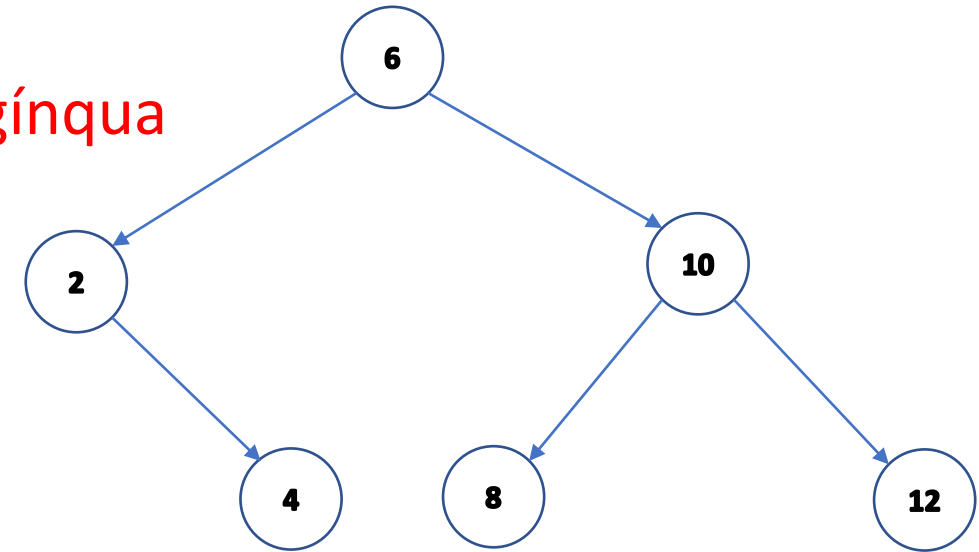


Melhor Caso

- O 1º elemento consultado é o valor procurado !!
- 1 comparação
- 1 iteração do ciclo while
- $B(n) = 1$
- Muito pouco habitual...

Pior Caso – Valor procurado pertence ao array

- Percorrer a árvore até atingir a **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 : ?**

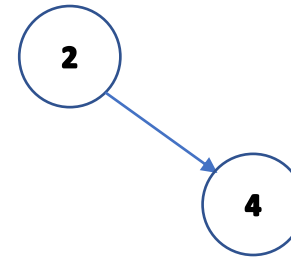


Pior Caso – Situações mais simples

- $n = 1$ 1 iteração

- $n = 2$

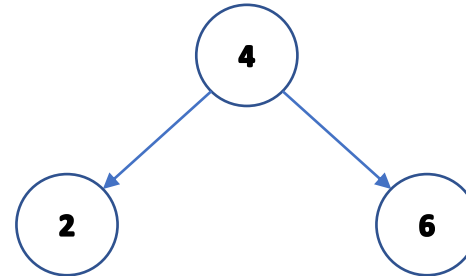
0	1
2	4



2 iterações

- $n = 3$

0	1	2
2	4	6

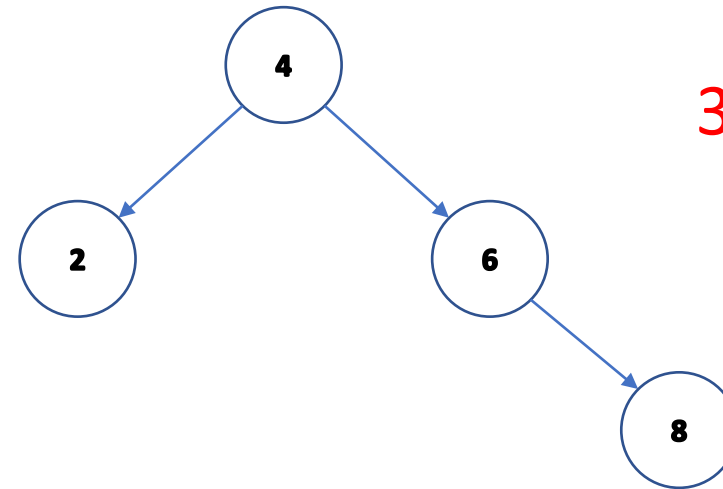


2 iterações

Pior Caso – Situações mais simples

- $n = 4$

0	1	2	3
2	4	6	8

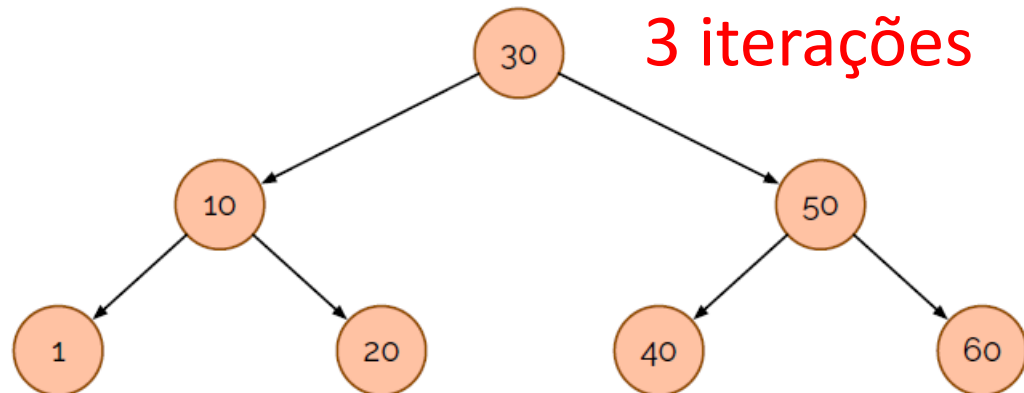


3 iterações

- ...

- $n = 7$

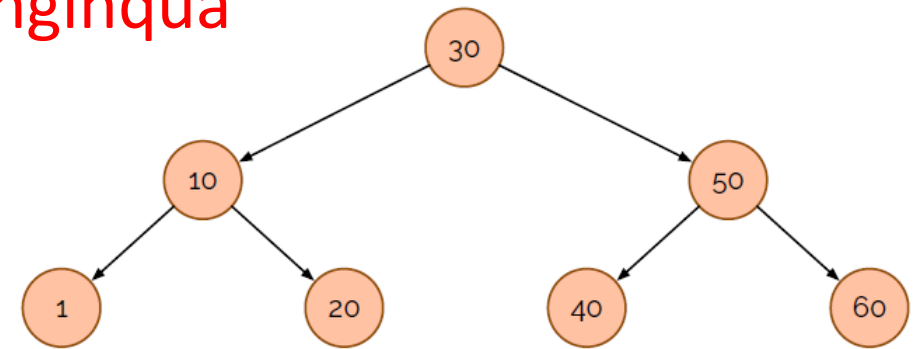
0	1	2	3	4	5	6
1	10	20	30	40	50	60



3 iterações

Pior Caso – Valor procurado pertence ao array

- Percorrer a árvore até atingir a **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

- 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

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

$$\mathbf{W(n) \in O(\log n)}$$

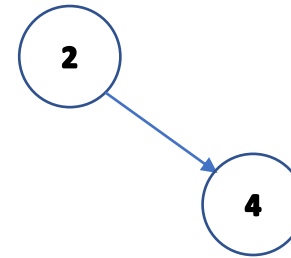
Caso Médio – Valor procurado pertence ao array

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

Caso Médio – Situações mais simples

- $n = 2$

0	1
2	4

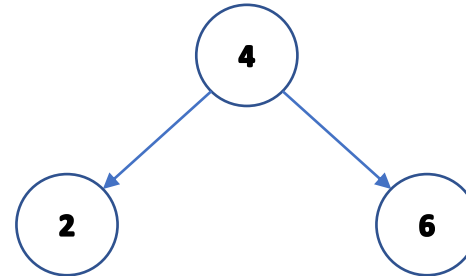


$$(1 + 2) / 2 = 1.5$$

1.5 iterações

- $n = 3$

0	1	2
2	4	6



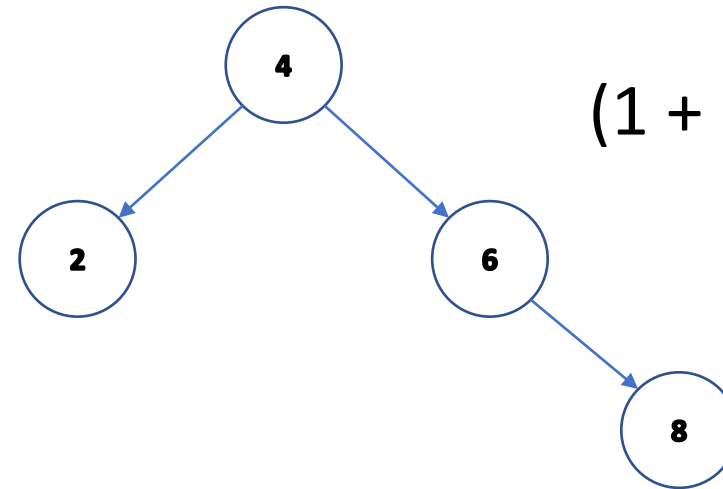
$$(1 + 2 + 2) / 3 = 1.67$$

1.67 iterações

Caso Médio – Situações mais simples

- $n = 4$

0	1	2	3
2	4	6	8



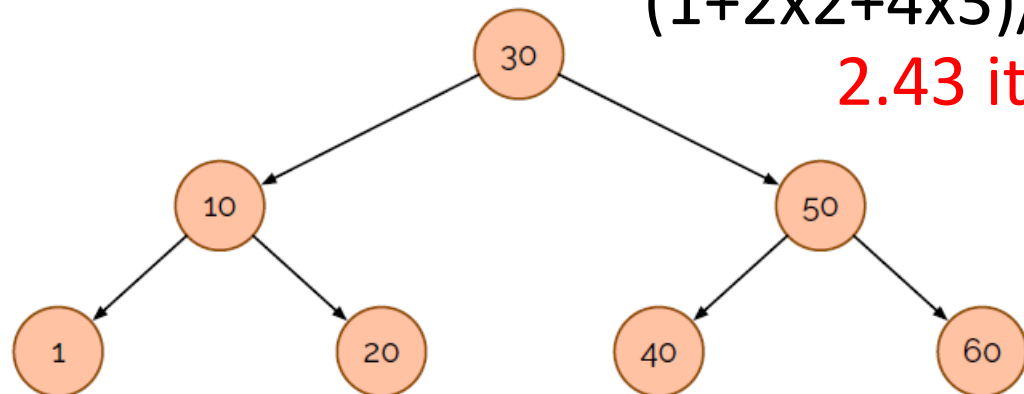
$$(1 + 2 + 2 + 3) / 4 = 2$$

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 – Generalização

- 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**

n : nº de elementos do array

k : nº de níveis da árvore

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

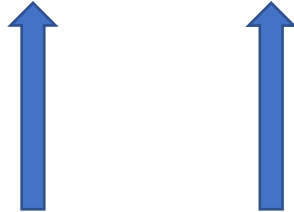
$n^\circ \text{ iter} = i + 1$

i é índice do nível

$n^\circ \text{ nó} = 2^i$

Caso Médio – Generalização

- Valor procurado pertence ao array – **Equiprobabilidade**
- Caso particular : $n = 2^k - 1$, $k = \log_2(n + 1)$
- **Nº de níveis** da árvore binária = k

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


Caso Médio – Generalização

- Expressões auxiliares

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

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

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

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

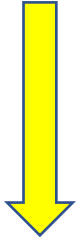
Comparação – Situações mais simples

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

- $A(1) = \log 2 - 1 + (\log 2)/1 = 1$ iterações médias
- $A(3) = \log 4 - 1 + (\log 4)/3 = 1 + 2/3$ iterações médias **OK!**
- $A(7) = \log 8 - 1 + (\log 8)/7 = 2 + 3/7$ iterações médias

base 2

Comparação – Caso Médio vs Pior Caso

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

E se puder **não pertencer** ao array ?

Nº de iterações

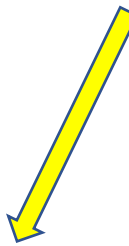
Casos possíveis		Nº de comparações	Probabilidade
Sucesso 0	É o 1º elemento	?	$1 / (2n + 1)$
Sucesso 1	É o 2º elemento	?	$1 / (2n + 1)$
...
Sucesso (n - 1)	É o último elemento	?	$1 / (2n + 1)$
Insucesso 0	Menor do que o 1º	Nº de níveis da árvore	$1 / (2n + 1)$
Insucesso 1	Entre o 1º e o 2º	Nº de níveis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

Caso Médio – Generalização

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

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

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

$$A(n) = \frac{1}{2n+1} [2k(n+1) - n] \approx k - \frac{1}{2} \approx \log_2(n+1) - \frac{1}{2}$$


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 : $n = 2^k - 1$, $k = \log_2(n + 1)$
- Cenário 1 : $A(n) \approx W(n) - 1$
- Cenário 2 : $A(n) \approx W(n) - \frac{1}{2}$ **$O(\log n)$**
- Para um **n qualquer**, obtermos **expressões “semelhantes”**

Algoritmos de Ordenação

Algoritmos de Ordenação

- Para **quê** ordenar os elementos de um conjunto/multi-conjunto ?
- Para **facilitar operações** de procura, intersecção, reunião, etc.
- Para **simplificar**, vamos considerar **arrays de números inteiros**
- MAS, **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 : $a[i-1] < a[i]$

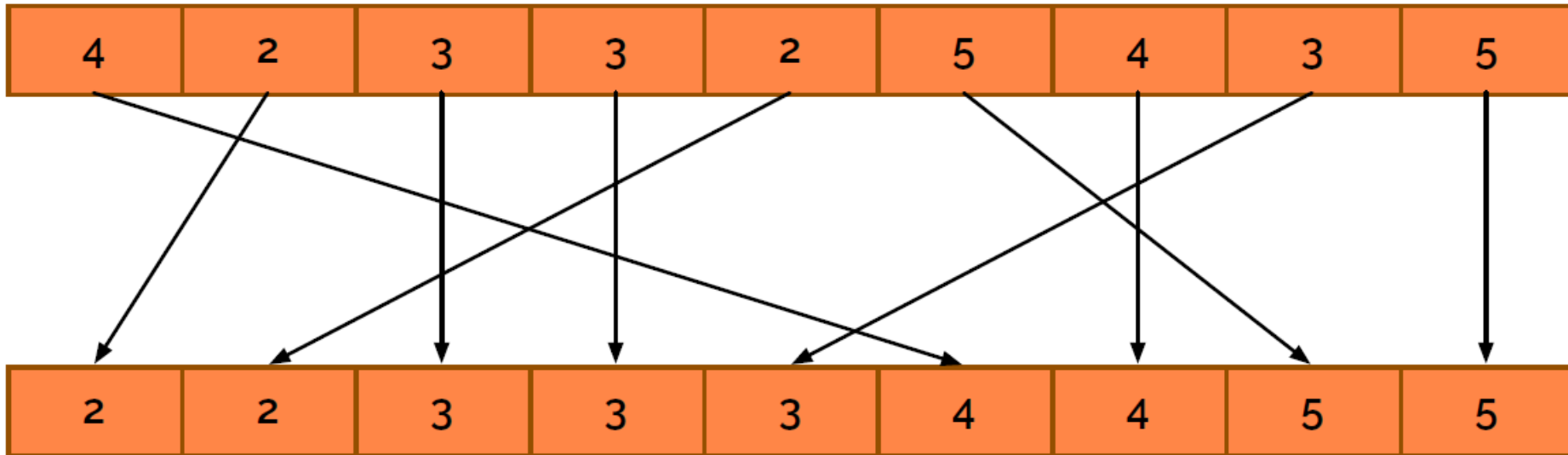
0	1	2	3	4	5
2	4	6	8	10	12

- 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 no 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

Ideia

- 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

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

- 4 comparações

Exemplo

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

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

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

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

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

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

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 !!



Nº de operações efetuadas ?

- Nº de comparações = ?
- $(n - 1) + (n - 2) + \dots + 2 + 1 = n \times (n - 1) / 2$ **$O(n^2)$**
- Nº de trocas = ?
- **Melhor caso** : $Bt(n) = 0$ - Quando ?
- **Pior caso** : $Wt(n) = n - 1$ - Quando ? **$O(n)$**

Trocar

```
void swap( int* a, int* b )    // Call-by-pointer
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```


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] );   
    }  
}
```

Nº de Comparações

- Número **fixo** de comparações !
- Mesmo que o array já esteja ordenado, continuamos a comparar !!

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

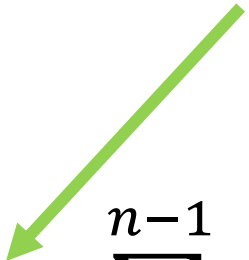
$$\mathbf{O(n^2)}$$

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 ?

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

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


Nº de Trocas – Caso Médio

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

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

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

Sugestão de leitura

Sugestão de leitura

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