Análise da Complexidade VII

Joaquim Madeira

08/04/2021

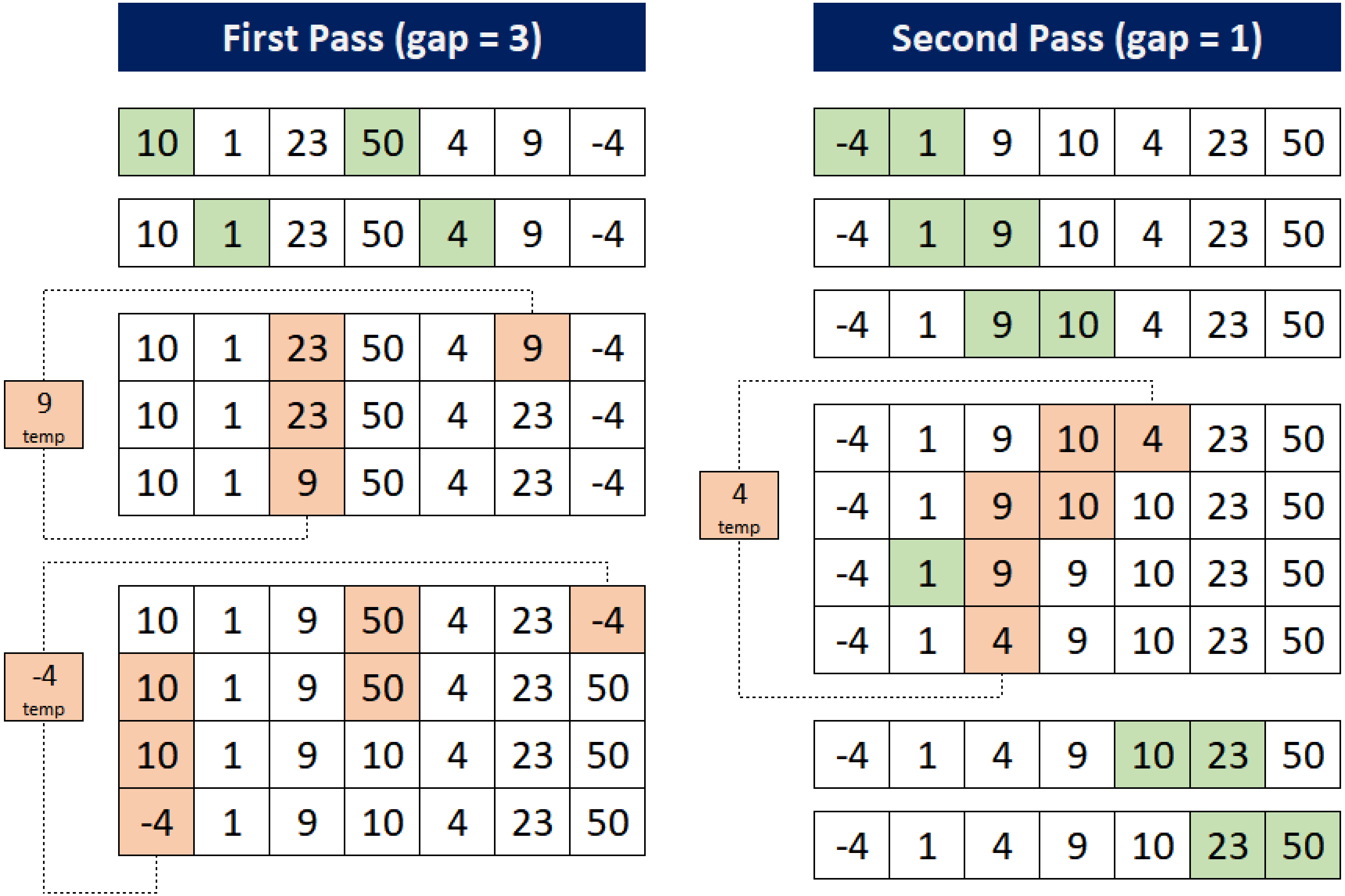
Sumário

- Recap
- A estratégia Transform-and-Conquer
- MAX-Heap – “Amontoado” binário
- Heap Sort
- Sugestão de leitura

Recapitulação

Shell Sort

- Seq. Hibbard
  - 1, 3, 7, …



[alphacodingskills.com]

Resolveram o exemplo ?

0	1	2	3	4	5	6	7	8	9	10	11	12
81	**94**	**11**	**96**	**12**	**35**	**17**	**95**	**28**	**58**	**41**	**75**	**15**

Shell Sort – Sequência original de distâncias

```
void shellSort( int a[], int n ) {
        for( int gap = n / 2; gap > 0; gap /= 2 )       // Gap sequence
                for( int i = gap; i < n; i++ ) {         // Elements to be sorted
                        int tmp = a[ i ];
                        int j = i;                       // Insertion sort
                        for( ; j >= gap && tmp < a[ j – gap ]; j – = gap )
                                a[ j ] = a[ j – gap ];
                        a[ j ] = tmp;
                }
}
```

Melhor Caso – Seq. original de distâncias

- Array ordenado !
- Considerar casos particulares, para simplificar : $n = 2^k$, $k = \log n$

$$B_C(n) = n \log n \; - (n-1) \qquad\qquad \boldsymbol{B_C(n) \in O(n \log n)}$$

- Esta ordem de complexidade, para o melhor caso, é comum à maioria das sequências de distâncias habitualmente consideradas

Pior Caso – Seq. original de distâncias

- Considerar casos particulares, para simplificar : $n = 2^k$, $k = \log n$

- Execuções (Pior Caso) do algoritmo Insertion Sort sobre
- 2^{k-1} conjuntos de 2 elementos
- 2^{k-2} conjuntos de 4 elementos

…
- 1 conjunto de $n = 2^k$ elementos

$$W_C(n) \in O(n^2)$$

Pior Caso

- MAS, obteve-se uma ordem de complexidade quadrática ?!?
- A sequência de distâncias original não é a melhor !

- Shell, 1959 : 1, …, n / 4, n / 2 $O(n^2)$
- Hibbard, 1963 : 1, 3, 7, 15, 31, … $O(n^{3/2})$
- Sedgewick, 1982 : 1, 8, 23, 77, 281, … $O(n^{4/3})$
- Sedgewick, 1982 : 1, 5, 19, 41, 109, … $O(n^{4/3})$

- A escolha da sequência de distâncias a usar tem um efeito “dramático” sobre a ordem de complexidade

Implementação Genérica

Desenvolvimento genérico

- Implementar cada algoritmo de ordenação uma só vez
  - Evitar redundância : copiar / colar / modificar

- MAS, arrays de diferentes tipos como argumento de entrada !!
- Como fazer ? -> Tipo genérico: void *

- MAS, a operação de comparação depende do tipo de elementos !!
- Como fazer ? -> Função de comparação é um argumento

Ponteiro para um função

- Em C, o identificador de uma função é um ponteiro !!

```
Int compare(int x, int y);  // protótipo

// declaração
// ponteiro para função; dois argumentos inteiros; devolve um inteiro
Int (*compFunc)(int a, int b);
compFunc = compare;
r = compFunc(5, 10);        // o mesmo que compare(5, 10)
```

void *

- O tipo void * é usado para definir ponteiros genéricos em C
  - Semelhante a Object em Java

- Um ponteiro de qualquer tipo pode ser atribuído a um ponteiro do tipo void *

- MAS, perde-se a informação quanto ao tipo da variável referenciada
- Fazer o casting para o tipo desejado, quando necessário

Tipos auxiliares – typedef

```c
// The type for the comparator less function: *p1 < *p2
typedef int (*lessFunc)(void* p1, void* p2);

// The type for the swap function
typedef void (*swapFunc)(void* p1, void* p2);

// The type for the print function
typedef void (*printFunc)(void* p);
```

Trocar dois elementos

```c
void swapForInts(void* p1, void* p2) {
  assert(p1 != NULL && p2 != NULL);
  int temp = *(int*)p1;
  *(int*)p1 = *(int*)p2;
  *(int*)p2 = temp;
}
```

```c
void swapForPersons(void *p1, void *p2) {
  assert(p1 != NULL && p2 != NULL);
  struct Person temp = *(struct Person *)p1;
  *(struct Person *)p1 = *(struct Person *)p2;
  *(struct Person *)p2 = temp;
}
```

Comparar dois inteiros

```c
int lessForInts(void* p1, void* p2) {
  assert(p1 != NULL && p2 != NULL);
  int a = *(int*)p1;
  int b = *(int*)p2;
  return a < b;
}
```

Duas funções de comparação

```c
int lessForFirstName(void *p1, void *p2) {
  assert(p1 != NULL && p2 != NULL);
  struct Person *first = (struct Person *)p1;
  struct Person *second = (struct Person *)p2;
  return strcmp(first->firstName, second->firstName) < 0;
}

int lessForLastName(void *p1, void *p2) {
  assert(p1 != NULL && p2 != NULL);
  struct Person *first = (struct Person *)p1;
  struct Person *second = (struct Person *)p2;
  return strcmp(first->lastName, second->lastName) < 0;
}
```

Função Genérica - bubbleSort

```
void bubbleSort(void *a, size_t numElems, size_t elemSize, lessFunc less,
                swapFunc swap) {
  assert(a != NULL && numElems > 0 && elemSize > 0 && less != NULL &&
         swap != NULL);

  void *previous;
  void *next;

  size_t k = numElems;
  int stop = 0;

  while (stop == 0) {
    stop = 1;
    k--;
```

Função Genérica - bubbleSort

```c
while (stop == 0) {
    stop = 1;
    k--;
    // Pointer arithmetic is not allowed on void pointers
    // The first element
    previous = a;
    // The second element
    next = (char *)a + elemSize;
    for (size_t i = 0; i < k; i++) {
        if (less(next, previous)) {
            swap(previous, next);
            stop = 0;
        }
        previous = next;
        next = (char *)next + elemSize;
    }
}
}
```

Tarefa 1

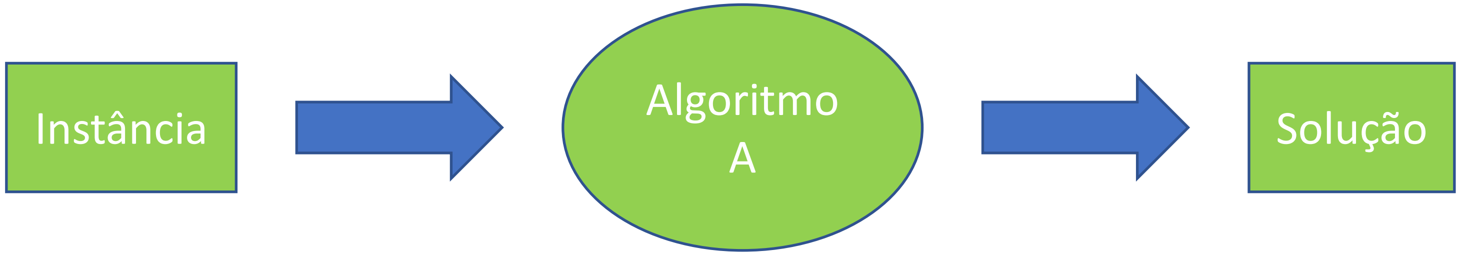
- Analisar os exemplos disponibilizados
  - Ordenar arrays de número inteiros
  - Ordenar arrays de registos

- Implementar versões genéricas dos outros algoritmos de ordenação
- Testar

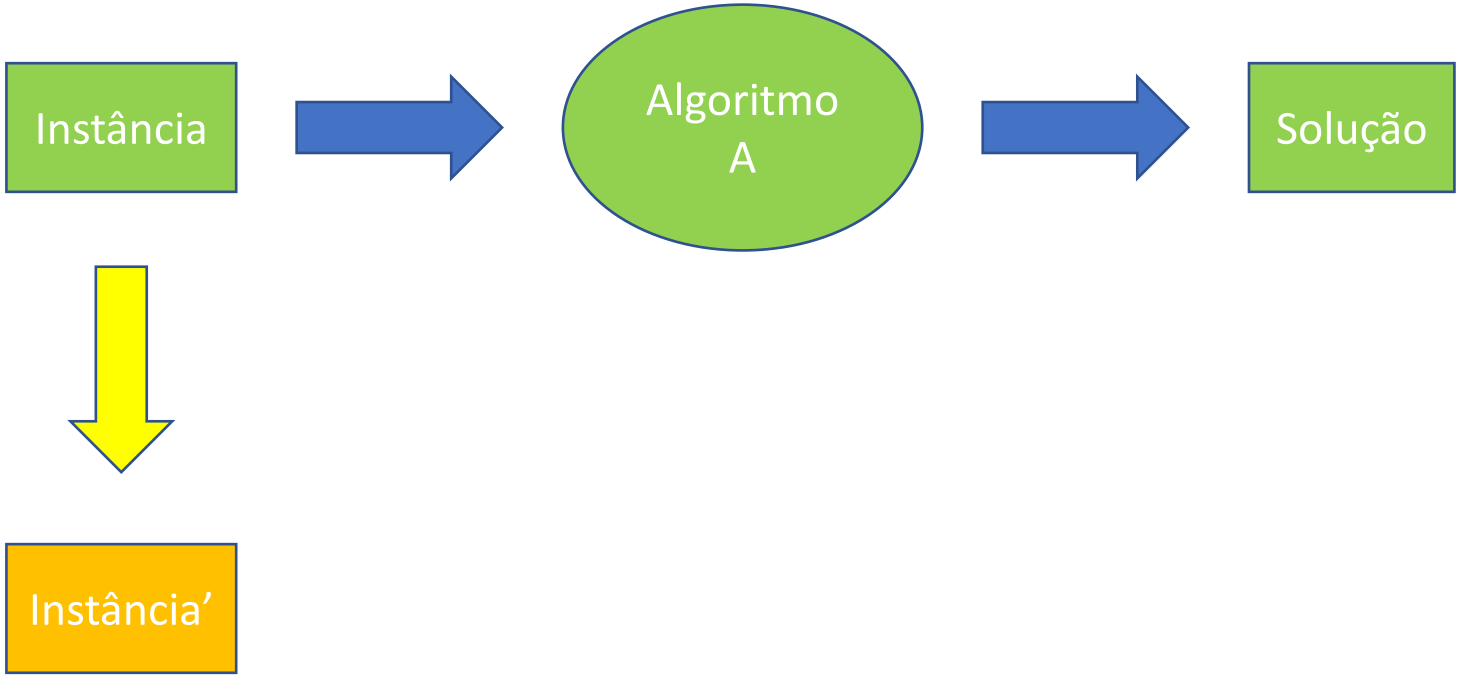
- Desenvolver outros exemplos, com arrays de diferentes tipos

Transform-and-Conquer

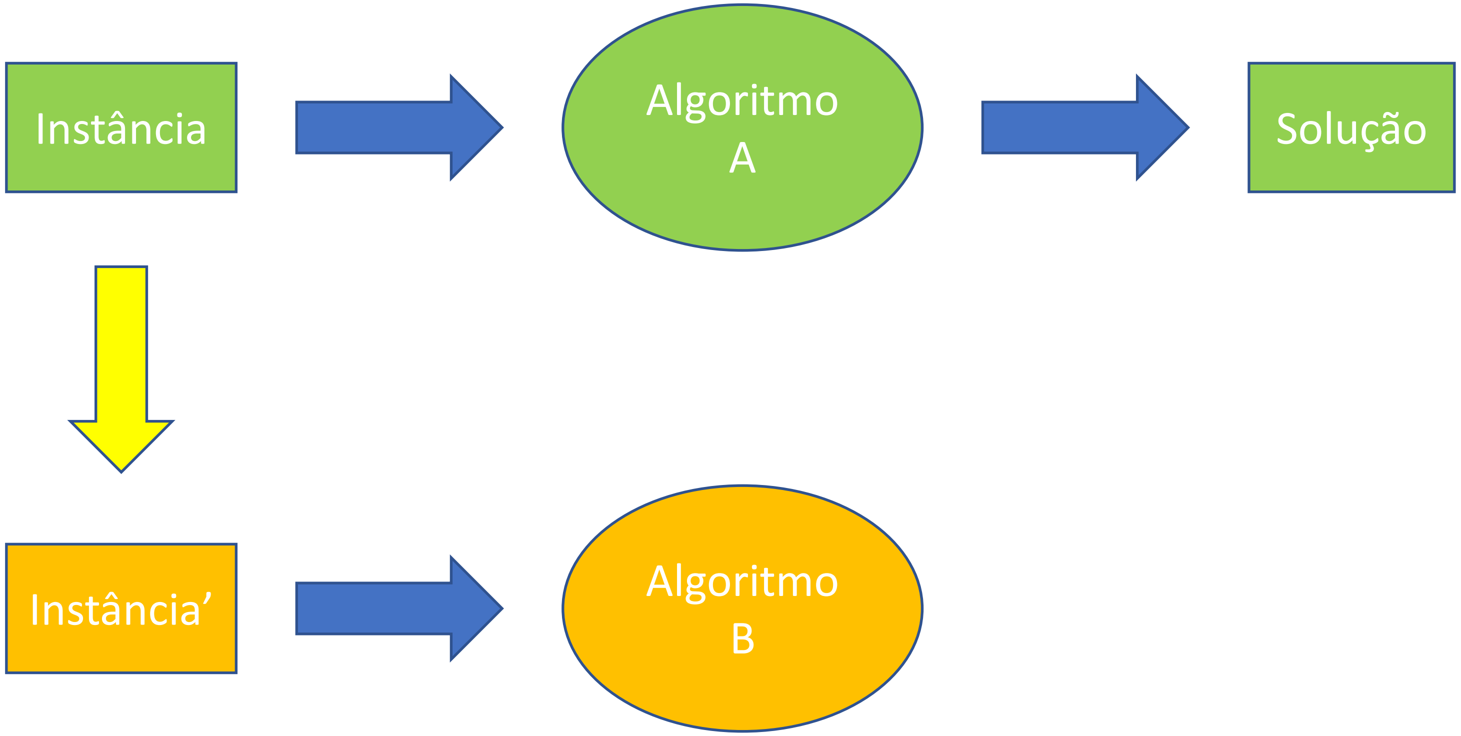
Transform-and-Conquer



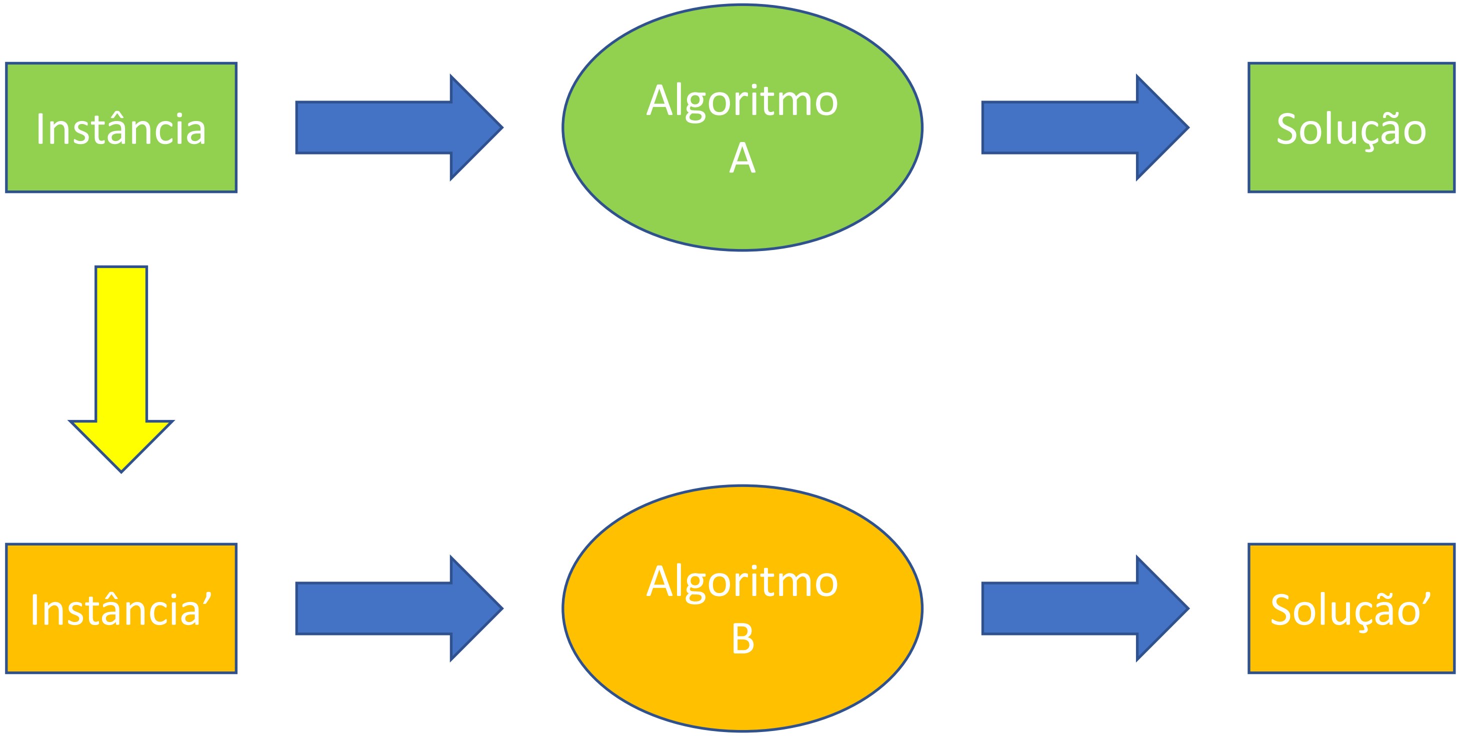
Transform-and-Conquer



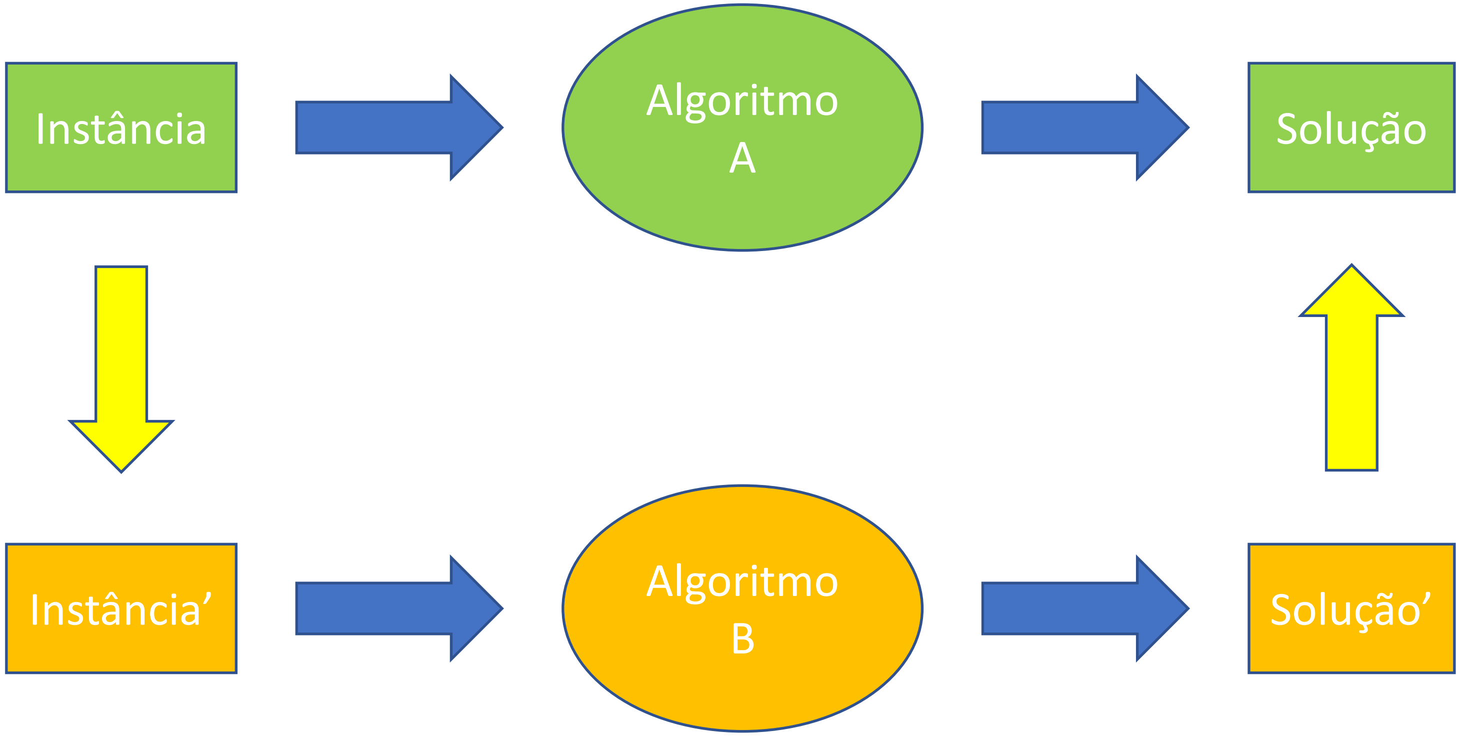
Transform-and-Conquer



Transform-and-Conquer



Transform-and-Conquer



Transform-and-Conquer

- Objetivo : baixo custo computacional !

- 1º passo : Transformação
- Modificar a instância dada, para que seja mais fácil resolver o problema proposto

- 2º passo : Conquista
- Resolver a instância modificada e obter a solução desejada

T&C – Alteração da representação

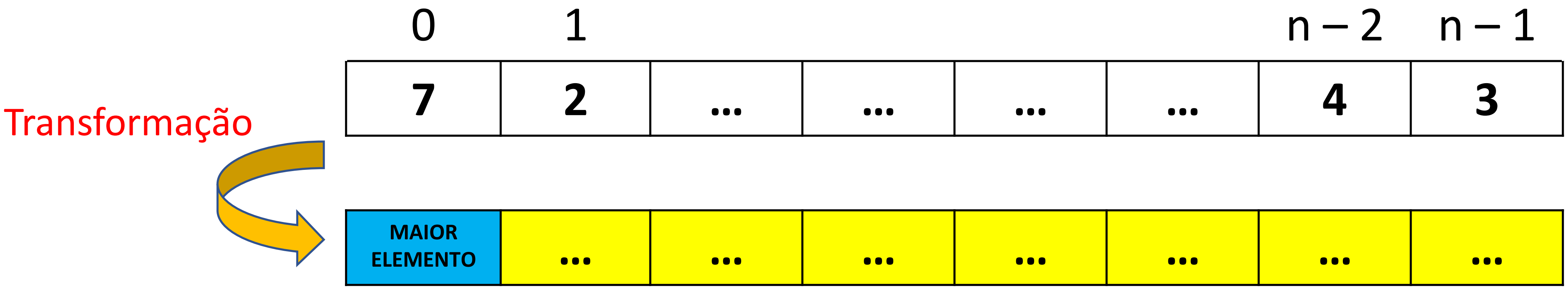
- Resolver uma instância de um problema transformando-a, primeiro, numa representação alternativa, que vai facilitar a resolução do problema inicial

- O custo computacional dessa transformação não deve ser elevado, para não penalizar o desempenho computacional de todo o processo

T&C – Como ordenar um array ?

0	1					n – 2	n – 1
7	**2**	**…**	**…**	**…**	**…**	**4**	**3**

T&C – Como ordenar um array ?



T&C – Como ordenar um array ?

0	1					n – 2	n – 1
7	**2**	…	…	…	…	**4**	**3**

Transformação

MAIOR ELEMENTO	**…**	**…**	**…**	**…**	**…**	**…**	**…**
…	**…**	**…**	**…**	**…**	**…**	**…**	**MAIOR ELEMENTO**

1 troca

T&C – Como ordenar um array ?

0	1					n – 2	n – 1
7	**2**	**…**	**…**	**…**	**…**	**4**	**3**

Transformação

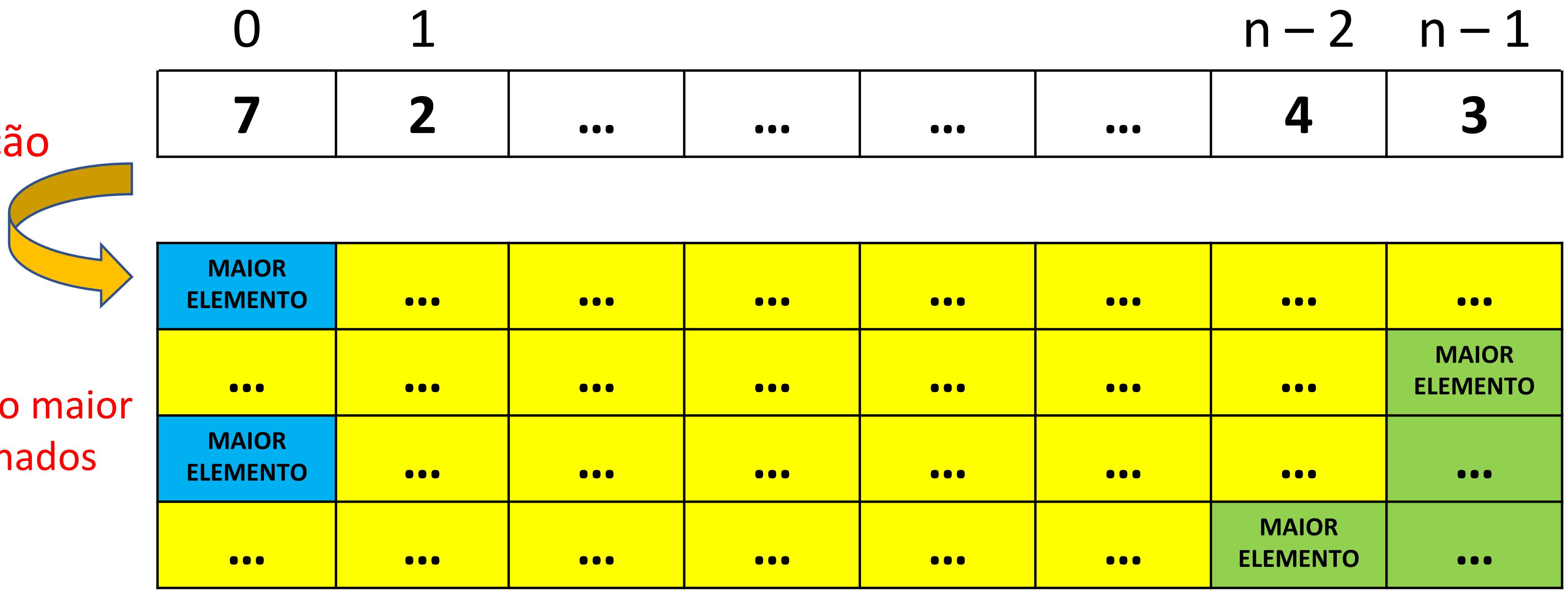
MAIOR ELEMENTO	…	…	…	…	…	…	…
…	…	…	…	…	…	…	MAIOR ELEMENTO
MAIOR ELEMENTO	…	…	…	…	…	…	…

1 troca

Reposicionar o maior dos não ordenados

T&C – Como ordenar um array ?

Transformação



Reposicionar o maior dos não ordenados

1 troca

1 troca

T&C – Como ordenar um array ?

0	1					n – 2	n – 1
7	**2**	…	…	…	…	**4**	**3**

Transformação

	MAIOR ELEMENTO	…	…	…	…	…	…	…	
Reposicionar o maior dos não ordenados	…	…	…	…	…	…	…	MAIOR ELEMENTO	1 troca
	MAIOR ELEMENTO	…	…	…	…	…	…	…	
Reposicionar o maior dos não ordenados	…	…	…	…	…	…	MAIOR ELEMENTO	…	1 troca
	MAIOR ELEMENTO	…	…	…	…	…	…	…	

… …

T&C – Como ordenar um array ?

- Objetivo : baixo custo computacional !

- Como obter o sucessivamente o maior elemento de um conjunto, sem manter ordenado esse conjunto de elementos ?

- Solução : usar uma representação alternativa – MAX-HEAP
- E não usar espaço de memória adicional, apenas o array dado

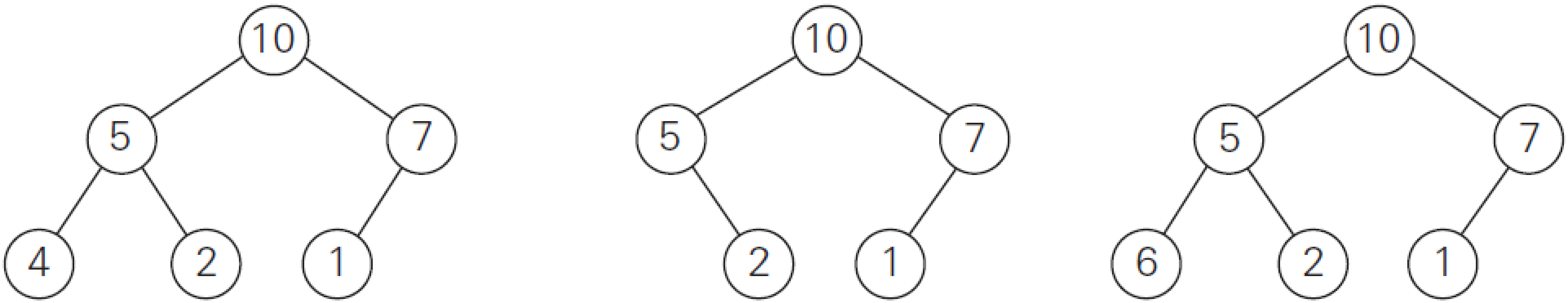
MAX-Heap
– Amontoado Binário

MAX-HEAP – O que é ?

- Árvore binária – Com restrições
- Forma – Árvore binária essencialmente completa
- Todos os níveis estão preenchidos, com a possível exceção do último, onde faltar algumas folhas do lado direito
- Critério de ordem – o valor registado em cada nó da árvore é maior ou igual que os valores registado nos seus filhos, caso existam

- Consequência : a sequência de valores em qualquer caminho da raiz da árvore até uma folha é não-crescente

São MAX-HEAPS ?



[Levitin]

Algumas propriedades – Encontrar exemplos

- Para uma MAX-HEAP com n nós :
- Altura = floor($\log_2$ n)
- Nº de folhas = ceil(n / 2)
- Nº de nós que não são folhas = floor(n / 2)

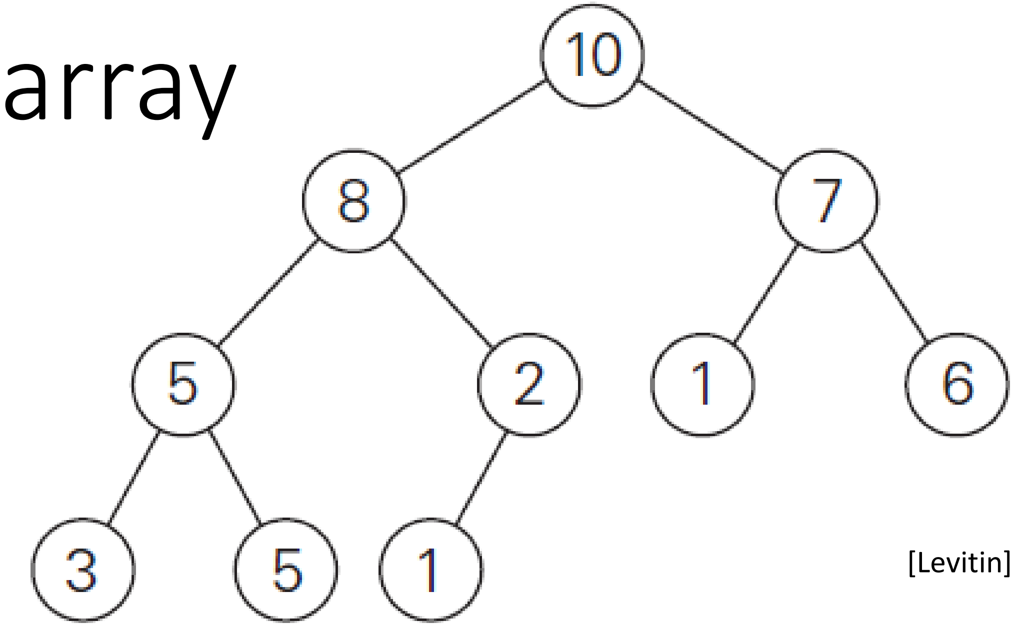
- A raiz de uma MAX-HEAP contém um seu elemento de maior valor

- Qualquer subárvore de uma MAX-HEAP é também uma MAX-HEAP

Representação usando um array

0	1	2	3	4	5	6	7	8	9
10	**8**	**7**	**5**	**2**	**1**	**6**	**3**	**5**	**1**

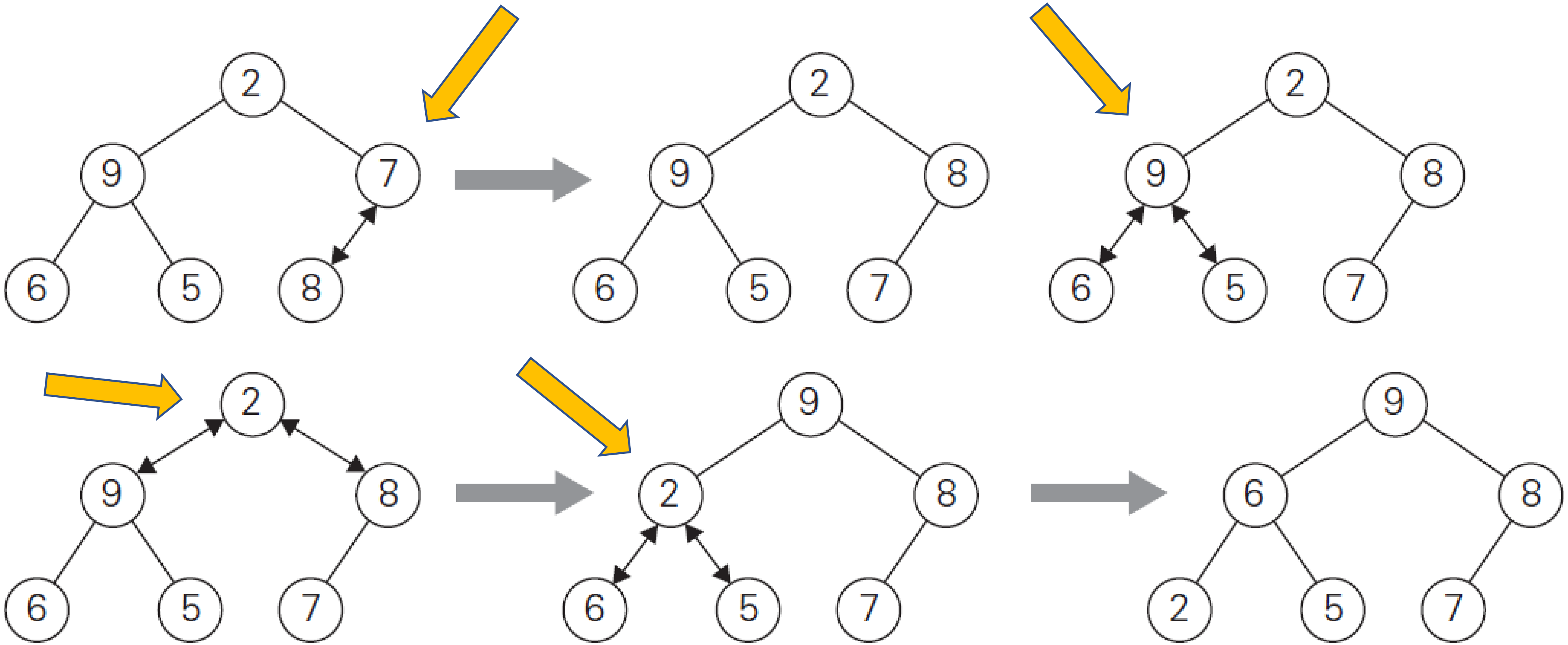
folhas



[Levitin]

- Armazenar de modo contíguo, da esquerda para a direita, num array
- LeftChild(i) = 2 x i + 1, se existir
- RightChild(i) = 2 x (i + 1), se existir
- Parent(i) = (i – 1) div 2, se i > 0

Construção de uma MAX-HEAP



[Levitin]

Construção de uma MAX-HEAP

```
void heapBottomUp( int a[], int n ) {
    for(int i = n / 2 - 1; i >= 0; i-- )       // Para cada elemento
        fixHeap( a, i, n );                    // que não é folha,
}                                              // reposicioná-lo,
                                               // se necessário
```

Construção de uma MAX-HEAP

```
void fixHeap( int a[], int index, int n ) {
        int child;
        for( int tmp = a[index]; leftChild(index) < n; index = child ) {
                child = leftChild(index);                                  // The largest
                if( child != (n - 1) && a[child + 1] > a[child] ) child++;  // moves up,
                if( tmp < a[child] ) a[index] = a [child];                  // if needed
                else break;
        }
        array[index] = tmp;                                        // Final position
}
```

Melhor Caso

- O array já é uma MAX-HEAP !!
- fixHeap é invocada floor(n /2) vezes
- O ciclo itera sempre uma só vez, para cada invocação de fixHeap
- E são efetuadas 2 comparações

- $B_C(n) = 2 \times \text{floor}(\ n\ /\ 2\ )$ **O(n)**

- Exemplo de uma configuração de melhor caso ?

Pior Caso

- Simplificação de análise: $n = 2^k - 1$ --- Níveis totalmente preenchidos
- Índice do último nível : $h = k - 1$

- fixHeap é invocada floor(n /2) vezes
- Em cada execução de fixHeap, um nó é deslocado do seu nível para o último nível da heap
- E são efetuadas 2 comparações, por cada nível, nesse percurso

Pior Caso

Nº de nós no nível i

$$W_C(n) = \sum_{i=0}^{h-1} 2^i \times 2 \times (h - i) = \ \ldots = 2[n - \log_2(n+1)]$$

Para cada nível, com a exceção do último

Distância até ao último nível

2 comparações

$$\boldsymbol{Wc(n) \in O(n)}$$

Tarefa 2

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

- Transformar numa MAX-HEAP usando o algoritmo heapBottomUp

Heap Sort

Estratégia T&C

Dado um array de n elementos

Construir uma MAX-HEAP **O(n)**

Repetir (n – 1) vezes

- Levar o maior elemento da MAX-HEAP para posição final – 1 TROCA
- Reorganizar os elementos não ordenados para MAX-HEAP – 1 x fixHeap

- Algoritmo in-place !!

Heap Sort

```
void heapSort( int a[], int n ) {
        heapBottomUp( a, n );
        for( int i = n - 1; i > 0; i-- ) {
                swap( &a[0], &a[i] );
                fixHeap( a, 0, i );          // Só a[0] pode
        }                                    //  necessitar de ser
}                                            // reposicionado !!
```

Análise da Complexidade – Comparações

- Construção inicial da MAX-HEAP : **O(n)**
- Ordenação do array : ?

- Quantas comparações são necessárias para reposicionar a raiz em MAX-HEAPs de tamanho decrescente ?
- k = (n – 1), (n – 2), …, 3, 2
- Altura = floor(log k)

- 2 comparações por nível até chegar ao nível desejado

Reorganização da Heap – Comparações

$$C(n) \le 2\lfloor\log_2(n-1)\rfloor + 2\lfloor\log_2(n-2)\rfloor + \cdots + 2\log_2(2)$$

$$C(n) \le 2\sum_{k=2}^{n-1}\log_2 k \le 2\log_2 n$$

$$\boldsymbol{C(n) \in O(n\, log_2\, n)}$$

heapSort – Comparações

- Nº de comparações realizadas pela função heapSort()

$$\boldsymbol{O(n) + O(n\, log_2\, n) = O(n\, log_2\, n)}$$

$$\boldsymbol{C(n) \in O(n\, log_2\, n)}$$

- $\boldsymbol{O(n\, log_2\, n)}$ no pior caso e no caso médio !! ☺

Tarefa 3

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

- Ordenar usando o algoritmo Heap Sort

Tarefa 4

0	1	2	3	4	5
2	**9**	**7**	**6**	**5**	**8**

- Ordenar usando o algoritmo Heap Sort

Tarefa 5

- Organizar configurações do array que correspondam:
- Ao melhor caso para as comparações
- Ao pior caso para as comparações
- Ao melhor caso para os deslocamentos
- Ao pior caso para os deslocamentos

- Alguns dos casos anteriores ocorrem em simultâneo ?

Sugestão de leitura

Sugestão de leitura

- J. J. McConnell, Analysis of Algorithms, 1st Edition, 2001
  - Capítulo 3: secção 3.5