

Complexidade de Algoritmos

Prof. Diego Buchinger
diego.buchinger@outlook.com
diego.buchinger@udesc.br

Prof. Cristiano Damiani Vasconcellos
cristiano.vasconcellos@udesc.br

Algoritmos Eficientes de Ordenação: Merge Sort, Quick Sort e Heap Sort

Dividir e Conquistar

Desmembrar o problema original em vários subproblemas semelhantes, resolver os subproblemas (executando o mesmo processo recursivamente) e combinar as soluções.

Merge Sort

A principal ideia do Merge Sort é ordenar partições do vetor e então reordenar o conjunto através da operação **merge**, uma mesclagem ordenada de dois vetores [$O(n)$].

O algoritmo de Merge Sort necessita de um espaço adicional de memória para trabalhar [$O(n)$].

2 3 7 9 10

1 4 5 6 8

Merge Sort

Algoritmo:

```
mergeSort( vet, ini, fim ){  
    se( ini >= fim ) retorna;  
    meio = (ini + fim) / 2;  
    mergeSort( vet, ini, meio );  
    mergeSort( vet, meio+1, fim);  
    merge( vet, ini, meio, fim);  
}
```

Aplicar Merge Sort sobre o vetor:

[6 – 5 – 3 – 1 – 8 – 7 – 2 – 4]

Merge Sort

Qual a complexidade de tempo do Merge Sort?

Existe um pior caso? Qual seria?

Qual a complexidade de espaço do Merge Sort?

Quick Sort

A principal ideia do Quick Sort é a ordenação com base em um elemento denominado **pivô**.

Deve-se ordenar o vetor mantendo todos os elementos menores do que o pivô a sua esquerda e todos os elementos maiores do que o pivô a sua direita (**pivoteamento**)

Após este processo realiza-se o mesmo procedimento para o grupo a esquerda do pivô e depois para o grupo a direita do pivô – enquanto o número de elementos for maior do que um.

Quick Sort

Algoritmo:

```
quickSort( vet, ini, fim ){  
    se( ini >= fim ) retorna;  
    meio = pivoteamento( vet, ini, fim );  
    quickSort( vet, ini, meio );  
    quickSort( vet, meio+1, fim );  
}
```

Aplicar Quick Sort sobre o vetor:

[6 – 5 – 3 – 1 – 8 – 7 – 2 – 4]

Quick Sort

Algoritmo de pivoteamento:

```
lomuto( vet, ini, fim ){  
    pivo = ini;  
    para( j=ini+1; j<=fim; j++ ){  
        se( vet[j] < vet[ini] ){  
            pivo++;  
            troca( vet[pivo], vet[j] );  
        }  
    }  
    troca( vet[ini], vet[pivo] );  
    retorne pivo;  
}
```

Quick Sort

Algoritmo de pivoteamento:

```
hoare( vet, ini, fim ){  
    pivo = vet[ini];  
    i = ini;  
    j = fim;  
    repita{  
        enquanto( vet[i] < pivo ) faça i = i+1;  
        enquanto( vet[j] > pivo ) faça j = j-1;  
        se( i < j ) troca( vet[i] , vet[j] );  
        senão retorne j;  
    }  
}
```

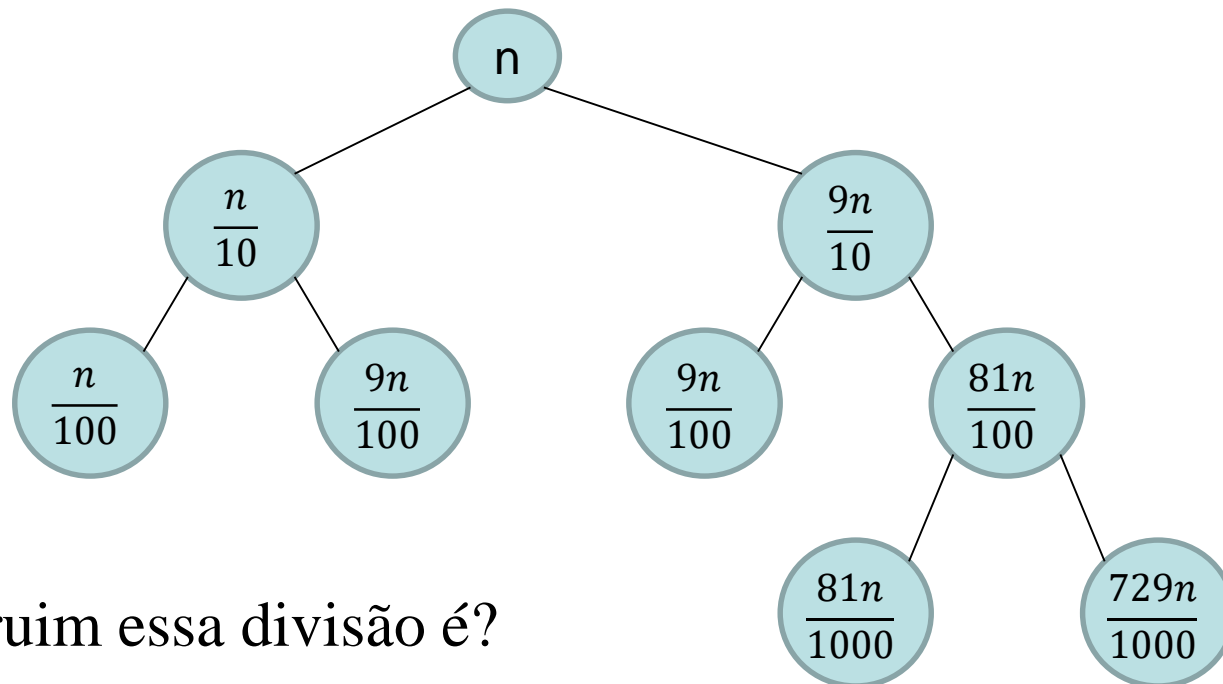
Quick Sort

Perguntas:

- Qualquer pivô serve?
- Existe um pivô ruim ou bom?
- Como escolher um pivô adequado?
- Qual o melhor caso para o Quick Sort? (complexidade)
- Qual o pior caso para o Quick Sort? (complexidade)
- Qual a complexidade de espaço do Quick Sort?

Quick Sort

Considere o seguinte cenário onde ocorre uma divisão desbalanceada de proporção constante 1:9



Quão ruim essa divisão é?

Quick Sort

Quão ruim essa divisão é?

$$T(n) = T(n/10) + T(9n/10) + n$$

$$T(1) = 1$$

(resolva a recursão com alguns valores arbitrários
para **n** e compare o crescimento com as funções n^2 e $n \log n$)

Heap Sort

É um arranjo, onde os dados estão organizados de forma que podem ser acessados como se estivessem armazenados em uma árvore binária.

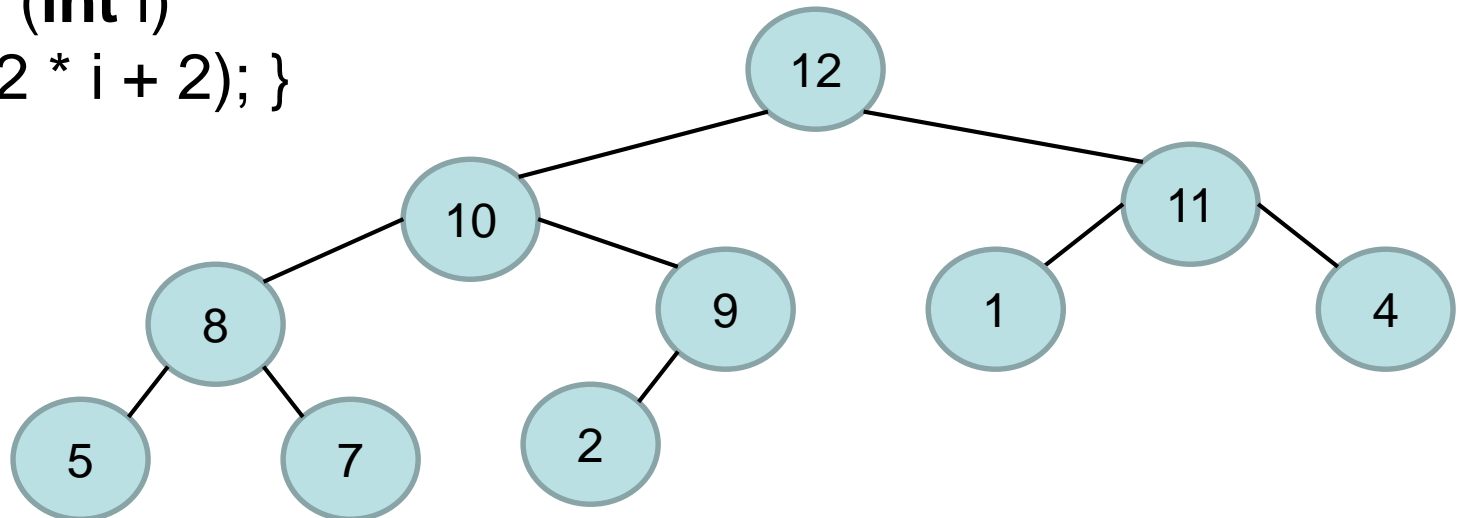
No caso de um *heap máximo*, os elementos armazenado em uma sub-árvore serão sempre menores que o elemento armazenado na raiz. Essa árvore é completa todos seus níveis, com a possível exceção do nível mais baixo.

Heap Sort

0	1	2	3	4	5	6	7	8	9
12	10	11	8	9	1	4	5	7	2

```
int esquerda (int i)
{ return (2 * i + 1); }
```

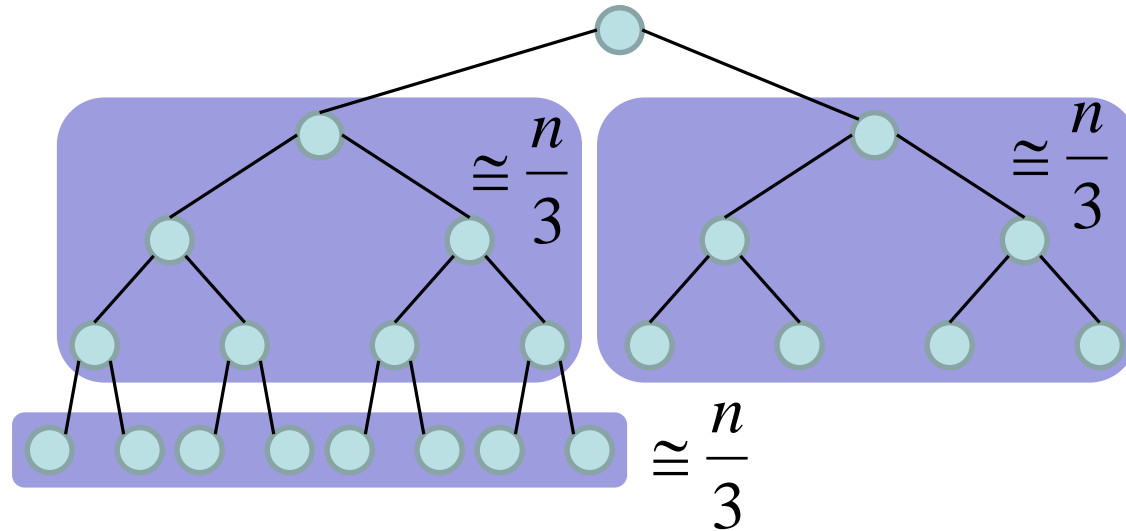
```
int direita (int i)
{ return (2 * i + 2); }
```



Heap Sort

```
// n = tamanho // i = índice
void heapify (int *a, int n, int i) {
    e = esquerda( i );
    d = direita( i );
    maior = i;
    if (e < n && a[e] > a[maior])
        maior = e;
    if (d < n && a[d] > a[maior])
        maior = d;
    if ( maior != i ) {
        swap (&a[i], &a[maior]);
        heapify(a, n, maior);
    }
}
```

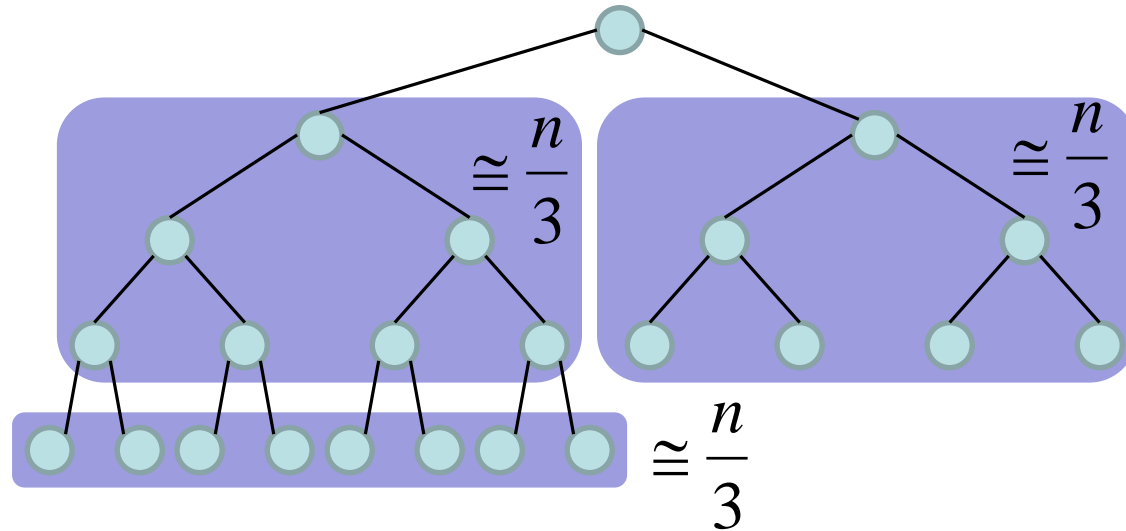

Heap Sort



$$T(n) = T(2n/3) + O(1)$$

$$T(1) = O(1)$$

Heap Sort



Note ainda que existem no máximo: $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nós de altura h

$n=23 \Rightarrow h=0 : 12$ (folha)

$h=1 : 6$

$h=2 : 3$

$h=3 : 2$

(Obs: altura de baixo p/ cima)

Heap Sort

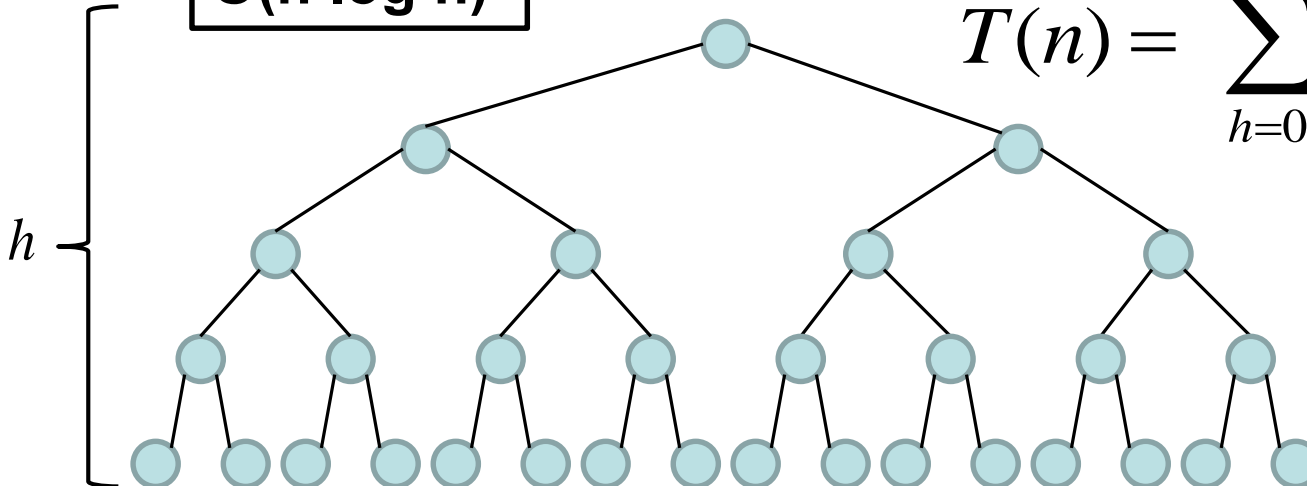
```
void buildHeap ( int *a, int n ) {
    int i;
    for (i = n/2; i >= 0; i--)      //O(n)
        heapify(a, n, i);          //O(log n)
}
```

$O(n \log n)$

~~$\Theta(n \log n)$~~

heapify varia com a altura do nó!

$$T(n) = \sum_{h=0}^{\lfloor \log_2 n \rfloor} [(n / 2^{h+1}) O(h)]$$



Heap Sort

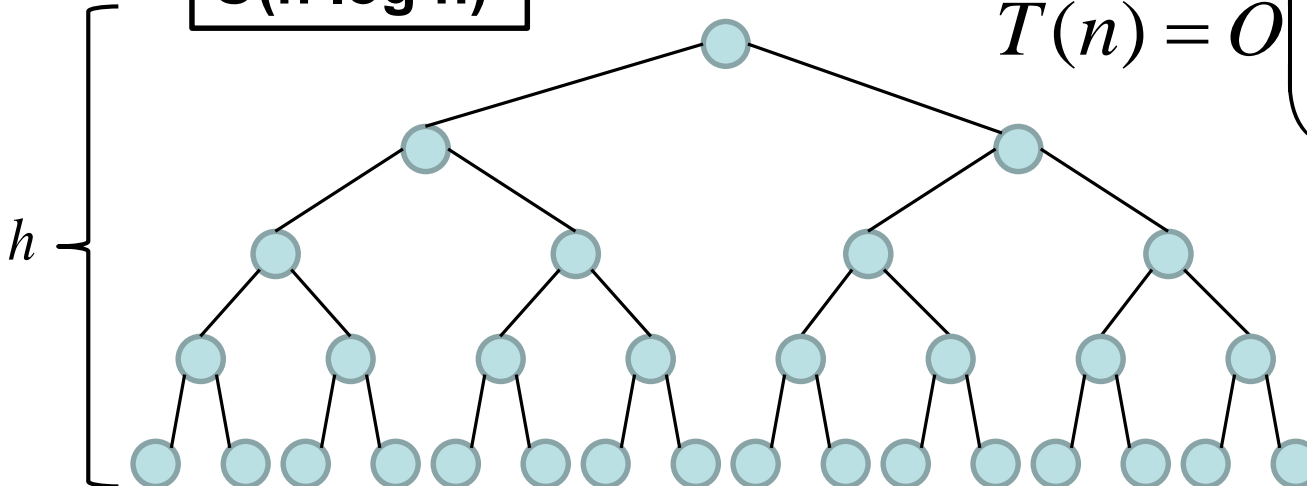
```
void buildHeap ( int *a, int n ) {
    int i;
    for (i = n/2; i >= 0; i--)      //O(n)
        heapify(a, n, i);          //O(log n)
}
```

$O(n \log n)$
 ~~**$\Theta(n \log n)$**~~

heapify varia com a altura do nó!

$$T(n) = O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} (h / 2^h)\right)$$

$\Theta(n)$



Heap Sort

```
void heapSort( int *a, int n) {  
    buildHeap( a , n );           //  $\Theta(n)$   
    for (i=n-1; i>0; i--) {       //  $\Theta(n)$   
        swap(&a[0], &a[i]);       //  $\Theta(1)$   
        heapify(a, i, 0);         //  $\Theta(\log n)$   
    }  
}
```

$\Theta(n \log n)$

Ordenação em tempo Linear

Counting Sort + Bucket Sort

Ordenações Lineares

Métodos de ordenação por comparação:

$$\Omega(n \log n)$$

Ordenações lineares [$\Omega(n)$] só são possíveis em **determinadas** condições.

Counting Sort

Pressupõe valores inteiros no intervalo 1 a k.

Algoritmo:

- contar o n° de elementos menores que 'x';
- usar esta informação para alocar o elemento na sua posição correta no vetor final;

Exemplos:

Caso bom

2	5	3	0	2	3	0	5	3	0
---	---	---	---	---	---	---	---	---	---

Caso ruim

2	25	13	100	250	300.000	1	5	3	0
---	----	----	-----	-----	---------	---	---	---	---

Bucket Sort

Pressupõe que a entrada consiste de elementos com distribuição de valores uniforme.

Algoritmo:

- separar os elementos em grupos / baldes;
- ordenar os elementos nos seus baldes;

Exemplos:

Caso bom

20	16	43	0	22	13	33	40	31	7
----	----	----	---	----	----	----	----	----	---

Caso ruim

2	43	3	5	0	7	8	1	6	4
---	----	---	---	---	---	---	---	---	---