Algoritmos e Estrutura de Dados II

Prof. Fellipe Guilherme Rey de Souza

Aula 23 – Algoritmos de Ordenação II

Agenda

- Quicksort
- Heapsort

Quicksort

- O *Quicksort* é um algoritmo de ordenação eficiente e amplamente utilizado, que segue a estratégia de divisão e conquista.

- Sua principal ideia é selecionar um elemento chamado pivô e, a partir dele, particionar o vetor em duas partes: uma com os elementos menores que o pivô e outra com os elementos maiores.

Quicksort

- Em seguida, o processo é repetido recursivamente nas duas partes, até que o vetor esteja totalmente ordenado.

- Por ser geralmente muito rápido, mesmo em grandes conjuntos de dados, o Quicksort é uma das escolhas preferidas em situações que exigem fim desempenho.

Quicksort

- Sua complexidade média é $O(n \log n)$, embora no pior caso possa chegar a $O(n^2)$, o que é raro se o pivô for bem escolhido.

- A complexidade do *Quicksort* em um vetor já ordenado (crescente ou decrescente) com o pivô sendo sempre o primeiro ou o último elemento também se torna o pior caso.

Quicksort

- Isso ocorre porque, em cada chamada recursiva, o particionamento divide o vetor em uma parte com $n - 1$ elementos e outra com 0 elementos.
  - O que leva o Quicksort a realizar muitas chamadas recursivas e comparações desbalanceadas.

Quicksort

- O passo-a-passo do *Quicksort* pode ser descrito como:
  1. Escolha um elemento da lista como pivô (normalmente o primeiro, o último, ou o elemento do meio).
  2. Reorganize os elementos da lista de forma que todos os elementos menores que o pivô fiquem à esquerda dele, e todos os maiores fiquem à direita. Esse processo é chamado de particionamento.

Quicksort

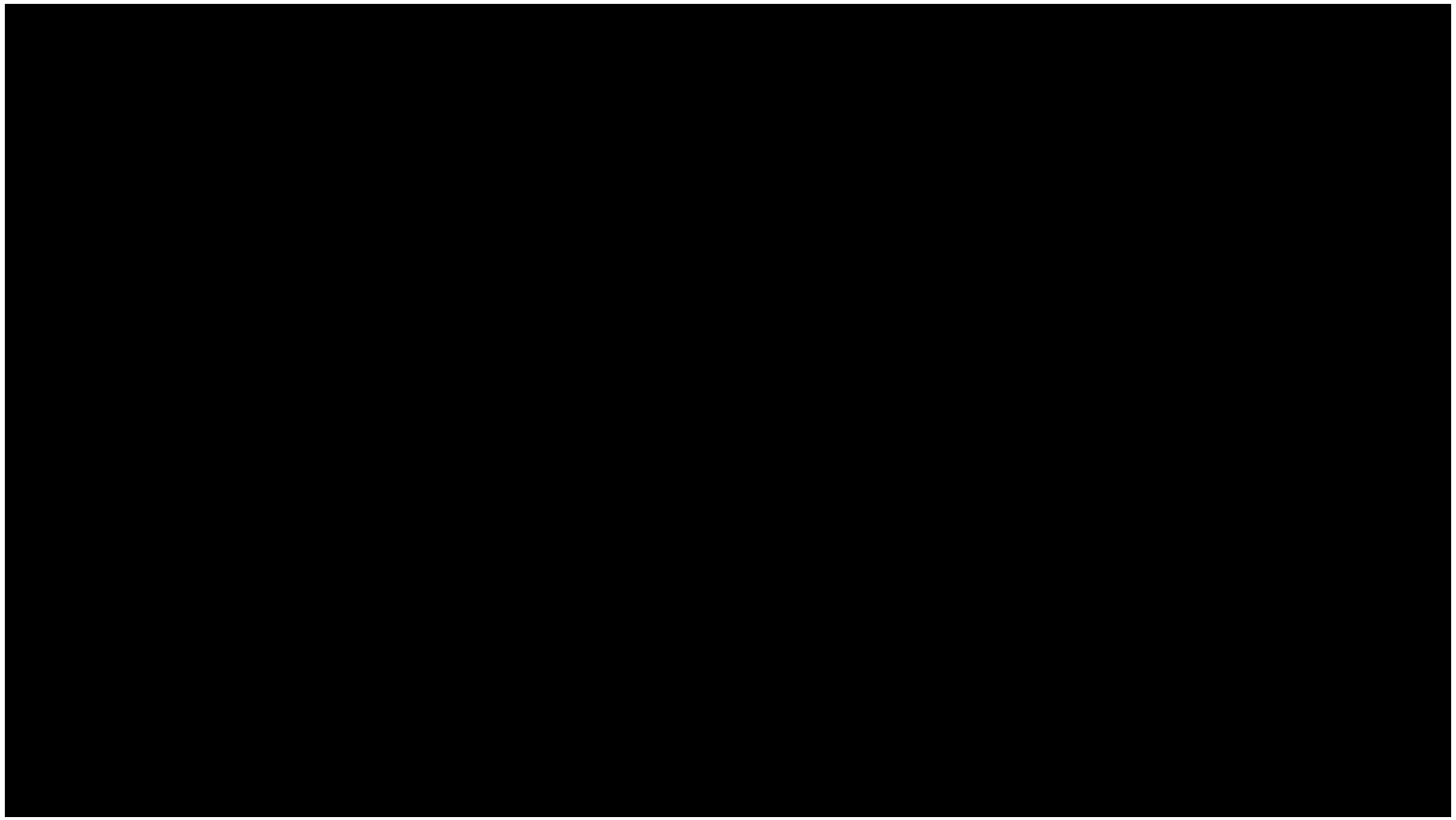
- O passo-a-passo do *Quicksort* pode ser descrito como (*cont.*):
    3. Após o particionamento, o pivô estará em sua posição correta na lista ordenada.
    4. Recursivamente, aplique o mesmo processo de ordenação nas sublistas à esquerda e à direita do pivô.
    5. O algoritmo termina quando as sublistas tiverem tamanho 0 ou 1, pois nesse caso já estão ordenadas.

Quicksort

An animated demonstration of Quicksort using Hoare's partition scheme.

Disponível em: <https://en.wikipedia.org/wiki/Quicksort#/media/File:Quicksort-example.gif>

Quicksort



Quick sort with Hungarian, folk dance.

Disponível em: <https://www.youtube.com/watch?v=3San3uKKHgg>

Quicksort

- A função particionar tem como objetivo reorganizar os elementos de um vetor em torno de um valor chamado pivô, que nesse caso é o último elemento da parte do vetor que está sendo considerada, ou seja, *arr[fim]*.

- O processo começa inicializando uma variável *i* como uma posição antes da primeira válida (*inicio - 1*).

```c
int particionar(int arr[], int inicio, int fim) {
    int pivô = arr[fim];
    int i = inicio - 1;

    for (int j = inicio; j < fim; j++) {
        if (arr[j] < pivô) {
            i++;
            trocar(&arr[i], &arr[j]);
        }
    }
    trocar(&arr[i + 1], &arr[fim]);
    return i + 1;
}
```

Quicksort

- Em seguida, um laço percorre todos os elementos entre os índices *inicio* e *fim - 1*, comparando cada um deles com o pivô.

- Sempre que um elemento menor que o pivô é encontrado, *i* é incrementado e o elemento atual é trocado com o que está na posição *i*, fazendo com que os elementos menores que o pivô fiquem agrupados à esquerda.

```c
int particionar(int arr[], int inicio, int fim) {
    int pivô = arr[fim];
    int i = inicio - 1;

    for (int j = inicio; j < fim; j++) {
        if (arr[j] < pivô) {
            i++;
            trocar(&arr[i], &arr[j]);
        }
    }
    trocar(&arr[i + 1], &arr[fim]);
    return i + 1;
}
```

Quicksort

- Depois que o laço termina, a função troca o pivô com o elemento que está logo após o último menor que ele, ou seja, na posição *i + 1*.

- Isso coloca o pivô exatamente onde ele deve estar no vetor ordenado.

- Por fim, a função retorna o índice dessa posição, que será usado para dividir o vetor em duas partes na próxima etapa do Quick Sort.

```c
int particionar(int arr[], int inicio, int fim) {
    int pivô = arr[fim];
    int i = inicio - 1;

    for (int j = inicio; j < fim; j++) {
        if (arr[j] < pivô) {
            i++;
            trocar(&arr[i], &arr[j]);
        }
    }
    trocar(&arr[i + 1], &arr[fim]);
    return i + 1;
}
```

Quicksort

```c
void quickSort(int arr[], int inicio, int fim) {
    if (inicio < fim) {
        int pi = particionar(arr, inicio, fim);

        quickSort(arr, inicio, pi - 1);
        quickSort(arr, pi + 1, fim);
    }
}
```

- A função quickSort ordena um vetor dividindo-o em partes menores com base em um pivô.

- Ela verifica se ainda há elementos a ordenar entre os índices dados.

Quicksort

```c
void quickSort(int arr[], int inicio, int fim) {
    if (inicio < fim) {
        int pi = particionar(arr, inicio, fim);

        quickSort(arr, inicio, pi - 1);
        quickSort(arr, pi + 1, fim);
    }
}
```

- Se houver, chama a função particionar, que posiciona o pivô no lugar certo, e em seguida chama a si mesma recursivamente para ordenar as partes à esquerda e à direita do pivô.

- O processo continua até que todas as partes estejam ordenadas.

Heapsort

- O Heapsort é um algoritmo de ordenação baseado em uma estrutura de dados chamada heap, que é uma árvore binária completa onde os elementos seguem uma ordem específica:
  - No heap máximo (*max heap*), cada pai é maior que os filhos e;
  - No heap mínimo (*min heap*), cada pai é menor.
  - Vimos sobre a Árvore Heap na Aula 17 ☺

Heapsort

- O Heapsort transforma o vetor em um heap máximo e, a partir disso, extrai repetidamente o maior elemento (a raiz), colocando-o no final do vetor. Esse processo continua até que todos os elementos estejam ordenados.

- É um algoritmo eficiente, com complexidade $O(n \log n)$ no pior caso, e tem a vantagem de não usar memória extra além do próprio vetor.

Heapsort

- O passo-a-passo do *Heapsort* pode ser descrito como:
  1. Construa um heap máximo a partir da lista, reorganizando os elementos para que o maior valor esteja na raiz (início da lista).
  2. Troque o primeiro elemento (o maior) com o último elemento da lista.
  3. Reduza o tamanho da lista considerada (ignorando o último elemento, que já está na posição correta).
  4. Refaça o heap máximo com os elementos restantes.
  5. Repita os passos 2 a 4 até que toda a lista esteja ordenada.

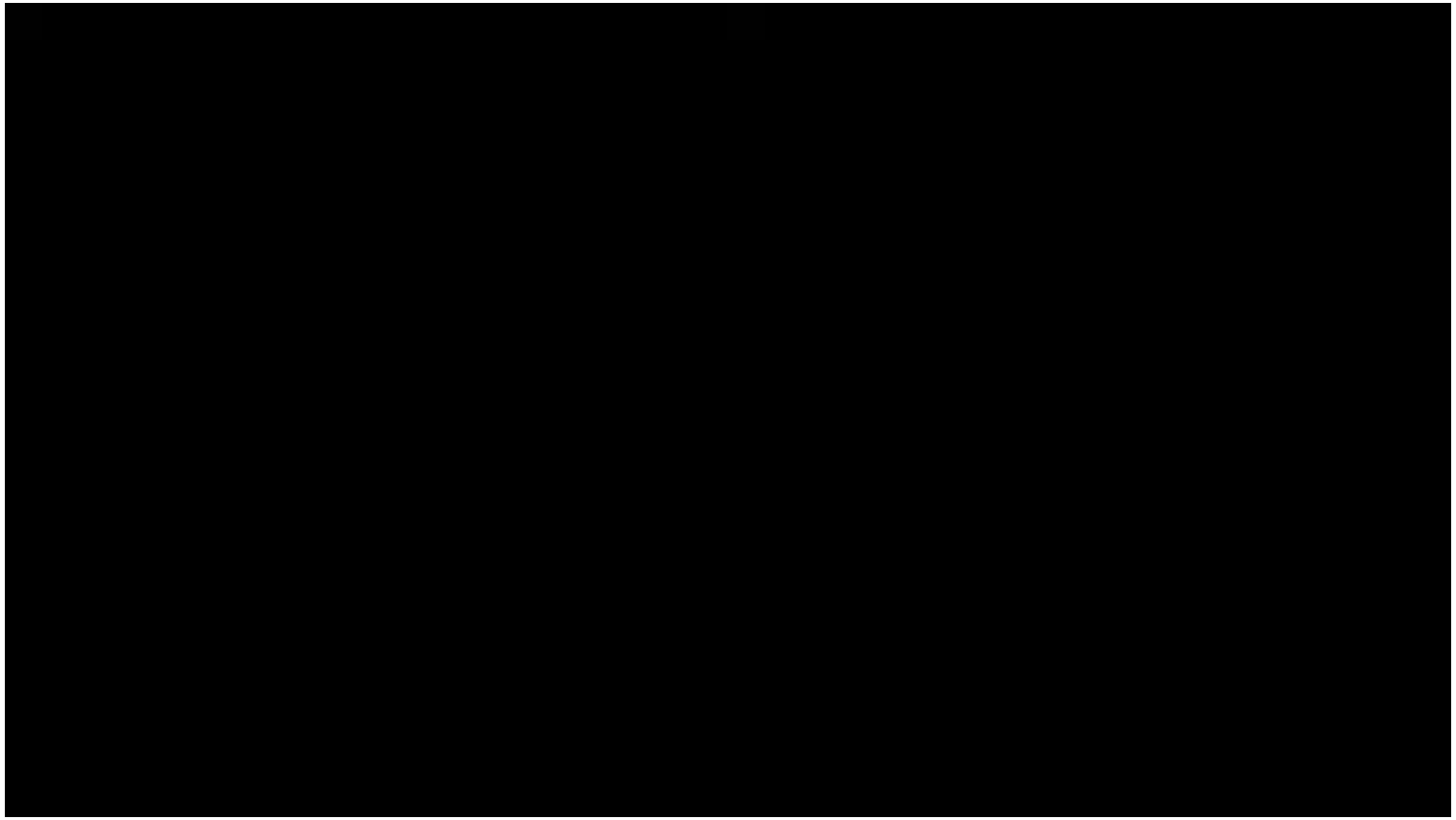
Heapsort

6 5 3 1 8 7 2 4

Exemplo de execução do Heapsort.

Disponível em: <https://pt.wikipedia.org/wiki/Heapsort#/media/Ficheiro:Heapsort-example.gif>

Heapsort



Quick sort with Hungarian, folk dance.

Disponível em: <https://www.youtube.com/watch?v=nms-xZlnHlM>

Heapsort

```c
void construirMaxHeap(MaxHeap* heap) {
    for (int i = (heap->tam / 2) - 1; i >= 0; i--) {
        restaurarMaxHeap(heap, i);
    }
}
```

- A função *construirMaxHeap* é responsável por transformar um vetor em uma estrutura de Max Heap, ou seja, ela organiza o vetor de modo que a propriedade de Max Heap seja mantida.

- Ela faz isso percorrendo o vetor a partir do meio, indo até o início. O motivo de começar do meio do vetor é que a partir dessa posição, todos os elementos já são folhas, e os elementos não-folhas começam a partir do índice *(tam / 2) - 1*.

Heapsort

```c
void construirMaxHeap(MaxHeap* heap) {
    for (int i = (heap->tam / 2) - 1; i >= 0; i--) {
        restaurarMaxHeap(heap, i);
    }
}
```

- Para cada elemento não-folha, a função chama restaurarMaxHeap para garantir que a subárvore com raiz nesse elemento siga a propriedade do Max Heap, ou seja, cada nó seja maior que seus filhos.

- Isso é repetido para todos os elementos não-folhas, o que garante que, no final, todo o vetor esteja estruturado como um Max Heap.

Heapsort

- A função heapsort começa chamando *construirMaxHeap*, que organiza o vetor em uma Max Heap.

- Em seguida, ela entra em um loop que percorre o vetor do último elemento até o segundo.

- Dentro do loop, o maior elemento (que está no topo da Max Heap) é trocado com o último elemento do vetor.

```c
void heapsort(MaxHeap* heap) {
    construirMaxHeap(heap);

    for (int i = heap->tam - 1; i > 0; i--) {
        troca(&heap->arr[0], &heap->arr[i]);

        heap->tam--;
        restaurarMaxHeap(heap, 0);
    }
}
```

Heapsort

- Após essa troca, o tamanho do heap é reduzido em um, e a função restaurarMaxHeap é chamada para garantir que o heap continue sendo uma Max Heap, agora considerando a parte do vetor que ainda não foi ordenada.

```
void heapsort(MaxHeap* heap) {
    construirMaxHeap(heap);

    for (int i = heap->tam - 1; i > 0; i--) {
        troca(&heap->arr[0], &heap->arr[i]);

        heap->tam--;
        restaurarMaxHeap(heap, 0);
    }
}
```

- Esse processo é repetido até que todos os elementos tenham sido ordenados, deixando o vetor em ordem crescente ao final.

Algoritmos e Estrutura de Dados II

Prof. Fellipe Guilherme Rey de Souza

Aula 23 – Algoritmos de Ordenação II