Técnicas e Análise de Algoritmos

Busca e Ordenação - Parte 03

Professor: **Jeremias Moreira Gomes**

E-mail: jeremias.gomes@idp.edu.br

Introdução

Introdução

- Definições de Ordenação
  - Parcial e Total
- Características de um algoritmo de ordenação
  - Interno ou externo
  - In-place ou out-place
  - Estável ou não-estável
- Ordenações Quadráticas

Ordenações Linearítmicas

Merge Sort

Merge Sort

- O MergeSort é um algoritmo de ordenação antigo, já conhecido por John von Neumann em 1945
- Ele usa o paradigma **dividir-e-conquistar** para ordenar os elementos de um vetor

Merge Sort

- Ele divide o vetor em duas metades, ordena cada uma delas e, em seguida, funde ambas partes em um todo ordenado
- O algoritmo é replicado em cada uma das metades, até que o tamanho de cada parte seja trivialmente ordenável
- A complexidade é linearítmica, isto é, O(N log N), onde N é o número de elementos no vetor

Merge Sort

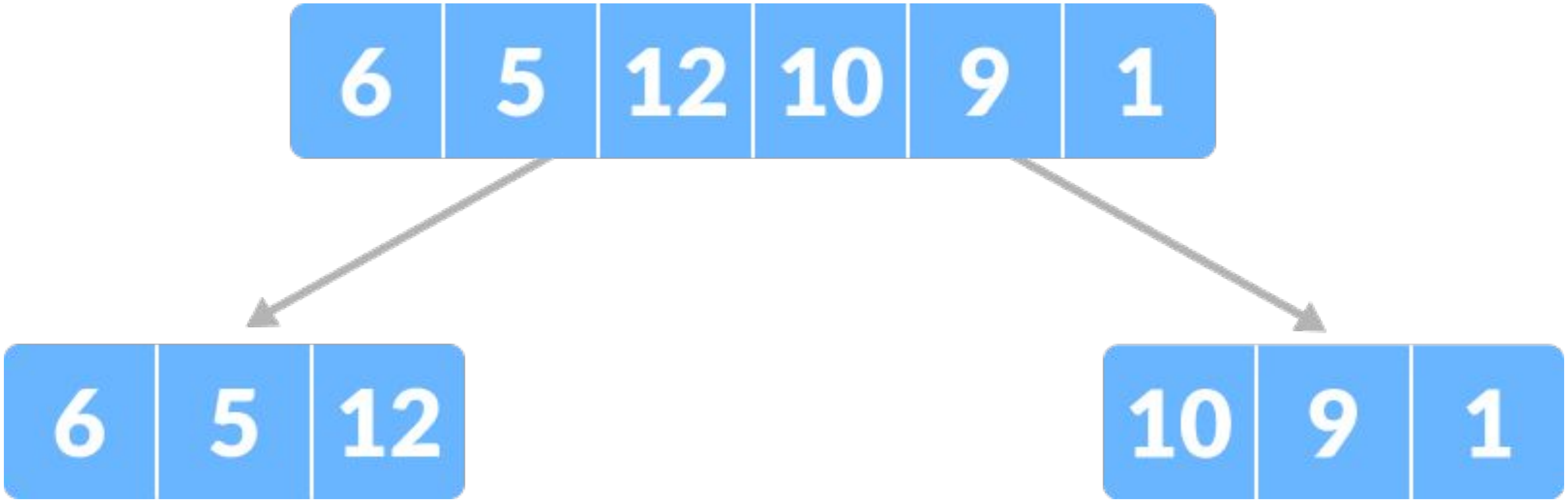
- O funcionamento do algoritmo possui duas funcionalidades distintas:
  - **Divisão do vetor em partes menores (dividir)**
  - **Fusão das partes ordenadas (conquistar)**

Merge Sort - Divisão em partes menores

- A função de divisão verifica se um vetor é maior que um valor x:
  - Caso o seja, ela irá dividir o vetor em duas partes iguais
  - Caso não o seja, irá retornar o próprio vetor

Merge Sort - Divisão em partes menores

- Exemplo da divisão com valor igual a 1



Merge Sort - Fusão das Partes Ordenadas

- O procedimento consiste em percorrer dois vetores ordenados, a partir do primeiro elemento de cada vetor, adicionado a um terceiro vetor os elementos de maneira ordenada
  - O menor de cada a cada comparação

Merge Sort - Fusão das Partes Ordenadas

- Exemplo da fusão das partes ordenadas

Merge Sort - Fusão das Partes Ordenadas

- A fusão é aplicada no algoritmo de merge sort porque vetores de tamanho 1 estão trivialmente ordenados
- Então, ao dividir um vetor recursivamente até o tamanho de 1, e aplicar a fusão das partes ordenadas nesses fragmentos do vetor resulta no vetor ordenado
  - Esta fusão não pode ser feita in-place, então gera custo de memória (O(N))

Merge Sort

- Assim, o algoritmo de merge sort funciona da seguinte forma:
  - A função mergesort é a própria função de divisão em partes menores, que chama recursivamente a própria função, para cada metade de um vetor
  - A função merge é a função de fusão das partes ordenadas, uma vez que, após alcançar o tamanho mínimo (1), este está trivialmente ordenado

Merge Sort

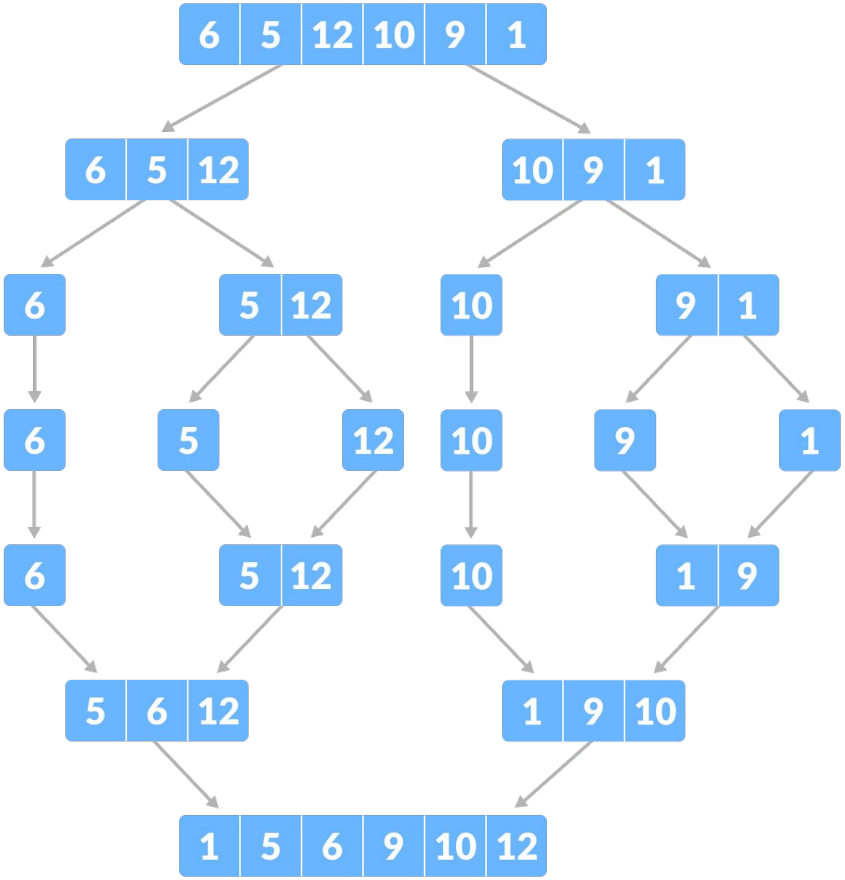
```
void mergesort(auto &v, auto begin, auto end)
{
    if (end - begin <= 1) {
        return;
    }

    auto mid = begin + (end - begin) / 2;

    mergesort(v, begin, mid);
    mergesort(v, mid, end);

    merge(v, begin, mid, end);
}
```

Merge Sort


6 5 12 10 9 1
6 5 12
10 9 1
6
5 12
10
9 1
6
5
12
10
9
1
6
5 12
10
1 9
5 6 12
1 9 10
1 5 6 9 10 12

```cpp
void merge(auto &v, auto begin, auto mid, auto end)
{
    vector<int> aux(end - begin);

    auto i = begin, j = mid, k = 0;

    while (i < mid && j < end) {
        if (v[i] <= v[j]) {
            aux[k] = v[i++];
        } else {
            aux[k] = v[j++];
        }
        k++;
    }
    while (i < mid) {
        aux[k++] = v[i++];
    }
    while (j < end) {
        aux[k++] = v[j++];
    }
    for (auto i = 0; i < aux.size(); i++) {
        v[begin + i] = aux[i];
    }
}
```

Relação de Recorrência

- Algoritmos recursivos, possuem uma maneira específica de se calcular sua complexidade, chamada de **relação de recorrência**
  - Relação de recorrência é uma equação ou inequação que descreve uma função em termos dela mesma considerando entradas menores
  - Nele, você define uma função T, e a resolução dessa função é escrita em termos de verificação do valor

Relação de Recorrência

- Para exemplificar, a função fatorial possui a seguinte relação de recorrência

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 1 \\ T(n-1), & \text{se } n > 1 \end{cases}$$

- Nesse caso, é fácil perceber que a complexidade final é O(n)

Relação de Recorrência - Merge Sort

- Relações de recorrência podem ficar mais complexas, a media que particionam o problema, ou pelo perfil de chamada da função recursiva
- No caso do Merge Sort, a relação de recorrência é a seguinte

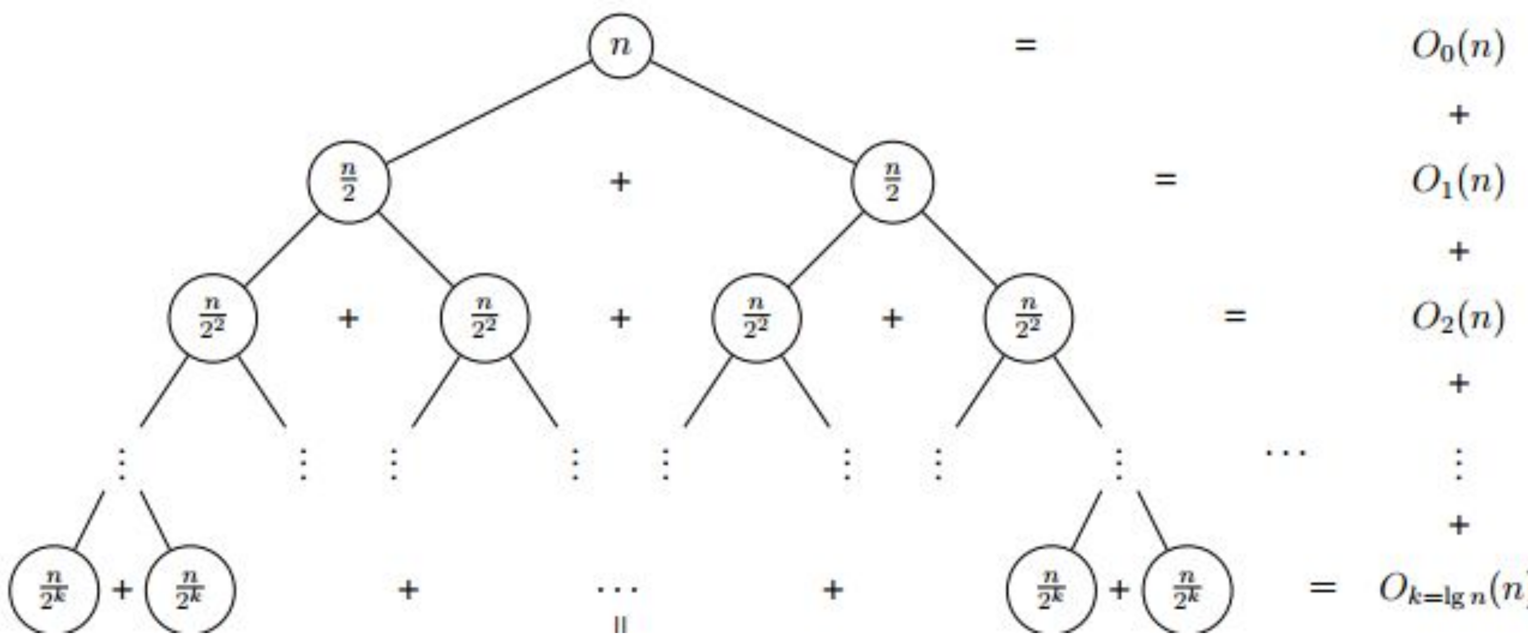
$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 1 \\ 2T(\frac{n}{2}) + \Theta(n), & \text{se } n > 1 \end{cases}$$

Relação de Recorrência - Merge Sort

- Existem algumas maneiras diferentes de se calcular a complexidade de tempo, a partir da relação de recorrência, a mais simples delas é desenhar uma árvore de recursão, para as chamadas da função recursiva

Relação de Recorrência - Merge Sort

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



Merge Sort - Características

- Assim, o Merge Sort é um algoritmo que possui complexidade de tempo linearítmica, ou seja O(n lg(n))
- Como ele precisa de um espaço auxiliar na função de merge, este possui complexidade de espaço linear, ou seja O(n)
- Além disso, ele é um algoritmo **estável**

Quick Sort

Quick Sort

- Embora o Merge Sort seja um algoritmo que atinja um limite inferior O(n lg(n)) para algoritmos de ordenação baseados em comparações, ele demanda uma memória adicional O(n), não sendo portanto um algoritmo in-place
- A ideia do QuickSort é aproveitar uma ideia similar da divisão do vetor em subvetores menores, como é feito no Merge Sort, porém de maneira in-place

Quick Sort

- Assim, a divisão do vetor não será posicional, mas sim baseada no valor de um elemento escolhido arbitrariamente, denominado pivô
- O pivô permite um rearranjo dos elementos usando a própria memória do vetor, tornando o algoritmo in-place
- Embora o QuickSort tenha complexidade média $O(n \lg(n))$, no pior caso ele pode se degenerar para $O(n^2)$

Quick Sort - Pivoteamento

- Pivoteamento é o processo de reposicionamento dos elementos do vetor de acordo com o valor x do elemento pivô que ocupa o índice p
- Ao final do pivoteamento, todos elementos com valores menores que x estarão à esquerda do pivô, e os demais à direita

Quick Sort - Pivoteamento

- O pivô já estará na posição correta em relação ao ordenamento global, de modo que o Quick Sort pode prosseguir recursivamente nas duas partes separadas pelo pivô
- Para simplificar a rotina, no início do pivoteamento o pivô troca de posição com o primeiro elemento do vetor
- Ao final, o pivô se move para a posição adequada e esta posição é retornada

Quick Sort - Pivoteamento

- A escolha do pivô pode ser feita de diferentes maneira:
  - Primeiro elemento
  - Elemento do meio
  - Último elemento
  - Elemento aleatório
- Para evitar o pior caso, a escolha do pivô costuma ser aleatória entre todos os índices possíveis

Quick Sort - Pivoteamento

- Vamos supor a seguinte lista inicial de elementos, escolhendo-se o último elemento como o pivô

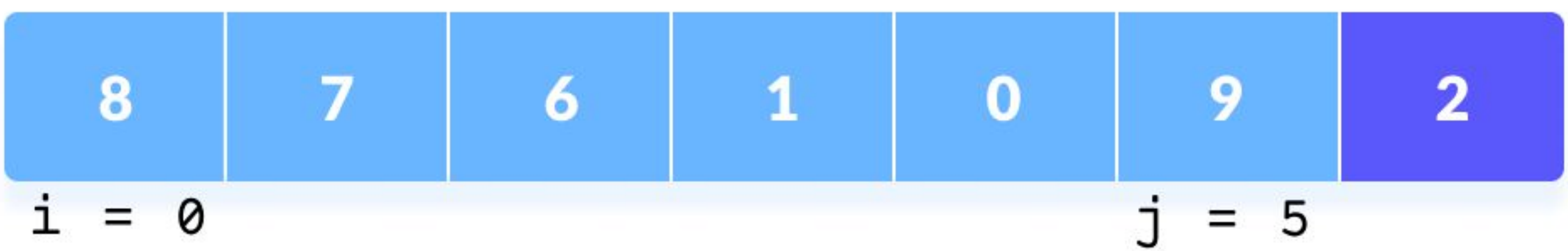
8	7	6	1	0	9	2

- O primeiro passo do algoritmo é posicionar os elementos menores que o pivô à esquerda do mesmo, e os maiores à direita

0	1	2	7	8	9	6

Quick Sort - Pivoteamento

- O passo a passo para esse reposicionamento é o seguinte:
  - Seleciona-se o pivô (por exemplo, a última posição)
  - Iniciam-se variáveis (i e j) apontando para a primeira e para a última posição (antes do pivô)



Quick Sort - Pivoteamento

- O passo a passo para esse reposicionamento é o seguinte:
  - Enquanto i <= j, faz-se o seguinte:
    - Se o valor na posição i for maior que o valor na posição j, troca-se os elementos
    - Senão se o valor na posição i for maior, j--
    - Senão se o valor na posição j for menor, i++
    - Senão, i++ e j--

Quick Sort - Pivoteamento

- O passo a passo para esse reposicionamento é o seguinte:
  - Ao sair do laço, troca-se o elemento da última posição, com o elemento da posição i
    - Este elemento está na posição correta da lista ordenada

Quick Sort

- O algoritmo do quick sort completo, funciona da seguinte forma:
    - Aplica-se o passo do pivoteamento
    - Em seguida, aplica-se o quick sort na lista restante do lado esquerdo do pivô
    - Por útlimo, aplica-se o quick sort na lista restante ao lado direito do pivô

Quick Sort - Pivoteamento


```
quicksort(L, inicio, i)    quicksort(L, i + 1, final)
```

```
void quicksort(auto &L, auto begin, auto end)
{
    if (end - begin <= 1) {
        return;
    }

    auto pivot = L[end - 1];
    auto i = begin, j = end - 2;

    while (i <= j) {
        if (L[i] > pivot && L[j] < pivot) {
            swap(L[i++], L[j--]);
        } else if (L[i] > pivot) {
            j--;
        } else if (L[j] < pivot) {
            i++;
        } else {
            i++;
            j--;
        }
    }
    swap(L[i], L[end - 1]);

    quicksort(L, begin, i);
    quicksort(L, i + 1, end);
}
```

Quick Sort

- No caso do Quick Sort, a complexidade depende da escolha do pivô:
  - Se o pivô escolhido (por sorte) estiver sempre na metade do vetor, a complexidade será linearítmica ($O(n \log(n))$
  - Porém, se por azar, o pivô escolhido sempre residir em uma das extremidades, a complexidade do algoritmo é quadrática ($O(n^2)$)

Quick Sort

- Ele não é um algoritmo estável, porém não possui espaço linear como o Merge Sort
    - Então, é bastante utilizado quando estabilidade não é um requerimento
- Além disso, diversas estratégias existem para melhorar o desempenho dele por meio de escolhas melhoradas para o pivô
    - Aleatório, multiplos pivôs, etc

Ordenação em C e C++

Ordenação em C

- A biblioteca stdlib.h da linguagem C contém a função qsort(), a qual implementa o algoritmo quicksort
- A assinatura da função qsort() é:

```
void qsort(void *base, size_t nmemb, size_t size, int
(*compar)(const void *, const void *));
```

Ordenação em C

- O último parâmetro da função qsort é um ponteiro para uma função de comparação
  - Esta função deve receber dois ponteiros constantes a e b do tipo void *
  - O retorno deve ser um número inteiro que representa a relação entre os ponteiros:
    - zero, se a e b são iguais
    - negativo, se a é menor do que b
    - positivo, se a é maior do que b

Ordenação em C

```
int compar(const void *a, const void *b)
{
    int *pa = (int *)a;
    int *pb = (int *)b;
    return *pa - *pb;
}
```

Ordenação em C

```c
int L[] = {3, 1, 4, 1, 5, 9, 2, 6, 3, 0};
int n = 10;

qsort(L, n, sizeof(int), compar);
```

Ordenação em C++

- A biblioteca algorithm da linguagem C++ oferece quatro algoritmos de ordenação: sort(), stable_sort(), partial_sort() e nth_element()
  - As interfaces são similares, com diferença apenas no funcionamento
  - A função de comparação é binária e deve retornar verdadeiro se a precede b ou falso, caso contrário

Ordenação em C++

- A biblioteca algorithm da linguagem C++ oferece quatro algoritmos de ordenação: sort(), stable_sort(), partial_sort() e nth_element()
  - Se a função de comparação for omitida, a comparação é feita utilizando o operador < (se for possível, senão retorna erro)
  - A complexidade dos três primeiros algoritmos é O(n log(n)), enquanto do último tem complexidade média de O(n)

Ordenação em C++

- A função sort() implementa um algoritmo de ordenação instável, in-place, com complexidade média O(n log n)
- Ao final da execução, o intervalo [first, last) estará ordenado, de acordo com o operador < ou o comparador respectivamente
- É possível customizar o critério de comparação, pela função passada (se for o caso)
- Existe uma estrutura greater para ordenar em ordem decrescente

Ordenação em C++

```
int main()
{

    vector<int> L = {8, 7, 6, 1, 0, 9, 2};

    for (auto x : L) cout << x << " "; cout << endl;

    sort(L.begin(), L.end(), greater<int>());

    for (auto x : L) cout << x << " "; cout << endl;

    return 0;
}
```

```cpp
bool compare(const string &a, const string &b)
{
    string x, y;
    auto to_lower = [](char c) {return tolower(c);};

    transform(a.begin(), a.end(), back_inserter(x), to_lower);
    transform(b.begin(), b.end(), back_inserter(y), to_lower);

    return x < y;
}

int main() {
    vector<string> L = {"verde", "amarelo", "Vermelho", "Branco", "Preto", "azul"};

    sort(L.begin(), L.end());                   // Branco Preto Vermelho amarelo azul verde
    for (auto x : L) cout << x << " "; cout << endl;

    sort(L.begin(), L.end(), compare);          // amarelo azul Branco Preto verde Vermelho
    for (auto x : L) cout << x << " "; cout << endl;

    return 0;
}
```

Ordenação em C++

- A função stable_sort() implementa um algoritmo de ordenação estável, in-place, com complexidade média O(n log n)
- Ao contrário da função sort(), a função stable_sort() preserva a ordem relativa dos elementos considerados iguais

Ordenação em C++

```
int main()
{
    vector<double> L1 = {2.7, 2.2, 1.3, 1.8, 1.1, 3.2, 2.9};
    auto L2 = L1;
    auto cmp = [](double a, double b) { return int(a) < int(b);};

    sort(L1.begin(), L1.end(), cmp);                // 1.3 1.8 1.1 2.7 2.2 2.9 3.2
    for (auto x : L1) cout << x << " "; cout << endl;

    stable_sort(L2.begin(), L2.end(), cmp);         // 1.3 1.8 1.1 2.7 2.2 2.9 3.2
    for (auto x : L2) cout << x << " "; cout << endl;
    return 0;
}
```

Ordenação em C++

- Já a função partial_sort() tem assinaturas ligeiramente diferentes das duas funções citadas anteriormente, por contar com um terceiro parâmetro, e tem um funcionamento similar a partition do quicksort
- Por útlimo, a função nth_element() posiciona corretamente apenas o elemento que ocuparia a n-ésima posição do vetor, caso estivesse ordenado

Ordenação em C++

```
int mediana(auto &L)
{
    nth_element(L.begin(), L.begin() + L.size() / 2, L.end());
    return L[L.size() / 2];
}

int main()
{
    vector<int> L = {6, 7, 8, 1, 0, 9, 2};
    cout << mediana(L) << endl;                  // 6

    return 0;
}
```

Conclusão