# Técnicas de Programação

## Algoritmos de Ordenação

Fábio Duncan de Souza

Instituto Federal Fluminense

# Sumário

# Conceitos Básicos

# Ordenação



- Em programação é comum a necessidade de ordenar dados.
- Para ordenar, o algoritmo precisa saber qual é a chave de ordenação, a partir dai ele manipula os elementos a ordenar, considerando como elemento de comparação a chave que lhe for dada.
- Exemplos:
  - Ordenar nomes por ordem alfabética crescente ou decrescente.
  - Ordenar de números por ordem crescem ou decrescente.
  - Ordenar nomes por ordem crescente ou decrescente do seu comprimento.

# Tipos de Algoritmos

- Se o volume de dados não for muito grande, não há problema de se escolher um algoritmo de ordenação mais simples ou um mais eficaz.
- Quando o volume de dados é reduzido o tempo de execução de um algoritmo de ordenação mais elementar é igual ou inferior ao tempo de execução de um algoritmo mais avançado de ordenação.
- Algoritmos mais avançados executam mais instruções e utilizam mais estruturas de dados auxiliares.
- Com o aumento do volume dos dados, a manipulação das estruturas de dados revela a sua utilidade e a sua contribuição para uma ordenação mais rápida.

# Complexidade de Algoritmos

# Análise de Eficiência

- A eficiência de tempo é calculada pelo número de operações críticas efetuadas.
  - número de comparações;
  - movimentação de registros ou de ponteiros para registros;
  - troca de dois registros.

# Análise Assintótica



- É comum que na análise de algoritmos os valores pequenos sejam ignorados e a avaliação se concentre nos valores enormes de n, onde n varia em função da entrada.
- A matemática que se interessa apenas pelos valores enormes de n é chamada assintótica.
  - Para valores enormes de $n$, as funções $n^2$, $999n^2$, $n^2 + 100n$, etc, têm todas a mesma taxa de crescimento e portanto são todas equivalentes.

# Notação BIG O

- Big O é uma forma de classificar quanto a uma função ou algoritmo é escalável.
- Nos ajuda a determinar qual será o comportamento do nosso algoritmo no PIOR dos casos ou quanto tempo ele levará para ser completado baseado nos seus valores de entrada.
- Ou seja, qual será a performance do código se o número de valores de entrada aumentar?
  - Constante
  - Linear
  - Exponencial
  - Logaritmo

# Big O



# Bubble Sort

# Bubble Sort

- Algoritmo de ordenação simples.
- O seu funcionamento é baseado na troca de elementos adjacentes que não estejam em ordem entre si.
- No melhor caso, o algoritmo executa $n$ operações relevantes, onde $n$ representa o número de elementos do vetor.
  - Isso ocorre quando o vetor já está inteiramente ordenado;
  - Então é necessário apenas uma verificação básica sobre todo o vetor, o que representa um custo de O(n).
- No pior caso, são feitas $n^2$ operações.
- A complexidade é $O(n^2)$, por isso não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados.

# Bubble Sort - Funcionamento



1. O Bubble Sort começa no final do vetor, e vai trocando todos os elementos adjacentes que não estejam em ordem entre si, até chegar à primeira posição do vetor.
   No final da 1º passagem o menor elemento estará na primeira posição do vetor.
2. O algoritmo vai novamente fazer trocas com elementos adjacentes, começando no último elemento do vetor, mas terminando agora, na segunda posição do vetor.
3. O processo se repete, começando sempre no ultimo elemento do vetor e, acabando uma posição á frente da posição em que tinha parado na passagem anterior.
4. O Bubble Sort faz n-1 passagem pelos dados. Onde o n é o numero de elementos para ordenar.

# Selection Sort

# Selection Sort

- É considerado um algoritmo de ordenação simples
  - Não usa estruturas de dados auxiliares;
  - Bastante intuitivo.
- O selection sort compara a cada interação um elemento com os outros, visando encontrar o menor.
- Não existe um melhor caso mesmo que o vetor esteja ordenado ou em ordem inversa serão executados os dois laços do algoritmo, o externo e o interno.
- A complexidade deste algoritmo será sempre $O(n^2)$.

# Selection Sort - Funcionamento

1. Pesquisa o menor elemento e troca o mesmo com o elemento da 1º posição;
2. Pesquisa o segundo menor elemento e troca com o elemento da segunda posição;
3. O processo se repete até a finalização da ordenação.

# Insertion Sort

# Insertion Sort

- É considerado um algoritmo de ordenação simples.
- No melhor caso, o algoritmo executa $n$ operações relevantes, onde $n$ representa o número de elementos do vetor.
  - Isso ocorre quando o vetor já está inteiramente ordenado;
  - Então é necessário apenas uma verificação básica sobre todo o vetor, o que representa um custo de O(n).
- No pior caso, são feitas $O(n^2)$ operações.
  - Ocorre quando o vetor está ordenado na ordem inversa.

# Insertion Sort - Funcionamento

- O seu funcionamento se assemelha a organização das cartas de um baralho na mão do jogador.
  - Percorra as posições do array, começando com o índice 1 (um).
  - Cada nova posição é como a nova carta recebida, e você precisa inseri-la no lugar correto no subarray ordenado à esquerda daquela posição.
- Funcionamento:
  1. Os elementos são divididos em uma sequência de destino $a_1, ..., a_{i-1}$ e uma sequência de origem $a_i, ..., a_n$.
  2. Em cada passo, a partir de $i = 2$, o i-ésimo item da sequência de origem é retirado e transferido par a sequência destino, sendo inserido na posição adequada.

# Comparativo de Métodos de Ordenação Simples

# Comparativo de Métodos de Ordenação Simples



| Algoritmo | Complexidade | | |
|---|---|---|---|
| | Melhor | Médio | Pior |
| Insertion sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ |
| Bubble sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ |
| Selection sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |

# QuickSort



- É um algoritmo avançado de ordenação;
- Inventado nos anos 60;
- É rápido e eficiente;
- Possui natureza recursiva.

# QuickSort

## Funcionamento

1. É escolhido um pivô aleatoriamente;
2. Os dados menores do que o pivô serão colocados a esquerda deste, enquanto os dados maiores serão colocados a direita;
3. O algoritmo coloca o pivô na sua posição final de ordenação;
4. Depois o algoritmo parte os dados em dois;
5. Em seguida o algoritmo vai fazer o mesmo para cada uma das duas partes:
   - Escolher o pivô;
   - Colocar o pivô na posição correta;
   - Partir novamente os dados em dois.
6. O processo se repete até ficar com vetores com apenas um elemento;
7. Neste momento o vetor inicial estará completamente ordenado.

- Pior Caso
  - Ocorre quando o quicksort tem as partições o mais desequilibradas possível;
  - A chamada original possui uma complexidade de processamento igual ao tamanho da entrada do problema, isto é , $n$;
  - A chamada recursiva em $n-1$ elementos possui complexidade $n-1$, a chamada recursiva em $n-2$ elementos possui complexidade $n-2$, e assim por diante;
  - Após realizar a soma das complexidades das chamadas chega-se a complexidade computacional $O(n^2)$;

# QuickSort

## Complexidade

- Melhor Caso
  - Ocorre quando as partições são tão equilibradas quanto possível;
  - Os tamanhos das partições são iguais ou possuem diferença de 1 elemento;
  - O primeiro caso ocorre quando o vetor tem um número ímpar de elementos e o pivô está bem no meio após o particionamento, e cada partição tem $(n-1)/2$ elementos;
  - Os casos posteriores ocorrem quando o subvetor tem um número par de elementos n, que formam uma partição de $n/2$ elementos e uma outra com $(n/2)-1$ elementos;
  - A complexidade neste caso é $n \log_2 n$.

# QuickSort

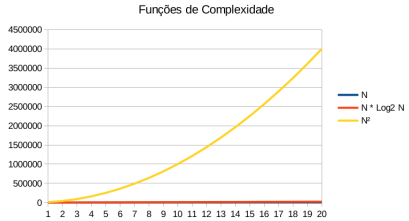
## Complexidade

- Caso Médio
  - A complexidade neste caso é $n \log_2 n$.
  - Obs: A demonstração matemática para este caso será vista na disciplina de estruturas de dados do próximo período.

# Comparativo das Funções de Complexidade dos Métodos



| Dados de Entrada | N | N * $Log_2$ N | $N^2$ |
|---|---|---|---|
| 100 | 100 | 664,385619 | 10000 |
| 200 | 200 | 1528,771238 | 40000 |
| 300 | 300 | 2468,645607 | 90000 |
| 400 | 400 | 3457,542476 | 160000 |
| 500 | 500 | 4482,892142 | 250000 |
| 600 | 600 | 5537,291214 | 360000 |
| 700 | 700 | 6615,847778 | 490000 |
| 800 | 800 | 7715,084952 | 640000 |
| 900 | 900 | 8832,403072 | 810000 |
| 1000 | 1000 | 9965,784285 | 1000000 |
| 1100 | 1100 | 11113,61659 | 1210000 |
| 1200 | 1200 | 12274,58243 | 1440000 |
| 1300 | 1300 | 13447,58468 | 1690000 |
| 1400 | 1400 | 14631,69556 | 1960000 |
| 1500 | 1500 | 15826,12018 | 2250000 |
| 1600 | 1600 | 17030,1699 | 2560000 |
| 1700 | 1700 | 18243,24235 | 2890000 |
| 1800 | 1800 | 19464,80614 | 3240000 |
| 1900 | 1900 | 20694,38904 | 3610000 |
| 2000 | 2000 | 21931,56857 | 4000000 |



# Referências Bibliográficas


Ana Fernanda Gomes Ascencio and Edilene Aparecida Veneruchi de Campos.
*Fundamentos da programação de computadores*.
Pearson Educación, 2008.


Harvey M Deitel, Paul J Deitel, and Edson Furmankiewicz.
*Java: como programar*.
Pearson educacion, 2017.