ALGORITMOS DE ORDENAÇÃO

Hugo Veríssimo 124348

João Cardoso 50750

universidade de aveiro

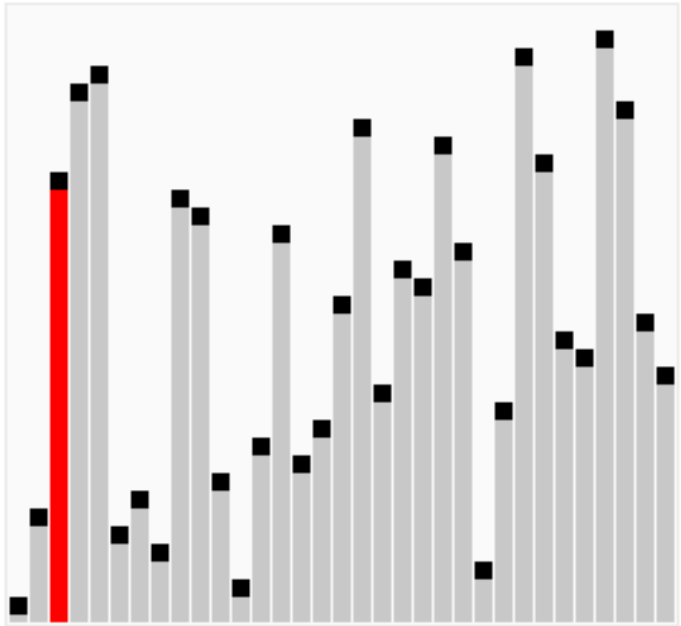
ENQUADRAMENTO

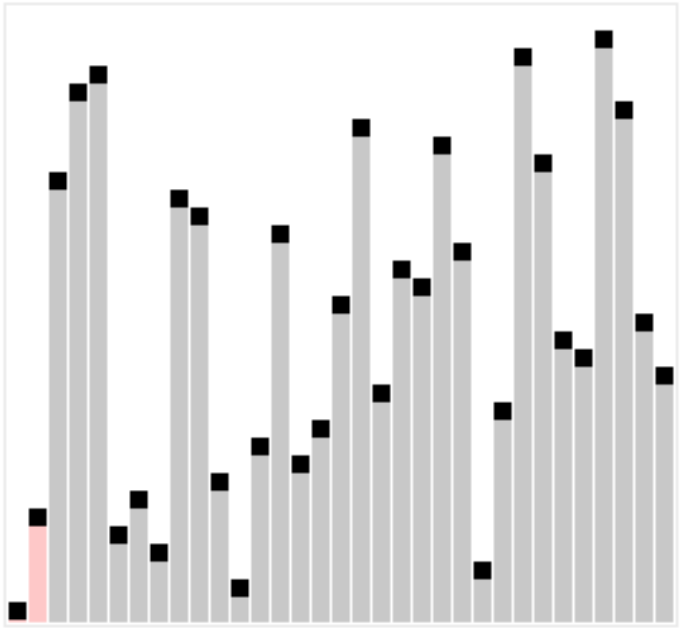
- Ordenação é fundamental em algoritmos de otimização e processamento eficiente de dados
- Usada para melhorar desempenho em problemas como Knapsack e Kruskal, em algoritmos gulosos, etc.
- Este trabalho analisa 9 algoritmos de ordenação:
  - Baseados em troca, seleção, inserção, divisão-e-conquista e distribuição
- Objetivo: comparar funcionamento, complexidade, vantagens/limitações e desempenho prático

BUBBLE SORT

- Algoritmo de ordenação por comparação
- Percorre repetidamente a lista
- Compara elementos adjacentes e troca se estiverem fora de ordem
- Repete o processo até a lista estar ordenada

$$\begin{cases} \mathcal{O}(n) & \text{melhor caso (lista já ordenada)} \\ \mathcal{O}(n^2) & \text{pior caso (lista em ordem inversa)} \\ \mathcal{O}(n^2) & \text{caso médio} \end{cases}$$





SELECTION SORT

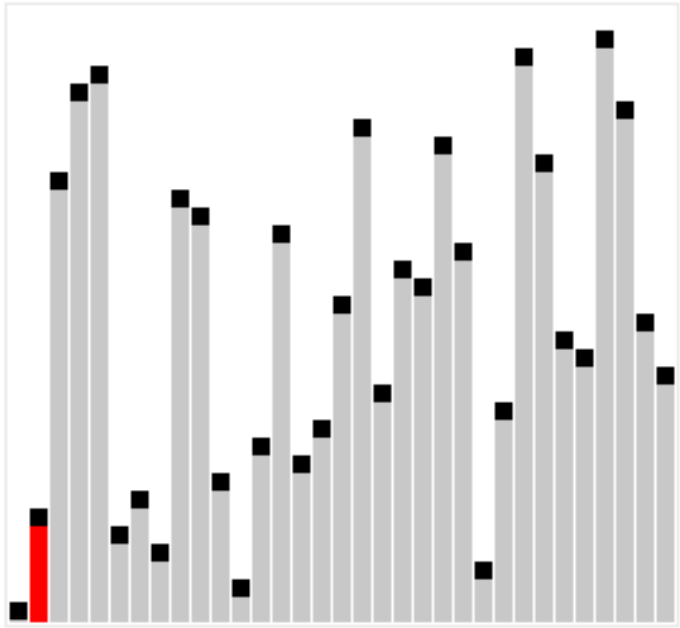
- Percorre a lista à procura do menor elemento
- Coloca o menor elemento na primeira posição
- Continua até todos os elementos estarem ordenados

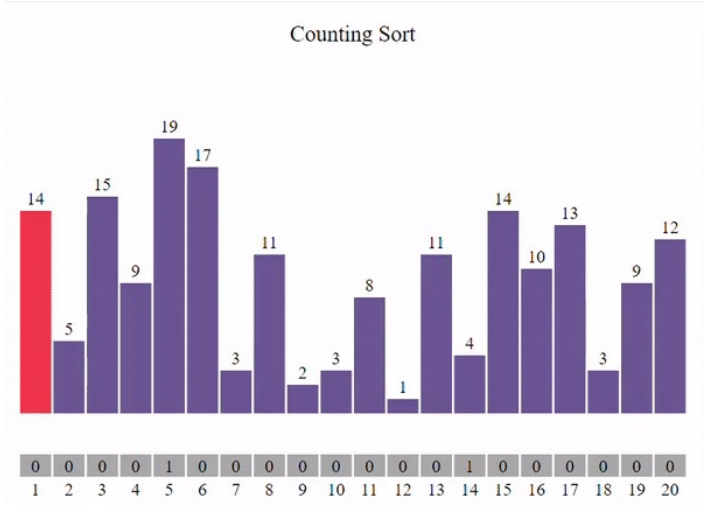
$$\left\{ \mathcal{O}(n^2) \quad \text{em todos os casos} \right.$$

INSERTION SORT

- Algoritmo de ordenação por inserção
- Percorre a lista da esquerda para a direita
- Para cada elemento, insere-o na posição correta na parte já ordenada
- A parte à esquerda do cursor está sempre ordenada

$$\begin{cases} \mathcal{O}(n) & \text{melhor caso (lista ordenada)} \\ \mathcal{O}(n^2) & \text{pior caso (lista inversamente ordenada)} \\ \mathcal{O}(n^2) & \text{caso médio} \end{cases}$$





COUNTING SORT

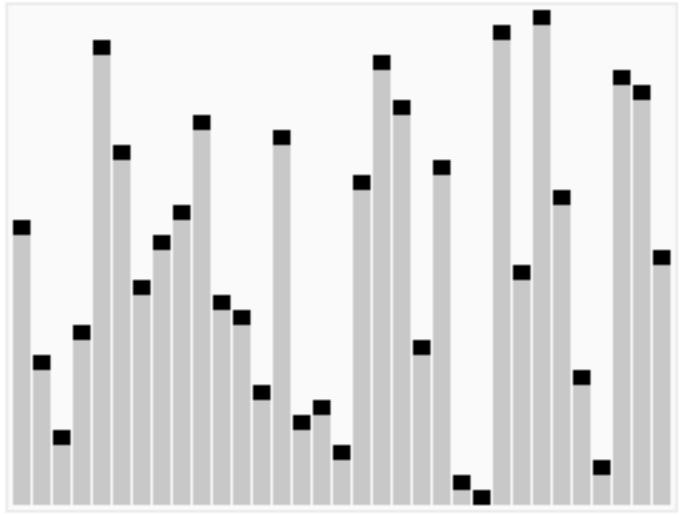
- Algoritmo de ordenação por contagem de frequências
- Não usa comparações
- Cria um array auxiliar para contar ocorrências de cada valor
- Reconstrói a lista ordenada com base nessas contagens
- Requer que os elementos sejam inteiros não negativos e num intervalo conhecido e limitado

$$\begin{cases} \mathcal{O}(n+k) & \text{em todos os casos} \end{cases}$$

RADIX SORT

- Algoritmo de ordenação por distribuição de dígitos
- Ordena os elementos dígito a dígito (começando pelo menos significativo)
- Usa um algoritmo de ordenação estável (ex: Counting Sort) em cada passo
- Evita comparações diretas entre elementos
- Ideal para inteiros ou strings de comprimento fixo ou conhecido

$$\begin{cases} \mathcal{O}(d \cdot n) & \text{em todos os casos} \end{cases}$$



QUICK SORT

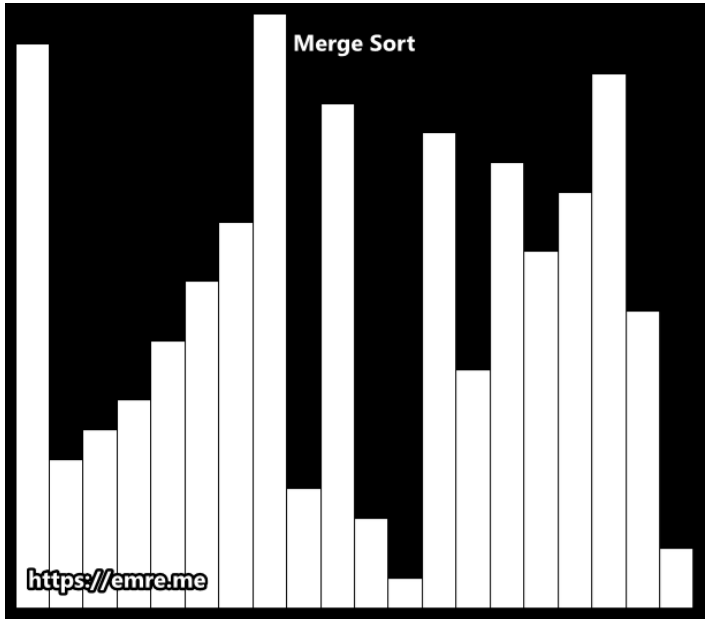
- Algoritmo de ordenação por divisão e Conquista
- Escolhe um elemento como pivô
- Reorganiza a lista colocando menores à esquerda e maiores à direita do pivô
- Aplica o mesmo processo recursivamente às sublistas

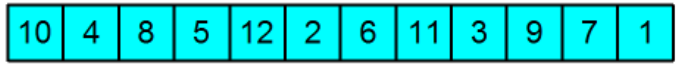
$$\begin{cases} \mathcal{O}(n \log n) & \text{melhor caso (divisões equilibradas)} \\ \mathcal{O}(n^2) & \text{pior caso (divisões desequilibradas)} \\ \mathcal{O}(n \log n) & \text{caso médio (pivôs aleatórios ou bons)} \end{cases}$$

MERGE SORT

- Algoritmo de ordenação por divisão e conquista
- Divide recursivamente a lista em metades até listas unitárias
- Combina (merge) as sublistas ordenadas em ordem crescente

$$\left\{ \mathcal{O}(n \log n) \quad \text{em todos os casos} \right.$$





HEAP SORT

- Algoritmo de ordenação baseado em estruturas de heap
- Constrói um heap a partir da lista
- Repetidamente extrai o maior elemento e coloca no final
- Garante ordenação ao reorganizar o heap após cada extração

$$\left\{ \mathcal{O}(n \log n) \quad \text{em todos os casos} \right.$$

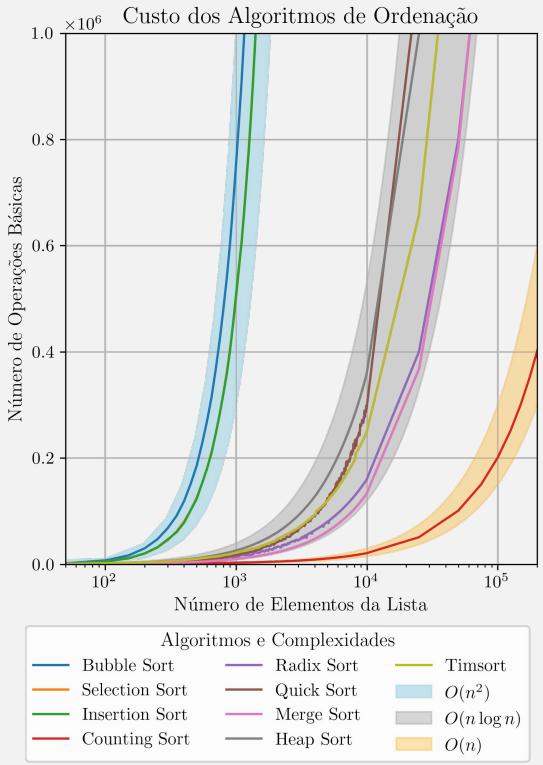
TIMSORT

- Algoritmo híbrido de ordenação, baseado em Merge Sort e Insertion Sort
- Divide a lista em pequenas sequências já parcialmente ordenadas (runs)
- Usa Insertion Sort para ordenar pequenas runs
- Junta as runs ordenadas com Merge Sort
- Otimizado para dados reais com padrões parciais
- Estável, eficiente e usado em implementações padrão (ex: Python, Java)

$$\begin{cases} \mathcal{O}(n) & \text{melhor caso (lista parcialmente ordenada)} \\ \mathcal{O}(n \log n) & \text{pior caso (dados desordenados)} \\ \mathcal{O}(n \log n) & \text{caso médio} \end{cases}$$

RESULTADOS

Algoritmo	Complexidade
Bubble Sort	$\mathcal{O}(n^2)$
Selection Sort	$\mathcal{O}(n^2)$
Insertion Sort	$\mathcal{O}(n^2)$
Counting Sort	$\mathcal{O}(n+k)$
Radix Sort	$\mathcal{O}(n \times d)$
Quick Sort	$\mathcal{O}(n \log n)$
Merge Sort	$\mathcal{O}(n \log n)$
Heap Sort	$\mathcal{O}(n \log n)$
Timsort	$\mathcal{O}(n \log n)$



- **Counting Sort** apresenta complexidade $\mathcal{O}(n)$, dado que $k = 1000 \ll 10^6$.
- **Bubble**, **Selection** e **Insertion Sort** apresentam um crescimento quadrático no seu número de operações básicas ($\mathcal{O}(n^2)$), como esperado.
- **Quick**, **Merge**, **Heap** e **Timsort** apresentam um crescimento compatível com $\mathcal{O}(n \log n)$.
- **Radix Sort** tem complexidade teórica $\mathcal{O}(nd) \approx \mathcal{O}(n \log k)$, acabando por se aproximar dos algoritmos de complexidade $\mathcal{O}(n \log n)$.

CONCLUSÃO

- Análise empírica confirmou previsões teóricas

- Escolha do algoritmo depende:
  - Complexidade teórica
  - Características concretas dos dados
  - Contexto de utilização e requisitos práticos de desempenho e memória