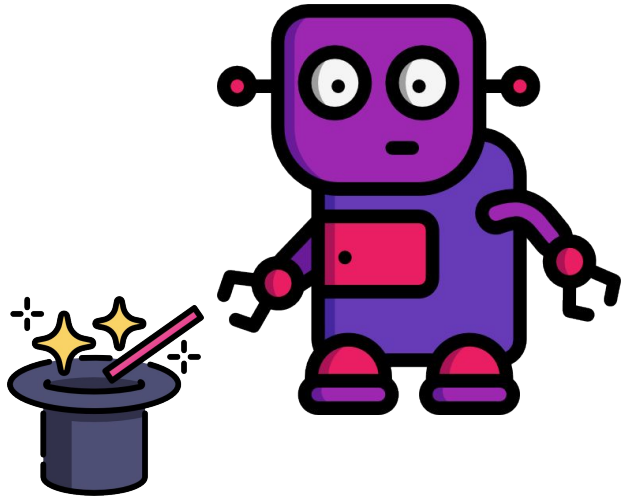
UNIFESP

Programa de Pós Graduação em Ciências da Computação
Análise de Algoritmos e Estruturas de Dados - Profa. Dra. Lilian Berton

Análise: Learned Sort



Willian Dihanster Gomes de Oliveira

12 de Abril de 2021
São José dos Campos - SP

Introdução

- Uso de **algoritmos inteligentes** cresceu muito nos últimos anos.
- **Resultados interessantes** para diversas aplicações.
- Podemos **aplicar em ordenação**?

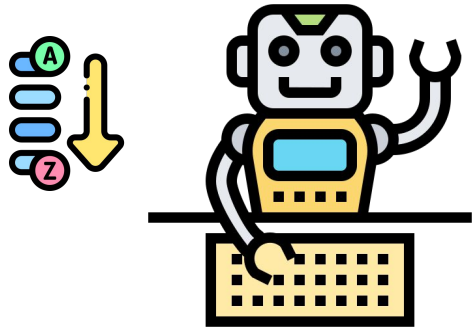


Figura 1: Ilustração de um robô fazendo uma tarefa de ordenação.

Ordenação + Algoritmos Inteligentes?

- Modelo **sabe** onde **colocar o elemento** atual.

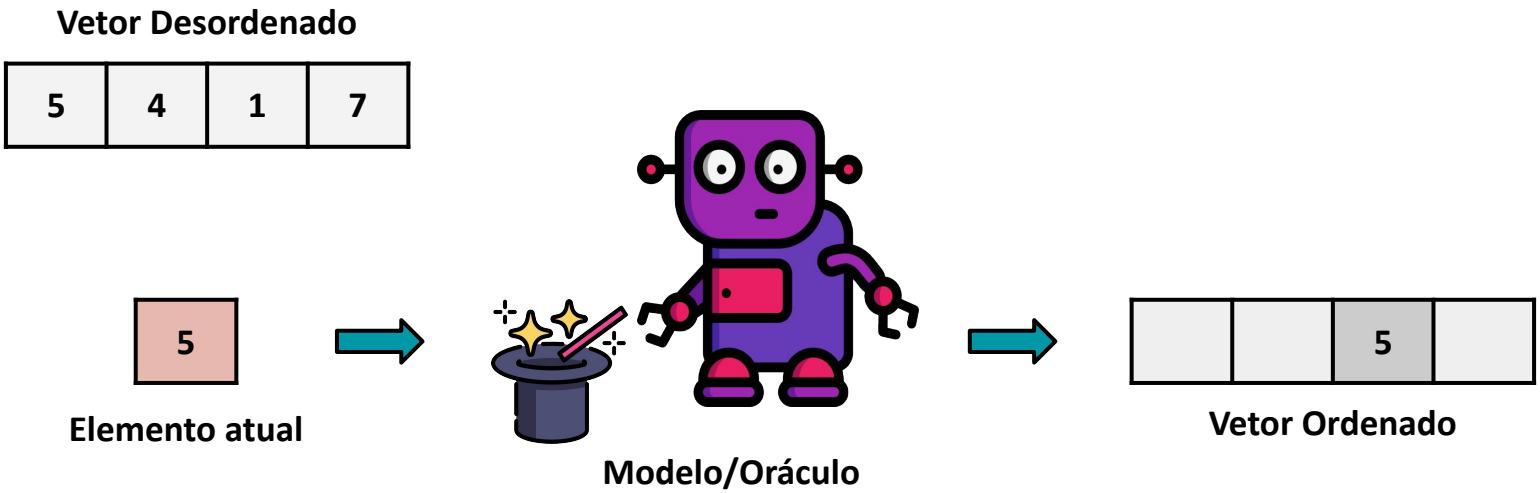


Figura 2: Esquema prático do uso de um algoritmo inteligente em ordenação.

Learned Sort v1.0 (KRISTO, Ani et al., 2020) [1]

- Supõe **padrão na distribuição** dos valores de entrada.
- Modelo **CDF (*Cumulative Density Function*)**.
  - Treina apenas com amostra para gerar aproximação;
  - Inferência para cada elemento **P(X ≤ x)**.

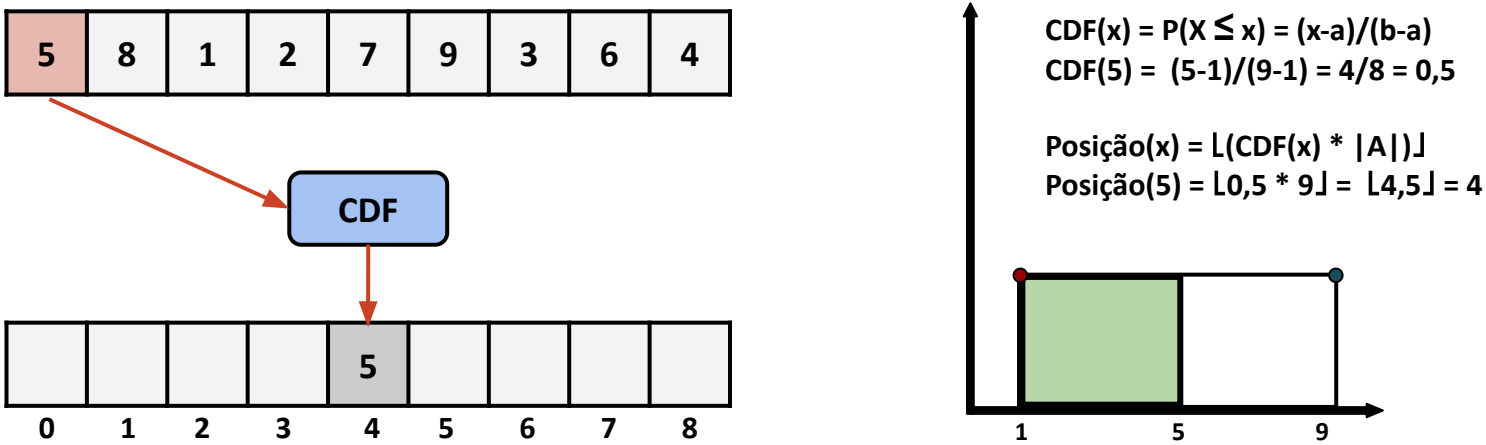


Figura 3: Esquema prático de aplicação do modelo CDF e cálculo de uma CDF.

Learned Sort v1.0 - Colisão

- **Linear:** coloca na **próxima posição livre**;
- **Encadeamento:** cria um **encadeamento na posição** que gerou colisão;
- ***Spill Bucket*:** aloca os valores em um ***bucket* especial** separado.

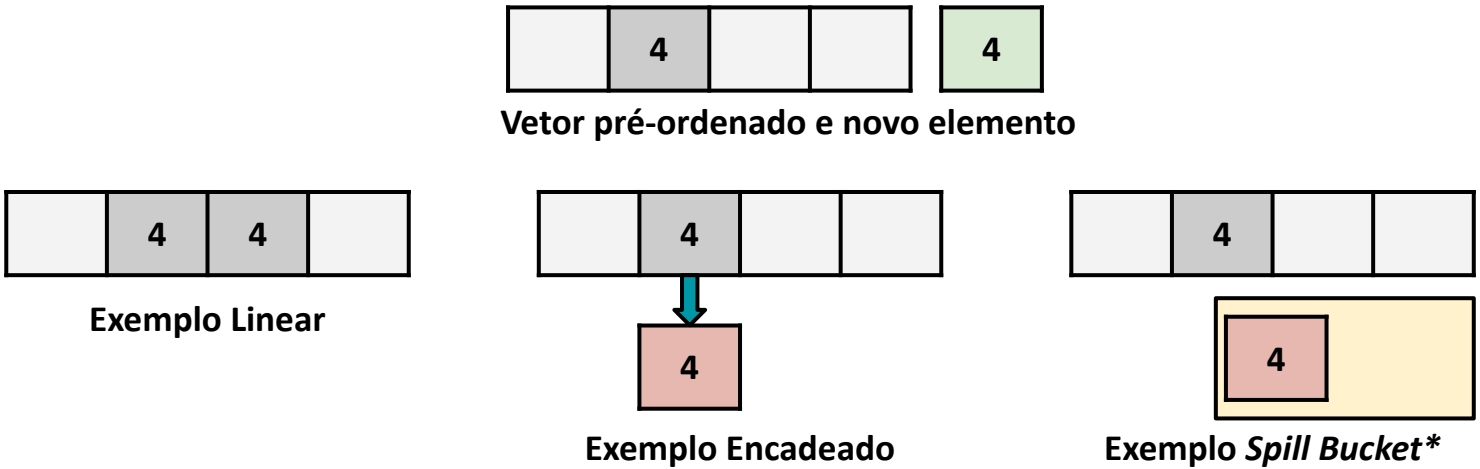


Figura 4: Esquema de tratamentos de colisões.

Learned Sort v1.0 - Imprecisão do Modelo

- Modelo pode **não acertar tudo**.
- Passada final com **Insertion Sort**.
- É esperado **pouca confusão**.

Figura 5: Esquema da passada final do algoritmo.

Learned Sort v1.0 - Algoritmo e Complexidade

Algorithm 1 A first Learned Sort

Input A - the array to be sorted
Input F_A - the CDF model for the distribution of A
Input o - the over-allocation rate. Default=1
Output A' - the sorted version of array A

```
1: procedure LEARNED-SORT(A, F_A, o)
2:     N ← A.length
3:     A' ← empty array of size (N · o)
4:     for x in A do                                   O(n)
5:         pos ← ⌊F_A(x) · N · o⌋                      O(1)
6:         if EMPTY(A'[pos]) then A'[pos] ← x          O(1)
7:         else COLLISION-HANDLER(x)                   O(m)
8:     if o > 1 then COMPACT(A')                       O(n)
9:     if NON-MONOTONIC then INSERTION-SORT(A')        O(n) ou O(n^2)
10:    return A'
```

Depende da qualidade do modelo!

Sem colisões e ordenação quase perfeita ≃ O(n)

O(n*m + n + n) = O(n*m)
ou
O(n*m + n + n^2) = O(n^2)

Figura 6: Pseudocódigo do 1º algoritmo [1].

Learned Sort v2.0 - Cache-Otimizado

- Evoluções:
  - Prevê ***bucket*** **em vez de posição exata**;
  - Ordenação baseada em **Counting** e **novo modelo CDF.**

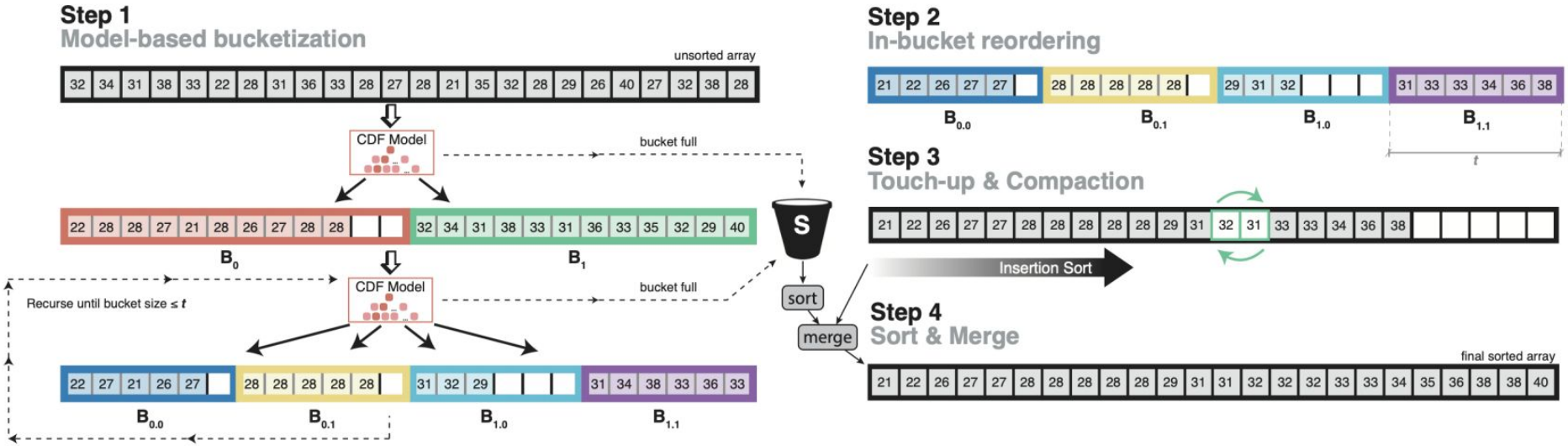


Figura 7: Esquema prático da versão 2.0 do algoritmo [1].

Learned Sort v2.0 - Novo Modelo

- **Modelo Hierárquico**
  - **“Sabe” qual** modelo pode **lidar melhor** com a chave.

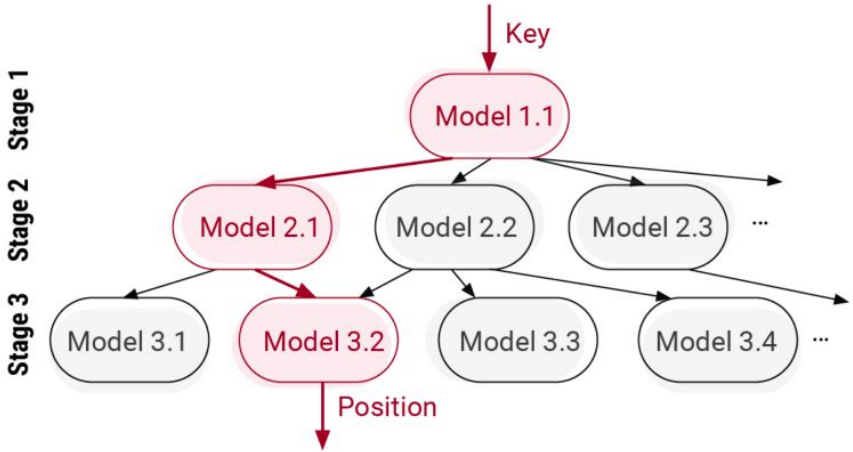


Figura 8: Esquema do 2º estimador [1].

Learned Sort v2.0 - Algoritmo e Complexidade

Algorithm 2 The Learned Sort algorithm

Input A - the array to be sorted
Input F_A - the CDF model for the distribution of A
Input f - fan-out of the algorithm
Input t - threshold for bucket size
Output A' - the sorted version of array A

```
1:  procedure LEARNED-SORT(A, F_A, f, t)
2:      N ← |A|                                   ▷ Size of the input array
3:      n ← f                                     ▷ n represents the number of buckets
4:      b ← ⌊N/f⌋                                 ▷ b represents the bucket capacity
5:      B ← [] × N                                ▷ Empty array of size N
6:      I ← [0] × n                               ▷ Records bucket sizes
7:      S ← []                                    ▷ Spill bucket
8:      read_arr ← pointer to A
9:      write_arr ← pointer to B
10:     // Stage 1: Model-based bucketization
11:     while b ≥ t do                            ▷ Until bucket capacity reaches the threshold t
12:         I ← [0] × n                           ▷ Reset array I
13:         for x ∈ read_arr do
14:             pos ← ⌊INFER(F_A, x) · n⌋
15:             if I[pos] ≥ b then                ▷ Bucket is full
16:                 S.append(x)                   ▷ Add to spill bucket
17:             else                              ▷ Write into the predicted bucket
18:                 write_arr[pos · b + I[pos]] ← x
19:                 INCREMENT I[POS]
20:         b ← ⌊b/f⌋                             ▷ Update bucket capacity
21:         n ← ⌊N/b⌋                             ▷ Update the number of buckets
22:         PTRSWP(read_arr, write_arr)           ▷ Pointer swap to reuse memory
23:     // Stage 2: In-bucket reordering
24:     offset ← 0
25:     for i ← 0 up to n do                      ▷ Process each bucket
26:         K ← [0] × b                           ▷ Array of counts
27:         for j ← 0 up to I[i] do               ▷ Record the counts of the predicted positions
28:             pos ← ⌊INFER(F_A, read_arr[offset + j]) · N⌋
29:             INCREMENT K[pos − offset]
30:         for j ← 1 up to |K| do                ▷ Calculate the running total
31:             K[j] ← K[j] + K[j − 1]
32:         T ← []                                ▷ Temporary auxiliary memory
33:         for j ← 0 up to I[i] do               ▷ Order keys w.r.t. the cumulative counts
34:             pos ← ⌊INFER(F_A, read_arr[offset + j]) · N⌋
35:             T[j] ← read_arr[offset + K[pos − offset]]
36:             DECREMENT K[pos − offset]
37:         Copy T back to read_arr[offset]
38:         offset ← offset + b
39:     // Stage 3: Touch-up
40:     INSERTION-SORT-AND-COMPACT(read_arr)
41:     // Stage 4: Sort & Merge
42:     SORT(S)
43:     A' ← MERGE(read_arr, S)
44:     return A'
```

O(l*n) = O(n)

O(n*t)

O(n*t) ou O(n^2)

O(slogs) + O(n)

O(n) ou O(n^2)

Figura 9: Pseudocódigo do 2º algoritmo [1].

Resultados

- Learned Sort foi melhor que Radix Sort somente em ordem crescente.
- Mas melhor que std::sort em todos os casos.

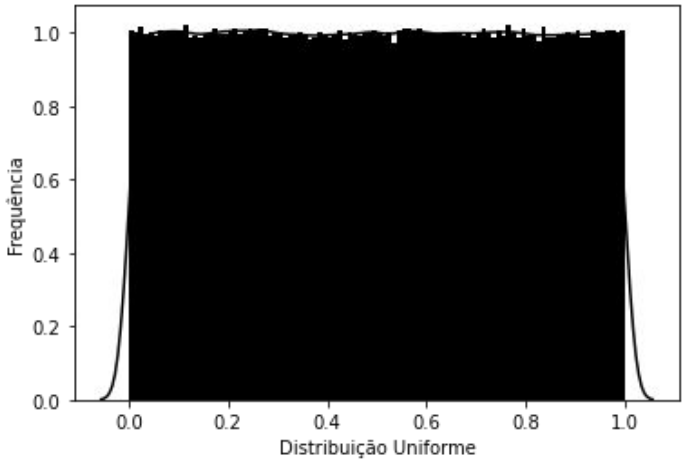


Figura 10: Distribuição Uniforme utilizada 1M com a=0 e b=1.

Algoritmo	Aleatório	Crescente	Decrescente
Learned Sort	212096	11285	230521
Radix Sort	38690	40929	38392
std::sort	377897	339338	342604

Tabela 1: Resultados obtidos (em µs).

Resultados

- Learned Sort foi melhor que Radix Sort somente em ordem crescente.
- Mas melhor que std::sort em todos os casos.

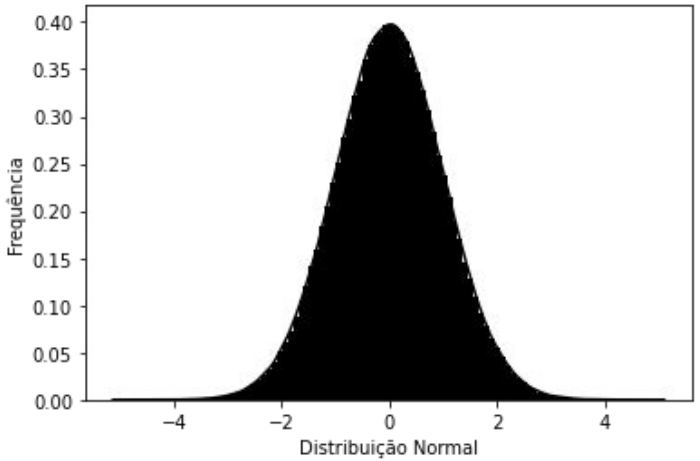


Figura 10: Distribuição Normal utilizada 1M com μ=0 e σ=1.

Algoritmo	Aleatório	Crescente	Decrescente
Learned Sort	218696	11710	219092
Radix Sort	44780	41175	39319
std::sort	347259	339811	342700

Tabela 2: Resultados obtidos (em μs).

Conclusões

- Mais **experimentos**.
  - Não foi obtido os resultados esperados. Parâmetros e ambiente computacional?
  - Autores conseguiram resultados melhores nos seguintes cenários:
    - Número de elementos <<<< K e quando o número de elementos é tão grande que não cabe na memória cache L3.
  - No algoritmo 1 mesmo com **modelo prevendo 100% não bateu RadixSort***, pois RadixSort faz **acesso sequencial** e Learned Sort faz **acesso aleatório.**

- Pode ser uma **boa opção** quando:
  - A **distribuição** dos dados é **conhecida**;
  - **Não** há muitas **duplicatas**.

Referências

- KRISTO, Ani et al. The case for a learned sorting algorithm. In: **Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data**. 2020. p. 1001-1016.

Código disponível em:
https://github.com/learnedsystems/LearnedSort

Obrigado!