selection sort:

começando do primeiro valor procura da lista, procura o menor valor que for encontrado na lista que seja menor que o valor atual, se houver troca os valores de posição, passa para o próximo valor e repete a lógica.

vantagens:

- Baixo uso de memória

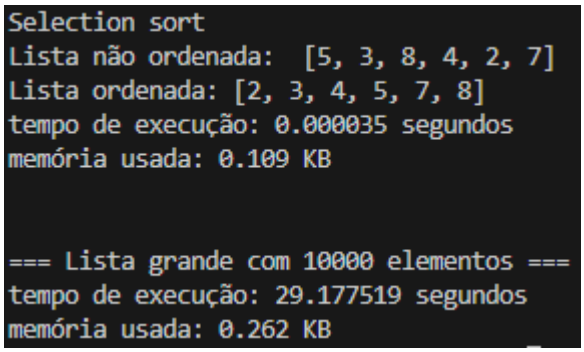
- Bom para listas pequenas

- É in-place

desvantagens:

- Lento para listas grandes

- Mesmo para lista pouco desordenadas continua fazendo comparações item por item



Bubble sort:

compara valores da lista em pares, a partir do primeiro valor, se o valor atual for maior que o próximo valor troca eles de posição, percorre a lista fazendo a comparação e troca de posição (se houver) até a lista estar organizada.

vantagens:

- Em lista quase ordenadas não compara a lista toda, item por item

- Bom para listas pequenas e pouco desordenadas

- É in-place

- Baixo uso de memória

desvantagens:

- Muito lento para listas grandes

- Se o foco for desempenho existem opções melhores

```
Bubble sort

Lista não ordenada:  [5, 3, 8, 4, 2, 7]
Lista ordenada: [2, 3, 4, 5, 7, 8]
tempo de execução: 0.000147 segundos
memória usada: 0.109 KB


=== Lista grande com 10000 elementos ===
tempo de execução: 91.792455 segundos
memória usada: 0.262 KB
```

insertion sort:

começando a partir do segundo valor em diante, compara se os valores anteriores ao atual são maiores, se sim move todos os valores maiores que atual para a direita.

vantagens:

- Muito eficiente em listas pequenas

- Baixo uso de memória

- É in-place

desvantagens:

- Lento para listas grandes

- Se o foco for desempenho existem opções melhores

```
Insertion sort
Lista não ordenada:  [5, 3, 8, 4, 2, 7]
Lista ordenada:  [2, 3, 4, 5, 7, 8]
tempo de execução: 0.000037 segundos
memória usada: 0.078 KB


=== Lista grande com 10000 elementos ===
tempo de execução: 33.666262 segundos
memória usada: 0.137 KB
```

Merge sort:

Segue a estratégia de dividir e conquistar, divide a lista em pedaços até que tenha várias listas unitárias, após isso será juntada de forma ordenado duas a duas listas, até forma uma lista completa totalmente ordenada

vantagens:

- Melhor desempenho

- Eficiente para listas grandes

desvantagens:

- Mais uso de memória

- Muito custoso para listas pequenas

- Não é in-place

```
Merge sort
Lista não ordenada:  [5, 3, 8, 4, 2, 7]
Lista ordenada:  [2, 3, 4, 5, 7, 8]
tempo de execução: 0.000062 segundos
memória usada: 0.188 KB


=== Lista grande com 10000 elementos ===
tempo de execução: 0.203774 segundos
memória usada: 166.195 KB
```

Quick sort:

Também segue a estratégia de dividir e conquistar, escolhe um elemento da lista e reorganiza a lista de forma que os elementos menores que o pivô fiquem de um lado, e os maiores fiquem de outro, então ordena a sub-lista abaixo e acima do pivô de forma recursiva.

vantagens:

- Muito rápido

- Bom para listas grandes

- Pouco uso de memória

- Escalável

- É in-place

desvantagens

- Não estável

- Não é bom dados grandes em disco

```
Quick sort

Lista não ordenada:  [5, 3, 8, 4, 2, 7]
Lista ordenada:  [2, 3, 4, 5, 7, 8]
tempo de execução: 0.000083 segundos
memória usada: 0.547 KB


=== Lista grande com 10000 elementos ===
tempo de execução: 0.065607 segundos
memória usada: 382.281 KB
```

Heap sort:

Se baseia em uma estrutura de dados chamada heap binário, onde tem o max-heap e min-heap

max-heap: cada nó é maior que seus filhos.

min-heap: cada nó é menor que seus filhos.

normalmente usa um max-heap nessa busca, transforma a lista em um heap máximo. O maior elemento sempre estará na raiz (índice 0) -> Troca com o último elemento da lista. -> remove o último elemento (que agora está na posição correta). -> Refaça o heap (heapify) no restante da list

vantagens:

- Desempenho garantido

- Pouco uso de memória

- Não precisa de listas auxiliares

- Ótimo para grande volume de dados

- É in-place

desvantagens:

- Mais lento comparado a Quick sort para listas grandes

- Não é estável

- Implementação complexa

```
Heap sort

Lista não ordenada:  [12, 11, 13, 5, 6, 7]
Lista ordenada:  [5, 6, 7, 11, 12, 13]
tempo de execução: 0.000038 segundos
memória usada: 0.078 KB


=== Lista grande com 10000 elementos ===
tempo de execução: 0.117679 segundos
memória usada: 0.559 KB
```

Counting Sort:

É um algoritmo que conta quantas vezes cada valor aparece e depois reconstrói a lista ordenada com base nessas contagens.

vantagens:

- Muito rápido, mais do que qualquer outro algoritmo de comparação
- Estável
- ótimo para dados de valores numéricos inteiros

desvantagens:

- Gasta memória proporcional ao maior valor.
- Funciona apenas com inteiros
- Não é in-place

```
Counting sort

Lista não ordenada: [4, 2, 2, 8, 3, 3, 1]
Lista ordenada:  [1, 2, 2, 3, 3, 4, 8]
tempo de execução: 0.000042 segundos
memória usada: 0.203 KB


=== Lista grande com 10000 elementos ===
tempo de execução: 0.054316 segundos
memória usada: 461.770 KB
PS C:\Users\UNIVASSOURAS\Documents\kayo>
```

Radix sort:

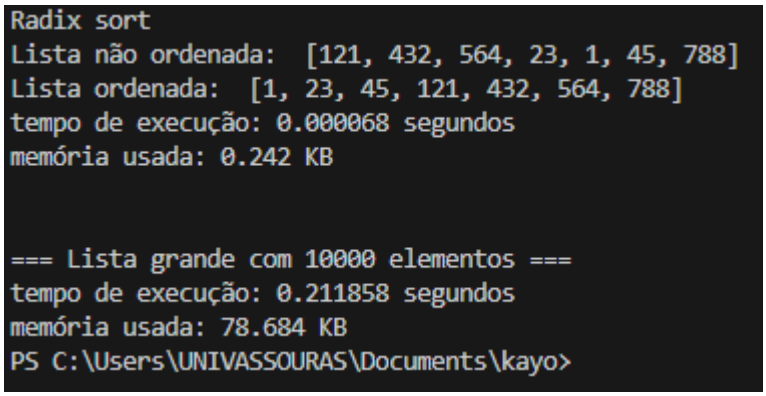
Um algoritmo não comparativo, ordena os números dígito por dígito, do menos significativo para o mais significativo. Encontra o maior número da lista -> Ordena os números por cada dígito, começando pelo dígito das unidades e assim segue. O algoritmo usa o Counting Sort estável para garantir que a ordem dos dígitos anteriores seja mantida.

vantagens:

- Não depende de comparações
- Estável
- Ideal para números inteiros e strings de tamanho fixo.

desvantagens:

- Uso de memória
- Usa outro algoritmo para auxliar
- Usa listas auxiliares
- Não é in-place



Bucket sort:

Algoritmo não comparativo, distribuir os elementos em “baldes” (listas menores), depois ordenar cada balde individualmente.

vantagens:

- Muito rápido se os dados estiverem uniformemente distribuídos (valores float).
- Combina bem com outros algoritmos.
- Rápido

desvantagens:

- Grande uso de memória
- Difícil de aplicar em dados inteiros grandes
- Não é in-place