1. Algoritmos de Busca:

Binary Search:

- **Complexidade de Tempo**:
  - **Melhor caso**: O(1)O(1)O(1) (quando o elemento procurado é o meio da lista).
  - **Pior caso**: O(logn)O(\log n)O(logn) (quando o elemento é encontrado após várias divisões).
  - **Caso médio**: O(logn)O(\log n)O(logn).
- **Complexidade de Espaço**:
  - **Espaço**: O(1)O(1)O(1), pois o algoritmo não requer espaço extra além da entrada.

Explicação: O Binary Search divide a lista em duas metades a cada iteração, portanto sua complexidade é logarítmica. Exige que a lista esteja ordenada.

Interpolation Search:

- **Complexidade de Tempo**:
  - **Melhor caso**: O(1)O(1)O(1) (quando o valor procurado é encontrado na posição correta da interpolação).
  - **Pior caso**: O(n)O(n)O(n) (quando os dados não são uniformemente distribuídos).
  - **Caso médio**: O(loglogn)O(\log \log n)O(loglogn) (quando os dados estão uniformemente distribuídos).
- **Complexidade de Espaço**:
  - **Espaço**: O(1)O(1)O(1), pois não necessita de memória adicional além da entrada.

Explicação: A eficiência do Interpolation Search depende de uma boa distribuição dos dados. Em listas uniformemente distribuídas, pode ser muito mais rápido que o Binary Search.

Jump Search:

- **Complexidade de Tempo**:
  - **Melhor caso**: O(1)O(1)O(1) (se o elemento procurado estiver no primeiro bloco).
  - **Pior caso**: O(n)O(\sqrt{n})O(n) (quando o elemento está no final ou não está presente).
  - **Caso médio**: O(n)O(\sqrt{n})O(n).
- **Complexidade de Espaço**:
  - **Espaço**: O(1)O(1)O(1), porque o Jump Search só usa variáveis para o controle do salto e do índice.

Explicação: O Jump Search divide a lista em blocos de tamanho n\sqrt{n}n e, em seguida, busca linearmente dentro do bloco onde o elemento pode estar. Sua eficiência é limitada ao tamanho do salto.

Exponential Search:

- **Complexidade de Tempo**:

    - **Melhor caso**: O(1)O(1)O(1).
    - **Pior caso**: O(logn)O(\log n)O(logn).
    - **Caso médio**: O(logn)O(\log n)O(logn).
- **Complexidade de Espaço**:
    - **Espaço**: O(1)O(1)O(1), pois o algoritmo não exige espaço adicional.

Explicação: O Exponential Search começa buscando exponencialmente por intervalos e, uma vez encontrado o intervalo de busca, usa Binary Search para realizar a busca dentro do intervalo. É eficiente para listas muito grandes e ordenadas.

---

2. Algoritmos de Ordenação:

Shell Sort:

- **Complexidade de Tempo**:
    - **Melhor caso**: O(nlogn)O(n \log n)O(nlogn) (com uma boa sequência de intervalos).
    - **Pior caso**: O(n2)O(n^2)O(n2) (quando a sequência de intervalos é ruim).
    - **Caso médio**: O(n3/2)O(n^{3/2})O(n3/2) (dependendo da sequência de intervalos utilizada).
- **Complexidade de Espaço**:
    - **Espaço**: O(1)O(1)O(1), pois o Shell Sort é um algoritmo de ordenação in-place.

Explicação: O Shell Sort melhora o Insertion Sort utilizando uma sequência de intervalos que permite a troca de elementos distantes. A complexidade depende da escolha da sequência de intervalos, sendo mais eficiente que o Insertion Sort em muitos casos.

Merge Sort:

- **Complexidade de Tempo**:
    - **Melhor caso**: O(nlogn)O(n \log n)O(nlogn).
    - **Pior caso**: O(nlogn)O(n \log n)O(nlogn).
    - **Caso médio**: O(nlogn)O(n \log n)O(nlogn).
- **Complexidade de Espaço**:
    - **Espaço**: O(n)O(n)O(n), pois é necessário espaço extra para as sublistas.

Explicação: O Merge Sort divide a lista em sublistas e as combina de maneira ordenada. Ele tem uma complexidade de tempo garantida de O(nlogn)O(n \log n)O(nlogn), mas consome espaço extra para realizar as divisões e combinações.

Selection Sort:

- **Complexidade de Tempo**:
    - **Melhor caso**: O(n2)O(n^2)O(n2).
    - **Pior caso**: O(n2)O(n^2)O(n2).
    - **Caso médio**: O(n2)O(n^2)O(n2).

- **Complexidade de Espaço**:
    - **Espaço**: O(1)O(1)O(1), pois o Selection Sort é um algoritmo in-place.

Explicação: O Selection Sort percorre a lista e, a cada iteração, seleciona o menor elemento e o coloca na posição correta. Sua complexidade é quadrática, tornando-o ineficiente para listas grandes.

Quick Sort:

- **Complexidade de Tempo**:
    - **Melhor caso**: O(nlog⁡n)O(n \log n)O(nlogn) (quando o pivô divide bem a lista).
    - **Pior caso**: O(n2)O(n^2)O(n2) (quando o pivô é mal escolhido, como no caso de uma lista já ordenada ou inversamente ordenada).
    - **Caso médio**: O(nlog⁡n)O(n \log n)O(nlogn).
- **Complexidade de Espaço**:
    - **Espaço**: O(log⁡n)O(\log n)O(logn), devido à recursão na pilha de chamadas.

Explicação: O Quick Sort é eficiente para grandes listas, mas sua eficiência depende da escolha do pivô. No pior caso, pode degenerar para O(n2)O(n^2)O(n2), mas, em geral, tem desempenho O(nlog⁡n)O(n \log n)O(nlogn).

Bucket Sort:

- **Complexidade de Tempo**:
    - **Melhor caso**: O(n+k)O(n + k)O(n+k) (quando os elementos são uniformemente distribuídos).
    - **Pior caso**: O(n2)O(n^2)O(n2) (quando todos os elementos caem em um único balde).
    - **Caso médio**: O(n+k)O(n + k)O(n+k).
- **Complexidade de Espaço**:
    - **Espaço**: O(n+k)O(n + k)O(n+k), onde nnn é o número de elementos e kkk o número de baldes.

Explicação: O Bucket Sort é eficiente quando os dados estão uniformemente distribuídos, dividindo-os em baldes e ordenando dentro desses baldes (geralmente usando outro algoritmo como o Insertion Sort). No pior caso, quando todos os elementos ficam em um único balde, a complexidade é O(n2)O(n^2)O(n2).

Radix Sort:

- **Complexidade de Tempo**:
    - **Melhor caso**: O(nk)O(nk)O(nk), onde kkk é o número de dígitos.
    - **Pior caso**: O(nk)O(nk)O(nk).
    - **Caso médio**: O(nk)O(nk)O(nk).
- **Complexidade de Espaço**:

- **Espaço**: O(n+k)O(n + k)O(n+k), onde nnn é o número de elementos e kkk o número de dígitos.

Explicação: O Radix Sort é eficiente para números inteiros e usa um algoritmo de contagem para ordenar por dígitos. Ele é linear com relação ao número de elementos nnn e ao número de dígitos kkk, o que o torna muito eficiente para números com poucos dígitos.