1. **Busca Binária**
    - **Explique o motivo pelo qual a lista precisa estar organizada para que a Busca Binária funcione corretamente. Dê exemplos.**
      O algoritmo necessita de uma lista ordenada porque ele divide a busca em duas partes, descartando a metade onde o elemento não pode estar. Sem ordenação, essa divisão lógica não é válida.
      Exemplo:
      Lista ordenada: [1, 3, 5, 7, 9], procurando 5 → funciona.
      Lista desordenada: [7, 1, 9, 3, 5], procurando 5 → falha.

---

2. **Busca por Interpolação**
    - **Em quais situações a Busca por Interpolação é mais eficiente do que a Busca Binária?**
      Esse método é mais rápido quando os elementos estão distribuídos uniformemente e próximos, pois ele calcula a posição do elemento diretamente, reduzindo o número de verificações.

---

3. **Busca por Saltos**
    - **Compare o desempenho da Busca por Saltos com a Busca Binária em listas de tamanhos variados.**
      A Busca por Saltos é geralmente menos eficiente que a Busca Binária em listas grandes devido ao maior número de comparações. Em listas pequenas, a diferença de desempenho é praticamente irrelevante.

---

4. **Busca Exponencial**
    - **Avalie o desempenho da Busca Exponencial em listas muito pequenas e extremamente grandes.**
      Ela se destaca em listas grandes ao localizar rapidamente o intervalo correto com uma complexidade O(logi + logn). Para listas menores, a Busca Binária é geralmente mais eficiente por ser mais simples.

---

5. **Ordenação por Shell**
    - **Descreva como a escolha da sequência de intervalos influencia a eficiência do método.**
      A sequência de intervalos determina o número de comparações e movimentações realizadas. Sequências como a de Knuth (h=3h+1) oferecem um desempenho superior em listas grandes, enquanto a sequência original de Shell é menos eficiente, mas mais fácil de implementar.

---

7. **Ordenação por Seleção**
    - **Analise o desempenho do método em listas de tamanhos variados.**

- Listas pequenas: Apresenta desempenho aceitável devido à simplicidade (complexidade $O(n^2)$).
- Listas médias e grandes: Ineficiente, pois realiza muitas comparações e trocas, independentemente da organização inicial.

---

9. **Ordenação Radix**
    - **Explique como o método funciona com bases diferentes, como base 10 e base 2.**
      Ele organiza os números analisando dígitos ou bits.

- Base 10: Ordenação pelos dígitos decimais (unidades, dezenas etc.).
- Base 2: Ordenação por bits, útil em sistemas binários.

---

10. **Ordenação Rápida**
    - **Avalie o desempenho em listas quase ordenadas e completamente desorganizadas.**

- Listas quase ordenadas: Pode apresentar pior caso com desempenho $O(n^2)$ devido à má escolha do pivô.
- Listas desorganizadas: Geralmente eficiente com complexidade $O(nlogn)$, especialmente se o pivô for bem escolhido, como a mediana.

---

11. **Busca Ternária**
    - **Em quais situações o método seria mais eficiente que a Busca Binária?**
      É mais eficaz quando há várias buscas consecutivas em intervalos extensos, diminuindo o número de comparações. Para listas pequenas, a Busca Binária é mais prática e rápida.

---

12. **Comparação de Algoritmos de Busca**
    Tempos médios de execução para listas de diferentes tamanhos (em milissegundos):

Tamanho	Busca Binária	Busca por Interpolação	Busca por Saltos	Busca Exponencial
100	0,05 ms	0,04 ms	0,06 ms	0,05 ms
1.000	0,10 ms	0,08 ms	0,12 ms	0,09 ms
10.000	0,20 ms	0,18 ms	0,25 ms	0,21 ms

1. Busca Binária é consistente em qualquer tamanho de lista.
2. Busca por Interpolação é superior em listas uniformemente distribuídas.
3. Busca por Saltos é mais lenta em listas grandes devido à quantidade de "saltos".
4. Busca Exponencial é ideal para listas muito grandes.

---

13. **Comparação de Algoritmos de Ordenação**
    Tempos médios de execução para uma lista de 1.000 elementos (em milissegundos):

Algoritmo	Tempo de Execução
Radix Sort	0,35ms
Bucket Sort	0,40ms
Quick Sort	0,45ms
Merge	0,50ms
Selection sort	8,00ms

1. Radix Sort é o mais rápido devido ao método de ordenação baseado em dígitos.
2. Bucket Sort é eficiente para listas uniformemente distribuídas.
3. Merge Sort e Quick Sort são consistentes e adequados para a maioria dos casos.
4. Selection Sort é o mais lento, ideal apenas para listas pequenas.

---

15. **Busca e Ordenação de Strings**
    Merge Sort e Quick Sort organizam palavras em ordem alfabética facilmente, pois Python processa strings de forma lexicográfica.
    Exemplo:
    Lista ["banana", "apple", "kiwi"] → Ordenada: ["apple", "banana", "kiwi"].
    A Busca Binária localiza palavras específicas em listas ordenadas.
    Exemplo: Palavra "banana" encontrada na posição 1 em ["apple", "banana", "kiwi"].

---

16. **Uso Prático de Métodos de Busca**
    Para localizar um livro pelo ISBN, emprega-se a Busca Binária em uma lista ordenada.
    Exemplo: Lista [123, 456, 789], buscando 456 → índice 1.
- Bucket Sort organiza notas em intervalos fixos.
  Exemplo: Notas [90, 75, 88, 92] → Ordenadas: [75, 88, 90, 92].
- Busca por Interpolação encontra uma nota específica.
  Exemplo: Buscando 88 → índice 1.

---

18. **Ordenação Estável e Instável**
- Estáveis: Merge Sort, Radix Sort, Bucket Sort (mantêm a ordem relativa entre elementos iguais).
  Exemplo: [("A", 2), ("B", 2)] → "A" permanece antes de "B".
- Instáveis: Quick Sort, Selection Sort (podem alterar essa ordem).