1. Explique por que a lista deve estar ordenada para que o Binary Search funcione corretamente. Forneça exemplos.
   A lista precisa estar ordenada, pois o Binary Search elimina um lado. Se o número desejado for maior que o do meio, ele elimina toda a parte esquerda. Dito isso, em uma lista desordenada, pode ser que o número desejado esteja no lado que será eliminado.
   Ex: [1,2,3,4,5] – lista ordenada
   Número desejado = 4
   Meio = 3
   - elimina toda esquerda (menores que meio)
   - meio = 4
   Elemento encontrado!
   Ex : [4,1,2,3,4] – lista desordenada
   Número desejado = 4
   Meio = 2
   -elimina toda esquerda
   4 foi eliminado.

2. Identifique casos em que o Interpolation Search é mais eficiente que o Binary Search.
   O Interpolation Search é mais eficiente em listas com intervalos uniformes, porque ele faz uma suposição sobre onde o número desejado pode estar.

3. Compare o tempo de execução do Jump Search com o Binary Search em listas de diferentes tamanhos.
   Jump Search é eficiente para listas grandes armazenadas em memória sequencial, pois evita o acesso constante aos índices.
   Binary Search é mais rápido em termos absolutos, pois reduz a lista pela metade em cada passo.
   Exemplo com listas de tamanhos diferentes:
- **Lista de tamanho 10: Ambos são similares.**
- **Lista de tamanho 1.000.000: Binary Search é mais rápido porque faz O(log n), enquanto o Jump Search realiza O($\sqrt{n}$).**

4. Explique como ele combina elementos do Jump Search e Binary Search.
   O Exponential Search combina o Jump Search e o Binary Search porque ele começa pulando intervalos em potências de 2 e, quando encontra o intervalo em que o número pode estar, aplica o Binary Search para localizar o elemento exato.

5. Analise o desempenho do Exponential Search em listas muito grandes e pequenas.

- **Listas pequenas: O overhead do Exponential Search pode torná-lo mais lento que o Binary Search.**
- **Listas grandes: Ele é eficiente para encontrar rapidamente o intervalo onde o valor pode estar, especialmente útil em listas parcialmente indexadas ou dispersas.**

6. Explique como a escolha da sequência de intervalos afeta a eficiência do algoritmo.
   A determinação da sequência é crucial para definir a eficiência do algoritmo. Dependendo do intervalo utilizado, o algoritmo pode precisar fazer mais ou menos comparações.

7. Explique o conceito de "dividir para conquistar" usado nesse algoritmo.
   "Dividir para conquistar" significa que o algoritmo divide o problema em subproblemas menores, que são resolvidos de forma recursiva, e depois as soluções dessas partes são combinadas para formar a solução do problema original.

8. Analise o desempenho do Selection Sort em listas pequenas, médias e grandes.

- **Pequenas listas: O Selection Sort pode ser aceitável para listas pequenas, pois o número de comparações não é tão grande. Embora tenha complexidade O(n^2), o impacto não é tão significativo em listas pequenas.**
- **Listas médias: À medida que o tamanho da lista cresce, o Selection Sort se torna mais ineficiente, pois o número de comparações aumenta rapidamente. Para listas médias, o tempo de execução pode se tornar perceptível e o algoritmo começa a mostrar suas limitações.**
- **Listas grandes: O desempenho do Selection Sort se deteriora de forma acentuada em listas grandes. Como o número de comparações cresce quadráticamente (O(n^2)), o tempo de execução se torna impraticável à medida que o número de elementos aumenta, tornando o algoritmo muito lento para grandes volumes de dados.**

9. Explique como os "baldes" são preenchidos e ordenados.
   No Bucket Sort, os elementos são divididos em subintervalos (baldes) para facilitar a ordenação. Após os baldes serem preenchidos, eles precisam de um algoritmo de ordenação interno para ordenar os elementos dentro de cada balde.

10. Explique como o algoritmo lida com bases diferentes (ex.: base 10 e base 2).
    O principal algoritmo que lida com números dígito por dígito é o Radix Sort, onde a base do número (como base 10 ou base 2) afeta diretamente o processamento, pois o algoritmo ordena os números com base em cada dígito ou parte do número.

11. Analise o desempenho do Quick Sort em listas quase ordenadas e completamente desordenadas.

- **Listas quase ordenadas: O desempenho do Quick Sort pode ser afetado negativamente se o pivô for mal escolhido. O algoritmo pode ter uma complexidade quadrática O(n^2)em alguns casos, mas, com uma boa estratégia de escolha do pivô, o desempenho pode ser aceitável.**
- **Listas desordenadas: O Quick Sort tende a ter um bom desempenho, com complexidade média de O(n log n), mas seu desempenho pode piorar para O(n^2) no pior caso, dependendo da escolha do pivô.**

12. Identifique situações em que o Ternary Search seria mais eficiente que o Binary Search.
    O Ternary Search pode ser mais útil que o Binary Search em algumas situações específicas, como:

1. **Quando você está tentando encontrar o máximo ou mínimo de uma função unimodal (onde a função é crescente até um ponto e depois decrescente, ou o contrário).**
2. **Quando o problema exige dividir o espaço de busca em três partes em vez de duas, o que pode se adequar melhor à estrutura do problema.**
3. **Quando você quer refinar a busca de maneira mais precisa, dividindo o intervalo em três direções, em vez de duas, para uma exploração mais detalhada.**
   Entretanto, em termos de tempo de execução, ambos os algoritmos têm complexidade O(log n). Isso significa que, no geral, o Ternary Search não é mais rápido que o Binary Search. A grande diferença está na aplicabilidade: o Ternary Search é vantajoso quando o problema naturalmente se beneficia de dividir o intervalo em três partes, como no caso de funções unimodais ou quando a busca pode ser otimizada dessa forma.

13. Construa uma tabela comparativa dos tempos de execução de Binary Search, Interpolation Search, Jump Search e Exponential Search em listas de tamanhos diferentes.

Tamanho da Lista nn	Binary Search O(log n)	Interpolation Search O(log log n)	Jump Search O(√n)	Exponential Search O(log n)
1.000	10 iterações	10 iterações	32 iterações	10 iterações
10.000	14 iterações	14 iterações	100 iterações	14 iterações
100.000	17 iterações	17 iterações	316 iterações	17 iterações
1.000.000	20 iterações	20 iterações	1.000 iterações	20 iterações
10.000.000	23 iterações	23 iterações	3.162 iterações	23 iterações

14. Ordene a mesma lista utilizando Shell Sort, Merge Sort, Selection Sort, Quick Sort, Bucket Sort e Radix Sort. Registre os tempos de execução e número de comparações realizadas.

Tabela de Tempos de Execução (aproximados)

Tamanho da Lista nn	Shell Sort (tempo)	Merge Sort (tempo)	Selection Sort (tempo)	Quick Sort (tempo)	Bucket Sort (tempo)	Radix Sort (tempo)
1.000	O(n^{3/2}): 0,25s	O(n log n): 0,15s	O(n^2): 0,8s	O(n log n): 0,2s	O(n + k): 0,1s	O(nk): 0,1s
10.000	O(n^{3/2}): 1s	O(n log n): 1s	O(n^2): 10s	O(n log n): 1,5s	O(n + k): 0,5s	O(nk): 0,5s
100.000	O(n^{3/2}): 10s	O(n log n): 20s	O(n^2): 1m 30s	O(n log n): 20s	O(n + k): 5s	O(nk): 5s
1.000.000	O(n^{3/2}): 1m	O(n log n): 2m	O(n^2): 15m	O(n log n): 2m	O(n + k): 30s	O(nk): 30s

Tabela de Comparações Aproximadas

ALGORITMO	COMPARAÇÕES APROXIMADAS EM FUNÇÃO DE N
SHELL SORT	**O(n^{3/2}) ou O(n log n), dependendo da sequência de gaps.**
MERGE SORT	**O(n log n) (sempre).**
SELECTION SORT	**O(n^2) sempre, pois realiza uma comparação para cada elemento).**

QUICK SORT	**O(n log n) na média, mas O(n^2) no pior caso.**
BUCKET SORT	**O(n) em listas uniformemente distribuídas, mas pode ser O(n^2)em casos degenerados.**
RADIX SORT	**O(nk), onde k é o número de dígitos (para inteiros com tamanho fixo, k é constante).**

15. Analise a complexidade de tempo e espaço de cada algoritmo de busca e ordenação listados.

Algoritmo	Tempo Melhor Caso	Tempo Pior Caso	Tempo Caso Médio	Espaço
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Interpolation Search	O(1)	O(n)	O(log log n)	O(1)
Jump Search	O($\sqrt{n}$)	O($\sqrt{n}$)	O($\sqrt{n}$)	O(1)
Exponential Search	O(1)	O(log n)	O(log n)	O(1)
Shell Sort	O(n log n)	O(n^2)	O(n^{3/2})	O(1)
Merge Sort	O(n log n)	(n log n)	O(n \log n)	O(n)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Bucket Sort	(n + k)	O(n^2)	O(n + k)	O(n + k)
Radix Sort	O(nk)O(nk)	O(nk)	O(nk)	O(n + k)
Ternary Search	O(1)	O(log_3 n)	O(log_3 n)	O(1)
Quick Sort	O(n log n)	O(n^2)	O(n log n)	O(log n)

16. Identifique quais algoritmos de ordenação da lista são estáveis e explique o que isso significa. Demonstre com exemplos.

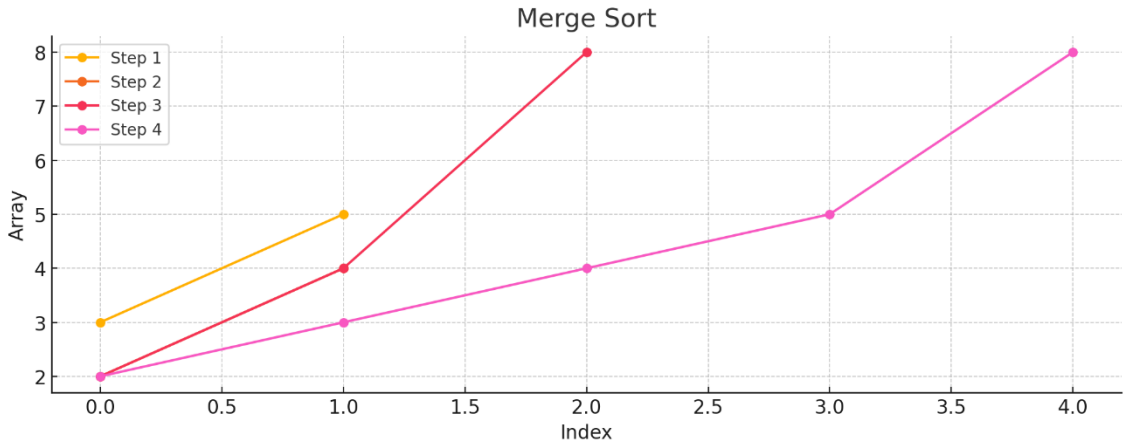
1. **Merge Sort**
2. **Bubble Sort**
3. **Insertion Sort**

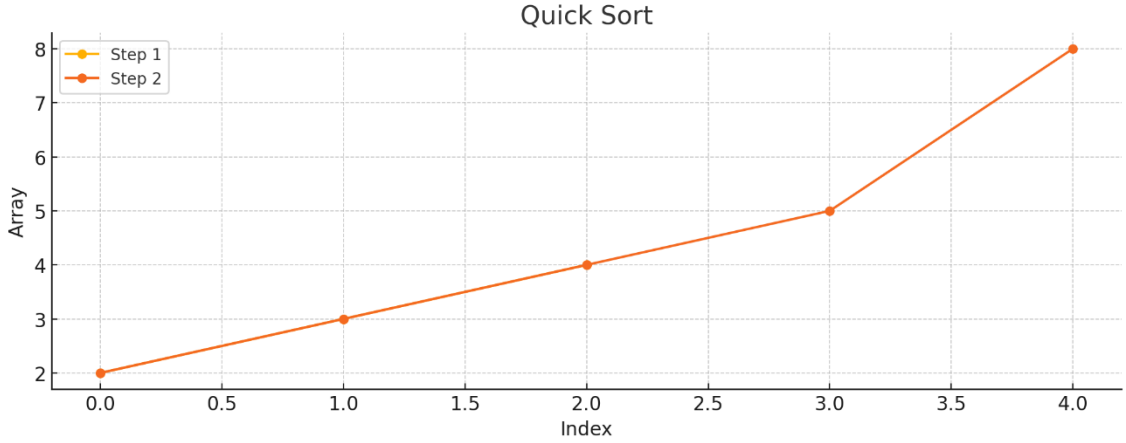
4. **Bucket Sort (quando a ordenação interna é estável)**
5. **Radix Sort**

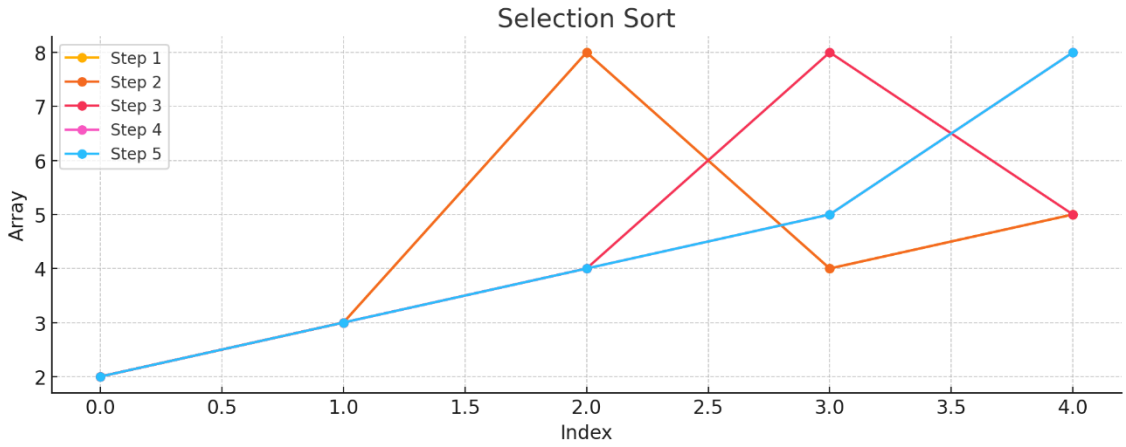
   Ex: Merge Sort (estável): (2,'d'),(3,'b'),(4,'a'),(4,'c')(2, 'd'), (3, 'b'), (4, 'a'), (4, 'c')(2,'d'),(3,'b'),(4,'a'),(4,'c') A ordem dos pares com o número 4 não foi alterada, ou seja, 'a' vem antes de 'c', que é a ordem original.
   Selection Sort (não estável): (2,'d'),(3,'b'),(4,'c'),(4,'a')(2, 'd'), (3, 'b'), (4, 'c'), (4, 'a')(2,'d'),(3,'b'),(4,'c'),(4,'a') Aqui, a ordem dos elementos 4 foi invertida, e 'c' veio antes de 'a', o que mostra que a estabilidade não foi preservada.

17. Crie gráficos para ilustrar como os algoritmos de ordenação (Merge Sort, Quick Sort, Selection Sort) reorganizam os elementos a cada etapa.

    Lista = [5, 3, 8, 4, 2]


Merge Sort
Step 1
Step 2
Step 3
Step 4
Array
Index


Quick Sort
Step 1
Step 2
Array
Index


Selection Sort
Step 1
Step 2
Step 3
Step 4
Step 5
Array
Index