Taxa de Crescimento de Algoritmos

Emannuel Levi de Assis Bezerra, Henrique Fernandes, Raimundo Rafael

Outubro de 2024

1 Algoritmos Selecionados

1.1 Complexidade O(1) – Inserção na Pilha (Push)

O algoritmo de inserção de elementos no topo de uma pilha, conhecido como *push*, possui complexidade constante $O(1)$. Isso significa que, independentemente do tamanho da pilha, a operação sempre será executada no mesmo tempo, pois o novo elemento é adicionado diretamente no topo da estrutura.

1.2 Complexidade O(n) – Busca em Vetor

A busca por um elemento em um vetor desorganizado tem complexidade linear $O(n)$. Nesse caso, o algoritmo pode ter que percorrer todo o vetor para encontrar o elemento desejado. O número de iterações depende diretamente do tamanho n do vetor.

1.3 Complexidade O(n^2) – Busca em Matriz Quadrada

A busca por um elemento em uma matriz quadrada desorganizada tem complexidade quadrática $O(n^2)$. Isso se deve ao fato de que o algoritmo percorre a matriz linha por linha, com dois laços de repetição (*for loops*) aninhados, um para as linhas e outro para as colunas, resultando em um número de iterações proporcional ao produto do número de linhas pelo número de colunas (que são iguais).

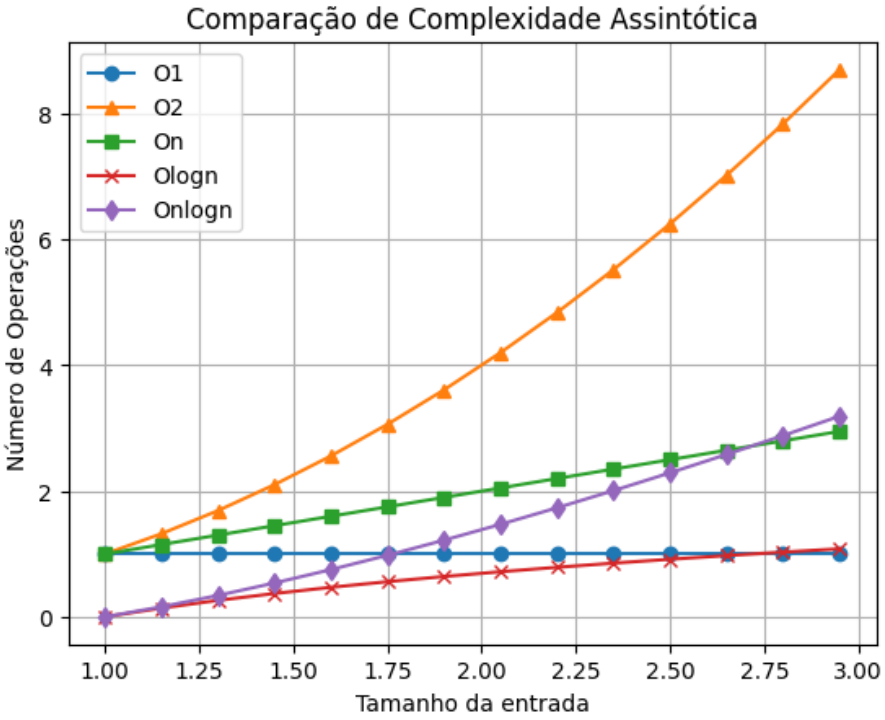
1.4 Complexidade O(log n) – Busca Binária em Vetor Organizado

A busca binária, aplicada a vetores previamente ordenados, tem complexidade $O(\log n)$. Nesse algoritmo, o vetor é dividido ao meio a cada iteração, reduzindo o espaço de busca em 50%. Isso faz com que o tempo de execução cresça de forma logarítmica em relação ao tamanho do vetor.

1.5 Complexidade O(n log n) – Ordenação de Vetores

Algoritmos de ordenação eficientes, como o *mergesort* e o *quicksort*, possuem complexidade $O(n \log n)$. O *mergesort*, por exemplo, funciona dividindo o vetor em partes menores recursivamente até que as partes tenham tamanho suficientemente pequeno para serem ordenadas diretamente. Em seguida, essas partes são combinadas (*merge*) em uma solução maior e organizada. Esse processo garante uma ordenação eficiente mesmo para grandes conjuntos de dados.

2 Análise de Complexidade



Para valores muito grandes de N, o algoritmo mais eficiente seria o de complexidade $O(1)$, uma vez que seu tempo de execução permanece constante, independentemente da quantidade de entradas. Isso significa que, mesmo para entradas extremamente grandes, o tempo de processamento não seria impactado.

No entanto, algoritmos com complexidade $O(1)$ são raros em problemas mais complexos. Por isso, entre os analisados, o segundo algoritmo mais eficiente é aquele com complexidade $O(\log n)$. Ele possui a menor taxa de crescimento entre os algoritmos com dependência do tamanho da entrada, sendo altamente recomendável para problemas que envolvem grandes quantidades de dados. Sua eficiência decorre da redução progressiva do espaço de busca a cada iteração, tornando-o uma excelente escolha quando algoritmos $O(1)$ não estão disponíveis.

Repositório no GitHub do código:
`https://github.com/dabzr/data-structures/tree/main/N1/Atividades/Notacao-Assintotica`