1. Escreva uma função em Python que reverte a ordem de uma lista de n inteiros. Em seguida, calcule a função T(n) (modelo simplificado) e faça o estudo comparativo em uma tabela e gráfico do desempenho de sua função *versus* o método nativo fornecido pela linguagem.

2. Escreva uma classe em Python contendo um método que recebe dois arrays (de tamanho n) *a* e *b* contendo valores inteiros e calcula o produto de *a* por *b*. Sendo assim, seu método deve retornar um novo array c, tal que c[i] = a[i] * b[i], para i = 0, …, n-1. Faça o estudo comparativo em uma tabela e gráfico do desempenho de seu código.

3. Escreva um programa em Python que recebe como argumento um polinômio em notação algébrica e gera como saída a **primeira derivada** desse polinômio. Em seguida, calcule a função T(n) (modelo simplificado) e faça o estudo do seu desempenho em uma tabela e um gráfico.

4. Escreva um programa em Python para encontrar o *mínimo* e *máximo* elementos de um dado array de tamanho n usando menos que *3n/2* comparações. (Dica: Primeiro, construa um grupo de mínimos candidatos e outro de máximos candidatos)

5. Para cada um dos algoritmos a seguir, *unique1* e *unique2* os quais resolvem o problema da singularidade de um elemento, executar uma análise experimental do tempo de execução para determinar o valor maior de n tal que o algoritmo dado execute em um minuto ou menos.

```
1  def unique1(S):
2    """Return True if there are no duplicate elements in sequence S."""
3    for j in range(len(S)):
4      for k in range(j+1, len(S)):
5        if S[j] == S[k]:
6          return False                # found duplicate pair
7    return True                       # if we reach this, elements were unique
```

```
1  def unique2(S):
2    """Return True if there are no duplicate elements in sequence S."""
3    temp = sorted(S)                  # create a sorted copy of S
4    for j in range(1, len(temp)):
5      if S[j-1] == S[j]:
6        return False                  # found duplicate pair
7    return True                       # if we reach this, elements were unique
```

6. Suponha que temos uma sequência de n elementos S tal que cada elemento em S representa um voto diferente para presidente, onde cada voto é dado como um inteiro que representa um determinado candidato. Projetar um algoritmo com *O (n log n)* de tempo de complexidade para ver quem ganha a eleição S, assumindo que ganha o candidato com mais votos.
7. Implementar uma versão *in-place* de ***insertion-sort*** e uma versão *in-place* do ***selection-sort***. Execute testes de *benchmarking* para determinar o intervalo de valores de n onde q ***selection-sort*** é, em média, melhor do que a ***insertion-sort***.