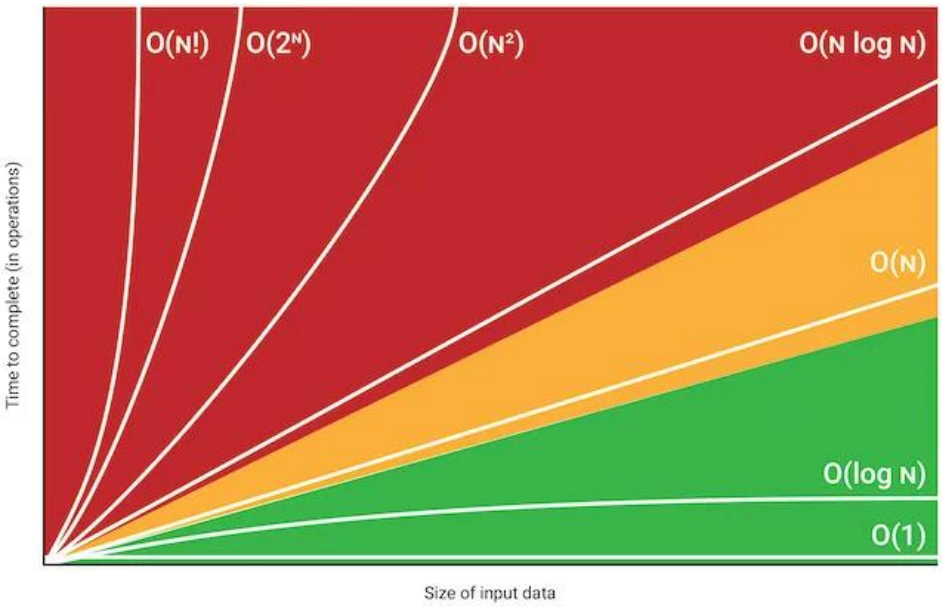
Trabalho prático 2 – Estrutura de Dados

Aluno: Áquila Oliveira Souza

Introdução

Este relatório tem como objetivo aprofundar a análise de desempenho de algoritmos, concentrando-se em duas funções fundamentais: o cálculo da sequência de Fibonacci e o cálculo do fatorial. Essas funções serão implementadas tanto de forma recursiva quanto iterativa, permitindo-nos explorar e comparar as características de desempenho de ambas as abordagens.

A análise que se segue tem como enfoque principal o tempo de execução, detalhando os componentes de tempo de usuário e tempo de sistema. Ao desmembrar esses aspectos, poderemos avaliar com precisão o desempenho e a eficiência de cada implementação. Esta investigação visa fornecer insights valiosos sobre como diferentes abordagens algorítmicas podem influenciar a eficácia de um programa em termos de consumo de recursos e tempo de execução, auxiliando assim na tomada de decisões informadas na seleção e otimização de algoritmos.



Fibonacci e Sua Ordem de Complexidade

No cálculo da sequência de Fibonacci, os termos F(n-1) e F(n-2) são computados independentemente, resultando em uma ineficiência notável. O custo computacional para calcular F(n) é da ordem O(φ^n), onde φ representa a constante matemática conhecida como "número de ouro" (φ ≈ 1,61803...). Essa constante revela que o custo é exponencial, o que se torna evidente nos testes realizados.

Fibonacci Iterativo

A abordagem iterativa para o cálculo de Fibonacci apresenta uma complexidade de tempo significativamente mais eficiente, sendo da ordem O(n). Isso demonstra que a recursividade cega não é a melhor escolha ao lidar com esse problema.

Fatorial Recursivo:

A função fatorial recursiva geralmente tem uma complexidade de tempo de O(n), onde "n" é o número para o qual você está calculando o fatorial. No entanto, se não houver otimização para armazenar valores já calculados (como memorização), a complexidade será exponencial, o que significa que a função demorará muito para números grandes. Com memorização, a complexidade será O(n) porque os resultados intermediários são armazenados e reutilizados.

Fatorial Iterativo:

A função fatorial implementada de forma iterativa geralmente tem uma complexidade de tempo de O(n), pois você realiza um loop simples que multiplica números de 1 até n.

Tempo de Execução para Fibonacci Recursivo em segundos:

Resultados para Diferentes Valores de n

n = 5:

Tempo do sistema: [0.000002146]

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000]

n = 10:

Tempo do sistema: [0.000006914]

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000000]

n = 20:

Tempo do sistema: [0.000664949

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000000]

n = 35:

Tempo do sistema: [0.624644041]

Tempo de usuário: [0.593750000]

Tempo de relógio: [0.640625]

n = 40:

Tempo do sistema: [6.855979919]

Tempo de usuário: [8.015625000]

Tempo de relógio: [7.093750]

Tempo de Execução para Fibonacci Iterativo

Resultados para Diferentes Valores de n

n = 5:

Tempo do sistema: [0.000001907]

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000000]

n = 10:

Tempo do sistema: [0.000001907]

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000000]

n = 20:

Tempo do sistema: [0.000001907]

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000000]

n = 35:

Tempo do sistema: [0.000000954]

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000000]

n = 40:

Tempo do sistema: [0.000000954]

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000000]

Tempo de Execução para Fatorial Recursivo

Resultados para Diferentes Valores de n

n = 5:

Tempo do sistema: [0.000002146]

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000000]

n = 10:

Tempo do sistema: [0.000001907]

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000000]

n = 20:

Tempo do sistema: [0.000003099]

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000000]

n = 35:

Tempo do sistema: [0.000002146]

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000000]

n = 40:

Tempo do sistema: [0.000002861]

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000000]

Tempo de Execução para Fatorial Iterativo

Resultados para Diferentes Valores de n

n = 5:

Tempo do sistema: [0.000002146]

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000000]

n = 10:

Tempo do sistema: [0.000001907]

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000000]

n = 20:

Tempo do sistema: [0.000000954]

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000000]

n = 35:

Tempo do sistema: [0.000000954]

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000000]

n = 40:

Tempo do sistema: [0.000000954]

Tempo de usuário: [0.000000000]

Tempo de relógio: [0.000000000]

Conclusão para cálculo dos tempos

Para cálculo da função Fibonacci:

A análise de tempo de execução demonstrou que a abordagem iterativa para calcular a sequência de Fibonacci é substancialmente mais eficiente do que a abordagem recursiva, especialmente para valores grandes de "n". Os resultados reforçam a importância de evitar a recursividade cega ao resolver problemas computacionais que podem ser otimizados de maneira mais eficaz com abordagens iterativas. É fundamental considerar a complexidade de tempo ao escolher um algoritmo para evitar ineficiências significativas.

Para cálculo da função fatorial:

Em resumo, a ordem de complexidade de ambas as implementações é O(n) no caso de implementações eficientes, mas a implementação recursiva sem memorização pode ser muito ineficiente para números grandes. A implementação iterativa é a mais eficiente para calcular fatoriais.

Relatório gerado pelo gprof

Vale destacar também que foi utilizado no projeto a ferramenta gprof.Os relatórios gerados pelo gprof são ferramentas essenciais para a análise de desempenho de programas. Eles fornecem informações detalhadas sobre a utilização de recursos e a execução das funções, permitindo aos desenvolvedores identificar áreas críticas de código que podem ser otimizadas. Esses relatórios são vitais para melhorar a eficiência e a velocidade dos programas, contribuindo para a qualidade do software e economia de recursos de computação.

Diante disso, a conclusão tirada usando essa ferramenta e que para casos com n como um valor muito grande, a maior parte das chamadas no código é por conta de Fibonacci recursivo, o que é justificado por sua ordem de complexidade.

Conclusão

Este estudo analisou o tempo de execução de duas funções fundamentais, o cálculo de Fibonacci e o cálculo fatorial, implementados tanto de forma recursiva quanto iterativa. Os resultados obtidos destacam a importância de escolher abordagens eficientes ao resolver problemas computacionais.

No caso do cálculo da sequência de Fibonacci, ficou evidente que a abordagem iterativa é substancialmente mais eficiente do que a abordagem recursiva. Isso é especialmente notável para valores grandes de "n". A complexidade de tempo da implementação iterativa é O(n), enquanto a implementação recursiva apresenta uma complexidade de tempo exponencial, tornando-a impraticável para valores elevados de "n". Portanto, ao lidar com o cálculo de Fibonacci, é aconselhável evitar a recursividade cega e optar por abordagens iterativas para obter um desempenho mais eficaz.

No que diz respeito ao cálculo do fatorial, ambas as implementações, recursiva e iterativa, têm uma complexidade de tempo eficiente de O(n). No entanto, a implementação recursiva sem otimizações, como a memorização, pode ser ineficiente para números grandes, devido à sua natureza de recursão. A implementação iterativa, por outro lado, oferece um desempenho estável e eficiente, tornando-se a escolha preferencial para calcular fatoriais.

Em resumo, ao abordar problemas computacionais que envolvem cálculos de sequências ou fatoriais, é essencial considerar a complexidade de tempo das implementações disponíveis. A escolha da abordagem adequada pode fazer uma diferença significativa no desempenho do algoritmo, tornando-o mais rápido e eficiente. Portanto, ao desenvolver soluções computacionais, é crucial evitar abordagens recursivas cegas e priorizar abordagens iterativas e otimizadas sempre que possível. Isso contribuirá para uma execução mais eficiente e rápida de tarefas computacionais.