UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PROJETO E ANÁLISE DE ALGORITMO

Programação Dinâmica

Diogo Augusto Martins Honorato
Rhayan de Sousa Barcelos

São João Del-Rei
2024

Sumário

1. INTRODUÇÃO

Este é um trabalho prático da disciplina de Projeto e Análise de Algoritmos no curso de Ciência da Computação na UFSJ, tendo como docente o professor Leonardo Rocha

1.1. Objetivo

Neste trabalho temos como objetivo implementar duas estratégias capazes de solucionar o jogo proposto por João.

O jogo consiste em: dada uma sequência de N inteiros, através de várias jogadas, em que cada uma, pode-se escolher um elemento da sequência e excluí-lo, bem como seu antecessor e sucessor. O número escolhido será somado à pontuação do jogador. O objetivo é atingir a pontuação máxima de determinada sequência.

2. ABORDAGEM DO PROBLEMA

Para solucionar o problema, foi exigido que uma das duas maneiras fosse através de programação dinâmica. A outra seria livre, desde que resolvesse o problema.

2.1. Força Bruta

A primeira maneira pensada foi através de força bruta. Desta forma, seriam testadas todas as possibilidades de jogo e aquela que retornasse a maior pontuação seria escolhida como a verdadeira. Esta alternativa foi abandonada devido ao seu alto custo computacional, além da descoberta de outra maneira mais eficiente.

2.2. Memoização

A escolha pelo método da memoização se deve a sua eficiência superior à força bruta.

O código, que envolve uma abordagem recursiva com memoização para calcular a pontuação máxima possível a partir de uma sequência de números. Neste código, é empregada uma abordagem recursiva com memoização para resolver problemas de otimização. A memoização é uma técnica que armazena os resultados de sub-problemas já resolvidos em um array (`memo`) para evitar recálculos, melhorando significativamente a eficiência computacional.

- Função `maiorValor`:
    - Similar ao primeiro código, esta função simplesmente retorna o maior valor entre dois inteiros long long.
- Função `calcularPontuacaoMaximaDFS`:
    - Esta função utiliza uma abordagem de busca em profundidade (DFS) recursiva com memoização para calcular a pontuação máxima possível.
    - Parâmetros:
        - `sequencia`: Array que contém a sequência de números.
        - `tamanho`: Tamanho da sequência.
        - `indice`: Índice atual na sequência.
        - `memo`: Array que armazena as melhores pontuações calculadas até cada índice.
    - Casos Base e Condição de Parada:
        - Se o índice `indice` ultrapassa ou iguala o `tamanho`, retorna zero, indicando o fim da sequência.
        - Se `memo[indice]` já contém um valor diferente de `-1`, significa que a pontuação máxima para este índice já foi calculada anteriormente, então retorna o valor memoizado para evitar recálculos.

    - Escolhas Recursivas:
        - Sem Escolha (`semEscolha`): Calcula a pontuação máxima sem escolher o elemento atual (`sequencia[indice]`), chamando recursivamente `calcularPontuacaoMaximaDFS` para o próximo índice (`indice + 1`).
        - Com Escolha (`comEscolha`): Calcula a pontuação máxima ao escolher o elemento atual (`sequencia[indice]`). Adiciona `sequencia[indice]` à pontuação e, se possível (verifica se `indice + 2 < tamanho`), chama recursivamente `calcularPontuacaoMaximaDFS` para dois índices à frente (`indice + 2`).
    - Atualização do Memo (`memo[indice]`): Armazena o maior valor entre `semEscolha` e `comEscolha` em `memo[indice]`, para memoizar a solução deste subproblema.
    - Retorno: Retorna `memo[indice]`, que contém a pontuação máxima possível até o índice `indice`.
- Função `inicializarMemo`:
    - Esta função inicializa um array de `long long` com tamanho especificado, preenchendo-o com o valor `-1` para indicar que nenhum cálculo foi feito ainda.

Ênfase na Estratégia (`calcularPontuacaoMaximaDFS`):

A função `calcularPontuacaoMaximaDFS` demonstra uma abordagem recursiva com memoização, ideal para resolver problemas de otimização onde a solução ótima pode ser construída a partir de subproblemas menores já resolvidos. A estratégia é eficaz em reduzir o tempo de execução ao evitar recalculos, garantindo que cada subproblema seja resolvido apenas uma vez e seu resultado

seja armazenado para reutilização. Isso não só facilita a implementação de soluções complexas como também melhora significativamente o desempenho computacional, sendo particularmente útil em problemas onde a abordagem direta seria computacionalmente impraticável.

2.3. Programação Dinâmica

A maneira de solucionar o problema através da programação dinâmica foi uma exigência nas especificações do trabalho.

Este código implementa um algoritmo de programação dinâmica para calcular a pontuação máxima de uma sequência de números inteiros. A programação dinâmica é uma técnica de otimização que resolve problemas complexos dividindo-os em subproblemas menores e armazenando os resultados dos subproblemas para evitar cálculos repetitivos.

A principal função do código, “calcularPontuacaoMaximaDinamica”, utiliza um array “pontuacaoMaxima” para armazenar as melhores pontuações possíveis até cada posição na sequência. A função “calcularPontuacaoMaximaDinamica” exemplifica o uso da programação dinâmica para resolver problemas de otimização de forma eficiente. Aqui está como ela realiza cada etapa:

1. Inicialização e Casos Base

Primeiro, verifica se o tamanho da sequência é zero ou um. Se for zero, retorna imediatamente zero, pois não há elementos para calcular a pontuação. Se for um, retorna o único elemento da sequência como a pontuação máxima possível.

1. Uso de Array Auxiliar (`pontuacaoMaxima`)
    - Cria um array `pontuacaoMaxima` para armazenar as melhores pontuações possíveis até cada posição na sequência. Isso permite armazenar e reutilizar resultados intermediários de forma eficiente.

2. Iteração e Decisões Ótimas:
    - Utiliza um loop a partir do terceiro elemento da sequência. Para cada posição `i` na sequência:
        - `escolha1`: Representa a decisão de não escolher o elemento atual (`sequencia[i]`). Neste caso, a pontuação máxima é simplesmente `pontuacaoMaxima[i - 1]`.
        - `escolha2`: Representa a decisão de escolher o elemento atual (`sequencia[i]`). Neste caso, soma o valor do elemento atual com a melhor pontuação possível até duas posições atrás (`sequencia[i] + pontuacaoMaxima[i - 2]`).
3. Atualização da Melhor Pontuação:
    - Para cada posição `i`, atualiza `pontuacaoMaxima[i]` com o valor máximo entre `escolha1` e `escolha2`, usando a função `obterMaiorValor`.
4. Retorno da Solução Ótima:
    - Finalmente, a função retorna `pontuacaoMaxima[tamanho - 1]`, que contém a pontuação máxima possível para toda a sequência.

Esse método eficiente não apenas resolve o problema de forma correta, mas também minimiza o tempo de execução ao evitar cálculos repetidos através do armazenamento de subproblemas já resolvidos. Essa abordagem é fundamental para lidar com problemas complexos onde uma abordagem ingênua seria impraticável devido ao tempo de processamento necessário.

3. ANÁLISE DE COMPLEXIDADE

3.1. Memoização

A função `calcularPontuacaoMaximaDFS` é recursiva e utiliza memoização para evitar recalculos. Vamos analisar seu desempenho:

- Chamadas Recursivas: A função faz duas chamadas recursivas em cada chamada:
    - Uma para o próximo índice (`indice + 1`).
    - Outra para dois índices à frente (`indice + 2`), se possível.
- Memoização: Utiliza um array `memo` para armazenar os resultados de sub-problemas já resolvidos. Isso garante que cada sub problema seja resolvido apenas uma vez.

- Complexidade de Tempo:
    - A função é chamada para cada índice da sequência, até o último elemento (`tamanho - 1`).
    - Para cada chamada, realiza um número constante de operações (cálculos e verificações).
    - Portanto, a complexidade de tempo é dominada pelo número de chamadas recursivas realizadas.

- Recorrência de Tempo:
    - Seja `T(n)` a complexidade de tempo da função `calcularPontuacaoMaximaDFS` para uma sequência de tamanho `n`.
    - A função faz duas chamadas recursivas (`T(n-1)` e `T(n-2)` em média), além de operações de comparação e memoização.
    - A recorrência de tempo pode ser aproximadamente descrita como `T(n)=T(n-1)+T(n-2)+0(1)`
    - Embora a abordagem recursiva com memoização seja eficaz em termos de evitar recalculos, ela pode se tornar impraticável para sequências muito grandes devido à sua complexidade.Portanto, ao utilizar essa estratégia, é importante considerar o tamanho máximo da entrada para garantir que o tempo de execução seja aceitável para os requisitos do problema.

3.2. Programação Dinâmica

Antes do loop principal, há duas verificações condicionais (if(tamanho==0) e if(tamanho==1)), ambas operando em tempo constante O(1).

A inicialização dos primeiros elementos do array “pontuacaoMaxima” também é O(1). As operações realizadas são em tempo constante O(1), incluindo atribuições simples e chamadas para “obterMaiorValor”. Assim, o loop executa tamanho - 2 vezes, resultando em complexidade temporal O(tamanho). Portanto, a complexidade temporal total da função “calcularPontuacaoMaximaDinamica” é O(tamanho), onde tamanho é o número de elementos na sequência passada como parâmetro.

Tabela de comparação de tempo Programação Dinâmica X Memoização

Entrada	Programação Dinâmica	Memoização
10	1238.000 microssegundos	1579.000 microssegundos
1000	1182.000 microssegundos	1665.000 microssegundos
100.000	1357.000 microssegundos	21691.000 microssegundos

4. CONCLUSÃO

Primeiramente, vale ressaltar a importância da análise das especificações iniciais do trabalho. Dadas as condições de tamanho das entradas, foi possível elaborar três soluções que conseguem resolver o problema proposto - força bruta, memoização e programação dinâmica.

Com isso, houve um desenvolvimento de habilidades que nos capacitaram no uso da programação dinâmica. Sendo uma técnica fundamental e poderosa em ciência da computação, utilizada para resolver uma ampla gama de problemas de otimização. Ao dividir problemas complexos em subproblemas menores e resolver

cada um apenas uma vez, a programação dinâmica permite alcançar soluções eficientes que de outra forma seriam computacionalmente inviáveis.

A programação dinâmica oferece várias vantagens significativas:

1. **Redução de Complexidade:** Permite resolver problemas complexos dividindo-os em partes menores, mais gerenciáveis, e resolvendo cada subproblema de forma ótima.
2. **Otimização de Recursos:** Ao armazenar resultados intermediários (memoização), evita recalculá-los repetidamente, melhorando drasticamente a eficiência computacional.
3. **Aplicabilidade Universal:** Pode ser aplicada a uma ampla variedade de problemas, desde problemas clássicos de otimização até questões práticas em algoritmos e estruturas de dados.
4. **Versatilidade e Elegância:** Proporciona soluções elegantes e eficientes para problemas que, de outra forma, exigiriam soluções mais complexas e menos eficientes.

No entanto, é importante notar que a programação dinâmica exige um entendimento profundo do problema e da estrutura de subproblemas para ser aplicada corretamente. Além disso, enquanto muitos problemas podem ser resolvidos de maneira eficiente com programação dinâmica, há casos onde outras abordagens podem ser mais adequadas dependendo das restrições e características específicas do problema.

Em resumo, a programação dinâmica continua a ser uma ferramenta essencial no arsenal de qualquer desenvolvedor ou cientista da computação, permitindo resolver problemas desafiadores de forma eficiente e elegante através da decomposição e otimização de subproblemas.

5. REFERÊNCIAS

[CORMEN et al., 2012] CORMEN, T. H., Leiserson, C., Rivest, R., and Stein, C (2012). Algoritmos: teoria e prática. LTC.

Acesso em: 16/07/2024
https://www.geeksforgeeks.org/program-for-factorial-of-a-number/

Acesso em: 16/07/2024
https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/