Universidade Federal
de São João del-Rei

PROJETO E ANÁLISE DE ALGORITMOS

Trabalho Prático II - Jogo da sequência

Diego Augusto Lucas Costa

Julho de 2024

1 Introdução

Em mais uma noite de inverno, João, muito entediado, decide criar um jogo com uma sequência composta por n inteiros. No jogo, o jogador pode fazer várias jogadas, escolhendo em cada uma um elemento da sequência para exclusão, junto com seus vizinhos imediatos. A pontuação do jogador é incrementada com o valor do elemento escolhido a cada jogada.

No contexto do jogo da sequência, o desafio de maximizar a pontuação ao excluir elementos adjacentes pode ser analogamente comparado a uma aplicação do mundo real, como o planejamento de plantio em uma fazenda. Nesse exemplo, cada parcela de terra representa um "elemento"onde podem ser plantadas culturas, cada uma com um potencial de lucro determinado. Assim como no jogo, onde a exclusão de um elemento afeta seus vizinhos na sequência, decidir quais parcelas plantar afeta diretamente as parcelas adjacentes devido a restrições de nutrientes ou outras condições agronômicas. O objetivo é maximizar o lucro total da colheita, garantindo que as decisões de plantio respeitem a limitação de não plantar em parcelas adjacentes.

O objetivo deste trabalho é ajudar João a alcançar a maior pontuação possível no jogo da sequência. Para isso, será desenvolvido um algoritmo que faz escolhas estratégicas nas jogadas para maximizar a pontuação. Serão exploradas duas abordagens distintas: uma utilizando programação dinâmica e outra por uma estratégia alternativa, que, ao longo do documento, serão comparadas quanto à sua complexidade e desempenho.

2 Desenvolvimento do Projeto

Para o desenvolvimento do projeto, foi necessário implementar duas estratégias: a dinâmica (Dyn) e a alternativa (Alt), como já foi dito anteriormente. Ambas as soluções encontram a resposta correta, porém elas utilizam ideias diferentes.

2.1 Solução Alternativa

```
int max(int a, int b) {
    if(a > b) return a;
    else return b;
}

int alternative(int* nums, int i) {
    if (i < 0) {
        return 0;
    }
    return max(alternative(nums, i - 2) + nums[i], alternative(nums, i - 1));
}
```

Figura 1: Código da Solução Alternativa

A função `alternative` (mostrada na figura 1) inicia buscando a melhor estratégia partindo dos elementos `i-1` e `i-2`, respeitando a regra de que ao selecionar um elemento, os adjacentes diretos não podem ser escolhidos. Em outras palavras, para cada elemento `i`, o algoritmo considera duas opções: selecionar o elemento atual `i` ou não selecioná-lo.

Ao selecionar o elemento atual `i`, o jogador obtém a pontuação deste elemento somada a pontuação máxima obtida a partir do elemento `i-2`. Caso o elemento atual não seja selecionado, a pontuação acumulada é a pontuação máxima obtida a partir `i-1`.

O algoritmo compara as duas opções usando a função auxiliar `max`, escolhendo a opção que proporciona a maior pontuação acumulada. Esse processo é repetido de forma recursiva para todos os elementos da sequência, garantindo que cada possível combinação seja analisada. A base da recursão é atingida quando o índice `i` é menor que zero, retornando 0, já que não há mais elementos para considerar. Assim, a solução final é construída, assegurando a maximização da pontuação total.

Embora este método resolva o problema, ele não é muito eficiente para grandes sequências. Isso ocorre devido às inúmeras chamadas recursivas, similar à resolução da sequência de Fibonacci usando recursão, onde a redundância das chamadas pode levar a um tempo de execução exponencial. Portanto, para sequências maiores, abordagens como programação dinâmica seriam mais adequadas para otimizar o desempenho.

2.2 Solução Dinâmica

```
int dynamic(int* nums, int i) {
    if (i <= 0) {
        return 0;
    }
    int previous = 0;
    int next = 0;
    for(; i >= 0; i--) {
        int guess = previous+nums[i];
        previous = next;
        if(guess > next) {
            next = guess;
        }
    }
    return next;
}
```

Figura 2: Código da Solução Dinâmica

A solução dinâmica (mostrada na figura 2) consiste em utilizar os conceitos definidos na forma alternativa, mas utilizando uma técnica de programação dinâmica chamada de tabulação para resolver os subproblemas.

A tabulação adota uma abordagem iterativa, resolvendo primeiro os subproblemas menores e armazenando suas soluções em uma tabela (geralmente uma matriz ou arranjo, mas nesse caso, as variáveis `previous` e `next`). Em seguida, ela usa esses valores armazenados para calcular as soluções de subproblemas maiores, chegando gradualmente à solução final. A natureza iterativa da tabulação normalmente envolve o uso de loops para preencher a tabela de forma sistemática [1].

Dessa forma, a cada etapa do algoritmo, são consideradas duas soluções candidatas: excluir a casa atual ou excluir a anterior. O ramo da casa atual tem valor `nums[i] + previous` onde `previous` é o valor do penúltimo passo e `nums[i]` o da casa atual. Já o ramo da casa anterior tem o valor `next`, onde `next` é o valor do último passo. A partir disso, é trivial observar que no passo subsequente, a melhor escolha será considerada pensando se a decisão feita anteriormente foi a correta, e será ajustada de acordo, considerando os valores do penúltimo e último passo.

3 Análises de complexidades

3.1 Solução alternativa

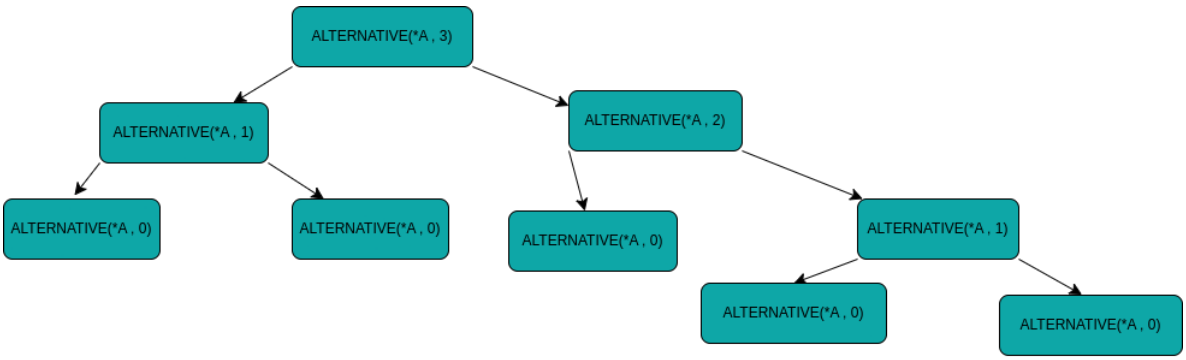

Figura 3: Exemplo de árvore de recursão

A complexidade da função `alternative` pode ser analisada com base na sua estrutura recursiva. O algoritmo utiliza recursão para explorar todas as possíveis combinações de elementos que podem ser escolhidos ou não. Para cada chamada de `alternative(nums, i)`, o algoritmo faz duas chamadas recursivas: `alternative(nums, i - 2)` e `alternative(nums, i - 1)`. Isso gera uma árvore de chamadas recursivas que pode ser visualizada como uma árvore binária completa. A cada nível da árvore de recursão, o número de chamadas recursivas duplica, resultando em uma complexidade de tempo exponencial. Para calcular a complexidade, considera-se a profundidade da árvore de recursão como `n`. A árvore de recursão possui aproximadamente (2^n) nós. Portanto, a complexidade de tempo é: $O(2^n)$.

A figura 3, que ilustra uma árvore de recursão para uma sequência de apenas 3 elementos, já demonstra uma quantidade significativa de chamadas recursivas, evidenciando a natureza exponencial deste algoritmo. Essa complexidade de tempo exponencial torna o algoritmo ineficiente para grandes valores de `n`, semelhante ao problema clássico da sequência de Fibonacci recursiva.

3.2 Solução dinâmica

Como explicado anteriormente, a solução dinâmica tem que considerar apenas dois estados, o último (`next`) e o penúltimo (`previous`), e por isso, não precisa navegar por vários subconjuntos do conjunto de solução. Graças à essa simplicidade, o algoritmo executa em tempo linear, pois percorre o vetor elemento por elemento, executando uma única comparação por passo, como pode ser visto na figura 4, resultando em uma complexidade de $O(n)$.

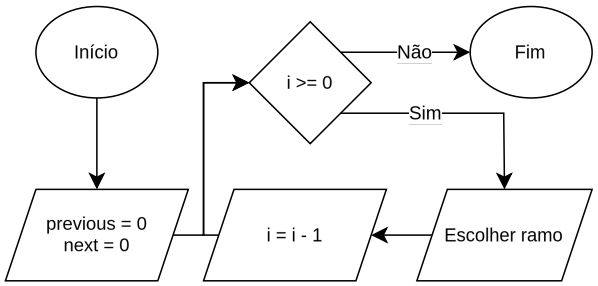

Figura 4: Fluxograma da solução dinâmica

4 Testes e resultados

Os testes a seguir foram sistematicamente executados numa máquina Arch Linux com processador AMD Ryzen 7 5700U, sendo o código compilado utilizando o comando `make`. Além disso, cada entrada foi resolvida 10 vezes para obter uma média mais precisa do tempo de execução. As entradas tiveram seus conteúdos gerados aleatoriamente, sendo geradas entradas de tamanho 1 a 40 para a solução alternativa, e de 10K até 100K com um passo de 2K para a solução dinâmica. Essa separação é devido ao fato de que a solução dinâmica é tão otimizada que produz valores muito próximos à margem de erro do tipo `double`, e por isso foi necessário um número enorme de entradas para que seja possível visualizar os tempos de execução com clareza.

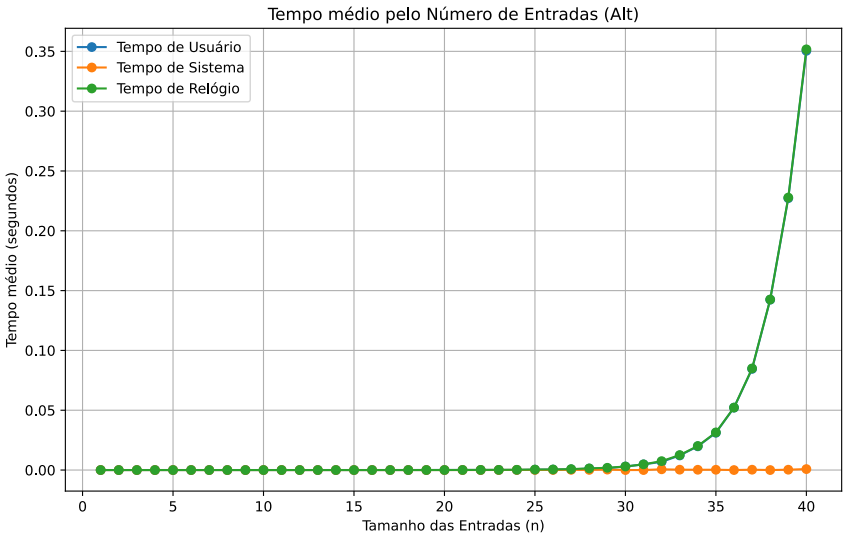

Figura 5: Tempos médios da solução alternativa

Os testes e resultados da solução alternativa revelaram um comportamento já esperado, especialmente ao analisar o gráfico na Figura 5. A solução alternativa, que

utiliza recursão para explorar todas as possíveis combinações de elementos, demonstrou claramente um crescimento exponencial do tempo de execução à medida que o tamanho da sequência aumentava. Este comportamento exponencial é visualmente evidente no gráfico, onde o tempo de execução cresce abruptamente com o aumento de `n`. Essa análise confirma a análise de complexidade feita na seção anterior, onde foi teorizada a complexidade $O(2^n)$. A ineficiência da abordagem recursiva pura para problemas desse tipo evidencia a necessidade de soluções mais otimizadas.

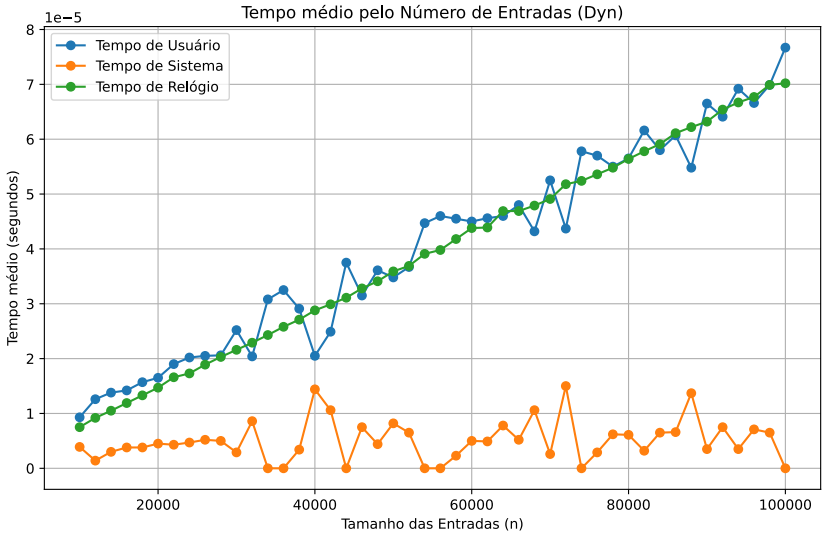

Figura 6: Tempos médios da solução dinâmica

Em contraste, os resultados da solução dinâmica apresentaram um comportamento significativamente melhor. Demonstrando o desempenho esperado com complexidade linear, a solução dinâmica se destacou, mesmo quando testada com entradas muito grandes, como sequências de até 100000 inteiros. Ao contrário da solução alternativa, cujo tempo de execução cresceu exponencialmente, a solução dinâmica manteve tempos de execução expressivamente bons. Essa eficiência considerável, visível no gráfico da Figura 6, confirma a superioridade da abordagem de programação dinâmica para este tipo de problema, destacando sua capacidade de lidar com grandes volumes de dados de maneira eficaz.

Os tempos de usuário e sistema são duas diferentes formas de medir o tempo de execução. O tempo de sistema mede quanto tempo o sistema operacional executou tarefas para o programa, como ler arquivos, alocar memória, etc. Enquanto o tempo de usuário mede a quantidade de tempo que o programa utilizou a CPU.

Utilizamos as bibliotecas `getrusage.h` e `gettimeofday.h` para avaliar o tempo de execução completo do nosso algoritmo, pois oferecem uma forma de consultar os valores dos tempos de usuário e de sistema e do tempo de relógio. Isso representa

uma vantagem significativa em comparação com a biblioteca `gettimeofday.h`, que só registra o tempo físico decorrido. Contudo, a soma dos valores de usuário e sistema resulta aproximadamente no valor retornado pela biblioteca `gettimeofday.h`, sendo mais precisa que a outra para tempos pequenos, devido à imprecisões do tipo de dado `double`.

5 Conclusão

Em resumo, foram encontradas duas soluções para o problema apresentado, cujo desempenho e tempo de execução dependem do tamanho das entradas. Além disso, foi mostrado o quão eficaz é a utilização de técnicas de programação dinâmica para a análise de otimização da execução de algoritmos como esse, por meio de testes sistemáticos executados. Dito isso, espera-se que seja evidente por meio dos testes que a solução dinâmica é significativamente mais eficiente que a alternativa.

Com base no que foi apresentado, enfatiza-se a importância de explorar novas formas de resolver problemas propostos, destacando a necessidade de utilizar técnicas de melhoria e paradigmas que otimizem algoritmos. Inicialmente, a solução mais trivial para o desenvolvimento foi a alternativa recursiva, dada a semelhança do problema com a conhecida sequência de Fibonacci, previamente discutida. No entanto, ao implementar essa abordagem, tornou-se evidente que otimizações eram necessárias para obter melhores resultados. A partir dessa experiência, fica claro a necessidade de conhecer e aplicar estratégias distintas. Isso é crucial para que, ao enfrentar cenários onde certas resoluções não são eficientes, seja possível recorrer a alternativas mais eficazes.

Por fim, o problema de descobrir a maior pontuação possível para o jogo da sequência enfrentado por João agora possui duas soluções possíveis, possibilitando que ele passe felizes noites executando o algoritmo alternativo ou criando entradas gigantescas para o algoritmo dinâmico.

Bibliografia

[1] Frank. *Dynamic Programming in C++ - Techniques and Insights — frankalcantara.com*. `https://frankalcantara.com/dynamic-programming/`. [Accessed 15-07-2024].