Ciência da Computação
Projeto e Análise de Algoritmos

Documentação Trabalho Prático 2
A Jogada Perfeita

Lucas Eduardo Bernardes de Paula

Messias Feres Curi Melo

2024.1

1 Introdução

João é uma pessoa que não suporta a monotonia. Sempre que o tédio se aproxima, ele busca maneiras de se entreter inventando novos jogos. Em uma longa e fria noite de inverno, ele teve a ideia de um jogo intrigante e desafiador. Este jogo envolve uma sequência de números inteiros, e o jogador tem a oportunidade de realizar várias jogadas. Em cada jogada, é possível escolher um elemento da sequência, que será excluído juntamente com seus elementos vizinhos. Cada exclusão rende ao jogador pontos equivalentes ao valor do elemento escolhido. O objetivo é conseguir a maior pontuação possível.

Com seu espírito perfeccionista, João quer sempre atingir o máximo de pontos em cada partida. Para isso, ele precisa de um programa que calcule a pontuação máxima possível, considerando todas as combinações possíveis de jogadas. Este desafio requer não apenas habilidade lógica, mas também uma abordagem eficiente para garantir que o resultado seja obtido de maneira rápida e precisa. A Figura 1 exemplifica um exemplo do tabuleiro do jogo:

7	2	3	6	3	4

Figure 1: Exemplo da disposição dos pontos em um tabuleiro do jogo

Esse problema representa um desafio único, pois as abordagens tradicionais de otimização não são adequadas para esse tipo específico de problema, onde a exclusão de elementos implica a remoção obrigatória de seus vizinhos imediatos. Em vez de uma simples seleção de valores máximos, é necessário considerar a interdependência entre os elementos adjacentes para garantir a pontuação máxima. Para superar esse obstáculo, utilizamos uma abordagem de Programação Dinâmica que permite armazenar e comparar de forma eficiente os pontos acumulados ao longo da sequência, garantindo que a exclusão de um elemento e seus vizinhos seja contabilizada corretamente. Alternativamente, uma estratégia de força bruta pode ser aplicada, embora seja menos eficiente, pois examina todas as combinações possíveis para encontrar a pontuação máxima, resultando em um alto custo computacional.

Faz-se necessário, portanto, desenvolver um programa que encontre a solução de maneira otimizada ao mesmo tempo que garanta a facilidade de manutenção do código, o que pode ser alcançado por meio de boas práticas de programação e documentação adequada, assim como demonstrado adiante.

2 Estruturando a Solução

Levando em consideração o problema apresentado, é de grande importância uma boa estruturação de código. Por isso, é imprescindível definir como será feita a estruturação antes de começar a programar definitivamente.

Como é visto no fluxograma da Figura 2, o código inicia a execução e recebe os argumentos de entrada, logo após ele cria um array com os pontos de cada um dos índices do array. Depois disso, de acordo com a estratégia recebida pela linha de comando, ele executa o algoritmo para encontrar a maior pontução e por fim escreve no arquivo a pontuação máxima obtida e o tempo de execução é impresso no terminal.

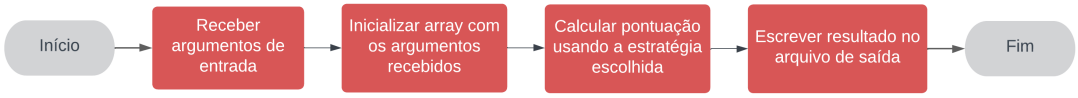

Figure 2: Fluxograma de funcionamento

2.1 Leitura de arquivos

Para receber as informações da sequência de João, foi utilizado como base um arquivo padrão que contém na primeira linha o inteiro N, representando o número de elementos na sequência. Na segunda linha, são fornecidos os N inteiros $a_1, a_2, \ldots, a_n$ que compõem a sequência. A compilação do código é feita a partir de um `Makefile`, que define os passos de compilação dos arquivos e cria um executável final automaticamente ao inserir o comando `make` no diretório do projeto. Na execução, basta utilizar o `./tp2` que foi gerado pelo `Makefile`, seguido pela definição da estratégia desejada e do arquivo de entrada, como representado na linha de comando:

./tp2 <estrategia> entrada.txt

Para receber esses dados é necessário utilizar os parâmetros *argc* e *argv*, recebidos pela função *main*(), verificando se as entradas são válidas para o funcionamento correto do programa.

Após obter o endereço do arquivo de entrada, é efetuada a leitura completa do mesmo utilizando a função `fscanf()`, pertencente à biblioteca `stdlib.h`.

2.2 Estruturar o tabuleiro do jogo

A estruturação do tabuleiro do jogo é realizada a partir da leitura do arquivo, onde um novo array é gerado com base no tamanho especificado e nos pontos atribuídos a cada um dos índices.

2.3 Solução proposta

Para resolver o problema, foram desenvolvidas duas estratégias distintas: uma baseada em programação dinâmica e outra utilizando um algoritmo de força bruta. A abordagem de programação dinâmica foi projetada para garantir eficiência ao calcular a pontuação máxima. Isso é alcançado por meio de uma relação de recorrência que permite armazenar e reutilizar resultados intermediários, evitando cálculos redundantes. Na figura 3, é possível visualizar um exemplo dessa prática.

Figure 3: Exemplo de pontuação máxima obtida

Por outro lado, a abordagem de força bruta, embora menos eficiente, assegura que sempre seja encontrada uma solução, explorando todas as combinações possíveis de jogadas. Essa técnica exaustiva garante a completude da solução, apesar de sua maior demanda computacional.

O funcionamento de ambas as estratégias é descrito com mais detalhes na seção 4 da documentação

2.4 Salvar os resultados no arquivo de saída

A pontuação máxima encontrada é salva em um arquivo de saída chamado `saida.txt`, utilizando a função `fprintf` da biblioteca `stdio.h`.

Após a conclusão do programa, todas as estruturas de dados alocadas dinamicamente são liberadas, garantindo que não ocorram vazamentos de memória indesejados.

3 Estrutura de dados

Para modelar o problema foi utilizada a estrutura de dados *array*, a qual oferece benefícios claros em termos de acesso direto e manipulação eficiente dos elementos.

A estrutura de *array* é ideal para representar a sequência de números porque permite acesso aleatório em tempo constante, o que é essencial para as operações frequentes de exclusão e cálculo de pontos em cada jogada.

Utilizar um *array* facilita a iteração sobre a sequência e a aplicação da lógica de programação dinâmica para calcular a pontuação máxima, mantendo a simplicidade e clareza no gerenciamento dos dados. Esse modelo não só simplifica a implementação do algoritmo, mas também garante um desempenho otimizado, crucial para lidar com o limite superior de 10^5 elementos na sequência. Dessa forma, a escolha do *array* como estrutura de dados principal assegura uma solução eficiente e de fácil compreensão, atendendo aos requisitos do problema de maneira eficaz.

3.1 Modelagem Alternativa com Árvore de Decisão

É possível também modelar o problema utilizando uma árvore de decisão para representar as decisões de exclusão de elementos da sequência. Cada nó da árvore representa a decisão de excluir ou não um elemento específico da sequência. A raiz da árvore representa a decisão sobre o primeiro elemento. A partir de cada decisão, há duas ramificações: uma que representa a decisão de excluir o próximo elemento possível (considerando que elementos adjacentes também devem ser excluídos) e outra que representa a decisão de pular o elemento atual e considerar o próximo elemento na sequência, conforme o exemplo da Figura 4. Dessa forma, a árvore de decisão nos permite visualizar todas as possíveis sequências de jogadas e calcular a pontuação máxima que pode ser alcançada.

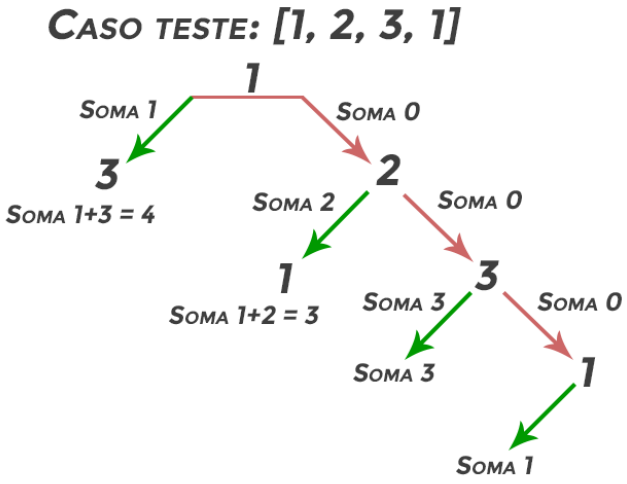

Figure 4: Exemplo de árvore de decisão

4 Estratégia de resolução

Conforme visto em seções anteriores, foram elaboradas duas estratégias distintas para encontrar a maior pontuação possível: um algoritmo de programação dinâmica (iterativo) e um algoritmo de força bruta (recursivo).

4.1 Força Bruta

A estratégia mais direta para resolver o problema de João é testar todas as combinações possíveis de jogadas na sequência de inteiros, registrando a maior pontuação obtida durante a busca. Esta abordagem recursiva examina cada elemento da sequência, comparando duas opções: ignorar o elemento atual e avançar para o próximo, ou selecionar o elemento atual, somando sua pontuação e pulando dois elementos à frente, conforme a regra de exclusão dos vizinhos. A função de busca começa no primeiro elemento da sequência e segue as instruções do fluxograma da estratégia de força bruta. Esse processo se repete até que todas as combinações possíveis sejam testadas, retornando ao nível anterior na pilha de execução para explorar novas possibilidades de jogadas.

Embora o algoritmo seja simples e direto, a abordagem de força bruta torna-se ineficiente para sequências longas, pois o número de chamadas recursivas cresce exponencialmente, tornando o algoritmo impraticável. No entanto, essa solução alternativa garante a correção do problema em casos de menor escala.

O fluxograma na Figura 5 ilustra o funcionamento da solução por força bruta.

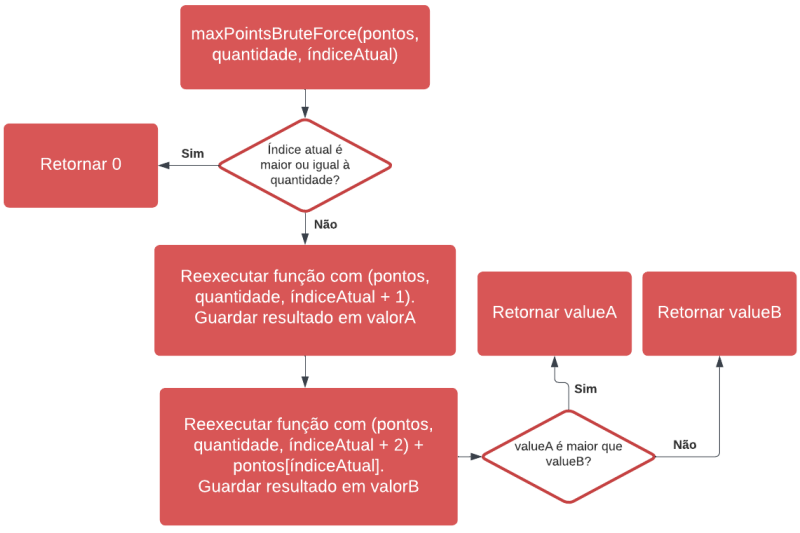

Figure 5: Fluxograma do funcionamento da solução por força bruta

4.2 Programação dinâmica

Um algoritmo de programação dinâmica consiste em dividir o problema em subproblemas menores e solucioná-los em cada iteração, armazenando os resultados obtidos para auxiliar na resolução dos próximos subproblemas e assim conseguir encontrar a solução global ótima para o problema.

Para resolver o desafio proposto por João, onde ele busca a pontuação máxima ao excluir elementos de uma sequência de inteiros e seus vizinhos, utilizamos uma abordagem eficiente de programação dinâmica. Inicialmente, tratamos casos triviais onde o número de elementos é zero, um ou dois.

Em seguida, criamos um array que armazena as pontuações máximas acumuladas até cada posição da sequência. Cada posição do array é preenchida utilizando uma relação de recorrência, onde decidimos entre não escolher o elemento atual ou escolhê-lo e somar à melhor pontuação possível até duas posições anteriores.

Essa abordagem garante que todas as subsoluções sejam aproveitadas para construir a solução global ótima, resultando na pontuação máxima possível ao final do processo. Além disso, cuidamos para liberar a memória alocada, garantindo a eficiência do algoritmo.

O fluxograma na Figura 6 ilustra o funcionamento da solução por programação dinâmica.

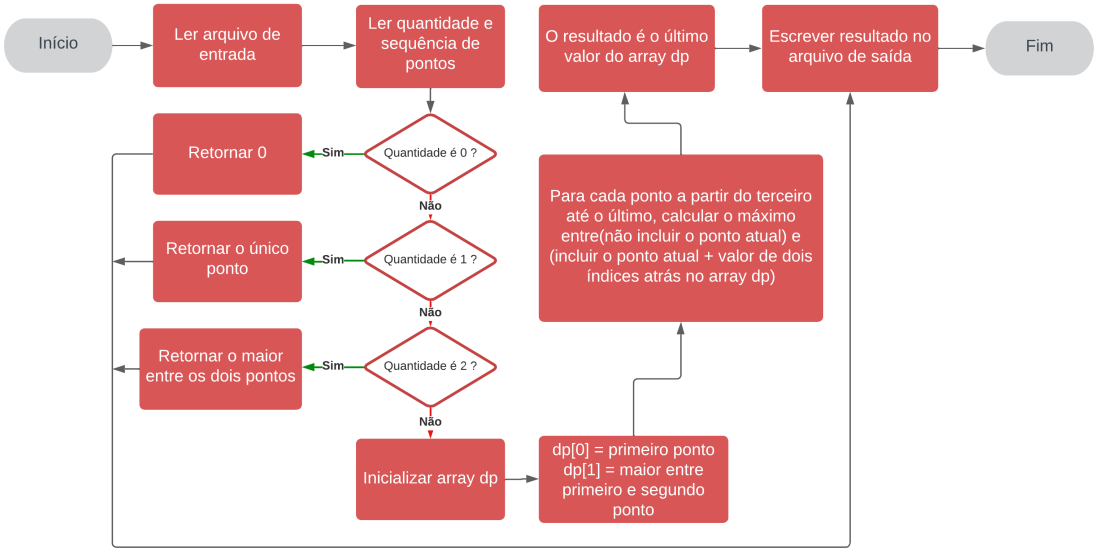

Figure 6: Fluxograma do funcionamento da solução por programação dinâmica

5 Análise matemática

Os dois algoritmos apresentados na seção anterior possuem funcionamentos notavelmente diferentes, o que resulta em tempos de execução distintos. Por isso, é importante realizar uma análise matemática detalhada do funcionamento de ambos.

5.1 Análise do algoritmo de força bruta

Tal como foi mencionado anteriormente, o algoritmo utiliza chamadas recursivas para testar todas as possibilidades de maximização de pontos, sendo chamado de duas maneiras principais: ignorando o elemento atual e movendo para o próximo índice ou somando o valor do elemento atual e pulando um índice. Essa abordagem resulta em uma árvore de chamadas recursivas onde cada nó gera dois filhos, levando a uma exploração completa de todas as combinações possíveis.

A relação de recorrência que descreve a complexidade do algoritmo é:

$$T(n) = T(n-1) + T(n-2)$$

Esta é a mesma da sequência de Fibonacci. Na sequência de Fibonacci, cada termo é a soma dos dois termos anteriores, o que resulta em um crescimento exponencial do número de termos conforme n aumenta. De forma similar, o algoritmo de força bruta para maximização de pontos possui uma estrutura onde cada chamada recursiva gera duas novas, resultando em crescimento exponencial.

Devido a essa estrutura recursiva, a complexidade temporal do algoritmo é exponencial, especificamente:

$$T(n) = O(2^n)$$

Para cada índice, há duas possibilidades a serem exploradas, o que significa que o número total de chamadas recursivas cresce exponencialmente com o aumento do tamanho da entrada. Isso pode ser visualizado como uma árvore binária de chamadas recursivas com profundidade aproximadamente igual a n e número total de nós aproximadamente 2^n.

Em resumo, a complexidade exponencial do algoritmo de força bruta é uma consequência direta das recursões que exploram todas as combinações possíveis, similar à sequência de Fibonacci.

5.2 Análise do algoritmo de programação dinâmica

Diferentemente do anterior, o algoritmo de programação dinâmica não necessita que todas as possibilidades sejam testadas; a solução é construída gradativamente con-

forme cada elemento da sequência é analisado, resultando na seguinte função de complexidade:

$$f(n) = n$$

A ordem de complexidade da estratégia é facilmente constatada nesse caso, sendo classificada como linear:

$$f(n) = O(n)$$

A complexidade de tempo é $O(n)$, pois as operações dominantes, incluindo verificações iniciais e a inicialização e preenchimento do array, são lineares em relação ao tamanho da entrada. A complexidade de espaço também é $O(n)$, devido à alocação do array 'dp', que armazena as pontuações máximas acumuladas até cada posição da sequência.

Assim, a estratégia de programação dinâmica oferece uma eficiência significativa em comparação com a abordagem de força bruta, que possui complexidade exponencial, tornando-a impraticável para sequências longas.

6 Testes

Para monitorar o tempo de execução total de cada algoritmo na busca de soluções, foi utilizada a biblioteca getrusage.h, que permite acompanhar especificamente o tempo de CPU, evitando assim inconsistências decorrentes do uso do tempo físico como parâmetro de medição.

Os testes estiveram presentes em todas as etapas do desenvolvimento do software, desde o funcionamento das funções mais elementares até a medição do tempo de execução da função para encontrar a solução do problema, garantindo uma aplicação performática e de fácil manutenção. Os dados de tempo obtidos foram utilizados para criar gráficos informativos sobre a eficiência das duas estratégias, proporcionando uma análise mais detalhada das informações coletadas.

6.1 Tempo do algoritmo de força bruta

O algoritmo de força bruta foi testado com sequências de inteiros de tamanhos variados. Os resultados mostraram que o tempo de execução aumenta exponencialmente com o tamanho da entrada, tornando essa abordagem inviável para sequências maiores. A Figura 7 ilustra esse comportamento.

Embora o algoritmo de força bruta explore todas as combinações possíveis e garanta uma solução, seu custo computacional é muito alto. Para sequências pequenas, o tempo de execução é aceitável, mas cresce exponencialmente com o aumento do tamanho da entrada, conforme a complexidade $O(2^n)$.

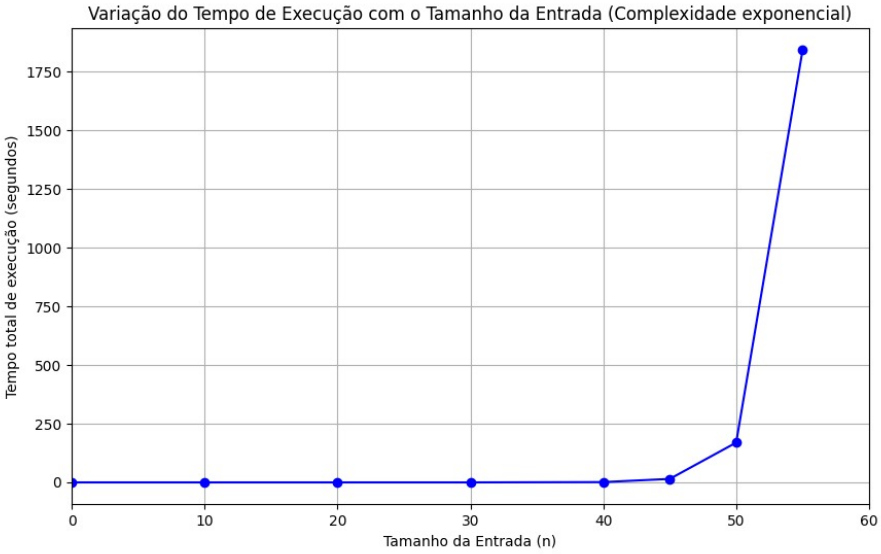

Figure 7: Gráfico do tempo de execução do algoritmo de força bruta

Comparando com a programação dinâmica, a força bruta se mostra inadequada para sequências maiores, devido ao tempo de execução excessivo.

6.2 Tempo do algoritmo de programação dinâmica

Em contrapartida, o algoritmo de programação dinâmica apresentou desempenho superior, lidando eficientemente com sequências de até 100.000 elementos, sem exceder 0,02 segundos de tempo de execução.

Conforme demonstrado na Figura 8, o tempo de execução do algoritmo de programação dinâmica aumenta linearmente com o tamanho da entrada, confirmando a complexidade teórica O(n). Esse comportamento linear deve-se à eficiência da programação dinâmica em evitar cálculos redundantes ao armazenar e reutilizar resultados intermediários.

Portanto, a programação dinâmica não só reduz o tempo de execução, mas também garante uma solução escalável e eficiente para o problema, tornando-se inviável utilizar a abordagem de força bruta para sequências maiores.

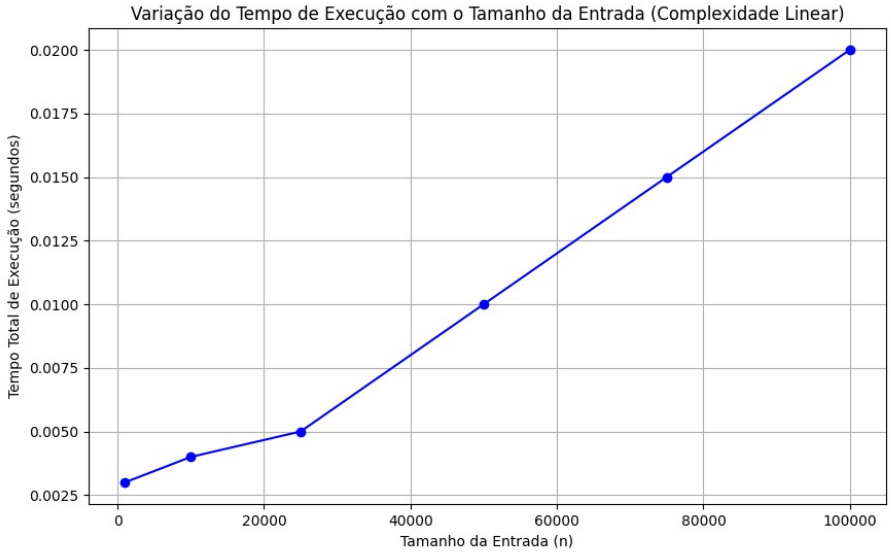

Figure 8: Gráfico do tempo de execução da programação dinâmica

7 Conclusão

O desenvolvimento deste projeto destacou a importância de analisar e comparar diferentes abordagens para a resolução de problemas complexos, especialmente no contexto de otimização e eficiência computacional. A implementação do algoritmo de força bruta, embora garantisse a correção da solução, mostrou-se inviável para grandes sequências devido à sua complexidade exponencial, conforme evidenciado pelos testes realizados. Em contrapartida, a abordagem de programação dinâmica demonstrou ser uma solução robusta e eficiente, capaz de lidar com grandes volumes de dados de forma linear.

A eficiência alcançada pela programação dinâmica se deve à sua capacidade de reutilizar resultados intermediários, evitando cálculos redundantes e, assim, garantindo uma solução escalável. Este estudo evidenciou que a escolha apropriada de estruturas de dados e algoritmos pode influenciar significativamente o desempenho de uma aplicação, impactando diretamente a experiência do usuário e os recursos computacionais necessários.

Portanto, ao abordar problemas de otimização como o descrito, é crucial explorar múltiplas estratégias e realizar testes exaustivos para identificar a solução mais adequada. A implementação de algoritmos eficientes não só melhora o tempo de execução, mas também contribui para a sustentabilidade de sistemas maiores e mais

complexos.

Em conclusão, a aplicação de boas práticas de programação e uma análise cuidadosa dos métodos disponíveis permitiram desenvolver uma solução eficaz para o desafio proposto, confirmando a relevância da programação dinâmica em cenários onde a eficiência e a escalabilidade são essenciais. Continuar a explorar e validar diferentes abordagens será sempre uma prática recomendada no desenvolvimento de software, garantindo soluções robustas e otimizadas para os problemas computacionais mais diversos.

8 Referências

1. Thomas H. Cormen. *Algoritmos: Teoria e prática*. LTC, 2012.
2. Nivio Ziviani. Projetos de Algoritmos em C e Pascal. 2. ed. São Paulo: Cengage Learning, 2006.
3. Abrahim Ladha. *CS 3510 Algorithms Lecture 9: Dynamic Programming I*. 2023.