João Victor Ottoni Garcia e Rafael Carvalho Avidago Geraldo

Trabalho Prático 2

Brasil - São João del Rei/MG

2024

João Victor Ottoni Garcia e Rafael Carvalho Avidago Geraldo

Trabalho Prático 2

Trabalho de Projeto e Análise de Algoritmos relacionado ao desenvolvimento de programa que dada uma sequência de inteiros, determine a pontuação máxima possível removendo elementos e seus vizinhos.

Universidade Federal De São João del Rei – UFSJ

Faculdade de Ciência da Computação

Orientador: Leonardo Chaves Dutra da Rocha

Brasil - São João del Rei/MG

2024

João Victor Ottoni Garcia e Rafael Carvalho Avidago Geraldo
Trabalho Prático 2 / João Victor Ottoni Garcia e Rafael Carvalho Avidago Geraldo. – Brasil - São João del Rei/MG, 2024-
15p. : il. (algumas color.) ; 30 cm.

Orientador: Leonardo Chaves Dutra da Rocha

Trabalho Acadêmico – Universidade Federal De São João del Rei – UFSJ
Faculdade de Ciência da Computação , 2024.

1. Maior somatório. 2. Força Bruta. 2. Programação Dinâmica. I. Leonardo Rocha. II. UFSJ. III. Ciência da Computação. IV. Trabalho Prático 2

Sumário

1 Introdução

No desafio apresentado, sugere-se a criação de um algoritmo que lide com problemas complexos de otimização. O objetivo do programa é calcular a pontuação máxima possível ao eliminar elementos de uma sequência de números inteiros, considerando que cada eliminação inclui também a exclusão dos números adjacentes. Este desafio é importante não só como um exercício acadêmico de algoritmos, mas também tem utilidade em campos como gerenciamento de recursos, alocação de tarefas e jogos de estratégia, que exigem decisões ideais para maximizar benefícios ou pontuações específicas.

Por exemplo, em administração de recursos, a exclusão de um recurso pode impactar recursos vizinhos, demandando uma estratégia para otimizar a eficiência total. Na atribuição de tarefas, pode ser preciso selecionar quais atividades realizar e quais dispensar para aumentar a eficiência ou os efeitos.

A tarefa é criar duas estratégias diferentes: uma utilizando Programação Dinâmica e a outra alternativa, ambas com o objetivo de maximizar a pontuação alcançada. A técnica de Programação Dinâmica emprega uma forma otimizada para solucionar subproblemas de maneira eficaz, guardando resultados intermediários para evitar repetições de cálculos. A estratégia alternativa proporciona uma solução distinta que pode incluir técnicas heurísticas ou métodos de força bruta, dependendo da abordagem selecionada.

O software precisa ler os dados de um arquivo contendo uma sequência de inteiros, analisar essa sequência para determinar a maior pontuação possível e criar um arquivo com o resultado. Para avaliar o desempenho das estratégias implementadas, é necessário usar as funções getrusage e gettimeofday para medir o tempo de execução, tempo de CPU e memória máxima utilizada durante a execução do programa.

2 Conceitos Abordados No Trabalho

2.1 Programação Dinâmica

A programação dinâmica é um método de otimização utilizado para resolver problemas complexos. Ela funciona dividindo os problemas em subproblemas menores e resolvendo cada um desses subproblemas, uma vez armazenando os resultados. A programação dinâmica é útil para problemas que necessitam de uma subestrutura otimizada e que existe a resolução de problemas sobrepostos.

A programação dinâmica é comumente utilizada em várias áreas da ciência da computação devido à sua eficiência e capacidade de lidar com problemas complexos. Alguns exemplos de uso consistem em algoritmos para encontrar o caminho mais curto, como o algoritmo de Dijkstra, problemas de otimização em cadeias de produção e em biologia computacional para alinhar sequências de DNA. A maior vantagem da programação dinâmica é a capacidade de transformar problemas intratáveis devido à quantidade massiva de subprocessos em problemas que podem ser resolvidos em tempos plausíveis. Ao armazenar e reutilizar os resultados dos subproblemas, a programação dinâmica reduz de maneira significativa a redundância, melhorando assim o tempo e a eficiência de um programa.

2.2 Força Bruta

O método de força bruta é uma abordagem simples e direta para a resolução de problemas, agindo explorando todas as possíveis maneiras de encontrar a solução ótima. Sua principal vantagem está na simplicidade e garantia de completude, pois são consideradas todas as combinações de resultados possíveis, assegurando que a solução ótima será encontrada, desde que o problema seja finito e limitado. Este método é intuitivo e fácil de implementar, tornando-se uma escolha popular em situações de aprendizado. Entretanto, a força bruta apresenta uma série de desvantagens quando levada em consideração a eficiência. Para problemas de grande dimensão, o número de possibilidades a ser explorado cresce de maneira exponencial, resultando em um tempo de execução inviável. Além disso, o consumo de recursos de memória é significativamente mais alto, especialmente à medida que o problema aumenta. Essas limitações tornam o método de força bruta não escalável e impraticável para aplicações em tempo real ou com um grande volume de dados. A aplicabilidade da força bruta é restrita apenas a problemas de pequena escala, isso quando todas as soluções podem ser exploradas em um tempo razoável. Em contrapartida, para problemas maiores ou mais complexos, técnicas mais eficientes são preferidas.

3 Implementação

Neste estudo, criamos duas versões de um mesmo programa, um utilizando força bruta e o outro utilizando programação dinâmica. Para avaliar a eficácia da implementação, iremos dedicar parte desta seção à avaliação teórica do desempenho do algoritmo.

3.1 Progamação Dinâmica

A estratégia de programação dinâmica implementada gera uma solução para o problema da "máxima soma de subsequência não adjacente". A função *programação_dinâmica* requer dois argumentos: o comprimento do array n e um ponteiro para o array de números inteiros. Primeiramente, o algoritmo verifica se o comprimento do array é igual a 0 ou 1, resultando em 0 ou no único elemento do array, devido a serem situações simples. Depois, um array *pd* de tamanho *n+1* é criado dinamicamente para guardar os resultados parciais da maior soma possível em cada fase. A verificação é feita para assegurar o sucesso da alocação; se não for bem sucedida, o programa é encerrado após a exibição de uma mensagem de erro.

A seguir, o array *pd* é criado com valores iniciais: *pd[0]* é estabelecido como 0, indicando a situação inicial sem elementos, e *pd[1]* é atribuído com o valor do primeiro elemento do array de entrada, já que com apenas um elemento, essa é a soma máxima possível. A etapa seguinte consiste em percorrer do índice 2 até n, atribuindo os valores ideais ao array *pd*. A cada i, são analisadas duas escolhas: excluir o valor atual *(pd[i-1]* ou adicionar o valor atual *(pd[i-2] + array[i-1])*. A função seleciona a alternativa com o maior total e salva esse valor em *pd[i]*. Esta seleção de escolhas assegura que *pd* mantenha os melhores resultados parciais em cada fase.

Por fim, o valor máximo calculado é salvo em *pd[n]*, que indica a maior soma de uma subsequência não consecutiva do array inicial. A função libera a memória alocada para o array *pd* e retorna o resultado ideal para evitar vazamentos de memória. Dessa forma, a função utiliza a técnica de programação dinâmica para quebrar o problema em subproblemas menores, resolvendo cada um de forma eficiente e acumulando os resultados para chegar à solução ótima final.

3.1.1 Pseudocódigo

```
INICIO:
1. se n == 0 então
2. retorna 0;

3. se n == 1 então
4. retorna array[0];



5. pd = alocar array de tamanho n+1;
6. se pd == NULL então
7. imprimir "Nao foi possivel alocar memoria";
8. terminar programa;

9. pd[0] = 0;
10. pd[1] = array[0];

11. para i de 2 até n faça
12. pd[i] = max(pd[i - 1], pd[i - 2] + array[i - 1]);



13. resultado = pd[n];
14. liberar pd;
15. retorna resultado;

FIM
```

3.2 Força Bruta

A solução implementada no código utiliza força bruta para resolver o problema da "máxima soma de subsequência não adjacente". A função *força_bruta* aceita como argumentos o número de elementos do array *n*, e um ponteiro para o array de inteiros. No começo, a função checa se o tamanho do conjunto é menor ou igual a 0, retornando 0, ou se é igual a 1, retornando o único elemento do conjunto, porque são situações simples. A variável *max_pontos* é definida para guardar a pontuação mais alta identificada.

A função começa um laço que passa por cada item da matriz. Para cada item, a função guarda sua pontuação na variável *pontos* e gera um novo array *novo_array* que remove o item atual e seus vizinhos. Verifica-se se a alocação de memória para *novo_array* foi bem-sucedida. Outro loop é empregado para preencher um novo array com os elementos adequados. Em seguida, a função se chama novamente com o *novo_array* e o novo tamanho *novo_n*, adicionando a pontuação retornada aos pontos existentes.

Após a chamada recursiva, a função verifica se a pontuação atual *pontos* com *max_pontos* atualiza *max_pontos* se a pontuação atual for maior. Isso assegura que a quantidade máxima possível seja guardada. Por fim, a memória reservada para o *novo_array* é liberada para prevenir possíveis vazamentos de memória. O ciclo continua até que todos os elementos sejam processados, e a função devolve o valor de *max_pontos*, que indica a maior soma de uma subsequência não adjacente obtida através da abordagem de força bruta.

3.2.1 Pseudocódigo

```
INICIO:
1. se n <= 0 então
2. retorna 0;

3. se n == 1 então
4. retorna array[0];

5. max_pontos = 0;

6. para i de 0 até n-1 faça
7. pontos = array[i];
8. novo_n = 0;
9. novo_array = alocar array de tamanho n;
10. se novo_array == NULL então
11. imprimir "Nao foi possivel alocar memoria";
12. terminar programa;


13. para j de 0 até n-1 faça
14. se j != i e j != i - 1 e j != i + 1 então
15. novo_array[novo_n] = array[j];
16. novo_n = novo_n + 1;


17. pontos = pontos + forca_bruta(novo_n, novo_array);
18. max_pontos = max(max_pontos, pontos);
19. liberar novo_array;


20. retorna max_pontos;

FIM
```

4 Resultados e Discussões

Nesta seção, mostramos os resultados alcançados ao realizar testes em ambos os algortimos. Estes resultados são essenciais para a avaliação do desempenho do algoritmo em várias situações de uso. A avaliação é feita com base em vários testes de performance, com ênfase no tempo de execução do usuario e do sistema.

4.1 Avaliação de Desempenho

Como podemos ver na comparação entre os dois algoritmos, existem diferenças significativas em termos de eficiência, escalabilidade e utilização de recursos do sistema. Ambos os algoritmos foram criados para resolver problemas de otimização; entretanto, suas abordagens e desempenhos são diferentes.

4.1.1 Exposição Gráfica

4.1.1.1 Programação Dinâmica

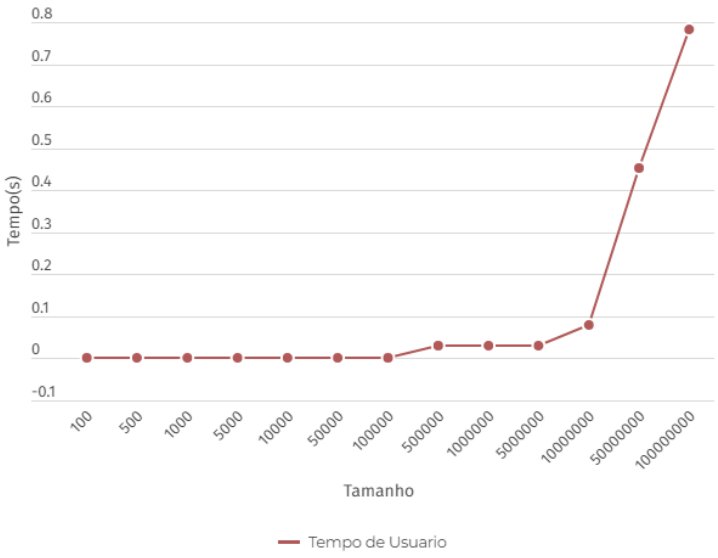

Figura 1 – Tempo de usuário em função do tamanho

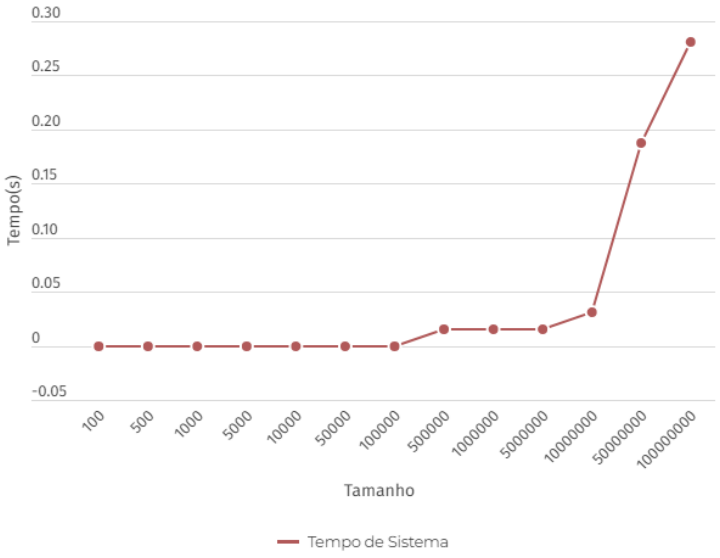

Figura 2 – Tempo de sistema em função do tamanho

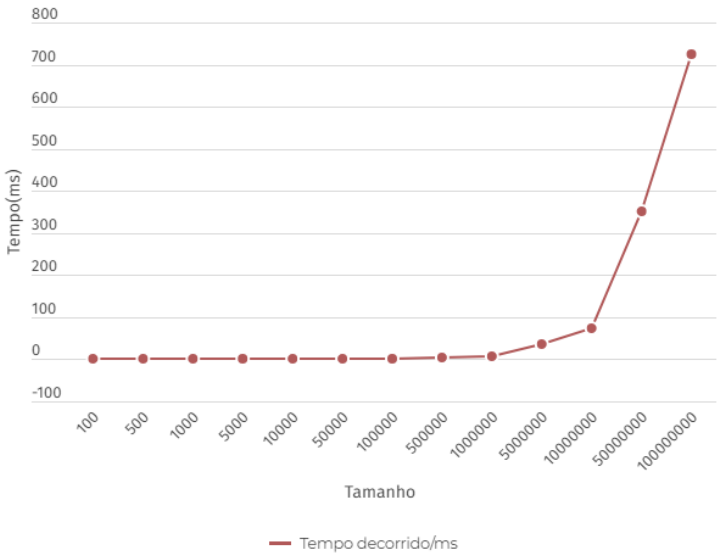

Figura 3 – Tempo total decorrido em função do tamanho

4.1.1.2 Força Bruta

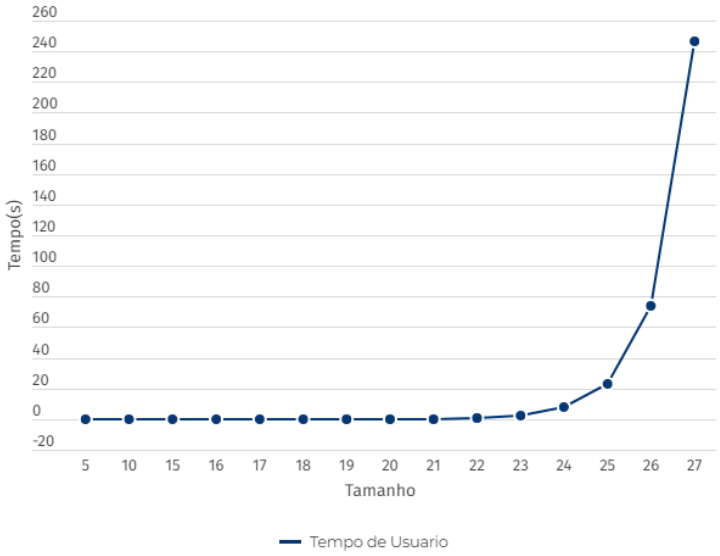

Figura 4 – Tempo de usuário em função do tamanho

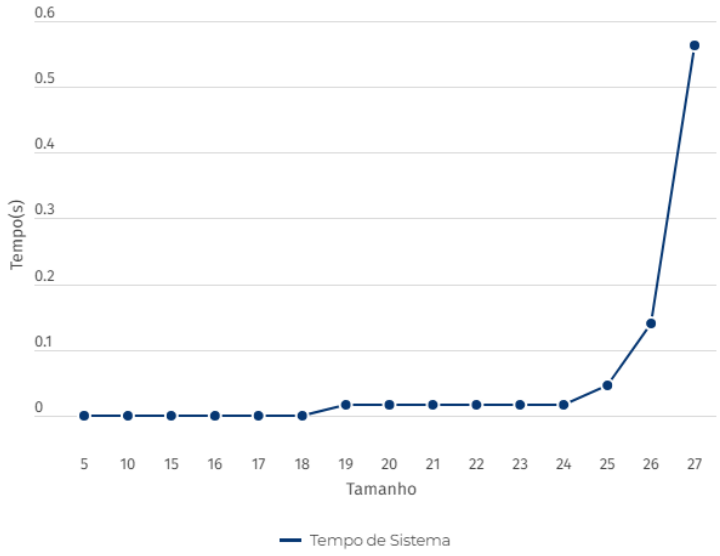

Figura 5 – Tempo de sistema em função do tamanho

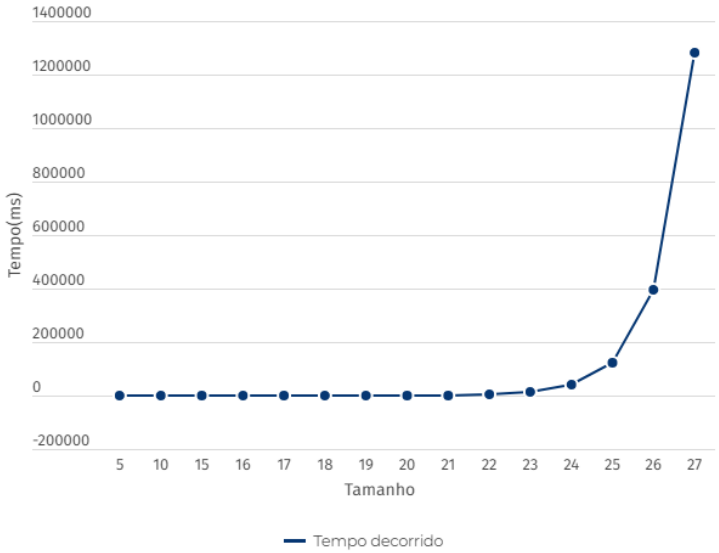

Figura 6 – Tempo total decorrido em função do tamanho

4.1.2 Eficiência e Escalabilidade

Na programação dinâmica e perceptível uma eficiência notável, especialmente quando maiores tamanhos de entrada são colocados à teste. O tempo de execução cresce de forma quase linear a quadrática com o aumento do vetor de dados. Para uma entrada de 100 milhões de elementos, o tempo decorrido é aproximadamente 723.34ms. Isso mostra a capacidade da programação dinâmica de dividir o problema em subproblemas menores, resolvendo-os uma vez e armazenando os resultados para a reutilização. Esse método é especialmente útil para problemas que têm inúmeros subproblemas e estrutura de solução ótima, tornando-o altamente eficiente e escalável.

Em contrapartida, o algoritmo de Força Bruta, embora eficiente para entradas pequenas, exibe uma escalabilidade extremamente limitada. O tempo de execução aumenta de maneira exponencial com o tamanho da entrada. Como testado, uma entrada de apenas 27 elementos teve o tempo decorrido de aproximadamente 1283 segundos, um alto custo computacional, que não só é lento como também consome muita memória e tempo de processamento. Isso ocorre porque o método testa todas as possibilidades para encontrar a solução ótima, o que gera um crescimento exponencial do tempo de execução à medida que o tamanho da entrada aumenta, sendo assim, não viável para vetores grandes. A abordagem, apesar de garantir a solução ótima, torna-se impraticável.

4.1.3 Análise de Complexidade

4.1.3.1 Progamação Dinâmica

O algoritmo de programação dinâmica é eficiente e escalável, demonstrando uma complexidade de tempo e espaço linear. Essa eficiência é alcançada através da abordagem de resolver os subproblemas e armazená-los para reutilização, evitando assim a redundância. Após as verificações iniciais e a alocação de memória, o loop principal percorre o array uma vez, resultando em uma operação constante $O(n)$ em cada iteração, indicando que o tempo de execução cresce de maneira linear com o aumento da entrada. Isso é particularmente vantajoso para problemas de grande escala, onde a eficiência temporal é importante.

4.1.3.2 Força Bruta

O algoritmo de força bruta, embora seja direto em sua implementação, demonstra uma complexidade de tempo exponencial $O(2^n)$, tornando-o impraticável para grandes entradas. Ao tentar todas as combinações possíveis para encontrar a solução ótima, um crescimento exponencial do tempo de execução é obtido como resultado.

A análise de complexidade de tempo do algoritmo mostra que, para cada elemento presente no array, é chamada uma função recursiva, excluindo o elemento atual e seus adjacentes. Isso resulta em aproximadamente n chamadas, com cada uma reduzindo o problema em aproximadamente 3 elementos. A relação de recorrência assim é equivalente a $T(n) = n * T(n-3)$, o que leva a uma solução exponencial, $T(n) = O(2^n)$.

4.2 Conclusão

A comparação entre os algoritmos de Programação Dinâmica e Força Bruta, considerando tanto a análise teórica quanto os dados exibidos nos gráficos, destaca a superioridade da Programação Dinâmica como técnica mais eficiente e que possui uma capacidade maior para vetores grandes. O algoritmo de programação dinâmica, com complexidade $O(n)$, oferece uma solução balanceada e eficiente para grandes problemas. Em contraste, o algoritmo de Força Bruta, com complexidade de $O(2^n)$, é altamente ineficiente para grandes entradas.

Esta análise aponta a escolha da programação dinâmica como a melhor abordagem entre as duas analisadas para resolver problemas de otimização, especialmente em cenários onde os vetores são escaláveis e a gestão de recursos é essencial. A abordagem da Força Bruta, embora garanta a solução ótima, é melhor utilizada apenas em contextos em que o tamanho da entrada é pequeno para permitir uma execução viável.

5 Referências

Slide dado em sala de aula (Paradigmas de Programação)

stackoverflow.blog.

ime.usp.

geeksforgeeks.com.