Relatório sobre a Atividade Pontuada

Aluna: Aline Argolo de Athaydes . Data: 02/11/2025

O objetivo deste projeto é resolver os problemas propostos, utilizando as técnicas de Divisão e Conquista (DC) puros (recursivos *top-down*), contra suas implementações com as técnicas de Programação Dinâmica (PD) *(bottom-up)*.

A análise foca em como a abordagem de PD resolve eficientemente o problema de subproblemas sobrepostos (overlapping subproblems), evitando o recálculo que leva as implementações de DC a uma complexidade exponencial.

Foi criado no projeto um arquivo com o código usando DC e um segundo usando PD e um terceiro código para gerar os gráficos solicitados pela atividade, para que possamos visualizar as diferenças de complexidades de cada técnica usada com as mesmas entradas.

Foi gerado também um gráfico de tempo de execução por cada algoritmo executado. Mas como não foi solicitado e percebi que o tempo de execução tem influência do hardware usado, e outros fatores, não vou fazer as análises deles.

Mas todo o código e imagens geradas poderão ser acessadas através do link do GitHub do qual subi a atividade para análises caso achar necessário, segue abaixo o link do repositório que está como público:

Link - GitHub - Atividade

A fim de comparar a complexidade de cada um, dado um mesmo valor e quantidades. Fiz o trabalho tentando sempre estressar o código com diversos tipos diferentes de entradas, simulando como seria no melhor caso, pior caso e caso médio. Em todos os problemas proposto pela a atividade que foram esses:

1. Subsequência crescente mais longa. Recebe como entrada uma sequência de tamanho n.

2. Distância de edição. Tem como entrada duas sequências, de tamanho n e m.

3. Subconjunto soma. Sua entrada é um conjunto de n valores numéricos.

No primeiro problema foi solicitado a Subsequência crescente mais longa, que recebe como entrada uma sequência de tamanho ‘n’. Para esse problema foi usado a fórmula recursiva vista em aula, presente no slide 12 de Programação Dinâmica, a seguir:

$$T_i = \begin{cases} 1 + \max_{j<i<} T_j & \text{se } x_i > x_j \text{ e } i >= 1 \\ 0 & \text{se } i = 0 \text{ <= caso base} \end{cases}$$

Nas primeiras imagens abaixo os 3 gráficos que foram gerados no meu primeiro ‘Experimento Aleatório’, no qual comparam as ‘operações’ feitas pelo PD (linha vermelha) com as ‘chamadas’ do DC (linha azul), à medida que o tamanho da entrada ‘n’ (eixo ‘X’) aumenta de tamanho. Nesse experimento foi realizado um código usando random.randint que gerou listas aleatórias de n = 1 até n = 14 repetindo em 3 *runs.* O que foi possível observar nesses gráficos foi que:

Linha Vermelha (PD): Em todos os 3 testes, a linha vermelha cresce de forma muito lenta e previsível. Ela mal sai do eixo zero, indicando um baixo número de operações (na casa das centenas) mesmo para n=14. Sendo sua Complexidade Polinomial de $\Theta(n^2)$. Linha Azul (DC): Em todos os 3 testes, a linha azul permanece baixa para n pequeno (aprox. 1 a 8) e então "explode" verticalmente, mostrando um crescimento exponencial claro. O número de chamadas atinge a casa dos milhares (8.000 a 13.000) já em n=14. Sendo uma Complexidade Exponencial de $\Theta(2^n)$.

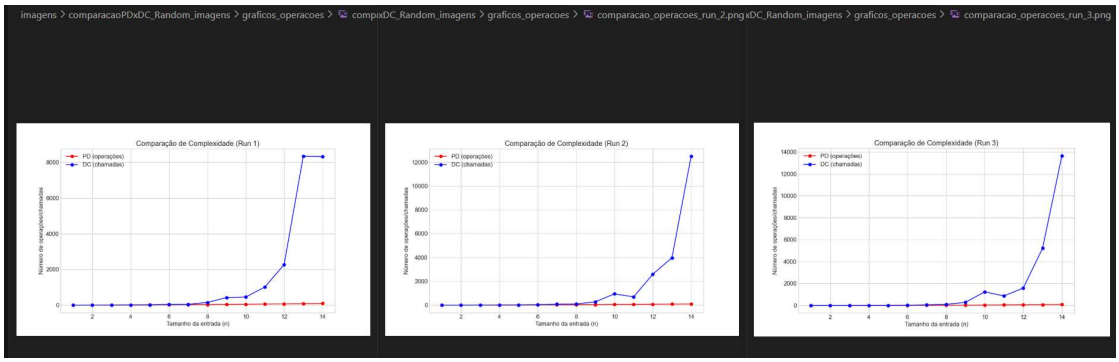

Figura 1: Comparativo de complexidade (PD vs. DC) para a Subsequência Crescente Mais Longa, executado 3 vezes (Runs) com entradas aleatórias de tamanho n (1 a 14).

Para estressar o código e verificar cenários mais controlados (sem uso do random), foi realizado o Experimento Prefixo. Em que coloquei entradas controladas a fim de testar o código para pior caso, melhor caso e caso médio. Foram gerados 4 gráficos, um com entradas semelhantes do meu início de codificação, e um para cada dos 3 casos mencionados acima. O que pode se observar que em todos os 4 gráficos linha vermelha (PD) e visualmente quase idêntica, com sua complexidade Polinomial $\Theta(n^2)$, se mantendo sempre próximo ao eixo 0 tendo muito pouco ou nada de alteração. Provando a eficiência do PD depende do tamanho de entrada ‘n’ mas não do conteúdo da lista (se está ordenada ou não). Já a linha azul (DC) explode exponencialmente em todos os casos, mas o mais importante é que o nível de explosão do DC muda dependendo do valor de entrada. Na Figura 4 do gráfico caso 4 , esta é a Lista-Base: [1, 2, 3, ...] (crescente), é o pior caso para esta formulação recursiva. O DC explode mais rápido e mais alto, ultrapassando 300.000 chamadas em n=15, já na Figura 5 gráfico caso 4, esta é a Lista-Base: [15, 14, 13, ...] (decrescente). Este é um "caso melhor" para o DC, mas ainda é exponencial. Ele "só" atinge ~8.000 chamadas em n=15. já na Figura 3 do gráfico caso 2, Mostra um caso intermediário, mas que também explode de forma extrema (quase 80.000 chamadas). Resumo pode-se concluir que, em todos os cenários testados, a implementação de DC sofre de subproblemas sobrepostos, levando a uma complexidade exponencial ($\Theta(2^n)$), visível nas curvas de "explosão" dos gráficos. A implementação de PD, por outro lado,

resolve cada subproblema uma única vez e armazena o resultado, garantindo uma complexidade polinomial ($\Theta(n^2)$)

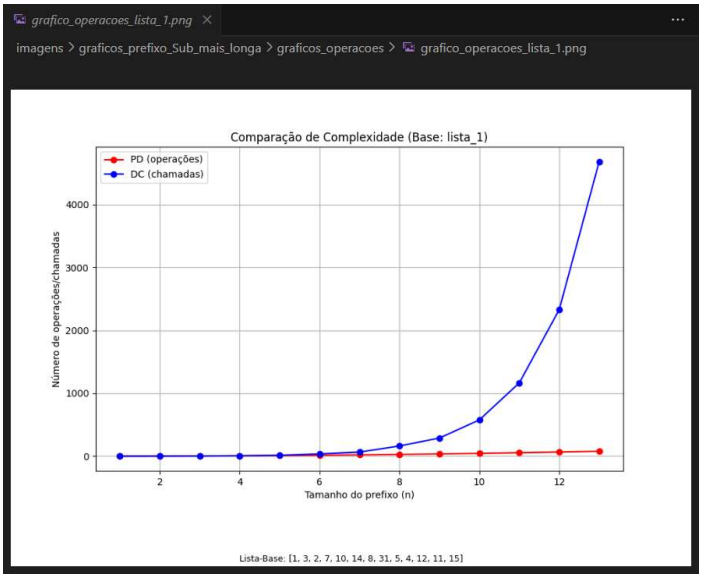

Figura 2: Caso 1 - comum DC x PD Subs. mais longa

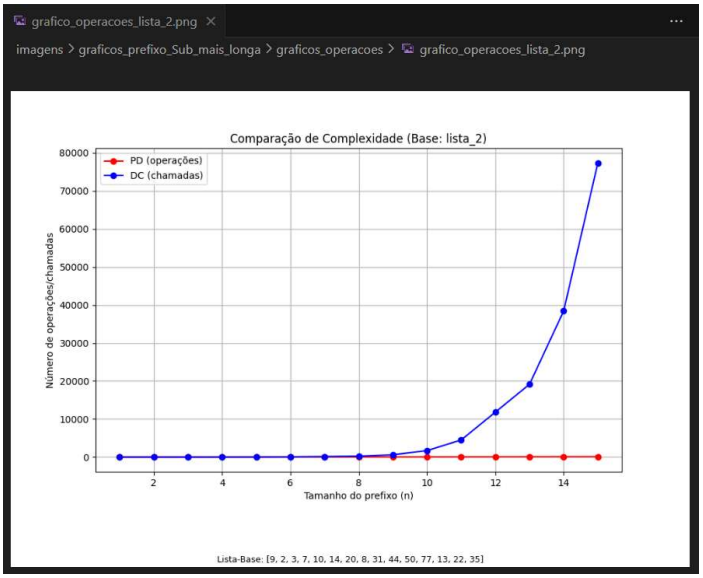

Figura 3: Caso 2 - Teste caso médio

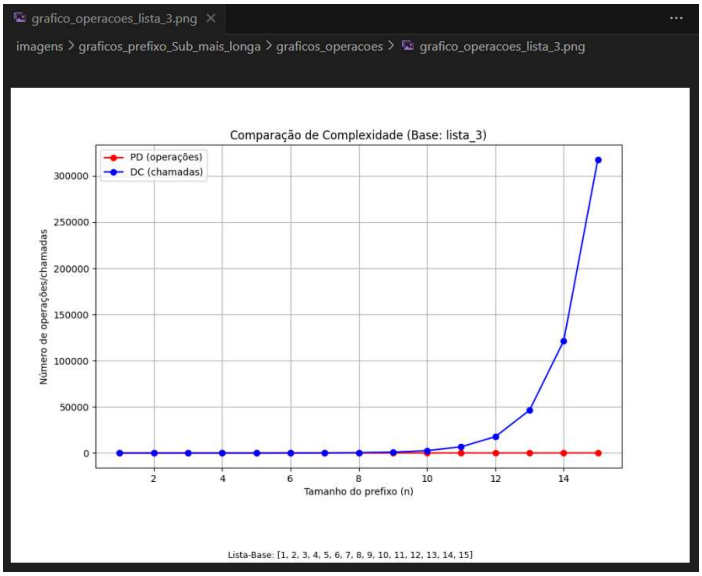

Figura 4: Caso 3 - Pior caso - crescente

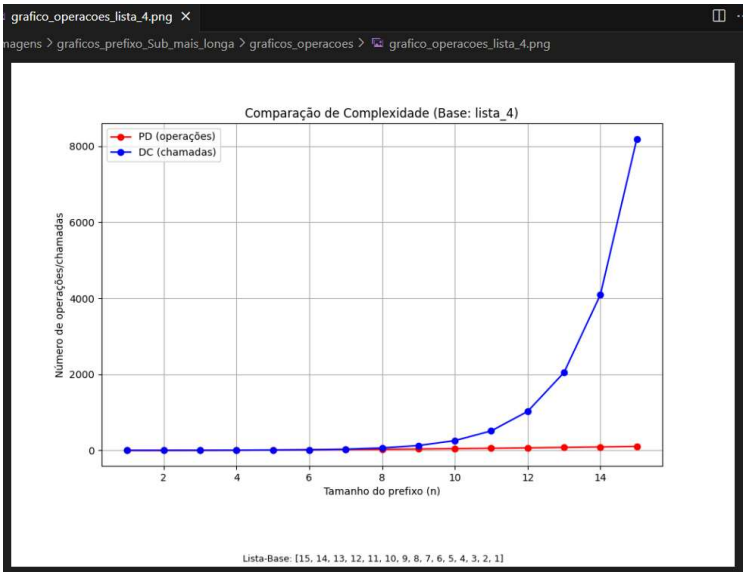

Figura 5: Caso 4 - Melhor caso - Decrescente

Para o segundo problema da atividade, Distância de edição. Tem como entrada duas sequências, de tamanho n e m. Foi implementada a solução de Programação Dinâmica (PD) para a Distância de Edição. Ele é baseado na formulação recursiva:

Linha 1 (Caso base j=0): i, se j = 0

Linha 2 (Caso base i=0): j, se i = 0

Linha 3 (Caso recursivo): $\min\{C_{i-1,j} + 1, C_{i,j-1} + 1, C_{i-1,j-1} + \text{diff}(i, j)\}$, se i, j > 0

Ele usa a abordagem bottom-up, preenchendo uma matriz (tabela) C de tamanho (n+1) x (m+1). Cada célula armazena a distância mínima para transformar os i primeiros caracteres de X nos j primeiros de Y. Como ele preenche cada uma das n * m células da tabela uma vez, a complexidade é Polinomial (Θ(nm)).

Foi usada a mesma formulação recursiva para escrever o código para PD e DC. Como: A função resolve o problema fazendo três chamadas recursivas: remoção,inserção e substituição/match. Como cada chamada gera até três novas chamadas, e os mesmos subproblemas são recalculados milhões de vezes, o algoritmo sofre de subproblemas sobrepostos e tem complexidade Exponencial de $\Theta(3^{n+m})$.

Com o propósito de estressar os código redigidos foi feito diversos cenários do qual gerou 4 gráficos, usamos como entrada: "ALGORITMOS" vs "LOGARITMOS", "ABBBAAB" vs "BBAABB", "BAHIA" e "VITORIA" e "COMPUTACAO" vs "COMPUTADOR". Em todos os casos a linha vermelha (PD) quase não sai do eixo 0, mais uma vez provando sua eficiência em diversos cenários de testes. Tendo sua complexidade Polinomial de Θ(nm). Já a linha azul (DC) em todos os 4 casos demonstra uma explosão exponencial de $\Theta(3^{n+m})$. Sendo que nos casos de entradas "ALGORITMOS" vs "LOGARITMOS", "COMPUTACAO" vs "COMPUTADOR" são os piores casos, como as strings são muito parecidas isso força a recursão do DC a explorar muitas ramificações “parecidas”, resultando em uma explosão de 12 milhões de chamadas com uma entrada de tamanho m+n=20. O exemplo retirado do slide ("ABBBAAB" vs "BBAABB") também mostra a explosão, chegando a ~14.000 chamadas para m+n=12. E para testar um cenário de "melhor" caso temos a entrada "BAHIA" e "VITORIA", como eles são totalmente diferentes, o algoritmo DC "percebe" mais rápido que os caminhos não batem. Mesmo assim, ele ainda explode exponencialmente, chegando a ~2.500 chamadas para m+n=10.

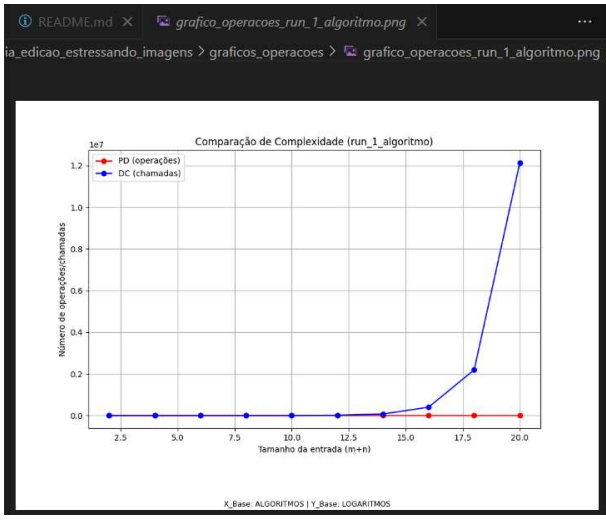

Figura 6: Distância de edição - "ALGORITMOS" vs "LOGARITMOS"

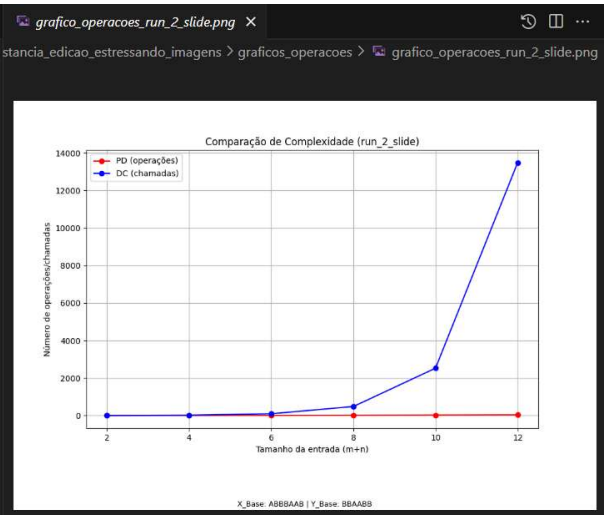

Figura 7: Distância de edição - "ABBBAAB" vs "BBAABB"

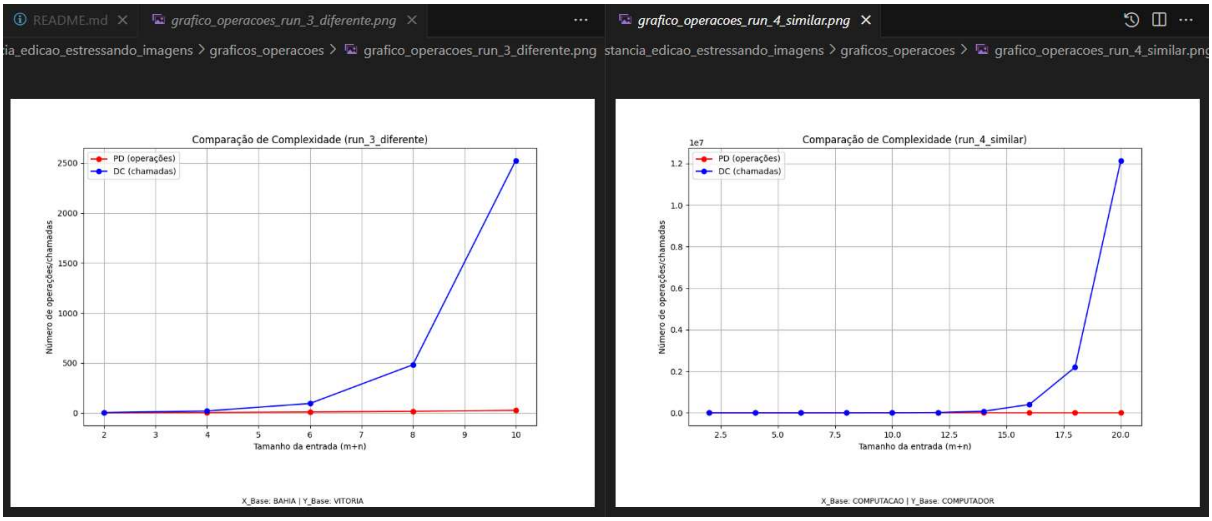

Figura 8: Distância de edição - "BAHIA" e "VITORIA"

Figura 7: Distância de edição-“COMPUTACAO"vs"COMPUTADOR"

Concluem-se que os experimentos provam que a abordagem DC é exponencialmente inviável, independentemente da similaridade das strings. A abordagem PD, por outro lado, permanece eficiente e polinomial (Θ(nm)) em todos os cenários.

E vamos para o 3 problema Subconjunto soma. Sua entrada é um conjunto de ‘n’ valores numéricos. Esse problema se mostrou o mais interessante, o mais rico para análises, pois neles podemos identificar não somente que o “DC explode, e o PD não”, conseguimos visualizar nos gráficos quando o DC explode e porque isso acontece. Para o Subconjunto de soma foi usado uma análoga da mochila em que foi usada a formulação recursiva implementada (e testada nas duas abordagens) é S(i,j), que verifica se é possível atingir a soma ‘j’ com os ‘i’ primeiros itens. O caso recursivo principal é: $S(i,j)=S(i-1,j) \vee S(i-1,j-a\ i\ )$. Esta formulação leva diretamente às complexidades observadas nos gráficos. Foi implementada a solução bottom-up (PD) onde tem uma tabela booleana F de (n+1) x (k+1). Onde cada célula F[i][j] armazena a resposta True/False. Para o código em PD apresenta uma complexidade Polinomial de Θ(nk). Já na recursão em DC foi implementada a mesma formulação recursiva F(i, j) como uma função de DC pura. Como cada chamada pode se ramificar em duas, e os mesmos subproblemas são recalculados, ele sofre de subproblemas sobrepostos assim resultando que o DC apresenta uma complexidade Exponencial de $\Theta(2^n)$.

Essa atividade gerou 3 gráficos de operações que contam 3 histórias bem diferentes e vou detalhar um pouco sobre cada uma por ter achado essas diferenças interessantes.

No primeiro gráfico foi dado como entrada: A_Base: [3, 34, 4, ...], Soma Alvo (k): 100, observamos neste gráfico que a linha vermelha (PD) se encontra, perfeitamente linear (Θ(n)), pois k=100 é fixo. Já a linha azul (DC) explode exponencialmente até o ponto n=11 depois para de explodir ficando quase reta. Porque? Na saída do console consegui ver o motivo em n=11 o resultado teve a saída Res: True. O código do DC usa ‘or’. Graças à avaliação de "curto-circuito" (short-circuit), no momento em que a primeira ramificação recursiva encontrou True, o Python parou de executar as outras ramificações, "podando" a árvore exponencial.

No segundo gráfico foi dado como entrada: A_Base: [1, 5, 2, ...], Soma Alvo (k): 150, e nela podemos observar que a Linha vermelha (PD) cresce de forma perfeitamente Linear. Isso é a prova visual do Θ(nk). Como k (150) é fixo nesse experimento, o custo Θ(n×150) se comporta como Θ(n) (Linear). Mas observando a linha azul (DC) vemos um crescimento inicial mais devagar comparado com o PD, mas depois ele explode mostrando uma explosão exponencial (Θ(2^n)) pura. Porque? Na saída do console mostrou um resultado Res: False em todas as etapas isso significa que o DC foi forçado a explorar toda a árvore de recursão (2^n caminhos) e nunca encontrou uma solução, sendo assim o “pior caso” do DC.

E por último temos o terceiro gráfico que foi dado como entrada: A_Base: [20, 31, 12, ...], Soma Alvo (k): 200. Observando as imagens dos gráficos gerados por essa recursão vemos que a linha vermelha (PD) começa bem mais elevada a linha azul (DC) ilustrando o "crossover" de complexidade Θ(nk) vs. Θ(2^n), onde a linha azul está bem mais baixa que a vermelha, mas no ponto n=10 elas se cruzam. Porque? Este foi o gráfico que ficou mais claro a diferença de complexidade das incursões de Θ(nk) do PD vs Θ(2^n) de DC. Como o alvo ‘k’ (200) era muito grande, o custo n * k do PD foi maior que o custo de 2^n do DC para valores pequenos de n. Onde vimos no console:
Em n=8, o PD fez 8 * 200 = 1600 ops. O DC fez só 511 ops. O DC foi mais rápido.
Em n=10, o PD fez 10 * 200 = 2000 ops. O DC fez 2040 ops. O PD se tornou mais lento.
Isso prova que, se k for muito maior que 2^n, a recursão pura (DC) pode ser temporariamente mais rápida, mas a complexidade exponencial do DC sempre vai perder a longo prazo.

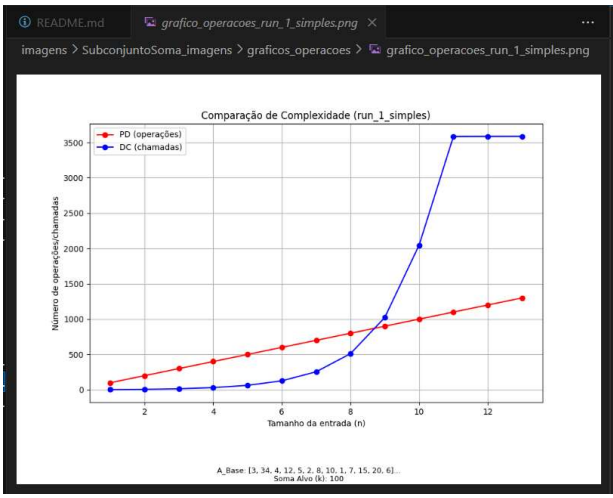

Figura 8: A_Base: [3, 34, 4, ...], Soma Alvo (k): 100

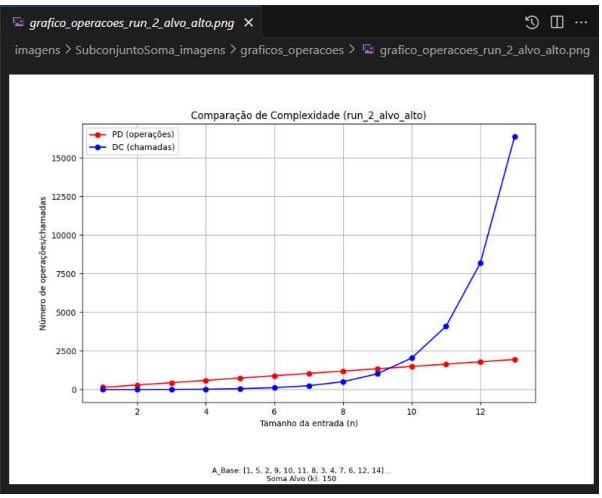

Figura 9: A_Base: [1, 5, 2, ...], Soma Alvo (k): 150

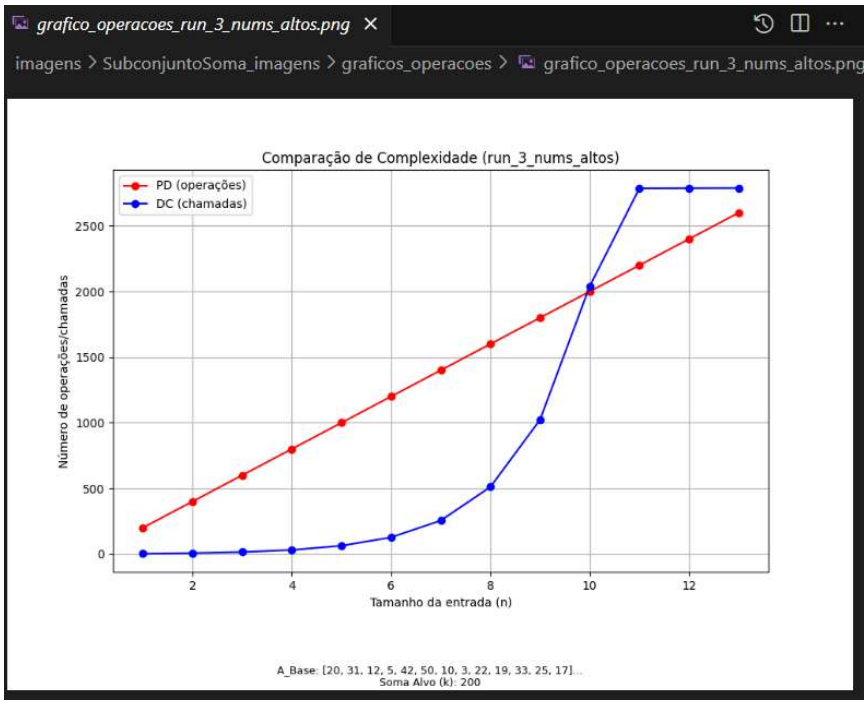

Figura 10: A_Base: [20, 31, 12, ...], Soma Alvo (k): 200

Com essas análises podemos concluir que o PD tem complexidade $\Theta(nk)$, que é linear em n se k for fixo. E foi provado que o DC é $\Theta(2^n)$, e seu desempenho pode ser otimizado por "short-circuit" se uma solução for encontrada cedo, mas que ele explode no pior caso.

Conclusão Geral:

A análise experimental realizada nos três problemas (Subsequência Crescente Mais Longa, Distância de Edição e Subconjunto Soma) demonstrou de forma conclusiva o objetivo da atividade.

Em todos os cenários, a implementação de Divisão e Conquista (DC), baseada puramente na formulação recursiva, sofreu com o recálculo de subproblemas sobrepostos, resultando em uma complexidade de tempo exponencial ($\Theta(2^n)$ ou ($\Theta(3^{n+m})$), como provado visualmente pelas curvas de "explosão" nos gráficos de operações.

A implementação de Programação Dinâmica (PD), por outro lado, ao utilizar uma abordagem bottom-up com tabelas para armazenar e reutilizar soluções parciais, manteve uma complexidade polinomial ($\Theta(n^2)$, $\Theta(nm)$) ou pseudo-polinomial ($\Theta(nk)$).

Conclui-se que, embora a formulação recursiva seja fundamental para definir a subestrutura ótima do problema, a Programação Dinâmica é a técnica de implementação eficiente para resolvê-los, evitando a ineficiência exponencial da recursão pura.