Algoritmos Gulosos vs Programação Dinâmica

Análise Detalhada de 10 Problemas Clássicos

Análise Comparativa

10 de novembro de 2025

Problema 1: Problema do Troco

Descrição do Problema:
Dado um valor V e um conjunto de moedas com denominações $\{c_1, c_2, ..., c_n\}$, encontrar o número mínimo de moedas necessárias para formar exatamente o valor V.

Abordagem Gulosa:

- **Estratégia:** sempre escolher a maior moeda disponível que não exceda o valor restante
- **Algoritmo:** ordenar moedas em ordem decrescente e repetidamente pegar a maior possível
- **Complexidade:** $O(n \log n)$ para ordenar + $O(V/c_{min})$ para construir solução
- **Resultado:** Não garante solução ótima para moedas não-canônicas

Problema 1: Problema do Troco

Abordagem de Programação Dinâmica:

- **Recorrência:** $dp[i] = \min_{c_j \leq i}(dp[i - c_j] + 1)$
- **Subestrutura:** solução ótima para i usa solução ótima para $i - c_j$
- **Complexidade:** $O(V \times n)$
- **Resultado:** Sempre encontra solução ótima

Exemplo Comparativo:

Caso 1 - Moedas Canônicas (guloso funciona):

- Moedas: $\{1, 5, 10, 25\}$, Valor: 30
- Guloso: $25 + 5 = 2$ moedas (ótimo)
- PD: $25 + 5 = 2$ moedas (ótimo)

Caso 2 - Moedas Não-Canônicas (guloso falha):

- Moedas: $\{1, 3, 4\}$, Valor: 6
- Guloso: $4 + 1 + 1 = 3$ moedas (subótimo)
- PD: $3 + 3 = 2$ moedas (ótimo)

Problema 1: Problema do Troco

Aplicações Práticas:

- Sistemas de caixas automáticos
- Máquinas de venda
- Otimização de pagamentos

Conclusão: Use guloso apenas para sistemas monetários conhecidos (canônicos). Para casos gerais, use PD.

Problema 2: Mochila 0/1 vs Fracionária

Descrição do Problema:
Dados n itens, cada um com valor v_i e peso w_i, e uma mochila com capacidade máxima W, maximizar o valor total dos itens na mochila.

Mochila Fracionária - Abordagem Gulosa:

- **Variação:** permite pegar frações de itens
- **Estratégia:** ordenar itens por razão v_i/w_i (valor por unidade de peso) em ordem decrescente
- **Algoritmo:** pegar itens completos até não caber mais, então pegar fração do próximo
- **Complexidade:** $O(n \log n)$
- **Resultado:** Solução ótima garantida!

Problema 2: Mochila 0/1 vs Fracionária

Mochila 0/1 - Abordagem PD:

- **Variação:** deve pegar item completo ou não pegar
- **Recorrência:** $dp[i][w] = \max(dp[i-1][w], v_i + dp[i-1][w-w_i])$
- **Complexidade:** $O(n \times W)$
- **Resultado:** Solução ótima para versão discreta

Problema 2: Mochila 0/1 vs Fracionária

Exemplo Comparativo:

Considere os seguintes itens e capacidade $W = 50$:

Item	**Valor (v_i)**	**Peso (w_i)**	v_i/w_i
A	60	10	6.0
B	100	20	5.0
C	120	30	4.0

Mochila Fracionária (Guloso):

- Pega A completo (10kg, valor=60)
- Pega B completo (20kg, valor=100)
- Pega 20kg de C (valor=$120 \times \frac{20}{30} = 80$)
- **Total:** $60 + 100 + 80 = 240$ (ótimo fracionário)

Mochila 0/1 (PD):

- Pega B completo (20kg, valor=100)
- Pega C completo (30kg, valor=120)
- **Total:** $100 + 120 = 220$ (ótimo inteiro)

Problema 3: Escalonamento de Tarefas com Pesos

Descrição do Problema:
Dadas n tarefas, cada uma com tempo de início s_i, tempo de fim f_i e peso (valor) w_i, selecionar um subconjunto de tarefas não sobrepostas que maximize o peso total.

Abordagem Gulosa:

- **Estratégia:** ordenar tarefas por peso decrescente, escolher gananciosamente se compatível
- **Algoritmo:** para cada tarefa em ordem de peso, adicionar se não sobrepor com já escolhidas
- **Complexidade:** $O(n \log n)$ para ordenar + $O(n^2)$ para verificar compatibilidade
- **Resultado:** Não garante solução ótima

Problema 3: Escalonamento de Tarefas com Pesos

Abordagem de Programação Dinâmica:

- **Pré-processamento:** ordenar por tempo de término
- **Recorrência:** $dp[i] = \max(dp[i-1], w_i + dp[p(i)])$
- $p(i)$ = índice da última tarefa compatível com tarefa i
- **Complexidade:** $O(n \log n)$ (ou $O(n^2)$ dependendo de busca)
- **Resultado:** Solução ótima garantida

Problema 3: Escalonamento de Tarefas com Pesos

Exemplo onde Guloso Falha:

Considere as seguintes tarefas:

Tarefa	Início	Fim	Peso
T1	0	10	100
T2	1	3	60
T3	4	6	60

Solução Gulosa (por peso decrescente):

- Escolhe T1 (maior peso = 100)
- T2 e T3 sobrepõem com T1, não podem ser escolhidas
- **Peso total:** 100 (subótimo)

Solução PD:

- Escolhe T2 (intervalo [1,3])
- Escolhe T3 (intervalo [4,6], compatível com T2)
- **Peso total:** $60 + 60 = 120$ (ótimo)

Aplicação: Alocação de recursos em servidores, agendamento de reuniões

Problema 4: Corte de Hastes

Descrição do Problema:
Dada uma haste de comprimento n e uma tabela de preços p_i para hastes de comprimento i $(1 \leq i \leq n)$, determinar a maneira de cortar a haste para maximizar o lucro total.

Abordagem Gulosa:

- **Estratégia:** sempre cortar no tamanho que tem maior valor por unidade de comprimento (p_i/i)
- **Algoritmo:** encontrar $\max(p_i/i)$ e cortar repetidamente nesse tamanho
- **Complexidade:** $O(n)$
- **Resultado:** Geralmente não produz solução ótima

Problema 4: Corte de Hastes

Abordagem de Programação Dinâmica:

- **Recorrência:** $dp[n] = \max_{1 \leq i \leq n}(p_i + dp[n-i])$
- **Ideia:** considerar todos os possíveis primeiros cortes
- **Complexidade:** $O(n^2)$
- **Resultado:** Solução ótima garantida

Problema 4: Corte de Hastes

Exemplo Comparativo:

Comprimento da haste: $n = 8$

Comp. i	1	2	3	4	5	6	7	8
Preço p_i	1	5	8	9	10	17	17	20
p_i/i	1.0	2.5	2.67	2.25	2.0	2.83	2.43	2.5

Solução Gulosa:

- Maior razão: $p_6/6 = 2.83$
- Corta: $6 + 2 \rightarrow$ preço $= 17 + 5 = 22$ (subótimo)

Problema 4: Corte de Hastes

Solução PD:

- Testa todas combinações
- Melhor: $2 + 6$ ou $2 + 2 + 2 + 2 \rightarrow$ preço $= 22$
- Ou ainda melhor: $2 + 2 + 2 + 2 = 5 + 5 + 5 + 5 = 20$, mas $p_6 + p_2 = 22$
- Solução ótima: dois pedaços de 2 e um de $4 = 5 + 5 + 9 = 19$ ou $2 + 6 = 22$
- **Máximo:** 22 (mas outras formas dão $10 + 10 + 1 + 1 = 22$ também)

Aplicação: Indústria madeireira, corte de tecidos, otimização de estoque.

Problema 5: Partição de Conjunto

Descrição do Problema:
Dado um conjunto de números inteiros positivos $S = \{a_1, a_2, ..., a_n\}$, determinar se é possível particioná-lo em dois subconjuntos S_1 e S_2 tais que $\sum S_1 = \sum S_2$.

Abordagem Gulosa (Heurística de Balanceamento):

- **Estratégia:** ordenar elementos em ordem decrescente, sempre adicionar ao grupo com menor soma atual
- **Algoritmo:** manter dois contadores, inserir cada elemento no de menor valor
- **Complexidade:** $O(n \log n)$
- **Resultado:** Não garante partição perfeita - apenas aproximação

Problema 5: Partição de Conjunto

Abordagem de Programação Dinâmica:

- **Reformulação:** existe subconjunto com soma $= \frac{\text{soma\_total}}{2}$?
- **Recorrência:** $dp[i][s] =$ é possível obter soma s usando primeiros i elementos
- $dp[i][s] = dp[i-1][s] \vee dp[i-1][s - a_i]$
- **Complexidade:** $O(n \times \text{soma})$ (pseudo-polinomial)
- **Resultado:** Resposta exata: SIM ou NÃO

Problema 5: Partição de Conjunto

Exemplo Comparativo:

Conjunto: $S = \{1, 5, 11, 5\}$, Soma total $= 22$

Solução Gulosa:

- Ordenado: $[11, 5, 5, 1]$
- Passo 1: $S_1 = \{11\}$ (soma=11), $S_2 = \{\}$ (soma=0)
- Passo 2: $S_1 = \{11\}$ (soma=11), $S_2 = \{5\}$ (soma=5)
- Passo 3: $S_1 = \{11, 5\}$ (soma=16), $S_2 = \{5\}$ (soma=5)
- Passo 4: $S_1 = \{11, 5\}$ (soma=16), $S_2 = \{5, 1\}$ (soma=6)
- **Resultado:** $16 \neq 6$ (falhou em encontrar partição)

Problema 5: Partição de Conjunto

Solução PD:

- Alvo: soma $= 22/2 = 11$
- Verifica se existe subconjunto com soma 11
- Encontra: $\{11\}$ tem soma 11, logo $\{5, 5, 1\}$ também tem soma 11
- **Resultado:** $S_1 = \{11\}$, $S_2 = \{5, 5, 1\}$ (partição perfeita existe!)

Aplicação: Balanceamento de carga em processadores, divisão equitativa de recursos.

Problema 6: Subsequência Crescente Mais Longa - LIS

Descrição do Problema:
Dada uma sequência de números $A = [a_1, a_2, ..., a_n]$, encontrar o comprimento da maior subsequência estritamente crescente (não necessariamente contígua).

Abordagem Gulosa (Heurística):

- **Estratégia:** percorrer sequência linearmente, adicionar elemento se for maior que o último da subsequência atual
- **Algoritmo:** manter lista da subsequência, adicionar apenas se $a_i >$ último elemento
- **Complexidade:** $O(n)$
- **Resultado:** Falha frequentemente - encontra uma crescente, mas não necessariamente a maior

Problema 6: Subsequência Crescente Mais Longa - LIS

Abordagem de Programação Dinâmica:

- **Estado:** $dp[i]$ = comprimento da LIS terminando em posição i
- **Recorrência:** $dp[i] = \max_{j<i, a_j<a_i}(dp[j] + 1)$
- **Complexidade:** $O(n^2)$ básico, $O(n \log n)$ otimizado com busca binária
- **Resultado:** Solução ótima garantida

Problema 6: Subsequência Crescente Mais Longa - LIS (Parte 2)

Exemplo Comparativo:

Sequência: $A = [10, 9, 2, 5, 3, 7, 101, 18]$

Solução Gulosa:

- Inicia com [10]
- $9 < 10$: ignora
- $2 < 10$: ignora
- $5 < 10$: ignora
- $3 < 10$: ignora
- $7 < 10$: ignora
- $101 > 10$: adiciona $\rightarrow [10, 101]$
- $18 < 101$: ignora
- **Comprimento:** 2 (muito subótimo!)

Problema 6: Subsequência Crescente Mais Longa - LIS

Solução PD:

- Calcula dp para cada posição
- Encontra: $[2, 3, 7, 18]$ ou $[2, 5, 7, 18]$
- **Comprimento:** 4 (ótimo)

Aplicações: Análise de séries temporais, controle de versões, análise de tendências.

Problema 7: Escalonamento com Deadlines

Descrição do Problema:
Dadas n tarefas, cada uma com deadline d_i e penalidade p_i se não concluída no prazo, minimizar a penalidade total. Cada tarefa leva exatamente 1 unidade de tempo.

Abordagem Gulosa:

- **Estratégia:** ordenar tarefas por penalidade decrescente, tentar agendar cada uma no slot mais tarde possível antes do deadline
- **Algoritmo:** usar estrutura de slots, alocar tarefa no último slot livre $\leq d_i$
- **Complexidade:** $O(n^2)$ ou $O(n \log n)$ com Union-Find
- **Resultado:** Funciona muito bem! Produz solução ótima

Problema 7: Escalonamento com Deadlines

Abordagem de Programação Dinâmica:

- **Estado:** dp[máscara][tempo] = mínima penalidade usando subconjunto de tarefas
- **Recorrência:** considerar todas permutações válidas
- **Complexidade:** $O(2^n \times n)$ - exponencial!
- **Resultado:** Impraticável para $n > 20$

Problema 7: Escalonamento com Deadlines

Exemplo onde Guloso Funciona Perfeitamente:

Tarefa	Deadline	Penalidade
T1	2	60
T2	1	100
T3	3	20
T4	2	40

Problema 7: Escalonamento com Deadlines

Solução Gulosa:

- Ordenar por penalidade: T2(p=100), T1(p=60), T4(p=40), T3(p=20)
- T2: deadline=1 $\rightarrow$ agendar no slot 1
- T1: deadline=2 $\rightarrow$ agendar no slot 2
- T4: deadline=2 $\rightarrow$ slot 2 ocupado, slot 1 ocupado $\rightarrow$ não pode agendar
- T3: deadline=3 $\rightarrow$ agendar no slot 3
- **Agendadas:** T2, T1, T3 — **Perdida:** T4
- **Penalidade total:** 40 (ótimo)

Aplicação: Gerenciamento de projetos, sistemas de produção just-in-time.

Problema 8: Compra e Venda de Ações

Descrição do Problema:
Dado um array de preços de ações $P = [p_1, p_2, ..., p_n]$ ao longo de n dias, determinar o lucro máximo possível com no máximo k transações (cada transação = 1 compra + 1 venda).

Abordagem Gulosa (k ilimitado):

- **Estratégia:** comprar antes de cada subida, vender antes de cada queda
- **Algoritmo:** somar todos os incrementos positivos consecutivos
- Lucro = $\sum_{i=1}^{n-1} \max(0, p_{i+1} - p_i)$
- **Complexidade:** $O(n)$
- **Resultado:** Ótimo quando transações ilimitadas

Problema 8: Compra e Venda de Ações

Abordagem de Programação Dinâmica (k limitado):

- **Estado:** $dp[t][d]$ = lucro máximo com no máximo t transações até dia d
- **Recorrência:**
  $dp[t][d] = \max(dp[t][d-1], \max_{j<d}(p_d - p_j + dp[t-1][j]))$
- **Complexidade:** $O(k \times n^2)$ ou $O(k \times n)$ otimizado
- **Resultado:** Solução ótima para qualquer k

Problema 8: Compra e Venda de Ações

Exemplo Comparativo:

Preços: $P = [7, 1, 5, 3, 6, 4]$

Caso 1 - Transações Ilimitadas (Guloso):

- Dia 1→2: $1 - 7 = -6$ (não compra)
- Dia 2→3: compra em 1, vende em 5 → lucro $= 4$
- Dia 3→4: $3 - 5 = -2$ (não faz nada)
- Dia 4→5: compra em 3, vende em 6 → lucro $= 3$
- Dia 5→6: $4 - 6 = -2$ (não faz nada)
- **Lucro total:** $4 + 3 = 7$ (ótimo para k=∞)

Problema 8: Compra e Venda de Ações

Caso 2 - Máximo 1 Transação (PD com k=1):

- Melhor estratégia: comprar no mínimo global (dia 2, preço=1)
- Vender no máximo após compra (dia 5, preço=6)
- **Lucro total:** $6 - 1 = 5$ (ótimo para k=1)

Observação: Guloso só funciona quando $k \geq \lfloor n/2 \rfloor$ (transações ilimitadas). Para k pequeno, PD é necessário.

Problema 9: Subsequência Comum Mais Longa - LCS

Descrição do Problema:
Dadas duas sequências $X = [x_1, x_2, ..., x_m]$ e $Y = [y_1, y_2, ..., y_n]$, encontrar o comprimento da maior subsequência presente em ambas (não necessariamente contígua).

Abordagem Gulosa (Heurística):

- **Estratégia:** percorrer primeira sequência, marcar matches na segunda em ordem
- **Algoritmo:** para cada elemento de X, procurar próxima ocorrência em Y
- **Complexidade:** $O(m \times n)$
- **Resultado:** Não garante LCS ótima - ordem de escolha importa

Problema 9: Subsequência Comum Mais Longa - LCS

Abordagem de Programação Dinâmica:

- **Estado:** $dp[i][j]$ = comprimento da LCS de $X[1..i]$ e $Y[1..j]$
- **Recorrência:**

$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1 & \text{se } x_i = y_j \\ \max(dp[i-1][j], dp[i][j-1]) & \text{se } x_i \neq y_j \end{cases}$$

- **Complexidade:** $O(m \times n)$
- **Resultado:** Solução ótima garantida

Problema 9: Subsequência Comum Mais Longa - LCS

Exemplo Comparativo:

$X =$ "ABCDGH"
$Y =$ "AEDFHR"

Solução Gulosa (possível falha):

- Percorre X: A, B, C, D, G, H
- Matches em Y na ordem encontrada: A (pos 1), D (pos 3), H (pos 5)
- Subsequência: "ADH"
- **Comprimento:** 3 (pode não ser ótimo)

Problema 9: Subsequência Comum Mais Longa - LCS

Solução PD:

- Constrói matriz $dp[6][6]$
- Matches: A-A, D-D, H-H formam "ADH"
- Verifica todas possibilidades sistematicamente
- **Comprimento:** 3 (ótimo confirmado)
- LCS = "ADH"

Aplicações Práticas:

- Comando `diff` para comparar arquivos e mostrar mudanças
- Bioinformática: alinhamento de sequências de DNA/proteínas
- Detecção de plágio e similaridade de textos
- Sistemas de controle de versão (Git, SVN)

Problema 10: Distância de Edição

Descrição do Problema:
Dadas duas strings S_1 e S_2, encontrar o número mínimo de operações necessárias para transformar S_1 em S_2. Operações permitidas: inserção, remoção e substituição de caracteres.

Abordagem Gulosa (Heurística ingênua):

- **Estratégia:** percorrer strings em paralelo, substituir quando diferentes
- **Algoritmo:** contar caracteres diferentes nas mesmas posições
- **Complexidade:** $O(\min(m, n))$
- **Resultado:** Falha completamente - não considera inserções/remoções estratégicas

Problema 10: Distância de Edição

Abordagem de Programação Dinâmica (Levenshtein):

- **Estado:** $dp[i][j]$ = distância entre $S_1[1..i]$ e $S_2[1..j]$
- **Recorrência:**

$$dp[i][j] = \min \begin{cases} dp[i-1][j] + 1 & \text{(remoção)} \\ dp[i][j-1] + 1 & \text{(inserção)} \\ dp[i-1][j-1] + \text{custo} & \text{(substituição, custo=0 se igua} \end{cases}$$

- **Complexidade:** $O(m \times n)$
- **Resultado:** Solução ótima garantida

Problema 10: Distância de Edição

Exemplo Comparativo:

Transformar $S_1 =$ "kitten" em $S_2 =$ "sitting"

Solução Gulosa (falha):

- Compara posição a posição:
- k!=s, i!=i (igual!), t!=t (igual!), t!=t (igual!), e!=i, n!=n (igual!), -!=g
- Conta apenas substituições: pelo menos 3+ operações
- Não otimiza sequência de operações
- **Operações:** ≥ 3 (subótimo e mal calculado)

Problema 10: Distância de Edição

Solução PD:

- Operação 1: substituir 'k' por 's' $\rightarrow$ "sitten"
- Operação 2: substituir 'e' por 'i' $\rightarrow$ "sittin"
- Operação 3: inserir 'g' no final $\rightarrow$ "sitting"
- **Total de operações:** 3 (ótimo)

Aplicações Práticas:

- Corretores ortográficos (sugestão de palavras similares)
- Busca aproximada em bancos de dados
- Bioinformática: comparação de sequências genéticas
- Sistemas de reconhecimento de voz e OCR

Conclusões: Guloso vs Programação Dinâmica

Quando Usar Algoritmos Gulosos:

- Problema possui **propriedade gulosa**: escolha localmente ótima leva ao ótimo global
- Exemplos onde funciona: Mochila Fracionária, Escalonamento por Deadline, Árvores de Huffman
- **Vantagens:** rápidos ($O(n \log n)$ ou $O(n)$), simples de implementar
- **Cuidado:** sempre verificar se existe contraexemplo!

Quando Usar Programação Dinâmica:

- Problema possui **subestrutura ótima** e **subproblemas sobrepostos**
- Necessário quando guloso comprovadamente falha
- **Vantagens:** garante solução ótima, aplicável a mais problemas
- **Desvantagens:** mais lento ($O(n^2)$, $O(n \times m)$, $O(n \times W)$), mais complexo

Conclusões: Guloso vs Programação Dinâmica

Estratégia Prática:

1. Tente provar que a abordagem gulosa funciona
2. Se encontrar contraexemplo $\rightarrow$ use Programação Dinâmica
3. Para problemas conhecidos, consulte literatura estabelecida