

Introdução à Programação Dinâmica

Greedy Algorithm

Recursão: O que é?

Recursão é uma técnica de projeto de algoritmos que nos permite resolver um problema a partir da solução de instâncias menores do mesmo problema.

Recursão: Ideia Prática

Exemplo:

- Cada chamada é independente das outras
- **Exemplo:** $f(4)$ chamando $f(3)$
- Sempre há um **caso base** (ex.: $0! = 1$)

Modelo Mental

- Para $f(5)$, assumimos saber $f(4)$
- Para $f(4)$, assumimos saber $f(3)$
- Continua até o caso base

Princípio Fundamental

“Recursão = Implementação computacional da indução”

Recursão como Indução

Indução Simples

- **Base:** Valor inicial (ex.: $0! = 1$)
- **Passo:** Assumimos $f(n - 1)$ funciona
- **Princípio:** Caso menor correto \rightarrow caso atual correto

Indução Forte

- $f(n)$ depende de múltiplos casos menores
- **Exemplo:** Fibonacci

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

- Assume todos $f(0), \dots, f(n - 1)$ corretos

Paralelo: Recursão Indução

Recursão	Indução
Chama casos menores	Assume casos menores verdadeiros
Precisa dos resultados	Usa hipótese indutiva
Caso base para parar	Base garante início
Combina resultados	Passo constrói novo caso

Conclusão

- Recursão implementa indução computacionalmente
- Recursão com múltiplas chamadas = indução forte

Prova: Fatorial Recursivo

Correção do Algoritmo

- **Base:** $0! = 1$ (correto)
- **Hipótese:** $(n - 1)!$ calculado corretamente
- **Passo:** $n! = n \cdot (n - 1)!$ (correto)

Conclusão

Função faz $n \cdot \text{factorial}(n - 1) \rightarrow$ correta para n

Isso é uma prova por indução!

Prova: Fibonacci com Indução Forte

Teorema

Podemos calcular $F(n)$ para todo $n \in \mathbb{N}$

Prova

- **Base:** $F(0) = 1, F(1) = 1$ (dados)
- **Hipótese:** Sabemos calcular $F(k)$ para $k \leq n - 1$
- **Passo:** $F(n) = F(n - 1) + F(n - 2)$ (soma de conhecidos)

Conclusão

Por indução forte, calculamos todos $F(n)$

Problema das Escadas

Problema Clássico

- Escada com n degraus
- Em cada passo: subir 1, 2 ou 3 degraus
- **Pergunta:** Quantas maneiras diferentes de chegar ao topo?

Aplicação de Conceitos

- Recursão e decomposição de estado
- Relação com indução forte
- Padrão de prova formal com múltiplas dependências

Modelo Geral

Estado → Ações → Validação → Transição

Estado Atual

- **Estado** = degrau atual
- **Inicial**: level = 0 (chão)
- **Final**: level = n (solução)
- **Inválido**: level > n

Ações Possíveis

A partir de qualquer estado:

- Subir 1 degrau
- Subir 2 degraus
- Subir 3 degraus

Gera três subproblemas distintos

Validar e transitar

Validação

Antes de executar ação:

- **level + passo > n** → inválido
- **level + passo = n** → solução encontrada
- **level + passo < n** → continua recursão

Transição Recursiva

Se ação válida:

$$\text{ways}(\textit{level}) = \text{ways}(\textit{level} + 1) + \text{ways}(\textit{level} + 2) + \text{ways}(\textit{level} + 3)$$

Cada chamada = subproblema menor

Resumo Visual

Estado → Escolhas $\{+1, +2, +3\}$ → Validação → Transição

Paralelo: Recursão Indução Forte

Recursão do Problema

$$\text{ways}(n) = \text{ways}(n - 1) + \text{ways}(n - 2) + \text{ways}(n - 3)$$

- Para resolver n , resolvo $n - 1$, $n - 2$, $n - 3$
- Dependência múltipla de casos anteriores

Indução Forte Correspondente

Para provar $P(n)$:

- **Hipótese:** $P(1), P(2), \dots, P(n - 1)$ verdadeiras
- **Prova:** Usar todas para provar $P(n)$

Relação Fundamental

Recursão = Construção computacional

Indução Forte = Justificativa matemática

Definição do Problema

Entrada: um grafo caminho $G = (V, E)$ com

- pesos não negativos nos vértices, $w : V \rightarrow \mathbb{R}^+$
- Um grafo caminho é um grafo conexo sem bifurcações:
 - o grau de todos os vértices é dois,
 - exceto pelos dois vértices dos extremos que têm grau um.

Saída: um conjunto independente de peso máximo.

- Um conjunto é independente se nenhum par de vértices é adjacente,
- i.e., se nenhum par do conjunto tem aresta em comum.

Exemplo

Exemplo: Caminho com quatro vértices e pesos 1, 4, 5, 4

Antes de projetar um algoritmo usando a nova técnica,

- vamos testar as técnicas que já conhecemos,
- para entender as dificuldades do problema
- e limitações das técnicas.

Abordagem por Força Bruta

- Vamos enumerar todos os 2^n diferentes subconjuntos,
- descartar todos os que tem dois vértices adjacentes,
- e encontrar o de peso máximo dentre os que sobraram.

Vantagem

Esta abordagem encontra o resultado correto.

Desvantagem

Ela leva tempo $\Omega(2^n)$

Subestrutura Ótima

Conceito Fundamental

A parte central do projeto de algoritmos utilizando programação dinâmica:

- É encontrar a subestrutura ótima das soluções do problema
- Isso significa mostrar como uma solução ótima é composta de soluções ótimas para subproblemas menores
- A princípio isso serve para entender melhor o problema e reduzir o espaço soluções de uma busca exaustiva
- O impacto no tempo de execução pode ser muito maior

Princípio da Subestrutura Ótima

“Solução ótima contém soluções ótimas de subproblemas”

Caso 1: $v_n \notin S$

Sendo $G^1 = G \setminus \{v_n\}$ temos que S^1 é uma solução ótima em G^1 . Por contradição, Suponha que S^* é uma solução ótima para G^1 com peso maior que S . Como S^* é independente em G^1 , então também é independente em G . Portanto, S^* é uma solução para G com peso maior que S (contradição).

Caso 2: $v_n \in S$

Neste caso v_{n-1} não pode estar em S . Seja $G^u = G \setminus \{v_n, v_{n-1}\}$ e $S^u = S \setminus \{v_n\}$ Temos que S^u é uma solução ótima em G^u .

Por contradição, Suponha que S^* é uma solução ótima para G^1 com peso maior que S . Como S^* é independente em G^1 , então também é independente em G . Portanto, S^* é uma solução para G com peso maior que S (contradição).

Algoritmo Recursivo

De posse da subestrutura ótima, conseguimos descrever uma solução ótima para

- como a melhor entre uma solução ótima em:

$$G_1 = G \setminus \{v_n\}$$

$\{v_n\} \cup$ uma solução ótima em $G_2 = G \setminus \{v_n, v_{n-1}\}$

ConjIndRec($G = (V, E)$, w):

$|V| = 1$ **return** $w(v_1)$

$|V| = 2$ **return** $\max\{w(v_1), w(v_2)\}$

$S^1 = \text{ConjIndRec}(G^1 = G \setminus \{v_n\})$

$S^2 = \text{ConjIndRec}(G^2 = G \setminus \{v_n, v_{n-1}\})$

return $\max\{S^1, S^2 + w(v_n)\}$

Relação de Recorrência

Da subestrutura ótima temos que uma solução ótima para G é a melhor entre:

- uma solução ótima em $G' = G \setminus \{v_n\}$
- uma solução ótima em $G'' = G \setminus \{v_n, v_{n-1}\} \cup \{v_n\}$

$$A[i] = \max\{A[i - 1], A[i - 2] + w(v_i)\}$$

sendo $A[i]$ valor do ótimo para o caminho com os i primeiros vértices.

- Observem o padrão de formação dos subproblemas.
- Apenas vértices do extremo direito do caminho são removidos.
- Assim, cada subproblema é um prefixo do caminho.
- Como o caminho original tem n vértices, temos n diferentes subproblemas.

Programação Dinâmica

- De fato, o algoritmo recursivo gasta tempo exponencial
- apenas por ficar recalculando os mesmos subproblemas.
- Podemos torná-lo polinomial, se guardarmos numa tabela
- o valor de um subproblema na primeira vez que o calcularmos
- e verificarmos essa tabela antes de recalcularmos um subproblema.
- Essa técnica é conhecida como memorização (memoization)
- e tem uma relação próxima com programação dinâmica.

Problema da Mochila Binária (0-1 Knapsack)

Definição Formal do Problema

- **Entrada:** n itens, cada item i tem:
 - Peso $p_i > 0$
 - Valor $v_i > 0$
- **Capacidade:** $C \in \mathbb{Z}^+$
- **Objetivo:** Encontrar subconjunto $S \subseteq \{1, \dots, n\}$ que:
 - Respeita capacidade: $\sum_{i \in S} p_i \leq C$
 - Maximiza valor total: $\sum_{i \in S} v_i$

Notação

$I(n, C)$ = instância com n itens e capacidade C

- Itens implicitamente têm valores v_i e pesos p_i para $i = 1, \dots, n$

Subestrutura Ótima: Análise do Último Item

Foco no Item n

Considere instância $I(n, C)$ e solução ótima S . Para o último item n , temos:

Caso 1: $n \notin S$ ou Caso 2: $n \in S$

Caso 1: Item n NÃO Está na Solução

- S é solução ótima para $I(n - 1, C)$
- **Prova:** Se existisse S^* melhor para $I(n - 1, C)$, então S^* seria melhor para $I(n, C)$ (contradição)
- Valor ótimo: $v(S) =$ valor ótimo de $I(n - 1, C)$

Princípio da Subestrutura

“Solução ótima contém soluções ótimas de subproblemas”

Caso 2: Item n Está na Solução

Análise do Caso com Item Incluído

Se $n \in S$, então:

- Defina $S^1 = S \setminus \{n\}$ (remove o item n)
- S^1 só usa os $n - 1$ primeiros itens
- Restrição de capacidade: $\sum_{i \in S^1} p_i \leq C - p_n$

Subproblema Resultante

- S^1 é solução ótima para $I(n - 1, C - p_n)$
- **Prova:** Se existisse S^* melhor para $I(n - 1, C - p_n)$, então $S^* \cup \{n\}$ seria melhor para $I(n, C)$ (contradição)
- Valor ótimo: $v(S) = v_n + \text{valor ótimo de } I(n - 1, C - p_n)$

Formulação da Recorrência

Definição do Estado DP

$A[i, \alpha]$ = valor ótimo usando os i primeiros itens com capacidade α

- $i = 0, 1, \dots, n$ (número de itens considerados)
- $\alpha = 0, 1, \dots, C$ (capacidade disponível)

Recorrência Principal

$$A[i, \alpha] = \max \begin{cases} A[i - 1, \alpha] & (\text{não inclui item } i) \\ v_i + A[i - 1, \alpha - p_i] & (\text{inclui item } i, \text{ se } p_i \leq \alpha) \end{cases}$$

Condição de Contorno

- Se $p_i > \alpha$: segundo caso é inválido (capacidade insuficiente)
- Caso base: $A[0, \alpha] = 0$ (sem itens, valor zero para qualquer

Casos Base e Implementação

Casos Base

$$A[0, \alpha] = 0 \quad \text{para todo } \alpha = 0, 1, \dots, C$$

- Sem itens disponíveis → valor sempre zero
- Capacidade zero → ainda pode ter valor zero (mochila vazia)

Tratamento de Casos Especiais

- **Capacidade negativa:** Estado inválido, retornar $-\infty$
- **Item não cabe:** $p_i > \alpha \rightarrow$ apenas primeiro caso aplicável
- **Inicialização:** $A[0, \cdot] = 0$

Resposta Final

$$\text{Valor ótimo} = A[n, C]$$

Complexidade Computacional

- **Estados:** $(n + 1) \times (C + 1)$
- **Transições:** 2 por estado (quando aplicável)
- **Total:** $O(n \cdot C)$ (pseudo-polynomial)

Quando Usar Esta Abordagem

- **Problemas de seleção** com restrição de capacidade
- **Otimização combinatorial** com subestrutura ótima
- **Orçamento/allocação de recursos** com itens discretos

Limitações Importantes

- **Pseudo-polynomial:** Depende do valor de C
- **Não escalável** para capacidades muito grandes
- **Alternativas:** Algoritmos de aproximação para instâncias grandes

Problema Clássico: Longest Increasing Subsequence (LIS)

Definição do Problema

- **Subsequência:** Sequência que mantém ordem relativa original (não precisa ser contígua)
- **Crescente Estrita:** Cada elemento > elemento anterior
- **Objetivo:** Encontrar comprimento da maior subsequência crescente

Exemplo: Array [3, 2, 5, 4, 5, 3, 2, 5, 4, 5]

- LIS possível: 2 → 4 → 5 (comprimento 3)
- Outra LIS: 2 → 5 (comprimento 2)
- **LIS máxima:** Comprimento 6 em sequências maiores

Primeira Formulação por Programação dinâmica

Modelagem com Form 1

- **Estado:** $dp[i][l]$ = LIS mais longa do $level(i)$ até o final, com $last(l)$ como último elemento
- **i:** Índice atual no array sendo considerado
- **l:** Último elemento incluído na subsequência

Transições

$$dp[i][l] = \max \begin{cases} dp[i + 1][l] & (\text{não pegar}) \\ 1 + dp[l + 1][a[l]] & (\text{pegar se } a[l] > l) \end{cases}$$

Características

- Pensamento: "A partir deste level, qual a melhor sequência?"
- Complexidade: $O(n^2)$ estados com valores limitados
- **Otimização:** Usar índice do último em vez de valor

Vantagens

Vantagens de usar essa formulação

- A cada level, decidir: incluir ou não o elemento atual
- Só pode incluir se elemento > último escolhido
- Mantém estado do "último elemento" para verificar restrição

Abordagem com Form 2: Ending Form (Mais Natural)

Modelagem com Form 2

- **Estado:** $dp[i]$ = comprimento da LIS **terminando** em $a[i]$
- **Foco:** "Qual a melhor sequência que termina exatamente aqui?"
- **Resposta final:** $\max(dp[i])$ para $i = 0 \dots n - 1$

Relação de Recorrência

$$dp[i] = 1 + \max_{\substack{j < i \\ a[j] < a[i]}} dp[j]$$

- Base: $dp[i] = 1$ (sequência contendo só $a[i]$)
- Para cada i , encontra o melhor j anterior que pode estender a sequência

Visualização Form 2 para LIS

Exemplo: Array [2, 1, 5, 3, 6]

- $dp[0] = 1$ (só 2)
- $dp[1] = 1$ (só 1)
- $dp[2] = 2$ ($1 \rightarrow 5$ ou $2 \rightarrow 5$)
- $dp[3] = 2$ ($1 \rightarrow 3$ ou $2 \rightarrow 3$)
- $dp[4] = 3$ ($1 \rightarrow 3 \rightarrow 6$ ou $2 \rightarrow 3 \rightarrow 6$)
- **Resposta:** $\max = 3$

Comparação: Form 1 vs Form 2 para LIS

Form 1: Abordagem por Level

- **Pensamento:** "A partir daqui"
- **Estado:** $dp[level][last]$
- **Transições:** Para frente
- **Vantagem:** Natural para restrições sequenciais
- **Desvantagem:** Mais estados para gerenciar

Form 2: Ending Form

- **Pensamento:** "Terminando aqui"
- **Estado:** $dp[index]$
- **Transições:** De trás para frente
- **Vantagem:** Mais intuitivo para subsequências
- **Desvantagem:** Não captura todas as restrições complexas

Problema: Caminho de Soma Máxima em Grid 2D

Entendendo o Problema

Dada uma matriz $n \times m$, você começa no canto superior esquerdo $(0,0)$ e precisa chegar ao canto inferior direito $(n-1, m-1)$. Você só pode se mover para direita ou para baixo. Cada célula tem um valor (que pode ser positivo ou negativo), e o objetivo é encontrar o caminho que maximiza a soma dos valores das células visitadas.

Definição Formal do Problema

- **Objetivo:** Encontrar caminho de soma máxima de $(0,0)$ até $(n-1,m-1)$
- **Movimentos permitidos:** Apenas \rightarrow (direita) ou \downarrow (baixo)
- **Soma:** Acumular valores das células visitadas
- **Valores:** Podem ser positivos ou negativos

Form 2 - Ending Form Aplicada ao Caminho Máximo

Definição do Estado

$dp[r][c]$ = MAIOR soma de caminho terminando na célula (r, c)

- **Interpretação:** Para cada célula, qual a melhor forma de alcançá-la?
- **Foco:** "Terminando exatamente aqui"

Transições (2 Possibilidades)

$$dp[r][c] = arr[r][c] + \max \begin{cases} dp[r-1][c] & \text{se } r > 0 \quad (\text{veio DE CIMA}) \\ dp[r][c-1] & \text{se } c > 0 \quad (\text{veio DA ESQUERDA}) \end{cases}$$

Caso Base

$$dp[0][0] = arr[0][0]$$

Pensamento por Trás da Form 2

- "Para chegar em (r, c) , qual foi o melhor caminho anterior?"
- "Como estender soluções ótimas de células vizinhas?"
- Cada célula herda a melhor soma possível de seus predecessores

1. Pruning (Estados Inválidos)

- **Código:** `if (r < 0 || c < 0) return -INF;`
- **Razão:** Estados impossíveis devem retornar valor que nunca será escolhido
- **Implementação:** Usar -10^{18} ou LLONG_MIN

3. Valor Default (CRÍTICO!)

- **CERTO:** `int ans = -INF;`
- **Perigo:** Se usar 0 e todos valores forem negativos, resposta será errada!
- **Exemplo:** Caminho só com valores negativos deve retornar o "menos pior"

Análise de Complexidade

Quantificação do Desempenho

- **Estados:** $n \times m$ (cada célula da matriz)
- **Transições por estado:** 2 (cima + esquerda)
- **Total:** $O(n \times m)$

Eficiência na Prática

- **Linear** em relação ao tamanho da matriz
- Para $n = 1000, m = 1000$: 10^6 estados, 2×10^6 operações
- **Viável** mesmo para grids grandes

Comparação com Força Bruta

- **Força bruta:** $O(2^{n+m})$ caminhos possíveis
- **DP:** $O(nm)$ - melhoria exponencial!
- Demonstra o poder da programação dinâmica

Referências



FELICE, D. F.

Projeto e Análise de Algoritmos - PAA.

Departamento de Computação, Universidade Federal de São Carlos,
2025.

Disponível em: <https://www.aloc.ufscar.br/felice/ensino/2025s2paa/paa.php>

Acesso em: 2025.



CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C.

Algoritmos: Teoria e Prática.

3. ed. Rio de Janeiro: Elsevier, 2009.

944 p.

Estas referências foram fundamentais para o desenvolvimento dos conceitos de programação dinâmica e análise de algoritmos apresentados nesta apresentação.