Lista 6 — MC458 — Gabarito
Tema: Programação Dinâmica

Lembrete: Princípios de Programação Dinâmica (DP). A Programação Dinâmica é uma técnica de projeto de algoritmos aplicada a problemas de otimização e contagem que exibem duas propriedades fundamentais:

- **Subestrutura Ótima:** Uma solução ótima para o problema global é construída eficientemente a partir de soluções ótimas de seus subproblemas.
- **Sobreposição de Subproblemas:** O espaço total de subproblemas é pequeno (geralmente polinomial), permitindo memorizar resultados em uma tabela para evitar recomputações exponenciais.

Roteiro Geral para Solução de Problemas de DP:

i) **Definição do Estado:** Identifique o significado exato de cada célula da tabela de DP (ex.: $DP[i]$, $DP[i][j]$ ou $DP[i][w]$).

ii) **Casos Base:** Especifique os estados triviais e condições de contorno que não dependem de outros estados.

iii) **Relação de Recorrência (Transição):** Formule a equação matemática que expressa um estado em função de subproblemas estritamente menores.

iv) **Ordem de Preenchimento e Reconstrução:** Determine a ordem topológica de computação da tabela e o método de rastreamento (*backtracking*) para recuperar a solução explícita.

v) **Prova de Corretude:** Prove por indução matemática que a relação de recorrência produz a resposta correta para qualquer entrada.

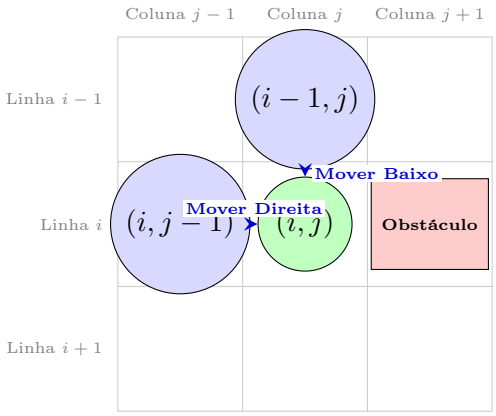
1. O Robô no Tabuleiro com Obstáculos

Resolução:

a) **Modelagem e Relação de Recorrência.**

Ideia Intuitiva: Como o robô só pode se mover para a **direita** ou para a **baixo**, para alcançar qualquer célula (i, j), o robô obrigatoriamente teve que passar imediatamente antes pela célula de cima $(i-1, j)$ ou pela célula da esquerda $(i, j-1)$. Portanto, o número total de caminhos para chegar em (i, j) é simplesmente a soma dos caminhos que chegam nesses dois vizinhos. Se a própria célula (i, j) contiver um obstáculo, é impossível pisar nela, logo a quantidade de caminhos válidos torna-se zero.

Visualização Gráfica da Transição de Estados no Grid:



Definição de Estado: Seja $DP[i][j]$ o número total de caminhos válidos partindo da célula inicial $(1,1)$ até a célula (i,j) respeitando os obstáculos.

Casos Base:

- Célula inicial $(1,1)$: $DP[1][1] = 1$ se $O[1][1] = 0$; caso contrário, $DP[1][1] = 0$.
- Primeira linha $(i = 1, j > 1)$: $DP[1][j] = DP[1][j-1]$ se $O[1][j] = 0$; caso contrário, 0.
- Primeira coluna $(j = 1, i > 1)$: $DP[i][1] = DP[i-1][1]$ se $O[i][1] = 0$; caso contrário, 0.

Relação de Recorrência $(i > 1, j > 1)$:

$$DP[i][j] = \begin{cases} 0 & \text{se } O[i][j] = 1 \\ DP[i-1][j] + DP[i][j-1] & \text{se } O[i][j] = 0 \end{cases}$$

b) **Prova de Corretude por Indução.**

Por que induzir na soma das coordenadas $s = i + j$?
Como o robô só se move para a direita ou para baixo, cada movimento avança exatamente uma unidade na linha ou na coluna, aumentando a soma $i+j$ em exatamente 1. Dessa forma, a quantia $s = i + j$ mede a "distância em passos"a partir da origem $(1,1)$ e define uma ordem topológica natural (em camadas diagonais): qualquer célula de onde o robô possa ter vindo para alcançar (i,j) tem soma de coordenadas igual a $s-1$.

Base da Indução $(s = 2)$: A única célula com $i + j = 2$ é a célula de origem $(1,1)$.

- Se $O[1][1] = 1$, a célula é um obstáculo. Nenhum caminho pode começar nela. O algoritmo define $DP[1][1] = 0$, o que está correto.
- Se $O[1][1] = 0$, existe exatamente 1 caminho válido (o caminho trivial de comprimento 0). O algoritmo define $DP[1][1] = 1$, correto.

Hipótese de Indução (HI): Assuma que para toda célula (i', j') com soma de coordenadas $i' + j' = k < s$, o valor $DP[i'][j']$ armazena exatamente a quantidade correta de caminhos válidos de $(1,1)$ até (i', j').

Passo Indutivo $(i+j = s)$: Considere uma célula arbitrária (i,j) tal que $i+j = s$.

- **Caso 1 $(O[i][j] = 1)$:** A célula é um obstáculo. Por definição do problema, nenhum caminho válido pode terminar em um obstáculo. Logo, o número exato de caminhos é 0, o que coincide com a fórmula $DP[i][j] = 0$.

- **Caso 2 ($O[i][j] = 0$):** A célula é livre. Todo caminho válido que chega a (i, j) deve fazer seu último passo vindo de $(i-1, j)$ ou de $(i, j-1)$. Como estes movimentos são mutuamente exclusivos, os conjuntos de caminhos são disjuntos. As células predecessoras possuem soma de coordenadas $(i-1)+j = s-1$ e $i+(j-1) = s-1$. Como $s-1 < s$, podemos aplicar a Hipótese de Indução em ambas: $DP[i-1][j]$ e $DP[i][j-1]$ contêm a contagem exata de caminhos até essas células.
  Pelo Princípio Aditivo da contagem combinatória, o número total de caminhos até (i, j) é a soma dos caminhos que vêm de cima com os que vêm da esquerda:

$$DP[i][j] = DP[i-1][j] + DP[i][j-1]$$

Portanto, por indução matemática, a fórmula computa a quantidade correta de caminhos para toda célula (i, j). □

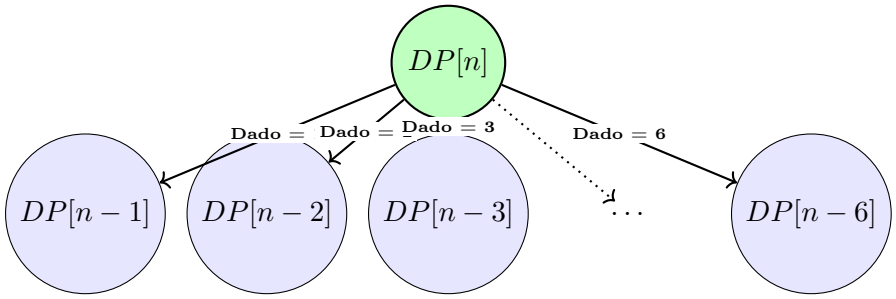
2. Lançamentos de Dado

Resolução:

a) **Modelagem e Relação de Recorrência.**

Ideia Intuitiva: Para obter uma soma final n através de lançamentos consecutivos de um dado de 6 faces, analisamos o valor obtido no **último lançamento**. Se o último dado mostrou a face $k \in \{1, 2, 3, 4, 5, 6\}$, os lançamentos anteriores deviam somar exatamente $n-k$. Como a última face pode ser qualquer número de 1 a 6 (desde que $k \leq n$), a quantidade total de sequências para somar n é a soma das maneiras de formar todas as quantias restantes $n-1, n-2, \ldots, n-6$.

Visualização Gráfica – Árvore de Recorrência Sem Sobreposições:



Caso Base:

$$DP[0] = 1$$

Justificativa: Existe exatamente 1 sequência para somar zero: a sequência vazia (fazer zero lançamentos). Para $n < 0$, temos $DP[n] = 0$.

Relação de Recorrência ($n \geq 1$):

$$DP[n] = \sum_{k=1}^{\min(n,6)} DP[n-k]$$

b) **Prova por Indução Forte para $1 \leq n \leq 6$.**

Base ($n = 1$): $DP[1] = DP[0] = 1$. Pela fórmula $2^{1-1} = 2^0 = 1$. A base se sustenta.

Hipótese de Indução Forte (HI): Assuma que $DP[m] = 2^{m-1}$ para todo m tal que $1 \leq m < n$, em que $n \leq 6$.

Passo Indutivo: Para $n \leq 6$, a somatória inclui todos os termos desde $DP[0]$ até $DP[n-1]$:

$$DP[n] = DP[n-1] + DP[n-2] + \cdots + DP[1] + DP[0]$$

Substituindo a HI nos termos 1 até $n-1$ e lembrando que $DP[0] = 1$:

$$DP[n] = 2^{n-2} + 2^{n-3} + \cdots + 2^0 + 1$$

Usando a soma da Progressão Geométrica $\sum_{j=0}^{n-2} 2^j = 2^{n-1} - 1$:

$$DP[n] = (2^{n-1} - 1) + 1 = 2^{n-1}$$

Logo, a fórmula $DP[n] = 2^{n-1}$ vale para todo $1 \leq n \leq 6$. □

c) **Quebra do Padrão em $n = 7$.**

Calculando $DP[7]$ pela recorrência (que só soma os últimos 6 termos):

$$DP[7] = DP[6] + DP[5] + DP[4] + DP[3] + DP[2] + DP[1]$$

Substituindo os valores conhecidos ($DP[m] = 2^{m-1}$ para $1 \leq m \leq 6$):

$$DP[7] = 32 + 16 + 8 + 4 + 2 + 1 = 63$$

No entanto, a fórmula fechada preveria $2^{7-1} = 2^6 = 64$.

Explicação da Falha: A fórmula 2^{n-1} incluía o termo $DP[0] = 1$, que representaria tirar a face 7 em um único lançamento. Como o dado só possui faces de 1 a 6, é **impossível** somar 7 com apenas 1 lançamento. A sequência (7) não existe, fazendo com que $DP[7] = 63 \neq 64$.

3. A Mochila do Aventureiro (Knapsack 0/1)

Resolução:

a) **Modelagem e Relação de Recorrência.**

Ideia Intuitiva ("Pegar ou Não Pegar"): Ao analisar o i-ésimo item com peso w_i e valor v_i, temos uma decisão binária simples:

1) **Não pegar o item i:** O valor máximo obtido é exatamente o mesmo valor que conseguiríamos usando apenas os primeiros $i-1$ itens com a mesma capacidade w.
2) **Pegar o item i:** Essa escolha só é possível se o peso do item couber na mochila ($w_i \leq w$). Ao pegar o item i, ganhamos seu valor v_i, porém reduzimos a capacidade disponível para $w - w_i$. O valor total será v_i somado ao valor ótimo obtido usando os primeiros $i-1$ itens para a capacidade restante $w - w_i$.

A Programação Dinâmica escolhe a opção que maximiza o valor final.

Definição de Estado: Seja $DP[i][w]$ o valor máximo acumulado considerando apenas os primeiros i objetos sob limite de capacidade w.

Casos Base:

- $DP[0][w] = 0$, para todo $0 \leq w \leq W$ (sem objetos disponíveis).
- $DP[i][0] = 0$, para todo $0 \leq i \leq n$ (mochila sem capacidade).

Relação de Recorrência ($1 \leq i \leq n, 1 \leq w \leq W$):

$$DP[i][w] = \begin{cases} DP[i-1][w] & \text{se } w_i > w \\ \max\Big(DP[i-1][w],\ v_i + DP[i-1][w-w_i]\Big) & \text{se } w_i \leq w \end{cases}$$

b) **Prova de Subestrutura Ótima por Contradição.**

Definição Formal de OPT: Definimos $OPT(i, w)$ como o valor da solução ótima (máximo valor total possível) ao selecionar um subconjunto dos itens $\{1, 2, \ldots, i\}$ cuja soma dos pesos não excede w.

Teorema: Seja $S \subseteq \{1, \ldots, i\}$ um subconjunto de itens que atinge a solução ótima $OPT(i, w)$. Se o item $i \in S$, então o subconjunto $S' = S \setminus \{i\}$ deve ser uma solução ótima para os itens $\{1, \ldots, i-1\}$ com capacidade limitada a $w - w_i$, isto é, o valor total dos itens em S' é igual a $OPT(i-1, w-w_i)$.

Demonstração. Provemos por contradição.
Suponha que $S' = S \setminus \{i\}$ **não seja** uma solução ótima para os primeiros $i-1$ itens com capacidade $w - w_i$.
Isso significa que o valor acumulado por S', dado por $V(S') = \sum_{j \in S'} v_j$, é estritamente menor do que o valor ótimo $OPT(i-1, w-w_i)$. Portanto, deve existir um outro subconjunto válido de itens $T \subseteq \{1, \ldots, i-1\}$ tal que:

$$\sum_{j \in T} w_j \leq w - w_i \qquad \text{e} \qquad V(T) = \sum_{j \in T} v_j > V(S')$$

Agora, construa um novo conjunto de itens $T^* = T \cup \{i\}$ para o problema original de capacidade w:

1) **Viabilidade de Peso:**

$$\text{Peso}(T^*) = \left(\sum_{j \in T} w_j\right) + w_i \leq (w - w_i) + w_i = w$$

Logo, T^* é uma solução perfeitamente viável para a capacidade w.

2) **Valor Acumulado:**

$$V(T^*) = V(T) + v_i > V(S') + v_i = V(S) = OPT(i, w)$$

Isso implica que T^* é uma solução viável com valor estritamente maior do que $OPT(i, w)$, o que é uma **contradição**, pois por definição $OPT(i, w)$ é o valor máximo possível.

Portanto, a suposição inicial é falsa: S' é obrigatoriamente uma solução ótima para a capacidade $w - w_i$, demonstrando a propriedade de subestrutura ótima. □

c) **Reconstrução da Solução (Backtracking).**

Partindo da célula final (n, W) da matriz DP:

1) Inicialize um conjunto vazio $S \leftarrow \emptyset$, $i \leftarrow n$, $w \leftarrow W$.
2) Enquanto $i > 0$ e $w > 0$:
   - Se $DP[i][w] \neq DP[i-1][w]$, conclui-se que o item i foi incluído na solução ótima. Adicione i a S, atualize $w \leftarrow w - w_i$ e decrementa $i \leftarrow i - 1$.
   - Caso contrário ($DP[i][w] == DP[i-1][w]$), o item i não foi incluído. Apenas decremente $i \leftarrow i - 1$.
3) Retorne o conjunto S.

4. O Caixa Eletrônico (Troco de Moedas)

Resolução:

a) **Modelagem e Relação de Recorrência.**

Ideia Intuitiva: Para fornecer um troco no valor exato v usando a menor quantidade de moedas, experimentamos usar cada moeda disponível $c \in C$ (com $c \leq v$) como a "última moeda"do troco. Se usarmos a moeda c, o problema se reduz a encontrar o troco ótimo para o valor restante $v - c$. O algoritmo testa todas as moedas viáveis e escolhe aquela que resulta no menor número total de moedas $(1 + DP[v-c])$.

Definição de Estado: Seja $DP[v]$ o menor número de moedas necessário para formar a quantia v.

Casos Base e Estados Inválidos:

- $DP[0] = 0$ (o valor 0 exige 0 moedas).
- Para quantias impossíveis ou negativas ($v < 0$), definimos $DP[v] = \infty$.

Relação de Recorrência ($v \geq 1$):

$$DP[v] = 1 + \min_{\substack{c \in C \\ c \leq v}} DP[v - c]$$

b) **Prova de Corretude por Indução Forte.**

Definição de $OPT(v)$: Seja $OPT(v)$ a quantidade mínima absoluta de moedas necessária para somar v.

Base ($v = 0$): $DP[0] = 0 = OPT(0)$. Correto.

Hipótese de Indução Forte (HI): Assuma que $DP[u] = OPT(u)$ para todo inteiro $0 \leq u < v$.

Passo Indutivo ($v > 0$): Seja $c^* \in C$ a última moeda usada em uma solução ótima que forma o valor v.
O valor restante $v - c^*$ deve ser formado de maneira ótima usando $OPT(v) - 1$ moedas. Como $v - c^* < v$, aplicamos a HI: $DP[v - c^*] = OPT(v) - 1$.
A equação da DP calcula o mínimo sobre todas as moedas possíveis $c \leq v$:

$$DP[v] = 1 + \min_{c \leq v} DP[v - c] \leq 1 + DP[v - c^*] = 1 + (OPT(v) - 1) = OPT(v)$$

Como nenhuma combinação válida de moedas pode usar menos do que $OPT(v)$ moedas, temos também $DP[v] \geq OPT(v)$. Portanto, $DP[v] = OPT(v)$. □

c) **Falha do Algoritmo Guloso (Contraexemplo).**

Considere as moedas $C = \{1, 3, 4\}$ e a quantia alvo $V = 6$.

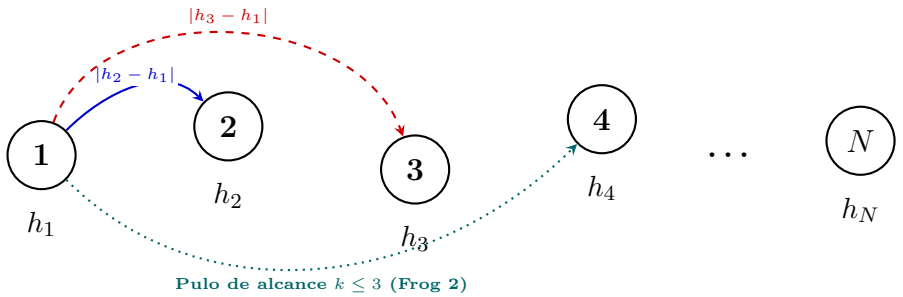
- **Abordagem Gulosa:** Sempre pega a maior moeda possível.
  1) Pega a moeda 4 (resta 2).
  2) Pega a moeda 1 (resta 1).
  3) Pega a moeda 1 (resta 0). Total: **3 moedas** ($\{4, 1, 1\}$).
- **Solução Ótima (DP):** Seleciona duas moedas de valor 3. Total: **2 moedas** ($\{3, 3\}$).

Conclusão: A escolha gulosa imediata pela moeda 4 impede o algoritmo de encontrar a solução global ótima de 2 moedas.

5. O Pulo do Sapo

Resolução:
Ideia Intuitiva: Para atingir a pedra i, o sapo veio de alguma pedra anterior $i-j$, onde o pulo j respeita o alcance máximo permitido. A energia gasta no último pulo é o valor absoluto da diferença de altura $|h_i - h_{i-j}|$. O sapo deve escolher a pedra anterior que minimize a soma da energia acumulada até ela mais o custo do pulo final.

Ilustração Gráfica do Problema (Sem Sobreposição de Arcos):



a) **Salto Curto (Frog 1: alcance $k = 2$).**

Seja $DP[i]$ o custo mínimo de energia para ir da pedra 1 à pedra i.

Casos Base: $DP[1] = 0$ e $DP[2] = |h_2 - h_1|$.

Relação de Recorrência ($i \geq 3$):

$$DP[i] = \min\Big(DP[i-1] + |h_i - h_{i-1}|,\ DP[i-2] + |h_i - h_{i-2}|\Big)$$

b) **Salto Longo (Frog 2: alcance $k \geq 1$) e Complexidade.**

Relação de Recorrência Geral ($i \geq 2$):

$$DP[i] = \min_{1 \leq j \leq \min(k,\, i-1)} \Big(DP[i-j] + |h_i - h_{i-j}|\Big)$$

com o caso base $DP[1] = 0$.

Complexidade Assintótica:

- **Tempo Execução:** $O(N \cdot k)$, pois o algoritmo calcula N estados e cada um realiza uma minimização testando até k predecessores.
- **Espaço Auxiliar:** $O(N)$ para a tabela de DP (que pode ser otimizado para $O(k)$ mantendo apenas as últimas k posições).

c) **Prova de Otimalidade por Indução Forte.**

Base ($i = 1$): $DP[1] = 0$. Na pedra inicial não há gasto de energia. Correto.

HI: Assuma que $DP[m] = OPT(m)$ para todo $1 \leq m < i$.

Passo Indutivo ($i > 1$): Seja P^* um caminho de menor energia até a pedra i, e seja $i - j^*$ a pedra de onde o sapo saltou diretamente para i. O salto j^* é válido, portanto $1 \leq j^* \leq \min(k, i-1)$.
Pelo Princípio da Otimalidade, o subcaminho do início até a pedra $i-j^*$ deve gastar energia mínima. Pela HI, esse custo é $DP[i-j^*]$. O custo ótimo total é:

$$OPT(i) = DP[i - j^*] + |h_i - h_{i-j^*}|$$

Como a recorrência testa todos os saltos possíveis $j \in \{1, \ldots, \min(k, i-1)\}$ e escolhe o mínimo, garante-se que $DP[i] \leq OPT(i)$. E como nenhum caminho pode custar menos que $OPT(i)$, segue que $DP[i] = OPT(i)$. □

6. Alinhamento de DNA (Maior Subsequência Comum – LCS)

Resolução:

a) **Formulação da Recorrência.**

Ideia Intuitiva: Comparando os últimos caracteres das cadeias $A[1..i]$ e $B[1..j]$:

- Se $a_i = b_j$, esse caractere obrigatoriamente faz parte de uma LCS estendida. Adicionamos 1 ao resultado da LCS dos prefixos restantes $A[1..i-1]$ e $B[1..j-1]$.
- Se $a_i \neq b_j$, eles não podem ser combinados simultaneamente ao final. A melhor subsequência virá de ignorar a_i (testando $A[1..i-1]$ com $B[1..j]$) ou de ignorar b_j (testando $A[1..i]$ com $B[1..j-1]$).

Definição de Estado: Seja $DP[i][j]$ o tamanho da LCS entre os prefixos $A[1..i]$ e $B[1..j]$.

Casos Base: $DP[0][j] = 0$ e $DP[i][0] = 0$.

Relação de Recorrência ($1 \leq i \leq m, 1 \leq j \leq n$):

$$DP[i][j] = \begin{cases} 1 + DP[i-1][j-1] & \text{se } a_i = b_j \\ \max\Big(DP[i-1][j],\ DP[i][j-1]\Big) & \text{se } a_i \neq b_j \end{cases}$$

b) **Prova Teórica para o Caso $a_i = b_j$.**

Demonstraremos que $DP[i][j] = 1 + DP[i-1][j-1]$ provando as duas desigualdades:

Parte 1 ($DP[i][j] \geq 1 + DP[i-1][j-1]$): Seja Z' uma LCS de $A[1..i-1]$ e $B[1..j-1]$, de tamanho $|Z'| = DP[i-1][j-1]$. Como $a_i = b_j$, podemos concatenar este caractere ao final de Z', gerando a sequência $Z = Z' \cdot a_i$. Z é uma subsequência comum válida de $A[1..i]$ e $B[1..j]$ com tamanho $|Z| = |Z'| + 1 = 1 + DP[i-1][j-1]$. Como $DP[i][j]$ é o tamanho da *maior* subsequência comum:

$$DP[i][j] \geq 1 + DP[i-1][j-1]$$

Parte 2 ($DP[i][j] \leq 1 + DP[i-1][j-1]$): Seja $Z = z_1 z_2 \dots z_k$ uma LCS qualquer de $A[1..i]$ e $B[1..j]$, com $|Z| = k = DP[i][j]$. O último caractere z_k deve ser obrigatoriamente $a_i = b_j$. Se omitirmos z_k, a sequência restante Z_{k-1} é uma subsequência comum válida de $A[1..i-1]$ e $B[1..j-1]$, possuindo tamanho $k-1$. Como por definição $DP[i-1][j-1]$ é o tamanho máximo para esses prefixos:

$$DP[i-1][j-1] \geq k - 1 = DP[i][j] - 1 \implies DP[i][j] \leq 1 + DP[i-1][j-1]$$

Combinando a Parte 1 e a Parte 2, conclui-se que $DP[i][j] = 1 + DP[i-1][j-1]$. □

c) **Reconstrução da String.**

Rastreando a matriz a partir de (m, n): se $a_i = b_j$, adicionamos o caractere à LCS e movemos na diagonal $(i-1, j-1)$. Caso contrário, movemos para a célula de maior valor entre $DP[i-1][j]$ e $DP[i][j-1]$.

7. O Fotógrafo de Silhuetas (Maior Subsequência Crescente – LIS)

Resolução:

a) **Modelagem e Relação de Recorrência.**

Ideia Intuitiva: Para encontrar a maior sequência crescente que **termina obrigatoriamente** no elemento h_i, olhamos para todos os elementos anteriores h_j (com $j < i$). Se $h_j < h_i$, a pessoa j pode anteceder a pessoa i na foto. Escolhemos o elemento j que possui a maior subsequência acumulada $DP[j]$ e acrescentamos a pessoa i ao final (somando +1).

Definição de Estado: Seja $DP[i]$ o comprimento da LIS que termina estritamente no elemento h_i.

Relação de Recorrência ($1 \leq i \leq n$):

$$DP[i] = 1 + \max\Big(\{0\} \cup \{DP[j] \mid 1 \leq j < i \text{ e } h_j < h_i\}\Big)$$

Resposta Global: $\text{LIS}_{\text{global}} = \max_{1 \leq i \leq n} DP[i]$.

b) **Prova de Corretude por Indução.**

Base ($i = 1$): $DP[1] = 1$. A única subsequência terminando em h_1 é o conjunto $[h_1]$. Correto.

HI: Assuma que $DP[j]$ está correto para todo $1 \leq j < i$.

Passo Indutivo (i): Seja S^* a LIS mais longa que termina em h_i.

- Se $|S^*| = 1$, não há antecessores menores. A fórmula devolve $1 + 0 = 1$.
- Se $|S^*| > 1$, seja h_{j^*} ($j^* < i$) o elemento imediatamente anterior a h_i em S^*. Como S^* é crescente, $h_{j^*} < h_i$. O subconjunto $S^* \setminus \{h_i\}$ é uma LIS ótima terminando em h_{j^*}. Pela HI, seu tamanho é $DP[j^*]$. Logo $|S^*| = 1 + DP[j^*]$.

A recorrência avalia todos os $j < i$ válidos ($h_j < h_i$) e seleciona o valor máximo, garantindo $DP[i] = |S^*|$. □

c) **Reconstrução e Complexidade.**

Complexidade Assintótica: Tempo $O(n^2)$ (pois cada elemento checa todos os anteriores) e espaço auxiliar $O(n)$.

Reconstrução: Mantemos um vetor de ponteiros $parent[1..n]$. Ao calcular $DP[i]$, salvamos $parent[i] \leftarrow j^*$ para o índice j^* que maximizou a expressão. No final, partimos do índice i^* de valor máximo global e seguimos os ponteiros de $parent$ até encontrar 0, invertendo a lista de elementos obtida.