Tópicos Especiais em Algoritmos Programação Dinâmica Editorial

Daniel Saad Nogueira Nunes

UVa 357: Let Me Count The Ways

Tome C[1,5] = (1,5,10,25,50) o vetor das moedas (note que está indexado de 1). Seja T(i,j) a quantidade de formas de pagar um valor j utilizando os i primeiras moedas através de uma quantidade qualquer delas. Podemos definir T(i,j) recursivamente como:

$$T(i,j) = \left\{ \begin{array}{l} 1, \; i=0 \; \mathrm{e} \; j=0 \\ 0, \; i=0 \; \mathrm{e} \; j>0 \\ T(i-1,j), \; \; \mathrm{se} \; C[i]>j \\ T(i-1,j)+T(i,j-C[i]), \; \mathrm{caso} \; \mathrm{contrário} \end{array} \right.$$

A interpretação das duas primeiras igualdades é a seguinte: existe apenas uma forma de pagar 0 centavos usando 0 moedas: é não fazendo nada. E existe 0 formas pegar um valor positivo utilizando 0 tipos de moeda. Este são os casos bases da relação de recorrência. O terceiro caso ocorre quando o valor da i-ésima moeda é superior ao valor a ser pago (j), assim a solução de T(i,j) é obtida ao resolver o mesmo problema para as i-1 primeiras moedas, logo, através de T(i-1,j). O quarto caso contempla a situação em que é possível utilizar a i-ésima moeda, possivelmente múltiplas vezes. Assim a solução de T(i,j) engloba a quantidade prevista em T(i-1,j) mais T(i,j-C[i]).

Através de uma tabela de programação dinâmica de tamanho $n+1 \times m+1$, sendo m o valor a ser pago, é possível computar cada célula T[i][j] como descrito na relação de recorrência. O resultado estará em T[n][m].

Detalhes de Implementação

Como cada linha da tabela depende imediatamente da anterior, é possível salvar memória e manter apenas duas linhas, sobrescrevendo elas conforme

a computação avança.

Complexidade

Esta solução possui complexidade $\Theta(m)$, visto que existe um número constante de moedas e cada célula da matriz de programação dinâmica pode ser computada em tempo constante.

UVA 10003: Cutting Sticks

Tome C[0, n+1] o vetor com os pontos de corte. Neste vetor, temos que C[0] = 0 e C[n+1] = l, em que l é o tamanho da barra. Os valores C[1, n] são os mesmos da entrada.

A solução pode ser modelada recursivamente como:

$$T(l,r) = \begin{cases} T(l,r) = 0, & l+1 = r \\ \min_{l < k < r} (T(l,k) + T(k,l) + C[r] - C[l]), & l+1 < r \end{cases}$$

Aqui, T(l,r) é a solução para a barra começando do segmento C[l] e terminando no segmento C[r]. A interpretação é a seguinte: temos que escolher o corte da barra de tamanho C[r] - C[l] que minimiza o valor a ser pago. Logo, buscamos o corte de custo mínimo dentre todos os possíveis segmentos entre C[l] e C[r]. O caso base é justamente quando só temos dois segmentos, isto é, l+1=r, pois não há possibilidade de cortes e portanto o custo é 0.

Utilizando uma tabela de *memoization* e uma solução recursiva a solução decorre imediatamente da relação de recorrência.

Complexidade

O algoritmo descrito possui custo $O(n^3)$, haja vista que existem $\binom{n}{2} \in O(n^2)$ possibilidades de chamadas recursivas e o custo no corpo da função é O(n).

UVa 10980: Lowest Price in Town

Seja p o valor unitário do item e seja $V[0, n-1] = ((a_0, b_0), (a_1, b_1), \dots, (a_{n-1}, b_{n-1}))$ o vetor com os pares: número de itens e preço (a_i, b_i) .

Suponha que T(i,j) seja o menor custo possível para comprar j ou mais itens utilizando as i primeiras promoções. T(i,j) pode ser modelado da seguinte forma:

$$T(i,j) = \begin{cases} p \cdot j, & i = 0\\ \min(T(i-1,j), T(i,j-V[i-1].a) + V[i-1].b) & i > 0 \end{cases}$$

Se não utilizamos nenhuma promoção (i=0), então a solução de menor custo é simplesmente pagar o preço unitário p multiplicado pela quantidade de itens j. Caso contrário, a solução pode ser obtida ao minimizar dentre a solução desconsiderando a i-ésima promoção T(i-1,j) e a solução considerando a i-ésima promoção: T(i,j-V[i-1].a)+V[i-1].b.

O algoritmo é facilmente implementável a partir da relação de recorrência e uma tabela M de programação dinâmica. A solução de menor custo para escolher k itens estará em M[n][k].

Complexidade

O algoritmo descrito possui complexidade de $\Theta(n \cdot k)$.

UVa 760: DNA Sequencing

Queremos descobrir dentre uma string X[1,n] e uma string Y[1,m], indexadas de 1, o tamanho da substring comum mais longa. Não confundir com a maior subsequência comum. Existem algoritmos mais eficientes para este problema do que o que será apresentado, mas optou-se aqui por uma solução baseada em programação dinâmica.

Seja T(i,j) o tamanho da maior substring comum considerando os prefixos X[1,i] e Y[1,j] de modo que a substring comum finalize obrigatoriamente com X[i] e Y[j]. T(i,j) pode ser modelado da seguinte forma:

$$T(i,j) = \begin{cases} 0, \ i = 0 \text{ ou } j = 0 \\ T(i-1,j-1) + 1, \text{ se } X[i] = Y[j] \text{ e } i,j > 0 \\ 0, \text{ caso contrário} \end{cases}$$

A explicação é simples. Se i=0 ou j=0, umas das strings é vazia, logo o tamanho da substring comum mais longa é 0. Caso contrário, temos os prefixos X[1,i] e Y[1,j], assim se X[i]=Y[j] a solução de T(i,j) é computada diretamente da solução de T(i-1,j-1) somada com 1, já que os caracteres X[i] e Y[j] são idênticos. Se $X[i] \neq Y[j]$, então T(i,j) só pode ser zero, pela definição de T.

Através de uma tabela de programação dinâmica M é possível computar todos os valores M[i][j] diretamente da relação de recorrência apresentada.

Detalhes de Implementação

Para recuperar as substrings comuns mais longas propriamente ditas, Basta varrer a matriz M e, sempre que M[i][j] possuir o valor máximo, suponha k, é possível concluir que a string X[i-k+1,i] (ou Y[j-k+1,j]) é uma substring comum mais longa. Com a ajuda de um conjunto, é possível armazenar todas as substrings comuns mais longas distintas na ordem lexicográfica.

Complexidade

A computação da tabela leva tempo $\Theta(n \cdot m)$.