PCC104 - Projeto e Análise de Algoritmos

Marco Antonio M. Carvalho

Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto

Conteúdo

Fonte

Este material é baseado nos livros

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
- S. Halim. *Competitive Programming*. 3rd Edition, 2013.
- Ian Parberry and William Gasarch. *Problems on Algorithms*. Second Edition, 2002.
- Ian Parberry *Lecture Notes on Algorithm Analysis and Complexity Theory*. Fourth Edition, 2001.

Licença

Ian Parberry e William Gasarch, *Problems on Algorithms*

"*Dynamic programming is a fancy name for [recursion] with a table. Instead of solving subproblems recursively, solve them sequentially and store their solutions in a table.*

The trick is to solve them in the right order so that whenever the solution to a subproblem is needed, it is already available in the table.

Dynamic programming is particularly useful on problems for which divide-and-conquer appears to yield an exponential number of subproblems, but there are really only a small number of subproblems repeated exponentially often.

In this case, it makes sense to compute each solution the first time and store it away in a table for later use, instead of recomputing it recursively every time it is needed."

Observação

A palavra **programação** na expressão "programação dinâmica" não tem relação direta com programação de computadores.

Ela significa **planejamento** e refere-se à construção da tabela que armazena as soluções dos subproblemas.

Introdução

A **Programação Dinâmica** (ou PD) é talvez o paradigma de solução de problemas mais desafiantes dentre os outros vistos.

Este paradigma pode "parecer mágica", até que tenhamos visto exemplos o suficiente, tornando-o relativamente fácil de ser aplicado.

É necessário nos assegurarmos de termos dominado todos os outros paradigmas anteriores antes de continuar.

Recursões e recorrências também serão vistas novamente.

Contextualização

Como visto anteriormente, problemas combinatórios nos pedem que determinemos a melhor solução possível, respeitando algumas restrições.

O ***Backtracking*** busca por todas as soluções possíveis (explícita ou implicitamente) e seleciona a melhor. Logo, possui corretude, mas é inviável para problemas grandes.

Algoritmos Gulosos tomam sempre a melhor decisão a cada instante. Porém, sem provas de corretude, falham em obter a solução ótima.

Divisão e Conquista divide o problema original em subproblemas menores e "mais fáceis"

- No entanto, pode resolvê-los repetidamente;
- O balanceamento pode ser primordial.

Contextualização

Se a soma do tamanho dos subproblemas é $O(n)$, temos uma boa probabilidade de que o algoritmo recursivo tenha complexidade polinomial.

Entretanto, se temos n subproblemas de tamanho $n-1$ cada, a tendência é de que o algoritmo recursivo associado tenha complexidade exponencial.

A Programação Dinâmica nos fornece uma maneira de projetar algoritmos que:

- Sistematicamente exploram todas as possibilidades (corretude);
- Evitam computação redundante (eficiência).

Características

Tais algoritmos são **quase** sempre definidos por recursividade:

- Definem a solução para um problema em termos da solução de problemas menores;
- De certa forma, lembram D&C e algoritmos gulosos.

Um possível defeito na busca recursiva é a computação redundante de subproblemas, ou a exploração redundante do espaço de busca

- Para contornar esta situação, podemos armazenar os resultados dos subproblemas já resolvidos;
- Em parte, por isto o *Backtracking* é ineficiente.

Características

À medida em que resolvemos subproblemas, armazenamos os resultados parciais, acelerando o algoritmo:

- Para cada subproblema inédito, o resolvemos e armazenamos o resultado;
- Para os subproblemas repetidos, apenas consultamos o resultado.

É importante primeiro nos certificarmos de que o algoritmo recursivo é correto, depois o aceleramos.

As soluções dos subproblemas são mantidas em uma **tabela**, a qual é consultada a cada subproblema encontrado.

Cada entrada na tabela é chamada de **estado** ou **estágio**.

Caracterização dos Problemas

A PD é aplicável a problemas que possuam as seguintes propriedades:

Formulação Recursiva Bem Definida: no caso de formulação recursiva, esta não deve possuir ciclos.

Subestrutura Ótima: O problema pode ser dividido sucessivamente, e a combinação das soluções ótimas dos subproblemas corresponde à solução ótima do problema original.

Superposição de Subproblemas: O espaço de subproblemas é pequeno e eles se repetem com frequência durante a solução do problema original.

Tópicos de Projeto

Ao contrário do paradigma D&C, o balanceamento ideal dos subproblemas requer o tamanho deles seja não muito distante do tamanho do problema original.

De fato, quando eles são menores que o problema original por um fator multiplicativo, a abordagem se encaixa melhor em D&C.

Tópicos de Projeto

A definição da estrutura da tabela em que são armazenadas as soluções é muito importante: a pesquisa de uma entrada e a recuperação da solução devem ser eficientes

- Quantas dimensões terá a tabela?
- O quê significa cada dimensão da tabela?
- Como determinar o(s) índice(s) de cada entrada da tabela?
- Será necessário informar somente o valor da função objetivo, ou também os elementos da solução?

Top-Down vs. Bottom-Up

A PD pode ser empregada em duas formas:

Top-Down: O problema original é decomposto em subproblemas que são resolvidos recursivamente e combinados para obtenção da solução;

Bottom-Up: Não se utiliza recursão. Os subproblemas são resolvidos e combinados sucessivamente de maneira a construir a solução do problema original.

Top-Down vs. Bottom-Up

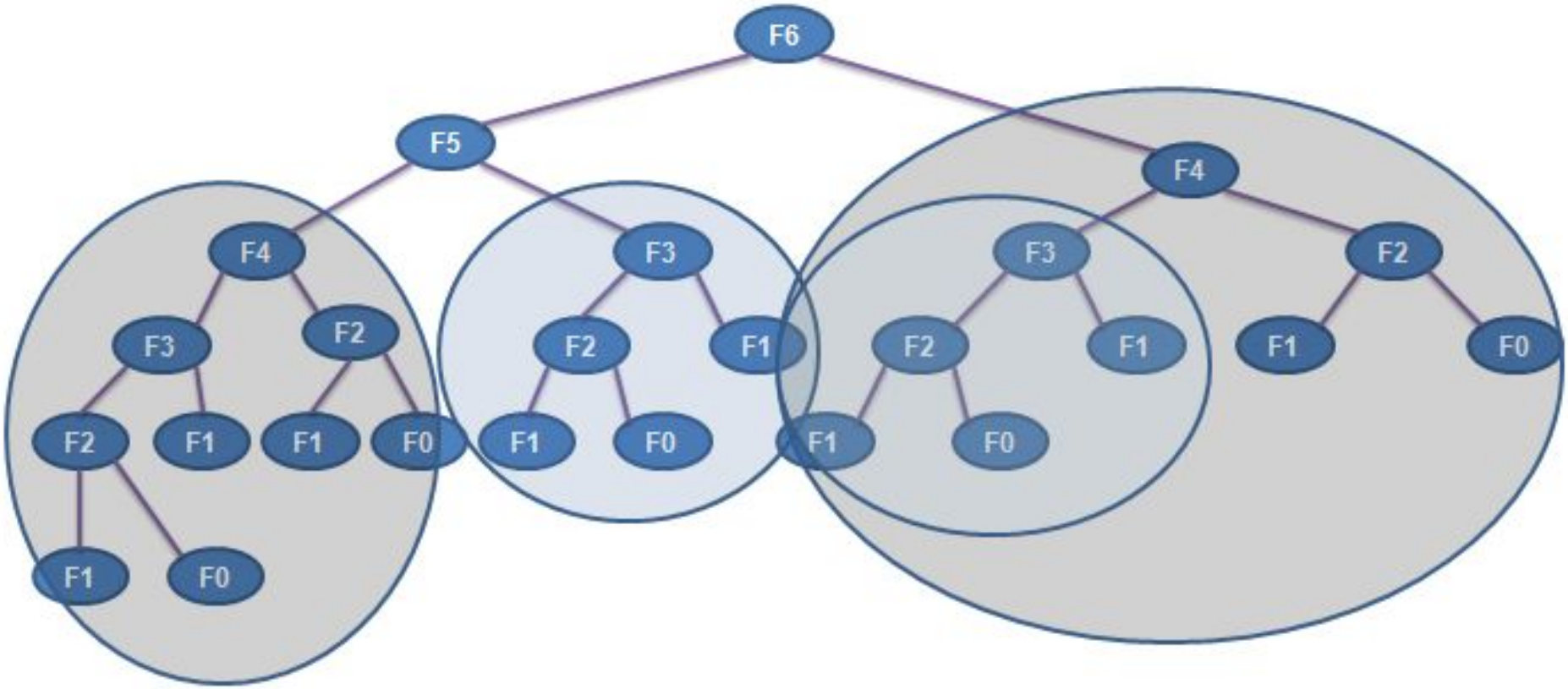
Em ambos os casos são utilizadas as tabelas para armazenamento das soluções, embora cada forma preencha a tabela de uma maneira diferente:

Top-Down: As entradas da tabela são preenchidas de acordo com a necessidade, ao longo da recursão. Pode não preencher toda a tabela;

Bottom-Up: As entradas da tabela são preenchidas "em ordem" e a tabela é totalmente preenchida.

PDTopDownFibonacci(n, t)
Entrada: Inteiro n, tabela t
se $n < 2$ **então**
  retorna $(t[n])$;
senão
  se $t[n] = \infty$ **então**
    $t[n] \leftarrow$ **PDTopDownFibonacci**$(n-1, t)$ +
      PDTopDownFibonacci$(n-2, t)$;
  senão
    retorna $(t[n])$;
  fim
fim

PDBottomUpFibonacci(n)
Entrada: Inteiro n
$t[0] \leftarrow 1;$
$t[1] \leftarrow 1;$
para $i \leftarrow 3$ ***até*** $i \leq n$ **faça**
$\quad t[i] \leftarrow t[i-1] + t[i-2];$
fim
retorna $(t[n]);$



Superposição de suproblemas ao determinarmos o sexto número da sequência de Fibonacci.

Top-Down vs. Bottom-Up

A PD *Top-Down* utiliza **memoização**, ou seja, registra valores para buscá-los posteriormente se necessário.

A PD *Bottom-Up* normalmente depende de uma noção de "tamanho" do problema e de uma noção de "ordem" dos subproblemas, tal que resolver um subproblema em particular depende de termos resolvidos os subproblemas menores anteriores.

As duas abordagens normalmente geram algoritmos com tempo de execução assintoticamente iguais, exceto em circunstâncias em que a PD *Top-Down* não examina todos os subproblemas recursivamente.

A PD *Bottom-Up* possui constantes melhores normalmente, devido ao menor *overhead* por chamadas recursivas de funções.

Exemplo de Projeto

Suponha que precisamos comprar roupas para um casamento, e somos oferecidos diferentes modelos de cada peça (e.g. 3 camisas, 2 cintos, 4 sapatos, etc.). Precisamos comprar uma peça de cada.

Como o orçamento é limitado, não podemos excedê-lo, mas podemos gastar o máximo possível.

É possível que não possamos comprar uma peça de cada, devido ao orçamento curto.

Exemplo de Projeto

A entrada é composta por $1 \leq M \leq 200$ e $1 \leq C \leq 20$, em que M denota o seu orçamento e C denota o número de peças a serem compradas.

Depois, para cada uma das C peças $[0 \ldots C-1]$ temos um inteiro $1 \leq K \leq 20$ indicando o número de modelos para aquela peça, seguido por K inteiros indicado o preço de cada modelo daquela peça.

A saída consiste de um inteiro indicando a quantia máxima de dinheiro necessária para comprar uma unidade de cada peça sem exceder o orçamento. Se não houver solução, devemos informar também.

O tempo limite para resolver qualquer instância é de 2 segundos.

Instância 1

$M = 20, C = 3$.

- 3 modelos da peça 0 $\rightarrow$ \$6 \$4 \$8
- 2 modelos da peça 1 $\rightarrow$ \$5 \$10
- 4 modelos da peça 2 $\rightarrow$ \$1 \$5 \$3 \$5

A resposta é \$19 (\$8+\$10+\$1 – itens denotados em vermelho).

Note que esta solução não é única, poderíamos ter (\$6+\$10+\$3) e (\$4+\$10+\$5).

Instância 2

$M = 9, C = 3$.

- 3 modelos da peça 0 $\rightarrow$ \$6 \$4 \$8
- 2 modelos da peça 1 $\rightarrow$ \$5 \$10
- 4 modelos da peça 2 $\rightarrow$ \$1 \$5 \$3 \$5

Não há solução para esta instância, pois se comprarmos todos os modelos mais baratos (\$4+\$5+\$1), ainda assim estouramos o orçamento.

Abordagem por *Backtracking*

Começamos com `orçamento_disponível = M` e `id_peça = 0`.

Tentamos todos os modelos para `id_peça = 0` (máximo de 20).

Se o modelo i for escolhido, subtraímos seu valor de `orçamento_disponível` e recursivamente processamos `id_peça = 1` da mesma maneira.

Quando processarmos `id_peça = C-1` terminamos.

Se, `orçamento_disponível < 0` antes de processarmos `id_peça = C-1`, podamos a solução parcial.

Dentre todas as soluções completas, selecionamos aquela cujo `orçamento_disponível` seja o mais próximo de zero, maximizando o valor gasto.

Abordagem por *Backtracking*

Podemos definir a abordagem de acordo com as recorrências:

- Se `orçamento_disponível < 0`, **backtracking**(`orçamento_disponível`, `id_peça`) $= -\infty$;
- Se um modelo para `id_peça` foi comprado, **backtracking**(`orçamento_disponível`, `id_peça`) = M-`orçamento_disponível`;
- No caso geral, para todos os modelos $[1 \ldots K]$, de `id_peça`, **backtracking**(`orçamento_disponível`, `id_peça`) = max(**backtracking**(`orçamento_disponível - preço[id_peça][modelo]`, `id_peça+1`)).

Abordagem por *Backtracking*

A abordagem por *Backtracking* possui corretude, mas é lenta.

No pior caso, cada peça terá 20 modelos diferentes, logo teríamos $20^{20} \approx 1,04 \times 10^{26}$ (104 septilhões).

Se Força Bruta e *Backtracking* fossem nossas únicas ferramentas, este problema não teria solução praticável.

Abordagem por Algoritmo Guloso

Uma vez que desejamos maximizar o dinheiro gasto, porquê não tentamos comprar o modelo mais caro de cada peça enquanto o orçamento permitir?

Com este critério guloso, para a primeira instância compraríamos o modelo 3 da peça 0 (custo \$8) e o modelo 2 da peça 1 (custo \$10), ambos os mais caros para cada. Para a peça 3, só teríamos dinheiro para o modelo 1 que custa \$1, totalizando \$19.

Esta estratégia gulosa funciona para as duas instâncias anteriores, dado que não há solução para a segunda instância.

Além disto, a estratégia é rápida: no pior caso, são realizadas 400 operações.

Instância 3

$M = 12, C = 3$.

- 3 modelos da peça 0 $\rightarrow$ \$6 \$4 \$8
- 2 modelos da peça 1 $\rightarrow$ \$5 \$10
- 4 modelos da peça 2 $\rightarrow$ \$1 \$5 \$3 \$5

Abordagem por Algoritmo Guloso

A abordagem por algoritmo guloso falha miseravelmente para a terceira instância.

Comprando o modelo 3 da peça 0 (custo \$8), não sobra dinheiro para comprarmos um modelo da peça 2. Neste caso reportaríamos não haver solução.

A solução ótima para esta instância é (\$4+\$5+\$3) = \$12, que utiliza todo o orçamento.

Abordagem por D&C

Este problema não pode ser resolvido por Divisão e Conquista.

Isto porque os subproblemas não são independentes entre si – a solução de um subproblema depende da diretamente solução dos anteriores.

Desta maneira, é impossível resolver os subproblemas separadamente e depois combinar suas soluções.

Note que a propriedade de subestrutura ótima não implica necessariamente em problemas que possam ser resolvidos de maneira independente.

Abordagem por PD

Para projetarmos uma abordagem por PD para este problema, precisamos verificar se ele possui as características necessárias:

Subestrutura Ótima: Demonstrada pela terceira recorrência do *Backtracking* anterior.

Sobreposição de Subproblemas: Na verdade, o espaço de busca de tamanho 20^{20} não é exatamente tão grande assim, pois há a repetição de subproblemas devido a simetrias, conforme descrito a seguir.

Abordagem por PD

Suponha que temos 2 modelos para uma determinada peça de roupa com o mesmo preço p:

- O *Backtracking* computaria o mesmo subproblema **backtracking**(`orçamento_disponível-p`, `id_peça+1`).
- A mesma situação ocorreria se alguma combinação de `orçamento_disponível` e preço de um modelo causasse $\texttt{orçamento\_disponível}_1 \texttt{ - p}_1 = \texttt{orçamento\_disponível}_2 \texttt{ - p}_2$.

Ambos subproblemas seriam computados mais do que uma vez, ineficientemente.

Abordagem por PD

Quantos subproblemas (ou estados) únicos existem então?
A resposta é $201 \times 20 = \mathbf{4.020}$!

A conta é simples: temos um orçamento de \$0 a \$200 e de 1 a 20 peças.

Cada subproblema só precisa ser resolvido uma única vez, e se pudermos garantir isto, podemos resolver o problema original muito mais rapidamente.

Muito menor do que 104 septilhões...

Abordagem PD

Para projetarmos a tabela, devemos considerar quais são os parâmetros que descrevem unicamente cada um dos estados: a quantidade de peças já compradas e o valor remanescente de dinheiro.

Se temos n parâmetros para representar os estados, preparamos uma tabela n-dimensional.

É importante caracterizar bem a equivalência entre subproblemas para que a tabela seja tão enxuta quanto possível, uma vez que suas dimensões influenciam diretamente a complexidade do algoritmo.

Abordagem PD *Top-Down*

Implementar uma abordagem PD *Top-Down* é relativamente fácil. Dada a relação de recorrência anterior, adicionamos poucos passos:

1. Inicializamos uma tabela com valores nulos, e.g. -1.
2. No início da função recursiva, checamos se o estado atual já foi computado anteriormente
   - Se sim, simplesmente retornamos o valor armazenado na tabela, em tempo $O(1)$.
   - Se não, resolvemos o subproblema, armazenamos o resultado na tabela e o retornamos.

Abordagem PD *Top-Down* – Análise

Se o problema possui $M \times C$ estados, então precisamos pelo menos de $O(MC)$ espaço.

Se o preenchimento de uma entrada da tabela requerer $O(k)$ passos, então a complexidade geral é limitada por $O(kMC)$.

De acordo com a descrição do problema, temos $M \times C = 4.020$ e $k = 20$, o que gera 80.400 operações, o que é seguramente praticável.

Abordagem PD *Top-Down* – Código

Ver código ch3_02_UVa11450_td.cpp

Abordagem PD *Bottom-Up*

Podemos também implementar uma abordagem PD *Bottom-Up* para o mesmo problema. Os passos são:

1. Identificar a recorrência da busca completa, como anteriormente;
2. Inicializar algumas entradas da tabela com valores conhecidos;
3. Determinar como preencher o resto da tabela, considerando a recorrência identificada e normalmente envolvendo um ou mais laços aninhados.

Abordagem PD *Bottom-Up*

O projeto da tabela segue o mesmo molde: se temos n parâmetros para representar os estados, preparamos uma tabela n-dimensional.

Preenchemos algumas entradas da tabela e determinamos quais serão as transições entre estados que nos permitirão preencher o resto das entradas.

Abordagem PD *Bottom-Up*

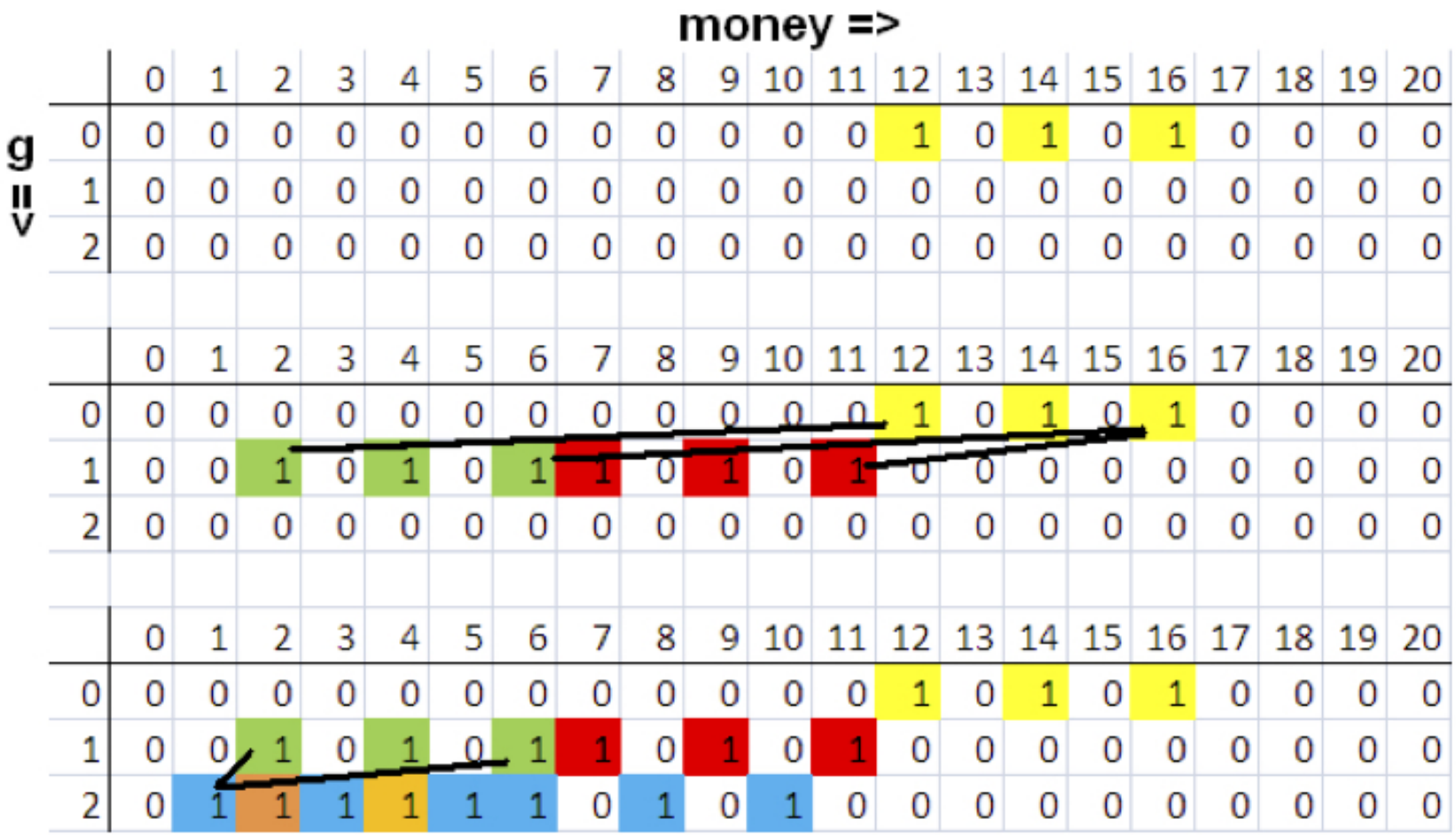
Primeiro, criamos uma tabela booleana 20×201 `alcançável[id_peça][orçamento_disponível]`.

Inicialmente, somente as entradas referentes a `id_peça=0` são marcadas como verdadeiras (na figura a seguir, linha 0 e colunas 14 (20-6), 16 (20-4) e 12 (20-8)) – o restante é falso.

Depois, iteramos a partir da segunda peça (linha 1 da tabela) até a última, preenchendo `alcançável[a][b]` se for possível alcançar este estado a partir do anterior.

Ou seja, `alcançável[a-1][dinheiro]` é verdadeiro e `alcançável[a][dinheiro-`p`]`, em que p é o preço de um dos modelos da peça a, não torna o dinheiro negativo.

Abordagem PD *Bottom-Up*



Exemplo de tabela na Abordagem PD *Bottom-Up* para a instância 1. As colunas de 21 a 200 não são apresentadas.

Abordagem PD *Bottom-Up*

Na figura anterior, à esquerda, `alcançável[1][16-5]` pode ser alcançado a partir de `alcançável[0][16]` pela compra de um modelo de custo \$5 para `id_peça=1`.

Da mesma forma, `alcançável[1][2]` pode ser alcançado a partir de `alcançável[0][12]` pela compra de um modelo de custo \$10 para `id_peça=1`, etc.

Abordagem PD *Bottom-Up*

Finalmente, a resposta pode ser obtida na última linha da tabela.

Basta determinar qual célula da referida linha possui um valor válido e está mais próxima da coluna 0.

Na figura a seguir, `alcançável[2][1]` é a resposta, indicando que podemos comprar uma combinação de peças de roupa e terminarmos com apenas \$1, ou em outras palavras, que gastamos \$19.

Se eventualmente não houver pelo menos uma célula com valor válido na última coluna, não há resposta para o problema.

Truque para economizar espaço

Não é necessário armazenar todas as linhas da tabela, apenas duas: o necessário é apenas a relação peça anterior $\times$ próxima peça.

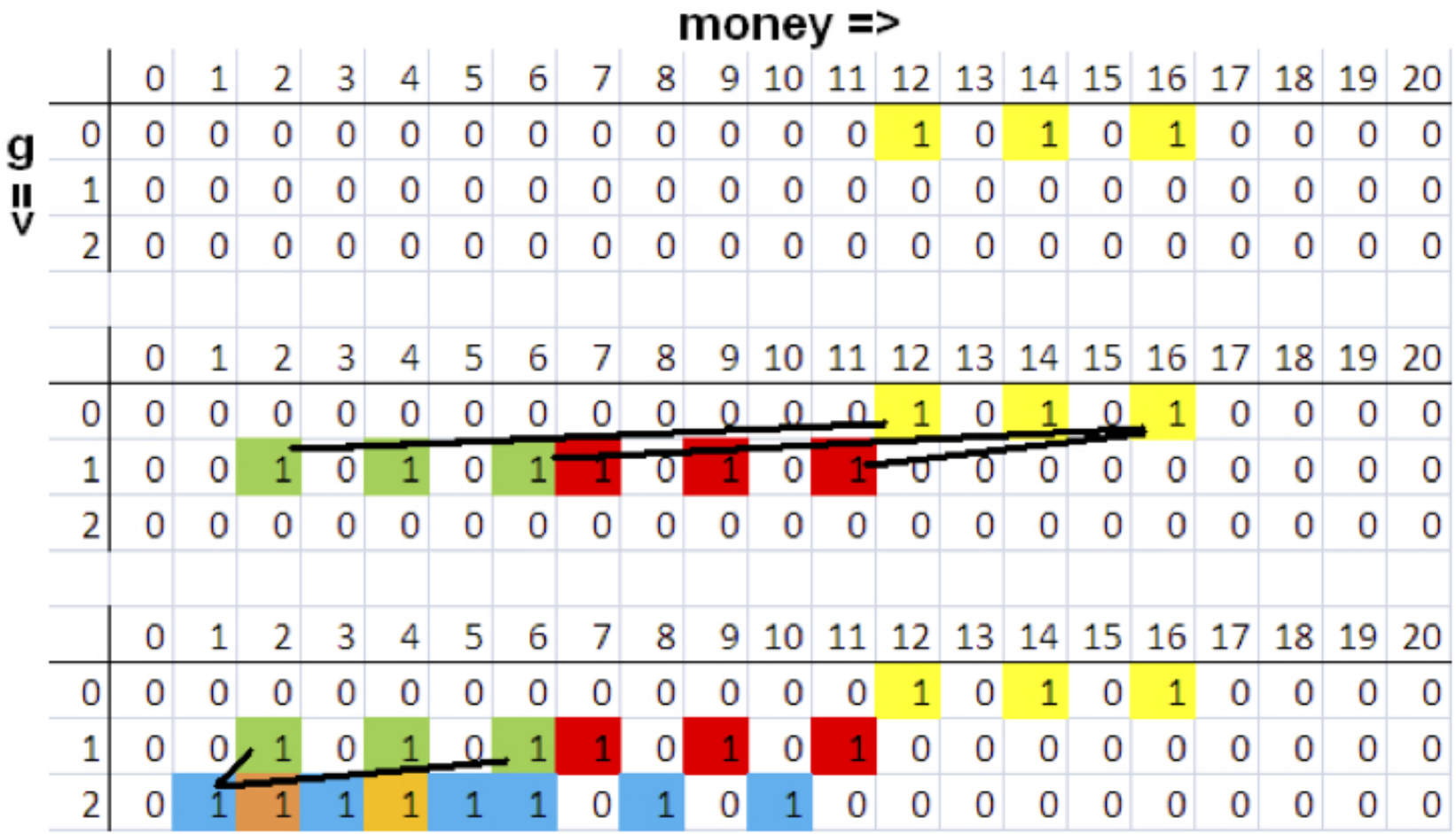
Abordagem PD *Bottom-Up*

Instância 1

$M = 20, C = 3$.

- 3 modelos da peça 0 $\rightarrow$ \$6 \$4 \$8
- 2 modelos da peça 1 $\rightarrow$ \$5 \$10
- 4 modelos da peça 2 $\rightarrow$ \$1 \$5 \$3 \$5

Abordagem PD *Bottom-Up*



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0
1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0
1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0
2	0	1	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0

Exemplo de tabela na Abordagem PD *Bottom-Up* para a instância 1. As colunas de 21 a 200 não são apresentadas.

Abordagem PD *Bottom-Up* – Código

Ver código ch3_03_UVa11450_bu.cpp

Programação Dinâmica

Vantagens

Economiza computação de soluções em problemas que possuem superposição de problemas e subestrutura ótima, gerando melhoria no desempenho.

Desvantagens

O número de soluções armazenadas na tabela pode crescer rapidamente caso o espaço de soluções não seja pequeno, exigindo assim muita memória.

Observação Final

A Programação Dinâmica usa memória adicional para economizar em tempo de computação, um exemplo claro de *trade-off*[a] entre tempo e memória.

[a] Implica um conflito de escolha e uma consequente relação de compromisso. Visa à resolução de problema mas acarreta outro.

Dúvidas?