

Programação Dinâmica I

SCC0210 – Algoritmos Avançados (2/2011)

Lucas Schmidt Cavalcante

- ▶ Introdução
- ▶ Soma máxima de uma subsequência contígua
- ▶ Problema do troco
- ▶ Quantidade de formas de dar troco
- ▶ Problema da mochila

Introdução

Técnica ampla e por isso de difícil definição, mas tem como característica **usar espaço para ganhar em velocidade!**

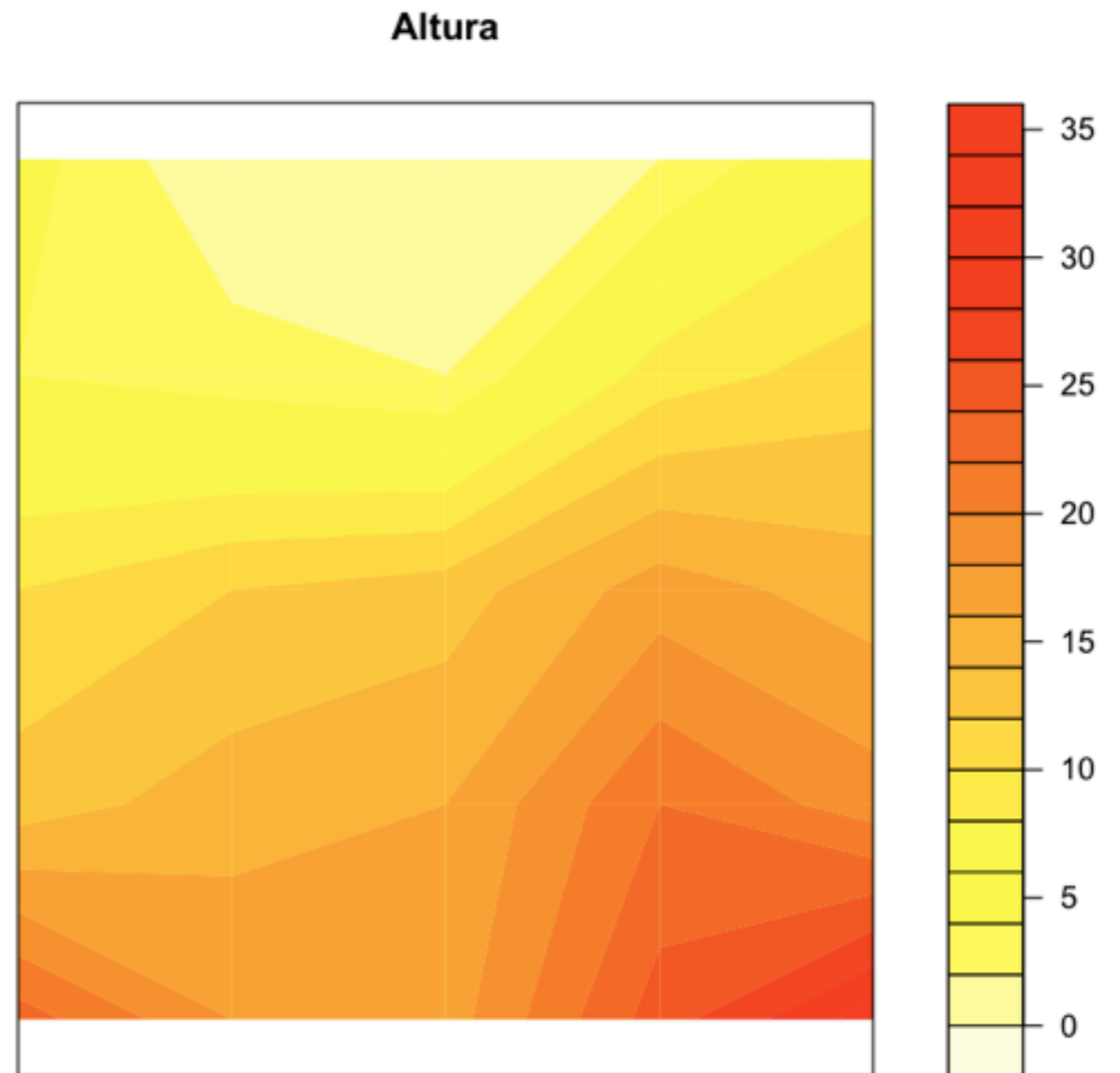
Um problema para ser resolvido com programação dinâmica (PD) requer apresentar as seguintes propriedades:

- ▶ **Sub-estrutura ótima:** a solução ótima é composta da solução ótima de instâncias menores (*Fibonacci*(n) é composto pela soma de *Fibonacci*($n-1$) e *Fibonacci*($n-2$));
- ▶ **Sobreposição de subproblemas:** ao calcular *Fibonacci*($n-1$) se calcula *Fibonacci*($n-2$), que é usado para calcular *Fibonacci*(n).

Solução com PD para fibonacci é memorizar num vetor os valores/estados (usar espaço) para não re-calcular (ganhar em velocidade).

Travessia: Problema

Uma formiga que só pode andar para o leste ou para o sul deseja atravessar uma região no sentido norte-sul de forma que a somatória das diferenças de altura entre regiões subsequentes de seu caminho seja a menor possível. Determine esse caminho.



Travessia: Exemplo

Dada a seguinte entrada, um possível caminho é exibido.

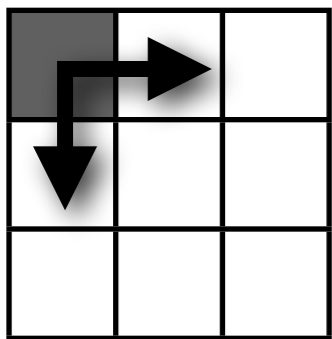
5	0	1	2	7	0
4	3	2	9	11	+ $ 0-3 + 3-2 $
10	12	13	17	15	+ $ 2-13 $
13	15	16	22	19	+ $ 13-16 $
23	18	17	25	31	+ $ 16-17 $
					<hr/> 19

Travessia: Análise

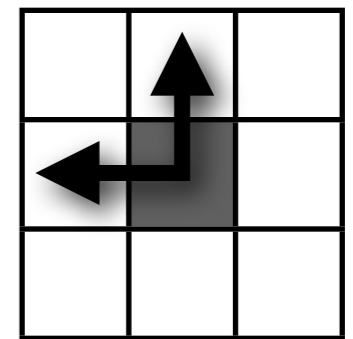
Vamos denotar por $PD[i][j]$ o menor custo para chegar à posição (i,j) .

Para a primeira linha ($PD[0][j]$, $0 \leq j \leq N$), que são os possíveis locais em que a formiga pode começar, não há nada o que fazer, são os casos base.

Para todas as outras posições, é preciso achar o melhor caminho:



Em PD se olha para o passado. Sendo assim, não se pergunte qual o melhor caminho para chegar no estado futuro, caminhe para o estado futuro e então se pergunte por qual caminho você chegou lá. Com isso, a possibilidade de andar sul-leste se torna norte-oeste.



$$PD[i][j] = \min(PD[i-1][j] + Diferenca(i,j,i-1,j), PD[i][j-1] + Diferenca(i,j,i,j-1)).$$

Travessia: Recursão

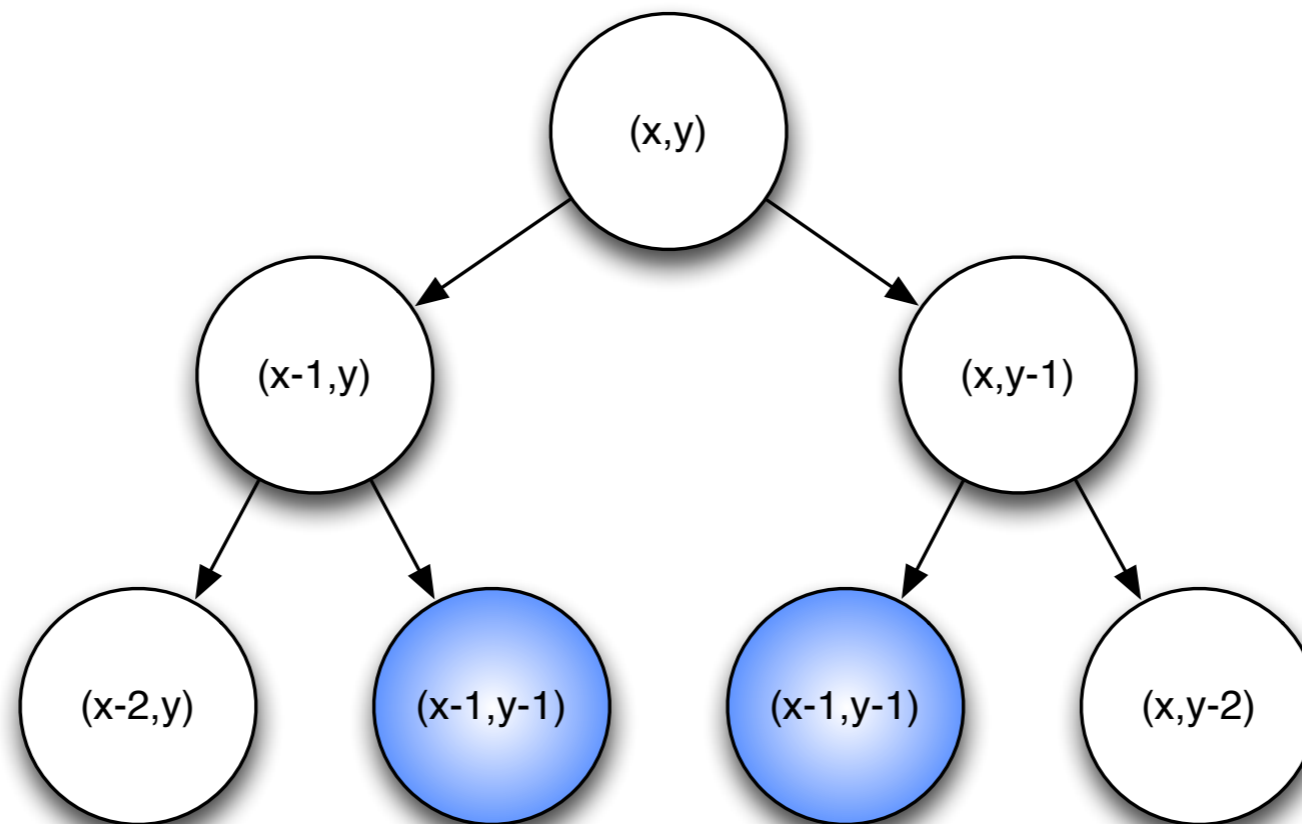
Solução por algoritmo recursivo, ignorando os casos de borda ($y=0$), e supondo que a matriz m guarda os valores da topologia:

```
int Resolve(int x, int y) {  
    if(x == 0)  
        return m[x][y];  
    else {  
        return min(Resolve(x-1, y) + abs(m[x][y]-m[x-1][y]),  
                  Resolve(x, y-1) + abs(m[x][y]-m[x][y-1]));  
    }  
}
```

Note que cada chamada recursiva tem que resolver $(x-1) \times (y-1)$ estados (a grosso modo).

Travessia: Recursão

Com o algoritmo proposto, podemos montar a seguinte árvore de recursão:



E assim nota-se que o algoritmo re-calcula estados.

Travessia: Recursão

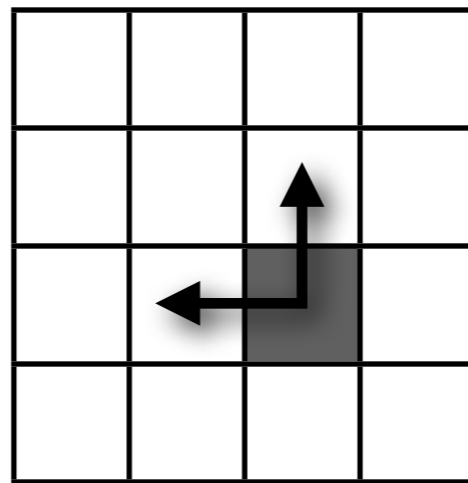
Como só há $x \times y$ estados, podemos modificar o código para memorizar estados já calculados (matriz auxiliar $pd[][]$):

```
int Resolve(int x, int y) {
    if(x == 0) return m[x][y];
    else {
        if(pd[x][y] == NAO_RESOLVIDO)
            pd[x][y] = min(Resolve(x-1, y) + abs(m[x][y]-m[x-1]
[y]), Resolve(x, y-1) + abs(m[x][y]-m[x][y-1]))
        return pd[x][y];
    }
}
```

Para preencher $pd[][]$ será preciso $O(n^2)$, depois as consultas se tornam $O(1)$.

Travessia: Versão Iterativa

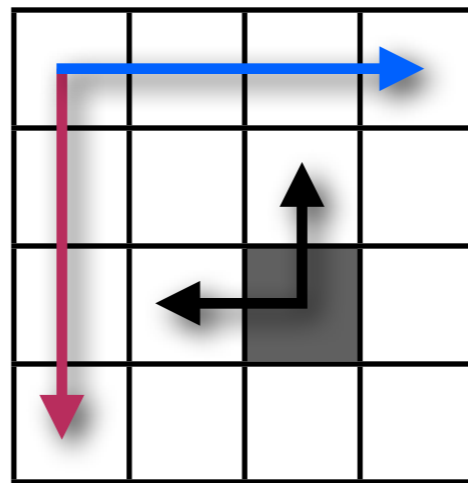
Em alguns problemas de PD é possível extrair da recursão uma versão iterativa. Observe a dependência dos estados:



Cada estado (i,j) depende do estado acima e do lado esquerdo, então basta determinar uma ordem de percorrer a matriz de modo que sempre que um estado é consultado ele já está calculado.

Travessia: Versão Iterativa

Em alguns problemas de PD é possível extrair da recursão uma versão iterativa. Observe a dependência dos estados:



Cada estado (i,j) depende do estado acima e do lado esquerdo, então basta determinar uma ordem de percorrer a matriz de modo que sempre que um estado é consultado ele já está calculado.

Primerio nesse **sentido** e depois no **outro**.

Travessia: Algumas iterações...

∞	5	0	1	2	7
∞	6				
∞					
∞					
∞					

← Casos base.

$$\min(5+|5-4|, \infty+|\infty-4|) = 6$$

Entrada. →

5	0	1	2	7
4	3	2	9	11
10	12	13	17	15
13	15	16	22	19
23	18	17	25	31

↑ Para não se preocupar com casos de borda, adicione uma coluna extra (o valor de infinito é para "afastar" o algoritmo de tentar usar um estado inexistente na solução).

∞	5	0	1	2	7
∞	6	3			
∞					
∞					
∞					

$$\min(0+|0-3|, 6+|4-3|) = 3$$

Travessia: Algumas iterações...

∞	5	0	1	2	7
∞	6	3	2	9	11
∞	12	12			
∞					
∞					

$$\min(|2+|10-12|, 3+|3-12|) = 12$$

5	0	1	2	7
4	3	2	9	11
10	12	13	17	15
13	15	16	22	19
23	18	17	25	31

∞	5	0	1	2	7
∞	6	3	2	9	11
∞	12	12	13		
∞					
∞					

$$\min(|2+|12-13|, 2+|2-13|) = 13$$

•	•	•	•	•
↑	↑	↑	←	←
↑	↑	←		

Como é necessário saber o caminho, use uma matriz auxiliar para marcar se aquele estado foi alcançado por vir da esquerda ou de cima.

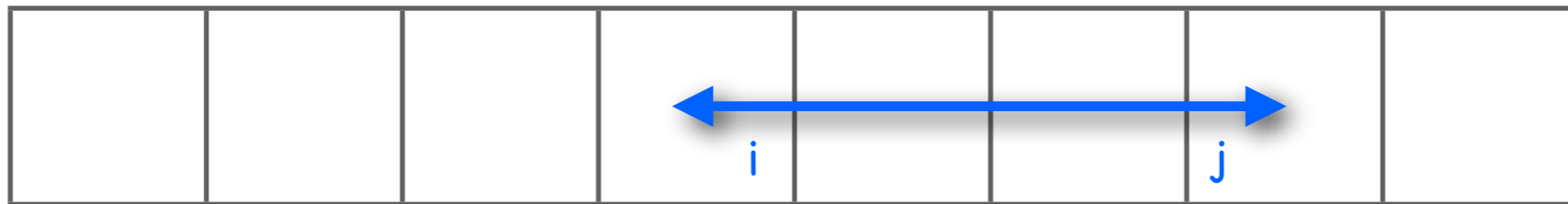
Estratégia Geral

A estratégia geral para resolver um problema de PD é:

- ▶ Determinar o estado de um problema pelo menor conjunto de seus parâmetros;
- ▶ Determinar os casos base;
- ▶ Construir uma recorrência que resolve todos os possíveis subproblemas.

Soma Máxima de uma Subsequência Contígua

Dado um vetor de números inteiros, determinar os índices i e j de tal forma que a soma de todos os valores entre i e j é máxima.



Obviamente que se não houver números negativos, a solução é a trivial, somar tudo.

5	0	1	-7	4	-1	-2	7
---	---	---	----	---	----	----	---

Soma Máxima de uma Subsequência Contígua

O algoritmo força bruta, de complexidade $O(n^2)$, começa uma soma em cada uma das posições.

1	5	2	0	3	1	4	-7	5	4	6	-1	7	-2	8	7
---	---	---	---	---	---	---	----	---	---	---	----	---	----	---	---

É interessante tentar começar uma soma na posição 2 ou 3? Não, porque pula o valor positivo 5.

Isso nos dá a primeira dica sobre um possível algoritmo: os locais onde se deve pensar em começar uma nova soma são os imediatamente a frente dos números negativos (além da posição 0, o início).

Soma Máxima de uma Subsequência Contígua

Vamos salvar a soma máxima em um vetor externo $pd[]$, mostrado logo abaixo do vetor de entrada ($v[]$).

1	5	2	0	3	1	4	-7	5	4	6	-1	7	-2	8	7
	5		5		6		?								

Soma Máxima de uma Subsequência Contígua

Vamos salvar a soma máxima em um vetor externo $pd[]$, mostrado logo abaixo do vetor de entrada ($v[]$).

1	5	2	0	3	1	4	-7	5	4	6	-1	7	-2	8	7
	5		5		6		-1		4		3		1		8

Uma forma de ler o vetor $pd[]$ é “a vantagem de começar a soma em $pd[i-1]$ e adicionar $v[i]$ ”. Quando a soma $pd[i-1] + v[i]$ dá negativo, então é uma desvantagem e o melhor é começar uma nova soma.

A cada iteração se tem duas escolhas: adicionar o elemento a soma atual ou começar uma nova, então: $pd[i] = \max(pd[i-1] + v[i], v[i])$.

Soma Máxima de uma Subsequência Contígua

Com este algoritmo em uma passada se determina a soma máxima, logo a complexidade é $O(n)$.

1	5	2	0	3	1	4	-7	5	4	6	-1	7	-2	8	7
	5		5		6		-1		4		3		1		8

Para os índices, ache o valor máximo em $pd[]$ e a partir dele comece a subtrair os elementos de $v[]$, quando chegar a zero é o começo.

Tome cuidado na implementação com o caso de um vetor composto por somente números negativos.

Subset-Sum

Dado um conjunto de números, quais são todos os possíveis valores que posso gerar ao somar um número com outro?

Se o conjunto for $\{2, 5, 9\}$, então os possíveis valores são $\{0, 2, 5, 7, 9, 11, 14, 16\}$. Neste conjunto estão a opção de não usar nenhum número (0), usar somente ele mesmo, somar 2 números quaisquer e somar todos.

Subset-Sum

Dado um conjunto, nós só estamos interessados em saber se um dado valor é possível ou não, então um vetor composto por 1s ou 0s é suficiente. Seja o conjunto $\{2,5,9\}$, inicializamos $pd[]$ da seguinte maneira:

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

↑ O tamanho do vetor será o valor da soma de todos os elementos, em alguns casos isso torna essa técnica proibitiva.

└ pd[] começa em zero para possibilitar a soma de nenhum dos elementos, mas também simplifica a PD.

Subset-Sum

Dado um conjunto, nós só estamos interessados em saber se um dado valor é possível ou não, então um vetor composto por 1s ou 0s é suficiente. Seja o conjunto $\{2,5,9\}$, inicializamos $pd[]$ da seguinte maneira:

1	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Sabemos que 2, 5 e 9 são possíveis. O que todos eles têm em comum?

Subset-Sum

Dado um conjunto, nós só estamos interessados em saber se um dado valor é possível ou não, então um vetor composto por 1s ou 0s é suficiente. Seja o conjunto $\{2,5,9\}$, inicializamos $pd[]$ da seguinte maneira:

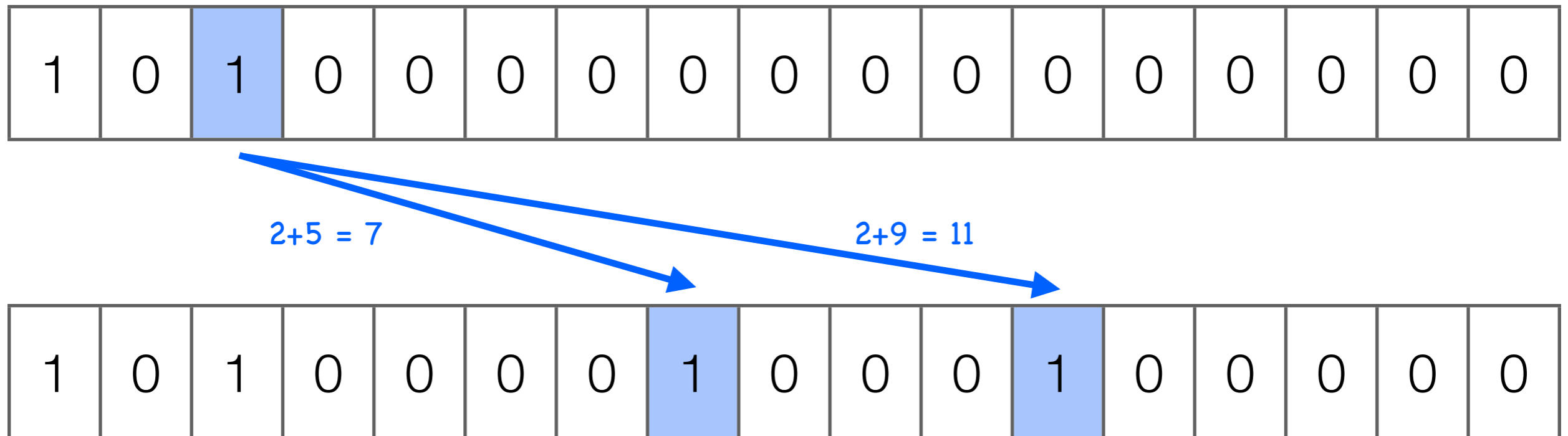
1	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Sabemos que 2, 5 e 9 são possíveis. O que todos eles têm em comum?

Interprete esses números como uma ponte, que levam de um estado alcançável para um novo.

Sendo assim, $pd[i][j] = pd[i-1][j-v[i]] ? i : 0$. No qual i itera sobre os elementos do conjunto e j de 0 até N (soma de todos os elementos). Uma matriz é usada para não repetir o mesmo elemento mais de uma vez, só temos uma instância de cada.

Subset-Sum



Interprete esses números como uma ponte, que levam de um estado alcançável para um novo.

Com a PD $pd[i][j] = pd[i-1][j-v[i]]$? $i : 0$ temos complexidade $O(NC)$ e uso de $O(NC)$ de espaço, no qual N é a soma de todos os elementos e C a quantidade de elementos. É possível usar somente $O(N)$ de espaço?

Subset-Sum

Era necessário usar uma matriz para não sujar o vetor com repetidos valores de uma mesma instância (no começo se faz $0+2$ e então $pd[2]=1$, ao chegar na posição 2 se faz $pd[2+2]=1$, e assim por diante...).

1	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Como evitar isso?

Subset-Sum

Era necessário usar uma matriz para não sujar o vetor com repetidos valores de uma mesma instância (no começo se faz $0+2$ e então $pd[2]=1$, ao chegar na posição 2 se faz $pd[2+2]=1$, e assim por diante...).

1	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Como evitar isso? Olhe para trás, no passado. Marque o vetor somente se $pd[j-v[i]]=1$, e isso implica em percorrer o vetor da posição $N-1$ até 0.

Mas se você tiver infinitas instâncias de um elemento, então ao usar somente um vetor e percorrer do começo para o fim você garante o uso de quantas instâncias forem possíveis usar.

Problema do Troco

Um sistema monetário possui moedas com os seguintes valores: 1, 4, 5 e 10. Deseja-se dar troco para c centavos com a menor quantidade de moedas possível.

Algoritmo guloso funciona para o caso de $c=7$ (5+1+1), mas falha para $c=8$ (guloso tentaria 5+1+1+1, mas a resposta certa é 4+4).

Problema do Troco

Um sistema monetário possui moedas com os seguintes valores: 1, 4, 5 e 10. Deseja-se dar troco para c centavos com a menor quantidade de moedas possível.

Algoritmo guloso funciona para o caso de $c=7$ (5+1+1), mas falha para $c=8$ (guloso tentaria 5+1+1+1, mas a resposta certa é 4+4).

Sabemos gerar todos os possíveis valores de troco, agora basta garantir que a solução encontrada seja mínima: $pd[j] = \min(pd[j], pd[j-v[i]] + 1)$.

Para afastar o algoritmo de um estado inválido, inicialize $pd[]$ com infinito, exceto a posição 0, que deve ter valor 0 (lembrando que i itera sobre os valores das moedas, ou seja, 1, 4, 5 e 10).

Quantidade de formas de dar troco

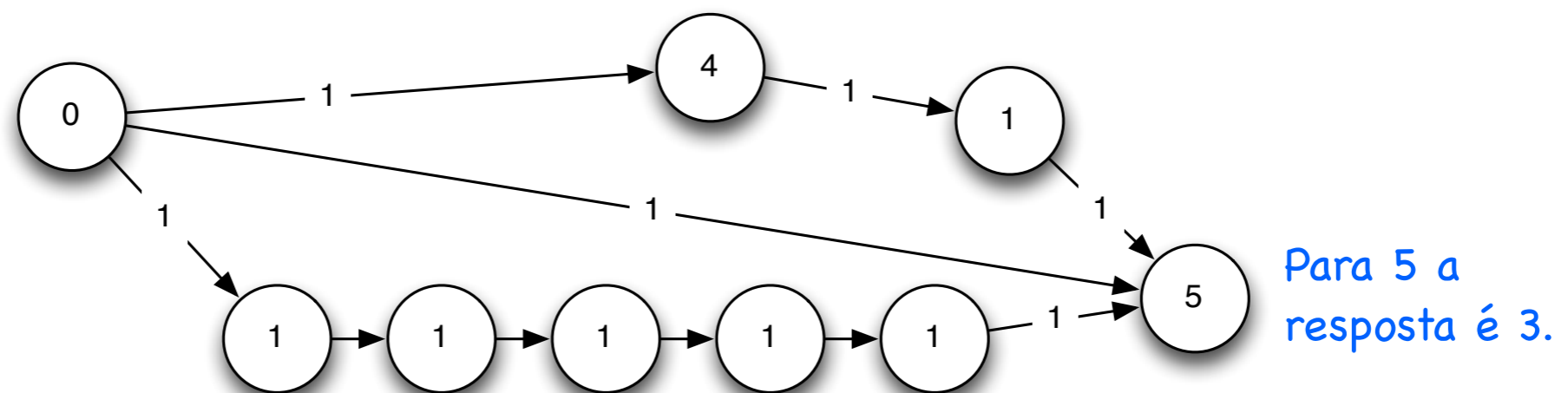
Um sistema monetário possui moedas com os seguintes valores: 1, 4, 5 e 10. Deseja-se dar troco para c centavos com a menor quantidade de moedas possível.

Algoritmo guloso funciona para o caso de $c=7$ (5+1+1), mas falha para $c=8$ (guloso tentaria 5+1+1+1, mas a resposta certa é 4+4).

Sabemos gerar todos os possíveis valores de troco, agora basta garantir que a solução encontrada seja mínima: $pd[j] = \min(pd[j], pd[j-v[i]] + 1)$.

Para afetar o algoritmo de um estado inválido, inicialize $pd[]$ com infinito, exceto a posição 0, que deve ter valor 0 (lembrando que i itera sobre os valores das moedas, ou seja, 1, 4, 5 e 10).

E se for pedido a quantidade de formas de dar troco, como fica a PD?



Problema da Mochila

Seja uma mochila com capacidade C e N itens, cada qual tem um peso ($p[i]$) e um valor ($v[i]$). Deseja-se maximizar o valor da mochila.

$p[i] =$	2	3	5	$C = 9$
$v[i] =$	1	2	6	

Problema da Mochila

Seja uma mochila com capacidade C e N itens, cada qual tem um peso ($p[i]$) e um valor ($v[i]$). Deseja-se maximizar o valor da mochila.

$p[i] =$	2	3	5	$C = 9$
$v[i] =$	1	2	6	

Já sabemos gerar todas as somas possíveis, mas precisamos ser inteligentes na escolha da mochila de peso 5, já que temos a possibilidade de escolher entre uma de valor **3** e outra de valor **6**. Como fica a PD?

Problema da Mochila

Seja uma mochila com capacidade C e N itens, cada qual tem um peso ($p[i]$) e um valor ($v[i]$). Deseja-se maximizar o valor da mochila.

$$p[] = \begin{array}{|c|c|c|} \hline 2 & 3 & 5 \\ \hline \end{array} \quad C = 9$$

$$v[] = \begin{array}{|c|c|c|} \hline 1 & 2 & 6 \\ \hline \end{array}$$

Apesar do interesse ser discutir a mochila de tamanho 5, a resposta é uma mochila de peso 8 com valor 8 (maior valor menor que a capacidade).

Já sabemos gerar todas as somas possíveis, mas precisamos ser inteligentes na escolha da mochila de peso 5, já que temos a possibilidade de escolher entre uma de valor **3** e outra de valor **6**. Como fica a PD?

$$pd[j] = \max(pd[j], pd[j-p[i]]+v[i]).$$

Referências

- “*Dynamic Programming*”, “*Overlapping subproblems*” & “*Optimal substructure*”. Wikipedia.
- “*Algoritmos I*”, Guilherme P. Telles. Instituto de Computação, UNICAMP.
- *Dynamic Programming Practice Problems*.
- “*Dynamic Programmimg*”, Cliff Stein. Columbia.
- “*Introduction to Algorithms: A Creative Approach*”, Udi Manber. Addison-Wesley.
- “*Programming Challenges*”, Steven S. Skiena. Springer.
- “*The Algorithm Design Manual*”, Steven S. Skiena. Springer.
- “*Competitive Programming: Increasing the Lower Bound of Programming Contests*”, Steven Halim & Felix Halim.