Programação Dinâmica 2

Bernardo Amorim

Universidade Federal de Minas Gerais

15 de outubro de 2024

PROBLEMA DA MOCHILA BEECROWD/1624

Dados $1 \leq N \leq 100$ itens de 1 a N, cada item tem um peso $1 \leq w_i \leq (30)10^5$ e um valor $1 \leq v_i \leq (1000)10^9$. Você tem uma mochila com capacidade W, qual é o maior valor que você consegue escolhendo alguns desses itens?

PROBLEMA DA MOCHILA BEECROWD/1624

Dados $1 \leq N \leq 100$ itens de 1 a N, cada item tem um peso $1 \leq w_i \leq (30)10^5$ e um valor $1 \leq v_i \leq (1000)10^9$. Você tem uma mochila com capacidade W, qual é o maior valor que você consegue escolhendo alguns desses itens?

- Exemplo: $N = 4$, $W = 10$

i	0	1	2	3
w_i	2	3	4	6
v_i	4	7	4	9

PROBLEMA DA MOCHILA BEECROWD/1624

Dados $1 \leq N \leq 100$ itens de 1 a N, cada item tem um peso $1 \leq w_i \leq (30)10^5$ e um valor $1 \leq v_i \leq (1000)10^9$. Você tem uma mochila com capacidade W, qual é o maior valor que você consegue escolhendo alguns desses itens?

- Exemplo: $N = 4$, $W = 10$

i	0	1	2	3
w_i	2	3	4	6
v_i	4	7	4	9

- O valor máximo atingível seria 16, caso pegássemos os itens 1 e 3, que somados tem peso 9.

REFUTANDO OS GULOSOS

- Se escolhêssemos os itens mais leves antes?

REFUTANDO OS GULOSOS

- Se escolhêssemos os itens mais leves antes?
  - $N = 3$, $W = 3$

i	0	1	2
w_i	1	2	3
v_i	1	1	3

REFUTANDO OS GULOSOS

- Se escolhêssemos os itens mais leves antes?
  - $N = 3$, $W = 3$

i	0	1	2
w_i	1	2	3
v_i	1	1	3

- O lucro máximo é 3 e o guloso responderia 2.

REFUTANDO OS GULOSOS

- Se escolhêssemos os itens mais valiosos antes?

REFUTANDO OS GULOSOS

- Se escolhêssemos os itens mais valiosos antes?
  - $N = 3$, $W = 4$

i	0	1	2
w_i	2	2	4
v_i	2	2	3

REFUTANDO OS GULOSOS

- Se escolhêssemos os itens mais valiosos antes?
  - $N = 3,\ W = 4$

i	0	1	2
w_i	2	2	4
v_i	2	2	3

- O lucro máximo é 4 e o guloso responderia 3.

REFUTANDO OS GULOSOS

- Se escolhêssemos os itens com maior custo benefício antes?

REFUTANDO OS GULOSOS

- Se escolhêssemos os itens com maior custo benefício antes?
  - $N = 3,\ W = 6$

i	0	1	2
w_i	3	3	4
v_i	2	2	3
$\frac{v_i}{w_i}$	$0.\overline{6}$	$0.\overline{6}$	0.75

REFUTANDO OS GULOSOS

- Se escolhêssemos os itens com maior custo benefício antes?
  - $N = 3,\ W = 6$

i	0	1	2
w_i	3	3	4
v_i	2	2	3
$\frac{v_i}{w_i}$	$0.\overline{6}$	$0.\overline{6}$	0.75

- O lucro máximo é 4 e o guloso responderia 3.

FORÇA BRUTA

- Uma alternativa para ter a resposta certa é tentar todas as possiblidades.

FORÇA BRUTA

- Uma alternativa para ter a resposta certa é tentar todas as possiblidades.
- Para isso tentaríamos, para cada item, pegá-lo ou não pegá-lo.

FORÇA BRUTA

- Uma alternativa para ter a resposta certa é tentar todas as possiblidades.
- Para isso tentaríamos, para cada item, pegá-lo ou não pegá-lo.

```
int brute(int cap, int item) {
    if(cap < 0) return -INF;
    if(item == n) return 0;

    return max(brute(cap-w[i], item + 1) + v[i], brute(cap, item + 1));
}
// ...
cout << brute(W, 0) << endl;
```

FORÇA BRUTA

- A força bruta certamente acha a resposta correta, uma vez que testa tudo.

FORÇA BRUTA

- A força bruta certamente acha a resposta correta, uma vez que testa tudo.
- Porém, a cada item dobramos o número de galhos da recursão.

FORÇA BRUTA

- A força bruta certamente acha a resposta correta, uma vez que testa tudo.
- Porém, a cada item dobramos o número de galhos da recursão.
- Logo nossa solução tem complexidade exponencial, $\mathcal{O}(2^N)$, o que é muito lento para nós!

FORÇA BRUTA

- A força bruta certamente acha a resposta correta, uma vez que testa tudo.
- Porém, a cada item dobramos o número de galhos da recursão.
- Logo nossa solução tem complexidade exponencial, $\mathcal{O}(2^N)$, o que é muito lento para nós!
- É fácil ver que é essa complexidade pensando que o número de subconjuntos de um conjunto com N elementos é 2^N, uma vez que é equivalente ao número de strings binárias de tamanho N.

FORÇA BRUTA

- A força bruta certamente acha a resposta correta, uma vez que testa tudo.
- Porém, a cada item dobramos o número de galhos da recursão.
- Logo nossa solução tem complexidade exponencial, $\mathcal{O}(2^N)$, o que é muito lento para nós!
- É fácil ver que é essa complexidade pensando que o número de subconjuntos de um conjunto com N elementos é 2^N, uma vez que é equivalente ao número de strings binárias de tamanho N.

Item	0	1	2	3	4	5	6	7	8	9
Pego?	0	0	1	1	0	1	0	1	1	0

RELAÇÃO DE RECORRÊNCIA

- Armazenar, para cada estado, quais itens já pegamos é exponencial (a complexidade depende do número de estados, queremos poucos estados!).

RELAÇÃO DE RECORRÊNCIA

- Armazenar, para cada estado, quais itens já pegamos é exponencial (a complexidade depende do número de estados, queremos poucos estados!).
- É útil pensar em PDs seguindo uma **ordem**. Procuramos $f(i)$ que retorna o maior lucro possível caso os itens considerados fossem apenas os itens em frente a i.

RELAÇÃO DE RECORRÊNCIA

- Precisamos saber quanto de espaço j temos sobrando. Logo podemos pensar em uma função $f(i, j)$ que retorna o maior lucro caso o problema se resumisse aos itens após i e tivéssemos j de espaço.

RELAÇÃO DE RECORRÊNCIA

- Precisamos saber quanto de espaço j temos sobrando. Logo podemos pensar em uma função $f(i, j)$ que retorna o maior lucro caso o problema se resumisse aos itens após i e tivéssemos j de espaço.
- Exemplo $f(1, 7) = 11$:

$$\begin{array}{c|cccc} i & \cancel{0} & 1 & 2 & 3 \\ w_i & \cancel{2} & 3 & 4 & 6 \\ v_i & \cancel{4} & 7 & 4 & 9 \end{array}$$

RELAÇÃO DE RECORRÊNCIA

- Precisamos construir uma relação de recorrência para f.

RELAÇÃO DE RECORRÊNCIA

- Precisamos construir uma relação de recorrência para f.

$$f(i,j) = \begin{cases} 0 & i = n \\ & \\ & \end{cases}$$

RELAÇÃO DE RECORRÊNCIA

- Precisamos construir uma relação de recorrência para f.

$$f(i,j) = \begin{cases} 0 & i = n \\ f(i+1,j) & i < n \text{ e } j < w_i \\ \end{cases}$$

RELAÇÃO DE RECORRÊNCIA

- Precisamos construir uma relação de recorrência para f.

$$f(i,j) = \begin{cases} 0 & i = n \\ f(i+1,j) & i < n \text{ e } j < w_i \\ \max\{f(i+1,j), f(i+1,j-w_i) + v_i\} & i < n \text{ e } j \geq w_i \end{cases}$$

COMPLEXIDADE

- A **transição** claramente custa $\mathcal{O}(1)$.

COMPLEXIDADE

- A **transição** claramente custa $\mathcal{O}(1)$.
- O número de estados é $N \times$ possível espaço na mochila.

COMPLEXIDADE

- A **transição** claramente custa $\mathcal{O}(1)$.
- O número de estados é $N \times$ possível espaço na mochila.
- Os valores dos itens são inteiros e o espaço inicial também, logo os espaços possíveis são inteiros.

COMPLEXIDADE

- A **transição** claramente custa $\mathcal{O}(1)$.
- O número de estados é $N \times$ possível espaço na mochila.
- Os valores dos itens são inteiros e o espaço inicial também, logo os espaços possíveis são inteiros.
- Logo o número de estados é $N \times W$, que pode ser no máximo 10^7.

COMPLEXIDADE

- A **transição** claramente custa $\mathcal{O}(1)$.
- O número de estados é $N \times$ possível espaço na mochila.
- Os valores dos itens são inteiros e o espaço inicial também, logo os espaços possíveis são inteiros.
- Logo o número de estados é $N \times W$, que pode ser no máximo 10^7.
- Complexidade:
  $\mathcal{O}(\text{número de estados} \times \text{custo por estado}) = \mathcal{O}(N.W.1) = \mathcal{O}(N.W)$.

SOLUÇÃO

```cpp
// ...
int N, preco[110], peso[110], memo[110][30010];

int dp(int i, int espaco) {
    if (espaco < 0) return -1e9;
    if (i == N) return 0;
    if (memo[i][espaco] != -1) return memo[i][espaco];

    return memo[i][espaco] = max(dp(i + 1, espaco),
                                 preco[i] + dp(i + 1, espaco - peso[i]));
}
```

SOLUÇÃO

```cpp
// ...
int main() {
    while (true) {
        cin >> N;
        if (N == 0) return 0;

        for (int i = 0; i < N; i++) cin >> preco[i] >> peso[i];

        int M; cin >> M;
        memset(memo, -1, sizeof memo);
        cout << dp(0, M) << endl;
```

LCS BEECROWD/2824

Dadas duas strings s e t tal que $1 \leq |s|, |t| \leq 5000$, ache e printe uma maior string r que é subsequência tanto de s quanto de t.

- Exemplo:

LCS BEECROWD/2824

Dadas duas strings s e t tal que $1 \leq |s|, |t| \leq 5000$, ache e printe uma maior string r que é subsequência tanto de s quanto de t.

- Exemplo:

$$\begin{array}{c|ccccc} s & a & x & y & b & \\ t & a & b & y & x & b \end{array}$$

LCS BEECROWD/2824

Dadas duas strings s e t tal que $1 \leq |s|, |t| \leq 5000$, ache e printe uma maior string r que é subsequência tanto de s quanto de t.

- Exemplo:

$$\begin{array}{c|ccccc} s & a & x & y & b & \\ t & a & b & y & x & b \end{array}$$

- Logo a resposta é $r = \text{axb}$.

LCS BEECROWD/2824

- Exemplo:

LCS BEECROWD/2824

- Exemplo:

s	a	b	r	a	c	a	d	a	b	r	a		
t	a	v	a	d	a	k	e	d	a	v	r	a	

LCS BEECROWD/2824

- Exemplo:

s	a	b	r	a	c	a	d	a	b	r	a				
t	a	v	a	d	a	k	e	d	a	v	r	a			

- Logo a resposta é $r =$ aaadara.

PENSANDO NA SOLUÇÃO

- Como é comum em programação dinâmica, usaremos sufixos na solução do nosso problema.

PENSANDO NA SOLUÇÃO

- Como é comum em programação dinâmica, usaremos sufixos na solução do nosso problema.
- Podemos pensar uma função $lcs\_size(i, j)$ que retorna o tamanho da LCS entre o sufixo de s que começa em i e o sufixo de t que começa em j, em particular, $lcs\_size(0, 0)$ é a LCS entre s e t.

PENSANDO NA SOLUÇÃO

- Como é comum em programação dinâmica, usaremos sufixos na solução do nosso problema.
- Podemos pensar uma função $lcs\_size(i, j)$ que retorna o tamanho da LCS entre o sufixo de s que começa em i e o sufixo de t que começa em j, em particular, $lcs\_size(0, 0)$ é a LCS entre s e t.
- Exemplo, $lcs\_size(7, 4) = 3$:

s	a	b	r	a	c	a	d	a	b	r	a	
t	a	v	a	d	a	k	e	d	a	v	r	a

PENSANDO NA SOLUÇÃO

- Se os primeiros caracteres forem iguais, eles claramente participam da melhor resposta:

PENSANDO NA SOLUÇÃO

- Se os primeiros caracteres forem iguais, eles claramente participam da melhor resposta:
- Se s[i] $=$ t[j], $lcs\_size(i, j) = 1 + lcs\_size(i + 1, j + 1)$:

PENSANDO NA SOLUÇÃO

- Se os primeiros caracteres forem iguais, eles claramente participam da melhor resposta:
- Se s[i] = t[j], $lcs\_size(i, j) = 1 + lcs\_size(i + 1, j + 1)$:

$$\begin{array}{c|cccccccccccc} s & a & b & r & a & c & a & d & a & b & r & a & \\ t & a & v & a & d & a & k & e & d & a & v & r & a \end{array}$$

RELAÇÃO DE RECORRÊNCIA

- Sabemos formar a solução caso ambos os primeiros caracteres do sufixo forem iguais, agora precisamos tratar o caso de quando eles forem diferentes.

RELAÇÃO DE RECORRÊNCIA

- Sabemos formar a solução caso ambos os primeiros caracteres do sufixo forem iguais, agora precisamos tratar o caso de quando eles forem diferentes.
- Certamente s[i] ou t[j] não participarão da nossa melhor resposta, dado que são diferentes.

RELAÇÃO DE RECORRÊNCIA

- Sabemos formar a solução caso ambos os primeiros caracteres do sufixo forem iguais, agora precisamos tratar o caso de quando eles forem diferentes.
- Certamente s[i] ou t[j] não participarão da nossa melhor resposta, dado que são diferentes.
- Porém, não sabemos ainda de qual deles abriremos mão, e é aí que entra a programação dinâmica, testaremos todas as possibilidades.

RELAÇÃO DE RECORRÊNCIA

- Sabemos formar a solução caso ambos os primeiros caracteres do sufixo forem iguais, agora precisamos tratar o caso de quando eles forem diferentes.
- Certamente s[i] ou t[j] não participarão da nossa melhor resposta, dado que são diferentes.
- Porém, não sabemos ainda de qual deles abriremos mão, e é aí que entra a programação dinâmica, testaremos todas as possibilidades.

RELAÇÃO DE RECORRÊNCIA

$$lcs\_size(i, j) = \begin{cases} 0 & i \geq |s| \text{ ou } j \geq |t| \\ & \\ & \end{cases}$$

RELAÇÃO DE RECORRÊNCIA

$$lcs\_size(i,j) = \begin{cases} 0 & i \geq |s| \text{ ou } j \geq |t| \\ 1 + lcs\_size(i+1, j+1) & s[i] = t[j] \\ \end{cases}$$

RELAÇÃO DE RECORRÊNCIA

$$lcs\_size(i,j) = \begin{cases} 0 & i \geq |s| \text{ ou } j \geq |t| \\ 1 + lcs\_size(i+1, j+1) & s[i] = t[j] \\ max(\{lcs\_size(i+1, j),\ lcs\_size(i, j+1)\}) & s[i] \neq t[j] \end{cases}$$

RELAÇÃO DE RECORRÊNCIA

$$lcs\_size(i,j) = \begin{cases} 0 & i \geq |s| \text{ ou } j \geq |t| \\ 1 + lcs\_size(i+1, j+1) & s[i] = t[j] \\ max(\{lcs\_size(i+1, j),\ lcs\_size(i, j+1)\}) & s[i] \neq t[j] \end{cases}$$

- O nosso número de estados é $|s| \cdot |t|$ e a transição é feita em $\mathcal{O}(1)$, logo a complexidade dessa solução é $\mathcal{O}(|s| \cdot |t|)$.

SOLUÇÃO LCS BEECROWD/2824

```
string s, t; cin >> s >> t;
vector lcs_size(s.size() + 1, vector<int>(t.size() + 1));

for(int i = s.size() - 1; i >= 0; i--) {
    for(int j = t.size() - 1; j >= 0; j--) {
        if(s[i] == t[j]) lcs_size[i][j] = 1 + lcs_size[i+1][j+1];
        else lcs_size[i][j] = max(lcs_size[i+1][j], lcs_size[i][j+1]);
    }
}
cout << lcs_size[0][0] << endl;
```

RECUPERANDO A RESPOSTA

- Durante a programação dinâmica, testávamos as escolhas possíveis (desiste de um ou outro elemento).

RECUPERANDO A RESPOSTA

- Durante a programação dinâmica, testávamos as escolhas possíveis (desiste de um ou outro elemento).
- Para reconstruir, precisaremos saber qual das escolhas foi a melhor, para isso já precisamos ter calculado as dependências da melhor resposta.

RECUPERANDO A RESPOSTA

- Durante a programação dinâmica, testávamos as escolhas possíveis (desiste de um ou outro elemento).
- Para reconstruir, precisaremos saber qual das escolhas foi a melhor, para isso já precisamos ter calculado as dependências da melhor resposta.
- Dado que a nossa escolha depende do máximo entre duas funções, simplesmente olharemos qual é a maior e "andaremos para esta".

RECUPERANDO A RESPOSTA

- Durante a programação dinâmica, testávamos as escolhas possíveis (desiste de um ou outro elemento).
- Para reconstruir, precisaremos saber qual das escolhas foi a melhor, para isso já precisamos ter calculado as dependências da melhor resposta.
- Dado que a nossa escolha depende do máximo entre duas funções, simplesmente olharemos qual é a maior e "andaremos para esta".
- Dado que $lcs(i, j)$ retorna a LCS entre o sufixo de s que começa em i e o sufixo de t que começa em j:

RECUPERANDO A RESPOSTA

$$lcs(i, j) = \begin{cases} "\ " & i \geq |s| \text{ ou } j \geq |t| \\ & \\ & \end{cases}$$

RECUPERANDO A RESPOSTA

$$lcs(i,j) = \begin{cases} "" & i \geq |s| \text{ ou } j \geq |t| \\ "s[i]" + lcs(i+1, j+1) & s[i] = t[j] \\ & \end{cases}$$

RECUPERANDO A RESPOSTA

$$lcs(i,j) = \begin{cases} \text{""} & i \geq |s| \text{ ou } j \geq |t| \\ \text{"}s[i]\text{"} + lcs(i+1, j+1) & s[i] = t[j] \\ lcs(i+1, j) & lcs\_size(i+1, j) \geq lcs\_size(i, j+1) \end{cases}$$

RECUPERANDO A RESPOSTA

$$
lcs(i,j) = \begin{cases} "\,"& i \geq |s| \text{ ou } j \geq |t| \\ "s[i]" + lcs(i+1, j+1) & s[i] = t[j] \\ lcs(i+1, j) & lcs\_size(i+1, j) \geq lcs\_size(i, j+1) \\ lcs(i, j+1) & lcs\_size(i+1, j) < lcs\_size(i, j+1) \end{cases}
$$

SOLUÇÃO

```
string lcs;
int i = 0, j = 0;
while(i < s.size() and j < t.size()) {
    if (s[i] == t[j]) {
        lcs.push_back(s[i]);
        i++, j++;
    } else if (lcs_size[i+1][j] >= lcs_size[i][j+1]) i++;
    else j++;
}
cout << lcs << endl;
```