Programação Dinâmica

Bruno Papa
Maurício Scarelli Arantes
Rodrigo Rossetti

Introdução

"Quem não se lembra do passado é condenado a repeti-lo".

A solução de alguns problemas dependem da execução de subproblemas ou estados semelhantes.

Recálculo é custoso.

O armazenamento dos resultados pode resolver o nosso problema.

Top Down vs Bottom Up

top-down recursion: `memoization`

Do geral para o específico, de cima para baixo.

Visita apenas os estados requisitados.

bottom-up iteration: `tabulation`

Do específico para o geral, de baixo para cima.

Visita todos os estados, sempre.

Exemplo - Fibonacci

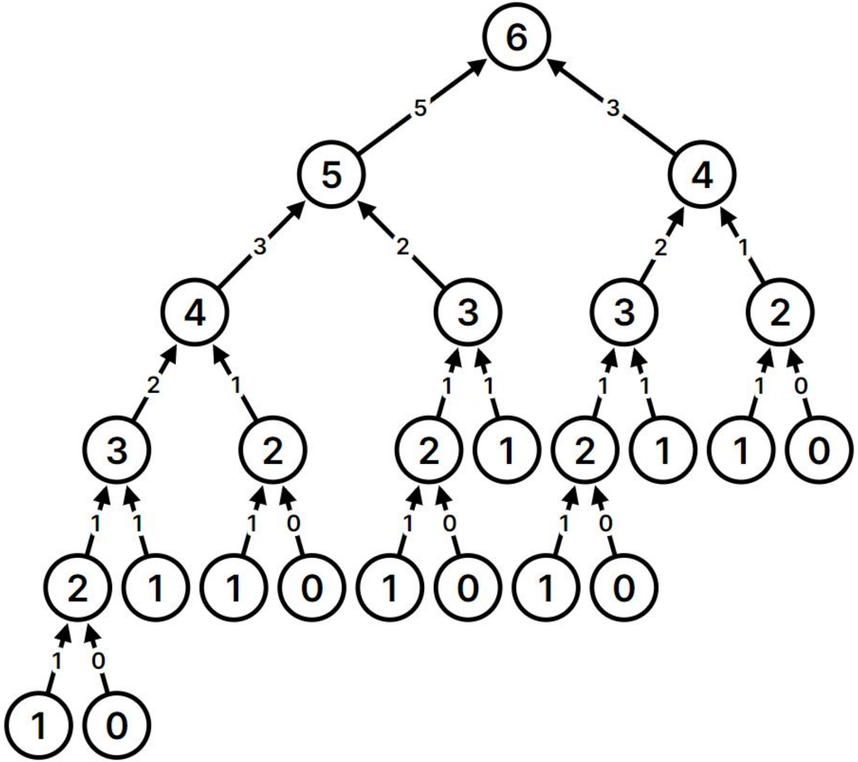
Motivação: Calcular o i-ésimo termo da sequência de Fibbonacci.

Podemos pensar como uma função recursiva simples:

```
int fib(int i) {
    if (i == 0 || i == 1)
        return 1;
    return fib(i-1) + fib(i-2);
}
```

Exemplo - Fibonacci

fib(6)



Exemplo - Fibonacci

Em algoritmos que utilizam a estratégia de divisão e conquista a repetição de subproblemas é algo comum (sobreposição / *overlapping*).

Para evitar o recálculo, a programação dinâmica é uma ótima solução.

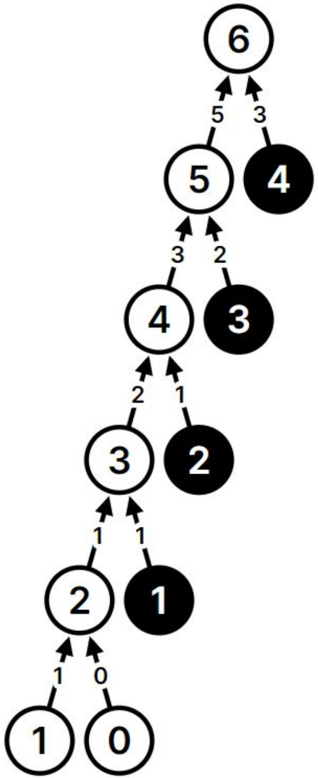
Fibonacci - Top Down

```
int memo[] = {1, 1, -1, -1, -1, -1, ...} //-1 = não calculado

int fib(int i) {
    if (memo[i] != -1)
        return memo[i];
    return memo[i] = fib(i-1) + fib(i-2);
}
```

Fibonacci - Top Down

fib(6)



Fibonacci - Bottom Up

```
int memo[] = {1, 1, -1, -1, -1, -1, …} //-1 = não calculado

int fib(int i) {
    for(int j = 2; j <= i; j++)
        memo[j] = memo[j-1] + memo[j-2];
    return memo[i];
}
```

Introdução - Guloso vs PD

	Guloso	PD
Método	Em um algoritmo guloso fazemos a melhor escolha local a cada passo do algoritmo na expectativa que ele leve à solução global ótima do problema.	Em um algoritmo de PD fazemos a escolha a cada passo considerando o problema atual e as soluções de subproblemas calculados anteriormente.
Solução Ótima	Para a maioria dos problemas não há a garantia de que levará a uma solução ótima.	É garantido que irá levar a uma solução ótima.

Problema do Troco

Motivação: Devolver o troco utilizando um número mínimo de moedas.

Matematicamente:

minimizar $$f(W) = \sum_{j=1}^{n} x_j$$

sujeito a $$\sum_{j=1}^{n} w_j x_j = W$$

Dado uma quantidade **W** positiva de troco e um conjunto **{w1, w2, ..., wn}** de denominações de moedas, determinar um conjunto de inteiros não negativos **{x1, x2, ..., xn}**, onde cada **xj** representa quantas moedas de denominação **wj** devolver a fim de minimizar a quantidade de moedas devolvidas **f(W)**, de forma que a soma dos valores das moedas devolvidas seja igual a W.

Problema do Troco

Solução gulosa: Percorrer o vetor das moedas decrescentemente e ir adicionando a moeda atual enquanto o troco restante for maior que ela. Quando o troco atingir zero, terminar o algoritmo.

Complexidade de tempo: O(NlogN) se as moedas não estiverem ordenadas, senão O(N), em que N = quantidade de denominações de moedas.

Problema do Troco

Exemplo: moedas = {0.5, 1, 2, 10, 20, 50, 100, 200} e troco = 175

1ª iteração: 200 > 175, não adiciona moedas de 200. Troco restante = 175

2ª iteração: 100 <= 175, adiciona uma moeda de 100. Troco restante = 75

3ª iteração: 50 <= 75, adiciona uma moeda de 50. Troco restante = 25

4ª iteração: 20 <= 25, adiciona uma moeda de 25. Troco restante = 5

6ª iteração: 2 <= 5, adiciona duas moedas de 2. Troco restante = 1

7ª iteração: 1 <= 1, adiciona uma moeda de 1. Troco restante = 0, termina o algoritmo

Resposta: r = {0, 1, 2, 0, 1, 1, 1, 0}, total = 6

Problema do Troco

O algoritmo guloso não funcionará sempre.

Exemplo: para moedas = {1, 15, 25} e troco = 30, resultará em r = {5, 0, 1} e total = 6, quando a solução ótima é 2 moedas de 15.

Problema do Troco

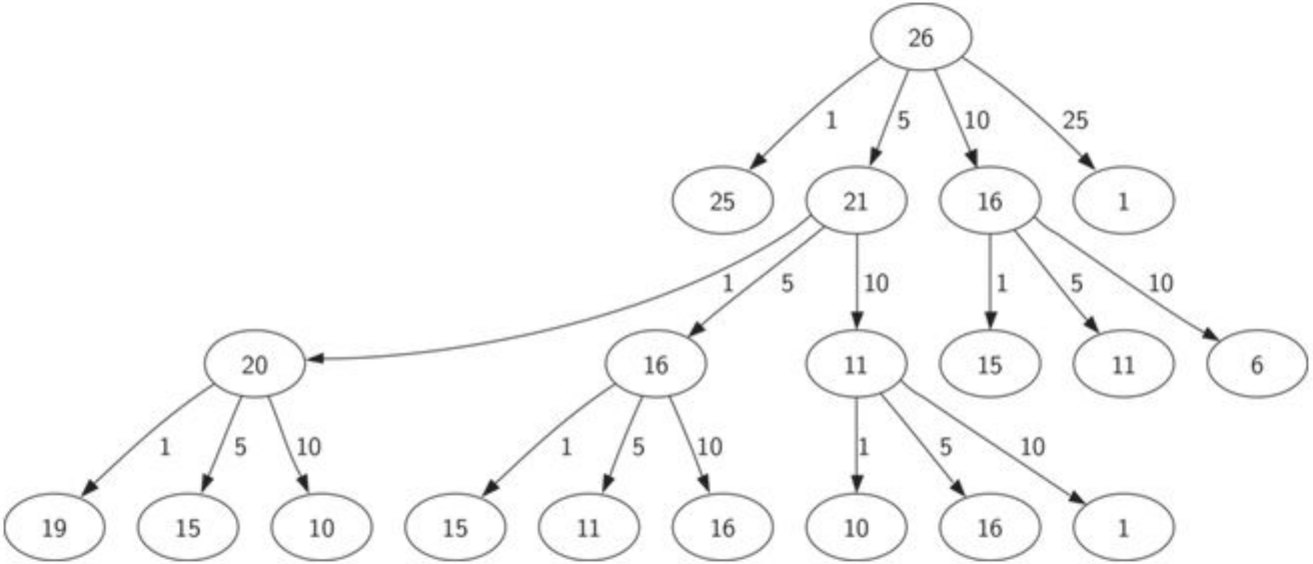
Subestrutura ótima:

Se W = 0, resposta = 0 (caso base)

senão: **f(W) = min{1 + f(W-moedas[i])}**, com i variando de 0 a n-1.

Problema do Troco

Para moedas = {1, 5, 10, 25} e troco = 26 temos a seguinte árvore de recursão:

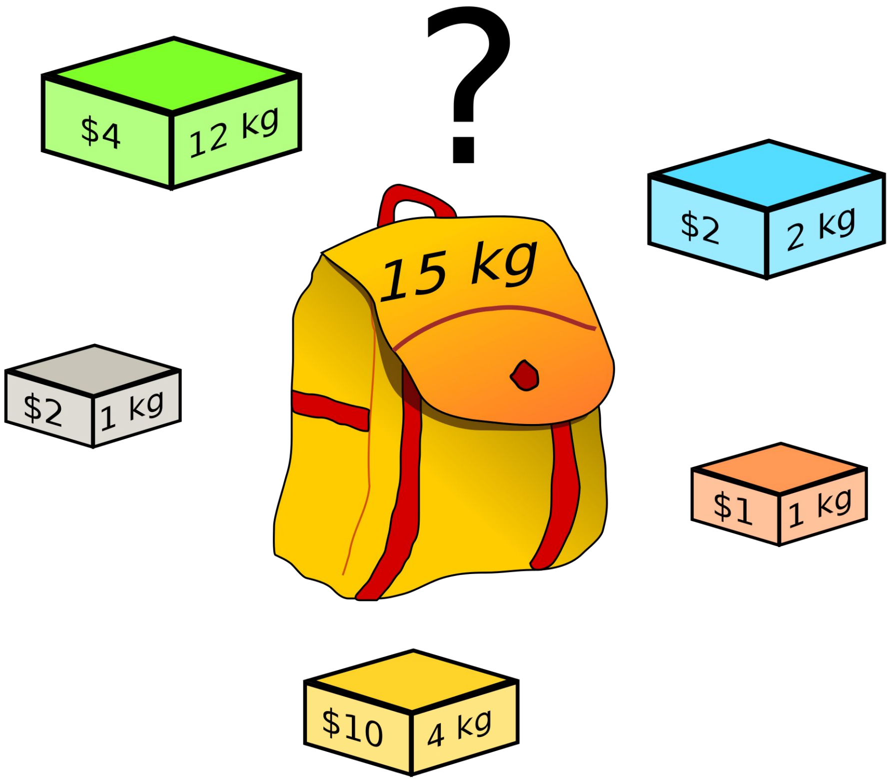


Problema do Troco

```cpp
// Complexidade de tempo: O(N*W)
int minCoins(vector<int>& moedas, int w) {
    int n = moedas.size();
    vector<int> dp(w+1, INT_MAX);
    dp[0] = 0;
    for (int i = 1; i <= w; i++)
        for (int j = 0; j < n; j++)
            if (moedas[j] <= i)
                dp[i] = min(dp[i], dp[i-moedas[j]]+1);
    return dp[w];
}
```

0-1 Knapsack

Motivação: dado os valores e pesos de N itens, determine o valor total máximo que a mochila pode carregar selecionando um subconjunto desses itens, de modo que a soma de seus pesos não exceda a sua capacidade máxima S.



0-1 Knapsack

Relação entre subproblemas:

i-ésimo item

mochila com capacidade disponível s

$$dp(i, s) = \begin{cases} 0 & , \ se\ i = N\ ou\ s = 0 \\ max(\underbrace{dp(i+1, s)}_{\text{ignora o item i}},\ \underbrace{v[i] + dp(i+1, s - w[i])}_{\text{pega o item i}}) & \end{cases}$$

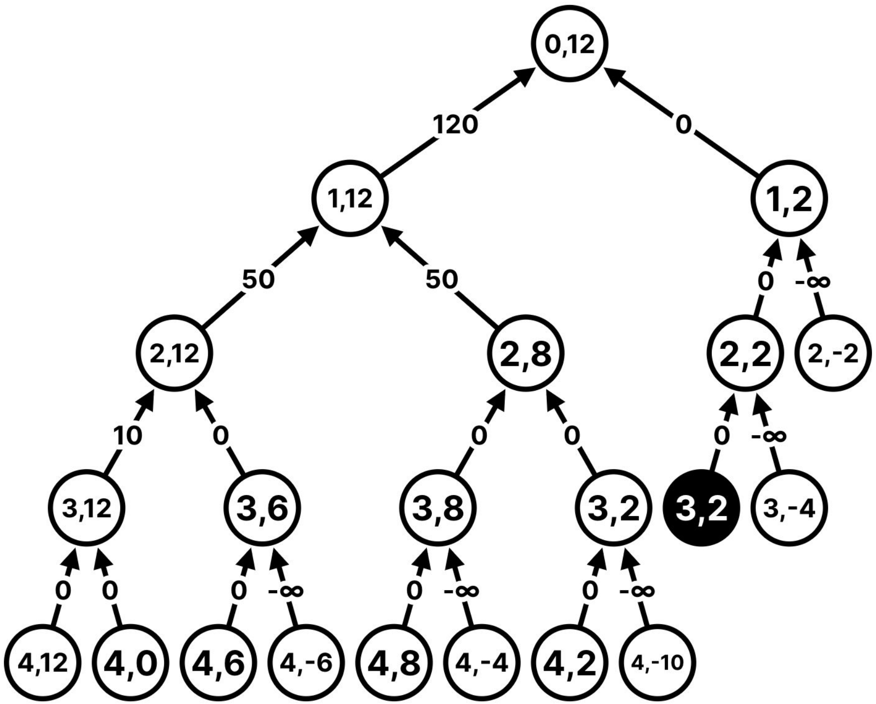
0-1 Knapsack

```
/* O(N*S) */
int dp(int i, int s) {
   if (s < 0)  return -INF;
   if (i == N || s == 0) return 0;

   int &ans = memo[i][s];
   if (ans != -1) return ans;

   return ans = max(
      dp(i+1, s),
      v[i] + dp(i+1, s-w[i])
   );
}
```

```
v = {100, 70, 50, 10};
w = {10, 4, 6, 2};
S = 12;
```

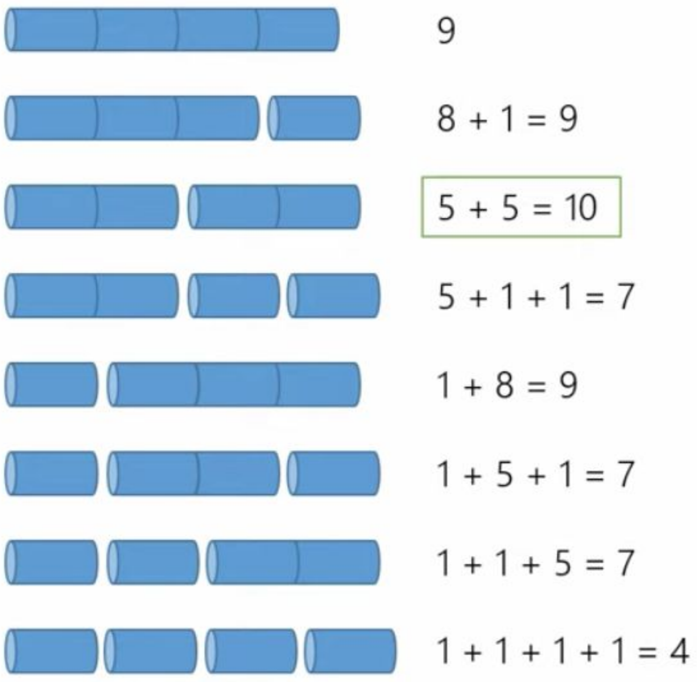


Problema do corte do bastão (rod cutting)

Motivação: dado um bastão de comprimento N e os preços de todos os pedaços de comprimento menor que N, determine o maior valor que você pode conseguir ao vender pedaços menores do bastão.

Exemplo:

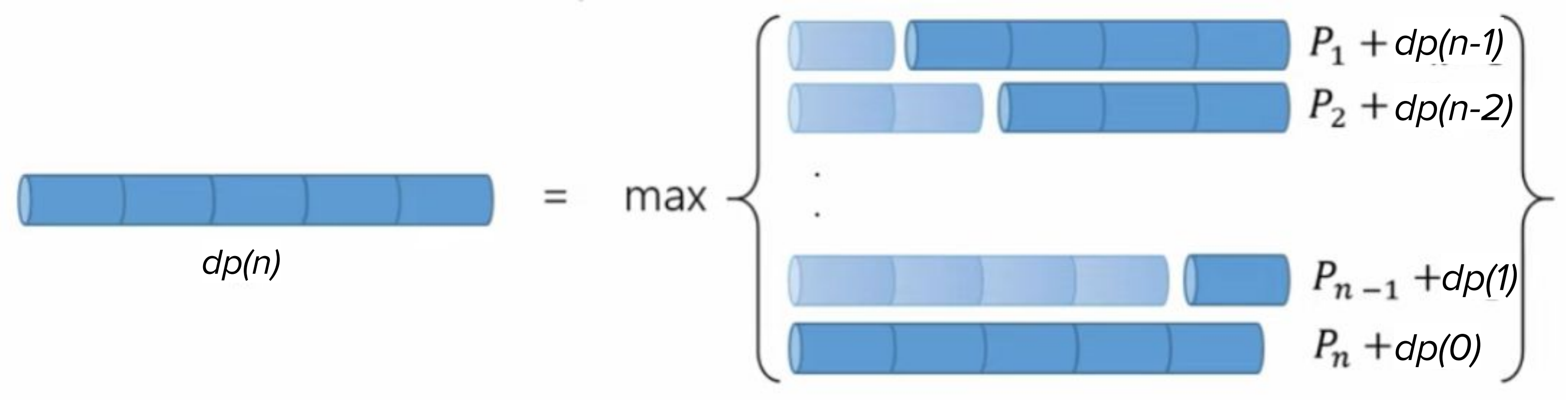
Length	1	2	3	4
Price	1	5	8	9



Problema do corte do bastão (rod cutting)

Relação entre subproblemas:

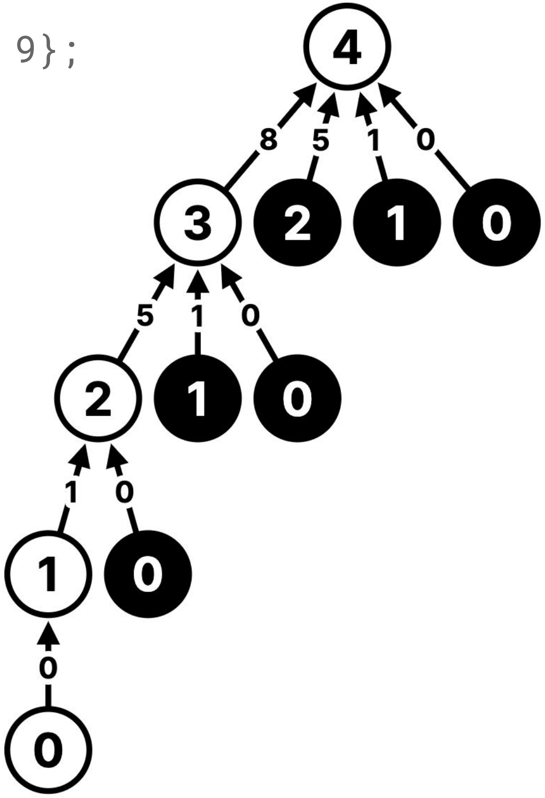
n: tamanho do bastão



Problema do corte do bastão (rod cutting)

```
int N = 4;
vector<int> prices = {1, 5, 8, 9};
```

```
/* O(N^2) */
int dp(int n) {
   if (n == 0) return 0;
   int &ans = memo[n];
   if (ans != -1) return ans;
   ans = -INF;
   for (int i = 0; i < n; i++)
       ans = max(ans, prices[i] + dp(n-(i+1)));
   return ans;
}
```



Longest Common Subsequence (LCS)

- Dado duas sequências, encontrar o comprimento da subsequência de maior comprimento comum a ambas.
- Uma subsequência é uma sequência que pode ser derivada de outra sequência a partir da remoção de zero ou mais elementos sem mudar a ordem dos elementos restantes.

Exemplo: s = “abcdefg”

“abc”, “abf”, “abg”, “bcd”, “deg” e “abcdefg” são subsequências válidas de s.

Longest Common Subsequence (LCS)

- **Subestrutura Ótima:** Dado as sequências a[0...n-1] e b[0...m-1] de entrada com comprimentos n e m respectivamente, e sendo dp(a[0...n-1], b[0...m-1]) o comprimento da LCS de a e b, temos:
- Caso base: dp[0][] = dp[][0] = 0
- Se os dois últimos elementos forem iguais: a[n-1] == b[m-1], então:

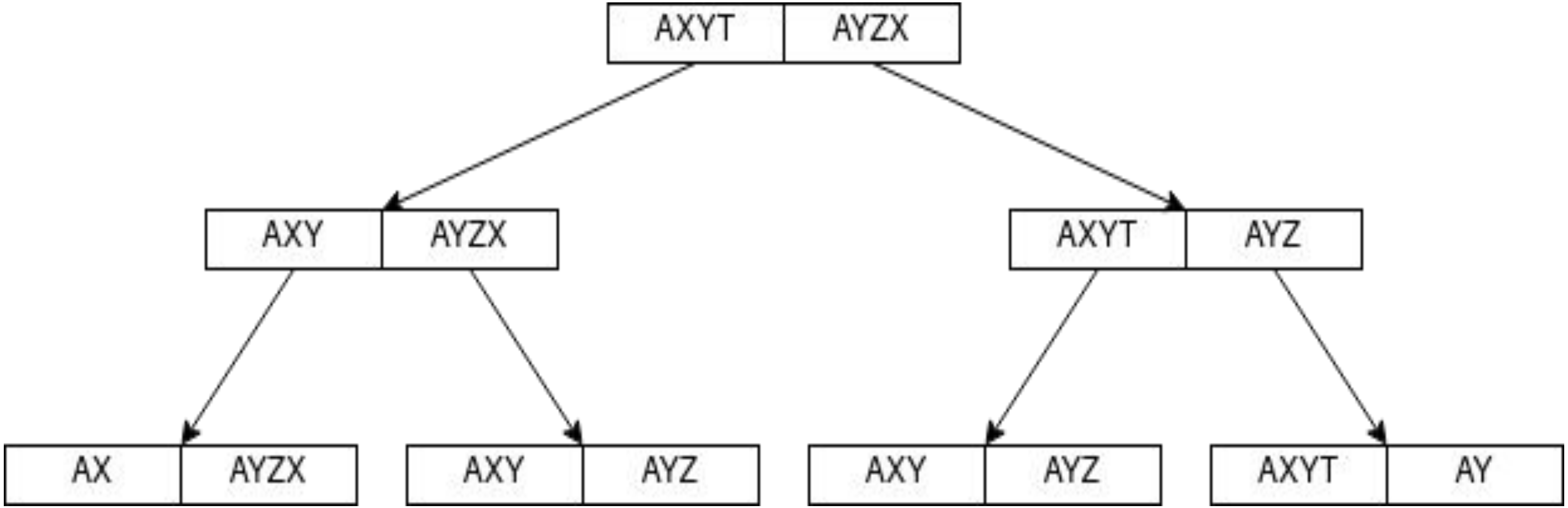
dp(a[0...n-1], b[0...m-1]) = 1 + dp(a[0...n-2], b[0...m-2])

- Se os dois últimos elementos forem diferentes: a[n-1] != b[m-1], então:

dp(a[0...n-1], b[0...m-1]) = max(dp(a[0...n-2], b[0...m-1]), dp(a[0...n-1], b[0...m-2]))

Longest Common Subsequence (LCS)

- Sobreposição de Subproblemas



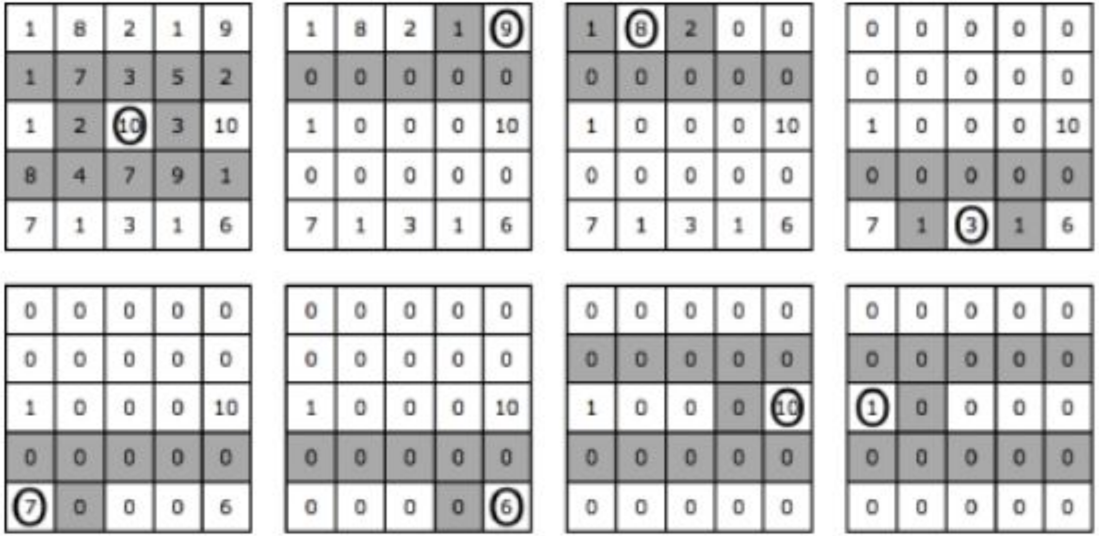
Longest Common Subsequence (LCS)

```cpp
// Complexidade de tempo: O(N*M)
int lcs(string& a, string& b) {
    int n = a.length(), m = b.length();
    vector<vector<int>> dp(n+1, vector<int>(m+1));
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= m; j++) {
            if (i == 0 || j == 0)
                dp[i][j] = 0;
            else if (a[i-1] == b[j-1])
                dp[i][j] = dp[i-1][j-1]+1;
            else
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }
    }
    return dp[n][m];
}
```

4212 - Candy (ICPC LIVE)

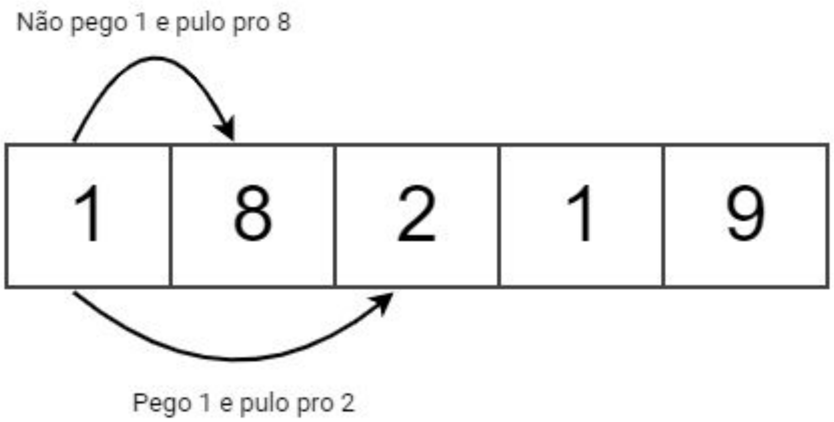
Motivação: obter o máximo de doces em um retângulo M x N

A cada doce pego, todos os doces vizinhos e da linha superior e inferior são descartados.



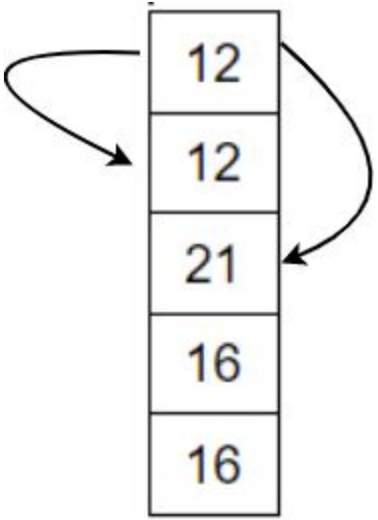
4212 - Candy (ICPC LIVE)

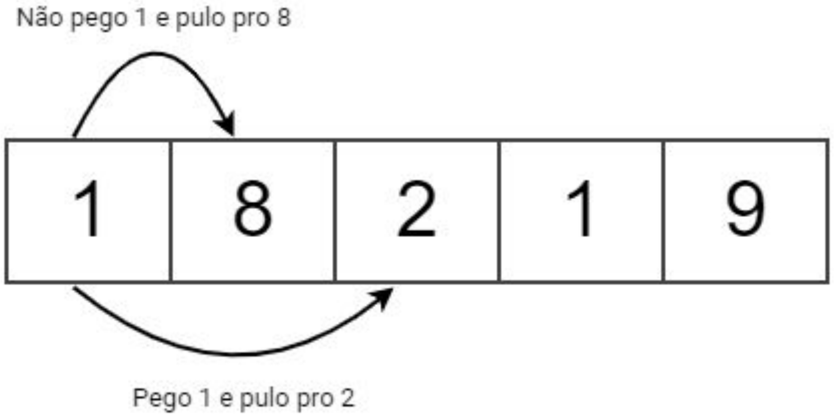
- Primeiro deve-se pensar no problema com a dimensão reduzida, como obter o máximo de doces em um vetor unidimensional.
- “Devo pegar este valor ou pular e pegar o próximo?”



4212 - Candy (ICPC LIVE)

- O máximo de doces que eu poderia pegar desta linha é 12 (1+2+9).
- Posso armazenar os resultados de cada linha em um vetor coluna e replicar a ideia, obtendo o máximo de doces do retângulo.





4212 - Candy (ICPC LIVE)

```
// Complexidade de tempo: O(N*M)

int dp[MAX], lin[MAX], col[MAX], r, c;

int solve(int i, int n, int *x)
{
    if (i >= n)
        return 0;
    if (dp[i])
        return dp[i];
    return dp[i] = max(x[i] + solve(i + 2, n, x), solve(i + 1, n, x));
}
```

4212 - Candy (ICPC LIVE)

```
// Complexidade de tempo: O(N*M)
int main()
{    int m, n;
     while (scanf("%d%d",&m,&n)==2 && (n || m))
     {
          for (int j = 0; j < m; j++)
          {
               for (int i = 0; i < n; i++)
                    scanf("%d",&lin[i]);
               fill(dp,dp+n,0);
               col[j] = solve(0,n,lin);
          }
          fill(dp,dp+m,0);
          printf("%d\n",solve(0,m,col));
     }
     return 0;
}
```

Referências

https://www.geeksforgeeks.org/greedy-algorithm-to-find-minimum-number-of-coins/

https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.57.3243&rep=rep1&type=pdf

https://panda.ime.usp.br/pythonds/static/pythonds_pt/04-Recursao/11-programacaoDinamica.html

https://www.geeksforgeeks.org/find-minimum-number-of-coins-that-make-a-change/

https://www.ics.uci.edu/~eppstein/161/960229.html

Referências

https://www.geeksforgeeks.org/greedy-algorithms/

https://www.geeksforgeeks.org/dynamic-programming/