# Paradigmas de Resolução de Problemas

## Programação Dinâmica: Mochila Binária

Prof. Edson Alves - UnB/FGA

2020

# Sumário

# Definição

# *Definição*

**Problema da Mochila Binária**

Considere uma conjunto $C = \{c_1, c_2, \ldots, c_N\}$, onde $c_i = (w_i, v_i)$, e seja $M$ um inteiro positivo.

O problema da mochila binária consiste em determinar um subjconjunto $S \subset \{1, 2, \ldots, N\}$ de índices de $C$ tal que

$$W = \sum_{j \in S} w_j \leq M$$

e que a soma

$$V = \sum_{j \in S} v_j$$

seja máxima.

# Características do problema da mochila binária

- Os elementos do conjunto $C$ são denominados objetos ou items
- $M$ é o capacidade da mochila
- $w_i$ é o peso/tamanho do objeto, o qual determina se o objeto pode ou não ser transportado na mochila, de acordo com a capacidade ainda disponível na mesma
- $v_i$ é o valor/ganho do objeto, e a soma dos valores dos objetos selecionados deve ser maximizada
- O termo "mochila binária" remete ao fato de que, para cada um dos objetos, há duas opções: escolhê-lo ou não

# Solução do problema da mochila binária

# Solução do problema da mochila binária

- Como $|C| = N$, há $2^N$ subconjuntos de índices a serem avaliados
- Como cada subconjunto pode ser avaliado em $O(N)$, uma solução de busca completa tem complexidade $O(N2^N)$
- É possível, entretando, reduzir esta complexidade por meio de um algoritmo de programação dinâmica
- Seja $v(i, m)$ a soma máxima dos valores que pode ser obtida a partir dos primeiros $i$ elementos de $C$ e uma mochila com capacidade $m$
- São dois casos-base: o primeiro deles acontece quando não resta mais nenhum elemento a ser considerado
- Neste casos, temos $v(0, m) = 0$

# Solução do problema da mochila binária

- O segundo caso-base acontece quando não há mais espaço disponível na mochila: $v(i, 0) = 0$
- Também são duas as transições possíveis:
  1. ignorar o $i$-ésimo elemento e considerar apenas os $i - 1$ primeiros; ou
  2. caso possar ser transportado, pegar o $i$-ésimo elemento e colocá-lo na mochila
- A primeira transição não modifica o estado da mochila e nem o total dos valores transportados
- Caso a mochila não consiga transportar o $i$-ésimo elemento, esta será a única transição possível
- Assim,

$$v(i, m) = v(i - 1, m), \quad \text{se } w_i > m$$

# Solução do problema da mochila binária

- A segunda transição só é possível se $w_i \leq m$
- Caso esta condição seja atendida, a capacidade da mochila é reduzida em $w_i$ unidades, e o total dos valores transportados é acrescido em $v_i$
- Assim, deve-se optar pela transição que produz o maior valor:

$$v(i, m) = \max\{\ v(i-1, m), v(i-1, m-w_i) + v_i\ \}, \quad \text{se } w_i \leq m$$

- A solução do problema será dada por $v(N, M)$
- O número de estados distintos é $O(NM)$ e cada transição é feita em $O(1)$
- Portanto a solução de programação dinâmica para o problema do troco tem complexidade $O(NM)$

# Implementação *top-down* da mochila binária

```
#include <bits/stdc++.h>

using namespace std;
using ii = pair<int, int>;
using ll = long long;

const int MAXN { 2010 }, MAXM { 2010 };

ll st[MAXN][MAXM];

ll dp(int i, int m, int M, const vector<ii>& cs)
{
    if (i < 0)
        return 0;

    if (st[i][m] != -1)
        return st[i][m];

    auto res = dp(i - 1, m, M, cs);
    auto [w, v] = cs[i];

```

# Implementação *top-down* da mochila binária

```cpp
    if (w <= m)
        res = max(res, dp(i - 1, m - w, M, cs) + v);

    st[i][m] = res;
    return res;
}

ll knapsack(int M, const vector<ii>& cs)
{
    memset(st, -1, sizeof st);

    return dp((int) cs.size() - 1, M, M, cs);
}

```

# Recuperação dos elementos selecionados

- Uma variante comum do problema é exibir a lista dos elementos selecionados que maximizaram a soma dos valores, respeitando a capacidade da mochila
- Para recuperar os elementos escolhidos é precisa uma tabela adicional $p$, com as mesmas dimensões da tabela de memorização
- Se $p(i, m) = 1$, a transição escolhida por $v(i, m)$ foi a segunda, isto é, o elemento $c_i$ foi escolhido
- Caso contrário, $p(i, m) = 0$
- Com esta informação basta recuperar os itens, usando os valores de $p(i, m)$ para rastrear os estados que levaram à solução ótima

# Implementação *bottom-up* da mochila binária

```cpp
#include <bits/stdc++.h>

using namespace std;
using ii = pair<int, int>;
using ll = long long;

const int MAXN { 2010 }, MAXM { 2010 };

ll st[MAXN][MAXM];
char ps[MAXN][MAXM];

pair<ll, vector<int>> knapsack(int M, const vector<ii>& cs)
{
    int N = (int) cs.size() - 1;  // Os elementos começam em 1

    // Casos-base
    for (int i = 0; i <= N; ++i)
        st[i][0] = 0;

    for (int m = 0; m <= M; ++m)
        st[0][m] = 0;
```

# Implementação *bottom-up* da mochila binária

```
    // Transições
    for (int i = 1; i <= N; ++i)
    {
        for (int m = 1; m <= M; ++m)
        {
            st[i][m] = st[i - 1][m];
            ps[i][m] = 0;

            auto [w, v] = cs[i];

            if (w <= m and st[i - 1][m - w] + v > st[i][m])
            {
                st[i][m] = st[i - 1][m - w] + v;
                ps[i][m] = 1;
            }
        }
    }
```

# Implementação *bottom-up* da mochila binária

```
    // Recuperação dos elementos
    int m = M;
    vector<int> is;

    for (int i = N; i >= 1; --i)
    {
        if (ps[i][m])
        {
            is.push_back(i);
            m -= cs[i].first;
        }
    }

    reverse(is.begin(), is.end());

    return { st[N][M], is };
}
```

# Aplicações

## *Subset sum*

- Um problema comum que pode ser resolvido pela mochila binária é o *subset sum*
- Dados $N$ inteiros positivos $\{x_1, x_2, \ldots, x_N\}$ e um inteiro $S$, o problema consiste em determinar se é possível escolher $k$ destes inteiros, com $1 \leq k \leq N$, de modo que a soma dos elementos escolhidos seja igual a $S$
- Veja que este problema pode ser reduzido a uma mochila binária
- A soma $S$ é a capacidade da mochila
- Os pesos dos elementos $c_i$ são dados pelos inteiros $x_i$
- Os valores podem ser desprezados
- Ao invés de escolher a transição de maior soma, deve-se considerar o "ou" lógico entre ambas
- O caso base $v(0, 0)$ será verdadeiro, e todos os demais casos-base serão falsos

# Implementação do *subset sum*

```cpp
#include <bits/stdc++.h>

using namespace std;

const int MAXN { 2010 }, MAXM { 2010 };

int st[MAXN][MAXM];

int dp(int i, int m, int M, const vector<int>& xs)
{
    if (i < 0)
        return m == 0 ? 1 : 0;

    if (st[i][m] != -1)
        return st[i][m];

    auto res = dp(i - 1, m, M, xs);

    if (xs[i] <= m)
        res |= dp(i - 1, m - xs[i], M, xs);

```

# Implementação do *subset sum*

```
    st[i][m] = res;
    return res;
}

bool subset_sum(int S, const vector<int>& xs)
{
    memset(st, -1, sizeof st);

    return dp((int) xs.size() - 1, S, S, xs);
}

```

# Referências


1. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.
2. **LAARKSONEN**, Antti. *Competitive Programmer's Handbook*, 2017.