

**CENTRO UNIVERSITÁRIO DE ANÁPOLIS UNIEVANGÉLICA**

**CURSOS SUPERIORES DE COMPUTAÇÃO**

**DISCIPLINA:** Análise e Complexidade de Algoritmos **CURSO:** Engenharia de Computação

**PROFESSOR:** Alexandre Moraes Tannus

**ACADEMICAS:1611296-** Laryssa Bitencourt Cardoso e **1612674-** Isabella Ribeiro Canedo

**RELATORIO**

O problema da mochila consiste em dado um conjunto Cn de n itens, representados por Cn = {1, 2, ..., n}, cada item i Î Cn tem um peso pi e utilidade ui (pi > 0 e ui > 0). Foi determinado um subconjunto S Í Cn tal que a soma dos pesos dos elementos de S seja a maior possível.

O primeiro algoritmo apresenta a resolução mais óbvia, o qual é implementado utilizando a força bruta, compara todas as possibilidades até encontrar a melhor. Este algoritmo devolve a resposta correta, mas quando consideramos um conjunto relativamente grande de dados, o tempo gasto para obtermos a resposta é exponencialmente difícil de ser encontrada, logo o tempo deste algoritmo é exponencial.

O segundo algoritmo é o que utiliza o método da programação dinâmica. Este algoritmo também encontra a resposta correta com um tempo melhor que o anterior, porém se aumentarmos consideravelmente a capacidade da mochila, este algoritmo gastará um tempo muito maior do que o esperado. Assim dizemos que este algoritmo possui um tempo pseudo-polinomial, mas na sua essência ele é exponencial.

**Programação Gulosa**

#include <iostream>

**using namespace** std;

**int** knapsack(int W, int wt[], int b[], int n)

{

// tabela que será preenchida

**int** V[n + 1][W + 1];

// inicializando a primeira linha e primeira coluna com 0

**for**(int w = 0; w <= W; w++)

V[0][w] = 0;

**for**(int i = 1; i <= n; i++)

V[i][0] = 0;

**for**(int i = 1; i <= n; i++)

{

**for**(int w = 1; w <= W; w++)

{

// elemento pode fazer parte da solução

**if**(wt[i - 1] <= w)

{

// max...

**if**((b[i - 1] + V[i - 1][w - wt[i - 1]]) > V[i - 1][w])

V[i][w] = b[i - 1] + V[i - 1][w - wt[i - 1]];

**else**

V[i][w] = V[i - 1][w];

}

**else**

V[i][w] = V[i - 1][w]; // wi > w

}

}

// retorna o valor máximo colocado na mochila

**return** V[n][W];

}

**int** main(int argc, char \*argv[])

{

// capacidade máxima da mochila: W

**int** W = 20;

// número de elementos

**int** n = 5;

// vetor com os valores (benefício) de cada elemento

**int** b[] = {3, 5, 8, 4, 10};

// vetor com os pesos de cada elemento

**int** wt[] = {2, 4, 5, 3, 9};

// obtém o máximo valor que pode ser colocado na mochila

**int** max\_valor = knapsack(W, wt, b, n);

**cout** << "Valor maximo: " << max\_valor << endl;

**return** 0;

**Programação Dinâmica**

##include <algorithm>

**using namespace** std;

// defino os maiores valores de n e s como 1010

#define MAXN 1010

#define MAXS 1010

// declaro as variáveis que a função utiliza

**int** n, valor[MAXN], peso[MAXN], tab[MAXN][MAXS]

**int** knapsack(int obj, int aguenta){

// se já calculamos esse estado da dp, retornamos o resultado salvo

**if**(tab[obj][aguenta]>=0) return tab[obj][aguenta];

// se não houver mais objetos ou espaço na mochila, retorno 0, pois não posso mais botar nada

**if**(obj==n or !aguenta) return tab[obj][aguenta]=0;

// não colocar avança para o estado em que tentmos o próximo, com o mesmo espaço disponível

**int** nao\_coloca=knapsack(obj+1, aguenta);

// se for possível colocar o objeto

**if**(peso[obj]<=aguenta){

// o melhor atingível é o valor dele mais o melhor entre colocar ou não os próximos

// que é o resultado do estado da dp em que olhamos o próximo objeto

// mas agora a mpchila aguenta o que aguentava antes menos o peso que coloquei nela

**int** coloca=valor[obj]+knapsack(obj+1, aguenta-peso[obj]);

// e a função deve retornar o melhor entre colocar ou não colocar

**return** tab[obj][aguenta]=max(coloca, nao\_coloca);

}

// se a função não retornou ainda, então ela não entrou no if

// logo não era possível colocar o objeto

**return** tab[obj][aguenta]=nao\_coloca;

// então retorno o valor de não colocá-lo

}