

Centro Federal de Educação Tecnológica de Minas Gerais

Engenharia de Computação

- Técnicas de programação -
Programação dinâmica

Adriel Araújo

André Neves

Pedro Otávio

Outubro de 2018

Sumário

1	Resumo	2
2	Introdução	3
2.1	Divisão e conquista - Merge sort	3
2.2	Superposição de subproblemas e a inviabilidade da divisão e conquista	4
2.2.1	Superposição de subproblemas - Sequência de Fibonacci	4
3	Programação Dinâmica	6
3.1	Introdução ao uso de programação dinâmica	6
3.2	Etapas para desenvolvimento de algoritmos de programação dinâmica	6
3.3	Onde aplicar a programação dinâmica?	7
3.3.1	Subestrutura Ótima	7
3.3.2	Superposição	7
3.4	Exemplos de algoritmos implementados com programação dinâmica	8
3.4.1	Sequência de Fibonacci	8
3.4.2	LCS - Longest Common Subsequence (Maior subsequência comum) . . .	10
3.5	Programação dinâmica para problemas de otimização combinatória	14
3.5.1	Problema do troco	14
3.5.2	Problema da mochila	16
3.6	Ordem de complexidade de algoritmos em programação dinâmica	21
4	Conclusão	22
5	Apêndice: Outros paradigmas de resolução de problemas	23
5.1	Busca Completa	23
5.2	Algoritmos gulosos	23

1 Resumo

Muitos problemas de computação, como o **Problema da Mochila**, por ter uma aparente complexidade de resolução, a princípio pode ser pensado com uma abordagem onde a força bruta é necessária, por ser a possibilidade mais trivial. Nesse caso são analisadas todas as possibilidades de caminhos para desenvolvimento do problema, para observar, dentre todas, qual é a solução ótima. Entretanto, esse tipo de algoritmo se mostra inadequado para casos onde a entrada é grande, já que esse tipo de problema geralmente possui uma quantidade exponencial de possibilidades em função do tamanho da entrada.

Nesse trabalho, abordaremos uma técnica avançada de algoritmos chamada **Programação dinâmica**, que em muitos problemas, é capaz de reduzir sua complexidade de exponencial para polinomial.

2 Introdução

Para muitos problemas, uma possível abordagem de resolução é a **Divisão e conquista**. Essa técnica consiste nos seguinte passos:

- Divisão de uma instância do problema em instâncias menores, definidas recursivamente;
- Resolução da(s) menores instâncias primeiramente
- Combinação dos resultados de instâncias menores para obter resultado de uma instância maior

Esta técnica é útil quando temos subproblemas independentes. Isso significa que não há subproblemas comuns entre dois problemas maiores, ou seja, nenhum subproblema aparece duas ou mais vezes. Essa característica também é reconhecida como **inexistência de superposição de subproblemas**. O algoritmo merge sort é um exemplo.

2.1 Divisão e conquista - Merge sort

Um algoritmo interessante e simples para ilustrar a divisão e conquista é o merge sort.

Seja o seguinte vetor:

8	4	1	20	15	13	9	10
---	---	---	----	----	----	---	----

Figura 1: Vetor desordenado

O merge sort consiste em dividir um vetor ao meio, ordenar cada metade separadamente, e então intercalar as duas metades, de forma a manter o vetor final ordenado. Caso o mergesort receba um vetor de um elemento, nada é feito. Com o algoritmo recursivo, a sucessão de operações está mostrada pelas figuras 2, 3, 4, 5 e 6.

8	4	1	20
---	---	---	----

15	13	9	10
----	----	---	----

Figura 2: 2 vetores e 4 elementos

8	4
---	---

1	20
---	----

15	13
----	----

9	10
---	----

Figura 3: 4 vetores de 2 elementos

4	8
---	---

1	20
---	----

13	15
----	----

9	10
---	----

Figura 4: 4 vetores de 2 elementos - grupos ordenados

1	4	8	20
---	---	---	----

9	10	13	15
---	----	----	----

Figura 5: 2 vetores de 4 elementos - grupos intercalados

1	4	8	9	10	13	15	20
---	---	---	---	----	----	----	----

Figura 6: Vetor completo - grupos intercalados

Nesse algoritmo é possível ver claramente um problema (ordenar um vetor) sendo dividido em subproblemas (ordenar cada metade do vetor). Cada subconjunto é tratado apenas uma vez, não havendo, assim, desperdício de uso computacional.

2.2 Superposição de subproblemas e a inviabilidade da divisão e conquista

Diferentemente do algoritmo de mergesort, existem problemas os quais, no processo de divisão, um mesmo subproblema é processado várias vezes. Nesse caso temos um problema de divisão e conquista com casos dependentes.

2.2.1 Superposição de subproblemas - Sequência de Fibonacci

Seja a sequência de Fibonacci, definida por:

$$F_n = \begin{cases} 1, & \text{Se } n \leq 2 \\ F_{n-1} + F_{n-2}, & \text{Se } n > 2 \end{cases}$$

Observe que é possível dividir o cálculo de qualquer número em subproblemas, onde o cálculo do número F_n é dividido no cálculo de F_{n-1} e F_{n-2} .

Supondo o cálculo de F_5 , temos os subproblemas mostrados na figura 7, e a quantidade de chamadas por parâmetros citada na tabela 1.

Número da sequência	Quantidade de chamadas
5	1
4	1
3	2
2	3
1	3

Tabela 1: Quantidade de chamadas de função

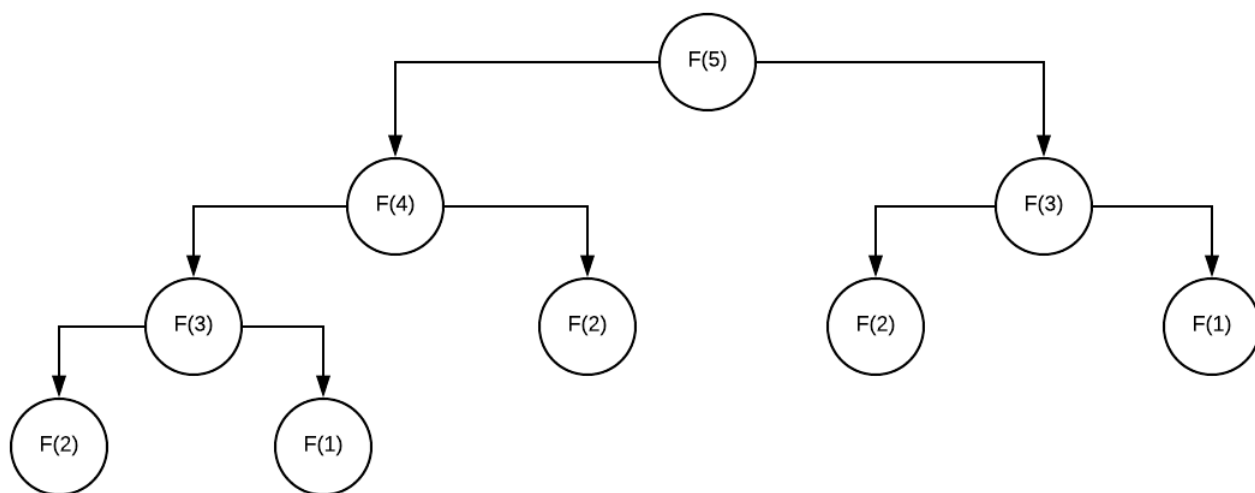


Figura 7: Divisão em subproblemas no cálculo de $F(5)$

Como mostra a Tabela 1, houve excesso de chamadas para os números 3 (1 vez), 2 (2 vezes) e 1 (2 vezes), ou seja, 5 chamadas desnecessárias. É possível prever que, quanto maior o número que se deseja calcular, maior o número de chamadas redundantes.

3 Programação Dinâmica

Programação dinâmica é uma técnica de resolução de problemas que pode ser vista como uma extensão à **Divisão e conquista**. Nesse caso, ela é útil para aqueles que possuem subproblemas dependentes, como o cálculo de algum número de Fibonacci.

A Programação dinâmica é baseada em particionar e resolver pequenos membros e armazenar as soluções para poder usá-las posteriormente, o que maximiza a eficiência, evita repetição de cálculo, e consequentemente, reduz a complexidade da solução.

3.1 Introdução ao uso de programação dinâmica

Para utilizar a técnica de programação dinâmica, é preciso definir a recorrência que resolve um problema maior em função de seus subproblemas.

Há dois tipos de resolução consideradas:

- **Top down:** Resolução do algoritmo de forma recursiva. Um problema é claramente definido em função de seus subproblemas;
- **Bottom up:** Resolução do algoritmo de forma iterativa. Primeiro calcula-se os valores menores, e em cada iteração, resolver os problemas maiores usando valores já calculados.

No geral, é mais fácil utilizar a abordagem **Top down**, já que a visão recursiva tende a ser mais natural ao analisar problemas desse tipo. Contudo, implementações Bottom up devem ser pensadas em caso de problemas que possuem potencial para exceder a pilha de recursão fornecida pela linguagem.

3.2 Etapas para desenvolvimento de algoritmos de programação dinâmica

1. Estruturar a solução ótima;
2. Definir o valor dessa solução recursivamente;
3. A partir da definição recursiva, opcionalmente é possível pensar na resolução bottom-up;
4. Construir uma solução ótima *a partir das soluções encontradas*;

Simplificando: A ideia básica é buscar a solução chamada ótima para o problema combinando as respostas obtidas para as partes menores.

3.3 Onde aplicar a programação dinâmica?

A programação dinâmica não é um método que resolve qualquer tipo de problema. Por isso, é necessário que o programador saiba quando e onde empregá-la. Logo, é possível descrever duas características que um problema deve possuir para que seja possível a aplicação da Programação dinâmica:

- Subestrutura ótima
- Superposição de problemas

3.3.1 Subestrutura Ótima

”Um problema apresenta uma subestrutura ótima se uma solução ótima para o problema contém em seu interior soluções ótimas para subproblemas. CORMEN, Tomas.H

Existe um padrão comum para descobrir se um problema possui uma subestrutura ótima:

- Mostrar que uma solução para o problema consiste em fazer uma escolha;
- Supor que, para um dado problema, você tem a escolha que conduz a uma solução ótima;
- Dada essa escolha, determinar quais subproblemas resultam dela;
- Mostrar que as soluções para os subproblemas usados dentro da solução ótima para o problema devem elas próprias ser ótimas.

3.3.2 Superposição

”Quando um algoritmo recursivo para um problema de otimização reexamina o mesmo problema inúmeras vezes, dizemos que o problema de otimização tem **subproblemas superpostos**.” CORMEN, Tomas.H.

Os algoritmos de programação dinâmica tiram proveito de subproblemas superpostos ao resolver e armazenar a resolução de cada subproblema uma única vez em uma tabela, onde ela possa ser examinada quando necessário.

3.4 Exemplos de algoritmos implementados com programação dinâmica

Abaixo serão listados alguns problemas básicos resolvidos com essa técnica. Problemas de otimização serão listados na próxima seção.


3.4.1 Sequência de Fibonacci

Como visto anteriormente, a sequência de Fibonacci é definida por:

$$F_n = \begin{cases} 1, & \text{Se } n \leq 2 \\ F_{n-1} + F_{n-2}, & \text{Se } n > 2 \end{cases}$$

Em uma abordagem Top down, podemos fazer uso literal dessa definição. O valor calculado para F_n deve ser armazenado para uso posterior.

Nas figuras 8, 9 e 10 estão os códigos de implementação do cálculo de número de Fibonacci, usando divisão e conquista básica, programação dinâmica Top-down e Bottom-up:



```
int fibonacci(int n) {  
    if(n <= 2)  
        return 1;  
  
    return fibonacci(n - 1) + fibonacci(n - 2);  
}
```

Figura 8: Fibonacci com divisão e conquista

```

int fibo[1000];

int fibonacci_pd(int n) {
    if(n <= 2)
        return 1;

    else if(fibo[n] == 0)
        fibo[n] = fibonacci_pd(n - 1) + fibonacci_pd(n - 2);

    return fibo[n];
}

```

Figura 9: Fibonacci com programação dinâmica Top-down

```

int fibonacci_pd_bottom_up(int n) {
    int fibo, ant, ant2;
    fibo = ant = ant2 = 1;

    for(int i = 3; i <= n; i++) {
        ant2 = ant;
        ant = fibo;

        fibo = ant + ant2;
    }

    return fibo;
}

```

Figura 10: Fibonacci com programação dinâmica Bottom-up

Os resultados dos testes com as três implementações estão descritos na tabela 2:

Implementação	Tempo (ms)
Divisão e conquista	14,0844
PD Top-down	0,00061
PD Bottom-up	0,00076

Tabela 2: Tempos de execução das diferentes implementações para cálculo do 30º termo da sequência e Fibonacci

3.4.2 LCS - Longest Common Subsequence (Maior subsequência comum)

Em aplicações biológicas, frequentemente é necessário comparar o DNA de dois ou mais organismos diferentes. Uma cadeia de DNA consiste em uma cadeia que pode ser formada das seguintes moléculas:

- Adenina
- Guanina
- Citosina
- Timina

Essas moléculas são representadas pelas letras A, G, C, T. Uma cadeia de DNA pode ser representada por uma sequência dessas seguintes moléculas.

Um objetivo da comparação de duas cadeias de DNA é determinar o quanto as duas cadeias são "semelhantes". Uma das análises que podem ser feitas é analisar a maior subsequência comum (Ou Longest Common Subsequence - LCS) entre duas cadeias de DNA. Uma LCS entre duas sequências de DNA S_1 e S_2 deve ser uma cadeia que pode ser formada inteiramente a partir de S_1 ou S_2 . **Exemplos:**

- **AGTCA** é uma subsequência de GTACTGTAGCA;
- **GGCCAT** é uma subsequência de GGGATCCGAT;
- **GGACA** é uma subsequência comum entre as duas sequências citadas acima.

3.4.2.1 Resolução por força bruta

Uma possível abordagem é gerar todas as subsequências da cadeia S_1 , e então, para cada uma, verificar se é uma subsequência de S_2 . Esse algoritmo tem ordem exponencial, já para uma cadeia de tamanho m , existem 2^m subsequências.

3.4.2.2 Resolução por programação dinâmica

Definição 1. (Subcadeia) Sejam as sequências X e Y . A notação X_i e Y_j referem-se às subcadeias de X e Y , representadas pelos caracteres $[x_0, x_1, x_2, \dots, x_i]$ e $[y_0, y_1, y_2, \dots, y_j]$ respectivamente.

Definição 2. (Função LCS) A função $LCS(X, Y, i, j)$ retorna o tamanho da maior subseqüência comum entre as subcadeias X_i e Y_j

O teorema a seguir mostra uma resolução que pode ser usada para programação dinâmica. (Cormen, Thomas H. Algoritmos - Teoria e Prática. Pg 282)

Teorema 1. (Subestrutura ótima de uma LCS)

Sejam as sequências $X = [x_1, x_2, \dots, x_i]$ e $Y = [y_1, y_2, \dots, y_j]$, e seja $Z = [z_1, z_2, \dots, z_k]$ qualquer LCS entre X e Y

- Se $x_i = y_j$, então $z_k = x_i = y_j$ e Z_{k-1} é uma LCS de X_{i-1} e Y_{j-1}
- Se $x_i \neq y_j$ e $z_k \neq x_i$, Z_k é uma LCS de X_{i-1} e Y_j
- Se $x_i \neq y_j$ e $z_k \neq y_j$, Z_k é uma LCS de X_i e Y_{j-1}

O teorema sugere que uma LCS de duas sequências contém dentro dela uma LCS de prefixos das duas strings, o que caracteriza uma subestrutura ótima.

A solução recursiva desse problema é:

$$LCS(i, j) = \begin{cases} 0, & \text{Se } i = 0 \text{ ou } j = 0 \\ LCS(i-1, j-1) + 1, & \text{Se } i, j > 0 \text{ e } x_i = y_j \\ \max(LCS(i, j-1), LCS(i-1, j)), & \text{Se } i, j > 0 \text{ e } x_i \neq y_j \end{cases}$$

Implementações top-down e bottom-up nas figuras 11 e 12.

A execução do LCS para as cadeias **GTACTGTAGCA** e **GGGATCCGAT** resulta nas tabelas 3 e 4.



Figura 11: Implementação do algoritmo LCS com programação dinâmica top-down

	G	T	A	C	T	G	T	A	G	C	A
G	1	1	1	1	1	1	1	1	-1	-1	-1
G	1	1	1	1	1	2	2	2	-1	-1	-1
G	1	1	1	1	1	2	2	-1	3	-1	-1
A	1	-1	2	2	2	2	-1	3	3	-1	-1
T	1	2	2	-1	3	3	3	3	3	-1	-1
C	1	2	2	3	3	3	3	3	3	-1	-1
C	1	2	2	3	3	3	3	3	-1	4	-1
G	1	2	2	3	3	4	4	-1	4	4	-1
A	-1	-1	3	3	3	4	-1	5	5	5	5
T	-1	-1	-1	-1	-1	-1	5	5	5	5	5

Tabela 3: Tabela resultante da execução do algoritmo LCS com programação dinâmica top-down. Células com -1 representa que a PD não passou por lá

A trajetória destacada na tabela 3 representa o caminho percorrido pelo algoritmo para encontrar a LCS. A cada caminho na diagonal, significa que houve igualdade entre caracteres das duas strings. Logo, esse caractere deve fazer parte da LCS. Com isso, uma possível LCS é **GGAGA**.

```

int lcs_bottomup(int i, int j) {
    if(i < 0 || j < 0)
        return 0;

    for(int a = 0; a <= i; a++) {
        for(int b = 0; b <= j; b++) {
            if(X[a] == Y[b]) {
                if(a == 0 || b == 0)
                    pd[a][b] = 1;
                else
                    pd[a][b] = pd[a-1][b-1] + 1;
            }
            else {
                if(a == 0 && b == 0) {
                    pd[a][b] = 0;
                } else if(a == 0 || b == 0) {
                    if(a == 0)
                        pd[a][b] = pd[a][b-1];
                    else
                        pd[a][b] = pd[a-1][b];
                } else
                    pd[a][b] = max( pd[a][b-1], pd[a-1][b] );
            }
        }
    }

    return pd[i][j];
}

```

Figura 12: Implementação do algoritmo LCS com programação dinâmica bottom-up.

	G	T	A	C	T	G	T	A	G	C	A
G	1	1	1	1	1	1	1	1	1	1	1
G	1	1	1	1	1	2	2	2	2	2	2
G	1	1	1	1	1	2	2	2	3	3	3
A	1	1	2	2	2	2	2	3	3	3	4
T	1	2	2	2	3	3	3	3	3	3	4
C	1	2	2	3	3	3	3	3	3	4	4
C	1	2	2	3	3	3	3	3	3	4	4
G	1	2	2	3	3	4	4	4	4	4	4
A	1	2	3	3	3	4	4	5	5	5	5
T	1	2	3	3	4	4	5	5	5	5	5

Tabela 4: Tabela resultante da execução do algoritmo LCS com programação dinâmica bottom-up.

3.5 Programação dinâmica para problemas de otimização combinatória

3.5.1 Problema do troco

Dado um sistema monetário que possui um conjunto de moedas $S = 1, 4, 5, 10$, deseja-se dar o troco para c centavos com a menor quantidade de moedas possível.

Um possível algoritmo guloso poderia usar a estratégia de sempre começar a dar o troco com a maior moeda possível. Por exemplo, para $c = 7$, podemos começar a distribuição com uma moeda de 1, 4 ou 5. Como primeiramente é sempre usada a maior moeda, a primeira opção seria 5, pois é a maior possível. Após ela, não é possível dar outra de 5 nem de 4, pois extrapola $c = 7$. Resta então dar duas moedas de 1: $5 + 1 + 1 = 7$.

Essa estratégia não funcionaria com $c = 8$. A primeira moeda escolhida seria 5 centavos, já que é a maior dentre as possíveis. Com isso restaria dar o troco de 3 centavos, o que necessita de três moedas de 1 centavo. Isso resulta em quatro moedas, porém é possível perceber que a melhor solução é dar duas moedas de 4 centavos.

A solução ótima desse problema consiste em verificar todas possibilidades, porém como existem propriedades de subestrutura ótima e superposição, é cabível uma abordagem em programação dinâmica.

Seja o caso de resolver o problema de dar o troco de 8 centavos. Para começar a dar o troco, podemos dar uma moeda de 1, 4 ou 5 centavos, restando dar o troco de 7, 4 e 3 centavos, como mostra a figura 13:

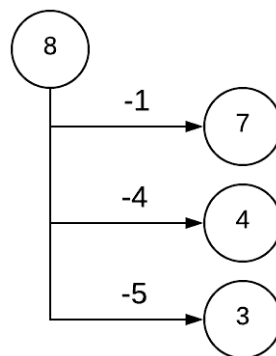


Figura 13: Possibilidades para dar troco de 8 centavos

Já foi citado que para um troco de 3 centavos, são necessárias três moedas de 1, e com isso, dar um troco de 8 centavos que utilize esse caminho gastaria 4 moedas.

Em contrapartida, dar um troco de 4 centavos, necessita apenas de 1 moeda. Logo, um troco de 8 centavos poderia gastar apenas 2. A figura 14 mostra esses dois caminhos.

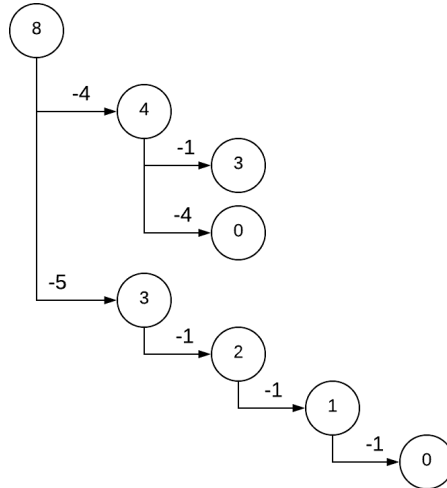


Figura 14: Possibilidades para dar troco de 8 centavos, diagrama estendido.

A estratégia adotada consiste em verificar, dado um troco de valor V , para cada possibilidade de moeda inicial, qual é o número de moedas necessárias para retornar o valor restante. A melhor possibilidade é sempre a que requer menos moedas. Para cada verificação, é armazenado na tabela qual foi o menor número de moedas necessárias para dar um valor V de troco, e a primeira moeda utilizada. Os subproblemas são demonstrados na figura 14.

O algoritmo implementado é mostrado na figura 15. O vetor $qt[]$ deve começar com -1 em todas células, os vetores $qt[]$ e $moedaAnterior[]$ devem ter o tamanho da entrada do problema, e $moedas[]$ deve ter todas as moedas possíveis. Caso a função retorne -1 ou 0 , significa que é impossível obter tal troco.

A execução do algoritmo do troco resulta na tabela 5. Para cada **Valor de troco**, é especificado a quantidade de moedas necessárias e qual a primeira moeda que deve ser dada:

Valor de troco	0	1	2	3	4	5	6	7	8
Quantidade de moedas necessárias	0	1	2	3	1	1	2	3	2
Moeda Anterior	-1	1	1	1	4	5	1	1	4

Tabela 5: Tabela resultante da execução do algoritmo do problema do Troco com programação dinâmica.


```

int troco(int valor) {
    if (valor == 0)
        return qt[valor] = 0;

    if (qt[valor] < 0) {
        for (int i = 0; i < moedas.length; i++) {
            if (valor - moedas[i] < 0)
                continue;

            int num = troco(valor - moedas[i]);

            if (num + 1 < qt[valor] || qt[valor] == -1) {
                qt[valor] = num + 1;
                moedaAnterior[valor] = moedas[i];
            }
        }
    }

    return qt[valor];
}

```

Figura 15: Implementação do algoritmo do troco.

3.5.2 Problema da mochila

Dada uma mochila com capacidade máxima C , e um conjunto $S = \{O_1, O_2, O_3, \dots, O_n\}$ de objetos distintos, cada um com um peso P_i e valor V_i . Pede-se para determinar o valor máximo que é possível conseguir com um subconjunto desses objetos, de forma que o peso total não exceda a capacidade da mochila.

Exemplo: Considere uma mochila com capacidade $C = 5$ Kg e o conjunto de objetos citados na tabela 6:

Número	Peso (Kg)	Valor (R\$)
1	4	85,00
2	2	40,00
3	2	40,00
4	3	46,00

Tabela 6: Objetos usados no problema da mochila

A princípio, poderia ser pensada em uma solução que utiliza alguma abordagem de algoritmo guloso. Existem duas estratégias gulosas possíveis para esse problema:

- **Prioridade a objetos com valores maiores:** Em muitos casos essa estratégia pode oferecer a solução ótima, quando não há muita diferença entre o peso dos objetos.

Ao escolher primeiramente o objeto 1, acumulará um valor de R\$ 85,00, porém restará apenas espaço para objetos com peso menor ou igual a 1, com isso mais nenhum objeto

poderá ser incluído.

Ao fazer a escolha dos objetos 3 e 4, possíveis de serem incluídos na mochila, acumula-se um valor de R\$ 86,00, o que mostra que essa estratégia gulosa não forneceu a solução ótima.

- **Prioridade a objetos de maior densidade de valor:** Já que o primeiro algoritmo não funciona sempre, podemos pensar em uma estratégia um pouco mais esperta, baseada na análise de valor por quilograma que o objeto possui. Se dois objetos possuem o mesmo valor, o que possui menor peso terá prioridade na escolha do subconjunto de composição da mochila.

Esse método parece no mínimo mais eficiente que o primeiro, já que busca uma média entre valores maiores e pesos menores. Os cálculos da densidade de valor de cada objeto estão citados na tabela 7:

Número	Peso (Kg)	Valor (R\$)	Densidade de valor (R\$ / Kg)
1	4	85,00	21,25
2	2	40,00	20,00
3	2	40,00	20,00
4	3	46,00	15,56

Tabela 7: Objetos usados no problema da mochila com cálculo da densidade de valor

Novamente, essa metodologia levaria a escolher primeiramente o objeto 1, o que resultaria novamente em uma escolha não ótima.

Embora os algoritmos gulosos para esse problema pareçam ser eficazes, eles podem falhar em casos especiais, como o citado acima. Com isso, no momento a opção que resta é fazer uma verificação por força bruta. Isso é, testar todas possibilidades de preenchimento da mochila e verificar qual é o valor que pode ser obtido.

Uma forma de realizar essa tarefa é gerar todos os possíveis subconjuntos de objetos, e para cada um, verificar se é possível incluí-lo na mochila, e armazenar o maior valor possível de ser obtido. A figura 16 mostra uma possível implementação com essa abordagem.

```

int mochila_subconjuntos(int pesos[], int valores[], int C) {
    int n = pesos.length;
    int num = 1 << n; // num = 2^n
    int valorMaximo = 0;

    for (int i = 0; i < num; i++) {
        int peso = 0, valor = 0;
        for (int j = 0; j < n; j++) {
            int a = i & (1 << j);
            a = a >> j;

            if (a == 1) {
                peso += pesos[j];
                valor += valores[j];
            }

            if (peso <= C && valor >= valorMaximo)
                valorMaximo = valor;
        }
    }
    return valorMaximo;
}

```

Figura 16: Implementação do algoritmo da mochila com geração de subconjuntos.

Claramente esse algoritmo funciona, pois ao gerar subconjuntos, são testados todos os casos possíveis. Apesar disso, a eficiência desse algoritmo é baixa para um número muito grande de objetos. A explicação desse fato é: Dado um conjunto S de tamanho n , existem 2^n subconjuntos possíveis, e com isso o algoritmo mostrado tem complexidade $O(2^n)$.

É possível uma resolução alternativa com a utilização de programação dinâmica. O estado da DP tem a seguinte definição:

- Seja:
 - n o número de objetos disponíveis;
 - i o índice de algum objeto, com $1 \leq i \leq n$;
 - C_k o peso máximo permitido para alocar todos objetos j , tal que $i \leq j \leq n$.
- O valor e peso de cada objeto i são representados por $v[i]$ e $p[i]$;
- $M(n, i, C_k)$ representa o valor máximo que pode ser obtido ao colocar a melhor combinação dos objetos j .

Com isso, a definição da função é:

$$M(i, C_k) = \begin{cases} 0, & \text{Se } i > n \text{ ou } C_k \leq 0 \\ M(i + 1, C_k) & \text{Se } p[i] > C_k \\ \max(M(i + 1, C_k), v[i] + M(i + 1, C_k - p[i])) & \text{Caso contrário} \end{cases} \quad (1)$$

O primeiro caso representa as situações onde não há mais espaço disponível ou já foi considerada a colocação de todos objetos. Nesse caso, o valor acumulado é 0.

O segundo caso representa a situação onde o peso do objeto i é maior que a capacidade da mochila, e com isso não é possível considerar a alocação desse objeto.

O terceiro caso considera duas hipóteses:

- **Não colocar o objeto i :** Com isso, é calculado o valor máximo que é possível obter ao colocar todos objetos a partir de $i + 1$. Como o objeto i não foi colocado, o espaço disponível para alocação dos $i + 1$ objetos seguintes não se altera.
- **Colocar o objeto i :** É calculado para essa hipótese o valor máximo que é possível obter ao colocar todos objetos a partir de $i + 1$ no espaço restante. Como o objeto i foi colocado, a capacidade da mochila foi reduzida.

Com isso, verifica-se qual hipótese fornece o maior valor, gerando a decisão de colocar ou não o objeto i .

Segue na figura 17 a implementação do algoritmo de PD para o problema da mochila:



```
int mochila_dp(int obj, int capacidade) {
    if (dp[obj][capacidade] < 0) {
        if (capacidade <= 0 || obj == n) {
            dp[obj][capacidade] = 0;
        } else {
            int nao_coloca = mochila_dp(obj + 1, capacidade);

            if (pesos[obj] <= capacidade) {
                int coloca = valores[obj];
                coloca += mochila_dp(obj + 1, capacidade - pesos[obj]);

                dp[obj][capacidade] = max(coloca, nao_coloca);
            } else {
                dp[obj][capacidade] = nao_coloca;
            }
        }
    }
    return dp[obj][capacidade];
}
```

Figura 17: Implementação do algoritmo da mochila com programação dinâmica top-down.

```

int mochila_dp_bottomUp(int pesos[], int valores[], int C) {
    int n = pesos.length;

    for(int i = n; i >= 0; i--) {
        for(int Ck = 0; Ck <= C; Ck++) {
            if(i == n || Ck == 0) {
                dp[i][Ck] = 0;
            } else if(pesos[i] > Ck) {
                dp[i][Ck] = dp[i + 1][Ck];
            } else {
                dp[i][Ck] = max(dp[i + 1][Ck], valores[i] + dp[i + 1][Ck - pesos[i]]);
            }
        }
    }
    return dp[0][C];
}

```

Figura 18: Implementação do algoritmo da mochila com programação dinâmica bottom-up.

Os vetores $pesos[n]$ e $valores[n]$ representam as propriedades de cada objeto, e a matriz $dp[n + 1][C + 1]$ representa os estados da programação dinâmica.

As tabelas 8 e 9 mostram os resultados da execuções dos algoritmos de programação dinâmica top-down e bottom-up, respectivamente. As tabelas atendem às especificações da equação 1

	0	1	2	3	4	5
1	-1	-1	-1	-1	-1	86
2	-1	0	-1	-1	-1	86
3	-1	0	-1	46	-1	86
4	-1	0	-1	46	-1	46

Tabela 8: Tabela resultante da execução do algoritmo do problema do mochila com programação dinâmica top-down. As colunas representam as capacidades das mochilas nos subproblemas. As células com valor igual a -1 não foram processadas.

	0	1	2	3	4	5
1	0	0	40	46	85	86
2	0	0	40	46	80	86
3	0	0	40	46	46	86
4	0	0	0	46	46	46

Tabela 9: Tabela resultante da execução do algoritmo do problema do mochila com programação dinâmica bottom-up. É possível perceber que nesse caso todas as células foram processadas, o que gera maior custo computacional em relação à solução top-down, que processa apenas o necessário.

3.6 Ordem de complexidade de algoritmos em programação dinâmica

A complexidade de um algoritmo em programação dinâmica está diretamente relacionada com a quantidade de parâmetros que envolvem a equação de recorrência.

Como foi visto anteriormente, problemas como LCS e mochila utilizam dois parâmetros, e gera uma tabela bidimensional, já algoritmos como fibonacci e troco utilizar apenas um parâmetro, e uma tabela unidimensional.

No pior caso, será necessário preencher completamente a tabela. Porém, pelo propósito de eficiência da programação dinâmica, cada célula da tabela é processada apenas uma vez.

Seja K o custo de processar cada célula da tabela, n o número que representa a maior ordem de grandeza de todos os parâmetros, e ρ a quantidade de parâmetros da equação de recorrência. O algoritmo nesse caso tem complexidade $O(K \cdot n^\rho)$.

Isso significa que os problemas Fibonacci e troco tem ordem de complexidade linear, enquanto LCS e mochila tem ordem de complexidade quadrática, além de mostrar que qualquer problema resolvido com programação dinâmica tem complexidade polinomial.

4 Conclusão

Como podemos perceber, a programação dinâmica é um tipo de algoritmo voltado para problemas de otimização. Sua maior dificuldade está em descobrir a expressão de recorrência para determinada situação, lembrando que nem sempre será possível usar uma PD para resolver um problema. Também é importante lembrar que a eficiência de problemas implementados usando PD está no aproveitamento do recálculo existente dos subproblemas que ficam armazenados em uma tabela.

Apesar de requerer um custo adicional de memória para alocação da tabela, a programação dinâmica apresenta-se como uma ótima alternativa, que possui como grande vantagem evitar repetição de processamento.

Uma solução que seja possível ser pensada em programação dinâmica terá grande vantagem de tempo de processamento, pois como foi visto, a complexidade de tempo é proporcional ao tamanho da tabela.

5 Apêndice: Outros paradigmas de resolução de problemas

Além da programação dinâmica e divisão e conquistas, podemos pensar nos seguintes paradigmas especiais para resolução de diversos problemas:

1. Busca Completa;
2. Algoritmos gulosos.

5.1 Busca Completa

Um algoritmo de busca completa pesquisa exaustivamente todas as possibilidades dadas pela definição do problema. Desta forma, mesmo havendo sub problemas que já foram solucionados anteriormente, o algoritmo refaz todos os passos necessários para encontrar a mesma solução.

Em uma analogia, poderíamos dizer que o algoritmo pesquisa nó por nó de uma árvore até encontrar aquele que estamos procurando.

5.2 Algoritmos gulosos

Um algoritmo guloso sempre faz a escolha que parece melhor no momento e nunca reconsidera o que escolheu. Esse princípio é utilizado esperando que as escolhas locais o leve a solução ótima para o problema.

Apesar de eficientes para alguns problemas, os algoritmos gulosos nem sempre chegam à solução ótima.

Um dos principais algoritmos considerado guloso e de grande importância para a computação e comunicação de dados é o "Algoritmo de compressão de Huffman". Esse algoritmo visa otimizar a quantidade de bits no envio de determinada informação digital, utilizando menor quantidade de bits para representar dados mais frequentes e o contrário para dados menos frequentes.

Referências

- [1] Avrim Blum. **Dynamic Programming**. Disponível em: [<https://www.cs.cmu.edu/~avrim/451f09/lectures/lect1001.pdf>] Acesso em 15 de Outubro de 2018.
- [2] Lucas Schmidt Cavalcante (ICMC USP). **Programação Dinâmica I - Algoritmos Avançados**. Disponível em: [<http://wiki.icmc.usp.br/images/1/1a/PD1.pdf>] Acesso em 03 de Outubro de 2018.
- [3] Rogério Júnior. **Maior Subsequência Comum**. Disponível em: [<http://www.codcad.com/lesson/40>] Acesso em 10 de Outubro de 2018.
- [4] Rogério Júnior. **Troco**. Disponível em: [<http://www.codcad.com/lesson/48>] Acesso em 10 de Outubro de 2018.
- [5] Rogério Júnior. **O Problema da Mochila**. Disponível em: [<http://www.codcad.com/lesson/39>] Acesso em 11 de Outubro de 2018.
- [6] Algoritmos: teoria e prática/ Tomas H. Cormen...[et al.]; tradução da segunda edição [americana] Vandenberg D. de Souza. -Rio de Janeiro: Campus, 2002 - 13ª reimpressão.