UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
São José do Rio Preto - SP

JOGI SUDA NETO

PROGRAMAÇÃO DINÂMICA E APLICAÇÕES

São José do Rio Preto - SP
2020

JOGI SUDA NETO

PROGRAMAÇÃO DINÂMICA E APLICAÇÕES

Dissertação apresentada como parte dos requisitos para a disciplina de Análise e Projeto de Algoritmos, junto ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de São José do Rio Preto.

Prof. Dr. Geraldo Francisco Donegá Zafalon

São José do Rio Preto - SP
2020

RESUMO

Muitos problemas computacionais-matemáticos de otimização apresentam estruturas que contém subestruturas seguindo alguma lógica de recursão, como o clássico exemplo de Fibonacci, ou o problema binário da mochila (estes e mais um exemplo serão explicados posteriormente). Muitas vezes, então, uma solução recursiva clássica acaba apresentando uma abordagem elegante e simples para a resolução.

Acontece, porém, que utilizar tal estratégia de maneira "ingênua"pode acarretar em um aumento de complexidade desnecessário do algoritmo: cada subproblema repetido (caso exista), será calculado inutilmente. Aqui será explicado um método de resolução desses problemas que elimina o uso repetitivo e desnecessário destas subestruturas, apresentando significativa melhora de desempenho em termos de complexidade: este método é conhecido como Programação Dinâmica, e como será visto, se trata basicamente da resolução destes problemas através do uso de algum tipo de memória que armazenará o resultado dos subproblemas já calculados.

Palavras-chave: Programação Dinâmica, Recursão, Otimização.

Lista de Figuras

Lista de Tabelas

Sumário

1 Introdução

.

1.1 História

Apesar de conceitos de programação Dinâmica aparecerem com outros pesquisadores anteriormente, como Von Neumann e Morgenstern em um seminário sobre teoria dos jogos (1944) (Von Neumann et al., 2007), o crédito é atribuído de fato a Richard Bellman em seu trabalho feito em 1949. "Bellman ... explicou que ele inventou o nome 'programação dinâmica' para esconder o fato que ele estava fazendo pesquisa matemática na RAND sob um secretário de defesa que tinha 'medo patológico e ódio do termo, pesquisa'. Ele escolheu o termo 'programação dinâmica' porque seria difícil dar um 'significado pejorativo' e porque 'era algo que nem mesmo um Congressista poderia se opuser a'" (Rust, 2008).

1.2 Requisitos

Para aplicar Programação Dinâmica em algum problema, existem alguns requisitos (Cormen et al., 2001):

- **Subestrutura ótima** (ou "Princípio da Optimalidade" (Bellman, 1957)): um problema contém uma subestrutura ótima se sua solução ótima maior contém soluções ótimas para suas subestruras.

- **Repetição de subestruturas**: a ideia principal da Programação Dinâmica é calcular subproblemas apenas uma vez, tirando proveito de algum tipo de memória que armazenará cada solução de cada subproblema. Assim, em um problema com muitas subestruturas repetidas, consegue-se tirar boa vantagem com o uso desta técnica.

- **Independência entre subestruturas**: uma subestrutura depender de outra pode implicar na formação de ciclos, o que jamais deve acontecer, pois o algoritmo nunca convergirá em uma solução.

1.3 Memoização

Não confundir com "memorização": apesar dos dois termos terem **mesma** origem etimológica do latim (*memorandum*, "algo a ser lembrado"), o termo "memoização"possui um sentido específico na computação, e aqui será explicado seu uso. Trata-se da primeira e principal estratégia de programação dinâmica: na solução recursiva, o primeiro passo sempre será verificar, antes de tudo, se o cálculo com os valores das variáveis passados por parâmetro já foram realizados anteriormente, ou seja, se já estão em nossa "memória"(que pode ser simplesmente um vetor, uma matriz, ou, de maneira geral, algum tensor de dimensão n). Se já estiverem, retornaremos a chamada atual imediatamente, evitando recálculos repetidos. Caso não estejam, seguimos normalmente com a solução recursiva, com um porém: antes da chamada atual retornar, os valores que foram calculados através de outras chamadas-filha deverão ser armazenados nessa memória. Com os exemplos, a ideia deste procedimento seguirá mais intuitiva. Interessante ressaltar que este método se utiliza da recursão, e portanto, é classificado como um método top-down (pois começamos o problema do topo e descemos recursivamente até o(s) caso(s) base, e depois retornando).

1.4 Bottom-up

Os problemas de programação dinâmica também podem ser resolvidos sem utilizar recursão: é do que se trata esta abordagem. Ainda haverá o uso de nossa "memória", porém os cálculos serão feitos de maneira iterativa. Uma vantagem deste método é que ele pode economizar complexidade de espaço. Já em termos de complexidade de tempo, é equivalente ao método principal.

2 Exemplo: Fibonacci

2.1 Solução Recursiva

Talvez o exemplo mais clássico dos textos introdutórios sobre recursão simples em programação, a sequência de Fibonacci é um ótimo exemplo inicial para entendermos melhor como aplicar PD. Relembrando sua fórmula de recorrência, temos: Fn = Fn-1 + Fn-2, para n = 1, 2, 3, ..., tal que F1 = F2 = 1 (casos-base). A solução simples recursiva é a seguinte:

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

Para n $\leq$ 2, retorna-se 1 (casos-base). Supondo que $fibonacci(n)$ esteja correto, temos que $fibonacci(n+1) = fibonacci(n) + fibonacci(n-1)$ por definição. Logo, por indução garantimos a corretude do algoritmo acima. E a árvore de recursão para a solução inicial é apresentada abaixo:

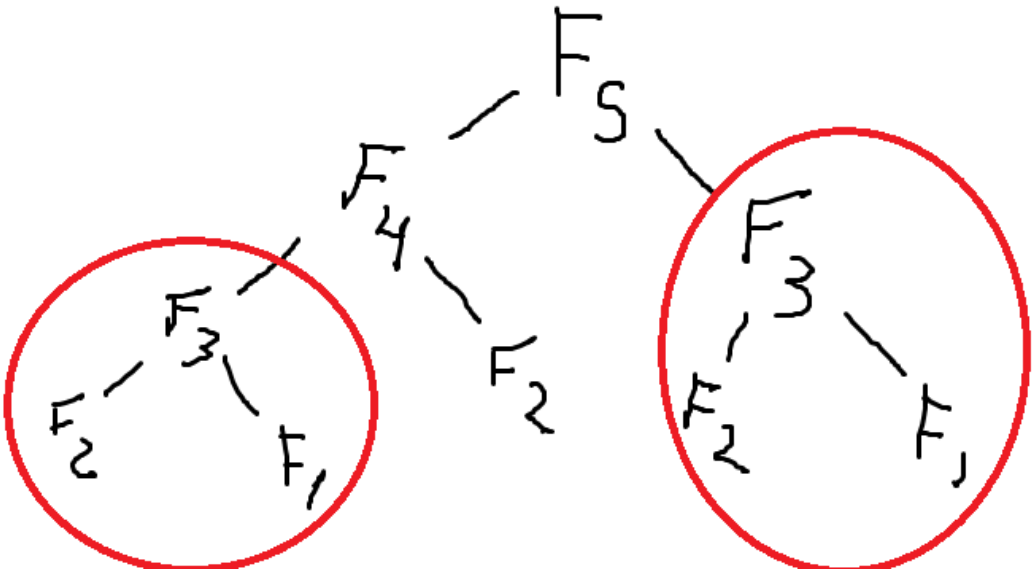

Figura 1:

Como pode se observar no exemplo, Ao calcular F5, há repetição das subestruturas F3 uma vez (e, por conseguinte, os casos-base também). Não é difícil demonstrar que a complexidade deste simples algoritmo é exponencial:

2.2 Solução Memoizada

Agora, será apresentada uma nova solução utilizando memoização. Para isto, basta lembrar que a solução recursiva continua a mesma, com a diferença que devemos salvar os resultados calculados (neste exemplo, serve um vetor).

```
int memo[n] = indefinido; #vetor inteiro começa com "indefinido".
int fibonacci(n){
    if (memo[n] != indefinido)
        return memo[n];
    if (n <= 2)
        return 1;
    memo[n] = fibonacci(n-1) + fibonacci(n-2);
    return memo[n];
}
```

Como pode ser visto, as mudanças são mínimas, e o que muda é o fato de, logo no começo de nossa função recursiva, verificarmos se o que foi passado nos parâmetros já não foi calculado em alguma outra chamada. Fazemos isto verificando se nossa memória (aqui, um vetor) já contém tal registro. Se houver, retornamos imediatamente o valor que já fora calculado. Caso negativo, prosseguimos, sendo que no final, antes de retornarmos a chamada atual, guardamos o que foi calculado nesta instância no vetor.

Agora, na nova árvore de recursão para o mesmo exemplo anterior, cada subarvore repetida será calculada apenas uma vez, ou seja, para encontrar fibonacci(n), o novo algoritmo calculará fibonacci(n-1), fibonacci(n-2), ..., fibonacci(1) apenas uma vez, e portanto a nova complexidade de tempo será O(n).

2.3 Solução Bottom-Up

A estratégia bottom-up também é simples e um algoritmo para a função de fibonacci é apresentado abaixo:

```
int fibonacci(n){
    int memo[n] = indefinido;
    for(int i=1; i <= n; i++){
        if(n <= 2) #casos-base
            memo[n] = 1;
        else
            memo[n] = memo[n-1] + memo[n-2];
    }
}
```

Como pode ser observado, não há uso de recursões, e a complexidade de tempo continua O(n), com a diferença que, se quisermos ir desalocando as posições anteriores de nossa memória, podemos assim fazer, pois para a i-ésima iteração, precisamos apenas das posições i-1 e i-2 do vetor.

3 Aplicação: problema binário da mochila

Aqui trataremos outro clássico problema da computação: dado uma mochila com capacidade C (com $C \geq 0$) e um conjunto de n items, cada um ocupando um espaço Wi e valor Vi, queremos encontrar o subconjunto máximo de itens que conseguimos colocar na mochila de modo a maximizar o preço total.

3.1 Introdução

Antes de apresentarmos uma solução recursiva simples, imaginemos o seguinte: ao começarmos a analisar o primeiro item, temos duas escolhas possíveis: colocá-lo ou não na mochila. Uma solução que analisaria todas as possibilidades de subconjuntos de itens seria, a cada instância de nossa função, retornar o seguinte: $max(MochilaBin(i+1, C), Vi + MochilaBin(i+1, C - Wi)$ se $Wi < C)$. Ou seja, queremos encontrar o máximo entre o subproblema onde o primeiro item **não** seja incluso na mochila e o subproblema onde o mesmo **seja** incluso na mochila (repare que o novo espaço disponível será $C - Wi$).

Sobre os casos-base: temos dois cenários que podem acontecer. Um ocorrerá quando, em alguma instância de nossa análise, a mochila estiver cheia. O outro será quando tivermos chegado ao fim da lista de itens, sendo que os dois casos podem ocorrer simultaneamente. Sendo a assim, a função deve retornar zero.

3.2 Solução Recursiva

Com as informações acima, tentando uma solução recursiva simples, chegamos em:

```
int mochilaBinaria(n, C){
    int tmp1;
    int tmp2;

    if(n == 0 || C == 0)
        result = 0;
    else if(w[n] > C) #volume do objeto > capacidade atual da mochila
```

```
        result = mochilaBinaria(n-1, C);
    else{
        tmp1 = mochilaBinaria(n-1, C) #caso obj n não seja incluído.
        tmp2 = v[n] + mochilaBinaria(n-1, C - w[n]) #caso seja incluído.
        result = max(tmp1, tmp2);
    }
    return result;
}
```

Para a análise de corretude, primeiro os casos-base retornam zero caso não haja itens a serem analisados (n=0), ou não haja mais espaço (C=0). Supondo que $mochilaBinaria(n, C)$ esteja correto, para $n+1$ o resultado virá de mochilaBinaria(n, C) caso w[n+1] > C, ou de $max(MochilaBin(n, C), v[n] + MochilaBin(n, C - w[n+1])$, caso contrário. Como chegamos na fórmula de recorrência, provamos a corretude do algoritmo, que apresenta uma árvore de recursão como no exemplo:

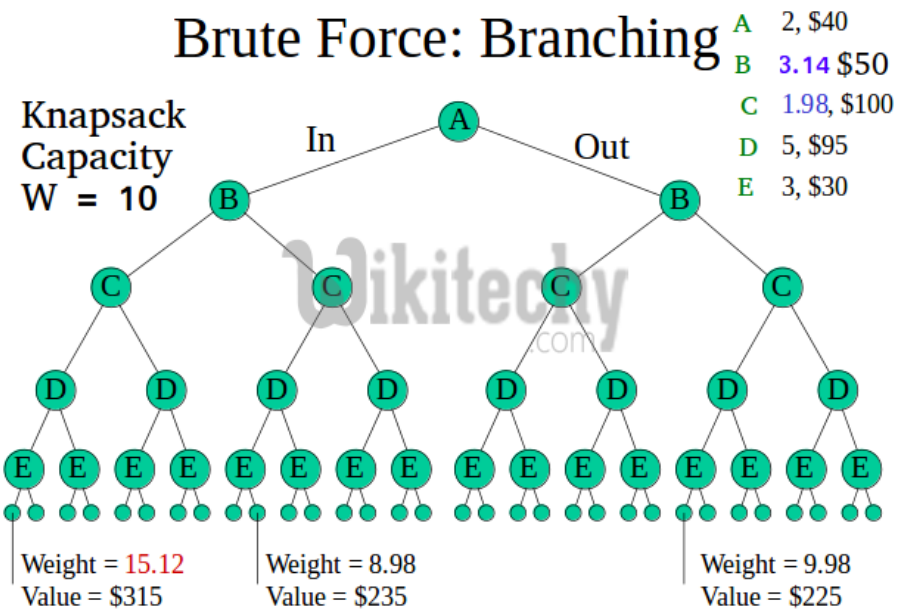

Figura 2: árvore de recursão do problema da mochila binária.

Aqui, pode-se perceber quantas vezes as subrotinas B, C, D e E são repetidas. Também é importante saber que uma solução de "força bruta"analisa todos os possíveis subconjuntos de itens, sendo que por haver um total de 2^n subconjuntos possíveis, a complexidade

de tal algoritmo também será $O(2^n)$, devido ao número de subproblemas a serem analisados, já que para resolver um subproblema levamos tempo constante $O(1)$. Novamente, com a PD, conseguiremos melhorar isto.

3.3 Solução Memoizada

A boa notícia é que conseguimos encontrar a solução ótima da estrutura maior se acharmos a solução ótima de suas subestruturas. Logo, nossa solução utilizando memoização fica:

```
# memo[n][C] retorna o custo ótimo da lista
# até o n-ésimo objeto, considerando
# capacidade C.
int memo[n][C] = indefinido
int mochilaBinaria(n, C){
    int tmp1;
    int tmp2;

    if(n == 0 || C == 0)
        result = 0;
    else if(w[n] > C) #volume do objeto > capacidade atual da mochila
        result = mochilaBinaria(n-1, C);
    else{
        tmp1 = mochilaBinaria(n-1, C) #caso obj n não seja incluído.
        tmp2 = v[n] + mochilaBinaria(n-1, C - w[n]) #caso seja incluído.
        result = max(tmp1, tmp2);
    }
    memo[n][C] = result;
    return result;
}
```

3.4 Solução Bottom-Up

A solução bottom-up pode ser vista abaixo:

```
int mochilaBinaria(n, C){
    for(k=0 to C)
        memo[0][k] = 0;
    for(i=1 to n){
        # k representa a capacidade atual, serão analisadas
        # todas as possibilidades.
        for(k=0 to C){
            if(w[i] < C) #cabe na mochila
                memo[i][k] = max(memo[i-1,k], v[i] + memo[i-1, k - w[i]);
            else #não cabe, passar para o próximo item.
                memo[i][k] = memo[i-1][k];
        }
    }
    return memo[n][C];
}
```

Onde fica fácil enxergar que, devido aos laços de repetição aninhados $i = 1, 2, ..., n$ e $k = 0, 1, 2, ..., C$, a nova complexidade cai de O(2^n) para apenas O(nC), sendo que este resultado se mantém para a abordagem recursiva de memoização (novamente chamando a atenção para as poucas mudanças que foram feitas, porém **significativas!**).

Ressaltando apenas que nesses algoritmos, o que é retornado é o **valor** do subconjunto ótimo de itens! Caso queira obter quais itens compõem o subconjunto, basta utilizar uma tabela booleana keep[i, C], onde keep[i, C] = 1 se o i-ésimo objeto foi colocado na mochila, e 0 caso contrário. Para construir um simples algoritmo que retorne os itens que foram colocados na mochila, basta começar checando se keep[i, C] == 1. Caso seja verdadeiro, repetimos o argumento para keep[i-1, C-w[i]]. Caso negativo, repetimos para keep[i-1, C]. Segue um trecho de código que permite recuperar os itens ótimos:

```
k = C;
for (i = n downto 1)
    if(keep[i][k] == 1){
        print(i);
        k = k - w[i];
    }
```

4 Aplicação: problema da subcadeia comum máxima (SCM)

4.1 Introdução e solução recursiva

Outro clássico problema com muitas aplicações na computação, como alinhamento de biossequências, linguistica computacional, entre outras. Considerando um alfabeto A = $\{a, b, c, ..., z\}$, podemos formar qualquer sequência X com elementos deste alfabeto (por exemplo: X = $adfxyz$). Além disso, vamos introduzir uma nova notação: considerando a cadeia do exemplo, denotaremos por X_1 = a, $X_2 = ad$, $X_3 = adf$, ..., $X_6 = adfxyz$. Para uma cadeia X de sequência, define-se como uma subsequência (ou subcadeia) de X uma sequência contida em X que mantenha uma mesma ordem, mas não seja necessariamente **contígua**. No exemplo anterior, a subcadeia **xyz** é uma subsequência de X, assim como **x**hjk**y**lmn**z** também é.

Dado duas cadeias de caracteres X e Y, queremos encontrar a subsequência **comum máxima** (ou LCS - Longest Common Subsequence). Para começar com uma solução recursiva simples, vamos começar analisando o último caractere das duas sequências X e Y. Daí, surgem os seguintes cenários:

- Se os últimos caracteres são iguais, devemos somar um ponto ao resultado e analisar agora SCM(X_{n-1}, Y_{m-1}).
- Se é diferente, pode ser que o último caractere de X_{n-1} seja igual ao último de Y, ou pode ser que o último de Y_{n-1} seja igual ao último de X. Neste caso, devemos analisar os dois subproblemas e ficar com o que apresentar maior pontuação: $max(SCM(X_n, Y_{m-1}), SCM(X_{n-1}, Y_m))$, sendo que o caso base é quando n==0 ou m==0 (fim de alguma, ou de ambas as cadeias). Segue o algoritmo recursivo abaixo:

```
int SCM(X, Y, n, m){
    int result = 0;
    if (n==0 || m==0)
        return 0;
```

```
        else if (X[n-1] == Q[m-1])
            result = 1 + SCM(X, Y, n-1, m-1);
        else if (X[n-1] != Q[m-1])
            result = max(SCM(X, Y, n-1, m), SCM(X, Y, n, m-1));
        return result;
    }
```

Para a análise de corretude, os casos-base são triviais. Supondo que $SCM(X, Y, n, m)$ esteja correto, para n-1, o resultado será ou $1+SCM(X, Y, n-2, m)$, ou $max(SCM(X, Y, n-2, m-1), SCM(X, Y, n-1, m-1))$, onde chegamos na equação de recorrência explicada anteriormente, portanto provando pela hipótese indutiva. Por simetria provamos a indução em m, e não é difícil provar para $m-1$ e $n-1$ simultaneamente. Com árvore de recursão abaixo:

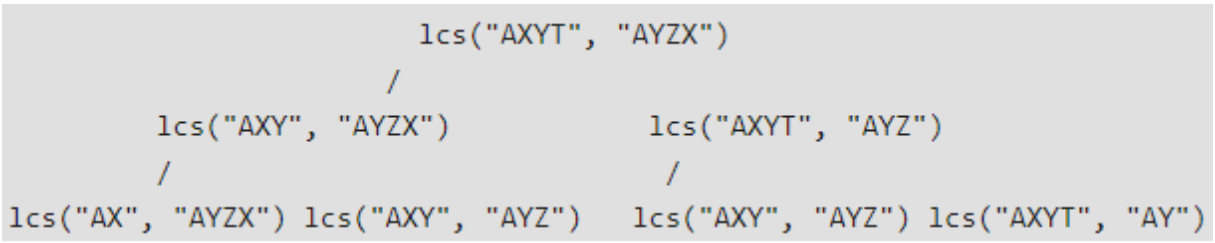

Figura 3:

Nesta árvore de recursão parcial, podemos observar lcs(“AXY”, “AYZ”) sendo resolvido duas vezes.

4.2 Solução memoizada

Para resolver por memoização, a mudança no algoritmo é simples: no código abaixo, memo[n][m] contém o resultado de $SCM(X_n, Y_m)$:

```
    int memo[n][m] = indefinido
    int SCM(X, Y, n, m){
        int result = 0;
        if (memo[n][m] != indefinido)
```

```
        return memo[n][m];
    if (n==0 || m==0)
        return 0;
    else if (X[n-1] == Q[m-1])
        result = 1 + SCM(X, Y, n-1, m-1);
    else if (X[n-1] != Q[m-1])
        result = max(SCM(X, Y, n-1, m), SCM(X, Y, n, m-1));
    memo[n][m] = result;
    return result;
}
```

4.3 Solução Bottom-Up

```
int SCM( X, Y, m, n )
{
    int L[m + 1][n + 1];
    int i, j;

    # Os passos seguintes controem L[m+1][n+1] em bottom-up.
    #Lembre-se que L[i][j] contém o tamanho de SCM de
    #X[0..i-1] e Y[0..j-1]
    for (i = 0; i <= m; i++)
    {
        for (j = 0; j <= n; j++)
        {
            if (i == 0 || j == 0)
                L[i][j] = 0;
            else if (X[i - 1] == Y[j - 1])
                L[i][j] = L[i - 1][j - 1] + 1;
            else
```

```
                L[i][j] = max(L[i - 1][j], L[i][j - 1]);
        }
    }

    # a partir daqui é para imprimir a subcadeia
    # comum máxima.

    # começamos do canto direito inferior da tabela e
    # imprimindo os caracteres que pertencem a subcadeia
    # comum máxima.

i = n, j = m;
while (i > 0 && j > 0)
{

    if (X[i-1] == Y[j-1]) #se são iguais, imprime.
    {
        print(X[i-1]);
        i--;
        j--;
    }

    # se não são iguais, ir na direção do SCM de
    # maior valor (ou L[i-1][j], ou L[i][j-1])
    else if (L[i-1][j] > L[i][j-1])
        i--;
    else
        j--;
}
```

```
        # L[m][n] contém o SCM de X e Y
        # para X[0..n-1] e Y[0..m-1]
        return L[m][n];
    }
```

Cada operação dentro do laço de repetição mais interno tem custo constante O(1), então o custo do algoritmo utilizando PD será O(mn) devido ao aninhamento dos laços de repetição (lembrando que a complexidade de tempo é a mesma tanto bottom-up quanto por memoização), o que é uma significativa melhor em relação à estratégia gulosa, já que há um total de 2^n possibilidades para analisar as subcadeias, e portanto complexidade O(2^n).

Se desejarmos recuperar a subcadeia comum máxima, devemos percorrer toda a tabela L, começando por L[m][n] até L[0][0]. A partir disso, é simples (como mostrado no código):

- Se X[i] == Y[j] correspondente a L[i][j], então o caractere pertence à cadeia.
- caso X[i] != Y[j], devemos percorrer a tabela na direção do maior valor entre L[i-1][j] e L[i][j-1].

Referências Bibliográficas

Bellman, R. (1957). Dynamic programming: Princeton univ. press. *NJ*, 95.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., e Stein, C. (2001). *Introduction to Algorithms*. The MIT Press, 2 edition.

Rust, J. (2008). Dynamic programming. *The New Palgrave Dictionary of Economics*, 1:8.

Von Neumann, J., Morgenstern, O., e Kuhn, H. W. (2007). *Theory of games and economic behavior (commemorative edition)*. Princeton university press.