Programação Dinâmica
Trabalho de conclusão de curso
Monografia

Stefano Tommasini
Nº USP: 7278080
Supervisor:
Carlos Eduardo Ferreira

Agradecimentos

Agradeço em especial ao meu supervisor pela sugestão do tema, reuniões, críticas e comentários.

Agradeço também a minha família pela ajuda com o texto.

Também um agradecimento a meus amigos Renzo Gomes Diaz e Marcio Oshiro pela revisão de texto.

Resumo

A falta de um bom material didático em português sobre programação dinâmica e minha experiência em competições de programação foi o que motivou o desenvolvimento desse trabalho. O objetivo do trabalho é facilitar o aprendizado do tópico. O presente trabalho desenvolve um material didático sobre programação dinâmica. Inicio pela abordagem dos conceitos fundamentais utilizados na área: funções recursivas, estados, memorização entre outros. Em seguida descrevo múltiplos exemplos onde a técnica é fundamental, e o trabalho é finalizado com exercícios para servir de apoio ao leitor.

Sumário

1 Introdução

Projeto e análise de algoritmos é uma das principais áreas da computação. Além da corretude e eficiência, qualidades fundamentais de programas bem escritos, estes devem ser também elegantes e belos. Não é a toa que o livro fundamental da área, escrito ainda nos anos 70 por Donald Knuth chama-se "The **ART** of Computer Programming". Escrever algoritmos corretos, eficientes, elegantes e bonitos é, antes de tudo, uma arte.

De todas as técnicas de programação nenhuma se aproxima tanto da arte como a chamada "programação dinâmica". Esta técnica passou a ser conhecida por este nome em 1957 quando Richard Bellman publicou o livro "Dynamic Programming" em que apresentava a técnica e dezenas de problemas que se resolviam com seu uso.

O livo de Bellman mereceu, inclusive, a atenção de Knuth em "The Art of Computer Programming": "The book "Dynamic Programming" by Richard Bellman is an important, pioneering work in which a group of problems is collected together at the end of some chapters under the heading 'Exercises and Research Problems', with extremely trivial questions appearing in the midst of deep, unsolved problems. It is rumored that someone once asked Dr. Bellman how to tell the exercises apart from the research problems, and he replied: 'If you can solve it, it is an exercise; otherwise it's a research problem.'"

O grande Thomas Cormen foi perguntado qual o capítulo mais díficil que ele escreveu para o seu livro, sua resposta foi: "When we were writing the first edition, back in the late 1980s, by the time we got to the dynamic programming chapter, we knew that we had to plan it out carefully before writing it. So that's exactly what we did. We planned it out very carefully. I wrote the first draft, exactly according to the specification that we had agreed upon. I was happy with it. But Charles Leiserson and Ron Rivest were not. They agreed that I had written the chapter to spec. But they felt that it just did not work. We asked a graduate student in the Theory Group at MIT to read the draft and tell us what he thought. He agreed with Charles and Ron. "

Ao mesmo tempo que se apresenta bela, a programação dinâmica se mostra desafiadora. Encontrar a recorrência e projetar o algoritmo que a resolve eficientemente pode se mostrar muito difícil. Isso é facilmente notado nas disciplinas de Algoritmos e Estrutura de Dados do curso de BCC, em que o tópico é, via de regra, temido pelos estudantes.

Também em competições de programação, como a OBI, ACM-ICPC, Topcoder, codeforces, codechef, google codejam, etc, problemas que podem ser resolvidos com o uso da técnica são usualmente classificados como complexos, e estão entre os que são resolvidos por menos competidores.

O objetivo desse trabalho é criar um material didático que apresenta, através de vários exemplos, a técnica de programação dinâmica. Nosso principal objetivo é desenvolver um

material que sirva de base para estudantes e competidores das olimpíadas aprenderem programação dinâmica e serem capazes de resolver problemas complexos de forma eficiente, elegante e bonita.

2 Online Judges

Um juiz online é um site com problemas de otimização onde qualquer usuário pode submeter seu código e o juiz vai dizer se está correto. O primeiro passo é criar uma conta no juiz. Para cada submissão, o juiz pode dar uma das seguintes respostas.

- Accepted - Seu código está correto.
- Wrong answer - Seu código responde algum caso de teste incorretamente.
- Time limit Exceeded - Seu código demora demais para executar (em geral ele tem 2-3 segundos para fazer isso).
- Memory Limit Exceeded - Seu código gasta memória demais (em geral ele pode gastar aproximadamente 256MB)

Durante esse trabalho serão listados muitos problemas que podem ser submetidos em juizes online..

3 Funções Recursivas

Uma função recursiva $f(x)$ é uma função cuja definição envolve, de alguma forma, a si mesma. O conjunto de instâncias em que a função assume valores definidos é chamado de base da recursão. Para os cálculos da função em valores fora da base a própria função é chamada. Vamos analisar um exemplo.

Sequência de Fibonacci: É uma sequência de números inteiros, começando por 1 e 1, na qual, cada termo subsequente corresponde à soma dos dois anteriores. A sequência recebeu o nome do matemático italiano Leonardo de Pisa, mais conhecido por Fibonacci (contração do italiano filius Bonacci), que descreveu, no ano de 1202, o crescimento de uma população de coelhos. No modelo de Fibonacci um casal de coelhos demorava um período para se tornar fértil, e ele supunha que a partir de então o casal tinha um casal de filhos em todos os períodos. Assim, começando com 1 casal teríamos 1 no seguinte período, 2 no terceiro (o primeiro casal está fértil), 3 no quarto (novamente o primeiro casal está fértil), 5 no quinto (os casais do terceiro período estão férteis) e assim por diante. Tal sequência já era no entanto, conhecida na antiguidade [Wikipédia.org]. Os primeiros termos da sequência são:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \cdots \tag{1}$$

ou, recursivamente, se chamássemos fib(i) o i-ésimo termo da sequência de *Fibonacci*, temos:

$$\begin{aligned} \text{fib}(0) &= 1 \\ \text{fib}(1) &= 1 \\ \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2), n \geq 2 \end{aligned}$$

A Figura 1 ilustra como a função fib(5) é calculada recursivamente.

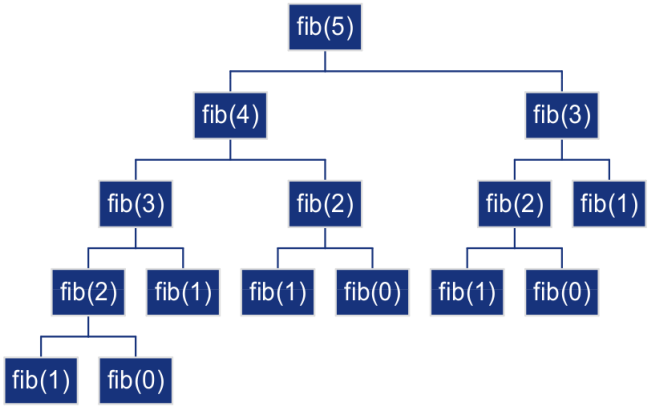


Figura 1: Fibonacci(5) [http://www.dcc.fc.up.pt/ pribeiro/pd-pribeiro.pdf]

Numa função recursiva em geral teremos uma relação desse tipo, com um ou mais parâmetros. Em outras palavras, a solução para uma instância do problema pode ser decomposta em soluções para instâncias menores. Veremos a seguir, por meio de mais exemplos, como as funções recursivas podem ser utilizadas na resolução de problemas. Seguem os casos do fatorial para números inteiros e a função exponencial na base 2. Na função $fat(x) = x!$, definida para números inteiros, temos que:

$$\begin{aligned} fat(x) &= x * (x-1) * (x-2) * \cdots * 2 * 1. \\ fat(x) &= x * (x-1)! \end{aligned}$$

que pode ser escrita recursivamente como:

$$\begin{aligned} fat(1) &= 1 \\ fat(x) &= x * fat(x-1), x \geq 2. \end{aligned}$$

Já para a função exponencial, $exp(x) = 2^x$, temos:

$$exp(x) = 2 * 2^{x-1}$$

que pode ser escrita recursivamente como

$$\begin{aligned} exp(0) &= 1 \\ exp(x) &= 2 * exp(x-1), x \geq 1 \end{aligned}$$

Para utilizarmos funções recursivas é conveniente definirmos o conceito de **estado**. Entende-se por estado um conjunto de parâmetros de uma função recursiva. Cada conjunto de parâmetros determina unicamente um estado. Vamos ilustrar a definição de estado por

meio de 2 exemplos.

Exemplo 1: Vamos supor que queremos encontrar o maior elemento em um vetor $v[0, \cdots, n-1]$, um problema muito comum em Computação. O problema é trivial, pois basta percorrer o vetor procurando o máximo. Vamos mostrar como podemos resolvê-lo recursivamente.

Se chamarmos de $vmax(k)$ o maior elemento do vetor no intervalo $v[k, \cdots, n-1]$, então a resposta para o problema do *Exemplo* 1 seria a função calculada para $k = 0$, ou seja $vmax(0)$. Dizemos que um estado é um valor $k \in \{0, 1, \cdots, n-1\}$ para o qual a função $vmax(k)$ pode ser calculada. A função recursiva pode ser definida:

$$\begin{aligned} vmax(n-1) &= v[n-1] \\ vmax(x) &= max(v[x], vmax(x+1)), 0 \leq x < n-1 \end{aligned}$$

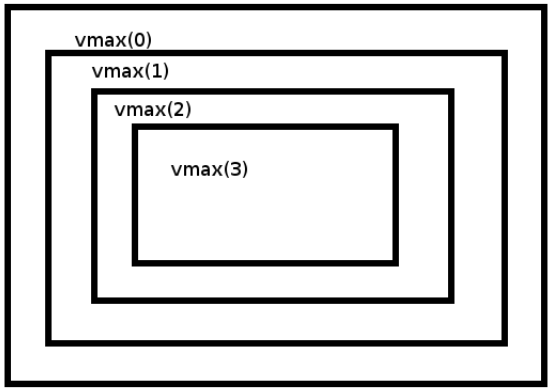


Figura 2: Cálculo de $vmax$ para $n = 4$

A Figura 2 corresponde a um diagrama que simula a execução do cálculo de $vmax$ num vetor de tamanho 4. Veja que $vmax(0)$ chama $vmax(1)$ até chegar em $vmax(3)$ que é a base. Cada estado é calculado uma única vez.

Exemplo 2: Torres de Hanói:

Torre de Hanói é um "quebra-cabeça" que consiste numa base contendo três pinos, num dos quais são dispostos n discos uns sobre os outros, de forma que os discos de diâmetro menor são dispostos sobre os de diâmetro maior. O problema consiste em passar todos

os discos de um pino para outro qualquer, podendo usar um dos pinos como auxiliar, de maneira que um disco maior nunca fique em cima de outro menor em nenhuma situação, e apenas um disco seja movido de cada vez [wikipedia.org].

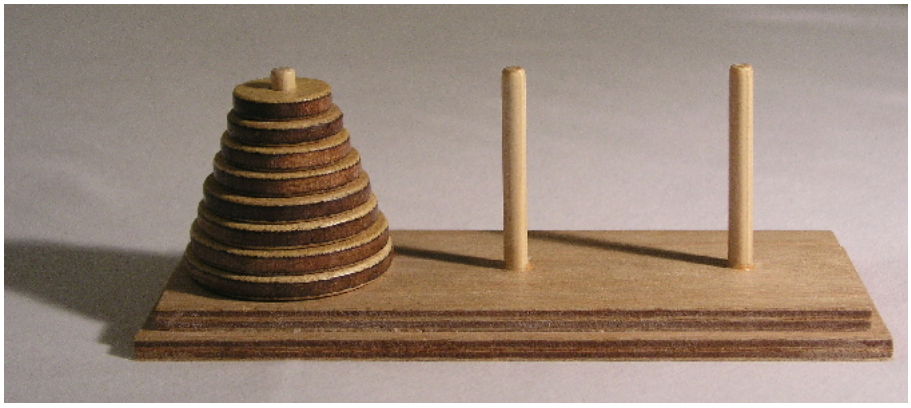


Figura 3: Torre de Hanói [Wikipédia.org]

Neste caso o estado será o número de discos da instância, ou seja, n. Podemos definir uma função $H(n)$ que representa o número mínimo de movimentos de discos necessários para resolver o problema. A base da recursão se dá para o estado $n = 1$: se temos um disco, podemos movê-lo para a torre desejada resolvendo o problema.

Agora considere uma instância com $n > 1$ discos. Podemos resolver em 3 etapas:

1. Mover $n - 1$ discos do pino 1 para o pino 2, que custa $H(n - 1)$ movimentos.
2. Mover o disco restante do pino 1 para o pino 3, que custa 1 movimento.
3. Mover os $n - 1$ discos do pino 2 para o pino 3, que custa $H(n - 1)$ movimentos.

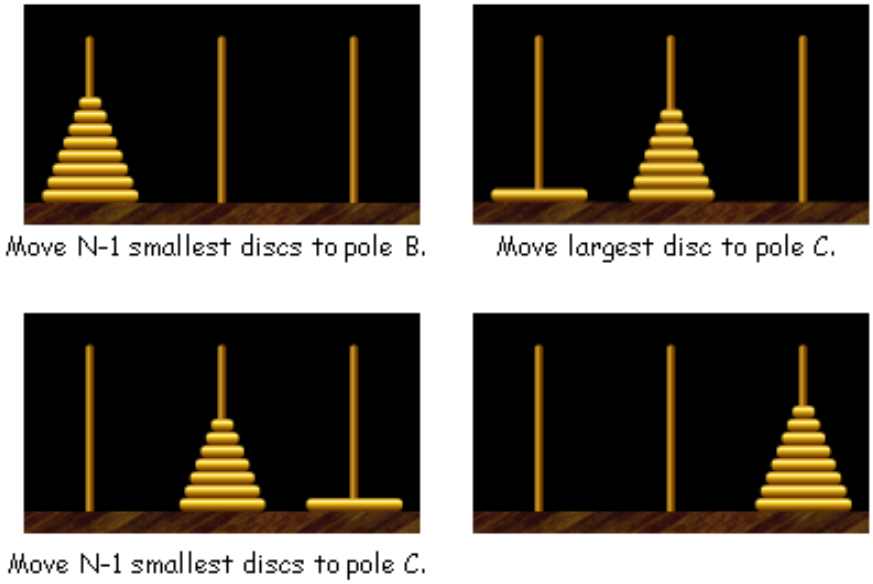


Figura 4: Ilustração dos 3 passos citados acima. [comscgate.com]

Logo segue a recursão:

$$\begin{aligned} H(1) &= 1 \\ H(n) &= H(n-1) + 1 + H(n-1) = 2 * H(n-1) + 1, n > 1 \end{aligned}$$

E a árvore de recorrência:

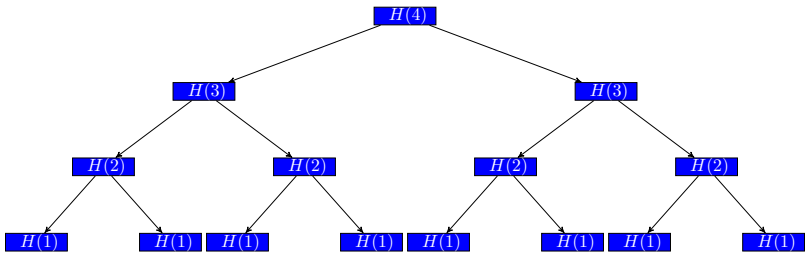


Figura 5: Árvore de Recorrência para $H(4)$

[1] **Exemplo 3:** Função de Ackermann:

Funções recursivas podem ter mais de um parâmetro. Um exemplo para isso é a função de Ackermann, conhecida desde os anos 20. Seu inventor, Wilhelm Ackermann era aluno do

[1] wikipedia.org

matemático Hilbert e pesquisava fundamentos teóricos da Computação. A função proposta é um exemplo de função computável que não é recursiva primitiva. Segue abaixo a função de Ackermann definida para 2 parâmetros m e n:

$$\begin{aligned} A(m,n) &= n+1, \text{ se } m = 0 \\ A(m,n) &= A(m-1,1), \text{ se } m > 0 \text{ e } n = 0 \\ A(m,n) &= A(m-1, A(m,n-1)), \text{ se } m > 0 \text{ e } n > 0 \end{aligned}$$

4 Implementando problemas recursivos

Nesta seção explicitaremos os pseudocódigos para os casos estudados até aqui.

Algoritmo FIBONACCI(x)

```
1 //aqui a base
2 se x = 0 ou x = 1 então
3       devolva 1
4 devolva FIBONACCI (x − 1)+ FIBONACCI (x − 2)
```

Algoritmo POTENCIADEDOIS(x)

```
1 se x = 0 então
2       devolva 1
3 devolva 2×POTENCIADEDOIS(x − 1)
```

Algoritmo H(n)

```
1 se n = 1 então
2       devolva 1
3 devolva 2×H(n − 1) + 1
```

Algoritmo VMAX(v, i, n)

```
1 se i = n − 1 então
2       devolva v[i]
3 devolva max(v[i],VMAX (v, i + 1, n))
```

Algoritmo A(m, n)

```
1 se m = 0 então
2       devolva n + 1
3 se n = 0 então
4       devolva A(m − 1, 1)
5 devolva A(m − 1,A(m, n − 1))
```

Nem sempre a implementação recursiva direta, como na definição de função, dá origem à forma mais eficiente de resolver o problema. Vamos analisar, por exemplo, a implementação acima para calcular fib(n). Retornamos ao cálculo do quinto elemento da sequência

de Fibonacci, ilustrado na Figura 6. Note que vários estados são recalculados: fib(2) é calculado 3 vezes, por exemplo.

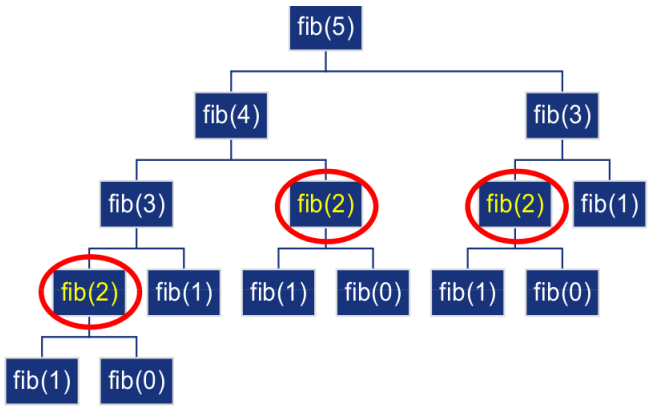


Figura 6: Fibonacci(5) [http://www.dcc.fc.up.pt/ pribeiro/pd-pribeiro.pdf]

Mais formalmente, se chamarmos de $T(n)$ o número de operações feitas no calculo de fib(n), temos:

$$\begin{aligned} T(0) &= 1 \\ T(1) &= 1 \\ T(n) &= T(n-1) + T(n-2) + 1, n \geq 2 \end{aligned} \tag{2}$$

Assim podemos observar que $T(n) \geq fib(n)$. Outra fórmula conhecida para $fib(n) = \frac{1}{\sqrt{5}}\{(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n\}$ [2], que por sua vez cresce exponencialmente, logo $T(n)$ é exponencial em relação a n. Nas próximas seções mostraremos como podemos implementar funções recursivas eficientemente.

Exercícios [3]

Exercício 1 Escreva uma função recursiva que calcule a soma dos elementos positivos do vetor $v[ini..fim-1]$.

Exercício 2 Escreva uma função recursiva que dado um vetor $v[ini..fim-1]$ ordenado crescentemente e um elemento x, retorna $true$ se x está em $v[ini..fim-1]$ e $false$ caso contrário.

[2] wikipedia.org

[3] Exercícios adaptados da página do professor Paulo Feofiloff.

Exercício 3 Utilize a técnica de busca binária para resolver o **Exercício 2** em tempo $O(log(fim - ini))$.

Exercício 4 Escreva uma função recursiva que recebe um inteiro n, representando o número de discos de uma instância do problema $Hanoi$. Faça uma função recursiva que imprima os movimentos que resolvem o problema.

Exercício 5 Escreva uma função recursiva que calcula $C(i, j)$, onde $C(i, j)$ é o número de jeitos de escolher i dentre j com $i \leq j$, também conhecido como Binômio de Newton.

4.1 Memorização

Os pseudocódigos apresentados acima não correspondem, em todos os casos, às melhores implementações conhecidas para se calcular as funções recursivas. Uma análise rigorosa da execução do algoritmo Fibonacci, mostrado na Figura 6, mostra que no cálculo de $fib(5)$, um mesmo estado pode ser calculado várias vezes podendo tornar o algoritmo ineficiente. Esse problema pode ser contornado memorizando o valor da solução do problema para cada estado, pois terá o mesmo valor toda vez que é calculado. Poderíamos então memorizá-lo após o seu primeiro cálculo e se precisarmos desse valor novamente ele já estará armazenado. O armazenamento pode ser feito por meio de uma matriz. A sua utilização para o caso da função de Fibonacci pode ser vista no pseudocódigo abaixo.

Algoritmo FIBONACCI(x)

```
1 se x = 0 ou x = 1 então
2       devolva 1
3 se memoriza[x] não foi visto então
4       memoriza[x] ←FIBONACCI(x − 1)+FIBONACCI(x − 2)
5 devolva memoriza[x]
```

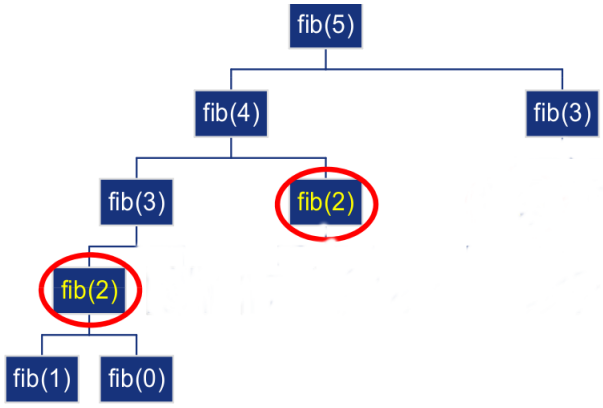


Figura 7: Fibonacci(5) com memorização [http://www.dcc.fc.up.pt/ pribeiro/pd-pribeiro.pdf]

Veja como $fib(5)$ é calculado com memorização, perceba que com a memorização a árvore é muito menor, deixando o algoritmo muito mais rápido. Na próxima seção vamos mostrar como calcular o tempo de execução de uma função recursiva com memorização.

4.2 Analise da complexidade de funções recursivas com memorização

Como foi visto anteriormente, a memorização garante que cada estado seja calculado uma única vez. A seguir definiremos formalmente a complexidade neste caso. Seja E o conjunto de estados, a complexidade de um algoritmo é:

$$\sum_{e \in E} f(e), \tag{3}$$

onde $f(e)$ é o custo para calcular o estado e.

Tome o algoritmo com memorização para Fibonacci, $f(e)$ é $O(1)$ para todo e em E, logo a complexidade do algoritmo é o número de estados. Para calcular fib(n) o custo é $O(n)$, o algoritmo é linear em n. Em vários outros casos de funções recursivas, memorizar o valor da solução nos estados implica numa redução de complexidade dos algoritmos recursivos que calcula a tal. Para toda função recursiva em que a memorização dos estados implica numa redução de complexidade, essa memorização é o que chamamos de **Programação Dinâmica**.

Nem toda função recursiva melhora ao colocarmos a memorização. Tome a função $vmax$, com a execução simulada na Figura 2. Veja que cada estado é calculado uma única vez, nesse caso a memorização não melhora o algoritmo, pois memorizamos uma resposta que nunca mais vamos consultar. Vamos mostrar quando vale a pena aplicar essa técnica.

5 Programação Dinâmica

Programação Dinâmica é uma técnica muito importante para a resolução de problemas. É aplicada a problemas onde escolhas têm de ser feitas para chegar em uma solução ótima e essas escolhas geram novos subproblemas da mesma forma. Ela consiste em quebrar um problema grande em subproblemas menores que se sobrepõem, e obedecem a propriedade de subestrutura ótima. Um problema apresenta uma subestrutura ótima quando uma solução ótima para o problema contém em seu interior soluções ótimas para subproblemas. A superposição de subproblemas acontece quando um determinado subproblema ocorre diversas vezes na solução de um problema. Já vimos quando analisamos o algoritmo que calcula o n-ésimo termo da série de Fibonacci. O valor de FIBONACCI(5) é composto pela soma do valor para duas instâncias menores 4 e 3 (subestrutura ótima). Além disso, a solução calcularia o valor de FIBONACCI(2) várias vezes (superposição de subproblemas). A técnica usada em programação dinâmica para evitar o recálculo é a memorização, apresentada na seção anterior. Ao desenvolver um algoritmo de programação dinâmica, podemos seguir 3 etapas:

1. Caracterizar a estrutura de uma solução ótima.
2. Recursivamente definir o valor de uma solução ótima.
3. Computar o valor de uma solução ótima utilizando memorização.

Em outras palavras, precisamos verificar se o problema pode ser decomposto em subproblemas menores do mesmo tipo e se a solução ótima do problema é uma combinação das soluções desses subproblemas. Vamos mostrar no restante desse trabalho diversos exemplos da aplicação da técnica.

5.1 Problema da Mochila

[4] **Descrição.** Stefano é um ladrão e ele leva consigo uma mochila de capacidade C. Na loja que pretende roubar existem n objetos numerados de 0 até $n-1$. Cada objeto i tem peso $W[i]$ e valor $V[i]$. Stefano quer saber qual é o valor máximo que ele pode roubar de tal forma que a soma dos pesos dos objetos roubados não ultrapasse a capacidade de sua mochila.

[4] wikipedia.org

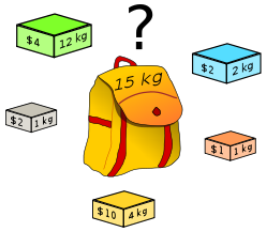


Figura 8: [wikipedia.org]

Curiosidades O problema da mochila vem sendo estudado desde 1897. Foi referenciado pela primeira vez pelo matemático Tobias Dantzig (1884–1956). Em 1998 o estudo no Stony Brook University Algorithm Repository mostrou que de 75 algoritmos estudados, o problema da mochila é o quarto mais importante e o 18^o mais popular.

Exemplo 3.1.1 A título de ilustração, tomemos uma mochila com capacidade $C = 3$ e 3 objetos numerados $\{0, 1, 2\}$, cujos pesos são $\{3, 1, 2\}$ e valores $\{5, 3, 3\}$ respectivamente. Seguem todas as possíveis combinações de itens cuja soma dos pesos é menor ou igual a $C = 3$.

Objetos escolhidos :

$\{0\} \rightarrow$ valor total $= 5$, peso total $= 3$

$\{1\} \rightarrow$ valor total $= 3$, peso total $= 1$

$\{2\} \rightarrow$ valor total $= 3$, peso total $= 2$

$\{1, 2\} \rightarrow$ valor total $= 6$, peso total $= 3$

Dentre todas as possibilidades, por inspeção, podemos ver que a solução ótima é o subconjunto$\{1, 2\}$, onde a soma dos valores é $3 + 3 = 6$

Caracterizando a subestrutura ótima. Tome uma instância (n, C, W, V). Pegue um objeto i qualquer, podemos separar 2 casos distintos para esse objeto.

1) Objeto i pertence a algum subconjunto ótimo. Nesse caso o valor da solução ótima do problema é $V[i]$ + valor da solução ótima para o restante dos objetos e agora com uma capacidade C-$W[i]$.

2) Objeto i não pertence a algum subconjunto ótimo. Nesse caso a solução ótima do problema é simplesmente a solução de um novo problema sem o item i.

Podemos ver que o problema obedece a subestrutura ótima, ou seja, pode ser resolvido por uma combinação de subproblemas menores.

Definindo a função recursiva. Dado que estabelecemos que o problema obedece a

propriedade da subestrutura ótima vamos definir a função recursiva que o resolve. Seja $Mochila$ essa função. Vamos supor que os objetos estejam em uma ordem qualquer, numerados de 0 até $n-1$. Neste caso necessitaremos de estados com 2 parâmetros, $objetoAtual$ e $weight$, onde $Mochila(objetoAtual, weight)$ é o maior valor que se pode obter com os objetos de $objetoAtual$ até $n-1$ com $weight$ de capacidade ainda disponível na mochila. A base é quando não há mais objetos a serem vistos, ou seja $objetoAtual = n$, nesse caso $Mochila(n, weight) = 0$. Então quando calculamos $Mochila(objetoAtual, weight)$, a ideia é olhar para o objeto $objetoAtual$, tentando colocá-lo ou não na mochila e escolhendo assim a melhor das duas soluções. A resposta do problema será o valor da função $Mochila(0, C)$.

Computando o valor da solução ótima com memorização

Algoritmo MOCHILA($objetoAtual, weight$)

```
1  se objetoAtual = n então
2      devolva 0
3  se memoriza[objetoAtual][weight] já foi calculado então
4      devolva memoriza[objetoAtual][weight]
5  r1 = infinito, r2 = infinito
6  //Tentar colocar o objeto de índice objetoAtual na mochila se ainda tiver espaço para
   ele.
7  se W[objetoAtual] ≤ weight então
8      r1 = V[objetoAtual]+MOCHILA(objetoAtual + 1, weight − W[objetoAtual]);
9  //Agora calcular a melhor resposta sem usar objetoAtual, simplesmente ignorando-o:
10 r2 =MOCHILA(objetoAtual + 1, weight)
11 resposta = max(r1, r2)
12 memoriza[objetoAtual][weight] = resposta
13 devolva resposta
```

Simulação Segue a árvore de recorrência para o cálculo da função $Mochila$ para o **Exemplo** 3.1.1. O filho da esquerda representa pegar o objeto e o filho da direita não pegar.

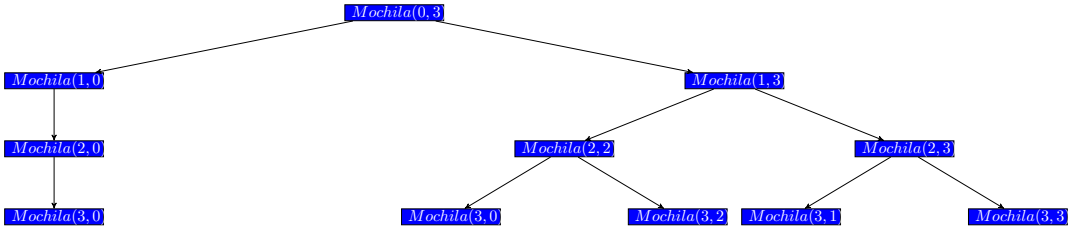


Figura 9:

Complexidade. Agora vamos analisar a complexidade. Seja E o conjunto de estados, para todo e em E, o custo para calcular a resposta é O(1) pois envolve um número constante de operações. Logo, basta contar a quantidade de estados. O parâmetro *objetoAtual* pode ter valor de 0 até n e *weight* pode variar de 0 até C, assim o número de estados é $O(nC)$.

Problemas relacionados:

- http://www.spoj.com/problems/BACKPACK/

5.2 Problema das Moedas

Descrição. Carlinhos é vendedor e precisa dar troco de S centavos para seu cliente Stefano. Carlinhos tem n tipos de moedas com valores diferentes, sendo esses valores dados num vetor V. Tem, também, infinitas moedas de cada um dos n tipos e quer saber de quantas formas diferentes pode dar esse troco.

Exemplo 3.2.1: Tome $n = 4$, moedas de valores $V = (50, 25, 15, 10)$, e $S = 50$. Temos as seguintes diferentes formas de dar o troco: (50), $(25, 25)$, $(25, 15, 10)$, $(15, 15, 10, 10)$, $(10, 10, 10, 10, 10)$. Note que $(15, 15, 10, 10)$ e $(15, 10, 15, 10)$ são equivalentes.

Caracterizando a subestrutura ótima Tome uma instância (n, V, S). Vamos analisar o que pode acontecer com uma dada moeda de índice i. Ela poderá ser usada de 0 até $\lfloor \frac{S}{V[i]} \rfloor$ vezes na solução e para cada uma dessas possibilidades precisamos contar de quantas formas podemos formar o restante do troco. Ou seja, para cada k em $[0, \lfloor \frac{S}{V[i]} \rfloor]$ temos que resolver um subproblema com $S' = S - k * V[i]$ podendo usar todas as moedas menos a de índice i. Logo o problema obedece a propriedade da subestrutura ótima.

Definindo a função recursiva Seja $Moeda(moedaAtual, D)$, uma função que calcula o número de conjuntos de moedas tal que a soma seja igual a D utilizando as moedas de $[moedaAtual...n - 1]$. Aqui temos um problema parecido com o da mochila, porém não queremos maximizar algo e sim fazer uma contagem. Da mesma forma que no problema anterior, podemos definir um estado com parâmetros $moedaAtual$ e D, tal que $Mochila(moedaAtual, D)$ é quantos subconjuntos de soma D existem usando as moedas de $[moedaAtual...n-1]$. Agora para montar a recursão, temos que considerar a variação no fato que podemos usar um número variável de cada moeda, assim tentamos colocar todas as quantidades possíveis de moedas, ou seja, se temos D e a moeda tem valor c, podemos colocar de 0 a $\lfloor \frac{D}{c} \rfloor$. A base da recursão é quando $D = 0$, ou seja, já conseguimos somar o valor desejado, e retornamos 1, ou quando $moedaAtual = n$, ou seja, passamos por todas as moedas e não conseguimos somar o valor desejado, assim retornamos 0.

Computando o valor da solução ótima com memorização

Algoritmo MOEDA($moedaAtual, D$)

```
1  se D = 0 então
2      devolva 1
3  se moedaAtual = n então
4      devolva 0
5  se estado (moedaAtual, D) já foi calculado então
6      devolva memoriza[moedaAtual][D]
7  vezes ← 0, resposta ← 0
8  enquanto vezes * V[moedaAtual] ≤ D faça
9      resposta ← resposta+MOEDA(moedaAtual + 1, D − vezes * V[moedaAtual])
10     vezes = vezes + 1
11 devolva memoriza[moedaAtual][D] = resposta
```

Complexidade. Agora vamos analisar a complexidade desse algoritmo. Primeiramente vamos ver quantos estados diferentes existem: $moedaAtual$ varia de 0 a n e D varia de 0 a S, tendo assim $O(nS)$ estados. O custo de cada estado é um loop dado na linha 8 do algoritmo, e esse loop pode no pior caso variar de 0 a S, sendo então o algoritmo $O(nS^2)$.

Melhoramento do algoritmo Será que não podemos fazer melhor? O loop da linha 8 não é necessário, é possível deixar isso por conta da recursão. Ou seja, ao pegarmos a moeda chamamos a função com a mesma $moedaAtual$ e com um D menor, possibilitando escolher a moeda novamente, ou definimos que não pegaremos mais essa moeda e chamamos a função com $moedaAtual + 1$. Segue em código:

Algoritmo MOEDA2($moedaAtual, D$)

```
1  se D = 0 então
2      devolva 1
3  se moedaAtual = n então
4      devolva 0
5  se estado (moedaAtual, D) já tiver sido calculado então
6      devolva memoriza[moedaAtual][D]
7  se V[moedaAtual] ≤ D então
8      respostaUsando = Moeda2(moedaAtual, D − V[moedaAtual])
9  respostaSemUsar = Moeda2(moedaAtual + 1, D)
10 resposta = respostaUsando + respostaSemUsar
11 devolva memoriza[moedaAtual][D] = resposta
```

Tem-se agora o mesmo número de estados e cada estado tem custo $O(1)$, deixando o algoritmo $O(nS)$.

Problemas relacionados:

- http://www.spoj.com/problems/PIGBANK/

5.3 Parentização de matrizes

Descrição. Dadas n matrizes $A_0, A_1, \cdots, A_{n-1}$, onde a matriz i tem $c[i]$ colunas e $l[i]$ linhas, imagine o processo de calcular $A_0 \times A_1 \times \cdots \times A_{n-1}$. Determine, através da colocação de parênteses, a ordem em que essas multiplicações devem ser realizadas de forma que o número de multiplicações escalares seja mínima. Vamos considerar que o número de multiplicações escalares realizadas para multiplicar uma matriz A_i por A_j é dada por $l[i] \times c[i] \times c[j]$.

Exemplo 1: Tome o exemplo para $n = 4$. As possíveis parentizações são:

$$\begin{array}{l} (A_0(A_1(A_2A_3))) \\ (A_0((A_1A_2)A_3)) \\ ((A_0A_1)(A_2A_3)) \\ ((A_0(A_1A_2))A_3) \\ (((A_0A_1)A_2)A_3) \end{array} \tag{4}$$

Exemplo 2: Tome $n = 3$ e as matrizes A_0, A_1, A_2 onde suas dimensões são: 10x100, 100x5, 5x50, respectivamente. Se multiplicarmos de acordo com a parentização $((A_0A_1)A_2)$, vamos realizar $10 * 100 * 50 = 5000$ multiplicações escalares para calcular $A_0 * A_1$ e mais $10 * 5 * 50 = 2500$ para multiplicar essa matriz por A_3, totalizando 7500 multiplicações escalares. Se realizarmos de acordo com a parentização $(A_0(A_1A_2))$ realizaremos $100*5*50 = 25000$ multiplicações escalares para calcular A_1*A_2 e mais $10*100*50 = 50000$ para multiplicar A_1 por essa matriz, totalizando 75000 multiplicações escalares, 10 vezes mais do que a primeira parentização.

Caracterizando a subestrutura ótima. Vamos chamar de $A_{i,j}$ a matriz resultante da multiplicação $A_i * A_{i+1} * \cdots * A_j$. Observe que essa matriz tem $l[i]$ linhas e $c[j]$ colunas. Vamos encarar o problema de trás para frente. Veja o que acontece na última multiplicação realizada: ela será do tipo $A_{0,k} * A_{k+1,n-1}$, para algum k inteiro onde $0 \leq k < n - 1$. Para calcular $A_{0,k}$ e $A_{k+1,n-1}$ a mesma propriedade é válida, ou seja, ambas devem ser calculadas com um número mínimo de multiplicações escalares, portanto o problema obedece a propriedade da subestrutura ótima.

Definindo a função recursiva Para cada matriz $A_{i,j}$ podemos definir uma função recursiva $MM(i, j)$ que calcula o menor número de operações para formar $A_{i,j}$. Basta

testar para todo k inteiro onde $i \leq k \leq j-1$.

$$\begin{gathered} MM(i,j) = 0, \text{ se } i = j \\ MM(i,j) = \min_{k, i \leq k \leq j-1} MM(i,k) + MM(k+1,j) + l[i] * c[k] * c[j], \text{ se } i \neq j \end{gathered} \tag{5}$$

Computando o valor da solução ótima com memorização

Algoritmo MM(i, j)

```
1  se i = j então
2      devolva 0
3  se estado (i, j) já tiver sido calculado então
4      devolva memoriza[i][j]
5  k ← i
6  resposta ← infinito
7  enquanto k < j faça
8      resposta ← min(resposta,MM(i, k)+MM(k + 1, j) + l[i] * c[k] * c[j])
9      k = k + 1
10 devolva memoriza[i][j] ← resposta
```

Complexidade. Agora vamos analisar a complexidade desse algoritmo no pior caso. Para cada i existem no máximo n diferentes j, portanto o número de estados é no máximo n^2. Para calcular cada estado fazemos um loop que tem no máximo n iterações, logo o algoritmo tem no máximo $n * n^2$ iterações, portanto é $O(n^3)$.

Problemas relacionados:

- www.spoj.com/problems/MIXTURES/
  Dica: Pensar num estado que represente um intervalo das poções dadas.
- http://www.spoj.com/problems/LISA/

5.4 Formiga no Tetraedro

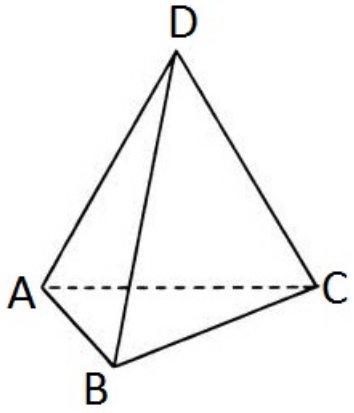


Figura 10: [codeforces.com]

Descrição. Uma formiga está sobre o vértice D do tetraedro. A cada segundo a formiga anda para algum vértice adjacente no tetraedro (Figura 10), com probabilidade uniforme, ou seja, todos os vértices vizinhos têm a mesma probabilidade de serem escolhidos. Dado um inteiro n, queremos saber qual a probabilidade da formiga estar no vértice D após n segundos.

Exemplo 4.4.1: Para a instância $n = 1$, a resposta é zero, pois do D ela poderá ir para A, B ou C.

Exemplo 4.4.2: Para a instância $n = 2$, temos os seguintes possíveis caminhos:

$$\begin{array}{l} D \to A \to B \\ D \to A \to C \\ D \to A \to D \\ D \to B \to A \\ D \to B \to C \\ D \to B \to D \\ D \to C \to A \\ D \to C \to B \\ D \to C \to D \end{array} \tag{6}$$

Desses 9 caminhos, 3 terminam em D, portanto temos que a probabilidade de terminarmos em D é $\frac{1}{3}$.

Caracterizando a subestrutura ótima. Se a formiga está num vértice v, ela pode ir a qualquer um dos 3 vértices adjacentes com probabilidade $\frac{1}{3}$. Se ela ainda tem s segundos para andar ela irá para cada um dos outros 3 vértices para andar, cada um com probabilidade $\frac{1}{3}$. Logo a probabilidade dela terminar no vértice D é $\frac{1}{3}$ vezes a solução para os problemas com $s-1$ segundos para cada vértice vizinho de D. Assim a solução para o problema pode ser escrita como uma combinação dos subproblemas menores.

Definindo a função recursiva Nesse problema podemos notar que se soubermos onde a formiga está e quantos segundos ainda restam, não importa o caminho que ela fez até então. Assim podemos definir um estado de 2 parâmetros $(v, falta)$, onde v é o vértice atual da formiga e $falta$ são quantos segundos a formiga ainda caminhará. Vamos definir uma função recursiva $Formiga(v, falta)$ que calcula a probabilidade da formiga a partir do vértice v, acabar no vértice D após $falta$ segundos. Quando a formiga se encontra num vértice v, ela tem probabilidade de $\frac{1}{3}$ de ir para qualquer um dos outros vértices e a partir desse andará $falta-1$ segundos, dessa forma podemos escrever $Formiga(v, falta)$:

$$Formiga(v, falta) = 0, \text{se } falta = 0 \text{ e } v \neq D$$
$$Formiga(v, falta) = 1, \text{se } falta = 0 \text{ e } v = D$$
$$Formiga(v, falta) = \sum_{\text{vértice } u, u \neq v} \frac{1}{3} * Formiga(u, falta - 1), \text{ caso contrário}$$

Computando o valor da solução ótima com memorização

Algoritmo FORMIGA$(v, falta)$

```
1  se falta = 0 então
2      se v = D então
3          devolva 1
4      se v ≠ D então
5          devolva 0
6  se estado (v, falta) já tiver sido calculado então
7      devolva memoriza[v][falta]
8  resposta ← 0
9  para cada vértice u ≠ v faça
10     resposta = resposta + 1/3 × FORMIGA(u, falta − 1)
11 devolva memoriza[v][falta] ← resposta
```

Complexidade. Vamos analisar a complexidade do algoritmo. O parâmetro $falta$ pode variar de 0 até n e temos 4 opções de vértices para v, totalizando $4 * n$ estados, onde

cada estado tem custo 3 para ser calculado (verificar os 3 vizinhos de v), dando uma soma de $12 * n$, assim podemos concluir que o algoritmo é $O(n)$.

Observação: Existe um algoritmo $O(\log n)$ para resolver esse problema utilizando exponenciação de matrizes. [5]

Problemas relacionados:

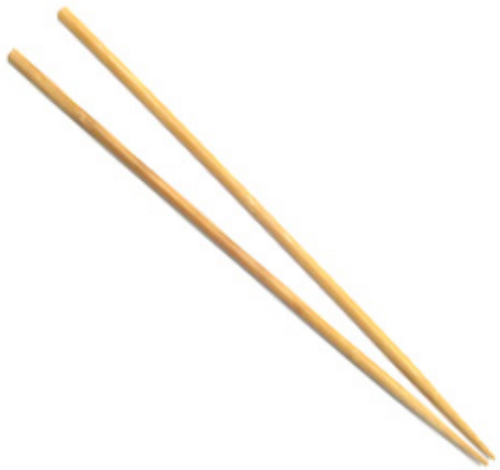
- http://www.spoj.com/problems/GNY07H/
  Dica: Contar no lugar de ver a probabilidade, tentar montar a recursão onde o estado é o tamanho do tabuleiro.

[5] https://pt.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/fast-modular-exponentiation

5.5 Palitos Chineses

Descrição. Stefano gosta de comer com 3 palitos. O maior desses palitos serve de apoio para os outros dois. Para um conjunto de palitos de comprimentos a, b e c ($a < b < c$), a função $R(a, b, c) = (b - a)^2$ mede quão ruim é o conjunto.

Stefano vai dar uma festa com k convidados. Ele tem n palitos com comprimentos no vetor C, todos de tamanhos diferentes. Queremos encontrar k conjuntos de palitos de tal forma a minimizar a soma de quão ruim são os conjuntos. Queremos saber a soma de quão ruim são a soma desses k conjuntos.



Exemplo 4.5.1 Tomemos como exemplo 4 palitos, $\{p_1, p_2, p_3, p_4\}$ com comprimentos $\{12, 10, 8, 5\}$ e $k = 1$ convidado. Os possíveis conjuntos são:

$$\begin{gathered} \{p_1, p_2, p_3\}, R(8, 10, 12) = (10 - 8)^2 = 4. \\ \{p_1, p_2, p_4\}, R(5, 10, 12) = (10 - 5)^2 = 25. \\ \{p_1, p_3, p_4\}, R(5, 8, 12) = (8 - 5)^2 = 9. \\ \{p_2, p_3, p_4\}, R(5, 8, 10) = (8 - 5)^2 = 9. \end{gathered} \tag{7}$$

Portanto, por inspeção, o conjunto $\{p_1, p_2, p_3\}$ é o melhor e a resposta é 4.

Caracterizando a subestrutura ótima. Primeiramente vamos ordenar os palitos em ordem decrescente de tamanho e identificá-los pelos números de 1 a n. Veja que c não

importa no cálculo da função $R(a, b, c)$. Podemos perceber, também, que b e a são sempre palitos vizinhos no vetor ordenado C (a demonstração fica a cargo do leitor). Vamos supor agora que queremos achar uma solução com $k' \leq k$ conjuntos usando os palitos no subconjunto $[p...n-1]$. Perceba que já verificamos p palitos e $k-k'$ conjuntos. Como cada conjunto tem 3 palitos temos $p-3*(k-k')$ palitos não utilizados, que podem ser portanto os maiores palitos dos próximos conjuntos.

Definindo a função recursiva Vamos definir uma função $Palito(p, k')$, que recebe um índice de um palito p e uma quantidade de conjuntos k'. A função retorna a resposta ótima para um subproblema que ainda temos que escolher k' conjuntos usando os palitos $[p...n-1]$. A base da recursão se dá quando $k' = 0$, ou seja, quando não precisamos formar mais conjuntos, ou quando $p \geq n-1$, ou seja, quando falta analisar 1 ou menos palitos e não conseguimos formar mais conjuntos.

Para cada estado temos 4 possibilidades (ser do tipo b, a, c ou não pertencer a nenhum conjunto):

1) O palito p é do tipo b de algum conjunto. Como já analisamos todos os palitos maiores que p, p só pode ser o palito b de algum conjunto se algum palito em $[0..p-1]$ não tiver sido usado. Ou seja, se $p-3*(k-k') > 0$. Nesse caso vamos criar o conjunto com $b = C[p]$ e $a = C[p+1]$ com $R(a, b, c) = (C[p] - C[p+1])^2$ e resolver o novo subproblema $Palito(p+2, k'-1)$.

2) O palito p é do tipo a de algum conjunto. Não precisamos analisar esse caso, pois ao escolhermos o b já escolhemos o a.

3 e 4) O palito p é do tipo c de algum conjunto ou p não pertence a nenhum conjunto. Nesses casos já vimos que basta ignorar p e resolver $Palito(p+1, k')$.

Computando o valor da solução ótima com memorização

Algoritmo PALITO(p, k')

```
1  se k' = 0 então
2      devolva 0
3  se p ≥ n − 1 então
4      Veja que é impossível criar um novo conjunto se p ≥ n − 1
5      devolva infinito
6  se estado (v, falta) já tiver sido calculado então
7      devolva memoriza[v][falta]
8  resposta ← Palito(p + 1, k')
9  se p − 3 ∗ (k − k') > 0 então
10     resposta = min(resposta, (C[p] − C[p + 1])^2 + PALITO(p + 2, k' − 1))
11 devolva memoriza[v][falta] ← resposta
```

Complexidade. É fácil ver que em cada estado fazemos um número constante de comparações, portanto cada estado tem custo $O(1)$. Logo a complexidade da função é o número de estados. O parâmetro k' varia de 0 até k e p varia de 0 até n. Logo, existem nk estados e a complexidade $O(nk)$.

Problemas relacionados:

- UVA - 10271 - Chopsticks

5.6 Empilhamento de Caixas

Descrição. Uma empresa tem n caixas e precisa empilhá-las. O processo de empilhamento é tal que para cada caixa existe no máximo uma caixa diretamente sobre ela. As caixas são numeradas de 0 a $n-1$ tal que cada caixa i tem resistência $R[i]$ e peso $P[i]$, ambos inteiros. A resistência de cada caixa representa o maior peso que esta pode suportar no empilhamento, ou seja, a soma dos pesos das caixas empilhadas acima dela, direta ou indiretamente, tem de ser menor ou igual a $R[i]$. Dados n, R, P, a empresa quer saber o tamanho da maior pilha que se pode formar. É dado também que $R[i]$ possa ter um valor muito grande, então a empresa quer um algoritmo que não dependa dos valores de R.

Figura 11:

Exemplo 4.6.1 Tomemos como exemplo 3 caixas, $\{c_1, c_2, c_3\}$, com $P = \{10, 6, 8\}$ e $R = \{6, 0, 20\}$. Veja que se empilharmos as caixas na ordem $c_3 \to c_1 \to c_2$, todas as resistências serão satisfeitas. $R[c_3] = 20 \geq P[c_2] + P[c_1] = 16$, $R[c_1] = 6 \geq P[c_2] = 6$. Logo a resposta para esse caso é 3.

Propriedades Primeiramente observe que podemos atribuir uma resistência para uma pilha que equivale a quanto peso aquela pilha ainda aguenta. Seja R_p tal resistência. Para calcular R_p basta examinar toda caixa na pilha e verificar quanto peso ela ainda suporta. R_p será o mínimo entre esses valores. Ao colocarmos uma nova caixa c no topo da pilha $R_p = \min(R_p - P[c], R[c])$.

Diferente do problema da Mochila e da Moeda a ordem da escolha é importante. Então vamos estudar algumas propriedades desse problema para tentarmos definir algum tipo de ordem.

Lema: Sejam c_1 e c_2 duas caixas, se $R[c_1] + P[c_1] > R[c_2] + P[c_2]$, deixar c_1 abaixo de c_2 é melhor ou igual a deixar c_1 acima de c_2. Ou seja, existe solução ótima tal que as caixas estão ordenadas em ordem decrescente pela soma da sua resistência com seu peso.

Demonstração: Seja $S[i] = R[i] + P[i]$, para $i \in \{0, 1, \cdots, n-1\}$. Sejam c_1, c_2 duas caixas onde $S[c_1] > S[c_2]$. Se p_1 é a pilha onde c_1 está logo abaixo de c_2 temos que $R_{p_1} = min(S[c1] - P[c_1] - P[c_2], S[c_2] - P[c_2])$. Analogamente se p_2 é a pilha onde c_1 está logo acima de c_2, $R_{p_2} = min(S[c_2] - P[c_1] - P[c_2], S[c_2] - P[c_1])$. É fácil ver que $S[c_2] - P[c_2] \geq S[c_2] - P[c_1] - P[c_2]$ e $S[c1] - P[c_1] - P[c_2] \geq S[c_2] - P[c_1] - P[c_2]$, logo $R_{p_1} \geq R_{p_2}$.

Veja que com esse lema basta decidir se a caixa pertence ou não auma solução ótima pois já sabemos a ordem.

Caracterizando a subestrutura ótima. Temos as caixas ordenadas descrescentemente por $S[i]$. Vamos tentar montar a solução a partir disso. Dado uma pilha p, para saber se podemos colocar uma caixa c no topo precisamos verificar se $R_p \geq P[c]$. Então, se temos c e p, a solução ótima inclui ou não c. Se incluir c temos que resolver um subproblema onde $R_p = min(R_p - P[c], R[c])$ sem a caixa c. Caso contrário temos que resolver um subproblema com a mesma resistência sem a caixa c. Portanto o problema obedece a propriedade da subestrutura ótima.

Definindo a função recursiva Temos então um problema do mesmo tipo da Mochila e podemos escrever uma função $Empilha(R_p, c)$, que recebe a resistência da pilha atual p, a caixa c que estamos analisando e retorna a maior quantidade de caixas que pode ser empilhada sobre p utilizando as caixas a partir de c. Ou seja:

$$
\begin{aligned}
Empilha(R_p, c) &= 0, \text{ se } c = n \\
Empilha(R_p, c) &= Empilha(R_p, c+1), \text{ se } R_p < P[c] \\
Empilha(R_p, c) &= \max(Empilha(R_p, c+1), 1 + Empilha(\min(R[p] - P[c], R[c]), c+1), \text{c.c}
\end{aligned}
$$

Porém não podemos passar R_p como parâmetro da recursão pois ele pode assumir valores muito grandes deixando assim o número de estados muito grande e portanto a complexidade muito alta. Precisamos contornar esse problema, para isso vamos definir uma nova função $Empilha2(c, tam)$, que recebe a caixa atual c, um tamanho tam e retorna a maior resistência que uma pilha de tamanho tam pode ter usando as caixas em $[0..c]$. Podemos separar $Empilha2(c, tam)$ em 2 casos:
1) Caixa c não pertence a solução ótima. Nesse caso $Empilha2(c, tam) = Empilha2(c - 1, tam)$, tentamos com as caixas $[0..c-1]$.
2) Caixa c pertence a solução ótima. Nesse caso vamos tentar montar a pilha de tamanho $tam - 1$ com as caixas $[0..c-1]$ para colocar a caixa c no topo, porque quanto maior a resistência dessa pilha maior a chance de conseguirmos colocar a caixa c.

A base da nossa recursão é o estado com uma única caixa (fácil de decidir), $c = 0$, ou quando o tamanho da pilha é igual a 0, $tam = 0$ (também fácil de resolver). Vamos definir $Empilha2(c, tam) = -1$ para um estado impossível, ou seja, é impossível formar uma pilha de tamanho tam com as caixas $[0..c]$. A resposta para o nosso problema pode ser definida como o maior tam onde $Empilha2(n-1, tam) \neq -1$. Em outras palavras é a maior pilha que pode se formar usando todas as caixas.

Computando o valor da solução ótima com memorização

Algoritmo EMPILHA2(c, tam)

```
1  se tam = 0 então
2      devolva infinito
3  se c = 0 então
4      se tam = 1 então
5          devolva R[0]
6      senão
7          Veja que é impossível criar uma pilha de tamanho maior que 1 com 1 caixa
8          devolva -1
9  se estado (c, tam) já tiver sido calculado então
10     devolva memoriza[c][tam]
11 resposta ← Empilha2(c − 1, tam)
12 R_p ← Empilha2(c − 1, tam − 1)
13 se R_p ≥ P[c] então
14     resposta ← max(resposta, min(R_p − P[c], R[c]))
15 devolva memoriza[c][tam] ← resposta
```

Complexidade. É fácil ver que cada estado tem custo $O(1)$, pois fazemos somente um número constante de comparações. Então a complexidade do algoritmo é equivalente ao número de estados. O parâmetro tam pode variar de 0 a n e c pode variar também de 0 a n, temos assim $O(n^2)$ estados.

Problemas relacionados:

- roller coaster - live archive

5.7 Jogo das caixas

Descrição. Um grupo de n amigos está participando de um jogo. Nesse jogo existem n caixas numeradas de 0 a $n-1$. Dentro de cada caixa encontra-se um nome de um dos amigos, de tal forma que cada amigo tem uma caixa que lhe corresponda. A probabilidade de um amigo encontrar seu nome numa caixa qualquer é de $\frac{1}{n}$, ou seja, obedece uma distribuição uniforme. No jogo cada integrante do grupo abrirá k caixas de sua escolha. Se não encontrar seu nome em uma dessas k caixas todos os participantes perdem. Cada participante desconhece as escolhas feitas pelos outros participantes. Porém podem conversar antes de começar o jogo e definir alguma estratégia. Se todos os participantes escolherem k caixas aleatoriamente, a probabilidade de vencerem é de $(\frac{k}{n})^n$. Não satisfeitos com essa probabilidade, os integrantes do grupo escolheram uma estratégia para abrir as caixas seguindo o seguinte algoritmo. Primeiramente, enumeram todos os integrantes do grupo de 0 a $n-1$. Cada integrante começa abrindo a caixa correspondente ao seu número e enquanto não encontrar o seu próprio nome, abrirá a caixa cujo número corresponde ao nome que acabou de encontrar.

Dados n e k, com $0 < k \leq n$, determine a probabilidade dos amigos vencerem utilizando o algoritmo descrito.

Exemplo 4.7.1 Tome um exemplo com $n = 3$ e $k = 2$. Existem 6 possibilidades para os números dentro das caixas:

1) $\{0, 1, 2\}$ (caixa 0 tem o número 0, caixa 1 tem o número 1 e a caixa 2 o número 2). Nesse caso o primeiro jogador, jogador 0, abre a caixa 0 e encontra o seu número, mesmo ocorre para os jogadores 1 e 2. Os jogadores vencem.

2) $\{0, 2, 1\}$. (caixa 0 tem o número 0, caixa 1 têm o número 2 e a caixa 2 o número 1). Nesse caso o jogador 0 abre a caixa 0 e já encontra o seu número. O jogador 1 abre a caixa 1 achando o número 2, em seguida ele vai para a caixa 2 achando o seu número. O jogador 2 encontra o número 1 em sua caixa, em seguida se dirige para a caixa 1 onde encontra seu número. Jogadores vencem.

3) $\{1, 0, 2\}$. Jogadores vencem.

4) $\{1, 2, 0\}$. O jogador 0 abre sua caixa encontrando o número 1, em seguida abre a caixa 1 encontrando o número 2. O jogador 0 já abriu $k = 2$ caixas e não encontrou seu número (0), portanto ele perdeu e todos os outros participantes perdem.

5) $\{2, 0, 1\}$. Os jogadores perdem.

6) $\{2, 1, 0\}$. Os jogadores vencem.

Veja que cada caso é equiprovável por definição do problema. Os jogadores vencem em 4 dos 6 casos, portanto a probabilidade deles vencerem é $\frac{2}{3}$.

Permutações Para a solução desse problema é conveniente definir uma permutação. Uma permutação p de tamanho n é uma sequência de n números inteiros diferentes em $[0, n-1]$. Denotamos uma permutação com um vetor $p[]$. Por exemplo a permutação $\{2, 0, 1\}$ é representada por $p[0] = 2$, $p[1] = 0$ e $p[2] = 1$. Podemos representar uma permutação através de um grafo com n vértices, em que para cada vértice i cria-se uma aresta de i para $p[i]$. Veja que cada vértice tem exatamente uma aresta de entrada e uma de saída, portanto esse grafo é formado por ciclos. A figura 12 ilustra o grafo definido para a permutação $\{1, 0, 3, 2\}$.

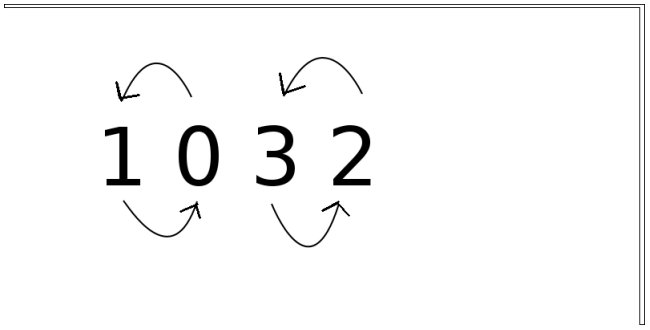


Figura 12: Grafo de Permutação com $n = 4$

Propriedades Para solucionar o problema precisamos identificar algumas propriedades. O problema se resume a definir a probabilidade de uma permutação aleatória p ser vencedora. Dado um jogador i, vamos analisar que tipo de permutação é perdedora para ele. Seja p uma permutação, o jogador i começará abrindo a caixa i e seguirá as arestas do grafo definido pela permutação até que ele encontre seu número ou já tenha aberto k caixas. Veja que para que o jogador vença, ele tem que achar o seu número em uma caixa, ou seja, ele encontra um ciclo no grafo com até k arestas. Podemos perceber que o jogador perde se, no grafo definido pela permutação p, o tamanho do ciclo começando em i é maior que k. Com isso podemos reescrever o problema.

Descrição 2. Calcule a probabilidade de uma permutação aleatória de tamanho n ter seu maior ciclo, no grafo definido por p, menor ou igual a um k dado.

Caracterizando a subestrutura ótima. Vamos olhar para o primeiro elemento da permutação, ele pode estar num ciclo de tamanho 1, 2, 3 ... ou n. Precisamos calcular a probabilidade de cada um desses casos acontecerem. Vamos supor que o elemento está num

ciclo do tamanho i. Vamos chamar de $P(n, tam)$ a probabilidade do primeiro elemento de uma permutação de tamanho n estar num ciclo de tamanho tam. Agora vamos analisar o que acontece quando o primeiro elemento está num ciclo de tamanho tam. Temos que analisar 2 casos:

1) $tam > k$. Nesse caso já podemos dizer que a permutação é perdedora.

2) $tam \leq k$. Nesse caso ainda não sabemos se a permutação é vencedora ou perdedora. Porém sabemos que tam elementos estão num ciclo de tamanho menor ou igual a k, basta saber a probabilidade dos outros $n - tam$ elementos. Ou seja precisamos resolver um subproblema com uma permutação de tamanho $n - tam$ e $k = k$.

Cálculo da função $P(n, tam)$. Para relembrar, $P(n, tam)$ é a probabilidade do primeiro elemento de um permutação aleatória estar num ciclo de tamanho tam. Como a permutação é aleatória a aresta que sai do primeiro vértice pode estar apontada para qualquer vértice (inclusive ele mesmo) com a mesma probabilidade. Vamos calcular $P(4, 2)$. O primeiro vértice pode estar ligado a qualquer vértice menos com si mesmo, probabilidade $\frac{3}{4}$, e esse vértice precisa voltar para o primeiro vértice, probabilidade $\frac{1}{3}$. Fazendo as contas temos $\frac{1}{3} * \frac{3}{4} = \frac{1}{4}$. Em termos mais gerais, o primeiro elemento pode estar ligado a qualquer um menos a si mesmo. O mesmo vale para o segundo até o tam-ésimo elemento que tem de fechar o ciclo. Logo temos:

$$P(n, tam) = \frac{n-1}{n} * \frac{n-2}{n-1} * \frac{n-3}{n-2} * \cdots * \frac{n-tam+1}{n-tam+2} * \frac{1}{n-tam+1} = \frac{1}{n}$$

Veja que $P(n, tam) = \frac{1}{n}$ para qualquer valor de tam!!!!.

Definindo a função recursiva Vamos definir uma função $R(n, k)$ que recebe o tamanho da permutação aleatória e o tamanho do ciclo máximo e devolve a probabilidade de uma permutação aleatória de tamanho n ter seu maior ciclo de tamanho menor ou igual a k. Veja que para $n \leq 1$ é trivialmente 1. Para os outros casos, fixamos o tamanho do ciclo do primeiro elemento, e multiplicamos pela probabilidade dos jogadores fora do ciclo também vencerem. Vamos chamar de t o tamanho do ciclo do primeiro elemento, temos:

$$R(n,k) = \begin{cases} 1, & \text{se } n \leq 1 \\ \displaystyle\sum_{t=1}^{\min(n,k)} \frac{1}{n} R(n-t,k), & \text{caso contrário.} \end{cases}$$

Computando o valor da solução ótima com memorização

Algoritmo R(n, k)

```
1 se n ≤ 1 então
2     devolva 1
3 se memoriza[n] já tiver sido calculado então
4     devolva memoriza[n]
5 resposta ← 0
6 para t = 1 até min(n, k) faça
7     resposta ← resposta + 1/n × R(n − t, k)
8 devolva memoriza[n] ← resposta
```

Complexidade. É fácil ver que o tamanho da permutação aleatória passada como parâmetro varia de 0 até n e k é sempre o mesmo. Para cada estado fazemos um loop que varia até no máximo n. Portanto temos um algoritmo $O(n^2)$.

Melhorando a Solução Podemos melhorar ainda mais essa solução!

Definimos $S(n,k) = \sum_{i=1}^{n} R(i,k)$.

Então, $P(n,k) = \frac{1}{n}(S(n-1,k) - S(n - min(n,k) - 1, k)$.

Pelas relações anteriores temos:

$$\begin{aligned} S(n,k) &= \tfrac{1}{n}(S(n-1,k) - S(n - min(n,k) - 1, k) + \sum_{i=1}^{n-1} P(i,k) \\ &= \tfrac{1}{n}(S(n-1,k) - S(n - min(n,k) - 1, k) + S(n-1,k). \end{aligned}$$

Finalmente, $P(n,k) = S(n,k) - S(n-1,k)$. Note que $S(n,k)$ pode ser calculado em tempo $O(n)$ usando programação dinâmica.

5.8 TSP

Descrição. Stefano quer visitar uma lista de n países diferentes em alguma ordem. Os países são numerados de 0 a $n-1$. Ele começa no país 0 e quer terminar no país 0, passando pelos países $1, 2, \cdots, n-1$ exatamente uma vez. Para cada par de países i,j ele sabe o custo da passagem. Seja $M[i][j]$ o custo para viajar de i para j. Dados n e M determine o menor custo para fazer a viagem.

Curiosidades TSP significa "Travelling salesman problem". É um dos problemas mais estudados na computação

Bitmask Em muitos problemas precisamos passar como parâmetro da função recursiva um subconjunto de elementos. Nem sempre passar o índice de um elemento basta. O TSP é um exemplo disso, pois o problema se resume a decidir uma ordem e para escolher a próxima cidade (elemento) precisamos sabe quais já escolhemos. Para passar um subconjunto como parâmetro de uma função recursiva vamos utilizar da técnica chamada bitmask. Uma bitmask é um número que representa um subconjunto. Vamos supor que temos n elementos, e uma bitmask b é um número entre [0 e $2^n - 1$]. Para saber qual conjunto esse número representa basta escrevê-lo em base 2. O i-ésimo elemento está em b se o i-ésimo bit de b for 1. **Exemplo:** Vamos supor que temos $n = 4$ elementos. Se $b = 8$, em binário $b = 1000$, portanto representa o conjunto $\{4\}$. Se $b = 11$, em binário $b = 1011$, portanto representa o conjunto $\{1, 3, 4\}$.

Caracterizando a subestrutura ótima Vamos supor que já fizemos uma parte do caminho, o que precisamos saber para poder decidir a próxima cidade a ser visitada? Basta saber a cidade atual e quais cidades já foram visitadas. Então seja b esse conjunto de cidades e c a cidade atual. Para cada cidade não visitada, escolhemos ela como a próxima cidade. Seja c_2 essa cidade, agora precisamos resolver um novo subproblema com $b' = b \cup c_2$ e $c' = c_2$. Ou seja adicionamos uma cidade as cidades visitadas e estamos atualmente na cidade c_2.

A solução ótima será composta, portanto, da solução ótima de uma instância iniciando na cidade c_2 com mais essa cidade já visitada. Com isso temos a propriedade da subestrutura ótima.

Definindo a função recursiva Vamos chamar de TSP (b, c), uma função recursiva que recebe um subconjunto de cidades já visitadas b e a cidade atual c e retorna o custo do caminho mais barato de c até a cidade 0 visitando todas as cidades ainda não visitadas.

Para cada estado basta iterarmos sobre as cidades ainda não visitadas e tentar aquela como a próxima. Nessa nova chamada teremos o mesmo subconjunto adicionando a nova cidade, operação fácil de fazer com bitmasks. A base na nossa função se dá quando já visitamos todas as cidades, nesse caso basta voltar para a cidade 0.

Computando o valor da solução ótima com memorização

Algoritmo TSP(c, b)

```
1  se b = 2^n − 1 então
2      devolva M[c][0]
3  se memoriza[c][b] já tiver sido calculado então
4      devolva memoriza[c][b]
5  resposta ← infinito
6  para i = 1 até n faça
7      se i não visitado então
8          resposta ← min(resposta, M[c][i] + TSP(i, b ∪ {i}))
9  devolva memoriza[n] ← resposta
```

Complexidade Na solução descrita acima, para cada estado fazemos um "for" que varia de 1 até n. Vamos calcular o número de estados. A variável c pode variar de 0 até n e b pode variar de 0 até 2^n, portanto existem $n2^n$ estados e a complexidade do aloritmo é $O(n^2 2^n)$. Veja que se fôssemos fazer a força bruta, o número de caminhos possíveis corresponde ao numero de permutações de $n-1$ cidades (todas menos a 0), portanto $(n-1)!$.

Veja nessa tabela como a complexidade melhorou:

	$n^2 2^n$	$(n-1)!$
$n = 5$	800	24
$n = 10$	102400	362880
$n = 15$	7372800	87178291200
$n = 20$	419430400	121645100408832000

Problemas relacionados:

- http://www.spoj.com/problems/ASSIGN/

5.9 Jogo num Grafo

Descrição Stefano e Carlinhos estão jogando um jogo. Nesse jogo temos um grafo dirigido acíclico e em um vértice *inicial* existe uma bola. Stefano é o primeiro a jogar. Em cada turno o jogador pega a bola e pode movê-la através de uma aresta do grafo incidente ao vértice que a bola se encontra naquele momento. O jogo termina quando um jogador não puder fazer nunhuma jogada, e esse jogador é declarado perdedor. Dado um grafo e a posição inicial da bola diga que jogador vence.

Caracterizando a subestrutura ótima É fácil ver que o jogo termina quando a bola se encontra num vértice do grafo com grau de saída igual a 0. Vamos chamar esses vértices de vértices perdedores. Podemos ver também que se a bola se encontra num vértice que tem aresta até um desses vértices perededores, o jogador vence, pois basta mover a bola para o vértice perdedor.

Dessa forma os vértices podem ser divididos entre vencedores e perdedores. Um vértice será vencedor quando tiver em sua vizinhança um vértice perdedor. Vamos mostrar em seguida uma forma recursiva de fazer essa partição.

Como o grafo não tem ciclos ele admite uma ordenação topológica. Assim podemos criar uma função recursiva $Joga(v)$, que recebe um vértice v e retorna 1 se o vértice for vencedor e 0 caso contrário. No cálculo da função basta fazer a chamada para todos os vértices que a bola pode se mover e verificar se existe um que é perdedor. A base se dá quando o vértice tem grau 0; nesse caso o vértice é perdedor.

Computando o valor da solução ótima com memorização

Algoritmo JOGA(v)

```
1 se memoriza[v] já tiver sido calculado então
2     devolva memoriza[v]
3 resposta ← perdedor
4 para todo vértice viz que existe a aresta v − viz faça
5     se JOGA(viz) = perdedor então
6         resposta ← ganhador
7 devolva memoriza[v] ← resposta
```

Complexidade Vamos chamar de n o número de vértices do grafo e m o número de arestas. Veja que cada vértice corresponde a um estado e cada aresta é vista uma única fez, logo temos uma complexidade de $O(n + m)$.

CuriosidadesVamos supor que tivéssemos k grafos e em cada grafo uma bola. Cada jogador na sua vez pode escolher uma bola e movê-la, o jogo termina quando um jogador não conseguir mover nenhuma bola de nenhum grafo e ele é considerado perdedor. Esse problema pode ser resolvido utilizando Sprague-Grundy Theory. [6]

E se o grafo tivesse ciclos? Esse problema também pode ser resolvido por uma técnica estudada por Cedric Smith. [7]

[6] http://en.wikipedia.org

[7]

5.10 Contando 1s

Stefano estava muito curioso um certo dia. Ele tinha 2 números, a e b, $a \leq b$, e ele escreveu todos os números de a a b inclusive e contou quantas vezes o dígito 1 aparecia. Mas e se a e b forem muito grandes? Nesse caso ele não é capaz de contar e pediu para você fazer um programa que contasse para ele.

Exemplo Tome $a = 5$ e $b = 13$, veja que em {5, 6, 7, 8, 9, **1**0, **11**, **1**2, **13**}, o dígito 1 ocorre 5 vezes.

Propriedades Seja $f(x, y)$ o número de vezes que o dígito 1 aparece entre x e y inclusive. É fácil notar que $f(x, y) = f(0, y) - f(0, x - 1)$. Logo o problema se resume a calcular $f(0, x)$, vamos chamar de $D(x) = f(0, x)$.

Caracterizando a subestrutura ótima Problemas de contagem num intervalo grande de números são clássicos em programação dinânimca e podemos utilizar uma técnica que gosto de chamar de "Programação Dinâmica nos dígitos". A técnica consiste em criar uma função recursiva que analisa dígito a dígito do número, tentando criar todos os números possíveis e passando como parâmetro da recursão as informaçãos importantes desse número.
Exemplo: Vamos supor que $x = 12$. Analisamos da esquerda pra direita. Estamos olhando para o dígito 1, então precisamos contar todos os números que tem o primeiro dígito 0 e 1 pois se for maior ou igual a 2 já passou de x. Se o primeiro dígito for 0 estamos livres para fazer qualquer escolha pro segundo pois sempre será menor que x. Se o primeiro dígito for 1 aí o segundo tem que ser no máximo 2.
Observando o exemplo podemos notar que ao analisarmos um dígito na função recursiva precisamos saber se o prefixo do número que estamos gerando é igual ao prefixo de x, nesse caso não temos escolha "livre" caso contrário não podemos colocar um dígito maior que o equivalente em x pois tornaria o número que estamos criando maior que x. Outra coisa que podemos notar é que precisamos saber o número de 1s que já apareceram para fazer essa contagem. A base da nossa recursão se dá quando já vimos todos os dígitos.

Veja que temos então dois casos:

1) Quando o prefixo bate. Nesse caso ou escolhemos um dígito igual ao correspondende em x e temos que resolver um novo subproblema andando um dígito para a direita com o prefixo igual, iu colocamos um dígito menor ao correspondente em x e temos que resolver um novo subproblema andando um dígito para a direita com prefixo diferente.

2) No caso em que o prefixo não bate. Nesse caso podemos escolher qualquer dígito e basta resolver um novo subproblema com o prefixo diferente um dígito para a direita.

Definindo a função recursiva Vamos chamar de $Dig(x, pos, uns, bate)$, onde x é o mesmo de $D(x)$, *pos* corresponde ao dígito que estamos analisando, *uns* é a quantidade de 1s já usados e *bate* é uma "flag"dizendo se o prefixo é o mesmo de x. A função retorna a soma dos dígitos 1 em todos os números em $\{0, 1, 2, \cdots, n\}$ que obedecem os parâmetros.

Vamos separar o cálculo da função em 2 casos:

1) A variável $bate = 0$: Nesse caso estamos livres para colocar qualquer dígito no número e basta contar os uns.

2) A variável $bate = 1$: Nesse caso só podemos colocar dígitos menors ou iguais ao *pos*-ésimo de x.

Computando o valor da solução ótima com memorização

Algoritmo DIG$(x, pos, uns, bate)$

```
1  se todos dígitos já vistos então
2      devolva uns
3  se memoriza[pos][uns][bate] já tiver sido calculado então
4      devolva memoriza[pos][uns][bate]
5  d ← pos-ésimo dígito de x
6  resposta ← 0
7  se bate = 0 então
8      para i de 0 até 9 faça
9          se i = 1 então
10             resposta ← resposta+DIG(x, pos, uns + 1, 0)
11         senão
12             resposta ← resposta+DIG(x, pos, uns, 0)
13 senão
14     para i de 0 até d faça
15         bate2 ← bate
16         se i ≠ d então
17             bate2 ← 0
18         se i = 1 então
19             resposta ← resposta+DIG(x, pos, uns + 1, bate2)
20         senão
21             resposta ← resposta+DIG(x, pos, uns, bate2)
22 devolva memoriza[n] ← resposta
```

Complexidade Vamos calcular o número de estados. Primeiramente veja que o número de dígitos de b é $O(\log b)$, o número de uns também e a variável *bate* varia entre 0 e 1. Temos assim $O(\log^2 b)$ estados. Para cada estado fazemos uma varredura vendo os 10 possíveis próximos dígitos que é uma constante. Sendo assim temos $O(\log^2 b)$ como complexidade final.

6 Exercícios

Seguem uma lista de problemas disponíveis em juizes online ordenados pelo site.

6.1 www.spoj.pl

- http://www.spoj.pl/problems/MARTIAN/
- http://www.spoj.pl/problems/MIXTURES/
- http://www.spoj.pl/problems/SQRBR/
- http://www.spoj.pl/problems/SAMER08D/
- http://www.spoj.pl/problems/ACMAKER/
- http://www.spoj.pl/problems/AEROLITE/
- http://www.spoj.pl/problems/AIBOHP/
- http://www.spoj.pl/problems/BACKPACK/
- http://www.spoj.pl/problems/COURIER/
- http://www.spoj.pl/problems/DP/
- http://www.spoj.pl/problems/EDIST/
- http://www.spoj.pl/problems/KRECT/
- http://www.spoj.pl/problems/GNY07H/
- http://www.spoj.pl/problems/LISA/
- http://www.spoj.pl/problems/MINUS/
- http://www.spoj.pl/problems/NAJKRACI/
- http://www.spoj.pl/problems/PHIDIAS/
- http://www.spoj.pl/problems/PIGBANK/
- http://www.spoj.pl/problems/PT07X/
- http://www.spoj.pl/problems/VOCV/

- http://www.spoj.pl/problems/TOURIST/
- http://www.spoj.pl/problems/MKBUDGET
- http://www.spoj.pl/problems/MMAXPER
- http://www.spoj.pl/problems/ANARC07G
- http://www.spoj.pl/problems/MENU

6.2 http://uva.onlinejudge.org/

- 111 - History Grading
- 103 - Stacking Boxes
- 10405 - Longest Common Subsequence
- 674 - Coin Change 26969
- 10003 - Cutting Sticks
- 116 - Unidirectional TSP
- 10131 - Is Bigger Smarter?
- 10066 - The Twin Towers
- 10192 - Vacation
- 147 - Dollars
- 357 - Let Me Count The Ways
- 562 - Dividing coins
- 348 - Optimal Array Multiplication Sequence
- 624 - CD
- 10130 - SuperSale
- 531 - Compromise
- 10465 - Homer Simpson

- 10285 - Longest Run on a Snowboard
- 437 - The Tower of Babylon
- 10404 - Bachet's Game
- 620 - Cellular Structure
- 825 - Walking on the Safe Side
- 10069 - Distinct Subsequences
- 10534 - Wavio Sequence
- 10051 - Tower of Cubes
- 10651 - Pebble Solitaire
- 590 - Always on the run
- 10306 - e-Coins
- 10739 - String to Palindrome
- 10304 - Optimal Binary Search Tree
- 10271 - Chopsticks
- 10617 - Again Palindrome
- 11137 - Ingenuous Cubrency
- 10154 - Weights and Measures
- 10201 - Adventures in Moving - Part IV
- 10453 - Make Palindrome
- 10029 - Edit Step Ladders
- 10313 - Pay the Price
- 10401 - Injured Queen Problem
- 10891 - Game of Sum
- 11151 - Longest Palindrome

- 10911 - Forming Quiz Teams
- 10635 - Prince and Princess
- 10564 - Paths through the Hourglass
- 662 - Fast Food
- 10626 - Buying Coke
- 10118 - Free Candies
- 607 - Scheduling Lectures
- 10604 - Chemical Reaction
- 10913 - Walking on a Grid
- 11008 - Antimatter Ray Clearcutting
- 10723 - Cyborg Genes
- 11258 - String Partition
- 10599 - Robots(II)
- 10817 - Headmaster's Headache
- 10163 - Storage Keepers
- 709 - Formatting Text
- 10280 - Old Wine Into New Bottles
- 10558 - A Brief Gerrymander
- 11081 - Strings103 - Stacking Boxes
- 10405 - Longest Common Subsequence
- 674 - Coin Change 26969
- 10003 - Cutting Sticks
- 116 - Unidirectional TSP
- 10131 - Is Bigger Smarter?

- 10066 - The Twin Towers
- 10192 - Vacation
- 147 - Dollars
- 357 - Let Me Count The Ways
- 562 - Dividing coins
- 348 - Optimal Array Multiplication Sequence
- 624 - CD
- 10130 - SuperSale
- 531 - Compromise
- 10465 - Homer Simpson
- 10285 - Longest Run on a Snowboard
- 437 - The Tower of Babylon
- 10404 - Bachet's Game
- 620 - Cellular Structure
- 825 - Walking on the Safe Side
- 10069 - Distinct Subsequences
- 10534 - Wavio Sequence
- 10051 - Tower of Cubes
- 10651 - Pebble Solitaire
- 590 - Always on the run
- 10306 - e-Coins
- 10739 - String to Palindrome
- 10304 - Optimal Binary Search Tree
- 10271 - Chopsticks

- 10617 - Again Palindrome
- 11137 - Ingenuous Cubrency
- 10154 - Weights and Measures
- 10201 - Adventures in Moving - Part IV
- 10453 - Make Palindrome
- 10029 - Edit Step Ladders
- 10313 - Pay the Price
- 10401 - Injured Queen Problem
- 10891 - Game of Sum
- 11151 - Longest Palindrome
- 10911 - Forming Quiz Teams
- 10635 - Prince and Princess
- 10564 - Paths through the Hourglass
- 662 - Fast Food
- 10626 - Buying Coke
- 10118 - Free Candies
- 607 - Scheduling Lectures
- 10604 - Chemical Reaction
- 10913 - Walking on a Grid
- 11008 - Antimatter Ray Clearcutting
- 10723 - Cyborg Genes
- 11258 - String Partition
- 10599 - Robots(II)
- 10817 - Headmaster's Headache

- 10163 - Storage Keepers
- 709 - Formatting Text
- 10280 - Old Wine Into New Bottles
- 10558 - A Brief Gerrymander
- 11081 - Strings

7 Conclusão

Por meio de exercícios e problemas, o presente trabalho introduziu ao leitor algumas ferramentas de trabalho para os que pretendem se aprofundar no tema da "Programação Dinâmica". Com alguma sorte, espero ter estimulado alguns de meus leitores a ingressar nesse fascinante mundo.

Para esse fim, minha preocupação central foi, desde o início, oferecer um material didático claro e compreensivo. Essa foi a orientação principal tanto na descrição dos problemas, como na resolução dos mesmos. Quero oferecer ao leitor uma referência nos seus estudos de programação dinâmica.

Nesse sentido, essa pequena contribuição pretende suprir uma ausência de material de estudo, principalmente para os que gostariam de competir em eventos como a Maratona de Programação e também para os que simplesmente querem se aprofundar no tema.

Por isso, no corpo do trabalho podem ser encontrados problemas simples assim como problemas bastante complexos, atendendo tanto ao estudante iniciante quanto ao avançado.

Enfim, espero que futuros alunos possam se beneficiar desse trabalho, auxiliando a formação de novos estudioso da Programação Dinâmica e da Ciência da Computação em Geral.