Análise e Desenho de Algoritmos: Trabalho #1

Lego Mosaics

Prof. Margarida Mamede

Turno Prático 1

Luis Marques 34213

April 16, 2013

Contents

1 Apresentação do Problema

É possível criar complexas figuras colocando sequências de pequenas peças coloridas de Lego. Neste problema, temos à disposição um número infinito de peças, de diversas cores, dos seguintes tamanhos: 1×1, 1×2, 1×3, 1×4, 1×6, 1×8, 1×10, 1×12 e é perguntado qual o número de combinações possíveis de colocação destas peças para igualar um dado mosaico.
Cada peça poderá apenas ser colocada na horizontal.

2 Resolução do Problema

Para resolver o problema indicado na secção anterior, entendi que não seria necessário calcular o valor das combinações para mosaico inteiro de uma vez mas que seria mais simples e intuitivo partir este cálculo por pequenos problemas. Assim sendo, restringi-me apenas ao problema de calcular o número de possibilidades de uma sequência de x blocos independentemente da cor, multiplicando depois no fim o número de combinações de cada subproblema.
Resolvendo o problema para a maior subsequência, e armazenando os resultados parciais numa estrutura(neste caso um vector) evito cálculos repetidos.
Para os cálculos parciais e finais utilizei apenas inteiros pois o enunciado diz que no máximo o valor final é 2000000000, que é inferior ao máximo de um inteiro(2^{31})

2.1 Resolução do subproblema

Para resolver o problema de comprimento x, decidi aplicar o seguinte raciocínio.
"Coloco" no mosaico uma peça de comprimento y, deixando apenas disponível x-y posições. Mas o resultado do problema de comprimento x-y já foi obtido anteriormente. Como só tenho uma maneira de colocar uma determinada peça no início da sequência, o total é a soma do resultado para o espaço restante após colocar cada uma das peças que tenho disponíveis. Naturalmente, se a peça tiver exactamente o comprimento disponível, apenas existe uma maneira. Por outro lado, se a peça exceder o espaço disponível não conta como maneira válida sendo adicionado zero ao resultado final.
Assumindo P um vector com os tamanhos das peças disponiveis. O texto anterior pode ser traduzido na seguinte expressão:

$$Comb(int\ size) = \begin{cases} 0 & size < 0 \\ 1 & size = 0 \\ \sum_i^{|P|} Comb(size - P[i]) & c.c. \end{cases}$$

3 Implementação do Algoritmo

3.1 Técnica de desenho de algoritmos utilizada

Programação Dinâmica, pois tratava-se de um problema de contagem que iria utilizar resultado de computações anteriores para chegar ao valor seguinte.

3.2 Breve apresentação das interfaces e das classes criadas

Neste problema não senti necessidade de criar objectos, que não os já oferecidos pelas bibliotecas do Java.

3.3 Estruturas de dados usadas

Lista duplamente ligada - Utilizei esta lista para guardar os tamanhos das sequências maiores do que um. Utilizei esta estrutura pois à partida não é conhecida a quantidade de elementos que vai ter. Como é uma estrutura que tem complexidade temporal constante quer para adições quer remoções na cauda(tail) da lista, torna-se uma escolha natural pois não ocupa mais espaço do que o estritamente necessário.

Vector de char - Utilizei este vector para armazenar uma linha de input. Estrutura que tem complexidade temporal de escrita e de leitura constante e para além disso é simples iterá-la.

Vector de inteiros - Utilizei este vector para armazenar as soluções parciais até ao número máximo do problema. Como o vector referido anteriormente, tem complexidades temporais constantes.

3.4 Pontos importantes do código

Interessante notar que sequências de tamanho inferior a dois não são armazenadas pois iriam resultar em multiplicações por 1, gastando quer memória quer ciclos de relógios desnecessariamente.

4 Análise do Algoritmo

R - Número de linhas

C - Número de colunas

M - Tamanho da maior sequência (Nesta instância do problema no máximo tem o valor 32, valor obtido experimentalmente.Qualquer número maior gera um resultado maior que o esperado)

S - Número de sequências de tamanho maior que dois (Nesta instância do problema tem o valor máximo de 30, caso em que apenas há sequências de tamanho dois)

P - Número de peças (Nesta instância do problema constantemente 9)

Complexidade temporal - Análise passo a passo

Acção			Melhor Caso	Pior Caso	Caso Esperado
Inicialização de variáveis			O(1)	O(1)	O(1)
Criação lista			O(1)	O(1)	O(1)
Ciclo(R vezes)					
	Construção vector input		Θ(1)	Θ(C)	Θ(C)
	Ciclo(C vezes)				
		Leituras e escritas vector	O(1)	O(1)	O(1)
		Adicionar a lista	O(1)	O(1)	O(1)
		Remover da lista	O(1)	O(1)	O(1)
		Comparações	O(1)	O(1)	O(1)
Ciclo(M vezes)					
	Ciclo(P vezes)				
		Comparações	O(1)	O(1)	O(1)
		Leituras e escritas vector	O(1)	O(1)	O(1)
Criação de iterador			O(1)	O(1)	O(1)
Ciclo(S vezes)					
	Múltiplicação		O(1)	O(1)	O(1)

Análise Complexidade temporal:

Complexidade temporal esperada (primeira análise): O(R*2C + M*P + S)

S pode ser maior que a parcela M*P no caso em que temos mais do que 9 sequências de dois e menor em casos como por exemplo quando temos apenas uma sequência de duas posições.

S é sempre menor ou igual que R*2C pois como apenas são guardadas sequências de comprimento superior ou igual a dois, no mínimo para fazer S sequências necessito de 2S espaços nas matriz de input(R*C).

M*P pode ser maior que R*2C, por exemplo, no caso R=1,C=32,M=32,P=9. Ainda assim pode ser menor, por exemplo, no caso R=1000,C=1000,M=1,P=9.

Nas situações em que S é maior que M*P, podemos "ignorar" ambas pois serão sempre inferiores a R*2C. O(R*C)

Nas situações em que S é menor que M*P, podemos "ignorá-lo". O(R*C + M*P)

Em ambos os casos S fica sempre omisso.

Apenas num pequeno leque do domínio M*P é maior que R*2C pois P=9 e M varia entre 1 e 32 enquanto R e C variam entre 1 e 1000.

Complexidade temporal do Algoritmo:

Melhor Caso: O(R*C) Pior Caso: O(R*C + M*P) Caso Esperado: O(R*C)

Análise complexidade espacial:

Complexidade espacial vector de peças - $\theta(P)$

Complexidade espacial lista ligada - $\theta(S)$

Complexidade espacial vector de char (input) - $\theta(C)$

Complexidade espacial vector de int - $\theta(M)$

C é sempre maior ou igual a M pois para ter uma sequência de M blocos é preciso ter pelo menos C colunas. No máximo S tem o valor 30, P tem o valor 9 e C tem valor 1000.

Complexidade Espacial do Algoritmo:

Melhor Caso: O(C) Pior Caso: O(S+C+P) Caso Esperado: O(C)

5 Conclusões

O principal ponto forte da solução é no caso esperado a maior complexidade ser a necessária para ler o input do ficheiro.

Foi experimentada uma solução em que em vez da lista ligada era usado um vector de 32 posições onde era guardada o número de vezes que cada tamanho ocorre. Teoricamente, se se considerar a função Math.pow() de complexidade temporal constante iria melhorar a complexidade temporal do algoritmo. Mas como esta assumpção penso não poder ser válida e ainda esta solução não resultar num código tão claro acabei por deixar cair esta hipótese.

A Code Listing

```
import java.util.*;

public class Main {

  private static final int[] pieces = { 1, 2, 3, 4, 6, 8, 10, 12, 16 };

  /**
   * @param args
   */
  public static void main(String[] args) {

    Scanner in = new Scanner(System.in);
    int rows = in.nextInt(), cols = in.nextInt();
    in.nextLine();

    int maxSequence = 0;
    LinkedList<Integer> list = new LinkedList<Integer>();
    list.add(1);

    for (; rows-- > 0;) {
      char[] input = in.nextLine().toCharArray();
      char lastChar = '.';
      for (int i = 0; i < cols; i++) {
        if (input[i] == '.') {
          lastChar = '.';
          continue;
        }
        if (input[i] == lastChar)
          list.addLast(list.removeLast() + 1);
        else {
          lastChar = input[i];
          if (list.getLast() != 1)
            list.add(1);
        }
        if (list.getLast() > maxSequence)
          maxSequence = list.getLast();
      }
    }

    int[] combinations = calculateComb(maxSequence);

    int res = 1;
    Iterator<Integer> it = list.iterator();
    while (it.hasNext())
```

```
      res *= combinations[it.next()];
    System.out.println(res);


  }

  private static int[] calculateComb(int maxSequence) {
    int[] res = new int[maxSequence + 1];
    res[1] = 1;
    for (int i = 2; i < res.length; i++)
      for (int j = 0; j < pieces.length; j++) {
        if (pieces[j] > i)
          break;
        if (pieces[j] == i) {
          res[i]++;
          break;
        }
        res[i] += res[i - pieces[j]];
      }
    return res;
  }

}
```

Listing 1: Main.java