2º Teste de Análise e Desenho de Algoritmos

Departamento de Informática
Universidade Nova de Lisboa

24 de Maio de 2013

1. [6 valores] Qualquer jogo que pode ser jogado por uma pessoa é um *solitaire*.

   A classe *Solitaire* permite implementar uma paciência com um número ilimitado de jogos, jogados consecutivamente sem qualquer interrupção. Em cada jogo (*game*): forma-se uma sequência arbitrária com os números $0, 1, \ldots, n-1$, onde n é múltiplo de 4; o jogador tem exactamente $n/4$ jogadas (*guesses*) e, em cada jogada, o jogador tenta adivinhar o número que está numa posição da sequência, ganhando um ponto se acertar.

```
import java.util.Random;

public class Solitaire
{
    private Random generator;
    private int[] sequence;          // Memory of the pseudorandom sequence.
    private int guessesPerGame;      // Number of guesses in each game.
    private int numGuesses;          // Number of guesses in the current game.
    private int score;               // Number of right guesses in the current game.

    public Solitaire( int numberOfGuessesPerGame )
    {
        generator = new Random();
        guessesPerGame = numberOfGuessesPerGame;
        sequence = new int[guessesPerGame * 4];
        for ( int i = 0;  i < sequence.length;  i++ )
            sequence[i] = i;
        this.permute();
        numGuesses = 0;
        score = 0;
    }

    public boolean guess( int pos, int value ) throws InvalidPositionException
    {
        if ( pos < 0  ||  pos >= sequence.length )
            throw new InvalidPositionException();
        if ( numGuesses == guessesPerGame )
        {
            this.permute();
            numGuesses = 0;
            score = 0;
        }
        numGuesses++;
        boolean goodGuess = sequence[pos] == value;
        if ( goodGuess )
            score++;
        return goodGuess;
    }
```

```
    public int getScore( )
    {
        return score;
    }

    private void permute( )
    {
        for ( int i = 0;   i < sequence.length − 1;   i++ )
        {
            int pos = i + generator.nextInt(sequence.length − i);
            // Swap elements at positions i and pos.
            int aux = sequence[i];
            sequence[i] = sequence[pos];
            sequence[pos] = aux;
        }
    }

}
```

Considere a função $\Phi(S)$, que atribui a cada objecto S da classe *Solitaire* o quádruplo do valor do atributo *numGuesses*:

$$\Phi(S) = 4 \times S.\text{numGuesses}\,.$$

Prove que Φ é uma função potencial válida e calcule as complexidades amortizadas dos métodos *getScore* e *guess*, justificando. No estudo da complexidade amortizada do método *guess*, assuma que não é levantada a excepção, mas analise separadamente os casos em que a condição do segundo *if* é verdadeira e falsa. Assuma que o método *nextInt* da classe *Random* tem complexidade temporal constante.

(Continue, porque o teste tem mais três perguntas.)

2. [2 valores] O método *hasJust1Entry*, da classe *FibQueue*, retorna *true* se, e só se, a fila de Fibonacci tem exactamente uma entrada.

```
class FibNode<K,V>
{
    public FibNode( K key, V value );
    public EntryClass<K,V> getEntry( );
    public K getKey( );
    public V getValue( );
    public int getDegree( );
    public FibNode<K,V> getChild( );
    public FibNode<K,V> getLeftSibling( );
    public FibNode<K,V> getRightSibling( );
    public FibNode<K,V> getParent( );
    public boolean isMarked( );
    public void setEntry( EntryClass<K,V> newEntry );
    public void setEntry( K newKey, V newValue );
    public void setKey( K newKey );
    public void setValue( V newValue );
    public void setDegree( int newDegree );
    public void incrementDegree( );
    public void decrementDegree( );
    public void setChild( FibNode<K,V> newChild );
    public void setLeftSibling( FibNode<K,V> newLeftSibling );
    public void setRightSibling( FibNode<K,V> newRightSibling );
    public void makeSingleton( );
    public void setParent( FibNode<K,V> newParent );
    public void mark( );
    public void unmark( );
}

class FibQueue<K extends Comparable<? super K>, V>
{
    // (Pointer to) a tree with the smallest key.
    protected FibNode<K,V> min;

    // Number of entries in the priority queue.
    protected int currentSize;

    ..........................................

    // Returns true if, and only if, the number of entries in the priority queue is one.
    public boolean hasJust1Entry( );
}
```

Implemente o método *hasJust1Entry* (na classe interna *FibQueue*) **sem usar o atributo currentSize e sem usar o método size**. Calcule a complexidade temporal do seu algoritmo, no melhor caso e no pior caso, justificando.

3. [6 valores] O **Problema da Soma de Subconjunto** formula-se da seguinte forma. Dados um conjunto C de números inteiros e um inteiro t, existe um subconjunto $S \subseteq C$ tal que:

$$\sum_{e \in S} e \;=\; t\,?$$

Por exemplo, a solução da instância $(\{1, 2, -7, 3, -10, -5\}, 0)$ é sim, porque a soma dos elementos de $\{2, 3, -5\}$ é $2 + 3 + (-5) = 0$.

Prove que o Problema da Soma de Subconjunto é NP-completo.

4. [6 valores] Acabaram de chegar n t-shirts a uma colónia de férias, que vão agora ser distribuídas pelos m monitores (onde $n \geq m > 1$). O número n é múltiplo de 6, porque há seis tamanhos de t-shirts e há o mesmo número de exemplares de cada tamanho. Os tamanhos são 1, 2, 3, 4, 5 e 6, que correspondem a XS, S, M, L, XL e XXL, respectivamente. Há exactamente dois tamanhos que servem a cada monitor. Pretende-se saber se é possível dar a cada um dos monitores uma t-shirt que lhe serve.

Para exemplificar, suponha que há 12 t-shirts e 7 monitores e que os tamanhos que servem a cada monitor estão indicados na Tabela 1. Neste caso, é possível dar a cada monitor uma t-shirt que lhe serve. Por exemplo, poder-se-ia dar: o tamanho 2 ao monitor 4; o tamanho 3 aos monitores 5 e 6; o tamanho 4 aos monitores 3 e 7; e o tamanho 5 aos monitores 1 e 2.

Monitor	Tamanhos que servem ao monitor	
1	4	5
2	5	4
3	4	5
4	2	3
5	3	2
6	3	4
7	3	4

Tabela 1: Tamanhos possíveis para cada monitor.

Escreva um algoritmo (em pseudo-código) que recebe:

- o número n de t-shirts e
- uma matriz de m por 2, com os tamanhos das t-shirts que servem aos m monitores,

e retorna *true* se, e só se, é possível dar a cada um dos monitores uma t-shirt que lhe serve. Assuma que os dados de entrada estão correctos (n é múltiplo de 6 e é superior ou igual a m, cada linha da matriz tem dois inteiros distintos entre 1 e 6, etc.). Estude (justificando) a complexidade temporal do seu algoritmo, no pior caso.