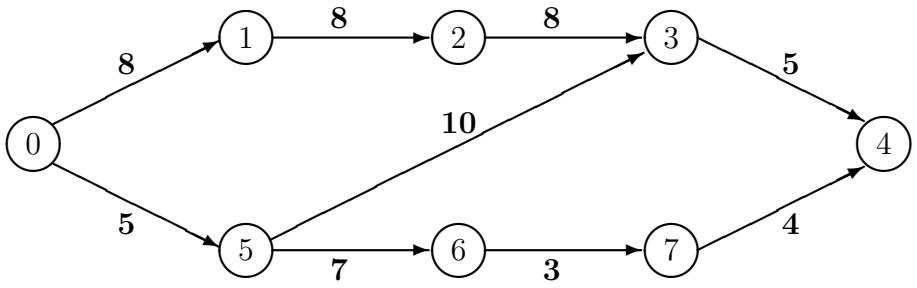
2º Teste de Análise e Desenho de Algoritmos

Departamento de Informática

Universidade Nova de Lisboa

9 de Junho de 2014

1. Suponha que se executa o algoritmo de Edmonds-Karp com o grafo G esquematizado na figura, a fonte 0 e o dreno 4.



Assuma que o método *outIncidentEdges* itera sempre os vértices por ordem crescente. Por exemplo, $G.outIncidentEdges(0)$ produz os vértices 1 e 5 (por esta ordem). Indique:

(a) [3 valores] a sequência de caminhos encontrados da fonte para o dreno e, para cada um desses caminhos, o valor do incremento;

(b) [1 valor] o valor do fluxo máximo;

(c) [1 valor] o fluxo de cada arco de G quando o algoritmo termina.

2. [6 valores] A classe *QueueWithMultiDequeue*, das filas com disciplina FIFO, de elementos do tipo E, implementadas com uma lista ligada, oferece a operação *multiDequeue*.

```
import java.util.Queue;
import java.util.LinkedList;

public class QueueWithMultiDequeue<E>
{
    private Queue<E> queue;

    public QueueWithMultiDequeue( )
    {
        queue = new LinkedList<E>();
    }

    public int size( )
    {
        return queue.size();
    }

    public void enqueue( E element )
    {
        queue.add(element);
    }

    public E dequeue( ) throws EmptyQueueException
    {
        if ( queue.isEmpty() )
            throw new EmptyQueueException();
        return queue.remove();
    }

    public int multiDequeue( int numElems )
    {
        int deletions = 0;
        while ( !queue.isEmpty() && deletions < numElems )
        {
            queue.remove();
            deletions++;
        }
        return deletions;
    }

}
```

Considere a função $\Phi(Q)$, que atribui a cada objeto Q da classe *QueueWithMultiDequeue* o número de elementos guardados na fila:

$$\Phi(Q) = Q.\text{size}() \, .$$

Prove que Φ é uma função potencial válida e calcule as complexidades amortizadas dos métodos *size*, *enqueue*, *dequeue* e *multiDequeue*, justificando. No estudo da complexidade amortizada do método *dequeue*, assuma que não é levantada a exceção.

3. [3 valores] O algoritmo *selectAtRandom* recebe um inteiro positivo *maxValue* e retorna um inteiro entre 1 e *maxValue*. A implementação usa duas classes.

   - A classe *FibonacciQueueInt* implementa a interface *AdaptMinPriQueue<Integer, Integer>* com:
     - uma fila de Fibonacci (uma instância de *FibQueue<Integer, Integer>*) e
     - um vetor cujos elementos são do tipo *FibNode<Integer, Integer>*.

     O construtor recebe um argumento inteiro (*DomainSizeOfValues*), que indica que os valores das entradas podem variar entre 0 e *DomainSizeOfValues* − 1 (incluindo os extremos).
   - A classe *Random* (do pacote *java.util*) implementa um gerador de números pseudo-aleatórios.

```
// Pre-condition: maxValue > 0  and  maxValue <= Integer.MAX_VALUE / 2.
int selectAtRandom( int maxValue )
{
    AdaptMinPriQueue<Integer, Integer> queue = new FibonacciQueueInt(maxValue);
    for ( int i = 0;  i < maxValue;  i++ )
        queue.insert(i, i);
    Random generator = new Random();
    int maxDecrement = 2 * maxValue;
    for ( int i = 0;  i < maxValue;  i++ )
    {
        // Assign a value between 1 and maxDecrement to decrement.
        int decrement = 1 + generator.nextInt(maxDecrement);
        queue.decreaseKey(i, i − decrement);
    }
    return 1 + queue.removeMin().getValue();
}
```

Assuma que o construtor do gerador de números pseudo-aleatórios (*Random*) e o método *nextInt* têm complexidade constante. Estude (justificando) a complexidade temporal de *selectAtRandom*, no pior caso.

4. [6 valores] O Rui, que adora desafios, aceitou participar num jogo a dinheiro. O jogo consiste em percorrer um labirinto, constituído por portas e corredores (de sentido único). Em cada troço de corredor (i.e., entre cada duas portas) existe:

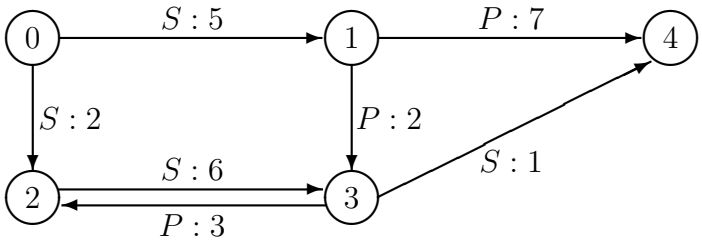
   - ou um saco com moedas de ouro;
   - ou um fosso com crocodilos, uma ponte levadiça e uma ATM (especial).
     Neste caso, para sair do troço, é necessário acionar o mecanismo que faz baixar a ponte, colocando numa ranhura a quantidade de moedas de ouro indicada na parede do troço. Como os jogadores podem não ter moedas suficientes, a ATM dá moedas de ouro com um cartão (normal) de débito ou crédito.

   As moedas de ouro são todas iguais e cada uma vale 300 euros. A quantidade de moedas nos sacos e requeridas para baixar as pontes varia consoante o troço. Se um jogador passar várias vezes pelo mesmo troço, depara-se com a mesma situação: ou encontra sempre um saco com o mesmo número de moedas de ouro, ou encontra sempre a ponte levantada, que baixa com a mesma quantidade de moedas de ouro. Nenhum troço tem saco e ponte.

   Entra-se no labirinto pela porta assinalada com ENTRADA e sai-se por uma outra porta, assinalada com SAÍDA. Cada troço tem uma seta pintada no chão que indica o sentido em que é permitido percorrer o troço. É expressamente proibido retroceder e sabe-se que, qualquer que seja o sítio onde o jogador se encontra, é possível chegar à SAÍDA. Se o jogador chegar à SAÍDA com moedas de ouro, tem de as entregar, recebendo 300 euros por cada uma.

   O Rui já decidiu que participava no jogo. Mas, imediatamente antes de entrar no labirinto, perguntou à organização se, com azar ou por nabice, poderia perder dinheiro. Pretende-se que descubra a resposta, com base na descrição do labirinto.

   Para exemplificar, considere o seguinte labirinto com 5 portas e 7 troços, onde 0 é a porta de entrada, 4 é a porta de saída, $S : m$ indica que o troço tem um saco com m moedas de ouro e $P : n$ significa que o troço tem uma ponte que é acionada com n moedas de ouro:



   Neste caso, o Rui poderia perder dinheiro (600 euros) se percorresse apenas dois troços: entre as portas 0 e 1 e entre as portas 1 e 4.

   Indique como guardaria o labirinto e escreva uma função booleana (em pseudo-código) que recebe um labirinto (na forma que indicou), a porta de entrada e a porta de saída, e retorna *true* se, e só se, o Rui puder perder dinheiro. Estude (justificando) a complexidade temporal do seu algoritmo.