Supervision work
Supervision 11

Tudor Avram
Homerton College, tma33@cam.ac.uk

2 Feb 2016

1 Ticks 1 and 1*

1.1 Tick 1

```
package uk.ac.cam.tma33.alg.tick1;

import uk.ac.cam.rkh23.Algorithms.Tick1.MaxCharHeapInterface;
import uk.ac.cam.rkh23.Algorithms.Tick1.EmptyHeapException;

public class MaxCharHeap implements MaxCharHeapInterface{

char[] mHeap;
int mLength;

public MaxCharHeap(String s) {
        mLength = 0;
        mHeap = new char[1];
        char[] input = s.toCharArray();
        for (int i = 0; i < input.length; i++ ) {
                insert(input[i]);
        }
}

@Override
public char getMax() throws EmptyHeapException {
        if (mLength < 1) {
                // the heap is empty
                throw new EmptyHeapException();
        }
        else {
                char result = mHeap[1];
```

```
                swap(0,mLength);
                mLength--;
                heapDOWN(0);
                // we return the head of the heap
                return result;
        }
}

private void heapDOWN(int node){
        int sonLeft = node*2;
        int sonRight = node*2+1;
        int nextNode;
        if (sonLeft <= mLength) {
                nextNode = sonLeft;
                if (sonRight <= mLength && mHeap[sonRight] >
                                mHeap[sonLeft]) {
                        nextNode = sonRight;
                }
                if (mHeap[nextNode] > mHeap[node]){
                        swap(nextNode,node);
                        heapDOWN(nextNode);
                }
        }
}

private void swap(int x, int y) {
        char aux = mHeap[x];
        mHeap[x] = mHeap[y];
        mHeap[y] = aux;
}

private void resize() {
        int newSize = mHeap.length * 2;
        char[] newHeap = new char[newSize];
        for (int i = 1; i <= mLength; i++) {
                newHeap[i] = mHeap[i];
        }
        mHeap = newHeap;
}

private void heapUP(int node) {
        int father = node/2;
        if (father > 0 && mHeap[node] > mHeap[father]) {
                swap(node,father);
                heapUP(father);
        }
```

```
}

@Override
public void insert(char i) {
        if (mLength + 1 >= mHeap.length) {
                // we double the size of the array
                resize();
        }
        char newChar = Character.toLowerCase(i);
        mLength++;
        mHeap[mLength] = newChar;
        heapUP(mLength);
}

@Override
public int getLength() {
        return mLength;
}

}
```

1.2 Tick 1*

(a) Implementation of MaxHeap :

```
package uk.ac.cam.tma33.alg.Tick1Star;

import java.util.ArrayList;
import java.util.List;

import uk.ac.cam.rkh23.Algorithms.Tick1.EmptyHeapException;
import uk.ac.cam.rkh23.Algorithms.Tick1Star.MaxHeapInterface;

public class MaxHeap<T extends Comparable<T> >
                        implements MaxHeapInterface<T>{

protected List<T> mHeap;

public MaxHeap() {
        mHeap = new ArrayList<T>();
}

public MaxHeap(List<T> input) {
        mHeap = new ArrayList<T>();
        for (T element : input) {
                mHeap.add(element);
```

```
        }
        heapify();
}

private void heapify() {
        int N = mHeap.size();
        N /= 2;
        for (int i = N; i >= 0; i--) {
                heapDOWN(i);
        }
}

@Override
public T getMax() throws EmptyHeapException {
        if (mHeap.size() == 0) {
                // empty heap => we throw exception
                throw new EmptyHeapException();
        }
        else {
                T result = mHeap.get(0);
                swap(0,mHeap.size() - 1);
                mHeap.remove(mHeap.size() - 1);
                heapDOWN(0);
                return result;
        }
}

private void heapDOWN(int node){
        int sonLeft = node*2+1;
        int sonRight = node*2+2;
        int nextNode;
        int length = mHeap.size();
        if (sonLeft < length) {
                nextNode = sonLeft;

        if (sonRight < length && mHeap.get(sonLeft).
          compareTo(mHeap.get(sonRight)) < 0) {
                nextNode = sonRight;
        }
        if (mHeap.get(nextNode).compareTo(mHeap.get(node)) > 0){
                swap(nextNode,node);
                heapDOWN(nextNode);
        }
}

}
```

```java
protected void swap(int x, int y) {
        T aux = mHeap.get(x);
        mHeap.set(x, mHeap.get(y));
        mHeap.set(y, aux);
}

private void heapUP(int node) {
        int father = (node-1)/2;
        if (father >= 0 && mHeap.get(node).
                compareTo(mHeap.get(father)) > 0) {
                swap(node,father);
                heapUP(father);
        }
}

@Override
public void insert(T i) {
        mHeap.add(i);
        // getting the number we just added to the right position
        heapUP(mHeap.size() - 1);
}

@Override
public int getLength() {
        return mHeap.size();
}

}
```

(b) Implementation of BottomUpMaxHeap :

```java
package uk.ac.cam.tma33.alg.Tick1Star;

import java.util.ArrayList;
import java.util.List;

import uk.ac.cam.rkh23.Algorithms.Tick1.EmptyHeapException;

public class BottomUpMaxHeap<T extends Comparable<T> >
                        extends MaxHeap<T>{

private ArrayList<Integer> mPath;

public BottomUpMaxHeap(){
        super();
```

```
}

public BottomUpMaxHeap(List<T> input){
        super(input);
}

@Override
public T getMax() throws EmptyHeapException {
        if (mHeap.size() == 0) {
                // empty heap => we throw exception
                throw new EmptyHeapException();
        }
        else {
                mPath = new ArrayList<Integer>();
                T result = mHeap.get(0);
                swap(0,mHeap.size() - 1);
                mHeap.remove(mHeap.size() - 1);
                goDown(0);
                T root = mHeap.get(0);
                int N = mPath.size() - 1;
                for (int i = N; i > 0; i--) {
                        int node = mPath.get(i);
                        if (root.compareTo(mHeap.get(node))
                                        < 0) {
                                shiftUP(i);
                                mHeap.set(node,root);
                                break;
                        }
                }
                return result;
        }


}

private void shiftUP(int N) {
        for (int i = 0; i < N; i++) {
                mHeap.set(mPath.get(i), mHeap.get(mPath.get(i+1)));
        }
}

private void goDown(int node) {
        int sonLeft = node*2+1;
        int sonRight = node*2+2;
        int nextNode;
        mPath.add(node);
```

```
        if (sonLeft >= mHeap.size()) return;
        nextNode = sonLeft;
        if (sonRight < mHeap.size() && mHeap.get(sonLeft).
                compareTo(mHeap.get(sonRight)) < 0) {
                nextNode = sonRight;
        }
        goDown(nextNode);
}


}
```

note : i submitted the BottomUpMaxHeap, but it says that it throws an unexpected exception... I don't understand why. it works on my tests without any problem

2 Algorithms sheet

2.1 Exercise 2

(a) **Brute-force :** A brute-force strategy consists in just implementing the most inefficient algorithm for a specific problem. The solution has to be correct, though.
(b) **Divide-and-conquer :** This strategy is based on splitting the problem into more sub-problems, that are easily to solve individually. The divide-and-conquer strategy usually is associated with a logarithmic execution time.
(c) **Dynamic programming** is also based on splitting the problem in multiple sub-problems. Unlike the divide-and-conquer, this strategy aims to find a general formula for the answer, by applying it to small examples.
(c) **Backtracking** is a programming strategy that tries multiple solutions for a problem, until it reaches a valid one. As the name indicates, this strategy involves "walking back" to previous solutions, to make changes onto them.

2.2 Exercise 3

The sudoku implementation :

```
package uk.ac.cam.tma33.s11;

public class Sudoku {

public static boolean ok(long a, long b, long x) {
        if ((a&x) != 0) return false;
        if ((b&x) != 0) return false;
        return true;
```

```
}

private static boolean full(long[] a, long[] b) {
        long MAX = 0;
        for (int i = 1; i <= 9; i++) {
                MAX += (1<<i)*1L;
                // generating the value MAX
        }
        for (int i = 0; i < 8; i++) {
                if (a[i] != MAX || b[i]!=MAX) {
                        // then the sudoku is not solved
                        return false;
                }
        }
        return true;
}

public static boolean Solve(long[][] A, long[] cols,
                long[] rows, int x, int y) {
        if (x > 8) {
                // We are at the end of the board
                return full(cols,rows))
        }
        for (long number = 1; number <= 9; number++) {
                long noCode = 1 << number;
                if (ok(cols[y],rows[x],noCode)) {
                        // number is not on the current row/
                        //column so we can proceed
                A[x][y] = number;
                int newX;
                int newY;
                if (y >= 8) {
                        // we are at the end of the row
                        newY = 0;
                        newX = x + 1; // get to the next row
                }
                else {
                        newY = y + 1;// next column, same row
                        newX = x;
                }
                cols[y] += noCode; // we add the code to the row/
                rows[x] += noCode; // column, to know
                                //that we used this no
                // we try to find a number to fit the next cell

                boolean solved = Solve(A,cols,rows,newX,newY);
```

```
                        if (solved) return true;
                        cols[y] -= noCode; //we substract the code this time,
                        rows[x] -= noCode; // because we want
                                        //to try a different solution
                        }
                }
return false;
}

public static void main(String[] args) {
                long[][] gameBoard = new long[9][9]; // the gameboard
                long[] cols = new long[9];
                // cols[i] encodes the numbers on collumn i
                long[] rows = new long[9];
                // rows[i] encodes the numbers on row i
                for (int i = 0;i <= 8; i++) {
                        cols[i] = 0; // we initialise the cols
                        rows[i] = 0; // and rows arrays with 0
                }

                boolean solved = Solve(gameBoard,cols,rows,0,0);

                // printing the solution
                for (int i = 0; i <= 8; i++) {
                        for (int j = 0; j <= 8; j++){
                                System.out.print(Long.toString(
                                        gameBoard[i][j]) + " ");
                        }
                System.out.println();
                }
}

}
```

2.3 Exercise 5

(i) In the Foundations of Computer Science course, the strategy used to make change was based on a rather optimised backtracking algorithm, looking for any number that could be put in the result and trying it until the amount we had reached 0.

(ii) **Backtracking approach :**

```
package uk.ac.cam.tma33.s11;

public class MakeChange {

public int[] makeChange(int amt, int[] coins, int[] Sol) {
        int N = coins.length;
        if (amt == 0) {
                // job done, we finished
                return Sol;
        }
        else if (amt < 0) {
                // WHOOPS, too far ahead
                return null;
        }
        else {
                // we try to add one more coin to the solution
                int lng = Sol.length;
                // resizing the Sol array
                int[] aux = new int[lng + 1];
                for (int i = 0; i < lng; i++) {
                        aux[i] = Sol[i];
                }
                Sol = aux;
                // trying different coins to see if they work
                for (int i = 0; i < N; i++ ) {
                        if (amt - coins[i] >= 0 ) {
                                // we find a coin that
                                //could go in
                                Sol[lng] = coins[i];
                                int[] result = makeChange(
                                        amt-coins[i],coins,Sol);
                                if (result != null) return result;
                        }
                }
        }
return null;
}

}
```

This implementation has a complexity of $O(2^n)$, where n is the number of coins available. We have this complexity due to the fact that, for every coin we have to options : to use it or not to use it.

Dynamic-programming approach :

```
package uk.ac.cam.tma33.s11;

public class DynamicMakingChange {

public int makeChange(int amt, int[] coins) {
        int N = coins.length;
        int[][] Sol = new int[amt+2][N + 2];
        for (int i = 0; i <= N; i++) {
                Sol[i][0] = 1;
        }
        for (int i = 1; i <= amt; i++) {
                Sol[0][i] = 0;
        }
        for (int i = 1; i <= N; i++) {
                for (int j = 1; j <= amt; j++) {
                        if (j - coins[i-1] >= 0) {
                                // we can make change with
                                // this coin
                                Sol[i][j] = Sol[i-1][j] +
                                  Sol[i][j - coins[i-1]];
                        }
                        else {
                                // we can't use coin #(i-1)
                                Sol[i][j] = Sol[i-1][j];
                        }
                }

        }
return Sol[amt][N];
}

}
```

This approach has a complexity of $O(n * amt)$, as it has to go through every element in the matrix exactly one time.

2.4 Exercise 6

```java
package uk.ac.cam.tma33.s11;

public class MinMax {

private int getSmaller(int x, int y) {
        if (x < y) return x;
        else return y;
}

public int getMin(int[] a, int left, int right) {
        int mid = (left + right)/2;
        if (right < left) return Integer.MAX_VALUE;
        if (right - left < 1) {
                // return the minimum of the 2
                return getSmaller(a[right],a[left]);
        }
        else {
                // we continue to divide the interval
                int x = getMin(a,left,mid);
                int y = getMin(a,mid+1,right);
                return getSmaller(a[mid],getSmaller(x,y));
        }
}

private int getLarger(int x, int y) {
        if (x > y) return x;
        else return y;
}

public int getMax(int[] a, int left, int right) {
        int mid = (left + right)/2;
        if (right < left) return Integer.MIN_VALUE;
        if (right - left < 1) {
                // return the maximum of the 2
                return getLarger(a[right],a[left]);
        }
        else {
                // we continue to divide the interval
                int x = getMin(a,left,mid);
                int y = getMin(a,mid+1,right);
                return getLarger(a[mid],getLarger(x,y));
        }
}
```

```
}
```

The number of operations that each of the 2 methods ($getMin$ and $getMax$) executes is defined by the following recursive rule : $T(n) = T(n/2) + T(n/2) + 2, \forall n \in N, n \geq 2$.

So, $T(n) = 2^{k-1} + 2^k - 2$, where $k = log(n)$.

$T(n) = \frac{n}{2} + n - 2 = \frac{3n}{2} - 2 = O(n)$.