INSTITUTE OF TECHNOLOGY

Worst Case Run-Time Analysis of Burger Class changePatties() Operation

Siyuan Zhou - June 5, 2015

1 INTRODUCTION

In this document is a detailed *Worst Case run-time analysis* of the changePatties() operation of the Burger class implemented in lecture. For convenience the relevant code for this operation is included here with numbered lines:

public void changePatties(String pattyType) {

1 MyStack<Object> temp = new MyStack<Object>();

2 while (!burger.isEmpty()) {

3 Object ingredient = burger.pop();

4 if (ingredient instanceof Patties) {

5 try {ingredient = Class.forName(Burger.classPrefix + pattyType).newInstance();

} catch (InstantiationException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

6 temp.push(ingredient);

7 }while (!temp.isEmpty()) {

8 burger.push(temp.pop());

}

}

2 ANALYSIS

2.1 LINE-BY-LINE

In this section we will look at each line and provide a precise estimate for the number of operations carried out on that line. For reference we will count all declarations, assignments, integer and Boolean arithmetic, and similar operations to be single operations. We will use *constants* and *functions* to estimate the number of operations of methods we are unsure of. In these cases we will seek *upper bounds* because we are looking for an expression of the worst case run-time.

For each line we will state the number of operations on that line and justify it in a few points. Recall that the variable temp is of type MyStack. Let *c*0, *c*1 , *c*2 and *c*3 be upper bounds on the running times of push, pop, isEmpty and instanceof respectively.

1. There is an assignment. The cost of this line is 1.
2. This line has call to isEmpty. The cost of this line is *c*2.
3. There is a declaration, an assignment. The cost of this line is *c*1*+2*.
4. This line has call to instanceof. The cost of this line is *c*3.
5. There is an addition and an assignment. The cost of this line is *2*.
6. This line has a call to push. The cost of this line is *c*0*.*
7. This line has a call to isEmpty. The cost of this line is *c*2.
8. This line has a call to push and pop. The cost of this line is *c*0*+ c*1.

2.2 LOOPS

There are three some kind of while loops in this method consisting of lines 2-6, 7 and 8. The loop begins with the current ingredient equal to *n* where *n* is the size of the MyStack myBurger. The loop ends when either size becomes 0 or the correct ingredient is found for the pop item. The *worst case* is when the size is position 0. This means the loop executes for values *n* down to 1. For analysis purposes, because we are adding up all the costs, it does not matter if we sum the costs forwards or backwards.

Let *f* (*n*) be a function expressing the total cost of the first (3-5) while loop. We can express this sum as:

=

Let *c*4 = . Then the total cost of the while loop is *c*4 · *n*.

Let *f* (*n*) be a function expressing the total cost of the second (7,8) while loop. We can express this sum as:

=

Let *c*5 = . Then the total cost of the while loop is *c*5 · *n*.

2.3 TOTAL COST

The total cost of the changePatties method is *g* (*n*) = *c*0 + *c*2 + 1 + *c*4 · *n +*  *c*5 · *n* = *c*0 + *c*2 + 1 + n \* (*c*4  *+*  *c*5). Simplifying and substituting *a*= *c*0 + *c*2 + 1 and *b*= *c*4  *+*  *c*5 we have *g*(*n*)=*a*+*b*·*n*.

We can see that *g* (*n*) ∈ O(*n*).