Assignment 2

EECS 2011 – Fundamentals of data structures.

Rajkumar Balakrishnan Lakshmi - 213141197

10/20/17

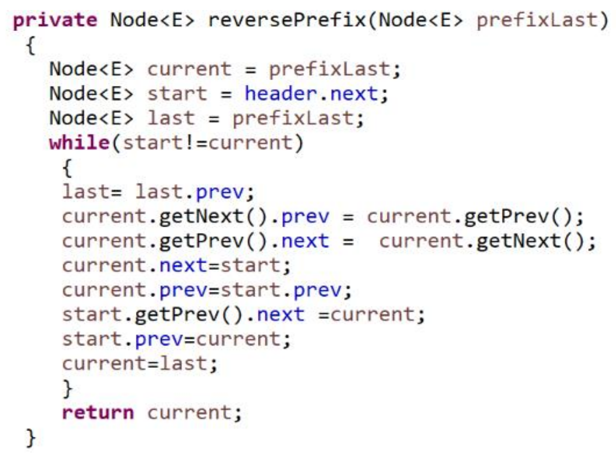
Problem 1: Pancake Sorting on Doubly Linked Lists

For this question, I could obtain the desired output using the Doubly Linked list class that was already provided. I created the PancakeSort.java class and copied the entire doubly linked list class into my class. Later I added the methods as specified in the question and worked on it respectively as stated below.

• getPrefixMaxNode(prefixLast) – This method iterates through a subset of the linked list and returns the node of the element which is greater than other elements of the subset(Header – prefixLast). Time complexity – 0(n).

• reversePrefix(prefixLast) – This method reverses the subset of the linked list and returns the current last position of the subset after modification. This is achieved by moving the elements one after the other for the last position to first. To do this I first store the node that is just before prefixlast into **last** and the node right next to header is stored in **start.** Then the prefix last node(**current)** is moved next to the header and the node **last** is moved next to node **current.** This continues until the node **start** is moved to the last position of the subset. Time complexity – 0(n).

Background Mechanism:



```
private Node<E> reversePrefix(Node<E> prefixLast)
 {
   Node<E> current = prefixLast;
   Node<E> start = header.next;
   Node<E> last = prefixLast;
   while(start!=current)
    {
    last= last.prev;
    current.getNext().prev = current.getPrev();
    current.getPrev().next =  current.getNext();
    current.next=start;
    current.prev=start.prev;
    start.getPrev().next =current;
    start.prev=current;
    current=last;
    }
   return current;
 }
```

Example: Doubly Linked List – {5, 8, 3, 2} prefixLast – 2

H – Header Red – start Blue - current

H 5 8 3 2 current = 2, start = 5, last = 3

H 2 5 8 3 current = 3, start = 5, last = 8

H 2 3 **5** **8** current = 8, start = 5, last = 5

H 2 3 8 **5** current = **5**, start = **5**, last = 2

Now both current and start positions clash or equalize, thus the loop stops and the output is achieved.

• buildList(E [] elements) - This method creates a doubly linked list out of the elements obtained in the parameter and returns it. Here I just traversed through the list of elements and added them into the doubly linked list. Time complexity – 0(n).

• pancakeSort() - This Method basically sorts the doubly linked list by applying the pancake sort mechanism. This method calls the reverseprefix(**prefixLast**) method twice for every iteration of the loop. The first-time node of the **max** element is passed as argument and second time the current prefix Last is passed as the argument. This way the list is sorted using the pancake sort mechanism.

Example: Doubly Linked List – {1, 2, 4, 3}

Round 1: 1 2 **4** 3 → **4** 2 1 3 → 3 1 2 **4**

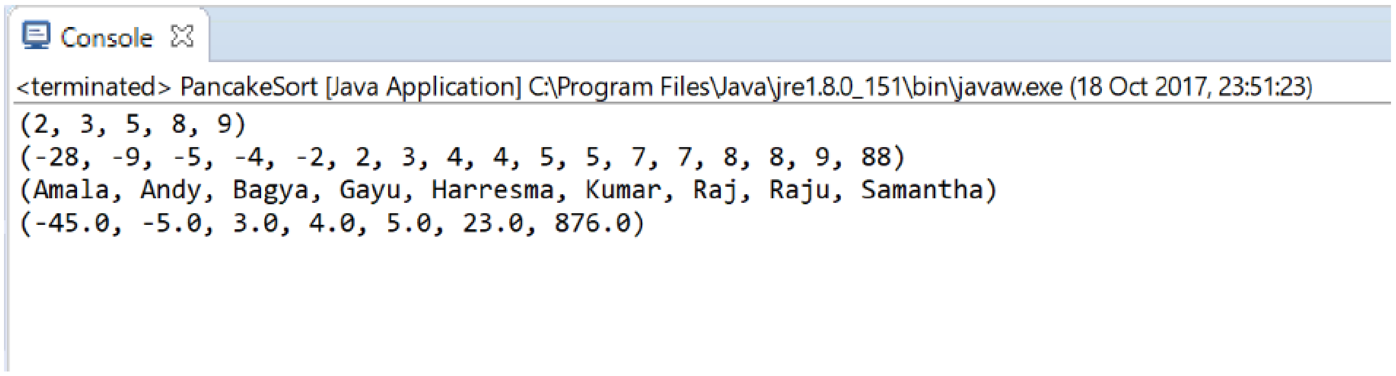
Round 2: **3** 1 2 **4** → **3** 1 2 **4** → **1 2 3 4**

Test Cases:

Input:

```
public static void main(String[] args)
 {

     Integer[] i = {5,8,3,9,2};
     PancakeSort<Integer> linkedList = buildList(i);
     linkedList.pancakeSort();
     System.out.println(linkedList.toString());

     Integer[] i2 = {8,3,5,4,8,9,2,4,7,5,-9,-5,-4,-2,-28,7,88};
     PancakeSort<Integer> linkedList2 = buildList(i2);
     linkedList2.pancakeSort();
     System.out.println(linkedList2.toString());

     String[] i3 = {"Raj","Kumar","Harresma","Gayu","Bagya","Amala","Samantha","Andy","Raju"};
     PancakeSort<String> linkedList3 = buildList(i3);
     linkedList3.pancakeSort();
     System.out.println(linkedList3.toString());

     Double[] i4 = {4.0,5.0,-45.0,-5.0,23.0,876.0,3.0};
     PancakeSort< Double> linkedList4 = buildList(i4);
     linkedList4.pancakeSort();
     System.out.println(linkedList4.toString());
 }
```

Output:


Console

```
<terminated> PancakeSort [Java Application] C:\Program Files\Java\jre1.8.0_151\bin\javaw.exe (18 Oct 2017, 23:51:23)
(2, 3, 5, 8, 9)
(-28, -9, -5, -4, -2, 2, 3, 4, 4, 5, 5, 7, 7, 8, 8, 9, 88)
(Amala, Andy, Bagya, Gayu, Harresma, Kumar, Raj, Raju, Samantha)
(-45.0, -5.0, 3.0, 4.0, 5.0, 23.0, 876.0)
```

Problem 2: A Recursive Function

```java
public static int f ( int n )
{
    if ( n > 1000 )
    {
    return n - 4;
    }

return f ( f (n + 5) );
}
```

a)

Input	Output
f (1001)	**997**
f (1000)	**997**
f (0)	**997**

b) **&** c) For all arbitrary integer value inputs n, the call to f(n) terminates after a finite number of recursive steps. The output falls under either of the two cases stated below:

- Case 1: If **n > 1000** Output is **n – 4**.
  Ex: f (1005) - returns the value 1004. This happens because it satisfies the if condition as shown in the code snippet above and right away returns the value (**n – 4**) without moving on to the next line.

- Case2: If **$n \leq 1000$** Output is **997**.
  Ex: f (998) – Here it does not satisfy the if condition, so it moves on to the next line where the function itself is called recursively i.e. f (f (n+5)), this eventually would result in returning the value **997.**

Background Mechanism:

f (998) Where n =998.

1. f(f(1003)

1.a - **f(999)** - Dosn't satisfy the if condition.

2. f(f(1004)

2.a - **f(1000)** - Dosn't satisfy the if condition.

3. f(f(1005)

3.a - **f(1001)** - Satisfies the if condition.

Returns **997**

So from the above example it can be understood that for any integer value n less than or equal to 1000 produces the result **997** at a finite number of steps (depending on the n input value).

Problem 3: Decimal to Binary Conversion

The DecimalToBinary.java classes main purpose is to convert the given decimal value to the binary version of it. This is executed though two different functionalities in the class file.

• Iter10To2(n) - This Method converts the given positive integer to binary format using the iteration mechanism. In this method stack is being used to keep track of the output. It first finds n modulus of 2 and pushes the result into stack and then divides the number itself by 2 and iterates over the loop. This iteration and stack pushing continues until the number is reduced to less than zero. Now all the required value being stored in stack is popped one after the other printing out the converted binary version of the integer.
Time complexity – 0(log n).

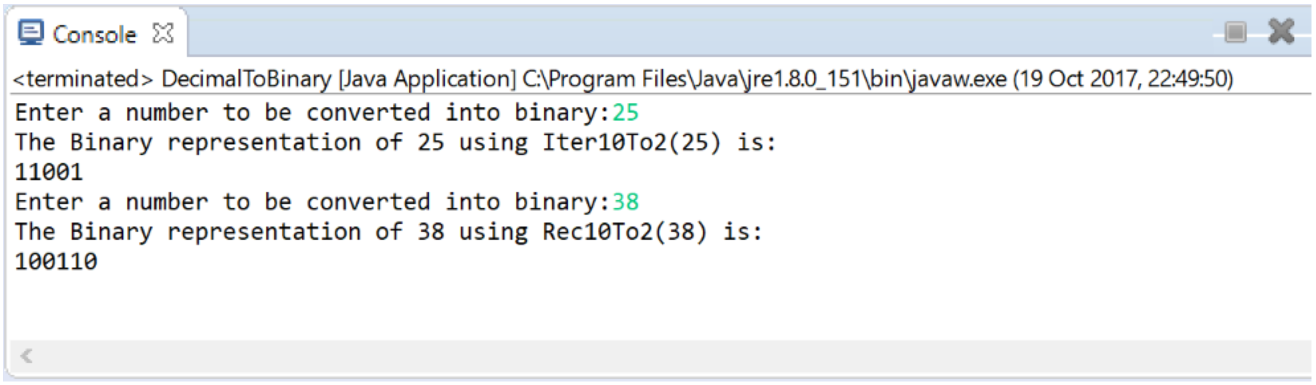
• Rec10To2(n) - This Method converts the given positive integer to binary format using the recursive mechanism. This is very similar to the iterative method but here the recursive mechanism is applied. The function keeps calling itself recursively for (n/2) until the number n is reduced to less than zero. Then it prints out n modulus of 2. Since it uses recursive mechanism the last calculated modulus value gets printed out first and followed by the rest.
Time complexity – 0(log n).

Test Cases:

Input:

```
public static void main(String[] args)
{
    Scanner input = new Scanner(System.in);

    System.out.print("Enter a number to be converted into binary:");
    int num = input.nextInt();
    System.out.println("The Binary representation of "+num+ " using Iter10To2("+num+") is: ");
    DecimalToBinary.Iter10To2(num);

    System.out.print("\nEnter a number to be converted into binary:");
    int num2 = input.nextInt();
    System.out.println("The Binary representation of "+num2+ " using Rec10To2("+num2+") is: ");
    DecimalToBinary.Rec10To2(num2);

    input.close();
}
```

Output:



```
<terminated> DecimalToBinary [Java Application] C:\Program Files\Java\jre1.8.0_151\bin\javaw.exe (19 Oct 2017, 22:49:50)
Enter a number to be converted into binary:25
The Binary representation of 25 using Iter10To2(25) is:
11001
Enter a number to be converted into binary:38
The Binary representation of 38 using Rec10To2(38) is:
100110
```

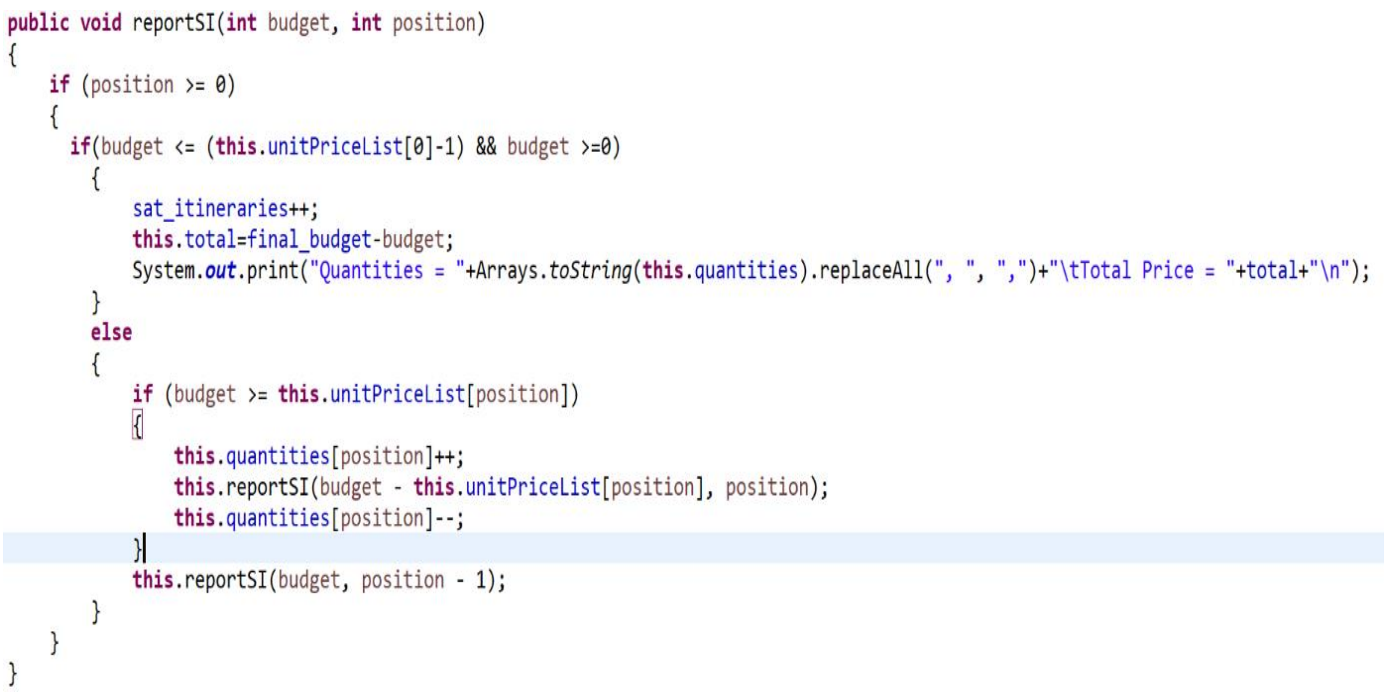
Problem 4. Budgeted Itinerary

I fist created the class Saturated Itineraries and added the required methods as specified in the question. The main purpose of this class is to efficiently find and print all saturated itineraries for a given list of products prices and the budget for it. The class has two methods, among which one method is a overloaded version of the other. The methods use pure recursive to find and print out the itineraries. The functioning of the methods is:

• reportSI (unitPriceList, budget) - This method is the main functional method that calls its overloaded version to obtain the output. It also initialises the arrays and other variables. It prints out the head and tail part of the output and body part is printed upon call to its overloaded version. Upon call to its overloaded version the argument passed to it is the budget of the list and the length of the list to denote the position in the list to start from. Time complexity – 0(budget^(items of the list)).

```
        this.reportSI(budget, this.unitPriceList.length - 1);
```

• reportSI (budget, position) - This overloaded method is where the main recursion mechanism is carried out. This method calculates all possible saturated itineraries for a given budget and its list. The method keeps calling itself recursively until all possible itineraries is evaluated, but only prints out the saturated list. For this to be carried out it goes through a sequence of if and else conditions as shown in the code snippet below.



```
public void reportSI(int budget, int position)
{
    if (position >= 0)
    {
      if(budget <= (this.unitPriceList[0]-1) && budget >=0)
        {
            sat_itineraries++;
            this.total=final_budget-budget;
            System.out.print("Quantities = "+Arrays.toString(this.quantities).replaceAll(", ", ",")+"\tTotal Price = "+total+"\n");
        }
        else
        {
            if (budget >= this.unitPriceList[position])
            {
                this.quantities[position]++;
                this.reportSI(budget - this.unitPriceList[position], position);
                this.quantities[position]--;
            }
            this.reportSI(budget, position - 1);
        }
    }
}
```

Ex:
Position 0 1
List {2, 3} **Budget=5**

reporSI (5,1)	Budget =5; Position =1;
↳ if (5>=3)	Quantities {0,1};
↳ reporSI (2,1)	Budget =2; Position =1;
↳ if (2>=3)	Quantities {0,1};
↳ reporSI (2,0)	Budget =2; Position =0;
↳ if (2>=2)	Quantities {1,1};
↳ reporSI (0,0)	Budget =2; Position =0;
Now since the budget is equal to zero it Prints out: }	**Quantities {1,1};**

Now the recursion returns and executes the next statement after reporSI (0,0). Thus, quantities become {0,1}; and reporSI (2, -1). This doesn't pass the if condition and eventually returns and executes from the left-over portion of the program. By this recursive mechanism all possible sets of itineraries are calculated and the ones that satisfy the condition of saturation is printed out.

Test Cases:

Input1:

```
public static void main(String[] args)
  {
     int[] unit = {2,3};
     SaturatedItineraries  si = new SaturatedItineraries();
     si.reportSI(unit,5);
  }
```

Output1:

```
Unit Price List = [2,3]    Budget = 10

The Saturated Itenaries are:
Quantities = [0,3]  Total Price = 9
Quantities = [2,2]  Total Price = 10
Quantities = [3,1]  Total Price = 9
Quantities = [5,0]  Total Price = 10
The total number of saturated itenaries are : 4
```

Input2:

```
public static void main(String[] args)
{
   int[] unit = {3, 5, 7};
   SaturatedItineraries  si = new SaturatedItineraries();
   si.reportSI(unit,88);
}
```

Output2:

```
Unit Price List = [3,5,7]  Budget = 88

The Saturated Itenaries are:

Quantities = [1,0,12]     Total Price = 87
Quantities = [0,2,11]     Total Price = 87
Quantities = [2,1,11]     Total Price = 88
Quantities = [3,0,11]     Total Price = 86
Quantities = [1,3,10]     Total Price = 88
Quantities = [2,2,10]     Total Price = 86
Quantities = [4,1,10]     Total Price = 87
Quantities = [6,0,10]     Total Price = 88
Quantities = [0,5,9]      Total Price = 88
Quantities = [1,4,9]      Total Price = 86
Quantities = [3,3,9]      Total Price = 87
Quantities = [5,2,9]      Total Price = 88
Quantities = [6,1,9]      Total Price = 86
Quantities = [8,0,9]      Total Price = 87
Quantities = [0,6,8]      Total Price = 86

.

.

.

.

.

.

Quantities = [21,5,0]     Total Price = 88
Quantities = [22,4,0]     Total Price = 86
Quantities = [24,3,0]     Total Price = 87
Quantities = [26,2,0]     Total Price = 88
Quantities = [27,1,0]     Total Price = 86
Quantities = [29,0,0]     Total Price = 87

The total number of saturated itenaries are : 127
```