```
/*********************************************************
 * EECS2011: Fundamentals of Data Structures, Fall 2016
 * Assignment 2, a2sol.pdf
 * Student Name: Khushal Patel
 * Student CSE account: york18
 * Student ID number: 214037618
 *********************************************************/
```

(1) Coin Counting:

I/O result:

```
“Enter an amount in cents:
17
This amount can be changed in the following ways:
      1) 17 pennies
      2) 1 nickel, 12 pennies
      3) 2 nickels, 7 pennies
      4) 3 nickels, 2 pennies
      5) 1 dime 7 pennies
      6) 1 dime, 1 nickel, 2 pennies


Enter an amount in cents:
25
This amount can be changed in the following ways:
      1) 25 pennies
      2) 1 nickel, 20 pennies
      3) 2 nickels, 15 pennies
      4) 3 nickels, 10 pennies
      5) 4 nickels, 5 pennies
      6) 5 nickels
      7) 1 dime 15 pennies
      8) 1 dime, 1 nickel, 10 pennies
      9) 1 dime, 2 nickels, 5 pennies
      10) 1 dime, 3 nickels
      11) 2 dimes 5 pennies
      12) 2 dimes, 1 nickel
      13) 1 quarter”
```

(2) Hypercube.

Running time for recursiveWalk: The first half of each n-bit code is generated by directly copying the values of the previous (n-1)-bit code sequence and prepending a leading "0" to it. The second recursive method(reverseRecursiveWalk) appends second half of each n-bit code sequence by copying the values of the previous (n-1)-bit code sequence in reverse order in which they appear in (n-1)-bit sequence and pre-pending a leading "1" to it. It produces and stores exactly 2^n values. The time complexity of this implementation is **$O(2^n)$.**

Running time for iterativeWalk: We need to generate 2^n numbers, so the complexity is **$O(2^n)$**. Same as recursive.

I/O result:

```
"Walking through a hypercube of 3D using recursiveWalk:
000
001
011
010
110
111
101
100


Walking through a hypercube of 4D using iterativeWalk:
0000
0001
0011
0010
0110
0111
0101
0100
1100
1101
1111
1110
1010
1011
1001
1000"
```

(3) Augmented Stack with getMin

Idea behind the getMin method: Every time an element is pushed into stack, check if that element is less than the current minimum element. If stack was empty to begin with then add the current element to a private stack that records all the minimum element and also make this element current min. Else, only push this element into stack. (Don't do anything with min.)

Now, when you pop an element from the stack and that element happens to be the minimum element of the stack, you pop that element from the stack and also pop an element from the private stack that has been recording all the minimum elements. After all the popping is done, the top element of the private stack is the minimum element to the stack passed as parameter.

The code for push and pop along with private stack and min field:

```
public class AugmentedStack<Item extends Comparable<Item>>
{
	private Item min;
	private Stack<Item> minStack = new Stack<Item>(); //Used to keep all
the			min elements


	/**
	 * Adds the item to this stack.
	 *
	 * @param item the item to add
	 * @param S Stack you want this item to add
	 *
	 */
	public void push(Item item, Stack<Item> S)
	{
		if(S.isEmpty())
		{
			min = item; // empty stack so the first element is the
minimum
			minStack.push(item); // also add it to minStack
		}
		else
		{
			Item MinNow = this.min;
			if(MinNow.compareTo(item) > 0) // if element to be added is
less than currentMin
			{
				this.min = item;
				minStack.push(item); // also push it to minStack
```

```
                }
            }
            S.push(item);
        }

        /**
         * Removes and returns the item most recently added to this stack.
         *
         * @return the item most recently added
         * @param S Stack you want to remove an element from
         * @throws NoSuchElementException
         *              if this stack is empty
         */
        public Object pop(Stack<Item> S)
        {
            Item i = S.peek();
            if(i.equals(this.getMin(S))) // if element we are popping is the
minimum element
            {
                minStack.pop(); // remove that element from the minStack
which is obviously the first element of it.
                this.min = minStack.peek(); // now the top element of the
private stack minStack is the minimum element.
            }
            return S.pop();
        }
```

Testing getMin code and output:

```
"public static void main(String[] args)
        {
          AugmentedStack<String> augStack = new AugmentedStack<String>();
          Stack<String> s = new Stack<String>();
          Stack<String> s2 = new Stack<String>();

          augStack.push("D", s);
          augStack.push("B", s);
          augStack.push("C", s);
          augStack.push("A", s);
          augStack.pop(s); //poping "A" so that min element in the s is "B"
          String str = "B";
          System.out.println("Testing getMin Test 1...");
          if(augStack.getMin(s).equals(str))
          {
              System.out.println("Passed");
          }
          else
          {
              System.out.println("Failed\nYour minimum element is: " +
augStack.getMin(s));
          }

          System.out.println("Testing getMin Test 2...");
          augStack.push("A", s2);
          augStack.push("A", s2);
          augStack.pop(s2);
          String str2 = "A";
          if(augStack.getMin(s2).equals(str2))
          {
              System.out.println("Passed");
          }
          else
          {
              System.out.println("Failed\nYour minimum element is: " +
  augStack.getMin(s2));
          }

      }”
```

Output:

```
Testing getMin Test 1...
Passed
Testing getMin Test 2...
Passed
```

As you can see, all the method in this class takes O(1) time.