Data Structure

Recitation VI
Topic

- Recursion
- Lab 3
What is recursion?

- The ability of a function to call itself. This is called Recursion.

- Examples that we can use recursion: factorial, Fibonacci number...
Factorial Example

- Factorial example:
  - $5! = 5 \times 4 \times 3 \times 2 \times 1$
  - $4! = 4 \times 3 \times 2 \times 1$
  - $3! = 3 \times 2 \times 1$
  - $2! = 2 \times 1$
  - $1! = 1$ this is base case
Factorial Example

- 5! = 5 * 4! = 120
- 5! = 5 * 4 * 3! = 24
- 5! = 5 * 4 * 3 * 2! = 6
- 5! = 5 * 4 * 3 * 2 *1! = 2
- 5! = 5 * 4 * 3 * 2 *1! * 1! = 1

A for loop will solve this problem. We can also solve this recursively. You reduce the size of the problem each time you solve for a factorial.
Factorial Example

- Recursive methods:
- Method that invokes itself.
- The arguments passed to the recursion take us closer to the solution with each call.
Factorial Example

- The simplest program to calculate factorial of a number is a loop with a product variable.

- Instead, it is possible to give a recursive definition for Factorial as follows:
  1) If $n \leq 1$, then Factorial of $n = 1$
  2) Otherwise, Factorial of $n = \text{product of } n \text{ and } \text{Factorial of } (n-1)$

- The important thing to remember when creating a recursive function is to give an 'end-condition'.
public static int factorial(int n)
{
    if (n <= 1) // base case
        return 1;
    else // general case
        return (n * factorial(n - 1));
}
}
Factorial Example

- Start handling the base case first, compiler hangs on to the value of the method but because it can't solve it, it starts building a stack and tries to solve it by continuing to call itself and keeps a record in memory called an activation record.

- $1 \times \text{fact}(0)$ this returns 1 as above and now stack executes the returns.

- $2 \times \text{fact}(1)$ this returns 2.

- $3 \times \text{fact}(2)$ this returns 6.

- $4 \times \text{fact}(3)$ this returns 24.

- $5 \times \text{fact}(4)$ this returns 120.

- This generates a lot of memory, more than the for loop. Efficiency is slower.
Tower of Hanoi
Tower of Hanoi

- Rule:
  1. Move only one disk at a time from one peg to another.
  2. Never put a disk on top of a smaller one.

- To move 1 disk (our base case):
  Move 1 disk from start tower to destination tower and we are done.

- To move 2 disks:
  Move smaller disk from start tower to intermediate tower, move larger disk from start tower to final tower, move smaller disk from intermediate tower to final tower and we are done.

- To move n disks (or think of, say, 3 disks):
  Solve the problem for n - 1 disks (i.e. 2 disks) using the intermediate tower instead of the final tower (i.e. get 2 disks onto the intermediate tower). Then, move the biggest disk from start tower to final tower. Then again solve the problem for n - 1 disks but use the intermediate tower instead of the start tower (i.e. get the 2 disks onto the final tower using the start tower as the intermediate tower).
Tower of Hanoi

- Order:
  - 4-3-2-1:FT
  - 3-2-1:FS
  - 4:FT.
  - 3-2-1:ST

- Another hanoi problem with 3 disks!

- Move a bunch of disks from one peg to another, first, move all but the bottom disk to the remaining (spare) peg, then move the bottom disk to the desired peg, and finally move the all-but-the-bottom pile from the spare to the desired peg.

The Tower of Hanoi

```java
public class Hanoi1 {
    public static void main (String[] args) {
        hanoi(4, 'F', 'T', 'S');
    }

    public static void hanoi(int n, char from, char to, char spare) {
        if (n>0) {
            hanoi(n-1, from, spare, to);
            System.out.println(n + ":" + from + to);
            hanoi(n-1, spare, to, from);
        }
    }
}
```

- Move the top N - 1 disks from From to Spare (using to as an intermediary peg)
- Move the bottom disk from From to To
- Move N - 1 disks from Spare to To (using From as an intermediary peg)
Another Example

A child is running up a stair with n steps, and can hop either 1 steps, 2 steps or 3 steps at a time. Implement a method to count how many possible ways the chair can ran up the stair.
Another Example

public int countWays()
{
    if(n < 0) {
        return 0;
    } else if (n ==0 ){
        return 1;
    } else {
        return countWays(n - 1) + countWays(n - 2) + countWays(n - 3);
    }
}
Lab 3

- status() Method
- toString() Method
- diff() Method
- Test Drivers
public class ArrayStack<T> implements BoundedStackInterface<T> {
    protected final int DEFCAP = 100; // default capacity
    protected T[] stack;              // holds stack elements
    protected int topIndex = -1;      // index of top element in stack

    public ArrayStack() {
        stack = (T[]) new Object[DEFCAP];
    }

    public ArrayStack(int maxSize) {
        stack = (T[]) new Object[maxSize];
    }
}
public void push(T element)
   // Throws StackOverflowException if this stack is full,
   // otherwise places element at the top of this stack.
   {
      if (!isFull())
      {
         topIndex++;
         stack[topIndex] = element;
      }
      else
         throw new StackOverflowException("Push attempted on a full stack.");
   }

public void pop()
   // Throws StackUnderflowException if this stack is empty,
   // otherwise removes top element from this stack.
   {
      if (!isEmpty())
      {
         stack[topIndex] = null;
         topIndex--;
      }
      else
         throw new StackUnderflowException("Pop attempted on an empty stack.");
   }
public T top()
// Throws StackUnderflowException if this stack is empty,
// otherwise returns top element from this stack.
{
  T topOfStack = null;
  if (!isEmpty())
    topOfStack = stack[topIndex];
  else
    throw new StackUnderflowException("Top attempted on an empty stack.");
  return topOfStack;
}

public boolean isEmpty()
// Returns true if this stack is empty, otherwise returns false.
{
  if (topIndex == -1)
    return true;
  else
    return false;
}

public boolean isFull()
// Returns true if this stack is full, otherwise returns false.
{
  if (topIndex == (stack.length - 1))
    return true;
  else
    return false;
}
Return a string of stack status:
- The capacity of the stack. ArrayStack\(<T>\) is a bounded stack, so each instance has a fixed capacity.
- The number of used and available slots.
- An indication of whether the stack is full, empty, or neither.
- If the stack is not full, the number of pushes that would fill it.
- If the stack is not full, the number of pops that would empty it.
- If the stack is neither full nor empty, both values should be printed.
**toString()**

- It should contain all the information produced by `status()`. Naturally, `toString()` should call `status()` to obtain that information.

- It should contain the contents of every slot in the stack.

- It should make clear which entry is on top, which is 2nd, etc.

- It should be readable.
Driver 3A

- Write a class TestInteger with a main() method that tests MyArrayStack<Integer>.

- Write a class TestDouble with a main() method that tests MyArrayStack<Double>.

- Write a class TestString with a main() method that tests MyArrayStack<String>.

- You must make no changes to MyArrayStack<T>, when testing with different T's.
Driver 3B

- Imports the stringLogs package from the book's library.
- Declares, but does not (indeed cannot) create 3 StringLogInterface's sLI1, sLI2, sLI3;
- Create (using new) a LinkedStringLog and 2 ArrayStringLog's. Each stringLog should have a unique name field and each must contain at least 4 log entries. Assign the linked version to sLI1 and the array versions to sLI2 and sLI3.
- Creates a MyArrayStack<StringLogInterface> object called myASSLI. Again, you make no changes to MyArrayStack<T>.
Driver 3B

- Pushes the first two of the StringLogs from part iii onto myASSLI.
- Prints the value of myASSLI.toString().
- Pops myASSLI.
- Prints the value of myASSLI.toString().
- Pushes the remaining StringLog from part iii onto myASSLI.
- Prints the value of myASSLI.toString().
Diff()

- Write a new MyArrayStack\langle T\rangle instance method diff() accepting one parameter, another MyArrayStack\langle T\rangle, and producing a String result. diff() prints nothing.

- As its name suggests diff(mAST) produces a string describing the differences between the instance object this and the parameter mAST. For example,

- It should say whether the parameter has more, fewer, or the same number of slots as the instance object. This comparison is between the two capacities.

- It should say whether the parameter has more, fewer, or the same number of in-use slots as the instance object. This comparison is between the current sizes of the stacks (note that top is quite useful here).

- For the slots that are in use in both the instance and the parameter, it should say whether the corresponding slots in the two stacks are equal using the equals() method guaranteed to exist for class T.