Recursion

- **Definition**:
  - A procedure or function that calls itself, directly or indirectly, is said to be recursive.

- **Why recursion**?
  - For many problems, the recursion solution is more natural than the alternative non-recursive or iterative solution
  - It is often relatively easy to prove the correction of recursive algorithms (often prove by induction).
  - Easy to analyze the performance of recursive algorithms. The analysis produces recurrence relation, many of which can be easily solved.

- **Format of a recursive algorithm**:

  - Algorithm name(parameters)
    Declarations;
    Begin
        if trivial case)
        then do trivial operations
        else begin
            - one or more call name(smaller values of parameters)
            - do few more operations: process the sub-solution(s).
        end;
    end;

- **Taxonomy:**

  - **Direct Recursion:**
    - It is when a function refers to itself directly
  - **Indirect Recursion:**
    - It is when a function calls another function which refer to it.
  - **Linear Recursion:**
    - It is when one a function calls itself only once.
  - **Binary Recursion:**
    - A binary-recursive routine (potentially) calls itself twice.

- **Examples**
  - **Printing a stack:**
    - **Recursive method to print the content of a stack**

Z
C
B
A

➔

Printing the stack

Z	**C**	**B**	**A**

```
public void printStackRecursive() {
 if (isEmpty())
        System.out.println("Empty Stack");
 else{
        System.out.println(top());
        pop();
        if (!isEmpty())
               printStackRecursive();
 }
}
```

  - **Palindrome:**

    int function Palindrome(string X)

    Begin

        If Equal(S,StringReverse(S))

        then return TRUE;

        else return False;

    end;

- **Reversing a String:**
  - **Pseudo-Code:**

```
string function StringReverse(string S)
/* Head(S): returns the first character of S */
/* Tail(S): returns S without the first character */
begin
        If (Length(S) <=1)
        then return S;
        else
                return (concat(StringReverse(Tail(S)) & Head(S));
        endif;
end;
```

  - **Java Code:**

```
public static String reverse1(String s) {
      //BASIS CASE
      if (s.length() == 0)
         return s;
        //RECURSIVE STEP (notice use of empty String to do
conversion of characters)
      return ("" + s.charAt(s.length() - 1) +
            reverse1(s.substring(0, s.length() - 1)));
   }
```

- **Performance**

  - **Definition**: A recurrence relation of a sequence of values is defined as follows:
    - (B) Some finite set of values, usually the first one or first few, are specified.
    - (R) The remaining values of the sequence are defined in terms of previous values of the sequence.

  - **Example**:

    - The familiar sequence of factorial is defined as:
      - (B) FACT(0) = 1
      - (R) FACT(n+1) = (n+1)*FACT(n)

  - **Time Complexity:**

    - The analysis of a recursive algorithm is done using recurrence relation of the algorithm.

    - **Example 1:**

      - The time complexity of StringReverse function:
        - Let T(n) be the time complexity of the function where n is the length of the string.
        - The recurrence relation is:

          (B) n=1 let T(n)=1
          (R) n>1 let T(n)=T(n-1)+1

- Solution:

$$T(n) = T(\cancel{n-1})+1$$
$$T(\cancel{n-1}) = T(\cancel{n-2})+1$$
$$T(\cancel{n-2}) = T(\cancel{n-3})+1$$

...

$$T(\cancel{3}) = T(\cancel{2})+1$$
$$\cancel{T(2)} = T(1)+1$$

$$T(n) = T(1)+1+1+1+...+1 = T(1)+(n-1) = n$$

===> T(n) = O(n

- **Example 2:**

$$T(n)=\begin{cases} 2T(\frac{n}{2})+C & if\ n>1 \\ C & if\ n=1 \end{cases}$$

Assume that n = 2^k

$$\begin{aligned}
T(n) &= 2\cancel{T(n/2)}+C \\
\cancel{2T(n/2)} &= \cancel{2^2T(n/4)}+2C \\
\cancel{2^2T(n/4)} &= \cancel{2^3T(n/8)}+ 2^2C \\
&\;\;\vdots \\
\cancel{2^{k-2}T(n/2^{k-2})} &= \cancel{2^{k-1}T(n/2^{k-1})}+ 2^{k-2}C \\
\cancel{2^{k-1}T(n/2^{k-1})} &= 2^kT(n/2^k)+ 2^{k-1}C
\end{aligned}$$

$T(n) = 2^kT(1)+C+2C+2^2C +\ldots+2^{k-2}C+2^{k-1}C =$

$T(n) = 2^kC +C(1+2+2^2 +\ldots+2^{k-2}+2^{k-1})$

$T(n) = nC +C\sum_{i=0}^{k-1}2^i = \frac{2^{(k-1)+1}-1}{2-1} = \frac{2^k}{1} = 2^k$

$T(n) = nC +C\frac{2^{(k-1)+1}-1}{2-1}$

$T(n) = nC +C\frac{2^k}{1}$

$T(n) = nC +C\,2^k$

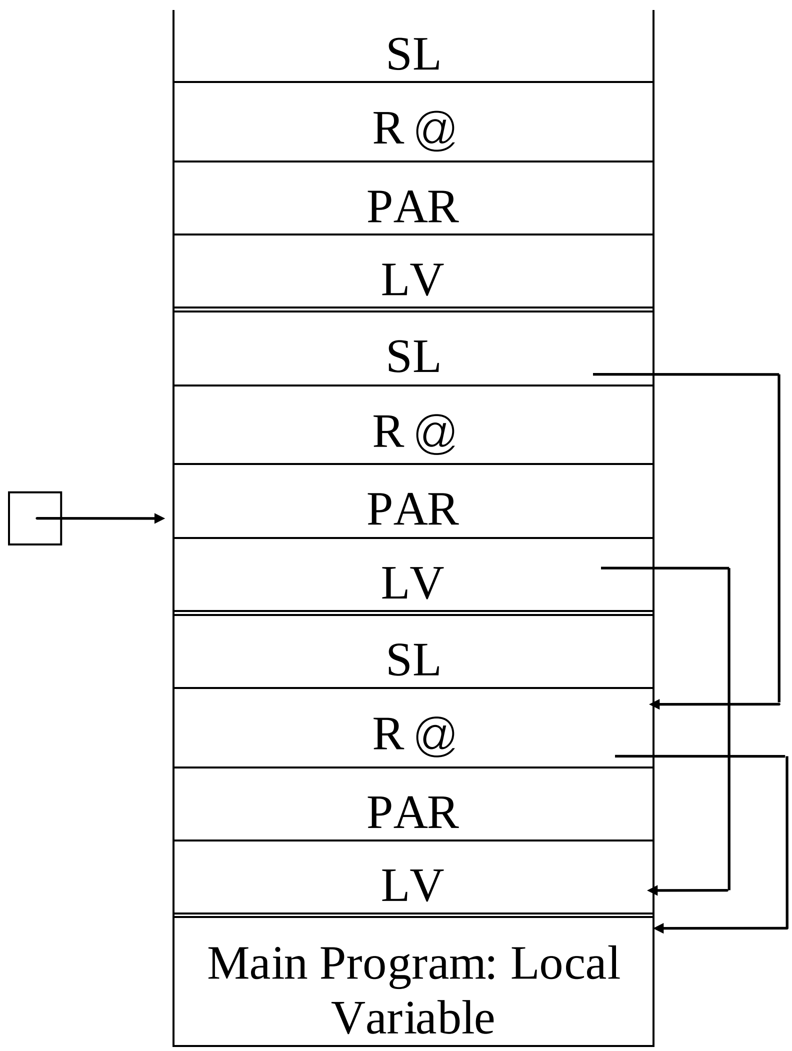
$T(n) = nC +nC=2nC$

==> $T(n) =O(n)$

- **Space Complexity:**

  - Each recursive call requires the creation of an activation record
  - Each activation record contains the following:
    - Parameters of the algorithm (PAR)
    - Local variable (LC)
    - Return address (R @)
    - Stack link (SL)

  - **Example**:



- Complexity:
    - Let

      P: parameters

      L: local variables

      2: SL and R @

      n: is the maximum recursive depth

    ➔ space= n*(P+L+2)

- **Disadvantages**:

    - Recursive algorithms require more time:
        - At each call we have to save the activation record of the current call and Branch to the code of the called procedure
        - At the exit we have the recover the activation record and return to the calling procedure.
        - If the depth of recursion is large the required space may be significant.

- **Lab Assignments:**
  - What is the time complexity of the following function:

$$T(n) = \begin{cases} T(\frac{n}{2}) & n > 1 \\ C & n = 1 \end{cases}$$

  - What is the time complexity of the following function:

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + n & n > 1 \\ C & n = 1 \end{cases}$$

  - Write a recursive function that returns the total number of nodes in a singly linked list.