EE 441 Data Structures Lecture 5

Recursion

- How to compute the power function: x^n
  - Solution 1: multiply x by itself n times:
    - $2^3=2*2*2=8$
    - Now compute 2^4, repeat all the previous multiplications: $2^4=2*2*2*2=16$
  - Solution 2: Use the previous result with a smaller argument to arrive at the answer

    $x^n=1$ if $n=0$

    $x^n=x*x^{(n-1)}$ if $n>0$
  - We use a smaller power to compute another → Recursive definition of the function

Recursive Functions

- Some calculations have recursive character:
  - $x^n = x * x^{n-1}$
  - $N! = N * (N-1)!$
- To be able to implement such algorithms, we write functions that call themselves, i.e., *recursive functions*
- Recursion
  - Method/Function calls itself
  - Each call is closer to "Base Case"
    - Base Case == Termination Condition
  - Each call == Loop iteration
  - Each call subject to "stacking"

Recursive Function Call

```
... doIt(.............)
{
   if (there is data)                    // EOF processing
   {
        do something
        ...
        doIt(.........)   <--  The recursive call...
                               doIt( ) calls itself...
   }
}
```

Input Argument of a Recursive Function

```
... doIt(... int count ...)
{
   if(count < terminalValue)
   {
        do something
        ...
        doIt(... count + 1 ...)
   }
}
```

The recursive call...

doIt( ) calls itself...

The value of count approaches the terminalValue each call

E.g. Filling an Array - Recursion

```
// int howmany is "global" and initialized to 0

... void getinput(... array[ ] ) ...
{
    int num;
    fin >> num;
    if (!fin.eof( )...)
    {
        array[howmany] = num;
        howmany++;
        getinput(array);
    }
}
```

E.g. Finding the Sum - Recursion

```
// sum is declared globally
... void findSum(int array[ ], int x, int howmany)...
{
    if(x < howmany)
    {
            sum += array[x];
          findSum(array, x+1, howmany);
    }
}
```

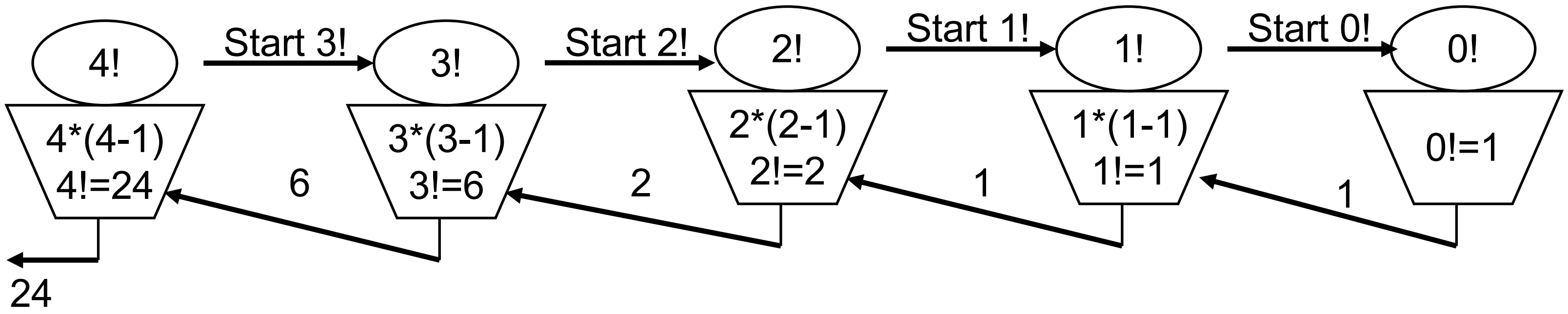
The value of x approaches howmany each call

- An algorithm is defined recursively by divide and conquer design:
  - Recursive step: Partition the problem into smaller sub-problems that are solved by using the same algorithm
  - Stopping condition: Partitioning terminates when we reach simpler sub-problems that cannot be solved with that algorithm

- E.g. Power function:
  - Stopping condition: $x^n=1$ if $n=0$
  - Recursive step (general condition): $x^n=x*x^{(n-1)}$ if $n>0$

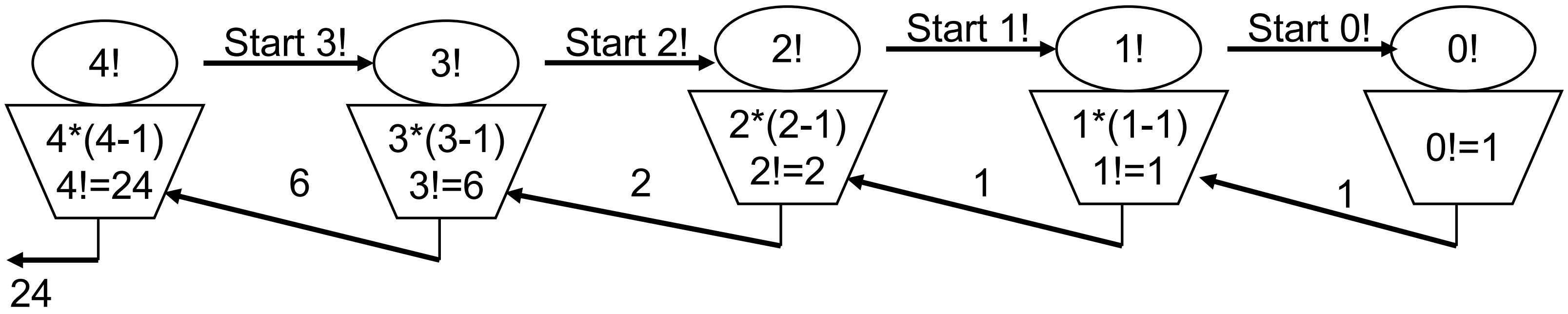
Design a Recursive Function: Factorial

- Factorial(n): n!=n*(n-1)*(n-2)*…*2*1
- 0!=1: special definition
- n!=1 if n=0 :stopping condition
- n!=n*(n-1)! if n>0 :recursive step
- factorial(n) : n-machine that computes the result using n*(n-1)!, so it needs result from (n-1) machine and it will output the final result



Design a Recursive Function: Factorial

- A network of machines that pass information back and forth
- Each needs the result of the previous machine, gives the result to the following machine
- 0 machine can work independently and produce result
- Machine n starts machine n-1
- Machine 1 starts machine 0
- Result passed back to machine 1
- Machine n produces final result



Design a Recursive Function: Factorial

- Stopping condition 0!

```
int Factorial (int n)
{
if (n==0)//stopping condition
  return 1;
else//recursive step
  return n*Factorial(n-1);
}
```

EXAMPLE: BINARY SEARCH

```
int BinarySearch (int list[ ], int low, int high, int key)
    {
      int mid;
      int midvalue;
      while (low<=high)
      {
            mid=(low+high)/2;
            midvalue=list[mid];
            if (key==midvalue)
                  return mid;
            else if (key<midvalue)
                  high=mid-1;
            else
                  low=mid+1;
      }
      return -1;
    }
```

Recursive Binary search

```
template <class T>
int BinSearch(T A[], int low, int high, T key)
{
    int mid;
    T midvalue;

    // key not found is a stopping condition
    if (low > high)
        return(-1);

    // compare against list midpoint and subdivide
    // if a match does not occur. apply binary
    // search to the appropriate sublist
```

```
else
    {
        mid = (low+high)/2;
        midvalue = A[mid];
        // stopping condition if key matched
        if (key == midvalue)
            return(mid);    // key found at index mid

        // look left if key < midvalue;
            //otherwise, look right
        else if (key < midvalue)
            // recursive step
            return BinSearch(A,low,mid-1,key);
        else
            // recursive step
            return BinSearch(A,mid+1,high,key);
    }
}
```

TEMPLATE

- We may want to use the same class or function definition for different types of items.

- It would be nice if we could define the data type with the object or with the function call.
- E.g.:

```
int SeqSearch(int list[ ], int n, int key)
 {
      for (int i=0; i<n; i++)
            if list[i]==key)
                  return i;
      return -1; }
```

```
template <class T>
int SeqSearch(T list[ ], int n, T key)
// T is a type that will be specified when SeqSearch
//is called
{
      for (int i=0; i<n; i++)
            if list[i]==key)
                  return i;
      return -1; }
```

….

int A[10], Aindex, Mindex;
float M[10], fkey=4.5;

Aindex=SeqSearch(A,10,25); //search for int 25 in int array A

Mindex=SeqSearch(M,100,fkey); //search for float 4.5 in float array M