Recursion

1. Recursion refers to a function that calls itself. For example, consider function *factorial*.

```
Line    Code
1       unsigned int factorial(unsigned int n)
2       {   if (n<1)
3           return 1;
4       else
5           return n*factorial(n-1);
6       }
```

Function *factorial* calls itself on line 5.

2. Recursion is implemented using a stack. Recall that the last element put on a stack is the first element retrieved from the stack. A function returns to its caller. *Activation records*, or *frames*, are pushed on and popped off a stack as functions are called and return. An activation record is put on the call-stack when a function is called and removed when the function returns.

   Parameters, local variables, and the return address are put on the call-stack.

   Consider the following example.

```
Line    Code
1       #include <iostream>
2       using namespace std;
1       unsigned int factorial(unsigned int n)
2       {   if (n<1)
3           return 1;
4       else
5           return n*factorial(n-1);
6       }
7       int main()
8       {   cout << factorial(3) << endl;
9           return 0;
10      }
```

   Function *factorial* is called once from function main and three times recursively as shown in Figure 1. Note that parameter n is diminished in each successive call to function *factorial*. No more calls are made to function factorial after the value of parameter n is reduced to zero.

   Activation records are removed from the call stack in Figure 2. The return value computed in the activation record on top of the call stack is transmitted to the calling function. Thus, *n***factorial*(*n*-**1**), is computed for every value of *n*. Please note the return values as they are computed in successive diagrams in Figure 2.

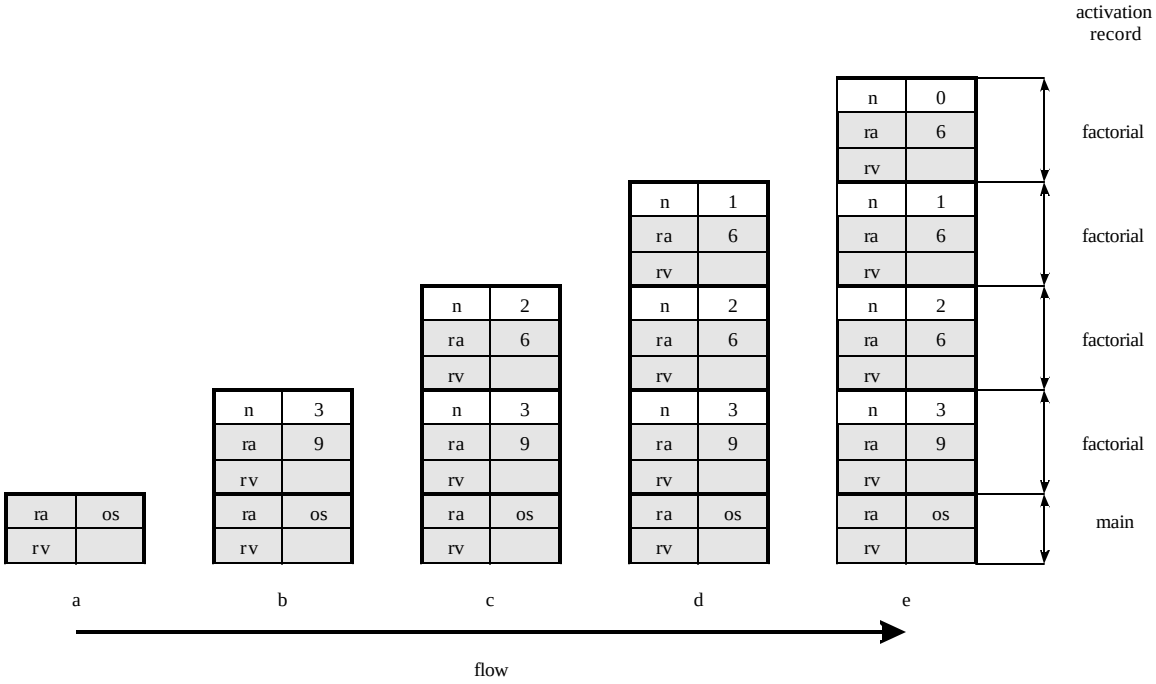

Figure 1. (a) Activation record for main. (b) Activation record for main and the first call to function `factorial`. (c) Activation record for main and two calls to function `factorial`. (d) Activation record for main and three calls to function `factorial`. (e) Activation record for main and four calls to function `factorial`.

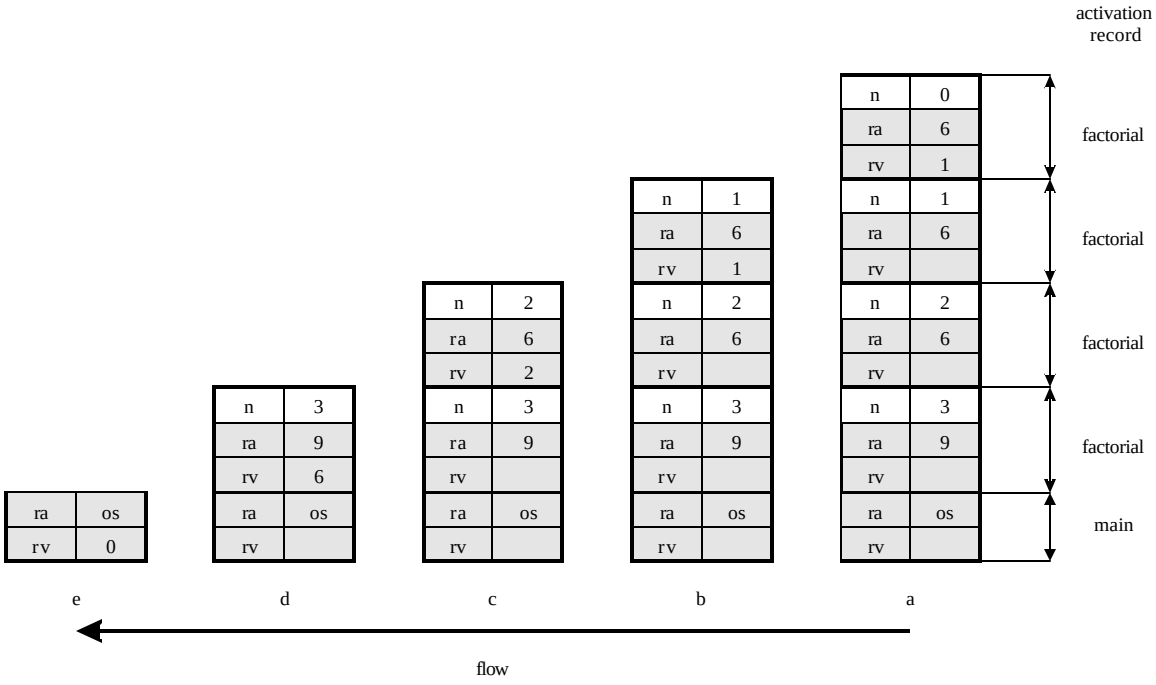

Figure 2. (a) Function factorial returns 0!. (b) Function factorial returns 1!. (c) Function factorial returns 2!, (d) Function factorial returns 3! (e) Function main returns to the operating system

3. Design recursive functions so they always terminate. Recursive functions that do not terminate, fail for lack of memory. Consider function *bad*.

```
Line    Code
1       int bad(unsigned int N)
2       { if (N==0)
3             return 0;
4         else
5             return bad(N/3 +1)+N-1;
6       }
```

For N>0, function bad never calls itself having a value of zero for its argument. Thus, function *bad* never terminates when the value of its parameter is greater than zero because function *bad* only terminates when its argument is equal to zero.

Recurrence Relations

1. A recurrence relation relates the *n*th element of a sequence to its predecessors. Because recurrence relations are closely related to recursive algorithms, recurrence relations arise naturally in the analysis of recursive algorithms.
2. Consider the sequence

   5, 8, 11, 14, 17, …

   Write a recurrence relation and initial conditions that can be used to generate the sequence given above.

   2.1. Recurrence relation: $a_n = a_{n-1} + 3$

   2.2. Initial conditions: $a_1 = 5$

3. Write recursive function `f` that implements the recurrence relation described in section 2.

```
Line    Code
1       struct ZeroException {
2         ZeroException() { }
3       };
4       unsigned int f(unsigned int n)
5       { switch (n) {
6             case 0: throw ZeroException();
7            case 1: return 5;
8            default: return f(n-1)+3;
9         }
10      }
```

4. Write a recurrence relation for a Fibonacci sequence. The current value is the sum of the two previous values. Define the first two values of the sequence to be 0 and 1.

   4.1. Recurrence relation: $a_n = a_{n-1} + a_{n-2}$

   4.2. Initial conditions: $a_0 = 0, a_1 = 1$

5. Write recursive function fib that implements the recurrence relations described in section 4.

```
Line    Code
1       unsigned int fib(unsigned int n)
2       { switch (n) {
3             case 0: return 0;
4             case 1: return 1;
5             default: return fib(n-1)+ fib(n-2);
6         }
7       }
```

Induction

1. Induction is often used to find an expression for a sequence. Recurrent relations produce many such sequences. For example, consider the sum of the sequence of odd numbers,

$$1+3+5+\cdots+(2n-1)=n^2$$

1.1. Hypothesis: $\sum_{i=1}^{n}(2i-1)=n^2$

1.2. Basis Step. Prove the hypothesis for n=1.

1.2.1. Left side of the equality $\sum_{i=1}^{1}(2i-1)=(2(1)-1)=1$

1.2.2. Right side of the equality $n=1, 1^2=1$

1.2.3. Left and right side match, completing the proof of the basis step.

1.3. Induction Step. Assume the hypothesis is true for $1\le i\le k$. Prove the hypothesis is true for $i=k+1$.

1.3.1. Prove $\sum_{i=1}^{k+1}(2i-1)=(k+1)^2$

$$\sum_{i=1}^{k+1}(2i+1)=\sum_{i=1}^{k}(2i+1)+(2(k+1)-1)$$

$\sum_{i=1}^{k}(2i-1)=k^2$ by applying the induction hypothesis

$\sum_{i=1}^{k+1}(2i+1)=k^2+(2(k+1)-1)$ by substitution

$$\sum_{i=1}^{k+1}(2i+1)=k^2+2k+2-1=k^2+2k+1=(k+1)^2$$