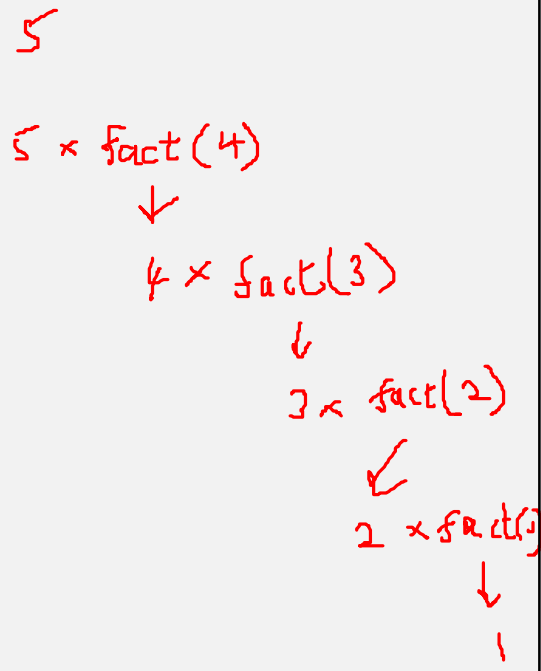
RECURSION IN C

- Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem.
- Any function which calls itself is called recursive function, and such function calls are called recursive calls.
- Recursion involves several numbers of recursive calls.
- However, it is important to impose a termination condition of recursion.
- Recursion code is shorter than iterative code however it is difficult to understand.

- Recursion cannot be applied to all the problem, but it is more useful for the tasks that can be defined in terms of similar subtasks.
- For Example, recursion may be applied to sorting, searching, and traversal problems.
- Generally, iterative solutions are more efficient than recursion since function call is always overhead.
- Any problem that can be solved recursively, can also be solved iteratively.
- However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

```c
#include <stdio.h>

int fact (int);

int main()

{

    int n,f;

    printf("Enter the number whose factorial you want to calculate?");

    scanf("%d",&n);

    f = fact(n);

    printf("factorial = %d",f);

}
```

```c
int fact(int n)
{
    if (n==0)
    {
        return 1;
    }
    else if ( n == 1)
    {
        return 1;
    }
    else
    {
        return n*fact(n-1);
    }
}
```




```
Enter the number whose factorial you want to calculate?5
factorial = 120
```

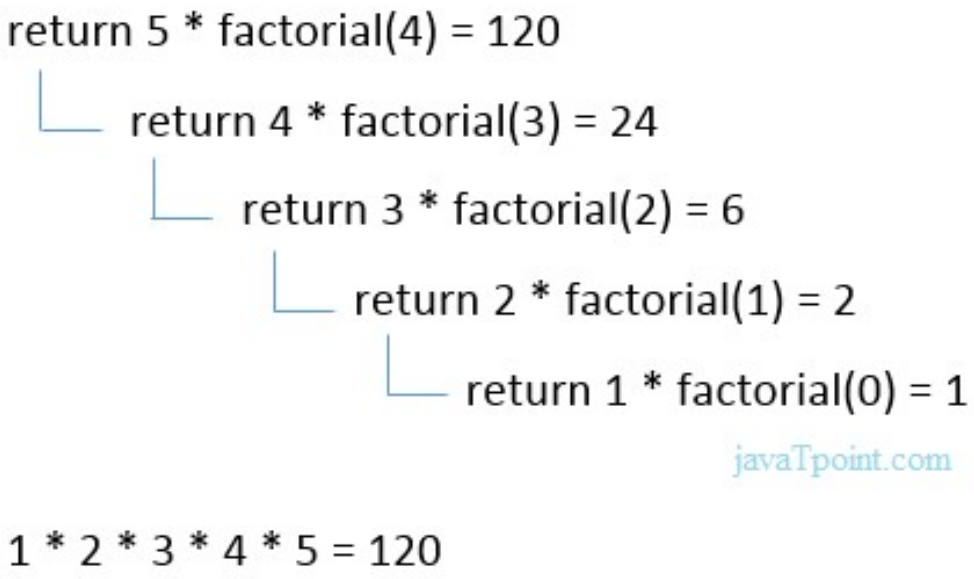


Fig: Recursion

RECURSIVE FUNCTION

- A recursive function performs the tasks by dividing it into the subtasks.
- There is a termination condition defined in the function which is satisfied by some specific subtask.
- After this, the recursion stops and the final result is returned from the function.
- The case at which the function doesn't recur is called the base case whereas the instances where the function keeps calling itself to perform a subtask, is called the recursive case.
- All the recursive functions can be written using this format.

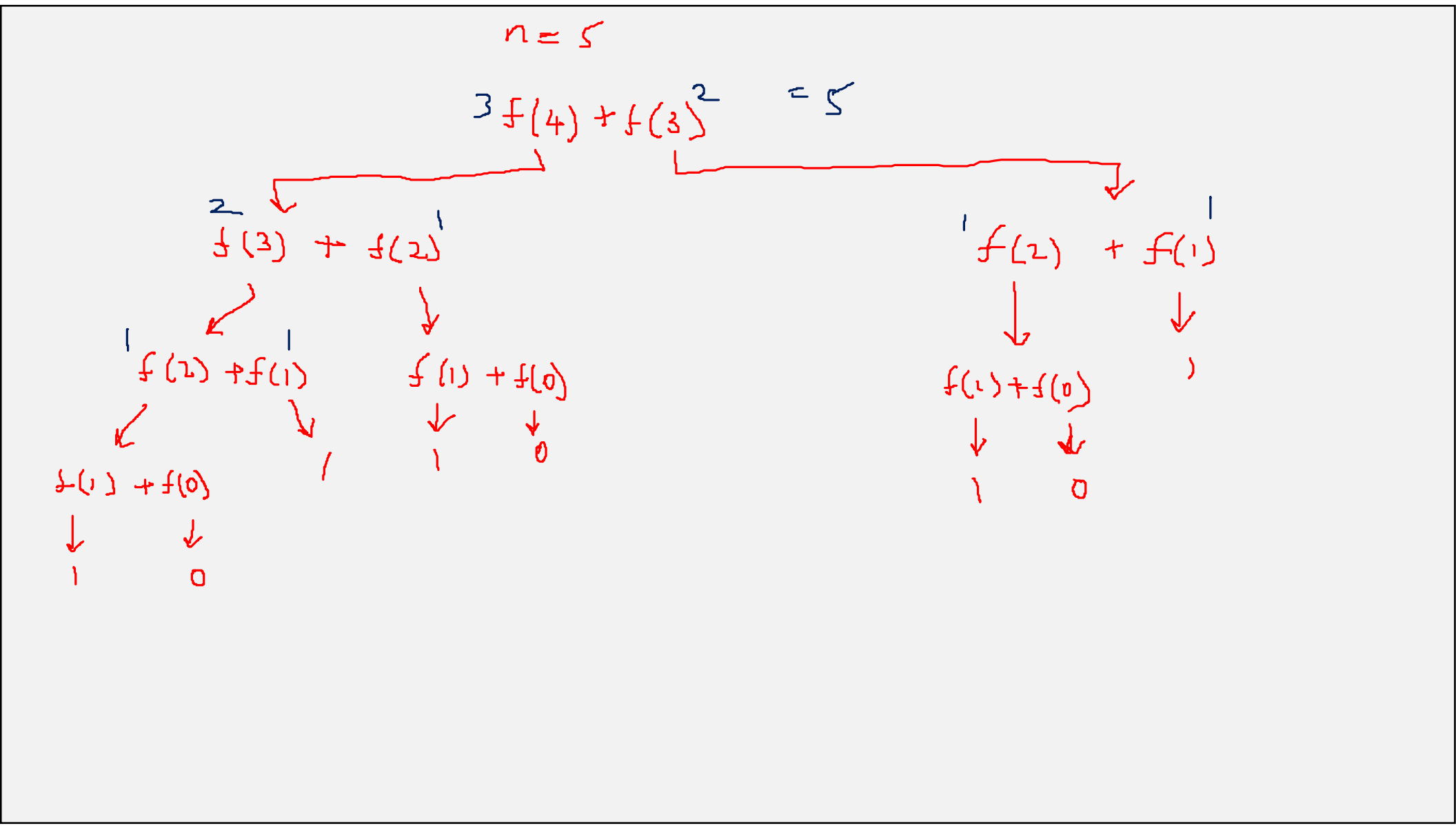
- Pseudocode for writing any recursive function is given below.

```
if (test_for_base)
{
    return some_value;
}
else if (test_for_another_base)
{
    return some_another_value;
}
else
{
    // Statements;
    recursive call;
}
```

```c
#include<stdio.h>
int fibonacci(int);
void main ()
{
    int n,f;
    printf("Enter the value of n?");
    scanf("%d",&n);
    f = fibonacci(n);
    printf("%d",f);
}
```

```c
int fibonacci (int n)
{
    if (n==0)
    {
    return 0;
    }
    else if (n == 1)
    {
        return 1;
    }
    else
    {
        return fibonacci(n-1)+fibonacci(n-2);
    }
}
```

```
Enter the value of n?12
144
```


n = 5
3f(4) + f(3)^2 = 5
2
f(3) + f(2)
1
1
f(2) + f(1)
1
f(2) + f(1)
f(1) + f(0)
f(1) + f(0)
1
1
1
0
1
0

MEMORY ALLOCATION OF RECURSIVE METHOD

- Each recursive call creates a new copy of that method in the memory.
- Once some data is returned by the method, the copy is removed from the memory.
- Since all the variables and other stuff declared inside function get stored in the stack, therefore a separate stack is maintained at each recursive call.
- Once the value is returned from the corresponding function, the stack gets destroyed.
- Recursion involves so much complexity in resolving and tracking the values at each recursive call.
- Therefore we need to maintain the stack and track the values of the variables defined in the stack.

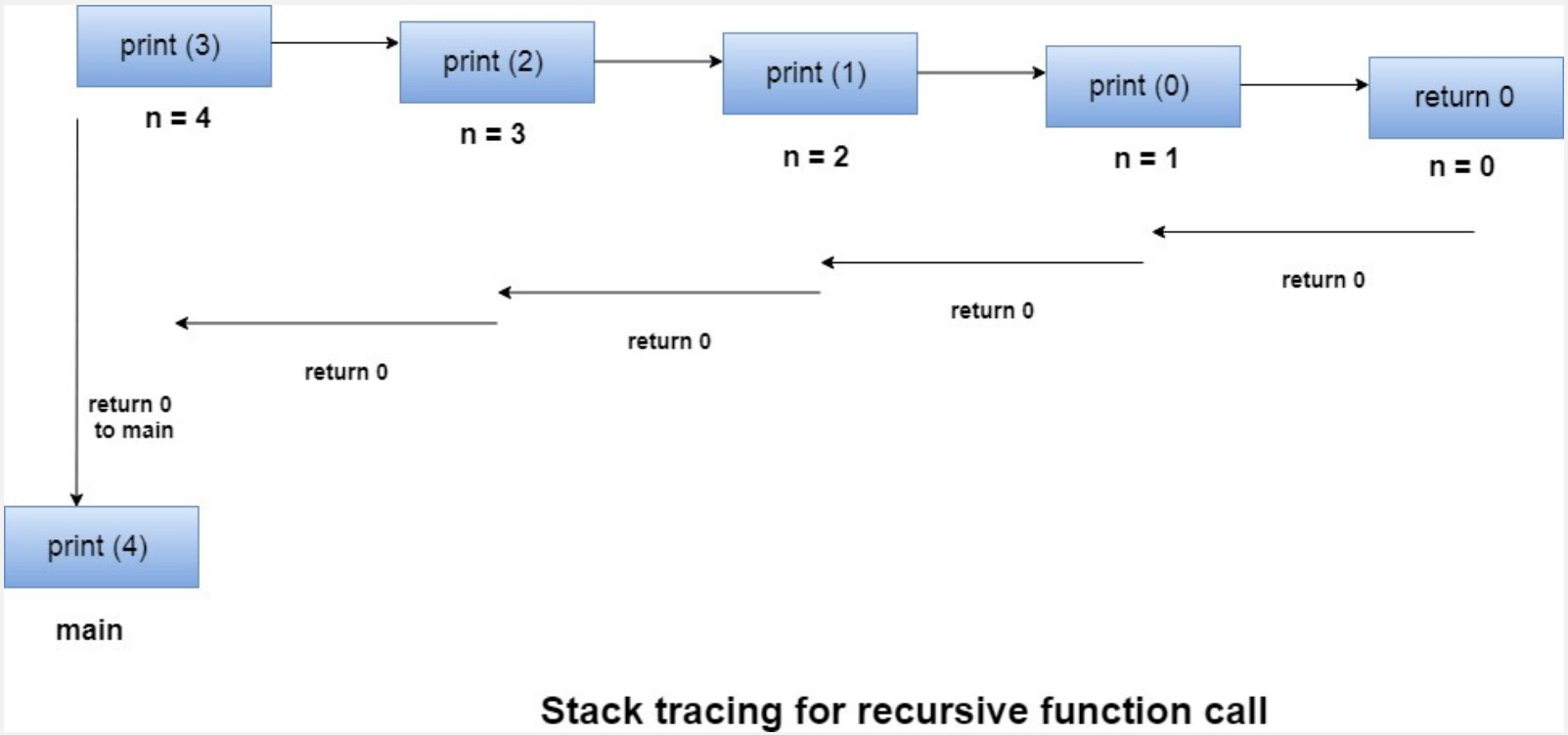
```
int display (int n)
{
    if(n == 0)
        return 0; // terminating condition
    else
    {
        printf("%d",n);
        return display(n-1); // recursive call
    }
}
```

Let us examine this recursive function for n = 4.

First, all the stacks are maintained which prints the corresponding value of n until n becomes 0,

Once the termination condition is reached, the stacks get destroyed one by one by returning 0 to its calling stack.

Consider the following image for more information regarding the stack trace for the recursive functions.



Stack tracing for recursive function call

STORAGE CLASSES IN C

- Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable.
- There are four types of storage classes in C
    - Automatic
    - External
    - Static
    - Register

Storage Classes	Storage Place	Default Value	Scope	Lifetime
auto	RAM	Garbage Value	Local	Within function
extern	RAM	Zero	Global	Till the end of the main program Maybe declared anywhere in the program
static	RAM	Zero	Local	Till the end of the main program, Retains value between multiple functions call
register	Register	Garbage Value	Local	Within the function

AUTOMATIC

- Automatic variables are allocated memory automatically at runtime.
- The visibility of the automatic variables is limited to the block in which they are defined.
- The scope of the automatic variables is limited to the block in which they are defined. The automatic variables are initialized to garbage by default.
- The memory assigned to automatic variables gets freed upon exiting from the block.
- The keyword used for defining automatic variables is auto.
- Every local variable is automatic in C by default.

```c
#include <stdio.h>
int main()
{
int a; //auto
char b;
float c;
printf("%d %c %f",a,b,c);
// printing initial default value of automatic variables a, b, and c.
return 0;
}
```

garbage garbage garbage

```c
#include <stdio.h>
int main()
{
int a = 10,i;
printf("%d ",++a);
{
int a = 20;
for (i=0;i<3;i++)
{
printf("%d ",a); // 20 will be printed 3 times since it is the local value of a
}
}
printf("%d ",a); // 11 will be printed since the scope of a = 20 is ended.
}
```

11 20 20 20 11

STATIC

- The variables defined as static specifier can hold their value between the multiple function calls.
- Static local variables are visible only to the function or the block in which they are defined.
- A same static variable can be declared many times but can be assigned at only one time.
- Default initial value of the static integral variable is 0 otherwise null.
- The visibility of the static global variable is limited to the file in which it has declared.
- The keyword used to define static variable is static.

```
#include<stdio.h>
static char c;
static int i;
static float f;
static char s[100];
void main ()
{
printf("%d %d %f %s",c,i,f,s); // the initial default value of c, i, and f will be printed.

}
```

0 0 0.000000 (null)

```
#include<stdio.h>
void sum()
{
static int a = 10;
static int b = 24;
printf("%d %d \n",a,b);
a++;
b++;
}
void main()
{
int i;
for(i = 0; i< 3; i++)
{
sum(); // The static variables holds their value between multiple function calls.
}
}
```

```
10 24
11 25
12 26
```

REGISTER

- The variables defined as the register is allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.
- We can not dereference the register variables, i.e., we can not use &operator for the register variable.
- The access time of the register variables is faster than the automatic variables.
- The initial default value of the register local variables is 0.

- The register keyword is used for the variable which should be stored in the CPU register. However, it is compiler?s choice whether or not; the variables can be stored in the register.
- We can store pointers into the register, i.e., a register can store the address of a variable.
- Static variables can not be stored into the register since we can not use more than one storage specifier for the same variable.

```c
#include <stdio.h>
int main()
{
register int a; // variable a is allocated memory in the CPU register. The initial default value of a is 0.
printf("%d",a);
}
```

0

```c
#include <stdio.h>
int main()
{
register int a = 0;
printf("%u",&a);
// This will give a compile time error since we can not access the address of a register variable.
}
```

```
main.c:5:5: error: address of register variable ?a? requested
printf("%u",&a);
^~~~~~
```

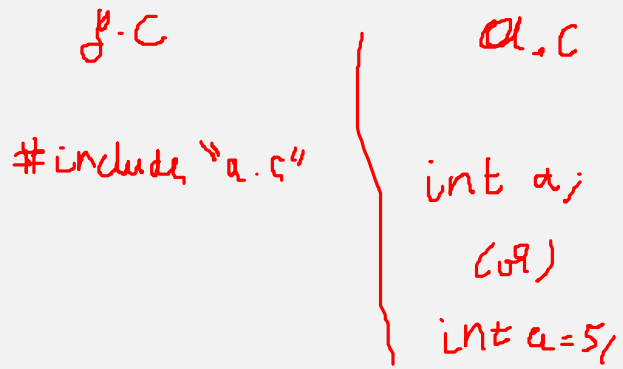
EXTERNAL

- The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.
- The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
- The default initial value of external integral type is 0 otherwise null.

- We can only initialize the extern variable globally, i.e., we can not initialize the external variable within any block or method.
- An external variable can be declared many times but can be initialized at only once.
- If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

```c
#include <stdio.h>
int main()
{
extern int a;
printf("%d",a);
}
```

```
main.c:(.text+0x6): undefined reference to `a'
collect2: error: ld returned 1 exit status
```




```
#include <stdio.h>
int a;
int main()
{
extern int a; // variable a is defined globally, the memory will not be allocated to a
printf("%d",a);
}
```

0

```c
#include <stdio.h>
int a;
int main()
{
extern int a = 0;
// this will show a compiler error since we can not use extern and initializer at same time
printf("%d",a);
}
```

```
compile time error
main.c: In function ?main?:
main.c:5:16: error: ?a? has both ?extern? and initializer
extern int a = 0;
```

```
#include <stdio.h>
int main()
{
extern int a; // Compiler will search here for a variable a defined and initialized somewhere in the pogram or not.
printf("%d",a);
}
int a = 20;
```

```
extern int a;
int a = 10;
#include <stdio.h>
int main()
{
printf("%d",a);
}
int a = 20; // compiler will show an error at this line
```

compile time error