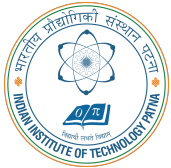
CS1101: Foundations of Programming

Recursion



Dept. of Computer Science & Engineering

Indian Institute of Technology Patna

Recursion

- A process by which function calls itself repeatedly
- A function call itself directly or indirectly
  - X calls X
  - X calls Y, Y calls Z, Z calls X
- The process is used for repetitive computations
- **Example:** `fact(n) = n * fact(n-1)`
- Many iterative problems can be written in this form

Base cases and Recursion

- As a function calls itself, we need some terminating condition. Otherwise, it will be infinite loop
- Usually, a recursion will have two components.
  - Base condition — stopping criteria
  - Recursive step — repetitive computation step
- Example:

```
fact(n)   = 1              if n = 0   // Stopping criteria
          = n * fact(n-1)  if n > 0   // Recursive step
```

- Usually, the stopping criteria is the scenario when we know the solution

Examples

- **Fibonacci sequence:**

Examples

- **Fibonacci sequence:**

```
Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-1)+Fib(n-2), if n>1
```

Examples

- **Fibonacci sequence:**

```
Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-1)+Fib(n-2), if n>1
```

- **Combination:**

Examples

- **Fibonacci sequence:**

```
Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-1)+Fib(n-2), if n>1
```

- **Combination:**

```
ncr(n,n) = 1
ncr(n,0) = 1
ncr(n,r) = ncr(n-1,r)+ncr(n-1,r-1), if n>r>0
```

Examples

- **Fibonacci sequence:**

```
Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-1)+Fib(n-2), if n>1
```

- **Combination:**

```
ncr(n,n) = 1
ncr(n,0) = 1
ncr(n,r) = ncr(n-1,r)+ncr(n-1,r-1), if n>r>0
```

- **GCD: assuming m $\geq$n**

Examples

- **Fibonacci sequence:**

```
Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-1)+Fib(n-2), if n>1
```

- **Combination:**

```
ncr(n,n) = 1
ncr(n,0) = 1
ncr(n,r) = ncr(n-1,r)+ncr(n-1,r-1), if n>r>0
```

- **GCD: assuming m $\geq$n**

```
gcd(m,0) = m
gcd(m,n) = gcd(n,m%n), if n>0
```

Example: Factorial

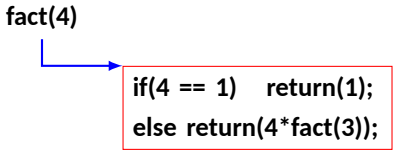
```
int fact(int n){
   if(n == 1){
     return 1;
   } else {
     return (n * fact(n-1));
   }
}
```

Recursion flow: Factorial

fact(4)

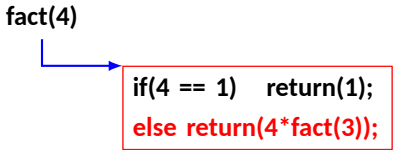
```
int fact(n){
   if(n==1) return(1);
   else return(n*fact(n-1));
}
```

Recursion flow: Factorial



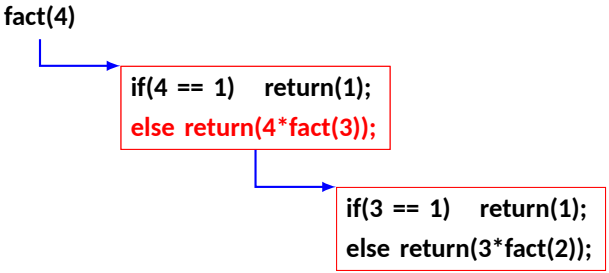

```
int fact(n){
   if(n==1) return(1);
   else return(n*fact(n-1));
}
```

Recursion flow: Factorial



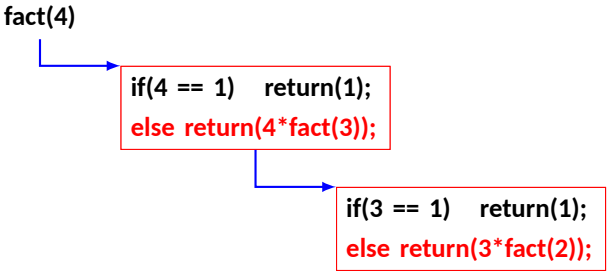

```
int fact(n){
   if(n==1) return(1);
   else return(n*fact(n-1));
}
```

Recursion flow: Factorial



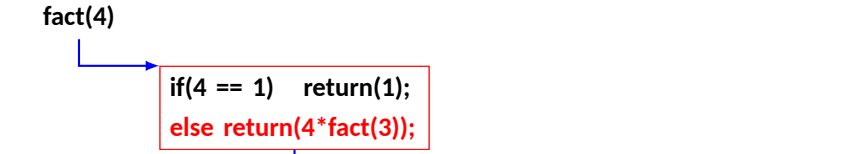

```
int fact(n){
   if(n==1) return(1);
   else return(n*fact(n-1));
}
```

Recursion flow: Factorial



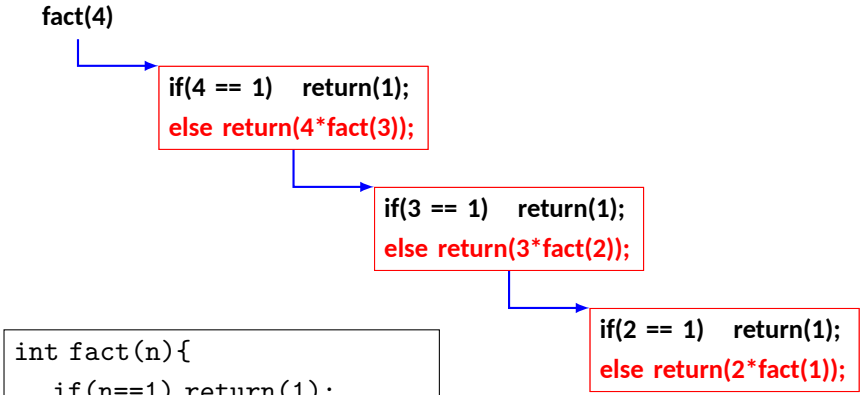

```
int fact(n){
   if(n==1) return(1);
   else return(n*fact(n-1));
}
```

Recursion flow: Factorial



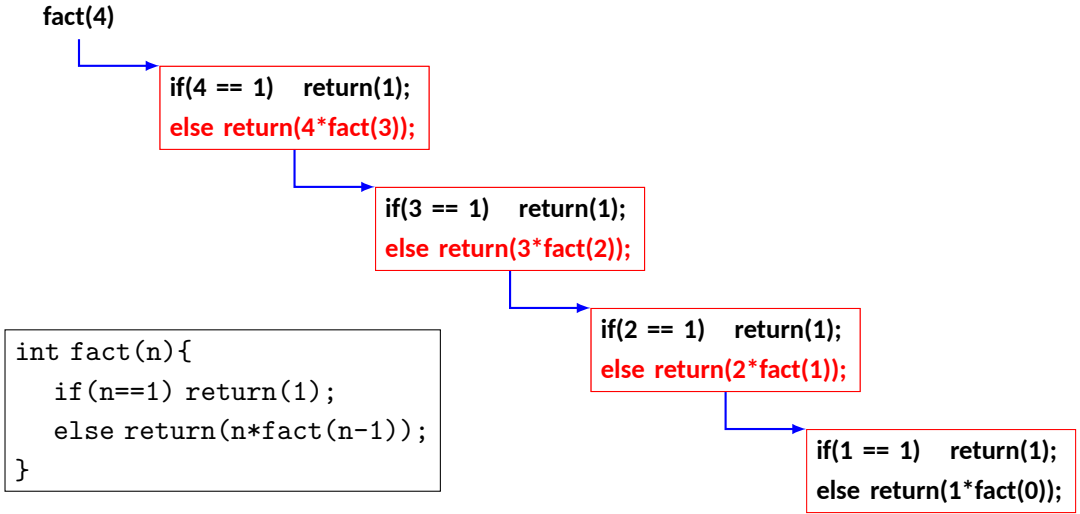

```
int fact(n){
   if(n==1) return(1);
   else return(n*fact(n-1));
}
```

Recursion flow: Factorial



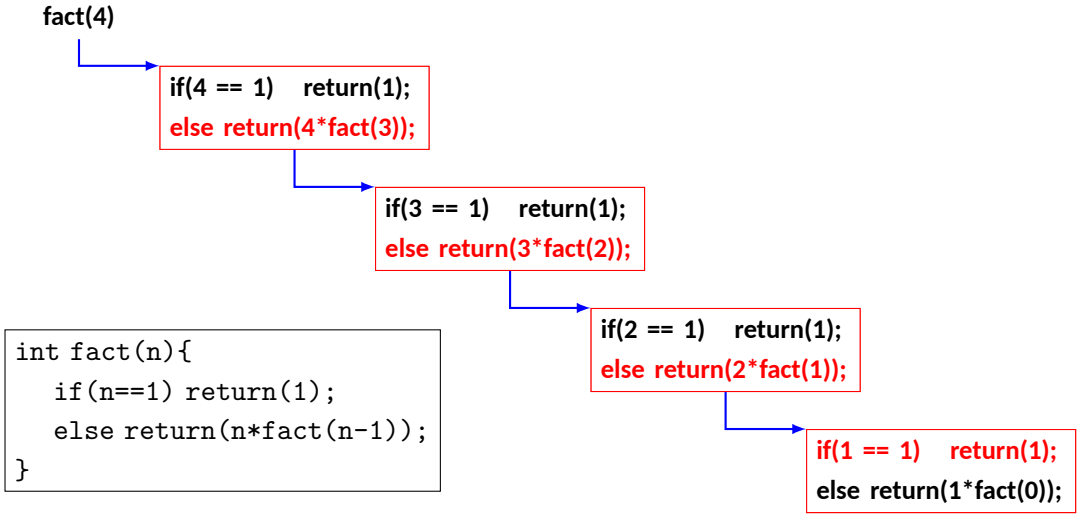

```
int fact(n){
   if(n==1) return(1);
   else return(n*fact(n-1));
}
```

Recursion flow: Factorial



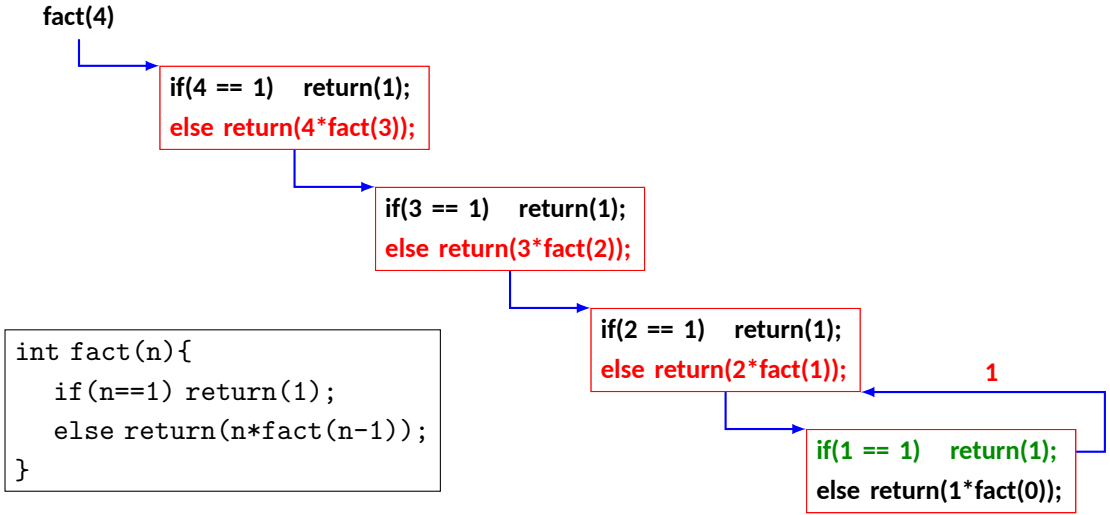

```
int fact(n){
   if(n==1) return(1);
   else return(n*fact(n-1));
}
```

Recursion flow: Factorial



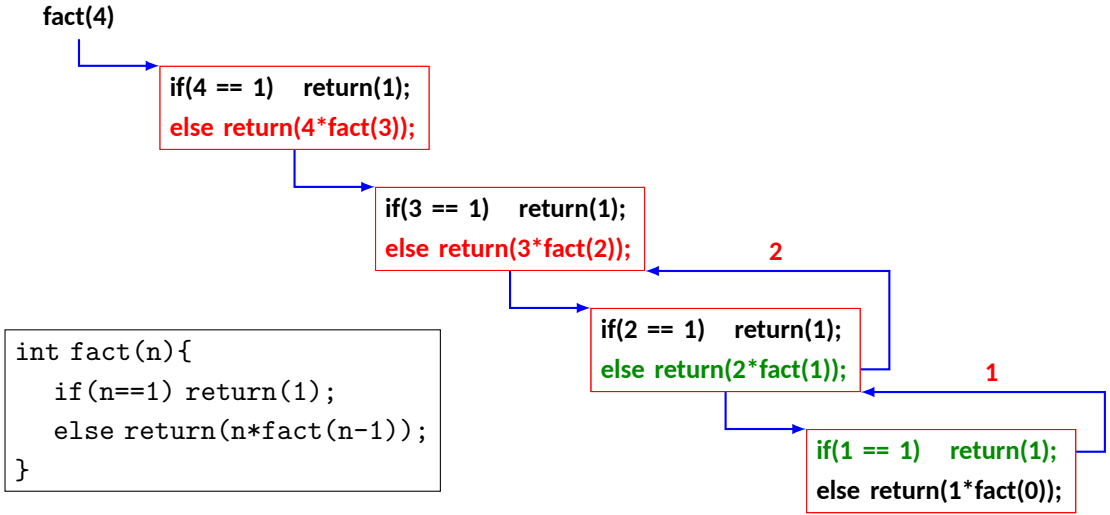

```
int fact(n){
   if(n==1) return(1);
   else return(n*fact(n-1));
}
```

Recursion flow: Factorial



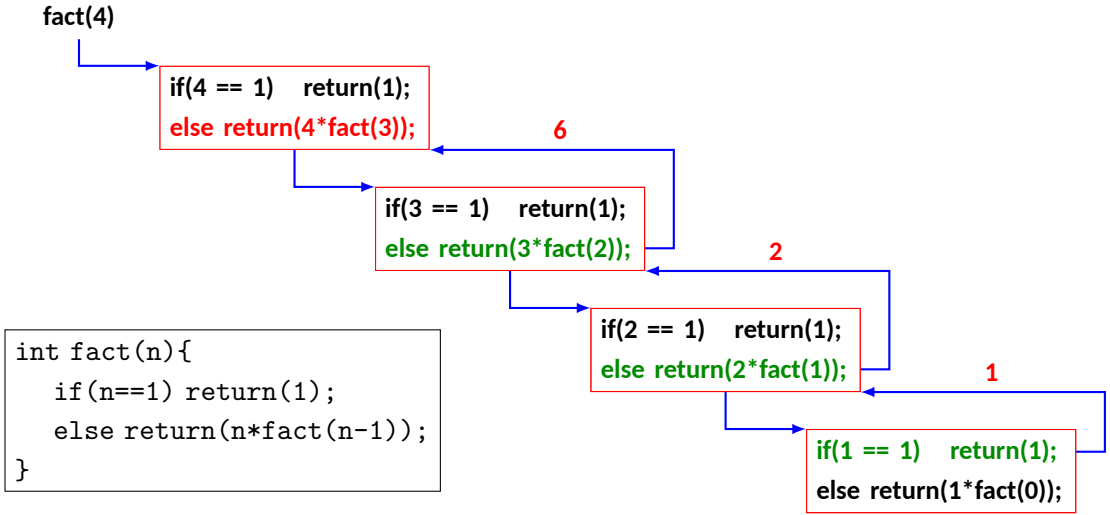

```
int fact(n){
   if(n==1) return(1);
   else return(n*fact(n-1));
}
```

Recursion flow: Factorial



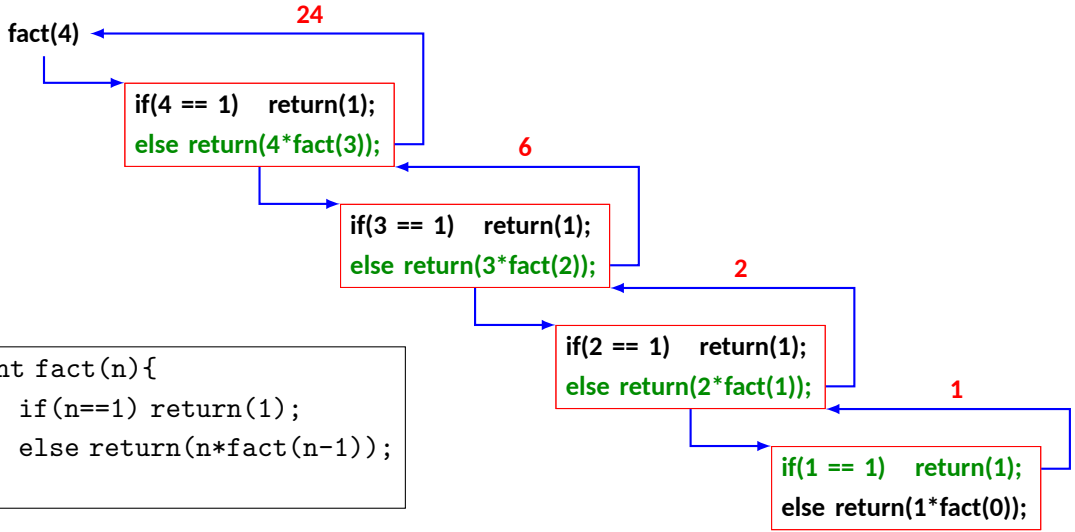

```
int fact(n){
   if(n==1) return(1);
   else return(n*fact(n-1));
}
```

Recursion flow: Factorial




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

Recursion flow: Factorial




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

Fibonacci sequence

- **Fibonacci sequence:** $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$
- **Sequence: 0,1,1,2,3,5,8,13,21,...**
- **Recursive code:**

```
int fib(n){
   if(n<2) return n;
   else return (fib(n-1)+fib(n-2));
}
```

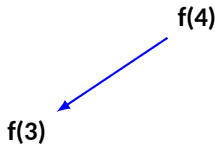
Recursion flow: Fibonacci

```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```

f(4)

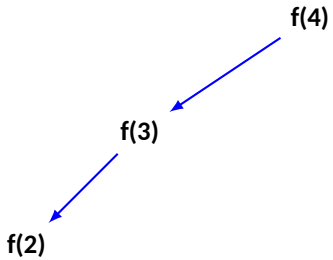
Recursion flow: Fibonacci

```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```



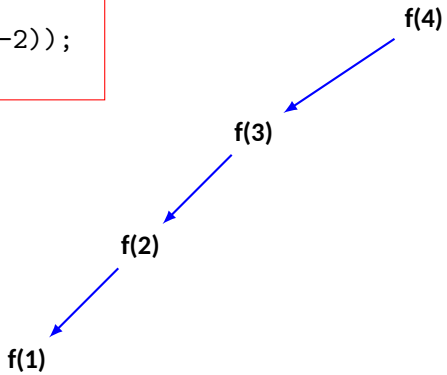
Recursion flow: Fibonacci

```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```



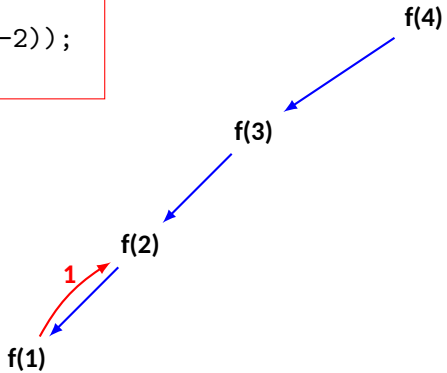
Recursion flow: Fibonacci

```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```



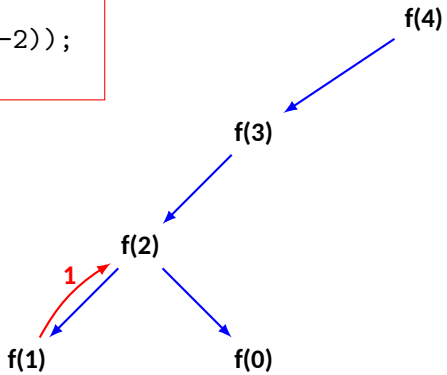
Recursion flow: Fibonacci

```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```



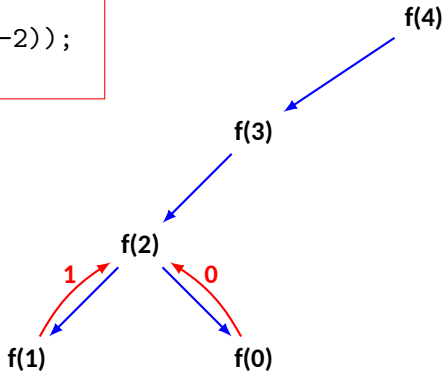
Recursion flow: Fibonacci

```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```



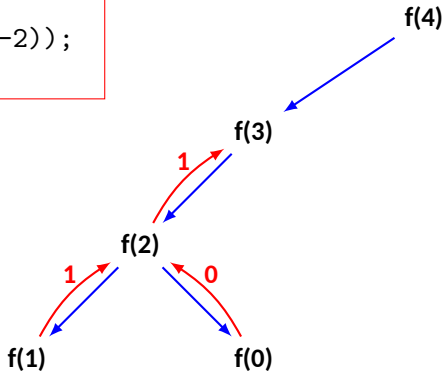
Recursion flow: Fibonacci

```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```



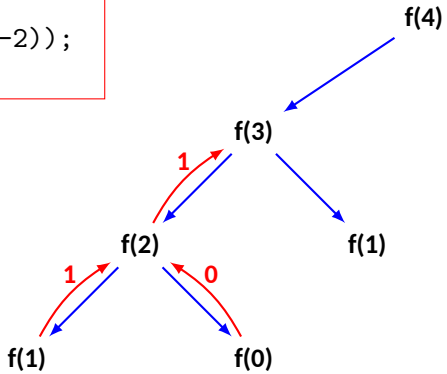
Recursion flow: Fibonacci

```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```



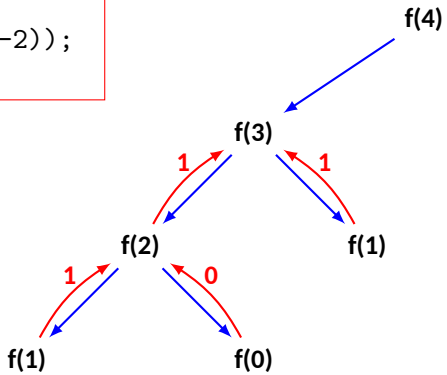
Recursion flow: Fibonacci

```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```



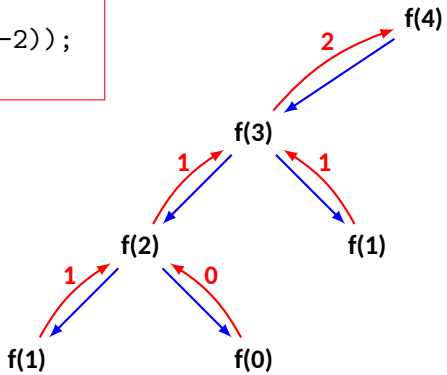
Recursion flow: Fibonacci

```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```



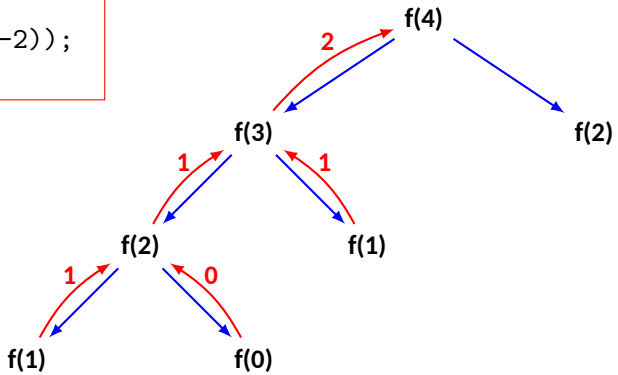
Recursion flow: Fibonacci

```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```



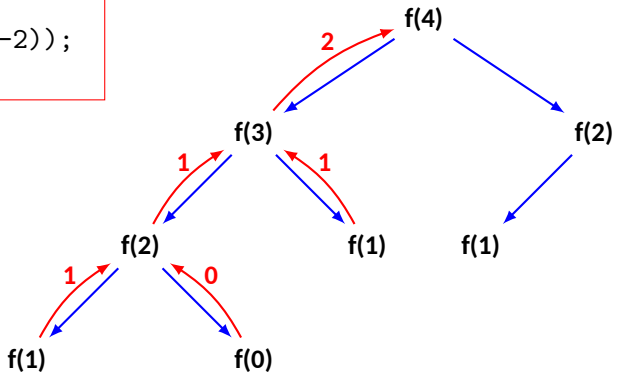
Recursion flow: Fibonacci

```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```



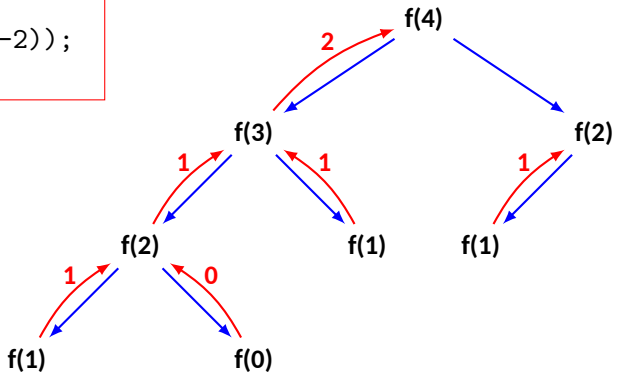
Recursion flow: Fibonacci

```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```



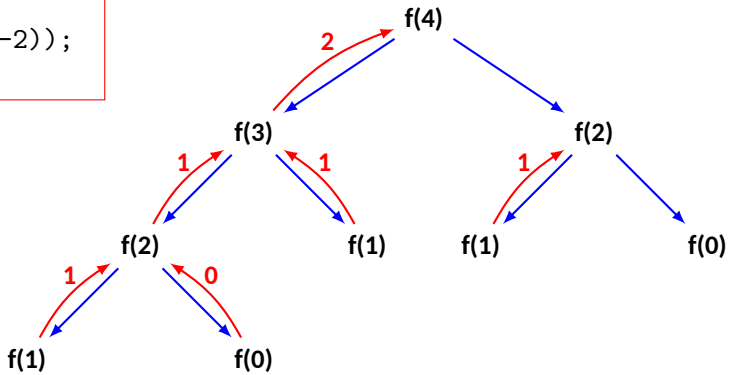
Recursion flow: Fibonacci

```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```



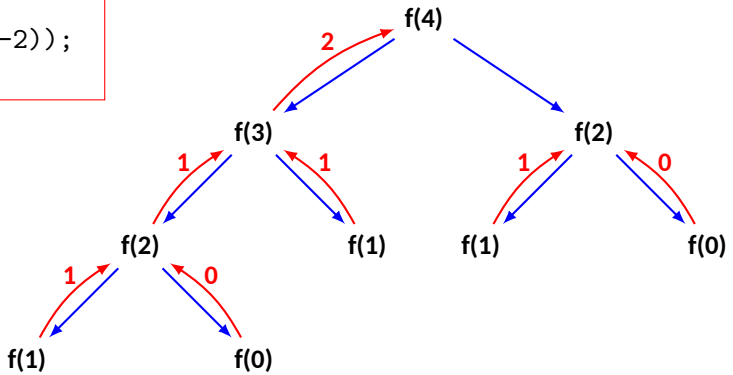
Recursion flow: Fibonacci

```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```



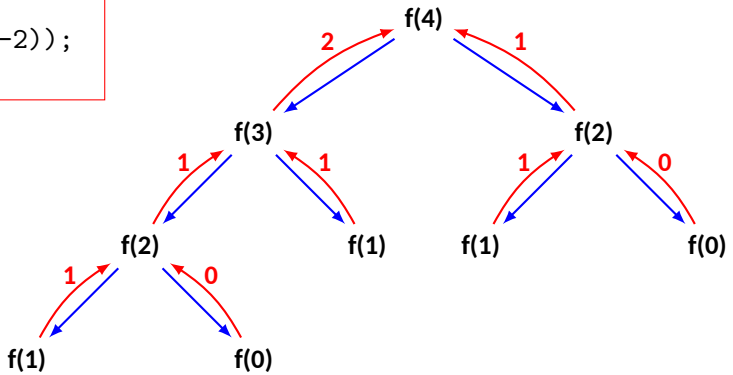
Recursion flow: Fibonacci

```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```



Recursion flow: Fibonacci

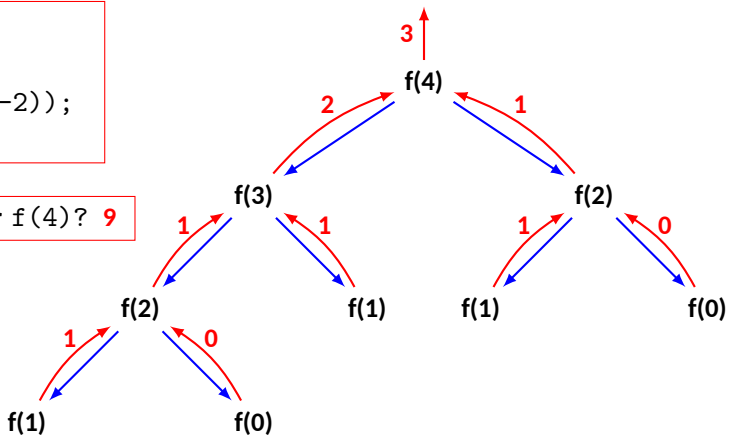
```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```



Recursion flow: Fibonacci

```
int f(int n){
   if(n<2) return(n);
   else return (f(n-1)+f(n-2));
}
```

How many `f()` calls are there for `f(4)`? 9



Inefficiency arises due to repeated computation of the same things

Notes

- Every recursion program can also be written without recursion
  - Easy to replace tail recursion by a loop
  - In general, removal of recursion may be a difficult task
- Recursion can be helpful in many solutions
  - Better readability, Ease of programming
  - Sometime, recursion gives best-possible or best known algorithms to solve problems
- Recursion can also be a killer
  - You solve the same subproblem multiple times
  - Every recursive call incurs a small overhead
- Use recursion carefully

Things to remember

- Think how the current problem can be solved if you can solve exactly the same problem on one or more smaller instances
- Do not think how the smaller problem will be solved, just call the function recursively assume that the recursive calls do the job correctly
- Do not forget to include base cases to solve the smallest instance
- This is similar to mathematical induction
- To write recursive function make sure of the following
  - First write the base condition (terminating step)
  - Write the rest of the part of the function

Example: Print the digits of an integer in reverse

- If the input is 2860 then you need to print 0682
- Use recursion to do this

Example: Print the digits of an integer in reverse

- If the input is 2860 then you need to print 0682
- Use recursion to do this

```
void printReverse(int j){
   if(j < 10){
     printf("%d",j);
   } else {
     printf("%d",j%10);
     printReverse(j/10);
   }
}
```

Print your name in reverse

```
#include <stdio.h>
void printReverse(){
   char c;
   scanf("%c",&c);
   if(c == '\n')
     return;
   printReverse();
   printf("%c",c);
}

int main(){
   printf("Enter your name:");
   printReverse();
   printf("\n");
   return 0;
}
```

Print your name in reverse

```
#include <stdio.h>
void printReverse(){
   char c;
   scanf("%c",&c);
   if(c == '\n')
     return;
   printReverse();
   printf("%c",c);
}

int main(){
   printf("Enter your name:");
   printReverse();
   printf("\n");
   return 0;
}
```

Output:

```
Enter your name:Doris Ursula
alusrU siroD
```

Print your name in reverse

```
#include <stdio.h>
void printReverse(){
   char c;
   scanf("%c",&c);
   if(c == '\n')
     return;
   printReverse();
   printf("%c",c);
}

int main(){
   printf("Enter your name:");
   printReverse();
   printf("\n");
   return 0;
}
```

Output:

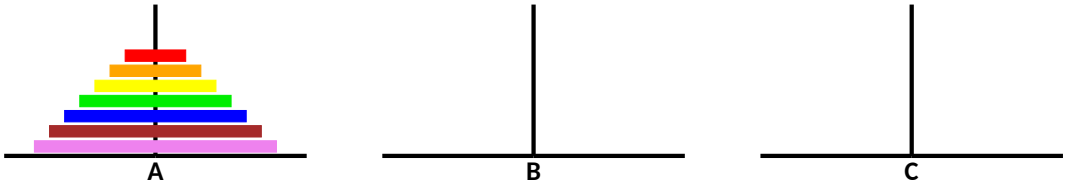
```
Enter your name:Doris Ursula
alusrU siroD
```

Modify the code to have following output:(Homework)

```
Enter your name:Doris Ursula
Name in reverse:alusrU siroD
```

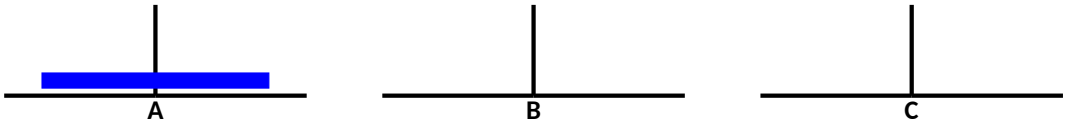
Tower of Hanoi

- **Problem:**
  - There are three pegs and 8 circular disks.
  - Each disk has a different size and a smaller disk can rest on a larger disk
  - At a time only one disk can be moved from one peg to the other
  - What is the minimum number of moves required?
- **How many moves will be required when the number of disk is 64? How much time will it take to execute?**
  - Known as Tower of Brahma



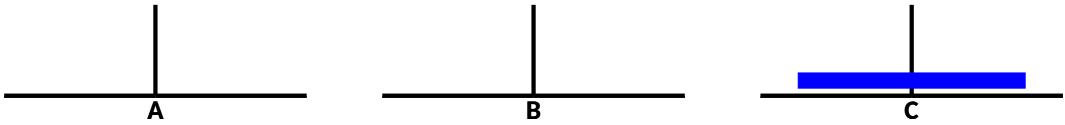
Tower of Hanoi - 1

- Moves when there is 1 disk



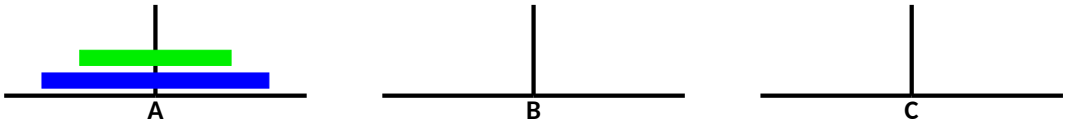
Tower of Hanoi - 1

- Moves when there is 1 disk



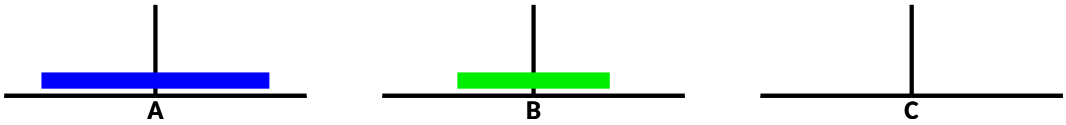
Tower of Hanoi - 2

- Moves when there are 2 disks



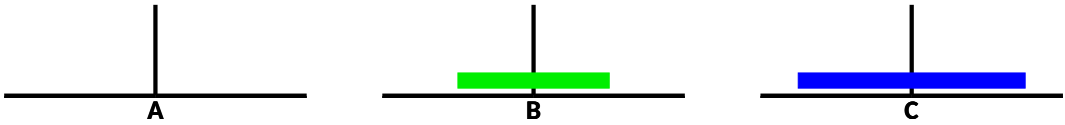
Tower of Hanoi - 2

- Moves when there are 2 disks



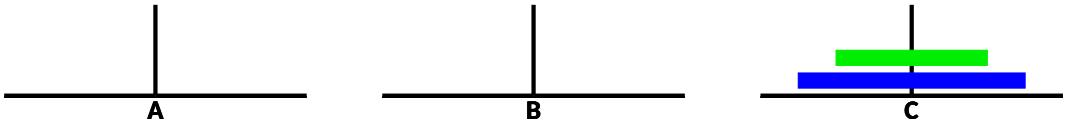
Tower of Hanoi - 2

- Moves when there are 2 disks



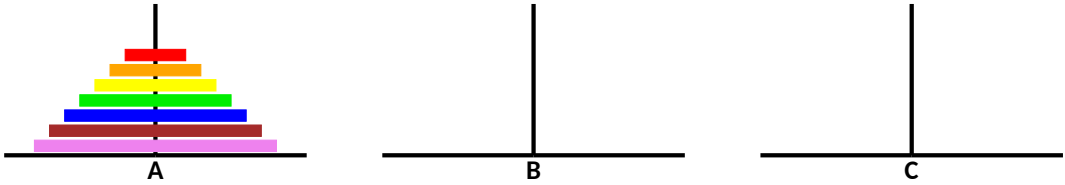
Tower of Hanoi - 2

- Moves when there are 2 disks



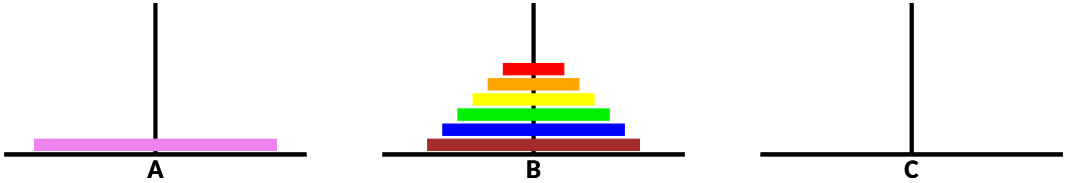
Tower of Hanoi

- **Problem:**
- **What is the strategy to move n disks? How many moves will be required when the number of disk is n?**



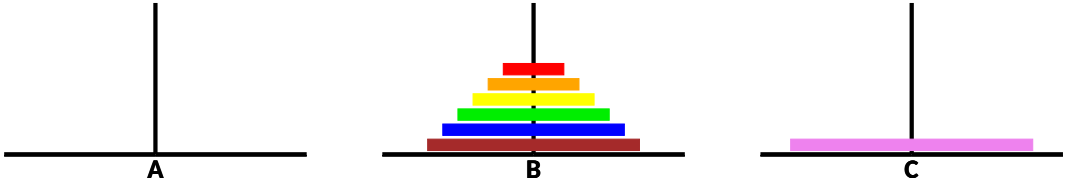
Tower of Hanoi

- **Problem:**
- **What is the strategy to move n disks? How many moves will be required when the number of disk is n?**



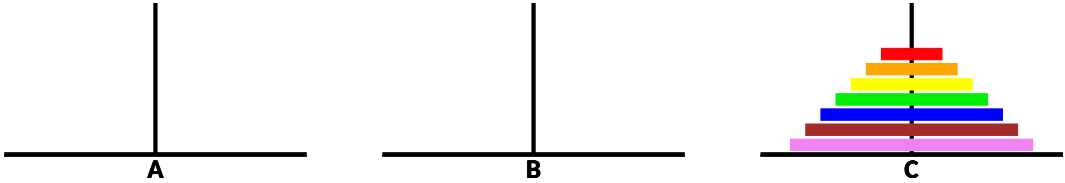
Tower of Hanoi

- **Problem:**
- **What is the strategy to move n disks? How many moves will be required when the number of disk is n?**



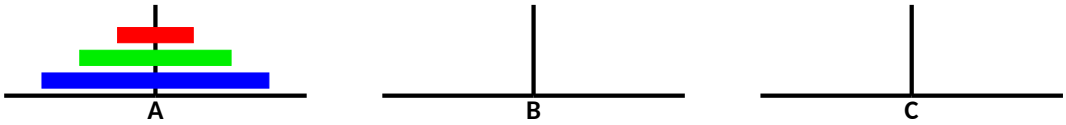
Tower of Hanoi

- **Problem:**
- **What is the strategy to move n disks? How many moves will be required when the number of disk is n?**



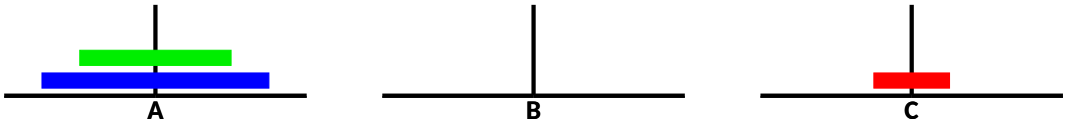
Tower of Hanoi - 3

- Moves when there are 3 disks



Tower of Hanoi - 3

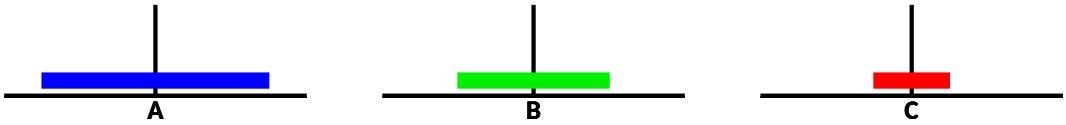
- Moves when there are 3 disks



Move A to C

Tower of Hanoi - 3

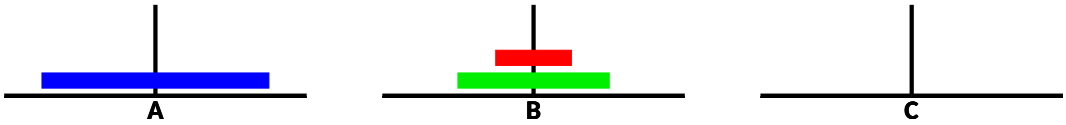
- Moves when there are 3 disks



Move A to C
Move A to B

Tower of Hanoi - 3

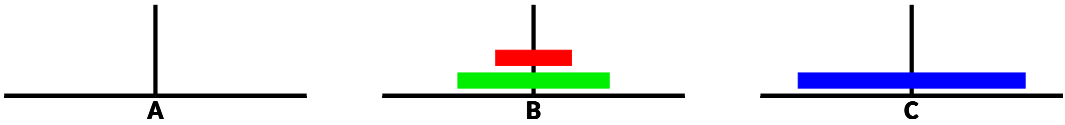
- Moves when there are 3 disks



Move A to C
Move A to B
Move C to B

Tower of Hanoi - 3

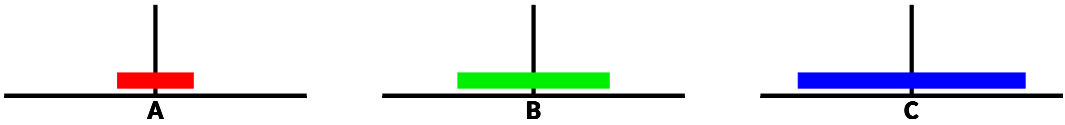
- Moves when there are 3 disks



Move A to C
Move A to B
Move C to B
Move A to C

Tower of Hanoi - 3

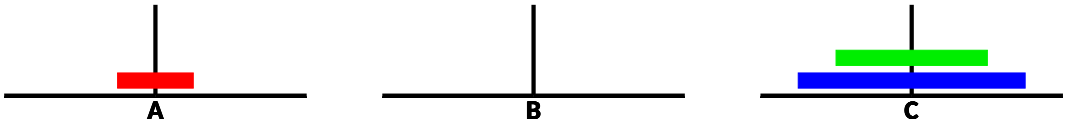
- Moves when there are 3 disks



Move A to C
Move A to B
Move C to B
Move A to C
Move B to A

Tower of Hanoi - 3

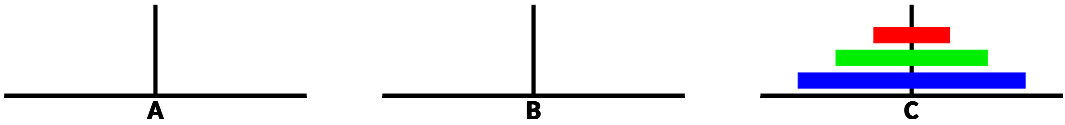
- Moves when there are 3 disks



Move A to C
Move A to B
Move C to B
Move A to C
Move B to A
Move B to C

Tower of Hanoi - 3

- Moves when there are 3 disks



Move A to C
Move A to B
Move C to B
Move A to C
Move B to A
Move B to C
Move A to C

Tower of Hanoi

```
#include <stdio.h>
void toh(int n, char F, char T, char A);

int main(){
  int n;
  scanf("%d",&n);
  toh(n,'A','C','B');
}

void toh(int n, char F, char T, char A){
  if(n>0){
    toh(n-1, F, A, T);
    printf("Move disk %d from %c to %c\n",n,F,T);
    toh(n-1, A, T, F);
  }
}
```

Common mistakes

- No base cases
- Terminating condition is never reached

Common mistakes

- No base cases
- Terminating condition is never reached

```
int fact(int n){
   return n*fact(n-1);
}
```

Common mistakes

- **No base cases**
- **Terminating condition is never reached**

```
int fact(int n){
   return n*fact(n-1);
}
```

```
int func1(int n){
   if(n==1) return 1;
   return func1(n--);
}
```

Common mistakes

- **No base cases**
- **Terminating condition is never reached**

```
int fact(int n){
   return n*fact(n-1);
}
```

```
int func1(int n){
   if(n==1) return 1;
   return func1(n--);
}
```

```
int func2(int n){
   if(n==0) return 1;
   return n*(n-1)*func2(n-2);
}
```

Common mistakes

- **No base cases**
- **Terminating condition is never reached**

```
int fact(int n){
   return n*fact(n-1);
}
```

```
int func1(int n){
   if(n==1) return 1;
   return func1(n--);
}
```

```
int func2(int n){
   if(n==0) return 1;
   return n*(n-1)*func2(n-2);
}
```

Base condition will not be reached for odd n

Recursion vs Iteration

- Common features
  - Both achieve repetition
  - Both have some terminating conditions, execution gradually approaches termination
  - Both can have infinite loop
- Issues
  - Recursion has overhead of function call
  - Many problems do not require recursive solutions
- Implementation / Readability is for recursion usually
- You need to take an educated decision

Data storage

```
int fact(int n){
   int val;
   if(n==1) return 1;
   val = n * fact(n-1);
   printf("n: %lu,%d  val: %lu,%d\n",&n,n,&val,val);
   return val;
}

int main(){
   int n=5;
   fact(n);
}
```

Data storage

```
int fact(int n){
   int val;
   if(n==1) return 1;
   val = n * fact(n-1);
   printf("n: %lu,%d  val: %lu,%d\n",&n,n,&val,val);
   return val;
}

int main(){
   int n=5;
   fact(n);
}
```

Output:

```
n: 140731046224636,2  val: 140731046224644,2
n: 140731046224684,3  val: 140731046224692,6
n: 140731046224732,4  val: 140731046224740,24
n: 140731046224780,5  val: 140731046224788,120
```

What will be the output?

```
void func(int n){
   int d;
   if(n==0) return;
   scanf("%d",&d);
   func(n-1);
   printf("%d\n",d);
}

int main(){
   int n=5;
   func(n);
}
```

What will be the output?

```
void func(int n){
   int d;
   if(n==0) return;
   scanf("%d",&d);
   func(n-1);
   printf("%d\n",d);
}

int main(){
   int n=5;
   func(n);
}
```

```
void func(int n){
   int d;
   if(n==0) return;
   func(n-1);
   scanf("%d",&d);
   printf("%d\n",d);
}

int main(){
   int n=5;
   func(n);
}
```

What will be the output?

```
void func(int n){
   int d;
   if(n==0) return;
   scanf("%d",&d);
   func(n-1);
   printf("%d\n",d);
}

int main(){
   int n=5;
   func(n);
}
```

```
void func(int n){
   int d;
   if(n==0) return;
   func(n-1);
   scanf("%d",&d);
   printf("%d\n",d);
}

int main(){
   int n=5;
   func(n);
}
```

```
void func(int n){
   int d;
   if(n==0) return;
   scanf("%d",&d);
   printf("%d\n",d);
   func(n-1);
}

int main(){
   int n=5;
   func(n);
}
```

What will be the output?

```
int func(int n){
   int s;
   if(n==0) return 0;
   s = n + func(n-1);
   printf("%d\n",s);
   return s;
}

int main(){
   int n=5;
   func(n);
}
```

What will be the output?

```
int func(int n){
   int s;
   if(n==0) return 0;
   s = n + func(n-1);
   printf("%d\n",s);
   return s;
}

int main(){
   int n=5;
   func(n);
}
```

Output:

1
3
6
10
15

What will be the output?

```
int func(int n){
   int sum,d;
   if(n==0) return 0;
   scanf("%d",&d);
   sum = func(n-1);
   sum += d;
   printf("%d\n",sum);
   return sum;
}

int main(){
   int n=5;
   func(n);
}
```

What will be the output?

```
int func(int n){
   int sum,d;
   if(n==0) return 0;
   scanf("%d",&d);
   sum = func(n-1);
   sum += d;
   printf("%d\n",sum);
   return sum;
}

int main(){
   int n=5;
   func(n);
}
```

```
void func(int n,int sum){
   int d;
   if(n==0) return;
   scanf("%d",&d);
   sum += d;
   printf("%d\n",sum);
   func(n-1,sum);
}

int main(){
   int n=5;
   func(n,0);
}
```

Input:
1 2 3 4 5
Output: ?

What will be the output?

```
int func(int n, int m){
   if(n==1) return m;
   else return (m + func(n-1, m));
}

int main(){
   int n=13, m=5, out;
   out = func(n, m);
}
```

What will be the output?

```
int func(int n, int m){
   if(n==1) return m;
   else return (m + func(n-1, m));
}

int main(){
   int n=13, m=5, out;
   out = func(n, m);
}
```

Multiplication using repeated sum

Largest element in the array

```
int max(int x[], int low, int size){
   int large;
   if(low == size) return x[low-1];
   large = max(x, low+1, size);
   return large > x[low-1] ? large : x[low-1];
}
```

Largest element in the array

```
int max(int x[], int low, int size){
   int large;
   if(low == size) return x[low-1];
   large = max(x, low+1, size);
   return large > x[low-1] ? large : x[low-1];
}
```

Intuition: Find the largest element (m) from 2 to n, then compare m with the first element and determine largest for 1 to n

Largest element in the array

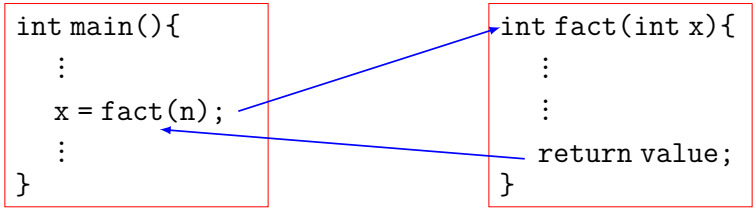
```
int max(int x[], int l, int h){
   int m, maxl, maxr;
   if(l == h) return x[l];
   maxl = max(x, l, (l+h)/2);
   maxr = max(x, (l+h)/2+1, h);
   return maxl > maxr ? maxl : maxr ;
}
```

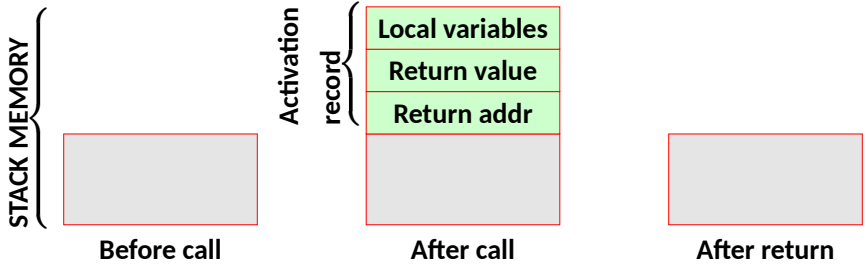
Largest element in the array

```
int max(int x[], int l, int h){
   int m, maxl, maxr;
   if(l == h) return x[l];
   maxl = max(x, l, (l+h)/2);
   maxr = max(x, (l+h)/2+1, h);
   return maxl > maxr ? maxl : maxr ;
}
```

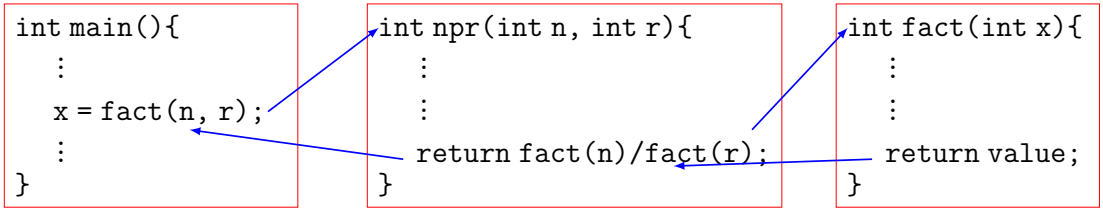
Intuition: Find the largest on the first half (m_1), then on the second half (m_2), and then return the maximum of m_1 and m_2

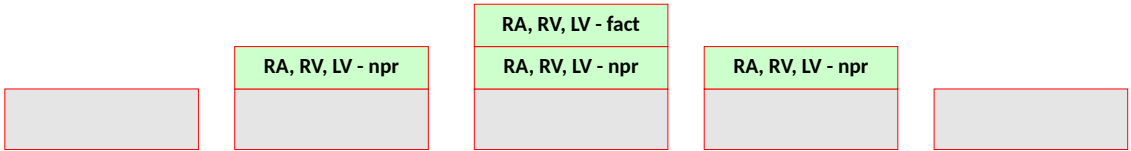
Memory: Function call





Memory: Function call





Practice problems

- Write a recursive function to find an element in an array
- Write a recursive function that prints the binary equivalent of a integer number
- Write a recursive function to find sum of the squares of a set of integers stored in an array
- Write a recursive function to copy one array to another
- Write a recursive function to determine x^n, x - float, n - int