CS101 Introduction to computing

Recursive Function

A. Sahu and P. Mitra
Dept of Comp. Sc. & Engg.
Indian Institute of Technology Guwahati

Outline

- Recursion
- Recursion vs iteration
- Base case
- Stack grow and shrink
- Number of recursive calls

Recursions
and
Recursive functions

Recursions: Recursive functions

- Functions that call themselves
- Can only solve a base case
- Divide a problem up into
  - What it can do
  - What it cannot do
    - What it cannot do resembles original problem
    - The function launches a new copy of itself (recursion step) to solve what it cannot do
- Eventually base case gets solved
  - Gets plugged in, works its way up and solves whole problem

Recursions: Recursive functions

- **Many Problem , we define the problem it self using recursive definition**
- Solving them using recursion
  - Easier to think and implement
- Example
  - Fibonacci, GCD, Binary Search, calculation of X^n, Reversing number
- Recursive Functions
  - Functions that call themselves (directly/indirectly)
  - Can only solve a base case

Recursion Example: factorials

- `5! = 5 * 4 * 3 * 2 * 1`
- Notice that
  - `5! = 5 * 4!`
  - `4! = 4 * 3! ...`
- Can compute factorials recursively
- Solve base case (`1! = 0! = 1`) then plug in
  - `2! = 2 * 1! = 2 * 1 = 2;`
  - `3! = 3 * 2! = 3 * 2 = 6;`

Recursion vs Iteration

- **Repetition**
  - Iteration: explicit loop
  - Recursion: repeated function calls
- **Termination**
  - Iteration: loop condition fails
  - Recursion: base case recognized
- **Both can have infinite loops**
- **Balance**
  - Choice between performance (iteration) and good software engineering (recursion)

Recursion Example: factorials

- From Definition: but no termination

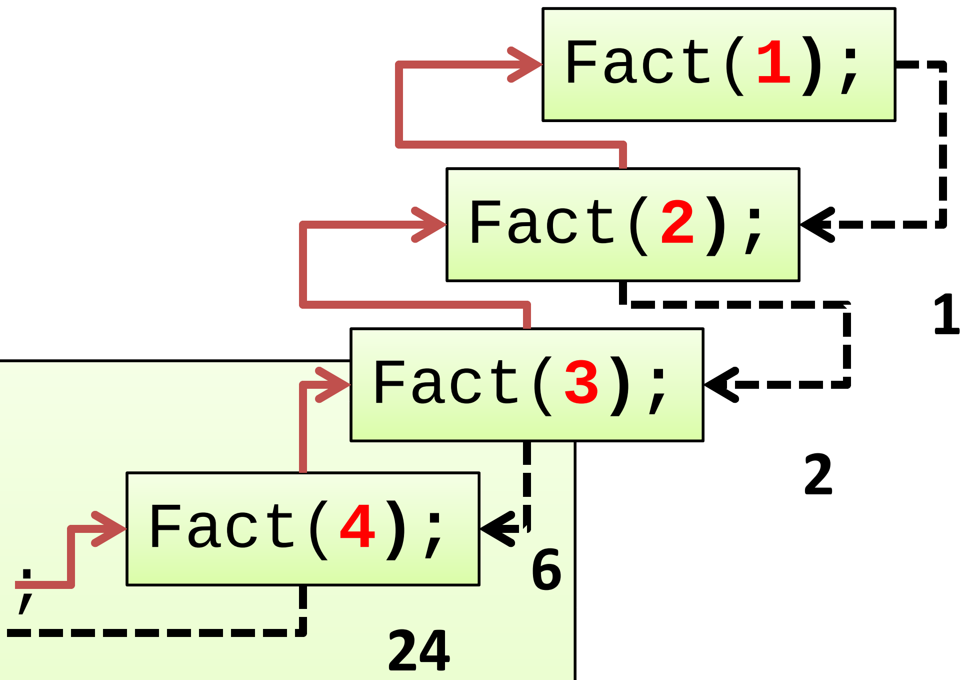
```
int   Fact(int N){

      return N * Fact (N-1);
}
```

- With proper base case (termination guaranteed)

```
int   Fact (int N){
      if(N<=1) return 1;
      return N * Fact (N-1);
}
```

Factorial: Recursive call

```c
int  Fact (int N){
   if(N<=1) return 1;
   return N*Fact(N-1);
}
int main(){
   int X;
   X=fact(5);
   printf("Fact of 5=%d \n",X);
   return 0;
}
```



120

Nested Function Call

- Nested Function call uses
  - Stack to store the return address, return result and any other information
- Recursion
  - Stack grows as deepen the nested function call
  - Recursive call: Stack grows when it call next recursive function
  - All the local variable need to be put into stack
  - Stack contains grows like this: Fib Example
    - Main, Fib(5), Fib(4), Fib(3), Fib(2), Fib(1)

Recursion vs Iteration

- **Any problem that can be solved recursively**
  - **Can also be solved iteratively (non-recursively)**
- A recursive approach is normally chosen in preference
  - To an iterative approach when the recursive approach more naturally mirrors the problem
  - And results in a program that is easier to understand and debug
- Another reason to choose a recursive solution
  - An iterative solution may not be apparent

Recursion vs Iteration

- **Avoid using recursion in performance situations**
  - Recursive calls take time
  - And consume additional memory
- **Common Error by programmer**
  - Accidentally a non-recursive function may call itself either directly, or indirectly through another function
- **Functionalizing programs in a neat, hierarchical manner promotes good practice**
  - More easier to program, test, debug, maintain, and evolve.
  - But it has a price, A heavily functionalized program: makes potentially large numbers of function calls

Solving Recursive Problems

- See recursive solutions as two sections:
  - Current
  - Rest

N! = N * (N-1)!

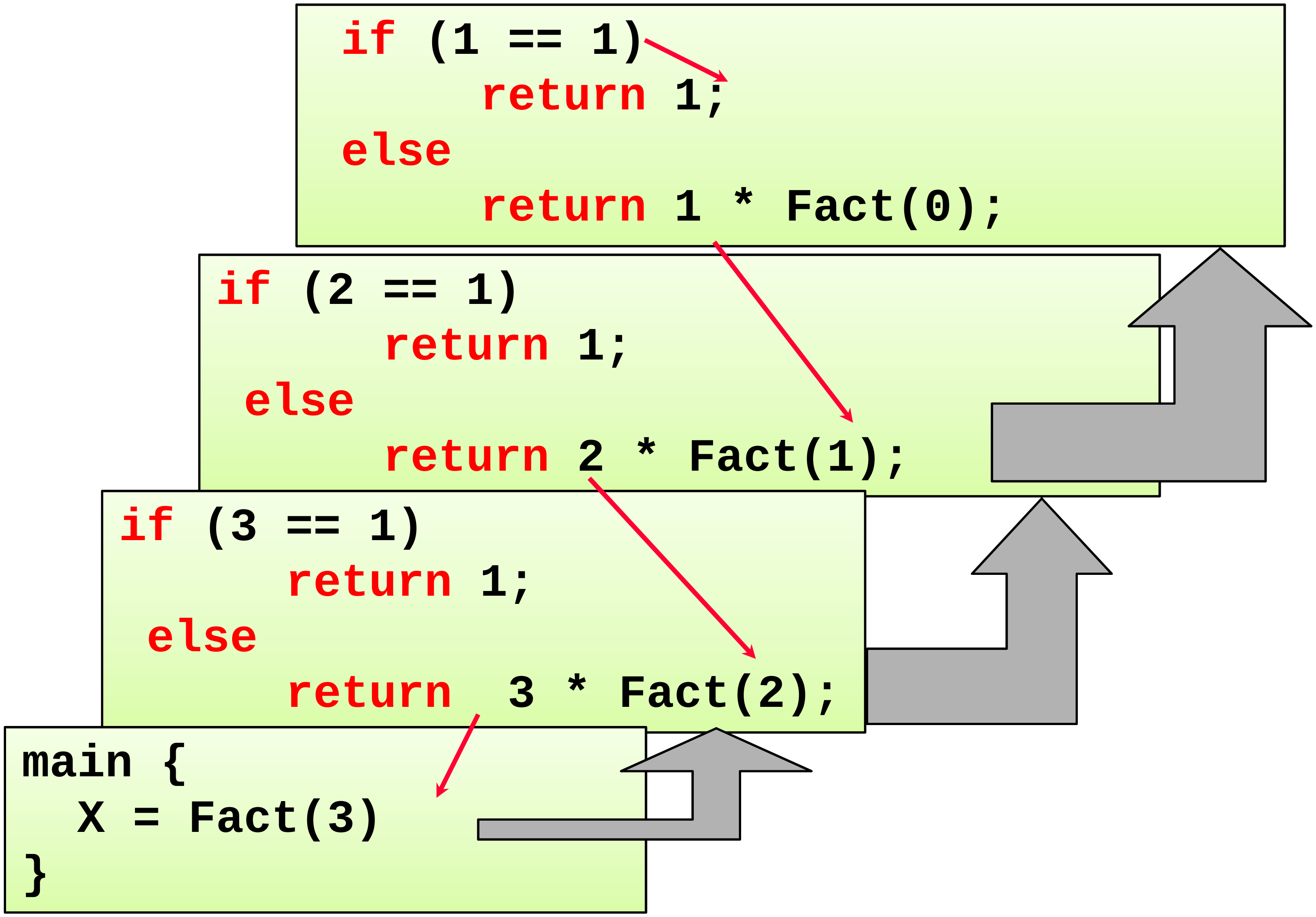
7! = 7 * 6!

7! = 7 * (6 * 5 * 4 * 3 * 2 * 1 * 1)

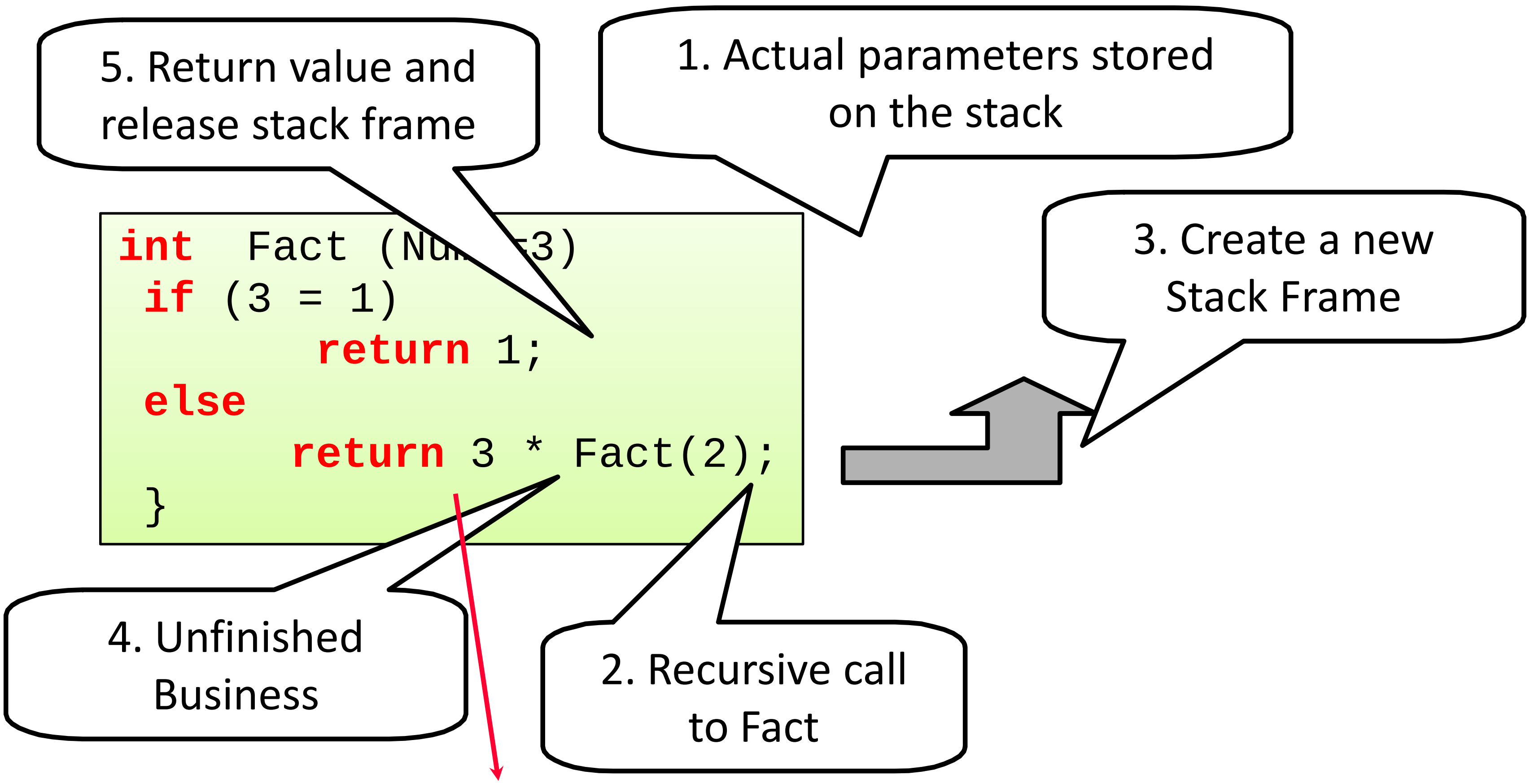
Factorial Function

```
int  Fact (int N){
   if(N<=1) return 1;
   return N*Fact(N-1);
}
```

Factorial Function



Tracing Details



Activation Stack for Factorial

Call the function: `X= Fact(5);`

Main Function: Unfinished: X = Fact (5);

Activation Stack for Factorial

Fact. 1st: N=5, Unfinished: 5*Fact(4)
Main Function: Unfinished: X = Fact (5);

Activation Stack for Factorial

Fact. 2nd: N=4,	Unfinished: 4*Fact(3)
Fact. 1st: N=5,	Unfinished: 5*Fact(4)
Main Function:	Unfinished: X = Fact (5);

Activation Stack for Factorial

Fact. 3rd: N=3,	Unfinished: 3*Fact(2)
Fact. 2nd: N=4,	Unfinished: 4*Fact(3)
Fact. 1st: N=5,	Unfinished: 5*Fact(4)
Main Function:	Unfinished: X = Fact (5);

Activation Stack for Factorial

Fact. 4th: N=2,	Unfinished: 2*Fact(1)
Fact. 3rd: N=3,	Unfinished: 3*Fact(2)
Fact. 2nd: N=4,	Unfinished: 4*Fact(3)
Fact. 1st: N=5,	Unfinished: 5*Fact(4)
Main Function:	Unfinished: X = Fact (5);

Activation Stack for Factorial

Fact. 5th: N=1,	Finished: returns 1
Fact. 4th: N=2,	Unfinished: 2*Fact(1)
Fact. 3rd: N=3,	Unfinished: 3*Fact(2)
Fact. 2nd: N=4,	Unfinished: 4*Fact(3)
Fact. 1st: N=5,	Unfinished: 5*Fact(4)
Main Function:	Unfinished: X = Fact (5);

Activation Stack for Factorial

Fact. 4th: N=2,	Finished: returns 2*1
Fact. 3rd: N=3,	Unfinished: 3*Fact(2)
Fact. 2nd: N=4,	Unfinished: 4*Fact(3)
Fact. 1st: N=5,	Unfinished: 5*Fact(4)
Main Function:	Unfinished: X = Fact (5);

Activation Stack for Factorial

Fact. 3rd: N=3,	Finished: returns 3*2
Fact. 2nd: N=4,	Unfinished: 4*Fact(3)
Fact. 1st: N=5,	Unfinished: 5*Fact(4)
Main Function:	Unfinished: X = Fact (5);

Activation Stack for Factorial

Fact. 2nd: N=4,	Finished: returns 4*6
Fact. 1st: N=5,	Unfinished: 5*Fact(4)
Main Function:	Unfinished: X = Fact (5);

Activation Stack for Factorial

Fact. 1st: N=5, Finished: returns 5*24
Main Function: Unfinished: X = Fact (5);

Activation Stack for Factorial

Main Function: finished: X = 120;

Recursive Function: X^n

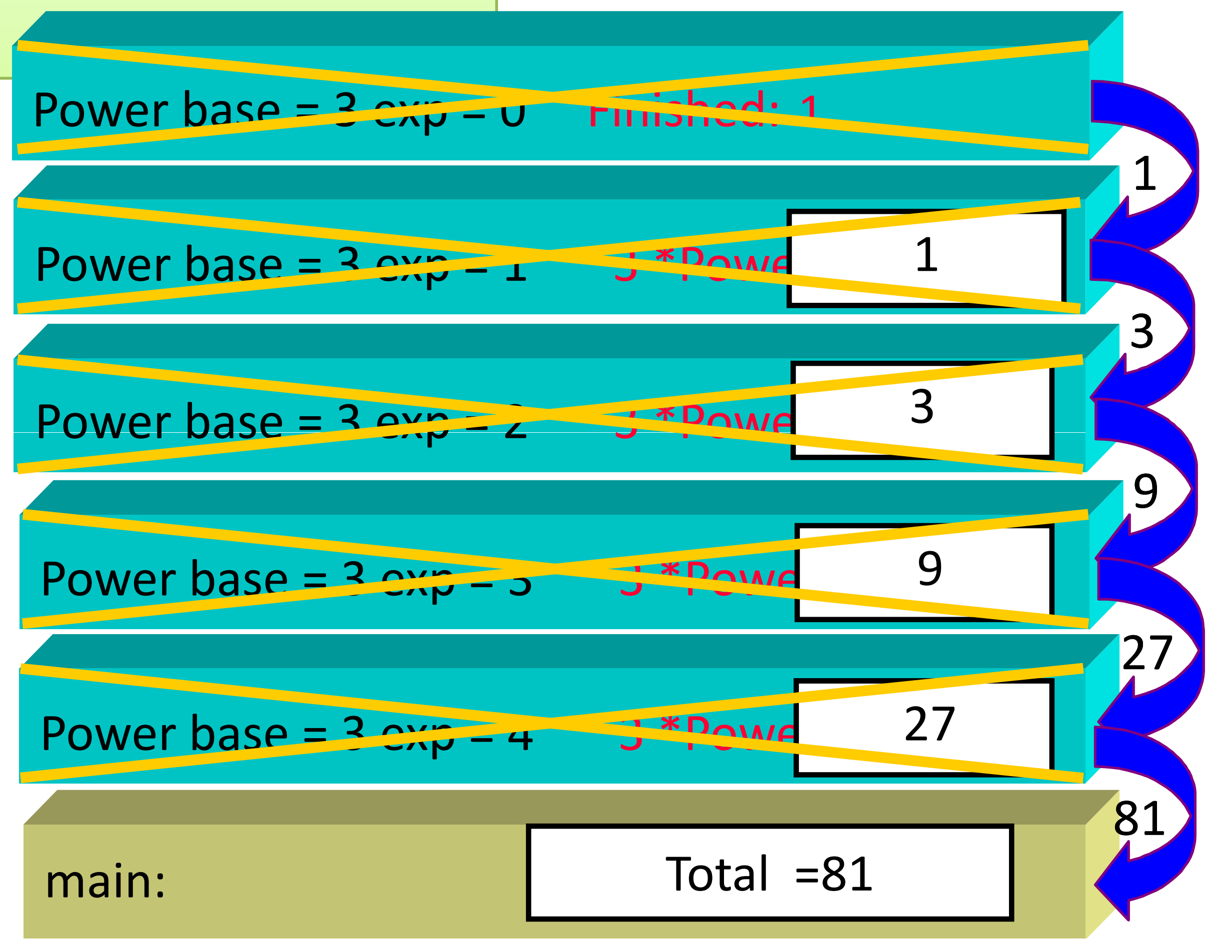
- $X^n = X * X^{(n-1)}$
- When n <=1; X^n=X

- Power(base, exp)

  = base * Power(base, exp-1);

Recursive Function: X^n

```
int Power (int  base, int  exp){
        if(exp <=0)   return 1;
        return  X *Power(base, exp-1);
}
```



The n^{th} power of X

- **Is there any better approach?**
- From basic algebra
  - if n is even == > $X^n = X^{n/2}.X^{n/2}$
  - If n is odd and n=2m+1 ==> $X^n = X^{2m+1} = x^m . x^m . x$
- From this above fact, can we calculate X^n in fewer steps

```
int Pow (int  X, int  N){
     if(N <=0)  return 1;
     int Y=Pow(X,N/2);
     if(N %2==0)  return  Y*Y;
     else   return  X*Y*Y;
}
```

A More Complex Recursive Function

Fibonacci Number Sequence

if $n = 1$, then $Fib(n) = 1$

if $n = 2$, then $Fib(n) = 1$

if $n > 2$, then $Fib(n) = Fib(n-2) + Fib(n-1)$

Numbers in the series:

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Fibonacci Sequence Function

```
int  Fib (int n){
  if (n == 1) ||(n == 2))
    return 1;
  else
   return Fib(n-2)+Fib(n-1);
}
```

```
main(){
 int answer; answer=fib(5);
 printf(“5 th Fib Num is %d”, answer);
}
```

Tracing with Multiple Recursive Calls

Main :	answer = Fib(5)

Tracing with Multiple Recursive Calls

Fib(5):	Fib returns Fib(3) + Fib(4)
Main :	answer = Fib(5)

Tracing with Multiple Recursive Calls

Fib(3):	Fib returns Fib(1) + Fib(2)
Fib(5):	Fib returns Fib(3) + Fib(4)
Main :	answer = Fib(5)

Tracing with Multiple Recursive Calls

Fib(1):	Fib returns 1
Fib(3):	Fib returns Fib(1) + Fib(2)
Fib(5):	Fib returns Fib(3) + Fib(4)
Main :	answer = Fib(5)

Tracing with Multiple Recursive Calls

Fib(3):	Fib returns 1 + Fib(2)
Fib(5):	Fib returns Fib(3) + Fib(4)
Main :	answer = Fib(5)

Tracing with Multiple Recursive Calls

Fib(2):	Fib returns 1
Fib(3):	Fib returns 1 + Fib(2)
Fib(5):	Fib returns Fib(3) + Fib(4)
Main :	answer = Fib(5)

Tracing with Multiple Recursive Calls

Fib(3):	Fib returns 1 + 1
Fib(5):	Fib returns Fib(3) + Fib(4)
Main :	answer = Fib(5)

Tracing with Multiple Recursive Calls

Fib(5):	Fib returns 2 + Fib(4)
Main :	answer = Fib(5)

Tracing with Multiple Recursive Calls

Fib(4):	Fib returns Fib(2) + Fib(3)
Fib(5):	Fib returns 2 + Fib(4)
Main :	answer = Fib(5)

Tracing with Multiple Recursive Calls

Fib(2):	Fib returns 1
Fib(4):	Fib returns Fib(2) + Fib(3)
Fib(5):	Fib returns 2 + Fib(4)
Main :	answer = Fib(5)

Tracing with Multiple Recursive Calls

Fib(4):	Fib returns 1 + Fib(3)
Fib(5):	Fib returns 2 + Fib(4)
Main :	answer = Fib(5)

Tracing with Multiple Recursive Calls

Fib(3):	Fib returns Fib(1) + Fib(2)
Fib(4):	Fib returns 1 + Fib(3)
Fib(5):	Fib returns 2 + Fib(4)
Main :	answer = Fib(5)

Tracing with Multiple Recursive Calls

Fib(1):	Fib returns 1
Fib(3):	Fib returns Fib(1) + Fib(2)
Fib(4):	Fib returns 1 + Fib(3)
Fib(5):	Fib returns 2 + Fib(4)
Main :	answer = Fib(5)

Tracing with Multiple Recursive Calls

Fib(3):	Fib returns 1 + Fib(2)
Fib(4):	Fib returns 1 + Fib(3)
Fib(5):	Fib returns 2 + Fib(4)
Main :	answer = Fib(5)

Tracing with Multiple Recursive Calls

Fib(2):	Fib returns 1
Fib(3):	Fib returns 1 + Fib(2)
Fib(4):	Fib returns 1 + Fib(3)
Fib(5):	Fib returns 2 + Fib(4)
Main :	answer = Fib(5)

Tracing with Multiple Recursive Calls

Fib(3):	Fib returns 1 + 1
Fib(4):	Fib returns 1 + Fib(3)
Fib(5):	Fib returns 2 + Fib(4)
Main :	answer = Fib(5)

Tracing with Multiple Recursive Calls

Fib(4):	Fib returns 1 + 2
Fib(5):	Fib returns 2 + Fib(4)
Main :	answer = Fib(5)

Tracing with Multiple Recursive Calls

Fib(5):	Fib returns 2 + 3

Main :	answer = Fib(5)

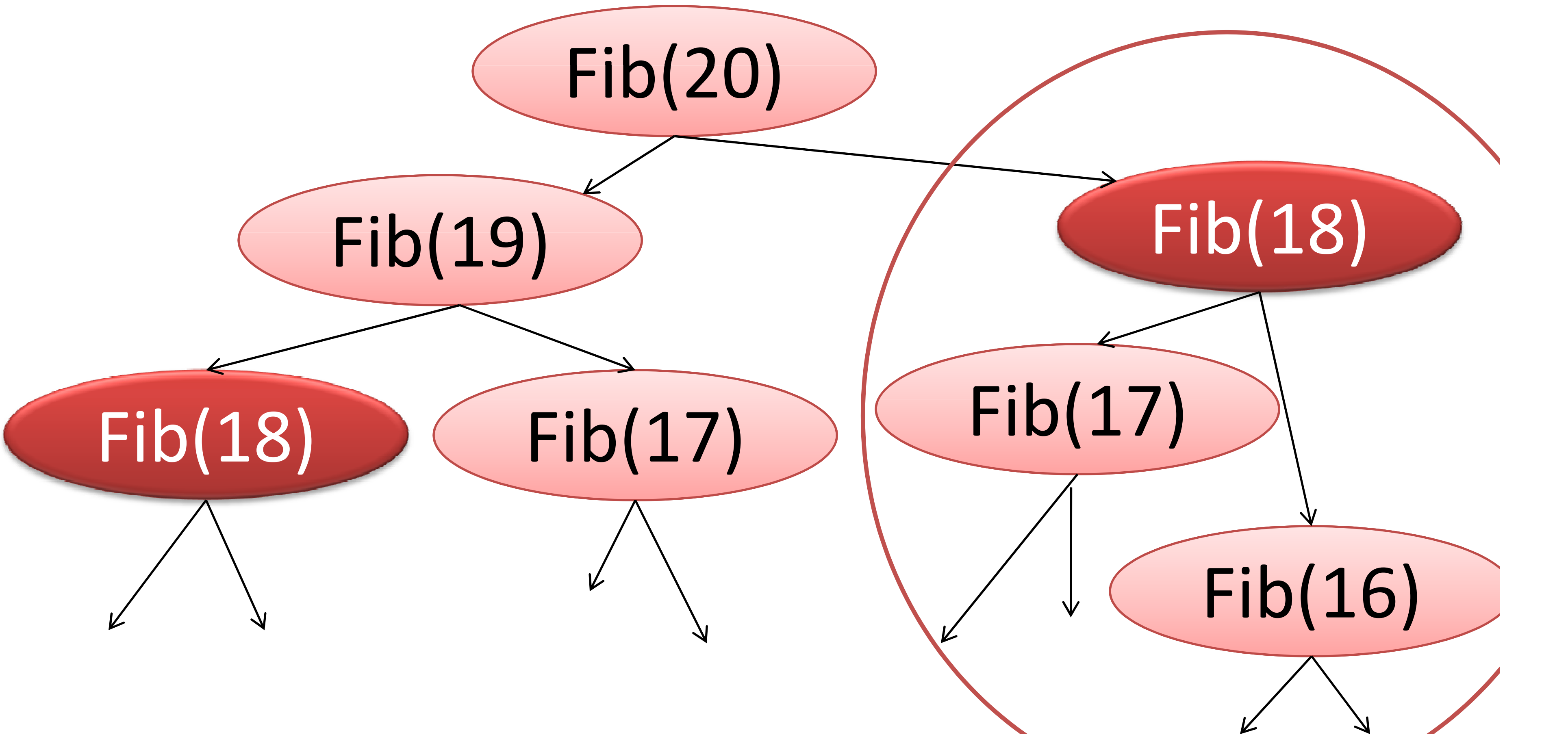
Tracing with Multiple Recursive Calls

Main :	answer = 5

Fib (N): Number of Recursive Call

- Multiple Recursive Calls

Fib(N) =Fib(n-1) + Fib(n-2)



Fib (N): Number of Recursive Call

- Multiple Recursive Calls

  Fib(N) =Fib(n-1) + Fib(n-2)

- Number of recursive call for N
  - Claim: Number of recursive call for Fib(n-1) is higher than number of recursive call for Fib(n-2)
  - Denote number of recursive call for Fib(n) $= f_n$
  - $f_{n-1} > f_{n-2}$

Fib (N): Number of Recursive Call

- Can I Say : $f_n = f_{n-1}+f_{n-2} > f_{n-2}+f_{n-2} = 2.\ f_{n-2}$
- Then $f_n > 2.f_{n-2} > 2.2.f_{n-4} > 2.2.2.f_{n-6}$

  $= \ldots = 2^{n/2} f_1$ **So $f_n > 2^{n/2}$**
- Number of recursive call require to compute Fib(n) is $> 2^{n/2}$
- Can you calculate for 200 Fibonacci using recursive program
  - **Will take at least 2^{100} recursive call : huge time and space**

Thanks