

Lecture 10: Recursion

1. Introduction
2. Recursive functions / subroutines

Introduction

- Recursion: A subprogram (i.e. a function or a subroutine) calls itself.
- Direct: the call is made inside the function. A calls A calls A
- Indirect: the call is made from another function/subroutine that is called from the recursive sub program. A calls B calls A
- Example: Factorials

```
recursive function fact(n)
```

```
implicit none
```

```
integer:: n
```

```
real(8):: fact
```

```
if(n==1) then
```

```
    fact=1.0
```

```
else
```

```
    fact=real(n,8)*fact(n-1)
```

```
endif
```

```
end function fact
```

```
function fact(n)
```

```
implicit none
```

```
integer:: n, i
```

```
real(8):: fact
```

```
fact=1.0
```

```
do i=1, n
```

```
    fact=fact*real(i,8)
```

```
enddo
```

```
end function fact
```

Structure of recursive solutions

Every recursive solution involves two major parts or cases, the second part having three components.

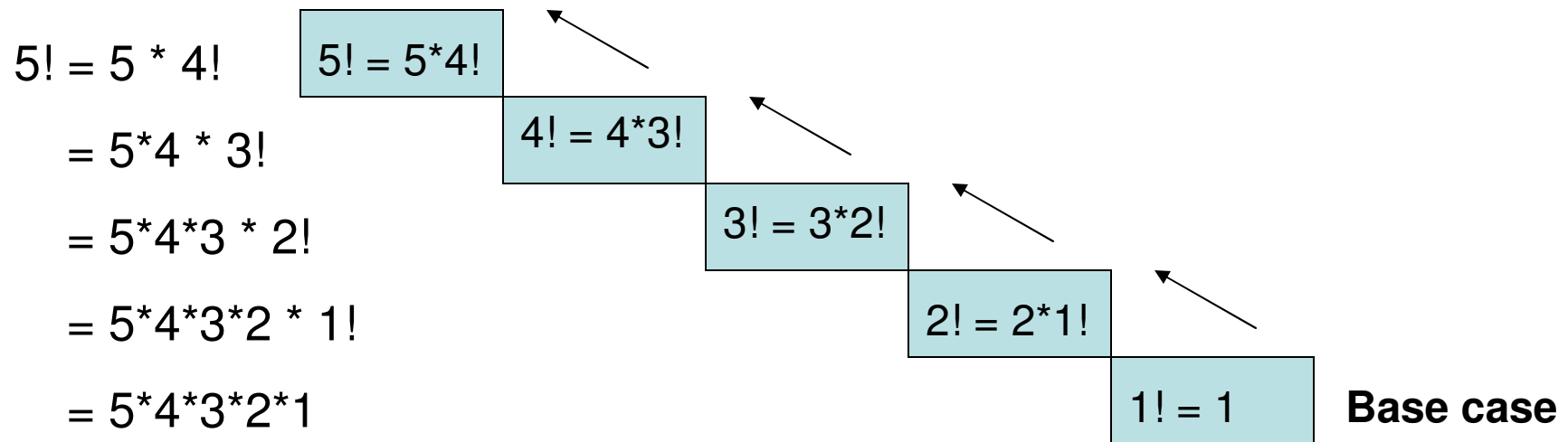
1. **base case**: The problem is simple enough to be solved directly. This terminates the recursion.
2. **recursive case**: A recursive case has three components:
 - **divide** the problem into one or more simpler or smaller parts of the problem,
 - **call** the function (recursively) on each part, and
 - **combine** the solutions of the parts into a solution for the problem.

Depending on the problem, any of these may be trivial or complex.

Any function that can be evaluated by a computer can be expressed in terms of recursive functions, without use of iteration, and conversely.

Recursive factorials

- Example: $n!$ for $n=5$

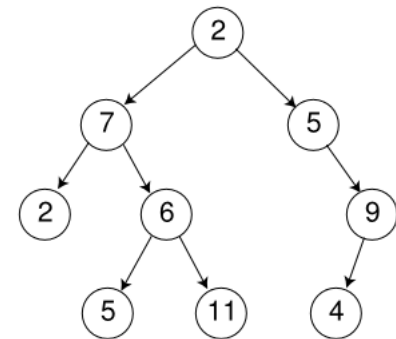


- The *recursion depth* is 5 in this case

Comments on recursion

- **recursion vs. iteration:** While recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code, it is often less efficient, in terms of both time and space, than iterative solutions.
- The main benefits of using recursion as a programming technique are:
 - invariably recursive functions are clearer, simpler, shorter, and easier to understand than their non-recursive counterparts.
 - the program directly reflects the abstract solution strategy (algorithm).

- There are other types of problems that seem to have an inherently recursive solution, i.e. they need to keep track of prior state. One example is traversal of a tree of unknown depths and breadth. This is a natural structure for games.



- During recursion, variables and returned results are put on the *stack*. This limits the depth of recursion. Use command `ulimit -s unlimited`

Example: Fibonacci sequence

```
function fib(n)
  if n = 0 or n = 1
    return 1
  else
    return fib(n - 1) + fib(n - 2)
```

$$F(n) = \begin{cases} 0, & \text{if } n = 0; \\ 1, & \text{if } n = 1; \\ F(n - 1) + F(n - 2) & \text{if } n > 1. \end{cases}$$

1. fib(5)
2. fib(4) + fib(3)
3. (fib(3) + fib(2)) + (fib(2) + fib(1))
4. ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
5. (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

Problem: fib(2) is computed twice. For higher n, many Fibonacci numbers will be computed very often, and the execution time of the recursive algorithm grows exponentially with n. Here, an iterative solution is preferred.

More interesting problems

Problem: A walker on a 2D lattice is at the origin (0,0) and has to walk to the point (n,m). How many ways are there, if the walker can only make steps in positive x and y direction? Output from right file for (n,m)=(2,2)

```
0, 0, 0
1, 0, 0
2, 0, 0
2, 1, 0
2, 2, 0
1, 1, 1
2, 1, 1
2, 2, 1
1, 2, 2
2, 2, 2
0, 1, 3
1, 1, 3
2, 1, 3
2, 2, 3
1, 2, 4
2, 2, 4
0, 2, 5
1, 2, 5
2, 2, 5
```

Note: It's easy to implement obstacles.

```
module steps
  implicit none
  integer:: count
  integer, parameter:: imax=2, jmax=2
```

contains

```
recursive subroutine step(i,j)
  implicit none
  integer:: i, j
```

```
  write(*,*) i,j, count
  if(i.lt.imax) call step(i+1,j)
  if(j.lt.jmax) call step(i,j+1)
  if(i.eq.imax .and. j.eq.jmax)
    count=count+1
```

end subroutine step

end module steps

```
program main
  use steps
  implicit none
```

```
  count=0
  call step(0,0)
  write(*,*) count
end program main
```

Number of partitions of integers

```
program main
  implicit none
  integer:: num
  integer:: n

10  write(*,*) 'input n'
    read(*,*) n
    if(n.gt.10) goto 10

    num=0
    call partition(n,n,num)
    write(*,*) 'number of partitions of ',n, ': ',num
end program main

recursive subroutine partition(n, limit, num)
  implicit none
  integer:: n, limit, num
  integer:: i

  if(n.gt.0) then
    do i=min(n, limit), 1, -1
      call partition(n-i,i,num)
    enddo
  else
    num=num+1
  endif
end subroutine partition
```