Recursion & Recursive Functions

Nayeb Maleki

What is recursion?

- Recursion is a technique that helps the programmer to break up large, complicated problems in one or more smaller sub-problems that are similar to the original problem. An "elegant" programming technique that is largely similar to the way we think and do.
- Recursion is an iterative process (which is repeated a number of times.), Where you have to specify what should be done if a particular event occurs, for example terminate the iterative process.
- Suppose we have a problem of the order of N, and we can divide the problem into two smaller ones of size N-1 and 1. Recursively, we can do the same with the N-1 until we have N number of problems on the order of 1, which in practice will be easier to solve!
- Think about how you would look for lost keys at home. Can you take all rooms in one go?

What is recursive function?

- A recursive function is a function that calls itself directly or indirectly from another function which in turn calls the first until the process ends with a condition.

- Recursive functions are built up in three steps:

  - 1. Identify a "Base Case": an instance of the problem that has a trivial solution. Such as: Exit if we have reached 0

2. Identify the recursive part. The part that recursively will be broken up into smaller parts. For example, by reducing N by 1, which is N-1 (see last picture).

3. Creating an algorithm by combining steps 1 and 2.

Example

- Direct recursion

```
int tal(int x)
{
    if(x<=0)
        return x;
  return tal(x-1);
}
```

- Indirect recursion

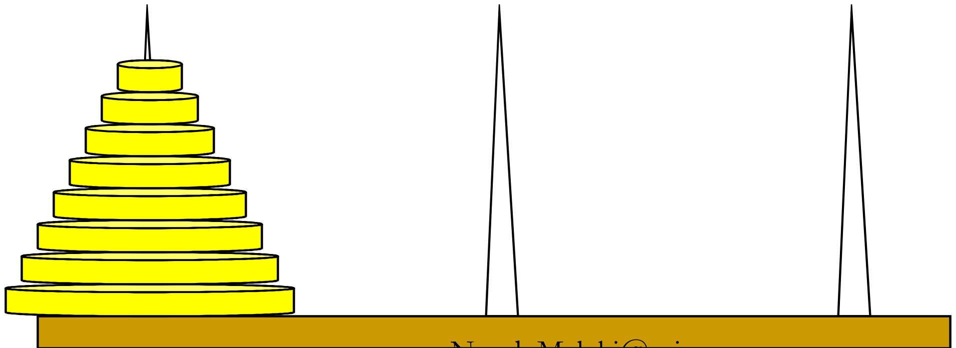
```
int tal(int x)
{
 if( x<=0) return x;
 return another(x);
}

 int another(int x)
  {
    return tal(x-1);
   }
```

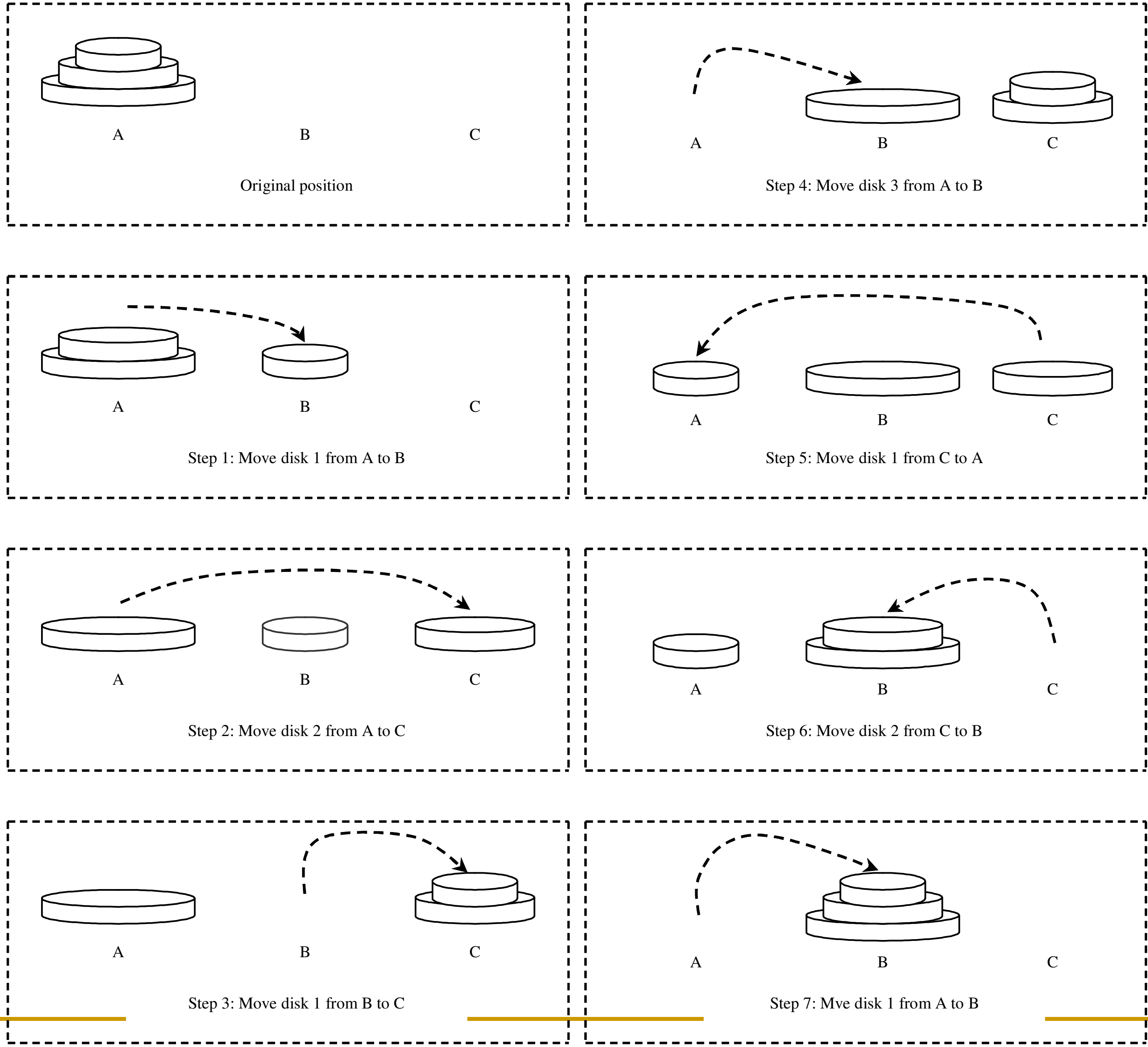
Examples of major problems

Towers of Hanoi

- There are *n* disks labeled 1, 2, 3, . . ., *n*, and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time, and it must be the top disk on the tower.

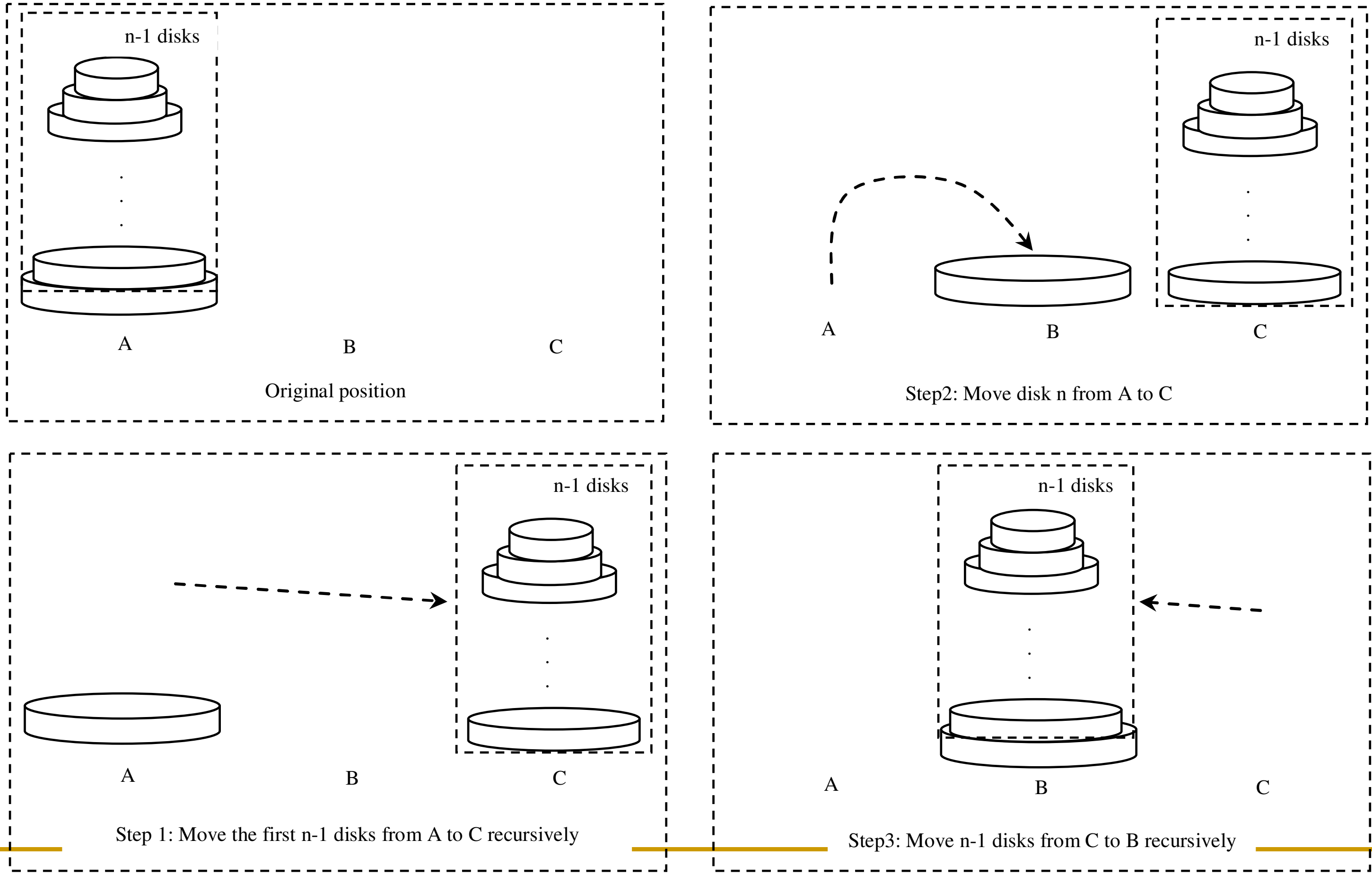


Towers of Hanoi, cont.



Solution to Towers of Hanoi

The Towers of Hanoi problem can be decomposed into three subproblems.



Examples of major problems

Analyzing the problem, one can see how many movements it must be done to solve the problem:

n	Number of movements
1	1
2	3
3	7
4	15
5	31
...	....
i	2^i-1
64	$2^{64}-1$ **@** 2^{11} century !!!

Examples of major problems: solution

- In fact, it would be very time consuming if possible to solve the problem without recursion that provides a very simple and elegant solution in couple of lines!

```
void Move(int n, char src, char dest, char aux)
{
  if (n > 1)
  {
    Move(n-1, src, aux, dest);
    Move(1, src, dest, aux);
    Move(n-1, aux, dest, src);
  }
  else
    cout << "Move the top disk from "
         << src << " to " << dest << endl;
}
```

Example: Lord of the Cake!

- Suppose you invite your friends to your birthday-party. The cake is on the table and begging to be eaten, but your friends, they belong to the weight watchers group! and refuse to eat so you are forced to eat it yourself. You can't resist the temptation also its impossible to eat the whole cake at once. You start to cut it into say 10 nice pieces and eat piece by piece, first piece # 1, then piece # 2, piece # 3, and so on. .. . until the whole cake is eaten. Obviously you can have piece # 10 first, until you have reached piece # 0 which does not exist and thus the process is completed.

Example: Lord of the Cake! (cont.)

- This is how it looks like :

The iterative process of eating a whole cake :

----

(1) **eat piece #1**

(2) **eat piece #2**

(3) **eat piece #3**

… …….

(10) eat piece #10

(11) The cake is gone!

- Or like this:

The iterative processe of eating a whole cake:

----

(1) **eat piece #10**

(2) **eat piece #9**

(3) **eat piece #8**

… …….

(10) eat piece #1

(11) eat piece #0 = The cake is gone!

Example: Lord of the Cake! Create an algorithm

- 3 steps:
  1. Identify a "Base Case": an instance of the problem that has a trivial solution.

     In this case it will be:

     return if we have reached 0(if ***cakePieces***== 0)

  2. Identify the ***recursive part***.

     In this case it will be:

     Reduce ***cakePieces*** with ***1*** for each time the iterative process is ongoing.

Example: Lord of the Cake! Create an algorithm

3. Creating an algorithm from the "trivial" step (Base Case) and the induction step.

- **The iterative process of eating a whole cake:**

--

```
Datatype function_eatckae( datatype cake)
    {
        if (cake is finished)
            Say:  The cake is finished.
            do:  return.
        else
            Say:  One piece has been eaten, more I want.
            do:  go to function_eatcake(cake – one_piece);
    }
```

Example: Lord of the Cake! Create an algorithm

- The code in C++:

```
int eatcake(int cake)
 {
   if (cake==0)
    {
      cout<<" The cake is finished. "<<endl;
      return (0); //return!!
     }
  else
   {
       cout One piece has been eaten."<<endl;
       return eatcake(cake -1);
     }
   //return 0;
 }
```

```
#include <iostream>
 using namespace std;

 int eatcake(int cake);

void main()
 {
     eatcake(10);
   }
```

Exercise!:

Given a number n, write a recursive function that computes n!

n! == n*(n-1) ***... * 2 *1**

Example:

0!=1

1!=1

5! == 5 * 4 * 3 * 2 * 1 == 120

Specification:

Read in : *n*, a number.

Condition that needs to be true (Precondition) : *n* >= 0

ie. (0! == 1 && 1! == 1).

Return : *n!*

Preliminary try without recursion

```
long Fakultet(int n)
{
  long result= 1;

  if(n==0 ||n==1)  //if(n <= 1 )
    return result;

  for (int i = 2; i <= n; i++)
    result *= i;

  return result;
}
```

Testing the solution:

```
#include <iostream>
#include <iomanip>
using namespace std;
long Fakultet(int n);
void main()
 {
    for(int i=0;  i<=5 ; i++)
    cout<< Fakultet(i) << setw(5);
    cout << endl;
  }
```

Utskrift: 1 1 2 6 24 120

Recursive version:

Reminder:

A function that is defined in terms of itself is called a "self-referencing" function or a recursive function.

1) Base Case, the triviala part:

if n == 0 → n! == 1

if n == 1 → n! == 1

2) Recursive part:

for each step, change n into n-1, ie. break it into smaller parts until there are nothing left to break. That is until n=1. When this (n=1) accurs the ”base case” is true and the process will end.

Algorithm:

```
// Fakultet(n)
0. Read a value to n
1. if n == 0 or n==1
      return 1
2. else
      return n* fakultet(n-1)
```

C++ code:

```
//alt. 1
long fakultet(int n)
{
  if (n ==0 || n ==1)
    return 1;

  return n * fakultet(n-1);
}
```

```
//alt. 2
long fakultet(int n)
{
  if (n <= 1)
    return 1;

  return n * fakultet(n-1);
}
```

Alternative solutions:

```
//alt. 3
//tail recursion for n>=0

int fakultet (int n, int result)
{
  if (n==0 || n==1)
    return result;
  return fakultet(n-1, n*result);
}
```

```
//alt. 4
//Linear recursion for all n
long fakultet(int n)
{
  if (n > 1)
    return n * fakultet(n-1);

  return 1
}
```

Tail and linear Recursion

A recursive function is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call.

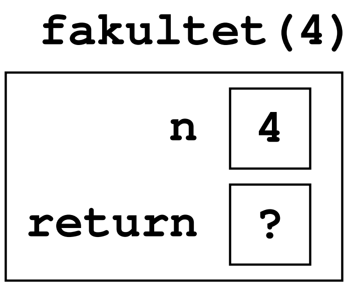
In *Linear recursion* the function calls itself repeatedly until it reaches the base case. After reaching the base case condition, it simply returns the result to the caller through a process called unwinding.

Tracing alt.4 :

Assume we call the function with $n = 4$.

```
long fakultet(int n)
{
  if (n > 1)
    return n* fakultet(n-1);
  return 1;
}
```

Function starts for n=4.




```
long fakultet(int n)
{
  if (n > 1)
    return n * fakultet(n-1);
  return 1;
}
```

if-statement executes for 4 > 1, ...

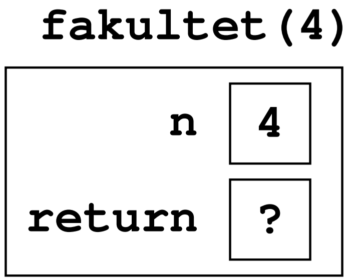
And the returnvalue is being calculated, but it needs to call fakultet(3) first.

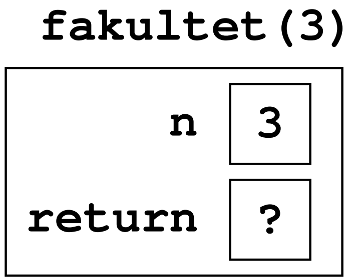
fakultet(4)

n	**4**
return	**?**

```
long fakultet(int n)
{
  if (n > 1)
    return n* fakultet(n-1);
  return 1;
}
```

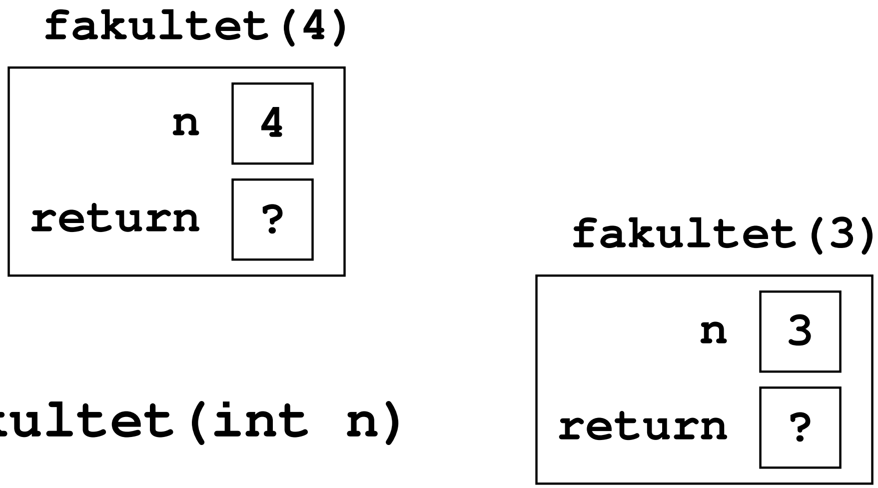
This begins with a new execution which means that the function is called again for n = 3.






```
long fakultet(int n)
{
  if (n > 1)
    return n * fakultet(n-1);
  return 1;
}
```

if-statement executes for 3 > 1, ... and calculates the return value again, that is calling faculty (2) first.




```
long fakultet(int n)
{
  if (n > 1)
    return n* fakultet(n-1);
  return 1;
}
```

This starts an execution for n = 2

fakultet(4)

n	4
return	?

fakultet(3)

n	3
return	?

fakultet(2)

n	2
return	?

```
long fakultet(int n)
{
  if (n > 1)
    return n* fakultet(n-1);
  return 1;
}
```

if-statement executes for 2 > 1, ...

Which tries to calculate the return value, ie. the call fakultet(1).

fakultet(4)

n	4
return	?

fakultet(3)

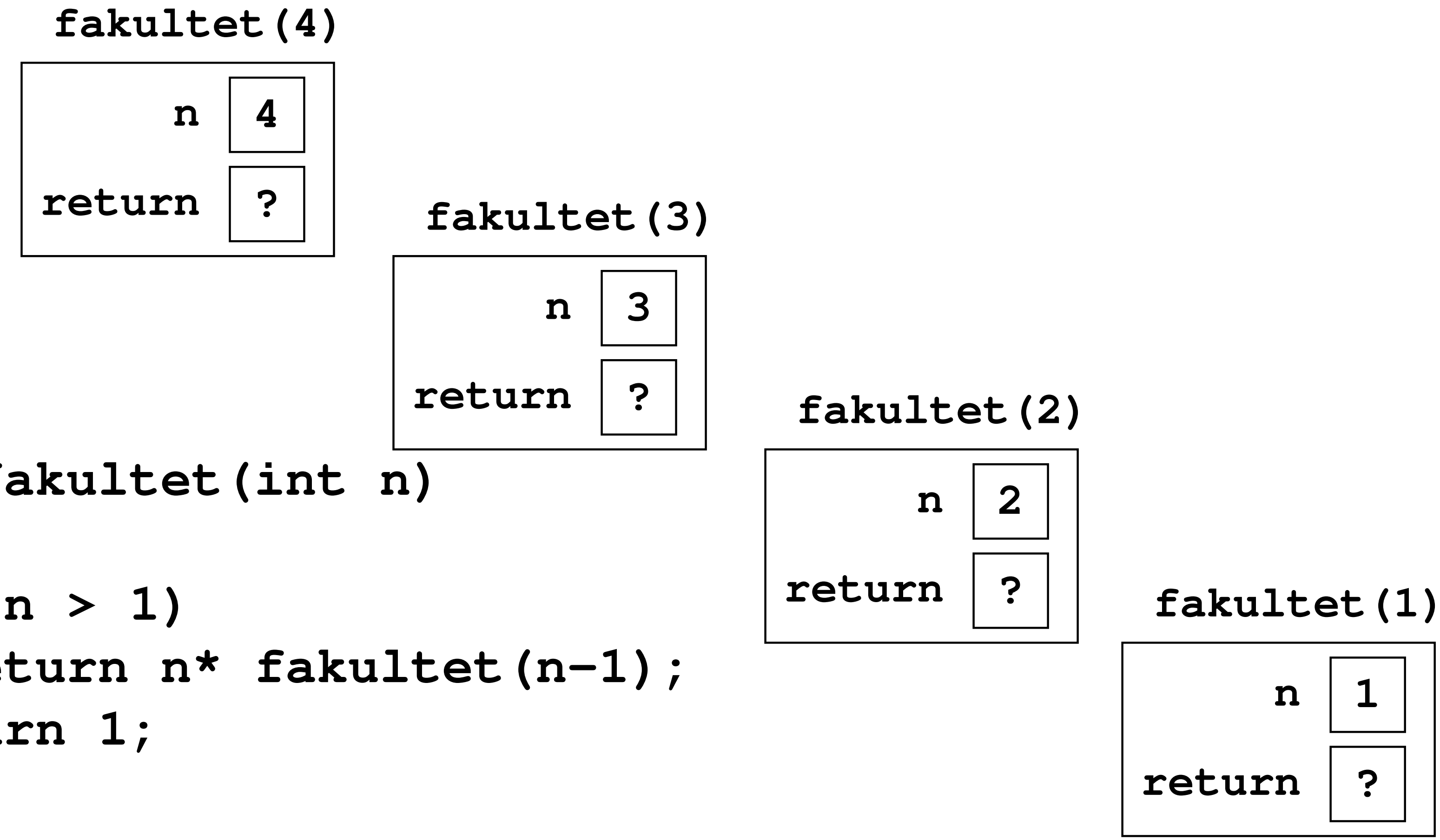
n	3
return	?

fakultet(2)

n	2
return	?

```
long fakultet(int n)
{
  if (n > 1)
    return n * fakultet(n-1);
  return 1;
}
```

This starts an execution for n = 1.




```
long fakultet(int n)
{
  if (n > 1)
    return n* fakultet(n-1);
  return 1;
}
```

if-satement executes, and the condition n > 1 is *false*.

fakultet(4)

n	4
return	?

fakultet(3)

n	3
return	?

fakultet(2)

n	2
return	?

fakultet(1)

n	1
return	?

```
long fakultet(int n)
{
  if (n > 1)
    return n* fakultet(n-1);
  return 1;
}
```

This means return 1; executes.

fakultet(1) finishes and returns 1 to fakultet(2).

fakultet(4)

n	4
return	?

fakultet(3)

n	3
return	?

fakultet(2)

n	2
return	?

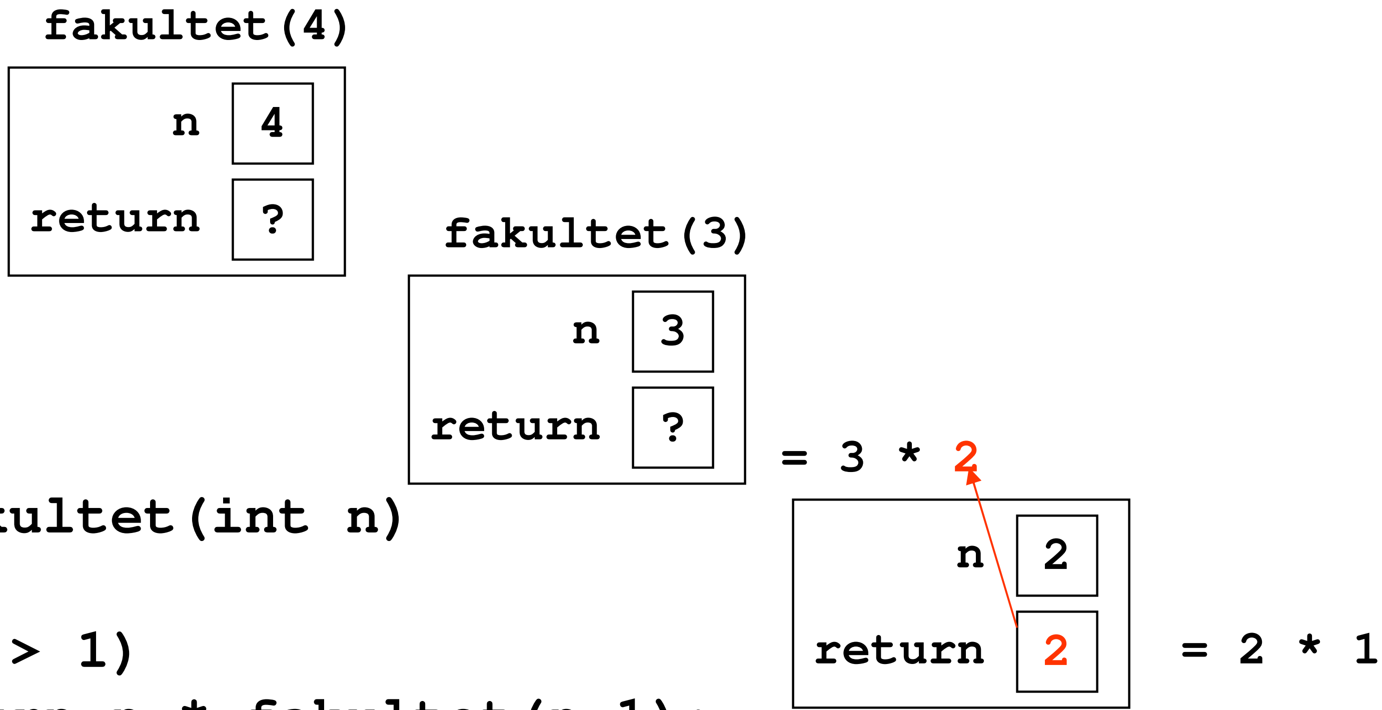
= 2 * 1

fakultet(1)

n	1
return	1

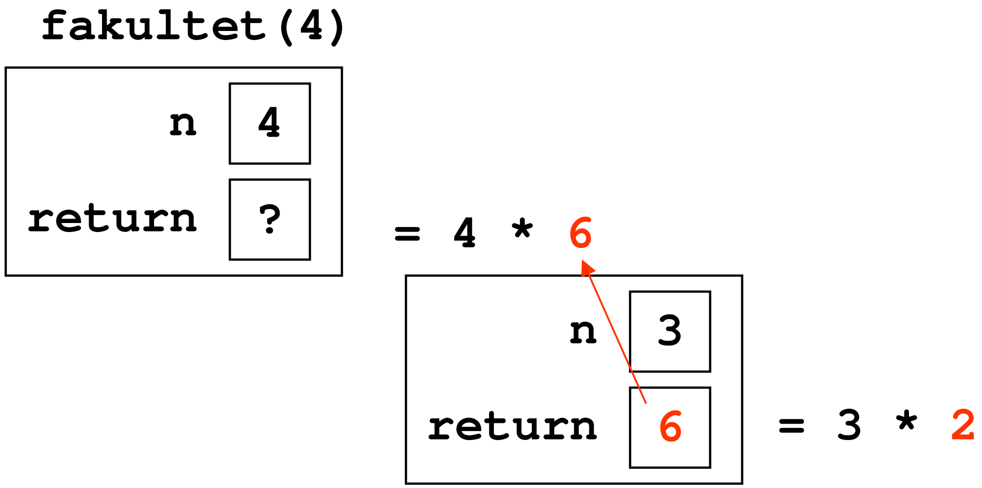
```
long fakultet(int n)
{
  if (n > 1)
    return n* fakultet(n-1);
  return 1;
}
```

fakultet(2) continues from where it stopped last time, calculates the return value(2) and sends to fakultet(3)




```
long fakultet(int n)
{
  if (n > 1)
    return n * fakultet(n-1);
  return 1;
}
```

fakultet(3) continues from where it stopped last time, calculates the return value(6) and sends to fakultet(4)




```
long fakultet(int n)
{
  if (n > 1)
    return n * fakultet(n-1);
  return 1;
}
```

fakultet(4) continues from where it stopped last time, calculates the return value and sends the result to where fakultet(4) was called (main function).

main( )

```
int n=4;
cout<< fakultet(n) <<endl;
```

fakultet(4)

n	4
return	24

= 4 * 6

```
long fakultet(int n)
{
  if (n > 1)
    return n* fakultet(n-1);

  return 1;
}
```

Example: Multiply the two numbers recursively

Assume that the two numbers are m=6 and n=3:

- Break the problem in two subproblems.
  1. Multiply 6 with 2
  2. Add 6 to the result from subproblem 1.

- Subproblem 1 can be broken even more:

  1.1 Multiply 6 with 1.

  1.2 Add 6 to the result.

  2. Add 6 to the result from subproblem 1.

Solution: 1th try

- //multiplication of two numbers:

```
int multiply(int m,int n)  //assume n>0
{
  if(n==1)
    return m;
   return m+multiply(m, n-1);
}
```

Trace an execution

- Call with: multiply(6,3)

m is 6 n is 3

3==1 is false

return 6 +multiply(6,2)

18

12

m is 6 n is 2

2==1 is false

return 6 +multiply(6,1)

6

m is 6 n is 1

1==1 is true

return 6

Solution: 2nd try

- //multiplication of two numbers:
- **//assume n>=0 & m>0**

```
int multiply(int m,int n)
{
 if(n==0)
   return 0;
 if(n==1)
     return m;
  return m+multiply(m, n-1);
}
```

Problem Solving Using Recursion

Let us consider a simple problem of printing a message for n times. You can break the problem into two subproblems: one is to print the message one time and the other is to print the message for n-1 times. The second problem is the same as the original problem with a smaller size. The base case for the problem is n==0. You can solve this problem using recursion as follows:

```
void nPrintln(string& message, int times)
{
 if (times >= 1) {
   cout << message << endl;
   nPrintln(message, times - 1);
 } // The base case is n == 0
}
```

Think Recursively

Many of the problems can be solved using recursion if you *think recursively*. For example, the palindrome problem can be solved recursively as follows:

```
bool isPalindrome(const string& s)
{
 if (s.size() <= 1) // Base case
  return true;
 else if (s[0] != s[s.size() - 1]) // Base case
  return false;
 else
  return isPalindrome(s.substr(1, s.size() - 2));
}
```

Recursive Helper Functions

The preceding recursive isPalindrome function is not efficient, because it creates a new string for every recursive call. To avoid creating new strings, use a helper function:

```
bool isPalindrome(const string& s, int low, int high)
{
  if (high <= low) // Base case
    return true;
  else if (s[low] != s[high]) // Base case
    return false;
  else
    return isPalindrome(s, low + 1, high - 1);
}
bool isPalindrome(const string& s)
{
  return isPalindrome(s, 0, s.size() - 1);
}
```

Some good advice

- A recursive function has no for/while statement.

For a recursive algorithm to work, these conditions has to be true:

- 1) There must be at least one trivial-part "base case" AND
- 2) at least one recursive part.

- Example that has no trivial part:

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

Some good advice

- The trivial part must occur before the recursive.

- Example that do not have the property:

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

Some good advice

- The recursive part is closer to the trivial portion than the first call.

Example that do not have that property!:

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

- The recursive part really reaches the trivial.

Example that do not have that property!:

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

Some good advice

- Be sure the trivial part is accurate!

Example that do not have that property!:

```
int fact (int n)
{
  if (n == 1)
    return 2;
  return n* fact(n +1);
 }
```

Exercise for you to do it at home!

- Binary search is a good example to solve it recursively. Next slide shows you the solution without recursion.
- Task: Solve/Change it using recursion:

Binary Search without recursion

```
int binarySearch(int a[], int key, int low, int high)  {
int midpoint=0;
while (low <= high) {
 midpoint = low + (high - low) / 2; // find middle of the a[]
// if key lower than middle value, look after the middle value
 if (key > a[midpoint])
      low = midpoint + 1;
  else if (key < a[midpoint])
            high = midpoint - 1;
        else
            return midpoint;
}
return -1;  //This means  low is not <= high
}
```