Recursion

Doan Nhat Quang

doan-nhat.quang@usth.edu.vn
University of Science and Technology of Hanoi
ICT department

Today Objectives

- Introduce recursion and recursive algorithms
- Study well-known problems and solve them with recursive algorithms
- Implement examples in C/C++.

Iteration

Iteration is a repetition of a mathematical or computational procedure applied to the result of a previous application.

Iteration

Iteration is a repetition of a mathematical or computational procedure applied to the result of a previous application.

In programming, iteration is often referred to as looping, because when a program iterates it loops to an earlier step. Iterative approach uses **for (...)** or **while (...) do** loops to solve problems.

- **Example**: calculate the factorial of n.
- **Solution**: to use a loop to run an index i from 1 to n and each iteration, we compute the factorial by multiplying the value of i.

Iteration

Simple pseudo-code to calculate the factorial of an integer

factorial (n)

$fac = 1$
for $i = 1 \rightarrow n$ **do**
    $fac = fac * i$
end for
return fac

Iteration

Iteration gives a lot of advantages:

- Iteration allows to simplify algorithm by stating that we will repeat certain steps until a pre-defined constraint has been reached.
- This makes designing algorithms quicker and simpler because they dont have to include lots of unnecessary steps.
- It is easy to conceptualize or track how a problem can be solved iteratively.

Iteration

Iteration gives a lot of advantages:

- Iteration allows to simplify algorithm by stating that we will repeat certain steps until a pre-defined constraint has been reached.
- This makes designing algorithms quicker and simpler because they dont have to include lots of unnecessary steps.
- It is easy to conceptualize or track how a problem can be solved iteratively.

Iteration is not the only way to deal with problems in which same problems are repeated.

Iteration

Consider the following example: compute the factorial of a given integer *n*

```c
int factorial(int n){
   int fac = 1;
   int i;
   for(i=1;i<=n;i++)
       fac = fac*i;
    return fac;
}
```

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

Iteration

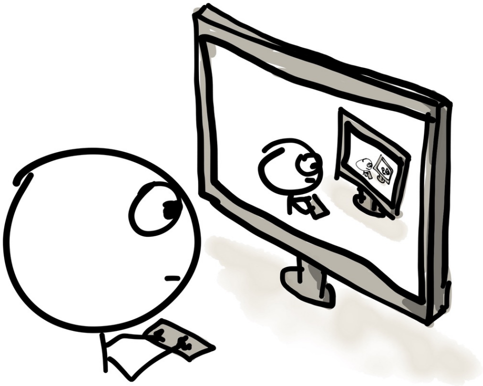
Consider the following example: compute the factorial of a given integer *n*

```
int factorial(int n){
   int fac = 1;
   int i;
   for(i=1;i<=n;i++)
        fac = fac*i;
    return fac;
}
```

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

- Less code and no iteration
- No local variable

Recursion



Recursion

Definition

Recursion in mathematics or in computer science is the process of repeating objects in **a self-similar way**.

Recursion

Definition

Recursion in mathematics or in computer science is the process of repeating objects in **a self-similar way**.

Example 1

Assume that we have a definition of natural numbers as following:

- $0 \in \mathbb{N}$
- if $n \in \mathbb{N}$ then $n + 1 \in \mathbb{N}$
- there are no other objects in the set $\mathbb{N}$.

Due to this definition of natural numbers $\mathbb{N}$, thus 0, $0 + 1 = 1$ are natural. Same for, $0 + 1 + 1 = 2$ is natural, etc.

Recursion

A recursive defnition consists of **two parts**:

- In the first part, called the anchor or the base case, the basic elements that are the building blocks of all other elements of the set are listed.
- In the second part, rules are given that allow for the construction of new objects out of basic elements or objects that have already been constructed.

These rules are applied again and again to generate new objects.

Recursion

Example 2

In mathematics, the factorial of a non-negative integer n, denoted by n!, is the product of all positive integers less than or equal to n.

- $0! = 1 \rightarrow$ base case
- $(n+1)! = (n+1)n! \rightarrow$ rules for the construction of new objects

According to this definition, we generate the sequence of the numbers 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800,... are respectively the factorials of the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,...

Recursion

Proving recursive solutions correct is related to mathematical induction:

- Also closely related to proof by induction
- Start by proving a base case
- Then show that if it is true for case n, it must also be true for case n+1

Recursion in Programming

- Methods can call other methods
- can a method call itself? Yes! This is called a recursive method (function)

Recursive methods:

- Each call solves an identical problem
  - the code is the same!
  - successive calls solve smaller/simpler instances
- Every recursive algorithm has at least one base case
  - a base case (often 1 or 0)

Recursive Methods

Definition

A function is recursive if it calls or defines itself during the execution (direct way).

Recursive Methods

Definition

A function is recursive if it calls or defines itself during the execution (direct way).

Compute the factorial of a given integer *n*

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

Recursive Methods

```
int main() {
  unsigned int r = factorial(5);
}
```

Recursive Methods

```
int main() {
  unsigned int r = factorial(5);
}
```

The obtained results are as following:

Main	factorial(5)			
Call 1	factorial(4)			
Call 2		factorial(3)		
Call 3			factorial(2)	
Call 4				factorial(1)
Result 4				1
Result 3			2*1	
Result 2		3*2*1		
Result 1	4*3*2*1			
Result	5*4*3*2*1 = 120			

Recursive Methods

Definition

A function is indirectly recursive if it calls its invoker and eventually results in the original call.

Recursive Methods

Definition

A function is indirectly recursive if it calls its invoker and eventually results in the original call.

Compute the factorial of a given integer *n*

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

Recursion

Designing Recursive Algorithms

General strategy: Divide and Conquer

- How can we divide the problem into smaller sub-problems?
- How does each recursive call make the problem smaller?
- How do we define the base case?
- Will we always reach the base case?

Recursive Methods

Attention!!

- Instructions must be clear and precise.
- Stopping conditions or base cases are required to avoid infinite recursive calls.

Euclid's Algorithm

- Finds the greatest common divisor of two non-negative integers
- Recursive definition of gcd algorithm
  - if gcd (a, b) = a (if b is 0)
  - if gcd (a, b) = gcd (b, a % b) (if b != 0)

```
int gcd (int a, int b){
    if (b == 0)
            return a;
    else
          return gcd (b, a % b);
}
```

Euclid's Algorithm

```
int gcd (int a, int b){
    int temp;
    while (b != 0){
        temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

```
int gcd (int a, int b){
   if (b == 0)
        return a;
   else
        a = a%b;
        return gcd (b, a);
}
```

Fibonacci Series

Example: Fibonacci numbers 0, 1, 1, 2, 3, 5, 8, ...

```
int fibo(int n){
    if ((n == 0) || (n == 1))  // base cases
        return n;
    return fibo(n-1) + fibo(n-2);
}
```

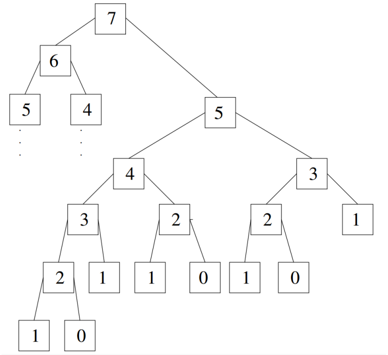
Recursive Algorithms

The picture shows that the solution computes solutions to the subproblems more than once for no reason:

7	6	5	4	3	2	1	0
1	1	2	3	5	8	13	21

→ Complexity is exponential, $O(2^n)$

→ How to reduce the complexity for this problem?



Fibonacci Series

- Reserve a string using Linked Lists and recursion.
- Find the less bill for a given amount of money.

Recursive Methods

Tracing a recursive method:

- As always, go line by line
- Recursive methods may have many copies
- Every method call creates a new copy and transfers flow of control to the new copy
- Each copy has its own:
  - code
  - parameters
  - local variables

Recursive Methods

Tracing a recursive method after completing a recursive call:

- Control goes back to the calling environment.
- Recursive call must execute completely before control goes back to previous call.
- Execution in previous call begins from point immediately following recursive call.

Recursive Types

- **Tail recursion**: a recursive method makes its recursive call as its last step.
  - e.g. recursive Greatest Common Divisor (GCD)
  - can be easily converted to non-recursive methods
- **Binary recursion**: there are two recursive calls for each non-base case
  - e.g. Fibonacci numbers
- **Multiple recursion**: makes potentially many recursive calls (more than one).
- e.g: Fibonacci numbers, merge sort, etc.

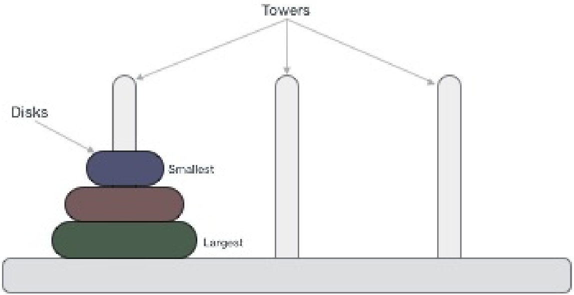
Recursive Types

```
//find max with multiple recursive calls
int max(int arr[],int first,int last){
    if (first==last)
        return arr[first];
    int mid=first+(last-first)/2;
    int a=max(arr, first,mid);
    int b=max(arr, mid+1,last);
    if (a<b)
       return b;
    return a;
}
```

Hanoi Tower

The mission is to move all the disks to some another tower without violating the sequence of arrangement and with few rules as following:

- Only one disk can be moved among the towers at any given time.
- Only the top disk can be removed.
- No large disk can be put over a small disk.



Hanoi Tower

A solution for Hanoi Tower movements.

Hanoi Tower

Problem

Move all n ($n \geq 2$) disks from the source tower S to the destination tower D using a third tower T as a temporary one.

Hanoi Tower

Problem

Move all n ($n \geq 2$) disks from the source tower S to the destination tower D using a third tower T as a temporary one.

Algorithm

- $n = 1$, move $\#1$ disk from S to D

Hanoi Tower

Problem

Move all n ($n \geq 2$) disks from the source tower S to the destination tower D using a third tower T as a temporary one.

Algorithm

- $n = 1$, move $\#1$ disk from S to D
- $n \geq 2$, suppose that we have had $n - 1$ disks from S to T

Hanoi Tower

Problem

Move all n ($n \geq 2$) disks from the source tower S to the destination tower D using a third tower T as a temporary one.

Algorithm

- $n = 1$, move $\#1$ disk from S to D
- $n \geq 2$, suppose that we have had $n - 1$ disks from S to T
  - $\#n$ disk is moved from S to D

Hanoi Tower

Problem

Move all n ($n \geq 2$) disks from the source tower S to the destination tower D using a third tower T as a temporary one.

Algorithm

- $n = 1$, move $\#1$ disk from S to D
- $n \geq 2$, suppose that we have had $n-1$ disks from S to T
  - $\#n$ disk is moved from S to D
  - move $n-1$ disks from T to D

Hanoi Tower

Problem

Move all n ($n \geq 2$) disks from the source tower S to the destination tower D using a third tower T as a temporary one.

Algorithm

- $n = 1$, move $\#1$ disk from S to D
- $n \geq 2$, suppose that we have had $n - 1$ disks from S to T
  - $\#n$ disk is moved from S to D
  - move $n - 1$ disks from T to D (now we consider T as the source tower and S as the temporary tower)

Hanoi Tower

Problem

Move all n ($n \geq 2$) disks from the source tower S to the destination tower D using a third tower T as a temporary one.

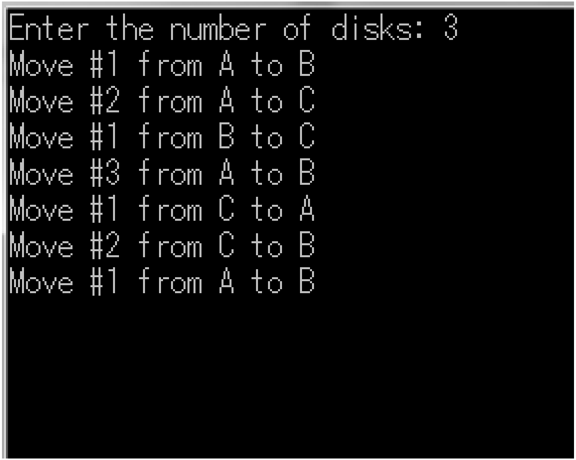
Algorithm

- $n = 1$, move $\#1$ disk from S to D
- $n \geq 2$, suppose that we have had $n - 1$ disks from S to T
  - $\#n$ disk is moved from S to D
  - move $n - 1$ disks from T to D (now we consider T as the source tower and S as the temporary tower)
- Repeat these steps until all the disks are moved to D.

Hanoi Tower

```cpp
#include <cstdlib>
#include <iostream>
#include <conio.h>
using namespace std;
void move(int n, char S, char D, char T){
        if (n == 1)
           cout << "Move #"<< n <<" from " << S <<" to " << D << endl;
        else{
           move(n-1, S, T, D);
           cout << "Move #"<< n <<" from " << S <<" to " << D << endl;
           move(n-1, T, D, S);
        }
}
int main(){
    int n;
     cout<<"Enter the number of disks: ";
     cin>>n;
     move(n, 'A', 'B', 'C');
     getch();
}
```

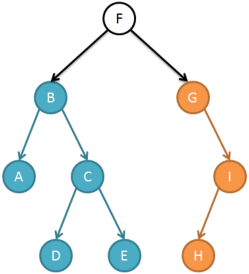
Hanoi Tower



```
Enter the number of disks: 3
Move #1 from A to B
Move #2 from A to C
Move #1 from B to C
Move #3 from A to B
Move #1 from C to A
Move #2 from C to B
Move #1 from A to B
```

Movement solution for 3 disks

Tree Searching Algorithms



Recursive search in a tree

- start searching on the root then descend to the lower level,
- consider the left and right subtrees as *new* trees,
- continue the searching for these trees.

Searching and Sorting Algorithms

We will see several searching and sorting algorithms using recursion in Chapter 5 - Searching and Sorting

Recursion vs Iteration

Roughly speaking, recursion and iteration perform the same kinds of tasks:

- solve a complicated task one piece at a time, and combine the results

Recursion vs Iteration

Roughly speaking, recursion and iteration perform the same kinds of tasks:

- solve a complicated task one piece at a time, and combine the results

- Emphasis of iteration: keep repeating all neccesary steps using results from previous steps until all tasks are done.
- Emphasis of recursion: solve a large problem by breaking it up into smaller and smaller pieces until you can solve it; combine the results.

Recursion vs Iteration

Mathematicians often prefer recursive approach:

- solutions often shorter, closer in spirit to abstract mathematical entity.
- recursive solutions may be more difficult to design and test

Recursion vs Iteration

Mathematicians often prefer recursive approach:

- solutions often shorter, closer in spirit to abstract mathematical entity.
- recursive solutions may be more difficult to design and test

Programmers often prefer iterative approach:

- somehow, it seems more appealing to many.
- controlling loops seems simple and easy.

Which one is better ? Recursion vs Iteration

- No clear answer, but there are known trade-offs
- Recursive isnt always better, e.g. for the fibonacci problem:
    - recursive algorithm has a complexity of $O(2^n)$
    - iterative algorithm costs only $O(n)$

Conclusion

Why recursion?

- Recursion leads to elegant solutions: less code, less need for local variables, etc
- If we can define a function mathematically, the solution is easy to implement.

Conclusion

However

- Once implemented, it is often very difficult to debug a recursive program.
- When reading recursive code, it is sometimes hard to really see how it solves the problem.
- Recursive functions are useful for many cases but we should be careful of using recursion
  - Memory complexity: many function calls, and variable creation
  - Time complexity: many computations.