Recursion 1

Mona Diab
RoadMap

• Introduction to Recursion
• Some examples
• Quick Sort
Definition

• Recursion is a fundamental programming technique that can provide an elegant solution certain kinds of problems

• A function or procedure or method that calls itself, directly or indirectly, is considered recursive

• An algorithm is called **recursive** if it solves a problem by reducing it to an instance of the same problem with smaller input.
Indirect Recursion

- A method invoking itself is considered to be *direct recursion*.
- A method could invoke another method, which invokes another, etc., until eventually the original method is invoked again.
- For example, method \texttt{m1} could invoke \texttt{m2}, which invokes \texttt{m3}, which in turn invokes \texttt{m1} again.
- This is called *indirect recursion*, and requires all the same care as direct recursion.
- It is often more difficult to trace and debug.
Indirect Recursion
Recursive Algorithms

• For every recursive algorithm, there is an equivalent iterative algorithm.

• Recursive algorithms are often shorter, more elegant, and easier to understand than their iterative counterparts.

• However, iterative algorithms are usually more efficient in their use of space and time.
Recursive Thinking

• A *recursive definition* is one which uses the word or concept being defined in the definition itself.

• When defining an English word, a recursive definition is often not helpful.

• But in other situations, a recursive definition can be an appropriate way to express a concept.

• Before applying recursion to programming, it is best to practice thinking recursively.
The Nature of Recursion

1. One or more simple cases of the problem (called the stopping cases or base case) have a simple non-recursive solution.

2. The other cases of the problem can be reduced (using recursion) to problems that are closer to stopping cases.

3. Eventually the problem can be reduced to stopping cases only, which are relatively easy to solve.

In general:

if (stopping case)
    solve it
else
    reduce the problem using recursion
Infinite Recursion

• All recursive definitions have to have a non-recursive part

• If they didn't, there would be no way to terminate the recursive path

• Such a definition would cause infinite recursion

• This problem is similar to an infinite loop, but the non-terminating "loop" is part of the definition itself

• The non-recursive part is often called the base case
Recursive Definitions

• Consider the following list of numbers:

  24, 88, 40, 37

• Such a list can be defined as follows:

  A LIST is a: number
  or a: number comma LIST

• That is, a LIST is defined to be a single number, or a number followed by a comma followed by a LIST

• The concept of a LIST is used to define itself
Recursive Definitions

• The recursive part of the LIST definition is used several times, terminating with the non-recursive part:

```
number comma LIST
  24 , 88, 40, 37

number comma LIST
  88 , 40, 37

number comma LIST
  40 , 37

number
  37
```
Recursive Definitions

• N!, for any positive integer N, is defined to be the product of all integers between 1 and N inclusive

• This definition can be expressed recursively as:

\[ 1! = 1 \]
\[ N! = N \times (N-1)! \]

• A factorial is defined in terms of another factorial

• Eventually, the base case of 1! is reached
Recursive Definitions

\[ 5! \]
\[ 5 \times 4! \]
\[ 4 \times 3! \]
\[ 3 \times 2! \]
\[ 2 \times 1! \]
\[ 1 \]

120
24
6
2
1
Four Criteria of A Recursive Solution

1. A recursive function calls itself.
   – This action is what makes the solution recursive.

2. Each recursive call solves an identical, but smaller, problem.
   – A recursive function solves a problem by solving another problem that is identical in nature but smaller in size.

3. A test for the base case enables the recursive calls to stop.
   – There must be a case of the problem (known as base case or stopping case) that is handled differently from the other cases (without recursively calling itself.)
   – In the base case, the recursive calls stop and the problem is solved directly.

4. Eventually, one of the smaller problems must be the base case.
   – The manner in which the size of the problem diminishes ensures that the base case is eventually is reached.
Four Questions for Constructing Recursive Solutions

1. How can you define the problem in terms of a smaller problem of the same type?

2. How does each recursive call diminish the size of the problem?

3. What instance of the problem can serve as the base case?

4. As the problem size diminishes, will you reach this base case?
Factorial Function – Iterative Definition

\[ n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 \quad \text{for any integer } n>0 \]

\[ 0! = 1 \]

Iterative Definition

```c
fval = 1;
for (i = n; i >= 1; i--)
    fval = fval * i;
```
Factorial Function - Recursive Definition

- To define \( n! \) recursively, \( n! \) must be defined in terms of the factorial of a smaller number.
- Observation (problem size is reduced):
  \( n! = n \times (n-1)! \)
- Base case: \( 0! = 1 \)
- We can reach the base case, by subtracting 1 from \( n \) if \( n \) is a positive integer.

**Recursive Definition:**

\[
\begin{align*}
n! &= 1 & \text{if } n = 0 \\
n! &= n \times (n-1)! & \text{if } n > 0
\end{align*}
\]
Factorial Function

// Computes the factorial of a nonnegative integer.
// Precondition: n must be greater than or equal to 0.
// Postcondition: Returns the factorial of n; n is unchanged.

int fact(int n)
{
    if (n == 0)
        return (1);
    else
        return (n * fact(n-1));
}

• This fact function satisfies the four criteria of a recursive solution.
Why do we build recursive algorithms?

• For many problems, the recursion solution is more natural than the alternative non-recursive or iterative solution.

• It is often relatively easy to prove the correction of recursive algorithms (often proof by induction).

• Easy to analyze the performance of recursive algorithms. The analysis produces recurrence relation, many of which can be easily solved.
Recursive Programming

• A method in Java can invoke itself; if set up that way, it is called a *recursive method*

• The code of a recursive method must be structured to handle both the base case and the recursive case

• Each call to the method sets up a new execution environment, with new parameters and local variables

• As with any method call, when the method completes, control returns to the method that invoked it (which may be an earlier invocation of itself)
Recursive Programming

• Consider the problem of computing the sum of all the numbers between 1 and any positive integer N

• This problem can be recursively defined as:

\[
\sum_{i=1}^{N} i = N + \sum_{i=1}^{N-1} i = N + N - 1 + \sum_{i=1}^{N-2} i
\]

\[
= N + N - 1 + N - 2 + \sum_{i=1}^{N-3} i
\]

\[
\vdots
\]
Recursive Programming

// This method returns the sum of 1 to num
public int sum (int num)
{
    int result;

    if (num == 1)
        result = 1;
    else
        result = num + sum (n-1);

    return result;
}
Recursive Programming

main

sum(3)

= sum

result = 6

sum(2)

result = 3

sum(1)

result = 1

sum
Recursive Programming

• Note that just because we can use recursion to solve a problem, doesn't mean we should

• For instance, we usually would not use recursion to solve the sum of 1 to N problem, because the iterative version is easier to understand

• However, for some problems, recursion provides an elegant solution, often cleaner than an iterative version

• You must carefully decide whether recursion is the correct technique for any problem