Contents

1. Recursion
2. Abstraction
3. Good Programming Practices
This lecture is mainly about recursion.

You've probably all heard of recursion, and in all likelihood think of it as a rather obscure and highly inefficient programming technique. ➔ It is neither!

Recursion is your friend!

Recursion is powerful!
Recursion

Recursion is more than just a programming technique.

It has two other uses in computer science and software engineering, namely:

- as a way of describing, defining, or specifying things.
- as a way of designing solutions to problems (divide and conquer).
How to design recursive algorithms

- Follow general pattern
  - *Wishful thinking*
  - *Decompose the problem*
  - *Identify non-decomposable (smallest) problems*
Wishful thinking

- Assume the desired procedure exist.
- Want to implement *factorial function*? OK, assume it exist.
- BUT, only solves a smaller version of the problem.
Decompose the problem

- **Solve a problem by**
  - Solve a smaller instance (using wishful thinking)
  - Convert that solution to the desired solution

- **Step 2 requires creativity !!!**
  - Must design the strategy before coding
  - \( N! = n*(n-1)*(n-2)\ldots1 = n*(n-1)! \)
  - Solve the smaller instance, multiply it by \( n \) to get the solution
Identify non-decomposable problems

- Decomposing not by itself.
- Must identify the “smallest” problems and solve directly.
- Define $1! = 1$
General form of recursive algorithms

Test, base case, recursive case.

```c
int fact(int n)
{
    if ( n == 1 )
        return 1;  // test for base case
    else
        n*fact(n-1);  // recursive case
}
```

Base case: smallest (non-decomposable) problem

Recursive case: larger (decomposable) problem
Recursive definition

-Mathematical example

_factorial function example:_

- Compute $n$ factorial, defined as $n! = n(n-1)(n-2)\ldots 1$
- Notice that $n! = n \times (n-1)!$ if $n>1$
Another classic is the Fibonacci function:

- \( \text{fibonacci}(0) = 1 \)
- \( \text{fibonacci}(1) = 1 \)
- \( \text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2) \) [for \( n > 1 \)]

1. It has two base cases, not just one; in fact, you can have as many as you like.
2. In the recursive case, there are two recursive calls, not just one. There can be as many as you like.
Recursive definition
- Mathematical example

```c
int s1, s2;
int fibonacci (int n) {
    if (n == 0) return 1;
    else if (n == 1) return 1;
    else {
        s1 = fibonacci(n-1);
        s2 = fibonacci(n-2);
        return s1 + s2;
    }
}
```

This is a common mistake!!
Definition of a Stack

- An intuitive definition of a stack:
  - A stack is either EMPTY
  - or it consists of two parts: (1) a `top' value and (2) a `remainder', which is a stack.

- This follows our normal pattern for a recursive definition.

- Consider the value $S = (7,(29,(11,\text{EMPTY})))$. 
Definition of an Arithmetic Expression.

- An **arithmetic expression** is any of the following:
  - a numeric constant
  - an identifier (of a numeric type variable or constant)
  - an **arithmetic expression** enclosed in parentheses
  - two **arithmetic expressions** separated by a binary arithmetic operator.

- $5 \times (2+3)$?
Summary of recursive process

- Design recursive algorithm by
  - Wishful thinking
  - Decompose the problem
  - Identify non-decomposable (smallest) problem

- Recursive algorithms have
  - Test
  - Recursive case
  - Base case

- What’s the drawback of naïve recursive algorithm ???
  - Can we improve recursive algorithm ???
Exercise 1

Design recursive exponentiation function Exp.

- Input int a, b
- Output a^b
- Restriction:
  - Storage should not increase linearly if possible.
  - But it still works as a recursive algorithm.
Exercise 2
needs your intelligence

The tower of Hanoi

Design a recursive algorithm to move n disks from pole A to pole C !!!

Diagram showing the towers of Hanoi with disks on pole A.
Exercise 2
needs your intelligence

❖ Wishful thinking
  ❖ Assume the desired procedure exist.
    • Move (n, src_pole, spare_pole, dest_pole)

❖ Decompose the problem

Red box is recursive case of original problem !!!!!!!!
Exercise 2
needs your intelligence

General form (test, base case, recursive case)
Move(n, A, B, C)
{
    if (n == 1) ➔ test
        Move disk to pole C directly ➔ base case
    else
    {
        Move(n-1, A, C, B);
        Move(1, A, B, C); ➔ recursive case
        Move(n-1, B, A, C);
    }
}
Abstractions

- Abstractions
  - Procedural
  - Data

- Relationship between data abstraction and procedures that operates on it
- Isolating use of data abstraction from details of implementation
Process of procedural abstraction

- Define formal parameters, capture process in body of procedure
- Give procedure a name
- Hide implementation details from user, who just invokes name to apply procedure

Input: type  procedure  Output: type
Details of contract for converting input to output
Procedural abstractions

- Isolate details of process from its use
- Designer has choice of which ideas to isolate, in order to support general patterns of computation
Data abstraction

- Elements of data abstraction
  - Constructor
  - Accessors
  - Contract
  - Operations
  - Abstraction barrier
  - Concrete representation & Implementation
A rational number is a ratio $n/d$

- $a/b + c/d = (ad+bc)/bd$
- $a/b \times c/d = (ac)/(bd)$
Rational number abstraction

- **Constructor**
  - Make-rat : int, int \(\rightarrow\) Rat \(r\)

- **Accessors**
  - Numer, denom : Rat \(\rightarrow\) int

- **Contract**
  - Numer (make-rat (n, d)) \(\rightarrow\) n
  - Denom (make-rat (n, d)) \(\rightarrow\) d

- **Operations**
  - Various operations

- **Abstraction barrier**
  - Say nothing about it !!!!!

- **Concrete representation & Implementation**
  - Actual C implementation !!!!!
Example

Abstract data type List

ADT operations

Data structure

List

1 2 3 4 5

add remove find display
Example
Abstract data type List

Data structure

List

add(2.5) remove find display

ADT operations
Example
Abstract data type List

Data structure

ADT operations

add  Remove(3)  find  display
Example
Abstract data type stack

ADT operations
- Push
- pop
- peek
- isempty
- popall

Data structure
- 8
- 7
- 6
- 5
- 4
- 3
- 3
- #

Top
Example
Abstract data type stack

ADT operations

Push(12)
pop
peek
isempty
popall

Data structure

12
8
7
6
5
4
3
3
#

Top
Example
Abstract data type stack

ADT operations
- Push
- Pop
- Peek
- Isempty
- Popall

Data structure
- 12
- 8
- 7
- 6
- 5
- 4
- 3
- 3
- #

Top
Good programming practices

- Code design
- Debugging
- Documentation
- Evaluation and verification
Design of

- **Data structures**
  - Natural collection of information
  - Suppression of detail from use of data
- Procedural modules
- Interface
Design of

- Data structures
- **Procedural modules**
  - Computation to be reused
  - Suppression of detail from use of procedure
- Interface
Code layout and design

♦ Design of
  ‣ Data structures
  ‣ Procedural modules
  ♦ Interface
    • “types” of inputs and outputs
Documenting code

- Supporting code maintenance
  - Can you read your code a year after writing it and still understand why you made particular design decision?
- Identifying input/output behaviors
Documenting code

- Description of input/output behavior
- Expected or required types of arguments
- Types of returned value
- List of constraints that must be satisfied by arguments or stages of computation
- Expected state of computation at key points in code
Debugging errors

- Common sources of errors
- Common tools to debug
Common errors

- **Unbound variable**
  - Cause: typo
  - Solution: search for instance

- **Unbound variable**
  - Cause: reference outside scope of binding
  - Solution
    - Search for instance
    - Use debugging tools to isolate instance
Syntax errors

- **Wrong number of arguments**
  - Source: programming error
  - Solution: use debugger to isolate instance

- **Type error**
  - As procedure
  - As arguments
    - Source: Calling error
    - Solution: trace back through chain of calls
Structure error

- Wrong initialization of parameters
- Wrong base case
- Wrong end test
- .....etc
Evaluation and verification

- Choosing good test cases
  - Pick values for input parameters at limits of legal range
    - Base case of recursive procedure
  - Pick values that span legal range of parameters
- Retest prior cases after making code changes
As of next week

We’re going to see the beauty of algorithms