Recursion

- Concept of recursion
- Examples
- Motivation
- How Recursion works
- Supporting Recursion
- Practical Issues

Concept of Recursion

```
int x = 1;
for (int i = l; i<= n; i++)
    x = x * i;
```

What does this <u>iterative</u> code do?

Concept of Recursion:
Factorial Example

n! = n *(n-1)*(n-2)*· · ·3*2*1

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

Concept of Recursion:
Factorial Example

FACT (n) = n * FACT (n-1)
FACT (1) = 1

- This is a Recurrence Relation.
- Typifies recursion

4

Recursion Characteristics

1) A general case expressed in terms of a simpler version of itself.

2) A stopping case - trivial, non recursive

3) Application of the general case leading to a stopping case. e.g. 5! = 6!/6
   - a numerically correct result
   - Leads away from the stopping case.

5

You **MUST** be sure
that Recursive Code
will **STOP!!**

"Stack Overflow"
"Heap Overflow"
"Timeout"

6

Concept of Recursion: Factorial Example

```
public int fact ( int n) {
   if n==1 then return 1
   else return n*Fact( n-1 );
}  //end Fact  -  needs error checking
```

FACT (n) = n * FACT (n-1)

FACT (1) = 1

7

Towers of Hanoi Example

- Standard example
  - Highly recursive
  - 3 poles
  - n disks
  - Move 1 disk at a time
  - Do not put larger disk on smaller one

- **How long to move n disks?**

8

9

Fibonacci Example

Developed to model rabbit populations.

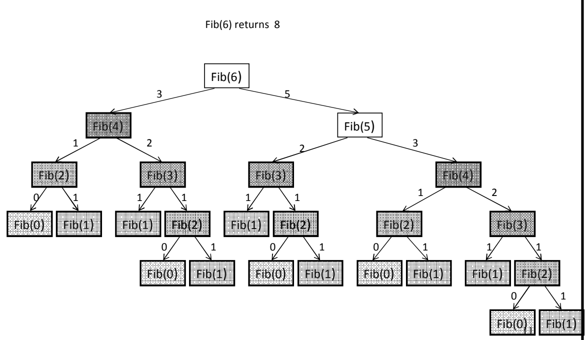
Fib (n) = Fib (n-1) + Fib (n-2)
Fib (0) = 0
Fib (1) = 1

2 stopping cases. Can have more.

Fib(n) = 0 1 1 2 3 5 8 13 21 34 55 …
n = 0 1 2 3 4 5 6 7 8 9 10

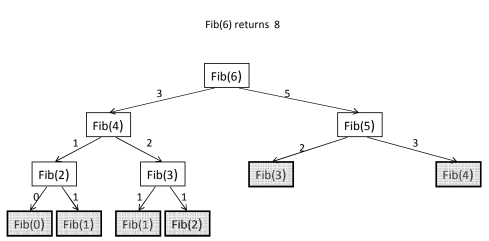
10

Fibonacci Example



11

Fibonacci Example



12

Fibonacci Example

```
int Fib(int n) {
   int x, y, z;
   if x <= 1 return x
   else {
      x = Fib(n-1);
      y = Fib(n-2);
      z = x + y;
      return z;
   }  // end else
}  //end Fib  -  needs error checking
```

13

Fibonacci Example

Alternate:

```
int Fib2(int n) {
   if x <= 1 return x
   else return Fib(n-1) + Fib(n-2);
}  //end Fib2  -  needs error checking
```

14

Fibonacci Example

- Recurrence relations are <u>easily</u> written recursively.
- Note: Fibonacci calls itself twice.
- No limit to the number of locations at which a recursive function can call itself.

eg. ABC(n) = ABC(n-1) + ABC(n-2) +
ABC(n-3) + ABC(n-4) +
ABC(n-5) + ABC(n-6)

15

Efficiency of Recursion

- Recursion has a bad rep.
- Somewhat deserved
  - Inefficient via redundancy ☹
    - e.g. Fibonacci function
  - Recursion is a resource hog ☹

16

Efficiency of Recursion: Redundancy

Use memo-ization to reduce redundancy.

- Useful with any recursive application. ☺
- Create a table.
- Enter each value calculated.
- Check table before calculating new values
- How should the table be organized?
- How big should table be?
- Extra work to maintain/check

17

Efficiency of Recursion: Resources

Typically, compilers use stacks to support recursion

Each instantiation is saved on the stack
Unnecesary things may be saved
Compiler may not be optimized
e.g. Towers of Hanoi

18

Other Examples

Other things besides recurrence relations are recursive

- POW (x, N) example
- Binary Search example.
- Max Array Problem
- K^{th} Smallest Value problem

19

Pow(x,N) Example

```
int Pow(int x, int N) {
   if (N==O) return 1;
   else return x*Pow(x,N-1);
}
```

$x^n = x * x^{n-1}$

O(n)

20

Pow(x,N) Example

```
int Pow2(int X, int N) {
   if (N==0) return 1;
   else {
      int HalfPower = Pow2(x,N/2);
      if (N%2==0) return HalfPower * HalfPower
      else     //N odd
        return x*HalfPower*HalfPower
   } // end else
} // end Pow2
```

O (?)

21

Binary Search

NONMATHEMATICAL RECURSION

e.g. BINARY SEARCH

	A
[1]	1
[2]	3
[3]	4
[4]	5
[5]	10
[6]	17
[7]	18
[8]	25
[9]	31
[10]	33
[11]	35

SEQUENTIAL

```
FOR i := 1 TO 11 DO
  IF A[i] = 31 THEN
    WRITELN('31 is',
     'element #', i);
```

BINARY

USES ORDER OF DATA
TO DETERMINE
SEARCH RANGE

O (?)

Binary Search

Assumes data is sorted

Compare middle element with search item
if match then stop
else if greater than search element
    then apply binary search to first half
else apply binary search to second half

Binary Search

BINARY SEARCH ALGORITHM
IN PSEUDOCODE

```
1  low = lowest array index
2  high = highest array index
3  IF low > high THEN
4    binsrch := 0
5  ELSE
6    BEGIN
7      mid := (low+high) % 2
8      IF x = a[mid] THEN
9        binsrch := mid
10     ELSE
11       IF x < a[mid] THEN
12         search for x between
            a[low] and a[mid-1]
13       ELSE
14         search for x between
            a[mid+1] and a[high]
```

FIND x = 25

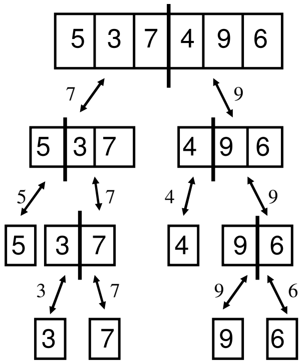
	a
[1]	1
[2]	3
[3]	4
[4]	5
[5]	10
[6]	17
[7]	18
[8]	25
[9]	31
[10]	33
[11]	35

line	mid	low	high
7	6	1	11
14		7	11
7	9	7	11
12		7	8
7	7		
14		8	8
7	8		
9	binsrch = 8		

RETURNS INDEX OF MATCHING VALUE

O (?)

Max Array Problem



ANS: 9

25

Kth Smallest Value problem

4	7	3	6	8	1	9	2
0	1	2	3	4	5	6	7

Suppose K=3?

26

Kth Smallest Value problem

1)Select Pivot

2)Partition Array

3)Recursively apply process to one partition

Partition	Pivot	Partition
<P	P	>P

O (?)

27

K^{th} Smallest Value problem

There is always a pivot. Since the pivot is not part of S1 or S2, the size of the array segment being searched decreases by at least one at each step. Thus you will reach the base case, eventually. The desired element is a pivot. Here is a pseudocode solution.

```
KSmallest(k,S,A,Z)     //Returns kth smallest value in S[A..Z]
        Choose a pivot p from S[A..Z]
        Partition the elements of S[A..Z] about pivot p

        if (k < Index – A + 1) return KSmallest(k,S,A, Index-1)
        else if (k== Index –A+1) return p
        else  return KSmallest(k-(Index-A+1), S, Index+1, Z)
```

28

Motivation

Problem is naturally recursive

- Simple, elegant, brief solution
- Best use of programmer effort.
- More intuitive
- Possible "proof of correctness" available.

29

Motivation

What if can't use recursion?

- Proof of correctness
- Learn more about problem
  e.g. Rapid prototyping
- Source of iterative solution
  Horowitz & Sahni, Fundamentals of Computer Algorithms

30

Supporting Recursion

- Consider an imaginary function

```
int ABC(char a, int b, float c) {
    int x,  y;
    float z;
    x =  . . .
    y = . . . ABC(. . .);

    z =            . . .  y . . .
}
```

31

Supporting Recursion

- Consider an imaginary function

ANS = ABC(A, 2, 3)

Get Addr for ABC
Set up parameters
  By value
  Allocate space -
    store value
Allocate space for
  local variables

```
int ABC(char a, int b, float c) {
    int x,  y;
    float z;
    x =  . . .
    y = . . . ABC(. . .);

    z =            . . .  y . . .
}
```

Save Return Address
PASS CONTROL to ABC

32

Supporting Recursion

- Consider an imaginary function

Return control

```
int ABC(char a, int b, float c) {
    int x,  y;
    float z;
    x =  . . .
    y = . . . ABC(. . .);

    z =            . . .  y . . .
}
```

Return Value under function name

Deallocate space for params by value
Deallocate space for local variables

Retrieve Return Address

33

Supporting Recursion

- Consider an imaginary function

ANS = ABC(A, 2, 3)

Get Addr for ABC
Set up parameters
By value
Allocate space - store value
Allocate space for local variables

Save current state
a, b, c, x, y, z, etc

```
int ABC(char a, int b, float c) {
      int x,  y;
      float z;
      x =  . . .
      y = . . . ABC(. . .);

      z =            . . .  y . . .
}
```

Save Return Address

PASS CONTROL to ABC

34

Supporting Recursion

- Consider an imaginary function

```
int ABC(char a, int b, float c) {
      int x,  y;
      float z;
      x =  . . .
      y = . . . ABC(. . .);

      z =            . . .  y . . .
}
```

Return control

Return Value under function name

Deallocate space for params by value

Deallocate space for local variables

Restore prior state
a, b, c, x, y, z, etc

Return Value under function name

Retrieve Return Address

35

Simulating Recursion

- Uses stacks (later)
- Eliminate local variables
- Eliminate items not used after Recursive call
- Remember refinement can introduce errors
- Example: Convert recursive Tree Traversal to iterative - (A&T Chapter 6.1)

36

Practical Issues

- Pass by value - not by reference
- Use local variables freely
- Use memo-ization if appropriate
- Use indentation or Boxes for planning
- Assume Recursive Call works

37

Practical Issues

```
(1)          int ABC(char a, int b, float c) {

                 int x,  y;
(2)              float z;
                 x =  . . .

(4)              y = . . . ABC(. . .);


(3)              z =               . . .  y . . .
             }
```

38

```
Below is an interactive session with a recursive language called SML (Standard
Meta Language Of NJ).  THe fibonacci function is defined and then executed
several times.   As you can see, it is a clean and simple piece of code.  THe
^M is a carriage return since this is a script file.
                               B. Chlan

                                The next two lines are standard response to
                                invoking SML
Standard ML of New Jersey, Version 0.93, February 15, 1993
val it = () : unit
                                The next three lines are the actual code.
- fun fib = 1 = 1
=         | fib 2 = 1
=         | fib n = fib(n-1) + fib(n-2);
                                The next line is a response from ML saying
                                it understands fib to be a (mathematical)
                                function mapping integers to integers.
val fib = fn : int -> int
                                The next line is a request to evaluate fib(1)
- fib 1 ;
                                The response - the value is one and is integer.
val it = 1 : int
                                More requests follow.
- fib 2;
val it = 1 : int
- fib 3;
val it = 2 : int
- fib 4;
val it = 3 : int
- fib 5;
val it = 5 : int
- fib 6;
val it = 8 : int
- fib 7;
val it = 13 : int
- fib 8;
val it = 21 : int
- fib 9;
val it = 34 : int
- fib 10;
val it = 55 : int
- fib 25;
val it = 75025 : int
```

Following is a prolog source file of a program to compute the fibonacci sequence. The first two lines are considered facts or true statements. They are followed by a rule in the form of a if-then or implication. (The conclusion is first, the :- is read "if" and the commas between clauses are read as logical "and".).

After the code is a copy of a script file showing the code bing executed. Prolog is an interpreter.

```
*****SOURCE CODE******

fib(0,0).
fib(1,1).
fib(N,X) :- N1 is N - 1, N2 is N - 2,
            fib(N1, X1),
            fib(N2, X2),
            X is X1 + X2.

*****SCRIPT FILE*****

Script started on Thu Mar  7 10:50:39 1996
> prolog
C-Prolog version 1.5
| ?- [fib].
fib consulted 292 bytes 9.93411e-10 sec.

yes
| ?- fib(1,X).

X = 1 ,

yes
| ?- fib(0,X).

X = 0 ,

yes
| ?- fib(2,X).

X = 1 ,

yes
| ?- fib(3,X).

X = 2 ,

yes
| ?- fib(4,X).

X = 3 ,

yes
| ?- fib(5,X).

X = 5 ,

yes
| ?- fib(8,X).

X = 21 ,
```

40