Recursion - Overview

Two forms of repetition in programming:

- iteration
- recursion

Iteration

- Repeated apply the same operations to different elements in a programmed loop

Recursion

- Repeatedly breaking the problem down into a simpler version of itself
- Simpler → Easier to solve
  (eventually problem will become trivial)

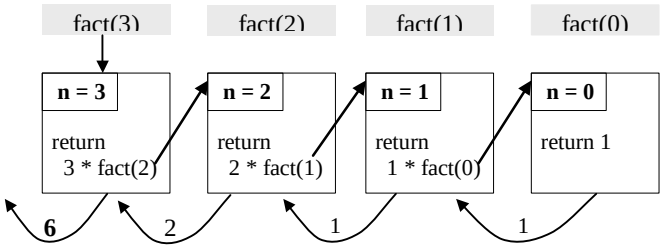
Factorial

fact(n) is the product of n and all lower integers

e.g. fact(5) = 5 * (4 * 3 * 2 * 1)
fact(5) = 5 * fact(4)
fact(5) = 5 * (4 * fact(3))
fact(5) = 5 * (4 * (3 * fact(2)))
…

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

Example – factorial(3)



Recursion

recursion is when functions call themselves

```
#include <stdio.h>

main() {
   printf("This goes on and on\n");
   main();
}
```

Like pointers, this may seem strange at first.

The rules are really very simple.

Recursion allows a program to repeatedly perform an action - ***it is another form of looping.***

Recursion can be used instead of using standard loops (**`for`**, and **`while`**)

Programs that use loops are called **iterative -** they use **iteration.**

- Some problems are solved more easily by iteration
- Some more easily by recursion
- Some problems can be solved by either method.

Sum of the first n integers

```
#include <stdio.h>

int sum_1_to_n(int n);


main() {
   int total;

   total = sum_1_to_n(10);

   printf("total = %d\n",total);

   return 0;
}

int sum_1_to_n(int n) {

   if (n <= 1) {
      return n;
   }
   else {
      return(n + sum_1_to_n(n-1));
   }
}
```

Rules for writing recursive procedures

- Ensure that the recursion will eventually stop.

  There must be a path through the function that doesn't involve a further recursive call

- Every recursive call should create a situation closer to the final solution than the original call.

Variables

- **Local variables** - a new one is created for each recursive call. You needn't worry about name clashes.
- **Formal parameters** – each level has its own copy

- If you need data accessible by all recursive calls:
  - pass it as an additional parameter

or

- store it in a global variable

Recursion - Some examples

Palindrome checker

Check the first and the last characters
different → NOT a palindrome

same → check rest of palindrome

If (first > Last) then all characters have matched
→ It's a palindrome

```
int palindrome(
   char s[],      // The string to check
   int first,     // which left hand char
   int last       // right hand character
   ) {


   if (first > last) return 1;

   if (char[first] != char[last]) return 0;

   return palindrome(first+1, last-1);
}
```


Finding the maximum in an array

- assume the first element is the maximum
- the find the maximum of the rest of the array
- if there aren't more elements, return current maximum

```
int A [10];
max = findmax(A[0], 1, 10);

int findmax(int MaxSoFar, /* current max    */
            int this, /* start pos in array */
            int size  /* Size of the array  */
{
  if (this == size) return MaxSoFar;

  if (A[this] > MaxSoFar)
     return findmax( A[this],  this+1, size);
  else
     return findmax( MaxSoFar, this+1 , size);
}
```


Recursion - some pitfalls

- Each recursive call needs memory to enable it to remember where it has come from.
- Deeply nested recursion may require lots of memory
- Recursive routines that are badly written can be very inefficient.

The Fibonacci sequence - Recursively

```
int fib(int n) {
  if ((n==0) || (n==1)) {
    return 1;
  }
  else {
    return fib(n-1) + fib(n-2);
  }
}
```

fib(6) calls **fib(5)** and **fib(4)**

fib(5) also calls **fib(4)** and **fib(3)**

fib(4) is called twice.

fib(3) will be called three times.

Saving Intermediate Results

Sometimes it is better to store away a number in a table once it has been calculated, rather than calculating it all over again.

The table has to be global.

```
int table[100];

main() {

   table[0] = 1;
   table[1] = 1;

   for (i=2;i<100;i++) {
      table[i] = 0;
   }

   printf("6'th fib' number is %d\n",fib(6));
}


int fib(int n) {
   if (table[n] != 0) {
      return table[n];
   }
   else {
      table[n] = fib(n-1) + fib(n-2);
      return table[n];
   }
}
```


Direct and Indirect Recursion

A function calling itself is **direct** recursion

eg fib(6) calls fib to calculate fib(5) and fib(4) …

A function calling itself via other functions is **indirect** recursion

Function A calls B and B calls A

OR

Function A calls B which calls C which calls D and D calls A

Recursive functions are used a lot with certain data structures (lists, stacks, queues, and binary trees) that will be introduced in the later courses.

Searching and Recursion

- To find the position of an item in an array, you can simply look at each element in turn.
- This is a "linear" search
- On average you look thru half the items before finding the wanted item

```
int table [500];

/* Find searches table and returns the
   location of a particular value or -1*/

int Find (int value; int table[]) {
int i=0;

  while (i<50)
     if (table[i] == value) return(i);
     else i++;
}

where = Find(47, table);
```

Doing something using a loop is called *iteration*

Linear Search Using Recursion

Rather than use looping, it is possible to reduce the problem to a smaller version of itself:

Find (item, table, Start)
if Start isn't a valid index → *item isn't here*
if Table[start] is wanted value → return *Start*
Search rest of table→ *return result from rest of table*

```
int Find(int value, start; int table[]){
  int where;
  if (start == 50)          return( -1)
  else
  if (table[start]==value) return(start)
  else{
    where= Find(value, start+1, table);
    return(where);
  }
```

Notice there is no Explicit Looping Constructs

```
   where = Find(47, 0, table);
```

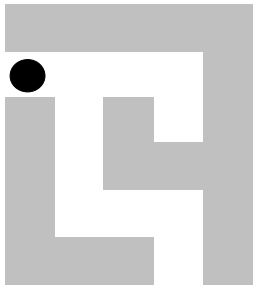
if table[0] doesn't contain 47, the next call is:

```
   where = Find(47, 1, table)
   where = Find(47, 2, table) …
```

The amount to be done (searched) gets smaller each time.

Maze Solving

Consider finding a path thru a maze



At any time:

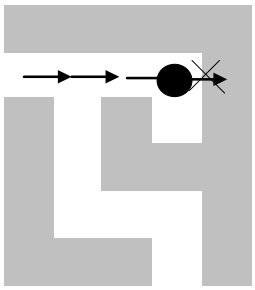
- we have a current position
- we have several possible directions
  (forward, back, left, right)
- we may have already tried one or more of these

  1. don't want to retry paths that have already failed
  2. this implies a record of where we have been

Finding a Path through a Maze

1. From our current position:
2. pick one possible direction
3. move there
4. go to 1

This works as long as we don't pick a wrong path

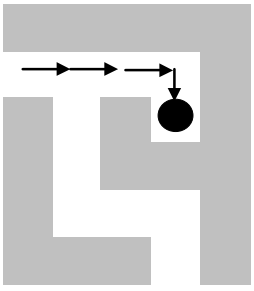


From the entrance on the left:
what do we do when hit the wall straight ahead
try another direction
pick a direction that:
- we havn't been in before
- that isn't blocked

for current position
UP doesn't work
LEFT is where we came from
DOWN is OK

Retracing the path

It's possible to get trapped



- need to retrace one's steps back to a fork in the path
- try an alternative route

To remember where we've been:

construct a list of visited locations
or
use recursive algorithm

How we got to the current locations is stored *as the parameters* to each call to SOLVE

- we don't need to explicitly store our path
- it's saved automatically in each procedure invocation

Recursive Maze Solving Algorithm

Recursive algorithms work by reducing the problem to a smaller one every time the function is applied

From one location, try each of four adjacent locations:

- if the new location is the exit we've finished
- if the new location is beyond the maze, ignore it
- move in to this location and try all its neighbours

Programming a Function

There will be a function called SOLVE that:

- has as a parameter a location
- returns **True** if there is a route from that location to the exit

Representing the Maze

Can use a character array:

```
char maze [9] [9] = {"        ",
                     "X   XXXX",
                     "  X   XX",
                     "X  XX XX",
                     "XX  X  X",
                     "   X X  ",
                     " XXX XX ",
                     "     XX "};
```

Characters on the Maze

‘X’ is a wall
‘ ‘ is an opening

Remembering where we’ve been

‘x’ is a wall that we’ve already bumped into
‘**.**’ is an opening we’ve tried

Locations

These can be represented by a row and column coordinate

Need to know

- start location
- end location
- current position

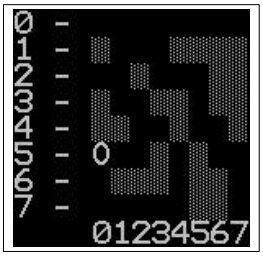
Print_ch – a Pretty Printing Routine

The PC can display graphic character such as solid blocks if given the right extended character code

This enables the maze to be printed with solid walls, instead of letters:

```
X   XXXX
  X   XX
X  XX XX
XX  X  X
   X X
 XXX XX
     XX
```

is printed as (see next slide)



if character is x or X print a BLOCK
else print character supplied

```
void print_ch (char ch) {
   switch (ch) {
     case 'x' :
       printf("%c", 177); break; //med block
     case 'X' :
       printf("%c", 176); break; //light block
     default  : printf("%c", ch);
   }
}
```


Displaying the Maze

Display an **O** for the current position, else print the supplied character

```
void printmaze(int row_position,
               int col_position) // our position
 {
  int r, c, ch;
    for (r=0; r<size-1; r++) { // Print row #
      printf("%d - ", r);
      // Print content of maze
      for (c=0; c<size-1; c++) {
       if (r==row_position && c==col_position)
            if (maze[r][c] == ' ')
                 print_ch('O');
            else print_ch(178);
           else print_ch(maze[r][c]);
      }
      printf("\n");
    }
    printf("    ");   //Print Column numbers
    for (ch='0', c=0; c<size-1; c++)
      printf("%c", ch++);
    printf("\n");
  }
```


Next possible move

If our current position is represented by the coordinates in (row, column)

From our current position, try

```
solve(row+1, column  )    → DOWN
solve(row  , column-1)    → LEFT
solve(row  , column+1)    → RIGHT
solve(row-1, column  )    → UP
```

Otherwise return FALSE – no route

Maze Solving Function

Function is called with the next position as the parameters

```
int solve (int row, int col) {
int found;

  // Beyond the Maze?
  if (row<0 || row==size-1  ||
      col<0 || col==size-1)     return 0;

  // Already tried this square?
  if (maze[row][col] == 'x'||
      maze[row][col]=='.')      return 0;

  printmaze(row, col);  // Show where we are


  if (maze[row][col] == 'X') { // Hit Wall
    maze[row][col]= 'x';        return 0;


  // Not Wall, and new untried square
  maze[row][col] = '.';    // Mark as visited


  // Found the Exit?
  if (row == endrow && col == endcol) {
   printf("**** Finished *****\n");
   printf("Path: %d,%d", row, col);
   return 1;
  }
```



```
  // We're on a square that ISN'T a wall
  // Is there a route from a neighbour

       if (solve(row+1, col  )) found = 1;
  else if (solve(row  , col-1)) found = 1;
  else if (solve(row  , col+1)) found = 1;
  else if (solve(row-1, col  )) found = 1;
  else found = 0;


  // Tried all the neighbours, did we succeed

  if (found) {
    maze[row][col]='p';
    printf(" <-%d,%d", row, col);
  }

  return found;
}
```

Maze Solver – Mainline

```
int main  () {
int ok;
  ok= solve(5,0);
  if (!ok) printf("Couldn't find a path\n");
  else {
    printf("\nFound Path:\n");
    printmaze(endrow, endcol);
  }
  printf("Solve was called %d times.\n",
        recursion_count);
}
```


Towers of Hanoi

To move a pile of n counters from peg a to peg c

move n-1 counters from a to b
move 1 counter from a to c
move n-1 counters from b to c

```
#include <stdio.h>

void move_n_a_to_c(int n);
void move_n_a_to_b(int n);
void move_n_b_to_c(int n);
void move_n_b_to_a(int n);
void move_n_c_to_a(int n);
void move_n_c_to_b(int n);

main() {
   move_n_a_to_c(10);
}

void move_n_a_to_c(int n) {
   if (n == 1) {
      printf("Move counter from a to c);
   }
   else {
      move_n_a_to_b(n-1);
      move_n_a_to_c(1);
      move_n_b_to_c(n-1);
   }
}
```

... similarly for the other 5 functions.

Rules for writing recursive procedures

- Ensure that the recursion will eventually stop. There must be a path that does not involve a further recursive call
- Every recursive call should involve a situation that is closer to solution than the original call.
- Local variables - a new one is created for each recursive call. You do not need to worry about name clashes.
  - If you need data that is accessible by all recursive calls, store it in a global variable.