CSE 142 Programming I

Recursion

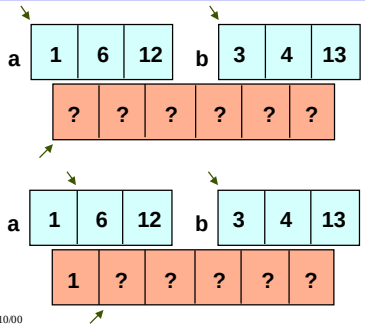
Chapter 10: Recursion

- 10.1 Nature of Recursion (skip "Mississippi" example 10.2)
- 10.2 Tracing: *Read!*
- 10.3 Recursive Math (but skip gcd example 10.6)
- 10.4 Skip
- 10.5 Skip
- 10.6 Towers of Hanoi: A Classic, but optional
- 10.7 Common Errors: *Read!*

Mergesort

- **Review**
  - **As we learned a few days ago:**
  - Start with some small sorted pieces: "runs"
  - Merge pairs of runs together to make larger sorted runs
  - When we finish merging the final pair, then we have sorted our array.
- **Basic operation is the merge.**
- **A merge works ONLY if both parts are ALREADY sorted. Keep this in mind!**

Subproblem: Merge

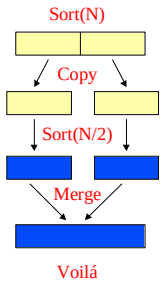


Turning a Merge into A Sort

Given a large, unsorted array

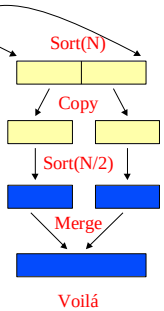
- **break array into two (unsorted) pieces**
- **sort each piece**
- **now merge the two pieces**
- **voilá**

What could be easier?



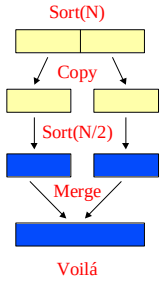
As Pseudo-C Code

```
void MergeSort(Array A, int l, int u)
{
   int   middle = (l+u)/2;
   Array lowerTemp;
   Array upperTemp;
   copy half arrays;
   MergeSort(lowerTemp, l, middle);
   MergeSort(upperTemp, middle+1, u);
   merge lowerTemp and upperTemp
         into A;
}
```



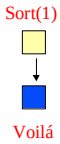
Recursion

- **This is *recursion***
  - Expressing the solution to the problem in terms of
    - The same solution of
    - One or more smaller problems
- **One problem:**
  - If solving every problem of every size required solving another problem by the same method, you'd "infinitely recurse"
  - So, you need a (trivial) *base case*



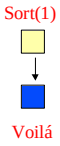
The Base Case

- **For merge sort, the base case is that the array to be sorted has only a single element in it**
- **In this case, it's already sorted, and we're done**



The Base Case

```
void MergeSort(Array A, int l, int u)
{
   int   middle = (l+u)/2;
   Array lowerTemp;
   Array upperTemp;
   if (l>=u) return;
   copy half arrays;
   MergeSort(lowerTemp, l, middle);
   MergeSort(upperTemp, middle+1, u);
   merge lowerTemp and upperTemp
        into A;
}
```



Actual C Code (typedefs)

```
#define   MAXSTRINGLENGTH   80
#define   MAXNUMLINES       700

typedef   char     String[MAXSTRINGLENGTH+1];
typedef   String   ArrayOfStrings[MAXNUMLINES];
```


Actual C Code

```
void MergeSort (ArrayOfStrings targetArray, int lowerFence, int upperFence)
{
        int                 middle, index, lowerIndex, upperIndex, compareResult;
        ArrayOfStrings      lowerArrayTemp, upperArrayTemp;

        /* base case - a single element is sorted */
        if (lowerFence >= upperFence) return;

        /* otherwise…                                   */
        /* (A) Divide the unsorted portion into half    */

        middle = (lowerFence + upperFence) / 2;
        CopyArray(targetArray, lowerArrayTemp, lowerFence, middle);
        CopyArray(targetArray, upperArrayTemp, middle+1, upperFence);

        /* (B) Sort each half */

        MergeSort(lowerArrayTemp, lowerFence, middle);
        MergeSort(upperArrayTemp, middle+1, upperFence);

        /* (C) Now merge the two halves */

        (… Full source available on web)
```


What is Recursion?

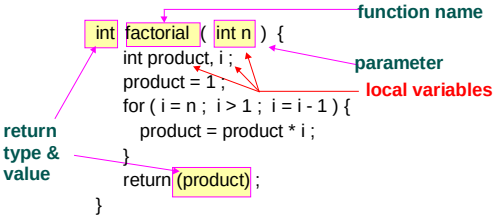
- **Defn: A function is recursive if it calls itself**

```
int foo(int x) {
  …
  y = foo(…);
  …
}
```

- **Questions:**
  - **How can recursion possibly work?**
    - **This we can explain**
  - **Why would I want to write a recursive function?**
    - **This we will try to motivate**

Factorial Function Revisited

0! is 1
1! is 1
2! is 1 * 2
3! is 1 * 2 * 3
. . .



Factorial via Recursion

Factorial's definition is inherently recursive:
0! = 1! = 1; for n > 1, n! = n(n-1)!

0! is 1
1! is 1
n! is n * (n-1)!, for n>1

E.g.: 3! =3 * 2!
= 3 * 2 * 1!
= 3 * 2 * 1

```
int factorial(int n)
{
   int t;

   if (n <= 1)
      t = 1;
   else
      t = n * factorial(n - 1);

   return t;}
```

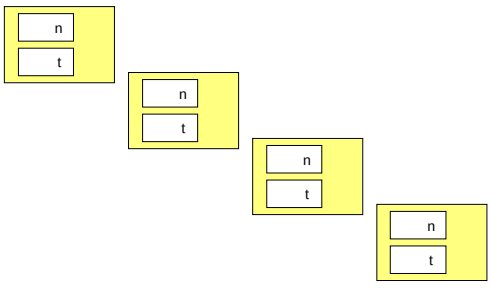

Review: Function Basics

- ***Tracing recursive functions is no sweat if you remember the basics about functions:***
  - Parameters and variables declared in a function are **local** to it
    - **Allocated** (created) on function entry.
    - **De-allocated** (destroyed) on function return.
  - Parameters are initialized by **copying values** of arguments when a function is called.

Factorial

factorial(4) =
4 * factorial(3) =
4 * 3 * factorial(2) =
4 * 3 * 2 * factorial(1) =
4 * 3 * 2 * 1 = 24

Factorial Trace



Insist on 'y' or 'n'

```
char yes_or_no (void) {
  char answer = 'X';
  while (answer != 'y' && answer != 'n') {
     printf ("Please enter 'y' or 'n':");
     scanf (" %c", &answer);
     }
  return answer;
}
```

Insisting without Looping

```
char yes_or_no (void) {
  char answer;
  printf ("Please enter 'y' or 'n':");
  scanf (" %c", &answer);
  if  (answer != 'y' && answer != 'n')
      answer = yes_or_no ( );
  return answer;
}
```

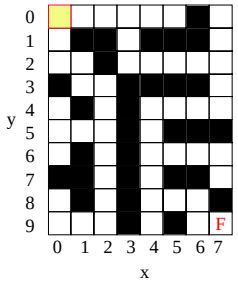

Iteration *vs.* Recursion

- Turns out *any* iterative algorithm can be reworked to use recursion instead (and vice versa).
- There are programming languages where recursion is the only choice(!)
- Some algorithms are more naturally written with recursion
  - But *naïve* applications of recursion can be inefficient

When to use Recursion?

- Problem has one or more simple cases
  - These have a straightforward nonrecursive solution, and:
- Other cases can be redefined in terms of problems that are closer to simple cases
  - By repeating this redefinition process one gets to one of the simple cases

Example: Path planning



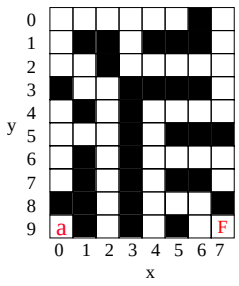

```
/* 'F' means finished!
   'X' means blocked
   ' ' means ok to move */
char maze[MAXX][MAXY];
int x =0, y=0;   /* start in yellow */
```

Unless blocked, can move up, down, left, right

Objective: determine if there is a path?

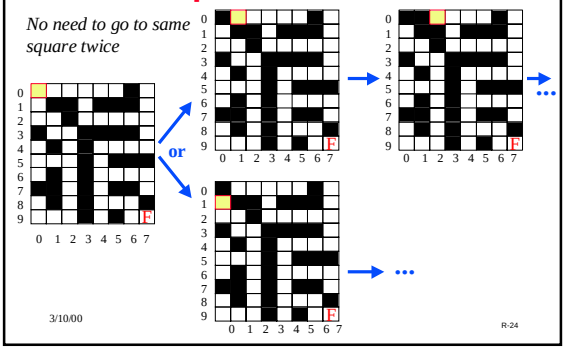
Simple Cases

- Suppose at x,y
- If maze[x][y]=='F'
  - Then "yes!"
- If no place to go
  - Then "no!"



Redefining a hard problem to several simpler ones

No need to go to same square twice



Helper function

```
/* Returns true if <x,y> is a legal move
   given the maze, otherwise returns false */
int legal_mv (char m[MAXX ][MAXY],
                                  int x, int y) {
  return(x>=0 && x<=MAXX &&
         y>=0 && y<= MAXY &&
          m [x][y] != 'X');
}
```

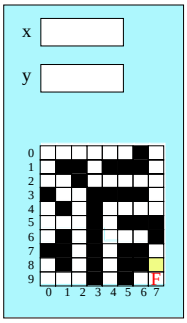

Elegant Solution

```
/* Returns true if there is a path from <x,y> to an element of maze
   containing 'F' otherwise returns false */
int is_path(char m[MAXX][MAXY ], int x, int y) {
  if (m [x][y] == 'F')
    return(TRUE);
  else {
    m[x][y] = 'X';
    return((legal_mv(m,x+1,y) && is_path(m,x+1,y)) ||
           (legal_mv(m,x-1,y) && is_path(m,x-1,y)) ||
           (legal_mv(m,x,y-1) && is_path(m,x,y-1)) ||
           (legal_mv(m,x,y+1) && is_path(m,x,y+1)))
}}
```


Example

is_path(maze, 7, 8)

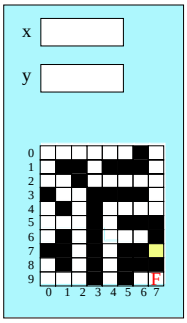
```
int is_path(char m[MAXX][MAXY ], int x, int y) {
  if (m [x][y] == 'F')
    return(TRUE);
  else {
    m[x][y] = 'X';
    return((legal_mv(m,x+1,y) && is_path(m,x+1,y)) ||
           (legal_mv(m,x-1,y) && is_path(m,x-1,y)) ||
           (legal_mv(m,x,y-1) && is_path(m,x,y-1)) ||
           (legal_mv(m,x,y+1) && is_path(m,x,y+1)))
```



Example Cont

is_path(maze, 7, 7)

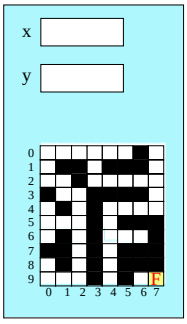
```
int is_path(char m[MAXX][MAXY ], int x, int y) {
  if (m [x][y] == 'F')
    return(TRUE);
  else {
    m[x][y] = 'X';
    return((legal_mv(m,x+1,y) && is_path(m,x+1,y)) ||
           (legal_mv(m,x-1,y) && is_path(m,x-1,y)) ||
           (legal_mv(m,x,y-1) && is_path(m,x,y-1)) ||
           (legal_mv(m,x,y+1) && is_path(m,x,y+1)))
```



Example Cont

is_path(maze, 7, 9)

```
int is_path(char m[MAXX][MAXY ], int x, int y) {
  if (m [x][y] == 'F')
    return(TRUE);
  else {
    m[x][y] = 'X';
    return((legal_mv(m,x+1,y) && is_path(m,x+1,y)) ||
           (legal_mv(m,x-1,y) && is_path(m,x-1,y)) ||
           (legal_mv(m,x,y-1) && is_path(m,x,y-1)) ||
           (legal_mv(m,x,y+1) && is_path(m,x,y+1)))
```



Recursion Wrapup

- Recursion is a programming technique
  - It works because of the way function calls and local variables work in C
  - New copy of everything whenever a function is called
- Recursion is more than a programming technique. It's also a way of thinking about problem solutions.
- It takes practice to get fluent at recursive thinking
- It may seem unnatural now, but... in CSE143, we will see data structures for which recursion is the natural approach, and not using recursion is awkward.