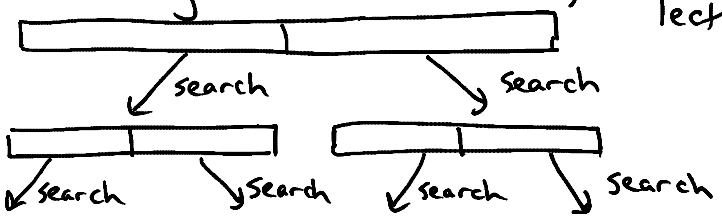
14.1 Recursive Functions for Tasks

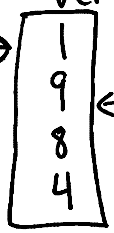
recursion - a function that calls itself

Ex: binary search (p 785, cover next lecture)



Case Study - Vertical Numbers (p 765)

1984 →



Write recursive function to display this

Think about how to break down problem

if number is one digit (< 10)

just print to screen

else

1) print 1 to n-1 digits to screen

2) print n^{th} digit to screen

(1) lends to recursive function

call function w/ n^{th} digit removed

number /10 removes last digit

when using integers

(2) n^{th} digit is number % 10

This leads to pseudocode of alg

```
if number < 10    //stopping condition
    print number
else
    write_vertical (number /10)
    print number %4
```

This has good recursive structure

one branch has no recursive call

stopping condition to prevent

endless function calls

one branch has recursive call that breaks task into subset

Code

```
void write_vertical(int n);
void write_vertical (int n)
{
    if (n < 10)
    {
        printf ("%d \n", n);
    }
    else
    {
        write_vertical (n /10);
        printf ("%d \n", n %10);
    }
}
```

Tracing a Function Call

Book uses write-vertical (123) on p767

Use 1984 in class

```
write-vertical (1984)
  if (1984 <10)    F
  else
    write_vertical (1984 /10)
      if ( 198 < 10)  F
      else
        write_vertical (198 /10)
          if (19 < 10)  F
          else
            write_vertical (19 /10)
              if (1 < 10)  T
                  output 1,
              output 19%10,
          output 198%10,
      output 1984%10,
```

1
9
8
4

on screen

output 198 % 10, 9
output 1984 % 10, 4

How It Works
- When any function call reached, system calls function & waits for results before continuing to next line
- Saves all info needed to continue when called function finishes
- So when write-vertical (1984) calls write-vertical (198), it must wait for that to finish completely before continuing to the printf line

Stopping Recursion
- The recursive calls have to stop at some point to allow all to finish
- Have "base case" or "stopping case" does not have a recursive call
- Every possible input must eventually decompose to stopping case or some calls will never complete

Infinite Recursion - Pitfall
- stopping case not reached or no stopping case at all
- will typically run until resources exhausted or computer terminates program / crashes
- if observed in code, check that computations correctly lead to stopping case

Stacks for Recursion
- a special memory structure used for function calls, organized LIFO (last in, first out)
- only data at top of stack is readable because it is LIFO
- push - put data on stack
- pop - take top data off stack

how it is used

before a function call is executed,
certain information about currently executing code is saved to the stack
- anything needed to continue execution after function call exits

as function is executing, if it has a function call it also pushes data onto stack

when function call completes, results noted & saved info about caller is removed from stack

caller continues execution

activation frame - portion of memory that contains info about caller

Stack Overflow

too many activation frames pushed
exceeds allocated max mem of stack
Ex: infinite recursion causes stack overflow

Recursion vs Iteration

Iterative version of write_vertical (p777)

```
Void write_vertical (int n)
{
    int tens = 1;
    int left_end = n;
    while (left_end > 10)
    {
        left_end = left_end /10;
        tens = tens * 10;  //find highest multiplier
    }
    //print from highest multiplier (1st digit)
    // to last digit
    for (int mod = tens; mod > 0; mod = mod/10)
    {
        printf("%od\n", n /mod);
```

```
        n = n % mod;
    }
}
```

Iterative version not always as easy to come up with as recursive version

Recursive is not as efficient due to having to save context on stack
- often takes more memory because of stack
- doing a function call always has costs

Weigh benefits of simpler alg vs costs of stack

14.2 Recursive Functions for Values

Similar to void functions shown in 14.1 but returns a value

Many cases for using recursion to calculate
factorial(n) = n * factorial(n-1)
power(n, x) = n * power(n, x-1)

Base case should return base value
factorial(n) returns 1 when n == 1
power(n, x) returns 1 when x == 0

Code

```
int factorial(int n)
{
    if (n <= 1) return 1;
    else return n * factorial(n-1);
}
// Power function from p 780
int power(int base, int pow)
{
    if (pow < 0)
    {
        printf("Illegal power %d\n", pow);
        exit(1);
    }
    if (pow > 0) return power(base, pow-1);
```

```
    }
    if (pow > 0) return power(base, pow-1);
    else return 1;
}
```

Tracing a factorial call

```
factorial(4)
    if(4 <= 1) F
    else return 4 * factorial(3)
        if (3 <= 1)  F
        else return 3 * factorial(2)
            if (2 <= 1) F
            else return 2 * factorial(1)
                if (1 <= 1) T
                    return 1
            2 * 1 = 2
        3 * 2 = 6
    4 * 6 = 24
```

14.3 Thinking Recursively

Design Techniques

Check that chain of calls reaches stopping case & stopping case is correct

Don't need to trace all calls like above

Instead, check following:

1) No infinite recursion
2) Stopping case(s) are correct
3) Final returned value is correct
   (or action is correct for void funcs)

Case Study - Binary Search

Efficient way to search sorted array

Array must be sorted for this to work

This code will deviate from book's example

Book: two call-by-ref params
- one bool - true if value found
- one int - index of element if found

one int - index of element if found

Lecture: function will return int

if int >= 0, index of found element

if int == -1, element not found

Pseudocode

if no elements, return NOT_FOUND

compute midpoint between start & end

if midpoint is element, return index

if element < midpoint

search start to midpoint - 1

else element > midpoint

search midpoint + 1 to end

Code

```
int binary_search (int a[],
    int start, int end, int elem)
{
    int mid;
    if (start > end) return -1;
    else
    {
        mid = (start + end)/2;
        if (elem == a[mid])
          return mid;
        else if (elem < a[mid])
          return binary_search(a,
              start, mid-1, elem);
        else
          return binary_search (a,
              mid+1, end, elem);
    }
}
```

Checking the Recursion

1) No infinite recursion

returns -1 when no more elements

always reducing num of elements

2) Stopping case correct

2 stopping cases: not found & found
not found - no elements so nothing could be found, correct
found - returns index where found

3) Final value is correct
-1 when not found
index when found

Programming Example - Recursive Member Func

member functions can use recursion too
Coded the same way as stand-alone funcs
Example is to add X years of interest to the balance of a bank account

Pseudocode

```
if years == 1
   add 1 yr of interest
if years > 1
   add 1 yr of interest
   call function w/ year - 1
```

Code

```
void BankAccount :: update ()
{
   balance += fraction (interest_rate) * balance;
}
void BankAccount :: update (int years)
{
    if (years == 1)  update ();
    else if (years > 1)
    {
       update ();
       update (years - 1);
    }
}
```