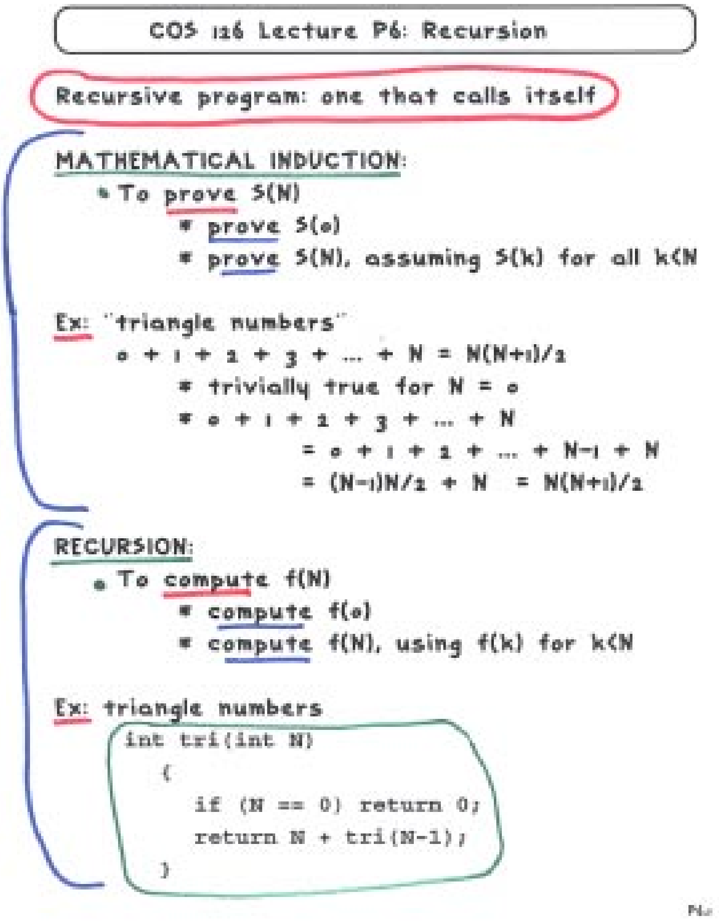
# CS 126 Lecture P6: Recursion

## Why Learn Recursion?

- Master a powerful programming tool
- Gain insight of how programs (function calls) work

# Outline

- **What is recursion?**
- How does it work?
- Examples



## COS 126 Lecture P6: Recursion

Recursive program: one that calls itself

MATHEMATICAL INDUCTION:

- To prove S(N)
  - prove S(0)
  - prove S(N), assuming S(k) for all k<N

Ex: "triangle numbers"

$0 + 1 + 2 + 3 + \ldots + N = N(N+1)/2$

- trivially true for $N = 0$
- $0 + 1 + 2 + 3 + \ldots + N$
  $= 0 + 1 + 2 + \ldots + N-1 + N$
  $= (N-1)N/2 + N = N(N+1)/2$

RECURSION:

- To compute f(N)
  - compute f(0)
  - compute f(N), using f(k) for k<N

Ex: triangle numbers

```
int tri(int N)
  {
    if (N == 0) return 0;
    return N + tri(N-1);
  }
```

P6.2

## Number conversion

- To convert an integer N to binary:
  - stop if N is 0
  - write "1" if N odd, "0" if N even
  - move left one position
  - convert N/2

Ex:

| | |
|---|---|
| 42 | 0 |
| 21 | 10 |
| 10 | 010 |
| 5 | 1010 |
| 2 | 01010 |
| 1 | 101010 |

check:

| | | | | | |
|---|---|---|---|---|---|
| 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 |

$$2^5 + 2^3 + 2^1$$

$$32 + 8 + 2 = 42$$

- Easiest way to convert to binary by hand
- Corresponds directly to a recursive program

## Recursive number conversion

- Computer prints from left to right
  - need to convert N/2, then print right bit

```
void convert(int N)
  {
    if (N/2 > 0) convert(N/2);
    printf("%c", '0'+ N % 2);
  }
```

Proof of correctness: N = 2*(N / 2) + (N % 2)

Indentation level denotes statements belonging to same "invocation"

```
convert(42)
  convert(21)
    convert(10)
      convert(5)
        convert(2)
          convert(1)
            printf("1")        1
          printf("0")          0
        printf("1")            1
      printf("0")              0
    printf("1")                1
  printf("0")                  0
```

↳ 101010 = 42

- Works to convert to any base
  (change "2" to "b" everywhere in code)

# Demo `convert()`

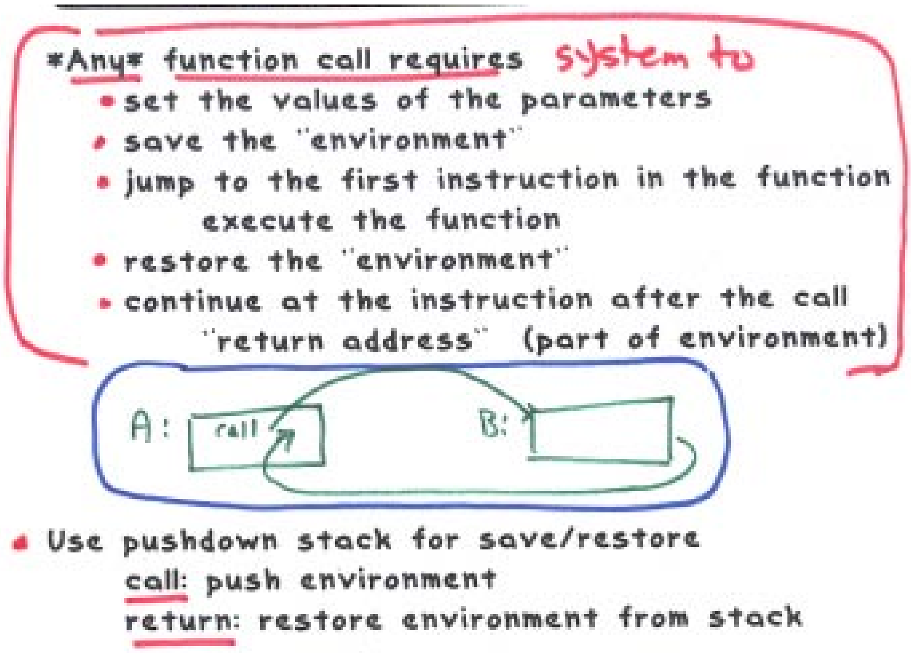
# Outline

- ~~What is recursion?~~
- **How does it work?**
- Examples

# Function "Environment"

- When a function executes, it lives in an "environment"
- What's an "environment"?
  - Value of local variables (scratch space)
  - Which statement the computer is executing currently

# Implementing Recursion

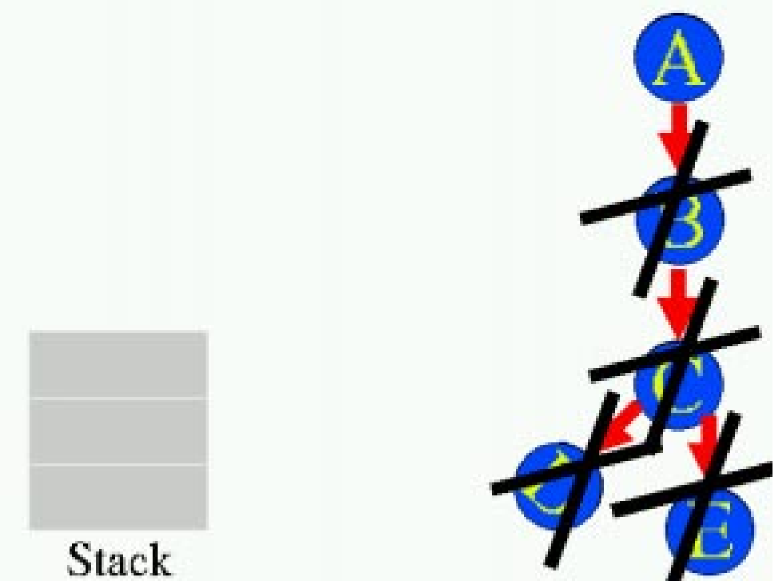


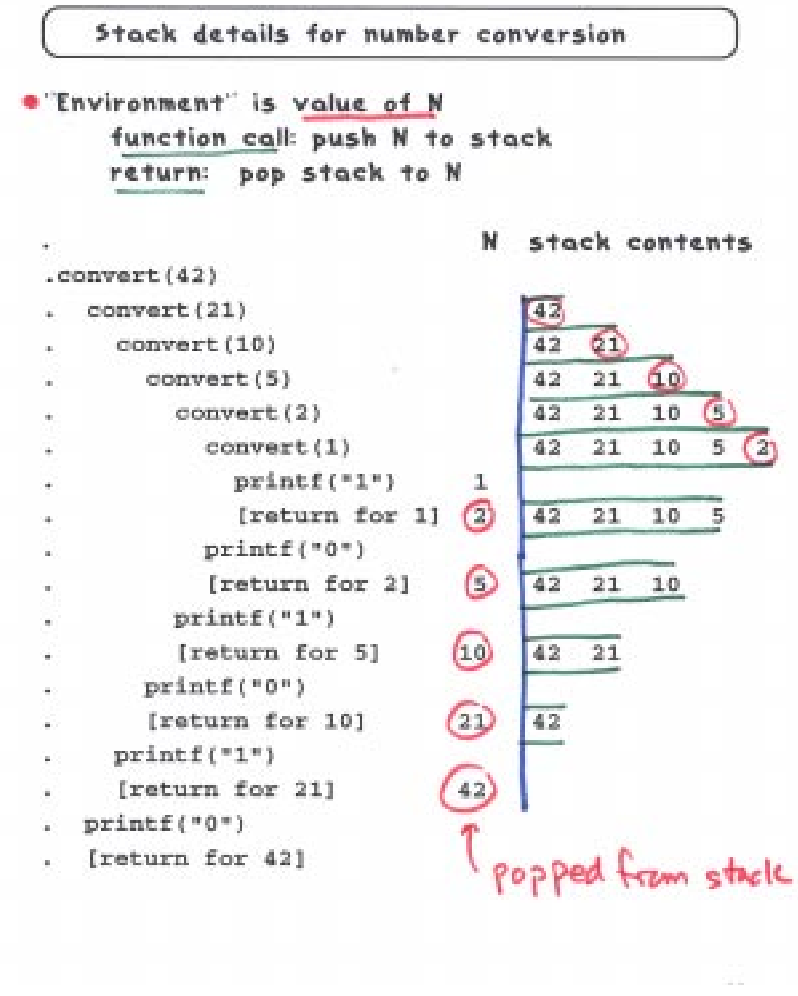
*Any* function call requires system to

- set the values of the parameters
- save the "environment"
- jump to the first instruction in the function
  execute the function
- restore the "environment"
- continue at the instruction after the call
  "return address" (part of environment)

- Use pushdown stack for save/restore
  call: push environment
  return: restore environment from stack

# Demo Use of Stacks to Implement Function Calls





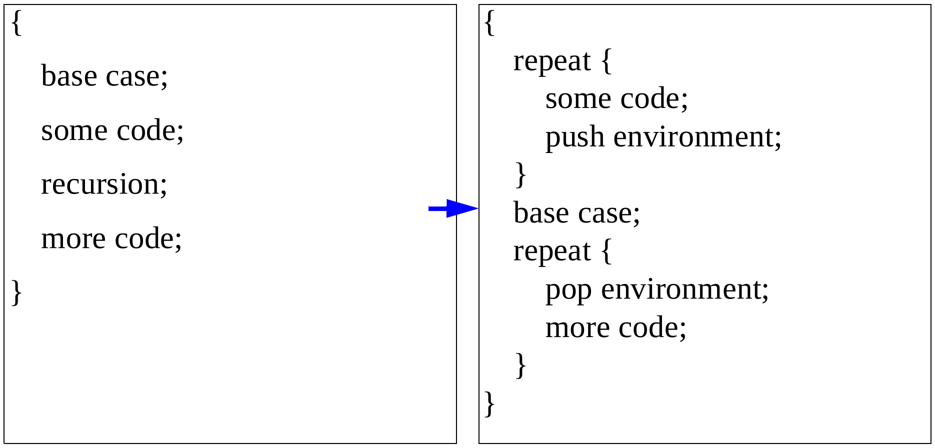
Stack details for number conversion

- "Environment" is value of N
  function call: push N to stack
  return: pop stack to N

```
.                                            N   stack contents
.convert(42)
.  convert(21)                                   42
.    convert(10)                                 42  21
.      convert(5)                                42  21  10
.        convert(2)                              42  21  10  5
.          convert(1)                            42  21  10  5  2
.            printf("1")                     1
.            [return for 1]                  2   42  21  10  5
.          printf("0")
.          [return for 2]                    5   42  21  10
.        printf("1")
.        [return for 5]                     10   42  21
.      printf("0")
.      [return for 10]                      21   42
.    printf("1")
.    [return for 21]                        42
.  printf("0")
.  [return for 42]
```

↑ popped from stack

## Removing Recursion

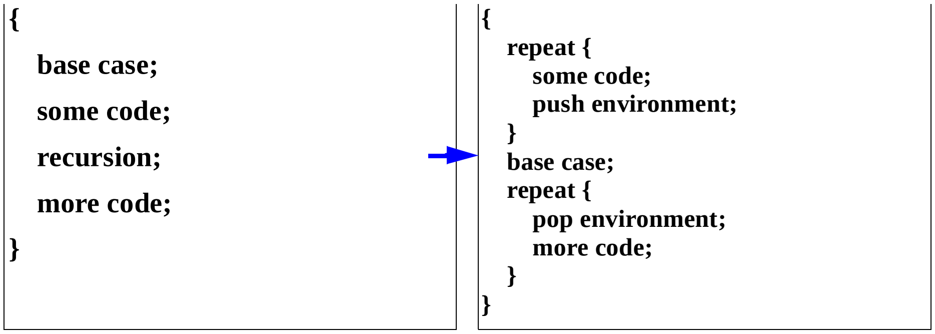


```
{
    base case;

    some code;

    recursion;

    more code;

}
```

```
{
   repeat {
      some code;
      push environment;
   }
   base case;
   repeat {
      pop environment;
      more code;
   }
}
```

- We can remove recursion from any function by using an explicit stack
- Helps us understand nature of the computation (no other reason to do so)

## Removing Recursion



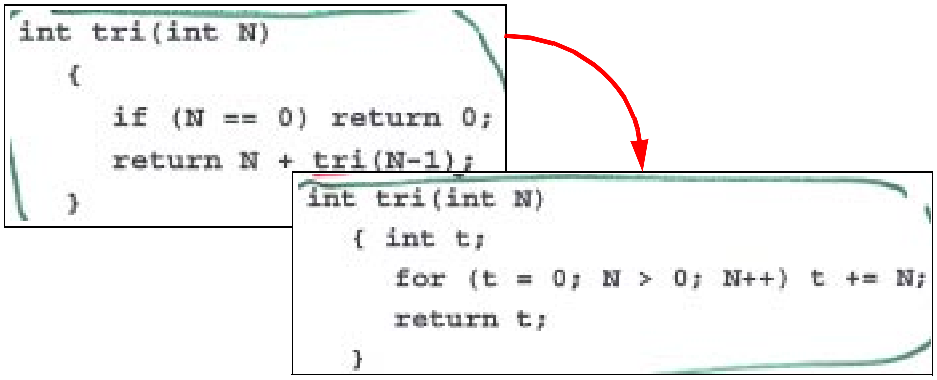
```
{
    base case;

    some code;

    recursion;

    more code;

}
```

```
{
   repeat {
      some code;
      push environment;
   }
   base case;
   repeat {
      pop environment;
      more code;
   }
}
```

Ex: number conversion

```
void convert(int N)
  {
    STACKinit();
    while (N > 0) { STACKpush(N); N = N/2; }
    while (!STACKempty())
      printf("%c", '0'+ STACKpop() % 2);
  }
```

# Tail Recursion

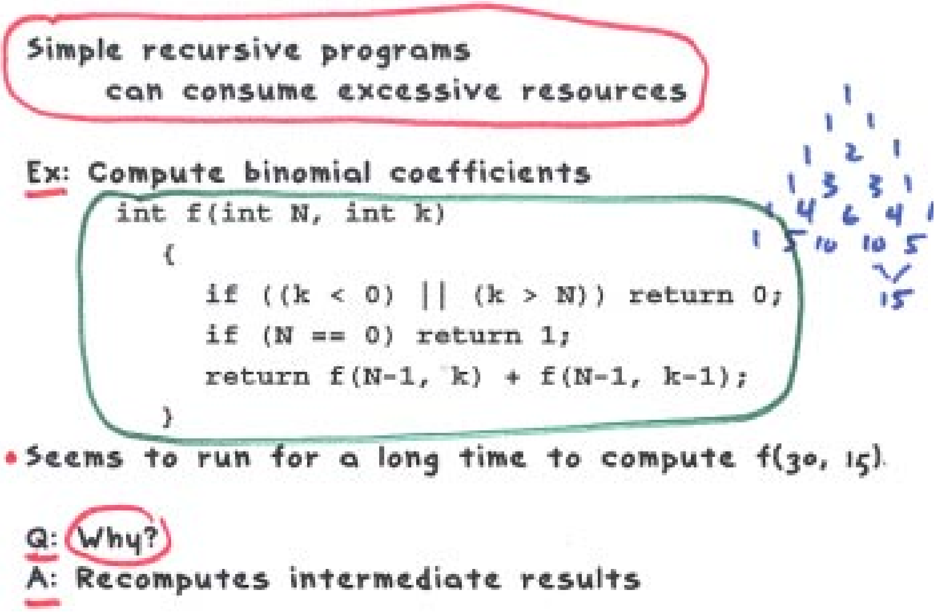


```
int tri(int N)
  {
     if (N == 0) return 0;
     return N + tri(N-1);
  }
```

```
int tri(int N)
  { int t;
     for (t = 0; N > 0; N++) t += N;
     return t;
  }
```

- If single recursive call is the last action, don’t need a stack
- Why?
  - nothing to do after recursion => no need to remember stuff => no need for stack

# Possible Pitfall with Recursion



Simple recursive programs
can consume excessive resources

Ex: Compute binomial coefficients

```
int f(int N, int k)
  {
     if ((k < 0) || (k > N)) return 0;
     if (N == 0) return 1;
     return f(N-1, k) + f(N-1, k-1);
  }
```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5
15

- Seems to run for a long time to compute f(30, 15).

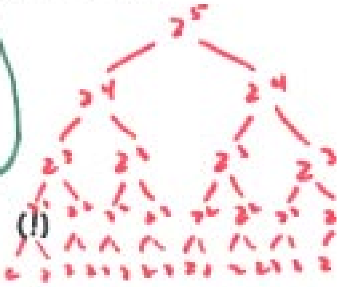
Q: Why?
A: Recomputes intermediate results

# Possible Pitfall with Recursion

- Simpler example: hard way to compute 2^N

```
int f(int N)
  {
    if (N == 0) return 1;
    return f(N-1)+f(N-1);
  }
```

Takes time proportional to 2^N (!)



- DO NOT use these programs!

Solution: DYNAMIC PROGRAMMING

save away intermediate results

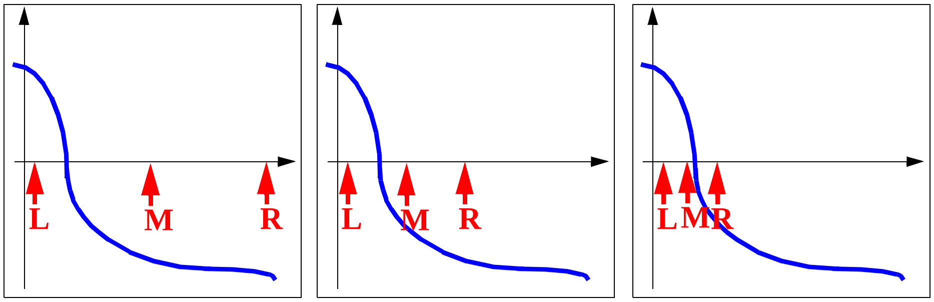
see Sedgewick, section 6.2

P6.9

# Outline

- ~~What is recursion?~~
- ~~How does it work?~~
- Examples

# Divide-and-Conquer



Many computations are
naturally expressed as recursive programs

ITERATION
another way to write "for" loop

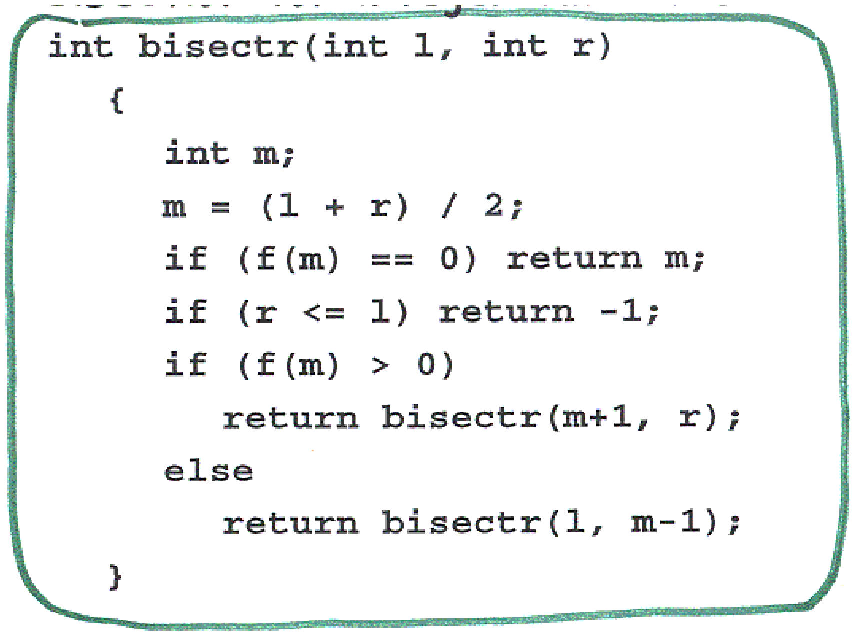
"DIVIDE and CONQUER"
solve a problem by dividing into smaller ones

Ex: root finding via "bisection"

# Finding Root via Bisection

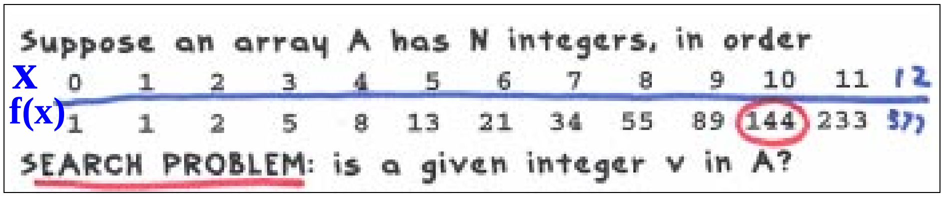
```
float bisectr(float l, float r)
  {
    float m;
    m = (l + r) / 2;
    if ((r - l) < epsilon) return m;
    if (f(m) > 0.0)
      return bisectr(m, r);
    else
      return bisectr(l, m);
  }
```

# Bisection for Integer Functions



```
int bisectr(int l, int r)
   {
      int m;
      m = (l + r) / 2;
      if (f(m) == 0) return m;
      if (r <= l) return -1;
      if (f(m) > 0)
         return bisectr(m+1, r);
      else
         return bisectr(l, m-1);
   }
```

# Binary Search

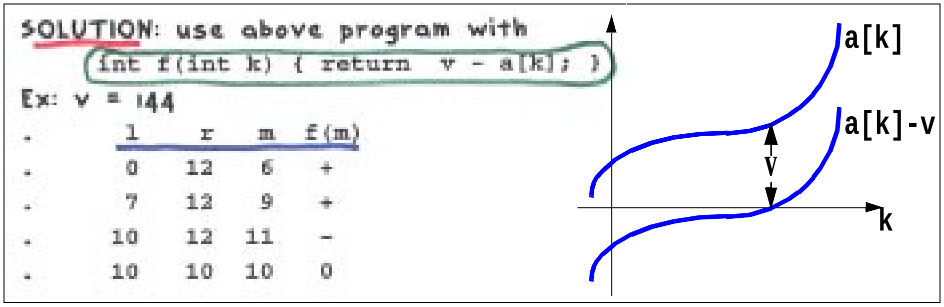


Suppose an array A has N integers, in order

| x | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| f(x) | 1 | 1 | 2 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233 | 377 |

SEARCH PROBLEM: is a given integer v in A?

- Observations:
  - An array is a function mapping integer indices to contents
  - A sorted array is a monotonically increasing function



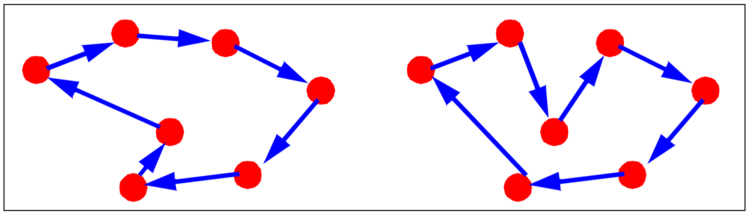
SOLUTION: use above program with

`int f(int k) { return v - a[k]; }`

Ex: v = 144

| | l | r | m | f(m) |
|---|---|---|---|---|
| . | 0 | 12 | 6 | + |
| . | 7 | 12 | 9 | + |
| . | 10 | 12 | 11 | - |
| . | 10 | 10 | 10 | 0 |

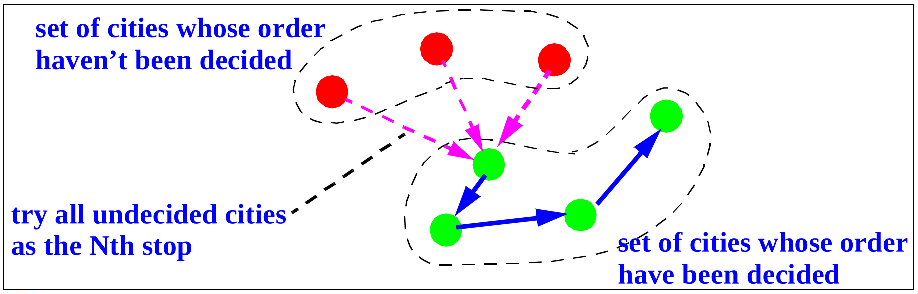
# Traveling Salesman Problem



Given a set of points, find the shortest tour connecting all the points

- Recursive solution for trying all possibilities

# Traveling Salesman Problem



set of cities whose order haven’t been decided

try all undecided cities as the Nth stop

set of cities whose order have been decided

- Recursive solution for trying all possibilities

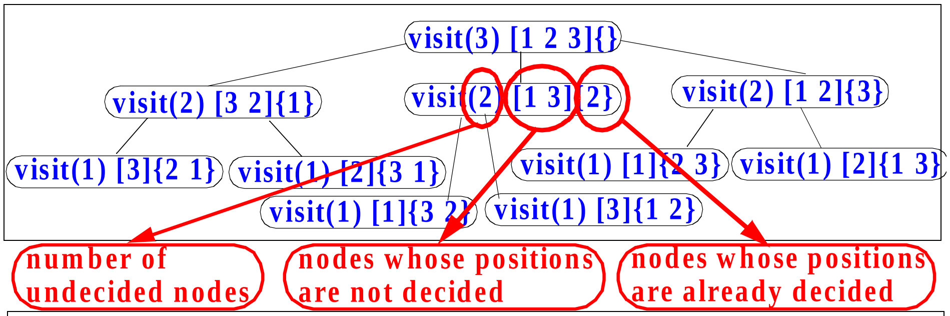
```
visit(int N)
    {
        int i;
        if ( N == 1) { checklength(); return; }
        for (i = 1; i <= N; i++)
            {
                swap(i, N);
                visit(N-1);
                swap(i, N);
            }
    }
```

Number of nodes whose positions have not been decided

Visit ith city as the last (Nth) step

Decide the positions of the other undecided cities

## Traveling Salesman Problem



visit(3) [1 2 3]{}

visit(2) [3 2]{1}

visit(2) [1 3]{2}

visit(2) [1 2]{3}

visit(1) [3]{2 1}

visit(1) [2]{3 1}

visit(1) [1]{3 2}

visit(1) [3]{1 2}

visit(1) [1]{2 3}

visit(1) [2]{1 3}

number of undecided nodes

nodes whose positions are not decided

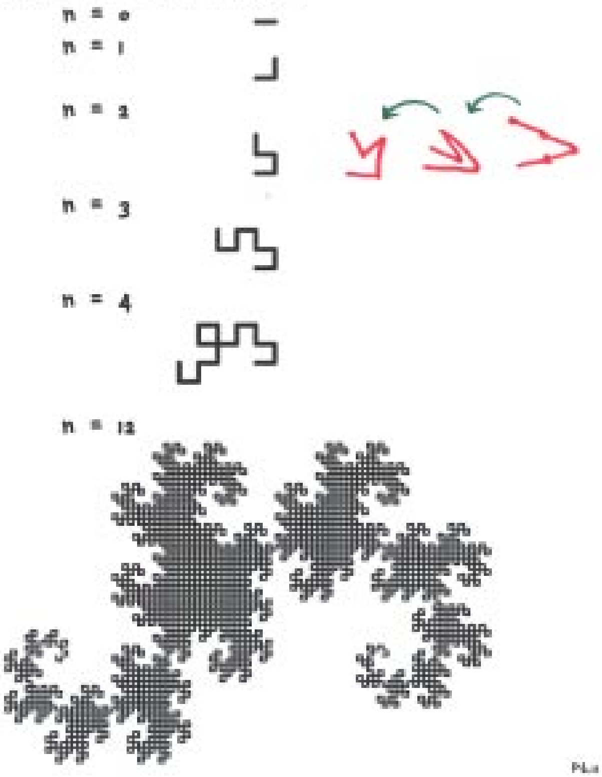
nodes whose positions are already decided

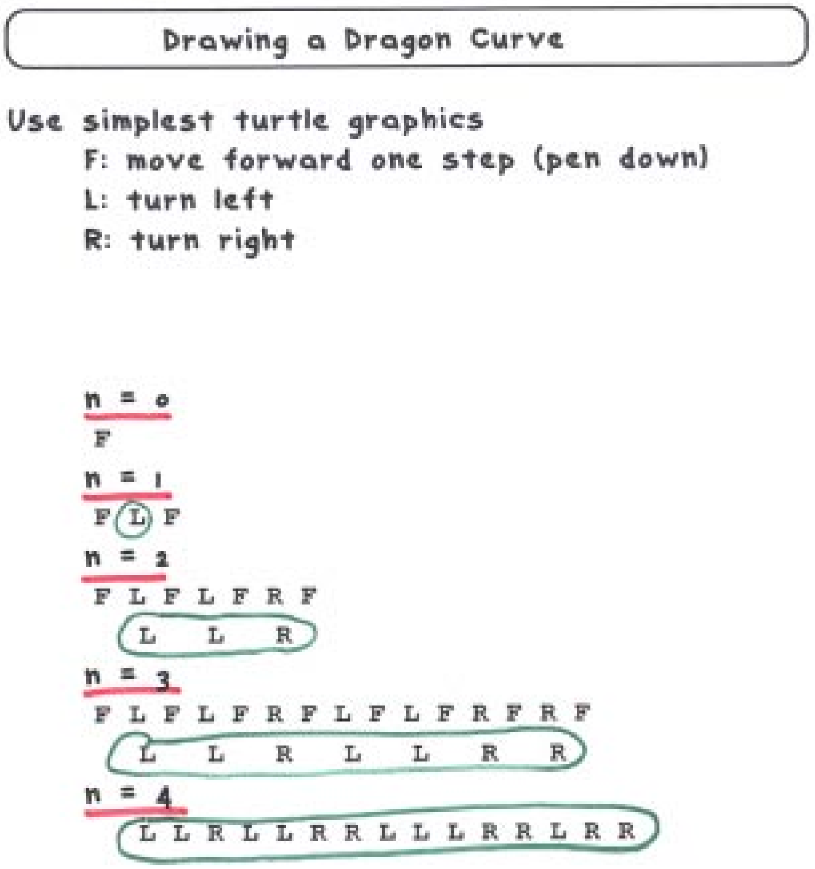
- Takes N! steps
- Can't run for very large N
  no computer can ever run this
  to completion for N = 100   [100! > 10^150]

[stay tuned]

### Recursion: Dragon Curve

Fold a strip of paper in half n times
unfold to right angles





## Drawing a Dragon Curve

Use simplest turtle graphics

F: move forward one step (pen down)

L: turn left

R: turn right

n = 0

F

n = 1

F L F

n = 2

F L F L F R F

L L R

n = 3

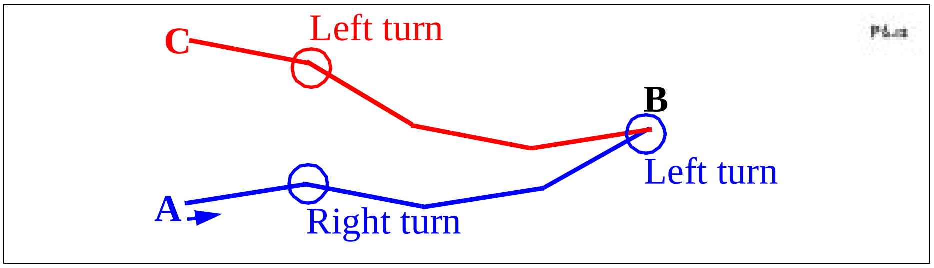
F L F L F R F L F L F R F R F

L L R L L R R

n = 4

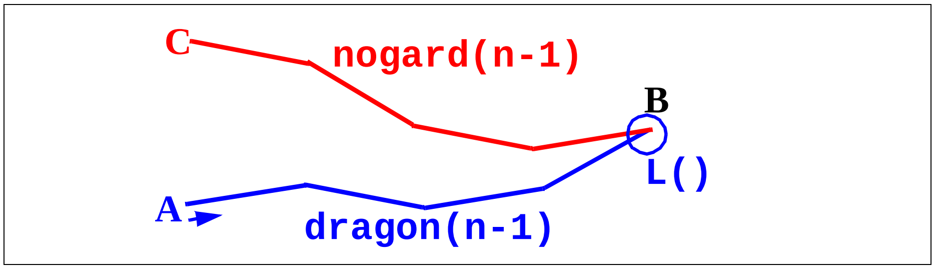
L L R L L R R L L L R R L R R

## Intuition of Algorithm



- AB is a smaller dragon curve by itself
- CB = AB
- Therefore BC is the reverse of AB
- Therefore every turn along BC is the opposite of the corresponding turn on AB

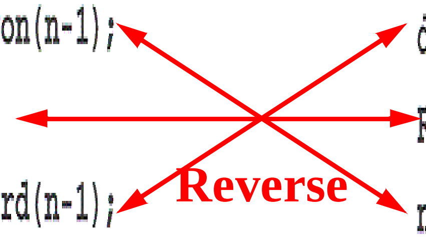
# Recursive Program for Dragon Curve



```
dragon(int n)
{
   if (n == 0) { F(); return; }
   dragon(n-1);
   L();
   nogard(n-1);
}
```

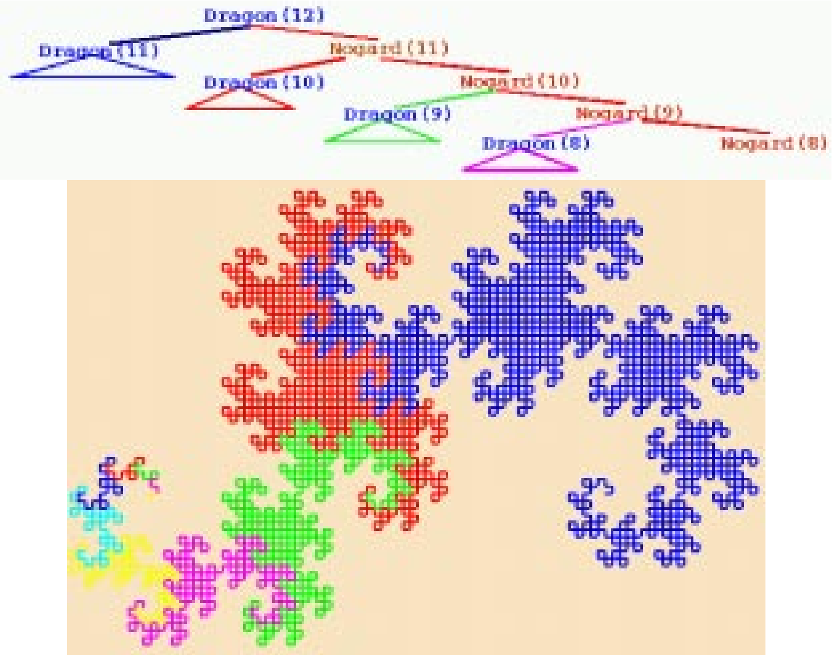
# Backwards Dragon Curve

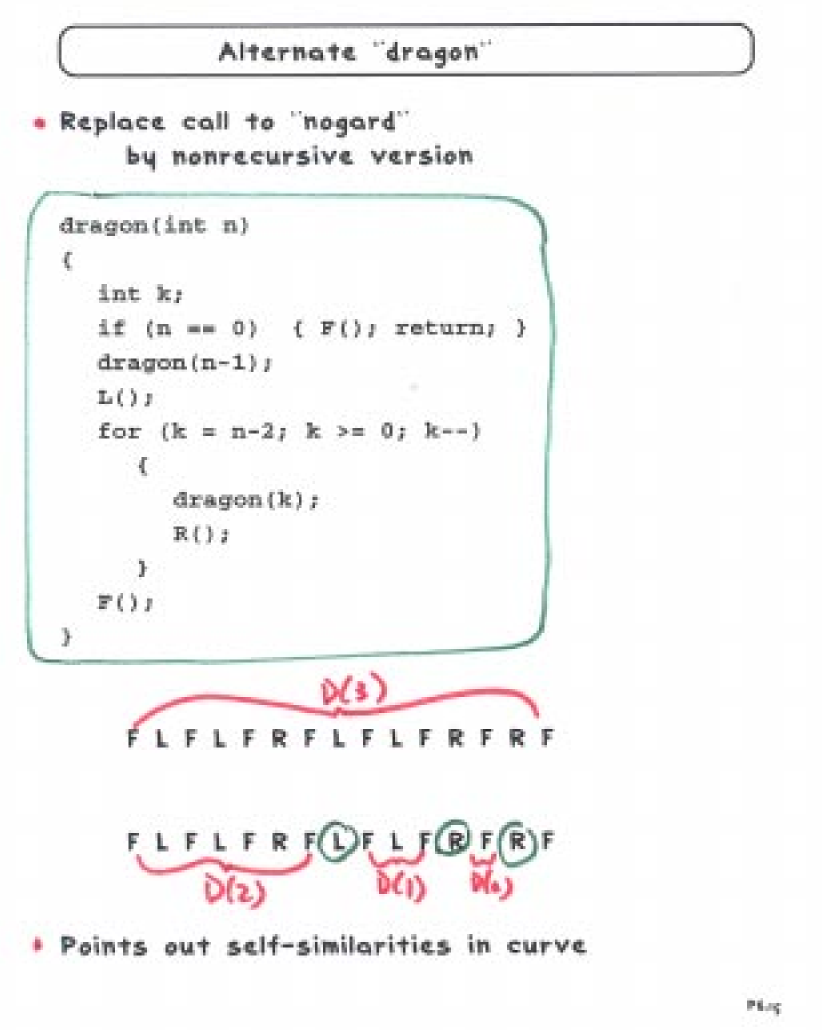
```
dragon(int n)
{
   if (n == 0) { F(); return; }
   dragon(n-1);
   L();
   nogard(n-1);
}
```



```
nogard(int n)
{
   if (n == 0) { F(); return; }
   dragon(n-1);
   R();
   nogard(n-1);
}
```

# `dragon` Demo





## Alternate "dragon"

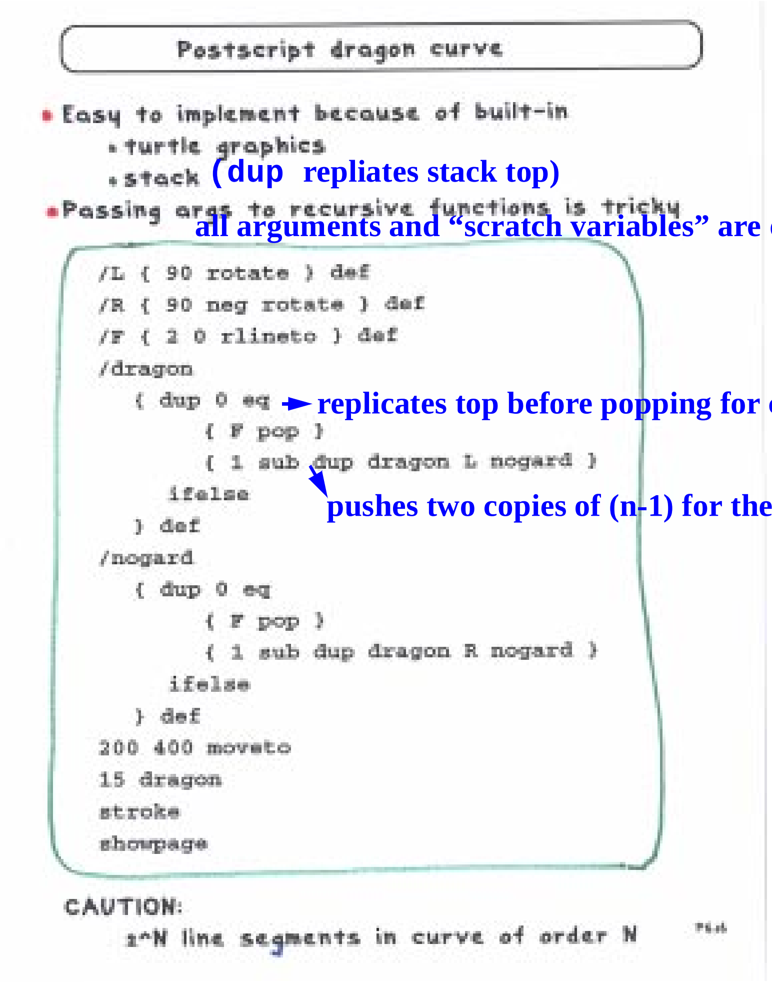
- Replace call to "nogard" by nonrecursive version

```
dragon(int n)
{
   int k;
   if (n == 0)  { F(); return; }
   dragon(n-1);
   L();
   for (k = n-2; k >= 0; k--)
     {
        dragon(k);
        R();
     }
   F();
}
```

$$\overbrace{F\ L\ F\ L\ F\ R\ F\ L\ F\ L\ F\ R\ F\ R\ F}^{D(3)}$$

$$\underbrace{F\ L\ F\ L\ F\ R\ F}_{D(2)}\ \boxed{L}\ \underbrace{F\ L\ F}_{D(1)}\ \boxed{R}\ \underbrace{F}_{D(0)}\ \boxed{R}\ F$$

- Points out self-similarities in curve



## Postscript dragon curve

- Easy to implement because of built-in
  - turtle graphics
  - stack (dup  repliates stack top)
- Passing args to recursive functions is tricky

all arguments and “scratch variables” are on the stack!

```
/L { 90 rotate } def
/R { 90 neg rotate } def
/F { 2 0 rlineto } def
/dragon
   { dup 0 eq
```

→ replicates top before popping for comparison

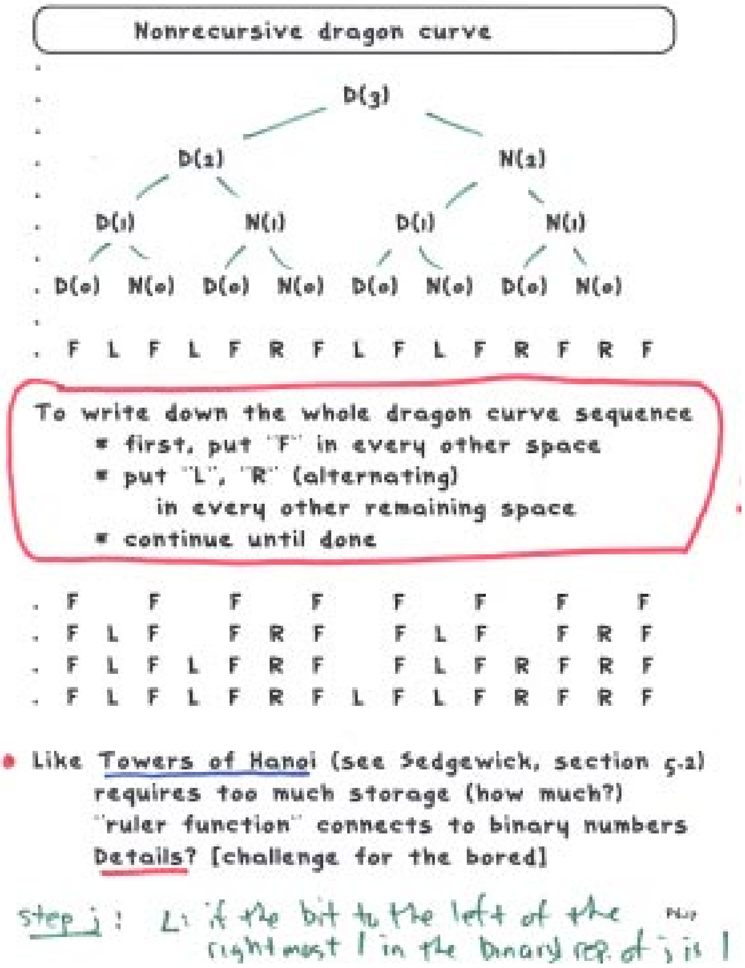
```
        { F pop }
        { 1 sub dup dragon L nogard }
     ifelse
   } def
```

pushes two copies of (n-1) for the two recursive calls

```
/nogard
   { dup 0 eq
        { F pop }
        { 1 sub dup dragon R nogard }
     ifelse
   } def
200 400 moveto
15 dragon
stroke
showpage
```

CAUTION:

2^N line segments in curve of order N



## Nonrecursive dragon curve

D(3)

D(2) N(2)

D(1) N(1) D(1) N(1)

D(0) N(0) D(0) N(0) D(0) N(0) D(0) N(0)

F L F L F R F L F L F R F R F

To write down the whole dragon curve sequence
- first, put "F" in every other space
- put "L", "R" (alternating) in every other remaining space
- continue until done

```
. F   F   F   F   F   F   F   F
. F L F   F R F   F L F   F R F
. F L F L F R F   F L F R F R F
. F L F L F R F L F L F R F R F
```

- Like Towers of Hanoi (see Sedgewick, section 5.2) requires too much storage (how much?) "ruler function" connects to binary numbers Details? [challenge for the bored]

step i : L: if the bit to the left of the rightmost 1 in the binary rep. of i is 1

# What We Have Learned

- How recursion works
  - A recursive call is no different from a “regular” call
  - It involves saving the old environment for later return
- Learn to trace the execution of given recursive programs (using pictures)
- Learn to write simple recursion
  - What’s the base case?
  - What’s the induction case?