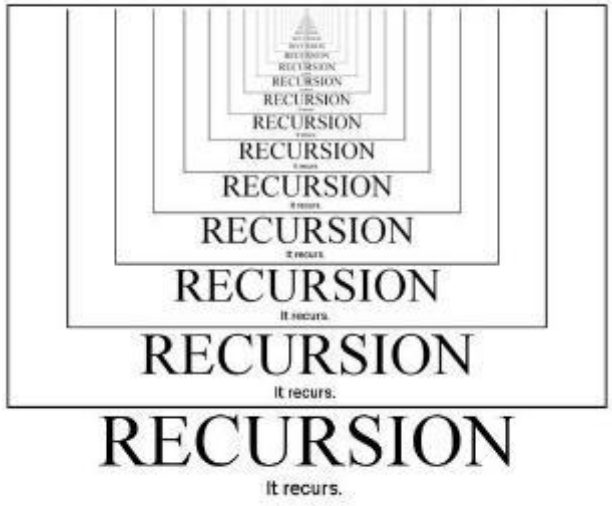
RECURSION

OVERVIEW

OVERVIEW

- **What is recursion?**

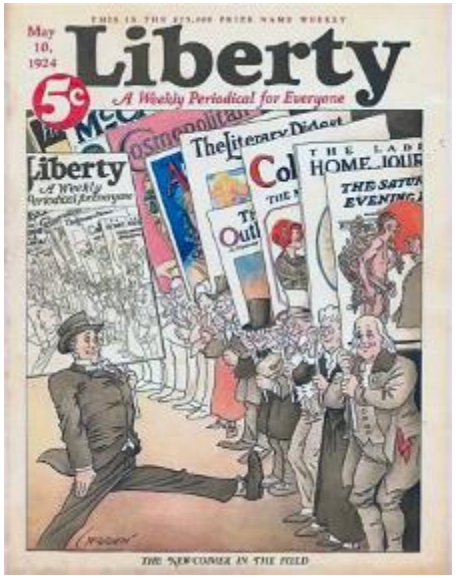
Nested Russian
Matryoshka dolls



Defining a problem
in terms of itself

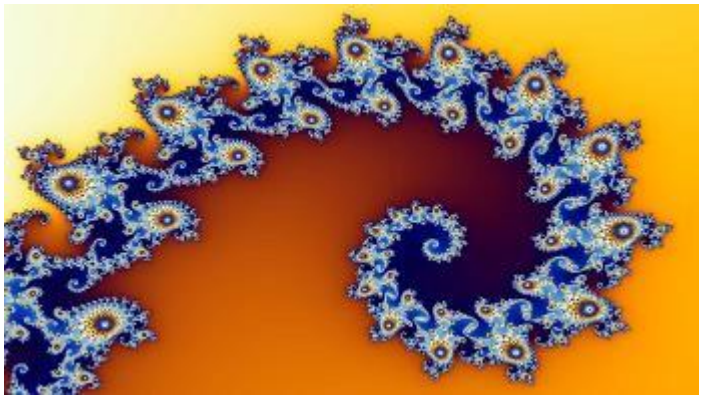
OVERVIEW

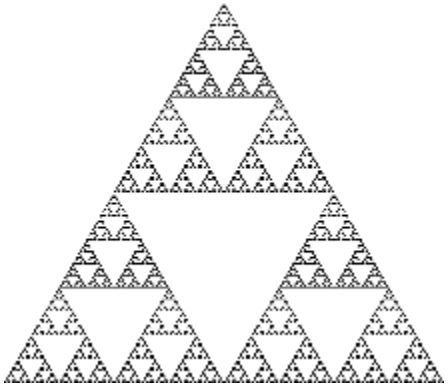
- **Recursive artwork: picture appears inside picture**

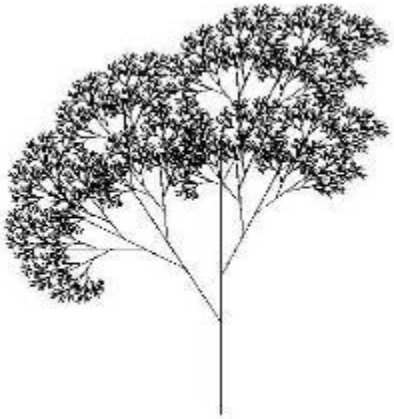


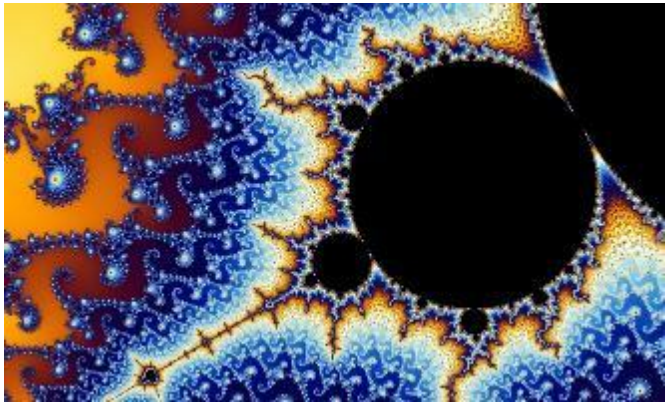
OVERVIEW

- **Recursive fractals: self similar patterns**



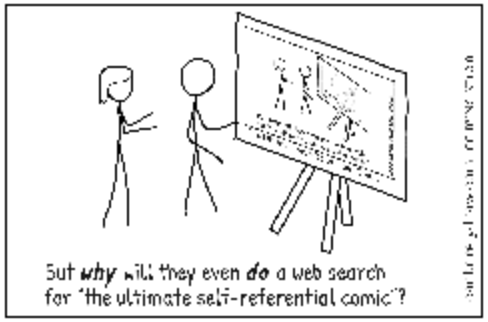






OVERVIEW

- **Recursive jokes: refer to themselves**



The Ultimate Self-Referential Comic.

OVERVIEW

- **In computer science, recursion is a method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem – Knuth**

- **Recursion is very old:**
  - Factorial – India 300BC
  - Greatest common divisor – Greece 300BC
  - Fibonacci – India 200BC

- **Recursion is very powerful:**
  - Binary Search
  - Towers of Hanoi
  - N-Queens

OVERVIEW

- **In this section we will discuss a variety of classic recursive algorithms for solving computer science problems**
  - Explain the recursive algorithms
  - Discuss their implementation
  - Use box method to trace execution

- **We will calculate and compare the speeds of algorithms**
  - Some are logarithmic – O(logN)
  - Some are linear – O(N)
  - Some are exponential – $O(2^N)$

RECURSION

FACTORIAL

FACTORIAL

- **Intuitive definition of N factorial**
  - The product of all integers from 1 to N
  - N ! = N * (N-1) * (N-2) * … 3 * 2 * 1

- **Recursive definition of N factorial**
  - N ! = N * (N-1) ! ← Recurrence relationship
  - 1 ! = 1
  - 0 ! = 1

  ← Two terminating conditions

- **We can turn this into code very easily**

FACTORIAL

```
int factorial(int num)
{
    // Handle terminating condition
    if (num <= 1)
        return 1;

    // Handle recursive case
    else
        return (num * factorial(num - 1));
}
```

Calling factorial function with smaller parameter value

FACTORIAL

```
int factorial(int num)
{
    // Handle terminating condition
    int result = 1;

    // Handle recursive case
    if (num > 1)
        result = num * factorial(num - 1);
    return result;
}
```

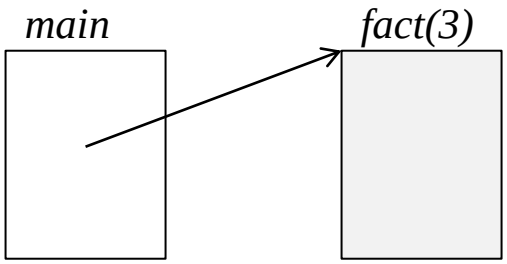
Calling factorial function with smaller parameter value

FACTORIAL

- **We can use the box method to trace the execution of the factorial function**
  - Goal is to draw a diagram to visualize execution
  - Draw a box representing each function call
  - Show parameters at top of box
  - Use arrows to show each recursive call
  - Use arrows to show return values

FACTORIAL

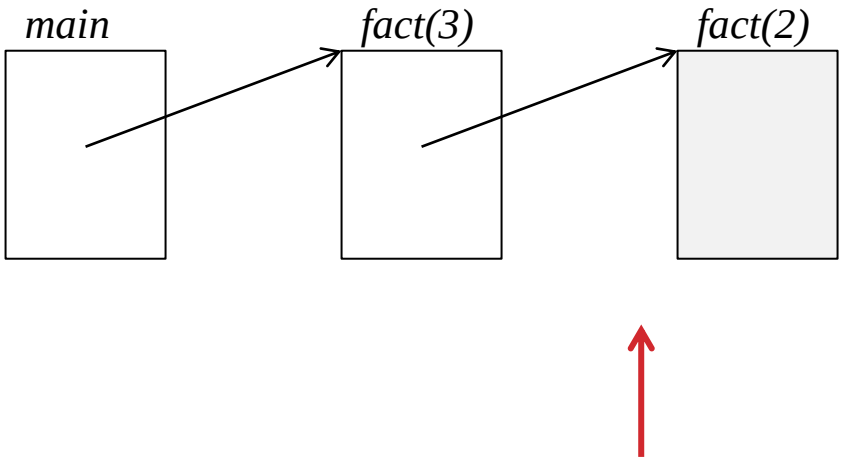
- **What happens when we call fact(3) in main program?**



We leave main program and
execute fact(3)

FACTORIAL

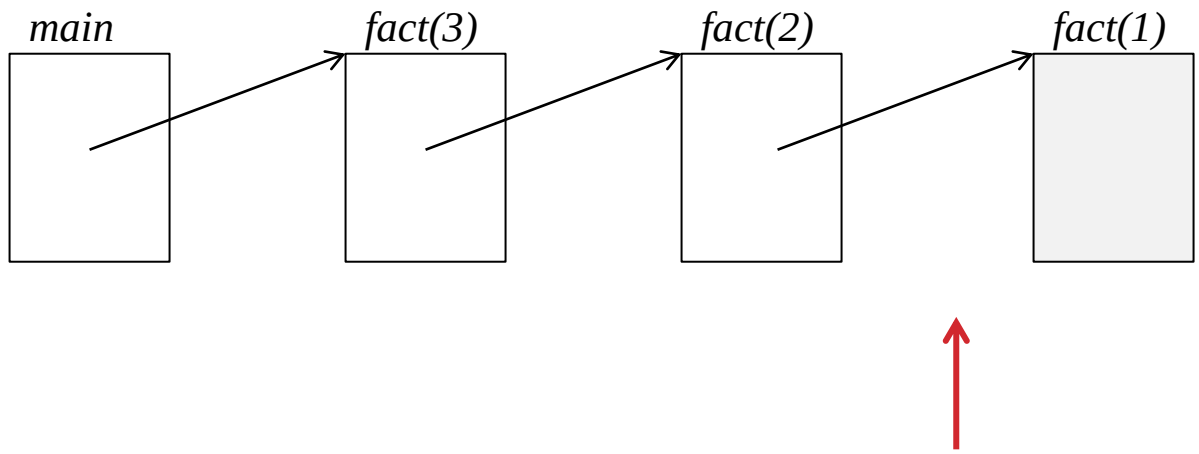
- **What happens when we call fact(3) in main program?**



We execute function with
num=3 and call fact(2)

FACTORIAL

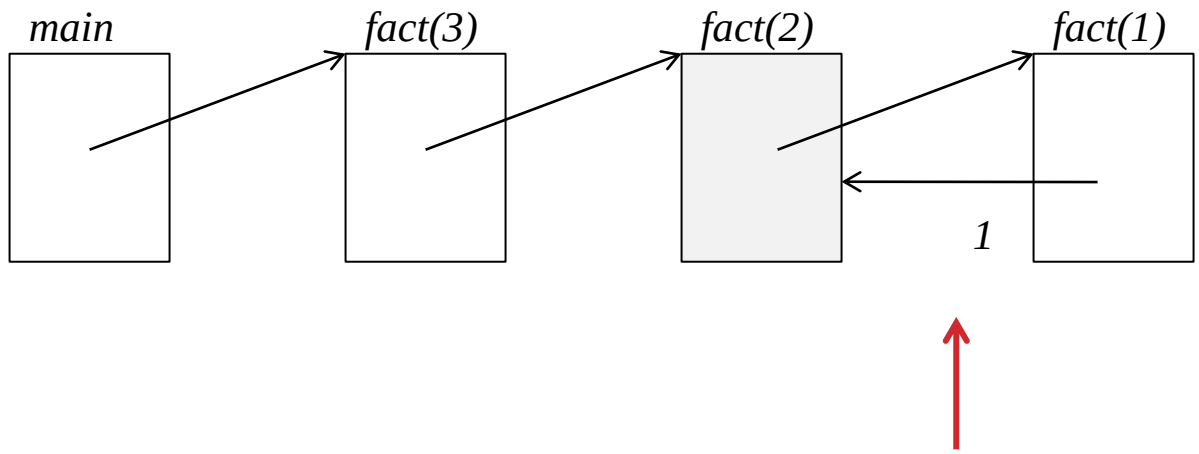
- **What happens when we call fact(3) in main program?**



We execute function with num=2 and call fact(1)

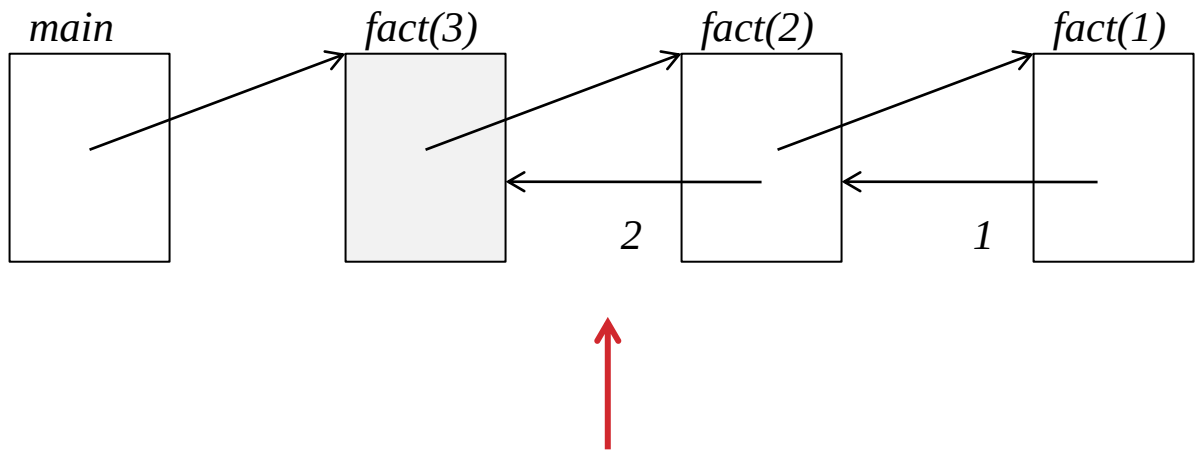
FACTORIAL

- **What happens when we call fact(3) in main program?**



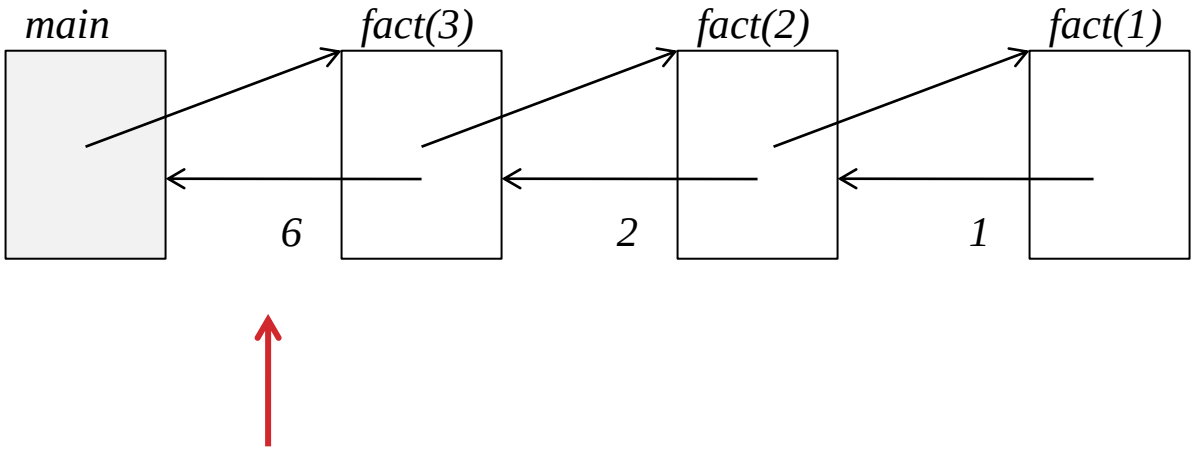
FACTORIAL

- **What happens when we call fact(3) in main program?**



FACTORIAL

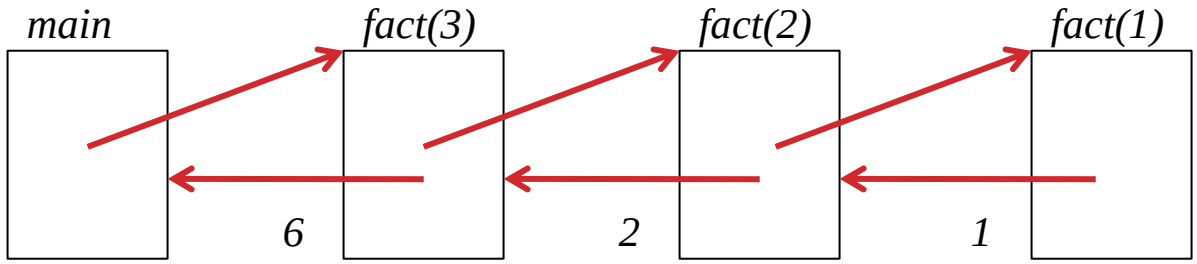
- **What happens when we call fact(3) in main program?**



We calculate and
return 3*2=6

FACTORIAL

- **What happens when we call fact(3) in main program?**



Notice that the arrows make a continuous closed loop and end up back at the main function

FACTORIAL

- **How calls to factorial are needed to get answer?**
  - calls(1) = 1
  - calls(2) = 1 + calls(1) = 2
  - calls(3) = 1 + calls(2) = 3
  - …
  - calls(N) = 1 + calls(N-1) = N

- **This recursive function is linear because it takes O(N) calls to calculate the correct answer**

FACTORIAL

```
int factorial(int num)
{
    // Iterative solution
    int result = 1;
    while (num > 1)
    {
        result = result * num;
        num--;
    }
    return result;
}
```

We can also calculate factorial with a loop that executes N times

This will be slightly faster because there is no function call overhead

FACTORIAL

- **Conclusions:**
  - The factorial function can be described recursively using
  - N ! = N * (N-1) !
  - 1 ! = 1
  - 0 ! = 1

- **Recursive solution takes O(N) steps**
- **Iterative solution also takes O(N) steps**

RECURSION

POWER

POWER

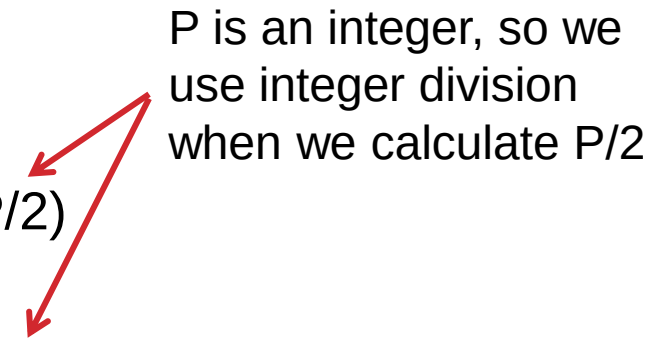
- **Intuitive definition of the power function X^P**
  - X^P is the product of X times itself P times
  - Simple to implement when P is an integer

```
float power(float X, int P)
{
   int result = 1;
   while (P >= 1)
   {
      result *= X;
      P--;
   }
   return result;
}
```

POWER

- **Is there a recursive solution?**
  - Consider power(2,6) = 2*2*2 * 2*2*2 = 64
  - Notice that we are calculating 2*2*2 two times above
  - Hence power(2,6) = power(2,3) * power(2,3)
  - We can generalize this as follows

- **When P is even:**
  - power(X,P) = power(X,P/2) * power(X,P/2)
- **When P is odd:**
  - power(X,P) = power(X,P/2) * power(X,P/2) * X



POWER

- **When should we stop the recursion?**
  - We have the following terminating conditions:
  - power(X,1) = X
  - power(X,0) = 1

- **Is this recursive solution faster?**
  - Yes, if we save and reuse our calculations of power(X,P/2)

POWER

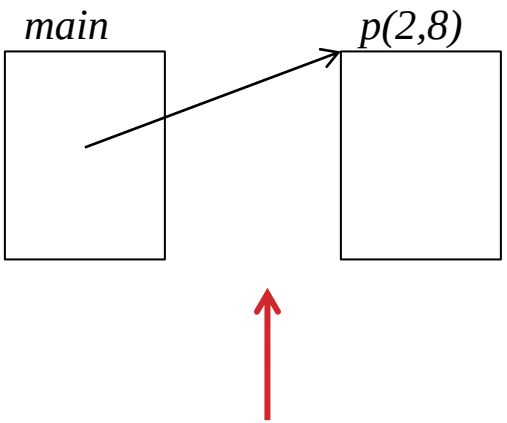
```
float power(float X, int P)
{
    // Handle terminating conditions
    if (P == 0) return 1;
    if (P == 1) return X;

    // Handle recursive cases
    float temp = power(X,P/2);
    if (P % 2 == 0)
        return temp * temp;
    if (P % 2 == 1)
        return temp * temp * X;
}
```

Calculate and save power result for P/2

Reuse this result to handle even and odd power cases

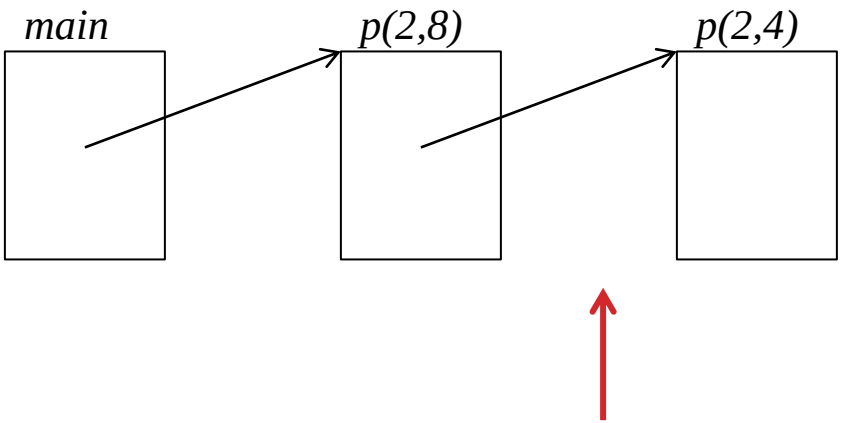
POWER

- **Consider box method trace of power(2,8)**



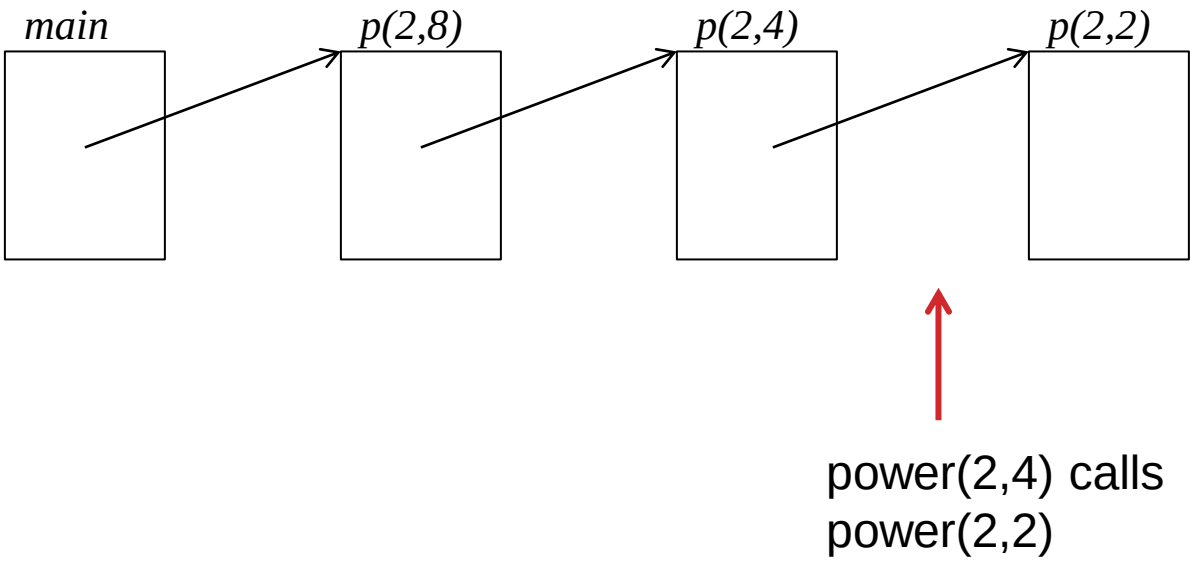
POWER

- **Consider box method trace of power(2,8)**



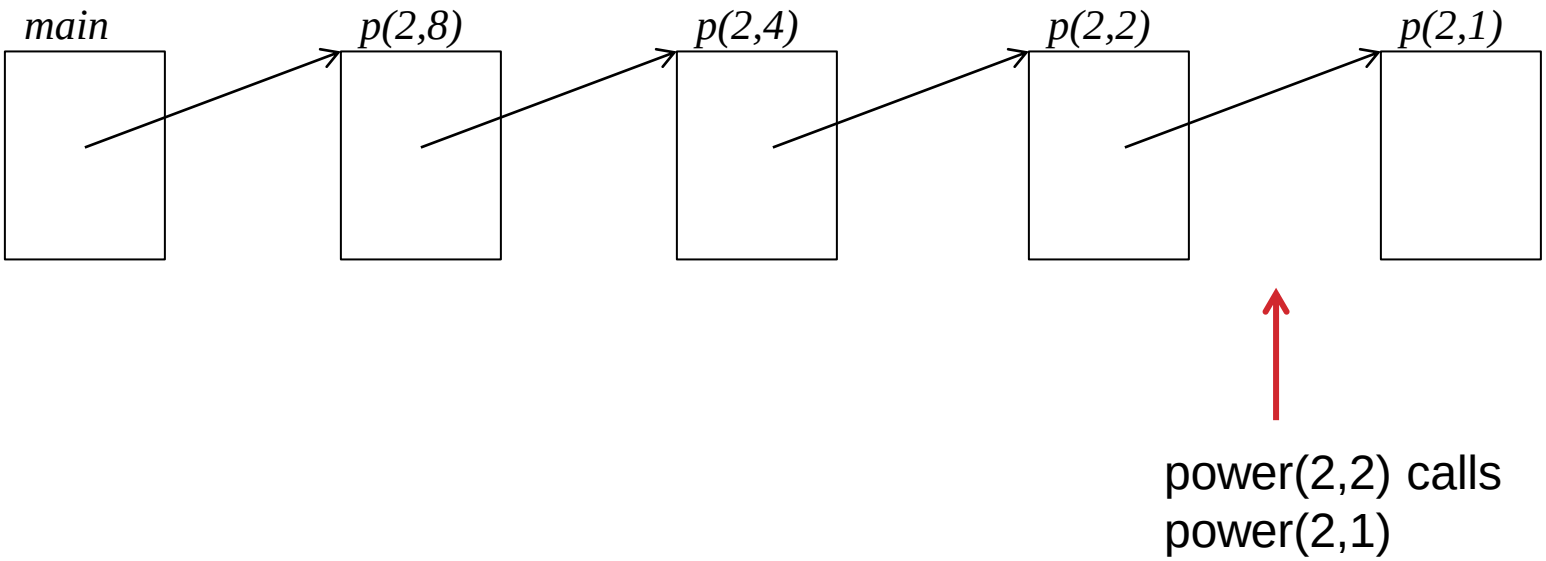
POWER

- **Consider box method trace of power(2,8)**



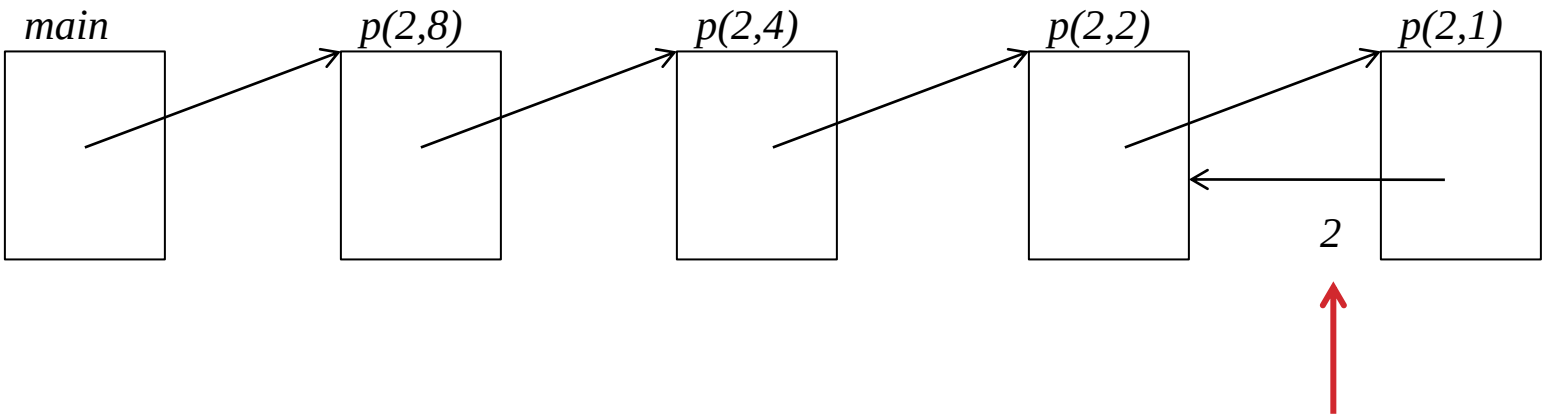
POWER

- **Consider box method trace of power(2,8)**



POWER

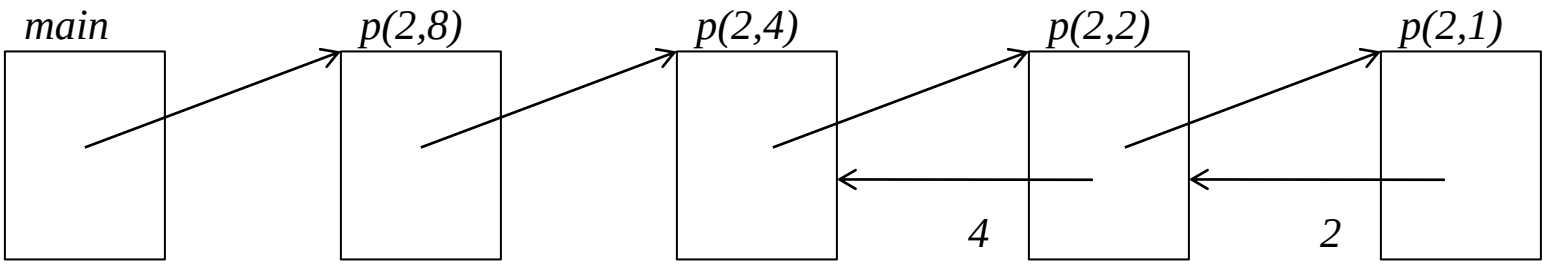
- **Consider box method trace of power(2,8)**



We reach P=1 terminating condition and return X=2

POWER

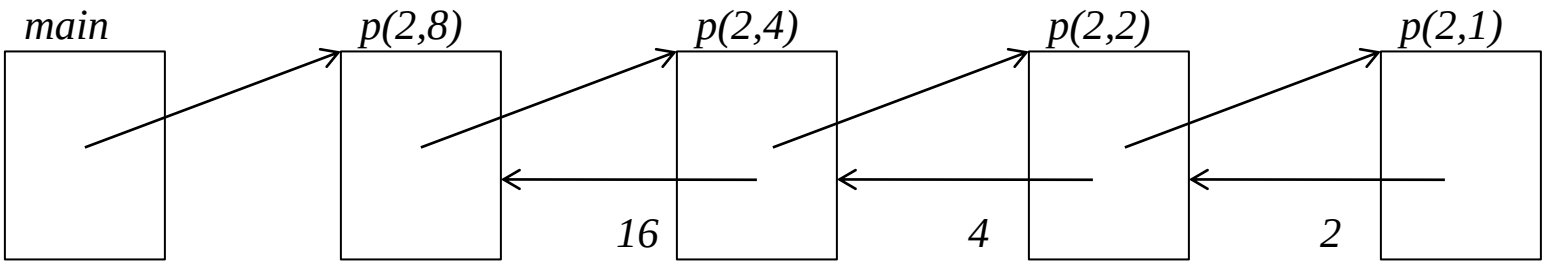
- **Consider box method trace of power(2,8)**



We calculate and return 2*2=4

POWER

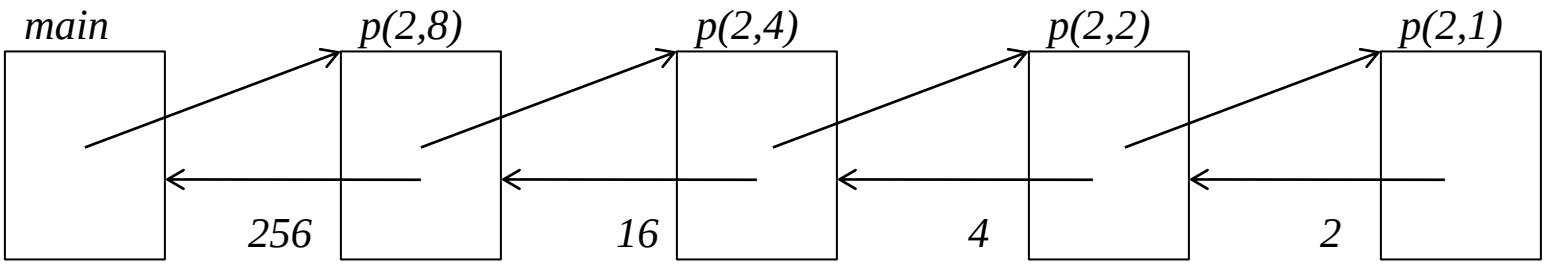
- **Consider box method trace of power(2,8)**



We calculate and return 4*4=16

POWER

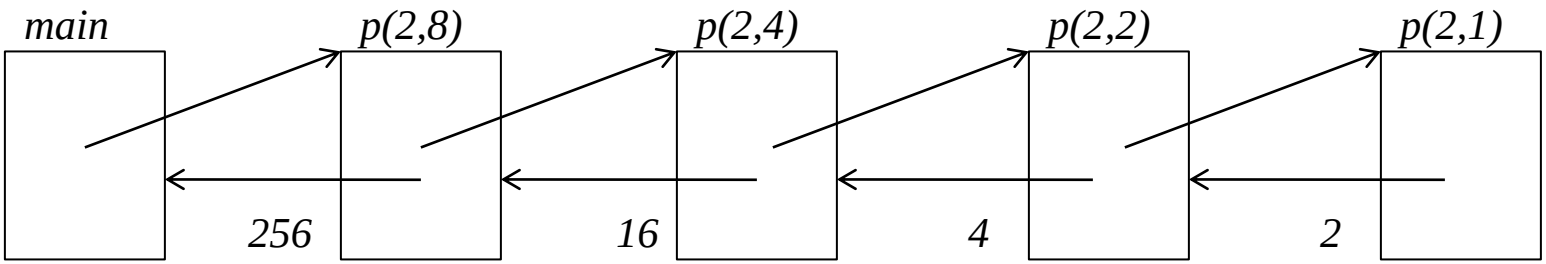
- **Consider box method trace of power(2,8)**



We calculate and
return 16*16=256

POWER

- **Consider box method trace of power(2,8)**



- Notice that we only performed 3 multiplies instead of 8 multiplies
- In general, we only need $O(\log_2 P)$ multiplies to calculate X^P
- Unfortunately, this technique does not work for float powers

POWER

- **Conclusions:**
  - The integer power function can be described recursively
  - power(X,P) = power(X,P/2) * power(X,P/2) for even P
  - power(X,P) = power(X,P/2) * power(X,P/2) * X for odd P
  - power(X,1) = X
  - power(X,0) = 1

- **Recursive solution is logarithmic and takes O(logN) steps**
- **Naive iterative solution is linear and takes O(N) steps**

RECURSION

FIBONACCI

FIBONACCI

- **Fibonacci was a 12th century Italian mathematician who popularized the Arabic numeral system in Europe and is known for the Fibonacci sequence 1,1,2,3,5,8,13,21,34,55…**

- **Notice that each number in the sequence is equal to the sum of the two previous numbers**
  - Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)

- **The terminating conditions for this recursion are:**
  - Fibonacci(1) = 1
  - Fibonacci(2) = 1

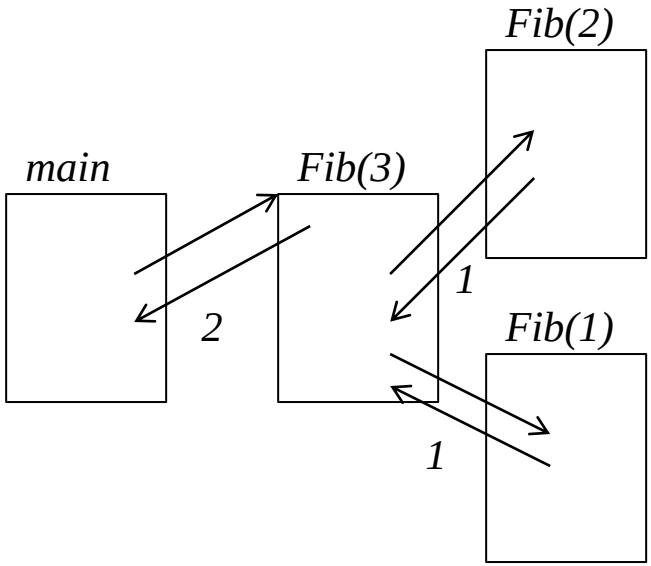
FIBONACCI

- **This recursive function calls itself two times each time it is executed so the number of recursive calls grows rapidly**

```
int Fibonacci(const int Num)
{
   cout << "calling Fibonacci " << Num << endl;
   if (Num <= 2)
      return(1);
   else
      return( Fibonacci(Num-1) + Fibonacci(Num-2) );
}
```

FIBONACCI

- **Consider box method trace of Fib(3)**
  - First, we call Fib(2)
  - Then, we return to Fib(3)
  - Then, we call Fib(1)



FIBONACCI

- **Program output for Fib(3)**

calling Fibonacci 3

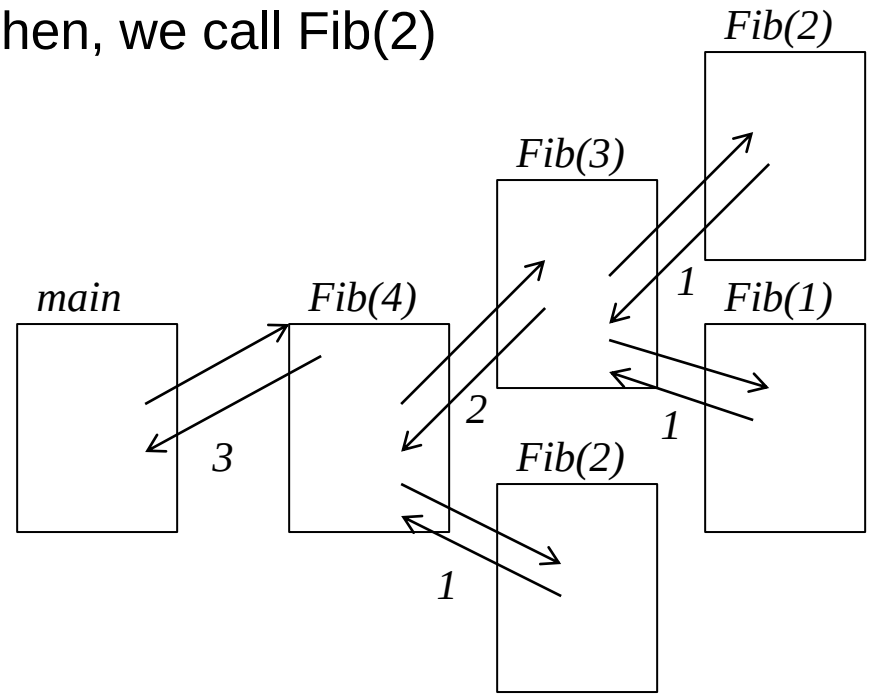
calling Fibonacci 2

calling Fibonacci 1

There are 2 Recursive calls from Fibonacci(3)

FIBONACCI

- **Consider box method trace of Fib(4)**
  - First, we call Fib(3) as above
  - Then, we return to Fib(4)
  - Then, we call Fib(2)



FIBONACCI

- **Program output for Fib(4)**

calling Fibonacci 4

calling Fibonacci 3

calling Fibonacci 2

calling Fibonacci 1

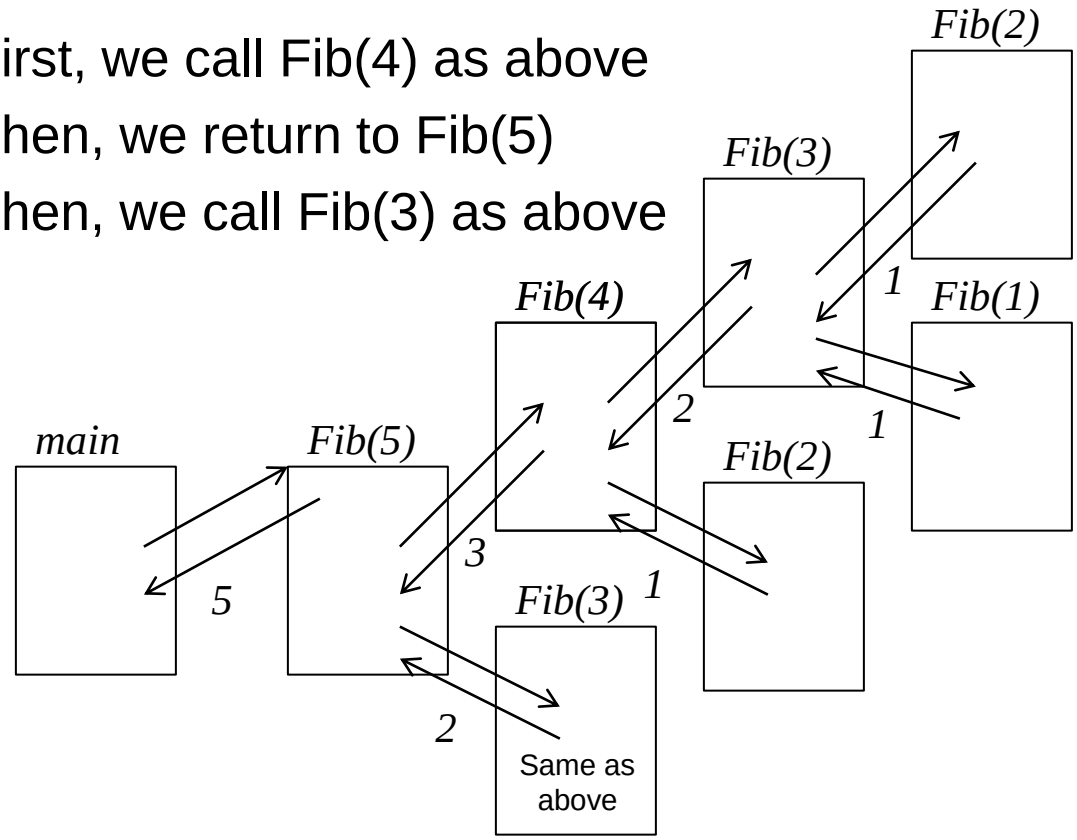
calling Fibonacci 2

First, we calculate Fibonacci(3)

Then, we calculate Fibonacci(2)

FIBONACCI

- **Consider box method trace of Fib(5)**
  - First, we call Fib(4) as above
  - Then, we return to Fib(5)
  - Then, we call Fib(3) as above



FIBONACCI

- **Program output for Fib(5)**

calling Fibonacci 5

calling Fibonacci 4
calling Fibonacci 3
calling Fibonacci 2
calling Fibonacci 1
calling Fibonacci 2

First, we calculate Fibonacci(4)

calling Fibonacci 3
calling Fibonacci 2
calling Fibonacci 1

Then, we calculate Fibonacci(3)

FIBONACCI

- **How can we calculate the number of recursive function calls needed to calculate the Nth Fibonacci number?**

- **Recursive definition:**
  - calls(N) = calls(N-1) + calls(N-2) + 1
  - calls(1) = 1
  - calls(2) = 1

- **The number of calls is 1,1,3,5,9,15,25,41,67,109…**
  - This is an example of exponential growth $O(2^N)$
  - As N increases calls(N) → 2 * Fibonacci(N)

FIBONACCI

```
int Fibonacci2(const int Num)
{
   int Num1 = 1;
   int Num2 = 1;
   for (int Count = 1; Count < Num; Count++)
   {
      int Num3 = Num1 + Num2;
      Num1 = Num2;
      Num2 = Num3;
   }
   return (Num1);
}
```

Question: How much work does this iterative algorithm require?

FIBONACCI

```
int Fibonacci3(const int Num)
{
    // Calculation using Binet's formula
    double golden = (1 + sqrt(5)) / 2;
    double Binet = (pow(golden, Num) –
                        pow(-golden, -Num)) / sqrt(5);
    return Binet;
}
```

Question: How much work does this closed form algorithm require?

FIBONACCI

- **Conclusions:**
  - The Fibonacci function can be described recursively
  - Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
  - Fibonacci(1) = 1
  - Fibonacci(2) = 1

- **Recursive solution take $O(2^N)$ steps**
- **Simple iterative solution takes O(N) steps**
- **Closed form solution takes O(1) step**

RECURSION

BINARY SEARCH

BINARY SEARCH

- **Assume we are given a sorted array of integers**

- **Binary search can be used to quickly search this array**
  - Look at middle element of sorted array
  - If equal to desired value, you found it
  - If less than desired value, search right half of array
  - If greater than desired value, search left half of array
  - Repeat until data is found (or no data left to search)

BINARY SEARCH

- **Search for value 7 in sorted array below**

- **Look at middle location (0+7)/2 = 3, which contains 5**

- **5 < 7, so search to right**

- **This cuts size of array we are searching in half**

BINARY SEARCH

- **Search for value 7 in unsearched array below**

- **Look at middle location (4+7)/2 = 5, which contains 7**

- **We found the desired value in only 2 searching steps!**

BINARY SEARCH

- **Search for value 2 in sorted array below**

- **Look at middle location (0+7)/2 = 3, which contains 5**

- **5 > 2, so search to left**

- **This cuts size of array we are searching in half**

BINARY SEARCH

- **Search for value 2 in unsearched array below**

- **Look at middle location (0+2)/2 = 1, which contains 3**

- **3 > 2, so search to left**

- **Now there is only one location to search!**

BINARY SEARCH

- **Search for value 2 in unsearched array below**

- **Look at middle location (0+0)/2 = 0, which contains 2**

- **We found the desired value in only 3 searching steps!**

BINARY SEARCH

- **This divide and conquer approach is very fast since the array we are searching is cut in half at each step**
  - Consider an array with 1024 sorted values
  - Searching we go from 1024 → 512 → 256 → 128 → 64 → 32 → 16 → 8 → 4 → 2 → 1
  - Only 10 steps needed to search array of 1024 elements

- **In general, binary search takes $\log_2 N$ steps to search a sorted array of N elements**
  - About 20 steps to search array of 1,000,000 elements
  - About 30 steps to search array of 1,000,000,000 elements

BINARY SEARCH

- **To implement binary search, we need to keep track of the portion of the array we are searching**
  - Min = smallest array index of unsearched portion
  - Max = largest array index of unsearched portion
  - Mid = (Min + Max) / 2 is middle position
  - We need to initialize Min=0 and Max=N-1
  - We need to update these values as we search

- **This binary search algorithm can be implemented using iteration or recursion**

BINARY SEARCH

```
// Iterative binary search
int Search(int Desired, int Data[], int Min, int Max)
{
   // Search array using divide and conquer approach
   int Mid = (Min + Max) / 2;
   while ((Min <= Max) && (Data[Mid] != Desired))
   {
      // Change min to search right half
      if (Data[Mid] < Desired)
         Min = Mid+1;
…
```

This loop will end when data is found, or no locations are left to search

We change lower array index here to be 1 to right of midpoint

BINARY SEARCH

```
…
    // Change max to search left half
    else if (Data[Mid] > Desired)
       Max = Mid-1;


    // Update mid location
    Mid = (Min + Max) / 2;
  }
…
```

We change upper array index here to be 1 to left of midpoint

BINARY SEARCH

```
…
   // Return results of search
   if ((Min <= Max) && Data[Mid] == Desired))
      return(Mid);
   else
      return(-1);
}
```

This returns the array index of desired data value or -1 if not found

BINARY SEARCH

- **Tracing iterative binary search for value 2**

Min	Max	Mid	Data[Mid]	Search
0	7	3	5	Left

BINARY SEARCH

- **Tracing iterative binary search for value 2**

Min	Max	Mid	Data[Mid]	Search
0	7	3	5	Left
0	2	1	3	Left

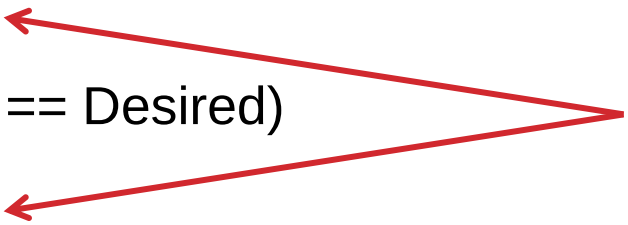
BINARY SEARCH

- **Tracing iterative binary search for value 2**

Min	Max	Mid	Data[Mid]	Search
0	7	3	5	Left
0	2	1	3	Left
0	0	0	2	Found

BINARY SEARCH

```
// Recursive binary search
int Search( int Desired, int Data[], int Min, int Max )
{
   // Terminating conditions
   int Mid = (Min + Max ) / 2;
   if (Max < Min)
      return(-1);
   else if (Data[Mid] == Desired)
      return(Mid);
…
```



This returns the array index of desired data value or -1 if not found

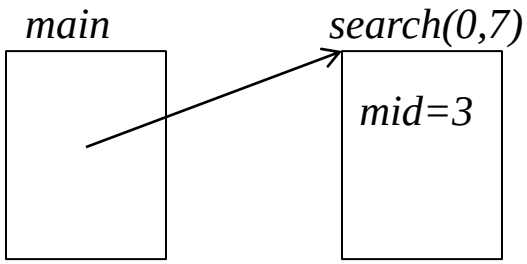
BINARY SEARCH

```
…
   // Recursive call to search right half
   else if (Data[Mid] < Desired)
      return( Search( Desired, Data, Mid+1, Max ) );

   // Recursive call to search left half
   else if (Data[Mid] > Desired)
      return( Search( Desired, Data, Min, Mid-1 ) );
}
```

Notice how we search a smaller part of the array with each recursive function call

BINARY SEARCH

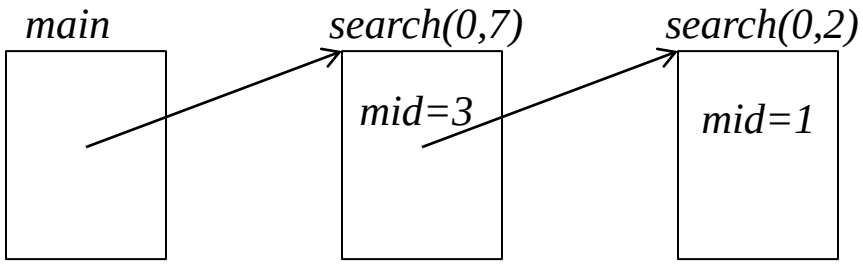
- **Box method trace of recursive binary search for value 2**



- We calculate mid = 3 and see data[3] = 5 is greater than 2

BINARY SEARCH

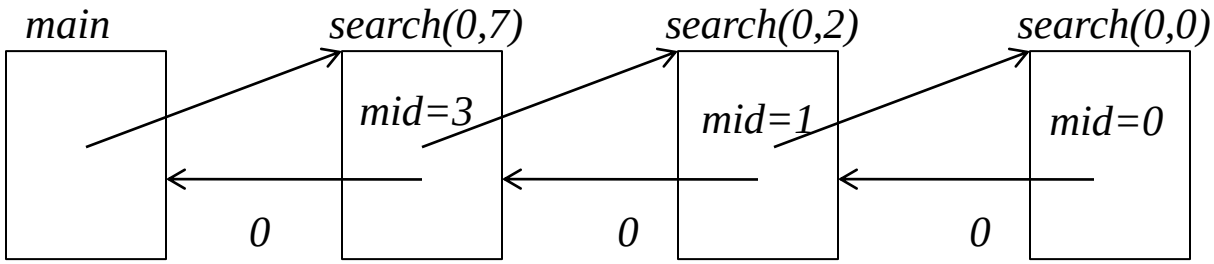
- **Box method trace of recursive binary search for value 2**



- We calculate mid = 3 and see data[3] = 5 is greater than 2
- Make recursive call to search(0,2)
- We calculate mid = 1 and see data[1] = 3 is greater than 2

BINARY SEARCH

- **Box method trace of recursive binary search for value 2**



- We calculate mid = 3 and see data[3] = 5 is greater than 2
- Make recursive call to search(0,2)
- We calculate mid = 1 and see data[1] = 3 is greater than 2
- Make recursive call to search(0,0)
- We calculate mid = 0 and see data[0] = 2 is equal to 2 so we return 0

BINARY SEARCH

- **Conclusions:**
  - Binary search can be described recursively
  - search(data,min,max,value) = -1 when min>max
  - search(data,min,max,value) = mid when data[mid] = value
  - search(data,min,max,value) = search(data,mid+1,max,value) when data[mid] < value
  - search(data,min,max,value) = search(data,min,mid-1,value) when data[mid] > value

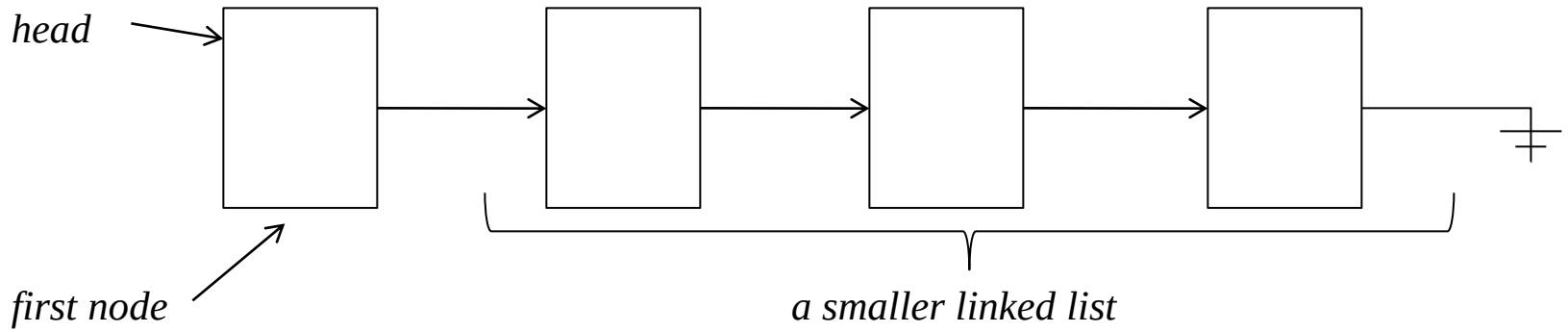
- **Recursive solution takes O(logN) steps**
- **Iterative solution also takes O(logN) steps**

RECURSION

LINKED LISTS

LINKED LISTS

- **A linked list is sometimes called a recursive data type because a linked list can be defined in terms of itself**



- **A linked list is either**
  - Empty (NULL head pointer)
  - One node connected to a smaller linked list

LINKED LISTS

- **Recursive list operations work as follows:**
  - Start at the head of the list
  - If the head pointer is NULL, then stop
  - Otherwise perform desired operation
  - Recursively process the rest of the list

- **When we implement these list operations, we will need to pass the list pointer as a parameter to the recursive calls**
  - Value parameter: Print, Search
  - Reference parameter: Insert, Delete

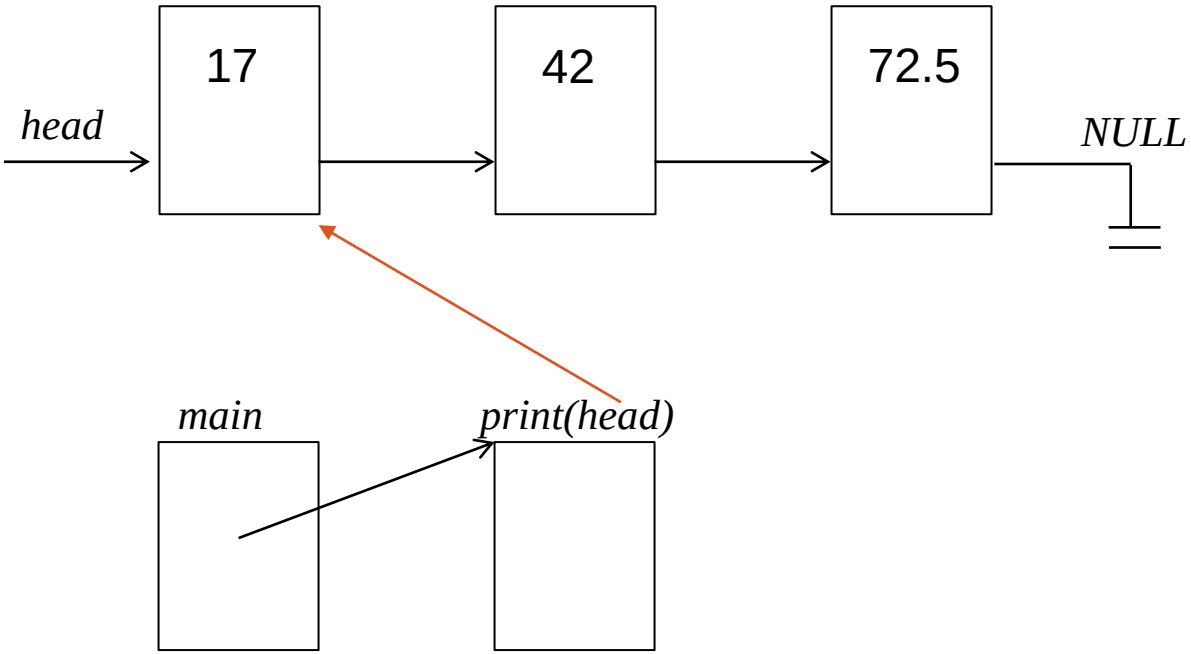
LINKED LISTS

```
void Print(Node * head)
{
   // Check terminating condition
   if (head == NULL) return;

   // Print first node
   cout << head->Value << endl;

   // Print rest of list
   Print(head->Next);
}
```

Notice that we do not have a loop in this recursive function

LINKED LISTS

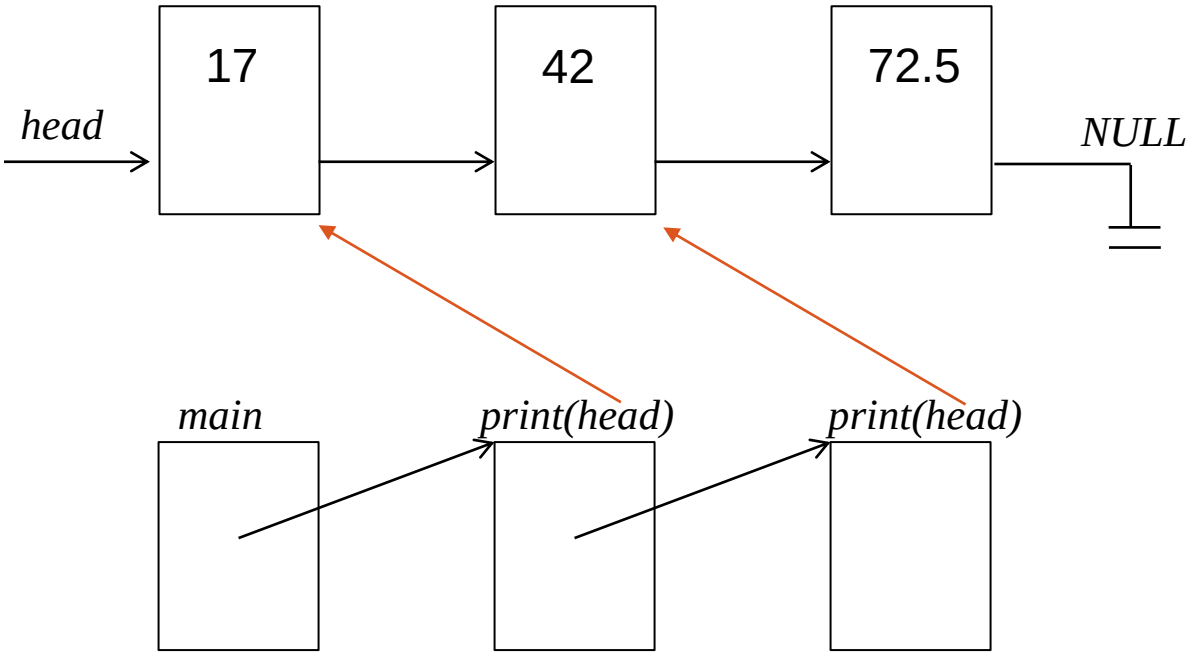
- **What happens when we call Print(head) in main program?**



Head points to first node in the linked list so 17 is printed

LINKED LISTS

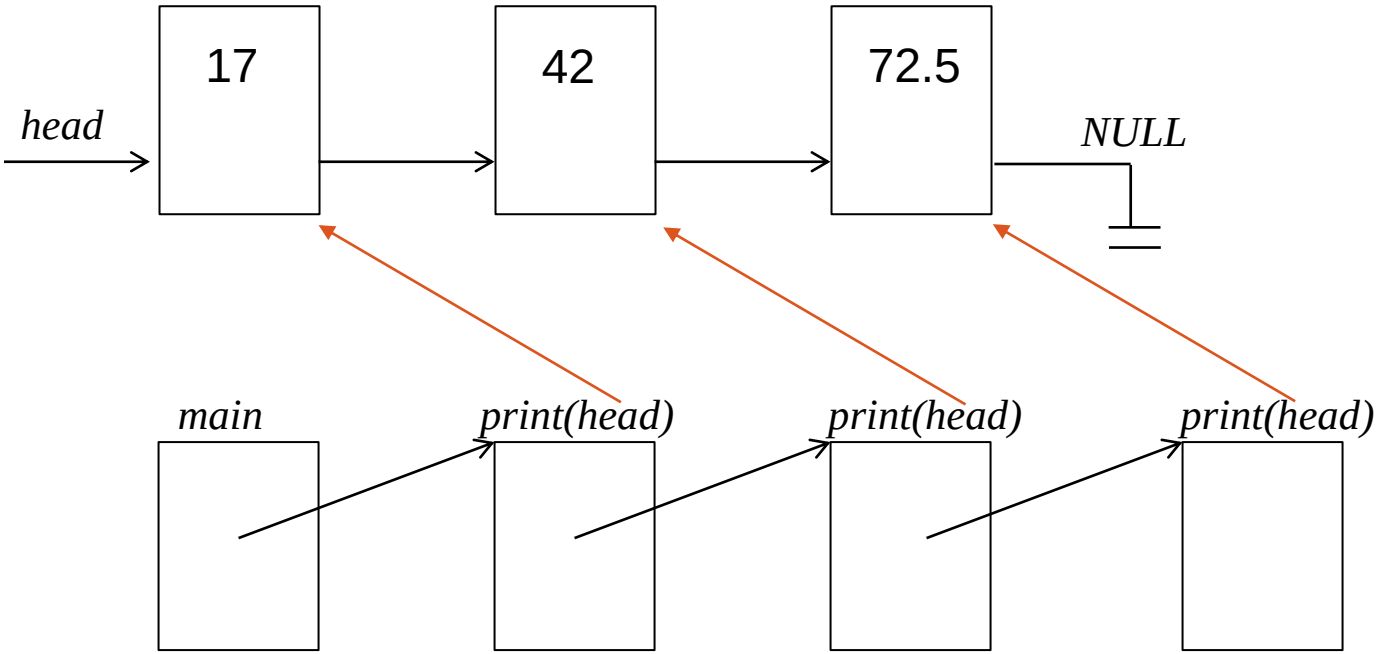
- **What happens when we call Print(head) in main program?**



Head points to second node in the linked list so 42 is printed

LINKED LISTS

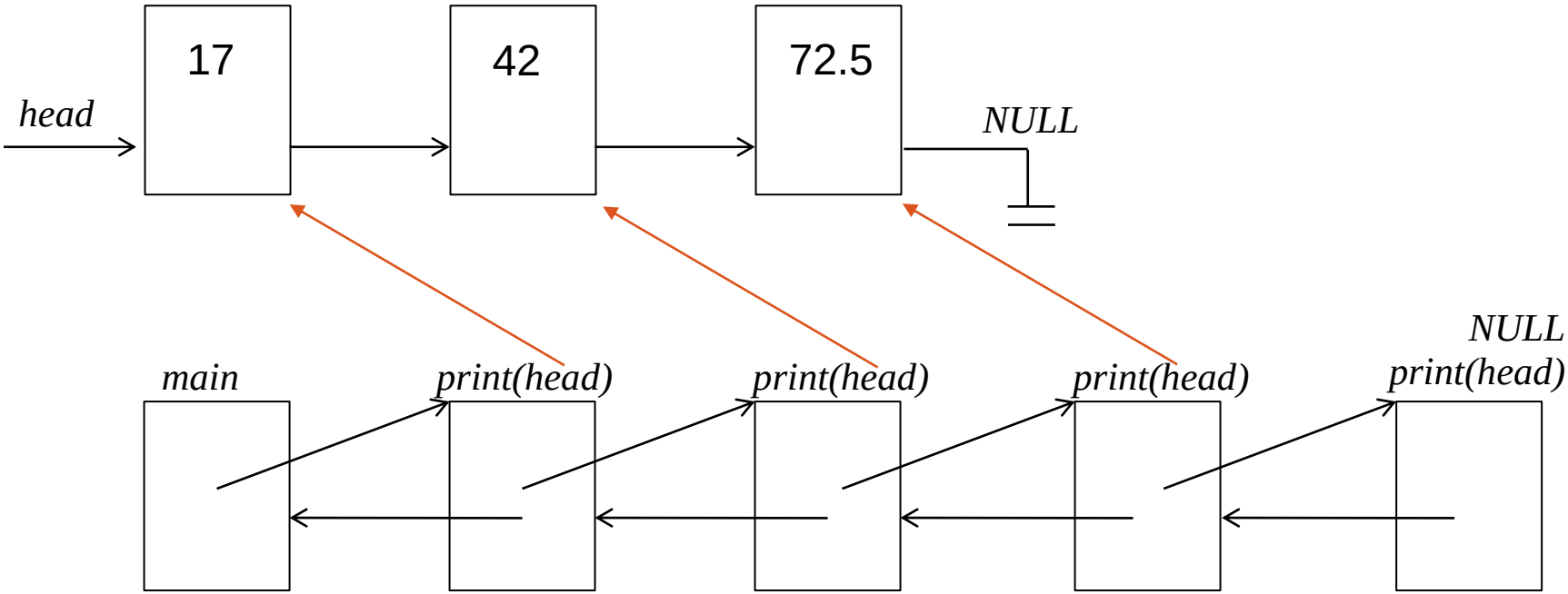
- **What happens when we call Print(head) in main program?**



Head points to third node in the linked list so 72.5 is printed

LINKED LISTS

- **What happens when we call Print(head) in main program?**



Head points to NULL so the function returns

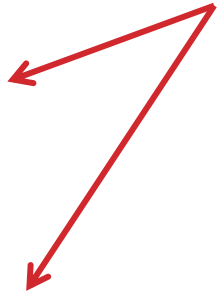
LINKED LISTS

```
void Print(Node * head)
{
   // Check terminating condition
   if (head == NULL) return;

   // Print rest of list
   Print(head->Next);

   // Print first node
   cout << head->Value << endl;
}
```

What happens if we move the recursive call before the print operation?

LINKED LISTS

```
bool Search(Node * head, string value)
{
   // Check terminating condition
   if (head == NULL) return false;

   // Check if node is found
   if (head->Value == value) return true;

    // Handle recursive case
   Search(head->Next, value);
}
```



This function returns T/F if the value is found in list or not

We only make recursive call if T/F not returned

LINKED LISTS

```
void SortedInsert(Node * & head, string value)
{
   // Check terminating condition
   if ((head == NULL) || (head->Value > value))  {
      Node *temp = new Node();
      temp->Value = value;
      temp->Next = head;
      head = temp;
      }
```

Head pointer is reference parameter because we will change the linked list

This will insert the new node before the head pointer to keep the linked list data in sorted order

LINKED LISTS

…

```
   // Handle recursive case
   else
      SortedInsert(head->Next, value);
}
```

Otherwise, go to the next node in linked list and try to insert data

LINKED LISTS

```
void Delete(Node * & head, string value)
{
   // Check terminating condition
   if (head == NULL) return;

   // Delete node if found
   if (head->Value == value)  {
      Node *temp = head;
      head = head->Next;
      delete temp;
    }
```

Head pointer is reference parameter because we will change the linked list

This will delete the head node if its value matches the function parameter

LINKED LISTS

…

```
   // Handle recursive case
   else
      Delete(head->Next, value);
}
```

Otherwise, recursively delete the value from the rest of the linked list

LINKED LISTS

- **The “head” pointer is a private variable in our list class so we should not make the recursive linked list methods public**
  - Create public linked list methods without head parameters
  - Create private “helper methods” with head parameters

```
bool List::SortedInsert(string value)
{
   return SortedInsert(Head, value);
}

bool List::SortedInsert(LNode* &head, string value)
{
   …
}
```

Private recursive methods are called from public methods

LINKED LISTS

- **Conclusions:**
  - Recursive linked list operations are easy to implement
  - They are often smaller than iterative operations
  - No loops, no previous pointers
  - Recursion is the same speed as iteration

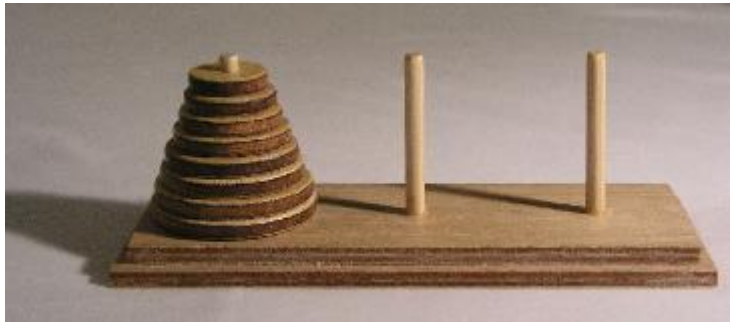
- **Recursive list operations take O(N) steps**
- **Iterative list operations take O(N) steps**

RECURSION

TOWERS OF HANOI

TOWERS OF HANOI

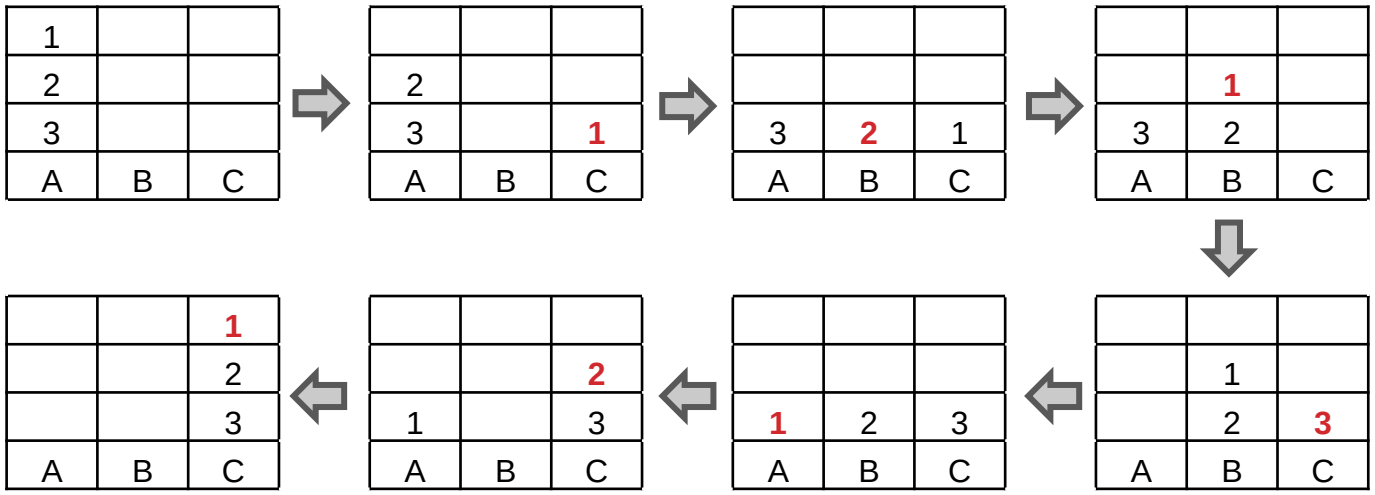
- **The goal of the Tower of Hanoi puzzle is to move all disks from one tower to another tower using the following rules:**
  - Start with 3 towers and N disks on one tower as shown below



  - You can move one disk at a time from one tower to another
  - You can only put a smaller disk on top of a larger disk
  - Finished when all N disks are on another tower in correct order

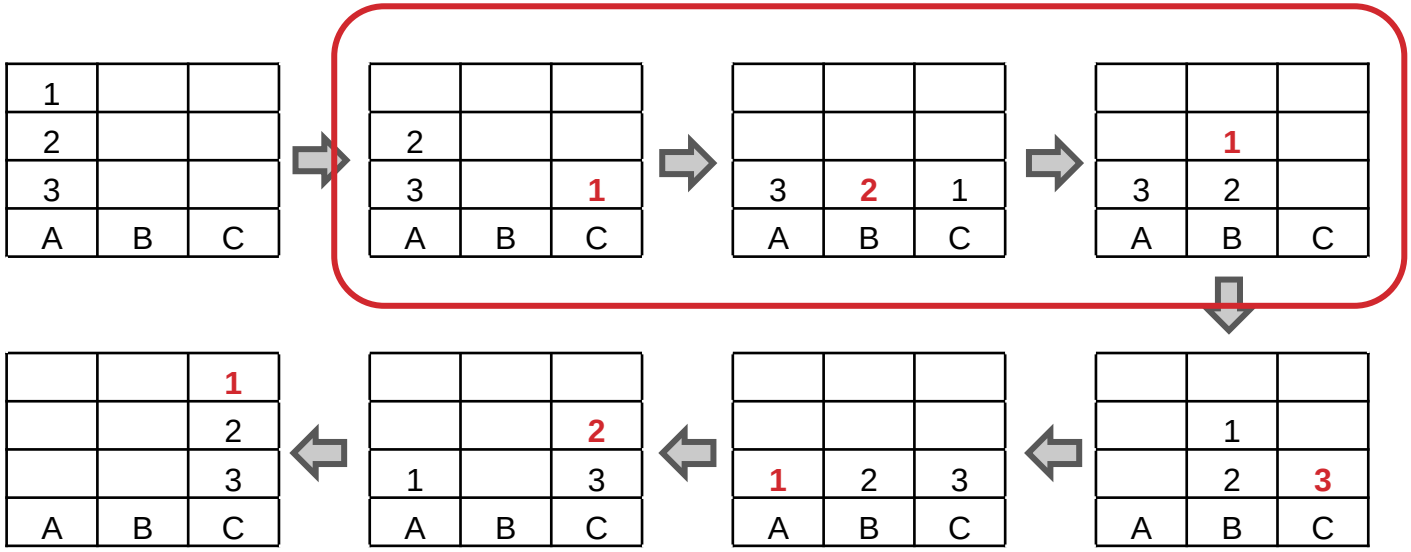
TOWERS OF HANOI

- **Consider example with 3 towers A,B,C and 3 disks 1,2,3**



TOWERS OF HANOI

- **Consider example with 3 towers A,B,C and 3 disks 1,2,3**

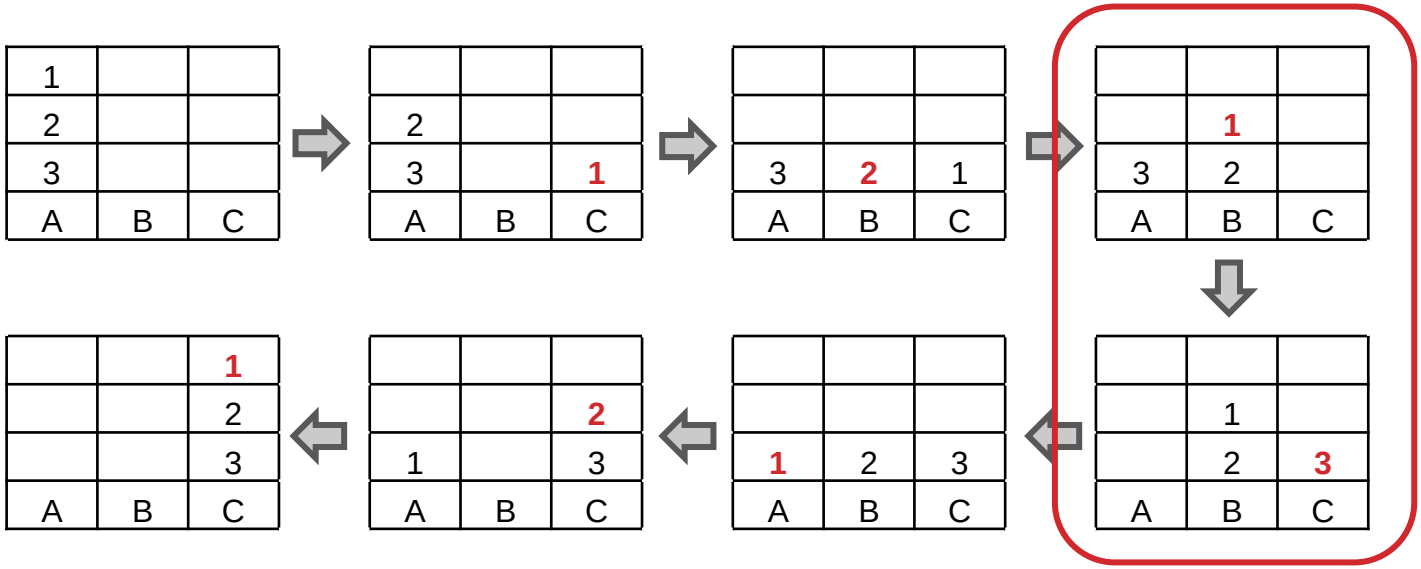


We can solve this puzzle as follows:

- move disks 1 and 2 from tower A to tower B (3 steps)

TOWERS OF HANOI

- **Consider example with 3 towers A,B,C and 3 disks 1,2,3**

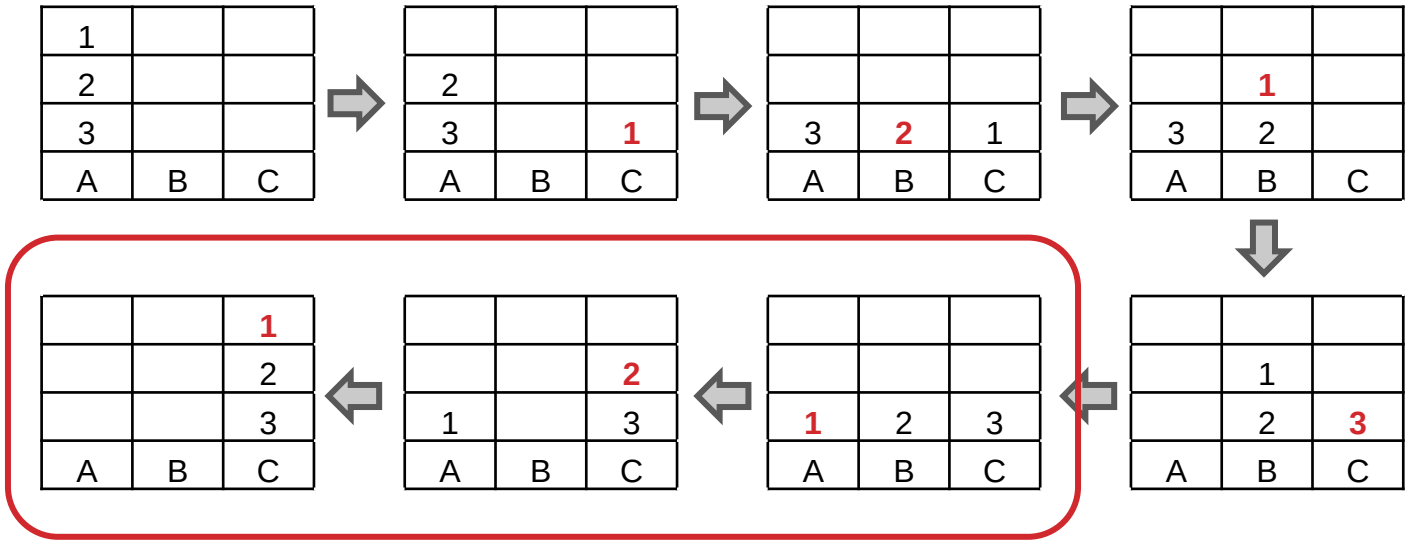


We can solve this puzzle as follows:

- move disks 1 and 2 from tower A to tower B (3 steps)
- move disk 3 from A to tower C (one step)

TOWERS OF HANOI

- **Consider example with 3 towers A,B,C and 3 disks 1,2,3**

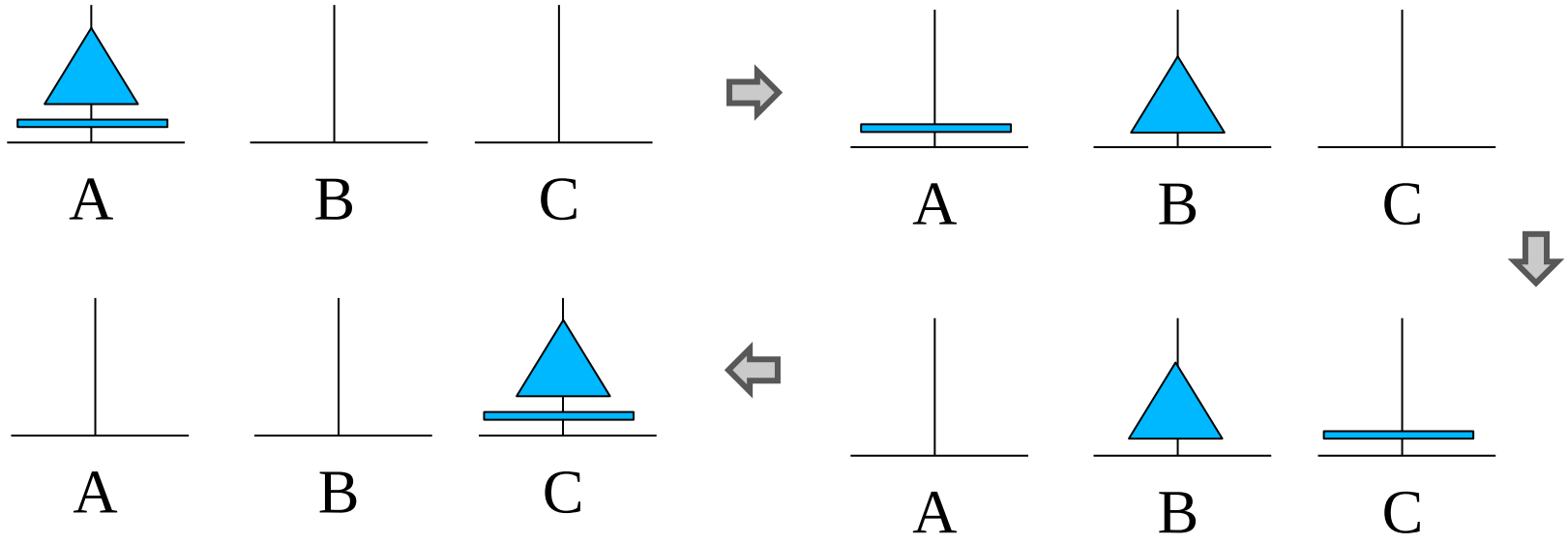


We can solve this puzzle as follows:

- move disks 1 and 2 from tower A to tower B (3 steps)
- move disk 3 from A to tower C (one step)
- move disks 1 and 2 from tower B onto tower C (3 steps)

TOWERS OF HANOI

- **Recursive solution to the puzzle:**
  - Move N-1 disks from A to B
  - Move 1 disk from A to C
  - Move N-1 disks from B to C



TOWERS OF HANOI

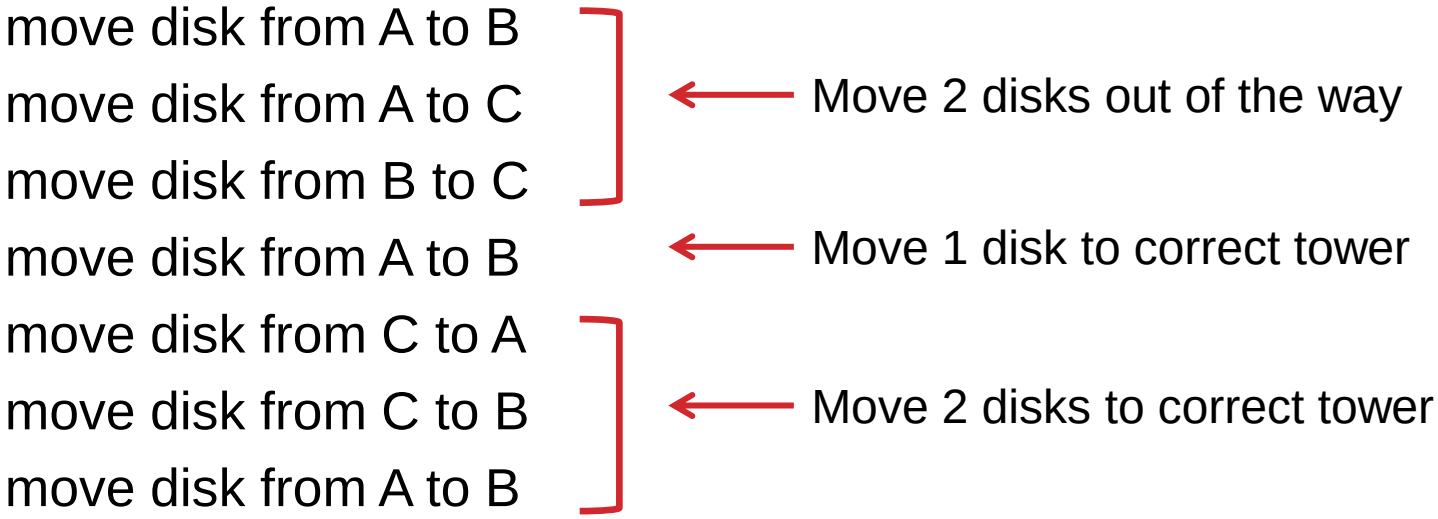
```
void tower(int count, char src, char dest, char extra)
{
    // Handle recursive case
  if (count > 0)
  {
     tower(count - 1, src, extra, dest);
     cout << "move disk from " << src << " to " << dest << endl;
     tower(count - 1, extra, dest, src);
  }
```

We use src, dest and extra for the names of the towers when printing the instructions

These two recursive calls will find the correct sequence of moves to solve the puzzle

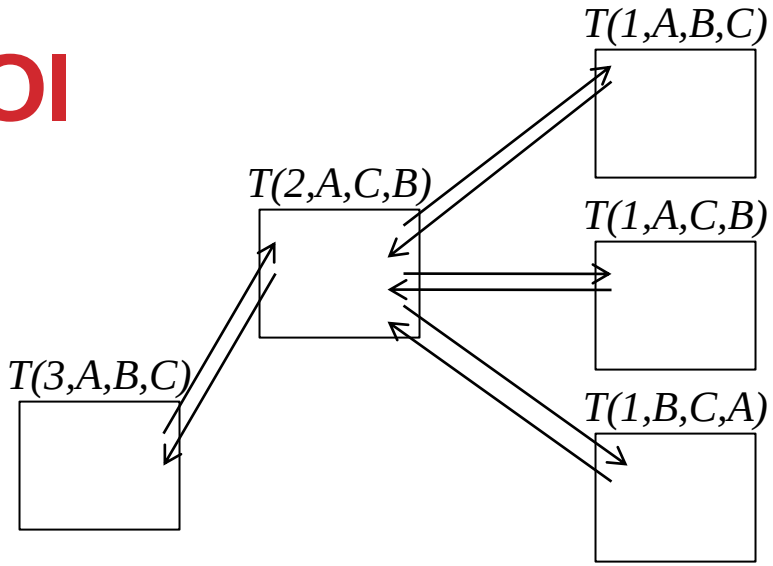
TOWERS OF HANOI

- **Program output for tower(3, 'A', 'B', 'C'):**



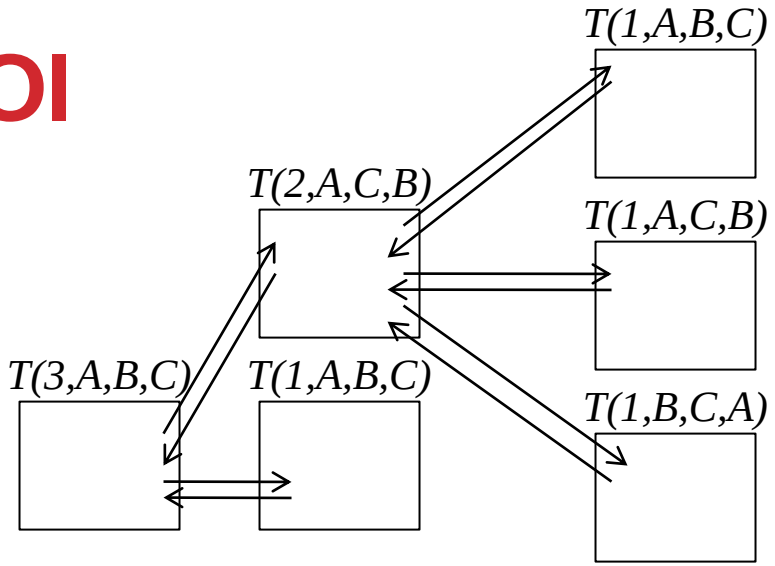
TOWERS OF HANOI

- **Box method trace of tower(3, ‘A’, ‘B’, ‘C’)**
  - Call tower(2, ‘A’, ‘C’, ‘B’)



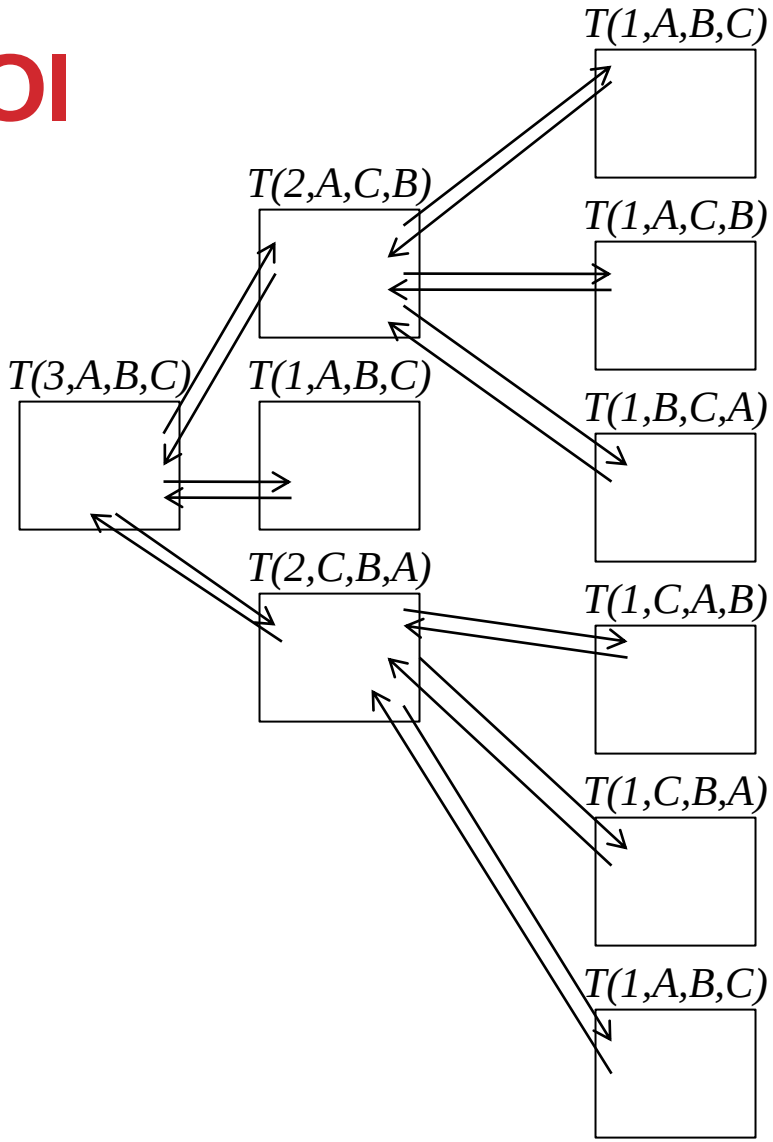
TOWERS OF HANOI

- **Box method trace of tower(3, 'A', 'B', 'C')**
  - Call tower(2, 'A', 'C', 'B')
  - Call tower(1, 'A', 'B', 'C')



TOWERS OF HANOI

- **Box method trace of tower(3, 'A', 'B', 'C')**
  - Call tower(2, 'A', 'C', 'B')
  - Call tower(1, 'A', 'B', 'C')
  - Call tower(2, 'C', 'B', 'A')



TOWERS OF HANOI

- **How many moves are needed to solve the puzzle?**
  - moves(1) = 1
  - moves(2) = 2 * moves(1) + 1 = 3
  - moves(3) = 2 * moves(2) + 1 = 7
  - moves(4) = 2 * moves(3) + 1 = 15
  - …
  - moves(N) = 2 * moves(N-1) + 1 = 2^N-1

- **Solving the towers of Hanoi puzzle recursively is an exponential algorithm O(2^N)**
- **Very complicated iterative solutions also exist for solving this puzzle that are also O(2^N)**

RECURSION

GREATEST COMMON DIVISOR

GREATEST COMMON DIVISOR

- **The greatest common divisor (GDC) of two integers is the largest positive integer that is a factor of both integers**
  - The GCD of integers a and b is equal to d if d is the largest integer where a=d*e and b=d*f for some integers e and f.

- **Example:**
  - The factors of 42 are 1,2,3,6,7,14,21,42
  - The factors of 24 are 1,2,3,4,6,8,12,24
  - The common factors are 1,2,3,6
  - 6 is the largest common factor
  - Hence gcd(42,24) = 6

GREATEST COMMON DIVISOR

- **Euclid's algorithm for calculating the GCD of two integers a and b was published in his "Elements" text in 300BC**

- **His GCD algorithm makes use of repeated subtraction to reduce either a or b until they converge to the GCD**

- **Algorithm:**

  Repeat until a equals b

  If a > b then set a=a-b

  If b > a then set b=b-a

  When equal the GCD is a

GREATEST COMMON DIVISOR

- **Example: gcd(42,24)**

a	b	
42	24	- a larger, subtract b
18	24	- b larger, subtract a
18	6	- a larger, subtract b
12	6	- a larger, subtract b
6	6	- equal, gcd(42,24) = 6

- **We know that a or b will be smaller in each iteration, so this loop is guaranteed to stop at some point**

GREATEST COMMON DIVISOR

```
// Euclid's original algorithm
int gcd1(int a, int b)
{
    while (a != b)
    {
        if (b > a)
            b = b - a;
        else if (a > b)
            a = a - b;
    }
    return a;
}
```

GREATEST COMMON DIVISOR

```
// Improved GCD algorithm
int gcd2(int a, int b)
{
    while (b != 0)
    {
        int t = b;
        b = a % b;
        a = t;
    }
    return a;
}
```

This loop continues until b divides evenly into a

By using modulo here we avoid numerous subtraction steps

GREATEST COMMON DIVISOR

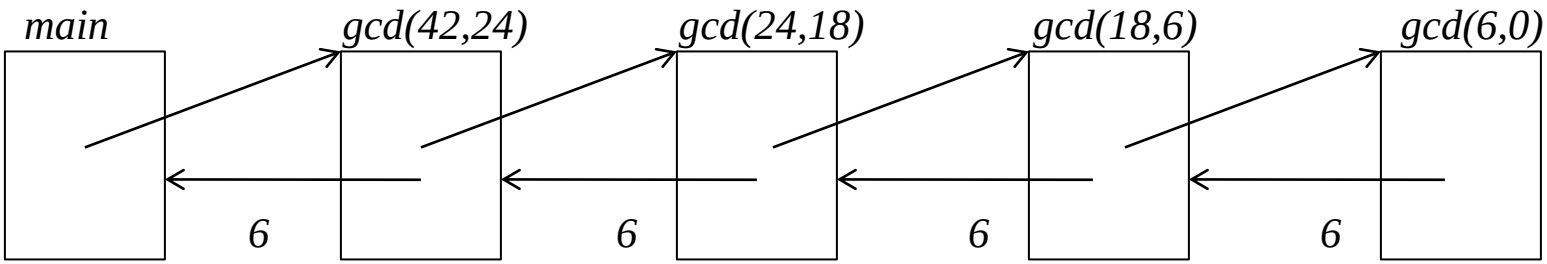
```
// Recursive GCD algorithm
int gcd3(int a, int b)
{
    if (b == 0)
        return a;
    else return
        gcd3(b, a % b);
}
```

This terminating condition is like the previous while loop condition

This recursive call always has smaller a,b values than the original function call

GREATEST COMMON DIVISOR

- **Consider box method trace of gcd(42,24)**



  - What happens if we call gcd(24,42)?
  - What about gcd(42,21)?
  - What about gcd(42,41)?

GREATEST COMMON DIVISOR

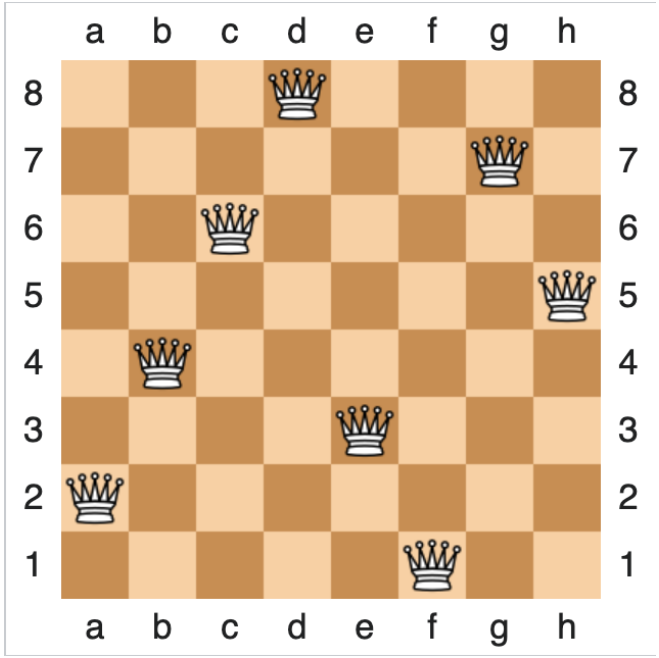
- **Conclusions:**
  - Euclid's GCD algorithm is over 2000 years old and has been extensively used and studied by many famous mathematicians
  - In 1884 Gabriel Lamé was able to prove that the GCD algorithm runs in no more than 5N steps where N is the number of decimal digits in the the number b. Hence this algorithm is O(N)
  - This was the beginning of computational complexity theory (which we study in more detail in CSCE 4323 – Formal Languages and Computability)

RECURSION

N-QUEENS

N-QUEENS

- **The goal of the 8-queens puzzle is to place 8 queens on an 8x8 chess board such that no two queens can threaten each other**



Notice that each row, column and diagonal has only one queen on it

N-QUEENS

- **This 8-queens puzzle was proposed by Max Bessel in 1848 and the first solution was published by Franz Nauck in 1850**
  - There are over 4 billion arrangements of 8 queens on a chess board but only 92 unique solutions
  - If we consider solutions with only one queen per row, there are 8^8 = 16,777,216 combinations
  - If we only consider row or column permutations, there are 8! = 40,320 possible combinations of 8 queens
  - Many famous mathematicians worked on this puzzle and the generalization to N queens on an NxN board
  - In 1972 Edsger Dijkstra proposed an O(N!) recursive depth first backtracking algorithm to solve the N-queens puzzle

N-QUEENS

- **Algorithm:**
  - Assume that columns 1 to k-1 are correctly solved
  - Pick next available “safe” row location on column k
  - Recursively solve next column with a queen in this row
  - If successful return success
  - If not successful, backtrack and remove queen from current row and try next available “safe” row location
  - If no “safe” row location can be found, return failure

By limiting our search to “safe” locations, this algorithm requires far fewer than 8! recursive function calls

N-QUEENS

- **Example:**

Q	*	*	*	*	*	*	*
*	*						
*		*					
*			*				
*				*			
*					*		
*						*	
*							*

Try row 1 in column 1
6 open rows in column 2

Q	*	*	*	*	*	*	*
*	*	*					
*	Q	*	*	*	*	*	*
*	*	*	*				
*	*		*	*			
*	*			*	*		
*	*				*	*	
*	*					*	*

Try row 3 in column 2
4 open rows in column 3

N-QUEENS

- **Example:**

Q	*	*	*	*	*	*	*
*	*	*			*		
*	Q	*	*	*	*	*	*
*	*	*	*				
*	*	Q	*	*	*	*	*
*	*	*	*	*	*		
*	*	*		*	*	*	
*	*	*			*	*	*

Try row 5 in column 3
3 open rows in column 4

Q	*	*	*	*	*	*	*
*	*	*	Q	*	*	*	*
*	Q	*	*	*	*	*	*
*	*	*	*		*		
*	*	Q	*	*	*	*	*
*	*	*	*	*	*		*
*	*	*	*	*	*	*	
*	*	*	*		*	*	*

Try row 2 in column 4
2 open rows in column 5

N-QUEENS

- **Example:**

Q	*	*	*	*	*	*	*
*	*	*	Q	*	*	*	*
*	Q	*	*	*	*	*	*
*	*	*	*	Q	*	*	*
*	*	Q	*	*	*	*	*
*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	*

Try row 4 in column 5
Column 6 full so backtrack

Q	*	*	*	*	*	*	*
*	*	*	Q	*	*	*	*
*	Q	*	*	*	*	*	*
*	*	*	*	X	*		
*	*	Q	*	*	*	*	*
*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	
*	*	*	*	Q	*	*	*

Try row 8 in column 5
Column 6 full so backtrack

N-QUEENS

- **Example:**

Q	*	*	*	*	*	*	*
*	*	*	X		*		
*	Q	*	*	*	*	*	*
*	*	*	*			*	
*	*	Q	*	*	*	*	*
*	*	*	*	*	*		
*	*	*	Q	*	*	*	*
*	*	*	*	*	*	*	*

Remove queen from row 2
Try row 7 in column 4

Q	*	*	*	*	*	*	*
*	*	*	X	Q	*	*	*
*	Q	*	*	*	*	*	*
*	*	*	*	*		*	
*	*	Q	*	*	*	*	*
*	*	*	*	*	*		
*	*	*	Q	*	*	*	*
*	*	*	*	*	*	*	*

Try row 2 in column 5
Continue with remaining columns

N-QUEENS

- **Example:**

Q							
						Q	
				Q			
							Q
	Q						
			Q				
					Q		
		Q					

1st solution found after checking 113 board configurations

		Q					
					Q		
			Q				
	Q						
							Q
				Q			
						Q	
Q							

92nd solution found after checking 1951 board configurations

Notice anything interesting here?

N-QUEENS

- **Chess board symmetry:**
  - We can rotate a chess board by 0, 90, 180, or 270 degrees to get 4 different orientations of pieces
  - We can reflect a board on the x-axis, the y-axis, the y = x diagonal, or the y = -x diagonal to get 4 more orientations
  - For the 8-queens problem there are 12 fundamental solutions to the puzzle
  - We can rotate or reflect one of solutions to obtain all 92 unique solutions to the 8-queens problem
  - One of the fundamental solutions is rotationally symmetric so there are fewer than 12x8 unique solutions

N-QUEENS

```
bool queen(char board[SIZE][SIZE], int col, bool stop)
{
   // Check terminating condition
   if (col >= SIZE)
   {
      print_board(board);
      cout << "Solution: " << ++SOLUTION << endl;
      return stop;
   }
```

We use this flag to stop recursion after one solution or all solutions are found

We print the solution when we have successfully placed queens in all SIZE columns

N-QUEENS

```
// Handle recursive case
else
{
   // Try all possible rows
   for (int row = 0; row < SIZE; row++)
   {
      // Check if location is safe
      if (safe(board, row, col))
      {
         board[row][col] = 'Q';
         print_board(board);
         cout << "Step: " << ++STEP << endl;
```

We check this board location to see if it is safe from other queens

We print the board after each step to illustrate the recursion and backtracking

N-QUEENS

```
        // Check next column
        if (queen(board, col + 1, stop))
            return true;


        // Remove queen
        else
            board[row][col] = '.';
      }
    }
    // Return false if no solution found
    return false;
  }
}
```

The recursive step solves the next column in the chess board

This is where we backtrack and remove the previous queen before trying the next location

N-QUEENS

- **Conclusion:**
  - The number of unique solutions increases rapidly with the size of the chess board and is only known up to n=27
  - In 2021 Michael Simkin at MIT proved that for large n the number of solutions is approximately $S(n) = (0.143n)^n$
  - This is a small fraction of the N! possible arrangements of queens

N	Solutions
8	92
9	352
10	724
11	2,680
12	14,200
13	73,712
14	365,596
15	2,279,184
16	14,772,512
17	95,815,104
18	666,090,624
19	4,968,057,848
20	39,029,188,884

RECURSION

SUMMARY

SUMMARY

- **Recursion is a very powerful problem solving tool**
  - To use recursion, we must solve a problem in terms of a smaller problem of the same type
  - Think like a manager and delegate work to someone else
  - Need to specify the recursive case (smaller problem) and the terminating condition (when to stop)
  - All recursive algorithms can be implemented iteratively but the recursive solution is often simpler and easier to understand (Towers of Hanoi)

SUMMARY

- **Recursive algorithms have very different speeds:**

Speed	Algorithm
O(logN)	Power
O(logN)	Binary Search
O(N)	Factorial
O(N)	Linked Lists
O(N)	Greatest Common Divisor
$O(2^N)$	Fibonacci
$O(2^N)$	Towers of Hanoi
O(N!)	N-Queens