

CS 101: Computer Programming and Utilization

Jan-Apr 2017
Sunita Sarawagi
(cs101@cse.iitb.ac.in)

Lecture 11: Recursive Functions

About These Slides

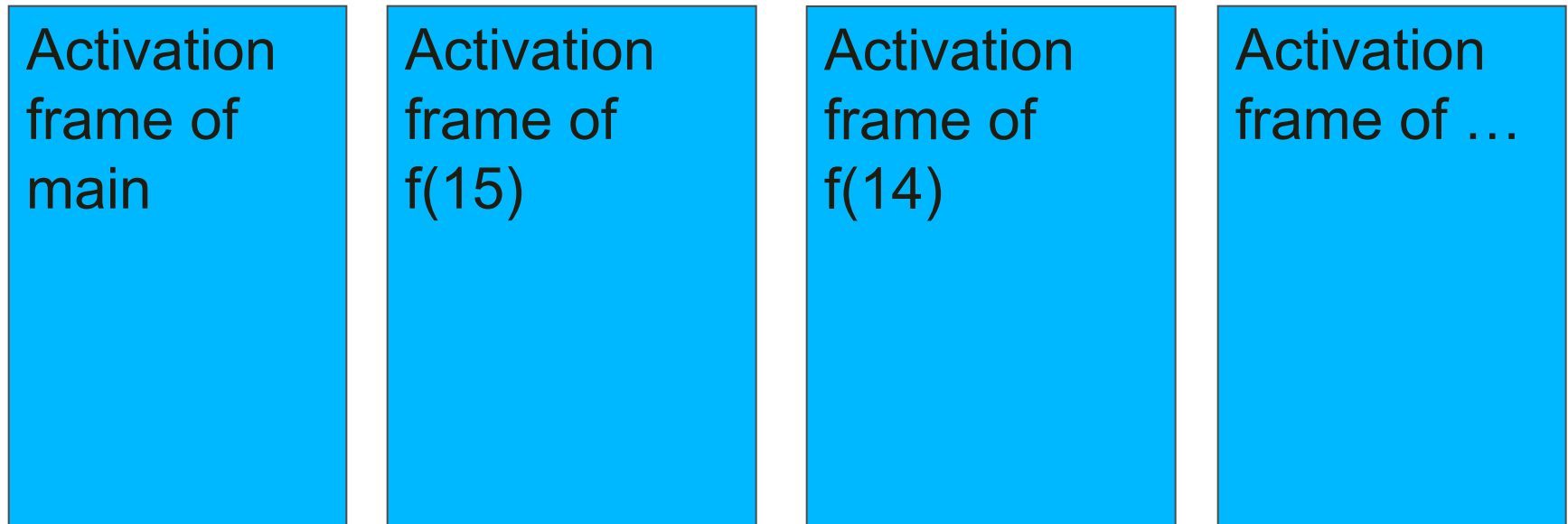
- Based on Chapter 10 and Chapter 16 of the book *An Introduction to Programming Through C++* by Abhiram Ranade (Tata McGraw Hill, 2014)
- Original slides by Abhiram Ranade
 - First update by Varsha Apte
 - Second update by Uday Khedker
 - Third update by Sunita Sarawagi

Can a Function Call Itself?

```
int f(int n){  
    ...  
    int z = f(n-1);  
    ...  
}  
main_program{  
    int z = f(15);  
}
```

- Allowed by execution mechanism
- `main_program` executes, calls `f(15)`
- Activation Frame (AF) created for `f(15)`
- `f` executes, calls `f(14)`
- AF created for `f(14)`
- Continues in this manner, with AFs created for `f(13)`, `f(12)` and so on, endlessly

Activation Frames Keep Getting Created in Stack Memory



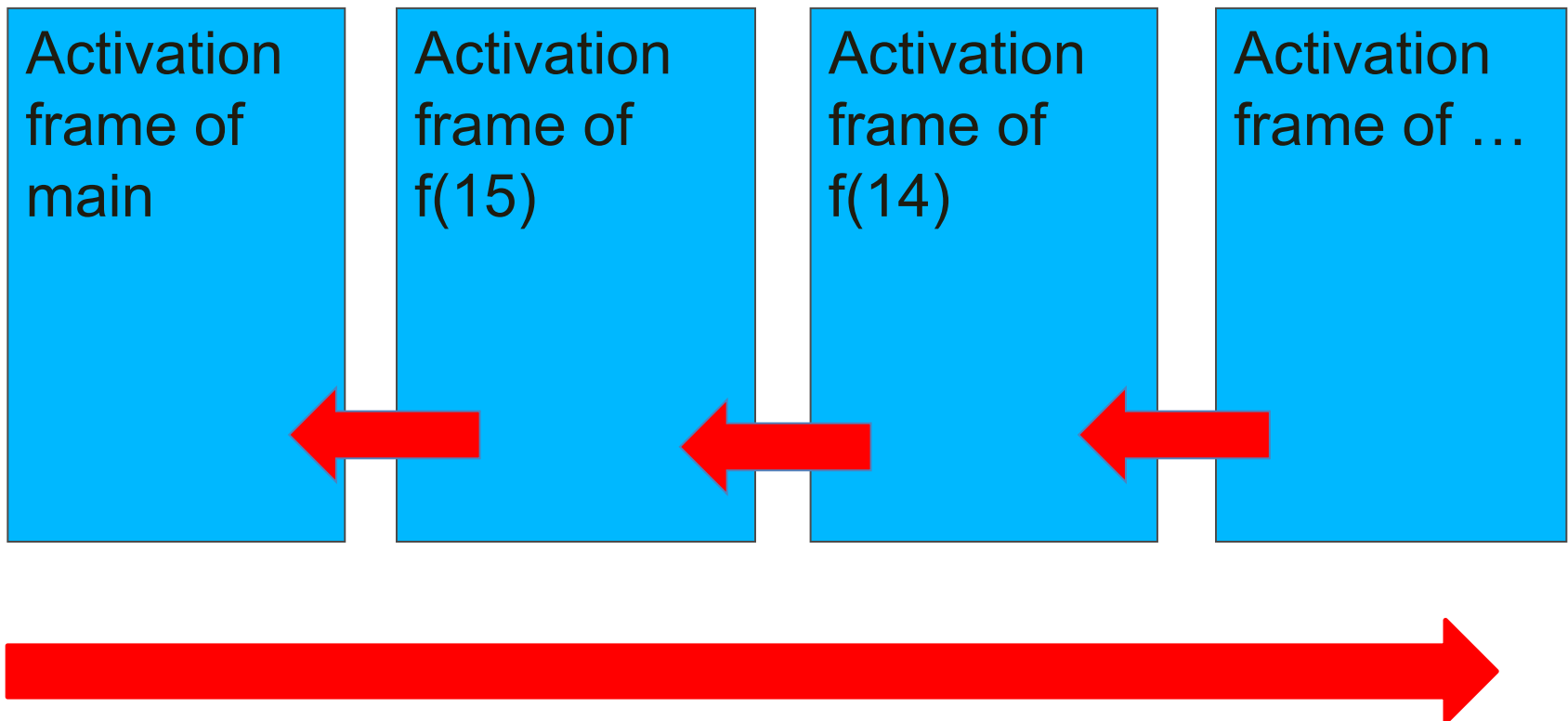
Another Function that Calls Itself

```
int f(int n){
    ...
    if(n > 13)
        z = f(n-1);
    ...
}
main_program{
    int w = f(15);
}
```

- `main_program` executes, calls `f(15)`
- AF created for `f(15)`
- `f(15)` executes, calls `f(14)`
- AF created for `f(14)`
- `f(14)` executes, calls `f(13)`
- AF created for `f(13)`
- `f(13)` executes, check `n > 13` fails. some result returned
- Result received in `f(14)`
- `f(14)` continues and in turn returns result to `f(15)`
- `f(15)` continues, returns result to `main_program`
- `main_program` continues and finished

Activation Frames Keep Getting Created in Stack Memory

and destroyed as the functions exit



Recursion

Function called from its own body

OK if we eventually get to a call which does not call itself

Then that call will return

Previous call will return...

But could it be useful?

Outline

- GCD Algorithm using recursion
- A tree drawing algorithm using recursion

Euclid's Theorem on GCD

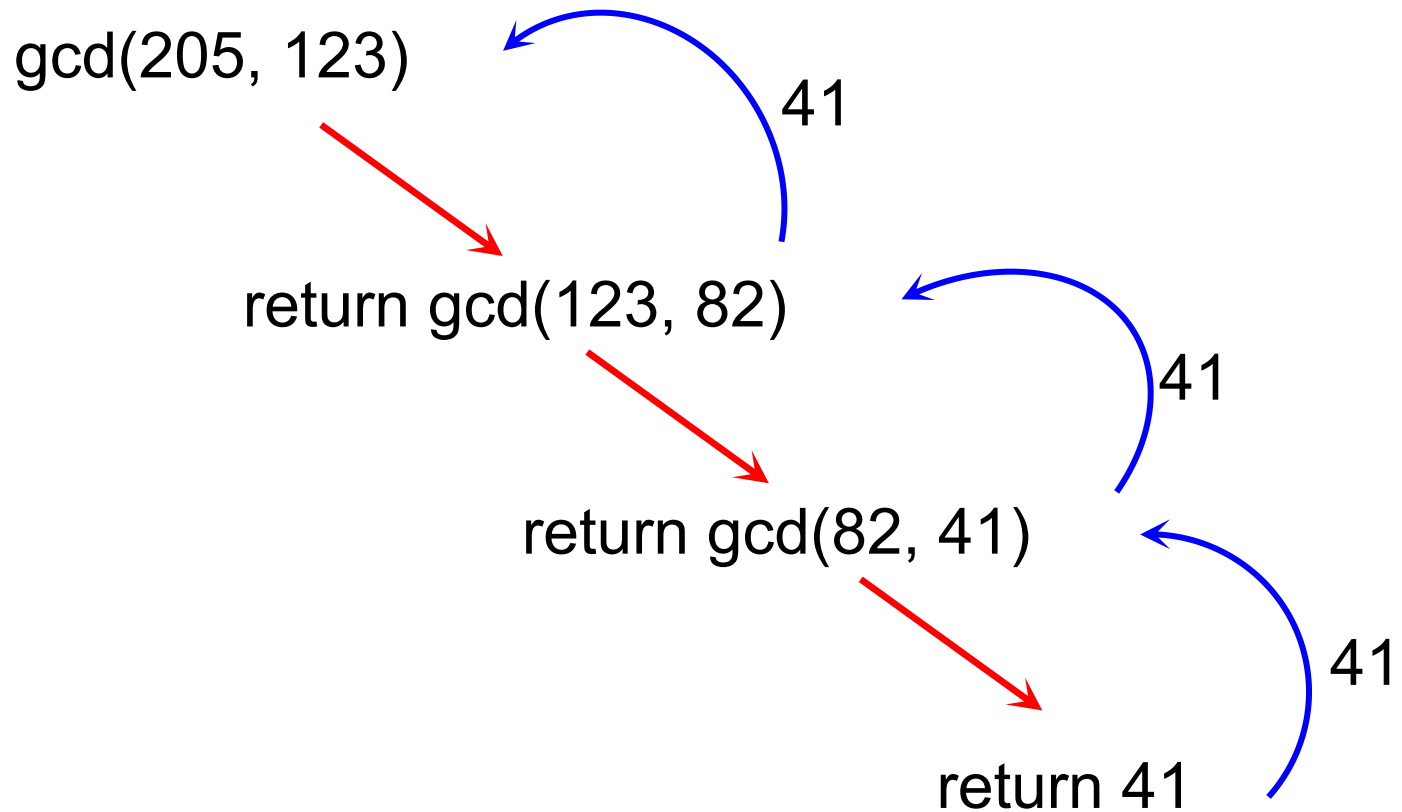
```
//If m % n == 0, then
//      GCD(m, n) = n,
// else GCD(m,n) = GCD(n, m % n)

int gcd(int m, int n){
    if (m % n == 0) return n;
    else return gcd(n, m % n);
}

main_program{
    cout << gcd(205,123) << endl;
}
```

Will this work?

Execution of Recursive gcd



Euclid's Theorem on GCD

```
int gcd(int m, int n){  
    if (m % n == 0) return n;  
    else return gcd(n, m % n);  
}
```

```
main_program{  
    cout  
    << gcd(205,123)  
    << endl;  
}
```

Execute
this

Execute
this

return 41

return 41

return 41

Activation frame
of main created

Activation
frame of gcd
(205, 123)
created

Activation
frame of gcd
(123, 82)
created

Activation
frame of gcd
(82, 41)
created

Recursion Vs. Iteration

- Recursion allows multiple distinct data spaces for different executions of a function body
 - Data spaces are live simultaneously
 - Creation and destruction follows LIFO policy
- Iteration uses a single data space for different executions of a loop body
 - Either the same data space is shared or one data space is destroyed before the next one is created
- Iteration is guaranteed to be simulated by recursion but not vice-versa

Correctness of Recursive gcd

We prove the correctness by induction on j

For a given value of j , $\text{gcd}(i,j)$ correctly
computes $\text{gcd}(i,j)$ for all value of i

We prove this for all values of j by induction

- Base case: $j=1$. $\text{gcd}(i,1)$ returns 1 for all i
Obviously correct
- Inductive hypothesis: Assume the correctness of $\text{gcd}(i,j)$ for a some j
- Inductive step: Show that $\text{gcd}(i,j+1)$ computes the correct value

Correctness of Recursive gcd

Inductive Step: Show that $\text{gcd}(i, j+1)$ computes the correct value, assuming that $\text{gcd}(i, j)$ is correct

- If $j+1$ divides i , then the result is $j+1$
Hence correct
- If $j+1$ does not divide i , then $\text{gcd}(i, j+1)$ returns the result of calling $\text{gcd}(j, i \% (j+1))$
 - $i \% (j+1)$ can at most be equal to j
 - By the inductive hypothesis, $\text{gcd}(j, i \% (j+1))$ computes the correct value
 - Hence $\text{gcd}(i, j+1)$ computes the correct value

Remarks

- The proof of recursive gcd is really the same as that of iterative gcd, but it appears more compact
- This is because in iterative gcd, we had to speak about “initial value of m, n ”, “value at the beginning of the iteration” and so on
- In general recursive algorithms are shorter than corresponding iterative algorithms (if any), and the proof is also more compact, though same in spirit

Factorial Function

- Iterative factorial function

```
int fact(int n) {  
    int res=1;  
    for (int i=1; i<=n; i++)  
        res = res*i;  
    return res;  
}
```

- Recursive factorial function

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


Fibonacci Function

- Iterative fibonacci function:

```
int fib(int n){  
    if (n <= 0) return 0;  
    if (n == 1) return 1;  
    int n_2 = 0, n_1 = 1, result = 0;  
    for (int i = 2; i <= n; i++) {  
        result = n_1 + n_2;  
        n_2 = n_1;  
        n_1 = result;  
    }  
    return result;  
}
```

Fibonacci Function

- Definition:

$$\text{fib}(0) = 0$$

$$\text{fib}(1) = 1$$

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2), \quad n > 1$$

- Recursive fibonacci function:

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

An Important Application of Recursion: Processing Trees

Botanical trees...

Organization Tree

Expression Tree

Search Tree: later

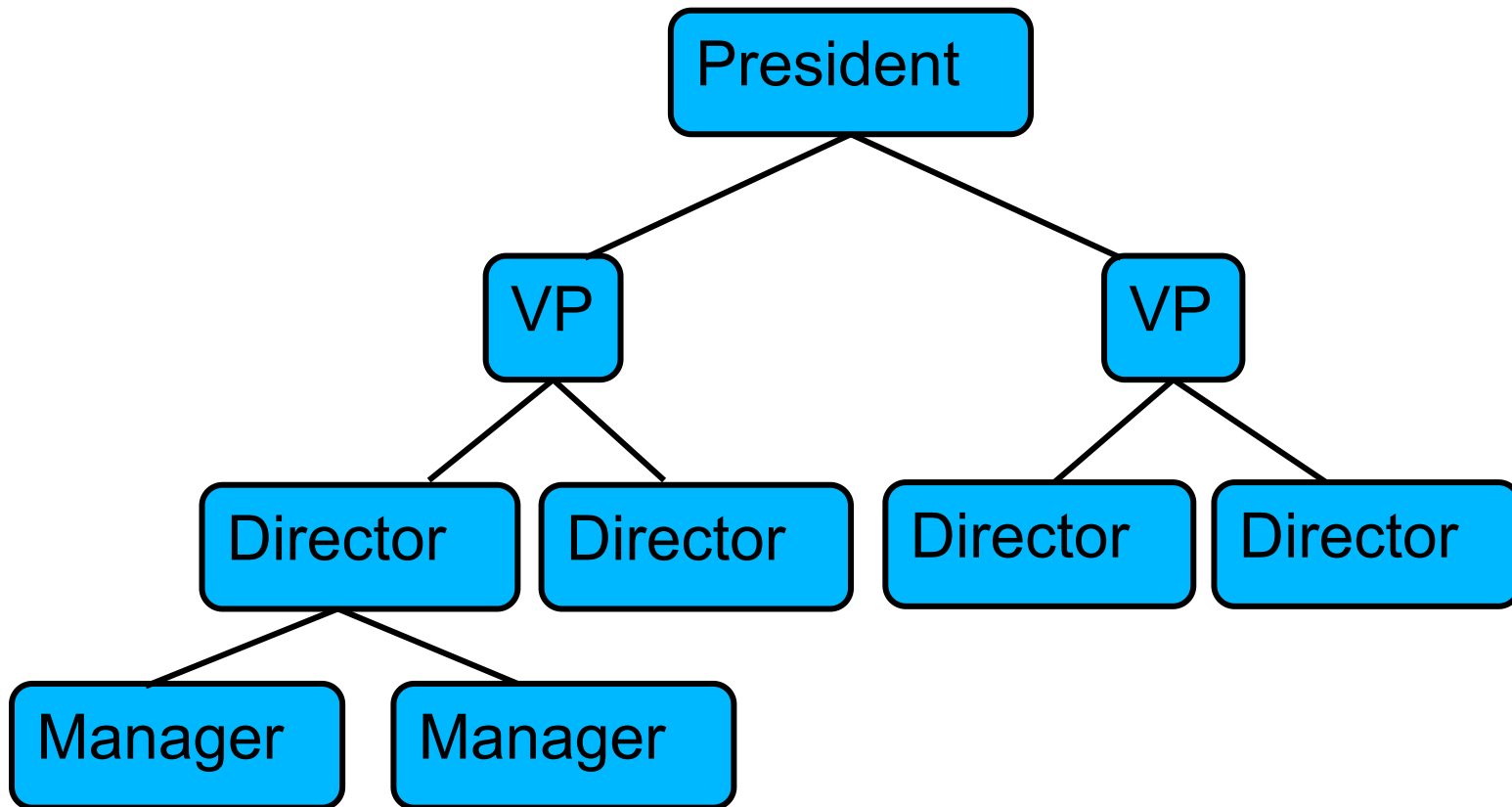
In this chapter we only consider how to draw trees

Must understand the structure of trees

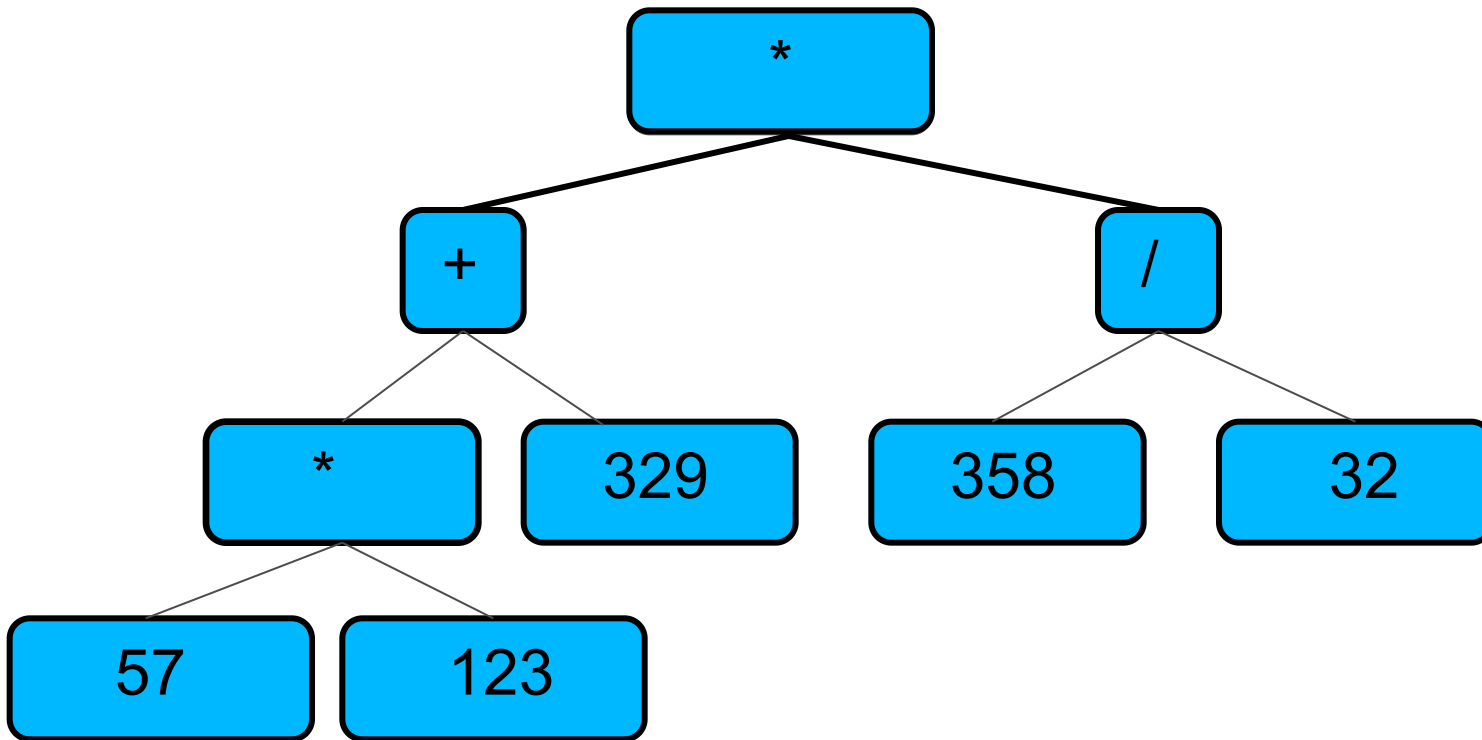
Structure will be relevant to more complex algorithms

Organization Tree

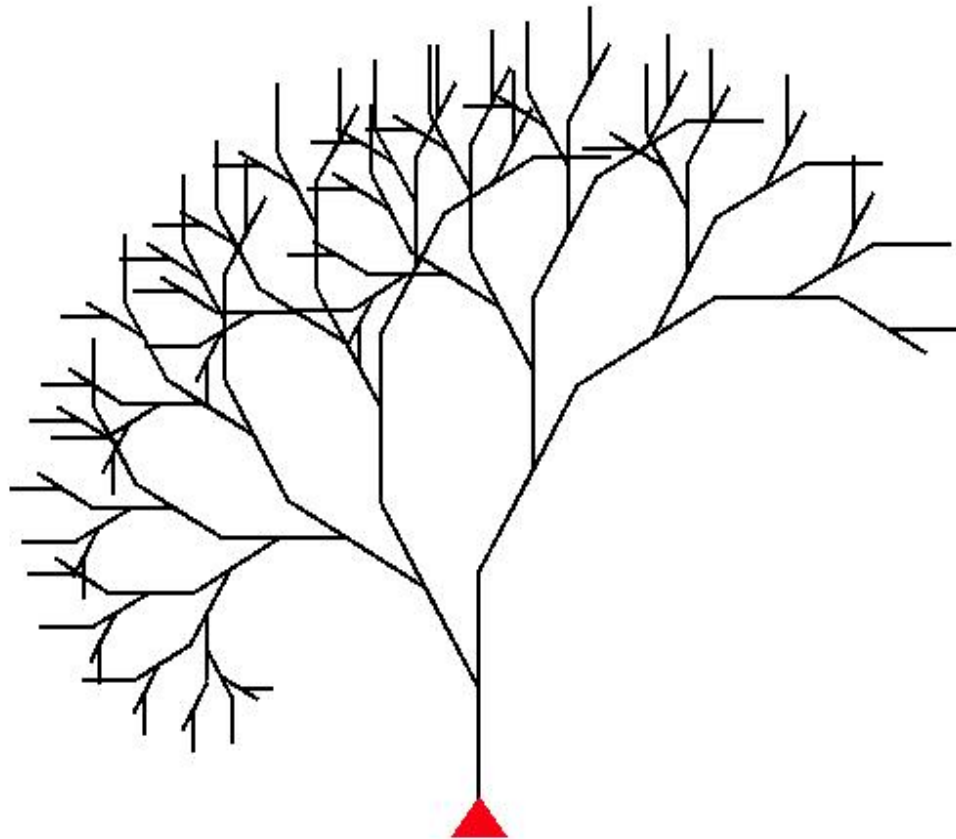
(Typically “grows” Downwards)



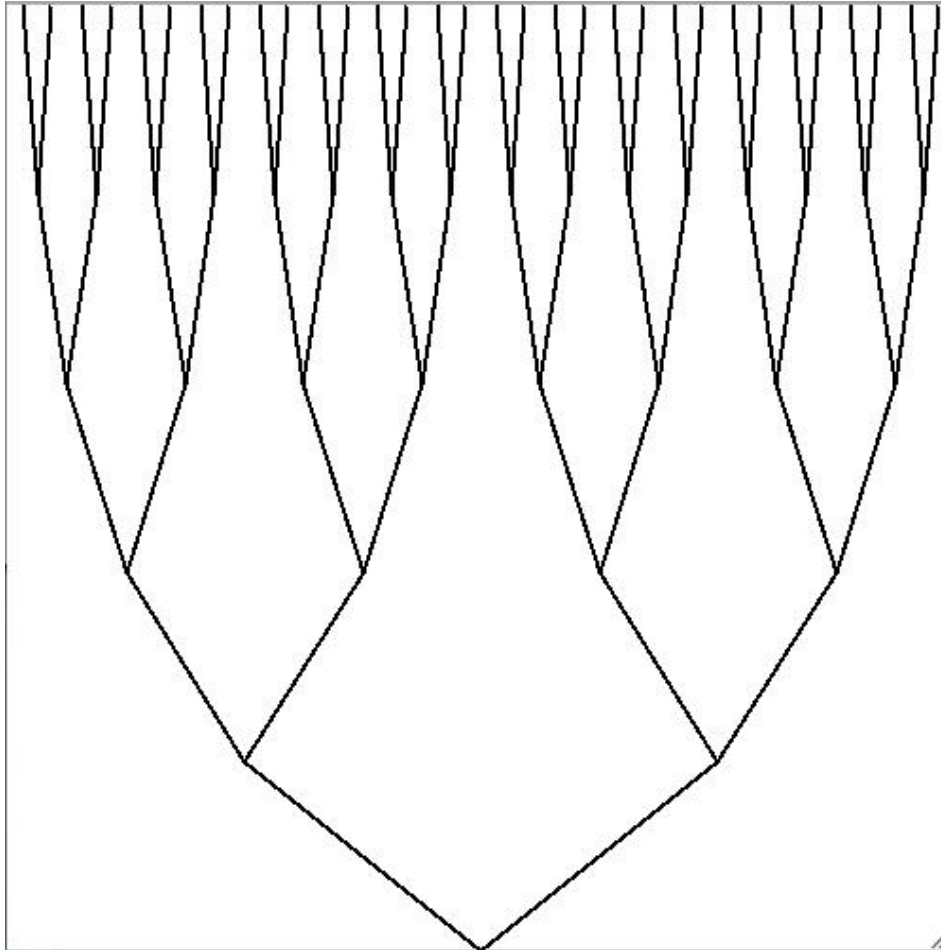
Tree Representing $((57*123)+329)*(358/32)$



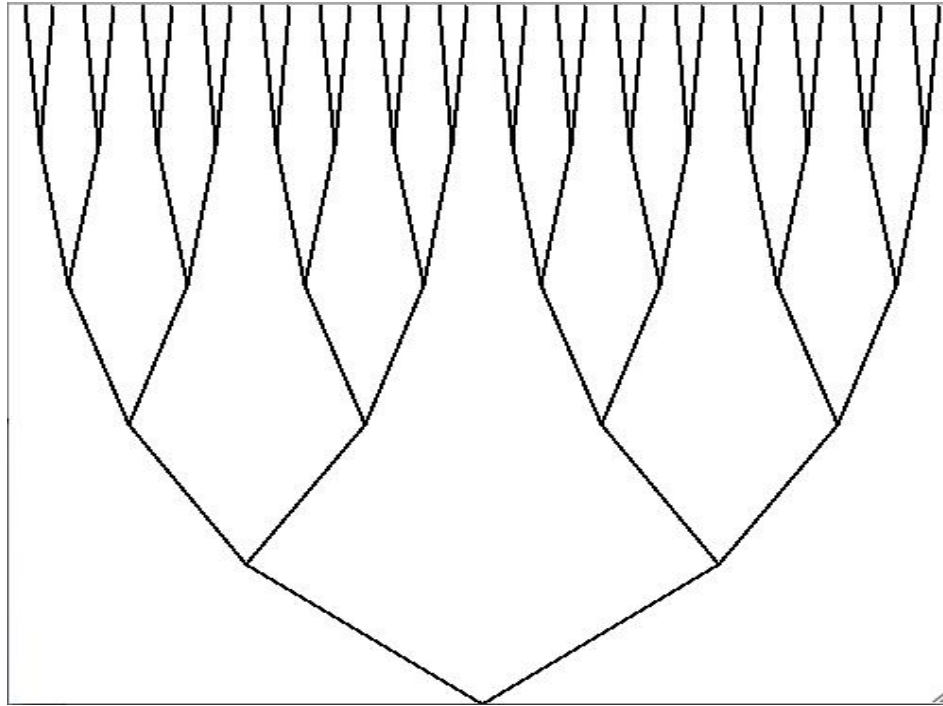
A Botanical Tree Drawn Using the Turtle in Simplecpp



A More Stylized Tree Drawn Using simplecpp



1 Stylized Tree =
2 Small Stylized Trees + V



When a part of an object is of the same type as the whole, the object is said to have a **recursive structure**.

Drawing The Stylized Tree

Parts:

Root

Left branch, Left subtree

Right branch, Right subtree

Number of levels: number of times the tree has branched going from the root to any leaf.

Number of levels in tree shown = 5

Number of levels in subtrees of tree: 4

Drawing The Stylized Tree

General idea:

To draw an L level tree:

if $L > 0$ {

 Draw the left branch, and a Level L-1 on top of it.

 Draw the right branch, and a Level L-1 tree on top of it.

}

We must give the coordinates where the lines are to be drawn

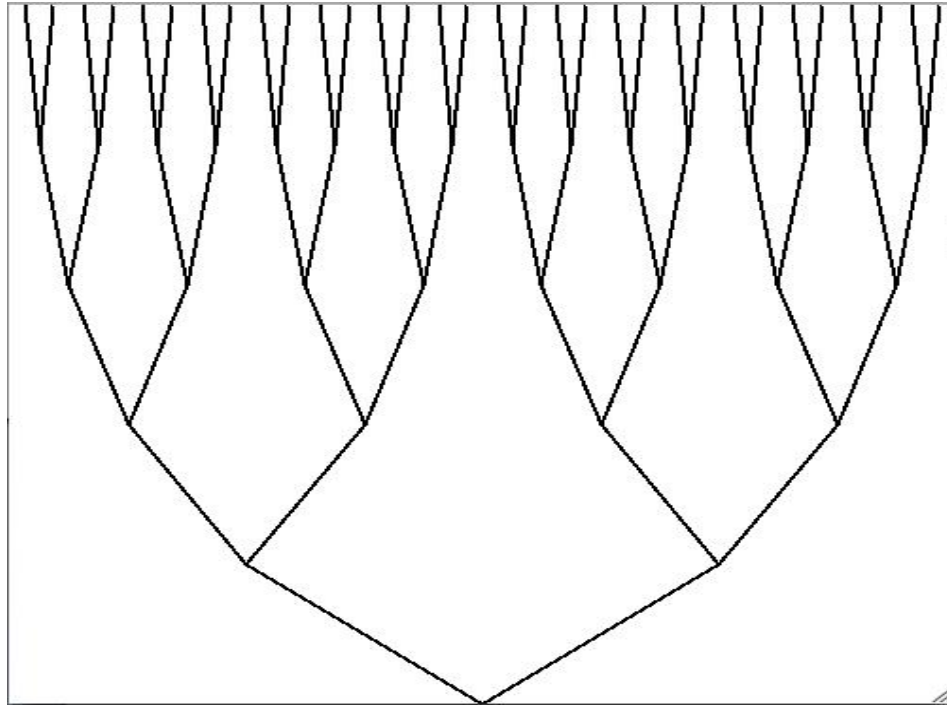
 Say root is to be drawn at (rx,ry)

 Total height of drawing is h.

 Total width of drawing is w.

We should then figure out where the roots of the subtrees will be.

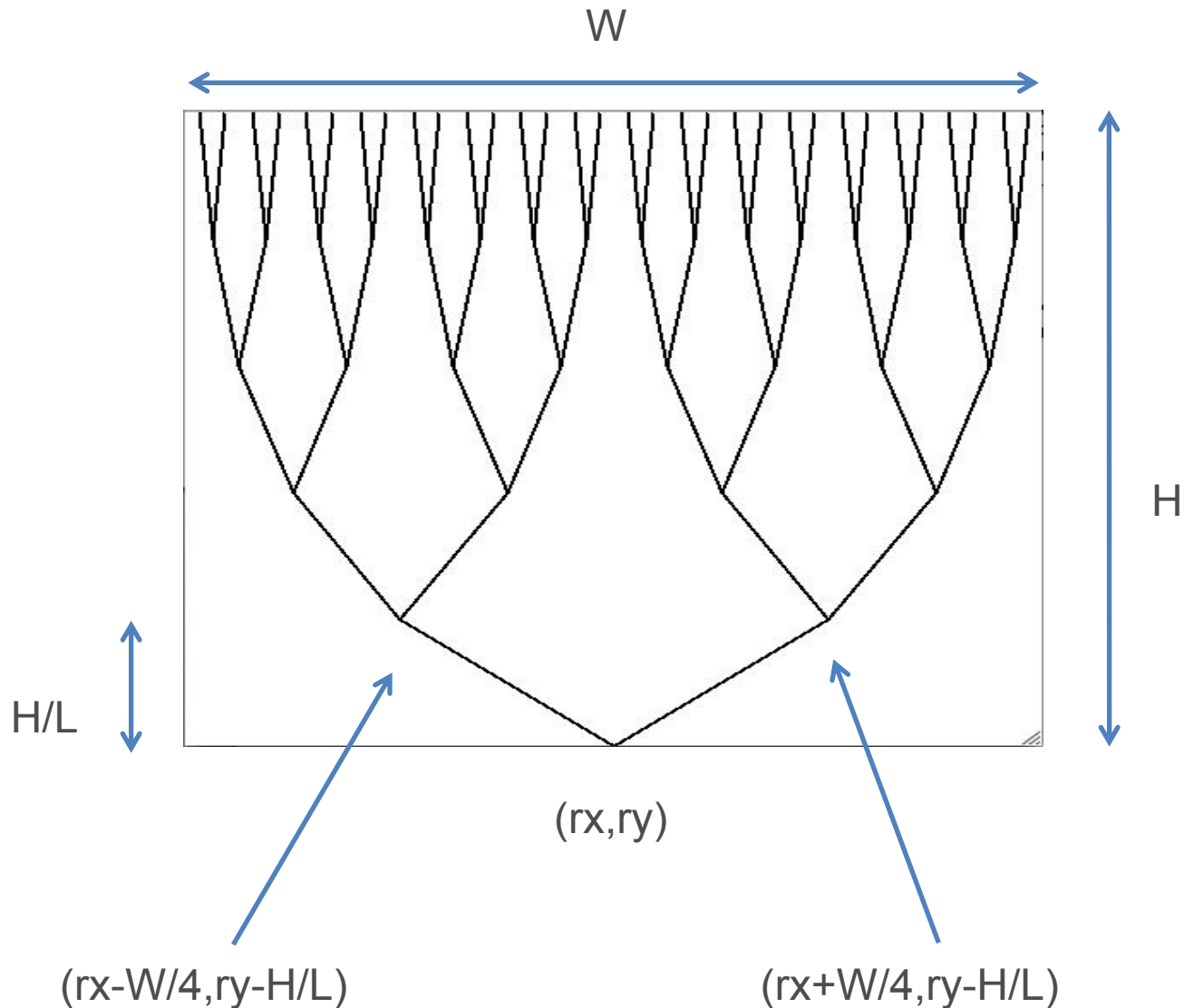
Drawing The Stylized Tree



Basic Primitive:

Drawing a line from (x_1, y_1) to (x_2, y_2)

Drawing The Stylized Tree



Drawing The Stylized Tree

Basic Primitive Required: Drawing a line

- Create a named shape with type Line

```
Line line_name(x1,y1,x2,y2);
```

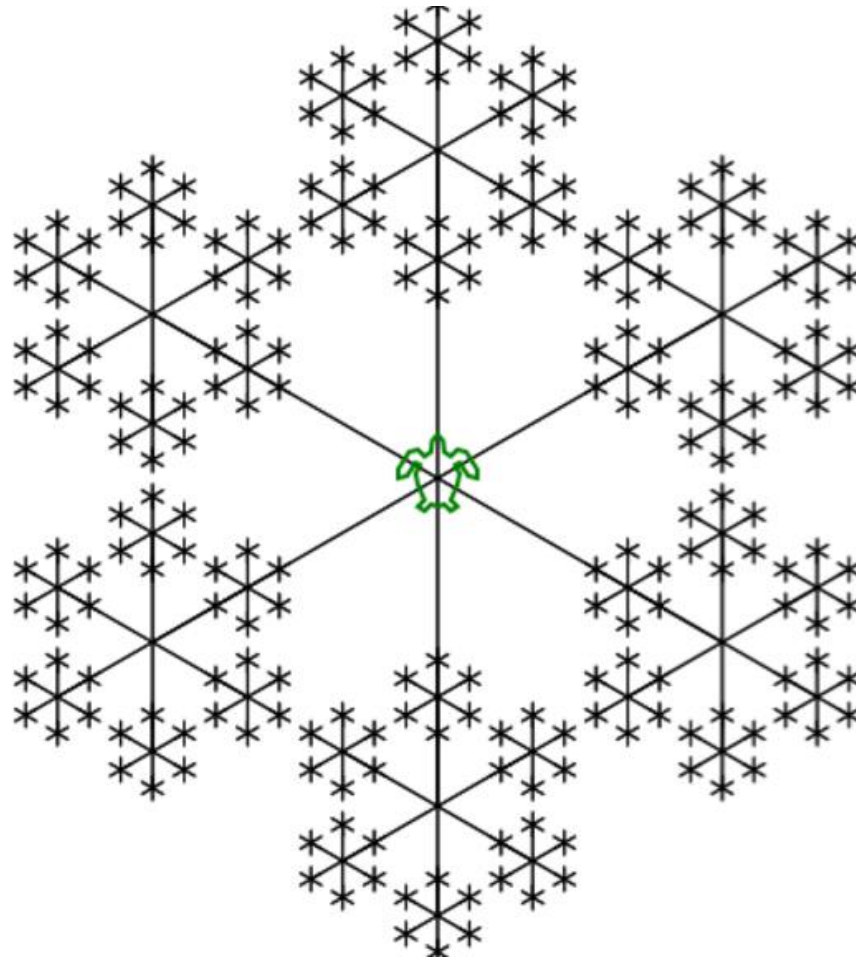
- Draw the shape

```
line_name.imprint();
```

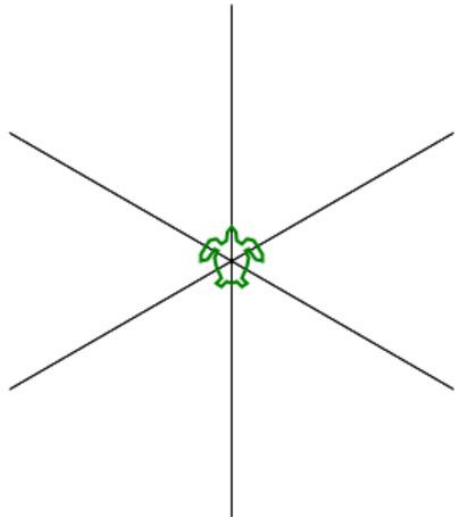
Drawing The Stylized Tree

```
void tree(int L, double rx, double ry,  
          double H, double W) {  
    if(L>0){  
        Line left(rx, ry, rx-W/4, ry-H/L);    // line called left  
        Line right(rx, ry, rx+W/4, ry-H/L); // line called right  
        right.imprint();                        // Draw the line called right  
        left.imprint();                        // Draw the line called left  
        tree(L-1, rx-W/4, ry-H/L, H-H/L, W/2); // left subtree  
        tree(L-1, rx+W/4, ry-H/L, H-H/L, W/2); // right subtree  
    }  
}
```

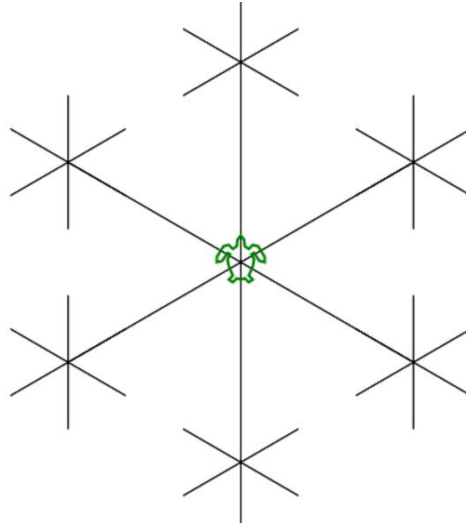
More fun drawings using recursion



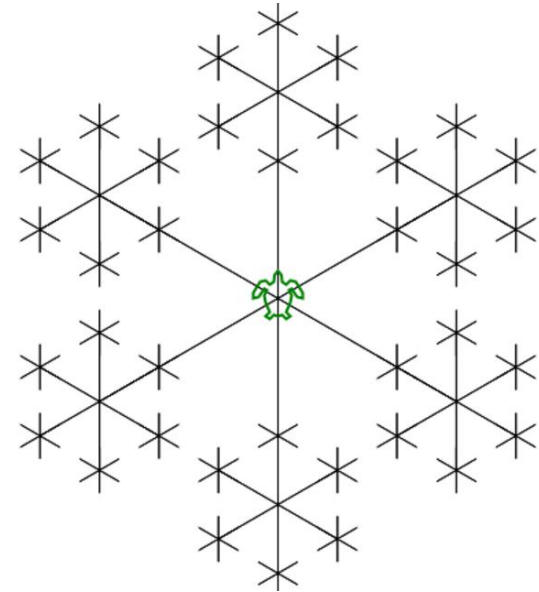
Fractals: self-similar patterns



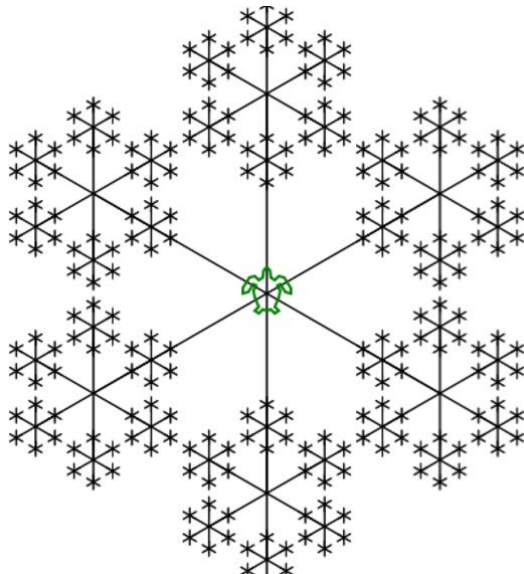
Level 1



Level 2



Level 3



Level 4

Code

```
void drawPattern(double side,int level){
    if(level>3) return;
    repeat(6){
        forward(side);
        drawPattern(side/3,level+1);
        right(180);
        forward(side);
        left(120);
    }
}

main_program{
    turtleSim();
    left(90);
    drawPattern(150,0);
}
```

Arrays and Recursion

- Recursion is very useful for designing algorithms on sequences
 - Sequences will be stored in arrays
- Topics
 - Binary Search
 - Merge Sort

Searching an array

- Input: An array A of length n storing numbers, number x (called “key”)
- Output: `true` if x is present in A , `false` otherwise.
- Natural algorithm: scan through the array and return `true` if found.

```
for(int i=0; i<n; i++){  
    if(A[i] == x) return true;  
}  
return false;
```

- Time consuming: we will scan through entire array if the element is not present, and on the average through half the array if it is present.
- Can we possibly do all this with fewer operations?

Searching a sorted array

- sorted array: (non decreasing order)

$$A[0] \leq A[1] \leq \dots \leq A[n-1]$$

- sorted array: (non increasing order)

$$A[0] \geq A[1] \geq \dots \geq A[n-1]$$

- How do we search in a sorted array (non increasing or non decreasing)?
 - Does the sortedness help in searching?

Searching a non decreasing sorted array

- Assume array is sorted in nondecreasing order.
- Key idea for reducing the number of comparisons: **First compare x with the “middle” element $A[n/2]$ of the array.**
- Suppose $x < A[n/2]$: Because A is sorted, $A[n/2 \dots n-1]$ will also have elements larger than x .
 - x if present will be present only in $A[0 \dots n/2 - 1]$.
 - So in the rest of the algorithm we will only search first half of A .
- Suppose $x \geq A[n/2]$:
 - x if present will be present in $A[n/2 \dots n-1]$
 - x may be present in first half too, but if it is present it will be present in second half too.
 - So in the rest of the algorithm we will only search second half.
- How to search the “halves”?
 - **Recurse!**

Plan

- We will write a function `Bsearch` which will search a region of an array instead of the entire array.
- Region: specified using 2 numbers: starting index S , length of region L
- When $L == 1$, we are searching a length 1 array.
 - So check if that element, $A[S] == x$.
- Otherwise, compare x to the “middle” element of $A[S \dots S+L-1]$
 - Middle element: $A[S + L/2]$
- Algorithm is called “Binary search”, because size of the region to be searched gets halved.

The code

```
bool Bsearch(int A[], int S, int L, int x)
// Search in A[S..S+L-1]
{
    if(L == 1) return A[S] == x;
    int H = L/2;
    if (x == A[S+H]) return true;
    if(x < A[S+H]) return Bsearch(A, S,    H,    x);
    else          return Bsearch(A, S+H, L-H, x);
}

int main(){
    int A[8]={-1, 2, 2, 4, 10, 12, 30, 30};
    cout << Bsearch(A,0,8,11) << endl;
    // searches for 11.
}
```

How does the algorithm execute?

- $A = \{-1, 2, 2, 4, 10, 12, 30, 30\}$
- First call: $\text{Bsearch}(A, 0, 8, 11)$
 - comparison: $11 < A[0+8/2] = A[4] = 10$
 - Is false.
- Second call: $\text{Bsearch}(A, 4, 4, 11)$
 - comparison: $11 < A[4+4/2] = A[6] = 30$
 - Is true.
- Third call: $\text{Bsearch}(A, 4, 2, 11)$
 - comparison: $11 < A[4+2/2] = A[5] = 12$
 - Is true.
- Fourth call: $\text{Bsearch}(A, 4, 1, 11)$
 - Base case. Return $11 == A[4]$. So false.

Proof of termination

- Will the algorithm always terminate?
- Parameter L always decreases, unless it is already 1.
 - Next call is with $L/2$ or $L-L/2$.
 - So it will eventually become 1.
 - So the base case will be reached.

Remarks

- If you are likely to search an array frequently, it is useful to first sort it. The time to sort the array will be compensated by the time saved in subsequent searches.
 - How do you sort an array in the first place?
- Next.
- Binary search can be written without recursion. Exercise.

Sorting

- Chapter 14 discusses a simple algorithm for sorting called Selection sort.
- Selection can require n^2 comparisons to sort n keys.
- Algorithms requiring fewer comparisons are known: $n \log n$ comparisons.
- One such algorithm is Merge sort.

Mergesort idea

To sort a long sequence:

- Break up the sequence into two small sequences.
- Sort each small sequence. (Recurse!)
- Somehow “merge” the sorted sequences into a single long sequence.
- Hope: “merging” sorted sequences is easier than sorting the large sequence.
- Our hope is correct, as we will see soon!

Example

- Suppose we want to sort the sequence
 - 50, 29, 87, 23, 25, 7, 64
- Break it into two sequences.
 - 50, 29, 87, 23 and 25, 7, 64.
- Sort both
 - We get 23, 29, 50, 87 and 7, 25, 64.
- Merge
 - Goal is to get 7, 23, 25, 29, 50, 64, 87.

Merge sort

```
void mergesort(int S[], int n){  
    // Sorts sequence S of length n.  
    if(n==1) return;  
    int U[n/2], V[n-n/2];  
    for(int i=0; i<n/2; i++) U[i]=S[i];  
    for(int i=0; i<n-n/2; i++) V[i]=S[i+n/2];  
    mergesort(U,n/2);  
    mergesort(V,n-n/2);  
    // "Merge" sorted U, V into S.  
    merge(U, n/2, V, n-n/2, S, n);  
}
```

Merging two sorted sequences

- Think of a sorted sequence as a row of students, ordered shortest to tallest.
- We are given two such rows, U , V .
- We want to move students from both rows into a new row S , but it should still be in shortest to tallest order.

Merging

U: 23, 29, 50, 87.

V: 7, 25, 64.

S:

- The smallest overall must move into S.
Smallest overall can be smaller of smallest in U and smallest in V.
- So after movement we get:

U: 23, 29, 50, 87.

V: -, 25, 64.

S: 7.

What do we do next?

U: 23, 29, 50, 87.

V: -, 25, 64.

S: 7.

- Now we need to move the second smallest into S.
- Second smallest:
 - smallest in U,V after smallest has moved out.
 - smaller of the students currently at the head of U, V.
- So we get:

U: -, 29, 50, 87.

V: -, 25, 64.

S: 7, 23.

General strategy

- While both U, V contain a student:
 - Move smallest from those at the head of U,V to the end of S.
- If only U contains students: move all to end of S.
- If only V contains students: move all to end of S.
- uf: index denoting which element of U is currently at the front.
 - $U[0..uf-1]$ have moved out.
- vf: similarly for V.
- sb: index denoting where next element should move into S next (sb: back of S)
 - $S[0..sb-1]$ contain elements that have moved in earlier.

Merging two sequences

```
merge(int U[], int p, int V[], int q, int S[], int n){
    for(int uf=0, vf=0, sb=0;
        sb < p + q; // while all elements havent moved
        sb++){
        if(uf<p && vf<q){ // both U,V are non empty
            if(U[uf] < V[vf]){
                S[sb] = U[uf]; uf++;
            } else{
                S[sb] = V[vf]; vf++;
            }
        } else if(uf < p){ // only U is non empty
            S[sb] = U[uf]; uf++;
        } else { // only V is non empty
            S[sb] = V[vf]; vf++;
        }
    }
}
```

Concluding Remarks

- Recursion allows many programs to be expressed very compactly
- The idea that the solution of a large problem can be obtained from the solution of a similar problem of the same type, is very powerful
- Euclid probably used this idea to discover his GCD algorithm
- More examples in the book