Data Structures are the programmatic way of storing data so that data can be used efficiently. Almost every enterprise application uses various types of data structures in one or the other way.

Why to Learn Data Structure and Algorithms?

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

- **Data Search** − Consider an inventory of 1 million(10^6) items of a store. If the application is to search an item, it has to search an item in 1 million(10^6) items every time slowing down the search. As data grows, search will become slower.
- **Processor speed** − Processor speed although being very high, falls limited if the data grows to billion records.
- **Multiple requests** − As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

To solve the above-mentioned problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

Applications of Data Structure and Algorithms

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms −

- **Search** − Algorithm to search an item in a data structure.

- **Sort** − Algorithm to sort items in a certain order.
- **Insert** − Algorithm to insert item in a data structure.
- **Update** − Algorithm to update an existing item in a data structure.
- **Delete** − Algorithm to delete an existing item from a data structure.

The following computer problems can be solved using Data Structures −

- Fibonacci number series
- Knapsack problem
- Tower of Hanoi
- All pair shortest path by Floyd-Warshall
- Shortest path by Dijkstra
- Project scheduling

Data Structure is a systematic way to organize data in order to use it efficiently. Following terms are the foundation terms of a data structure.

- **Interface** − Each data structure has an interface. Interface represents the set of operations that a data structure supports. An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations.
- **Implementation** − Implementation provides the internal representation of a data structure. Implementation also provides the definition of the algorithms used in the operations of the data structure.

Characteristics of a Data Structure

- **Correctness** − Data structure implementation should implement its interface correctly.
- **Time Complexity** − Running time or the execution time of operations of data structure must be as small as possible.

- **Space Complexity** − Memory usage of a data structure operation should be as little as possible.

Need for Data Structure

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

- **Data Search** − Consider an inventory of 1 million(10^6) items of a store. If the application is to search an item, it has to search an item in 1 million(10^6) items every time slowing down the search. As data grows, search will become slower.
- **Processor speed** − Processor speed although being very high, falls limited if the data grows to billion records.
- **Multiple requests** − As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

To solve the above-mentioned problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

Execution Time Cases

There are three cases which are usually used to compare various data structure's execution time in a relative manner.

- **Worst Case** − This is the scenario where a particular data structure operation takes maximum time it can take. If an operation's worst case time is f(n) then this operation will not take more than f(n) time where f(n) represents function of n.
- **Average Case** − This is the scenario depicting the average execution time of an operation of a data structure. If an operation takes f(n) time in execution, then m operations will take mf(n) time.

- **Best Case** − This is the scenario depicting the least possible execution time of an operation of a data structure. If an operation takes f(n) time in execution, then the actual operation may take time as the random number which would be maximum as f(n).

Basic Terminology

- **Data** − Data are values or set of values.
- **Data Item** − Data item refers to single unit of values.
- **Group Items** − Data items that are divided into sub items are called as Group Items.
- **Elementary Items** − Data items that cannot be divided are called as Elementary Items.
- **Attribute and Entity** − An entity is that which contains certain attributes or properties, which may be assigned values.
- **Entity Set** − Entities of similar attributes form an entity set.
- **Field** − Field is a single elementary unit of information representing an attribute of an entity.
- **Record** − Record is a collection of field values of a given entity.
- **File** − File is a collection of records of the entities in a given entity set.

Introduction
to
Data Structures

Definition

- Data structure is representation of the logical relationship existing between individual elements of data.
- In other words, a data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

Introduction

- Data structure affects the design of both structural & functional aspects of a program.

Program=algorithm + Data Structure

- You know that a algorithm is a step by step procedure to solve a particular function.

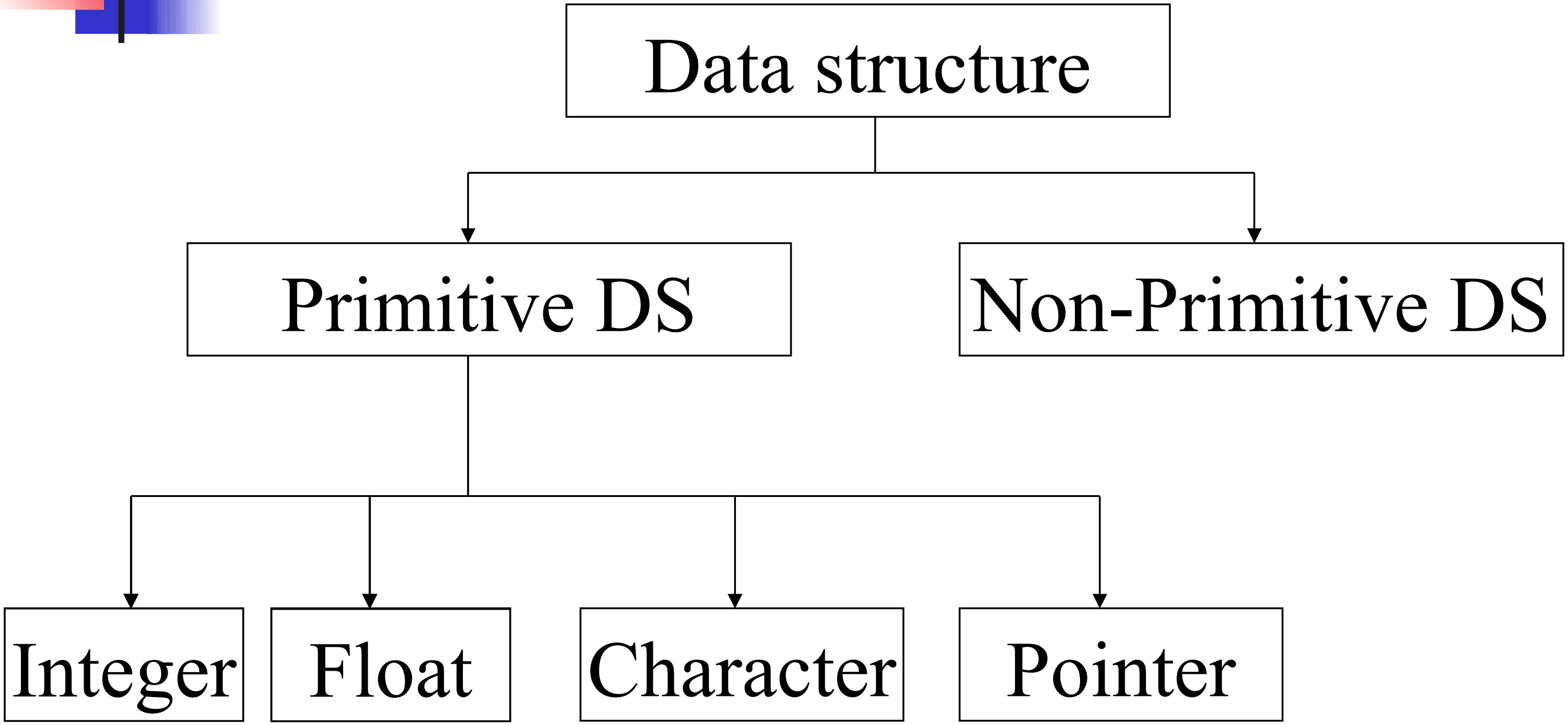
Introduction

- That means, algorithm is a set of instruction written to carry out certain tasks & the data structure is the way of organizing the data with their logical relationship retained.
- To develop a program of an algorithm, we should select an appropriate data structure for that algorithm.
- Therefore algorithm and its associated data structures from a program.

Classification of Data Structure

- Data structure are normally divided into two broad categories:
    - Primitive Data Structure
    - Non-Primitive Data Structure

Classification of Data Structure



Data structure

Primitive DS

Non-Primitive DS

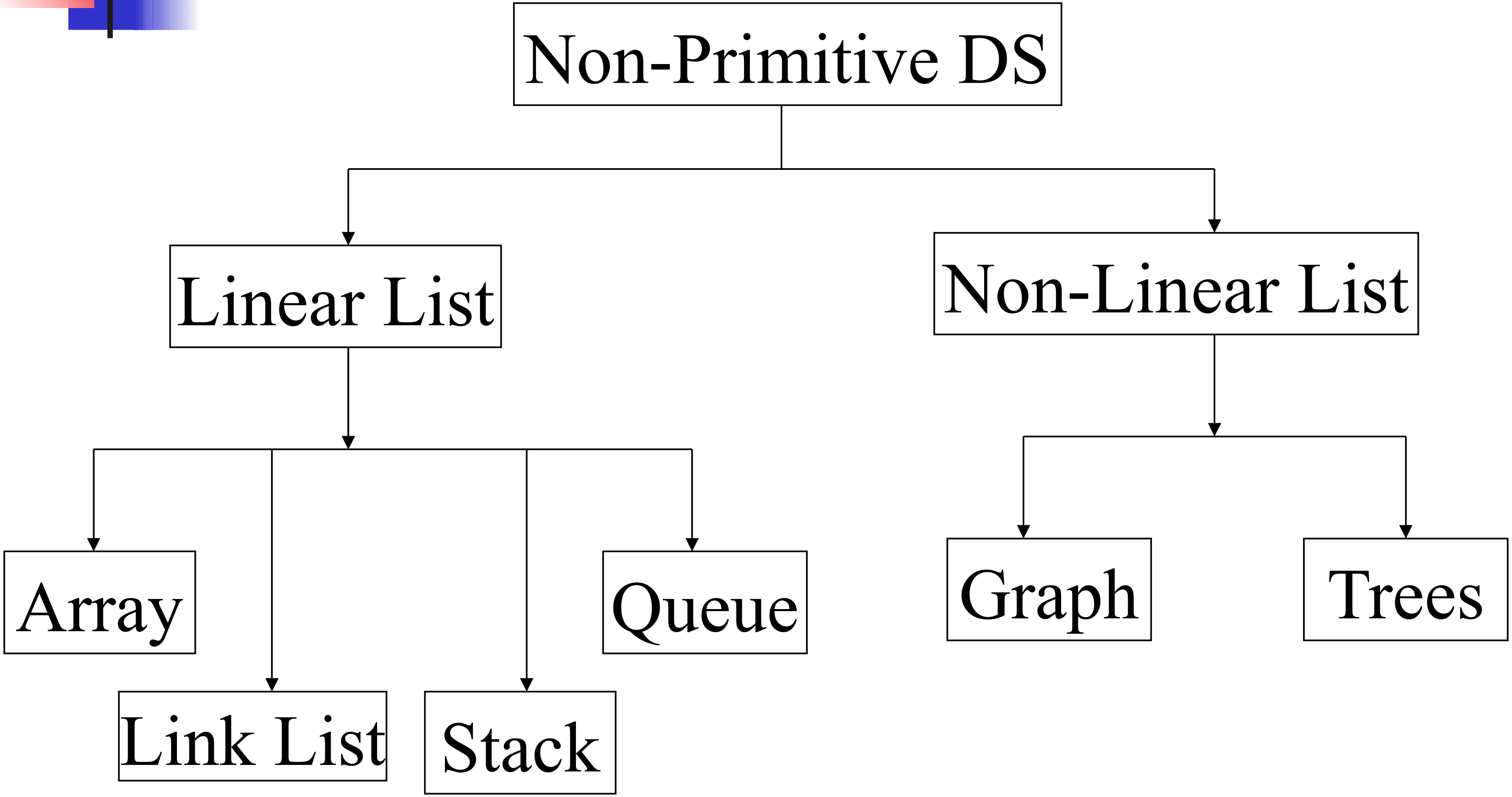
Integer

Float

Character

Pointer

Classification of Data Structure



Primitive Data Structure

- There are basic structures and directly operated upon by the machine instructions.
- In general, there are different representation on different computers.
- Integer, Floating-point number, Character constants, string constants, pointers etc, fall in this category.

Non-Primitive Data Structure

- There are more sophisticated data structures.
- These are derived from the primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.

Non-Primitive Data Structure

- Lists, Stack, Queue, Tree, Graph are example of non-primitive data structures.
- The design of an efficient data structure must take operations to be performed on the data structure.

Non-Primitive Data Structure

- The most commonly used operation on data structure are broadly categorized into following types:
  - Create
  - Selection
  - Updating
  - Searching
  - Sorting
  - Merging
  - Destroy or Delete

Different between them

- A primitive data structure is generally a basic structure that is usually built into the language, such as an integer, a float.
- A non-primitive data structure is built out of primitive data structures linked together in meaningful ways, such as a or a linked-list, binary search tree, AVL Tree, graph etc.

Description of various Data Structures : Arrays

- An array is defined as a set of finite number of homogeneous elements or same data items.
- It means an array can contain one type of data only, either all integer, all float-point number or all character.

Arrays

- Simply, declaration of array is as follows:
int arr[10]
- Where int specifies the data type or type of elements arrays stores.
- “arr” is the name of array & the number specified inside the square brackets is the number of elements an array can store, this is also called sized or length of array.

Arrays

- Following are some of the concepts to be remembered about arrays:
  - The individual element of an array can be accessed by specifying name of the array, following by index or subscript inside square brackets.
  - The first element of the array has index zero[0]. It means the first element and last element will be specified as:arr[0] & arr[9]

Respectively.

Arrays

- The elements of array will always be stored in the consecutive (continues) memory location.
- The number of elements that can be stored in an array, that is the size of array or its length is given by the following equation:

(Upperbound-lowerbound)+1

Arrays

- For the above array it would be (9-0)+1=10,where 0 is the lower bound of array and 9 is the upper bound of array.
- Array can always be read or written through loop. If we read a one-dimensional array it require one loop for reading and other for writing the array.

Arrays

- For example: Reading an array

```
For(i=0;i<=9;i++)
    scanf(“%d”,&arr[i]);
```

- For example: Writing an array

```
For(i=0;i<=9;i++)
    printf(“%d”,arr[i]);
```

Arrays

- If we are reading or writing two-dimensional array it would require two loops. And similarly the array of a N dimension would required N loops.
- Some common operation performed on array are:
  - Creation of an array
  - Traversing an array

Arrays

- Insertion of new element
- Deletion of required element
- Modification of an element
- Merging of arrays

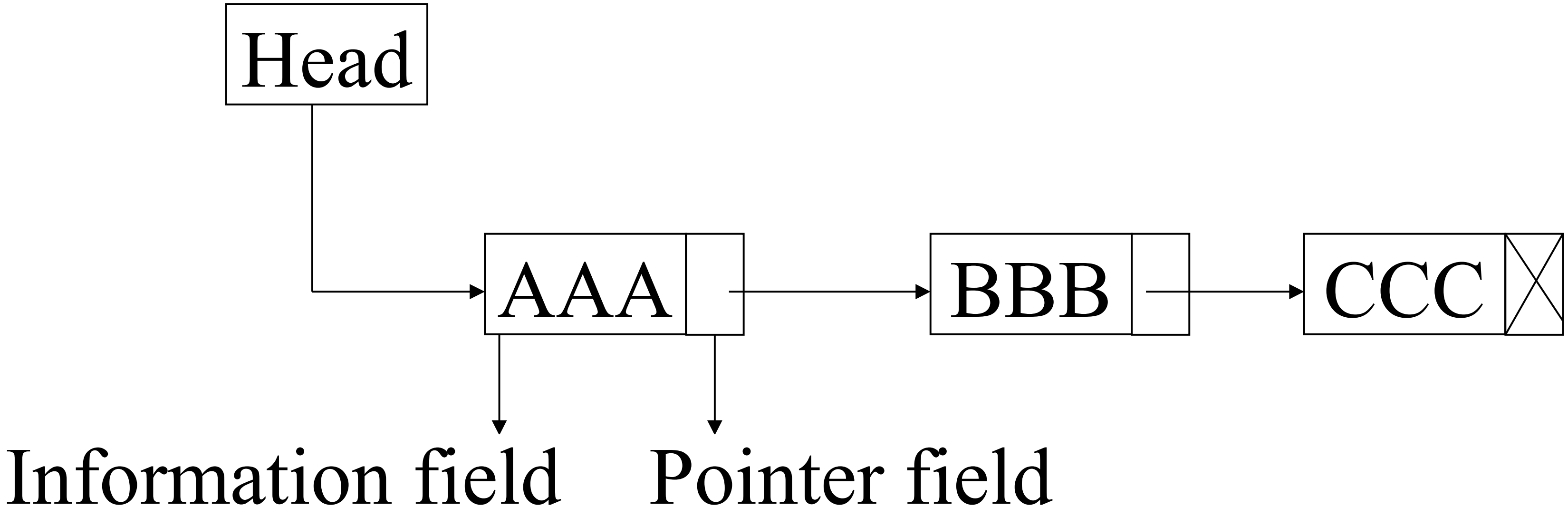
Lists

- A lists (Linear linked list) can be defined as a collection of variable number of data items.
- Lists are the most commonly used non-primitive data structures.
- An element of list must contain at least two fields, one for storing data or information and other for storing address of next element.
- As you know for storing address we have a special data structure of list the address must be pointer type.

Lists

- Technically each such element is referred to as a node, therefore a list can be defined as a collection of nodes as show bellow:

[Linear Liked List]



Lists

- Types of linked lists:
  - Single linked list
  - Doubly linked list
  - Single circular linked list
  - Doubly circular linked list

Stack

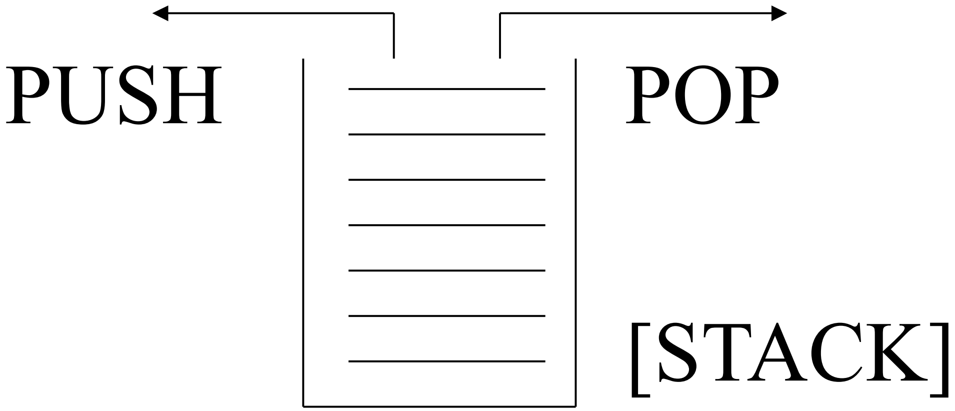
- A stack is also an ordered collection of elements like arrays, but it has a special feature that deletion and insertion of elements can be done only from one end called the top of the stack (TOP)
- Due to this property it is also called as last in first out type of data structure (LIFO).

Stack

- It could be through of just like a stack of plates placed on table in a party, a guest always takes off a fresh plate from the top and the new plates are placed on to the stack at the top.
- It is a non-primitive data structure.
- When an element is inserted into a stack or removed from the stack, its base remains fixed where the top of stack changes.

Stack

- Insertion of element into stack is called PUSH and deletion of element from stack is called POP.
- The bellow show figure how the operations take place on a stack:



Stack

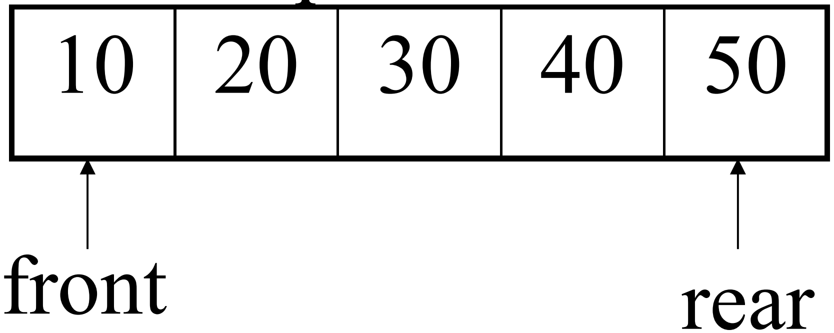
- The stack can be implemented into two ways:
  - Using arrays (Static implementation)
  - Using pointer (Dynamic implementation)

Queue

- Queue are first in first out type of data structure (i.e. FIFO)
- In a queue new elements are added to the queue from one end called REAR end and the element are always removed from other end called the FRONT end.
- The people standing in a railway reservation row are an example of queue.

Queue

- Each new person comes and stands at the end of the row and person getting their reservation confirmed get out of the row from the front end.
- The bellow show figure how the operations take place on a stack:



10	20	30	40	50

front

rear

Queue

- The queue can be implemented into two ways:
  - Using arrays (Static implementation)
  - Using pointer (Dynamic implementation)

Trees

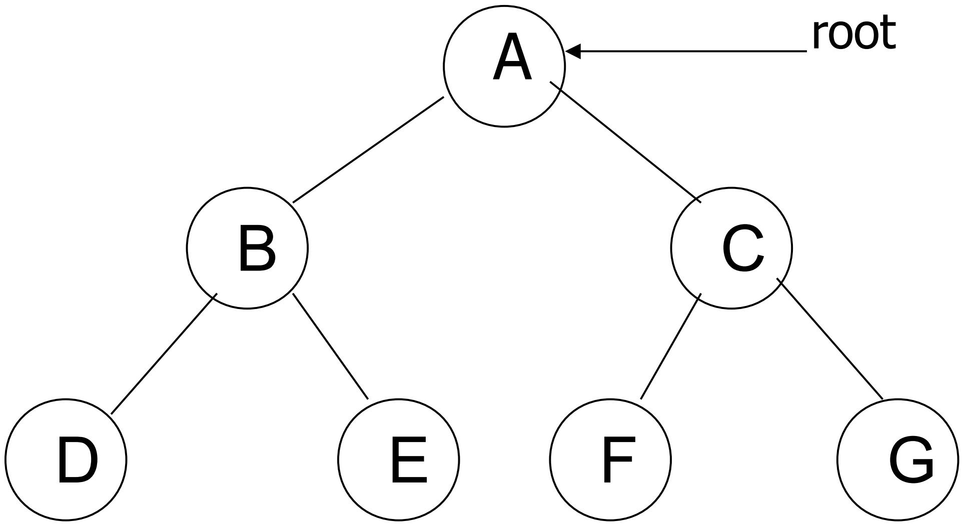
- A tree can be defined as finite set of data items (nodes).
- Tree is non-linear type of data structure in which data items are arranged or stored in a sorted sequence.
- Tree represent the hierarchical relationship between various elements.

Trees

- In trees:
- There is a special data item at the top of hierarchy called the Root of the tree.
- The remaining data items are partitioned into number of mutually exclusive subset, each of which is itself, a tree which is called the sub tree.
- The tree always grows in length towards bottom in data structures, unlike natural trees which grows upwards.

Trees

- The tree structure organizes the data into branches, which related the information.



Graph

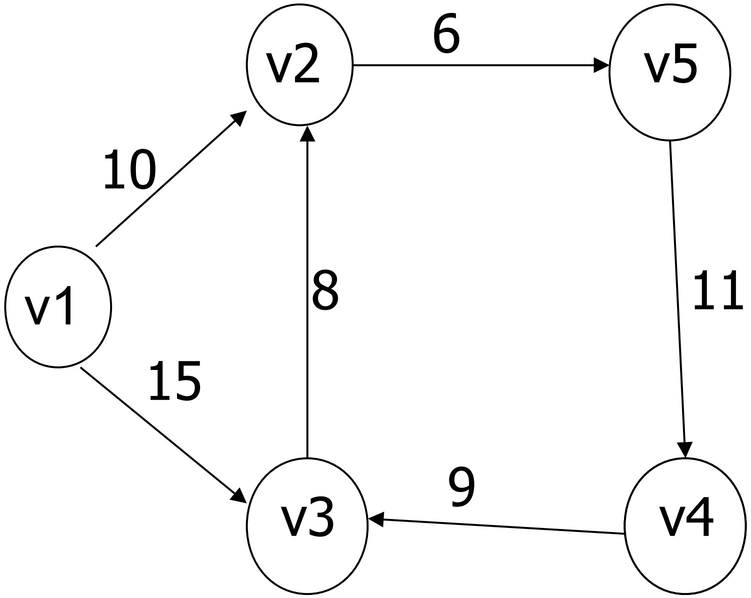
- Graph is a mathematical non-linear data structure capable of representing many kind of physical structures.
- It has found application in Geography, Chemistry and Engineering sciences.
- Definition: A graph G(V,E) is a set of vertices V and a set of edges E.

Graph

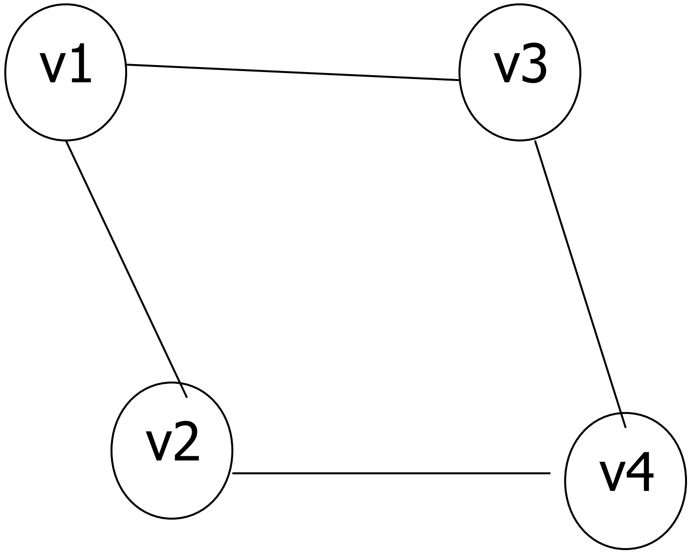
- An edge connects a pair of vertices and many have weight such as length, cost and another measuring instrument for according the graph.
- Vertices on the graph are shown as point or circles and edges are drawn as arcs or line segment.

Graph

- Example of graph:



[a] Directed & Weighted Graph



[b] Undirected Graph

Graph

- Types of Graphs:
  - Directed graph
  - Undirected graph
  - Simple graph
  - Weighted graph
  - Connected graph
  - Non-connected graph

Algorithm Complexity and Time-Space trade-off

Algorithms

- An algorithm is a procedure for solving a problem in finite number of steps
- Algorithm is a well defined computational procedure that takes some value (s) as input, and produces some value (s) as output.
- Algorithm is finite number of computational statements that transform input into the output
- An Algorithm is said to be accurate and truthful only when it provides the exact wanted output.

Algorithms

- An algorithm may be given in different forms.
  - A description using English/other languages
  - A real computer program, e.g. C++
  - A pseudo-code, C-like program, program-language-like program.
- Program = algorithms + data structures

Analysis of Algorithms

- Why need algorithm analysis?
  - Just writing a syntax-error-free program is not enough. We need to know whether the algorithm is correct or not, i.e. whether it can give correct answers for the inputs.

  - If the program is run on a large data set, the running time becomes an issue. We want to know how the algorithm performs when the input size is large. The program may be computationally inefficient, or may need lots of memory. We analyze the resources that the algorithm requires: memory, and computation time. We can also compare algorithms without implementations.

  - We analysis algorithms, rather than problems. A problem can be solved with several algorithms, some are more efficient than others.

Analysis of Algorithms

- Analysis of Algorithm refers to calculating or guessing resources needful for the algorithm.
- Resources means computer memory, processing time etc.
- In all these factors, time is most important because the program developed should be fast enough.

Analysis of Algorithms

- Time complexity
  - The amount of time that an algorithm needs to run to completion

- Space complexity
  - The amount of memory an algorithm needs to run

- We will occasionally look at space complexity, but we are mostly interested in time complexity.

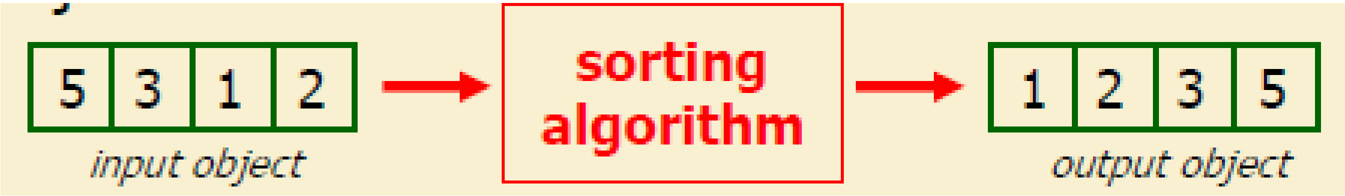
- Thus the better algorithm is the one which runs faster (has smaller time complexity)

Analysis of Algorithms

- There are several factors affecting the running time
  - computer
  - compiler
  - algorithm
  - input to the algorithm
    - The *content of the input affects the running time* typically, the *input size (number of items in the input) is the main* consideration
    - E.g. sorting problem ⇒ the number of items to be sorted.

Running time of an Algorithm

- Running time of an algorithm depends upon the input size and nature of input.
- Most algorithms transform input objects into output objects



- The running time of an algorithm typically grows with the input size
  - idea: analyze running time as a function of input size

Running Time of an Algorithm

- Even on inputs of the same size, running time can be very different
  - Example: algorithm that finds the first prime number in an array by scanning it left to right
- Idea: analyze running time in the
  - best case
  - worst case
  - average case

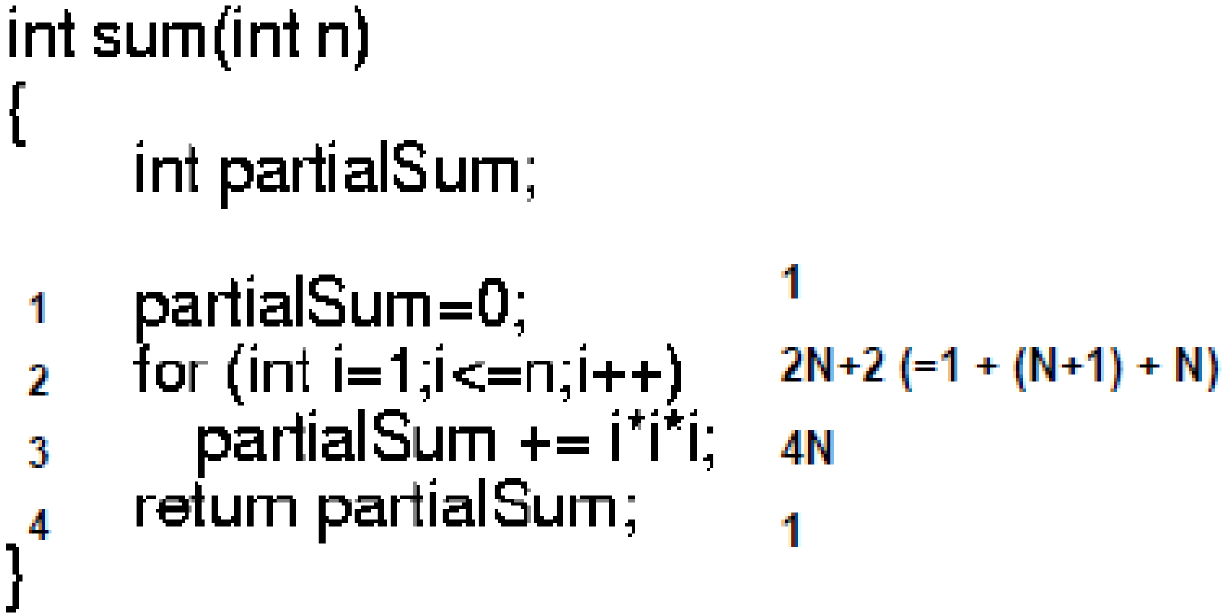
Finding Running Time of an Algorithm

- Running time is measured by number of steps/primitive operations performed

- We use RAM (Random Access Model), in which each operation (e.g. +, -, x, /,=) and each memory access take one run-time unit. Loops and functions can take multiple time units.

- Count the number of basic operations of an algorithm
  - Read, write, compare, assign, jump, arithmetic operations (increment, decrement, add, subtract, multiply, divide), open, close, logical operations (not/complement, AND, OR), …

Example



```
int sum(int n)
{
        int partialSum;

  1     partialSum=0;                  1
  2     for (int i=1;i<=n;i++)         2N+2 (=1 + (N+1) + N)
  3         partialSum += i*i*i;       4N
  4     return partialSum;             1
}
```

- Lines 1 and 4 count for one unit each.
- Line 3: executed N times, each time four units.
- Line 2: (1 for initialization, N+1 for all tests, N for all increments) total 2N + 2.
- total time units: 6N + 4. (⇒ O(N), will be discussed later.)

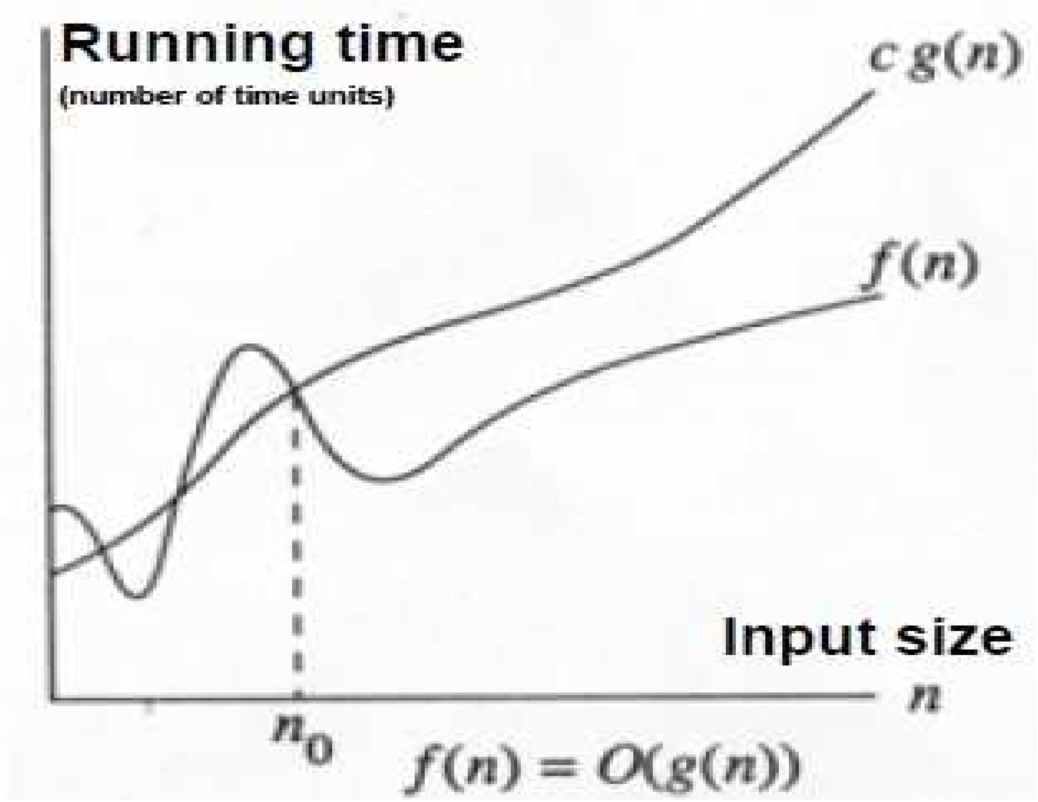
Asymptotic Complexity

- The 6N+4 time bound is said to "grow asymptotically" like N
- This gives us an approximation of the complexity of the algorithm
- Ignores lots of (machine dependent) details, concentrate on the bigger picture

Comparing Functions: Asymptotic Notation

- Big Oh Notation: Upper bound
- Omega Notation: Lower bound
- Theta Notation: Tighter bound

Growth Rate of the Algorithm Running Time



- Exact running time f(N) is difficult to find for some cases. It is easier to find the upper and lower bounds of f(N).
- Now, we find the orders of growth of f(N) for large N.

Asymptotic Notation
Big-Oh

- If f(N) and g(N) are two complexity functions, we say

  f(N) = O(g(N))

  (read "f(N) is order g(N)", or "f(N) is big-O of g(N)")

- There are positive constants c and n0 such that

  f(N) ≤ c g(N) when N ≥ n0

- The growth rate of f(N) is less than or equal to the growth rate of g(N). In other words, f(N) grows no faster than g(N) for large N.
- g(N) is an upper bound on f(N).

Big-Oh Example

- Let f(N) = $2N^2$ Then
  - f(N) = O(N^4) (loose bound)
  - f(N) = O(N^3) (loose bound)
  - f(N) = O(N^2) (It is the best answer and the bound is asymptotically tight.)

- O(N^2): reads “order N-squared” or “Big-Oh N-squared”.

Big-Oh Notation Rules

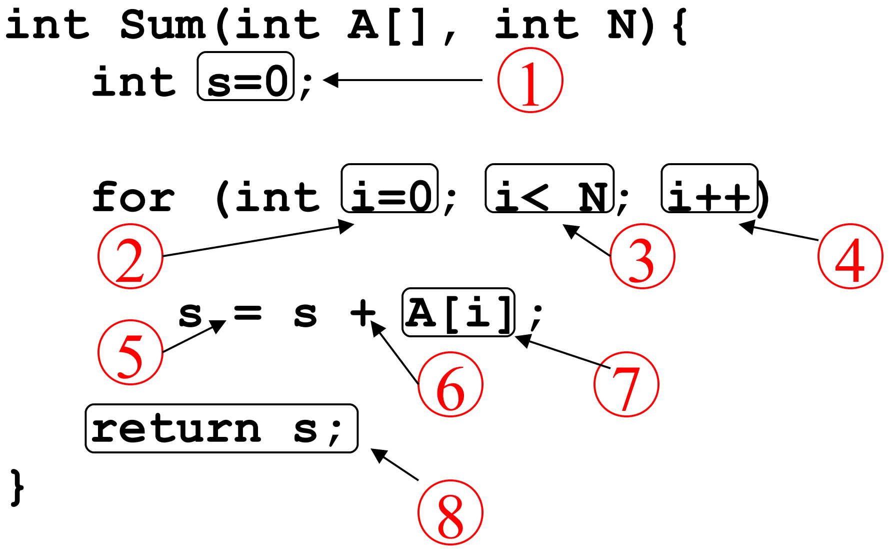
- When considering the growth rate of a function using Big-Oh notation,
  - we ignore the lower order terms
  - We ignore the coefficients of the highest-order term; and
  - we don't need to specify the base of logarithm
    - Note that changing the base from one constant to another changes the value of the logarithm by only a constant factor

- If T1(N) = O(f(N)) and T2(N) = O(g(N)), then
  - T1(N) + T2(N) = O(f(N) + g(N))

    or max(O(f(N)), O(g(N))),
  - T1(N) * T2(N) = O(f(N) * g(N))

Big-Oh Example (2)

```
// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A
```



```
int Sum(int A[], int N){
    int s=0;

    for (int i=0; i< N; i++)
       s = s + A[i];
    return s;
}
```

1,2,8: Once

3,4,5,6,7: Once per each iteration
of for loop, N iteration

Total: 5N + 3

The *complexity function* of the algorithm is : $f(N) = 5N + 3$

Big-Oh Example (2)

- Estimated running time for different values of N:

N = 10	=> 53 steps
N = 100	=> 503 steps
N = 1,000	=> 5003 steps
N = 1,000,000	=> 5,000,003 steps

As N grows, the number of steps grow in *linear* proportion to N for this function *“Sum”*

- <u>What Dominates in Previous Example?</u>

  What about the +3 and 5 in 5N+3?

  - As N gets large, the +3 becomes insignificant
  - 5 is inaccurate, as different operations require varying amounts of time and also does not have any significant importance

More Big-Oh Example

- N^2/ 2 – 3N = O(N^2)

(We throw away leading constants and low-order terms.)

- 1+4N = O(N)
- 7N^2+ 10N + 3 = O(N^2) = O(N^3) (loose bound)
- $\log_{10} \text{N} \quad \log_2 \text{N} / \log_2 10 \quad \text{O}(\log_2 \text{N}) \quad \text{O}(\log \text{N})$

- log N + N = O(N)

To prove that mathematically:

Example:

Consider f(n) = $2n^2$ – 3n + 6. Then f(n) = O(n^2) = O(n^3)

Try some values of c and find out the corresponding n_0 which satisfies the condition.

1. f(n) = O(n^2)

(a) Suppose we choose c = 2:

$$2n^2 - 3n + 6 \le 2n^2$$

$$-3n + 6 \le 0$$

$$n \ge 2$$

So we can see that if we choose c = 2 and n_0 = 2, the condition is satisfied.

(b) Suppose we choose c = 3:

$$2n^2 - 3n + 6 \le 3n^2$$

$$n^2 + 3n - 6 \ge 0$$

$$n \le -4.37\text{(ignored) or } n \ge 1.37$$

So we can see that if we choose c = 3 and n_0 = 1.37, the condition is satisfied.

* There are other values of c and n_0 which satisfy the condition.

2. f(n) = O(n3)

Suppose we choose c = 1:

$$2n^2 - 3n + 6 \le n^3$$

$$n^3 + 2n^2 - 3n + 6 \le 0$$

$$n \ge 2$$

So we can see that if we choose c = 1 and n0 = 2, the condition is satisfied.

* There are other values of c and n0 which satisfy the condition.

Performance Classification

f(*n*)	Classification
1	*Constant*: run time is fixed, and does not depend upon n. Most instructions are executed once, or only a few times, regardless of the amount of information being processed
log n	*Logarithmic*: when *n* increases, so does run time, but much slower. Common in programs which solve large problems by transforming them into smaller problems.
n	*Linear*: run time varies directly with *n*. Typically, a small amount of processing is done on each element.
n log n	When *n* doubles, run time slightly more than doubles. Common in programs which break a problem down into smaller sub-problems, solves them independently, then combines solutions
n2	*Quadratic*: when *n* doubles, runtime increases fourfold. Practical only for small problems; typically the program processes all pairs of input (e.g. in a double nested loop).
n3	*Cubic*: when n doubles, runtime increases eightfold
2n	*Exponential*: when n doubles, run time squares. This is often the result of a natural, “brute force” solution.

What happens if we double the input size N?

Nlog2N		5N	N log2N	N2	2N
8	3	40	24	64	256
16	4	80	64	256	65536
32	5	160	160	1024	~109
64	6	320	384	4096	~1019
128	7	640	896	16384	~1038
256	8	1280	2048	65536	~1076

Standard Analysis Techniques

- Constant time statements
- Analyzing Loops
- Analyzing Nested Loops
- Analyzing Sequence of Statements
- Analyzing Conditional Statements

Constant time statements

- Simplest case: O(1) time statements

- Assignment statements of simple data types

```
int x = y;
```

- Arithmetic operations:

```
x = 5 * y + 4 - z;
```

- Array assignment:

```
A[j] = 5;
```

- Most conditional tests:

```
if (x < 12) ...
```

Analyzing Loops[1]

- Any loop has two parts:
  - How many iterations are performed?
  - How many steps per iteration?

```
int sum = 0,j;
for (j=0; j < N; j++)
sum = sum +j;
```

  - Loop executes N times (0..N-1)
  - 4 = O(1) steps per iteration
- Total time is N * O(1) = O(N*1) = O(N)

Analyzing Loops[2]

- What about this **for** loop?

```
 int sum =0, j;
 for (j=0; j < 100; j++)
    sum = sum +j;
```

- Loop executes 100 times
- 4 = O(1) steps per iteration
- Total time is 100 * O(1) = O(100 * 1) = O(100) = O(1)

Analyzing Nested Loops[1]

- Treat just like a single loop and evaluate each level of nesting as needed:

```
int j,k;
for (j=0; j<N; j++)
for (k=N; k>0; k--)
sum += k+j;
```

- Start with outer loop:
  - How many iterations? N
  - How much time per iteration? Need to evaluate inner loop
- Inner loop uses O(N) time
- Total time is N * O(N) = O(N*N) = O(N2)

Analyzing Nested Loops[2]

- What if the number of iterations of one loop depends on the counter of the other?

```
int j,k;
for (j=0; j < N; j++)
for (k=0; k < j; k++)
sum += k+j;
```

- Analyze inner and outer loop together:
- Number of iterations of the inner loop is:
- $0 + 1 + 2 + ... + (N-1) = O(N2)$

Analyzing Sequence of Statements

- For a sequence of statements, compute their complexity functions individually and add them up

```
for (j=0; j < N; j++)
for (k =0; k < j; k++)
sum = sum + j*k;
for (l=0; l < N; l++)
sum = sum -l;
cout<<“Sum=”<<sum;
```

O(N2)

O(N)

O(1)

Total cost is O(N2) + O(N) +O(1) = O(N2)

Analyzing Conditional Statements

What about conditional statements such as

```
if (condition)
  statement1;
else
  statement2;
```

- Running time = never more than the running time of the **test Condition** plus the larger of the running times of **Statement1** and **Statement2**.

Arrays

What is Array

- An Array is a structured collection of components, all of same type, that is given a single name. Each component (array element) is accessed by an index that indicates the component’s position within the collection.

Defining Array

- Like other variables in C++, an array must be defined before it can be used to store information.
- Like other definitions, an array definition specifies a variable type and a name. But it includes another feature i.e. size.

DataType ArrayName [Const Int Expression];

Array Elements

- The items in an array are called array elements.
- All elements in an array are of the same type; only the values vary.
- Example

```
int   array1[4] = { 10, 5, 678, -400 } ;
```

Accessing Array Elements

- To Access an individual array component, we write the array name, followed by an expression enclosed in square brackets. The expression specifies which component to access.
- Syntax

  ArrayName [IndexExpression]

- Example

  array1[2]

  array1[i]   where i = 3

Initializing array in Declarations

- To initialize an array, you have to specify a list of initial values for the array elements, separate them with commas and enclose the list within braces.

```
int  array1[5] = {23, 10, 16, 37, 12};
```

- We don't need to use the array size when we initialize all the array elements, since the compiler can figure it out by counting the initializing variables.

```
int array1[ ] = { 23, 10, 16, 37};
```

- What happens if you do use an explicit array size, but it doesn't agree with the number of components ?
  - If there are too few components/ items , the missing element will be set to zero.
  - If there are two many, an error is signaled.

Lack of Aggregate Array Operations

- C++ does not allow aggregate operations on arrays.

```
        int x[50], y[50] ;
```

- There is no aggregate assignment of y to x

```
        x = y;   //not valid
```

- To copy array y into array x, you must do it yourself, element by element.

```
    for ( i=0; i<50; i++)
        x[i] = y[i];            //valid operation
```

- Similarly, there is no aggregate comparison of arrays.

```
    if (x == y )   //Not valid
```

Lack of Aggregate Array Operations

- Also, you cannot perform aggregate input / output operations on arrays.

```
cin>>x;     //not valid, where x is an array
cout<<x     //not valid
```

- You cannot perform aggregate arithmetic operations on arrays

```
x = x + y  // not valid, where x and y are arrays
```

- Finally, it is not possible to return an entire array as the value of a value-returning function

```
return x;  //not valid, where x is an array.
```

Example of One Dimensional Array

```
void main()
  {
        double sales [6], average, total=0;
        cout<< “Enter sales of 6 days”;
        for( int j=0; j<6; j++)
             cin >> sales[ i ];
        for (int j=0; j<6; j++)
             total += sales[ j ] ;
        average = total / 6;
        cout<< “Average =”<< average;
  }
```

Multidimensional Arrays

- A two dimensional array is used to represent items in a table with rows and columns, provided each item in the table is of same data type.
- Each component is accessed by a pair of indexes that represent the component’s position in each dimension.

Defining Multidimensional Array

- **Two Dimensional Array**
- The array is defined with two size specifiers, each enclosed in brackets

  DataType ArrayName[ConstIntExp][ConstIntExp]
- Example

  ```
  double array2[3][4];
  ```
- **Three Dimensional Array**

  ```
  float  array3[x][y][z]
  ```

Accessing Multidimensional Array Elements

- Array elements in two dimensional arrays required two indexes

```
    array2[1][2]
```

- Notice that each index has its own set of brackets. Don’t write commas.

```
    array2[1,2]   // not valid syntax
```

Example Two Dimensional Array

```
void main()
{
   float array2[3][3]= {{ 12.2,  11.0,  9.6   },
                        {  23.9, -50.6,  2.3  },
                        {  2.2,     3.3,   4.4   }  };

   for (int row=0; row<3; row++)
      for (int col=0; col<3; col++)
         cout<< array2[row][col];
}
```

Arrays, Pointers and Strings

Arrays

- An Array is a collection of variables of the same type that are referred to through a common name.
- Declaration

type var_name[size]

e.g

```
int A[6];
double d[15];
```

Array Initialization

After declaration, array contains some garbage value.

Static initialization

```
int month_days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Run time initialization

```
int i;
int A[6];
for(i = 0; i < 6; i++)
        A[i] = 6 - i;
```

Memory addresses

- Memory is divided up into one byte pieces individually addressed.

  - minimum data you can request from the memory is 1 byte

- Each byte has an address.

  for a 32 bit processor, addressable memory is 232

0A	0x00001234
23	0x00001235
6C	0x00001236
1D	0x00001237
‘W’	0x00001238
‘o’	0x00001239
‘w’	0x0000123A
‘\0’	0x0000123B

. .

. .

4 . .

Array - Accessing an element

int A[6];

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
0x1000	0x1004	0x1008	0x1012	0x1016	0x1020
6	5	4	3	2	1

6 elements of 4 bytes each,
total size = 6 x 4 bytes = 24 bytes

```
int tmp = A[2];
```

```
A[3] = 5;
```

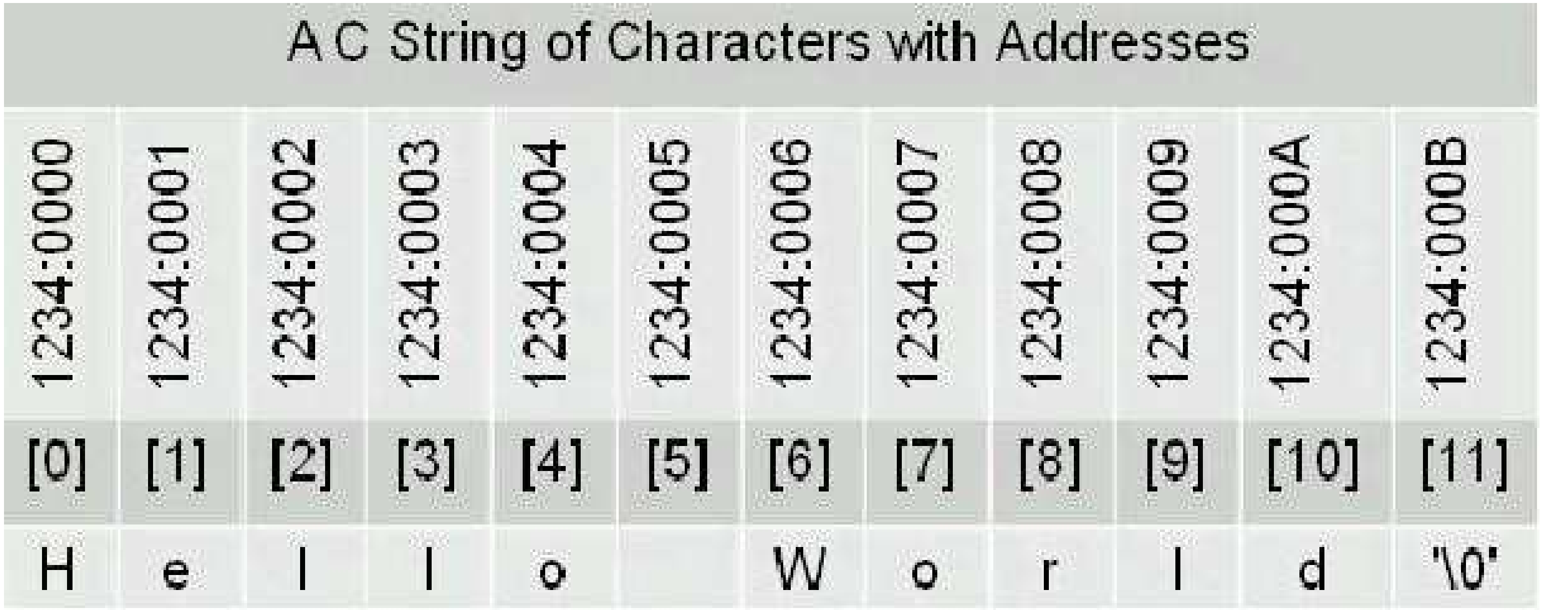
Read an element

Write to an element

Strings in C

- No “Strings” keyword
- A string is an array of characters.

```
char string[] = “hello world”;
OR
char *string = “hello world”;
```



A C String of Characters with Addresses

1234:0000	1234:0001	1234:0002	1234:0003	1234:0004	1234:0005	1234:0006	1234:0007	1234:0008	1234:0009	1234:000A	1234:000B
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
H	e	l	l	o		W	o	r	l	d	'\0'

Significance of NULL character ‘\0’

```
char string[] = “hello world”;
printf(“%s”, string);
```

- Compiler has to know where the string ends
- ‘\0’ denotes the end of string

{program: hello.c}

Some more characters (do $man ascii):

‘\n’ = new line, ‘\t’ = horizontal tab, ‘\v’ = vertical tab, ‘\r’ = carriage return

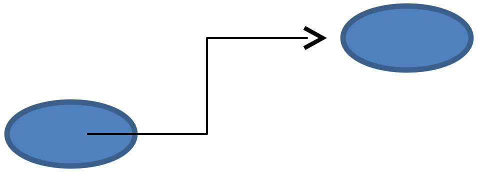
Pointers in C

- A char pointer points to a single byte.
- An int pointer points to first of the four bytes.

```
int *p; ⇔ int* p;
```

- A pointer itself has an address where it is stored in the memory. Pointers are usually four bytes.

```
int i = 4;
p = &i;
```



- * is called the dereference operator
- *p gives the value pointed by p

4 i

More about pointers

```
int x = 1, y = 2, z[10];
int *ip;       /* A pointer to an int */

ip = &x;       /* Address of x */
y = *ip;       /* Content of ip */
*ip = 0;       /* Clear where ip points */
ip = &z[0];      /* Address of first element
               of z */
```

{program: pointer.c}

Pointer Arithmetic

- A 32-bit system has 32 bit address space.
- To store any address, 32 bits are required.

- Pointer arithmetic : p+1 gives the next memory location assuming cells are of the same type as the base type of p.

Pointer arithmetic: Valid operations

- pointer +/- integer → pointer
- pointer - pointer → integer

- pointer <any operator> pointer → invalid
  - pointer +/- pointer → invalid

Pointer Arithmetic: Example

```
int *p, x = 20;
p = &x;
printf("p      = %p\n", p);
printf("p+1 = %p\n", (int*)p+1);
printf("p+1 = %p\n", (char*)p+1);
printf("p+1 = %p\n", (float*)p+1);
printf("p+1 = %p\n", (double*)p+1);
```

Sample output:

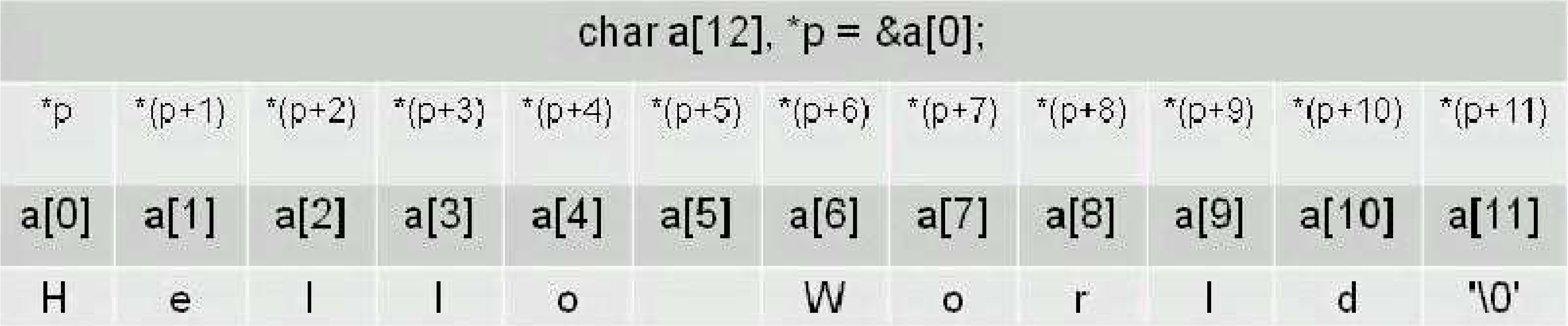
```
p      = 0022FF70
```

Pointers and arrays

- Pointers and arrays are tightly coupled.

char a[] = “Hello World”;

char *p = &a[0];



char a[12], *p = &a[0];											
*p	*(p+1)	*(p+2)	*(p+3)	*(p+4)	*(p+5)	*(p+6)	*(p+7)	*(p+8)	*(p+9)	*(p+10)	*(p+11)
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
H	e	l	l	o		W	o	r	l	d	'\0'

Pointers and function arguments

- Functions only receive copies of the variables passed to them.

{program: swap_attempt_1.c}

- A function needs to know the address of a variable if it is to affect the original variable

{program: swap_attempt_2.c}

```
printf(“hello world\n”);
```

- Large items like strings or arrays cannot be passed to functions either.

- What is passed is the address of “hello

2-Dimensional Arrays (Array of arrays)

```
int d[3][2];
```

Access the point 1, 2 of the array:

```
d[1][2]
```

Initialize (without loops):

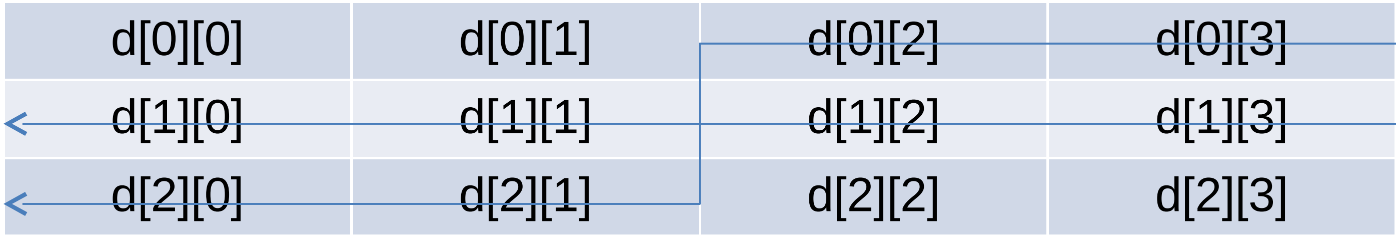
```
int d[3][2] = {{1, 2}, {4, 5}, {7, 8}};
```

More about 2-Dimensional arrays

A Multidimensional array is stored in a row major format.

A two dimensional case:

➔ next memory element to d[0][3] is d[1][0]



d[0][0]	d[0][1]	d[0][2]	d[0][3]
d[1][0]	d[1][1]	d[1][2]	d[1][3]
d[2][0]	d[2][1]	d[2][2]	d[2][3]

What about memory addresses sequence of a three dimensional array?

➔ next memory element to t[0][0][0] is t[0][0][1]

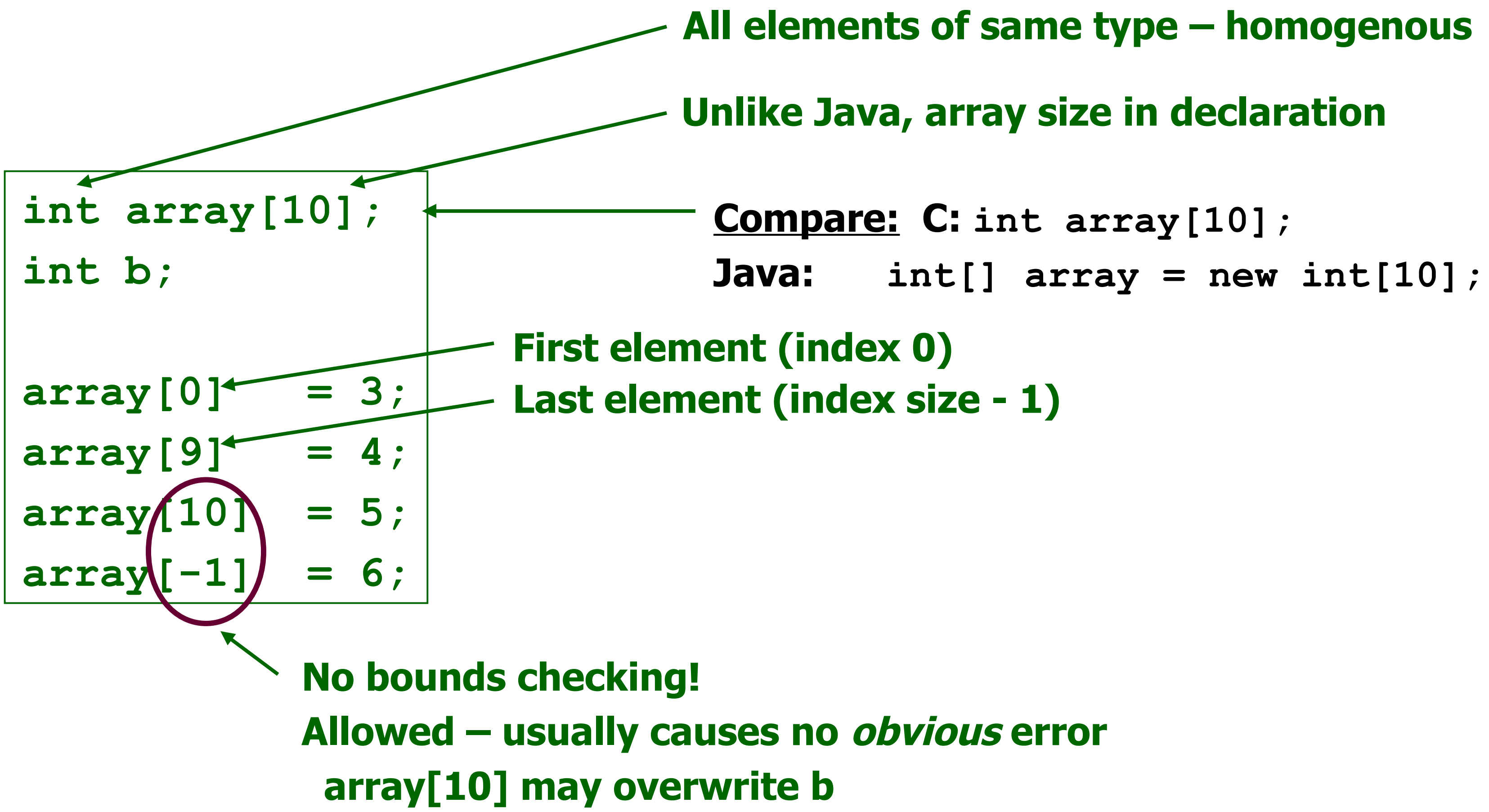
Arrays and Pointers in C

Objectives

Be able to use arrays, pointers, and strings in C programs

Be able to explain the representation of these data types at the machine level, including their similarities and differences

Arrays in C



All elements of same type – homogenous

Unlike Java, array size in declaration

```
int array[10];
int b;

array[0]  = 3;
array[9]  = 4;
array[10] = 5;
array[-1] = 6;
```

Compare: C: `int array[10];`

Java: `int[] array = new int[10];`

First element (index 0)

Last element (index size - 1)

No bounds checking!

Allowed – usually causes no *obvious* error

array[10] may overwrite b

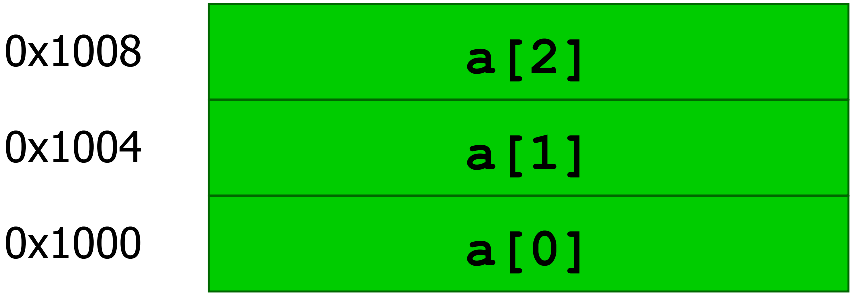
Array Representation

Homogeneous   Each element same size – s bytes

- An array of m data values is a sequence of m s bytes
- Indexing: 0th value at byte s 0, 1st value at byte s 1, …

m and s are <u>not</u> part of representation

- Unlike in some other languages
- s known by compiler – usually irrelevant to programmer
- m often known by compiler – if not, must be saved by programmer



0x1008 a[2]

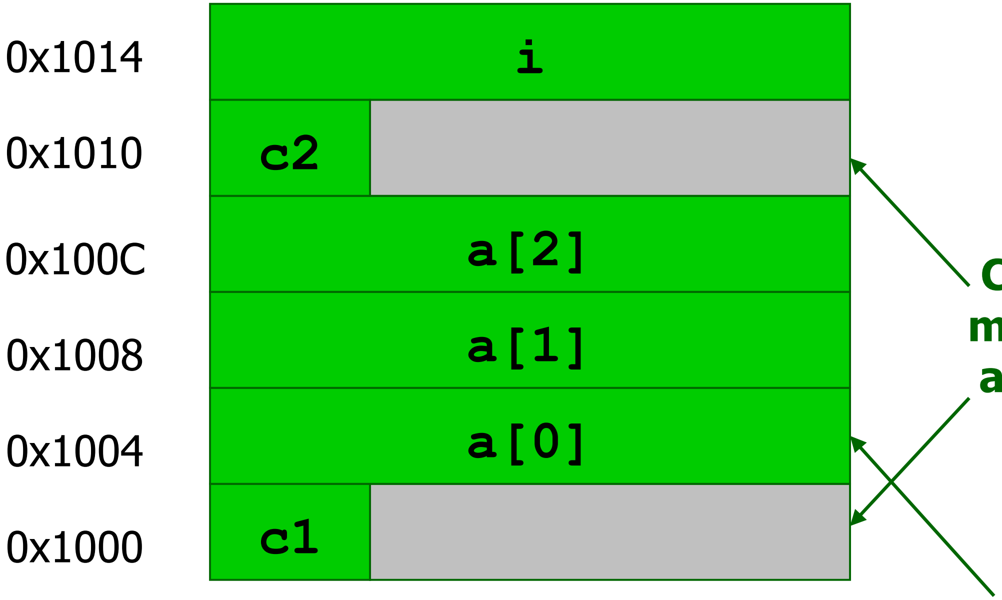
0x1004 a[1]

0x1000 a[0]

```
int a[3];
```

Array Representation

```
char     c1;
int      a[3];
char     c2;
int      i;
```



Could be optimized by making these adjacent, and reducing padding (by default, not)

Array aligned by size of elements

Array Sizes

```
int  array[10];
```

What is

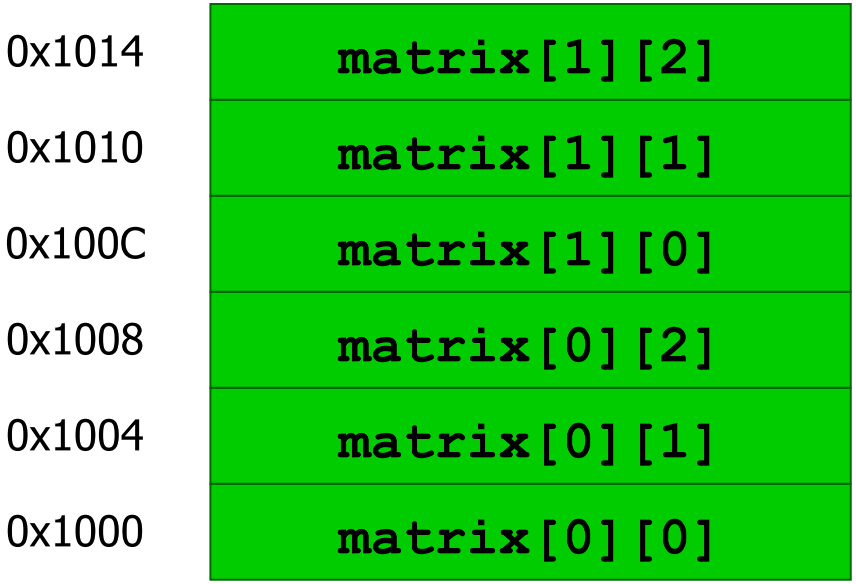
`sizeof(array[3])`**?** **4**

returns the size of an object in bytes

`sizeof(array)`**?** **40**

Multi-Dimensional Arrays

```
int  matrix[2][3];

matrix[1][0] = 17;
```



0x1014	`matrix[1][2]`
0x1010	`matrix[1][1]`
0x100C	`matrix[1][0]`
0x1008	`matrix[0][2]`
0x1004	`matrix[0][1]`
0x1000	`matrix[0][0]`

Recall: no bounds checking

What happens when you write:

```
  matrix[0][3] = 42;
```

"Row Major" Organization

Variable-Length Arrays

```
int
function(int n)
{
      int  array[n];
      …
```

New C99 feature: Variable-length arrays defined within functions

Global arrays must still have fixed (constant) length

Memory Addresses

- **Storage cells are typically viewed as being byte-sized**
  - **Usually the smallest addressable unit of memory**
    - **Few machines can directly address bits individually**
  - **Such addresses are sometimes called *byte-addresses***
- **Memory is often accessed as words**
  - **Usually a word is the largest unit of memory access by a single machine instruction**
    - **CLEAR’s word size is 8 bytes (= `sizeof(long)`)**
  - **A *word-address* is simply the byte-address of the word’s first byte**

Pointers

Special case of bounded-size natural numbers

- **Maximum memory limited by processor word-size**
- **232 bytes = 4GB, 264 bytes = 16 exabytes**

A pointer is just another kind of value

- **A basic type in C**

```
int *ptr;
```

The variable “ptr” stores a pointer to an “int”.

Pointer Operations in C

Creation

& ***variable*** **Returns variable’s memory address**

Dereference

* ***pointer*** **Returns contents stored at address**

Indirect assignment

* ***pointer*** = ***val*** **Stores value at address**

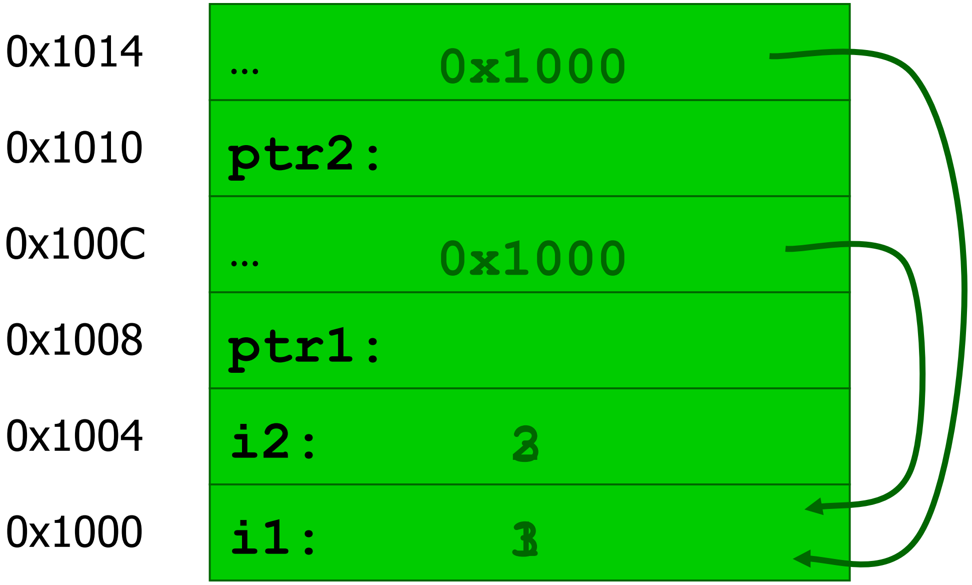
Of course, still have…

Assignment

pointer = ***ptr*** **Stores pointer in another variable**

Using Pointers

```
int  i1;
int  i2;
int *ptr1;
int *ptr2;

i1 = 1;
i2 = 2;

ptr1 = &i1;
ptr2 = ptr1;

*ptr1 = 3;
i2 = *ptr2;
```



Using Pointers (cont.)

```
int  int1     = 1036;   /* some data to point to  */
int  int2     = 8;

int *int_ptr1 = &int1;  /* get addresses of data  */
int *int_ptr2 = &int2;

*int_ptr1 = int_ptr2;

*int_ptr1 = int2;
```

What happens?

Type check warning: `int_ptr2` **is not an** `int`

`int1` **becomes 8**

Using Pointers (cont.)

```
int  int1     = 1036;   /* some data to point to  */
int  int2     = 8;

int *int_ptr1 = &int1;  /* get addresses of data  */
int *int_ptr2 = &int2;

int_ptr1 = *int_ptr2;

int_ptr1 = int_ptr2;
```

What happens?

Type check warning: `*int_ptr2` is not an `int *`

Changes `int_ptr1` – doesn't change `int1`

Pointer Arithmetic

pointer* + *number ***pointer* – *number***

E.g., *pointer* + 1 **adds 1 something to a pointer**

```
char   *p;
char    a;
char    b;

p = &a;
p += 1;
```

```
int    *p;
int     a;
int     b;

p = &a;
p += 1;
```

In each, p now points to b (Assuming compiler doesn't reorder variables in memory)

Adds 1*sizeof(char) to the memory address

Adds 1*sizeof(int) to the memory address

Pointer arithmetic should be used cautiously

A Special Pointer in C

Special constant pointer `NULL`

- **Points to no data**
- **Dereferencing illegal – causes *segmentation fault***

- **To define, include `<stdlib.h>` or `<stdio.h>`**

Generic Pointers

void *: a “pointer to anything”

```
void   *p;
int     i;
char    c;
p = &i;
p = &c;
putchar(*(char *)p);
```

type cast: tells the compiler to “change” an object’s type (for type checking purposes – does not modify the object in any way)

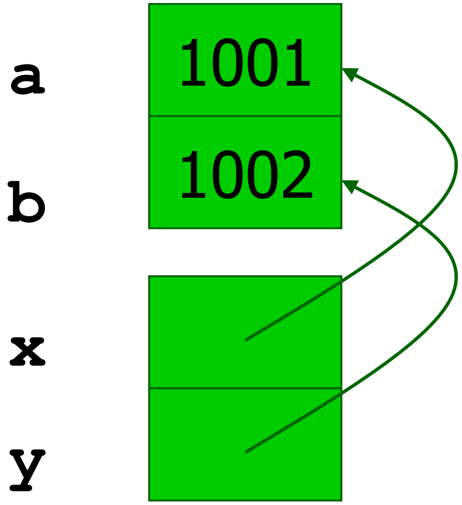
Dangerous! Sometimes necessary…

Lose all information about what type of thing is pointed to

- Reduces effectiveness of compiler’s type-checking
- Can’t use pointer arithmetic

Pass-by-Reference

```
void
set_x_and_y(int *x, int *y)
{

   *x = 1001;
   *y = 1002;
}

void
f(void)
{
   int a = 1;
   int b = 2;

   set_x_and_y(&a, &b);
}
```



Arrays and Pointers

Dirty "secret":

Array name   a pointer to the initial (0th) array element

```
a[i]      *(a + i)
```

An array is passed to a function as a pointer

- **The array size is lost!**

Usually bad style to interchange arrays and pointers

- **Avoid pointer arithmetic!**

Passing arrays:

Really `int *array`

Must explicitly pass the size

```
int
foo(int array[],
    unsigned int size)
{
   … array[size - 1] …
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5)  …
}
```

Arrays and Pointers

```
int
foo(int array[],
    unsigned int size)
{
   …
   printf(“%d\n”, sizeof(array));
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5)  …
   printf(“%d\n”, sizeof(a));
}
```

What does this print? **8**

… because **`array`** is really a pointer

What does this print? **40**

Arrays and Pointers

```
int  i;
int  array[10];

for (i = 0; i < 10; i++)
{
  array[i] = …;
}
```

```
int *p;
int  array[10];

for (p = array; p < &array[10]; p++)
{
  *p = …;
}
```

These two blocks of code are functionally equivalent

Strings

In C, strings are just an array of characters

- **Terminated with '\0' character**
- **Arrays for bounded-length strings**
- **Pointer for constant strings (or unknown length)**

```
char  str1[15] = “Hello, world!\n”;
char *str2     = “Hello, world!\n”;
```

C, …

H	e	l	l	o	,		w	o	r	l	d	!	\n	terminator

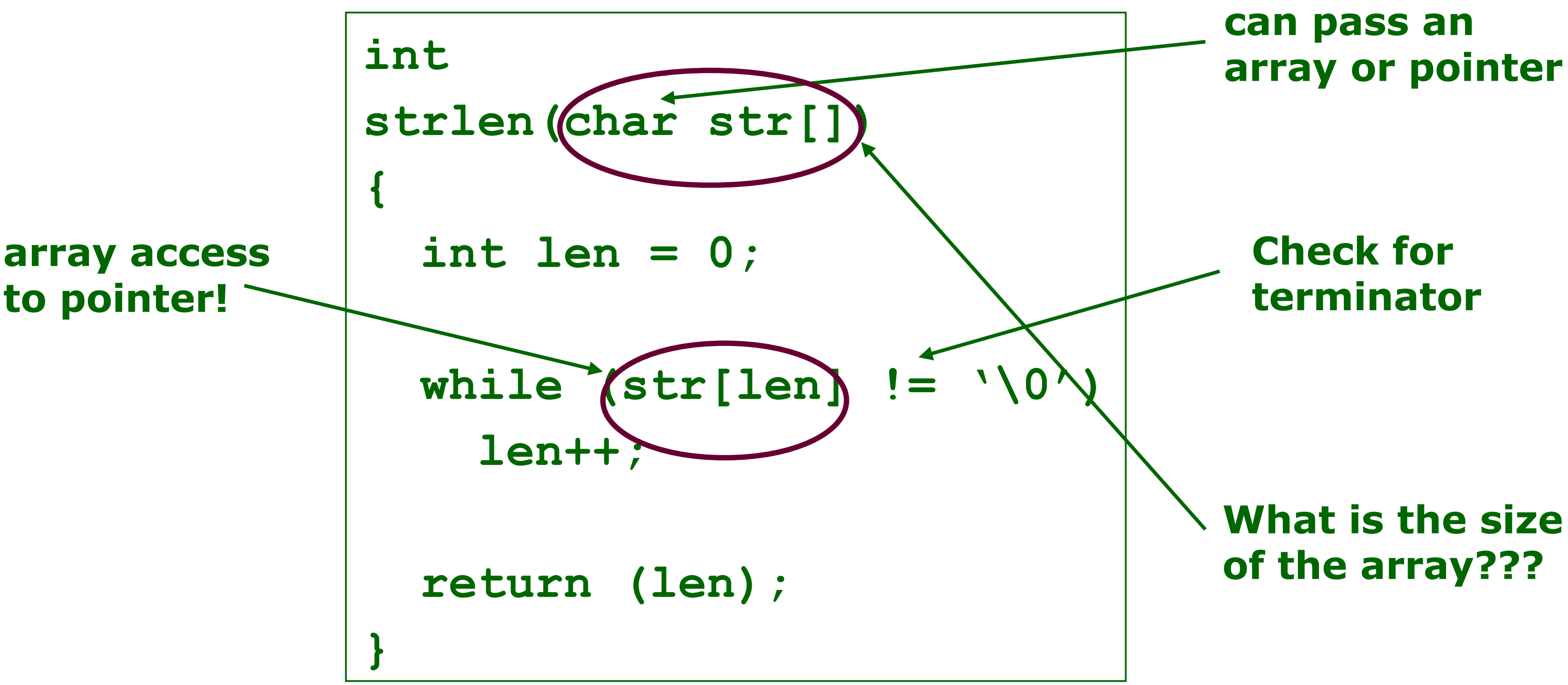
C terminator: `'\0'`

Pascal, Java, …

length	H	e	l	l	o	,		w	o	r	l	d	!	\n

String length

Must calculate length:



```
int
strlen(char str[])
{
  int len = 0;

  while (str[len] != '\0')
    len++;

  return (len);
}
```

can pass an array or pointer

array access to pointer!

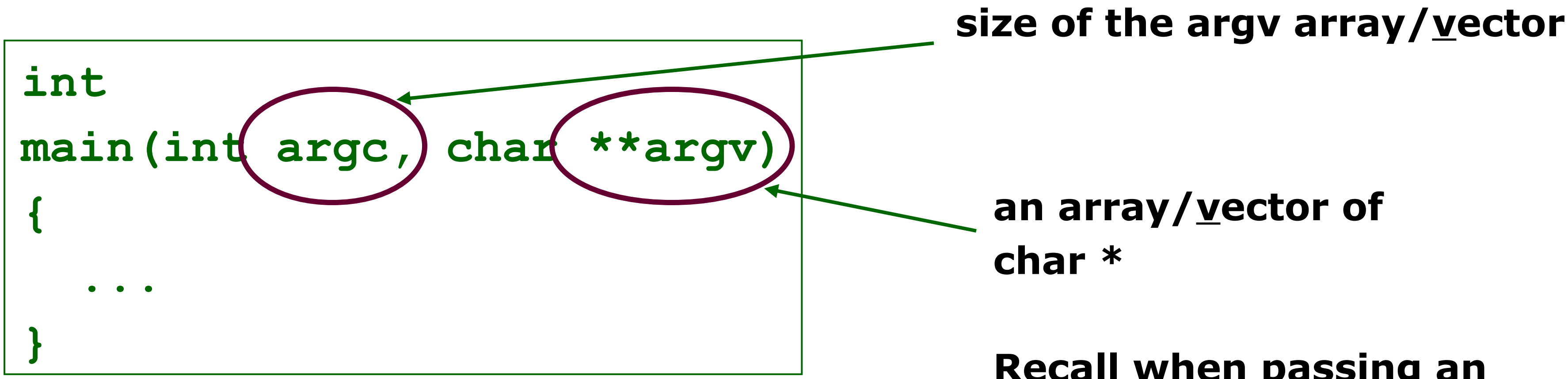
Check for terminator

What is the size of the array???

Provided by standard C library: `#include <string.h>`

Pointer to Pointer (char **argv)

Passing arguments to main:



```
int
main(int argc, char **argv)
{
   ...
}
```

size of the argv array/vector

an array/vector of char *

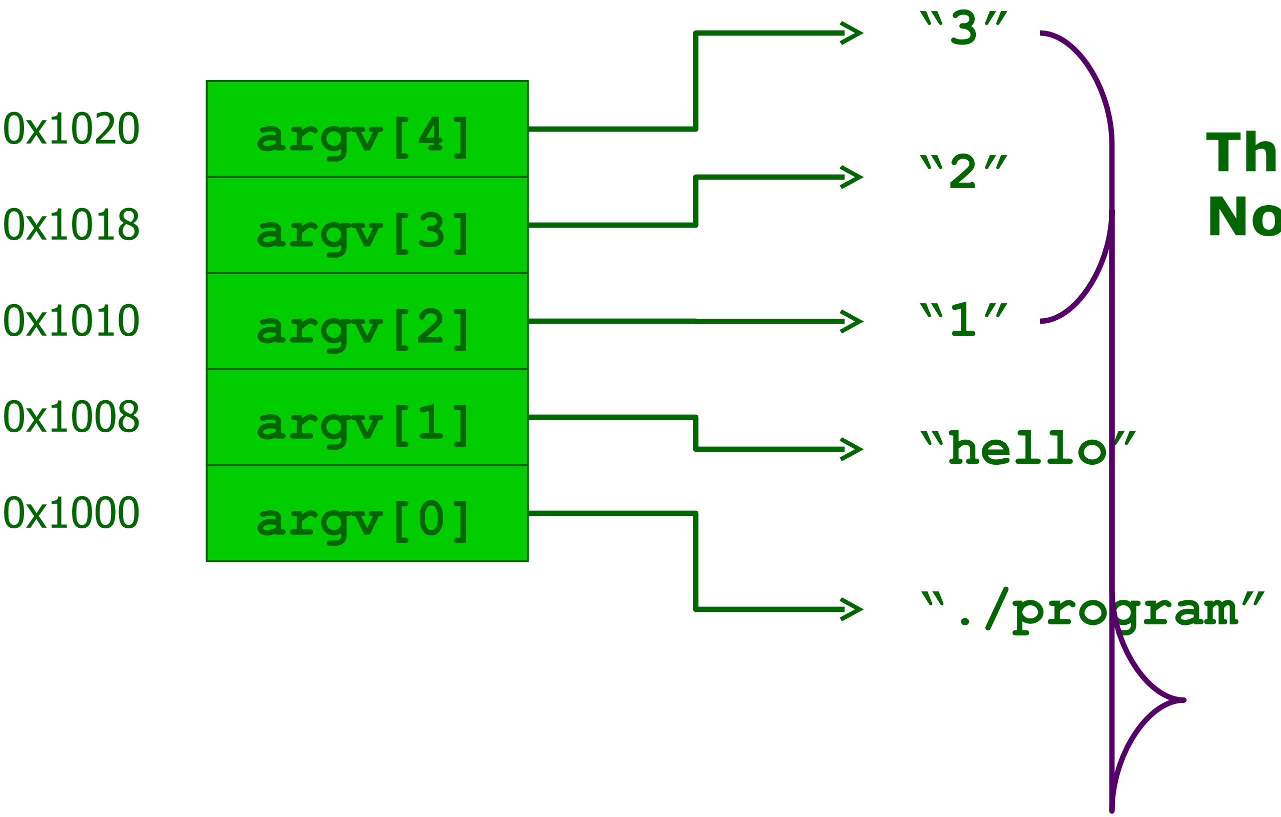
Recall when passing an array, a pointer to the first element is passed

Suppose you run the program this way

```
UNIX% ./program hello 1 2 3
```

`argc == 5` **(five strings on the command line)**

char **argv



These are strings!!
Not integers!

Next Time

Structures and Unions

Stacks are dynamic data structures that follow the **Last In First Out (LIFO)** principle. The last item to be inserted into a stack is the first one to be deleted from it.

For example, you have a stack of trays on a table. The tray at the top of the stack is the first item to be moved if you require a tray from that stack.

Inserting and deleting elements

Stacks have restrictions on the insertion and deletion of elements. Elements can be inserted or deleted only from one end of the stack i.e. from the top. The element at the top is called the top element. The operations of inserting and deleting elements are called push() and pop() respectively.

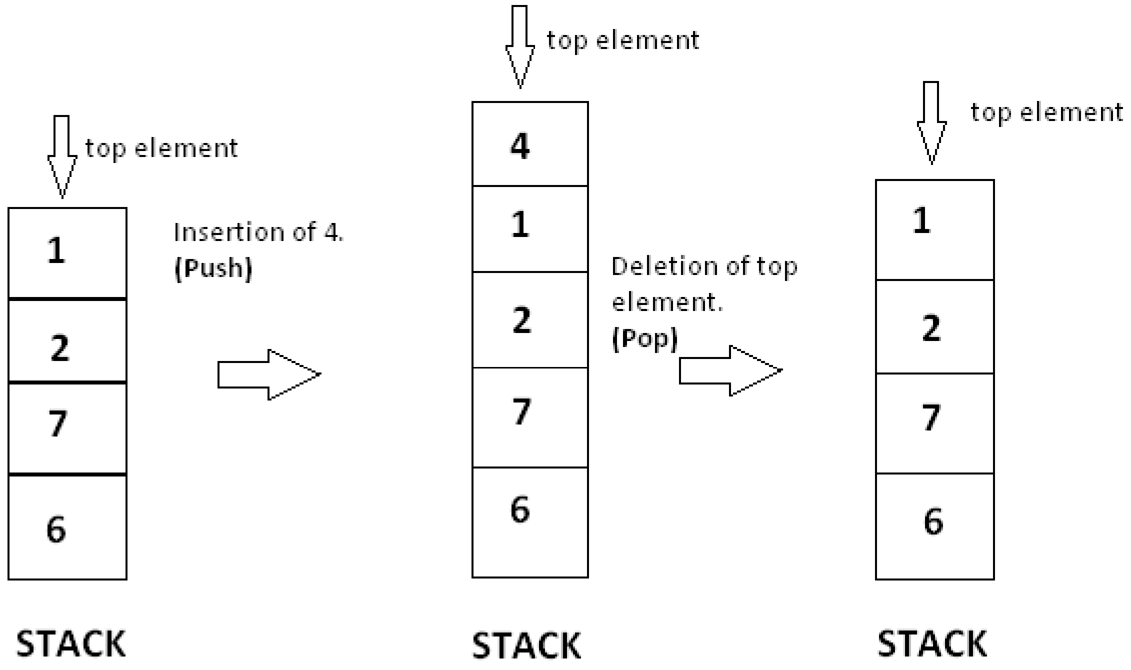
When the top element of a stack is deleted, if the stack remains non-empty, then the element just below the previous top element becomes the new top element of the stack.

For example, in the stack of trays, if you take the tray on the top and do not replace it, then the second tray automatically becomes the top element (tray) of that stack.

Features of stacks

- Dynamic data structures
- Do not have a fixed size
- Do not consume a fixed amount of memory
- Size of stack changes with each push() and pop() operation. Each push() and pop() operation increases and decreases the size of the stack by 1, respectively.

A stack can be visualized as follows:



Operations

push(x): Insert element x at the top of a stack

```
void push (int stack[ ] , int x , int n) {
 if ( top == n-1 ) {            //If the top position is the last of
position in a stack, this means that the stack is full
    cout << "Stack is full.Overflow condition!" ;
    }
    else{
        top = top +1 ;              //Incrementing top position
        stack[ top ] = x ;              //Inserting element on incremented
position
    }
}
```

pop( ): Removes an element from the top of a stack

```
   void pop (int stack[ ] ,int n )
    {

         if( isEmpty ( ) )
         {
             cout << “Stack is empty. Underflow condition! ” << endl ;
         }
         else
         {
              top = top - 1 ; //Decrementing top’s position will detach
last element from stack
         }
    }
```

topElement ( ): Access the top element of a stack

```
  int topElement ( )
    {
         return stack[ top ];
    }
```

isEmpty ( ) : Check whether a stack is empty

```
    bool isEmpty ( )
    {
         if ( top == -1 )  //Stack is empty
         return true ;
         else
         return false;
    }
```

size ( ): Determines the current size of a stack

```
   int size ( )
    {
         return top + 1;
    }
```

Implementation

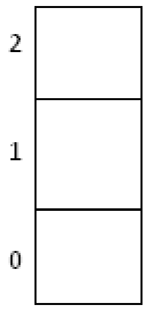
```
#include <iostream>
using namespace std;
int top = -1; //Globally defining the value of top as the stack is empty

    void push (int stack[ ] , int x , int n)
    {
        if ( top == n-1 )         //If the top position is the last of
position of the stack, this means that the stack is full.
        {
            cout << "Stack is full.Overflow condition!" ;
        }
        else
        {
            top = top +1 ;            //Incrementing the top position
            stack[  top  ]  =  x  ;            //Inserting  an  element  on
incremented position
        }
    }
    bool isEmpty ( )
    {
        if ( top == -1 )  //Stack is empty
            return true ;
        else
            return false;
    }
    void pop ( )
    {

        if( isEmpty ( ) )
        {
            cout << "Stack is empty. Underflow condition! " << endl ;
        }
        else
        {
             top = top - 1  ; //Decrementing top’s position will detach
last element from stack
        }
    }
    int size ( )
    {
        return top + 1;
    }
    int topElement (int stack[])
    {
        return stack[ top ];
    }
    //Let's implement these functions on the stack given above

    int main( )
    {
        int stack[ 3 ];
        // pushing element 5 in the stack .
        push(stack , 5 , 3 ) ;

        cout << "Current size of stack is " << size ( ) << endl ;

        push(stack , 10 , 3);
        push (stack , 24 , 3) ;

        cout << "Current size of stack is " << size( ) << endl ;

        //As the stack is full, further pushing will show an overflow
condition.
        push(stack , 12 , 3) ;

        //Accessing the top element
        cout << "The current top element in stack is " <<
topElement(stack) << endl;

        //Removing all the elements from the stack
        for(int i = 0 ; i < 3;i++ )
            pop( );
        cout << "Current size of stack is " << size( ) << endl ;

        //As the stack is empty , further popping will show an underflow
condition.
        pop ( );

    }
```

Output

- Current size of stack: 1
- Current size of stack: 3
- Current top element in stack: 24 (Stack is full. Overflow condition!)
- Current size of stack: 0 (Stack is empty. Underflow condition!)

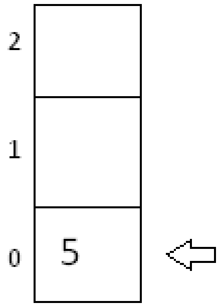
Refer to the following image for more information about the operations performed in the code.

1)



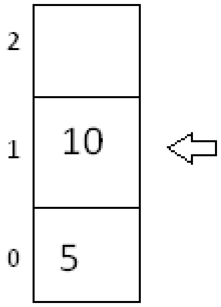
Initially **stack** is empty.
top = -1.

2) **push(stack, 5, 3)**



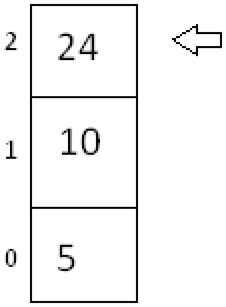
top element
top = 0

3) **push(stack, 10, 3)**



top element
top = 1

4) **push(stack, 24, 3)**



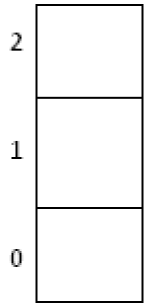
top element
top = 2

5) As **top = 2,** current size of stack is top+1 , i.e 3 . Now **stack** is full,as 3 is maximum size of **stack**

6) **push(stack, 12, 3)**

As ,stack is full ,it will show
OVERFLOW CONDTION!

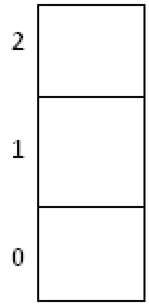
7) Deleting all elements from stack. ⇨ **pop(stack, 3)**
pop(stack, 3)
pop(stack, 3)



EMPTY STACK !!

top = -1

8) **pop(stack, 3)**



As **stack** is empty, further deleting will cause

UNDERFLOW CONDITION!

Application

Consider the balanced parentheses problem.

You have a bracket sequence made up of opening '(' and closing ')' parentheses. You must check if this bracket sequence is balanced.

A bracket sequence is considered balanced if for every prefix of the sequence, the number of opening brackets is greater than or equal to the number of closing brackets, and the total number of opening brackets is equal to the number of closing brackets.

You can check this using stack. Let's see how.

You can maintain a stack where you store a parenthesis. Whenever, you come across an opening parenthesis, push it in the stack. However, whenever you come across a closing parenthesis, pop a parenthesis from the stack.

```
#include <iostream>
using namespace std;
int top;
void  check (char str[ ], int n, char stack [ ])
{
    for(int i = 0 ; i < n ; i++ )
    {
        if (str [ i ] == ‘(’)
        {
            top = top + 1;
            stack[ top ] = ‘ ( ’;
        }
        if(str[ i ] == ‘)’ )
        {
            if(top == -1 )
            {
                top = top -1 ;
                break ;
             }
            else
            {
                 top = top -1 ;
            }
        }
    }
    if(top == -1)
        cout << “String is balanced!” << endl;
    else
```

```
            cout << “String is unbalanced!” << endl ;
    }

    int main ( )
    {
        //balanced parenthesis string.
        char str[  ] = { ‘(‘ , ‘a’ , ‘+’, ‘ ( ’, ‘b ’ , ‘-’ , ‘ c’ ,‘)’ ,
‘ ) ’} ;

        // unbalanced string .
        char str1 [ ] = { ‘(’ , ‘(’ , ‘a’ , ‘ + ’ , ‘ b’ , ‘)’ } ;
        char stack [ 15 ] ;
        top = -1;
        check (str , 9 , stack );      //Passing balanced string
        top = -1 ;
        check(str1 , 5 , stack) ;    //Passing unbalanced string
        return 0;

    }
```

Data Structure & Algorithms

Problem Solving with Stack

Problem Solving with Stacks

- Many mathematical statements contain nested parenthesis like :-
  - (A+(B*C)) + (C – (D + F))
- We have to ensure that the parenthesis are nested correctly, i.e. :-
  1. There is an equal number of left and right parenthesis
  2. Every right parenthesis is preceded by a left parenthesis
- Expressions such as ((A + B) violate condition 1
- And expressions like) A + B (- C violate condition 2

Problem Solving (Cont….)

- To solve this problem, think of each left parenthesis as opening a scope, right parenthesis as closing a scope
- Nesting depth at a particular point in an expression is the number of scopes that have been opened but not yet closed
- Let “parenthesis count” be a variable containing number of left parenthesis minus number of right parenthesis, in scanning the expression from left to right

Problem Solving (Cont….)

- For an expression to be of a correct form following conditions apply
  - Parenthesis count at the end of an expression must be 0
  - Parenthesis count should always be non-negative while scanning an expression
- Example :
  - Expr: 7 – (A + B) + ((C – D) + F)
  - ParenthesisCount: 0 0 1 1 1 1 0 0 1 2 2 2 2 1 1 1 0
  - Expr: 7 – ((A + B) + ((C – D) + F)
  - ParenthesisCount: 0 0 1 2 2 2 2 1 1 23 3 3 3 2 2 21

Problem Solving (Cont….)

- Evaluating the correctness of simple expressions like this one can easily be done with the help of a few variables like "Parenthesis count"
- Things start getting difficult to handle by your program when the requirements get complicated e.g.
- Let us change the problem by introducing three different types of scope de-limiters i.e. (parenthesis), {braces} and [brackets].
- In such a situation we must keep track of not only the number of scope delimiters but also their types
- When a scope ender is encountered while scanning an expression, we must know the scope delimiter type with which the scope was opened
- We can use a stack ADT to solve this problem

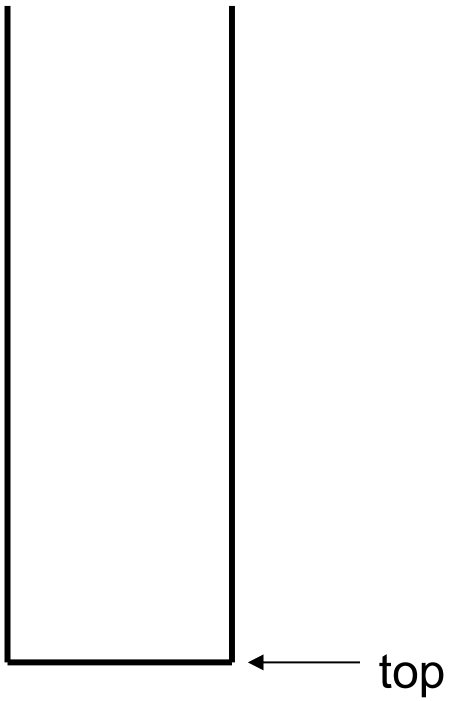
Problem Solving with Stack

- A stack ADT can be used to keep track of the scope delimiters encountered while scanning the expression
- Whenever a scope “opener” is encountered, it can be “pushed” onto a stack
- Whenever a scope “ender” is encountered, the stack is examined:
  - If the stack is “empty”, there is no matching scope “opener” and the expression is invalid.
  - If the stack is not empty, we pop the stack and check if the “popped” item corresponds to the scope ender
  - If match occurs, we continue scanning the expression
- When end of the expression string is reached, the stack must be empty, otherwise one or more opened scopes have not been closed and the expression is invalid

Why the need for a Stack

- Last scope to be opened must be the first one to be closed.
- This scenario is simulated by a stack in which the last element arriving must be the first one to leave
- Each item on the stack represents a scope that has been opened but has yet not been closed
- Pushing an item on to the stack corresponds to opening a scope
- Popping an item from the stack corresponds to closing a scope, leaving one less scope open

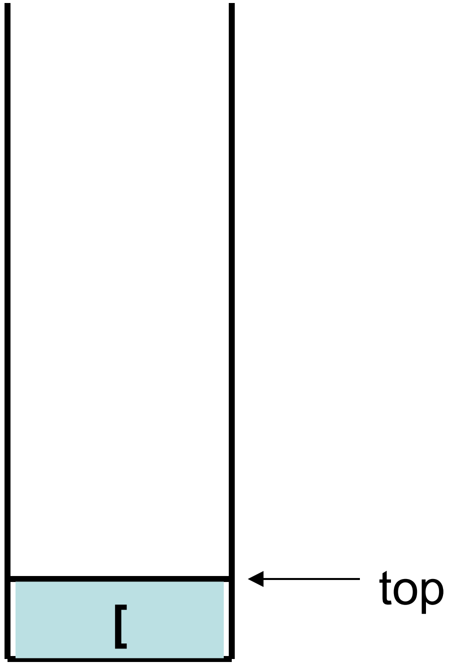
Stack in Action ….



[A+ { B - C + (D + E) }]

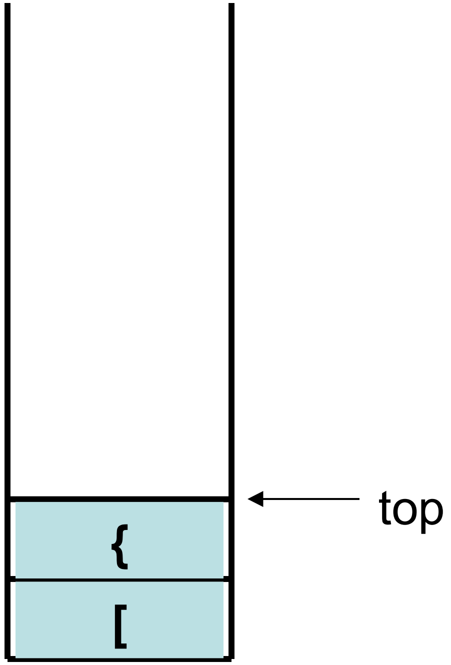
Stack in Action ….

Push(‘[‘);



[A+ { B - C + (D + E) }]

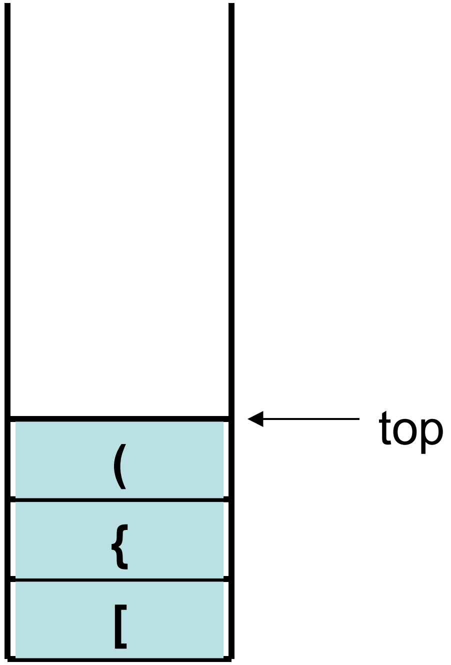
Stack in Action ….



Push(‘{‘);

[A+ { B - C + (D + E) }]

Stack in Action ….



top

(

{

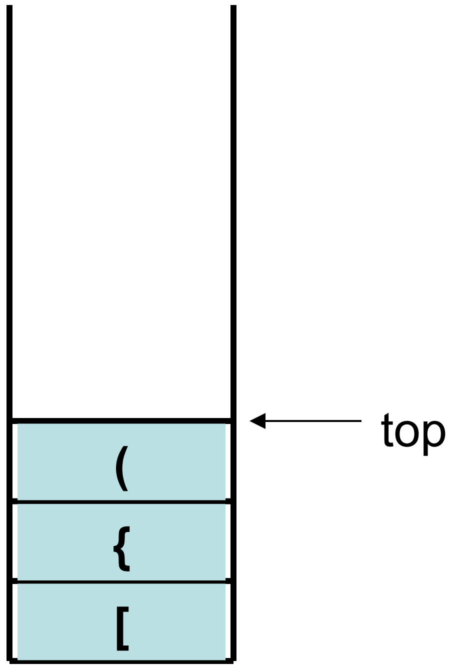
[

Push(‘(‘);

[A+ { B - C + (D + E) }]

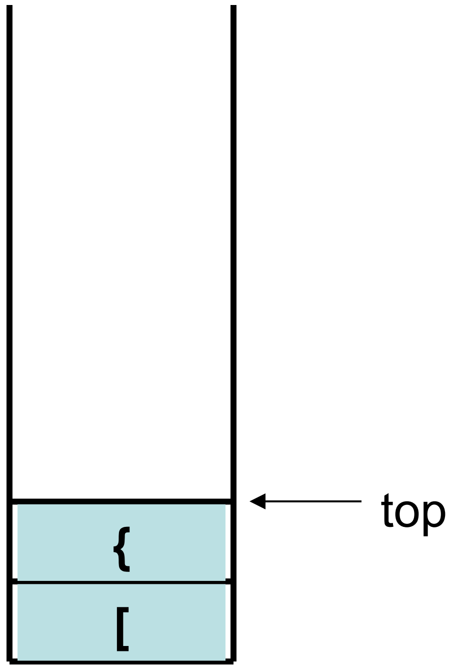
Stack in Action ….

Pop( );



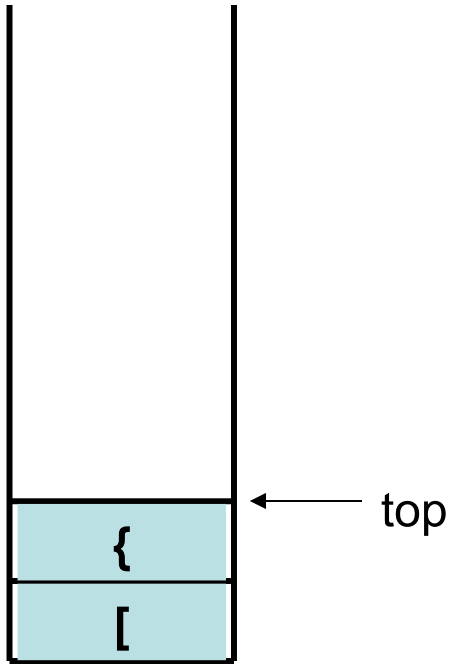
[A+ { B - C + (D + E) }]

Stack in Action ….



[A+ { B - C + (D + E) }]

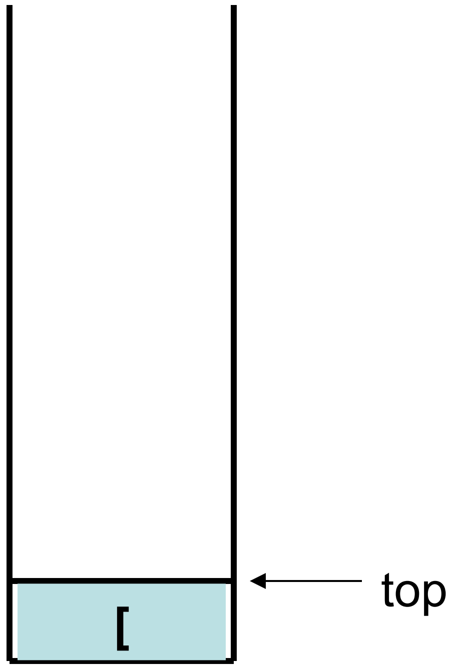
Stack in Action ….



Pop( );

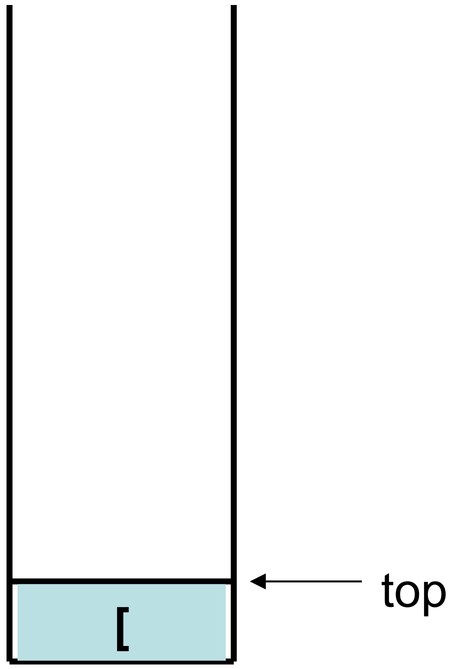
[A+ { B - C + (D + E) }]

Stack in Action ....



[A+ { B - C + (D + E) }]

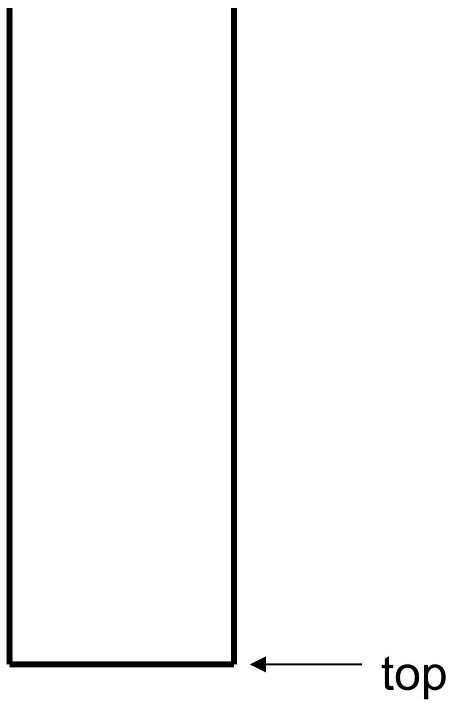
Stack in Action ….



Pop( );

[A+ { B - C + (D + E) }]

Stack in Action ....



[A+ { B - C + (D + E) }]

Result = A valid expression

Infix, Prefix and Postfix Notations

Infix, Postfix and Prefix Notations

- The usual way of expressing the sum of two numbers A and B is :

  A+B

- The operator '+' is placed between the two operands A and B
- This is called the "***Infix Notation***"
- Consider a bit more complex example:

  (13 – 5) / (3 + 1)

- When the parentheses are removed the situation becomes ambiguous

  13 – 5 / 3 + 1

  is it (13 – 5) / (3 + 1)

  or 13 – (5 / 3) + 1

- To cater for such ambiguity, you must have operator precedence rules to follow (as in C++)

Infix, Postfix and Prefix Notations

- In the absence of parentheses

  13 – 5 / 3 + 1
- Will be evaluated as 13 – (5 / 3) + 1
- Operator precedence is **by-passed** with the help of parentheses as in (13 – 5) / (3 + 1)
- The infix notation is therefore cumbersome due to
  - Operator Precedence rules and
  - Evaluation of Parentheses

Postfix Notation

- It is a notation for writing arithmetic expressions in which operands appear before the operator
- E.g. A + B is written as A B + in postfix notation
- There are no precedence rules to be learnt in it.
- Parentheses are never needed
- Due to its simplicity, some calculators use postfix notation
- This is also called the “Reverse Polish Notation or RPN”

Postfix Notation – Some examples

Infix Expressions	Corresponding Postfix
5 + 3 + 4 + 1	5 3 + 4 + 1 +
(5 + 3) * 10	5 3 + 10 *
(5 + 3) * (10 – 4)	5 3 + 10 4 - *
5 * 3 / (7 – 8)	5 3 * 7 8 - /
(b * b – 4 * a * c) / (2 * a)	b b * 4 a * c * - 2 a * /

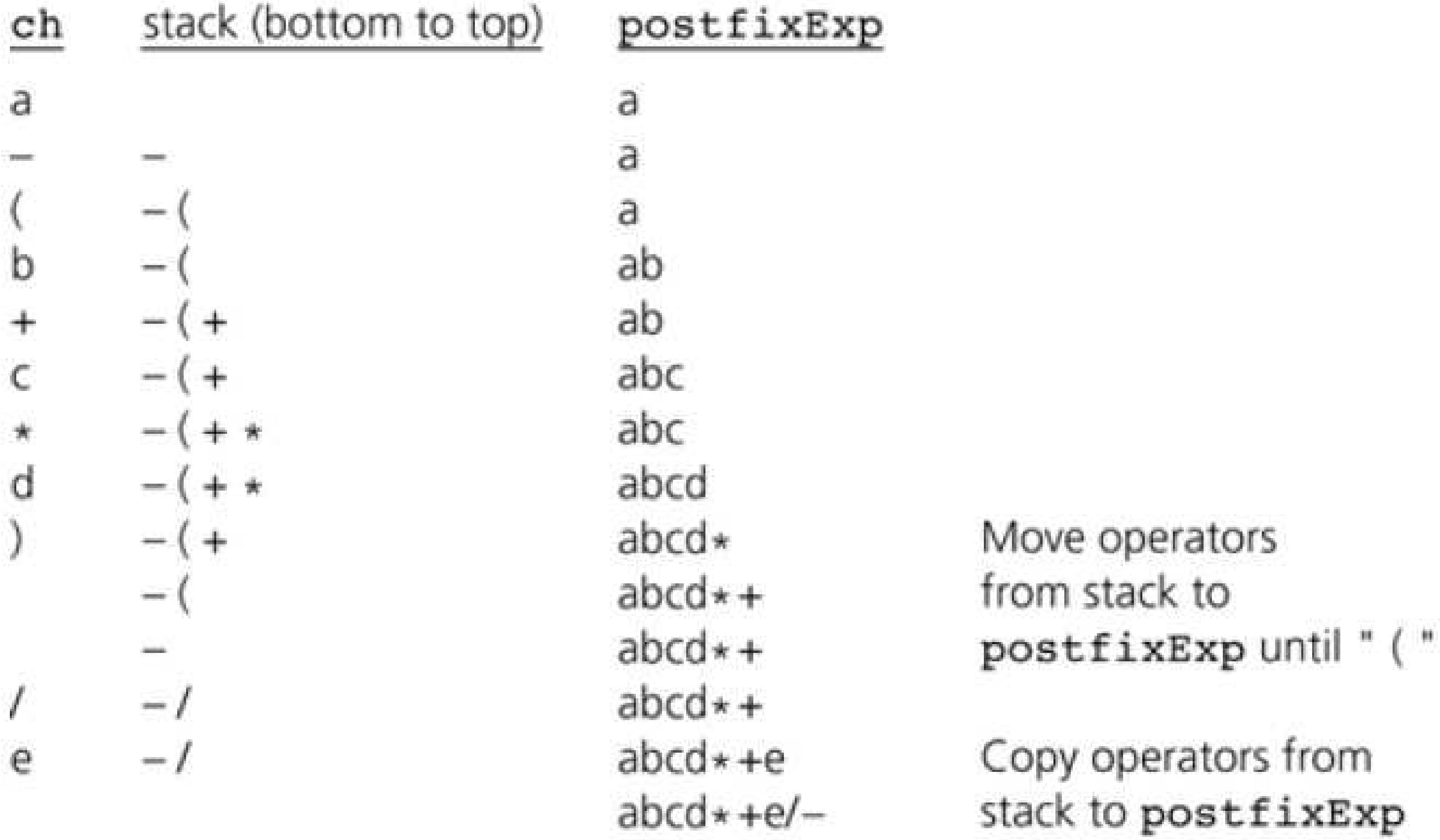
Conversion from Infix to Postfix Notation

- We have to accommodate the presence of operator precedence rules and Parentheses while converting from infix to postfix
- Data objects required for the conversion are
  - An operator / parentheses stack
  - A Postfix expression string to store the resultant
  - An infix expression string read one item at a time

Conversion from Infix to Postfix

- The Algorithm
  - What are possible items in an input Infix expression
  - Read an item from input infix expression
  - If item is an operand append it to postfix string
  - If item is “(“ push it on the stack
  - If the item is an operator
    - If the operator has higher precedence than the one already on top of the stack then push it onto the operator stack
    - If the operator has lower precedence than the one already on top of the stack then
      - pop the operator on top of the operator stack and append it to postfix string, and
      - push lower precedence operator onto the stack
  - If item is “)” pop all operators from top of the stack one-by-one, until a “(“ is encountered on stack and removed
  - If end of infix string pop the stack one-by-one and append to postfix string

Converting Infix Expressions to Equivalent Postfix Expressions



ch	stack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators from stack to postfixExp until " ("
	– (	abcd*+	
	–	abcd*+	
/	– /	abcd*+	
e	– /	abcd*+e	Copy operators from stack to postfixExp
		abcd*+e/–	

A trace of the algorithm that converts the infix expression *a* - (*b* + *c* * *d*)/*e* to postfix form

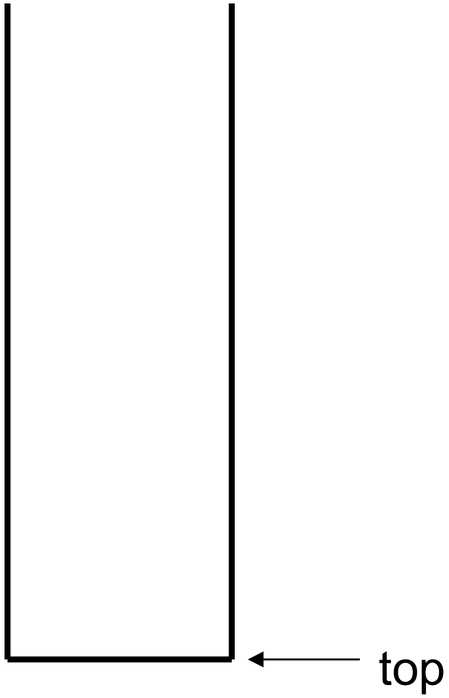
Try it yourself

- Show a trace of algorithm that converts the infix expression
  - (X + Y) * (P – Q / L)
  - L – M / (N * O ^ P)

Evaluation of Postfix Expression

- After an infix expression is converted to postfix, its evaluation is a simple affair
- Stack comes in handy, AGAIN
- The Algorithm
  - Read the postfix expression one item at-a-time
  - If item is an operand push it on to the stack
  - If item is an operator pop the top two operands from stack and apply the operator
  - Push the result back on top of the stack, which will become an operand for next operation
  - Final result will be the only item left on top of the stack

Stack in Action ….



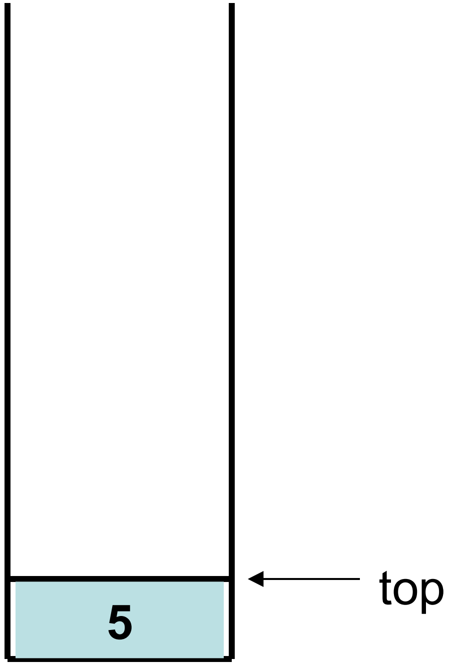
Postfix Expression

5 7 + 6 2 - *

Stack in Action ….

Postfix Expression

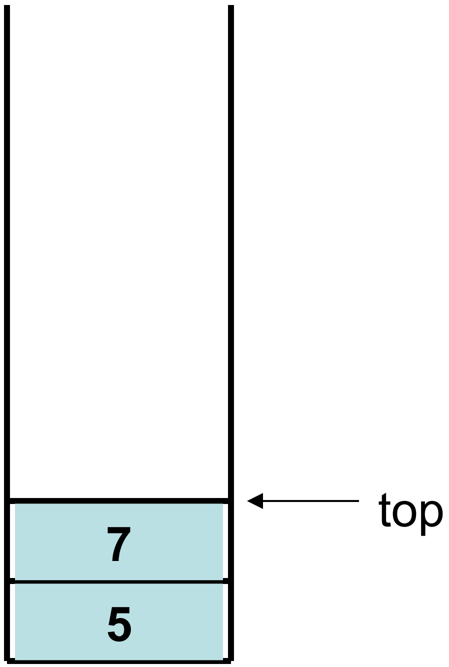
5 7 + 6 2 - *



Stack in Action ….

Postfix Expression

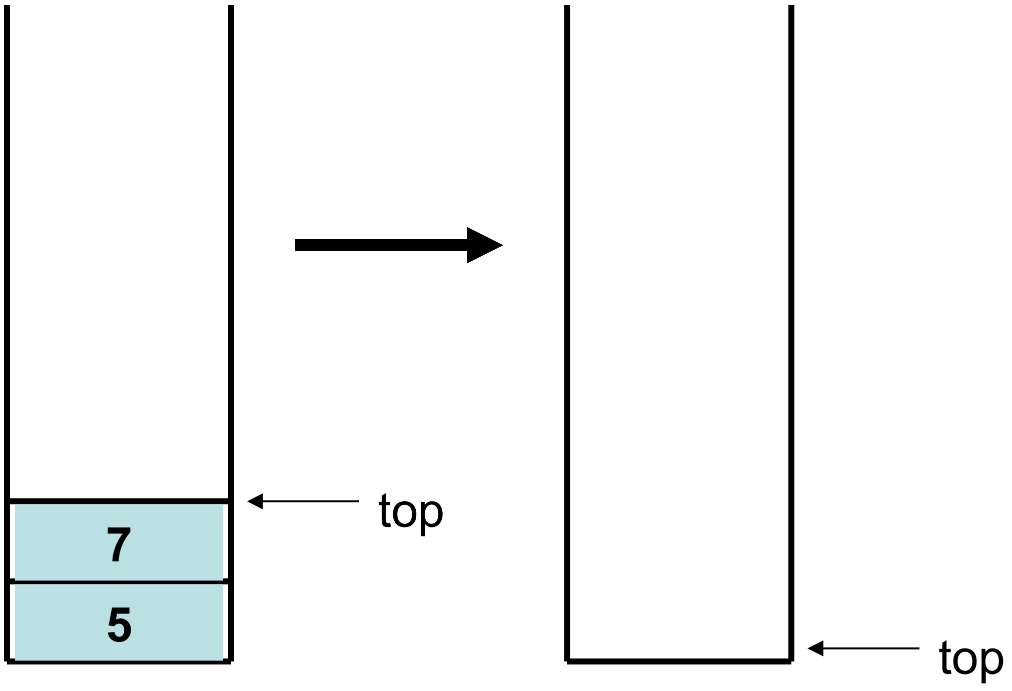
5 7 + 6 2 - *



Stack in Action ….

Postfix Expression

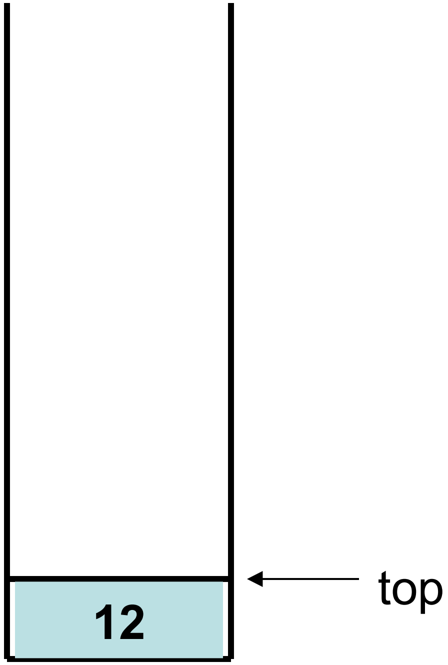
5 7 + 6 2 - *



Result = Pop( ) “+” Pop( )

Push (Result)

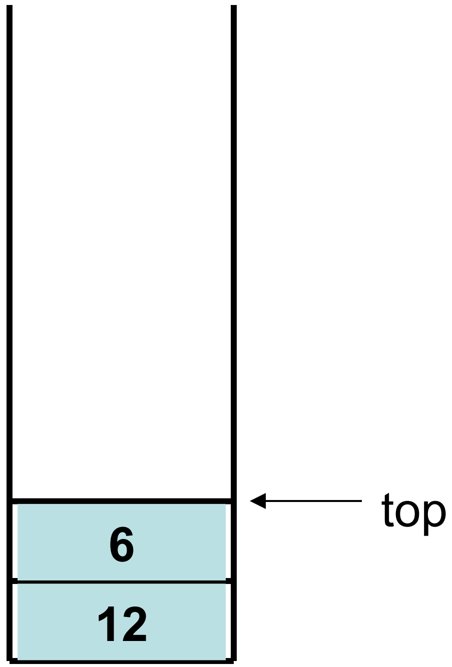
Stack in Action ….



Postfix Expression

5 7 + 6 2 - *

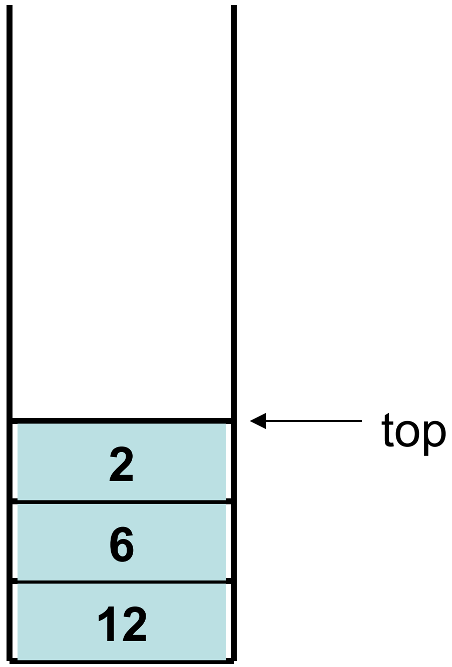
Stack in Action ….



Postfix Expression

5 7 + 6 2 - *

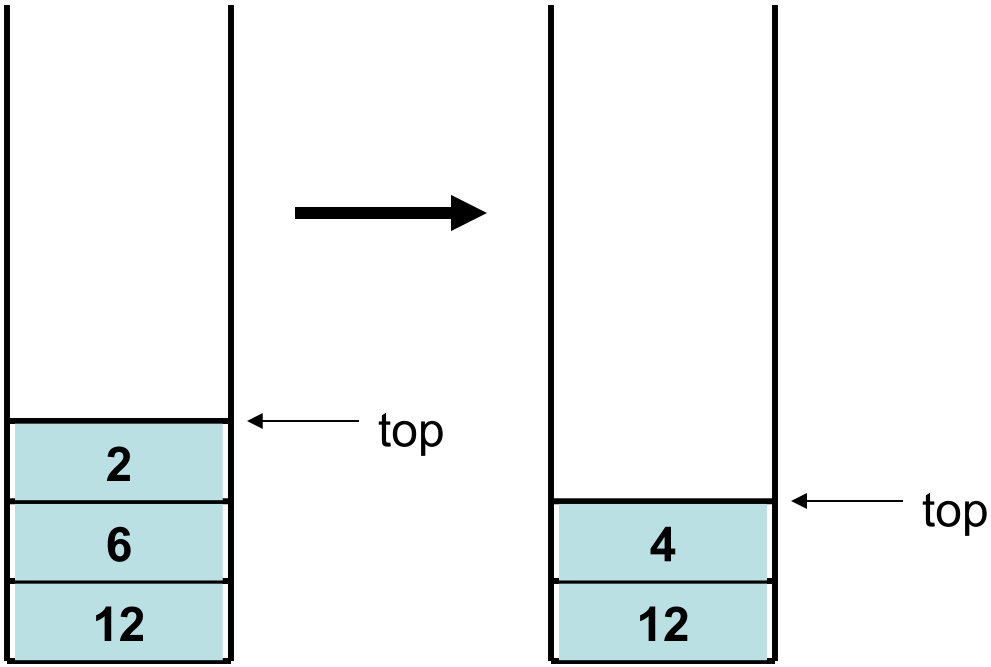
Stack in Action ….



Postfix Expression

5 7 + 6 2 - *

Stack in Action ….



2

6

12

top

4

12

top

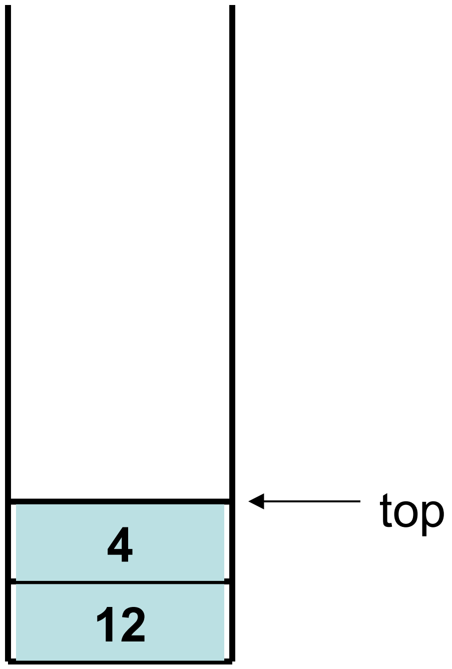
Postfix Expression

5 7 + 6 2 - *

Result = Pop( ) “-” Pop( )

Push (Result)

<u>Stack in Action ….</u>

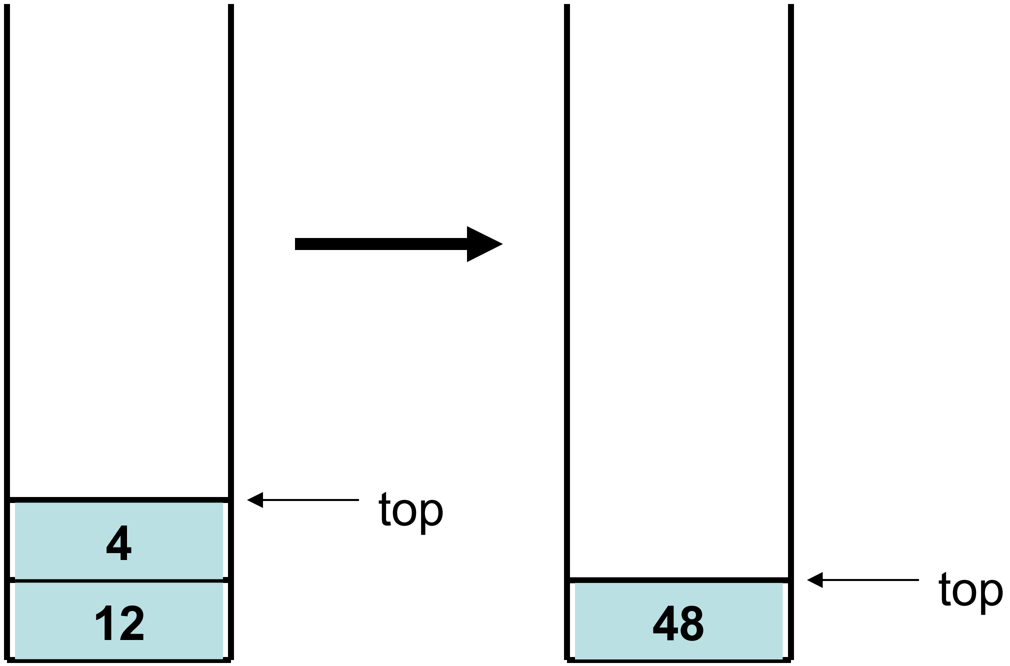


<u>Postfix Expression</u>

5 7 + 6 2 - *

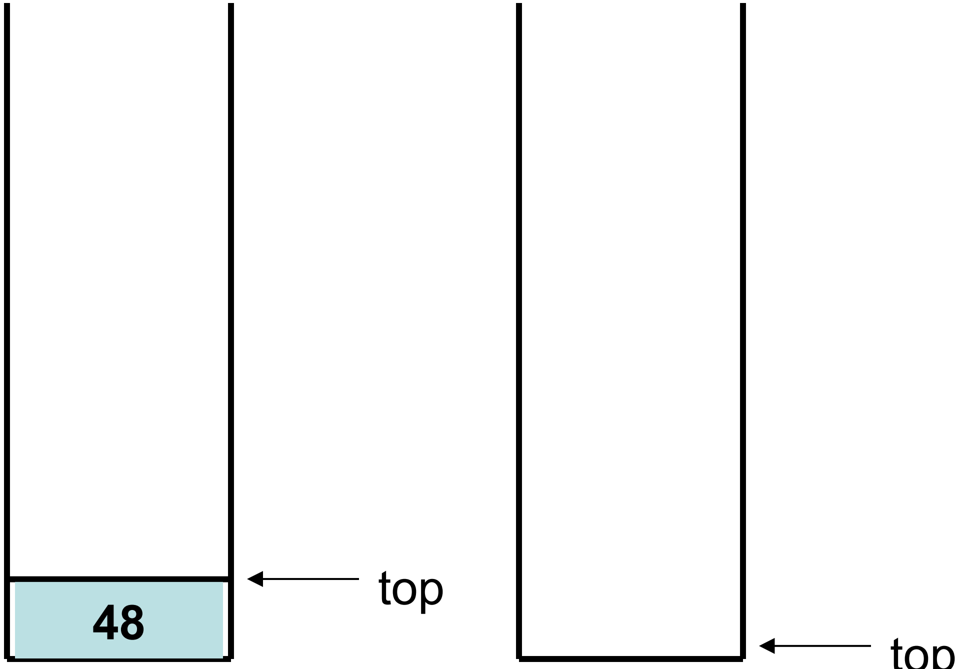
Postfix Expression

5 7 + 6 2 - *



Result = Pop( ) “ * ” Pop( )

Push (Result)



Postfix Expression

5 7 + 6 2 - *

Result = Pop( )

Result = 48

Evaluation of Postfix Expression

- Evaluation of infix Expression is difficult because :
  - Rules governing the precedence of operators are to be catered for
  - Many possibilities for incoming characters
  - To cater for parentheses
  - To cater for error conditions / checks
- Evaluation of postfix expression is very simple to implement because operators appear in precisely the order in which they are to be executed

Motivation for the conversion

- Motivation for this conversion is the need to have the operators in the precise order for execution
- While using paper and pencil to do the conversion we can “foresee” the expression string and the depth of all the scopes (if the expressions are not very long and complicated)
- When a program is required to evaluate an expression, it must be accurate
- At any time during scanning of an expression we cannot be sure that we have reached the inner most scope
- Encountering an operator or parentheses may require frequent “backtracking”
- Rather than backtracking, we use the stack to **“remember”** the operators encountered previously

Assignment # 1

- Write a program that gets an Infix arithmetic expression and converts it into postfix notation
- The program should then evaluate the postfix expression and output the result
- Your program should define the input and output format, enforce the format and handle Exceptions (exceptional conditions).
- Use appropriate comments at every stage of programming

Data Structures and Algorithms V22.0102

Otávio Braga

Infix to Postfix Conversion

- We use a stack
- When an operand is read, output it
- When an operator is read
  - Pop until the top of the stack has an element of lower precedence
  - Then push it
- When) is found, pop until we find the matching (
- (has the lowest precedence when in the stack
- but has the highest precedence when in the input
- When we reach the end of input, pop until the stack is empty

Infix to Postfix Conversion Example 1

- 3+4*5/6

Infix to Postfix Conversion Example 1

- 3+4*5/6
- Stack:
- Output:

Infix to Postfix Conversion Example 1

- 3+4*5/6
- Stack:
- Output: 3

Infix to Postfix Conversion Example 1

- 3+4*5/6
- Stack: +
- Output: 3

Infix to Postfix Conversion Example 1

- 3+4*5/6
- Stack: +
- Output: 3 4

Infix to Postfix Conversion
Example 1

- 3+4*5/6
- Stack: + *
- Output: 3 4

Infix to Postfix Conversion Example 1

- 3+4*5/6
- Stack: + *
- Output: 3 4 5

Infix to Postfix Conversion Example 1

- 3+4*5/6
- Stack: +
- Output: 3 4 5 *

Infix to Postfix Conversion Example 1

- 3+4*5/6
- Stack: + /
- Output: 3 4 5 *

Infix to Postfix Conversion Example 1

- 3+4*5/6
- Stack: + /
- Output: 3 4 5 * 6

Infix to Postfix Conversion Example 1

- 3+4*5/6
- Stack: +
- Output: 3 4 5 * 6 /

Infix to Postfix Conversion
Example 1

- 3+4*5/6
- Stack:
- Output: 3 4 5 * 6 / +

- Done!

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack:
- Output:

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack: (
- Output:

Infix to Postfix Conversion
Example 2

- (300+23)*(43-21)/(84+7)
- Stack: (
- Output: 300

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack: (+
- Output: 300

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack: (+
- Output: 300 23

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack: (
- Output: 300 23 +

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack:
- Output: 300 23 +

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack: *
- Output: 300 23 +

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack: * (
- Output: 300 23 +

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack: * (
- Output: 300 23 + 43

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack: * (-
- Output: 300 23 + 43

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack: * (-
- Output: 300 23 + 43 21

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack: * (
- Output: 300 23 + 43 21 -

Infix to Postfix Conversion
Example 2

- (300+23)*(43-21)/(84+7)
- Stack: *
- Output: 300 23 + 43 21 -

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack:
- Output: 300 23 + 43 21 - *

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack: /
- Output: 300 23 + 43 21 - *

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack: / (
- Output: 300 23 + 43 21 - *

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84$\underline{+}$7)
- Stack: / (
- Output: 300 23 + 43 21 - * 84

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack: / (+
- Output: 300 23 + 43 21 - * 84

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack: / (+
- Output: 300 23 + 43 21 - * 84 7

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack: / (
- Output: 300 23 + 43 21 - * 84 7 +

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack: /
- Output: 300 23 + 43 21 - * 84 7 +

Infix to Postfix Conversion Example 2

- (300+23)*(43-21)/(84+7)
- Stack:
- Output: 300 23 + 43 21 - * 84 7 + /

- Done!

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack:
- Output:

Infix to Postfix Conversion
Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: (
- Output:

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: (
- Output: 4

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: (+
- Output: 4

Infix to Postfix Conversion
Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: (+
- Output: 4 8

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: (
- Output: 4 8 +

Infix to Postfix Conversion
Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack:
- Output: 4 8 +

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: *
- Output: 4 8 +

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: * (
- Output: 4 8 +

Infix to Postfix Conversion
Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: * (
- Output: 4 8 + 6

Infix to Postfix Conversion
Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: * (-
- Output: 4 8 + 6

Infix to Postfix Conversion
Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: * (-
- Output: 4 8 + 6 5

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: * (
- Output: 4 8 + 6 5 -

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: *
- Output: 4 8 + 6 5 -

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack:
- Output: 4 8 + 6 5 - *

Infix to Postfix Conversion
Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: /
- Output: 4 8 + 6 5 - *

Infix to Postfix Conversion
Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: / (
- Output: 4 8 + 6 5 - *

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: / ((
- Output: 4 8 + 6 5 - *

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: / ((
- Output: 4 8 + 6 5 - * 3

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-$\underline{2}$)*(2+2))
- Stack: / ((-
- Output: 4 8 + 6 5 - * 3

Infix to Postfix Conversion
Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: / ((-
- Output: 4 8 + 6 5 - * 3 2

Infix to Postfix Conversion
Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: / ((
- Output: 4 8 + 6 5 - * 3 2 -

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: / (
- Output: 4 8 + 6 5 - * 3 2 -

Infix to Postfix Conversion
Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: / (*
- Output: 4 8 + 6 5 - * 3 2 -

Infix to Postfix Conversion
Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: / (* (
- Output: 4 8 + 6 5 - * 3 2 -

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2$\underline{+}$2))
- Stack: / (* (
- Output: 4 8 + 6 5 - * 3 2 - 2

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: / (* (+
- Output: 4 8 + 6 5 - * 3 2 - 2

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: / (* (+
- Output: 4 8 + 6 5 - * 3 2 – 2 2

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: / (* (
- Output: 4 8 + 6 5 - * 3 2 – 2 2 +

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: / (*
- Output: 4 8 + 6 5 - * 3 2 – 2 2 +

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: / (
- Output: 4 8 + 6 5 - * 3 2 – 2 2 + *

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack: /
- Output: 4 8 + 6 5 - * 3 2 – 2 2 + *

Infix to Postfix Conversion Example 3

- (4+8)*(6-5)/((3-2)*(2+2))
- Stack:
- Output: 4 8 + 6 5 - * 3 2 – 2 2 + * /

- Done!

Recursion

Introduction

- In C++, any function can call another function.
- A function can even call itself.
- When a function call itself, it is making a recursive call.
- **Recursive Call**
  - **A function call in which the function being called is the same as the one making the call**.
- Recursion is a powerful technique that can be used in the place of iteration(looping).
- **Recursion**
  - **Recursion is a programming technique in which procedures and functions call themselves.**

Recursive Algorithm

- **Definition**
  - An algorithm that calls itself
- **Approach**
  - Solve small problem directly
  - Simplify large problem into 1 or more smaller subproblem(s) & solve recursively
  - Calculate solution from solution(s) for subproblem

Recursive Definition

- **A definition in which something is defined in terms of smaller versions of itself.**
- To do recursion we should know the followings
  - **Base Case:**
    - The case for which the solution can be stated non-recursively
    - The case for which the answer is explicitly known.
  - **General Case:**
    - The case for which the solution is expressed in smaller version of itself. Also known as recursive case.

Example 1

- We can write a function called **power** that calculates the result of raising an integer to a positive power. If X is an integer and N is a positive integer, the formula for X^N is

$$X^N = \underbrace{X * X * X * X * X * \cdots * X}_{N - times}$$

- We can also write this formula as

$$X^N = X * \underbrace{X * X * X * X * \cdots * X}_{(N - 1)\_times}$$

- Also as

$$X^N = X * X * \underbrace{X * X * X * \cdots \cdots * X}_{(N - 2)\_times}$$

- In fact we can write this formula as

$$X^N = X * X^{N - 1}$$

Example 1(Recursive Power Function)

- Now lets suppose that X=3 and N=4

$$X^N \qquad 3^4$$

- Now we can simplify the above equation as

$$3^4 \qquad 3*3^3$$

$$3^3 \qquad 3*3^2$$

$$3^2 \qquad 3*3^1$$

- So the base case in this equation is

$$3^1 \qquad 3$$

```
int Power ( int  x , int  n )
{
     if ( n == 1 )
          return x;     //Base case
     else
        return x * Power (x, n-1);
                    // recursive call
}
```

Example 2 (Recursive Factorial Function)

Factorial Function:

Given a +ive integer n, n factorial is defined as the product of all integers between 1 and n, including n.

So we can write factorial function mathematically as

$$f(n) \qquad \begin{array}{cc} 1 & \text{if } n \quad 0 \\ n \quad f(n^{\tilde{}} \quad 1) & else \end{array}$$

Example 2 (Recursive Factorial Function)

- **Factorial definition**

  n! = n × n-1 × n-2 × n-3 × … × 3 × 2 × 1

  0! = 1

- **To calculate factorial of n**
  - Base case
    - If n = 0, return 1
  - Recursive step
    - Calculate the factorial of n-1
    - Return n × (the factorial of n-1)

Example 2 (Recursive Factorial Function)

```
int factorial(n)
    {
        if(n == 0)                          //Base Case
            return 1;
        else
            return  n * factorial(n-1);    //Recursion
    }
```

Evaluation of Factorial Example

To evaluate Factorial(3)

evaluate 3 * Factorial(2)

To evaluate Factorial(2)

evaluate 2 * Factorial(1)

To evaluate Factorial(1)

evaluate 1 * Factorial(0)

Factorial(0) is 1

Return 1

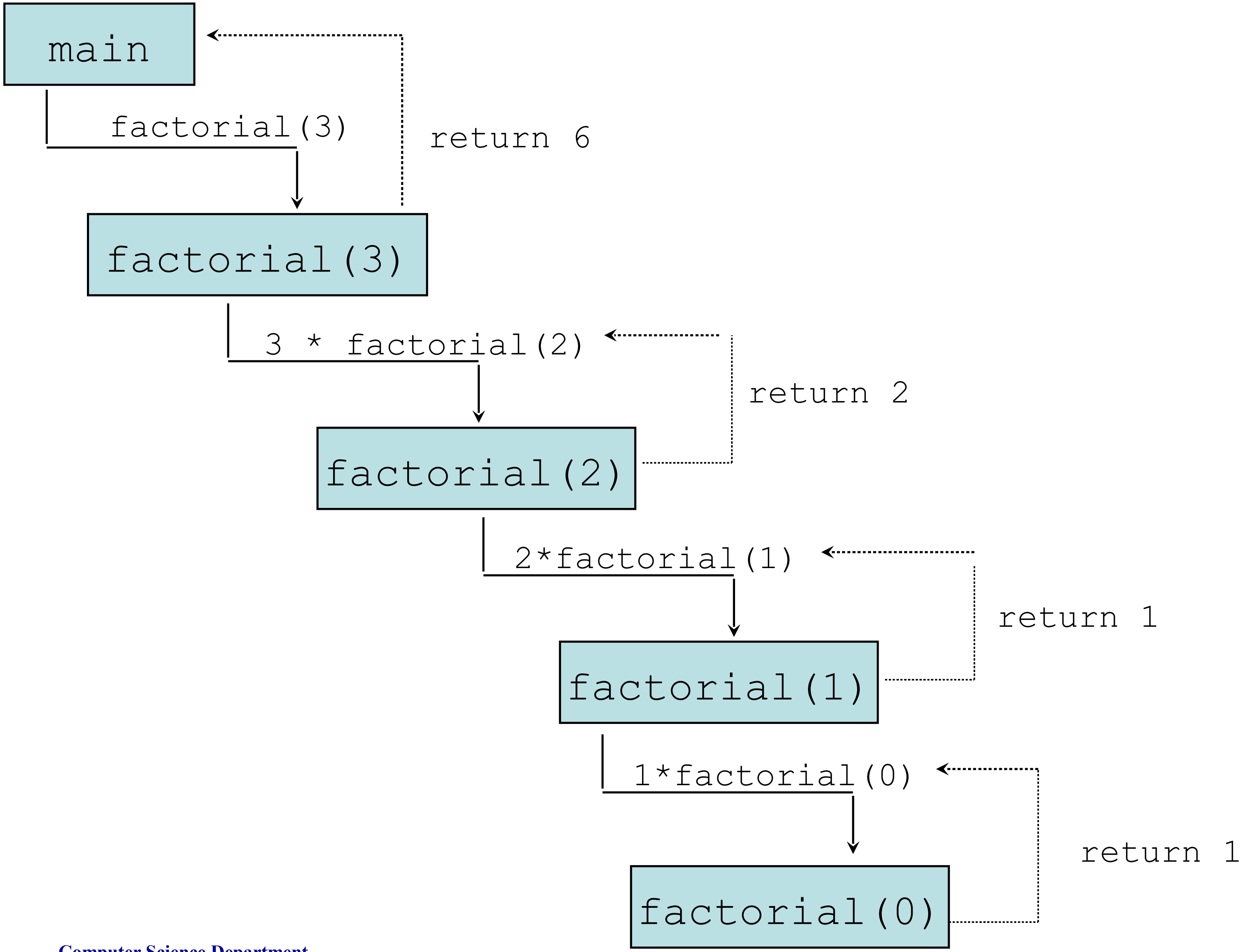
Evaluate 1 * 1

Return 1

Evaluate 2 * 1

Return 2

Recursive Programming



main

factorial(3)

return 6

factorial(3)

3 * factorial(2)

return 2

factorial(2)

2*factorial(1)

return 1

factorial(1)

1*factorial(0)

return 1

factorial(0)

Rules For Recursive Function

1. In recursion, it is essential for a function to call itself, otherwise recursion will not take place.
2. Only user define function can be involved in recursion.
3. To stop the recursive function it is necessary to base the recursion on test condition and proper terminating statement such as *exit()* or *return* must be written using if() statement.
4. When a recursive function is executed, the recursive calls are not implemented instantly. All the recursive calls are pushed onto the stack until the terminating condition is not detected, the recursive calls stored in the stack are popped and executed.
5. During recursion, at each recursive call new memory is allocated to all the local variables of the recursive functions with the same name.

The Runtime Stack during Recursion

- To understand how recursion works at run time, we need to understand what happens when a function is called.

- Whenever a function is called, a block of memory is allocated to it in a run-time structure called the **stack.**

- This block of memory will contain
  - the function's **local variables,**
  - local copies of the function's **call-by-value parameters,**
  - pointers to its **call-by-reference parameters**, and
  - a **return address,** in other words where in the program the function was called from. When the function finishes, the program will continue to execute from that point.

Exercise

1. The problem of computing the sum of all the numbers between 1 and any positive integer N can be recursively defined as:

$$\sum_{i=1}^{N} = N + \sum_{i=1}^{N-1} = N + (N-1) + \sum_{i=1}^{N-2}$$

$$= \quad \text{etc.}$$

Exercise

```
int sum(int n)
{
if(n==1)
    return n;
else
    return n + sum(n-1);
}
```

Recursion in ADT

- You can also use recursion in the data structures such as Stacks, queues, linked list, trees etc.
- **Task**
  - Write a recursive function Search. Which will search the given number in the linked list.
  - Steps involved in searching
    - **Base case**
      - If list is empty or search reached end of the list, return false
      - If number is found in the list, return true
    - **Recursive step**
      - Perform recursion for the next node in the list until you find the required element.

Tail Recursion

- A recursive function is called **tail recursive** if only one recursive call appears in the function and that recursive call is the last statement in the function body.

Recursion

Motivations

Suppose you want to find all the files under a directory that contains a particular word. How do you solve this problem? There are several ways to solve this problem. An intuitive solution is to use recursion by searching the files in the subdirectories recursively.

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

n! = n * (n-1)!

0! = 1

ComputeFactorial

Run

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4)

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

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

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))

Computing Factorial

$$\text{factorial}(0) = 1;$$

$$\text{factorial}(n) = n*\text{factorial}(n-1);$$

$$\begin{aligned}
\text{factorial}(4) &= 4 * \text{factorial}(3) \\
&= 4 * 3 * \text{factorial}(2) \\
&= 4 * 3 * (2 * \text{factorial}(1)) \\
&= 4 * 3 * (\ 2 * (1 * \text{factorial}(0))) \\
&= 4 * 3 * (\ 2 * (\ 1 * 1)))
\end{aligned}$$

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

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

= 4 * 3 * (2 * (1 * factorial(0)))

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

= 4 * 3 * (2 * 1)

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))
= 4 * 3 * (2 * (1 * 1)))
= 4 * 3 * (2 * 1)
= 4 * 3 * 2

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

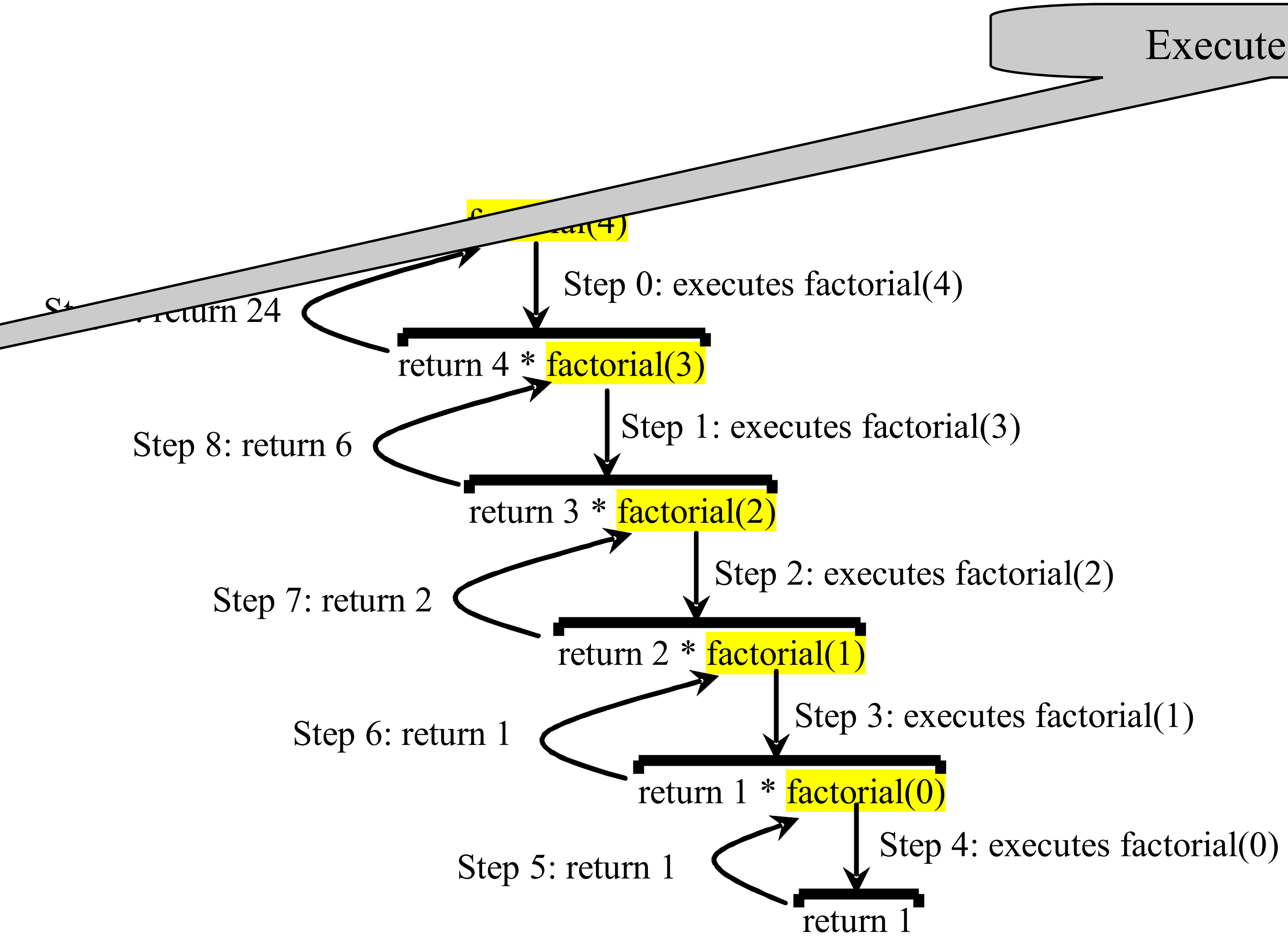
factorial(4) = 4 * factorial(3)
= 4 * (3 * factorial(2))
= 4 * (3 * (2 * factorial(1)))
= 4 * (3 * (2 * (1 * factorial(0))))
= 4 * (3 * (2 * (1 * 1))))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * (6)

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

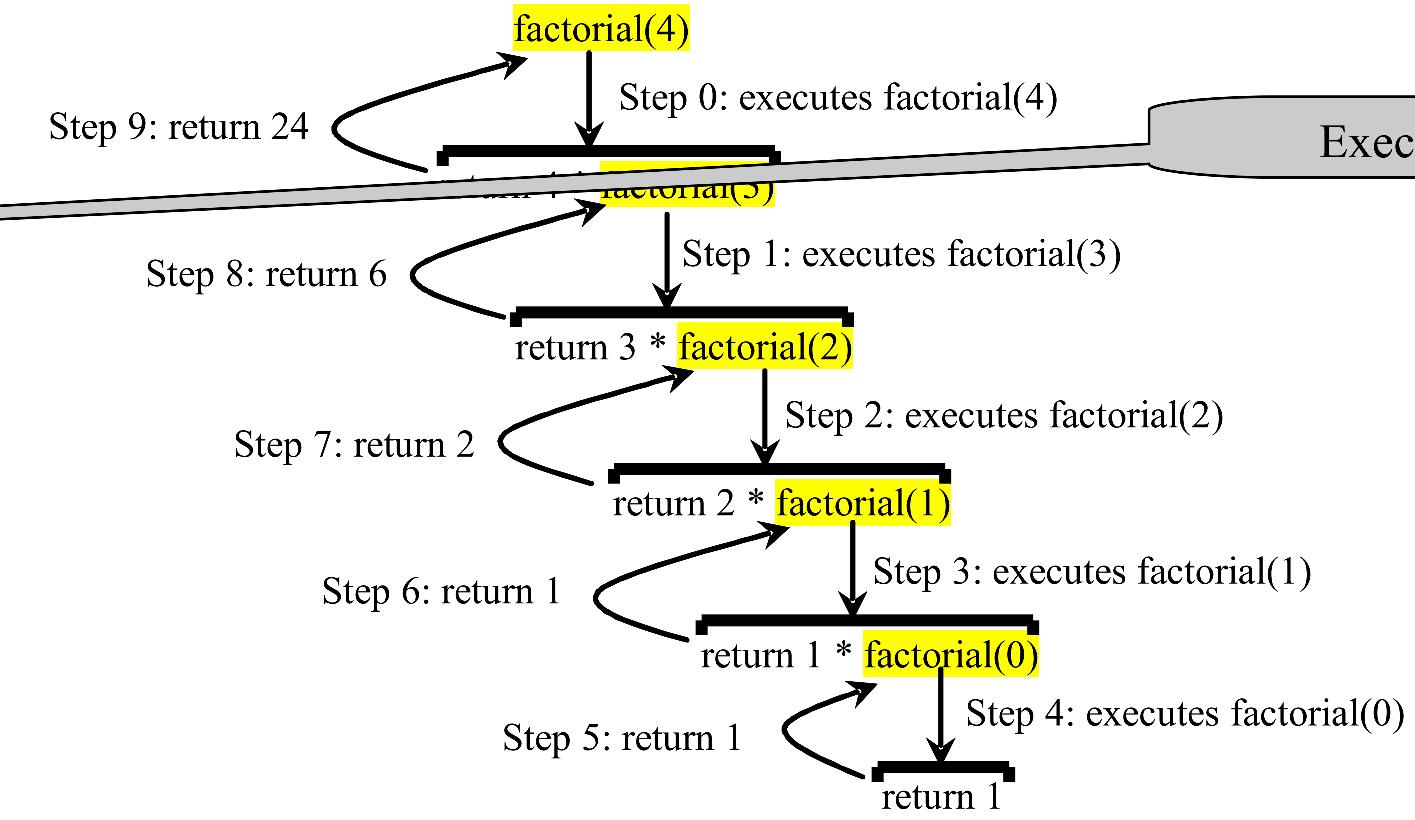
factorial(4) = 4 * factorial(3)
= 4 * (3 * factorial(2))
= 4 * (3 * (2 * factorial(1)))
= 4 * (3 * (2 * (1 * factorial(0))))
= 4 * (3 * (2 * (1 * 1))))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * (6)
= 24

Trace Recursive factorial



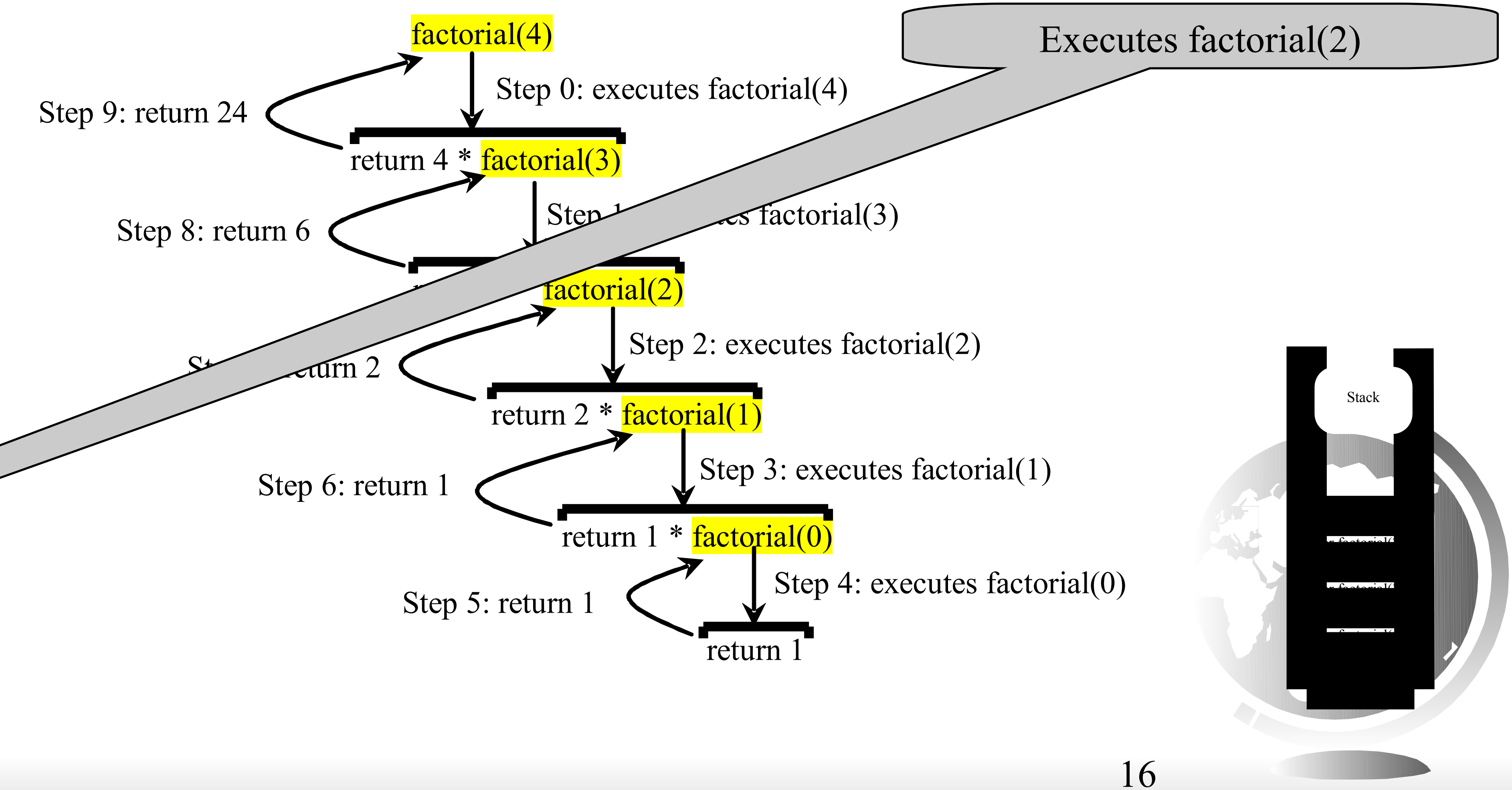
Executes factorial(4)

Trace Recursive factorial



Executes factorial(3)

Trace Recursive factorial



Executes factorial(2)

factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 1 es factorial(3)

Step 8: return 6

factorial(2)

Step 2: executes factorial(2)

St urn 2

return 2 * factorial(1)

Step 3: executes factorial(1)

Step 6: return 1

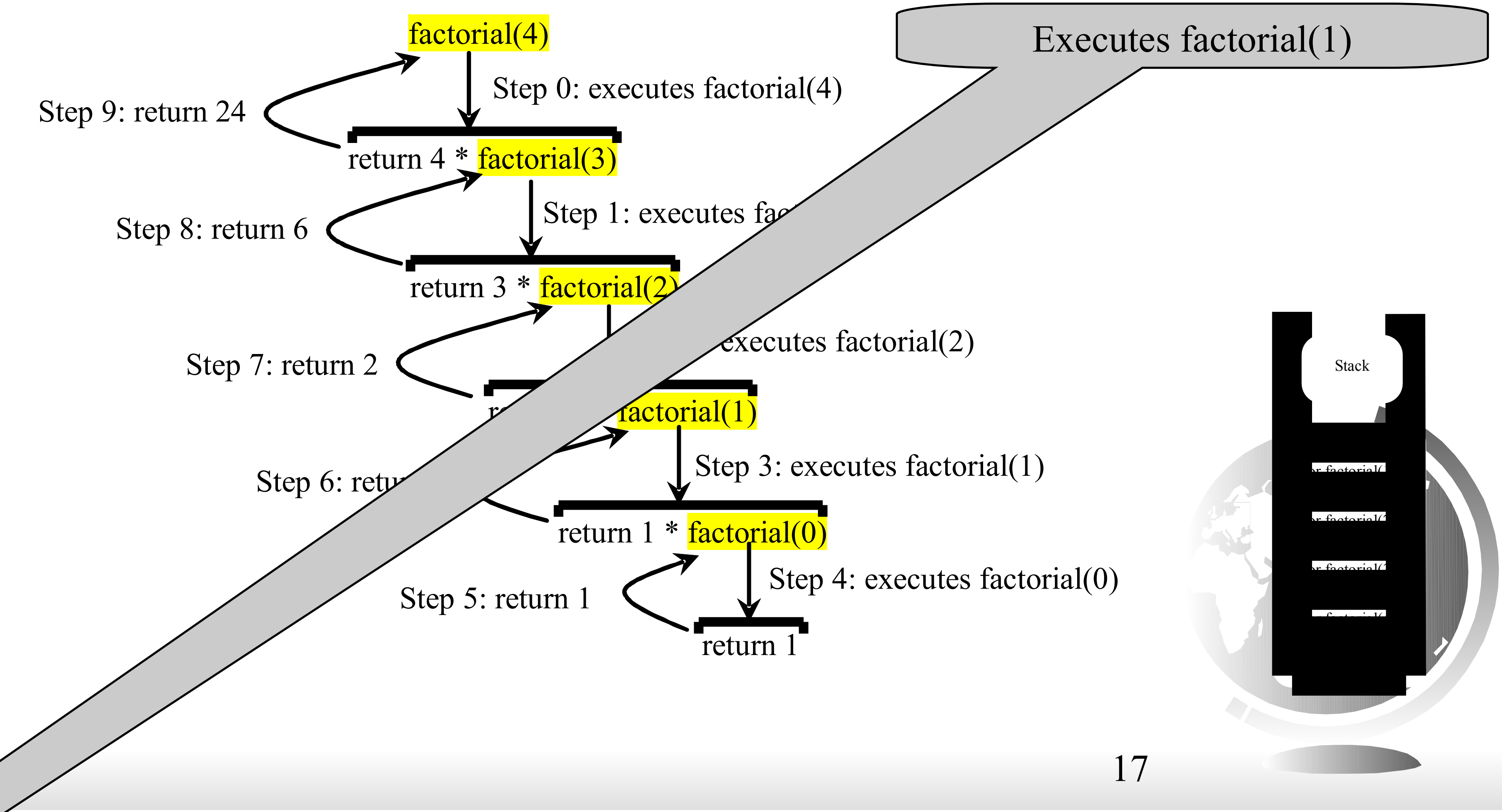
return 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

return 1

Trace Recursive factorial



Executes factorial(1)

factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 1: executes fa

Step 8: return 6

return 3 * factorial(2)

executes factorial(2)

Step 7: return 2

r factorial(1)

Step 3: executes factorial(1)

Step 6: retu

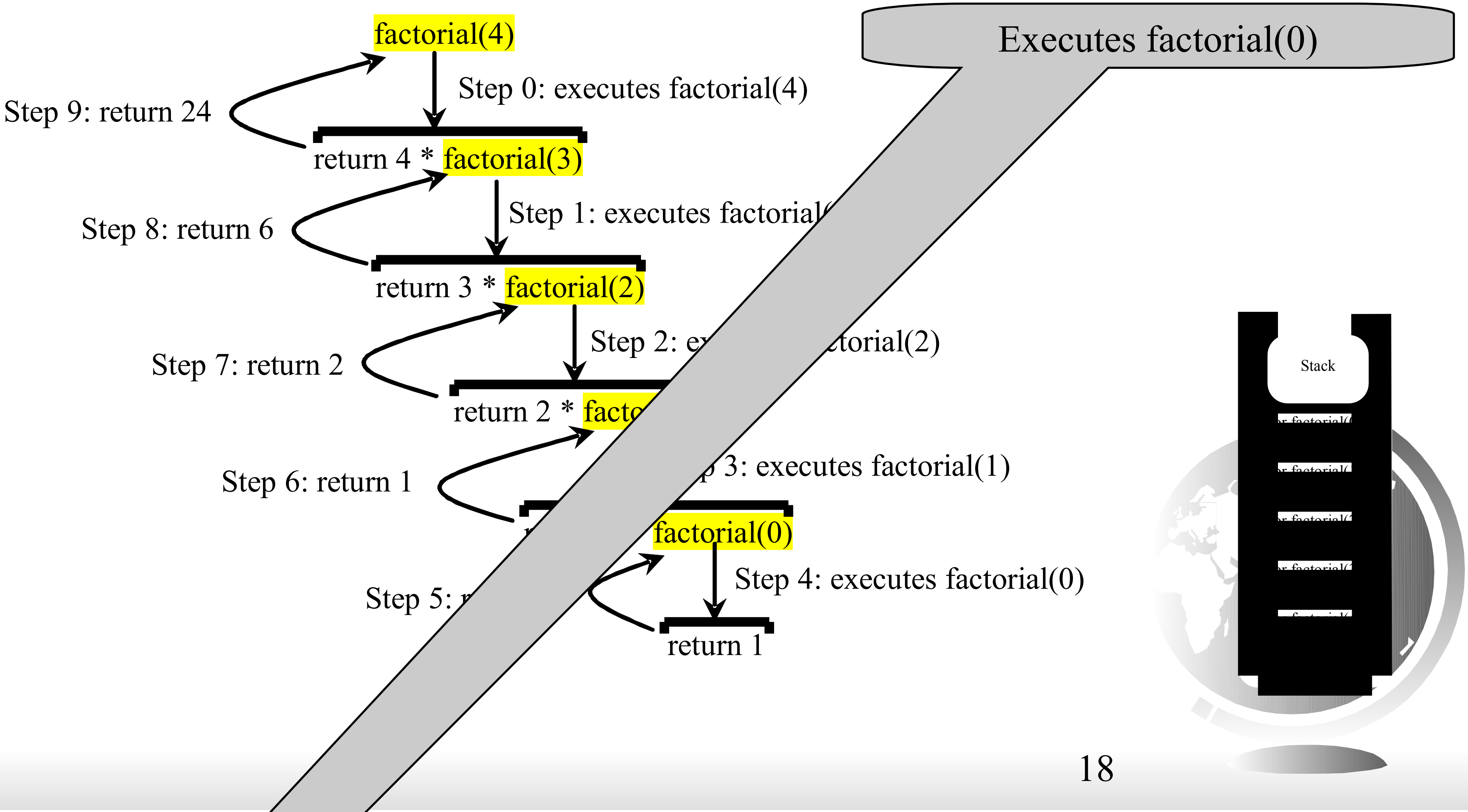
return 1 * factorial(0)

Step 4: executes factorial(0)

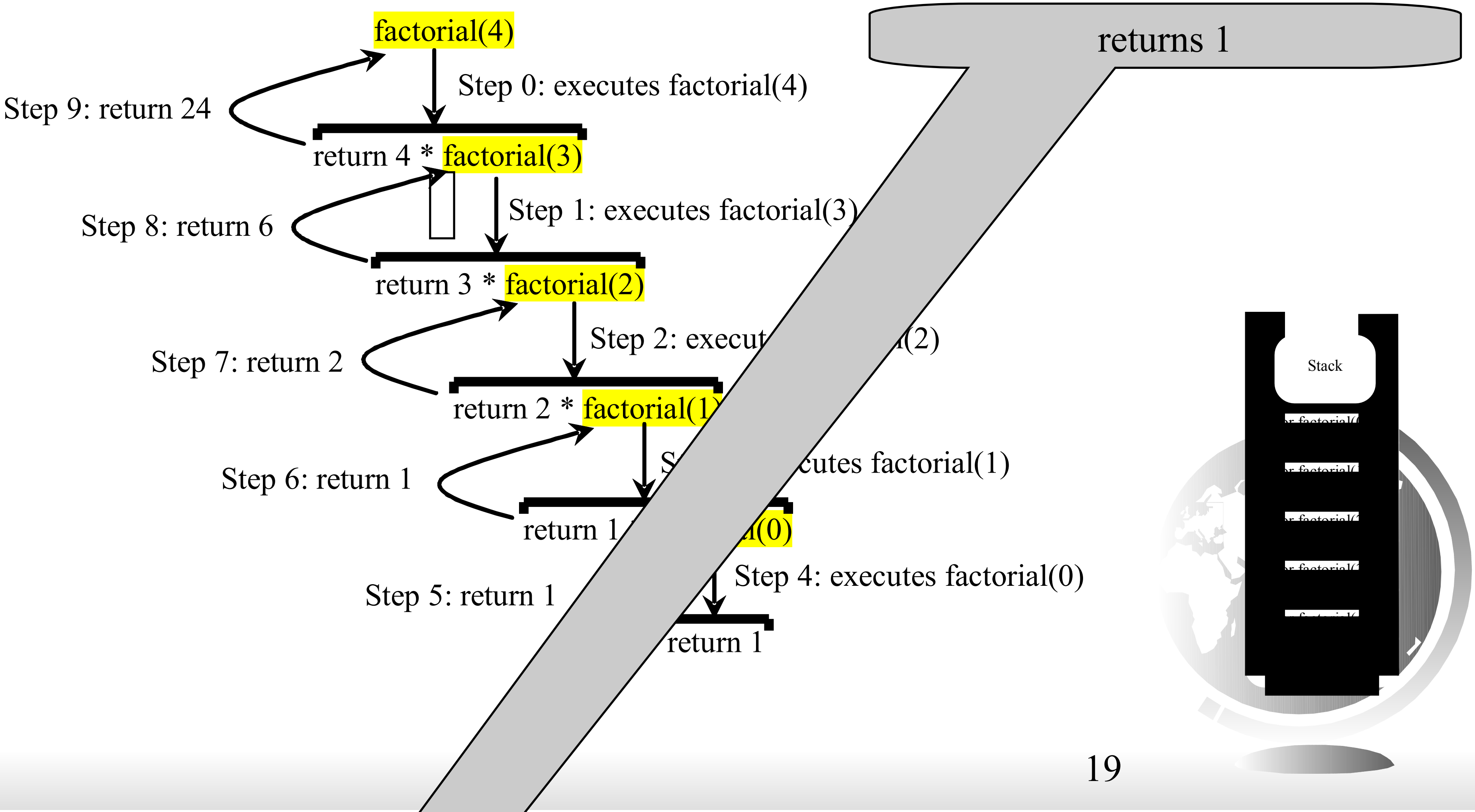
Step 5: return 1

return 1

Trace Recursive factorial



Trace Recursive factorial



factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 1: executes factorial(3)

Step 8: return 6

return 3 * factorial(2)

Step 2: execut (2)

Step 7: return 2

return 2 * factorial(1

Step 6: return 1

S cutes factorial(1)

return 1 (0)

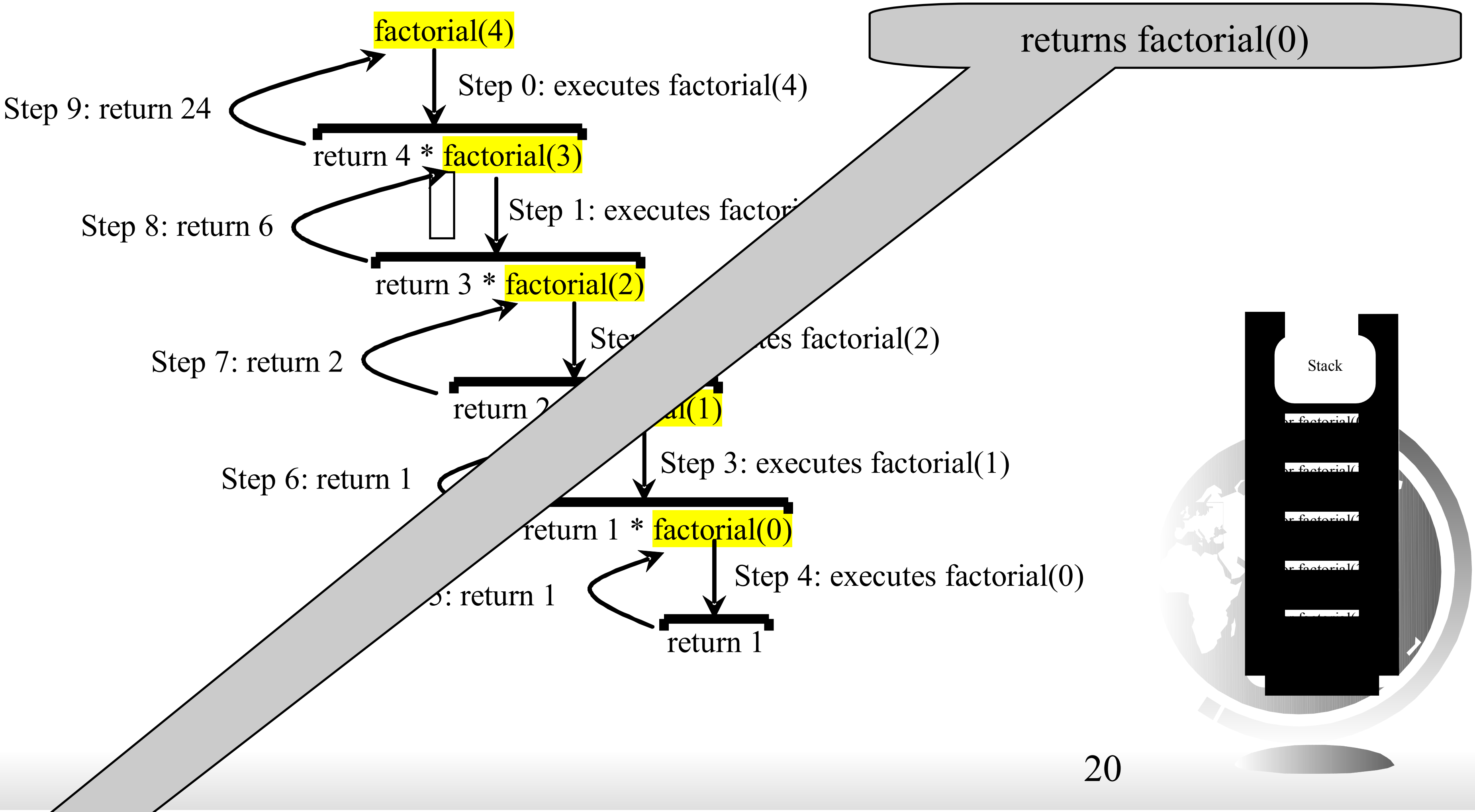
Step 4: executes factorial(0)

Step 5: return 1

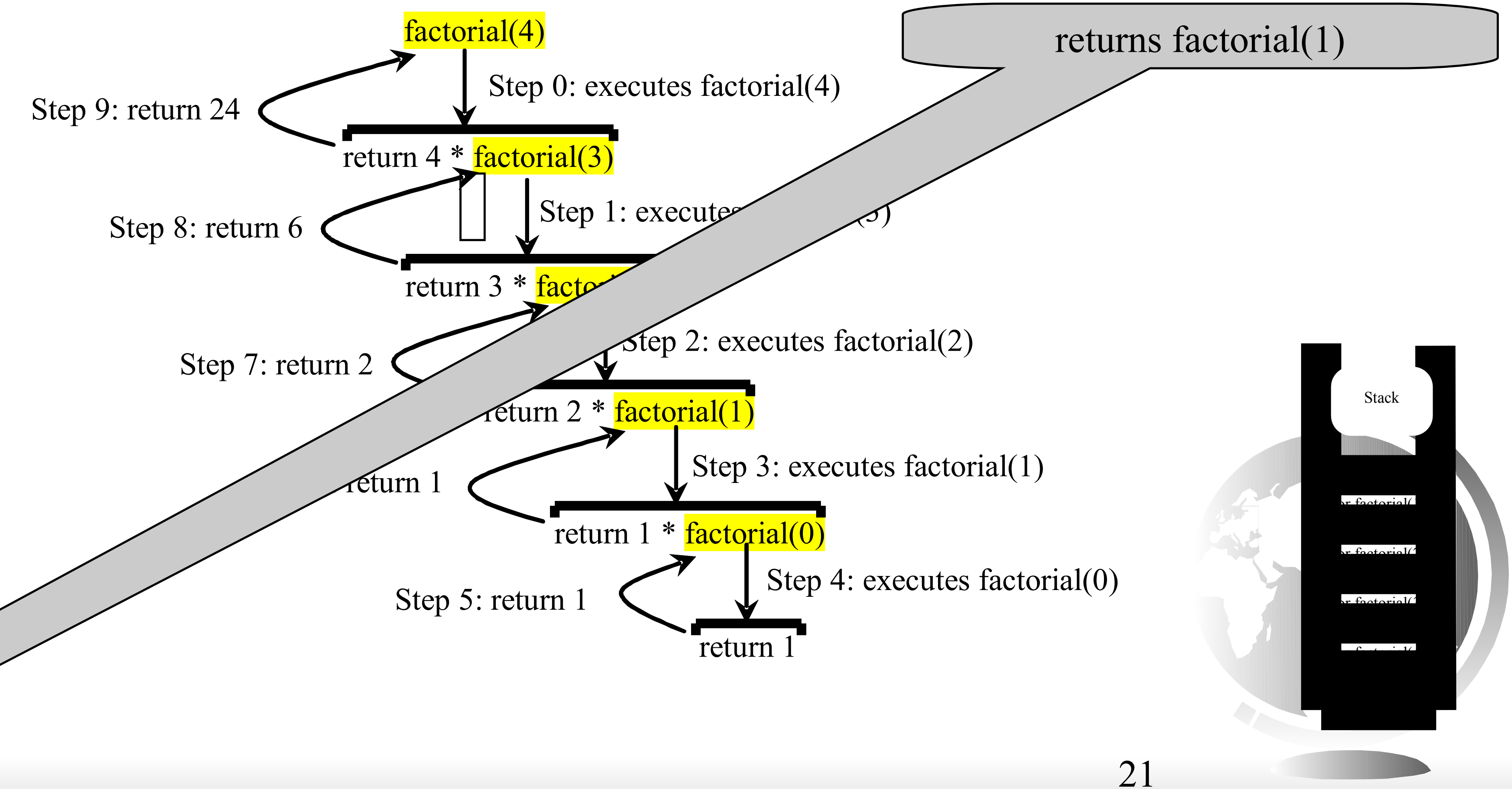
return 1

returns 1

Trace Recursive factorial



Trace Recursive factorial



returns factorial(1)

factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 1: executes (3)

Step 8: return 6

return 3 * facto

Step 2: executes factorial(2)

Step 7: return 2

return 2 * factorial(1)

return 1

Step 3: executes factorial(1)

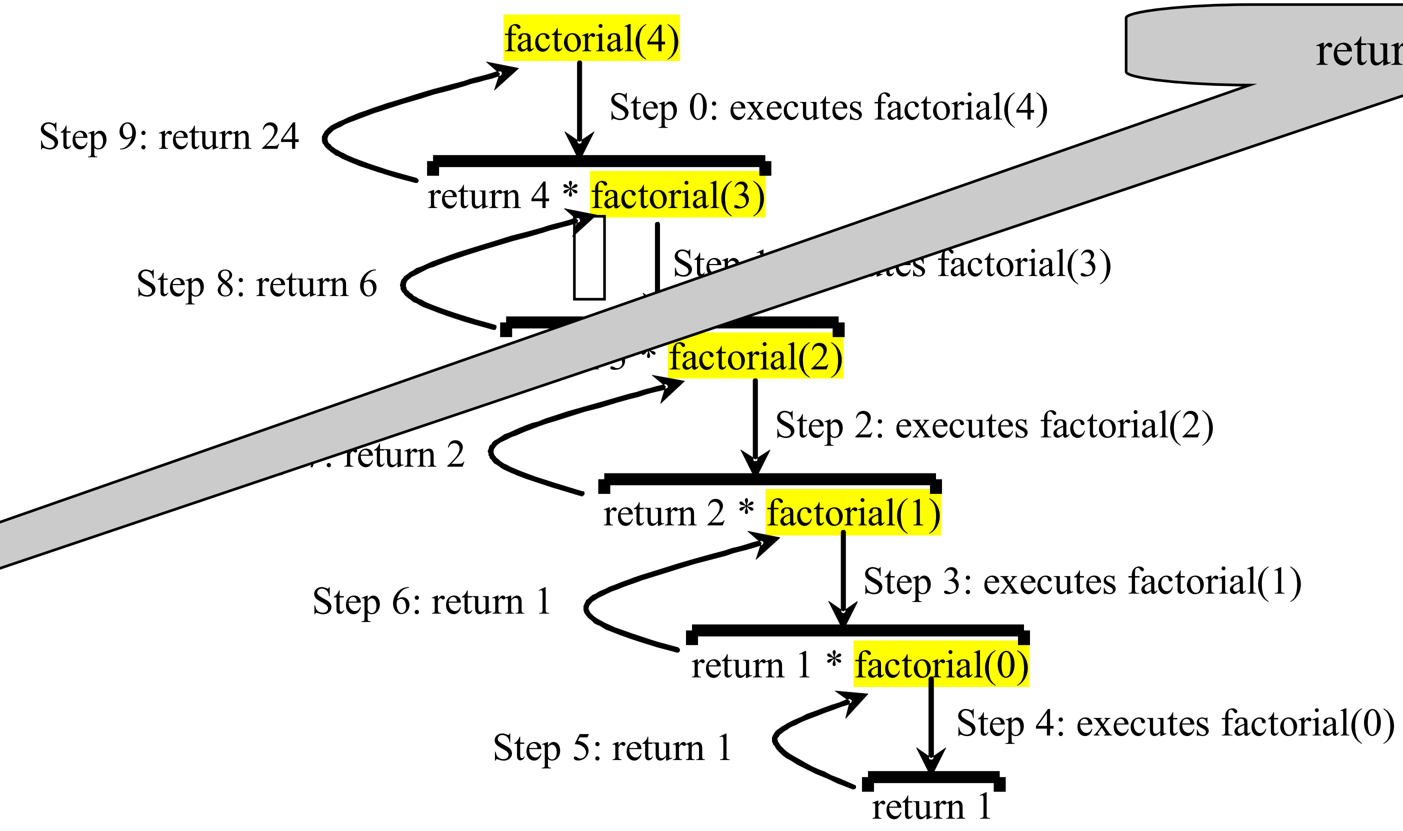
return 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

return 1

Trace Recursive factorial



factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 8: return 6

Step es factorial(3)

3 * factorial(2)

: return 2

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 6: return 1

Step 3: executes factorial(1)

return 1 * factorial(0)

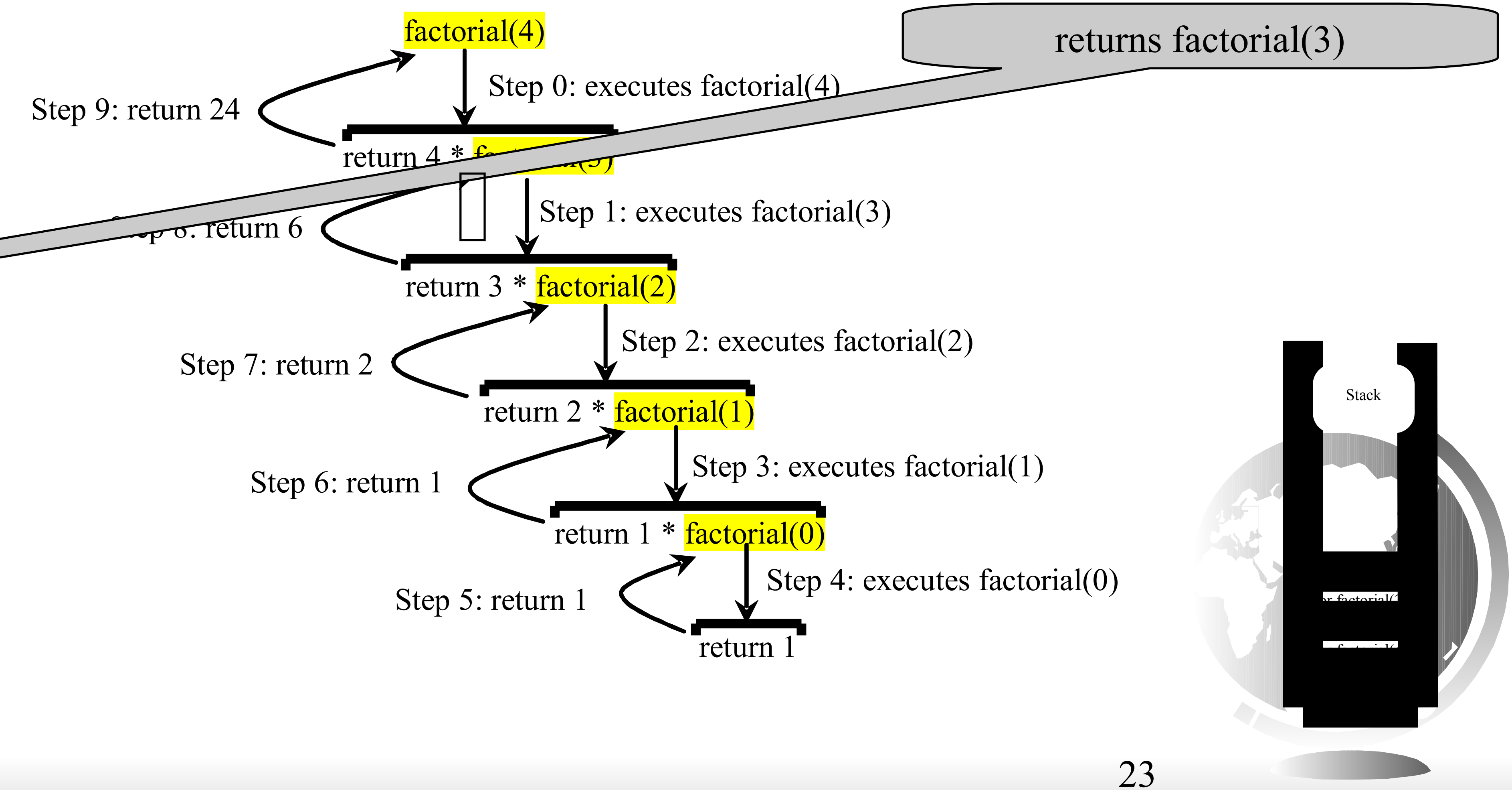
Step 5: return 1

Step 4: executes factorial(0)

return 1

returns factorial(2)

Trace Recursive factorial



factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 1: executes factorial(3)

Step 8: return 6

return 3 * factorial(2)

Step 2: executes factorial(2)

Step 7: return 2

return 2 * factorial(1)

Step 3: executes factorial(1)

Step 6: return 1

return 1 * factorial(0)

Step 4: executes factorial(0)

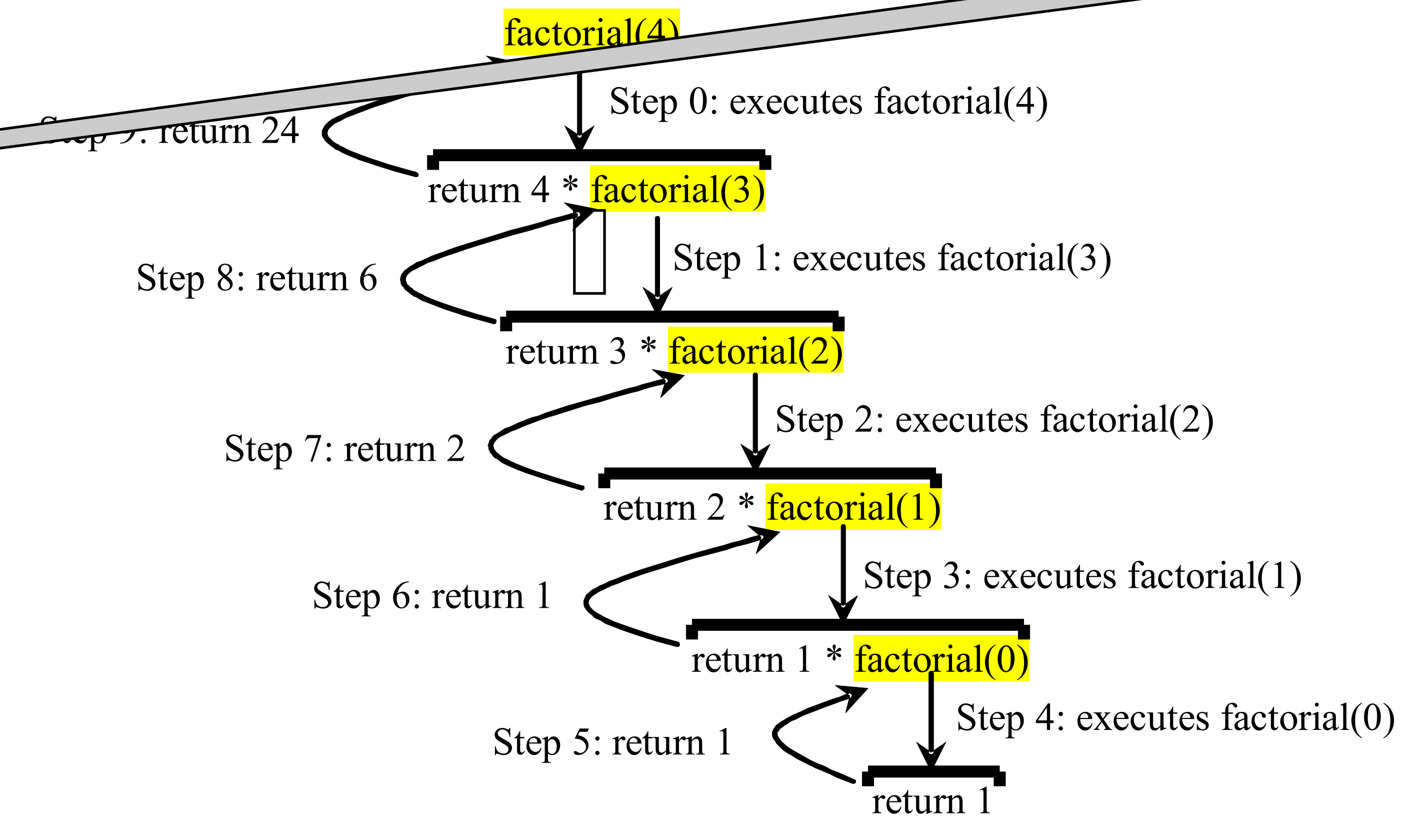
Step 5: return 1

return 1

returns factorial(3)

Trace Recursive factorial

returns factorial(4)



factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 1: executes factorial(3)

Step 8: return 6

return 3 * factorial(2)

Step 2: executes factorial(2)

Step 7: return 2

return 2 * factorial(1)

Step 3: executes factorial(1)

Step 6: return 1

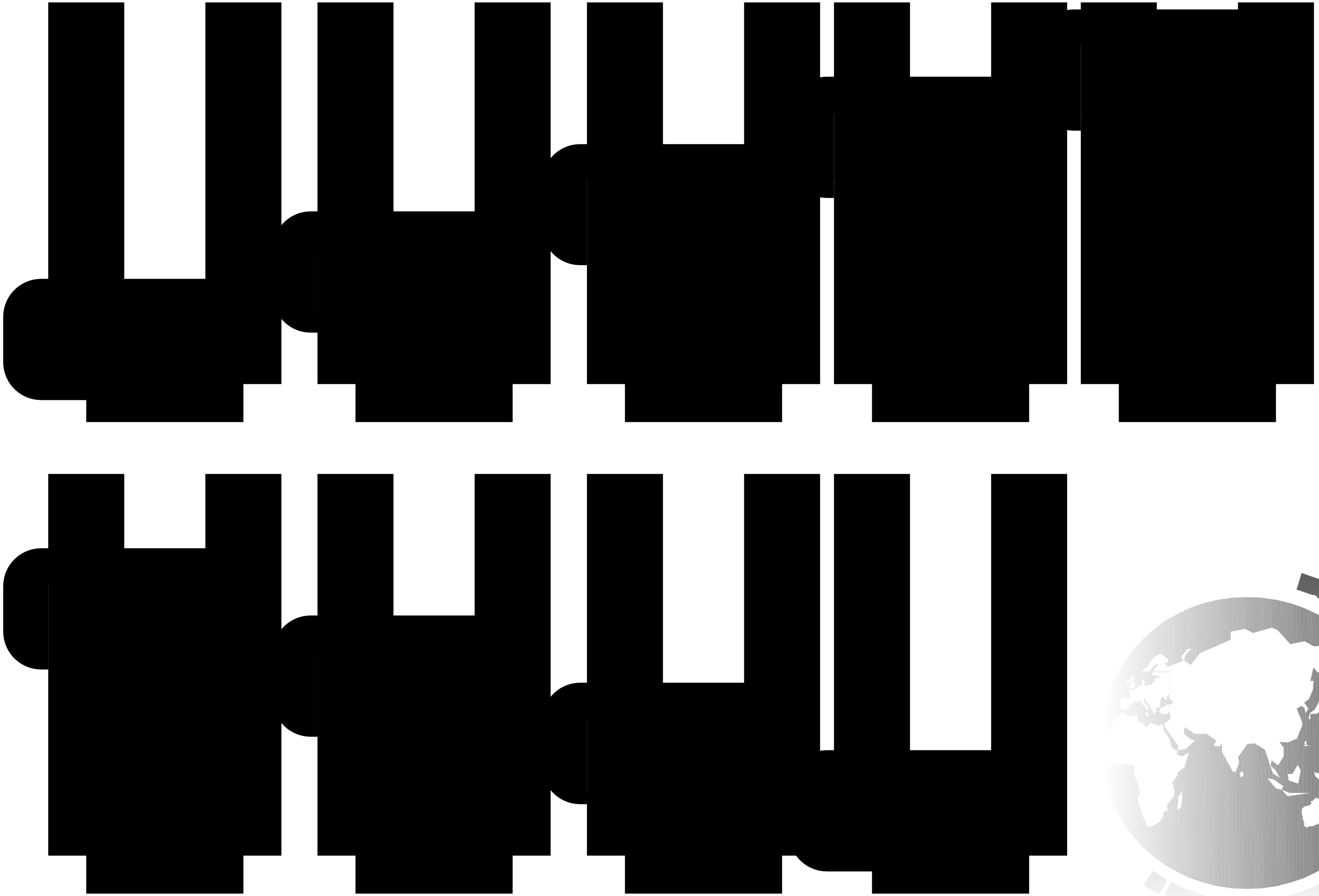
return 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

return 1

factorial(4) Stack Trace



Other Examples

$f(0) = 0;$

$f(n) = n + f(n-1);$

Fibonacci Numbers

```
Fibonacci series: 0 1 1 2 3 5 8 13 21 34 55 89…
         indices: 0 1 2 3 4 5 6 7  8  9  10 11
```

fib(0) = 0;

fib(1) = 1;

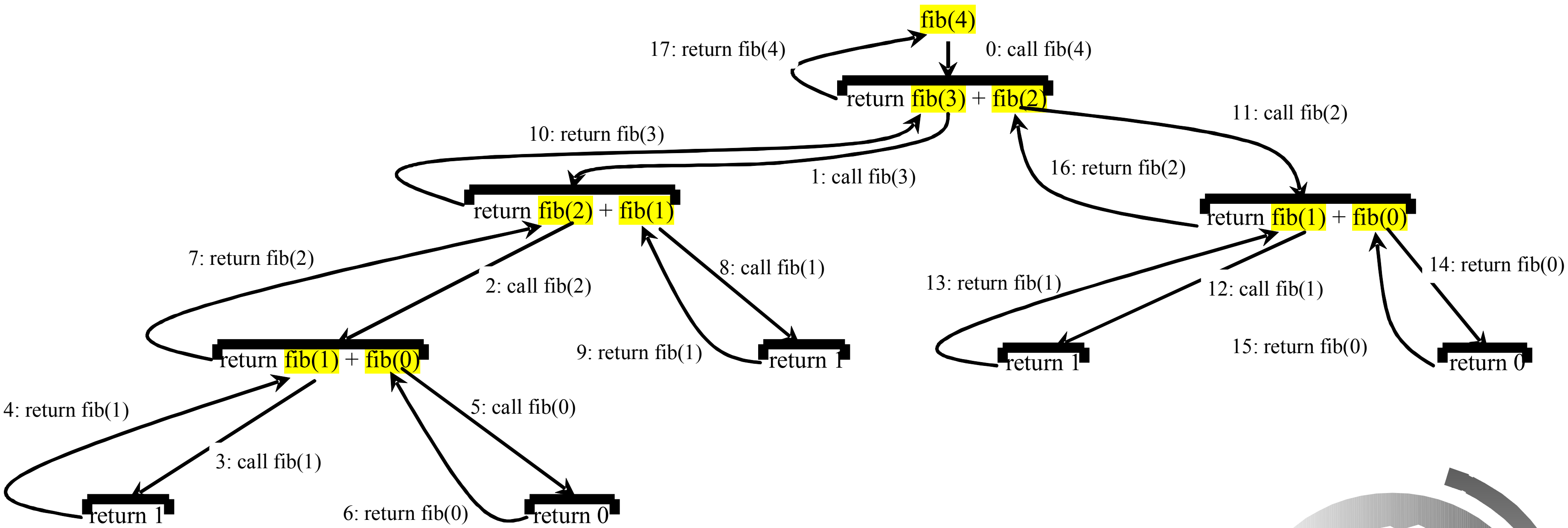
fib(index) = fib(index -1) + fib(index -2); index >=2

fib(3) = fib(2) + fib(1) = (fib(1) + fib(0)) + fib(1) = (1 + 0) +fib(1) = 1 + fib(1) = 1 + 1 = 2

ComputeFibonacci

Run

Fibonnaci Numbers, cont.



fib(4)

17: return fib(4)

0: call fib(4)

return fib(3) + fib(2)

11: call fib(2)

10: return fib(3)

1: call fib(3)

16: return fib(2)

return fib(2) + fib(1)

return fib(1) + fib(0)

7: return fib(2)

2: call fib(2)

8: call fib(1)

13: return fib(1)

12: call fib(1)

14: return fib(0)

return fib(1) + fib(0)

9: return fib(1)

return 1

return 1

15: return fib(0)

return 0

4: return fib(1)

5: call fib(0)

3: call fib(1)

return 1

6: return fib(0)

return 0

Characteristics of Recursion

All recursive methods have the following characteristics:

- One or more base cases (the simplest case) are used to stop recursion.
- Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

In general, to solve a problem using recursion, you break it into subproblems. If a subproblem resembles the original problem, you can apply the same approach to solve the subproblem recursively. This subproblem is almost the same as the original problem in nature with a smaller size.

Problem Solving Using Recursion

Let us consider a simple problem of printing a message for n times. You can break the problem into two subproblems: one is to print the message one time and the other is to print the message for n-1 times. The second problem is the same as the original problem with a smaller size. The base case for the problem is n==0. You can solve this problem using recursion as follows:

nPrintln("Welcome", 5);

```
public static void nPrintln(String message, int times) {
  if (times >= 1) {
    System.out.println(message);
    nPrintln(message, times - 1);
  } // The base case is times == 0
}
```

Think Recursively

Many of the problems presented in the early chapters can be solved using recursion if you *think recursively*. For example, the palindrome problem can be solved recursively as follows:

```
public static boolean isPalindrome(String s) {
  if (s.length() <= 1) // Base case
    return true;
  else if (s.charAt(0) != s.charAt(s.length() - 1)) // Base case
    return false;
  else
    return isPalindrome(s.substring(1, s.length() - 1));
}
```

Recursive Helper Methods

The preceding recursive isPalindrome method is not efficient, because it creates a new string for every recursive call. To avoid creating new strings, use a helper method:

```
public static boolean isPalindrome(String s) {
  return isPalindrome(s, 0, s.length() - 1);
}
public static boolean isPalindrome(String s, int low, int high) {
  if (high <= low) // Base case
    return true;
  else if (s.charAt(low) != s.charAt(high)) // Base case
    return false;
  else
    return isPalindrome(s, low + 1, high - 1);
}
```

Recursive Selection Sort

1. Find the smallest number in the list and swaps it with the first number.
2. Ignore the first number and sort the remaining smaller list recursively.

RecursiveSelectionSort

Recursive Binary Search

1. Case 1: If the key is less than the middle element, recursively search the key in the first half of the array.
2. Case 2: If the key is equal to the middle element, the search ends with a match.
3. Case 3: If the key is greater than the middle element, recursively search the key in the second half of the array.

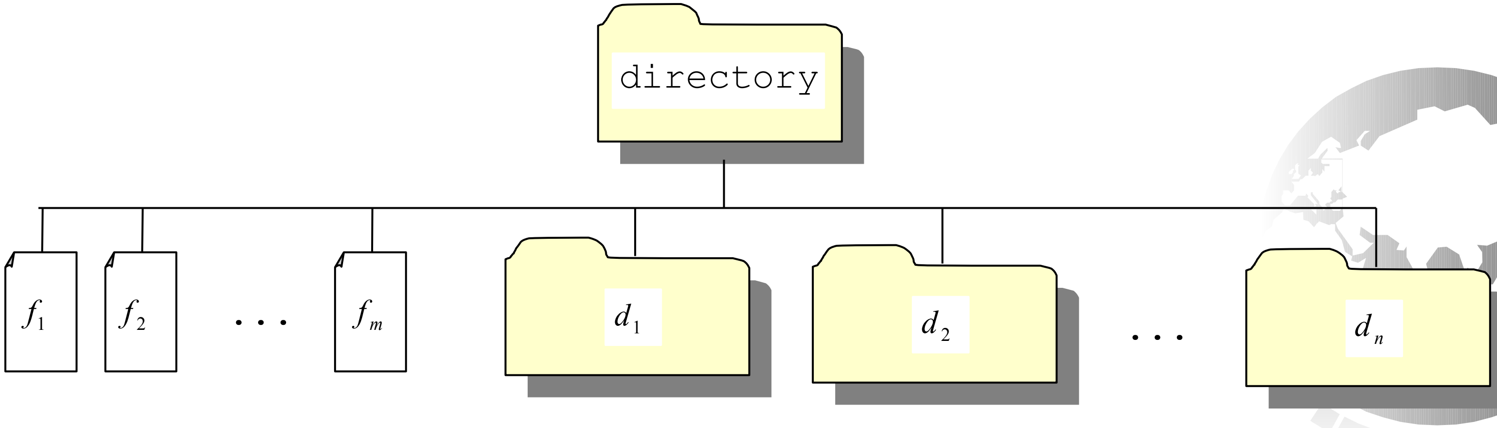
RecursiveBinarySearch

Recursive Implementation

```
/** Use binary search to find the key in the list */
public static int recursiveBinarySearch(int[] list, int key) {
  int low = 0;
  int high = list.length - 1;
  return recursiveBinarySearch(list, key, low, high);
}

/** Use binary search to find the key in the list between
    list[low] list[high] */
public static int recursiveBinarySearch(int[] list, int key,
  int low, int high) {
  if (low > high)  // The list has been exhausted without a match
    return -low - 1;

  int mid = (low + high) / 2;
  if (key < list[mid])
    return recursiveBinarySearch(list, key, low, mid - 1);
  else if (key == list[mid])
    return mid;
  else
    return recursiveBinarySearch(list, key, mid + 1, high);
```

Directory Size

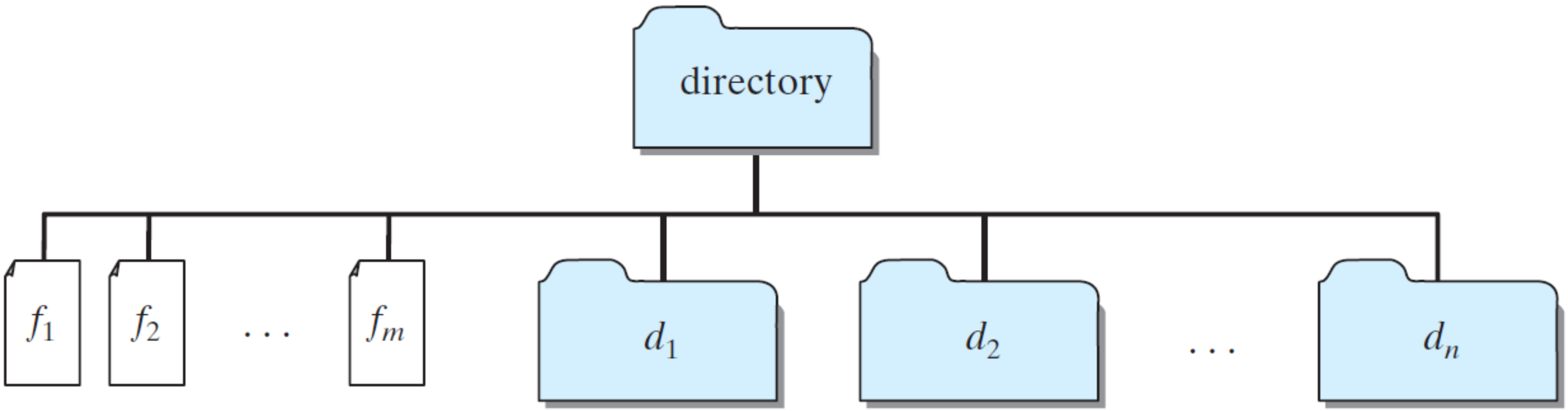
The preceding examples can easily be solved without using recursion. This section presents a problem that is difficult to solve without using recursion. The problem is to find the size of a directory. The size of a directory is the sum of the sizes of all files in the directory. A directory may contain subdirectories. Suppose a directory contains files , , ..., , and subdirectories , , ..., , as shown below.



Directory Size

The size of the directory can be defined recursively as follows:

$$size(d) \quad size(f_1) \quad size(f_2) \quad \ldots \quad size(f_m) \quad size(d_1) \quad size(d_2) \quad \ldots \quad size(d_n)$$



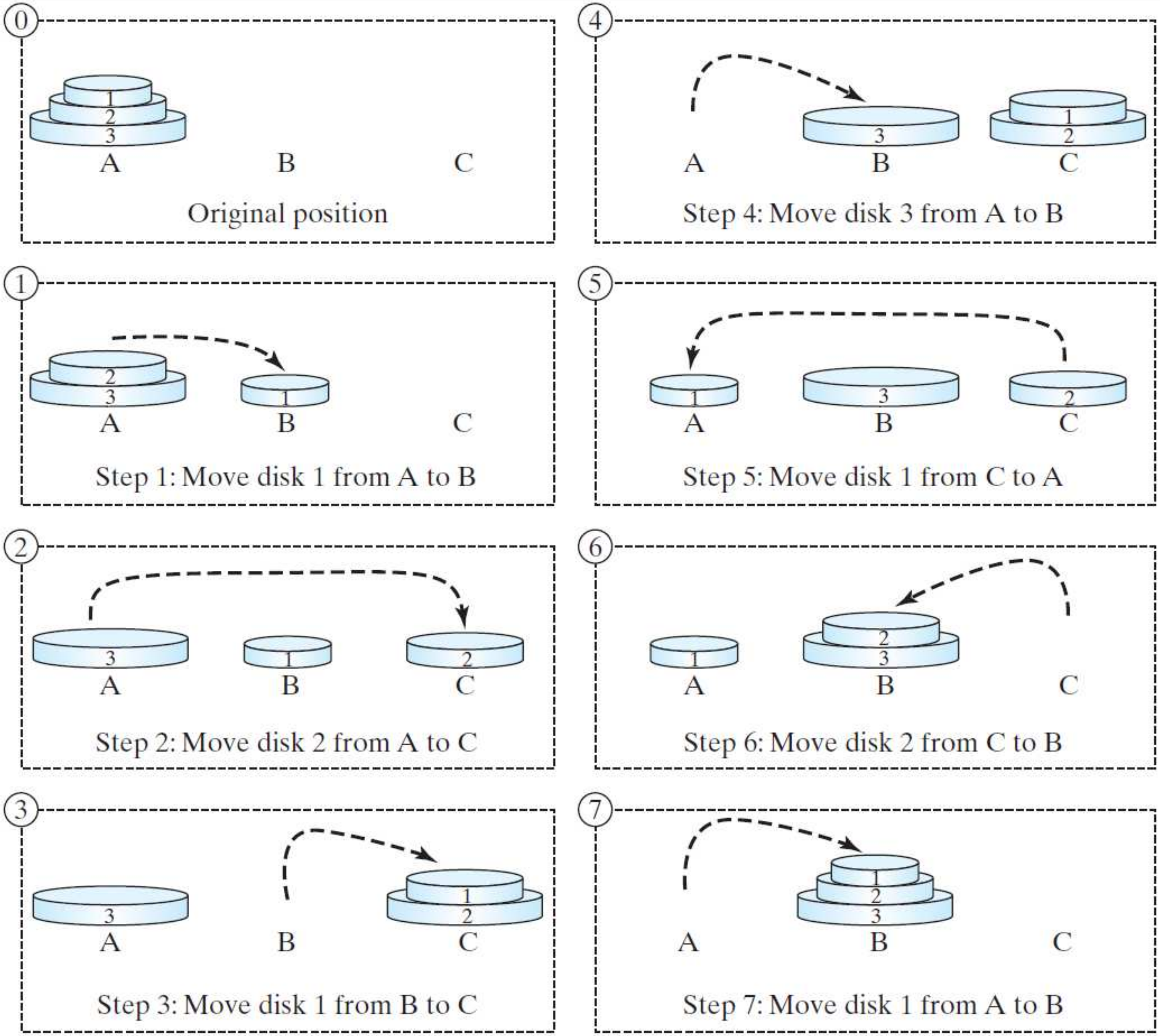
DirectorySize

Run

Tower of Hanoi

- There are *n* disks labeled 1, 2, 3, . . ., *n*, and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time, and it must be the top disk on the tower.

Tower of Hanoi cont



Original position

Step 1: Move disk 1 from A to B

Step 2: Move disk 2 from A to C

Step 3: Move disk 1 from B to C

Step 4: Move disk 3 from A to B

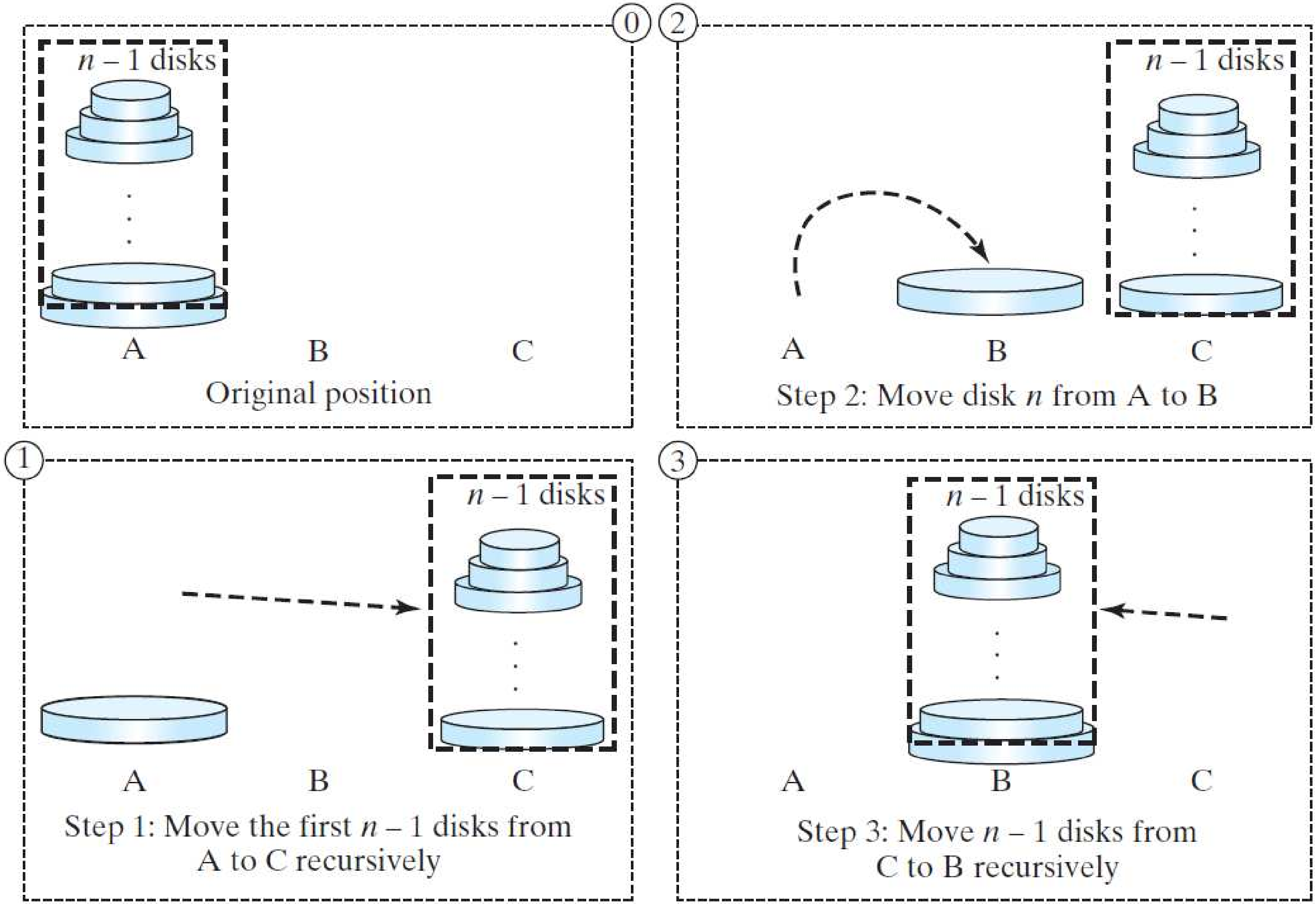
Step 5: Move disk 1 from C to A

Step 6: Move disk 2 from C to B

Step 7: Move disk 1 from A to B

Solution to Tower of Hanoi

The Tower of Hanoi problem can be decomposed into three subproblems



Original position

Step 2: Move disk n from A to B

Step 1: Move the first $n-1$ disks from A to C recursively

Step 3: Move $n-1$ disks from C to B recursively

Solution to Tower of Hanoi

- Move the first n - 1 disks from A to C with the assistance of tower B.
- Move disk n from A to B.
- Move n - 1 disks from C to B with the assistance of tower A.

TowerOfHanoi

Run

Exercise 18.3 GCD

gcd(2, 3) = 1

gcd(2, 10) = 2

gcd(25, 35) = 5

gcd(205, 301) = 5

gcd(m, n)

Approach 1: Brute-force, start from min(n, m) down to 1, to check if a number is common divisor for both m and n, if so, it is the greatest common divisor.

Approach 2: Euclid’s algorithm

Approach 3: Recursive method

Approach 2: Euclid's algorithm

```
// Get absolute value of m and n;
t1 = Math.abs(m); t2 = Math.abs(n);
// r is the remainder of t1 divided by t2;
r = t1 % t2;
while (r != 0) {
  t1 = t2;
  t2 = r;
  r = t1 % t2;
}

// When r is 0, t2 is the greatest common
// divisor between t1 and t2
return t2;
```

Approach 3: `Recursive Method`

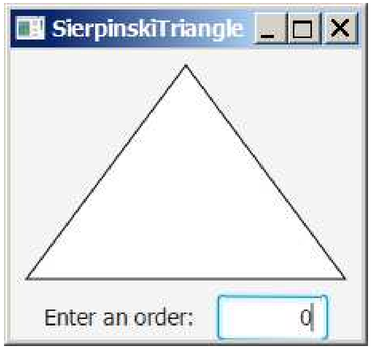
gcd(m, n) = n if m % n = 0;
gcd(m, n) = gcd(n, m % n); otherwise;

Fractals?

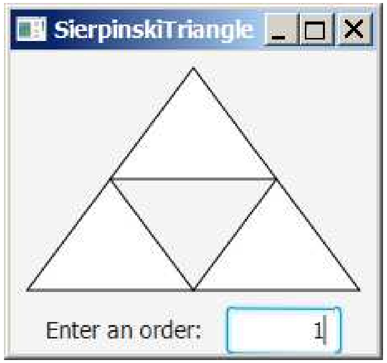
A fractal is a geometrical figure just like triangles, circles, and rectangles, but fractals can be divided into parts, each of which is a reduced-size copy of the whole. There are many interesting examples of fractals. This section introduces a simple fractal, called *Sierpinski triangle*, named after a famous Polish mathematician.

Sierpinski Triangle

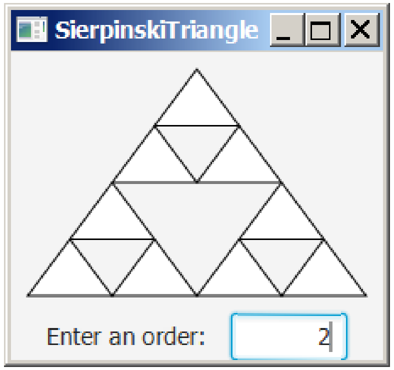
1. It begins with an equilateral triangle, which is considered to be the Sierpinski fractal of order (or level) 0, as shown in Figure (a).
2. Connect the midpoints of the sides of the triangle of order 0 to create a Sierpinski triangle of order 1, as shown in Figure (b).
3. Leave the center triangle intact. Connect the midpoints of the sides of the three other triangles to create a Sierpinski of order 2, as shown in Figure (c).
4. You can repeat the same process recursively to create a Sierpinski triangle of order 3, 4, ..., and so on, as shown in Figure (d).



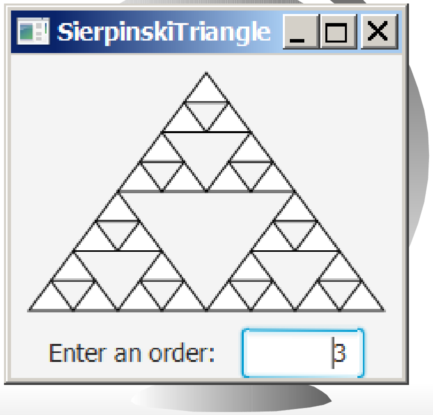
Enter an order: 0



Enter an order: 1

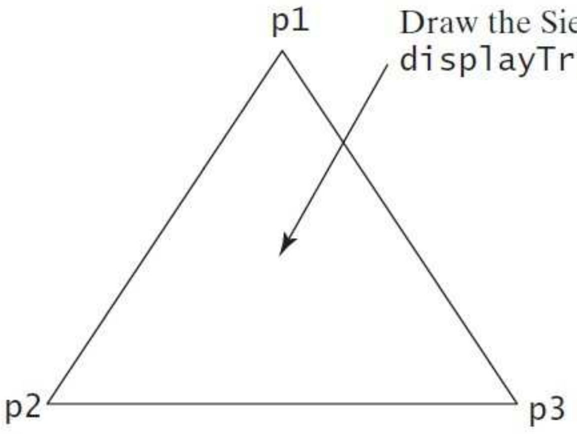


Enter an order: 2



Enter an order: 3

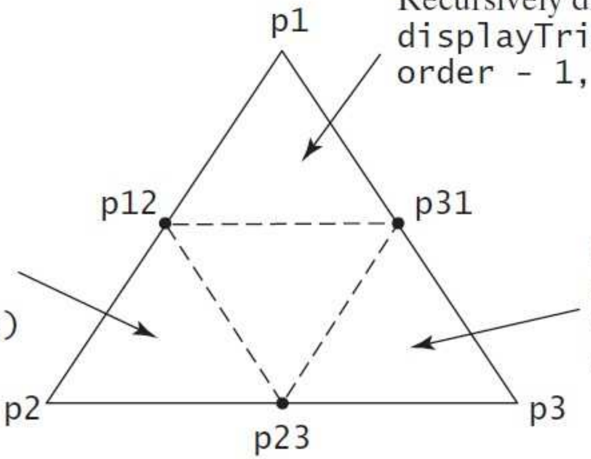
Sierpinski Triangle Solution



Draw the Sierpinski triangle

```
displayTriangles(order, p1, p2, p3)
```

(a)



Recursively draw the small Sierpinski triangle

```
displayTriangles(
order - 1, p1, p12, p31)
```

Recursively draw the small Sierpinski triangle

```
displayTriangles(
order - 1, p12, p2, p23)
```

Recursively draw the small Sierpinski triangle

```
displayTriangles(
order - 1, p31, p23, p3)
```

(b)

SierpinskiTriangle

Run

Recursion vs. Iteration

Recursion is an alternative form of program control. It is essentially repetition without a loop.

Recursion bears substantial overhead. Each time the program calls a method, the system must assign space for all of the method’s local variables and parameters. This can consume considerable memory and requires extra time to manage the additional space.

Advantages of Using Recursion

Recursion is good for solving the problems that are inherently recursive.

Tail Recursion

A recursive method is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call.

Non-tail recursive

ComputeFactorial

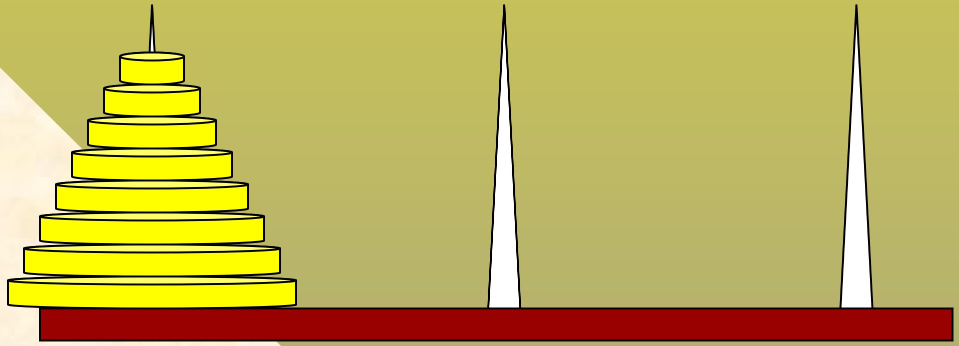
Tail recursive

ComputeFactorialTailRecursion

The Towers of Hanoi

A Legend

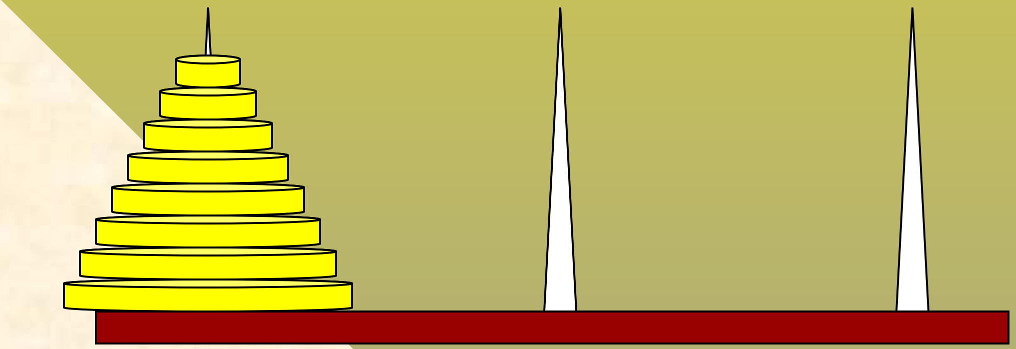
Legend has it that there were three diamond needles set into the floor of the temple of Brahma in Hanoi.



Stacked upon the leftmost needle were 64 golden disks, each a different size, stacked in concentric order:

A Legend (*Ct'd*)

The priests were to transfer the disks from the first needle to the second needle, using the third as necessary.

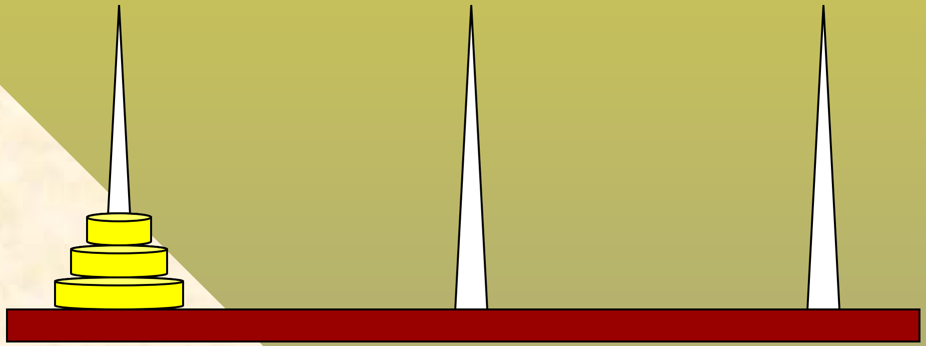


But they could *only move one disk at a time*, and could *never put a larger disk on top of a smaller one*.

When they completed this task, <u>the world would end</u>!

To Illustrate

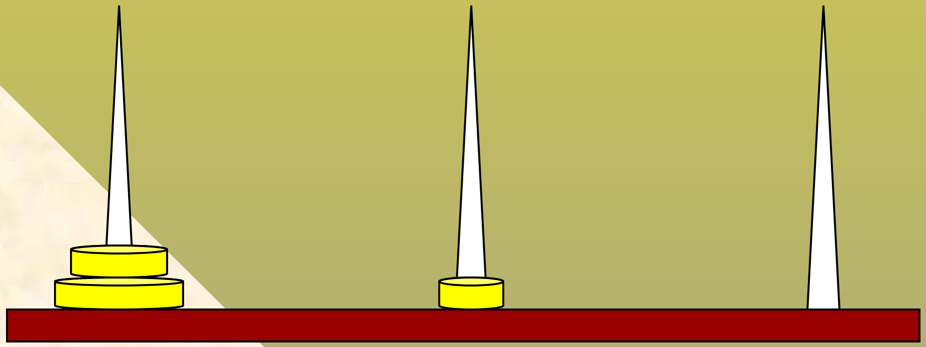
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as A, B, and C...



Since we can only move one disk at a time, we move the top disk from A to B.

Example

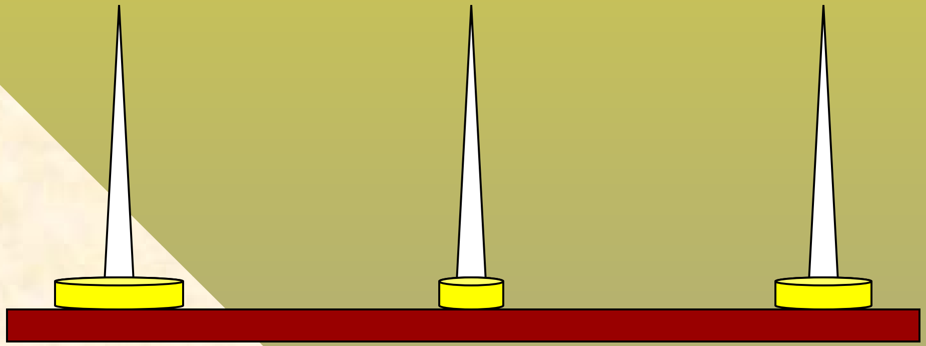
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as A, B, and C...



We then move the top disk from A to C.

Example (*Ct'd*)

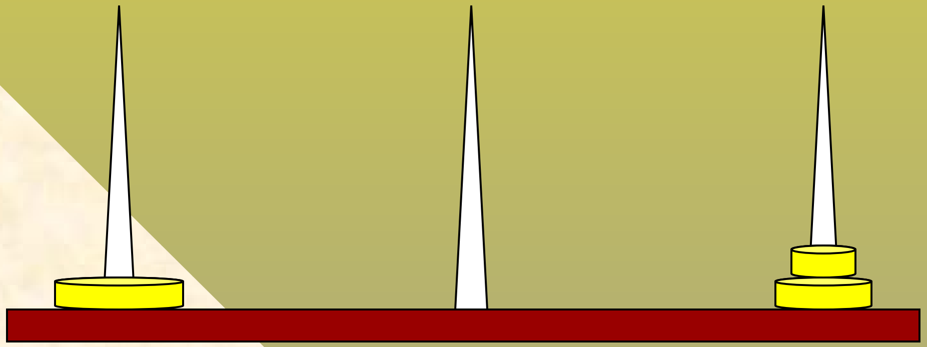
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as A, B, and C...



We then move the top disk from B to C.

Example (*Ct'd*)

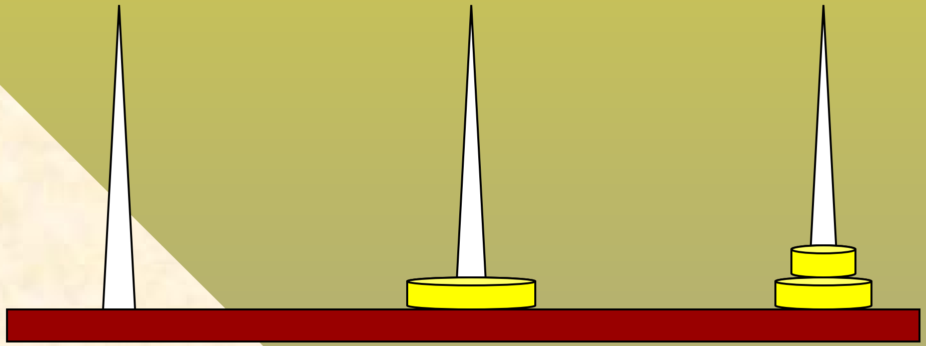
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as A, B, and C...



We then move the top disk from A to B.

Example (*Ct'd*)

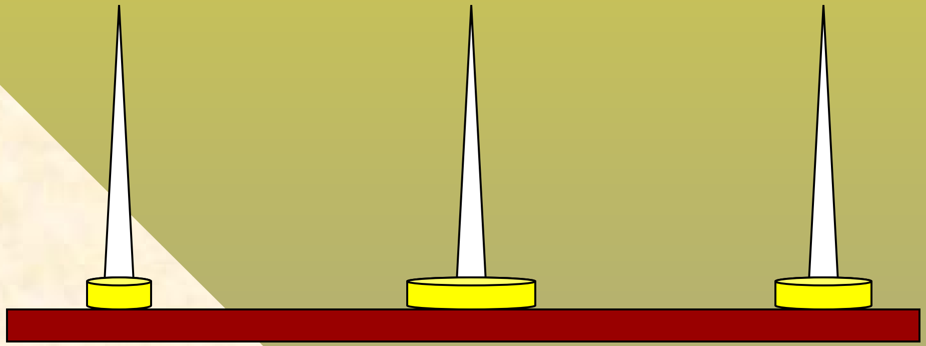
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as A, B, and C...



We then move the top disk from C to A.

Example (*Ct’d*)

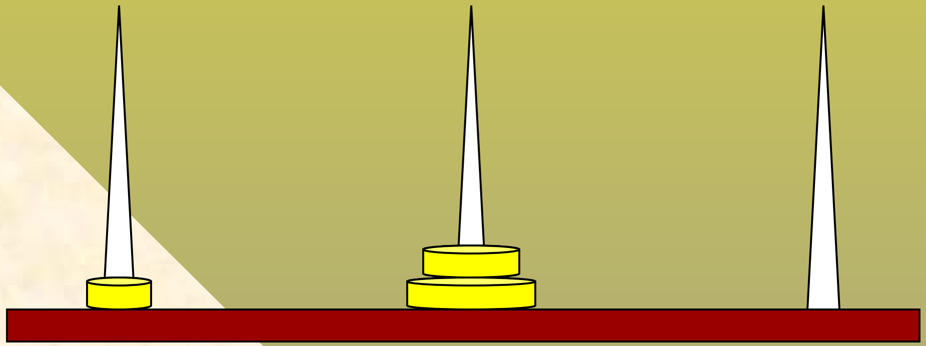
For simplicity, suppose there were just 3 disks, and we’ll refer to the three needles as A, B, and C...



We then move the top disk from C to B.

Example (*Ct'd*)

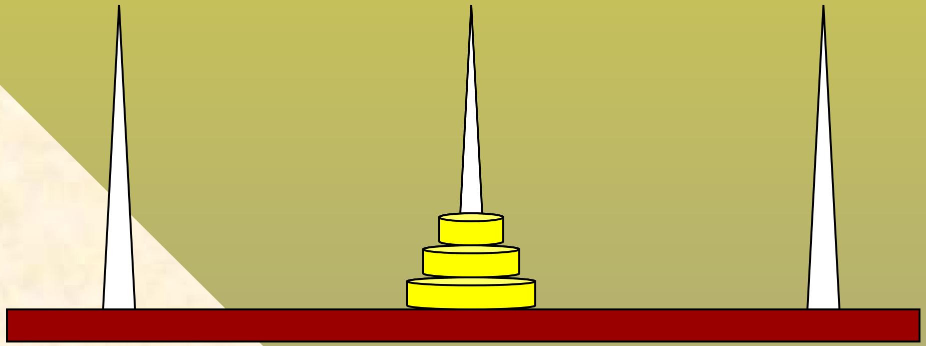
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as A, B, and C...



We then move the top disk from A to B.

Example (*Ct'd*)

For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as A, B, and C...

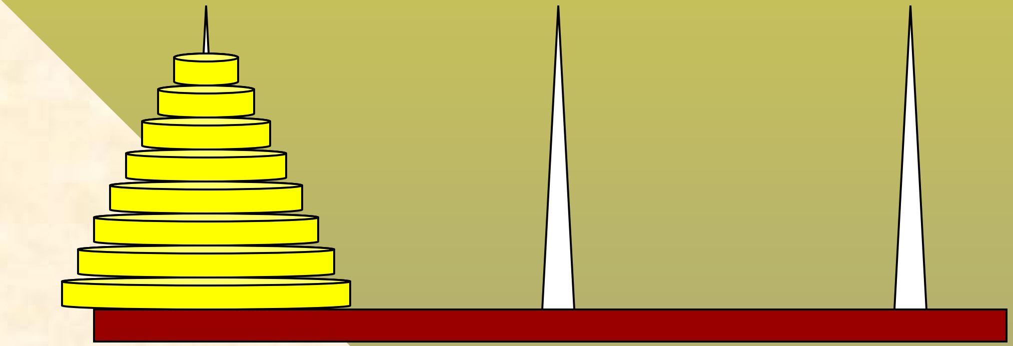


and we're done!

The problem gets more difficult as the number of disks increases...

Our Problem

Today’s problem is to write a program that generates the instructions for the priests to follow in moving the disks.



While quite difficult to solve iteratively, this problem has a simple and elegant *recursive* solution.

Analysis

For flexibility, let’s allow the user to enter the number of disks for which they wish a set of instructions:

```
/* hanoi.cpp
 * ...
 */

void Move(int n, char src, char dest, char aux);

int main()
{
   cout << “\n\nThe Hanoi Towers!\n\n”
        << “Enter how many disks: “;
   int numDisks;
   cin >> numDisks;

   Move(numDisks, ‘A’, ‘B’, ‘C’);
}
```

Analysis (Ct’d)

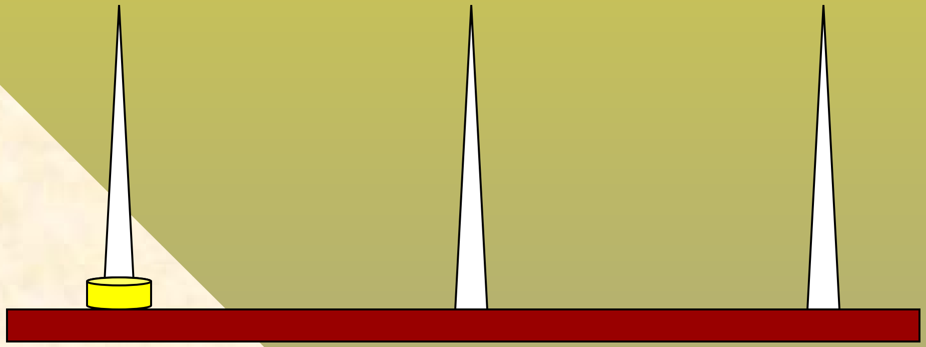
Our task, then is to write function Move() that does all the work:

```
/* hanoi.cpp
 * ...
 */

void Move(int n, char src, char dest, char aux);

int main()
{
  cout << “\n\nThe Hanoi Towers!\n\n”
       << “Enter how many disks: “;
  int numDisks;
  cin >> numDisks;

  Move(numDisks, ‘A’, ‘B’, ‘C’);
}
```

Design

Basis: What is an instance of the problem that is trivial?

→ n == 1

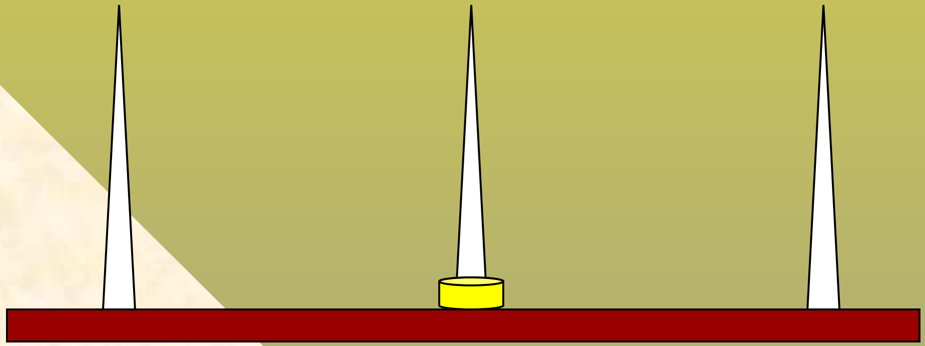


Since this base case could occur when the disk is on any needle, we simply output the instruction to move the top disk from *src* to *dest*.

Design

Basis: What is an instance of the problem that is trivial?

→ n == 1

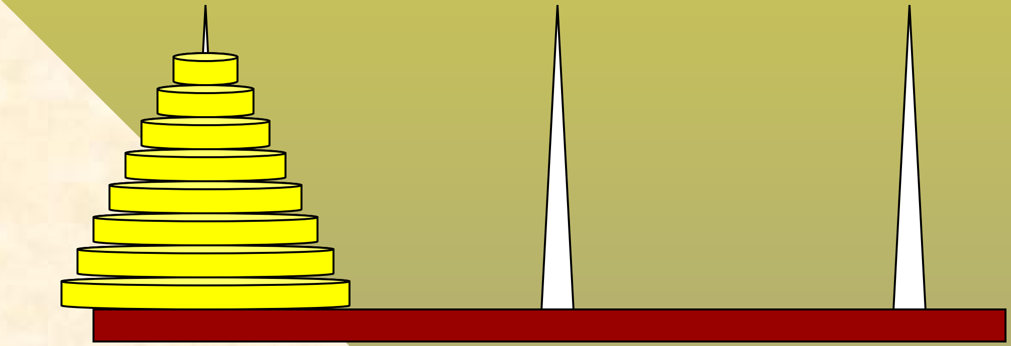


Since this base case could occur when the disk is on any needle, we simply output the instruction to move the top disk from *src* to *dest*.

Design (*Ct'd*)

Induction Step: n > 1

→ How can recursion help us out?

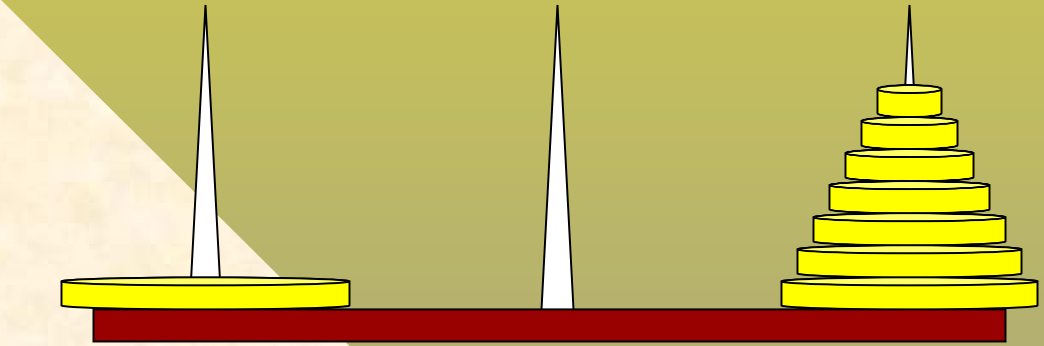


a. <u>Recursively</u> move n-1 disks from *src* to *aux*.

Design (*Ct'd*)

Induction Step: n > 1

→ How can recursion help us out?

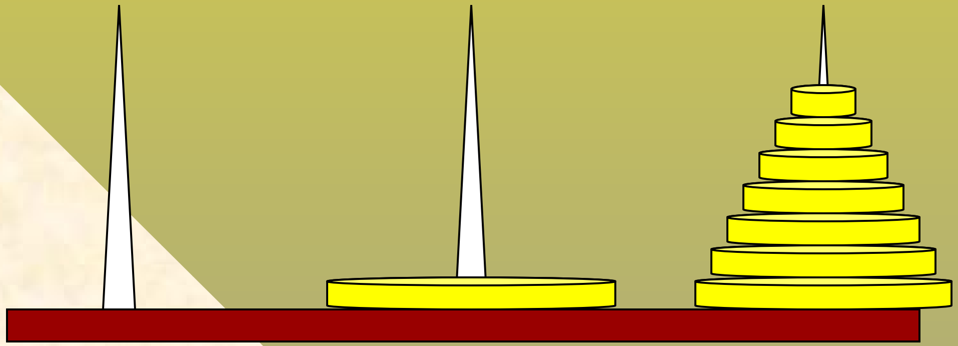


b. Move the one remaining disk from *src* to *dest*.

Design (*Ct'd*)

Induction Step: n > 1

→ How can recursion help us out?

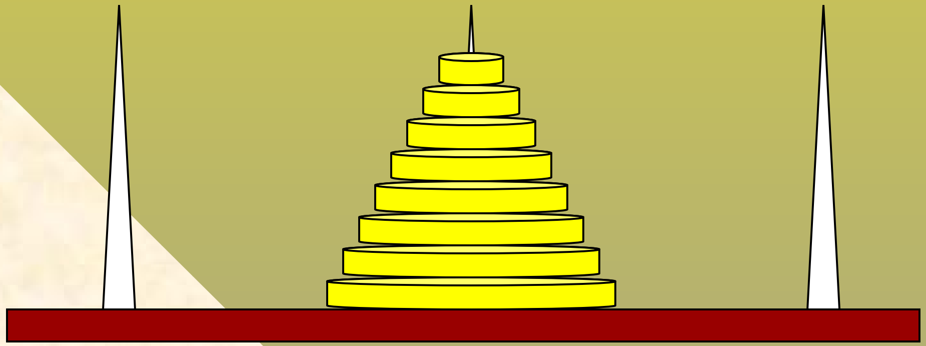


c. *Recursively* move n-1 disks from *aux* to *dest*...

Design (*Ct'd*)

Induction Step: n > 1

→ How can recursion help us out?



d. We're done!

Algorithm

We can combine these steps into the following algorithm:

```
0.  Receive n, src, dest, aux.
1.  If $n > 1$:
        a. Move(n-1, src, aux, dest);
        b. Move(1, src, dest, aux);
        c. Move(n-1, aux, dest, src);
    Else
        Display "Move the top disk from ", src, " to ", dest.
    End if.
```

Coding

```
// ...

void Move(int n, char src, char dest, char aux)
{
  if (n > 1)
  {
    Move(n-1, src, aux, dest);
    Move(1, src, dest, aux);
    Move(n-1, aux, dest, src);
  }
  else
    cout << “Move the top disk from “
         << src << “ to “ << dest << endl;
}
```

Testing

```
The Hanoi Towers

Enter how many disks: 1
Move the top disk from A to B
```

Testing (*Ct'd*)

```
The Hanoi Towers

Enter how many disks: 2
Move the top disk from A to C
Move the top disk from A to B
Move the top disk from C to B
```

Testing (*Ct'd*)

```
The Hanoi Towers

Enter how many disks: 3
Move the top disk from A to B
Move the top disk from A to C
Move the top disk from B to C
Move the top disk from A to B
Move the top disk from C to A
Move the top disk from C to B
Move the top disk from A to B
```

Testing (*Ct'd*)

```
The Hanoi Towers

Enter how many disks: 4
move a disk from needle A to needle B
move a disk from needle C to needle B
move a disk from needle A to needle C
move a disk from needle B to needle A
move a disk from needle B to needle C
move a disk from needle A to needle C
move a disk from needle A to needle B
move a disk from needle C to needle B
move a disk from needle C to needle A
move a disk from needle B to needle A
move a disk from needle C to needle B
move a disk from needle A to needle C
move a disk from needle A to needle B
move a disk from needle C to needle B
```

Analysis

Let’s see how many moves” it takes to solve this problem, as a function of *n*, the number of disks to be moved.

n	Number of disk-moves required
1	1
2	3
3	7
4	15
5	31
...	
i	2*i*-1
64	264-1 (a big number)

Analysis (*Ct'd*)

How big?

Suppose that our computer and "super-printer" can generate and print 1,048,576 (220) instructions/second.

How long will it take to print the priest's instructions?

- There are 264 instructions to print.
  - Then it will take 264/220 = 244 *seconds* to print them.
- 1 minute == 60 seconds.
  - Let's take 64 = 26 as an approximation of 60.
  - Then it will take ≅ 244 / 26 = 238 *minutes* to print them.

Analysis (*Ct'd*)

Hmm. 238 minutes is hard to grasp. Let's keep going...

- 1 hour == 60 minutes.
  - Let's take 64 = 26 as an approximation of 60.
  - Then it will take ≅ 238 / 26 = 232 *hours* to print them.
- 1 day == 24 hours.
  - Let's take 32 = 25 as an approximation of 24.
  - Then it will take ≅ 232 / 25 = 227 *days* to print them.

Analysis (*Ct'd*)

Hmm. 227 days is hard to grasp. Let's keep going...

- 1 year == 365 days.
  - Let's take 512 = 29 as an approximation of 365.
  - Then it will take ≅ 227 / 29 = 218 *years* to print them.
- 1 century == 100 years.
  - Let's take 128 = 27 as an approximation of 100.
  - Then it will take ≅ 218 / 27 = 211 *centuries* to print them.

Analysis (*Ct'd*)

Hmm. 211 centuries is hard to grasp. Let's keep going...

- 1 millenium == 10 centuries.
  - Let's take 16 = 24 as an approximation of 10.
  - Then it will take ≅ 211 / 24 = 27 = *128 millenia* just to *print* the priest's instructions (assuming our computer doesn't crash, in which case we have to start all over again).

How fast can the priests actually *move* the disks?

I'll leave it to you to calculate the data of the apocalypse...

Summary

Recursion is a valuable tool that allows some problems to be solved in an elegant and efficient manner.

Functions can sometimes require more than one recursive call in order to accomplish their task.

There are problems for which we can design a solution, but the nature of the problem makes solving it *effectively uncomputable*.