DATA STRUCTURE & ALGORITHM

Introduction

Lecture 3

Instructor: Engr. Tayyab Abdul Hannan
Email: tayyab.abdulhannan@gmail.com

Recursivity.

Recursivity is the property that functions have to be called by themselves. It is useful for tasks such as some sorting methods or to calculate the factorial of a number.

FACTORIAL FUNCTION – Linear recursion
The Factorial Function is defined mathematically by

$$f(n) = n! = \begin{cases} 1 & \text{if } n = 0 \\ n*(n-1)! & \text{If } n > 0 \end{cases}$$

C++ implementation is a Direct Translation of mathematical expression

```
long f(int n)
{
   if (n<2) return 1;    \\basis
   return  n*f(n-1);     \\recursion
}
```

Recursivity.

```
// factorial calculator
#include <iostream.h>

long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return 1;
}

void main ()
{
cout << factorial (4);
}
```

Tracing Recursive Calls.

```
void main ()
{
cout << factorial (4);
}
```

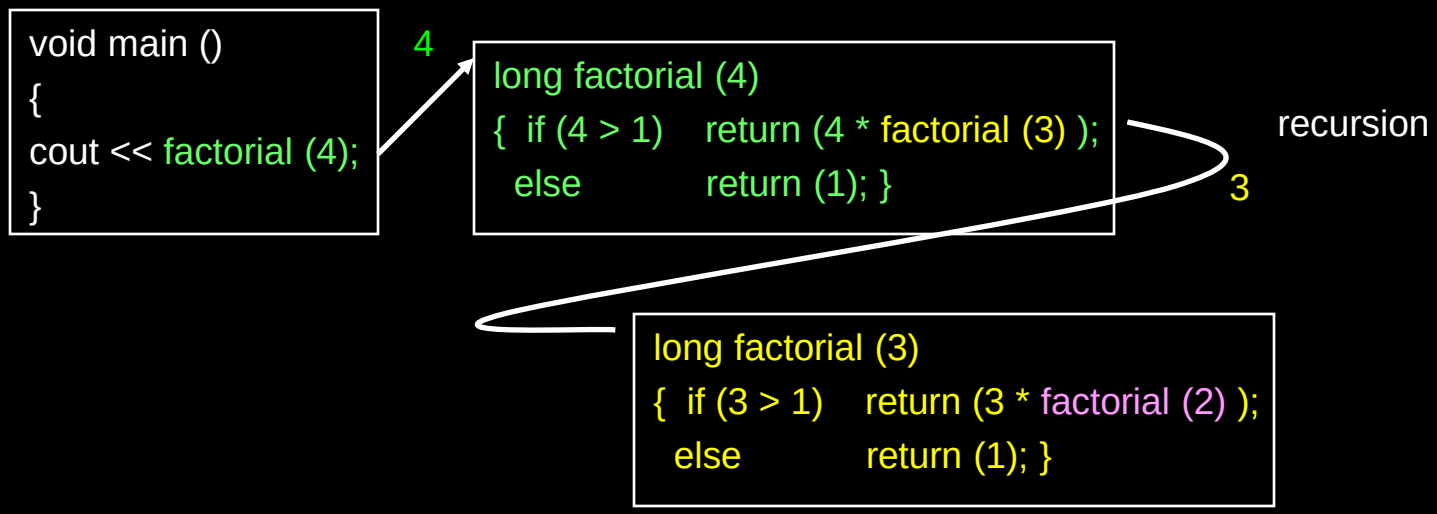
Tracing Recursive Calls.

```
void main ()
{
cout << factorial (4);
}
```

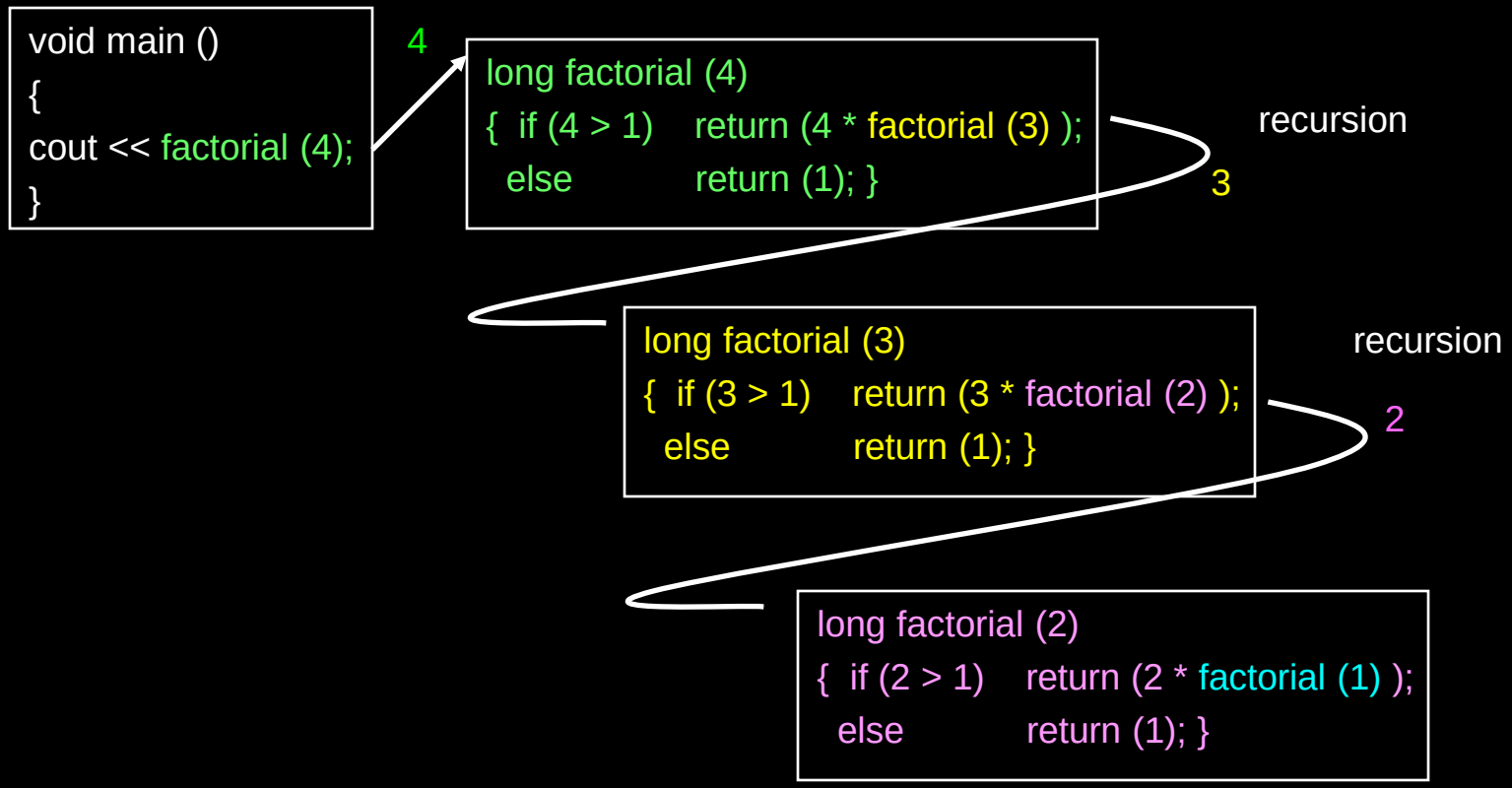
4

```
long factorial (4)
{  if (4 > 1)    return (4 * factorial (3) );
  else           return (1); }
```

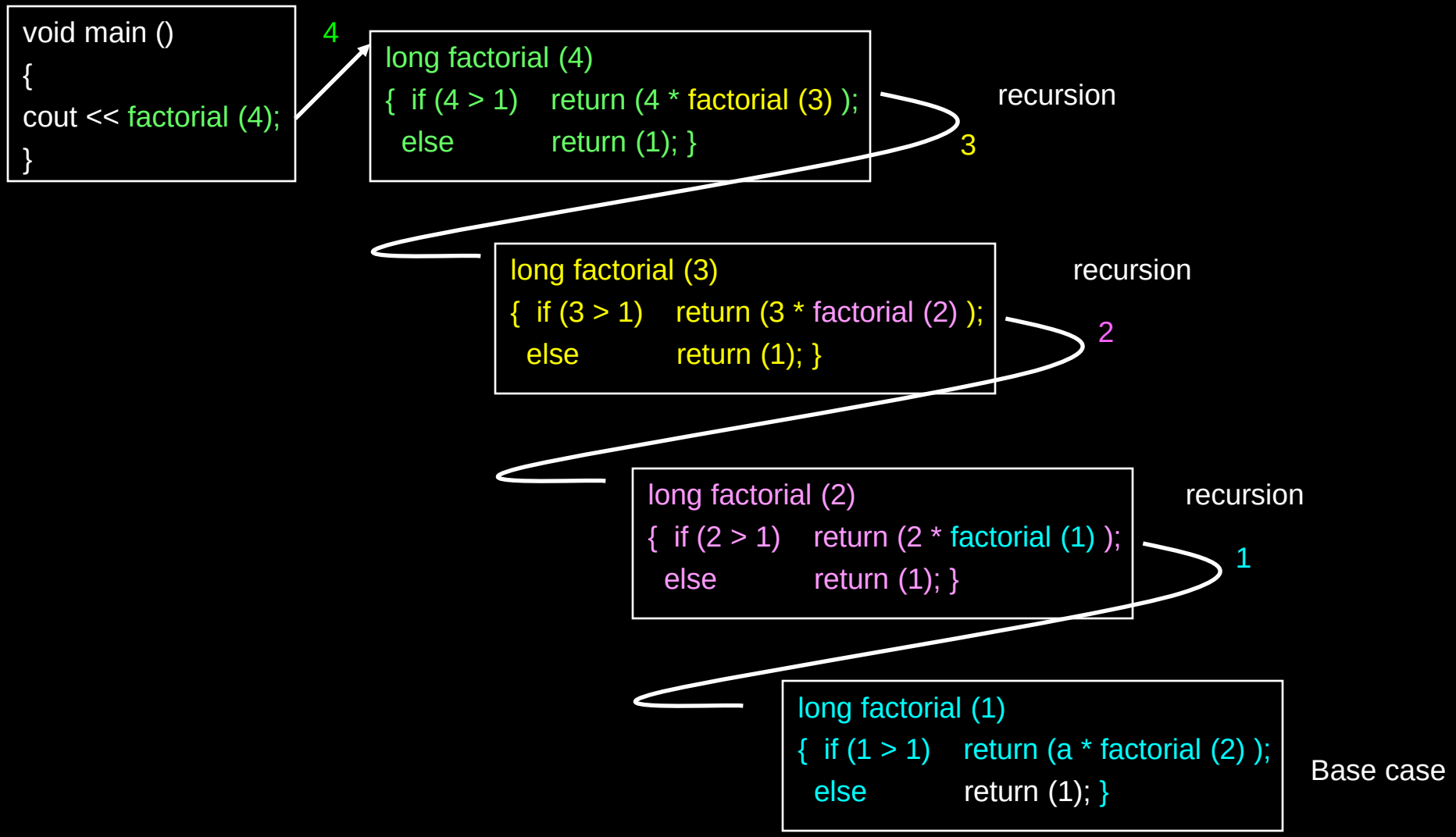
Tracing Recursive Calls.



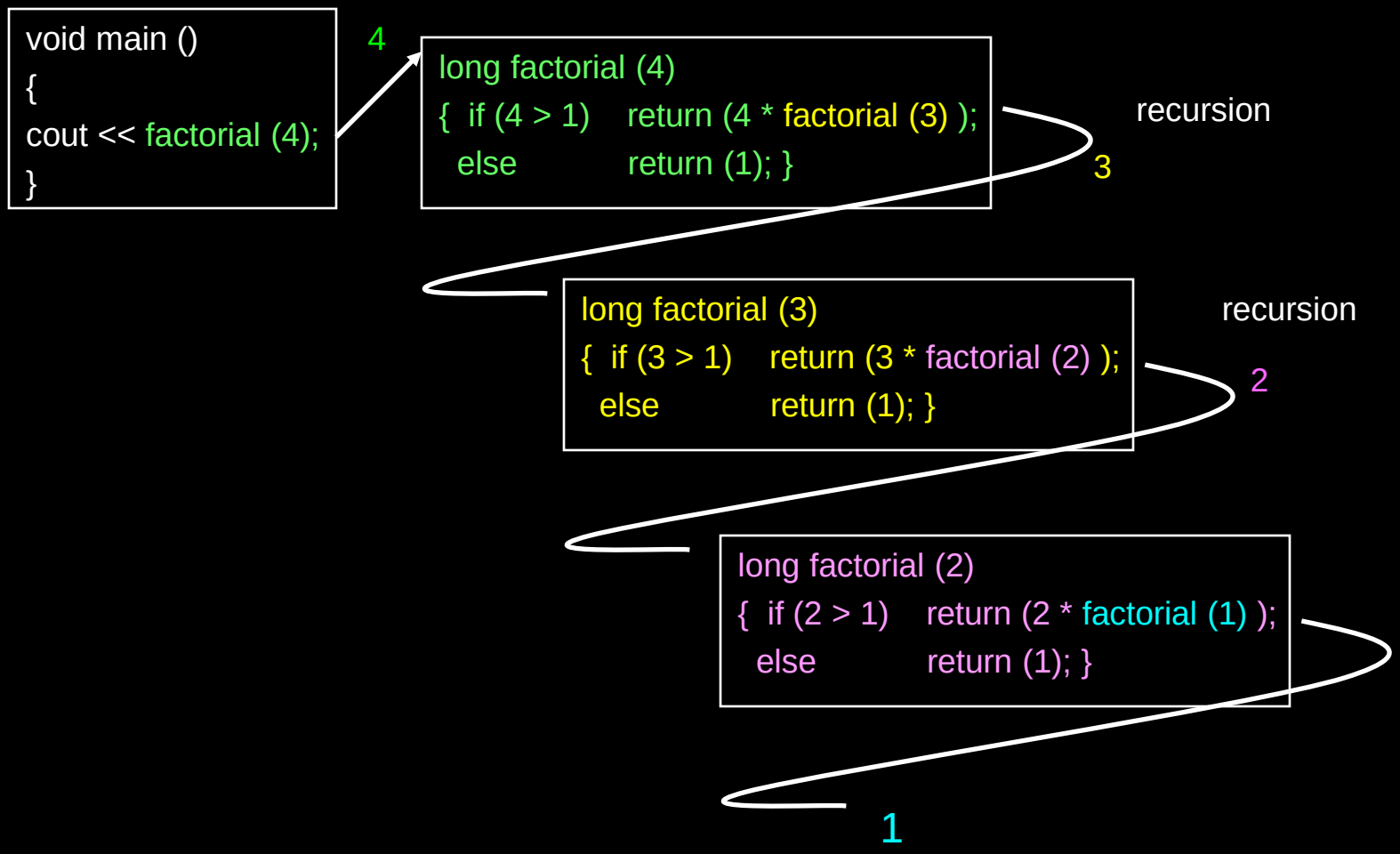
Tracing Recursive Calls.



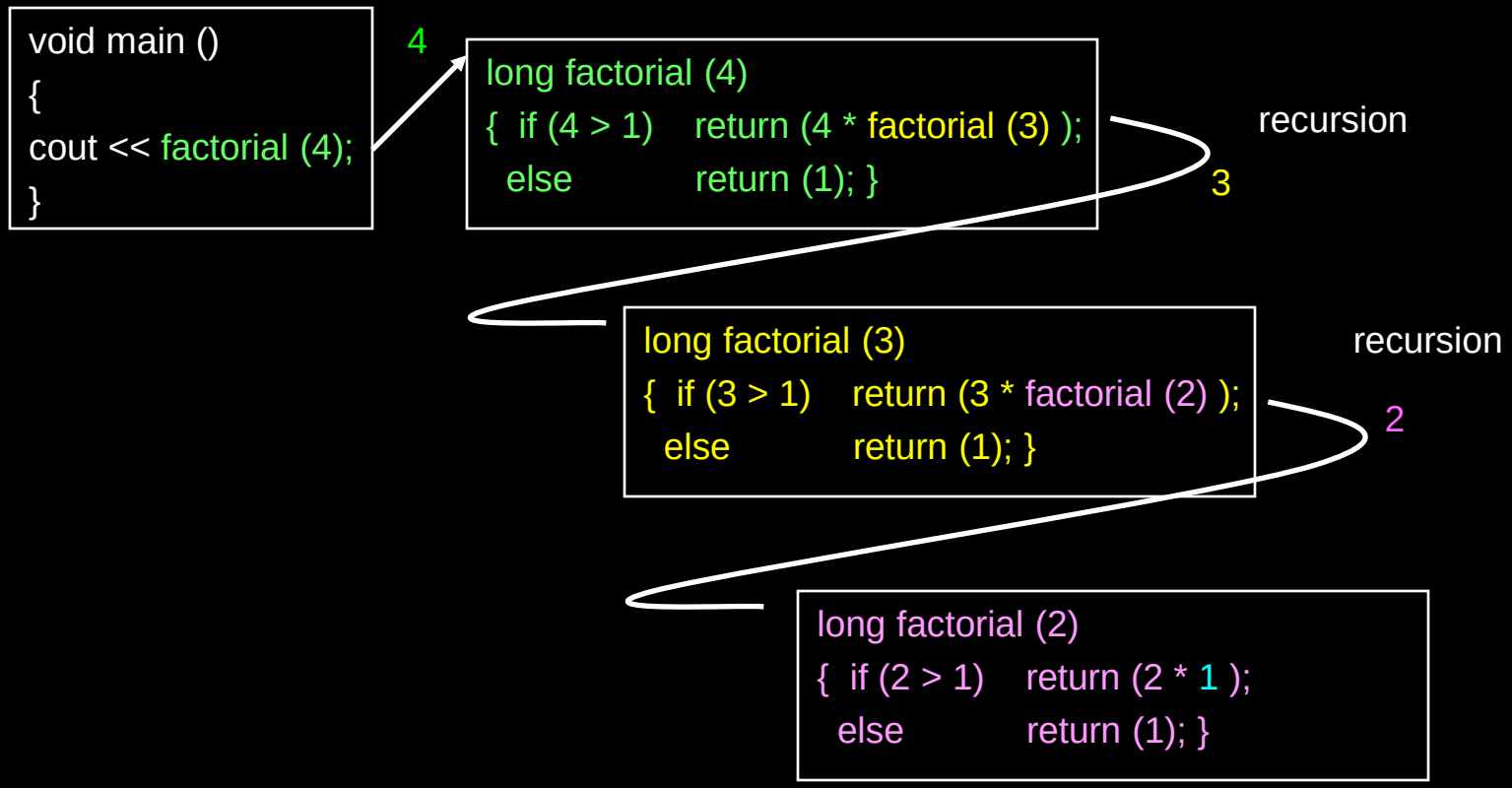
Tracing Recursive Calls.



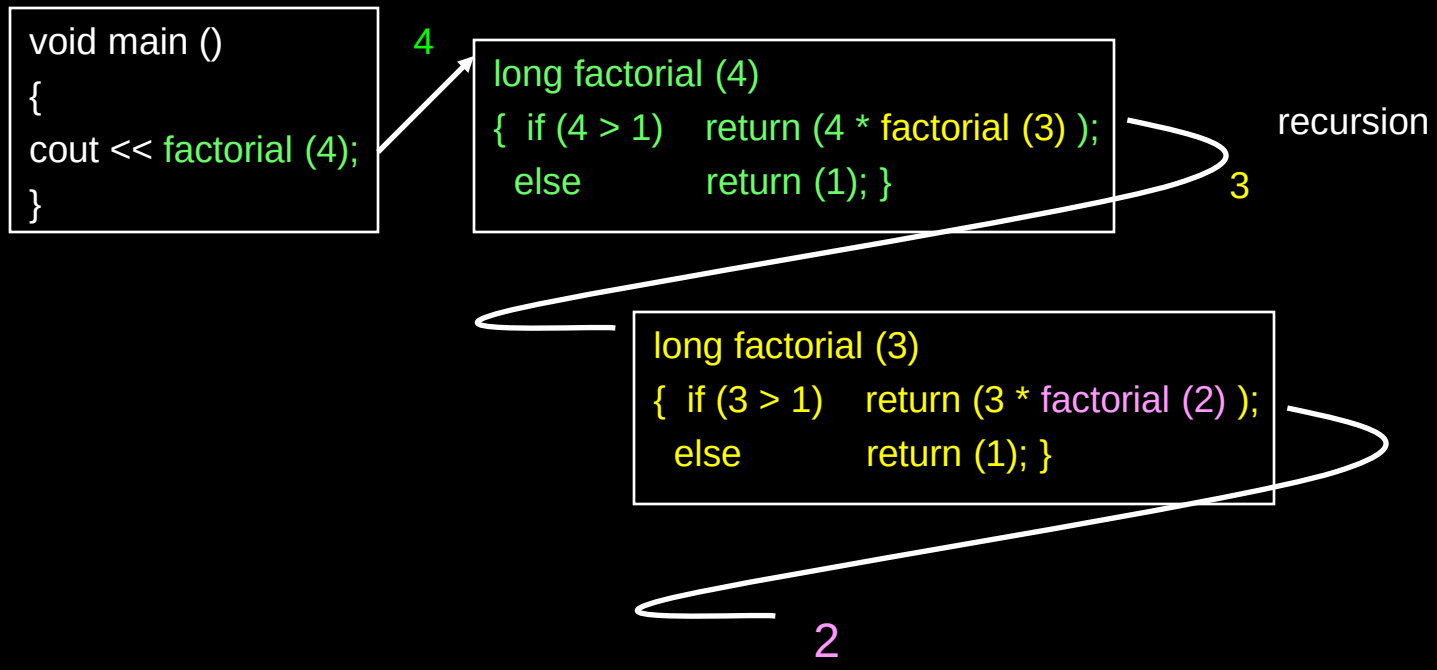
Tracing Recursive Calls.



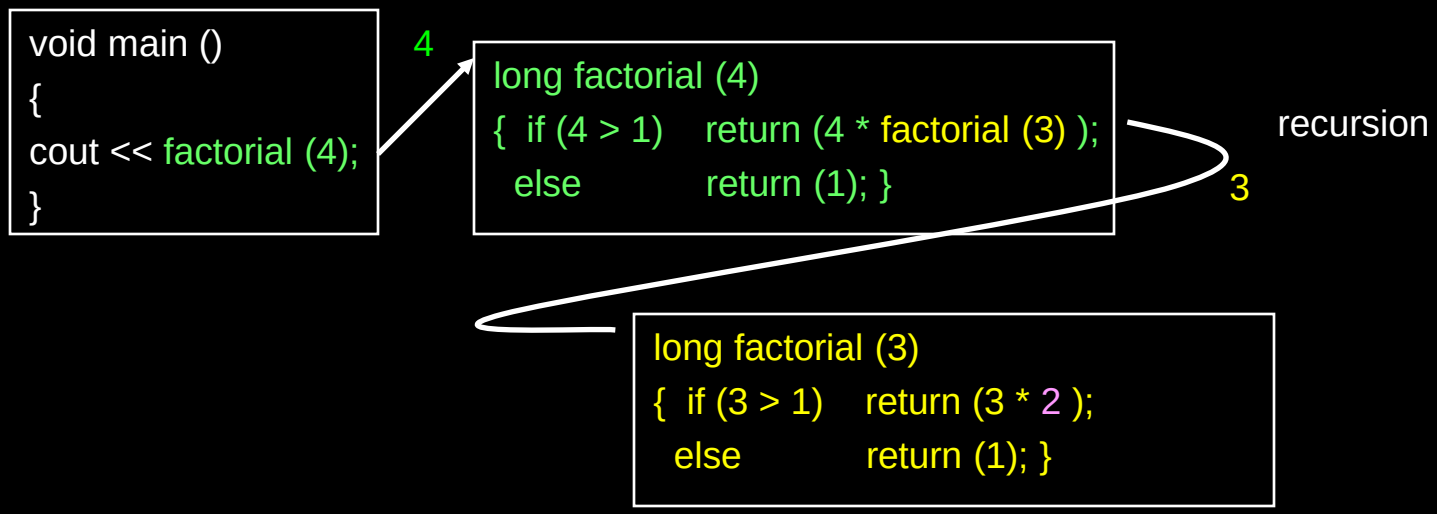
Tracing Recursive Calls.



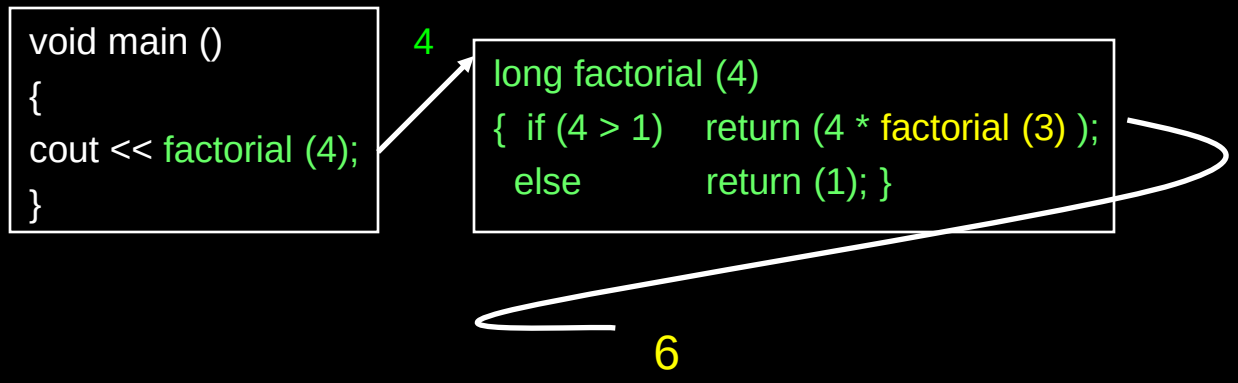
Tracing Recursive Calls.



Tracing Recursive Calls.



Tracing Recursive Calls.



Tracing Recursive Calls.

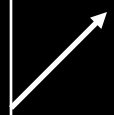
```
void main ()
{
cout << factorial (4);
}
```

4

```
long factorial (4)
{  if (4 > 1)    return (4 * 6 );
  else           return (1); }
```

Tracing Recursive Calls.

```
void main ()
{
cout << factorial (4);
}
```

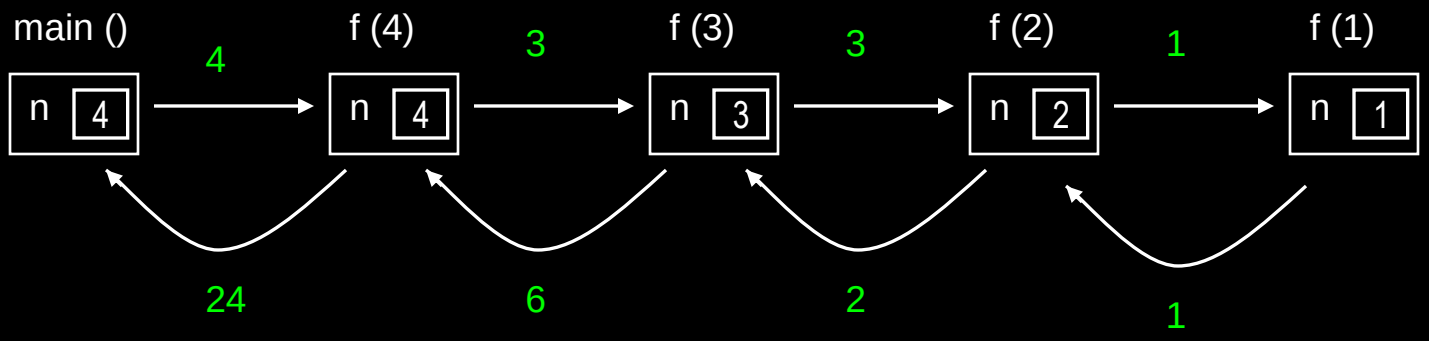


24

Tracing Recursive Calls.

```
void main ()
{
cout << 24 ;
}
```

Tracing Recursive Calls (Factorial Function).



Note that the call f(n) to the recursive implementation of the factorial function will generate n-1 recursive calls. This is very inefficient compared to the iterative implementation given below

```
long f(int n)
{  long f =1;
       for (int i=2; i<=n; i++)
                        f =f*I ;
   return f;
}
```

There is only one recursion call, so it is linear recursion
Depth of Recursion is 3

Recursivity.

```
// factorial calculator
#include <iostream.h>

long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return 1;
}

int main ()
{
  long l;
  cout << "Type a number: ";
  cin >> l;
  cout << "!" << l << " = " << factorial (l);
  return 0;
}
```

Recursivity – Fibonacci Sequence.
(Binary Recursion)

Fibonacci sequence is 0,1,1,2,3,5,8,13,21,34,55,…

i.e., First Two terms are 0 and 1, Each number after the second is the sum of the two preceding numbers

Mathematically

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n>1 \end{cases}$$

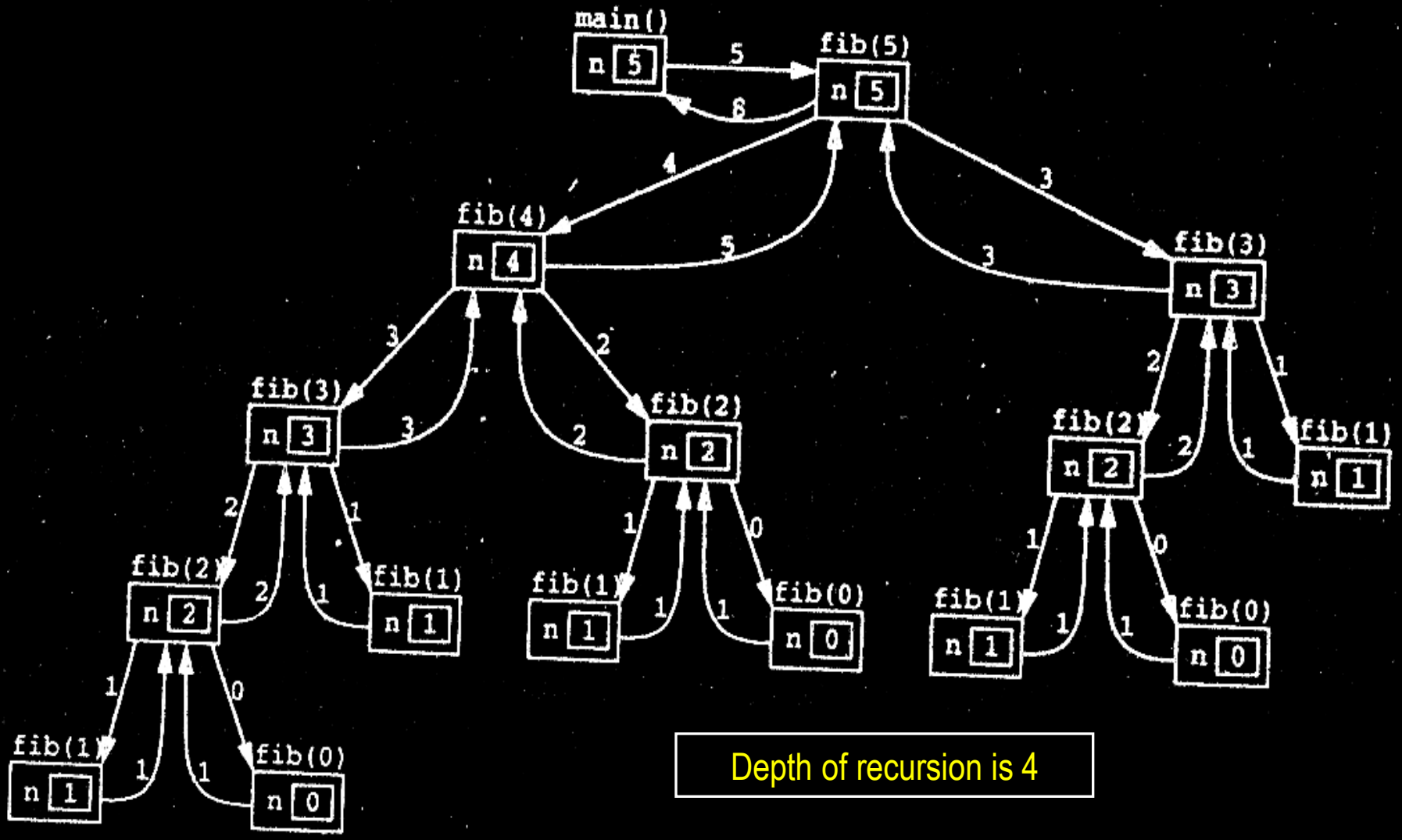
n	F_n
0	0
1	1
2	1
3	2
4	3
5	5
6	8

```
Long fib(int n)
{    if (n<2)     return n;        // basis
     return fib(n-1)+fib(n-2);    // recursion
}
```

Each call to function will generate two recursive calls, so binary recursion

Recursivity – Fibonacci Sequence.

```
Long fib(int n)
{   if (n<2)    return n;       // basis
    return fib(n-1)+fib(n-2);   // recursion
}
```



Merge-sort *(1)*

- Idea for sorting an array:
  - Divide the array into two subarrays of about equal length.
  - Sort the subarrays separately.
  - Merge the sorted subarrays.
- This is an application of the **divide-and-conquer** strategy.

Divide-and-conquer strategy

- The **divide-and-conquer** strategy is effective for solving many problems.
- To solve a “hard” problem:
  - Break the problem down into two or more “easier” sub-problems.
  - Solve these sub-problems separately.
  - Combine their answers.

Merge-sort *(2)*

- **Merge-sort algorithm**:

  To sort *a*[*left*…*right*] into ascending order:

  1. If *left* < *right*:
     1.1. Let *m* be an integer about midway between *left* and *right*.
     1.2. Sort *a*[*left*…*m*] into ascending order.
     1.3. Sort *a*[*m*+1…*right*] into ascending order.
     1.4. Merge *a*[*left*…*m*] and *a*[*m*+1…*right*] into auxiliary array *b*.
     1.5. Copy all components of *b* into *a*[*left*…*right*].
  2. Terminate.

Merge-sort *(3)*

- Animation:

To sort *a*[*left*…*right*] into ascending order:

1. If *left* < *right*:
   1.1. Let *m* be an integer about midway between *left* and *right*.
   1.2. Sort *a*[*left*…*m*] into ascending order.
   1.3. Sort *a*[*m*+1…*right*] into ascending order.
   1.4. Merge *a*[*left*…*m*] and *a*[*m*+1…*right*] into auxiliary array *b*.
   1.5. Copy all components of *b* into *a*[*left*…*right*].
2. Terminate.

left = 0	1	2	3	4	5	6	7	8 = *right*
a	cat	cow	dog	fox	goat	lion	pig	rat	tiger

Divide and Conquer Strategy

- The ancient Roman politicians understood an important principle of good algorithm design.
- You divide your enemies (by getting them to distrust each other) and then conquer them piece by piece.
- This is called *divide-and-conquer*.
- In algorithm design, the idea is to take a problem on a large input, break the input into smaller pieces, solve the problem on each of the small pieces, and then combine the piecewise solutions into a global solution.

Divide and Conquer Strategy

- The process ends when you are left with such tiny pieces remaining (e.g. one or two items) that it is trivial to solve them.
- Summarizing, the main elements to a divide-and-conquer solution are
- **Divide:** the problem into a small number of pieces
- **Conquer:** solve each piece by applying divide and conquer to it *recursively*
- **Combine:** the pieces together into a global solution.

Merge Sort

- Divide and conquer strategy is applicable in a huge number of computational problems.
- The first example of divide and conquer algorithm we will discuss is a simple and efficient sorting procedure called We are given a sequence of n numbers A, which we will assume are stored in an array A[1..n].
- The objective is to output a permutation of this sequence sorted in increasing order.

Merge Sort

- Here is how the merge sort algorithm works:
- (Divide:) split A down the middle into two subsequences, each of size roughly n/2
- (Conquer:) sort each subsequence by calling merge sort recursively on each.
- (Combine:) merge the two sorted subsequences into a single sorted list.

Merge Sort

- We'll assume that the procedure that merges two sorted list is available to us.
- We'll implement it later.
- Because the algorithm is called recursively on sublists, in addition to passing in the array itself, we will pass in two indices, which indicate the first and last indices of the sub-array that we are to sort.
- The call MergeSort(A, s, e) will sort the sub-array A[s : e] and return the sorted result in the same sub-array

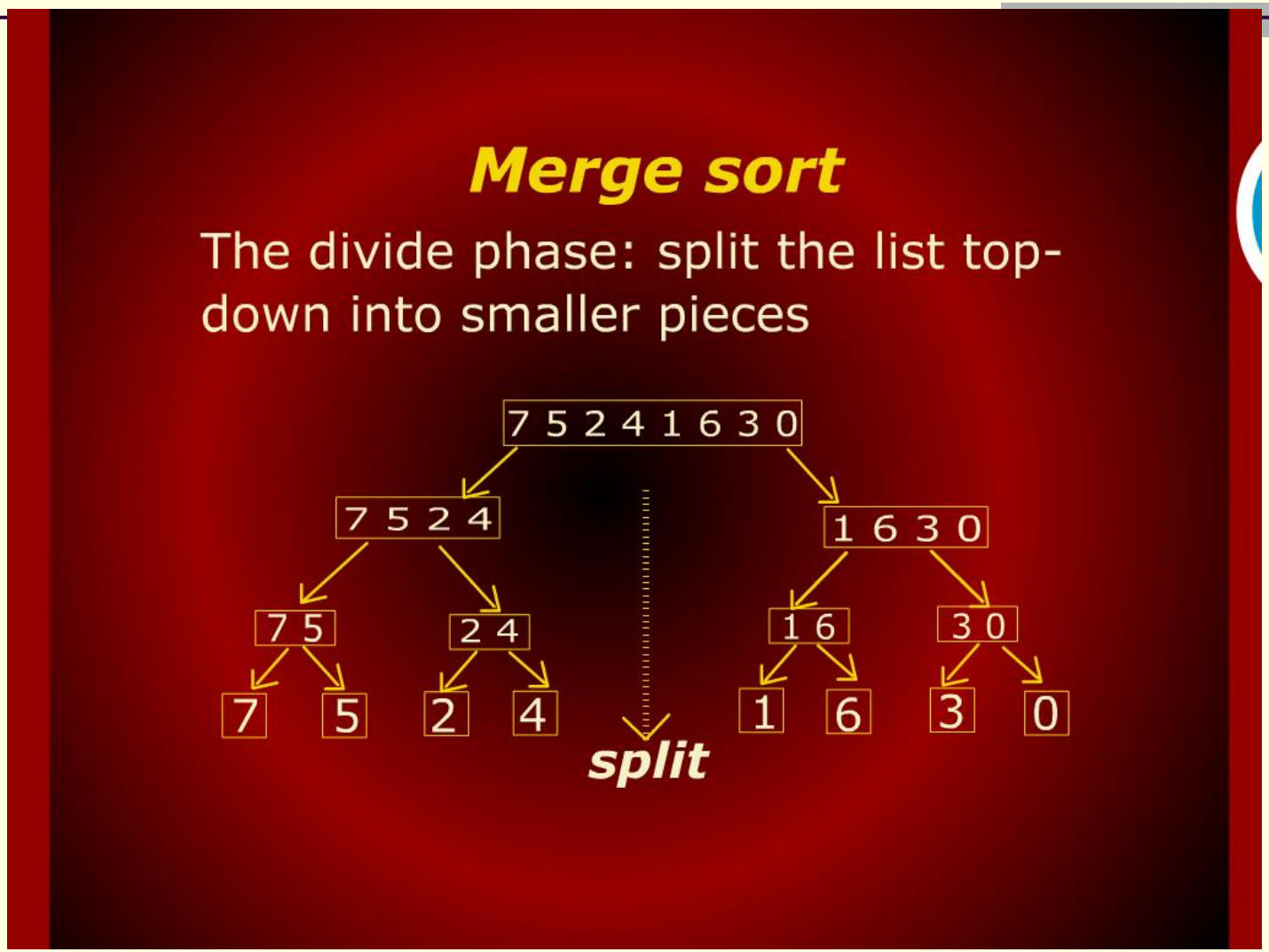
Merge Sort

- If e = s, then this means that there is only one element to sort, and we may return immediately.
- Otherwise if (s ≠ e) there are at least two elements, and we will invoke the divide-and-conquer.
- We find the index mid, midway between p and r, namely mid= (s + e)/2.
- Then we split the array into sub-arrays A[s : mid] and A[mid + 1 : e].
- Call MergeSort recursively to sort each sub-array.
- Finally, we invoke a procedure which merges these two sub-arrays into a single sorted array.

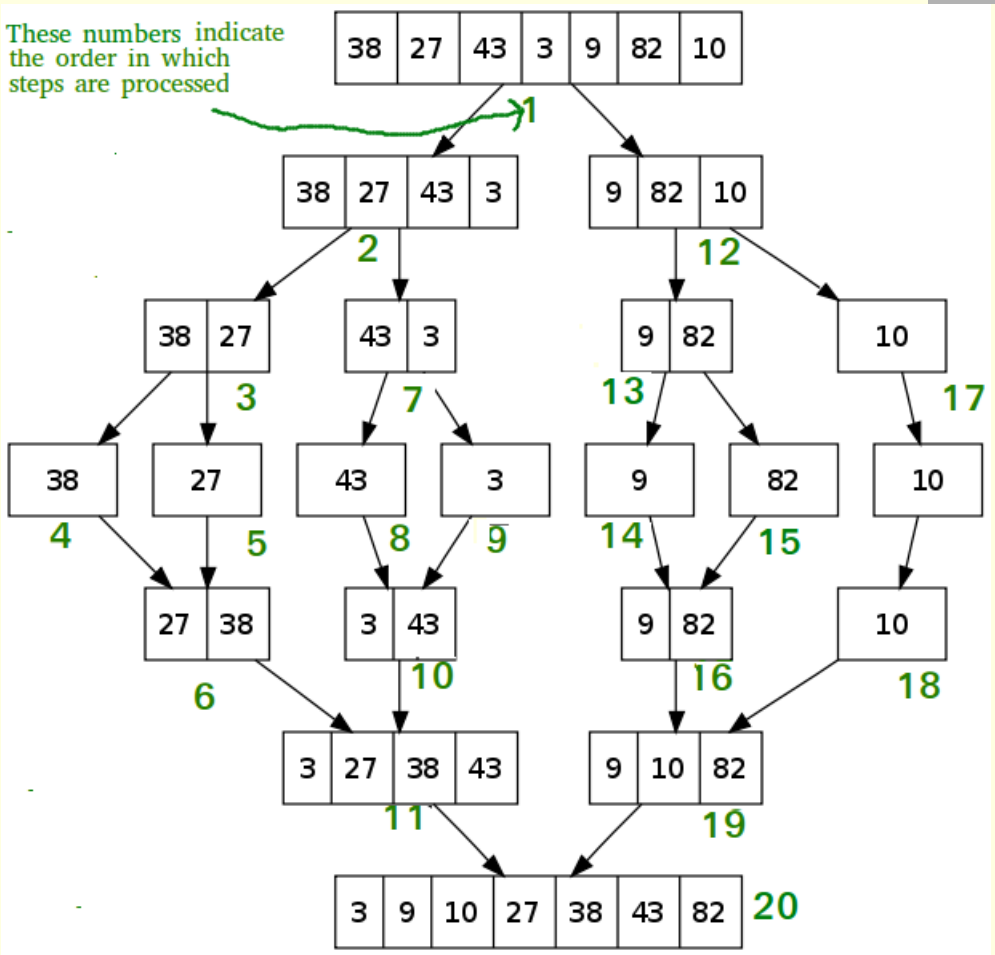
Merge Sort

- Here is the MergeSort algorithm.
- MERGE-SORT(array A, int s, int e)
- 1 **if** (s < e)
- 2 **then**
- 3 mid←(s + e)/2
- 4 MERGE-SORT(A, s, mid) // sort A[s..mid]
- 5 MERGE-SORT(A, mid+ 1,e) // sortA[mid+ 1..e]
- 6 MERGE(A, s, mid, e)// merge the two pieces

Merge Sort


Merge sort
The divide phase: split the list top-down into smaller pieces
7 5 2 4 1 6 3 0
7 5 2 4
1 6 3 0
7 5
2 4
1 6
3 0
7
5
2
4
1
6
3
0
split

Merge Sort


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

Merge Sort

- **Merging:** All that is left is to describe the procedure that merges two sorted lists.
- Merge(A, s, mid, re) assumes that the left sub-array, A[s : mid], and the right sub-array, A[mid + 1 : e], have already been sorted.
- We merge these two sub-arrays by copying the elements to a temporary working array called B.
- For convenience, we will assume that the array B has the same index range A, that is, B[s : e].

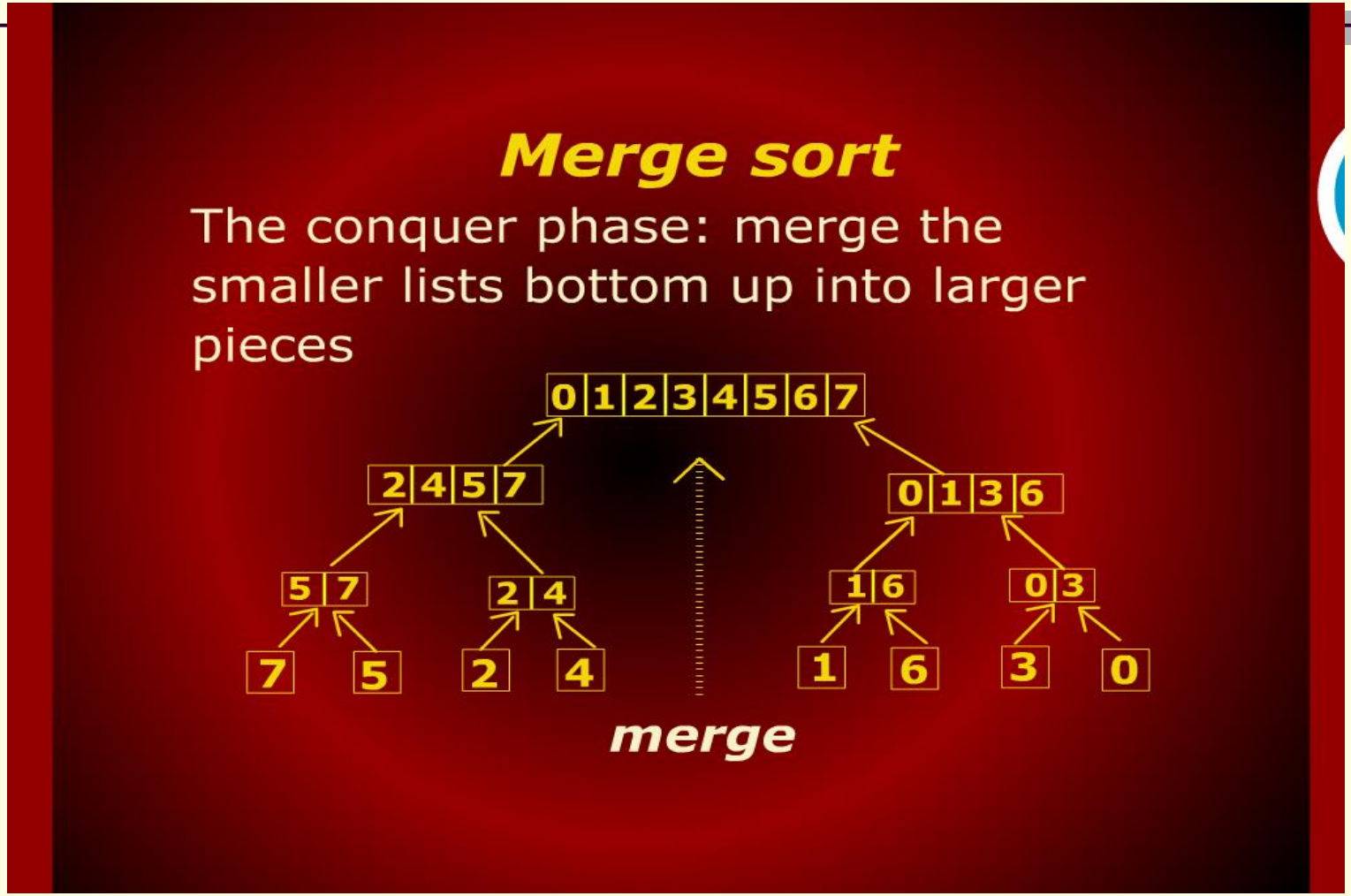
Merge Sort

- We have two indices i and j, that point to the current elements of each sub-array.
- We move the smaller element into the next position of B (indicated by index k) and then increment the corresponding index (either i or j).
- When we run out of elements in one array, then we just copy the rest of the other array into B.
- Finally, we copy the entire contents of B back into A.

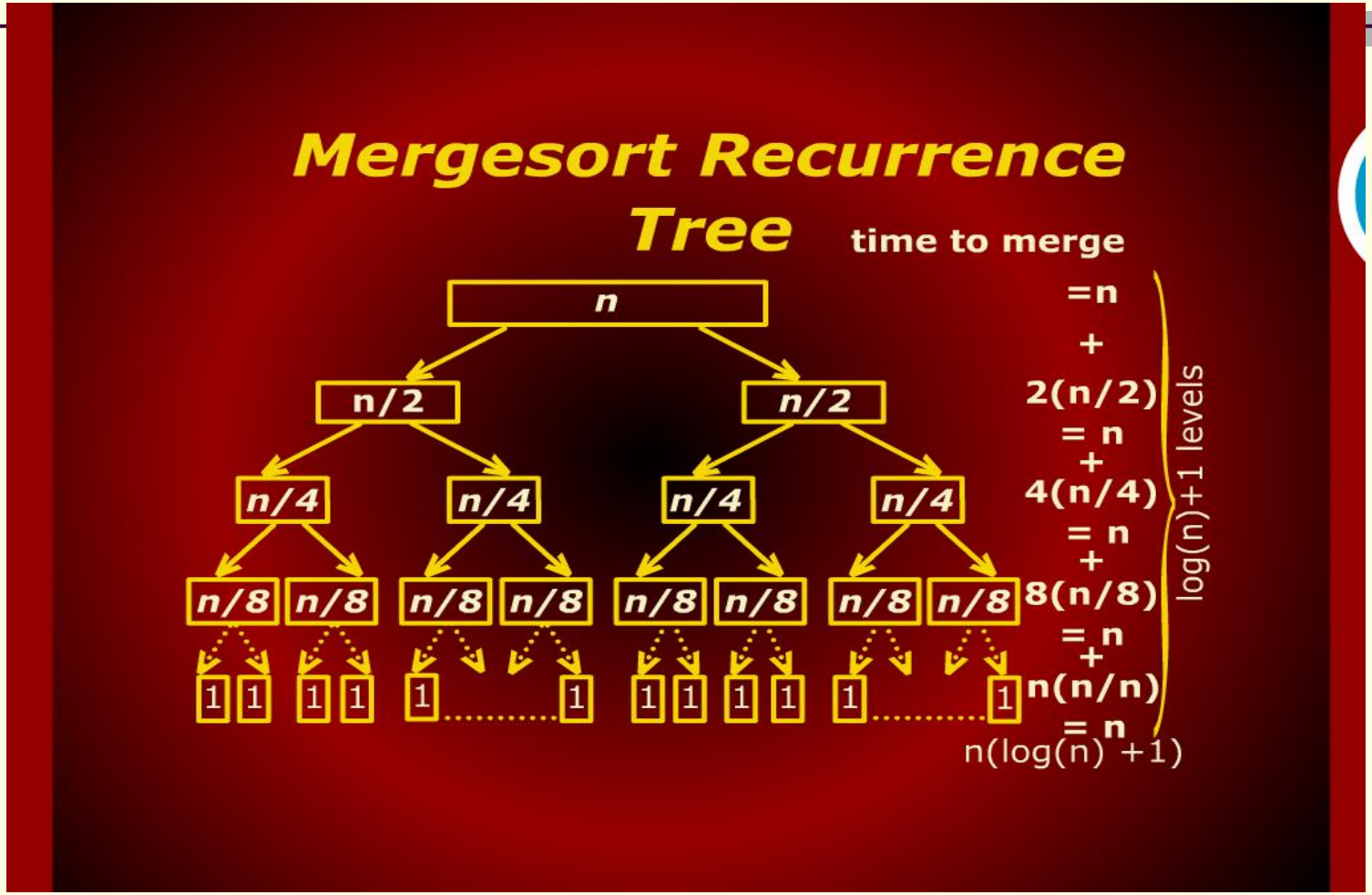
Merge Sort

- MERGE(array A, int s, int mid int e)
- 1 int B[s..e]; int i←k←s; int j←mid + 1
- 2 **while** (i ≤ mid) and (j≤ e)
- 3 **do if** (A[i] ≤ A[j])
- 4 **then** B[k++]←A[i++]
- 5 **else** B[k++]← A[j++]
- 6 **while** (i ≤ mid)
- 7 **do** B[k++] ← A[i++]
- 8 **while** (j ≤ e)
- 9 **do** B[k++] ← A[j++]
- 10 **for** i ← s **to** e
- 11 **do** A[i] ← B[i]

Merge Sort


Merge sort
The conquer phase: merge the smaller lists bottom up into larger pieces
0 1 2 3 4 5 6 7
2 4 5 7
0 1 3 6
5 7
2 4
1 6
0 3
7
5
2
4
1
6
3
0
merge

Merge Sort


Mergesort Recurrence Tree
time to merge
n
n/2
n/2
n/4
n/4
n/4
n/4
n/8
n/8
n/8
n/8
n/8
n/8
n/8
n/8
1
=n
+
2(n/2)
= n
+
4(n/4)
= n
+
8(n/8)
= n
+
n(n/n)
= n
log(n)+1 levels
n(log(n)+1)

Merge-sort

- Analysis (counting comparisons):

  Let $n = right - left + 1$ be the length of the array. Let the total no. of comparisons required to sort n values be $comps(n)$.

  The left subarray's length is about $n/2$, so step 1.2 takes about $comps(n/2)$ comparisons to sort it.

  Similarly, step 1.3 takes about $comps(n/2)$ comparisons to sort the right subarray.

  Step 1.4 takes about $n-1$ comparisons to merge the subarrays.

Merge-sort

- Analysis *(continued)*:

  Therefore:

  $$comps(n) \approx 2\, comps(n/2) + n - 1 \quad \text{if } n > 1$$
  $$comps(n) = 0 \quad \text{if } n \leq 1$$

  Solution:

  $$comps(n) \approx n \log_2 n$$

  Time complexity is $O(n \log n)$.

- Space complexity is $O(n)$, since step 1.4 needs an auxiliary array of length n.