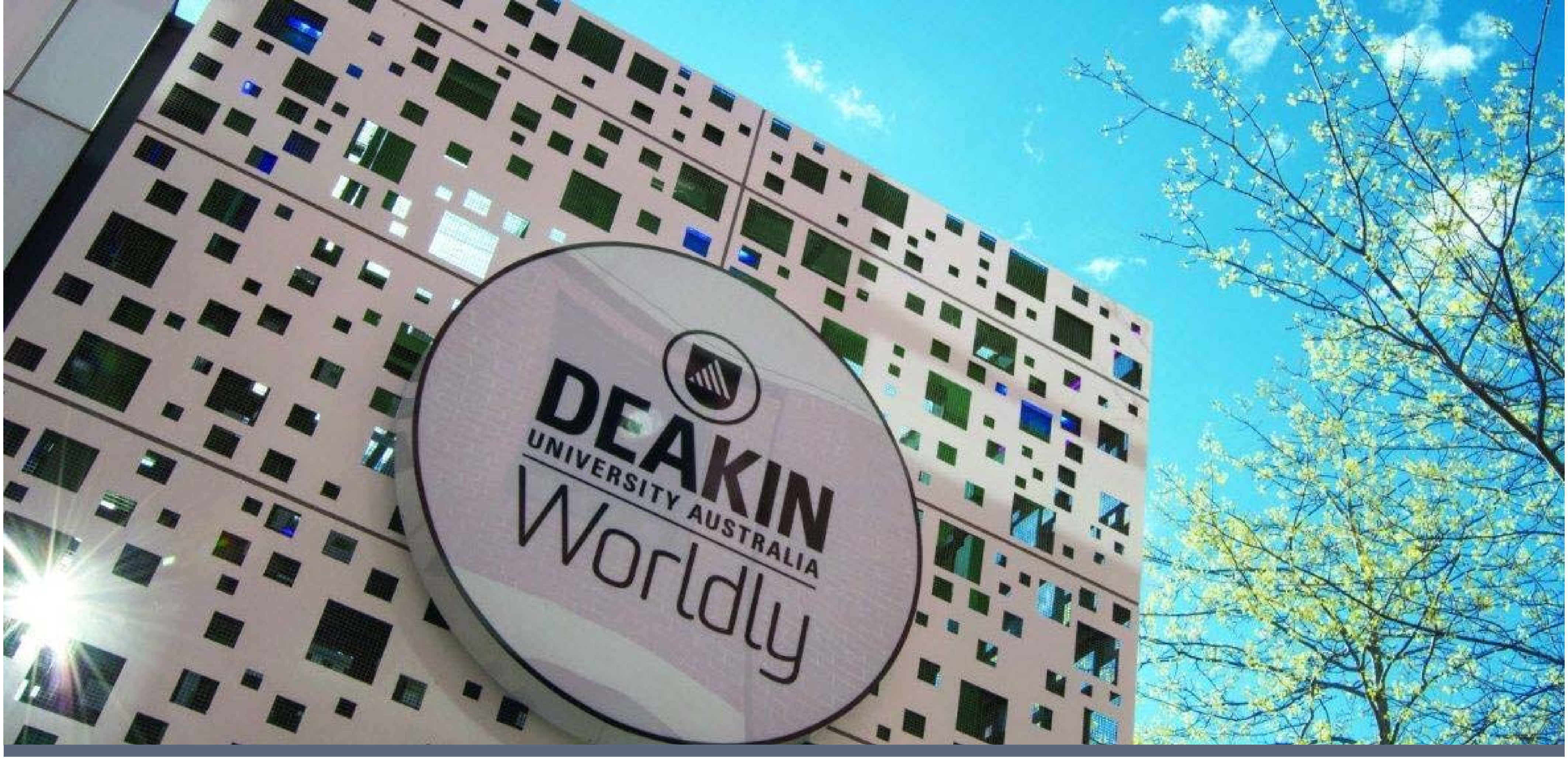


Lecture 2. Analysis of Algorithms. Recursion and its Application

SIT221 Data Structures and Algorithms

Algorithm Analysis

As the size of a problem grows for an algorithm

- Time: How much longer does it run?
- Space: How much memory does it use?

Space Complexity

Here is what moving a 5MB IBM Hard Drive in 1956 was like.

Is memory space still an issue now days?

Experimental study of algorithms

Why not just code the algorithm and time it?

- Hardware: processor(s), memory, and etc.
- Operating System, version of libraries, drivers
- Programs running in the background
- Implementation dependent
- Choice of input
- Number of inputs to test

All these factors may significantly affect the running time.

Experimental study of algorithms: Issues

Timing does not really evaluate the algorithm, but merely evaluates a specific implementation

- Use asymptotic analysis to evaluate an algorithm
  - Examine the algorithm itself, **not** the implementation
  - Reason about performance as a function of n
  - Mathematically prove things about performance

- Use timing to evaluate an implementation

  In the real world, we do want to know whether implementation A runs faster than implementation B on data set C

Linear Search: Complexity

```
int LinearSearch( int[] A, int n, int value )
{
    for ( int i = 0 ; i < n; i++ )
    {
        if ( A[i] == value )
            then return i;
    }
    return -1;
}
```

- Worst case: T(n) = $O(n)$ comparisons
- Best case: T(n) = $O(1)$ comparisons
- Average case: T(n) = $O(n)$ comparisons
- Worst-case space complexity: $O(1)$ iterative

Linear Search: Average Case Analysis

- We shall focus on the probable position for the desired element x, rather than the elements of the array A.
- Since for any input A, the element x is equally likely to be present in any location of A. Therefore, $\Pr[A[i] = x] = \frac{1}{n}$ for $1 \leq i \leq n$.
- The cost of accessing the i^{th} location is i and the associated probability is $\frac{1}{n}$. Therefore, the expected cost is

$$1 \times \frac{1}{n} + 2 \times \frac{1}{n} + \cdots + n \times \frac{1}{n} = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}.$$

- Finally, $\frac{n+1}{2} = O(n)$ comparisons on average.

Bubble Sort: Complexity

Given is an array A of size n of integer numbers.

```
for ( i = 1 ; i < n ; i++ )
{
        int swaps = 0;
        for ( j = 0 ; j < n – i; j++ )
        {
                if ( a[j] > a[j + 1] )
                {
                        swap( a[j], a[j + 1] );
                        swaps = swaps + 1;
                }
        }
        if ( swaps == 0 )  then  break;
}
```

- Worst case: T(n) = $O(n^2)$ comparisons
- Best case: T(n) = $O(n)$ comparisons
- Average case: T(n) = $O(n^2)$ comparisons
- Worst-case space complexity: $O(1)$ auxiliary

Insertion Sort: Complexity

Given is an array A of size n of integer numbers.

```
for ( i = 1 ; i < n ; i++ )
{
        j = i;
        while ( ( j>0 ) && ( A[j-1] > A[j] ) )
        {
                swap( A[j], A[j-1] );
                j = j – 1;
        }
}
```

- Worst case: T(n) = $O(n^2)$ comparisons
- Best case: T(n) = $O(n)$ comparisons
- Average case: T(n) = $O(n^2)$ comparisons
- Worst-case space complexity: $O(1)$ auxiliary

Selection Sort: Complexity

Given is an array A of size n of integer numbers.

```
for ( int i = 0 ; i < n; i++ )
{
        int min = i;
        for ( int j = i+1; j < n; j++ )
        {
                if ( A[j] < A[min] )  then  min = j;
        }
        if ( min != i )  then  swap( A[i], A[min] );
}
```

- Worst case: T(n) = $O(n^2)$ comparisons
- Best case: T(n) = $O(n^2)$ comparisons
- Average case: T(n) = $O(n^2)$ comparisons
- Worst-case space complexity: $O(1)$ auxiliary

Simple Integer Arithmetics: Complexity Analysis

We want to have algorithms to carry out

- Addition
- Multiplication

of two numbers.

Recall what you have learned in school!

Representation of integer numbers

- Assume that integers are represented as digit strings stored in an array a of size n containing n digits

$$a_{n-1}a_{n-2}\ldots a_1a_0$$

- Base B number system, $B > 1$
- Digits $0, 1, \ldots, B-1$

Then $\sum_{i=0}^{n-1} a_i B^i$ represents our integer.

Representation of integer numbers

$$\boxed{\sum_{i=0}^{n-1} a_i B^i}$$

- For $B = 10$, "924" is represented as

$$9 \cdot 10^2 + 2 \cdot 10^1 + 4 \cdot 10^0 = 924$$

- For $B = 2$, "10101" is represented as

$$1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 21$$

Choice of primitive operations

Assume that we have access to

- addition of 3 digits with a 2 digits result (full adder);
- multiplication of 2 digits with 2 digits result.

For example:

Addition: $$\begin{array}{r} 3 \\ 5 \\ 5 \\ \hline 13 \end{array}$$

Multiplication: $6 \cdot 7 = 42$

School Method for Addition: Example

$a = 1289, \quad b = 1342, \quad \boxed{B = 10 \quad \text{base}}$

$$
\begin{array}{llr}
 & a & 1289 \\
+ & & \\
 & b & 1342 \\
\hline
 & c & 00110 \\
\hline\hline
 & \text{sum} & 02631
\end{array}
$$

School Method for Addition: Analysis

Input: Two integers $a = (a_{n-1} \dots a_0)$ and $b = (b_{n-1} \dots b_0)$

Compute: $s = a + b$, where $s = (s_n \dots s_0)$ is an $n + 1$ digit number.

$$
\begin{array}{cccccl}
 & a_{n-1} & \dots & a_1 & a_0 & \text{first operand} \\
 & b_{n-1} & \dots & b_1 & b_0 & \text{second operand} \\
c_n & c_{n-1} & \dots & c_1 & 0 & \text{carries} \\
\hline
s_n & s_{n-1} & \dots & s_1 & s_0 & \text{sum}
\end{array}
$$

$c_n \dots c_0$ is a sequence of carries.

$c_0 = 0$

$c_{i+1} \cdot B + s_i = a_i + b_i + c_i, 0 \le i < n$ (invariant holds)

$s_n = c_n$

School Method for Addition: Analysis

Pseudo-code:

```
c = 0;
for (i = 0; i < n; c + +)
{
    s_i = a_i + b_i + c;
    c = 0;
    if ( s_i ≥ B )  then
    {
        s_i = s_i − B;
        c = 1;
    }
}
s_n = c;
```

Theorem: The addition of two n–digit integers requires exactly n primitive operations. The result is an $n+1$ integer.

School Method for Multiplication: Example

$$a = 342, \quad b = 26, \quad \boxed{B = 10 \qquad \text{base}}$$

1. Compute partial products

$$a \cdot b_0 = 342 \cdot 6 \implies p_0 = 2052$$
$$a \cdot b_1 = 342 \cdot 2 \implies p_1 = 684$$

2. Sum up aligned partial products

$$\begin{array}{r} 2052 \\ + \; 6840 \\ = 8892 \end{array}$$

School Method for Multiplication: Analysis

Input: Two integers $a = (a_{n-1} \dots a_0)$ and $b = (b_{n-1} \dots b_0)$

Compute: $p = a \cdot b$, where $p = (p_{2n-1} \dots p_0)$ is a $2n$ digit number.

Procedure:

1. Multiply n −digit integer a by a one-digit integer b_j to obtain partial product p_j, $0 \leq j \leq n - 1$.
2. Sum up the aligned products $p_j \cdot B^j$.

School Method for Multiplication: Analysis

To compute partial product $p_j = a \cdot b_j$,

1. Compute for each i, $0 \le i \le n-1$, c_i and d_i such that

$$a_i \cdot b_j = c_i \cdot B + d_i$$

2. Form two integers

$$c = (c_{n-1} \dots c_0 0) \text{ and } d = (d_{n-1} \dots d_0)$$

3. Add c and d to obtain $p_j = a \cdot b_j$

$$\begin{array}{cccccccc} c_{n-1} & c_{n-2} & \dots & c_i & c_{i-1} & \dots & c_0 & 0 \\ & d_{n-1} & \dots & d_{i+1} & d_i & \dots & d_1 & d_0 \\ \hline \end{array}$$

$$\text{sum of } c \text{ and } d$$

Number of primitive operations:

n multiplications, $n+1$ additions, $2n+1$ in total

School Method for Multiplication: Analysis

Example of $p_j = a \cdot b_j$ calculation:

$$a = 342,\ b_j = 6,\ B = 10$$

Computing c's and d's:

$$a_0 \cdot b_j = 2 \cdot 6 = 12$$
$$c_0 = 1, d_0 = 2$$

$$a_1 \cdot b_j = 4 \cdot 6 = 24$$
$$c_1 = 2, d_1 = 4$$

$$a_2 \cdot b_j = 3 \cdot 6 = 18$$
$$c_2 = 1,\ d_2 = 8$$

Summing up:

$$\begin{array}{r} 1210 \\ +\quad 842 \\ = 2052 \end{array}$$

Encoding:

$$\begin{array}{l} (\, c_2\, c_1\, c_0\ \ 0\,) \\ (\quad\ d_2 d_1 d_0) \end{array}$$

School Method for Multiplication

Pseudo-code:

$r = 0;$
for $(j = 0; j < n; j++)$
$\{$
$\quad r = r + a \cdot b_j \cdot B^j;$
$\}$

$p_j = a \cdot b_j$

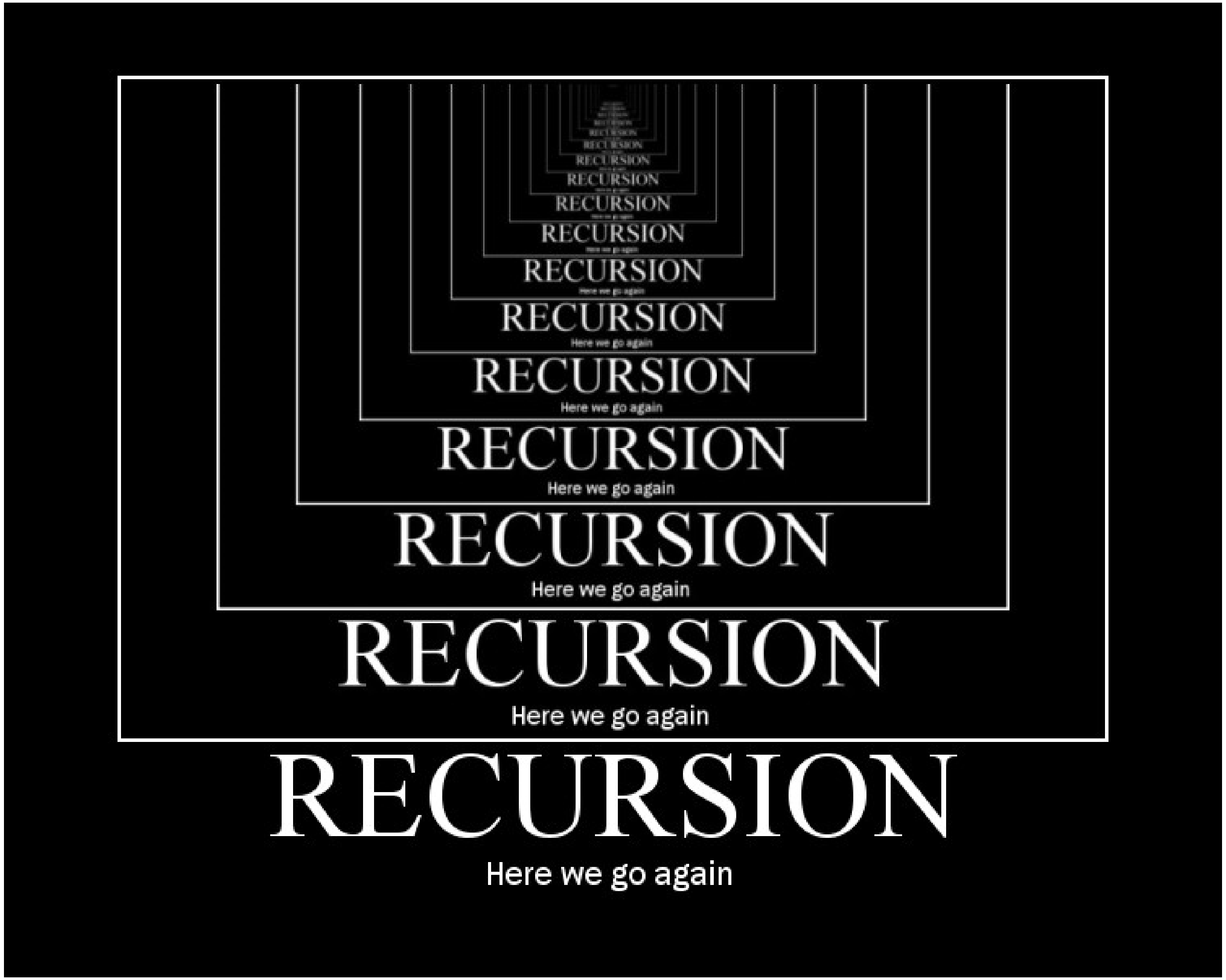
- $2n + 1$ operations for each p_j, $0 \leq j \leq n$,
  $\Longrightarrow 2n^2 + n$ operations.
- n summations of numbers having $(n + 1)$-digits that are nonzero
  $\Longrightarrow n^2 + n$ operations.
- In total $3n^2 + 2n$ operations

Theorem: The school method multiplies two n-digit integers with $3n^2 + 2n = \Theta(n^2)$ primitive operations.

Simple Integer Arithmetics: Summary

- Addition can be done using n primitive operations
- School method for multiplication needs $\theta(n^2)$ primitive operations.
- Question: Are there faster algorithms for multiplication?

Recursive approach



In order to understand recursion, you must first understand recursion.

Divide and Conquer Paradigm

For a given problem:

- Is it **small enough** to solve trivially?
  - If **YES**, solve it.
  - If **NOT**, break down a problem into sub-problems and see if these can be solved **directly**. If not, divide again.
- Solutions to the sub-problems are then **combined** to give a solution to the original problem.

Examples of problems:

- Sorting data (Quick Sort, Merge Sort)
- Integer multiplication (School method, Karatsuba algorithm)
- Closest pair of points problem

Recursive approach

Recursion – a method of defining a function in terms of its own definition.

Why write a method that calls itself?

- Recursion is a good problem solving approach.
- Recursive solutions are often shorter.
- Solve a problem by reducing the problem to smaller sub-problems; this results in recursive calls.

However…

- Good recursive solutions may be more difficult to design and test.
- Recursive calls can result in an infinite loop of calls

Recursive algorithm: Template

To solve a problem recursively

- break into smaller problems;
- solve sub-problems recursively;
- assemble sub-solutions.

```
recursive-algorithm(input) {
// base case
if (isSmallEnough(input))
   compute the solution and return it
else
// recursive case
   break input into simpler instances input1, input 2,...
   solution1 = recursive-algorithm(input1)
   solution2 = recursive-algorithm(input2)
   ...
   figure out solution to this problem from solution1, solution2,...
   return solution
}
```

Recursion versus iterative approach

- **Emphasis of iteration:**
  keep repeating until a task is “done”.

- **Emphasis of recursion:**
  break the problem up into smaller parts;
  combine the results.

Which is better?

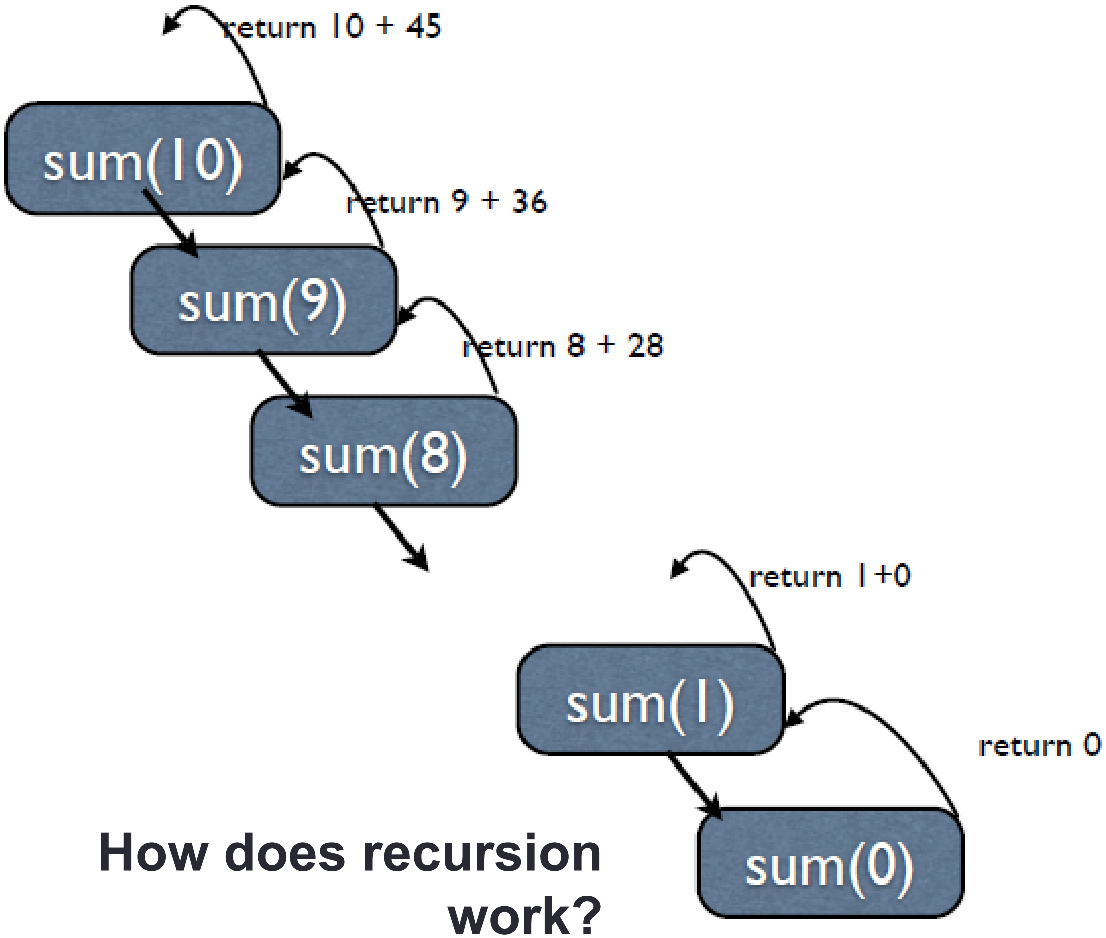
- Iterative solutions keep control local to loop, less “magical”
- Good recursive solutions may be more difficult to design and test.

Recursion versus iterative approach: Example

Write a function that computes the sum of numbers from 0 to n (i) using a loop; (ii) recursively.

```
// with a loop
int sum (int n) {
  int s = 0;
  for (int i=0; i<=n; i++)
       s+= i;
  return s;
}


// recursively
int sum (int n) {
  // base case
  if (n == 0) return 0;
  // else
  return n + sum(n-1);
}
```



How does recursion work?

Recursion versus iterative approach: Example

Recursive is not always better!

```
// Fibonacci: recursive version
int Fibonacci_R(int n) {
   if(n<=0) return 0;
   else if(n==1) return 1;
   else return Fibonacci_R(n-1)+Fibonacci_R(n-2);
}
```

This takes $O(2^n)$ steps! Impractical for large n.

```
// Fibonacci: iterative version
int Fibonacci_I(int n) {
   int fib[] = {0,1,1};
   for(int i=2; i<=n; i++) {
      fib[i%3] = fib[(i-1)%3] + fib[(i-2)%3];
   }
   return fib[n%3];
}
```

This iterative approach is “linear”; it takes $O(n)$ steps.

Recursion: Implementation issues

- Recursion is no different than a function call.
- The system keeps track of the sequence of method calls that have been started but not finished yet (active calls). **Order matters!**

Recursion pitfalls:

- **Missed base-case**: infinite recursion, stack overflow.
- **No convergence**: solve recursively a problem that is not simpler than the original one.
- Recursion has an ***overhead*** (keep track of all active frames). Modern compilers can often optimize the code and eliminate recursion.

Unless you write super-duper optimized code, recursion is good.

Merge Sort: Idea

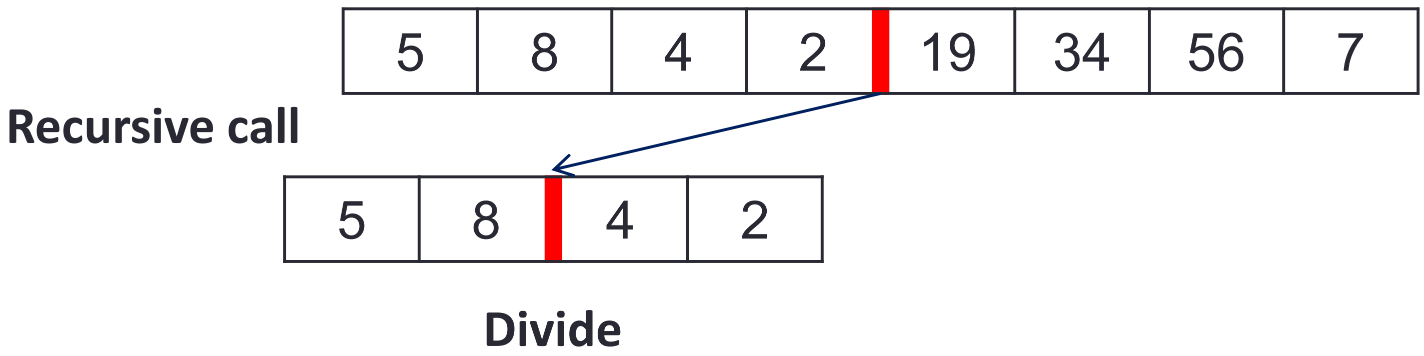
- **Divide:** Divide the unsorted list into two sub-lists of about half the size.
- **Conquer:** Sort each of the two sub-lists recursively until we have list sizes of length 1, in which case the list itself is returned.
- **Merge:** Merge the two sorted sub-lists back into one sorted list.

Merge Sort: Example

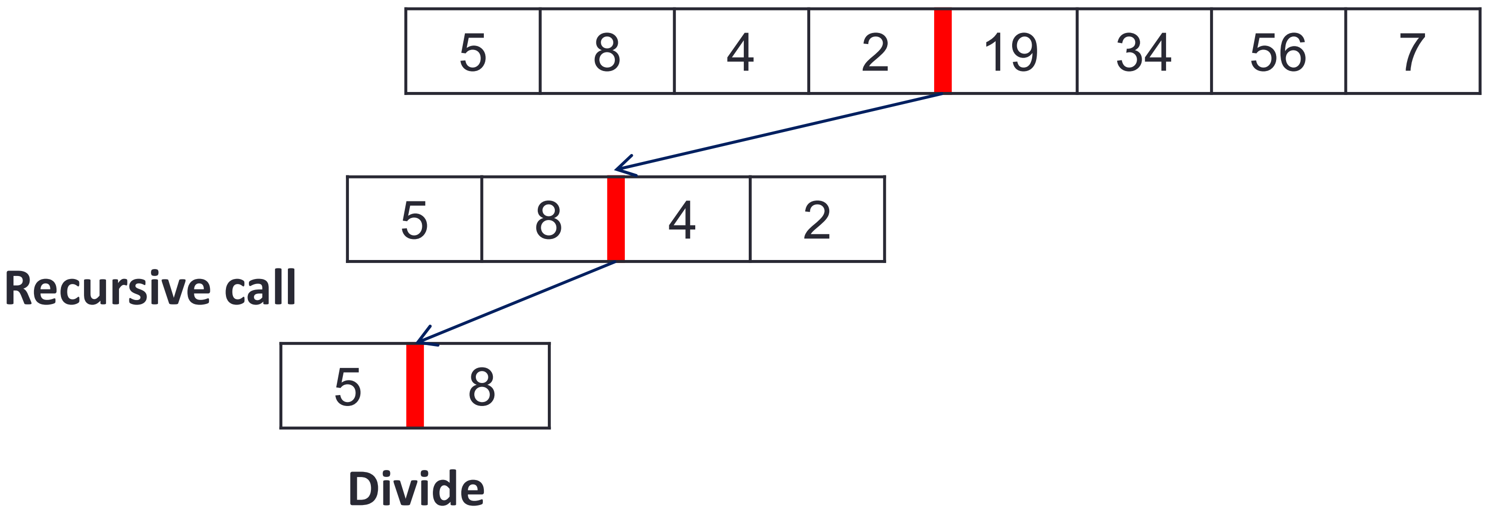
5	8	4	2	19	34	56	7

Divide

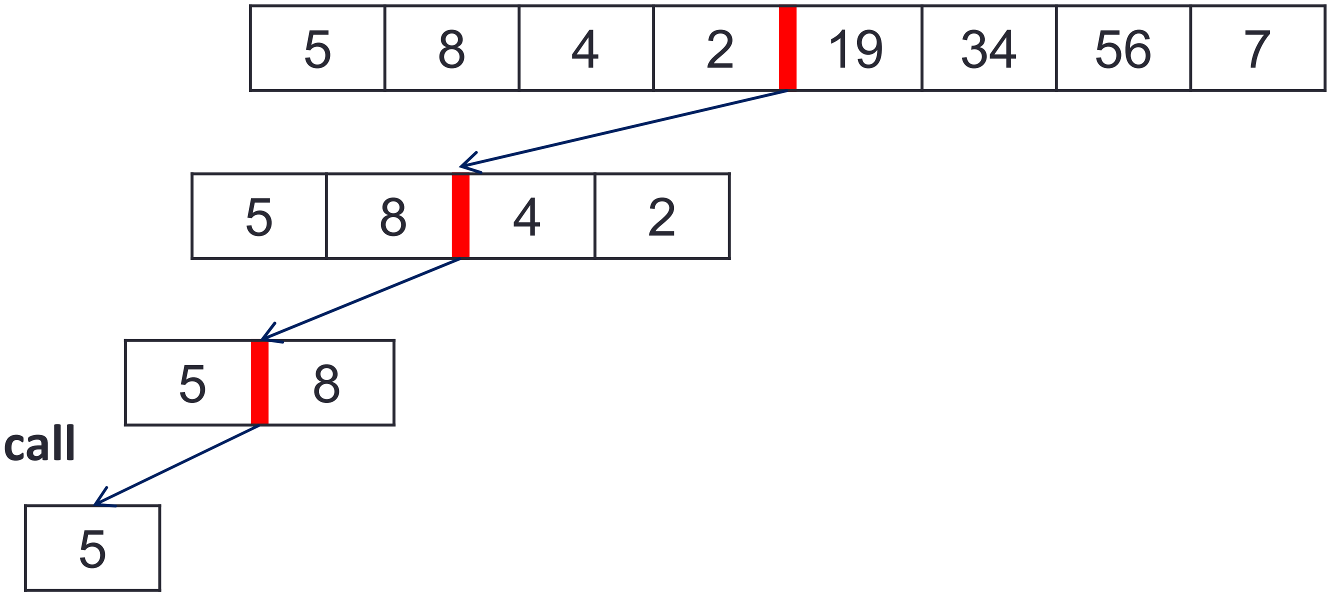
Merge Sort: Example



Merge Sort: Example



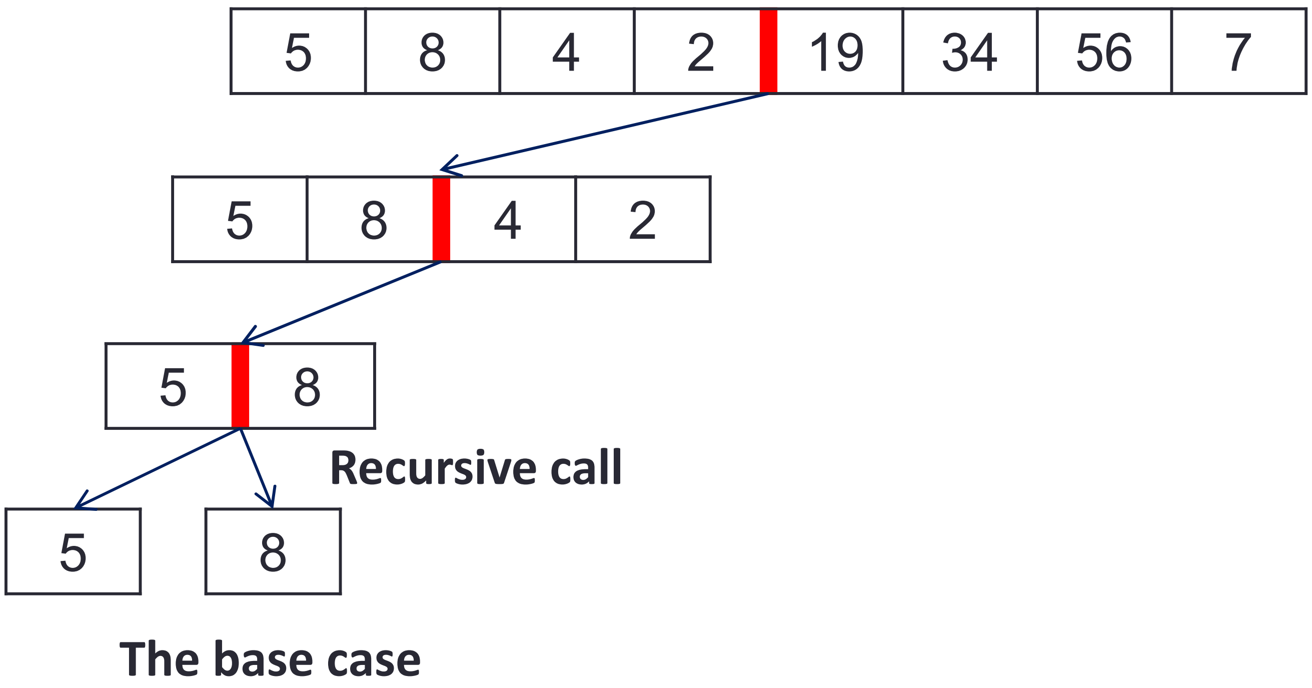
Merge Sort: Example



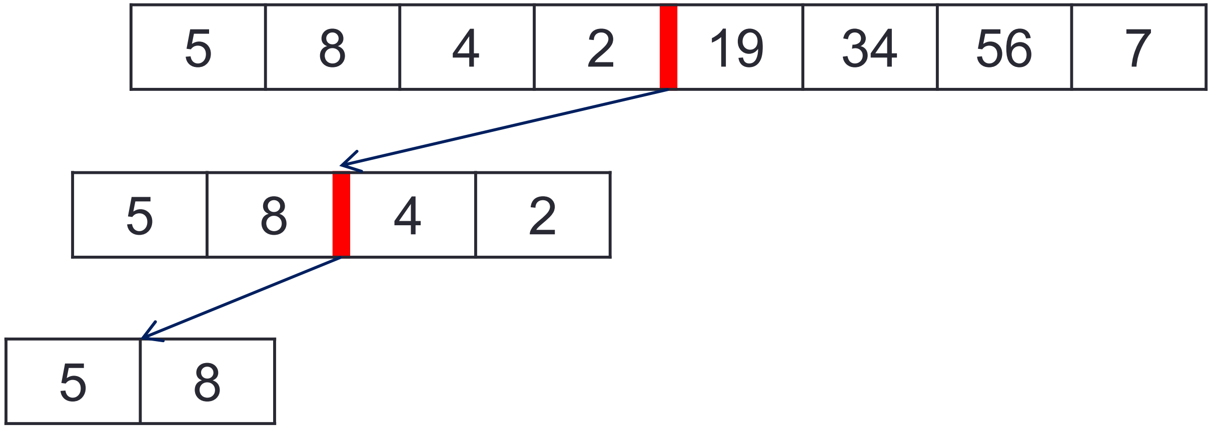
Recursive call

The base case

Merge Sort: Example

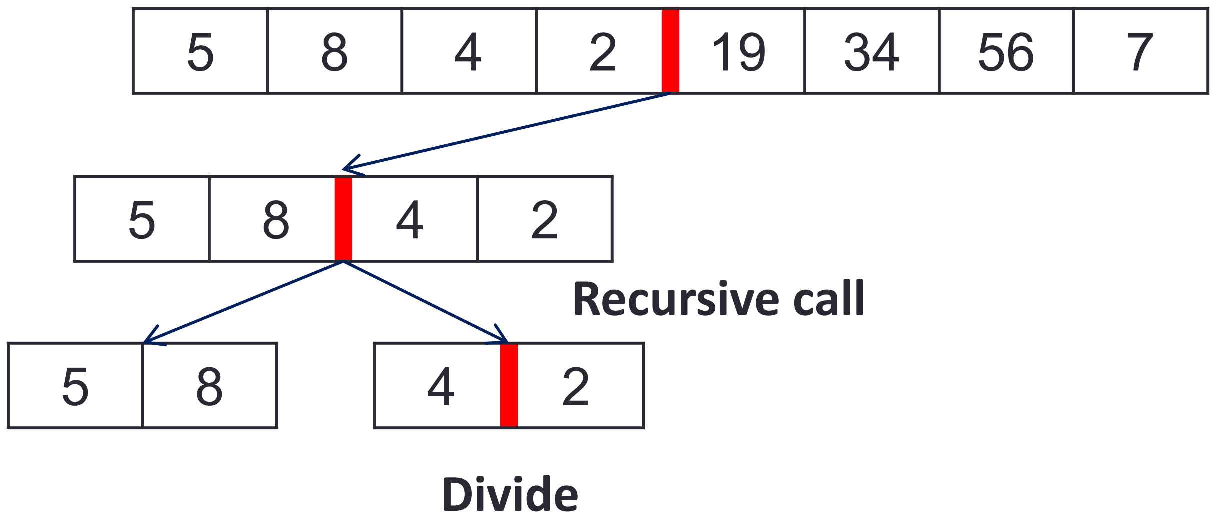


Merge Sort: Example

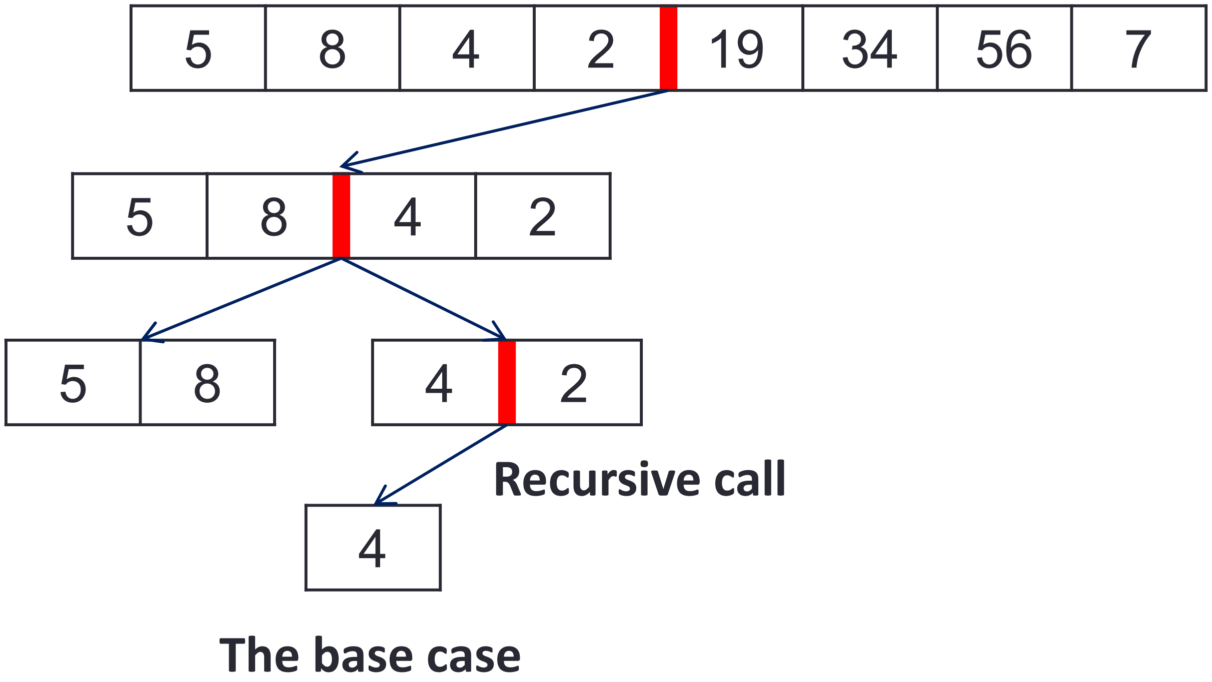


Return from recursive calls.
Merge (sort) the sub-lists

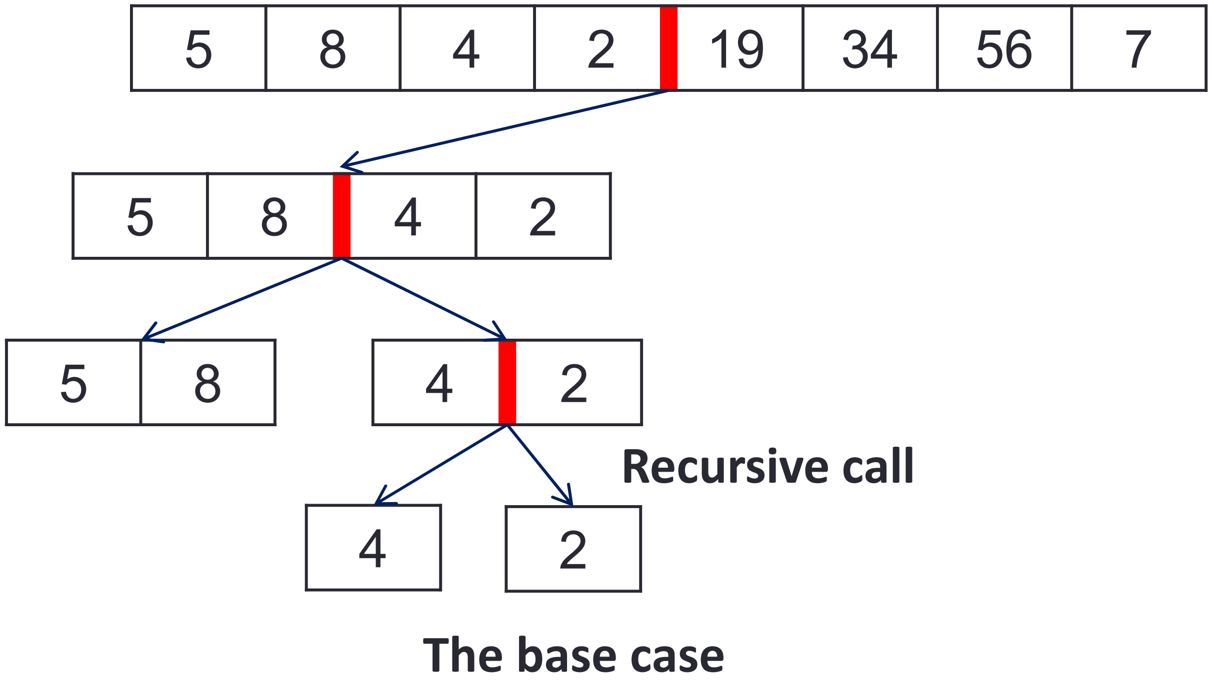
Merge Sort: Example



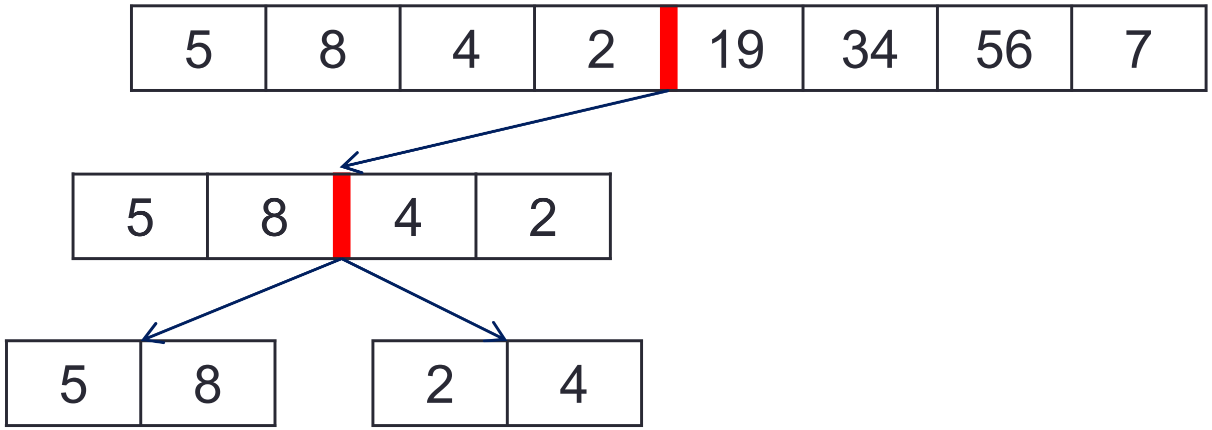
Merge Sort: Example



Merge Sort: Example

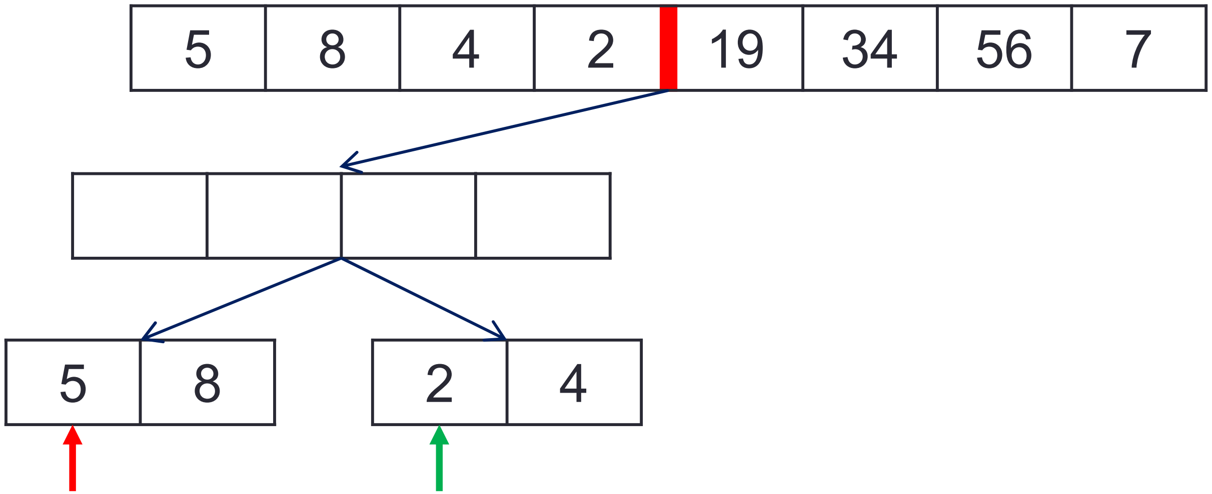


Merge Sort: Example



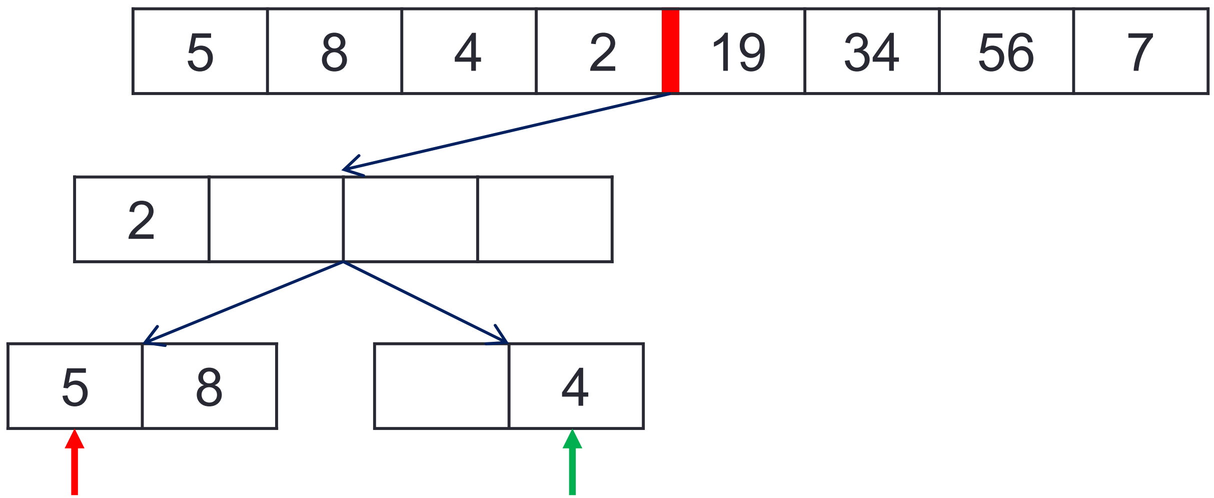
Return from recursive calls.
Merge (sort) the sub-lists

Merge Sort: Example

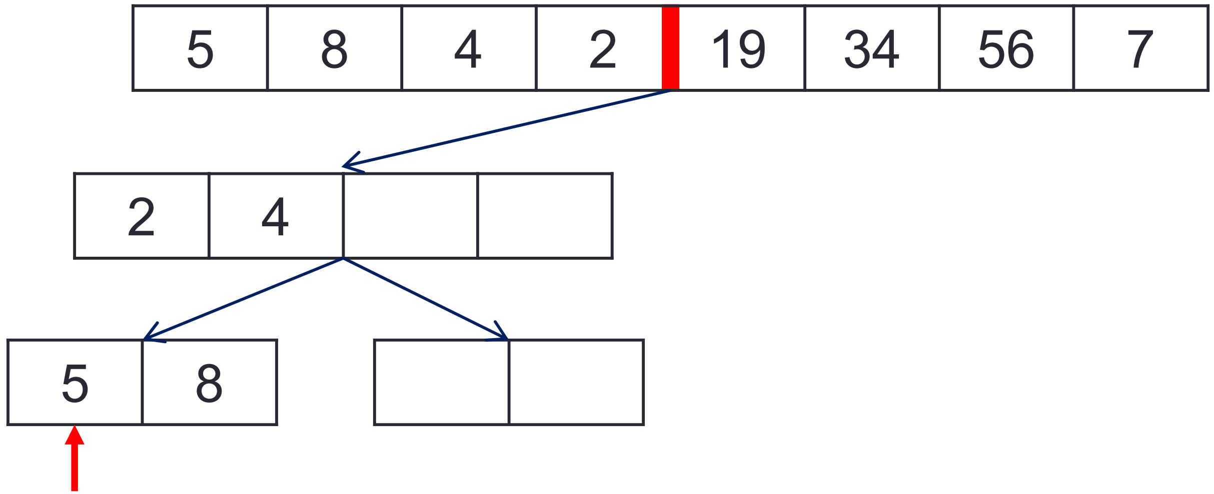


Return from recursive calls.
Merge (sort) the sub-lists

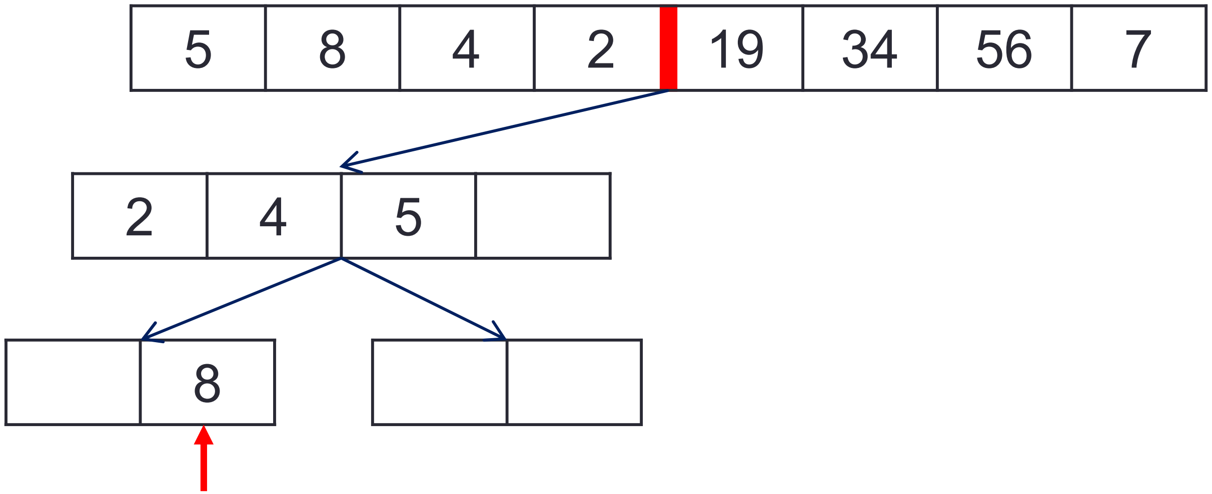
Merge Sort: Example



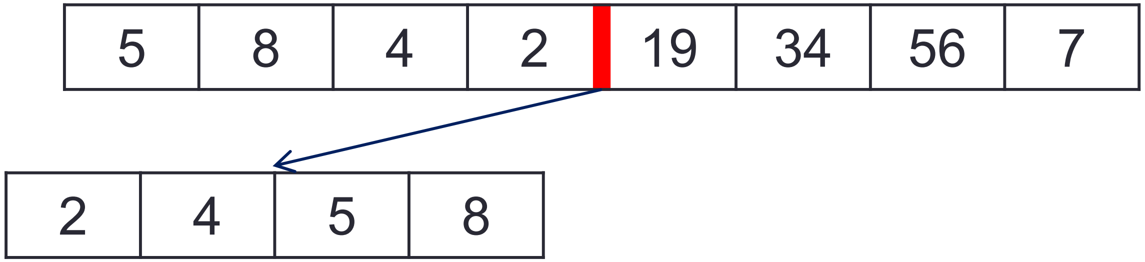
Merge Sort: Example



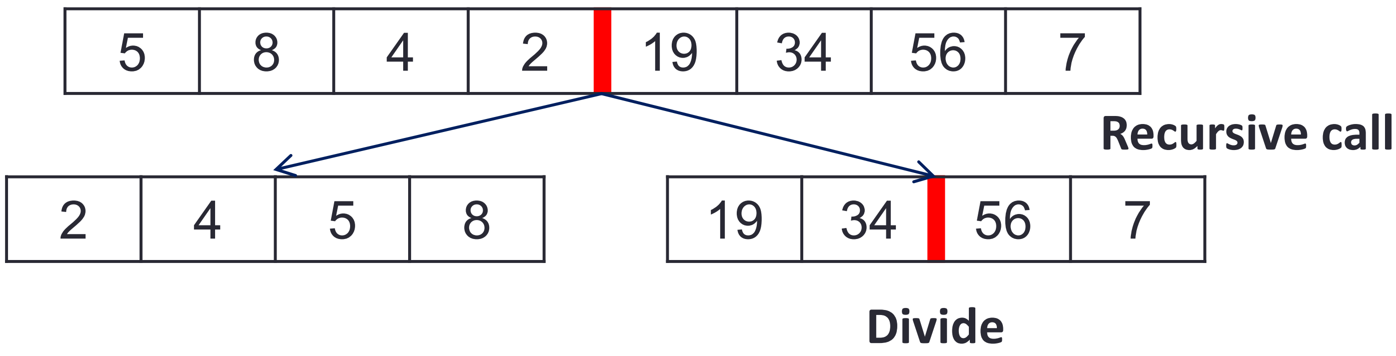
Merge Sort: Example



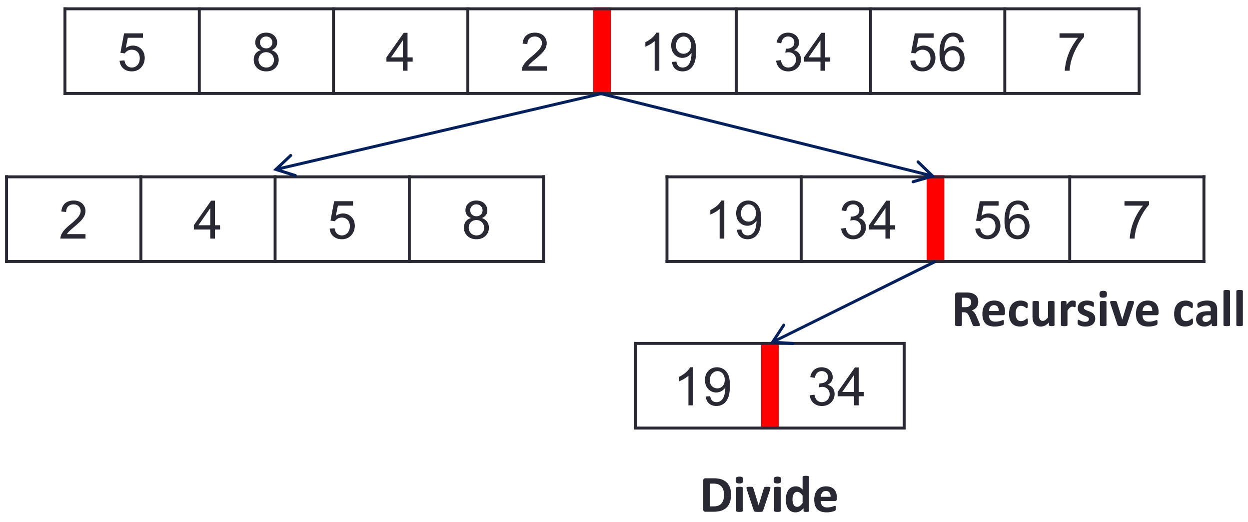
Merge Sort: Example



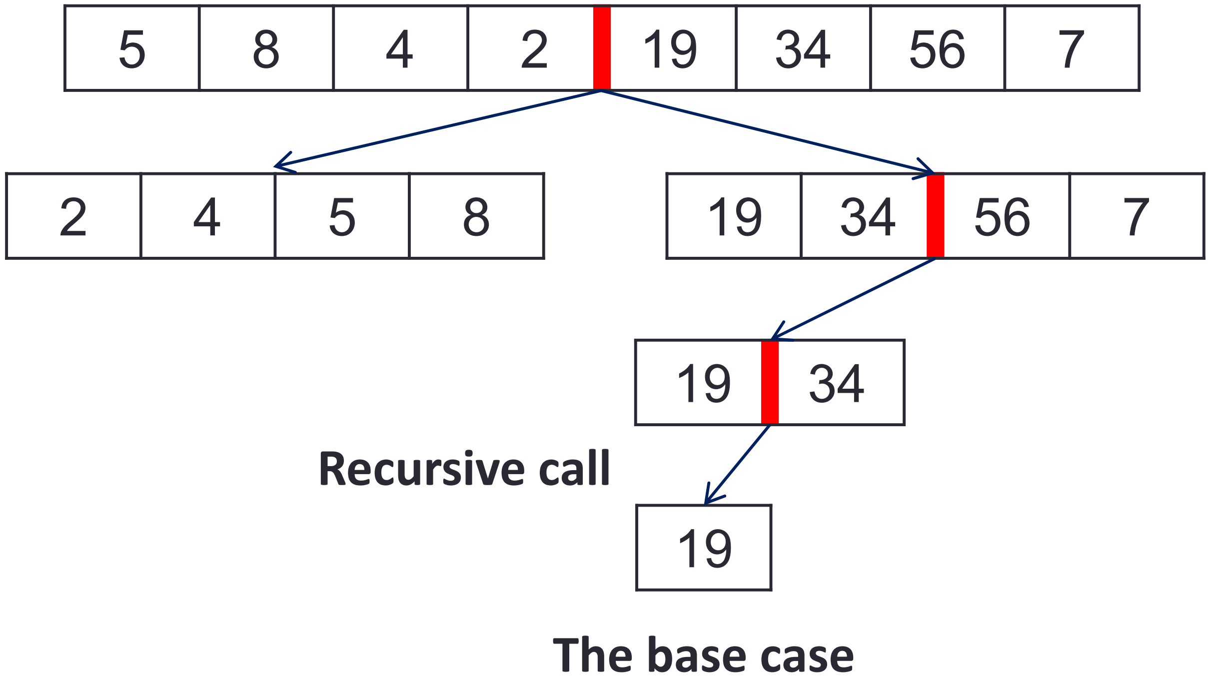
Merge Sort: Example



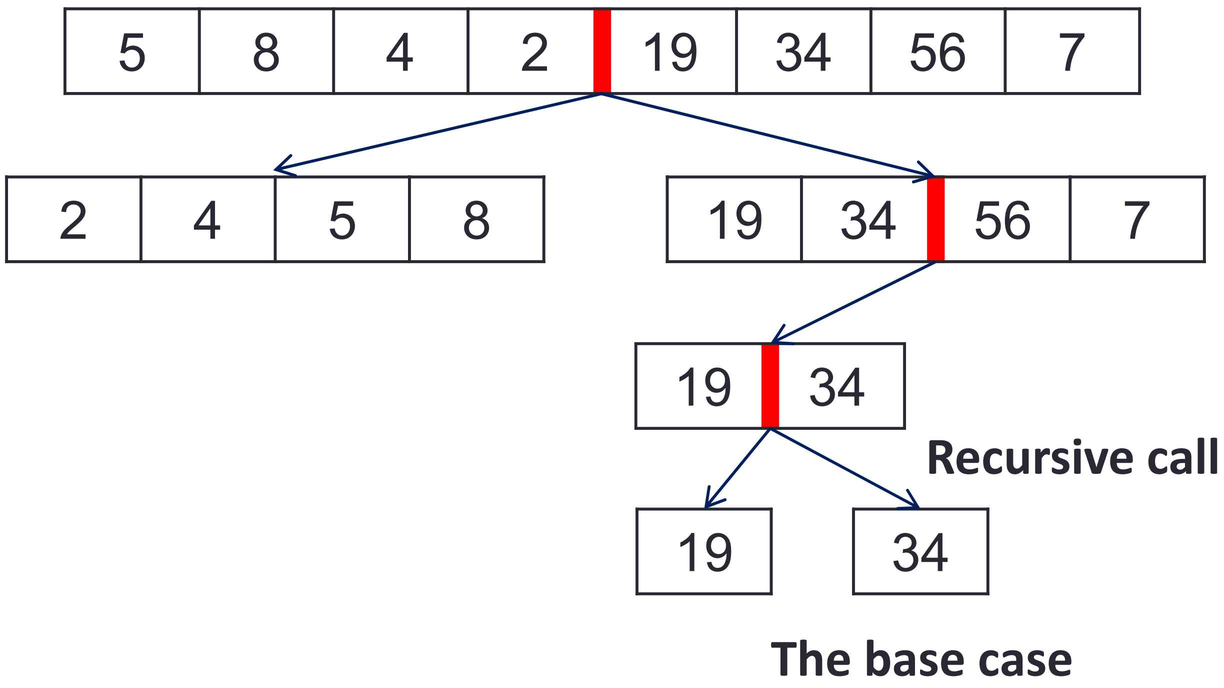
Merge Sort: Example



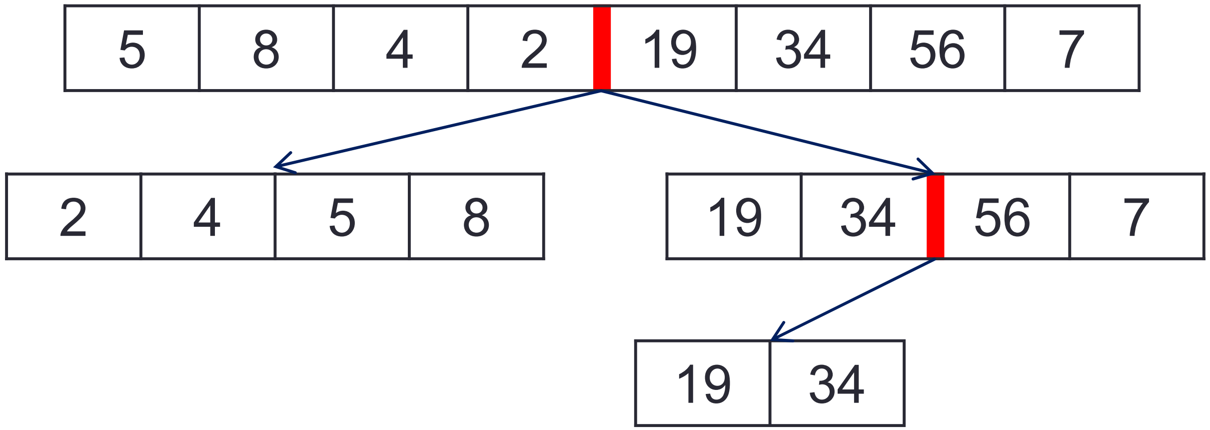
Merge Sort: Example



Merge Sort: Example

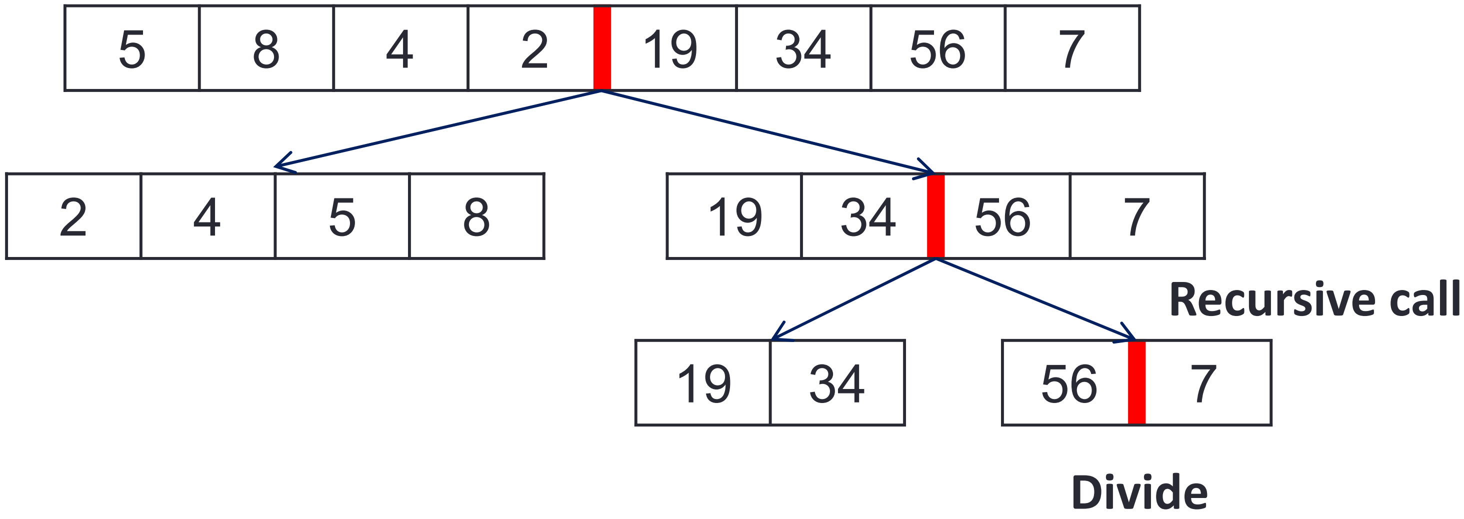


Merge Sort: Example

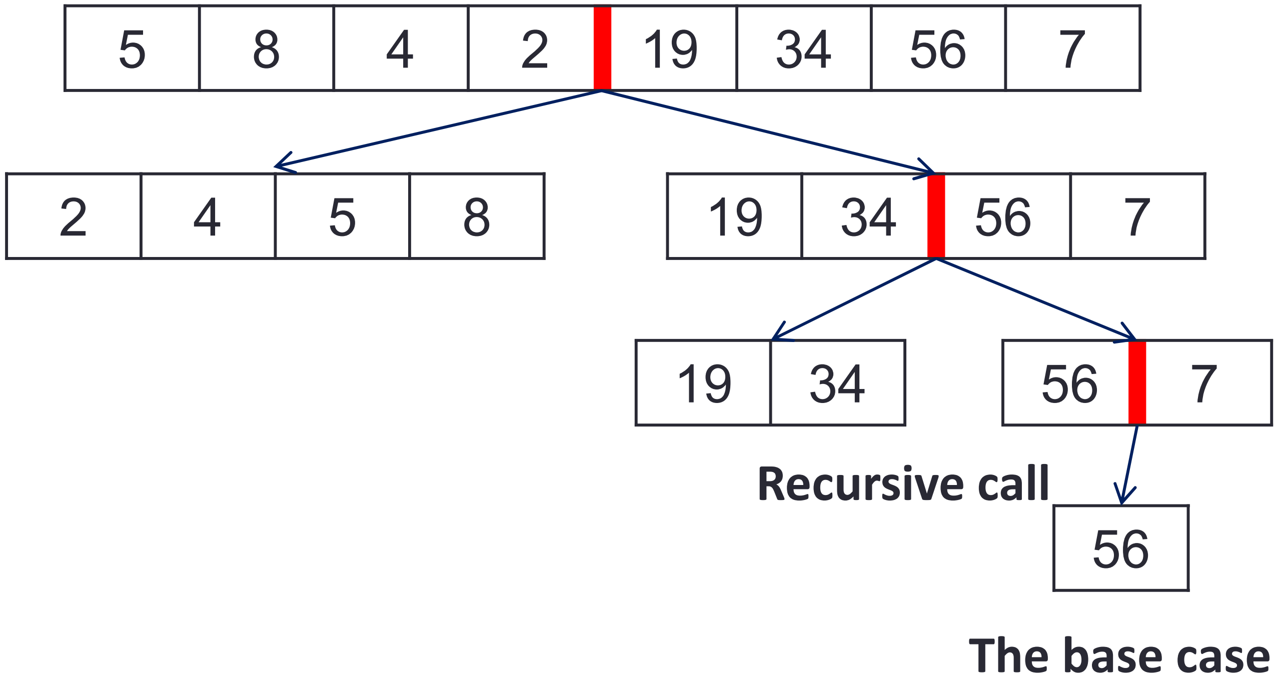


Return from recursive calls.
Merge (sort) the sub-lists

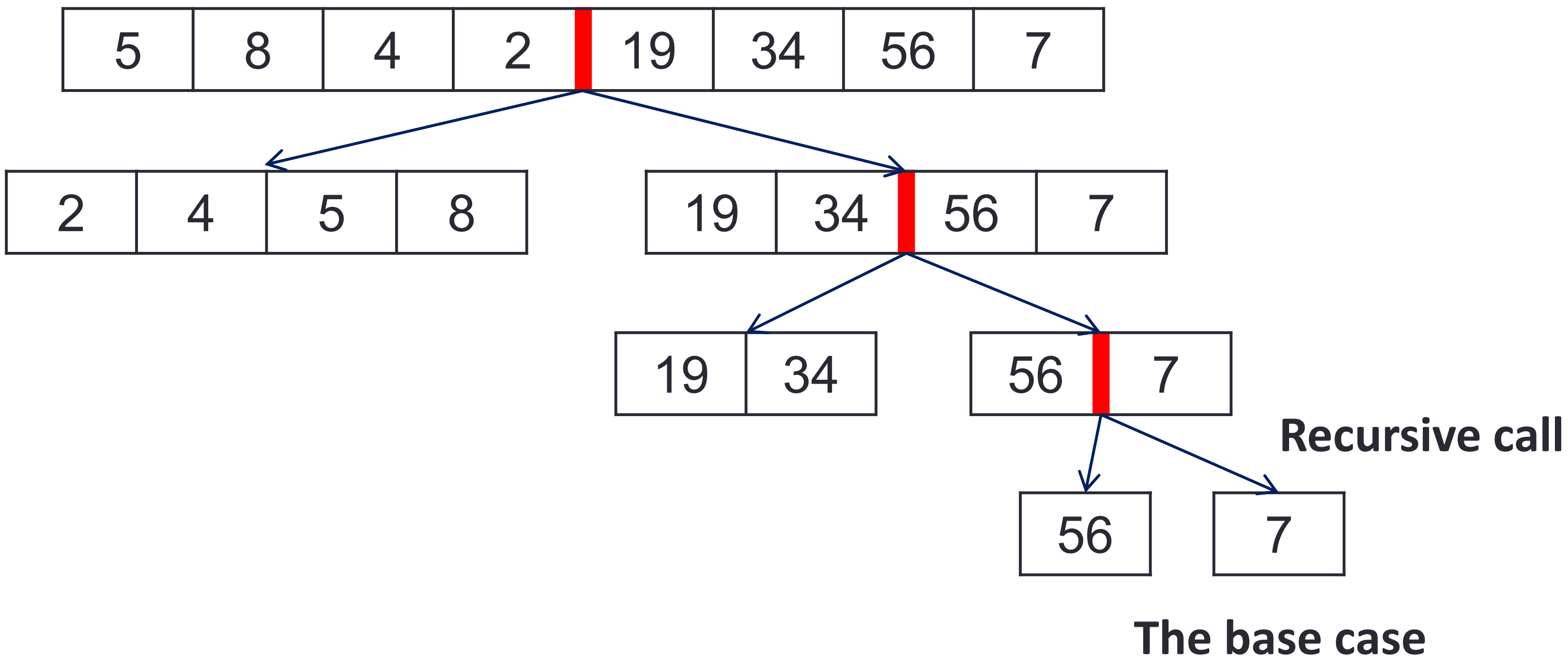
Merge Sort: Example



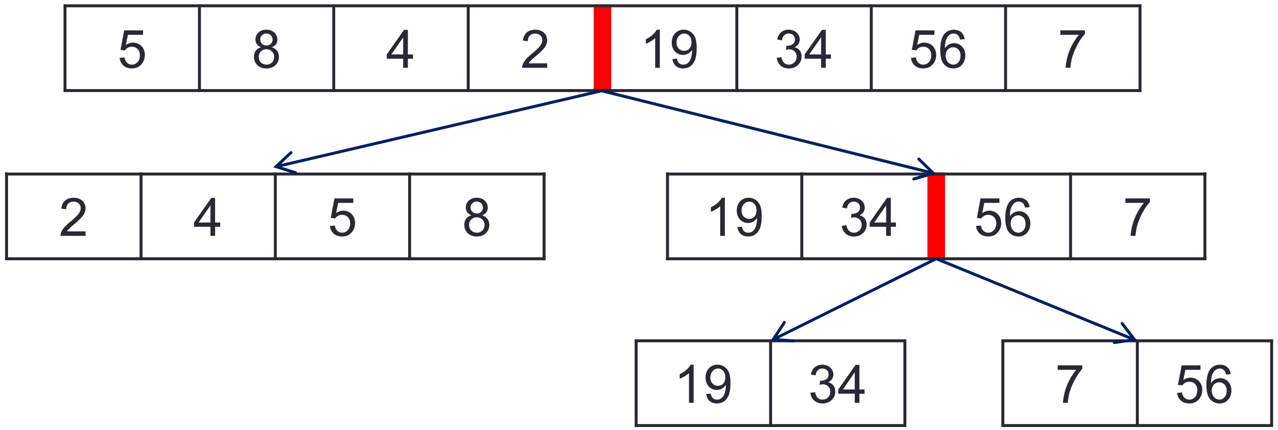
Merge Sort: Example



Merge Sort: Example

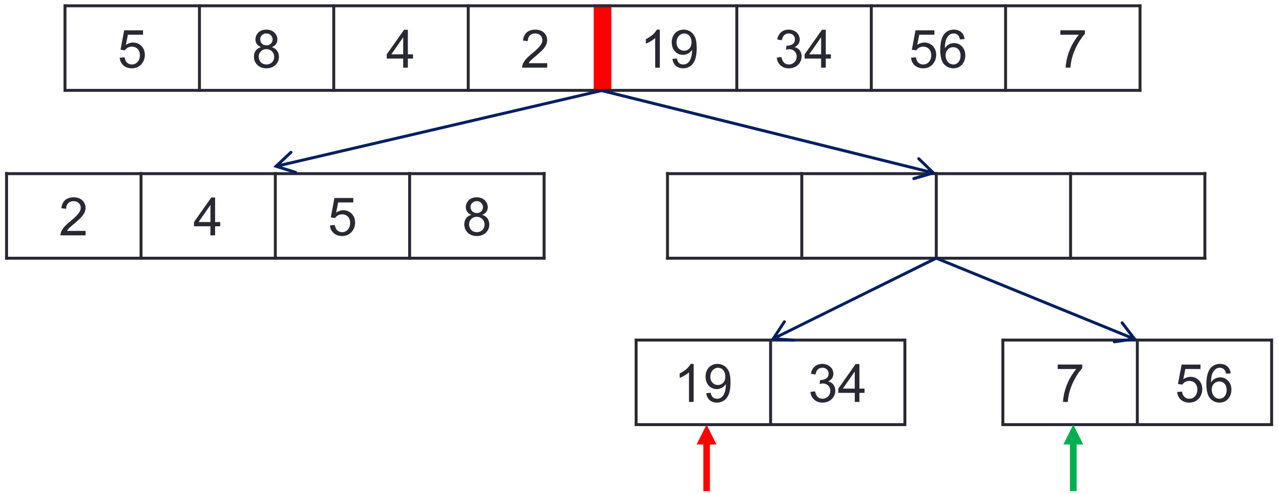


Merge Sort: Example

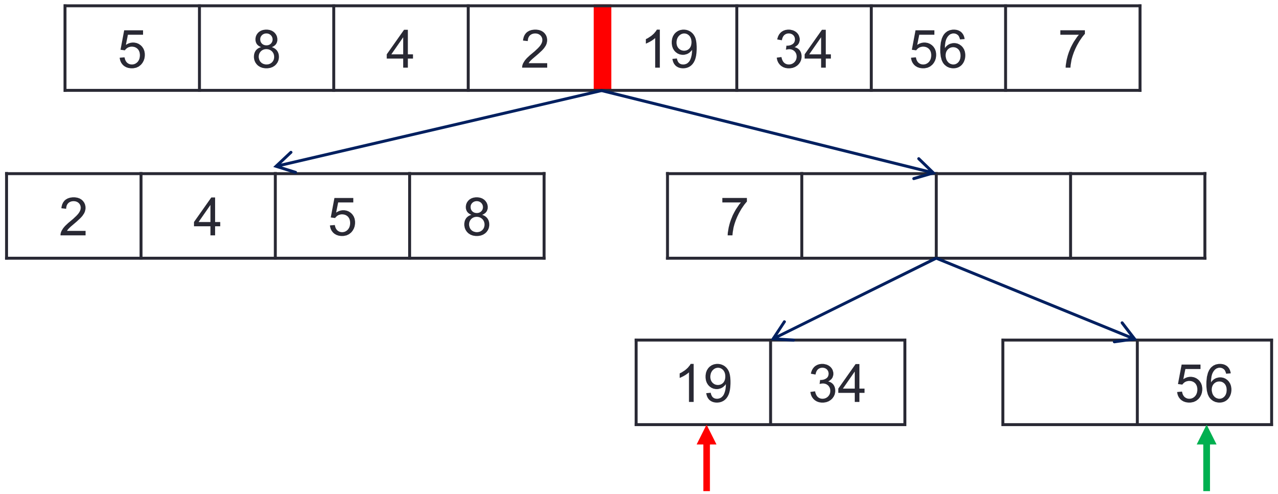


Return from recursive calls.
Merge (sort) the sub-lists

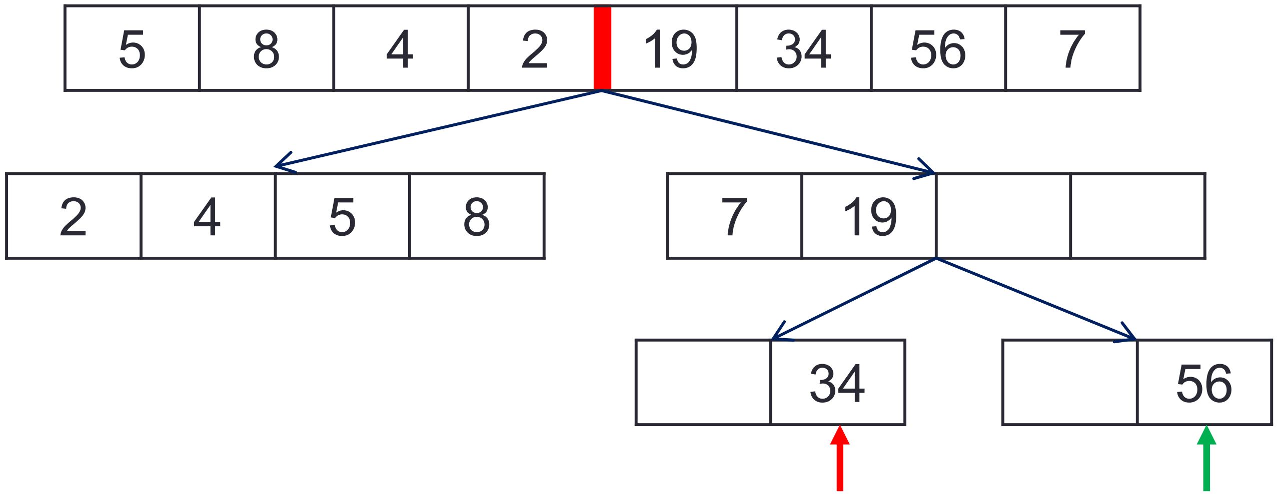
Merge Sort: Example



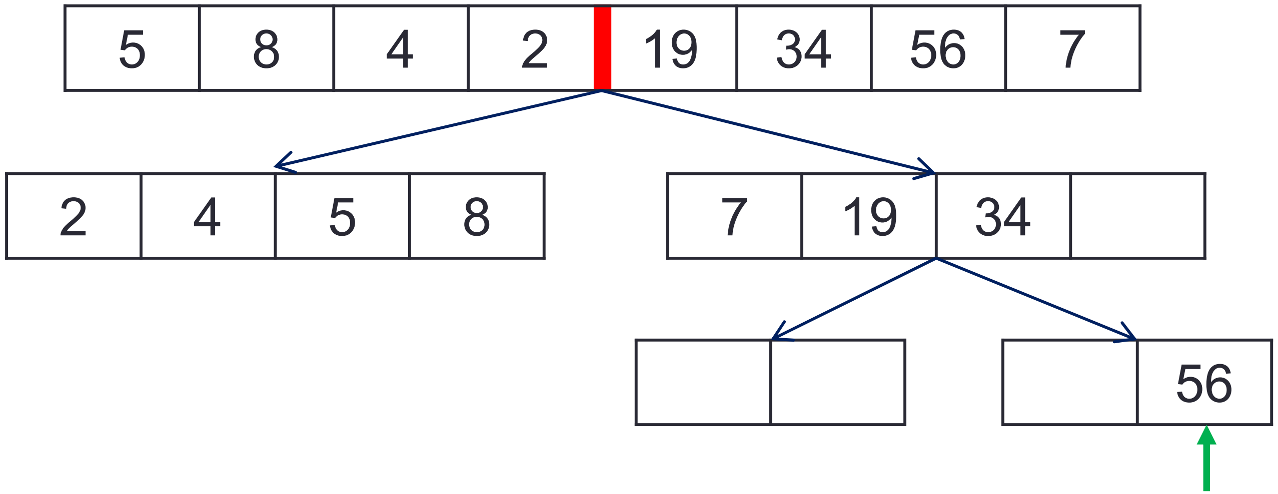
Merge Sort: Example



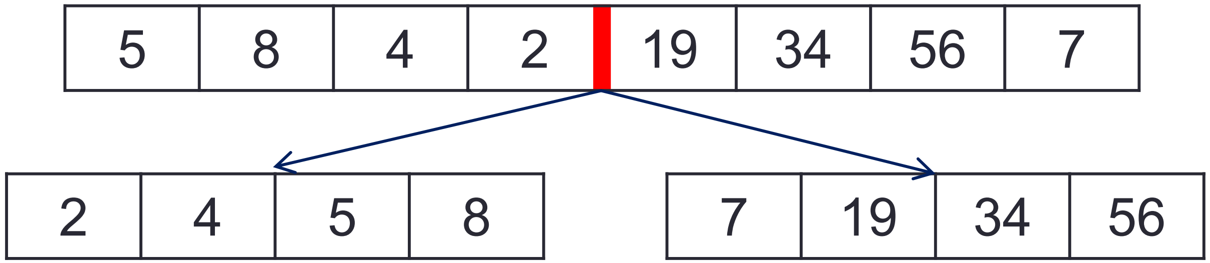
Merge Sort: Example



Merge Sort: Example

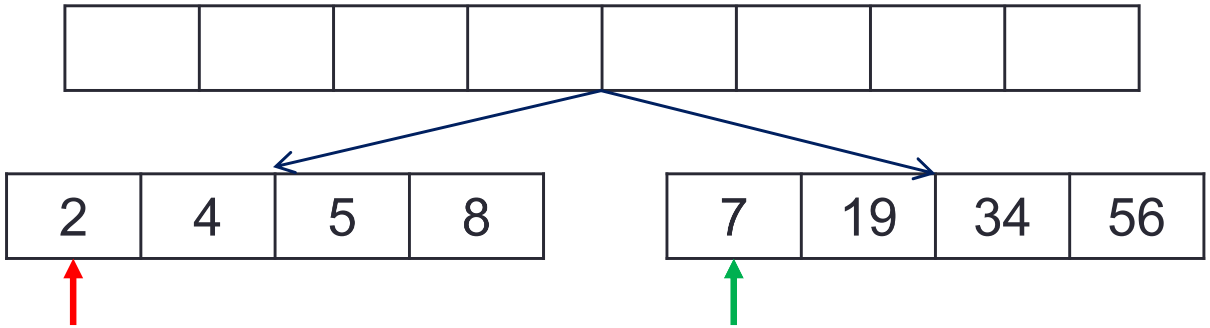


Merge Sort: Example



Return from recursive calls.
Merge (sort) the sub-lists

Merge Sort: Example

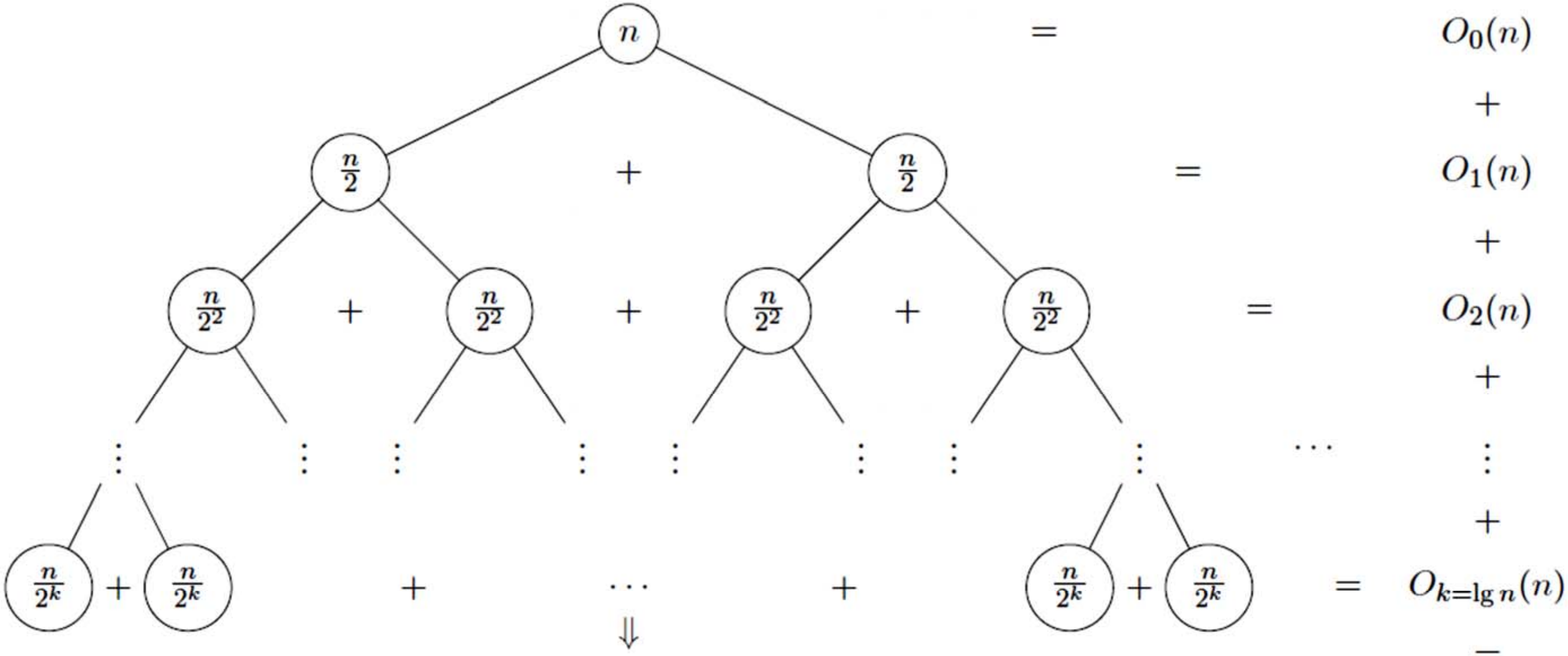


Proceed with merging the two sub-lists to obtain the final solution

Merge Sort: Example

2	4	5	8	7	19	34	56

Merge Sort: Complexity



$$O\left(\sum_{i=0}^{k} 2^i \cdot \frac{n}{2^i}\right) = O\left(\sum_{i=0}^{k} n\right) = O(k \cdot n) \Leftrightarrow O(n \cdot \lg n)$$

Every level i has 2^i sub-lists, k levels in total.

Every sub-list has $\frac{n}{2^i}$ elements to compare

Merge Sort: Complexity and recurrence relation

The running time of merge sort for a list of length n is

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + n, & \text{if } n > 1 \end{cases}$$

apply the algorithm to two lists of half the size of the original list

and add the n steps taken to merge the resulting two lists

Merge Sort: Properties and complexity

- Provides **stable sort**, which means that the implementation preserves the input order of equal elements in the sorted output.
- Consistent speed in all type of data sets. Good performance for huge inputs.
- Most common implementation does not sort in place; therefore, requires additional memory space to store the auxiliary arrays.

- Worst case: T(n) = $O(n\log n)$ comparisons
- Best case: T(n) = $O(n\log n)$ comparisons
- Average case: T(n) = $O(n\log n)$ comparisons
- Worst-case space complexity $O(n)$ auxiliary

Quick Sort: Idea

Quick Sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:

- **Divide**: pick a random element x (called **pivot**) and partition array A into
  - L elements less than x
  - E elements equal x
  - G elements greater than x
- **Recur**: sort L and G
- **Conquer**: join L, E and G

Quick Sort: Example

Pivot point selection

7 2 9 4 3 7 **6** 1

Quick Sort: Example

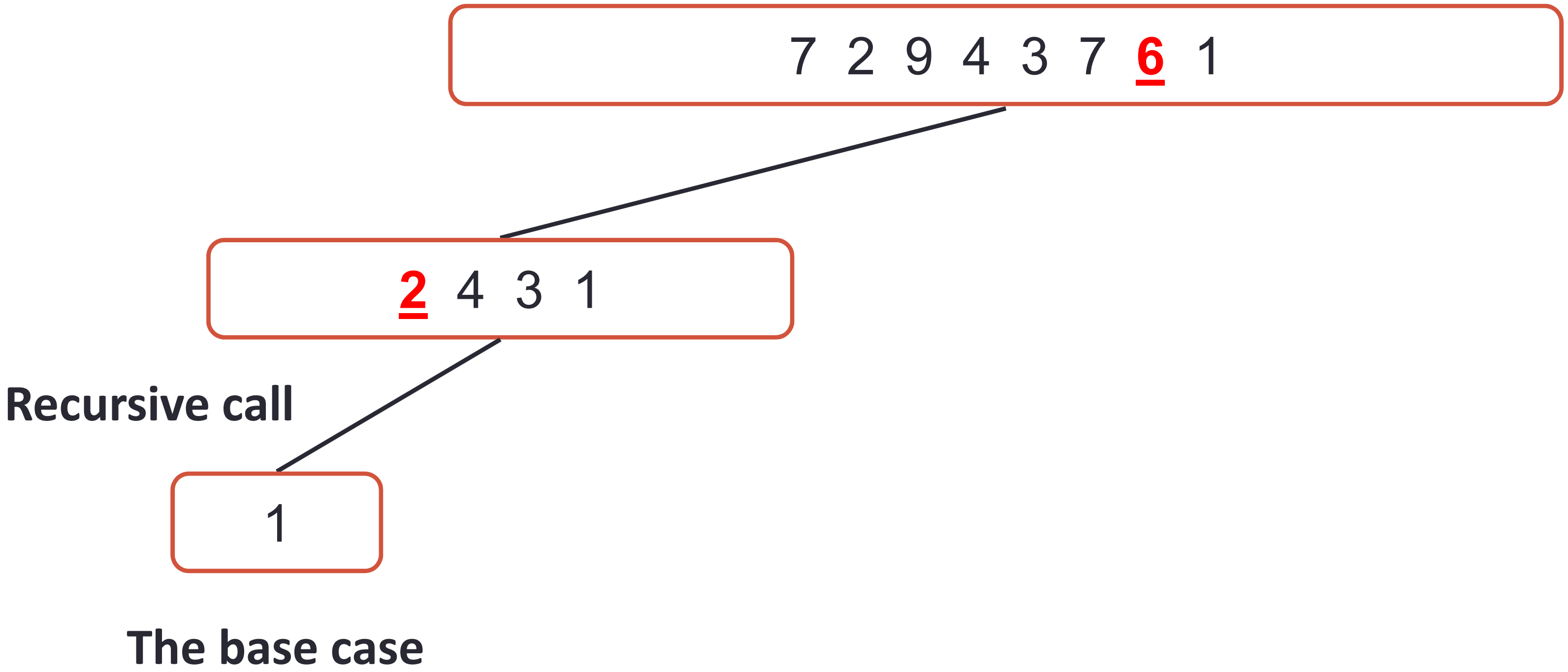
7 2 9 4 3 7 **6** 1

Recursive call

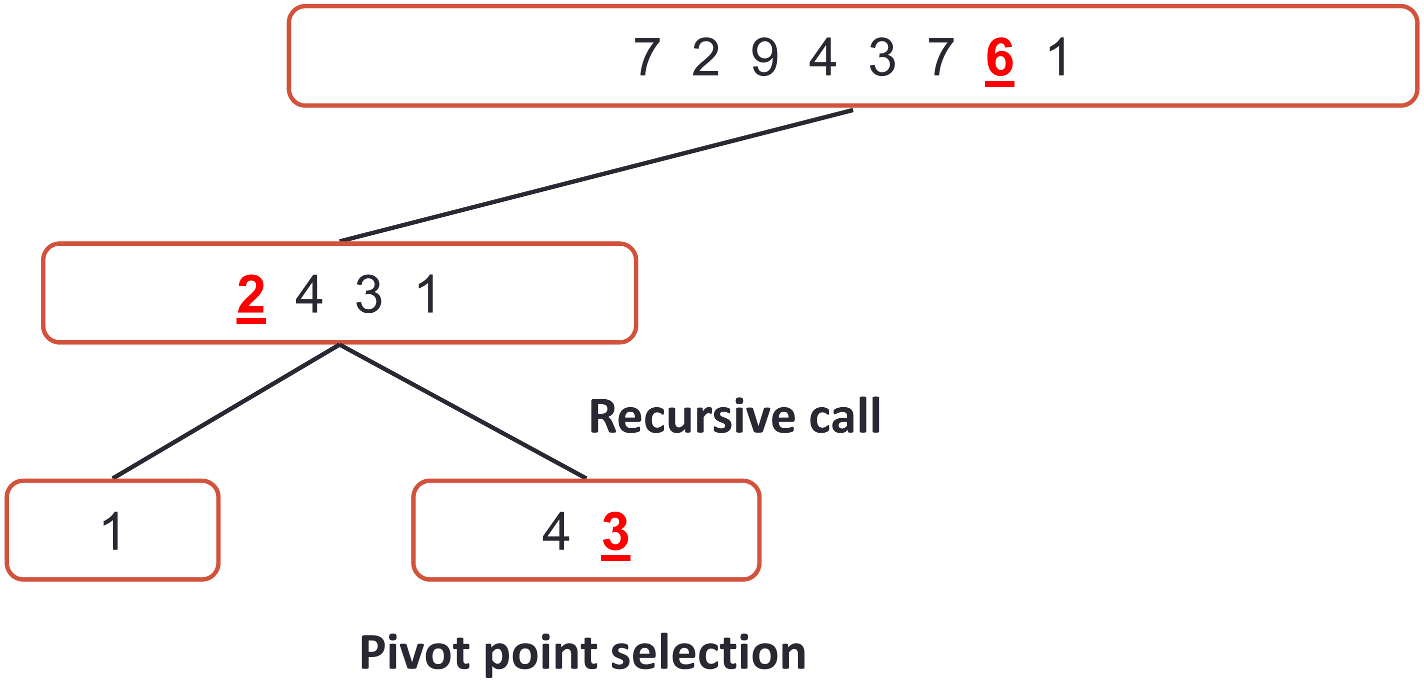
2 4 3 1

Pivot point selection

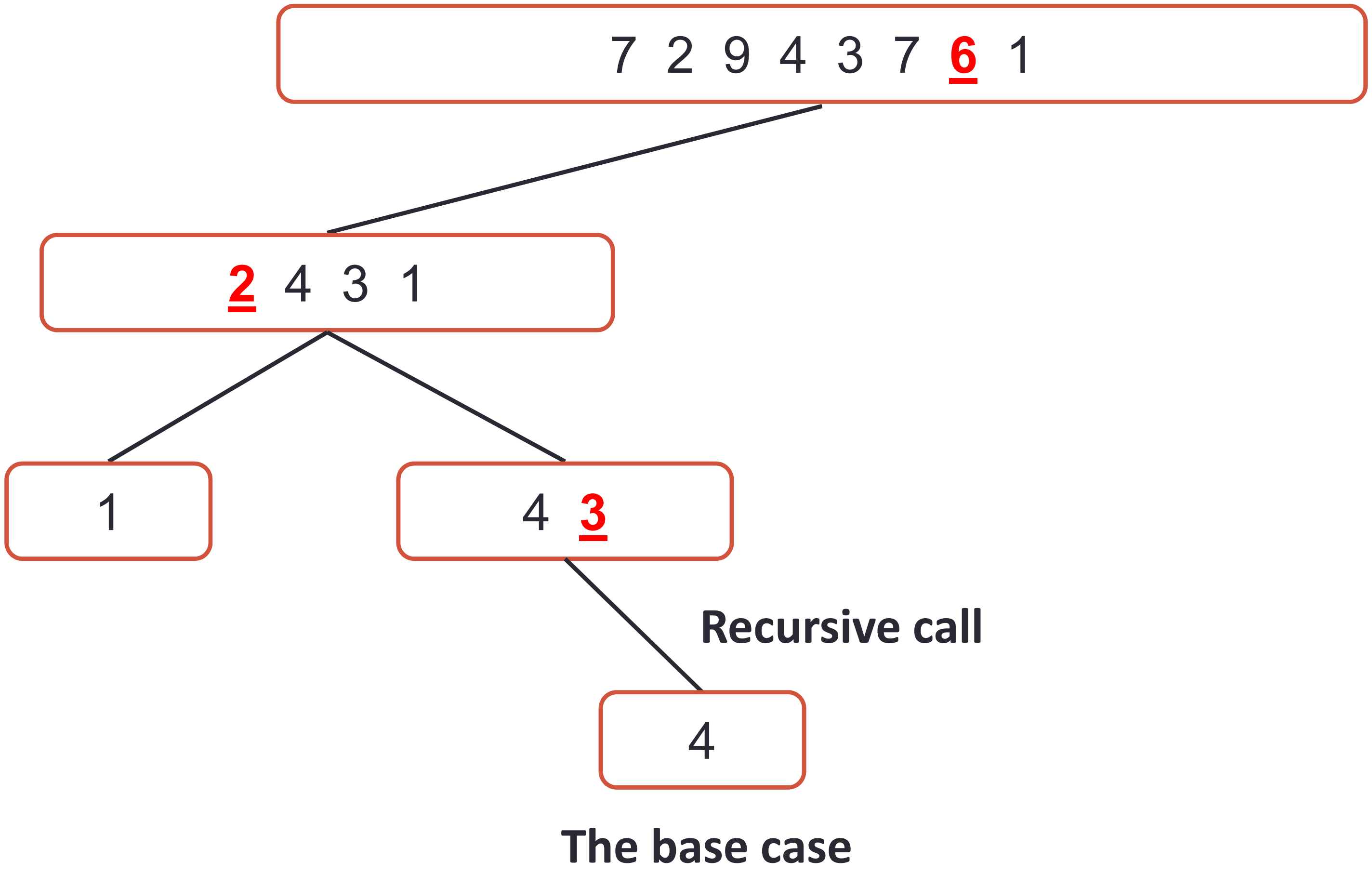
Quick Sort: Example



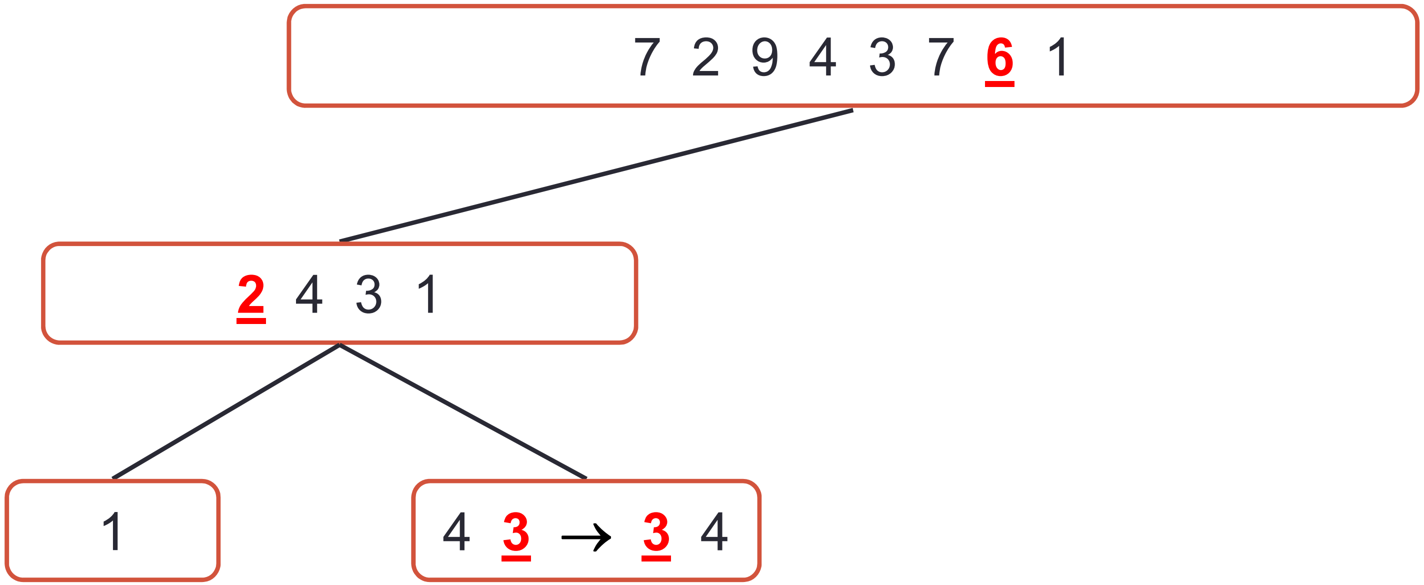
Quick Sort: Example



Quick Sort: Example



Quick Sort: Example



Return from recursive calls.
Combine the sub-lists

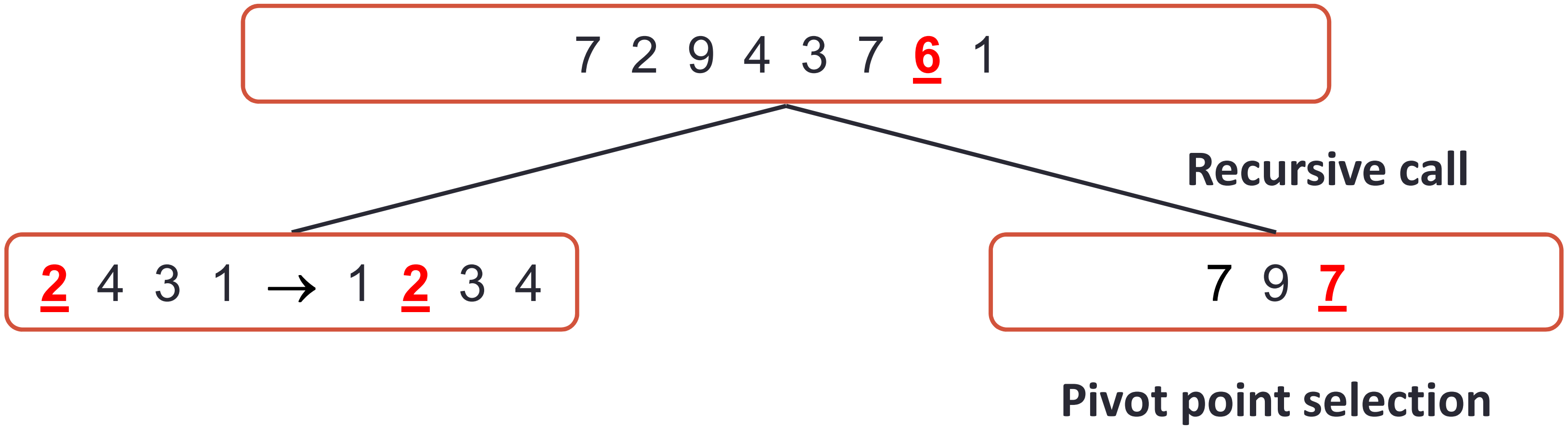
Quick Sort: Example

7 2 9 4 3 7 **6** 1

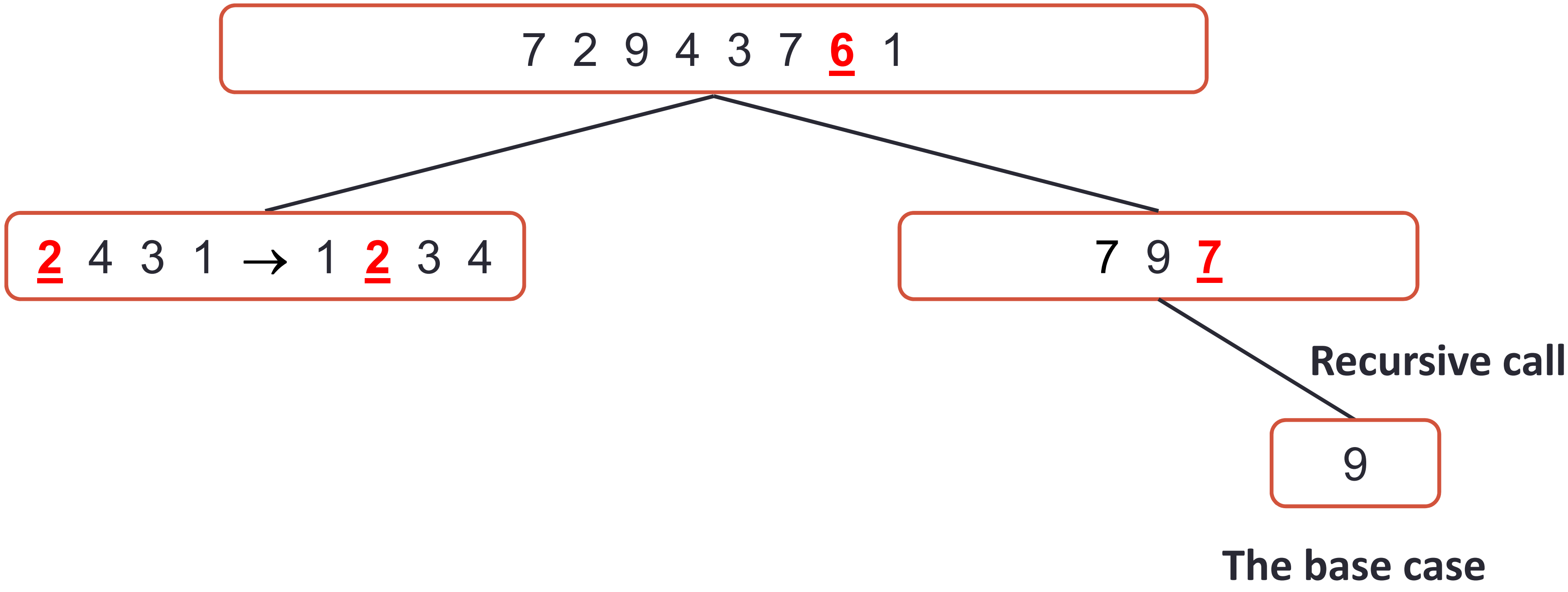
2 4 3 1 → 1 **2** 3 4

Return from recursive calls.
Combine the sub-lists

Quick Sort: Example



Quick Sort: Example



Quick Sort: Example

7 2 9 4 3 7 **6** 1

2 4 3 1 → 1 **2** 3 4

7 9 **7** → 7 **7** 9

Return from recursive calls.
Combine the sub-lists

Quick Sort: Example

7 2 9 4 3 7 <u>**6**</u> 1 → 1 2 3 4 <u>**6**</u> 7 7 9

Return from recursive calls.
Combine the sub-lists

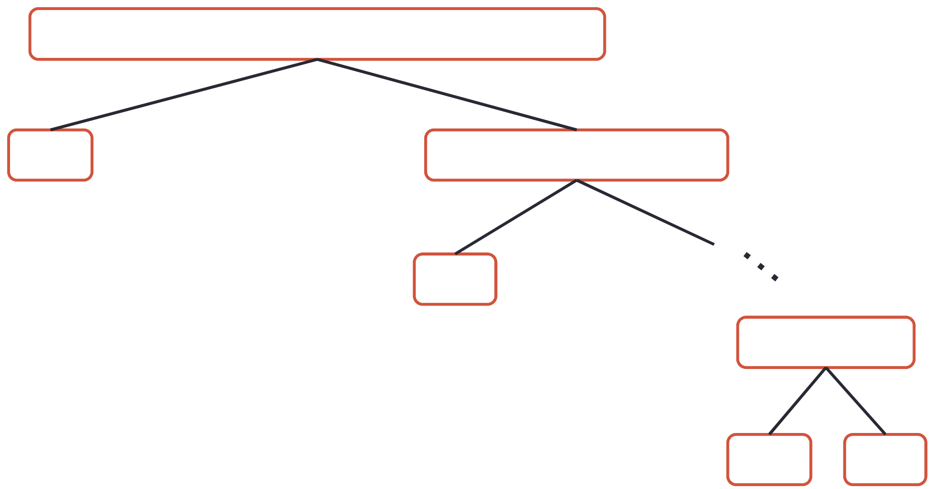
Quick Sort: Complexity of partition step

- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x.
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time.
- Thus, the partition step of Quick Sort takes $O(n)$ time.

Quick Sort: Worst case complexity

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element.
- One of L and G has size $n-1$ and the other has size L.
- The running time is proportional to the sum
$$n + (n-1) + \cdots + 2 + 1$$
- Thus, the worst-case running time of Quick Sort is $O(n^2)$.

depth	time
0	n
1	$n-1$
…	…
$n-1$	1



Quick Sort: Properties and complexity

- Not stable because of long distance swapping.
- High probability of choosing the right pivot
  $O(n^2)$ runtime complexity, but typically $O(n\log n)$ time.
- Requires little space and exhibits good cache locality
  Can run **in-place** using only $O(\log n)$ additional storage space to perform sorting.

• Worst case:	$\mathrm{T(n)} = O(n^2)$ comparisons
• Best case:	$\mathrm{T(n)} = O(n\log n)$ comparisons
• Average case:	$\mathrm{T(n)} = O(n\log n)$ comparisons
• Worst-case space complexity	$O(n)$ auxiliary $O(\log n)$ auxiliary using bounded recursion

Sorting algorithm in C#

List<T>.Sort Method

This method uses the **Introspective Sort** (Introsort) algorithm for an array of n elements as follows:

- If the partition size is fewer than 16 elements, it uses Insertion Sort.
- If the number of partitions exceeds $2\log n$, it uses a Heapsort algorithm.
- Otherwise, it uses a QuickSort algorithm.

- This implementation performs an unstable sort; that is, if two elements are equal, their order might not be preserved.
- On average, this method is an $O(n\log n)$ operation; in the worst case it is an $O(n^2)$ operation.

Other references and things to do

- Have a look at attached references in CloudDeakin.
- Read chapters 5.1.1, 5.2-5.4, 5.6, 12.1, and 12.2 in Data Structures and Algorithms in Java. Michael T. Goodrich, Irvine Roberto Tamassia, and Michael H. Goldwasser, 2014.