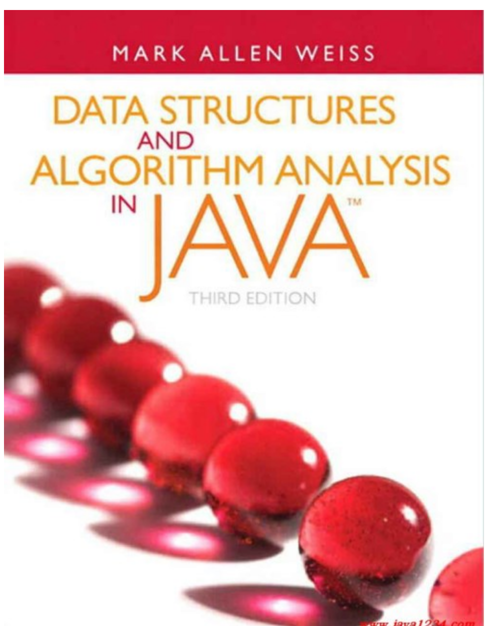
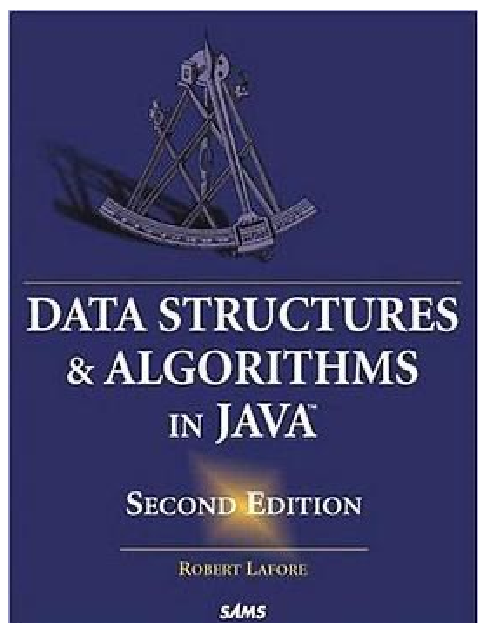
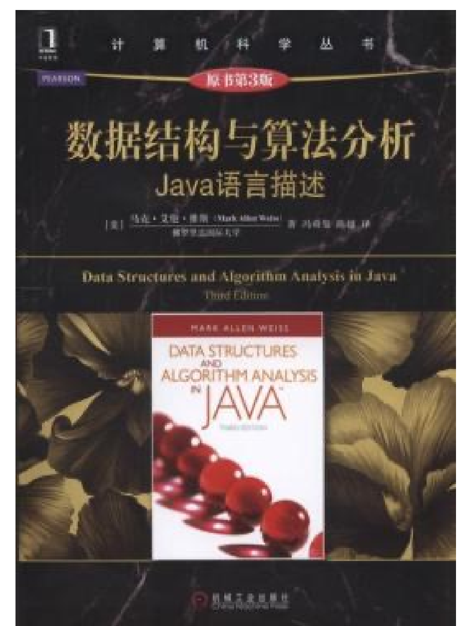
Topic 9 – Recursion







Topics

- Introduction
- Programming Revision
- Methods and Objects
- Arrays and Array Algorithms
- Big O Notation
- Sorting Algorithms
- Stacks and Queues
- Linked Lists
- **Recursion**
- Bit Manipulation

Content

- **Introduction**
- Towers of Hanoi
- Palindromes
- Raising to a power
- Mergesort

Recursive methods

- When a method is called the program goes off and runs it
- Consider the following method…

```
public static void sayHello() {
    System.out.println(“Hello World!”);
    sayHello();
}
```

What will happen?

Recursive methods

- As you'd expect,

  the main method calls sayHello(), which prints out "Hello World!" and

  calls sayHello(), which prints out "Hello World!" and

  calls sayHello(), which prints out "Hello World!" and

  calls sayHello(), which prints out "Hello World!" and

  calls sayHello(), which prints out "Hello World!" and

  calls sayHello(), which prints out "Hello World!" and

  calls sayHello(), which prints out "Hello World!" and

  calls sayHello(), which prints out "Hello World!" and

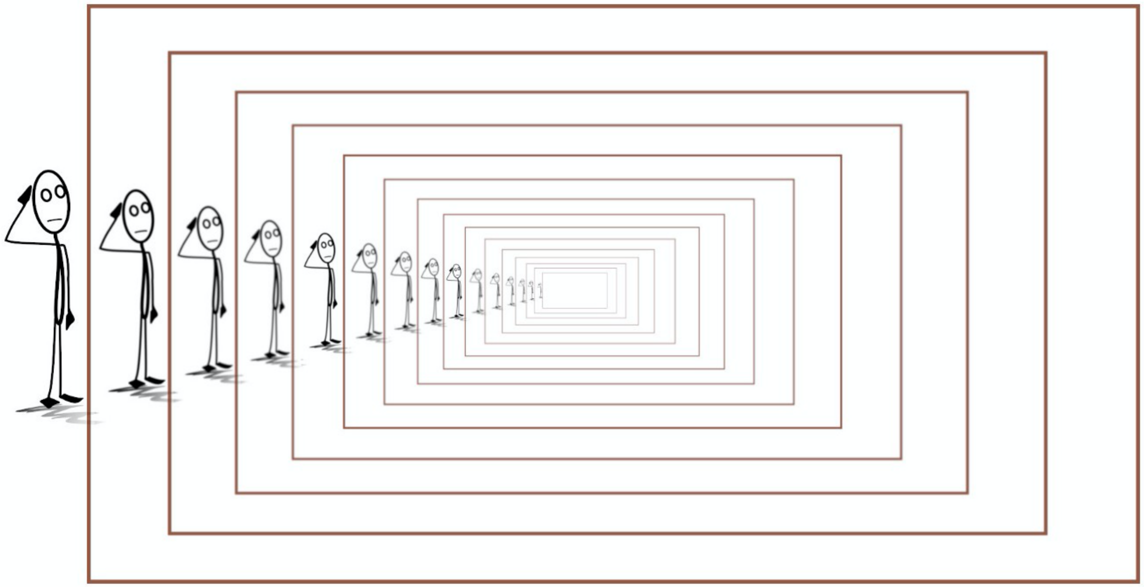
  calls sayHello(), which prints out "Hello World!" and

  …

  all of a sudden, the program crashes!

Recursion

- **Recursion** is when a method **calls itself**
- It is used when a complicated problem can be broken down into a simpler problem of the same type
- In that case you can just call the same method on the simpler problems and use the results to solve the bigger problem



Iterative example

- This program figures out the factorial of a number using an ordinary loop: factorial(5) = 1 x 2 x 3 x 4 x 5 = 120

```
public static int factorial( int N ) {
    int product = 1;

    for ( int j = 1; j <= N; j++ )          // for loop
        product = product * j;              // run several times

    return product;
}
```

Recursive example

- The following program does exactly the same thing but cleverly calls itself
- It can do this because factorial(5) = factorial(4) x 5 and factorial(4) = factorial(3) x 4
- Figure out the answer to the smaller problem first and use this to figure out the answer to the bigger problem

```
public static int factorial( int N ) {
    return N * factorial( N-1 ) ;
}
```

```
// for loop
for ( int j = 1; j <= N; j++ )
    product = product * j;
```

Base case

- There's a flaw with the previous piece of code
- The problem is that it goes on forever
- We need to get it to **stop at some point**
- In order to stop, all recursive methods must have **base case** where the answer is known automatically without calling another recursive method

```
public static int factorial( int N ) {
    return N * factorial( N-1 ) ;
}
```

// It won't stop

Base case

- The answer to **factorial(1)** will always be 1
- We can set this as the base case using an if statement

```
public static int factorial( int N ) {

    if ( N == 1 )          // Base case
        return 1;          // It will stop when N == 1

    else
        return N * factorial( N-1 );
}
```

Recursive example

- So, let us to calculate factorial(5):

  factorial(5)
  = 5 x factorial(4)
  = 5 x (4 x factorial(3))
  = 5 x (4 x (3 x factorial(2)))
  = 5 x (4 x (3 x (2 x factorial(1))))
  = 5 x (4 x (3 x (2 x 1)))
  = 5 x (4 x (3 x 2))
  = 5 x (4 x 6)
  = 5 x 24
  = 120

Recursion

- The method keeps calling itself until it reaches the **base case** and then it filters back up
- Recursion can be used to simplify a problem conceptually
- It also reduces the amount of code needed

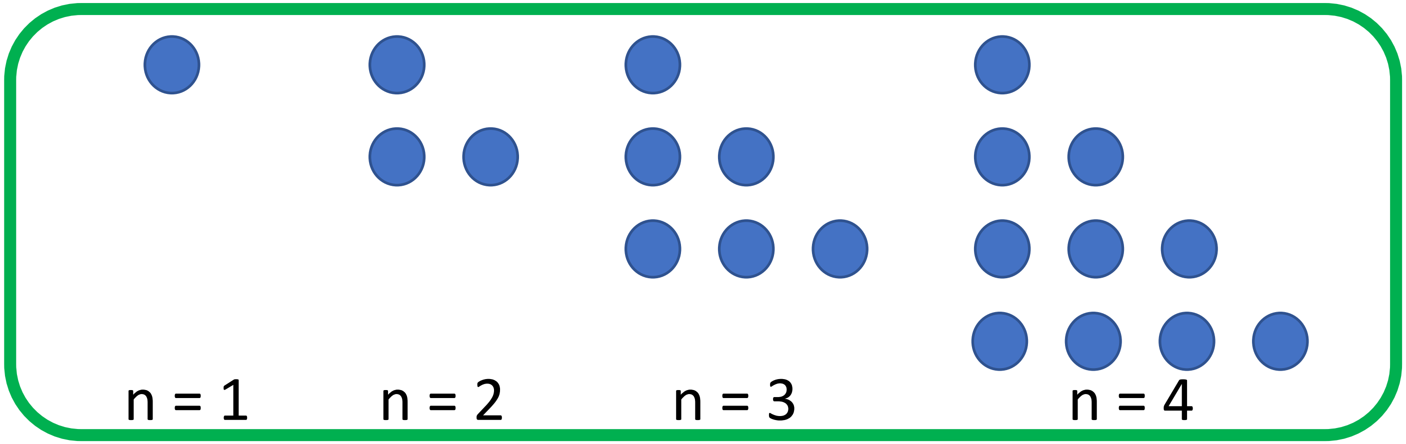
- The characteristics of a recursive method are
  - It calls itself
  - When it calls itself, it does so to solve a smaller problem
  - There's some version of the problem that is simple enough that the routine can solve it and return without calling itself

Example

- How can we find the n-th term of this sequence?

[1 3 6 10 15 21 …]

- Each time the n-th term is obtained by adding n to the term before
- The numbers are called triangular numbers because they can be visualized as a triangular arrangement of numbers
- The 3rd term is the total number of items in 3 rows etc.

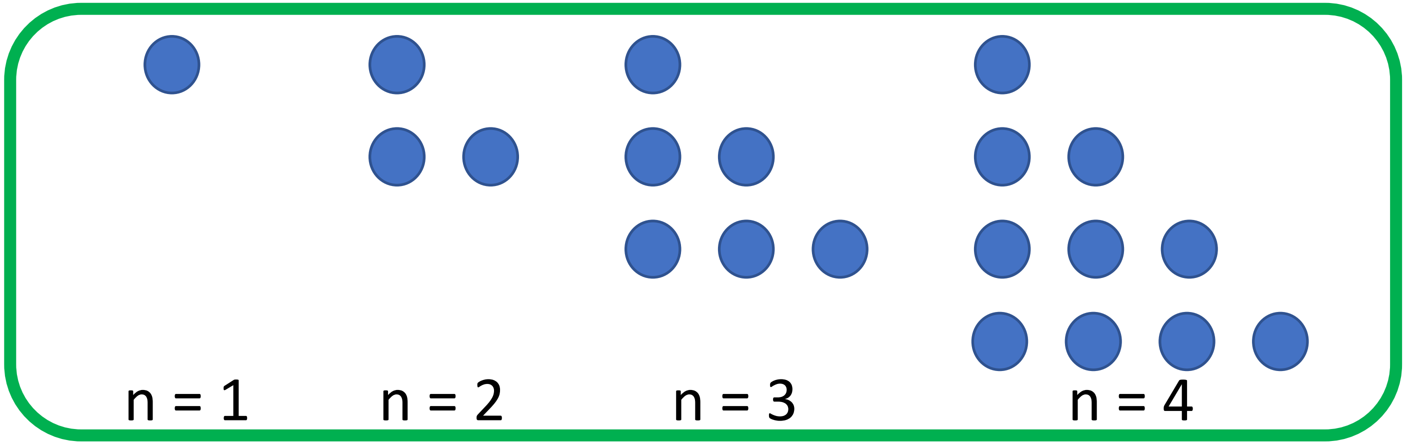


Example

- How can we find the n-th term of this sequence?

[1 3 6 10 15 21 …]

- Thus, we have the n-th term is $\sum_{i=1}^{n}(i\text{th term})$
- Example.
    - n = 1: 1
    - n = 2: 1 + 2 = 3
    - n = 3: 1 + 2 + 3 = 6



Example

- Say we want to get the fourth term we have to add

4 +3 +2 + 1 = 10

- We can do this with a while loop:

```
public static int triangle (int n) {
    int total = 0;
    while(n > 0) {
        total = total + n;
        n--;
    }
    return total;
}
```

You can write as...

====================

```
for (int i = n; i>0; i--) {
    total = total + i;
}
```

// Input: an positive integer n

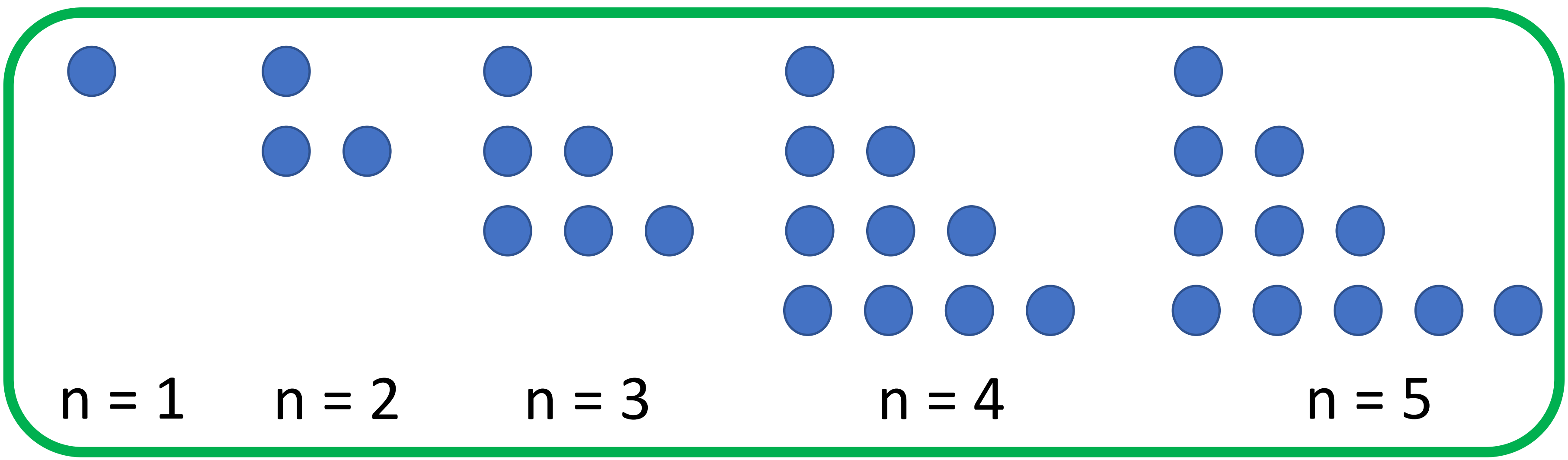
// Output: $n + (n-1) + (n-2) + \cdots + 1 = \sum_{i=1}^{n} i$

Algorithm

- Therefore the following is true:

  pins in N rows = pins in row N + pins in N-1 rows

- Here is the algorithm for recursive triangular numbers:
  - Triangle(1) = 1
  - Triangle(N) = N + Triangle(N - 1)



Example

- So, let us to calculate triangle(5):
- triangle(5)

  = 5 + triangle (4)

  = 5 + (4 + triangle (3))

  = 5 + (4 + (3 + triangle (2)))

  = 5 + (4 + (3 + (2 + triangle (1))))

  = 5 + (4 + (3 + (2 + 1)))

  = 5 + (4 + (3 + 3))

  = 5 + (4 + 6)

  = 5 + 10

  = 15

Fibonacci Series

- The Fibonacci series is as follows:

1, 1, 2, 3, 5, 8, 13…

- Each term is calculated by the sum of the two terms before it

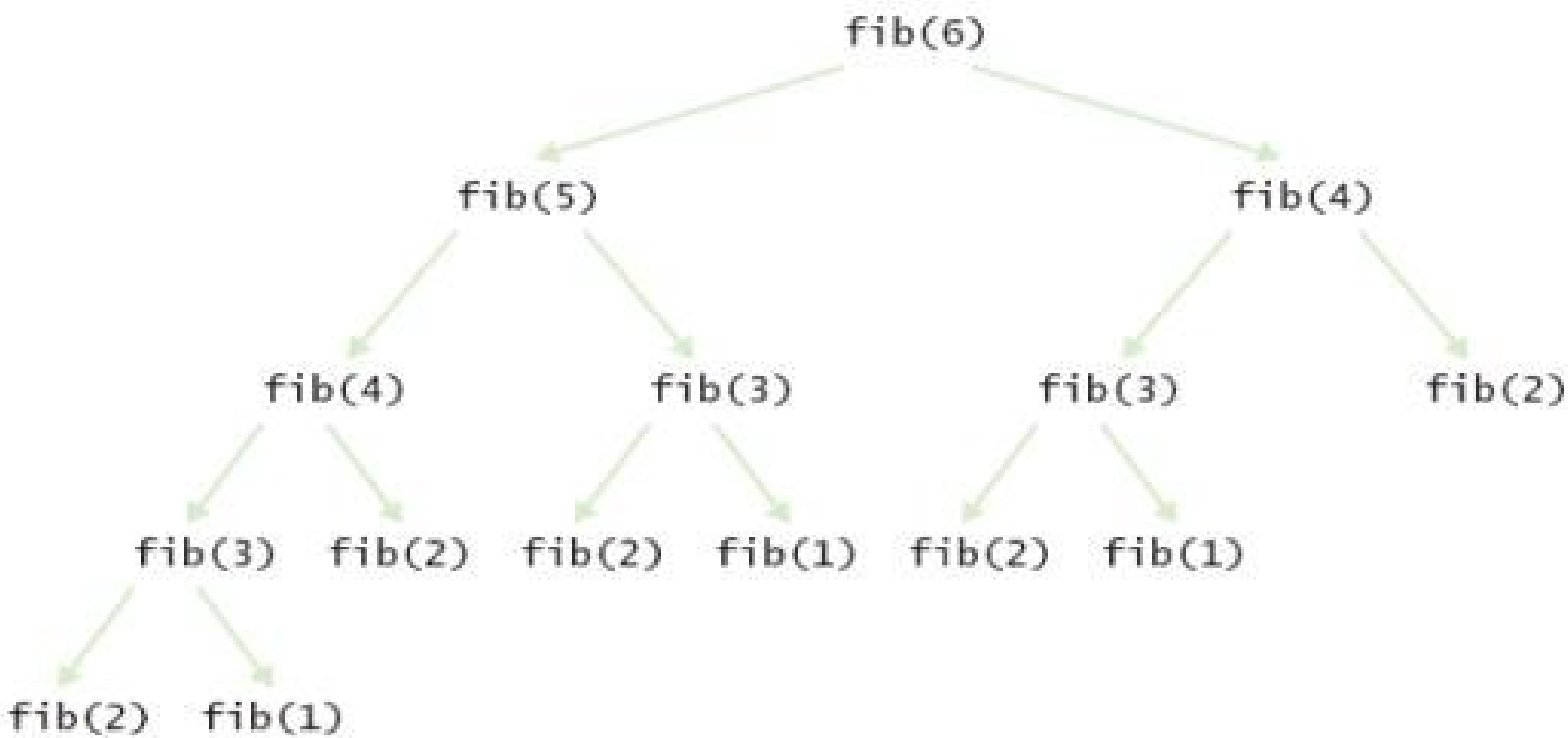
```
public static long fibonacci(int n) {
    if (n == 0) return 1;
    if (n == 1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

$$
\begin{aligned}
&f_0 = 1 \\
&f_1 = 1 \\
&f_2 = 2 = 1 + 1 = f_1 + f_0 \\
&f_3 = 3 = 2 + 1 = f_2 + f_1 \\
&\dots \\
&f_n = f_{n-1} + f_{n-2}
\end{aligned}
$$

The Efficiency of Recursion

- Recursive implementation of **fib** is straightforward
- First few calls to **fib** are quite fast
- For larger values, the program pauses an amazingly long time between outputs
- It turns out that using recursion to compute Fibonacci numbers is terrible!!
- Lesson: recursion can simplify code but is not always more efficient

Call tree for computing fib(6)



The efficiency of recursion

- Method takes so long because every level doubles the number of recursive calls

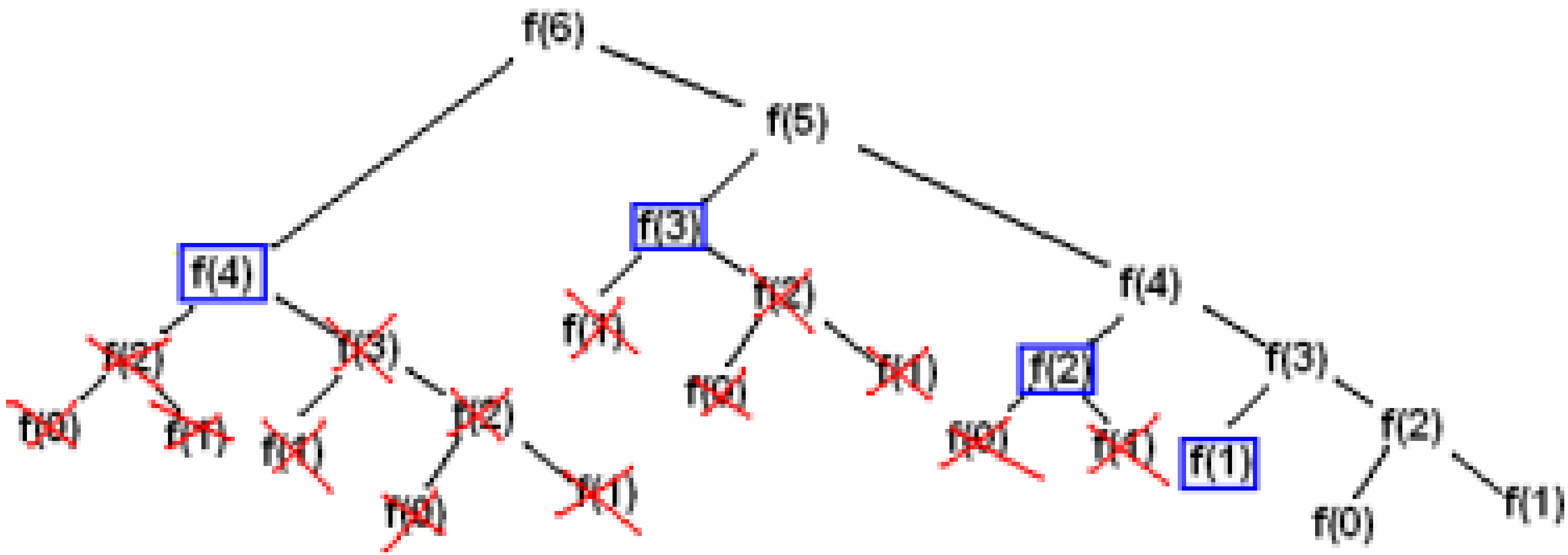
- The computation of **fib(6)** calls **fib(3)** three times and **fib(2)** five times!

- A recursive method should not call itself more than once

- If we want to use recursion, without exponential increase in number of method calls, we can use **dynamic programing**

Dynamic programming

- **Dynamic programming** (also known as **dynamic optimization**) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just **once**, and storing their solutions



Dynamic programming

- **Time Complexity: $\boldsymbol{O(n)}$**
- **Space Complexity: $\boldsymbol{O(n)}$**
- To decide whether problem can be solved by applying Dynamic programming we check for two properties.
  1. The problem involves ***overlapping sub-problems*** that need to be solved again and again. In dynamic programming we solve these sub problems only once and store it for future use
  2. If a big problem can be solved by using the solutions of the sub problems then we say that problem also has an ***optimal substructure property***

Dynamic programming

- **Bottom-Up** solution for Fibonacci Series:

```
public int fibBU(int x) {
    int fib[] = new int[x + 1];
    fib[0] = 1;
    fib[1] = 1;
    for (int i = 2; i < x + 1; i++) {
        fib[i] = fib[i - 1] + fib[i - 2];
    }
    return fib[x];
}
```

// recursion

Dynamic programming

- **Top-Down recursive** approach, involving filling up an array which lies outside the recursive method:

```
public int fibTD(int n) {
    if(n == 0) return 1;
    if(n == 1) return 1;
    if(fib[n] != 0) {
        return fib[n];
    }
    else {
        fib[n] = fibTD(n-1) + fibTD(n-2);
        return fib[n];
    }
}
```

Remember Binary Search?

- We keep dividing our search space and therefore need to keep track of the bounds
  - Upperbound
  - Lowerbound
- If the number is bigger than 51 then 51 is the new lower bound

try this

lowerBound upperBound

Data	11	13	15	…	43	**51**	62	…	99

check = (lowerBound + upperBound) / 2

Recursive Binary Search

- The binary search for arrays keeps dividing the array in half and then selecting the half where the desired cell can be
- Instead of using a loop as before, we can simply call the search function on the half of the array

- The **find** method makes the initial call to search the whole array using **recursive find**

```
public int find(long searchKey) {
    return recFind(searchKey, 0, nElems-1);
}
```

Recursive Binary Search

```
private int recFind(long searchKey, int lowerBound, int upperBound) {
    int middle;
    middle = (lowerBound + upperBound ) / 2;
    if(a[middle] == searchKey) return middle; // found it
    else if(lowerBound > upperBound) return -1; // can't find it
    else { // divide range
        if(a[middle] < searchKey) // it's in upper half
            return recFind(searchKey, middle+1, upperBound);
        else // it's in lower half
            return recFind(searchKey, lowerBound, middle-1);
    } // end else divide range
} // end recFind()
```

Recursive Binary Search Method

- Call with lowerbound=0 upperbound=15
  - Lowerbound=0
  - Upperbound=15
    - Lowerbound=0
    - Upperbound=6
      - Lowerbound=0
      - Upperbound=2
        - Lowerbound=2
        - Uppperbound=2
          - Found it at 2
          - Return 2
        - Return 2
      - Return 2
    - Return 2
  - Return 2

Divide and Conquer

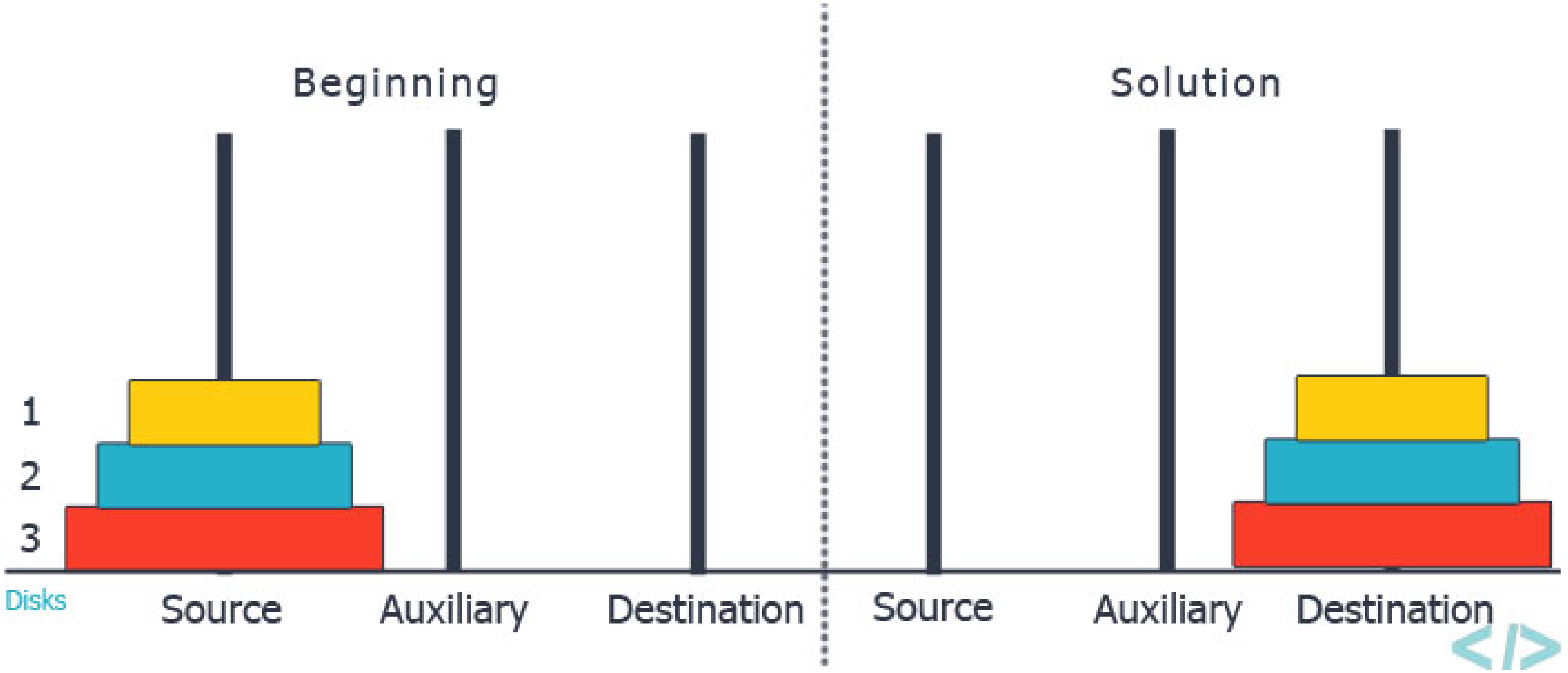
- Recursive binary search is an example of the **divide and conquer approach**
- The big problem is divided into two smaller problems and each one is solved separately
- These are then divided into even smaller problems etc.
- The process continues until you get to the base case which can be solved easily with no further division
- The answer is then filtered back through all the recursive method calls

Content

- Introduction
- **Towers of Hanoi**
- Palindromes
- Raising to a power
- Mergesort

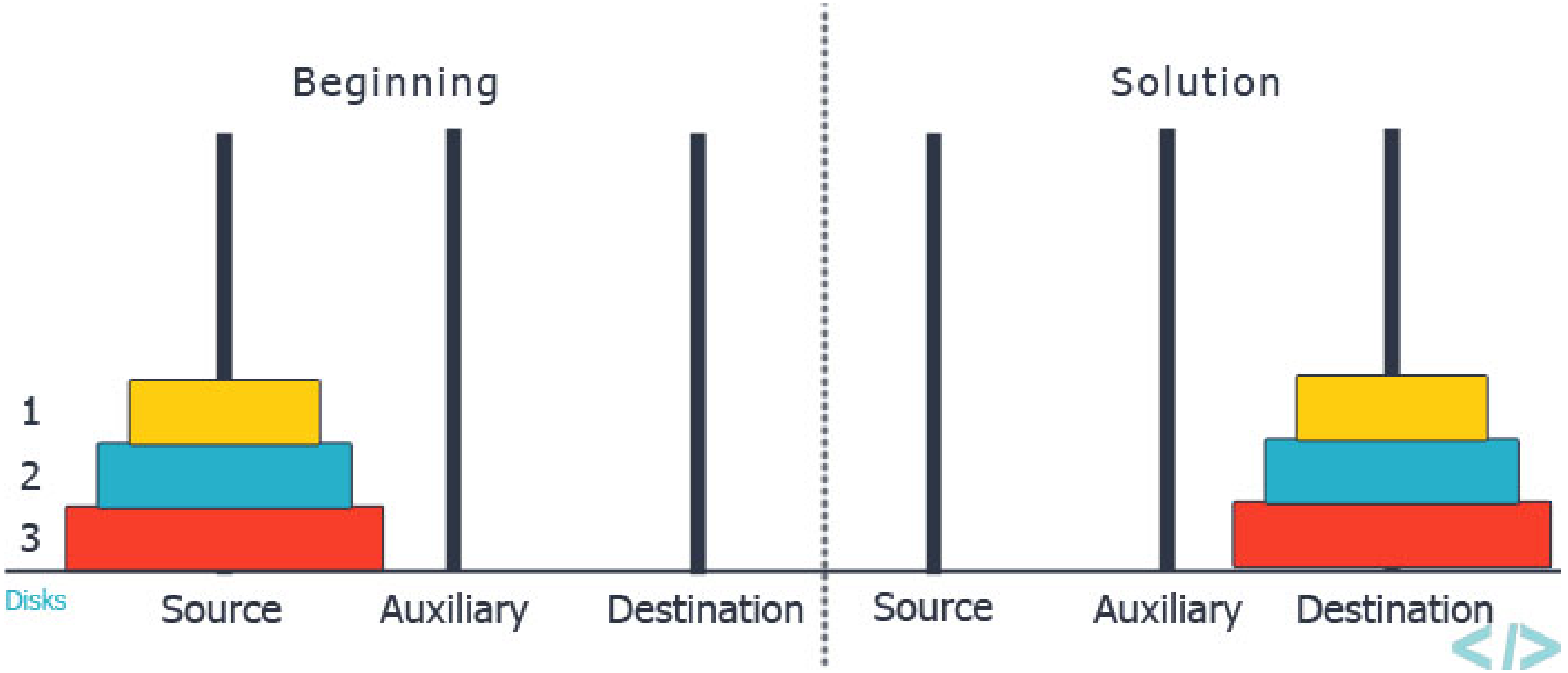
Towers of Hanoi

- We need to move all the disc from the first pole to the third pole with the smallest disc at the top and the largest at the bottom under the below conditions
- 1. Only one disc can be moved at a time.
- 2. Larger disc cannot be placed on a smaller disc.

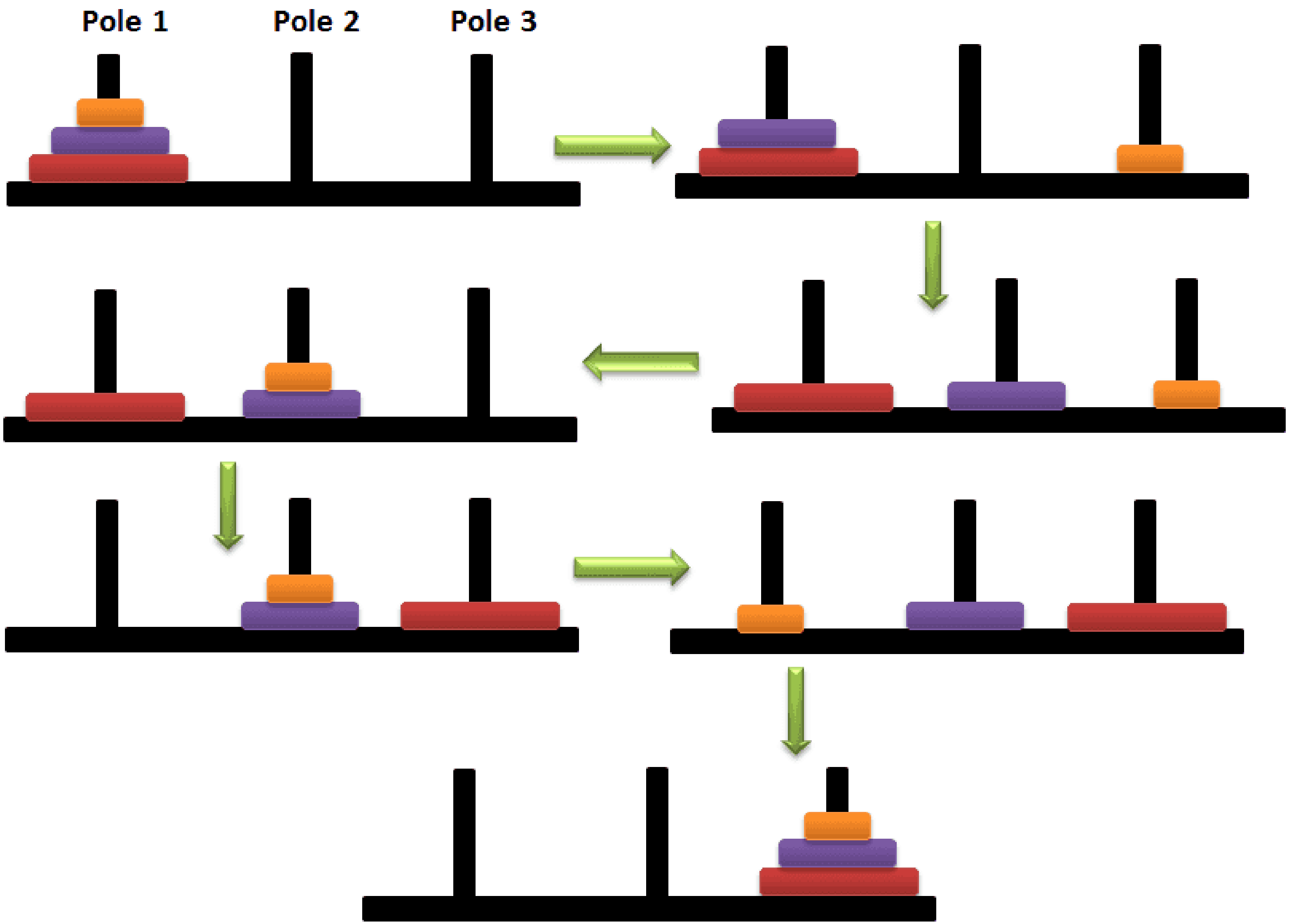


Towers of Hanoi

- In the late Victorian era, a toy came out that was based on an idea of a French mathematician, Edouard Lucas. The toy consisted of a set of three rods and a set of discs
- The idea is to move all the discs one at a time without ever placing a disc on top of a smaller one



Solution

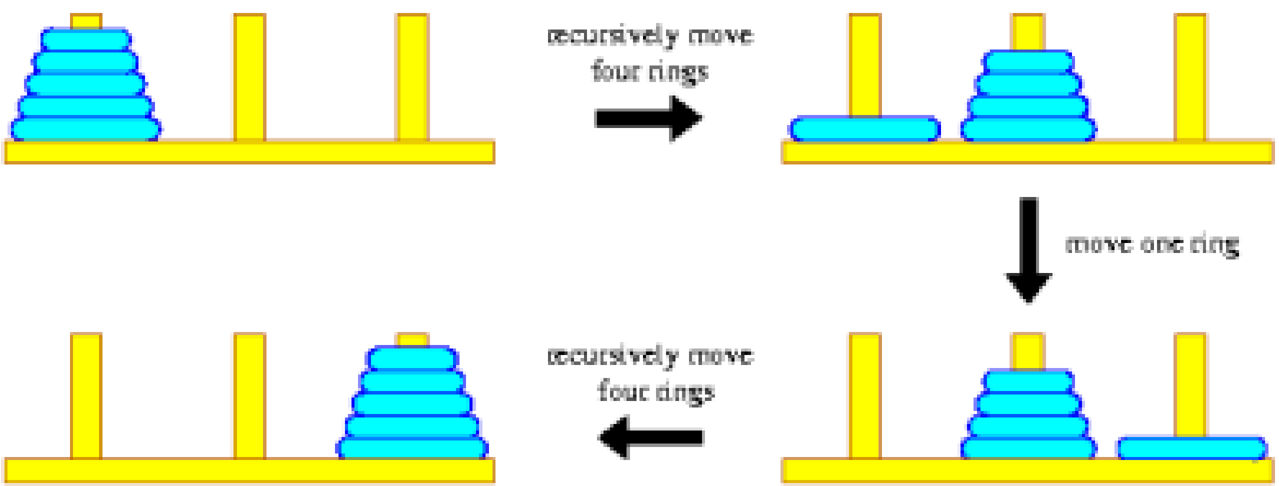


Legend

- Somewhere in Tibet, in a remote temple, monks labor day and night to transfer 64 golden disks from one of three diamond-studded towers to another
- When all the discs have been correctly moved, the universe will end
- Do we need to be worried? – Not if we remember Big O Notation
- The problem is exponential $O(2^n)$
- At a second per move, it will take the monks about five hundred billion years

Recursive Algorithm

- Lets call the initial tree-shaped arrangement of disks a tree and the smaller groupings subtrees
- In order to transfer 5 disks, one of the intermediate steps involves a subtree of 4 disks
- The creation of a subtree is the only way to transfer a larger disk from one tower to another



Recursive Algorithm

- We can break the problem into smaller and smaller subtrees
- Assuming there are *n* disks, the algorithm is
  - 1. Move the subtree of the top *n-1* disks from source rod to intermediate rod
  - 2. Move the remaining largest disk from source rod to destination rod
  - 3. Now solve moving the subtree from intermediate rod to destination rod
- Keep calling the recursive algorithm for moving the smaller subtrees
- What's the base case? When you're moving only one disk, you just move it, there's nothing else to do

Java Implementation

```
public class TowersApp {
	static int nDisks = 3;
	public static void main(String[] args) {
		doTowers(nDisks, 'A', 'B', 'C');
	}
	public static void doTowers(int topN, char src, char inter, char dest) {
		if(topN==1) System.out.println("Disk 1 from " + src + " to "+ dest);
		else {
			doTowers(topN-1, src, dest, inter); // src to inter
			System.out.println("Disk " + topN +  " from " + src + " to "+ dest);
			// move bottom
			doTowers(topN-1, inter, src, dest); // inter to dest
		}
	}
}
```

Output

Disk 1 from A to C

Disk 2 from A to B

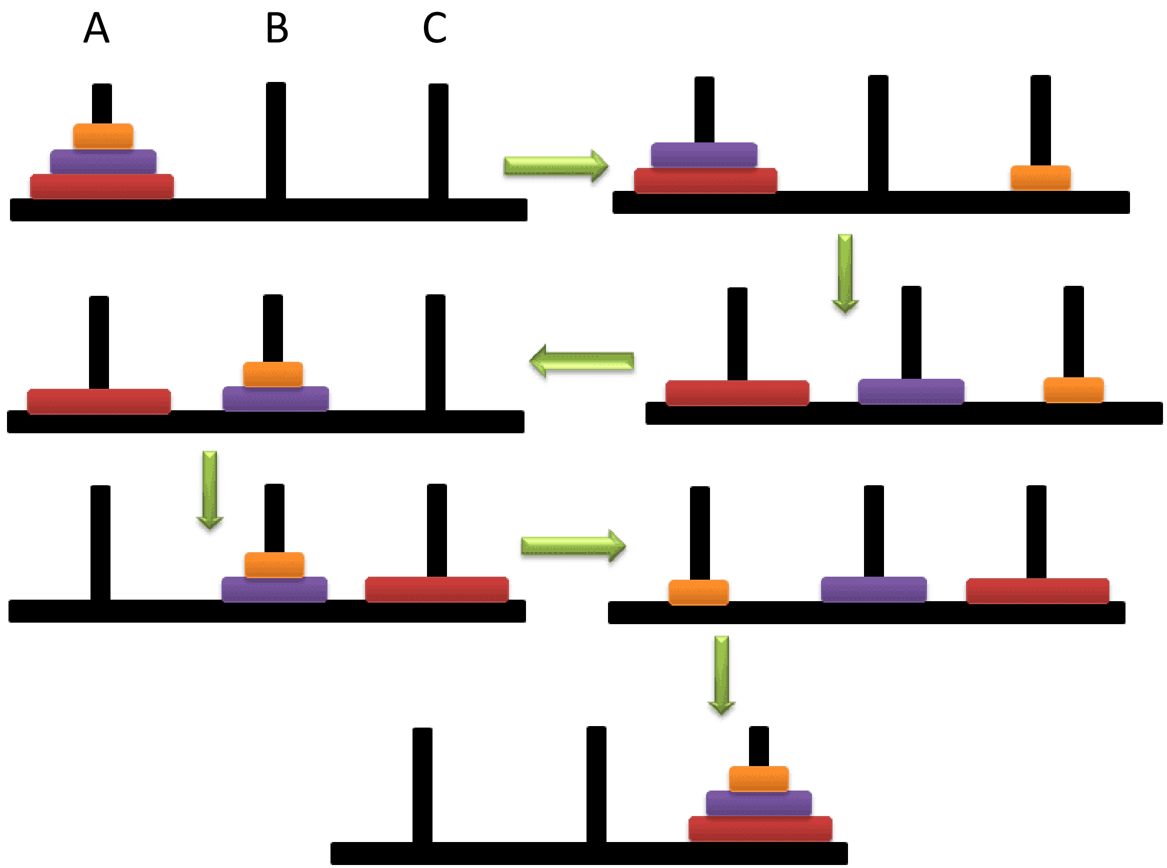
Disk 1 from C to B

Disk 3 from A to C

Disk 1 from B to A

Disk 2 from B to C

Disk 1 from A to C



- How about those monks?
- Put nDisks = 64 and what output do you get?

Content

- Introduction
- Towers of Hanoi
- **Palindromes**
- Raising to a power
- Mergesort

Palindromes

回文

- A palindrome is a phrase that reads the same forwards as backwards
- You can also think about it as a string whose first half is a mirror image of its second half

“rats live on no evil star”

“ten animals I slam in a net”

A
EVE
RADAR
REVIVER
ROTATOR
LEPERS REPEL
MADAM I’M ADAM
STEP NOT ON PETS
DO GEESE SEE GOD
PULL UP IF I PULL UP
NO LEMONS, NO MELON
DENNIS AND EDNA SINNED
ABLE WAS I ERE I SAW ELBA
A MAN, A PLAN, A CANAL, PANAMA
A SANTA LIVED AS A DEVIL AT NASA
SUMS ARE NOT SET AS A TEST ON ERASMUS
ON A CLOVER, IF ALIVE, ERUPTS A VAST, PURE EVIL; A FIRE VOLCANO

Think Recursively

- How do we make the problem smaller?
  - Remove both the first and last characters
  - Remove a character from the middle
- Most promising simplification: remove first and last characters

“AVA", is a palindrome too!

- A word is a palindrome if
  - The first and last characters match, and
  - Word obtained by removing the first and last characters is a palindrome

Palindromes

- If we have a palindrome like **radar** we can check if the first and last letters are the same
- If they are not then it is not a palindrome
- We now need to check the middle letters to see if they are a palindrome
- Call the recursive method on the middle letters

Palindromes

- To pick out the middle letters we can use the substring method

word.substring(1, word.length()-1);

- The base case is when we have just one or two letters left and all the other letters have checked out

`r a d a r`

Content

- Introduction
- Towers of Hanoi
- Palindromes
- **Raising to a power**
- Mergesort

Raising to a power

- We can use recursion to raise a number to a power
- For example, 2^8 is $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$
- Do we really need to multiply these all out?
- After we've calculated $2 \times 2 \times 2 \times 2$we know we can just square it
- But in order to get $2^4 = (2^2)^2$ we only need to multiply 2×2 and then square that!
- We can keep halving the problem
- In fact, we can figure out 2^8 using just 3 multiplications instead of 7 – that is $O(logn)$ instead of $O(n)$

Raising to a power

- In the previous example, the power was always even, so could be easily divided by two
- If the power happens to be odd then we need to subtract one from the power, solve that, and then multiply it again to account for the power we subtracted
- Base case is raising to the power of zero

```
int power(int k, int n) { // raise k to the power n
    if (n == 0) return 1;
    else{
        int t = power(k, n/2); //if odd, will discard remainder
        if ((n % 2) == 0) return t * t;
        else return k * t * t; //extra multiplication to make up
    }
}
```

Content

- Introduction
- Towers of Hanoi
- Palindromes
- Raising to a power
- **Mergesort**

Mergesort

- Our final example of recursion is Mergesort
- This is more efficient than any of the other sorting algorithms we have considered, which were all $\boldsymbol{O(n^2)}$
- Mergesort is $\boldsymbol{O(n\, log\, n)}$
- If the number of items to be sorted is 10,000 then $\boldsymbol{n^2}$ is 10 million whereas $\boldsymbol{n\, log\, n}$ is only 40,000
- If this many items can be sorted in a second by Mergesort, it would take nearly an hour using insertion sort

Merge Sort Example

- The idea of mergesort is to divide an array in half and sort each half

- They don’t have to be the same size
- Both halves are ***merged*** into a separate workspace array
- Keeps comparing the lowest number in each of the halves before copying one into a workspace array

Merge Sort Example

- Merge the two sorted arrays into a single sorted array

Merge

5	9	10	12	17
5	9	10	12	17
5	9	10	12	17
5	***9***	10	12	17
5	9	***10***	12	17
5	9	10	12	17
5	9	10	***12***	17
5	9	10	12	***17***
5	9	10	12	17
5	9	10	12	17

1	8	11	20	32
1	8	11	20	32
1	***8***	11	20	32
1	8	11	20	32
1	8	11	20	32
1	8	***11***	20	32
1	8	11	20	32
1	8	11	20	32
1	8	11	***20***	32
1	8	11	20	***32***

Workspace array

1									
1	**5**								
1	5	**8**							
1	5	8	**9**						
1	5	8	9	**10**					
1	5	8	9	10	**11**				
1	5	8	9	10	11	12			
1	5	8	9	10	11	12	17		
1	5	8	9	10	11	12	17	20	
1	5	8	9	10	11	12	17	20	32

How to Merge

- **Here are two lists to be merged:**
  - First: (12, 16, 17, 20, 21, 27)
  - Second: (9, 10, 11, 12, 19)
- **Step 1. Checkout 12 and 9**
  - Because 9 < 12, so move 9 to workspace
  - First: (12, 16, 17, 20, 21, 27)
  - Second: (10, 11, 12, 19)
  - Workspace: (9)
- **Step 2. Checkout 12 and 10**
  - Because 10 < 12, so move 10 to workspace
  - First: (12, 16, 17, 20, 21, 27)
  - Second: (11, 12, 19)
  - Workspace: (9, 10)

Merge Example

- **Step 3. Checkout 12 and 11**
  - Because 11 < 12, so move 11 to workspace
  - First: (12, 16, 17, 20, 21, 27)
  - Second: (12, 19)
  - Workspace: (9, 10, 11)
- **Step 4. Checkout 12 and 12**
  - Because 12 = 12, so move 12 to workspace
    - We don’t care that you chose which one
  - First: (16, 17, 20, 21, 27)
  - Second: (12, 19)
  - Workspace: (9, 10, 11, 12)

Merge Example

- **Step 5. Checkout 16 and 12**
  - Because 12 < 16, so move 12 to workspace
  - First: (16, 17, 20, 21, 27)
  - Second: (19)
  - Workspace: (9, 10, 11, 12, 12)
- **Step 6. Checkout 16 and 19**
  - First: (17, 20, 21, 27)
  - Second: (19)
  - Workspace: (9, 10, 11, 12, 12, 16)
- **Step 7. Checkout 17 and 19**
  - First: (20, 21, 27)
  - Second: (19)
  - Workspace: (9, 10, 11, 12, 12, 16, 17)

Merge Example

- **Step 9. Checkout 20 and 19**
  - First: (20, 21, 27)
  - Second: ( )
  - Workspace: (9, 10, 11, 12, 12, 16, 17, 19)

- **Step 10. Checkout 20 and empty list**
  - **Because the First array is empty, we can move all elements of Second to Worksapce**
  - First: ( )
  - Second: ( )
  - Workspace: (9, 10, 11, 12, 12, 16, 17, 19, 20, 21, 27)

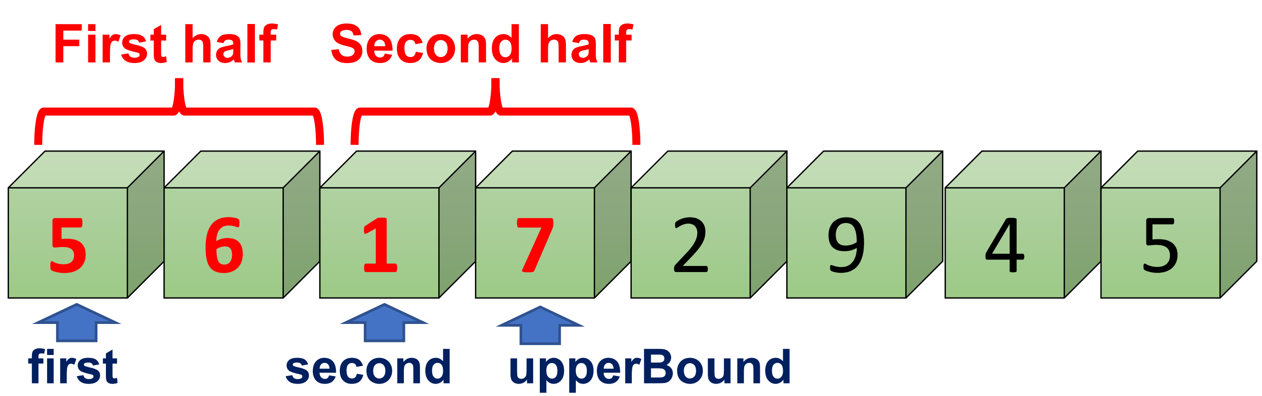
Analysis of Merge Example

- To merge the following lists, we use 10 steps.
  - First: (12, 16, 17, 20, 21, 27)
  - Second: (9, 10, 11, 12, 19)

- Except one array being empty, we move an element to Workspace at each step
- So, if we want to marge any two lists **s1** and **s2**, we need at most **s1.length() + s2.length()** steps

Java Implementation

- In the following merge method
  - theArray is the array being sorted
  - first, second and upperBound define the edges of the two halves being sorted. first and second are incremented as items are copied into the workspace
    - first ➔ second - 1 is the first half
    - second ➔ upperBound is the second half
  - workSpace is the intermediate array into which values are copied
  - The values from workSpace are then copied back into theArray

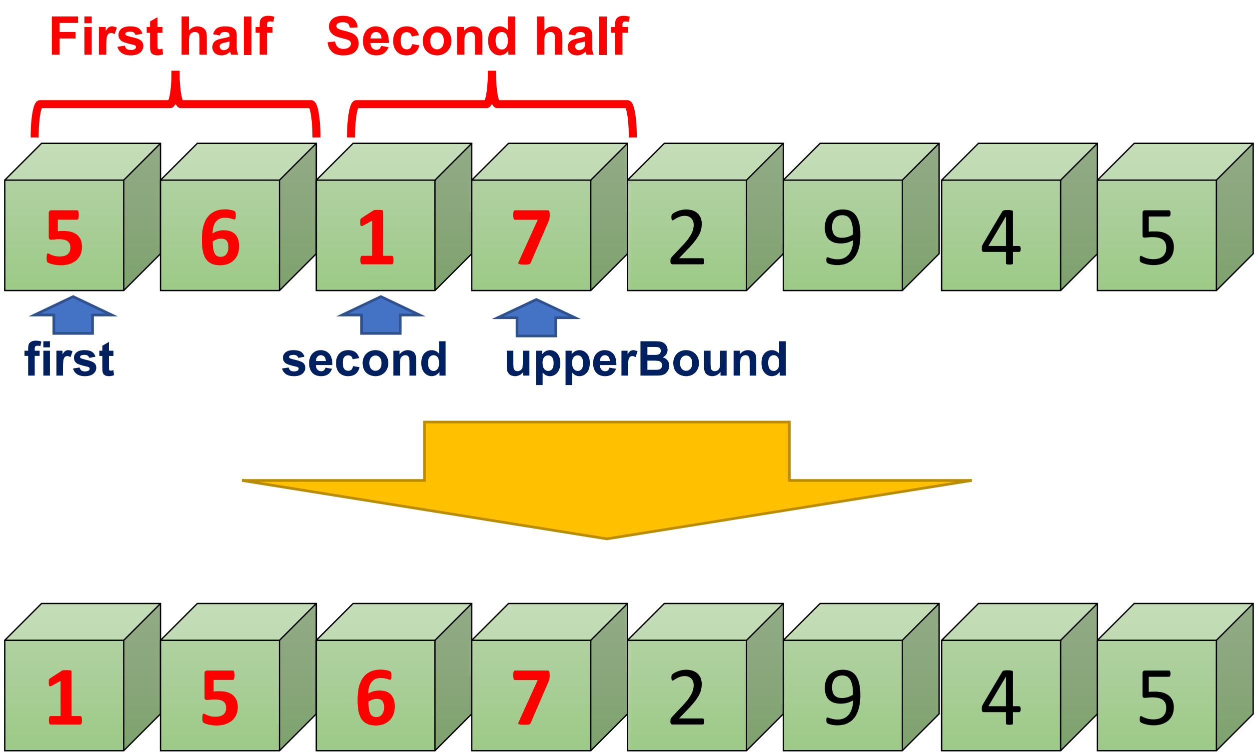


Java Implementation

```
public void merge(long[] workSpace, int first, int second, int upperBound) {
    int j = 0; // workspace index
    int lowerBound = first;
    int mid = second - 1;
    int n = upperBound-lowerBound+1; // # of items
    while(first <= mid && second <= upperBound) //halves not empty
        if( theArray[first] < theArray[second] )
            workSpace[j++] = theArray[first++];
        else
            workSpace[j++] = theArray[second++];
        while(first <= mid) //check first half for remaining
            workSpace[j++] = theArray[first++];
        while(second <= upperBound) //check second half for remaining
            workSpace[j++] = theArray[second++];
        for(j = 0; j<n; j++)
            theArray[lowerBound+j] = workSpace[j]; //copy the workspace back
} // end merge()
```

Merge

- Merges two areas of an array into a workspace (temporary array)
- Copies the workspace back onto original array



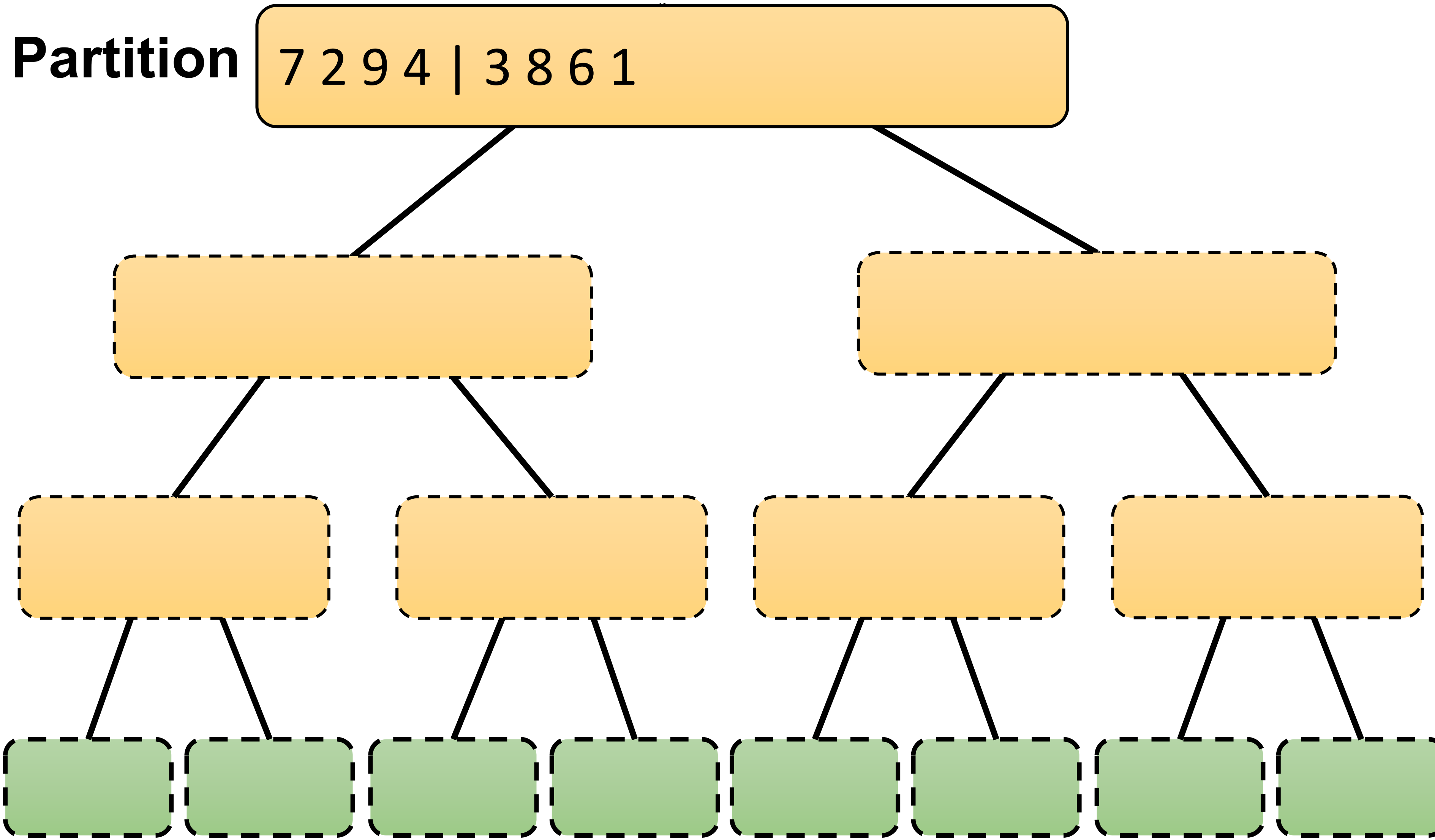
recMergeSort method

- Computes the midpoint
- Calls recMergeSort method on each of the halves
- Calls merge method to merge the two sorted halves back together
- Base case is when the range contains only one element (lowerBound==upperBound)
- A single element is always sorted (obviously!) so it just returns itself

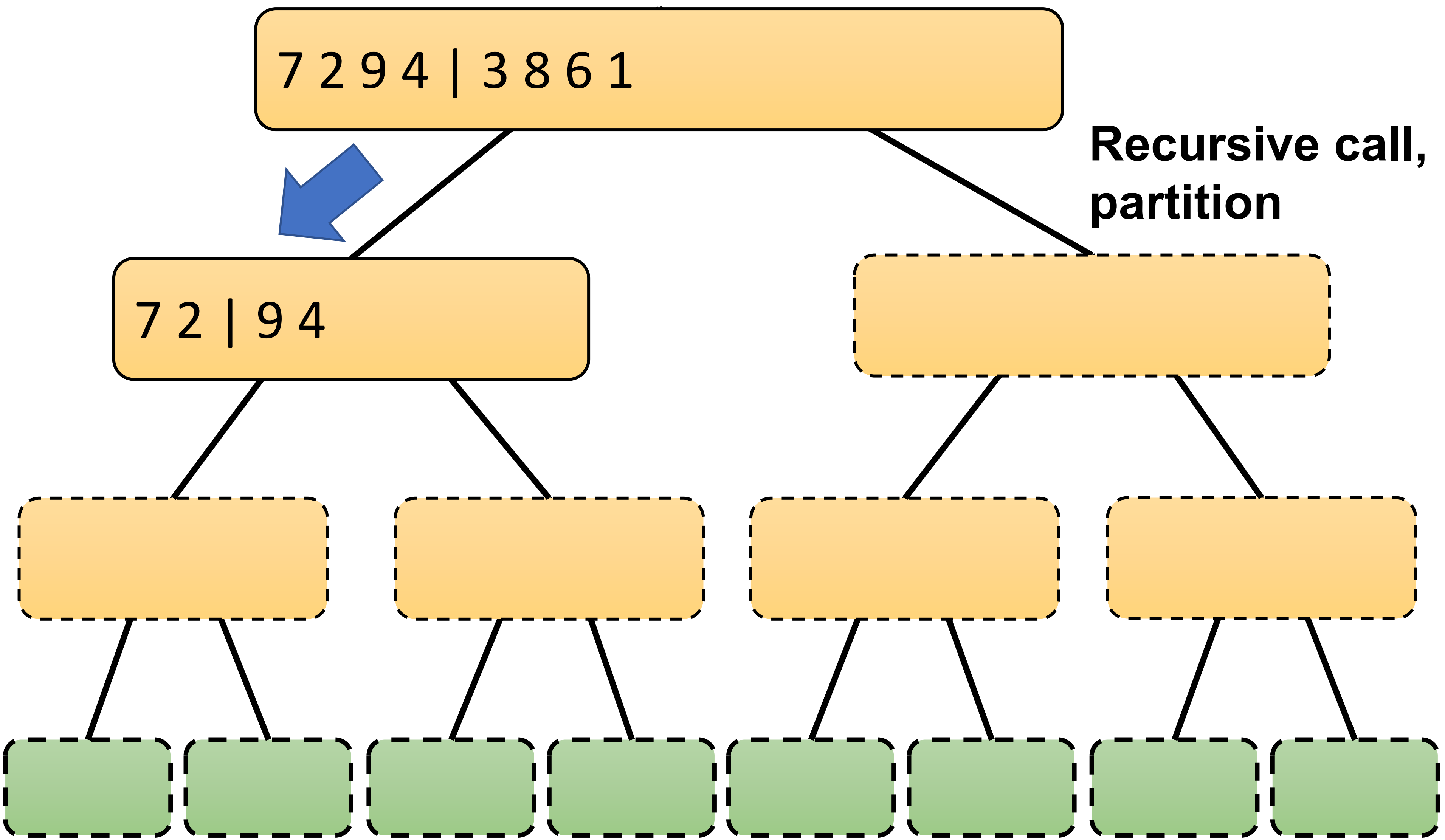
MergeSort

Original	24	13	26	1	12	27	38	15							
Divide in 2	24	13	26	1		12	27	38	15						
Divide in 4	24	13		26	1		12	27		38	15				
Divide in 8	**24**		**13**		**26**		**1**		**12**		**27**		**38**		**15**
Merge 2	**13**	**24**			**1**	**26**			**12**	**27**			**15**	**38**	
Merge 4	**1**	**13**	**24**	**26**					**12**	**15**	**27**	**38**			
Merge 8	**1**	**12**	**13**	**15**	**24**	**26**	**27**	**38**							

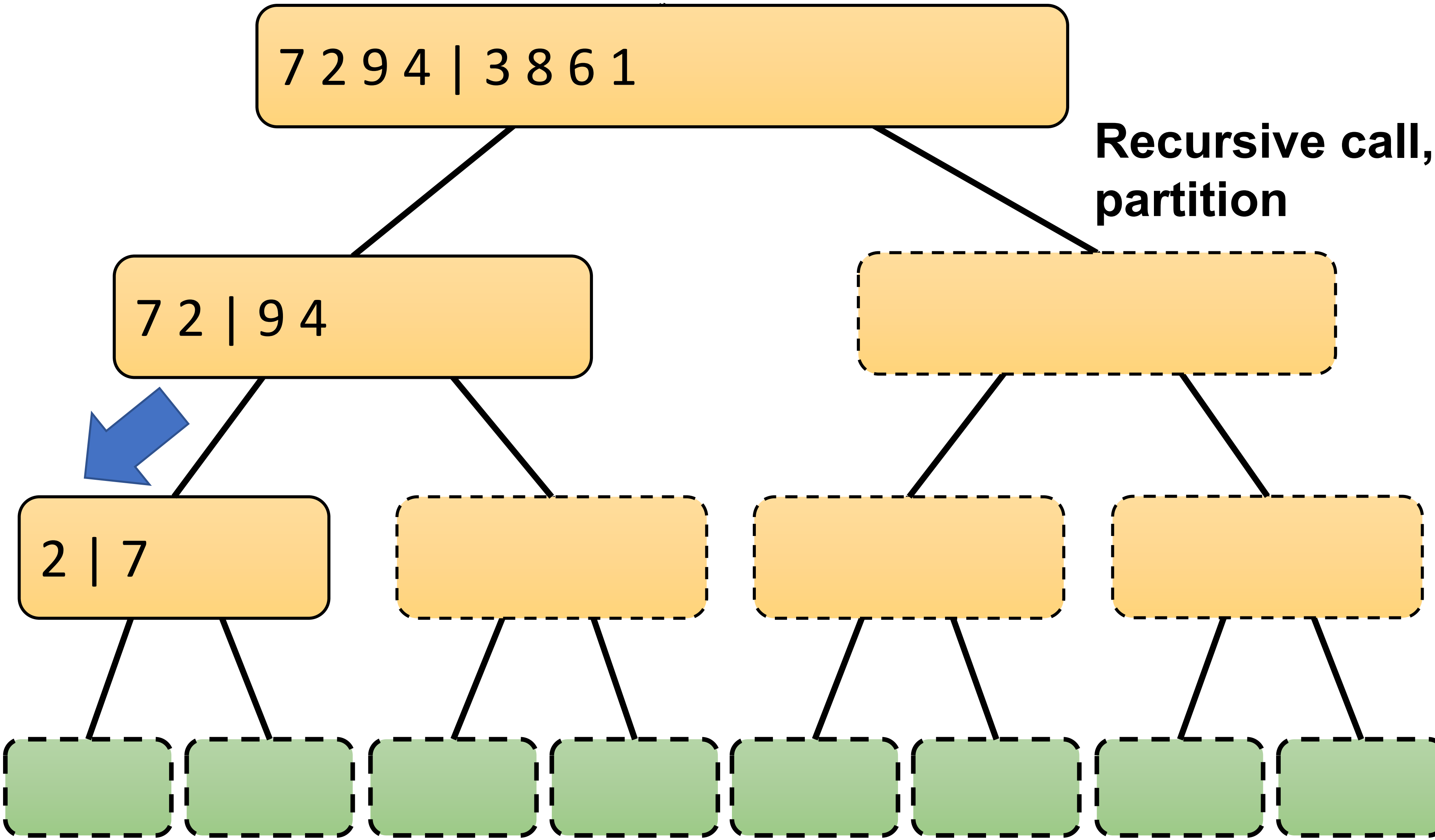
Execution Example



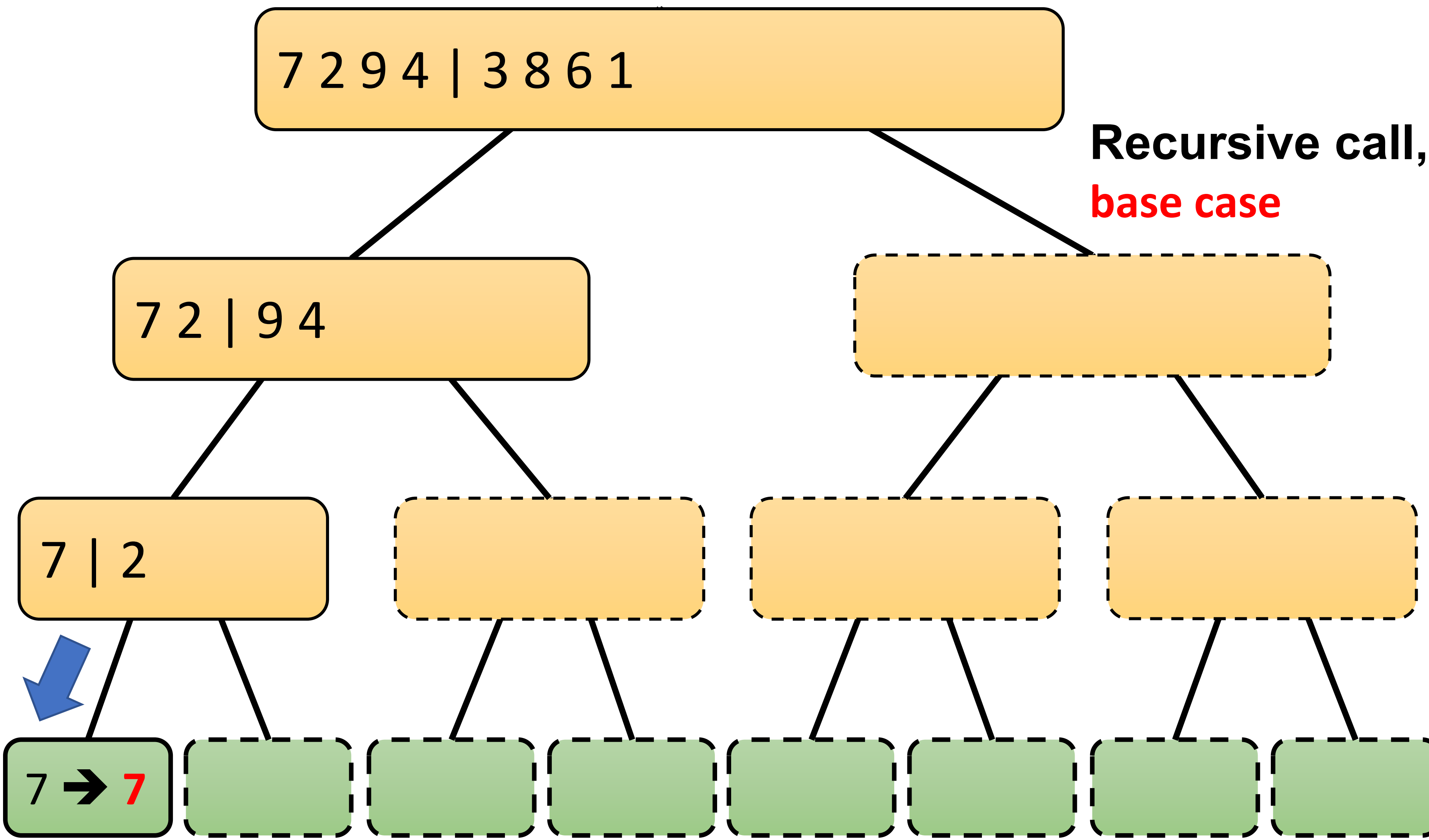
Execution Example



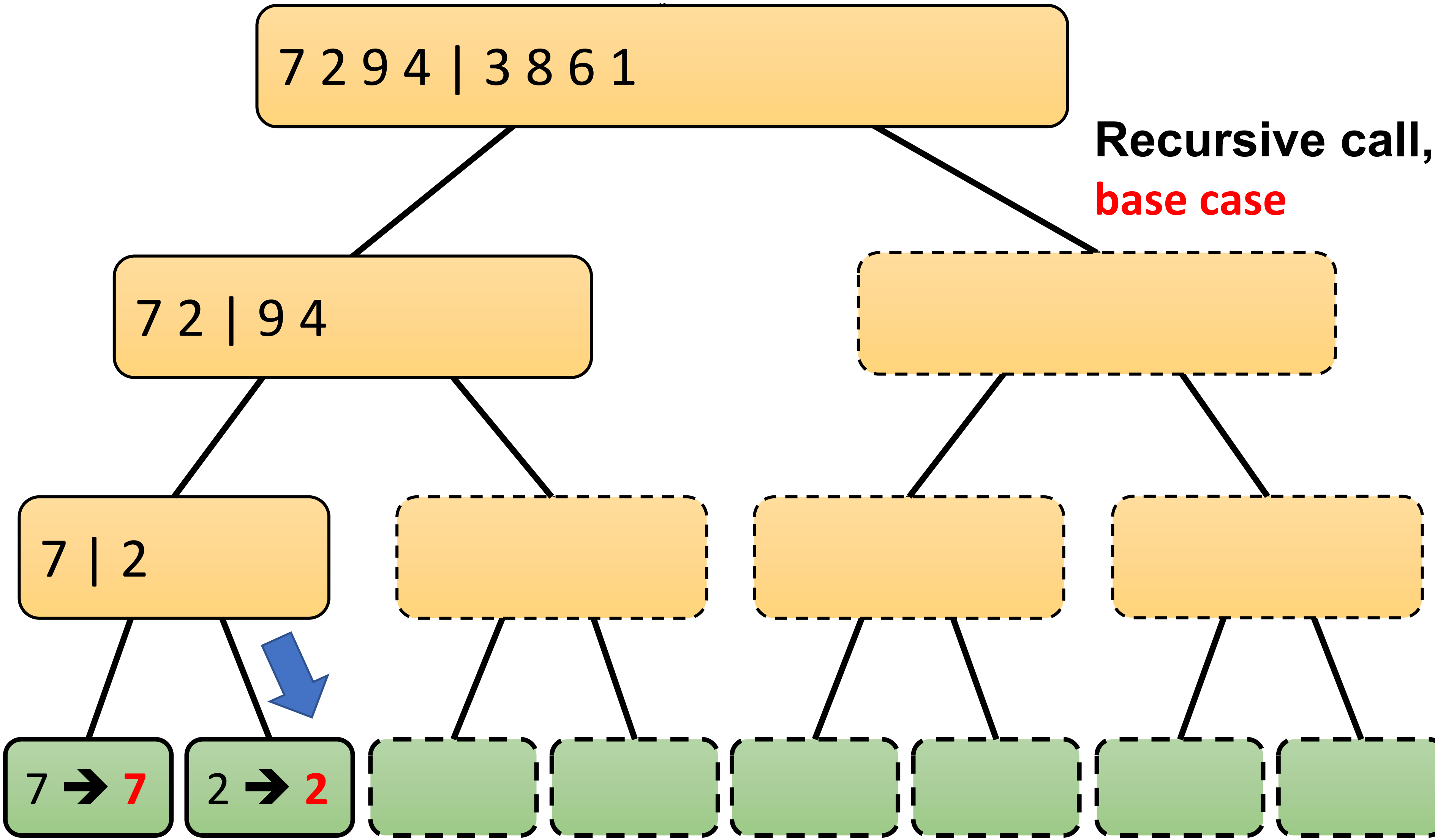
Execution Example



Execution Example

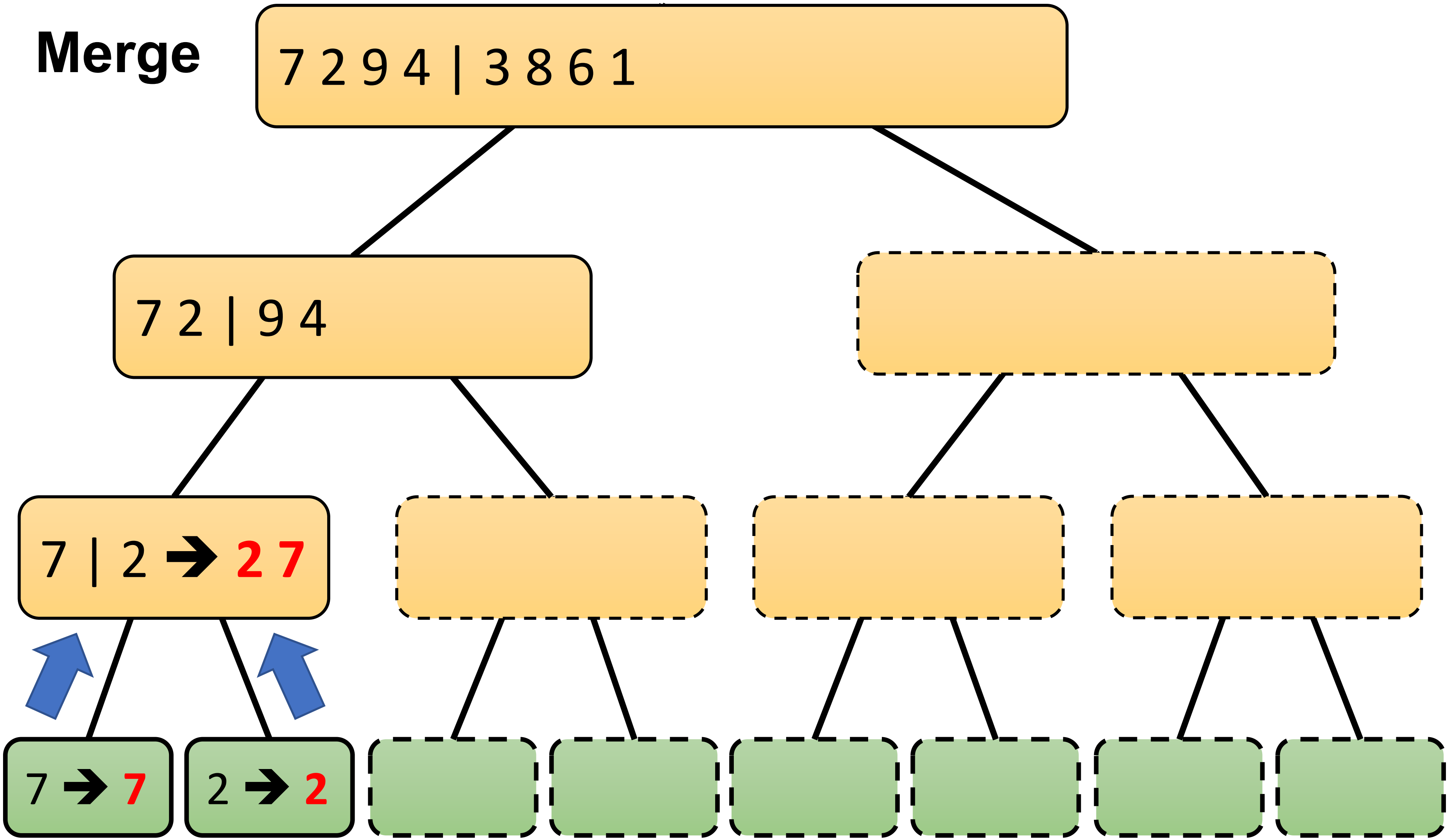


Execution Example



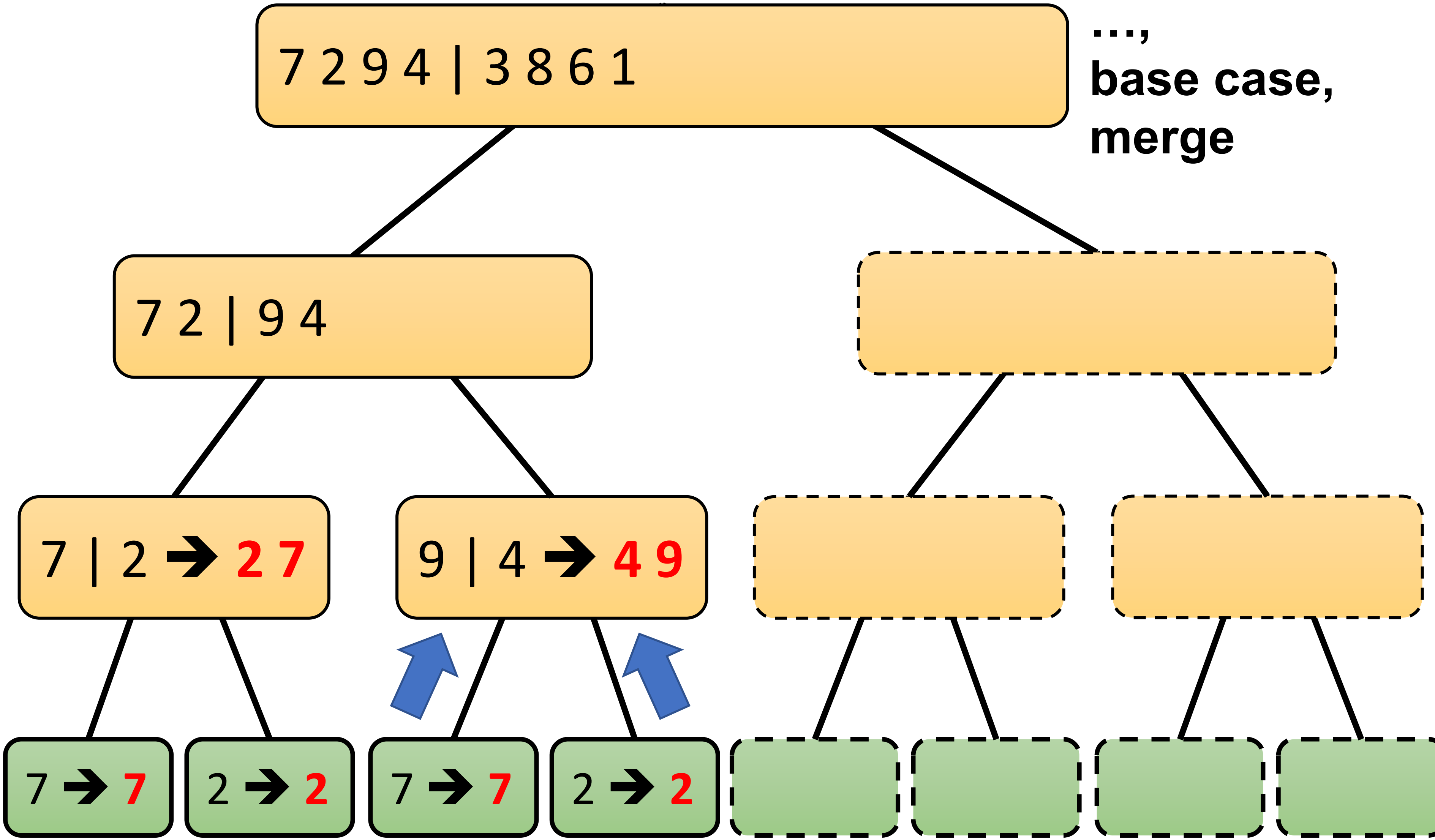
Execution Example

Merge

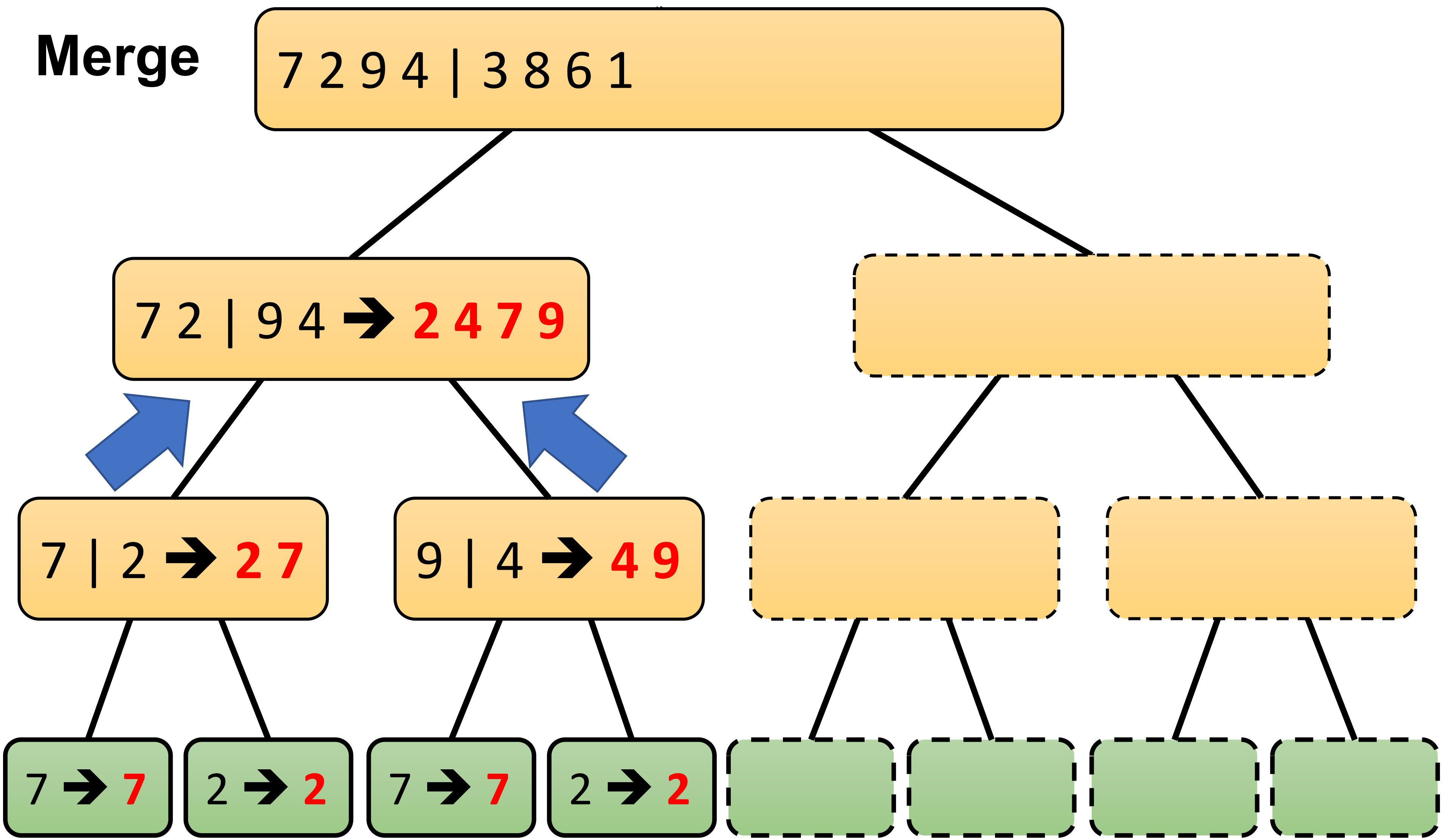


Execution Example

Recursive call, …, base case, merge

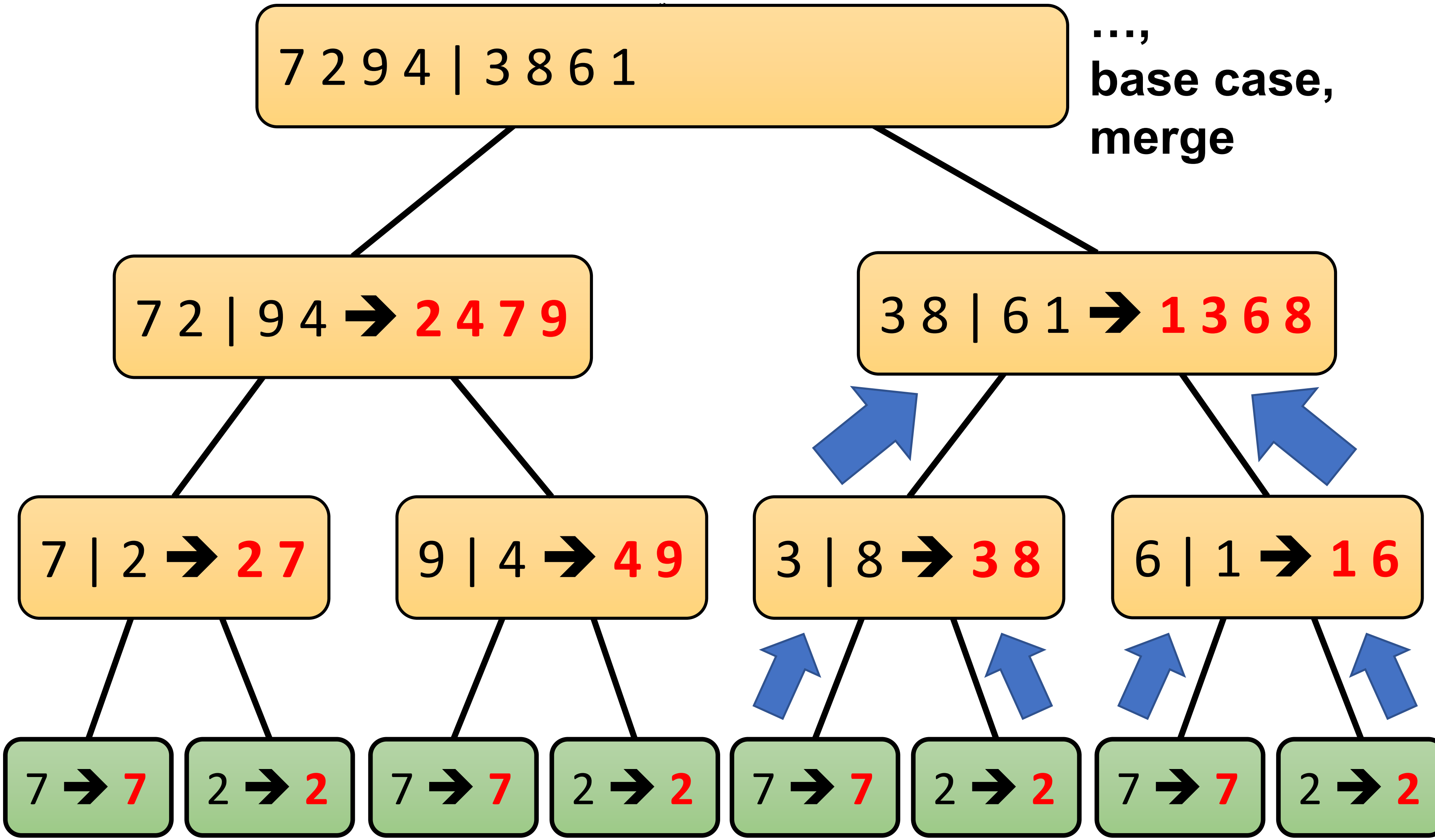


Execution Example

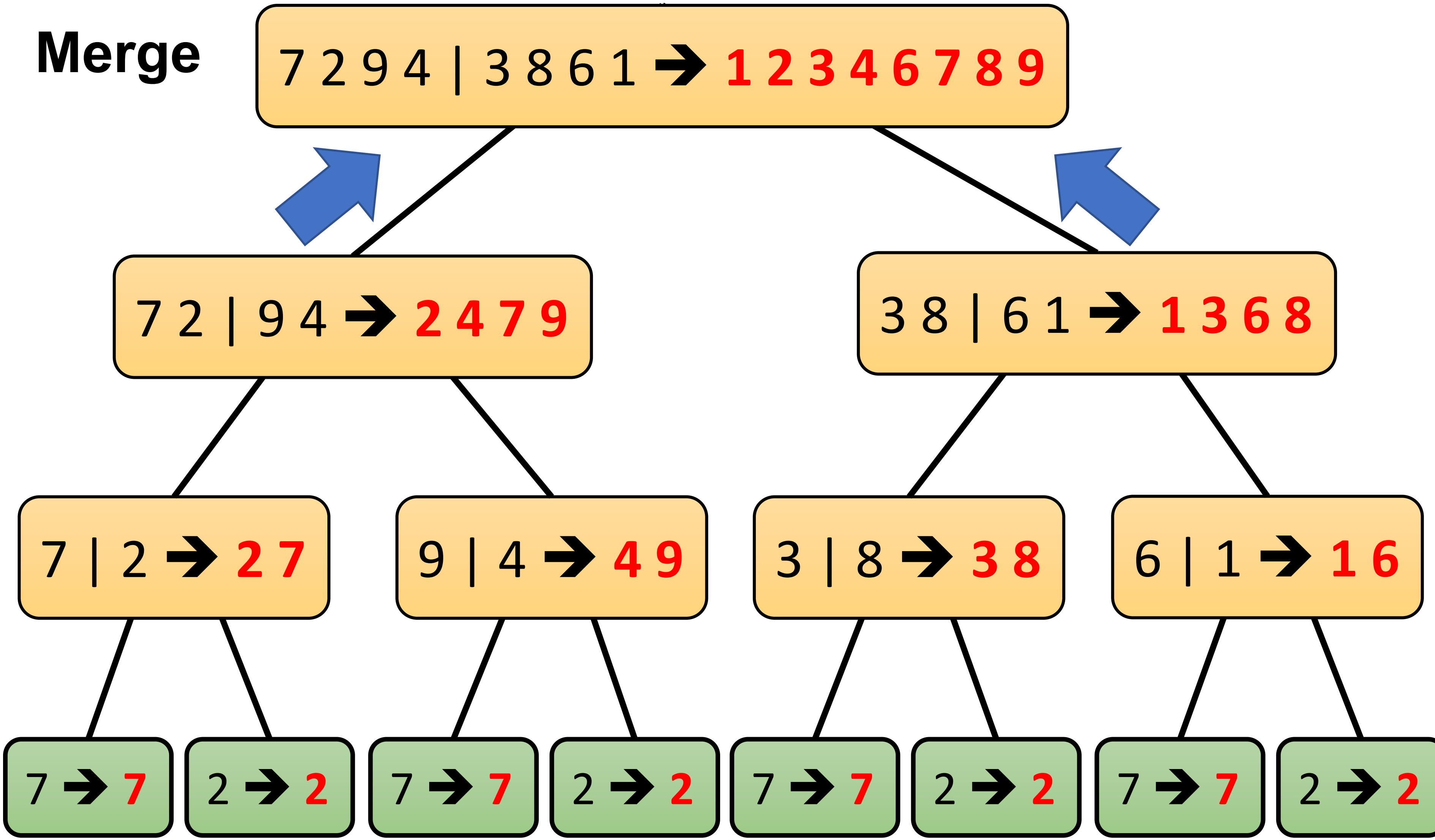


Execution Example

Recursive call, …, base case, merge



Execution Example

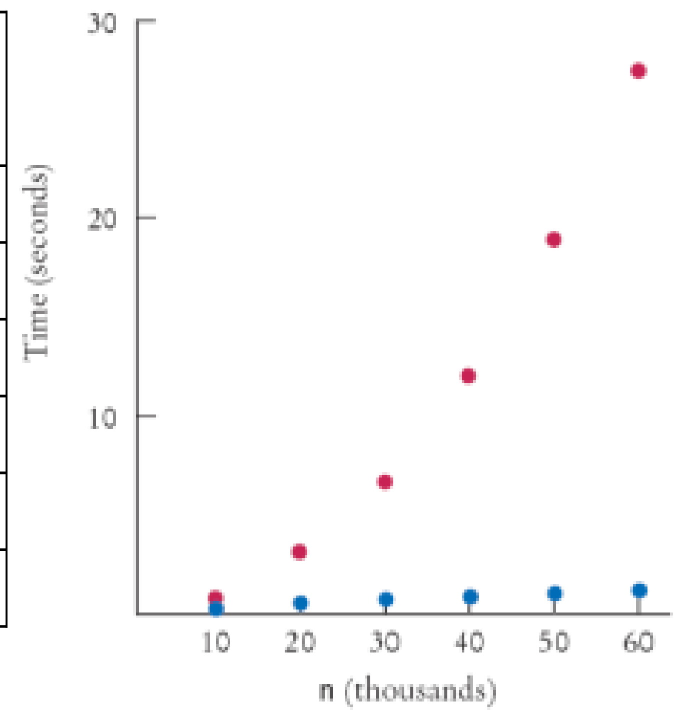


Java Implementation

```
public void recMergeSort(long[] workSpace, int lowerBound, int
upperBound) {
    if(lowerBound == upperBound) // if range is 1,
        return; // no use sorting
    else { // find midpoint
        int mid = (lowerBound+upperBound) / 2;  // sort low half
        recMergeSort(workSpace, lowerBound, mid);  // sort high half
        recMergeSort(workSpace, mid+1, upperBound);  // merge them
        merge(workSpace, lowerBound, mid+1, upperBound);
    }
}
```

Analyzing the Merge Sort Algorithm

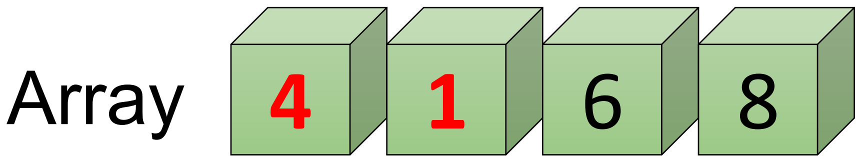
n	Merge Sort (ms)	Selection Sort (ms)
10,000	31	772
20,000	47	3,051
30,000	62	6,846
40,000	80	12,188
50,000	97	19,015
60,000	113	27,359



Merge Sort Timing (blue) versus Selection Sort (red)

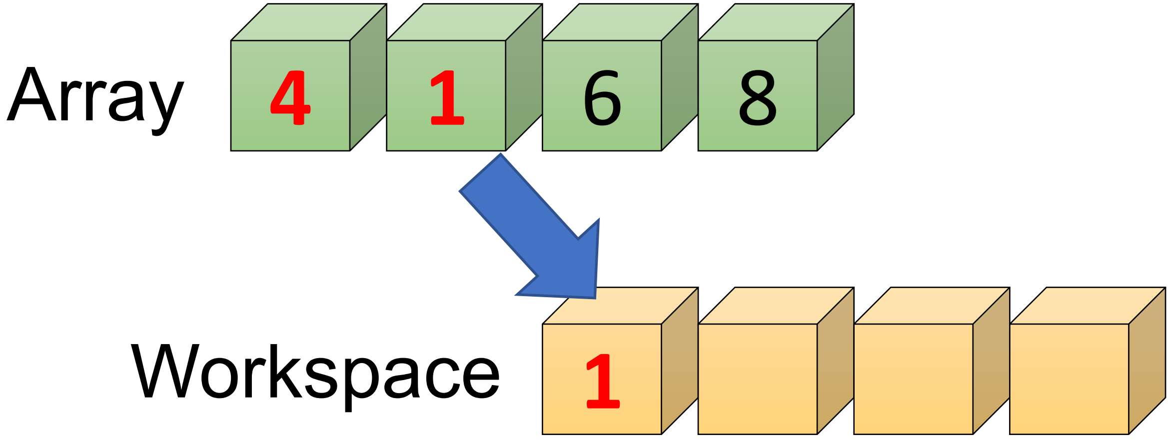
Copies and Comparisons

- A comparison:



  - In order to merge the first two cells we need to check is 4 bigger than 1?

- A copy:



Analyzing the Merge Sort Algorithm

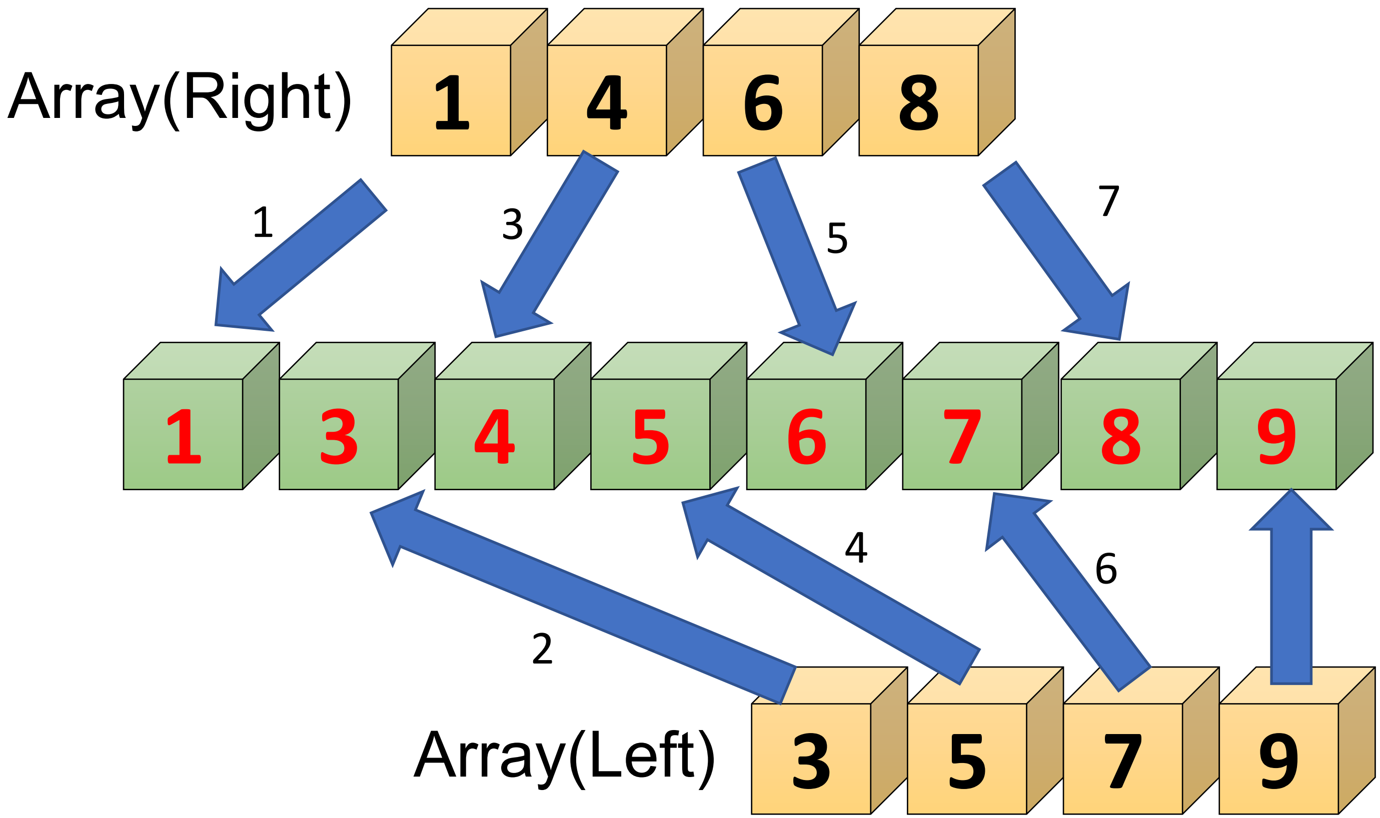
- In an array of size n, lets figure out the number of copies and comparisons needed
- Copies take longer than comparisons but the order of the algorithm is determined by **whichever of these has the highest order**
- Every time the array is split again, each element is going to end up being copied (see tree diagram a few slides back)
- The number of levels / splits required will be $\log_2 n$ since each step halves the search space (see Topic 3)
- Number of copies will be the number of levels multiplied by n since the full array is copied on each level
- Total number will be doubled because these need to be copied back into the array (copied from workspace back into original array)
- Number of copies is proportional to $\boldsymbol{n \log n}$

Number of Comparisons

- When merging, the maximum number of comparisons needed will be at most one less than the number of items being merged $n - 1$
- Minimum comparisons will be half the number of items being merged $\frac{n}{2}$
- There are still $log_2 n$ number of levels each involving a full merging of the array
- If the number of comparisons for each level is somewhere between $\frac{n}{2}$ and $n - 1$ then the total number of comparisons is $O(n \, log \, n)$
- Both copies and comparisons are $O(n \, log \, n)$ so algorithm is more efficient than insertion sort

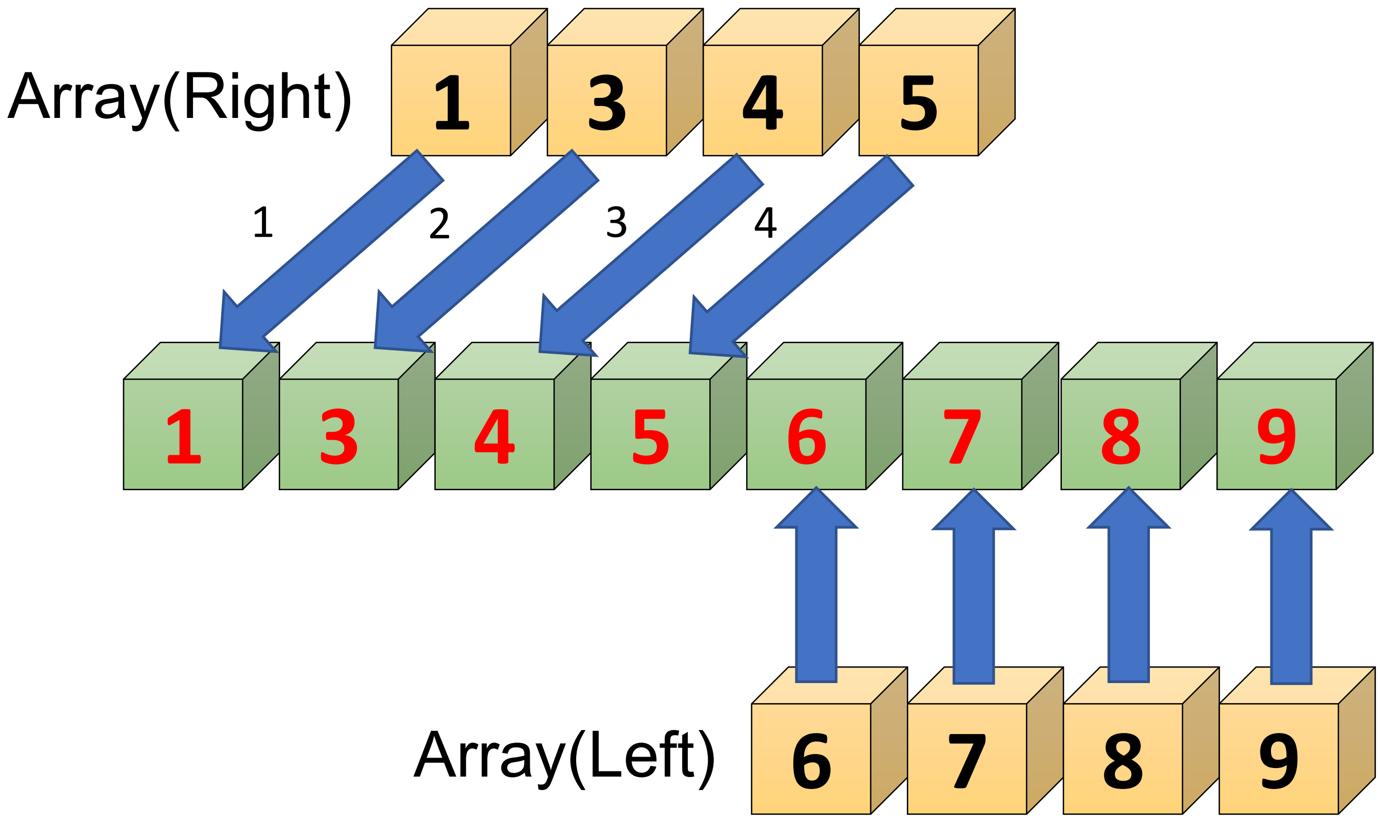
Worst-Case Scenario: $n - 1$

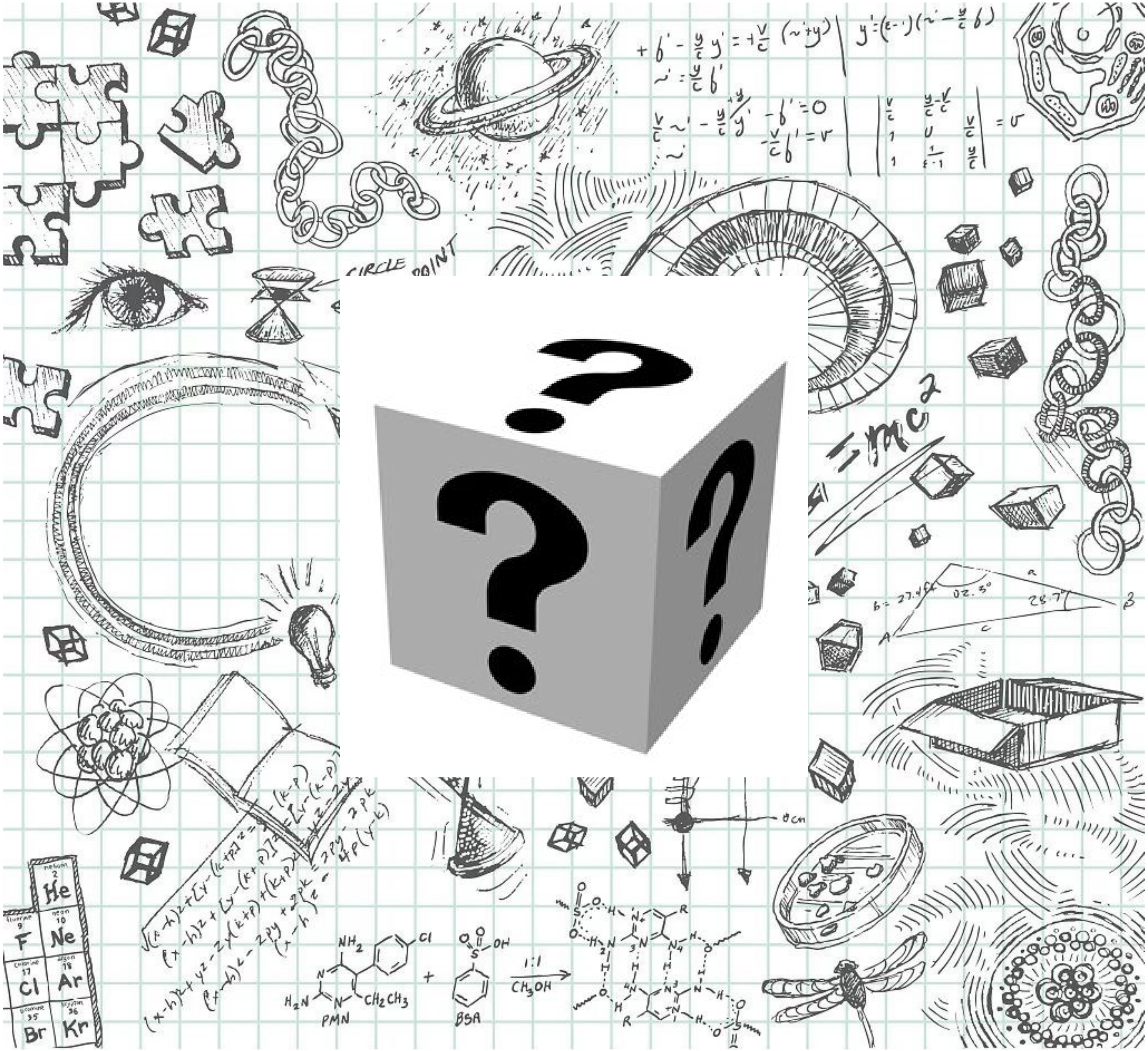
- Comparisons
    1. 1-3
    2. 3-4
    3. 4-5
    4. 5-6
    5. 6-7
    6. 7-8
    7. 8-9



Best-Case Scenario: $\frac{n}{2}$

- Comparisons
  1. 1-6
  2. 3-6
  3. 4-6
  4. 5-6




CIRCLE
POINT
He
F
Ne
Cl
Ar
Br
Kr
PMN
BSA