BIRZEIT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT FACULTY OF ENGINEERING AND TECHNOLOGY

COMP242

Data Structures And Algorithms in Java

Chapter 7
Sort

Instructor: Murad Njoum

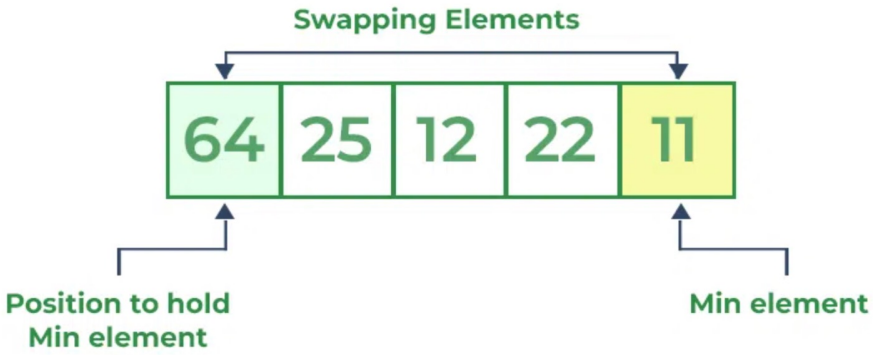
Chapter 7
Sort

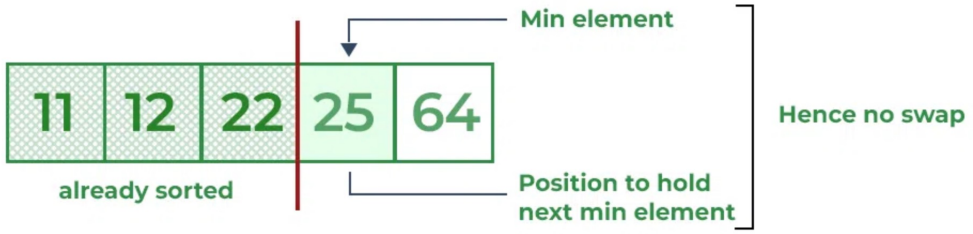
- Selection
- Bubble
- Radix/Bucket
- Heap Sort

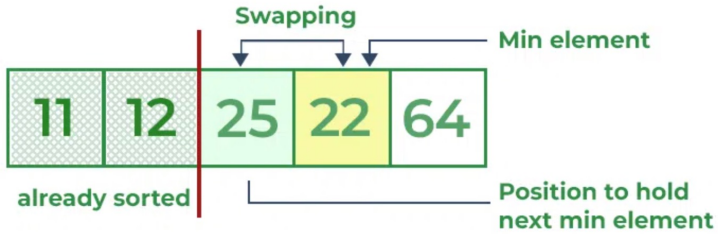
- Merge Sort
- Quick Sort
- insertion sort
- Shell sort
- External Sort

Selection sort

Selection sort *is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.*







```
void sort(int arr[])
    {
        int n = arr.length;

        // One by one move boundary of unsorted subarray
        for (int i = 0; i < n-1; i++)
        {
            // Find the minimum element in unsorted array
            int min_idx = i;
            for (int j = i+1; j < n; j++)
                if (arr[j] < arr[min_idx])
                    min_idx = j;

            // Swap the found minimum element with the first
            // element
            int temp = arr[min_idx];
            arr[min_idx] = arr[i];
            arr[i] = temp;
        }
    }
```

Complexity Analysis of Selection Sort

Time Complexity: The time complexity of Selection Sort is **$O(N^2)$** as there are two nested loops:

- One loop to select an element of Array one by one = O(N)
- Another loop to compare that element with every other Array element = O(N)
- Therefore overall complexity = O(N) * O(N) = O(N*N) = $O(N^2)$

Auxiliary Space: O(1) as the only extra memory used is for temporary variables while swapping two values in Array. The selection sort never makes more than O(N) swaps and can be useful when memory writing is costly.

Advantages of Selection Sort Algorithm

- Simple and easy to understand.
- Works well with small datasets.

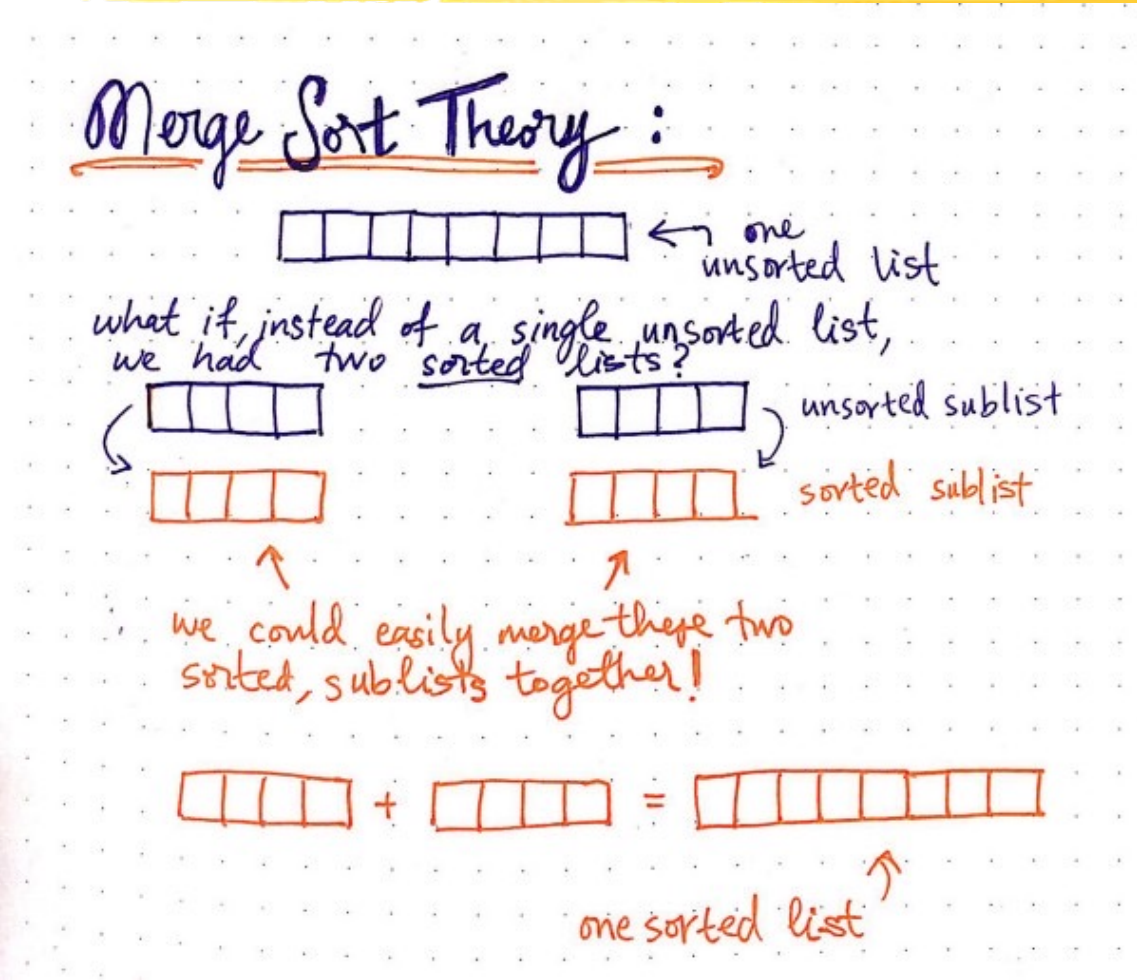
Disadvantages of the Selection Sort Algorithm

- Selection sort has a time complexity of O(n^2) in the worst and average case.
- Does not work well on large datasets.
- Does not preserve the relative order of items with equal keys which means it is not stable.

Divide and conquer algorithms (Merge Algorithm)

The basic idea behind merge sort is this: it tends to be a lot easier to sort ***two smaller, sorted lists rather than sorting a single large, unsorted one.***

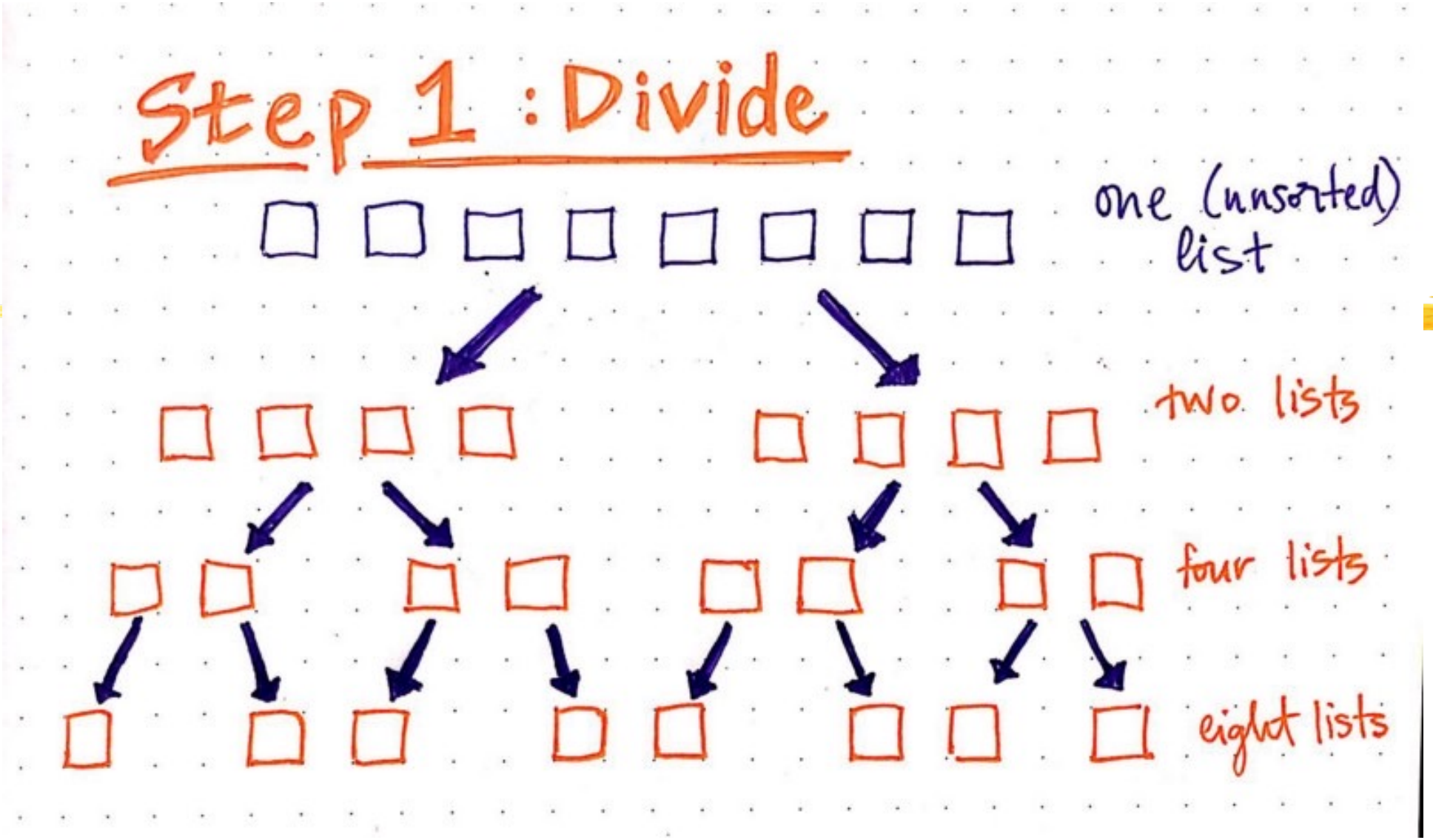
It's useful when data set is huge (in Terabytes) and memory is low (in Gega bytes)

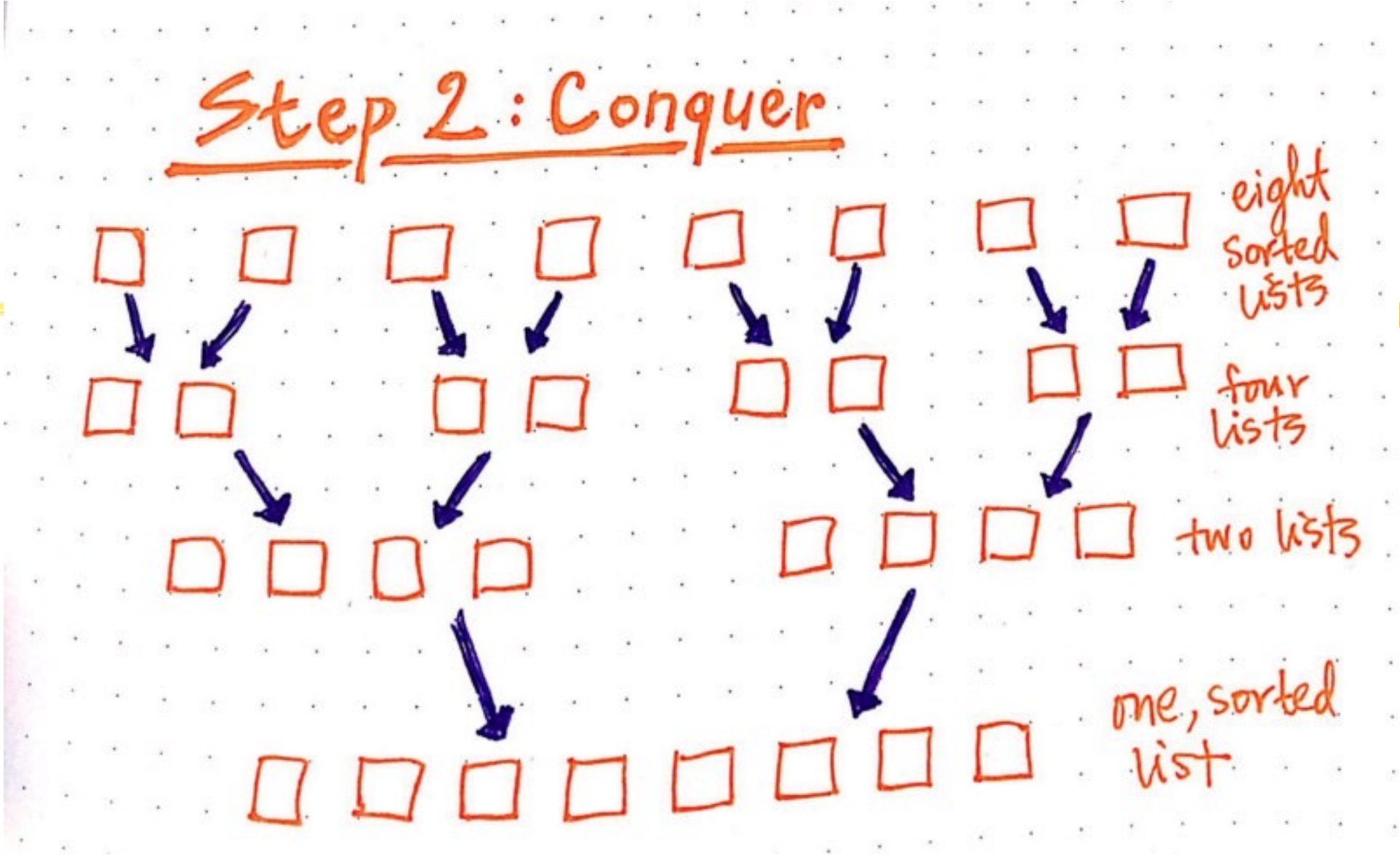


Instructor: Murad Njoum

Divide + Conquer!

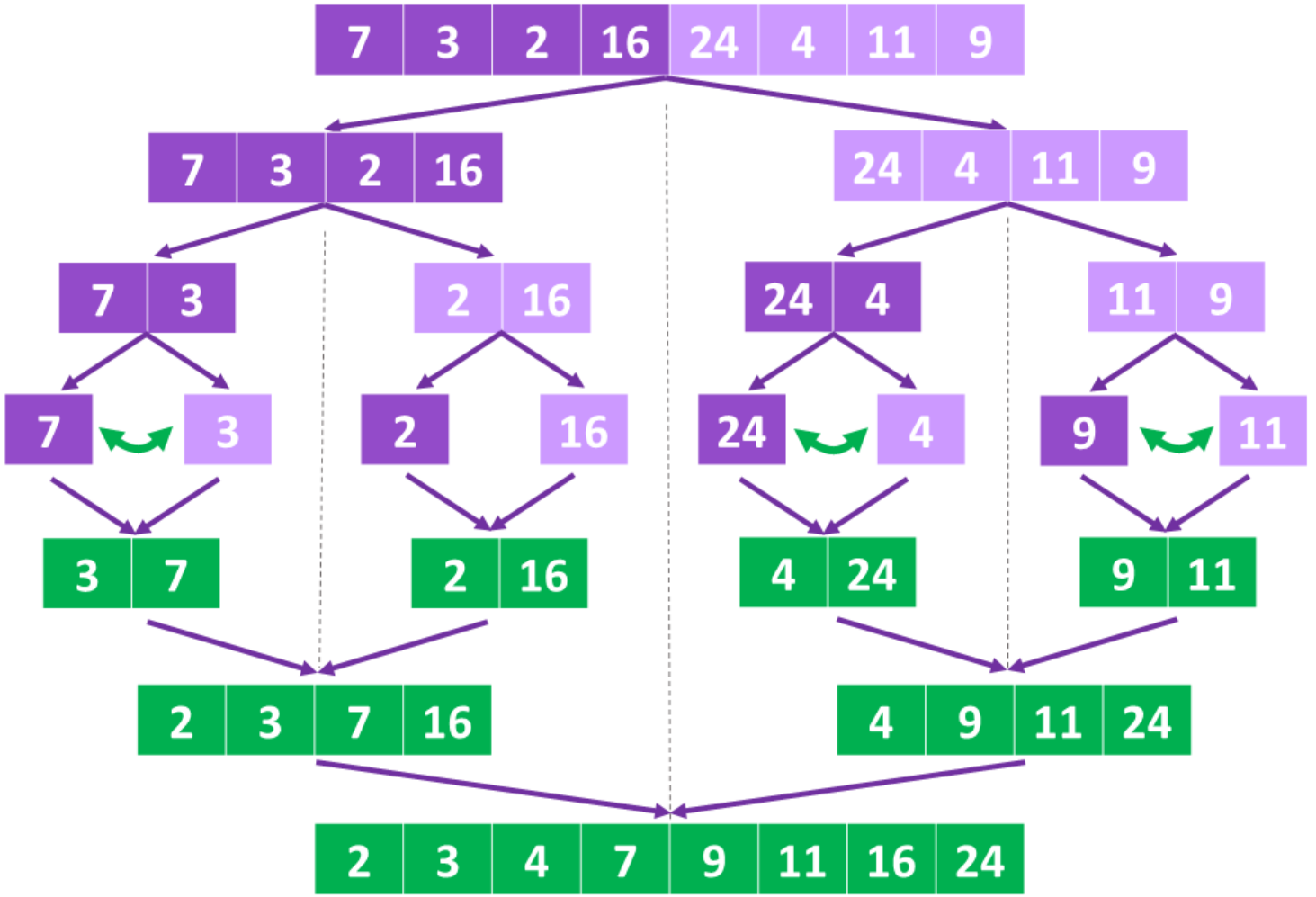
- A divide and conquer algorithm divides a problem into simpler versions of itself.
- By breaking down a problem into smaller parts, they become easier to solve. Usually, the solution for the smaller sub-problems can be applied to the larger, complicated one.
- Conquering the large problem using the same solution is what makes d+c recursive.


Step 1 : Divide
one (unsorted) list
two lists
four lists
eight lists


Step 2: Conquer
eight sorted lists
four lists
two lists
one, sorted list

Example 1

Merge Sort



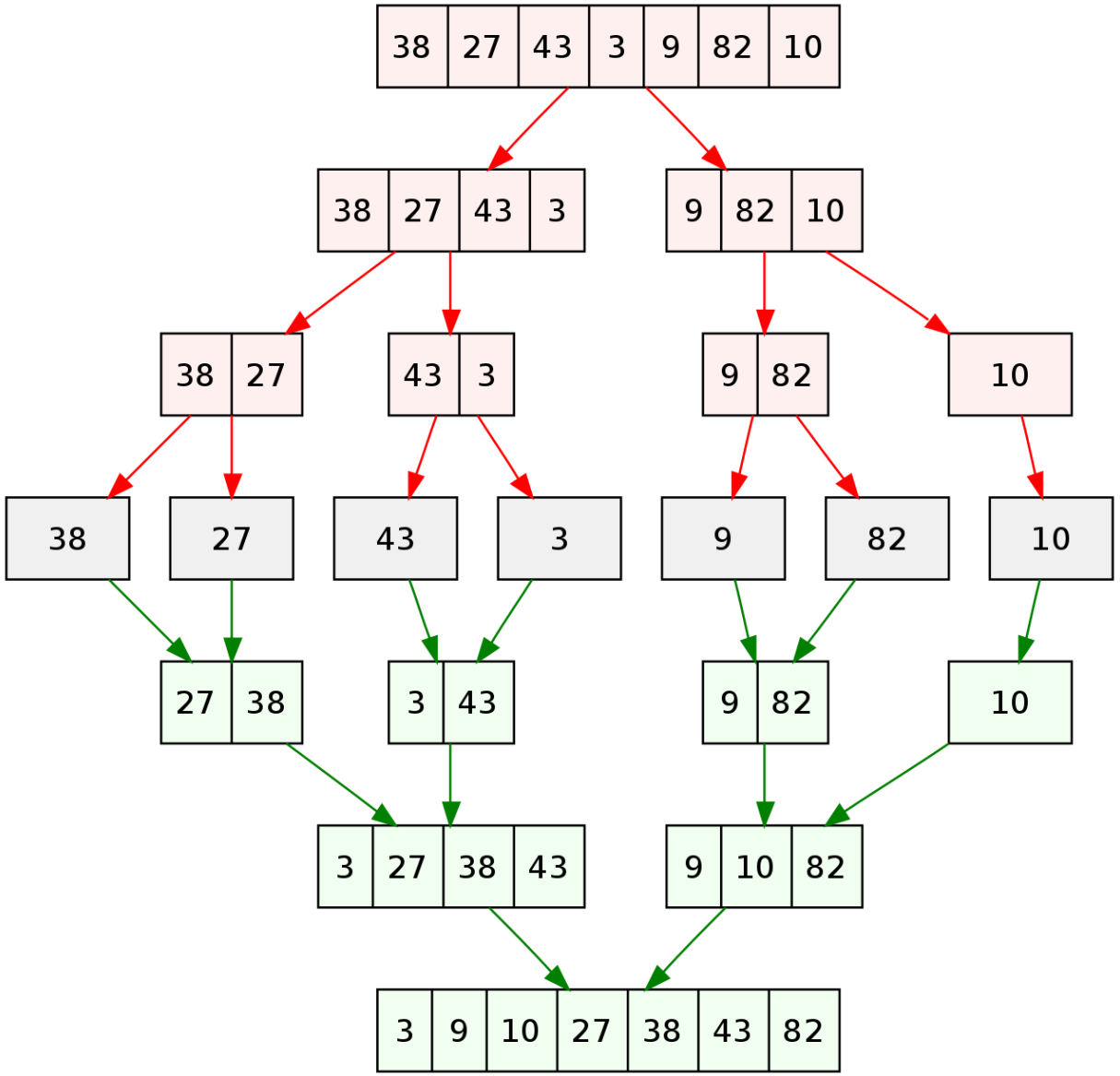
Step 1:
Split sub-lists in two until you reach pair of values.

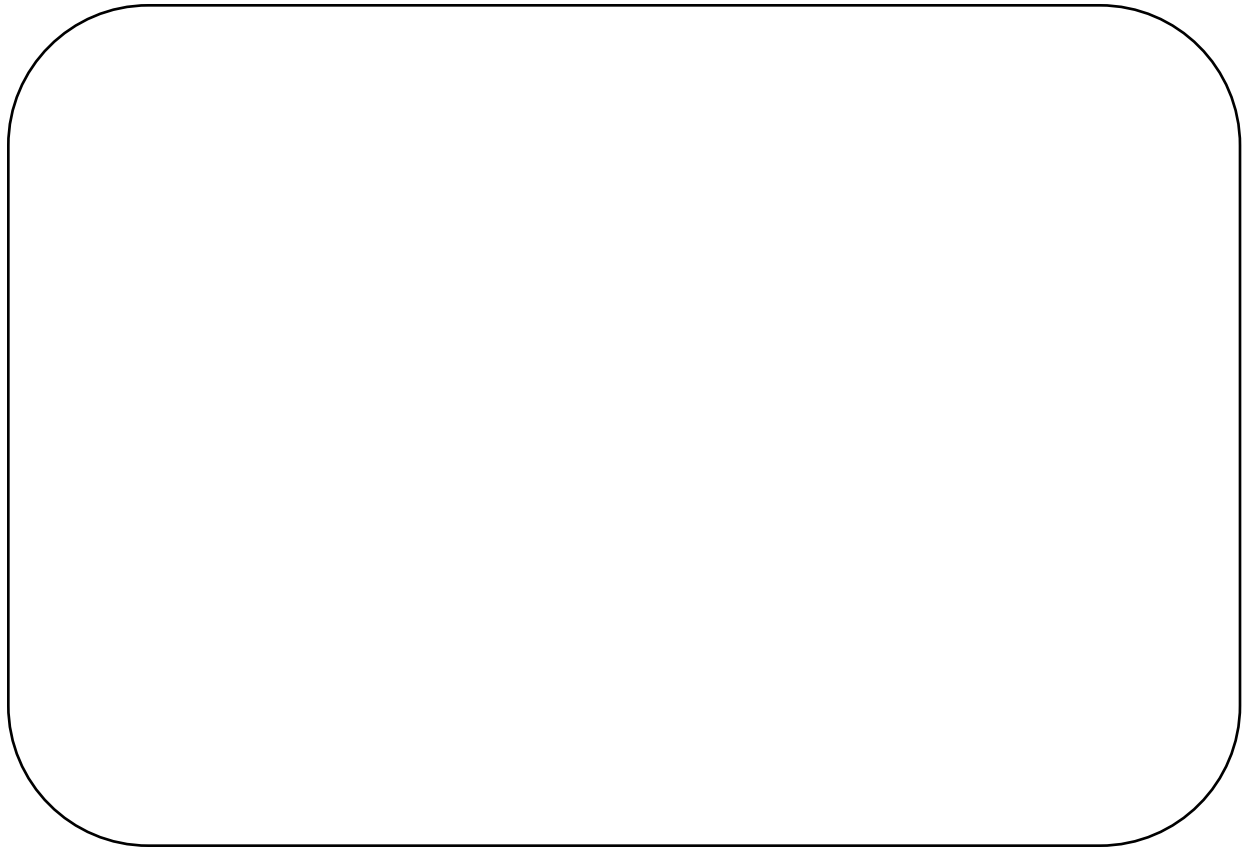
Step 3:
Sort/swap pair of values if needed.

Step 4:
Merge and sort sub-lists and repeat process till you merge to the full list.

Instructor: Murad Njoum

Example 2


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82



5	3	7	1	0	8	5

mergesort([5, 3, 7, 1, 0, 8, 5])

Algorithm

```
/* low is for left index and high is right
index of the  sub-array of array to be
sorted */
void mergeSort(int low, int high)
{
    if (low < high)
    {
            int mid = (low +high)/2;

        // Sort first and second halves
        mergeSort(low, mid);
        mergeSort(mid +1, high);

        merge(low, mid, high);
    }
}
```

Algorithm (Time Complixity)

```
/* low is for left index and high is right
index of the  sub-array of array to be
sorted */
void mergeSort(int low, int high)
{
    if (low < high)
    {
            int mid = (low +high)/2;

        // Sort first and second halves
        mergeSort(low, mid);
        mergeSort(mid +1, high);

        merge(low, mid, high);
    }
}
```

→Analysis of Merge Sort

$$T(n) = \begin{cases} a & n=1 \\ 2T(n/2)+Cn & n>1 \end{cases}$$

$T(n/2) = 2T(n/4) + Cn/2$

$T(n) = 2[2T(n/4) + Cn/2] + Cn$

$= 2^2 T(n/4) + 2Cn$

$T(n) = 2^3 T(n/2^3) + 3Cn$

. . .

$T(n) = 2^K T(n/2^K) + KCn$

Let $2^K = n \rightarrow k = \log n$

$T(n) = KT(1) + Cn \log n$

$T(n) = O(n \log n)$

```
void merge(int low, int mid, int high)
{
   int i, j, k;
   i=low; //for Another Array copied
    j=low;
   k=mid+1;


   while (j < =mid && k < =high)
   {
      if (A[j] <= A[k])
      {
         B[i] = A[j];
         j++;
      }
      else
      {
        B[i] = A[k];
        k++;
      }
     i++;
   }
```

```
    if(j < =mid) // copy all remines
elements  to Array
     {
          for(k=j; k<=mid;k++, i++)
           B[i] = A[k];
                }
  else{   // copy all remines elements  to
Array
           for(j=k; j<=high;j++,i++)
             B[i] = A[j];
                  }
     }


    /* Copy the remaining elements of B[],
back to A[]*/
  for(i=low; j<=high; i++)
             A[i] = B[i];
}
```

QuickSort

Like Merge Sort, QuickSort is a **Divide and Conquer** algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick first element as pivot.
- Always pick last element as pivot
- Pick a random element as pivot.
- Pick median as pivot.

Quicksort

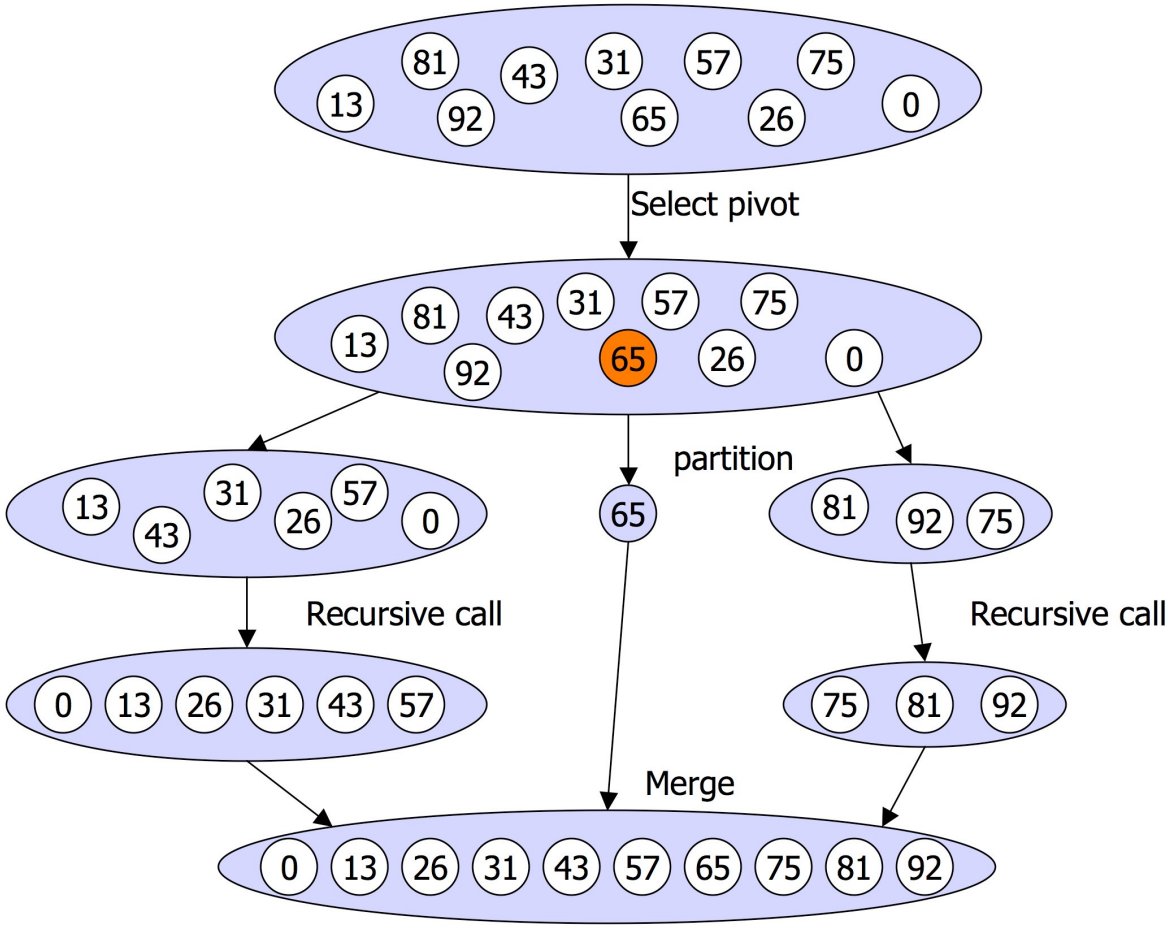
- Fastest known sorting algorithm in practice
  - Caveats: not stable
  - Vulnerable to certain attacks

- Average case complexity → $O(N \log N)$

- Worst-case complexity → $O(N^2)$
  - Rarely happens, if coded correctly

Quick Sort

Quicksort example



Picking the Pivot

- How would you pick one?
- Strategy 1: Pick the first element in `S`
  - Works only if input is random
  - What if input `S` is sorted, or even mostly sorted?
    - All the remaining elements would go into either `S1` or `S2`!
    - Terrible performance!
  - Why worry about sorted input?
    - Remember → Quicksort is recursive, so sub-problems could be sorted
    - Plus mostly sorted input is quite frequent

Picking the Pivot (contd.)

- Strategy 2: Pick the pivot randomly
  - Would usually work well, even for mostly sorted input
  - Unless the random number generator is not quite random!
  - Plus random number generation is an expensive operation

Picking the Pivot (contd.)

- Strategy 3: Median-of-three Partitioning
  - *Ideally*, the pivot should be the median of input array **s**
    - Median = element in the middle of the sorted sequence
  - Would divide the input into two almost equal partitions
  - Unfortunately, its hard to calculate median quickly, without sorting first!
  - So find the approximate median
    - Pivot = median of the left-most, right-most and center element of the array **s**
    - Solves the problem of sorted input

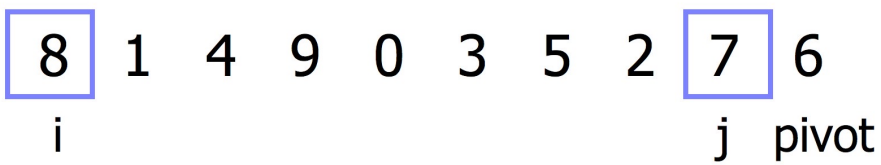
Picking the Pivot (contd.)

- Example: Median-of-three Partitioning
  - Let input `S = {6, 1, 4, 9, 0, 3, 5, 2, 7, 8}`
  - `left=0` and `S[left] = 6`
  - `right=9` and `S[right] = 8`
  - `center = (left+right)/2 = 4` and `S[center] = 0`
  - Pivot
    - = Median of `S[left]`, `S[right]`, and `S[center]`
    - = median of 6, 8, and 0
    - `= S[left] = 6`

Partitioning Algorithm

- Original input : `S = {6, 1, 4, 9, 0, 3, 5, 2, 7, 8}`
- Get the pivot out of the way by swapping it with the last element

- Have two 'iterators' – `i` and `j`
  - `i` starts at first element and moves forward
  - `j` starts at last element and moves backwards

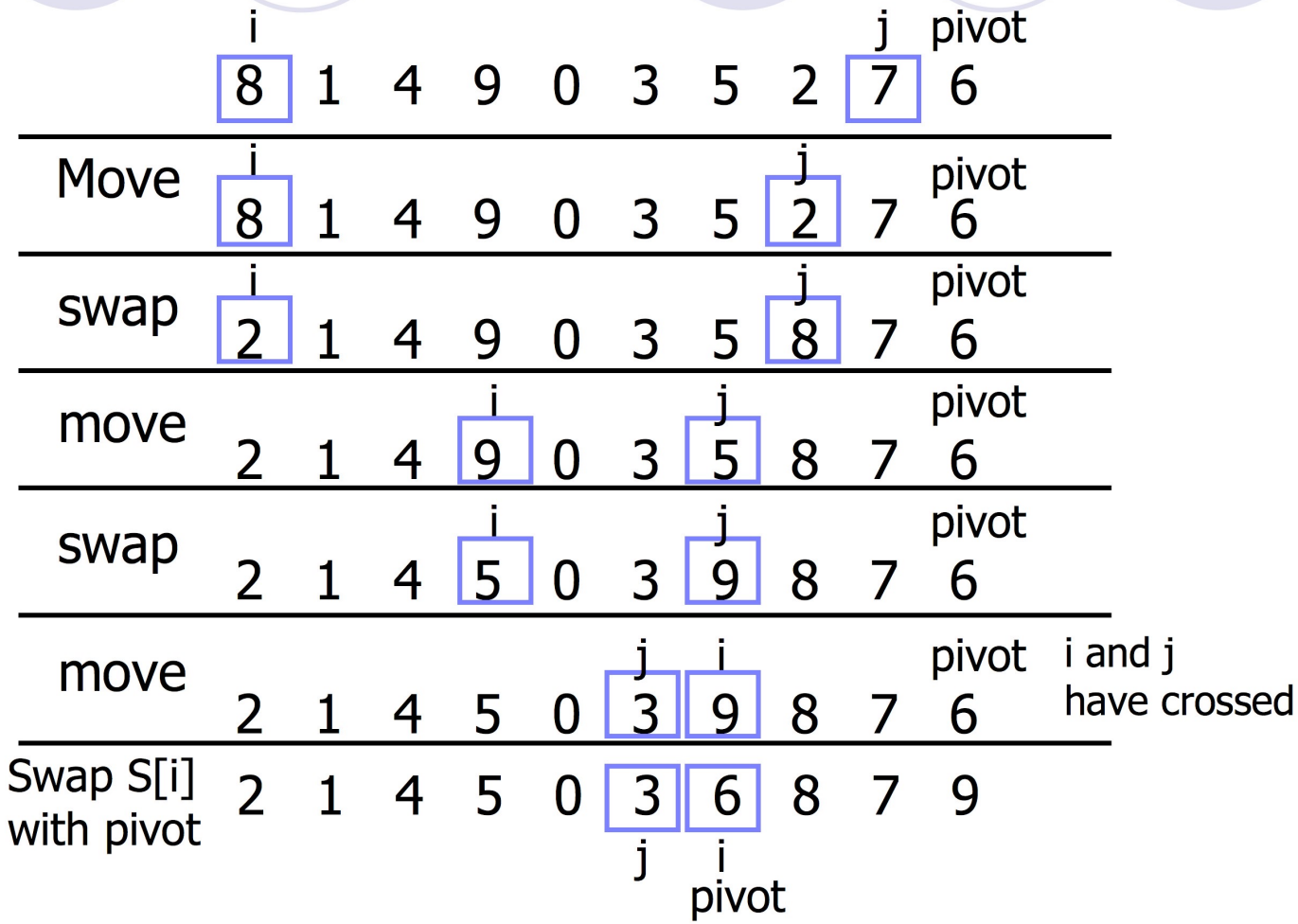


Partitioning Algorithm (contd.)

- **`While (i < j)`**
  - Move **`i`** to the right till we find a number greater than **`pivot`**
  - Move **`j`** to the left till we find a number smaller than **`pivot`**
  - **`If (i < j) swap(S[i], S[j])`**
  - (The effect is to push larger elements to the right and smaller elements to the left)

Swap the **`pivot`** with **`S[i]`**

Partitioning Algorithm Illustrated



For instance, with input 8, 1, 4, 9, 6, 3, 5, 2, 7, 0 the left element is 8, the right element is 0, and the center (in position (*left* + *right*)/2)
element is 6. Thus, the pivot would be *v* = 6.

Median of :

8, 1, 4, 9, 6, 3, 5, 2, 7, 0

center = (*left* + *right*)/2)
=[0+9]/2=4 , median is

Is left(8) > center (6) , swap them

6, 1, 4, 9, 8, 3, 5, 2, 7, 0

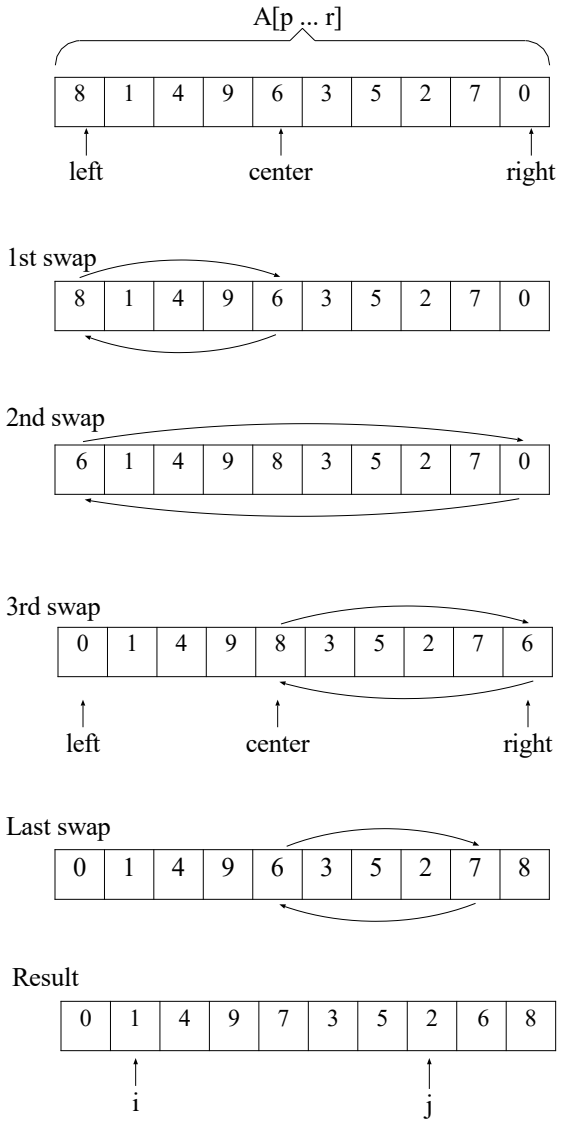
Is left(6) > right (0) , swap them

0, 1, 4, 9, 8, 3, 5, 2, 7, 6

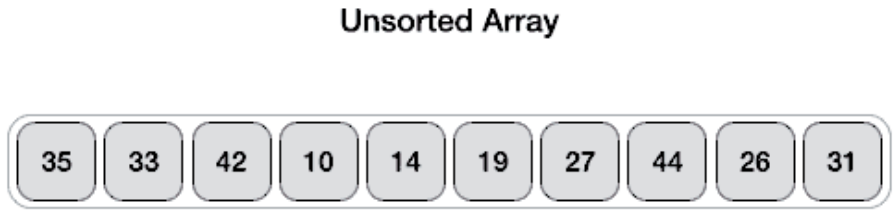
Is center(8) > right (6) , swap them

0, 1, 4, 9, 6, 3, 5, 2, 7, 8

0, 1, 4, 9, 8, 3, 5, 2, 7, 6



```
void Q_sort(int A[], int left, int right)
{
int i, j, pivot;
if ( left < right)
{
pivot = median3(A, left, right);
  i = left;
  j = right -1;
for(;;)   //while(i<j)  omit else, break
{
    while( A[i] < pivot){++i;}
    while( A[j] > pivot){--j;}
if ( i < j)
  exchange (A, i , j);
    else
   break;
  }
    exchange(A, i, right ); //swap occur between i and pivot
    Q_sort(A, left, i-1);
    Q_sort(A, i+1, right);//i not included because index i is the pivot,(all elements in indexes
are less or more than pivot)     }
}
```



```
int median3(int A[], int left, int right)
{
    int center = ( left + right )/2;
if ( A[left] > A[center])
        exchange(A, left, center);
if ( A[left] > A[right])
        exchange(A, left, right);
if ( A[center] > A[right])
      exchange(A, center, right); //rearrange elements

exchange(A, center, right ); //swap median pivat with most right
elements in array
return A[right ]; //return the pivot
}
```

Analysis of QuickSort

Time taken by QuickSort in general can be written as following.

`T(n) = T(k) + T(n-k-1) + (n)`

The first two terms are for two recursive calls, the last term is for the **partition process**.

k is the number of elements which are **smaller than pivot.**

The time taken by QuickSort depends upon the input array and partition strategy.

Following are three cases.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

`T(n) = T(0) + T(n-1) +(n) which is equivalent to T(n) = T(n-1) + (n)`

The solution of above recurrence is **$O(n^2)$.**

Best Case: The best case occurs when the partition process always picks the **middle element** as pivot. Following is recurrence for best case.

`T(n) = 2T(n/2) + (n)`

The solution of above recurrence is **O(nLogn).**

→ Analysis of Quick Sort

- Worst case Analysis

$T(n) = T(i) + T(n - i - 1) + Cn$

$T(n) = T(n-1) + Cn \qquad n>1$

$T(n-1) = T(n-2) + C(n-1)$

$T(n-2) = T(n-3) + C(n-2)$

. . .

. . .

$T(2) = T(1) + C(2)$

$$T(n) = T(1) + C \sum_{i=2}^{n} i = O(n^2)$$

- Best case Analysis

$T(n) = 2\,T(n/2) + Cn$

. . .

. . .

$T(n) = O(n \log n)$

- Average case Analysis

$T(n) = T(i) + T(n-i-1) + Cn$

Average $T(i) = 1/n \sum_{j=0}^{n-1} T(j)$

$T(n) = 2/n [\sum_{j=0}^{n-1} T(j)] + Cn$

$nT(n) = 2 [\sum_{j=0}^{n-1} T(j)] + Cn^2$ ....(1)

$(n-1) T(n) = 2 [\sum_{j=0}^{n-2} T(j)] + C(n-1)^2$ ....(2)

(1) – (2)

$nT(n) - (n-1) T(n-1) = 2 T(n-1) + 2Cn - C$

$nT(n) = 2T(n-1) + (n-1) T(n-1) + 2Cn$

$nT(n) = (n+1)T(n-1) + 2Cn$ $1/n(n+1)$

$T(n)/(n+1) = T(n-1)/n + 2C(n+1)$

$T(n-1)/n = T(n-2)/(n-1) + 2C/n$

. . .

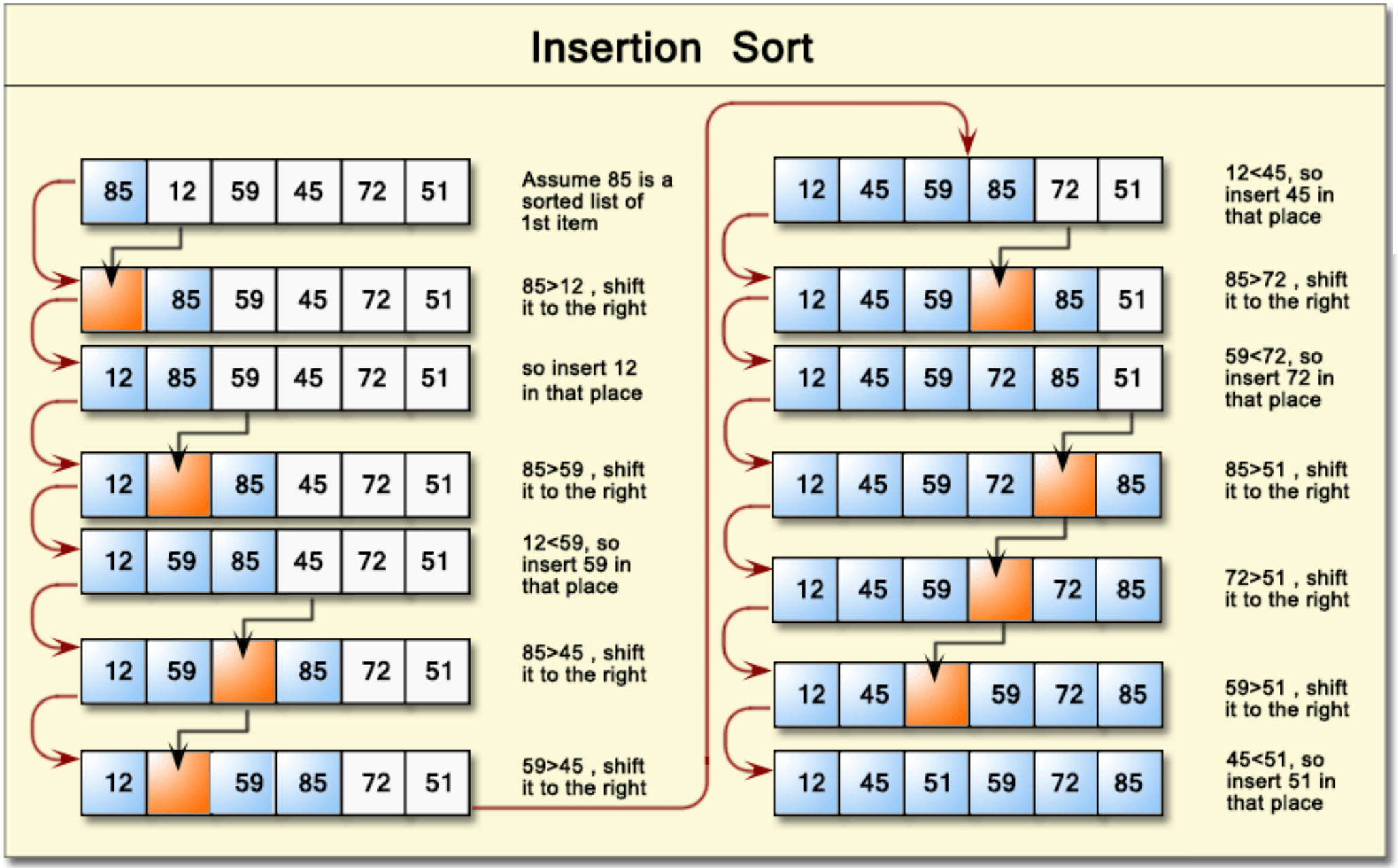
. . .

$T(n)/n+1 = C \sum_{i=1}^{n} 1/i$

$T(n) = (n+1) C \sum_{i=1}^{n} 1/i$

$T(n) = O(n \log n)$

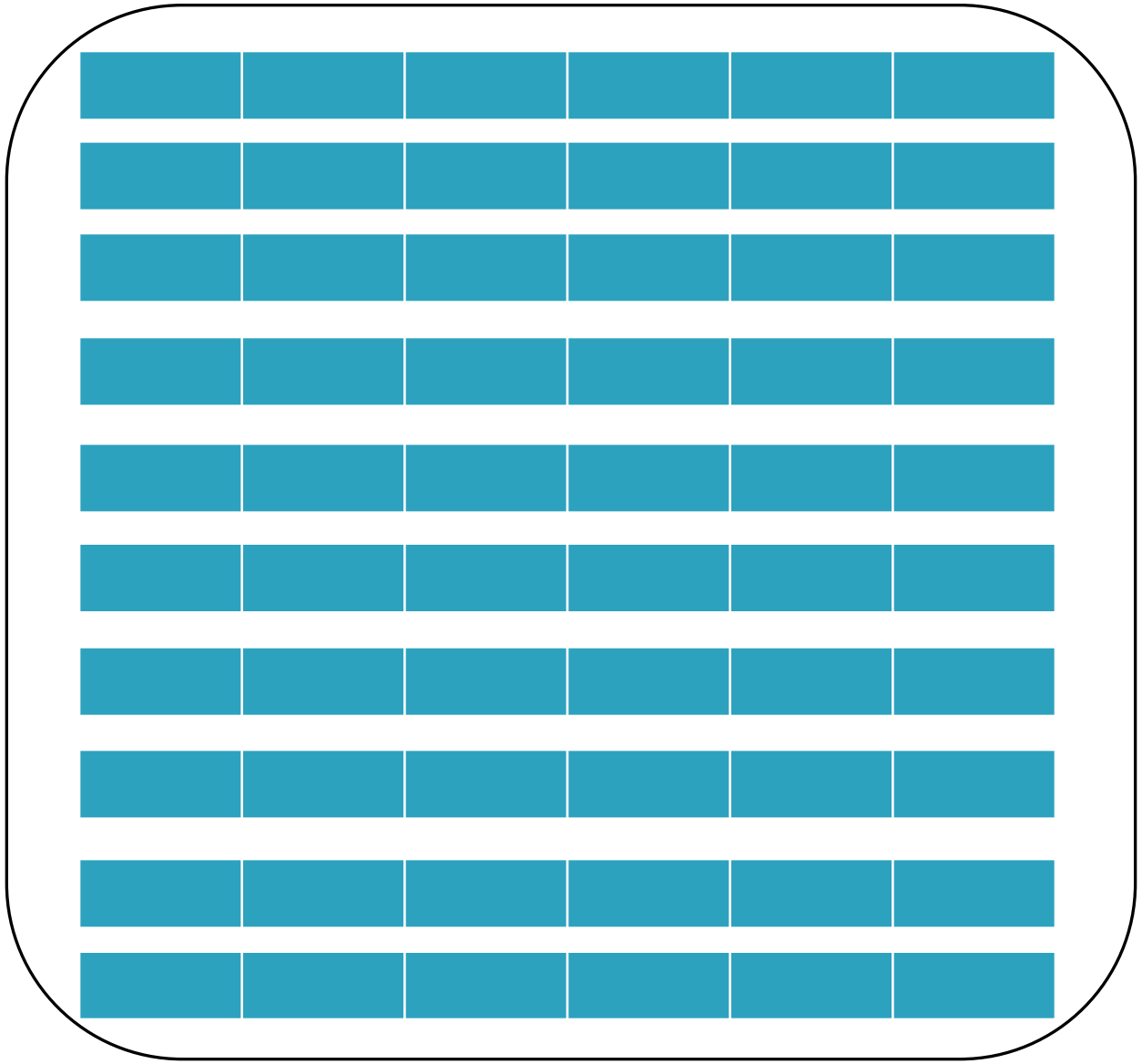
Insertion Sort: Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.



Exercise:

5	4	10	1	6	2

Solution in class At board



temp

```
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
   int i, temp, j;
   for (i = 1; i < n; i++) {
      temp = arr[i];
      j = i - 1;

      /* Move elements of arr[0..i-1], that are  greater than key, to one
        position ahead of their current position */
      while (j >= 0 && arr[j] > temp) {
         arr[j + 1] = arr[j];
         j --;
      }
      arr[j + 1] = temp;
   }
}
```

Time Complexity: $O(n^2)$

Auxiliary Space: O(1)

Boundary Cases: Insertion sort takes maximum time if elements are sorted in **reverse order**. And it takes **minimum time (O(n)) when elements are already sorted.**

Uses:

- Insertion sort is used when number of elements is small.
- It can also be **helpful when the input array is almost sorted,**
- **only a few elements are misplaced in a complete big array.**

Shell Sort

Shell Sort is mainly a variation of Insertion Sort.

In insertion sort, **we move elements only one position ahead**. Many movements are involved when an element has to be moved far ahead.

The idea of **shell Sort is to allow the exchange of far items**. In shell Sort, we make the array h-sorted for a significant value of h.

We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sub-lists of every h'th element are sorted.

Shell Sort:

23	29	15	19	31	7	9	5	2

Solution in class At board

Suppose we have an array like this

15	19	23	29	31	7	9	5	2

It's case of insertion sort ?

How many shifts we need to move 7 to correct postion?

5 shifts

shell techniques:

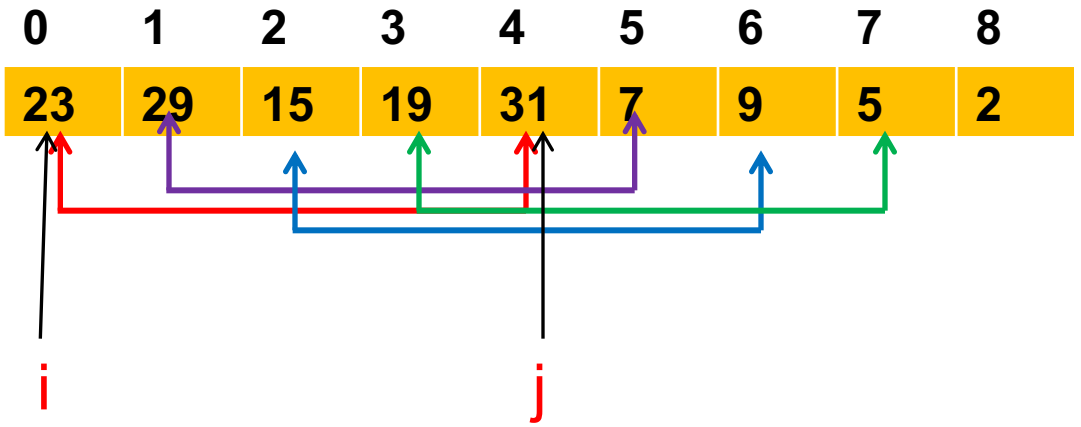
What if we move 7 to first position in just one movement.

This is called GAP/increment

We use distincit elemenst, not near elements

efficenciy of algoritm depends on gap

gap =5 ,3,1, it could be any gap,
we use gap=n/2

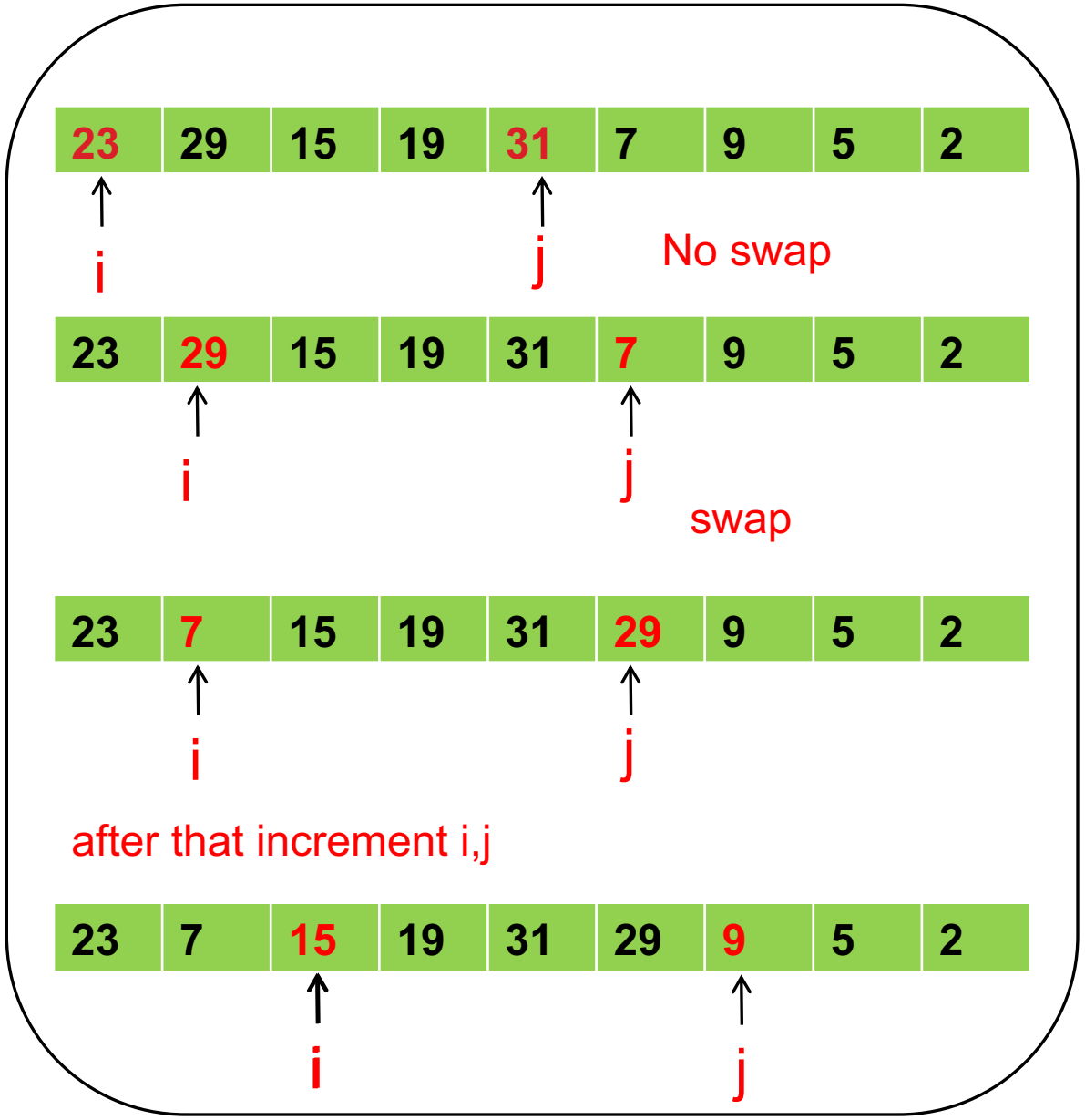


compare a[i] ,a[j] , if (a[i]>a[j]) , then swap

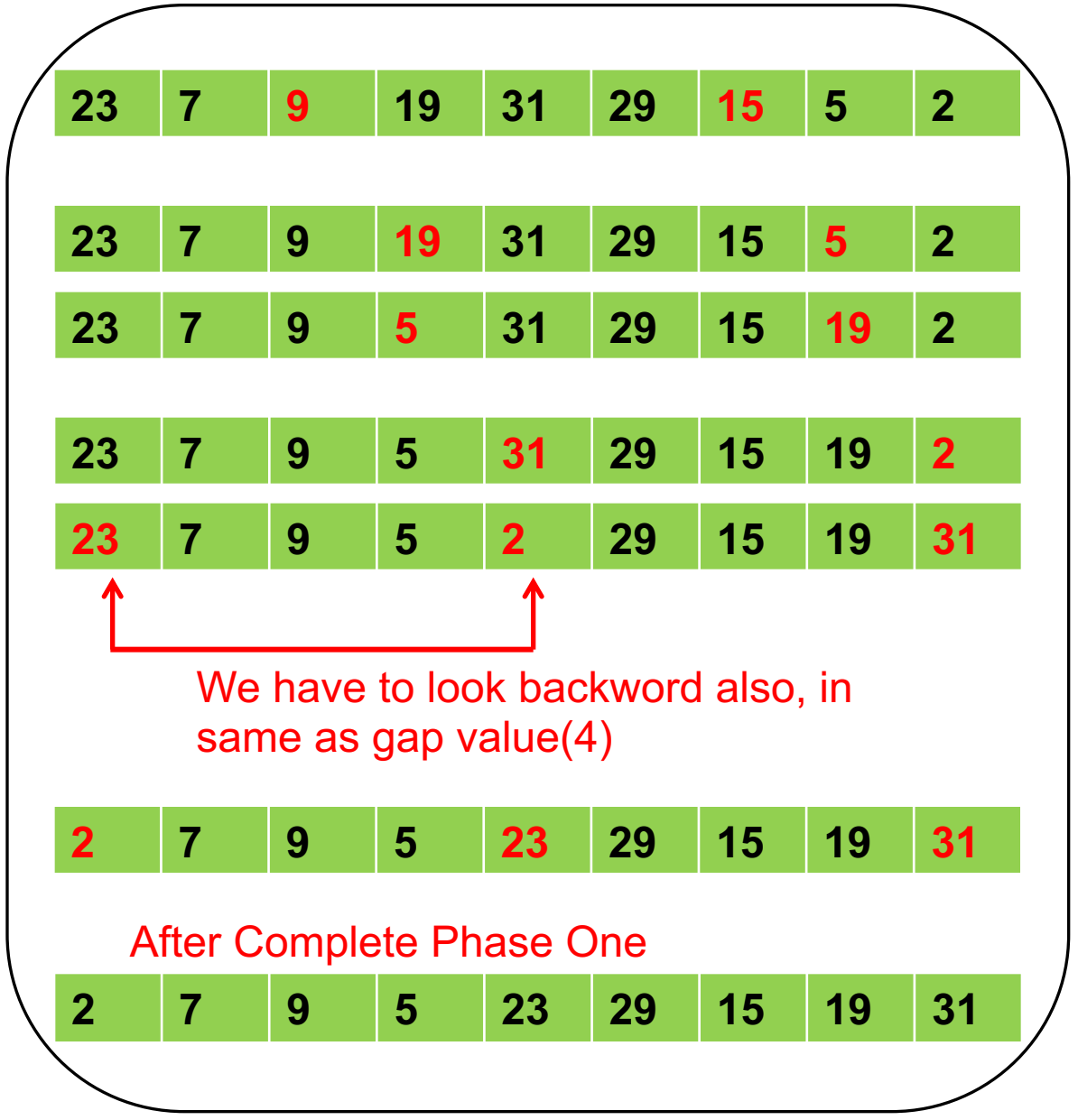
i++, j++

Shell Sort:

23	29	15	19	31	7	9	5	2

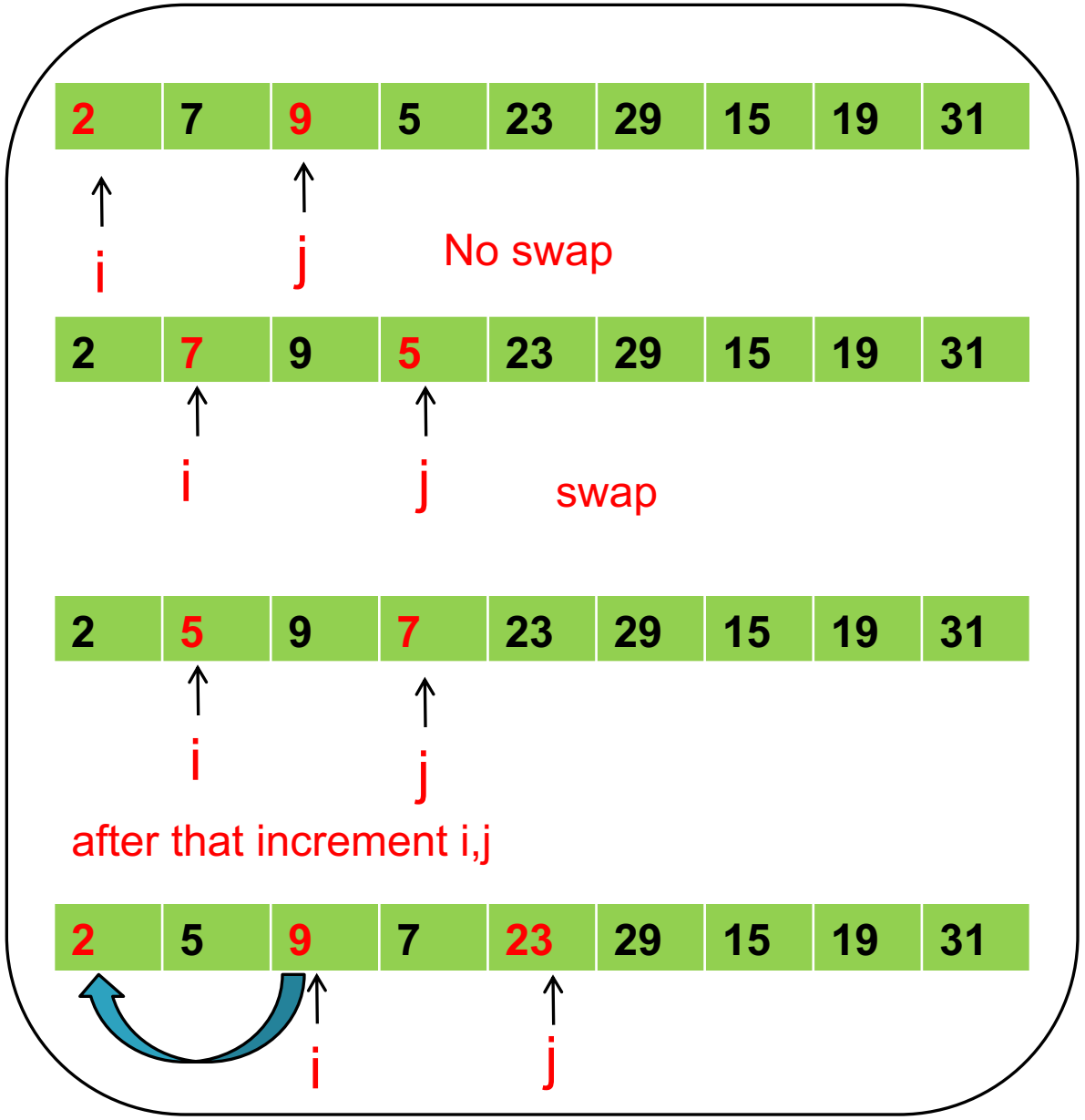


Solution in class At board

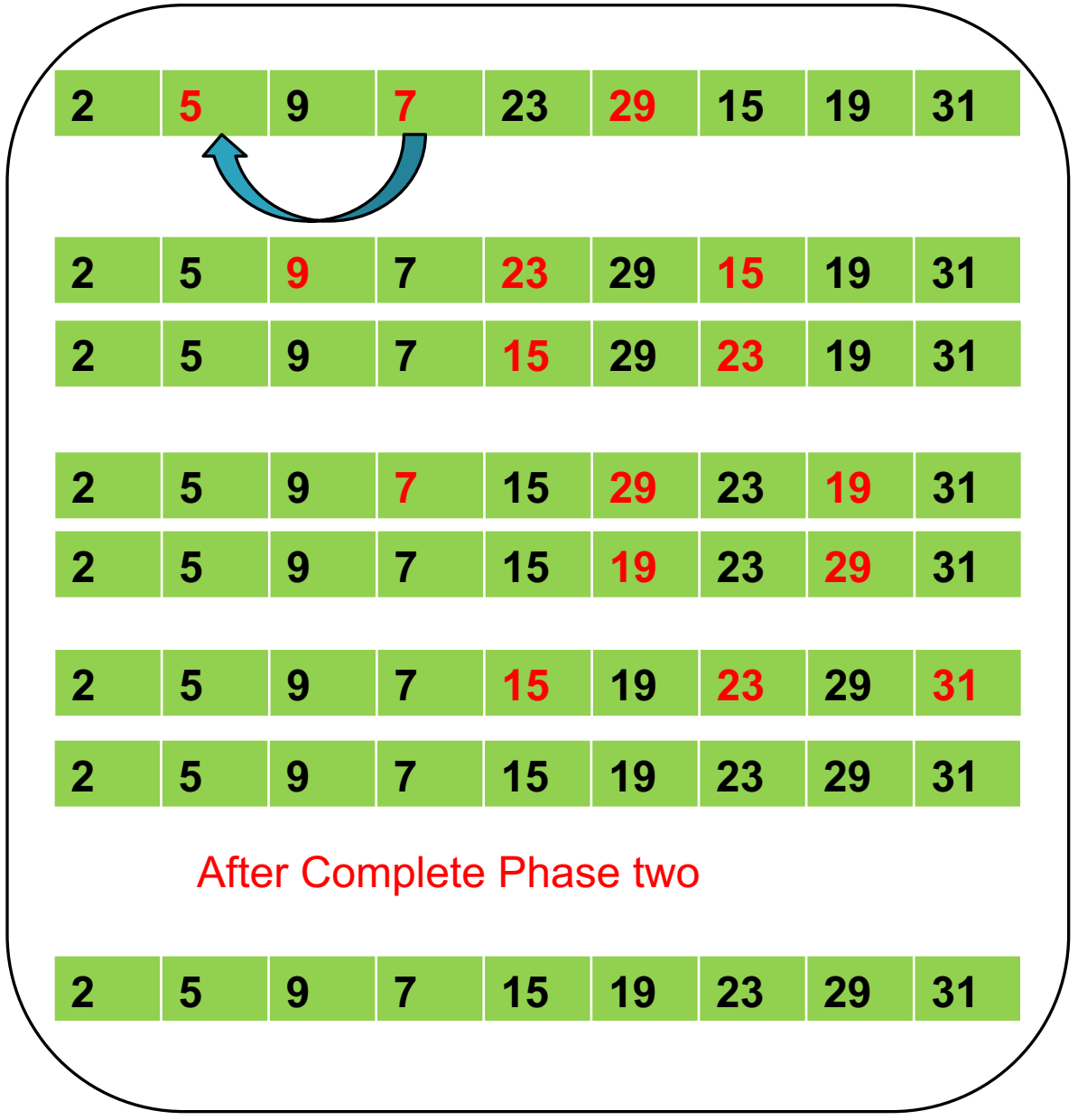


Shell Sort:

Gap= 4/2=2

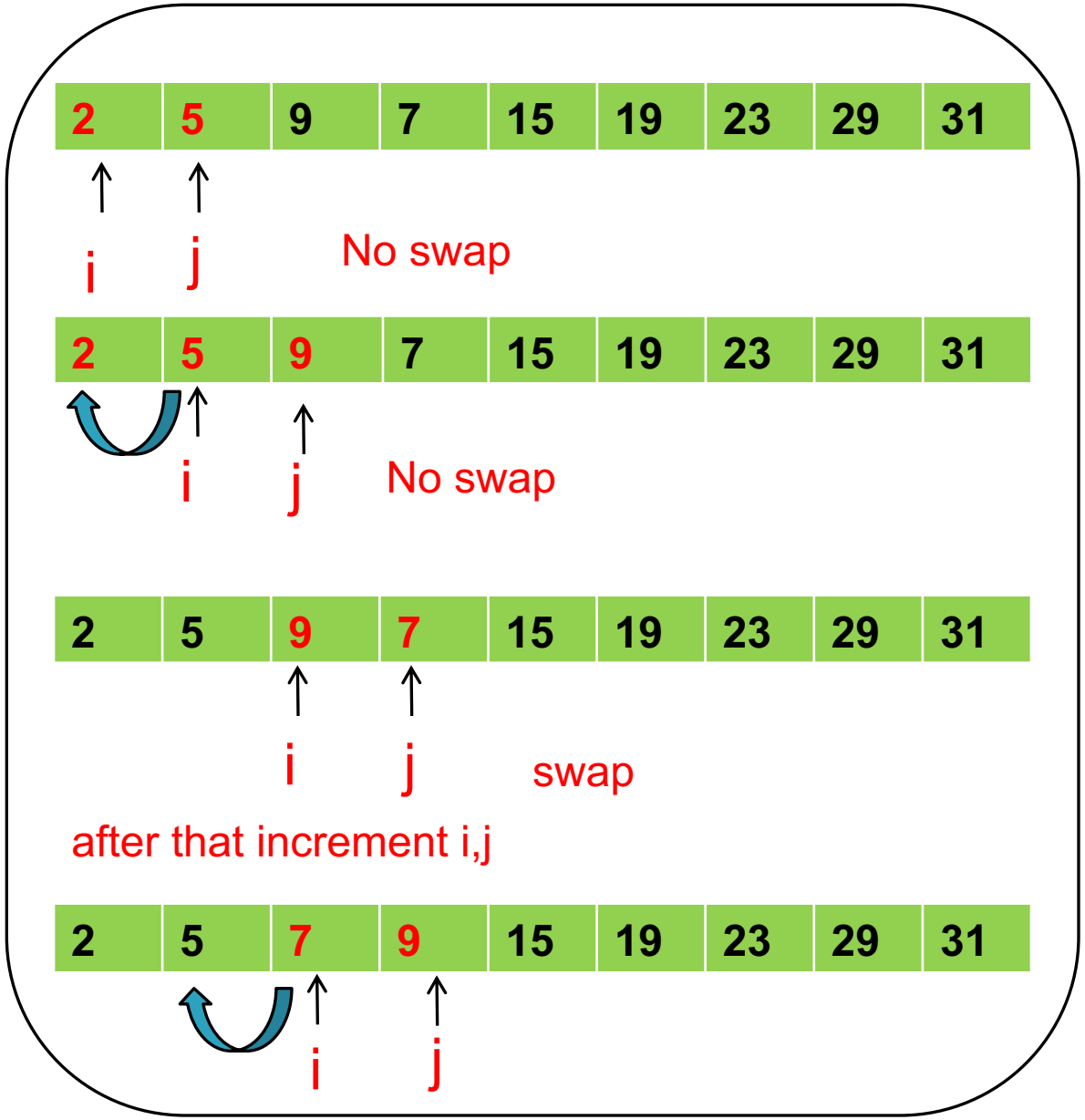


Solution in class At board

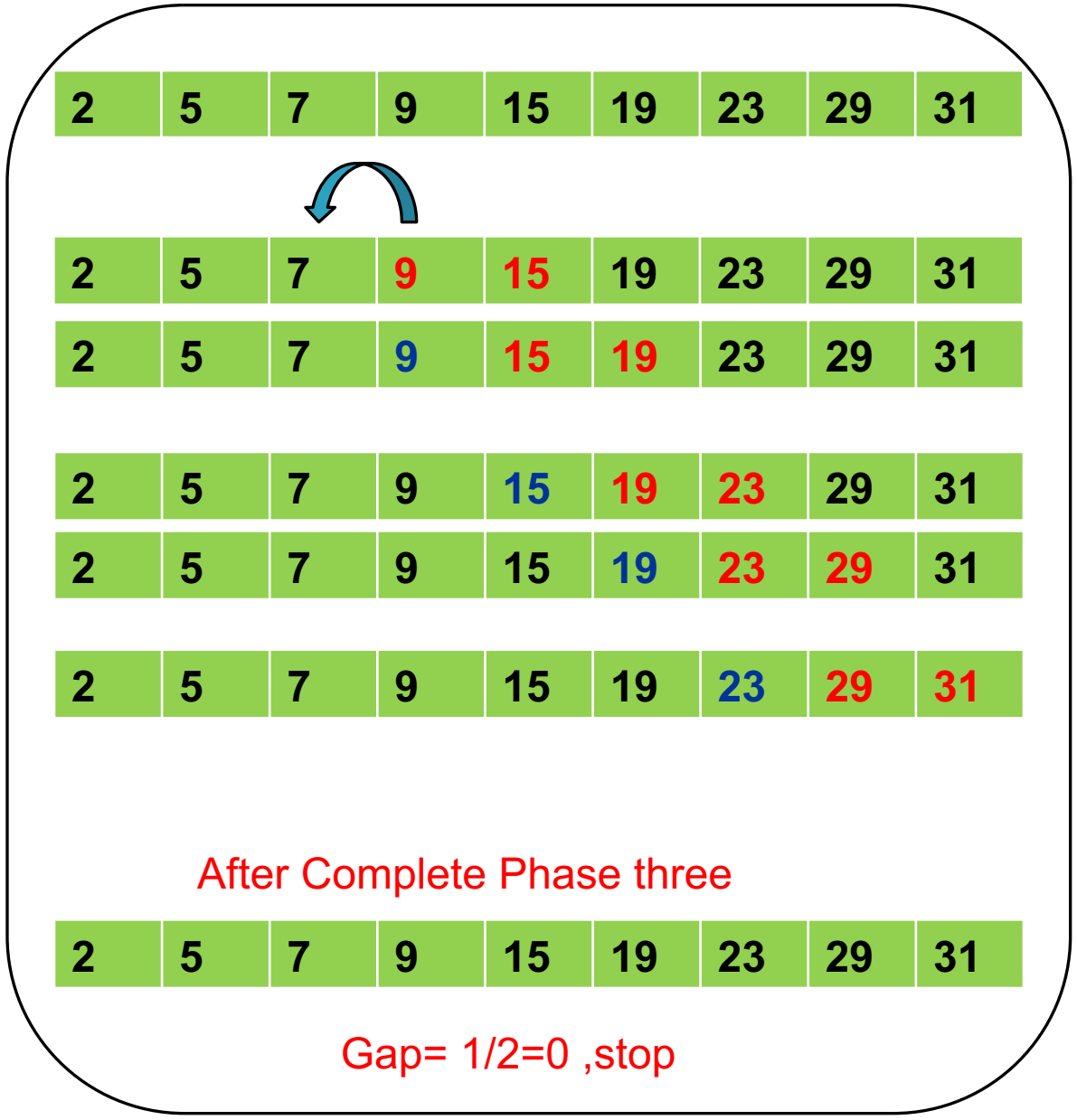


Shell Sort:

Gap= 2/2=1



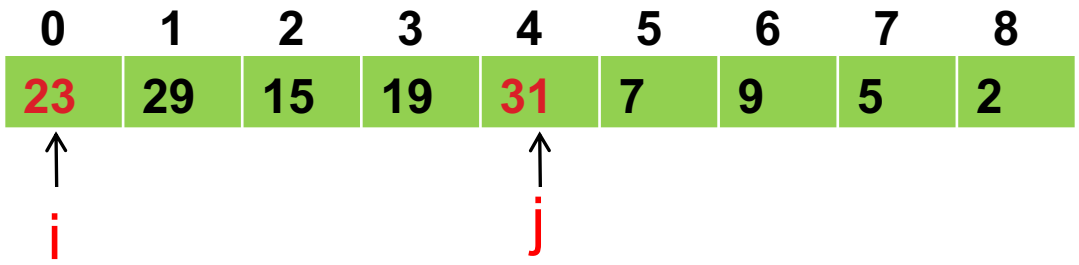
Solution in class At board



Shell sort

```
void Shellsort( ElementType A[ ], int N )
{
    int i, j, Increment;
    ElementType Tmp;

    for( gap = N / 2; gap > 0; gap /= 2 )
        for( j = gap ; j < N; j++ )
        {
            Tmp  ;
            for( i = j- gap; i >=0 ;i -= gap ) //test backword
                if(  A[ i ]< A[ i + gap ] )  //test for swap
                    {temp = A[ i + gap ];
                     A[ i + gap ]=A[i];
                     A[ i ] = Tmp;
                     }
                else
                    break;
             }
}
```



gap=4

j=4, i = 4-4=0 ===>break

j=5, i = 5-4=1, ==>swap
i=i-gap=1-4=-3 , condition is false

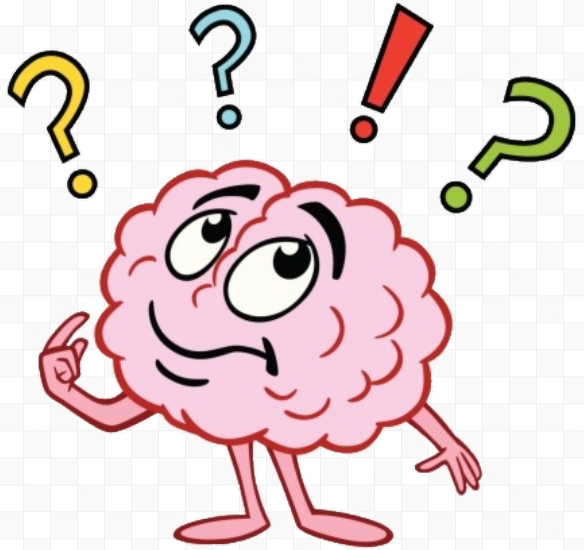
...

j=8,i=8-4=4.....if it true then swap
i=i-gap=4-4=0 ...if it true then swap

Time Complexity: Time complexity of above implementation of **shellsort** is **$O(n^2)$** In the above implementation gap is reduce by half in every iteration. There are many other ways to reduce gap which lead to better time complexity.

Suppose we have 5 GB of data using only 1 GB of RAM , what is the best sorting algoritm could you use?

External Sorting



Solution in class At board (We will Back later)

External Sorting

- Used when the data to be sorted is so large that we cannot use the computer's internal storage (main memory) to store it
- We use secondary storage devices to store the data
- The secondary storage devices we discuss here are tape drives. Any other storage device such as disk arrays, etc. can be used

Two-way Sorting Algorithm: Sort Phase

Algorithm:

I.Sort Phase

1. Read M records from one pair of tape drives. Initially, all the records are present only on one tape drive
2. Sort the M records in the computer's internal storage. If M is small (< 10) **use insertion sort**. For larger values of M **use quick sort**.
3. Write the M sorted records into the other pair of tape drives (i.e., the pair which does not contain the input records). While writing the records, alternate between the two tape drives of that pair.
4. Repeat steps 1-3 until the end of input

Example 2 For sorting 10 GB of data using only 1 GB of RAM:

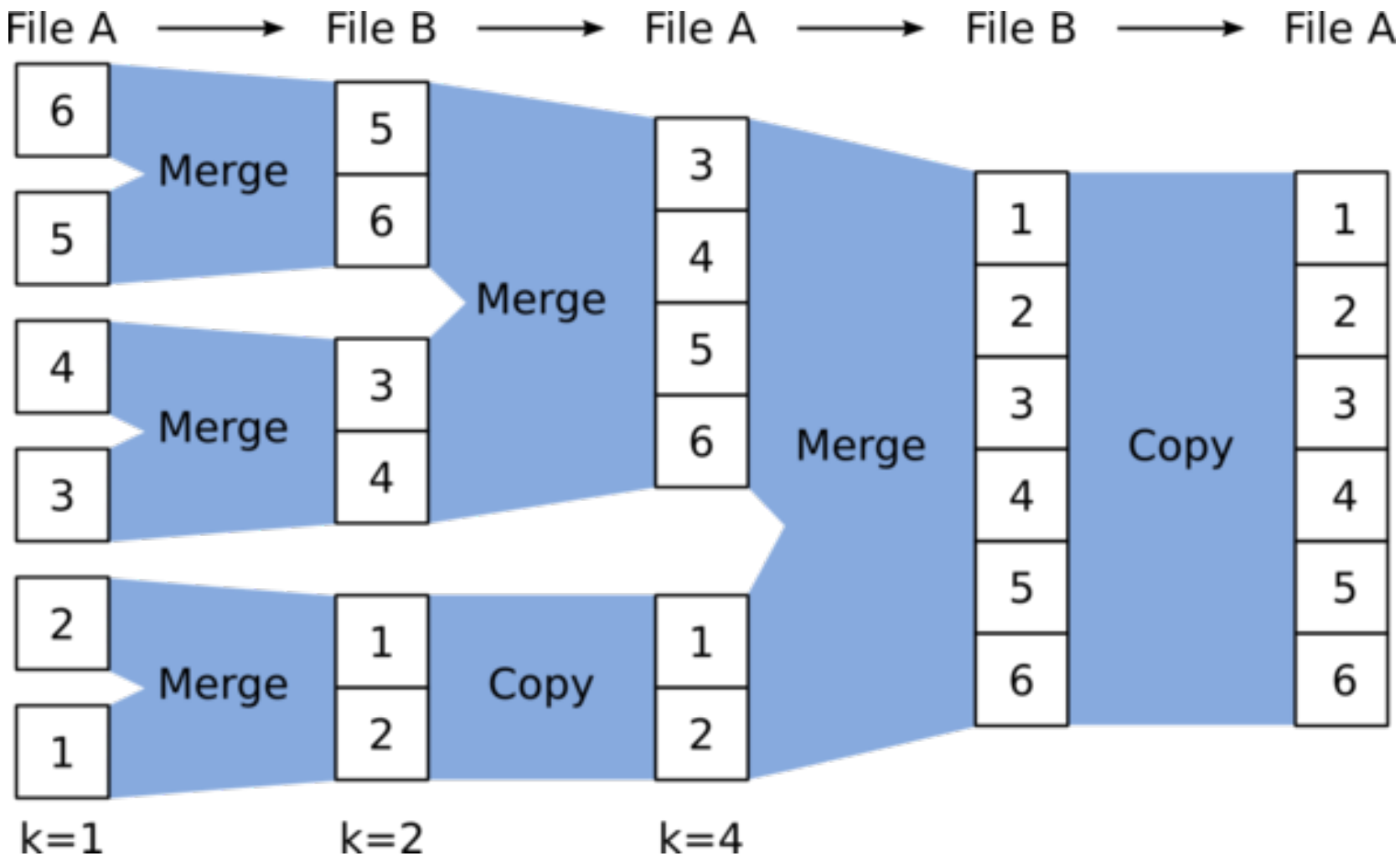
1. Read 1 GB of the data in main memory and sort by using quicksort.
2. Write the sorted data to disk.

3. Repeat steps 1 and 2 until all of the data is in sorted 1 GB chunks (there are 10 GB / 1 GB = 10 chunks), which now need to be merged into one single output file.

4. Read the first 90 MB of each sorted chunk (of 1 GB) into input buffers in main memory and allocate the remaining 100 MB for an output buffer.
(For better performance, we can take the output buffer larger and the input buffers slightly smaller.)

5. Perform a 10-way merge and store the result in the output buffer.

6. Whenever the output buffer fills, write it to the final sorted file and empty it.
Whenever any of the 90 MB input buffers empty, fill it with the next 90 MB of its associated 1 GB sorted chunk until no more data from the chunk is available.


File A
File B
File A
File B
File A
6
5
4
3
2
1
Merge
Merge
Merge
5
6
3
4
1
2
3
4
5
6
1
2
3
4
5
6
Merge
Copy
1
2
1
2
3
4
5
6
Copy
k=1
k=2
k=4

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(k)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n + k)
Bucket Sort	O(n + k)	O(n + k)	O(n^2)	O(n)


THANK
YOU