Computer Science Department Faculty
of Engineering and Technology

Comp2321

Data Structures

SORT

Chapter 7
Sorting Algorithms

OptimalSort

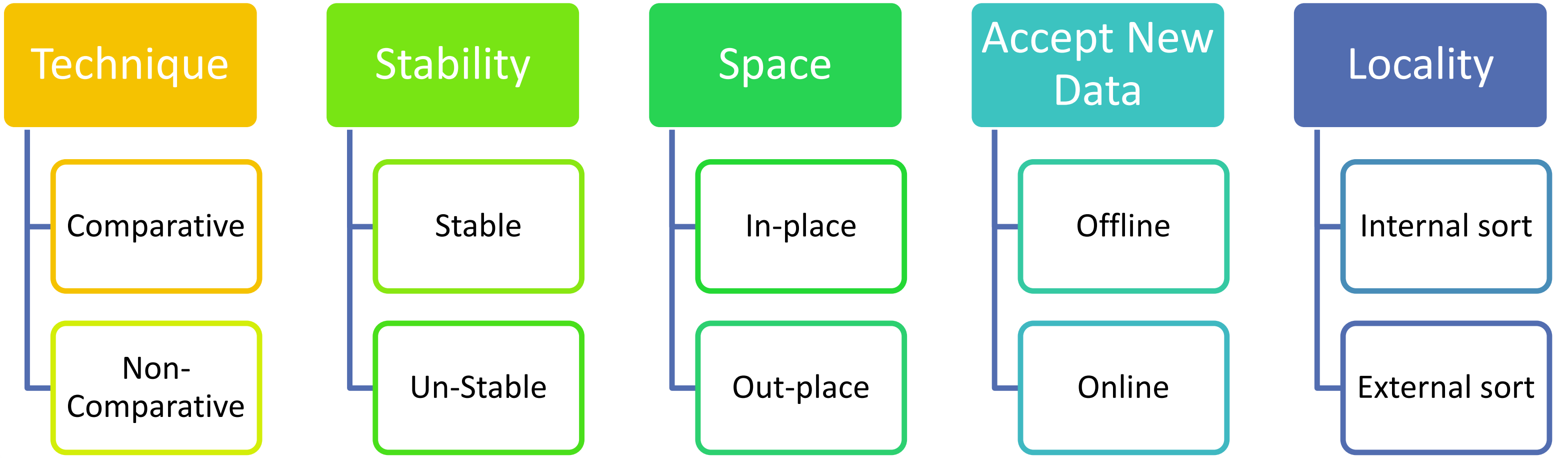
- **Sorting** is a process of arranging a collection of data items into either ascending or descending order.
- Suppose that we want to search for
  - particular record in a database
  - telephone number in telephone directory
  - TV channel in a list of more than 2000 channels.
- Majority of programming projects use a sort somewhere, and in many cases, the sorting cost determines the running time.
- **Sorting** arranges data in a sequence which makes searching easier.

Sort Algorithms

- Selection
- Bubble
- Radix/Bucket
- Heap Sort

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

Sorting Algorithms



- Technique
  - Comparative
  - Non-Comparative
- Stability
  - Stable
  - Un-Stable
- Space
  - In-place
  - Out-place
- Accept New Data
  - Offline
  - Online
- Locality
  - Internal sort
  - External sort

Sorting Algorithms

- **Types regard to the main technique:**
  - **Comparison-based**: the elements are compared with each other to construct the sorted array.
    - E.g. Bubble, Selection, Quick.
  - **Non Comparison-based**: the elements are not compared with each other to construct the sorted array.
    - E.g. Radix, Count.
- **Types regard to the memory used:**
  - **In-place**: the algorithm does not use any extra memory to sort the array. E.g. Bubble, selection.
  - **Out-place**: the algorithm uses any extra memory to sort the array. E.g. Merge, Radix

Sorting Algorithms

- **Online/Offline technique:**
- **Online:** the algorithm can accept new data while the algorithm is running, i.e. complete data is not required to start the sorting operation.
- Insertion Sort is one of the rare algorithm which satisfies this property.
- Insertion sort processes the array from left to right and if new elements are added to the right, it doesn't impact the ongoing operation.
- Most algorithms are offline.

Sorting Algorithms

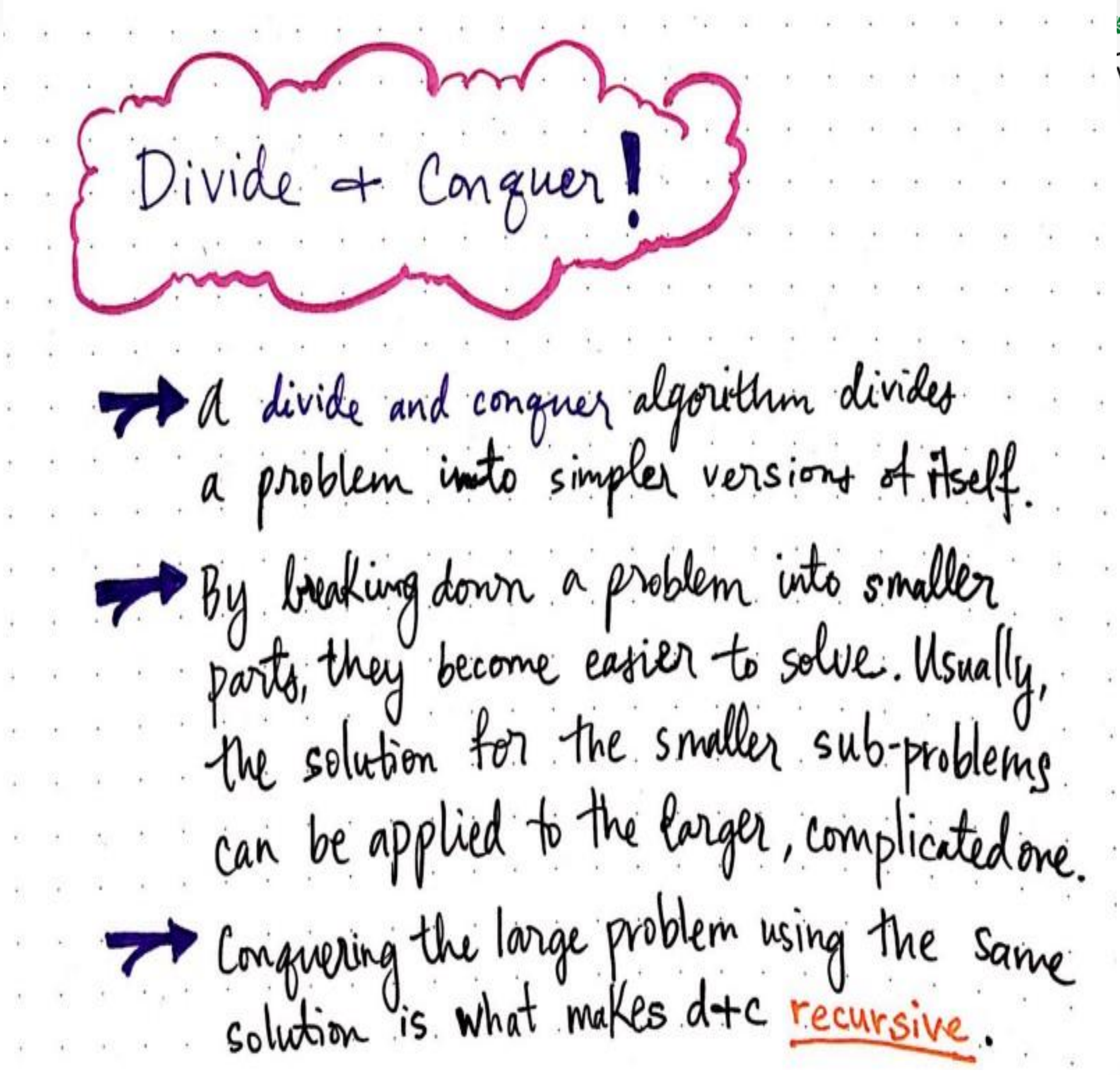
- **Types regard to the order of elements:**
- **Stable**: if the algorithm does not change the order of elements with the same value.
  - if there are two items F and S with the same key values, and F appear before S in the original list. Then F must appear before S in the sorted list.
- **Unstable:** if the algorithm may change the order of elements with the same value.
- For example, consider the array 4, 4, 1, 3. ➔ 4’, 4”, 1, 3.
  - Stable : 1, 3, 4’, 4”.
  - Unstable: 1, 3, 4”, 4’.
- Bubble sort, insertion sort and merge sort are stable algorithms.
- Selection sort is unstable.

Sorting Algorithms

- An **internal sort** requires that the collection of data fit entirely in the computer's main memory.
- An **external sort is used** when the collection of data cannot fit in the computer's main memory all at once, but must reside in secondary storage such as on a disk.
- The external merge sort is a popular example for external sorting.

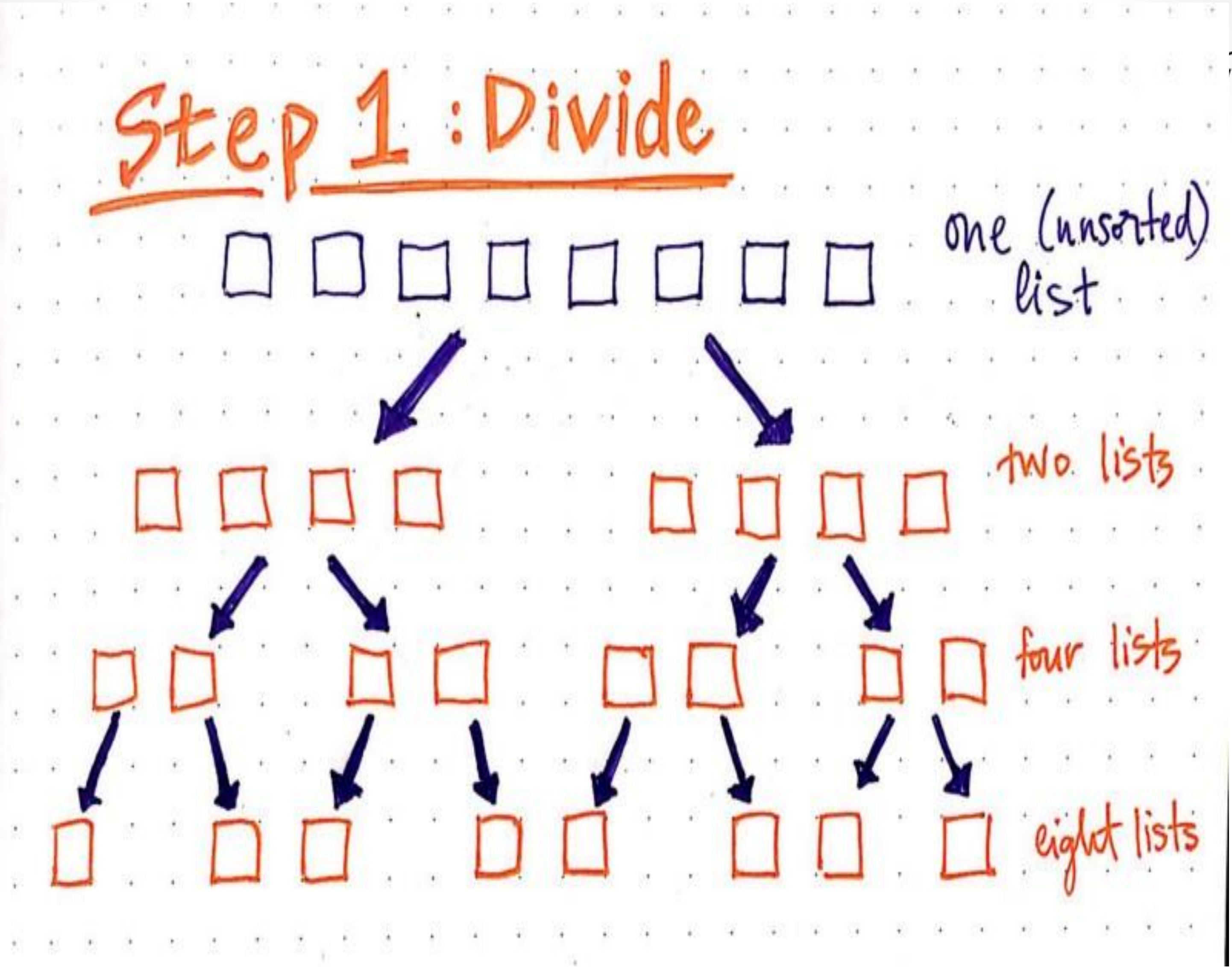
1- Merge Sort

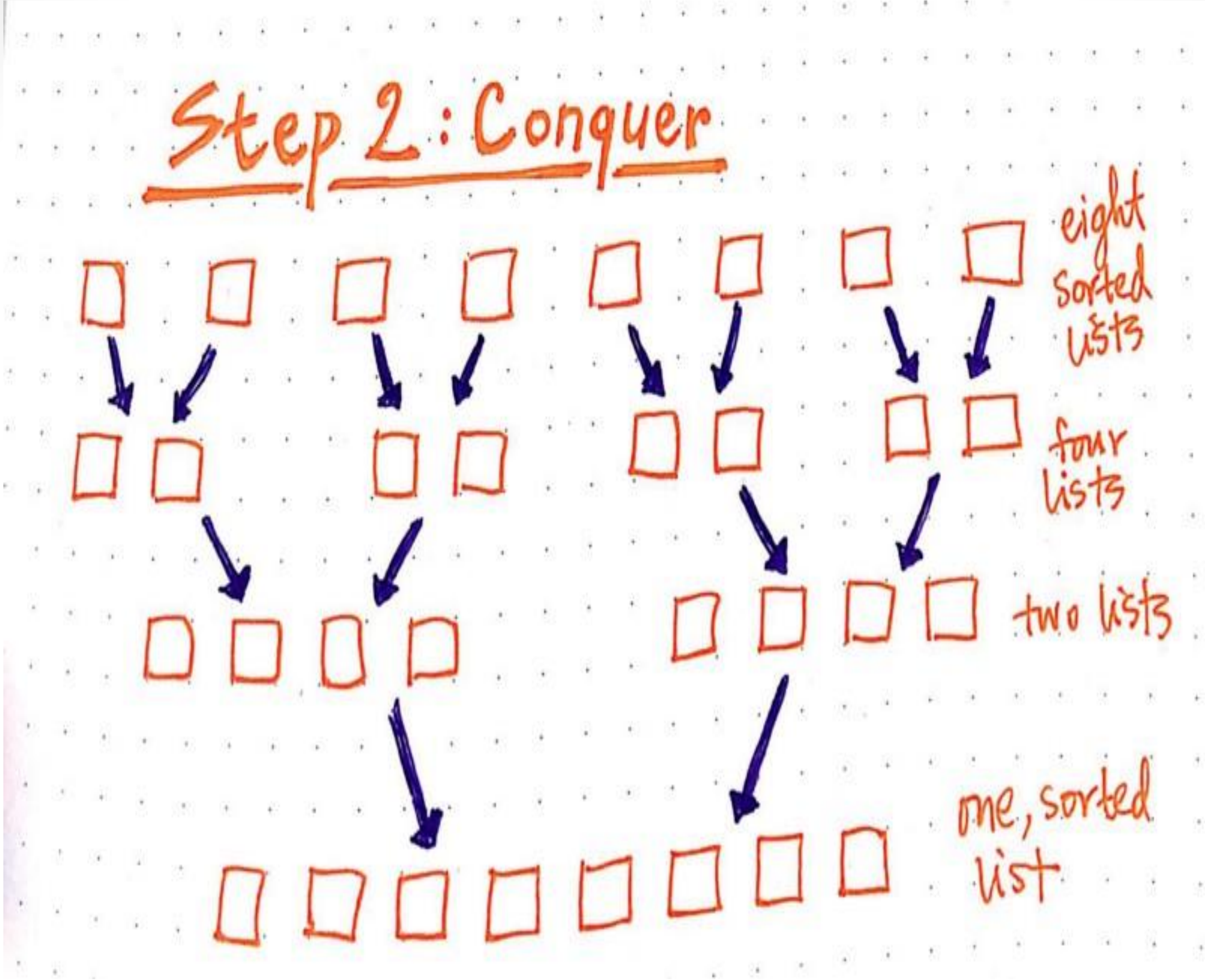
- **Divide and conquer algorithm**
- *The basic idea behind merge sort is this: it tends to be a lot easier to sort* ***two smaller, sorted lists rather than sorting a single large, unsorted one.***
- ***It's useful when data set is huge (in Gbytes) and memory is low (in Mega bytes)***



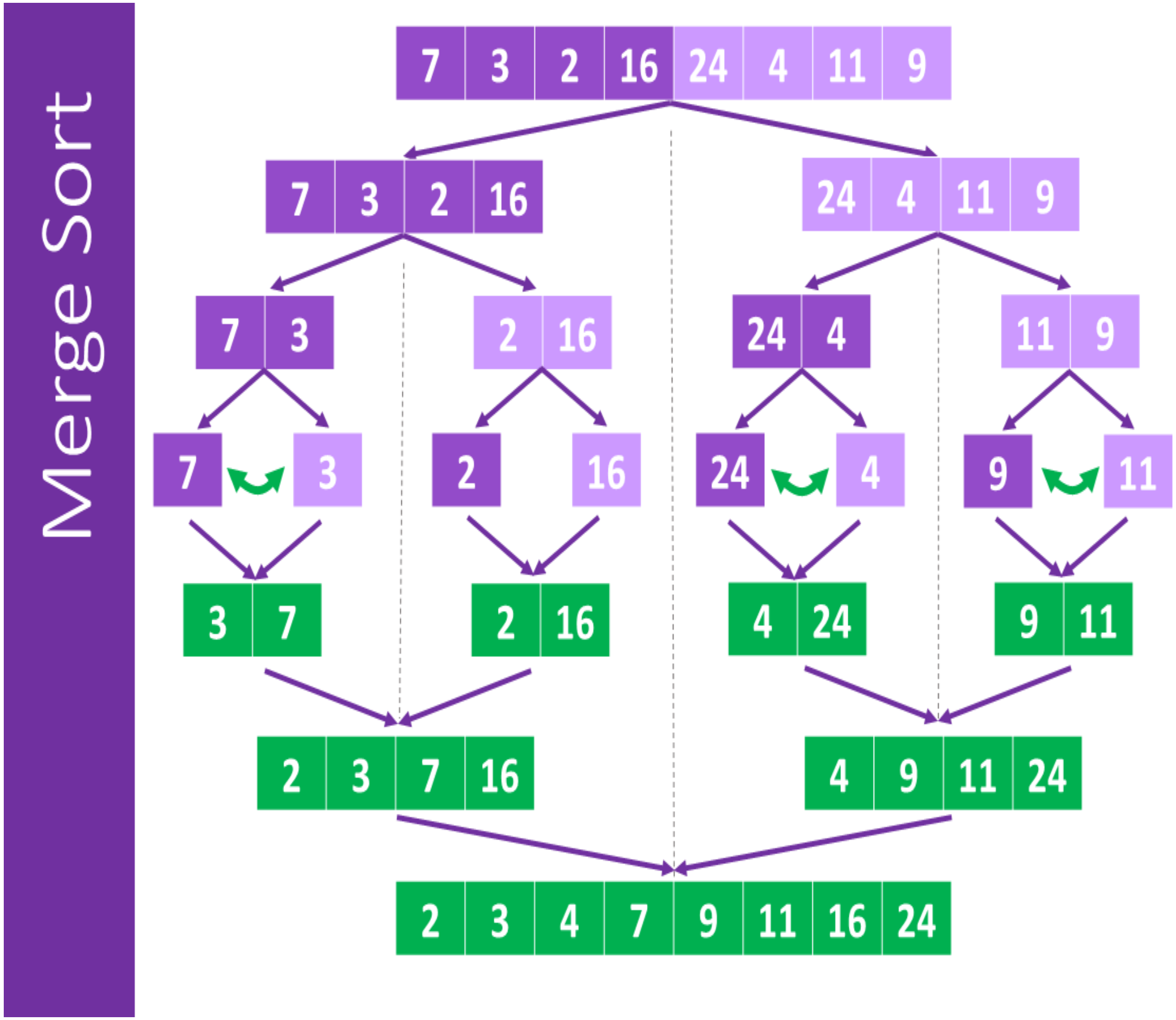
Divide + Conquer!

- → 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

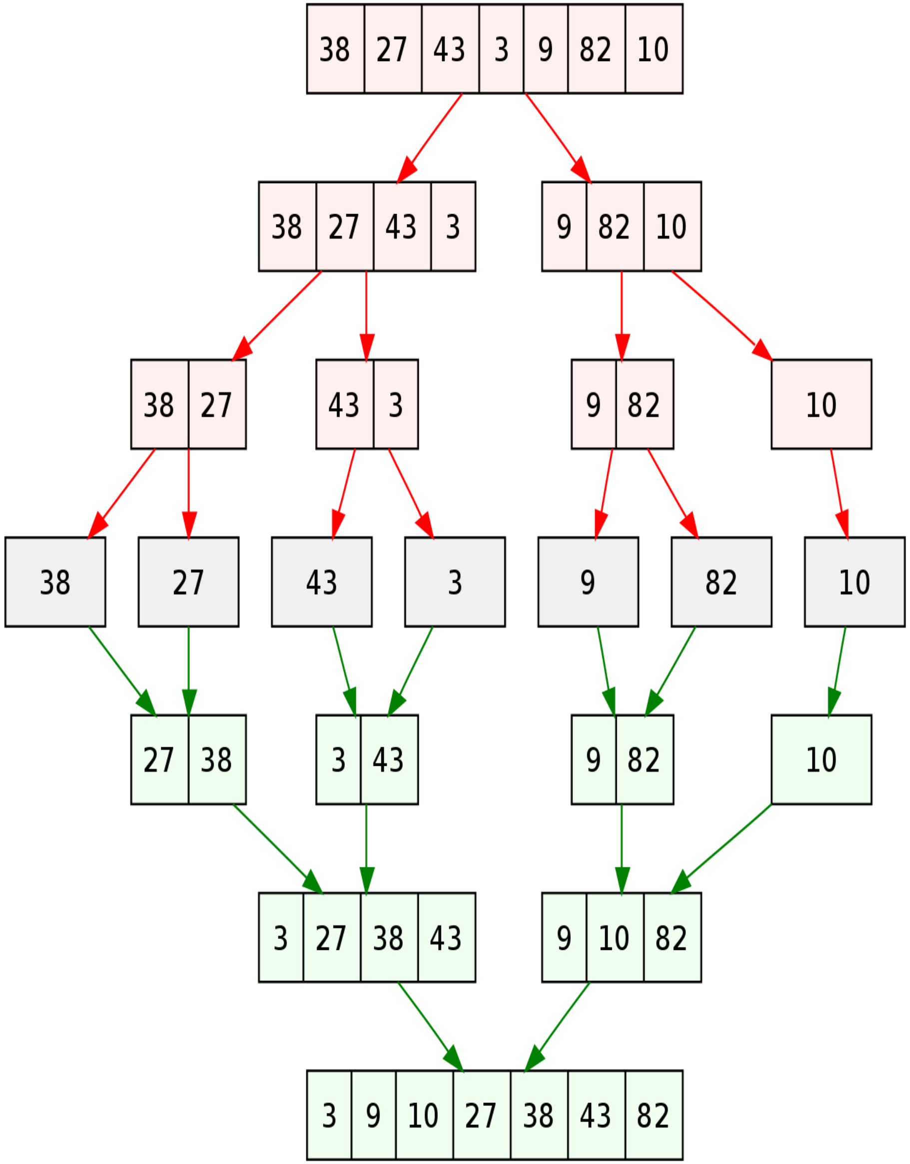


Step 1:
Split sub-lists in two until you reach pair of values.

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.

Example 2

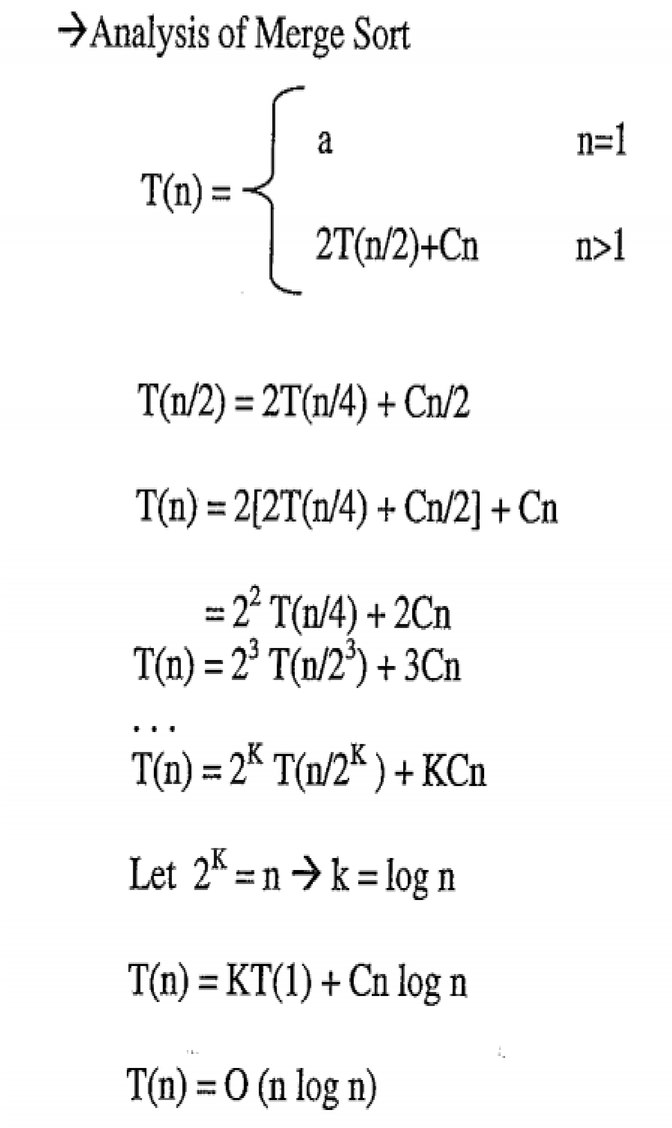

38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Merge Sort 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);
    }
}
```

Merge Sort : 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

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

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

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

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

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

$$\ldots$$

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

$$\text{Let } 2^K = n \rightarrow k = \log n$$

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

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

```
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];
}
```

Merge Sort Properties

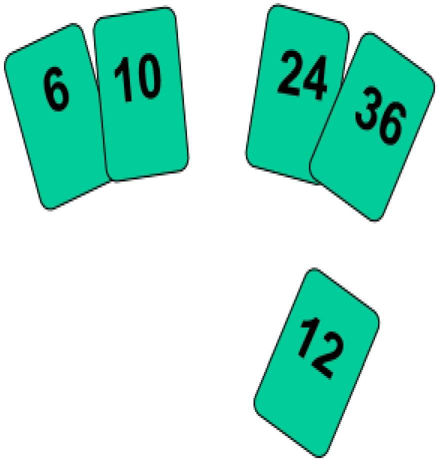
- **Time complexity:**
    - Best : O(n log n)
    - Average: O(n log n)
    - Worst: O(n log n)
- **Stable or un-stable?**
    - Stable
- **Comparative or Non-Comparative?**
    - Comparative
- **In-place or out-place?**
    - Out-place

2- Insertion Sort

- Insertion sort is a simple sorting algorithm that is appropriate for small inputs.
  - Most common sorting technique used by card players.
    - The list is divided into two parts:

      sorted and unsorted.
- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and inserted at the appropriate place.
- A list of **n** elements will take at most **n-1** passes to sort the data.

Insertion Sort



Insertion Sort

- Sorted | Unsorted

50	10	30	60	80	40
50	10	30	60	80	40
10	50	30	60	80	40
10	30	50	60	80	40
10	30	50	60	80	40
10	30	50	60	80	40
10	30	40	50	60	80

Insertion Sort: **code**

```
• void insertionSort(float a[], int n)
  {
     for (int i = 1; i < n; i++) {
          float tmp = a[i];
          for (int j=i; j>0 && tmp <
  a[j-1]; j--)
                a[j] = a[j-1];
          a[j] = tmp;
     }
  }
```

Insertion Sort-Analysis

- Running time depends on not only the size of the array but also the contents of the array.
- **Best-case:**
  - Array is already sorted in ascending order.
  - Inner loop will not be executed.
  - The number of moves: 2*(n-1) ➔ O(n)
  - The number of key comparisons: (n-1) ➔ O(n)
- **Best-case: O(n)**

Insertion Sort-Analysis

- **Worst-case:**
- – Array is in reverse order:
  – Inner loop is executed i-1 times, for i = 2,3, …, n
  – The number of moves: 2*(n-1)+(1+2+...+n-1)= 2*(n-1)+ n*(n-1)/2 ➔ O(n^2)
  – The number of key comparisons: (1+2+...+n-1)= n*(n-1)/2 ➔ O(n^2)
- **Worst-case: O(n^2)**
- **Average-case: O(n^2)**
  – We have to look at all possible initial data organizations.
- **Insertion Sort ➔Best : O(n), Average: O(n^2), Worst: O(n^2)**

Insertion Sort Properties

- **Time complexity:** Best : O(n), Average: O(n^2), Worst: O(n^2)
- **Stable or un-stable?**
  - Stable
- **Comparative or Non-Comparative?**
  - Comparative
- **In-place or out-place?**
  - In-place

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.

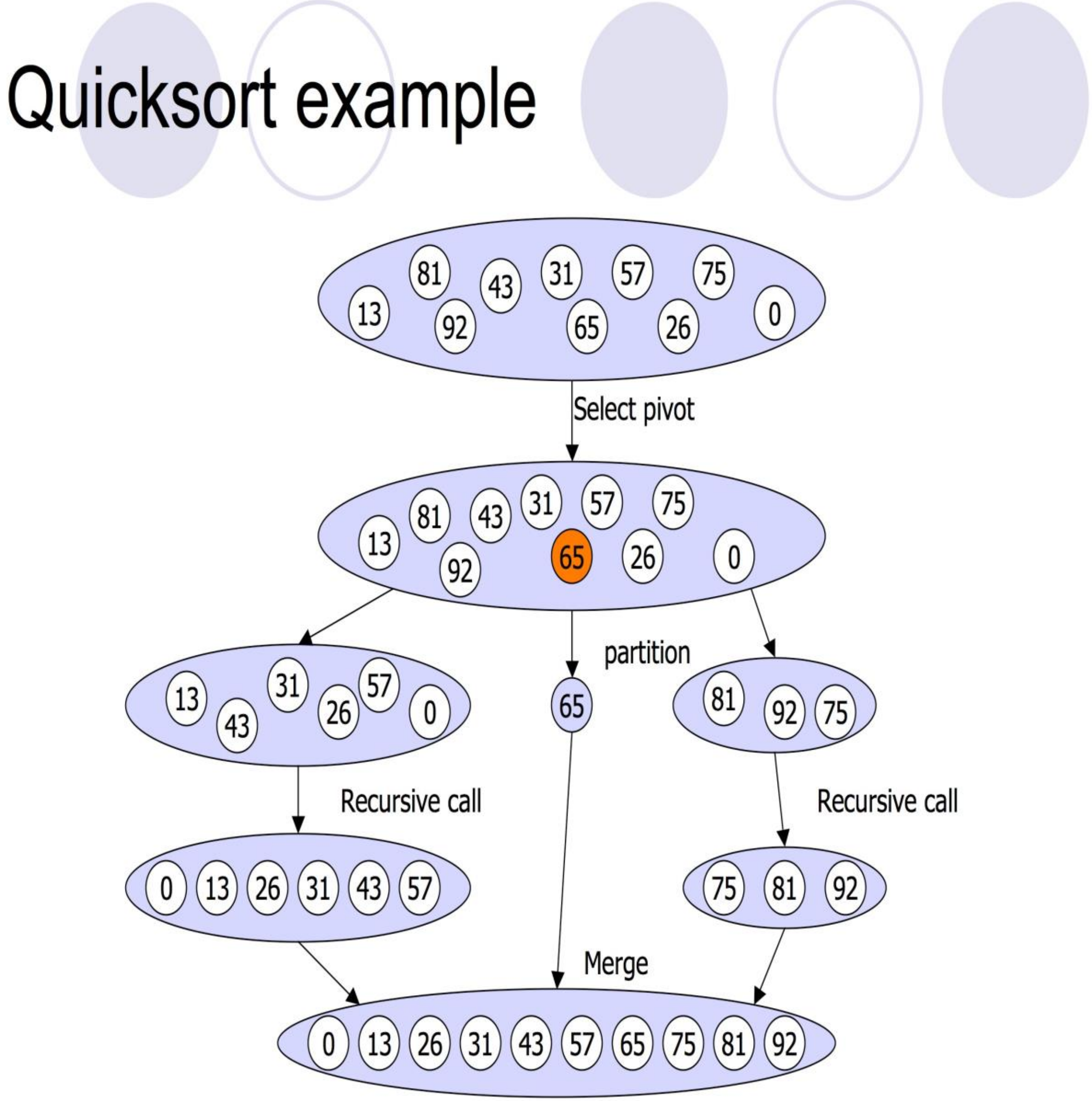
- Always pick first element as pivot.
- Always pick last element as pivot
- Pick a random element as pivot.
- Pick median as pivot.

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

Instructor: Murad Njoum

Quick Sort


Quicksort example

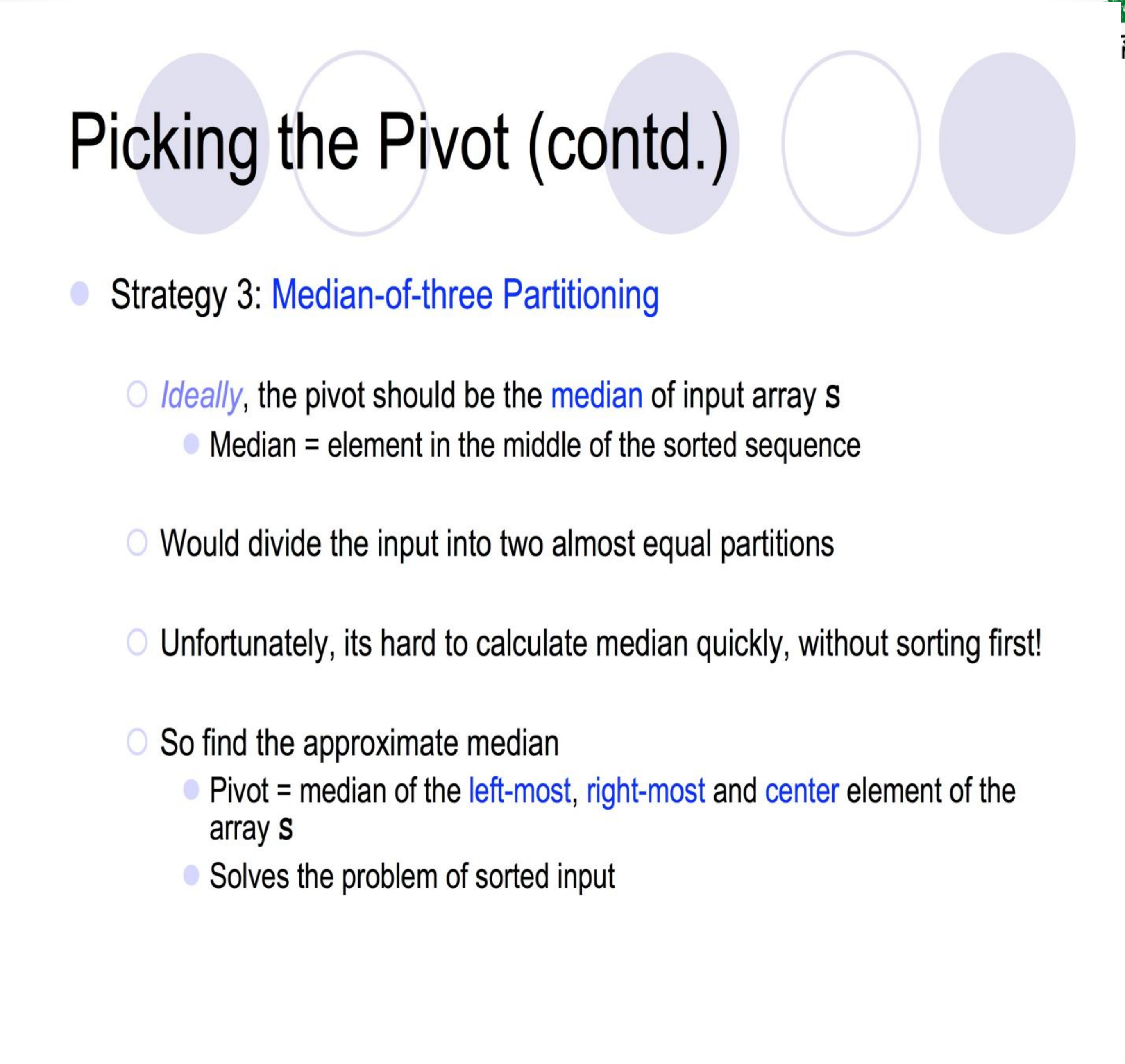
Picking the Pivot

- 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

Instructor: Murad Njoum

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

Instructor: Murad Njoum

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`

Instructor: Murad Njoum

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

8 1 4 9 0 3 5 2 7 6

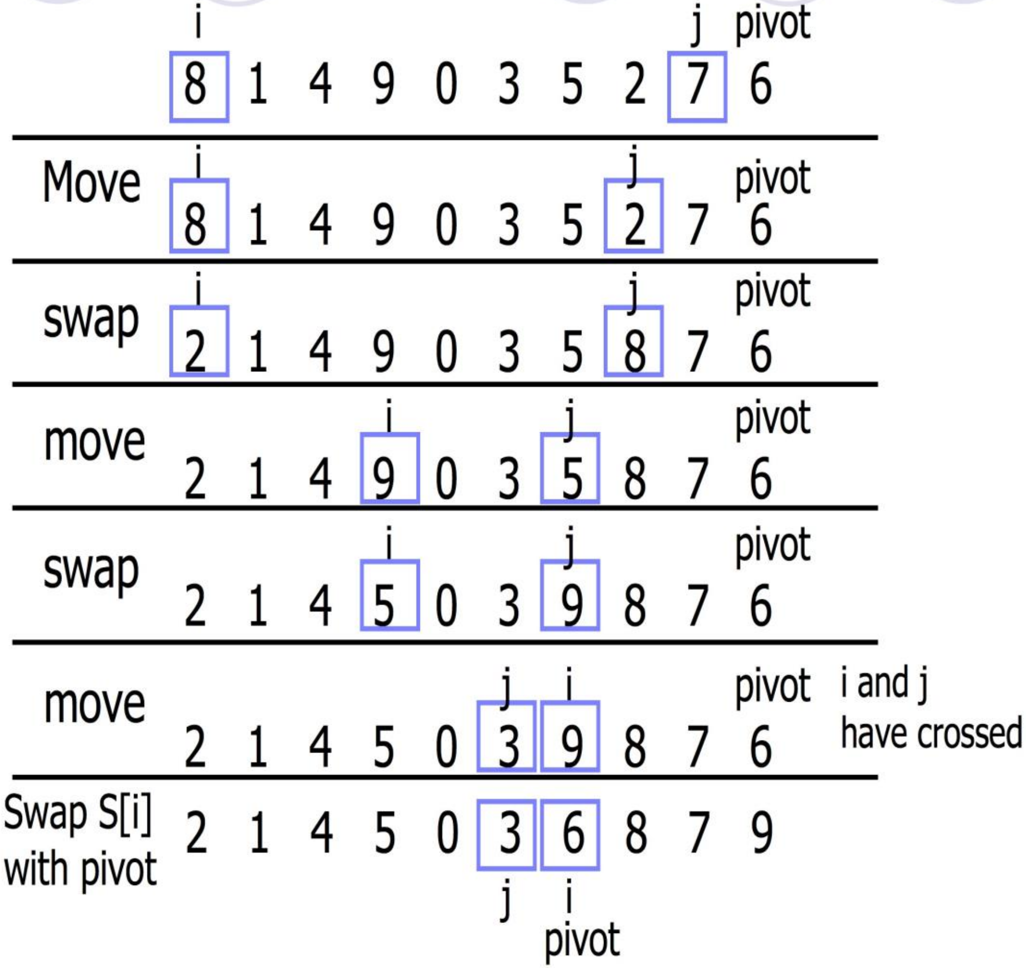
pivot

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

8 1 4 9 0 3 5 2 7 6

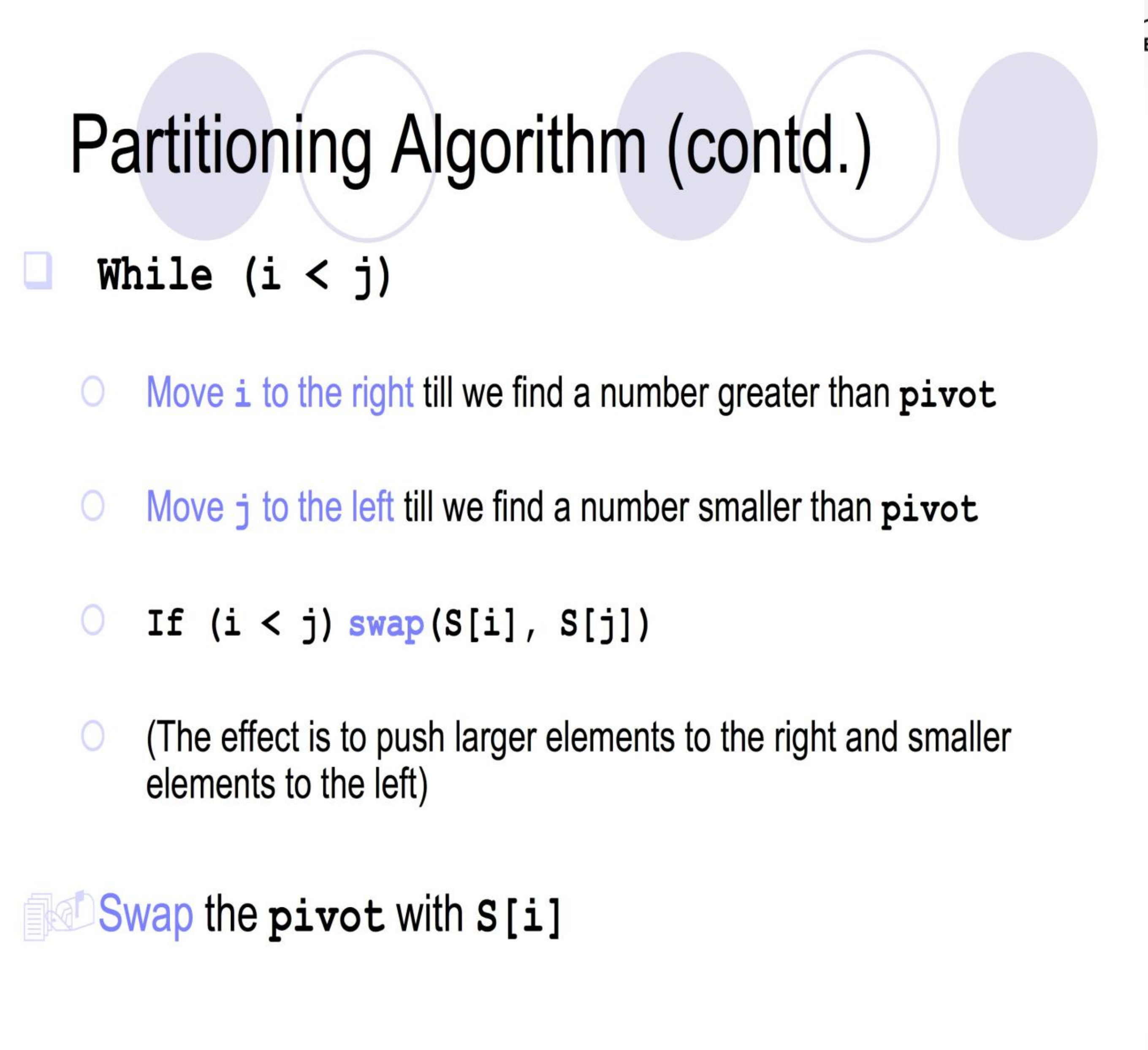
i j pivot

Partitioning Algorithm Illustrated



	i								j	pivot	
	8	1	4	9	0	3	5	2	7	6	
Move	i							j		pivot	
	8	1	4	9	0	3	5	2	7	6	
swap	i							j		pivot	
	2	1	4	9	0	3	5	8	7	6	
move				i			j			pivot	
	2	1	4	9	0	3	5	8	7	6	
swap				i			j			pivot	
	2	1	4	5	0	3	9	8	7	6	
move						j	i			pivot	i and j have crossed
	2	1	4	5	0	3	9	8	7	6	
Swap S[i] with pivot	2	1	4	5	0	3	6	8	7	9	
						j	i pivot				

Instructor: Murad Njoum



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]`

Instructor: Murad Njoum

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

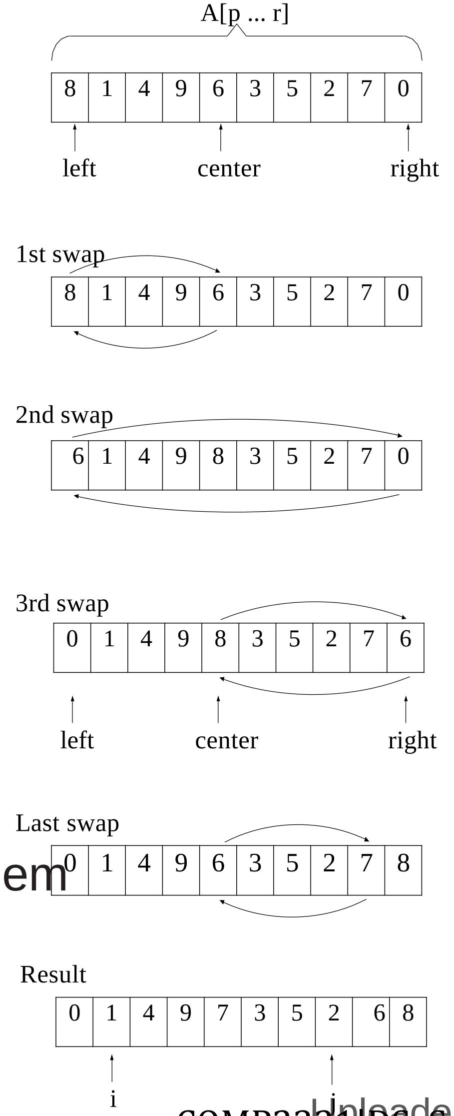
Is left(6) > right (0) , swap them

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

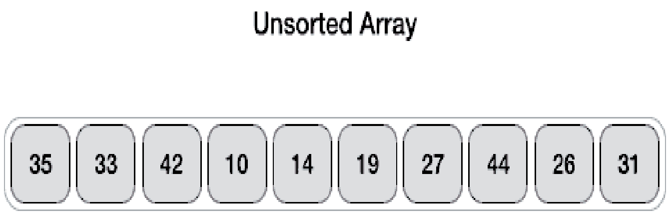
Is center(8) > right (6) , swap them

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 - 1); //swap occur between i and pivot
```



Unsorted Array

35	33	42	10	14	19	27	44	26	31

```
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 – 1); //swap median pivat with most right elements in array
return A[right – 1]; //return the pivot
}
```

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

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.

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

→ Analysis of Quick Sort

- Worst case Analysis

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

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

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

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

. . .

. . .

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

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

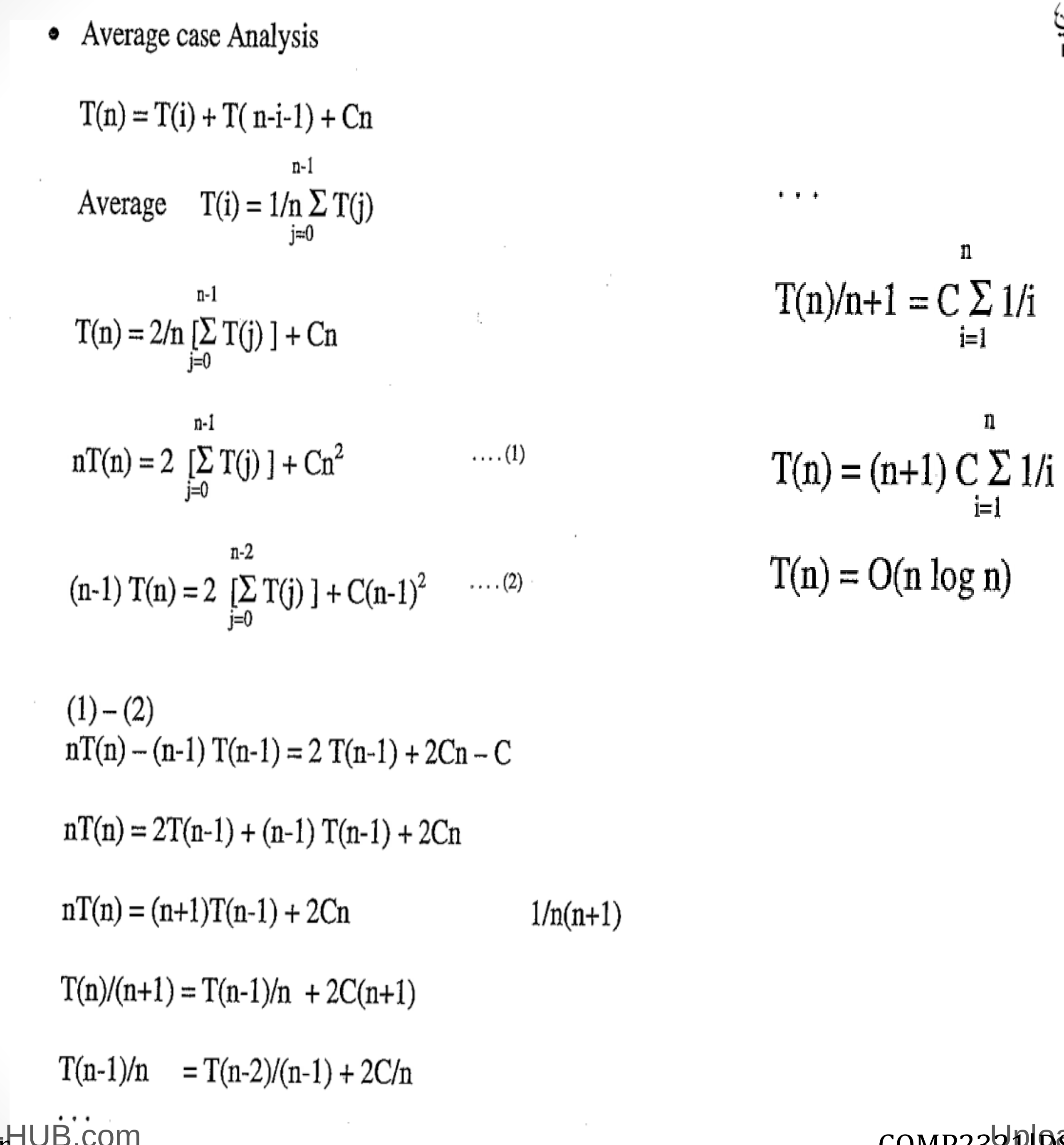
- Best case Analysis

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

. . .

. . .

$$\boxed{T(n) = O(n \log n)}$$



- Average case Analysis

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

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

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

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

$$(n-1)\,T(n) = 2 \left[\sum_{j=0}^{n-2} T(j)\right] + C(n-1)^2 \qquad \ldots(2)$$

(1) – (2)

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

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

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

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

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

. . .

. . .

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

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

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

Shell Sort

Shell Sort is mainly a variation of Insertion Sort.

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

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

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

5 shifts

shell techniques:

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

This is called GAP

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

0	1	2	3	4	5	6	7	8
23	29	15	19	31	7	9	5	2

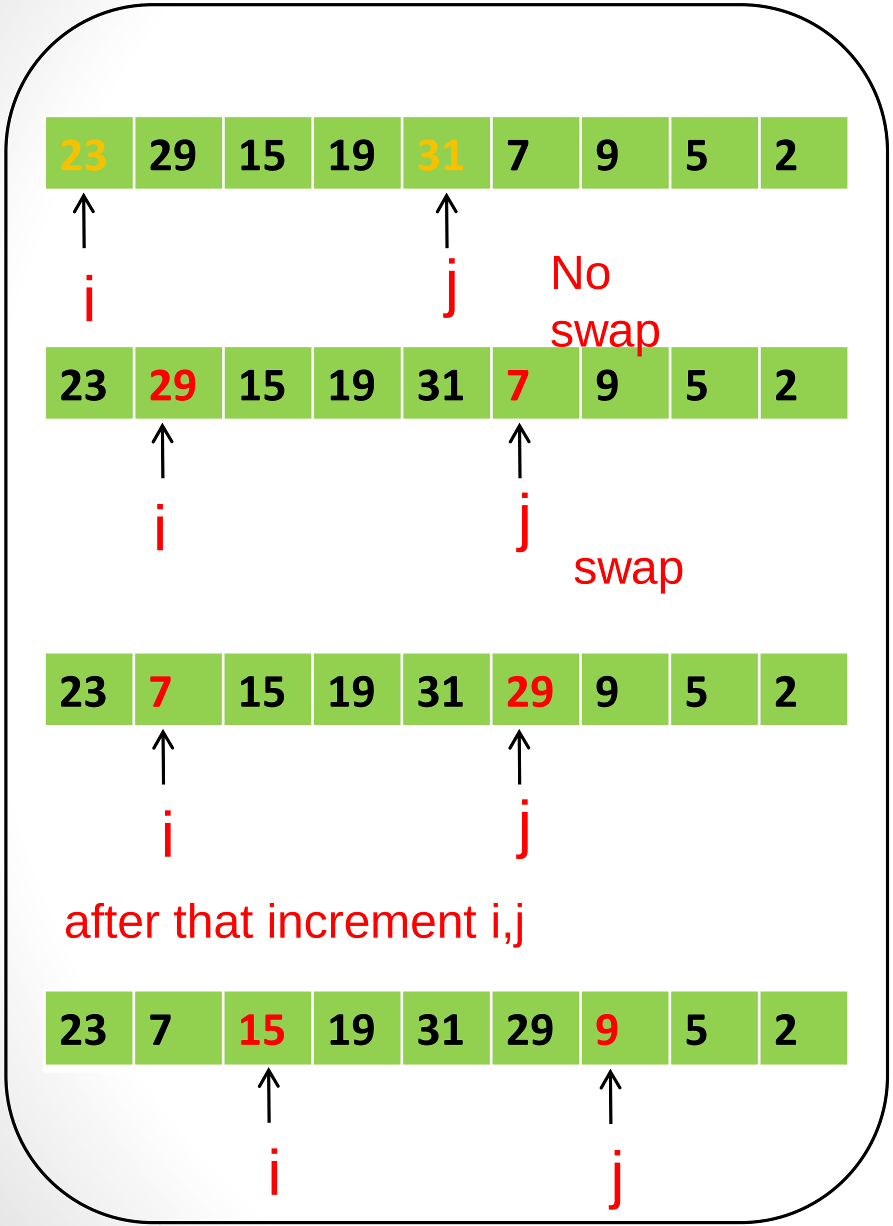
i j

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



23	29	15	19	31	7	9	5	2

No swap

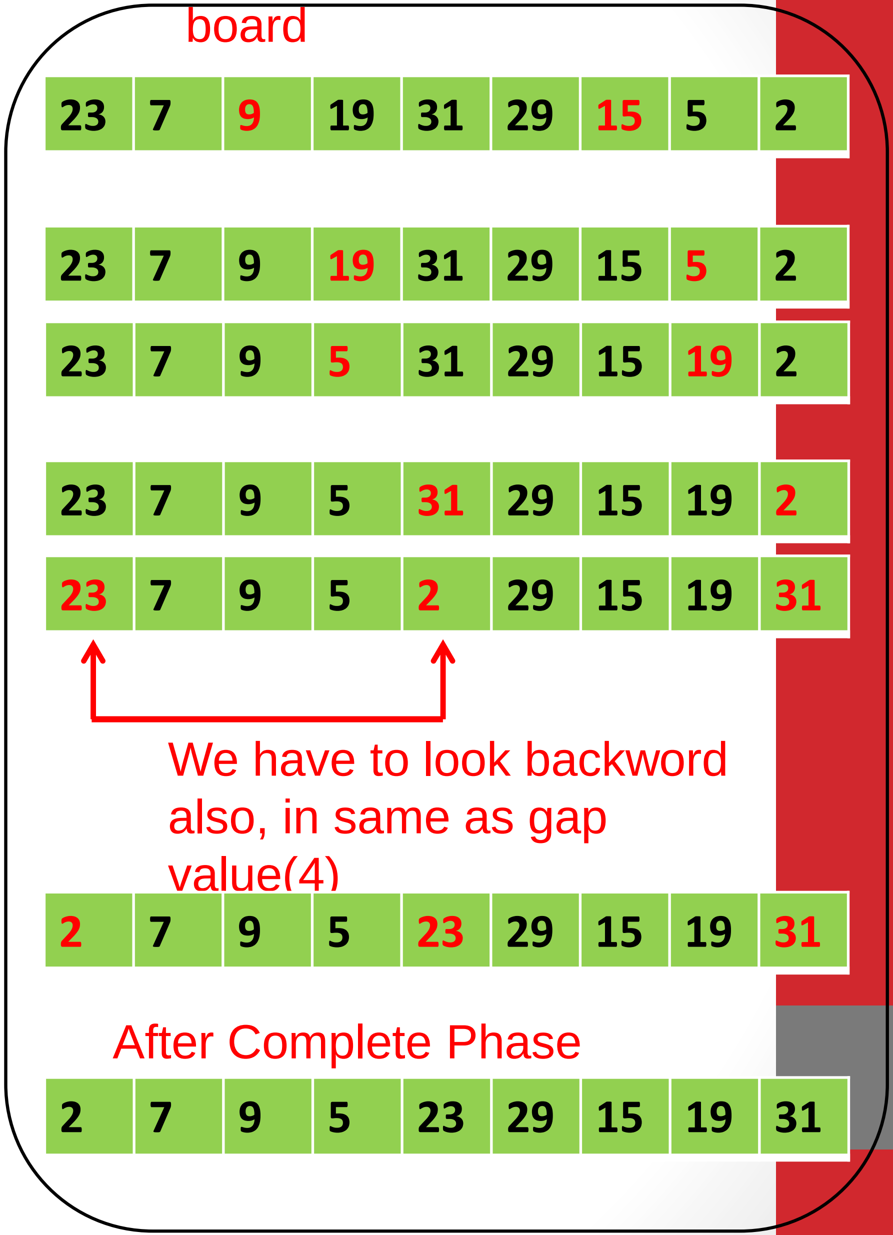
23	29	15	19	31	7	9	5	2

swap

23	7	15	19	31	29	9	5	2

after that increment i,j

23	7	15	19	31	29	9	5	2



Solution in class At board

23	7	9	19	31	29	15	5	2

23	7	9	19	31	29	15	5	2
23	7	9	5	31	29	15	19	2

23	7	9	5	31	29	15	19	2
23	7	9	5	2	29	15	19	31

We have to look backword also, in same as gap value(4)

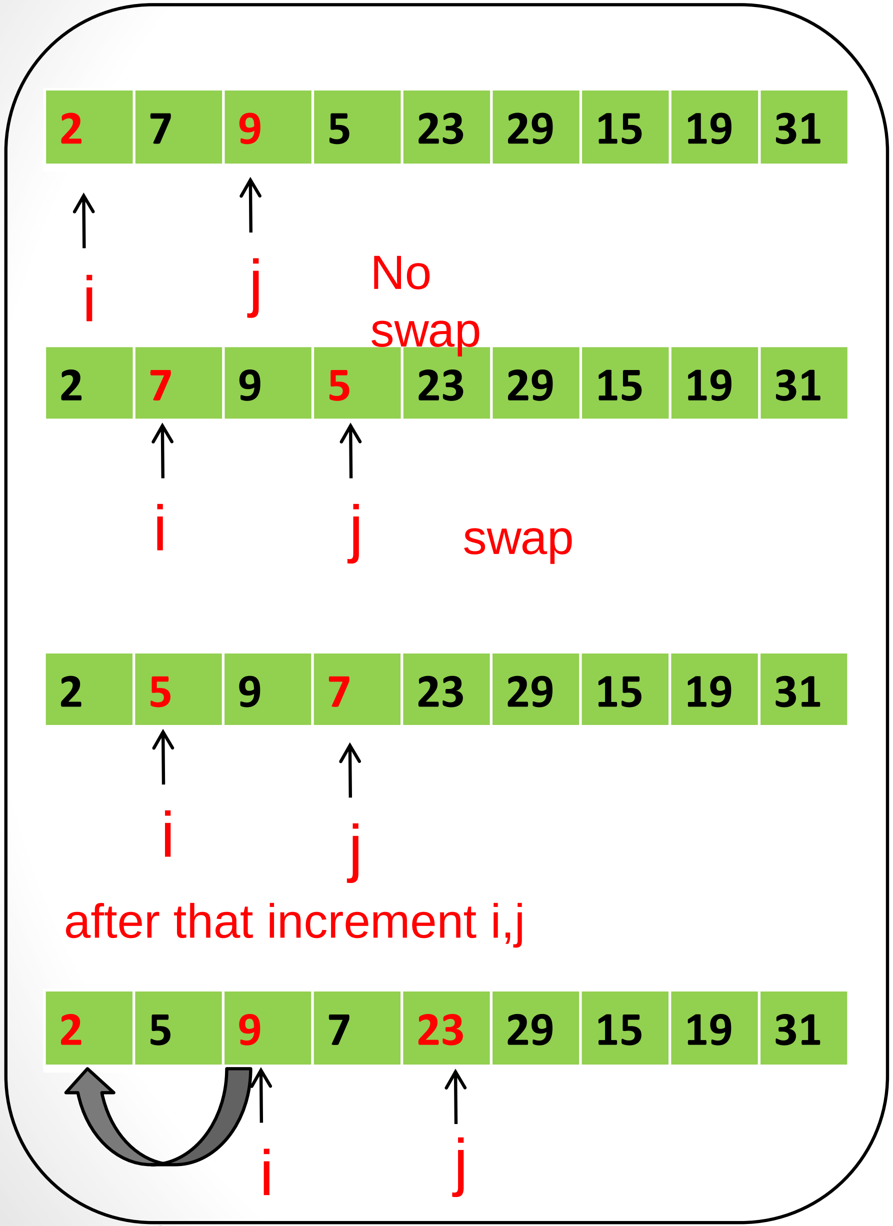
2	7	9	5	23	29	15	19	31

After Complete Phase

2	7	9	5	23	29	15	19	31

Shell Sort:

Gap= 4/2=2



2	7	9	5	23	29	15	19	31

No swap

2	7	9	5	23	29	15	19	31

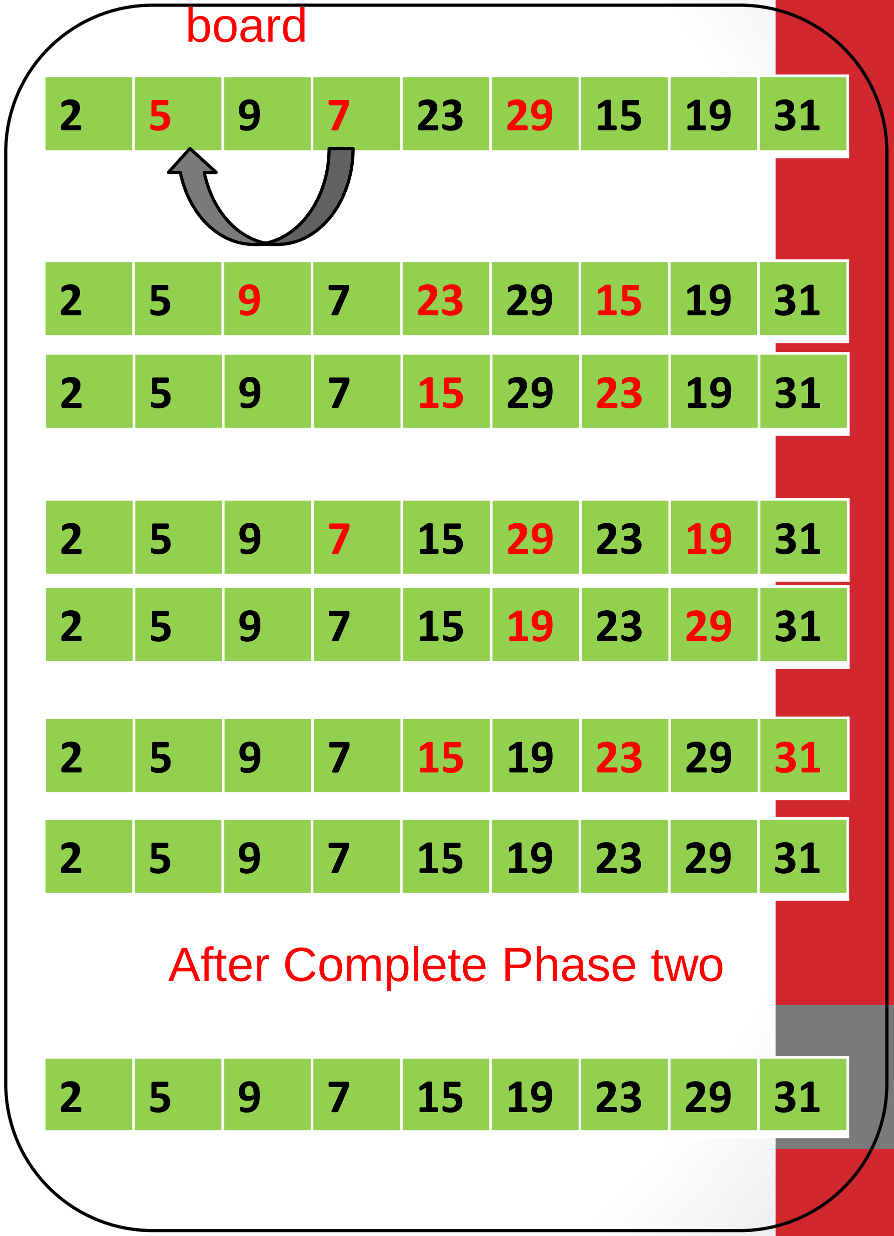
swap

2	5	9	7	23	29	15	19	31

after that increment i,j

2	5	9	7	23	29	15	19	31

Solution in class At board



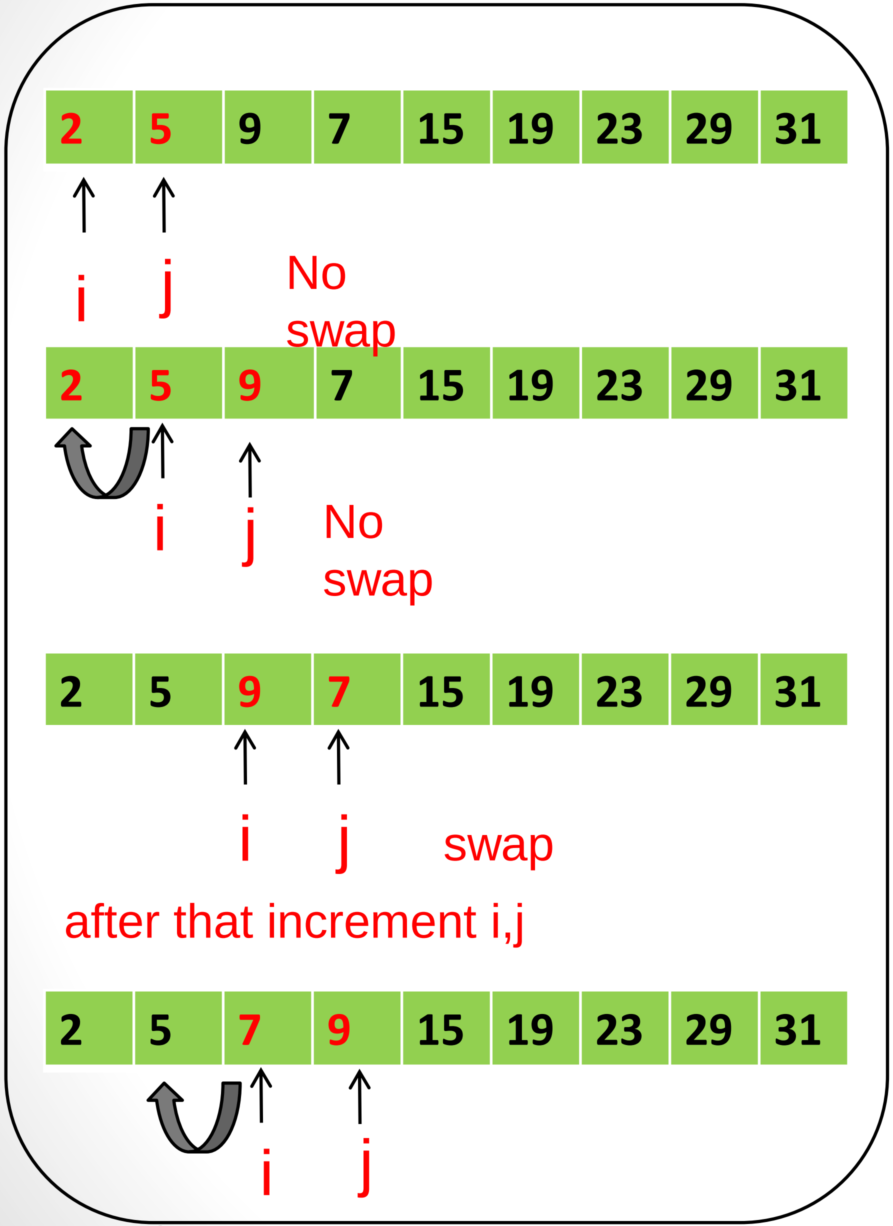
2	5	9	7	23	29	15	19	31
2	5	9	7	23	29	15	19	31
2	5	9	7	15	29	23	19	31
2	5	9	7	15	29	23	19	31
2	5	9	7	15	19	23	29	31
2	5	9	7	15	19	23	29	31
2	5	9	7	15	19	23	29	31

After Complete Phase two

2	5	9	7	15	19	23	29	31

Shell Sort:

Gap= 2/2=1



2	5	9	7	15	19	23	29	31

No swap

2	5	9	7	15	19	23	29	31

No swap

2	5	9	7	15	19	23	29	31

swap

after that increment i,j

2	5	7	9	15	19	23	29	31

Solution in class At board

2	5	7	9	15	19	23	29	31
2	5	7	9	15	19	23	29	31
2	5	7	9	15	19	23	29	31
2	5	7	9	15	19	23	29	31
2	5	7	9	15	19	23	29	31
2	5	7	9	15	19	23	29	31

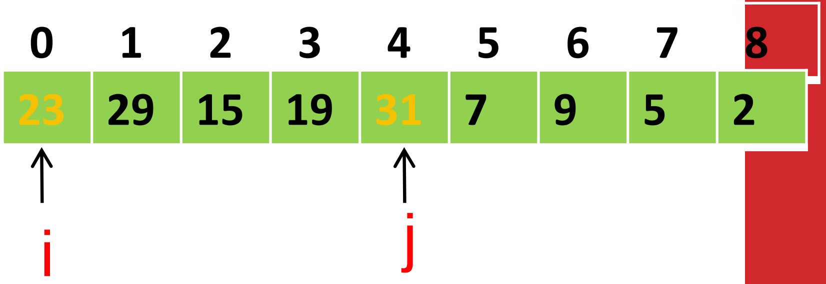
After Complete Phase three

2	5	7	9	15	19	23	29	31

Gap= 1/2=0 ,stop

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 )
            if(  A[ i ]< A[ i + gap ] )  //test for
swap
                  {temp = A[ i + gap ];
                   A[ i + gap ]=A[i];
                  A[ i ] = Tmp;
                   }
            else
                  break;
         }
}
```



0	1	2	3	4	5	6	7	8
23	29	15	19	31	7	9	5	2

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

...

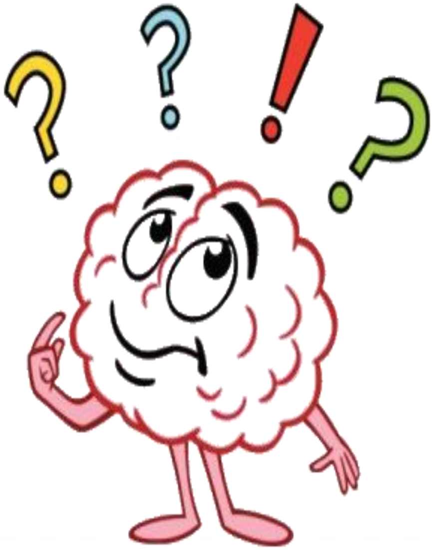
j=8,i=8-4=4......if it true then swap

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

Time Complexity: Time complexity of above implementation of **shellsort** is **O(n²)**. In the above implementation gap is reduce by half in every iteration. There are many other ways to reduce gap which lead to

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

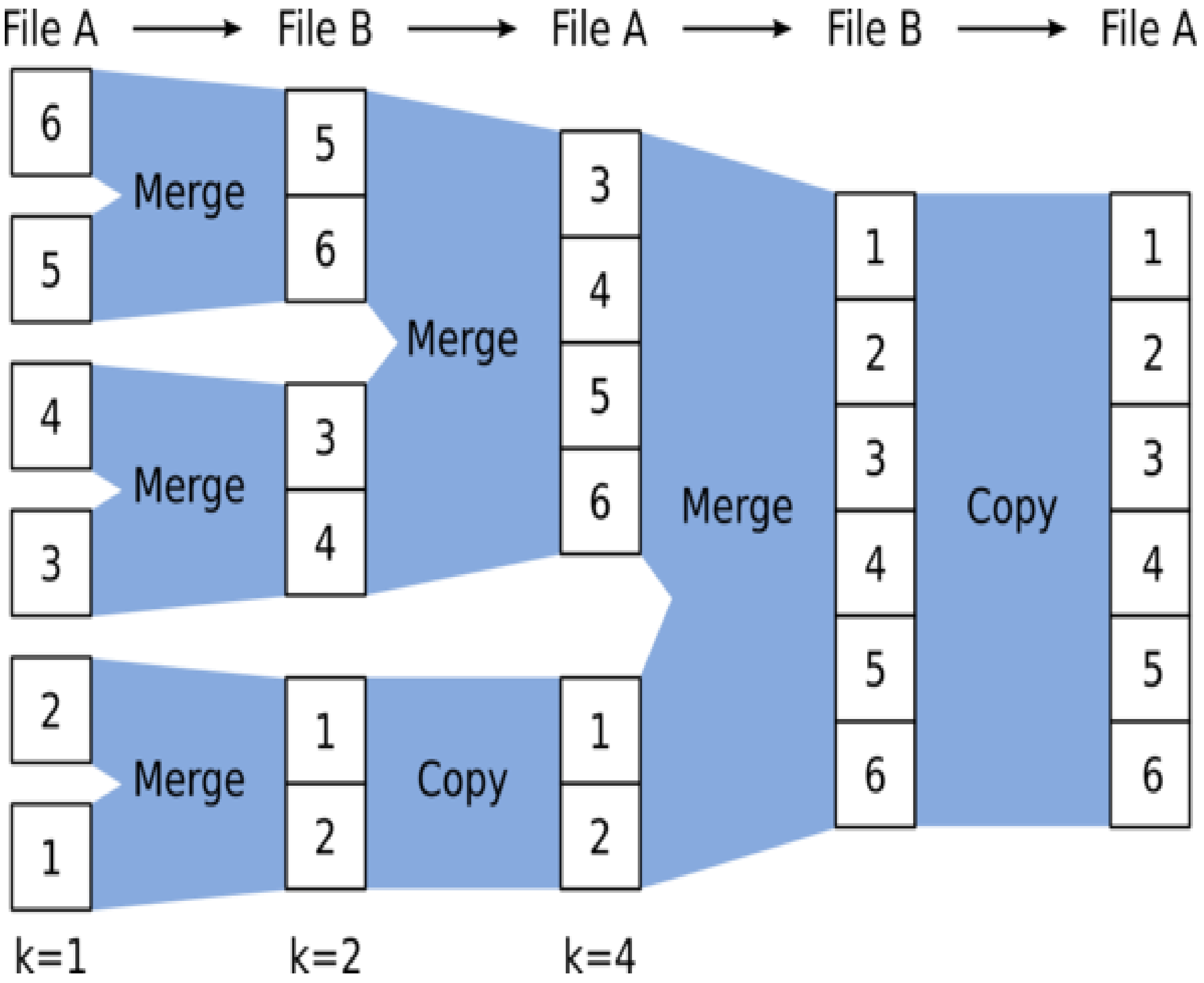
Example 2 For sorting 10 GB of data using only 1 GB of RAM:

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


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
Merge
Copy
3
4
5
6
1
2
Merge
1
2
3
4
5
6
Copy
1
2
3
4
5
6
k=1
k=2
k=4

Dear Students

Always Remember...

1. You are important.
2. You are special!
3. I believe in you.
4. I trust you.
5. You are listened to.
6. Your opinion matters.
7. I care about you.
8. I respect you.
9. You are a winner!
10. I will help you succeed!

Instructor: Murad Njoum


THANK
YOU