Sorting

Why sorting

- Sorting solves togetherness problem or to find the similar elements
- Matching items in two or more files
- Searching by key values

Factors that may affect sorting

Range of Numbers

Count of Numbers

Uniqueness of Numbers

Already Sorted or not

Availability of Numbers

Type of data to be sorted

Want to retain stableness of numbers

To use extra memory or in place

Some Definitions

- Internal Sort
  - The data to be sorted is all stored in the computer's main memory.
- External Sort
  - Some of the data to be sorted might be stored in some external, slower, device.
- In Place Sort
  - The amount of extra space required to sort the data is constant with the input size.

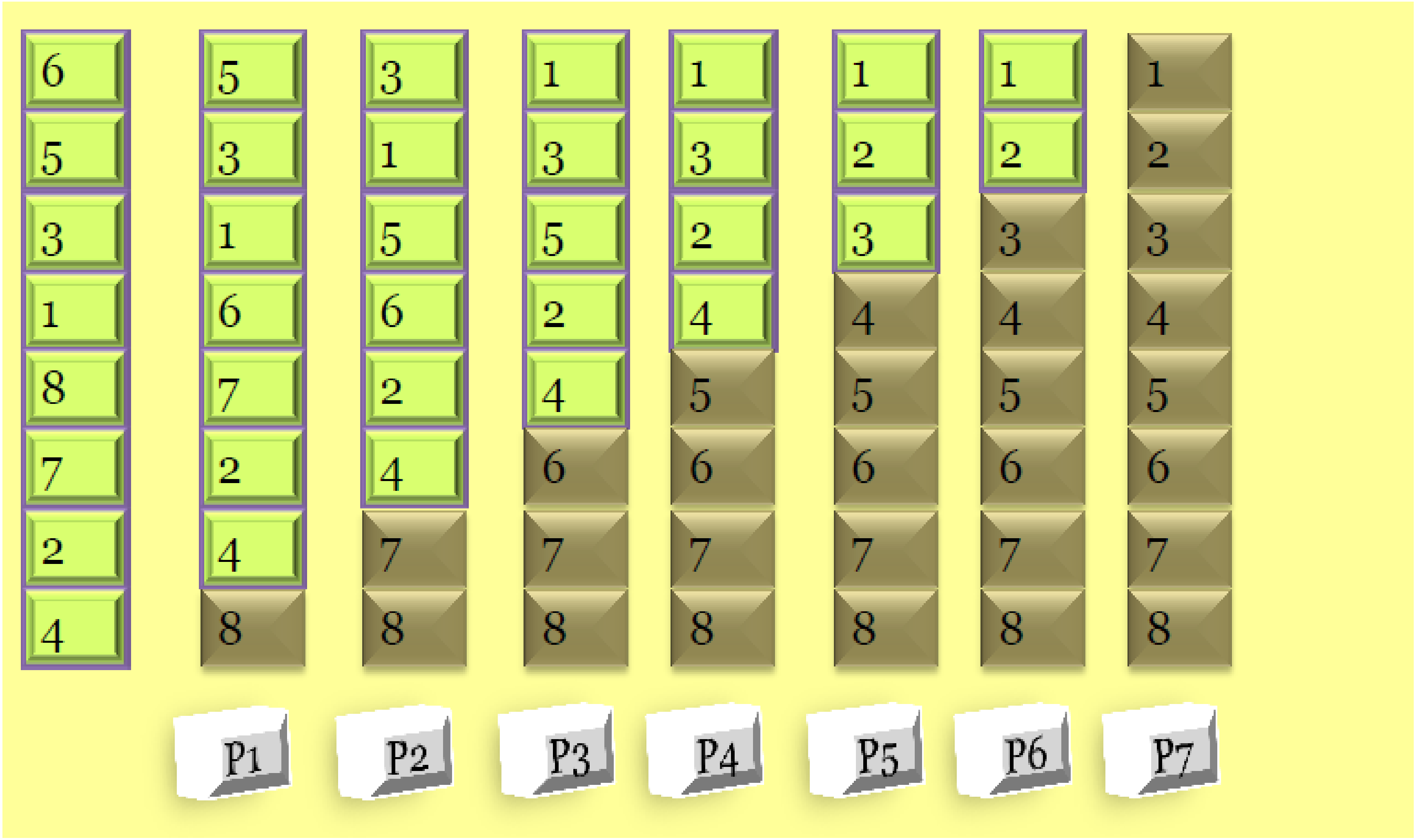
Stable and unstable sort

Ram	3
Krishan	4
Shyam	5
Narhari	8
Vaman	4
Kapil	5
Budha	4
Balram	8

Ram	3
Krishan	4
Vaman	4
Budha	4
Shyam	5
Kapil	5
Narhari	8
Balram	8

Ram	3
Budha	4
Krishan	4
Vaman	4
Shyam	5
Kapil	5
Balram	8
Narhari	8

Bubble Sort Example



Bubble Sort

It works by comparing adjacent elements of an array and exchanges them if they are not in order.

After each iteration (pass) the largest element sinks to the last position of the array. In next iteration second largest element sinks to second last position and so on

```
1. for i = 1 to n-1 do
1.1 for j = 1 to n-i do
1.1.1 If (a[j+1] < a[j]) then swap a[j] and a[j+1]
```

Bubble Sort

- Takes relatively less swaps for sorting the nearly sorted data
- Slowest algorithm for sorting nearly reverse sorted data
- Speed with which largest element sinks is high - it occupies its proper place in one pass
- Speed with which small element bubbles up is slow- only one exchange is done on each path
- Largest elements are rabbits and smallest tortoises
- It is stable, in place and popular due to tiny code

Improvement in Bubble Sort

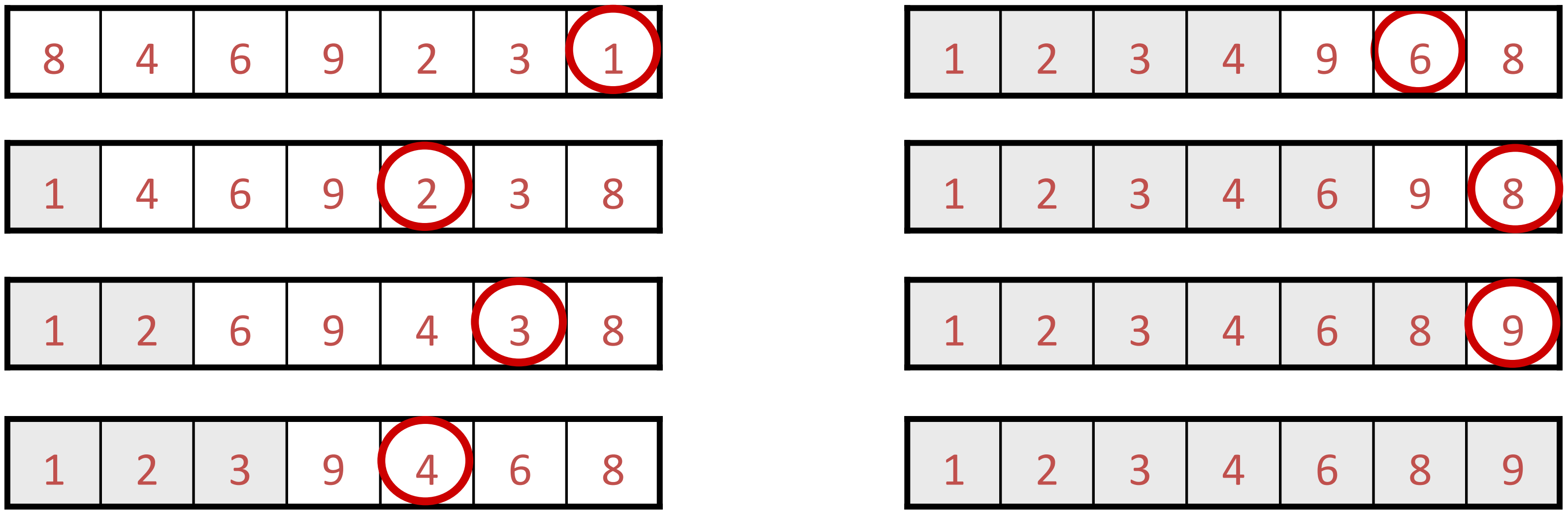
- Traversing in opposite direction in alternate passes.
- If two adjacent elements don’t exchange for two consecutive passes then we can fix their position.
- Sorting after implementing these improvements is also Called cocktail sort.
- Here’s an example *data set* which would require 9 iterations with a Bubble Sort, but only 1 iteration (of two stages) with a Cocktail Sort:

  20, 33, 45, 55, 64, 74, 77, 83, 98, 13

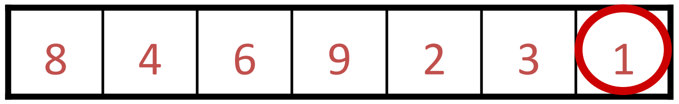
Selection Sort

- Idea:
  - Find the smallest element in the array
  - Exchange it with the element in the first position
  - Find the second smallest element and exchange it with the element in the second position
  - Continue until the array is sorted

Example


8 4 6 9 2 3 1
1 4 6 9 2 3 8
1 2 6 9 4 3 8
1 2 3 9 4 6 8
1 2 3 4 9 6 8
1 2 3 4 6 9 8
1 2 3 4 6 8 9
1 2 3 4 6 8 9

Selection Sort



Alg.: SELECTION-SORT(*A, N*)

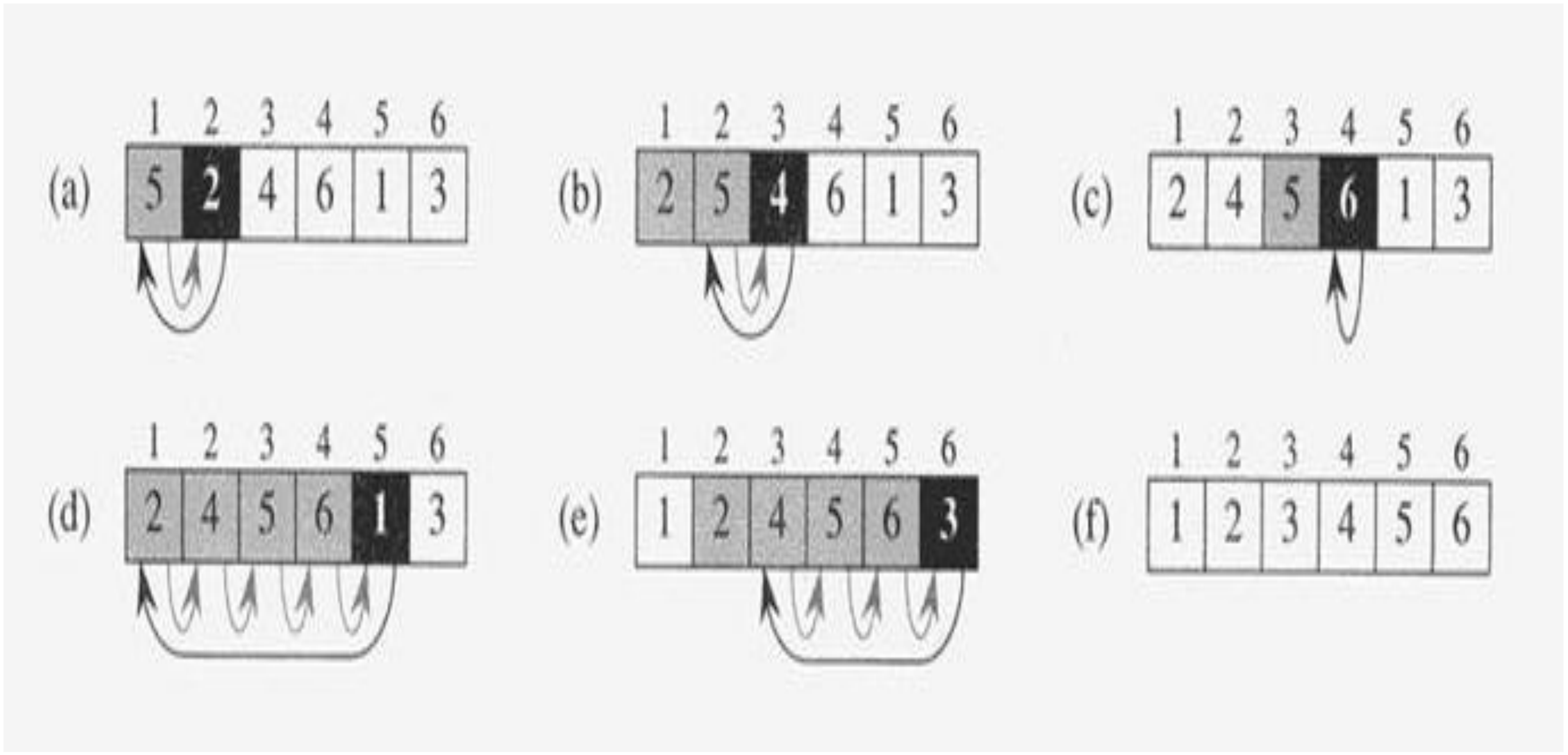
```
for j ← 1 to n - 1
    do smallest ← j
        for i ← j + 1 to n
            do if A[i] < A[smallest]
                then smallest ← i
        exchange A[j] ↔ A[smallest]
```

Key Points

- Complexity is O(n^2) for best, average and worst case.
- Stable and in place sorting algorithm. It outperforms bubble sort.
- Number of swap operations is very less (Linear).
- Independent of the initial ordering of the elements.
- As n increases, selection sort keeps slowing down. Movements are less. So when we have heavy file of records to be sorted based on some small index, then this type of sort may be useful. e.g. To reorganize your video files or music files, because cost of comparison will be negligible in terms of data movement.

Insertion Sort

- Insertion sort keeps making the left side of the array sorted until the whole array is sorted. It sorts the values seen far away and repeatedly inserts unseen values in the array into the left sorted array.
- It is the simplest of all sorting algorithms.


(a)
(b)
(c)
(d)
(e)
(f)

```
INSERTION-SORT(A, N)
   for j ← 2 to n
        do key ← A[ j ]
         //Insert A[ j ] into the sorted sequence A[1 . . j -1]
           i ← j - 1
           while i > 0 and A[i] > key
                do A[i + 1] ← A[i]
                     i ← i – 1
           A[i + 1] ← key
```

Insertion Sort Improvement

- Binary Search with in sorted sequence
- $O(n^2)$ in worst and average case. Inserting in between is costly because we have to move remaining elements.
- Good algorithm for small list up to 25-30 numbers. It is stable and inplace. It outperforms bubble and selection.
- Best case $O(n)$. Works well for nearly sorted lists.
- Good for cases where whole data is not available initially and new data values keep on adding in the list

Running Time

- Best Case: O(n), or O(1) if only 1 element
- Worst Case: O(*n*^2), if given in reverse order.
- Average Case: O(n^2), still a quadratic running time.

Best Times to Use Insertion Sort

- When the data sets are relatively small.
  - Moderately efficient.
- When you want a quick easy implementation.
  - Not hard to code Insertion sort.
- When data sets are mostly sorted already.
  - (1,2,4,6,3,2)

Worst Times to Use Insertion Sort

- When the data sets are relatively large.
  - Because the running time is quadratic
- When data sets are completely unsorted
  - Absolute worst case would be reverse ordered. (9,8,7,6,5,4)

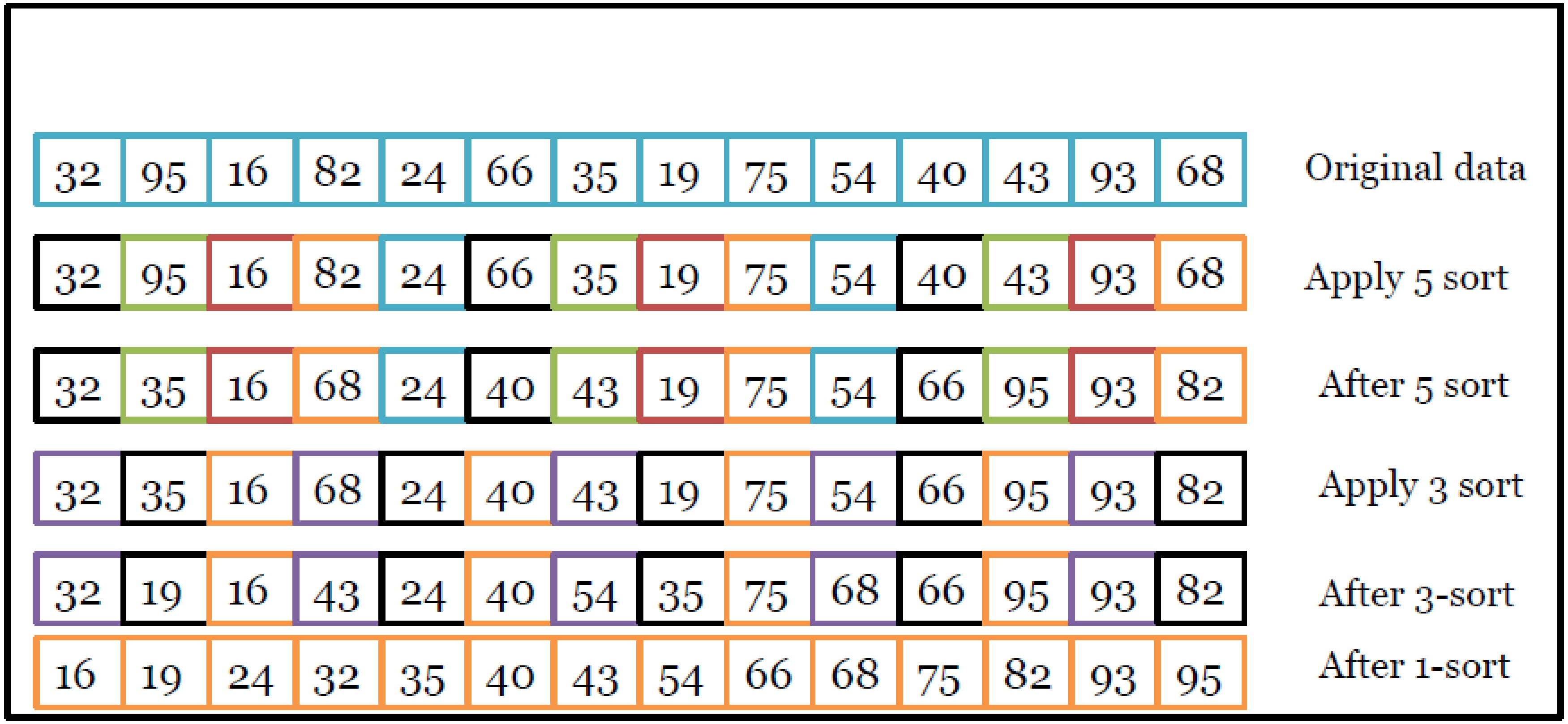
Insertion Sort Works in Place

- No extra data structures needed. It works off of original data structure that it is fed with and simply swaps the position of the items in the set.
- It does not require any extra memory as data sets get larger. Will always require the same amount of memory. M(1) – memory.
- It outperforms bubble and selection sort.

Shell Sort

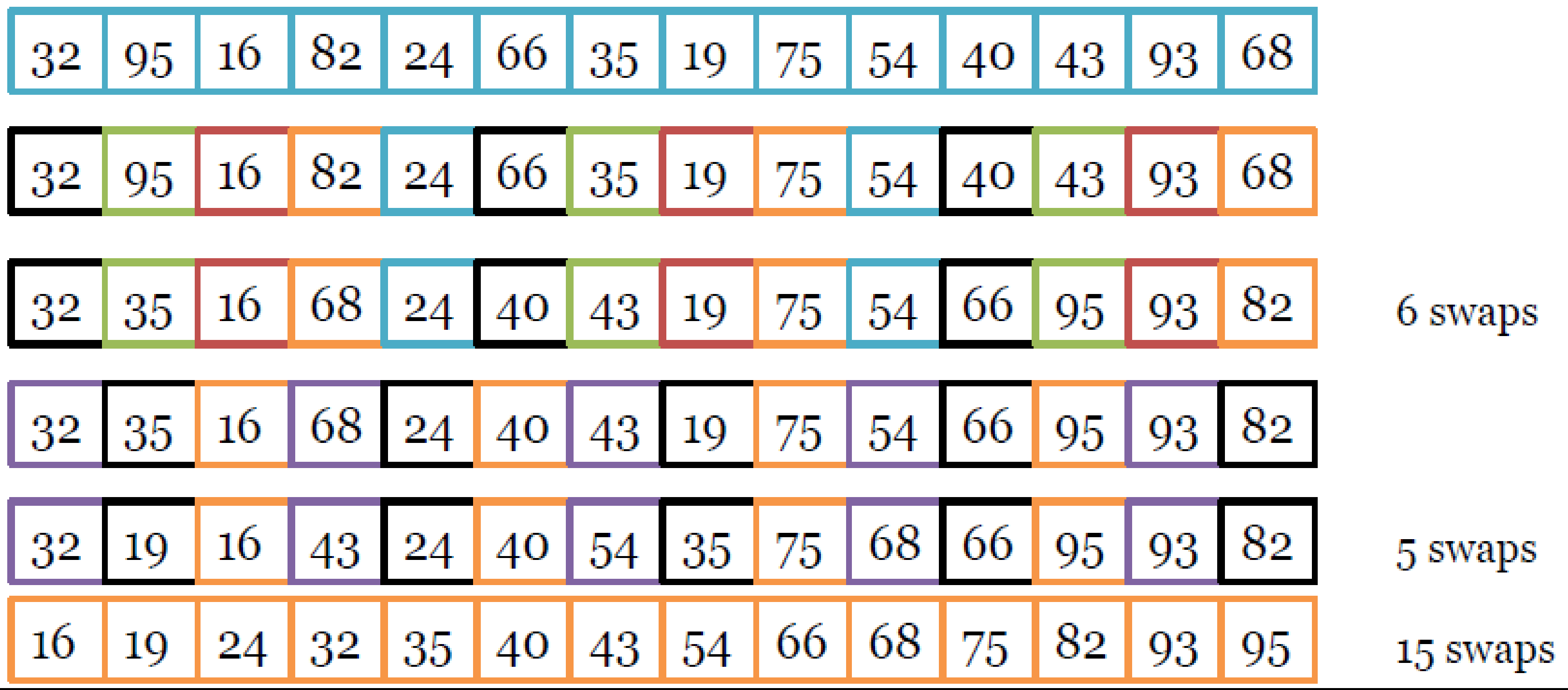
- Shell Sort performs better then Bubble, Selection or insertion sort. It is not a stable sort.
- Divides an array into several smaller non-contiguous segments.
- The distance between successive elements in one segment is called a *gap.*
- Each segment is sorted within itself using insertion sort.
- Then resegment into larger segments (smaller gaps) and repeat sort.
- Continue until only one segment (gap = 1) - final sort finishes array sorting.
- It is in place.
- The Worst case complexity is $O(n\log_2 n)$ or $O(n^{3/2})$ or others depending upon the Gap sequence.

Shell Sort Example



32	95	16	82	24	66	35	19	75	54	40	43	93	68	Original data
32	95	16	82	24	66	35	19	75	54	40	43	93	68	Apply 5 sort
32	35	16	68	24	40	43	19	75	54	66	95	93	82	After 5 sort
32	35	16	68	24	40	43	19	75	54	66	95	93	82	Apply 3 sort
32	19	16	43	24	40	54	35	75	68	66	95	93	82	After 3-sort
16	19	24	32	35	40	43	54	66	68	75	82	93	95	After 1-sort

Insertion sort will take 38 swaps while Shell sort is taking 26 swaps



32	95	16	82	24	66	35	19	75	54	40	43	93	68	
32	95	16	82	24	66	35	19	75	54	40	43	93	68	
32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 swaps
32	35	16	68	24	40	43	19	75	54	66	95	93	82	
32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 swaps
16	19	24	32	35	40	43	54	66	68	75	82	93	95	15 swaps

Shellsort Examples

- Sort: 18 32 12 5 38 33 16 2

8 Numbers to be sorted, Shell's increment will be floor(n/2)

*** floor(8/2) ➔ floor(4) = 4**

increment 4: **1 2 3 4** (visualize underlining)

18 32 12 5 38 33 16 2

Step **1**) Only look at **18** and **38** and sort in order ;
18 and **38** stays at its current position because they are in order.

Step **2**) Only look at **32** and **33** and sort in order ;
32 and **33** stays at its current position because they are in order.

Step **3**) Only look at **12** and **16** and sort in order ;
12 and **16** stays at its current position because they are in order.

Step **4**) Only look at **5** and **2** and sort in order ;
2 and **5** need to be switched to be in order.

Shellsort Examples (con't)

- Sort: 18 32 12 5 38 33 16 2

Resulting numbers after increment 4 pass:

18 32 12 *2* 38 33 16 *5*

*** floor(4/2) ➔ floor(2) = 2**

increment 2: **1 2**

18 32 12 2 38 33 16 5

Step **1**) Look at **18**, **12**, **38**, **16** and sort them in their appropriate location:

12 38 16 2 18 33 38 5

Step **2**) Look at **32**, **2**, **33**, **5** and sort them in their appropriate location:

12 2 16 5 18 32 38 33

Shellsort Examples (con't)

- Sort: 18 32 12 5 38 33 16 2

* floor(2/2) ➔ floor(1) = 1

increment 1: **1**

12	**2**	**16**	**5**	**18**	**32**	**38**	**33**
2	**5**	**12**	**16**	**18**	**32**	**33**	**38**

The last increment or phase of Shellsort is basically an Insertion Sort algorithm.

Shell Sort Pseudocode

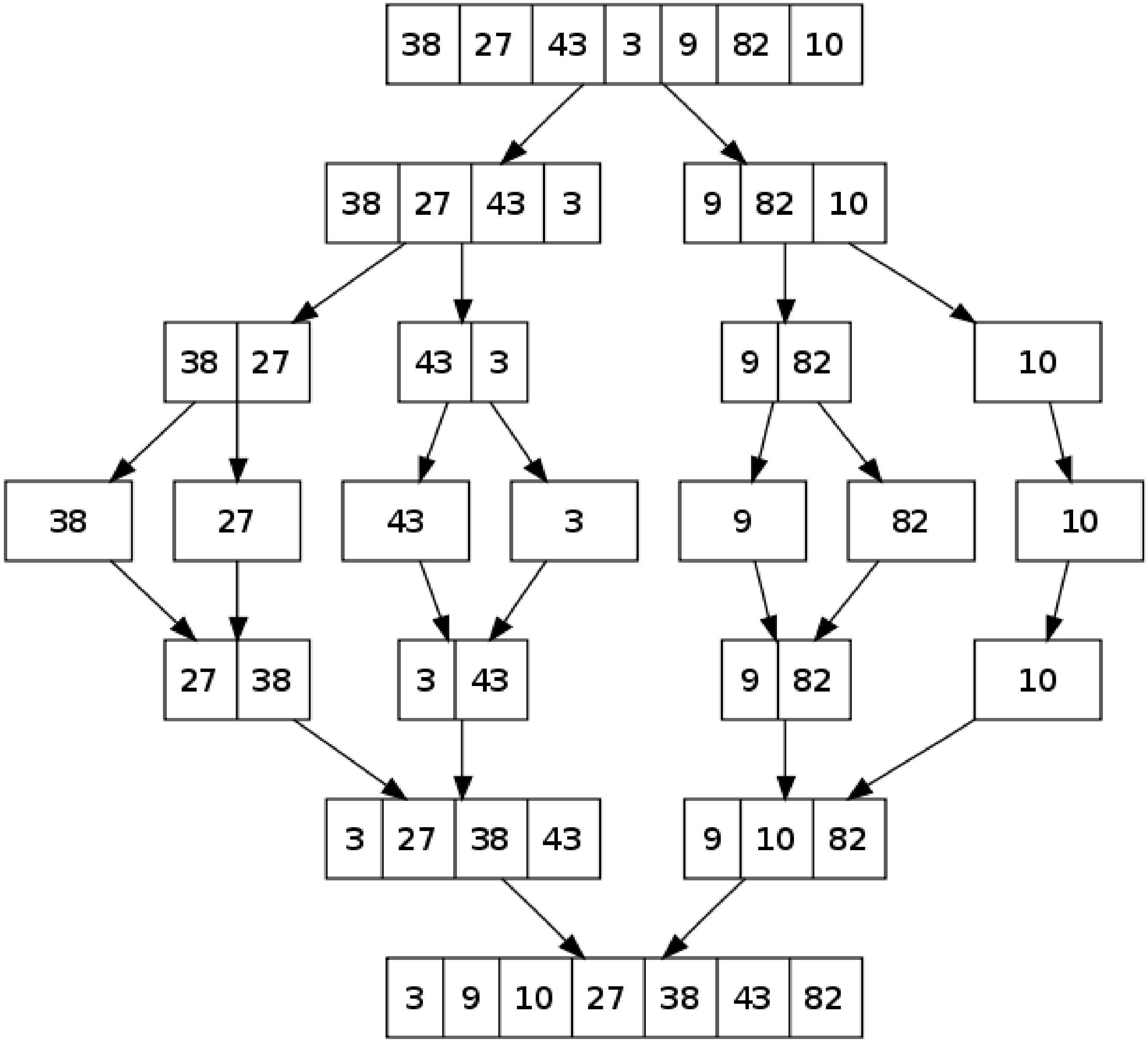
```
for (gap = N/2; gap > 0; gap = gap/2)
{
    for ( p = gap; p < N ; p++)
    {
        temp = a[p];
        for (j = p; j >= gap && temp < a[j- gap]; j = j - gap)
                a[j] = a[j-gap];

        a[j] = temp;
        }

}
```

Merge Sort

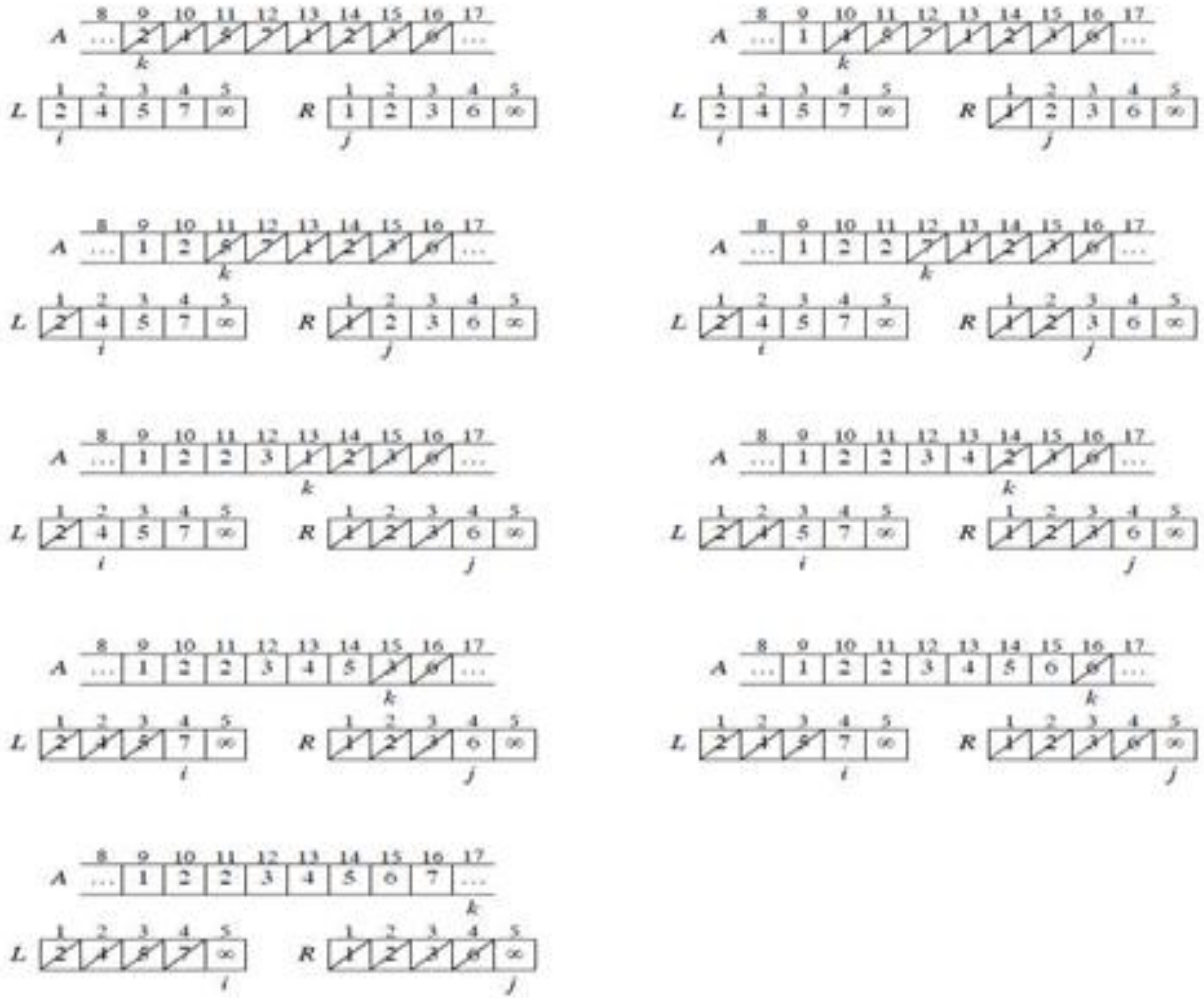
- Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e.,1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until atlast two lists are merged into the final sorted list.
- Its worst and average case running time is O(nlogn). It is better than the previously discussed algorithms.
- It can be implemented in place or out of place. It is a divide and conquer algorithm. It can be easily parallelized.



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($A, p, r$)
if $p < r$
    then $q \leftarrow \lfloor (p + r)/2 \rfloor$
        MERGE-SORT($A, p, q$)
        MERGE-SORT($A, q + 1, r$)
        MERGE($A, p, q, r$)
```

```
MERGE($A, p, q, r$)
$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
create arrays $L[1 \mathinner{.\,.} n_1 + 1]$ and $R[1 \mathinner{.\,.} n_2 + 1]$
for $i \leftarrow 1$ to $n_1$
    do $L[i] \leftarrow A[p + i - 1]$
for $j \leftarrow 1$ to $n_2$
    do $R[j] \leftarrow A[q + j]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow p$ to $r$
    do if $L[i] \le R[j]$
        then $A[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        else $A[k] \leftarrow R[j]$
            $j \leftarrow j + 1$
```



A
L
R
i
j
k
8 9 10 11 12 13 14 15 16 17
1 2 3 4 5

Quicksort

- Sorts in place
- Sorts O(n lg n) in the average case
- Sorts O(n^2) in the worst case
- *So why would people use it instead of merge sort?*

Quicksort

- Another divide-and-conquer algorithm
  - The array A[p..r] is *partitioned* into two non-empty subarrays A[p..q] and A[q+1..r]
    - Invariant: All elements in A[p..q] are less than all elements in A[q+1..r]
  - The subarrays are recursively sorted by calls to quicksort
  - Unlike merge sort, no combining step: two subarrays form an already-sorted array

Quicksort Code

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q);
        Quicksort(A, q+1, r);
    }
}
```

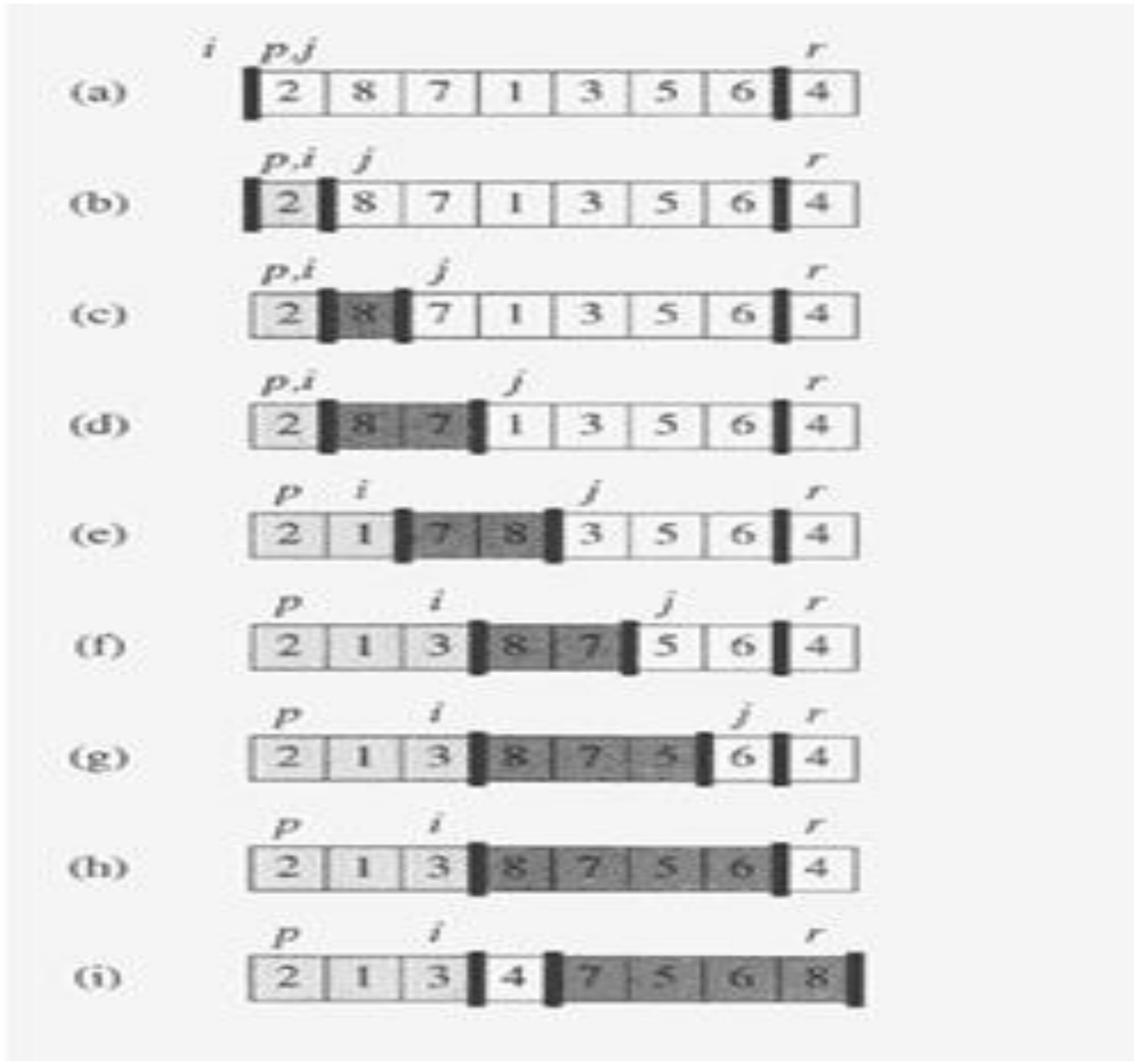
Partition

- Clearly, all the action takes place in the **`partition()`** function
  - Rearranges the subarray in place
  - End result:
    - Two subarrays
    - All values in first subarray ≤ all values in second
  - Returns the index of the “pivot” element separating the two subarrays
- *How do you suppose we implement this function?*

Partition In Words

- Partition(A, p, r):
  - Select an element to act as the "pivot" (*which?*)
  - Grow two regions, A[p..i] and A[j..r]
    - All elements in A[p..i] <= pivot
    - All elements in A[j..r] >= pivot
  - Increment i until A[i] >= pivot
  - Decrement j until A[j] <= pivot
  - Swap A[i] and A[j]
  - Repeat until i >= j
  - Return j

```
PARTITION($A, p, r$)
$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j \leftarrow p$ to $r - 1$
    do if $A[j] \leq x$
          then $i \leftarrow i + 1$
               exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$
```


(a) i p,j r
2 8 7 1 3 5 6 4
(b) p,i j r
2 8 7 1 3 5 6 4
(c) p,i j r
2 8 7 1 3 5 6 4
(d) p,i j r
2 8 7 1 3 5 6 4
(e) p i j r
2 1 7 8 3 5 6 4
(f) p i j r
2 1 3 8 7 5 6 4
(g) p i j r
2 1 3 8 7 5 6 4
(h) p i r
2 1 3 8 7 5 6 4
(i) p i r
2 1 3 4 7 5 6 8

Improving Quicksort

- The real liability of quicksort is that it runs in $O(n^2)$ on already-sorted input
- Book discusses two solutions:
  - Randomize the input array, OR
  - *Pick a random pivot element*
- *How will these solve the problem?*
  - By insuring that no particular input can be chosen to make quicksort run in $O(n^2)$ time