Sorting Algorithms Design and Complexity Analysis

Why we need sorting

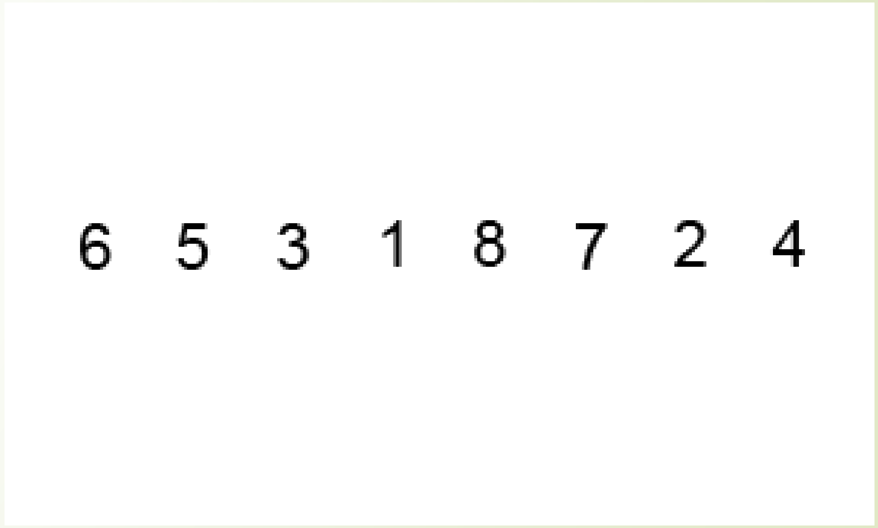
- What is sorting?
  - Resorting a set items in prescribed order
    - 7, 3, 8, 2, 5
  - Ascending 2, 3, 5, 7, 8
  - Descending 8, 7, 5, 3, 2
- Why we need sorting
  - In academia and industry
    - Better organization (phone book)
    - Make other problems easy to be solved (finding median)

Applications of sorting algorithms

- Data organization for better utilization
  - Phone book Faster access to contacts
  - Sorting flights on a screen at an air ports
  - Sorting items in BestBuy, eBay, amazon, Walmart
  - Which patient should be seen next at an Emergency Department?
  - Search engines (google.com , bing.com)
    - Top related (most relevant) sources in web
- Problems become easy with sorting
  - finding median
  - Finding duplicates in a set
  - Finding similar values (e.g., find the two numbers with the smallest difference)

Bubble sort

- Easy to implement and uses a comparison approach.
- Works by performing n-1 passed on the list and after each pass an element is deployed in its right location.
- like bubbles rising in a glass of soda, larger elements bubble up to the right of the list in their right place.
- Algorithm
  1. Compare each pair of adjacent elements from the beginning of a list and, if they are in reversed order, swap them.
  2. If at least one swap has been done, repeat step 1.



Example Bubble sort

7	2	5	9	1

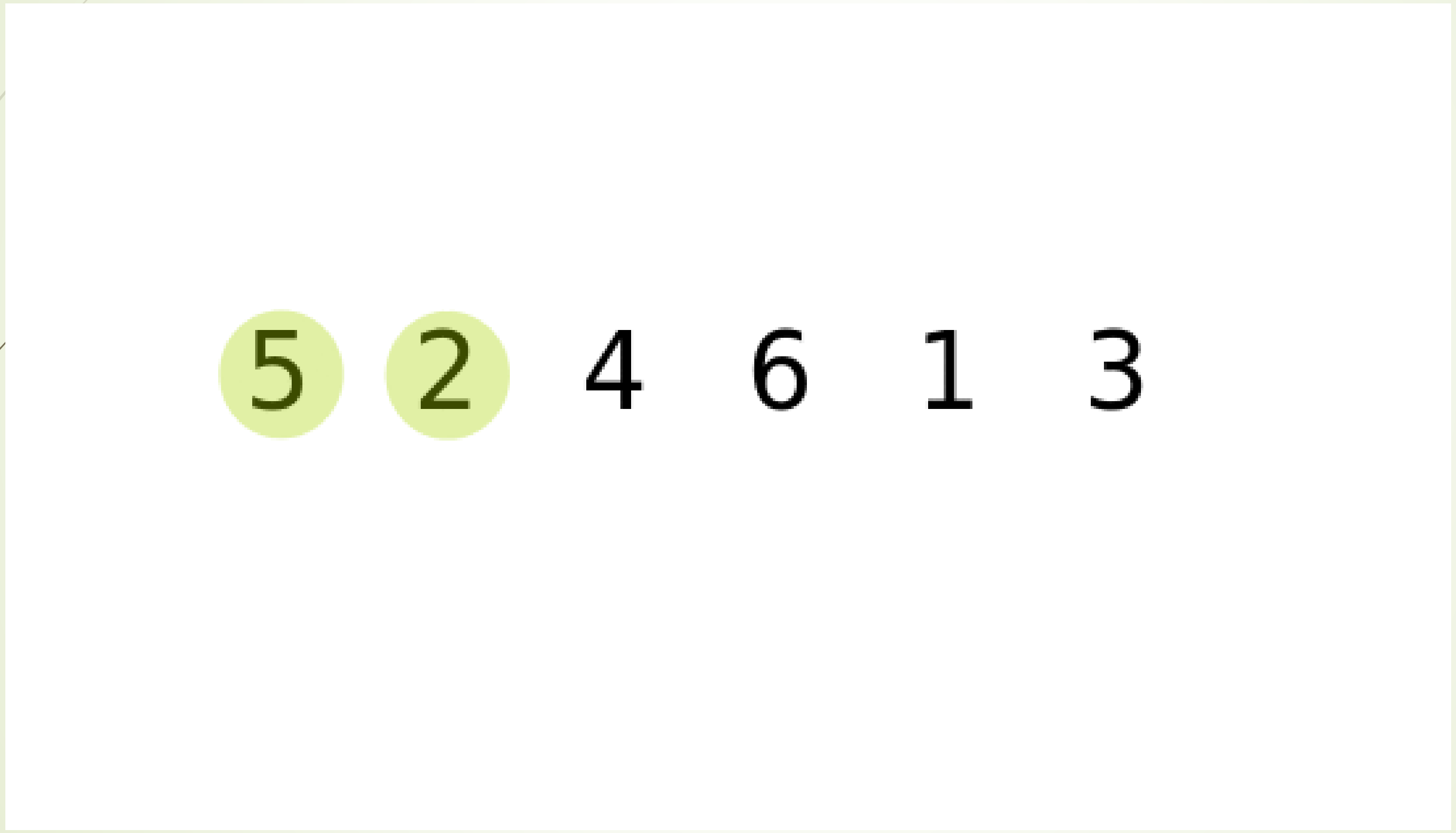
- **n**=5 (Needs n-1 passes.)

Pass1	pass3	pass3	pass4
7, 2, 5, 9, 1 **2, 7**, 5, 9, 1	**2, 5**, 7, 1, 9 2, **5, 7**, 1, 9	**2, 5**, 1, 7, 9 2, 5, 1, 7, 9	**2, 1**, 5, 7, 9 1, 2, 5, 7, 9
2, **7, 5**, 9, 1 2, **5, 7**, 9, 1	2, **5, 7**, 1, 9 2, 5, 7, 1, 9	2, **5, 1**, 7, 9 2, 1, 5, 7, 9	
2, 5, **7, 9**, 1 2, 5, **7, 9**, 1	2, 5, **7, 1**, 9 2, 5, 1, 7, 9		
2, 5, 7, **9, 1** 2, 5, 7, **1, 9**			

Sorte
d

1	2	5	7	9

Example Bubble sort



Bubble sort Pseudo-code

```
SEQUENTIAL BUBBLESORT (A)
for i ← 0 to (length [A]-2) do
    for j ← 0 to ((length [A]-2)- i) do
        If A[j+1] < A[j] then
            Exchange A[j] ↔ A[j+1] //swap
```

Here the number of comparison made

$$1 + 2 + 3 + \ldots + (n - 1) = n(n - 1)/2 = O(n^2)$$

About Bubble sort

- Inefficient for large data (not really used in practice)
- O(n2) sorting algorithm (discussed later)
  - Here the number of comparison made

    1 + 2 + 3 + . . . + (n - 1) = n(n - 1)/2 = O(n2)
- Stable Algorithm (discussed later)
- In-place
- Adaptive (It means that for almost sorted array it gives O(n) estimation)

Bubble sort Pseudo-code (different implementation)

```
SEQUENTIAL BUBBLESORT (A)
for i ← 0 to (length [A] – 1) do
    for j ← (length [A]-1) downto i +1 do
        If A[j] < A[j-1] then
            Exchange A[j] ↔ A[j-1] //swap
```

Here the number of comparison made

$$1 + 2 + 3 + . . . + (n - 1) = n(n - 1)/2 = O(n^2)$$

Example Bubble sort

7	2	5	9	1

- **n**=5 (Needs n-1 passes.)

Pass1	pass3	pass3	pass4
7, 2, 5, **9, 1** 7, 2, 5, 1, 9	1, 7, 2, 5,9 1, 7, 2, 5, 9	1, 2, 7, 5,9 1, 2, 7, 5, 9	1, 2, 5, 7,9 1, 2, 5, 7, 9
7, 2, 5, **1, 9** 7, 2, 1, 5, 9	1, 7, 2, 5, 9 1, 7, 2, 5, 9	1, 2, 7, 5, 9 1, 2, 5, 7, 9	
7, **2, 1**, 5, 9 7, 1, 2, 5, 9	1, 7, 2, 5, 9 1, 2, 7, 5, 9		
7, 1, 2, 5, 9 1, 7, 2, 5, 9		Sorted	

Sorted

1	2	5	7	9

About

- Inefficient for large data and not really used in practice (very rarely)
- O(n2) sorting algorithm
  - Here the number of comparison made

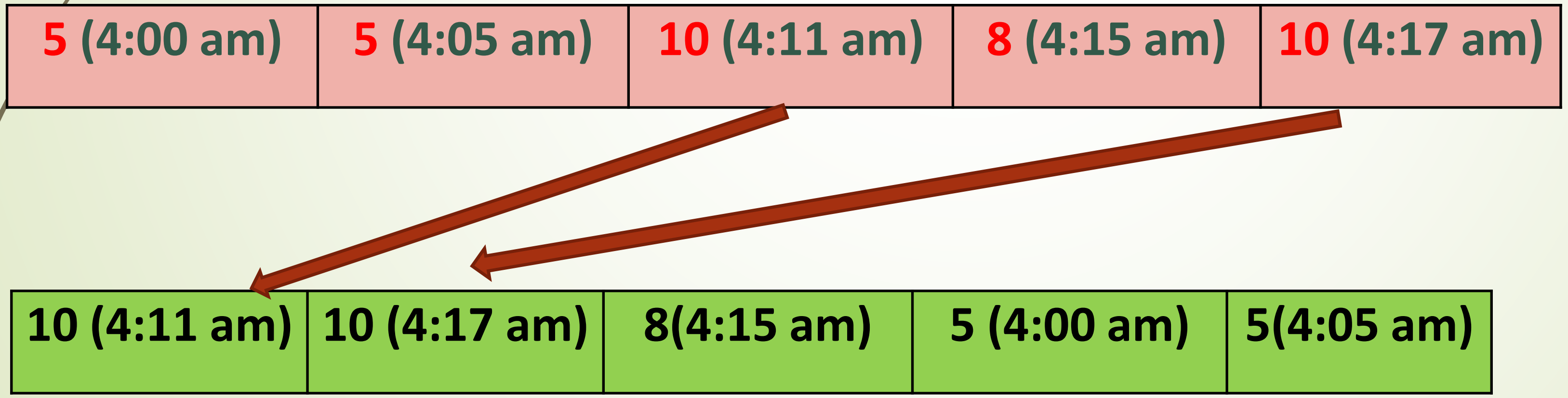
    1 + 2 + 3 + . . . + (n - 1) = n(n - 1)/2 = O(n2)
- Stable Algorithm
- Adaptive (It means that for almost sorted array it gives O(n) estimation)

Stability

If two items have the same key as each other, they should have the same relative position in the output as they did in the input.

Example: The process of determining the next patient to be seen at Emergency Department (when resources are insufficient for all to be treated immediately)

Emergency Severity Index EDI (time stamp)

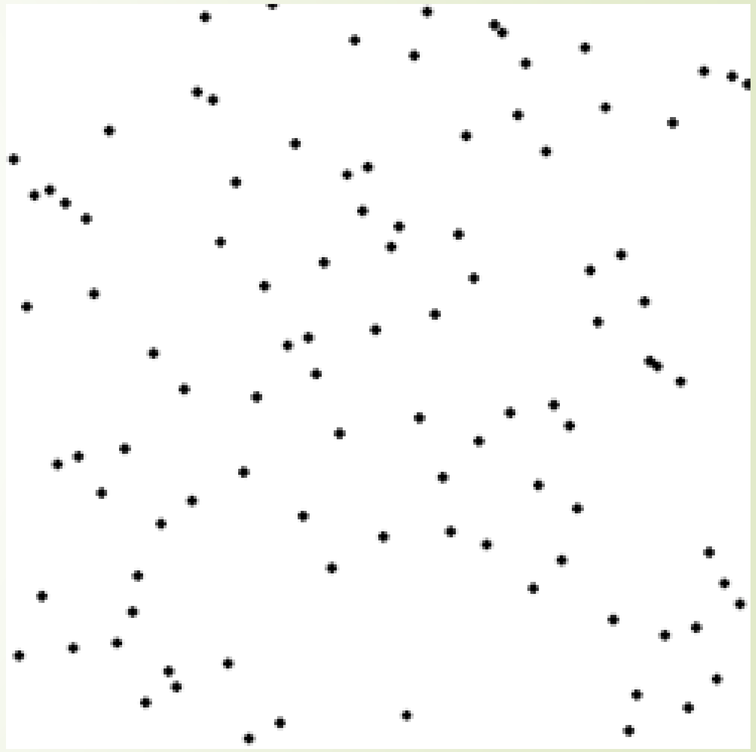


5 (4:00 am)	5 (4:05 am)	10 (4:11 am)	8 (4:15 am)	10 (4:17 am)

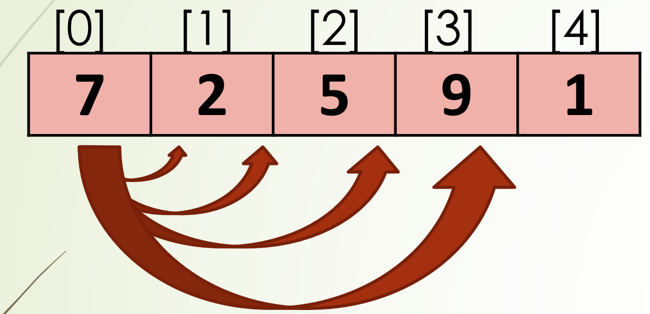
10 (4:11 am)	10 (4:17 am)	8(4:15 am)	5 (4:00 am)	5(4:05 am)

Selection Sort

- Works by finding the minimum value and swap it with first location
- finds the next minimum and swap it with the second element.
- Continue the same process until all list is sorted.



Example Selection Sort



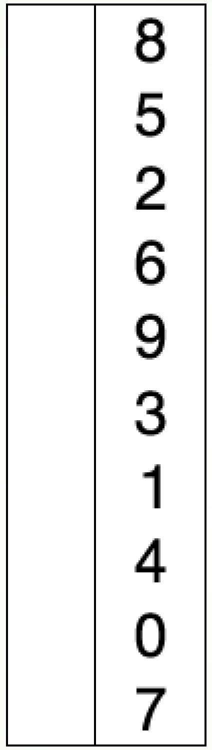
unsorted

Pass1	pass3	pass3	pass4
7, 2, 5, 9, 1 Min in index=4 1, 2, 5, 9, 7	1, 2, 5, 9, 7 index=1 1, 2, 5, 9, 7	1, 2, 5, 9, 7 index=2 1, 2, 5, 9, 7	1, 2, 5, 9, 7 Index=4 1, 2, 5, 7, 9

sorted

1	2	5	7	9

Example Selection Sort



	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Pseudo-code - Selection sort

```
1.  for j ← 0 to n-2
2.    smallest ← j
3.      for i ← j + 1 to n-1                  ⎫
4.      if A[ i ] < A[ smallest ]             ⎬ Finding the minimum
5.              smallest ← i                  ⎭
6.    Exchange A[ j ] ↔ A[ smallest ]
```

➢Inefficient in terms of time complexity (will discuss that later on)

About Selection Sort

- In-place (insignificant extra memory)
- Comparison based algorithm
- Inefficient for large data
  - huge time proportional to size of the data ($O(n^2)$ sorting algorithm)
  - Here the number of comparison made

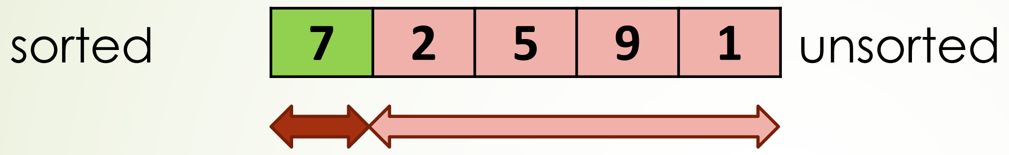
    $1 + 2 + 3 + . . . + (n - 1) = n(n - 1)/2$ ----> $O(n^2)$
- Not Stable
  - (can be stable with little change in the implementation)

Why Selection Sort Is Not Stable!!!

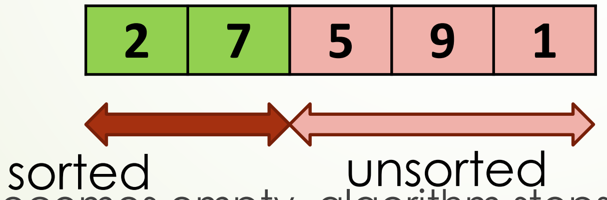
- Selection sort exchanges elements from the front of the list with the minimum element, which can change the original order of the elements.
- For example, (**EDI**, Time)
- (8, 4:10), (8, 4:15), (10, 4:12)
- Selection sort first swaps the (10, 4:12) to the front:
- (10, 4:12), (8, 4:15), (8, 4:10)
- And now the (8, 4:15) and (8, 4:15) are out of order from where they started in the sort.
- Can be stable if an insertion technique is used instead of swapping in a small time

Insertion sort

- Applied in practice for sorting small lists of data.
- List is imaginary divided into two parts - sorted one and unsorted one.

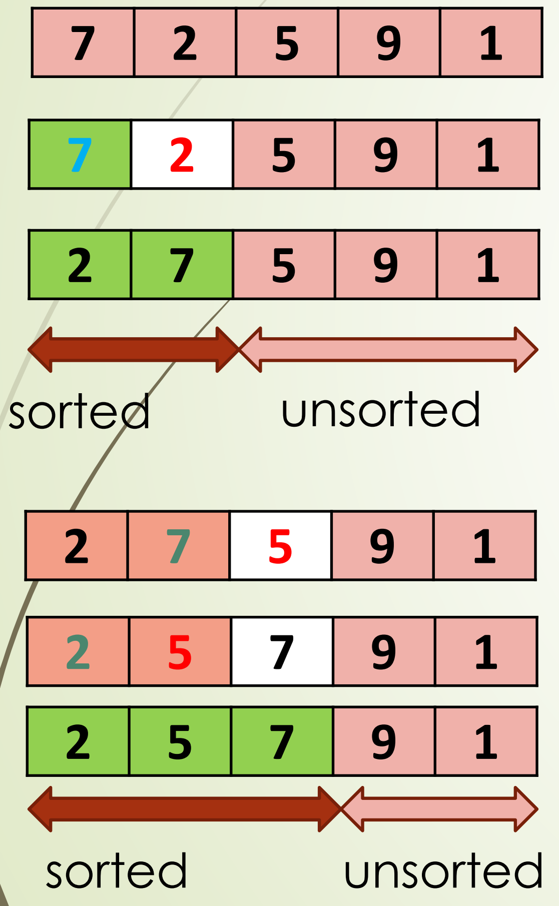


- At the beginning, sorted part contains first element of the array and unsorted one contains the rest.
- At every step, algorithm takes first element in the unsorted part and inserts it into its correct position in the sorted one.



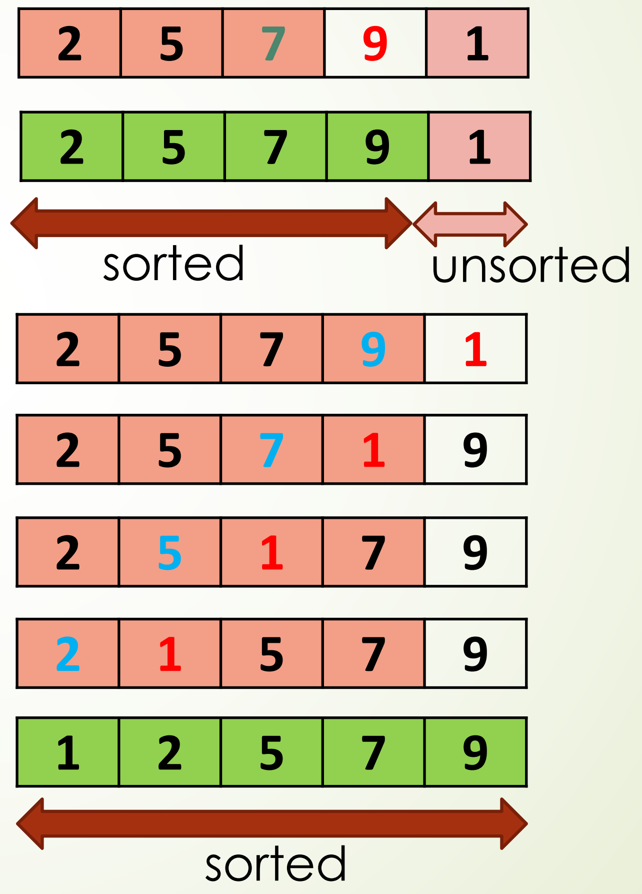
- When unsorted part becomes empty, algorithm stops.

Example – insertion sort

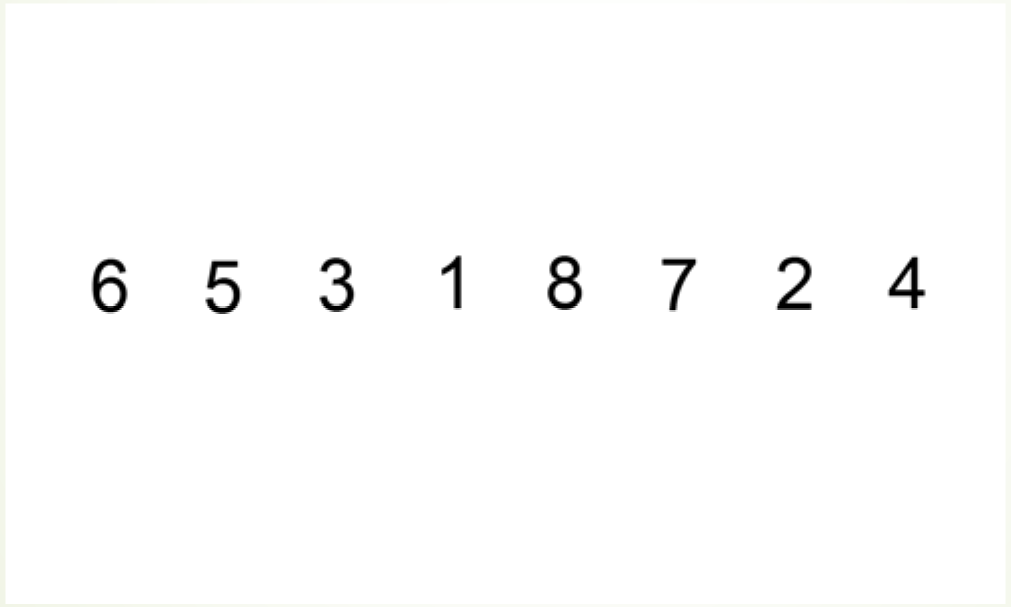


2 to be inserted
7>2 ---> swap
Reached left boundary so insert 2

Insert 5



Example – insertion sort



Insertion sort, pseudo-code

```
1:  for i ← 1 to i < list.size
 2:     j ← i
 3:         while j > 0 and list[j - 1] > list[j]
 4:                 tmp ← list[j];          ┐
 5:                 list[j] ← list[j - 1];   ├ // Swap (list[j] and list[j-1])
 6:                 list[j - 1] ← tmp;      ┘
 7:         j ← j-1
```

About Insertion Sort

- Inefficient for large data
  - O(n2) sorting algorithm (huge time proportional to size of the data)
  - Here the number of comparison made

    $1 + 2 + 3 + . . . + (n - 1) = n(n - 1)/2$ ----> $O(n^2)$
- Stable
- In-place (No significant extra memory)
- Time is proportional to the number of the inversions
  - Better when list is almost sorted o(k+n) : k is inversions number.

Which Algorithm Is Better For Me?

- Time complexity
  - Dependency relation between size of data and time (time proportional to size of data)
  - How time evolve if size of data is increased?
- Space complexity
  - In-place (insignificant amount of extra storage)
  - Auxiliary storage (grows with input size)
- Stability
  - Preserve original ranks for items with same key values
- Others
    - Recursive, comparison, linear, internal vs external storage

Summary of Sorting Algorithms

Algorithm	Time	Notes
bubble-sort	$O(n^2)$	-in-place -slow (good for small inputs)
selection-sort	$O(n^2)$	-in-place -slow (good for small inputs)
insertion-sort	$O(n^2)$	-in-place -slow (good for small inputs)

Linear time sorting algorithms

- Counting sort and radix sort assume that the input consists of integers in a small range.
- Despite of linear time usually these algorithms are not very desirable from practical point of view
  - the efficiency of linear-time algorithms depend on the keys randomly ordered. If this condition is not satisfied, the result is the degrading in performance.
  - these algorithms require extra space proportional to the size of the array being sorted and largest value of the elements, so if we are dealing with large file, extra array becomes a real liability

Counting Sort

- it is an integer sorting algorithm.
- Note that Counting sort beats the lower bound of $\Omega(n \lg n)$, because it is not a comparison sort. $O(n + k)$
- Counting sort uses the actual values of the elements to index into an array.
- The basic idea of Counting sort is to determine, for each input elements x, the number of elements less than x.
- Not in-place Because it uses arrays of length $k + 1$ and n, the total space usage of the algorithm is also $O(n + k)$

Example, Counting Sort

```
for j ← n-1 downto 0 do
    B[c[A[j]] -1] ← A[j]
    c[A[j]] ← c[A[j]] - 1
```

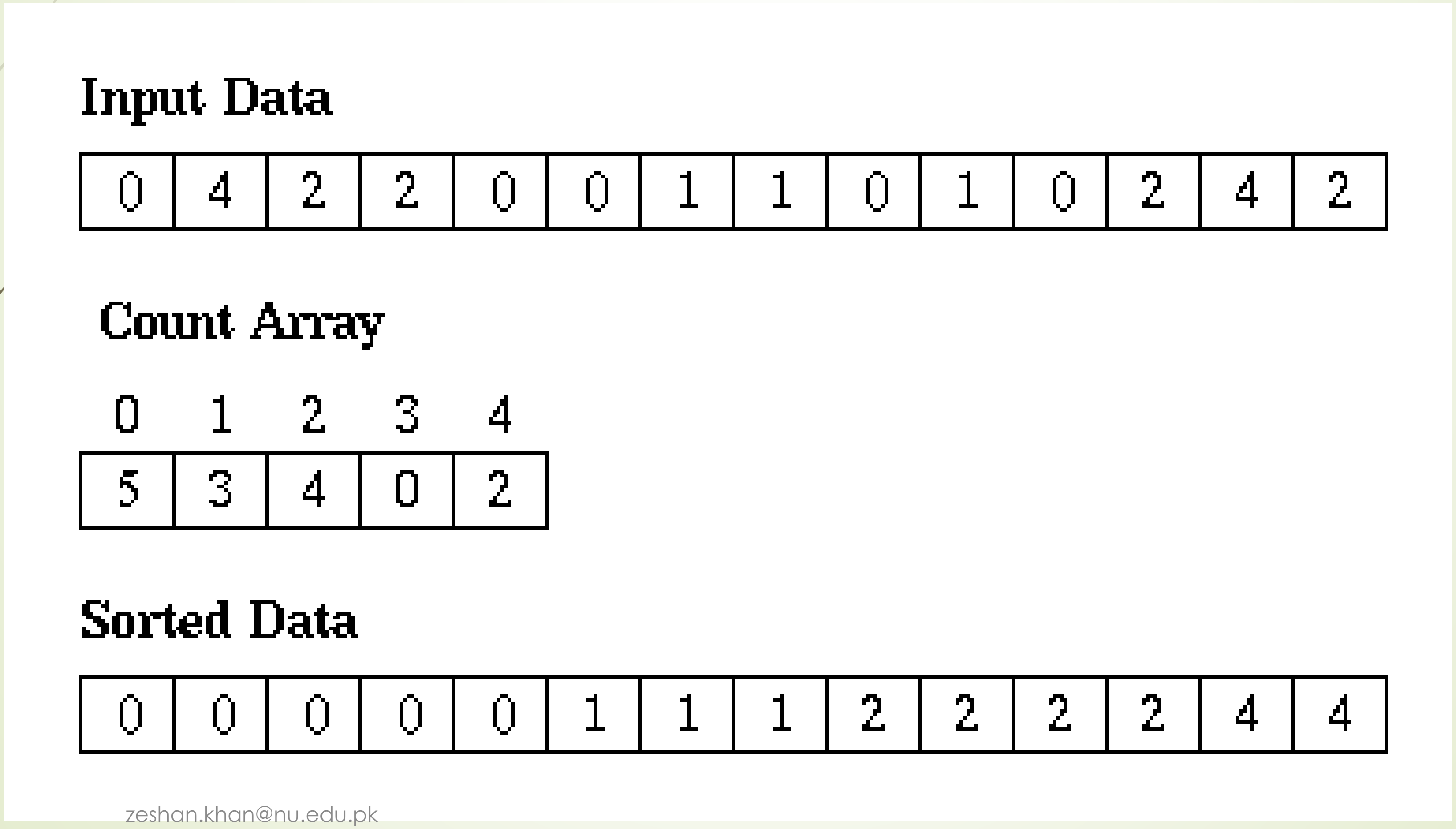
A	3	2	3	1	1
B	-	1	-	-	-
B	1	1	-	-	-
B	1	1	-	-	3
B	1	1	2	-	3
B	1	1	2	3	3

	C[0]	[1]	[2]	[3]
C	0	2	1	2
C	0	2	3	5
C	0	1	3	5
C	0	0	3	5
C	0	0	3	4
C	0	0	2	4
C	0	0	2	3

```
for j ← 0 to Max do
    c[A[j]] ← c[A[j]] + 1
```

```
for j ← 1 to Max do
    c[A[j]] ← c[A[j]] + 1
```

Example, Counting Sort



Input Data

0	4	2	2	0	0	1	1	0	1	0	2	4	2

Count Array

0	1	2	3	4
5	3	4	0	2

Sorted Data

0	0	0	0	0	1	1	1	2	2	2	2	4	4

Pseudo-code, counting sort

- **COUNTING_SORT (*A*, *B*, *k*)**

```
1. for $i \leftarrow 0$ to $k-1$ do
2.     $c[i] \leftarrow 0$
3. for $j \leftarrow 0$ to $n-1$ do
4.     $c[A[j]] \leftarrow c[A[j]] + 1$
//c[i] now contains the number of elements equal to i
5. for $i \leftarrow 1$ to $k-1$ do
6.     $c[i] \leftarrow c[i] + c[i-1]$
// c[j] now contains the number of elements $\leq j$
7. for j $\leftarrow n-1$ downto 0 do
8.     $B[c[A[j]] -1] \leftarrow A[j]$
9.     $c[A[j]] \leftarrow c[A[j]] - 1$
```

Analysis

The loop of lines 1-2 takes $O(k)$ time

The loop of lines 3-4 takes $O(n)$ time

The loop of lines 5-6 takes $O(k)$ time

The loop of lines 7-9 takes $O(n)$ time

Overall time of the counting sort is

$O(k) + O(n) + O(k) + O(n) =$

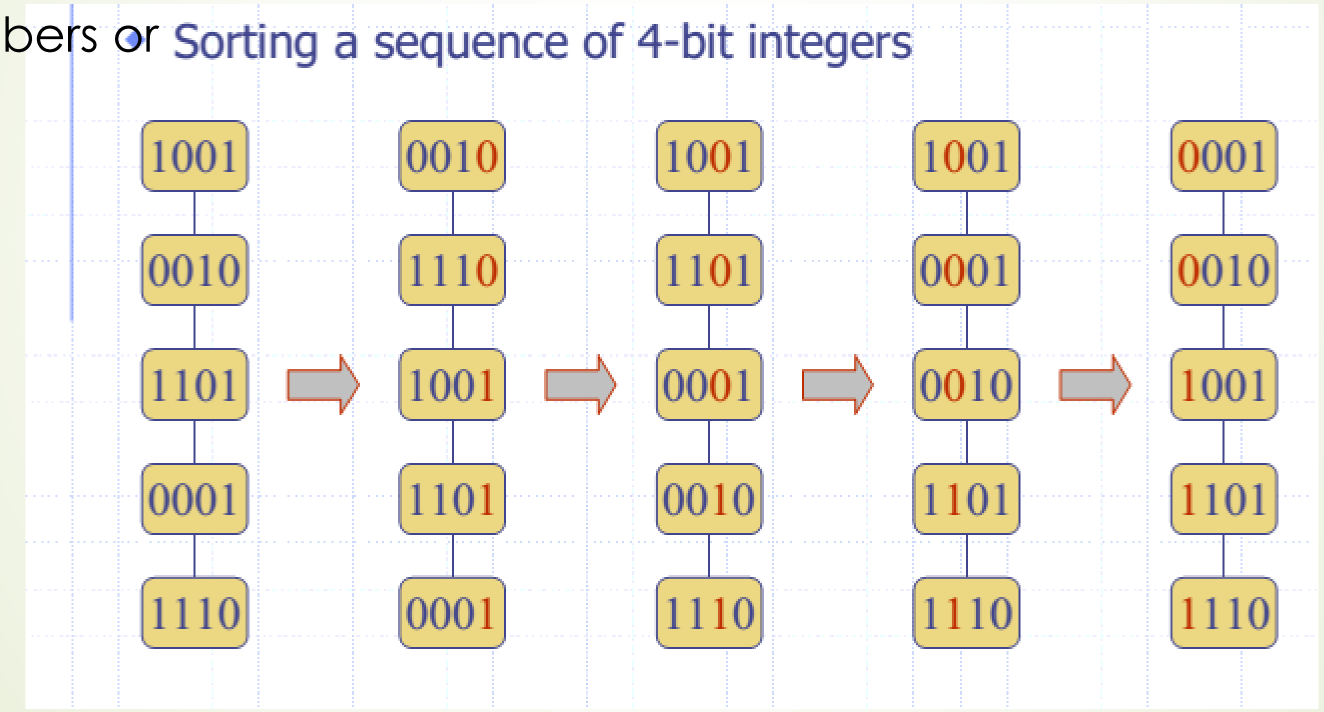
$O(k + n)$

Summary of Sorting Algorithms

Algorithm	Time	Notes
bubble-sort	$O(n^2)$	-in-place -slow (good for small inputs)
selection-sort	$O(n^2)$	-in-place -slow (good for small inputs)
insertion-sort	$O(n^2)$	-in-place -slow (good for small inputs)
counting sort	$O(n + k)$: k is max items value	-not a comparison sort -Work with integers

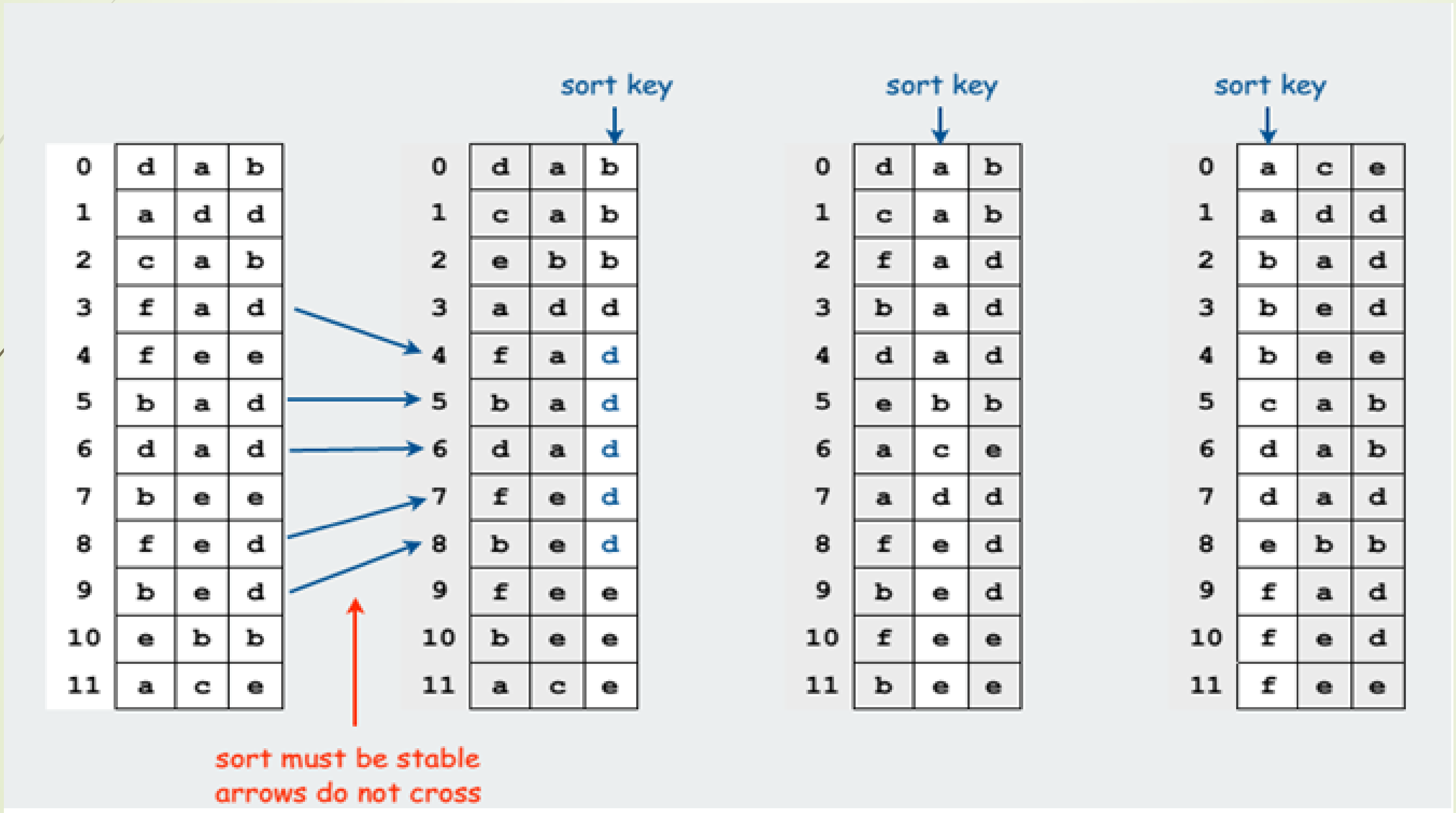
Radix Sort

Linear time
Works with numbers or data in form of numbers.
Stable
O(n+k)



***Algorithms design. By Goodrich

Least significant digit radix sort



0	d	a	b
1	a	d	d
2	c	a	b
3	f	a	d
4	f	e	e
5	b	a	d
6	d	a	d
7	b	e	e
8	f	e	d
9	b	e	d
10	e	b	b
11	a	c	e

0	d	a	b
1	c	a	b
2	e	b	b
3	a	d	d
4	f	a	d
5	b	a	d
6	d	a	d
7	f	e	d
8	b	e	d
9	f	e	e
10	b	e	e
11	a	c	e

0	d	a	b
1	c	a	b
2	f	a	d
3	b	a	d
4	d	a	d
5	e	b	b
6	a	c	e
7	a	d	d
8	f	e	d
9	b	e	d
10	f	e	e
11	b	e	e

0	a	c	e
1	a	d	d
2	b	a	d
3	b	e	d
4	b	e	e
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	d
11	f	e	e

Practical observation

- Elements to be ordered are usually objects/structures (struct) made of many variables (members/fields)
- The key of such object/structure is usually one member/field (or a value calculated from one or more members/fields)
- Remaining members/fields are additional data but usually useless for ordering
- Ordering is made for increasing values (ascending order) of the key
- People tend to use comparison based sorting algorithms although other algorithm types some times faster e.g., Radix. (avoid complex implementations)

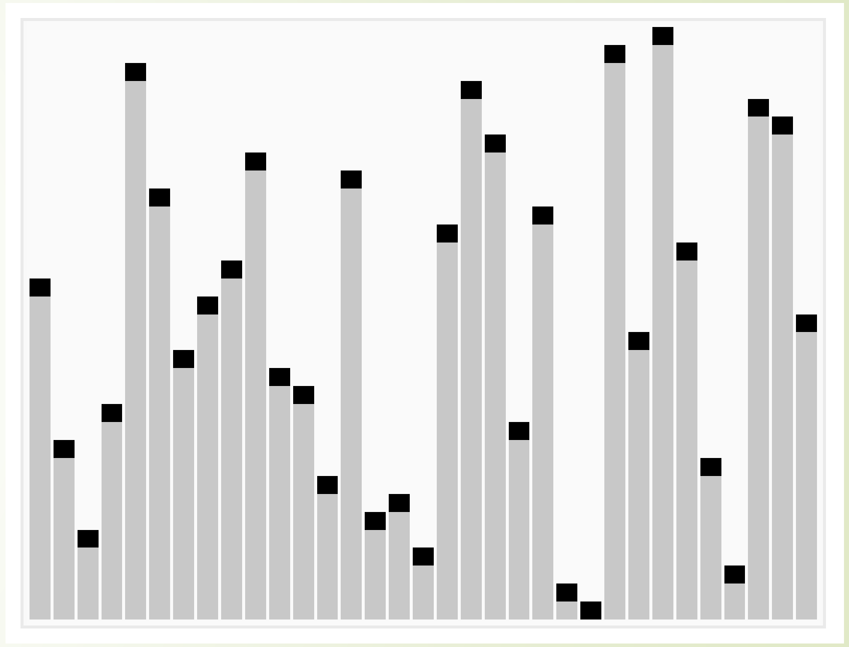
Summary of Sorting Algorithms

Algorithm	Time	Notes
bubble-sort	$O(n^2)$	-in-place -slow (good for small inputs)
selection-sort	$O(n^2)$	-in-place -slow (good for small inputs)
insertion-sort	$O(n^2)$	-in-place -slow (good for small inputs)
counting sort	$O(n + k)$: k is max items value	-not a comparison sort -Work with integers
radix sort	$O(n + k)$	-not a comparison sort

Quick sort

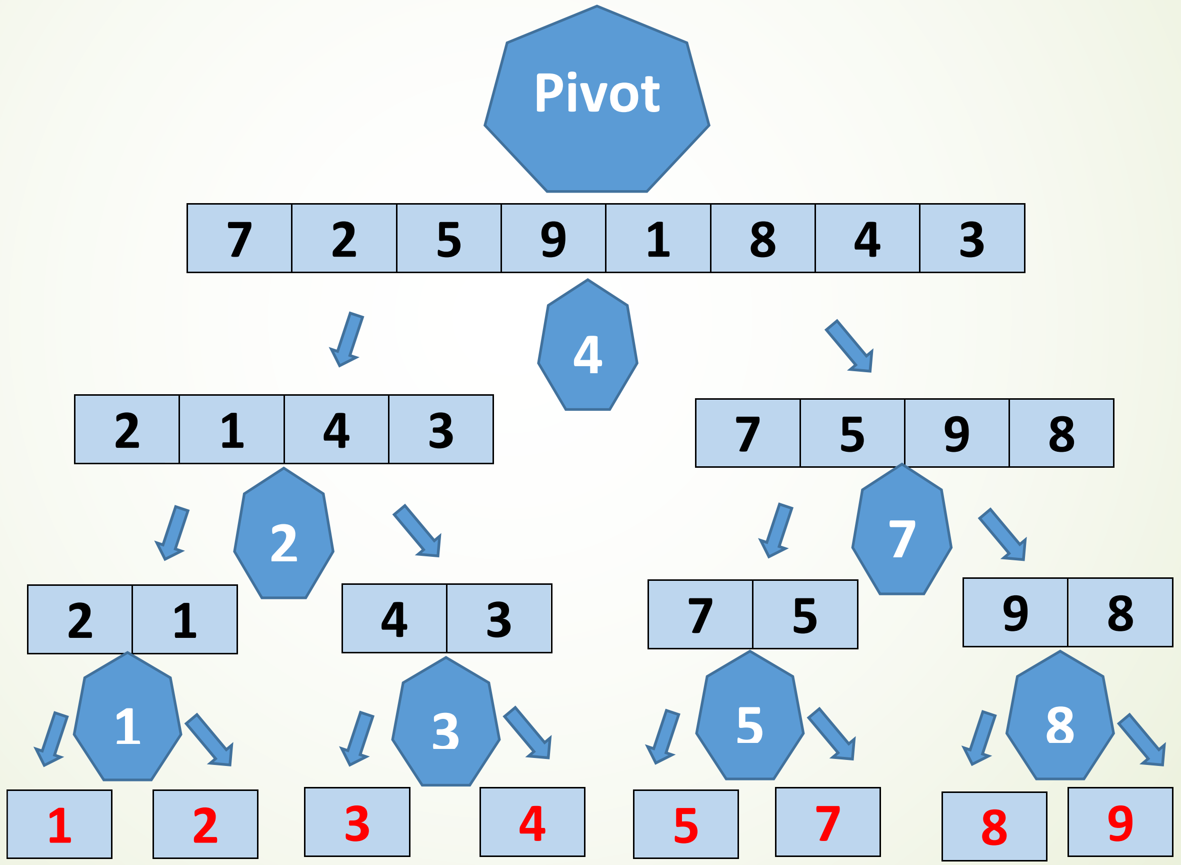
Quicksort is a divide and conquer algorithm.

- Algorithm steps:
  - Select an element (pivot)
  - partition operation: two sub lists (left and right)
  - Left contain all elements less than pivot and right sub list holds values greater than pivot
  - Recursively apply the above steps to the sub-list of elements with smaller values and separately to the sub-list of elements with greater values.



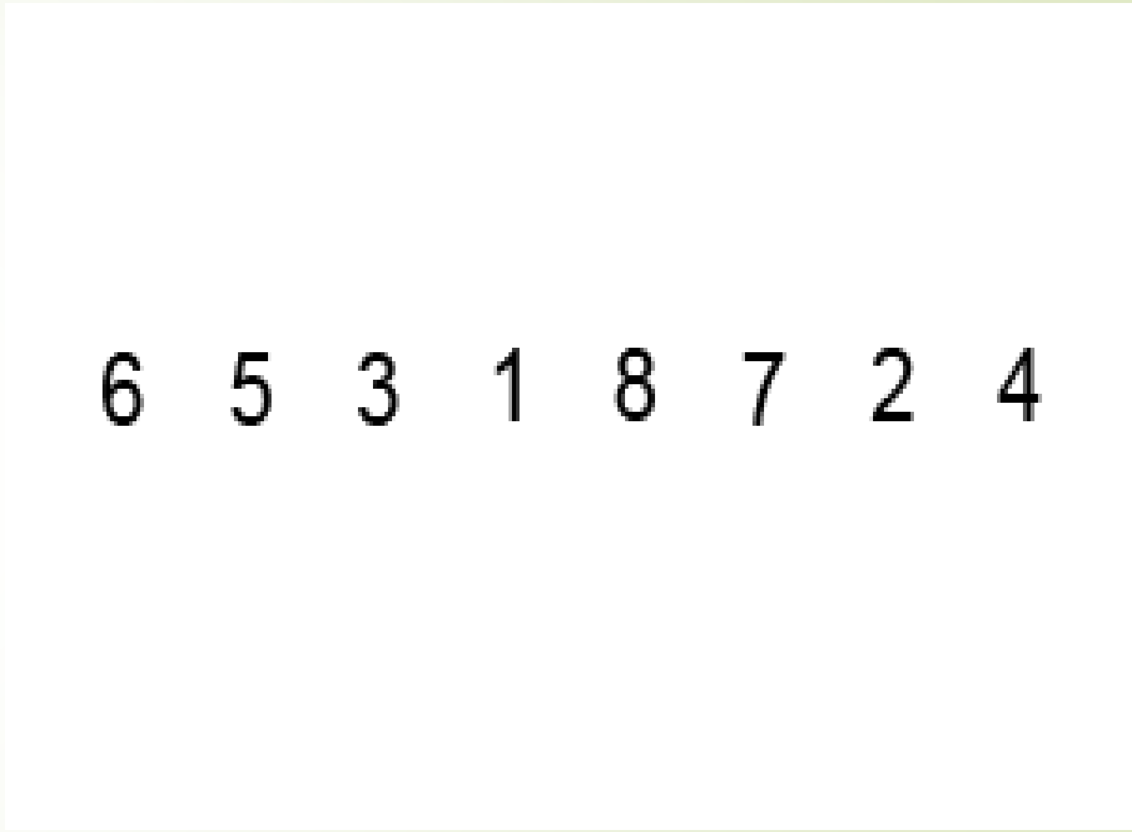
Example Quick Sort

- For simplicity, assume that Pivot partitions the list evenly (not realistic)



Pseudo-code

```
Sort(A,start,end)
    If(start<end)
        Mid=Partition(A,start,end)
        Sort(A,start,mid)
        Sort(A,mid+1,end)
Partition(A,start,end)
    Pivot=A[start]
    Left=start
    For(i=start;i<end;i++)
        A[i]<pivot?swap(A[i],A[left++]);
    Swap(pivot,A[left])
    Return left
```

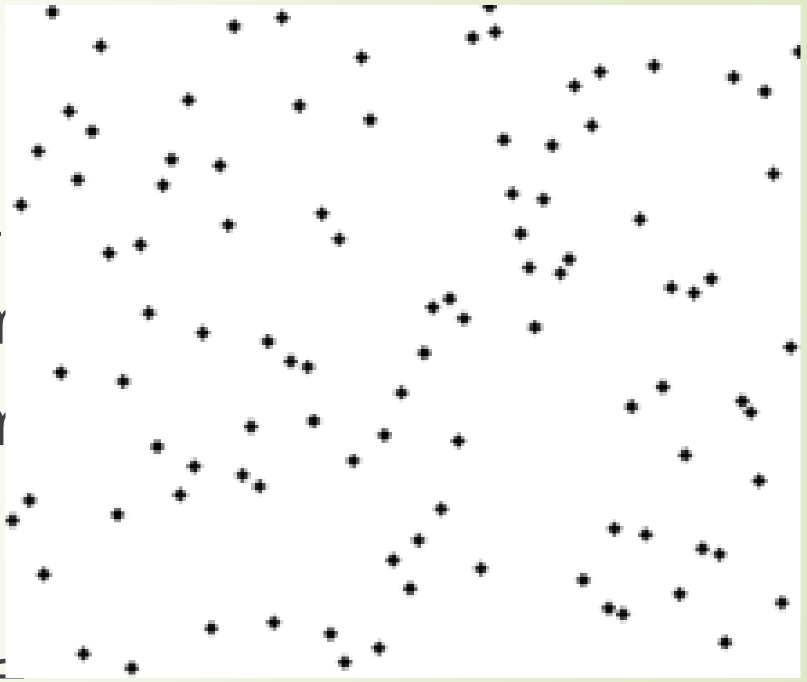


About Quick Sort

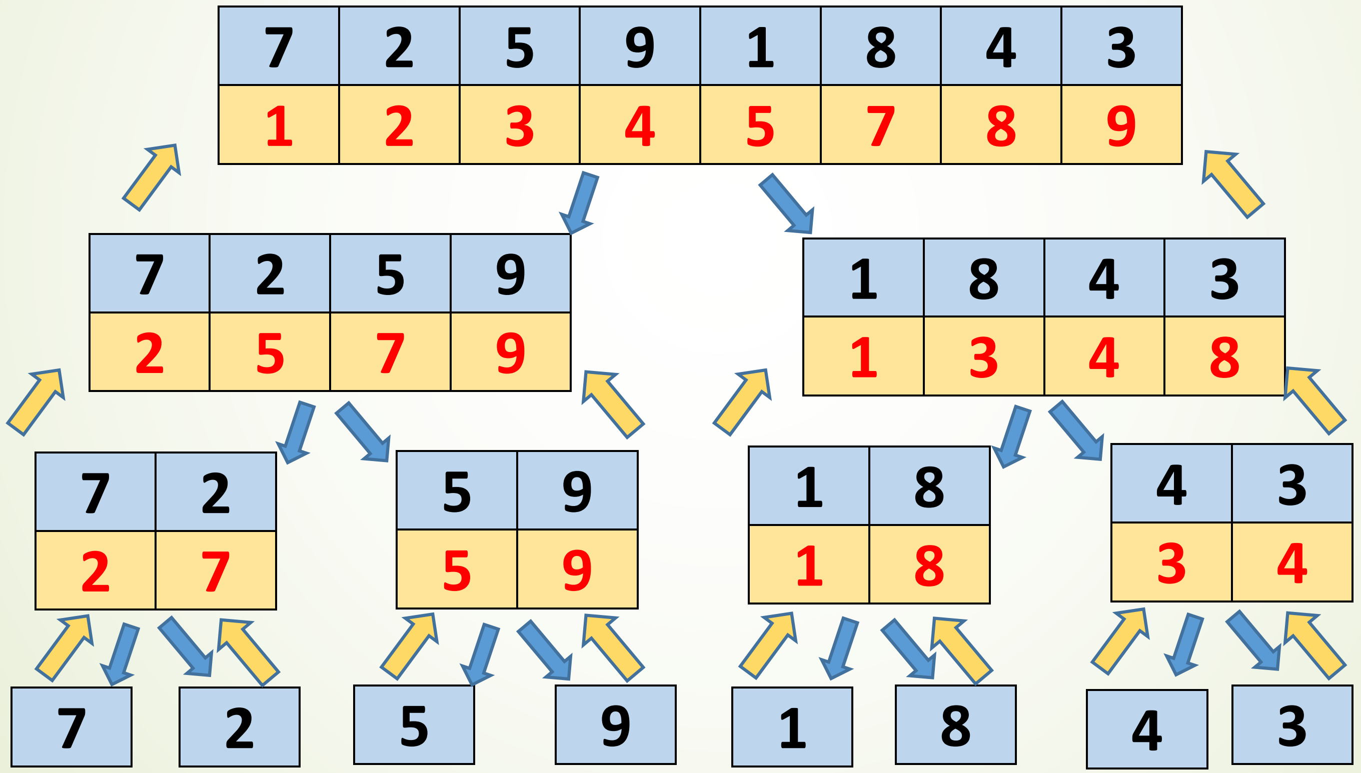
- Divide and Conquer (Recursive)
- Time complexity
  - Average o(n log n)
  - Worst case o(n^2)
- in-place o(log n) with array implementation
- Not Stable Algorithm
- Can be improved with randomization and multi partitioning e.g.,3

Merge sort

- Divide and Conquer (Recursively)
  - Divide the problem into subproblems that are smaller instances of the same problem
  - Conquer the subproblems by solving them recursively. If the subproblems are small enough, solve them trivially.
  - Combine the subproblem solutions to give a solution to the original problem.

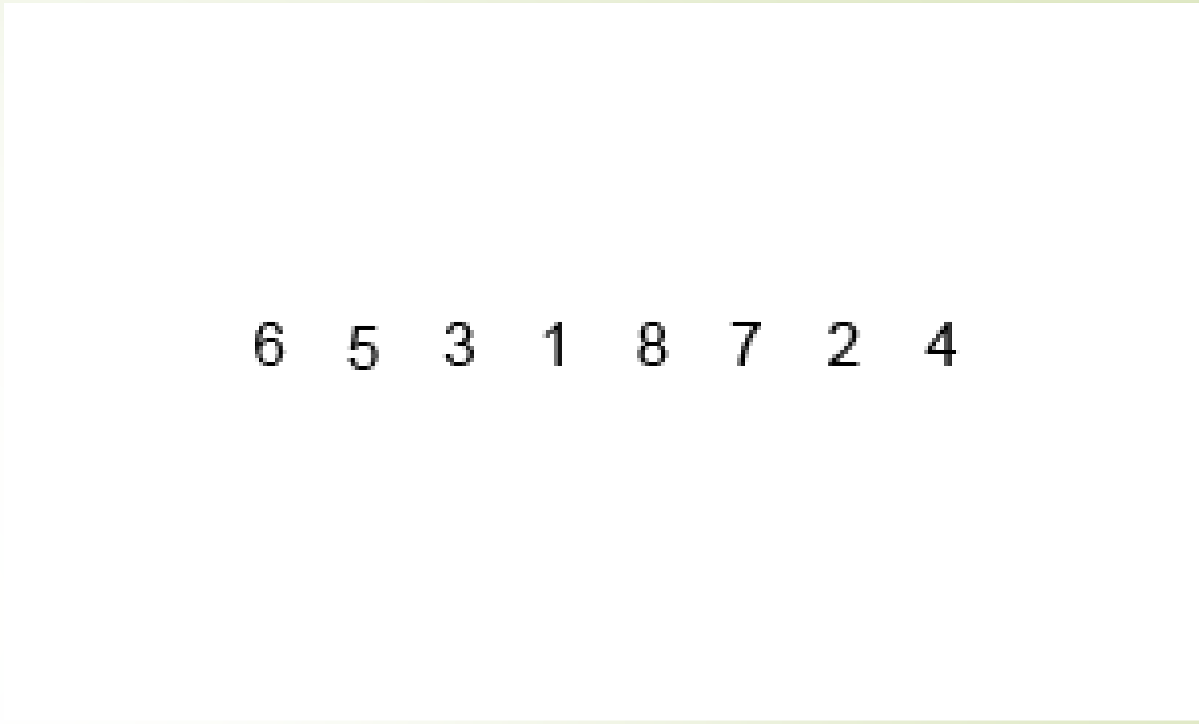


Example Merge Sort



Pseudo-code

```
• MS(A,start,end)
    • If(start<end)
        • Mid=(end-start)/2+start;
        • MS(A,start,Mid)
        • MS(A,Mid+1,end)
    • Combine(A,start,mid,mid+1,end)
• Combine(A,start1,end1,start2,end2)
    • For(i=start1,j=start2;i<=end1,j<=end2;)
        • A[i]<A[j]? R[k++]=A[i++]: R[k++]=A[j++]
    • For(i=start1;i<=end1;)
        • R[k++]=A[i++]
    • For(j=start2; j<=end2;)
        • R[k++]=A[j++]
```



About Merge Sort

- Divide and Conquer (Recursively)
- efficient for large data sets
  - O(n log n) sorting algorithm
- Stable Algorithm
- Not in-place o(n)
- Works good for data stored in secondary storage
  - Good candidate for huge amount of data stored in a disk

Complexity Analysis

- General Divide and Conquer
- $T(n) = aT\left(\frac{n}{b}\right) + g(n)$
- *Merge Sort*
- $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

Iterative Substitution

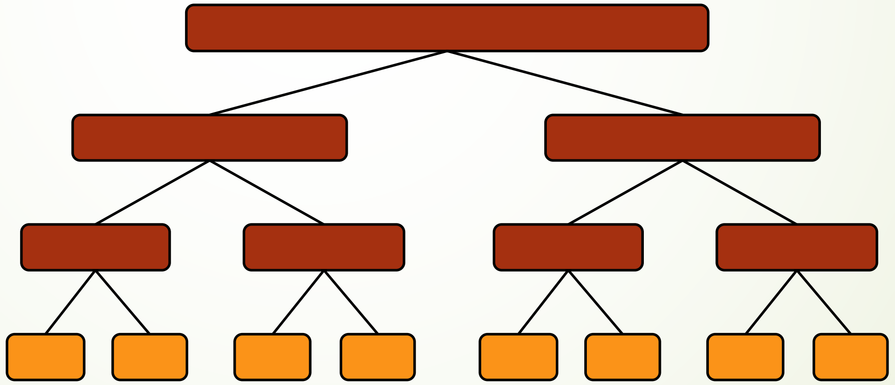
- $T(n) = 2T\left(\frac{n}{2}\right) + n$ (1)
- $T(n/2) = 2T\left(\frac{n}{4}\right) + n/2$ (2)
- $T(n/4) = 2T\left(\frac{n}{8}\right) + n/4$ (3)
- $T(n/8) = 2T\left(\frac{n}{16}\right) + n/8$ (4)
- By substitution (2) in (1)
- $T(n) = 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n$
- $T(n) = 4T\left(\frac{n}{4}\right) + n \ + n$
- $T(n) = 4T\left(\frac{n}{4}\right) + 2n$
- By substitution (3)
- $T(n) = 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n$
- $T(n) = 8T\left(\frac{n}{8}\right) + 3n$
- By substitution (4)
- $T(n) = 8\left(2T\left(\frac{n}{16}\right) + \frac{n}{8}\right) + 3n$
- $T(n) = 16T\left(\frac{n}{16}\right) + 4n$
- $T(n) = 2^4T\left(\frac{n}{2^4}\right) + 4n$
- $\dots$
- $T(n) = 2^kT\left(\frac{n}{2^k}\right) + kn$
- $T(n) = 2^kT\left(\frac{n}{2^k}\right) + kn$
- $If\ 2^k = n\ then\ k = logn$
- $T(n) = nT\left(\frac{n}{n}\right) + logn * n$
- $T(n) = nT(1) + nlogn$
- $T(1) = 1$
- $T(n) = n + nlogn$

The Recursion Tree

- Draw the recursion tree for the recurrence relation and look for a pattern:

$$T(n) = \begin{cases} 0 & \text{if } n < 2 \\ 2T(n/2) + n & \text{if } n \geq 2 \end{cases}$$

depth	T's	size	Time
0	1	$\boldsymbol{n}$	$\boldsymbol{n}$
1	2	$\boldsymbol{n}/2$	$\boldsymbol{n}$
$\boldsymbol{i}$	2^i	$\boldsymbol{n}/2^i$	$\boldsymbol{n}$
…	…	…	…



Total time = $\boldsymbol{n} + \boldsymbol{n} \log \boldsymbol{n}$

(last level plus all previous levels)

Master Theorem: Analysis of Divide & Conquer

<u>Define:</u>

a	= number of sub-instances that must be solved
n	= original instance size (variable)
n/b	= size of sub-instances
d	= polynomial order of $g(n)$, where $g(n)$ = cost of dividing and recombining

$$t(n) = a \cdot t(n/b) + g(n) \qquad \text{for } g(n) \in O(n^d)$$

Then:

$$t(n) = \begin{cases} \theta(n^d) & \text{if } a < b^d \Leftrightarrow d > \log_b a \\ \theta(n^d \log n) & \text{if } a = b^d \Leftrightarrow d = \log_b a \\ \theta(n^{\log_b a}) & \text{if } a > b^d \Leftrightarrow d < \log_b a \end{cases}$$

Summary of Sorting Algorithms

Algorithm	Time	Notes
bubble-sort	$O(n^2)$	-in-place -slow (good for small inputs)
selection-sort	$O(n^2)$	-in-place -slow (good for small inputs)
insertion-sort	$O(n^2)$	-in-place -slow (good for small inputs)
merge-sort	$O(n \log n)$	-sequential data access -fast (good for huge inputs)
quick-sort	$O(n \log n)$	-in-place -fast (good for large inputs)
counting sort	$O(n + k)$: k is max items value	-not a comparison sort -Work with integers
radix sort	$O(n + k)$	-not a comparison sort

Master Method, Example 1

- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

- Example: $T(n) = 4T(n/2) + n$

Solution: $\log_b a=2$, so case 1 says T(n) is $O(n^2)$.

Master Method, Example 2

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
     provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example: $T(n) = 2T(n/2) + n \log n$

Solution: $\log_b a=1$, so case 2 says T(n) is $O(n \log^2 n)$.

Master Method, Example 3

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$

- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
     provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

- Example: $T(n) = T(n/3) + n \log n$

Solution: $\log_b a=0$, so case 3 says T(n) is O(n log n).

Master Method, Example 4

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a-\varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example: $T(n) = 8T(n/2) + n^2$

Solution: $\log_b a=3$, so case 1 says T(n) is $O(n^3)$.

Master Method, Example 5

- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$$

- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a-\varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

- Example: $T(n) = 9T(n/3) + n^3$

  Solution: $\log_b a=2$, so case 3 says T(n) is $O(n^3)$.

Master Method, Example 6

- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example: $T(n) = T(n/2) + 1$ (binary search)

Solution: $\log_b a=0$, so case 2 says T(n) is O(log n).

Master Method, Example 7

- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$$

- The Master Theorem:

  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
     provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

- Example: $T(n) = 2T(n/2) + \log n$ (heap construction)

  Solution: $\log_b a=1$, so case 1 says T(n) is O(n).