Data Structures and Algorithms

(ESO207)

Lecture 15:

- Algorithm paradigm of Divide and Conquer :

  Counting the number of Inversions
- Another sorting algorithm based on Divide and Conquer : Quick Sort

Divide and Conquer paradigm
An Overview

1. **Divide** the problem instance into two or more instances of the same problem
2. Solve each smaller instances **recursively** (base case suitably defined).
3. $\boxed{\textbf{Combine}}$ the solutions of the smaller instances
   to get the solution of the original instance.

This is usually the main **nontrivial** step in the design of an algorithm using divide and conquer strategy

2 IMPORTANT LESSONS

THAT WE WILL LEARN TODAY…

1. Role of Data structures in algorithms
2. Learn from the past …

Role of Data Structures in designing efficient algorithms

Definition: A collection of data elements *arranged* and *connected* in a way which can facilitate efficient executions of a (possibly long) sequence of operations.

Parameters:

- Query/Update time
- Space
- Preprocessing time

Role of Data Structures in designing efficient algorithms

Definition: A collection of data elements *arranged* and *connected* in a way which can facilitate efficient executions of a (possibly long) sequence of operations.

Consider an Algorithm $\boldsymbol{A}$.

Suppose $\boldsymbol{A}$ performs many operations of **same type** on some data.

Improving time complexity of these operations → Improving the time complexity of $\boldsymbol{A}$.

So, it is worth designing a suitable **data structure**.

Counting Inversions in an array

Problem description

Definition (Inversion): Given an array **A** of size n,

a pair (i,j), $0 \le i < j < n$ is called an inversion if **A**$[i] >$ **A**$[j]$.

Example:

	0	1	2	3	4	5	6	7
A	3	15	8	19	9	67	11	27

Inversions are :

(1,2), (1,4), (1,6),

(3,4),(3,6),

(5,6), (5,7)

AIM: An efficient algorithm to count the number of inversions in an array **A**.

Counting Inversions in an array

Problem familiarization

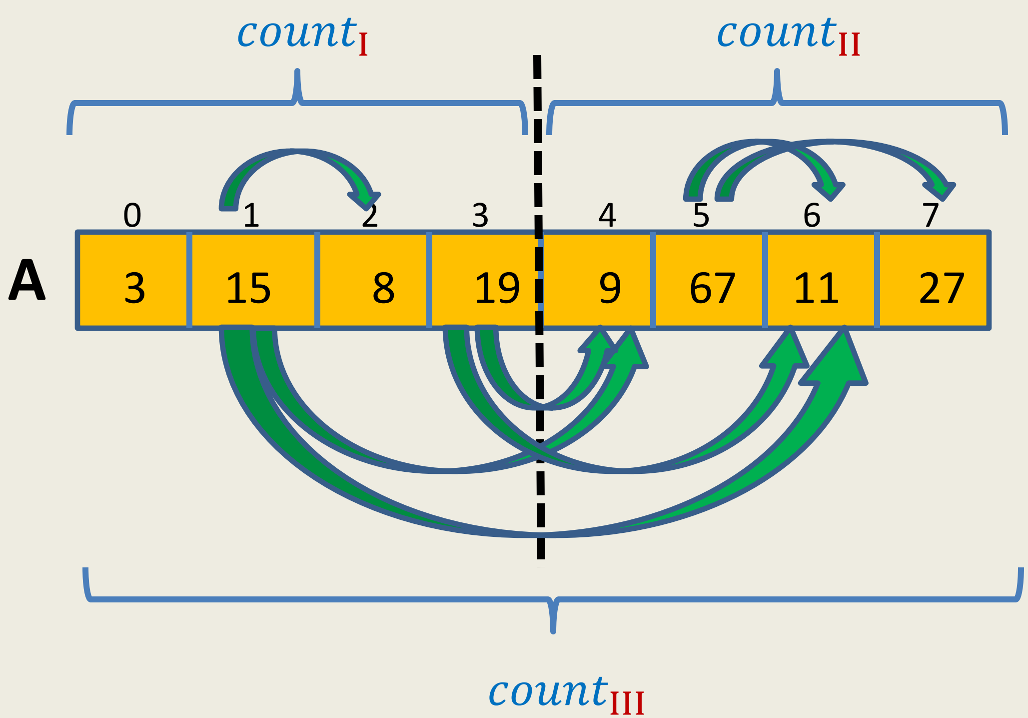
```
Trivial-algo(A[0..n − 1])
{ count ← 0;
  For(j=1  to n − 1) do
  {      For( i=0 to j − 1 )
         {      If (  A[i]>A[j]  )  count ← count + 1;
         }
  }   return count;
}
```

Time complexity: O(n^2)

Let us try to design a

Divide and Conquer based algorithm

How do we approach using divide & conquer



Counting Inversions
Divide and Conquer based algorithm

CountInversion(**A**,$\boldsymbol{i}$, $\boldsymbol{k}$) // Counting no. of inversions in **A**[$\boldsymbol{i}..\boldsymbol{k}$]

If ($\boldsymbol{i} = \boldsymbol{k}$) return 0;

Else{ $\mathbf{mid} \leftarrow (\boldsymbol{i} + \boldsymbol{k})/2$;

$count_{\mathrm{I}} \leftarrow$ **CountInversion**(**A**,$\boldsymbol{i}$, $\mathbf{mid}$);

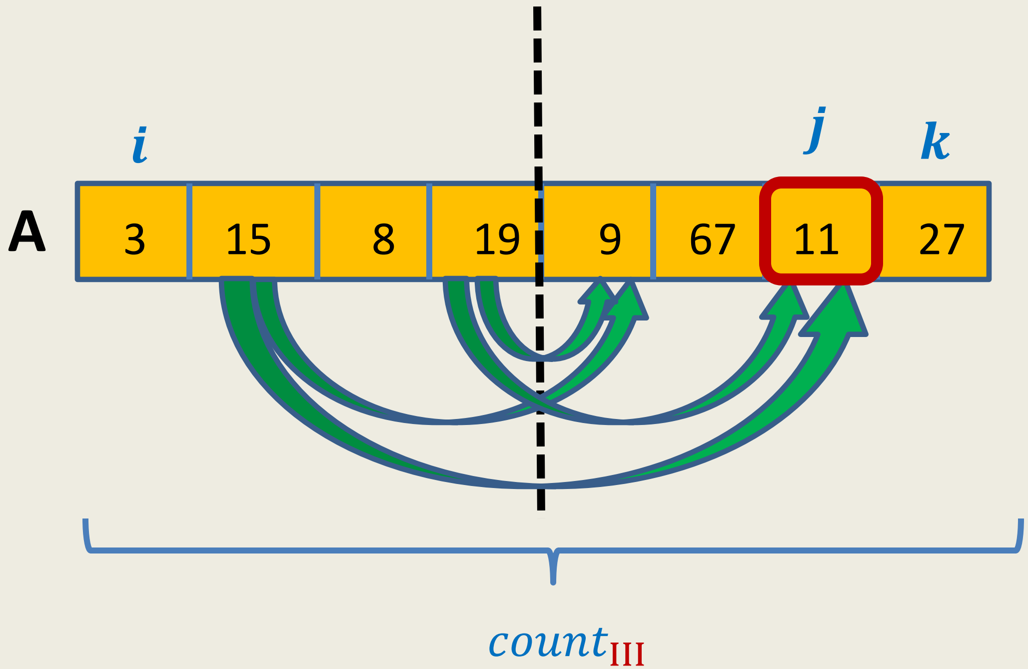
$count_{\mathrm{II}} \leftarrow$ **CountInversion**(**A**,$\mathbf{mid} + \mathbf{1}, \boldsymbol{k}$);

…. *Code for* $count_{\mathrm{III}}$ ….

return $count_{\mathrm{I}}$ + $count_{\mathrm{II}}$ + $count_{\mathrm{III}}$;

}

How to efficiently compute $count_{\text{III}}$ (Inversions of type III) ?



$\mathbf{O}(n^2)$ time algo

Aim: For each $\mathbf{mid} < j \le k$, count the elements in **A**[i..$\mathbf{mid}$] that are **greater** than **A**[j].

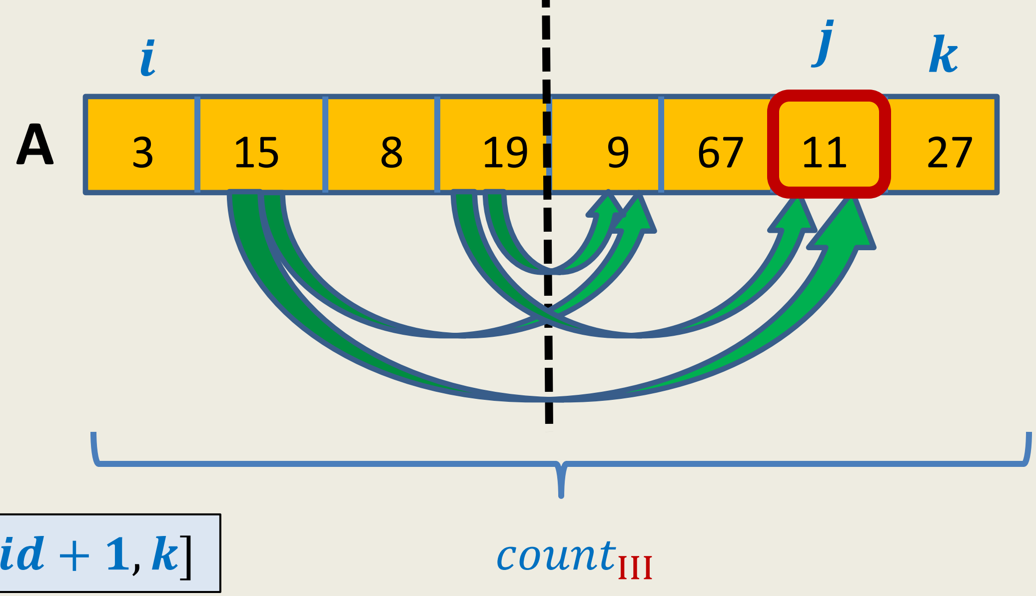
Trivial way: **O**(**size of the subarray A**[i..$\mathbf{mid}$]) time for a given j.

➔ **O**(n) time for a given j in the first call of the algorithm.

➔ **O**(n^2) time for computing $count_{\text{III}}$ since there are $n/2$ possible values of j.

How to efficiently compute $count_{\text{III}}$ (Inversions of type III) ?

Let us state clearly what we want to achieve …



For each $\boldsymbol{j} \in [\boldsymbol{mid} + \boldsymbol{1}, \boldsymbol{k}]$

count the elements in **A**[$\boldsymbol{i}$..$\mathbf{mid}$] that are **greater** than **A**[$\boldsymbol{j}$].

Time to apply Lesson 1

What should be the **data structure** ?

Sorted subarray **A**[$\boldsymbol{i}$..$\mathbf{mid}$].

Counting Inversions
First algorithm based on divide & conquer

CountInversion(**A**,i, k)

If (i=k) return 0;

Else{ $\mathbf{mid} \leftarrow (i + k)/2$;

$count_{\mathrm{I}} \leftarrow$ **CountInversion**(**A**,i, $\mathbf{mid}$);

$count_{\mathrm{II}} \leftarrow$ **CountInversion**(**A**,$\mathbf{mid}+\mathbf{1}$, k);

2 **T**($n/2$)

Sort(**A**,i, $\mathbf{mid}$);

For each $\mathbf{mid} < j \le k$

do **binary search** for **A**[j] in **A**[i..$\mathbf{mid}$] to compute the *number* of elements greater than **A**[j].

Add this *number* to $count_{\mathrm{III}}$;

c n **log** n

return $count_{\mathrm{I}}$ + $count_{\mathrm{II}}$ + $count_{\mathrm{III}}$;

}

Counting Inversions
First algorithm based on divide & conquer

Time complexity analysis:

If $n = 1$,

$\quad$ $T(n) = c$ for some constant c

If $n > 1$,

$\quad$ $T(n) = c\, n \log n + 2\, T(n/2)$

$\quad\quad = c\, n \log n + c\, n ((\log n)-1) + 2^2\, T(n/2^2)$

$\quad\quad = c\, n \log n + c\, n ((\log n)-1) + c\, n ((\log n)-2) + 2^3\, T(n/2^3)$

$\quad\quad = O(n \log^2 n)$

Can we improve it further ?

Counting Inversions

First algorithm based on divide & conquer

CountInversion(A,i, k)

If (i=k) return 0;

Else{ $\mathbf{mid} \leftarrow (i+k)/2$;

$count_{\mathrm{I}} \leftarrow$ CountInversion(A,i, $\mathbf{mid}$);

$count_{\mathrm{II}} \leftarrow$ CountInversion(A,$\mathbf{mid}+1, k$);

2 T($n/2$)

Sort(A,i, $\mathbf{mid}$);

For each $\mathbf{mid} < j \le k$

do binary search for A[j] in A[i..$\mathbf{mid}$] to compute the *number* of elements greater than A[j].

Add this *number* to $count_{\mathrm{III}}$;

c n log n

return $count_{\mathrm{I}}$ + $count_{\mathrm{II}}$ + $count_{\mathrm{III}}$;

}

Sequence of observations
To achieve better running time

- The extra **log** n factor arises because for the "**combine**" step, we are spending **O**(n **log** n) time instead of **O**(n).

- The reason for **O**(n **log** n) time for the "**combine**" step:
  - **Sorting A**[0.. n/2] takes **O**(n **log** n) time.
  - Doing **Binary Search** for n/**2** elements from **A**[n/**2… n -1**]

- Each of the above tasks have optimal running time.

- So the only way to improve the running time of "**combine**" step is some new idea

Revisiting MergeSort algorithm

MSort(**A**, i, k)// Sorting **A**[$i..k$]

{ **If** ($i < k$)

{ $\mathbf{mid}$ ← ($i + k$)/2;

MSort(**A**, i, $\mathbf{mid}$);

MSort(**A**, $mid + \mathbf{1}$, k);

Create a temporary array **C**[$\mathbf{0}..k - i$]

Merge(**A**, i, mid, k, **C**);

Copy **C**[$\mathbf{0}..k - i$] to **A**[$i..k$]

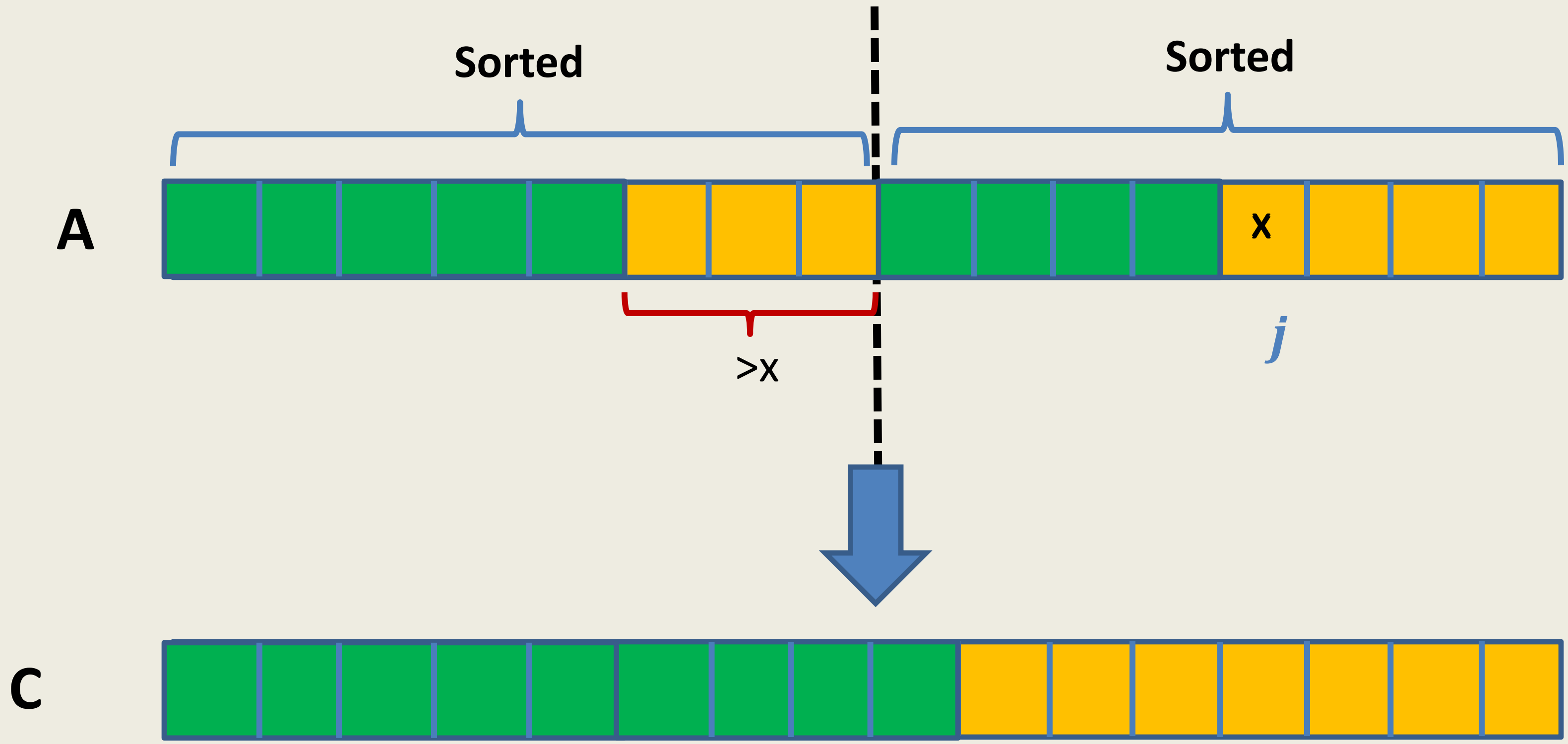
}

}

We shall carefully look at the **Merge**() procedure to find an efficient way to count the number of elements from **A**[$i..\mathbf{mid}$] which are smaller than **A**[j] for any given $\mathbf{mid} < j \le k$

Relook

Merging A[$i..\mathbf{mid}$] and A[$\mathbf{mid}+\mathbf{1}..k$]



Pesudo-code for Merging two sorted arrays

```
Merge(A,i, mid, k,C)
  p← i;  j← mid + 1; r← 0;
  While(p ≤ mid  and j  ≤  k)
  {      If(A[p]< A[j]) {    C[r] ← A[p]; r++;  p++  }
         Else            {    C[r] ← A[j]; r++;  j++  }
  }
  While(p ≤ mid)    {  C[k] ← A[i]; k++;  i++   }
  While(j ≤ k)      {  C[k] ← A[j]; k++;  j++   }
  return C ;
```

We shall make **just a slight change** in the above pseudo-code to achieve our main objective of computing $count_{\mathrm{III}}$. If you understood the discussion of the previous slide, can you guess it now ?

Pesudo-code for
Merging and counting inversions

```
Merge_and_CountInversion(A, i, mid, k, C)
  p ← i;  j ← mid + 1; r ← 0;
  count_III ← 0;
  While(p ≤ mid  and j ≤ k)
  {      If(A[p]< A[j]) {    C[r] ← A[p]; r++;  p++  }
         Else             {    C[r] ← A[j]; r++;  j++
                                   count_III ← count_III +  (mid − p + 1);
                          }
  }
  While(p ≤ mid)   {  C[k] ← A[i]; k++;  i++   }
  While(j ≤ k)     {  C[k] ← A[j]; k++;  j++   }
  return count_III;
```

Nothing extra is needed here.

Counting Inversions
Final algorithm based on divide & conquer

Sort_and_CountInversion(**A**, $\boldsymbol{i}$, $\boldsymbol{k}$)

{ **If** ($\boldsymbol{i} = \boldsymbol{k}$) return 0;

else

{ $\mathbf{mid} \leftarrow (\boldsymbol{i} + \boldsymbol{k})/\mathbf{2}$;

$count_{\mathrm{I}}$ ← **Sort_and_CountInversion** (**A**,$\boldsymbol{i}$, $\mathbf{mid}$);

$count_{\mathrm{II}}$ ← **Sort_and_CountInversion** (**A**,$\mathbf{mid} + \mathbf{1}$, $\boldsymbol{k}$);

Create a temporary array **C**[$\mathbf{0}..\boldsymbol{k} - \boldsymbol{i}$]

$count_{\mathrm{III}}$ ← **Merge_and_CountInversion**(**A**,$\boldsymbol{i}$, $\mathbf{mid}$, $\boldsymbol{k}$,**C**);

Copy **C**[$\mathbf{0}..\boldsymbol{k} - \boldsymbol{i}$] to **A**[$\boldsymbol{i}..\boldsymbol{k}$];

return $count_{\mathrm{I}}$ + $count_{\mathrm{II}}$ + $count_{\mathrm{III}}$;

}

}

2 T($\boldsymbol{n}/\mathbf{2}$)

O($\boldsymbol{n}$)

Counting Inversions
Final algorithm based on divide & conquer

Time complexity analysis:

If $n = 1$,

$\mathbf{T}(n) = c$ for some constant c

If $n > 1$,

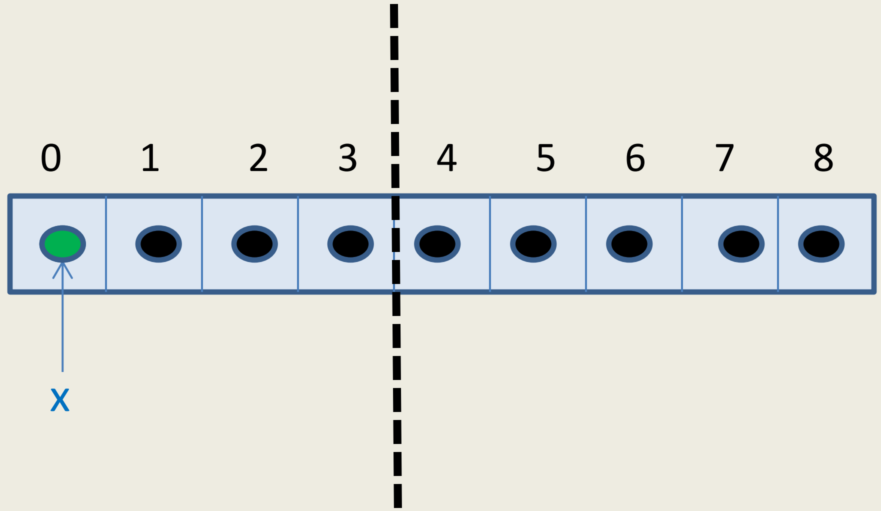
$$\mathbf{T}(n) = c\,n + \mathbf{2}\,\mathbf{T}(n/\mathbf{2})$$
$$= \mathbf{O}(n \log n)$$

Theorem: There is **a divide and conquer** based algorithm for computing the number of inversions in an array of size n. The running time of the algorithm is $\mathbf{O}(n \log n)$.

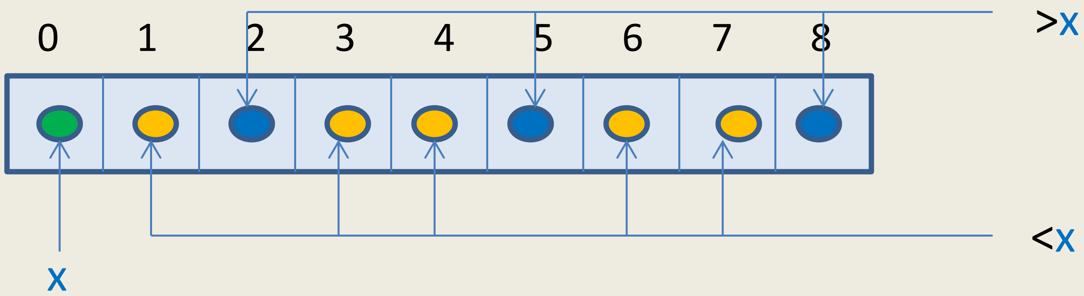
Another sorting algorithm based on divide and conquer

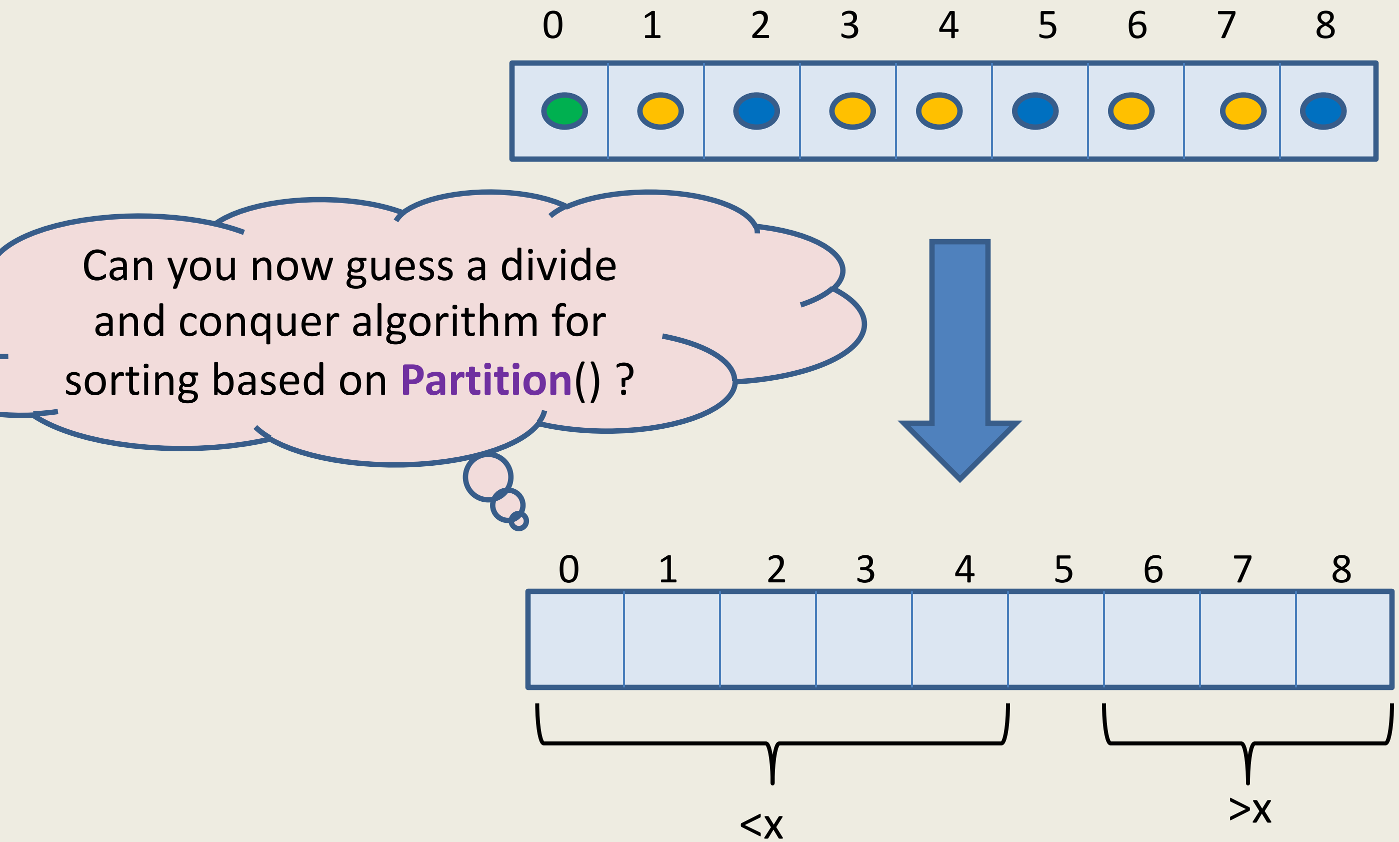
QuickSort

Is there any alternate way to divide ?



In MergeSort, we divide the input instance in an obvious manner.


0
1
2
3
4
5
6
7
8
>x
<x
x



This procedure is called **Partition**.

It **rearranges** the elements so that all elements less than x appear to the left of x and all elements greater than x appear to the right of x.

Pseudocode for QuickSort(S)

```
QuickSort(S)
{       If (|S|>1)
                Pick and remove an element x from S;
                (S_{<x}, S_{>x}) ← Partition(S,x);
                return( Concatenate(QuickSort(S_{<x}), x, QuickSort(S_{>x}))
}
```

Pseudocode for QuickSort(S)

When the input S is stored in an array

```
QuickSort(A, l, r)
{       If (l < r)
            i ← Partition(A, l, r);  // i is index where element A[l] is finally placed
            QuickSort(A, l, i − 1);
            QuickSort(A, i + 1, r)
}
```

View this algorithm from various perspectives. For almost all practical purposes, this is the most efficient algorithm for sorting. It outperforms **MergeSort** by a significant factor.

QuickSort

Homework:

- The running time of Quick Sort depends upon
  the element we choose for partition in each recursive call.
- What can be the worst case running time of Quick Sort ?
- What can be the best case running time of Quick Sort ?

- Give an implementation of **Partition** that takes **O**($r - l$) time and using **O**(1) extra space only. (Given as homework earlier)

*Sometime later in the course, we shall revisit **QuickSort** and analyze it theoretically (average time complexity) and experimentally.*

The outcome will be surprising and counterintuitive. ☺