The Role of Algorithms in Computing:

1. What are algorithms?
2. Why is the study of algorithms worthwhile?
3. What is the role of algorithms relative to other computer technologies?

Ans .

Informally, an algorithm is any well-defined computational step that takes some value or set of values as input and produces some value or set of values as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

E.g.1 Sort a sequence of numbers into non-decreasing order

Input:

A sequence of n numbers $(a_1 + a_2 + \cdots + a_n)$

Output:

A permutation (re-ordering) $(a_1 + a_2 + \cdots + a_n)$ of input sequence such as that $(a_1 \leq a_2 \leq \cdots \leq a_n)$

E.g-2

Input: $(31,41,59,26,41,58)$

output:$(26,31,41,41,58,59)$

- **<u>Data Structure:</u>** A data structure is a way to store and organize data to facilitate access and modification
  $A = (31,41,59,26,41,58)$
  $=> 31,41,59,26,41,58$
  $=> 31,41,26,59,41,58$
  $=> 26,31,41,59,41,58$
  $=> 26,31,41,41,59,58$
  $=> 26,31,41,41,58,59$

1. $for\ j = 2\ to\ A.length\ do$
2. $key = A[j]$
3. $i = j - 1$
4. $while\ i > 0\ and\ A[i] > key$

5. $A[i+1] = A[i]$
6. $i = i \neq 1$
7. $A[i+1] = key$

Find

i. Linear Searching
ii. Max
iii. Min

Input : $A = (10,15,5,6,7,8,25,2,1)$

Pseudocode for Linear Searching

1. $V\ key\ to\ search$
2. $for\ i = 1\ to\ A.length$
3. $if\ A[i] = V$
4. $return\ i$
5. $end\ if$
6. $end\ for$
7. $return\ NIL$

Time complexity

- Number of loops 1 which runs from 1 to n times therefore

$$n\ times$$

- Number of memory access 1 on line number 3 i.e. $A[i]$

$$1\ access$$

Let :

$$T(n) = \sum_{i=1}^{n} i\ 1 = 1 \sum_{i=1}^{n} i = 1 * n = n$$

Hence

$$\boldsymbol{\theta(n)}$$

Pseudocode for finding Max

1. $\max \leftarrow A[i]$
2. $for\ i \leftarrow 2\ to\ n$
3. $if\ \max < A[i]$
4. do
5. $\max \leftarrow A[i]$
6. $end\ if$
7. $end\ for$
8. $output\ max$

First line memory access $1\ time$

Summation of for i.e.

$$\sum_{i=2}^{n} i\ 2$$

Let

$$T(n) = 1 \left(\sum_{i=2}^{n} i\ 2 \right) = 2 \sum_{i=2}^{n} i = 2n$$

Hence in terms of n

$$\theta(n)$$

Pseudocode for finding Min

1. $\min \leftarrow A[i]$
2. $for\ i \leftarrow 2\ to\ n$
3. $if\ \min > A[i]$
4. do
5. $\min \leftarrow A[i]$
6. $end\ if$
7. $end\ for$
8. $output\ max$

First line memory access 1 $time$

Summation of for i.e.

$$\sum_{i=2}^{n} i\ 2$$

Let

$$T(n) = 1\left(\sum_{i=2}^{n} i\ 2\right) = 2\sum_{i=2}^{n} i\ = 2n$$

Hence in terms of n

$$\theta(n)$$

- **2-diamential Maxima**

See lecture slides

Time analysis of 2-diamential Maxima

Pseudocode

Assume it's a function

$MAXIMA(int\ n, Point\ P[1 \ldots n])$

1. $for\ i \leftarrow 1\ to\ n$ $n\ times$
2. $do\ maximal \leftarrow true$

3. $for\ j \leftarrow 1\ to\ n$ $n\ times$
4. do
5. $if\ (i \neq j)\ and\ (P[i].x\ \leq P[j].x)\ and\ (P[i].y\ \leq P[j].y)$ $4\ access$
6. $then\ maximal \leftarrow false$
7. $break$
8. $end\ if$
9. $end\ for$
10. $if\ maximal$
11. $then\ output\ P[i].x, P[i].y$ $2\ access$
12. $end\ if$
13. $end\ for$

In terms of $T(n)$

1. Calculate inner loop i.e. on line no. 3, Lets say $T(I)$

$$I = \sum_{j=1}^{n} 4 = 4\sum_{j=1}^{n} j\ = 4n$$

For outer for loop lets say $T(M)$

$$M = \sum_{i=1}^{n}(2 + T(I)) = \sum_{i=1}^{n}(2 + 4n) = 2\sum_{i=1}^{n} + 4n\sum_{i=1}^{n} = 2\sum_{i=1}^{n} + 4n^2$$

$$= 2n + 4n^2 => 4n^2 + 2n$$

In terms of n Hence the largest n would be the worst-time case i.e.

$$\theta(n^2)$$

- **<u>Complexity of Algorithm</u>**

We calculate the complexity on running time and memory access. There are three cases of the running time

i. Worst-case
ii. Average-case
iii. Best-case

For the definition check lecture slides

There are here steps to calculate complexity

i. Nos. of steps

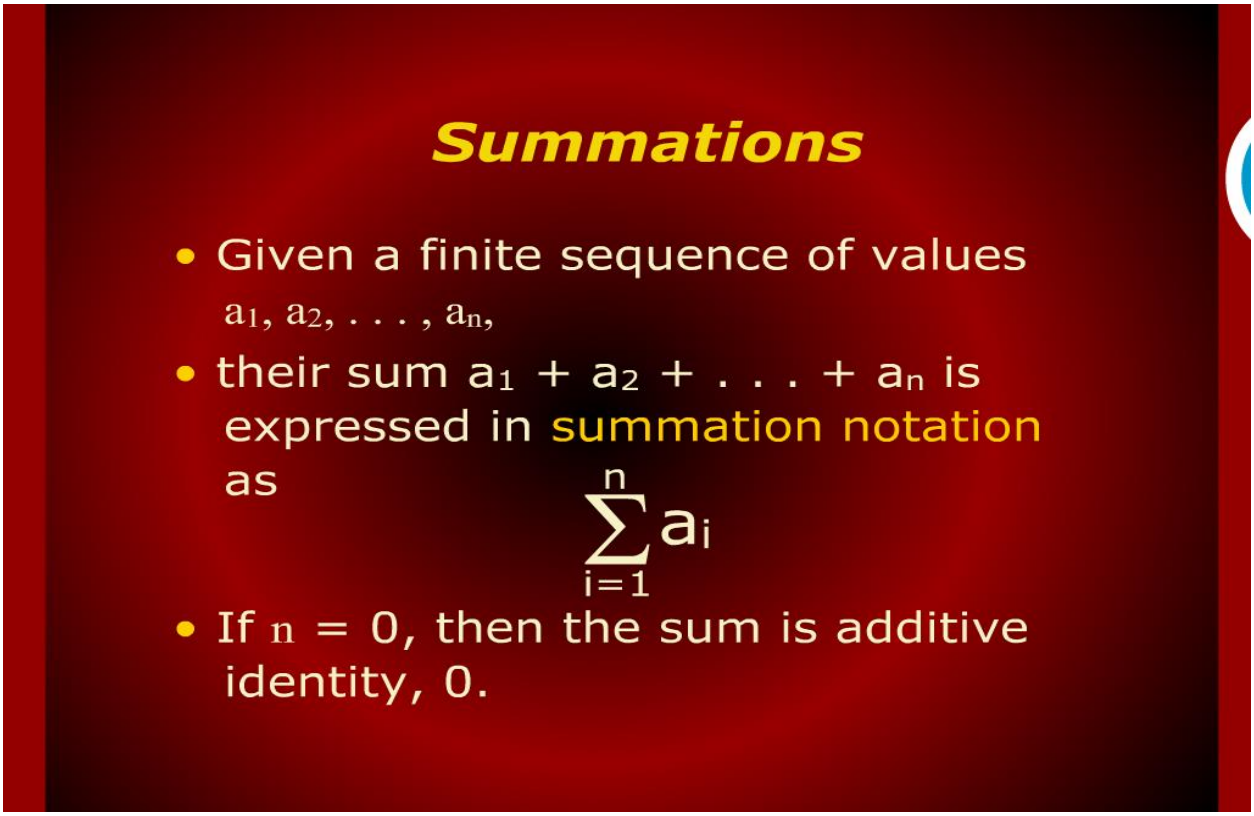
Or

ii. Nos. of comparisons

Or

iii. Nos. of Access of Array and running time of loops

Summations rules



Summations

Some facts about summation:

- If c is a constant

$$\sum_{i=1}^{n} c a_i = c \sum_{i=1}^{n} a_i$$

and

$$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$$

Summations

Some important summations that should be committed to memory.

Arithmetic series:

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n$$

$$= \frac{n(n+1)}{2} = \Theta(n^2)$$

Summations

Quadratic series:

$$\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \ldots + n^2$$

$$= \frac{2n^3+3n^2+ n}{6} = \Theta(n^3)$$

Summations

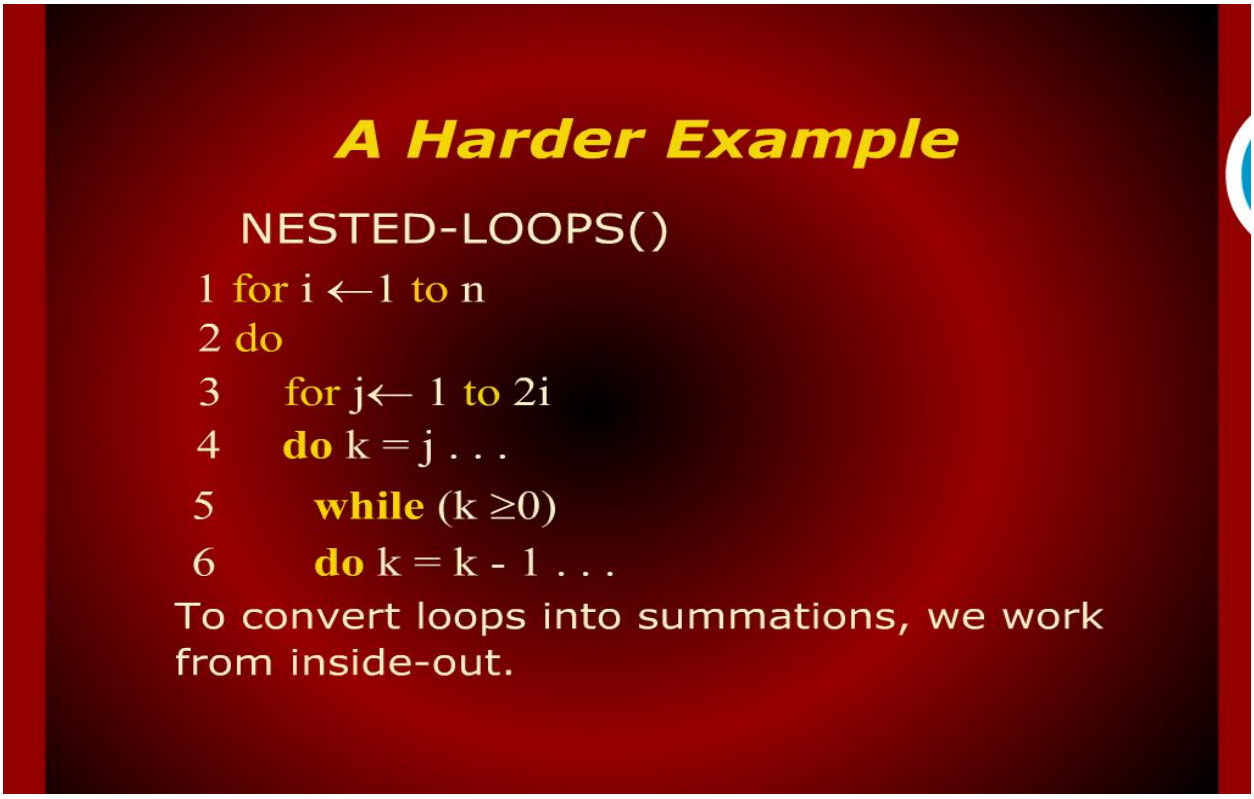
Geometric series:

$$\sum_{i=1}^{n} x^i = 1 + x + x^2 + \ldots + x^n$$

$$= \frac{x^{(n+1)} - 1}{x - 1} = \Theta(n^2)$$

If $0 < x < 1$ then this is $\Theta(1)$, and if $x > 1$, then this is $\Theta(x^n)$.

A HARDER EXAMPLE



Lets first solve inner loop i.e. from line no. 4 to 6

As we know the while loop based on the value of k where $k = j$

Lets say $T(J)$ for while loop

$$J = \sum_{k=0}^{j} 1 = j + 1$$

Hence $T(J) = j + 1$

Now calculate the inner for loop i.e. from 3 to 6

Lets say

$$T(I) = \sum_{j=1}^{2i} T(J) = \sum_{j=1}^{2i} (j + 1)$$

$$= \sum_{j=1}^{2i} j + \sum_{j=1}^{2i} 1$$

$$= \sum_{j=1}^{2i} j + 2i - - - - - - - - - eq1$$

Now calculate summation of j by applying summation rule arithmetic series we get:

$$\sum_{j=1}^{2i} j = \frac{n(n+1)}{2} \; in\ our\ case\ n = 2i\ so\ equation\ would\ be$$

$$= \frac{2i(2i+1)}{2} \; please\ this\ value\ in\ the\ eq1\ we\ get$$

$$T(I) = \frac{2i(2i+1)}{2} + 2i$$

Solve the above equation we get

$$T(I) = \frac{\cancel{2}i(2i+1)}{\cancel{2}} + 2i = 2i^2 + i + 2i = 2i^2 + 3i$$

Hence running time of inner loop is

$$T(I) = 2i^2 + 3i$$

Now calculate the outer-most loop i.e. from line no. 1 to so on.

$$T(n) = \sum_{i=1}^{n} T(I) = \sum_{i=1}^{n} (2i^2 + 3i)$$

$$T(n) = 2\sum_{i=1}^{n} i^2 + 3\sum_{i=1}^{n} i$$

$$T(n) = 2\sum_{i=1}^{n} i^2 + 3\frac{n(n+1)}{2}$$

$$T(n) = 2\frac{2n^3 + 3n^2 + n}{6} + 3\frac{n(n+1)}{2}$$

$$T(n) = \frac{4n^3 + 6n^2 + 2n}{6} + 3\frac{n(n+1)}{2} = \frac{4n^3 + 6n^2 + 2n}{6} + 3\frac{n^2 + n}{2}$$

$$T(n) = \frac{4n^3 + 6n^2 + 2n}{6} + \frac{3n^2 + 3n}{2}$$

Remember the LCM in above case it would be 6

So

$$T(n) = \frac{4n^3 + 6n^2 + 2n}{6} + \frac{9n^2 + 9n}{6} = \frac{4n^3 + 6n^2 + 2n + 9n^2 + 9n}{6}$$

$$T(n) = \frac{4n^3 + 15n^2 + 11n}{6}$$

Hence the highest value of n is n^3

Thus

$\boldsymbol{\theta(n^3)}$

- **Plane Sweep Algorithm on 2-dimentional Maxima**

See lecture slides Lecture-5

- **Asymptotic Notation**

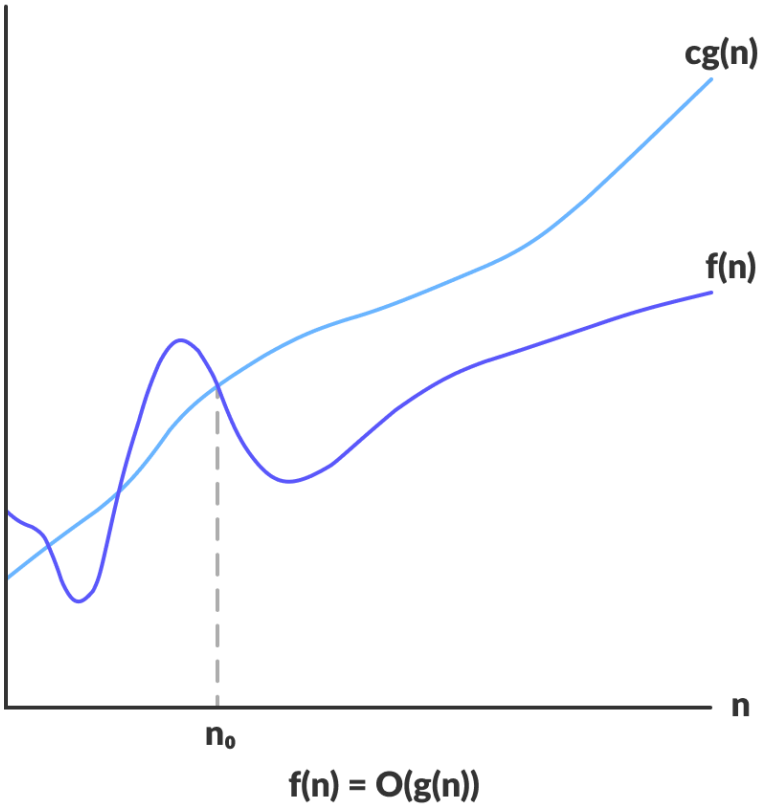
Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

We can say equivalent of two functions coming/raising together but not meet (very near).

There are mainly three asymptotic notations:

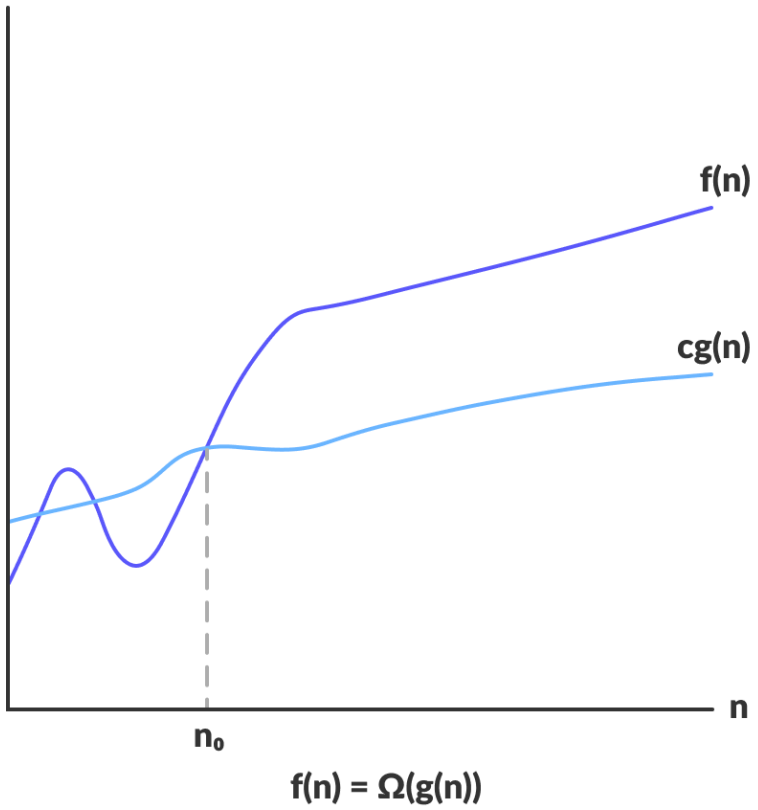
1. Big-O notation
2. Omega notation
3. Theta notation

1. **Big-O notation**



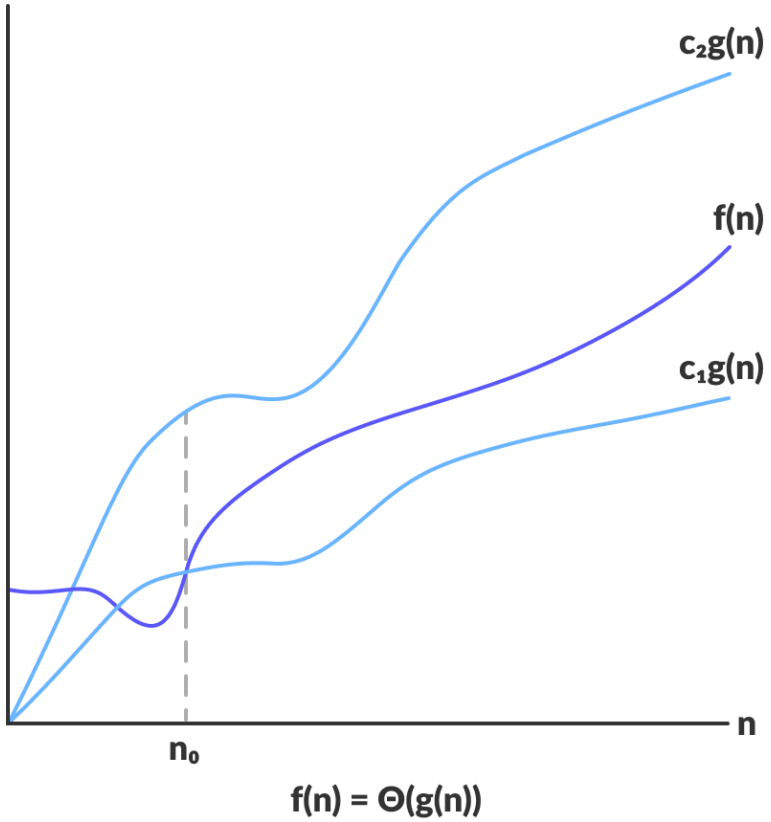
Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

2. **Omega Notation (Ω-notation)**



Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

3. **Theta Notation (Θ-notation)**



Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

Asymptotic Intuition

Here is a list of common asymptotic running times:

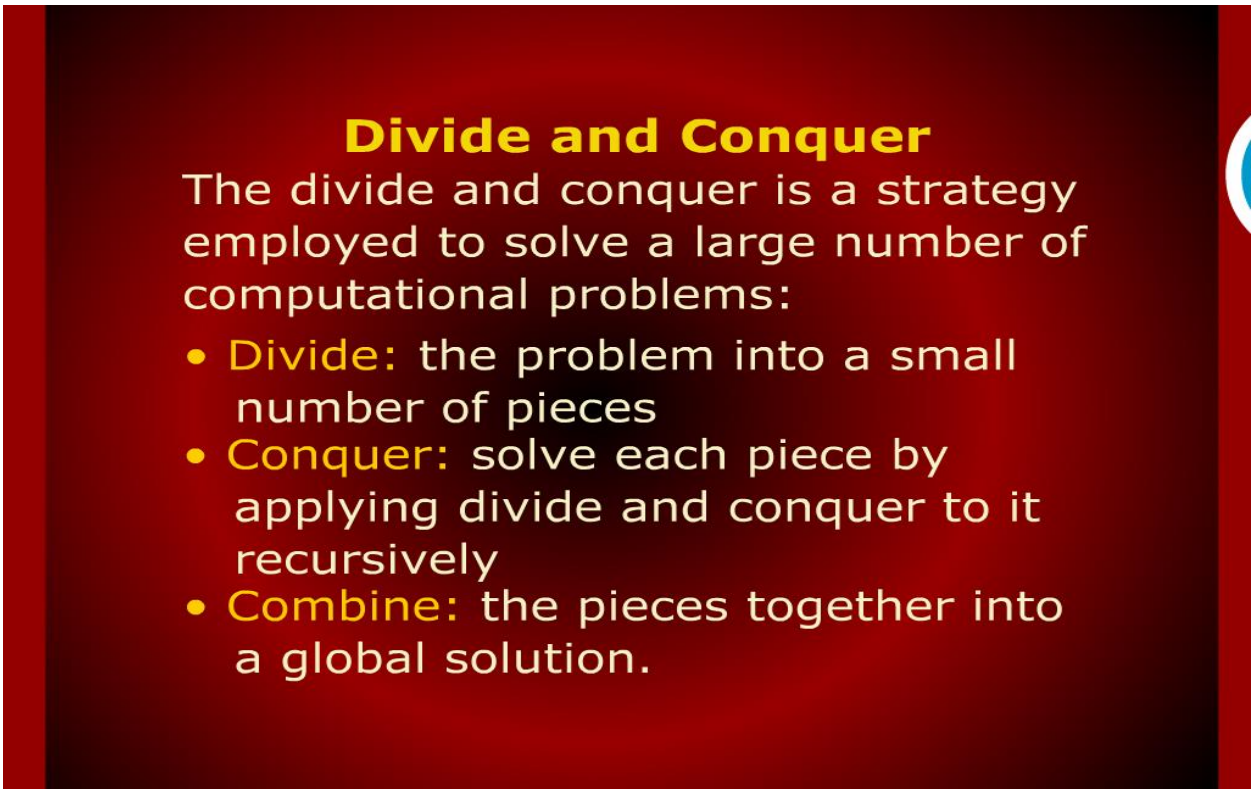
- $\Theta(1)$: Constant time; can't beat it!
- $\Theta(\log n)$: Inserting into a balanced binary tree; time to find an item in a sorted array of length n using binary search.
- $\Theta(n)$: About the fastest that an algorithm can run.

- $\Theta(n \log n)$: Best sorting algorithms.

Asymptotic Intuition

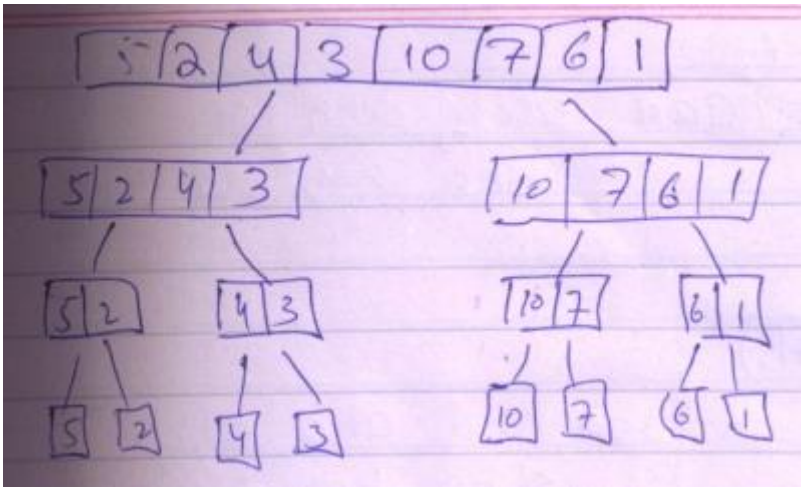
- $\Theta(n^2)$, $\Theta(n^3)$: Polynomial time. These running times are acceptable when the exponent of n is small or n is not too large,e.g., $n \leq 1000$.
- $\Theta(2^n)$, $\Theta(3^n)$: Exponential time. Acceptable only if n is small, e.g., $n \leq 50$.
- $\Theta(n!)$, $\Theta(n^n)$: Acceptable only for really small n, e.g. $n \leq 20$.

- **Divide and Conquer Algorithm.**



- **MERGE SORT:**

Which uses a recursive technique known as Divide-and-Conquer.



Merge Sort

- Divide and conquer strategy is applicable in a huge number of computational problems.
- The first example of divide and conquer algorithm we will discuss is a simple and efficient sorting procedure called *Merge Sort.*

Merge Sort

- We are given a sequence of n numbers A,which we will assume are stored in an array A[1..n].
- The objective is to output a permutation of this sequence sorted in increasing order.
- This is normally done by permuting the elements within the array A.

Merge Sort

Here is how the merge sort algorithm works:

Merge Sort

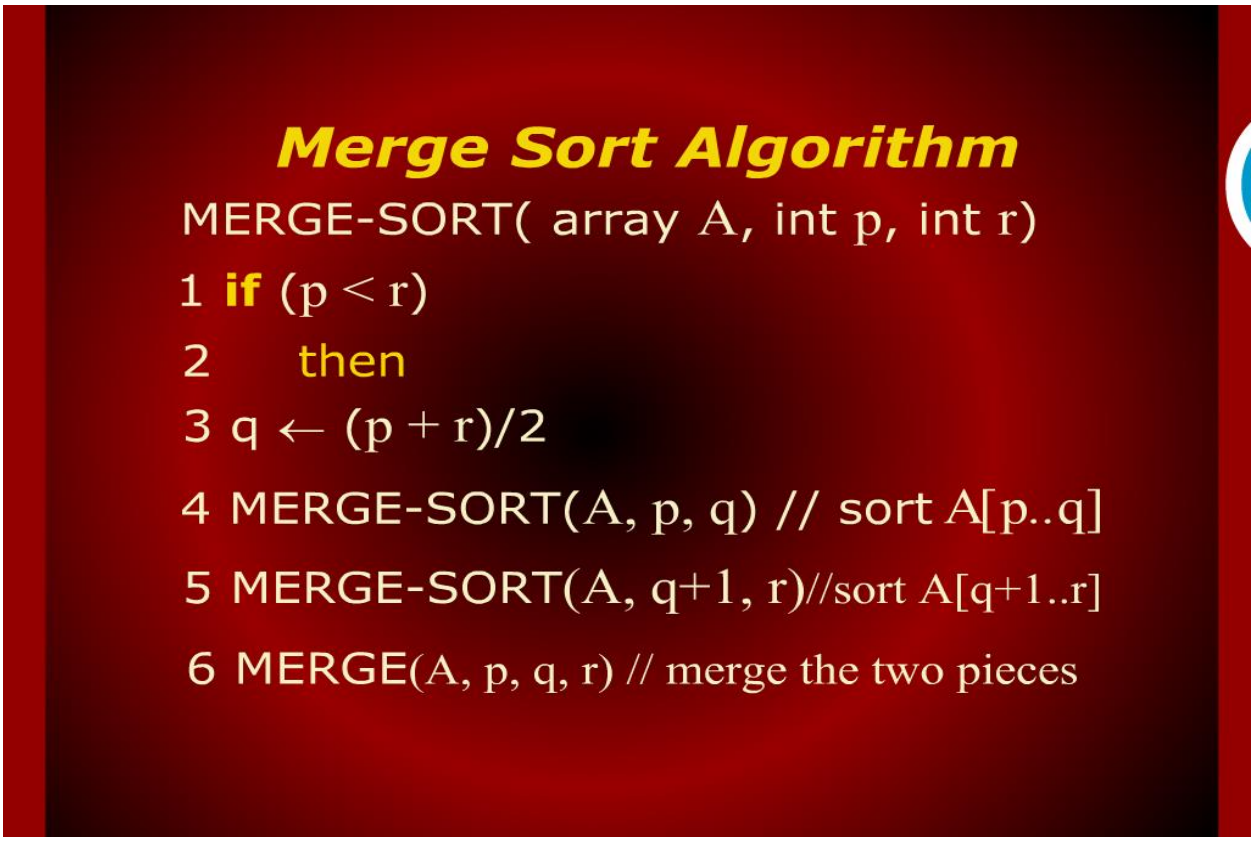
- Divide: split A down the middle into two subsequences, each of size roughly n/2
- Conquer: sort each subsequence by calling merge sort recursively on each.
- Combine: merge the two sorted subsequences into a single sorted list.

Merge Sort

- The dividing process ends when we have split the subsequences down to a single item.
- A sequence of length one is trivially sorted.
- The key operation is the combine stage which merges together two sorted lists into a single sorted list.

Merge Sort

- A sequence of length one is trivially sorted.
- The key operation is the combine stage which merges together two sorted lists into a single sorted list.
- Fortunately, the combining process is quite easy to implement.



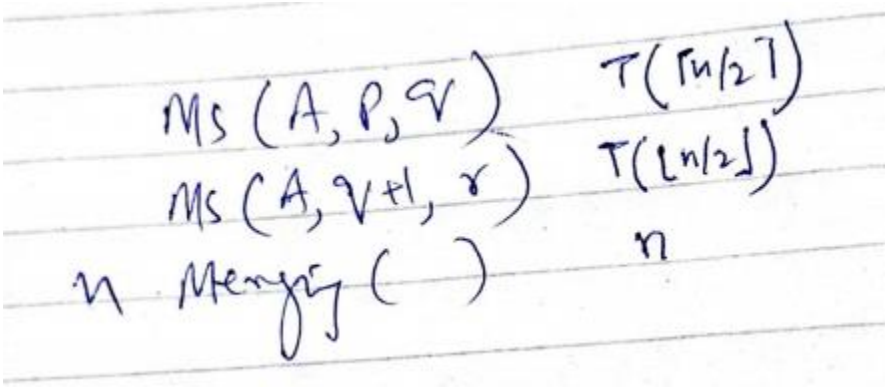
Where p is beginning index of array list A and r is ending index of array list A

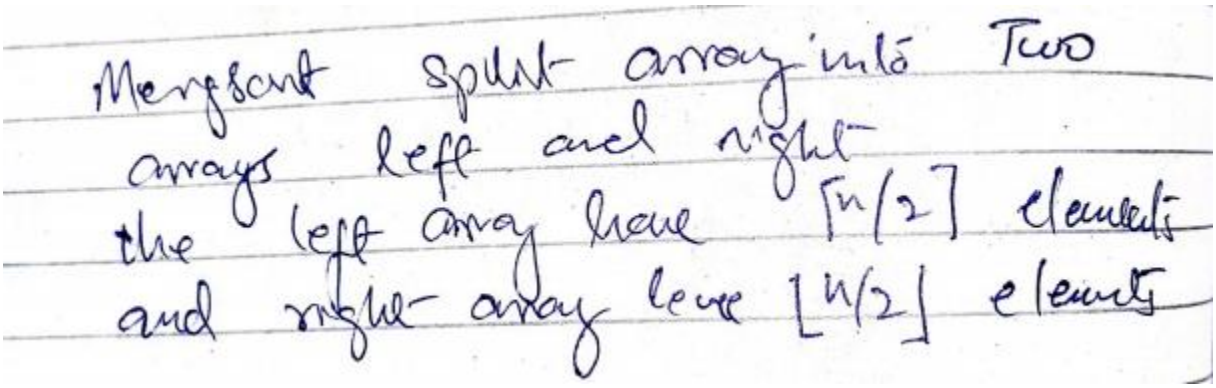
Merge Sort Algorithm

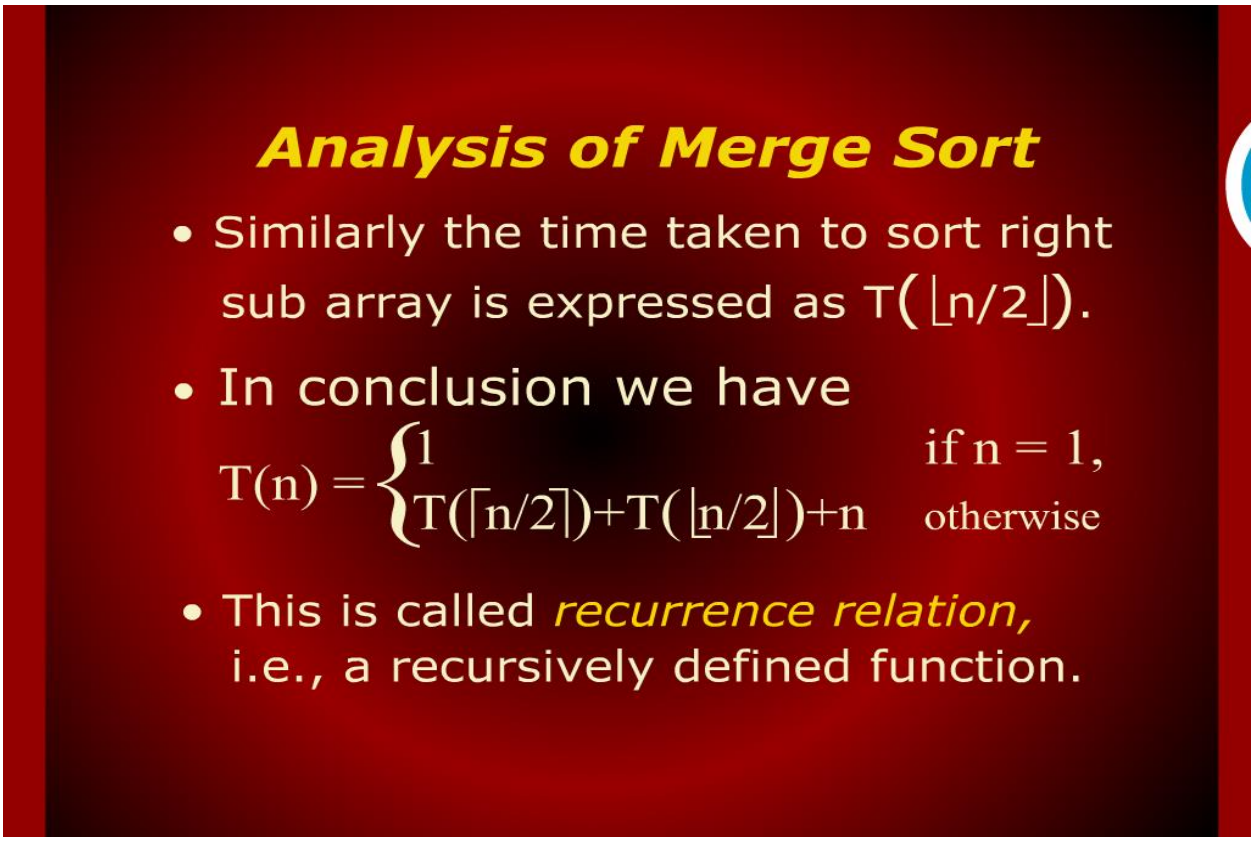
```
MERGE( array A, int p, int q, int r)
1  int B[p..r]; int i ← k ← p; int j ← q + 1
2    while (i ≤ q) and ( j ≤ r)
3    do if (A[i] ≤ A[ j])
4          then B[k++ ] ← A[i++ ]
5          else B[k++ ] ← A[ j++ ]
6    while (i ≤ q)
7    do B[k++ ] ← A[i++ ]
8    while ( j ≤ r)
```

```
9    do B[k++ ] ← A[ j++ ]
10   for i ← p to r
11   do A[i] ← B[i]
```

Time analysis







Lets expand the above equation terms

We know that if $n = 1$ then

$T(1) = 1$

If $n = 2\ then$

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

Substitute the values of $T(1)$ in the above equation we get

$T(2) = 1 + 1 + 2 = 4$

$if\ n = 3\ then$

$T(3) = T(2) + T(1) + 3$

Substitute the values of $T(1)\ and\ T(2)\ we\ get$

$T(3) = 4 + 1 + 3 = 8$

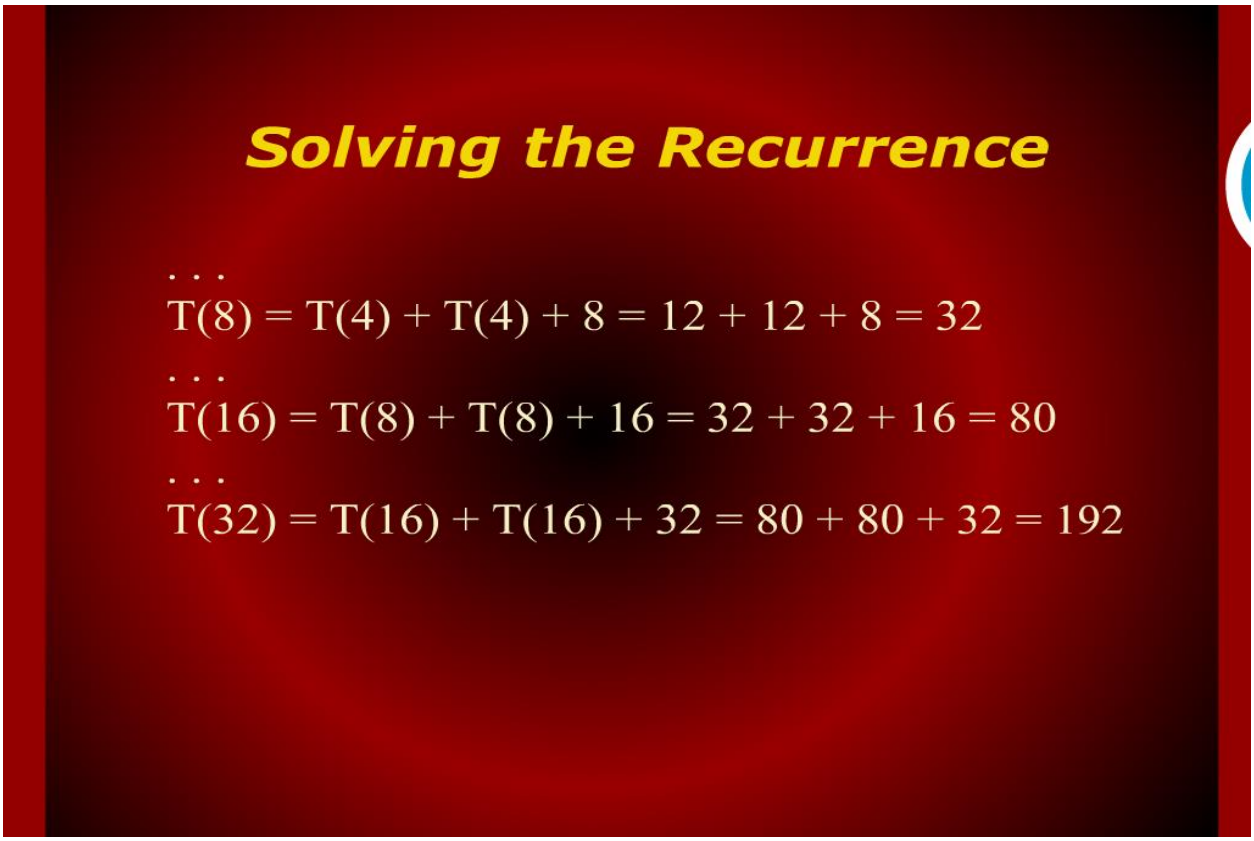
$if\ n = 4\ then$

$T(4) = T(2) + T(2) + 4$

Substitute the values of $T(2)$ $we\ get$

$T(4) = 4 + 4 + 4 = 12$

Rest are given below in the slide



Solving the Recurrence

What is the pattern here?

- Let's consider the ratios T(n)/n for powers of 2:

  $T(1)/1 = 1$ $\quad$ $T(8)/8 = 4$

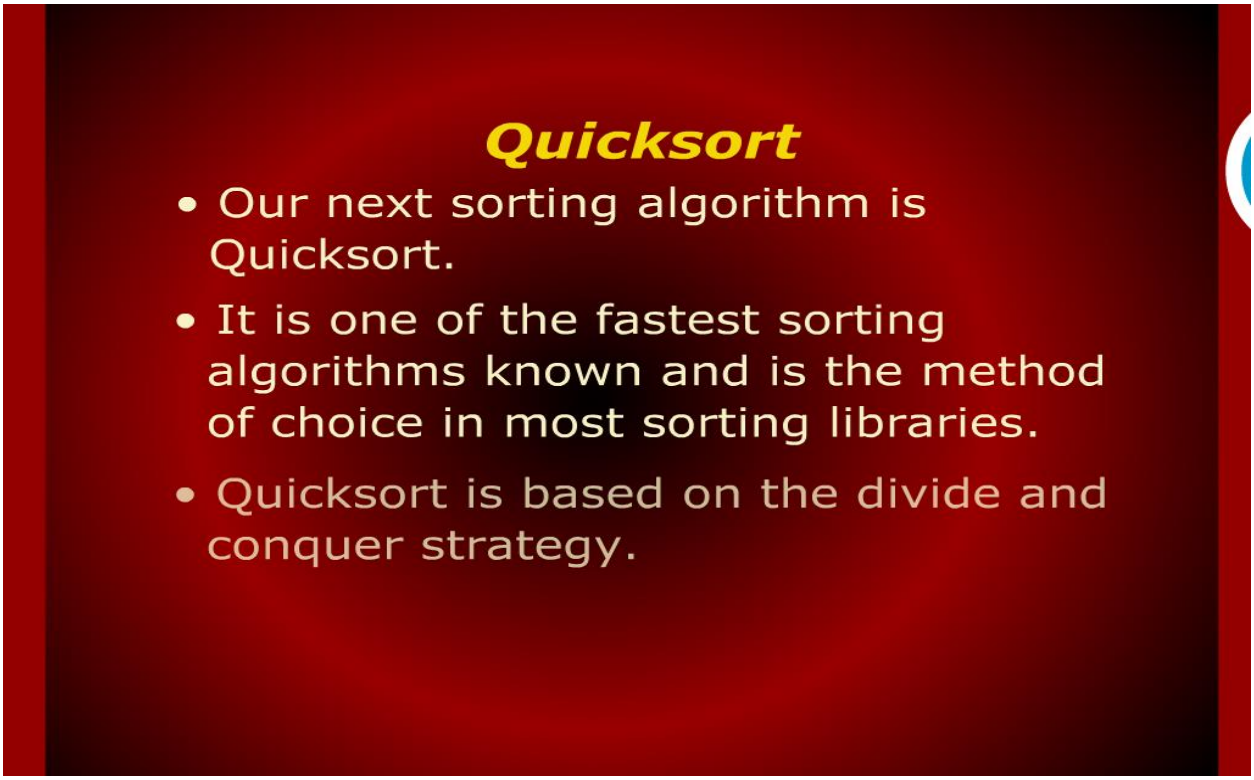
  $T(2)/2 = 2$ $\quad$ $T(16)/16 = 5$

  $T(4)/4 = 3$ $\quad$ $T(32)/32 = 6$

- This suggests $T(n)/n = \log n + 1$
- Or, $T(n) = n \log n + n$ which is $\Theta(n \log n)$ (use the limit rule).

NOW READ CHAPTER -4 SORTING (PDF File)

Available at github



Quicksort

```
QUICKSORT( array A, int p, int r)
1  if (r > p)
2     then
3        i ← a random index from [p..r]
4          swap A[i] with A[p]
5        q ← PARTITION(A, p, r)
6        QUICKSORT(A, p, q – 1)
7        QUICKSORT(A, q + 1, r)
```

Partition Algorithm

Recall that the partition algorithm partitions the array A[p..r] into three sub arrays about a pivot element x.

- A[p..q – 1] whose elements are less than or equal to x,
- A[q] = x,
- A[q + 1..r] whose elements are greater than x

Choosing the Pivot

- We will choose the first element of the array as the pivot, i.e. x = A[p].
- If a different rule is used for selecting the pivot, we can swap the chosen element with the first element.
- We will choose the pivot randomly.

Partition Algorithm

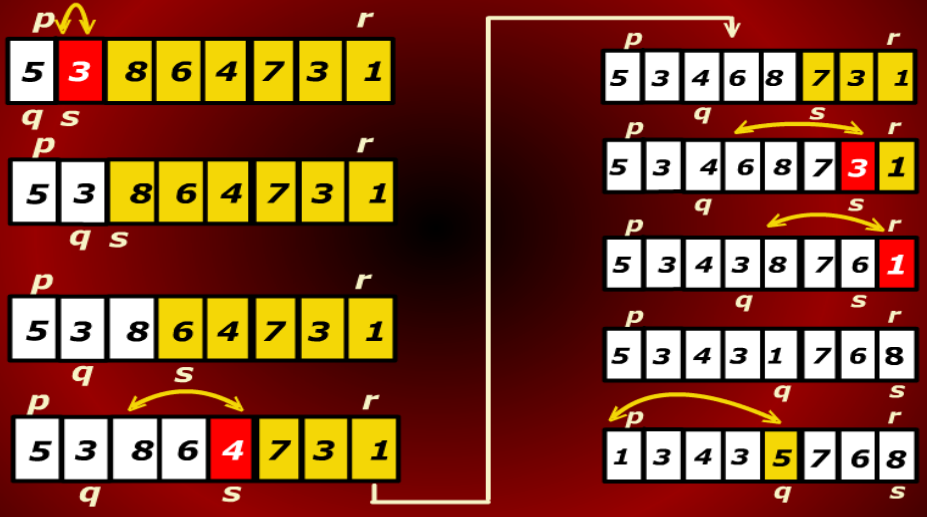
The algorithm works by maintaining the following *invariant condition.*

1. A[p] = x is the pivot value.
2. A[p..q – 1] contains elements that are less than x.
3. A[q + 1..s – 1] contains elements that are greater than or equal to x
4. A[s..r] contains elements whose values are currently unknown.

Partition Algorithm

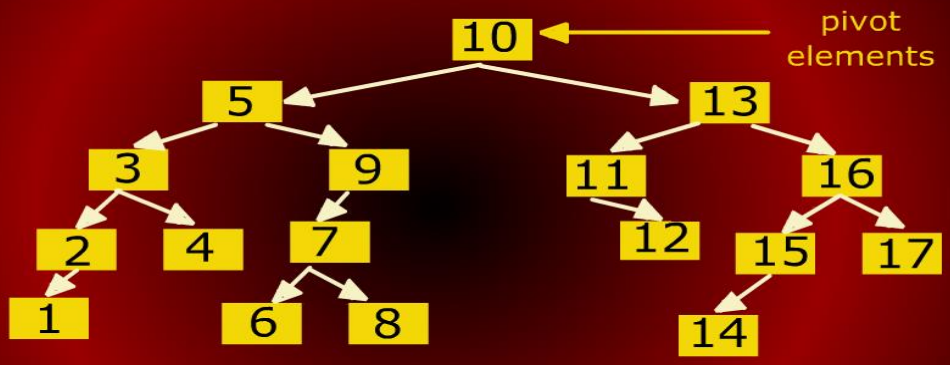
```
PARTITION( array A, int p, int r)
1   x ← A[p]
2   q ← p
3   for s ← p + 1 to r
4   do if (A[s] < x)
5         then q ← q + 1
6              swap A[q] with A[s]
7
8    swap A[p] with A[q]
9   return q
```

Partition Algorithm



It is interesting to note (but not surprising) that the pivots form a binary search tree

Quicksort BST



Analysis of Quicksort

- The running time of quicksort depends heavily on the selection of the pivot.
- If the rank of the pivot is very large or very small then the partition (BST) will be unbalanced.
- Since the pivot is chosen randomly in our algorithm, the expected running time is O(n log n).

Analysis of Quicksort

- It takes $\Theta(n)$ time to do the partitioning.
- The time taken for sorting array A[1..n] is

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

- To get the worst case, we maximize over all possible values of q.

Analysis of Quicksort

Putting is together, we get the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \max_{1 \leq q \leq n}(T(q-1)+T(n-q)+n) & \text{otherwise} \end{cases}$$

The worse case happens at q = 1 or q = n. If we expand the recurrence for q = 1, we get:

Analysis of Quicksort

$$
\begin{aligned}
T(n) &\le T(0) + T(n-1) + n \\
&= 1 + T(n-1) + n \\
&= T(n-1) + (n+1) \\
&= T(n-2) + n + (n+1) \\
&= T(n-3) + (n-1) + n + (n+1) \\
&= T(n-4) + (n-2) + (n-1) + n \\
&\quad + (n+1) \\
&= T(n-k) + \sum_{i=-1}^{k-2} (n-i)
\end{aligned}
$$

Analysis of Quicksort

For the basis T(1) = 1 we set k = n– 1 and get

$$T(n) \le T(1) + \sum_{i=-1}^{n-3} (n - i)$$

$$= 1 + (3 + 4 + 5 + \cdots + (n - 1) + n + (n + 1))$$

$$\le \sum_{i=1}^{n+1} i = \frac{(n + 1)(n + 2)}{2} \in \Theta(n^2)$$

Selection Problem

- Suppose we are given a set of n numbers.
- Define the *rank* of an element to be one plus the number of elements that are smaller.
- Thus, the rank of an element is its final position if the set is sorted.
- The minimum is of rank 1 and the maximum is of rank n.

Medians and Selection

- Consider the set: {5, 7, 2, 10, 8, 15, 21, 37, 41}.
- The rank of each number is its position in the sorted order.

position	1	2	3	4	5	6	7	8	9
Number	2	5	7	8	10	15	21	37	41

- For example, rank of 8 is 4, one plus the number of elements smaller than 8 which is 3.

The Selection Problem

Given a set A of n distinct numbers and an integer k, $1 \leq k \leq n$, output the element of A of rank k.

Median

- Of particular interest in statistics is the *median.*
- If n is odd then the median is defined to be element of rank $(n + 1)/2$.
- When n is even, there are two choices: $n/2$ and $(n + 1)/2$.
- In statistics, it is common to return the average of the two elements.

Median

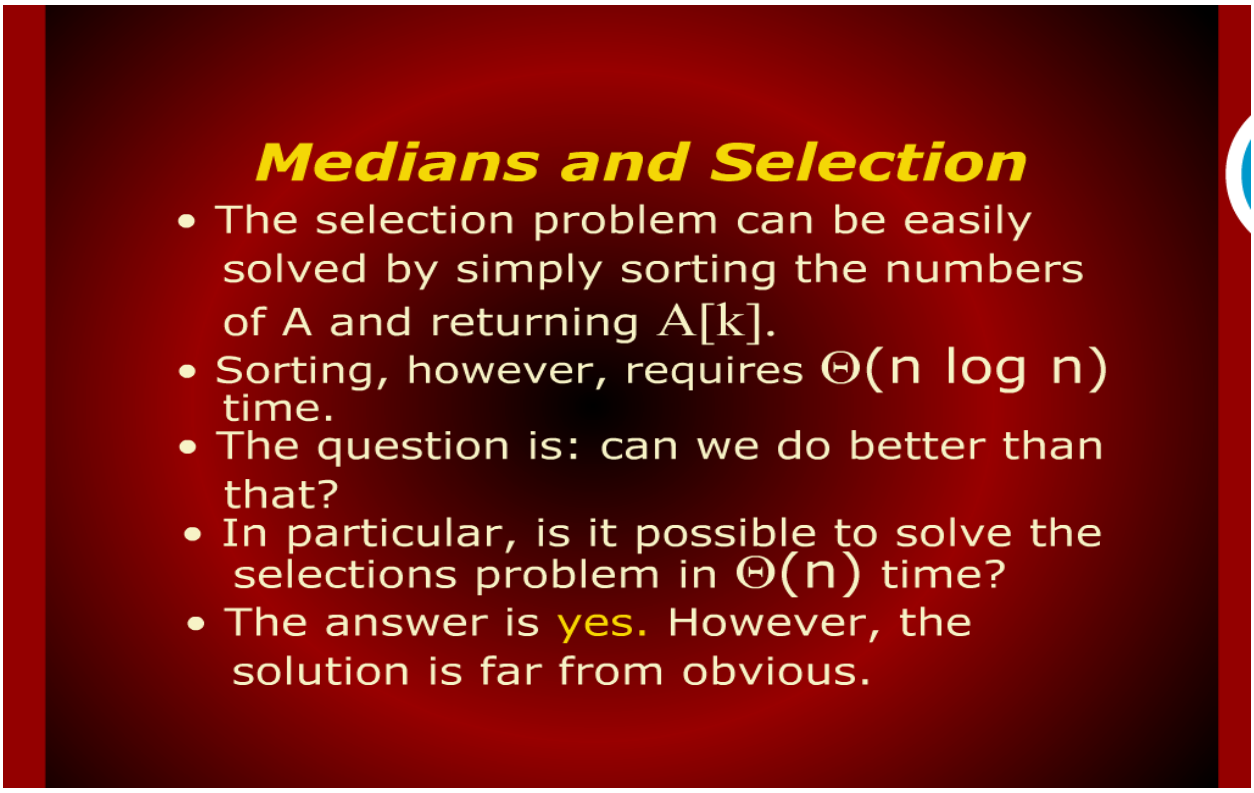
- Medians are useful as a measures of the *central tendency* of a set especially when the distribution of values is highly skewed.
- For example, the median income in a community is a more meaningful measure than average.
- Suppose 7 households have monthly incomes 5000, 7000, 2000, 10000, 8000, 15000 and 16000.

Median

- In sorted order, the incomes are 2000, 5000, 7000, 8000, 10000, 15000, 16000.
- The median income is 8000; median is element with rank 4: $(7 + 1)/2 = 4$.
- The average income is 9000.

Median

- Suppose the income 16000 goes up to 450,000.
- The median is still 8000 but the average goes up to 71,000.
- Clearly, the average is not a good representative of the majority income levels.



SIEVE TECHNIQUE

Medians and Selection

- We will use a variation of the divide and conquer strategy called the *sieve technique*.
- In divide and conquer, we break the problem into a small number of smaller subproblems which we solve recursively.

Medians and Selection

- In the selection problem, we are looking for an item.
- We will divide the problem into subproblems.
- However, we will discard those small subproblems for which we determine that they do not contain the desired answer *(the variation)*.

Medians and Selection

Here is how the sieve technique will be applied to the selection problem.

- We will begin with the given array $A[1..n]$.
- We will pick an item from the array, called the *pivot element* which we will denote by x.
- We will talk about how an item is chosen as the pivot later; for now just think of it as a random element of A.

Medians and Selection

We partition A into three parts.

1. A[q] contains the pivot element x,
2. A[1..q −1] will contain all the elements that are less than x and
3. A[q + 1..n] will contains all elements that are greater than x.

Within each sub array, the items may appear in any order.

Partitioning

A[p..r] partitioned about the pivot x.

Medians and Selection

- The rank of the pivot x is $q - p + 1$ in $A[p..r]$.
- Let $rank\_x = q - p + 1$.
- If $k = rank\_x$ then the pivot is k^{th} smallest.
- If $k < rank\_x$ then search $A[p..q - 1]$ recursively.

Medians and Selection

- If $k = rank\_x$ then the pivot is k^{th} smallest.
- If $k < rank\_x$ then search $A[p..q - 1]$ recursively.
- If $k > rank\_x$ then search $A[q + 1..r]$ recursively. Find element of rank $(k - q)$ because we eliminated q smaller elements in A.

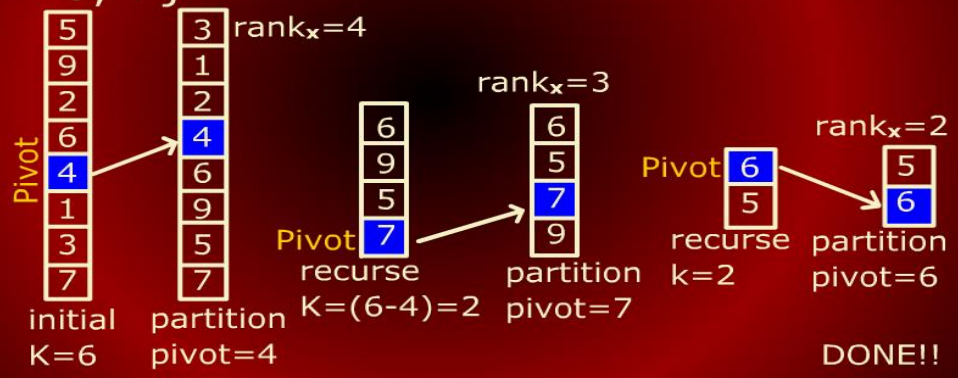
Select Algorithm

```
SELECT( array A, int p, int r, int k)
1 if (p = r)
2   then return A[p]
3   else x ← CHOOSE PIVOT(A, p, r)
4   q ← PARTITION(A, p, r, x)
5   rank_x ← q – p + 1
6   if k = rank_x
7     then return x
8   if k < rank x
```

```
9     then return SELECT(A, p, q–1, k)
10 else return SELECT(A, q+1, r, k–q)
```

Select Algorithm

- Example: select the 6th smallest element of the set {5, 9, 2,6, 4, 1, 3, 7}



Medians and Selection

- We will discuss how to choose a pivot and the partitioning later.
- For the moment, we will assume that they both take $\Theta(n)$ time.
- How many elements do we eliminate in each time?

Medians and Selection

- If x is the largest or the smallest then we may only succeed in eliminating one element.

5, 9, 2, 6, 4, [1], 3, 7 pivot is 1

[1], 5, 9, 2, 6, 4, 3, 7 after partition

- Ideally, x should have a rank that is neither too large or too small.

Analysis of Selection

- Suppose we are able to choose a pivot that causes exactly half of the array to be eliminated in each phase.
- This means that we recurse on the remaining $n/2$ elements.
- This leads to the following recurrence:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(n/2) + n & \text{otherwise} \end{cases}$$

Analysis of Selection

If we expand this recurrence, we get

$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \cdots$$

$$\leq \sum_{i=0}^{\infty} \frac{n}{2^i}$$

$$= n \sum_{i=0}^{\infty} \frac{1}{2^i}$$

Analysis of Selection

- Recall the formula for infinite geometric series; for any |c| < 1,

$$\sum_{i=0}^{\infty} c^i = \frac{1}{1 - c}$$

- Using this we have

$$T(n) \leq 2n \in \Theta(n)$$

Analysis of Selection

- Let's think about how we ended up with a $\Theta(n)$ algorithm for selection.
- Normally, a $\Theta(n)$ time algorithm would make a single or perhaps a constant number of passes of the data set.
- In this algorithm, we make a number of passes. In fact it could be as many as log n.

Analysis of Selection

- However, because we eliminate a constant fraction of the array with each phase, we get the convergent geometric series in the analysis.
- This shows that the total running time is indeed *linear* in n.
- This lesson is well worth remembering.
- It is often possible to achieve linear running times in ways that you would not expect.

Sorting

Sorting

- For the next series of lectures, we will focus on sorting.
- There a number of reasons for sorting. Here are a few important ones.
- Procedures for sorting are parts of many large software systems.
- Design of efficient sorting algorithms is necessary to achieve overall efficiency of these systems.

Sorting

- Sorting is well studied problem from the analysis point of view.
- Sorting is one of the few problems where provable lower bounds exist on how fast we can sort.
- In sorting, we are given an array A[1..n] of n numbers
- We are to reorder these elements into increasing (or decreasing) order.

Sorting

- More generally, A is an array of objects and we sort them based on one of the attributes - the key value.
- The key value need not be a number. It can be any object from a totally ordered domain.
- Totally ordered domain means that for any two elements of the domain, x and y, either $x < y$, $x = y$ or $x > y$.

Slow Sorting Algorithms

Bubble sort

- Scan the array.
- Whenever two consecutive items are found that are out of order, swap them.
- Repeat until all consecutive items are in order.

Slow Sorting Algorithms

Insertion sort

- Assume that A[1..i – 1] have already been sorted.
- Insert A[i] into its proper position in this sub array.
- Create this position by shifting all larger elements to the right.

Slow Sorting Algorithms

Selection sort

- Assume that A[1 .. i–1] contain the i–1 smallest elements in sorted order.
- Find the smallest element in A[i..n]
- Swap it with A[i].

Slow Sorting Algorithms

- These algorithms are easy to implement.
- But they run in $\Theta(n^2)$ time in the worst case.

n log n Sorting

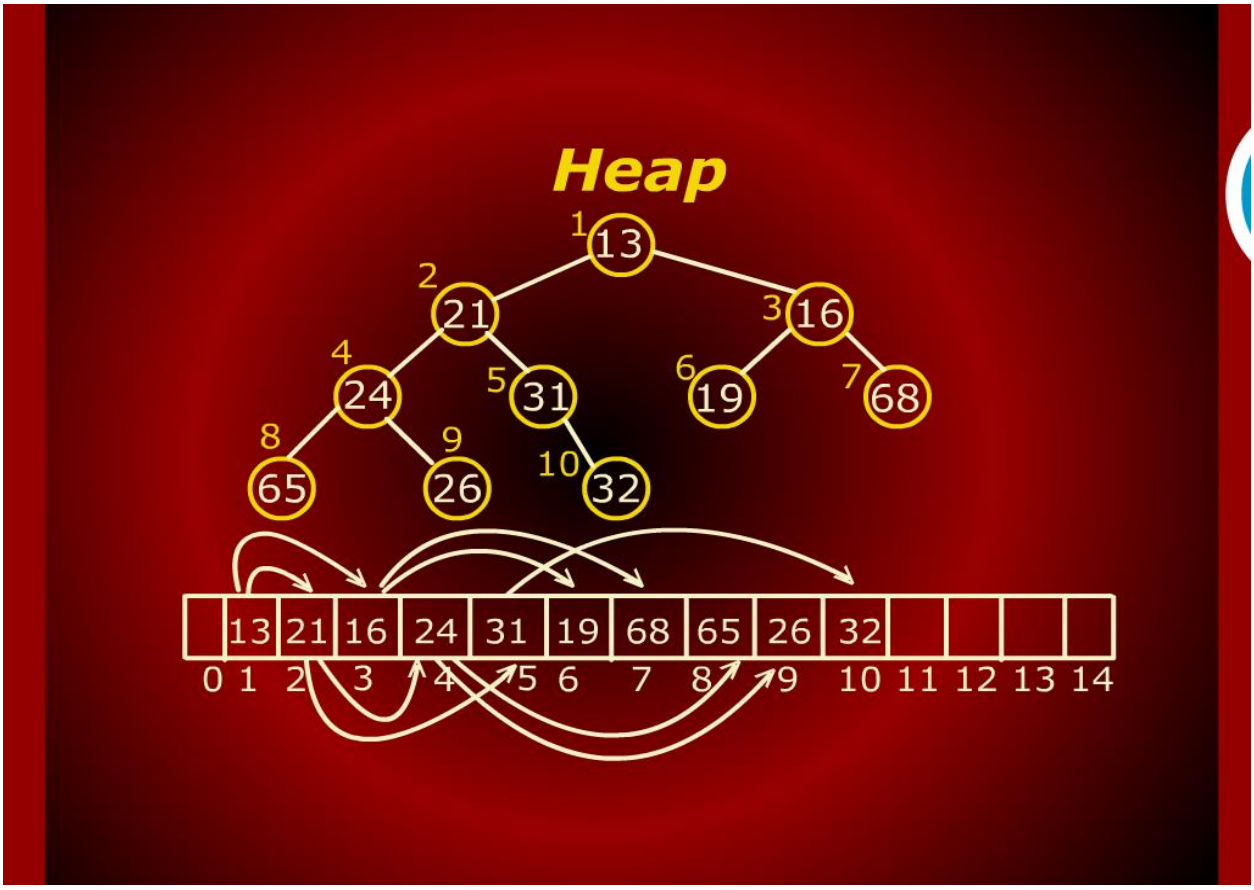
- We have already seen that Mergesort sorts an array of numbers in $\Theta(n \log n)$ time.
- We will study two others: *Heapsort* and *Quicksort.*

Heap

- A heap is a left-complete binary tree that conforms to the *heap order.*
- The heap order property: in a (min) heap, for every node X, the key in the parent is smaller than or equal to the key in X.
- In other words, the parent node has key smaller than or equal to both of its children nodes.

Heap

- Similarly, in a max heap, the parent has a key larger than or equal both of its children
- Thus the smallest key is in the root in a min heap; in the max heap, the largest is in the root.



Heap Sort
Quick Sort
O(nlogn)

Bubble Sort
Insertion Sort
Selection Sort
O(n2)

Linear Time Sorting

- The lower bound implies that if we hope to sort numbers faster than O(n log n), we cannot do it by making comparisons alone.
- Is it possible to sort without making comparisons?
- The answer is yes, but only under very restrictive circumstances.

Counting Sort

- We will consider three algorithms that are faster and work by not making comparisons.
- Counting sort assumes that the numbers to be sorted are in the range 1 to k where k is small.
- The basic idea is to determine the rank of each number in final sorted array.

Counting Sort

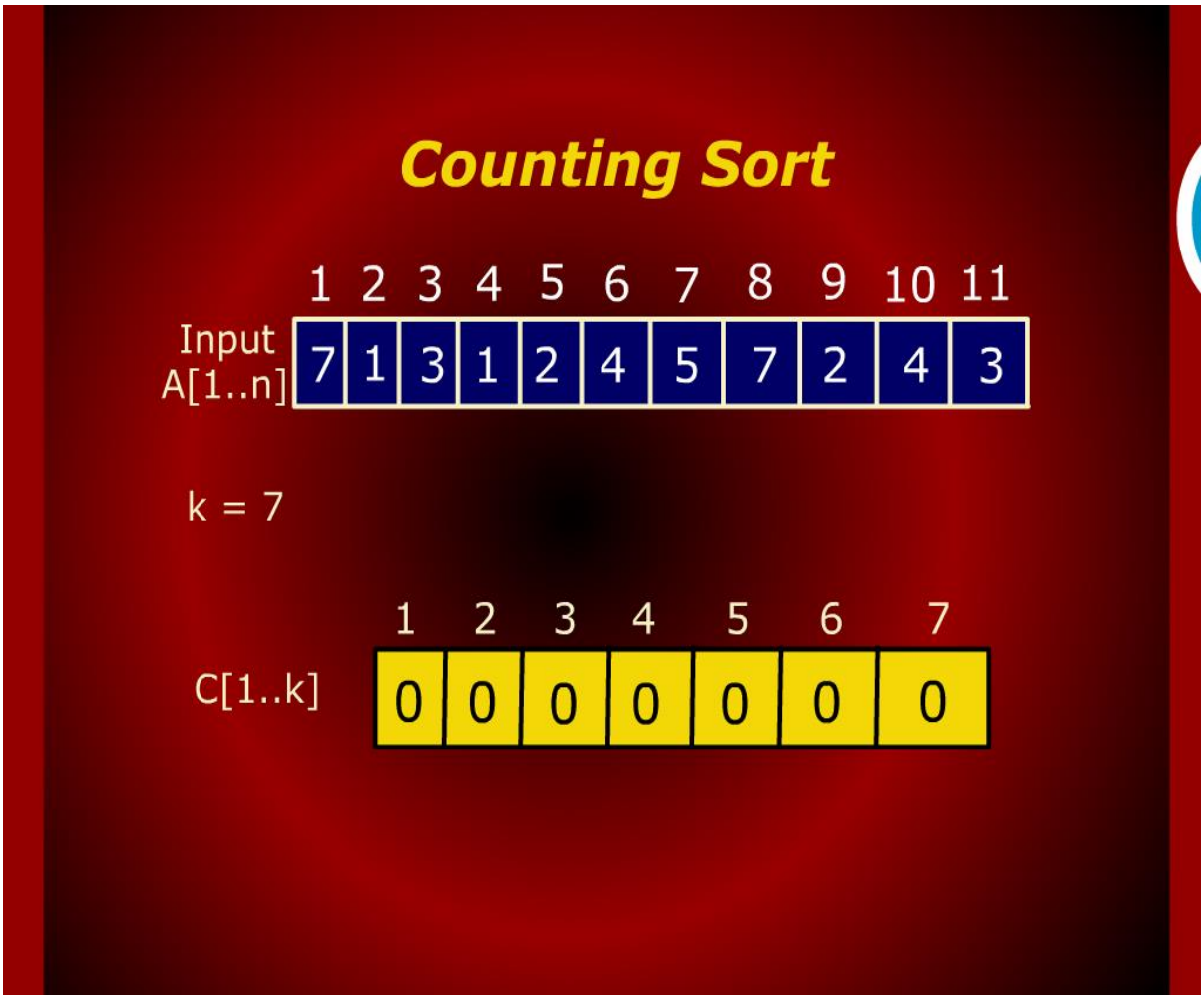
- Recall that the rank of an item is the number of elements that are less than or equal to it.
- Once we know the ranks, we simply copy numbers to their final position in an output array.
- The algorithm sorts in $\Theta(n + k)$.
- If k is $\Theta(n)$ then counting sort is an $\Theta(n)$ time algorithm.

Counting Sort

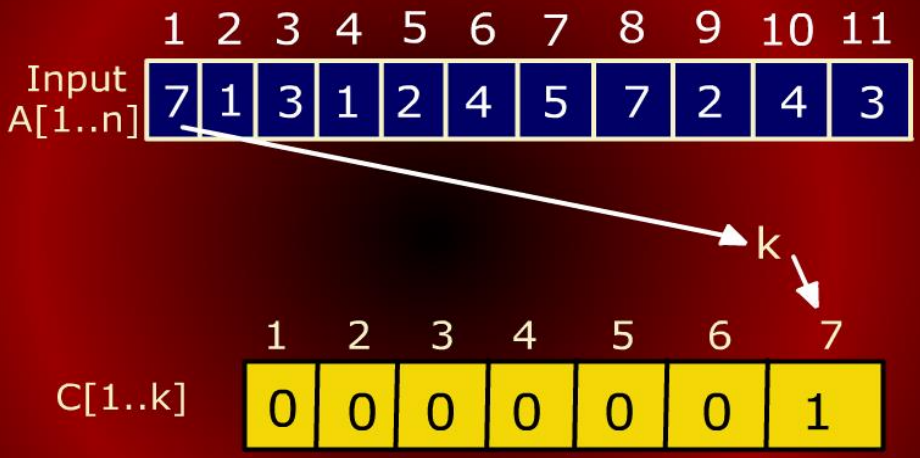
The algorithm uses three arrays.

1. A[1..n]: Holds the initial input.
2. B[1..n]: Array that holds the sorted output.
3. C[1..k]: Array of integers. C[x] is the rank of x in A, where $x \in [1..k]$.

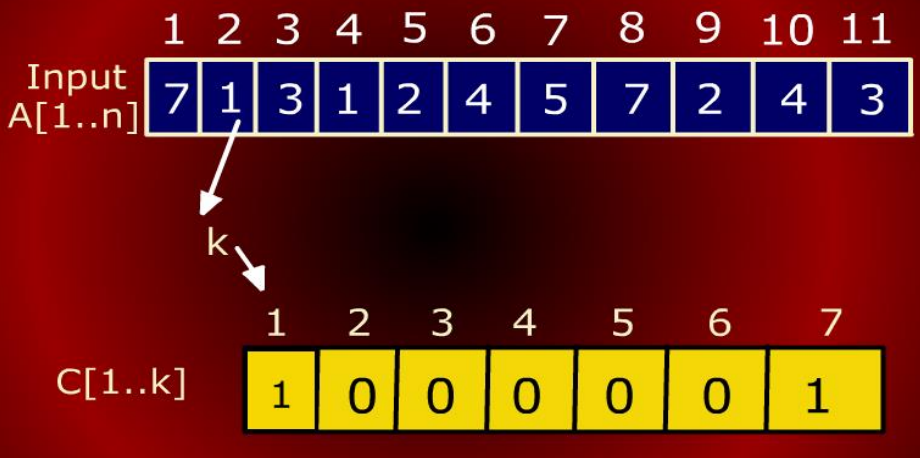
Counting Sort

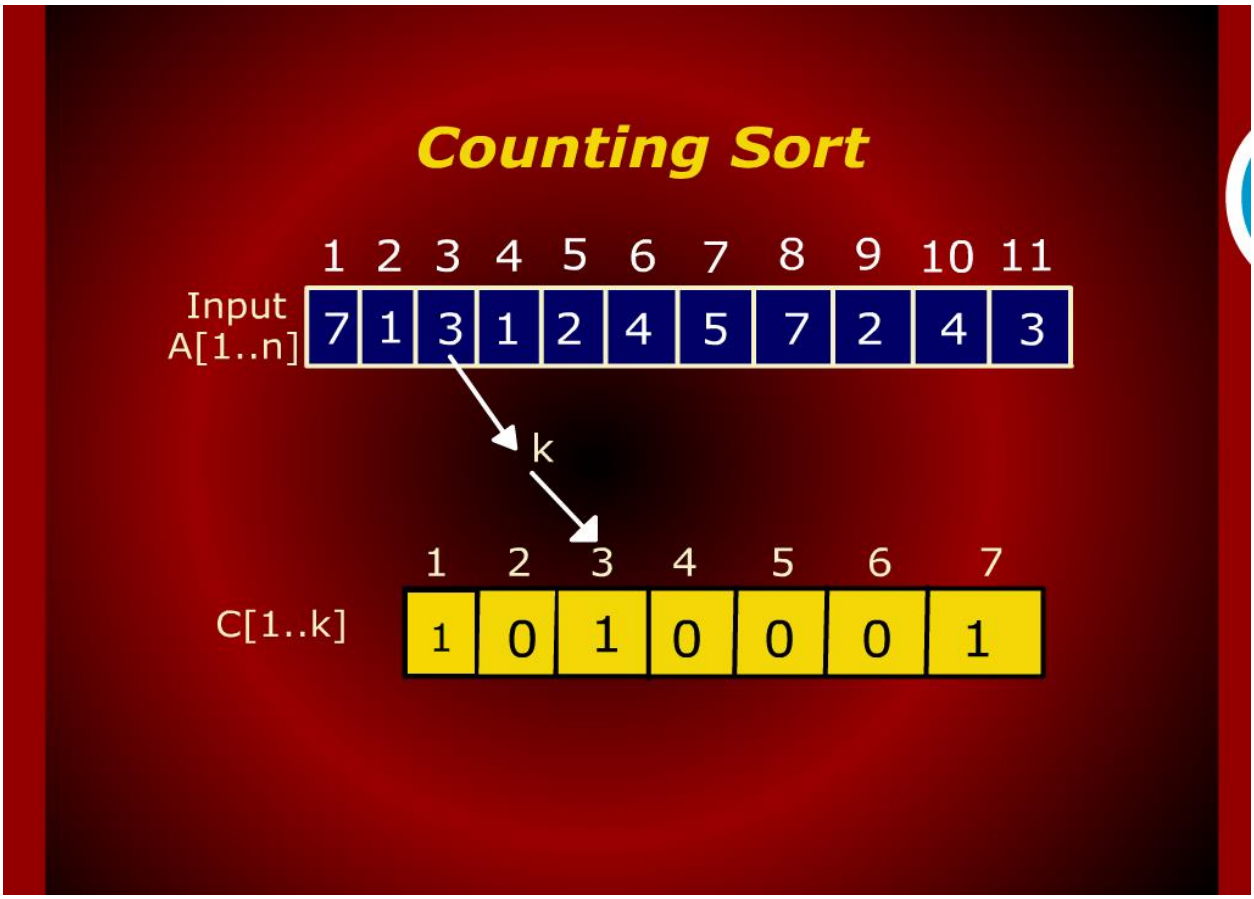


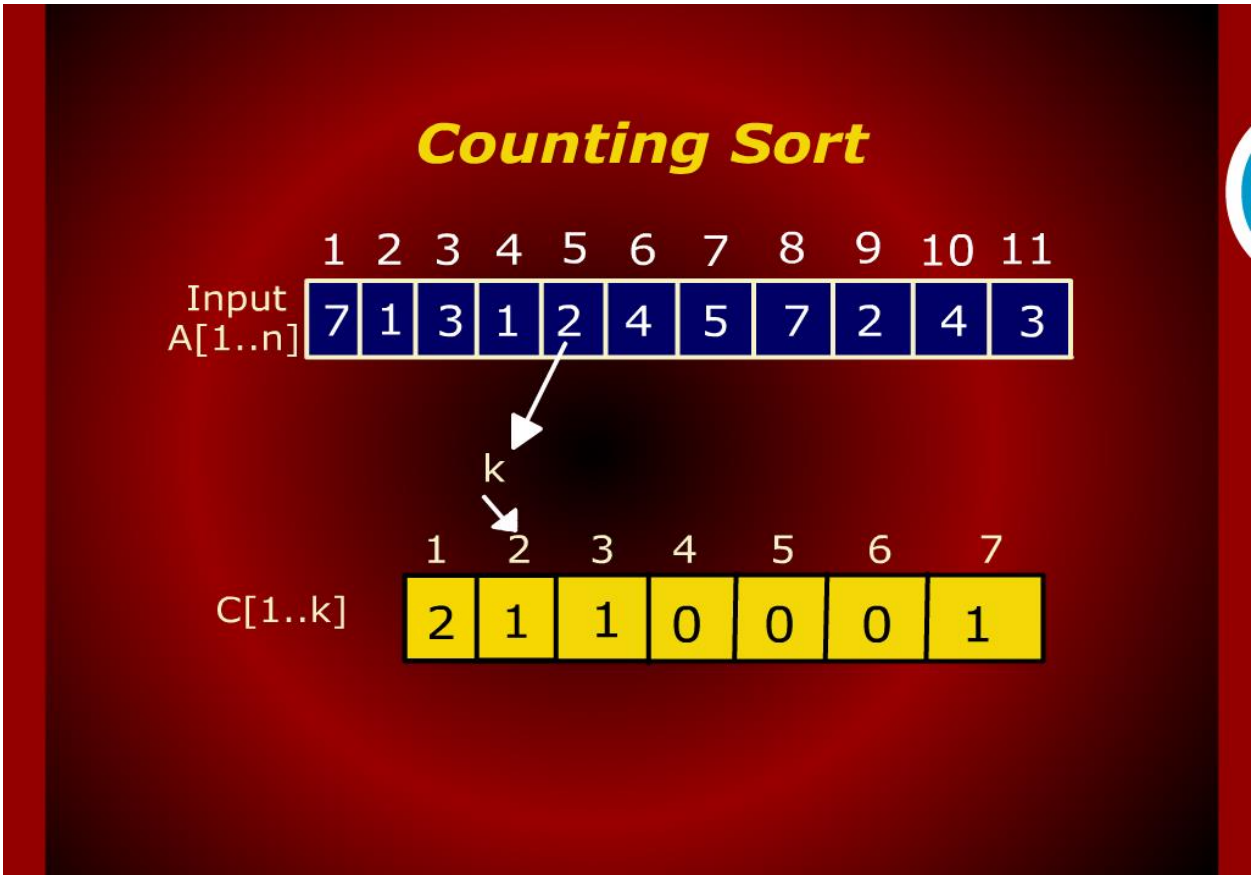
Counting Sort



Counting Sort




Counting Sort
1 2 3 4 5 6 7 8 9 10 11
Input
A[1..n]
7 1 3 1 2 4 5 7 2 4 3
k
1 2 3 4 5 6 7
C[1..k]
1 0 1 0 0 0 1


Counting Sort
1 2 3 4 5 6 7 8 9 10 11
Input
A[1..n]
7 1 3 1 2 4 5 7 2 4 3
k
1 2 3 4 5 6 7
C[1..k]
2 1 1 0 0 0 1

Counting Sort

	1	2	3	4	5	6	7	8	9	10	11
Input A[1..n]	7	1	3	1	2	4	5	7	2	4	3

Finally

	1	2	3	4	5	6	7
C[1..k]	2	2	2	2	1	0	2

Counting Sort

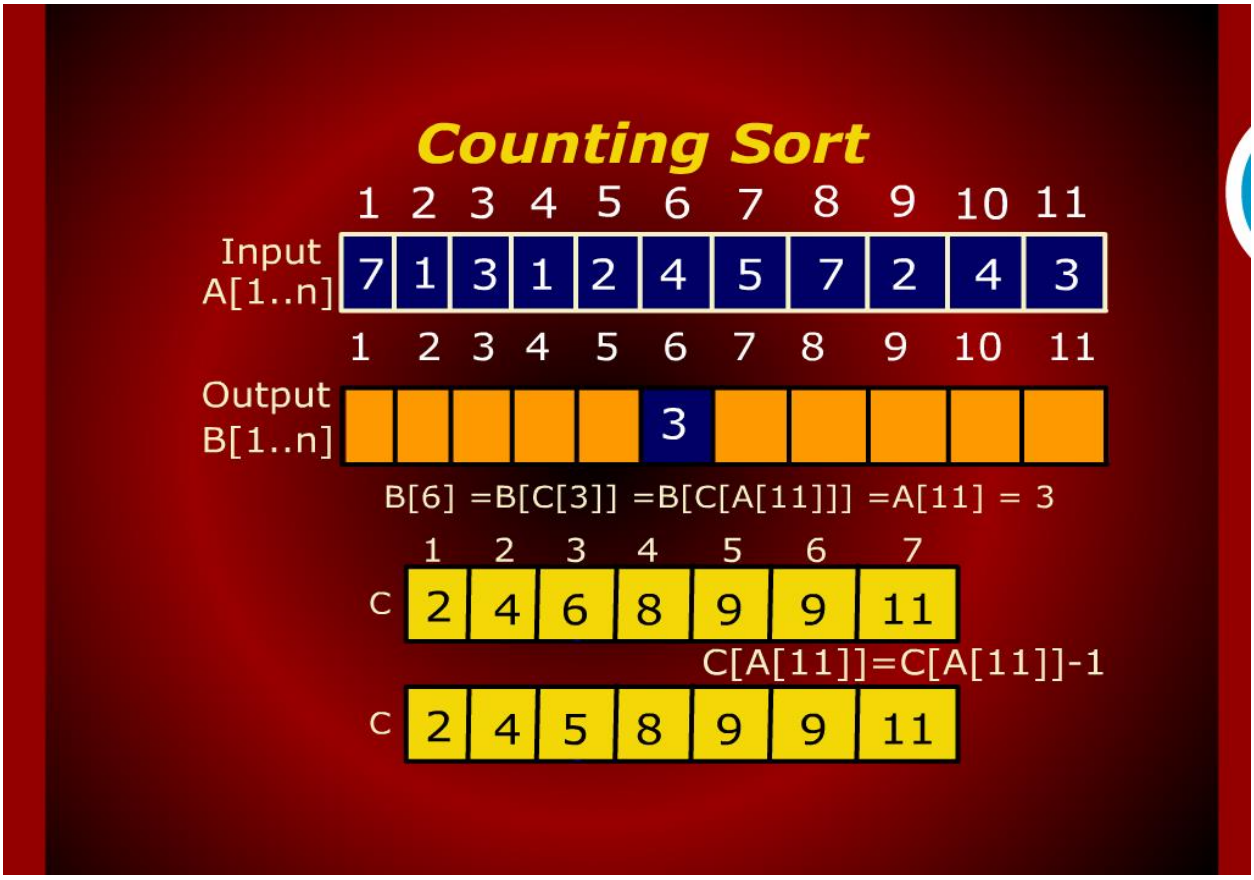
	1	2	3	4	5	6	7	8	9	10	11
Input A[1..n]	7	1	3	1	2	4	5	7	2	4	3

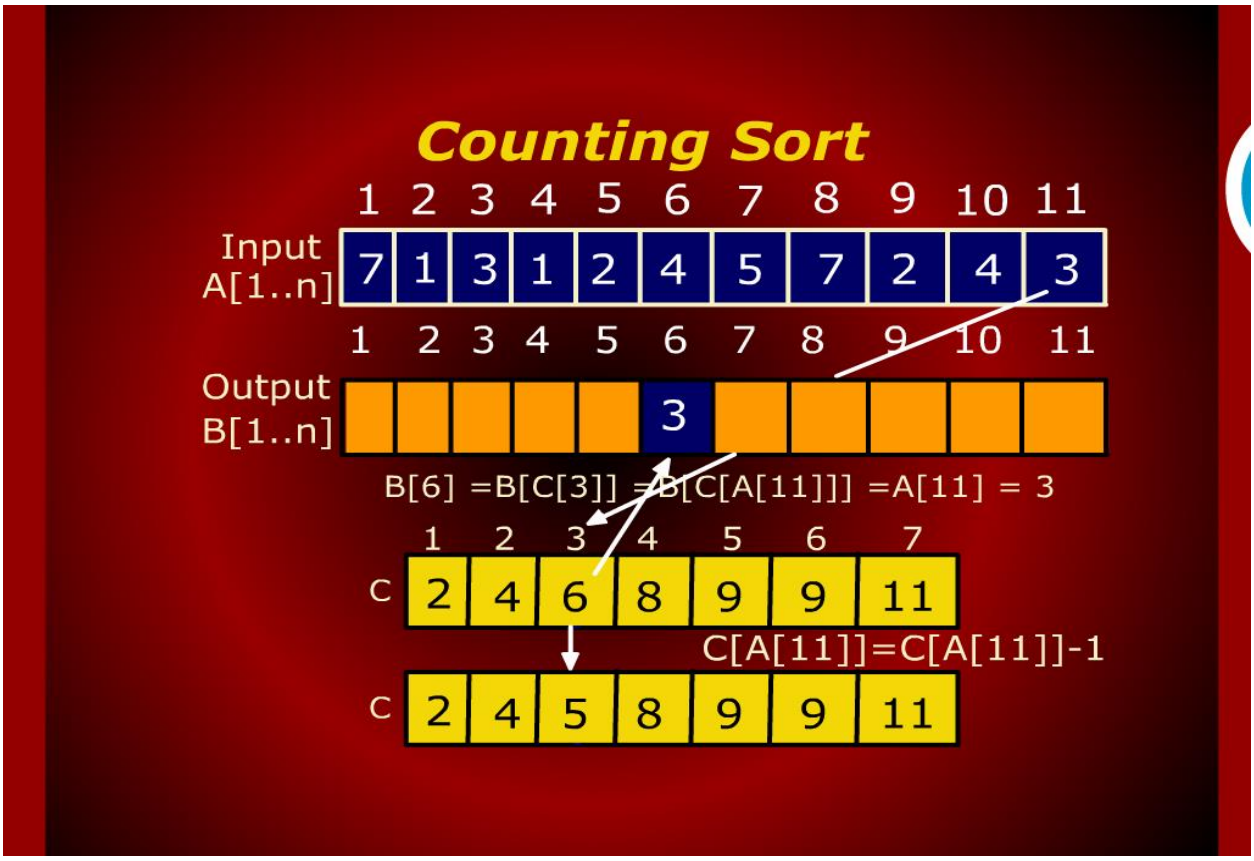
	1	2	3	4	5	6	7
C[1..k]	2	2	2	2	1	0	2

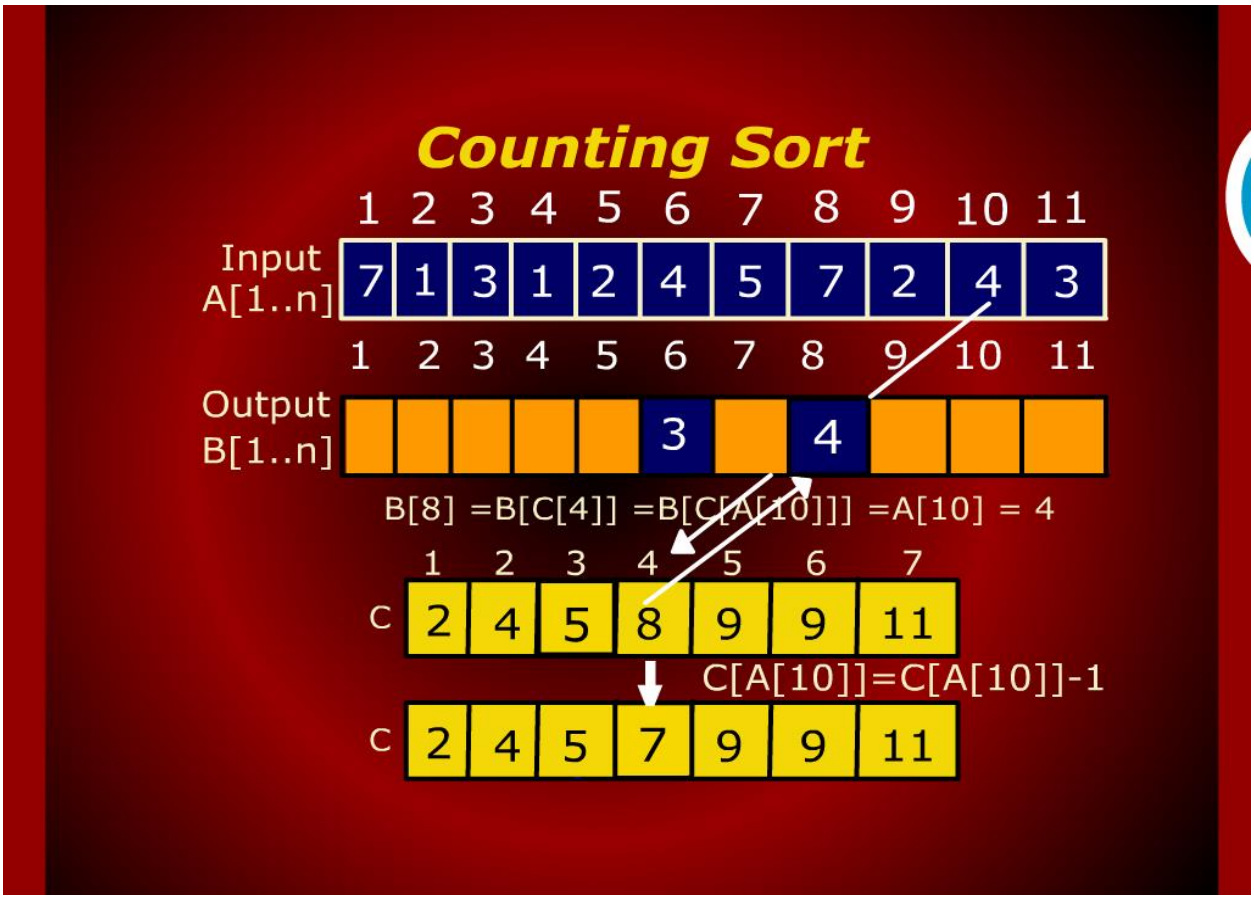
for i= 2 to 7
doC[i] =C[i] +C[i- 1]

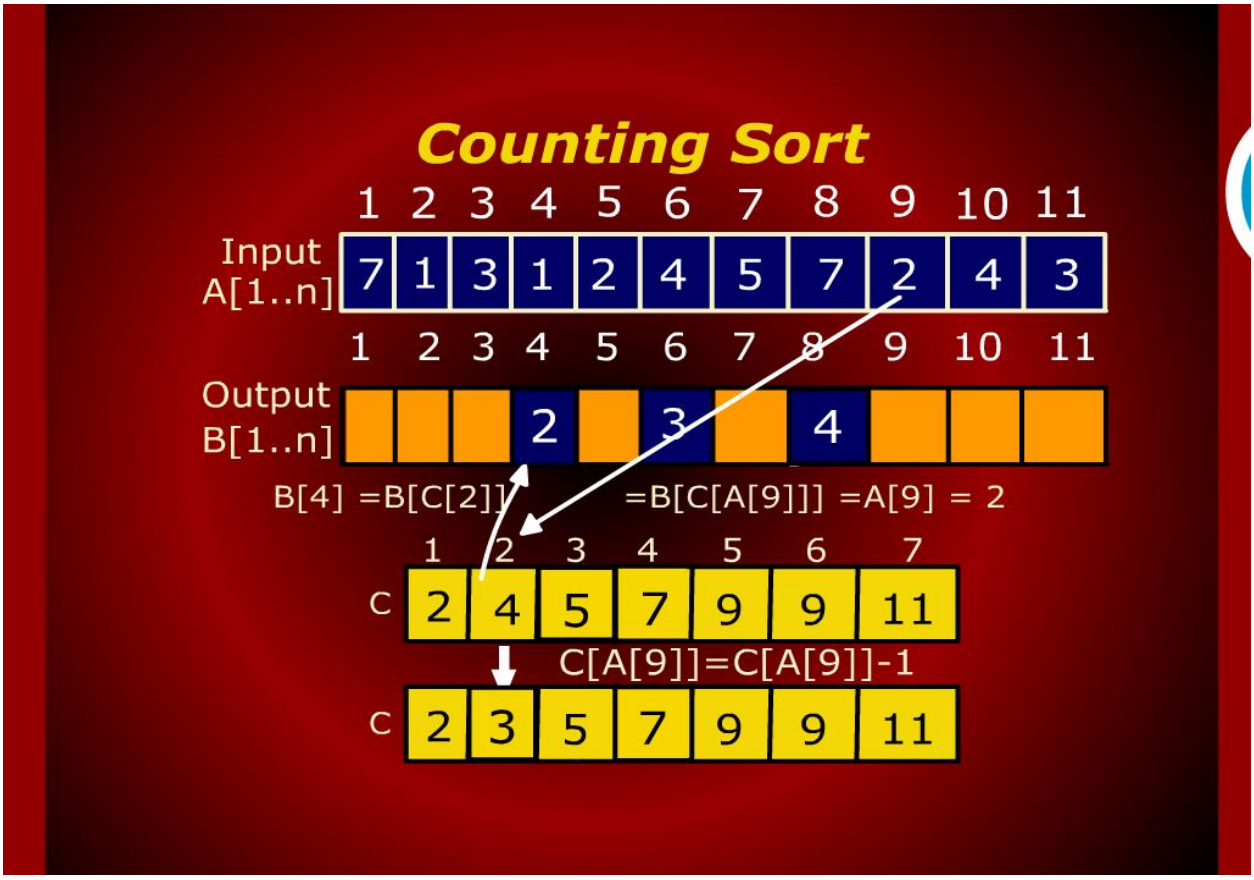
	1	2	3	4	5	6	7
C	2	4	6	8	9	9	11

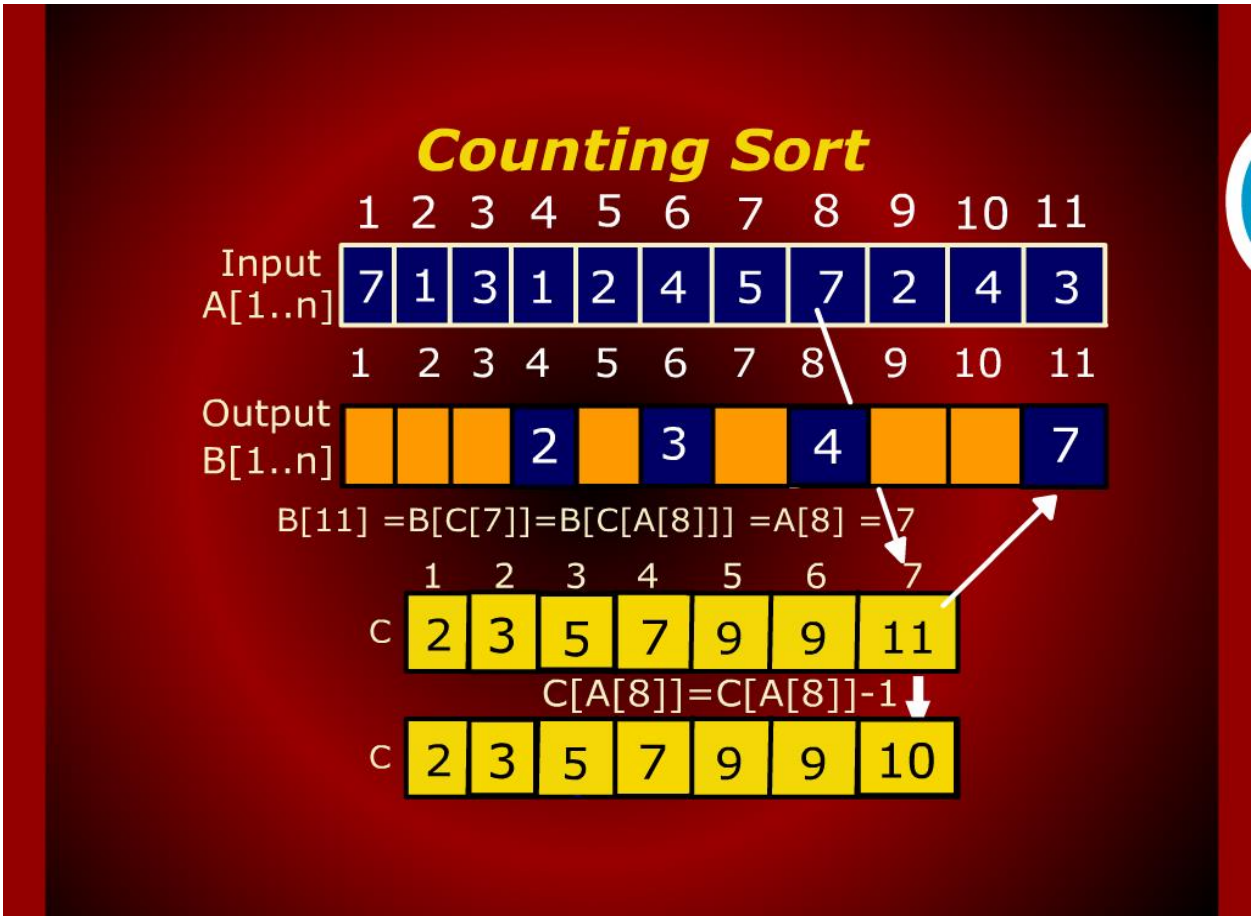
6 elements <= 3

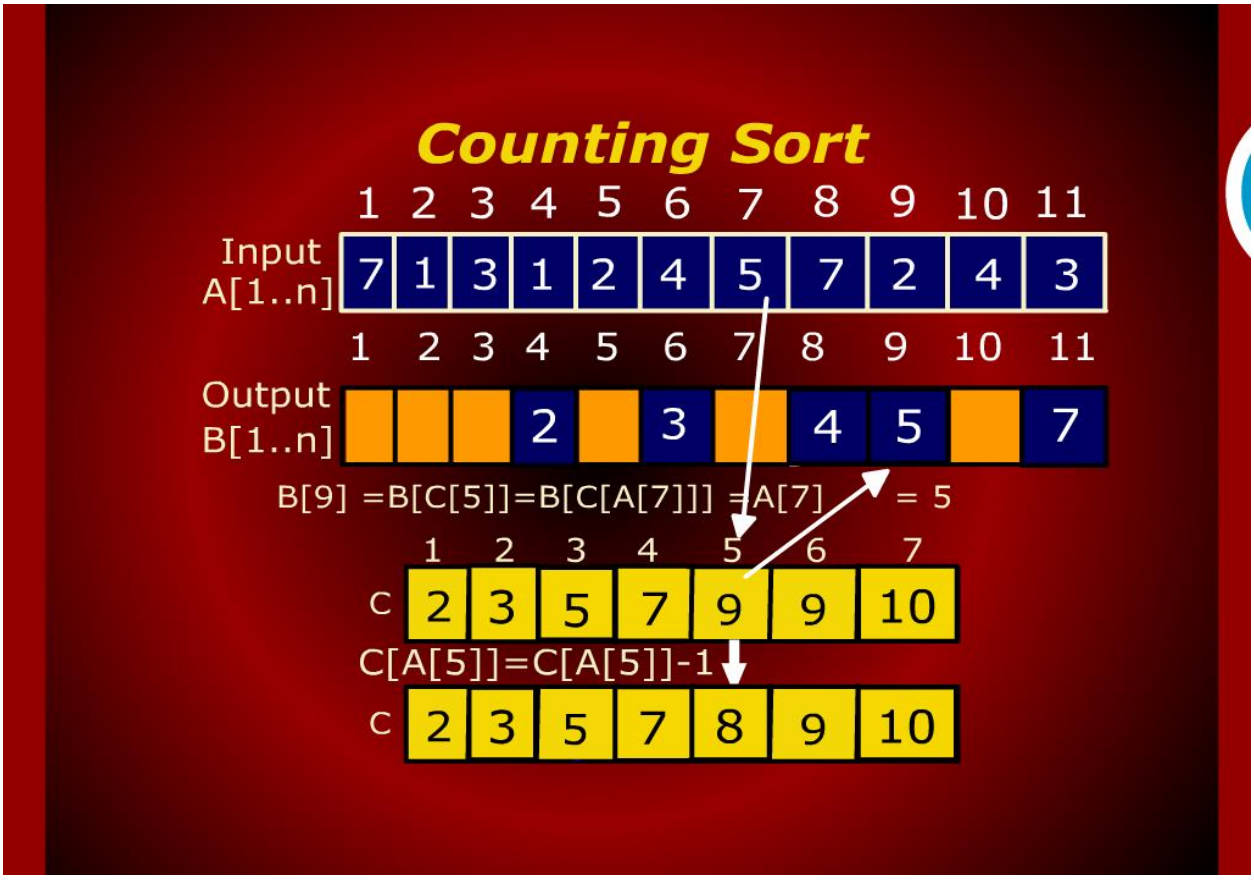

Counting Sort
1 2 3 4 5 6 7 8 9 10 11
Input A[1..n]
7 1 3 1 2 4 5 7 2 4 3
1 2 3 4 5 6 7 8 9 10 11
Output B[1..n]
3
B[6] =B[C[3]] =B[C[A[11]]] =A[11] = 3
1 2 3 4 5 6 7
C
2 4 6 8 9 9 11
C[A[11]]=C[A[11]]-1
C
2 4 5 8 9 9 11

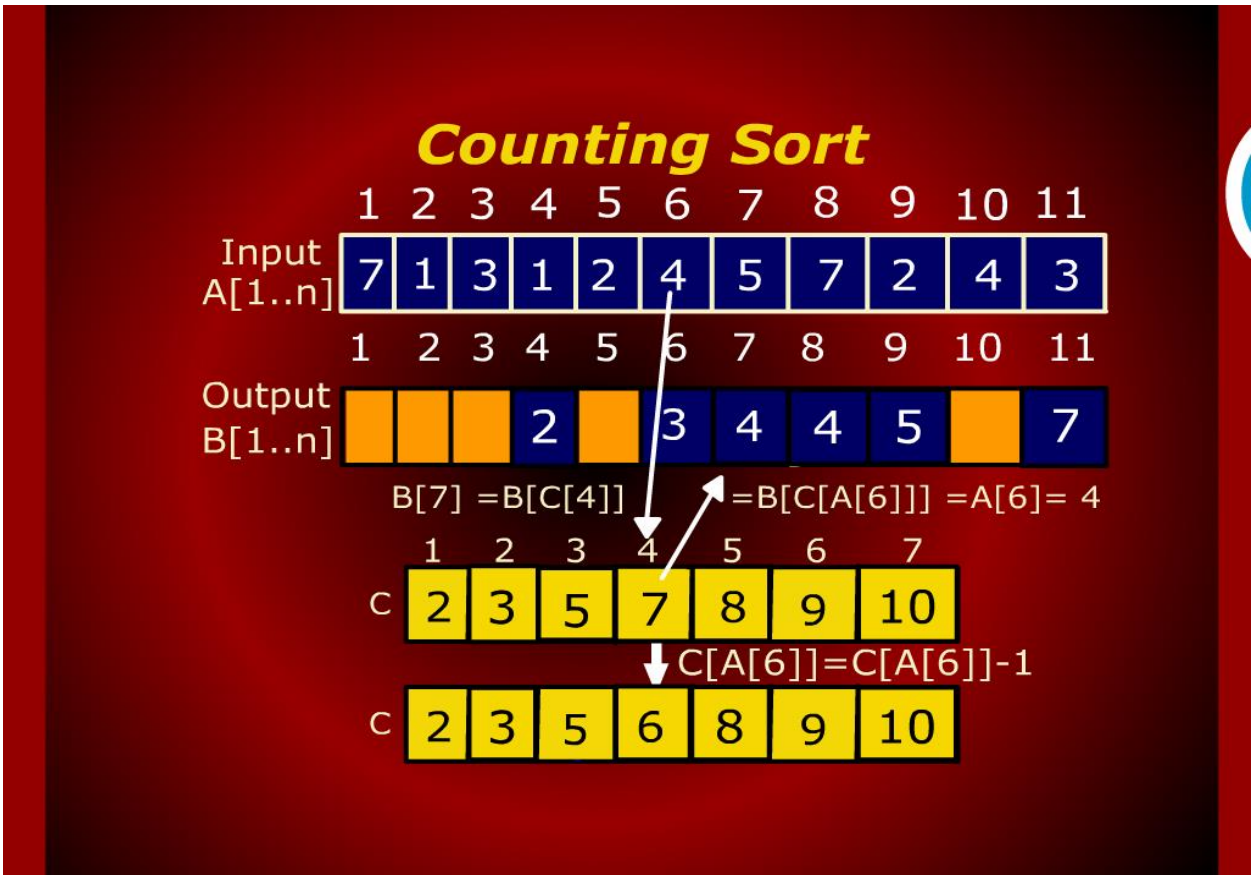

Counting Sort
1 2 3 4 5 6 7 8 9 10 11
Input
A[1..n]
7 1 3 1 2 4 5 7 2 4 3
1 2 3 4 5 6 7 8 9 10 11
Output
B[1..n]
3
B[6] =B[C[3]] =B[C[A[11]]] =A[11] = 3
1 2 3 4 5 6 7
C
2 4 6 8 9 9 11
C[A[11]]=C[A[11]]-1
C
2 4 5 8 9 9 11

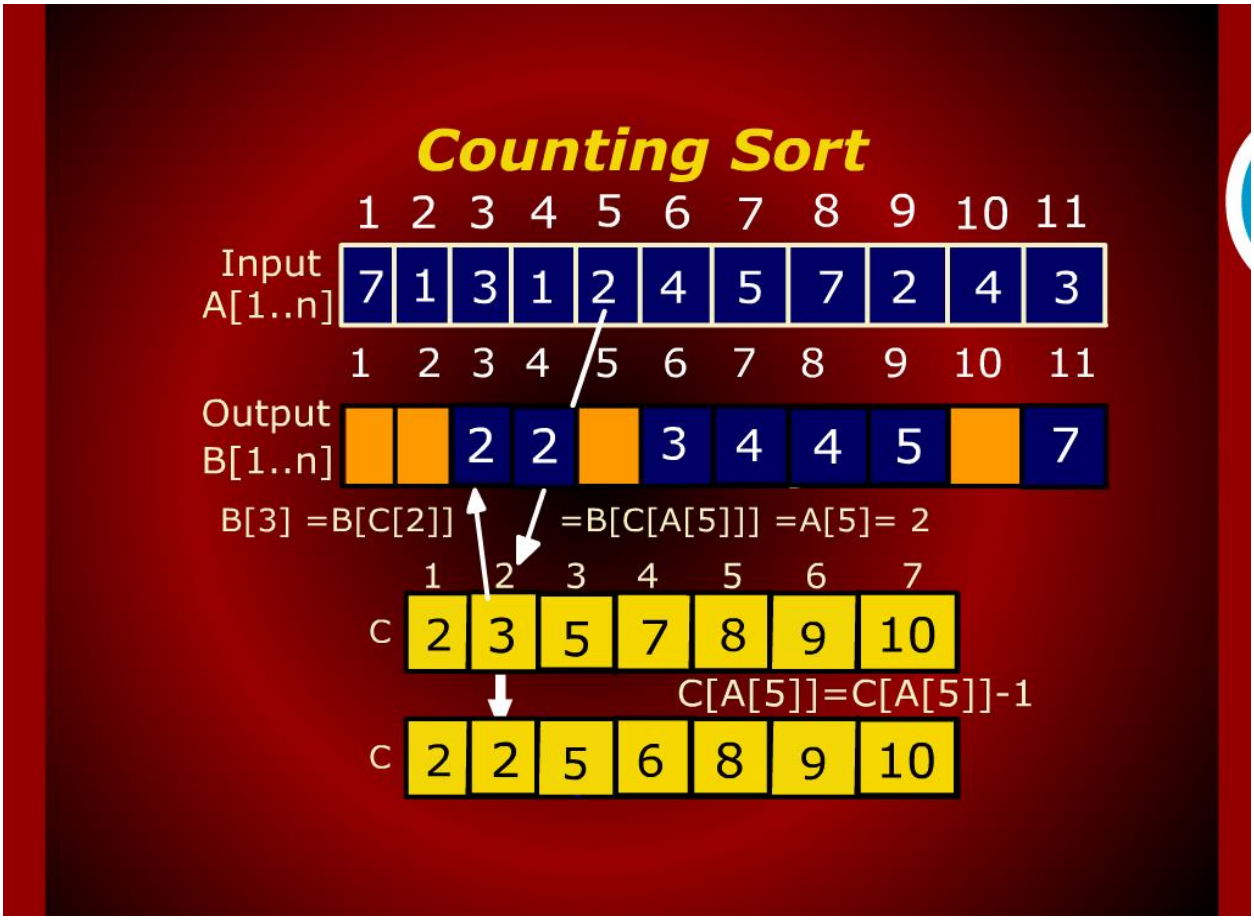

Counting Sort
1 2 3 4 5 6 7 8 9 10 11
Input
A[1..n]
7 1 3 1 2 4 5 7 2 4 3
1 2 3 4 5 6 7 8 9 10 11
Output
B[1..n]
3 4
B[8] =B[C[4]] =B[C[A[10]]] =A[10] = 4
1 2 3 4 5 6 7
C
2 4 5 8 9 9 11
C[A[10]]=C[A[10]]-1
C
2 4 5 7 9 9 11

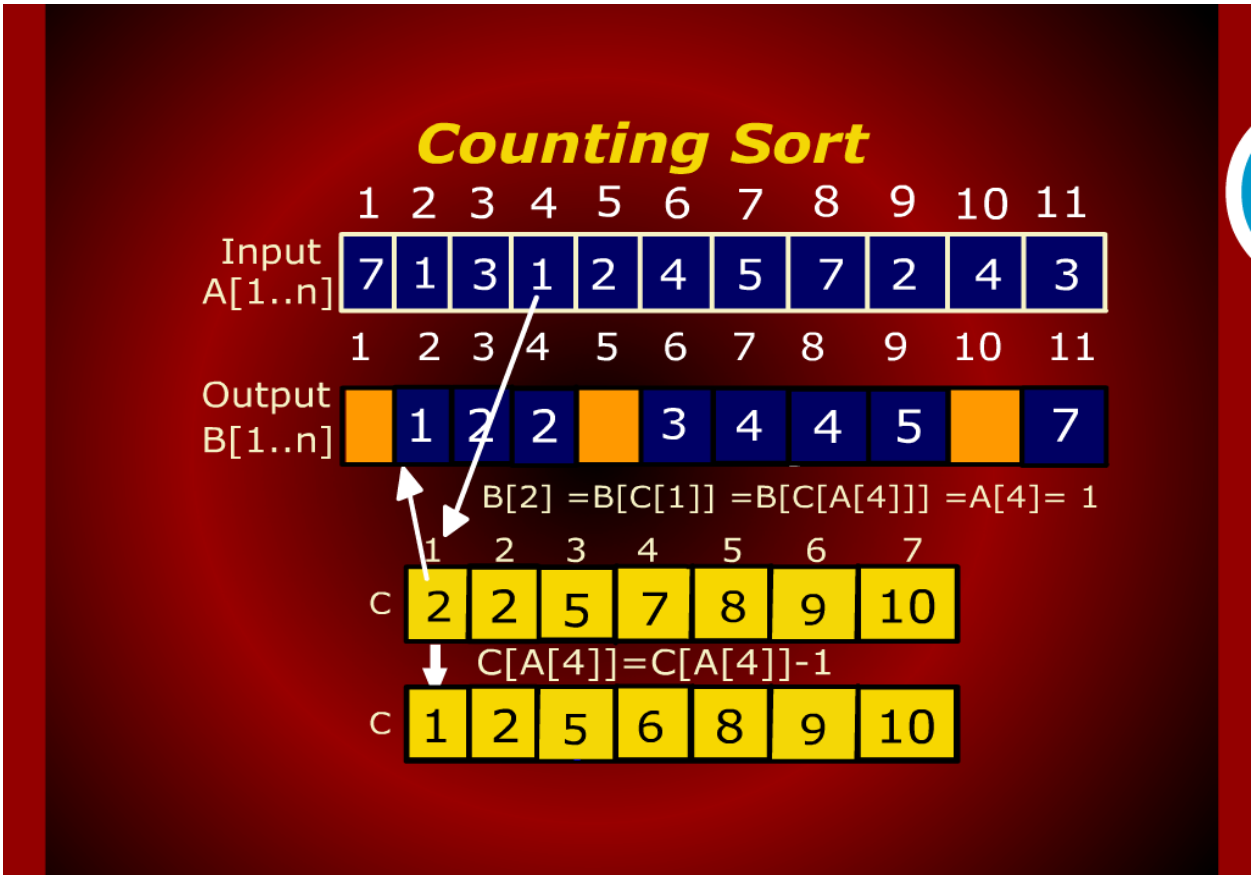

Counting Sort
1 2 3 4 5 6 7 8 9 10 11
Input A[1..n]
7 1 3 1 2 4 5 7 2 4 3
1 2 3 4 5 6 7 8 9 10 11
Output B[1..n]
2 3 4
B[4] =B[C[2]] =B[C[A[9]]] =A[9] = 2
1 2 3 4 5 6 7
C 2 4 5 7 9 9 11
C[A[9]]=C[A[9]]-1
C 2 3 5 7 9 9 11

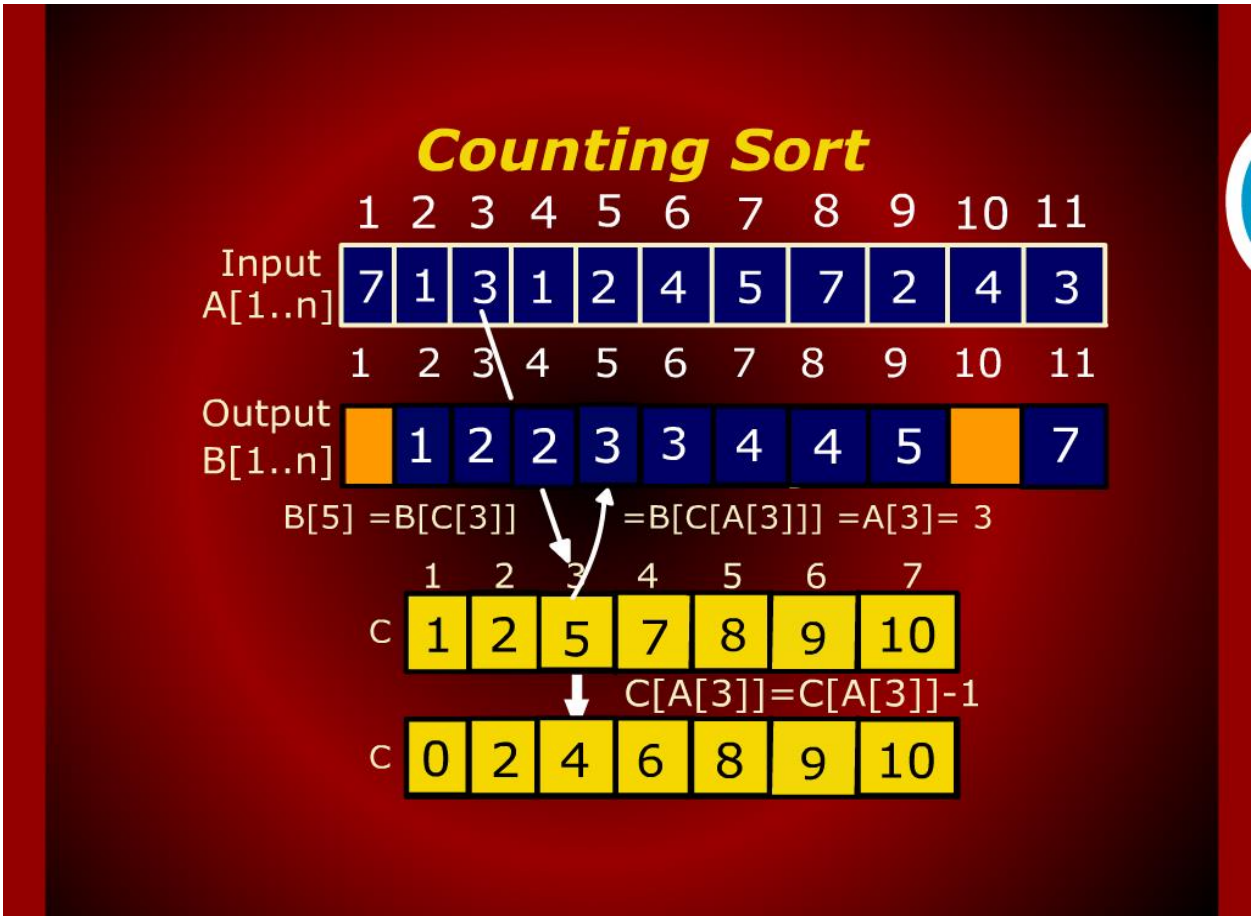

Counting Sort
1 2 3 4 5 6 7 8 9 10 11
Input
A[1..n]
7 1 3 1 2 4 5 7 2 4 3
1 2 3 4 5 6 7 8 9 10 11
Output
B[1..n]
2 3 4 7
B[11] =B[C[7]]=B[C[A[8]]] =A[8] =7
1 2 3 4 5 6 7
C
2 3 5 7 9 9 11
C[A[8]]=C[A[8]]-1
C
2 3 5 7 9 9 10

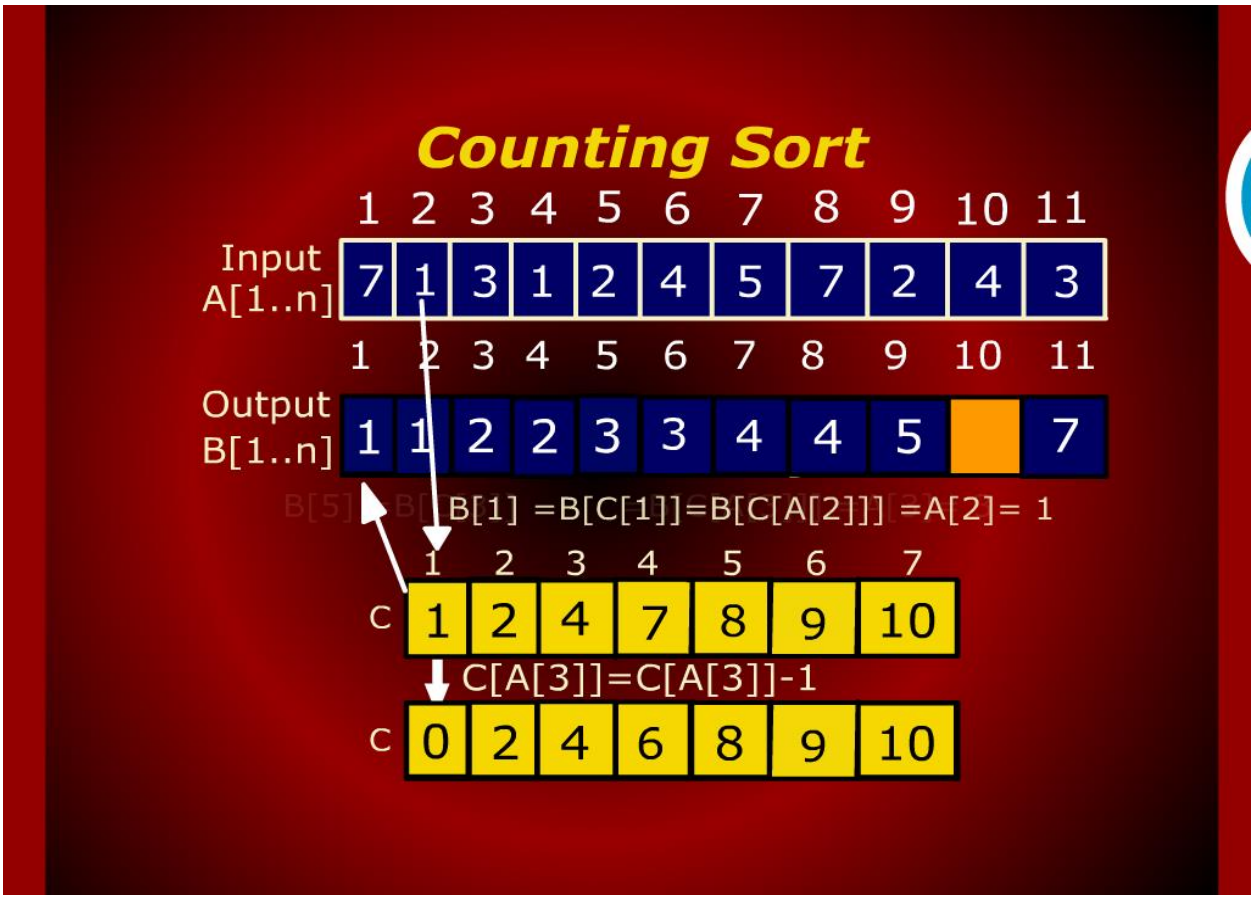

Counting Sort
1 2 3 4 5 6 7 8 9 10 11
Input A[1..n]
7 1 3 1 2 4 5 7 2 4 3
1 2 3 4 5 6 7 8 9 10 11
Output B[1..n]
2 3 4 5 7
B[9] =B[C[5]]=B[C[A[7]]] =A[7] = 5
1 2 3 4 5 6 7
C 2 3 5 7 9 9 10
C[A[5]]=C[A[5]]-1
C 2 3 5 7 8 9 10


Counting Sort
1 2 3 4 5 6 7 8 9 10 11
Input A[1..n]
7 1 3 1 2 4 5 7 2 4 3
1 2 3 4 5 6 7 8 9 10 11
Output B[1..n]
2 3 4 4 5 7
B[7] =B[C[4]] =B[C[A[6]]] =A[6]= 4
1 2 3 4 5 6 7
C 2 3 5 7 8 9 10
C[A[6]]=C[A[6]]-1
C 2 3 5 6 8 9 10

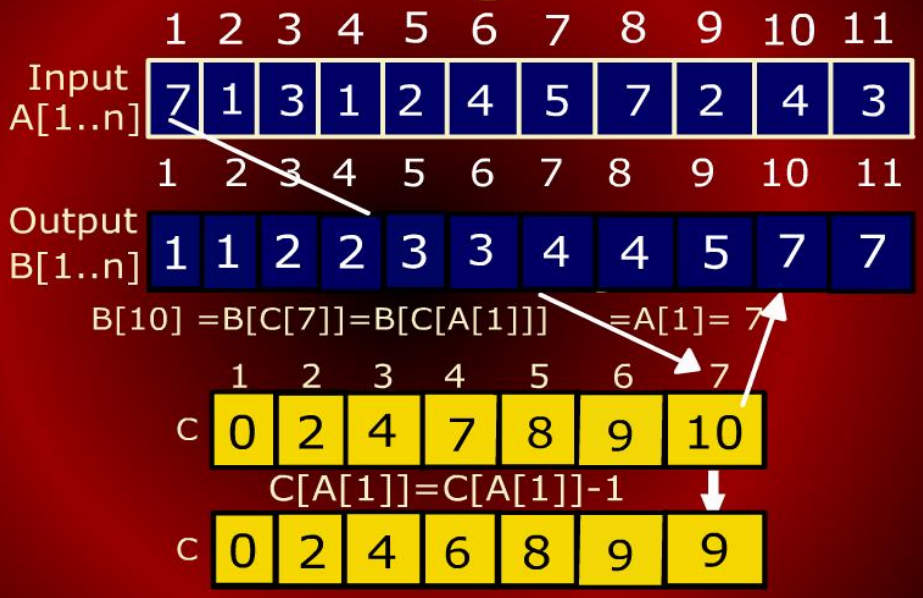

Counting Sort
1 2 3 4 5 6 7 8 9 10 11
Input
A[1..n]
7 1 3 1 2 4 5 7 2 4 3
1 2 3 4 5 6 7 8 9 10 11
Output
B[1..n]
2 2 3 4 4 5 7
B[3] =B[C[2]]
=B[C[A[5]]] =A[5]= 2
1 2 3 4 5 6 7
C
2 3 5 7 8 9 10
C[A[5]]=C[A[5]]-1
C
2 2 5 6 8 9 10


Counting Sort
1 2 3 4 5 6 7 8 9 10 11
Input A[1..n]
7 1 3 1 2 4 5 7 2 4 3
1 2 3 4 5 6 7 8 9 10 11
Output B[1..n]
1 2 2 3 4 4 5 7
B[2] =B[C[1]] =B[C[A[4]]] =A[4]= 1
1 2 3 4 5 6 7
C
2 2 5 7 8 9 10
C[A[4]]=C[A[4]]-1
C
1 2 5 6 8 9 10


Counting Sort
1 2 3 4 5 6 7 8 9 10 11
Input
A[1..n]
7 1 3 1 2 4 5 7 2 4 3
1 2 3 4 5 6 7 8 9 10 11
Output
B[1..n]
1 2 2 3 3 4 4 5 7
B[5] =B[C[3]] =B[C[A[3]]] =A[3]= 3
1 2 3 4 5 6 7
C
1 2 5 7 8 9 10
C[A[3]]=C[A[3]]-1
C
0 2 4 6 8 9 10


Counting Sort
1 2 3 4 5 6 7 8 9 10 11
Input A[1..n]
7 1 3 1 2 4 5 7 2 4 3
1 2 3 4 5 6 7 8 9 10 11
Output B[1..n]
1 1 2 2 3 3 4 4 5 7
B[1] =B[C[1]]=B[C[A[2]]] =A[2]= 1
1 2 3 4 5 6 7
C 1 2 4 7 8 9 10
C[A[3]]=C[A[3]]-1
C 0 2 4 6 8 9 10

Counting Sort



Counting Sort

```
COUNTING-SORT
( array A, array B,int k)
1     for i ← 1 to k
2     do C[i] ← 0          k times
3     for j ← 1 to length[A]
4     do C[A[j]] ← C[A[j]] + 1   n times
5     // C[i] now contains the number
      of elements = i
6     for i ← 2 to k
```

```
7    do C[i] ← C[i] + C[i - 1]   k times
8    // C[i] now contains the number
     of elements ≤ i
9    for j ← length[A] downto 1
10   do B[C[A[j]]] A[j]
11   C[A[j]] C[A[j]] - 1          n times
```

BUCKET SORT

- In this sorting algorithm, buckets are created to put elements into them.
- Then we apply some sorting algorithm (insertion sort) to sort the elements in each bucket
- Finally take out and join them to sorted array

Bucket Sort

- Assumption: input elements are uniformly distributed over [0, 1]

- n inputs dropped into n equal-sized subintervals of [0, 1].

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68

Consider total no. of elements are n=10. So we create 10 buckets.

Bucket array looks like

0
1
2
3
4
5
6
7
8
9

Now insert the values from the original array into the bucket array according to:

$$bucket\lfloor n * arr[i] \rfloor$$

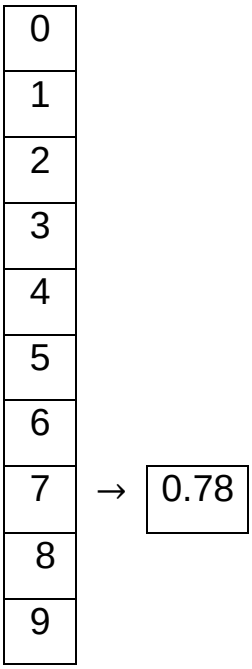
Where n = 10 then

If $arr[i] = 0.78$, then this value will be sorted on

$bucket[10 * 0.78]$ that would be $10 * 0.78 = 7.8$ take digit before decimal (NOTE do not round off)

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68

↑



0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68

↑

$now\ a[i] = 0.17$ will sort in the bucket of 1

0		
1	→	0.17
2		
3		
4		
5		
6		
7	→	0.78
8		
9		

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68

↑

$now\ a[i] = 0.39$ will sort in the bucket of 1

0		
1	→	0.17
2		
3	→	0.39
4		
5		
6		
7	→	0.78
8		
9		

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68

↑

$now\ a[i] = 0.39$ will sort in the bucket of 1

0		
1	→	0.17
2	→	0.26
3	→	0.39
4		
5		
6		
7	→	0.78
8		
9		

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68

↑

$now\ a[i] = 0.39$ will sort in the bucket of 1

0				
1	→	0.17		
2	→	0.26		
3	→	0.39		
4				
5				
6				
7	→	0.78	→	0.72
8				
9				

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68

↑

$now\ a[i] = 0.39$ will sort in the bucket of 1

0				
1	→	0.17		
2	→	0.26		
3	→	0.39		
4				
5				
6				
7	→	0.78	→	0.72
8				
9	→	0.94		

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68

↑

$now\ a[i] = 0.39$ will sort in the bucket of 1

0				
1	→	0.17		
2	→	0.26	→	0.21
3	→	0.39		
4				
5				
6				
7	→	0.78	→	0.72
8				
9	→	0.94		

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68

↑

$now\ a[i] = 0.39$ will sort in the bucket of 1

0				
1	→	0.17	→	0.12
2	→	0.26	→	0.21
3	→	0.39		
4				
5				
6				
7	→	0.78	→	0.72
8				
9	→	0.94		

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68

↑

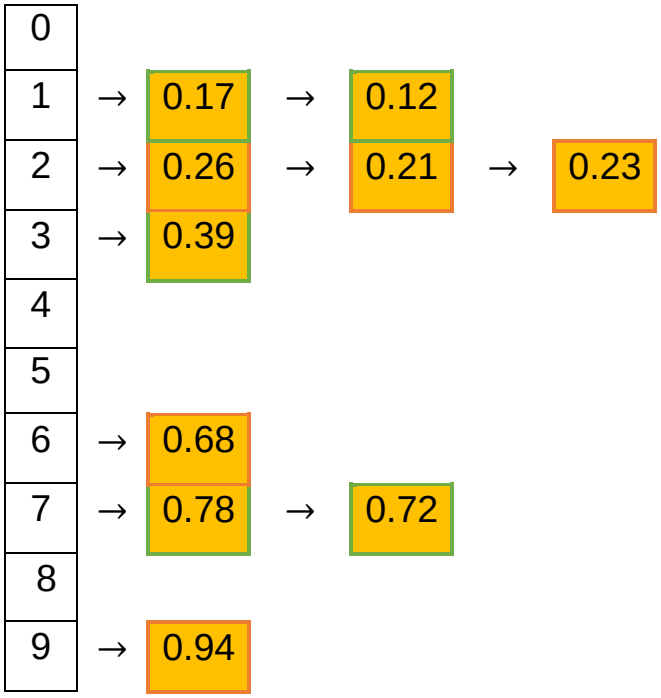
$now\ a[i] = 0.39$ will sort in the bucket of 1

0							
1	→	0.17	→	0.12			
2	→	0.26	→	0.21	→	0.23	
3	→	0.39					
4							
5							
6							
7	→	0.78	→	0.72			
8							
9	→	0.94					

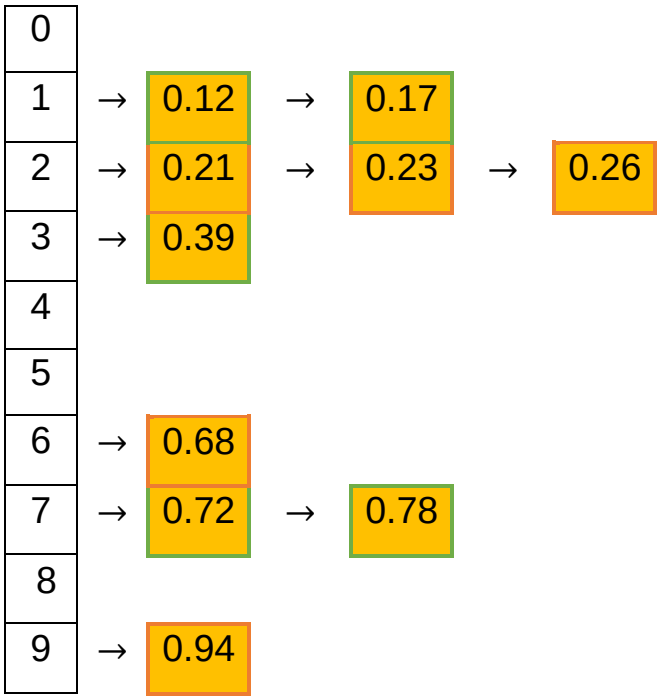
0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68

↑

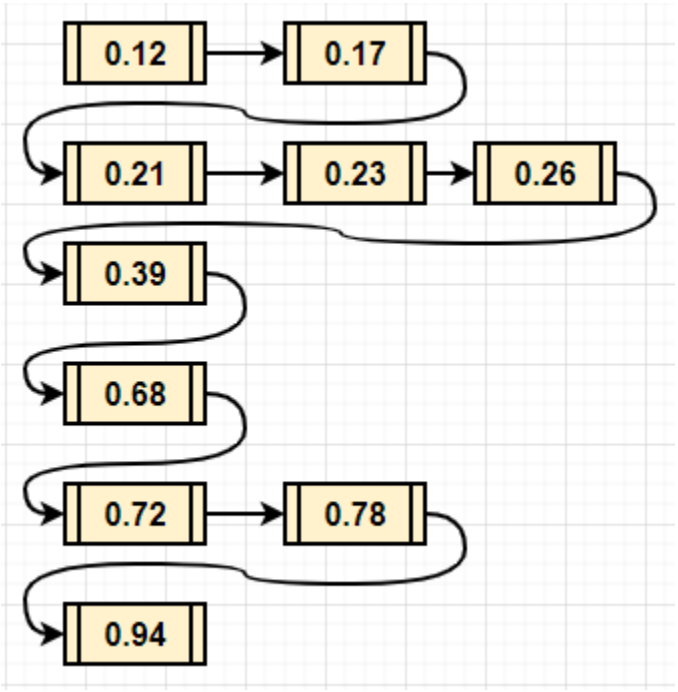
$now\ a[i] = 0.39$ will sort in the bucket of 1



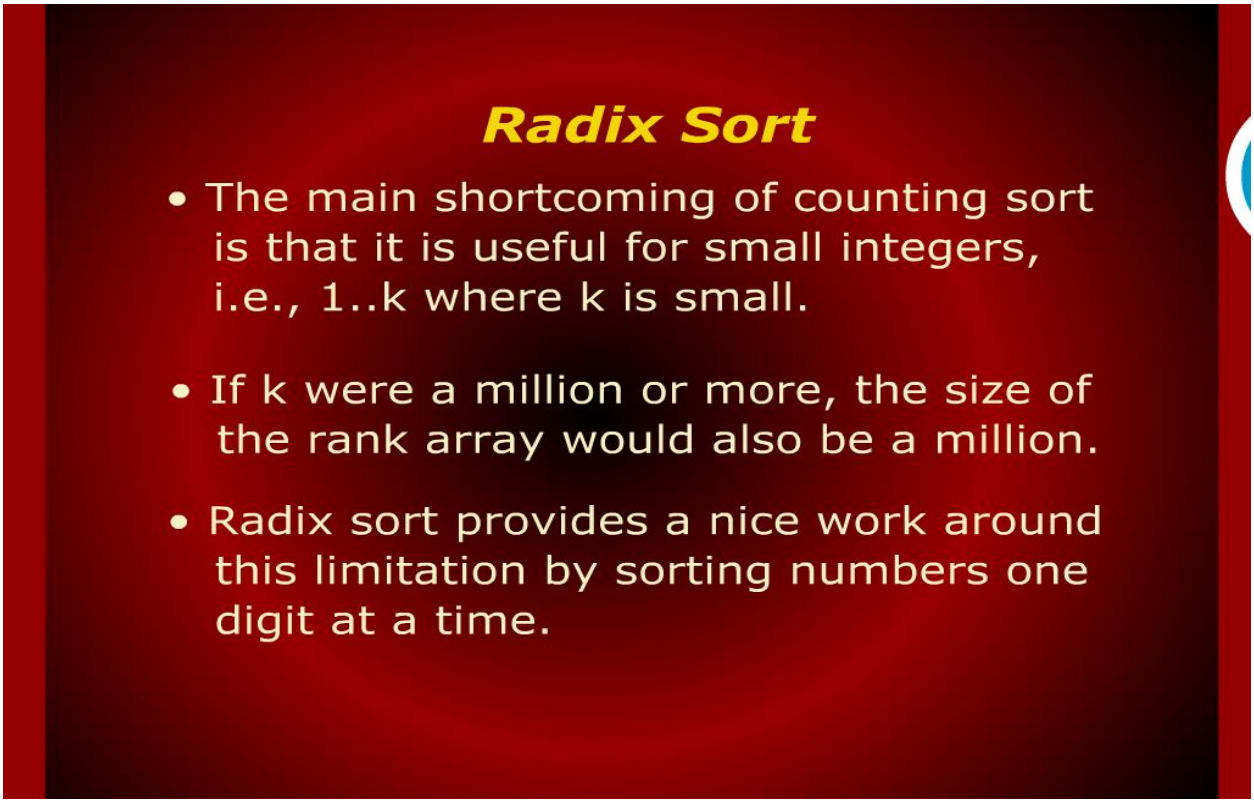
Now sort each bucket individually using insertion sort we get



Now, concatenate each bucket in the array, or you may use a link list for that.



Assignment 2: write the pseudocode of Bucket Sort.



Consider the unsorted input array as

170	45	75	90	802	24	2	66

Now apply the counting sort on the above input array but in a base 10^{th} manner.

1. First consider the one's place

17<u>0</u>	4<u>5</u>	7<u>5</u>	9<u>0</u>	80<u>2</u>	2<u>4</u>	<u>2</u>	6<u>6</u>

Counting sort applied

C=	0	1	2	3	4	5	6
	2	0	2	0	1	2	1

Now do $C[i] = c[i] + c[i-1]$ where $i = 2$ we get

C=	0	1	2	3	4	5	6
	2	2	4	4	5	7	8

Now create output array B of size $B[1..n]$

Starting reading original array 'A' from right to left

170	45	75	90	802	24	2	66
							↑

$$B\left[C\left[A[i]\right]\right] = A[i]$$

B=	1	2	3	4	5	6	7	8
								66

C=	0	1	2	3	4	5	6
	2	2	4	4	5	7	7

170	45	75	90	802	24	2	66
						↑	

$$B\left[C\left[A[i]\right]\right] = A[i]$$

B=	1	2	3	4	5	6	7	8
				2				66

C=	0	1	2	3	4	5	6
	2	2	3	4	5	7	7

170	45	75	90	802	24	2	66

↑

$$B\left[C\left[A[i]\right]\right] = A[i]$$

B=	1	2	3	4	5	6	7	8
				2	24			66

C=	0	1	2	3	4	5	6
	2	2	3	4	4	7	7

170	45	75	90	802	24	2	66

↑

$$B\left[C\left[A[i]\right]\right] = A[i]$$

B=	1	2	3	4	5	6	7	8
			802	2	24			66

C=	0	1	2	3	4	5	6
	2	2	2	4	4	7	7

170	45	75	90	802	24	2	66

↑

$$B\left[C\left[A[i]\right]\right] = A[i]$$

B=	1	2	3	4	5	6	7	8
		90	802	2	24			66

C=	0	1	2	3	4	5	6
	1	2	2	4	4	7	7

170	45	75	90	802	24	2	66

↑

$$B\left[C\left[A[i]\right]\right] = A[i]$$

B=	1	2	3	4	5	6	7	8
		90	802	2	24		75	66

C=	0	1	2	3	4	5	6
	1	2	2	4	4	6	7

170	45	75	90	802	24	2	66

↑

$$B\left[C\left[A[i]\right]\right] = A[i]$$

B=	1	2	3	4	5	6	7	8
		90	802	2	24	45	75	66

C=	0	1	2	3	4	5	6
	1	2	2	4	4	5	7

170	45	75	90	802	24	2	66

↑

$$B\left[C\left[A[i]\right]\right] = A[i]$$

B=	1	2	3	4	5	6	7	8
	170	90	802	2	24	45	75	66

C=	0	1	2	3	4	5	6
	0	2	2	4	4	5	7

Now consider 10th place; consider zero if the number. don't have 10th place

170	90	802	02	24	45	75	66

Again, apply the counting sort for the 10^{th} place, as shown in the grey area, we get

802	2	24	45	66	170	75	90

Now consider 100^{th} place.

802	002	024	045	066	170	075	090

Consider zero if the number, don't have 100^{th} place, again, apply counting sort as shown in the grey area, and we get

2	24	45	66	75	90	170	802

Now the array is sorted

Assignment 2: write the pseudocode of Radix Sort.