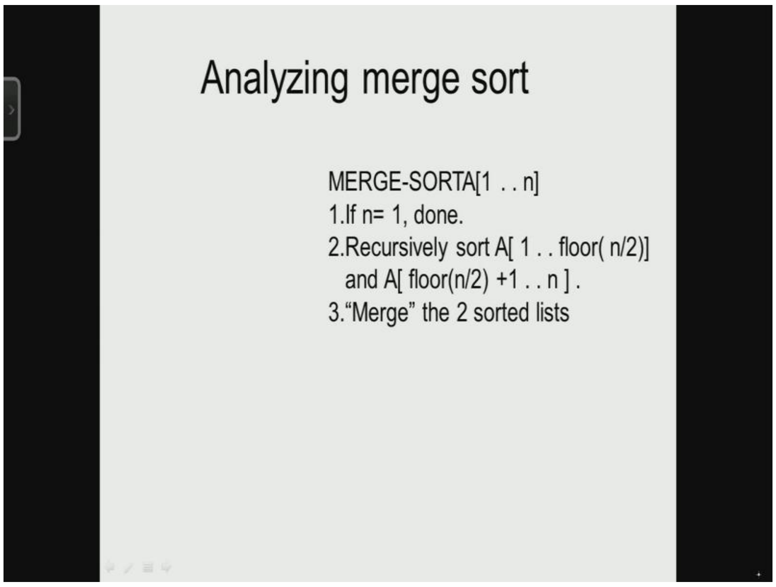
**Programming, Data Structures and Algorithms**
**Prof. N. S. Narayanaswamy**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Madras**

**Module - 06**
**Lecture – 33**
# Merge sort-Completion of running time analysis
# Counting sort-Sorting in linear time (in some cases): Stable sorting
# Radix sort-Correctness and analysis

So, we now study the analysis of merge sort and also learn a new algorithm for sorting; which under special cases is better than merge sort called radix sort.

(Refer Slide Time: 00:21)

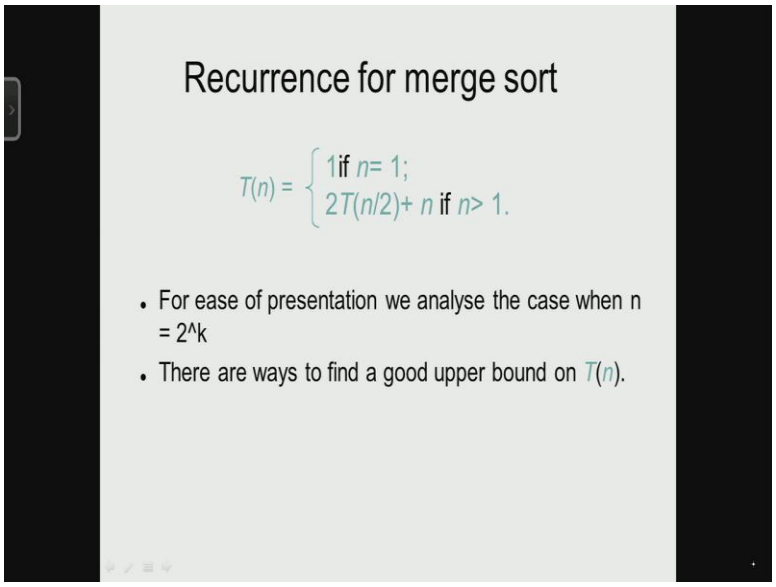


## Analyzing merge sort

```
MERGE-SORTA[1 . . n]
1.If n= 1, done.
2.Recursively sort A[ 1 . . floor( n/2)]
  and A[ floor(n/2) +1 . . n ] .
3.“Merge” the 2 sorted lists
```

Recall merge sort; it is an algorithm implemented on an array of elements. If the array size is just a single element, it is done; otherwise, we sort the array of elements in the indices 1 to n by 2 – the floor of n by 2. And then the second half of the array, whose indices are in the range floor of n by 2 plus 1 to n; and then we merge these sorted list. We did see that the effort involved merging two sorted lists is linear in the size of the data that is to be merged. In other words, if we merge an array of size l with an array of size m, then the total number of comparisons taken is l plus m.

(Refer Slide Time: 01:14)



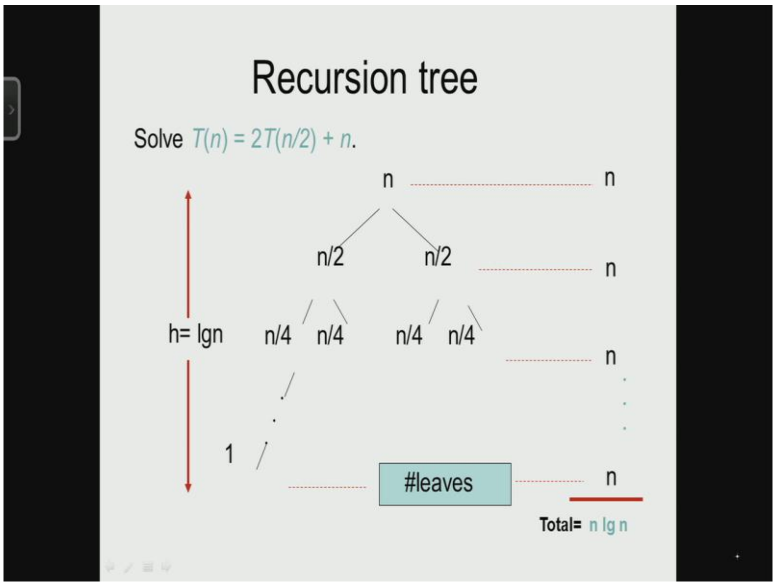
## Recurrence for merge sort

$$T(n) = \begin{cases} 1 \text{ if } n = 1; \\ 2T(n/2) + n \text{ if } n > 1. \end{cases}$$

- For ease of presentation we analyse the case when n = 2^k
- There are ways to find a good upper bound on $T(n)$.

The recurrence of merge sort is T of n is equal to 1 if n is 1 and it is 2 times T of n by 2 plus n if n is more than 1. The two times T of n by 2 comes from the two recursive sub problems that are generated from a given problem. And the additional term of n comes for the efforts spent in merging two arrays of size n by 2 – sorted arrays of size n by 2. We now complete the analysis of this recurrence. And for the ease of presentation, we analyze the case when n is a power of 2. There are many ways to find good upper bounds on t of n for n in general. However, we expose the way by which T of n is solved when n is a power of 2, because it is an extremely instructive presentation.

(Refer Slide Time: 02:12)



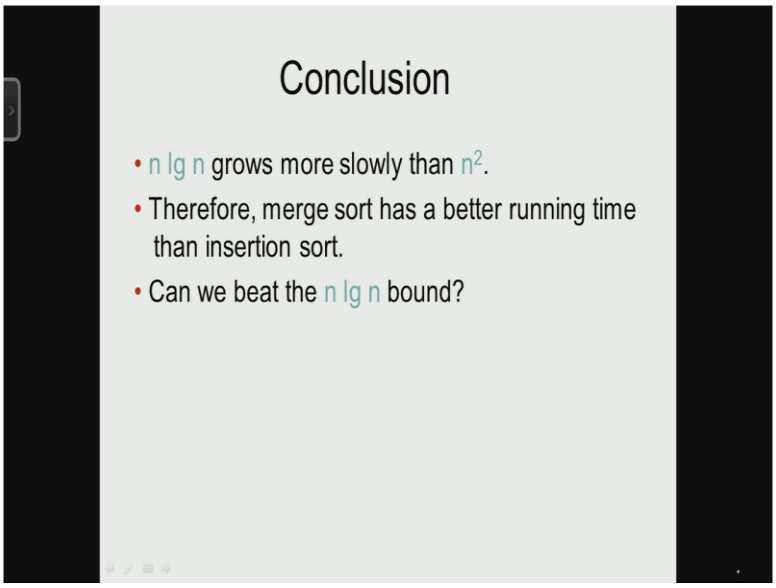
## Recursion tree

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

The recursion tree for the recurrence is T of n is equal 2 times T of n by 2 plus n. This is a recurrence. And the recursion tree is constructed as follows. At the root of the tree, is the function that we want to compute, which is T of n; its two children are T of n by 2 and T of n by 2. Now, the root is now replaced by the value n, which is added to the sum of T of n by 2 and T of n by 2. The tree is further expanded using the same root; where, n by 2 replaces the element T of n by 2; and its two children become the values T of n by 4 and T of n by 4. One branch of this whole recursion tree right up to the leaf – one can note that, the values are in the range n, n by 2, n by 4 all the way down to 1. In other words, as the depth of the tree increases from the root, the size of the sub problems being solved falls by a ratio of 2 – falls by a fraction of 2. It becomes n, n by 2, and then n by 4 and goes down all the way to 1. Therefore, the height of this tree is log n to the base 2.

The contribution to the whole recursion, which is essentially the summation of the values at all the nodes now; at the root, the contribution is n, because the value is just n. At the level 1 – depth 1, the contribution is again n, because there are two nodes of value n by 2. At the next level 2, there are two nodes, four nodes of value n by 4; and the contribution is a value n and so on all the way up to the leaves. And it is clear that, the number of leaves is exactly equal to n, because every leaf node contributes a value 1. And therefore, the number of leaf nodes is n. Therefore, the contribution from leaf level is also n. Consequently, the solution to this recurrence is n log n.

(Refer Slide Time: 04:50)

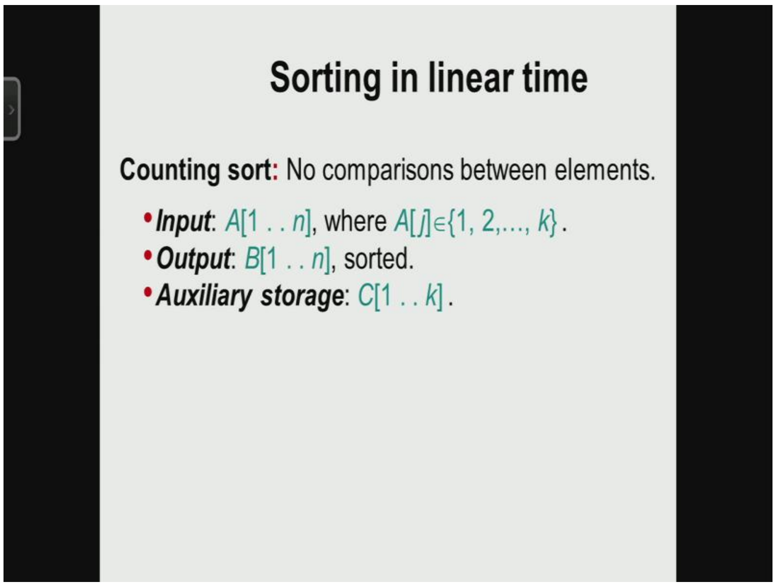


## Conclusion

- n lg n grows more slowly than $n^2$.
- Therefore, merge sort has a better running time than insertion sort.
- Can we beat the n lg n bound?

Now, we have placed in upper bound, the total number of comparisons made by merge sort; which is an algorithm, which uses the divide and concur approach. And clearly, n log n grows more slowly than n square. This is very easily visible to us; observe the n log n and the n square; the common term is n and log n and n; n is exponentially larger than log n. And therefore, n log n grows much more slowly than n square. Therefore, merge sort has a better asymptotic running time than the algorithm insertion sort. We go forward with the very natural question of can we beat the n log n bound?

(Refer Slide Time: 05:28)



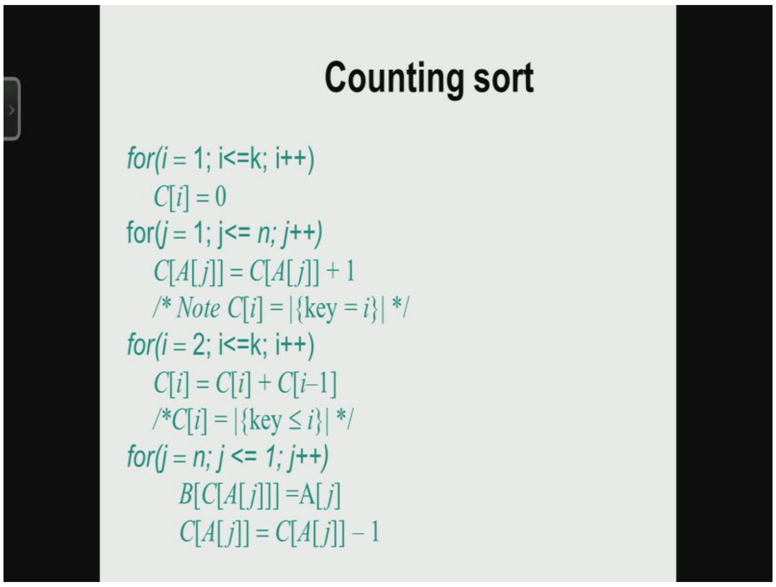
## Sorting in linear time

**Counting sort:** No comparisons between elements.

- ***Input***: $A[1 \,..\, n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
- ***Output***: $B[1 \,..\, n]$, sorted.
- ***Auxiliary storage***: $C[1 \,..\, k]$.

It is indeed possible to sort linear time when the data that we are sorting has some very special properties. In this case, we study an algorithm called counting sort; counting sort is a very special algorithm; it does not have any comparisons among the elements. The input array is an n element array, which contains the values to be sorted. In this case, we assume that, the values come from a range 1, 2 to k. And we make an assumption that, the values are all numbers. The output is presented in an array B, which is also an n element array and the output is sorted. The algorithm uses an auxiliary array called C, which has k elements. Recall that, k is a total number of distinct elements that once C is in the input array from the range 1 to k and C is an array of size k elements; in essence, every value, which is in the array A can be an index into the array C. And this is an idea that we use to obtain a linear time sorting algorithm in this special case.

(Refer Slide Time: 06:52)



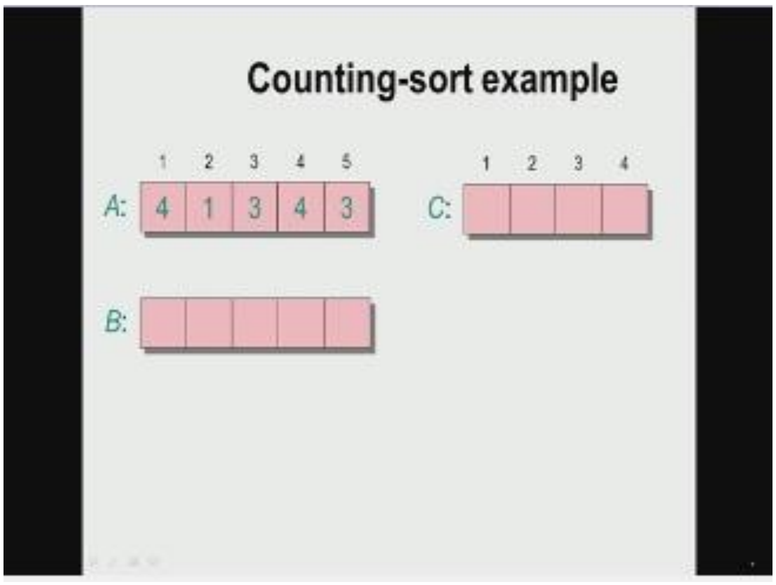
## Counting sort

```
for($i$ = 1; i<=k; i++)
    $C[i] = 0$
for($j$ = 1; j<= $n$; $j$++)
    $C[A[j]] = C[A[j]] + 1$
    /* Note $C[i] = |\{key = i\}|$ */
for($i$ = 2; i<=k; i++)
    $C[i] = C[i] + C[i-1]$
    /*$C[i] = |\{key \leq i\}|$ */
for($j$ = $n$; $j$ <= 1; $j$++)
        $B[C[A[j]]] = A[j]$
        $C[A[j]] = C[A[j]] - 1$
```

So, let us just look at counting sort. Counting sort basically has four loops. The first loop is the initialization loop, where the count array or the array C is initialized to take the value 0. In the next iteration, in the next for loop, we count the number of occurrences of each element of the array A by making one pass of the array A. Observe what the for loop does. The statement inside the for loop; say for example, the first element – C of A of j; if the first element is a 1, then the array index 1 in the array C is incremented by 1. So, every time we see a value in the array A, the corresponding index location – corresponding location in C is indexed by the value is incremented by 1. At end of the first for loop, every array location – in particular, the i-th array location counts the number of occurrences of the key i in array A.

In the third for loop, the consecutive elements of the array C are added to ensure that, the array element i keeps track of the number of elements of value less than or equal to i in the given array A. And finally, these count information are used to rearrange and obtain the sorted array B, which is as follows. The key at the element j – the key at the index j in array A is moved to its rank in array B. In other words, if the key at array index j has four occurrences in the array A, then C of A of j would be 4; and the value A of j would be kept in the array location B of 4. And then the number of occurrences of the value at A of j at the index j is reduced by 1.

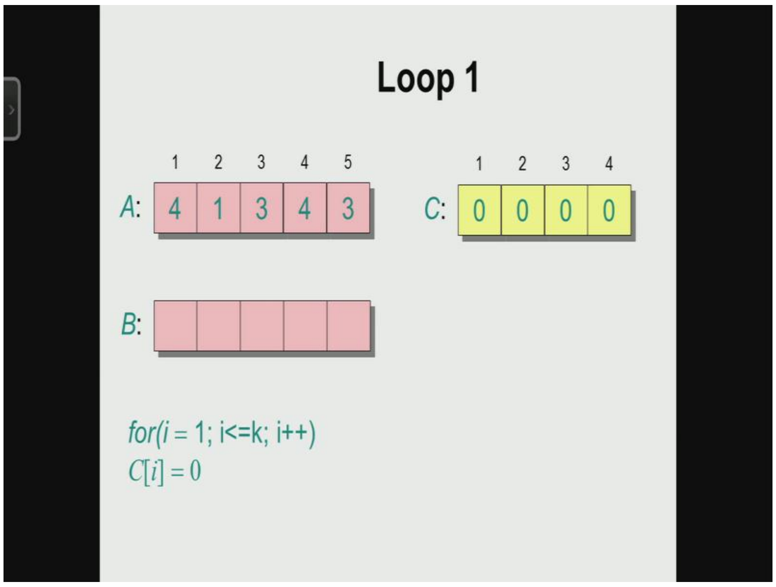
(Refer Slide Time: 09:42)



## Counting-sort example

Let us run one example of this algorithm; the array A has the values 4, 1, 3, 4, 3 in the five locations.

(Refer Slide Time: 09:53)

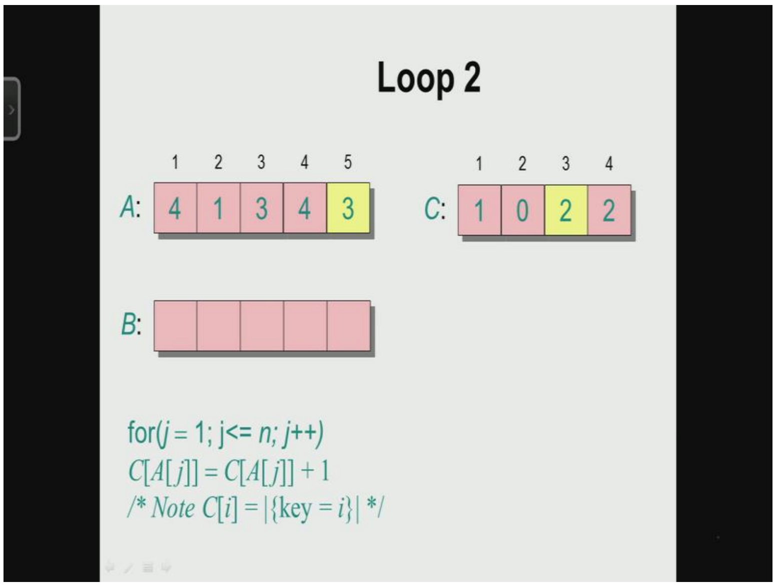


## Loop 1

```
for(i = 1; i<=k; i++)
C[i] = 0
```

And let us see how the initial array C looks like. It is initialized to 0. And since the distinct keys are in the range – 1 to 4 in the array A, the array size C has only four elements; array C has only 4 elements to keep track of the counts of the values 1, 2, 3 and 4 respectively.

(Refer Slide Time: 10:23)

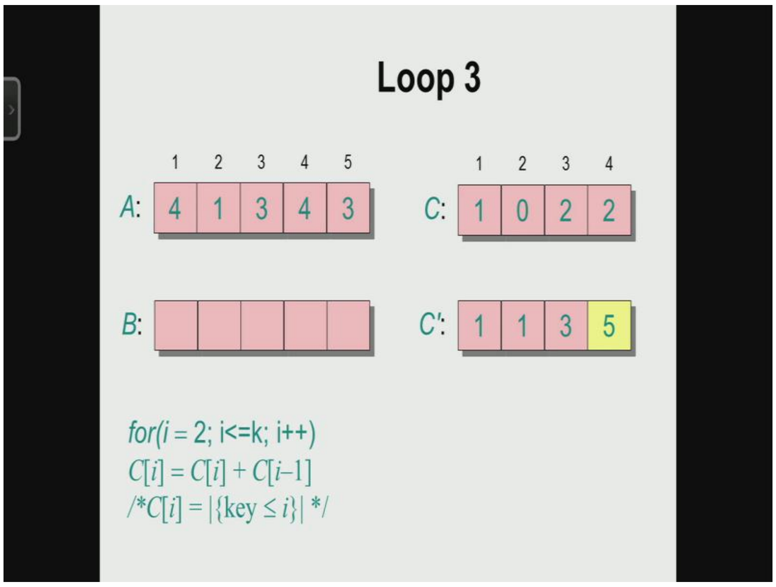


## Loop 2

```
for(j = 1; j<= n; j++)
    $C[A[j]] = C[A[j]] + 1$
/* Note $C[i] = |\{\text{key} = i\}|$ */
```

The first element in an array A is the value of 4 and the second loop increments C of 4. Then the value C of 1 is incremented by 1. Then the value C of 3 is incremented by 1. Then the value C of 4 is incremented by 1 and it becomes 2 now. And the value C of 3 is incremented by 1 now, which becomes 2. It is easy to observe the following invariant that, the sum of the array elements C is the total number of elements in the array A. In this case, it is 5.

(Refer Slide Time: 11:17)

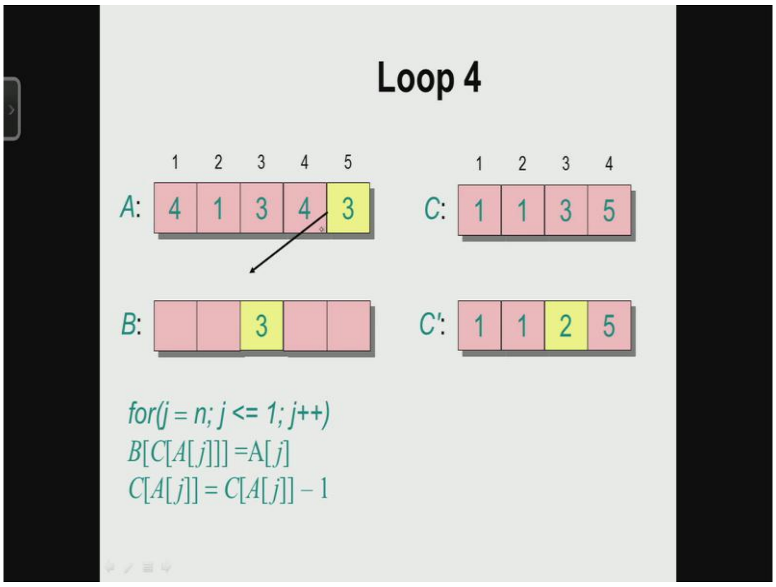


## Loop 3

```
for(i = 2; i<=k; i++)
    $C[i] = C[i] + C[i-1]$
/*$C[i] = |\{\text{key} \le i\}|$ */
```

In the next iteration, we modify and store C. And let us see how we do this. Observe that, we now keep track of the number of elements, whose value is less than or equal to 2 in the array A and in the location C of 2; that is obtained by adding C of 1 and C of 2 to get the value 1, which is a count of the number of elements, whose value is less than or equal to 2. In the next iteration, the number of elements is less than or equal to the value 3 is kept track off. And we can see that, the value should be 3 and let us see this. That is what it becomes. And then the number of elements of value less than or equal to 4 is in the five elements; all of them are of value less than or equal to 4. And C of 4 is now updated to the value 5.

(Refer Slide Time: 12:19)

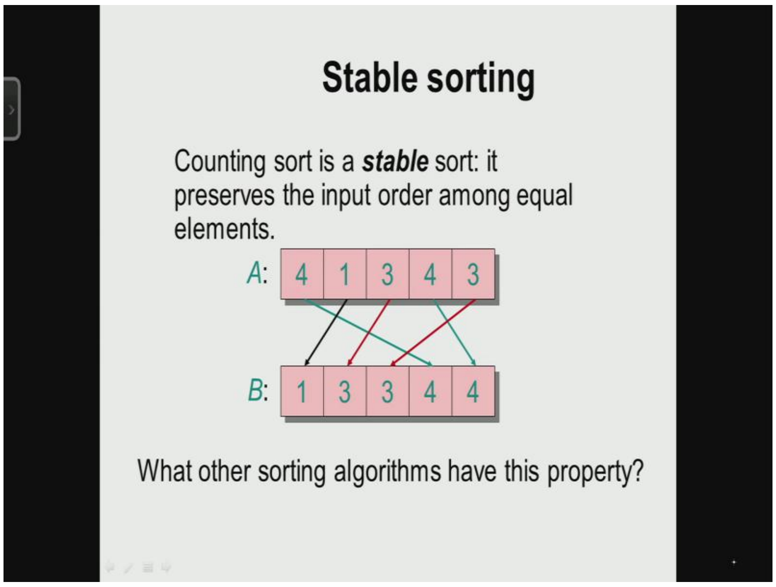


## Loop 4

$for(j = n; j <= 1; j{+}{+})$
$B[C[A[j]]] = A[j]$
$C[A[j]] = C[A[j]] - 1$

Now, comes the phase when the sorted elements are output. And this is a loop that runs in the reverse order. Let us look at the last element in the array A; it is a value 3. Now, there three occurrences of elements less than or equal to 3 in the array A, that is, the element 1, 3 and 3; now, the element 3 should definitely be in the third position in the sorted array. And therefore, 3 is now placed into the third position. And the number of relevant occurrences of 3 is now reduced by… The number of elements smaller than 3 is now reduced by 1. Then 4 is considered. The number of elements of value is less than or equal to 4 is 5. And therefore, this 4 should occur in the fifth location in the sorted array. In other words, 4 is the largest element and it must occur in the fifth location in the sorted array and then the value of C of 4 is reduced by 1. Next element considered as 3; and the number of elements less than or equal to 3 in the given array A, which has not been considered so far is 2. And therefore, three goes into the second location. And then naturally, 1 goes into the first location. And the number of elements less than or equal to 1 is reduced to 0. And then the last element is considered 4. And the number of elements less than or equal to 4 in the sorted array – the array index is given by 4. And 4 goes into the fourth location and the algorithm terminates when the loop comes to an end.

(Refer Slide Time: 14:19)



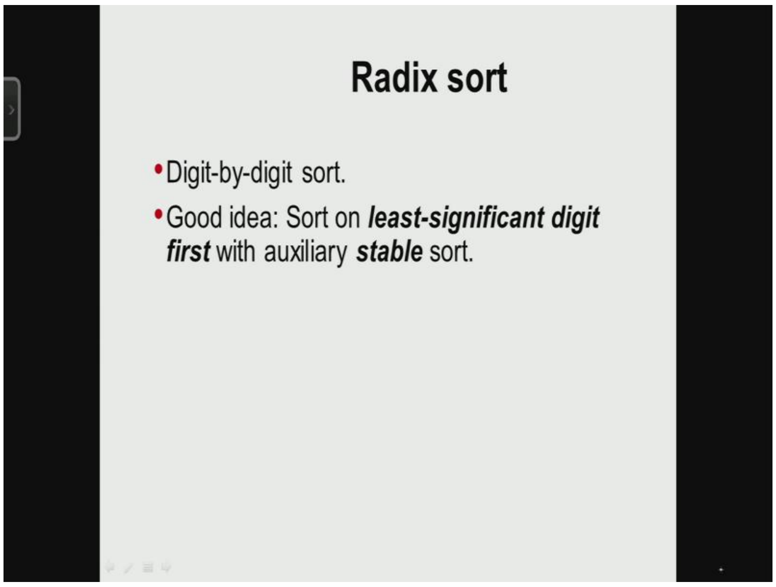
## Stable sorting

Counting sort is a ***stable*** sort: it preserves the input order among equal elements.

What other sorting algorithms have this property?

One of the most interesting properties of this algorithm is that, counting sort is a stable sorting algorithm. What is the meaning of a stable sorting algorithm? If two elements have the same value; if two array indices have the same value; then in the sorted array, the order in which they occur is preserved. For example, if you see the two occurrences of 4 in the array indices 1 and 4 respectively, observe that, the occurrence of 4 in the array location 1 in A goes to B of 4. And the occurrence of 4 in the fourth location goes to B of 5. Same is true with 3. Therefore, not only is the array sorted, but interestingly, the array also maintains a certain stability. In other words, it guarantees that, the input order among elements of the same value is respected in the sorted output. It is interesting to ask this question as to which other algorithm that we have seen have this property.

(Refer Slide Time: 16:12)

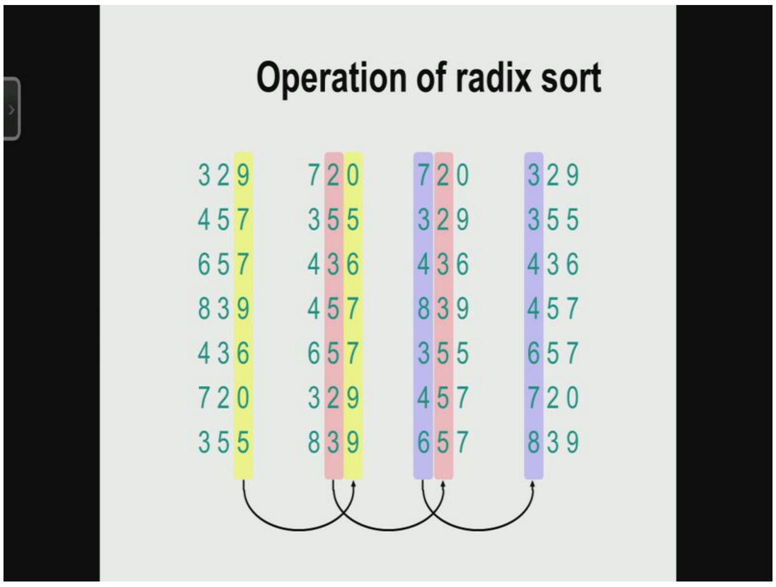


## Radix sort

- Digit-by-digit sort.
- Good idea: Sort on ***least-significant digit first*** with auxiliary ***stable*** sort.

Based on this algorithm, we present this approach to sort numbers, which is called the radix sort. The idea of the radix sort is to sort numbers digit-by-digit. And the idea is used to sort the numbers based on the first digit and use a stable sorting algorithm to sort the numbers based on the first digit. In other words, we will come up with an approach, where we just sort the first digit of all the numbers and we will use a stable sorting algorithm to do this and iterate over all the digits.

(Refer Slide Time: 16:17)



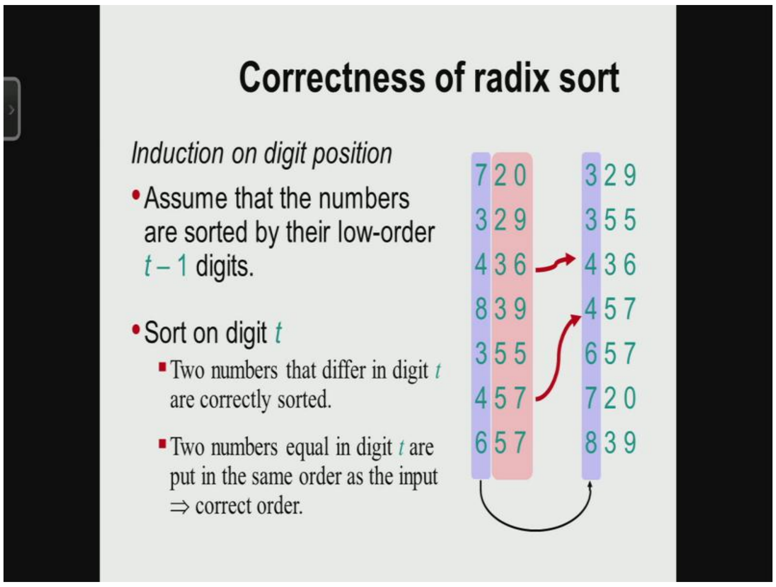
## Operation of radix sort

| | | | |
|---|---|---|---|
| 329 | 720 | 720 | 329 |
| 457 | 355 | 329 | 355 |
| 657 | 436 | 436 | 436 |
| 839 | 457 | 839 | 457 |
| 436 | 657 | 355 | 657 |
| 720 | 329 | 457 | 720 |
| 355 | 839 | 657 | 839 |

This is very nicely pictorially represented. So, let us consider the numbers, which are given to be sorted, which are the numbers 329, 457, 657, 839, 436, 720, and 355. Let us consider the least significant digit in each number and visualize this as the array that should be sorted by counting sort, which is the stable sorting algorithm that we have just seen. On sorting the array consisting of the least significant digits, the sorted array would look like this, which is natural. The least significant digits are when they are sorted, occur in the order 0, 5, 6, 7, 7, and 9, 9. And because counting sort was a stable sorting algorithm, observe that, the remaining two digits have also been moved to the appropriate location along with the least significant digit. For example, the number 720 has the smallest least significant digit among these. And however, it is a second largest number; but at the end of the first call to counting sort on the least significant digits, 720 occupies the first position and the whole number 720 goes to the first place and 0's at the correct position.

Now, the second least significant digit is picked up; and now, this array is sorted again using counting sort; and the remaining numbers in some sense are typed to the digit that is being sorted. And as a consequence of using a stable sorting algorithm, one can observe that, if you focus just on the numbers in the first two digits, they occur in the sorted order. And this is an inductive invariant that will be used to argue the correctness of the radix sort algorithm. Finally, we pick up the third digit and perform a stable sorting. And in this case, we use a counting sort, which we have just discussed. And now we can observe that, all the numbers are in sorted order. And observe that, the stability of counting sort is very crucial in the correctness of the algorithm. And when a certain array is being sorted, the remaining numbers are visualized as being typed to this particular number.

(Refer Slide Time: 19:20)



## Correctness of radix sort

*Induction on digit position*

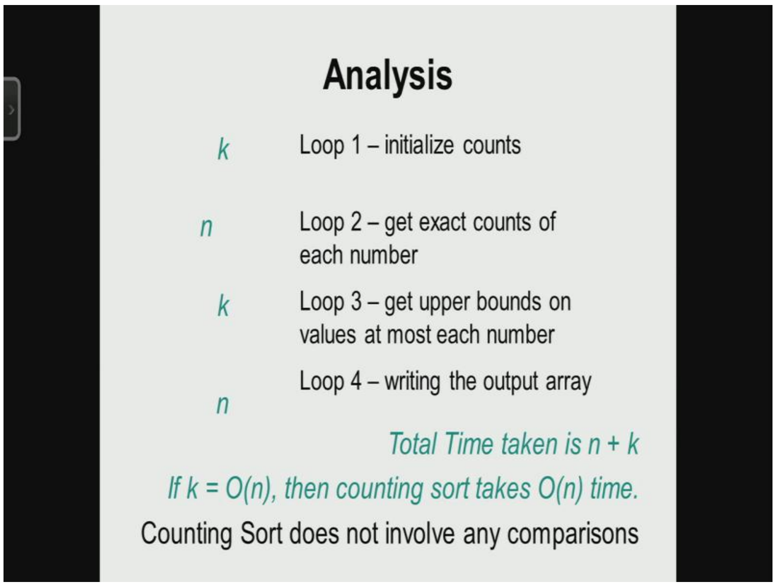
- Assume that the numbers are sorted by their low-order $t-1$ digits.
- Sort on digit $t$
  - Two numbers that differ in digit $t$ are correctly sorted.
  - Two numbers equal in digit $t$ are put in the same order as the input $\Rightarrow$ correct order.

720
329
436
839
355
457
657
329
355
436
457
657
720
839

So, the correctness of radix sort is argued by induction on the digit position. So, clearly, after the first call to counting sort; that is, after sorting the least significant digit, the number, whose lower order digits – the array of least significant digits are sorted. So, let us assume that, the numbers are sorted by their lower order t minus 1 digits after sorting the first t minus 1 applying counting sort on the first t minus 1 digits.

Now, when we sort on digit t… Now, when we sorted on digit t, the arrays is now sorted; it is a stable sorting algorithm. And let us look at two elements, which are in the opposite order. In other words, 720 and 329 – the first two elements observe that – they have exchanged their places; and 329 is smaller than 720; and 720 has gone to its correct location in the sorted array. Therefore, the numbers that digit that differ in digit t are correctly sorted. And two numbers, which have the same digit – two indices, which have the same value at digit t are also correctly sorted, because we are using a stable sorting algorithm. Therefore, we have the correct order among the t digits. And this completes the inductive argument.

(Refer Slide Time: 21:30)



## Analysis

| | |
|---|---|
| *k* | Loop 1 – initialize counts |
| *n* | Loop 2 – get exact counts of each number |
| *k* | Loop 3 – get upper bounds on values at most each number |
| *n* | Loop 4 – writing the output array |

*Total Time taken is n + k*

*If k = O(n), then counting sort takes O(n) time.*

Counting Sort does not involve any comparisons

Let us complete the analysis of counting sort. The loop 1, which was initializing the count array performed k operations, because every… there are k indices and each of them was initialized to 0. Loop 2 performs a computation by looking at every element of the input array A exactly once and then increasing an appropriate index in the count array C by 1. So, this takes n time, that is, it takes a constant number of arithmetic operations per stack. And loop 3 then gets upper bounds on the values at most on the number of terms, whose values are at most each index in C. And this again takes k time. And loop 4, which writes out the output array B also takes n time. And therefore, the total time taken is order of n plus k. In this case, it is 2n plus 2k arithmetic operations. Now, if the key set – the smallest and the largest value differ by a linear amount by at most n; then counting sort takes order of n time. It is also clear that, counting sort does not perform any arithmetic, does not perform any comparison operations; and it is completely an arithmetic operation based sorting algorithm and it takes linear time in some special cases.