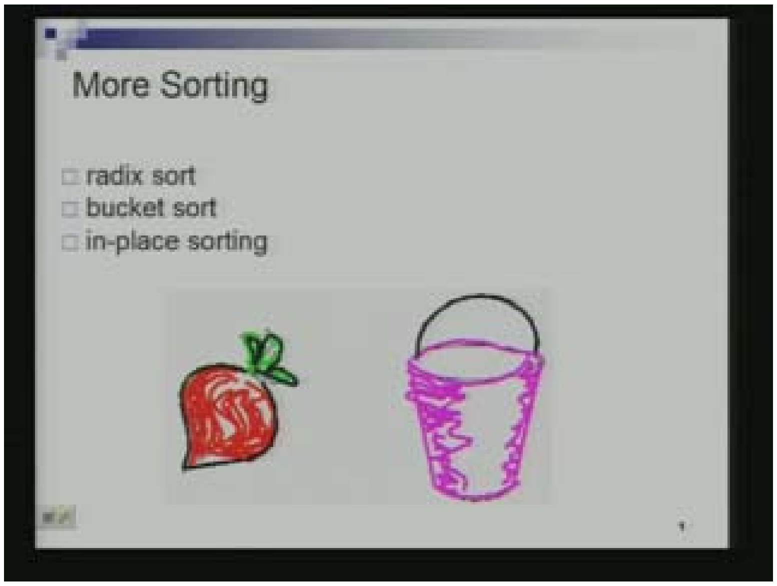
Data Structures and Algorithms

Dr. Naveen Garg
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 23 #
More Sorting

Today we are going to continue our discussion on sorting. We will begin with radix sort. Then we are going to look at bucket sort. In place sorting is something that we have seen before which are the examples that we know of in place sorting. Heap sort, quick sort, insertion sort, selection sort, bubble sort. Bubble sort we have not done. We are going to see which is in place and which is not in place, not all of these are in place what you just said. We are going to see and then we are going to finally look at how fast we can sort. Can we do better than what we have been discussing so far. What is a radix sort?

(Refer Slide Time: 1.21)

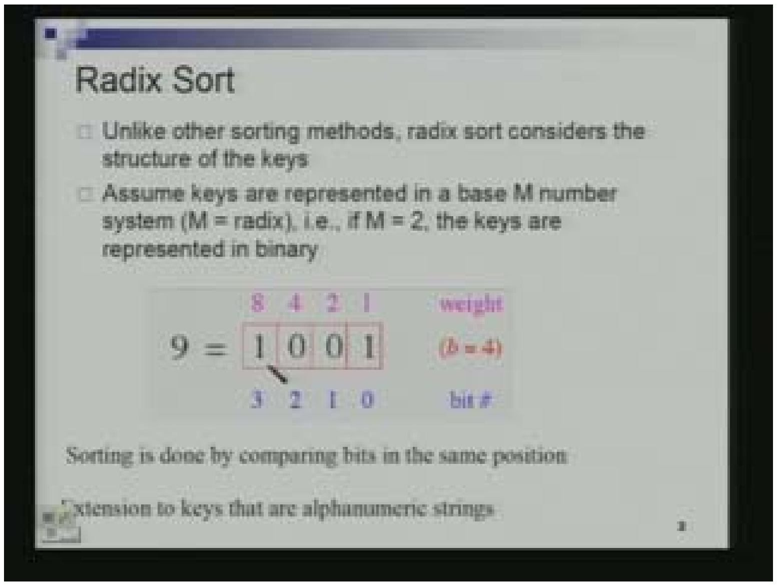


More Sorting

- radix sort
- bucket sort
- in-place sorting

See in radix sort we are going to look at the keys that we are going to sort. So recall that in all other sorting algorithm, we are only comparing the keys. We are not looking at what the actual structure of the keys is. It doesn't really matter if the keys or people provided you have a way of comparing two persons. If you have a comparator function which is given two persons who can say one is less than the other then you can also sort people. But today or in radix sort, we are going to look at the key itself that we are trying to sort, the collection of keys. We are going to assume that the keys are represented in some base M number system and M is called the radix. If M equals 2 then the keys are essentially in binary, base two.

(Refer Slide Time: 02.58)



Radix Sort

- Unlike other sorting methods, radix sort considers the structure of the keys
- Assume keys are represented in a base M number system (M = radix), i.e., if M = 2, the keys are represented in binary

$$\begin{array}{ccccccc} & 8 & 4 & 2 & 1 & & \text{weight} \\ 9 = & \boxed{1} & \boxed{0} & \boxed{0} & \boxed{1} & & (b = 4) \\ & 3 & 2 & 1 & 0 & & \text{bit \#} \end{array}$$

Sorting is done by comparing bits in the same position

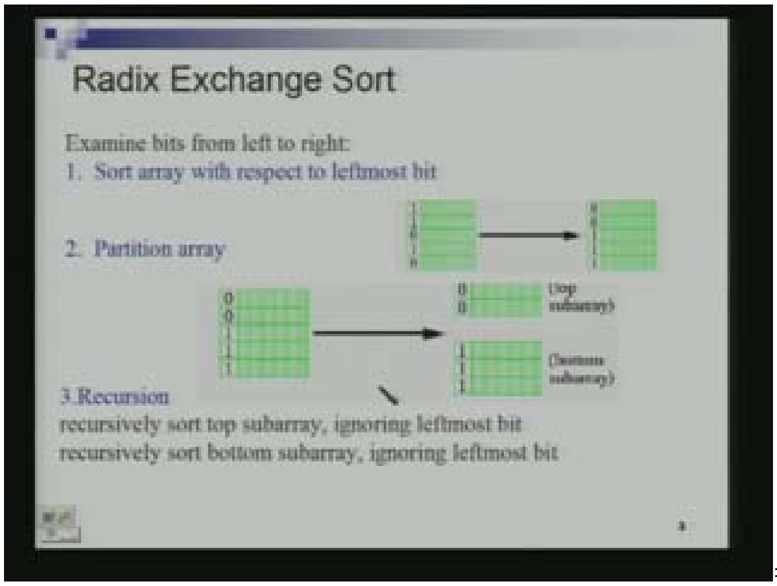
Extension to keys that are alphanumeric strings

This could be an example 9 is 1 0 0 1 in binary, in base two. We are going to use this representation of 9 to do the sorting first. I can represent 9 in base form or some other base also, base three let us say. Then M would become three and I can still use that to do the sorting and we will see how. And in radix sorting, the sorting is done by comparing bits in the same position. If instead of comparing numbers, instead of comparing 9 with 11, we are not going to compare 9. We are just going to compare the bits in 9 and 11. Which bits? Let us say look at the bit at position three and we are going to compare them. We will see all of that in a second and this idea can be extended when the keys are let us say alphanumeric strings also. Not just binary numbers, not just numbers like this but alphanumeric strings so name of people or some such thing. You can also use a same idea to do sorting on that and we will see how.

I am going to talk of two variants on radix sort. One is called the radix exchange sort so what we are going to do in radix exchange sort is we are going to examine the bit. I am going to assume now that the keys we are trying to sort are some numbers represented in binary first and am going to examine the bits from left to right. Let us assume also that each of the numbers that are given to us have a fixed representation. They are expressed in the same number of bits. That can always be done if the largest number in your collection is let us say some N then you need basically log N bits to represent that largest number. So with log N bits, you can express all the other numbers in log N bits and that would be the number of bits that you would use to represent all the numbers in the collection you have. We are going to sort the array with respect to the leftmost bit first.

Suppose the numbers are sitting in this array and the left most bit of this numbers. So it really doesn’t matter what these other bits are. I am just looking at the left most bit of each of these numbers. This is the first number, this row the second number is in this row. So there are 5 numbers, 5 rows. I am going to look at the left most bits and I am going to sort the numbers according to this bit which means these are two zeros, so they come first. So this number goes at this position, this number goes at this position and the ones come later. This goes here, this would go next and this would be the last (Refer Slide Time: 05:54). This is what I would do. Sort according to the left most bits and since the bit can have only two different values, it is easy. The zeros comes before the ones. Clear to everyone? Now we partition this array, so this is not yet sorted, this set of numbers is not sorted. I will now divide it into two parts. This is the top sub array and bottom sub array. I am going to sort this top sub array independently of the bottom sub array. What do I mean by sort the top sub array? Just look at these numbers, forget this bit because this bit is same for all of these numbers. It will not make any difference in the value of the number, I will just forget this bit.

(Refer Slide Time: 07:54)



Radix Exchange Sort

Examine bits from left to right:

1. Sort array with respect to leftmost bit

2. Partition array

(top subarray)
(bottom subarray)

3.Recursion

recursively sort top subarray, ignoring leftmost bit

recursively sort bottom subarray, ignoring leftmost bit

#

Similarly when I am sorting this, I will forget this bit. What does it mean to forget this bit, when I am sorting these numbers? I am saying, I would take the number or remove and subtract some number from that. What is that number? If these where k bits number, I am subtracting 2 to the k from that number. If I subtract 2 to the k from each of the numbers and then sort them, then it is same as the sorting the original collection of numbers, it doesn’t makes a difference. I can sort this bottom sub array independently of the top sub array and then I just put them together and I would get a sorted sequence. It is a divide and conquer algorithm once again.

There is a divide step in which zeros come before the ones. There is a conquer step and the combined step is triviled here so the conquer step, this is the recursion. We will recursively sort the top sub array ignoring the leftmost bit. We will recursively sort the bottom sub array, ignoring the left most bit once again. How are we going to sort these? Using the same idea, we are going to partition this using the second left most bit, this left most bit and so on. How much time does it take to sort these n-bit numbers? I claim it takes order b n time, if I have n numbers and b bits. Why is it?

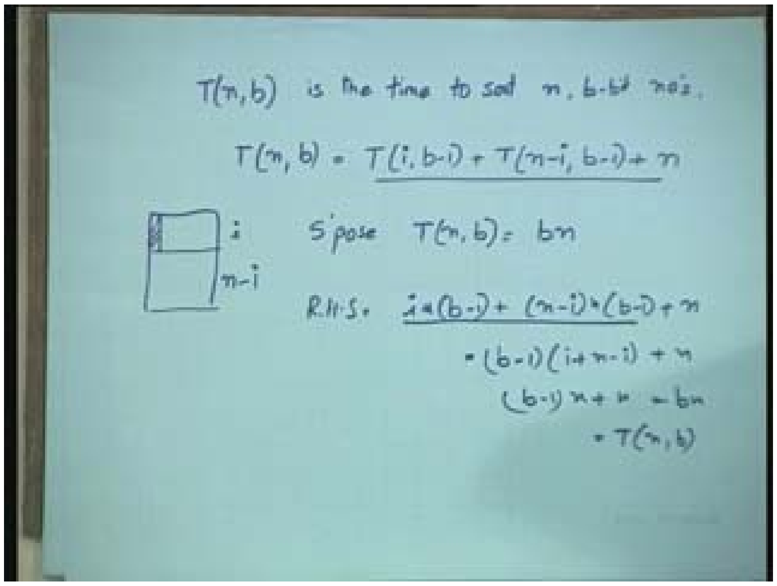
Why can you do it in so much time? Pardon. No of bits is log n. Is the number of bits log n? Log of the largest number. The largest number need not be n. It could be much larger than n, it could be much smaller than n. [student: the number of keys can be at most the largest, n is only the number] n is the number of numbers and not the largest number. [Student: exactly sir the number of keys n is at most the largest number.] Number of keys is at most largest number, this statement is not true. You could have duplicates and also the largest number can be much larger than the number of keys [student: 1 to 20,000] okay great.

We will continue this discussion later. So I claim to sort n b-bit numbers you will require order b n time and why is this? Let us try and understand this. Can I write a recurrence for this? Can someone write a recurrence for this? Let's write a recurrence. Let me stretch and write a recurrence so T (n, b). Why comma b? Number of bits. So T (n, b) is the time to sort n b-bit numbers. Let's say this is T (n, b). What is this equal to? We are going to partition this n numbers into two pieces. This was the top sub array, let's say this has i numbers in it and the remaining would be n minus i. Now how much time does it take to sort these i numbers? Because we are going to do it recursively. So it should be T (i, b-1) that's important the b minus 1.

Why is it important? Because we are going to ignore the left most bit because the left most bit is going to be the same for all of these numbers. The time required to sort the other n minus i numbers is n minus i, b minus 1 plus something more. How much time? Because I had time to partition this numbers, the numbers in which the left most was 0, come before the numbers in which the leftmost bit was a 1. They had to be rearranged for each and that can be done in order n time. This is the kind of recurrence we would get.

Now what is the solution for this recurrence? Anyone? What is the solution to this recurrence? b n. No, it's perfectly right if you are saying b n. Let's substitute, so this is the third method we had talked about. Solving recurrences by guessing a solution and substituting that solution on actually verifying weather its true or not. Let's see what it should be? Let's not worry too much about it. Let's see, suppose T (n, b) was equal to b of n. May be I am wrong. Let's see whether I am right. What is my right hand side then? The right hand side equals i because n is i times (b minus 1) plus (n minus i) times (b minus 1) plus n which is, this part is (b minus 1) times (i plus n minus i) plus n (b minus 1) times n plus n which is b n which is also our left hand side. So this is correct. This is a solution to this reference. No harm, so this is the time taken and we will say other ways of arguing the same thing.

(Refer Slide Time: 13.22)



$T(n,b)$ is the time to sort n, b-bit no's.

$T(n,b) = \underline{T(i, b-1) + T(n-i, b-1) + n}$

S'pose $T(n,b) = bn$

R.H.S. $= \underline{i*(b-1) + (n-i)*(b-1)} + n$

$= (b-1)(i+n-i) + n$

$(b-1)n + n = bn$

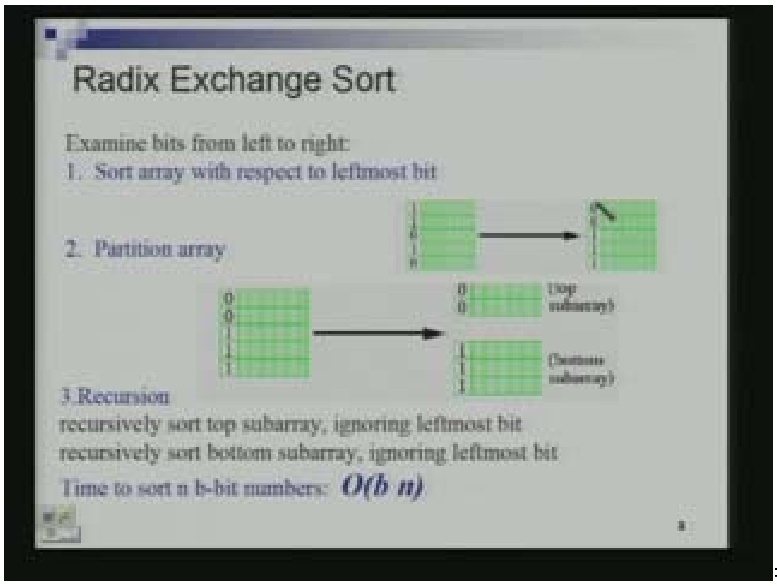
$= T(n,b)$

No, we don't have to do that. This would be the same for all choices of i, we don't have to any averaging here [student: but when we did quick sorting]. That was because we were doing a randomized quick sort. We were computing the expected time [i can be anything], i can vary but the whole point is no matter what i is. It will always be the same. Their depending upon what i were, the time would be different and so we were computing the average. If we try to compute the average here, you would get still the same. The point is it is always the same no matter what i is. No, even then you don't have to go. If you do repeated substitution, once again you will see of course it will be more complicated which is why I did not do that method here. You can also solve it by repeated substitution and you will get the same answer. It is just that you will have to keep track of what i's you use in the various points in the recursion and that will make it a bit more cumbersome. That you will get a same solution. These are all b bit numbers that is what we assume.

The maximum number of bits is b exactly, n has to be less than or equal to two to the power b. No, this is not true. I never said these are distinct numbers, I could have repetitions. [Student: can't we just say that we have] so that would be one way of arguing it. I just wanted you to show, how to solve recurrence relations also. There could be many ways of arguing the same thing, that's one argument. Everyone understands what the algorithm is. Yes, how do you combine in this divide and conquer step. We don't need to combine once we have sorted this top sub array and we have sorted this bottom sub array. All the numbers are b bit exactly. Here that is why this is all uniform. That is why I said, you will take the largest number, see how many bits you need to represent in that and you will use that many bits to represent every number. Otherwise it does not work.

You just target with zero's to the left. If 8 is 1 0 0 0, 4 bits then 2 would be 0 0 1 0, 4 bits. Great so let's continue [student: what about negative] we will not worry about negative numbers right now. If you had negative numbers and positive numbers what will you do? First split the numbers into negatives and positives, sort them separately and put them together. Why make life more complicated in that? You can always sort them separately. No, that will not happen. We will see more examples and it will be clear. In the previous slide I said that you will have your zero's before your one's. That was the first step we did let me go back [HINDI] we took these numbers, we changed this so that you had the numbers in which the leftmost bit was a 0 appearing before the numbers in which the leftmost bit was a 1. We did this kind of partition.

(Refer Slide Time: 17:28)


Radix Exchange Sort

Examine bits from left to right:

1. Sort array with respect to leftmost bit
2. Partition array

0
0
1
1
1
1
0
0
(top subarray)
1
1
1
(bottom subarray)

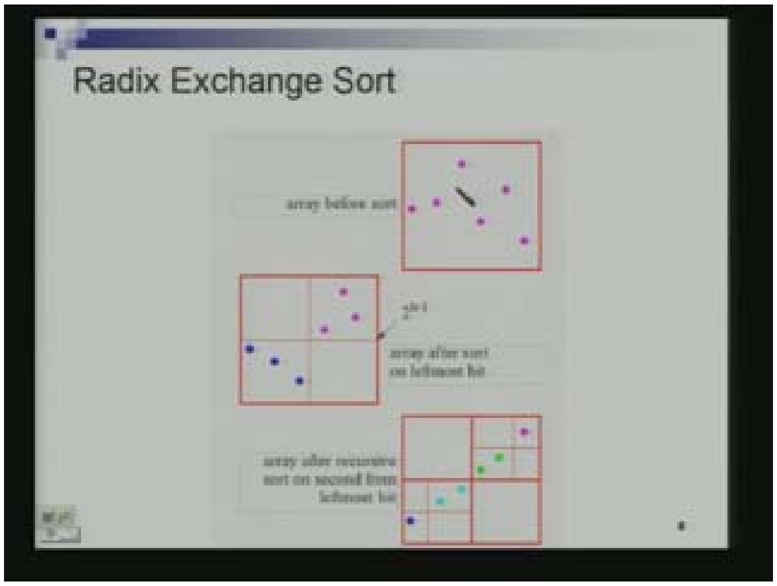
3 Recursion
recursively sort top subarray, ignoring leftmost bit
recursively sort bottom subarray, ignoring leftmost bit
Time to sort n b-bit numbers: $O(b\ n)$

#

How do you do this quickly? If you recall in my recurrence, I wrote order n for this and you did not arrays occur on that. The point is we can use the partitioning algorithm that we employed in quick sort to do this. What does that mean? We will scan from top to down, finding the first key with the one in the left most bit and from bottom to up finding the first key with a zero in the left most bit and swap. The same kind of technique we use for quick sort and we will exchange these keys and we will keep doing that until you know the scan indices exchanged. So that we know that there are zero's above and ones below. How much time did it take? At most the size of the array.

In this manner we can do the partition and then we can just call it recursively. We will get a time of order n b. So that is what we are doing here. We are scanning from the top to bottom so this is the first place we find a 1, this is the first place we find a 0. We swap them, 0 comes here, 1 comes here. Then the next index would be this one and the next here would be this zero. We will also scan them and we would get this and now we are done.

(Refer Slide Time: 18.41)

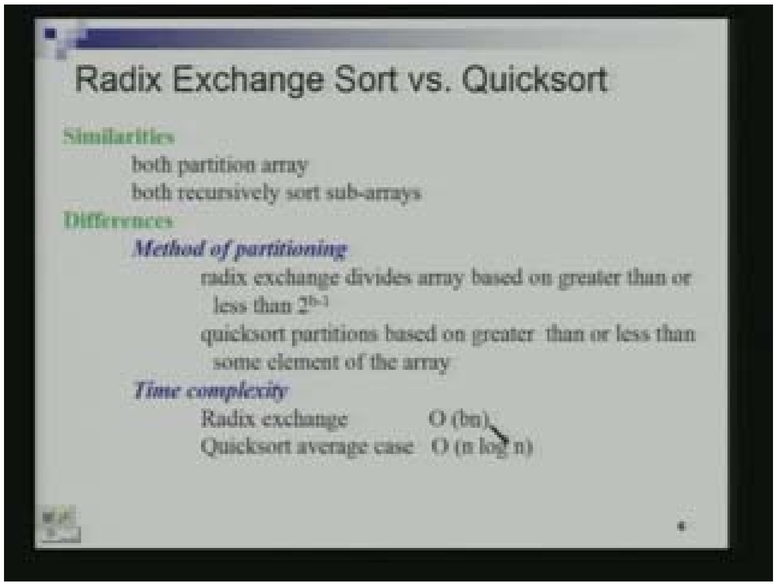

Radix Exchange Sort
array before sort
2^{b-1}
array after sort on leftmost bit
array after recursive sort on second from leftmost bit

What is happening? How is this different or related to a quick sort? I will come to that in a minute but before that see what we are doing at each step. Suppose these are the numbers before we sort them. What does this mean? These are the numbers, the first number in my array is this let's say this is the value. The y coordinate is the value of the number, this is the number, the second number is this, this is the third, this is the fourth, this is the fifth, this is the sixth. What I am doing at the first step is that I am partitioning the numbers according to numbers which are more than 2 to the b minus 1 and numbers which are less than 2 to the b minus 1. The numbers which are less than 2 to the b minus 1. Why 2 to the b minus 1? Because the first bit, the most significant bit we are saying should be a 1. They will come together and the ones for which it is a 0, they will come together.

I am partitioning the numbers according to 2 to the b minus 1. Those numbers whose value is more than 2 to the b minus 1, I am moving it to the right part of my array. This is what is happening. These are the numbers with values more than 2 to the b minus 1, so they are in the right part of my array and the numbers which are less than 2 to the b minus 1, they are in the left part of my array. I repeat this on this one. Once again I am going to divide this up and now I am going to consider the numbers here which are larger than 2 to the b minus 1 plus 2 to the b minus 2 essentially. Within this part of course more than 2 to the b minus 2 after I ignore the leftmost bit but otherwise, so we keep doing this eventually we will get something like this which corresponds to a sorted sequence. This is pictorially what is happening. But you understand the algorithm. Now how does this compared to quick sort? Both the algorithms partition the array, both recursively sort the sub arrays. So structure is very similar, the difference is in the way we partition. In the case of radix exchange sort, we are partitioning with respect to not a pivot element but with respect to a fixed quantity.

We are saying anything more than 2 to the power b minus 1 goes there in the bottom half of the array, anything less than 2 to the b minus 1 goes in the upper part of the array. It is the method of partitioning which makes the difference. In the radix exchange, divide the array based on whether the number is larger than 2 to the b minus 1 or less than. While in quick sort we are partitioning based on a pivot element. The difference is also in the time complexity. For radix exchange we argued just now that the time complexity is order bn.

(Refer Slide Time: 21.52)



Radix Exchange Sort vs. Quicksort

Similarities
- both partition array
- both recursively sort sub-arrays

Differences

Method of partitioning
- radix exchange divides array based on greater than or less than 2^{b-1}
- quicksort partitions based on greater than or less than some element of the array

Time complexity

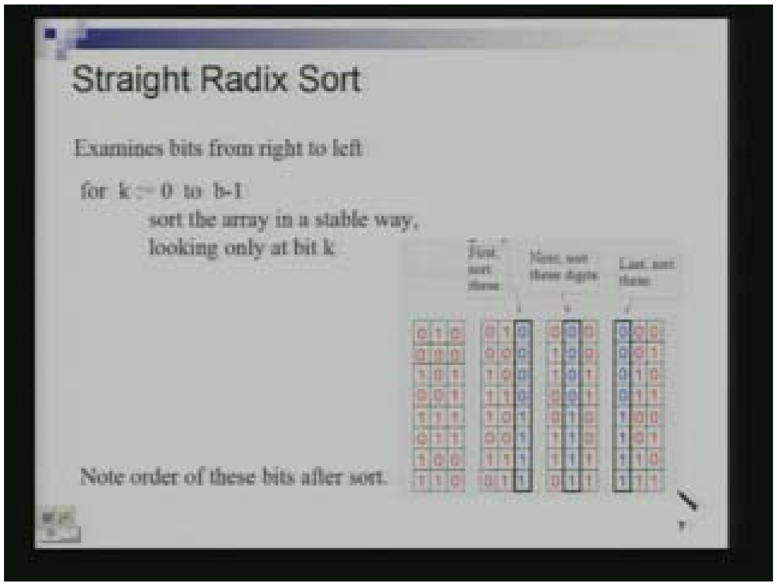
Radix exchange	O (bn)
Quicksort average case	O (n log n)

For quick sort we argued that the average case is n log n. Sometimes this might be a better scheme depending upon what the value of b is for you. That was a radix exchange sort where we were exchanging the elements in the array. I am going to look at another one version of radix sort, the principle is the same. This is also another way of implementing radix sort. Quick sort can be better when b is larger than log n. b can be larger than log n. You can have one number, you can have numbers which are 2, 3, 7, 11 and one number which is one million and three. Now you need a huge numbers of bits because there is one very large number so that b would be much larger than log n. Because the number of numbers is very small but the largest number is very large so that the number of bits you require to do your sorting is large.

We will continue. This is another version as I said of radix sort, straight radix sort. We are going to examine once again the bits from right to left now. Not from left to right but from right to left. The k equal to 0 corresponds to the right most bit now. The least significant bit also called the least significant bit. So k equal to 0 is a least significant bit and b minus 1 is the most significant bit. We are going to sort the array based on this bit, the k th bit in a stable way. Sort the array in table way looking at only k th bit. Let’s see what this means? Do you understand what’s table way means? No, great we will come to that in next slide but let me show you what this is. This is the collection of numbers you have, what the algorithm is. You are first going to look at the rightmost bit and you are going to sort these numbers based on the right most bit.

As you can see after you sorted them, you first have the numbers which have the rightmost bit as 0 then you have the numbers which have the rightmost bit as a 1. Then you sort these numbers based on the second right most bit. This corresponds to k equals 1, so you are sorting them based on this and then finally you are sorting these numbers based on this. What will happen? At the end of that you will have a sorted sequence.

(Refer Slide Time: 24.29)



Straight Radix Sort

Examines bits from right to left

```
for k := 0 to b-1
        sort the array in a stable way,
        looking only at bit k
```

Note order of these bits after sort.

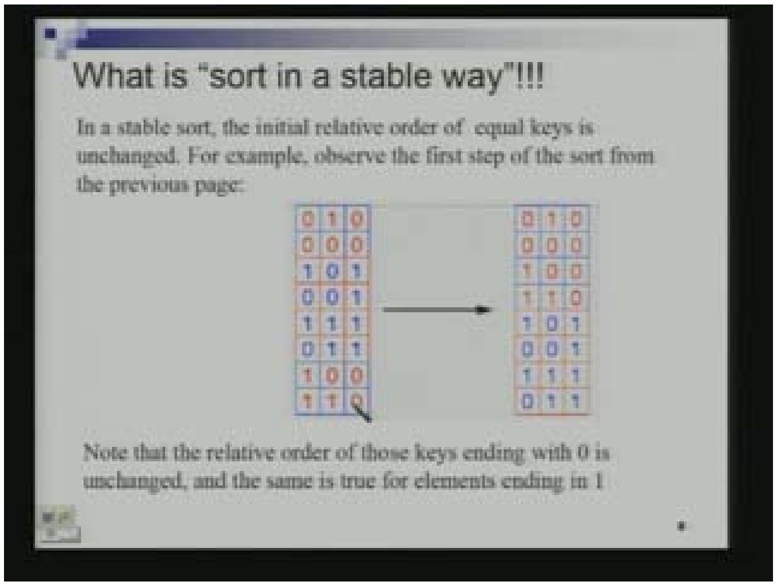
As you can see this is originally non-sorted and what you have here is a sorted sequence. Why is this magic happening? You understand the algorithm. Take the rightmost bit, sort the numbers based on that which means that just restrict your attention to the rightmost bit. Anything that is a 0 comes before everything that is a 1. Now you have done some rearranging of the numbers. Now look at the second rightmost bit. They do the sorting with respect to the second right most bit. Everything which is a 0 comes before everything which is a 1 and so on and on.

How much time does this take? b n once again, because for each bit you are spending time proportional to n [student: partitioning] partitioning will not be used. We will see how to do it. It is not completely clear why b n but will come to that argument also in a minute. First we need to understand what is sorting in a stable way. What does this mean? A table sort is one in which two numbers are the same then after sorting, so if two keys are the same, equal keys then after sorting their relative order remains unchanged. Suppose I have two numbers so I have a collection of numbers. I have 1 3 11 3 5, these are my numbers. I have two threes in there, equal keys. Now if after sorting, of course after sorting this array would become 3 3 5 11. But the two 3's I have now, suppose with those threes I had one which was colored red and the other was colored blue.

The first was colored red and the second was colored blue and after sorting, the red should appear before the blue. Although they are still threes, they should appear in the same relative order as was the case before we did the sorting.

That is called table sort and we will see why it is relevant here and why it is important here. Let’s look at this now. This is what we were sorting and let us say we are sorting with respect to the right most bit.

(Refer Slide Time: 27.05)



What is “sort in a stable way”!!!

In a stable sort, the initial relative order of equal keys is unchanged. For example, observe the first step of the sort from the previous page:

0	1	0
0	0	0
1	0	1
0	0	1
1	1	1
0	1	1
1	0	0
1	1	0

$\longrightarrow$

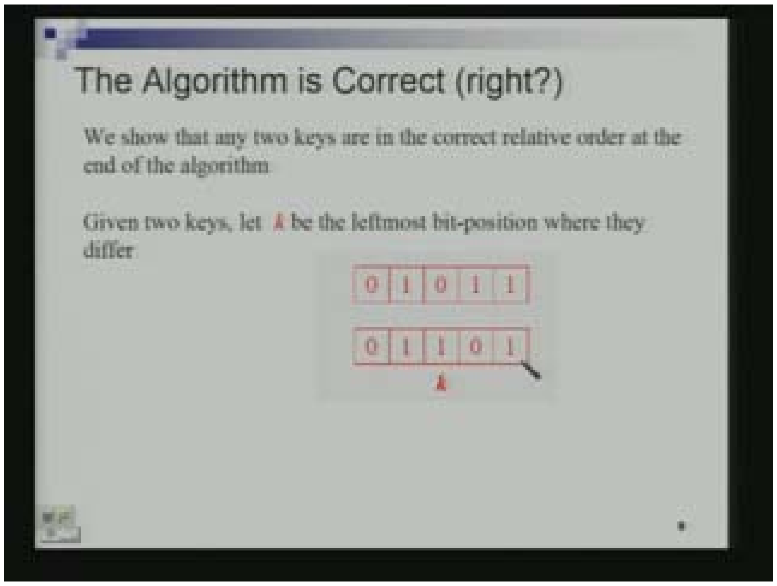
0	1	0
0	0	0
1	0	0
1	1	0
1	0	1
0	0	1
1	1	1
0	1	1

Note that the relative order of those keys ending with 0 is unchanged, and the same is true for elements ending in 1

I have 4 keys with 0, they will all come before the 4 keys which are at a one which have the rightmost bit at a one. But I would like that this appears before this, so the first number should better be 0 1 0. The next should be all three zero’s. The next should be 1 0 0, the next should be 1 1 0 which is the case here. I am not permitted to rearrange them. When I am saying I just sort with respect to the last bit, you could also create this array post sorted in which this was the first number but that would be wrong. That would not be a stable sorting and similarly for the one’s. The first should be a 1 0 1, the next should be 0 0 1, the next should be 1 1 1 and the next should be 0 1 1, this is crucial for the correctness of the sorting algorithm. If you don’t do it this way, this will not give you the sorted sequence at the end. So everyone understands what table sorting is.

Now let’s understand the correctness of the algorithm. We are now going to show that any two keys are in the correct relative order at the end that means if I take two keys, one is smaller than the other. Then in the end, the key which is smaller appears before the key which is larger, very simple proof. So suppose these are the two keys that were given to me. Let me look at the leftmost position at which they differ.

(Refer Slide Time: 28.43)



The Algorithm is Correct (right?)

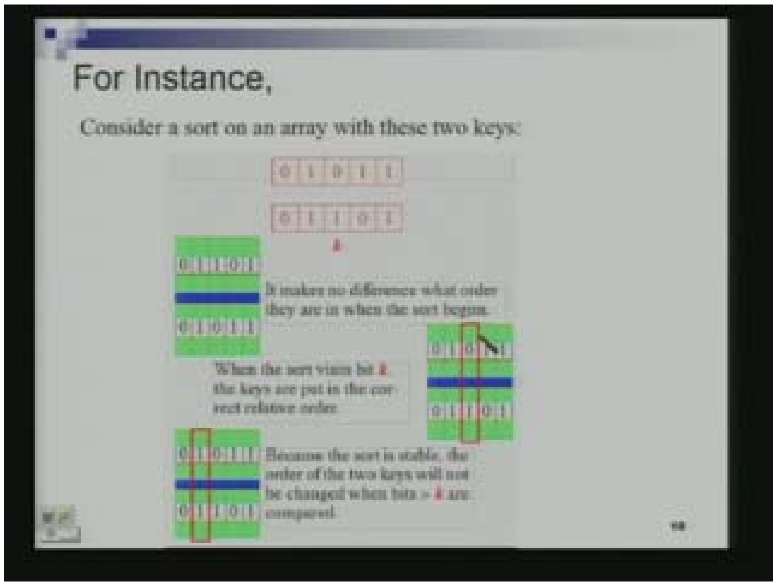
We show that any two keys are in the correct relative order at the end of the algorithm

Given two keys, let k be the leftmost bit-position where they differ

Leftmost, so this they don't differ at this place. They don't differ at this place even but they differ at this place. Let me call this position k (Refer Slide Time: 28:50). Now when I am sorting this bunch of numbers what is going to happen? When I am sorting remember I am sorting by considering first the right most then the second right most then the third right most and so on. Let's us understand this. The claim is that the step k, the two keys are put in the corrective relative order. At the first step they may be rearranged I don't care, they are put in some order. At the second step also they are put in some order, I don't know. But at the k th step, this key would put before this key because this is a 0 and this is a 1. At the k th step the two keys are put in the correct relative order but all is not done now.

We want to argue that at the latest step, at the k plus 1 th step and the k plus 2 th step and so on, the relative order is not interchanged [HINDI] because of stability. Now when I am looking at the k plus one th step, at the k plus one th step these are the same. These are the same, because it is a table sort this key which is appearing before this would continue to appear before this and similarly at this next step and so on and on. Beyond the k th step, the relative order would not change anymore. At the k th step you would get the right order between these two keys, the smaller key will appear before the larger key and add sub sequent steps, this relative order would be preserved. The smaller key would be continued to appear before the larger key [HINDI].

(Refer Slide Time: 31.13)



For Instance,

Consider a sort on an array with these two keys:

0 1 0 1 1

0 1 1 0 1

k

It makes no difference what order they are in when the sort begins.

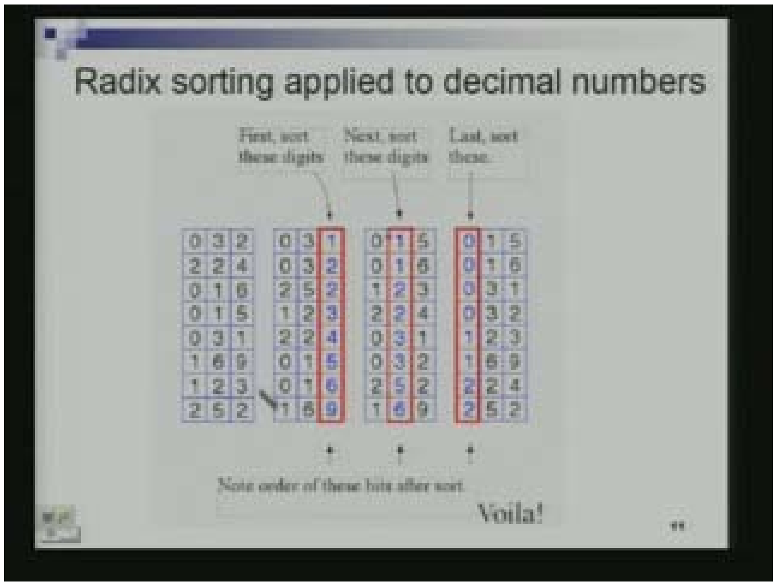
When the sort visits bit k the keys are put in the correct relative order.

Because the sort is stable, the order of the two keys will not be changed when bits > k are compared.

Let’s take an example. This is the two keys once again that I am considering. Initially they could be in some arbitrary order. I have in fact that the larger keys appearing before the smaller key. This is the array in which the numbers are… [HINDI]. This is location zero of the array and so on. You would like that the keys be in increasing order but right now bigger number is appearing before the smaller number. When I am looking at the k th step, at this step when I am sorting with respect to this bit, the k th bit I would have put 0 1 0 1 1 before 0 1 1 0 1 clearly because at the k th position, this is a 0 and this is a 1.

Now when I am looking at the next more significant bit, I would continue this relative order because at the next more significant bit they are the same. If they are the same then stable sorting ensures that I have to maintain the relative order that was there till that point in which this was appearing before this. So this will continue to appear before this step and in subsequent steps also. So because the sort is stable, the order of the two keys will not be changed when bits more than k are greater than or bits at position larger than k are compared. You can also see now, why I had to start from the right end. If I start from the left end then this technique is not going to work. Take this is an exercise, think of an example if I were to start from the left end, this would not give me a sorted sequence at the end. There is nothing sacrosanct about binary numbers, I can also apply the same technique to decimal numbers.

(Refer Slide Time: 33.15)



Radix sorting applied to decimal numbers

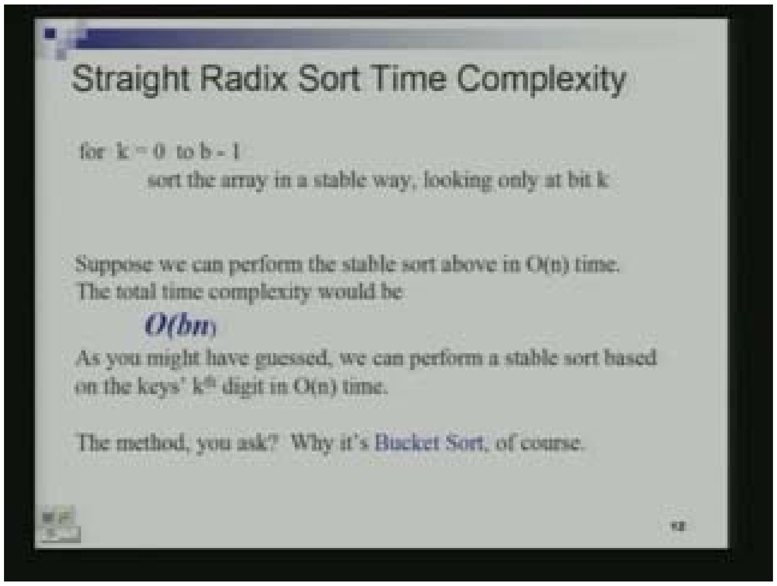
0 3 2	0 3 1	0 1 5	0 1 5
2 2 4	0 3 2	0 1 6	0 1 6
0 1 6	2 5 2	1 2 3	0 3 1
0 1 5	1 2 3	2 2 4	0 3 2
0 3 1	2 2 4	0 3 1	1 2 3
1 6 9	0 1 5	0 3 2	1 6 9
1 2 3	0 1 6	2 5 2	2 2 4
2 5 2	1 6 9	1 6 9	2 5 2

What do I do? First I sort with respect to the right most digit which means that the ones would come before the two's would come before the three's and so on and on. Again the sorting is stable. After the first step as you can see, the first number would be, so there is a 1 1, there are two 2's here and so on and on. There is a unique one 0 3 1, so 3 1 becomes the first number. Then there are these two 2's, 0 3 2 and 2 5 2 which should be the second number. 0 3 2 because it's stable. So 0 3 2 is the next, 2 5 2 is the third and so on and on. So I sort with respect to this. Next I sort with respect to this, so there is a 1 here, 0 1 5, there is another 1 so I will first put this and then I will put this and so on. As you can see at the end I get a sorted sequence.

Now we need to figure out the time complexity, how much time have you taken? So how man passes, such passes are we are making? We are making as many passes as the number of digits or the number of bits or whatever it is. But we need to now see what we are going to do in one pass. How are we getting a one's before the two's before the three's and so on in a stable manner. How much time does that take? What kind of a scheme should be employed for that?

Exactly, for exchange radix sort we basically wanted to partition the array into two parts only but here because these are digits, not just 0 1 it is not a two way partition anymore. For decimal numbers, we will have to represent it in a binary form to be able to do this. What is like an insertion? I don't quite follow what you are saying. We will discuss this later. Let's figure out what the time complexity is. For k equal to 0 to b minus 1, we are sorting the array in a stable way looking only at k th bit. Suppose this could be performed in order n time then the total time complexity would be order bn. That's completely clear provided we can do this sorting in order n time which sorting we are talking about. We are looking at a particular digit, one pass. We are looking at a particular digit or a bit and we want to ensure that all the numbers, if I am looking at decimal numbers all the numbers are sorted based on that digit. The one's before the two's before the three's and that the sorting is stable. We want to be able to do this in order n time and the method of choice is what is called the bucket sort algorithm.

(Refer Slide Time: 36.34)



Straight Radix Sort Time Complexity

```
for k = 0 to b - 1
        sort the array in a stable way, looking only at bit k
```

Suppose we can perform the stable sort above in O(n) time.
The total time complexity would be

$O(bn)$

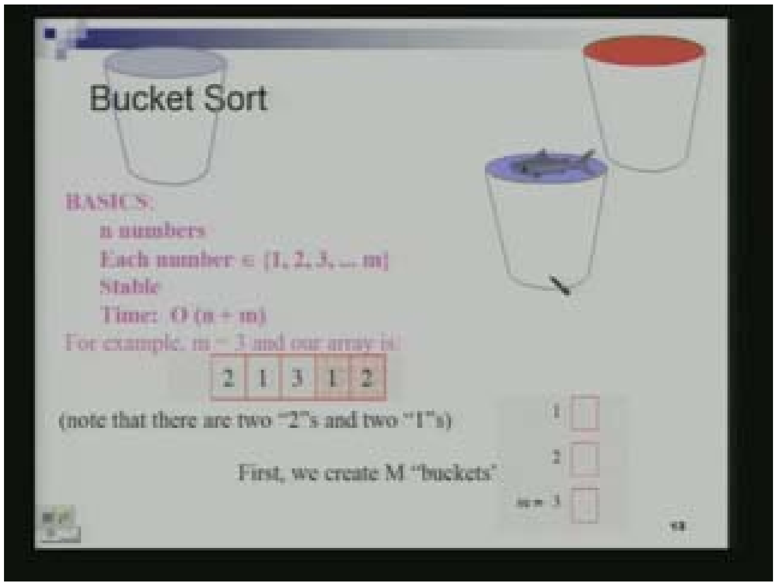
As you might have guessed, we can perform a stable sort based on the keys' k^{th} digit in O(n) time.

The method, you ask? Why it's Bucket Sort, of course.

That brings us to the second sorting scheme that we talked about. So what is bucket sort? Lots of buckets. We have n numbers, each number is in a certain range. Let's say one through m, so bucket sort is a stable sorting algorithm and it will take time order n plus m. You understand what we are talking of. This is very useful when you have a large number of numbers with lots of duplicates perhaps and the numbers are coming from the small range.

Then you don't need something like n log n time or some such thing. You can then do it in time order n plus the range of the numbers essentially. Let's see how this works. Suppose this is my collection of numbers 2 1 3 1 2 so m is 3 because you can see the numbers are in the range 1 2 3. The m is 3, there are two two's and two one's. So first what we are going to do is you are going to create an M buckets. You can understand what you will do in each bucket. Just take a number and throw it in an appropriate bucket, so we have these 3 buckets.

(Refer Slide Time: 38.04)



Bucket Sort

BASICS:

- n numbers
- Each number ∈ {1, 2, 3, ... m}
- Stable
- Time: O (n + m)

For example, m = 3 and our array is:

2	1	3	1	2

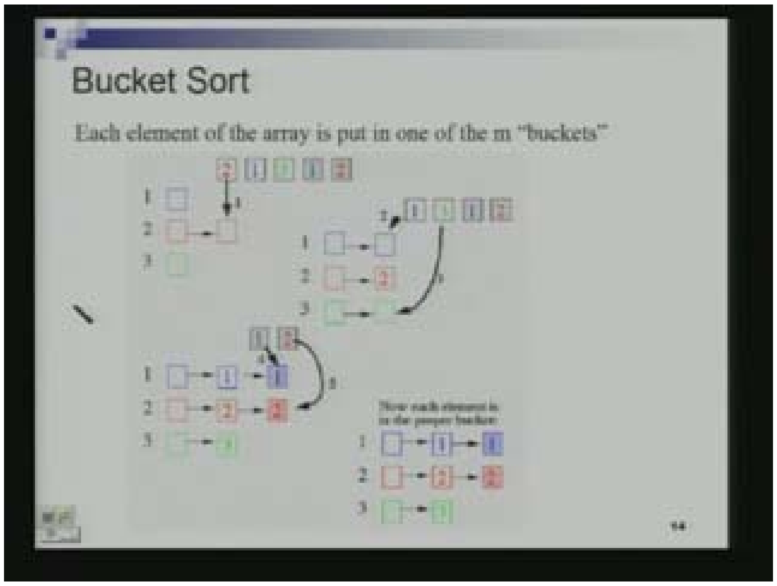
(note that there are two "2"s and two "1"s)

First, we create M "buckets"

13

One corresponding to each possible value in this range and each m element of array is put in one of the m buckets. So these are my buckets, take the first number it goes into bucket two, I put it here.

(Refer Slide Time: 38.17)



Bucket Sort

Each element of the array is put in one of the m "buckets"

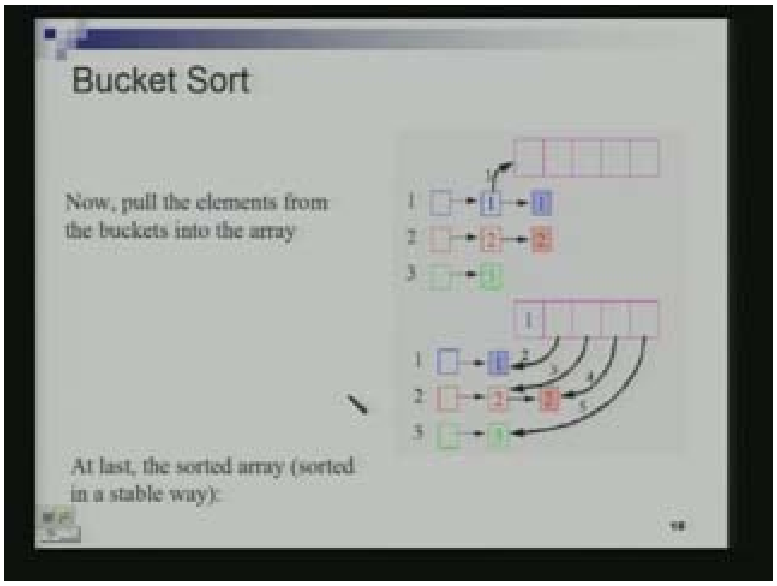
Now each element is in the proper bucket:

14

Then I take the next number it goes into bucket one, so I put into here. The third number goes into bucket three, I put it here. The fourth number goes into bucket one, so I append it at the end of this list. End is important to maintain stability and then I take this two and append it at the end.

Now I will just read the numbers, I will take the numbers in the first bucket, basically append all of these lists. So 1 1, 2 2, 3 and so on. I will put the elements from the buckets into an array and just read it off in this manner.

(Refer Slide Time: 39.05)



Bucket Sort

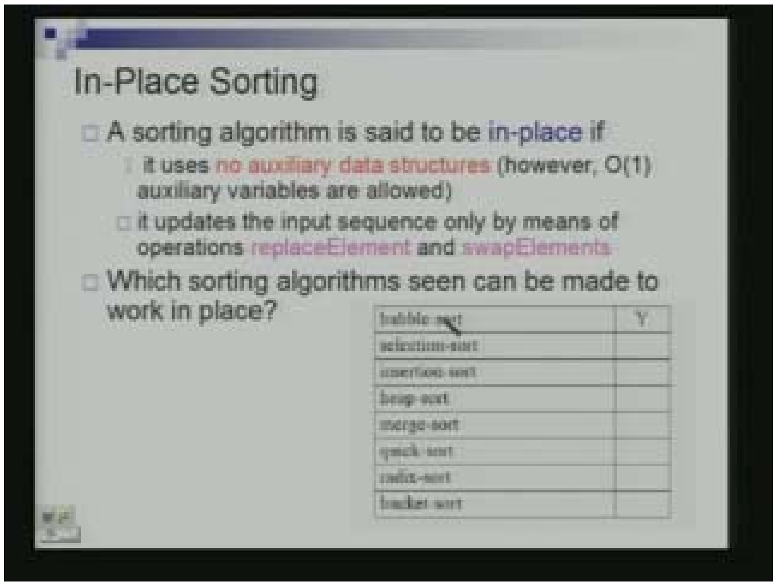
Now, pull the elements from the buckets into the array

At last, the sorted array (sorted in a stable way):

This gives us a stable sorting. You understand why it is stable? Because if two keys are the same then they would be in the bucket but we would also have put them in the right order. That's why we are appending. If we were attaching at the front then we would have to read it the other way around which is the same as that. So with that you should be able to argue that our straight radix sort takes order bn time now. We said for each pass we want to do it in order n time and you can do it order n time using such a scheme, using bucket sort. So you do that in order n time, there are b passes in all so it becomes order bn. Yes, you are going to get one question from this in the exam. Yeah, okay in-place sorting. Yes, you want to know what the question is.

A sorting algorithm is said to be in-place if it uses no auxiliary data structures. It could use a constant amount of additional space over here and it updates the input sequence only by means of the operations replaceElement and swap. Basically it's just you have a bunch of numbers, it replaces one of them by some other or it just swaps two numbers. That's when we call the algorithm to be in-place. So let's see which algorithms we have seen can be made to work in in-place.

(Refer Slide Time: 40:59)



In-Place Sorting

- A sorting algorithm is said to be in-place if
  - it uses no auxiliary data structures (however, O(1) auxiliary variables are allowed)
  - it updates the input sequence only by means of operations replaceElement and swapElements
- Which sorting algorithms seen can be made to work in place?

bubble-sort	Y
selection-sort	
insertion-sort	
heap-sort	
merge-sort	
quick-sort	
radix-sort	
bucket-sort	

Bubble sort actually you have not seen right or you know what bubble sort is? You don't know what bubble sort is? I will not go into bubble sort then. Let's see. Who can tell me heap sort. Is heap sort in-place? [Student: yes sir] We can have one array and basically implement a heap in that array and we are just changing the elements in that array. Merge sort. Is merge sort in-place? [Student: no] Why not? To merge two list, you need additional space. You cannot merge in the same list, so merge sort is not in-place. Quick sort [student: yes sir] Quick sort is in-place because we partition in the same array and then we just did recursively left and right.

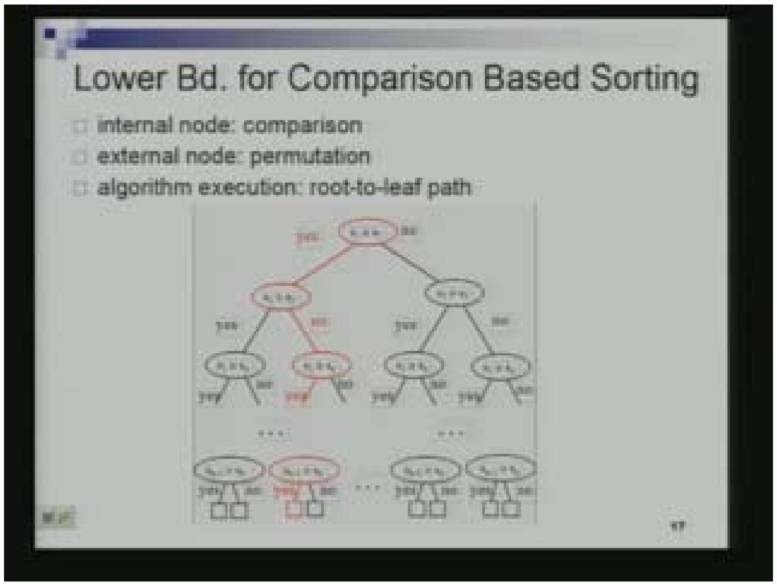
To do the merge, to merge two lists you need additional space because what were we doing in merge? We are taking the first element of the two list, comparing them and putting it out into some other space. You can't just copy it back there, it would not work. [Student: in an algorithm uses what are one space would increase] Yeah, order one space is okay. It is in-place but not space proportional to the number of elements, it should be independent of the number of elements. You can look at the other algorithms and think off whether they are going to be in-place or not. Radix sort. Is radix sort in-place? But number of buckets is independent of the number of numbers. [Student: but we are inserting] then nodes that we are creating. Yes, that is additional space.

So can you modify the scheme to make it in-place? [Student: to keep every bucket just count the number of those elements that are in the array] Yeah, you will have to think about it, think about this. Can you make it in-place? Can you make radix sort in-place? It is a good thing to think about. Let me get to the last topic that we are going to cover as far as sorting is concerned and that is a lower bound for comparison base sorting. What does comparison based sorting mean? It basically means that we are only looking at sorting algorithms in which all you have permitted to do is to compare two numbers.

Suppose you have a bunch of elements, I am not even saying numbers now and you want to sort those elements. I give you those elements, I have a comparison function which you have to use to do the sorting. So like your comparator, so you give me the two numbers, I will tell which of them is smaller than the other because may be these are not numbers but some objects and I am the only one, who knows how to compare these objects. That is a comparison operation, you give me these two numbers, I decide whether one is less than the other or not, whether the first is less than the second or the second is less than the first and I give you the answer.

Now the question is how many times will you have to ask me for a comparison? You understand, you wanted to sort this n numbers, how many times will you have to ask me? May be I charge you 1 rupee every time you give me certain comparison to do. The comparison is let us say an expensive operation. Every time you say compare these two numbers for me, I am going to charge you 1 rupee. How much money are you going to spend? n log n, you have seen algorithms which would take no more than n log n time. What we are going to argue now is that there can be no algorithm which takes less than n log n time, n log n comparisons. No one can come up with an algorithm so that algorithm will always take less than n log n comparisons for all inputs. For certain inputs it could take less than n log n comparisons but for all inputs, it would take n log n comparisons. That is not possible at all. We are going to understand this in the following way.

(Refer Slide Time: 45:50)



Lower Bd. for Comparison Based Sorting

- internal node: comparison
- external node: permutation
- algorithm execution: root-to-leaf path

yes
no
yes
no
yes
no
yes
no
yes
no
yes
no
yes
no
...
...
yes
no
yes
no
...
yes
no
yes
no
17

Let’s look at the particular algorithm. You have a certain algorithm and let’s say your objects are sitting in some array and you algorithm works on that array. The first step it’s going to ask me to compare two elements of that array. Let say that two elements are at position S_1 and S_2 [HINDI]. So my very top node here is this node whether S_1 is greater than S_2 or not. It is going to ask me this and I am free to say whether one is less or whether one is more based on what my comparative functions says. These are so to say, the questions that the algorithm is asking me.

First it asks me to compare S_1 and S_2. If I said a yes then it would have asked me for a comparison of S_1 and S_3. Let's say I have such a thing. If I had said no, maybe it ask me for a comparison of something else, this need not be S_1 S_3. It could be something else, depends upon what the algorithm is. But at each point it is coming back to me with certain comparisons, with certain numbers to compare.

First time it says something then depending upon whether I say yes or no, then the next comparison it ask me something else. Now depending upon whether I said yes or no, the next comparison it could ask me would be something else and so on and on. The execution of the algorithm is really a path down this stream. Yes. Now at some point the algorithm is going to stop. It doesn't ask for anymore comparison, it says well I am done, this is your sorted sequence, so which means that this path ends in this external node here, this leaf node here which corresponds to a particular permutation of those numbers. This sequence of moves would have been made, if I had certain ordering on the numbers for a certain permutation of the numbers.

Let's understand this. [student: there would be some finite criteria of the let me go for a S noise it is not necessary say S_1 is less than S_2 we can compare any] we can compare anything, yeah. [Student: so those numbers which we are comparing that is randomized over the entire set, any two numbers we pick up randomly] No, so don't look at it that way. We are saying your algorithm, you have a certain algorithm which has the numbers written in an array, let's say one through n. At the very first step it is going to come and make certain comparison. Let's assume it is deterministic algorithm, no randomization for now. So same argument applies for randomization also but for now let's assume it's deterministic.

It will say compare lets say, first time it comes to me it will say look at the number in location 3, look at the number in 7 and tell me which is smaller. Whether the number in location 3 is less than the number in location 7 or whether the number in location 7 is less than the number in location 3. I put a node here saying let's say this array was S, so whether S_3 is less than S_7 is the first comparison it asked me for. If I had said a yes, I do not know. If I said a yes, it would go and ask me to compare some other two numbers. It's your algorithm, I don't know what it is going to ask me. But whatever it is going to ask me, I am going to put down here.

Suppose it came and said S_2 verses S_5, whether S_2 is less than S_5 and if had said a no, may be it came back and asked me something else. It came and asked me S_6 is less than S_{13}, it may be. It is your algorithm [HINDI] so depending upon what option do I have. I have an option of basically saying my yes and no. That's all I say. I say a yes then I say a no and so on and on, depending upon what I rate as the relative order of these elements in these array. Think of these objects are some complicated objects. You don't know what the relative order is so that's why you are asking me about that. I have some way of figuring out what the relative order is. May be today I feel like that the relative order should be based on gpa. Then tomorrow I feel like relative order should be based on height or whatever it is like, I can decide. For that I would make a certain choices of yes and no, which would eventually end up in leaf, in an external node of this. When you reach this say, you well I have sorted it. You will sort because this is all the comparisons you did.

The question is how many comparisons did you have to do? The number of comparisons is basically the length of the path that you took, the height of this tree. Yes. So how many comparisons would you have to do? How height does this tree have to be? [Student: n log n] Why n log n? [Student: n factorial permutations of the array so they have to be n factorial leaves] Yes, so that's right. There have to be n factorial leaves in this tree. Now this is not a straight forward thing to understand [HINDI]. Depending upon what these objects are, I should be able to get all kinds of permutations. Every permutation is a possible solution at the end, every permutation of these n numbers. What is sorting? A sorting you are given the elements like this and eventually what do you generate? You generate a permutation of these elements. Yes or no? [HINDI] what is this? This is just a permutation of this set of elements [HINDI] this is just a permutation of this. Now all possible permutation should exist as leaves. That depends upon what the relative order is, what I have picked as the relative order.

So given a certain permutation, if that is what I picked as the relative order then your algorithm should end up in that. If I took some other permutation and picked that as the relative order then your algorithm should end up in that and so on and on. Every leaf of this tree corresponds to a permutation and further more every permutation should be representable as a leaf. So which means that this tree has n factorial leaves. It is not a complete binary tree, may be it is. I don't care; I know it is binary tree. It's a binary tree with n factorial leaves. So what is its height going to be? At least log of n factorial. So height therefore is at least log of n factorial, at least [HINDI] yeah [student: sir our order to comparator, does it satisfy the symmetric or the transitive property] Yes, it satisfies all of them.

Even then it can. I take a particular permutation of these elements and I say I am going to answer with respect to this permutation. All queries that are asked for me, I will answer with respect to this particular permutation that I have in my head. [student: is it possible, if suppose I have S_1 there are 3 S_1 S_2 and S_3] yeah [student: and at some point at that we have some decisions of S_1 S_2 and S_2 S_3] yeah [student: and then other point in the tree the other point in the tree decision based on S_1 and S_3] yeah [student: that cannot take both the paths yes and no because] Yeah I understand that, but that is not the point. The point all we are trying to say here is that [student: it is not necessary every node] that I could have a certain permutation in my head and I could use that to answer all your questions and it could consistent.

So there have to be n factorial leaves and if in a binary tree, there are n factorial leaves. Then basically if there are some n leaves in a binary tree then it has to have a height of at least log n. So which means that the tree has to have a height of at least log of n factorial. Height of the tree is at least log of n factorial. This is roughly n log n which means that there is a certain permutation. What is the height? The height is the distance of the longest of the farthest leaf from the root.

If this is the furthest leaf from the root, then if this were the permutation then your algorithm is going to take a time of at least n log n, on number of comparisons at least n log n it's going to take [HINDI]. This is the argument, we can go over the slides once again and understand it more carefully.

This is the argument for why any comparison based sorting algorithm has to have at least n log n time. So recall you are only permitted to compare two keys. Radix sort is not an example of a comparison based sorting algorithm because you are not comparing keys. You are going into the keys, looking at the bits or the digits and so that's why radix sort doesn't have the complexity n log n. It could be less than n log n, if B the number of bits or digits is less than log n. Yeah, so radix sort is the only such. All other algorithms have to have because they are all comparisons based sorting algorithm, they have to have complexity of at least n log n and there are many which achieve that bound. So with that I am going to end today's class. We looked at radix sort, we looked at bucket sort, we understood what stable sorting is and finally we saw this lower bound on comparison based sorting.