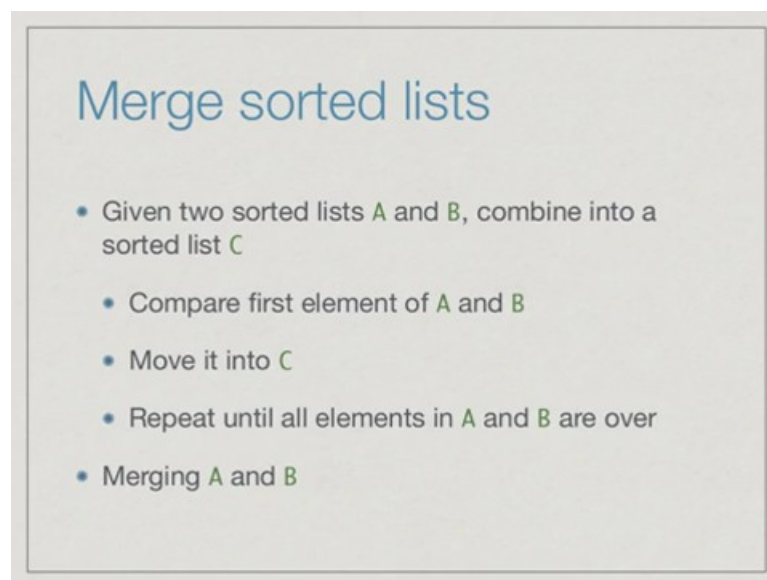**Programming, Data Structures and Algorithms in Python**
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 04
Lecture - 02
Merge Sort, Analysis

In the last lecture we looked at Merge Sort and we informally claimed that it was much more efficient than insertion sort or selection sort and we claimed also that it operates in time order n log n.
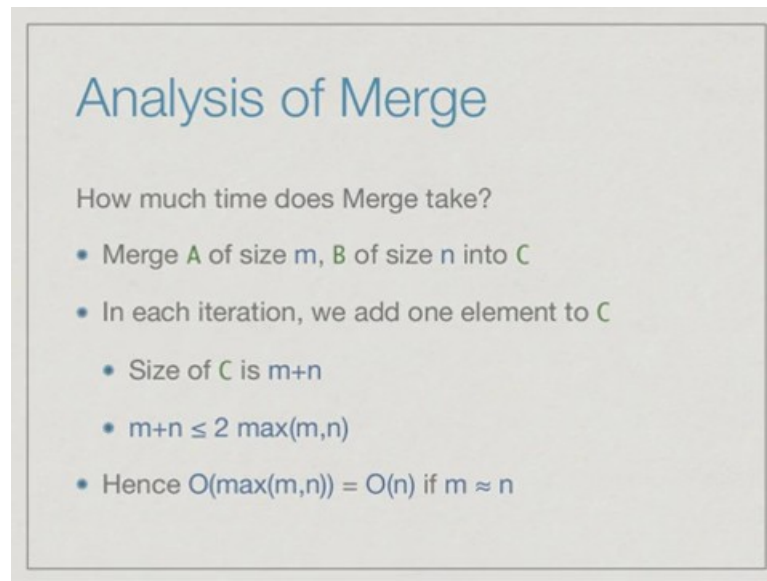
(Refer Slide Time: 00:03)

Recall that merge sort as its base a merging algorithm which takes two sorted lists, A and B and combines them one at a time by doing a scan over all elements.
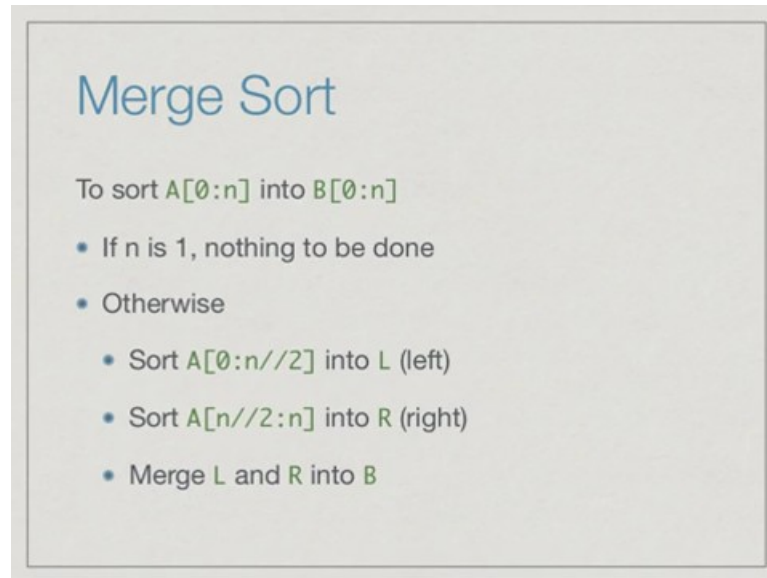
In order to analyze merge sort, the first thing we need to do is to give an analysis of the merge function itself. How much time does merge take in terms of the input sizes of the two lists, A and B. So, suppose A has m elements and B has n elements and we want to put all these elements together into a single sorted list in C. Remember that we had an iteration where in each step of the iteration we looked at, the first element in A and B and move the smaller of the two to C. So clearly C grows by one element with each iteration and since we have to move all m plus n elements from A and B to C, the size of C is m plus n.

What do we do in each iteration? Well we do a comparison and then, we do an assignment and then, we increment some indices. So, this is a fixed number of operations. So, it is a constant. So, the total amount of work is proportional to m plus n. Notice that m plus n is at most twice the maximum of m plus n. So, m is 7 and n is 15, then 5 plus 7 plus 15 will be less than two times 15.

We can say that merge as a function takes time of the order of maximum of m and n and in particular very often like in merge sort, we are taking two lists of roughly the same size like we divide a list into two halves and then, we merge them. If both m and n are of

the same, approximately the same size, then the max of m and n is just one of them in itself. Essentially merge is linear in the size of the input list.

(Refer Slide Time: 02:06)

Merge Sort

To sort `A[0:n]` into `B[0:n]`

- If n is 1, nothing to be done
- Otherwise
  - Sort `A[0:n//2]` into L (left)
  - Sort `A[n//2:n]` into R (right)
  - Merge L and R into B

Now, having analyzed merge, let us look at merge sort. So, merge sort says that if a list is small, has zero or one elements, nothing is to be done. Otherwise you have to solve the problem for two half lists and then, merge them.

As with any recursive function, we have to describe the time taken by such a function in terms of a recurrence. So, let us assume for now since we are going to keep dividing by 2 that we can keep dividing by 2 without getting an odd number in between.

Let us assume that the input n is some perfect power of 2. It is n is 2 to the k. When we breakup merge sort into two lists, we have two lists of size n by 2. The time taken for n elements is two times time taken for two list of n by 2 and this is the merge component. We have an order n step requires us to merge two lists of size n by 2 and remember we just said that merge is linear in the size of the input. So, we have two sub problems of size n by 2, that is two times n by 2 and we have merging which requires order n. As with binary search and with recursive insertion sort, we can solve this recurrence by unwinding it.

(Refer Slide Time: 03:33)

We start with the base case. If we have a list of size 1, then we have nothing to do. So, T of n is 1 and T of n in general is two times T n by 2 plus n. If we expand this out, we read substitute for T n by 2 and we get two times T n by 4 plus n by 2 because that if we take this as a new input, this expands using the same definition and if we rewrite this. So, we write two times 2 as 2 square and we write this 4 as two squared. We will find that this is equivalent to writing it in this form 2 into 2, 2 squared T n by 2 square and now notice that you have two times n by 2 over here. This 2 and this 2 will cancel. So, we have one factor n and the other factor of n. The important thing is that you have 2 here in the exponent and you have 2 here before the n.

Now, like wise what we will do in the next step is to expand this two times T n by 4. So, we expand two times T n by 4 and that will give us another n by 8 which you write as n 2 cube which used to be 2 squared 2 n by 2 square plus 2 n will turn out to be 2 cubed 2 n by 2 cube plus 3 n. So, notice that the two's have become threes uniformly. So, in this way if we keep going after k steps or j steps, we will have 2 to the j times T n by 2 to the j plus j times n. Now, how long do we keep doing this? We keep doing this till we hit the base case. So, when j is log of n, where log by log we usually mean log to the base 2, then n by 2 to the j will be 1. So, T by T of n by 2 to the j will also be 1.

After log n steps, this expression simplifies to 2 to the log n plus log n times n everywhere we have a j, we put a log n and take this has become 1, so it has disappeared. We have 2 to the j is 2 to the log n plus this j has log n and then, we have n and this is 2 to the log n by definition is just n. So, 2 to the log n is n and we have n log n and by our rule that we keep the higher term when we do, we go n log n is bigger than n. We get a final value of O n log n for merge sort.

(Refer Slide Time: 06:00)

Merge turns out to be a very useful operation. What we saw was to combine two lists faithfully into a single sorted list inparticular our list if we had duplicates. So, if we merge say 1, 3 and 2, 3, then we end up with the list of the form 1, 2, 3, 3. This is how merge would work. It does not lose any information. It keeps duplicates and faithfully copies into the final list.

On the other hand, we might want to have a situation where we want the union. We do not want to keep multiple copies and we want to only keep single copy. In the union case here is what we would do. Let us assume that we have two lists and in general, we could have already duplicates within the lists. Let us suppose that we have 1, 2, 2, 6 and 2, 3, 5 then we do the normal merge. So, we move one here and now, when we hit two elements

which are equal, right then we need to basically scan till we finish this equal thing and copy one copy of it and then finally, we will put 3 and then, 5 and then 6.

When A and A i is equal to B j will increment both sides and make sure that we go to the end of that block. The other option is to do intersection. Supposing we want to take 1, 2, 6 and 2, 6, 8 and come out with the answer 2, 6 as the common elements, then if one side is smaller than the other side, we can skip that element because it is not there in both lists. So, if A i is less than B j we increment i, if B j is less than A i, we increment j and if they are equal we will take union, we keep one copy of the common element.

So, merge can be used to implement various combinations, combination operations on this. It can be used to take the union of two lists and discard duplicates. It can be used to take the intersection of two lists and finally, as an exercise to test that you understand it and see if you can use merge to do list difference.

List difference is a following operation. If I have say 1, 2, 3, 6 and I have 2, 4, 6, 8 then this difference is all the elements in the first list which are not there in the second list. Two is there here and it is here, you remove 2, 6 is there here and here remove 2. So, you should get 1, 3. So, if this is A, and this is B, then this is so-called list difference A minus B. So, see if you can write a version of merge which gives you all the elements in A which are not also in B, also known as list difference.

(Refer Slide Time: 08:49)

Now, merge sort is clearly superior to insertion sort and selection sort because it is order n log n, it can handle lists as we saw of size 100,000 as opposed to a few thousand, but it does not mean that merge sort does not have limitations. One of the limitations of merge sort is that we are forced to create a new array, every time we merge two lists. There is no obvious way to efficiently merge two lists without creating a new list. So, there is a penalty in terms of extra storage. We have to double this space that we use when we start with lists and we want to sort it within merge sort.

The other problem with merge sort is that it is inherently recursive and so, merge sort calls itself on the first half and the second half. Now, this is conceptually very nice. We saw that recursive definitions rec recursive functions are very naturally related to inductive definitions and they help us to understand the structure of a problem in terms of smaller problems of the same type.

Now, unfortunately a recursive call in a programming language involves suspending the current function, doing a new computation and then, restoring the values that we had suspended for the current function. So, if we currently had values for local names like i, j, k, we have to store them somewhere and then, retreat them and continue with the old values when the recursive call is done. This requires a certain amount of extra work.

Recursive calls and returns turn out to be expensive on their own time limits. So, it could be nice if we could have both the order n log n behavior of merge sort. And we could do away with this recursive thing, but this is only a minor comment. But conceptually merge sort is the basic order n log n sorting algorithm and it is very useful to know because it plays a role in many other things indirectly or directly.