




# IIT Madras

ONLINE DEGREE

**Programming, Data Structures and Algorithms Using Python**  
**Professor. Madhavan Mukund**  
**Analysis of Merge Sort**

(Refer Slide Time: 0:09)

**Merge sort**

 IIT Madras  
Rise. Degree.

- To sort  $A$  into  $B$ , both of length  $n$
- If  $n \leq 1$ , nothing to be done
- Otherwise
  - Sort  $A[:n//2]$  into  $L$
  - Sort  $A[n//2:]$  into  $R$
  - Merge  $L$  and  $R$  into  $B$

**Merging two sorted lists  $A$  and  $B$  into  $C$**

- If  $A$  is empty, copy  $B$  into  $C$
- If  $B$  is empty, copy  $A$  into  $C$
- Otherwise, compare first elements of  $A$  and  $B$ 
  - Move the smaller of the two to  $C$
- Repeat till all elements of  $A$  and  $B$  have been moved

Madhavan Mukund      Analysis of Merge Sort

So, let us analyze merge sort. So, Merge Sort remember was this divide and conquer algorithm, which divided the list into two halves sorted the first half sorted the second half and then merge the two halves into  $B$  and merging basically looked at the first element in each list and move to smaller one. And if any of the lists become empty, we just copy the other list. So, this is just a summary of the basic high level thing of merge sort.

So, merge sort on the left, just tells you that you sort the two halves and then merge them and the merge operations on the. So, we want to now analyze how long Merge Sort takes. So, we have to basically analyze both these things separately, we have to check how long merge takes, and then how long it takes.

(Refer Slide Time: 0:48)

## Analysing merge



- Merge  $A$  of length  $m$ ,  $B$  of length  $n$

```
def merge(A,B):  
    (m,n) = (len(A),len(B))  
    (C,i,j,k) = ([],0,0,0)  
    while k < m+n:  
        if i == m:  
            C.extend(B[j:])  
            k = k + (n-j)  
        elif j == n:  
            C.extend(A[i:])  
            k = k + (m-i)  
        elif A[i] < B[j]:  
            C.append(A[i])  
            (i,k) = (i+1,k+1)  
        else:  
            C.append(B[j])  
            (j,k) = (j+1,k+1)  
    return(C)
```



Madhavan Mukund

Analysis of Merge Sort

## Analysing merge



- Merge  $A$  of length  $m$ ,  $B$  of length  $n$
- Output list  $C$  has length  $m+n$
- In each iteration we add (at least) one element to  $C$

```
def merge(A,B):  
    (m,n) = (len(A),len(B))  
    (C,i,j,k) = ([],0,0,0)  
    while k < m+n:  
        if i == m:  
            C.extend(B[j:])  
            k = k + (n-j)  
        elif j == n:  
            C.extend(A[i:])  
            k = k + (m-i)  
        elif A[i] < B[j]:  
            C.append(A[i])  
            (i,k) = (i+1,k+1)  
        else:  
            C.append(B[j])  
            (j,k) = (j+1,k+1)  
    return(C)
```

0 2 3 4 5  
1 3 5  
2 4 5



Madhavan Mukund

Analysis of Merge Sort

So, let us try to analyze merge. So, remember that this is our merge function. So this, in this functions on where there is a table, yeah, so in this function, we basically want to merge two lists of length  $m$  and  $n$  and remember that we have this condition, which terminates the loop. So, that should give us a clue as to what the complexity of this function is going to be.

So essentially, the output list has  $m$  plus  $n$  elements, because all the elements of  $A$  and all the elements of  $B$  have to go into the list and the crucial thing is that when I go around this loop, then I make progress by moving at least one element. So, in these two cases, I move one element to  $C$  either the first element of  $A$  or the first element of  $B$  moves to  $C$ .

In some cases, I make a lot of progress. So in these two cases, I actually move a whole chunk of elements. But in the worst case, it might happen only at the end, I might be alternately moving. So if I have something like 0, 1, 2, and 3, 4, 5. So I might, or 0, 2, 4 and 1, 3, 5 then I might move as follows, I will first move the 0, then I will move the 1 then I am going to move 2, then I move the 3, then I move the 4, and then I move the 5.

So, I will only move one at a time. But in the worst case, I am not going to do any worse than that; I have to move  $m$  plus  $n$  elements. And an every time I go around this loop, I moved at least one. So, I am going to definitely finish an implicit time.

(Refer Slide Time: 2:12)

**Analysing merge**

- Merge  $A$  of length  $m$ ,  $B$  of length  $n$
- Output list  $C$  has length  $m+n$
- In each iteration we add (at least) one element to  $C$
- Hence `merge` takes time  $O(m+n)$
- Recall that  $m+n \leq 2(\max(m, n))$
- If  $m \approx n$ , `merge` take time  $O(n)$

```
def merge(A,B):
    (m,n) = (len(A),len(B))
    (C,i,j,k) = ([],0,0,0)
    while k < m+n:
        if i == m:
            C.extend(B[j:])
            k = k + (n-j)
        elif j == n:
            C.extend(A[i:])
            k = k + (m-i)
        elif A[i] < B[j]:
            C.append(A[i])
            (i,k) = (i+1,k+1)
        else:
            C.append(B[j])
            (j,k) = (j+1,k+1)
    return(C)
```

Madhavan Mukund Analysis of Merge Sort

So basically, it is easy to see that merge takes time proportional to the sum of the total number of elements to be merged  $m$  plus  $n$ . Now, in a situation like merge sort, we are applying merge in a special case where the two lists are almost the same size, because we are doing half and half. So, even if it is not exactly half, because say there is an odd number of elements or something, they are roughly the same.

So, we are looking at a situation where  $m$  is approximately equal to  $n$ . And we have seen before that if I take  $m$  plus  $n$ , it is going to be less than 2 times a maximum. So, we had seen this when we did this asymptotic complexity, we said  $f$  plus  $g$ ,  $f_1$  plus  $f_2$  will be 2 times the maximum of  $g_1, g_2$ .

So,  $m$  plus  $n$  will be 2 times the maximum, but the maximum of  $m$  and  $n$  when  $m$  and  $n$  are almost the same as just  $n$  or  $m$  whichever, because they are almost the same. So, merge will take essentially it will take time order  $n$  which makes sense.

(Refer Slide Time: 3:09)

Analysing mergesort

■ Let  $T(n)$  be the time taken for input of size  $n$   
■ For simplicity, assume  $n = 2^k$  for some  $k$

```
def mergesort(A):  
    n = len(A)  
  
    if n <= 1:  
        return(A)  
  
    L = mergesort(A[:n//2])  
    R = mergesort(A[n//2:])  
    B = merge(L,R)  
  
    return(B)
```

Madhavan Mukund Analysis of Merge Sort

So, now what about Merge Sort? Now Merge Sort is a recursive algorithm, Merge Sort of a requires me to solve my sort of half of  $A$ . So, let us first of all assume that the  $n$  that we are dealing with, because we are going to keep dividing by 2, let us assume that the  $n$  we divide, dealing by dealing with is actually a power of 2.

So, every time we divide by 2, we will come down nicely to  $n$  number and in some uniform way. So,  $T$  of  $n$  is what we want to calculate where we are assuming  $n$  is actually of the form  $2^k$ , it does not really matter, as you will see for the analysis, but it is simpler to calculate.

(Refer Slide Time: 3:47)

## Analysing mergesort



- Let  $T(n)$  be the time taken for input of size  $n$ 
  - For simplicity, assume  $n = 2^k$  for some  $k$
- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$ 
    - Solve two subproblems of size  $n/2$
    - Merge the solutions in time  $n/2 + n/2 = n$

```
def mergesort(A):  
    n = len(A)  
    if n <= 1:  
        return(A)  
    L = mergesort(A[:n//2])  
    R = mergesort(A[n//2:])  
    B = merge(L,R)  
    return(B)
```

Navigation icons

Madhavan Mukund

Analysis of Merge Sort

So, now our recurrence basically says that, if  $n$  is 0, or 1 then I return immediately because  $n$  is less than or equal to 1 return immediately. So,  $T$  of 0 and  $T$  of 1 are both 1 and otherwise, I have to do two times  $T$  of  $n$  by 2 work, because that is the cost of sorting half the array, twice the first half and the second half in binary search, we only had to do either the first half or the second half. Here we are doing both.

So, we sort both halves and then we are merging them and we saw that merging two lists of roughly the same size is basically big  $O$  of that size. So, it is going to take  $n$  steps to merge  $n$  plus 2 plus  $n$ ,  $n$  by 2 plus  $n$  by 2. So we have this is our merging cost.

(Refer Slide Time: 4:32)

## Analysing mergesort



### ■ Recurrence

- $T(0) = T(1) = 1$

- $T(n) = 2T(n/2) + n$

- $T(n) = 2T(n/2) + n$   
 $= 2[2T(n/4) + n/2] + n$

```
def mergesort(A):  
    n = len(A)  
  
    if n <= 1:  
        return(A)  
  
    L = mergesort(A[:n//2])  
    R = mergesort(A[n//2:])  
  
    B = merge(L,R)  
  
    return(B)
```



Madhavan Mukund

Analysis of Merge Sort

## Analysing mergesort



### ■ Recurrence

- $T(0) = T(1) = 1$

- $T(n) = 2T(n/2) + n$

- $T(n) = 2T(n/2) + n$   
 $= 2[2T(n/4) + n/2] + n$

```
def mergesort(A):  
    n = len(A)  
  
    if n <= 1:  
        return(A)  
  
    L = mergesort(A[:n//2])  
    R = mergesort(A[n//2:])  
  
    B = merge(L,R)  
  
    return(B)
```



Madhavan Mukund

Analysis of Merge Sort

सिद्धिर्भवति कर्मजा



## ■ Recurrence

- $T(0) = T(1) = 1$
- $T(n) = 2T(n/2) + n$

$$\begin{aligned}
 T(n) &= 2T(n/2) + n \\
 &= 2[2T(n/4) + n/2] + n \\
 &\quad \downarrow \quad \downarrow \\
 &\quad 2^2 \quad 2^2
 \end{aligned}$$

```

def mergesort(A):
    n = len(A)

    if n <= 1:
        return(A)

    L = mergesort(A[:n//2])
    R = mergesort(A[n//2:])

    B = merge(L,R)

    return(B)

```



Madhavan Mukund

Analysis of Merge Sort

So now, as usual, we unwind, so we start with this recurrence. We start with  $2T(n/2) + n$  and then we take this and we expand this is  $2T(n/4) + n/2$ , which is  $n/4$  plus  $n/2$ . So, basically, I am substituting this  $n/2$  in this function, so I am getting another factor of 2 here and this thing is getting copied here. So, I am just using the same expansion. This is what we always do for our unwinding.

So, I am replacing  $T(n/2)$  by  $2T(n/4) + n/2$ . Now I am going to do a little bit of cleaning up like we had done. When we did binary search, I am going to combine these 2s as 2 squared, and I am going to combine this and I am going to come, I am going to do some cleaning up. So, first of all, notice that there are several things happening. So, this plus this is, is 2 squared. This is again, 2 squared and this cancels with this. So, I get an  $n$  from here, and I get an  $n$  from here.



(Refer Slide Time: 5:31)

## Analysing mergesort



### ■ Recurrence

- $T(0) = T(1) = 1$
- $T(n) = 2T(n/2) + n$

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n = 2^2 T(n/2^2) + 2n \end{aligned}$$

```
def mergesort(A):  
    n = len(A)  
    if n <= 1:  
        return(A)  
    L = mergesort(A[:n//2])  
    R = mergesort(A[n//2:])  
    B = merge(L,R)  
    return(B)
```

Navigation icons

Madhavan Mukund

Analysis of Merge Sort

## Analysing mergesort



### ■ Recurrence

- $T(0) = T(1) = 1$
- $T(n) = 2T(n/2) + n$

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n = 2^2 T(n/2^2) + 2n \\ &= 2^2 [2T(n/2^3) + n/2^3] + 2n = 2^3 T(n/2^3) + 3n \end{aligned}$$

```
def mergesort(A):  
    n = len(A)  
    if n <= 1:  
        return(A)  
    L = mergesort(A[:n//2])  
    R = mergesort(A[n//2:])  
    B = merge(L,R)  
    return(B)
```

Navigation icons

Madhavan Mukund

Analysis of Merge Sort

सिद्धिर्भवति कर्मजा

## Analysing mergesort



### ■ Recurrence

- $T(0) = T(1) = 1$

- $T(n) = 2T(n/2) + n$

- $T(n) = 2T(n/2) + n$

$$= 2[2T(n/4) + n/2] + n = 2^2 T(n/2^2) + 2n$$

$$= 2^2 [2T(n/2^3) + n/2^2] + 2n = 2^3 T(n/2^3) + 3n$$

$$\vdots$$

$$= 2^k T(n/2^k) + kn$$

- When  $k = \log n$ ,  $T(n/2^k) = T(1) = 1$

- $T(n) = 2^{\log n} T(1) + (\log n)n = n + n \log n$

```
def mergesort(A):
    n = len(A)
    if n <= 1:
        return(A)
    L = mergesort(A[:n//2])
    R = mergesort(A[n//2:])
    B = merge(L,R)
    return(B)
```

Navigation icons

Madhavan Mukund

Analysis of Merge Sort

## Analysing mergesort



### ■ Recurrence

- $T(0) = T(1) = 1$

- $T(n) = 2T(n/2) + n$

- $T(n) = 2T(n/2) + n$

$$= 2[2T(n/4) + n/2] + n = 2^2 T(n/2^2) + 2n$$

$$= 2^2 [2T(n/2^3) + n/2^2] + 2n = 2^3 T(n/2^3) + 3n$$

$$\vdots$$

$$= 2^k T(n/2^k) + kn$$

- When  $k = \log n$ ,  $T(n/2^k) = T(1) = 1$

- $T(n) = 2^{\log n} T(1) + (\log n)n = n + n \log n$

- Hence  $T(n)$  is  $O(n \log n)$

```
def mergesort(A):
    n = len(A)
    if n <= 1:
        return(A)
    L = mergesort(A[:n//2])
    R = mergesort(A[n//2:])
    B = merge(L,R)
    return(B)
```

Navigation icons

Madhavan Mukund

Analysis of Merge Sort

So, if I expand this out, I claim I have 2 squared, this 2 squared times T of n by 2 squared and then because I have this 2 times n by 2 plus this n, I have 2 times n. So, now let us do it one more time just to see what happens. So, I take this and I expand it, and I get 2 times T of n by one more power of 2 at the bottom, so 2 times n by T cubed, plus n by 2 squared n by 2 squared and then I have this 2 n.

Now again, I do this calculation. So, this 2 squared, 2 squared cancels, I get one more n, so I get 3 n, and then this 2 cube, and this 2 cube comes here. So, you can now see a pattern after two expansions, I get to 2 squared, T of n by 2 squared plus 2 and after three expansions, I get to cube T of n by 2 cubed plus 3 n.

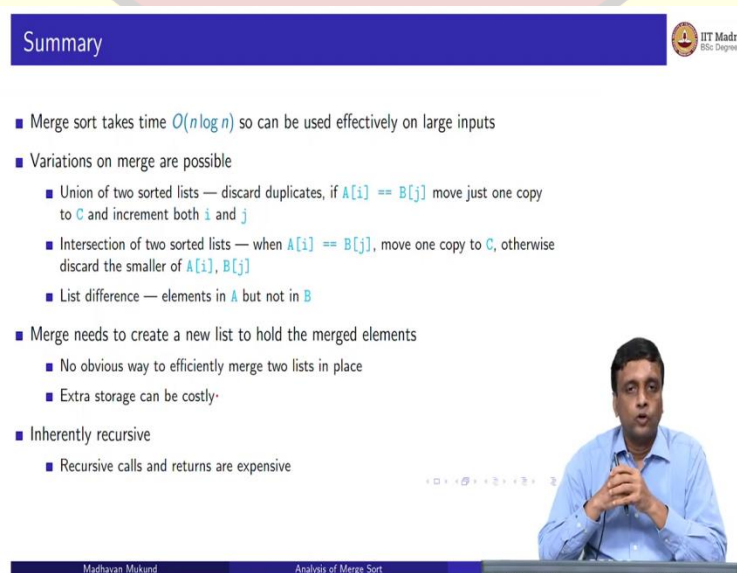
So, you can work out that after  $k$  steps, I will get  $2$  to the power  $k$ ,  $T$  of  $n$  by  $2$  to the power  $k$ . So, this is just saying that I am dividing that interval by  $2$   $k$  times and  $k$  times  $n$ . Now, when do we kind of reach this base case when this becomes  $1$ ? So as usual, we look at the case where  $k$  is  $\log n$ .

So, if  $k$  is  $\log n$ , then  $n$  by  $2$  to the  $k$  becomes  $1$ . So,  $T$  of  $n$  by  $2$  to the case  $T$  of  $1$  is  $1$ . So at this value of  $k$ , I get  $T$  of  $n$  is  $2$  to the power  $k$ , so that is  $2$  to the power  $n \log n$ , times  $T$  of  $1$ , which is one and then this is again  $k$ . So, we are looking at this expression. And we are plugging in  $k$  is equal to  $\log n$ . So, for  $k$  is equal to  $\log n$ , I am getting this  $\log n$ . With this  $K$ , I am getting this  $\log n$ .

And for this  $n$  by  $2$  to the  $k$ , I am getting  $1$ . So, this is just  $1$ ,  $2$  to the  $\log n$  is just  $n$ ,  $2$  to the, you can just check that for yourself, but  $2$  to the  $\log n$  is just  $n$ . So, this whole thing on the left hand side simplifies to  $n$ , and this right hand side becomes  $n \log n$ . And because  $n \log n$  is bigger than  $n$ , I can throw away the  $n$  term and say that this is big  $O$  of  $n \log n$ .

So actually, merge sort is big  $O$  of  $n \log n$ , which is really what we want. Because we said that finally, when we are using that SIM card thing we need to do  $n \log n$  work to search. So, if the sorting takes more than  $n \log n$  time, we are in bad, bad shape. But now we are saying that Merge Sort will actually take  $n \log n$  time.

(Refer Slide Time: 7:54)



**Summary**

- Merge sort takes time  $O(n \log n)$  so can be used effectively on large inputs
- Variations on merge are possible
  - Union of two sorted lists — discard duplicates, if  $A[i] == B[j]$  move just one copy to  $C$  and increment both  $i$  and  $j$
  - Intersection of two sorted lists — when  $A[i] == B[j]$ , move one copy to  $C$ , otherwise discard the smaller of  $A[i], B[j]$
  - List difference — elements in  $A$  but not in  $B$
- Merge needs to create a new list to hold the merged elements
  - No obvious way to efficiently merge two lists in place
  - Extra storage can be costly
- Inherently recursive
  - Recursive calls and returns are expensive

Madhavan Mukund Analysis of Merge Sort

So, this takes  $n \log n$  time and I claimed without justifying it, you can look it up if you are interested that you cannot do better than in  $\log n$  and such kind of sorting, where you are comparing and exchanging. So, it can be used effectively on large inputs. Another thing is that that merge function that we saw here actually can be used in a number of contexts.

So one thing is, if you take two lists of values, without duplicates, you can take the union and remove duplicates. So basically, whenever I move from A and B to C, if I see the same value at A and B I keep only one copy, but I move both the pointers. Similarly, you can do intersection, if I see the same value, I move it, if I do not see the same value, I skip it.

So, the same merge function, there are lots of variations. And they can do very interesting things with two sorted lists. You can also do the list difference, everything which is in A but which is not in B, and so on. One of the drawbacks of this thing is that merge needs to create a new list. There is no obvious way to put those lists back into the list that you started with. Remember that an insertion and selection, we had a clever way of moving the things to the beginning so that we did not have to create a new list.

But in merge, there is no obvious way to actually put it back because you have no idea really where it is going to come. So, you cannot really take and reuse space and A and B to store the merge of A and B. You do not know which of them is going to move faster compared to the other. So, you have no option but to create extra space.

And this extra space is an extra resource that we have to use. And the other thing is that this Merge Sort there is no way to avoid this recursive call. I mean, we saw for Insertion Sort, for example, that we could do Insertion Sort with recursion without recursion. And sometimes it is easier to do one than the other. But merge sort is kind of inherently recursive because there is no easy way to describe Merge Sort without recursion. So, we will try to address some of these issues by looking at yet another sorting algorithm.