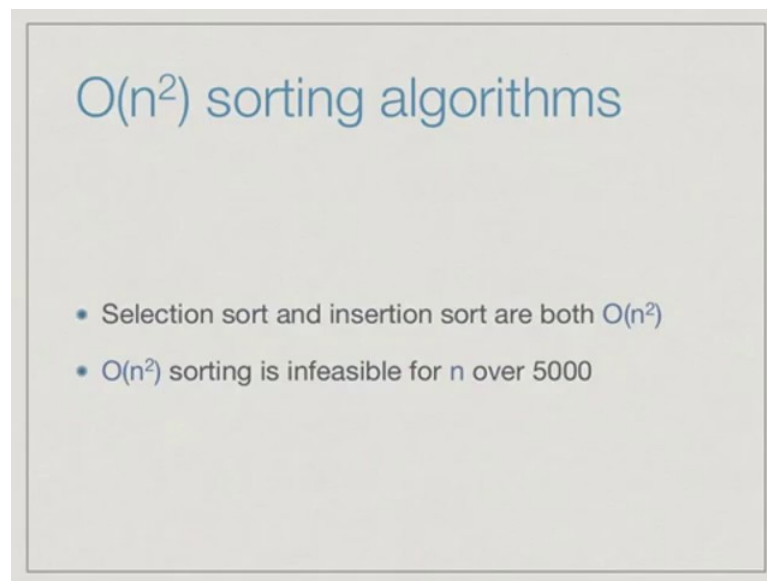


Programming, Data Structures and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 04
Lecture - 01
Merge Sort

(Refer Slide Time: 00:02)

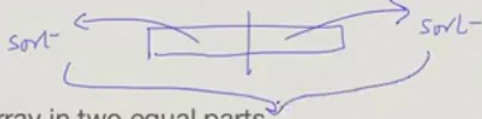


Last week, we saw two simple sorting algorithms, selection sort and insertion sort. These were attractive, because they corresponded to the manual way in which we would sort items by hand.

On the other hand, we analyzed these to see that the worst case complexity is order n^2 where n is the length of the input list to be sorted. And unfortunately, n^2 sorting algorithms are infeasible for n over about 5000, because it will just take too long and on the other hand, 5000 is the rather small number when we are dealing with real data.

(Refer Slide Time: 00:37)

A different strategy?




- Divide array in two equal parts
- Separately sort left and right half
- Combine the two sorted halves to get the full array sorted

Let us examine a different strategy all together. Suppose we had the example where you **were** teaching assistant and you were supposed to sort the answer papers for the instructor and supposing the instructor had not one teaching assistant, but two teaching assistants. And the job is distributed to the two teaching assistants, so each one is told to go with halves the papers, sort them separately and come back and then the instructor has to put these two lists together.

In other words, you divide the array initially, the unsorted array or list into two parts and then you hand over these two parts to two different people or two different programs if you want to sort. So, you sort these two halves separately and now the key is to be able to combine these two sorted things efficiently **in a** single sorted list.

(Refer Slide Time: 01:27)

Combining sorted lists



- Given two sorted lists A and B, combine into a sorted list C
- Compare first element of A and B
- Move it into C
- Repeat until all elements in A and B are over
- Merging A and B

Let us focus on the last part, how we combine two sorted lists into a single sorted list. Now this is again something that you would do quite naturally. Supposing you have the two outputs from the two teaching assistants then what you would do is you would examine of course, the top paper in both. Now, this top paper on the left hand side is the highest mark on the left hand side. The top paper on the right hand side is the highest mark on the right hand side. The maximum among these two is a top overall. So, you could take the maximum say this one and move it aside.

Now you have the second highest on the right hand side and the first the highest on the left hand side. Again, you look at the bigger one and move that one here and so on. So, at each time, you look at the current head or top of each of the lists and move the bigger one to the output, right. And if you keep repeating this until all the elements are over, you will have merged them preserving the sorted order overall.

(Refer Slide Time: 02:24)



Let us examine how this will work in a simple example here. So, we have two sorted lists 32, 74, 89 and 21, 55, 64. So, we start from the left and we examine these two elements initially and pick the smaller of the two because we are sorting in ascending order. So, we pick the smaller of the two that is 21 and now the second list has reduced the two elements.

At the next step, we will examine the first element in the first list and the second element in the second list because that is what is left. Among these two 32 is smaller, so we will move 32. Now 55 is the smaller of the two at the end, now 64 is the smaller of the two at the end. Notice we have reached the situation where the second list is empty, so since the second list is empty we can just copy the first list as it is without having to compare anything because those are all the remaining elements that need to be merged.

(Refer Slide Time: 03:23)

Merge Sort

- Sort $A[0:n//2]$ *Left*
- Sort $A[n//2:n]$ *Right*
- Merge sorted halves into $B[0:n]$
- How do we sort the halves?
 - Recursively, using the same strategy!

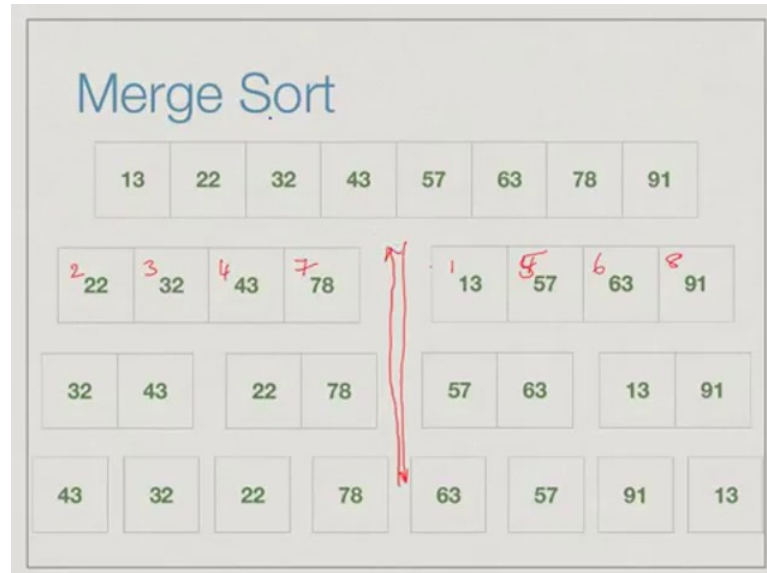
Having done this, now that we have a procedure to merge two-sorted list into a single-sorted list; we can now clarify exactly what we mean by merging the things using merge sort. So, merge sort is this algorithm which divides the list into two parts and then combines the sorted halves into a single part. So, what we do is we first sort the left hand side. So, we take the positions from 0 to n by 2 minus 1 where n is the length. This is left and this is the right.

Now one thing to note is in python notation, we use the same subscript here and here, because this takes us to the position n by 2 minus 1 and this will start at n by 2. So, we will not miss anything nor will we duplicate anything, so it is very convenient. This is another reason why python has its convention that the right hand side of a slice goes up to the slice minus 1.

If we write something like this we do not have to worry about whether we have to do plus 1, minus 1, we can just duplicate the index of the right hand side and the left hand side and it will correctly span the entire list. So, what we do is this is a naturally recursive algorithm; we recursively use this algorithm to sort the first half and the second half and then we merge these two sorted halves into the output. The important thing is we keep repeatedly doing the same thing; we keep **halving, sort** the half, sort the other half and merge and when do we reach a base case where when we reach a list which has

only one element or zero elements there is nothing to sort. When such a situation, we can just return the list as it is and then rely on merging to go ahead.

(Refer Slide Time: 04:58)



Once again let us look at an illustrative example. Supposing, we have eight items to sort which are **organized like** this. The first step is dividing it into two and sort each separately. So, we divide it into two groups; we have the left half and the right half. Now these are still things, which we do not know how to sort directly, so again we divide into two. The left half gets divided into two further subdivisions and so does the right.

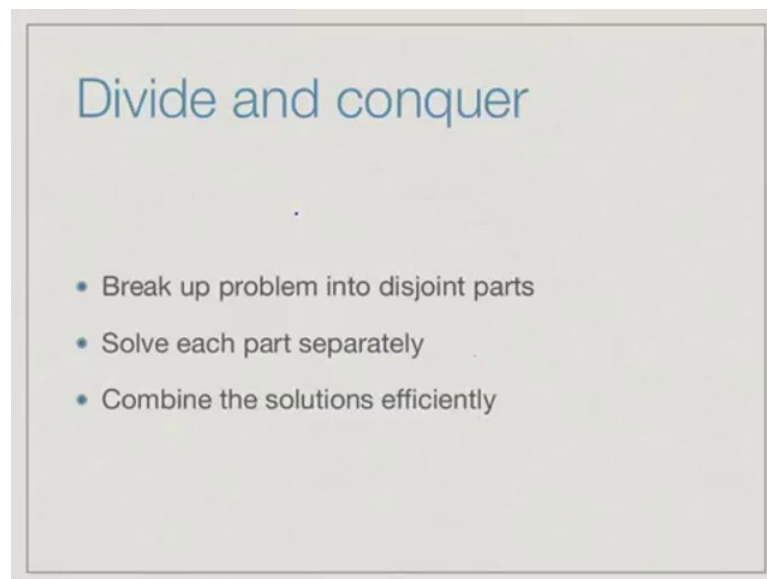
Now we have list of length two, we could sort them by hand, but we say that we do not know how to sort anything except a list of length 1 or 0, so we further break it up. Now, we have trivial lists 43 and 32 on the left, 22 78 and so on, so we end up with eight lists of length one which are automatically sorted.

At this stage, the recursion comes back and says, you have sorted the sub list 43 for example, this list we have sorted the left into 43 and the right into 32 in a trivial way, so we need to combine them **by** merging. So, we merge 43 and 32 **by** applying a merge procedure and we get 32 before 43 when we merge 22 and 78 they remain in the same order. Here 57 come before 63 and finally, 13 comes before 91.

Now, at this level, we have two lists of lengths two which are sorted and so they must be merged and similarly here we have two lists of lengths two which are sorted and they

must be merged. So, we merge the first pair, we get 22 followed by 32, followed by 43, followed by 78. And similarly, here we get 13 followed by 57 followed by 63 followed by 91. So, after doing these two merges we have now two lists of length four each of which are sorted. And now we will end up picking from this 13 and then so this is 13 and then we will pick 22 then we pick 32 and then we pick 43 then 57, then 63, then 78 and then 91 right. This is how this recursion goes, you first keep breaking it up, down till the base case and then you keep combining backwards using the merge.

(Refer Slide Time: 07:08)

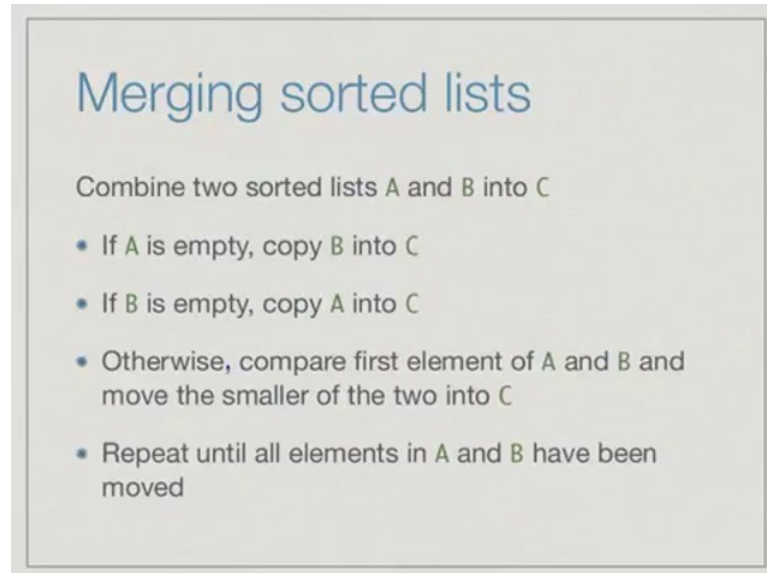


This strategy that we **just** outlined from merge sort is a general paradigm called divide and conquer. So, if you have a problem where you can break the problem up into sub problems, which do not have any interference **with** each other. So, here for instance, sorting the first half of the list and sorting the second half of the list can be done independently. You can take the papers assigned by the instructor, give them to two separate teaching assistants, ask them to go to two separate rooms; they do not need to communicate with each other to finish sorting their halves.

In such a situation, you break up the problem into independent sub problems and then you have efficient way to combine the solved sub problems. So that is the key **there**, how efficiently you can combine the problems. If you takes you a very long time to combine the problems then it is not **going** to help you at all. But, if you can do it in a simple way like this merge sort where we do the merging by just scanning the two lists from

beginning to end and assigning each one of them to the final thing as we see it, then you can actually derive a lot of benefit from **divide and conquer**.

(Refer Slide Time: 08:08)



Merging sorted lists

Combine 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 element of A and B and move the smaller of the two into C
- Repeat until all elements in A and B have been moved

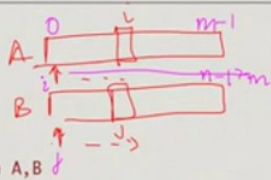
Let us look a little more in detail at the actual algorithmic aspect of how we are going to do this. First, since we looked at merging as the basic procedure, how do we merge two sorted list. As a base case, if either of them is empty as we saw in the example, we do not need to do anything; we just copy the other one. So, we are taking two input lists A and B, which are both sorted and we are trying to return a sorted list C.

So, if A is empty, we just copy B into C; if B is empty, we just copy A into C. Otherwise, what we do is **if** both are not empty, so we want to take the smaller one of the head of A and the head of B and move that to C, because that will be the smallest one overall in what is remaining. So, we compare the first element of A and B and we move the smaller one into C and we keep repeating this until all the elements in A and B has been moved.

(Refer Slide Time: 09:05)

Merging

```
def merge(A,B): # Merge A[0:m],B[0:n]
    (m,n) = (len(A),len(B))
    (i,j) = (0,0) # Current positions in A,B
    while i+j < m+n: # i+j is number of elements merged so far
        if i == m: # Case 1: A is empty
            C.append(B[j])
            j = j+1
        elif j == n: # Case 2: B is empty
            C.append(A[i])
            i = i+1
        elif A[i] <= B[j]: # Case 3: Head of A is smaller
            C.append(A[i])
            i = i+1
        elif A[i] > B[j]: # Case 4: Head of B is smaller
            C.append(B[j])
            j = j+1
    return(C)
```



This is a python implementation of this merge function in general the two lists need not be the same length. So, we are merging A of length m and B of length n into an output list of length C. Initially, we set up m and n, because we need to keep track of how many elements we have moved in order to decide when to terminate. So, we set up m and n to point to the lengths of A and B respectively and we initialize the output list to be the empty list, because remember that in python the type of C will only be known after it is assigned a value. So, we need to tell it that initially the output merge list is an empty list, so that we can then use append to keep adding items to it.

Now what we are going to do is essentially start from the left hand side of both A and B, so we are going to start here and walk to the right; as we go along we are going to move one of the two elements. So, we need an index to point here, so we use the index i and j to point into A and B respectively.

And initially, these indices point to the starting element which is 0. So, as we move along if we move i from 0 to 1 that means, we have processed one element in A; if we move it to 3; we have processed three elements in A and so on. So, at any given time i plus j will tell us how many elements have been moved so far to the output; eventually, everything in A and everything in B must be moved to the output. This will go on so long as i plus j is not reached the total m plus n which was the total number of elements we had to move to begin with.

While $i + j$ is less than $m + n$, we have to look at different cases. The first two cases are where one of the two **lists** is empty; either we have reached the end of A, so i has actually reached m . So, remember the indices go from 0 to $m - 1$ and 1 to $n - 1$. So, if i has actually gone to m ; that means, that we have exhausted the elements in A. So, we append the next element in B and we keep going by incrementing the pointer in B or the index in B.

Similarly, if we have reached the end of B, we append the next element A and we go back. Now remember that at this point because $i + j$ is less than $m + n$, if we have finished m elements, but $m + n$ is not been reached, there must be some element in B. Similarly, if we have finished n elements in B, but $i + j$ is not **yet** $m + n$, there must be at least one element left in it. These two things will definitely work just by checking the fact that we have not finished all the elements, but one of the lists is exhausted.

Now, if neither list is exhausted then we have to do a comparison. So, we come to this case and we check whether the element in A is smaller than or equal to the element in B. So, at this point we are in general looking at some $A[i]$ and some $B[j]$. So, we have to decide which of these two goes into C next. The smaller of the two **if it comes in A**, we append that to C at the increment i pointer; otherwise, we append the B value increment the j pointer. At the end of this loop, what we would have done is to have transferred $m + n$ elements in the correct order from A and B into C.

(Refer Slide Time: 12:29)

```
madhavan@dolphinair:...ul/week4/python/mergesort$ more merge.py
def merge(A,B): # Merge A[0:m],B[0:n]

    (C,m,n) = ([],len(A),len(B))
    (i,j) = (0,0) # Current positions in A,B

    while i+j < m+n: # i+j is number of elements merged so far

        if j == n: # Case 1: A is empty
            C.append(A[i])
            i = i+1
        elif i == m: # Case 2: B is empty
            C.append(B[j])
            j = j+1
        elif A[i] <= B[j]: # Case 3: Head of A is smaller
            C.append(A[i])
            i = i+1
        elif A[i] > B[j]: # Case 4: Head of B is smaller
            C.append(B[j])
            j = j+1

    return(C)

madhavan@dolphinair:...ul/week4/python/mergesort$ python3.5
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Let us just verify that this code that we have described works in python as we expect. So, here is a file merge dot **py** in which we had exactly the same code as we had on the slide. So, you can check that the code is exactly the same it goes through this loop while i plus j is less than m plus n and it checks the four cases and according to that copy is either in element from A to C, or B to C and finally returns **the list C**.

(Refer Slide Time: 13:00)

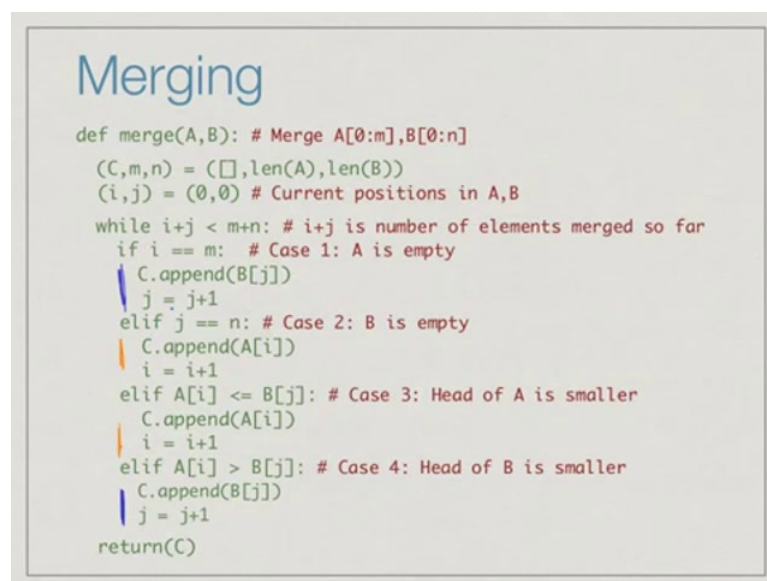
```
>>> from merge import *
>>> a = list(range(0,100,2))
>>> b = list(range(1,75,2))
>>> len(a)
50
>>> len(b)
37
>>> a
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98]
>>> b
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73]
>>> merge(a,b)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98]
>>> len(merge(a,b))
87
>>> 
```

The simplest way to do this is to try and construct two lists; suppose, we take a list of numbers where the even numbers from 0 to 100. So, we start at 0 and go to 100 in steps

of 2. And we might take the odd numbers from say 1 to 75, so we do not have the same length here right. The length of a is 50, the length of b is 37. And a has it's in ascending order 0 to 98 in steps of 2; B is 1 to 73 in steps of 1 and in steps of 2 again.

Now if we say merge a, b, and then we get something which actually returns this merge list. Notice that up to 73, which is the last element in b, we get all the numbers. And then from here, we get only the even numbers because we **are** only copying from a. And if you want to check the length of the merge list then it is correctly 37 plus 50 is 87.

(Refer Slide Time: 14:12)



```
def merge(A,B): # Merge A[0:m],B[0:n]
    (C,m,n) = ([],len(A),len(B))
    (i,j) = (0,0) # Current positions in A,B
    while i+j < m+n: # i+j is number of elements merged so far
        if i == m: # Case 1: A is empty
            C.append(B[j])
            j = j+1
        elif j == n: # Case 2: B is empty
            C.append(A[i])
            i = i+1
        elif A[i] <= B[j]: # Case 3: Head of A is smaller
            C.append(A[i])
            i = i+1
        elif A[i] > B[j]: # Case 4: Head of B is smaller
            C.append(B[j])
            j = j+1
    return(C)
```

If we go back and look at this code again, then it appears as though we have duplicated the code in a couple of places. So, we have two situations case 1 and case 4 where we **are** appending the element from B into C and we are incrementing j. And similarly, we have two different situations - case 2 and case 3, where we are appending the element from A into C and then incrementing i. So, it is tempting to argue that we **would** have a more compact version of this algorithm, if we combine these cases.

(Refer Slide Time: 14:57)

Merging, wrong

```
def mergewrong(A,B): # Merge A[0:m],B[0:n]
    (C,m,n) = ([],len(A),len(B))
    (i,j) = (0,0) # Current positions in A,B
    while i+j < m+n:
        # i+j is number of elements merged so far
        # Combine Case 1, Case 4
        if i == m or A[i] > B[j]:
            C.append(B[j])
            j = j+1
        # Combine Case 2, Case 3:
        elif j == n or A[i] <= B[j]:
            C.append(A[i])
            i = i+1
    return(C)
```

If we combine these cases then we can combine case 1 and 4. Remember one and four are the ones where we take the value from B. So, we combine 1 and 4 and say either if A is empty or if B has a smaller value then you take the value from B and append it to C and say j equal to j plus 1.

On the other hand, either if B is empty or A has a smaller value then you take the value from A and append the index in that right. Let us see what happens if we try to run this code.

(Refer Slide Time: 15:20)

```
madhavan@dolphinair:...ul/week4/python/mergesort$ more mergewrong.py
def merge(A,B): # Merge A[0:m],B[0:n]

    (C,m,n) = ([],len(A),len(B))
    (i,j) = (0,0) # Current positions in A,B

    while i+j < m+n: # i+j is number of elements merged so far

        if i == m or A[i] > B[j]: # Combine Case 1 and 4
            C.append(B[j])
            j = j+1
        elif j == n or A[i] <= B[j]: # Combine Case 2 and 3
            C.append(A[i])
            i = i+1

    return(C)
madhavan@dolphinair:...ul/week4/python/mergesort$ python3.5
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from mergewrong import
```

Here we have a file merge wrong dot py, the name suggests that is going to be a problem, where we have combined case 1 and 4, where we append the element from B into C and 2 and 3 where we combine the element append the element from A into C. Let us run this and see. Now, we take merge wrong at the starting point.

(Refer Slide Time: 15:46)

```
>>> a = [2,4,6]
>>> b = [1,3,5]
>>> merge(a,b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/madhavan/mirror/projects/NPTEL/python-2016-jul/week4/python/mergesort/mergewrong.py", line 8, in merge
    if i == m or A[i] > B[j]: # Combine Case 1 and 4
IndexError: list index out of range
>>>
```

And let us just do a simple case. Supposing we take a as 2, 4, 6 and b as 1, 3, 5 then we have to expect 1, 2, 3, 4, 5, 6. Let us try to merge a and b right and now we get an error which says that we have a list index out of range. List index out of range suggests that we are trying to access some element which is not present and it so happens that this is in the case if i equal to m or a i equal to b j, greater than b j. Let us see if we can diagnose what is going on.

(Refer Slide Time: 16:25)

```
madhavan@dolphinair:...ul/week4/python/mergesort$ more mergewrong.py
def merge(A,B): # Merge A[0:m],B[0:n]

    (C,m,n) = ([],len(A),len(B))
    (i,j) = (0,0) # Current positions in A,B

    while i+j < m+n: # i+j is number of elements merged so far

        print(m,n,i,j)

        if i == m or A[i] > B[j]: # Combine Case 1 and 4
            C.append(B[j])
            j = j+1
        elif j == n or A[i] <= B[j]: # Combine Case 2 and 3
            C.append(A[i])
            i = i+1

    return(C)
madhavan@dolphinair:...ul/week4/python/mergesort$ python3.5
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from mergewrong import *
```

One simple way of diagnosing what is going on is to just insert some statements to print out the values of the names at some appropriate point. Now here since we are having an error at inside the while loop, what we have done is we have added the statement print which as I said we have not formally seen we will see it in the next week. But it does **the intuitive** thing it take the names m, n, i and j and prints them out in that sequence on the screen, so that we can see what is happening. Let us now run this again. So, we run the interpreter load this updated version of merge wrong.

(Refer Slide Time: 17:03)

```
>>> a = [2,4,6]
>>> b = [1,3,5]
>>> merge(a,b)
3 3 0 0
3 3 0 1
3 3 1 1
3 3 1 2
3 3 2 2
3 3 2 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "/Users/madhavan/mirror/projects/NPTEL/python-2016-jul/week4/python/mergesort/mergewrong.py", line 10, in merge
        if i == m or A[i] > B[j]: # Combine Case 1 and 4
    IndexError: list index out of range
>>>
```

Setup a and b as before. So, a is 2, 4, 6; b is 1 3 5. And now we run merge. And now we see what is happening. So, m and n are the initial lengths 3 and 3. And these are the values 0 and 0 are i and j the pointers. So, j becomes 1 then i become 1 and so on.

At this point, this is where the problem is right. What we have found is that if i is equal to n or a i greater than b j, so i is not equal to m. So, i is not yet 3, so then it is trying to check whether a i is bigger than b j, but at this point unfortunately j is n. If we had had these cases in order we would have first checked if i is 3 then if j is 3 and only if neither of them are 3 would be a try to compare them. Because now we are only checking if i is 3, since i is not 3 we are going at and checking the value at a i against b j, but unfortunately j has become 3 already and we have not checked it yet.

By combining these two cases, we have allowed a situation where we are trying to compare a i and b j where one of them is a valid index and the other is not a valid index. Although it looks tempting to combine these two cases one has to be careful when doing so especially when we have these boundary conditions when we are indexing list, we must make sure that the index we are trying to get to is a valid index. And sometimes it is implicit and sometimes we have to be careful and this is one of those cases where you have to be careful and not optimize these things.

Otherwise, we have to have a separate condition saying if i is equal to m or j is less than n and which becomes more complicated than the version we had with four explicit cases. So, we may as well go back to the version with four explicit cases.

(Refer Slide Time: 18:52)

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 that we have seen how to merge the list, let us sort them. So, what we want to do is take a list of elements A and sort it into an output list B . So, if n is 1 or 0 actually, so if it is empty or it has got length 1, we have nothing to do; otherwise, we will sort the first half into a new list L and sort the second half into a new list R . L for left and R for right. And then we will apply the earlier merge function to obtain the output list B .

(Refer Slide Time: 19:31)

Merge Sort

```
def mergesort(A, left, right):  
    # Sort the slice A[left:right]  
    if right - left <= 1: # Base case  
        return(A[left:right])  
    if right - left > 1: # Recursive call  
        mid = (left+right)//2  
        L = mergesort(A, left, mid)  
        R = mergesort(A, mid, right)  
        return(merge(L, R))
```



This is a very simple function except that we are going to be sorting different segments or slices of our list. So, we will actually have merge sort within input list and the left and

right endpoints of the slice to be sorted. If the slice of length 1 or 0 then we just return the slice as it is. It is important that we have to return that part of the slice and not the entire part A, because they are only sorting.

Remember when we broke up something into two parts, for example, right, so then at this point we have to return the sorted version of this slice, not the entire slice. So, we have to return A from left to right if it is a base case; otherwise, we find the midpoint then we sort recursively sort the portion from the left hand side of the current slice to the midpoint, we put it in L. Then we take the midpoint to the right, put it in R and we use our earlier function merge to get a sorted list out of these two parts L and R and return this. This is a very straight forward implementation; there are no tricks or pitfalls here.

The only thing to remember is that we have to augment our merge sort function with these two things, the left point and the right point. We had a similar thing if you remember for binary search, where we recursively kept having the interval to search. So, we have to keep telling it in which interval we are searching.

(Refer Slide Time: 20:53)

```
def merge(A,B): # Merge A[0:m],B[0:n]

    (C,m,n) = ([],len(A),len(B))
    (i,j) = (0,0) # Current positions in A,B

    while i+j < m+n: # i+j is number of elements merged so far

        if i == m: # Case 1: A is empty
            C.append(B[j])
            j = j+1
        elif j == n: # Case 2: B is empty
            C.append(A[i])
            i = i+1
        elif A[i] <= B[j]: # Case 3: Head of A is smaller
            C.append(A[i])
            i = i+1
        elif A[i] > B[j]: # Case 4: Head of B is smaller
            C.append(B[j])
            j = j+1

    return(C)

def mergesort(A,left,right): # Sort the slice A[left:right]

    if right - left <= 1: # Base case
        return(A[left:right])

mergesort.py
```

Let us look at a python implementation of merge sort. So, here we have a file merge sort dot py. We start with the function merge, which we saw before with a four way case split in order to shift elements from A and B to C.

(Refer Slide Time: 21:06)

```
    if i == m: # Case 1: A is empty
        C.append(B[j])
        j = j+1
    elif j == n: # Case 2: B is empty
        C.append(A[i])
        i = i+1
    elif A[i] <= B[j]: # Case 3: Head of A is smaller
        C.append(A[i])
        i = i+1
    elif A[i] > B[j]: # Case 4: Head of B is smaller
        C.append(B[j])
        j = j+1

    return(C)

def mergesort(A,left,right): # Sort the slice A[left:right]

    if right - left <= 1: # Base case
        return(A[left:right])

    if right - left > 1: # Recursive call
        mid = (left+right)//2
        L = mergesort(A,left,mid)
        R = mergesort(A,mid,right)
        return(merge(L,R))
madhavan@dolphinair:...ul/week4/python/mergesort$
```

And then we add at the bottom of the file, the new function merge sort which we saw on the previous slide which takes a slice of A from left to right and sorts it. If it is a trivial slice, it returns the slice as it is. Otherwise, it breaks it into two parts and recursively sorts that. Let us see how this would actually run.

(Refer Slide Time: 21:25)

```
>>> from mergesort import *
>>> a = list(range(1,100,2)) + list(range(0,100,2))
>>> a
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45,
 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87,
 89, 91, 93, 95, 97, 99, 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76,
 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98]
>>> mergesort(a,0,len(a))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66,
 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88,
 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
>>> a = list(range(1,1000,2)) + list(range(0,1000,2))
>>> mergesort(a,0,len(a))
```

We take python and we say from merge sort, import all the functions. And now let us take a larger range. Suppose, if we take all the odd numbers followed by all the even numbers. So, we say range 1 to 100 in steps of 2. Those are the odd numbers and then all

the even numbers in the same range say. So, A has now odd numbers followed by even numbers. You would imagine that if I sort this from 0 to the length of A, then you get the numbers sorted in **sequence**.

(Refer Slide Time: 22:12)

```
538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554,
555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 5
72, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 58
9, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606
, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623,
624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640,
641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 6
58, 659, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 67
5, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692
, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709,
710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726,
727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 7
44, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 76
1, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778
, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795,
796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812,
813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 8
30, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 84
7, 848, 849, 850, 851, 852, 853, 854, 855, 856, 857, 858, 859, 860, 861, 862, 863, 864
, 865, 866, 867, 868, 869, 870, 871, 872, 873, 874, 875, 876, 877, 878, 879, 880, 881,
882, 883, 884, 885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 896, 897, 898,
899, 900, 901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915, 9
16, 917, 918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 93
3, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950
, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967,
968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984,
985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999]
```

Now what if I take a larger list 1000 then I get, again everything sorted.

(Refer Slide Time: 22:16)

```
>>> a = list(range(1,10000,2)) + list(range(0,10000,2))
>>> mergesort(a,0,len(a))
```

Now **our** claim is that this is an order $n \log n$ algorithm. It should work well for even bigger list. If I say 10000 which remember would take a very long time with insertion sort or selection sort.

(Refer Slide Time: 22:26)

```
616, 9617, 9618, 9619, 9620, 9621, 9622, 9623, 9624, 9625, 9626, 9627, 9628, 9629, 9630, 9631, 9632, 9633, 9634, 9635, 9636, 9637, 9638, 9639, 9640, 9641, 9642, 9643, 9644, 9645, 9646, 9647, 9648, 9649, 9650, 9651, 9652, 9653, 9654, 9655, 9656, 9657, 9658, 9659, 9660, 9661, 9662, 9663, 9664, 9665, 9666, 9667, 9668, 9669, 9670, 9671, 9672, 9673, 9674, 9675, 9676, 9677, 9678, 9679, 9680, 9681, 9682, 9683, 9684, 9685, 9686, 9687, 9688, 9689, 9690, 9691, 9692, 9693, 9694, 9695, 9696, 9697, 9698, 9699, 9700, 9701, 9702, 9703, 9704, 9705, 9706, 9707, 9708, 9709, 9710, 9711, 9712, 9713, 9714, 9715, 9716, 9717, 9718, 9719, 9720, 9721, 9722, 9723, 9724, 9725, 9726, 9727, 9728, 9729, 9730, 9731, 9732, 9733, 9734, 9735, 9736, 9737, 9738, 9739, 9740, 9741, 9742, 9743, 9744, 9745, 9746, 9747, 9748, 9749, 9750, 9751, 9752, 9753, 9754, 9755, 9756, 9757, 9758, 9759, 9760, 9761, 9762, 9763, 9764, 9765, 9766, 9767, 9768, 9769, 9770, 9771, 9772, 9773, 9774, 9775, 9776, 9777, 9778, 9779, 9780, 9781, 9782, 9783, 9784, 9785, 9786, 9787, 9788, 9789, 9790, 9791, 9792, 9793, 9794, 9795, 9796, 9797, 9798, 9799, 9800, 9801, 9802, 9803, 9804, 9805, 9806, 9807, 9808, 9809, 9810, 9811, 9812, 9813, 9814, 9815, 9816, 9817, 9818, 9819, 9820, 9821, 9822, 9823, 9824, 9825, 9826, 9827, 9828, 9829, 9830, 9831, 9832, 9833, 9834, 9835, 9836, 9837, 9838, 9839, 9840, 9841, 9842, 9843, 9844, 9845, 9846, 9847, 9848, 9849, 9850, 9851, 9852, 9853, 9854, 9855, 9856, 9857, 9858, 9859, 9860, 9861, 9862, 9863, 9864, 9865, 9866, 9867, 9868, 9869, 9870, 9871, 9872, 9873, 9874, 9875, 9876, 9877, 9878, 9879, 9880, 9881, 9882, 9883, 9884, 9885, 9886, 9887, 9888, 9889, 9890, 9891, 9892, 9893, 9894, 9895, 9896, 9897, 9898, 9899, 9900, 9901, 9902, 9903, 9904, 9905, 9906, 9907, 9908, 9909, 9910, 9911, 9912, 9913, 9914, 9915, 9916, 9917, 9918, 9919, 9920, 9921, 9922, 9923, 9924, 9925, 9926, 9927, 9928, 9929, 9930, 9931, 9932, 9933, 9934, 9935, 9936, 9937, 9938, 9939, 9940, 9941, 9942, 9943, 9944, 9945, 9946, 9947, 9948, 9949, 9950, 9951, 9952, 9953, 9954, 9955, 9956, 9957, 9958, 9959, 9960, 9961, 9962, 9963, 9964, 9965, 9966, 9967, 9968, 9969, 9970, 9971, 9972, 9973, 9974, 9975, 9976, 9977, 9978, 9979, 9980, 9981, 9982, 9983, 9984, 9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
```

Question is how long it takes here and it comes out quite fast.

(Refer Slide Time: 22:29)

```
>>> a = list(range(1,100000,2)) + list(range(0,100000,2))
>>> mergesort(a,0,len(a))
```

We can go further and say 100000 for example, and even here it comes reasonably fast.

(Refer Slide Time: 22:37)

```
99676, 99677, 99678, 99679, 99680, 99681, 99682, 99683, 99684, 99685, 99686, 99687, 99688, 99689, 99690, 99691, 99692, 99693, 99694, 99695, 99696, 99697, 99698, 99699, 99700, 99701, 99702, 99703, 99704, 99705, 99706, 99707, 99708, 99709, 99710, 99711, 99712, 99713, 99714, 99715, 99716, 99717, 99718, 99719, 99720, 99721, 99722, 99723, 99724, 99725, 99726, 99727, 99728, 99729, 99730, 99731, 99732, 99733, 99734, 99735, 99736, 99737, 99738, 99739, 99740, 99741, 99742, 99743, 99744, 99745, 99746, 99747, 99748, 99749, 99750, 99751, 99752, 99753, 99754, 99755, 99756, 99757, 99758, 99759, 99760, 99761, 99762, 99763, 99764, 99765, 99766, 99767, 99768, 99769, 99770, 99771, 99772, 99773, 99774, 99775, 99776, 99777, 99778, 99779, 99780, 99781, 99782, 99783, 99784, 99785, 99786, 99787, 99788, 99789, 99790, 99791, 99792, 99793, 99794, 99795, 99796, 99797, 99798, 99799, 99800, 99801, 99802, 99803, 99804, 99805, 99806, 99807, 99808, 99809, 99810, 99811, 99812, 99813, 99814, 99815, 99816, 99817, 99818, 99819, 99820, 99821, 99822, 99823, 99824, 99825, 99826, 99827, 99828, 99829, 99830, 99831, 99832, 99833, 99834, 99835, 99836, 99837, 99838, 99839, 99840, 99841, 99842, 99843, 99844, 99845, 99846, 99847, 99848, 99849, 99850, 99851, 99852, 99853, 99854, 99855, 99856, 99857, 99858, 99859, 99860, 99861, 99862, 99863, 99864, 99865, 99866, 99867, 99868, 99869, 99870, 99871, 99872, 99873, 99874, 99875, 99876, 99877, 99878, 99879, 99880, 99881, 99882, 99883, 99884, 99885, 99886, 99887, 99888, 99889, 99890, 99891, 99892, 99893, 99894, 99895, 99896, 99897, 99898, 99899, 99900, 99901, 99902, 99903, 99904, 99905, 99906, 99907, 99908, 99909, 99910, 99911, 99912, 99913, 99914, 99915, 99916, 99917, 99918, 99919, 99920, 99921, 99922, 99923, 99924, 99925, 99926, 99927, 99928, 99929, 99930, 99931, 99932, 99933, 99934, 99935, 99936, 99937, 99938, 99939, 99940, 99941, 99942, 99943, 99944, 99945, 99946, 99947, 99948, 99949, 99950, 99951, 99952, 99953, 99954, 99955, 99956, 99957, 99958, 99959, 99960, 99961, 99962, 99963, 99964, 99965, 99966, 99967, 99968, 99969, 99970, 99971, 99972, 99973, 99974, 99975, 99976, 99977, 99978, 99979, 99980, 99981, 99982, 99983, 99984, 99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
```

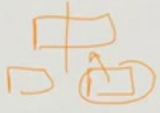
So, we can see that we are really greatly expanded the range of lists that we can sort when moving to $n \log n$ algorithm because now merge sort can handle things which are 100 times larger 100,000 as suppose to a few 1000 then insertion sort or selection sort. Another small point to keep in mind is notice that we did not run it to its recursion limit problem that we had with the insertion sort which we defined recursively. There for each element in the list, we were making a recursive call. If we had 1000 elements, we have making a 1000 recursive calls and then we have to increase the limit.

Now here even for 100,000 we do not have the problem and the reason is that the recursive calls here are not one per element, but one per half the list. So, we are only making $\log n$ recursive calls. So, 100,000 elements also requires only $\log 100,000$. Remember a $\log 1000$ is about 10. So, we are making less than 20 recursive calls, so we do not have a problem with the recursion limit, we do not have any pending recursions of that depth in this function.

(Refer Slide Time: 23:45)

Merge Sort

```
def mergesort(A, left, right):  
    # Sort the slice A[left:right]  
  
    if right - left <= 1: # Base case  
        return(A[left:right])  
  
    if right - left > 1: # Recursive call  
        mid = (left+right)//2  
        L = mergesort(A, left, mid)  
        R = mergesort(A, mid, right)  
        return(merge(L, R))
```



$O(n \log n)$
100,000

We have seen merge sort in action and we have claimed without any argument that it is actually order $n \log n$ and **demonstrated** that it works for inputs of size 100,000. In the next lecture, we will actually calculate why merge sort is order $n \log n$.