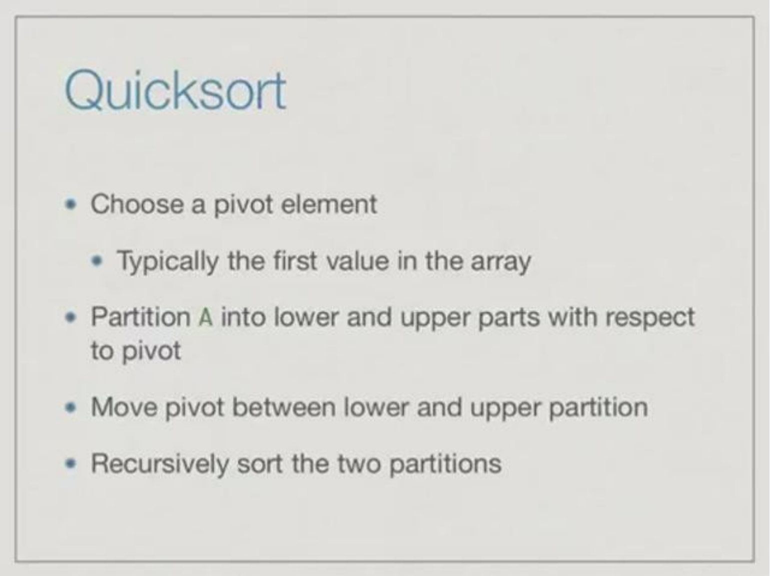
Programming, Data Structures and Algorithms in Python

Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 04
Lecture - 04
Quicksort Analysis

(Refer Slide Time: 00:01)

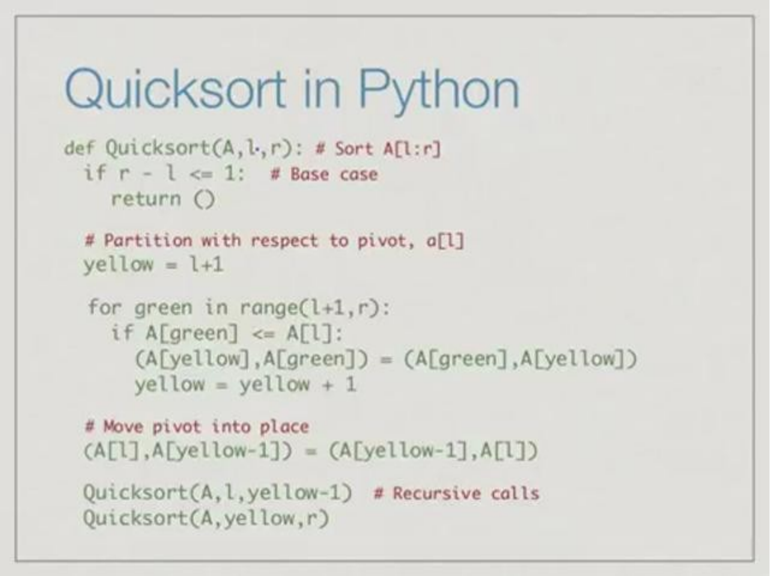


Quicksort

- Choose a pivot element
  - Typically the first value in the array
- Partition A into lower and upper parts with respect to pivot
- Move pivot between lower and upper partition
- Recursively sort the two partitions

In the previous lecture, we saw quicksort which works as follows. You first rearrange the elements with respect to a pivot element which could be, well, say the first value in the array. You partition A, into the lower and upper parts, those which are smaller than the pivot, and those which are bigger than the pivot. You rearrange the array so that pivot lies between the smaller and the bigger parts and then you recursively sort the two partitions.

(Refer Slide Time: 00:31)

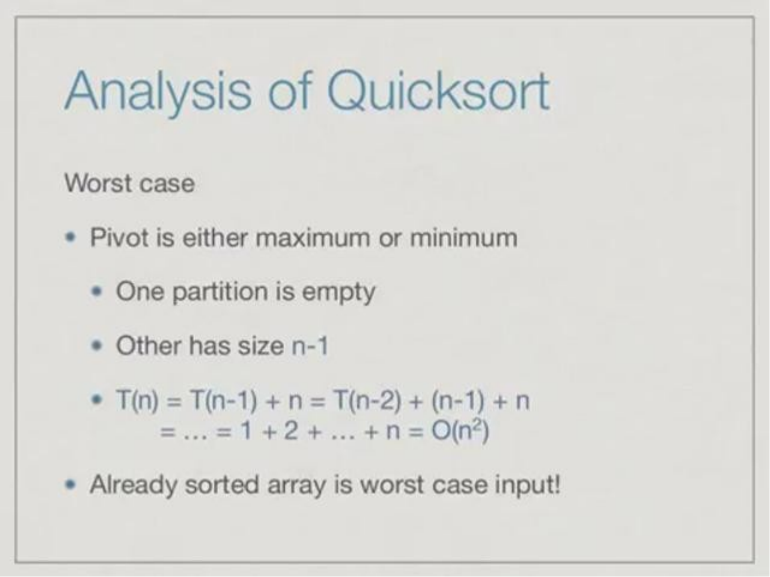


Quicksort in Python

```
def Quicksort(A,l,r): # Sort A[l:r]
  if r - l <= 1:  # Base case
    return ()

  # Partition with respect to pivot, a[l]
  yellow = l+1

  for green in range(l+1,r):
    if A[green] <= A[l]:
      (A[yellow],A[green]) = (A[green],A[yellow])
      yellow = yellow + 1

  # Move pivot into place
  (A[l],A[yellow-1]) = (A[yellow-1],A[l])

  Quicksort(A,l,yellow-1)  # Recursive calls
  Quicksort(A,yellow,r)
```

And this was the actual python code which we ran and we saw that it actually behaved not as well as merge sort even for the list of size 7500, we saw it took an appreciatively long time.

(Refer Slide Time: 00:42)



Analysis of Quicksort

Worst case

- Pivot is either maximum or minimum
  - One partition is empty
  - Other has size n-1
  - $T(n) = T(n-1) + n = T(n-2) + (n-1) + n$
    $= \ldots = 1 + 2 + \ldots + n = O(n^2)$
- Already sorted array is worst case input!

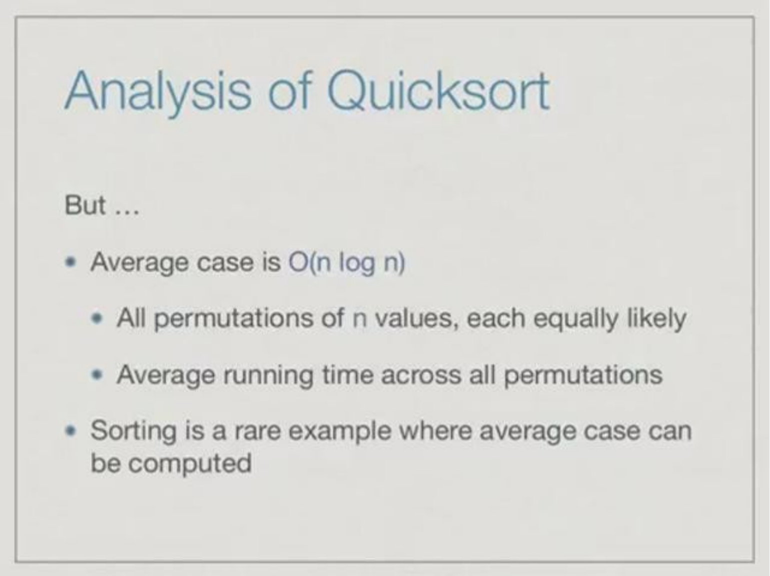
What is the worst case behavior of quicksort? The worst case behavior of quicksort comes when the pivot is very far from the ideal value we want. The ideal value we want is median, the median would split the list into two equal parts and there by divide the work into two equal parts, but if the pivot happens to be either the minimum or the maximum value in the overall array, then supposing it is the maximum then every other value will go into the lower part and nothing will go into higher part. So, the recursive call it consists of sorting of n minus 1 elements.

If one partition is empty, the other one has size n minus 1, this would happen if that the pivot is one of the extreme elements and if this happens and this is the worst case then in order to sort n elements, we have to recursively sort n minus 1 elements and we are still doing this order n work in order to rearrange the array because we do not find this until we have sorted out, gone through the whole array and done the partition.

This says t n is t n minus 1 plus n and if we expand this this comes out to be exactly the same recurrence that we saw for insertion sort and selection sort. So, t n would be 1 plus 2 up to n and this summation is just n into n plus 1 by 2 which would be order n square. The even more paradoxical thing about quicksort is that this would happen, if the array is sorted either in the correct order or in the wrong order.

Supposing you are trying to sort in ascending order and we already have an array in ascending order then the first element will be the smallest element. The split will give us an array of n minus 1 on one side and put 0 on the other side and this keep happening. The worst case of quicksort is actually an already sorted array. Remember, we saw that for insertion sort and already sorted array works well because the insert steps stops very fast. So, quicksort is actually in the worst case doing even worse than insertion sort and specifically on already sorted arrays.

(Refer Slide Time: 02:40)



Analysis of Quicksort

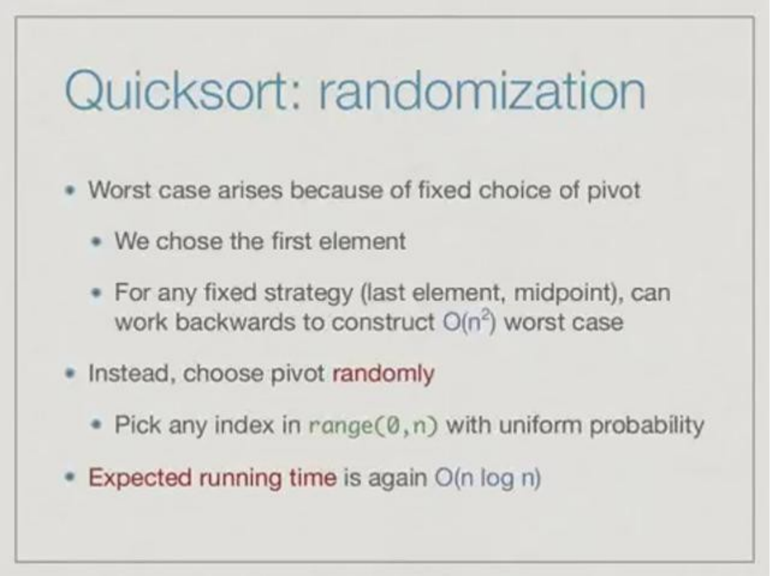
But …

- Average case is O(n log n)
  - All permutations of n values, each equally likely
  - Average running time across all permutations
- Sorting is a rare example where average case can be computed

However, it turns out that this is a very limited kind of worst case and one can actually try and quantify the behavior of quicksort over every possible permutation. So, if we take an input array with end distinct values, we can assume that any permutation of these n values is equally likely and we can compute how much time quicksort takes in each of these different n permutations and assuming all are equally likely, if we average out over n permutations we can compute an average value.

Now, this sound simple but mathematically it is sophisticated and sorting is one of the rare examples where you can meaningfully enumerate all the possible inputs and probabilities of those inputs, but if you do this calculation it turns out that in a precise mathematical sense quicksort actually works in order n log n in the average. So, the worst case though it is order n square actually happens very rare

(Refer Slide Time: 03:41)



Quicksort: randomization

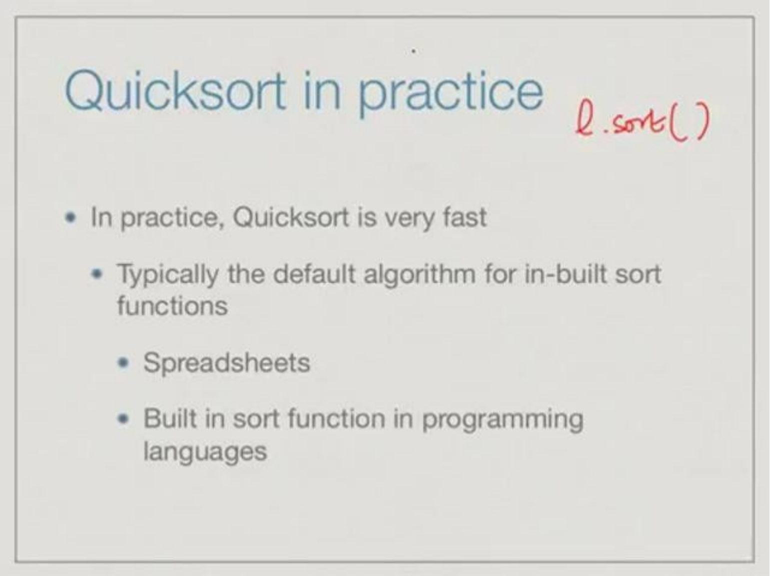
- Worst case arises because of fixed choice of pivot
  - We chose the first element
  - For any fixed strategy (last element, midpoint), can work backwards to construct O(n^2) worst case
- Instead, choose pivot randomly
  - Pick any index in `range(0,n)` with uniform probability
- Expected running time is again O(n log n)

The worst case actually arises because of a fixed choice of the pivot element. So, we choose the first element as the pivot in our algorithm and so in order to construct the worst case input, if we always put the smallest or the largest element at the first position we get a worst case input. This tells us that a sorted input either ascending or descending is worst case for our choice object.

Supposing, instead we wanted to choose the midpoint we take the middle element in the array as a random then again we can construct a worst case input by always putting the smallest value and working backward so that at every stage, the middle value is the smallest or largest value. So, for any fixed choice of pivot, if we have a pivot, which is picked in a fixed way in every iteration, we can always reconstruct the worst case.

However, if we change our strategy and say that each time we call quicksort we randomly choose a value within the range of elements and pick that as the pivot, then it turns out that we can beat this order n squared worst case and get an expected running time, again an average running time probabilistically of order n log n.

(Refer Slide Time: 04:56)



Quicksort in practice

l.sort()

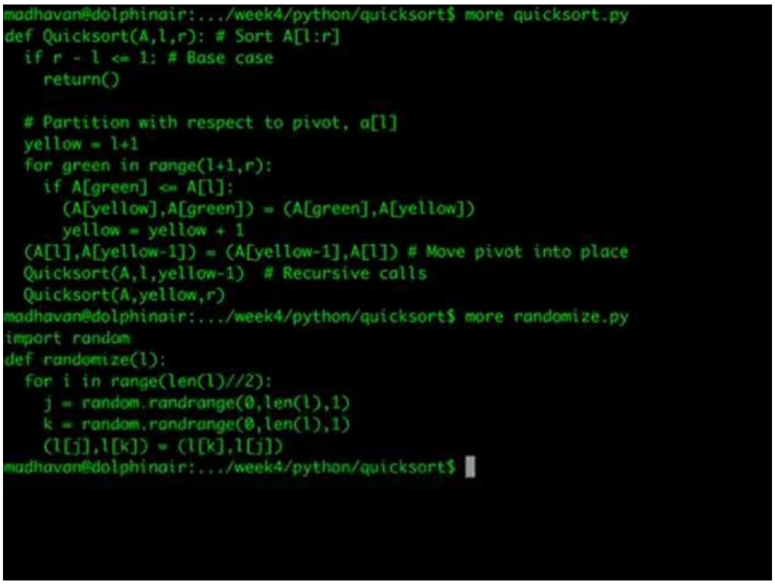
- In practice, Quicksort is very fast
  - Typically the default algorithm for in-built sort functions
    - Spreadsheets
    - Built in sort function in programming languages

As a result of this because though it is worst case order n squared, but an average order n log n, quicksort is actually very fast. What we saw is it addresses one of the issues with merge sort because by sorting the rearranging in place we do not create extra space. What we have not seen in which you can see if you read up another book somewhere is that we can even eliminate the recursive part we can actually make quicksort operate iteratively over the intervals on which we want to solve.

So, quicksort as a result of this has turned out to be in practice one of the most efficient sorting algorithms and when we have a utility like a spread sheet where we have a button, which says sort this column then more often they are not the internal algorithm that is implemented is actually quicksort we saw that python has a function l dot sort which allows us to sort a list built in. You might ask, for example, what sort is sorting algorithm is python using; very often it will be quicksort. Although, in some cases some algorithm will decide on the values in the list and apply different sorting algorithm according to the type of values, but default usually is quciksort.

So, before we precede let us try and validate our claim that quicksort's worst case behavior is actually tied to the description of the worst case input as in already sorted list.

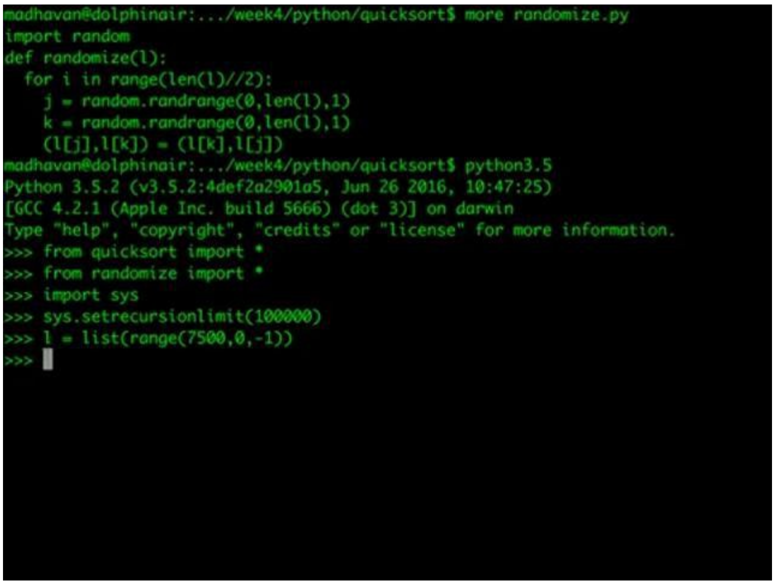
(Refer Slide Time: 06:22)



```
madhavan@dolphinair:.../week4/python/quicksort$ more quicksort.py
def Quicksort(A,l,r): # Sort A[l:r]
  if r - l <= 1: # Base case
    return()

  # Partition with respect to pivot, a[l]
  yellow = l+1
  for green in range(l+1,r):
    if A[green] <= A[l]:
      (A[yellow],A[green]) = (A[green],A[yellow])
      yellow = yellow + 1
  (A[l],A[yellow-1]) = (A[yellow-1],A[l]) # Move pivot into place
  Quicksort(A,l,yellow-1)  # Recursive calls
  Quicksort(A,yellow,r)
madhavan@dolphinair:.../week4/python/quicksort$ more randomize.py
import random
def randomize(l):
  for i in range(len(l)//2):
    j = random.randrange(0,len(l),1)
    k = random.randrange(0,len(l),1)
    (l[j],l[k]) = (l[k],l[j])
madhavan@dolphinair:.../week4/python/quicksort$ 
```

Here, we have as before our python implementation of quick sort in which we have just repeated the code we wrote before. Now, we are going to write another function which will do the following. It will shuffle the elements of a list by a repeatedly picking two indexes and just swapping them. This will allow us to take care of range output just sorted and create a suitably random shuffle of it; here is the code for it.

(Refer Slide Time: 06:52)

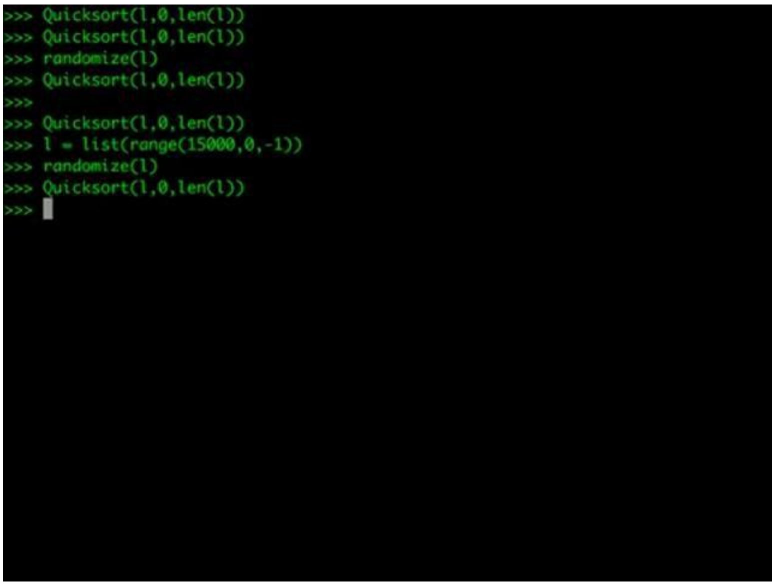


```
madhavan@dolphinair:.../week4/python/quicksort$ more randomize.py
import random
def randomize(l):
  for i in range(len(l)//2):
    j = random.randrange(0,len(l),1)
    k = random.randrange(0,len(l),1)
    (l[j],l[k]) = (l[k],l[j])
madhavan@dolphinair:.../week4/python/quicksort$ python3.5
Python 3.5.2 (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 quicksort import *
>>> from randomize import *
>>> import sys
>>> sys.setrecursionlimit(100000)
>>> l = list(range(7500,0,-1))
>>> 
```

It is very simple, you use a python library called random which allows you some functions to generate random numbers and one of the things that this library has is this function randrange. A randrange generates an integer in the range 0 to length of l minus 1. So, we pass it a list and we repeatedly pick two indexes j and k in the range 0 to length of l minus 1 and we exchange lj and lk and how many times we do this? Well we just do it a large number of times in this case say we have 10000 elements in a list, we will do this 5000 times we do it length of l by 2 times.

Let us see how this works, we load as usual the python interpreter and then we import quicksort and then we import randomize. So, you can do this, you can write python functions in multiple files and import them one after the other and they will all get loaded. Now as before we will say l for instance could be the list. So, let us also include sys and finish off that recursion limit process because we know this is gonna kill us. So, set a large recursion limit and now we set up a fairly large list we had done last for an instance 7500 down to 0 right.

(Refer Slide Time: 08:19)



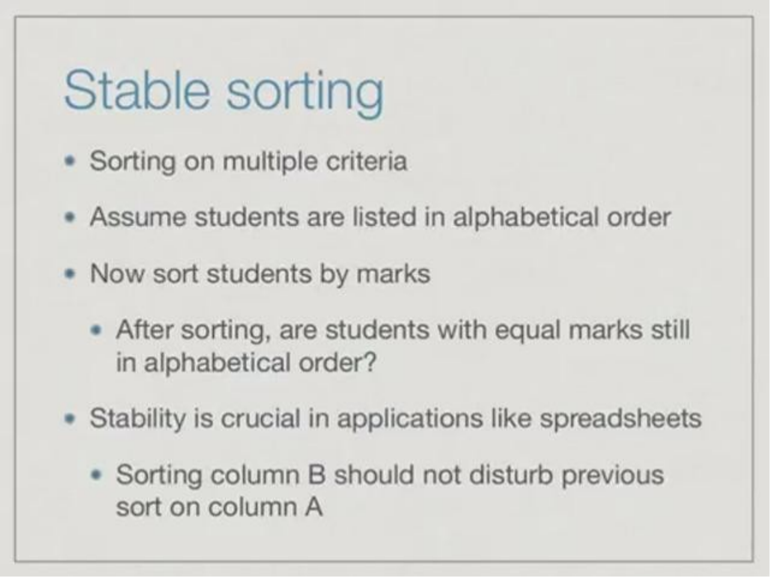
```
>>> Quicksort(l,0,len(l))
>>> Quicksort(l,0,len(l))
>>> randomize(l)
>>> Quicksort(l,0,len(l))
>>>
>>> Quicksort(l,0,len(l))
>>> l = list(range(15000,0,-1))
>>> randomize(l)
>>> Quicksort(l,0,len(l))
>>> 
```

And what we saw was that, if we try to quick sort this list it takes a long time because it is 7500 and it is a worst case in. What we are going to try and do now is and the same thing will happen even after it is sorted because even after it is sorted is still a worst case input expect now it is an ascending order. So, both descending order and ascending order take a long time. Now, supposing we randomize l. So, if you look at l now you can see that the numbers are no longer in order. So, you see some 6000 between the 7500 and 2000 and so on.

Our claim is that this will go faster and indeed you can see that if you run quicksort on this it returns almost immediately and it is not because quicksort has become any faster, it is because of order of input, because again if we have quicksort on sorted list again it is going to be slow. This just demonstrates in a very effective way that if we randomize the list and we run quicksort it comes out immediately, but if we do not randomize it and if we actually ask to sort the sorted list then it takes a long time.

So, we could actually check that, for instance, if we go back to this list and we make it say even something bigger like 10000 maybe 15000 and then we randomize it and then we sort it right it comes little fast. So, this validates our claim that quicksort on an average is fast, it is only when you give it these very bad inputs which are the already sorted once that it behaves in a poor manner.

(Refer Slide Time: 09:54)



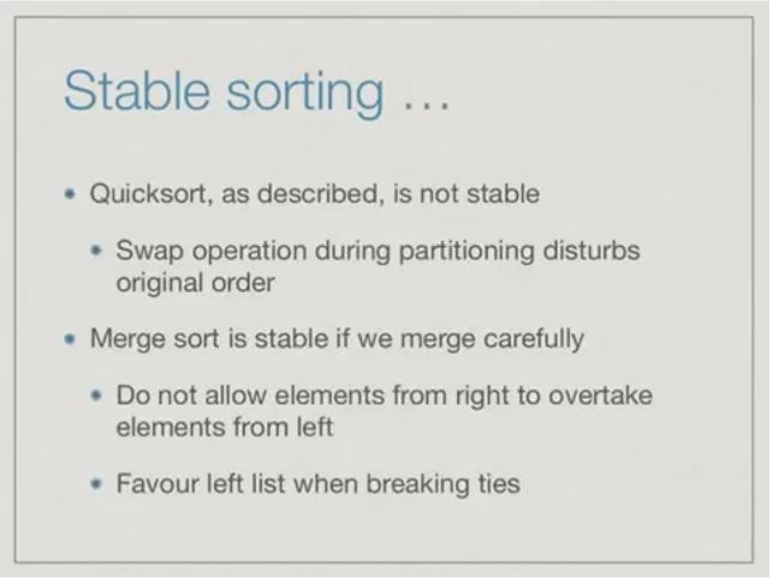
Stable sorting

- Sorting on multiple criteria
- Assume students are listed in alphabetical order
- Now sort students by marks
  - After sorting, are students with equal marks still in alphabetical order?
- Stability is crucial in applications like spreadsheets
  - Sorting column B should not disturb previous sort on column A

Now, there is one more criterion that one has to be aware of when one is sorting data. So, very often this sorting happens in stages on multiple attributes, for example, you might have a list of students who are listed in alphabetical order in the roll list after a quiz or a test, they all get marks. Now, you want to list them in order of marks, but where there are ties where two or more students have the same marks you want to continue to have them listed in alphabetical order. So, in another words you have an original order in alphabetic order and then you take another attribute namely marks and sorting by a marks should not disturb the sorting that already exists in alphabetical order.

What it amounts to saying is that if we have two list two items in the original list which are equal then they must retain the same order as they had after the sorting. So, I should not take two elements that are equal and, sort, swap them while sorting and this would be crucial when you are using something like a spreadsheet because if you sort by one column you do not want to disturb the sorting that you did by another column.

(Refer Slide Time: 11:04)



Stable sorting …

- Quicksort, as described, is not stable
  - Swap operation during partitioning disturbs original order
- Merge sort is stable if we merge carefully
  - Do not allow elements from right to overtake elements from left
  - Favour left list when breaking ties

Unfortunately, quicksort the way we have described it is not stable because whenever we extend a partition in this partition stage or move the pivot to the center what we end up doing is disturbing the order of elements which were already there in the unsorted list. So, we argued earlier that disturbing this order does not matter because any way we are going to sort it, but it does matter if the sorting has to be stable. If there was a reason why these elements were in particular order not for the current attribute, but for the different attribute and we move them around then we are destroying the original sorted order.

On the other hand, merge sort we can see is actually stable if you are careful to make sure that we always pick from one side consistently if the values are equal. So, when we are merging left and right when we have the equal to case we have to either put the element from left into the final list or right. If we consistently choose the left then it will always keep elements on to the left to the left of the ones in the right and therefore, it will remain a stable sort.

Similarly, insertion sort will also be stable if you make sure that we move things backwards only if they are strictly smaller when we go backwards and we find something which is equal to the current value we stop the insertion. So, insertion sort merge sort as stable sort quicksort as we have described it is not stable though it is possible to do a more careful implementation and make it stable.