



IIT Madras

ONLINE DEGREE

Programming, Data Structures and Algorithms using Python
Professor Madhavan Mukund
Implementation of Quicksort algorithm

(Refer Slide Time: 00:09)



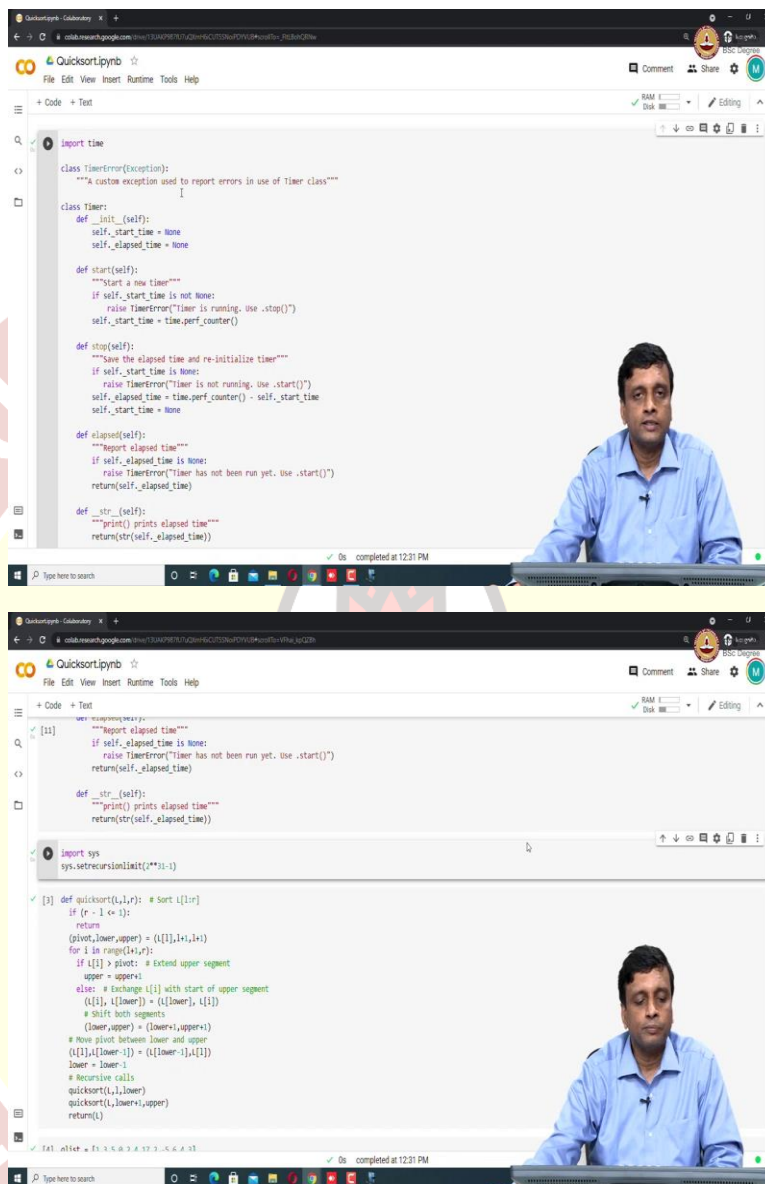
Implementation of Quicksort Algorithm

Madhavan Mukund
<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 3

So, last week we looked at some experiments with sorting algorithms with insertion sort, selection sort and merge sort to find out how they actually behaved on different types of large inputs to understand whether they were uniform across different things or they were better on ascending on descending. So, let us do the same now with quicksort.

(Refer Slide Time: 00:26)



The image displays two screenshots of a web-based Python IDE, QuickSortPython, showing code for a Timer class and a QuickSort function. A large, semi-transparent watermark for 'INDIAN INSTITUTE OF TECHNOLOGY MADRAS' is overlaid on the images, along with the motto 'सिद्धिर्भवति कर्मजा' at the bottom.

Top Screenshot: Timer Class

```
import time

class Timer(Exception):
    """A custom exception used to report errors in use of timer class"""

class timer:
    def __init__(self):
        self.start_time = None
        self.elapsed_time = None

    def start(self):
        """Start a new timer"""
        if self.start_time is not None:
            raise TimerError("timer is running, use .stop()")
        self.start_time = time.perf_counter()

    def stop(self):
        """Save the elapsed time and re-initialize timer"""
        if self.start_time is None:
            raise TimerError("timer is not running, use .start()")
        self.elapsed_time = time.perf_counter() - self.start_time
        self.start_time = None

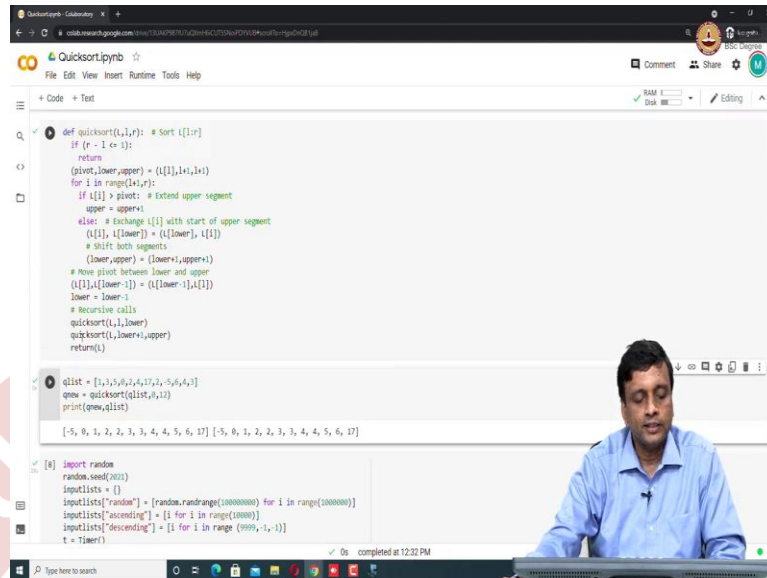
    def elapsed(self):
        """Report elapsed time"""
        if self.elapsed_time is None:
            raise TimerError("timer has not been run yet, use .start()")
        return(self.elapsed_time)

    def __str__(self):
        """print() prints elapsed time"""
        return(str(self.elapsed_time))
```

Bottom Screenshot: QuickSort Function

```
import sys
sys.setrecursionlimit(10**5)

def quicksort(l, lower, upper):
    if (upper - lower) <= 1:
        return
    (pivot, lower, upper) = (l[lower], lower, upper)
    for i in range(lower+1, upper+1):
        if l[i] > pivot: # Extend upper segment
            upper = i-1
        else: # Exchange l[i] with start of upper segment
            (l[i], l[lower]) = (l[lower], l[i])
            # Shift both segments
            (lower, upper) = (lower+1, upper-1)
    # Move pivot between lower and upper
    (l[lower], l[upper]) = (l[upper], l[lower])
    lower = lower+1
    # Recursive calls
    quicksort(l, lower, upper)
    quicksort(l, lower+1, upper)
    return(l)
```



```
def quicksort(l, l1, l2): # Sort l[l1:l2]
    if (l2 - l1 <= 1):
        return
    (pivot, lower, upper) = (l[l1], l1+1, l2)
    for i in range(l1+1, l2):
        if (l[i] > pivot): # Extend upper segment
            upper = upper+1
        else: # Exchange l[i] with start of upper segment
            (l[i], l[lower]) = (l[lower], l[i])
            # Shift both segments
            (lower, upper) = (lower+1, upper+1)
    # Move pivot between lower and upper
    (l[l1], l[lower-1]) = (l[lower-1], l[l1])
    lower = lower-1
    # Recursive calls
    quicksort(l, l1, lower)
    quicksort(l, lower+1, upper)
    return l

qlist = [1, 5, 9, 3, 4, 17, 2, -5, 6, 4, 3]
qnew = quicksort(qlist, 0, 11)
print(qnew, qlist)

[-5, 0, 1, 2, 2, 3, 3, 4, 4, 5, 6, 17] [-5, 0, 1, 2, 2, 3, 3, 4, 4, 5, 6, 17]

import random
random.seed(8021)
inputlists = []
inputlists["random"] = [random.randrange(10000000) for i in range(100000)]
inputlists["ascending"] = [i for i in range(10000)]
inputlists["descending"] = [i for i in range(9999, -1, -1)]
t = Timer()
t.start()
```

So, we begin with timer class which will allow us to time the execution and get an idea of how things happen on large inputs. And quicksort is fundamentally a recursive implementation, the one that we have given in the class. So, we are going to use that. And that is going to create problems with this built-in small recursion limit. So, we will, as we did for insertion sort in the recursive case, we will increase the recursion limit to the maximum allowed which is 2 to the power 31 minus 1.

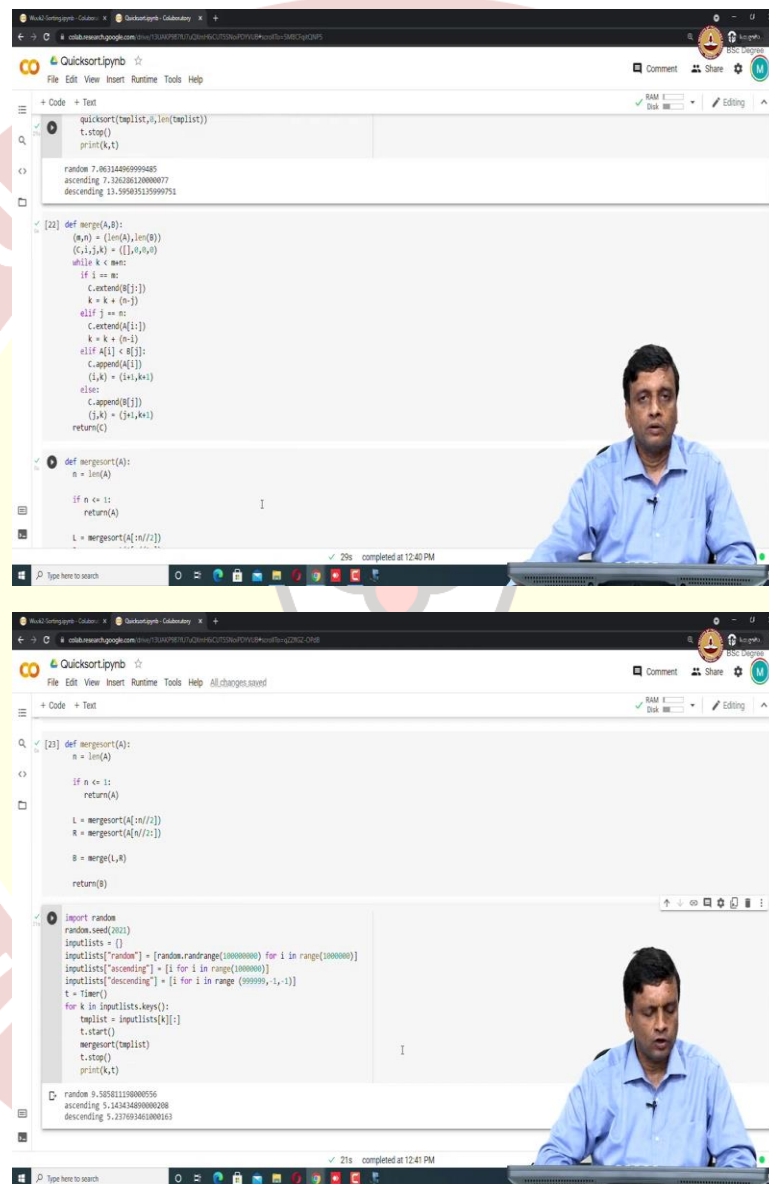
So, now, this is our quicksort algorithm that we give in class. So, what we do is we sort from a lower limit to an upper limit and we choose the first element of the list as the pivot and then we go from left to right and we partition the pivot into this lower, the remaining things lower and upper, and then we move the pivot to the middle and then we recursively sort the two parts. So, eventually we come down here and we recursively sort two parts and then we return here.

So, one the things that we want to check is, of course, this is correct. So, let us take some sort of random array and just sort it and validate that both. So, this technically sorting in place, but it is also returning the sorted list. So, you can see that if I kind of take this qlist and sort it and return it as qnew, both qnew and qlist are the same. So, it is basically in place sort that we have implemented, because we are doing all these exchanges which will actually happen inside this.

Now, the thing that we want to see is how this behaves on large inputs. So, one of the things we argued about quicksort is that typically it behaves well. So, what this means is that hopefully if we create a random array, it is going to behave fast. But because of the nature of the pivot when I

have extreme values at the 0th element, either the largest or the smallest, is going to break up these things unequally into an empty list and a list of size n minus 1. So, I am going to get this n square behavior like insertion sort and selection sort.

(Refer Slide Time: 02:24)



The image displays two screenshots of a Jupyter Notebook interface, likely from a video lecture. The background features a large, semi-transparent watermark of the Indian Institute of Technology Madras logo.

The top screenshot shows the following code and output:

```
quicksort(taplist, 0, len(taplist))
t.stop()
print(k, t)
```

random 7.0613448699999485
ascending 7.126208120000077
descending 13.595815135999751

The bottom screenshot shows the following code and output:

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

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
```

```
import random
random.seed(8021)
inputlists = []
inputlists["random"] = [random.randrange(10000000) for i in range(1000000)]
inputlists["ascending"] = [i for i in range(1000000)]
inputlists["descending"] = [i for i in range(999999, -1, -1)]
t = Timer()
for k in inputlists.keys():
    taplist = inputlists[k][:]
    t.start()
    mergesort(taplist)
    t.stop()
    print(k, t)
```

random 9.505811180000056
ascending 5.143434800000208
descending 5.237693461000161

So, let us compare this performance of quicksort with say merge sort. So, merge sort, remember, will take good time on any input. It does not have this problem ascending and descending, but let us, in particular, look at this random behavior and see which is better.

So, this is a last times implementation of merge sort. So, here, we are now doing the same thing. We are creating a random input of size 10 to the 6, exactly the same size as we had for quicksort. And for the ascending and descending for merge sort, it does not matter. We can actually have it of the same size because it is not going to blow up like the quicksort worst case.

So, if I run this now on merge sort, the same 10 to the power 6 which took 7 seconds in quicksort is actually going to take a little longer in merge sort. So, this actually shows that though they are both similar algorithms in terms of divide and conquer, we said that quicksort is the worst-case time of n^2 and we have, average case of $n \log n$, and merge sort is a worst case of $n \log n$.

So, we can see that when we have large lists, we always get something between 0 and 10 seconds, but it is interesting that merge on this random array takes actually about 50 percent more time almost than quicksort on the random array. So, this is what we meant by saying that although quicksort does not have a very attractive worst-case time, it seems to be a very effective algorithm and it is one that is often used in practice. So, that is why we typically see quicksort being taught and used even though it theoretically looks like insertion sort and selection sort in having an n^2 worst time.

