



IIT Madras

ONLINE DEGREE

Programing, Data Structures and Algorithms using Python

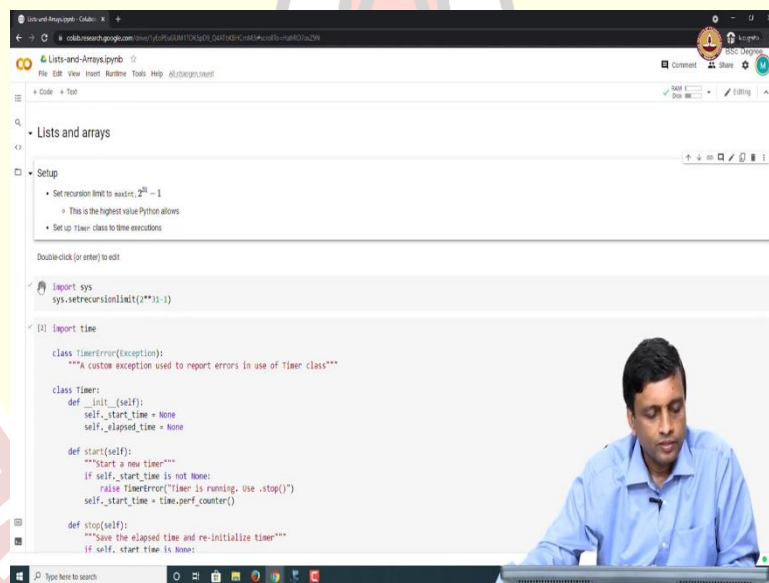
Professor Madhavan Mukund

Difference between Lists and Arrays (Implementation)

Earlier, we ran some experiments on lists and arrays in the context of searching and sorting. So, we tried to investigate, for instance, whether there was a real distinction between the order n squared sorts, namely selection sort and insertion sort. And we saw that selection sort behaves pretty much badly on any type of input, whereas, insertion sort works well on sorted inputs and so on.

Now, we have also added into this mix the problem of lists and arrays. So, we have seen that there are two different ways you can represent sequences. And we were trying to understand exactly what a Python list means. Is it a list or an array, and we claimed, actually, it is an array. So, let us see if we can validate this using some experiments.

(Refer Slide Time: 00:48)



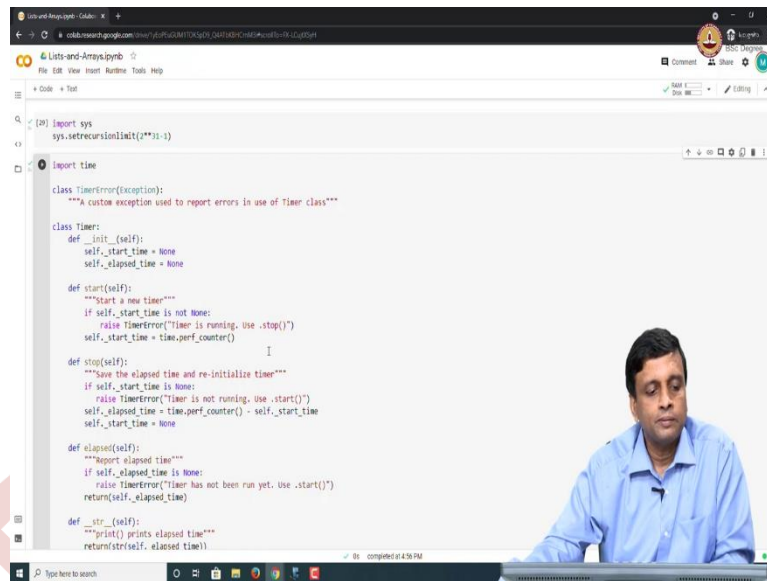
```
Lists-and-Arrays.ipynb
File Edit View Insert Runtime Tools Help JupyterLab
Code + Text
Lists and arrays
Setup
• Set recursion limit to sys.setrecursionlimit(2**31 - 1)
  ↳ This is the highest value Python allows
• Set up timer class to time executions
Double-click (or enter) to edit
In [1]: Import sys
sys.setrecursionlimit(2**31 - 1)

In [2]: Import time
class TimerError(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
```



```
(20) import sys
sys.setrecursionlimit(2**31-1)

import time

class TimerError(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()

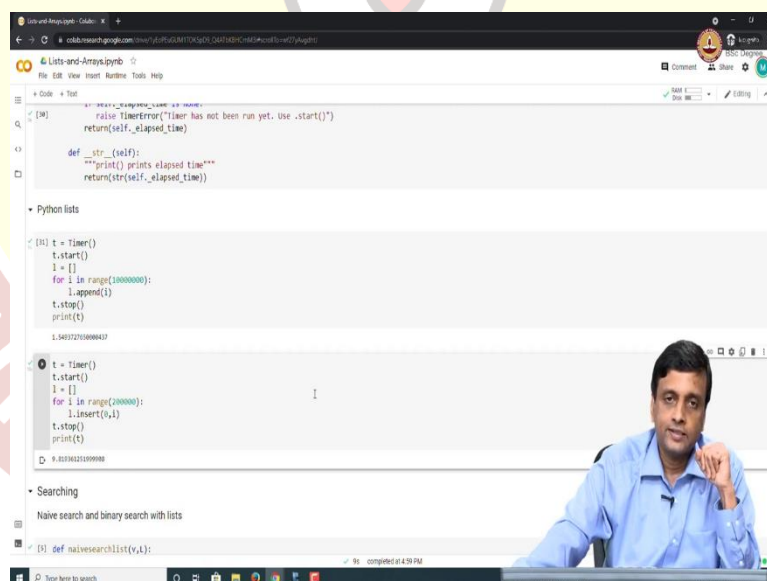
    def stop(self):
        """Save the elapsed time and re-initialize timer"""
        if self.start_time is None:
            raise TimerError("Timer is not running. Use .start()")
        self.elapsed_time = time.perf_counter() - self.start_time
        self.start_time = None

    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))
```

So, we start with our setup as before, so we set this recursion limit to be 2 to the power 31 minus 1, because this is needed in order for quicksort and insertions are to work properly when we use the recursive versions. And then we as before, create our timer class, which we will use to actually run our experiments and time our behaviors.

(Refer Slide Time: 01:09)



```
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))

Python lists

(11) t = Timer()
t.start()
l = []
for i in range(1000000):
    l.append(i)
t.stop()
print(t)

1.54937270898437

t = Timer()
t.start()
l = []
for i in range(200000):
    l.insert(0,i)
t.stop()
print(t)

9.83394221099988

Searching

Naive search and binary search with lists

(1) def naiveSearchList(v,l):
```

So, the first thing I want to check is this Python list. So, Python, we said, is a list anomaly in a list, growing and shrinking a list should be a constant time operation. But we said that, Python actually has this array implementation, so it allocates an array at a time and it keeps really, in some sense, a prefix of the array at your list with the right pointer, and the right pointer keeps moving by 1, and appending is essentially a constant time operation.

Except, when you overflow when you have to kind of double the size, but that happens only once every time it doubles, and so, amortized it is an order 1 operation. So, here is something which basically tries to create a list of 10 to the 7, and it does 10 to the 7 append. So, it starts with an empty list. So, we start with an empty list and 10 to the 7 times, we append something to it. So, we append i from i equal to 0 to 10 to 7 minus 1.

So, if we run this, and we time it a 10 to the 7 is roughly one second of Python execution. And as we would expect, since this is a constant time operation takes one and a half seconds. So, this validates the fact that append is $O(1)$, but it does not validate the fact that this is not a list in the flexible sense. So, what is the corresponding thing we can do? The corresponding thing we can do is to grow this list from the other end.

Instead of appending, we can prefix. So, we can, Python has a function called insert, which gives a position and a value. So, instead of appending, which is to take the n th or the n minus one position and add something at the end, I am going to take the zeroth position and put it before. So, I am going to do `l.insert(0, i)`. And just because it is going to take a lot of time, I am going to do it for a much smaller thing.

So I am going to do it for 10 to the power 5, not 10 to the power 7, because if I did 10 to the power 7 will be sitting here a long time. So, let us run this for 10 to the power 5. I am doing 10 to the power 5 inserts, as opposed to 10 to the power 7 apprentices so it is 100 times less work, and I end up taking two seconds. So, this shows that a 100 times less work takes about the same amount of time.

And if you want to validate something is actually quadratic, which is what we claimed it. Every insert needs to push n time, so it is $1 + 2 + 3 + \dots + n - 1$, it is a bit like selection sort. So, supposing I take this 10 to the power 5, that is 1 lakh and I make it 2 lakhs, I make it 20,000. So, I am doubling, so what would you expect that if it is n squared, if I double it is 2 squared, so it will take me 4 times as much time.

So basically, if this is 2 to the power 4 it should take me something like 10 seconds, so we can validate that. So, we can run this now. So earlier, it was taking 2 to 2.5 seconds and now it is taking a lot longer. And in fact, it takes 9.8 seconds. So, it takes roughly 4, so 2.4 became 9.8. You can do it one more time, if you really want to check, so I make it into 30,000.

(Refer Slide Time: 4:10)

```
class Timer:
    def __init__(self):
        self.start_time = None
        self.end_time = None
        self.elapsed_time = None

    def start(self):
        self.start_time = time.time()

    def stop(self):
        self.end_time = time.time()

    def __str__(self):
        return f"Timer: {self.elapsed_time}"

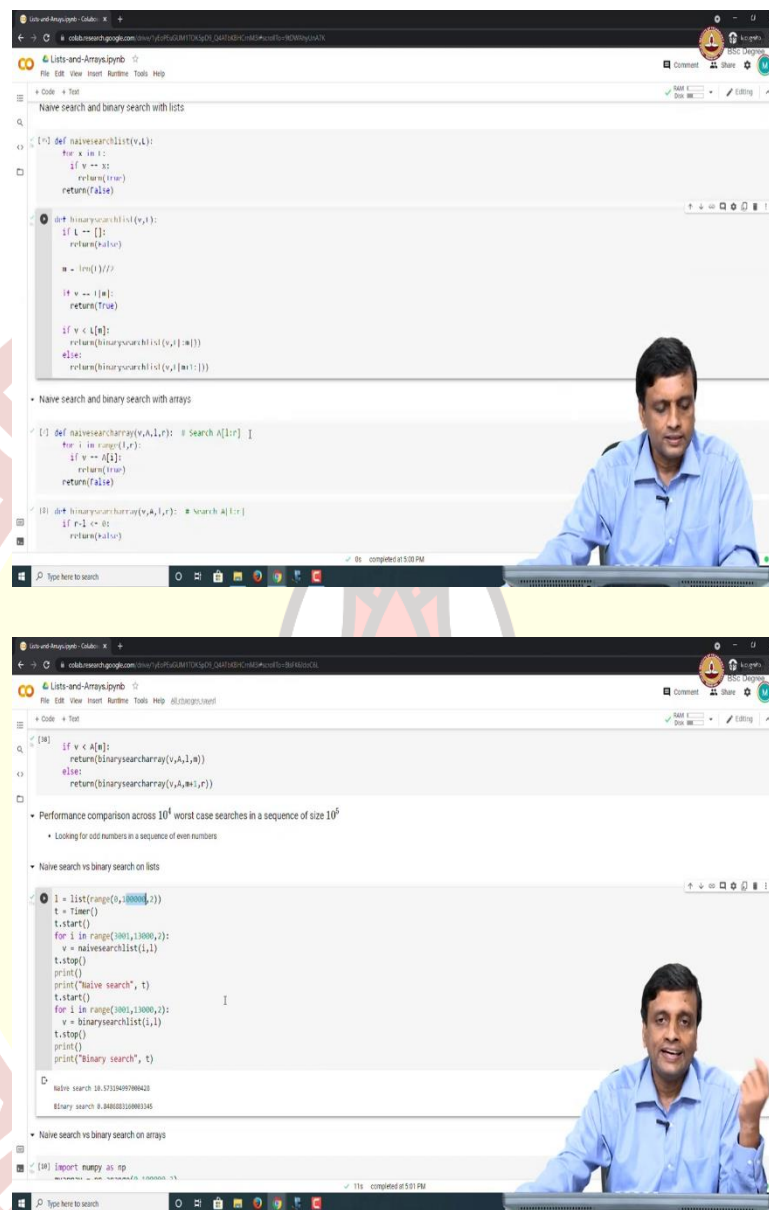
(1) t = Timer()
t.start()
l = []
for i in range(100000):
    l.append(i)
t.stop()
print(t)
1.5403270888432

(2) t = Timer()
t.start()
l = []
for i in range(100000):
    l.insert(0, 1)
t.stop()
print(t)
22.0413075698813
```

So, if I started with 10,000, and I do 30,000 it is 3 times, so 3 squared is 9. So, it should take me 9 times 2, it should take me 18 to 20 seconds. It initially 2 seconds, I will now triple the input, so the time is going to take 3 times 3, 9 times. So, if I run this on an input of size 30,000. So, what am I doing? I am taking a Python list, and instead of appending I am inserting.

So, when I insert 10,000 times it took me 2 seconds. 20,000 times took me almost 10 seconds 30,000 times it is taking me 22 seconds, which is something like 9 point something times 2.5. So, 9 into 2.5 would be 23 seconds. So, roughly, it is growing. You can see, visibly that is growing quadratically so this validates our claim that Python lists are actually arrays, and so insert are pushing things, delete, contracting, expanding a list from middle is going to be expensive, adding at the end is cheap.

(Refer Slide Time: 05:07)



The image shows two screenshots of a video lecture. The top screenshot displays a Jupyter Notebook titled 'Lists-and-Arrays.ipynb' with the following code:

```
def naive_search(list, v):
    for x in list:
        if v == x:
            return True
    return False

def binary_search(list, v):
    if len(list) == 0:
        return False
    m = len(list) // 2
    if v == list[m]:
        return True
    if v < list[m]:
        return binary_search(list[:m], v)
    else:
        return binary_search(list[m+1:], v)
```

The bottom screenshot shows the same notebook with additional code for arrays and a performance comparison:

```
def naive_search_array(v, A, l, r):
    for i in range(l, r):
        if v == A[i]:
            return True
    return False

def binary_search_array(v, A, l, r):
    if l >= r:
        return False
    m = (l + r) // 2
    if v == A[m]:
        return True
    elif v < A[m]:
        return binary_search_array(v, A, l, m)
    else:
        return binary_search_array(v, A, m + 1, r)
```

The performance comparison code is as follows:

```
l = list(range(0, 1000000))
t = Timer()
t.start()
for i in range(1000, 1000000):
    v = naive_search(l, i)
t.stop()
print("Naive search", t)
t.start()
for i in range(1000, 1000000):
    v = binary_search(l, i)
t.stop()
print("Binary search", t)
```

The output shows that the naive search took approximately 18.57 seconds, while the binary search took approximately 0.000002 seconds.

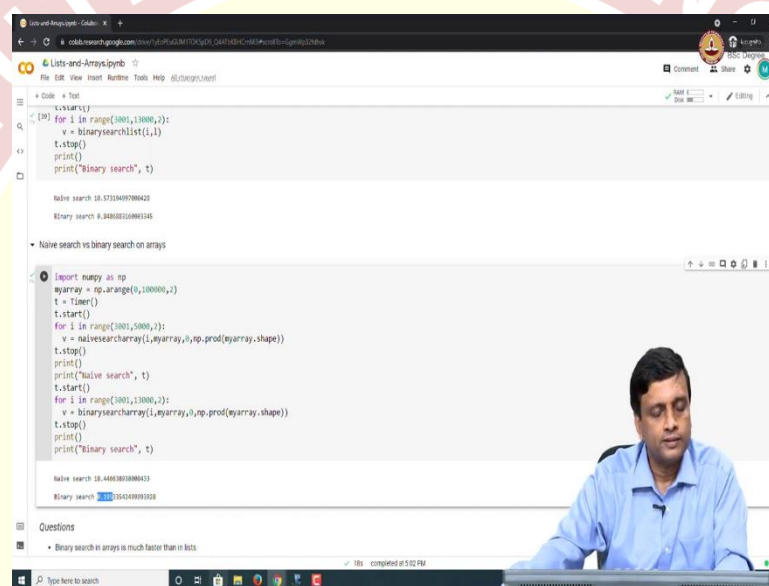
Now, let us run our searching things on lists and arrays. So, we have this naive search, which we wrote, which is just a scan of a list and binary search, which is of using lists. And we can do the same thing now with arrays. So, I am using now numpy array. So, as far as the code goes, it does not really distinguish. But the fact that it is an array means that I cannot shrink and grow, so I have to give the left end point and the right end point explicitly.

So, the main difference between the naive search and the array searches is I have to actually tell you which segment I am searching, but otherwise, it is the same function. So, you don't have to worry too much you can see the code later on. But the point is that if I now do a

naive, and binary search on a list of. So, what we have is we have a list of 10 to the power 5 elements, and we are doing about from 10,000 searches on it.

And if we do 10,000 searches on that, so it's sorry, 1,000 searches. So, this is 10 to the power 7. So, what we find is that this linear scan basically, is going to look at all the elements, so I am going to do 10 to the power 4, times 10 to the 5, 10 to the power 10 to the power 8 work, which is roughly 10 seconds. So, naive search takes 10 seconds, binary search takes a fraction of a second.

(Refer Slide Time: 06:38)



```
%%timeit
for i in range(1000, 10000, 1):
    v = binarysearchlist(i, l)
    t.stop()
    print()
    print("Binary search", t)

Naive search 10.57134007086422
Binary search 0.0000000000000000
```

```
%%timeit
import numpy as np
myarray = np.arange(0, 100000, 1)
t = Timer()
t.start()
for i in range(1000, 10000, 1):
    v = naiveSearchArray(i, myarray, 0, np.prod(myarray.shape))
    t.stop()
    print()
    print("Naive search", t)
t.start()
for i in range(1000, 10000, 1):
    v = binarySearchArray(i, myarray, 0, np.prod(myarray.shape))
    t.stop()
    print()
    print("Binary search", t)

Naive search 10.440000000000000
Binary search 0.0000000000000000
```

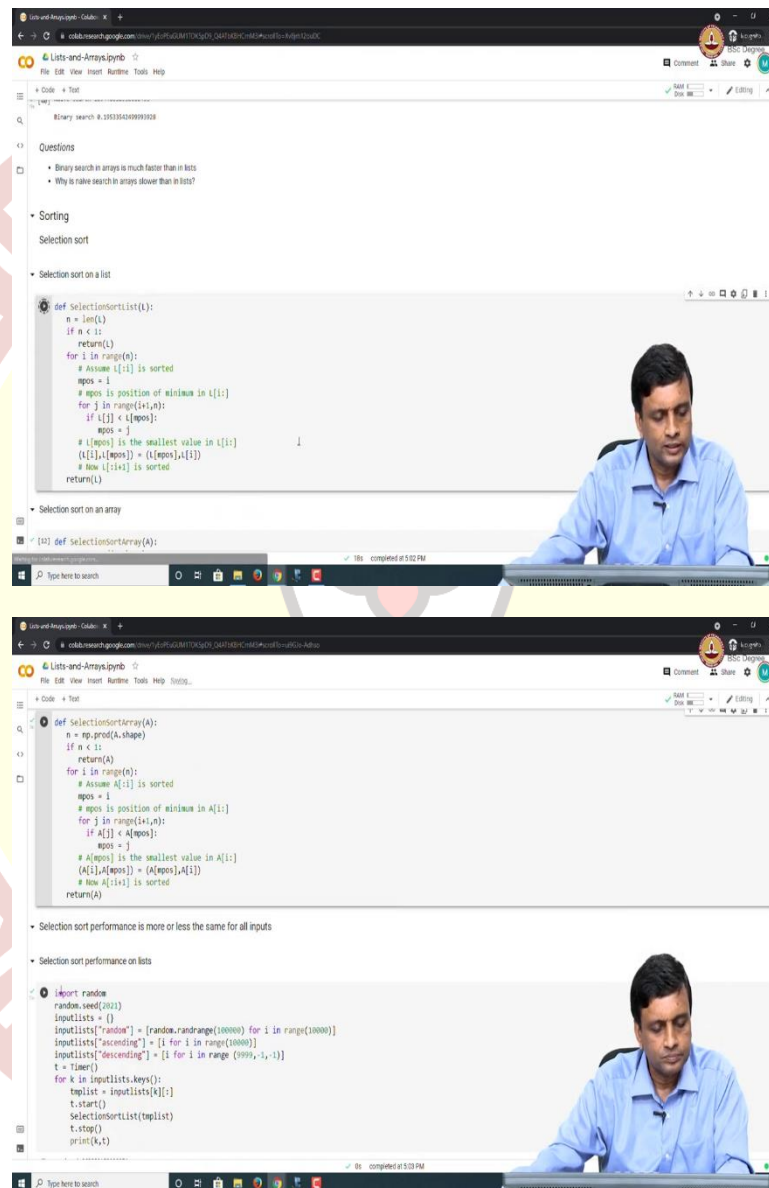
So, what happens when we do the same thing on arrays? So, again, I do the same thing. But just because I know in advance it is going to work badly, I have reduced the naive part, not the binary search. The binary search is still doing the same thing. I have 10 to the 5 elements, and I'm doing 1,000 binary searches. But then I think I have reduced it to from 10,000 binary searches to 2,000 binary searches. And the reason is that it takes a long time.

So, here I am using notice I am using a numpy array. So, I am instead of using a list of that size, I am using a numpy array of that size, and it takes a lot longer. So, this shows that actually Python lists are actually quite optimized to behave almost like arrays, but numpy arrays as we said are convenient. We will use them for graphs and other things where it is convenient to set up these matrices.

So, therefore, here for insertion, not insertion, naive search of one-fifth. Instead of 10,000 I am doing 2,000. For one-fifth work it takes almost double the time. Now, binary search

behaves as we would like. So binary search, which was taking something like 0.8 seconds now is reduced to something like 0.2 seconds. So binary search has actually speeded up for a numpy array, but it seems that this is the only one which works. Because as we go down and look at other sorting, we will find that this does not help us much.

(Refer Slide Time: 07:58)



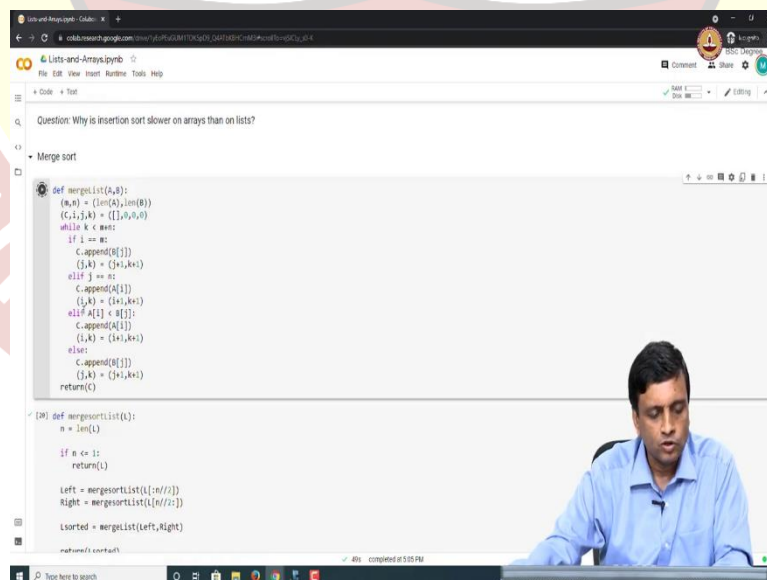
So, let us look at selection sort. This is selection sort on a list and this is selection sorting an array, again, it's very similar, except that I have to extract the size of the array using the numpy shape function, as we said. So, one of the things is that, selection sort performs pretty much the same on all lists. So, we take random lists of size 10 to the power 4, ascending and descending, and we run them all, and all of them will take roughly n^2 time.

So, 10 to the power 4 will be 10 to the power 8, so it will take something of the order of a few seconds. More than 1 second, roughly 10 seconds. So, you can say that random takes 5 seconds and all of them take about the same amount. Because remember, the selections are basically has you keep scanning, finding the maximum bring it then finding the maximum bring it and it does not really matter.

Whereas, we saw that if I look at, so let us see what it does on arrays. So, now an array is just like we saw that this naive search takes a lot longer. Even though it is supposed to be just a linear scan, the fact that you are using a numpy array seems to slow down things. So, even selection sort actually slows down. So, if I do the same thing in selection sort, as I did before, then it takes about 3 times a time.

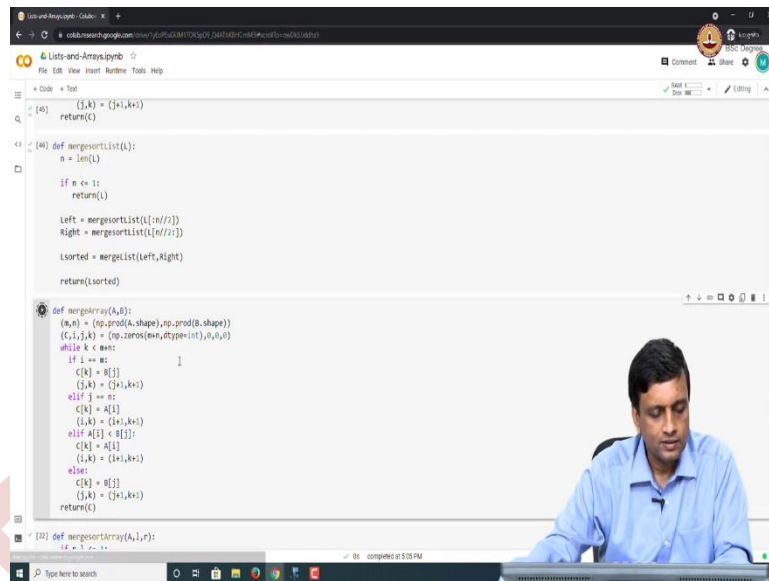
What we were seeing 5 seconds is now taking 16 seconds. So, this seems to suggest that actually this numpy array implementation though it gives you some bonus in terms of the array operations it is not actually giving you some performance in terms of the indexing, which you are primarily using for searching and sorting. So, selection sort is actually slower with arrays, you can verify that the same thing actually happens for insertion sort also. So, I will not in the interest of time, you can take this same thing and run it on insertion sort.

(Refer Slide Time: 09:56)



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

def mergesortlist(l):
    n = len(l)
    if n <= 1:
        return l
    left = mergesortlist(l[:n//2])
    right = mergesortlist(l[n//2:])
    lsorted = mergesort(left,right)
```



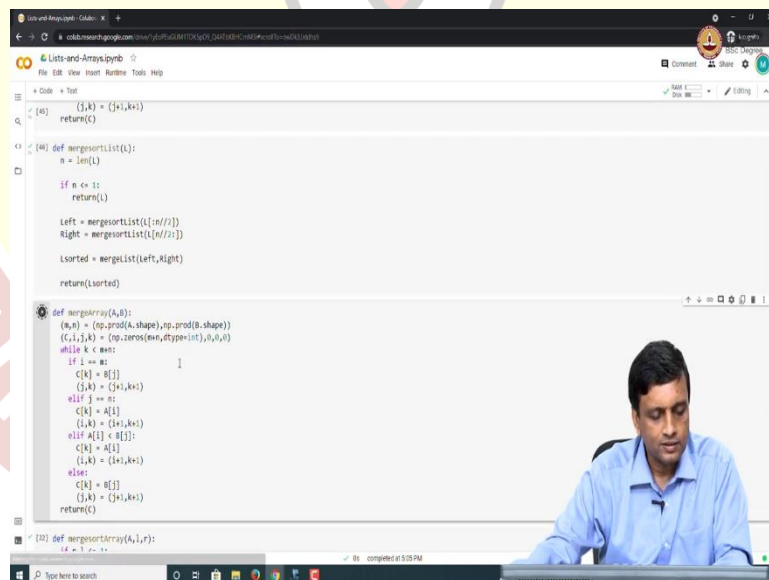
```
def mergeSort(L):
    if len(L) <= 1:
        return L
    Left = mergeSort(L[:len(L)//2])
    Right = mergeSort(L[len(L)//2:])
    Lsorted = mergeList(Left, Right)
    return Lsorted

def mergeSortArray(A):
    (n,n) = (np.prod(A.shape), np.prod(B.shape))
    (C,i,j,k) = (np.zeros((n,n), dtype=int), 0, 0, 0)
    while k < n:
        if i == 0:
            C[k] = A[i]
            (j,k) = (j+1, k+1)
        elif j == n:
            C[k] = A[i]
            (i,k) = (i+1, k+1)
        elif A[i] < A[j]:
            C[k] = A[i]
            (i,k) = (i+1, k+1)
        else:
            C[k] = A[j]
            (j,k) = (j+1, k+1)
    return C

def mergeSortArray(A):
    (n,n) = (np.prod(A.shape), np.prod(B.shape))
    (C,i,j,k) = (np.zeros((n,n), dtype=int), 0, 0, 0)
    while k < n:
        if i == 0:
            C[k] = A[i]
            (j,k) = (j+1, k+1)
        elif j == n:
            C[k] = A[i]
            (i,k) = (i+1, k+1)
        elif A[i] < A[j]:
            C[k] = A[i]
            (i,k) = (i+1, k+1)
        else:
            C[k] = A[j]
            (j,k) = (j+1, k+1)
    return C
```

So, what we will do is we will go ahead and look directly at merge sort. Because merge sort is also something interesting. So, this is my merge sort on lists, and this is my merge sort on arrays. Essentially the same thing except I have to manipulate the indices differently when I use arrays because lists, arrays, numpy arrays and Python lists have slightly different syntax.

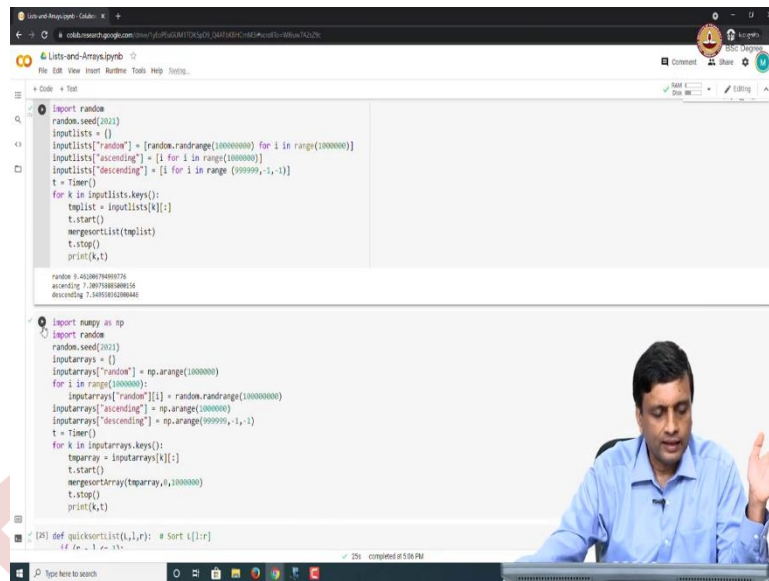
(Refer Slide Time: 10:15)



```
def mergeSort(L):
    if len(L) <= 1:
        return L
    Left = mergeSort(L[:len(L)//2])
    Right = mergeSort(L[len(L)//2:])
    Lsorted = mergeList(Left, Right)
    return Lsorted

def mergeSortArray(A):
    (n,n) = (np.prod(A.shape), np.prod(B.shape))
    (C,i,j,k) = (np.zeros((n,n), dtype=int), 0, 0, 0)
    while k < n:
        if i == 0:
            C[k] = A[i]
            (j,k) = (j+1, k+1)
        elif j == n:
            C[k] = A[i]
            (i,k) = (i+1, k+1)
        elif A[i] < A[j]:
            C[k] = A[i]
            (i,k) = (i+1, k+1)
        else:
            C[k] = A[j]
            (j,k) = (j+1, k+1)
    return C

def mergeSortArray(A):
    (n,n) = (np.prod(A.shape), np.prod(B.shape))
    (C,i,j,k) = (np.zeros((n,n), dtype=int), 0, 0, 0)
    while k < n:
        if i == 0:
            C[k] = A[i]
            (j,k) = (j+1, k+1)
        elif j == n:
            C[k] = A[i]
            (i,k) = (i+1, k+1)
        elif A[i] < A[j]:
            C[k] = A[i]
            (i,k) = (i+1, k+1)
        else:
            C[k] = A[j]
            (j,k) = (j+1, k+1)
    return C
```



```
import random
random.seed(1011)
inputlists = []
inputlists["random"] = [random.randrange(10000000) for i in range(100000)]
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():
    toplist = inputlists[k][:]
    t.start()
    mergesortlist(toplist)
    t.stop()
    print(k,t)
```

```
random 9.403061740000756
ascending 7.300750000000000
descending 7.500000000000000
```

```
import numpy as np
import random
random.seed(1011)
inputarrays = []
inputarrays["random"] = np.arange(1000000)
for i in range(1000000):
    inputarrays["random"][i] = random.randrange(10000000)
inputarrays["ascending"] = np.arange(1000000)
inputarrays["descending"] = np.arange(999999, -1, -1)
t = Timer()
for k in inputarrays.keys():
    toptarray = inputarrays[k][:]
    t.start()
    mergesortarray(toptarray, 0, 1000000)
    t.stop()
    print(k,t)
```

```
def quicksortlist(l, l1, l2):
    if l1 < l2:
```

So, now, what we did earlier was we took merge sort, and we ran it on inputs of size 10 to the power 7. And if you run it on 10 to the power seven $n \log n$ should be 10 to the power 8, so it should take roughly 10 seconds to run is what we think. So, indeed, it takes 9.5 seconds on a random input, and it works slightly better on ascending and descending order, so it takes 7 seconds on ascending. And how much time on descending?

It should also takes a similar amount of time, I think 7.5. So, it is slightly better on ascending, but roughly the same. But the point is, it is able to do 10 to the power 7, which we cannot even hope to do with insertion sort or selection sort. But if I do the same thing with arrays, again, what we will find like in the, so the same thing, I am just using the A range function of numpy to create a range of values rather than the range function for lists, and then I am resetting it to some random numbers.

So, I want random integers, because it turns out numpy as a float, random, which takes more time. So now if I run this, merge sort on the same size sequence, but as an array and not as a Python list, a numpy array and not a Python list, it actually takes appreciably more time. So, this is learning that, theoretically, we saw that lists have this flexible structure, arrays have random access, arrays should be better than lists, but Python lists are not lists.

Python lists are these kind of arrays masquerading as lists, so we already saw in the experiment above, that insert was slow compared to append because of this asymmetry in the way that lists are handled. And notice here that, merge sort on an array takes 10, 5 times as much time. This is something earlier, we saw their selections sort took about 3 times is a

much time. So, there is a certain cost that numpy arrays impose on us, which comes maybe, I do not really have an explanation as to why it happens, but and numpy has its own.

It is a layer built on top of with its own advantages in terms of matrix operations. But the cost you pay is that when you try to use them in these things, where the only thing that matters to you is the indexing. Then actually, Python lists seem to work better than numpy arrays. So, you can see now that merge sort has taken 45 seconds, so and it is going to take again similarly, another.

So, it is an order of 5 to 10 times slower for these things than it is for the list. So, I hope this shows you that, first of all, there is a difference. So, you cannot just take an algorithm on a sequence and assert something about it without knowing how the sequence is implemented. Now, theoretically, arrays are better than lists. So, theoretically, we should be able to analyze these on arrays and lists saying that arrays or random access lists are not random access, and so on. But then what seems to be a list may not be a list, and this is what we have seen.

So, a python list is not a list, a numpy array is an array, but it seems to be less friendly for these kinds of operations than a Python list is. So, you have to be a bit careful about the data structures that you use, for the context that you are using it. So, if you do not need to do any complex operations, you are just maintaining a sequence of values, go ahead, by all means and use a Python list.

If you are going to take a two-dimensional representation, like in a graph and do some operations on it, then it is a real nuisance. We saw that using that list comprehension and all that setting up a two-dimensional Python array is not the best of things. So, then it is much more convenient to work with numpy arrays and then it does not really cost you much because you are not going to be sorting those graphs and so on. So, just keep the data structure in mind when you do the operations that you have.