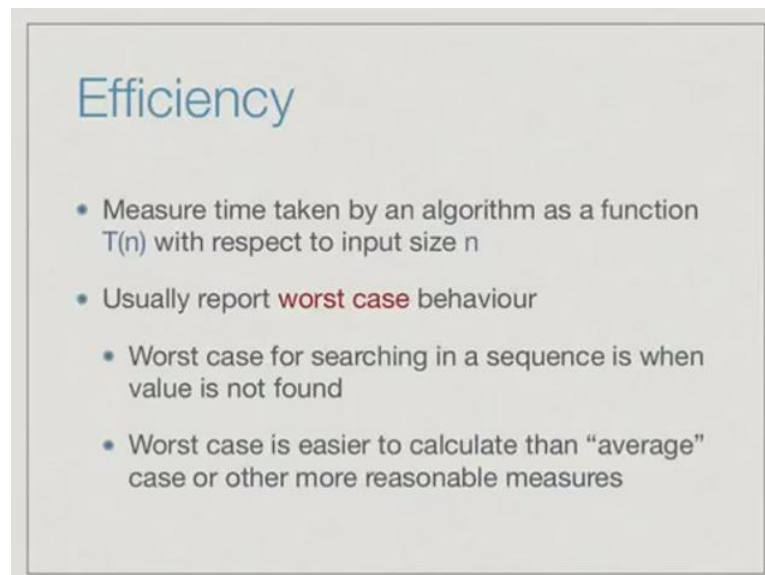


Programming, Data Structures and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 03
Lecture – 05
Efficiency

When we looked at binary search, we talked about how efficient it **was**. So let us just spend a little bit of time informally understanding how we look at efficiency of algorithms.

(Refer Slide Time: 00:02)



The slide is titled "Efficiency" in a blue font. It contains four bullet points:

- Measure time taken by an algorithm as a function $T(n)$ with respect to input size n
- Usually report **worst case** behaviour
 - Worst case for searching in a sequence is when value is not found
 - Worst case is easier to calculate than "average" case or other more reasonable measures

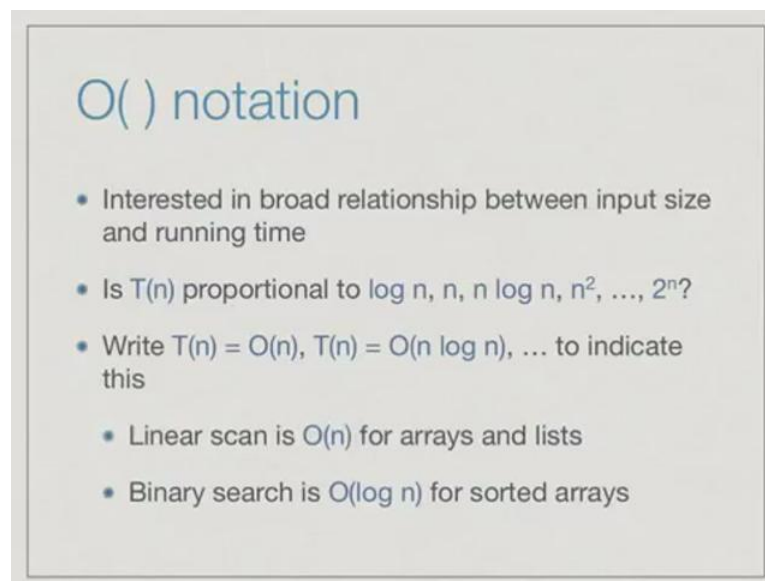
In general an algorithm will work on many different sizes of inputs, so it makes sense to talk about the efficiency as a function of the input size. The input size is n we will use a function such as T of n to talk about that time taken on an input of size n . Of course, even of the same size, different inputs will take different time for an algorithm to execute, so which of these should **we** taken as our measure of efficiency. The convention is to use the worst case behavior. Among all the inputs of size n which one will force our algorithm to take the longest time, and this is what we call usually the worst case efficiency.

Now in the case of searching for instance, binary search or even a linear scan, we said that the worst case would occur typically when the value that we are trying to find is not

found in this sequence. So, we actually have to scan through the entire sequence or array or list before we find it in case of a linear scan. And in terms of a binary search we have to reduce the search interval to a trivial interval before we can declare **that** the value is not there. So that is the worst case.

Now, it may turn out that in many algorithms the worst case is rare. It may not be a representative idea about how bad or good the algorithm is and may be it could be better to give something like the average case behavior. Now unfortunately in order to determine something like an average case in a mathematically **precise** way is not easy, we have to have a probability distribution over all inputs and then measure different inputs and different outputs and then compute a probabilistic mean for this. So in most cases this is not possible **which** is why we settle for the worst case efficiency.

(Refer Slide Time: 01:59)



O() notation

- Interested in broad relationship between input size and running time
- Is $T(n)$ proportional to $\log n$, n , $n \log n$, n^2 , ..., 2^n ?
- Write $T(n) = O(n)$, $T(n) = O(n \log n)$, ... to indicate this
 - Linear scan is $O(n)$ for arrays and lists
 - Binary search is $O(\log n)$ for sorted arrays

When we talk about efficiency, as we said we **are** broadly interested in the connection between input size and output size so we express this up to proportionality. So we are not really interested in exact constants we want to know for instance is T of n proportional to \log of n , for example in the case of binary search or n in the case of linear scan or larger **values** like $n \log n$, n **squared**, n **cubed**, or is it even exponentially dependent on the input, is it 2 to the n . We write this using this, what is called the big O notation. So when you say T of n is big O of n what we mean is that T of n is some constant times n . Same way T of n is big O $n \log n$ means T of n is some **constant** times $n \log n$. In other words,

is proportional by some constant to that value.

So, we are not going to go into much detail in this course about how **big O** is **defined** and calculate it, but **it is a** useful short hand **to** describe the efficiency of algorithms. So we will use it informally and you can go and read an algorithms text book to find out how it is more formally defined. In terms of this notation when we say that linear scan is proportional to the length of an array or a list we can say that linear scan takes time big O of n. In the same way for a sorted array binary search will take time big O log of n.

(Refer Slide
Time: 03:24)

Input	log n	n	n log n	n ²	n ³	2 ⁿ	n!
10	3.3	10	33	100	1000	1000	10 ⁶
100	6.6	100	66	10 ⁴	10 ⁶	10 ³⁰	10 ¹⁵⁷
1000	10	1000	10 ⁴	10 ⁶	10 ⁹		
10 ⁴	13	10 ⁴	10 ⁵	10 ⁸	10 ¹²		
10 ⁵	17	10 ⁵	10 ⁶	10 ¹⁰			
10 ⁶	20	10 ⁶	10 ⁷				
10 ⁷	23	10 ⁷	10 ⁸				
10 ⁸	27	10 ⁸	10 ⁹				
10 ⁹	30	10 ⁹	10 ¹⁰				
10 ¹⁰	33	10 ¹⁰					

Python can do about 10⁷ steps in a second

So, here is a table which tabulates for different values of input n what would be the corresponding values of log n, n, n log n, n **squared** and so on. And what we want to probably estimate is given these values, **these** absolute numbers, what could be reasonable inputs that we can expect to compute within a few seconds.

Now if we type something on our computer and we do not get a response very soon these days we realize that something may be wrong. So, let us say we want to see the input **in** one or two seconds otherwise we will **deem it** to be **inefficient**. So, if we look at **this**, we have to now figure out how fast our computers are. So, by some simple hand experiments you can validate that Python can do about 10 to the 7 basic steps in a second.

(Refer Slide Time: 04:19)

```
madhavan@dolphinair:~$ time python3.5 speed8.py
9999999
real    0m13.131s
user    0m12.635s
sys     0m0.187s
madhavan@dolphinair:~$
```

So what we can do is **try** and **execute** a large loop and see how much time it takes. Here we have a bunch of programs if you already written and here is a template. So if I say look at speed4 dot py. It basically executes a loop 10 to the 4 times, hence the name 4. So, for m in range 0 to 10000 minus 1, **it** just assigns m to be the value **i** and finally **there** is **this** statement we have not seen so far, but it should be quite intuitive which **says** print the value of n.

In the same way speed5 does this for 10 to the 5 times, speed6 does this 10 to the 6 times, speed7 does this 10 to the 7 times and so on. These are a bunch of scripts we have written for Python from speed4 to speed9. Now if you are working in Unix or in Linux there is a nice command called time.

First of all I can take python and I can take directly use a name of the Python program like this. So, I can say Python 3.5 and give the name of this script and it will execute it and give you the answer. But now in addition there is also a useful command called time. So, time tells us how much **time** this **thing** takes to execute and it typically reports this **in** three quantities; real time, user time, and system time. So, what we really need to look at is the so called user time it says that if I do this loop 10 to the 4 times it takes us fraction of a second 0.03 seconds. If i do this on the other hand 5 times, then it goes from 0.03 to 0.5. So, it is roughly a factor of 10 as you would imagine which is reasonable.

If I do this point 6 times then again it goes up not quite **by** a factor of 10, but it is gone up

to about 0.2 seconds. Now we come to the limit that we claim 10 to the 7. So, if we run speed7 dot py, which is the loop 10 to the 7 it takes about 1 second. I mean this is not a precise calculation, but if you run it repeatedly you say at each time, because there are some other factors like how long it takes for the system to load the Python interpreter and all that, but if you just do it repeatedly you see that the 10 to the 7 takes about the second or more. This is the basis of my saying that Python can do about 10 to the 7 operations in a second.

And just to illustrate, if you actually do it for 10 to the 8 you can see it takes a very long time, and in fact it takes roughly 10 to 12 seconds to execute so soon we would hopefully see the output. As you can see 10 seconds does not seem to us like a very long time, but it is a enormously long time when you are sitting in front of a screen waiting for the response. So what we claim now is that, something that takes a couple of seconds is what we will deem as an effective input that we can solve on our computer.

(Refer Slide Time: 07:10)

Typical functions $T(n)$..

$2^{10} = 1024$
 $2^{20} = 10^6$
 $2^{30} = 10^9$

Input	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	3.3	10	33	100	1000	1000	10^6
100	6.6	100	66	10^4	10^6	10^{30}	10^{157}
1000	10	1000	10^4	10^6	10^9		
10^4	13	10^4	10^5	10^8	10^{12}		
10^5	17	10^5	10^6	10^{10}			
10^6	20	10^6	10^7				
10^7	23	10^7	10^8				
10^8	27	10^8	10^9				
10^9	30	10^9	10^{10}				
10^{10}	33	10^{10}					

Python can do about 10^7 steps in a second

So, coming back to our table assuming that 10 to the 7 is the limit that we are looking at, let us see what happens when we mark of 10 to the 7 on these different columns. It turns out as something takes $\log n$ of time then even for 10 to the 10 it takes only 33 steps and we are fine. Of course, if input is linear then we are ignoring the constant then the input of size 10 to the 7 will take 10, so this line comes here. On the other hand if we have $n \log n$.

Now it turns out that $n \log n$, so it is useful to know that 2 to the 10 as we mentioned before is 1024 . Therefore, 2 to the 20 will be 10 to the power 6 , and 2 to the 30 will be 10 to the power 9 . Here the log grows linearly as this thing grows in terms of powers of 10 . So, when we have 10 to the 7 then the log is going to be something like 20 something, so it is going to be of the order of 10 , its going to drop one 0 . So, that is why we say that for input of size 10 to the 6 , here the log is going to contribute a factor of 10 so that is going to take time 10 to the 7 .

Now notice that **when you** do square then 10 to the 3 is already going to take 10 to the 6 . So, **somewhere** between 1000 and 10000 say around 5000 may be if you are lucky will be the feasible limit for something which takes n squared **time**. And as we go to n cubed the limit drops from a few thousand to a few hundred. So, here we have between 10 to the 6 and 10 to the 9 . So, **somewhere** between 100 and 1000 the scaling goes from 10 to the 6 to 10 to the 9 , so where 10 to the 7 will be somewhere around 200 or 300 . When you get to the exponentials like 2 to the n and n factorial, then unless you have an input that is really small like 10 or something like that we are going to hit problems, because we have a few tens you already get to enormous numbers like 10 to the 30 .

This gives us an idea that given that our system that we are working which Python can do about 10 to the 7 steps in a second, we need to really examine this table to understand what kind of inputs will be realistic to process given the time type of algorithm that we are executing. Now Python is 10 to the 7 . Python is a bit slower than other languages, but even if you are using a very fast language like C or C++ you cannot realistically expect to go beyond 10 to the 8 or 10 to the 9 . So this table is more or less valid up to a scaling of a few **tens** in different languages. So, you can take this as a reasonable **estimate** across languages.

(Refer Slide Time: 09:43)

The slide is titled "Efficiency" in blue text. Below the title, the text n^7 vs 2^n is written in red. There are four bullet points in black text:

- Theoretically $T(n) = O(n^k)$ is considered efficient
- Polynomial time
- In practice even $T(n) = O(n^2)$ has very limited effective range
- Inputs larger than size 5000 take very long

Theoretically if you look at algorithms books or complexity theoretic books, any polynomial, any dependence on n which is of the form n to the k for a constant k is considered efficient. These are the so called Polynomial time algorithms. So n cubed, n to the 5, n to the 7, all of these are considered to be theoretically efficient algorithms as compared to 2 to the n and so on. So you have n to the 7 versus 2 to the n . So, n to the 7 is considered efficient, 2 to the n is not.

But what the table tells us if you look at the previous table, is that even n square has a very severe limit, we can only do about 4 to 5000. If you are doing something in n squared time we cannot process something larger than a few thousands. Now many of the things that we see in real life, like if we have a large spreadsheet or we have anything like that and we want to sort it then it is very likely to have a few thousand entries.

Supposing, even if you want to just look at all the employees in a medium sized company or all those children in a class and in a school or something like that, a few thousands is not at all a large number. Therefore, what we see is that if we go beyond that an n squared algorithm would take enormously long time to compute. So really we have to think very hard about what are the limits of what we can hope to do and that is why it is very important to use the best possible algorithm. Because by using something which is better you can dramatically improve the range of inputs on which your algorithm works.