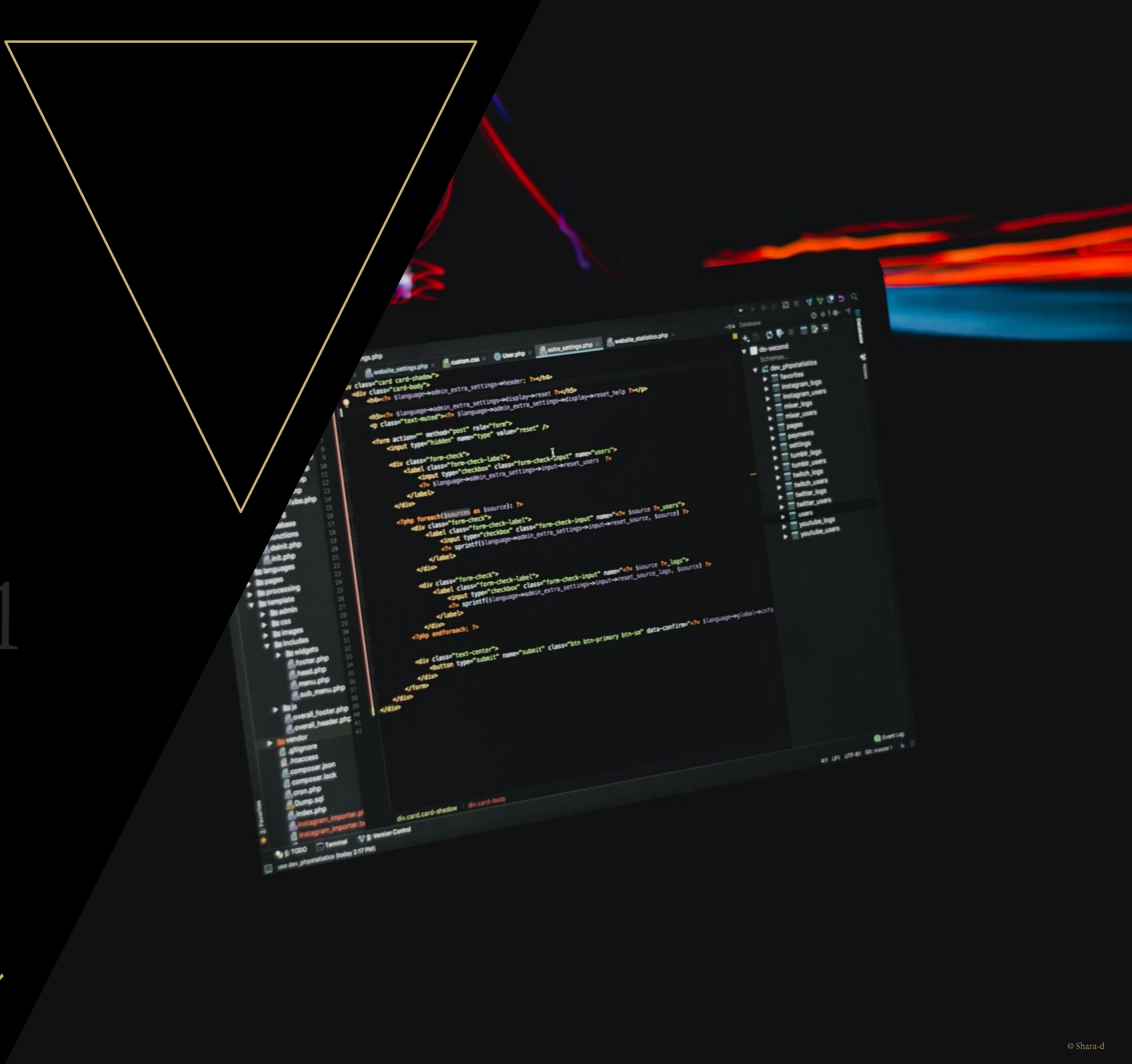
CSE 2003

DSA ETh
Digital Assignment - 1

FALL SEMESTER 2020-21



Group 4

Running time, Time-complexity, & Performance analysis of an algorithm

MJ

Mateen Jamal Khan
19BCE0003
mateen.jamal2019@vitstudent.ac.in

SA

Sharadindu Adhikari
19BCE2105
sharadindu.adhikari2019@vitstudent.ac.in

KH

Kapadia Dhvani Hemish
19BCE2079
kapadiadhvani.hemish2019@vitstudent.ac.in

LS

Lagisetty Pullaiah Sampath
18BCI0234
lagisetty.sampath2018@vitstudent.ac.in

Running Time

The **running time of an algorithm** for a specific input depends on the number of operations executed.
The greater the number of operations, the longer the **running time of an algorithm**.

It could take nanoseconds, or it could go on forever. It also depends on the input.

For example:

```
function addEm(a, b){
var c = a + b;
return c;
};
```

This algorithm takes two arguments, adds them together, then returns their sum. This wouldn't take long at all. Fractions of a second. However, check this one :

```
function endless() {
while(2 === 2) {
console.log(“aaaaaaaaah");
};
```

This is a “while true" algorithm. It takes a condition that will always return true (2 will always equal 2) and does something over and over again ad infintium. In this case, it will continuously print “aaaaaaaah" to the console until you force it to end or your computer catches fire.

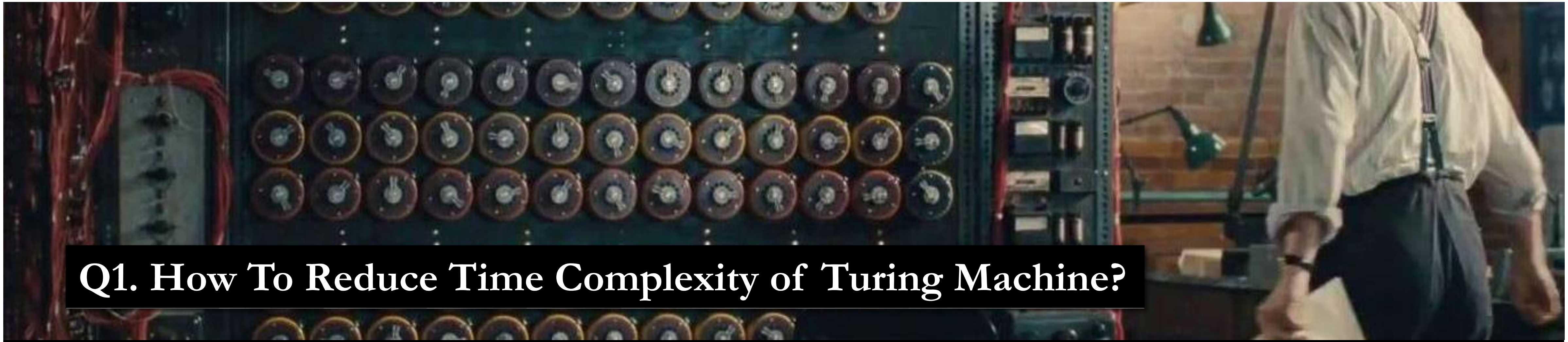
Time Complexity

Big O Notation	Name	Example(s)
$O(1)$	Constant	#Odd or Even number, #Look-up table (on average)
$O(\log n)$	Logarithmic	#Finding element on sorted array with **binary search**
$O(n)$	Linear	#Find max element in unsorted array #Duplicate elements in array with Hash Map
$O(n \log n)$	Linearithmic	#Sorting elements in array with **merge sort**
$O(n^2)$	Quadratic	#Duplicate elements in array **(naïve)** #Sorting array with **bubble sort**
$O(n^3)$	Cubic	#3 variables equation solver
$O(2^n)$	Exponential	#Find all subsets
$O(n!)$	Factorial	#Find all permutations of a given set/string

Time complexity estimates how an algorithm performs regardless of the kind of machine it runs on. It can be determined by "counting" the number of operations performed by the code. This is defined as a function of the input size n using Big-O notation. n indicates the size of the input, while O is the worst-case scenario growth rate function.

We use the Big-O notation to classify algorithms based on their running time or space (memory used) as the input grows. The O function is the growth rate in function of the input size n.

The notation is useful when comparing algorithms used for similar purposes.



Q1. How To Reduce Time Complexity of Turing Machine?

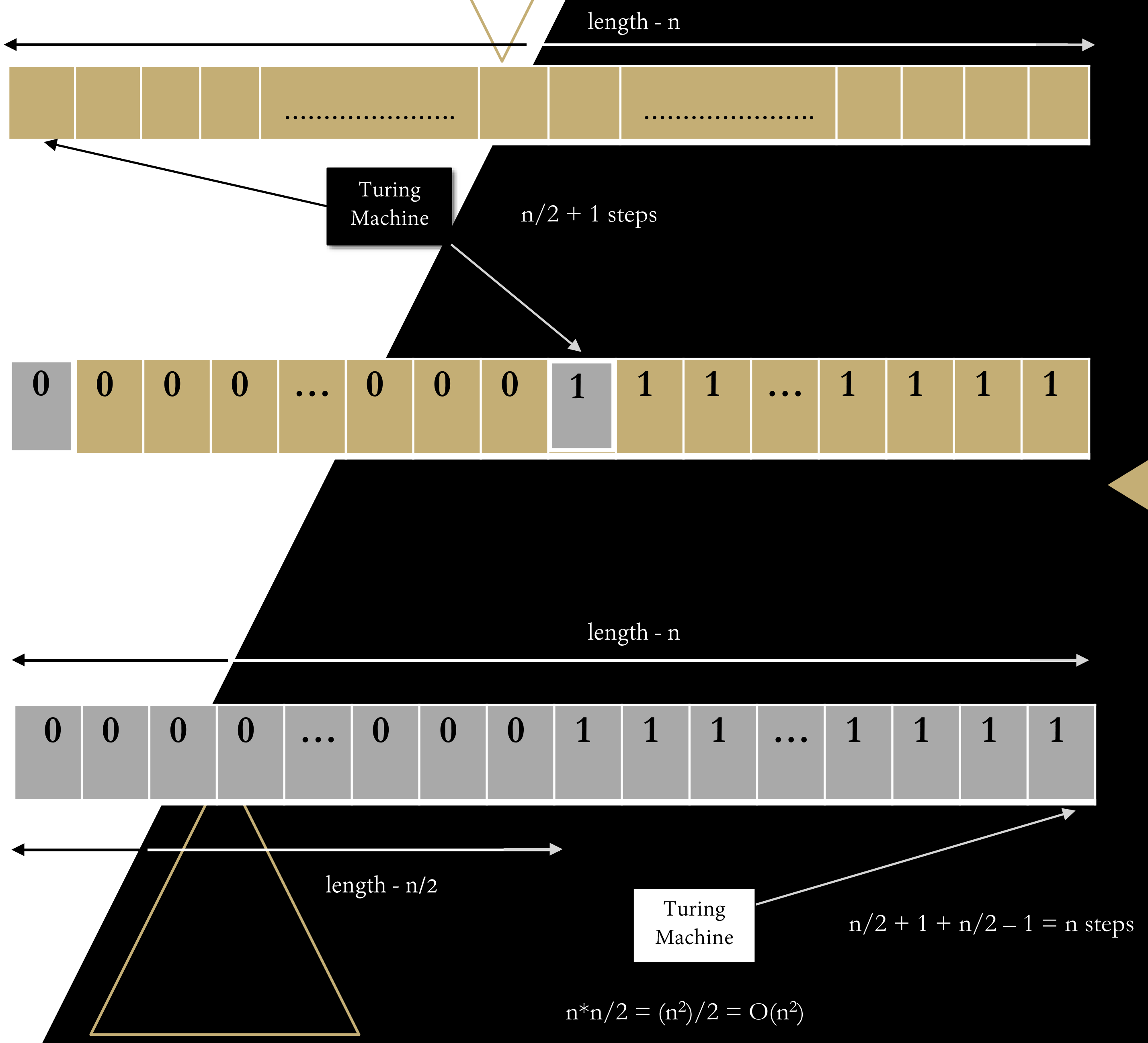
- During WW2, a brilliant mathematician named Alan Turing joined the British Military to crack the German Enigma code. He went on to become one of the founding fathers of Computer Science by inventing the TM.

- So the Turing Machine basically is used for the verification whether a string belongs to a particular language or not.

i. First, we check for the corner cases that every 0 in the string must be followed by A 1. Else we reject the string

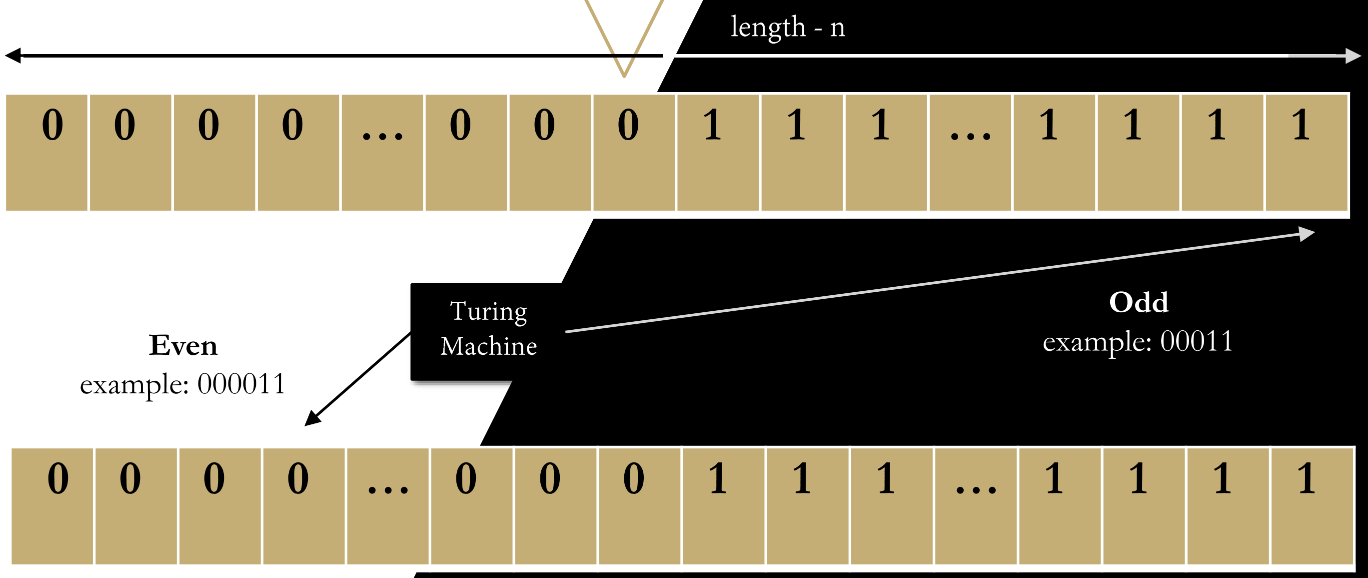
ii. Now, in the 1st algorithm we check every zero and corresponding to that we check A 1. This takes n/2 + n/2 = n steps

iii. So, each scan takes n steps and we repeat it for n/2 times, thereby achieving a total of n^2 time complexity in this way; represented by $O(n^2)$.

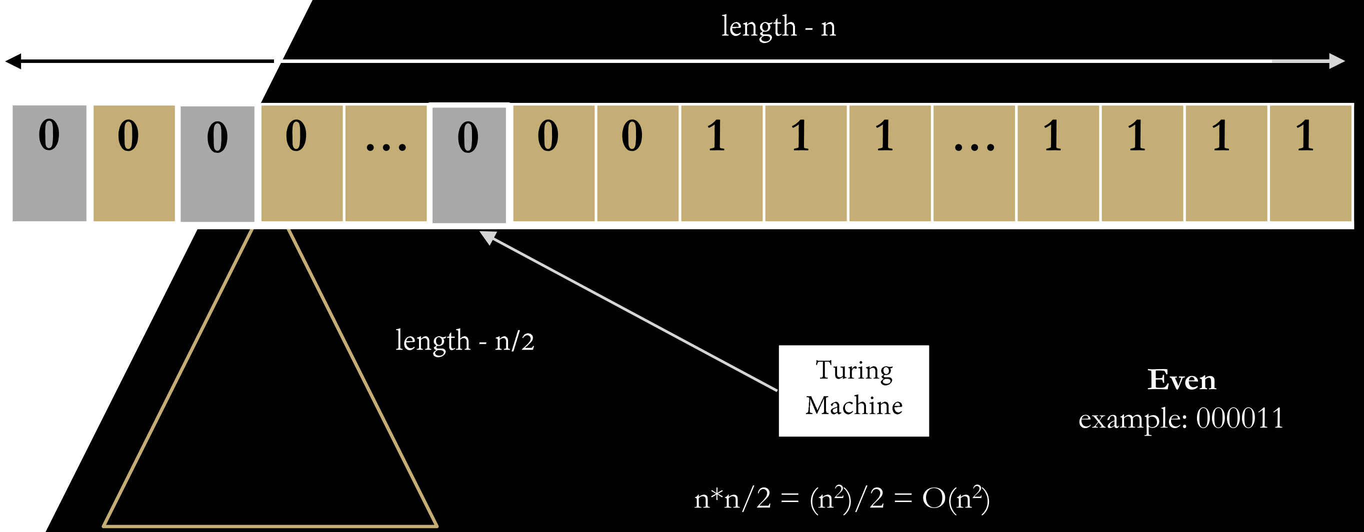


[Better Optimal Strategy]

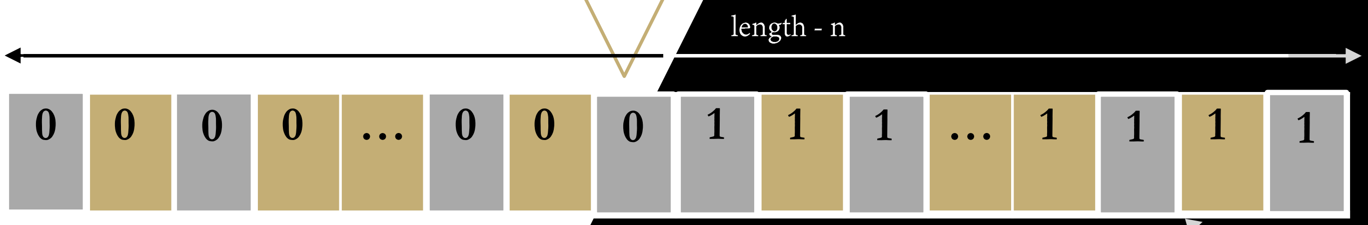
iv. Now, here also at first we check the corner cases. Then we count the total number of O's and 1's. If it's odd, we reject it and if it's even, we accept it.



v. After this we'll cross every other 0 that we see in the input string. Same operation is performed on 1's as well.



vi. After performing the above operation, we'll count the remaining 0's and 1's. Now if that sum is zero, we'll accept it and if it's odd, we reject it. And the same operation is performed until we are left with nothing.



n = 64

32

16

8

4

2

0 - accept

log2(n) + 1 =

log2(64) + 1 =

7

Turing Machine

Examples:
000011 = 001
000111 = 01

If it is Odd, Reject
If it is Even, continue till the end

Each turn, the head does a full scan of tape – O(n) steps

vii. So in this way, in each repetition we'll deal with half of the actual number of bits.

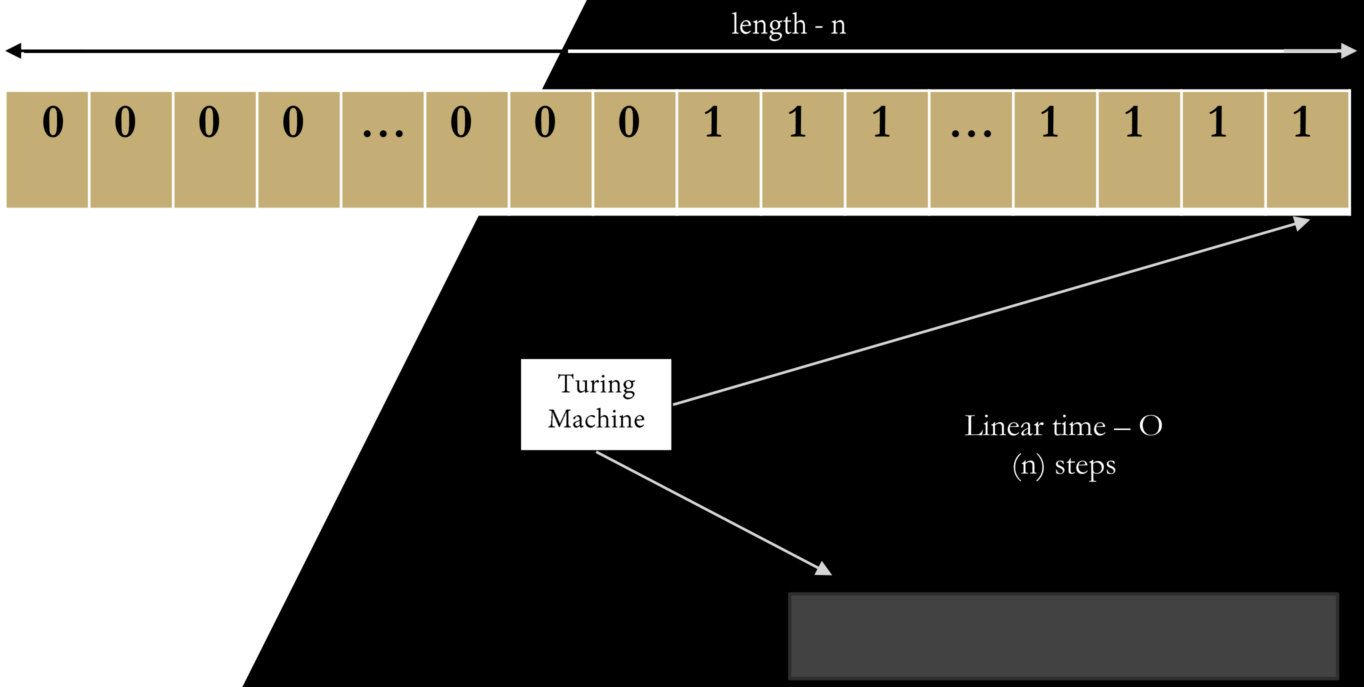
$$O(n)*O(\log(n)) = O(n*\log(n))$$

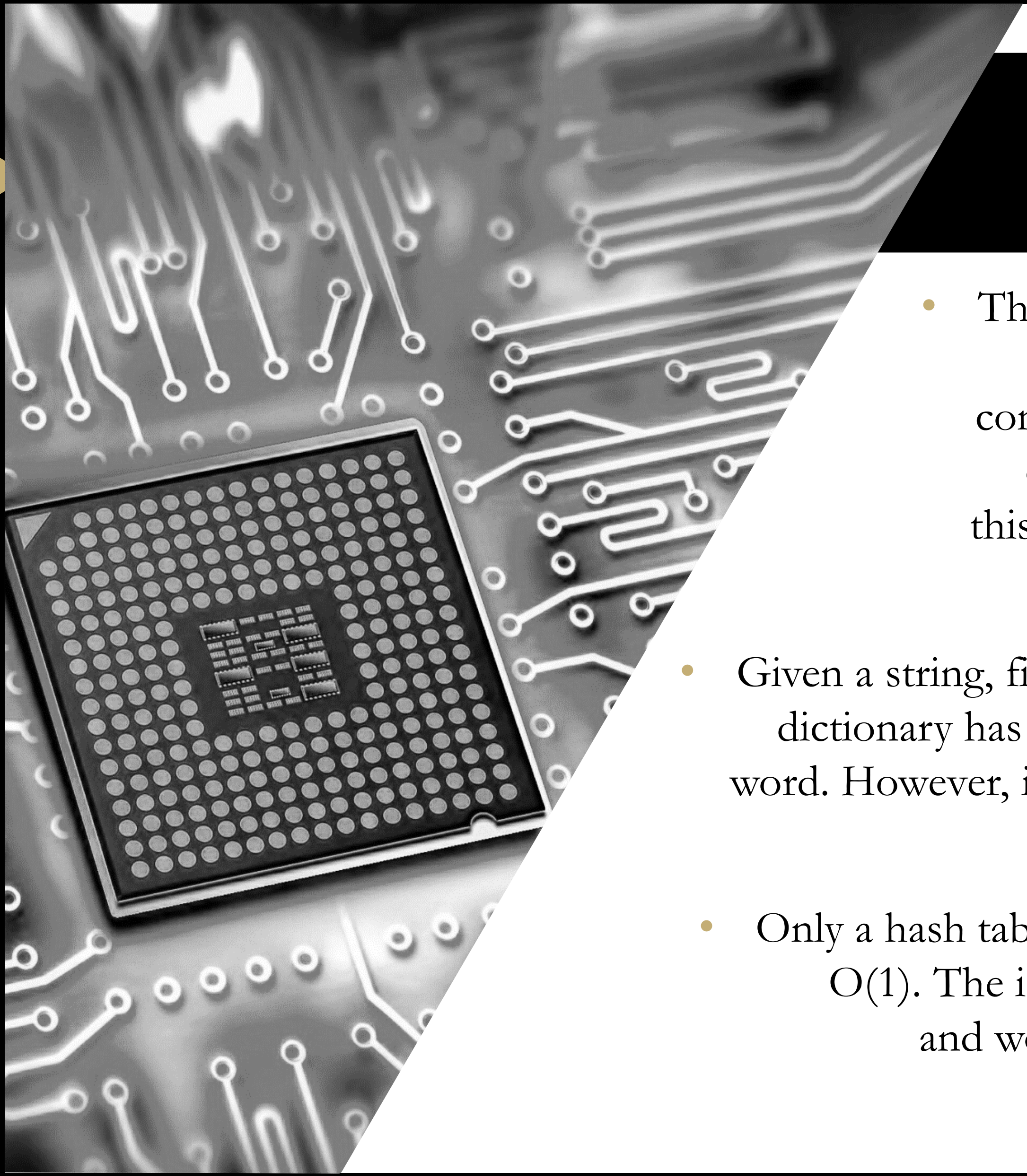
Optimal solution for one tape Turing Machine

viii. So the number of times we need to repeat this log2(n) and in each turn the head does a full scan of tape that is O(n) steps. So the resultant reduced time complexity is obtained as O(n*log2(n)).

[What if we use a Two-Tape Turing Machine?]

ix. So in yet another algorithm if we employ a stack and pop in all zeroes and pop out them on occurrence of A 1 one by one time complexity in this case will be O(n)





Q2. How does processors search for a word from the entire document?

- This is predominantly done through string searching: finding a string withing a string. For example, when we execute the “Find” command in the word processor, the program starts at the beginning of the string holding all the text (let’s assume for the moment that this is how our word processor stores text, which it probably doesn’t) and searches within that text for another string we’ve specified.

- Given a string, find its word frequency data. Again, we can be sure that even if the dictionary has 10 or a million words, it would still execute line 4 once to find the word. However, if we decided to store the dictionary as an array rather than a hash map, then it would be a different story.

- Only a hash table with a perfect hash function will have a worst-case runtime of O(1). The ideal hash function is not practical, so there will be some collisions and workarounds that leads to a worst case runtime of O(n). Still, on an average, the lookup time is O(1).

Rabin-Karp example:

String: LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL
LLLLLLLLM
Pattern: LLLLLM
Let Hash Value of “LLLLLL” = 0;
And Hash Value of “LLLLLM” = 1;

Step 1: LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLM
LLLLLM -------------- 0! = 1 (One Comparison)
Step 2: LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLM
LLLLLM -------------- 0! = 1 (One Comparison)

....
....

Step N: LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLM
LLLLLM ----------- 1 = 1

Hash Value is matching so compare elements one by one. (6+1 Comparisons)

[How can we optimise and do better?]

Using Rabin-Karp String Search

- Michael O. Rabin, a professor at Harvard, along with his colleague Richard Karp devised a method from using hashing to do string search in O(m+n), as opposed to O(mn). In other words, in linear time as opposed to quadratic time, a nice speedup.
- The Rabin-Karp algorithm uses a technique called finger-printing.

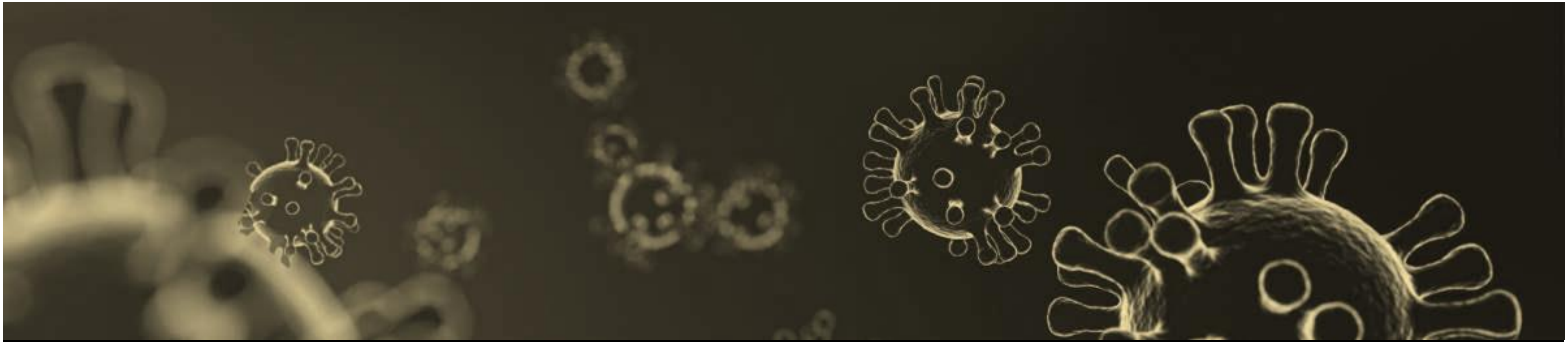
Performance Analysis

Let’s take an example. If we want to go from city A to city B, there can be many ways of doing this. We can go by flight, using a bus, a train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science, there are multiple algorithms to solve a problem. When we have more than one algorithm, we need to select the most efficient one. Performance analysis helps us in this regard.

Q3. How coronavirus outbreak can end?

- The present outbreak of coronavirus acute respiratory disease called covid-19 has resulted in a major epidemic. The main reason why coronavirus is a major problem is because its spread can be modelled by a tree.

- Before the world took lockdown measures, estimates stated that each infected person was infecting between two to four other people. This number is called R_0, a mathematical denotation that indicates how contagious an infectious disease is. For instance, if a disease has an R_0 of 15, a person who has the disease will transmit it to an average of 15 other people.

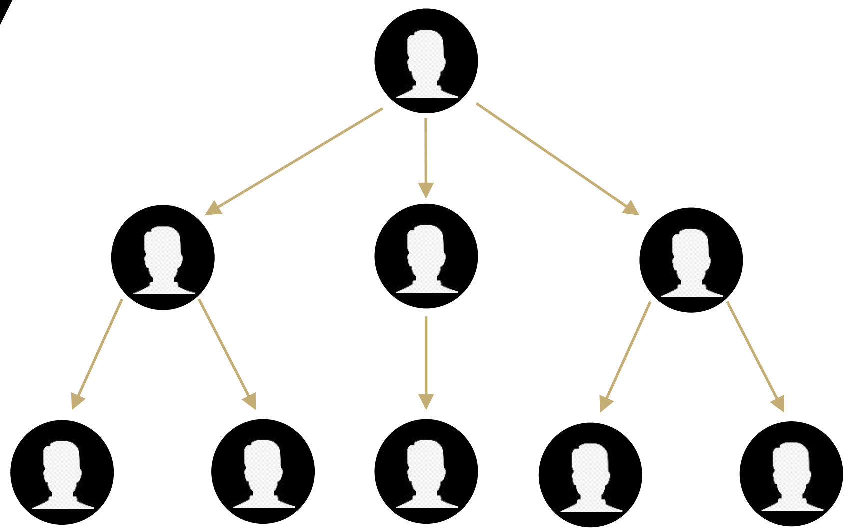


- Importantly, a disease's R_0 value only applies when everyone in a population is completely vulnerable to the disease, as in the case of covid-19 where no one has been vaccinated, on one has had the disease before and there's no way to control the spread. In our model, R_0 is the average number of children each node in the tree has. This means, each model in our tree has (on average) between two and four children.

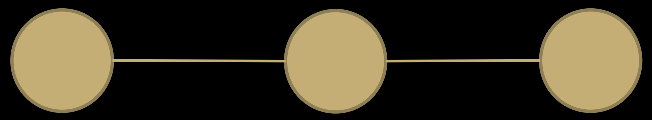
A total of 3 possibilities exist for the transmission or decline of a disease, depending on its R_0 value:

Cases	Values of R_0	Condition of disease
1	$R_0 < 1$	Will eventually die out
2	$R_0 = 1$	Disease will stay alive and stable but there won't be any outbreak
3	$R_0 > 1$	May be an epidemic or outbreak

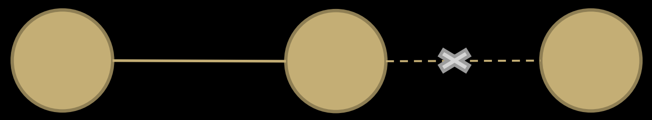
As this is a tree structure, by analysing it we know that this is going to get very large very quickly. The early objective of health organizations worldwide was to reduce R_0 to around one (or less). If $R_0 = 1$, then each leaf node in our tree now becomes the head of a linked list. Each person is infecting exactly one other person, in the same way, that a (singly) linked list has a reference to the next node in the list.



If $R_0 = 1$



If $R_0 < 1$



If $R_0 < 1$, then at some point a person will infect no-one else, and the line of infection (for the leaf) is broken. We can model that in code by having the node point to a null reference, meaning it is the ultimate node in the linked list.

One way to "solve" the coronavirus situation is to change the behaviour of the virus so that it can be modelled by a collection of (eventually finite) linked lists, rather than a tree. And hence, by analysing the structure of the virus we can potentially stop its spread.

Q4. How performance analyzing can help create our own algorithm?

It is sometimes best to write algorithms from first principles rather than to modify a large computer program already written. The way to do this would be to go back to the original problem. Develop a paradigm for analysis. It may be along a similar line to what has been done before, or it may be a totally new approach. However, to make a good comparison, we should understand the existing algorithm thoroughly. That is one reason why performance analysis of existing algorithms is important. To understand the existing algorithm, we may need to analyse it piece by piece. It should be kept simple. If we've more than one degree of freedom, we cannot know what variable caused a certain result. So break the algorithm into components with one degree of freedom, and then see what each does. Performing the analysis of the algorithm, it is possible to improve on it and develop an efficient method, thereby creating our own version.

Q5. What is the real-life difference between Performance & Complexity analysis?

- The only way to know for sure what solution is best for a problem is to measure it. However, measuring performance is actually quite a hard thing to do; there are many things that go into making sure we measure what we want to measure. Outside noise introduced by the OS, other processors, random lags, compiler optimizations, etc. It's impossible to know all the factors that influence the performance of your algorithms, after all even a random burst of electromagnetic radiation from the sun might fry your computer and your algorithm will never finish performing its task. This is where performance measurements are so much different from complexity analysis.

- Obviously no one is expected to predict solar flares, or other random events that we have no control over. But when talking about performance the factors that the developer can consistently account for are very important to ensure the maximum performance of our software. One such very important concern is hardware constraints. When considering which algorithm and data structure to choose, from a performance analysis standpoint it is very important to take into account things like on what machine will this code run in and optimize it for the environment it will be used in.

</END>