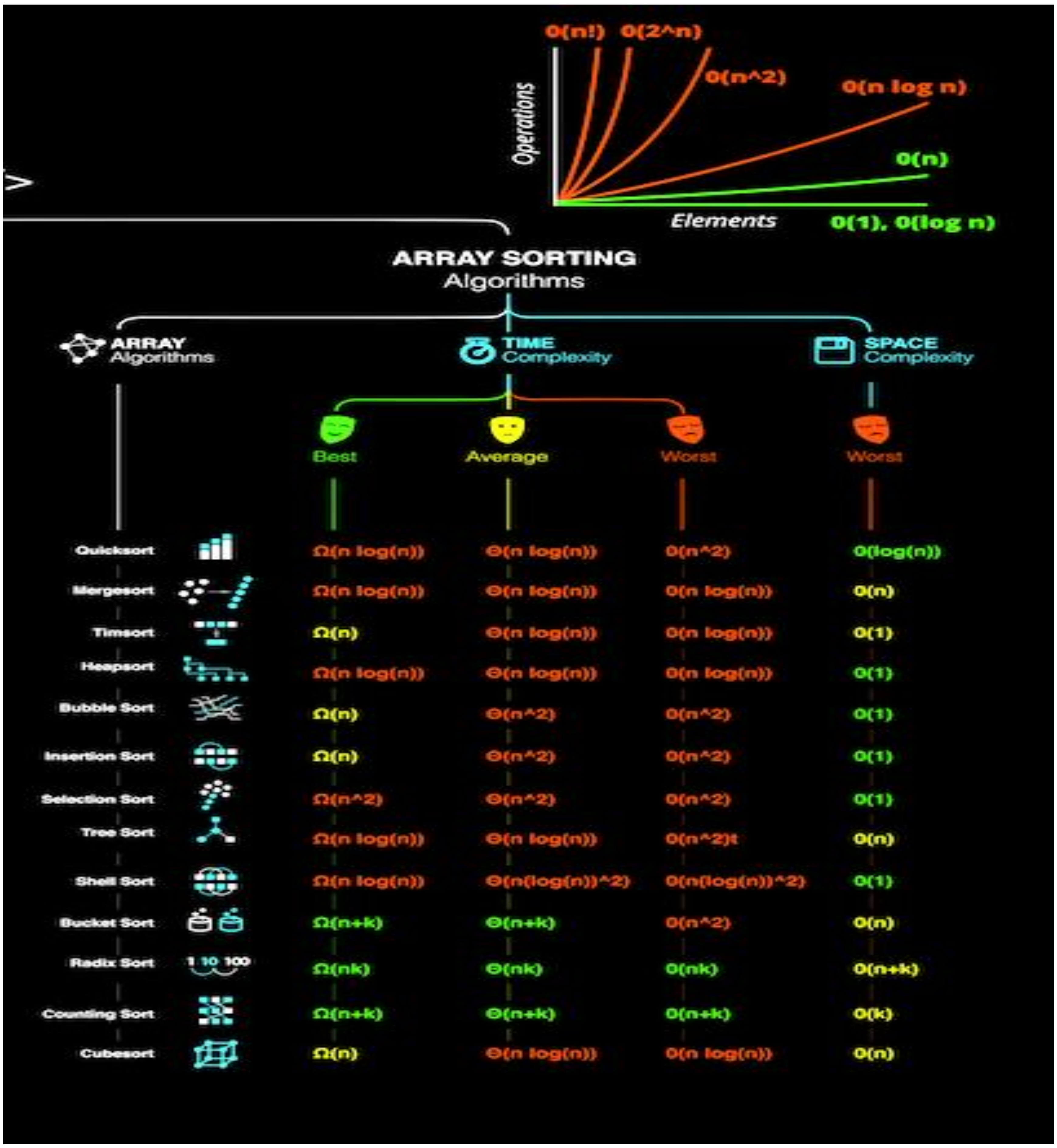

O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(1), O(log n)
Operations
Elements
ARRAY SORTING Algorithms
ARRAY Algorithms
TIME Complexity
SPACE Complexity

ARRAY Algorithms	TIME Complexity			SPACE Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(1)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)t	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

ANALYSIS OF ALGORITHMS

Sunil Sitaram Shelke

ABSTRACT
These notes discuss, briefly, topics listed in MU syllabus of course CSC402.

Analysis of Algorithms
CSC402

MODULE 1:- Introduction

0.0. Algorithm and it's Analysis:- A Bird's Eye View

- An algorithm is a ***finite*** sequence of ***well-defined*** instructions to solve a well-defined ***computational*** problem.

- ***Well-defined***:- Means unique interpretation. Having no ambiguity. Ex:- Do x+4, Concatenate strings x and y. not well defined:- Add 4 or 5 to x, Get leafy vegetables, If (x==0) { either sort array A or search an element in A}

- ***Computational Problem***:- A solvable problem which can be solved by a ***series of computations***.

- ***Computation:-*** Evolving process through a sequence of simple and ***local*** steps. (Theory of Computation is all about mathematical theory of defining and working on concept of a computation. So do not worry about this now).

  Eq:- Is 2^n a local step? Maybe. May be not. Most of the computer architectures have ***left shift*** operation which can be performed in 1 clock cycle. So, for a 32-bit architecture, doing $2^{1\le n\le 32}$ can be considered as a local step. But calculating $2^{n>32}$ will still be not considered a local step, as at least 2 clock cycles will be required to do so. So, whether or not to consider exponentiation by 2 a local step is entirely up to us.
  Eq:- Sorting an array of n elements can almost surely be considered as a ***non local*** step. We clearly can see lot many comparisons and other operations required to achieve sorted effect. This is so for n in general.
  But can we consider sorting an array of 5 elements a local step? Surely, 5 elements can not be sorted in 1 step. But then it does not require more than 8 comparisons. If we do not want to pay attention to number of steps required to sort 5 elements then it can well be considered as a local step.

- Basically, given any problem, an algorithm is a sequence of computations which ALWAYS stops with a correct answer, for all inputs. But ***what is a problem***?

Problem, Description of a Problem

- **Example1:- Very informal way**:- Consider **checking whether a given natural number n is a prime number or not**. Bold sentence is an English description of what is it that we want to solve.

- **More Formal**:- Above problem can be described more formally as below:-

  i/p:- $n \in \mathbb{N}$ **o/p:- Yes , n prime**
  No, otherwise

  Problem can then be defined as i/p – o/p description along with set of all possible values that n can take i.e set of all possible input instances. So, say, we label above problem as $CHECKPRIME$ then problem, more formally, is $\boldsymbol{CHECKPRIME} = \{\mathbf{2},\mathbf{3},\mathbf{4},\dots\}$, apart from it's i/p – o/p description.

- **Example2**:- **Very informal way**:- Given n and an n size array L of integers, sort L increasingly.

- **More Formal:-** First, describe set of all possible inputs i.e input instances. Any particular input instance will be a sequence L of n integers, for eq. for $n = 8$ as an example, $L = [12, 9, 2, 4, -29, -19, 56, 101]$. So, mathematically, any such $L \in \mathbb{Z}^n$, where $\mathbb{Z}$ is the set of all integers. $\mathbb{Z}^n$ creates a sequence of n integers, by a cartesian product of $\mathbb{Z}$ with itself $n - 1$ times.

  Any input instance will be such a sequence. We can get an array of any size n and elements of array can be any integers. We will say we have solved sorting problem if we can sort any such random array for any value of n. So, we collect all such sequences for all possible values of $n \in \mathbb{N}$. Any such sequence will be a particular input at a particular time. So,

  $SORT = \{L \mid \exists n \in \mathbb{N}\ L \in \mathbb{Z}^n\}$ …set of input instances
  Particular I/P instance :- $L[1]L[2]L[3] \dots L[n]$
  O/P:- permutation L' of L such that $L'[1] \le L'[2] \le L'[3] \dots L'[n]$

- Why do all this? Later we will be interested mostly in runtime and space complexity of any algorithm which solves a particular given problem. We will see soon that time complexity surely depends on size of a particular input instance and it's structure also. So we can write unambiguously about time and space requirements dependent on instance. Moreover, a mathematical description of an input instance gives us more clear idea of how instance looks like and how can we go about processing it for an answer.

- Basically, a problem, formally, is a set of all possible input instances. But, ***how to represent an instance***? Does it even matter if we represent an instance in binary or hexadecimal or some other fancy or cryptic symbol system?

Instance Representation, Size of an Input Instance

- Consider **Example1** above of prime checking. An instance of this problem $CHECKPRIME$ will be a natural number. If we represent instance in decimal system then $CHECKPRIME = \{2,3,4,\dots\}$. However, if we choose to represent a natural number in binary then $CHECKPRIME = \{10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, \dots\}$.

Notes

This sub-Section is added as a preliminary note for helping understand why and how theoretical analysis is done and how that may be different from so called practi cal aspects. Reasons, why it is still important is a question whose ans wer is readily understood once these details are digested.

- Why should we worry about binary representation? We need not, actually, theoretically. But do we really have a device which understands things in decimal representation? Electronic computers work in binary so they do not understand decimal. Also, any algorithm which solves prime checking problem will spend time in reading input instance and will read it multiple times, probably, to solve it correctly. Naturally, time taken by algorithm will be dependent on time taken to read input instance, which depends on size of an instance which in turn depends on representation of instance. For eg. time to read binary is more than time to read same number in decimal as there are less spaces required in decimal.

- Consider **Example2** above of sorting. An example instance given above $L = [12, 9, 2,4, -29, -19, 56, 101]$ can be written in binary as $L = [\ 1100, 1001, 10, 100,\ \ 00011,\ 01101, 111000,\ 1100101\ ]$.. Notice that $-19, -29$ are represented in $2'$s complement.

- So, time taken by an algorithm will be dependent on size of an input instance and that size depends on how instance is represented. So, which representation should we choose? The answer is that representation will not matter as long as 2 different representations are only ***constantly*** different from each other. Means, size of an instance in one representation is only constantly different from size of the same instance in other representation. For eg, it takes $\log_{10} n$ places to represent integer n. But requires $\log_2 n$ places in binary. $\log_2 n / \log_{10} n = \log_2 10 = const$.

- We will not discuss technical reasons why above arguments about independence from actual representation. This is a subject matter of ***Theory of Computation*** where these things are discussed in detail to establish *theory of computational complexity*. So, size of an input instance is generally, traditionally represented by a parameter n. **Notice that this single parameter n talks about total size of instance in 1 single number**.

- But wait. Consider problem of adding two matrices $A_{p\times q}$ and $B_{p\times q}$, for some p and q i.e adding two matrices having p rows and q columns. Consider it takes b bits to store a single number. So, total input instance size will be $n = 2pqb$ of both matrices combined together. Generally, we see that number of operations to solve this problem is given as pq. So, we generally see it as time requirement of matrix addition algorithm as $T(p, q) = pq$. This becomes 2 parameter time equation. This does not look like $\boldsymbol{T(n)}$.

- So why this change? Point is that we maintain that time complexity is a function of input size but when we want get just an estimate of time requirement then we may relax a bit and calculate time in terms of ***important parameters*** about instance, ignoring everything else. So, as in example above we may relax and write time requirement as $T(p, q)$ as an estimate instead of $T(n)$ as an exact equation. ***So we are allowed to slightly misuse/abuse concept of size of input instance, as long as we are not abusing/misusing too much. Difference between time requirement calculation after abusing/misusing/relaxing concept of size must be only constantly different from exact time requirement***.

- Another example of such relaxation is seen in sorting. If one integer requires some constant b bits and we have n size array of such numbers then total size of an instance is $n' = nb$, because we have already used n for number of elements in array. So we should be writing time requirement equation of any sorting algorithm as a function of size $n'\ as\ T(n')$. But we generally see time equation as $T(n)$ i.e as a function of n=number of elements in array. Why? Assume that we do time analysis of bubble sort and say equation is $T(n) = 2n^2 + 3n + 11$. Now, if we choose to write it as a function of exact input size n' then replacing $n = {n'}/{b}$, we get $T(n') = \frac{2n'^2}{b^2} + \frac{3n'}{b} + 11$. If you see, ratio $\frac{T(n')}{T(n)}$ is in terms of b , which is assumed to be constant. So, if algorithm requires quadratic time then after relaxing input size, it still requires quadratic time. **So, if we are interested in seeing only what kind of function time function is or growth of time function then ignoring exact size and measuring time in terms of other important parameters is fine too**.

- ***So, to mention clearly, time requirement/complexity of an algorithm is a function $\boldsymbol{T(n)}$ of input instance size $\boldsymbol{n}$ or it is $\boldsymbol{T(x, y, z, ..)}$ if we are being relaxed and interested in estimating time in terms of a few important parameters of input instance $\boldsymbol{x, y, z, ...}$. Whatever we choose, it is always better to write down very clearly which parameters are you using to do time and/or space requirement analysis***.

1.1. Performance Analysis of an Algorithm

- If someone asks, what is it that any algorithm requires, apart from input and output, what will you answer? When someone asks, what do you mean by analysing an algorithm, what will you answer?

- Almost surely, everyone will quote ***time*** and ***space*** requirement of a proposed algorithm, as analysis of that algorithm. Time and Space are universal resources required by any algorithm whatsoever. So doing time and space analysis of an algorithm is a necessary activity. But is it true that performance analysis of an algorithm means doing only time and space analysis?

- In general, NO. Performance analysis of an algorithm means resource requirement analysis of an algorithm, for every important resource it uses. Below are a few examples of resources other than time and space:-
  - In Internet routing algorithms, how much network bandwidth is required is one of the important resources to be considered as a performance analysis exercise.
  - In parallel computing, a parallel algorithm is run on multiple independent processors/processing units at the same time, to reduce time, utilizing parallel architecture of system. So, naturally every processor becomes a resource and analysis of number of processors required simultaneously is an important exercise.
  - For a machine learning algorithm, more the data, mostly better it's performance. Not necessarily, if it's a bad algorithm. But, having a lot of data is not enough. Interesting and important is to know how much minimal

amount of data is required for ML algorithm to learn automatically. Training data is an important resource. Training data is called sample. Knowing minimum number of training data required to learn automatically is an important performance analysis exercise. That is, doing ***sample complexity*** analysis of a learning algorithm is another example of analysis other time and space analysis.

- ***Time and Space are, as said, universal resources required by any algorithm. For this course, we will focus only on time and space analysis of an algorithm as a performance analysis exercise***.

Time and Space Complexity

Time Complexity as measured by Program Execution Time

- For us computer engineering people, every algorithm will be implemented as an electronic computer program. What if we do time analysis of that program? Program execution time is dependent on the performance of underlying OS, size of executable created by the compiler and most importantly on the quality of high level source code written by programmer. These factors change from system to system. But then program execution time analysis gives an estimate near to real run time performance. For situations which are real time, for eq. automated trading in markets, program execution time analysis is a must because of sensitivity to time.

- A possible drawback of documenting program execution time analysis is that in order to get the trend of time requirement graph the program has to be executed on different instances of different sizes and multiple instances of same size and characteristic. Such an analysis may be called ***empirical analysis,*** as purpose is to generate data of execution time as a function of input size by choosing input instances carefully and running over several such instances. We have to do curve fitting exercise/regression exercise on execution time data generated. Also, 2 different algorithms having really different time performance may not be separated out by such an empirical analysis. It may happen that difference in time performance starts showing out sharply for very large input size which we might not have considered for doing empirical analysis.

  For eq. to emphasize the point, consider an algorithm **A** whose run-time is say $T_A(n) = 100\,n^2$ and an algorithm **B** with run-time $T_B(n) = n^{2.5}/10$. Algorithm **B** is asymptotically slower than algorithm **A.** But, B is better than A if $T_B(n) \le T_A(n)$ i.e $100\,n^2 \le \frac{n^{2.5}}{10}$. After solving, we get B is better than A for $n \le 10^6$. That means a slower algorithm **B** is as good or better than a faster algorithm **A** as long as input size does not cross 10^6. For larger input size i.e $\ge 10^7$, **A** is substantially better than **B**.

  Suppose these time functions are hidden from us because we are doing empirical analysis and we have not fit any curve over our execution time data plot right now. Say we run programs of both B and A for $10^5 \le n \le 10^7$, we will see that execution time graphs of these 2 programs are not too much different from each other. To see sharp difference we have to have data points for small and large input sizes. Also, we have to make sure that our empirical analysis is ok by making sure that systems characteristics like processor load, circuit slow down does not influence true run time of programs on both systems.
  The example is very superficial. But it gives intuition about possibility of us getting mislead by empirical analysis if we are not careful about choosing input instance sizes, particular input instances having certain structures, to get true run-time performance,

- Hence, we go towards theoretical run-time analysis also i.e run-time analysis of algorithm. But there we have to spend additional efforts because in theoretical analysis we really are not talking about physical time i.e we do not talk in terms of seconds, nano seconds or physical space i.e number of bits or bytes or so. An analogy from physics may help here.
  - We have to choose what is meant by time and space and then theoretical unit of time and unit of space.
  - Before doing that we have to choose the universe in which we are going to measure time and space.
  - The universe for us will be a ***mathematical model of a computer***. We have to chalk down components of our universe i.e mathematical model of a computer. Such a conceptual universe should not be too far away from actual universe i.e an actual electronic computer. It should not be too close to actual electronic computer also. Otherwise, why do theoretical analysis in the first place !!
  - We have to conceptualise our theoretical model by ignoring enough of things from actual electronic computer.
  - Just like we have basic elements in universe, say atoms, we have to choose atoms in our mathematical model of a computer. This means we have to choose those instructions/operations which can be considered as simple and very basic, from which complicated operations can be built, just like complex physical objects are made of elementary atoms.

Lets see briefly what those things may be, so that we can start doing time and space complexity analysis of algorithms.

Theoretical Model of a Computer for performance analysis of an Algorithm:- rough overview

Basic Instruction Set:- Set of Basic Steps

- **Intel i7** processor has nearly **338** individual instructions to operate with. In theoretical analysis, we want to build a model of a computer which is not too far away. But is it really convenient and fruitful to deal with 338 types of basic steps in theoretical analysis? Not really.
- So we say that we have a few basic steps in our model. How many? That number is not that important. Why? You look at algorithm and decide dynamically, looking at algorithm, which operation you want to call

basic i.e atomic and which not. That way you have instruction set, just like i7. Guidelines are not to ignore certain operations altogether and always.

- **Example basic step set** may be {*add, subtract, multiply, divide, assignment, comparison, pointer dereferencing, reading an arbitrary node, pointer arithmetic, dynamic memory allocation of arbitrary size, freeing memory, referencing a random array element, reallocating array, dereferencing higher dimensional pointers*} etc.

- **Just like we do not have sorting as a basic instruction in an actual processor, we should not think of adding sorting arbitrary size array as a basic step in theoretical model too.**

Time and Execution time of a basic step

- **Consider Intel i7** processor to be of frequency nearly 3 GHz. This means a clock cycle is of $0.3\ ns$. So, we can consider $0.3\ ns$ to be one unit of time for i7. Is it really convenient and insightful to deal with exactly this scale of time unit? Not really.

- So, in our model we say that we have unit of time as just 1. Not 1 ns, not $1\ \mu s$, not $1\ s$. Just 1. What does that **1** stand for? We say that time cost 1 is added if any basic instruction/step is used by an algorithm. Simply, we charge 1 for every execution of every basic step used by an algorithm.

- **Now,** i7 may require nearly 3 clock cycles i.e 3 units of time for multiply instruction. That means some basic instructions in i7 require multiple units. If we want, we can add this feature in our model by saying something like algorithm costs $c_{add} \geq 1$ time step cost for *add* basic step, instead of just 1. We can say $c_{comp} \geq 1$ time step cost for a *comparison* step, instead of charging just 1 for its execution . This can be done to show behaviour that even though basic steps are really atomic, each may not cost same. Cost is time, somehow.

- **So, if we have say, for example, $\mathbf{11}$ basic steps, theoretically, in our model of a computer, we can have $\boldsymbol{c_1, c_2, \dots, c_{11}}$, all $\geq \mathbf{1}$ time costs, all different from each other. All these will potentially show in time analysis equation. Eventually, you will not like it much, as a student. You will ask whether tackling all these $\mathbf{11}$ constants in time equation has given any good insight.**

  No matter whether you get bored of such equation, you should not ignore those constants in the first attempt if you want an estimate which is not too bad. In asymptotic analysis, you will have chance to make one approximation by setting $c_1, c_2, \dots, c_{11}$ to 1 i.e assuming that all basic steps have same time cost and that is 1. Furthering asymptotic analysis, you will do further approximation by dropping all constants in equation altogether and just deal with kind of function your time equation is i.e whether it is quadratic equation, cubic equation, polynomial time equation, more than polynomial, exponential etc.

- **If we do not want to do a very tight analysis of time performance then we can ignore time costs of certain basic steps.** This can be done for a few basic steps which are executed rarely or a constant number of times where of course that constant is not large. In such a case, you can ignore time cost from equation, to simplify equation. For eq. there is really no noticeable difference between $T(n) = 4n + 2$ and $T(n) = 4n$. Additional 2 in $T(n) = 4n + 2$ may be because of an assignment step that has been executed twice. You can drop that, for convenience.

  But of course there is a noticeable difference between $T(n) = 3n^2 + 10n + 2$ and $T(n) = 3n^2 + 2$, even for certain large values. Dropping a variable term for convenience takes you away from good estimate, if your intention is to get good estimates.

  But, in asymptotic analysis, there is no difference between above two equations, as we are interested in noticing only that both are quadratic equations and hence for larger and very very large values of n, both equations are not much different. Hence asymptotically 2 algorithms corresponding to above 2 time equations are nearly as good as each other for very very large values of n.

- **What is the point of above discussion?**
  Point is that your time equation depends on how much you are ignoring in it. Your willingness to ignore depends on whether you want nearly exact time performance estimate or good enough estimate or you are doing asymptotic analysis of algorithm's time performance, where asymptotic analysis means for very very large values of n i.e analyzing $\lim_{n \to \infty} T(n)$.

  Read this entire block again, if you have confusion about time costing.

Finally:- Calculating Time Complexity for an input instance

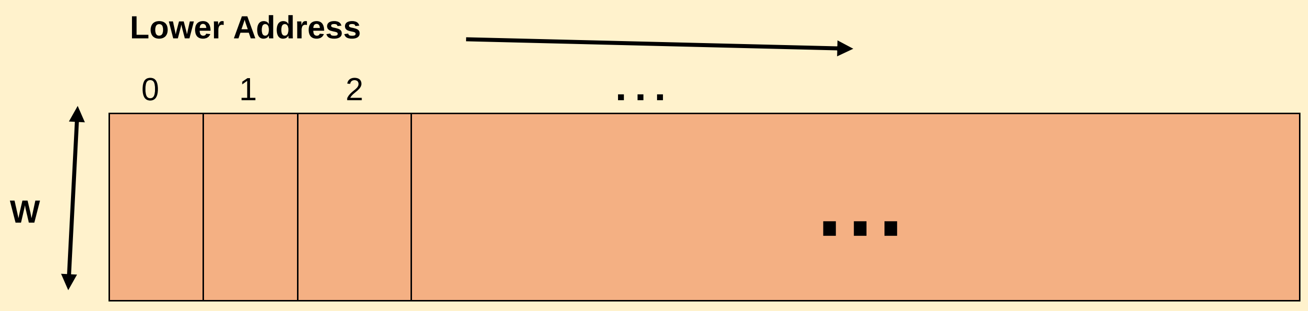
- For an algorithm, say A , Decide parameters of input instance in which you want to measure time performance. i.e you may choose to measure time as a function of input size absolutely or as seen above in case of sorting, you may not care about exact size but measure time in terms of number of elements in array. Decide such parameters, say $m, n, p, q, \dots$.

- Decide basic steps of algorithm. From above subsection, decide time cost of each basic step of algorithm i.e decide whether you want to charge different time units for different steps or charge 1 for each execution of each basic step. Decide that.

- **Finally, $\boldsymbol{T_A(m, n, p, \dots) = \sum_{1 \leq i \leq z} f_i * c(s_i)}$, where there are say $\boldsymbol{s_1, s_2, \dots, s_z}$ basic steps and $\boldsymbol{c(s_i)}$ is the time cost per execution you decided of basic step $\boldsymbol{s_i}$ and $\boldsymbol{f_i}$ is the number of times step $\boldsymbol{s_i}$ is executed throughout algorithm execution.** You will mostly set $c(s_i) = 1$ for simplifying equation.

- As discussed in previous sub section, once you get time equation, you may approximate that equation depending on whether you are doing asymptotic analysis or not.

Memory i.e Space



Random Access Machine memory model

- **Electronic computer memory comes** as an array of typically 32 or 64 bit size words. We measure on location chunk as a ***word***. We can add this feature in theoretical model of a computer, saying one memory location is of constant size, say, w , for some constant w. How large is w? Does not matter as long as it is a constant. That saves us from fraction calculations.

- **Just like von-Neumann** stored program-data memory model, we call it as RAM standing for Random Access Machine (not random access memory), which stores theoretical algorithm executable in this memory.

- **Here comes a bad assumption:-** We assume that no matter how large is a particular data element accessed by a basic step like add, compare etc, it takes only 1 or a constant time cost to read that element no matter how large it really is. We basically assume that all the individual data elements which come as a part of input instance, are constant in size i.e each data elements either fit in 1 word in RAM requiring w size and hence require 1 time step cost to read OR require at most c such locations in RAM requiring at most c time steps to read a data element, for some constant c.

  Is this really true? Does $compare\ 2, 19$ require same amount of time as $compare\ 2, 19^{19}$? NO. Size of an operand really matters to the time requirement of a basic theoretical instruction also. But lets look at how much actual read time contributes to time complexity equation.
  If in $compare\ a, b$, data elements require c_1 and c_2 words in RAM. Both a and b will require $c_1$and c_2 time steps to read respectively, assuming 1 word can be read in 1 time step. If $compare\ a, b$ is executed f times then total time step charge for it will $(c_1 + c_2) \times f \times c_{comp}$. **Whereas,** if we ignore reading time then time charge will be $f \times c_{comp}$. First is accurate, but only constantly different from second. This constant difference shows only constantly in final run-time equation i.e for example, using first results in cubic equation then using second also results in a cubic equation.
  If we are willing to give up on actual constants then assuming that every data element takes only 1 time step every time algorithm reads it. This is not really close to actual performance. But, again, in this course we will be dealing asymptotic analysis, mostly, where not only constants but all lower order terms in an equation are also grossly neglected.

-

Measuring Time and Space Performance:- Examples

NOTE:-

- **This section calculates exact time equation, for the given algorithm/code. So naturally, this task becomes complicated, unrewarding and boring when we have slightly complicated loop structures and also depends on number of nested levels of loops.**
- **Matters become even more twisting if things involve functions like log. But as you read through, we will move away quickly from exact calculation because it really becomes hard to demand exactness with functions like *log*.**
- **But please do go through given examples meticulously once and you will understand why we go for approximate time equations. That will help you appreciate asymptotic analysis and use of operators O, Ω, Θ.**

Below are a few examples of Time and Space performance calculations.

EX 1

Algorithm

```
$Sum = 0;$
For( $i = 1; i \leq n;\ i = i + 1$)
    For( $j = 1; j \leq n; j = j + 1$)
        $sum = sum + 1;$
```

<u>ANSWER</u>:-

Let us assume that we charge time and space for ***every*** elementary operation in the above algorithm. Lets decide *assignment, comparison, increment* to be elementary operations. Let us assume that $c_+,\ c_\leq,\ c_=$ be constant time units required per execution of increment, comparison and assignment operations respectively.
Whenever we can do exact calculation of time and space requirements, we should go for it. When formulas/calculations become involved, we can do some approximation and decide if approximation is good or not. We do line

by line analysis where we calculate total time spent in executing a particular line throughout. Then we add costs for all lines giving total time spent for algorithm.
Below we calculate and mention time spent for each line throughout execution of algorithm.

1 $Sum = 0;$	c_+
2 For($i = 1; i \le n;\ i = i + 1$)	$c_=(1+n) + \ c_\le\,(n+1) + c_+ n$
3 For($j = 1; j \le n; j = j + 1$)	$n(\,c_=(1+n) + \ c_\le(n+1) + \ c_+ n\,)$
4 $sum = sum + 1;$	$n\,(\,c_=\,n + \ c_+ n\,)$

Explanation of time costs:-

Cost of Line 1 is just $\boldsymbol{c_+}$ since it contains only one elementary instruction with cost c_+.

Cost of Line 2 involves initializing i once i.e one assignment, followed by $\boldsymbol{n}$ comparisons evaluated TRUE and 1 evaluated as FALSE, followed by updating i which involves $\boldsymbol{n}$ times addition and n assignments.

Cost of Line 3:- 3 involves 1 initialization, followed by n comparisons evaluated TRUE and 1 evaluated as FALSE, followed by updating $\boldsymbol{j}$ which involves $\boldsymbol{n}$ times addition and n assignments. ***But***, this whole sequence of operations occurs $\boldsymbol{n}$ times, as outer for loop enters it's body $\boldsymbol{n}$ times. Hence, total time spent is $\boldsymbol{n}$ times the time spent in one cycle.

Cost of Line 4:- involves one addition followed by one assignment which happens n times. But that is time spent in one full execution of j for loop. j for loop itself executes n times. Hence total spent in line 4 is n times the time spent in one full execution.

Hence,
$T(n) = c_+ + c_=(1+n) + \ c_\le\,(n+1) + c_+ n + \ n(\,c_=(1+n) + \ c_\le(n+1) + \ c_+ n\,) + n\,(\,c_=\,n + \ c_+ n\,)$

$$\begin{aligned} T(n) &= c_+(2n^2+n+1) + c_=(2n^2+n+1) + \ c_\le\left(n+1+n(n+1)\right) \\ &= \boldsymbol{c_+(2n^2+n+1) + c_=(2n^2+n+1) + \ c_\le\,(n^2+2n+1)} \\ &= \boldsymbol{n^2(2\,c_+ + 2c_= + c_\le) + n(c_+ + c_= + 2c_\le) + c_+ + c_= + c_\le} \qquad \textbf{Eq(1.1.1)} \end{aligned}$$

- Now, time complexity has been expressed as a function of n, as we can see that n decides the number of operations, so far. That explains $T(n)$ part.
- Also, we see that, $T(n)$ is a quadratic equation with exact coefficients as given in equation.
- Above is an exact time equation for the above algorithm.

EX 2

Algorithm

```
Sum = 0;
For( i = 1; i ≤ n; i = i + 1)
     For( j = 1; j ≤ i; j = j + 1)
          sum = sum + 1;
```

<u>ANSWER</u>:-
Below we calculate and mention time spent for each line throughout execution of algorithm.

1 $Sum = 0;$	c_+
2 For($i = 1; i \le n;\ i = i + 1$)	$c_=(1+n) + \ c_\le\,(n+1) + c_+ n$
3 For($j = 1; j \le i; j = j + 1$)	$\sum_{i=1}^{n}(\,c_=(1+i) + \ c_\le(i+1) + \ c_+ i\,) = \left[\frac{(n+1)(n+2)}{2} - 1\right][c_= + c_\le] + \frac{c_+ n(n+1)}{2}$
4 $sum = sum + 1;$	$\sum_{i=1}^{n} i\,(\,c_= + c_+\,) = \ (\,c_= + c_+\,)\frac{n(n+1)}{2}$

Explanation of Time Cost:-

Explanation remain almost same as in Ex.1 except costing for Line 3 and Line 4.

Cost of Line 3:- 3 involves 1 initialization, followed by $\boldsymbol{i}$ comparisons evaluated TRUE and 1 evaluated as FALSE, followed by updating $\boldsymbol{j}$ which involves $\boldsymbol{i}$ times addition and $\boldsymbol{i}$ assignments. ***But***, this whole sequence of operations occurs $\boldsymbol{n}$ times, as outer for loop enters it's body $\boldsymbol{n}$ times. This time we can not simply say overall cost is n times cost of one execution, because time required at each execution is different. Hence we sum the time requirement of line 3 over $1 \le i \le n$. This produces overall time requirement of Line 3 as given in above formula.

Cost of Line 4:- involves one addition followed by one assignment and each of these happens i times. Hence cost of Line 4, given i is $i\,(\,c_= + c_+\,)$. But that is time spent in one full execution of j for loop for one value of i. And we know $1 \le i \le n$. Hence we sum time spent for each value of i, producing formula given above.

Hence,
$$T(n) = c_+ + c_=(1+n) + \ c_\le\,(n+1) + c_+ n + \left[\frac{(n+1)(n+2)}{2} - 1\right][c_= + c_\le] + \frac{c_+ n(n+1)}{2} + (\,c_= + c_+\,)\frac{n(n+1)}{2}$$

$$= \boldsymbol{n^2\left(c_= + c_+ + \frac{c_\le}{2}\right) + n\left(2c_= + 2c_+ + \frac{3}{2}c_\le\right) + c_+}$$ **Eq(1.1.2)**

- Now, time complexity has been expressed as a function of n, as we can see that n decides the number of operations, so far. That explains $T(n)$ part.
- Also, we see that, $T(n)$ is a quadratic equation with exact coefficients as given in equation.
- Above is also an exact time equation for the above algorithm.

EX 3

Algorithm

```
$Sum = 0;$
For( $i = 1; i \le n;\ i++$)
    For( $j = 1; j \le i;\ j++$)
        For( k= $1; k \le j;\ k++$)
            $sum = sum + 1;$
```

<u>ANSWER</u>:-

Lets calculate exact time equation of above piece of code and convince ourselves how complicated algebra becomes very quickly. Below we calculate and mention time spent for each line throughout execution of algorithm.

1 $Sum = 0;$	$c_=$
2 For($i = 1; i \le n;\ i++$)	$c_=(1+n) + c_\le(n+1) + c_+ n$
3 For($j = 1; j \le i;\ j++$)	$\sum_{i=1}^{n}(c_=(1+i) + c_\le(i+1) + c_+ i) = \left[\frac{(n+1)(n+2)}{2} - 1\right][c_= + c_\le] + \frac{c_+ n(n+1)}{2}$
4 For(k= $1; k \le j;\ k++$)	$\sum_{i=1}^{n}\sum_{j=1}^{i}(c_=(1+j) + c_\le(j+1) + c_+ j)$
5 $sum = sum + 1;$	$\sum_{i=1}^{n}\sum_{j=1}^{i} j(c_= + c_+) = (c_= + c_+)\sum_{i=1}^{n}\sum_{j=1}^{i} j = (c_= + c_+)\sum_{i=1}^{n}\frac{i(i+1)}{2}$

Hence,

$$\boldsymbol{T(n) = c_= + c_=(1+n) + c_\le(n+1) + c_+ n + \left[\frac{(n+1)(n+2)}{2} - 1\right][c_= + c_\le] + \frac{c_+ n(n+1)}{2} + Line4 + Line5}$$

Now,

$$Line4 = \sum_{i=1}^{n}(c_= + c_\le)\sum_{j=1}^{i}(1+j) + \sum_{i=1}^{n}c_+\sum_{j=1}^{i} j = \sum_{i=1}^{n}(c_= + c_\le)\left[\frac{(i+1)(i+2)}{2} - 1\right] + \sum_{i=1}^{n}c_+\frac{i(i+1)}{2}$$

$$= \frac{n^3}{6}[c_= + c_\le + c_+] + \frac{n^2}{4}[c_= + c_\le + 2c_+] + \frac{n}{12}[4c_+ - 11c_= - 11\,c_\le]$$

$$Line5 = \frac{(c_= + c_+)}{12}(2n^3 + 9n^2 + 7n)$$

Putting above costs into grand equation, we get an exact time equation $T(n)$, which is not written here to save my time and sanity.

We see that, $T(n)$ is a cubic equation i.e a polynomial equation of degree 3.

From **Ex.3** we clearly some to know that with code as simple as a 3 nested level deep loops, exact calculations involve lot of time consuming calculations.

Insertion Sort

1. Strategy

Notes

2. Informal Algorithm

3. Initial Algorithm

```
Insertion Sort (A, n)
1  For  $k = 2$ to n
2       i = k − 1;                        // we want to fix A[k]
3       While i > 0 and A[i] > A[i + 1]   //  Go backwards starting from current position
4            $Swap(A[i+1], A[i])$;        // A[k] will move backwards through swaps, whenever required
5            $i = i - 1$;
```

Algo0 Insertion Sort

- Swap is ***write intensive*** operation, taking 3 assignment operations. There are at least 3 ways to implement swap. All require 3 assignments but XOR based swap is fastest, *avoiding temporary variable*, in terms of number of actual clock cycles required to evaluate rvalue of assignment operation. But independent of actual implementation of swap, 3 assignment operations are strictly required to swap.

- If $A[k]$ is not at its correct position in $A[1 \dots k]$ then let, correct position of $A[k]$ in $A[1 \dots k]$ is say $1 \leq x < k$. Algorithm takes $(k-1) - x$ swaps and consequently $3(k-1-x)$ assignments to get $A[k]$ at it's correct position in $A[1 \dots k]$. Basically, algorithm literally moves element $A[k]$ one place left whenever it finds chance to do so.

- Imagine that in a theatre, in a row of 10 seats, your seat number is say 3 and everyone from seat number 4 onwards are seated mistakenly to seat numbers 1 less than their correct seats. You come and you first seat at seat number 10. To move to seat number 3, you are literally swapping seat with people starting with person seating at 9 and hence you are kind of sliding towards your actual seat number 3. Is it really required to do so? How awkward and frustrating it would be to you and everyone around?

- You can reduce amount of work by asking everyone from seat number 9 to seat number 3 (*which they have to*) to shift to seat number 1 more than they are right now. Once they do that, seat number 3 is vacant and you get there by making only 1 move yourself.

- This reduces amount of movements quite much. But you are at seat number 10 already. So, how can person at seat number 9 move to seat number 10? This can be done if you get up and stand outside the row.

- Implementing these changes to above algorithm we get the following algorithm. $A[k]$ is playing your role and $[1 \dots k]$ are seat numbers.

Swap(x,y)
{
x=x+y; y=x-y;
x=x-y;
}
Swap(x,y)
{
x=x^y; y= x^y;
x= x^y;
}
Swap(x,y)
{
temp=x; x=y;
y= temp;
}

Did you notice that either x or y will be A[k] i.e swapping involves element A[k] again and again ?

4. Refined Algorithm After avoiding Swap operations

```
Insertion Sort (A, n)
1  For  $k = 2$ to n
2       currentval= A[k];              //  Insert A[k] into it's proper position in A[1…k-1]
3       i = k − 1;
4       While i > 0 and A[i] > currentval   //  Go backwards starting from current position
5            $A[i+1] = A[i]$;          //  Shift elements greater than currentval 1 position right, to make place
6            $i = i - 1$;
7       $A[i+1] = currentval$;         //  Correct position found…copy currentval to this position
```

Algo1 Insertion Sort

A Boolean operator is short-circuited if second operand is not evaluated, if 1st operator decides the result.

*What if **and** operator in line 4 is not short-circuited?*

Have you already noticed redundant assignment that may happen in some input instance? Spot out the line number and answer in which input instance redundant assignment operation may occur. Also figure out if you can get rid of that redundant operation without adding other cost to algorithm. If adjustments you suggest introduce other cost to get rid of this redundant operation then adjustment will turn to be useless as overall adjustment may cancel out.

5. Example

Consider a given below array A of 8 elements.

	$currentval$	$A[k] = A[2] = -18$						
A	-16	-18	-11	16	-19	-1	12	-10
Index	1	2	3	4	5	6	7	8
	$i = 1$							

6. Correctness

7. Is it a Stable algorithm? YES

s

- For simplicity, without loss of generality, assume that one element of array is duplicated once i.e $A[x] = A[y] = key$, for some $1 \le x < y \le n$. In line 4, element at index x will be fixed first.
- Coming to $k = y$, algorithm tries to fix $A[y]$ in already sorted array $A[1 \dots y-1]$. Let position of old $A[x] i.e\ key$ in $A[1 \dots y-1]$ be index $1 \le p < y$. Due to correctness of the algorithm, sorted position of $A[y]$ will turn out to be $p+1$ since test condition at line 4 works on $>$ and not on $\ge$.
- This proves that in sorted order $input\ element\ A[x]$ comes before $input\ element\ A[y]$. Hence, algorithm is stable.

8. Space and Time Complexity Analysis :- Best and Worst Case

Time Complexity Analysis

Consider the *constant* cost of each elementary/important operation to be as below:-

$\boldsymbol{c_a}$=*cost of an assignment/arithmetic operation*, $\boldsymbol{c_c}$ =*cost of a comparison*

```
Insertion Sort (A, n)                                  COST

1  For  $k = 2$ to n                                   $c_a(n+1) + c_c n$
2        currentval = A[k];                           $c_a(n-1)$
3        i = k − 1;                                   $c_a(n-1)$
4        While  i > 0 and A[i] > currentval           $2c_c \sum_{k=2}^{n} t_k$,  $t_k$ = count of test condition evaluated in $k^{th}$ iteration
5              $A[i+1] = A[i]$;                       $c_a \sum_{k=2}^{n}(t_k - 1)$, $t_k - 1$ times test condition is TRUE in $k^{th}$ iteration
6              $i = i - 1$;                           $c_a \sum_{k=2}^{n}(t_k - 1)$
7        $A[i+1] = currentval$;                       $c_a(n-1)$
```

CT1.1 Costing of Insertion Sort

As seen in CT1.1, the running time equation of Insertion sort becomes

$$T(n) = c_a(n+1) + c_c n + 3c_a(n-1) + 2c_c \sum_{k=2}^{n} t_k + 2c_a \sum_{k=2}^{n} (t_k - 1)$$

$$= (2c_c + 2c_a)\sum_{k=2}^{n} t_k + (2c_a + c_c)n$$

General Run Time Function (1.1)

As seen from above time equation,

- Algorithm is sensitive not only to n (*which should be true*), but also to what instance of size n it is executing on. This is seen by another variable t_k. It basically tells, given an instance, to find correct position of k^{th} element of array, how many steps backwards were taken (*and hence how many elements were copied to right by 1 position*).
- There are 2 very important operations. Given k^{th}element, ***number of comparisons i.e search time*** and ***number of copy to right*** operations done to find correct position and put it there. We may not see significance of both of them together in above given algorithm. But, a slight modification of Insertion sort makes these operations clearly distinct from each other.

- It is true that $1 \leq t_k \leq k$. Below are the reasons.

- Is it possible that $t_k = 1$, for every k ?. **YES**. This happens when given instance is already ***increasingly*** sorted. So, only 1 comparison breaks out of while loop. As a result, no element is copied 1 place to the right to find correct position of k^{th} elements. Putting $t_k = 1$ in Equation(1.1), we get

$T(n) = (2c_c + 2c_a)\sum_{k=2}^{n} 1 + (2c_a + c_{tc})n \quad = \quad (4c_a + 3c_c)n - (2c_a + 2c_c)$ $= An - B$, for constants $A = 4c_a + 3c_c$ and $B = 2c_a + 2c_c$ $T(n) \in \Theta(n)$	**Best Case** (1.2)
Number of ***copy to right*** operations = 0	

As seen above, $T(n)$ becomes a linear equation in n. Hence $T(n) \in \Theta(n)$. **But this situation being a *best case,* time spent is minimum. Hence this is the minimum time requirement for the given algorithm, given any input instance. Hence,**
$\boldsymbol{T(n) \in \Omega(n)}$.

- Max value for t_k? It is $t_k = k$. This happens when given instance is ***decreasingly*** sorted. So, every given k^{th}element finds 1st position of array as its correct position. As a result, $(k-1)$ elements are copied 1 place to the right to find correct position of k^{th} element. Putting $t_k = k$ in Equation(1.1), we get

$T(n) = (2c_c + 2c_a)\sum_{k=2}^{n} k + (2c_a + c_c)n \quad = \quad (c_a + c_c)n^2 + (3c_a + 2c_c)n - (2c_a + 2c_c)$ $= An^2 + Bn - C$, for constants $A = c_a + c_c$ and $B = 3c_a + 2c_c$ and $C = 2c_a + 2c_c$ $T(n) \in \Theta(n^2)$	**Worst Case** (1.3)
Number of ***copy to right*** operations $= \sum_{k=2}^{n}(k-1) = \frac{n(n-1)}{2} = \binom{n}{2}$	

- As seen above, $T(n)$ becomes a ***quadratic*** equation in n. Hence $T(n) \in \Theta(n^2)$. **But this situation being a *worst case,* time spent is maximum. Hence this is the maximum time requirement for the given algorithm, given any input instance. Hence**
$\boldsymbol{T(n) \in O(n^2)}$

Space Complexity Analysis

- As seen from the algorithm, 3 extra memory places are required to maintain algorithm's control variables and maintaining current k^{th} element in *currentval*. ***Assuming, every data element requires a constant amount of memory, say*** $\leq D$ ***number of bytes, for some constant*** D, algorithms requires $\leq 3D \in \Theta(1)$ bytes, independent of size of the input i.e n and instance characteristic (*no matter whether it is best case i/p or worst case i/p*).
- Hence, $S(n) \in O(1)$, or ***most appropriately*** $S(n) \in \Theta(1)$ (*although* $O(1)$ *is not mathematically wrong*)

9. Average case Time Complexity:- An Attempt

10. Input Instance Characteristic and Run time – Relation between Inversions and Run time

- What we have analysed till now is time complexity behaviour of algorithm **ALGO1** from *best*, *worst* and *average* case perspective. We have figured out lower and upper bounds independent of any given input instance. Given an input instance I_i, where of course $|I_i| = n$, we know $T(|I_i| = n) \in \Omega(n)$ *and* $T(|I_i| = n) \in O(n^2)$. These upper and lower bounds are different functions. So there is no way we can bound time complexity function by Θ operator. But we know that algorithm is sensitive to input permutation i.e it takes time depending on how good/bad permutation input is.

- It is natural to ask whether we can find time function $f(|I_i| = n)$ for algorithm, given input instance I_i , such that $T(|I_i| \in \boldsymbol{\Theta}(f(n))$. If so, we will know around exact time for a given input instance, rather than knowing just upper and lower bounds.

- We already know that given k^{th} element $A[k]$, search time and hence copy to right operations depend on how many elements in sorted array $A[1 \dots k-1]$ are greater than $A[k]$ and that decides time complexity function.

In an array A, a pair of indi-Ces (i,j) is an ***Inversion*** *if i<j and A[i]>A[j]*

If every pair (i,j) is an inversion then there can be maximum n(n-1)/2 inversions, since there are at most n(n-1)/2 pairs (i, j)

- In an array A of size n, a pair of indices (i,j) is called as an *inversion*, if $i < j \ and \ A[i] > A[j]$ i.e higher element is at lower index. Let $INV(I_i, n)$ be the count of inversions in given input instance I_i.
$$0 \le INV(I_i, n) \le \frac{n(n-1)}{2}$$

Which instances have 0 or n(n-1)/2 inversions?

- In **ALGO1**, given input instance I_i, for fixing element $A[k]$ while loop of lines **4 to 6** runs exactly as many times as many elements are greater than $A[k]$ in array $A[1 \dots k-1]$ i.e depends on how many indices $1 \dots k-1$ are in inversion with index k. Summing inversions over all values of k, we get $INV(I_i, n)$.

- Also, line **1** executes $n-1$ times. Adding this and above point, we get $T(|I_i|) \in \Theta(\ n - 1 + INV(I_i, n)\ )$.

- Above time equation gives tight bound on run time, given an input instance. If I_i is best case input instance characterized by $INV(I_i, n) = 0$ we get $T(n) \in \Theta(n - 1 + 0)$ i.e $T(n) \in \Theta(n)$. If I_i is worst case input instance characterized by $INV(I_i, n) = \frac{n(n-1)}{2}$, we get $T(n) \in \Theta(\ n - 1 + \frac{n(n-1)}{2})$ i.e $T(n) \in \Theta(n^2)$.

- So, in general, given an input instance I_i with $|I_i| = n$, we have

$$T(n) = \Theta(n + INV(I_i, n)) \qquad (1.4)$$

- We can alternatively write down time complexity equation as a 2-parameter equation

$$T(n, INV(I_i, n)\ ) = \Theta(n + INV(I_i, n)) \qquad (1.4)$$

Input size= n	$T(n)$	Number of Comparisons(*among array elements*)	Copy to Right Ops	Example Instance Characteristic	How many Instances?
BEST Case	$\Omega(n)$	$n-1$	0	Increasingly Sorted	Exactly 1 instance requires Maximum time. But > 1 instances have time function $\in \Omega(n)$. For eg. increasingly sorted but just first 2 elements swapped.
WORST Case	$O(n^2)$	$\binom{n}{2} = \frac{n(n-1)}{2}$	$\binom{n}{2} = \frac{n(n-1)}{2}$	Decreasingly Sorted	Exactly 1 instance requires Maximum time. But > 1 instances have time function $\in O(n)$. For eg. same as in best case explanation.
Instance I_i	$\Theta(n + INV(I_i, n))$	$n - 1 + INV(I_i, n)$	$INV(I_i, n)$	---	---

11. Speeding up ***Search*** time for finding correct position…***possibilities***

Algo1 fixes correct position of k^{th}element, in an already sorted array $A[1 \dots k-1]$ using *linear search* backwards. Since, linear search requires $O(k-1)$ comparisons, we can speed up search *in the worst case* using *binary search* on $A[1 \dots k-1]$ which takes $O(\log\ (k-1))$ comparisons.

***Given sorted** array B of size n, and a search key say x, binary searc requires O(log n) comparisons.*

```
Insertion Sort (A, n)
1  For  $k = 2$ to n
2       $correctpost = Binarysearch(A, 1, k, A[k])$;   //Find correct position of A[k] using binary search on sorted A[1…k]
3       if( $correctpos \neq k$ )                       // Need to work only if $k$ is not $A[k]$'s correct position already
4           currentval = A[k];                         //  Insert A[k] into it's proper position in A[1…k-1]
5           i = k − 1;
6           While i ≥ currentpos                       //  Go backwards starting from current position
7               $A[i+1] = A[i]$;                       //  Shift elements higher than currentval to 1 position right, to make place
8               $i = i - 1$;
9           $A[i+1] = currentval$;                     //   Correct position found…copy currentval to this position
```
Algo2 Insertion Sort with Binary Search

- How would binary search affect number of comparisons? Will there be any change in asymptotic time complexity of insertion sort using binary search?
- Consider fixing $A[k]$. **ALGO1** requires worst case $O(k)$ comparisons due to linear search. Above algorithm would require worst case $O(\log k)$ comparisons to find correct position of $A[k]$.
- Summing search time in the worst case of line 2 in above algorithm, over $2 \le k \le n$, we get worst case search comparisons $\sum_{k=2}^{n} O(\log k) = O(\log n!) = O(n \log n)$.

O(log n!)=O(n log n)

- Hence, it is sure that **ALGO3** is better than **ALGO1** ***in terms of number of comparisons in worst case***.
- But what about *copy 1 place to right* assignment operations? That remains same in both the algorithms no matter whether it is the best case input instance or worst case input instance. So, in worst case run time is still $O(n^2)$ due to $\frac{n(n-1)}{2}$ number of inversions. So asymptotic time complexity of **ALGO3** is same as $T(n) = O(n^2)$.
- **The worst case for ALGO1 remains worst case for ALGO2 and worst case asymptotic run time is hence $\boldsymbol{T(n) = O(n^2)}$.**
- What about best case of **ALGO1** given as input to **ALGO2**? This time increasingly sorted array input is not best case input for **ALGO2**. Lets see why.
  To find correct position of $A[k]$ in $A[1 \dots k-1]$, **ALGO1** requires 1 comparison in best case because $A[k-1] < A[k]$ so linear search terminates with 1 comparison. But **ALGO2** requires $\log(k-1)$ comparisons before finding out that $A[k-1] < A[k]$. This is so because binary search starts from middle index always. So in case input is already increasingly sorted then for **ALGO2** $\Theta(\sum_{k=2}^{n} \log(k-1) = \log(n!)) = \Theta(n \log n)$ comparisons and hence $T(n) = \Theta(n \log n)$. **Hence, increasingly sorted input DOES NOT result into same asymptotic time complexity for binary insertion sort.**

- So what is the best case time complexity of binary insertion sort? There are 2 components. Search time and number of inversions. Best case instance should be the one which minimizes both together. But there is a slight problem here. To search correct position of $A[k]$,when will comparisons be reduced to minimum i,e 1 or 2 or some constant? This happens if $A\left[\frac{1+k-1}{2}\right] < A[k]$ i.e correct position is almost the middle index. So search terminates with $\Theta(1)$ comparisons. But this means nearly half the elements need to move 1 place to right because they are greater than $A[k]$ i.e there are nearly $\frac{k}{2}$ copy to 1 place right assignments. Summing over all $2 \le k \le n$ we get $\Theta\left(\sum_{k=2}^{n}\left(1+\Theta\left(\frac{k}{2}\right)\right)\right) = \Theta\left(n-1+\Theta(n^2)\right) = \Theta(n^2)$. That's as bad as worst case asymptotically. This happens because of large number of inversion present in input.

- So lets see when can we have minimized inversions. This happens only when input is already sorted. But we have already seen that run time in this case is $T(n) = \Theta(n \log n)$.

- So clearly we can not have best case input which minimizes both, number of comparisons and number of inversions. Why? If an input has minimum i.e $\Theta(1)$ comparisons (*not necessarily 1 or 2*) resulting out of binary search. That means correct position of $A[k]$ is nearly middle position of $A[1 \dots k-1]$. This means, as seen above, nearly $\frac{k}{2}$ shifts are required, resulting in total run time of $\Theta(n^2)$. And if an input has minimum i.e 0 or $\Theta(1)$ inversions then it means correct position of $A[k]$ is near to the end of $A[1 \dots k-1]$. As a result, binary search would require $O(\log k - 1)$ i.e high comparisons to reach to such an index. Clearly, we can not have any input array where we have $\Theta(1)$ comparisons and $\Theta(1)$ inversions simultaneously.

- Considering above discussion, we can figure out that best case running time of **ALGO2** belongs to $\Theta(n \log n)$.

We can summarize ALGO1 and ALGO2 behaviour in below table,

	Best Case Asymptotic Run time	**Worst Case Asymptotic Run time**	**Minimum Number of Comparisons in Best case**	**Maximum Number of Comparisons in Best case**	**Min number of shift right operations**	**Max number of shift right operations**	**Best case input instance**	**Worst case input instance**
Linear search Insertion Sort	$\Omega(n)$	$O(n^2)$	$n-1$ Happens in best case i/p	$\frac{n(n-1)}{2}$ Happens in worst case i/p	0 Happens in best case i/p	$\frac{n(n-1)}{2}$ Happens in worst case i/p	Increasingly sorted array	Decreasingly sorted array
Binary Insertio sort	$\Omega(\mathrm{n} \log n)$	$O(n^2)$	$n-1$ Such an i/p is worst case i/p	$n \log n$ Such an i/p is best case i/p	0 Such case is best case i/p	Such case is worst case i/p	Increasingly sorted array	Decreasingly sorted array

In a nutshell, we can say that as input instances are nearly as worst then **ALGO2** is definitely better (not asymptotically better) than **ALGO1** since number of comparisons are bound to $O(n \log n)$ in **ALGO2**. Also, if in certain situations cost of a comparison is greater than cost of a copy right assignment then **ALGO2** has an upper hand. This may happen when array elements are not $\Theta(1)$ size, as assumed throughout this whole document. If elements are of variable size then a single comparison itself take time proportional to element size. In such a case, comparisons should be avoided as much as possible. But it all depends on how variable size element collection is handled either as an array or using some other data structure.

12. Block Copy to Right Assignments

As seen in either **ALGO1** or **ALGO2**, given $A[k]$, elements in $A[1 \dots k-1]$ greater than $A[k]$ are copied 1 place to right individually. We can rewrite these copy right assignments as a block write operations, as given below.

```
Insertion Sort (A, n)
1    For  $k = 2$ to n
2         currentval= A[k];
3         i = k − 1;
```

```
4        While i > 0 and A[i] > currentval      //counting number of elements greater than $A[k]$
6              $i = i - 1;$
7        $j = i + 1;$
8        While $j < k$                          // copy to right greater elements
9               $A[j + 1] = A[j];$
10        $A[i + 1] = currentval;$
```

Algo3 Insertion Sort with Block copy right

- Is there any difference in asymptotic time complexity or even in exact number of steps required compared to **ALGO1**? **NO**. As an algorithm, **ALGO3** is exactly as efficient as **ALGO1.**
- **ALGO3** turns out to be slightly *more efficient as a computer program* than **ALGO1** implemented as a program. This happens only if programming language has efficient low level functions to do block memory copy. If such a facility is available then ***while*** loop of line 8 and 9 can be converted to a single efficient programming language statement.
- **Example:-** ***memmove()*** function in C/C++

13. Solved Exercises

1 Consider sorting an n size array of strings where length of each string is $\Theta(L)$, for some fixed, unknown and very large $L \in \mathbb{N}$ i.e $\forall_{1 \le i \le n} |A[i]| \in \theta(L)$, where $|S|$ means size of element S.
Derive time and space complexity behaviour of ***Insertion Sort algorithm (Algo1)*** for this array.

```
Insertion Sort (A, n)                                   COST

1  For  $k = 2$ to n                                    $c_a(n+1) + c_c n$
2          currentval = A[k];                           $c_a(n-1)$
3          i = k − 1;                                   $c_a(n-1)$
4          While i > 0 and A[i] > currentval            $2c_c \sum_{k=2}^{n}(t_k \Theta(L))$
5                 $A[i+1] = A[i];$                       $c_a \sum_{k=2}^{n}(t_k - 1)$
6                 $i = i - 1;$                           $c_a \sum_{k=2}^{n}(t_k - 1)$
7          $A[i+1] = currentval;$                        $c_a(n-1)$
```

ANSWER:-

- As in Equation(1.1), analysis goes same way. Difference lies in costing of line 4, as shown above.
- Since array elements don't have constant size, a comparison of array elements is not $\Theta(1)$, as assumed previously in Algo1. Even if L may be fixed, if it is too large then comparing two L sized elements can not be considered to be $\Theta(1)$ time operation.
- Hence, a comparison of two $\Theta(L)$ size elements takes $\Theta(L)$ time. This has been reflected in cost of line 4 above.

$$T(n) = c_a(n+1) + c_c n + 3c_a(n-1) + 2c_c\mathbf{\Theta}(\boldsymbol{L}) \textstyle\sum_{k=2}^{n} t_k + 2c_a \sum_{k=2}^{n}(t_k - 1)$$
$$= (2c_c\mathbf{\Theta}(\boldsymbol{L}) + 2c_a)\sum_{k=2}^{n} t_k + (2c_a + c_c)n$$

- We can use Best and Worst case analysis done to Equation 1.1 as it is with cost of line 4 modified as given.

$T(n) = (2c_c\mathbf{\Theta}(\boldsymbol{L}) + 2c_a)\sum_{k=2}^{n} 1 + (2c_a + c_{tc})n \;=\; (4c_a + 3c_c\mathbf{\Theta}(\boldsymbol{L}))n - (2c_a + 2c_c)$ $= \mathbf{\Theta}(\boldsymbol{L})\boldsymbol{n} - B$, for constant $B = 2c_a + 2c_c$ and after dropping constant $4c_a$ and recognizing that $3c_c\mathbf{\Theta}(\boldsymbol{L}) \in \mathbf{\Theta}(\boldsymbol{L})$ $T(n) \in \Omega(Ln)$	**Best Case**
$T(n) = (2c_c\mathbf{\Theta}(\boldsymbol{L}) + 2c_a)\sum_{k=2}^{n} k + (2c_a + c_c)n$ $= (c_a + c_c\mathbf{\Theta}(\boldsymbol{L}))n^2 + (3c_a + c_c(\mathbf{\Theta}(\boldsymbol{L}) + 1))n - (2c_a + 2c_c)$ $= \mathbf{\Theta}(\boldsymbol{L})n^2 + \mathbf{\Theta}(\boldsymbol{L})n - C$, for constant $C = 2c_a + 2c_c$ and after recognizing that $c_a + c_c\Theta(L) \in \Theta(L)$ and $3c_a + c_c(\Theta(L) + 1) \in \Theta(L)$ $T(n) \in O(Ln^2)$	**Worst Case**

- above algorithm.

Thtrhgrrt
