

DATA STRUCTURES
AND ALGORITHMS

Reg #01

Before Mids

4th Sem

RAYYAN SHABBIR

BITF 20m535



DATA STRUCTURE AND ALGORITHM:

DATA:

* DATA is defined as Raw facts and figures. (Material)

* It can also be referred as unarranged / unorganized information.

* Everything we see, we perceive is data. (Bird's eye view)

* In simple words, Data is raw material.

* Any data also have some figures → min length
 "no figures" → max length
 → avg count.

There is no conclusion about these raw materials.

NOTE:

computer is a data processor.

→ It takes data as input and process it.

STRUCTURE

DATA

- * Form to manage data.
- * Data is organized data.
i.e. **structures** in any form.
These structures could be simple architecture, form, etc.
- * After organizing the data (which is initially in scattered form), connects we decide appropriate structure for data.
- * Structure is used to organize data in well-organized form.

INFORMATION

DATA

STRUCTURE

- * Structure is blue-print
- * We choose structure wisely.

DATA STRUCTURES

NOTE:

- * If we use good structure, then processing will be efficient.
- * To store data in memory first we decide appropriate structure for data.
- * And overall organization of data would become efficient (efficient processing).

- * Management of data in computer's memory (Main memory / RAM), for further processing.
- * Here, at this stage, data is just arranged but we need to process it using algorithms.

DATA AND INFORMATION

DIFFERENCE

- * Data and information are technically different.

DATA

- * These structures are the storage of data.

- * Step by step execution of your program.

ALGORITHMS

WE PROCESS DATA
by using efficient algorithm.

- * Step by step execution of your program.

* Sequence-wise processing of data.

* Algorithms are tools and techniques for processing data (as well as arrangement of data in computer's memory).

* Algorithms are those programs that stop / cease / hold and gives an output.

* Infinite loops are not Algorithms.

* we follow well-defined Algorithms.

* we select efficient (both) algorithms among data-structures.

- Array (mostly used DS)
- Table

- linked list
- Tree (It is also a graph)

- Stack
- Graphs

- Queue
- Sets

DATA PROCESSING : DATA STRUCTURE

DATA STRUCTURE

b

DATA PROCESSING : DATA STRUCTURE

b

DATA STRUCTURE TYPES/CATEGORIES:

Data structure can be divided into two categories:

- 1) Linear Data Structure
- 2) Non-Linear Data Structure.

1) Linear DS

2) Non-Linear DS

NOTE:

* We assume:

$\boxed{\text{int } a = 5;}$ mostly it takes '2' steps to execute.

- * Time will see after how many steps (time) our program will execute.
- * **Complexity of Algorithm,**

- * Complexity of Algorithm is measured in terms of
 - Time complexity
 - Space complexity

• TIME COMPLEXITY: (No. OF STEPS / STEPS COUNT)

- * The amount of time taken by an algorithm to execute itself.

- * This time is basically the number of steps or step count.

- * Whenever we analyse an algorithm, we do not consider the hardware of it. (Independent of hardware)

* FREQUENCY: (f)

- * How many times program / spec step repeats.

- * Time complexity is independent of hardware, machine or any operating system.

• TIME COMPLEXITY: (TOTAL)

- * It gives exact measure for its execution.
- * If we see about hardware it is called "PERFORMANCE MEASUREMENT".

DENOTATION OF TIME COMPLEXITY:

$$t_{\text{program}}(n) \rightarrow \text{input size}$$

To measure the time complexity (t_n) now, we will multiply the number of steps, i.e., 'S/e' and 'f' and add the answers.

(4) EXAMPLE: We have given a code and we need to count its time complexity:

~~1. No. of test()~~ NOT INCLUDED IN STEP COUNT.

STATEMENT NO.	S/e	f	S/e × f = TOTAL
1-	0	1	0
2-	1	1	1
3-	1	1	1
4-	1	1	1
5-	0	0	0

No. of void test() } → NOT Included in step count as well as in frequency.

1- { → Not execute

2- [int a, b;] → Not matter how many variables declare in one line, we always take its S/e.

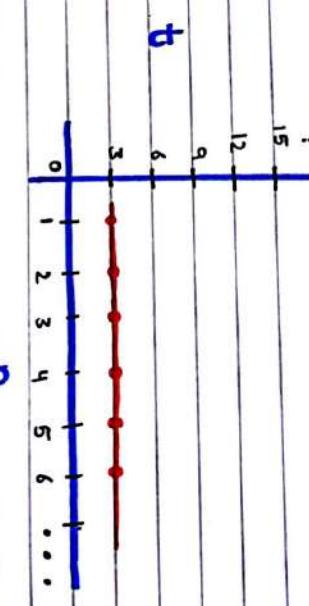
(S.E.) → NOT Included in step count as well as in frequency.

3- Cin >> a >> b;

4- cout << a+b;

5- } → Brackets does not execute

NOTE:
* Brackets and Comments does not execute. So, there S/e is '0'.



We have just considered that these steps are executing in one step only.

GRAPH OF TIME COMPLEXITY OF test().

$$t_{\text{test}(n)} = 3$$

The "GROWTH RATE" of this function is "CONSTANT" for all inputs.

(2)

EXAMPLE :

→ If we are given a loop.

```
for (int i=0; i<5; i++) {
    temp = i + 1;
}
```

As statement	S/E	f	TOTAL
$i = 0$	1	$5+1 = 6$	6
$i = 1$	1	5	5
$i = 2$	1	6	6
$i = 3$	1	5	5
$i = 4$	1	6	6
$i = 5$	1	5	5

* A loop will always executes extra time bcz it will also check for (last) false comparison.

IN ABOVE CASE:

i CHECK RUN

1

2

3

4

5

X

* We can also use formula to calculate frequency of loop statement.

NOTE: $i = 1$ ALSO "CONST"

GROWTH RATE OF FUNCTION IS

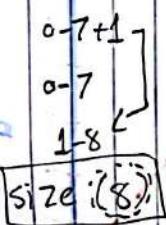
FORMULA: $\Rightarrow UL - LL + 1$

OR

$$5 - 0 + 1 = 6$$

$O(1) = 1$ result

* An algorithm / program which takes constant time for its execution are the best one.



(3) EXAMPLE:

OPERATION	COUNT	CONSTANT
for(i=0; i<7; i++)	7	1
temp = i + 1;	7	1

Lower Upper Limit	S/E	f	TOTAL
0-7	1	7	7
1-7	1	7	7
2-7	1	7	7
3-7	1	7	7
4-7	1	7	7
5-7	1	7	7
6-7	1	7	7
7-7	1	7	7

$O(7) = O(1)$

GENERALIZED Time Complexity OF Loop:

⑤ Example: Considering two loops.

④ Example: Consider a general loop.

for($i=0$; $i < n$; $i++$) \rightarrow $n+1$
 for($j=0$; $j < n$; $j++$) \rightarrow $n+1$ Hence $t = n+1$

temp; \rightarrow n
 $t(n) = \frac{2n+1}{2} + 1$

$$t(n) = 2n+1$$

\rightarrow This program has linear growth rate.

Bcz 'n' (input)'s power is '1'.

We mostly only see 'n' and its power.

• When $t=1$ $t(1) = 3$

• When $t=4$ $t=4+0=4$

* Output increases as t value $t(4) = 9$
 $\delta = \text{of } t + \text{input } \delta$ increases.

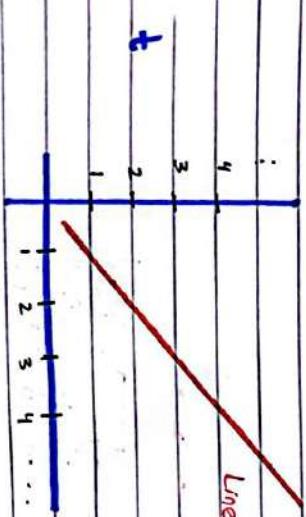
$$t(n) = 2n+1$$

\rightarrow $t(n) = 2n+1$

This program has linear growth rate.

• When $t=100$ $t(100) = 201$

Linear growth.



$$t_{\text{calculate}}(n) = t_{\text{add}}(n) + t_{\text{sub}}(n) + t_{\text{mul}}(n)$$

$$t_{\text{calculate}}(n) = t_{\text{add}}(n) + t_{\text{sub}}(n) + t_{\text{mul}}(n)$$

• If we have function calls we calculate the time complexity of functions (that are being called) separately, and then add it in overall time complexity of function.

LIKE:

(1) Calculate $t(n)$

add(); \rightarrow $t_{\text{add}}(n)$

sub(); \rightarrow $t_{\text{sub}}(n)$

mul(); \rightarrow $t_{\text{mul}}(n)$

$t(n) = t_{\text{add}}(n) + t_{\text{sub}}(n) + t_{\text{mul}}(n)$

⑥

EXAMPLE: OF `main()` having a function call.

NESTED Loops:

We will see how to calculate time complexity of nested loops.

int main()
{
 int a=3, b=4;
 int s = add(a, b);
 cout << s; // s=7
}

	(sle x f)	TOTAL
1- {	0	0
2- int a=3, b=4;	1	1

3- int s = add(a, b); → $s = a + b$

① Example: consider a nested loop.

for ($i=0$; $i < n$; $i++$) → $n+1$

for ($j=0$; $j < n$; $j++$) → $n \times (n+1) = n^2+n$

(temp_i)

→ $n \times n = n^2$

4- cout << s; → $\text{cout} \times n$

5- return 0; → 0

6- } → 0

$t_{main}(n) = 4 + t_{add}(n)$

* This is the main statement we mostly interested (calculate time complexity) of this algorithm.

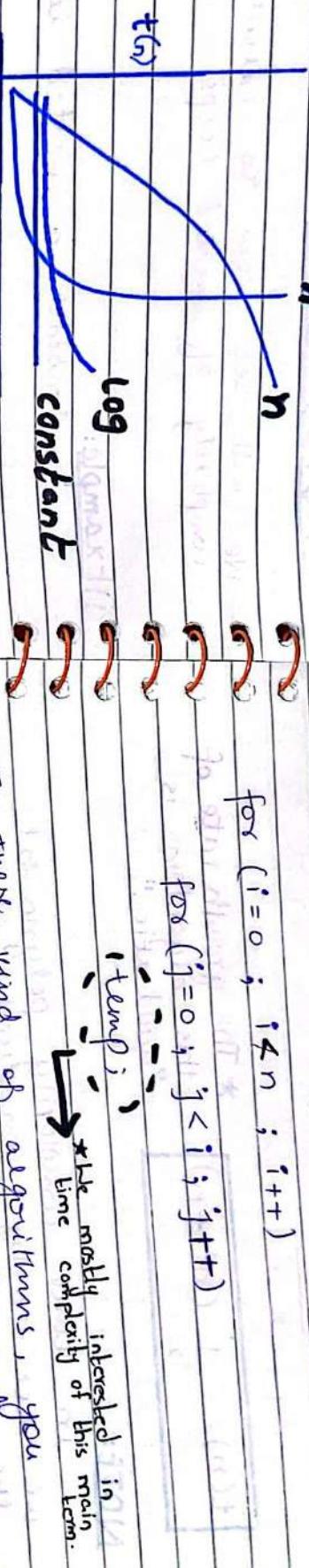
* The growth rate of this algorithm is "quadratic".

It is not linear.

CHECK:

GRAPH OF TIME COMPLEXITIES ($n, n^2, \text{constant}$):

② Example: consider given loop.



- * In these kind of algorithms, you should ignore the loops and only consider the deepest statement i.e temp (in given case)

* For finding time complexity:

CHENNAI: i no. of executions (temp)

* Every statement inside loop executes one time less than the loop.

0 (does not run)

1 (does not run)

2 (does not run)

3 (does not run)

4 (does not run)

5 (does not run)

6 (does not run)

7 (does not run)

8 (does not run)

9 (does not run)

10 (does not run)

11 (does not run)

12 (does not run)

13 (does not run)

14 (does not run)

15 (does not run)

16 (does not run)

17 (does not run)

18 (does not run)

19 (does not run)

20 (does not run)

21 (does not run)

22 (does not run)

23 (does not run)

24 (does not run)

25 (does not run)

26 (does not run)

27 (does not run)

28 (does not run)

29 (does not run)

30 (does not run)

31 (does not run)

32 (does not run)

33 (does not run)

34 (does not run)

35 (does not run)

36 (does not run)

37 (does not run)

38 (does not run)

39 (does not run)

40 (does not run)

41 (does not run)

42 (does not run)

43 (does not run)

44 (does not run)

45 (does not run)

46 (does not run)

47 (does not run)

48 (does not run)

49 (does not run)

50 (does not run)

51 (does not run)

52 (does not run)

53 (does not run)

54 (does not run)

55 (does not run)

56 (does not run)

57 (does not run)

58 (does not run)

59 (does not run)

60 (does not run)

61 (does not run)

62 (does not run)

63 (does not run)

64 (does not run)

65 (does not run)

66 (does not run)

67 (does not run)

68 (does not run)

69 (does not run)

70 (does not run)

71 (does not run)

72 (does not run)

73 (does not run)

74 (does not run)

75 (does not run)

76 (does not run)

77 (does not run)

78 (does not run)

79 (does not run)

80 (does not run)

81 (does not run)

82 (does not run)

83 (does not run)

84 (does not run)

85 (does not run)

86 (does not run)

87 (does not run)

88 (does not run)

89 (does not run)

90 (does not run)

91 (does not run)

92 (does not run)

93 (does not run)

94 (does not run)

95 (does not run)

96 (does not run)

97 (does not run)

98 (does not run)

99 (does not run)

100 (does not run)

101 (does not run)

102 (does not run)

103 (does not run)

104 (does not run)

105 (does not run)

106 (does not run)

107 (does not run)

108 (does not run)

109 (does not run)

110 (does not run)

111 (does not run)

112 (does not run)

113 (does not run)

114 (does not run)

115 (does not run)

116 (does not run)

117 (does not run)

118 (does not run)

119 (does not run)

120 (does not run)

121 (does not run)

122 (does not run)

123 (does not run)

124 (does not run)

125 (does not run)

126 (does not run)

127 (does not run)

128 (does not run)

129 (does not run)

130 (does not run)

131 (does not run)

132 (does not run)

133 (does not run)

134 (does not run)

135 (does not run)

136 (does not run)

137 (does not run)

138 (does not run)

139 (does not run)

140 (does not run)

141 (does not run)

142 (does not run)

143 (does not run)

144 (does not run)

145 (does not run)

146 (does not run)

147 (does not run)

148 (does not run)

149 (does not run)

150 (does not run)

151 (does not run)

152 (does not run)

153 (does not run)

154 (does not run)

155 (does not run)

156 (does not run)

157 (does not run)

158 (does not run)

159 (does not run)

160 (does not run)

161 (does not run)

162 (does not run)

163 (does not run)

164 (does not run)

165 (does not run)

166 (does not run)

167 (does not run)

168 (does not run)

169 (does not run)

170 (does not run)

171 (does not run)

172 (does not run)

173 (does not run)

174 (does not run)

175 (does not run)

176 (does not run)

177 (does not run)

178 (does not run)

179 (does not run)

180 (does not run)

181 (does not run)

182 (does not run)

183 (does not run)

184 (does not run)

185 (does not run)

186 (does not run)

187 (does not run)

188 (does not run)

189 (does not run)

190 (does not run)

191 (does not run)

192 (does not run)

193 (does not run)

194 (does not run)

195 (does not run)

196 (does not run)

197 (does not run)

198 (does not run)

199 (does not run)

200 (does not run)

201 (does not run)

202 (does not run)

203 (does not run)

204 (does not run)

205 (does not run)

206 (does not run)

207 (does not run)

208 (does not run)

209 (does not run)

210 (does not run)

211 (does not run)

212 (does not run)

213 (does not run)

214 (does not run)

215 (does not run)

216 (does not run)

217 (does not run)

218 (does not run)

219 (does not run)

220 (does not run)

221 (does not run)

222 (does not run)

223 (does not run)

224 (does not run)

225 (does not run)

226 (does not run)

227 (does not run)

228 (does not run)

229 (does not run)

230 (does not run)

231 (does not run)

232 (does not run)

233 (does not run)

234 (does not run)

235 (does not run)

236 (does not run)

237 (does not run)

238 (does not run)

239 (does not run)

240 (does not run)

241 (does not run)

242 (does not run)

243 (does not run)

244 (does not run)

245 (does not run)

246 (does not run)

247 (does not run)

248 (does not run)

249 (does not run)

250 (does not run)

251 (does not run)

252 (does not run)

253 (does not run)

254 (does not run)

255 (does not run)

256 (does not run)

257 (does not run)

258 (does not run)

259 (does not run)

260 (does not run)

261 (does not run)

262 (does not run)

263 (does not run)

264 (does not run)

265 (does not run)

266 (does not run)

267 (does not run)

268 (does not run)

269 (does not run)

270 (does not run)

271 (does not run)

④ Example:

consider the loop.

NOTE:

RULE OF THUMB:

for ($i=n$; $i>1$; $i = \frac{i}{2}$) { } goal

* if problem is increasing by multiplying

temp;

OR

* In this case steps will be opposite of expressions example.

* To calculate time complexity:

check:

then the growth rate of that problem is "LOGARITHMIC".

NOTE:

* Logarithmic techniques/algorithms depends

on

"DIVIDE AND CONQUER" RULE

(means to divide a bigger problem into smaller steps (problem) and then solve those smaller steps (problem) individually.)

⑤ Example:

consider a loop.

for ($i=1$; $i<n$; $i = i*5$) $\rightarrow \log_5 n + 1$

we get:

"DIVIDE AND CONQUER"

$i = n/5^k$

* The growth rate

of this algorithm

is "LOGARITHMIC".

$t(n) = 2 \log_5 n + 1$

$n = 2^k$

$t(n) = \log_2 n$

RECURSION:

- * In recursion, we check/calculate complexities for two things:

1 → Base case complexity (where if condition is true)

2 → recursion function calling (where if condition is false)

① Example:

consider recursion algorithm.

```
void print (int n)
{
```

if ($n \leq 0$) $\rightarrow 1$

$\text{cout} \ll n; \rightarrow 1$

else

$\text{print}(n-1); \rightarrow 1 + t_{\text{print}}(n-1)$

CHECK:

1. if ($n = 0$):

$$t_{\text{print}}(0) = 2$$

- * This function will stop calling itself when the statement of "if" will be true.

② Example:

consider a recursive algorithm.

```
int func (int n) {
```

```
    if (n==0) { + 1 → 1 }
```

```
    else { + 1 → 1 }
```

$$t_{\text{func}}(n) = \begin{cases} 3 & ; n=0 \\ 1 + t_{\text{func}}(n-1) & ; n>0 \end{cases}$$

```
return func(n-1)+n; → 1 + t_{\text{func}}(n-1)
```

not compulsory
to write its time complexity.

* Now we have to do substitution of this.

CHECK:

1. if (n==0) (0) + n =

$t_{\text{func}}(0) = 3$

2. $(n \neq 0)$ this step is to check condition of "if" (whenever it is false)

$t_{\text{func}}(n) = 1 + 1 + t_{\text{func}}(n-1)$

$t(n) = 1 + t(n-1)$

SUBSTITUTION: (Taking time complexity where "if" will be false (means function calling time complexity))

* This statement is known as

"RECURRANCE RELATION"

(a statement in which there is a recursive function calling)

$$t(n) = 3 + n$$

→ The growth rate of this algorithm is "linear".

PERFORMANCE ANALYSIS OF ALGORITHM

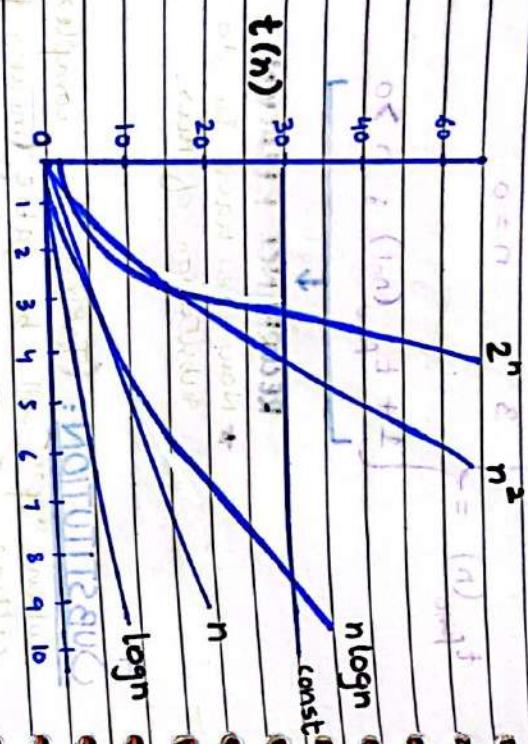
PERFORMANCE MEASUREMENT

COMPARISON OF TIME COMPLEXITIES

* Now, For time taken on machine.

- 1st thing we have to know is "COMPUTER SPEED".

$$\text{COMPUTER SPEED} = \frac{\text{No. of STEPS / SEC}}{\text{Instructions}}$$



* The execution time of an algorithm running on a particular machine based on input can be calculated by following formula.

FORMULA: (For time taken by an algorithm on particular machine)

$$\text{time} = \frac{\text{total step count of an algorithm}}{\text{speed of a machine (steps/sec)}}$$

(taken by an algorithm on particular machine)

$\log n$	n	$n \log n + \ln^2 n + 1 = n^3$	2^n
0	1	0	1
1	2	4	2
2	4	16	4
3	8	512	8
4	16	4096	16
5	32	32768	32

NOTE:

- * "PERFORMANCE MEASUREMENT" is dependant of machine.
- * "PERFORMANCE ANALYSIS" is independent of machine.

SPACE COMPLEXITY

- * The amount of "memory" (RAM) taken by an algorithm to execute itself.

DENOTATION: $S_p(n)$

* In space complexity, we didn't consider the language (don't bother about how many bytes it will take on memory for a specific language)

* Here (in space complexity) we only see/take/use no. of words because int, float, etc. size may change from machine to machine and language to language (so we don't bother about their size, we only consider "THAT^{es}" their size in words; 1, 2, etc.)

* We only see about no. of variables which we declare by ourselves in our algorithm / program.

NOTE:

* In space complexity we are independent of language.

* If we have a function (algorithm) in which we have "int" as well as "float" variables (datatype), we don't bother about their sizes on machine. We only consider all separately declared variable are same; whether they are of "1" words or "2" words, so on.

To store n words $\rightarrow n \times 4 = (n)$ words

so that we don't need to write lengthy functions.

→ Asymptotic Notations generally have '3'

languages:

- Big-Oh O
- Big-Omega Ω
- Big-Theta Θ

* On machine, we calculate the time by

End time - start time

NOTE:

for n words \rightarrow $n \times 4$ bytes
for n int to n bytes
"Machine" \rightarrow

ASYMPTOTIC NOTATIONS

* These are mathematical notations.

* These are used to describe the time complexity of an algorithm when the input tends towards a particular value or a limiting value.

* These notations help us to represent time complexities in simple way instead of writing lengthy functions.

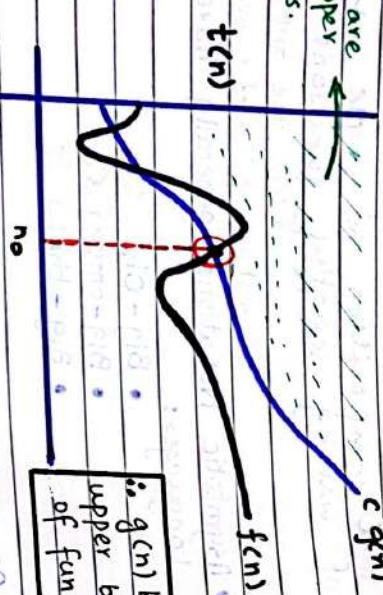
(We use asymptotic notation so that we don't need to write lengthy function

* We only find growth rate/time of processing, not the time of input or output.

① Big-OH (O)

This represent the upper bound of a function.

- The function $f(n) = O(g(n))$ iff there exist some positive constant (c) and n_0 (both ' c ' and ' n_0 ' positive) where $(n \geq n_0)$ such that $f(n) \leq c g(n)$.



$$c g(n) = 2n$$

n	$f(n) = 2n+3$	$c g(n) = 2n$	$f(n) \leq c g(n)$
1	5	2	NO (x)
2	7	4	NO (x)
3	9	6	NO (x)
4	11	8	NO (x)
5	13	10	NO (x)

- The function ' $2n$ ' ($c g(n)$) is not upper bound of $f(n)$ because it is not greater than or equal to (\geq) $f(n)$.

→ Let's choose some function $c g(n)$

$$c g(n) = 3n$$

n	$f(n) = 2n+3$	$c g(n) = 3n$	$f(n) \leq c g(n)$
1	5	3	NO (x)
2	7	6	NO (x)
3	9	9	YES (v)
4	11	12	YES (v)
5	13	15	YES (v)

EXAMPLE: UPPER BOUND Big-OH

$$f(n) = 2n + 3$$

Let's choose some function $c g(n)$ which is greater than from $f(n)$

- $2n+3 = O(n)$ with $c=3$ and $n_0=3$

Let's

choose $f(n) = 2n+3$ and

→ If we choose $f(n) = 2n+3$ and

$$cg(n) = 2^n$$

<u>n</u>	<u>$f(n) = 2n+3$</u>	<u>$c g(n) = 7n$</u>	<u>$f(n) \leq c g(n)$</u>
1	5	7	✓
2	7	14	✓
3	9	21	✓
4	11	28	✓
...	⋮	⋮	⋮
100	203	700	✓

<u>n</u>	<u>$f(n) = 2n+3$</u>	<u>$c g(n) = 2^n$</u>	<u>$f(n) \leq c g(n)$</u>
1	5	4	✗
2	7	16	✗
3	9	64	✗
4	11	256	✗
...	⋮	⋮	⋮
100	203	1.2676506e30	✓

* $2n+3 = O(2^n)$ with $c=1$ and $n_0=4$

→ If we choose $f(n) = 2n+3$ and $n_0=1$

∴ Here '2ⁿ' is not a whole function. We don't consider it as 'c=2' and power 'n' as a function.

<u>n</u>	<u>$f(n) = 2n+3$</u>	<u>$c g(n) = n^2$</u>	<u>$f(n) \leq c g(n)$</u>
1	5	1	✗
2	7	4	✗
3	9	9	✓
4	11	16	✓
5	13	25	✓

* $2n+3 = O(n^2)$ with $c=1$ and $n_0=3$

$$3n\lg n + 2n + 4 = O(n\lg n) \text{ at } c=9, n_0=1$$

⇒ SIMPLEST WAY / EASIEST WAY TO

FIND $c g(n)$ AS UPPER BOUND

OF $f(n)$: $f(n) = (n^2 + 3n + 2)^2$

* As we can see that any functions

like $f(n)$ would have many upper bounds which will be called $c g(n)$.

(Like we saw in previous examples, $2n, 3n, 7n, 2^n, n^2$, etc. are all upper bounds of $f(n) = 2n+3$)

* So, here we have a question that which upper bound function ($c g(n)$) should be chosen.

* To choose the $c g(n)$:

we should put the highest degree polynomial with all the variables.

* So, easiest way of taking $c g(n)$:

For example:

$$f(n) = 2n+3 \text{ and its } c g(n) = 2n+3n = 5n$$

$$c g(n) = 5n^2 + n^2 + 3n^2 = (n^2)^2$$

* So,

$5n$

but $< 5n$ means

$$1 = n, 6n > 5n, 10n > 5n, \dots$$

$7n$ and $8n$ are not upper bounds

Kn are all upper bounds of $f(n)$

① EXAMPLE:

If we choose $f(n) = 2n+3$ and

$$c g(n) = 2n+3n$$

$$c g(n) = 5n$$

$$f(n) \leq c g(n)$$

n	$f(n) = 2n+3$	$c g(n) = 5n$
1	5	5
2	7	10
3	9	15
...
100	203	500

$$2n+3 = O(n) \text{ with } c=5 \text{ and } n_0=1$$

② EXAMPLE:

$$f(n) = (n^2)^2 = n^4$$

$$c g(n) = 2n^2 + 3n^2 + 1n^2$$

$$c g(n) = 2n^2 + 3n^2 + 1n^2$$

$$2n^2 + 3n^2 + 1 = O(n^2) \text{ with } n_0=1, c=6$$

Substituted $n=1$ into $O(n^2)$ doesn't deviate from it.

Going to prove $c g(n)$ is not many than $f(n)$.

Like $c g(n)$ and $f(n)$ are equal but $c g(n)$ is not many than $f(n)$.

It's because $c g(n)$ has $c+1$ s.

③ EXAMPLE

If

$$f(n) = 5n^3 + 2n^2 + 3n + 6$$

$$\hookrightarrow c g(n) = 5n^3 + 2n^3 + 3n^3 + 6n^3$$

$$c g(n) = 16n^3$$

* $5n^3 + 2n^3 + 3n + 6 = O(n^3)$ with $n_0 = 1, c = 16$

* By this method (of finding $c g(n)$), "n" will always be "1".

CHOOSING APPROPRIATE UPPER BOUND ($c g(n)$):

$$\begin{aligned} 1 &= cn \\ 2n+3 &= O(n) \quad \text{all are upper bounds of } n \\ 2n+3 &= O(n^2) \\ 2n+3 &= O(2^n) \\ 2n+3 &= O(n^n) + n^{1.5} = (n)^{1.5} \end{aligned}$$

* Which one to choose as upper bound of $f(n)$?

* Although, they all are correct because all of them are upper bound of $2n+3$.

* But always choose the meaningful $g(n)$

* So, here $O(n)$ is the closest to $2n+3$ and meaningful (so, we choose it as Big-Oh of $f(n)$).

ASSYMPTOTIC NOTATIONS REPRESENTING CONSTANT FUNCTION:

* A constant function is represented asymptotically by digit '1' like $O(1)$, $\Omega(1)$, $\Theta(1)$.

* For example:

$T(n) = 3$; this function was a constant growth.

so, we can simply represent it with $O(1)$ instead of writing 2, 3, or any other term.

$T(n) = O(1)$ with $c = 1, n_0 = 1$.

$$O(1) = t(n)$$

⇒ SIMPLEST WAY / EASIEST WAY TO

FIND $c g(n)$ AS LOWER BOUND
OF $f(n)$:

* As we can see that any functions

(like $f(n)$) would have many lower bounds which will be called $c g(n)$

(Like we saw in previous examples; n , $2n$, 4 , $2\log n$, etc are all lower bounds of $f(n) = 2n+3$)

* So, here is a question that which lower bound function ($g(n)$) I should choose?

* To choose the $c g(n)$:

We take highest degree polynomial of $f(n)$ with its constant and neglect all other variable or constant, etc.

Like:

for $f(n) = 2n+3$, we take $c g(n) = 2n$

① EXAMPLE: If we consider function $f(n) = 2n^2 + 3n + 4$

$$f(n) = 2n^2 + 3n + 4$$

$\hookrightarrow c g(n) = 2n^2$ is a meaningful one.

② EXAMPLE: If

$$f(n) = 3n\log n + 2n + 4$$

$\hookrightarrow c g(n) = 3n\log n$ is a meaningful one.

* By this method (of finding $c g(n)$), it will always be '1'.

CHOOSING APPROPRIATE LOWER BOUND ($c g(n)$):

$1 < \log n < \sqrt{n} < n$

\hookrightarrow all are lower bound of $f(n)$.

$$2n+3 = \Omega(n)$$

$$\text{and } 2n+3 = \Omega(\log n)$$

$$\text{and } 2n+3 = \Omega(\sqrt{n})$$

* Which one to choose as lower bound of $f(n)$?

* Although, they are all correct because all of them are lower bound of $2n+3$.

* But

* always choose the meaningful $g(n)$.

* so here $\Omega(n)$ is the closest to $n^{1.5}$ and meaningful (so, we choose as big-Omega $(n^{1.5})$)

Note:

- * In asymptotic notations, there is no need to write base of \log and \ln .
 $\ln(n) = \ln(n) \cdot \Omega = p + n^{\frac{1}{2}} + n^{\frac{1}{3}}$
- * Whenever base is not mentioned, we assume:
 - \ln to be base 2 ($\ln = \text{base}_2 \log$)
 - \log to be base 10 ($\log = \text{base}_{10}$)

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Upper Bound Lower Bound

③ Big-THETA (Θ)

denotation

- * This represent the average bound of a function.
- * It is also known as 'tight bound'.

- * Every algorithm must have some upper bound and lower bound.
 $f(n) = \Omega(n)$
- * Upper bound (and) lower bound can be mostly same because there is also equality sign in their definition.

In broad sense 20 seconds at 1000 m/s

EXAMPLE ②

If (we take ' $c_1 g(n)$ ' and ' $c_2 g(n)$ ')

$$f(n) = 2n+3$$

$$c_1 g(n) = 1n^2$$

$$2n+3 = O(n^2) \text{ with } c=1, n_0=1$$

AND

$$c_2 g(n) = 1n$$

$$2n+3 = \Omega(n) \text{ with } c=1, n_0=1$$

- It does not have big-theta bound

because there is no upper bound and lower bound for same.

- This $f(n)$ only have Big-Oh = $\Theta(n)$ Big-Theta, because $c_1 g(n) \neq c_2 g(n)$

$$1n^2 \neq 1n$$

 not same.

NOTE:

Tell

* We cannot find Big Theta (Θ) of \sqrt{n} because its upper and lower bound are always different (means $c_1 g(n) \neq c_2 g(n)$).

OR

MCQ

* Big Theta does not exist for every / all the algorithms (functions).

↳ if big-theta ' Θ ' does not exist for a function, they will go (find for Big-Oh 'O' or Big-Omega ' Ω ').

Lecture #1

To MAKE A LIST (FOR A LIST)

→ It is a Data Structure.

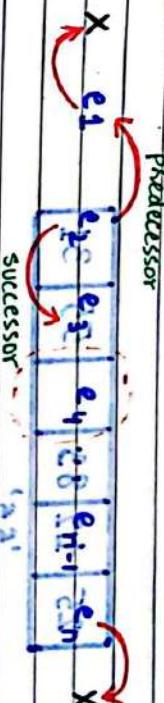
→ It is ADT (Abstract Data Type).

→ A set of values written down one below the other.

→ A number of connected items or names (written) or printed consecutively, typically one below the other.

→ List is a contiguous memory allocation of homogeneous elements.

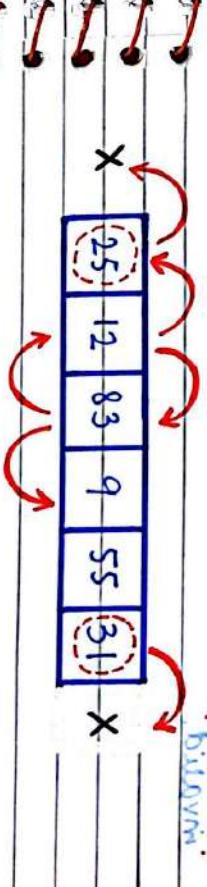
- * The elements of the list must have predecessor and successor relationship.



* The first element of the list has no predecessor, therefore it is marked as the beginning of the list. Drawn yellowish after twice already gone out.

* The last element of the list has no successor therefore it is marked as the end of the list. Drawn pink pink.

X	25	12	83	9	55	31	X
---	----	----	----	---	----	----	---



* Element '25' does not have any predecessor therefore indicates the beginning of the list.

* Element '31' does not have any successor therefore indicates the end of the list.

* All other elements have their predecessor and successor to maintain the list properly.

→ INVALID LIST

- * A list having vacant / empty spaces
- * A list having elements being present at the mid of the list.

25	12	83	55	55	31
					19

- * Element '55' does not have any predecessor so it will not be considered as the last element of which is 55 logically wrong because there are two more elements exist after the empty cell.
- * This list will be considered as invalid.

15	22	0	88	51	20
					19

TYPES OF LIST

(1) UNORDERED LIST

- * Elements are placed in the list in no particular order.
- * Unordered list does not have in symmetry or logical relation among them.

- * It is only a list.

- * Here, only relationship of predecessor and successor holds.

- * We mostly maintains unordered list.

- * In unordered list, the given list could be sorted as well as unsorted.

sun	sand	tan	250	25	frank	*
girl	250	tan	professor	professor	tan	*
dei	girl	tan	professor	professor	tan	*

but rokengborg didn't send Almudha volta like
but rokengborg didn't send Almudha volta like

(2) ORDERED LIST

→ UNSORTED LIST

- * Elements maintain certain order either numerically or alphabetically

RAYAN
SHABIR
ALI
SHAHMEER

- * Elements are ordered according to some rule (relationship)

- * Each element followed by one another.

- * In ordered list there have only relationship according to which order data is arranged.

ALI
RAYAN
SHABIR
SHAHMEER

→ SORTED LIST

- * Names are sorted alphabetically.

SHAHMEER
RAYAN
SHABIR
ALI

SORTED LIST

NOTE:

- * It is not necessary that ordered list will always be sorted and it is also not necessary that unordered list will always be unsorted.

PROPERTIES / TERMINOLOGIES OF A LIST:

GENERAL OPERATIONS PERFORMED

→ To maintain a list of elements, following things should be considered

• Max Size

* Every list must have some "Max size" → total available size (memory)

↳ represent total capacity (which it maximum holds)

(2) **DISPLAY:** Display the complete data of list
 $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

(3) **SEARCH:** Searches an element using the given key.

(4) **UPGRADE:** Upgrades an element with new data.

(5) **DELETION:** Deletes an element using its index.

(6) **MERGE:** It merges two lists.

* Every list must also have "curr size"

→ sorts (memory) which we use

↳ current number of elements or not holds.

* Initial currsize is '0'.

NOTE:

* We mostly deals with "currSize".

If maxsize is '10' and currsize is '5', then traverse / processing will be done upto currsize (i.e. 5th location).

ON A LIST:

1. Insertion
2. Deletion
3. Search
4. Upgrade
5. Merge

(1) INSERTION IN UNORDERED LIST:

- * It inserts / adds an element in the list.
- * To perform insertion in the list, we first have to check if either the list is full or not:
 - currSize == MaxSize?**
 - If at start **currSize = 0** and user wants to insert element (2), then this new element will be inserted at the position **currSize + 1** (i.e. 1st location).
 - if curr size is 1 → The max size of this is '6' and current size is '4'.
 - currSize = 1
 - After this insertion : 102134910 (8)
 - Then if user again want to insert element '3' then again check **currSize == maxSize**
 - Then this new element will be inserted at the position **currSize + 1**
 - **currSize++** → The current size raises to '5'.
 - **currSize = 2**
 - Whenever we do insertion, we will increment the currSize by '1'.
 - Insertion takes constant time to evaluate so, its time complexity is **O(1)**.

5	3	4	7	1	2
---	---	---	---	---	---

Initially sum till 6 will be 0 *
 After inserting 6 sum will be 6 * *

1	2	3	4	5	6
25	12	83	9	66	

(2) DISPLAY IN UNORDERED LIST:

* To display a list, we print data upto the **curSize** only.
We don't print upto **maxSize** because we want to print data which is in use, and we only print those location addresses which are in use.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525	526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541	542	543	544	545	546	547	548	549	550	551	552	553	554	555	556	557	558	559	560	561	562	563	564	565	566	567	568	569	570	571	572	573	574	575	576	577	578	579	580	581	582	583	584	585	586	587	588	589	590	591	592	593	594	595	596	597	598	599	600	601	602	603	604	605	606	607	608	609	610	611	612	613	614	615	616	617	618	619	620	621	622	623	624	625	626	627	628	629	630	631	632	633	634	635	636	637	638	639	640	641	642	643	644	645	646	647	648	649	650	651	652	653	654	655	656	657	658	659	660	661	662	663	664	665	666	667	668	669	660	661	662	663	664	665	666	667	668	669	670	671	672	673	674	675	676	677	678	679	680	681	682	683	684	685	686	687	688	689	690	691	692	693	694	695	696	697	698	699	700	701	702	703	704	705	706	707	708	709	710	711	712	713	714	715	716	717	718	719	720	721	722	723	724	725	726	727	728	729	730	731	732	733	734	735	736	737	738	739	740	741	742	743	744	745	746	747	748	749	750	751	752	753	754	755	756	757	758	759	760	761	762	763	764	765	766	767	768	769	770	771	772	773	774	775	776	777	778	779	770	771	772	773	774	775	776	777	778	779	780	781	782	783	784	785	786	787	788	789	790	791	792	793	794	795	796	797	798	799	800	801	802	803	804	805	806	807	808	809	800	801	802	803	804	805	806	807	808	809	810	811	812	813	814	815	816	817	818	819	810	811	812	813	814	815	816	817	818	819	820	821	822	823	824	825	826	827	828	829	830	831	832	833	834	835	836	837	838	839	830	831	832	833	834	835	836	837	838	839	840	841	842	843	844	845	846	847	848	849	850	851	852	853	854	855	856	857	858	859	850	851	852	853	854	855	856	857	858	859	860	861	862	863	864	865	866	867	868	869	860	861	862	863	864	865	866	867	868	869	870	871	872	873	874	875	876	877	878	879	870	871	872	873	874	875	876	877	878	879	880	881	882	883	884	885	886	887	888	889	880	881	882	883	884	885	886	887	888	889	890	891	892	893	894	895	896	897	898	899	890	891	892	893	894	895	896	897	898	899	900	901	902	903	904	905	906	907	908	909	900	901	902	903	904	905	906	907	908	909	910	911	912	913	914	915	916	917	918	919	910	911	912	913	914	915	916	917	918	919	920	921	922	923	924	925	926	927	928	929	920	921	922	923	924	925	926	927	928	929	930	931	932	933	934	935	936	937	938	939	930	931	932	933	934	935	936	937	938	939	940	941	942	943	944	945	946	947	948	949	940	941	942	943	944	945	946	947	948	949	950	951	952	953	954	955	956	957	958	959	950	951	952	953	954	955	956	957	958	959	960	961	962	963	964	965	966	967	968	969	960	961	962	963	964	965	966	967	968	969	970	971	972	973	974	975	976	977	978	979	970	971	972	973	974	975	976	977	978	979	980	981	982	983	984	985	986	987	988	989	980	981	982	983	984	985	986	987	988	989	990	991	992	993	994	995	996	997	998	999	990	991	992	993	994	995	996	997	998	999	1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023	1024	1025	1026	1027	1028	1029	1020	1021	1022	1023	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055	1056	1057	1058	1059	1050	1051	1052	1053	1054	1055	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071	1072	1073	1074	1075	1076	1077	1078	1079	1070	1071	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087	1088	1089</

2) DISPLAY IN UNORDERED LIST:

→ If it is return true that list is

not empty, then traverse (loop will be evaluated

to process data upto curSize).

* We don't print upto maxsize because we want to print the data which is present on the locations we are using.

* So, that is the reason, we execute our loop from 0 to less than curSize (< curSize) that data is in use.

→ So, here, all elements (i.e. n elements) are being checked one by one. So, its time complexity will be $O(n)$ or $\Theta(n)$.

* The searching operation usually performed before the deletion and/or upgradation of desired list element.

(3) SEARCHING IN UNORDERED LIST:

* Searching of an element in the list simply be done by comparing the required key element one by one with the elements of the list.

* Checks each element with a "key" element.

* But here we first have to check "if the list is empty or not"

curSize = 0? (if list is empty)

if curSize > 0 (if list is not empty)

(4) DELETION IN UNORDERED LIST:

- * "Deletion" operation cannot perform without "Search" operation.

* Whenever we want to delete in an unordered list, first we have to search.

If we have a list

25	12	83	9	6	
----	----	----	---	---	--

- And we want to delete '12'.

the deleted elements with the last element of list and decrement the curSize by one.

25	83	9	6	
----	----	---	---	--

Like:

25	83	9	6	
----	----	---	---	--

- After removal an empty (vacant) space is left, so here rule of predecessor and successor violates.

It becomes invalid list.

- * → To maintain the property of the list (predecessor and successor), after removing / deleting an element from unordered list and still makes it a list.

so, list will be:

25	6	83	9	
----	---	----	---	--

The time complexity for deletion for unordered list is also ' $n \rightarrow O(n)$ '.

* Here we shift an element towards the empty cell and decrement the curSize.

(by 1 unit). increased time requirement.

→ But the shifting here is not valid because it requires ' $n-1$ ' (or 'n').

These shifting's have relation with time shifting's.

* Shifting element like this will take $O(n)$.

* As list in unordered, so best way to remain it as an empty list (maintain property of the list).

To make it more efficient, simply replace

the deleted elements with the last element of list and decrement the curSize by one.

25	83	9	6	
----	----	---	---	--



* It is because deletion also first finds the element for which it goes

time complexity will become ' n ' ($O(n)$)

* (Then), after searching for the element will be deleted and last element will be replaced to empty cell for this, its time complexity will be $O(1)$.

: insertion in sorted list

* So, originally the time complexity for deleting will be $O(1)$, and for overall

searching will be $O(n)$. The time complexity for deletion operation will be $O(n)$.

But we know that $O(n)$ is not good for insertion.

So, we did **size--**

because here value is only updating; neither adding nor decreasing.

So, time complexity of insertion is also $O(n)$.

(5) UPGRADE IN UNORDERED LIST:

* First check if list is not empty;

* "Upgrade" operation also cannot perform without "Search" operation.

* To upgrade an unordered list, first we have to search for the element which we want to update with new value.

* But here neither **ysize++** nor **ysize--**

because here value is only updating; neither adding nor decreasing.

So, time complexity of unordered list is also $O(n)$.

3	9	88	25
---	---	----	----

i = 98900

class NewList{public: int size(); void insert(int); void delete(int); void display();};

: add item to list

return

int NewList::size(){int count=0; for(int i=0;i<list.size();i++) count++; return count;}

GENERAL OPERATIONS FOR ORDERED LIST:

1) MERGE:

- * We merge/join two lists.
- * The resultant will be a combination of the elements of both lists.
- * If we want to merge lists a and b .

2) INSERTION IN SORTED LIST:

↳ Here list is sorted.

- * Here every element is placed directly at last position, we first have to check where it to place the element according to some relationship.
- * If we want to insert e in sorted list a :

3) SPLITTING:

- * We divide a list into two parts.

$$\text{element 1 : } e_1 = 1 \\ \text{element 2 : } e_2 = 4$$

4) COMPARISON:

- * We compare two lists with each other.

$$e_1 < e_2 < e_3 < e_4 < e_5 \dots < e_n$$

→ So, after checking

and then insert.

→ then if we want to again insert 13?

$$c_3 = 7$$

1	4	7	(3)	7	$1 < 3$
1	(3)	4	7	7	✓

- * Then again, we will compare element.
But remember we always compare with last element i.e. **currSize**.

- * And if the element (which is being compared) comes according to relationship i.e. "less than currSize" ($< currSize$) then we keep comparing backward until we place the element at right position.

* In ordered list (sorted list), we must have to shift elements, to maintain order of list.

* Thus, time complexity of insertion in unsorted list is $O(n)$.

NOTE:

So, Best way to make a list first maintain unordered list and then maintain its order by using some sorting algorithm.

- * List became:

1	4	7	13	7
---	---	---	----	---

(2) SEARCHING IN SORTED LIST

- * In sorted list, we have to traverse/reach element linearly one by one.
- * So, its time complexity is $O(n)$ in worst case but $O(1)$ in best case.

→ Then suppose we want to insert a new element '3' in a sorted list, we have to shift all elements one step forward to place '3' in right place.

1	4	7	13	7
---	---	---	----	---



1	4	7	13	7
---	---	---	----	---

$$n_2 = n + n$$

(3) DELETION IN SORTED LIST:

- * First, we will search the element which we want to delete.
- * Here we cannot place the last element to the deleted position (like we did in unsorted list) because its order will not be maintained.
- * So, here only way to delete element is by shifting, so that order maintains. So, its time complexity will be n .

$$\text{Search} \leftarrow \begin{cases} \text{shift} & \text{if found} \\ \text{end} & \text{otherwise} \end{cases}$$

(4) UPGRADE IN SORTED LIST:

- * First, we will have to "search" the element which we want to upgrade.
- * Then, we will check the value with which we want to upgrade the element.

We will check if the element is on its right position or not by comparing elements.

- * So, here time complexity will be n .

Search \leftarrow

$$n + n = 2n$$

(5) MERGE:

: VASNA

- * To merge two sorted lists, we cannot directly merge them because we have to maintain order of the resultant list.

* Splitting is easy in a sorted list.

We will simply divide the list into two parts and each part will

be in sorted form. Now we have to merge both the parts.

(6) SPLITTING:

IN COMPARISON:

- * We compare two sorted lists with each other.

NOTE:
 No division to smallest part

No loop no

* Generally, it is a good idea to maintain unsorted lists at first instead of maintaining order of elements.

- * List can be sorted easily with help of some efficient sorting algorithms later on when needed.

* In DSA, we avoid to user "break" and "goto" statements.

Lecture #8

→ **Word:**

Word M denotes the size of an element

* "Memory Representation of List is done by Array" \rightarrow It is a group of contiguous memory locations.

* It is a data structure which stores a group of data together in one place.

* Array data structure enables us to store a group of data together in one place.

→ **Range of Indices:**

Indices of the array elements may change from lower bound (LB) to an upper bound (UB), and these bounds are also called the "Boundaries of an Array".

* An array is a finite collection of homogeneous data elements. All the elements of arrays are stored in contiguous locations of memory.

ARRAY TERMINOLOGY:

→ **SIZE:**

The number of elements in an array.

→ **Type:**

The kind of data an array represents e.g. integers, character string, etc.

→ **Base:**

memory address of the first element of array.

ARRAY MEMORY ALLOCATION:

: Right ↗

* Array can be declared in one of two ways in C++.

↓

(1) IN STACK: possible to grow ↗

possible only for variable length arrays

- Declaration "array (ar) of 50 integers" on stack memory.
- Automatically deallocated when goes out of scope.

(2) IN HEAP:

- int *ar = new int[50];
- Declares an array (ar) of 50 integers on Heap memory.
- Needs to be deallocated by programmer with the help of delete []ar; statement.

ARRAY DESCRIPTOR TABLE

↓

* Whenever array creates, there is a descriptor table behind it.

* This table is known as: **"ARRAY DESCRIPTOR TABLE"**

↓

- It holds Name of array.
- It holds Base Address (Starting Address of array).

* It holds Word size.

* It holds Lower Bound (LB) and

↑

* Upper Bound (UB). Subsequent item,

int ar[10]; *ar* → not exist (logical Address only)



Index	Value	Memory Address of Array Element
0	100	Address of first element
1	102	Address of second element
2	104	Address of third element
3	106	Address of fourth element
4	108	Address of fifth element
5	110	Address of sixth element
6	112	Address of seventh element

Base Address = 100, Element Size (Word) = 2 bytes

Upper Bound (Index) = 5

Lower Bound (Index) = 0

Size = 2 bytes, Lower Bound (index) = 0
and Upper Bound (index) = 5.

These can also be used for boundary checking.

Name	Base Address	Word Size	Lower Bound	Upper Bound
ar	100	2 bytes	0	5

$$001 = \text{Alpha word}$$

$$\text{Count} = UB - LB + 1$$

$$\text{Size(Array)} = 5 - 0 + 1$$

$$\text{Size(Array)} = 6$$

In most of the arrays, Base Address is written as "Alpha" or "Beta".

Now, calculating Memory size of Array:

$$col = 5 + 001 = 100 = (\text{110})_{\text{workba}}$$

$$poi = \text{Base Address} = 100 = (\text{110})_{\text{workba}}$$

$$801 = R + col = (100) + 001 = (101)_{\text{workba}}$$

$$\text{Element Size} = 2 \text{ bytes (Word size)}$$

$$\text{Size (Array)} = 6$$

$$\text{Memory size (Array)} = \text{Size (Array)} \times \text{Word size}$$

But most of the languages also have different starting indices.

Some languages also provide us opportunity to define your own indices (USER DEFINED INDICES).

$$011 \quad 001 \quad 001 \quad 001 \quad 001$$

$$= 12 \text{ bytes}$$

So, To calculate this: (Memory needs to be allocated)