DATA STRUCTURE WITH "C" IN HINDI

BccFalna.com

097994-55505

Kuldeep

If you are really interested in Professional Development, you will sure have to learn various types of **Data Structures and Algorithms** propery so that you can create Well Performing Application Softwares.

In this EBook, I have covered various important espects of Data Structures and Algorithms like **Array, Linked List, Stack, Queue, Tree, Graph, etc…** which are very important to learn not only for Degree Level Cources but also for Efficient and Well Performing Professional Development.

In this EBook, I have covered various Data Strucutre Concepts with “C” Programming Language, but I have also described Algorithm of each concept, which can be easily implemented in any Programming Language.

Data Structure with "C" In Hindi

■■■
Kuldeep Chand

Betalab Computer Center
Falna

Data Structure and Algorithms with "C" in Hindi

Lead Editors: Kuldeep Chand

Distributed to the book trade worldwide by Betalab Computer Center, Behind of Vidhya Jyoti School, Falna Station Dist. Pali (Raj.) Pin 306116

e-mail *bccfalna@gmail.com*

or

visit *http://www.bccfalna.com*

For information on translations, please contact Betalab Computer Center, Behind of Vidhya Jyoti School, Falna Station Dist. Pali (Raj.) Pin 306116

Contact: 097994-55505

This book is dedicated to those

who really wants to be

a

PROFESSIONAL DEVELOPER

INDEX OF CONTENTS

Table of Contents

DATA, FIELD, RECORD & FILE

Data Structure Fundamentals and Arrays

Data - Field, Record and File

किसी भी समस्या के समाधान के लिए Computer को विभिन्न प्रकार के मानों की जरूरत होती है। Computer में किसी मान या मानों के समूह को **Data** कहा जाता है और मानों के समूह के किसी Single Item को **Data Item** कहा जाता है।

जिस Data Item को Sub Item में विभाजित किया जा सकता है, उस Data Item को **Group Item** या **Composite Data Item** कहते हैं और जिस Data Item को Sub Item में विभाजित नहीं किया जा सकता, उस Data Item को **Elementary Data Item** या **Primary Data Item** कहते हैं।

उदाहरण के लिए किसी School में जो विभिन्न Students Study करते हैं उन सभी Students का कोई ना कोई नाम होता है। हर Student का नाम Computer के लिए एक Data है। हम हर Student के नाम को तीन Sub Items First Name, Middle Name व Last Name विभाजित कर सकते हैं। इसलिए Name एक Group Data Item है।

इसी तरह जब भी कोई Student किसी School में Admission लेता है तो उस Student को एक Unique Serial Number प्रदान किया जाता है। एक Student को जो Number Allot किया जाता है वह Number किसी दूसरे Student को Allot नहीं किया जाता। ये Number किसी अमुक विधार्थी की Unique पहचान होती है। विभिन्न Students को दिए जाने वाले Serial Number को हम अन्य Data Items में विभाजित नहीं कर सकते हैं। इसलिए Serial Number एक Elementary Data Item कहलाता है।

Data के किसी समूह को Fields, Records व Files की Hierarchy के रूप में Organize किया जा सकता है। हम जिस किसी भी चीज को Computer में Manage करना चाहते हैं, उसे एक **Entity** या इकाई के रूप में लेते हैं। दुनिया की हर चीज Computer के लिए एक **Object** या **Entity** है। जैसे Table, Chair, Computer, CPU, RAM आदि।

इसी तरह से किसी Company के विभिन्न Employees उस Company के लिए Entities या Objects हैं और किसी School के विभिन्न Teachers, उस School के Objects या Entities हैं। इसी तरह किसी Class के विभिन्न Students उस Class के Objects या Entities हैं।

यानी दुनिया की हर वस्तु Computer के लिए एक Object या Entity है जिसे Computer में Data Item के रूप में Organize किया जा सकता है। हर Object या Entity की कुछ विशेषताएं होती हैं, जो उसे अन्य Object या Entity से अलग बनाती है। जैसे एक Student की विभिन्न विशेषताएं उसका नाम, उसका Serial Number, उसकी उम्र, उसका Color उसका Sex आदि हो सकती हैं।

इसी तरह से किसी Company के विभिन्न Employees की भी अपनी विशेषताएं हो सकती हैं। किसी भी Entity या Object की विभिन्न विशेषताओं को Entity की **Characteristics** या

Properties या **Attributes** कहते हैं। इन Attributes में कोई ना कोई मान Assign किया जा सकता है। ये मान Numeric या Non - Numeric हो सकते हैं। जैसे किसी Student के विभिन्न Attributes को निम्नानुसार मान प्रदान किया जा सकता है:

Sr_No	Name	Age	Sex	Class
123	Amit Sharma	15	Male	10
234	Rahul Varma	16	Male	10
121	Salini Bohra	15	Female	9
544	Silpa Roy	14	Female	8
534	Prince Mishra	13	Male	6
532	Devendra Bhati	14	Male	9

Entity का वह समूह जो कि Similar Attributes को Share करता है, **Entity Set** कहलाता है। जैसे Table 1 में विभिन्न Students समान Attributes को Share कर रहे हैं इसलिए ये समूह Students Entities का एक Set कहलाता है। Entity के हर Attribute को प्रदान किए जा सकने वाले मान की एक Range होती है। हम Table 1 में देख सकते हैं कि हमें हर Row में किसी Students की विभिन्न जानकारियां प्राप्त हो रही हैं। यहां किसी Attribute को प्रदान किया जाने वाला मान एक Processed Data होता है। इस Table में कई Attributes मिलकर किसी एक Student के बारे में पूरी Information प्रदान कर रहे हैं।

वह तरीका जिसमें Data को Fields, Records व Files के Hierarchy के रूप में Organized करते हैं, Data, Entity व Entity Set के बीच में एक Relationship को Represent करता है। Field किसी Single Entity के किसी Attribute को Represent करता है। किसी Entity के विभिन्न Attributes को प्रदान किया जाने वाला मान Record को Represent करता है और किसी Entity Set के विभिन्न Entities को File Represent करता है। किसी Record में किसी Entity के कई Fields हो सकते हैं लेकिन जो Field किसी Record को Uniquely Identify करता है उसे **Primary Key Field** कहते हैं। जैसे किसी School के विभिन्न Students को Uniquely Identify करने के लिए हर Student का एक Serial Number होता है।

Data को Fields, Records व Files के रूप में अच्छी तरह Organized करने के बाद भी Data को Maintain व Process करना काफी जटिल होता है। इस वजह से Data को और अधिक जटिल Structure में Organize किया जाता है। किसी भी Data Structure को समझते समय हमें निम्न बातों पर ध्यान देना होता है–

1. Structure की Logical या Mathematical Description
2. Structure की Computer पर Processing
3. Structure का Analysis जिसके आधार पर ये तय किया जाता है कि कोई Data Structure Memory में कितनी Space लेगा और Data को Process करने में कितना समय लगेगा।
4. Memory में सूचनाएं किस प्रकार से संगठित हो कर रहेंगी।

किसी भी Program की सार्थकता सूचनाओं के संगठन के आधार पर निर्भर होती है। प्रोग्राम की योग्यता इस बात पर निर्भर करती है, कि Data Memory में किस प्रकार से संगठित (Organized) हैं व उनका आपस में क्या सम्बंध है। यदि Data सही प्रकार से Memory में संगठित ना हों, तो प्रोग्राम के Execution में अधिक समय लगता है।

अतः किसी भी प्रोग्राम के Fast Execution के लिये उचित Data Structure का चयन बहुत ही जरूरी है। Data का वह समूह, जो Memory में कम से कम स्थान लेता हो और सामुहिक रूप से आपस में सम्बंधित हों तथा प्रोग्राम मे Fast Execution में सहयोग करते हों, **Data Structure** कहलाते हैं।

Data Structure वास्तव में Program बनाते समय अपनाए जाने वाले विभिन्न तरीकों में से सबसे सरल व अच्छा तरीका उपयोग में लेना होता है। ये उपयोग में लिये जाने वाले तरीके पर निर्भर करता है कि हमारा Program व उस Program के Data Memory में कितना Space लेंगे।

जैसे कि हम एक Array में ढेर सारे Data रख सकते हैं, लेकिन यदि Array की Size अधिक रखी जाए, तो Array द्वारा फालतू में ही Memory का दुरूपयोग होता है। इस Array के स्थान पर यदि Dynamic Memory Allocation का प्रयोग किया जाए, तो ये उतनी ही Memory Use करता है, जितनी जरूरत होती है।

इस प्रकार से Data, Array के बजाय Dynamic Memory Allocation से Memory में अधिक अच्छी तरह से Organized रहते हैं। इसलिये Dynamic Memory Allocation एक अधिक अच्छा Data Structure या Data के साथ प्रक्रिया करने का माध्यम है।

कोई भी Data Structure Data सदस्यों के बीच Relationship भी प्रदर्शित करता है। जैसे कि एक Telephone Directory में उन सभी व्यक्तियों के नाम, पते व Phone Number लिखे होते हैं, जिनके पास Phone है। ये सभी एक क्रम में होते हैं। ये क्रम **Data Structure** है और सभी Phone Numbers को उसके मालिक के नाम व पते के साथ लिखा जाता है, जिससे किसी भी Phone Number से उस Phone के मालिक का नाम पता आदि जाना जा सकता है। इस प्रकार से सभी Phone Numbers का उसके मालिक से सम्बंध है।

इसलिये हम कह सकते हैं कि Data Structure के सदस्य आपस में Related होते हैं। किसी भी Program को दो कसौटियों पर अच्छा या बुरा कहा जा सकता हैः Program के Execution द्वारा लिया जाने वाला समय और Program द्वारा Memory में लिया जाने वाला स्थान। यदि Program Execute होने में काफी समय लगाता है, तो Program को अच्छा नहीं कहा जा सकता।

साथ ही यदि Program Memory में काफी अधिक स्थान लेता है, तो ये Program की कमी है। एक अच्छा Program Memory में कम से कम स्थान लेता है और कम से कम समय में अच्छा से अच्छा परिणाम प्रदान करता है। हम एक उचित Data Structure का प्रयोग करके यानी एक सरल व उचित तरीका अपना कर ये दोनों जरूरतें पूरी कर सकते हैं।

Data Structures

Data को Organize करने के कई तरीके हो सकते हैं। Data को Organize करने के Logical या Mathematical Model को **Data Structure** कहा जा सकता है। हम किस Data Structure को Choose करें ये बात दो तथ्यों पर निर्भर करती है:

1 Structure इतना सक्षम होना चाहिए कि वह उसी तरह से Logically Data के विभिन्न Elements के बीच Relationship प्रदर्शित कर सके जिस तरह से वास्तविक जीवन में विभिन्न Data Items आपस में Related होते हैं।

2 Data Structure इतना सरल होना चाहिए कि कोई भी Programmer किसी भी Computer Language में Coding लिख कर Data को आसानी से Process कर सके।

Linear Data Structure

जब किसी Data Structure के सभी Items एक Continuous Memory Locations पर उपलब्ध हों, तो इसे **Linear Data Structure** या Linear List कहते हैं। जैसे कि एक **Array** के सभी Elements लगातार Memory Locations पर उपलब्ध रहते हैं। विभिन्न Memory Locations पर उपलब्ध विभिन्न Data Items के बीच Relationship Represent करने का एक तरीका ये है कि हम Array का प्रयोग करें। Array एक Linear Data Structure है। दूसरे तरीके में विभिन्न Data Items के बीच के Relation को एक Linked List के रूप में Represent किया जाता है। इस तरीके के Data Structure में हम **Linked Lists** का प्रयोग करते हैं।

Non-Linear Data Structure

जब किसी Data Structure में सभी इकाईयां एक Continues Memory Locations पर उपलब्ध ना हों, तो ये एक **Non-Linear Data Structure** कहलाता है। Non–Linear Data Structures के रूप में हम Trees व Graphs का प्रयोग करते हैं। किसी भी Data Structure पर हम निम्न क्रियाएं कर सकते हैं:

1 नई इकाई जोडना।
2 किसी इकाई को Delete करना।
3 Processing के लिये हर इकाई पर Move करना।
4 किसी मान को सभी इकाईयों में खोजना।
5 इकाईयों की Sorting करना।
6 दो Structures को जोड कर एक Structure बनाना।

जब हम किसी Data Structure को Choose करते हैं तब किसी Data Item के साथ किस प्रकार से प्रक्रिया करनी है, ये तथ्य उस Data Structure पर निर्भर करता है कि हमने किस प्रकार का Data Structure Choose किया है।

Array एक सबसे सरल Data Structure है जिस पर विभिन्न प्रकार के Operations करना काफी आसान होता है। Array का प्रयोग तब किया जाता है जब Data के Permanent Collection पर विभिन्न Operations करने होते हैं। क्योंकि Array की Size एक ही बार में Set करनी पडती है इसलिए इसमें जो भी Data होते हैं वे Permanent होते हैं। लेकिन जब Data Structure की Size Changeable होती है तब हम Array के स्थान पर Linked List का प्रयोग करते हैं।

Algorithm : Time – Space Tradeoff and Complexity

किसी समस्या के समाधान को प्राप्त करने के लिए विभिन्न Steps की एक Well Defined List को **Algorithm** कहते हैं। Data को Efficiently Process करने के लिए एक Efficient Algorithm की आवश्यकता होती है। कोई Algorithm कितना Efficient है यानी किसी समस्या के समाधान के लिए Algorithm किस तरह लिखा गया है और Algorithm में लिखे गए Steps कितनी Efficiently Data की Processing करते हैं, इसे दो तथ्यों **Time** व **Space** के आधार पर तय किया जाता है।

हर Algorithm में Data पर विभिन्न तरीके से Processing की जाती है। इसलिए हम हमेंशा Data को Process करने के लिए सबसे Efficient Algorithm को Use नहीं कर सकते। कोई Algorithm कितना Efficiently Data पर Processing करेगा ये बात कुछ अन्य तथ्यों पर भी निर्भर करती है। जैसेकि हम किस प्रकार के Data पर Processing कर रहे हैं और Data पर विभिन्न Operations करने के लिए कितने Steps लेने पडते हैं।

Time – Space Tradeoff Use किए जा रहे Data Structure पर निर्भर करता है। यानी यदि हम Data को Store करने के लिए Space बढा दें तो Data की Processing में लगने वाला समय कम हो जाता है और यदि Data को Store करने के लिए Use होने वाले Space को कम कर दें, तो Data की Processing में लगने वाला समय बढ जाता है। Time–Space Tradeoff को समझने के लिए हम एक उदाहरण लेते हैं।

मानलो कि एक File में विभिन्न Students की Information हैं। File को Name Wise Sort करके और Binary Searching Algorithm को Use करके हम बहुत Efficient तरीके से इस File में से किसी विशेष नाम के Record को प्राप्त सकते हैं।

लेकिन यदि हमें किसी Student का Serial Number दिया गया हो और हमें उस Serial Number वाले Student के Record को Search करना हो तो हम Binary Searching तरीके को Use करके Record को नहीं खोज सकते हैं।

क्योंकि Binary Searching Algorithm को Use करने के लिए हमें Sorted Records की जरूरत होती है और किसी File के विभिन्न Records को या तो Name Wise Sort करके रख सकते हैं या Serial Number के अनुसार। हम Name व SR_No दोनों को एक साथ Sort करके नहीं रख सकते। इसलिए यदि हमें SR_No के अनुसार किसी Record को खोजना हो तो हमें File के हर Record को किसी अमुक SR_No के लिए Check करना होगा। यदि Search की जाने वाली File में काफी अधिक Records हों तो इस तरह की Searching में बहुत Time लगेगा।

इस समस्या का एक समाधान ये हो सकता है कि हम एक और File बनाए और उसे Serial Number Wise Sort करके रखें। लेकिन ऐसा करने पर समान प्रकार के Data की दो File बन जाएगी जिससे Memory में दुगुना Space Use होगा। इसलिए इस तरीके को भी एक Efficient तरीका नहीं कहा जा सकता।

इस समस्या के समाधान के रूप में हम एक तरीका और अपना सकते हैं। हम Main File को SR_No Number के अनुसार Sort कर देते हैं और एक और Array लेते हैं और उसमें केवल दो Columns एक नाम के लिए व दूसरा Pointer के लिए लेते हैं। इस Array को Name Wise Sort कर लेते हैं। इस Array के हर SR_No का एक Pointer Main File के किसी Record को Point करता है।

इस तरीके में हालांकि दूसरे Array के लिए Extra Space Use हो रहा है लेकिन फिर भी इस Array में केवल दो Fields हैं, इसलिए इस तरीके में कम से कम Space Use होगा। इस तरीके के Algorithm को हम एक **Efficient Algorithm** कह सकते हैं।

किसी Algorithm की **Complexity** एक Function होता है जो किसी Input Data के आधार पर Data की Processing में लगने वाला समय या Space या दोनों को दर्शाता है। यानी किसी Algorithm को Execute होने में कितना समय लगेगा और वह Algorithm Memory में कितना Space लेगा, इन दोनों या दोनों में से किसी एक बात को दर्शाने के लिए हम जिस Function को Use करते हैं, वह Function बताता है कि कोई Algorithm कितना Complex है और कितने समय में किसी Data की Processing करेगा तथा Data की Processing के लिए कितना Space Use करेगा। इन Functions द्वारा हम Mathematically ये जान सकते हैं कि कोई Algorithm किसी अन्य Algorithm की तुलना में कितना Efficient है।

Computer Science में Algorithms को Analyze करना एक बहुत ही बडा व जटिल काम है। किन्ही Algorithms की तुलना करने के लिए हमारे पास कुछ Criteria होना बहुत जरूरी होता है, जिससे हम पता लगा सकें कि कोई Algorithm कितना Efficient है।

यदि किसी Data Structure में **n** Data हों तो Algorithm **M** के Input Data की Size **n** होती है। Input Data की Size किसी भी Algorithm का पहला Criteria होता है। हम किसी Algorithm से किस प्रकार के Steps का प्रयोग करके Data Process करते हैं, उन Steps की संख्या किसी Algorithm की **Efficiency** ज्ञात करने का दूसरा Criteria होता है।

यानी कोई Algorithm कितना Efficient है, ये Input Data व Data को Process करने के लिए Use किए जाने वाले Steps की संख्या यानी Comparisons पर निर्भर करता है।

कोई Algorithm किसी समस्या का समाधान प्रदान करने के लिए कितने समय व Space का उपयोग करता है, इन दोनों तथ्यों के आधार पर उस Algorithm की **Efficiency** का पता चलता है। मानलो कि **M** एक Algorithm है और उसके Input Data की Size **n** है। किसी Searching या Sorting के Algorithm में जितने Operations के बाद Data Process होता है, उन Operations की संख्या के आधार पर Algorithm के Time का पता चलता है।

जैसे मानलो कि एक File में 1000 Records हैं और उन में से किसी विशेष नाम के Record को प्राप्त करना है, तो Algorithm को वांछित Record प्राप्त करने के लिए अधिकतम 1000 Comparisons करने पड सकते हैं। इन अधिकतम 1000 Comparisons में लगने वाले समय को Algorithm द्वारा Use किया जाने वाला समय कहते हैं और इन 1000 Comparisons में Algorithm जितनी Memory को Use करता है, वह Memory किसी Algorithm द्वारा Use की जाने वाली Space होती है।

किसी Algorithm M की Complexity ज्ञात करने के लिए एक **Function f(n)** का प्रयोग किया जाता है। ये Function किसी Algorithm की Complexity प्रदान करता है जहां **n** Input Data की Size है। उदाहरण के लिए किसी File में केवल 1 Record है तो उस Record से किसी नाम के Record को Search करने पर Algorithm की Complexity कम होगी जबकि उसी File में यदि 100 Record हों तो Use होने वाले Algorithm की Complexity अधिक होगी। यदि वांछित Record पूरी File में कहीं प्राप्त ना हो तो Algorithm की Complexity अनन्त होगी।

मानलो कि जो Record Search किया जा रहा है वह File में पहले स्थान पर ही उपलब्ध हो तो Algorithm की Complexity बिल्कुल कम होगी और Algorithm को **Best Case Algorithm** कहा जाएगा। यदि Search किया जाने वाला Record File के मध्य में हो तो Algorithm को **Average Case Algorithm** कहा जाएगा और यदि जो Record Search किया जा रहा है वह Record पूरी File में कहीं ना हो तो ऐसे Algorithm को **Worst Case Algorithm** कहा जाता है।

किसी Worst Case Algorithm में Record को Search करने के लिए उतनी बार Comparison का Operations करना पडता है, जितने File में Records हैं। मानलो कि File में 100 Record हैं तो Algorithm को 100 बार Comparison करना पड सकता है। इसे हम Mathematically निम्नानुसार प्रदर्शित कर सकते हैं:

C(n) = n

जहां C = Comparisons की संख्या है और n = Input Data की Size है। किसी Linear Search Algorithm की Worst Case Complexity में C(n) = n होती है।

Average Case Algorithm में किसी Data के हर Location पर मिलने की सम्भावना 1/n होती है जहां **n** Data Items की संख्या है। यानी यदि किसी Data Structure में n Data Items हों तो Data 1 से लेकर n तक में किसी भी स्थान पर हो सकता है। इसलिए किसी Data Item को किसी List में खोजने के लिए Algorithm को कुल **n** Comparisons करने पड सकते हैं। इसे हम निम्नानुसार Mathematically दर्शा सकते हैं:

```
C(n)      = 1 X 1/n + 2 X 1/n + ... n X 1/n
= ( 1 + 2 + ... + n ) X 1/n
= n( n + 1 )/2 X 1/n
= n + 1/2
```

इस Probability Equation से हम देख सकते हैं कि किसी Data के List में मिलने की सम्भावना लगभग n/2 होती है जहां **n** List के कुल Data Items की संख्या है।

Algorithms

किसी समस्या का समाधान Computer द्वारा प्राप्त करने के लिए हमें एक विशेष क्रम में विभिन्न Steps Use करने होते हैं। Steps की एक Well Defined List जिसके आधार पर किसी भी Computer Language में Program Create करके किसी Problem को अच्छी तरीके से Solve किया जा सके, **Algorithm** कहलाता है।

दूसरे शब्दों में कहें तो हम कह सकते हैं कि किसी समस्या के समाधान के लिए जिन Steps को Use किया जाता है, उन Steps को यदि एक निश्चित क्रम में सरल भाषा में लिख लिया जाए, तो इन Steps की List को **Algorithm** कहा जा सकता है।

Algorithm का हर Step ये बताता है कि कब और किस काम के बाद क्या काम हो रहा है। Algorithm किसी Problem के Solution का एक Specification होता है जिसके आधार पर किसी समस्या को Solve किया जाता है।

किसी Algorithm को सीमित Instructions की एक Sequence के रूप में देखा जा सकता है जिसमें निम्न गुण होते हैं:

1. Algorithm शुरू होने से पहले उसे कुछ Initial मान प्रदान किए जाते हैं। इन मानों को Input कहा जाता है और इन्हीं Input पर कोई Algorithm Processing करता है।

2. Algorithm के विभिन्न Steps इतने सरल व समझने योग्य होते हैं कि उस Algorithm का प्रयोग करके हम किसी भी Computer Language में उस Algorithm के आधार पर Program Create करके किसी समस्या का समाधान प्राप्त कर सकते हैं।

3. Algorithm का हर Step इतना Clear होना चाहिए कि कोई भी व्यक्ति उस Algorithm के आधार पर एक सीमित समय में उस समस्या का समाधान प्राप्त कर ले जिसके लिए Algorithm को लिखा गया है।

4. किसी Algorithm द्वारा किसी समस्या के समाधान के लिए लिखे गए सभी Steps एक सीमित समय में पूरे होने चाहिए। कई बार किसी समस्या के समाधान के लिए Repetitive Steps Use किए जाते हैं। ये Steps ऐसे होने चाहिए कि सीमित समय में समस्या का समाधान प्रदान कर सकें यानी Loop Infinite नहीं होना चाहिए।

5. एक Algorithm का कम से कम एक या एक से अधिक Output होना चाहिए। Steps का कोई भी ऐसा समूह जो किसी प्रकार का कोई Result Provide ना करता हो, उसे Algorithm नहीं कहा जा सकता।

Analysis of Algorithm

जब भी हम कोई Algorithm लिखते हैं तो ये जरूरी हो जाता है कि हम ये Analyze करें कि हमारे द्वारा लिखा गया Algorithm कितना Efficient है। Algorithm के Analysis का पहला तरीका ये है कि हम ये Check करें कि Algorithm सही लिखा गया है या नहीं। इसके लिए हमें निम्न काम करने होते हैं–

1. Algorithm की Tracing करना। यानी Algorithm के हर Step को Check करना कि जो काम जहां होना चाहिए वह वहीं हो रहा है या नहीं।

2. Algorithm की Reading करना। यानी ये पता लगाना कि Algorithm Logically Correct है या नहीं।

3. Algorithm की Implementing व Testing करना। यानी Algorithm के आधार पर किसी भी Programming Language में Program बनाना तथा Check करना कि Program सही Output प्रदान कर रहा है या नहीं। या फिर Mathematical Techniques द्वारा Algorithm की Correctness को Prove करना।

Algorithm के Analysis का दूसरा तरीका ये है हम Algorithm को Simplest Form में Create करें। यदि Algorithm को Simplest Form में लिखा गया हो तो उसे Implement करना यानी उसके आधार पर Program Create करना व उस Algorithm पर अन्य प्रकार के Analysis करना सरल होता है। फिर भी किसी समस्या को Solve करने के लिए Use किया जाने वाला सरल व साफ–सुथरा तरीका कई बार कुछ ज्यादा अच्छा तरीका नहीं होता है। ऐसा तब होता है जब Use किया जाने वाला सरल व साफ–सुथरा तरीका या तो काफी अधिक Memory का उपयोग करता हो या Solution प्रदान करने में काफी ज्यादा समय लगाता हो।

इस स्थिति में ये जरूरी होता है कि ये Analyze किया जाए कि जो Algorithm किसी समस्या के समाधान के लिए लिखा गया है वह कम से कम कितना Time व Space Use करता है। उदाहरण के लिए, यदि किसी Company में 120 Employees हैं और हर Employee के Record को Memory में Load होने में 3 मिनट लगते हैं तो इस Algorithm को Use नहीं किया जा सकता क्योंकि यदि किसी दिन Company के सभी Employees को Access करना हुआ, तो सभी Employees के Records को Memory में Load होने में ही 4 घण्टे लग जाएंगे, फिर हर Record की Processing में तो और भी अधिक समय लगेगा। इसलिए कोई Algorithm कितना अच्छा है इसे **Time** व **Space** के आधार पर ही नापा जा सकता है।

Rate of Growth

कोई Algorithm Perform होने में कितने समय का उपयोग करेगा ये Analyze करने का कोई साधारण तरीका नहीं है। किसी भी Algorithm के Time Requirement को प्रभावित करने वाली सबसे पहली Complexity तो ये है कि कोई Algorithm किस Computer पर Perform हो रहा है, उस Computer पर ही Algorithm के Perform होने का Time निर्भर होगा।

मानलो यदि हम एक ही Algorithm को Pentium III Processor वाले CPU पर Execute करते हैं और उसी Algorithm को Pentium IV के Processor पर Execute करें, तो दोनों Computer पर Algorithm के Perform होने के समय में अन्तर रहेगा। यानी सबसे पहले तो कोई भी Algorithm कितना समय लेगा ये उस Computer की Speed पर निर्भर करता है जिस पर Algorithm को Use किया जा रहा है।

Algorithm के Perform होने के Time को प्रभावित करने वाली दूसरी Complexity Input Data Items की संख्या पर निर्भर करती है। उदाहरण के लिए यदि किसी Array में 100 Data Elements को जोडना हो तो कम समय लगेगा जबकि यदि Array में 10000 Data Elements हों, तो उन सभी की जोड करने में 100 Data की तुलना में अधिक समय लगेगा।

परिणामस्वरूप कोई Algorithm Perform होने में अनुमानतः कितना समय लेगा, इसे Input Data Size के फलन के रूप में Express किया जा सकता है। उदाहरण के लिए यदि किसी Array में Data Items की संख्या n हो तो Algorithm द्वारा n Data की Processing में लगने वाले समय को Time T(n) व Space S(n) के फलन के रूप में Express किया जा सकता है, जहां T व S Input Data n के फलन हैं।

इससे पहले कि हम किसी Algorithm को Analyze करके उसकी Efficiency ज्ञात करें, कुछ Functions को समझना ठीक रहेगा, जिनका प्रयोग T व S को Express करने के लिए किया जाता है। ये कुछ Standard Functions हैं जो Input Data Item n की Size के आधार पर Algorithm द्वारा लिया जाने वाला समय ज्ञात करने के लिए Use किए जाते हैं।

उदाहरण के लिए मान लो कि किसी Array में 16 Data Items हैं। इन 16 Data Items को अलग–अलग तरह से अलग–अलग कामों के लिए Process करने के लिए अलग–अलग Algorithms की जरूरत होती है। यदि इन 16 Items पर Processing करने के लिए हम **$f(n) = 2^n$** फलन का प्रयोग करें तो हमें Array के 16 Elements को Process करने के लिए कम से कम 2^{16} Operations करने पडेंगे, यानी हमें कुल 65536 Operations करने होंगे।

यदि इस फलन का Graph बनाया जाए, तो बनने वाला Graph काफी तेजी से बढेगा। यानी इस फलन के बढने की दर सबसे तेज होती है। ये फलन किसी भी Algorithm की Complexity ज्ञात करने का सबसे पहला फलन है जो किसी Algorithm के Perform होने में Algorithm द्वारा लिए जाने वाले सबसे अधिक समय को दर्शाता है।

यदि इस फलन के स्थान पर Algorithm **$f(n) = n^3$** फलन का प्रयोग करे, तो कुल Operations की संख्या 16^3 होगी। यदि इस फलन का भी ग्राफ बनाया जाए तो ये ग्राफ भी काफी तेजी से बढेगा लेकिन फिर भी ये Graph पहले वाले फलन की तुलना में कम तेजी से बढेगा।

किसी Algorithm की Complexity ज्ञात करने के लिए Use किया जाने वाला तीसरा फलन **$f(n) = n^2$** है। यदि इसी 16 Element वाले Array के 16 Data को Process करने के लिए यदि n^2 Operations करने पडे तो, Algorithm को कुल 256 Operation करने होंगे। यदि इस फलन के Growth का Graph बनाया जाए तो ये Graph भी काफी तेजी से बढेगा लेकिन पहले वाले दोनों फलनों की तुलना में काफी कम तेजी से बढेगा।

यदि Algorithm Perform होने के लिए **f(n) = n $\log_2$ n** Operations करता है तो इस फलन का Growth Rate पहले बताए गए सभी फलनों से कम होगा। यानी यदि इस फलन का Graph बनाया जाए तो Graph काफी धीमी गति से बढेगा।

यदि 16 Data Element के Array पर Processing के लिए **n $\log_2$ n** Operations करने पडें तो कुल Operations की संख्या **16 X $\log_2$ 16 = 16 X 4 = 64** होगी जो कि पहले के सभी फलनों की तुलना में काफी कम है।

यदि जो Algorithm Use किया जा रहा है वह Algorithm 16 Elements को Process करने में केवल 16 ही Operations करे, तो Algorithm की Complexity **f(n) = n** होती है। यदि इसका Graph बनाया जाए, तो एक Linear Graph बनता है जो कि Data Elements n की संख्या बढने के साथ बढता है और घटने के साथ घटता है।

किसी Algorithm की Complexity ज्ञात करने का अन्तिम फलन **f(n) = $\log_2$ n** है जिसमें 16 Item को Process करने में केवल 4 Operations करने होंगे। यदि इन सभी फलनों का Graph बनाया जाए तो ये Graph क्रम से अधिक Complex Algorithm को दर्शाएंगे। इस Graph को किसी Algorithm की Complexity का **Rate of Growth** कहा जाता है।

कोई Algorithm Perform होने के समय किस फलन के अनुसार Operations करता है, इसके आधार पर Algorithm की Complexity निर्भर करती है। एक ही काम को करने के लिए विभिन्न तरीके हो सकते हैं। जो तरीका कम से कम Operations में हमें हमारा Required Result प्रदान कर देता है वह Algorithm कम Complex और अधिक Efficient कहलाता है जबकि इसके विपरीत होने पर Algorithm अधिक Complex व कम Efficient कहलाता है। Algorithm की Growth निम्न क्रम में होती हैः

$\log_2$ n n n $\log_2$ n n^2 n^3 2^n

किसी भी Algorithm को Analyze करके ये पता किया जाता है कि Algorithm की Complexity किस फलन के अनुसार Grow हो रही है। हमें यही कोशिश करनी चाहिए कि किसी भी Algorithm की Complexity Linear क्रम में बढे। ऐसा Algorithm काफी Efficient होता है जिसका Graph Linearly बढता है।

Time Analysis

निम्न Algorithm द्वारा किसी Array के N Elements का जोड किया जा सकता है–

Algorithm

SUMMETION(LA, N)

Here LA is a Linear Array and N is the size of LA.

1 SUM = 0, LB = 0, UB = N-1 [Initialization]

```
2   REPEATE STEP 3 WHILE I <= UB STEP I = I + 1
3   SUM = SUM + LA[I]                    [Sum the Values of Array's All Elements ]
4   EXIT                                                                  [Finish]
```

जैसाकि हमने पहले बताया कि किसी भी Algorithm को Perform होने में कितना समय लगेगा ये ज्ञात करना काफी मुश्किल है। इसलिए हम Algorithm की Complexity को Data Items की Size के फलन के रूप में ही ज्ञात कर सकते हैं।

Algorithm के Time को हम किसी Algorithm द्वारा Perform होने वाली Comparisons संख्या के आधार पर ज्ञात करते हैं। कोई Algorithm Perform होते समय कुल कितने Operations करता है, उन्हीं Operations की संख्या को हम Algorithm के Perform होने के Time के रूप में मान लेते हैं।

यानी जितने अधिक Operations Algorithm द्वारा Perform होते हैं उस Algorithm को Perform होने में उतना ही अधिक समय लगता है। Time Algorithm में मानलो कि Data Item की संख्या n=10 है तो दसों संख्याओं को जोडने के लिए इस Algorithm में कुल 10 Operations होते हैं।

इस स्थिति में इस Algorithm को Perform होने में कुल 10 Operation करने पडते हैं तो इस Algorithm की Complexity का फलन f(n)=n होगा। यानी इस Algorithm को Perform होने में यदि Data Item की संख्या N है तो कुल Operation की संख्या भी N होगी और हर Operation में 1/N समय लगता है अतः कुल समय भी N ही लगेगा।

मानलो कि यदि Array के प्रथम Element को दूसरे Element से जोडने में एक सेकण्ड लगता है तो Data Item की संख्या 10 होने पर कुल दस बार जोड होगी और लगने वाला समय 10 सेकण्ड होगा।

Complexity (Big O Notation)

दो Algorithms में से कौनसा Algorithm अधिक Efficient है ये Analyze करने के लिए हमें दोनों Algorithms से प्राप्त होने वाले Results को Compare करना होता है। हम दो Algorithms से प्राप्त होने वाले Results को Compare कर सकें इसके लिए हमें "O" Notation को जानना जरूरी है।

कोई Algorithm कितने Operations करने के बाद Required काम को पूरा करता है, ये जानने के लिए हम कुछ Standard Function का प्रयोग करते हैं जो कि निम्नानुसार हैं–

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n

ये सभी Functions (फलन) किसी Algorithm का आयाम ज्ञात करने के लिए Use होते हैं। यानी n^2 से n^3 का आयाम अधिक होता है। यदि कोई एक Algorithm n^2 Operations के बाद हमें Required Results प्रदान करता है जबकि दूसरा Algorithm n^3 Operations के बाद हमें

Required Result Provide करता है, तो पहले वाले Algorithm से दूसरा वाला Algorithm अधिक Complex होगा। ये सभी Function एक निश्चित क्रम के अनुसार Input Data **n** के लिए आयाम प्रदान करते हैं। विभिन्न Functions जो आयाम प्रदर्शित करते हैं उन सभी आयाम प्रदर्शित करने वाले Functions को Handle करने के लिए एक Notation को Develop किया गया है।

एक Function **f(n)** को **O(g(n))** के रूप में परिभाषित किया जा सकता है। यानी इसे **f(n)=O(g(n))** लिख सकते हैं और इस फलन को **g(n)** का क्रम कहा जाता है। यानी यदि निम्नानुसार $\mathbf{n_0}$ व **c** Positive Constant हों:

$| f(n) | <= | g(n) |$ जबकि सभी $n > n_0$

तो हम इसे "O" Notation के अनुसार लिख सकते हैं। जैसे निम्न फलनों का "O" Notation निम्नानुसार लिखा जा सकता है–

$100\ n^5$	का "O" Notation $\mathbf{O(n^5)}$ होगा।
$200n^3 + 50n^2 + 20n^1 + 544n^0$	का "O" Notation $\mathbf{O(n^3)}$ होगा।
$1 + 2 + \ldots + (n-1) + n = n(n+1)/2 = n^2 + O(n)$	का "O" Notation $\mathbf{O(n^2)}$ होगा।
3223	का "O" Notation **O(1)** या **O(n)** होगा।

सरल शब्दों में कहें तो हम कह सकते हैं कि O(g) फलनों (Functions) का एक ऐसा समूह होता है जो कि g के आधार पर Increase या Grow होता है जहां फलन g फलन O(g) का Upper Bound है। हम O(g) के विभिन्न फलनों के आधार पर विभिन्न Algorithms की Complexity ज्ञात कर सकते हैं और दो अलग–अलग Algorithms की Efficiency की आपस में तुलना करके पता लगा सकते हैं कि कौनसा Algorithm अधिक Efficient या कम Complex है। ये फलन Computer या Programming Language पर निर्भर नहीं होते हैं, इसलिए किसी Algorithm को बिना Implement किए हुए यानी Algorithm के आधार पर बिना Program बनाए हुए ही इन फलनों द्वारा हम ये जान सकते हैं कि कोई Algorithm कितना Complex है।

Properties of "O" Notation

किसी भी "O" Notation की कुछ General विशेषताएं होती हैं जिन्हें हम निम्नानुसार समझ सकते हैं:

1 किसी भी Algorithm को जब Mathematically Represent किया जाता है तो उसके जितने भी Constant Factors होते हैं उन्हें Ignore किया जा सकता है। उदाहरण के लिए यदि किसी Array में N Data Elements हैं और कोई Algorithm Array के सभी Elements को Process करता है जबकि N का मान 0 से अधिक है तो इस Algorithm की Complexity O(n) होगी। यानी

For All N > 0 , N ***f*** is **O(n)**

उदाहरण के लिए xz^2 व yn^2 दोनों की Complexity समान है और दोनों को "O" Notation के रूप में **$O(n^2)$** ही लिखेंगे।

2 n की Higher Power (उच्चतम घातांक) Lower Power (निम्नतम घातांक) से अधिक तेजी से Grow होता है। यानी

$200n^3 + 50n^2 + 20n^1 + 544n^0$ का "O" Notation **$O(n^3)$** होगा।

इस उदाहरण में देख सकते हैं कि फलन n^0 भी है और n^3 भी लेकिन इसका "O" Notation **$O(n^3)$** ही है क्योंकि ये उच्चतम घातांक है और अन्य सभी घातांकों की तुलना में इसका मान बहुत ही ज्यादा तेजी से Grow होगा और ये अन्य घातांको की तुलना में बहुत ही ज्यादा Operations को Represent करेगा, जिससे अन्य घातांकों के Operations की संख्या को Ignore किया जा सकता है।

3 कुल Operations के योग के बढने की दर वही होती है जो सबसे अधिक Growth Term की होती है। यानी यदि किसी Algorithm में

$an^3 + bn^4 + cn^2 + dn^1$

Operations होते हैं तो इस Algorithm की Complexity bn^4 के बढने की दर के बराबर होगी। यानी इस Algorithm के Operations को "O" Notation के रूप में यदि लिखा जाए तो **$O(n^4)$** ही लिखना होगा।

4 यदि फलन f फलन g की तुलना में अधिक तेजी से Grow होता है और फलन g फलन h की तुलना मे अधिक तेजी से Grow होता है तो फलन f फलन h की तुलना में भी अधिक तेजी से बढता है।

5 किसी फलन के Upper Bound (अधिकतम Operations की संख्या) का किसी अन्य फलन से गुणा किया जाता है तो प्राप्त होने वाला मान Upper Bound (अधिकतम Operations की संख्या) को Represent करता है। जैसे

IF f is O(g) and h is O(r) then fh is O(gr)

उदाहरण के लिए

IF f is O(n2) and h is O(log n) then fh is O(n2 log n)

6 Exponential फलन Power फलन की तुलना में अधिक तेजी से बढते हैं। जैसे

n^k is $O(b^n)$, जबकि b के सारे मान 1 से बडे हों और k का मान 0 या 0 से बडा हो। यानी
n^k is $O(b^n)$, For all **b > 1** ; **k >= 0**

उदाहरण के लिए

n4 is O(2n), and n4 is O(exp(n))

7 Logarithms Powers की तुलना में बहुत ही धीमी गति से बढते हैं। जैसे

Log_b n is **$O(n^k)$** for all **b > 1** ; **k > 0**

उदाहरण के लिए **log_2 n** is **$O(n^{0.5})$**

किसी Algorithm को Analyze करते समय हमारे सामने दो सबसे बडी समस्याएं होती हैं:

CPU Time

CPU की Speed मुख्य रूप से Algorithm द्वारा ही प्रभावित नहीं होती है बल्कि कुछ अन्य तथ्य भी CPU की Speed को प्रभावित करते हैं। इनमें से कुछ तथ्य निम्नानुसार हैं–

1 हम जो Computer Use कर रहे हैं उस Computer के CPU की Actual Speed क्या है और उस Computer पर कितना Load है। जिस Computer में जितने अधिक Softwares Installed होते हैं, Computer उतना ही अधिक Loaded होता है।

2 हम कौनसी Programming Language Use कर रहे हैं और कौनसा Compiler Use कर रहे हैं इस पर भी Algorithm के Perform होने की Speed निर्भर करती है। जैसे यदि हम ऐसे Compiler को Use कर रहे हैं जो कि Directly Computer के विभिन्न Hardware को Access करने में सक्षम है तो Algorithm अधिक तेजी से Perform होगा।

"C" एक ऐसी ही Language है जिसमें हम Computer के किसी भी Hardware को Directly Access कर सकते हैं। जबकि Visual Basic में यदि Algorithm को Perform करने के लिए Program लिखा जाएगा तो उस Program की Speed "C" के Program की Speed से कम होगी।

3 कुछ अन्य Factors भी होते हैं जो Algorithm की Performance को प्रभावित करते हैं। जैसे यदि Data Structure के रूप में Array को Use किया जाता है तो हमें हमेंशा Algorithm में ये Check करना होता है कि Array में Data Insert करने के लिए Space

है या नहीं। यानी Array की Bounding को Check करना जरूरी होता है अन्यथा Algorithm किसी अन्य Program के Data को Corrupt कर सकता है। किसी Array की Indexing करने में या किसी Record को Access करने में जो Extra समय लगता है वह Algorithm की Performance को प्रभावित करता है।

जब हम हमारे Program में बहुत से Procedures यानी Functions Create करते हैं, तो हर Function के Call होने पर Program Control एक स्थान से दूसरे स्थान पर Jump करता है। इस Jumping में भी कुछ समय व्यर्थ होता है। ये समय भी Algorithm के Performance पर असर करता है।

Input / Output

Constant Time के Algorithms परिभाषा के रूप में सभी Input Data पर Processing के लिए समान समय लेते हैं। लेकिन ज्यादातर Algorithms Constant Time नहीं होते हैं। यानी सभी Data की Processing में समान समय नहीं लेते हैं।

इस स्थिति में हम एक निश्चित अंक के रूप में किसी Algorithm की Efficiency को परिभाषित नहीं कर सकते हैं। इसलिए हमें Algorithm की Efficiency को Input Data की Size n के फलन के रूप में ही परिभाषित करना पडता है।

पहली समस्या का समाधान ये है कि हमें Algorithm की Efficiency को Analytically Measure करना चाहिए ना कि Experimentally, क्योंकि दो अलग Computer पर समान Algorithm भी अलग Time में Perform होगा।

अन्य शब्दों में कहें तो हमें Algorithm को उसकी Efficiency व उस Algorithm को प्रभावित करने वाले तत्वों के आधार पर परिभाषित करना होता है। सामान्यतया हम किसी Algorithm की Efficiency ज्ञात करने के लिए निम्न में से किसी एक को Measure करते हैं:

1. Numerical Algorithms के लिए हम कुल Addition, Subtraction, Multiplication आदि की संख्या ज्ञात करते हैं।
2. Searching या Sorting के Algorithm में हम कुल Comparisons की संख्या ज्ञात करते हैं।
3. Assignment Statements व Parameters के आधार पर Data Movement की कुल संख्या को ज्ञात करते हैं।
4. किसी Algorithm के लिए Required कुल Memory Space की Limit को ज्ञात करते हैं।

दूसरी समस्या के समाधान के लिए हम N Data Items के लिए Algorithm की Efficiency दर्शाने वाला एक फलन ज्ञात करते हैं जिससे किसी Algorithm की Efficiency या Complexity को Represent किया जाता है।

किसी Algorithm की Complexity या Efficiency को ज्ञात करते समय हमें सबसे पहले ये तय करना होता है कि हम Input की किस Property को Measure करने जा रहे हैं। किसी Algorithm की सबसे अच्छी Property वह होती है जो Algorithm के Efficiency Factor को प्रभावित करती हो।

हम यहां पर Algorithm की Key Property के रूप में Input Size शब्द को Use करके Algorithm को Analyze करेंगे और Input Size को Variable N से Represent करेंगे। फिर भी ये जरूरी नहीं है कि हम केवल एक ही Property का प्रयोग करें। हम किसी Algorithm की Efficiency को कई Properties के आधार पर एक फलन के रूप में Express कर सकते हैं।

लेकिन केवल एक ही Property के आधार पर किसी Algorithm की Efficiency को Express करना काफी सरल तरीका होता है तथा एक ही Property को सभी Properties के रूप में Represent किया जा सकता है। इसी तरीके को सबसे ज्यादा Use किया जाता है।

उदाहरण के लिए किसी List के विभिन्न Elements को Sort करने के लिए जितना समय Use होता है उसे List की Length के एक फलन के रूप में लिखा जा सकता है। Sorting के लिए हम जिस Algorithm का प्रयोग करते हैं वह Typically एक Quadratic Function होता है। यानी

```
TIME(n) = n * n
```

MergeSort व QuickSort के Algorithm अधिक तेज होते हैं। उनको फलन के रूप में निम्नानुसार लिखा जा सकता है–

```
TIME(n) = n * log (n)
```

इस Algorithm की Length Quadratic से कम लेकिन Linear से अधिक होती है।

ये तरीका अभी भी दूसरी समस्या का जवाब प्रदान नहीं कर रहा है। मानलो एक Algorithm किसी List में से किसी Required मान को Search कर रहा है। हम ये मान लेते हैं कि जो भी मान Search किया जा रहा है वह हमेंशा List में उपलब्ध होता है। ऐसा कोई मान Search नहीं किया जाता जो कि List में उपलब्ध ही ना हो। इस स्थिति में Algorithm हमेंशा Successful होता है।

अब इस Algorithm की Speed को प्रभावित करने वाला Factor वह स्थान है जहां पर Search किया जाने वाला मान प्राप्त होता है। यदि Algorithm वहां से शुरू होता है जहां पर Search किया जाने वाला मान उपलब्ध है तो Algorithm बहुत ही जल्दी समाप्त हो जाएगा। जबकि यदि Search किया जाने वाला मान List में कहीं अन्य स्थान पर है, तो Algorithm को अन्य स्थानों पर भी Search किए जाने वाले मान को खोजना होगा।

इस स्थिति में हम जो फलन यहां पर प्राप्त करना चाहते हैं वह पूरा नहीं है। ये फलन Input Size n पर निर्भर है। यदि Search किया जाने वाला Data तुरन्त प्राप्त हो जाता है तो ये Best Case स्थिति है लेकिन यदि Search किया जाने वाला Data काफी Comparisons के बाद प्राप्त होता है, तो इसे Worst Case स्थिति कहेंगे।

हमारे इस Algorithm में Best Case = Constant Time होगा जबकि Worst Case = Length of List या N-1 होगा। Average Case में Search किया जाने वाला Data List में प्रथम व अन्तिम स्थान के अलावा कहीं भी प्राप्त हो सकता है।

अब जब हम दो Algorithms को Compare करते हैं तो हमें दो Algorithms के फलनों को Compare करना पडता है। इस Analysis में हमें हमेंशा सावधान रहना होता है क्योंकि ये फलन कई स्थानों पर एक दूसरे को Cross कर सकते हैं।

यानी किसी एक Input के लिए पहला Algorithm उपयुक्त लगता है, जबकि किसी दूसरे Input के लिए पहले के स्थान पर दूसरा Algorithm अधिक उपयुक्त लगता है। यदि ऐसा होता है तो इसका मतलब है कि दोनों ही Algorithm सभी Inputs के लिए एक दूसरे से अधिक उपयुक्त या Efficient नहीं हैं।

दो Algorithms की वास्तव में उचित Comparing करने के लिए जरूरी है कि हमारे पास दोनों Algorithms के उचित फलन हों और हम ये पता कर सकें कि दोनों Algorithm किस Input के लिए एक दूसरे को Cross करते हैं। प्रायोगिक रूप से ये पता करना बहुत ही मुश्किल काम है। इसलिए हम दोनों Algorithm के Input का लगभग मान प्राप्त करते हैं जहां पर दोनों Algorithms एक दूसरे को Cross करते हैं। इस प्रक्रिया को **"Ballpark Estimate"** कहा जाता है।

"Ballpark Estimate" किसी Algorithm का लगभग Complexion होता है जिसे **Asymptotic Complexity** कहा जाता है। Asymptotic Complexity किसी Algorithm का मुख्य Term होता है जिससे किसी Algorithm के फलन की Limit का पता चलता है। उदाहरण के लिए मानलो कि हमारे पास दो Algorithms हैं:

Algorithm

1	A1 - Efficiency(n) = 29 [Constant Time]
2	A2 - Efficiency(n) = 3 + n/2 [Linear Time]

जब तक n = 52 होता है तब तक A1(n) > A2(n) होता है। यानी जब n का मान 52 होता है तब दोनों Algorithm की Complexity समान होती है। इस Limit पर दोनों Algorithm एक–दूसरे को Cross करते हैं।

हालांकि एक बहुत ही छोटी List के लिए A2 Algorithm काफी Efficient है लेकिन यदि List बडी हो तो A1 Algorithm A2 Algorithm की तुलना में अधिक Efficient रहता है। Asymptotic Complexity के पीछे यही Idea है।

मानलो कि एक Array में 52 Items के साथ 29 Operations करने पर Required काम हो जाता है। इस स्थिति में दोनों ही Algorithms को एक दूसरे के स्थान पर Use किया जा सकता है। लेकिन यदि Data Items की संख्या 52 से कम हो तो पहला Algorithm अधिक Efficient होगा जबकि यदि Data Items की संख्या 52 से अधिक हो तो दूसरा Algorithm पहले की तुलना में अधिक Efficient होगा।

Asymptotic Complexity में फलन के सभी Lower Order Terms व Constant Multipliers को Ignore कर दिया जाता है, इसलिए सभी Linear फलनों को "O" Notation में O(n) लिखा जाता है और सभी Constant Time फलन को "O" Notation में O(1) लिखा जाता है। यानी A1 की Asymptotic Complexity O(1) है जबकि A2 की Asymptotic Complexity O(n) है।

स्पष्ट रूप से हम कह सकते हैं कि Asymptotic Complexity तभी उपयुक्त है जब हमें बहुत बडे Input के साथ प्रक्रिया करनी हो। किसी Application Program में हमें तब अधिक सटीकता से Analysis करना पडता है जब हम कम Input Data के साथ प्रक्रिया कर रहे होते हैं।

हमने अभी तक विभिन्न प्रकार के फलनों को देखा है। इन सभी फलनों द्वारा हम किसी Algorithm को Analyze करते हैं और Algorithm की Efficiency या Complexity ज्ञात करते हैं। Efficiency या Complexity को प्रदर्शित करने के लिए हम Capital Letter "O" का प्रयोग करते हैं। विभिन्न प्रकार के फलनों को Represent करने के लिए हम "O" Notation का प्रयोग करते हैं। इस "O" Notation को Use करके किसी फलन की Efficiency या Complexity को Represent करने की प्रक्रिया को **Big – O Notation** कहा जाता है।

जब हम Big – O Notation द्वारा किसी Algorithm की Complexity के फलन को Represent करते हैं तब फलन के विभिन्न Low – Order Terms, Dominant Terms व Constants Coefficients को Ignore कर देते हैं व केवल Highest Power या Highest Value Represent करने वाले Notation को ही **Big – O** Notation के रूप में व्यक्त करते हैं। जैसे **Efficiency(n)** को यदि **Big – O** Notation के रूप में व्यक्त करना हो तो हम इसे केवल **O(n)** लिखते हैं।

Analyzing Algorithms

Analysis के लिए हमेंशा Simple Statements Sequence को आधार बनाना चाहिए। सबसे पहले ये Note करें कि Statements की एक Sequence जो कि एक समय में केवल एक बार Execute होती है, उसे Big – O Notation के रूप में Represent करने के लिए हमें केवल O(1) लिखना होता है। क्योंकि एक Statement Execute होने में जितना समय लगाता है सभी Statements Execute होने में उसी अनुपात में समय लगाएंगे। उदाहरण के लिए यदि किसी Data Item की Size n को किसी Loop द्वारा निम्नानुसार Solve किया जा सकता है :

```
for(i=0; i<n; i++)
{
    Statement x;
}
```

तो इस Loop में Statement x एक **O(1)** Sequence Statement है इसलिए इस Loop की Time Complexity **n O(1)** या **O(n)** होगी।

यदि हमारे पास निम्नानुसार दो Nested Loop हों तो

```
for(i=0; i<n; i++)
{
    for(j=0; j<n; j++)
    {
        Statement x;
    }
}
```

इस स्थिति में Loop i के हर Iteration में Loop j **n** बार Iterate होता है। यानी ये Nested Loop n * n बार चलेगा इसलिए इस Loop में लगने वाला समय **$O(n^2)$** के बराबर होगा। निम्न Loop तब तक चलता है जब तक कि n का मान h के मान से बडा या बराबर रहता है–

```
h = 1;
for(i=0; i<=n; i++)
{
    Statement x;
    h = 2 * h;
}
```

इस Loop की Complexity 1 + log n के बराबर है। इसे Big – O Notation के रूप में **$O(\log_2 n)$** लिख सकते हैं।

यदि Inner Loop Outer Loop के Index पर निर्भर हो तो Inner Loop में j का मान 0 से n तक चलता है। इस स्थिति में Loop की Complexity निम्नानुसार ज्ञात की जा सकती है–

```
for(i=0; i<=n; i++)
{
    for(j=0; j<=i; j++)
    {
        Statement x;
    }
}
```

f(j) = 0 to n यानी
= 1 + 2 + 3 + 4 + . . . (n-2) + (n-1) + n यानी
= n * (n + 1) / 2
= $(n^2 + n)/2$

इस Algorithm की Complexity को **Big – O** Notation के रूप में हम **$O(n^2)$** लिख सकते हैं। यदि हम निम्नानुसार किसी जरूरत के अनुसार Loop चलाते हैं:

```
h = n;
for(i=0; i<=h; i++)
{
    for(j=0; j<=n; j++)
    {
        Statement x;
    }
h = h/2;
}
```

तो यहां Outer Loop में **$\log_2$ n** Iterations होंगे और Inner Loop की Complexity **O(n)** होगी। इस स्थिति में कुल Complexity **O(n log n)** होगी। ये Complexity पहले वाले Loop की Complexity की तुलना में अधिक Efficient है।

चलिए, अब हम कुछ Program देखते हैं और उनकी Complexity ज्ञात करने की कोशिश करते हैं। हालांकि इन Programs का वास्तव में कहीं कोई उपयोग नहीं है। ये केवल कुछ Pattern मात्र Create करते हैं। इन उदाहरणों में Programs की Efficiency printf() Function के कुल Executions की संख्या पर आधारित है जो कि n का फलन है।

Example

```
main(){
    int i, j, n;
    printf("Enter the limit of Pattern");
    scanf("%d", &n);

    for(i=0; i<n; i++){
        for(j=0; j<n; j++){
            printf("%4d, %4d\n", i, j);
        }
    }
    getch();
}
```

हम देख सकते हैं कि दोनों Loops में केवल Inner Loop में ही Execute होने वाला Statement है और केवल एक ही Statement है। हमारे Complexity फलन हमें ये बताएगा कि n के फलन के रूप में printf() Function कितनी बार Execute होगा।

इस पूरे Program की Complexity Outer Loop की Complexity के बराबर है क्योंकि यही एक Top Level का Statement है जिसमें printf() Function Use किया गया है। किसी Loop की Complexity को हम निम्न सूत्र से जान सकते हैं–

Complexity of Loop = (Number of Loop's **Repetitions**) * (Complexity of each **Iteration**)

हम मान सकते हैं कि Loop के हर Iteration की Complexity समान है। ऐसा हम Outer Loop के लिए मान सकते हैं इस Loop की Complexity को निम्नानुसार Big – O Notation के रूप में लिख सकते हैं:

O(n) * **O(** Complexity of Inner Loop **)**

Inner Loop में भी केवल एक ही printf() Statement है इसलिए Inner Loop के लिए भी ये माना जा सकता है कि Inner Loop के सभी Iteration की Complexity समान है। यानी

Complexity of Inner Loop = **O(n)** * **O(** printf **)**

Printf() Statement केवल एक ही बार लिखा गया है इसलिए स्वयं printf() Statement की Complexity **O(1)** है। सभी मानों को एक साथ लिखने पर हमें इस Program की Complexity निम्नानुसार प्राप्त होती है–

f(n) = O(n) * O(n) * O(1)
= O(n * n)
= **O(n2)**

चलिए, एक और Sample Program की Complexity ज्ञात करते हैं। Program निम्नानुसार है:

Example

```
main()
{
    unsigned int i, j, n;
    printf("Enter a Nonzero Positive Value as a limit : ");
    scanf("%d", &n);

    for( ; n>1; )
    {
        n = n/2;
        printf("%d\n", n);
    }
getch();
}
```

इस Loop की Complexity ज्ञात करने के लिए भी हम वही Formula Use कर सकते हैं जिसे पिछले Program के लिए Use किया है। इस Loop के हर Iteration की Complexity समान है इसलिए इसे हम **O(1)** के रूप में लिख सकते हैं। Loop कितनी बार Iterate होगा इसे ज्ञात करने के लिए हमें ये ज्ञात करना होगा कि हम किसी Number n को कितनी बार दो भागों में बांट सकते हैं जबकि n का मान 1 से बडा रहे। मानलो कि n के हर मान को K बार दो भागों में विभाजित किया जा सकता है तो इसे हम निम्नानुसार लिख सकते हैं:

$2^K <= n$

ये तो निश्चित है कि जब हम 2^K में 2 का भाग देते हैं तो K-1 बार 2^K का मान 2 होता है और जब हम इसे एक बार और 2 से विभाजित करते हैं तो 2^K का मान 1 हो जाता है। 2^K का मान 1 होते ही Loop Terminate हो जाता है। इसलिए जब n का मान 2^K के बराबर हो **($n = 2^K$)** तब Loop **K** बार Repeat होगा।

चूंकि **$n = 2^K$** है इसलिए ये Loop n के विभिन्न मानों के लिए कई बार Repeat होगा लेकिन ये Loop K + 1 Times Repeat नहीं हो सकता क्योंकि K का मान K + 1 के मान से कम होता है। इसलिए यदि **n, K** व घात **K+1** से Bounded है तो ये Loop K बार चलता है। K व n के बीच में ये सम्बंध है कि K उस **log** का Integer Part है जिसका आधार 2 है। इसलिए इस उदाहरण की Complexity निम्नानुसार होगीः

O(log n) * O(1) = O(log n)

यदि हम log के आधार 2 के स्थान पर आधार 10 को Use करें तो भी Complexity पर किसी प्रकार का कोई असर नहीं पडता है।

Example

```
main(){
    unsigned int i, j, n;
    printf("Enter a Nonzero Positive Value as a limit : ");
    scanf("%d", &n);

    for( i=1, m=n+66; i<=m; i++ ) {
        printf("%d\n", i );
    }

    for( j=n/21, m=n/5; j<=m; j++ ) {
        printf("%d\n", j );
    }
getch();
}
```

यहां दो Independent Loops में दो printf() Statements हैं। इसलिए इस Program की कुल Complexity दोनों printf() Statements की कुल Complexity के योग के बराबर होगी। यदि इस Program को p से प्रदर्शित किया जाए तो इस Program की Efficiency को हम निम्नानुसार Big–O Notation के रूप में प्रदर्शित कर सकते हैं–

Efficiency**(p)** = **O(** Outer Loop **)** + **O(** Inner Loop **)**

Outer Loop dh Efficiency = **O(n)**

Inner Loop dh Efficiency	=	**O(n)**
Total Efficiency	=	**O(n) + O(n)**
	=	**O(n)**

चलिए, एक और उदाहरण देखते हैं। ये उदाहरण सबसे पहले उदाहरण के समान ही है लेकिन इसमें Loop के Iteration Complexity में अन्तर है। Program निम्नानुसार है:

Example

```
main()
{
    int i, j, n;
    printf("Enter the limit of Pattern");
    scanf("%d", &n);

    for(i=1; i<=n; i++)
    {
        for(j=1; j<=n; j++)
        {
            printf("%4d, %4d\n", i, j);
        }
    }
    getch();
}
```

इसकी Efficiency हम निम्नानुसार ज्ञात कर सकते हैं–

Efficiency(Outer Loop)	=	SUM(i, 1, n) of (Efficiency of Ith Iteration)
	=	SUM(i, 1, n) of (n-i)
	=	n * (n-1) / 2
	=	O(n * n)
	=	**O(n^2)**

Inserting and Deleting

किसी Array में यदि जगह उपलब्ध हो तो नई इकाई को Array के अन्त में जोडना काफी आसान होता है। लेकिन जब हमें Array के किसी विशेष Index Number पर मान को Insert करना होता है तो इसके लिये Array के जिस Element के बाद नई इकाई जोडनी है, उससे बाद के सारे Elements को एक–एक स्थान आगे प्रतिस्थापित किया जाता है। फिर नए मान को Array में जोडा जाता है।

यदि हम Array में मान Insert करने से पहले जिस स्थान पर मान Insert करना है, उससे आगे के सभी मानों को प्रतिस्थापित नहीं करते हैं तो हमारा नया मान पुराने मान पर Over Write हो जाता है।

इसी तरह से किसी Array के अन्तिम Element को Delete करना काफी आसान होता है लेकिन जब किसी Array के किसी अन्य Element को Delete किया जाता है, तो Array के उस Element से आगे के सभी Elements को एक स्थान पीछे प्रतिस्थापित करना पडता है। यदि ऐसा ना किया जाए तो जिस स्थान के मान को Delete किया गया है उस स्थान पर Garbage मान Store हो जाता है।

मानलो कि Name एक 10 Elements का Linear Array है जिसमें 6 नाम Stored हैं। हम चाहते हैं कि चौथे नाम के बाद एक नया नाम Add करें। इस स्थिति में हमें Array के चौथे Data item के बाद एक जगह बनानी होगी। जगह बनाने के लिए Array के पांचवे व छठे Data Item को Move करके छठे व सातवें स्थान पर Mode करना होगा। उसके बाद नए Data Item को चौथे स्थान पर Insert करना होगा। इस पूरी प्रक्रिया का Algorithm हम निम्नानुसार लिख सकते हैं–

माना एक Array LArray[N] है जिसमें N Items हैं। इस Array के Index Number K पर एक Element ITEM को Insert करना है जबकि हम ये मान कर चलते हैं कि इस Array में अभी इतना स्थान है कि हम इसमें नया Item Insert कर सकें।

चूंकि हमें Index Number K पर नया ITEM Insert करना है इसलिए हमें Index Number K को खाली करना होगा ताकि नया Data इसमें Store हो सके। चूंकि नया Data Store करने के लिए हम Index Number K पर जगह बना रहे हैं इसलिए हमें Index Number K से Array के अन्तिम Data Items तक के सभी Data Items को एक स्थान Right में Move करना होगा। इस प्रक्रिया को हम निम्न चित्र द्वारा समझ सकते हैं:

10	25	32	45	95	75				

माना K का मान 4 है तो हमें Index Number 4-1 = 3 को खाली करना होगा। ऐसा करने पर Index Number 4 के बाद के सभी Data Items को एक स्थान Right में Move करना होगा। ऐसा करने पर ये Array निम्नानुसार दिखाई देगा–

10	25	32	45	45	95	75			

अब हम Index number 4 पर नया ITEM Insert कर सकते हैं। Insert करने का Algorithm निम्नानुसार हो सकता है–

Algorithm of Inserting

1 START
2 DECLARE LArray[N], I, K, ITEM

```
3   REPEATE FOR I = N-1 TO I >= K STEP I = I - 1
4   SET LArray[I] = LArray[I-1]                    [Shift Data Items to Right]
          [End of the Loop]
5   SET LArray[K] = ITEM                           [Insert Element]
6   END
```

इसी तरह माना कि एक Array LArray[N] है जिसमें N Items हैं। इस Array के Index Number K पर स्थित Element को Delete करना है। चूंकि हम Index Number K पर स्थित Item को Delete कर रहे हैं इसलिए हमें Index Number K के बाद के सभी Data Items को एक स्थान पीछे की तरफ Move करना होगा। इस प्रक्रिया को हम निम्न चित्र द्वारा समझ सकते हैं–

10	25	32	100	45	95	75			

यदि हम K का मान 5 मानें तो Index Number 5-1 = 4 के Data Item को Delete करना है। जब हम Index Number 4 के Data Item को Delete करना चाहते हैं तो हमें बस इतना ही करना है कि Index Number 4 के Data Item पर Index Number 5 के Data Item को Place कर दें। यानी Index Number 4 के बाद के सभी Data Items को एक स्थान आगे सरका दें। ऐसा करने पर ये Array निम्नानुार दिखाई देगाः

10	25	32	100	95	75	0			

किसी Linear Array से Deletion का Algorithm निम्नानुसार हो सकता है–

Algorithm

```
1   START
2   DECLARE LArray[N], I, K
3   SET ITEM = LArray[K]
4   REPEATE FOR I = K TO N-1 STEP I = I + 1
5   SET LArray[I] = LArray[I+1]      [Shift Data Item to Left]
                                     [End of the Loop]
6   END
```

इन दोनों Algorithms का हम निम्नानुसार प्रयोग करके Program बना सकते हैं–

Program

```
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#define SIZE 10
main()
```

```
{
    int Array[SIZE] = {1,10,20,0};
    int i, j, item, id;
    char choice;

    while(1)
    {
        printf("\n1.   Insert Data Item");
        printf("\n2.   Delete Data Item");
        printf("\n3.   Display Data Item");
        printf("\n4.   Exit");
        printf("\n\nEnter Your Choice  ");
        scanf("%d", &choice);

    switch(choice)
    {
        case 1:        //Insertion Operation on the Array
            printf("Enter Index Number [0 to 9] ");
            Label:
            fflush(stdin);
            scanf("%d", &id);
            if(id < 0 || id > SIZE-1)
            {
                    printf("Index Number Must Be BETWEEN 0 to 9 ");
                    goto Label;
            }
            printf("Enter Value  ");
            scanf("%d", &item);

        for(i = SIZE-1; i >= id; i--)
            Array[i] = Array[i-1];

            Array[i+1] = item;
        break;

        case 2:        //Deletion Operation on the Array
            printf("Enter ID  ");
            scanf("%d", &id);

            for(j=id; j<SIZE; j++)
            Array[j] = Array[j+1];
              Array[j-1] = 0;
            break;

        case 3:        //Traversing Operation on the Array
```

```
            for(i=0; i<SIZE; i++)
            printf("Value at ID %d is %d \n", i, Array[i]);
        break;

        case 4:
            exit(1);
        }
    }
}
```

Sorting

किसी Data Structure के सभी Data को एक व्यवस्थित क्रम में रखना Sorting कहलाता है। ये दो प्रकार की होती है, आरोही क्रम में या अवरोही क्रम में। आरोही क्रम में सबसे कम मान का Data List की शुरूआत में व सबसे अधिक मान का Data List के अंत में Store होता है, जबकि अवरोही क्रम में ठीक इसके विपरीत क्रिया होती है। यानी सबसे अधिक मान का Data सबसे पहले व सबसे कम मान का Data List में सबसे बाद में Store होता है। Data Processing के अंतर्गत Sorting को मुख्यतः तीन भागों में बांटा गया है:

Bubble Sort

यह अत्यधिक काम में आने वाली सबसे साधारण तकनीक है। इसमें किसी भी Array के प्रथम मान की तुलना Array के दूसरे मान से करते हैं। यदि Array का दूसरा मान प्रथम मान से बडा है, तो आरोही क्रम में जमाने के लिये दूसरे Data को प्रथम स्थान पर रख दिया जाता है व प्रथम स्थान के Data को दूसरे स्थान पर।

फिर Array के दूसरे मान की तुलना तीसरे मान से करते हैं और यदि तीसरा मान दूसरे मान से बडा है तो तीसरे मान की जगह दूसरा मान व दूसरे मान की जगह तीसरा मान रख दिया जाता है। यदि दूसरा मान तीसरे मान से बडा नहीं है तो Array के दूसरे व तीसरे मानों के स्थान में कोई परिवर्तन नहीं किया जाता है।

मानों के स्थान परिवर्तन का ये क्रम तब तक चलाया जाता है, जब तक कि सारे मान आरोही क्रम में व्यवस्थित ना हो जाए। ये क्रम N-1 बार चलाया जाता है, जहां N Array के कुल मानों की संख्या है। Bubble Sort का Algorithm निम्नानुसार है:

Algorithm

Here LArray[N] is an Array with N Elements. This Algorithm SORTS the Data Items of the Array

1 START
2 REPEATE FOR I = 1 To N – 1 STEP I = I + 1[Outer Loop]

```
3    REPEATE FOR J = 1 To N – I STEP J = J + 1 [ Inner Loop ]
4    IF LArray[ J ] > LArray[ J + 1 ]
5    LArray[ J ] = LArray[ J + 1 ]                    [ Interchange Data Items ]
     [ End of Inner Loop ]
     [ End of Outer Loop ]
6    End
```

Bubble Sort तब बहुत उपयोगी होता है जब List लगभग Sorted हो और केवल कुछ ही इकाईयों की Sorting करनी हो। जब इकाईयों की संख्या अधिक होती है, तब इस विधि में Program की गति काफी कम हो जाती है, क्योंकि N-1 बार List को व्यवस्थित करना पडता है। इस Algorithm का प्रयोग करते हुए हम निम्नानुसार एक Function बना सकते हैं जिसे किसी भी Main Function में Call किया जा सकता है। Function निम्नानुसार हैः

Function

```
void BubbleSort(int *Array, int size)
{
    int i, j, temp;
  for(i=0; i<size; i++)
  {
    for(j=0; j<size-i; j++)
     {
    if(Array[j]>Array[j+1])
        {
        temp = Array[j];
            Array[j] = Array[j+1];
            Array[j+1] =temp;
        }
     }
  }
}
```

इस Function में दो Argument Calling Function से आते हैं। पहले Argument में एक Array का Base Address आता है जिसके विभिन्न Elements को Sorting Order में रखना है और दूसरे Argument में Array की Size प्रदान की जाती है। चूंकि इस Function में Array का Base Address आता है इसलिए इस Function द्वारा जो Sorting होती है वह Calling Function के Array की Sorting होती है।

Bubble Sort के Algorithm से हम समझ सकते हैं कि ये Algorithm List में से सबसे छोटे Data Item को Search करता है और उस Data Element को उसकी Final Position पर Place कर देता है। फिर दूसरे Iteration में दूसरे Element से शुरू करता है और बचे हुए Elements में से सबसे छोटे Data Item को खोज कर उसकी Final Position पर भेजता है।

Algorithm में हम देख सकते हैं कि यदि Data Items की संख्या n हो तो Outer Loop n बार चलता है और Outer Loop के आधार पर Smallest Data Find करने के लिए Inner Loop n-i बार Comparison करता है। यानी कुल Comparisons की संख्या निम्नानुसार होती है:

$$\begin{aligned} \text{Elements } E(n) &= \sum_{i=1}^{n} (n-i) \\ &= (n-1) + (n-2) + (n-2) + \ldots + 2 + 1 \\ &= n(n-1)/2 \\ &= \mathbf{O(n^2) - O(n/2)} \\ &= \mathbf{O(n^2)} \end{aligned}$$

सारांश में कहें तो Bubble Sort Algorithm की Complexity $o(n^2)$ होती है।

Selection Sort

इस प्रकार की Sorting में List की प्रथम इकाई से शुरू करके पूरी List में न्यूनतम मान को खोजा जाता है और उस मान को List के प्रारंभ में रख दिया जाता है। उसके बाद शेष List में से दूसरे स्थान के लिये न्यूनतम मान को खोजा जाता है और प्राप्त मान को दूसरे स्थान पर रख दिया जाता है। फिर तीसरे स्थान के लिये यही क्रम अपनाते है और तीसरे स्थान पर शेष List में से प्राप्त न्यूनतम मान को रख दिया जाता है। ये क्रम तब तक चलता है, जब तक कि पूरी List की Sorting नहीं हो जाती। इसका Algorithm निम्नानुसार होता है:

Algorithm

Here LArray[N] is an Array with N Elements. This Algorithm SORTS the Data Items of the Array

```
1  START
2  REPEATE FOR I = 1 To N – 1 STEP I = I + 1                    [Outer Loop]
3  REPEATE FOR J = I + 1 To N – 1 STEP J = J + 1 [ Inner Loop ]
4  IF LArray[ I ] > LArray[ J ]
5  LArray[ I ] = LArray[ J ]                                    [ Interchange Data Items ]
       [ End of Inner Loop ]
       [ End of Outer Loop ]
6  End
```

इस Algorithm की Complexity इस बात पर निर्भर करती है कि Data Structure में Data किस तरह से Organized हैं। यदि Data लगभग Sorted हो तो इस Algorithm की Complexity कम होती है। लेकिन फिर भी इस Algorithm की Complexity $O(n^2)$ से अधिक नहीं हो सकती। यानी Data का अधिकतम Comparison n^2 बार हो सकता है। इस Algorithm का प्रयोग करके हम निम्नानुसार Sorting का Function बनाकर अपने Program में Use कर सकते हैं।

Program

```
void SelectionSort(int *Array, int size)
```

```
{
    int i, j, temp;
  for(i=0; i<size; i++)
  {
    for(j=i+1; j<size; j++)
     {
    if(Array[i]>Array[j])
         {
        temp = Array[i];
             Array[i] = Array[j];
             Array[j] =temp;
         }
     }
  }
}
```

Insertion Sort

इस Method में सम्पूर्ण List को दो भागों Sorted Part व Unsorted Part मे बांटा जाता है। List के Unsorted Part से एक इकाई लेकर Sorted Part में उपयुक्त स्थान पर रखते हैं। यह प्रक्रिया तब तक चलाते हैं जब तक कि Unsorted Part की सभी इकाईयां Sorted Part में व्यवस्थित नहीं हो जाती। इसका Algorithm निम्नानुसार होता है। इस Method में दो पद होते हैं।

Sorted भाग को पढ कर उस स्थान को चिन्हित करना होता है जहां पर Unsorted भाग से इकाई लाकर Insert करवाना है। इस प्रक्रिया में इकाई के लिये स्थान बनाने के लिये शेष इकाईयों को Right Side में Shift करना होता है। उसके बाद बनाए गए स्थान में इकाई को प्रवेश करवा देते हैं। इस Method का Algorithm निम्नानुसार होता हैः

Algorithm

Here LArray[N] is an Array with N Elements. This Algorithm SORTS the Data Items of the Array

```
1   START
2   REPEATE FOR I = 1 To N – 1 STEP I = I + 1          [ Outer Loop ]
3   REPEATE FOR J = 0 To I STEP J = J + 1              [ Inner Loop ]
4   IF LArray[ J ] > LArray[ I ]
5   TEMP = LARRAY[ J ]
6   LArray[ J ] = LArray[ I ]
7   REPEATE FOR K = I To J STEP K = K + 1              [ Inner Loop ]
8   LARRAY[ K ] = LARRAY[ K – 1 ]
9   LARRAY[ K + 1 ] = TEMP
        [ End of Inner Loop ]
        [ End of Inner Loop ]
```

```
        [ End of Outer Loop ]
10  End
```

इस विधि से Sorting तब की जानी चाहिये जब इकाईयों की संख्या कम होती है। जब इकाईयों की संख्या अधिक होती है, तब ये विधि ठीक नहीं रहती क्योंकि इकाईयों के लिये जगह बनाने में समय लगता है जिससे Program की गति कम हो जाती है। इस Algorithm की Complexity में कम से कम n–1 बार Comparison करना पडता है। इस Algorithm का प्रयोग करके हम निम्नानुसार एक Function बना सकते हैं जिसे किसी भी Program में Use किया जा सकता है:

Program

```
void sort(int *Array, int size)
{
    int i, j, k, temp;
  for(i=0; i<size; i++)
  {
    for(j=0; j<i; j++)
     {
    if(Array[j]>Array[i])
         {
        temp = Array[j];
            Array[j] = Array[i];

            for(k=i; k>j; k--)
            Array[k] = Array[k-1];

            Array[k+1] = temp;
         }
     }
  }
}
```

Insertion Algorithm का प्रयोग तब किया जा सकता है जब Data Items **n** की संख्या कम हो। इस Algorithm का Inner Loop i-1 Comparisons करता है। यानी

$$\begin{aligned} \text{Elements } E(n) &= \sum_{i=1}^{n} (i-1) \\ &= 1 + 2 + 3 + \ldots + n-1 \\ &= n(n-1)/2 \\ &= \mathbf{O(n^2) - O(n/2)} \\ &= \mathbf{O(n^2)} \end{aligned}$$

Average Case में इस Algorithm की Complexity को निम्नानुसार दर्शाया जा सकता है:

$$\text{Elements } E(n) = \sum_{i=1}^{n} (i-1)/2$$

= (1 + 2 + 3 + . . . + n-1)/2
= n(n-1)/4
= **$O(n^2)$**

हम देख सकते हैं कि दोनों ही स्थितियों में इस Algorithm की Complexity **$O(n^2)$** होती है।

एक प्रोग्राम बनाओ जो दस अंक Input ले और उसे आरोही क्रम में Output में Print करें।

Program

```
#include<stdio.h>
main()
{
int a[10], n, j, i;
clrscr();

for(i=0;i<10;i++)
{
     printf("enter the a:");
     scanf("%d", &a[i]);
}

for(i=0;i<10;i++)
{
     for(j=0; j<=i; j++)
     {
          if(a[i]<a[j])
          {
            n=a[i];
            a[i]=a[j];
            a[j]=n;
          }
     }
}
for(i=0;i<10;i++)
     printf("\n%d", a[i]);
getch();
}
```

एक प्रोग्राम बनाओ जो दस अंक Input ले और उसे अवरोही क्रम में Output में Print करें।

Program

```
#include<stdio.h>
```

```
main()
{
int a[10], n, i, j, sum = 0;
clrscr();
for(i=0; i<10; i++)
{
 printf("Enter the a:");
 scanf("%d", &a[i]);
 }

for(i=0;i<10;i++)
{
  for(j=0; j<=i; j++)
  {
      if(a[i]>a[j])
      {
      n=a[i];
      a[i]=a[j];
      a[j]=n;
      }
  }
}

for(i=0;i<10;i++)
     printf("%d\n", a[i]);

sum = sum + a[j+i];
printf("sum = %d", sum);

getch();
}
```

Searching

जिस प्रकार से किसी Telephone Directory में किसी नाम से Telephone Number खोजते हैं या किसी Telephone Number से नाम खोजते हैं, या किसी student के Roll Number द्वारा Student का Record खोजा जाता है, ठीक इसी प्रकार से एक Key द्वारा किसी भी Data Structure से उस Key से सम्बंधित सारी जानकारी प्राप्त की जा सकती है।

जैसे यदि हमें Telephone Directory से 223344 Number किस व्यक्ति का है और वह व्यक्ति कहां रहता है, ये जानना हो, तो हम इस Number को Telephone Directory के हर Number से Compare करते हैं, और जहां ये Comparison एकदम मेल करती है, सम्बंधित व्यक्ति का नाम

पता आदि हम जान लेते हैं। इस प्रकार से ये Telephone Number एक Key के रूप में Use किया जाता है, जो बाकी की सम्बंधित जानकारी दे देता है। यही प्रक्रिया हम Data Structure के साथ भी करते हैं। Computer में Searching दो प्रकार की होती हैः

Internal Search

जब सभी Records Computer की Main Memory में होते हैं, तो Main Memory से की जाने वाली Searching Internal Search कहलाती है।

External Search

जब Records बडे होते हैं व Records की संख्या अधिक होती है, तब Records को Hard disk पर संग्रहित कर लिया जाता है। फिर इस Hard Disk से Searching की जाती है। इस Searching को External Searching कहा जाता है। Internal Searching के भी दो भाग हैं।

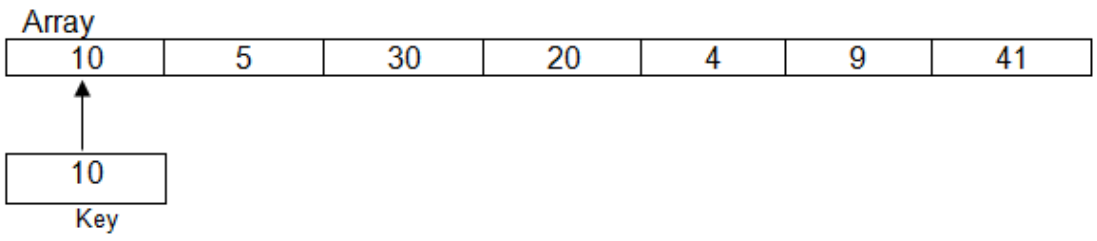
Linear Searching

किसी अव्यवस्थित List में से Searching करने की यह सबसे सरल विधि होती है। जैसे किसी Data Structure Array में हमें ये खोजना हो, कि अंक 10 उपस्थित है या नहीं। यह जानने के लिये एक Key Variable लेंगे और उस Key का मान 10 कर देंगे। फिर क्रम से इस Key के मान की तुलना Array के हर Element से करेंगे। Array के हर Element से Key के मान की तुलना का ये क्रम तब तक चलता रहता है, जब तक कि Array के किसी Element का मान 10 प्राप्त ना हो जाए या फिर Array के सभी Elements का अंत ना हो जाए। इसका Algorithm निम्नानुसार लिख सकते हैं। माना पुर्णांक मानों का एक Array **ARRAY[N]** है।

Algorithm

```
1   LET COUNT = 1; and  FOUND =0;
2   While ( COUNT < N ) DO
        If ARRAY[COUNT] =  VALUE then FOUND = COUNT
        COUNT = COUNT + 1;
3   Print FOUND
4   END
```

यह एक अच्छी तकनीक है, लेकिन इस तकनीक में Array के हर Element को Check करना पडता है, जिससे Program की गति कम हो जाती है। इसे निम्न चित्र में दिखाया गया है–



इस Algorithm पर आधारित एक प्रोग्राम देखते हैं, जिसमें एक Array में Store विभिन्न अंकों में से मनचाहे अंक को खोजना है, कि अमुक अंक List में उपलब्ध है या नहीं?

Program

```
#include<stdio.h>
main()
{
int j, k[10] , key, found=-1;
clrscr();
for ( j = 0; j<10; j++)
{
      printf("Enter %d digit  ", j+1);
      scanf("%d", &k[j] );
}
printf("\n Which Number Do you want to FIND ");
scanf("%d", &key);

for ( j = 0; j<10; j++)
{
      if( k[j] == key )
      found=key;
}

if(found > -1 )
      printf("\n Number is present in The List ");
if(found < 0)
      printf("\n The Number is not Present in the List ");
getch();
}
```

Output 1

```
Enter 1 digit  4
Enter 2 digit  344
Enter 3 digit  344
Enter 4 digit  34
Enter 5 digit  434
Enter 6 digit  434
Enter 7 digit  34
Enter 8 digit  555
Enter 9 digit  55
Enter 10 digit 5
Which Number Do you want to FIND 2
```

```
The Number is not Present in the List
```

Output 1

```
Enter 1 digit  1
Enter 2 digit  2
Enter 3 digit  3
Enter 4 digit  4
Enter 5 digit  5
Enter 6 digit  6
Enter 7 digit  7
Enter 8 digit  87
Enter 9 digit  89
Enter 10 digit  9

Which Number Do you want to FIND 9
Number is present in The List
```

इस प्रोग्राम में Linear Searching की गई है। सबसे पहले एक Array में मानों को Input किया गया है। फिर एक Key नाम के Variable में वो अंक लेते हैं जिसे List में खोजना है। इस अंक को Loop द्वारा Array के हर Element से Check किया जाता है।

यदि key का अंक Array में प्राप्त होता है तो if condition सत्य हो जाती है और Found नाम के Variable में key का अंक प्राप्त हो जाता है। यदि key का अंक list में प्राप्त नहीं होता है, तो found का मान -1 जो कि found को Declare करते समय ही दे दिया गया था, रहता है।

यदि List में key का मान मिल जाता है तो found का मान -1 से अधिक रहता है और Output में Massage मिलता है कि List मे अंक उपलब्ध है। यदि List में अंक उपलब्ध नहीं होता है तो found का मान -1 होने के कारण 0 से छोटा रहता है। इसलिये Output में Massage प्राप्त होता है कि जो अंक खोजा जा रहा है वह अंक List में उपलब्ध नहीं है। इस प्रकार से Linear Searching की जाती है।

Linear Search Algorithm की Complexity को भी हम f(n) Function का प्रयोग करके पता कर सकते हैं। यदि किसी Linear Data Structure में n Data हों तो उस Data Structure से किसी ITEM को Find करने के लिए Worst Case Algorithm में हमें अधिकतम n+1 बार Comparisons करने होंगे, जबकि यदि Data ITEM Data Structure के अन्त में हो या ITEM Data Structure में ना हो। जबकि Average Case Algorithm में ITEM के प्राप्त होने की सम्भावना n+1/2 होती है।

Binary Searching

ये अत्यधिक उपयुक्त तकनीक है लेकिन इसका प्रयोग केवल Sorted List पर ही किया जा सकता है। Linear Searching की तुलना में Binary Searching बहुत Fast गति से काम करती है। माना कि

Lowest = List का न्यूनतम मान
Highest = List का अधिकतम मान
Mid = List का मध्यमान

Mid = Lowest + Highest / 2

इस प्रक्रिया में Key के मान की List के मध्यमान से तुलना की जाती है। यदि मध्यमान Mid Key के मान से अधिक होता है तो Key के मान की तुलना List के Lowest भाग से की जाती है।

Highest = Mid - 1

तथा Mid के नए मान को वापस **Mid = Lowest + Highest / 2** द्वारा प्राप्त किया जाता है। यदि Key का मान Mid के मान से अधिक होता है, तो Key के मान को List के Highest भाग में खोजा जाता है। यानी

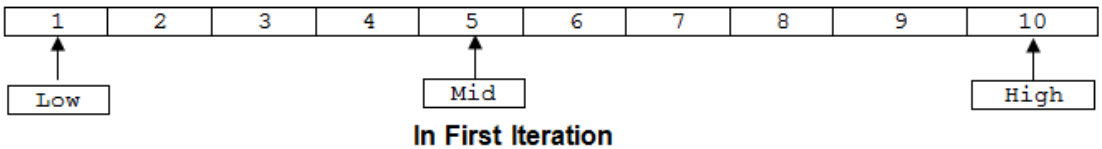
Lowest = Mid + 1

वापस से उपरोक्त सुत्र द्वारा Mid का नया मान प्राप्त किया जाता है। ये क्रम तब तक चलता रहता है जब तक कि Key का मान प्राप्त ना हो जाए या List समाप्त ना हो जाए। Binary Search की Algorithm निम्नानुसार है।

माना N Elements की एक Sorted List **int X[N]** है। यदि Key का मान List में मिल जाता है, तो उसे FOUND में Store करना है। निम्न चित्र द्वारा इसे समझाने की कोशिश की जा रही है। जिसमें 1 को खोजा जा रहा है। यानी x = 1 है।

```
Array[10]
low = 0;
high = n-1; so high = 10-1 = 9
mid = 0 + 9 / 2
mid = 4
```

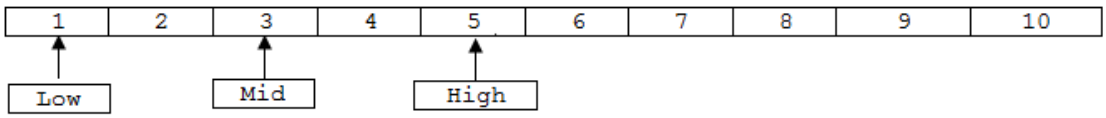
Index Number 4 पर मान 5 है इसलिए x का मान array[mid] यानी Array[4] के बराबर नहीं है।



In First Iteration

x का मान mid से छोटा है इसलिए Key का मान Array के Lower Part में होगा। इसलिए Loop के Second Iteration में low का मान 0 ही रहेगा लेकिन high का मान Change होकर mid+1 हो जाएगा। यानी

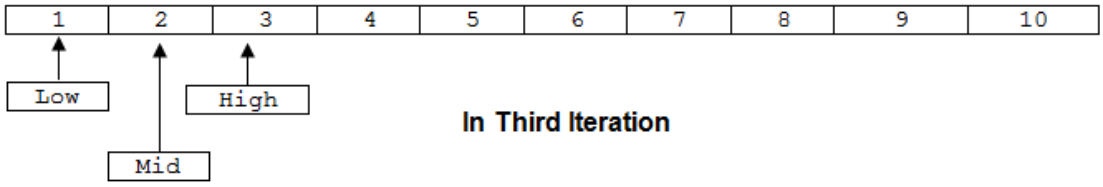
```
low = 0;
high = n-1; so high = 3-1 + 1 = 4
mid = 0 + 4 / 2
mid = 2
```



In Second Iteration

Index Number 3 पर स्थित मान Key x के मान के बराबर नहीं है, इसलिए x के मान को वापस Check किया जाता है कि x का मान mid के मान से कम है या अधिक। चूंकि x का मान 1 है और mid मान 3 है इसलिए वापस x का मान Array के Lower Part में होगा। वापस low के मान में कोई परिवर्तन नहीं होगा लेकिन high का मान mid+1 हो जाएगा। यानी

```
low = 0;
high = n-1; ( mid + 1 )so high = 2-1 + 1 = 2
mid = 0 + 2 / 2
mid = 1
```



In Third Iteration

अगले Iteration में x का मान Array के प्रथम Element पर प्राप्त हो जाएगा। Binary Search का Algorithm हम निम्नानुसार लिख सकते हैं–

Algorithm:

1 Let LOWEST = 0 ; HIGHEST = N; FOUND = 0; and flag = false;
2 WHILE [(LOWEST = HIGHEST) .AND. (flage = false)]
  Perform steps 3 to 5
3 Mid = Lowest + Highest / 2
4 IF X[MID] = ('A') then [FOUND = MID; flage = true];
5 IF X[mid] < 'A' then LOWEST = MID – 1;
  ELSE HIGHEST = MID – 1;
6 IF (flage = true) then print FOUND

ELSE 'unsuccessful search';

7 END

Program

```
#include<stdio.h>
#include<conio.h>
main()
{
int j, n, x, found, count, a[10], low, high, mid;
clrscr();
printf("Number of Elements in the Array ( <=10): ");
scanf("%d",&n);
for(j=0; j<n; j++)
{
printf("\n Enter The Elements:");
scanf("%d", &a[j]);
}
for(j=0; j<n; j++)
    printf("\n The Entered Array is %d \t", a[j]);

printf("\n Enter The Element to be searched : ");
scanf("%d", &x);

low = 0;
high=n-1;
found = -1;

while((low <= high ) && ( found == -1 ))
{
mid=(low + high ) / 2;
if(a[mid] == x )
    found = mid;

else if(a[mid] < x )
low = mid + 1;
    else
high = mid -1;
}

printf("\n");
if(found > -1)
    printf("\n Position of the required element ( from 0 position ) is:
    %d", found);
else
```

How to Get Complete PDF EBook

आप **Online Order** करके **Online** या **Offline** Payment करते हुए इस Complete EBook को तुरन्त Download कर सकते हैं।

Order करने और पुस्तक को Online/Offline Payment करते हुए खरीदने की पूरी प्रक्रिया की विस्तृत जानकारी प्राप्त करने के लिए आप **BccFalna.com** के निम्न Menu Options को Check Visit कर सकते हैं।

How to Make Order

How to Order?

How to Buy Online

How to Pay Online using PayUMoney

How to Pay Online using Instamojo

How to Pay Online using CCAvenue

How to Buy Offline

How to Pay Offline

Bank A/c Details

जबिक हमारे Old Buyers के **Reviews** भी देख सकते हैं ताकि आप इस बात का निर्णय ले सकें कि हमारे Buyers हमारे PDF EBooks से कितने Satisfied हैं और यदि आप एक से अधिक EBooks खरीदते हैं, तो **Extra Discount** की Details भी Menubar से प्राप्त कर सकते हैं।