CHAPTER 1

FUNDAMENTALS OF LOGIC

Objectives

After completing this course, you will be able to:

- Learn and understand proposition
- Learn logical operations
- Understand rules of replacement.

1

Discrete mathematics

Discrete mathematics is the study of mathematical structures that are fundamentally discrete rather than continuous. In contrast to real numbers that have the property of varying "smoothly", the objects studied in discrete mathematics – such as integers, graphs, and statements in logic do not vary smoothly in this way, but have distinct, separated values.

Logic

Logic is the study of the principles of valid reasoning and inference, as well as of consistency, soundness, and completeness. For example, in most systems of logic (but not in intuitionistic logic) Peirce's law (((P→Q)→P)→P) is a theorem. For classical logic, it can be easily verified with a truth table. The study of mathematical proof is particularly important in logic, and has applications to automated theorem proving and formal verification of software.

Logical formulas

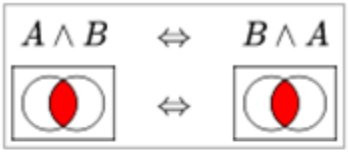
Logical formulas are discrete structures, as are proofs, which form finite treesor, more generally, directed acyclic graph structures (with each inference step combining one or more premise branches to give a single conclusion). The truth values of logical formulas usually form a finite set, generally restricted to two values: true and false, but logic can also be continuous-valued, e.g., fuzzy logic. Concepts such as infinite proof trees or infinite derivation trees have also been studied, e.g. infinitary logic.

Logical conjunction

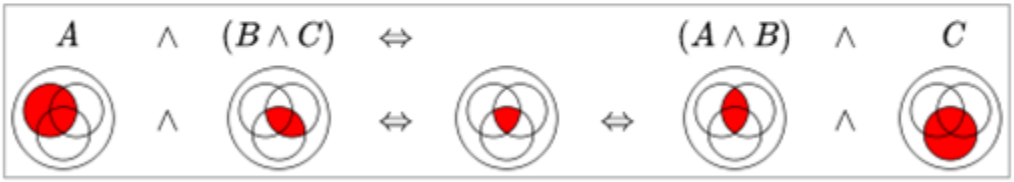
Logical conjunction is an operation on two logical values, typically the values of two propositions, that produces a value of *true* if and only if both of its operands are true.

The conjunctive identity is true, which is to say that AND-ing an expression with true will never change the value of the expression. In keeping with the concept of vacuous truth, when conjunction is defined as an operator or function of arbitrary arity, the empty conjunction (AND-ing over an empty set of operands) is often defined as having the result true.

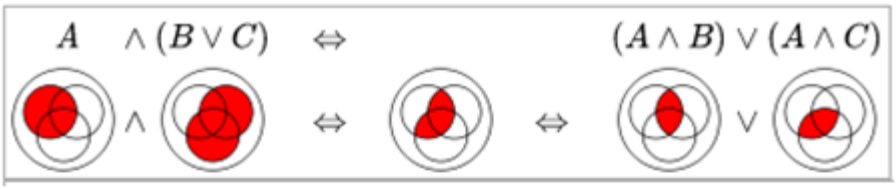
Properties



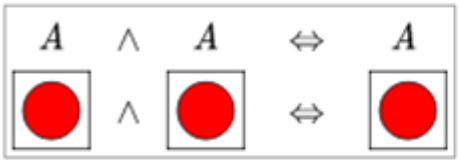
associativity: yes



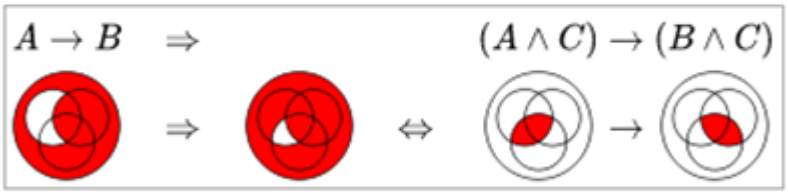
distributivity: with various operations, especially with *or*



idempotency: yes

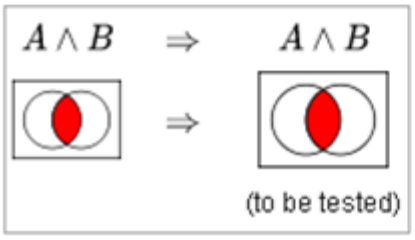


monotonicity: yes



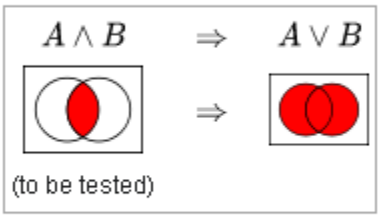
truth-preserving: yes

When all inputs are true, the output is true.



falsehood-preserving: yes

When all inputs are false, the output is false.



Walsh spectrum: (1,-1,-1,1)

Nonlinearity: 1 (the function is bent)

If using binary values for true (1) and false (0), then *logical conjunction* works exactly like normal arithmetic multiplication. A rule of replacement is a transformation rule that may be applied to only a particular segment of an expression. A logical system may be constructed so that it uses either axioms, rules of inference, or both as transformation rules for logical expressions in the system.

Video links:

Introduction to Sets - Discrete Mathematics

- *https://www.youtube.com/watch?v=tyDKR4FG3Yw&list=PLDDGPdw7e6Ag1EIznZ-m-qXu4XX3A0cIz*

Set Operations - Discrete Mathematics

- *https://www.youtube.com/watch?v=4TlCToZZ5gA&list=PLDDGPdw7e6Ag1EIznZ-m-qXu4XX3A0cIz&index=6*

[Discrete Mathematics] Set Operations Examples #2

- *https://www.youtube.com/watch?v=6RsudHXe6ZM&list=PLDDGPdw7e6Ag1EIznZ-m-qXu4XX3A0cIz&index=7*

[Discrete Mathematics] Truth Tables

- *https://www.youtube.com/watch?v=UiGu57JzLkE&list=PLDDGPdw7e6Ag1EIznZ-m-qXu4XX3A0cIz&index=12*

[Discrete Mathematics] Proofs with Truth Tables

- *https://www.youtube.com/watch?v=9fX6n0_MDic&list=PLDDGPdw7e6Ag1EIznZ-m-qXu4XX3A0cIz&index=13*

[Discrete Mathematics] Logic Laws

- *https://www.youtube.com/watch?v=eihhu72YdpQ*

Rules of Inference - Discrete Mathematics

- *https://www.youtube.com/watch?v=8DW0K3mnc-0*

Reference:

- Norman L. Biggs (2002-12-19). Discrete Mathematics. Oxford University Press
- John Dwyer (2010). An Introduction to Discrete Mathematics for Business & Computing
- Susanna S. Epp (2010-08-04). Discrete Mathematics With Applications.

CHAPTER 2

PROOF OF VALIDITY

Objectives

After completing this course, you will be able to:

- Learn and understand the proof of validity quantifiers
- Explain Quantification rules
- Understand Algebraic approaches to quantification

1

Validity

- Is the extent to which a concept, conclusion or measurement is well-founded and likely corresponds accurately to the real world.
- The word "valid" is derived from the Latin validus, meaning strong. The validity of a measurement tool (for example, a test in education) is the degree to which the tool measures what it claims to measure.
- Validity is based on the strength of a collection of different types of evidence (e.g. face validity, construct validity, etc.) described in greater detail below.

Mathematical proof

- Is an inferential argument for a mathematical statement, showing that the stated assumptions logically guarantee the conclusion.
- The argument may use other previously established statements, such as theorems; but every proof can, in principle, be constructed using only certain basic or original assumptions known as axioms, along with the accepted rules of inference.

Proofs

- Are examples of exhaustive deductive reasoning which establish logical certainty, to be distinguished from empirical arguments or non-exhaustive inductive reasoning which establish "reasonable expectation".
- Presenting many cases in which the statement holds is not enough for a proof, which must demonstrate that the statement is true in *all* possible cases.
- An unproven proposition that is believed to be true is known as a conjecture, or a hypothesis if frequently used as an assumption for further mathematical work.
- Employ logic expressed in mathematical symbols, along with natural language which usually admits some ambiguity
- Proofs are written in terms of rigorous informal logic.
- Purely formal proofs, written fully in symbolic language without the involvement of natural language, are considered in proof theory.
- The distinction between formal and informal proofs has led to much examination of current and historical mathematical practice, quasi-empiricism in mathematics, and so-called folk mathematics, oral traditions in the mainstream mathematical community or in other cultures.
- The word "proof" comes from the Latin *probare* (to test). Related modern words are English "probe", "probation", and "probability", Spanish *probar* (to smell or taste, or sometimes touch or test), Italian *provare* (to try), and German *probieren* (to try). The legal term "probity" means authority or credibility, the power of testimony to prove facts when given by persons of reputation or status.

Quantifiers

- A quantifier turns a sentence about something having some property into a sentence about the number (quantity) of things having the property.

2

Algebraic approaches to quantification

- It is possible to devise abstract algebras whose models include formal languages with quantification, but progress has been slow[clarification needed] and interest in such algebra has been limited.
- Three approaches have been devised to date:

Relation algebra, invented by Augustus De Morgan, and developed by Charles Sanders Peirce, Ernst Schröder, Alfred Tarski, and Tarski's students. Relation algebra cannot represent any formula with quantifiers nested more than three deep.

Surprisingly, the models of relation algebra include the axiomatic set theory ZFC and Peano arithmetic; Cylindric algebra, devised by Alfred Tarski, Leon Henkin, and others; The polyadic algebra of Paul Halmos.

Quantification Rules

- A universal quantification is a type of quantifier, a logical constant which is interpreted as "given any" or "for all".
- It expresses that a propositional function can be satisfied by every member of a domain of discourse. In other words, it is the predication of a property or relation to every member of the domain.
- It asserts that a predicate within the scope of a universal quantifier is true of every value of a predicate variable.
- It is usually denoted by the turned A (∀) logical operator symbol, which, when used together with a predicate variable, is called a universal quantifier ("∀x", "∀(x)", or sometimes by "(x)" alone).
- Universal quantification is distinct from *existential* quantification ("there exists"), which only asserts that the property or relation holds for at least one member of the domain.
- Quantification in general is covered in the article on quantification (logic)

Video links:

Four Basic Proof Techniques Used in Mathematics

- ***https://www.youtube.com/watch?v=V5tUc-J124s***

Universal and Existential Quantifiers, ∀ "For All" and ∃ "There Exists"

- ***https://www.youtube.com/watch?v=GJpezCUMOxA***

Negating Universal and Existential Quantifiers

- ***https://www.youtube.com/watch?v=q1rKFGSiZE8***

Reference:

- **Cupillari, Antonella (2005) [2001]. The Nuts and Bolts of Proofs: An Introduction to Mathematical Proofs (Third ed.)**
- **Peirce, C. S., 1885, "On the Algebra of Logic: A Contribution to the Philosophy of Notation, American Journal of Mathematics, Vol. 7, pp. 180–202. Reprinted in Kloesel, N. et al., eds., 1993. Writings of C. S. Peirce, Vol. 5.**
- **Miller, Jeff. "Earliest Uses of Symbols of Set Theory and Logic"**
- **Hinman, P. (2005). Fundamentals of Mathematical Logic.**

CHAPTER 3
SETS

Objectives

After completing this course, you will be able to:

- Definition of sets.
- Discuss the basic concept of set.
- Explained the set operation and algebra sets.

1

Set

A set is an unordered collection of different elements. A set can be written explicitly by listing its elements using set bracket. If the order of the elements is changed or any element of a set is repeated, it does not make any changes in the set.

Some Example of Sets

- A set of all positive integers
- A set of all the planets in the solar system
- A set of all the states in India
- A set of all the lowercase letters of the alphabet

Representation of a Set

Sets can be represented in two ways −

1. Roster or Tabular Form
2. Set Builder Notation

Roster or Tabular Form

The set is represented by listing all the elements comprising it. The elements are enclosed within braces and separated by commas.

- **Example 1** − Set of vowels in English alphabet, A={a,e,i,o,u}A={a,e,i,o,u}
- **Example 2** − Set of odd numbers less than 10, B={1,3,5,7,9}B={1,3,5,7,9}

Set Builder Notation

The set is defined by specifying a property that elements of the set have in common. The set is described as A={x:p(x)}A={x:p(x)}

- **Example 1** − The set {a,e,i,o,u}{a,e,i,o,u} is written as −
  A={x:x is a vowel in English alphabet}A={x:x is a vowel in English alphabet}
- **Example 2** − The set {1,3,5,7,9}{1,3,5,7,9} is written as −
  B={x:1≤x<10 and (x%2)≠0}B={x:1≤x<10 and (x%2)≠0}

If an element x is a member of any set S, it is denoted by x∈Sx∈S and if an element y is not a member of set S, it is denoted by y∉Sy∉S.

- **Example** − If S={1,1.2,1.7,2},1∈SS={1,1.2,1.7,2},1∈S but 1.5∉S1.5∉S

Some Important Sets

N − the set of all natural numbers = {1,2,3,4,.....}{1,2,3,4,.....}
Z − the set of all integers = {.....,−3,−2,−1,0,1,2,3,.....}{.....,−3,−2,−1,0,1,2,3,.....}
Z^+ − the set of all positive integers
Q − the set of all rational numbers
R − the set of all real numbers
W − the set of all whole numbers

Cardinality of a Set

Cardinality of a set S, denoted by |S||S|, is the number of elements of the set. The number is also referred as the cardinal number. If a set has an infinite number of elements, its cardinality is ∞∞.

- **Example** − |{1,4,3,5}|=4,|{1,2,3,4,5,…}|=∞|{1,4,3,5}|=4,|{1,2,3,4,5,…}|=∞
  If there are two sets X and Y,
  1. |X|=|Y||X|=|Y| denotes two sets X and Y having same cardinality. It occurs when the number of elements in X is exactly equal to the number of elements in Y. In this case, there exists a bijective function 'f' from X to Y.
  2. |X|≤|Y||X|≤|Y| denotes that set X's cardinality is less than or equal to set Y's cardinality. It occurs when number of elements in X is less than or equal to that of Y. Here, there exists an injective function 'f' from X to Y.
  3. |X|<|Y||X|<|Y| denotes that set X's cardinality is less than set Y's cardinality. It occurs when number of elements in X is less than that of Y. Here, the function 'f' from X to Y is injective function but not bijective.
  4. If |X|≤|Y|If |X|≤|Y| and |X|≥|Y||X|≥|Y| then |X|=|Y||X|=|Y|. The sets X and Y are commonly referred as equivalent sets.

Types of Sets

Sets can be classified into many types. Some of which are finite, infinite, subset, universal, proper, singleton set, etc.

1. **Finite Set**

   A set which contains a definite number of elements is called a finite set.

   Example − S={x|x∈NS={x|x∈N and 70>x>50}70>x>50}
2. **Infinite Set**

   A set which contains infinite number of elements is called an infinite set.

   Example − S={x|x∈NS={x|x∈N and x>10}x>10}
3. **Subset**

   A set X is a subset of set Y (Written as X⊆YX⊆Y) if every element of X is an element of set Y.

   Example1 −Let, X={1,2,3,4,5,6}X={1,2,3,4,5,6} and Y={1,2}Y={1,2} . Here set Y is a subset of set X as all the elements of set Y is in set X. Hence, we can write Y⊆XY⊆X.

   Example 2 − Let, X={1,2,3}X={1,2,3} and Y={1,2,3}Y={1,2,3}. Here set Y is a subset (Not a proper subset) of set X as all the elements of set Y is in set X. Hence, we can write Y⊆XY⊆X.
4. **Proper Subset**

   The term "proper subset" can be defined as "subset of but not equal to". A Set X is a proper subset of set Y (Written as X⊂YX⊂Y) if every element of X is an element of set Y and |X|<|Y||X|<|Y|.

   Example − Let, X={1,2,3,4,5,6}X={1,2,3,4,5,6} and Y={1,2}Y={1,2}. Here set Y⊂XY⊂X since all elements in YY are contained in XX too and XX has at least one element is more than set YY.
5. **Universal Set**

   It is a collection of all elements in a particular context or application. All the sets in that context or application are essentially subsets of this universal set. Universal sets are represented as UU.

3

Example − We may define UU as the set of all animals on earth. In this case, set of all mammals is a subset of UU, set of all fishes is a subset of UU, set of all insects is a subset of UU, and so on.

6. **Empty Set or Null Set**

   An empty set contains no elements. It is denoted by ∅∅. As the number of elements in an empty set is finite, empty set is a finite set. The cardinality of empty set or null set is zero.

   Example − S={x|x∈NS={x|x∈N and 7<x<8}=∅7<x<8}=∅

7. **Singleton Set or Unit Set**

   Singleton set or unit set contains only one element. A singleton set is denoted by {s}{s}.

   Example − S={x|x∈N, 7<x<9}S={x|x∈N, 7<x<9} = {8}{8}

8. **Equal Set**

   If two sets contain the same elements they are said to be equal.

   Example − If A={1,2,6}A={1,2,6} and B={6,1,2}B={6,1,2}, they are equal as every element of set A is an element of set B and every element of set B is an element of set A.

9. **Equivalent Set**

   If the cardinalities of two sets are same, they are called equivalent sets.

   Example − If A={1,2,6}A={1,2,6} and B={16,17,22}B={16,17,22}, they are equivalent as cardinality of A is equal to the cardinality of B. i.e. |A|=|B|=3|A|=|B|=3

10. **Overlapping Set**

    Two sets that have at least one common element are called overlapping sets.In case of overlapping sets −

    A. n(A∪B)=n(A)+n(B)−n(A∩B)n(A∪B)=n(A)+n(B)−n(A∩B)
    B. n(A∪B)=n(A−B)+n(B−A)+n(A∩B)n(A∪B)=n(A−B)+n(B−A)+n(A∩B)
    C. n(A)=n(A−B)+n(A∩B)n(A)=n(A−B)+n(A∩B)
    D. n(B)=n(B−A)+n(A∩B)n(B)=n(B−A)+n(A∩B)

    Example − Let, A={1,2,6}A={1,2,6} and B={6,12,42}B={6,12,42}. There is a common element ‘6’, hence these sets are overlapping sets.

11. **Disjoint Set**

    Two sets A and B are called disjoint sets if they do not have even one element in common. Therefore, disjoint sets have the following properties −

    A. n(A∩B)=∅n(A∩B)=∅
    B. n(A∪B)=n(A)+n(B)n(A∪B)=n(A)+n(B)

    Example − Let, A={1,2,6}A={1,2,6} and B={7,9,14}B={7,9,14}, there is not a single common element, hence these sets are overlapping sets.

4

Video links:

Introduction of Set

- *https://youtu.be/ZMhoRLGNR5Y*

Set Operation

- *https://youtu.be/4TlCToZZ5gA*

Reference:

- *https://www.tutorialspoint.com/discrete_mathematics/discrete_mathematics_sets.htm*
- *https://en.m.wikipedia.org/wiki/Set_(mathematics)*

CHAPTER 4

FUNDAMENTAL PRINCIPLES OF COUNTING

Objectives

After completing this course, you will be able to:

- Explained the Principles of Counting.
- Discuss the Permutation and Combination.

1

Principle of Counting

If we are dealing with the occurrence of more than one event or activity, sometimes it is important to be able to determine how many possible outcomes exist. The counting principle helps us with that:

If there are m ways for one activity to occur, and n ways for a second activity to occur, then there are m*n ways for both to occur.

Permutations

A permutation is an arrangement of some elements in which order matters. In other words a Permutation is an ordered Combination of elements.

Examples

- From a set S ={x, y, z} by taking two at a time, all permutations are − xy,yx,xz,zx,yz,zyxy,yx,xz,zx,yz,zy.
- We have to form a permutation of three digit numbers from a set of numbers S={1,2,3}S={1,2,3}. Different three digit numbers will be formed when we arrange the digits. The permutation will be = 123, 132, 213, 231, 312, 321

Number of Permutations

The number of permutations of 'n' different things taken 'r' at a time is denoted by nPr.

$$n_{P_r} = \frac{n!}{(n-r)!}$$

where n!=1.2.3.…(n−1).nn!=1.2.3.…(n−1).n

Proof − Let there be 'n' different elements.

There are n number of ways to fill up the first place. After filling the first place (n-1) number of elements is left. Hence, there are (n-1) ways to fill up the second place. After filling the first and second place, (n-2) number of elements is left. Hence, there are (n-2) ways to fill up the third place. We can now generalize the number of ways to fill up r-th place as [n – (r–1)] = n–r+1

So, the total no. of ways to fill up from first place up to r-th-place −

nPr=n(n−1)(n−2).....(n−r+1)nPr=n(n−1)(n−2).....(n−r+1)

=[n(n−1)(n−2)...(n−r+1)][(n−r)(n−r−1)…3.2.1]/[(n−r)(n−r−1)…3.2.1]=[n(n−1)(n−2)...(n−r+1)][(n−r)(n−r−1)…3.2.1]/[(n−r)(n−r−1)…3.2.1]

Hence,

nPr=n!/(n−r)!

Some important formulas of permutation

- If there are *n* elements of which a1a1 are alike of some kind, a2a2 are alike of another kind; a3a3 are alike of third kind and so on and arar are of rthrth kind, where (a1+a2+...ar)=n(a1+a2+...ar)=n.

  Then, number of permutations of these n objects is = n!/[(a1!(a2!)…(ar!)]n!/[(a1!(a2!)…(ar!)].

- Number of permutations of n distinct elements taking n elements at a time = nPn=n!nPn=n!
- The number of permutations of n dissimilar elements taking r elements at a time, when x particular things always occupy definite places = n−xpr−xn−xpr−x
- The number of permutations of n dissimilar elements when r specified things always come together is − r!(n−r+1)!r!(n−r+1)!
- The number of permutations of n dissimilar elements when r specified things never come together is − n!–[r!(n−r+1)!]n!–[r!(n−r+1)!]
- The number of circular permutations of n different elements taken x elements at time = npx/xnpx/x
- The number of circular permutations of n different things = npn/nnpn/n

Some Problems

1. **Problem 1** − From a bunch of 6 different cards, how many ways we can permute it?
   Solution − As we are taking 6 cards at a time from a deck of 6 cards, the permutation will be ${}^{6}P_{6}=6!=720{}^{6}P_{6}=6!=720$
2. **Problem 2** − In how many ways can the letters of the word 'READER' be arranged?
   Solution − There are 6 letters word (2 E, 1 A, 1D and 2R.) in the word 'READER'.Thepermutationwillbe =6!/[(2!)(1!)(1!)(2!)]=180.=6!/[(2!)(1!)(1!)(2!)]=180.
3. **Problem 3** − In how ways can the letters of the word 'ORANGE' be arranged so that the consonants occupy only the even positions?
   Solution − There are 3 vowels and 3 consonants in the word 'ORANGE'. Number of ways of arranging the consonants among themselves $={}^{3}P_{3}=3!=6={}^{3}P_{3}=3!=6$. The remaining 3 vacant places will be filled up by 3 vowels in ${}^{3}P_{3}=3!={}^{63}P_{3}=3!=6$ ways. Hence, the total number of permutation is 6×6=36

Combinations

A combination is selection of some given elements in which order does not matter.

The number of all combinations of n things, taken r at a time is −

nCr=n!/r!(n−r)!

1. **Problem 1**
   Find the number of subsets of the set {1,2,3,4,5,6}{1,2,3,4,5,6} having 3 elements.
   Solution:
   The cardinality of the set is 6 and we have to choose 3 elements from the set. Here, the ordering does not matter. Hence, the number of subsets will be ${}^{6}C_{3}={}^{206}C_{3}=20$.
2. **Problem 2**
   There are 6 men and 5 women in a room. In how many ways we can choose 3 men and 2 women from the room?
   Solution:

The number of ways to choose 3 men from 6 men is ${}^6C_3$, ${}^6C_3$ and the number of ways to choose 2 women from 5 women is ${}^5C_2$. Hence, the total number of ways is ${}^6C_3 * {}^5C_2 = 20*10 = 200$

3. **Problem 3**

How many ways can you choose 3 distinct groups of 3 students from total 9 students?

Solution:

Let us number the groups as 1, 2 and 3. For choosing 3 students for 1st group, the number of ways − 9C39C3. The number of ways for choosing 3 students for 2nd group after choosing 1st group − 6C36C3. The number of ways for choosing 3 students for 3rd group after choosing 1st and 2nd group − 3C3. Hence, the total number of ways is ${}^9C_3 * {}^6C_3 * {}^3C_3 = 84*20*1$.

Video links:

The Counting Principle, Permutations and Combinations

- *https://youtu.be/s_LfN4ltCs4*

Fundamental Counting Principle

- *https://youtu.be/8WdSJhEIrQk*

Reference:

- *https://www.tutorialspoint.com/discrete_mathematics/discrete_mathematics_counting_theory.htm*
- *https://www.mathplanet.com/education/algebra-2/discrete-mathematics-and-probability/counting-principle*

CHAPTER 5

PRINCIPLES OF INCLUSION-EXCLUSION, PIGEONHOLE PRINCIPLE

Objectives

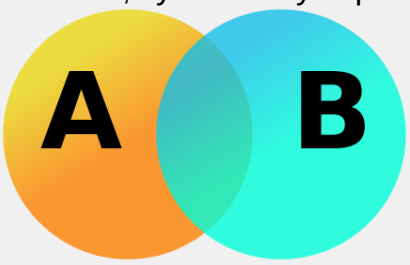
After completing this course, you will be able to:

- Learn the Inclusion-Exclusion principle including the statement
- Identify the use of Pigeonhole Principle and;
- Learn the etomology and the generalization of Pigeonhole principle.

1

Inclusion–exclusion principle

In combinatorics, a branch of mathematics, the inclusion–exclusion principle is a counting technique which generalizes the familiar method of obtaining the number of elements in the union of two finite sets; symbolically expressed as



Venn diagram showing the union of sets *A* and *B* as everything not in white

$$|A \cup B| = |A| + |B| - |A \cap B|,$$

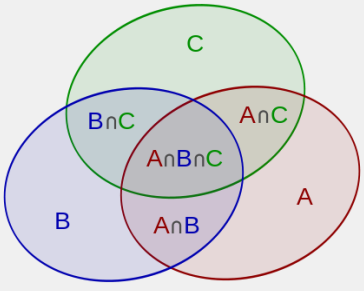
where A and B are two finite sets and |S| indicates the cardinality of a set S (which may be considered as the number of elements of the set, if the set is finite). The formula expresses the fact that the sum of the sizes of the two sets may be too large since some elements may be counted twice. The double-counted elements are those in the intersection of the two sets and the count is corrected by subtracting the size of the intersection.

The principle is more clearly seen in the case of three sets, which for the sets A, B and C is given by

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|.$$

This formula can be verified by counting how many times each region in the Venn diagram figure is included in the right-hand side of the formula. In this case, when removing the contributions of over-counted elements, the number of elements in the mutual intersection of the three sets has been subtracted too often, so must be added back in to get the correct total.

Inclusion–exclusion illustrated by a Venn diagram for three sets



Generalizing the results of these examples gives the principle of inclusion–exclusion. To find the cardinality of the union of n sets:

1. Include the cardinalities of the sets.
2. The cardinalities of the pairwise intersections.
3. The cardinalities of the triple-wise intersections.

4. The cardinalities of the quadruple-wise intersections.
5. The cardinalities of the quintuple-wise intersections.
6. Continue, until the cardinality of the n-tuple-wise intersection is included (if n is odd) or excluded (n even).

The name comes from the idea that the principle is based on over-generous inclusion, followed by compensating exclusion. This concept is attributed to Abraham de Moivre (1718); but it first appears in a paper of Daniel da Silva (1854), and later in a paper by J. J. Sylvester (1883). Sometimes the principle is referred to as the formula of Da Silva, or Sylvester due to these publications. The principle is an example of the sieve method extensively used in number theory and is sometimes referred to as the sieve formula, though Legendre already used a similar device in a sieve context in 1808.

As finite probabilities are computed as counts relative to the cardinality of the probability space, the formulas for the principle of inclusion–exclusion remain valid when the cardinalities of the sets are replaced by finite probabilities. More generally, both versions of the principle can be put under the common umbrella of measure theory.

In a very abstract setting, the principle of inclusion–exclusion can be expressed as the calculation of the inverse of a certain matrix. This inverse has a special structure, making the principle an extremely valuable technique in combinatorics and related areas of mathematics.

Statement

In its general form, the principle of inclusion–exclusion states that for finite sets A_1, ..., A_n, one has the identity

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{i=1}^{n} |A_i| - \sum_{1\leqslant i<j\leqslant n} |A_i \cap A_j| + \sum_{1\leqslant i<j<k\leqslant n} |A_i \cap A_j \cap A_k| - \cdots + (-1)^{n-1} |A_1 \cap \cdots \cap A_n|$$

This can be compactly written as

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{k=1}^{n} (-1)^{k+1} \left(\sum_{1\leqslant i_1<\cdots<i_k\leqslant n} |$$

or

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{\emptyset\neq J\subseteq\{1,\ldots,n\}} (-1)^{|J|+1} \left|\bigcap_{j\in J} A_j\right.$$

In words, to count the number of elements in a finite union of finite sets, first sum the cardinalities of the individual sets, then subtract the number of elements that appear in at least two sets, then add back the number of elements that appear in at least three sets, then subtract the number of elements that appear in at least four sets, and so on. This process always ends since there can be no elements that appear in more than the

number of sets in the union. (For example, if *n*=4, *n*=4, there can be no elements that appear in more than 4 sets; equivalently, there can be no elements that appear in at least 5 sets.)
In applications it is common to see the principle expressed in its complementary form. That is, letting S be a finite universal set containing all of the A_i and letting $\bar{A}_i$ denote the complement of A_i in S, by De Morgan's laws we have

$$\left|\bigcap_{i=1}^{n} \bar{A}_i\right| = \left|S - \bigcup_{i=1}^{n} A_i\right| = |S| - \sum_{i=1}^{n} |A_i| + \sum_{1 \leqslant i < j \leqslant n} |A_i \cap A_j| - \cdots + (-1)^n |A_1 \cap \cdots \cap A_n|$$

As another variant of the statement, let P_1, ..., P_n be a list of properties that elements of a set S may or may not have, then the principle of inclusion–exclusion provides a way to calculate the number of elements of S which have none of the properties. Just let Ai be the subset of elements of S which have the property Pi and use the principle in its complementary form. This variant is due to J. J. Sylvester.
Notice that if you take into account only the first m<n sums on the right (in the general form of the principle), then you will get an overestimate if m is odd and an underestimate if m is even.

Pigeonhole principle

In mathematics, the pigeonhole principle states that if *n* items are put into *m* containers, with *n*>*m*, then at least one container must contain more than one item. For example, if you have three gloves, then you must have at least two right-hand gloves, or at least two left-hand gloves, because you have three objects, but only two categories of handedness to put them into. This seemingly obvious statement, a type of counting argument, can be used to demonstrate possibly unexpected results. For example, if you know that the population of London is greater than the maximum number of hairs that can be present on a human's head, then the pigeonhole principle requires that there must be at least two people in London who have the same number of hairs on their heads.

Pigeons in holes. Here there are n = 10 pigeons in m = 9 holes. Since 10 is greater than 9, the pigeonhole principle says that at least one hole has more than one pigeon. (The top left hole has 2 pigeons.)

Although the pigeonhole principle appears as early as 1624 in a book attributed to Jean Leurechon, it is commonly called Dirichlet's box principle or Dirichlet's drawer

principle after an 1834 treatment of the principle by Peter Gustav Lejeune Dirichlet under the name Schubfachprinzip ("drawer principle" or "shelf principle").

The principle has several generalizations and can be stated in various ways. In a more quantified version: for natural numbers *k* and *m*, if *n*=*km*+1 objects are distributed among *m* sets, then the pigeonhole principle asserts that at least one of the sets will contain at least *k*+1 objects. For arbitrary *n* and *m* this generalizes to $k+1 = \lfloor (n-1)/m \rfloor + 1 = \lceil n/m \rceil$, where $\lfloor \cdots \rfloor$ and $\lceil \cdots \rceil$ denote the floor and ceiling functions, respectively.

Though the most straightforward application is to finite sets (such as pigeons and boxes), it is also used with infinite sets that cannot be put into one-to-one correspondence. To do so requires the formal statement of the pigeonhole principle, which is "there does not exist an injective function whose codomain is smaller than its domain". Advanced mathematical proofs like Siegel's lemma build upon this more general concept.

Etymology

Dirichlet published his works in both French and German, using either the German Schubfach or the French tiroir. The strict original meaning of these terms corresponds to the English drawer, that is, an open-topped box that can be slid in and out of the cabinet that contains it. (Dirichlet wrote about distributing pearls among drawers.) These terms were morphed to the word pigeonhole in the sense of a small open space in a desk, cabinet, or wall for keeping letters or papers, metaphorically rooted in structures that house pigeons.

Because furniture with pigeonholes is commonly used for storing or sorting things into many categories (such as letters in a post office or room keys in a hotel), the translation pigeonhole may be a better rendering of Dirichlet's original drawer metaphor. That understanding of the term pigeonhole, referring to some furniture features, is fading—especially among those who do not speak English natively but as a lingua franca in the scientific world—in favour of the more pictorial interpretation, literally involving pigeons and holes. The suggestive (though not misleading) interpretation of "pigeonhole" as "dovecote" has lately found its way back to a German back-translation of the "pigeonhole principle" as the "Taubenschlagprinzip".

Besides the original terms "Schubfachprinzip" in German[6] and "Principe des tiroirs" in French, other literal translations are still in use in Bulgarian ("принцип на чекмеджетата"), Chinese ("抽屜原理"), Danish ("Skuffeprincippet"), Dutch ("ladenprincipe "), Hungarian ("skatulyaelv"), Italian ("principio dei cassetti"), Japanese ("引き出し論法"), Persian ("اصل لانه کبوتری"), Polish ("zasada szufladkowa"), Swedish ("Lådprincipen"), and Turkish ("çekmece ilkesi").

Generalizations of the pigeonhole principle

A probabilistic generalization of the pigeonhole principle states that if n pigeons are randomly put into m pigeonholes with uniform probability 1/m, then at least one pigeonhole will hold more than one pigeon with probability

$$1 - \frac{(m)_n}{m^n},$$

5

where $(m)_n$ is the falling factorial $m(m-1)(m-2)...(m-n+1)$. For $n = 0$ and for $n = 1$ (and $m > 0$), that probability is zero; in other words, if there is just one pigeon, there cannot be a conflict. For $n > m$ (more pigeons than pigeonholes) it is one, in which case it coincides with the ordinary pigeonhole principle. But even if the number of pigeons does not exceed the number of pigeonholes ($n \leq m$), due to the random nature of the assignment of pigeons to pigeonholes there is often a substantial chance that clashes will occur. For example, if 2 pigeons are randomly assigned to 4 pigeonholes, there is a 25% chance that at least one pigeonhole will hold more than one pigeon; for 5 pigeons and 10 holes, that probability is 69.76%; and for 10 pigeons and 20 holes it is about 93.45%. If the number of holes stays fixed, there is always a greater probability of a pair when you add more pigeons. This problem is treated at much greater length in the birthday paradox.

A further probabilistic generalization is that when a real-valued random variable X has a finite mean E(X), then the probability is nonzero that X is greater than or equal to E(X), and similarly the probability is nonzero that X is less than or equal to E(X). To see that this implies the standard pigeonhole principle, take any fixed arrangement of n pigeons into m holes and let X be the number of pigeons in a hole chosen uniformly at random. The mean of X is n/m, so if there are more pigeons than holes the mean is greater than one. Therefore, X is sometimes at least 2.

Video links:

Discrete Math Inclusion–exclusion Principle

- ***https://youtu.be/GS7dIWA6Hpo***

Pigeonhole Principle

- ***https://youtu.be/2-mxYrCNX60***

Reference:

- ***https://en.m.wikipedia.org/wiki/Inclusion%E2%80%93exclusion_principle***
- ***https://en.m.wikipedia.org/wiki/Pigeonhole_principle***

CHAPTER 6

GENERATING FUNCTIONS AND RECURRENCE RELATIONS

Objectives

After completing this course, you will be able to:

- Learn the Linear Homogenous Recurrence Relations
- Learn to solve homogenous recurrence relations with constant and not constant coefficient
- Identify Generating Function

1

Linear Homogenous Recurrence Relations

In mathematics, a recurrence relation is an equation that recursively defines a sequence or multidimensional array of values, once one or more initial terms are given; each further term of the sequence or array is defined as a function of the preceding terms.

The term difference equation sometimes (and for the purposes of this article) refers to a specific type of recurrence relation. However, "difference equation" is frequently used to refer to any recurrence relation.

A recurrence relation is an equation that expresses each element of a sequence as a function of the preceding ones. More precisely, in the case where only the immediately preceding element is involved, a recurrence relation has the form

$$u_n = \varphi(n, u_{n-1}) \quad \text{for} \quad n > 0,$$

where

$$\varphi : \mathbb{N} \times X \to X$$

is a function, where *X* is a set to which the elements of a sequence must belong. For any $u_0 \in X$, this defines a unique sequence with u_0 as its first element, called the *initial value*.

It is easy to modify the definition for getting sequences starting from the term of index 1 or higher.

This defines recurrence relation of first order. A recurrence relation of order k has the form

$$u_n = \varphi(n, u_{n-1}, u_{n-2}, \ldots, u_{n-k}) \quad \text{for} \quad n \geq k,$$

where $\varphi : \mathbb{N} \times X^k \to X$ is a function that involves *k* consecutive elements of the sequence. In this case, *k* initial values are needed for defining a sequence.

Factorial

The factorial is defined by the recurrence relation

$$n! = n(n-1)! \quad \text{for} \quad n > 0,$$

and the initial condition

$$0! = 1.$$

Logistic map

An example of a recurrence relation is the logistic map:

$$x_{n+1} = r x_n (1 - x_n),$$

with a given constant r; given the initial term x0 each subsequent term is determined by this relation. Solving a recurrence relation means obtaining a closed-form solution: a non-recursive function of n.

2

Fibonacci numbers

The recurrence of order two satisfied by the Fibonacci numbers is the archetype of a homogeneous linear recurrence relation with constant coefficients (see below). The Fibonacci sequence is defined using the recurrence

$$F_n = F_{n-1} + F_{n-2}$$

with initial conditions (seed values)

$$F_0 = 0$$
$$F_1 = 1.$$

Explicitly, the recurrence yields the equations

$$F_2 = F_1 + F_0$$
$$F_3 = F_2 + F_1$$
$$F_4 = F_3 + F_2$$

etc.

We obtain the sequence of Fibonacci numbers, which begins

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

The recurrence can be solved by methods described below yielding Binet's formula, which involves powers of the two roots of the characteristic polynomial t2 = t + 1; the generating function of the sequence is the rational function

$$\frac{t}{1-t-t^2}.$$

Solving homogeneous linear recurrence relations with constant coefficients

Roots of the characteristic polynomial

An order-d homogeneous linear recurrence with constant coefficients is an equation of the form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_d a_{n-d},$$

where the *d* coefficients c_i (for all *i*) are constants, and $c_d \neq 0.$

A constant-recursive sequence is a sequence satisfying a recurrence of this form. There are d degrees of freedom for solutions to this recurrence, i.e., the initial values $a_0, \ldots, a_{d-1}$ can be taken to be any values but then the recurrence determines the sequence uniquely.

The same coefficients yield the characteristic polynomial (also "auxiliary polynomial")

$$p(t) = t^d - c_1 t^{d-1} - c_2 t^{d-2} - \cdots - c_d$$

whose *d* roots play a crucial role in finding and understanding the sequences satisfying the recurrence. If the roots r_1, r_2, ... are all distinct, then each solution to the recurrence takes the form

$$a_n = k_1 r_1^n + k_2 r_2^n + \cdots + k_d r_d^n,$$

where the coefficients k_i are determined in order to fit the initial conditions of the recurrence. When the same roots occur multiple times, the terms in this formula corresponding to the second and later occurrences of the same root are multiplied by

increasing powers of n. For instance, if the characteristic polynomial can be factored as $(x-r)^3$, with the same root r occurring three times, then the solution would take the form

$$a_n = k_1 r^n + k_2 n r^n + k_3 n^2 r^n.$$

Solving non-homogeneous linear recurrence relations with constant coefficients

If the recurrence is non-homogeneous, a particular solution can be found by the method of undetermined coefficients and the solution is the sum of the solution of the homogeneous and the particular solutions. Another method to solve a non-homogeneous recurrence is the method of symbolic differentiation. For example, consider the following recurrence:

$$a_{n+1} = a_n + 1$$

This is a non-homogeneous recurrence. If we substitute $n \mapsto n+1$, we obtain the recurrence

$$a_{n+2} = a_{n+1} + 1$$

Subtracting the original recurrence from this equation yields

$$a_{n+2} - a_{n+1} = a_{n+1} - a_n$$

or equivalently

$$a_{n+2} = 2a_{n+1} - a_n$$

This is a homogeneous recurrence, which can be solved by the methods explained above. In general, if a linear recurrence has the form

$$a_{n+k} = \lambda_{k-1} a_{n+k-1} + \lambda_{k-2} a_{n+k-2} + \cdots + \lambda_1 a_{n+1} + \lambda_0 a_n + p(n)$$

where, $\lambda_0, \lambda_1, \ldots, \lambda_{k-1}$ are constant coefficients and p(n) is the inhomogeneity, then if p(n) is a polynomial with degree r, then this non-homogeneous recurrence can be reduced to a homogeneous recurrence by applying the method of symbolic differencing r times.

Generating function

In mathematics, a generating function is a way of encoding an infinite sequence of numbers (an) by treating them as the coefficients of a formal power series. This series is called the generating function of the sequence. Unlike an ordinary series, the formal power series is not required to converge: in fact, the generating function is not actually regarded as a function, and the "variable" remains an indeterminate. Generating functions were first introduced by Abraham de Moivre in 1730, in order to solve the general linear recurrence problem. One can generalize to formal power series in more than one indeterminate, to encode information about infinite multi-dimensional arrays of numbers.

There are various types of generating functions, including ordinary generating functions, exponential generating functions, Lambert series, Bell series, and Dirichlet series; definitions and examples are given below. Every sequence in principle has a generating function of each type (except that Lambert and Dirichlet series require indices to start at 1 rather than 0), but the ease with which they can be handled may differ considerably. The particular generating function, if any, that is most useful in a

given context will depend upon the nature of the sequence and the details of the problem being addressed.

Generating functions are often expressed in closed form (rather than as a series), by some expression involving operations defined for formal series. These expressions in terms of the indeterminate x may involve arithmetic operations, differentiation with respect to x and composition with (i.e., substitution into) other generating functions; since these operations are also defined for functions, the result looks like a function of x. Indeed, the closed form expression can often be interpreted as a function that can be evaluated at (sufficiently small) concrete values of x, and which has the formal series as its series expansion; this explains the designation "generating functions". However such interpretation is not required to be possible, because formal series are not required to give a convergent series when a nonzero numeric value is substituted for x. Also, not all expressions that are meaningful as functions of x are meaningful as expressions designating formal series; for example, negative and fractional powers of x are examples of functions that do not have a corresponding formal power series.

Generating functions are not functions in the formal sense of a mapping from a domain to a codomain. Generating functions are sometimes called generating series, in that a series of terms can be said to be the generator of its sequence of term coefficients.

A generating function is a device somewhat similar to a bag. Instead of carrying many little objects detachedly, which could be embarrassing, we put them all in a bag, and then we have only one object to carry, the bag.

Ordinary generating function (OGF)

The ordinary generating function of a sequence a_n is

$$G(a_n; x) = \sum_{n=0}^{\infty} a_n x^n.$$

When the term generating function is used without qualification, it is usually taken to mean an ordinary generating function.

If a_n is the probability mass function of a discrete random variable, then its ordinary generating function is called a probability-generating function.

The ordinary generating function can be generalized to arrays with multiple indices. For example, the ordinary generating function of a two-dimensional array $a_{m,\ n}$ (where n and m are natural numbers) is

$$G(a_{m,n}; x, y) = \sum_{m,n=0}^{\infty} a_{m,n} x^m y^n.$$

Exponential generating function (EGF)

The exponential generating function of a sequence a_n is

$$\mathrm{EG}(a_n; x) = \sum_{n=0}^{\infty} a_n \frac{x^n}{n!}.$$

5

Exponential generating functions are generally more convenient than ordinary generating functions for combinatorial enumeration problems that involve labelled objects.

Poisson generating function

The Poisson generating function of a sequence an is

$$\mathrm{PG}(a_n; x) = \sum_{n=0}^{\infty} a_n e^{-x} \frac{x^n}{n!} = e^{-x}\ \mathrm{EG}(a_n; x).$$

Lambert series

The Lambert series of a sequence a_n is

$$\mathrm{LG}(a_n; x) = \sum_{n=1}^{\infty} a_n \frac{x^n}{1 - x^n}.$$

The Lambert series coefficients in the power series expansions $b_n := [x^n]\,\mathrm{LG}(a_n; x)$ for integers $n \geq 1$ are related by the divisor sum $b_n = \sum_{d|n} a_d.$. The main article provides several more classical, or at least well-known examples related to special arithmetic functions in number theory. In a Lambert series the index *n* starts at 1, not at 0, as the first term would otherwise be undefined.

Bell series

The Bell series of a sequence an is an expression in terms of both an indeterminate x and a prime p and is given by

$$\mathrm{BG}_p(a_n; x) = \sum_{n=0}^{\infty} a_{p^n} x^n.$$

Video links:

Recurrence Relation - Discrete Mathematics

- *https://youtu.be/eAaP4XaB8hM*

Homogenous Recurrence Relations

- *https://youtu.be/7mhvA5L7KqY*

Generating Functions

- *https://youtu.be/-drdeNMoe8w*

Reference:

- *https://en.m.wikipedia.org/wiki/Recurrence_relation*
- *https://en.m.wikipedia.org/wiki/Generating_function*

6

CHAPTER 7

BASIC CONCEPT AND NOTATION OF ALGORITHM

Objectives

After completing this course, you will be able to:

- Give brief introduction into basic data structures and algorithms.
- Introduce fundamental notation and algorithmic concepts.
- Learn algorithms exist for many common problems, and designing efficient algorithms plays a crucial role in developing large-scale computer systems.

1

What is Algorithm?

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages. The algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms:

- **Search** - Algorithm to search an item in a data structure.
- **Sort** - Algorithm to sort items in a certain order.
- **Insert** - Algorithm to insert item in a data structure.
- **Update** - Algorithm to update an existing item in a data structure.
- **Delete** - Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics:

- **Unambiguous** - Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** - An algorithm should have 0 or more well-defined inputs.
- **Output** - An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** - Algorithms must terminate after a finite number of steps.
- **Feasibility** - Should be feasible with the available resources.
- **Independent** - An algorithm should have step-by-step directions, which should be independent of any programming code.
- **Definiteness** - Each instruction is clear and unambiguous.
- **Effectiveness** - Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible

In computational theory, one distinguishes between an algorithm and a program, the latter of which does not have to satisfy the fourth condition. For example, we can think of an operating system that continues in a wait loop until more jobs are entered. Such a program does not terminate unless the system crashes. Since our programs will always terminate, we will use algorithm and program interchangeably in this text.

We can describe an algorithm in many ways. We can use a natural language like English, although, if we select this option, we must make sure that the resulting instructions are definite. Graphic representations called flowcharts are another possibility, but they work well only if the algorithm is small and simple.

Example 1.1: [Selection sort]

Suppose we must devise a program that sorts a set of $n \geq 1$ integers. A simple solution is given by the following:

From those integers that are currently unsorted, find the smallest and place it next in the sorted list.

Although this statement adequately describes the sorting problem, it is not an algorithm since it leaves several unanswered questions. For example, it does not tell us where and how the integers are initially stored, or where we should place the result. Notice that it is written partially in C and partially in English.

```
for (i = 0; i < n; i++) {
  Examine list[i] to list[n-1] and suppose that the
  smallest integer is  at list[min];

  Interchange list[i] and list[min];
}
```

Program 1.2: Selection sort algorithm

To tum Program 1.2 into a real C program, two clearly defined subtasks remain: finding the smallest integer and interchanging it with *list [i].* We can solve the latter problem using either a function (Program 1.3) or a macro.

```
void swap(int *x, int *y)
{/* both parameters are pointers to ints */
   int temp = *x;    /* declares temp as an int and assigns
                        to it the contents of what x points to */
   *x = *y; /* stores what y points to into the location
               where x points */
   *y = temp; /* places the contents of temp in location
                 pointed to by y */
}
```

Program 1.3: Swap function

Using the function, suppose a and b are declared as ints. To swap their values one would say:

swap (&a, &b);

passing to swap the addresses of a and h. The macro version of swap is:

```
#define SWAP(x,y,t) ((t) = (x), (x) = (y), (y) = (t))
```

The function's code is easier to read than that of the macro but the macro works with any data type. We can solve the first subtask by assuming that the minimum is *list [i],* checking *list [i] with list [i + 1], list [i +2],…,list [n -1].* Whenever we find a smaller number we make it the new minimum. When we reach list [n -1] we are finished. Putting all these observations together gives us sort (Program 1.4). Program 1.4 contains a complete program which you may run on your computer. The program uses the *rand* function defined in *math.h* to randomly generate a list of numbers which are then passed into *sort*. At this point, we should ask if this function works correctly.

Binary search - Assume that we have $n \geq 1$ distinct integers that are already sorted and stored in the array *list*. That is, $list\,[0] \leq list\,[1] \leq \ldots \leq list\,[n-l]$. We must figure out if an integer searchnum is in this list. If it is we should return an index, *i,* such that *list[i] = searchnum*. If *searchnum* is not present, we should return -1. Since the list is sorted we may use the following method to search for the value.

Let *left* and *right*, respectively, denote the left and right ends of the list to be searched. Initially, *left = 0 and right = n-1.* Let *middle = (left+right)/2* be the middle position in the list. If we compare *list [middle]* with *searchnum,* we obtain one of three results:

- **searchnum < list[middle]:** In this case, if *searchnum* is present, it must be in the positions between 0 and *middle* - 1. Therefore, we set *right to middle - 1*.

- **searchnum = list[middle]:** In this case, we return *middle.*
- **searchnum > list[middle]:** In this case, if *searchnum* is present, it must be in the positions between *middle + 1 and n - 1*. So, we set *left to middle + 1.*

If *searchnum* has not been found and there are still integers to check, we recalculate middle and continue the search. Program 1.5 implements this searching strategy. The algorithm contains two subtasks: (1) determining if there are any integers left to check, and (2) comparing *searchnum to list [middle].*

```
while (there are more integers to check ) {
   middle = (left + right) / 2;
   if (searchnum < list[middle])
      right = middle – 1;
   else if (searchnum == list[middle])
            return middle;
         else left = middle + 1;
}
```

Program 1.4: Selection sort

```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x,y,t) ((t) = (x), (x)= (y), (y) = (t))
void sort(int [],int); /*selection sort */
void main(void)
{
   int i,n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d",&n);
   if( n < 1 || n > MAX_SIZE) {
     fprintf(stderr, "Improper value of n\n");
     exit(EXIT_FAILURE);
   }
```

Program 1.5: Searching a sorted list

We can handle the comparisons through either a function or a macro. In either case, we must specify values to signify less than, equal, or greater than. We will use the strategy followed in C's library functions:

- We return a negative number (-1) if the first number is less than the second.
- We return a 0 if the two numbers are equal.
- We return a positive number (1) if the first number is greater than the second.

Although we present both a function (Program 1.6) and a macro, we will use the macro throughout the text since it works with any data type.

```
int compare(int x, int y)
{/* compare x and y, return –1 for less than, 0 for equal,
    1 for greater */
   if (x < y) return –1;
   else if (x == y) return 0;
        else return 1;
}
```

Program 1.6: Comparison of two integers

The macro version is:

```
#define COMPARE(x,y) (((x) < (y)) ? -1: ((x) == (y))? 0: 1)
```

We are now ready to tackle the first subtask: determining if there are any elements left to check. You will recall that our initial algorithm indicated that a comparison could cause us to move either our left or right index. Assuming we keep moving these indices, we will eventually find the element, or the indices will cross, that is, the left index will have a higher value than the right index. Since these indices delineate the search boundaries, once they cross, we have nothing left to check. Putting all this information together gives us *binsearch* (Program 1.7).

```
int binsearch(int list[], int searchnum, int left,
                                         int right)
{/* search list[0] <= list[1] <=  · · ·  <= list[n-1] for
 searchnum. Return its position if found. Otherwise
 return -1 */
  int  middle;
  while (left <= right)  {
     middle = (left + right)/2;
     switch (COMPARE(list[middle], searchnum)) {
        case -1: left = middle + 1;
                 break;
        case 0 : return middle;
        case 1 : right = middle - 1;
     }
  }
  return -1;
}
```

Program 1.7: Searching an ordered list

The search strategy just outlined is called binary search. The previous examples have shown that algorithms are implemented as functions in C. Indeed functions are the primary vehicle used to divide a large program into manageable pieces. They make the program easier to read, and, because the functions can be tested separately, increase the probability that it will run correctly. Often we will declare a function first and provide its definition later. In this way the compiler is made aware that a name refers to a legal function that will be defined later. In C, groups of functions can be compiled separately, thereby establishing libraries containing groups of logically related algorithms.

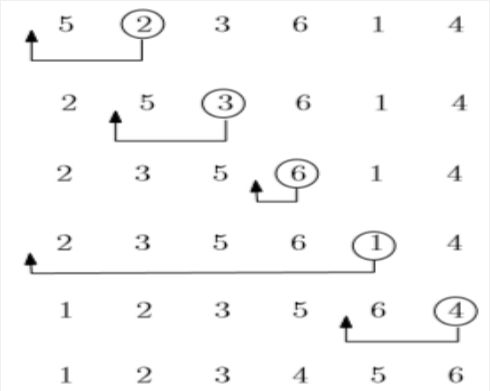
Sorting

Sorting is a fundamental task that needs to be performed as subroutine in many computer programs. Sorting also serves as an introductory problem that computer science students usually study in their first year. As *input*, we are given a sequence of n natural numbers $\langle a_1, a_2, \ldots, a_n \rangle$ that are not necessarily all pairwise different. As an output, we want to receive a permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the numbers such that $\langle a'_1, a'_2, \ldots, a'_n \rangle$. Some methods will be introduced in the following. In general, all input that is necessary for the method to determine a solution is called an *instance*.

Insertion Sort

Our first sorting algorithm is called insertion sort. To motivate the algorithm, let us describe how in a card player usually orders a deck of cards. Suppose the cards that

are already on the hand are sorted in increasing order from left to right when a new card is taken. In order to determine the “slot” where the new card has to be inserted, the player starts scanning the cards from right to left. As long as not all cards have yet been scanned and the value of the new card is strictly smaller than the currently scanned card, the new card has to be inserted into some slot further left. Therefore, the currently scanned card has to be shifted a bit, say one slot, to the right in order to reserve some space for the new card.
Example:



Sorting using the Divide-and-Conquer Principle

A general algorithmic principle that can be applied for sorting consists in *divide and conquer.* Loosely speaking, this approach consists in *dividing* the problem into sub-problems of the same kind, *conquering* the sub-problems by recursive solution or direct solution if they are small enough, and finally *combining* the solutions of the sub-problems to one for the original problem.

Merge Sort

The well-known merge sort algorithm specifies the divide and conquer principle as follows. When merge sort is called for array A that stores a sequence of n numbers, it is divided into two sequences of equal length. The same merge sort algorithm is then called recursively for these two shorter sequences. For arrays of length one, nothing has to be done. The sorted subsequences are then merged in a zip-fastener manner which results in a sorted sequence of length equal to the sum of the lengths of the subsequences.

```
1void merge_sort(int* A, int p, int r)
2{
3  int q;
4  if (p < r) {
5    q = p + ((r - p) / 2);
6    merge_sort(A, p, q);
7    merge_sort(A, q + 1,r);
8    merge(A, p, q, r);
9  }
10}
```

Quicksort

6 Quicksort is another divide-and-conquer sorting algorithm that is widely used in practice. For a sequence with length at most one, nothing is done. Otherwise, we take a

specific element a_i from the sequence, the so-called *pivot* element. Let us postpone for the moment a discussion how such a pivot should be chosen.

Binary Search

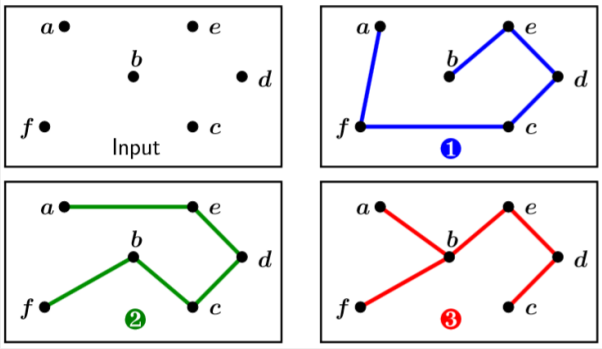
Suppose we have some sorted sequence at hand and want to know the position of an element with a certain key k. For example, let the keys $\langle 1,3,3,7,9,15,27 \rangle$. A straight forward linear-time search algorithm would scan each element in the sequence, compare it with the element we search for and stops when either the element we look for has been found or the whole sequence has been scanned without success. If we however know that the sequence is sorted, searching for a specific element can be done more efficiently with the divide and conquer principle.

Elementary Data Structures

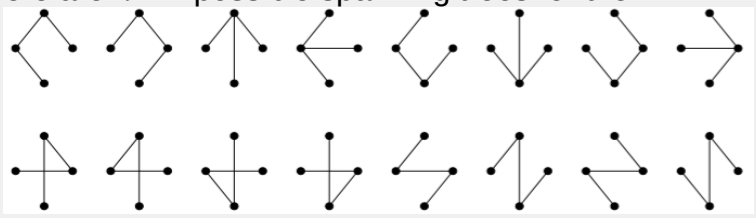
Up to now, we have mainly focused on sorting algorithms and their performance. In the given C implementations, the elements were simply stored in arrays of length *n*. whereas arrays are well suited for our purposes, it is sometimes necessary to use more involved data structures. As an easy example, suppose we have an application in which we do not know beforehand how many elements we need to store. Or suppose we want to remove an element somewhere in the middle.

Minimal spanning tree problem

Given a set of points, find a spanning tree with the shortest total length.



MST problem: given a set of points, find a spanning tree with the shortest total length.
Brute force method: enumerate all possible spanning trees and select the best one among them.
Given n points, there are n^{n-2} possible spanning trees for them.



Time complexity of algorithm A

Time complexity of an algorithm:

- Equal to number of operations in algorithm A.

- Usually represented by a function of the size of the input

Size of the input:

- **Sorting:** number of items
- **Graph problems**: number of vertices and edges
- **Multiplying two integers**: number of bits

  Example: For MST problem,

  Prim's algorithm = $|V|^2\ time$

  Kruskal's algorithm = $|E|\log|E| + |V|\ time$

Decision/Optimization problems

- **Decision problem**: the problem whose solution is simply "yes" or "no"

  Example: *Traveling salesperson decision problem*: Given a set of points and a constant c, is there a tour starting from any point v_0 whose total length is less than c?

- **Optimization problem:** the problem of finding a solution whose value is optimal

  Example: *Traveling salesperson problem*: Given a set of points, find a shortest tour which starts from any point v_0.

P and NP problems

- **P problem**: a decision problem which can be solved by a polynomial algorithm, such as the MST decision problem and the longest common subsequence decision problem.
- **NP problem**: a decision problem which can be solved by a nondeterministic polynomial algorithm, such as the SAT problem and the traveling salesperson decision problem.
- Every P problem must be an NP problem (i.e., P ⊆ NP).

Are all problems NP problems?

- **Halting problem**: given an arbitrary program with an arbitrary input data, will the program terminate or not? Halting problem is not an NP problem because it is un-decidable.

Video links:

Discrete Math: Algorithm and their properties

- *https://youtu.be/CN-UiM3of0E*

Algorithm: Binary Search

- *https://youtu.be/P3YID7liBug*

Algorithm: Merge Sort

- *https://youtu.be/KF2j-9iSf4Q*

Mathematical Notation

- *https://youtu.be/Y-c_CgxxPF0*

Reference:

- *https://www.csie.ntu.edu.tw/~hsinmu/courses/_media/dsa_13spring/horowitz_28_41.pdf*
- *https://www.tutorialspoint.com/data_structures_algorithms/data_structures_algorithms_tutorial.pdf*
- *http://erdos.csie.ncnu.edu.tw/~rctlee/course/biology/slide2_algorithm.pdf*

CHAPTER 8

THE EUCLIDEAN ALGORITHM

Objectives

After completing this course, you will be able to:

- Compute the greatest common divisor (GCD) of two integers.
- Find the set of positive divisors of each number
- Find the intersection of the two sets computed in the previous step.

1

Origins of the Analysis of the Euclidean Algorithm

The Euclidean algorithm for computing the greatest common divisor of two integers is, as D. E. Knuth has remarked, "the oldest non trivial algorithm that has survived to the present day." Credit for the first analysis of the running time of the algorithm is traditionally assigned to Gabriel Lame, for his 1844 paper. A weak bound on the running time of this algorithm was given as early as 1811 by Antoine Andre Louis Reynaud. Furthermore, Lame's basic result was known to t-mile Larger in 1837, and a complete, valid proof along different lines was given by Pierre Joseph Etienne Finck in 1841.

Gabriel Lame

Gabriel Lame was a famous French mathematician who was primarily interested in geometry, thermodynamics, applied mechanics, and number theory. Since the life and work of Lame has been covered at length elsewhere in easily available sources.

In his well-known 1844, Lame proved that if u > v > 0, then the number of division steps E(u, v) performed by the Euclidean algorithm is always less than 5 times the number of decimal digits in v.

The method of proof used by Lame was as follows:

i. He proved that, for all *k*, there are at most five Fibonacci numbers whose decimal expansions contain k digits. Then, given an arbitrary input u_0 and u_1 to the algorithm, he considered how the sequence $u_2, u_3, .., u_n$, of remainders determined by the algorithm are distributed among the intervals given by successive Fibonacci numbers, F_k, F_{k+1}. He showed:
   - (i) no more than two remainders can appear between successive Fibonacci numbers and
   - (ii) when two such remainders appear in an interval F_k, F_{k+1} the following interval [Fk-l, Fk] contains no remainder. The result now follows.

Previous Analysis of the Euclidean Algorithm

Although Lame is generally recognized as the first to analyze the Euclidean algorithm in his 1844 paper, in fact there was much previous work on this problem. For example, in 1733 Thomas Fantet de Lagny (1660-1734) described his *"thorie generale des rapports"*; this was essentially based on the terms of the simple continued fraction expansion of the quotient of two integers. He divided the rapports into different genres, depending on the length of the continued fraction expansion. He also introduced a method of calculation that is essentially what we call continuants today. As an example, de Lagny gave (what would now be called) the series of convergents to $(1 + \sqrt{5})/2$:

$$\frac{1}{1}, \frac{2}{1}, \frac{3}{2}, \frac{5}{3}, \frac{8}{5}, \frac{13}{8}, \frac{21}{13}, \frac{34}{21}, \frac{55}{34}, \frac{89}{55}, \frac{144}{89}, \cdots$$

The Euclidean Algorithm for finding GCD(A,B) is as follows:

- If $A = 0$ $then$ $GCD\ (A,B) = B, since\ the\ GCD(0,B) = B$, and we can stop.
- If $B = 0$ $then$ $GCD\ (A,B) = A, since\ the\ GCD(A,0) = A$, and we can stop.

Write A in quotient remainder form (A = B·Q + R)

Find $GCD\ (B,R)$ using the Euclidean Algorithm since $GCD(A,B) = GCD(B,R)$

Example: Find the $GCD\ of\ 270\ and\ 192$

$$A = 270, B = 192$$
$$A \neq 0$$
$$B \neq 0$$

Use long division to find that $270/192 = 1$ with a remainder of 78. We can write this as:

$$270 = 192 * 1 + 78$$

Find $GCD\ (192,78)$, since $GCD\ (270,192) = GCD(192,78)$

$$A = 192, B = 78$$
$$A \neq 0$$
$$B \neq 0$$

Use long division to find that $192/78 = 2$ with a remainder of 36. We can write this as:

$$192 = 78 * 2 + 36$$

Understanding the Euclidean Algorithm

If we examine the Euclidean Algorithm we can see that it makes use of the following properties:

$$GCD(A, 0) = A$$
$$GCD(0, B) = B$$

If A = $B \cdot Q + R\ and\ B \neq 0$ then $GCD\ (A, B) = GCD(B, R)$ where Q is an integer, R is an integer between $0\ and\ B - 1$.

The first two properties let us find the GCD if either number is 0. The third property lets us take a larger, more difficult to solve problem, and reduce it to a smaller, easier to solve problem.

The Euclidean Algorithm makes use of these properties by rapidly reducing the problem into easier and easier problems, using the third property, until it is easily solved by using one of the first two properties.

We can understand why these properties work by proving them. We can prove that $GCD\ (A, 0) = A$ is as follows:

- The largest integer that can evenly divide A is A.
- All integers evenly divide 0, since for any integer C, we can write $C \cdot 0 = 0$. So we can conclude that A must evenly divide 0.
- The greatest number that divides both A and 0 is A.
- The proof for $GCD(0, B) = B$ is similar. (Same proof, but we replace A with B).
- To prove that $GCD(A, B) = GCD(B, R)$ we first need to show that $GCD(A, B) = GCD(B, A - B)$.

Recursive Algorithm

Typically, beginning programmers view a function as something that is invoked called by another function. It executes its code and then returns control to the calling function. This perspective ignores the fact that functions can call themselves direct recursion or they may call other functions that invoke the calling function again indirect recursion. These recursive mechanisms are not only extremely powerful, but they also frequently allow us to express an otherwise complex process in very clear terms. It is for these reasons that we introduce recursion here. Frequently computer science students regard recursion as a mystical technique that is useful for only a few special problems such as computing factorials or *Ackermann's function*. This is unfortunate because any

function that we can write using assignment, if-else, and while statements can be written recursively. Often this recursive function is easier to understand than its iterative counterpart. How do we determine when we should express an algorithm recursively? One instance is when the problem itself is defined recursively. Factorials and Fibonacci numbers fit into this category as do binomial coefficients where:

$$\binom{n}{m} = \frac{n!}{m!\,(n-m)!}$$

can be recursively computed by the formula:

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$

We would like to use two examples to show you how to develop a recursive algorithm. In the first example, we take the binary search function that we created in example and transform it into a recursive function. In the second example, we recursively generate all possible permutations of a list of characters.

Binary search - To transform this function into a recursive one, we must:

- ✓ Establish boundary conditions that terminate the recursive calls, and
- ✓ Implement the recursive calls so that each call brings us one step closer to a solution.
- ✓ If we examine there are two ways to terminate the search:
  - Signaling a success (list [middle] = searchnum),
  - The other signaling a failure (the left and right indices cross). We do not need to change the code when the function terminates successfully.

However, while statement that is used to trigger the unsuccessful search needs to be replaced with an equivalent if statement whose then clause invokes the function recursively. Creating recursive calls that move us closer to a solution is also simple since it requires only passing the new left or right index as a parameter in the next recursive call. Program l.8 implements the recursive binary search. Notice that although the code has changed, the recursive function call is identical to that of the iterative function.

```
int binsearch(int list[], int searchnum, int left,
                                         int right)
{/* search list[0] <= list[1] <= ... <= list[n-1] for
    searchnum. Return its position if found. Otherwise
    return -1 */
   int middle;
   if (left <= right) {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: return
            binsearch(list, searchnum, middle + 1, right);
         case 0 : return middle;
         case 1 : return
            binsearch(list, searchnum, left, middle - 1);
      }
   }
   return -1;
}
```

Recursive implementation of binary search

Permutations - Given a set of $n \geq 1$ elements, print out all possible permutations of this set. For example, if the set is $\{a, b, c\}$ then the set of permutations is $\{(a, b, c), (a, c, b), (b, a, c), (b, c, a), (c, a, b), (c, b, a)\}$. It is easy to see that, given *n* elements, there are $n!$ permutations. We can obtain a simple algorithm for generating the permutations if we look at the set $\{a, b, c, d\}$. We can construct the set of permutations by printing:

- a followed by all permutations of (b, c, d)
- b followed by all permutations of (a, c, d)
- c followed by all permutations of (a, b, d)
- d followed by all permutations of (a, b, c)

The clue to the recursive solution is the phrase "followed by all permutations." It implies that we can solve the problem for a set with n elements if we have an algorithm that works on $n - 1$ elements. These considerations lead to the development of example below. We assume that list is a character array. Notice that it recursively generates permutations until $i = n$. The initial function call is *perm* (list, 0, n -1).

```
void perm(char *list, int i, int n)
{/* generate all the permutations of list[i] to list[n] */
   int j, temp;
   if (i == n) {
      for (j = 0; j <= n; j++)
         printf("%c", list[j]);
      printf("    ");
   }
   else {
   /* list[i] to list[n] has more than one permutation,
      generate these recursively */
      for (j = i; j <= n; j++) {
         SWAP(list[i],list[j],temp);
         perm(list,i+1,n);
         SWAP(list[i],list[j],temp);
      }
   }
}
```

Recursive permutation generator

Data Abstraction

The reader is no doubt familiar with the basic data types of C. These include *char, int, float, and double*. Some of these data types may be modified by the keywords *short, long, and unsigned*. Ultimately, the real world abstractions we wish to deal with must be represented in terms of these data types. In addition to these basic types, C helps us by providing two mechanisms for grouping data together. These are the array and the structure. Arrays are collections of elements of the same basic data type. They are declared implicitly, for example, “int list [5]” defines a five-element array of integers whose legitimate subscripts are in the range 0 ... 4. Structures are collections of elements whose data types need not be the same. They are explicitly defined. For example:

5

```
struct student {
       char lastName;
       int studentId;
       char grade;
       }
```

This defines a structure with three fields, two of type character and one of type integer. The structure name is student. All programming languages provide at least a minimal set of predefined data types, plus the ability to construct new, or user-defined types. It is appropriate to ask the question, "What is a data type?"

Data Type is a collection of objects and a set of operations that act on those objects. Whether your program is dealing with predefined data types or user-defined data types, these two aspects must be considered: *objects and operations.*

For example, the data type int consists of the objects $\{0, +1, -1, +2, -2, \ldots, INT - MAX, INT - MIN\}$, where $INT - MAX \; and \; INT - MIN$ are the largest and smallest integers that can be represented on your machine.

Abstract Data Type (ADT) is a data type that is organized in such a way that the specification of the objects and the specification of the operations on the objects are separated from the representation of the objects and the implementation of the operations.

Some programming languages provide explicit mechanisms to support the distinction between specification and implementation. For example, Ada has a concept called a package, and C++ has a concept called a class. Both of these assist the programmer in implementing abstract data types. Although C does not have an explicit mechanism for implementing ADTs, it is still possible and desirable to design your data types using the same notion.

The specification consists of the names of every function, the type of its arguments, and the type of its result. There should also be a description of what the function does, but without appealing to internal representation or implementation details. This requirement is quite important, and it implies that an abstract data type is implementation-independent. Furthermore, it is possible to classify the functions of a data type into several categories:

- **Creator/constructor**: These functions create a new instance of the designated type.
- **Transformers**: These functions also create an instance of the designated type, generally by using one or more other instances. The difference between constructors and transformers will become clearer with some examples.
- **Observers/reporters**: These functions provide information about an instance of the type, but they do not change the instance.

Abstract data type Natural Number as this is the first example of an ADT, we will spend some time explaining the notation. ADT 1.1 contains the ADT definition of Natural Number.

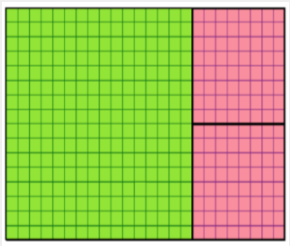
ADT Natural Number is **Objects:** an ordered sub-range of the integers starting at zero and ending at the maximum integer $(lNT - MAX)$ on the computer

functions:
for all $x, y \in$ *NaturalNumber*; *TRUE*, *FALSE* $\in$ *Boolean*
and where +, −, <, and == are the usual integer operations

NaturalNumber Zero( )	::=	0
Boolean IsZero(x)	::=	**if** (x) **return** *FALSE*
		else return *TRUE*
Boolean Equal(x, y)	::=	**if** (x == y) **return** *TRUE*
		else return *FALSE*
NaturalNumber Successor(x)	::=	**if** (x == *INT_MAX*) **return** x
		else return $x + 1$
NaturalNumber Add(x, y)	::=	**if** (($x + y$) <= *INT_MAX*) **return** $x + y$
		else return *INT_MAX*
NaturalNumber Subtract(x, y)	::=	**if** ($x < y$) **return** 0
		else return $x - y$

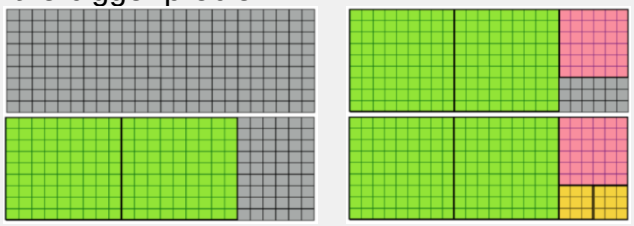
end *NaturalNumber*

Difference - Consider $gcd(16,24)$ our rule doesn't work because 16 doesn't divide 24 evenly. But $24 - 16 = 8$ does.



Since 24−16 divides 16 evenly, it must also divide 24 evenly. Note the 16×16 square. The 8×8 square fits on one side of it. But, it's a square, so it fits on the other side, too. It's a common divisor. Could there be a larger one? No. It would have to divide 16 evenly (green), so it would cover green. So, it would have to cover pink, too. Pink square is the biggest such square.

Sneaking up on an idea - Make as many big $a \times a$ squares as possible. The biggest square that covers the leftover rectangle is necessarily the biggest square the covers the original rectangle. Why? It has to evenly divide both sides. So, solve the smaller problem and we solve the bigger problem.



$$Gcd\ (9,24)$$
$$= gcd(6,9)$$
$$= gcd(3,6)$$

= 3, by the original special case.

Remainder : $gcd\ (a, b) = gcd(rem(b, a), a)$. Here, $a \leq b$.
Proof: Write $b = q \cdot a + r$ where $r = rem(b, a)$. Why? *Division theorem*. So, b is a linear combination of a *and* r, which implies that any divisor of *a and r* is a divisor of *b*.

Why? *Integer linear combination property*. Similarly, r is a linear combination of $a \; and \; b$, specifically $r = 1 \cdot b - q \cdot a$. Thus, any divisor of $a \; and \; b$ is a divisor of r. Why? *Integer linear combination property, again.*

So, A and B have the same common divisors as A and R. They must also then have the same greatest common divisor QED.

Video links:

Euclidean Algorithm

- *https://youtu.be/cOwyHTiW4KE*

How to find Greatest Common

- *https://youtu.be/P3YID7liBug*

Divisor using The Euclidean Algorithm

- *https://youtu.be/JUzYI1TYMcU*

Algorithm: Recursive

- *https://youtu.be/KEEKn7Me-ms*

Reference:

- *https://cs.brown.edu/courses/csci0220/static/files/notes/2-14.pdf*
- *http://gauss.math.luc.edu/greicius/Math201/Fall2012/Lectures/euclidean-algorithm.article.pdf*
- *https://pdfs.semanticscholar.org/1ef5/94e3f87c149fa3ec556147397580c1935a3e.pdf*

8

CHAPTER 9

COMPLEXITY OF ALGORITHM AND ANALYSIS OF ALGORITHMS

Objectives

After completing this course, you will be able to:

- Learn More About Algorithms
- Know More About Different Types of Algorithms
- Learn About the Analysis Algorithms

1

Algorithm Complexity

We cannot talk about efficiency of algorithms and data structures without explaining the term "algorithm complexity", which we have already mentioned several times in one form or another. We will avoid the mathematical definitions and we are going to give a simple explanation of what the term means.

Algorithm complexity is a measure which evaluates the order of the count of operations, performed by a given or algorithm as a function of the size of the input data. To put this simpler, complexity is a rough approximation of the number of steps necessary to execute an algorithm. When we evaluate complexity we speak of order of operation count, not of their exact count. For example if we have an order of N^2 operations to process N elements, then $N^2/2$ and $3*N^2$ are of one and the same quadratic order.

Algorithm complexity is commonly represented with the O(f) notation, also known as asymptotic notation or “Big O notation”, where f is the function of the size of the input data. The asymptotic computational complexity O(f) measures the order of the consumed resources (CPU time, memory, etc.) by certain algorithm expressed as function of the input data size.

Complexity can be constant, logarithmic, linear, n*log(n), quadratic, cubic, exponential, etc. This is respectively the order of constant, logarithmic, linear and so on, number of steps, are executed to solve a given problem. For simplicity, sometime instead of “algorithms complexity” or just “complexity” we use the term “running time”.

Constant	O(1)	It takes a constant number of steps for performing a given operation (for example 1, 5, 10 or other number) and this count does not depend on the size of the input data.
Logarithmic	O(log(N))	It takes the order of log(N) steps, where the base of the logarithm is most often 2, for performing a given operation on N elements. For example, if N = 1,000,000, an algorithm with a complexity O(log(N)) would do about 20 steps (with a constant precision). Since the base of the logarithm is not of a vital importance for the order of the operation count, it is usually omitted.
Linear	O(N)	It takes nearly the same amount of steps as the number of elements for performing an operation on N elements. For example, if we have 1,000 elements, it takes about 1,000 steps. Linear complexity means that the number of elements and the number of steps are linearly dependent, for example the number of steps for N elements can be N/2 or 3*N.
2	O(n*log(n))	It takes N*log(N) steps for performing a

		given operation on N elements. For example, if you have 1,000 elements, it will take about 10,000 steps.
quadratic	$O(n^2)$	It takes the order of N^2 number of steps, where the N is the size of the input data, for performing a given operation. For example if N = 100, it takes about 10,000 steps. Actually we have a quadratic complexity when the number of steps is in quadratic relation with the size of the input data. For example for N elements the steps can be of the order of $3*N^2/2$.
cubic	$O(n^3)$	It takes the order of N^3 steps, where N is the size of the input data, for performing an operation on N elements. For example, if we have 100 elements, it takes about 1,000,000 steps.
exponential	$O(2^n)$, O(N!), $O(n^k)$, …	It takes a number of steps, which is with an exponential dependability with the size of the input data, to perform an operation on N elements. For example, if N = 10, the exponential function 2^N has a value of 1024, if N = 20, it has a value of 1 048 576, and if N = 100, it has a value of a number with about 30 digits. The exponential function N! grows even faster: for N = 5 it has a value of 120, for N = 10 it has a value of 3,628,800 and for N = 20 – 2,432,90,008,176,640,000.

When evaluating complexity, constants are not taken into account, because they do not significantly affect the count of operations. Therefore an algorithm which does N steps and algorithms which do N/2 or 3*N respectively are considered linear and approximately equally efficient, because they perform a number of operations which is of the same order.

Complexity and Execution Time

The execution speed of a program depends on the complexity of the algorithm, which is executed. If this complexity is low, the program will execute fast even for a big number of elements. If the complexity is high, the program will execute slowly or will not even work (it will hang) for a big number of elements.

If we take an average computer from 2008, we can assume that it can perform about 50,000,000 elementary operations per second. This number is a rough approximation, of course. The different processors work with a different speed and the different elementary operations are performed with a different speed, and also the computer technology constantly evolves. Still, if we accept we use an average home

computer from 2008, we can make the following conclusions about the speed of execution of a given program depending on the algorithm complexity and size of the input data.

We can draw many conclusions from the above table:

- Algorithms with a constant, logarithmic or linear complexity are so fast that we cannot feel any delay, even with a relatively big size of the input data.
- Complexity O(n*log(n)) is similar to the linear and works nearly as fast as linear, so it will be very difficult to feel any delay.
- Quadratic algorithms work very well up to several thousand elements.
- Cubic algorithms work well if the elements are not more than 1,000.
- Generally these so called polynomial algorithms (any, which are not exponential) are considered to be fast and working well for thousands of elements.
- Generally the exponential algorithms do not work well and we should avoid them (when possible). If we have an exponential solution to a task, maybe we actually do not have a solution, because it will work only if the number of the elements is below 10-20. Modern cryptography is based exactly on this – there are not any fast (non-exponential) algorithms for finding the secret keys used for data encryption.

The data in the table is just for orientation, of course. Sometimes a linear algorithm could work slower than a quadratic one or a cubic algorithm could work faster than O(n*log(n)). The reasons for this could be many:

- It is possible the constants in an algorithm with a low complexity to be big and this could eventually make the algorithm slow. For example, if we have an algorithm, which makes 50*n steps and another one, which makes 1/100*n*n steps, for elements up to 5000 the quadratic algorithm will be faster than the linear.
- Since the complexity evaluation is made in the worst case scenario, it is possible a quadratic algorithm to work better than O(n*log(n)) in 99% of the cases. We can give an example with the algorithm QuickSort (the standard sorting algorithm in .NET Framework), which in the average case works a bit better than MergeSort, but in the worst case QuickSort can make the order of n^2 steps, while MergeSort does always O(n*log(n)) steps.
- It is possible an algorithm, which is evaluated to execute with a linear complexity, to not work so fast, because of an inaccurate complexity evaluation. For example if we search for a given word in an array of words, the complexity is linear, but at every step string comparison is performed, which is not an elementary operation and can take much more time than performing simple elementary operation (for example comparison of two integers).

Complexity by Several Variables

Complexity can depend on several input variables at once. For example, if we look for an element in a rectangular matrix with sizes M and N, the searching speed depends on M and N. Since in the worst case we have to traverse the entire matrix, we will do M*N number of steps at most. Therefore the complexity is O(M*N).

Best, Worst and Average Case

Complexity of algorithms is usually evaluated in the worst case (most unfavorable scenario). This means in the average case they can work faster, but in the worst case they work with the evaluated complexity and not slower.

Let's take an example: searching in array. To find the searched key in the worst case, we have to check all the elements in the array. In the best case we will have luck and we will find the element at first position. In the average case we can expect to check half the elements in the array until we find the one we are looking for. Hence in the worst case the complexity is O(N) – linear. In the average case the complexity is O(N/2) = O(N) – linear, because when evaluating complexity one does not take into account the constants. In the best case we have a constant complexity O(1), because we make only one step and directly find the element.

Roughly Estimated Complexity

Sometimes it is hard to evaluate the exact complexity of a given algorithm, because it performs operations and it is not known exactly how much time they will take and how many operations will be done internally. Let's take the example of searching a given word in an array of strings (texts). The task is easy: we have to traverse the array and search in every text with Substring() or with a regular expression for the given word.

We can ask ourselves the question: if we had 10,000 texts, would this work fast? What if the texts were 100,000? If we carefully think about it, we will implement that in order to evaluate adequately, we have to know how big are the texts, because there is a difference between searching in people's names (which are up to 50-100 characters) and searching in scientific articles (which are roughly composed by 20,000 – 30,000 characters). However, we can evaluate the complexity using the length of the texts, through which we are searching: it is at least O(L), where L is the sum of the lengths of all texts. This is a pretty rough evaluation, but it is much more accurate than complexity O(N), where N is the number of the texts, right? We should think whether we take into account all situations, which could occur. Does it matter how long the searched word is? Probably searching of long words is slower than searching of short words. In fact things are slightly different. If we search for "aaaaaaa" in the text "aaaaaabaaaaacaaaaaabaaaaacaaaaab", this will be slower than if we search for "xxx" in the same text, because in the first case we will get more sequential matches than in the second case. Therefore, in some special situations, searching seriously depends on the length of the word we search and the complexity O(L) could be underestimated.

Complexity by Memory

Besides the number of steps using a function of the input data, one can measure other resources, which an algorithm uses, for example memory, count of disk operations, etc. For some algorithms the execution speed is not as important as the memory they use. For example if a given algorithm is linear but it uses RAM in the order of N^2, it will be probably shortage of memory if N = 100,000 (then it will need memory in order of 9 GB RAM), despite the fact that it should work very fast.

5

Estimating Complexity – Examples

We are going to give several examples, which show how you can estimate the complexity of your algorithms, and decide whether the code written by you will work fast:
If we have a single loop from 1 to N, its complexity is linear – O(N):

When to Use a Particular Data Structure?

Let's skim through all the structures in the table above and explain in what situations we should use them as well as how their complexities are evaluated.

Array

The arrays are collections of fixed number of elements from a given type (for example numbers) where the elements preserved their order. Each element can be accessed through its index. The arrays are memory areas, which have a predefined size.

Adding a new element in an array is a slow operation. To do this we have to allocate a memory with the same size plus one and copy all the data from the original array to the new one.

Searching in an array takes time because we have to compare every element to the searched value. It takes N/2 comparisons in the average case.

Removing an element from an array is a slow operation. We have to allocate a memory with the same size minus one and copy all the old elements except the removed one.Accessing by index is direct, and thus, a fast operation.

The arrays should be used only when we have to process a fixed number of elements to which we need a quick access by index. For example, if we have to sort some numbers, we can keep them in an array and then apply some of the well-known sorting algorithms. If we have to change the elements' count, the array is not the correct data structure we should use.

Singly / Doubly Linked List (LinkedList<T>)

Singly and doubly linked lists hold collection of elements, which preserve their order. Their representation in the memory is dynamic, pointer-based. They are linked sequences of element.

Adding is a fast operation but it is a bit slower than adding to a List<T> because every time when we add an element to a linked list we allocate a new memory area. The memory allocation works at speed, which cannot be easily predicted.
Searching in a linked list is a slow operation because we have to traverse through all of its elements.

Accessing an element by index is a slow operation because there is no indexing in singly and doubly linked lists. You have to go through all the elements from the start one by one instead.

Removing an element at a specified index is a slow operation because reaching the element through its index is a slow operation. Removing an element with a specified value is a slow operation too, because it involves searching.

Linked list can quickly add and remove elements (with a constant complexity) at its two ends (head and tail). Hence, it is very handy for an implementation of stacks, queues and similar data structures.

Linked lists are rarely used in practice because the dynamic arrays (List<T>) can do almost exact same operations LinkedList does, plus for the most of them it works faster and more comfortable.

When you need a linked list, use List<T> instead of LinkedList<T>, because it doesn't work slower and it gives you better speed and flexibility. Use LinkedList when you have to add and remove elements at both ends of the data structure.

When you need to add and remove elements at both ends of the list, use LinkedList<T>. Otherwise use List<T>.

Dynamic Array (List<T>)

Dynamic array (List<T>) is one of the most popular data structures used in programming. It does not have fixed size like arrays, and allows direct access through index, unlike linked lists (LinkedList<T>). The dynamic array is also known as "array list", "resizable array" and "dynamic array".

List<T> holds its elements in an array, which has a bigger size than the count of the stored elements. Usually when we add an element, there is an empty cell in the list's inner array. Therefore this operation takes a constant time. Occasionally the array has been filled and it has to expand. This takes linear time, but it rarely happens. If we have a large amount of additions, the average-case complexity of adding an element to List<T> will a constant – O(1). If we sum the steps needed for adding 100,000 elements (for both cases – "fast add" and "add with expand") and divide by 100,000, we will obtain a constant which will be nearly the same like for adding 1,000,000 elements. This statistically-averaged complexity calculated for large enough amount of operations is called amortized complexity. Amortized linear complexity means that if we add 10,000 elements consecutively, the overall count of steps will be of the order of 10,000. In most cases add it will execute in a constant time, while very rarely adding will execute in linear time.

Searching in List<T> is a slow operation because you have to traverse through all the elements.

Removing by index or value executes in a linear time. It is a slow operation because we have to move all the elements after the deleted one with one position to the left.

The indexed access in List<T> is instant, in a constant time, since the elements are internally stored in an array.

Practically List<T> combines the best of arrays and lists, for which it is a preferred data structure in many situations. For example if we have to process a text file and to extract from it all words (with duplicates), which match a regular expression, the most suitable data structure in which we can accumulate them is List<T>, because we need a list, the length of which is unknown in advance and can grow dynamically. The dynamic array (List<T>) is appropriate, when we have to add elements frequently as well as keeping their order of addition and access them through index. If we often we have to search or delete elements, List<T> is not the right data structure.

Stack

Stack is a linear data structure in which there are 3 operations defined: adding an element at the top of the stack (push), removing an element from the top of the stack (pop) and inspect the element from the top without removing it (peek). All these operations are very fast – it takes a constant time to execute them. The stack does not support the operations search and access through index.

The stack is a data structure, which has a LIFO behavior (last in, first out). It is used when we have to model such a behavior – for example, if we have to keep the path to the current position in a recursive search.

Queue

Queue is a linear data structure in which there are two operations defined: adding an element to the tail (enqueue) and extract the front-positioned element from the head (dequeue). These two operations take a constant time to execute, because the queue is usually implemented with a linked list. We remind that the linked list can quickly add and remove elements from its both ends.

The queue's behavior is FIFO (first in, first out). The operations searching and accessing through index are not supported. Queue can naturally model a list of waiting people, tasks or other objects, which have to be processed in the same order as they were added (enqueued).

As an example of using a queue we can point out the implementation of the BFS (breadth-first search) algorithm, in which we start from an initial element and all its neighbors are added to a queue. After that they are processed in the order they were added and their neighbors are added to the queue too. This operation is repeated until we reach the element we are looking for or we process all elements.

Video links:

What are Data Structures and Algorithms? Data Structures and Algorithms 002

- *https://www.youtube.com/watch?v=TEI5VspLVik*

Introduction to Big O Notation and Time Complexity (Data Structures & Algorithms #7)

- *https://www.youtube.com/watch?v=D6xkbGLQesk*

Reference:

- *https://introprogramming.info/english-intro-csharp-book/read-online/chapter-19-data-structures-and-algorithm-complexity/#_Toc362296554*
- *https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwiAhaKxz-PrAhUKVpQKHcDhDLUQFjAAegQIBRAB&url=https%3A%2F%2Fwww.cs.purdue.edu%2Fhomes%2Fayg%2FCS251%2Fslides%2Fchap2.pdf&usg=AOvVaw3fbK2l5pHR0Spv4kF247i-*
- *https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwiAhaKxz-PrAhUKVpQKHcDhDLUQFjACegQIAhAB&url=http%3A%2F%2Fcslabcms.nju.edu.cn%2Fproblem_solving%2Fimages%2Ff%2Ff0%2FAn_Introduction_to_the_Analysis_of_Algorithms_%25282nd_Edition_Robert_Sedgewick%252C_Philippe_Flajolet%2529.pdf&usg=AOvVaw0Wg81OKPg56ASA1-xO_Fxj*

CHAPTER 10

INTRODUCTION OF GRAPH THEORY

Objectives

After completing this course, you will be able to:

- Learn more about graphs theory
- Have a knowledge about different types of graphs

1

Graph

A graph is a diagram of points and lines connected to the points. It has at least one line joining a set of two vertices with no vertex connecting itself. The concept of graphs in graph theory stands up on some basic terms such as point, line, vertex, edge, degree of vertices, properties of graphs, etc. Here, in this chapter, we will cover these fundamentals of graph theory.

Point

A point is a particular position in a one-dimensional, two-dimensional, or three-dimensional space. For better understanding, a point can be denoted by an alphabet. It can be represented with a dot.

Example:

Here, the dot is a point named ‘a’.

Line

A Line is a connection between two points. It can be represented with a solid line.

Example:

Here, ‘a’ and ‘b’ are the points. The link between these two points is called a line.

Vertex

A vertex is a point where multiple lines meet. It is also called a node. Similar to points, a vertex is also denoted by an alphabet.

Example:

Here, the vertex is named with an alphabet ‘a’.

Edge

An edge is the mathematical term for a line that connects two vertices. Many edges can be formed from a single vertex. Without a vertex, an edge cannot be formed. There must be a starting vertex and an ending vertex for an edge.

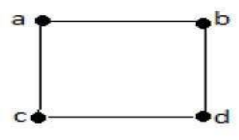
Example:

Here, ‘a’ and ‘b’ are the two vertices and the link between them is called an edge.

Graph

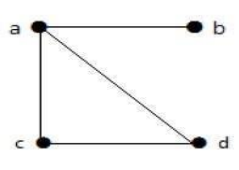
A graph ‘G’ is defined as G = (V, E) Where V is a set of all vertices and E is a set of all edges in the graph.

Example 1:



In the above example, ab, ac, cd, and bd are the edges of the graph. Similarly, a, b, c, and d are the vertices of the graph.

Example 2:



In this graph, there are four vertices a, b, c, and d, and four edges ab, ac, ad, and cd.

Loop

In a graph, if an edge is drawn from vertex to itself, it is called a loop.

Example 1:

In the above graph, V is a vertex for which it has an edge (V, V) forming a loop.

Example 2:

In this graph, there are two loops which are formed at vertex a, and vertex b.

Degree of Vertex

It is the number of vertices adjacent to a vertex V.

Notation − deg(V)

In a simple graph with n number of vertices, the degree of any vertices is − $\deg(v) \leq n - 1 \; \forall \; v \in G$

A vertex can form an edge with all other vertices except by itself. So the degree of a vertex will be up to the number of vertices in the graph minus 1. This 1 is for the self-vertex as it cannot form a loop by itself. If there is a loop at any of the vertices, then it is not a Simple Graph.

Degree of vertex can be considered under two cases of graphs −

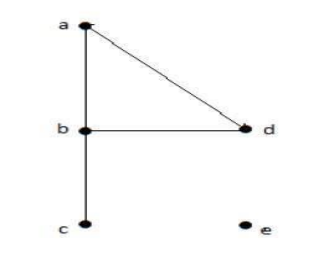
- Undirected Graph
- Directed Graph

Degree of Vertex in an Undirected Graph

An undirected graph has no directed edges. Consider the following examples.

Example 1:

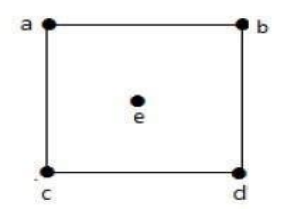
Take a look at the following graph −



In the above Undirected Graph,

- deg(a) = 2, as there are 2 edges meeting at vertex ‘a’.
- deg(b) = 3, as there are 3 edges meeting at vertex ‘b’.
- deg(c) = 1, as there is 1 edge formed at vertex ‘c’
- So ‘c’ is a pendent vertex.
- deg(d) = 2, as there are 2 edges meeting at vertex ‘d’.
- deg(e) = 0, as there are 0 edges formed at vertex ‘e’.
- So ‘e’ is an isolated vertex.

Example 2:
Take a look at the following graph −



In the above graph,
deg(a) = 2, deg(b) = 2, deg(c) = 2, deg(d) = 2, and deg(e) = 0.
The vertex ‘e’ is an isolated vertex. The graph does not have any pendent vertex.

Degree of Vertex in a Directed Graph

In a directed graph, each vertex has an indegree and an outdegree.

Indegree of a Graph

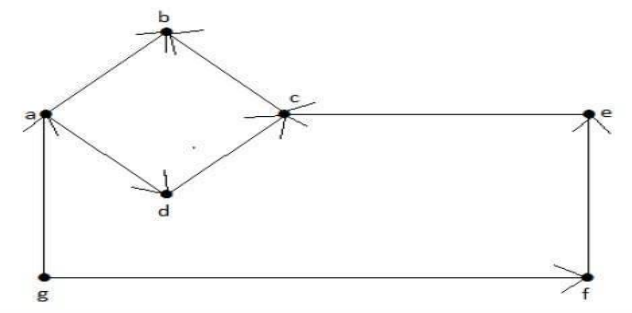
- Indegree of vertex V is the number of edges which are coming into the vertex V.
- Notation − deg−(V).

Outdegree of a Graph

- Outdegree of vertex V is the number of edges which are going out from the vertex V.
- Notation − deg+(V).

Consider the following examples.

Example 1:
Take a look at the following directed graph. Vertex ‘a’ has two edges, ‘ad’ and ‘ab’, which are going outwards. Hence its outdegree is 2. Similarly, there is an edge ‘ga’, coming towards vertex ‘a’. Hence the indegree of ‘a’ is 1.

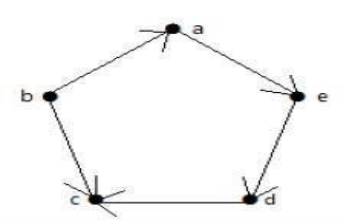


The indegree and outdegree of other vertices are shown in the following table −

Vertex	Indegree	Outdegree
a	1	2
b	2	0
c	2	1
d	1	1
e	1	1
f	1	1
g	0	2

Example 2:

Take a look at the following directed graph. Vertex ‘a’ has an edge ‘ae’ going outwards from vertex ‘a’. Hence its outdegree is 1. Similarly, the graph has an edge ‘ba’ coming towards vertex ‘a’. Hence the indegree of ‘a’ is 1.



The indegree and outdegree of other vertices are shown in the following table −

Vertex	Indegree	Outdegree
a	1	1
b	0	2
c	2	0
d	1	1
e	1	1

Pendent Vertex

By using degree of a vertex, we have a two special types of vertices. A vertex with degree one is called a pendent vertex.

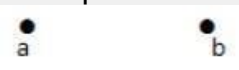
Example:

Here, in this example, vertex ‘a’ and vertex ‘b’ have a connected edge ‘ab’. So with respect to the vertex ‘a’, there is only one edge towards vertex ‘b’ and similarly with respect to the vertex ‘b’, there is only one edge towards vertex ‘a’. Finally, vertex ‘a’ and vertex ‘b’ has degree as one which are also called as the pendent vertex.

Isolated Vertex

A vertex with degree zero is called an isolated vertex.

Example:



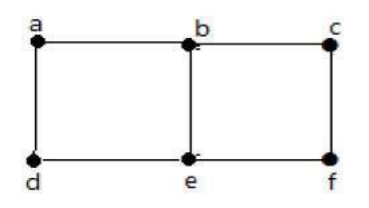
Here, the vertex ‘a’ and vertex ‘b’ has a no connectivity between each other and also to any other vertices. So the degree of both the vertices ‘a’ and ‘b’ are zero. These are also called as isolated vertices.

Adjacency

Here are the norms of adjacency −

- In a graph, two vertices are said to be adjacent, if there is an edge between the two vertices. Here, the adjacency of vertices is maintained by the single edge that is connecting those two vertices.
- In a graph, two edges are said to be adjacent, if there is a common vertex between the two edges. Here, the adjacency of edges is maintained by the single vertex that is connecting two edges.

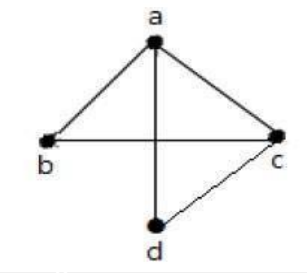
Example 1:



In the above graph −

- ‘a’ and ‘b’ are the adjacent vertices, as there is a common edge ‘ab’ between them.
- ‘a’ and ‘d’ are the adjacent vertices, as there is a common edge ‘ad’ between them.
- ab’ and ‘be’ are the adjacent edges, as there is a common vertex ‘b’ between them.
- be’ and ‘de’ are the adjacent edges, as there is a common vertex ‘e’ between them.

Example 2:

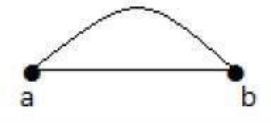


In the above graph −

- ‘a’ and ‘d’ are the adjacent vertices, as there is a common edge ‘ad’ between them.
- ‘c’ and ‘b’ are the adjacent vertices, as there is a common edge ‘cb’ between them.
- ‘ad’ and ‘cd’ are the adjacent edges, as there is a common vertex ‘d’ between them.
- ‘ac’ and ‘cd’ are the adjacent edges, as there is a common vertex ‘c’ between them.

Parallel Edges

In a graph, if a pair of vertices is connected by more than one edge, then those edges are called parallel edges.

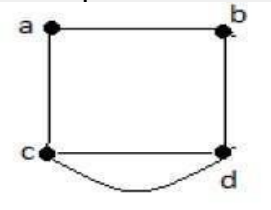


In the above graph, ‘a’ and ‘b’ are the two vertices which are connected by two edges ‘ab’ and ‘ab’ between them. So it is called as a parallel edge.

Multi Graph

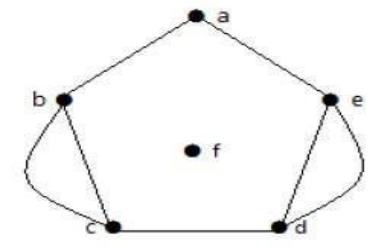
A graph having parallel edges is known as a Multigraph.

Example 1:



In the above graph, there are five edges ‘ab’, ‘ac’, ‘cd’, ‘cd’, and ‘bd’. Since ‘c’ and ‘d’ have two parallel edges between them, it a Multigraph.

Example 2:

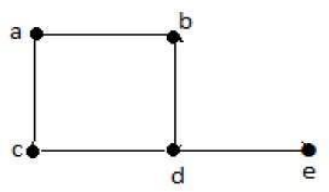


In the above graph, the vertices ‘b’ and ‘c’ have two edges. The vertices ‘e’ and ‘d’ also have two edges between them. Hence it is a Multigraph.

Degree Sequence of a Graph

If the degrees of all vertices in a graph are arranged in descending or ascending order, then the sequence obtained is known as the degree sequence of the graph.

Example 1:



Distance between Two Vertices

It is number of edges in a shortest path between Vertex U and Vertex V. If there are multiple paths connecting two vertices, then the shortest path is considered as the distance between the two vertices.

Notation − d (U, V)

There can be any number of paths present from one vertex to other. Among those, you need to choose only the shortest one.

Eccentricity of a Vertex

The maximum distance between a vertex to all other vertices is considered as the eccentricity of vertex.

Notation − e(V)

The distance from a particular vertex to all other vertices in the graph is taken and among those distances, the eccentricity is the highest of distances.

Radius of a Connected Graph

The minimum eccentricity from all the vertices is considered as the radius of the Graph G. The minimum among all the maximum distances between a vertex to all other vertices is considered as the radius of the Graph G.

Notation − r(G)

From all the eccentricities of the vertices in a graph, the radius of the connected graph is the minimum of all those eccentricities.

Diameter of a Graph

The maximum eccentricity from all the vertices is considered as the diameter of the Graph G. The maximum among all the distances between a vertex to all other vertices is considered as the diameter of the Graph G.

Notation − d(G) − From all the eccentricities of the vertices in a graph, the diameter of the connected graph is the maximum of all those eccentricities.

Central Point

If the eccentricity of a graph is equal to its radius, then it is known as the central point of the graph.

Centre

The set of all central points of ‘G’ is called the centre of the Graph.

8

Circumference

The number of edges in the longest cycle of ‘G’ is called as the circumference of ‘G’.

Girth

The number of edges in the shortest cycle of ‘G’ is called its Girth.

Video links:

Graph Theory - An Introduction!

- *https://www.youtube.com/watch?v=HmQR8Xy9DeM*

Graph Theory Overview

- *https://www.youtube.com/watch?v=82zIRaRUsaY*

Reference:

- *https://en.wikipedia.org/wiki/Graph_theory*
- *http://discrete.openmathbooks.org/dmoi2/ch_graphtheory.html*
- *http://discrete.openmathbooks.org/dmoi2/ch_graphtheory.html*

9