GATE 2015

Praneeth A S (UG201110023)

B.Tech(Computer Science & Engineering)

Indian Institute of Technology Jodhpur
Jodhpur, Rajasthan 342011, India

August 4, 2014

Dedicated to my beloved parents who have stood beside me entire my career.

Acknowledgements

Many Sources have been used.

Preface

To be used for GATE Preparation.

Contents

List of Algorithms

List of Figures

List of Tables

List of listings

Todo list

Part I

Algorithms

Chapter 1

Test

Algorithm 1 Calculate $y = x^n$

Input: $n \geq 0 \vee x \neq 0$
Output: $y = x^n$

1. $y \leftarrow 1$
2. **if** $n < 0$ **then**
3. $\quad X \leftarrow \frac{1}{x}$
4. $\quad N \leftarrow -n$
5. **else**
6. $\quad X \leftarrow x$
7. $\quad N \leftarrow n$
8. **end if**
9. **while** $N \neq 0$ **do**
10. $\quad$ **if** N is even **then**
11. $\quad\quad X \leftarrow X \times X$
12. $\quad\quad N \leftarrow N/2$
13. $\quad$ **else** // N is odd
14. $\quad\quad y \leftarrow y \times X$
15. $\quad\quad N \leftarrow N - 1$
16. $\quad$ **end if**
17. **end while** // end of while

Chapter 2

Complexity Analysis(Asymptotic Notations)

2.1 Θ Notation

Θ((g(n)) = f(n): ∃ positive constants c_1,c_2 and n_0 such that

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad \forall n \geq n_0 \tag{2.1}$$

2.2 Big O Notation

O((g(n)) = f(n): ∃ positive constants c_0 and n_0 such that

$$0 \leq f(n) \leq c_0 g(n) \qquad \forall n \geq n_0 \tag{2.2}$$

2.3 Ω Notation

Ω((g(n)) = f(n): ∃ positive constants c_0 and n_0 such that

$$0 \leq c_0 g(n) \leq f(n) \qquad \forall n \geq n_0 \tag{2.3}$$

Chapter 3

Solving Recurrences

3.1 Substitution Method

Definition 1 - Substitution Method.

We make a guess for the solution and then we use mathematical induction to prove the the guess is correct or incorrect. For example consider the recurrence $T(n) = 2T\left(\frac{n}{2}\right) + n$.

3.1.1 Example

We guess the solution as $T(n) = O(n \log n)$. Now we use induction to prove our guess. We need to prove that $T(n) \leq cn \log n$. We can assume that it is true for values smaller than n.

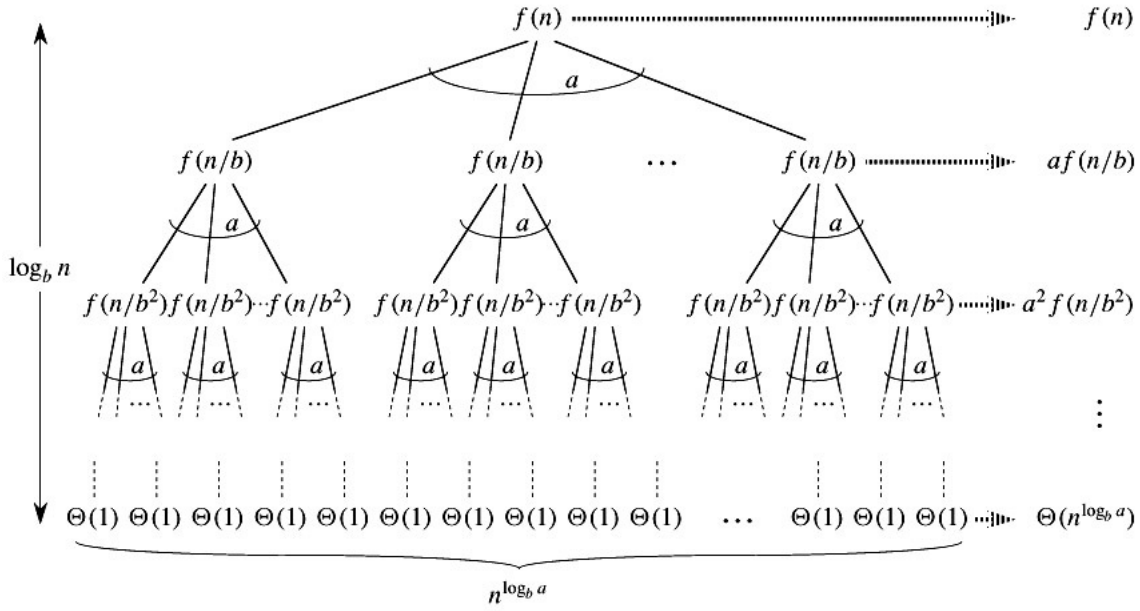
$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ &\leq c\left(\frac{n}{2}\right)\log\left(\frac{n}{2}\right) + n \\ &= cn\log n - cn\log 2 + n \\ &= cn\log n - cn + n \\ &\leq cn\log n \end{aligned}$$

3.2 Recurrence Tree Method

Read it from here Recurrence Tree Method

3.3 Master Method

Figure 3.1: Master Theorem



$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $a \geq 1$ and $b > 1$
3 cases:

1. If f(n) $= \Theta(n^c)$ where $c < \log_b a$ then $T(n) = \Theta\left(n^{\log_b a}\right)$
2. If f(n) $= \Theta(n^c)$ where $c = \log_b a$ then $T(n) = \Theta(n^c \log n)$
3. If f(n) $= \Theta(n^c)$ where $c > \log_b a$ then $T(n) = \Theta(f(n))$

Chapter 4

Sorting

Sorting Method	**Usage**	**Time Complexity**			**Space Complexity**	**No. of swaps**	**Stability**
		Best	Average	Worst			
Insertion Sort							✓
Bucket Sort							
Bubble Sort							✓
Heap Sort							×
Quick Sort							×
Merge Sort							✓

Table 4.1: Summary of Sorting Methods

Auxiliary Space is the extra space or temporary space used by an algorithm.
Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both **Auxiliary space** and **space used by input.**
Stability: If two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array.

Chapter 5

Searching

5.1 Linear Search

5.2 Binary Search

5.3 Interpolation Search [1]

This is a margin note using the geometry package, set at 0cm vertical offset to the first line it is typeset.

5.4 Summary

Method	Time Complexity

Table 5.1: Summary of Search Methods

Chapter 6

Divide & Conquer

Chapter 7

Greedy Algorithms

7.1 Introduction

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems. An optimization problem can be solved using Greedy if the problem has the following property: *At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.*
Examples: *Kruskal MST, Prim MST, Dijsktra Shortest Path, Huffman Coding, Activity Selection Problem* etc.,

1. **Kruskal's Minimum Spanning Tree (MST):** In Kruskal's algorithm, we create a MST by picking edges one by one. The Greedy Choice is to pick the smallest weight edge that doesn't cause a cycle in the MST constructed so far.

2. **Prim's Minimum Spanning Tree:** In Prim's algorithm also, we create a MST by picking edges one by one. We maintain two sets: set of the vertices already included in MST and the set of the vertices not yet included. The Greedy Choice is to pick the smallest weight edge that connects the two sets.

3. **Dijkstra's Shortest Path:** The Dijkstra's algorithm is very similar to Prim's algorithm. The shortest path tree is built up, edge by edge. We maintain two sets: set of the vertices already included in the tree and the set of the vertices not yet included. The Greedy Choice is to pick the edge that connects the two sets and is on the smallest weight path from source to the set that contains not yet included vertices.

4. **Huffman Coding:** Huffman Coding is a loss-less compression technique. It assigns variable length bit codes to different characters. The Greedy Choice is to assign least bit length code to the most frequent character.

7.2 Spanning Trees

MST

Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A **minimum spanning tree (MST)** or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each

edge of the spanning tree.
A minimum spanning tree has $(V - 1)$ edges where V is the number of vertices in the given graph.

Kruskal's MST Algorithm

7.3 Activity Selection Problem

Definition 2 - Activity Selection Problem.

You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

Chapter 8

Dynamic Programming

8.1 Introduction

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again. Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.
Example: Floyd Warshall Algorithm, Bellman Ford Algorithm

1. **Overlapping Subproblems:** Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these donâĂŹt have to recomputed.
   Uses: Fibonacci Numbers

2. **Optimal Substructure:** A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.
   For example the shortest path problem has following optimal substructure property: If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v.
   Uses: Longest Increasing Subsequence, Shortest Path.

Two Approaches:

1. Memoization (Top Down)

2. Tabulation (Bottom Up)

8.2 Fibonacci Numbers

8.3 Ugly Numbers

Definition 3 - Ugly Numbers.

Ugly numbers are numbers whose only prime factors are 2, 3 or 5. The sequence $1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, \ldots$ shows the first 11 ugly numbers. By convention, 1 is included.

See 15 for Dynamic Programming version of problem.

Chapter 9

NP Classes

9.1 Introduction

Definition 4 - P.

A decision problem that can be solved in polynomial time. That is, given an instance of the problem, the answer yes or no can be decided in polynomial time.

Definition 5 - NP.

NP - Non-Deterministic Polynomial Time. A decision problem where instances of the problem for which the answer is yes have proofs that can be verified in polynomial time. This means that if someone gives us an instance of the problem and a certificate (sometimes called a witness) to the answer being yes, we can check that it is correct in polynomial time.
A non-deterministically solvable problem by a turing machine(non-deterministic turing machine) in polynomial time.

Definition 6 - NP Complete.

A decision problem L is **NP-complete** if it is in the set of NP problems and also in the set of NP-hard problems. **NP-complete** is a subset of NP, the set of all decision problems whose solutions can be verified in polynomial time; NP may be equivalently defined as the set of decision problems that can be solved in polynomial time on a non-deterministic Turing machine. A problem p in NP is also NP-complete if every other problem in NP can be transformed into p in polynomial time.
A decision problem C is NP-complete if:

1. C is in NP, and
2. Every problem in NP is reducible to C in polynomial time.

Consequence: If we had a polynomial time algorithm (on a UTM, or any other Turing-equivalent abstract machine) for C, we could solve all problems in NP in polynomial time. ie., **P = NP = NP-Complete**
Examples: Boolean satisfiability problem (Sat.), Knapsack problem, Hamiltonian path problem, Travelling salesman problem, Subgraph isomorphism problem, Subset sum problem, Clique problem, Vertex cover problem, Independent set problem, Dominating set problem, Graph coloring problem etc.,

Definition 7 - NP Hard.

The precise definition here is that a problem X is **NP-hard** if there is an NP-complete problem Y such that Y is reducible to X in polynomial time. But since any NP-complete problem can

be reduced to any other NP-complete problem in polynomial time, all NP-complete problems can be reduced to any NP-hard problem in polynomial time. Then if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems in polynomial time.
If a problem is **NP-hard**, this means I can reduce any problem in NP to that problem. This means if I can solve that problem, I can easily solve any problem in NP. If we could solve an NP-hard problem in polynomial time, this would prove P = NP.
Examples: The halting problem is the classic NP-hard problem. This is the problem that given a program P and input I, will it halt?

Definition 8 - Polynomial Reducible.

Can arbitrary instances of problem Y be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to a black box that solves problem X?
Suppose $Y \leq_P X$. If X can be solved in polynomial time, then Y can be solved in polynomial time. X is at least as hard as Y (with respect to polynomial time). Y is polynomial-time reducible to X. Suppose $Y \leq_P X$. If Y cannot be solved in polynomial time, then X cannot be solved in polynomial time.

9.2 Problem Definitions

Definition 9 - Circuit SAT.

Definition 10.

Part II

Data Structures

Chapter 10

Hashing

10.1 Introduction

- Prime number to be used for hashing should be very large to avoid collisions.
- Two keys hash to the same value (this is known as a collision).
- This hash function involves all characters in the key and can generally be expected to distribute well (it computes $\sum_{i=0}^{\text{KeySize-1}}$ Key(KeySize - i - 1) $\times 37^i$ and brings the result into proper range). The code computes a polynomial function (of 37) by use of Horner's rule. For instance, another way of computing $h_k = k_0 + 37 \times k_1 + 37^2 k_2$ is by the formula $h_k = ((k_2) \times 7 + k_1) \times 37 + k_0$. HornerâĂŹs rule extends this to an n^{th} degree polynomial.

```
/**
* A hash routine for string objects.
*/
unsigned int hash( const string & key, int tableSize ){
unsigned int hashVal = 0;
for( char ch : key )
hashVal = 37 * hashVal + ch;
return hashVal % tableSize;
}
```

- The first strategy, commonly known as **separate chaining**, is to keep a list of all elements that hash to the same value. Might be preferable to avoid their use (since these lists are doubly linked and waste space)
- We define the load factor, λ, of a hash table to be the ratio of the number of elements in the hash table to the table size.
- The average length of a list is λ. The effort required to perform a search is the constant time required to evaluate the hash function plus the time to traverse the list.
- In an unsuccessful search, the number of nodes to examine is λ on average.
- A successful search requires that about 1 + $(\frac{\lambda}{2})$ links be traversed.
- The general rule for separate chaining hashing is to make the table size about as large as the number of elements expected (in other words, let $\lambda \approx 1$). If the load factor exceeds 1, we expand the table size by calling **rehash**.

Table 10.1: Empty Table

0	1	2	3	4	5	6	7	8	9

Table 10.2: Empty Table for Quadratic Probing

0	1	2	3	4	5	6	7	8	9

- Generally, the load factor should be below $\lambda = 0.5$ for a hash table that doesnâĂŹt use separate chaining. We call such tables **probing hash tables**.
- Three common collision resolution strategies:
  - Linear Probing
  - Quadratic Probing
  - Double Hashing
- Linear Probing
  - In linear probing, f is a linear function of i, typically f(i) = i. This amounts to trying cells sequentially (with wraparound) in search of an empty cell. Table 10.3 shows the result of inserting keys 89, 18, 49, 58, 69 into a hash table using the same hash function as before and the collision resolution strategy, $f(i) = i$.
  -

0	1	2	3	4	5	6	7	8	9
									89

0	1	2	3	4	5	6	7	8	9
								18	89

0	1	2	3	4	5	6	7	8	9
49								18	89

0	1	2	3	4	5	6	7	8	9
49	58							18	89

0	1	2	3	4	5	6	7	8	9
49	58	69						18	89

  - Primary clustering, means that any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.
- Quadratic Probing:
  - $f(i) = i^2$
  - When 49 collides with 89, the next position attempted is one cell away. This cell is empty, so 49 is placed there. Next, 58 collides at position 8. Then the cell one away is tried, but another collision occurs. A vacant cell is found at the next cell tried, which is $2^2 = 4$ away. 58 is thus placed in cell 2.
  -

0	1	2	3	4	5	6	7	8	9
									89

0	1	2	3	4	5	6	7	8	9
								18	89

0	1	2	3	4	5	6	7	8	9
49								18	89

0	1	2	3	4	5	6	7	8	9
49		58						18	89

0	1	2	3	4	5	6	7	8	9
49		58	69					18	89

  -

Table 10.3: Empty Table for Double Hashing

0	1	2	3	4	5	6	7	8	9

Theorem 1.

If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

- Double Hashing
  - For double hash- ing, one popular choice is f(i) = i.$hash_2$(x).
  - $hash_2$(x) must be never evaluated to 0.
  - $hash_2$(x) = R - (x mod R), with R = 7, a prime smaller than TableSize
  - The first collision occurs when 49 is inserted. $hash_2$(49) = 7 - 0 = 7, so 49 is inserted in position 6. $hash_2$(58) = 7 - 2 = 5, so 58 is inserted at location 3. Finally, 69 collides and is inserted at a distance $hash_2$(69) = 7 - 6 = 1 away. If we tried to insert 60 in position 0, we would have a collision. Since $hash_2$(60) = 7 - 4 = 3, we would then try positions 3, 6, 9, and then 2 until an empty spot is found.
  -

0	1	2	3	4	5	6	7	8	9
									89

0	1	2	3	4	5	6	7	8	9
								18	89

0	1	2	3	4	5	6	7	8	9
						49		18	89

0	1	2	3	4	5	6	7	8	9
					58			49		18	89

0	1	2	3	4	5	6	7	8	9
69			58			49		18	89

  - If the table size is not prime, it is possible to run out of alternative locations prematurely.

10.2 ReHashing

- As an example, suppose the elements 13, 15, 24, and 6 are inserted into a linear probing hash table of size 7. The hash function is h(x) = x mod 7.

0	1	2	3	4	5	6
6	15		24			13

- If 23 is inserted into the table, the resulting table will be over 70 percent full. Because the table is so full, a new table is created. The size of this table is 17, because this is the first prime that is twice as large as the old table size. The new hash function is then h(x) = x mod 17. The old table is scanned, and elements 6, 15, 23, 24, and 13 are inserted into the new table. The resulting table:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
						6	23	24					13		15	

Part III

Theory Of Computation

References [2], Wikipedia articles for definitions of Halting Problem, Recursive Sets etc.,

Chapter 11

Finite Automaton

Definition 11 - DFA,NFA, $\epsilon - NFA$**.**

$$\boldsymbol{M = (Q, \Sigma, \delta, q_0, F)}$$

Q : Set of states. In $\boldsymbol{\epsilon - NFA}$ $\boldsymbol{\epsilon}$ is also a state.
$\boldsymbol{\Sigma}$: Set of alphabets
$\boldsymbol{\delta}$: Moving function
$\boldsymbol{q_0}$: Initial State
$\boldsymbol{F}$: Set of Final States. In NFA $\boldsymbol{2^Q}$ is the total no. of possible final states.
In $\boldsymbol{\epsilon - NFA\ \delta(q_0, 01) = \epsilon - CLOSURE(\delta()\delta(q_0, 0), 1))}$.

Theorem 2.

If L is accepted by an NFA, there exists an equivalent DFA accepting by L.

Definition 12 - ϵ**-closure.**

ϵ-closure denote the set of all vertices p such that there is a path from q to p labelled ϵ.

Definition 13.

If L is accepted by an NFA with ϵ transitions, then L is accepted by an NFA without ϵ transitions.

Definition 14.

$L^* = \cup_{i=0}^{\infty} L^i$. $L^+ = \cup_{i=1}^{\infty} L^i$

Theorem 3.

Let r be a regular expression. Then there exists an NFA with ϵ-transitions that accepts $L(r)$.

Definition 15 - Regular Language.

If L is accepted by DFA, then L is denoted by a regular expression.
$R_{ij}^k = R_{ik}^{k-1}(R_{ik}^{k-1})^* R_{kj}^{k-1} \cup R_{ij}^{k-1}$
$R_{ij}^0 = \{a | \delta(q_i, a) = q_j\} if i \neq j$
$\{a | \delta(q_i, a) = q_j\} \cup \{\epsilon\} if i = j$
R_{ij}^k is the set of all strings that take the finite automaton from state q_i to state q_j without going through any state numbered higher than k.

Chapter 12

Undecidability

12.1 Definitions

Definition 16 - Undecidable Problems.

An **undecidable problem** is a decision problem for which it is known to be impossible to construct a single algorithm that always leads to a correct yes-or-no answer.

Definition 17 - Halting Problem.

Given the description of an arbitrary program and a finite input, decide whether the program finishes running or will run forever. (Undecidable)

Definition 18 - Recursive Sets.

A set of natural numbers is called **recursive**, computable or decidable if there is an algorithm which terminates after a finite amount of time and correctly decides whether or not a given number belongs to the set.

Definition 19 - Recursively Enumerable Sets.

A set S of natural numbers is called **recursively enumerable**, computably enumerable, semidecidable, provable or Turing-recognizable if: there is an algorithm such that the set of input numbers for which the algorithm halts is exactly S.
Example: Set of languages we can accept using a Turing machine.

NOTE: There is no algorithm that decides the mebership of a particular string in a language.

Chapter 13

Turing Machines

13.1 Notation

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

1. **Q**: The finite set of states of the finite control.
2. **Σ**: The finite set of input symbols.
3. **Γ**: The complete set of tape symbols; **Σ** is always a subset of **Γ**.
4. δ: The transition function. The arguments of $\delta(q, X)$ are a state q and a tape symbol X. The value of $\delta(q, X)$, if it is defined, is a triple (p, Y, D), where:
   (a) p is the next state, in Q.
   (b) Y is the symbol, in **Γ** , written in the cell being scanned, replacing whatever symbol was there.
   (c) D is a direction, either Lor R, standing for "left" or "right" respectively, and telling us the direction in which the head moves.
5. q_0 : The start state, a member of Q, in which the finite control is found initially.
6. **B**: The blank symbol. This symbol is in **Γ** but not in E; i.e., it is not an input symbol. T he blank appears initially in all but the finite number of initial cells that hold input symbols.
7. **F**: The set of final or accepting states, a subset of Q.

Use the string $X_1X_2 \ldots X_{i-1}qX_iX_{i+1} \ldots X_n$ to represent an ID in which

1. q is the state of the Turing machine.
2. The tape head is scanning the i^{th} symbol from the left.
3. $X_1X_2 \ldots X_n$ the portion of the tape between the leftmost and the rightmost nonblank. As an exception, if the head is to the left of the leftmost nonblank or to the right of the rightmost nonblank, t hen some prefix or suffix of $X_1X_2 \ldots X_n$ will be blank, and i will be 1 or n, respectively.

A move M is defined as $\boldsymbol{X_1X_2 \ldots X_{i-1}qX_iX_{i+1} \ldots X_n \vdash_M X_1X_2 \ldots X_{i-2}pX_{i-1}X_iX_{i+1} \ldots X_n}$

State	Symbol 0	1	X	Y	B
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	(q_4, B, R)
q_4	–	–	–	–	–

Figure 13.1: Turing machine accepting $\boldsymbol{0^n1^n}$ for $\boldsymbol{n \geq 1}$

Chapter 14

Regular Sets

14.1 Pumping Lemma

Pumping Lemma is used to test whether a language is regular or not based on the property that languages obatined from regular languages are regular. If a language is regular, it is accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ with some particular number of states, say n.

Lemma 3.1 - Pumping Lemma.

Let L be a regular set. Then there is a constant n such that if z is any word in L, and $|z| > n$, we may write $z = uvw$ in such a way that $|uv| \leq n, |v| \geq 1$, and for all $i > 0$, $uv^i w$ is in L. Furthermore, n is no greater than the number of states of the smallest FA accepting L.

Examples:

1. $0^{i^2} is not a regular language.$
2. L be the set of strings of 0's and 1's, beginning with a 1, whose value treated as a binary number is a prime. is not a regular language.

Theorem 4.

The regular sets are closed under union ($L_1 \cup L_2$), concatenation ($L_1.L_2$), and Kleene closure (L^*).

Theorem 5.

The class of regular sets is closed under complementation. That is, if L is a regular set and $L \in \Sigma^*$, then $\sigma^* - L$ is a regular set.

Theorem 6.

The regular sets are closed under intersection.

Part IV

Computer Organization [3]

Chapter 15

Machine Instructions & Addressing Modes

15.1 Definitions

Definition 20 - Instruction Set.

Instruction set The vocabulary of commands understood by a given architecture.

Definition 21 - Stored Program Concept.

Stored-program concept The idea that instructions and data of many types can be stored in memory as numbers, leading to the stored program computer.

Definition 22 - Word.

Word The natural unit of access in a computer, usually a group of 32 bits; corresponds to the size of a register in the MIPS architecture.

Definition 23 - Registers.

Registers are limited number of special locations built directly in hardware. The size of a register in the MIPS architecture is 32 bits.

Is the size of register = word size always? **Interesting Point**:The word size of an Architecture is often (but not always!) defined by the Size of the General Purpose Registers.

Definition 24 - Data transfer Instructions.

Data transfer instruction A command that moves data between memory and registers.

Definition 25 - Address.

Address A value used to delineate the location of a specific data element within a memory array. To access a word in memory, the instruction must supply the memory **address**.

Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0.

Definition 26 - Base Register.

Definition 27 - Offset.

Definition 28 - Alignment Restriction.

Words must start at addresses that are multiples of 4(for MIPS). This requirement is called an **alignment restriction**.

Definition 29 - Little & Big Endian.

Computers divide into those that use the address of the leftmost or "**big end**"byte as the word address versus those that use the rightmost or "**little end**"byte.

Definition 30 - Instruction Format.

Instruction format A form of representation of an instruction composed of fields of binary numbers. **Example:** add $t0, $s1, $s2 t0 = 8(reg. no.) s0 = 16(reg.no)

0	17	18	8	0	32
000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
`op`	`rs`	`rt`	`rd`	`shamt`	`funct`

1. `op` Basic operation of the instruction, traditionally called the **opcode**.
2. `rs` The first register source operand.
3. `rt` The second register source operand.
4. `rd` The register destination operand. It gets the result of the operation.
5. `shamt` Shift amount. (Section 2.5 explains shift instructions and this term; it will not be used until then, and hence the field contains zero.)
6. `funct` Function. This field selects the specific variant of the operation in the op field and is sometimes called the function code.

Definition 31 - Opcode.

An **opcode** (operation code) is the portion of a machine language instruction that specifies the operation to be performed.

Definition 32 - Program Counter.

Program counter (PC) The register containing the address of the instruction in the program being executed.

Definition 33 - Addressing Modes.

Addressing mode One of several addressing regimes delimited by their varied use of operands and/or addresses.

15.2 Design Principles

1. **Simplicity favors regularity**.
2. **Samller is faster** A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther.
3. **Make common case fast** Constant operands occur frequently, and by including constants inside arithmetic instructions, they are much faster than if constants were loaded from memory.
4. **Good design demands good compromises**. Different kinds of instruction formats for different kinds of instructions.

15.3 Machine Instructions [3, p.67,90]

1. **Load** `lw`: copies data from memory to a register. **Format:** `lw` register, mem loc. of data(num,reg. where base add. is stored) item **Store** `sw`: copies data from register to memory. **Format:** `sw` register, mem loc. of data(num,reg. where base add. is stored)
2. **Add** `add`:
3. **Add Immediate** `addi`:
4. **Subtract** `sub`:

```
lw $t0,8($s3) # Temporary reg $t0 gets A[8]
```

Chapter 16

Pipelining

Definition 34 - Pipelining.

Pipelining is an implementation technique in which multiple instructions are overlapped in execution.

Pipelining doesn't decrease the time to finish a work but increases the throughput in a given time. MIPS instrructions takes 5 steps:

1. Fetch instruction from memory.
2. Read registers while decoding the instruction. The format of MIPS instructions allows reading and decoding to occur simultaneously.
3. Execute the operation or calculate an address.
4. Access an operand in data memory.
5. Write the result into a register.

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

Speedup due to pipelining $\approx$ No. of pipe stages. 5 stage piplelined process is 5 times faster. (Only for perfectly balanced processes).
But process involve overhead. So speedup will be quite less.

16.1 Designing Instruction sets for pipelining

1. All MIPS instructions are the same length. This restriction makes it much easier to fetch instructions in the first pipeline stage and to decode them in the second stage.
2. MIPS has only a few instruction formats, with the source register fields being located in the same place in each instruction. This symmetry means that the second stage can begin reading the register file at the same time that the hardware is determining what type of instruction was fetched.
3. Memory operands only appear in loads or stores in MIPS. This restriction means we can use the execute stage to calculate the memory address and then access memory in the following stage.
4. Operands must be aligned in memory. Hence, we need not worry about a single data transfer instruction requiring two data memory accesses; the requested data can be transferred between processor and memory in a single pipeline stage.

16.2 Pipeline Hazards

16.2.1 Structural Hazards

It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle.

16.2.2 Data Hazards

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete. **For example**, suppose we have an add instruction followed immediately by a subtract instruction that uses the sum ($s0):

```
add \$s0, \$t0, \$t1
sub \$t2, \$s0, \$t3
```

Without intervention, a data hazard could severely stall the pipeline.

Solution

Forwarding also called bypassing. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory.
Load-use data hazard A specific form of data hazard in which the data requested by a load instruction has not yet become available when it is requested. **Pipeline stall** Also called bubble. A stall initiated in order to resolve a hazard.
For **example** If first instruction were `load` instead of `add` then $s0 would not be available until 4^{th} stage resulting in a pipeline stall.

16.2.3 Hazards

Chapter 17

Arithmetic for Computers

17.1 Definitions

Definition 35 - LSB & MSB.

LSB : right-most bit .**MSB :** left-most bit.

Chapter 18

Speedup

Definition 36 - Amdahl's Law.

content...

Part V

First Order Logic

Chapter 19

Propositional Logic

19.1 Introduction

- Simple Statement: A simple statement is one that does not contain any other statement as a component.
  Ex., Fast foods tend to be unhealthy.
- Compound Statement: A compound statement is one that contains at least one simple statement as a component.
  Ex., Dianne Reeves sings jazz, and Christina Aguilera sings pop.
-

Operator	Name	Logical function	Used to translate
$\sim$	tilde	negation	not, it is not the case that
$\wedge$	dot	conjunction	and, also, moreover
$\vee$	wedge	disjunction	or, unless
$\rightarrow$	horseshoe	implication	if ... then ... , only if
$\leftrightarrow$	triple bar	equivalence	if and only if

Symbol	Words	Usage
∼	doesnot	Rolex does not make computers.
	not the case	It is not the case that Rolex makes computers.
	false that	It is false that Rolex makes computers.
∧	and	Tiff any sells jewelry, and Gucci sells cologne.
		Tiff any and Ben Bridge sell jewelry.
	but	Tiff any sells jewelry, but Gucci sells cologne.
	however	Tiff any sells jewelry; however, Gucci sells cologne.
→	if . . .,then . . . (S → P)	If Purdue raises tuition, then so does Notre Dame.
	. . . if . . . (P → S)	Notre Dame raises tuition if Purdue does.
	. . . only if . . . (S → P)	Purdue raises tuition only if Notre Dame does.
	. . . provided that . . . (P → S)	Cornell cuts enrollment provided that Brown does.
	. . . provided that . . . (P → S)	Cornell cuts enrollment on condition that Brown does.
	. . . provided that . . . (S → P)	BrownâĂŹs cutting enrollment implies that Cornell does.
∨	Either . . . or	Aspen allows snowboards or Telluride does.
		Either Aspen allows snowboards or Telluride does.
	Unless	Aspen allows snowboards unless Telluride does.
		Unless Aspen allows snowboards, Telluride does.
↔	. . . if and only if . . . (S → P)	JFK tightens security if and only if OâĂŹHare does.
	. . . sufficient and necessary condition that . . . (S → P)	JFKâĂŹs tightening security is a suffi cient and necessary condition for OâĂŹHareâĂŹs doing so.

- [1]

- The → symbol is also used to translate statements phrased in terms of sufficient conditions and necessary conditions. Event A is said to be a sufficient condition for event B whenever the occurrence of A is all that is required for the occurrence of B. On the other hand, event A is said to be a necessary condition for event B whenever B cannot occur without the occurrence of A. For example, having the flu is a sufficient condition for feeling miserable, whereas having air to breathe is a necessary condition for survival. Other things besides having the flu might cause a person to feel miserable, but that by itself is suffi cient; other things besides having air to breathe are required for survival, but without air survival is impossible. In other words, air is necessary.

- To translate statements involving sufficient and necessary conditions into symbolic form, place the statement that names the sufficient condition in the antecedent of the conditional and the statement that names the necessary condition in the consequent.

  1. HiltonâĂŹs opening a new hotel is a sufficient condition for MarriottâĂŹs doing so. H → M
  2. HiltonâĂŹs opening a new hotel is a necessary condition for MarriottâĂŹs doing so. M → H

-

Not either A or B.	∼(A ∨ B)
Either not A or not B.	∼A ∨ ∼B
Not both A and B.	∼(A ∧ B)
Both not A and not B.	∼A ∧ ∼B

19.2 Truth Table

- NEGATION

p	∼p
0	1
1	0

- AND

p	q	p∧q
0	0	0
0	1	0
1	0	0
1	1	1

[1] S - First Part, P - Second Part

- OR

p	q	p∨q
0	0	0
0	1	1
1	0	1
1	1	1

- IF

p	q	p→q
0	0	1
0	1	1
1	0	0
1	1	1

- IF AND ONLY IF

p	q	p↔q
0	0	1
0	1	0
1	0	0
1	1	1

- To find truth value of long propositions
  $(B \wedge C) \rightarrow (E \rightarrow A)$
  $(T \wedge T) \rightarrow (F \rightarrow T)$
  T → T
  (T)

-

Column under main operator	Statement classification
all true tautologous	(logically true)
all false	self-contradictory (logically false)
at least one true, at least one false	contingent

- If all Premises are true and conclusion is false, then argument is invalid.
- Assume argument invalid (true premises, false conclusion)
  1. Contradiction → Argument is valid
  2. No Contradiction → Argument is as assumed (i.e., invalid)
- Assume statements consistent (assume all of them true)
  1. Contradiction → Statements is consistent
  2. No Contradiction → Statements is as assumed (i.e., inconsistent)

19.3 Rules

Rules of Implication			
Name	Premises1	Premises2	Conclusion
Modus Ponens(MP)	$p \to q$	p	q
Modus Tollens(MT)	$p \to q$	$\sim q$	$\sim p$
Hypothetical Syllogism(HS)	$p \to q$	$q \to r$	$p \to r$
Disjunctive Syllogism(DS)	$p \vee q$	$\sim p$	q
Constructive Dilemma(CD)	$(p \to q) \wedge (r \to s)$	$p \vee r$	$q \vee s$
Destructive Dilemma(DD)	$(p \to q) \wedge (r \to s)$	$\sim q \vee \sim s$	$\sim p \vee \sim r$
Conjunction	p	q	$p \wedge q$
Addition	p		$p \vee q$
Simplification	$p \wedge q$		p

Rules of Replacement		
De-Morgan's Rule	$\sim(p \wedge q)$ $\sim(p \vee q)$	$(\sim p \vee \sim q)$ $(\sim p \wedge \sim q)$
Commutative	$(p \vee q)$ $(p \wedge q)$	$(q \vee p)$ $(q \wedge p)$
Associative	$p \vee (q \vee r)$ $p \wedge (q \wedge r)$	$(p \vee q) \vee r$ $(p \wedge q) \wedge r$
Distributive	$p \wedge (q \vee r)$ $p \vee (q \wedge r)$	$(p \wedge q) \vee (p \wedge r)$ $(p \vee q) \wedge (p \vee r)$
Double-Negation	p	$\sim\sim p$
Transposition	$p \to q$	$\sim q \to \sim p$
Exportation	$(p \wedge q) \to r$	$p \to (q \to r)$
Material Implication	$p \to q$	$\sim p \wedge q$
Tautology	p	$p \wedge p$ $p \vee p$
Material Equivalence	$p \leftrightarrow q$	$(p \to q) \wedge (q \to p)$ $(p \wedge q) \vee (\sim p \wedge \sim q)$

19.4 Conditional Proof

Conditional proof is a method for deriving a conditional statement (either the conclusion or some intermediate line) that off ers the usual advantage of being both shorter and simpler to use than the direct method.

Let us suppose, for a moment, that we do have A. We could then derive $B \wedge C$ from the first premise via modus ponens. Simplifying this expression we could derive B, and from this we could get $B \vee D$ via addition. E would then follow from the second premise via modus ponens. In other words, if we assume that we have A, we can get E. But this is exactly what the conclusion says. Thus, we have just proved that the conclusion follows from the premises.

1. $A \to (B \wedge C)$
2. $(B \vee D) \to E$ / $A \to E$
   - 3. A — ACP
   - 4. $B \wedge C$ — 1,3 MP
   - 5. B — 4 Simp
   - 6. $B \vee D$ — 5 Add
   - 7. E — 2,6 MP
8. $A \to E$ 3-7 CP

19.5 Indirect Proof

It consists of assuming the negation of the statement to be obtained, using this assumption to derive a contradiction, and then concluding that the original assumption is false. Th is last step, of course, establishes the truth of the statement to be obtained. Th e following proof sequence uses indirect proof to derive the conclusion:

1. $(A \vee B) \rightarrow (C \wedge D)$
2. $C \rightarrow \sim D$ / $\sim A$
3. A AIP
4. $A \vee B$ 3 Add
5. $C \wedge D$ 1,4 MP
6. C 5 Simp
7. $\sim D$ 2,6 MP
8. $D \wedge C$ 5 Com
9. D 8 Simp
10. $D \wedge \sim D$ 7,9 Conj
11. $\sim A$ 3-10 IP

Chapter 20

Predicate Logic

20.1 Introduction

Part VI

Probability

Syllabus: Conditional Probability; Mean, Median, Mode and Standard Deviation; Random Variables; Distributions; uniform, normal, exponential, Poisson, Binomial.

Chapter 21

Definitions

21.1 Definitions

1. **Probability Density Function**

$$\Pr[a \leq X \leq b] = \int_a^b f_X(x)\,dx$$

.

2. **Cumulative Distributive Function**

$$F_X(x) = \int_{-\infty}^{x} f_X(u)\,du$$

$$f_X(x) = \frac{d}{dx} F_X(x)$$

.

$$F_X(x) = \mathrm{P}(X \leq x)$$

$$\mathrm{P}(a < X \leq b) = F_X(b) - F_X(a)$$

3. **Chebysev's Inequality**

$$\Pr(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$$

where X be a random variable with finite expected value μ and finite non-zero variance σ^2. $k > 0$

4. Let $\bar{x}$ and s be the sample mean and sample standard deviation of the data set consisting of the data $x_1, \ldots, x_n$ where $s > 0$. Let

$$S_k = i, 1 \leq i \leq n : |x_i - \bar{x}| < ks$$

and let $N(S_k)$ be the number of elements in the set S_k . Then, for any $k \geq 1$,

$$\frac{N(S_k)}{n} \geq 1 - \frac{n-1}{nk^2} \geq 1 - \frac{1}{k^2}$$

5. **One sided Chebysev's Inequality**. For $k > 0$

$$\frac{N(k)}{n} \leq \frac{1}{1+k^2}$$

6. **Weak Law of Large Numbers**

$$\overline{X}_n = \frac{1}{n}(X_1 + \cdots + X_n)$$

converges to the expected value

$$\overline{X}_n \to \mu \qquad \text{for} \qquad n \to \infty$$

where $X_1, X_2, \ldots$ is an infinite sequence of independent and identically distributed random variables.

7. **Markov's Equality** If X is a random variable that takes only nonnegative values, then for any value $a > 0$

$$P(x \geq a) \leq \frac{E[X]}{a}$$

8. **Chebysev's Inequality** If X is a random variable with mean μ and variance σ^2 , then for any value $k > 0$

$$P(|X - \mu|) \leq \frac{\sigma^2}{k^2}$$

$$P(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$$

9. **Baye's Theorem** Assume come an event E to happen with A as well as B.

$$P(E) = P(E \cap A) + P(E \cap B) \tag{21.1}$$

$$= P(A)P\left(\frac{E}{A}\right) + P(B)P\left(\frac{E}{B}\right) \tag{21.2}$$

Given that E has already happened:

$$P\left(\frac{A}{E}\right) = \frac{P(A \cap E)}{P(E)} \tag{21.3}$$

$$= \frac{P(A)P\left(\frac{E}{A}\right)}{P(A)P\left(\frac{E}{A}\right) + P(B)P\left(\frac{E}{B}\right)} \tag{21.4}$$

Chapter 22

Probability

22.1 Terms

- **Independent Events:** Since $P(E|F) = \frac{P(E \cap F)}{P(F)}$, we see that E is independent of F if

$$P(E \cap F) = P(E)P(F)$$

- The three events E, F, and G are said to be independent if

$$P(E \cap F \cap G) = P(E)P(F)P(G)$$
$$P(E \cap F) = P(E)P(F)$$
$$P(E \cap G) = P(E)P(G)$$
$$P(F \cap G) = P(F)P(G)$$

22.2 Propositions

- If E and F are independent, then so are E and F^{c} .
- **Mean :** X = Distribution, $P(X_i)$ = Probability of X_i, X_i is the distribution

$$E[X] = \sum_{i=0}^{n} X_i P(X_i)$$

- **Median :**
- **Mode :**
- **Standard Deviation :**

22.3 Conditional Probability

- Conditional probability of E given that F has occurred, is denoted by $P(E|F)$
-

$$P(E|F) = \frac{P(E \cap F)}{P(F)}.$$

Defined only when P(F) > 0 and hence P(E|F) is defined only when P(F) > 0.

- Suppose that one rolls a pair of dice. The sample space S of this experiment can be taken to be the following set of 36 outcomes

$$S = (i, j), i = 1, 2, 3, 4, 5, 6, j = 1, 2, 3, 4, 5, 6$$

where we say that the outcome is (i, j) if the first die lands on side i and the second on side j.
Suppose further that we observe that the first die lands on side 3. Then, given this information, what is the probability that the sum of the two dice equals 8?
Answer: Given that the initial die is a 3, there can be at most 6 possible outcomes of our experiment, namely, (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), and (3, 6). In addition, because each of these outcomes originally had the same probability of occurring, they should still have equal probabilities. That is, given that the first die is a 3, then the (conditional) probability of each of the outcomes (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6) is $\frac{1}{6}$, whereas the (conditional) probability of the other 30 points in the sample space is 0. Hence, the desired probability will be $\frac{1}{6}$.

22.3.1 Bayes Formula

- Let E and F be events. We may express E as

$$E = E \cap F \cup E \cap F^c$$

for, in order for a point to be in E, it must either be in both E and F or be in E but not in F.

- As $E \cap F$ & $E \cap F^c$ are clearly mutually exclusive,

$$\begin{aligned} P(E) &= P(E \cap F) + P(E \cap F^c) && (22.1)\\ &= P(E|F)P(F) + P(E \cap F^c)P(F^c) && (22.2)\\ &= P(E|F)P(F) + P(E \cap F^c)[1 - P(F)] && (22.3) \end{aligned}$$

22.4 Mean, Median, Mode and Standard Deviation

- Properties of Mean:
  1. General Formula for continuous distributions:
$$E[X] = \int_{-\infty}^{\infty} xf(x)dx$$
  2. General Formula for continuous distributions:
$$E[g(X)] = \int_{-\infty}^{\infty} g(x)f(x)dx$$
  3. $E[c] = c$ where c is a constant
  4. Linearity: $E[X] \leq E[Y]$
  5. General Linearity: $E[aX + bY + c] = aE[X] + bE[Y] + c$
  6. $$E[X|Y = y] = \sum_x x.P(X = x|Y = y)$$
  7. $E[X] = E[E[X|Y]]$

8. $$|E[X]| \leq E[|X|]$$
9. Cov(X, Y) = $E[XY] - E[X]E[Y]$
10. if Cov(X, Y) = 0: X & Y are are said to be uncorrelated (independent variables are a notable case of uncorrelated variables).
11. Variance = $E\left[X^2\right] - (E[X])^2$

- Properties of Median:
  1. Value of m for discrete distributions:
  $$P(X \geq m) \geq \frac{1}{2}$$
  $$P(X \leq m) \geq \frac{1}{2}$$
  2. Value of m for continuous distributions:
  $$\int_{(-\infty,m]} dF(x) \geq \frac{1}{2}$$
  $$\int_{(m,\infty]} dF(x) \geq \frac{1}{2}$$
  where $F(x)$ is the cumulative distriution function.
  3. All three measures have the following property: If the random variable (or each value from the sample) is subjected to the linear or affine transformation which replaces X by aX+b, so are the mean, median and mode.
  4. However, if there is an arbitrary monotonic transformation, only the median follows; for example, if X is replaced by exp(X), the median changes from m to exp(m) but the mean and mode won't.
  5. In continuous unimodal distributions the median lies, as a rule of thumb, between the mean and the mode, about one third of the way going from mean to mode. In a formula, median $\approx$ (2 $\times$ mean + mode)/3. This rule, due to Karl Pearson, often applies to slightly non-symmetric distributions that resemble a normal distribution, but it is not always true and in general the three statistics can appear in any order.
  6. or unimodal distributions, the mode is within $\sqrt{3}$ standard deviations of the mean, and the root mean square deviation about the mode is between the standard deviation and twice the standard deviation.

- Properties of Standard Deviation:
  1. Represented by σ.
  2. $$\sigma = \sqrt{E[(X-\mu)^2]} \tag{22.4}$$
  $$= \sqrt{E[X^2] - (E[X])^2} \tag{22.5}$$
  3. σ^2 = Variance
  4. **Continuous Random Variable:** The standard deviation of a continuous real-valued random variable X with probability density function p(x) is:
  $$\sigma = \sqrt{\int_{\mathbf{X}} (x-\mu)^2 p(x)dx}, \quad \text{where} \quad \mu = \int_{\mathbf{X}} xp(x)dx$$

- **Discrete Raqndom Variable:** In the case where X takes random values from a finite data set $x_1, x_2, \ldots, x_N$, let x_1 have probability p_1, x_2 have probability $p_2, \ldots, x_N$ have probability p_N. In this case, the standard deviation will be

$$\sigma = \sqrt{\sum_{i=1}^{N} p_i (x_i - \mu)^2}, \quad \text{where} \quad \mu = \sum_{i=1}^{N} p_i x_i$$

22.5 Distributions

22.5.1 Bernoulli

- Suppose that a trial, or an experiment, whose outcome can be classified as either a "success" or as a "failure" is performed. If we let X = 1 when the outcome is a success and X = 0 when it is a failure, then the probability mass function of X is given by

$$P(X = 0) = 1 - p \tag{22.6}$$
$$P(X = 1) = p \tag{22.7}$$

  where p, $0 \leq p \leq 1$, is the probability that the trial is a "success."

22.5.2 HyperGeometric

A bin contains N + M batteries, of which N are of acceptable quality and the other M are defective. A sample of size n is to be randomly chosen (without replacements) in the sense that the set of sampled batteries is equally likely to be any of the $\binom{N+M}{n}$ subsets of size n. If we let X denote the number of acceptable batteries in the sample, then

$$P(X = i) = \frac{\binom{N}{i}\binom{M}{n-i}}{\binom{N+M}{n}}$$

22.5.3 Uniform

$$f(x) = \begin{cases} \frac{1}{\beta - \alpha} & \text{if} \quad \alpha \leq x \leq \beta \\ 0 & \text{otherwise} \end{cases}$$

22.5.4 Normal

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

22.5.5 Exponential

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

22.5.6 Poisson

$$P(X = i) = e^{-\lambda} \frac{\lambda^i}{i!}$$

22.5.7 Binomial

$$P(X = i) = \binom{n}{k} p^k (1-p)^{n-k} \qquad i \leq n$$

22.5.8 Summary

Table 22.1: Probability Formula Table

Name	Notation	Mean	Median	Mode	Variance
Bernoulli		p			
Binomial	$(B(n, p))$	np	$\lfloor np \rfloor$ $\lceil np \rceil$	$\lfloor (n+1)p \rfloor$ $\lfloor (n+1)p - 1 \rfloor$	np(1 - p)
Exponential					
Normal					
Poisson					
Bernoulli					

Table 22.2: Probability Formula Table

Name	Probability Generating Function(PMF)	Probability Generating Function(PGF)	Moment Generating Function(MGF)	Cumulative Distributive Function(CDF)
Bernoulli				
Binomial		$G(z) = [(1-p) + pz]^n$	$(1 - p + pe^t)^n$	$(1 - p + pe^{it})^n$
Exponential				
Normal				
Poisson				

Part VII

HTML

Chapter 23

Basic Commands

23.1 Basics

1. The DOCTYPE declaration defines the document type.
2. The text between <html> and </html> describes the web page.
3. The text between <body> and </body> is the visible page content.
4. The text between <h1> and </h1> is displayed as a heading.
5. The text between and is displayed as a paragraph.
6. HTML stands for **Hyper Text Markup Language**.
7. HTML is a markup language.
8. HTML tags are not case sensitive.
9. HTML elements with no content are called empty elements.
10. HTML elements can have attributes.
11. Attributes provide additional information about an element.
12. Attributes are always specified in the start tag.
13. Attributes come in name/value pairs like: name="value".
14. Attribute names and attribute values are case-insensitive(lower case recommended).
15. <hr> tag creates a horizontal line in an HTML page.

23.2 Tags

1. <h1> </h1> ... <h6> </h6>
2.
3. <a>
4. <img>
5.
 is an empty element without a closing tag

23.3 Tags Reference

Tag	Description
Basic	
<!DOCTYPE>	Defines the document type
<html>	Defines an HTML document
<title>	Defines a title for the document
<body>	Defines the document's body
<h1> to <h6>	Defines HTML headings
<p>	Defines a paragraph
 	Inserts a single line break
<hr>	Defines a thematic change in the content
<!–...–>	Defines a comment
Formatting	
<acronym>	**Not supported in HTML5.** Use <abbr> instead. Defines an acronym
<abbr>	Defines an abbreviation
<address>	Defines contact information for the author/owner of a document/article
<b>	Defines bold text
<bdi>	New Isolates a part of text that might be formatted in a different direction from other text outside it
<bdo>	Overrides the current text direction
<big>	**Not supported in HTML5.** Use CSS instead. Defines big text
<blockquote>	Defines a section that is quoted from another source
	Not supported in HTML5. Use CSS instead. Defines centered text
<cite>	Defines the title of a work
<code>	Defines a piece of computer code
<del>	Defines text that has been deleted from a document
<dfn>	Defines a definition term
	Defines emphasized text
<font>	**Not supported in HTML5.** Use CSS instead. Defines font, color, and size for text
<i>	Defines a part of text in an alternate voice or mood
<ins>	Defines a text that has been inserted into a document
<kbd>	Defines keyboard input
<mark>	New Defines marked/highlighted text
<meter>	New Defines a scalar measurement within a known range (a gauge)
<pre>	Defines preformatted text
<progress>	New Represents the progress of a task
<q>	Defines a short quotation
<rp>	New Defines what to show in browsers that do not support ruby annotations
<rt>	New Defines an explanation/pronunciation of characters (for East Asian typography)
<ruby>	New Defines a ruby annotation (for East Asian typography)
<s>	Defines text that is no longer correct
<samp>	Defines sample output from a computer program
<small>	Defines smaller text

<strike>	**Not supported in HTML5.** Use <del> instead. Defines strikethrough text
	Defines important text
<sub>	Defines subscripted text
<sup>	Defines superscripted text
<time>	New Defines a date/time
<tt>	**Not supported in HTML5.** Use CSS instead. Defines teletype text
<u>	Defines text that should be stylistically different from normal text
<var>	Defines a variable
<wbr>	New Defines a possible line-break
Forms	
<form>	Defines an HTML form for user input
<input>	Defines an input control
<textarea>	Defines a multiline input control (text area)
<button>	Defines a clickable button
<select>	Defines a drop-down list
<optgroup>	Defines a group of related options in a drop-down list
<option>	Defines an option in a drop-down list
<label>	Defines a label for an <input> element
<fieldset>	Groups related elements in a form
<legend>	Defines a caption for a <fieldset> element
<datalist>	New Specifies a list of pre-defined options for input controls
<keygen>	New Defines a key-pair generator field (for forms)
<output>	New Defines the result of a calculation
Frames	
<frame>	**Not supported in HTML5.** Defines a window (a frame) in a frameset
<frameset>	Not supported in HTML5. Defines a set of frames
<noframes>	**Not supported in HTML5.** Defines an alternate content for users that do not support frames
<iframe>	Defines an inline frame
Images	
<img>	Defines an image
<map>	Defines a client-side image-map
<area>	Defines an area inside an image-map
<canvas>	New Used to draw graphics, on the fly, via scripting (usually JavaScript)
<figcaption>	New Defines a caption for a <figure> element
<figure>	New Specifies self-contained content Audio/Video
<audio>	New Defines sound content
<source>	New Defines multiple media resources for media elements (<video> and <audio>)
<track>	New Defines text tracks for media elements (<video> and <audio>)
<video>	New Defines a video or movie Links
<a>	Defines a hyperlink
<link>	Defines the relationship between a document and an external resource (most used to link to style sheets)
<nav>	New Defines navigation links Lists

	Defines an unordered list
	Defines an ordered list
	Defines a list item
<dir>	**Not supported in HTML5.** Use instead. Defines a directory list
<dl>	Defines a description list
<dt>	Defines a term/name in a description list
<dd>	Defines a description of a term/name in a description list
<menu>	Defines a list/menu of commands
<command>	New Defines a command button that a user can invoke
Tables	
<table>	Defines a table
<caption>	Defines a table caption
<th>	Defines a header cell in a table
<tr>	Defines a row in a table
<td>	Defines a cell in a table
<thead>	Groups the header content in a table
<tbody>	Groups the body content in a table
<tfoot>	Groups the footer content in a table
<col>	Specifies column properties for each column within a <colgroup> element
<colgroup>	Specifies a group of one or more columns in a table for formatting
Style/Sections	
<style>	Defines style information for a document
	Defines a section in a document
	Defines a section in a document
<header>	New Defines a header for a document or section
<footer>	New Defines a footer for a document or section
<section>	New Defines a section in a document
<article>	New Defines an article
<aside>	New Defines content aside from the page content
<details>	New Defines additional details that the user can view or hide
<dialog>	New Defines a dialog box or window
<summary>	New Defines a visible heading for a <details> element
Meta Info	
<head>	Defines information about the document
<meta>	Defines metadata about an HTML document
<base>	Specifies the base URL/target for all relative URLs in a document
<basefont>	**Not supported in HTML5.** Use CSS instead. Specifies a default color, size, and font for all text in a document
Programming	
<script>	Defines a client-side script
<noscript>	Defines an alternate content for users that do not support client-side scripts
<applet>	**Not supported in HTML5.** Use <object> instead. Defines an embedded applet
<embed>	New Defines a container for an external (non-HTML) application
<object>	Defines an embedded object
<param>	Defines a parameter for an object

Part VIII

Numerical Analysis

Chapter 24

Numerical solutions to non-algebraic linear equations

24.1 Bisection Method

Theorem 7.

If a function $f(x)$ is continuous between a and b, and f(a) and f(b) are of opposite signs, then $\exists$ atleast one root between a and b.

1. Choose two real numbers a and b such that $f(a)$ and $f(b) < 0$
2. Set $x_r = \frac{(a+b)}{2}$
3. (a) If $f(a)f(x_r) < 0$, the root lies in the interval (a, x_r). Then set $b = x_r$ and go to step 2 above.
   (b) If $f(a)f(x_r) > 0$, the root lies in the interval (x_r, b). Then set $a = x_r$ and go to step 2 above.
   (c) If $f(a)f(x_r) = 0$, it means that x_r is the root of the equation and the computation can be terminated.

Important Points:

1. Percentage error ϵ_r is defined as

$$\epsilon_r = \left| \frac{x_r^{'} - x_r}{x_r^{'}} \right| \times 100\% \tag{24.1}$$

   where $x_r^{'}$ is the new computed value of x_r

2. To find the number of iterations required for achieving particular accuracy ϵ is

$$n \geq \frac{\log_e \left(\frac{|b-a|}{\epsilon} \right)}{\log_e 2} \tag{24.2}$$

24.2 Newton-Raphson Method

Let x_0 be the approximate root of $f(x) = 0$. Let $x_1 = x_0 + h$ be the correct root so that $f(x_1) = 0$. Expanding $f(x_0 + h)$ by taylor series we get,

$$f(x_0 + h) = f(x_0) + hf^{'}(x_0) + \frac{h^2}{2!}f^{''}(x_0) + \ldots = 0$$

Neglecting higher order terms we get

$$f(x_0) + hf^{'}(x_0) = 0 \qquad h = -\frac{f(x_0)}{f^{'}(x_0)} \qquad x_1 = x_0 - \frac{f(x_0)}{f^{'}(x_0)}$$

$$x_{n+1} = x_n - \frac{f(x_n)}{f^{'}(x_n)} \tag{24.3}$$

24.3 is called **Newton Raphson Formula**. This methods assumes that the function $f(x)$ has $f^{'}(x)$ & $f^{''}(x)$ existing. For $f(x)$ having multiple roots method converges slowly.

$$\epsilon_{n+1} \approx \frac{1}{2}\epsilon_n^2 \frac{f^{''}(\xi)}{f^{'}(\xi)} \tag{24.4}$$

where ξ is the exact value of the root of $f(x)$.

24.3 Secant Method

Also called **Modified Newton's Method**. In newton's method it is not always possible that $f^{'}(x)$ exists. So we use **Mean Value Theorem** :

Check for theorem's name

Theorem 8.

If a function $f(x)$ is continuous on the closed interval $[a, b]$, where $a < b$, and differentiable on the open interval (a, b), then $\exists$ a point c in (a, b) such that

$$f^{'}(c) = \frac{f(b) - f(a)}{b - a}$$

Using 8 we replace $f^{'}(x_i)$ by $\frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$ and obtain the formula:

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})} \tag{24.5}$$

Chapter 25

Numerical Integration

25.1 Introduction

Numerical Integration over $[a, b]$ on $f(x)$:

$$I = \int_a^b f(x)dx \tag{25.1}$$

For numerical integration $f(x)$ in 25.1 is replaced by an interpolation polynomial $\phi(x)$ & obtains on integration value of the definite integral. Here we use Newton's forward difference polynomial . Let the interval $[a, b]$ be divided into n equal sub-intervals such that $a = x_0 < x_1 < \ldots < x_n = b$. $x_n = x_0 + nh$.

Check for Newton's forward difference polynomial formula

$$I = \int_{x_0}^{x_n} ydx \tag{25.2}$$

$$I = \int_{x_0}^{x_n} \left[y_0 + p\triangle y_0 + \frac{p(p-1)}{2}\triangle^2 y_0 + \frac{p(p-1)(p-2)}{6}\triangle^3 y_0 + \ldots \right] dx \tag{25.3a}$$

$$I = h\int_0^n \left[y_0 + p\triangle y_0 + \frac{p(p-1)}{2}\triangle^2 y_0 + \frac{p(p-1)(p-2)}{6}\triangle^3 y_0 + \ldots \right] dp \tag{25.3b}$$

$$\int_{x_0}^{x_n} ydx = nh\left[y_0 + \frac{n}{2}\triangle y_0 + \frac{n(2n-3)}{12}\triangle^2 y_0 + \frac{n(n-2)^2}{24}\triangle^3 y_0 + \ldots \right] \tag{25.3c}$$

$x = x_0 + ph$, $dx = hdp$

25.2 Trapezoidal Rule

Put $n = 1$ in 25.3c we get(all other differnces would become zero),

$$\int_{x_0}^{x_1} ydx = h\left[y_0 + \frac{1}{2}\triangle y_0 \right] = h\left[y_0 + \frac{1}{2}(y_1 - y_0) \right] = \frac{h}{2}(y_0 + y_1) \tag{25.4a}$$

For interval $[x_1, x_2]$

$$\int_{x_1}^{x_2} ydx = \frac{h}{2}(y_1 + y_2) \tag{25.4b}$$

For interval $[x_{n-1}, x_n]$

$$\int_{x_{n-1}}^{x_n} ydx = \frac{h}{2}(y_{n-1} + y_n) \tag{25.4c}$$

Summing up,

$$\int_{x_0}^{x_n} ydx = \frac{h}{2}\left[y_0 + 2(y_1 + y_2 + \ldots + y_{n-1}) + y_n\right] \tag{25.4d}$$

Total Error **E**:

$$E = \frac{-1}{12}h^3 n y''(\bar{x}) = -\frac{(b-a)}{12}h^2 y''(\bar{x}) \tag{25.5}$$

$nh = b - a$. 25.5 is called **Error of Trapezoidal Rule**.

25.3 Simpson's 1/3 Rule

Put $n = 2$ in 25.3c we get(all other differnces would become zero),

$$\int_{x_0}^{x_2} ydx = 2h\left[y_0 + \triangle y_0 + \frac{1}{6}\triangle^2 y_0\right] = \frac{h}{3}(y_0 + 4y_1 + y_2) \tag{25.6a}$$

$$\int_{x_2}^{x_4} ydx = \frac{h}{3}(y_2 + 4y_3 + y_4) \tag{25.6b}$$

$$\int_{x_{n-2}}^{x_n} ydx = \frac{h}{3}(y_{n-2} + 4y_{n-1} + y_n) \tag{25.6c}$$

$$\int_{x_0}^{x_n} ydx = \frac{h}{3}\left[y_0 + 4(y_1 + y_3 + y_5 + \ldots + y_{n-1}) + 2(y_2 + y_4 + y_6 + \ldots + y_{n-2}) + y_n\right] \tag{25.6d}$$

25.6d is called **Simpson's 1/3 rule.**
Total Error E:

$$E = -\frac{b-a}{180}h^4 y^4(\bar{x}) \tag{25.7}$$

25.7 is the Error **E** for Simpson's 1/3 Rule.

25.4 Simpson's 3/8 Rule

Put $n = 3$ in 25.3c we get(all other differnces would become zero),

$$\int_{x_0}^{x_3} ydx = 3h\left[y_0 + \frac{3}{2}\triangle y_0 + \frac{3}{4}\triangle^2 y_0 + \frac{1}{8}\triangle^3 y_0\right] = \frac{3h}{8}(y_0 + 3y_1 + 3y_2 + y_3) \tag{25.8a}$$

$$\int_{x_3}^{x_6} ydx = \frac{3h}{8}(y_3 + 3y_4 + 3y_5 + y_6) \tag{25.8b}$$

$$\int_{x_0}^{x_n} ydx = \frac{3h}{8}\left[y_0 + 3y_1 + 3y_2 + 2y_3 + 3y_4 + 3y_5 + 2y_6 + \ldots + 2y_{n-3} + 3y_{n-2} + 3y_{n-1} + y_n\right] \tag{25.8c}$$

25.8c is called **Simpson's 3/8 rule.**
Total Error E:

$$E = -\frac{3}{80}h^5 y^4(\bar{x}) \tag{25.9}$$

25.9 is the Error **E** for Simpson's 3/8 Rule .

Check the formula for Error of Simpson's 3/8 rule

Chapter 26

LU decomposition for system of linear equations

26.1 Introduction

LU factorization without pivoting:

$$A = LU$$

where A is any matrix, L is any lower triangular matrix, U is any upper triangular matrix.

26.2 Factorizing A as L and U

$\text{A} = \begin{bmatrix} a_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, $\text{L} = \begin{bmatrix} 1 & 0 \\ L_{21} & L_{22} \end{bmatrix}$, $\text{U} = \begin{bmatrix} u_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}$

$$\begin{bmatrix} a_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} u_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} = \begin{bmatrix} u_{11} & U_{12} \\ u_{11} + L_{21} & L_{21}U_{12} + L_{22}U_{22} \end{bmatrix}$$

we get

$$u_{11} = a_{11}, U_{12} = A_{12}, L_{21} = \frac{A_{21}}{a_{11}}$$

26.2.1 Example

$$\text{A} = \begin{bmatrix} 8 & 2 & 9 \\ 4 & 9 & 4 \\ 6 & 7 & 9 \end{bmatrix}$$

Split A as $\mathbf{A} = \mathbf{LU}$ where L is the lower triangular matrix & U is theupper triangular matrix.

$$\text{A} = \begin{bmatrix} 8 & 2 & 9 \\ 4 & 9 & 4 \\ 6 & 7 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$
$$= \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21}u_{11} & l_{21}u_{12} + u_{22} & l_{21}u_{13} + u_{23} \\ l_{31}u_{11} & l_{31}u_{12} + l_{32}u_{22} & l_{31}u_{13} + l_{32}u_{23} + l_{33}u_{33} \end{bmatrix}$$

Obtained Values:

$$u_{11} = 8, u_{12} = 2, u_{13} = 9$$

To obtain other values:

$$l_{21}u_{11} = 4 \implies l_{21} = \frac{4}{8} = \frac{1}{2} \qquad (\because u_{11} = 8)$$

$$9 = l_{21}u_{12} + u_{22} \implies 9 = \frac{1}{2} * 2 + u_{22} \implies u_{22} = 8$$

$$4 = l_{21}u_{13} + u_{23} \implies 4 = \frac{1}{2} * 9 + u_{23} \implies u_{23} = -\frac{1}{2}$$

$$l_{31}u_{11} = 6 \implies l_{31} = \frac{6}{8} = \frac{3}{4} \qquad (\because u_{11} = 8)$$

$$7 = l_{31}u_{12} + l_{32}u_{22} \implies 7 = \frac{3}{4} * 2 + l_{32} * 8 \implies l_{32} = \frac{11}{16}$$

$$9 = l_{31}u_{13}+l_{32}u_{23}+l_{33}u_{33} \implies 9 = \frac{3}{4}*9+\frac{11}{16}*\left(-\frac{1}{2}\right)+u_{33} \implies u_{33} = 9-\frac{27}{4}+\frac{11}{32} \implies u_{33} = \frac{83}{32}$$

Therefore,

$$\begin{bmatrix} 8 & 2 & 9 \\ 4 & 9 & 4 \\ 6 & 7 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{3}{4} & \frac{11}{16} & 1 \end{bmatrix} \begin{bmatrix} 8 & 2 & 9 \\ 0 & 8 & -\frac{1}{2} \\ 0 & 0 & \frac{83}{32} \end{bmatrix}$$

Every non-singular A cannot be factorized as A= LU. Example: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 2 \\ 0 & 1 & -1 \end{bmatrix}$

26.3 Algorithm For solving

1. $\mathsf{AX} = \mathsf{b} \implies \mathsf{LUX} = \mathsf{b}$. Solve for L and U using above algorithm.
2. $\mathsf{LZ} = \mathsf{b}$ where Z = $\begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$
3. $\mathsf{UX} = \mathsf{Z}$ where X = $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

Part IX

XML

Chapter 27

Basic Information

27.1 Basics

1. XML stands for eXtensible Markup Language.
2. XML is designed to transport and store data.
3. XML is important to know, and very easy to learn.
4. XML was designed to carry data, not to display data.
5. XML tags are not predefined. You must define your own tags.
6. XML is designed to be self-descriptive.
7. XML is a W3C Recommendation.

Difference between XML and HTML:

1. XML was designed to transport and store data, with focus on what data is.
2. HTML was designed to display data, with focus on how data looks.

Use of XML:

1. If you need to display dynamic data in your HTML document, it will take a lot of work to edit the HTML each time the data changes.
2. With XML, data can be stored in separate XML files. This way you can concentrate on using HTML/CSS for display and layout, and be sure that changes in the underlying data will not require any changes to the HTML.
3. With a few lines of JavaScript code, you can read an external XML file and update the data content of your web page.
4. Different applications can access your data, not only in HTML pages, but also from XML data sources.
5. With XML, your data can be available to all kinds of "reading machines" (Handheld computers, voice machines, news feeds, etc), and make it more available for blind people, or people with other disabilities.

27.2 Rules for XML Docs

1. In XML, it is illegal to omit the closing tag. All elements must have a closing tag.
2. XML tags are case sensitive. The tag <Letter> is different from the tag <letter>.
3. In XML, all elements must be properly nested within each other.
4. XML documents must contain one element that is the parent of all other elements. This element is called the root element.
5. XML elements can have attributes in name/value pairs just like in HTML. In XML, the attribute values must always be quoted.
6. Some characters have a special meaning in XML. If you place a character like "<" inside an XML element, it will generate an error because the parser interprets it as the start of a new element.
7. The syntax for writing comments in XML is similar to that of HTML. <!– This is a comment –>
8. With XML, the white-space in a document is not truncated.
9. XML stores a new line as LF(Line Feed). Special Pre-Defined Symbols in XML:

<	<	less than
>	>	greater than
&	&	ampersand
'	’	apostrophe
"	"	quotation mark

10. XML elements can be extended to carry more information without breaking applications.

27.3 XML Elements

An XML element is everything from (including) the element's start tag to (including) the element's end tag.

An element can contain:

1. other elements
2. text
3. attributes
4. or a mix of all of the above...

Naming Rules:
XML elements must follow these naming rules:

1. Names can contain letters, numbers, and other characters
2. Names cannot start with a number or punctuation character
3. Names cannot start with the letters xml (or XML, or Xml, etc)
4. Names cannot contain spaces Any name can be used, no words are reserved.

27.4 XML Attributes

1. Attributes often provide information that is not a part of the data.
2. Attribute values must always be quoted. Either single or double quotes can be used.
3. Some of the problems with using attributes are:
   (a) attributes cannot contain multiple values (elements can)
   (b) attributes cannot contain tree structures (elements can)
   (c) attributes are not easily expandable (for future changes) Attributes are difficult to read and maintain. Use elements for data. Use attributes for information that is not relevant to the data.
4. Sometimes ID references are assigned to elements. These IDs can be used to identify XML elements in much the same way as the id attribute in HTML.

```
<messages>
  <note id="501">
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
  </note>
  <note id="502">
    <to>Jani</to>
    <from>Tove</from>
    <heading>Re: Reminder</heading>
    <body>I will not</body>
  </note>
</messages>
```

5. DTD - Document Type Definition
6. The purpose of a DTD is to define the structure of an XML document. It defines the structure with a list of legal elements.
7. A "Valid" XML document is a "Well Formed" XML document, which also conforms to the rules of a Document Type Definition (DTD).
8. XML with correct syntax is "Well Formed" XML. XML validated against a DTD is "Valid" XML.
9. XSLT is the recommended style sheet language of XML. XSLT (eXtensible Stylesheet Language Transformations) is far more sophisticated than CSS. XSLT can be used to transform XML into HTML, before it is displayed by a browser

Part X

Computer Networks

Chapter 28

Network Security

Definition 37 - Kerckhoff's principle.

Kerckhoff's principle: All algorithms must be public; only the keys are secret.

28.0.1 Substitution Ciphers

Ceaser Cipher: Shift of letters by k position. *KEY:* k
Transposition Cipher: Reordering of letters without disguising them. *KEY:* Word or phrase not containing any repeated letters.
One-time Pad:
Quantum Cryptography:
Cryptographic Principles:

1. Messages must contain some redundancy.
2. Some method is needed to foil replay attacks.

28.1 Public Key Algorithms

28.1.1 Diffie-Hellman Key Exchange

28.1.2 RSA

Requirements:

1. Choose two large primes, p and q (typically 1024 bits).
2. Compute $n = p \times q$ and $z = (p-1) \times (q-1)$.
3. Choose a number relatively prime to z and call it d.
4. Find e such that $e \times d = 1 \bmod z$.

Steps:

1. Divide the plaintext (regarded as a bit string) into blocks, so that each plaintext message, P, falls in the interval $0 \leq P < n$.
2. Do that by grouping the plaintext into blocks of k bits, where k is the largest integer for which $2^k < n$ is true.

3. To encrypt a message, P, compute $C = P^e(\bmod n)$. To decrypt C, compute $P = C^d(\bmod n)$.
4. The public key consists of the pair (e, n) and the private key consists of (d, n).

28.1.3 Knapsack Algorithm

28.1.4 El Gamal Algorithm

28.2 Digital Signatures

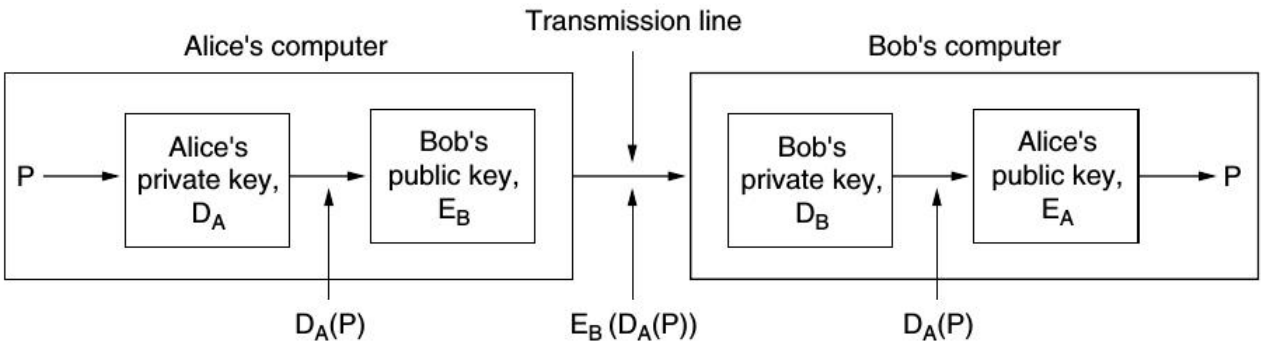
28.2.1 Symmetric Key Signatures

A central authority that knows everything and whom everyone trusts, say, Big Brother (BB). Each user then chooses a secret key and carries it by hand to BB's office.

1. When Alice wants to send a signed plaintext message, P, to her banker, Bob, she generates $K_A(B, R_A, t, P)$, where B is Bob's identity, R_A is a random number chosen by Alice, t is a timestamp to ensure freshness, and $K_A(B, R_A, t, P)$ is the message encrypted with her key, K_A .
2. BB sees that the message is from Alice, decrypts it, and sends a message to Bob as shown.
3. The message to Bob contains the plaintext of Alice's message and also the signed message $K_{BB}(A, t, P)$. Bob now carries out Alice's request.
4. To guard against instant replay attacks, Bob just checks the R_A of every incoming message to see if such a message has been received from Alice in the past hour.

28.2.2 Public Key Signatures

Figure 28.1: Digital Signatures using public key



Problems:

1. If D_A is changed, then applying old key doesnot lead to message P which is a porblem.
2. If D_A is secret, then Bob can prove any forgery case against him. If D_A is public, then problem arises.

Digital Signature Standard (DSS) used a variant of El Gamal encrytpion with 1024 bit keys.

28.2.3 Message Digests

Authentication scheme that does not require encrypting the entire message.
This scheme is based on the idea of a one-way hash function that takes an arbitrarily long piece of plaintext and from it computes a fixed-length bit string.
This hash function, *MD*, often called a message digest, has four important properties:

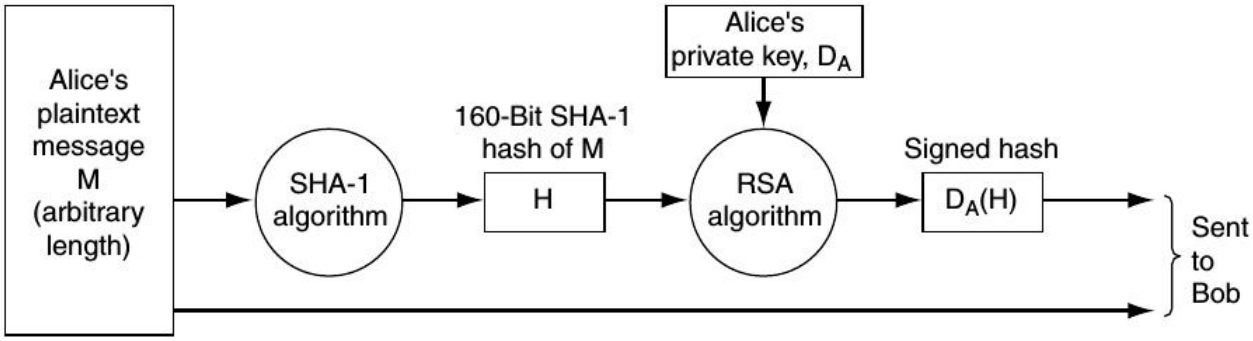
1. Given P, it is easy to compute $MD(P)$.
2. Given $MD(P)$, it is effectively impossible to find P.
3. Given P, no one can find P' such that $MD(P') = MD(P)$.
4. A change to the input of even 1 bit produces a very different output.

Message digests save encryption time and message transport costs. In the Fig 28.1 Instead, of signing P with $K_{BB}(A, t, P)$, BB now computes the message digest by applying MD to P, yielding $MD(P)$. BB then encloses $K_{BB}(A, t, MD(P))$ as the fifth item in the list encrypted with KB that is sent to Bob, instead of $K_{BB}(A, t, P)$.
In public key cryptosystems: Here, Alice first computes the message digest of her plaintext. She then signs the message digest and sends both the signed digest and the plaintext to Bob. If Trudy replaces P along the way, Bob will see this when he computes $MD(P)$.

SHA-1 & SHA-2

SHA: Secure Hash Algorithm.
SHA-1: It processes input data in 512-bit blocks, and it generates a 160-bit message digest.

Figure 28.2: SHA-1 algorithm from Alice to Bob



After receiving the message, Bob computes the SHA-1 hash himself and also applies AliceâĂŹs public key to the signed hash to get the original hash, H. If the two agree, the message is considered valid. Since there is no way for Trudy to modify the (plaintext) message while it is in transit and produce a new one that hashes to H, Bob can easily detect any changes Trudy has made to the message. Used for **integrity but not secrecy**.
Working of SHA-1

1. It starts out by padding the message by adding a 1 bit to the end, followed by as many 0 bits as are necessary, but at least 64, to make the length a multiple of 512 bits.

2. Then a 64-bit number containing the message length before padding is ORed into the low-order 64 bits.
3. During the computation, SHA-1 maintains five 32-bit variables, H_0 through H_4 , where the hash accumulates. They are initialized to constants specified in the standard.
4. Each of the blocks M_0 through M_{n1} is now processed in turn.
5. For the current block, the 16 words are first copied into the start of an auxiliary 80-word array, W. Then the other 64 words in W are filled in using the formula

$$W_i = S^1(W_{i-3} \text{ XOR } W_{i-8} \text{ XOR } W_{i-14} \text{ XOR } W_{i-16})(16 \leq i \leq 79)$$

   where $S^b(W)$ represents the left circular rotation of the 32-bit word, W, by b bits.
6. Now five scratch variables, A through E, are initialized from H_0 through H_4 , respectively.
7. The actual calculation can be expressed in pseudo-C as

```
/*for (i = 0; i < 80; i++) {
        temp = S^5(A) + f_i(B, C, D) + E + W_i + K_i ;
        E = D; D = C; C = S^30(B); B = A; A = temp;
}*/
```

   where the K_i constants are defined in the standard.
8. The mixing functions f_i are defined as

$$f_i(B,C,D) = (B \text{ AND } C)OR(\text{ NOT } B \text{ AND } D) \quad (0 \leq i \leq 19)$$
$$f_i(B,C,D) = B \text{ XOR } C \text{ XOR } D \quad (20 \leq i \leq 39)$$
$$f_i(B,C,D) = (B \text{ AND } C)OR(B \text{ AND } D)OR(C \text{ AND } D) \quad (40 \leq i \leq 59)$$
$$f_i(B,C,D) = B \text{ XOR } C \text{ XOR } D \quad (60 \leq i \leq 79)$$

9. When all 80 iterations of the loop are completed, A through E are added to H_0 through H_4 , respectively.
10. Now that the first 512-bit block has been processed, the next one is started. The W array is reinitialized from the new block, but H is left as it was.
11. When this block is finished, the next one is started, and so on, until all the 512-bit message blocks have been tossed into the soup.
12. When the last block has been finished, the five 32-bit words in the H array are output as the 160-bit cryptographic hash.

New versions of SHA-1 have been developed that produce hashes of 224, 256, 384, and 512 bits. Collectively, these versions are called **SHA-2**.

MD5

1. The message is padded to a length of 448 bits (modulo 512).
2. Then the original length of the message is appended as a 64-bit integer to give a total input whose length is a multiple of 512 bits.
3. Each round of the computation takes a 512-bit block of input and mixes it thoroughly with a running 128-bit buffer.

4. For good measure, the mixing uses a table constructed from the sine function.
5. The point of using a known function is to avoid any suspicion that the designer built in a clever back door through which only he can enter.
6. This process continues until all the input blocks have been consumed. The contents of the 128-bit buffer form the message digest.

Birthday Attack

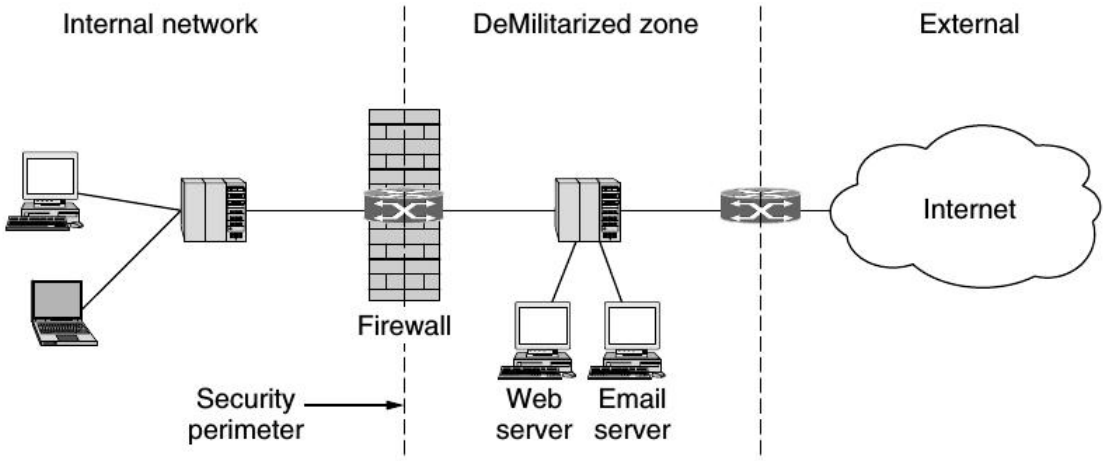
One might think that it would take on the order of 2^m operations to subvert an m-bit message digest. In fact, $2^{\frac{m}{2}}$ operations will often do using the birthday attack. Birthday attack takes a lot of time in computing message digrsts from SHA-1.

28.3 Communication Security

28.3.1 Firewalls

Keeps digital pests & intruders from using company's LAN. A company can have many LANs connected in arbitrary ways, but all traffic to or from the company is forced through an electronic drawbridge (firewall), as shown in Fig. 28.3. No other route exists.

Figure 28.3: Firewall Protecting internal network



The firewall acts as a **packet filter**. It inspects each and every incoming and outgoing packet. Packets meeting some criterion described in rules formulated by the network administrator are forwarded normally. Those that fail the test are uncermoniously dropped.
The filtering criterion is typically given as rules or tables that list sources and destinations that are acceptable, sources and destinations that are blocked, and default rules about what to do with packets coming from or going to other machines.
In the common case of a TCP/IP setting, a source or destination might consist of an IP address and a port. Ports indicate which service is desired.
The DMZ is the part of the company network that lies outside of the security perimeter. Anything goes here. By placing a machine such as a Web server in the DMZ, computers on the Internet can contact it to browse the company Web site. Now the firewall can be configured to block incoming TCP traffic to port 80 so that computers on the Internet cannot use this port

to attack computers on the internal network.
Stateful firewalls map packets to connections and use TCP/IP header fields to keep track of connections.
Another level of sophistication up from stateful processing is for the firewall to implement **application-level gateways**.
Looking inside packets, beyond even the TCP header, to see what the application is doing. With this capability, it is possible to distinguish HTTP traffic used for Web browsing from HTTP traffic used for peer-to-peer file sharing.
Firewalls cannot prevent **Denial of Service** attacks, where an intruder sends thoudsands of req. after establishing connection to bring own the web server.

Chapter 29

Application Layer

29.1 DNS - Domain Name System

The essence of DNS is the invention of a hierarchical, domain-based naming scheme and a distributed database system for implementing this naming scheme. It is *primarily used for mapping host names to IP addresses* but can also be used for other purposes.

1. To map a name onto an IP address, an application program calls a library procedure called the resolver, passing it the name as a parameter.
2. The resolver sends a query containing the name to a local DNS server, which looks up the name and returns a response containing the IP address to the resolver, which then returns it to the caller.
3. The query and response messages are sent as UDP packets.
4. Armed with the IP address, the program can then establish a TCP connection with the host or send it UDP packets.

DNS Name Space:

Figure 29.1: A portion of the Internet domain name space

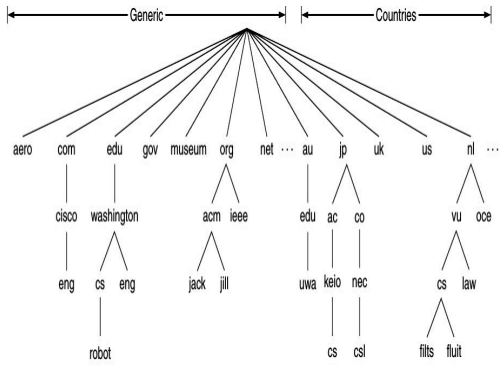
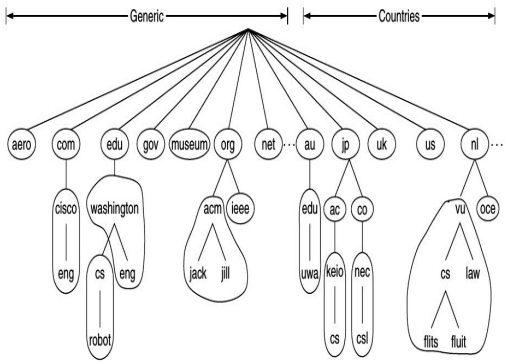


Figure 29.2: Domain Name Space mapped to zones



Each domain is named by the path upward from it to the (unnamed) root. The components are separated by periods. Thus, the engineering department at Cisco might be eng.cisco.com., rather than a UNIX -style name such as /com/cisco/eng. Notice that this hierarchical naming

means that eng.cisco.com. does not conflict with a potential use of eng in eng.washington.edu., which might be used by the English department at the University of Washington.
Domain Names: absolute or relative, case-insensitive .
Domain Resource Records: Every domain, whether it is a single host or a top-level domain, can have a set of resource records associated with it. These records are the DNS database. The primary function of DNS is to map domain names onto resource records. A resource record is a five-tuple.

```
Domain_name     Time_to_live     Class     Type     Value
```

1. **Domain name** Tells the domain to which this record applies. Primary Key
2. **Time to live** Field gives an indication of how stable the record is. Seconds
3. **Class** For Internet information, it is always IN. For non-Internet information, other codes can be used, but in practice these are rarely seen.
4. **Type** Tells what kind of record this is.
5. **Value** This field can be a number, a domain name, or an ASCII string. The semantics depend on the record type.

Figure 29.3: DNS Resource Records Types

Type	Meaning	Value
SOA	Start of authority	Parameters for this zone
A	IPv4 address of a host	32-Bit integer
AAAA	IPv6 address of a host	128-Bit integer
MX	Mail exchange	Priority, domain willing to accept email
NS	Name server	Name of a server for this domain
CNAME	Canonical name	Domain name
PTR	Pointer	Alias for an IP address
SPF	Sender policy framework	Text encoding of mail sending policy
SRV	Service	Host that provides it
TXT	Text	Descriptive ASCII text

29.1.1 Name Servers

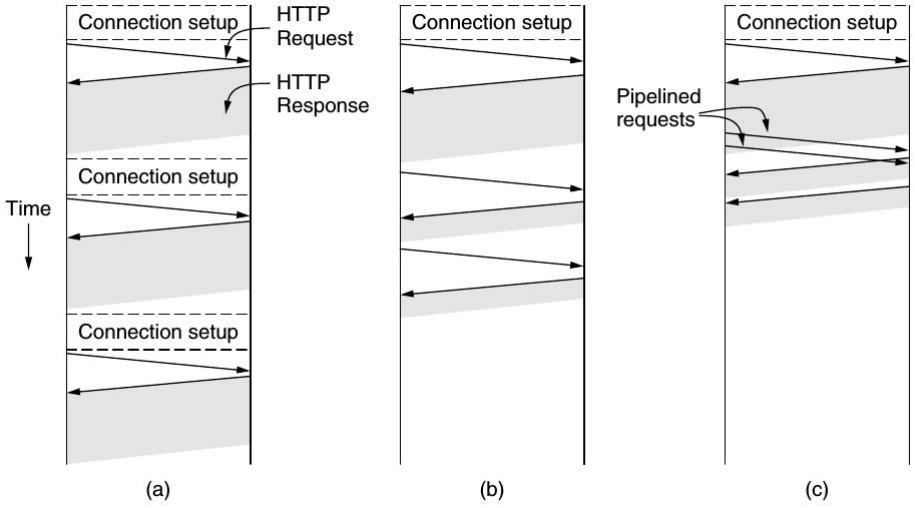
A single name server could contain the entire DNS database and respond to all queries about it. The DNS name space is divided into nonoverlapping zones. Where the zone boundaries are placed within a zone is up to that zone's administrator.

29.2 HTTP - Hyper Text Transfer Protocol

Used to transport all this information between Web servers and clients. HTTP is a simple request-response protocol that normally runs over TCP. It specifies what messages clients may

send to servers and what responses they get back in return.
The request and response headers are given in ASCII, just like in SMTP. The contents are given in a MIME-like format, also like in SMTP. **Default Port:** 80.
HTTP 1.0 After the connection was established a sin-
gle request was sent over and a single response was sent back. **HTTP 1.1**

Figure 29.4: HTTP 1.1 with (a) multiple connections and sequential requests. (b)A persistent connection and sequential requests. (c) A persistent connection and pipelined requests.



Chapter 30

Routing Algorithms

30.1 Introduction

1. The **routing algorithm** is that part of the **network layer** software responsible for deciding which output line an incoming packet should be transmitted on.
2. If the network uses datagrams internally, this decision must be made anew for every arriving data packet since the best route may have changed since last time.
3. If the network uses virtual circuits internally, routing decisions are made only when a new virtual circuit is being set up. (**Session Routing**)
4. Difference between routing & forwarding: **Routing -** making the decision which routes to use (filling & updating routing tables). **Forwarding -** handles each packet as it arrives, looking up the outgoing line to use for it in the routing tables.
5. Desirable in a routing algorithm: correctness, simplicity, robustness, stability, fairness, and efficiency.
6. Desired: Minimizing the mean packet delay, maximizing total network throughput.
7. Routing algorithms can be grouped into two major classes: nonadaptive and adaptive.
   (a) Non Adaptive: Static Routing
   (b) Adaptive: Dynamic Routing
8. **Optimality Principle**: It states that if router J is on the optimal path from router I to router K, then the optimal path from J to K also falls along the same route.
9. The set of optimal routes from all sources to a given destination form a tree rooted at the destination. Such a tree is called a **sink tree**.
10. DAG - Directed Acyclic Graphs. No Loops.

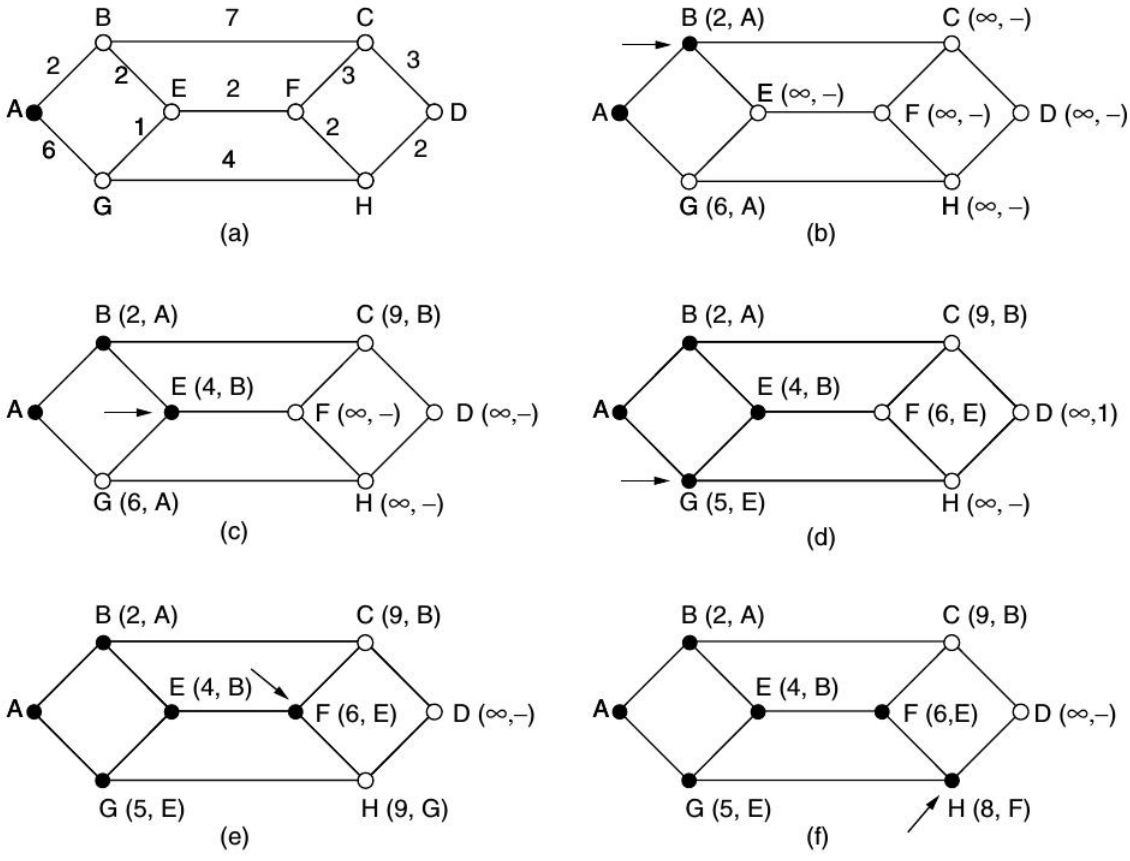
30.2 Shortest Path Algorithm - Dijsktra

Algorithm 2 Calculate the shortest path from source node to all destination nodes

Input: A weighted undirected graph with source node and non-negative weights
Output: Shortest path between source node and all other nodes

1. Source node labelled(permanent : path fixed)
2. Other nodes unalbelled(temporary : path not fixed)
3. Node labelling $\leftarrow (\infty, -)$: represents predecessor in shortest path route.
4. **while** All nodes are not visited **do**
5. **while** All adjacent nodes are not visited **do**
6. dist. $\leftarrow$ distance between previous node and visited node.
7. pred. $\leftarrow$ Previous node
8. Mark visited nodes with as(dist., pred.)
9. **end while**
10. Mark remaining nodes as $(\infty, -)$
11. Mark the smallest dist. node as permanent
12. **end while**

Figure 30.1: Sample run of Dijsktra's Algorithm



30.3 Flooding

1. Every incoming packet is sent out on every outgoing line except the one it arrived on.
2. Measures to prevent duplication:
   (a) **Hop counter** contained in the header of each packet that is decremented at each hop, with the packet being discarded when the counter reaches zero. Ideally, the hop counter should be initialized to the length of the path from source to destination.
   (b) Have routers keep track of which packets have been flooded, to avoid sending them out a second time. One way to achieve this goal is to have the source router put a **sequence number** in each packet it receives from its hosts.

30.4 Distance Vector Routing - Bellman Ford

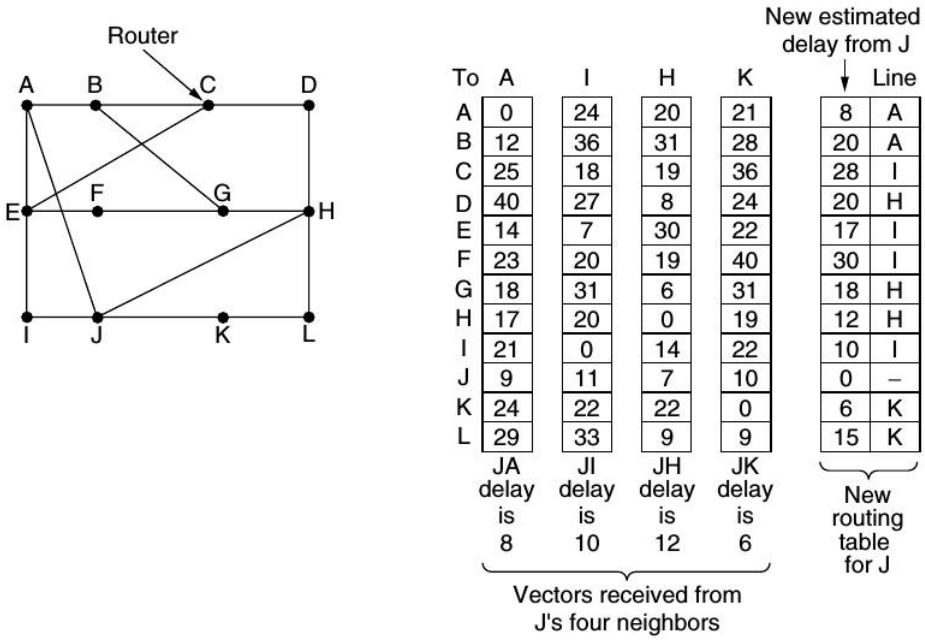
A **distance vector routing algorithm** operates by having each router maintain a table (i.e., a vector) giving the best known distance to each destination and which link to use to get there. These tables are updated by exchanging information with the neighbors. Eventually, every router knows the best link to reach each destination.

Algorithm 3 Calculate the shortest path from source node to all destination nodes

Input: A weighted undirected graph with source node and non-negative weights
Output: Shortest path between any node and any nodes

1. shortestPath[n][n] ← Vector table containing shortest distances from V_i to V_j
2. distArray[n][n] ← distance array containing distance between distance between any node to any node
3. distArray[i][j] ← distance between V_i & V_j.
4. delay[n][n] ← delay from any node to its neighbour nodes.
5. delay[i][j] ← delay from V_i to V_j. - if not a neighbour. $k \in$ W if neighbour.
6. **while** all nodes are not visited **do**
7. V_i ← visited, adj = 0
8. **for** All nodes V_j adjacent to V_i **do**
9. Calculate delay[i][j]
10. adj++
11. **end for**
12. paths[adj] = 0
13. **for** All nodes V_k **do**
14. **for** All nodes V_j adjacent to V_i **do**
15. paths[j] = delay[i][j] + distArray[j][k];
16. **end for**
17. min ← minimum value in paths[].
18. shortestPath[i][k] = min
19. **end for**
20. **end while**

Figure 30.2: Sample run of Bellman Ford's Algorithm



Count to infinity Problem: The settling of routes to best paths across the network is called **convergence**.

30.5 Link State Routing

Variants of link state routing called **IS-IS (Intermediate System-Intermediate System)** and **OSPF (Open Shortest Path First)**.
Each router must do the following things to make it work:

1. Discover its neighbors and learn their network addresses.
2. Set the distance or cost metric to each of its neighbors.
3. Construct a packet telling all it has just learned.
4. Send this packet to and receive packets from all other routers.
5. Compute the shortest path to every other router.

In effect, the complete topology is distributed to every router. Then DijkstraâĂŹs algorithm can be run at each router to find the shortest path to every other router. Compared to distance vector routing, link state routing requires more memory and computation. But no problem of slow convergence as in distance vector routing.

30.5.1 Learning abput neighbours

Sending a special HELLO packet on each point-to-point line. The router on the other end is expected to send back a reply giving its name. A better way to model the LAN connecting different routers is to consider it as a node itself.

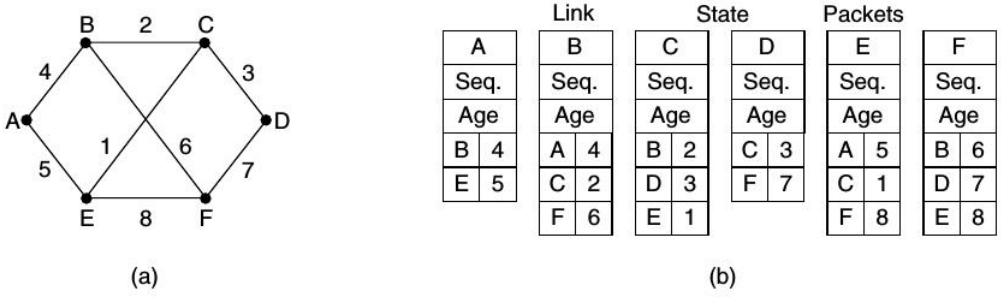
30.5.2 Setting Link Costs

A common choice is to make the cost inversely proportional to the bandwidth of the link. For example, 1-Gbps Ethernet may have a cost of 1 and 100-Mbps Ethernet a cost of 10. This makes higher-capacity paths better choices. Ig routers geogfraphoically spread, then delay could be link cost.

30.5.3 Building Link State Packets

One possibility is to build them periodically, that is, at regular intervals. Another possibility is to build them when some significant event occurs, such as a line or neighbor going down or coming back up again or changing its properties appreciably.

Figure 30.3: Link Sate Packets



30.5.4 Distributing the Link State Packets

All of the routers must get all of the link state packets quickly and reliably.

Fundamental Idea

Use flooding to distribute the link state packets to all routers. To keep the flood in check, each packet contains a sequence number that is incremented for each new packet sent. Routers keep track of all the (source router, sequence) pairs they see. When a new link state packet comes in, it is checked against the list of packets already seen. If it is new, it is forwarded on all lines except the one it arrived on. If it is a duplicate, it is discarded.

Problems

1. Wrap around of sequence numbers: **Solution** - Use 32 bit seq. numbers.
2. If a router ever crashes, it will lose track of its sequence number.
3. If a sequence number is ever corrupted and 65,540 is received instead of 4 (a 1-bit error), packets 5 through 65,540 will be rejected as obsolete, since the current sequence number will be thought to be 65,540. **Solution:** include age of packet amd decrement it once per second.

The send flags mean that the packet must be sent on the indicated link. The acknowledgement flags mean that it must be acknowledged there. The situation with the third packet, from E, is different. It arrives twice, once via EAB and once via EFB. Consequently, it has to be sent only to C but must be acknowledged to both A and F, as indicated by the bits.
If a duplicate arrives while the original is still in the buffer, bits have to be changed. For

example, if a copy of CâĂŹs state arrives from F before the fourth entry in the table has been forwarded, the six bits will be changed to 100011 to in- dicate that the packet must be acknowledged to F but not sent there.

Figure 30.4: Packet Buffer for Router B

			Send flags			ACK flags			
Source	Seq.	Age	A	C	F	A	C	F	Data
A	21	60	0	1	1	1	0	0	
F	21	60	1	1	0	0	0	1	
E	21	59	0	1	0	1	0	1	
C	20	60	1	0	1	0	1	0	
D	21	59	1	0	0	0	1	1	

30.5.5 Computing Shortest Path

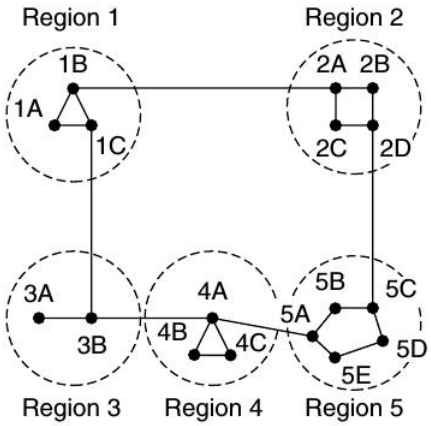
Every link in the newtwork graph is, in fact, represented twice, once for each direction.Dijkstra's algorithm can be run locally to construct the shortest paths to all possible destinations. The results of this algorithm tell the router which link to use to reach each destination. This information is installed in the routing tables, and normal operation is resumed.

Use a 32-bit sequence number. With one link state packet per second, it would take 137 years to wrap around. Time = $\frac{2^{\text{no. of bits}}}{BW}$. Here BW = 1lsp/sec. no. of bits = 32

30.6 Hierarchial Routing

When hierarchical routing is used, the routers are divided into what we will call regions. Each router knows all the details about how to route packets to destinations within its own region but knows nothing about the internal structure of other regions. For huge networks, a two-level hierarchy may be insufficient; it may be necessary to group the regions into clusters, the clusters into zones, the zones into groups, and so on.

Figure 30.5: Hierarchial Routing



(a)

Full table for 1A

Dest.	Line	Hops
1A	–	–
1B	1B	1
1C	1C	1
2A	1B	2
2B	1B	3
2C	1B	3
2D	1B	4
3A	1C	3
3B	1C	2
4A	1C	3
4B	1C	4
4C	1C	4
5A	1C	4
5B	1C	5
5C	1B	5
5D	1C	6
5E	1C	5

(b)

Hierarchical table for 1A

Dest.	Line	Hops
1A	–	–
1B	1B	1
1C	1C	1
2	1B	2
3	1C	2
4	1C	3
5	1C	4

(c)

For example, consider a network with 720 routers. If there is no hierarchy, each router needs 720 routing table entries. If the network is partitioned into 24 regions of 30 routers each, each router needs 30 local entries plus 23 remote entries for a total of 53 entries. If a three-level hierarchy is chosen, with 8 clusters each containing 9 regions of 10 routers, each router needs 10 entries for local routers, 8 entries for routing to other regions within its own cluster, and 7 entries for distant clusters, for a total of 25 entries. Optimal no. of levels in N routers is $\ln N$ making total of $e \ln N$ no. of entries per router.

30.7 Broadcast Routing

Sending a packet to all destinations simultaneously is called **broadcasting**. The network bandwidth is therefore used more efficiently.
Problems: Source to know all the destinations, Router to determine where to send one multidestination packet as it is for multiple distinct packets.
Better broadcast routing technique: Flooding. When implemented with a sequence number per source, flooding uses links efficiently with a decision rule at routers.
Reverse path forwarding: **Spanning tree Algorithm:** A spanning tree is a subset of the network that includes all the routers but contains no loops. Sink trees are spanning trees. If each router knows which of its lines belong to the spanning tree, it can copy an incoming broadcast packet onto all the spanning tree lines except the one it arrived on. This method makes excellent use of bandwidth, generating the absolute minimum number of packets necessary to do the job.

did not understand

30.8 Multicast Routing

Sending a message to such a group is called multicasting, and the routing algorithm used is called **multicast routing**. All multicasting schemes require some way to create and destroy groups and to identify which routers are members of a group.
Multicast routing schemes build on the broadcast routing schemes we have already studied, sending packets along spanning trees to deliver the packets to the members of the group while making efficient use of bandwidth.
Types of pruning spanning trees:

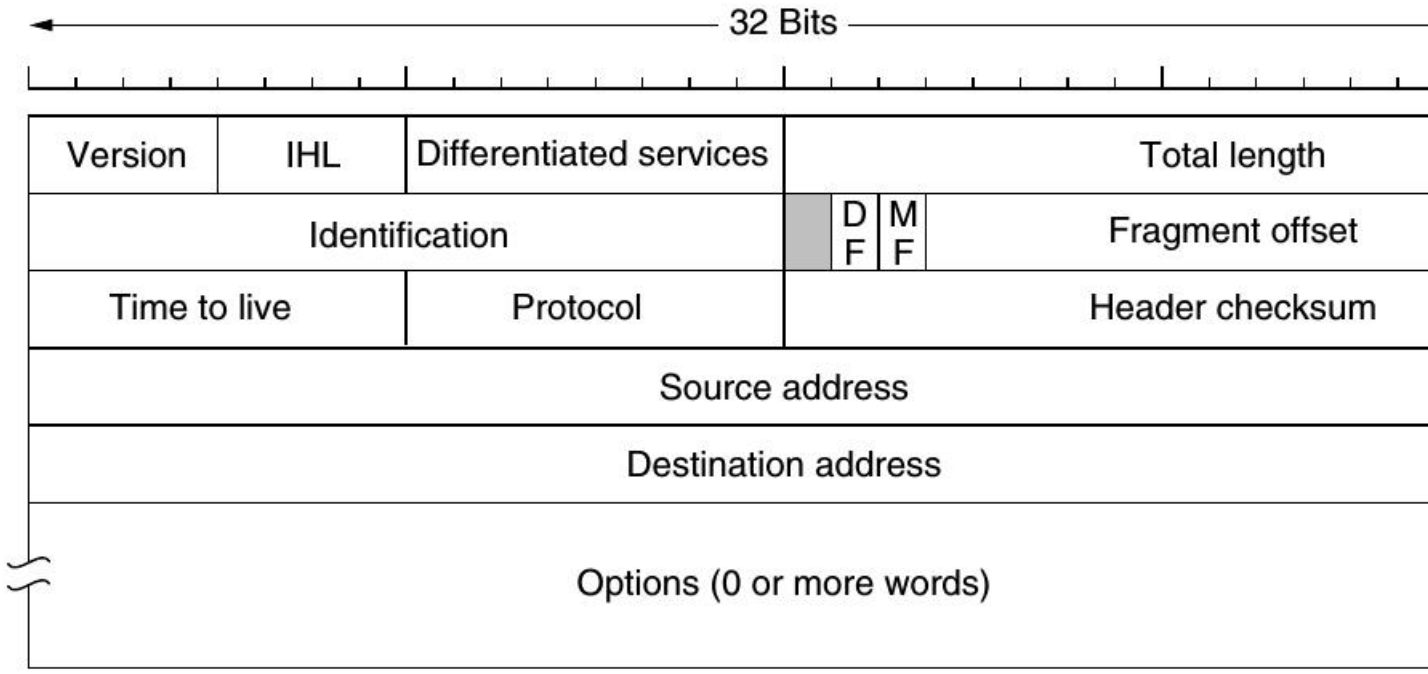
1. **MOSPF (Multicast OSPF)**
   (a) If **link state routing** is used and each router is aware of the complete topology, including which hosts belong to which groups.
   (b) Each router can then construct its own pruned spanning tree for each sender to the group in question by constructing a sink tree for the sender as usual and then removing all links that do not connect group members to the sink node.
2. **DVMRP (Distance Vector Multicast Routing Protocol)**
   (a) With **distance vector routing**, a different pruning strategy can be followed.
   (b) The basic algorithm is **reverse path forwarding**. However, whenever a router with no hosts interested in a particular group and no connections to other routers receives a multicast message for that group, it responds with a PRUNE message, telling the neighbor that sent the message not to send it any more multicasts from the sender for that group.
   (c) When a router with no group members among its own hosts has received such messages on all the lines to which it sends the multicast, it, too, can respond with a PRUNE message.

30.9 IPv4 - Network Layer

30.9.1 Communication in Internet

The transport layer takes data streams and breaks them up so that they may be sent as IP packets. In theory, packets can be up to 64 KB each, but in practice they are usually not more than 1500 bytes (so they fit in one Ethernet frame). IP routers forward each packet through the Internet, along a path from one router to the next, until the destination is reached. At the destination, the network layer hands the data to the transport layer, which gives it to the receiving process. When all the pieces finally get to the destination machine, they are reassembled by the network layer into the original datagram. This datagram is then handed to the transport layer.

Figure 30.6: IPv4 Header



Chapter 31

Error & Flow control - Data Link Layer

31.1 Introduction

The data link layer uses the services of the physical layer to send and receive bits over communication channels. It has a number of functions, including:

1. Providing a well-defined service interface to the network layer.
2. Dealing with transmission errors.
3. Regulating the flow of data so that slow receivers are not swamped by fast senders.

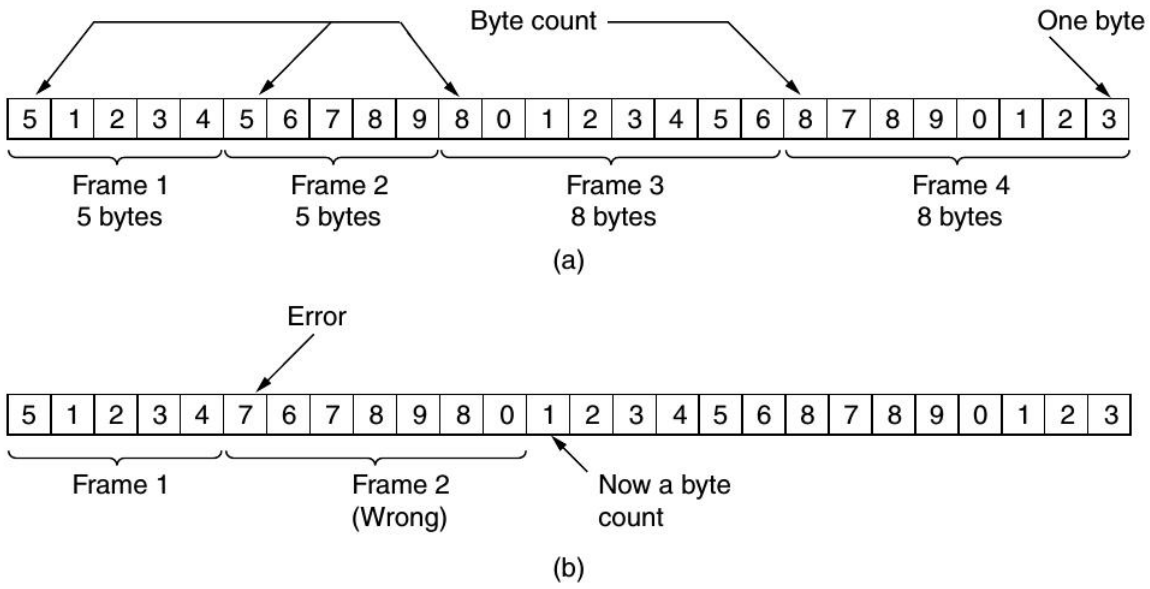
4 methods to **break bit streams into frames**:

1. Byte count.
2. Flag bytes with byte stuffing.
3. Flag bits with bit stuffing.

31.1.1 Byte Count

A field in the header to specify the number of bytes in the frame. When the data link layer at the destination sees the byte count, it knows how many bytes follow and hence where the end of the frame is.

Figure 31.1: Byte Count (a) Without errors. (b) With one error.

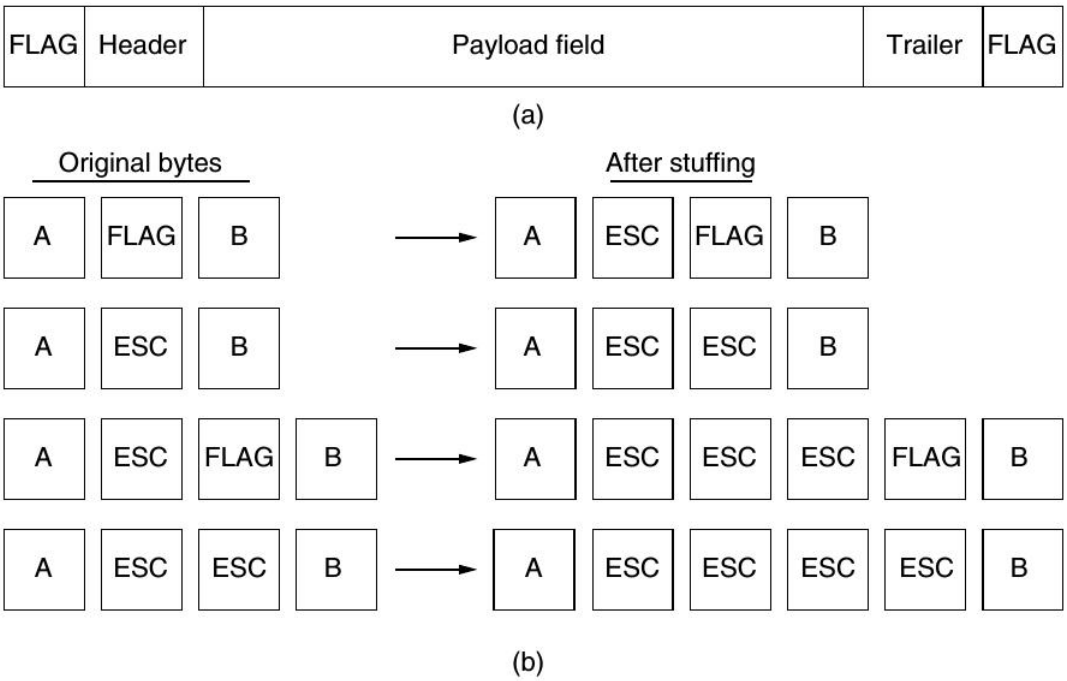


31.1.2 Byte Stuffing

1. Add **FLAG** byte at the start and end of each frame. Two consecutive flag bytes indicate the end of one frame and the start of the next.
2. If **FLAG** byte in data of the frame, then add **ESC** byte to the before of **FLAG** byte.
3. If **ESC** occurs in th e middle of the data, then stuff it with an **ESC** byte.

Byte Stuffing is used in PPP(Point to Point Protocol).

Figure 31.2: Byte Stuffing (a) A frame delimited by flag bytes. (b) Four examples of byte sequences before and after byte stuffing.



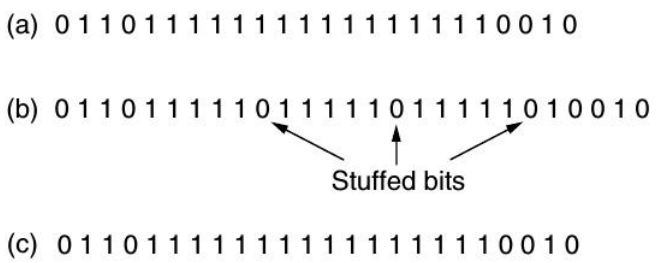
31.1.3 Bit Stuffing

Used in HDLC(High-level Data Link Control).

1. Each frame begins and ends with a special bit pattern, 01111110 or 0x7E in hexadecimal. - FLAG Byte.
2. Whenever the sender's data link layer encounters five consecutive 1s in the data, it automatically stuffs a 0 bit into the outgoing bit stream - ESC Byte.
3. When the receiver sees five consecutive incoming 1 bits, followed by a 0 bit, it automatically destuffs (i.e., deletes) the 0 bit.

Suppose 111110 is in data, does receiver destuff that leading to wrong information (in Bit Stuffing)?

Figure 31.3: Bit Stuffing (a) The original data. (b) The data as they appear on the line. (c) The data as they are stored in the receiver's memory after destuffing.



31.2 Error Correction & Detection

31.2.1 Error Correcting Codes

1. Hamming codes.
2. Binary convolutional codes.
3. Reed-Solomon codes.
4. Low-Density Parity Check codes.

In a **systematic code**, the m data bits are sent directly, along with the check bits, rather than being encoded themselves before they are sent. In a **linear code**, the r check bits are computed as a linear function of the m data bits.

Definition 38 - Codeword.

A frame generally consists of m data bits (message) and r code (redundant/check) bits to form an n bit codeword where $n = m + r$.

Hamming Codes

Definition 39 - Hamming distance.

The number of different digits $d(x, y)$ between two codewords x and y. For example, 000 and 111 have a Hamming distance of 3.

Definition 40 - Coderate.

The fraction of the codeword that carries information that is not redundant, or $\frac{m}{n}$.

Definition 41 - Even Parity.

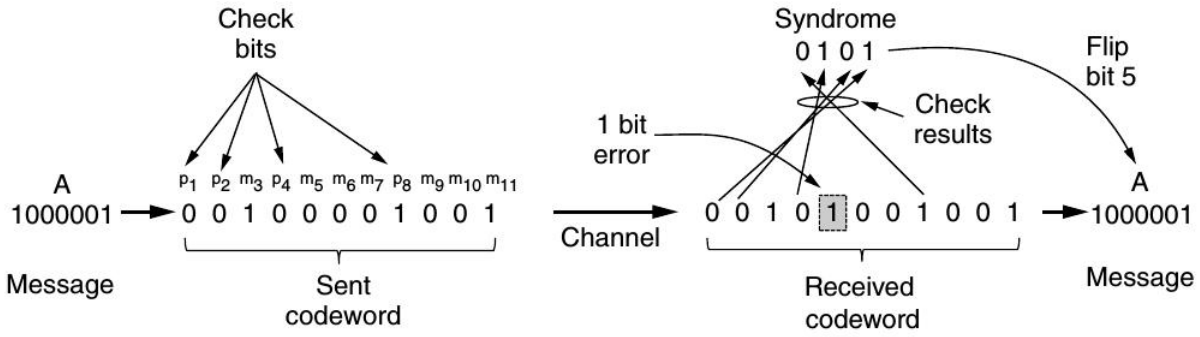
In the case of even parity, **the number of bits whose value is 1 in a given set are counted.** If that total is odd, the parity bit value is set to 1, making the total count of 1's in the set an even number. If the count of ones in a given set of bits is already even, the parity bit's value remains 0.

Definition 42 - Odd Parity.

In the case of odd parity, the situation is reversed. Instead, if the sum of bits with a value of 1 is odd, the parity bit's value is set to zero. And if the sum of bits with a value of 1 is even, the parity bit value is set to 1, making the total count of 1's in the set an odd number.

If two codewords are a Hamming distance d apart, it will require d single-bit errors to convert one into the other. To reliably detect d errors, you need a distance $d+1$ code because with such a code there is no way that d single-bit errors can change a valid codeword into another valid codeword. To correct d errors, you need a distance $2d+1$ code because that way the legal codewords are so far apart that even with d changes the original codeword is still closer than any other codeword. Number of check bits needed to correct single errors: $(m+r+1) \leq 2^r$. The bits that are powers of 2 (1, 2, 4, 8, 16, etc.) are check bits. The rest (3, 5, 6, 7, 9, etc.) are filled up with the m data bits. This construction gives a code with a Hamming distance of 3 (11 = 1 + 2 + 8. max 3 numbers), which means that it can correct single errors (or detect double errors).
Example: Message - 1000001. To compute check bits: express rest of bits in as a summation of check bits. ex., 11 = 1 + 2 + 8. No. of 2's in expanded sum gives the parity for 2 in new message. Sent Message: 0 0 1 0 0 0 0 1 0 0 1, Received Mwssage: 0 0 1 0 1 0 0 1 0 0 1.
Check bits should be computed including check bits: $1:1, 2:0, 4:1, 8:0$. Recheck parity bits: $1:1, 2:0, 4:1, 8:1$. A single-bit error occurred on the channel so the check results are 0, 1, 0, and 1 for k = 8, 4, 2, and 1. Parity bits of 4 and 1 are changed. Therefore, bit $4+1=5$ should be flipped.

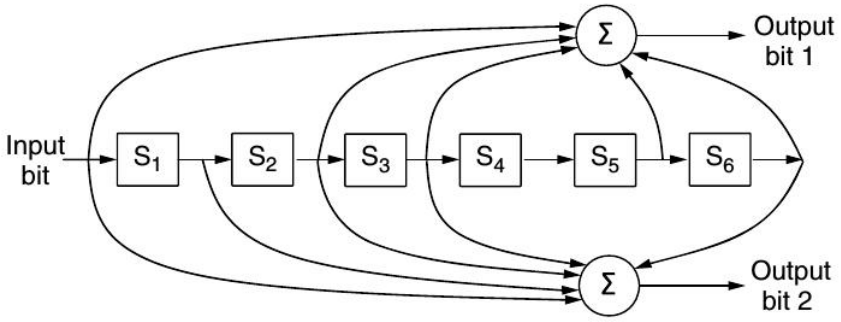
Figure 31.4: Hamming Codes for (11, 7) for even parity



Convolution Codes

Not a block code. An encoder processes a sequence of input bits and generates a sequence of output bits. Output depends on current and previous input bits(memory). The number of previous bits on which the output depends is called the constraint length of the code.

Figure 31.5: Convolution Codes



Reed Solomon Codes

Reed-Solomon codes are linear block codes. Reed-Solomon codes are based on the fact that every n degree polynomial is uniquely determined by $n + 1$ points. we have two data points that represent a line and we send those two data points plus two check points chosen to lie on the same line. If one of the points is received in error, we can still recover the data points by fitting a line to the received points. Three of the points will lie on the line, and one point, the one in error, will not. By finding the line we have corrected the error.
For m bit symbols, the codewords are $2^m - 1$ symbols long. A popular choice is to make $m = 8$ so that symbols are bytes. A codeword is then 255 bytes long. The (255, 233) code is widely used; it adds 32 redundant symbols to 233 data symbols.
They are based on m bit symbols, a single-bit error and an m-bit burst error are both treated simply as one symbol error. When $2t$ redundant symbols are added, a Reed-Solomon code is able to correct up to t errors in any of the transmitted symbols. This means, for example, that the (255, 233) code, which has 32 redundant symbols, can correct up to 16 symbol errors. Since the symbols may be consecutive and they are each 8 bits, an error burst of up to 128 bits can be corrected.

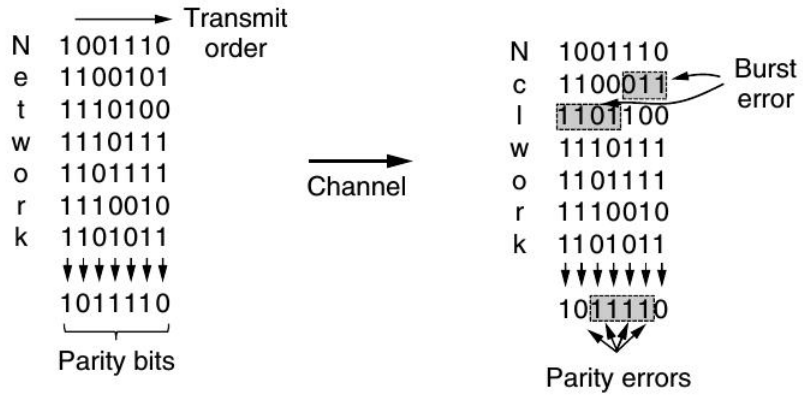
31.2.2 Error Detecting Codes

1. Parity.
2. Checksums.
3. Cyclic Redundancy Checks (CRCs).

Parity

A single parity bit is appended to the data. The parity bit is chosen so that the number of 1 bits in the codeword is even (or odd). For example, when 1011010 is sent in even parity, a bit is added to the end to make it 10110100. With odd parity 1011010 becomes 10110101. A code with a single parity bit has a distance of 2, since any single-bit error produces a codeword with the wrong parity. This means that it can detect single-bit errors.

Figure 31.6: Interleaving parity



Checksums

The checksum is usually placed at the end of the message, as the complement of the sum function. This way, errors may be detected by summing the entire received codeword, both data bits and checksum. If the result comes out to be zero, no error has been detected.

How to calculate and detect errors in checksum?

CRC(Cyclic Redundancy Check)

For example, 110001 has 6 bits and thus represents a six-term polynomial with coefficients 1, 1, 0, 0, 0, and 1: $1x^5 + 1x^4 + 0x^3 + 0x^2 + 0x^1 + 1x^0$. Addition and substraction in XOR format. When the polynomial code method is employed, the sender and receiver must agree upon a generator polynomial, $G(x)$, in advance. Both the high and low-order bits of the generator must be 1. To compute the CRC for some frame with m bits corresponding to the polynomial $M(x)$, the frame must be longer than the generator polynomial. The idea is to append a CRC to the end of the frame in such a way that the polynomial represented by the checksummed frame is divisible by $G(x)$. When the receiver gets the checksummed frame, it tries dividing it by $G(x)$. If there is a remainder, there has been a transmission error.

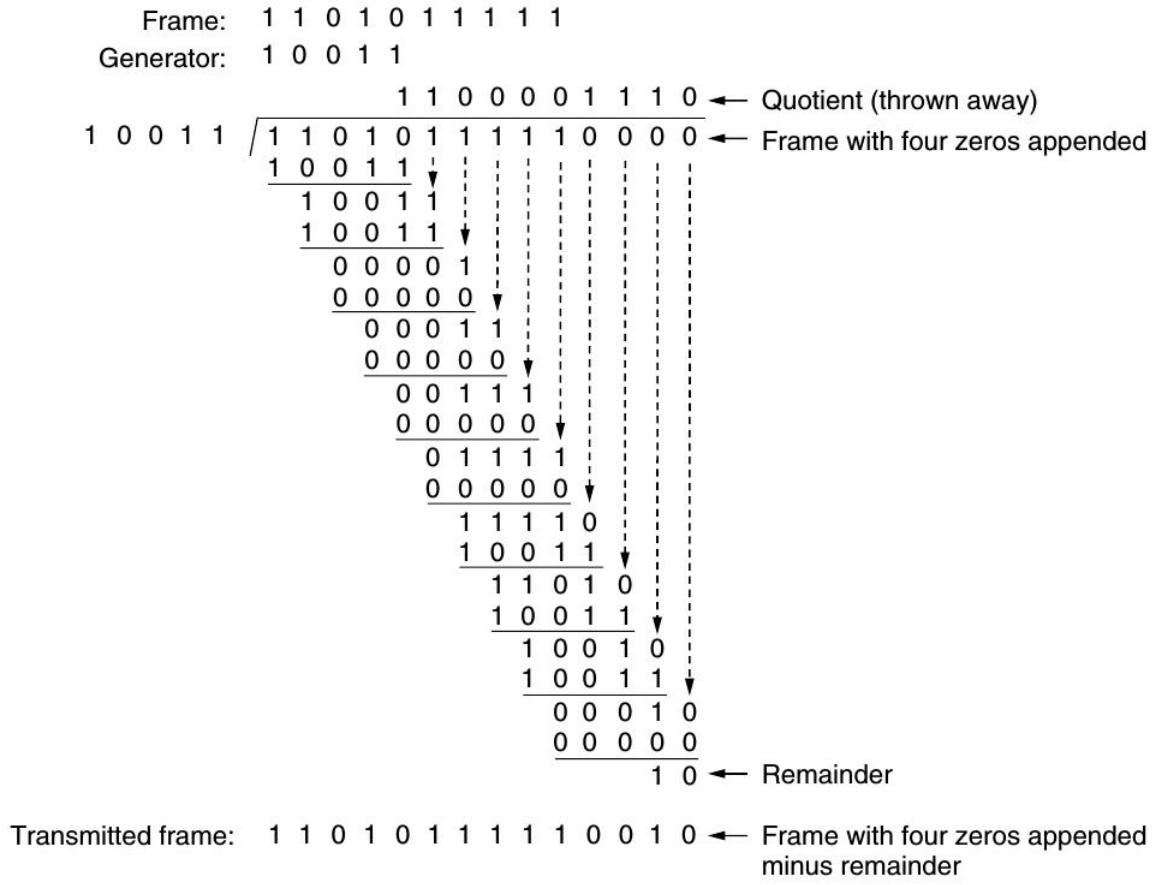
1. Let r be the degree of $G(x)$. Append r zero bits to the low-order end of the frame so it now contains $m + r$ bits and corresponds to the polynomial $x^r M(x)$.

2. Divide the bit string corresponding to $G(x)$ into the bit string corresponding to $x^r M(x)$, using modulo 2 division.

3. Subtract the remainder (which is always r or fewer bits) from the bit string corresponding to $x^r M(x)$ using modulo 2 subtraction. The result is the checksummed frame to be transmitted. Call its polynomial $T(x)$.

How are error detected in CRC?

Figure 31.7: Calculating CRC



31.3 Data Link Protocols

31.3.1 Simplex Protocol

Assumptions: Error free flow of information, simplex protocol. **Sender**: The sender in data link layer receives packet fom the network layer of sender. The data link layer breaks the packets into frames and passes onto the receiver in the data link layer. **Receiver**: The receiver receives the frames from the sender of data link layer. It then passes it to network layer of receiver and waits foor other frames..

31.3.2 Stop & Wait Protocol for Error free

Preventing the sender from flooding the receiver with frames faster than the latter is able to process them. After having passed a packet to its network layer, the receiver sends a little dummy frame back to the sender which, in effect, gives the sender permission to transmit the next frame. After having sent a frame, the sender is required by the protocol to bide its time until the little dummy (i.e., acknowledgement) frame arrives.

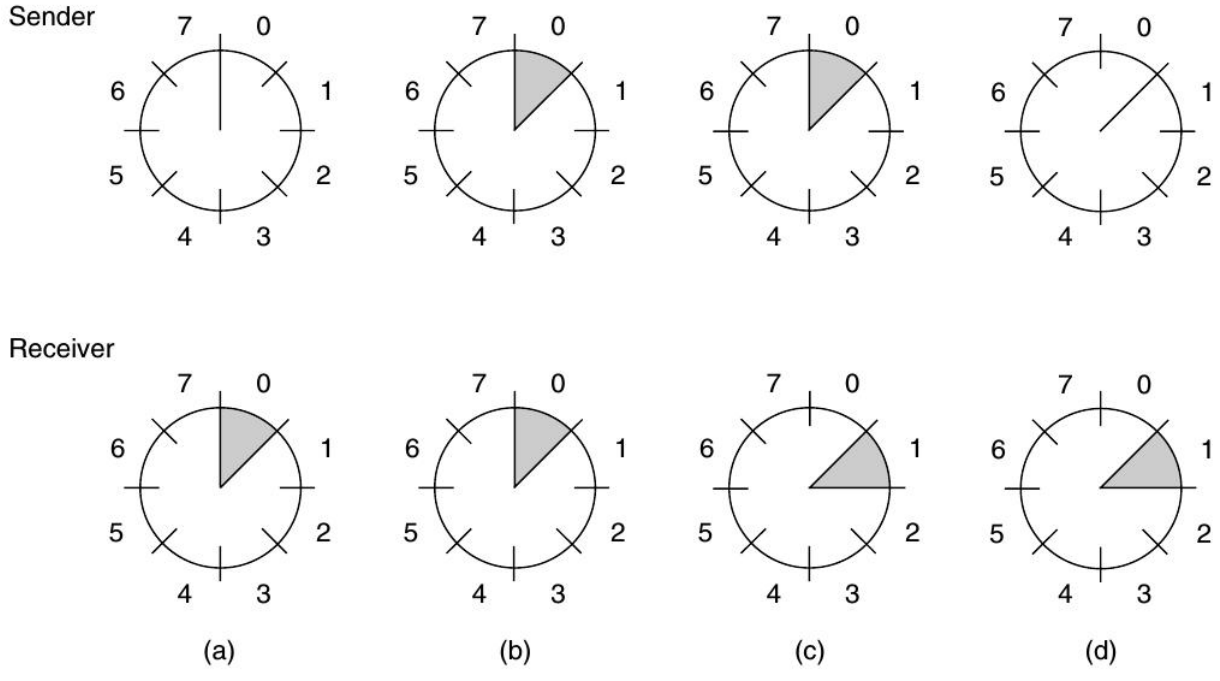
31.3.3 Stop & Wait Protocol for Noisy Channel

Example: ARQ (Automatic Repeat reQuest)

31.3.4 Sliding Window Protocols

Piggybacking: Sender of data link layer sends frames to data link layer of receiver and starts timer for acknowledgement to come. After receiving the frame, instead of sending acknowledgement the receiver waits for packet from network layer and attaches the ack to it and sends it to the sender of dll. If n/w layer doesn't send any packet ack frame is sent separately. Use of sequence nos. to avoid duplication. Use of kind field in header to find out if frame is data or control(ack) in full-duplex channel(two way communication).
The sender maintains a set of sequence numbers corresponding to frames it is permitted to send. These frames are said to fall within the sending window. Similarly, the receiver also maintains a receiving window corresponding to the set of frames it is permitted to accept.
The sequence numbers within the sender's window represent frames that have been sent or can be sent but are as yet not acknowledged. Whenever a new packet arrives from the network layer, it is given the next highest sequence number, and the upper edge of the window is advanced by one. When an acknowledgement comes in, the lower edge is advanced by one. In this way the window continuously maintains a list of unacknowledged frames.

Figure 31.8: A sliding window of size 1, with a 3-bit sequence number. (a) Initially. (b) After the first frame has been sent. (c) After the first frame has been received. (d) After the first acknowledgement has been received.

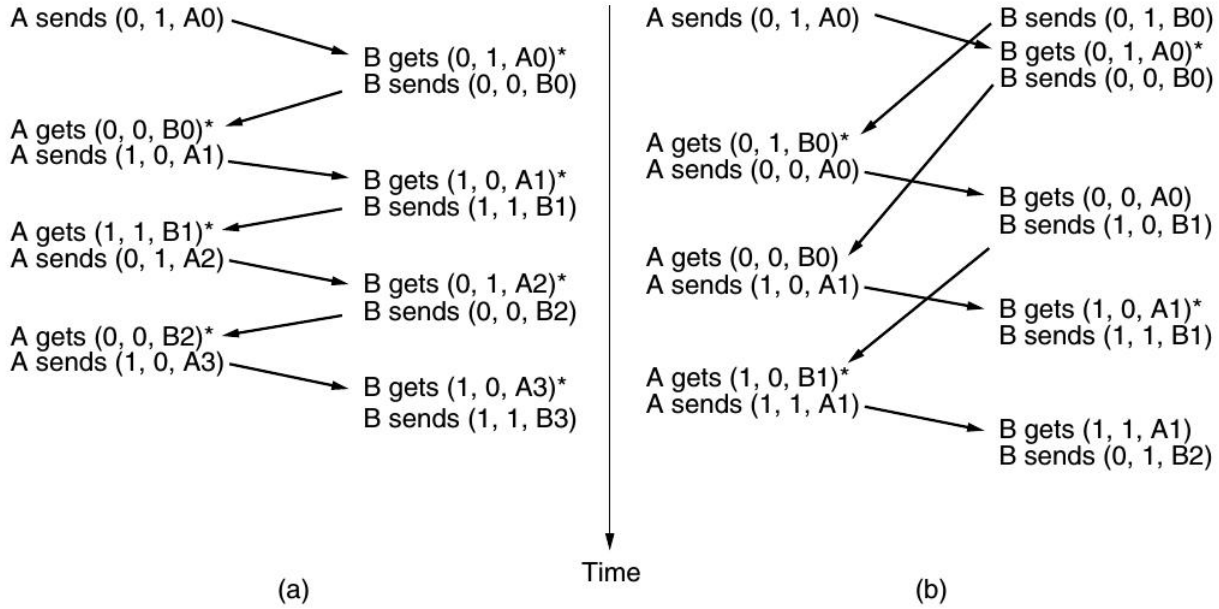


1 bit Sliding Window Protocol

Assume that computer A is trying to send its frame 0 to computer B and that B is trying to send its frame 0 to A. Suppose that A sends a frame to B, but A's timeout interval is a little too

short. Consequently, A may time out repeatedly, sending a series of identical frames, all with seq = 0 and ack = 1. When the first valid frame arrives at computer B, it will be accepted and frame expected will be set to a value of 1. All the subsequent frames received will be rejected because B is now expecting frames with sequence number 1, not 0. Furthermore, since all the duplicates will have ack = 1 and B is still waiting for an acknowledgement of 0, B will not go and fetch a new packet from its network layer.
After every rejected duplicate comes in, B will send A a frame containing seq = 0 and ack = 0. Eventually, one of these will arrive correctly at A, causing A to begin sending the next packet. No combination of lost frames or premature timeouts can cause the protocol to deliver duplicate packets to either network layer, to skip a packet, or to deadlock. The protocol is correct.

Figure 31.9: (a) Normal case. (b) Abnormal case. The notation is (seq, ack, packet number). An asterisk indicates where a network layer accepts a packet.



Go Back N

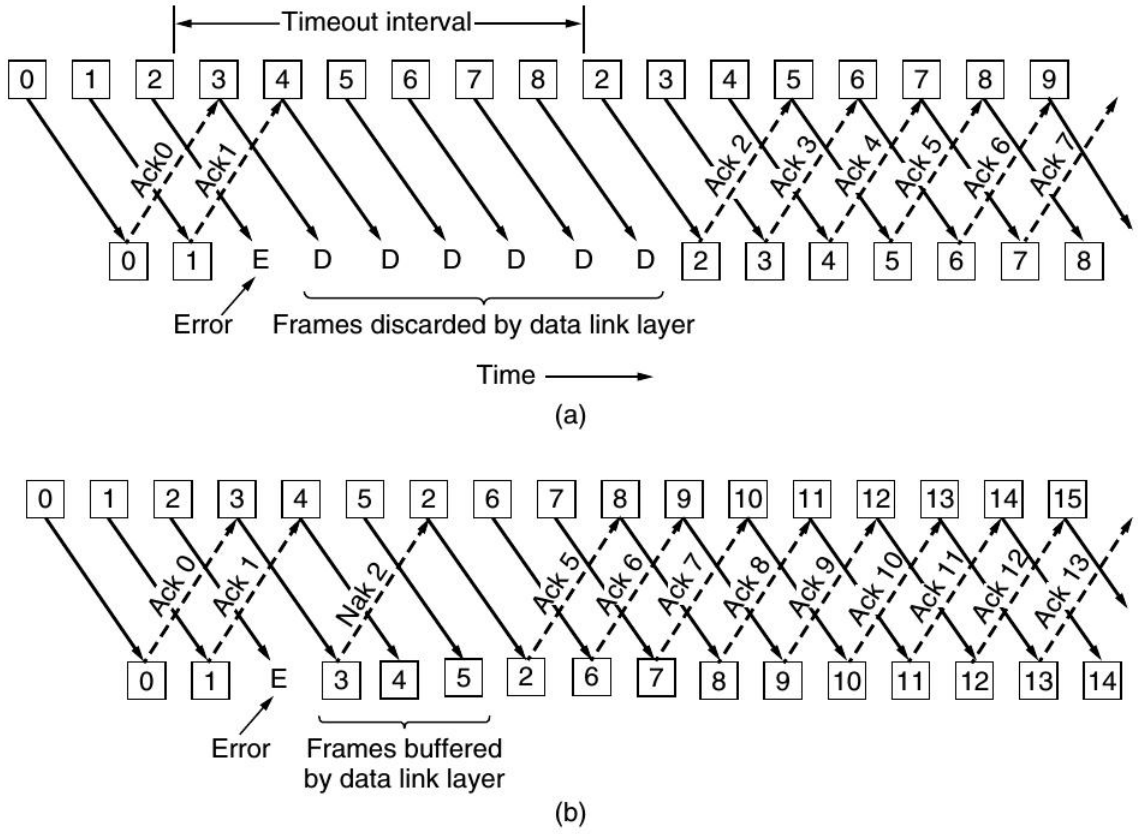
Bandwidth Delay Product(BDP) = Bandwidth(bits/sec) × One-way transit time

$$BD = \frac{BDP}{\text{No. of frames}}$$

Max no. of frames to be sent before blocking = Max. window size = $w = 2BD + 1$.

$$\text{Link Utilization} \leq \frac{w}{2BD + 1}$$

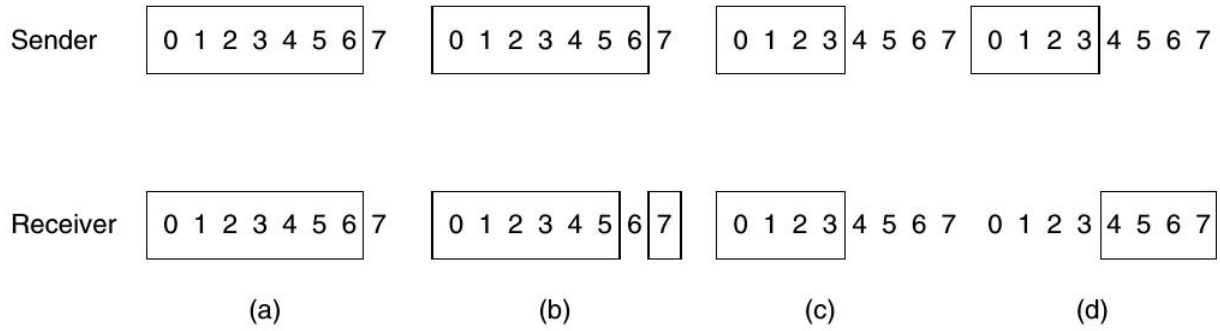
Figure 31.10: Pipelining and error recovery. Effect of an error when (a) receiver's window size is 1(Go Back n) and (b) receiver's window size is large(Selective Repeat).



Selective Repeat

Please understand seletive repeat

Figure 31.11: (a) Initial situation with a window of size 7. (b) After 7 frames have been sent and received but not acknowledged. (c) Initial situation with a window size of 4. (d) After 4 frames have been sent and received but not acknowledged.



Window size $\leq$ Sequence number. [**?**], [**?**], [5]. What is the maximum window size? The receiver must be able to distinguish a retransmission of an already received packet from the original transmission of a new packet. Thus the maximum window size is:

1. $2^n - 1$ in the case of Go-Back-N. Here the receiver accepts only the next expected packet and discards all out-of-order packets. In the example, with a 2-bit sequence number the maximum window size is 3
2. $\frac{2^n}{2}$ in Selective Repeat. Since the receiver accepts out-of-order packets, two consecutive windows should not overlap. Otherwise it is not able to distinguish duplicates from new transmissions. Hence, in the example the maximum window size is 2

Chapter 32

Data Link Layer Switching

32.1 Bridges

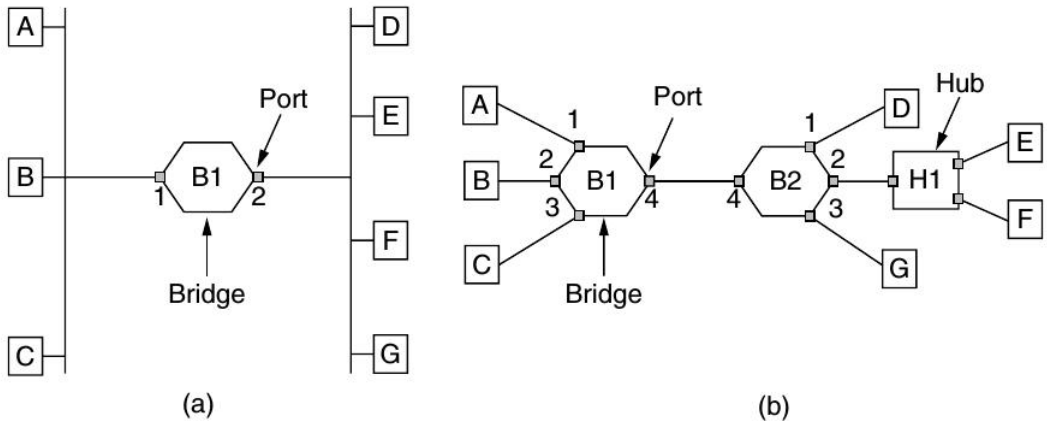
Definition 43 - Bridges.

Bridges are used for connecting multiple LAN's into a single logical LAN in an organization. Bridges operate in the data link layer, so they examine the data link layer addresses to forward frames. They can handle IP packets as well as other kinds of packets. Different kinds of cables can also be attached to one bridge.

Definition 44 - VLAN(Virtual LAN).

Treat one physical LAN as multiple logical LANs.

Figure 32.1: a) Bridge connecting two multidrop LANs. (b) Bridges (and a hub) connecting seven point-to-point stations.



32.1.1 Learning Bridges

The bridge must decide whether to forward or discard each frame, and, if the former, on which port to output the frame. A simple way to implement this scheme is to have a big (hash) table inside the bridge. The table can list each possible destination and which output port it belongs on.

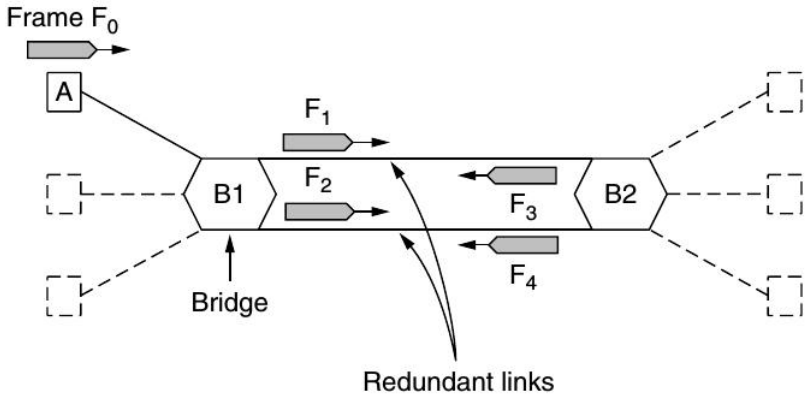
When the bridges are first plugged in, all the hash tables are empty. None of the bridges know where any of the destinations are, so they use a flooding algorithm: every incoming frame for an unknown destination is output on all the ports to which the bridge is connected except the one it arrived on. As time goes on, the bridges learn where destinations are. Once a destination is known, frames destined for it are put only on the proper port; they are not flooded.
To handle dynamic topologies, whenever a hash table entry is made, the arrival time of the frame is noted in the entry. Whenever a frame whose source is already in the table arrives, its entry is updated with the current time. Thus, the time associated with every entry tells the last time a frame from that machine was seen.
The routing procedure for an incoming frame depends on the port it arrives on (the source port) and the address to which it is destined (the destination address).

1. If the port for the destination address is the same as the source port, discard the frame.
2. If the port for the destination address and the source port are different, forward the frame on to the destination port.
3. If the destination port is unknown, use flooding and send the frame on all ports except the source port.

Bridges only look at the MAC addresses to decide how to forward frames, it is possible to start forwarding as soon as the destination header field has come in, before the rest of the frame has arrived (provided the output line is available, of course). This design reduces the latency of passing through the bridge, as well as the number of frames that the bridge must be able to buffer. It is referred to as **cut-through switching or wormhole routing** and is usually handled in hardware.

32.1.2 Spanning Trees

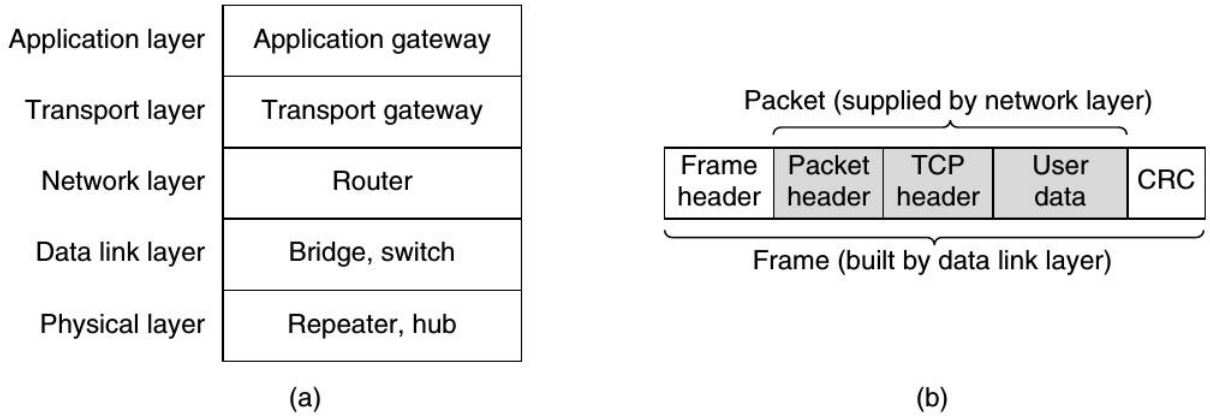
Figure 32.2: Bridges with two parallel links.



Bridges to communicate with each other and overlay the actual topology with a spanning tree that reaches every bridge (to avoid looping).
To build the spanning tree, the bridges run a distributed algorithm. Each bridge periodically broadcasts a configuration message out all of its ports to its neighbors and processes the messages it receives from other bridges, as described next. These messages are not forwarded, since their purpose is to build the tree, which can then be used for forwarding.
To make the choice of root, they each include an identifier based on their MAC address in the

configuration message, as well as the identifier of the bridge they believe to be the root. MAC addresses are installed by the manufacturer and guaranteed to be unique worldwide, which makes these identifiers convenient and unique. The bridges choose the bridge with the lowest identifier to be the root.
To find these shortest paths, bridges include the distance from the root in their configuration messages. Each bridge remembers the shortest path it finds to the root. The bridges then turn off ports that are not part of the shortest path.

Figure 32.3: (a) Which device is in which layer. (b) Frames, packets, and headers.



Chapter 33

Transport Layer

33.1 Introduction

Figure 33.1: The primitives of Transport Layer.

Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	Request a release of the connection

Figure 33.2: The primitives of TCP socket.

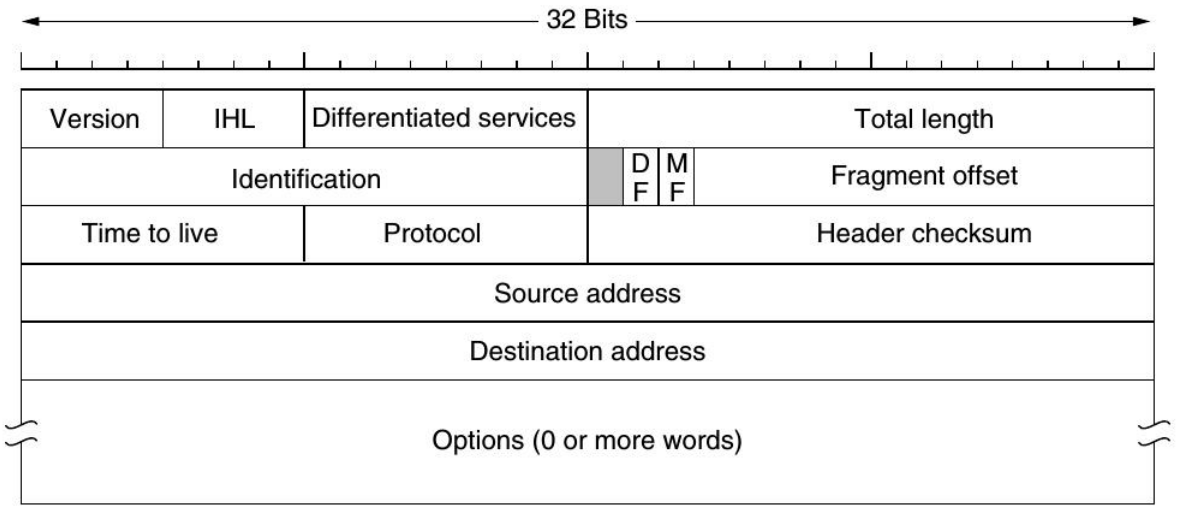
Primitive	Meaning
SOCKET	Create a new communication endpoint
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Chapter 34

Internet Protocol - IP

34.1 IPv4

Figure 34.1: IPv4 Header



Chapter 35

Network Layer

1. Physical Address - MAC Address: System Address:LAN card address:Ethernet address. Points to main memory.
2. Communication done with logical address - IP address. Types: Classful addressing, subnetting, supernetting.
   (a) Classful Addressing: 2 parts: net-id, hosts; 2 notations: binary: 00000011 00000011 00000011 00000011, octet: 255.255.255.255
   (b) Unicasting: Transfer data from one system to another system. Supported by class A,B,C.
   (c) Multicasting: Transfers data from one systems to many systems. Suported by Class D.
   (d) Class E : Future Use. First 4 bits in net-ID: 1111. IP address range(240 - 255).

1. **Part of CIDR (Classless Inerdomain Routing)**
2. Class A: 1^{st} bit of net ID should be 0. net-ID(8 bits): 2^7-2 bits(no. of possible networks). host-ID(24 bits): $2^{24}-2$(no. of possible hosts). IP address range(0 -126). 0.0.0.0 - DHCP, 127.x.y.z - Loopback.
3. Class B: $1^{st}, 2^{nd}$ bit of net ID should be 10. net-ID(16 bits): 2^{14} bits(no. of possible networks). host-ID(16 bits): $2^{16}-2$(no. of possible hosts). IP address range(128 -191).
4. Class C: $1^{st}, 2^{nd}, 3^{rd}$ bit of net ID should be 110. net-ID(24 bits): 2^{21} bits(no. of possible networks). host-ID(8 bits): 2^8-2(no. of possible hosts). IP address range(192 -223).
5. Class D: $1^{st}, 2^{nd}, 3^{rd}, 4^{th}$ bit of net ID should be 1110. net-ID(bits): 2 bits(no. of possible networks). host-ID(bits): $2-2$(no. of possible hosts). IP address range(224 - 239).
6. Class D & Class E cannot be divided into net-ID & host-ID because of multicasting.
7. Net masks for various classes:(net-ID all 0's and host-Id all 1's)
   (a) Class A - 255.0.0.0
   (b) Class B - 255.255.0.0
   (c) Class C - 255.255.255.0
8. Calculation of net-ID:
   (a) Identify class of IP address.

Check about net mask

   (b) Calculate net mask for that class.
   (c) Perform bitwise AND operation between IP address and net mask.

9. Range of private IP addresses:(for communication within LAN)
   (a) Class A: 10.0.0.0 - 10.255.255.255 - 1n/w
   (b) Class B: 172.16.0.0 - 172.31.255.255 - 16 n/w
   (c) Class C: 192.168.0.0 - 192.168.255.255 - 255 n/w
10. NAT - Network Address Translator. Converts private IP to public IP when pkt going outside the n/w & public IP to private IP if pkt coming in the n/w.
11. Destination Broadcast address
12. Limited Broadcast address

1. Service addressing system - Port address: 16 bits. also known as logical ports
2. Predefined ports like http, smtp etc., (0 - 1023).
3. Registered ports (1024 - 49151)
4. Dynamic Ports - (49152 - 65535)
5. Learn about **DHCP**

35.1 Subnetting

1. Borrows bits from host-ID. Security is high.
2. Given subnet mask, we can calculate no. of subnets and no. of hosts in each subnet
3. Subnet-ID = IP Address AND subnet mask.
4. **Example**: Subnet mask of class B: 255.255.240.0. Default Mask for class B: 255.255.0.0. Subnet bits are no. of bits having 1's in the host bits of mask.
   Subnet mask = $\underbrace{255.255}_{\text{Net-ID}}.\underbrace{240}_{\text{subnet}}.\underbrace{0}_{\text{host}}$
   No. of subnets = $2^4 - 2 = 14$. no. of hosts in each subnet = $2^{12} - 2 = 4094$
5. **Example**: IP address: 197.111.121.199 & Subnet mask: 255.255.255.240. Calculate subnet-ID.
   197.111.121.199 & 255.255.255.240 = 197.111.121.192
   197.111.121.192 = 11000101.01101111.01111001.11000000. no. of 1's in host part = 4.
   Total no .of subnets = 14. Total no. of hosts = 126.
   (a) /28 means 28 1's in the subnet mask.
   (b) 1^{st} subnet id: 197.111.121.16/28(in slash notaion of CIDR) - 11000101.01101111.01111001.00010000
   (c) 2^{nd} subnet id: 197.111.121.32/28 - 11000101.01101111.01111001.00100000
   (d) 3^{rd} subnet id: 197.111.121.48/28 - 11000101.01101111.01111001.00110000
   (e) Last subnet id: 197.111.121.224/28 - 11000101.01101111.01111001.11100000
   (f) Last host of subnet id: 197.111.121.238/28 - 11000101.01101111.01111001.11101110

35.2 Classless Addressing

1. It consists of only blocks.
2. IP address given along with mask.
3. 1^{st} address of the block should be divisible by no. of addresses in the block.
4. Addresses must be contiguous.
5. Every address in a block must be a power of 2.
6. **Example**: One of the addresses in a block is 167.199.170.82/27. find no. of addresses and 1^{st} & last address.
   This block consists of No .of hosts = $2^5 = 32$. No. of address in a block is determined by subnet mask.
   1^{st} address 82 $\rightarrow$ 01010010. Replace the last 5 bits by 0's. Therefore 1^{st} address is 01000000. Last address is 01011111.
7. When all subnet bits are 0 $\rightarrow$ zero subnet.
8. When all subnet bits are 1 $\rightarrow$ all ones subnet.

Chapter 36

Physical Layer

36.1 Introduction

Definition 45 - Transmission time.

Time taken to place the data on the channel.

$$\text{Transmission Time}(T_X) = \frac{\text{Mesage size}}{\text{Bandwidth}}$$

$$\text{Transmission Time}(T_{ack}) = \frac{\text{Acknowledgement size}}{\text{Bandwidth}}$$

Definition 46 - Propagation Time.

Time taken by data to propagate to reach the destination.

$$\text{Propagation Time}(P_X) = \frac{\text{Distance}}{\text{Velocity}}$$

$$\text{Propagation Time}(P_{ack}) = \frac{\text{Distance}}{\text{Velocity}}$$

$$\text{Total Time} = T_X + P_X + Pack = T_X + 2P_X$$

$$\text{Link Utilization of Sender} = \frac{T_X}{\text{Total Time}} = \frac{\text{Message size}}{\text{Message size + Bandwidth*RTT}}$$

$$\text{Round Trip Time(RTT)} = 2 \times P_X$$

Definition 47 - Processing Time.

Time taken to generate the data.

1. At sender headers are added & at receiver headers are removed.
2. Physical address points to Data Link Layer - Bridge
3. Logical address points to Network Layer - Router
4. Port address points to Transport Layer - Gateway

Part XI

Calculus

Chapter 37

Continuity

Definition 48 - Continuity.

Let $D \in \mathbb{R}$. Consider a function $f : D \to \mathbb{R}$ and a point $c \in D$. We say that f is **continuous** at c if

$$(x_n) \text{ any sequence in } D \text{ and } x_n \to c \implies f(x_n) \to f(c).$$

If f is not continuous at c, we say that f is **discontinuous** at c. In case f is continuous at every $c \in D$, we say that f is continuous on D.

Definition 49 - Dirichlet Function.

$$f(x) = \begin{cases} 1 & \text{if } x \text{ is rational} \\ 0 & \text{if } x \text{ is irrational} \end{cases}$$

Lemma 8.1.

Let D $\subseteq \mathbb{R}, c \in D$, and let $f : D \to \mathbb{R}$ be a function that is continuous at c. If $f(c) > 0$, then there is $\delta > 0$ such that $f(x) > 0$ whenever $x \in D$ and $|x - c| < \delta$. Likewise, if $f(c) < 0$, then there is $\delta > 0$ such that $f(x) < 0$ whenever $x \in D$ and $|x - c| < \delta$.

Proposition 8.1.

Let $D \subseteq R, c \in D$, and let $f, g : D \to R$ be functions that are continuous at c. Then

1. $f + g$ is continuous at c,
2. rf is continuous at c for any $r \in R$,
3. fg is continuous at c,
4. if $f(c) = 0$, then there is $\delta > 0$ such that $f(x) = 0$ whenever $x \in D$ and $|x - c| < \delta$; moreover the function $\dfrac{1}{f} : D \cap (c - \delta, c + \delta) \to R$ is continuous at c,
5. if there is $\delta > 0$ such that $f(x) \geq 0$ whenever $x \in D$ and $|x - c| < \delta$, then for any $k \in N$, the function $f^{\frac{1}{k}} : D \cap (c - \delta, c + \delta) \to R$ is continuous at c.

Chapter 38

Differentiation

Lemma 8.2.

Let $D \subseteq R$ and c be an interior point of D. If $f : D \to R$ is differentiable at c and has a local extremum at c, then $f(c) = 0$.

Proposition 8.2 - Rolle's Theorem.

If $f : [a, b] \to R$ is continuous on $[a, b]$ and differentiable on (a, b) and if $f(a) = f(b)$, then there is $c \in (a, b)$ such that $f(c) = 0$.

Proposition 8.3 - Mean Value Theorem.

If a function $f : [a, b] \to R$ is continuous on $[a, b]$ and differentiable on (a, b), then there is $c \in (a, b)$ such that

$$f(b) - f(a) = f'(c)(b - a).$$

Chapter 39

Integration

Proposition 8.4.

Let $f : [a, b] \to R$ be a bounded function. Then for any partition P of $[a, b]$, we have

$$m(f)(b-a) \leq L(P, f) \leq U(P, f) \leq M(f)(b-a).$$

where

$$L(P, f) := \sum_{i=0}^{n} m_i(f)(x_i - x_{i-1})$$

$$U(P, f) := \sum_{i=0}^{n} M_i(f)(x_i - x_{i-1}).$$

L(P, f) = Lower Sum, U(P, f) = Upper Sum

Proposition 8.5 - Basic Inequality on Riemann's Integral.

Suppose $f : [a, b] \to R$ is an integrable function and there are $\alpha, \beta \in R$ such that $\beta \leq f \leq \alpha$. Then

$$\beta(b-a) \leq \int_a^b f(x)dx \leq \alpha(b-a).$$

. In particular, if $|f| \leq \alpha$, then

$$\int_a^b f(x)dx \leq \alpha(b-a).$$

Proposition 8.6 - Domain Additivity of Riemann Integrals.

Let $f : [a, b] \to R$ be a bounded function and let $c \in (a, b)$. Then f is integrable on $[a, b]$ if and only if f is integrable on $[a, c]$ and on $[c, b]$. In this case,

$$\int_a^b f(x)dx = \int_a^c f(x)dx + \int_c^b f(x)dx$$

Proposition 8.7.

Let $f : [a, b] \to R$ be a function.

1. If f is monotonic, then it is integrable.
2. If f is continuous, then it is integrable.

Proposition 8.8.

Let $f, g : [a, b] \to R$ be integrable functions. Then

1. $f+g$ is integrable and $\int_a^b (f+g)(x)dx = \int_a^b f(x)dx + \int_a^b g(x)dx$,
2. rf is integrable for any r âĹĹ R and a b $\int_a^b (rf)(x)dx = r\int_a^b f(x)dx$,
3. fg is integrable,
4. If there is $\delta > 0$ such that $|f(x)| \geq \delta$ and all $x \in [a,b]$, then $\frac{1}{f}$ is integrable,
5. If $f(x) \geq 0$ for all $x \in [a,b]$, then for any $k \in N$, the function $f^{\frac{1}{k}}$ is integrable.

Proposition 8.9.

Let $f, g : [a,b] \to R$ be integrable.

1. If $f \leq g$, then $\int_a^b f(x)dx \leq \int_a^b g(x)dx$.
2. The function $|f|$ is integrable and $\left|\int_a^b f(x)dx\right| \leq \int_a^b |f|(x)dx$.

Proposition 8.10 - Fundamental Theorem of Calculus.

Let $f : [a,b] \to R$ be integrable.

1. If f has an antiderivative F , then

$$\int_a^x f(t)dt = F(x) - F(a) \text{ for all } x \in [a,b]$$

.

2. Let $F : [a,b] \to R$ be defined by

$$F(x) = \int_a^x f(t)dt.$$

If f is continuous at $c \in [a,b]$, then F is differentiable at c and

$$F'(c) = f(c).$$

In particular, if f is continuous on $[a,b]$, then F is an antiderivative of f on $[a,b]$.

Proposition 8.11 - Fundamental Theorem of Riemann Integration.

Let $F : [a,b] \to R$ be a function. Then F is differentiable and F' is continuous on $[a,b]$ if and only if there is a continuous function $f : [a,b] \to R$ such that

$$F(x) = F(a) + \int_a^x f(t)dt \text{ for all } x \in [a,b].$$

In this case, we have $F'(x) = f(x)$ for all $x \in [a,b]$.

Chapter 40

Definite Integrals & Improper Integrals

Part XII

Linear Algebra

Chapter 41

Determinants

41.1 Introduction

41.2 Properties

1. $|A^{\mathrm{t}}| = |A|$
2. $|A| = 0$ if:
   (a) It has two equal lines.
   (b) All elements of a line are zero.
   (c) The elements of a line are a linear combination of the others.
3. A triangular determinant is the product of the diagonal elements.
4. If a determinant switches two parallel lines its determinant changes sign.
5. If the elements of a line are added to the elements of another parallel line previously multiplied by a real number, the value of the determinant is unchanged.
6. If a determinant is multiplied by a real number, any line can be multiplied by the above mentioned number, but only one.
7. If all the elements of a line or column are formed by two addends, the above mentioned determinant decomposes in the sum of two determinants.
8. $|A.B| = |A| \,.\, |B|$.
9. If A is invertible, then $\left|A^{-1}\right| = \dfrac{1}{|A|}$.
10. Suppose if 2 rows of a square matrix "A" are the same, then $|A| = 0$.
11. $|cA| = c^n \,|A|$ for an $n \times n$ matrix.
12. $A^{-1} = \dfrac{\mathrm{adj(A)}}{|A|}$.
13. Vandermode determinant: $\begin{vmatrix} 1 & 1 & 1 & \cdots & 1 \\ x_1 & x_2 & x_3 & \cdots & x_n \\ x_1^2 & x_2^2 & x_3^2 & \cdots & x_n^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1^{n-1} & x_2^{n-1} & x_3^{n-1} & \cdots & x_n^{n-1} \end{vmatrix} = \prod_{1 \le i < j \le n} (x_j - x_i)$

14. Circulant Matrix: $\begin{vmatrix} x_1 & x_2 & x_3 & \cdots & x_n \\ x_n & x_1 & x_2 & \cdots & x_{n-1} \\ x_{n-1} & x_n & x_1 & \cdots & x_{n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_2 & x_3 & x_4 & \cdots & x_1 \end{vmatrix} = \prod_{j=1}^{n} \left(x_1 + x_2\omega_j + x_3\omega_j^2 + \ldots + x_n\omega_j^{n-1}\right)$
where ω_j is an n^{th} root of 1.

15. $|A| = \left|A^T\right| = |-A| = (1)^n |A|$. Hence $det(A) = 0$ when n is odd.

41.3 Determinants & Eigenvalues

1. $|A - xI| = 0$.
2. A symmetric $n \times n$ real matrix M is said to be **positive definite** if $z^T Mz$ is positive for every non-zero column vector z of n real numbers. Here z^T denotes the transpose of z.
3. The negative definite, positive semi-definite, and negative semi-definite matrices are defined in the same way, except that the expression $z^T Mz$ or $z*Mz$ is required to be always negative, non-negative, and non-positive, respectively.
4. Eigenvalues of Hermitian Matrices($A = \overline{A^{\mathrm{T}}}$) are always real.
5. For any real non-singular matrix A, the product $A^T A$ is a **positive definite** matrix.
6. All eigenvalues of positive definite matrix are positive.
7. Every positive definite matrix is invertible and its inverse is also positive definite.
8. If M is positive definite and $r > 0$ is a real number, then rM is positive definite. If M and N are positive definite, then the sum $M + N$ and the products MNM and NMN are also positive definite. If $MN = NM$, then MN is also positive definite.($M\&N$ are square matrices.)
9. Every principal submatrix of a positive definite matrix is positive definite.
10. The trace tr(A) is by definition the sum of the diagonal entries of A and also equals the sum of the eigenvalues.
11. $\mathrm{tr}(A) = \log(|(\exp(A))|)$.,

41.4 Eigenvectors & Eigenvalues

1. Eigenvectors with Distinct Eigenvalues are Linearly Independent.
2. Singular Matrices have Zero Eigenvalues. ie, eigenvalue = 0.
3. Suppose A is a square matrix and λ is an eigenvalue of A. Then $\alpha\lambda$ is an eigenvalue of αA.
4. Suppose A is a square matrix, λ is an eigenvalue of A, and $s \geq 0$ is an integer. Then λ^s is an eigenvalue of A^s.
5. Suppose A is a square matrix and λ is an eigenvalue of A. Let q(x) be a polynomial in the variable x. Then q(λ) is an eigenvalue of the matrix q(A).
6. Suppose A is a square nonsingular matrix and λ is an eigenvalue of A. Then λ^{-1} is an eigenvalue of the matrix A^{-1}.

7. Suppose A is a square matrix and λ is an eigenvalue of A. Then λ is an eigenvalue of the matrix A^t.

8. Suppose A is a square matrix with real entries and x is an eigenvector of A for the eigenvalue λ. Then $\bar{x}$ is an eigenvector of A for the eigenvalue $\bar{\lambda}$.

9. Hermitian Matrices have Real Eigenvalues, orthogonal eigenvectors.

10. An eigenvalue λ has algebraic multiplicity $l > 0$ if $\det(A - \lambda I) = 0$.

11. Characteristic Polynomial: $\det(A - \lambda I) = 0$.

12. An $n \times n$ symmetric matrix has n orthogonal eigenvectors($X^T Y = 0$) for distinct eigenvalues.

13. Symmetric matrices only have real eigenvalues.

14. **Geometric multiplicity** $\gamma_T(\lambda)$ of an eigenvalue λ is the dimension of the eigenspace associated to λ, i.e. number of linearly independent eigenvectors with that eigenvalue.

15. Let A be an arbitrary $n \times n$ matrix of complex numbers with eigenvalues $\lambda_1, \lambda_2, ...\lambda_n$.

    (a) Every eigenvalue of a unitary matrix has absolute value $|\lambda| = 1$.

    (b) If A is invertible, then the eigenvalues of A^{-1} are $1/\lambda_1, 1/\lambda_2, \ldots, 1/\lambda_n$.

    (c) The eigenvalues of the k^{th} power of A, i.e. the eigenvalues of A^k, for any positive integer k, are $\lambda_1^k, \lambda_2^k, \ldots, \lambda_n^k$ with same eigenvectors.

    (d) The matrix A is invertible if and only if all the eigenvalues λ_i are nonzero.

    (e) The trace of A, defined as the sum of its diagonal elements, is also the sum of all eigenvalues: $\operatorname{tr}(A) = \sum_{i=1}^n A_{ii} = \sum_{i=1}^n \lambda_i = \lambda_1 + \lambda_2 + \cdots + \lambda_n$.

    (f) The determinant of A is the product of all eigenvalues: $\det(A) = \prod_{i=1}^n \lambda_i = \lambda_1 \lambda_2 \cdots \lambda_n$.

    (g) New Item

16. The eigenvalues of a diagonal or triangular matrix are its diagonal elements.

17. An $n \times n$ matrix is invertible if and only if it doesn't have 0 as an eigenvalue.

18. Row reductions don't preserve eigenvalues . However, similar matrices have the same characteristic polynomial, so theyhave the same eigenvalues with the same multiplicities.

19. A complete set of eigenvectors for $A_{n\times n}$ is any set of n linearly independent eigenvectors for A.

20. A $n\times n$ is diagonalizable if and only if A possesses a complete set ofeigenvectors. Moreover,$P^1 AP = diag(\lambda_1, \lambda_2, \ldots, \lambda_n)$ if and only if the columns of P constitute a complete set of eigenvectors and the λ_j's are the associated eigenvalues.

21. **Cayley Hamilton Theorem**: Every square matrix satisfiesits own characteristic equation $p(\lambda) = 0$.That is, $p(A) = 0$.

22. If no eigenvalue of A is repeated, then A is diagonalizable. Converse is not true.

41.5 Cramer's rule

A linear system of m equations with n unknowns $x_1, x_2, \ldots, x_n$.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \vdots + \quad \vdots + \quad \vdots + \quad \vdots &= \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &= b_m \end{aligned}$$

Solution:
$x_1 = \dfrac{A_1}{A}$, ... $x_i = \dfrac{A_i}{A}$ ..., $x_n = \dfrac{A_n}{A}$ where

$$A_i = \begin{vmatrix} a_{11} & a_{12} & \ldots & b_1 & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & b_2 & \ldots & a_{2n} \\ \vdots & \vdots & \ldots & \vdots & \ldots & \vdots \\ a_{m1} & a_{m2} & \ldots & b_n & \ldots & a_{mn} \end{vmatrix}$$

where b_i's are present in i^{th} column.

41.6 Rank of a Matrix

1. The rank of a matrix is the number of pivots when in row echelon form. A pivot is the first non zero entry in a row.
2. Rank of diagonal matrix is the no. of non-zero elements in the principal diagonal of the matrix.
3. Rank of A $(m \times n)$ matrix is $\text{rank}(A) \leq (m, n)$.
4. A square matrix A of order n is invertible if rank(A) = n.
5. Rank(A) + nullity(A) = no. of columns in A.
6. Row equivalent matrices have same rank.

41.7 System of Linear Equations

A linear system of m equations with n unknowns $x_1, x_2, \ldots, x_n$.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \vdots + \quad \vdots + \quad \vdots + \quad \vdots &= \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &= b_m \end{aligned}$$

is consistent if and only if A and $\tilde{A}$ have the same rank.

$$\text{A} = \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ a_{21} & \ldots & a_{2n} \\ \vdots & \ldots & \vdots \\ a_{m1} & \ldots & a_{mn} \end{bmatrix} \tilde{A} = \left[\begin{array}{ccc|c} a_{11} & \ldots & a_{1n} & b_1 \\ a_{21} & \ldots & a_{2n} & b_2 \\ \vdots & \ldots & \vdots & \vdots \\ a_{m1} & \ldots & a_{mn} & b_m \end{array}\right]$$

1. **Unique Solution**: If and only if common rank of A & $\tilde{A}$ is same and is equal to n.
2. **No Solution**: $\text{rank}(A) \neq \text{rank}(\tilde{A})$
3. **Infinitely many Solution**: If and only if common rank of A & $\tilde{A}$ is less than n.
4. For homogenous systems(all b_i's are 0), there exists non-trivial solution if $\text{rank}(A) < n$.
5. Let $Ax = b$ be a consistent system of equations with n variables. Then number of free variables is equal to $n - \text{rank}(A)$.

Check this part for m, n comaprison condition for no, infintely many solutions. Also for $m = n$.

Part XIII

Set Theory & Algebra

Chapter 42

Sets

Chapter 43

Relations

Definition 50 - Cartesian Product.

The Cartesian product of two sets A and B (also called the product set, set direct product, or cross product) is defined to be the set of all points (a, b) where $a \in A$ and $b \in B$. It is denoted by $A \times B$. Cartesian Product is not commutaitve ie., $A \times B \neq B \times A$.

Definition 51 - Relation.

A relation is any subset of a Cartesian product. $R \circ S = (x, z)|(x, y) \in S, (y, z) \in R.R \circ S \neq S \circ R.R \circ (S \circ T) = (R \circ S) \circ T.R^{-1} \neq R \Rightarrow Ris\text{Symmetric}.R^{-1} \neq \bar{R}$

Definition 52 - Reflexive Closure.

The reflexive closure S of a relation R on a set X is given by

$$S = R \cup \{(x, x) : x \in X\}$$

Definition 53 - Symmetric Closure.

The symmetric closure S of a relation R on a set X is given by

$$S = R \cup \{(x, y) : (y, x) \in R\}.$$

Definition 54 - Transitive Closure.

The transitive closure of R is then given by the intersection of all transitive relations containing R. Warshall Alorithm is used to find **transitive closure**.
R^+ is representation for transitive closure.

$$R^+ = \bigcup_{i \in \{1,2,3,\ldots\}} R^i.$$

where R^i is the i^{th} power of R, defined inductively by $R^1 = R$ and, for $i > 0$, $R^{i+1} = R \circ R^i$

Definition 55 - Reachability Relation.

Reachability refers to the ability to get from one vertex to another within a graph. Floyd Warshall Algorithm used to find it.

$$R^\infty = R^+ \cup I$$

. contains self loops. $M_{R \circ S} = M_S \odot M_R$ where $\odot$ is binary multiplication.

Definition 56 - Reachability Matrix.

$$R^k = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ \vdots & \vdots & \dots & \vdots \end{bmatrix}$$

where a_{1n} indicates $\exists$ a path of length k from 1 to n.

Theorem 9.

If R(Relation) is symmetric, transitive, reflexive, equivalence then R^{-1} is also symmetric, transitive, reflexive, equivalence.

Definition 57 - Binary Relation.

A relation on set A to set B. $R \subseteq A_1 \times A_2$.

43.1 Properties

1. **Reflexive Relation** $\forall x \in S, xRx$.
2. **Symmetric Relation** $\forall x, y \in S, xRy \rightarrow yRx$.
3. **Transitive Relation** $\forall x, y, z \in S, (xRy \wedge yRz) \rightarrow xRz$.
4. **Equivalence Relation** All of reflexive, symmetric & transitive relation.
5. If R and S are equivalence relations, then $R \cup S$ is also equivalence relation, not true for $R \cup S$.

43.2 Other Properties Names

1. **Asymmetry** $\neg\exists x, y \in S, xRy \wedge yRx$
2. **AntiSymmetric** $\forall x, y \in S, xRy \wedge yRx \rightarrow x = y$.
3. **Irreflexive** $\neg\exists x \in S, xRx$.

Chapter 44

Functions

Chapter 45

Group

Chapter 46

Lattices

Chapter 47

Partial Orders

Definition 58 - Weak Partial Order.

A relation R on a set A is a **weak partial order** if it is transitive, antisymmetric, and reflexive.

Definition 59 - Strong Partial Order.

The relation is said to be a strong partial order if it is transitive, antisymmetric, and irreflexive.

Definition 60 - Total Order.

A total order is a partial order in which every pair of distinct elements is comparable. Comparable elements $A, B \iff A \leq B \cup B \leq A$.

Theorem 10.

A poset has no directed cycles other than self-loops. Partially ordered set is a digraph without cycles (DAG).

Definition 61.

Given a digraph $G = (V, E)$, the transitive closure of G is the digraph $G^+ = (V, E^+)$ where $E^+ = u \to v \mid$ there is a directed path of positive length from u to v in G.

Definition 62 - Lattices.

A lattice is a partially ordered set in which every two elements have a supremum (also called a least upper bound or join) and an infimum (also called a greatest lower bound or meet).
An example is given by the natural numbers, partially ordered by divisibility, for which the supremum is the least common multiple and the infimum is the greatest common divisor.

Part XIV

Combinatory

Chapter 48

Generating Functions

Definition 63 - Generating functions.

Generating Function for the sequence $\langle g_0, g_1, g_2, g_3, \ldots \rangle$ is the power series:

$$G(x) = g_0 + g_1 x + g_2 x^2 + g_3 x^3 + \ldots$$

Chapter 49

Recurrence Relations

Determinate Order Recurrence Relations $a_n = k_1 a_{n-1} + k_2 an - 2 + k_3$ where k_1, k_2, k_3 are constants and the relation is of 2^{nd} order. **Determinate Order Recurrence Relations** $a_n = k_1 a_{\frac{n}{2}} + k_2$ where k_1, k_2 are constants. $a_n = k_1 a_{n-1}^2 + k_2 an - 2 + k_2$ Non-Linear Order
$a_n = k_1 a_{n-1} + k_2 an - 2 + k_2$ Linear Order
Example: $a_n - 5a_{n-1} + 6a_{n-2} = 0$ $a_n = f(n) = x^n$. No. of constants $\rightarrow$ Order of relation.
Solution: $a_n = c_1 2^n + c_2 3^n$. c_1, c_2 can be obtained by inital conditions given like $a_0 = 1, a_1 = 2$.
When roots are same $a_n = c_1 x_1^n + c_2 n x_2^n + c_3 n^2 x_3^n + \ldots$.
$a_n = a\,\overline{\frac{n}{r}} + k_1$ where k_1 is a constant. $n = r^s$. $\frac{n}{r} = r^{s-1}$. $a_{r^s} = a_{r^{s-1}} + k_1$. Let $b_s = a_{r^s}$. Solve for b_s and then replace it with a_n.
For particular solution $a = a^h + a^p$ where $a^p = d_1 a^{p1}$ where d_1 is a constant.

Part XV

C Programs for Algorithms I [6]

Chapter 50

Sorting

```c
#include <stdio.h>

// A recursive binary search function. It returns location of x in
// given array arr[l..r] is present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
   if (r >= l)
   {
        int mid = l + (r - l)/2;

        // If the element is present at the middle itself
        if (arr[mid] == x)  return mid;

        // If element is smaller than mid, then it can only be present
        // in left subarray
        if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);

        // Else the element can only be present in right subarray
        return binarySearch(arr, mid+1, r, x);
   }

   // We reach here when element is not present in array
   return -1;
}

int main(void)
{
   int arr[] = {2, 3, 4, 10, 40};
   int n = sizeof(arr)/ sizeof(arr[0]);
   int x = 10;
   int result = binarySearch(arr, 0, n-1, x);
   (result == -1)? printf("Element is not present in array")
```

```
                    : printf("Element is present at index %d", result);
    return 0;
}
```

Listing 1: Binary Search in C (Recursive)

```c
#include <stdio.h>

// A iterative binary search function. It returns location of x in
// given array arr[l..r] if present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
  while (l <= r)
  {
    int m = l + (r-l)/2;

    if (arr[m] == x) return m;  // Check if x is present at mid

    if (arr[m] < x) l = m + 1; // If x greater, ignore left half

    else r = m - 1; // If x is smaller, ignore left half
  }
  return -1; // if we reach here, then element was not present
}

int main(void)
{
   int arr[] = {2, 3, 4, 10, 40};
   int n = sizeof(arr)/ sizeof(arr[0]);
   int x = 10;
   int result = binarySearch(arr, 0, n-1, x);
   (result == -1)? printf("Element is not present in array")
                 : printf("Element is present at index %d", result);
   return 0;
}
```

Listing 2: Binary Search in C (Iterative)

```c
// C program for insertion sort
#include <stdio.h>
#include <math.h>

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
   int i, key, j;
   for (i = 1; i < n; i++)
   {
       key = arr[i];
       j = i-1;

       /* Move elements of arr[0..i-1], that are
          greater than key, to one position ahead
          of their current position */
       while (j >= 0 && arr[j] > key)
       {
           arr[j+1] = arr[j];
           j = j-1;
       }
       arr[j+1] = key;
   }
}

// A utility function ot print an array of size n
void printArray(int arr[], int n)
{
   int i;
   for (i=0; i < n; i++)
       printf("%d ", arr[i]);
   printf("\n");
}



/* Driver program to test insertion sort */
int main()
{
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

Listing 3: Insertion Sort in C

```c
// C program for implementation of Bubble sort
#include <stdio.h>

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n; i++)
       for (j = 0; j < n-i-1; j++) //Last i elements are already in place
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Listing 4: Bubble Sort in C

```c
/* C program for merge sort */
#include<stdlib.h>
#include<stdio.h>

/* Function to merge the two haves arr[l..m] and arr[m+1..r] of array arr[] */
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for(i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for(j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there are any */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
```

```
    /* Copy the remaining elements of R[], if there are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

/* l is for left index and r is right index of the sub-array
  of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l+(r-l)/2; //Same as (l+r)/2, but avoids overflow for large l and h
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);
        merge(arr, l, m, r);
    }
}


/* UITLITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}
```

Listing 5: Merge Sort in C

```c
/* A typical recursive implementation of quick sort */
#include<stdio.h>

// A utility function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function takes last element as pivot, places the pivot element at its
   correct position in sorted array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right of pivot */
int partition (int arr[], int l, int h)
{
    int x = arr[h];    // pivot
    int i = (l - 1);  // Index of smaller element

    for (int j = l; j <= h- 1; j++)
    {
        // If current element is smaller than or equal to pivot
        if (arr[j] <= x)
        {
            i++;    // increment index of smaller element
            swap(&arr[i], &arr[j]);  // Swap current element with index
        }
    }
    swap(&arr[i + 1], &arr[h]);
    return (i + 1);
}

/* arr[] --> Array to be sorted, l  --> Starting index, h  --> Ending index */
void quickSort(int arr[], int l, int h)
{
    if (l < h)
    {
        int p = partition(arr, l, h); /* Partitioning index */
        quickSort(arr, l, p - 1);
        quickSort(arr, p + 1, h);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
```

```c
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    quickSort(arr, 0, n-1);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Listing 6: Quick Sort in C

```c
// An iterative implementation of quick sort
#include <stdio.h>

int n;

// A utility function to print contents of arr
void printArr( int arr[], int n )
{
    int i;
    for ( i = 0; i < n; ++i )
        printf( "%d ", arr[i] );
    printf("\n");
}

// A utility function to swap two elements
void swap ( int* a, int* b )
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function is same in both iterative and recursive*/
int partition (int arr[], int l, int h)
{
    int x = arr[h];
    int i = (l - 1);
    //printf("Pivot: %d, i = %d, h = %d\n", x, i, h);
    for (int j = l; j <= h- 1; j++)
    {
        if (arr[j] <= x)
        {
            i++;
            swap (&arr[i], &arr[j]);
        }
        //printf("Partition: i = %d, j = %d\n",i,j);
        printArr( arr, n);
    }
    swap (&arr[i + 1], &arr[h]);
    return (i + 1);
}

/* A[] --> Array to be sorted, l  --> Starting index, h  --> Ending index */
void quickSortIterative (int arr[], int l, int h)
{
    // Create an auxiliary stack
    int stack[ h - l + 1 ];
```

```
    // initialize top of stack
    int top = -1;

    // push initial values of l and h to stack
    stack[ ++top ] = l;
    stack[ ++top ] = h;

    // Keep popping from stack while is not empty
    while ( top >= 0 )
    {
        // Pop h and l
        h = stack[ top-- ];
        l = stack[ top-- ];

        // Set pivot element at its correct position in sorted array
        int p = partition( arr, l, h );
        //printf("After Partition:\n");
        printArr( arr, n);
        // If there are elements on left side of pivot, then push left
        // side to stack
        if ( p-1 > l )
        {
            stack[ ++top ] = l;
            stack[ ++top ] = p - 1;
        }

        // If there are elements on right side of pivot, then push right
        // side to stack
        if ( p+1 < h )
        {
            stack[ ++top ] = p + 1;
            stack[ ++top ] = h;
        }
    }
}

// Driver program to test above functions
int main()
{
    int arr[] = {4, 3, 5, 2, 1, 3, 2, 3};
    n = sizeof( arr ) / sizeof( *arr );
    quickSortIterative( arr, 0, n - 1 );
    printf("\n");
    printArr( arr, n );
    return 0;
}
```

Listing 7: Quick Sort in C (Iterative)

```c
// C implementation of Heap Sort
#include <stdio.h>
#include <stdlib.h>

// A heap has current size and array of elements
struct MaxHeap
{
    int size;
    int* array;
};

// A utility function to swap to integers
void swap(int* a, int* b) { int t = *a; *a = *b;  *b = t; }

// The main function to heapify a Max Heap. The function assumes that
// everything under given root (element at index idx) is already heapified
void maxHeapify(struct MaxHeap* maxHeap, int idx)
{
    int largest = idx;  // Initialize largest as root
    int left = (idx << 1) + 1;  // left = 2*idx + 1
    int right = (idx + 1) << 1; // right = 2*idx + 2

    // See if left child of root exists and is greater than root
    if (left < maxHeap->size && maxHeap->array[left] > maxHeap->array[largest])
        largest = left;

    // See if right child of root exists and is greater than the largest so far
    if (right < maxHeap->size && maxHeap->array[right] > maxHeap->array[largest])
        largest = right;

    // Change root, if needed
    if (largest != idx)
    {
        swap(&maxHeap->array[largest], &maxHeap->array[idx]);
        maxHeapify(maxHeap, largest);
    }
}

// A utility function to create a max heap of given capacity
struct MaxHeap* createAndBuildHeap(int *array, int size)
{
    int i;
    struct MaxHeap* maxHeap = (struct MaxHeap*) malloc(sizeof(struct MaxHeap));
    maxHeap->size = size;   // initialize size of heap
    maxHeap->array = array; // Assign address of first element of array

    // Start from bottommost and rightmost internal mode and heapify all
```

```
    // internal modes in bottom up way
    for (i = (maxHeap->size - 2) / 2; i >= 0; --i){
        printf("\nHeapifying at index %d\n",i);
            maxHeapify(maxHeap, i);
    }
    return maxHeap;
}

// The main function to sort an array of given size
void heapSort(int* array, int size)
{
    // Build a heap from the input data.
    struct MaxHeap* maxHeap = createAndBuildHeap(array, size);

    // Repeat following steps while heap size is greater than 1. The last
    // element in max heap will be the minimum element
    while (maxHeap->size > 1)
    {
       // The largest item in Heap is stored at the root. Replace it with the
       // last item of the heap followed by reducing the size of heap by 1.
        swap(&maxHeap->array[0], &maxHeap->array[maxHeap->size - 1]);
        --maxHeap->size;  // Reduce heap size

        // Finally, heapify the root of tree.
        maxHeapify(maxHeap, 0);
    }
}

// A utility function to print a given array of given size
void printArray(int* arr, int size)
{
    int i;
    for (i = 0; i < size; ++i)
        printf("%d ", arr[i]);
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7, 20, 1, 10};
    int size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, size);

    heapSort(arr, size);

    printf("\nSorted array is \n");
```

```
    printArray(arr, size);
    return 0;
}
```

Listing 8: Heap Sort in C

```cpp
// C++ program to sort an array using bucket sort
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

// Function to sort arr[] of size n using bucket sort
void bucketSort(float arr[], int n)
{
    // 1) Create n empty buckets
    vector<float> b[n];

    // 2) Put array elements in different buckets
    for (int i=0; i<n; i++)
    {
       int bi = n*arr[i]; // Index in bucket
       b[bi].push_back(arr[i]);
    }

    // 3) Sort individual buckets
    for (int i=0; i<n; i++)
       sort(b[i].begin(), b[i].end());

    // 4) Concatenate all buckets into arr[]
    int index = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < b[i].size(); j++)
          arr[index++] = b[i][j];
}

/* Driver program to test above funtion */
int main()
{
    float arr[] = {0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434};
    int n = sizeof(arr)/sizeof(arr[0]);
    bucketSort(arr, n);

    cout << "Sorted array is \n";
    for (int i=0; i<n; i++)
       cout << arr[i] << " ";
    return 0;
}
```

Listing 9: Bucket Sort in C++

Chapter 51

Spanning Trees

```c
// Kruskal's algortihm to find Minimum Spanning Tree of a given connected,
// undirected and weighted graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, undirected and weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges. Since the graph is
    // undirected, the edge from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );
```

```
    return graph;
}

// A structure to represent a subset for union-find
struct subset
{
    int parent;
    int rank;
};

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree
    // (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and increment
    // its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
```

```
{
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST(struct Graph* graph)
{
    int V = graph->V;
    struct Edge result[V];  // Tnis will store the resultant MST
    int e = 0;  // An index variable, used for result[]
    int i = 0;  // An index variable, used for sorted edges

    // Step 1:  Sort all the edges in non-decreasing order of their weight
    // If we are not allowed to change the given graph, we can create a copy of
    // array of edges
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

    // Allocate memory for creating V ssubsets
    struct subset *subsets =
        (struct subset*) malloc( V * sizeof(struct subset) );

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1)
    {
        // Step 2: Pick the smallest edge. And increment the index
        // for next iteration
        struct Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge does't cause cycle, include it
        // in result and increment the index of result for next edge
        if (x != y)
        {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
        // Else discard the next_edge
```

```
    }

    // print the contents of result[] to display the built MST
    printf("Following are the edges in the constructed MST\n");
    for (i = 0; i < e; ++i)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest,
                                                  result[i].weight);
    return;
}

// Driver program to test above functions
int main()
{
    /* Let us create following weighted graph
             10
        0--------1
        |  \     |
       6|   5\   |15
        |      \ |
        2--------3
            4       */
    int V = 4;  // Number of vertices in graph
    int E = 5;  // Number of edges in graph
    struct Graph* graph = createGraph(V, E);


    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 15;

    // add edge 2-3
    graph->edge[4].src = 2;
```

```
    graph->edge[4].dest = 3;
    graph->edge[4].weight = 4;

    KruskalMST(graph);

    return 0;
}
```

Listing 10: Kruskal MST in C

Chapter 52

Dynamic Programming

```c
#include<stdio.h>

/* simple recursive program for Fibonacci numbers */
int fib(int n)
{
   if ( n <= 1 )
      return n;
   return fib(n-1) + fib(n-2);
}

int main(){
        printf("Fibonacci of 4 is %d", fib(4));
        return 0;
}
```

Listing 11: Fibonacci Numbers in C

```c
/* Memoized version for nth Fibonacci number */
#include<stdio.h>
#define NIL -1
#define MAX 100

int lookup[MAX];

/* Function to initialize NIL values in lookup table */
void _initialize()
{
  int i;
  for (i = 0; i < MAX; i++)
    lookup[i] = NIL;
}

/* function for nth Fibonacci number */
int fib(int n)
{
   if(lookup[n] == NIL)
   {
    if ( n <= 1 )
      lookup[n] = n;
    else
      lookup[n] = fib(n-1) + fib(n-2);
   }

   return lookup[n];
}

int main ()
{
  int n = 40;
  _initialize();
  printf("Fibonacci number is %d ", fib(n));
  getchar();
  retur
```

Listing 12: Fibonacci Numbers in C(Memoization)

```c
/* tabulated version */
#include<stdio.h>
int fib(int n)
{
  int f[n+1];
  int i;
  f[0] = 0;   f[1] = 1;
  for (i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];

  return f[n];
}

int main ()
{
  int n = 9;
  printf("Fibonacci number is %d ", fib(n));
  getchar();
  return 0;
}
```

Listing 13: Fibonacci Numbers in C(Tabulation)

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

/*This function divides a by greatest divisible
  power of b*/
int maxDivide(int a, int b)
{
  while (a%b == 0)
   a = a/b;
  return a;
}

/* Function to check if a number is ugly or not */
int isUgly(int no)
{
  no = maxDivide(no, 2);
  no = maxDivide(no, 3);
  no = maxDivide(no, 5);

  return (no == 1)? 1 : 0;
}

/* Function to get the nth ugly number*/
int getNthUglyNo(int n)
{
  int i = 1;
  int count = 1;   /* ugly number count */

  /*Check for all integers untill ugly count
    becomes n*/
  while (n > count)
  {
    i++;
    if (isUgly(i))
      count++;
  }
  return i;
}

/* Driver program to test above functions */
int main()
{
    unsigned no = getNthUglyNo(150);
    printf("150th ugly no. is %d ",  no);
    getchar();
    return 0;
}
```

Listing 14: Ugly Numbers Numbers in C

```
# include<stdio.h>
# include<stdlib.h>
# define bool int

/* Function to find minimum of 3 numbers */
unsigned min(unsigned , unsigned , unsigned );

/* Function to get the nth ugly number*/
unsigned getNthUglyNo(unsigned n)
{
    unsigned *ugly =
             (unsigned *)(malloc (sizeof(unsigned)*n));
    unsigned i2 = 0, i3 = 0, i5 = 0;
    unsigned i;
    unsigned next_multiple_of_2 = 2;
    unsigned next_multiple_of_3 = 3;
    unsigned next_multiple_of_5 = 5;
    unsigned next_ugly_no = 1;
    *(ugly+0) = 1;

    for(i=1; i<n; i++)
    {
       next_ugly_no = min(next_multiple_of_2,
                          next_multiple_of_3,
                          next_multiple_of_5);
       *(ugly+i) = next_ugly_no;
       if(next_ugly_no == next_multiple_of_2)
       {
           i2 = i2+1;
           next_multiple_of_2 = *(ugly+i2)*2;
       }
       if(next_ugly_no == next_multiple_of_3)
       {
           i3 = i3+1;
           next_multiple_of_3 = *(ugly+i3)*3;
       }
       if(next_ugly_no == next_multiple_of_5)
       {
           i5 = i5+1;
           next_multiple_of_5 = *(ugly+i5)*5;
       }
    } /*End of for loop (i=1; i<n; i++) */
    return next_ugly_no;
}

/* Function to find minimum of 3 numbers */
unsigned min(unsigned a, unsigned b, unsigned c)
```

```
{
    if(a <= b)
    {
      if(a <= c)
        return a;
      else
        return c;
    }
    if(b <= c)
      return b;
    else
      return c;
}

/* Driver program to test above functions */
int main()
{
    unsigned no = getNthUglyNo(150);
    printf("%dth ugly no. is %d ", 150, no);
    getchar();
    return 0;
}
```

Listing 15: Ugly Numbers in C(Dynamic Programming)

```c
/* A Naive recursive implementation of LIS problem */
#include<stdio.h>
#include<stdlib.h>

/* To make use of recursive calls, this function must return two things:
   1) Length of LIS ending with element arr[n-1]. We use max_ending_here
      for this purpose
   2) Overall maximum as the LIS may end with an element before arr[n-1]
      max_ref is used this purpose.
The value of LIS of full array of size n is stored in *max_ref which is our final result
*/
int _lis( int arr[], int n, int *max_ref)
{
    /* Base case */
    if(n == 1)
        return 1;

    int res, max_ending_here = 1; // length of LIS ending with arr[n-1]

    /* Recursively get all LIS ending with arr[0], arr[1] ... ar[n-2]. If
       arr[i-1] is smaller than arr[n-1], and max ending with arr[n-1] needs
       to be updated, then update it */
    for(int i = 1; i < n; i++)
    {
        res = _lis(arr, i, max_ref);
        if (arr[i-1] < arr[n-1] && res + 1 > max_ending_here)
            max_ending_here = res + 1;
    }

    // Compare max_ending_here with the overall max. And update the
    // overall max if needed
    if (*max_ref < max_ending_here)
       *max_ref = max_ending_here;

    // Return length of LIS ending with arr[n-1]
    return max_ending_here;
}

// The wrapper function for _lis()
int lis(int arr[], int n)
{
    // The max variable holds the result
    int max = 1;

    // The function _lis() stores its result in max
    _lis( arr, n, &max );
```

```
    // returns max
    return max;
}

/* Driver program to test above function */
int main()
{
    int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of LIS is %d\n",  lis( arr, n ));
    getchar();
    return 0;
}
```

Listing 16: Longest Increasing Subsequence in C

```c
/* Dynamic Programming implementation of LIS problem */
#include<stdio.h>
#include<stdlib.h>

/* lis() returns the length of the longest increasing subsequence in
    arr[] of size n */
int lis( int arr[], int n )
{
   int *lis, i, j, max = 0;
   lis = (int*) malloc ( sizeof( int ) * n );

   /* Initialize LIS values for all indexes */
   for ( i = 0; i < n; i++ )
      lis[i] = 1;

   /* Compute optimized LIS values in bottom up manner */
   for ( i = 1; i < n; i++ )
      for ( j = 0; j < i; j++ )
         if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)
            lis[i] = lis[j] + 1;

   /* Pick maximum of all LIS values */
   for ( i = 0; i < n; i++ )
      if ( max < lis[i] )
         max = lis[i];

   /* Free memory to avoid memory leak */
   free( lis );

   return max;
}

/* Driver program to test above function */
int main()
{
  int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
  int n = sizeof(arr)/sizeof(arr[0]);
  printf("Length of LIS is %d\n", lis( arr, n ) );

  getchar();
  return 0;
}
```

Listing 17: Longest Increasing Subsequence in C(Dynamic Programming)

```c
/* A Naive recursive implementation of LCS problem */
#include<stdio.h>
#include<stdlib.h>

int max(int a, int b);

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
   if (m == 0 || n == 0)
     return 0;
   if (X[m-1] == Y[n-1])
     return 1 + lcs(X, Y, m-1, n-1);
   else
     return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Driver program to test above function */
int main()
{
  char X[] = "AGGTAB";
  char Y[] = "GXTXAYB";

  int m = strlen(X);
  int n = strlen(Y);

  printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );

  getchar();
  return 0;
}
```

Listing 18: Longest Common Subsequence in C

Listing 19: Longest Common Subsequence in C(Dynamic Programming)

```c
// Dynamic Programming implementation of edit distance
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

// Change these strings to test the program
#define STRING_X "SUNDAY"
#define STRING_Y "SATURDAY"

#define SENTINEL (-1)
#define EDIT_COST (1)

#define MIN(a, b) a>b?a:b
inline
int min(int a, int b) {
   return a < b ? a : b;
}

// Returns Minimum among a, b, c
int Minimum(int a, int b, int c)
{
    return MIN(MIN(a, b), c);
}

// Strings of size m and n are passed.
// Construct the Table for X[0...m, m+1], Y[0...n, n+1]
int EditDistanceDP(char X[], char Y[])
{
    // Cost of alignment
    int cost = 0;
    int leftCell, topCell, cornerCell;

    int m = strlen(X)+1;
    int n = strlen(Y)+1;

    // T[m][n]
    int *T = (int *)malloc(m * n * sizeof(int));

    // Initialize table
    for(int i = 0; i < m; i++)
        for(int j = 0; j < n; j++)
            *(T + i * n + j) = SENTINEL;

    // Set up base cases
    // T[i][0] = i
    for(int i = 0; i < m; i++)
        *(T + i * n) = i;
```

```
    // T[0][j] = j
    for(int j = 0; j < n; j++)
        *(T + j) = j;

    // Build the T in top-down fashion
    for(int i = 1; i < m; i++)
    {
        for(int j = 1; j < n; j++)
        {
            // T[i][j-1]
            leftCell = *(T + i*n + j-1);
            leftCell += EDIT_COST; // deletion

            // T[i-1][j]
            topCell = *(T + (i-1)*n + j);
            topCell += EDIT_COST; // insertion

            // Top-left (corner) cell
            // T[i-1][j-1]
            cornerCell = *(T + (i-1)*n + (j-1) );

            // edit[(i-1), (j-1)] = 0 if X[i] == Y[j], 1 otherwise
            cornerCell += (X[i-1] != Y[j-1]); // may be replace

            // Minimum cost of current cell
            // Fill in the next cell T[i][j]
            *(T + (i)*n + (j)) = Minimum(leftCell, topCell, cornerCell);
        }
    }

    // Cost is in the cell T[m][n]
    cost = *(T + m*n - 1);
    free(T);
    return cost;
}

// Recursive implementation
int EditDistanceRecursion( char *X, char *Y, int m, int n )
{
    // Base cases
    if( m == 0 && n == 0 )
        return 0;

    if( m == 0 )
        return n;

    if( n == 0 )
```

```
            return m;

        // Recurse
        int left = EditDistanceRecursion(X, Y, m-1, n) + 1;
        int right = EditDistanceRecursion(X, Y, m, n-1) + 1;
        int corner = EditDistanceRecursion(X, Y, m-1, n-1) + (X[m-1] != Y[n-1]);

        return Minimum(left, right, corner);
    }

    int main()
    {
        char X[] = STRING_X; // vertical
        char Y[] = STRING_Y; // horizontal

        printf("Minimum edits required to convert %s into %s is %d\n",
               X, Y, EditDistanceDP(X, Y) );
        printf("Minimum edits required to convert %s into %s is %d by recursion\n",
               X, Y, EditDistanceRecursion(X, Y, strlen(X), strlen(Y)));

        return 0;
    }
```

Listing 20: Edit Distance in C

Part XVI

C Programs for Testing

```c
#include<stdio.h>

int main(){
    if(sizeof(int) > -1){
        printf("True");
    }
    else{
        printf("False");
    }
    return 0;
}
```

Output: False
Reason: sizeof returns size_t which is unsigned int. When we compare int and unsigned int is converted to unsigned and comparison takes place between two unsigned values. Thus any unsigned positive int value will compare less with any negative signed value, if its MSB is 0.

```c
#include <stdio.h>

int main(int argc, char const *argv[])
{
    int x = 5;
    if (5 == x)
    {
        printf("True\n");
    }
    else{
        printf("False\n");
    }
    return 0;
}
```

Output: True
Reason: x == 5 and 5 == x both work.

```c
#include <stdio.h>

int main()
{
    float x = 1.1;
    while( x == 1.1 )
    {
        printf("%f\n",x);
        x=x-0.1;
    }
    return 0;
}
```

Output: No Output
Reason: 1.1 is a double value. To have output replace 1.1 with $1.1f$ in line 6. 1.1 is default treated as double.

```c
#include <stdio.h>

int staticFn() {
  static int x = 0;
  x++;
  return x;
}

int autoFn() {
  auto int x = 0;
  x++;
  return x;
}

int registerFn() {
  register int x = 0;
  x++;
  return x;
}

int main(void) {
  int j;
  for (j = 0; j < 5; j++) {
    printf("Value of autoFn(): %d\n", autoFn());
  }
  for (j = 0; j < 5; j++) {
    printf("Value of registerFn(): %d\n", registerFn());
  }
  for (j = 0; j < 5; j++) {
    printf("Value of staticFn(): %d\n", staticFn());
  }
  return 0;
}
```

Output: Value of auto(): 1 Value of auto(): 1 Value of auto(): 1 Value of auto(): 1 Value of auto(): 1 Value of register(): 1 Value of register(): 1 Value of register(): 1 Value of register(): 1 Value of register(): 1 Value of static(): 1 Value of static(): 2 Value of static(): 3 Value of static(): 4 Value of static(): 5

Reason: sizeof returns size_t which is unsigned int. When we compare int and unsigned int is converted to unsigned and comparison takes place between two unsigned values. Thus any unsigned positive int value will compare less with any negative signed value, if its MSB is 0.

Bibliography

[1] Interpolation search. http://en.wikipedia.org/wiki/Interpolation_search.

[2] Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Addison Wesley Publishing Company, USA., ISBN*, 10:020102988X, 1979.

[3] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013.

[4] Fundamentals of algorithms. http://webmuseum.mi.fh-offenburg.de/index.php?view=exh&src=73.

[5] Selective repeat go back n. http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/.

[6] Fundamentals of algorithms. http://www.geeksforgeeks.org/fundamentals-of-algorithms/.

Index