Department of Computer Science and Engineering

M.Tech in Computer Science and Engineering

2022-23

1st Semester

LABORATORY MANUAL

Advanced Data Structures and Algorithms

[22MCE12TL]

LAB IN-CHARGE: **Dr. Rajashree Shettar**

RV College of Engineering®, Bengaluru - 59
(An Autonomous Institution Affiliated to Visvesvaraya Technological University, Belagavi)

Department of Computer Science and Engineering

M. Tech. in Computer Science and Engineering

Vision: To achieve leadership in the field of Computer Science and Engineering by strengthening fundamentals and facilitating interdisciplinary sustainable research to meet the ever growing needs of the society.

Mission:

- To evolve continually as a center of excellence in quality education in computers and allied fields.
- To develop state-of-the-art infrastructure and create environment capable for interdisciplinary research and skill enhancement.
- To collaborate with industries and institutions at national and international levels to enhance research in emerging areas.
- To develop professionals having social concern to become leaders in top-notch industries and/or become entrepreneurs with good ethics.

Program Specific Criteria for M.Tech in Computer Science and Engineering

Professional Bodies: IEEE-CS, ACM

The M.Tech program in Computer Science and Engineering prepares the students for career in computer related courses that deal with design concepts with implementation. The program enables the students to acquire breadth and depth wise knowledge in computer science domain. The curriculum emphasizes courses on Mathematics and Statistics, Humanities, Ethics and Professional Practice, Computer Architecture, Analysis of Algorithms, Operating Systems, Computer Networks and Information Security, Computer Security along with elective courses. The program enables students in problems solving, critical thinking and communication skills with focus on team work.

Program Educational Objectives (PEO)

M. Tech. in Computer Science and Engineering graduates will be able to:

PEO 1. Exhibit analytical and computational skills to solve problems of the real world in conventional and advanced areas of Computer Science and Engineering.

PEO 2. Learn, compete and adapt to constantly evolving technology to meet the challenging needs of the industries.

PEO 3. Conceptualize, innovate and collaborate for facilitating interdisciplinary research with focus on professional ethics and team work.

PEO 4. Apply skills acquired in Computer Science and Engineering domain to design solutions using sustainable and inclusive technology for career advancement and life-long learning.

Program Outcomes (PO)

The graduates of M.Tech in Computer Science and Engineering program will be able to accomplish/attain:

PO1 An ability to independently carry out research/investigation and development work to solve practical problems.

PO2 An ability to write and present a substantial technical report/document.

PO3 Students should be able to demonstrate a degree of mastery over the area as per the specialization of the program. The mastery should be at a level higher than the requirements in the appropriate bachelor program.

PO4 Acquire knowledge to evaluate, analyze complex problems by applying principles of Mathematics and Computer Science & Engineering with a global perspective.

PO5 Explore, select, learn and model applications through use of state-of-art tools.

PO6 Recognize opportunities and contribute synergistically towards solving engineering problems effectively, individually and in teams, to accomplish a common goal and exhibit professional ethics, competence and to engage in lifelong learning.

Rubrics for Advanced Data Structures and Algorithms [22MCE12TL]

Each program is evaluated for 10 marks

Lab Write-up and Execution rubrics (Max: 6 marks)					
Sl. No.	**Criteria**	**Measuring methods**	**Excellent**	**Good**	**Poor**
1	**Understanding of problem statement (2 Marks) CO1**	Observations	Student exhibits thorough understanding of requirements and applies suitable algorithm for the problem **(2 M)**	Student has sufficient understanding of requirements and applies suitable algorithm for the problem. **(<2 M and >=1 M)**	Student does not have a clear understanding of requirements and is unable to apply suitable algorithm for the problem. **(0 M)**
2	**Execution (2 Marks) CO4**	Observations	Student demonstrates the execution of the program with optimized code and shows performance efficiency. Appropriate validations with all test cases are handled. **(2 M)**	Student demonstrates the execution of the program without optimization of the code and shows performance efficiency with only few test cases. **(<2 M and >=1 M)**	Student has not executed the program. **(0 M)**
3	**Results (1 Mark) CO3**	Observations	Results are appropriate to different inputs with proper validation and output is well interpreted. **(1 M)**	Results works for different inputs and covers validation and output is not so well interpreted. **(1.5 M)**	Results are inappropriate for few inputs and no proper validation. Output is not well interpreted. **(0.5 M)**

4	**Documentation (1 Mark) CO2**	Observations	Documentation with appropriate comments and output is covered in data sheets and manual. **(1 M)**	Documentation with only few comments and only few output cases is covered in data sheets and manual. **(1.5 M)**	Documentation with no comments and no output cases is covered in data sheets and manual. **(0.5 M)**
Viva Voce rubrics (Max: 4 marks)					
1	**Conceptual Understanding (1 Mark) CO1**	Viva Voce	Explains thoroughly the algorithm and the data structure used along with related concepts. **(1 M)**	Adequately explains the algorithms and data structures with related concepts **(0.5 M)**	Unable to explain the algorithm and data structure. **(0 M)**
2	**Use of appropriate Design Techniques (1 Mark) CO2**	Viva Voce	Insightful explanation of appropriate design techniques for the given problem to derive solution. **(1 M)**	Sufficiently explains the use of appropriate design techniques for the given problem to derive solution. **(0.5 M)**	Unable to explain the design techniques for the given problem. **(0 M)**
3	**Communication of Concepts (1 Mark) CO3**	Viva Voce	Communicates the concept used in problem solving well. **(1 M)**	Sufficiently communicates the concepts used in problem solving. **(0.5 M)**	Unable to communicate the concepts used in problem. **(0 M)**
4	**Analysis of Time and Space Complexity (1 Mark) CO4**	Viva Voce	Can analyze the Time and Space Complexity for the given problem **(1 M)**	Partially can analyze the Time and Space Complexity for the given problem **(0.5 M)**	Unable to analyze the Time and Space Complexity for the given problem **(0 M)**

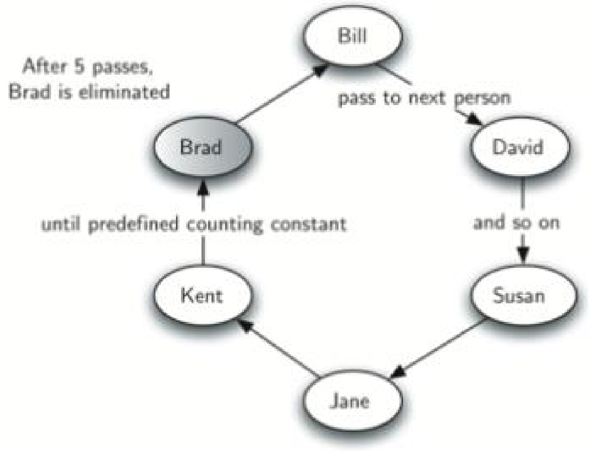
Table of Contents

Note: The following Experiments can be executed on Java/C#/Python/any equivalent language

Sl. No	**Experiments**	**Pg. No.**
1	If the costs of collecting the data for a genome were as expensive today as it was in 2001, there would be no hope of doing genome sequence comparison to find genome variation. But, since then, costs have dropped dramatically as researchers have developed new technologies for collecting genome sequence data. Several experimental techniques are now available, and recently automated genome sequencing became the most popular and least expensive experimental method, using what are called high-throughput sequencing technologies. Sequencing determines the order of the nucleotides on each chromosome so that when looked at the representation of chromosome sequence, it appears as a long string of the letters A, C, G, and T. Apply an appropriate algorithmic technique to sort the genome sequence for the data given in the below table and determine its performance.	**8**

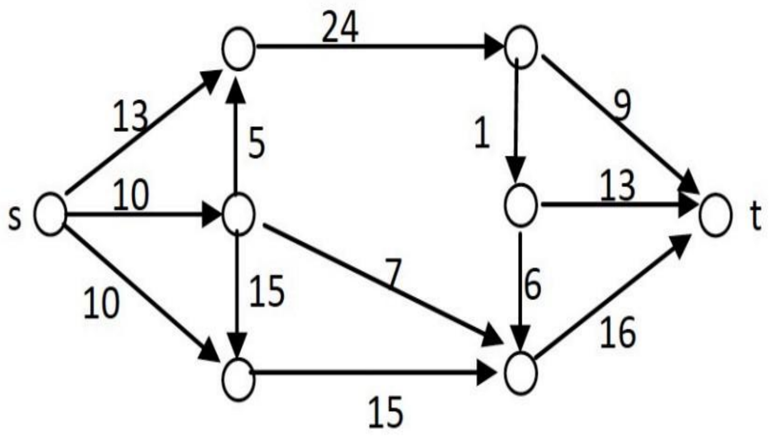
A	C	G
T	C	A
G	A	C
C	T	A
G	T	T
A	T	G
A	A	G
T	G	T
T	C	C
A	A	A
C	T	C
T	G	A
G	C	G
A	C	T

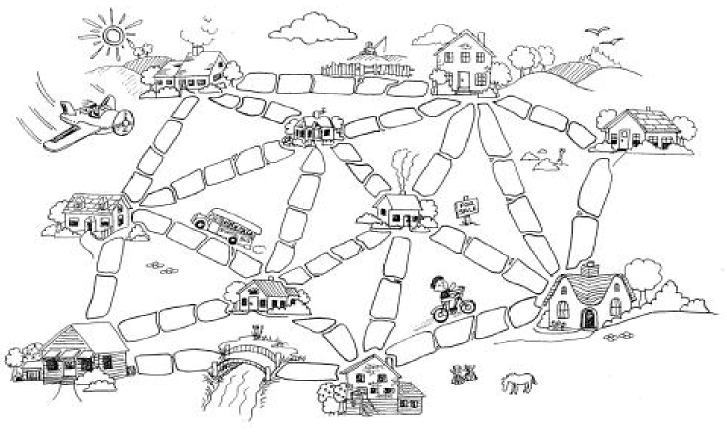
2	Using suitable Abstract Data Types (ADTs) perform the following tasks: (i) Checking if the parentheses are well formed i.e. [( ) ( )] is well-formed, but [(( ])) is not. (ii) Reverse the contents of a List using List ADT (iii) Use dictionary ADT to count the number of occurrences of each word in an online book.	**9**

3	Consider the children's game of Hot Potato. In this game (see Figure 1) children line up in a circle and pass an item from neighbor to neighbor as fast as they can. At a certain point in the game, the action is stopped and the child who has the item (the potato) is removed from the circle. Play continues until only one child is left. Implement a general simulation of Hot Potato using a suitable Abstract Data Type (ADT). Program should input a list of names and a constant, call it "num" to be used for counting. It will return the name of the last person remaining after repetitive counting by num. At that point print a suitable message as output. Assume that the child holding the potato will be at the front of the queue. Upon passing the potato, the simulation will simply dequeue and then immediately enqueue that child, putting her at the end of the line. She will then wait until all the others have been at the front before it will be her turn again. After num dequeue/enqueue operations, the child at the front will be removed permanently and another cycle will begin. This process will continue until only one name remains (the size of the queue is 1). Eg: A call to the hot potato function using 7 as the counting constant returns Susan Figure 1: A Six Person Game of Hot Potato  Bill After 5 passes, Brad is eliminated pass to next person Brad David until predefined counting constant and so on Kent Susan Jane	**11**
4	Suppose you have a hash table of size 19, the keys are words, and the hash map is defined as follows: Each letter is assigned a number according to its position in the alphabet and the primary hash function is "x modulo 19", where x is the number corresponding to the first letter of the word. Why is this hash function not ideal? Suppose instead you have a hash table of size 13 and the primary hash function is "x modulo 13", where x is the sum of the numbers corresponding to all the letters in the key word. Insert the following list of words into an initially empty hash table using linear probing: [computer, science, in, birmingham, dates, back, to, the, sixties]	**13**

	(a) What is the load factor of the resulting table, and how many collisions occurred? (b) What is the effort (i.e. number of comparisons) involved in checking whether each of the following words are in the hash table: teaching, research, admin? (c) Show what the resulting hash table would look like if direct chaining had been used rather than linear probing (d) Now what is the effort (i.e. number of comparisons, not including the processing of NULL pointers) involved in checking whether each of the following words are in the hash table: teaching, research, admin?	

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

5	Implement the following algorithms. Compare the time complexities of all the below given algorithms. **1. Simple Quick Sort** – Simple Quicksort with A[last] as the pivot. **2. Random Pivot** – Random Pivot Quicksort	**14**
6	Consider the electricity network as the graph shown below with utility poles as the different nodes in the graph. Calculate the maximum electricity that could be sent over the network without affecting the power grid. Use appropriate algorithm to solve the problem. 	**16**

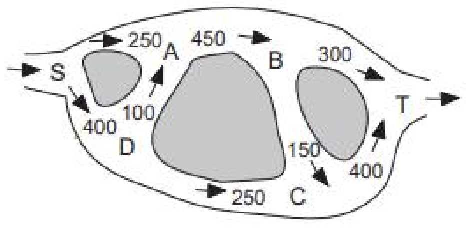
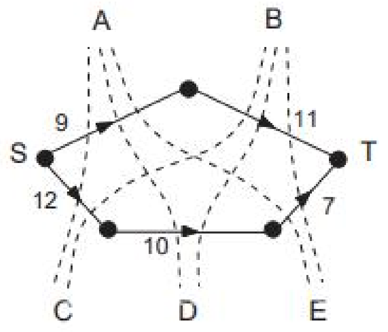
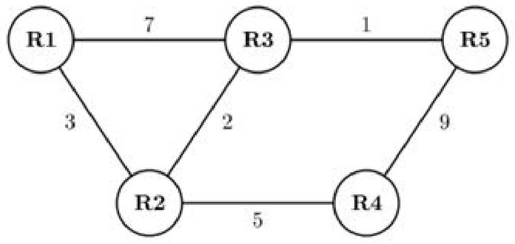
7	Once upon a time there was a city that had no roads. Getting around the city was particularly difficult after rainstorms because the ground became very muddy cars got stuck in the mud and people got their boots dirty. The mayor of the city decided that some of the streets must be paved, but didn't want to spend more money than necessary because the city also wanted to build a swimming pool. The mayor therefore specified two conditions: 1. Enough streets must be paved so that it is possible for everyone to travel from their house to anyone else's house only along paved roads 2. The paving should cost as little as possible. Here is the layout of the city. The number of paving stones between each house represents the cost of paving that route. Find the best route that connects all the houses, but uses as few counters (paving stones) as possible. Solve the above problem using suitable design paradigm and algorithm 	**18**
8	Consider the following scenario: The following 4 points - A, B, C, D denote 4 corners of a city. You are standing at A and want to reach D as soon as possible. However, there are only two ways in which you can do this. You can either go via B or you can go via C. Reaching B from A takes 1 min and reaching D from B takes 10 mins. Reaching C from A takes 5 min and reaching D from C takes 10 mins. If we follow the greedy technique, we will choose to go via B. If we go to C, it would take 5 mins. However, going to B only takes 1 min. Therefore, while standing at A, going via B seems to be the better solution. Apply appropriate Design Paradigm to find the final solution to reach D in minimum amount of time. Let's consider the same scenario again but with slight changes. This time, let the time taken to reach D from B be 20 minutes. If we apply the same Design Paradigm, we will have to go via B because when we are at A going to B seems like a better choice. But notice that if we go via B, the total time taken to reach D will be more compared to, going via C. Justify your answer with suitable explanation and suggest an alternate solution to this. Highlight the important takeaway from this problem.	**19**

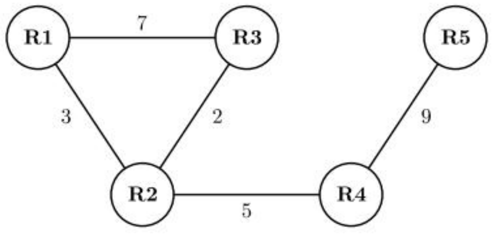
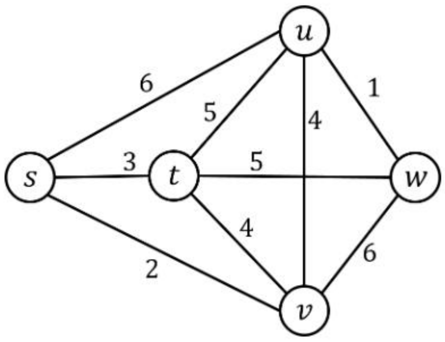
9	Let's assume that you are tasked with matching two lists of customer names. The customer names come from two different sources: list A is neat, while list B is polluted. The customer names should match 1:1 but the ones in list B are spelled out wrong. In addition, list B contains blank cells, a punctuation mark, and duplicate names (see image below). One option to solve this task would be to perform the matches manually (Accenture:Accent llxure, Adobe:Addobbe, etc.). If the lists are too long or the task needs to be repeated, one should opt for a more efficient solution. Solve this problem by applying appropriate design paradigm and algorithm.	**20**
10	A programming contest generally involves the host presenting a set of logical and mathematical problems to the contestants. The contestants are required to write computer programs that are capable of solving these problems. An online judge system is used to automate the judging procedure of the programs that are submitted by the users. Online judges are systems designed for the reliable evaluation of the source codes submitted by the users. Traditional online judging platforms are not ideally suitable for programming labs, as they do not support partial scoring and efficient detection of plagiarized codes. Apply a suitable design paradigm to solve the automatic scoring of codes by detecting plagiarized contents efficiently.	**21**

Sl.No.	**LIST-A**	**LIST-B**
1	Accenture	Accent llxure
2	Adobe	Addobbe
3	Akamai	.
4	Alexandria Real Estates	Alexandira Reel Estates
5	Biotech	Biotec
7	Boeing	Boing
8	Catamarine	Catamaine
9	Cadmaxx Solutions	Cadmax Soltion
10	Cisco Systems	Cisc Systm

OPEN-ENDED LAB PROGRAMS

1	You are given the integer array and you have to count the frequency of each element in the array. This problem can be solved by using the brute force approach by using the loop and increase the count of the element as its frequency increases. But using this approach the time complexity will be $O(N^2)$ and space complexity will be $O(1)$ for an array of size N which is not optimal. Think of an appropriate data structure to solve the above problem.
2	A library has its books organized primarily according to 20 categories represented by the two digit codes 01, 02, 03, ... 20, and secondarily according to the first two letters of the first author's surname Aa, Ab, ..., Az, Ba, ..., Zz. Use Radix Sort to sort the set of books with keys: [09 Ce, 09 Fa, 16 Mo, 16 Fa, 07 Ce, 13 Fa, 09 Mo, 07 Ba, 13 Ca]. Show the state of the book list and what is being done at each stage

3	The diagram represents a road network. All vehicles enter at S and leave at T. The numbers represent the maximum flow rate in vehicles per hour in the direction from S to T. What is the maximum number of vehicles which can enter and leave the network every hour? Which single section of road could be improved to increase the traffic flow in network?  S 250 A 450 B 300 T 400 100 D 150 400 250 C
4	The network below shows maximum capacities of each edge. Draw up a table showing the values of all the cuts from A, B to C,D,E. Which is the minimum cut? Draw the network with flows which give this maximum total flow. 
5	The figure below shows a network of five routers. Each link has a cost. Use appropriate algorithm to solve the above problem. Assume the link between R3 and R5 fails. How many iterations does it take for the algorithm to converge? 

	
6	You have a business with several offices; you want to lease phone lines to connect them up with each other, and the phone company charges different amounts of money to connect different pairs of cities. You are assigned the task of designing a telephone network. You want a set of lines that connects all your offices with a minimum total cost. Design an efficient network to remove some edges in the telephone and save money, say, for example, the telephone network looks like the one given below. Calculate the minimum total cost of the telephone network.  u s t w v 6 5 4 1 3 5 4 2 6

Advanced Algorithms Laboratory – 22MCE12TL

General Guidelines

1. The programs can be executed on Java/C#/Python/any other equivalent language
2. Exception handling to be applied to the programs
3. Tabulate the results for various input size to analyze the time complexity of the algorithm used to solve the problem.

Program No. 1

DNA Sequencing and Mapping

If the costs of collecting the data for a genome were as expensive today as it was in 2001, there would be no hope of doing genome sequence comparison to find genome variation. But, since then, costs have dropped dramatically as researchers have developed new technologies for collecting genome sequence data. Several experimental techniques are now available, and recently automated genome sequencing became the most popular and least expensive experimental method, using what are called high-throughput sequencing technologies.
Sequencing determines the order of the nucleotides on each chromosome so that when looked at the representation of chromosome sequence, it appears as a long string of the letters A, C, G, and T.
Apply an appropriate algorithmic technique to sort the genome sequence for the data given in the below table and determine its performance.

A	C	G
T	C	A
G	A	C
C	T	A
G	T	T
A	T	G
A	A	G
T	G	T
T	C	C
A	A	A
C	T	C
T	G	A
G	C	G
A	C	T

Solution:

Radix sort is a small method that many people intuitively use when alphabetizing a large list of names. Specifically, the list of names is first sorted according to the first letter of each name, that is, the names are arranged in 26 classes.

Intuitively, one might want to sort numbers on their most significant digit. However, Radix sort works counter-intuitively by sorting on the least significant digits first. On the first pass, all the numbers are sorted on the least significant digit and combined in an array. Then on the second pass, the entire numbers are sorted again on the second least significant digits and combined in an array and so on.

```
Algorithm: Radix-Sort (list, n)
shift = 1
for loop = 1 to keysize do
for entry = 1 to n do
bucketnumber = (list[entry].key / shift) mod 10
append (bucket[bucketnumber], list[entry])
list = combinebuckets()
shift = shift * 10
```

Analysis

Each key is looked at once for each digit (or letter if the keys are alphabetic) of the longest key. Hence, if the longest key has **m** digits and there are **n** keys, radix sort has order **O(mn)**.

However, the size of the keys will be relatively small when compared to the number of keys and this algorithm is of linear complexity **O(n)**.

Program No. 2

Using suitable Abstract Data Types (ADTs) perform the following tasks:
(i) Checking if the parentheses are well formed i.e. [( ) ( )] is well-formed, but [(( ])) is not.
(ii) Reverse the contents of a List using List ADT
(iii) Use dictionary ADT to count the number of occurrences of each word in an online book.

(i) Steps to find whether a given expression is balanced or unbalanced

- Input the expression and put it in a character stack.
- Scan the characters from the expression one by one.
- If the scanned character is a starting bracket (‘ (‘ or ‘ { ‘ or ‘ [‘), then push it to the stack.

- If the scanned character is a closing bracket (‘) ’ or ‘ } ’ or ‘] ’), then pop from the stack and if the popped character is the equivalent starting bracket, then proceed. Else, the expression is unbalanced.
- After scanning all the characters from the expression, if there is any parenthesis found in the stack or if the stack is not empty, then the expression is unbalanced.
- Now, let us see a program to check balanced parentheses in the given expression.

  Input: exp = “[()]{}{[()()]()}”
  Output: Balanced
  Explanation: *all the brackets are well-formed*

  Input: exp = “[(])”
  Output: Not Balanced
  Explanation: ***1*** *and* ***4*** *brackets are not balanced because there is a closing ‘]’ before the closing ‘(‘*

(ii) Reverse the contents of a List using List ADT

List ADT mean the abstract List object in java, then look into using Collections.reverse()
List<String> myList = new LinkedList<String>();
Collections.reverse(myList);

Input: Head of following linked list
1->2->3->4->NULL
Output: Linked list should be changed to,
4->3->2->1->NULL

Input: Head of following linked list
1->2->3->4->5->NULL
Output: Linked list should be changed to,
5->4->3->2->1->NULL

(iii) Use dictionary ADT to count the number of occurrences of each word in an online book.

To count the occurrences of words in a dictionary:
- ♣ Each dictionary entry is keyed by the word
- ♣ The related value is the count
- ♣ When entering words into dictionary
- ♣ Check if word is already there
- ♣ If no, enter it with a value of 1
- ♣ If yes, increment its value

Program No. 3

Consider the children's game of Hot Potato. In this game (see Figure 1) children line up in a circle and pass an item from neighbour to neighbour as fast as they can. At a certain point in the game, the action is stopped and the child who has the item (the potato) is removed from the circle. Play continues until only one child is left. Implement a general simulation of Hot Potato using a suitable Abstract Data Type (ADT). Program should input a list of names and a constant, call it "num" to be used for counting. It will return the name of the last person remaining after repetitive counting by num. At that point print a suitable message as output.
Assume that the child holding the potato will be at the front of the queue. Upon passing the potato, the simulation will simply dequeue and then immediately enqueue that child, putting her at the end of the line. She will then wait until all the others have been at the front before it will be her turn again. After num dequeue/enqueue operations, the child at the front will be removed permanently and another cycle will begin. This process will continue until only one name remains (the size of the queue is 1).
Eg: A call to the hot_potato function using 7 as the counting constant returns Susan

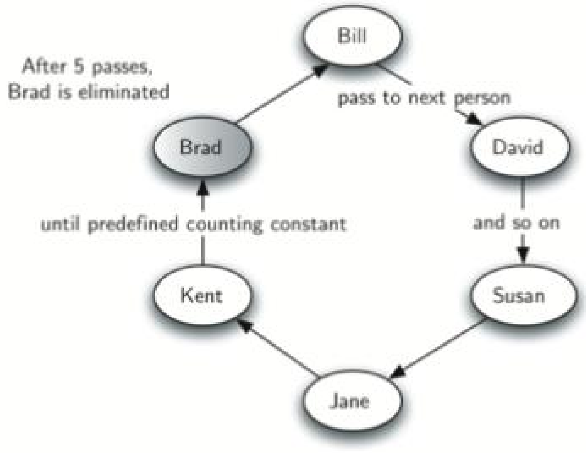
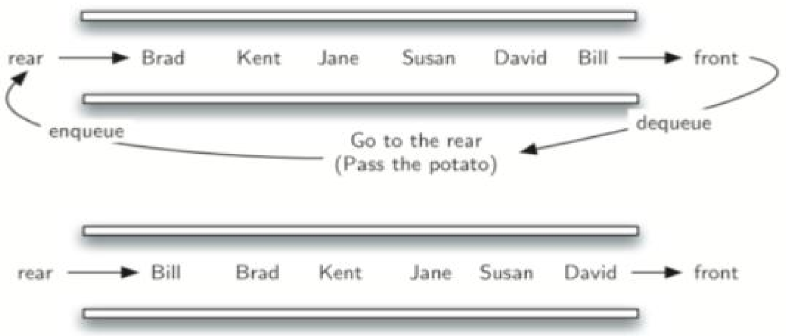

Bill
After 5 passes, Brad is eliminated
pass to next person
Brad
David
until predefined counting constant
and so on
Kent
Susan
Jane

Figure 1: A Six Person Game of Hot Potato

This game is a modern-day equivalent of the famous Josephus problem. Based on a legend about the famous first-century historian Flavius Josephus, the story is told that in the Jewish revolt against Rome, Josephus and 39 of his comrades held out against the Romans in a cave. With defeat imminent, they decided that they would rather die than be slaves to the Romans. They arranged themselves in a circle. One man was designated as number one, and proceeding clockwise they killed every seventh man. Josephus, according to the legend, was among other things an accomplished mathematician. He instantly figured out where he ought to sit in order to be the last to go. When the time came, instead of killing himself, he joined the Roman side. You can find many different versions of this story. Some count every third man and some allow the last man to escape on a horse. In any case, the idea is the same.

The program will input a list of names and a constant, call it "num," to be used for counting. It will return the name of the last person remaining after repetitive counting by num. To simulate the circle, we will use a queue. Assume that the child holding the potato will be at the front of the queue. Upon passing the potato, the simulation will simply dequeue and then immediately enqueue that child, putting her at the end of the line. She will then wait until all the others have been at the front before it will be her turn again. After num dequeue/enqueue operations, the child at the front will be removed permanently and another cycle will begin. This process will continue until only one name remains (the size of the queue is 1).

```
from pythonds.basic import Queue
def hotPotato(namelist, num):
   simqueue = Queue()
   for name in namelist:
      simqueue.enqueue(name)
   while simqueue.size() > 1:
      for i in range(num):
         simqueue.enqueue(simqueue.dequeue())
      simqueue.dequeue()
   return simqueue.dequeue()
print(hotPotato(["Bill","David","Susan","Jane","Kent","Brad"],7))
```



A Queue Implementation of Hot Potato

Output: A call to the hotPotato function using 7 as the counting constant returns Susan.

Note that in this example the value of the counting constant is greater than the number of names in the list. This is not a problem since the queue acts like a circle and counting continues back at the beginning until the value is reached. Also, notice that the list is loaded into the queue such that the first name on the list will be at the front of the queue. Bill in this case is the first item in the list and therefore moves to the front of the queue.

Program No. 4

Suppose you have a hash table of size 19, the keys are words, and the hash map is defined as follows:
Each letter is assigned a number according to its position in the alphabet i.e.

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

and the primary hash function is "x modulo 19", where x is the number corresponding to the first letter of the word. Why is this hash function not ideal?

Taking just one letter as the basis of the hash function means that words with the same first letter will be mapped to the same position, and since the letters in words are far from equally distributed, that will result in many collisions and a clustered table. This will be made worse because several consecutive positions correspond to two letters, and the rest only to one.

Suppose instead you have a hash table of size 13 and the primary hash function is "x modulo 13", where x is the sum of the numbers corresponding to all the letters in the key word.

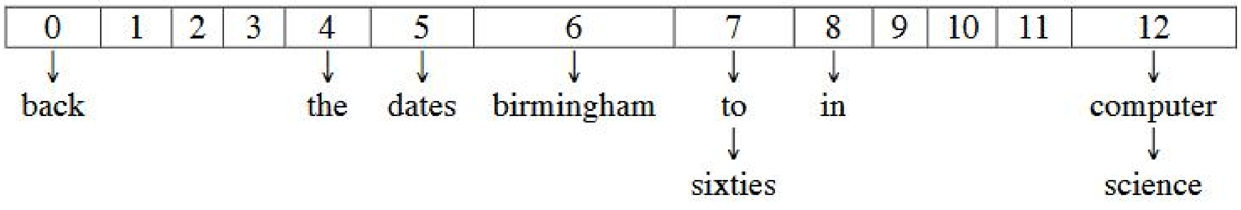
Insert the following list of words into an initially empty hash table using linear probing:
[computer, science, in, Birmingham, dates, back, to, the, sixties]

0	1	2	3	4	5	6	7	8	9	10	11	12
back			sixties	the	dates	birmingham	to	in			science	computer

(a) What is the load factor of the resulting table, and how many collisions occurred?
The load factor is 9/13 ~ 0.69 and 5 collisions occurred

(b) What is the effort (i.e. number of comparisons) involved in checking whether each of the following words are in the hash table: teaching, research, admin?
Checking for teaching takes 6 comparisons, research takes 3 comparisons, and admin takes 1 comparison.

(c) Show what the resulting hash table would look like if direct chaining had been used rather than linear probing



(d) Now what is the effort (i.e. number of comparisons, not including the processing of NULL pointers) involved in checking whether each of the following words are in the hash table: teaching, research, admin?

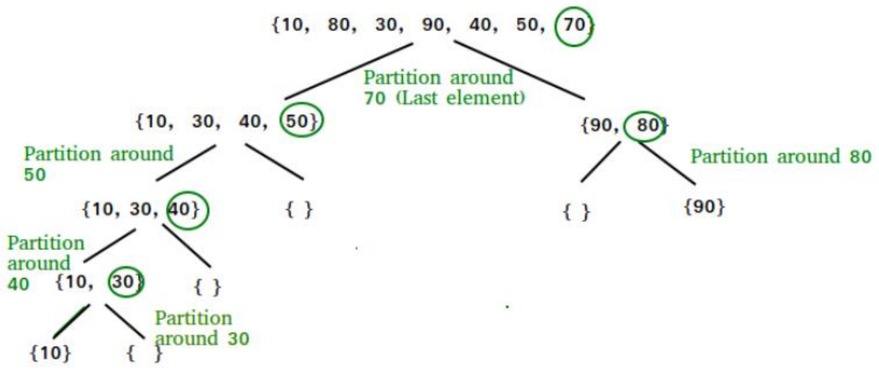
Checking for teaching takes 2 comparisons, research takes 1 comparison, and admin takes 0 comparisons

Program No. 5

Implement the following algorithms. Compare the time complexities of all the below given algorithms.

1. **Simple Quick Sort** – Simple Quicksort with A[last] as the pivot.
2. **Random Pivot** – Random Pivot Quicksort

The key process in **quickSort** is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.



Pseudo Code for recursive QuickSort function:

```
/* low  -> Starting index,  high  -> Ending index */

quickSort(arr[], low, high) {
  if (low < high) {
      /* pi is partitioning index, arr[pi] is now at right place */
      pi = partition(arr, low, high);
      quickSort(arr, low, pi – 1);  // Before pi
      quickSort(arr, pi + 1, high); // After pi
  }
}
```

Pseudo code for partition()

```
/* 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 */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];
    i = (low – 1)  // Index of smaller element and indicates the
    // right position of pivot found so far
    for (j = low; j <= high- 1; j++){
        // If current element is smaller than the pivot
        if (arr[j] < pivot){
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Analysis of QuickSort

Time taken by QuickSort, in general, can be written as follows.

$T(n) = T(k) + T(n-k-1) + \Theta(n)$

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements that are smaller than the pivot.
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case:
The worst case occurs when the partition process always picks the greatest or smallest element as the pivot. If we consider the above partition strategy where the last element is always picked as a pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for the worst case.
$T(n) = T(0) + T(n-1) + \Theta (n)$which is equivalent to $T(n) = T(n-1) + \Theta (n)$
The solution to the above recurrence is $\Theta (n^2)$.

Best Case:
The best case occurs when the partition process always picks the middle element as the pivot. The following is recurrence for the best case.
$T(n) = 2T(n/2) + \Theta (n)$

Quicksort has a couple of other differences from merge sort. Quicksort works in place. And its worst-case running time is as bad as selection sort's and insertion sort's: Θ(n2). But its average-case running time is as good as merge sort's: Θ(nlog2n)
So why think about quicksort when merge sort is at least as good? That's because the constant factor hidden in the big-Θ notation for quicksort is quite good. In practice, quicksort outperforms merge sort, and it significantly outperforms selection sort and insertion sort.

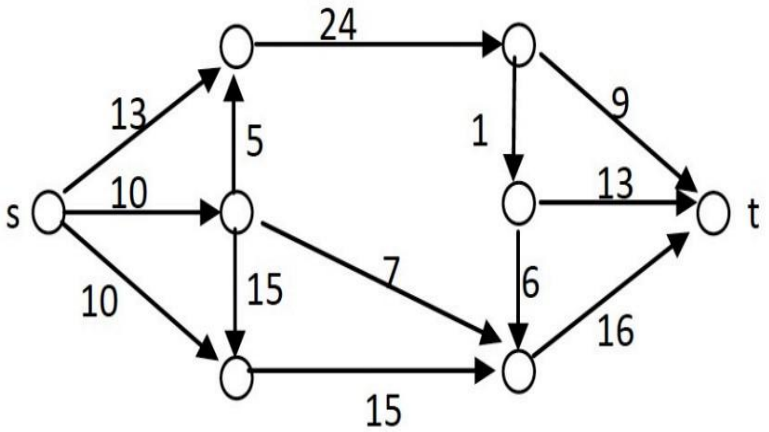
2. Algorithm for random pivoting

```
partition(arr[], lo, hi)
    pivot = arr[hi]
    i = lo     // place for swapping
    for j := lo to hi – 1 do
        if arr[j] <= pivot then
            swap arr[i] with arr[j]
            i = i + 1
    swap arr[i] with arr[hi]
    return i

partition_r(arr[], lo, hi)
    r = Random Number from lo to hi
    Swap arr[r] and arr[hi]
    return partition(arr, lo, hi)

quicksort(arr[], lo, hi)
    if lo < hi
        p = partition_r(arr, lo, hi)
        quicksort(arr, lo , p-1)
        quicksort(arr, p+1, hi)
```

Using random pivoting we improve the expected or average time complexity to O (N log N).
The Worst-Case complexity is still O (N^2).
Auxiliary Space: O(N) // due to recursive call stack

Program No. 6

Consider the electricity network as the graph shown below with utility poles as the different nodes in the graph. Calculate the maximum electricity that could be sent over the network without affecting the power grid. Use appropriate algorithm to solve the problem.



As the name shows, these problems can be used to estimate the maximum volume (depending on the problem) a graph can accommodate. For example, if we consider the electricity network as our graph and the utility poles as the different nodes in the graph. We can have assumptions on how much electricity could be sent over the network without affecting the power grid. Another example is a mobile network where each user acts as a node in the graph. As in the former example, we can figure out the maximum number of users who can stay online without network traffic. This formulation can answer the maximum of all and predict potential bottlenecks. Different algorithms used are *Ford-Fulkerson* and *Edmonds Karp & Dinic's algorithms.*

The Ford-Fulkerson algorithm assumes that the input will be a graph, GG, along with a source vertex, ss, and a sink vertex, tt. The graph is any representation of a weighted graph where vertices are connected by edges of specified weights. There must also be a source vertex and sink vertex to understand the beginning and end of the flow network.

Ford-Fulkerson Algorithm ((Graph GG, source ss, sink t):t):

```
initialize flow to 0
path = findAugmentingPath(G, s, t)
while path exists:
    augment flow along path                #This is purposefully ambiguous for now
    G_f = createResidualGraph()
    path = findAugmentingPath(G_f, s, t)
return flow
```

Version of the pseudo-code that explains the flow augmentation:

```
flow = 0
for each edge (u, v) in G:
    flow(u, v) = 0
while there is a path, p, from s -> t in residual network G_f:
    residual_capacity(p) = min(residual_capacity(u, v) : for (u, v) in p)
    flow = flow + residual_capacity(p)
    for each edge (u, v) in p:
        if (u, v) is a forward edge:
            flow(u, v) = flow(u, v) + residual_capacity(p)
        else:
            flow(u, v) = flow(u, v) - residual_capacity(p)
return flow
```

Residual graphs are an important middle step in calculating the maximum flow. As noted in the pseudo-code, they are calculated at every step so that augmenting paths can be found from the source to the sink. To understand how these are created and how they are used, we can use the graph from the intuition section.

However, one important attribute is important to understand before looking at residual graphs. *Residual capacity* is a term used in the above pseudo-code, and it plays an important role in residual graph creation. Residual capacity is defined as the new capacity after a given flow has been taken away. In other words, for a given edge (u,v)(u,v), the residual capacity, cfcf is defined as

cf(u,v)=c(u,v)−f(u,v).

However, there must also be a residual capacity for the *reverse* edge as well. The max-flow min-cut theorem states that flow must be preserved in a network. So, the following equality always holds:

f(u,v)=−f(v,u).f(u,v)=−f(v,u).

Time Complexity:

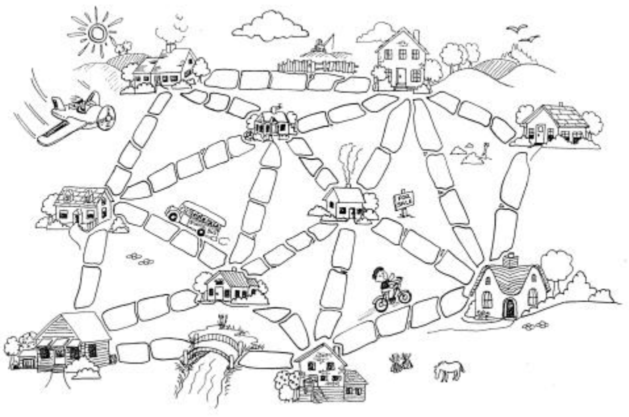
Ford Fulkerson algorithm has time complexity **O(|E| * f)**, where |E| is the number of edges and 'f' is the maximum flow of a network because it depends on the maximum flow of the network. Edmonds Karp algorithm has time complexity O(|V| * E^2), where E is the number of edges and V is the number of vertices.

Program 7

Once upon a time there was a city that had no roads. Getting around the city was particularly difficult after rainstorms because the ground became very muddy cars got stuck in the mud and people got their boots dirty. The mayor of the city decided that some of the streets must be paved, but didn't want to spend more money than necessary because the city also wanted to build a swimming pool.

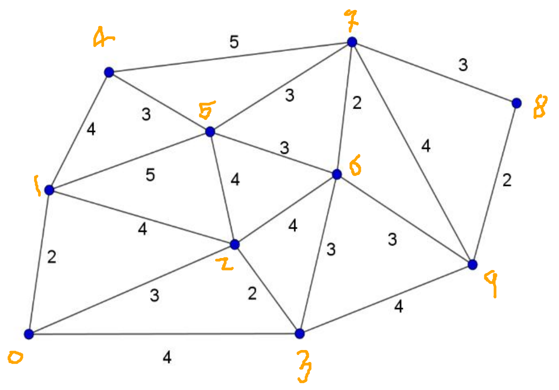
The mayor therefore specified two conditions:
1. Enough streets must be paved so that it is possible for everyone to travel from their house to anyone else's house only along paved roads
2. The paving should cost as little as possible. Here is the layout of the city. The number of paving stones between each house represents the cost of paving that route. Find the best route that connects all the houses, but uses as few counters (paving stones) as possible.

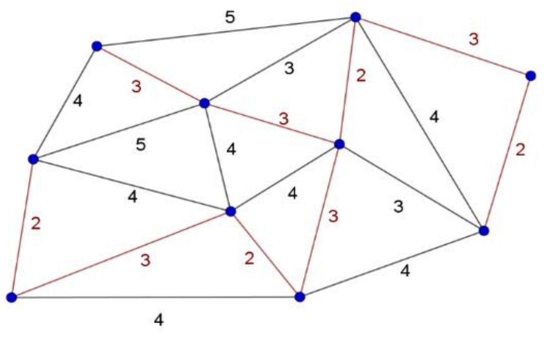
Solve the above problem using suitable design paradigm and algorithm

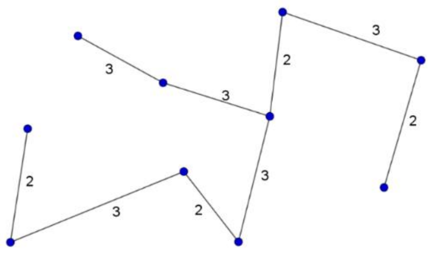


Sample solution: The minimum number of paving stones required is 23. This is obtained from the minimum spanning tree. Network diagram:

Minimum spanning tree:







Program 8

Consider the following scenario: The following 4 points - A, B, C, D denote 4 corners of a city. You are standing at A and want to reach D as soon as possible. However, there are only two ways in which you can do this. You can either go via B or you can go via C. Reaching B from A takes 1 min and reaching D from B takes 10 mins. Reaching C from A takes 5 min and reaching D from C takes 10 mins. If we follow the greedy technique, we will choose to go via B. If we go to C, it would take 5 mins. However, going to B only takes 1 min. Therefore, while standing at A, going via B seems to be the better solution.
Apply appropriate Design Paradigm to find the final solution to reach D in minimum

amount of time.
Let's consider the same scenario again but with slight changes. This time, let the time taken to reach D from B be 20 minutes.
If we apply the same Design Paradigm, we will have to go via B because when we are at A going to B seems like a better choice. But notice that if we go via B, the total time taken to reach D will be more compared to, going via C.

Justify your answer with suitable explanation and suggest an alternate solution to this. Highlight the important takeaway from this problem.

Solution:

Solve the problem using Greedy technique, Dijkstra’s algorithm. Prove that this does not work. Hence Greedy cannot be applied to solve all problems. Apply Dynamic Programming Bellman Ford algorithm

Dijkstra’s algorithm

```
void Graph::dijkstra(Vertex s){
Vertex v,w;
Initialize s.dist = 0 and set dist of all other
vertices to infinity
while (there exist unknown vertices, find the one b with the smallest distance)
b.known = true;
}
for each a adjacent to b
if (!a.known)
if (b.dist + Cost_ba < a.dist){
decrease(a.dist to= b.dist + Cost_ba);
a.path = b;   } } }
```

Program No. 9

Let’s assume that you are tasked with matching two lists of customer names. The customer names come from two different sources: list A is neat, while list B is polluted. The customer names should match 1:1 but the ones in list B are spelled out wrong. In addition, list B contains blank cells, a punctuation mark, and duplicate names (see image below). One option to solve this task would be to perform the matches manually (Accenture:Accent llxure, Adobe:Addobbe, etc.). If the lists are too long or the task needs to be repeated, one should opt for a more efficient solution.
Solve this problem by applying appropriate design paradigm and algorithm.

Sl.No.	LIST-A	LIST-B
1	Accenture	Accent llxure
2	Adobe	Addobbe
3	Akamai	.
4	Alexandria Real Estates	Alexandira Reel

		Estates
5	Biotech	Biotec
7	Boeing	Boing
8	Catamarine	Catamaine
9	Cadmaxx Solutions	Cadmax Soltion
10	Cisco Systems	Cisc Systm

Levenshtein distance is used to carry out approximate string matching. The Levenshtein distance is a robust and effective measurement of the distance between two sequences. The algorithm calculates the edit distance — the minimum number of edits (insertions, deletions, or substitutions) needed for a single-word sequence to match the target sequence.

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j)+1 \\ \operatorname{lev}_{a,b}(i,j-1)+1 \\ \operatorname{lev}_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

The **Levenshtein distance** is a similarity measure between words. Given two words, the distance measures the number of edits needed to transform one word into another. There are three techniques that can be used for editing:

1. Insertion
2. Deletion
3. Replacement (substitution)

Program No. 10

A programming contest generally involves the host presenting a set of logical and mathematical problems to the contestants. The contestants are required to write computer programs that are capable of solving these problems. An online judge system is used to automate the judging procedure of the programs that are submitted by the users. Online judges are systems designed for the reliable evaluation of the source codes submitted by the users. Traditional online judging platforms are not ideally suitable for programming labs, as they do not support partial scoring and efficient detection of plagiarized codes. Apply a suitable design paradigm to solve the automatic scoring of codes by detecting plagiarized contents efficiently.

Rabin Karp Algorithm:

The Rabin-Karp algorithm is a string searching algorithm that uses hashing to find any one of a set of pattern strings in a text.

The Rabin-Karp string searching algorithm calculates a hash value for the pattern, and for each M-character subsequence of text to be compared. If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence. If the hash values are equal, the algorithm will do a Brute Force comparison between the pattern and the M character sequence. In this way, there is only one comparison per text subsequence, and Brute Force is only needed when hash values match.

Rabin-Karp string searching algorithm compares a string's hash values, rather than the strings themselves. For efficiency, the hash value of the next position in the text is easily computed from the hash value of the current position.

The hash function may vary depending on many things, so it may consist of ASCII char to number converting, but it can be also anything else. The only thing we need is to convert a string (pattern) into some hash that is faster to compare. Let's say we have the string "hello world", and let's assume that its hash is hash ('hello world') = 12345. So if hash ('he') = 1 we can say that the pattern "he" is contained in the text "hello world". Thus on every step we take from the text a sub-string with the length of m, where m is the pattern length. Thus we hash this sub-string and we can directly compare it to the hashed pattern, as on the picture above.

Let characters in both arrays T and P be digits in radix-S notation. (S = (0, 1,..., 9). Let p be the value of the characters in P. Choose a prime number *q* such that fits within a computer word to speed computations. Compute (p mod q). The value of p mod q is what we will be using to find all matches of the pattern P in T. Compute (T[s+1,.., s+m] mod q) for s = 0 .. n-m. Test against P only those sequences in T having the same (mod q) value. (T[s+1, .., s+m] mod q) can be incrementally computed by subtracting the high-order digit, shifting, adding the low-order bit, all in modulo q arithmetic.

Consider an M-character sequence as an M-digit number in base b, where b is the number of letters in the alphabet. The text subsequence t[i .. i+M-1] is mapped to the number.

$x(i) = t[i].b^{M-1} + t[i+1].b^{M-2} + ... + t[i+M-1]$

Furthermore, given x(i) we can compute x(i+1) for the next subsequence t[i+1 .. i+M] in constant time, as follows:

$x(i+1) = t[i+1].b^{M-1} + t[i+2].b^{M-2} + ... + t[i+M]$x(i+1) = x(i).b //Shift left one digit

$- t[i].b^{M}$ // Subtract leftmost digit

$+ t[i+M]$ //Add new rightmost digit

In this way, we never explicitly compute a new value. Let's say that our alphabet consists of 10 letters. Our alphabet = a,b, c, d, e, f, g, h, i, j Let's say that ―a‖ corresponds to 1, ―b‖ corresponds to 2 and so on. The hash value for string ―cah‖ would be:-

3*100 + 1*10 + 8*1 = 318
If M is large, then the resulting value (~bM) will be enormous.

For this reason, we hash the value by taking it mod a prime number q. The mod function is particularly useful in this case due to several of its inherent properties:
- [(x mod q) + (y mod q)] mod q = (x+y) mod q
- (x mod q) mod q = x mod q

For these reasons:
h(i) = ((t[i].b^{M-1} mod q) + (t[i+1].b^{M-2} mod q) + ... + (t[i+M-1] mod q)) mod q
h(i+1) = (h(i).b mod q //Shift left one digit
-t[i]. b^{M} mod q //Subtract leftmost digit
+t [i+M] mod q) //Add new rightmost digit mod q

Rabin-Karp Matcher Algorithm:

Rabin-Karp-Matcher (T, P, d, q)

```
n ← length[ T ] m ← length[P ] h ← $d^{m-1}$ mod
q←0
t0 ← 0
for i ← 1 to m      // Preprocessing
        do    p ← ( dp + P[ i ] ) mod q
        t0 ← ( dt0 + T[ i ] ) mod q
for s ← 0 to n - m      // Matching
        do if p = ts
                then if P[ 1..m ] = T[ s+1 .. s+m ]
                        then print "Pattern occurs with shift" s
            if s < n - m
                then ts+1 ← ( d ( ts - T[ s + 1 ] h ) + T[ s + m + 1 ] ) mod q
```

Rabin-Karp Matcher Algorithm Example:

Given T = 2359023141526739921 and P = 314152, let q = 13,

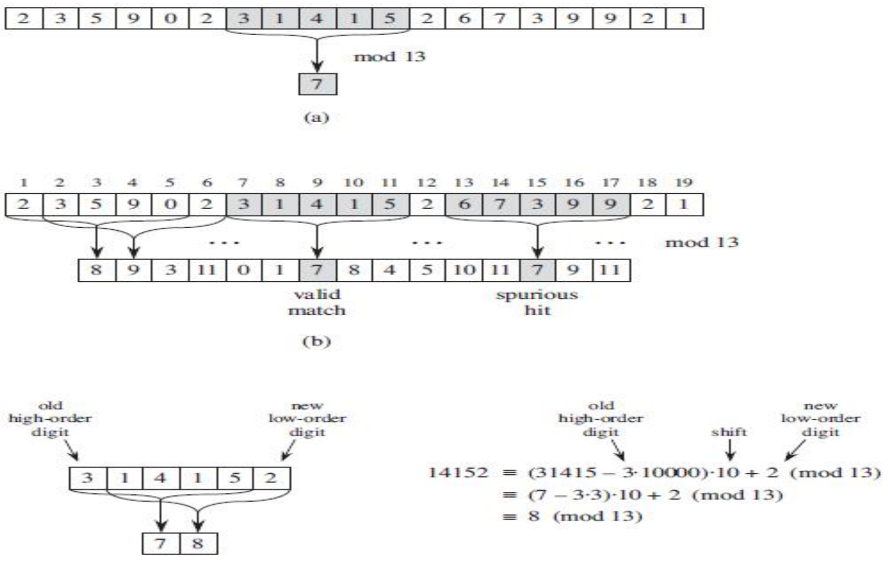
P mod q = 314152 mod 13 = 7

Each character is a decimal digit, and we compute values modulo 13. In the following figure:

(a) A text string. A window of length 5 is shown shaded. The numerical value of the shaded number is computed modulo 13, yielding the value 7

(b) The same text string values computed modulo 13 for each possible position of a length 5 window. Assuming the pattern p=31415, we look for windows whose value modulo 13 is 7, since 31415= 7(mod 13). Two such windows are found, shown shaded in the figure. The first, beginning at text position 7, is indeed an occurrence of the pattern, while the second, beginning at ext position 13, is a spurious hit.

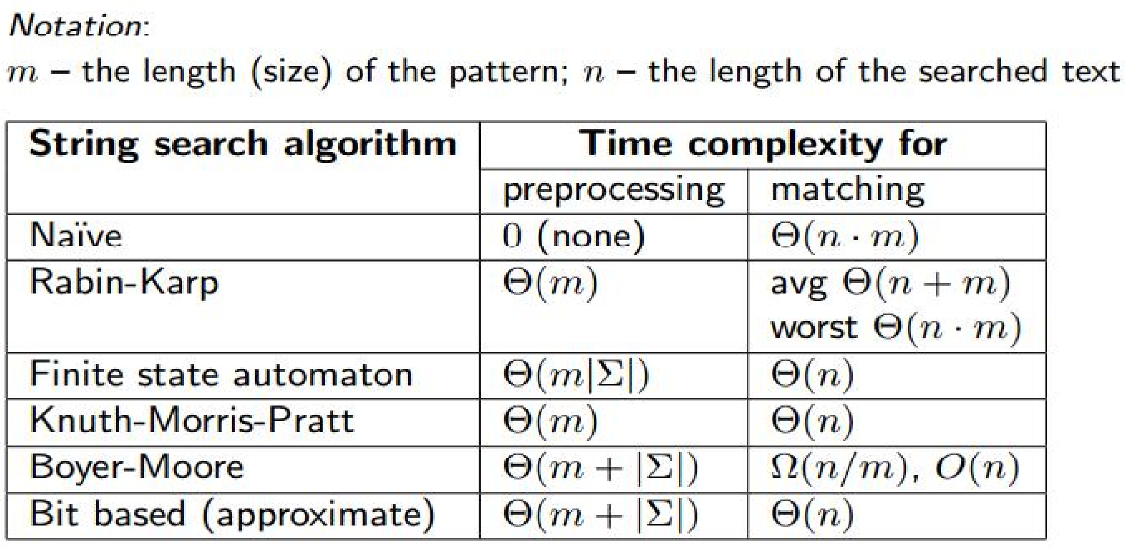
(c) Computing the value for a window in constant time, given the value for the previous window. The first window has value 31415. Dropping the high-order digit 3, shifting left (multiplying by 10), and then adding in the low-order digit 2 gives us the new value 14152. All computations are performed modulo 13, however, so the value for the first window is 7, and the value computed for the new window is 8.



Rabin-Karp Matcher Algorithm Running Time:
For text of length n and p patterns of combined length m, its average and best case running time is O(n+m) in space O(p), but its worst-case time is O(nm).

Rabin-Karp Matcher Algorithm Application:

- Detecting plagiarism
- Bioinformatics-Used in looking for similarities of two or more proteins; i.e. high sequence similarity usually implies significant structural or functional similarity.
- Spam Filters/Spam Detection System
- Intrusion Detection System



Notation:
m – the length (size) of the pattern; n – the length of the searched text

String search algorithm	**Time complexity for**	
	preprocessing	matching
Naïve	0 (none)	$\Theta(n \cdot m)$
Rabin-Karp	$\Theta(m)$	avg $\Theta(n+m)$ worst $\Theta(n \cdot m)$
Finite state automaton	$\Theta(m\|\Sigma\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$
Boyer-Moore	$\Theta(m + \|\Sigma\|)$	$\Omega(n/m)$, $O(n)$
Bit based (approximate)	$\Theta(m + \|\Sigma\|)$	$\Theta(n)$

Reference

- Thomas Cormen, et al. "Introduction to Algorithms", Third Edition, MIT Press, Cambridge.