A Project Report on

Analysis and Design of Algorithm

ADA Lab Mini Project (MCA – 552)

Submitted to the

S. S. Jain Subodh P. G. College

Affiliated to the

Rajasthan Technical University,
Kota

MASTER OF COMPUTER APPLICATIONS

Submitted To	**Submitted by**
Faculty Name : Dr. Sonal Bordia Jain **Associate Professor**	**Name :** Nimisha Vilayatarani **(18CPGXX264)**

CERTIFICATE

This is to certify that the mini project entitled, **Analysis and Design of Algorithm** submitted by Nimisha vilayatarani, Roll. No. 18CPGXX264 in partial fulfillment of the requirements for the award of MASTER OF COMPUTER APPICATIONS at the S.S. Jain Subodh P.G. (Autonomous) College, Jaipur is an authentic work carried out by him under my supervision and guidance.
To the best of my knowledge, the matter embodied in the project has not been submitted to any other University / Institute for the award of any Degree.

Date :

Dr. Sonal Bordia Jain
Associate Professor
S.S. Jain Subodh P.G College, Jaipur

Acknowledgment

I express my deep sense of gratitude to my respected and learned guide **Dr. Sonal Bordia Jain** for his valuable help and guidance. I am thankful for the encouragement he has given me in the completion of my project. I am also grateful to our respected Prof. K.B. Sharma, Director and Mr. Ashish Chandra Swami, Head of Department for allowing me to utilize all the necessary resources of the institution to complete my project. I am also thankful to all the other faculty and staff members of our department for their kind co-operation and help. Lastly, I would like to express my deep appreciation towards my classmates and indebtedness to my parents for providing me the moral support and encouragement.

Date :

Nimishavilayatrani
MCA (IV Sem.)
18CPGXX264

Table of contents :

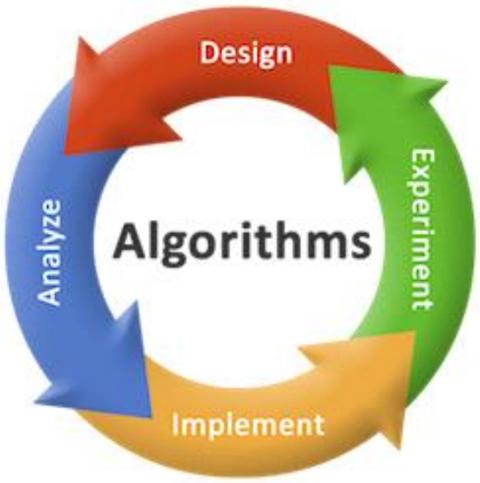
• Introduction

What is Algorithm?

A finite set of instruction that specifies a sequence of operation is to be carried out in order to solve a specific problem or class of problems is called an Algorithm.
An Algorithm is a sequence of steps to solve a problem. Design and Analysis of Algorithm is very important for designing algorithm to solve different types of problems in the branch of computer science and information technology.

Why study Algorithm?

As the speed of processor increases, performance is frequently said to be less central than other software quality characteristics (e.g. security, extensibility, reusability etc.). However, large problem sizes are commonplace in the area of computational science, which makes performance a very important factor. This is because longer computation time, to name a few mean slower results, less through research and higher cost of computation (if buying CPU Hours from an external party). The study of Algorithm, therefore, gives us a language to express performance as a function of problem size.



- **<u>Complexity of Algorithm</u>**

- **Time complexity**

In computer science, the time complexity is the computational complexity that describes the amount of time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform. Thus, the amount of time taken and the number of elementary operations performed by the algorithm are taken to differ by at most a constant factor.

Understanding Time Complexity with Simple Examples

Order of growth is how the time of execution depends on the length of the input. In the above example, we can clearly see that the time of execution is linearly depends on the length of the array. Order of growth will help us to compute the running time with ease. We will ignore the lower order terms, since the lower order terms are relatively insignificant for large input. We use different notation to describe limiting behavior of a function.

O-notation:

To denote asymptotic upper bound, we use O-notation. For a given function g(n), we denote by O(g(n)) (pronounced "big-oh of g of n") the set of functions: O(g(n))= { f(n) : there exist positive constants c and n0 such that 0≤f(n)≤c∗g(n) for all n≥n0 }

Ω-notation:

To denote asymptotic lower bound, we use Ω-notation. For a given function g(n), we denote by Ω(g(n)) (pronounced "big-omega of g of n") the set of functions: Ω(g(n))= { f(n) : there exist positive constants c and n0 such that 0≤c∗g(n)≤f(n) for all n≥n0 }

Θ-notation:

To denote asymptotic tight bound, we use Θ-notation. For a given function g(n), we denote by Θ(g(n)) (pronounced "big-theta of g of n") the set of functions: Θ(g(n))= { f(n) : there exist positive constants c1,c2 and n0 such that 0≤c1∗g(n)≤f(n)≤c2∗g(n) for all n>n0 }

Constant time

An algorithm is said to be constant time (also written as O(1) time) if the value of *T*(*n*) is bounded by a value that does not depend on the size of the input. For example, accessing any single element in an array takes constant time as only one operation has to be performed to locate it. In a similar manner, finding the minimal value in an array sorted in ascending order; it is the first element. However, finding the minimal value in an unordered array is not a constant time operation as scanning over each element in the array is needed in order to determine the minimal value. Hence it is a linear time operation, taking O(n) time. If the number of elements is known in advance and does not change, however, such an algorithm can still be said to run in constant time.

Despite the name "constant time", the running time does not have to be independent of the problem size, but an upper bound for the running time has to be bounded independently of the problem size. For example, the task "exchange the values of *a* and *b* if necessary, so that *a*≤*b*" is called constant time even though the time may depend on whether or not it is already true that *a* ≤ *b*. However, there is some constant *t* such that the time required is always *at most t*.

Sub-linear time

An algorithm is said to run in sub-linear time (often spelled sublinear time) if $T(n) = o(n)$. In particular this includes algorithms with the time complexities defined above.

Typical algorithms that are exact and yet run in sub-linear time use parallel processing (as the NC_1 matrix determinant calculation does), or alternatively have guaranteed assumptions on the input structure (as the logarithmic time binary search and many tree maintenance algorithms do). However, formal languages such as the set of all strings that have a 1-bit in the position indicated by the first log(n) bits of the string may depend on every bit of the input and yet be computable in sub-linear time.

The specific term *sublinear time algorithm* is usually reserved to algorithms that are unlike the above in that they are run over classical serial machine models and are not allowed prior assumptions on the input. They are however allowed to be randomized, and indeed must be randomized for all but the most trivial of tasks.

As such an algorithm must provide an answer without reading the entire input, its particulars heavily depend on the access allowed to the input. Usually for an input that is represented as a binary string $b_1,\ldots,b_k$ it is assumed that the algorithm can in time O(1) request and obtain the value of b_i for any i.

Sub-linear time algorithms are typically randomized, and provide only approximate solutions. In fact, the property of a binary string having only zeros (and no ones) can be easily proved not to be decidable by a (non-approximate) sub-linear time algorithm. Sub-linear time algorithms arise naturally in the investigation of property testing.

Linear time

An algorithm is said to take linear time, or $O(n)$ time, if its time complexity is $O(n)$. Informally, this means that the running time increases at most linearly with the size of the input. More precisely, this means that there is a constant c such that the running time is at most cn for every input of size n. For example, a procedure that adds up all elements of a list requires time proportional to the length of the list, if the adding time is constant, or, at least, bounded by a constant.

Linear time is the best possible time complexity in situations where the algorithm has to sequentially read its entire input. Therefore, much research has been invested into discovering algorithms exhibiting linear time or, at least, nearly linear time. This research includes both software and hardware methods. There are several hardware technologies which exploit parallelism to provide this. An example is content-addressable memory. This concept of linear time is used in string matching algorithms such as the Boyer–Moore algorithm and Ukkonen's algorithm.

Polynomial time

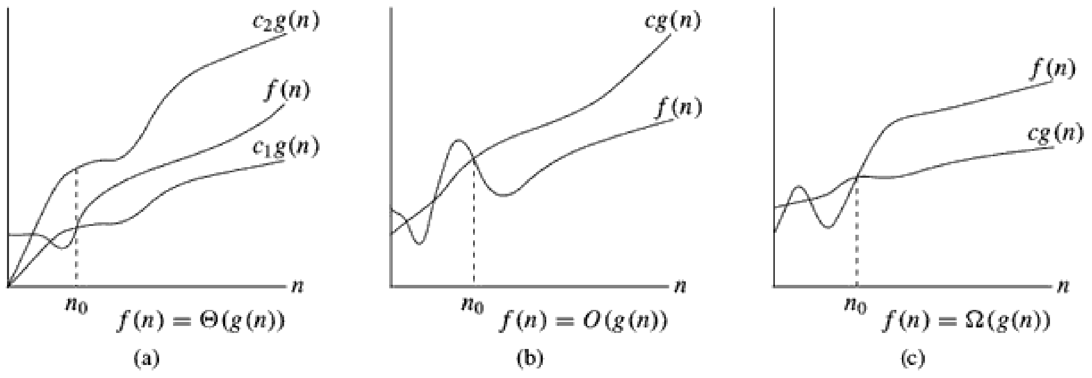
An algorithm is said to be of polynomial time if its running time is upper bounded by a polynomial expression in the size of the input for the algorithm, i.e., $T(n) = O(n^k)$ for some positive constant k. Problems for which a deterministic polynomial time algorithm exists belong to the complexity class P, which is central in the field of computational complexity theory. Cobham's thesis states that polynomial time is a synonym for "tractable", "feasible", "efficient", or "fast".

Logarithmic time

An algorithm is said to take logarithmic time when $T(n) = O(\log n)$. Since $\log_a n$ and $\log_b n$ are related by a constant multiplier, and such a multiplier is irrelevant to big-O classification, the standard usage for logarithmic-time algorithms is $O(\log n)$ regardless of the base of the logarithm appearing in the expression of T.

Algorithms taking logarithmic time are commonly found in operations on binary trees or when using binary search.

An O(log n) algorithm is considered highly efficient, as the ratio of the number of operations to the size of the input decreases and tends to zero when n increases. An algorithm that must access all elements of its input cannot take logarithmic time, as the time taken for reading an input of size n is of the order of n.



Time complexity notations

- **Space complexity**

In computer science, the space complexity of an algorithm or a computer program is the amount of memory space required to solve an instance of the computational problem as a function of the size of the input. It is the memory required by an algorithm to execute a program and produce output.

The term Space Complexity is misused for Auxiliary Space at many places. Following are the correct definitions of Auxiliary Space and Space Complexity.

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.

For example, if we want to compare standard sorting algorithms on the basis of space, then Auxiliary Space would be a better criteria than Space Complexity. Merge Sort uses O(n) auxiliary space, Insertion sort and Heap Sort use O(1) auxiliary space. Space complexity of all these sorting algorithms is O(n) though.

Memory Usage while Execution

While executing, algorithm uses memory space for three reasons:

- **Instruction Space**

  It's the amount of memory used to save the compiled version of instructions.
- **Environmental Stack**

  Sometimes an algorithm(function) may be called inside another algorithm(function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm(function) is made.
- **Data Space**

  Amount of space used by the variables and constants.

  But while calculating the Space Complexity of any algorithm, we usually consider only Data Space and we neglect the Instruction Space and Environmental Stack

• <u>Application – I</u>

Finding the Nearest Guard in a Shopping Mall

3.1 Introduction to Problem

You are inside a Mall, where there are some walls, and some guards. You are in an open space, and you have to find the nearest guard to your position. Find Shortest distance from a guard in a Shopping Mall.

Given a matrix that is filled with ‘O’, ‘G’, and ‘W’ where ‘O’ represents open space, ‘G’ represents guards and ‘W’ represents walls of the Malls. Replace all of the O’s in the matrix with their shortest distance from a guard, without being able to go through any walls. Also, replace the guards with 0 and walls with -1 in output matrix.

```
O ==> Open Space
G ==> Guard
W ==> Wall
```

```
Input:
 O O  O  O  G
 O  W  W O  O
 O O  O  W O
 G  W  W  W O
 O O  O  O  G
```

```
Output:
 3  3  2  1  0
 2  -1  -1  2  1
 1  2  3  -1  2
 0  -1  -1 -1  1
 1  2  2  1  0
```

3.2 Approach to Solve

Dynamic Programming -

The idea is to do BFS. We first enqueue all cells containing the guards and loop till queue is not empty. For each iteration of the loop, we dequeue the front cell from the queue and for each of its four adjacent cells, if cell is an open area and its distance from guard is not calculated yet, we update its distance and enqueue it. Finally after BFS procedure is over, we print the distance matrix.

3.3 Coding Implementation

```
#include <bits/stdc++.h>
using namespace std;

// store dimensions of the matrix
#define M 5
#define N 5

// An Data Structure for queue used in BFS
struct queueNode
{
    // i, j and distance stores x and y-coordinates
    // of a matrix cell and its distance from guard
    // respectively
    int i, j, distance;
};

// These arrays are used to get row and column
// numbers of 4 neighbors of a given cell
int row[] = { -1, 0, 1, 0};
int col[] = { 0, 1, 0, -1 };

// return true if row number and column number
// is in range
bool isValid(int i, int j)
{
    if ((i < 0 || i > M - 1) || (j < 0 || j > N - 1))
        return false;

    return true;
}

// return true if current cell is an open area and its
// distance from guard is not calculated yet
bool isSafe(int i, int j, char matrix[][N], int output[][N])
{
    if (matrix[i][j] != 'O' || output[i][j] != -1)
        return false;
    return true;
}
```

```
// Function to replace all of the O's in the matrix
// with their shortest distance from a guard
void findDistance(char matrix[][N])
{
    int output[M][N];
    queue<queueNode> q;

    // finding Guards location and adding into queue
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
        {
            // initialize each cell as -1
            output[i][j] = -1;
            if (matrix[i][j] == 'G')
            {
                queueNode pos = {i, j, 0};
                q.push(pos);
                // guard has 0 distance
                output[i][j] = 0;
            }
        }
    }

    // do till queue is empty
    while (!q.empty())
    {
        // get the front cell in the queue and update
        // its adjacent cells
        queueNode curr = q.front();
        int x = curr.i, y = curr.j, dist = curr.distance;

        // do for each adjacent cell
        for (int i = 0; i < 4; i++)
        {
            // if adjacent cell is valid, has path and
            // not visited yet, en-queue it.
            if (isSafe(x + row[i], y + col[i], matrix, output)
                && isValid(x + row[i], y + col[i]))
            {
                output[x + row[i]][y + col[i]] = dist + 1;

                queueNode pos = {x + row[i], y + col[i], dist + 1};
                q.push(pos);
            }
        }
         // dequeue the front cell as its distance is found
        q.pop();
    }
       // print output matrix
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
            cout << std::setw(3) << output[i][j];
        cout << endl;
    }
}
int main()
{
    char matrix[][N] =
    {
        {'O', 'O', 'O', 'O', 'G'},
        {'O', 'W', 'W', 'O', 'O'},
        {'O', 'O', 'O', 'W', 'O'},
        {'G', 'W', 'W', 'W', 'O'},
        {'O', 'O', 'O', 'O', 'G'}
    };

    findDistance(matrix);
    return 0;
}
```

Output:

```
  3  3  2  1  0
  2 -1 -1  2  1
  1  2  3 -1  2
  0 -1 -1 -1  1
  1  2  2  1  0
```

3.4 Analyzing Complexity

Time complexity is O(MN) for a M x N matrix.

• <u>Application – II</u>

Platforms and Trains

• Introduction to Problem

We are given n-platform and two main running railway tracks for both directions. Trains which needs to stop at your station must occupy one platform for their stoppage and the trains which need not to stop at your station will run away through either of main track without stopping. Now, each train has three value first arrival time, second departure time and third required platform number. We are given m such trains you have to tell maximum number of train for which you can provide stoppage at your station.

```
Input : n = 3, m = 6
Train no.|  Arrival Time |Dept. Time | Platform No.
    1    |   10:00       |  10:30    |    1
    2    |   10:10       |  10:30    |    1
    3    |   10:00       |  10:20    |    2
    4    |   10:30       |  12:30    |    2
    5    |   12:00       |  12:30    |    3
    6    |   09:00       |  10:05    |    1
Output : Maximum Stopped Trains = 5
Explanation : If train no. 1 will left
to go without stoppage then 2 and 6 can
easily be accommodated on platform 1.
And 3 and 4 on platform 2 and 5 on platform 3.

Input : n = 1, m = 3
Train no.|Arrival Time|Dept. Time | Platform No.
    1    | 10:00      |  10:30    |    1
    2    | 11:10      |  11:30    |    1
    3    | 12:00      |  12:20    |    1
Output : Maximum Stopped Trains = 3
Explanation : All three trains can be easily
stopped at platform 1.
```

3.2 Approach to Solve

Greedy Algorithm -

If we start with a single platform only then we have 1 platform and some trains with their arrival time and departure time and we have to maximize the number of trains on that platform. So, for n platforms we will simply make n-vectors and put the respective trains in those vectors according to platform number. After that by applying greedy approach we easily solve this problem.

3.3 Coding Implementation

```
#include <bits/stdc++.h>
using namespace std;

// number of platforms and trains
#define n 2
```

```
#define m 5

// function to calculate maximum trains stoppage
int maxStop(int arr[][3])
{
    // declaring vector of pairs for platform
    vector<pair<int, int> > vect[n + 1];

    // Entering values in vector of pairs
    // as per platform number
    // make departure time first element
    // of pair
    for (int i = 0; i < m; i++)
        vect[arr[i][2]].push_back(
             make_pair(arr[i][1], arr[i][0]));

    // sort trains for each platform as per
    // dept. time
    for (int i = 0; i <= n; i++)
        sort(vect[i].begin(), vect[i].end());

    // perform activity selection approach
    int count = 0;
    for (int i = 0; i <= n; i++) {
        if (vect[i].size() == 0)
            continue;

        // first train for each platform will
        // also be selected
        int x = 0;
        count++;
        for (int j = 1; j < vect[i].size(); j++) {
            if (vect[i][j].second >=
                             vect[i][x].first) {
                x = j;
                count++;
            }
        }
    }
    return count;
}

// driver function
int main()
{
    int arr[m][3] = { 1000, 1030, 1,
                      1010, 1020, 1,
                      1025, 1040, 1,
                      1130, 1145, 2,
                      1130, 1140, 2
```

```
                        };
    cout << "Maximum Stopped Trains = " << maxStop(arr);

    return 0;
}
```

Output:

```
Maximum Stopped Trains = 3
```

3.4 Analyzing Complexity

Time complexity for this approach would be $O(N^2)$.

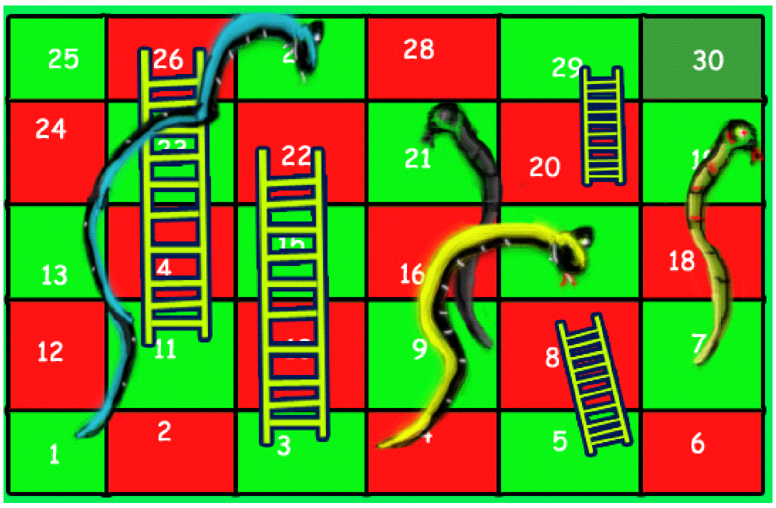
• Application – III

Snake and Ladder Problem

• Introduction to Problem

Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from source or 1st cell. Basically, the player has total control over outcome of dice throw and wants to find out minimum number of throws required to reach last cell.

If the player reaches a cell which is base of a ladder, the player has to climb up that ladder and if reaches a cell is mouth of the snake, has to go down to the tail of snake without a dice throw.



For example, consider the board shown, the minimum number of dice throws required to reach cell 30 from cell 1 is 3.

Following are the steps:

a) First throw two on dice to reach cell number 3 and then ladder to reach 22

b) Then throw 6 to reach 28.

c) Finally through 2 to reach 30.

There can be other solutions as well like (2, 2, 6), (2, 4, 4), (2, 3, 5).. etc.

4.2 Approach to Solve

Shortest Path using BFS -

The idea is to consider the given snake and ladder board as a directed graph with number of vertices equal to the number of cells in the board. The problem reduces to finding the shortest path in a graph. Every vertex of the graph has an edge to next six vertices if next 6 vertices do not have a snake or ladder. If any of the next six vertices has a snake or ladder, then the edge from current vertex goes to the top of the ladder or tail of the snake. Since all edges are of equal weight, we can efficiently find shortest path using Breadth First Search.

4.3 Coding Implementation

```
#include<iostream>
#include <queue>
using namespace std;

// An entry in queue used in BFS
struct queueEntry
```

```
{
    int v;     // Vertex number
    int dist;  // Distance of this vertex from source
};

// This function returns minimum number of dice throws required to
// Reach last cell from 0'th cell in a snake and ladder game.
// move[] is an array of size N where N is no. of cells on board
// If there is no snake or ladder from cell i, then move[i] is -1
// Otherwise move[i] contains cell to which snake or ladder at i
// takes to.
int getMinDiceThrows(int move[], int N)
{
    // The graph has N vertices. Mark all the vertices as
    // not visited
    bool *visited = new bool[N];
    for (int i = 0; i < N; i++)
        visited[i] = false;

    // Create a queue for BFS
    queue<queueEntry> q;

    // Mark the node 0 as visited and enqueue it.
    visited[0] = true;
    queueEntry s = {0, 0};  // distance of 0't vertex is also 0
    q.push(s);  // Enqueue 0'th vertex

    // Do a BFS starting from vertex at index 0
    queueEntry qe;  // A queue entry (qe)
    while (!q.empty())
    {
        qe = q.front();
        int v = qe.v; // vertex no. of queue entry

        // If front vertex is the destination vertex,
        // we are done
        if (v == N-1)
            break;

        // Otherwise dequeue the front vertex and enqueue
        // its adjacent vertices (or cell numbers reachable
        // through a dice throw)
        q.pop();
        for (int j=v+1; j<=(v+6) && j<N; ++j)
        {
            // If this cell is already visited, then ignore
            if (!visited[j])
            {
                // Otherwise calculate its distance and mark it
                // as visited
                queueEntry a;
                a.dist = (qe.dist + 1);
                visited[j] = true;

                // Check if there a snake or ladder at 'j'
                // then tail of snake or top of ladder
                // become the adjacent of 'i'
                if (move[j] != -1)
                    a.v = move[j];
                else
                    a.v = j;
                q.push(a);
            }
        }
    }

    // We reach here when 'qe' has last vertex
    // return the distance of vertex in 'qe'
    return qe.dist;
}

// Driver program to test methods of graph class
int main()
{
    // Let us construct the board given in above diagram
    int N = 30;
    int moves[N];
    for (int i = 0; i<N; i++)
        moves[i] = -1;

    // Ladders
    moves[2] = 21;
    moves[4] = 7;
    moves[10] = 25;
    moves[19] = 28;

    // Snakes
    moves[26] = 0;
    moves[20] = 8;
    moves[16] = 3;
    moves[18] = 6;

    cout << "Min Dice throws required is " << getMinDiceThrows(moves, N);
    return 0;
}
```

Output:

```
Min Dice throws required is 3
```

4.4 Analyzing Complexity

Time complexity of the above solution is O(N) as every cell is added and removed only once from queue. And a typical enqueue or dequeue operation takes O(1) time.

• <u>**References**</u>

Bibliography

- https://www.geeksforgeeks.org/
- https://www.codechef.com
- https://en.wikipedia.org/wiki/Algorithm
- https://www.hackerrank.com/

Text Books:

- **Narasimha karumanchi** “Data Structures and Algorithms made easy”.
- Anany Levitin, “Introduction to the Design and Analysis of Algorithms”, Third Edition, Pearson Education, 2012.

References:

- Donald E. Knuth, “The Art of Computer Programming”, Volumes 1& 3 Pearson Education,2009.
- Steven S. Skiena, “The Algorithm Design Manual”, Second Edition, Springer, 2008.