

SAMPLE OF THE STUDY MATERIAL

PART OF CHAPTER 1

Data Structure and Algorithm Analysis

Once an algorithm is given for a problem and decided to be correct, then an important step is to determine how much in the way of resources, such as time or space, the algorithm will be required.

The analysis required to estimate use of these resources of an algorithm is generally a theoretical issue and therefore a formal framework is required. In this framework, we shall consider a normal computer as a model of computation that will have the standard repertoire of simple instructions like addition, multiplication, comparison and assignment, but unlike the case with real computer, it takes exactly one time unit to do anything (simple) and there are no fancy operations such as matrix inversion or sorting, that clearly cannot be done in one time unit. We also always assume infinite memory.

Asymptotic Notation

The asymptotic notations are used to represent the *relative growth rate* between functions.

Big-Oh

Represent upper bound on the running time and the memory being consumed by the algorithms. $O(n)$ essentially conveys that the growth rate of running time/memory consumption rate will not be more than “n” for all inputs of size n for a given algorithm. However, it may be less than this.

More formally Big-Oh is defined as follows:

The function $f(n) = O(g(n))$ if and only if $f(n) \leq c.g(n)$ for all $n, n \geq n_0$ where c, n_0 are positive constants.

Thus, if $f(n) = O(g(n))$ statement is said to be true then the growth rate of function $g(n)$ is surely higher/equal to $f(n)$.

Example 1:

$$f(n) = 3n + 2$$

$$3n + 2 \leq 4n \text{ For all } n \geq 2$$

$$\therefore 3n + 2 = O(n) \text{ Here } c = 4, n_0 = 2$$

Example 2:

$$f(n) = n^2 + 3n + 5$$

$$n^2 + 3n + 5 \leq 2n^2 \text{ For } n \geq 3$$

$$\therefore n^2 + 3n + 5 = O(n^2) \text{ Here } c = 2, n_0 = 3$$

Example 3:

$$f(n) = 3.4^n + n^2$$

$$3.4^n + n^2 \leq 5.4^n$$

$$\therefore 3.4^n + n^2 = O(4^n) \text{ for } n \geq 1$$

Example 4:

$$3n^2 + 2n + 4 \neq O(n)$$

Because here doesn't exist any positive n_0 and c so that Big-Oh equation gets satisfied.

Remarks:

For the function $4n+3$,

$4n+3$ is $O(n)$

$4n+3$ is also $O(n^2)$ and $O(n^3)$

Even though $4n+3$ is $O(n^2)$ and $O(n^3)$ but the best answer is, $4n+3$ is $O(n)$ only, as $O(n)$ shows most tighter upper bound than the other in the question.

Big-Oh Properties

1. If $f(n)$ is $O(g(n))$ then $a.f(n)$ is also $O(g(n))$
2. If $f(n)$ is $O(g(n))$ and $h(n)$ is $O(p(n))$ then $f(n)+h(n) = O(\max(g(n), p(n)))$

Example 5:

$$f(n) = n^2, h(n) = \log n$$

$$n^2 + \log n = O(n^2)$$

3. If $f(n)$ is $O(g(n))$ and $h(n)$ is $O(p(n))$ then $f(n).h(n)$ is $O(g(n).p(n))$
4. If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$ then $f(n)$ is also $O(h(n))$
5. $\log n^k$ is $O(\log n)$
6. If $f(n)$ is any polynomial of degree m , $f(n) = a_m.n^m + a_{m-1}n^{m-1} + \dots + a_1n + a_0$, then $f(n)$ is $O(n^m)$

In general one should remember order of the following functions which will help while solving the relative growth rate of more complicated functions.

$$O(1), O(\log n), O(n), O(n \log n), O(n^2), O(n^3) \dots O(n^k), O(2^n)$$

All the functions are arranged in increasing order of growth rate.

If an algorithm has the time complexity $O(1)$, then the time complexity is said to be called constant, that means running time is independent of input size.

Big Omega (Ω):

Big Omega represents lower bound on the running time and the memory being consumed by the algorithms. $\Omega(n)$ essentially conveys that the growth rate of running time/memory consumption rate will not be less than “n” for all inputs of size n for a given algorithm. However, it may be greater than this.

More formally Big-Omega is defined as follows:

If $f(x)$ and $g(x)$ are any two functions and $f(x)$ is $\Omega(g(x))$,

iff $f(x) \geq c.g(x)$ for $x \geq k$ where c and k are any two positive constants.

Thus, if $f(x)$ is $\Omega(g(x))$ statement is said to be true then the growth rate of function $g(x)$ is surely lower/equal to $f(x)$.

Example 6:

$$f(n) = 2n - 4$$

$$2n + 4 \geq 2n \text{ for } n \geq 1$$

$$2n + y \geq 2n \text{ for } n \geq 1$$

$$\therefore 2n + 4 \text{ is } \Omega(n) \text{ here } c = 2, k = 1$$

we can also say that $2n + 4 > n$ for $n \geq 1$ then $c = 1, k = 1$

Remarks:

If $f(n)$ is $O(g(n))$, then $g(n)$ is $\Omega(f(n))$.

Example: 7

$$n^2 \text{ is } O(n^3)$$

$$n^3 \text{ is } \Omega(n^2)$$

Theta Notation (θ)

Theta represents tightest bound on the running time and the memory being consumed by the algorithms. $\theta(n)$ essentially conveys that the growth rate of running time/memory consumption rate will be equal to “n” for all inputs of size n for a given algorithm. It actually conveys that both lower and upper bounds are equal.

More formally **Theta** is defined as follows:

If $f(x)$ and $g(x)$ are two functions, and

if $f(x) = c.g(x)$ for $x > x_0$, then

$f(x) = \theta(g(x))$ here c and x_0 are two positive constants.

Thus, if $f(x) = \theta(g(x))$ statement is said to be true then the growth rate of function $g(x)$ is surely equal to $f(x)$ and not less and not more than $f(x)$.

Example 8:

$$f(n) = n^2 + n + 1; g(n) = 5n^2 + 1; h(n) = 2^{\log n} + n^2$$

Then, $f(n) = \theta(g(n))$ because both has same degree and hence will have same growth rate.

$f(n) = \theta(h(n))$ statement is also true because both has same degree and hence will have same growth rates.

$h(n)$ can be simplified as follows:

$2^{\log n}$ is n only,

$$\text{Let } 2^{\log n} = n \text{ -----} > 1$$

By taking log on both sides in equation 1.

$$\log n * \log_e 2 = \log_e n$$

Then, after simplifying the above equation

$$\log n = \log_e n / \log_e 2 = \log n.$$

Therefore, $h(n) = n + n^2$.

Remarks:

- If $f(x) = \theta(g(x))$ then $g(x)$ is also $\theta(f(x))$
- If $f(x) = \theta(g(x))$ we can say that $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$ and also $g(x)$ is $O(f(x))$ and $g(x)$ is $\Omega(f(x))$

Algorithm Definition

An algorithm is a finite set of steps or instructions to accomplish a particular task represented in a step by step procedure. Algorithm posses the following basic properties:

- An algorithm may have some input.
- An algorithm should produce at least one output.
- Each statement should be clear without any ambiguity.
- An algorithm in contrast to a program should terminate in a finite amount of time.

Algorithm Analysis

The following two components need to be analyzed for determining algorithm efficiency. While we have more than one algorithm for solving a problem then we really need to consider these two before utilizing one of them.

- **Running time complexity:** The time required for running an algorithm.
- **Space complexity:** The amount of space required at run-time by an algorithm for solving a given problem.

In general these measurements are expressed in terms of asymptotic notations, like Big-Oh, theta, Omega etc.

General Rules for Space Complexity Calculation

While computing space complexity we need to analyze whether the memory requirement is dependent on input size. Three things mainly need to be considered;

- 1) Maximum width of system stack which holds the activation records during the run time of the function.
- 2) Whether the size of activation record is dependent on input size.

3) And in each invocation how much memory being used from heap.

Then space complexity is as follows:

Max-system-stack-width*((size of the memory allocated on activation record which is dependent on n)+ (size of the memory allocated from heap which is dependent on n))

Example 9:

Consider the simple program fragment for analyzing space complexity.

```
int sum(int n)
{
    1 int partialSum = 0;
    2 for(int i = 0; i<n; i++)
    3 partialSum = partialSum + i*i*i;
    4 return partialSum;
}
```

Notice that the memory used by this program is absolutely independent of input size because this is non-recursive program and has only one activation record to be pushed on the system stack whose size is not going to change with n and also each invocation doesn't have any heap space requirement which is dependent on n. So whether n = 10, 20, 100, etc, the number of records to be pushed is O(1). Therefore Space Complexity is O(1).

Example 10:

Consider the recursive C program which prints the null terminated string in the reverse order.

```
void printRev(char *str)
{
    if( *str == '\0') return;
    printRev(str+1);
    printf("%c",*str);
}
```

Maximum width of the system stack is O(L) where L is string length. And the size of activation record is constant w.r.t length L. Therefore, space complexity is O(L).

General Rules for Running Time Calculations

Rule 1. FOR/WHILE loops

The running time of a **for/while** loop is =(the number of iteration perform)*(the running time of statements inside the loop).

Example 11:

Consider the simple program fragment whose running time cost is $O(n)$ where n is a positive integer.

```
int sum(int n)
{
    1 int partialSum = 0;
    2 for(int i = 1; i<= n; i++)
    3 partialSum = partialSum + i*i*i;
    4 return partialSum;
}
```

Lines 1 and 4 count one unit each. Line 3 counts for four units per time executed (two multiplications, one addition, and one assignment) and is executed n times, for a total of $4n$ units. Line 2 has hidden cost costs of initializing i , testing $i \leq n$ and incrementing i . The total cost of all these is 1 to initialize, $n+1$ for all the tests, and n for all the increments, which $2n+2$. Thus, total cost of $6n+4$, which is $O(n)$.

Rule 2 –Nested loops

The total running time of a statement inside a group of nested loop is the running time of the statement multiplied by the product of the sizes of all loops.

As an example following program fragment is $O(n^2)$.

```
for( i = 0; i< n ; i++)
    for( j = 0; j<n; j++)
        k++;
```

Rule 3 - Consecutive Statements

Just add running time of all these statements.

As an example, the following program fragment, which has $O(n)$ work followed by $O(n^2)$ work, is also $O(n^2)$.

```
for( i = 0; i< n ; i++)
    k++;
for( i = 0; i< n ; i++)
    for( j = 0; j<n; j++)
        k++;
```

Rule 4 – if/else

For the fragment

if(condition)

S1

else

S2

Running time of an if/else statement is never more than the running time of test plus the larger of the running times of S1 and S2.

Rule 5 – Recursive function

Deriving the recurrence relation and then solving it for getting the running time is always the better way than any other solution.

Example 12:

Consider the code given in Example 10 for printing null terminated string in reverse order.

Let T(n) be the running time of print Rev for which string length is n. Then, T(n-1) would represent running time of the same function which is given string of length n – 1.

Thus

$$T(n) = T(n - 1) + O(1)$$

Logarithm Time Complexity

There are several algorithms, which require logn cost in worst case. Binary Search, Heap ADT operations, etc are example of that.

Typically, an algorithm is O(logn) if it takes constant time to cut the problem size by a half. That’s what exactly happens in binary search algorithm.

```
int binarySearch(int a[], int len, int x)
{
    int low = 0 , high = len - 1;
    while(low <= high)
    {
        int mid = (low + high)/2;
        if( a[mid] < x)
            low = mid + 1;
        else if(a[mid] > x)
            high = mid - 1;
        else
            return mid;
    }
    return NOT_FOUND;
}
```

At first looks this algorithm give the illusion of $O(n)$ cost as we might think that since it has a while loop and that will always run for the entire length of the given array. A careful examination would expose that the loop will not run more than $O(\log n)$ times since in each iteration high or low getting adjusted such that the problem size decrease by half of the current size. The following recurrence relation can best represent the running time of binary search algorithm. $T(n) = T(n/2) + 1$, where $T(n)$ be the running time for n input elements.

Look at another interesting code given below.

```
sum = 0;
for( i = 1; i < n ; 2*i)
    sum++;
```

The i variable values getting incremented in each iteration such that its becoming double of current hence certainly would not take much longer than $\log n$ to reach its value as n or more.

Notion of Abstract Data Types

An abstract data type is a set of objects together with a set of operations. Abstract data types are mathematical abstraction; nowhere in an ADT's definition is there any mentioned of how the set of operation are implemented. Objects such lists, sets, and graph, along with their operations, can be viewed as abstract data types, just integers, real and Booleans are data types. Integers, real and Boolean have basic operations associated with them and so do abstract data types. For the lists ADT, we might have such operations as add, delete, search and etc.

Data Structure

A data structure is a model for organizing and storing data. Linked Lists, Stacks, and Trees are the examples of different data structures. Data structures are classified as either linear or non linear:

- A data structure is said to be linear if only one element can be accessed from an elements directly. **Examples:** Lists, Stacks, Queues.
- Non-linear if more than one element can be accessed from an element directly. **Examples:** Trees, Binary Heaps, Graphs.

Linked List

A *linked list* is a chain of structs or records called nodes. Each node has at least two members, one of which points to the next item or node in the list and one holds the data. These are defined as *Single Linked Lists* because they only point to the next item, and not the previous. Those that do point to both are called *Doubly Linked Lists*. According to this definition, we could have our record holding anything we want! List is a data structure, which allows insertion and deletion at any place in the list depending on the application need.

The only drawback is that each record must be an instance of the same structure. This means that we couldn't have a record with a char pointing to another structure holding a short, a char array, and a long. Again, they have to be instances of the same structure for this to work. Another cool aspect is that each structure can be located anywhere in memory; each node doesn't have to be linear in memory! *However because of this nature the binary search can't be performed on the lists even if items are stored in a sorted manner hence element lookup will not take less than $O(n)$ in worst case irrespective of type of linked list.*

The basic operations of linear linked list include insert, delete, and search. Most of the operation on the list would require traversal over the list. This is actually the most time consuming part in dealing with Singly Linked Lists. This is because we can't immediately access the previous node of a given member in constant time, like when we want to delete a node and reconnect the node before to the node after the one being deleted.

Applications of Linked List

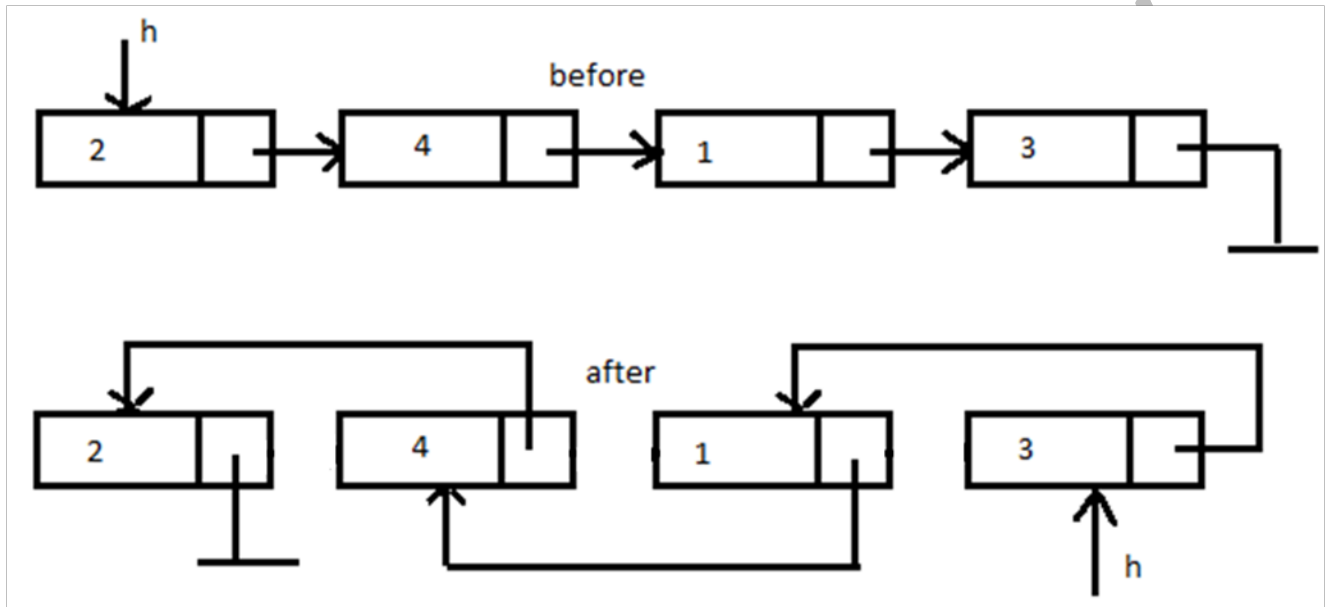
- Linked lists are used widely for implementing many data structures and applications efficiently. Some of the usage are listed below.
- Implementing Stack and Queue efficiently .
- Implementing big numbers addition, multiplication and factorial efficiently.

Reversing Linked List

This algorithm basically reverses a list such that successor becomes predecessor and predecessor becomes successor.

Example 13:

The following diagram illustrates linked list state before and after reverse.



```

Node* reverseList(Node* list)
{
    Node* lastVisited = NULL, *current = list, *nextCurrent;
    while(current)
    {
        1. nextCurrent = current->next;
        2. current->next = lastVisited
        3. lastVisited = current
        4. current = nextCurrent;
    }
}
    
```

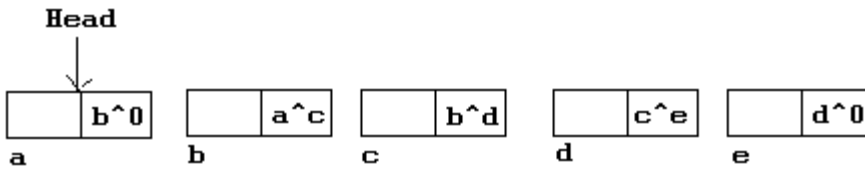
The algorithm essentially reverses each node and thus makes reversed the entire list. Last visited node is stored since required in each current iteration so that current node can be reversed easily.

Running time is quite clear which is $O(n)$ as “while” loop runs for the entire length of the list and each iteration takes constant time. Space complexity is $O(1)$ since non-recursive and doesn't take extra memory neither on system stack nor on heap to reverse the links except few constant amount of variables.

Doubly Linked List Feature-Using Node Structure with One Link Field

In this linked list implementation each node link field is assigned the resultant of the bit-wise XOR between the successor address and predecessor address of that node.

The given picture illustrates how the each node is connected here.



Unlike in usual singly linked list implementation, accessing the successor and predecessor will be not straightforward and the following computation would be required to reach successor and predecessor.

Let's say two adjacent nodes are p (pointing to node c in the above picture) and q (pointing to node d in the above picture).

Then the expression for getting access to node e is:

$p \wedge (q \rightarrow \text{next})$ and this would be certainly e which is the address of successor of q.

And the expression for getting access to node b is:

$(p \rightarrow \text{next}) \wedge q$ and this would be certainly b which is the address of predecessor of p

Traversal on such list would require housekeeping of the last visited node in order to be able to reach the next/prev of the current node.

Example 14:

This program fragment prints all values of the given list:

```
void traversal(Node *list)
{
    struct node *current = list, *lastVisited=0, *tmp;
    while(current)
    {
        printf("%d", current->value);

        tmp = current;
        current = (struct node*)((unsigned)current ->next ^ (unsigned)lastVisited);

        lastVisited = tmp;
    }
}
```

Assignment:

1. For each of the following program fragments, give an analysis of the running time (Big-Oh).

a)

```
sum = 0;
for( i = 0; i < n ; i++)
    for( j = 0; j < n*n; j++)
        sum++;
```

(A) $O(1)$

(B) $O(n^2)$

(C) $O(n)$

(D) $O(n^3)$

b)

```
sum = 0;
for( i = 0; i < n ; i++)
    for( j = 0; j < i; j++)
        sum++;
```

(A) $O(1)$

(B) $O(n^2)$

(C) $O(n)$

(D) $O(n^3)$

c)

```
sum = 0;
for( i = 0; i < n ; i++)
    for( j = 0; j < i*i; j++)
        for( k = 0; k < j; k++)
            sum++;
```

(A) $O(1)$

(B) $O(n^5)$

(C) $O(n)$

(D) $O(n^3)$

d)

```
sum = 0;
for( i = 1; i < n ; i++)
    for( j = 1; j < i; j++)
        if( j % i == 0)
            for( k = 0; k < j; k++)
                sum++;
```

- (A) $O(1)$ (C) $O(n^2)$
(B) $O(n)$ (D) $O(n^3)$

e)

Procedure A(n)

```
{
    if(n <= 2) return 1;
    else return A(√n);
}
```

- (A) $O(1)$ (C) $O(n^2)$
(B) $O(\log n)$ (D) $O(\log \log n)$

f)

```
sum = 0;
for( i = 1; i < n ; 3*i)
    sum++;
```

- (A) $O(1)$ (C) $O(n^2)$
(B) $O(\log_3 n)$ (D) $O(\log n)$

2. Given an array of 0's and 1's, what would be running time of an efficient algorithm to re-arrange the array such that all 0's precedes the all 1's.
- (A) $O(n^2)$ (C) $O(n)$
(B) $O(n \log n)$ (D) $O(1)$

GATE Questions CS:

1. The following C function takes a singly-linked list as input argument. It modifies the list by moving the last element to the front of the list and returns the modified list. Some part of the code is left blank.

```
typedef struct node {
    int value;
    struct node *next;
} Node;
Node *move_to_front (Node *head) {
    Node *p, *q;
    if ( (head == NULL) || (head -> next == NULL) ) return head;
    q = NULL; p = head;
    while (p->next != NULL) {
        q = p;
        p = p -> next;
    }
    return head;
}
```

Chose the correct alternative to replace the blank line.

- (A) $q = \text{NULL}; p -> \text{next} = \text{head}; \text{head} = p;$
(B) $q -> \text{next} = \text{NULL}; \text{head} = p; p -> \text{next} = \text{head};$
(C) $\text{head} = p; p -> \text{next} = q; q -> \text{next} = \text{NULL};$

(D) q -> next = NULL; p -> next = head; head = p;

[GATE- CS-2010]

GATE Questions IT:

1. When $n = 2^{2^k}$ for some $k \geq 0$, the recurrence relation $T(n) = \sqrt{2} T(n/2) + \sqrt{n}$, $T(1) = 1$ evaluates to
- (A) $\sqrt{n} (\log n + 1)$ (C) $\sqrt{n} \log \sqrt{n}$
 (B) $\sqrt{n} \log n$ (D) $n \log \sqrt{n}$

[GATE-IT-2008]

Answer Keys

Assignment:

| | | | | | | | | | | | | | |
|------|---|------|---|------|---|------|---|------|---|------|---|---|---|
| 1(a) | D | 1(b) | B | 1(c) | B | 1(d) | C | 1(e) | D | 1(f) | B | 2 | C |
|------|---|------|---|------|---|------|---|------|---|------|---|---|---|

GATE Questions CS:

| | |
|---|---|
| 1 | D |
|---|---|

GATE Questions IT:

| | |
|---|---|
| 1 | A |
|---|---|

Explanations:

Assignment

1.

- a) [Ans. D]
 b) [Ans. B]
 c) [Ans. B]

$O(n^5)$. Hint: The total number of iterations approximately is

$$1 + (1 + 2 + 3 + 4) + (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9) + \dots + (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + \dots + n^2) = 1 + 2^4 + 3^4 + \dots + n^4 - (1 + 2^2 + 3^2 + \dots + n^2) = n^5/5 + n^4/2 + n^3/3 - n/30 - n^3/3 - n^2/2 - n/6 = O(n^5)$$

- d) [Ans. C]

$O(n^2)$. Hint: innermost for loop doesn't run at all.

- e) [Ans. D]

Let running time be $T(n) = T(\sqrt{n}) + 1$

Let converts the non-linear recurrence relation to linear one by replacing n by 2^l , where $l = 2^k$, thus $k = \log \log(n)$. So now the new recurrence relation would be $T(2^l) = T(2^{l/2}) + 1$. By applying the substitution method recursively the we would see $T(2^l) = T(2) + k$ as the simplified final equation, where $k = \log \log(n)$ and $T(2)$ would be some constant value. Therefore the running time is $O(\log \log(n))$

- f) [Ans. B]

The running time of above can be represented by the below recurrence relation;

$$T(n) = T(n/3) + 1 \text{ where } n = 3^k$$

By solving it we get $T(n) = O(\log_3 n)$.

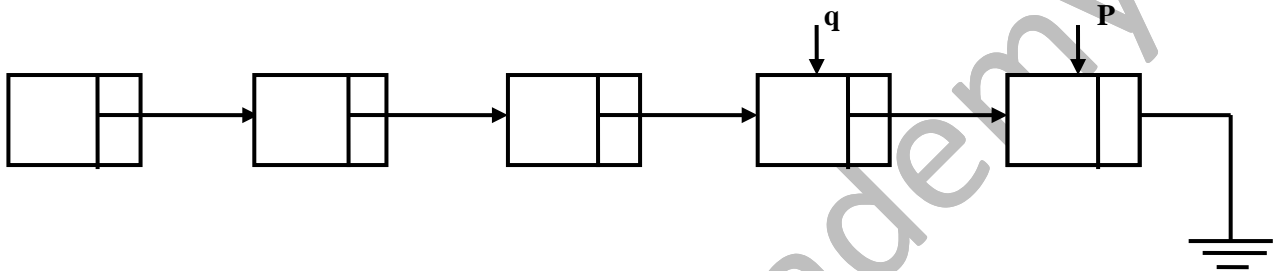
2. [Ans. C]

First count number of 0's. From that we can find out 1's present in the array. Then write the 0's followed by 1's for the counted times.

GATE Questions CS:

1. [Ans. D]

When 'while' loop is over, the states of P and q variables are as given in the diagram. Now node q



will become last node, Thus, $q \rightarrow \text{next} = \text{NULL}$ will be required

Since, P is inserting in the front of the list $P \rightarrow \text{next} = \text{head}$ is required and P become first node of the modified list hence $\text{head} = P$ is required for the successful operation.

GATE Questions IT:

1. [Ans. A] Simple recurrence relation based question