**G. Laboratory Experiments**

**Total: 30 Hours (P)**

**Part – 1**

**Task 1: Algorithmic problem solving 1**

**AIM**

Calculating the time complexities for the given algorithms and estimating the orders of growth

**EX1**

**Constant Complexity - O(1)**

* If an algorithm’s time complexity is constant, it means that it will always run in the same amount of time, no matter the input size. For example, if we want to get the first item of an array, it doesn’t matter how big the input size is.

**#include <stdio.h>**

**Time Complexity Calculation:**

The time complexity of the above-given program is O(1), as this program consists of only assignment, arithmetic operations and all those will be executed only once.

**int main()**

**{**

**int a = 4;**

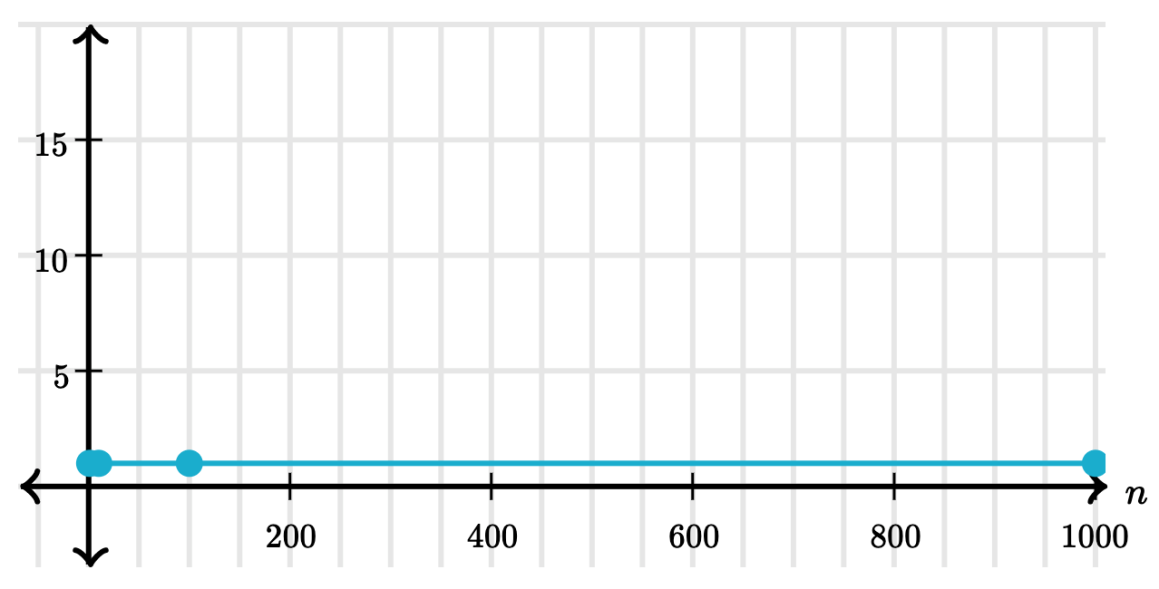
**int b = 6;**

**int c;**

**c = a + b;**

**printf(%d, c);**

**}**



**EX2**

**Linear time – O(n)**

An algorithm is said to have a linear time complexity when the running time increases linearly with the length of the input. When the function involves checking all the values in input data, with this order O(n).

**for (int i = 1; i <= n; i += c)**

**Time Complexity Calculation:**

loop variables is incremented / decremented by a constant amount

**{**

**// some expressions**

**}**

**function(int n) {**

**if (n==1)**

**Time Complexity Calculation:**

Even though the inner loop is bounded by n, but due to break statement it is executing only once.

**return;**

**for (int i=1; i<=n; i++)**

**{**

**for (int j=1; j<=n; j++)**

**{**

**printf("\*");**

**break;**

**}     }**

**Time Complexity Calculation:**

         for (int i = 0; i < n; i++) // Executed n times

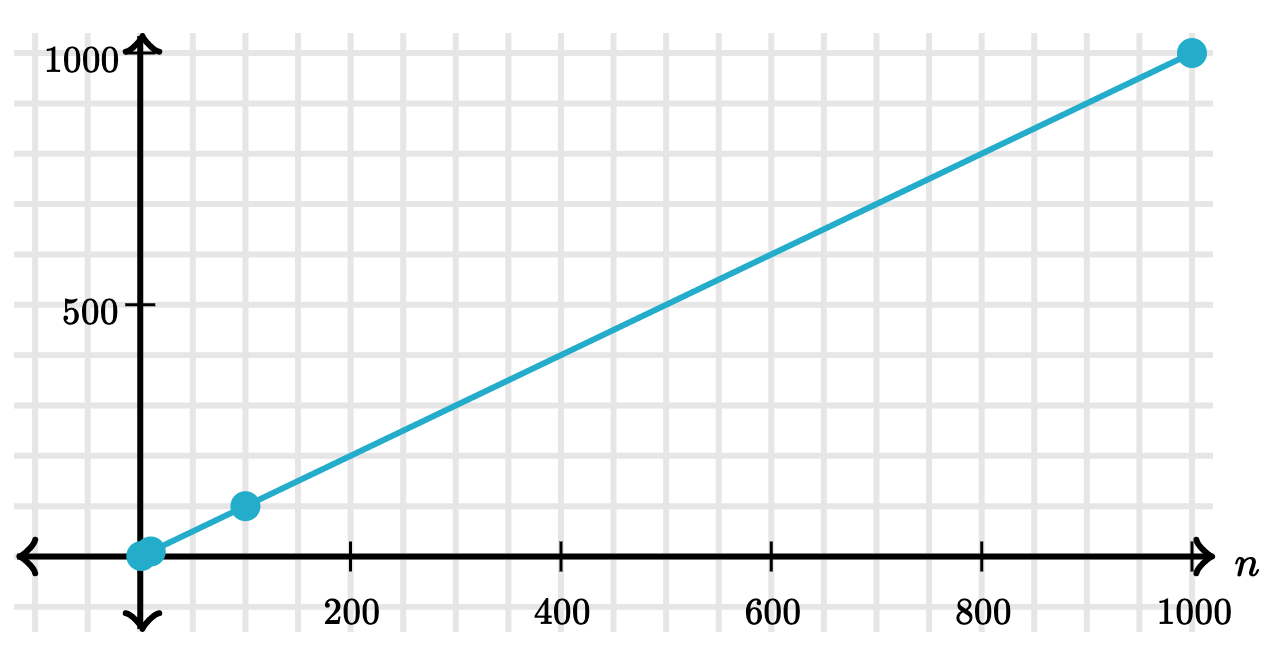
Complexity will be O(n)

**int m=0;**

**for (int i = 0; i < n; i++) {**

**m=m+1;**

**}**



**EX3**

**Quadratic time – O (n2)**

An algorithm is said to have a non-linear time complexity where the running time increases non-linearly (n2) with the length of the input. Generally, nested loops come under this order where one loop takes O(n) and if the function involves a loop within a loop, then it goes for O(n)\*O(n) = O(n2) order.

**#include <stdio.h>**

**int main()**

**Time Complexity Calculation:**

In the given snippet, the first & the second for loops get executed n times individually. So the time complexity accounts to n\*n = O(n2)

**{**

**int i,j,n = 8;**

**for (int i = 1; i <= n; i++)**

**{**

**for (int j = 1; j <= n; j++)**

**{**

**printf("VELTECH\n");**

**} } }**

**for (int i = 1; i <=n; i += c)**

**Time Complexity Calculation:**

* nested loops is equal to the number of times the innermost statement is executed
* Selection sort and Insertion Sort have O(n2) time complexity

**{**

**for (int j = 1; j <=n; j += c)**

**{**

**// some expressions**

**}**

**}**

**int m=0;**

**// Executed n times**

**for (int i = 0; i < n; i++) {**

**m=m+1;**

**}**

**// outer loop executed n times**

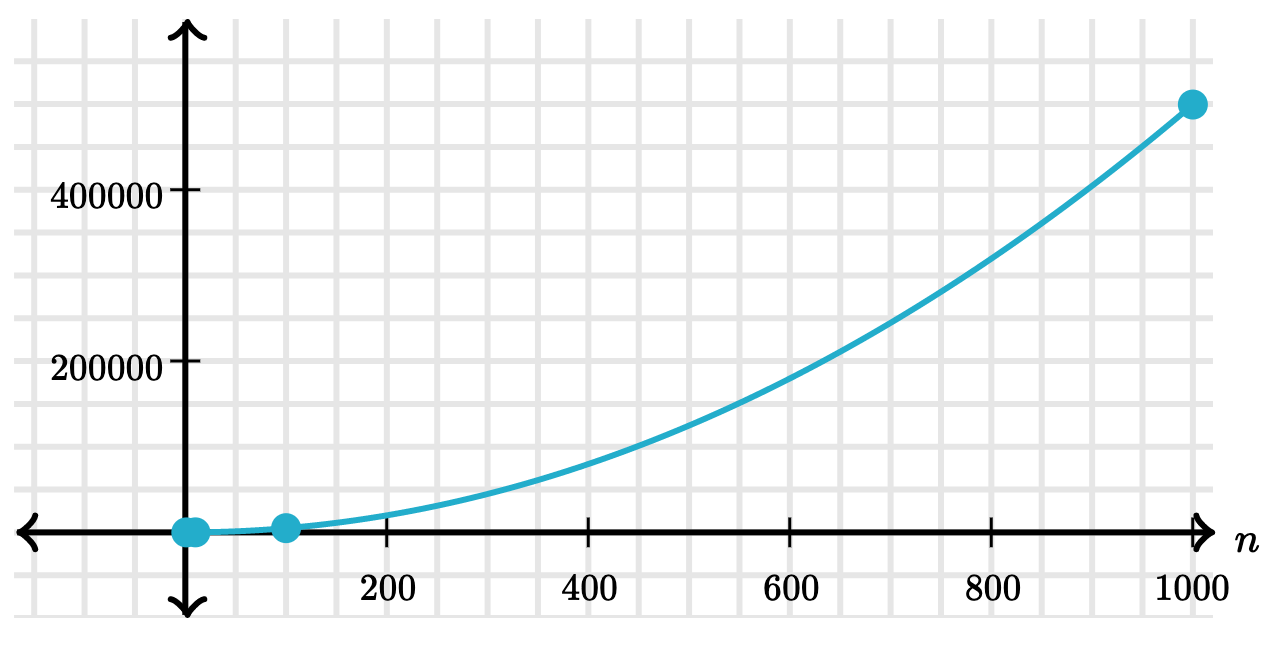
**for (int i = 0; i < n; i++) {**

**// inner loop executed n times**

**for(int j = 0; j < n; j++)**

**m=m+1;**

**}**



**EX4**

**Logarithmic time – O (log n)**

An algorithm is said to have a logarithmic time complexity when it reduces the size of the input data in each step. This indicates that the number of operations is not the same as the input size. The number of operations gets reduced as the input size increases. Algorithms are found in binary trees or binary search functions. This involves the search of a given value in an array by splitting the array into two and starting searching in one split. This ensures the operation is not done on every element of the data.

**for (int i = 1; i <=n; i \*= c)**

**{**

**Time Complexity Calculation:**

* The loop variables is divided / multiplied by a constant amount
* Binary Search has O(Logn) time complexity.

**// some expressions**

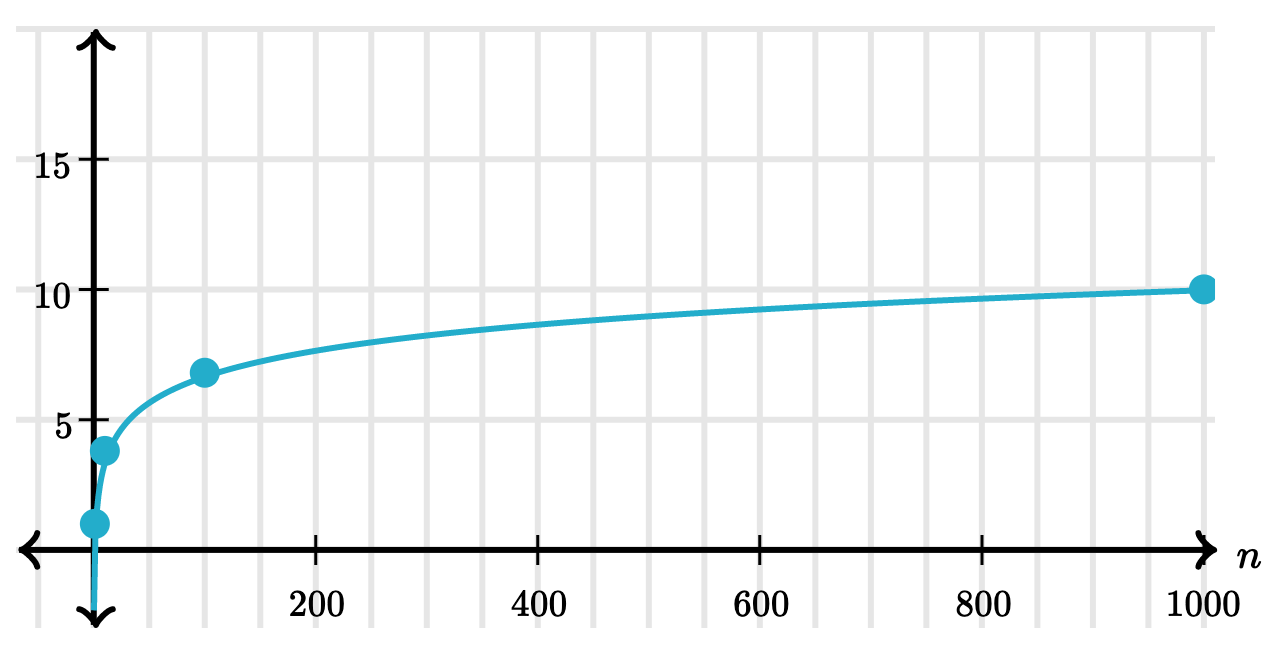
**}**

**for (int i = n; i > 0; i /= c)**

**{**

**// some expressions**

**}**



**EX5**

**O(nlogn) — A log-linear algorithm**

O(nlogn) is known as log-linear**complexity.**O(n logn) implies that logn operations will occur n times. O(n logn) time is common in recursive sorting algorithms, sorting algorithms using a binary tree sort, and most other types of sorts

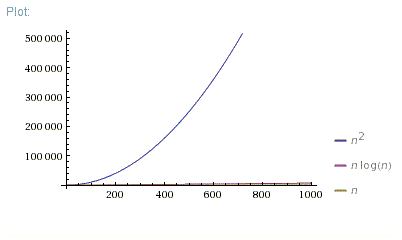
This time complexity is similar to O(log n) but it executes inside a loop that has time complexity O(n).

  int m=0;

**Time Complexity Calculation:**

Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation. **T(n) = 2T(n/2) + O(n)** The solution of the above recurrence is **O(nLogn)**. The list of size N is divided into a max of Logn parts, and the merging of all sublists into a single list takes O(N) time, the worst-case run time of this algorithm is O(nLogn) Best Case Time Complexity: **O(n\*log n)** Worst Case Time Complexity: **O(n\*log n)** Average Time Complexity: **O(n\*log n)** The time complexity of MergeSort is O(n\*Log n) in all the 3 cases (worst, average and best) as the mergesort always divides the array into two halves and takes linear time to merge two halves.

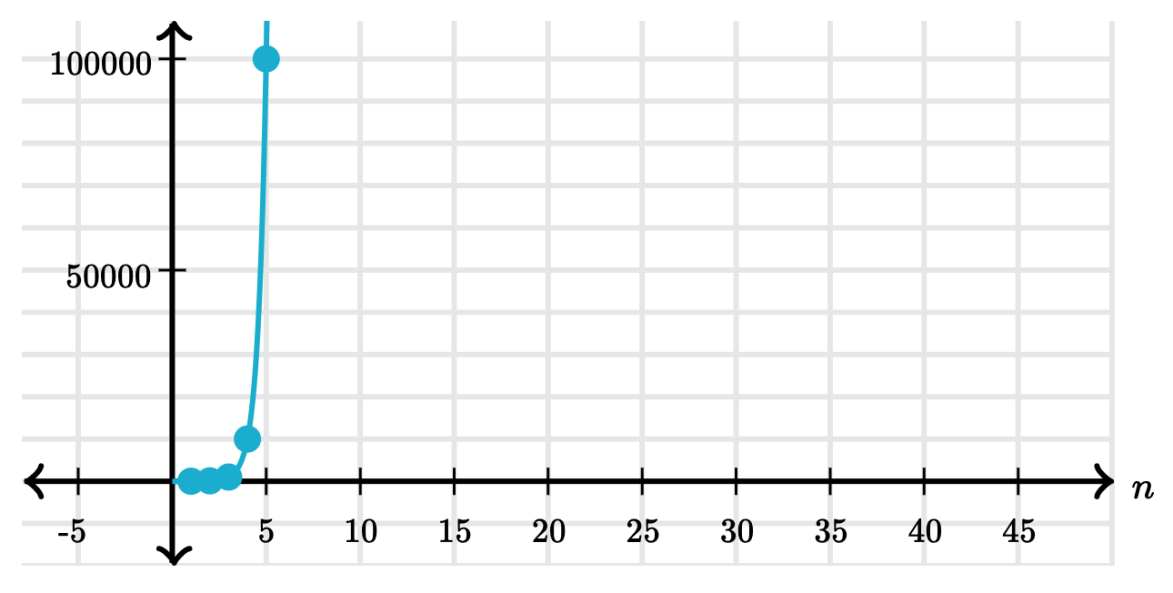
MERGE SORT



**EX5**

**Exponential time**

Exponential growth is the opposite of Logarithmic growth. Logarithmic growth becomes efficient after a certain number of inputs but exponential gets slower and inefficient as the input size grows



**EX5**

**Cubic time – O (n3)**

An algorithm is said to run in cubic time if the running time of the three loops is proportional to the cube of N. When N triples, the running time increases by N \* N \* N.

**int m=0;**

**2\*n = n3**

**// outer loop executed n/2 times**

**for (int i = n/2; i < n; i++) {**

**// middle loop executed n/2 times**

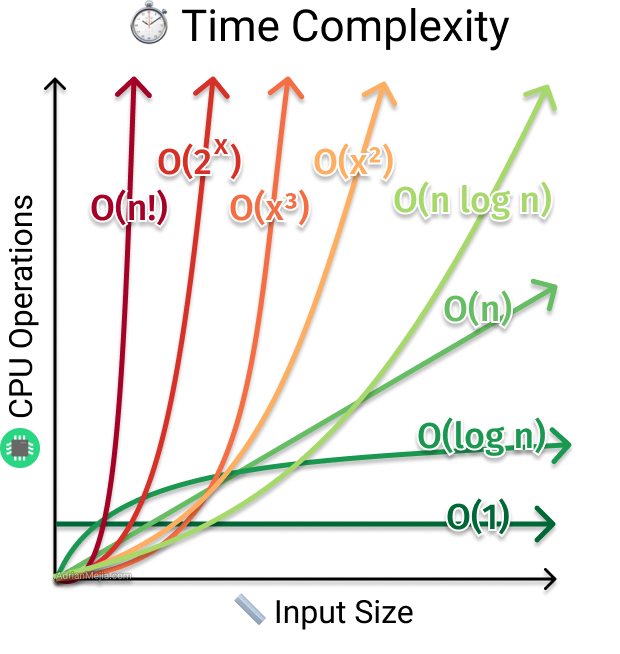
**for(int j = n/2; j < n; j++)**

**// inner loop executed n times**

**for(int k=0;k < n; k++ )**

**m=m+1;**

**}**



**Result**

Thus calculating the time complexities for the given coding and estimating the orders of growth was learn successfully

**Task 2: Algorithmic problem solving 2**

**DEFINITION** A function *t (n)* is said to be in *O(g(n))*, denoted *t (n)* € *O(g(n)),*if *t (n)* is bounded above by some constant multiple of *g(n)* for all large *n,* i.e., if there exist some positive constant *c* and some nonnegative integer *n*0 such that

***t (n)* ≤ *cg(n)* for all *n* ≥ *n*0*.***

Big oh = F(n)<=C\*g(n)

**Orders of Growth**

**1 log2n n nlog2n n2 n3 2n n!**

**Orders of Growth table**

|  |  |
| --- | --- |
| **Class** | **name** |
| **O(1)** | **constant** |
| **O(log log n)** | **bi-logarithmic or log log n** |
| **O(log n)** | **logarithmic or log n** |
| **O((log n)k) or O(logkn(** | **poly-logarithmic** |
| **O(n)** | **linear** |
| **O(n log n)** | **linear logarithmic or n log n** |
| **O(n2)** | **quadratic** |
| **O(n2 log n)** | **quadratic logarithmic** |
| **O(n3)** | **cubic** |
| **O(2n)** | **base-2 exponential** |
| **O(en)** | **natural exponential** |
| **O(3n)** | **base-3 exponential** |
| **O(n!)** | **factorial** |
| **O(nn)** | **hyper-exponential** |

|  |  |
| --- | --- |
| **Big-O Notation** | **Examples of Algorithms** |
| **O(1)** | **Accessing an array element, Constant loops, Push, Pop, Enqueue (if there is a tail reference), Dequeue** |
| **O(log(n))** | **Binary search** |
| **O(n)** | **Linear search, Summing a 1D-array** |
| **O(n log(n))** | **Heap sort, Quick sort (average-case), Merge sort** |
| **O(n2)** | **Selection sort, Insertion sort, Bubble sort, Summing a 2D-array of size n\*n** |
| **O(n3)** | **Matrix multiplication** |
| **O(2n)** | **Towers of Hanoi, Recursive Fibonacci, Finding the (exact) solution to the traveling salesman problem (TSP) using dynamic programming** |
| **O(n!)** | **Solving the traveling salesman problem via brute-force search** |
| **O(nn)** | **Ackermann function** |

**Ex1**

loglinear complexity

O(nlogn), also known as loglinear complexity, implies that logn operations will occur n times. It’s commonly used in recursive sorting algorithms and binary tree sorting algorithms.

**Coding**

#include <stdio.h>

// lets take a[5] = {32, 45, 67, 2, 7} as the array to be sorted.

**// merge sort function**

void mergeSort(int a[], int p, int r)

{

int q;

if(p < r)

{

q = (p + r) / 2;

mergeSort(a, p, q);

mergeSort(a, q+1, r);

merge(a, p, q, r);

}

}

// function to merge the subarrays

void merge(int a[], int p, int q, int r)

{

int b[5]; //same size of a[]

int i, j, k;

k = 0;

i = p;

j = q + 1;

while(i <= q && j <= r)

{

if(a[i] < a[j])

{

b[k++] = a[i++]; // same as b[k]=a[i]; k++; i++;

}

else

{

b[k++] = a[j++];

}

}

while(i <= q)

{

b[k++] = a[i++];

}

while(j <= r)

{

b[k++] = a[j++];

}

for(i=r; i >= p; i--)

{

a[i] = b[--k]; // copying back the sorted list to a[]

}

}

// function to print the array

void printArray(int a[], int size)

{

int i;

for (i=0; i < size; i++)

{

printf("%d ", a[i]);

}

printf("\n");

}

int main()

{

int arr[] = {32, 45, 67, 2, 7};

int len = sizeof(arr)/sizeof(arr[0]);

printf("Given array: \n");

printArray(arr, len);

// calling merge sort

mergeSort(arr, 0, len - 1);

printf("\nSorted array: \n");

printArray(arr, len);

return 0;

}

### **Complexity Analysis of Merge Sort**

Merge Sort is quite fast, and has a time complexity of O(n\*log n). It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

**Ex 2.1** Sort all the functions below in increasing order of asymptotic (Big O) growth. If some

have the same asymptotic growth, then be sure to indicate that. Note: log means base 2.

5n, 4 logn,4log log n,n4,n1/2log4n, n1/2 log4n,(log n)5logn, nlogn, 5n, 4n4, 44n, 55n, 55n, nn1/5, nn/4,(n/4)n/4

**Solution:** 4 log log n < 4 log n < n1/2 log4n < 5n < n4 < (log n)5logn < nlogn < nn1/5< 5n < 55n < (n/4)n/4 < nn/4 < 4n4 < 44n < 55n

**Ex 2.2** Order the functions according to their growth from slowest to fastest growing

6n3, n log6n, 4n, 8n2, log2n, nlog2n, 64, 82n

**Solution:** 64, log2n, 4n, n log6n, nlog2n, 8n2, 6n3, 82n

**Result:**

Thus Order the functions according to their growth from slowest to fastest growing using Big O has learnt successfully