Sardar Patel Institute of Technology

Name: Atharva Bilonikar

Uid : 2021700011

CSE DS D1

DAA Experiment 1 A

Problem Statement:

To implement the various functions e.g. linear, non-linear, quadratic, exponential etc.

Algorithm:

The code starts with the inclusion of two libraries: stdio.h and math.h, which are used for input/output operations and mathematical operations, respectively.

It then defines ten functions:

a. calc_loge_loge: returns the logarithm of the logarithm of an integer n.

b. calc_log2: returns the logarithm base 2 of an integer n.

c. calc_n_log2_n: returns the product of n and logarithm base 2 of n.

d. calc_log2_fact: returns the logarithm base 2 of the factorial of an integer n.

e. calc_log2_square: returns the square of the logarithm base 2 of an integer n.

f. calc_linear: returns the integer n as is.

g. calc_cubic: returns the cube of an integer n.

h. calc_power2: returns 2 raised to the power of an integer n.

i. calc_power32: returns (3/2) raised to the power of an integer n.

j. calc_exp: returns the exponential of an integer n.

The main function starts with the declaration of an integer variable 'n'.

It then prints a header for the table of results.

It then uses a for loop to calculate and print the results of the above functions for values of n from 1 to 100.

Theory:

Time Complexity: The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve given problem is called ***time complexity*** of the algorithm. Time complexity is very useful measure in algorithm analysis.

It is the time needed for the completion of an algorithm. To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed

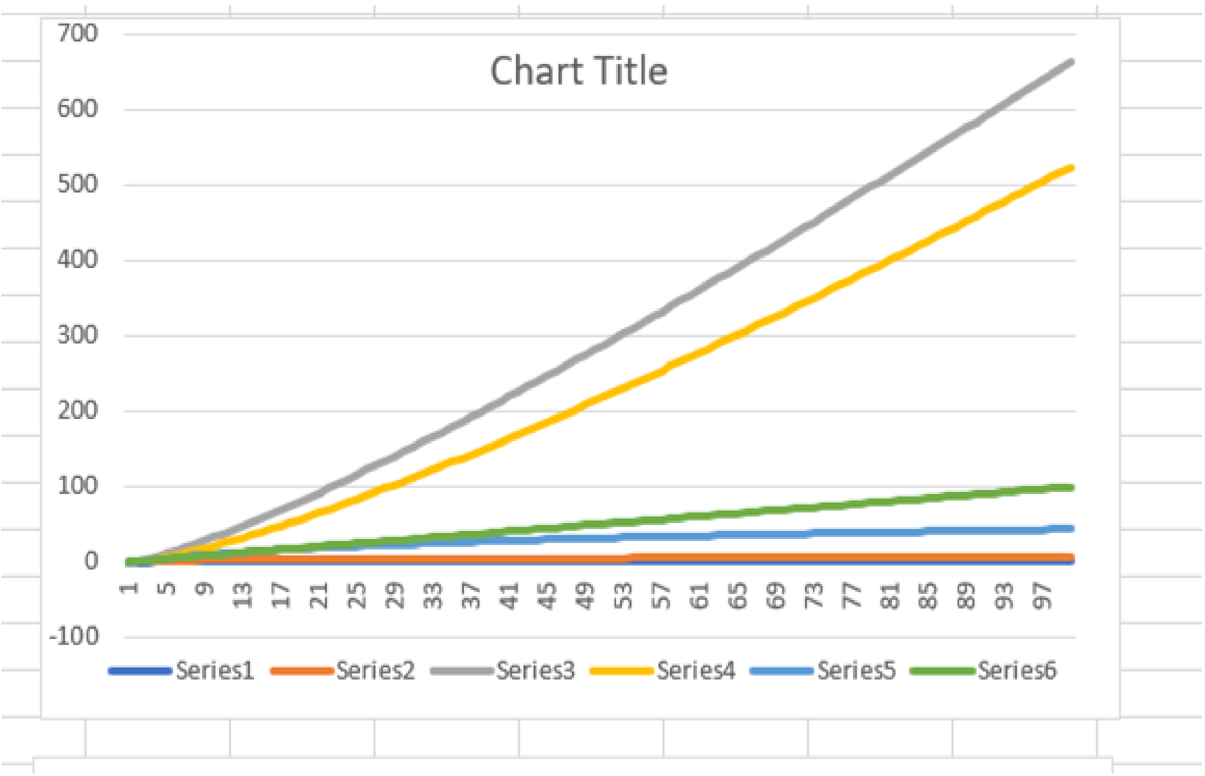
Order of growth is how the time of execution depends on the length of the input. In the above example, it is clearly evident that the time of execution quadratically depends on the length of the array. Order of growth will help to compute the running time with ease

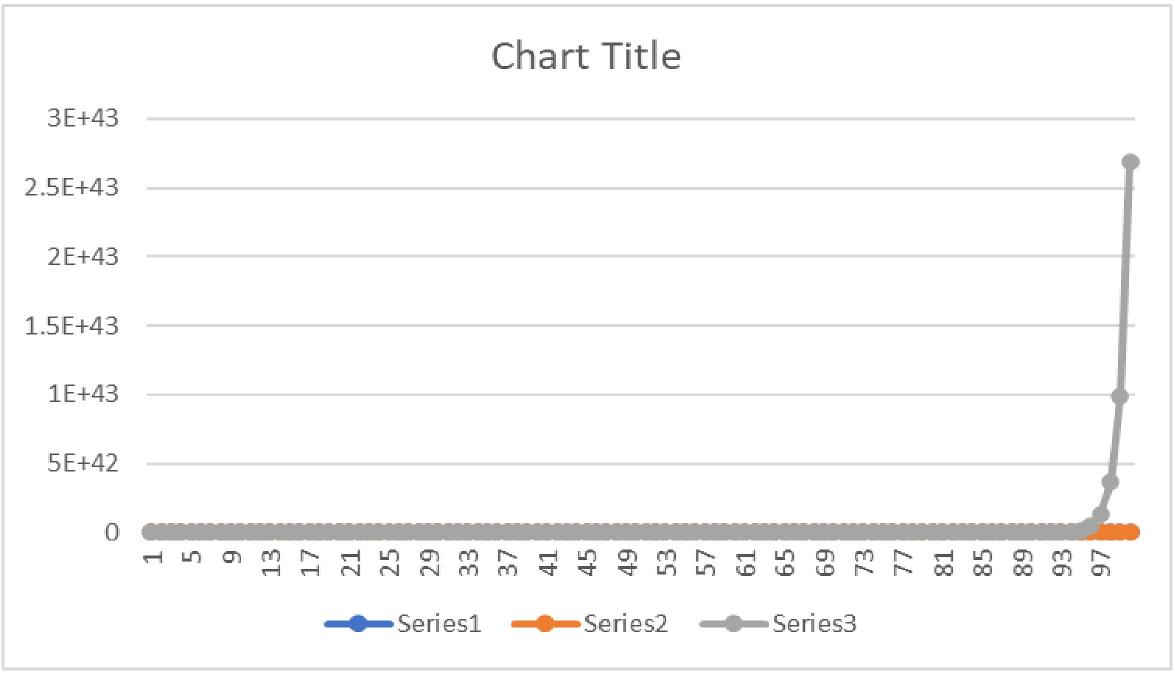
Code :

```
#include <stdio.h>
#include <math.h>

double calc_loge_loge(int n) {
  return log(log(n));
```

```
}

double calc_log2(int n) {
  return log2(n);
}

double calc_n_log2_n(int n) {
  return n * log2(n);
}

double calc_log2_fact(int n) {
  int i;
  double fact = 0;
  for (i = 1; i <= n; i++) {
    fact += log2(i);
  }
  return fact;
}

double calc_log2_square(int n) {
  return pow(log2(n), 2);
}

double calc_linear(int n) {
  return n;
```

```
}


double calc_cubic(int n) {
  return n * n * n;
}


double calc_power2(int n) {
  return pow(2, n);
}


double calc_power32(int n) {
  return pow(1.5, n);
}


double calc_exp(int n) {
  return exp(n);
}


int main() {
  int n;



printf("n\tloge(loge(n))\tlog2(n)\tn*log2(n)\tlog2(n!)\t(log2(n))^2\tn\t
n^3\t2^n\t(3/2)^n\te^n\n");


  for (n = 1; n <= 100; n++) {
    double loge_loge_n = calc_loge_loge(n);

    double log2_n = calc_log2(n);

    double n_log2_n = calc_n_log2_n(n);

    double log2_fact_n = calc_log2_fact(n);

    double log2_square_n = calc_log2_square(n);

    double linear_n = calc_linear(n);

    double cubic_n = calc_cubic(n);

    double power2_n = calc_power2(n);

    double power32_n = calc_power32(n);

    double exp_n = calc_exp(n);


printf("%d\t%.1lf\t\t%.1lf\t%.1lf\t%.1lf\t%.1lf\t%.1lf\t%.1lf\t%.1lf\t
%.1lf\t%.1lf\n", n, loge_loge_n, log2_n, n_log2_n, log2_fact_n,
log2_square_n, linear_n, cubic_n, power2_n,power32_n,exp_n);

    // printf("%.3f\n",exp_n);

  }

    return 0;

}
```





Observations on the growth of functions and comparison:

1.loge(loge(n)): logarithmic growth, increases slowly as n increases. It grows slower than log2(n) and nlog2(n).

2.log2(n): logarithmic growth, increases slowly as n increases. It grows slower than nlog2(n) and faster than loge(loge(n)).

3. nlog2(n): linear growth, increases linearly as n increases. It grows faster than log2(n) and loge(loge(n)).

4. log2(n!): logarithmic growth, increases slowly as n increases. It grows slower than all other functions except loge(loge(n)).

5. (log2(n))^2: polynomial growth, increases more quickly as n increases. It grows faster than log2(n) and loge(loge(n)), but slower than n, n^3, 2^n, (3/2)^n and e^n.

6. n: linear growth, increases linearly as n increases. It grows faster than log2(n), loge(loge(n)) and (log2(n))^2 but slower than nlog2(n), n^3, 2^n, (3/2)^n and e^n.

7. n^3: polynomial growth, increases more quickly as n increases. It grows faster than n, log2(n), loge(loge(n)), (log2(n))^2 but slower than nlog2(n), 2^n, (3/2)^n and e^n.

8. 2^n: exponential growth, increases extremely quickly as n increases. It grows faster than all other functions.

9. (3/2)^n: exponential growth, increases quickly as n increases. It grows faster than log2(n), loge(loge(n)), (log2(n))^2, n and n^3 but slower than nlog2(n), 2^n and e^n.

10. e^n: exponential growth, increases extremely quickly as n increases. It grows faster than all other functions.

Conclusion:

In this experiment I was able to understand basics of space and time complexity and understand how various functions grow .