**Software Engineering Class: Counting Factors, Prime Numbers, and Iterations**

**Agenda for Today**

* **Count the Factors**
* **Optimisation for Counting Factors**
* **Check if a Number is Prime**
* **Sum of N Natural Numbers**
* **Understanding Iterations**
* **Comparing Two Algorithms**

**Counting Factors**

**Definition:**  
A number *i* is a factor of *N* if it divides *N* completely, resulting in a remainder of 0. This can be checked programmatically using the modulus operator %:  
*N%i==0N%i==0*

**Task:**  
Given a positive integer *N*, count the factors of *N*.

**Brute Force Solution:**  
The naive approach involves iterating from 1 to *N* and checking if each number is a factor.

function countFactors(N){

int count = 0;

for(int i = 1; i <= N; i++) {

if (N % i == 0) {

count++;

}

}

return count;

}

**Optimization for Counting Factors**

1. **Observation of Factor Pairs:**  
   If *i* is a factor of *N*, then *NiiN​* is also a factor. It suffices to run the iteration only up to *NN​*, leveraging the factor pairs.
2. **Revised Solution:**  
   This reduced the number of necessary iterations from *N* to *NN​*, drastically improving efficiency.

function countFactorsOptimized(N){

int count = 0;

for(int i = 1; i \* i <= N; i++) {

if (N % i == 0) {

if (i == N / i) {

count++;

} else {

count += 2;

}

}

}

return count;

}

The crucial point is to ensure that if *i* is equal to *N/iN/i*, it is counted only once【8:1†source】【8:14†source】.

**Checking If a Number is Prime**

* **Definition of a Prime Number:**  
  A number is prime if it has exactly two distinct positive divisors: 1 and itself.
* **Algorithm:**  
  Similar to the factor counting optimization, iterate only till *NN​* to verify if a number is prime.

function isPrime(N){

if (N <= 1) return false;

for(int i = 2; i \* i <= N; i++) {

if (N % i == 0) return false;

}

return true;

}

**Sum of N Natural Numbers**

**Formula:**  
The sum is calculated by the formula:  
*Sum=n×(n+1)2Sum=2n×(n+1)​*

This is a straightforward result from the properties of arithmetic sequences .

**Understanding Iterations**

* **Iteration Defined:**  
  The number of times a loop runs.
* **Example:**  
  For a loop running from 1 through *N*, the number of iterations is *N*.

**Comparing Two Algorithms**

* **Argument for Evaluating Algorithms:**  
  Comparing execution time can be misleading due to various factors like differing hardware or languages.
* **Iteration Count as a Measure:**  
  A stable measure of efficiency is the number of iterations, independent of hardware and software differences .

**Conclusion**

* Efficiency in solving algorithmic problems significantly improves through observation and optimization.
* The power of optimized algorithms cuts down execution time substantially, transforming infeasible computations into feasible ones .

This concludes our session on counting factors, prime checking, and the importance of efficient algorithm design through iteration evaluation. Future sessions will delve into advanced topics like time complexity and data structure analysis.

**Revision Notes: Asymptotic Analysis of Algorithms and Related Concepts**

**Introduction**

This class session focused on the basics of algorithm performance analysis, including asymptotic analysis, Big O notation, and basic algorithmic problems. The session was designed as an introduction to problem-solving in the context of Data Structures and Algorithms (DSA), particularly focusing on time complexity and iteration during the first classes.

**Asymptotic Analysis and Big O Notation**

**Asymptotic Analysis**

Asymptotic analysis refers to evaluating the performance of algorithms in terms of their time and space requirements when the input size becomes very large. This measure helps determine which algorithms are more efficient or scalable .

**Calculating Big O**

The primary steps in calculating the Big O notation are:

* **Calculate Iterations:** Determine how many iterations occur based on input size.
* **Ignore Lower Order Terms:** Focus only on the highest degree of the variable to simplify the calculation.
* **Ignore Constant Coefficients:** Remove constant multipliers as they do not affect the growth rate significantly .

**Example:**

* For an algorithm with iterations 100 \* log2(N), the Big O notation is O(log2(N)).
* If an algorithm involves N / 10 iterations, the Big O is O(N) .

**Comparing Orders**

The typical order of growth from lowest to highest is:

* log(N), sqrt(N), N, N log(N), N sqrt(N), N^2, N^3, 2^N, N!, N^N .

**Why Ignore Lower Order Terms and Constants?**

* **Lower Order Terms:** Their contribution becomes negligible as N grows; thus, they can be ignored for larger input sizes .
* **Constant Coefficients:** Do not change the growth rate of the function for large N .

**Examples and Problems**

**Counting Factors**

A factor of a number N is an integer that divides N with zero remainder. To count the factors, iterate from 1 to N, checking divisibility .

**Example:**

The factors of 24 are 1, 2, 3, 4, 6, 8, 12, and 24, making a total of 8 factors .

**Geometric Progression (G.P.)**

A geometric progression is a sequence where each term after the first is found by multiplying the previous one by a fixed non-zero number called the common ratio (R).

**Formula:**

* Sequence: a, a \* r, a \* r^2, ..., a \* r^(n-1)
* Sum of the first N terms: S = a \* (r^N - 1)/(r - 1), provided r ≠ 1 .

**Real-Life Examples of G.P.**

Geometric progressions can model scenarios like the spread of a virus, population growth, or financial interest calculations.

**Iteration and Time Complexity**

**Definition of Iteration**

Iteration refers to the number of times a part of a program executes, such as in loops. Understanding iterations is key to evaluating the time complexity of algorithms.

**Comparing Algorithms**

* For larger input sizes, Big O notation helps choose the more efficient algorithm by abstracting away lower order terms and constants.
* Limitations of Big O exist, such as not distinguishing between algorithms with the same highest-order term but different constant factors or lower-order terms .

**Conclusion**

This session established the foundational concepts of algorithm analysis, focusing on the principles of asymptotic behaviour and practical computation of Big O notation through iterative examples. Proper understanding of these concepts is essential for problem-solving in computer science, particularly when dealing with data structures and algorithms.

Factor:

A factor of N is any integer that divides N completely without leaving remainder

Check if number is prime:

A number is prime if it has only two factors: one and itself

Sum of N natural numbers:

The sum of N natural numbers is given by the formula N(N+1)/2

Brute Force counting factors:

Check all number up to N to see if they are factors of N using modulo operation

Optimized Factor counting:

Count factors of N by iterating up to , checking for factors pair

Iteration:

The number of times loop runs, crucial for understanding program complexity

Time complexity:

Measure of the number of iterations required to complete an algorithm

Big O notation:

Notation to describe the upper limit of the time complexity of an algorithm

Number of Iterations:

A key factor in comparing algorithms, independent of language or environment

Prime check Optimization:

Check for divisor of N only up to to determine if it is prime.

Geometric Progression (GP):

A sequence with a constant ratio between successive terms.

A sequence where each term after the first is found by multiplying the previous one by a fixed, non-zero number called the common ratio.

Sum of GP series:

The sum of a GP series uses the formula a(rn-1)/(r-1)

Space Complexity:

A measure of the amount of memory space an algorithm uses relative to its input size

n Log n:

A time complexity often associated with more efficient sorting algorithms.

Square root complexity:

Algorithms with a time complexity of , often involving checking up to

Possibilities

Summation Formula:

Formula for sum of first n natural numbers: n(n+1)/2

Prime number test:

Identifying a prime by confirming it has exactly two factors:1 and itself.

Range Inclusion:

In range [a,b], both a and b are included.

Code optimization:

The process of reducing the time complexity of an algorithm to make it run faster.

Executing Iterations:

Executing a set number of iteration based on loop conditions.

Intro problem solving

Time Complexity

Intro to Arrays

Prefix sum

Carry Forward

Subarrays

2D metrices

Sorting Basics

Hashing Basics

String Basics

Bit Manipulation Basics

Interview Problems

Contest [Covers Full Intermediate DSA]

Today's content

-Count the factors

-Optimization for counting the factors

-Check if a number is prime

-sum of N Natural Numbers

-Definition of AP and GP

-How to find the number of a times piece of code runs i.e. number of iterations

-How to compare two algorithms

what is factor?

i is factor of N ? that means N%i==0, i divide N completely

N factors: 1 is smallest factor

N is largest factor

function countFactor(N){

count =0;

for i-> 1to N;

if N%i==0

count++

return count;

}

above pseudo code require N iteration

Standard online system having 10^8 iterations in 1 sec.

N iterations Time

10^8 10^8 1sec

10^9 10^8 1sec \* 10=10sec

10^18 10^18 1sec \* 10^10= approx. 317years

Optimization:

i\*j=N {i and j are factors of N}

6\*3=18 {6 and 3 are factors of 18}

7\*4=28 {7 and 4 are factors of 28}

i\*j=N

j=N/i

This means i & N/i are factors of N

N=24

i N/i

1 24

2 12

3 8

4 6

------------------

6 4

8 3

12 2

24 1

i<=N/i

i^2<=N

i<=root(N)

N=24, root(24) .ie 4.xyz, iteration till 4

N=100, root(100) i.e 10 , iteration till 10

N=40

loop 1 to 6(root 40)

i count factors

1 +2 1 & 40

2 +2 2 & 20

4 +2 4 & 10

5 +2 5 & 8

optimize code:

function countFactor(N){

count=0;

for(i to root(N)){

if(N%i==0){

count+=2;

}

}

return count;

}

This above code having issue for perfect square number like 100.

N=100

i Count factors

1 +2 1 & 100

2 +2 2 & 50

4 +2 4 & 25

5 +2 5 & 20

10 +2 10 & 10 // here both 10 and 10 are same number , counting +2 not work

correct optimize code:

function countFactor(N){

count=0;

for(i to root(N) || 1 to i\*i<=N){ // either root(N) or i\*i<=N

if(N%i==0){

if(i==N/i){

count++;

}

else{

count+=2;

}

}

}

return count;

}

with optimize code:

N iteration time

10^8 10^4 0.5 1sec

10^9 10^4.5 5 sec

10^18 10^9 10 sec

instead of using inbuild lib for root N we can do i\*i<=N

-----------------------------------------

calculate sum 1+2+3+....+100

sum=1 + 2 + 3 + ... +100

sum=100 + 99 + 98 + ... +1

---------------------------------

2sum=101+101+101+...+101

2sum=101\*100

sum=(101\*100)/2

sum of N Numbers:

sum=1 + 2 + 3 + ... +N

sum=N + N-1 + N-2 + ... +1

---------------------------------

2sum=(N+1)+(N+1)+(N+1)+...+(N+1)

2sum=(N+1)\*N

sum=((N+1)\*N)/2

--------------------------------------------------------------------

some basic math properties

[a,c] a,b,c

(a,c) b

[a,c] This type of range means that a & c are both inclusive.

(a,c) This type of range means that a & c are both excluded.

[a,c] c-a+1

for(i to N){

}

[1 to N] : N-1+1=N

----------------------------------------------------------------------

Geometric Progression (GP): the ratio between any consecutive is same

5 10 20 40 80 -> series to this type is GP means ratio is exact same

5 10 20 40 80

10/5=2 20/10=2 40/20=2 80/40=2

here ratio(r)=2 // this 2 also called common ratio

Generic Notation:

a ar ar^2 ar^3 ....

r r r

Q. Given N terms of GP then what is sum?

sum=>a((r^n)-1)/r-1 where r!=1

a\*n where r=1: a a a a....n here ratio is 1

for 5 10 20 40 80:

a=5, r=2, n=5(total number)

a(r^n-1)/r-1 where r!=1 : 5((2^5)-1)/2-1 : 5\*(32-1)/1 : 5\*31 :155

Arithmetic Progression (AP): the difference between any consecutive is same

2 4 6 8 10 : difference is 2

sum=n/2(2a+(n-1)d) : d is difference, a is 1st term, n is total number of terms

----------------------------------------------------------

we cannot evaluate algorithms performance using execution time as it depends on a lot of

factors like OS, place of execution, programming language etc.

comparing two algorithms whose measure does not depend on any factor is

calculating number of iterations.

The number of iterations of an algorithm remains the same irrespective of OS, Place of

execution, programming language etc.

----------------------------------------------------------

Next class:

big O

logarithms

space Complexity

Time Limit exceed (TLE) error

importance of constraints

6 JUNE 2025:

What is factor of a number?

i is a factor of N if i divide N completely

N%i==0

24 🡪 1, 2, 3, 4, 6, 8, 12, 24 🡪 Total 8 factor of 24

10 🡪 1, 2, 5, 10 🡪 Total 4 factors of 10

Smallest factor of any number is 1.

Largest factor of any number is number itself.

So, all factors lie between 1 to N.

countFactor(N){

ans=0;

for(i=1 to N){

if(N%i==0){ // i is factor of N

ans++;

}

}

return ans;

}

Above is pseudo code, pseudo code looks like real code but it’s not.

Assume Any CPU/Machine execute approx. 108 iteration per second

Above code taken N iteration.

Total iteration to execute is 108 sec. then for 1 iteration it takes 1/108 sec.

So, N iteration in N/108 sec.

Say,

N=109 🡪N/108 sec. = 109/108 = 10 sec.

N=1018 🡪 N/108 sec. = 1018/108 = 1010 sec. ≈ 317 Years.

1018 in java stored in long and in python there is no restriction for value to store in variable.

OPTIMIZATION:

If I \* J = N 🡪 {I, J} are factors of N

J=N/I 🡪 {I, N/I} are factors of N

**If I is a factor of N then N/I is also factor of N**

N=24

I N/I

1 > 24/1 🡪 24

2 > 24/2 🡪 12

3 > 24/3 🡪 8

4 > 24/4 🡪 6

------------------------------- Below are the repetitions

6 < 24/6 🡪 4

8 < 24/8 🡪 3

12 < 24/12 🡪 2

24 < 24/24 🡪 1

N=100

I N/I

1 < 100/1 🡪 100

2 < 100/2 🡪 50

4 < 100/4 🡪 25

5 < 100/5 🡪20

10 < 100/10 🡪10

---------------------------------------- Below are the repetitions

20 > 100/20 🡪5

25 > 100/25 🡪4

50 > 100/50 🡪2

100 > 100/100 🡪1

* When I < N/I count +2 once it goes to I>N/I break loop
* When I==N/I then count +1

Condition:

I<=N/I 🡪 I\*I <=N 🡪 I2<=N 🡪 I<=

So total iteration required for above code is

OPTIMIZECODE:

countFactor(N){

count=0;

for(i= 1 to ){ // i\*i<=N

if(N%i ==0){

count = i==N%I? count+1 : count+2;

}

}

return count;

}

For above code

N=1018

It takes iteration to execute

Then, 🡪 🡪109 🡪 108 \*10 🡪 10 sec.

Prime Number:

* 1. Number divisible by 1 & number itself.
  2. Number has exactly 2 factors.

1st definition is wrong because 1 is exception in prime number.

Given N, check if N is prime or not:

isPrime(N){

if(countFactor(N)==2){

return true;

else

return false

}

This function also takes iterations

SOME BASIC MATH PROPERTIES: