Data Structures

Algorithms and Complexity Analysis

Chapter 2_2

The Big O

- Many different notations and formulations can be used in the complexity analysis. Among these, Big O (the "O" stands for "order of") is perhaps the simplest and most widely used

- It is used to describe the upper bound on growth rate of algorithms as a function of the problem size

Big O – Formal Definition

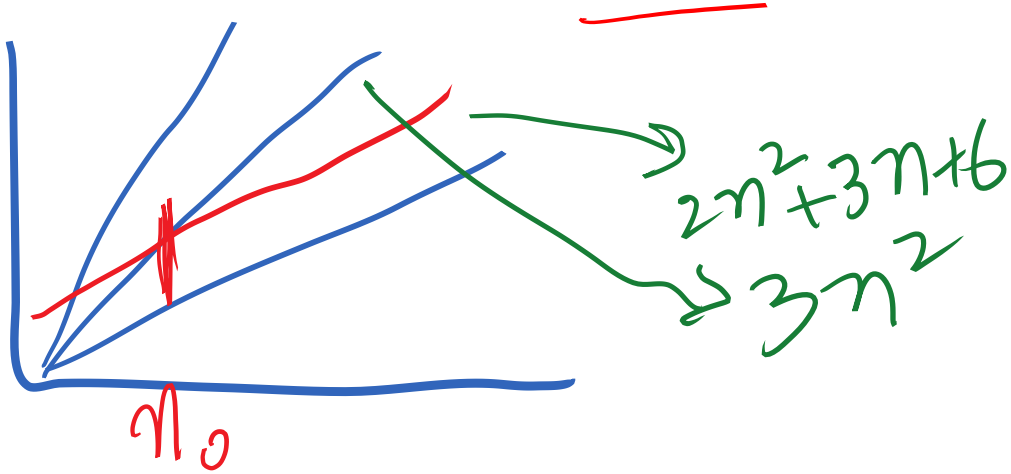
Formally, Big O is defined as:

$f(n) = O(g(n))$ iff there exists positive constant c and n_0 such that $f(n) \leq cg(n)$ for all values of $n \geq n_0$

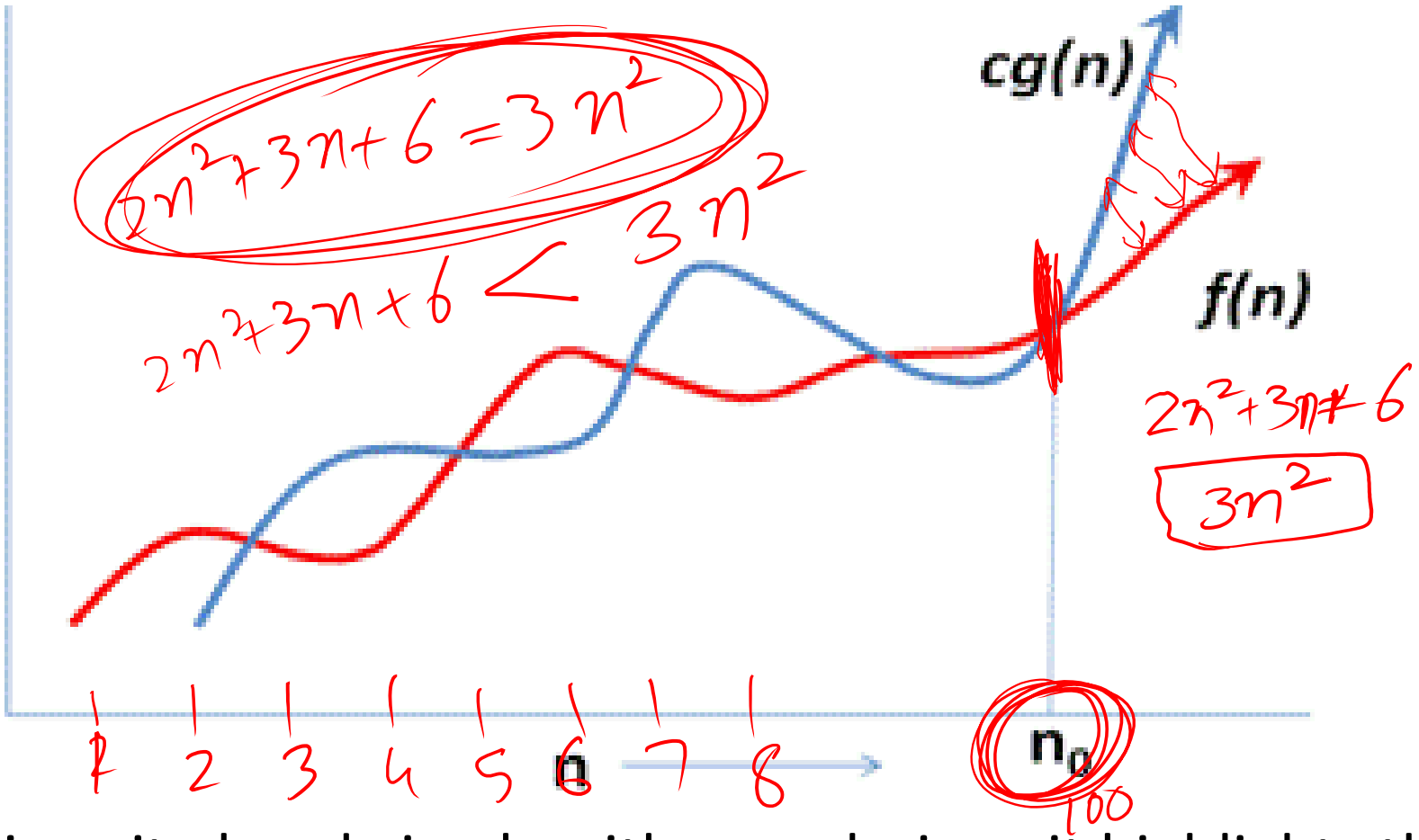
It is important to note that the equal sign in the expression $f(n) = O(g(n))$ does not denote mathematical equality. It is therefore incorrect to read that $f(n) = O(g(n))$ as $f(n)$ is equal to $O(g(n))$. Instead it is read as $f(n)$ is $O(g(n))$

$2n^2+3n+6$

$O(n^2)$
$O(n^3)$
⋮



The Big O – Graph Representation



Big O is quite handy in algorithm analysis as it highlights the growth rate by allowing the user to simplify the underlying function by filtering out unnecessary details.

The Big O - Example

For example, if $f(n) = 1000n$ and $g(n) = n^2$, then $f(n)$ is $O\big(g(n)\big)$ because for $n > 100$ and $C = 10$, $cg(n)$ is always greater than $f(n)$

$1000n = 100 \times 10n$

It can be easily seen that for a given function $f(n)$, its corresponding Big O is not unique – rather there are infinitely many functions $g(n)$ such that $f(n) = O\big(g(n)\big)$. For example, $N = O(N)$, $N = O(N^2)$, $N = O(N^3)$ and so on

$2N^2 + 3N + 500$ ✓ $O(N^4)$

Upper Bound

Tight Bound

Tight Bound

We would naturally like to have an estimate which is as close to the actual function as possible. For example, N= $O(N)$, N = $O(N^2)$, N= $O(N^3)$ are all correct upper bounds for N, the tightest among these is $O(N)$ and hence that should be chosen to describe the growth-rate of the function N

Loose Upper Bound

Significance of Higher Order Term

For large values of n, the growth rate of a function is dominated by the term with the largest exponent and the lower order terms become insignificant

See table in next slide for $f(n) = n^2 + 500n + 1000$, observe the ratio $\frac{n^2}{f(n)}$ approaches 1 as n becomes really large. That is, $f(n)$ gets closer to n^2 as n becomes larger and other terms become insignificant in the process.

$2n^4 + 100n^3 + 1000n$

Very large n

$f(n) = n^2 + 500n + 1000$

n	$f(n)$	n^2	$500n$	1000
1	1501	1	500	1000
10	6100	100	5000	1000
100	61000	10000	50000	1000
500	501000	250000	250000	1000
1000	1501000	1000000	500000	1000
5000	27501000	25000000	2500000	1000
10000	105001000	100000000	5000000	1000
100000	10050001000	10000000000	50000000	1000

$f(n) = n^2 + 500n + 1000$

n	$\frac{n^2}{f(n)}$	$\frac{500n}{f(n)}$	$\frac{1000}{f(n)}$
1	0.000666	0.333111	0.666223
10	0.016393	0.819672	0.163934
100	0.163934	0.819672	0.016393
500	0.499002	0.499002	0.001996
1000	0.666223	0.333111	0.000666
5000	0.909058	0.090906	3.64×10^{-5}
10000	0.952372	0.047619	9.52×10^{-6}
100000	0.995025	0.004975	9.95×10^{-8}

In fact, it can be easily shown that if $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$.

Higher Order Term Example

$$f(x) = 10x^3 - 12x^2 + 3x - 20$$

$$f(x) = O(x^3)$$

This is an extremely useful property of Big O as, for systems where derivation of the exact model is not possible, it allows us to concentrate on the most dominating term by filtering out the insignificant details (constants and lower order terms)

Big O Usefulness

Even with such simplification, Big O is quite useful. It tells us about the scalability of an algorithm

Programs that are expected to process large amounts of data are usually tested for small inputs

The complexity analysis helps us in predicting how well and efficiently our algorithm will handle large real input data.

Since Big O denotes an upper bound for the function, it is used to describe the worst case scenario.

Big O Important Points

If $f(n) = O(g(n))$ and $h(n) = O(g(n))$, then it does not follow that $f(n) = h(n)$

$$O(g(n)) + O(h(n)) = max\left(O(g(n)), O(h(n))\right)$$

$$O(g(n)).O(h(n)) = O(g(n)).O(h(n))$$

Common Growth-rate Functions in Computer Science

Problem size n	Growth Rate						
	Constant $O(1)$	Logarithmic $O(\log n)$	Linear $O(n)$	Log-Linear $O(n \log n)$	Quadratic $O(n^2)$	Cubic $O(n^3)$	Exponential $O(2^n)$
1	1	1	1	1	1	1	2
2	1	2	2	4	4	8	4
3	1	2	3	6	9	27	8
4	1	3	4	12	16	64	16
5	1	3	5	15	25	125	32
6	1	3	6	18	36	216	64
7	1	3	7	21	49	343	128
8	1	4	8	32	64	512	256
9	1	4	9	36	81	729	512
10	1	4	10	40	100	1000	1,024
20	1	5	20	100	400	8000	1.5×10^6
30	1	5	30	150	900	27000	1.07×10^9
100	1	7	100	700	10000	10^6	1.26×10^{30}
1,000	1	10	1,000	10000	10^6	10^9	1.07×10^{301}
1,000,000	1	20	10^6	20×10^6	10^{12}	10^{18}	too big a number

Determining Big O– A Mini Cookbook

Except for a few very tricky situations, Big O of an algorithm can be determined quite easily. We shall first review some basic mathematical formulae needed for the analysis and then discuss how to find the Big O for algorithms

Basic Mathematical Formulae

- Let us review some basic mathematical formulae required to find the Big O for algorithms

- Sum of terms of a sequence is of special interest. Among them most frequently used are:
  - Arithmetic Series
  - Geometric Series
  - Sum of Squares
  - Sum of Logs

Arithmetic Sequence

An arithmetic sequence is a sequence in which two consecutive terms differ by a constant value. For example, 1, 3, 5, 7, … is an arithmetic sequence. In general, the sequence is written as:

$$a_1, (a_1 + d), (a_1 + 2d), \ldots, (a_1 + (n-2)d), (a_1 + (n-1)d)$$

$$S_n = a_1 + (a_1 + d) + (a_1 + 2d) + \cdots + (a_1 + (n-2)d) + (a_1 + (n-1)d)$$

$$= \frac{n(a_1 + a_n)}{2} \quad \text{for } d = 1$$

$$S_n = \frac{n}{2}\{2a + (n-1)d\}$$

Geometric Series

In a geometric sequence, the two adjacent terms are separated by a fixed ratio called the common ratio. For example, 1, 3, 9, 27, 81, … is a geometric sequence where the common ratio is 3

$$S_n = ar^0 + ar^1 + ar^2 + \cdots + ar^{n-1}$$

$$= \frac{a(r^n - 1)}{r - 1}$$

Sum of Log of Arithmetic Sequence

$$\log a_1 + \log(a_1 + d) + \log(a_1 + 2d) + \ldots + \log(a_1 + (n-1)d)$$

$$\log 1 + \log 2 + \cdots + \log n = \log(1.2.3 \ldots n) = \log n! = n \log n$$

$$\log n! \approx n \log n$$

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Stirling's Formula

Floor and Ceiling Functions

Let us consider the following question: If 3 people can do a job in 5 hours, how many people would be required to do the job in 4 hours?

3.75 people

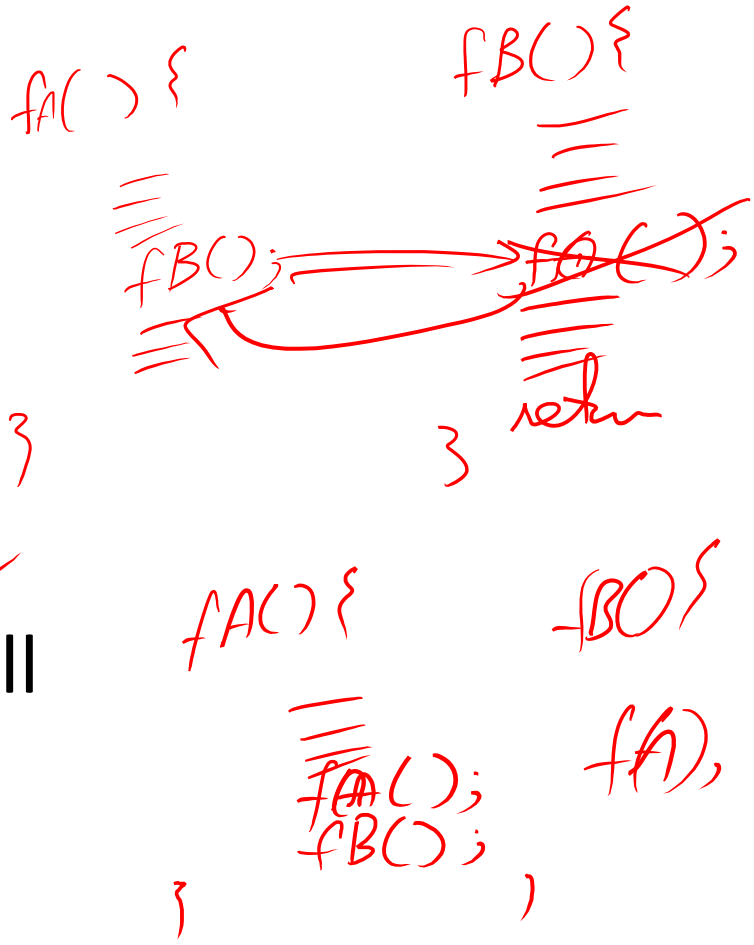
1. floor(x) (mathematically written as $\lfloor x \rfloor$ is the largest integer not greater than x

2. ceiling(x) (written as $\lceil x \rceil$) is the smallest integer not less than x.

Calculating Big O

A structured algorithm may have following different types of statements:

- Sequence of statements
- Block statement
- Selection
- Loop
- Non-recursive subprogram call
- Recursive subprogram call



Simple Statement and O(1)

- For the purpose of this discussion, we define a simple statement as the one that does not have any control flow component (subprogram call, loop, selection, etc.) in it
- An assignment is a typical example of a simple statement
- Time taken by a simple statement is considered to be constant
- constant amount of time is denoted by $O(1)$

{addition
multiplication
division
assignment

Not same time operation
O(1)

Sequence of Statements

```
statement1    // O(n²)
statement2    // O(1)
statement3    // O(n)
statement4    // O(n)
```

$$O(n^2) + O(1) + O(n) + O(n)$$

$$= \max\big(O(n^2), O(1), O(n), O(n)\big) = O(n^2)$$

Selection

```
if (cond1)          statement1      // O(n²)
else if (cond2)     statement2      // O(1)
else if (cond3)     statement3      // O(n)
else                statement4      // O(n)
```

$$\max\left(O(n^2), O(1), O(n), O(n)\right) = O(n^2)$$

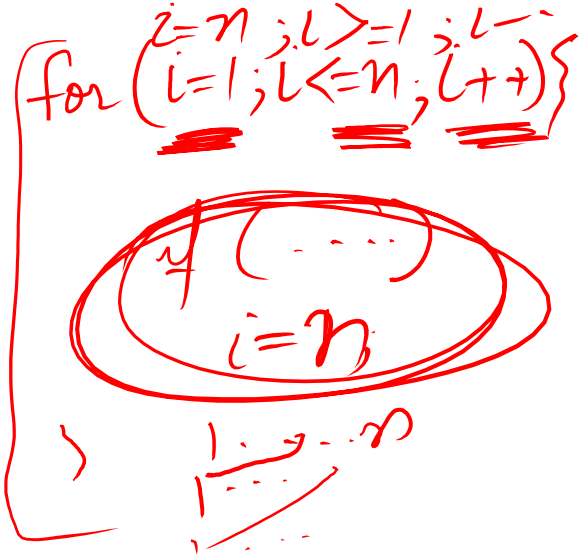
Simple Loops

- We define a simple loop as a loop which does not contain another loop inside its body (that is no nested loops) and the loop control variable is not set backwards
- Determine the complexity of the code written in the loop body as a function of the problem size

  Let it be $O\ (f(n))$
- Determine the number of iterations of the loop as a function of the problem size

  Let it be $O\ (g(n))$
- Then the complexity of the loop would be:

  $O\ (f(n))\ .O(g(n)) = O(f(\ n)\ .g\ (n)\ )$



Simple Loop - Uniform - Variable

- Simple loops come in many different varieties.
- The most common ones are:
  - the counting loops with uniform step size (Loop controlling variable) is incremented or decremented uniformly) and
  - loops where the step size (Loop controlling variable) is multiplied or divided by a fixed number

for (i=1; i<=n; i=i+3 4 ⋮)

i++
i+=k

Loops with Uniform Step Size

```
index = -1;
for (i = 0; i < N; i++)
        if (a[i] == key)
                return true;
return false;
```

Simple Loops with Variable Step Size

```
High = N – 1
Low = 0;
Index = -1
While (low <= high){
      mid = (high + low ) / 2;
      if (key == a[mid])       return true;
      else if (key > a[mid]) low = mid + 1;
      else                        high = mid – 1;
}
```

A Deceptive Case

```
for(int i=1; i<=10; i++)
    cout << a[i] << endl;
```

constant

i <= 1000

for (i = 1 ; i <= 100 000 ; i++)

constant time

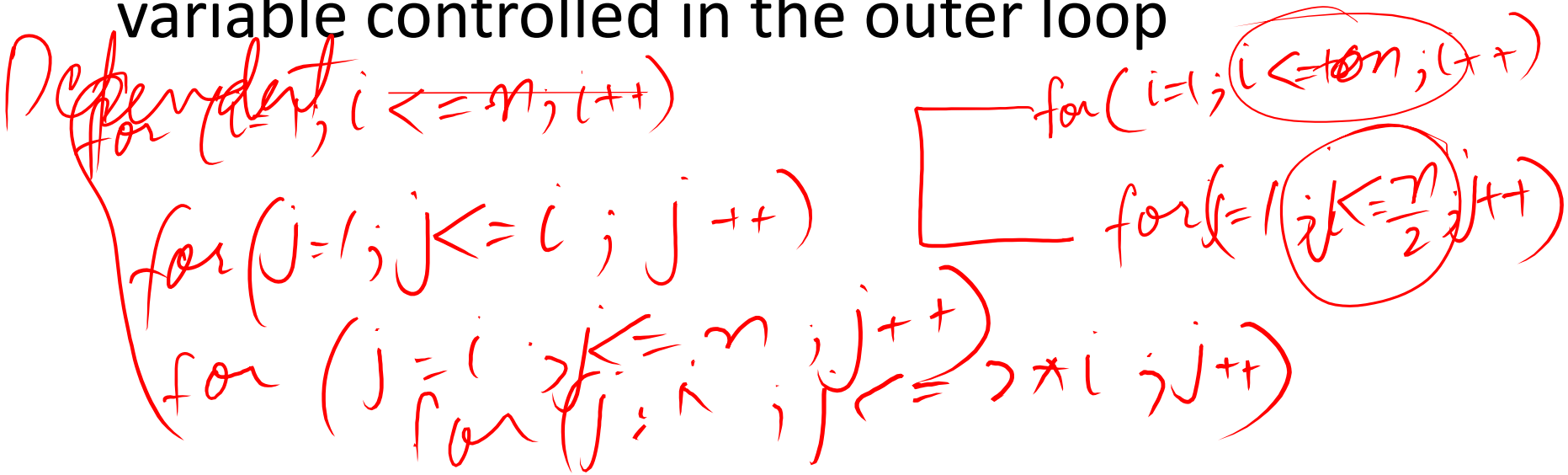
O(1)

Time Complexity

Nested Loop

- Independent loops – the number of iterations of the inner loop are independent of any variable controlled in the outer loop

- Dependent loops – the number of iterations of the inner loop are dependent upon some variable controlled in the outer loop



Independent Loops

```
for (i = 0; i < ROWS ; i++)
     for (j = 0; j < COLS; j++)
          c[i][j] = a[i][j] + b[i][j];
```

Dependent Loops

```
for (i = 0; i < N ; i++) {
      min = i;

      for (j = i; j < N; j++)
            if (a[min] > a[j]) min = j;

      t = a[i];
      a[i] = a[min];
      a[min] = t;
}
```

Dependent Loops (Cont…)

```
k = 0;
for (i = 1; i < N; i++)
     for (j = 1; j <= i; j = j * 2)
          k++;
```

```
k = 0;
for (i = 1; i < N; i = i * 2)
     for (j = 0; j < i; j++)
          k++;
```

Loop with Step Size Decrementing

```
j = 0;
len1 = strlen(a);
len2 = strlen(b);
for (i = 0; i < len1; i++)
   if (a[i] == b[j]) {
      j++;
      if (j == len2) break;
   }
   else {
      i = i - j;
      j = 0;
   }
```

Non-recursive subprogram call

- A non-recursive subprogram call is simply a statement whose complexity is determined by the complexity of the subprogram

- Therefore, we simply determine the complexity of the subprogram separately and then use that value in our complexity derivations

Sub Program to Find Sum

```
float sum(float a[], int size) {
 // assuming size > 0

   float temp = 0;
   for (int i = 0; i < size; i++)
       temp = temp + a[i];
   return temp;
 }
```

1 + 3N + 1 ≤ 4N

O(n)

Approximation

3N + 4 < 1000 n

O(n²)

for (i = 1; i <= j; i++)

j = rand(); // Any Value

Program Calling Sub Program

```
float average(float data[], int size){
// assuming size > 0
   float total, mean, ;
   total = sum(data, size); //O(N)
   mean = total/size;       //O(1)
   return mean;             //O(1)
}
```

O(1) ✓

✓ ✓ ✓

O(n²) O(1) + O(N) + O(1) =

max(O(1), O(N), O(1))

O(N)

Another Example Sub Program

```
int foo(int n){
   int count = 0;
   for (int j = 0; j < n; j++)
        count++;
   return count;
}
int bar(int n){
   temp = 0;
   for (i = 1; i < n; i = i * 2){
        temp1 = foo(i);
        temp = temp1 + temp;
   }
}
```

O(n)

1
2
4
8
16
32
⋮

O(logn)

O(n)

O(logN) · O(N) = O(N logN)

Find Complexity

تم سلامت رہو ہزار برس

O(1)

ہر برس کے ہوں دن پچاس ہزار

ہزاروں خواہشیں ایسی کہ ہر خواہش پہ دم نکلے

بہت نکلے میرے ارماں لیکن پھر بھی کم نکلے