Algorithm Analysis

- Sonali Gogate

Algorithm Analysis (1/2)

- Checking after writing the program?
  - If algorithms are to be compared?
  - Can we test any program for all the possible values?
- Analyze such that
  - All the possible inputs are taken into account
  - Allows us to evaluate efficiency independent of hardware and software environments
  - Can be performed without implementing

Algorithm Analysis (2/2)

Analysis framework

- A way to describe the algorithm (consistent/ standard)
- Computational model (in which we can theoretically execute the algorithms)
- A metric to measure the running time of the algorithm
- Approach to characterizing running times

Space & Time complexity

- The space complexity of an algorithm is the amount of memory it needs to run to completion.
- The time complexity of an algorithm is the amount of computer time it needs to run to completion.

Space Complexity

Space Complexity (1)

The space complexity of an algorithm is the amount of memory it needs to run to completion.

The space needed by each algorithm is the sum of following components –

Fixed Part consisting of -

1. Instruction space
2. Space for simple variables & fixed sized component variables
3. Space for constants

Variable Part consisting of -

1. Component variables whose sizes depend on problem instance
2. Space needed by reference variables
3. Recursion stack space

Space Complexity (2)

The space requirement S(P) of any algorithm P may therefore be written as –

$$S(P) = c + S_P$$

Where c is a constant.

While Analyzing the space complexity we focus on S_P.

Space Complexity (3)

```
EquationABC(a, b, c)
{
        Result = a + b + b*c + (a+b-c)/(a+b) + 4;
        Return Result;
}
```

What is the space complexity for this?

$S(P) = c + S_P$

Space Complexity (4)

```
SumI(A, n)
{
        S = 0;
        for (i=1 to n)
            S = S + A[i];
        Return S;
}
```

What is the space complexity for this?

$S(P) = c + S_P$

Space for n

Space for S

Space for i

Space for A

So at least n+3

$S_{SumI(n)} >= (n+3)$

Space Complexity (5)

```
SumR(A, n)
{
    if (n = 0)
        Return 0;
    else
        Return SumR(A, n-1) + A[n];
}
```

What is the space complexity for this?

$S(P) = c + S_P$

Stack space

- Formal parameters
- Local variables
- Return address

Each call to SumR

- Value of n
- Return address
- Pointer to A[ ]

Depth of recursion is (n+1)

$S_{SumR(n)} >= 3(n+1)$

Time Complexity

Time Complexity of an Algorithm

The time complexity of an algorithm is the amount of computer time it needs to run to completion.

For Time Complexity analysis, we use an analysis framework that consists of -

1. A language to describe the algorithm
2. Computational Model - in which we can theoretically execute the algorithms
3. A metric for measuring the algorithms' running times
4. An approach for characterizing the running times

Time Complexity Basics (1)

The Time T(P) taken by a program is sum of compile time and run (execution) time.

Compile Time

- Does not depend on instance characteristics
- Comes into picture very few times

Run Time

- Denoted as $\mathbf{t_p}$

- In the analysis phase, we estimate the $\mathbf{t_p}$
- Best case, Worst case, Average case

Time Complexity Basics (2)

- Program step – A program step is (loosely) defined as a syntactically or semantically meaningful statement of a program that has execution time which is independent of the instance characteristics.

- For the purpose of the analysis, we assume that each program step takes the same amount of fixed time.

- Determining the number of steps needed to be executed for program completion

Time Complexity Basics (3)

- Time for
  - comments
  - Assignment
  - Function call
  - Control statements

- Counting the steps –
  - Using a counter to increment on every step
  - Using a table to determine the number of times each step is getting executed in the program

Using Counter for Counting steps

```
Algorithm RSum(A, n)
{
        Count = Count + 1;
        If (n = 0)
        {
                Count = Count + 1;
                Return 0;
        }
        Else
        {
                Count = Count + 1;
                Return RSum(A, n-1) + A[n];
        }
}
```

Time Complexity of RSum

$$t_{RSum}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{RSum}(n-1) & \text{if } n > 0 \end{cases}$$

Recurrence Relation

```
Algorithm ISum(A , n)
{
        S = 0;
        Count = Count + 1;
        For (I = 1 to n)
        {
                Count = Count + 1; // For the for
                S = S + A[i];
                Count = Count + 1; // For  the assignment
        }
        Count = Count + 1;          //When i becomes n+1 and check happens
                                    // to decide not to enter the for loop

        Count = Count + 1;          //For return
        Return S;
}
```

Using a table to determine the steps

Statement	s/ e	Frequency	Total steps
Algorithm ISum(A , n)	0	-	0
{	0	-	0
S = 0;	1	1	1
For (I = 1 to n)	1	n + 1	n + 1
S = S + A[i];	1	n	n
Return S;	1	1	1
}	0	-	0
Total			**2n + 3**

Matrix Addition

```
Algorithm MatAdd(A, B, C, m, n)
{
    for i := 1 to m
        for j := 1 to n
            C[i, j] := A[i, j] + B[i, j];
}
```

Statement	s/ e	Frequency	Total steps
Algorithm MatAdd(A, B, C, m, n)	0	-	0
{	0	-	0
for i := 1 to m	1	m + 1	m + 1
for j := 1 to n	1	m (n + 1)	mn + m
C[i, j] := A[i, j] + B[i, j];	1	mn	mn
}	0	-	0
Total			**2mn + 2m + 1**

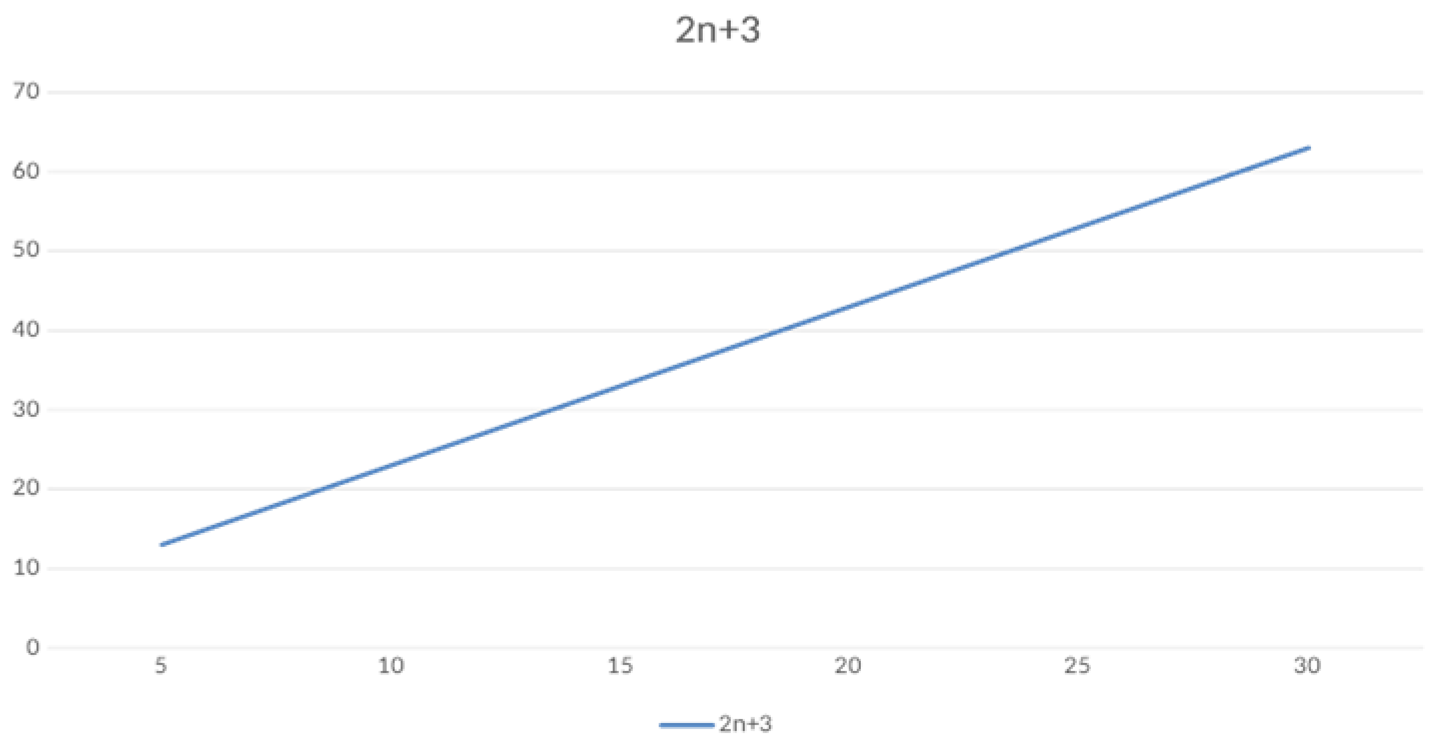
Analysis of the time complexity

We need to determine how long the algorithm will take based on the size of the input.

We also need to understand how fast the time complexity function of an algorithm grows with the size of the input. **This is the rate of growth of the running time of the algorithm.**

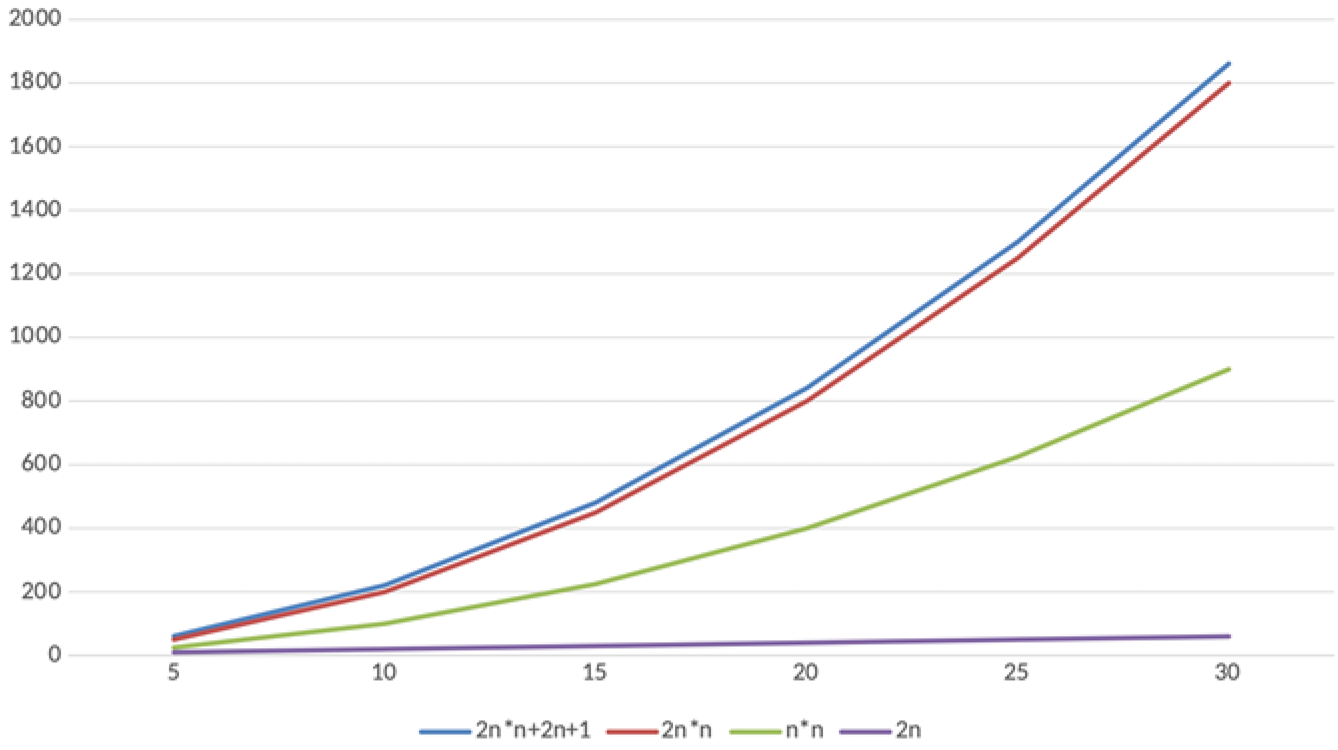
Statement	s/ e	Frequency	Total steps
Algorithm ISum(A , n)	0	-	0
{	0	-	0
S = 0;	1	1	1
For (I = 1 to n)	1	n + 1	n + 1
S = S + A[i];	1	n	n
Return S;	1	1	1
}	0	-	0
Total			**2n + 3**

Time Complexity of Sum(A, n)



Statement	s/ e	Frequency	Total steps
Algorithm MatAdd(A, B, C, m, n)	0	-	0
{	0	-	0
for i := 1 to m	1	m + 1	m + 1
for j := 1 to n	1	m (n + 1)	mn + m
C[i, j] := A[i, j] + B[i, j];	1	mn	mn
}	0	-	0
Total			**2mn + 2m + 1**

Time Complexity of Matrix Sum (m=n)



Asymptotic Analysis & Notations

- Asymptotic Analysis of algorithms is analysis of the run time based on the size of the input and the growth of the runtime.

- Asymptotic notations are the notations used in Asymptotic analysis

Asymptotic Notations (1)

Big O notation

If $f(n)$ and $g(n)$ are functions of n; c is a real constant > 0 and n_0 is an integer constant $>= 1$, then we say

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

If For $\forall\ n > n_0$, $0 <= f(n) <= c.g(n)$

Asymptotic Notations (2)

Big Ω notation

If f(n) and g(n) are functions of n; c is a real constant > 0 and n_0 is an integer constant >= 1, then we say

$$f(n) = \Omega(g(n))$$

If For $\forall$ $n > n_0$, $0 <= c.g(n) <= f(n)$

Asymptotic Notations (3)

Big Θ notation

If f(n) and g(n) are functions of n; c1 and c2 are real constants > 0 and n_0 is an integer constant >= 1, then we say

$$f(n) = \Theta(g(n))$$

If For $\forall\ n > n_0$, $0 <= c1.g(n) <= f(n) <= c2.g(n)$

Big O examples

- $2n + 3$ (ISum)

  $O(n)$

  as $2n + 3 <= 4n$ for all $n >= 2$

- $2n^2 + 2n + 1$ (Matrix addition)

  $O(n^2)$

  as $2n^2 + 2n + 1 <= 4n^2$ for all $n >= 2$

Search Algorithms

Linear Search Algorithm

Given an array **A[ ]** of **n** elements, search for the given element **x** in **A[ ]**

Algorithm

```
Search(A[ ], n, x)
{
    for (i=1 to n)
        if (A[i] == x)
            return i;
    return -1;
}
```

Time Complexity

O(n)

Linear

Binary Search Algorithm (1/2)

Given a sorted array **A[ ]** of **n** elements, search for the given element **x** in **A[ ]**

```
binarySearchR(A[ ], l, r, x)
{
    if (r >= l)
    {
        mid = l + (r - l) / 2;
        if (A[mid] == x)
           return mid;
        if (A[mid] > x)
            return binarySearch(A[ ], l, mid - 1, x);
       else
           return binarySearch(A[ ], mid + 1, r, x);
    }
    return -1;
}
```

Binary Search Algorithm (2/2)

Given a sorted array **A[ ]** of **n** elements, search for the given element **x** in **A[ ]**

```
binarySearchI(A[], l, r, x)
{
    while(l <= r)
    {
        mid = l + (r - l) / 2;
        if (A[mid] == x)
           return mid;
        if (A[mid] > x)
            r = mid - 1;
       else
           l = mid + 1;
    }
    return -1;
}
```

Time Complexity of Binary Search (1)

Example:

Sorted Array of 12 elements - 2,4, 7, 11, 16, 21, 25, 30, 34, 38, 43, 49

Element to search for - 38

Iteration	array	Mid Element	found	Compare	Sub Array for next iteration
1	2,4, 7, 11, 16, 21, 25, 30, 34, 38, 43, 49	21	N	21 < 38	25 to 49
2	25, 30, 34, 38, 43, 49	34	N	34 < 38	38 to 49
3	38, 43, 49	43	N	43 > 38	38
4	38	38	Y		

Time Complexity of Binary Search (2)

Array size goes from n to n/2 to n/4 ($n/2^2$) to n/8 ($n/2^3$)and so on.

It is **halved** in each iteration.

Let us say the array becomes of size 1 after it is halved k times.

Thus,

$$n/2^k = 1$$

That is,

$$n = 2^k$$

Taking log

$$\log_2 n = \log_2 2^k$$

$$\Rightarrow \quad \log_2 n = k\log_2 2$$

$$\Rightarrow \quad k = \log_2 n$$

Recurrence equation for Binary search

$$T(n) = \begin{cases} 1 & \text{for } n = 1 \\ T(n/2) + 1 & \text{for } n > 1 \end{cases}$$

Other names for binary search

Half Interval Search

Logarithmic Search

Common Orders of Execution

Common Orders of Execution

O(1)	Constant time

Common Orders of Execution

O(1)	Constant time
O(log n)	Logarithmic time

Common Orders of Execution

O(1)	Constant time
O(log n)	Logarithmic time
O(n)	Linear time

Common Orders of Execution

O(1)	Constant time
O(log n)	Logarithmic time
O(n)	Linear time
O(n^2)	Quadratic time

Common Orders of Execution

O(1)	Constant time
O(log n)	Logarithmic time
O(n)	Linear time
O(n^2)	Quadratic time
O(n logn)	

Common Orders of Execution

$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n^2)$	Quadratic time
$O(n \log n)$	
$O(n^3)$	Cubic time

Common Orders of Execution

$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n^2)$	Quadratic time
$O(n \log n)$	
$O(n^3)$	Cubic time
$O(n^c)$	Polynomial time

Common Orders of Execution

O(1)	Constant time
O(log n)	Logarithmic time
O(n)	Linear time
$O(n^2)$	Quadratic time
O(n logn)	
$O(n^3)$	Cubic time
$O(n^c)$	Polynomial time
$O(2^n)$	Exponential time

Rules for functions & orders of growth

1. $cn^m = O(n^k)$ for any constant c and any $m \leq k$

2. $O(f(n)) + O(g(n)) = O(f(n) + g(n))$

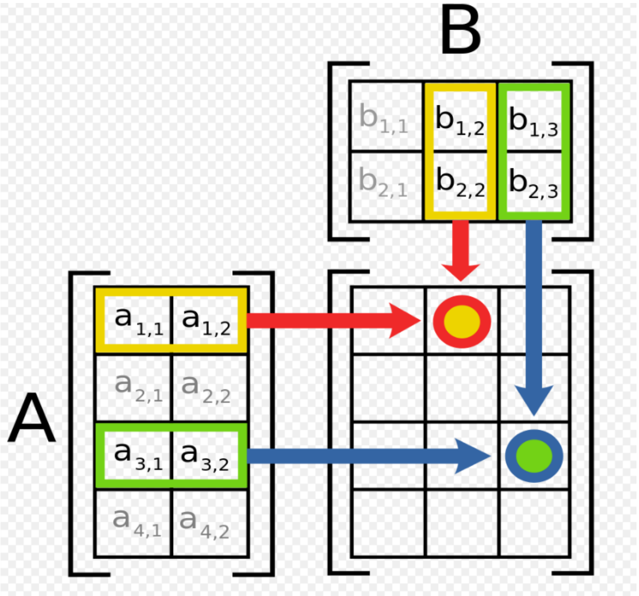
3. $O(f(n))O(g(n)) = O(f(n)g(n))$

4. $O(cf(n)) = O(f(n))$ for any constant c

5. c is $O(1)$ for any constant c

6. $\log_b n = O(\log n)$ *for any base b*

Matrix Multiplication



a11b11+ a12b21	a11b12+ a12b22	a11b13 +a12b23
a21b11+ a22b21	a21b12+ a22b22	a21b13 +a22b23
a31b11+ a32b21	a31b12+ a32b22	a31b13 +a32b23
a41b11+ a42b21	a41b12+ a42b22	a41b13 +a42b23

$$C_{i,j} = \sum_{i=1}^{k} A_{i,k} \cdot B_{k,j}$$

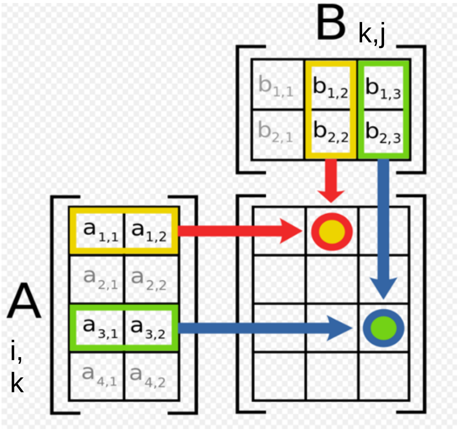
Matrix Multiplication

```
MatrixMult(A[M][N], B[P][Q])
{
    if (N != P)
        Exit;

    int i, j, k;

    for (i = 1 to M)
        for (j= 1 to Q)
        {
            C[i][j] = 0;
            for (k = 1 to N)
                C[i][j] = C[i][j] + A[i][k]*B[k][j];
        }
}
```

$$C_{i,j} = \sum_{i=1}^{k} A_{i,k} \cdot B_{k,j}$$



Matrix Multiplication Rules

A(m x n), B(p x q), C(r x s)

When can we multiply A & B?

- A x B (possible when n = p)

Is A x B = B x A?

- AxB != BxA
- B x A (possible when q = m)

Is A x (B x C) = (A x B) x C?

Matrix Chain Multiplication (1)

$$A = \begin{vmatrix} A11 & A12 \\ A21 & A22 \\ A31 & A32 \end{vmatrix} \qquad B = \begin{vmatrix} B11 & B12 & B13 & B14 \\ B21 & B22 & B23 & B34 \end{vmatrix} \qquad C = \begin{vmatrix} C11 \\ C21 \\ C31 \\ C41 \end{vmatrix}$$

M = A x B

Y = M x C

N = B x C

Z = A x N

Matrix Chain Multiplication (2)

$$A = \begin{vmatrix} A11 & A12 \\ A21 & A22 \\ A31 & A32 \end{vmatrix} \qquad B = \begin{vmatrix} B11 & B12 & B13 & B14 \\ B21 & B22 & B23 & B34 \end{vmatrix} \qquad C = \begin{vmatrix} C11 \\ C21 \\ C31 \\ C41 \end{vmatrix}$$

Option 1

M = A x B (3 x 4)

24 multiplications
12 additions

<u>Y = M x C</u> (3 x 1)

12 multiplication
9 additions

Option 2

N = B x C (2 x 1)

8 multiplication
6 additions

<u>Z = A x N</u> (3 x 1)

6 multiplication
3 additions