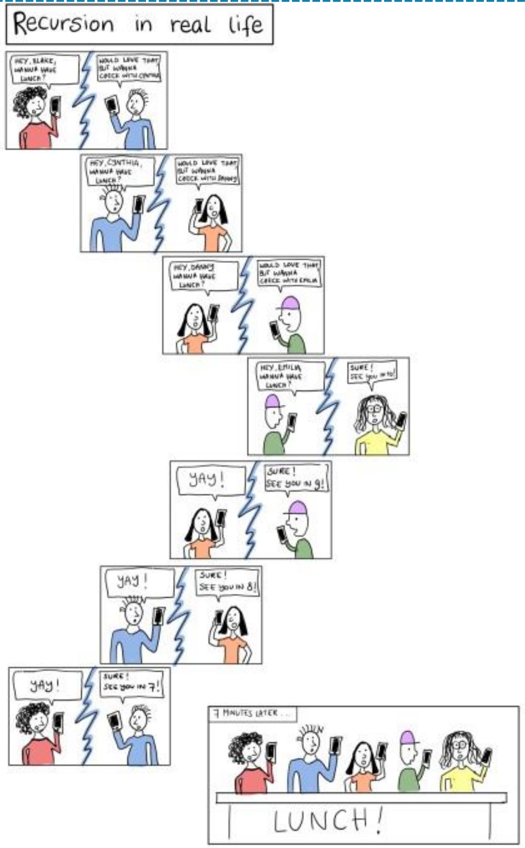
CS F211: DATA STRUCTURES & ALGORITHMS (2ND SEMESTER 2024-25) RECURSION & ALGORITHM COMPLEXITY

Chittaranjan Hota, PhD
Sr. Professor of Computer Sc.
BITS-Pilani Hyderabad Campus
hota[AT]hyderabad.bits-pilani.ac.in

WHAT IS RECURSION?

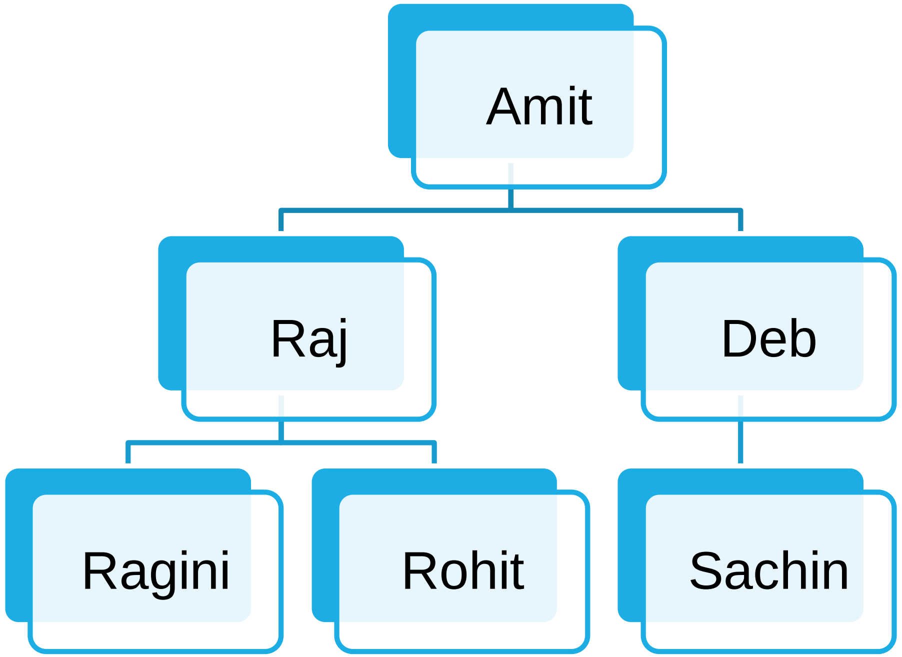


https://abetterscientist.wordpress.com/

```
root
├── dir1
│   └── file1.txt
└── dir2
    └── dir3
        └── targetFile.txt
```

```
function search(currentDir):
  if targetFile exists in currentDir:
    return currentDir;
  for each childDir in currentDir:
    result = search(childDir)
    if result is not null:
      return result;
  return null;
```

(How many people work under Amit?)



(Business organization chart)

Many more: Factorial, Fibonacci seq., Towers of Hanoi, Merge sort, Quick sort, Binary search …

LINEAR RECURSION

- A linear recursive function is a function that makes at most one recursive call each time it is invoked (as opposed to one that would call itself multiple times during its execution).

```
int gcd(int a, int b) {
  if (b == 0) {
    return a;
  } else {
    return gcd(b, a % b);
  }
}
```

Euclidean Algorithm (Recursive)

```
int sumArrayRecursive(int arr[], int n) {
// What is the base case?

//Recursive step:
    return arr[0] + sumArrayRecursive(???,    ???);
  }
}
int main() {
  int arr[] = {1, 2, 3, 4, 5};
  int n = sizeof(arr) / sizeof(arr[0]);
  int sum = sumArrayRecursive(arr, n);
  cout << "Sum of array elements: " << sum << endl;
  return 0;
}
```

TAIL RECURSION: REVERSING AN ARRAY

```
void reverseArray(int arr[], int start, int end)
{
  if (start >= end) { //reached ???
    return;
  }
  swap(arr[start], arr[end]);
  reverseArray(arr, start + 1, end - 1);
}
```

```
void reverseArray(int arr[], int size) {
   int start = 0;
   int end = size - 1;
   while (start < end) {
      swap(arr[start], arr[end]);
      start++;
      end--;
   }
}
```

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
- Such methods can be easily converted to non-recursive methods (which saves on some resources).

WHAT ABOUT FACTORIAL?

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return                    ;
    }
}
```

Is it tail recursive?

```
int tail_factorial(int n, int acc) {
    if (n == 0) {
        return acc;
    } else {
        return tail_factorial(n - 1, n * acc);
    }
}
int factorial(int n) {
    return tail_factorial(n, 1);
}
```

What about this?

```
int factorial_iterative(int n) {int prod = 1; for (int i = 1; i <= n; ++i) { prod *= i;} return prod;}
```

BINARY RECURSION

- What is binary recursion?

Algorithm BinarySum(A, i, n):
Input: An array A and integers i and n
Output: The sum of the n integers in A starting at index i
if $n == 1$ **then**
  return $A[i]$;
return
  BinarySum(A, i, $n/2$) + BinarySum(A, $i + n/2$, $n/2$)

```
void towerOfHanoi(int n, char source, char
dest, char aux) {
    if (n == 1) {
        cout << "Move disk 1 from " <<
          source << " to " << dest << endl;
        return;
    }
    towerOfHanoi(n - 1, source, aux, dest);
    cout << "Move disk " << n << " from "
     << source << " to " << dest << endl;
    towerOfHanoi(n - 1, aux, dest, source);
}
```

Let us see the recursion trace…

Used heavily in merging and tree traversals…

COMPUTING FIBONACCI NUMBERS: BETTER WAY…

```cpp
#include<bits/stdc++.h>
using namespace std;

int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}

int main ()
{
    int n = 9;
    cout << fib(n);
    getchar();
    return 0;
}
```

Is binary recursion better here?

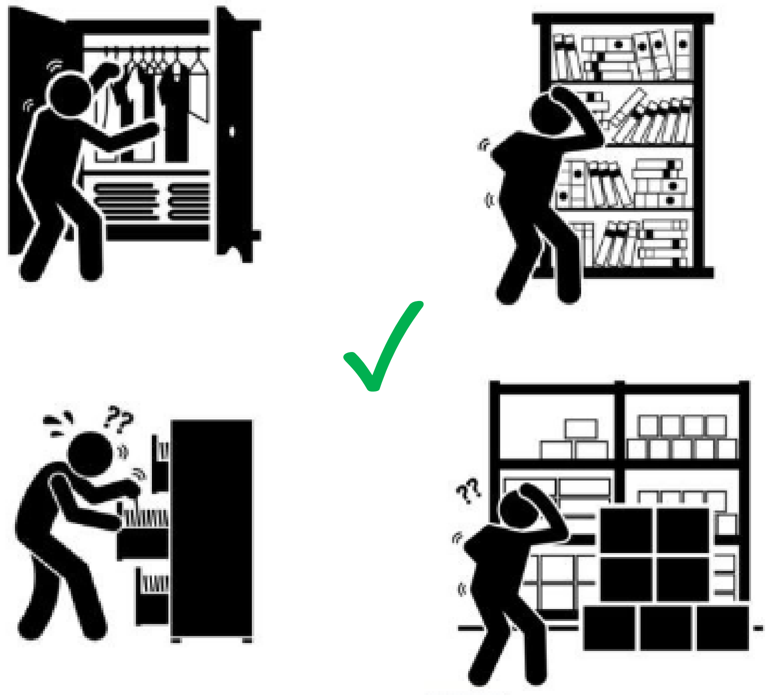
```cpp
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    int first = 0;
    int second = 1;
    int result;
    for (int i = 2; i <= n; i++) {
        result = first + second;
        first = second;
        second = result;
    }
    return result;
}
```

```cpp
#include <iostream>
using namespace std;



int fib(int n, int a = 0, int b = 1)
{
    if (n == 0)
        return a;
    if (n == 1)
        return b;
    return fib(n - 1, b, a + b);
}

// Driver Code
int main()
{
    int n = 9;
    cout << "fib(" << n << ") = "
         << fib(n) << endl;
    return 0;
}
```

What is the type of this rec.?

WHAT IS ALGORITHM COMPLEXITY?

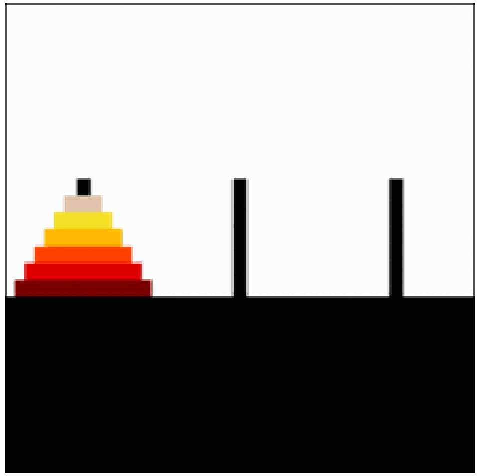
ALGORITHM 1



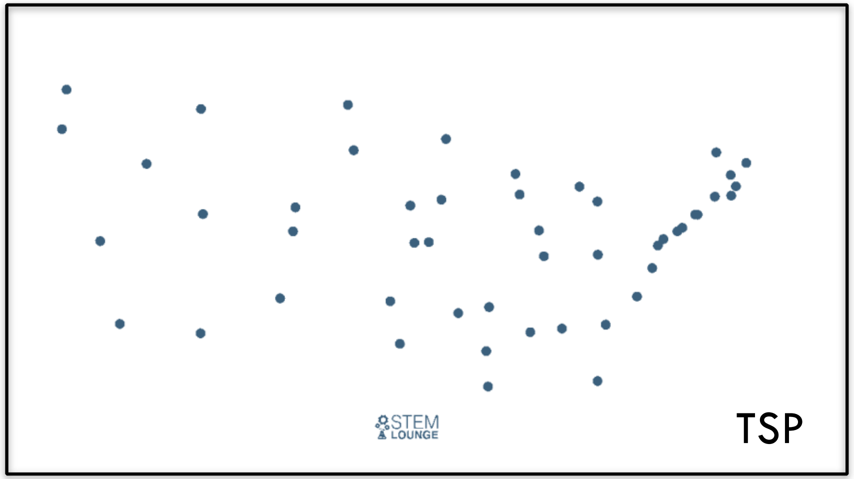
ALGORITHM 2

(A metaphor: searching car keys in your home)

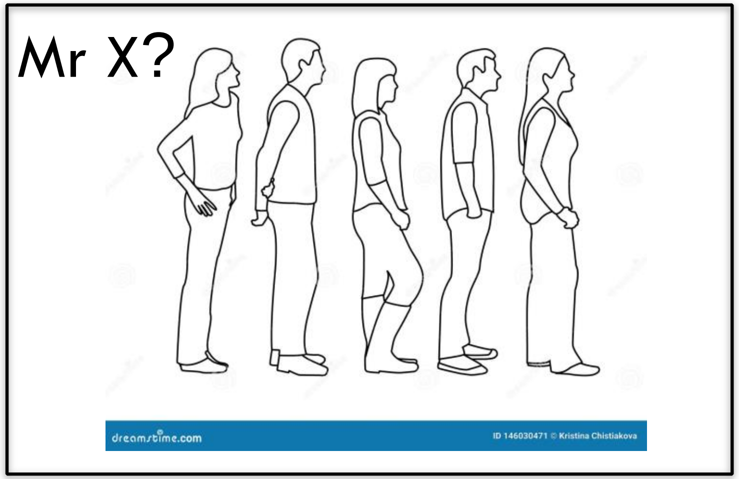
FEW MORE COMPLEXITY EXAMPLES…



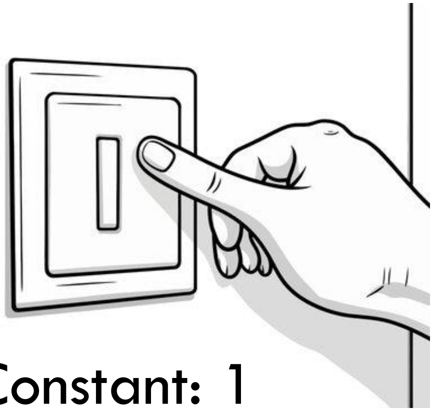
Exponential: 2^n



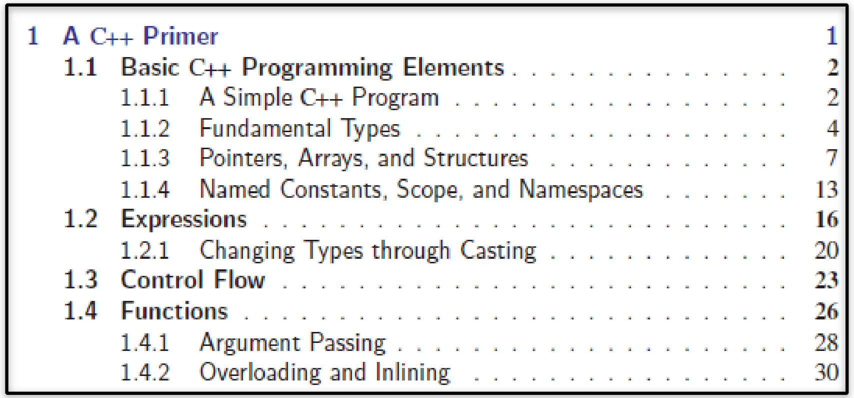
Quadratic (greedy heuristics): n^2



Linear: n



Constant: 1



Logarithmic: log(n)

Log-linear: n.log(n)

TASK FOR YOU…

You want to look for a word in a dictionary that has every word sorted alphabetically. How many algorithms are there and which one would you prefer?

WHY IS IT SO IMPORTANT?

Metaphor: Daily Budget and Spending

Problem Statement:

Problem Name: "Maximum Subarray Sum"

Input: An array of integers.

Output: maximum sum.

Constraints:

- Time Limit: 1 second
- Memory Limit: 256 MB
- 1 <= Array Size <= 10^6
- -10^9 <= Array Element <= 10^9

Hypothetical Example

Input :arr[]= {100, 200, 300, 400}, k = 2 Output : ???

```
for (int i = 0; i < n; i++) {
  for (int j = i; j < n; j++) {
    int currentSum = 0;
    for (int k = i; k <= j; k++) {
        currentSum += arr[k];
    }
    if (currentSum > maxSum) {
      maxSum = currentSum;
    }
  }
}
```

Complexity?

```
for(int i = 0; i < n; i++) {
  int currentSum = 0;
  for(int j = i; j < n; j++) {
   currentSum += arr[j];
   if (currentSum>maxSum){
     maxSum = currentSum;
   }
  }
}
```

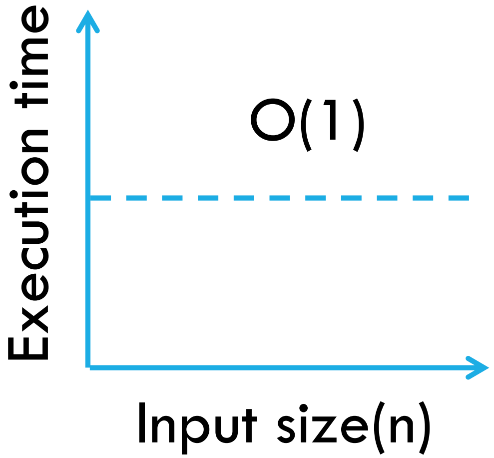
Complexity?

```
for (int i = 1; i < n; i++) {
   currentSum=max(arr[i], currentSum + arr[i]);
   maxSum = max(maxSum, currentSum);
}
```

Efficient solution: Kadane's Algorithm using DP (Ignore –ve subarray sum): Complexity is ???

FUNCTIONS FOR ALGORITHM ANALYSIS

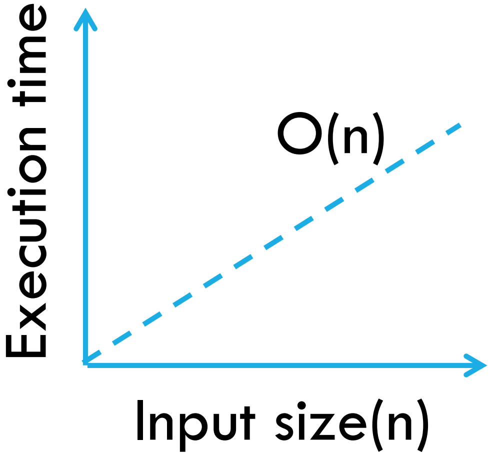
(Seating in the class everyday)



(Constant Function)

Ex.s: Array indexing, Variable assignment, Basic arithmetic operations, …

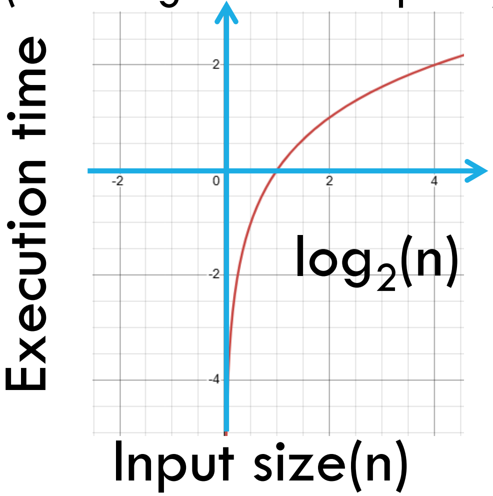
(Reading chapters of book)



(Linear Function)

Ex.s: Searching in unsorted array, Printing all elements of a list, …

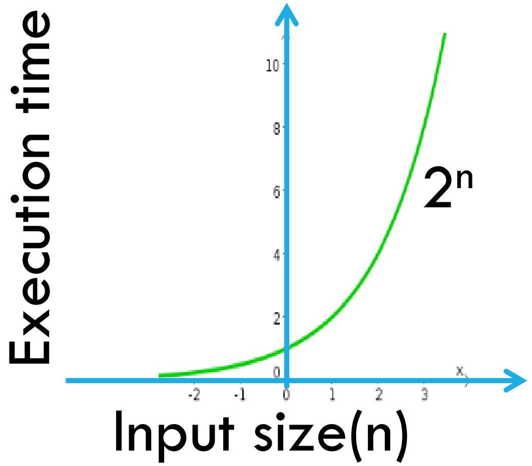
(Climbing a ladder quickly)



(Logarithmic Function)

Ex.s: Finding a word in a dictionary, Treasure hunt, etc…

(Population growth)



(Exponential Function)

Ex.s: Fibonacci sequence, Towers of Hanoi, Generating all subsets of set, etc…

COMPLEXITY EXAMPLES FROM REAL LIFE

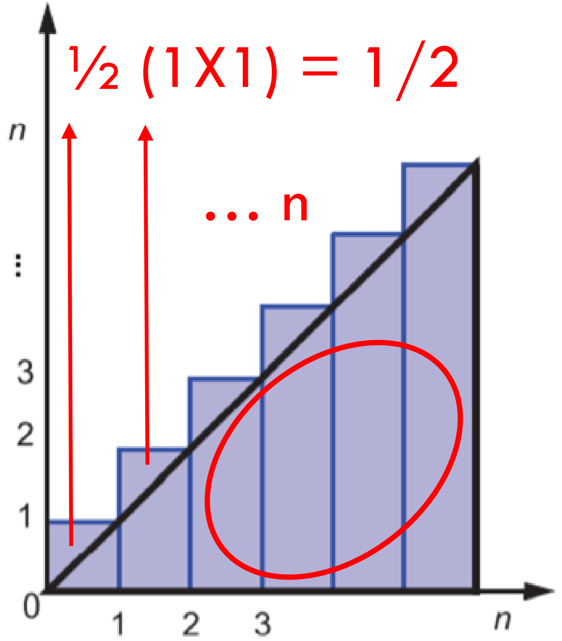
1km	1km	1km	1km	… (Ex.1)
Mon	Tues	Wed	Thurs	

- Total distance over time? (linear)
- Per day effort? (constant)

- Total distance over time? (quadratic)
- Per day effort? (linear)

1km	2kms	3kms	(Ex.2) …
Mon	Tues	Wed	

$\boxed{\tfrac{1}{2}\,(n\text{X}n) = n^2/2 \;+\; n/2}$ ❯ $n(n+1)/2$, for $n>=1$



Que: $1 + 4 + 9 + 16 + 25 + \ldots n$ cubic $\{[n(n+1)(2n+1)]/6\}$

TASKS FOR YOU…COMPLEXITY?

```
function isEvenOrOdd(n) {
    if (n%2 == 0)
        return even;
    else
        return odd;
}
```

```
list<int> numbers {1, 2, 3, 4};
for(int number : numbers)
{
    cout << number <<", ";
}
```

(printing out all the elements)

```
int binarySearch(int array[], int x,
                              int low, int high)
{
  while (low <= high)
  {
      int mid = low + (high - low) / 2;
      if (array[mid] == x) return mid;
      if (array[mid] < x)
         low = mid +1;
      else
         high = mid - 1;
  }
  return -1;
}
```

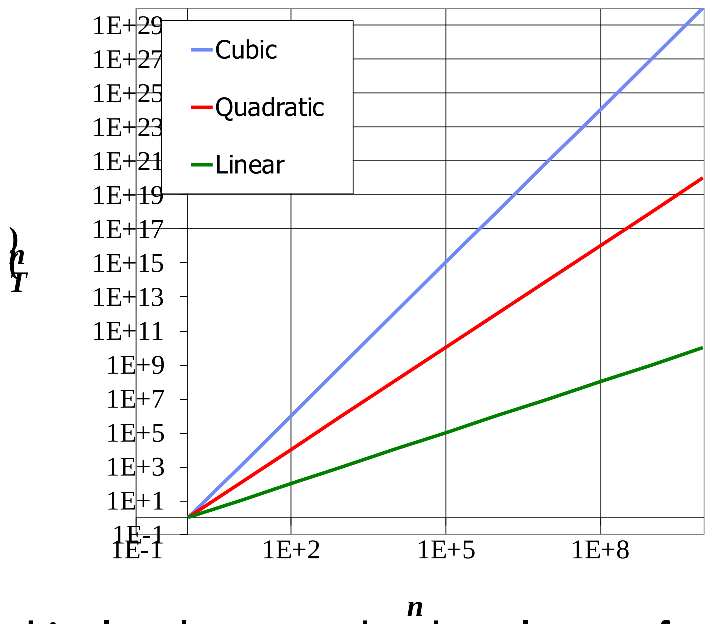
```
int partition(int arr[], int low, int high) { int pivot=arr[high];
   int i = (low - 1);
   for (int j = low; j <= high - 1; j++) {
     if (arr[j] < pivot) { i++; swap(&arr[i], &arr[j]); } }
   swap(&arr[i + 1], &arr[high]); return (i + 1);
}
```

POLYNOMIAL FUNCTIONS AND LOG-LOG PLOT

```
procedure NAIVE-MATRIX-MULTIPLY(A, B)
    n = A.rows
    let C be a new n × n matrix
    for i = 1 to n do
        for j = 1 to n do
            c_ij = 0
            for k = 1 to n do
                c_ij = c_ij + a_ik · b_kj
            end for
        end for
    end for
    return C
end procedure
```

No of multiplications:

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}1 \Rightarrow \sum_{i=1}^{n}\sum_{j=1}^{n} n \Rightarrow \sum_{i=1}^{n} n^2 \Rightarrow n^3$$



(In this log-log graph, the slope of the line corresponds to the growth rate)

-Power law relations become linear.

$y = k.x^n$ (y: dependent and x is independent)

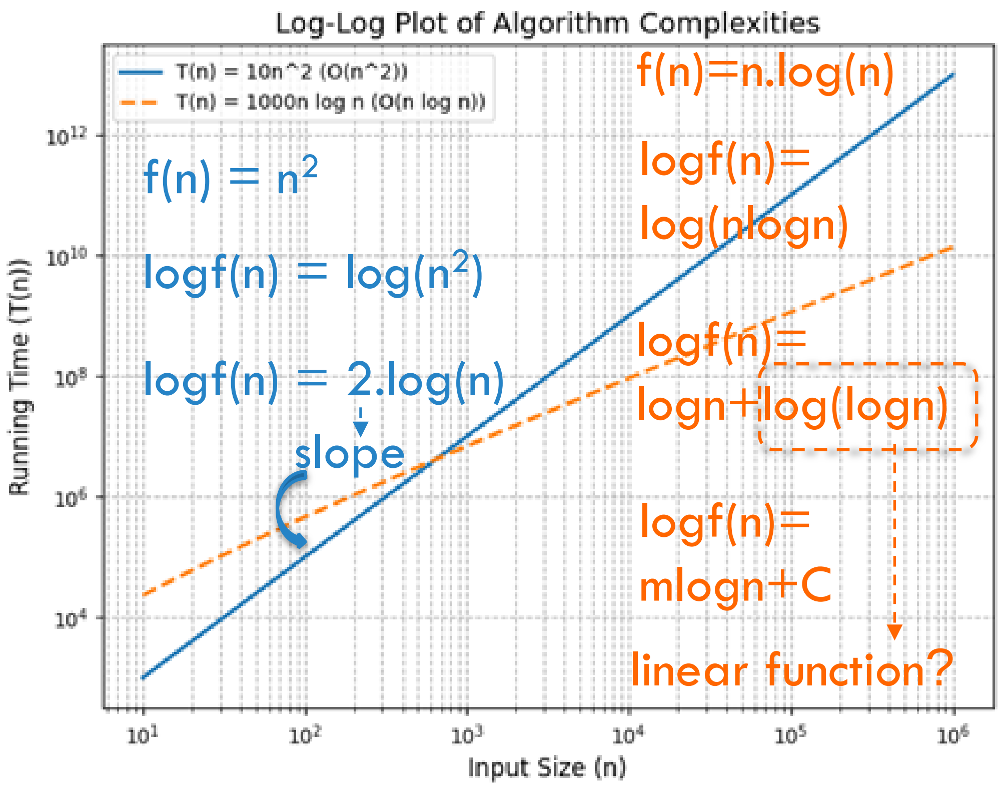
$\log(y) = \log(k) + n.\log(x)$, where n is the slope and log(k) is the intercept.

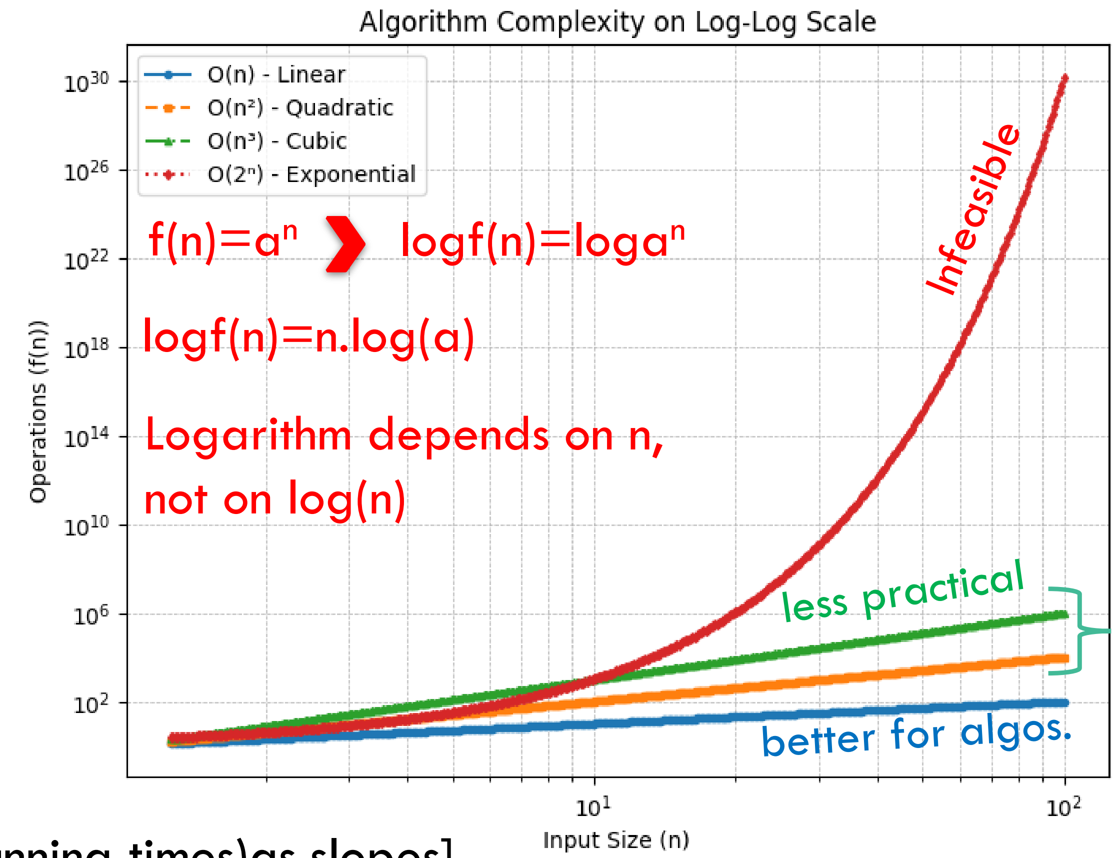
Interestingly, all the functions that we have listed are part of large class of functions called, <u>polynomials</u>:

$g(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \ldots + a_d n^d$ What is d?

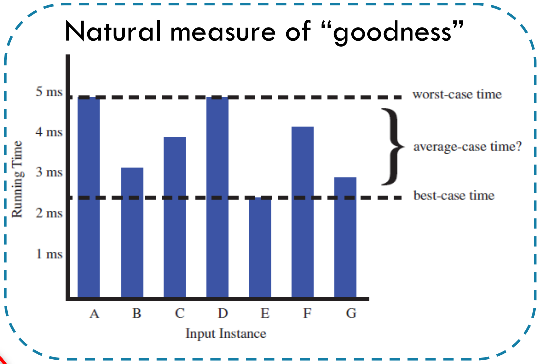
GROWTH RATE OF AN ALGORITHM

-It describes the rate at which the algorithm's resource requirements (time or memory) grow relative to the input size.

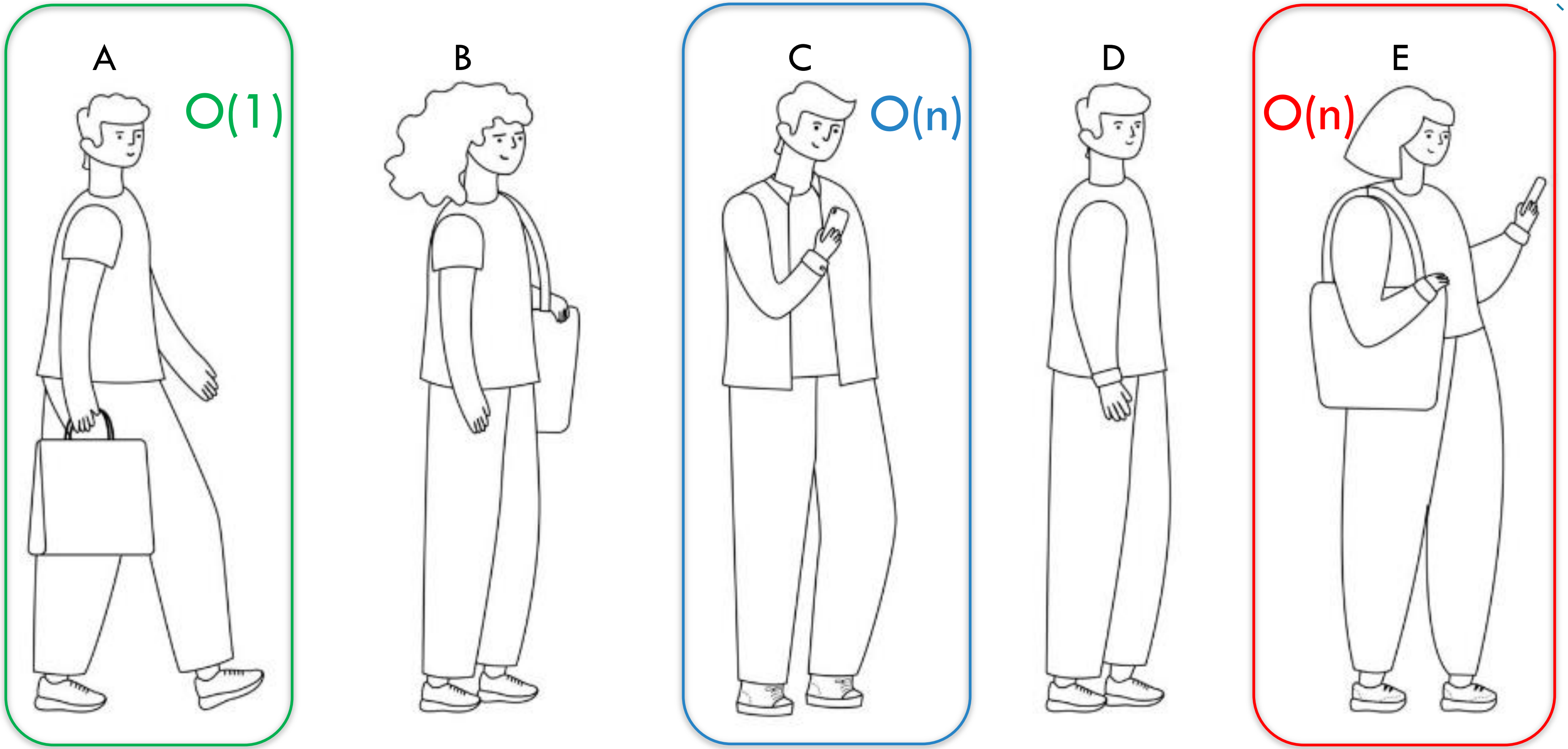




[log-log plot with growth rates (running times)as slopes]



BEST CASE, WORST CASE, AVERAGE CASE



Find A: how many comparisons?

Lower bound

Find E: how many comparisons?

Upper bound

Middle: how many comparisons?

Typical performance

→ (1+2+3+…n)/n → (n(n+1)/2)/n → (n+1)/2 ≈ n/2 Linear fun of n (ignoring ½)

- Why worst case is important and Average case is most difficult to compute?

(This week's Lab: Lab 5)

EMPIRICAL ANALYSIS: COMPLEMENT TO BIG-O

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

… (with a main function calling it)

gprof

Flat profile:

% time	cumulative seconds	self seconds	calls	self ms	total ms	name
95.00	1.90	1.90	1	1900	1900	bubbleSort
5.00	2.00	0.10				main

Call graph:

caller

callee

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.10    1.90                 main [1]
                1.90    0.00       1/1           bubbleSort [2]
-----------------------------------------------
                1.90    0.00       1/1           main [1]
[2]     95.0    1.90    0.00                 bubbleSort [2]
-----------------------------------------------
```

Question: How do you now verify the quadratic complexity of this algorithm? What are the challenges?

USING A TIMER FROM CHRONO

```cpp
#include <chrono>
class Timer
{
private:
    std::chrono::time_point<std::chrono::high_resolution_clock> startTimePoint;
    std::chrono::time_point<std::chrono::high_resolution_clock> endTimePoint;
    double getTimeDifference();

public:
    Timer();
    void start();
    void stop();
    double getDurationInSeconds();
    double getDurationInMilliSeconds();
    double getDurationInMicroSeconds();
};
Timer::Timer() {}
void Timer::start()
{
    startTimePoint = std::chrono::high_resolution_clock::now();
}
void Timer::stop()
{
    endTimePoint = std::chrono::high_resolution_clock::now();
}
double Timer::getTimeDifference()
{
    auto start = std::chrono::time_point_cast<std::chrono::microseconds>(
    auto end = std::chrono::time_point_cast<std::chrono::microseconds>(en
    return end - start;
}
double Timer::getDurationInSeconds()
{
    return getDurationInMilliSeconds() * 0.001; // in seconds
}
double Timer::getDurationInMilliSeconds()
{
    return getTimeDifference() * 0.001; // in milli seconds
}
double Timer::getDurationInMicroSeconds()
{
    return getTimeDifference(); // in micro-seconds
}
```

This week's Lab

```
Linear Search took: 37.647 ms.
Binary Search took: 0.002 ms.
Enter the size of the array: 7
Enter a sorted list of 7 elements:
10 20 30 40 50 60 70
Enter the target item to search for: 40
40 FOUND at index 3
Binary Search took: 0 ms.

...Program finished with exit code 0
Press ENTER to exit console.
```

recursive

Inside main()

```cpp
Timer timer; // initialize timer class object.

timer.start(); // start timer.

linearSearch(arr, n, n); // call to linear search

timer.stop(); // stop timer.

// function to get time in milli seconds
double milliSecs = timer.getDurationInMilliSeconds();

cout << "Linear Search took: " << milliSecs << " ms." << endl;

timer.start(); // start timer.

binarySearch(arr, n, n); // call to binary search

timer.stop(); // stop timer.

// function to get time in milli seconds
milliSecs = timer.getDurationInMilliSeconds();

cout << "Binary Search took: " << milliSecs << " ms." << endl;
```

CONTINUED…

```cpp
// finds and returns the n'th node from the end of the list.
template <typename DT>
SinglyLinkedNode<DT> *SinglyLinkedList<DT>::nthNodeFromEnd(int n)
{
    // code here
    counter = 0;
    tmp = NULL;
    nthNodeFromEndRecursive(head, n);
    return tmp; // return the n'th node from the end
}
```

```cpp
// recursive solution to find out the n'th node from the end.
template <typename DT>
void SinglyLinkedList<DT>::nthNodeFromEndRecursive(SinglyLinkedNode<DT>*head,int n)
{
    if (head == NULL)
        return;

    nthNodeFromEndRecursive(head->next, n);
    counter++;
    if (counter == n)
    {
        tmp = head;
    }
}
```

```cpp
        case '6':
            cout << "Enter N: ";
            cin >> a;
            timer.start();
            node = list.nthNodeFromEnd(a);
            timer.stop();
            if (node == NULL)
                cout << "Such a node does not exist." << endl;
            else
                cout << "N'th node from the end: " << node->dataItem << endl;
            cout << "Time spent: " << timer.getDurationInMilliSeconds() << " ms." << endl;
            break;
```

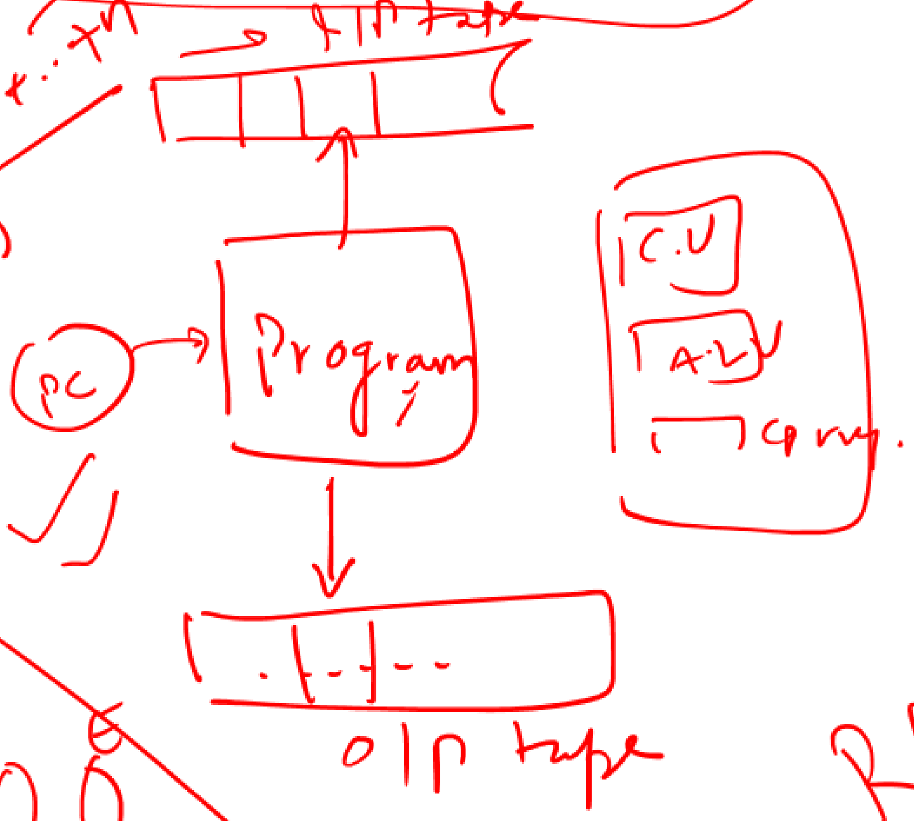
```
Please enter one of the following choices:
1 : Insert at end
2 : Delete from end
3 : Print Forward
4 : Print Backward
5 : Reverse List
6 : Get N'th node from the end
7 : Exit
3
10 20 30 40
Time spent: 0.019 ms.
+-------------------------------------------+
Please enter one of the following choices:
1 : Insert at end
2 : Delete from end
3 : Print Forward
4 : Print Backward
5 : Reverse List
6 : Get N'th node from the end
7 : Exit
6
Enter N: 2
N'th node from the end: 30
Time spent: 0.001 ms.
+-------------------------------------------+
Please enter one of the following choices:
1 : Insert at end
2 : Delete from end
3 : Print Forward
4 : Print Backward
5 : Reverse List
6 : Get N'th node from the end
7 : Exit
```

Lab no:5

THEORETICAL ANALYSIS: THE RAM MODEL

$n(n+1)/2 = \frac{n+1}{2}$ ③ $O(1)$

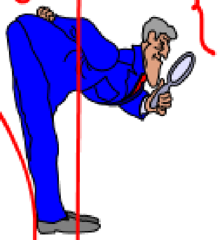
$1+2+3+4+\ldots+n$



Add
Sub
Mul
Div
cmp

RAM

$O(n)$.



```
int max(int a, int b)
{ int max;
  if (a<b)      → 1
     max = a;   → 1
  else
     max = b;   → 1
  return max;
}
```

int a = 5;

b = a[10];

Ex. From previous class

Middle

ARRAYMAX: ANOTHER EXAMPLE

Let us see an example!

O(n)

int arrymax(int a[], int n) {

max = A[0]; 2

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

if A[i] > max

max = A[i];

return max; }

$n-1+1+2(n-1)$

$2(n-1)$

$2(n-1)$

0

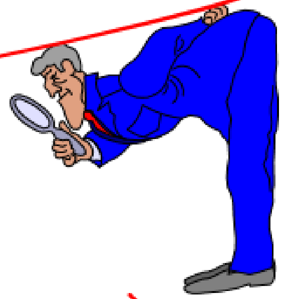
i++

i=i+1

$2+3n-2+2n$

$-2+2n-2+1$

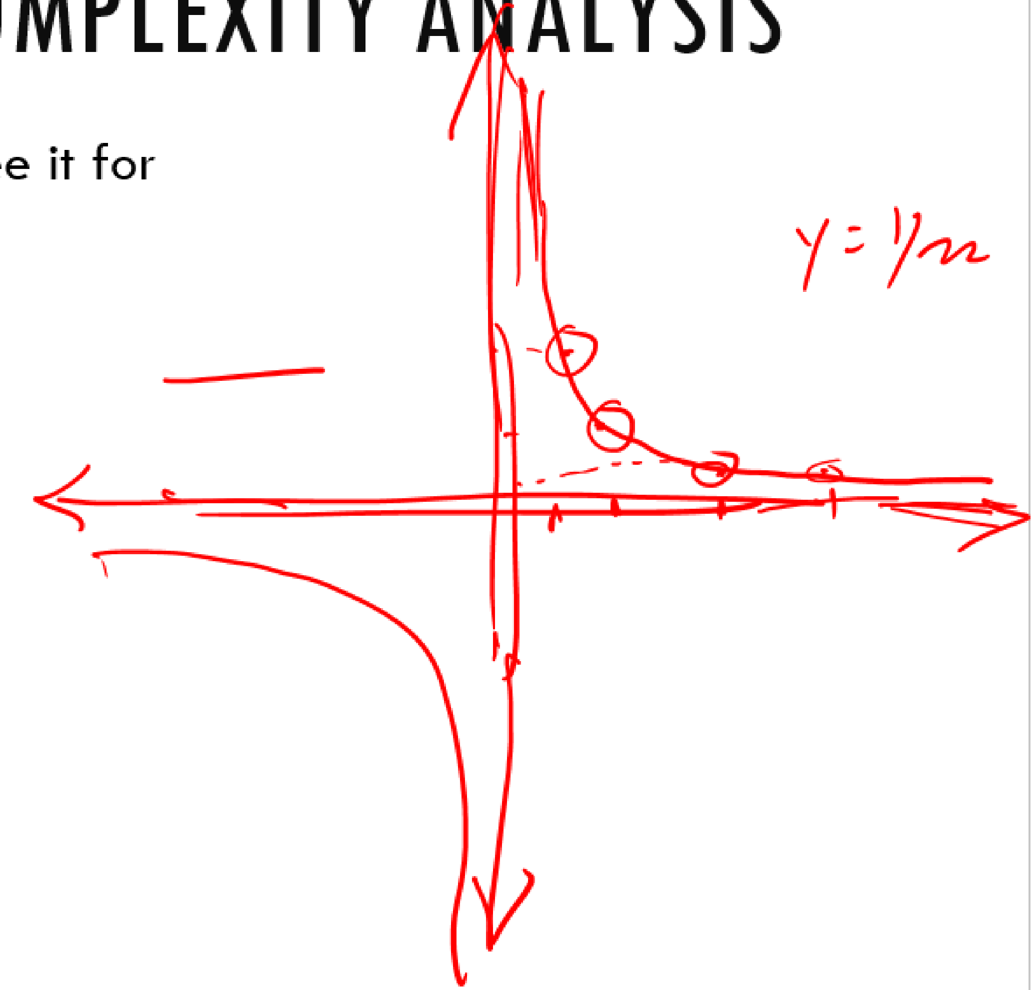
$7n-3$



ASYMPTOTIC NOTATION FOR COMPLEXITY ANALYSIS

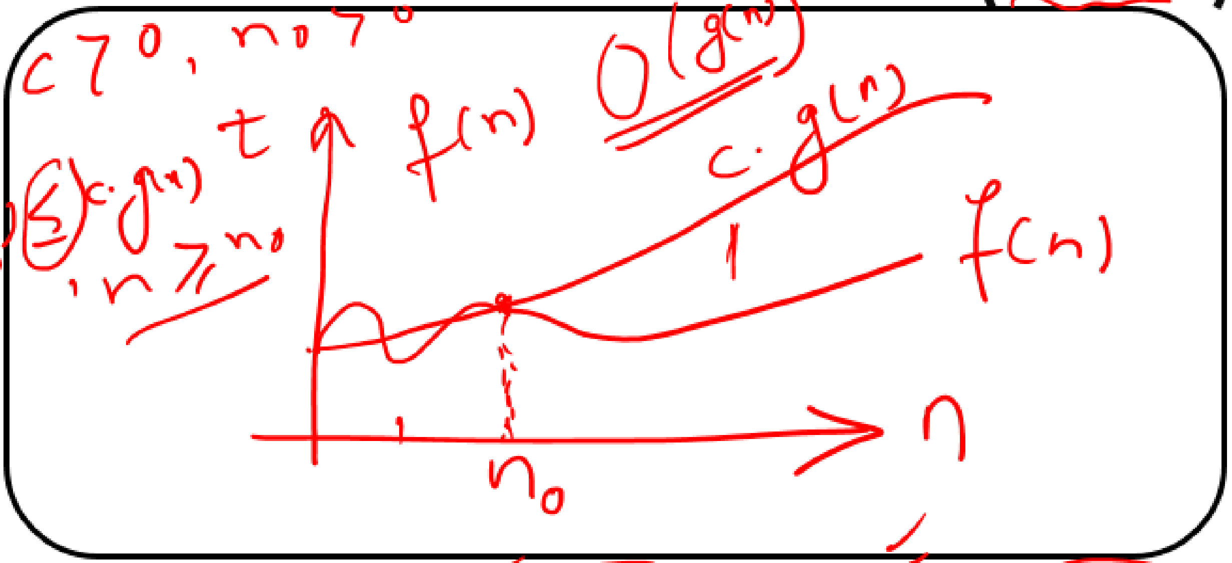
- What is an Asymptote of a curve in maths? Let us see it for $y = 1/x$.

x	y
-1	-1
-2	$-1/2$
-3	$-1/3$
4	$1/4$
$1/2$	2



COMPLEXITY ANALYSIS USING ASYMPTOTIC NOTATION (BIG O)

- Why Big O?
- Let us define it!
- Why is it called a tight upper bound?
- Because, it is as close as possible to the actual growth rate of the function. (Ex: Aryan is at most 20 years old, ~~Aryan is at most 25 years~~ old, Aryan is at most 30 years old. Which one is tight and which one is a loose bound?)
- Linear search is $O(n)$, also $O(n^2)$, also $O(n^3)$, …which one is tight?

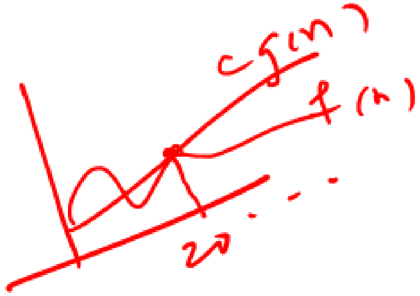


Data (n)	Fast Comp. (0.01n)	Slow Comp. (100logn)
10	0.1 seconds	~332 seconds
100	1 second	~664 seconds
1,000	10 seconds	~997 seconds
10,000	100 seconds	~1329 seconds
1,00,000	1,000 sec (16.7 min)	~1661 sec (27.7 min)
10,00,000	10,000 sec (2.8 hrs)	~1993 sec (33.2 min)
100,00,000	100,000 sec(27.8 hr)	~2325 sec (38.8 min)

Linear search

Binary search

BIG O EXAMPLES



$f(n) \in O(g(n))$

$f(n)$ is $O(g(n))$

Justify: the function, f(n)=4n+20 is O(n)

$4n+20 \leq c \cdot n$

$4n+20 \leq 5n$

c ? n_0

$cn - 4n \geq 20$

$n(c-4) \geq 20$

$n \geq \frac{20}{c-4}$

$c = 5$

$n_0 = 20$

Justify: the function, f(n)=n² ≠ O(n)

$n^2 \leq c \cdot n$

$\frac{n^2}{n} \leq \frac{c n}{n}$

$n \leq c$

$n \geq n_0$

$50 \leq 100$

$200 \geq 500$

$40,000 \leq 20,000$

BIG O EXAMPLES CONTINUED…

$2 \times 1000 + 100 \times \log 1000$

$2996 \le 3000$

$n_0 = 100$

$200 + 100 \log 100$

$864 \le 300$

$8 \le 4 \times 2 \qquad 16 \le 16$

Justify: the function, $f(n)=2^{n+2}$ is $O(2^n)$

$\boxed{2^{n+2} \le c \cdot 2^n}$

$\cancel{2^n} \cdot 2^2 \le c \cdot \cancel{2^n}$

$c = 4$

$n_0 = 1$ ✓

Justify: the function, $f(n)=2n + 100.\log_2 n$ is $O(n)$

Solutions: $c = 102$, $n_0 = 1$ And $c = 3$, $n_0 = 1000$. Which one to choose?

$2n + 100 \log n \le c \cdot n$

$2 + 100 \frac{\log n}{n} \le c$

$c = 3$ ✓

$2 + 100 \log 1 \le 3 \cdot 1$

$2n + 100 \log n \le c \cdot n$

$2n + 100 \log n \le 2n + 100n$

$\le 102n$

$2 \le 102$

c

Analogy: It takes at most 102 minutes to reach your destination. (an overestimate, but works no matter what?)

It actually takes around 3 minutes per kilometer, but only if you're on the highway. (accurate, but applies to long distances)

BIG O EXAMPLES CONTINUED…

$f(n) = (n+1)^2$

$n^2 + 1 + 2n \le c \cdot n^2$

$c = 5$

$n_0 = 1$

Justify: the function, $f(n)=n^2 + n + 2$ is $O(n^2)$

Ans: $c = 2$, and $n_0 = 2$.

$n^2 + n + 2 \le c \cdot n^2$

$1 + \frac{1}{n} + \frac{2}{n^2} \le c$

$c = 2$

$n^2 + n + 2 \le 2 \cdot n^2$

$8 \le 8$

Justify: the function $f(n) = 3n^3 + 20n^2 + 5$ is $O(n^3)$

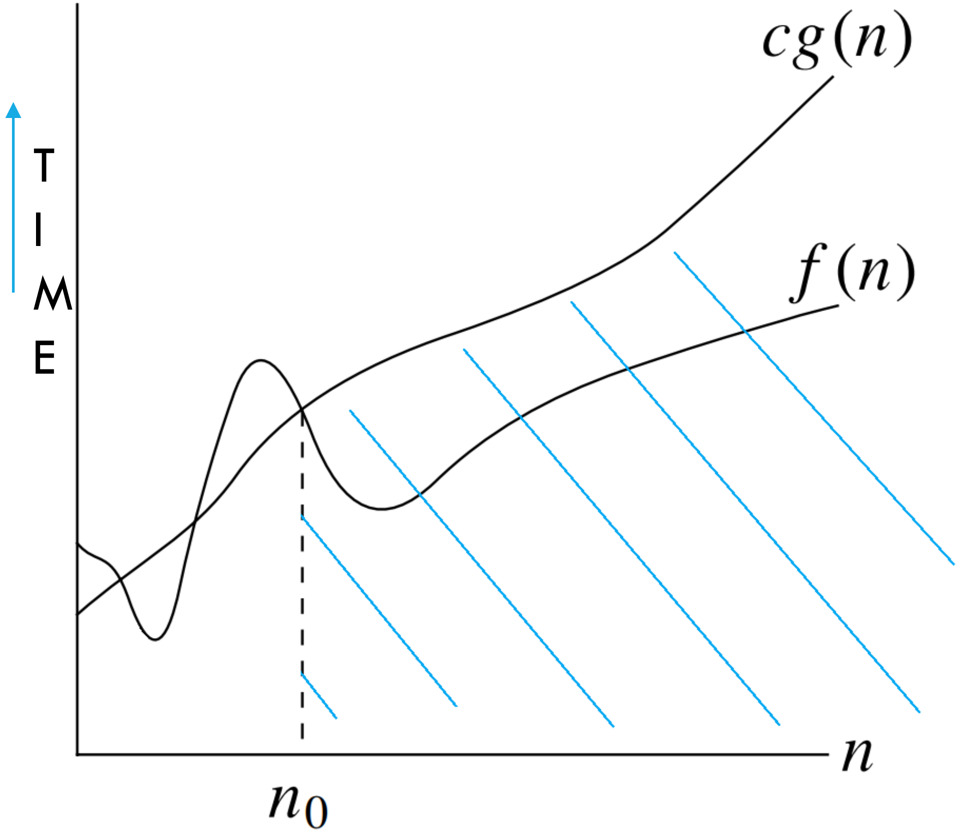
Ans: $c = 4$, and $n_0 = 21$.

$3n^3 + 20n^2 + 5 \le c \cdot n^3$

$3 + \frac{20}{n} + \frac{5}{n^3} \le c$

$c = 4$

RECAP: BIG-O



$f(n)$ is $O(g(n))$

- Why is it called tight upper bound?
- What is the relation between Big-O and growth rate?
- How to choose an appropriate combination of c and n_0 out of many possible ones?

There exists $c > 0$ and n_0 such that $f(n) \leq cg(n)$ whenever $n \geq n_0$.

RECURSIVE FUNCTIONS: RECURRENCE RELATION (EX1)

```
void fun(int n) {
   if (n > 0)
   {
     printf("%d", n);
     fun(n-1);
   }
}

Int main() {
  int x = 4;
  fun(x);
  return 0;
}
```

- A recurrence relation is a way to define a function or sequence in terms of itself.
- Let us solve it?

$$T(n) = \begin{cases} c, n \leq 0 \\ T(n-1) + k \text{ , n > 0} \end{cases}$$

O(n)

$T(n-1) = T(n-1-1) + k$

$\rightarrow T(n) = T(n-2) + k + k = T(n-2) + 2k$

…

$\rightarrow T(n) = T(n-i) + i.k \rightarrow n-i = 0 \rightarrow T(0) + n.k \rightarrow \boxed{c + nk}$

EXAMPLE 2

$T(n) = 2\ T(n/2) + n$ Where, $T(1) = 1$

$T(n/2) = 2\ T(n/4) + (n/2)$

$\rightarrow T(n) = 2\ \{2.T(n/4) + (n/2)\} + n\ = 4\ T(n/4) + 2.n$

$T(n/4) = 2.T(n/8) + (n/4)$

$\rightarrow T(n) = 4.\ \{2.T(n/8) + (n/4)\} + 2.n\ = 8.T(n/8) + 3.n$

$T(n) = 2^i.T(n/2^i) + i.n$

Assuming $n/2^i = 1$

$\rightarrow n = 2^i$

$T(n) = n.T(1) + \log_2 n.n$

$\rightarrow T(n) = n + n\log_2 n$

$\rightarrow O(n\log_2 n)$

$\downarrow$

Log-linear

BIG-O RULES

1. If an algorithm performs a certain sequence of steps f(N)times for a function f, it takes O(f(N)) steps.

   This algorithm examines each of the N items once, so it's performance ???.

```
int findBiggestNumber(int arr[], int size) {
    int biggest = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] > biggest) {
            biggest = arr[i];
        }
    }
    return biggest;
}
```

2. If an algorithm performs an operation that takes f(N) steps and then performs another operation that takes g(N) steps for function f and g, the algorithm's total performance is ???.

   The total runtime of the algorithm is ???.

```
int findBiggestNumber(int arr[], int size) {
    int biggest = arr[0];  // ?
    for (int i = 1; i < size; i++) { //?
        if (arr[i] > biggest) {
            biggest = arr[i];
        }
    }
    return biggest;  //?
}
```

CONTINUED…

3. If an algorithm takes O (f(N) + g(N)) steps and the function f(N) is bigger than g(N), algorithm's performance can be simplified to O (f(N)).

   findBiggestNumber algorithm has O(N+ 2) runtime. When N grows very large, the function N is larger than our constant value 2, so algorithm's runtime can be simplified to ???.

4. If an algorithm performs an operation that takes f(N) steps, and every step performs another operation that takes g(N) steps, algorithm's total performance is ???.

```
int findBiggestNumber(int arr[], int size) {
    int biggest = arr[0];  // O(1)
    for (int i = 1; i < size; i++) { //O(n)
        if (arr[i] > biggest) {
            biggest = arr[i];
        }
    }
    return biggest;  //O(1)
}
```

```
bool containsDuplicates(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[i] == arr[j]) {
                return true; // duplicate found
            } } }
    return false; }
```

$O(n^2)$

BIG-O AND GROWTH RATE

- The big-Oh notation gives an upper bound on the growth rate of a function without capturing hardware details.
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$.
- We can use the big-O notation to rank functions according to their growth rate.

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

- $O(1)$ – constant time, the time is independent of n, e.g. array look-up
- $O(\log n)$ – logarithmic time, e.g. binary search
- $O(n)$ – linear time, e.g. linear search
- $O(n\log n)$ – e.g. efficient sorting algorithms
- $O(n^2)$ – quadratic time, e.g. selection sort
- $O(n^k)$ – polynomial (where k is some constant)
- $O(2^n)$ – exponential time, very slow!
- $O(1) < O(\log n) < O(n) < O(n * \log n) < O(n^2) < O(n^3) < O(2^n)$

STRICT UPPER BOUND: SMALL-O

- What about small o ? (Upper bound that is not tight)
- Used in Asymptotic proofs.
- If f(n) = o(g(n)), it means f(n) grows strictly slower than g(n).

not a tight bound but an overestimate

$$0 \le f(n) < c \cdot g(n) \qquad \text{Or,} \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Let f(n) = nlogn, and g(n) = n^2 Prove that f(n) is o(g(n))

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \longrightarrow \lim_{n\to\infty} \frac{\text{nlogn}}{n^2} \longrightarrow \lim_{n\to\infty} \frac{\text{logn}}{n} \longrightarrow$$

L-Hospital Rule $\lim_{n\to\infty} \frac{d(\text{logn})/dn}{d(n)/dn} \longrightarrow \lim_{n\to\infty} \frac{1/n}{1} = 0$

Hence, nlogn = o(n^2)

Prove that: n = o(n^2)

$$\lim_{n\to\infty} \frac{n}{n^2} \longrightarrow \lim_{n\to\infty} \frac{1}{n} = 0$$

Hence, n = o(n^2)

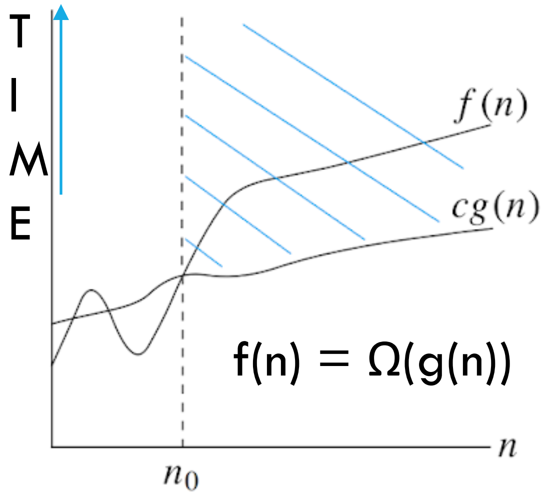
Prove that: $n^2 \neq$ o(n^2)

$$\lim_{n\to\infty} \frac{n^2}{n^2} = 1 \neq 0$$

Hence, $n^2 \neq$ o(n^2) Rather, n^2 = O(n^2)

Big-O allows equality, but small-o requires strict growth separation.

BEST CASE BIG Ω AND AVERAGE CASE BIG Θ

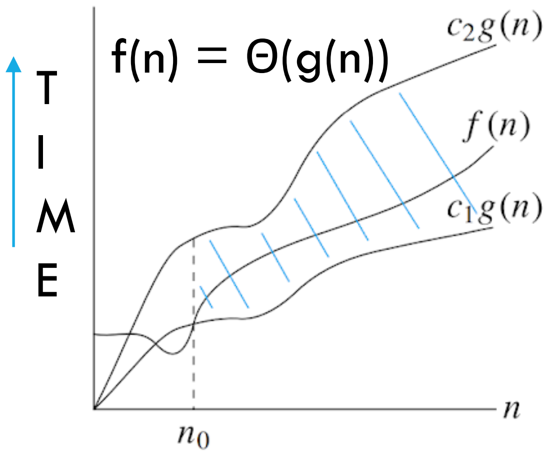


Big-Omega Notation (Ω)

- Just like Bio-O provides asymptotic upper-bound, Big-Ω provides asymptotic lower-bound on the running time.
- f(n) is Ω(g(n)) if there exists a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c.g(n)$ for all $n \geq n_0$

Ω(nlog n)

Let, f(n) = 3n.logn + 2n   Justification: 3n.logn+2n ≥ 3n.logn, for n ≥ 2



Big-Theta Notation (Θ)

f(n) is Θ(g(n)) , if: *f*(*n*) is <u>both</u> *O*(*g*(*n*)) and Ω(*g*(*n*))

f(n) is Θ(g(n)) if there are constants $c_1 > 0$ and $c_2 > 0$ and an integer constant $n_0 \geq 1$ such that $c_1.g(n) \leq f(n) \leq c_2.g(n)$ for $n \geq n_0$

3nlogn+4n+5logn   is Θ(nlog n)   3nlogn ≤   3nlogn+4n+5logn   ≤ (3+4+5) nlogn for n ≥ 2

Que. For You: Linear search, Binary search?   O(n), Ω(1), Θ(n/2)   O(logn), Ω(1), Θ(logn)

EXAMPLES OF BIG-Ω AND BIG-Θ

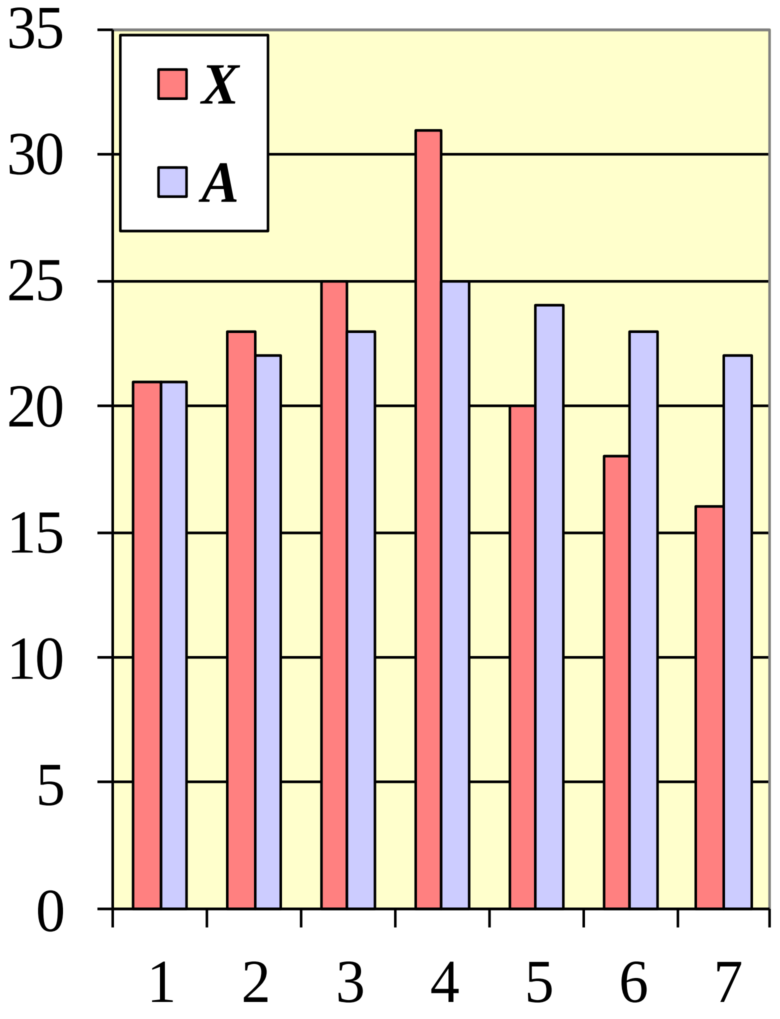
- $5n^2$ is $\Omega(n^2)$
  - $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c.g(n)$ for $n \geq n_0$
  - For, say $c = 5$ and $n_0 = 1$ ➔ $5.1^2 \geq 5.1^2$ True.
- $5n^2$ is $\Omega(n)$
  - $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c.g(n)$ for $n \geq n_0$
  - For, say $c = 1$ and $n_0 = 1$ ➔ $5.1^2 \geq 1.1$ True.
- $5n^2$ is $\Theta(n^2)$
  - $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c.g(n)$ for $n \geq n_0$
  - Let $c = 5$ and $n_0 = 1$

PREFIX AVERAGE EXAMPLE

- The i-th prefix average of an array X is average of the first (i + 1) elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

Applications:
Eco (Mutual
Fund Avgs.)



Algorithm prefixAverages1(*X, n*)
Input array *X* of *n* integers
Output array *A* of prefix averages of *X*

A ← new array of *n* integers	*n*
for *i* ← 0 **to** *n* − 1 **do**	*n*
s ← *0*	*n*
for *j* ← 0 **to** *i* **do**	1 + 2 + …+ n
s ← *s* + *X*[*j*]	1 + 2 + …+ n
A[*i*] ← *s* / (*i* + 1)	*n*
return *A*	1

$O(n^2)$

Algorithm *prefixAverages2*(*X, n*)
Input array *X* of *n* integers
Output array *A* of prefix averages of *X*

A ← new array of *n* integers	*n*
s ← 0	1
for *i* ← 0 **to** *n* − 1 **do**	*n*
s ← *s* + *X*[*i*]	*n*
A[*i*] ← *s* / (*i* + 1)	*n*
return *A*	1

$O(n)$

THANK YOU!

Next Class: Common Data structures (Stacks, Queues, Deques etc.)