Introduction to Algorithms

UNIT 1

Algorithm

- An algorithm is a sequence of unambiguous instructions for obtaining a required output for any legitimate input in finite amount of time.
- The name comes after Persian mathematician "Abu Jafer Mohammed Ibn Musa Alkhowarizmi".

Properties of an Algorithm

- An algorithm can take **zero or more inputs**.

  - Data to be transformed to produce the output .
  - Must specify type, amount, and form of data

- It should give **one or more outputs**.

  - It is possible to have no output

- **Definiteness**: Each instruction should be clear and unambiguous.

  - Specify the sequence of events including how to handle errors.

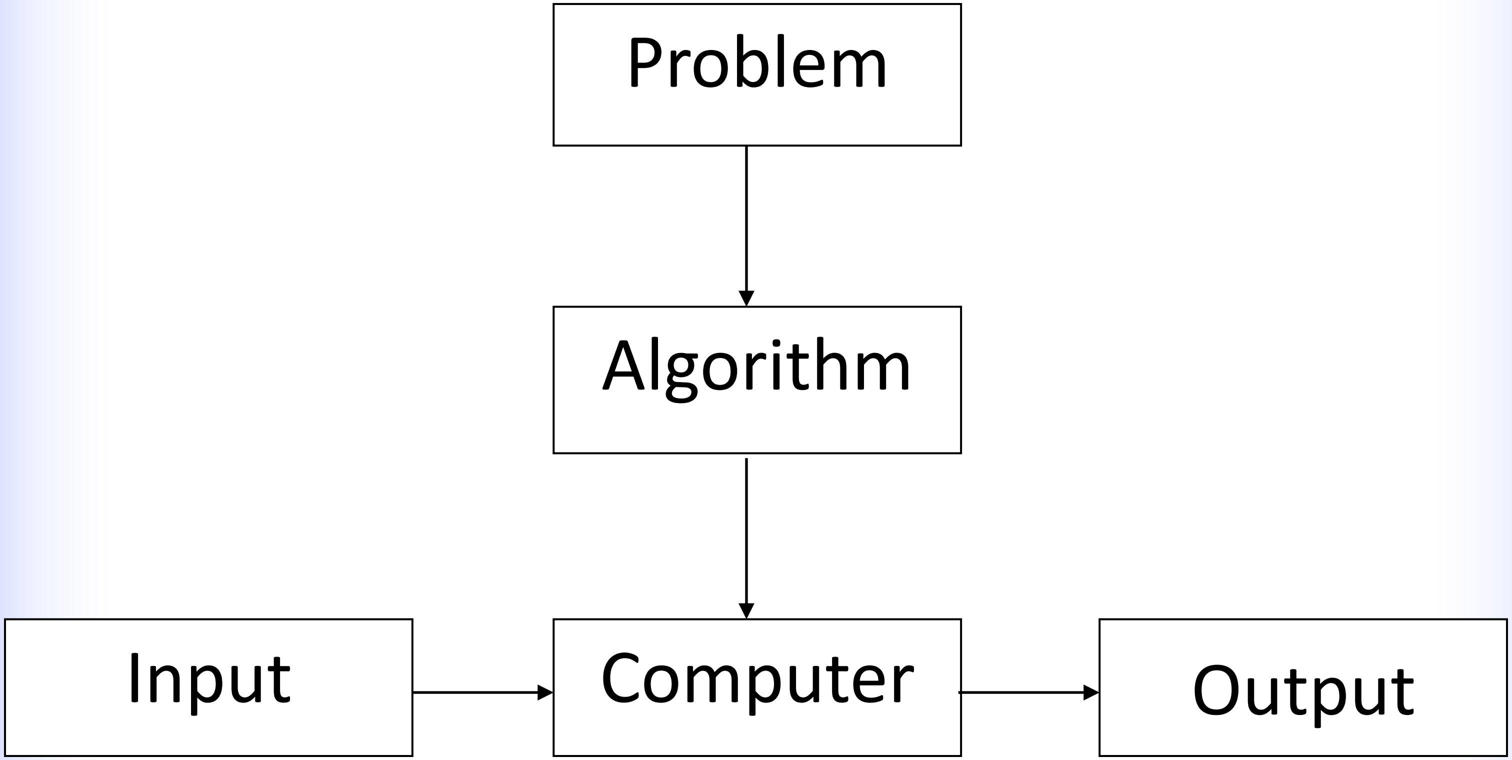
Properties of an Algorithm

- **Finiteness**: All cases must terminate in finite number of steps

  - Must eventually stop.
- **Effectiveness**: It must be feasible.

  - all steps are doable

Notion of Algorithm



Notion of Algorithm

- The non ambiguity requirement for each step of an algorithm can not be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- Several algorithms for solving the same problem may exist.
- Algorithm for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

Euclid's algorithm for computing gcd (m, n):

- gcd(m,n) is defined as the largest integer that divides both m and n evenly with a remainder zero.

1. If n=0 return the value of m as the answer and stop else proceed to step 2.
2. Divide m by n and assign the value of the remainder to r.
3. Assign the value of n to m and the value of r to n. Go to step 1.

The pseudo code of the Euclidian's algorithm

ALGORITHM Euclid(m,n) // Computes gcd (m,n) by Euclid's Algorithm.

// Input : Two non negative, not both zero integers m & n.

// Output : gcd of m &n.

```
while n≠0 do
    r←m mod n
      m←n
    n←r
return m.
```

Recursive Integer Checking Algorithm:

1. Assign the value of min{m,n} to t.
2. Divide m by t.

   If the remainder is zero go to Step 3. Otherwise go to step 4.
3. Divide n by t.

   If the remainder is zero then gcd=t and Stop. Otherwise go to step 4
4. Decrement the value of t by 1 and go to Step 2.

School Procedure to find out gcd (m,n):

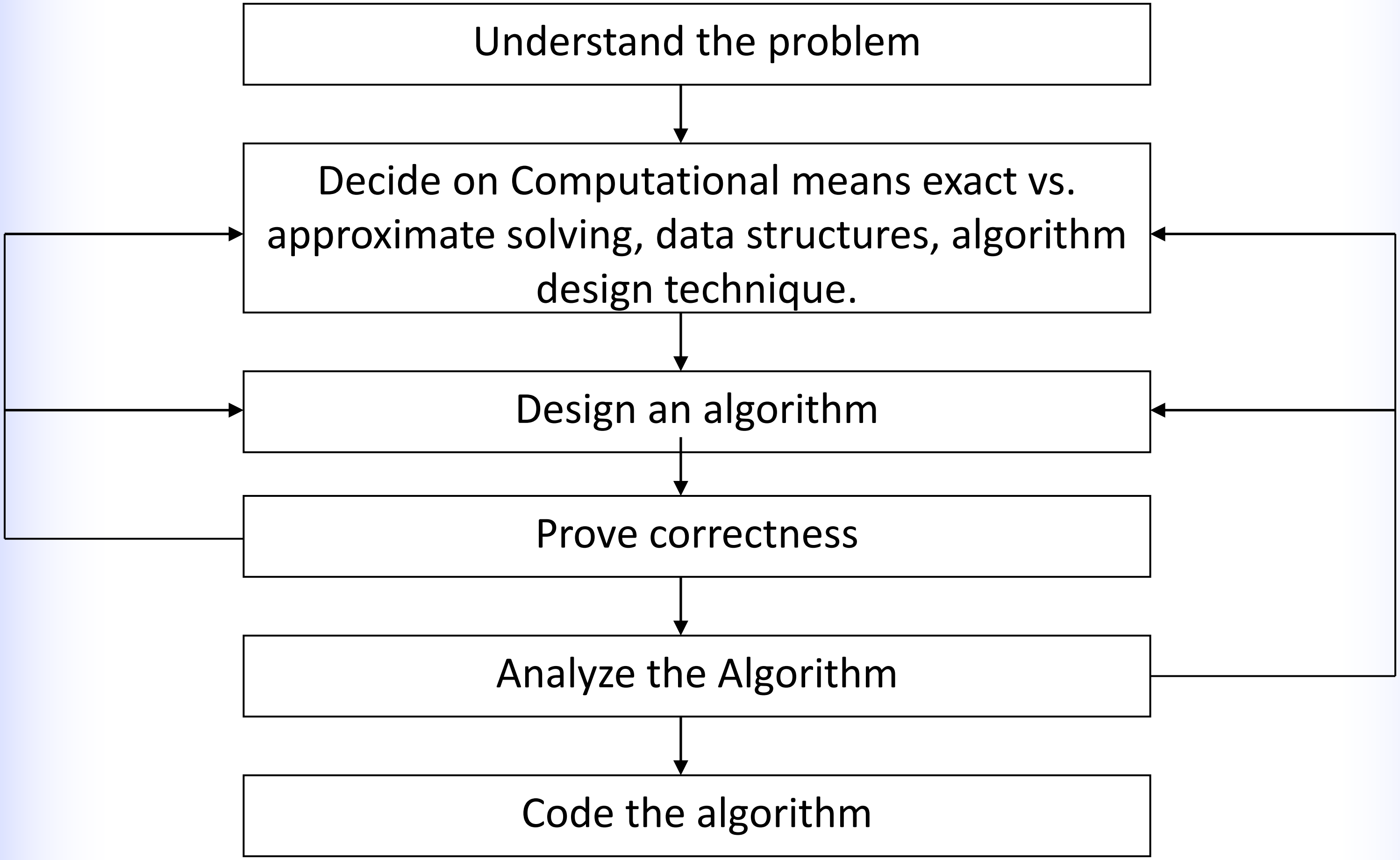
1. Find the prime factors of m.
2. Find the prime factors of n.
3. Identify the common factors in the 2 prime expansions. [If p is a common factor

   appearing pm and pn times in m and n then it should be repeated min (pm, pn) times.
4. Gcd = product of all common factors. Return gcd.

Algorithm to generate prime numbers up to a given integer n.

- ALGORITHM Sieve of Eratostenes(n)
- //It generates the list of prime numbers upto an integer n
- // INPUT : A positive integer n>=2
- // OUTPUT : Array of all prime numbers <=n.
- For p← 2 to n do
-     a[p]←p
- For p← 2 to $\lfloor \sqrt{n} \rfloor$ do
-     if a[p]≠0
-         j←p*p
-         while j≤n do
-             a[j] ← 0
-             j←j+p
- // copy remaining elements of A to L. of Primes
- i←0
- For p← 2 to n do
-     If a[p]≠0
-         L[i]←A[p]
-         i ←i+1
- Return L.

Fundamentals of Algorithmic Problem Solving



Understand the problem:

- Read the problem description carefully and ask questions if u have any doubt about the problem, do a small examples by hand, think about special cases, and ask questions again if needed.
- It is very important to specify exactly the **range of inputs** of an algorithm needs to handle.
- **Correct algorithm is not the one which works most of the time, but is the one which works correctly for all legitimate input.**

Ascertaining the capabilities of a Computational device:

- Most of the algorithms today are destined to be programmed for a computer closely resembling the von Neumann Machine architecture where instructions are executed sequentially. Such algorithms are called **Sequential algorithms**.
- Some newer computer can execute operations parallel. The algorithms that take advantage of this capability are called **Parallel algorithms**.

Choosing between the Exact and Approximate Problem solving:

- Problem can be solved **exactly** or **approximately**.
- Sometimes it is necessary to opt for approximation algorithm because,
  - There are some problems which **can not be s**olved **exactly** like **finding square roots**.
  - Algorithm for solving a problem exactly can be **unacceptably slow** because of problem's intrinsic complexity like TSP.

Deciding appropriate data structure:

- Some algorithms do not demand any ingenuity in representing their inputs.
  - Dynamic programming , Sorting
- But some of the algorithm designing techniques depends intimately on structuring data specifying a problem instance.

Algorithm design techniques:

- It is a general approach to solving a problem algorithmically that is applicable to a variety of problems from different areas of computing.
- They **provide guidance** for designing algorithms for new problems.
- Algorithm design technique makes it possible to **classify** algorithms according to an underlying idea.

Methods of Specifying an Algorithm:

1. **Using Natural Language:** This method leads to ambiguity. Clear description of algorithm is difficult.
2. **Using Pseudo codes:** It is a mixture of natural language and programming language like constructs. It is more precise than a natural language.
3. **Using Flow Charts:** It is a method of expressing an algorithm by a collection of connected geometric shapes containing the description of algorithm.

Proving algorithms correctness:

- We have to prove that the algorithm yields required result for every legitimate input in a finite amount of time.
- **Validation**
- **Common Technique – use Mathematical Induction**
- **Prove – with a specific instance or disprove – by example**

Analyzing an Algorithm:

1. **Time efficiency**: How fast the algorithm runs.
2. **Space efficiency**: How much extra space the algorithm needs.
3. **Simplicity**: Simpler algorithms are easier to understand. But it is relative and depends on the user
4. **Generality**: It is easier to design an algorithm for a problem posed in more general terms. Specify **Range of inputs**

- **If any of the above steps are not satisfied, go back to design**

Coding an algorithm:

- **Good Algorithm is a result of repeated effort and rework**
- Most of the algorithms are destined to be implemented on computer programs.
- **Not only implementation but also the optimization of the code is necessary.** This increases the speed of operation.
- A working program provides the opportunity in allowing empirical analysis of the underlying program.
- **Do Testing and debugging regourously**

Important problem types

1. Sorting
2. Searching
3. String Processing
4. Graph Problems
5. Combinational Problems
6. Geometric Problems
7. Numerical Problems

Analysis Framework

- General Framework for analyzing the efficiency of algorithms
- Time efficiency indicates how **fast an algorithm runs.**
- Space efficiency deals with the **extra space the algorithm needs.**
- Now days the amount of extra space required by an algorithm is typically not of much concern.

Measuring input size

- Almost all algorithms run longer on larger inputs.
- It takes longer time for sorting larger arrays, multiplying larger element matrix.
- Algorithm efficiency must be a function of input size n.
- strings – no of characters
- numbers – bits in binary representation
- Sometimes two inputs - nodes and edges

Units of measuring Running Time

- Use physical unit of time i.e. Milliseconds

  \- not possible (varies with machine speed, compiler, program quality etc.)

- Identify the basic operation and compute the no of times it is executed

- The operation that contributes most towards the running time of the algorithm.
- It is the most time-consuming operation in the algorithm's inner most loop

- Ex: Key comparison operation in Searching and sorting algorithms

$$T(n) \approx c_{op}\, C(n)$$

- n : Input size
- T(n) : Running time
- c_{op} : Execution time for basic operation
- C(n) : no of times the basic operation is executed.
- The count C(n) does not contain any info about operations that are not basic.
- This count is often computed only approximately.

Order of Growth

- Difference in algorithm efficiencies may not be significant for smaller size so consider order of growth.
- For example, in the case of Euclid’s algorithm the efficiency of the algorithm becomes clear only in the case of the large input difference.

Orders of growth

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

•The function growing the slowest among these is the logarithmic function. On the other end the exponential and the factorial function grows fast.

•It is better to have an algorithm whose basic operation is logarithmic in nature.

Worst-case, Best-case and Average-case Efficiencies.

- For some algorithms, the running time depends on the specifics of a particular input.
- In such a scenario, we may have to analyze for three cases
- Worst-case
- Best-case
- Average -case

Worst-case, Best-case and Average-case Efficiencies.

ALGORITHM *SequentialSearch* $\{A[0..n-1], K)$

//Searches for a given value in a given array by sequential search

//Input: An array $A[0..n-1]$ and a search key K

//Output: Returns the index of the first element of A that matches K or

-1 if there are no matching elements

$i \leftarrow 0$

while $i < n$ and $A[i] \neq K$ do

    $i \leftarrow i + 1$

if $i<n$

    return i

else

    return -1

- **Worst-case efficiency:**
  - The **worst-case efficiency** of an algorithm is its efficiency for the worst case input of size *n*
  - An input (or inputs) of size *n* for which the **algorithm runs the longest** among all possible inputs of that size.
  - $\mathbf{C_{worst}(n)= n.}$
- **Best-case efficiency:**
  - The **best-case efficiency** of an algorithm is its efficiency for the best case input of size *n*
  - An input (or inputs) of size *n* for which the **algorithm runs the fastest** among all possible inputs of that size.
  - $\mathbf{C_{best}(n)= 1.}$

- **Average case efficiency:**
  - The **average case efficiency** of an algorithm is its efficiency for the **random input of size *n*.**
  - $C_{avg}(n) = p(n+1)/2 + n(1-p).$
- **Amortized efficiency**
- It applies not to a single run of an algorithm but rather to a **sequence of operations** performed on the same data structure.
- It turns out that in some situations a single operation can be expensive, but total for an entire sequence of n operation is always significantly better than worst-case efficiency of that single operation multiplied by n.

Three Asymptotic Notation:

O-notation:

A function t(n) is said to be in O(g(n)), denoted t(n) € O(g(n)), if t(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

t(n) ≤ c g(n)   for all n ≥ n_0

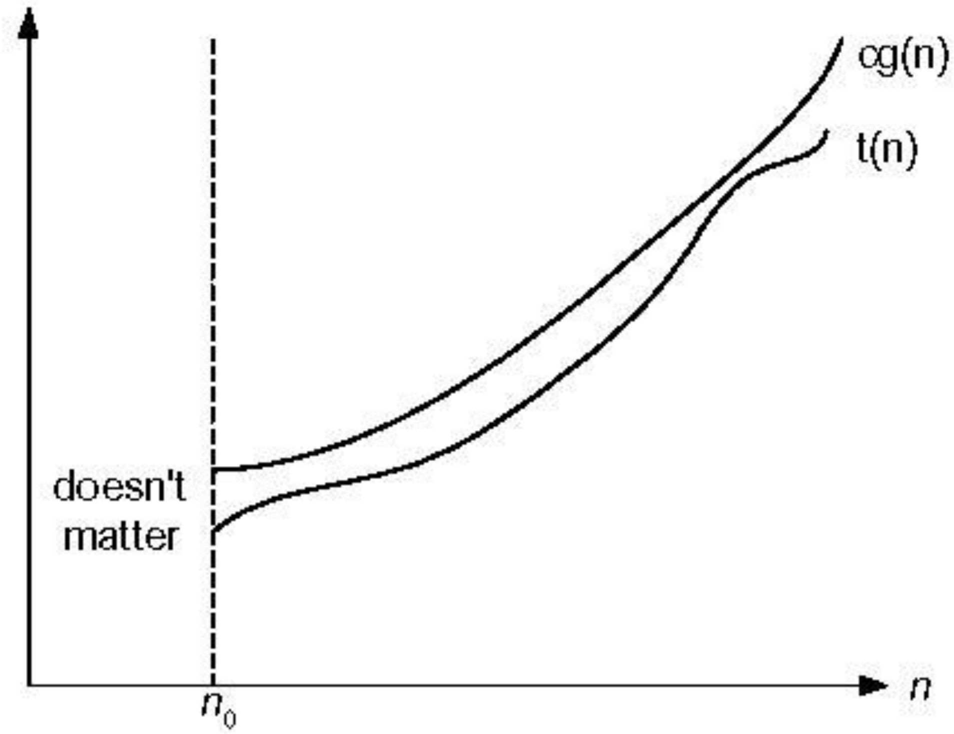
Ex: $100n + 5$ € $O(n^2)$ with $n_0=5$

$100n + 5 \leq 100n + n$ (for all $n \geq 5$)

$= 101n$

$\leq 101n^2$

O-notation:



Ω-notation:

- A function t(n) is said to be in Ω(g(n)), denoted t(n) € Ω(g(n)), if t(n) is bounded below by some positive constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that:

  $t(n) \geq c\, g(n)$ for all $n \geq n_0$.

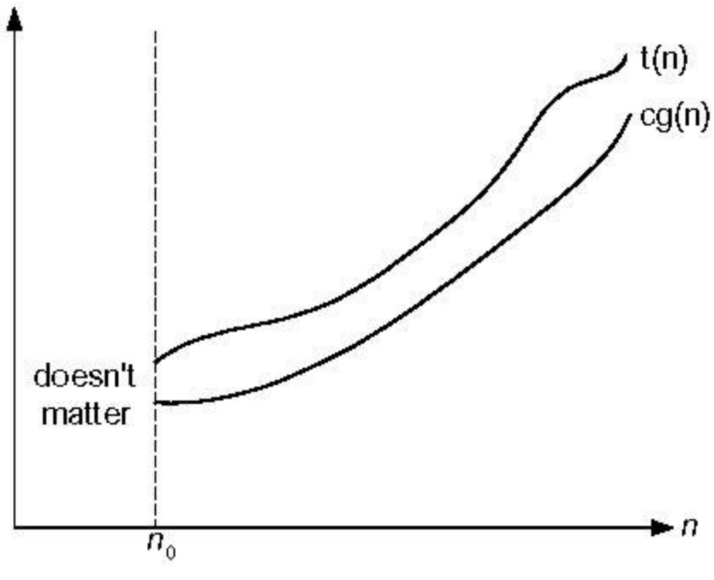


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Θ-notation:

- A function $t(n)$ is said to be in Θ $(g(n))$, denoted $t(n)$ € **Θ**$(g(n)$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that

$c_2 g(n) \le t(n) \le c_1 g(n)$ for all $n \ge n_0$.

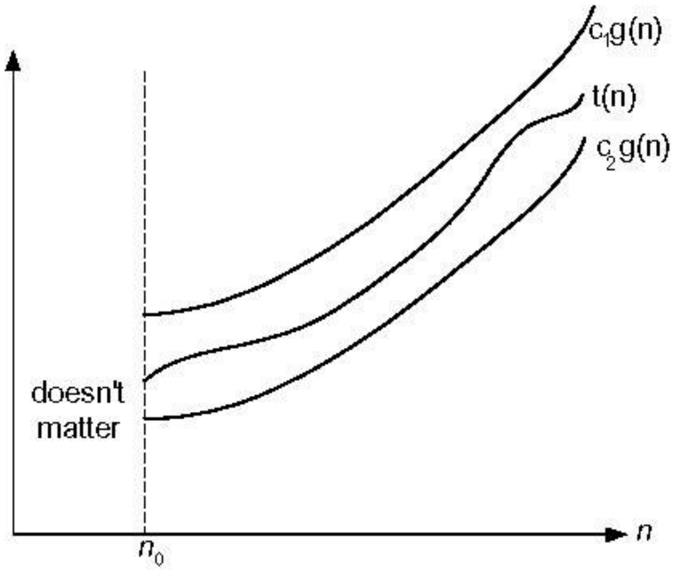


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Informal Definitions

- **$O(g(n))$ is the set of all functions with a smaller or same order of growth as $g(n)$.**
  - Linear functions have smaller order of growth of n^2
  - Quadratic functions have same order of growth of n^2
- $\Omega(g(n))$ is the set of all functions with a larger or same order of growth as $g(n)$.
- $\Theta(g(n))$ is the set of all functions that have the same order of growth as $g(n)$.
  - Every quadratic equation an^2+bn+c with $a>0$ is in $\Theta(n^2)$.

Asymptotic growth rate

- A way of comparing functions that ignores constant factors and small input sizes

- $O(g(n))$: class of functions $f(n)$ that grow ***no faster*** than $g(n)$

- $\Theta(g(n))$: class of functions $f(n)$ that grow ***at same rate*** as $g(n)$

- $\Omega(g(n))$: class of functions $f(n)$ that grow ***at least as fast*** as $g(n)$

Basic Asymptotic Efficiency Classes

Class	Name	Comments
1	constant	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	logarithmic	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm. Note that a logarithmic algorithm cannot take into account all its input (or even a fixed fraction of it): any algorithm that does so will have at least linear running time.
n	linear	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	n-log-n	Many divide-and-conquer algorithms, including mergesort and quicksort in the average case, fall into this category.
n^2	quadratic	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elemen-tary sorting algorithms and certain operations on n-by-n matrices are standard examples.
n^3	cubic	Typically, characterizes efficiency of algorithms with three embedded loops. Several nontrivial algorithms from linear algebra fall into this class.
2^n	exponential	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and faster orders of growth as well.
$n!$	factorial	Typical for algorithms that generate all permutations of an n-element set.

Using limits for comparing order of growth

$$\lim_{n\to\infty} t(n)/g(n) = \begin{cases} 0 & \boldsymbol{t(n)} \text{ has smaller order of growth than } \boldsymbol{g(n)} \\ c > 0 & \boldsymbol{t(n)} \text{ has the same order of growth than } \boldsymbol{g(n)} \\ \infty & \boldsymbol{t(n)} \text{ has larger order of growth than } \boldsymbol{g(n)} \end{cases}$$

Mathematical Analysis of a Nonrecursive Algorithm

1. Decide on a parameter(s) indicating an input's size.
2. Identify the algorithm's **basic operation**.
3. Check whether the number of times the basic operation is executed depends only on the input size. If it also depends on some additional property, determine the worst-case, average-case, and best-case complexities separately.
4. Find out C(n) [the number of times the algorithm's basic operation is executed.]
5. Using standard formulas establish the order of growth.

Finding Largest Element in an array

ALGORITHM Maxelements (A[0..n - 1])

//Determines the value of the largest element in a given array

//Input: An array A [0..n - 1] of real numbers

//Output: The value of the largest element in the array

maxval = A[0]

for i ← 1 to n — 1 do

    if A[i] > maxval

        maxval ← A[i]

return maxval

$$\sum_{i=1}^{n-1} i = n - 1 \in \theta(n)$$

Element uniqueness problem:

ALGORITHM Distinct_elements (A[0..n - 1])

//Checks whether all the elements in a given array are distinct

//Input: An array A [0..n - 1]

//Output: Returns "true" if A contains distinct elements, otherwise

//Returns "false“.

```
for i ← 0 to n – 2 do
    for j ← i + 1 to n - 1 do
        if A[i] = A [j]
            return false
return true
```

Summation formulae To Calculate C(n)(Refer Appendix A of Anany Leviton)

- Formulae:

$$\sum_{i=l}^{u} 1 = 1 + 1 + \cdots \ldots \ldots + 1 = u - l + 1$$

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots \ldots \ldots + n = \frac{n(n+1)}{2} = \frac{1}{2} n^2$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} (a_i) \pm \sum_{i=l}^{u} (b_i)$$

Mathematical Analysis of a Recursive Algorithm

- Decide on a parameter (or parameters) indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
- Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
- Solve the recurrence or at least ascertain the order of growth of its solution.

The factorial function F(n)=n!

- The factorial function F(n)=n! for an arbitrary non-negative integer.
- n!= 1….(n-1).n = (n-1)!.n for n≥1.

```
ALGORITHM   F(n)
//Computes n! recursively.
//Input: A non-negative integer.
//Output: The value of n!
If n=0
    return 1
else
        return  F(n-1) * n
```

$F(n) = F(n - 1) * n$ for $n > 0$ and **$0! = 1$**

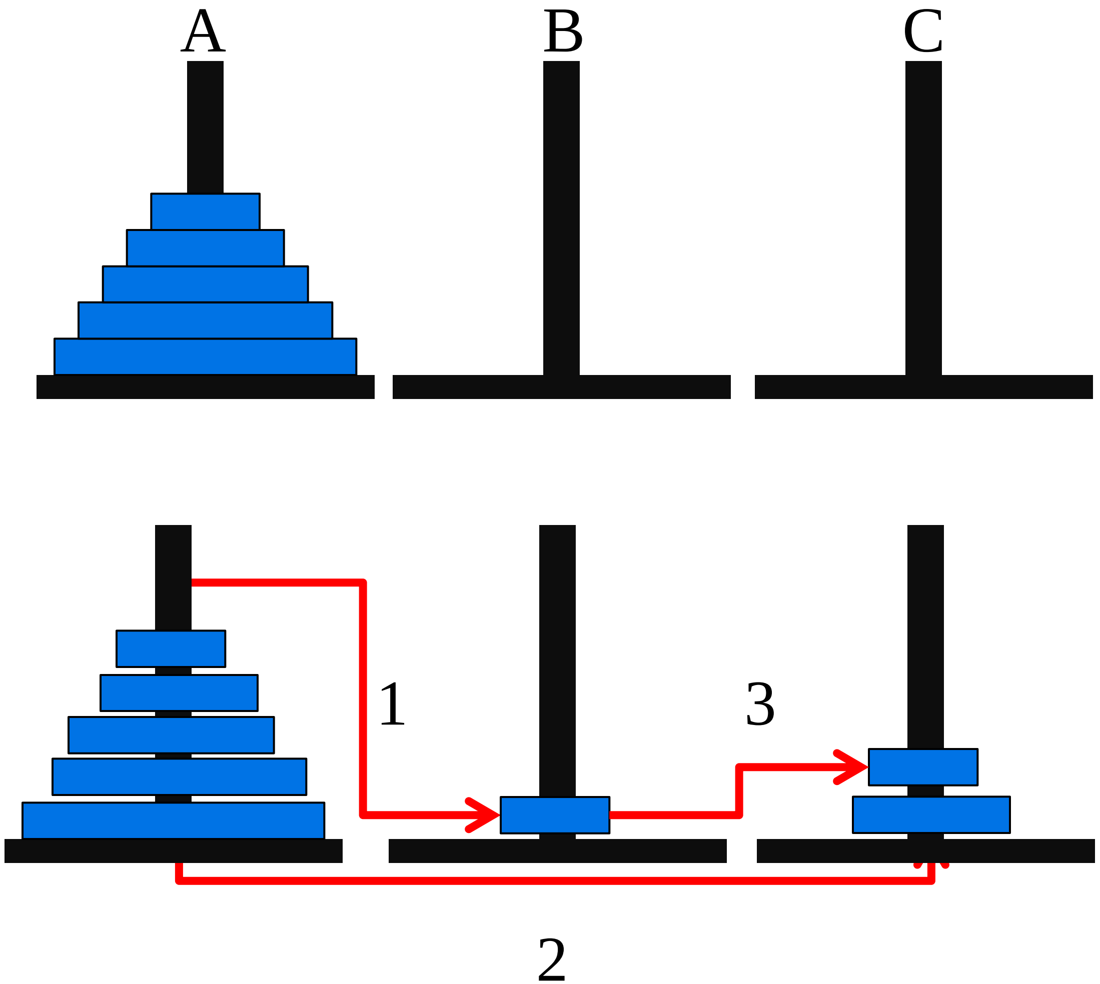
Number of Multiplications required can be represented by the recurrence

$M(n) = M(n - 1) + 1$ for $n > 0$

and $M(0) = 0$

Use Backtracking to solve the recurrence

Tower of Hanoi



- Move n-1 disks recursively from peg1 to peg2 keeping peg3 as auxiliary
- Move the largest disk directly from peg1 to peg3
- Move the remaining n-1 disks recursively from peg2 to peg3 by keeping peg1 as auxiliary
- Number of movement of disks can be represented as a recurrence equation.

- $M(n) = M(n-1) + 1 + M(n-1)$ for $n > 1$
- i.e., $M(n) = 2\,M(n-1) + 1$ for $n > 1$

  and $M(1) = 1$ € $\Theta(2^n)$

The End

Thank You