RAMAIAH
Institute of Technology

DESIGN AND ANALYSIS OF ALGORITHMS - CI43

Prepared by
Ms. Kavya Natikar

UNIT - 1

Asymptotic Bounds and Representation Problems of Algorithms

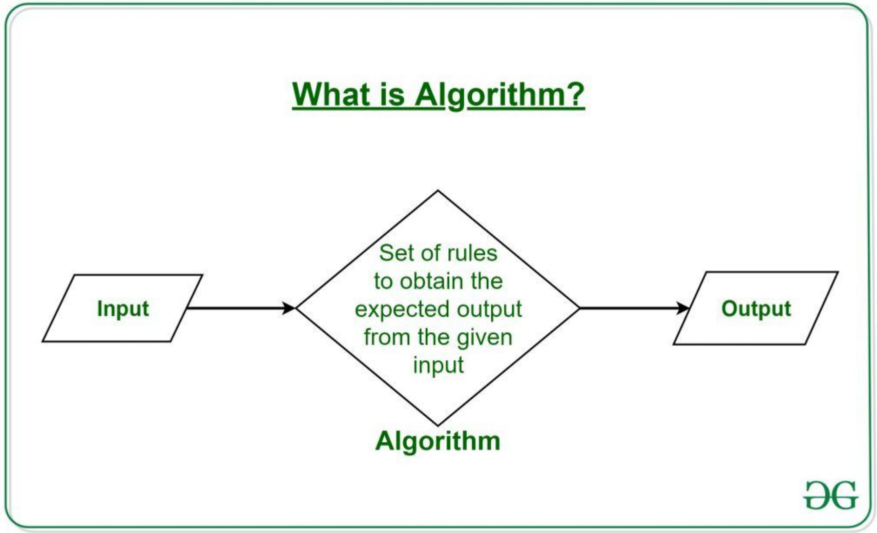
UNIT - 1

Asymptotic Bounds and Representation Problems of Algorithms: Computational Tractability: Some Initial Attempts at Defining Efficiency, Worst-Case Running Times and Brute-Force Search, Polynomial Time as a Definition of Efficiency, Asymptotic Order of Growth: Properties of Asymptotic Growth Rates, Asymptotic Bounds for Some Common Functions, A Survey of Common Running Times: Linear Time, O(n log n) Time, O(nk) Time, Beyond Polynomial Time, Recurrence Relations Substitution Method **Some Representative Problems**: A First Problem: Stable Matching.

(Textbook 1- 2.1,2.2,2.4,1.1)

Algorithm

- Algorithm is a **step-by-step** procedure for solving a problem.
- It is a set of **well-defined instructions** for performing a specific computational task.
- Understanding algorithms is essential for anyone interested in mastering data structures and algorithms.


What is Algorithm?
Input
Set of rules to obtain the expected output from the given input
Output
Algorithm

What is an Algorithm?

An algorithm is a finite sequence of well-defined instructions that can be used to solve a **computational problem.** It provides a step-by-step procedure that convert an input into a desired output.

Algorithms typically follow a logical structure:

Input: The algorithm receives input data.

Processing: The algorithm performs a series of operations on the input data.

Output: The algorithm produces the desired output.

Example: Algorithm for Adding Two Numbers

- Problem Statement: Add two numbers and display the result.
- Algorithm:

  a)Start

  b)Take two numbers as input (A and B)

  c)Compute the sum: Sum = A + B

- Print the result
- Stop

Types of Algorithms

- **Brute Force Algorithm** – Tries all possible solutions (e.g., Linear Search).
- **Divide and Conquer** – Breaks the problem into smaller subproblems and solves them recursively (e.g., Merge Sort, Quick Sort).
- **Greedy Algorithm** – Makes the best local choice at each step (e.g., Dijkstra’s Shortest Path).
- **Dynamic Programming** – Solves problems by breaking them into overlapping subproblems (e.g., Fibonacci using memoization).
- **Backtracking** – Explores possible solutions and backtracks when a solution fails (e.g., N-Queens Problem).

Analysis of Algorithms

- Analysis of Algorithms is the process of **evaluating the efficiency of algorithms**, focusing mainly on the **time and space complexity.**
- It's an investigation of an algorithm's efficiency with respect to 2 resources: **running time and memory space.**
- This helps in evaluating how the algorithm's running time or space requirements grow as the **size of input** increases.
- Algorithm analysis is an important part of computational complexity theory, which provides **theoretical estimation** for the required resources of an algorithm to solve a specific computational problem.

Why Analysis of Algorithms is important?

- To predict the behavior of an algorithm for **large inputs**
- It is much more convenient to have simple measures for the efficiency of an algorithm than to **implement the algorithm** and test the efficiency every time a certain parameter in the underlying computer system changes.
- More importantly, by analyzing different algorithms, we can **compare them to determine** the best one for our purpose.

Given 2 algorithms for a task, how do we find out which one is better?

One way of doing this is – to implement both the algorithms and run the two programs on your computer for different inputs and see which one **takes less time**. There are many problems with this approach for the analysis of algorithms.

- It might be possible that for some inputs, the first algorithm performs better than the second and for some inputs second performs better.
- It might also be possible that for some inputs, the first algorithm performs better on one machine, and the second works better on another machine for some other inputs

Key Focus Areas:

- **Asymptotic Bounds** – Theoretical performance limits of algorithms.
- **Representation of Algorithms** – How we express algorithms mathematically or visually.
- **Computational Tractability** – Understanding which problems can be solved efficiently.

Computational Tractability

- A major focus is to find **efficient algorithms** for computational problems.
- **Stable Matching Problem**, they will involve an implicit search over a large set of combinatorial possibilities and the goal will be to efficiently find a solution that satisfies certain clearly delineated conditions.
- **Computational tractability** refers to whether a problem can be solved efficiently using an algorithm that runs in **polynomial time** (O(n^k) for some constant k).

Some Initial Attempts at Defining Efficiency

A first attempt at a working definition of efficiency is the following.

Proposed Definition of Efficiency (1): An algorithm is efficient if, when implemented, it runs quickly on real input instances.

- The first is the omission of **where,** and **how well**, we implement an algorithm.
- Even bad algorithms can run quickly when applied to small test cases on extremely **fast processors**; even good algorithms can run slowly when they are **coded sloppily**.

- we could ask for is a concrete definition of efficiency that **platform-independent, instance-independent, and of predictive value** with respect to increasing input sizes.
- We can use the **Stable Matching Problem** as an example to guide
- The input has a natural “size” parameter **N**
- we could take this to be the total size of the representation of all preference lists, since this is what **any algorithm for the problem** will receive as input.

- **N** is closely related to the other **natural parameter** in this problem: **n**, the number of men and the number of women.
- Since there are **2n preference lists**, each of length n, we can view N = 2n^2, suppressing more fine-grained details of how the data is represented.
- In considering the problem, we will seek to describe an algorithm at a high level, and then analyze its running time **mathematically as a function** of this input size N.

1. Worst Case Analysis (Mostly used)

The **worst-case running time** of an algorithm is the maximum time it takes for any input of size **n**.

Examples:

- **Linear Search**: Worst-case time is **O(n)** when the target element is at the end of the list.
- **Binary Search**: Worst-case time is **O(log n)** when the element is not in the list.
- **Brute-Force Search** algorithms try all possible solutions and are typically inefficient. Ex: **Traveling Salesman Problem (TSP)**: Checking all possible routes requires **factorial time** O(n!). **String Matching**: Checking every position of a text for a pattern requires **quadratic time** O(nm)**.**

1. Worst Case Analysis (Mostly used)

- In the worst-case analysis, we calculate the **upper bound** on the running time of an algorithm. We must know the case that causes a **maximum number of operations** to be executed.
- For **Linear Search**, the worst case happens when the element to be searched **(x) is not present in the array**. When x is not present, the search() function compares it with all the elements of arr[] **one by one**.
- This is the **most commonly used analysis** of algorithms. Most of the time we consider the case that causes maximum operations.

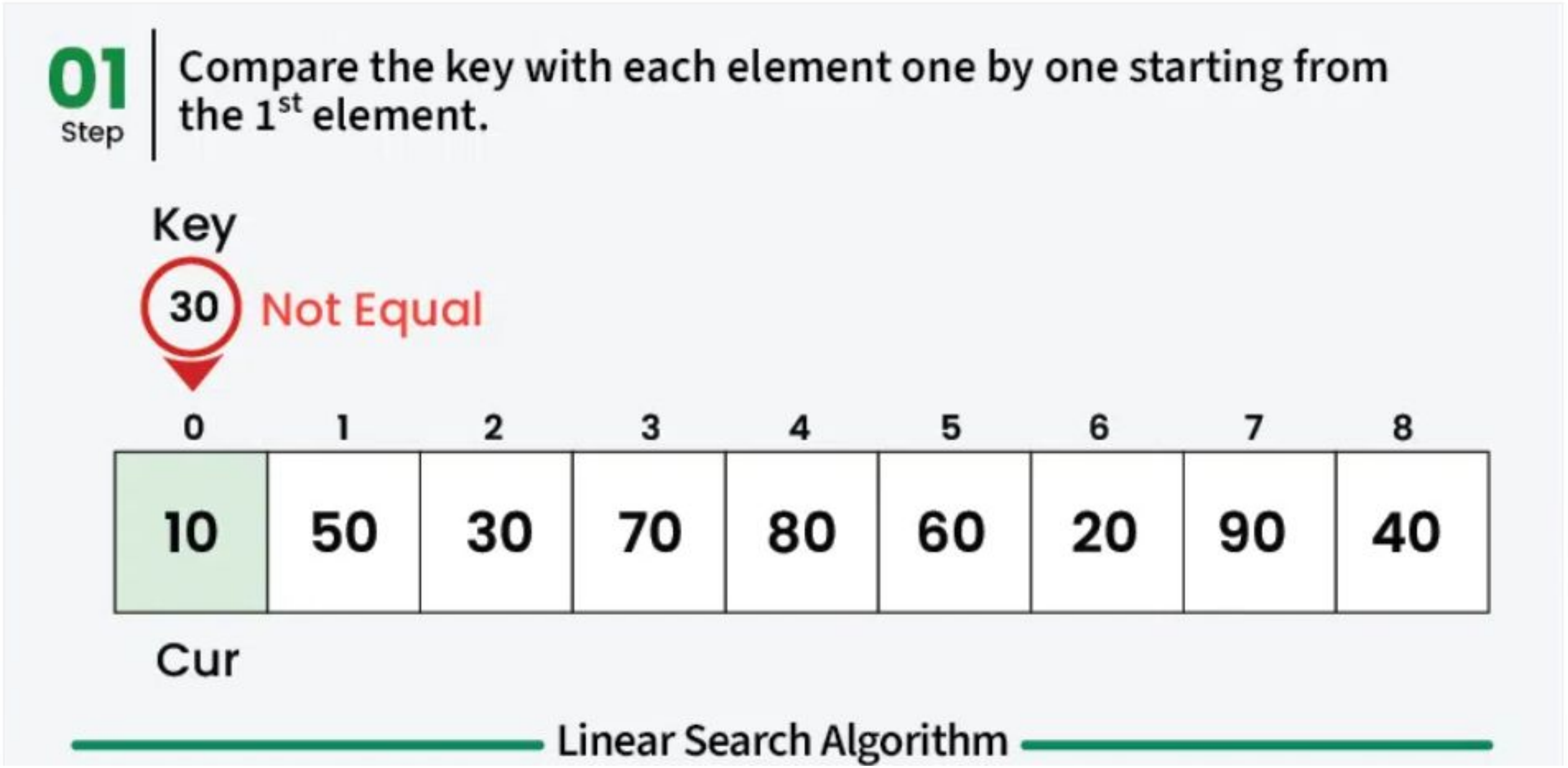
2. Best Case Analysis (Very Rarely used)

- In the best-case analysis, we calculate the **lower bound** on the running time of an algorithm. We must know the case that causes a **minimum number of operations** to be executed.
- For **linear search**, the best case occurs when **x** is present at the **first location**. The number of operations in the best case is constant (not dependent on n). So the order of growth of time taken in terms of input size is constant O(n).

3. Average Case Analysis (Rarely used)

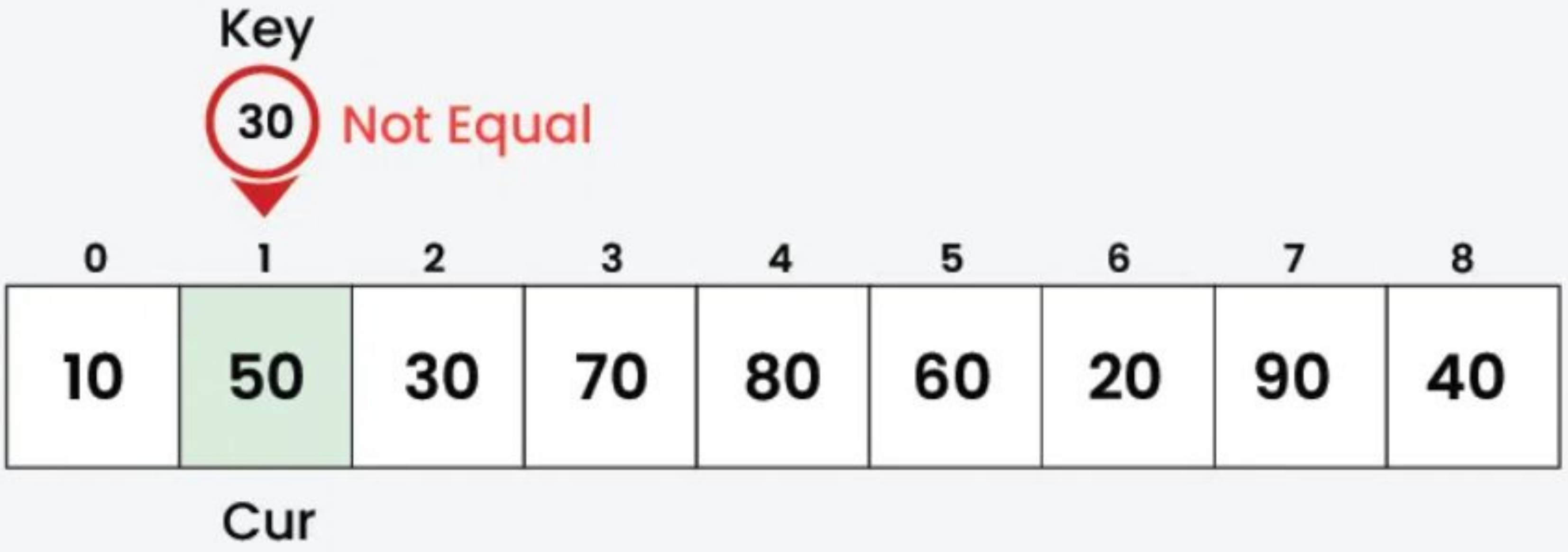
- In average case analysis, we take all possible **inputs and calculate** the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs.
- We must know (or predict) the distribution of cases. For the **linear search problem**, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So we sum all the cases and divide the sum by **(n+1).** We take (n+1) to consider the case when the element is not present.

For example: Consider the array **arr[] = {10, 50, 30, 70, 80, 20, 90, 40}** and **key** = 30


01
Step
Compare the key with each element one by one starting from the 1st element.
Key
30
Not Equal
0
1
2
3
4
5
6
7
8
10
50
30
70
80
60
20
90
40
Cur
Linear Search Algorithm

02 Step

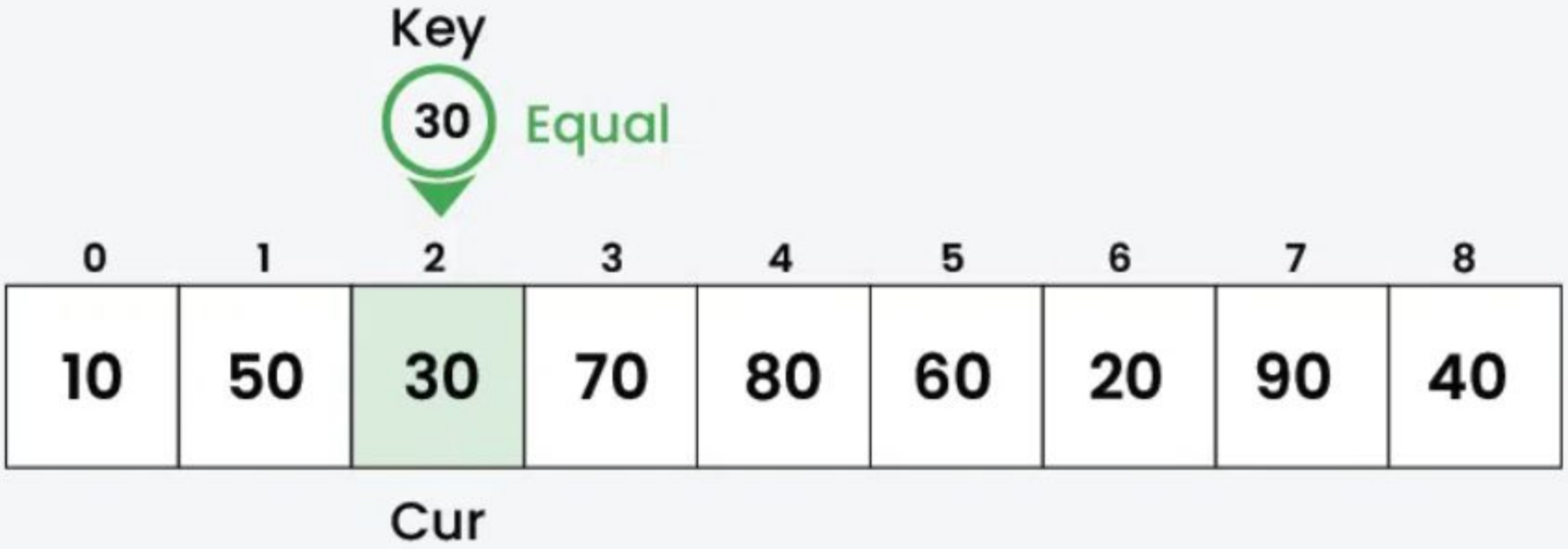
Compare the key with 2nd element which is not equal to the key, so move to the next element



Linear Search Algorithm

03 Step

Compare the key with the 3rd element. Key is found so stop the search.



Linear Search Algorithm

Time and Space Complexity of Linear Search Algorithm:

Time Complexity:

- **Best Case:** In the best case, the key might be present at the first index. So the best case complexity is O(1)
- **Worst Case:** In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is O(N) where N is the size of the list.
- **Average Case:** O(N)

Advantages of Linear Search Algorithm:

- Linear search can be used irrespective of whether the **array is sorted or not**. It can be used on arrays of any data type.
- Does not require any additional **memory**.
- It is a well-suited algorithm for **small datasets.**

Disadvantages of Linear Search Algorithm:

- Linear search has a time complexity of **O(N)**, which in turn makes it **slow for large datasets**.
- Not suitable for **large arrays.**

Worst-Case Running Times and Brute-Force Search

- It emphasizes that the goal is to **determine the maximum possible running time for an algorithm over all inputs of a given size (N)**, and how this time scales as N increases.
- This discusses the **focus on worst-case running time when analyzing algorithms.**
- Since real-world inputs are rarely random, average-case analysis may not always offer meaningful insights into an algorithm's **true performance.**

- One possible approach is to count the number of times each of the algorithm’s operations is **executed.**
- The thing to do is to identify the most important operation of the algorithm, called the **basic operation,** the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.
- It is reasonable to measure an **algorithm’s efficiency** as a function of a parameter indicating the **size** of the algorithm’s input.

Proposed Definition of Efficiency (2): An algorithm is efficient if it achieves qualitatively better worst-case performance, at an analytical level, than brute-force search.

- Algorithms that improve substantially on brute-force search nearly always contain a **valuable heuristic idea** that makes them work
- It will tell us something about the intrinsic structure, and **computational tractability**, of the underlying problem itself.

Polynomial Time as a Definition of Efficiency

- Search spaces for natural combinatorial problems tend to grow exponentially in the **size N of the input; if the input size increases by one, the number of possibilities increases multiplicatively.**
- For now, we will remain deliberately vague on what we mean by the **notion of a “primitive computational step”**— but it can be easily formalized in a model where each step corresponds to a single assembly-language instruction on a standard processor, or one line of a standard programming language such as **C or Java.**

- In any case, if this running-time bound holds, for some **c and d**, then we say that the algorithm has a **polynomial running time,** or that it is a **polynomial-time algorithm.**
- Note that any polynomial-time bound has the **scaling property** we're looking for. If the input size increases **from N to 2N**, the bound on the running time increases from cN^d to $c(2N)^d = c \cdot 2^d N^d$

  which is a slow-down by a factor of 2d.

- Since **d** is a constant, lower-degree polynomials exhibit better scaling behavior than higher-degree polynomials.

Proposed Definition of Efficiency (3): An algorithm is efficient if it has a **polynomial running time.**

- Problems for which polynomial-time algorithms exist almost invariably turn out to have algorithms with **running times proportional to very moderately** growing polynomials like **n, n log n, n2, or n3**.
- Conversely, problems for which no polynomial-time algorithm is known tend to be very difficult in practice.

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

- One further reason why the **mathematical formalism and the empirical evidence** seem to line up well in case of polynomial-time solvability is that the gulf between the growth rates of polynomial and exponential functions is **enormous**.
- Suppose, for example, that we have a **processor that executes a million high-level instructions per second**, and we have algorithms with running-time bounds of n, n log2 n, n2, n3, 1.5n, 2n, and n!.
- In Table 2.1, we show the running times of these algorithms (in seconds, minutes, days, or years) for inputs of size n = 10, 30, 50, 100, 1,000, 10,000, 100,000, and 1,000,000.

Measuring an Input’s Size

- **Investigate an algorithm’s efficiency a**s a function of some parameter n indicating the algorithm’s input size.
- Measuring input size for algorithms solving problems such as checking primality of a positive integer **n**. **(n = input size)**
- It is preferable to measure size by the number **b of bits** in the n’s binary representation:

$$b = \lfloor \log_2 n \rfloor + 1.$$

- This metric usually gives a better idea about the efficiency of algorithms

Asymptotic Order of Growth

- Algorithm's worst-case running time on **inputs of size n grows** at a rate that is at most proportional to some f**unction f(n).**
- The **function f(n)** then becomes a bound on running time of algorithm.
- When we provide a bound on the running time of an algorithm, we will generally be counting the **number of such pseudo-code steps** that are executed
- As just discussed, we will generally be **counting steps** in a pseudo-code specification of an algorithm that resembles a high-level programming language.

Asymptotic Analysis

- Asymptotic Analysis is the big idea that handles the above issues in **analyzing algorithms**.
- In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of **input size** (we don’t measure the actual running time).
- We calculate, **order of growth of time taken** (or space) by an algorithm in terms of input size.
- For example linear search grows **linearly** and **Binary Search** grows **logarithmically** in terms of input size.

O, Ω, and Θ (“big oh”),(“big omega”),(“big theta”)

- Borrowed from mathematics, these notations have become the language for discussing the **efficiency of algorithms.**
- To express the growth rate of running times and other functions.
- **Time efficiency**, also called time complexity, indicates how fast an algorithm in question runs.
- **Space efficiency**, also called space complexity, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output.

Asymptotic Upper Bounds (Big-O Notation - Worst case)

- Let T(n) be a function—say, the **worst-case** running time of a certain algorithm on an input of size **n.**
- We will also sometimes write this as **T(n) = O(f(n))**. In this case, we will say that T is asymptotically upper bounded by f.
- It is important to note that this definition requires a constant c to exist that works for all n; in particular, c cannot depend on n. **O(·) expresses only an upper bound**, not the exact growth rate of the function.
- **Ex:** Bubble Sort – $O(n^2)$

Asymptotic Lower Bounds (Big-Ω Notation - Best case)

- To do this, we want to express the notion that for arbitrarily large input sizes n, the **function T(n)** is at least a constant multiple of some specific function **f(n)**. Thus, we say that

$$T(n) \text{ is } \Omega(f(n)) \text{ (also written } T(n) = \Omega(f(n)))$$

- By analogy with **O(·) notation**, we will refer to T in this case as being asymptotically **lower- bounded by f**. Again, note that the constant must be fixed, independent of n.
- **Ex**: Insertion sort

Asymptotically Tight Bounds (Big Θ Notation - Average case)

- If we can show that a running time **T(n)** is both **O(f(n))** and also **(f(n))**
- There is notation to express this: if function T(n) is both O(f(n)) and $\Omega(f(n))$
- we say that **f(n)** is an asymptotically tight bound for **T(n)**. Essentially, if the ratio of functions f(n) and g(n) converges to a positive constant as n goes to infinity, then

we say that $T(n)$ is $\Theta(f(n))$ $\qquad$ $f(n) = \Theta(g(n))$

- **Ex:** Merge Sort – Θ(n log n)

Time Complexity Table

Algorithm	Best Case	Average Case	Worst Case	Best for
Linear Search	O(1)	O(n)	O(n)	Small or unordered datasets
Binary Search	O(1)	O(logn)	O(logn)	Sorted datasets
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	Nearly sorted data
Merge Sort	O(n log n)	O(n log n)	O(n log n)	Large datasets
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	General-purpose sorting
Dijkstra’s Algorithm	O(1)	O((V+E)logV)	O(V2)	Shortest path problems

Common Time Complexities Calculations:

1. O(1) → Constant Time

- ✅ **Happens when:**
- No loops, just direct operations.
- Example: Accessing an element in an array.

🔹 **Shortcut:** If the number of operations **doesn't depend on n**, it's **O(1)**.

Ex:

```
#include <stdio.h>
void getFirstElement(int arr[]) {
    printf("%d", arr[0]);  // Always takes constant time
}
```

2.O(n) → Linear Time

- ✅ **Happens when:**
- One loop running n times.

🔹 Shortcut: Single loop → **O(n).**

Ex:

```
#include <stdio.h>
void printArray(int arr[], int n) {
      for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);  // Runs n times
      }
}
```

3.O(n^2) → Quadratic Time

✅ Happens when:

Two nested loops, each running n times.

- 🔹 Shortcut: Nested loops → Multiply complexities (O(n) × O(n) = O(n^2)).

Ex:

```
#include <stdio.h>
void printPairs(int arr[], int n) {
    for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        printf("(%d, %d) ", arr[i], arr[j]);  // Runs n * n times
    }
    }
}
```

4.O(log n) → Logarithmic Time

✅ Happens when:

The input shrinks by half in each step (e.g., Binary Search).

🔹 Shortcut: If n is divided repeatedly → O(log n).

EX:

```
#include <stdio.h>
int binarySearch(int arr[], int left, int right, int key) {
    while (left <= right) {
    int mid = left + (right - left) / 2;
    if (arr[mid] == key) return mid;
    else if (arr[mid] < key) left = mid + 1;
    else right = mid - 1;
```

5.O(n log n) → Efficient Sorting

✅ Happens when:

Sorting algorithms like Merge Sort, QuickSort.

These divide the problem (log n) and do work (n) → O(n log n).

🔹 Shortcut: Divide & Conquer sorting → O(n log n).

EX:

```
#include <stdio.h>
void mergeSort(int arr[], int l, int r) {
    if (l >= r) return;
    int mid = l + (r - l) / 2;
    mergeSort(arr, l, mid);
    mergeSort(arr, mid + 1, r);
    // Assume merge() function exists
```

6.O(2^n) → Exponential Time

✅ Happens when:

Recursive calls double in each step (e.g., Fibonacci).

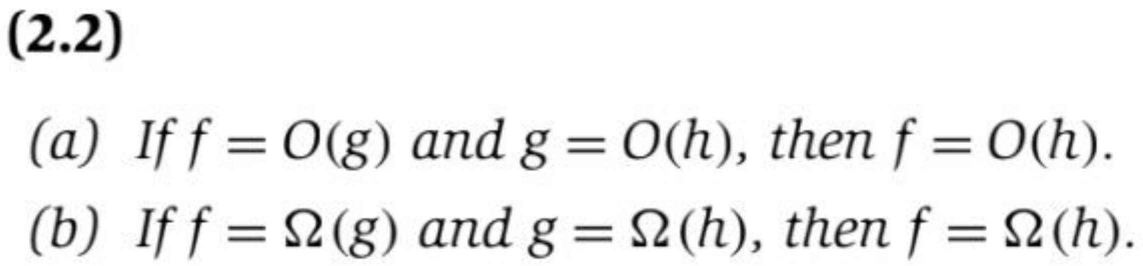
🔹 Shortcut: If each call makes 2 recursive calls → O(2^n).

EX:

```
#include <stdio.h>
int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

Properties of Asymptotic Growth Rates

Transitivity: if a function **f** is asymptotically upper-bounded by a function **g**, and if **g** in turn is asymptotically upper - bounded by a function **h, then f** is asymptotically upper-bounded by **h**. **A similar property holds for lower bounds.** We write this more precisely as follows.



(2.2)

(a) *If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.*

(b) *If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.*

Proof. We'll prove part (a) of this claim; the proof of part (b) is very similar.

For (a), we're given that for some constants c and n_0, we have $f(n) \leq cg(n)$ for all $n \geq n_0$. Also, for some (potentially different) constants c' and n'_0, we have $g(n) \leq c'h(n)$ for all $n \geq n'_0$. So consider any number n that is at least as large as both n_0 and n'_0. We have $f(n) \leq cg(n) \leq cc'h(n)$, and so $f(n) \leq cc'h(n)$ for all $n \geq \max(n_0, n'_0)$. This latter inequality is exactly what is required for showing that $f = O(h)$. ■

Combining parts (a) and (b) of (2.2), we can obtain a similar result for asymptotically tight bounds. Suppose we know that $f = \Theta(g)$ and that $g = \Theta(h)$. Then since $f = O(g)$ and $g = O(h)$, we know from part (a) that $f = O(h)$; since $f = \Omega(g)$ and $g = \Omega(h)$, we know from part (b) that $f = \Omega(h)$. It follows that $f = \Theta(h)$. Thus we have shown

(2.3) *If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$.*

Sums of Functions It is also useful to have results that quantify the effect of adding two functions. First, if we have an asymptotic upper bound that applies to each of two functions f and g, then it applies to their sum.

(2.4) *Suppose that f and g are two functions such that for some other function h, we have $f = O(h)$ and $g = O(h)$. Then $f + g = O(h)$.*

Proof. We're given that for some constants c and n_0, we have $f(n) \leq ch(n)$ for all $n \geq n_0$. Also, for some (potentially different) constants c' and n'_0, we have $g(n) \leq c'h(n)$ for all $n \geq n'_0$. So consider any number n that is at least as large as both n_0 and n'_0. We have $f(n) + g(n) \leq ch(n) + c'h(n)$. Thus $f(n) + g(n) \leq (c + c')h(n)$ for all $n \geq \max(n_0, n'_0)$, which is exactly what is required for showing that $f + g = O(h)$. ■

(2.5) *Let k be a fixed constant, and let $f_1, f_2, \ldots, f_k$ and h be functions such that $f_i = O(h)$ for all i. Then $f_1 + f_2 + \cdots + f_k = O(h)$.*

There is also a consequence of (2.4) that covers the following kind of situation. It frequently happens that we're analyzing an algorithm with two high-level parts, and it is easy to show that one of the two parts is slower than the other. We'd like to be able to say that the running time of the whole algorithm is asymptotically comparable to the running time of the slow part. Since the overall running time is a sum of two functions (the running times of the two parts), results on asymptotic bounds for sums of functions are directly relevant.

(2.6) *Suppose that f and g are two functions (taking nonnegative values) such that $g = O(f)$. Then $f + g = \Theta(f)$. In other words, f is an asymptotically tight bound for the combined function $f + g$.*

Proof. Clearly $f + g = \Omega(f)$, since for all $n \geq 0$, we have $f(n) + g(n) \geq f(n)$. So to complete the proof, we need to show that $f + g = O(f)$.

But this is a direct consequence of (2.4): we're given the fact that $g = O(f)$, and also $f = O(f)$ holds for any function, so by (2.4) we have $f + g = O(f)$. ■

This result also extends to the sum of any fixed, constant number of functions: the most rapidly growing among the functions is an asymptotically tight bound for the sum.

Asymptotic Bounds for Some Common Functions

Polynomials Recall that a polynomial is a function that can be written in the form $f(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d$ for some integer constant $d > 0$, where the final coefficient a_d is nonzero. This value d is called the *degree* of the polynomial. For example, the functions of the form $pn^2 + qn + r$ (with $p \neq 0$) that we considered earlier are polynomials of degree 2.

A basic fact about polynomials is that their asymptotic rate of growth is determined by their "high-order term"—the one that determines the degree. We state this more formally in the following claim. Since we are concerned here only with functions that take nonnegative values, we will restrict our attention to polynomials for which the high-order term has a positive coefficient $a_d > 0$.

(2.7) *Let f be a polynomial of degree d, in which the coefficient a_d is positive. Then $f = O(n^d)$.*

Proof. We write $f = a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d$, where $a_d > 0$. The upper bound is a direct application of (2.5). First, notice that coefficients a_j for $j < d$ may be negative, but in any case we have $a_j n^j \le |a_j| n^d$ for all $n \ge 1$. Thus each term in the polynomial is $O(n^d)$. Since f is a sum of a constant number of functions, each of which is $O(n^d)$, it follows from (2.5) that f is $O(n^d)$. ∎

One can also show that under the conditions of (2.7), we have $f = \Omega(n^d)$, and hence it follows that in fact $f = \Theta(n^d)$.

- This is a good point at which to discuss the relationship between these types of **asymptotic bounds and the notion of polynomial time**, which we arrived at in the previous section of efficiency.
- Using **O(·) notation**, it’s easy to formally define polynomial time: a polynomial-time algorithm is one whose running time T(n) is O(nd) for some constant d, where d is independent of the input size. So, algorithms with running-time bounds like **O(n2) and O(n3)** are polynomial-time algorithms.

Logarithms

- Recall that **logb** n is the number **x** such that **bx = n**. One way to get an approximate sense of how fast **logb n** grows is to note that, if we round it down to the nearest integer, it is one less than the number of digits in the base-b representation of the number **n**.
- So logarithms are **very slowly growing functions**. In particular, for every **base b**, the function **logb** n is asymptotically bounded by every function of the form **nx**, even for (noninteger) values of x arbitrary close to **0**.

(2.8) *For every $b > 1$ and every $x > 0$, we have $\log_b n = O(n^x)$.*

One can directly translate between logarithms of different bases using the following fundamental identity:

$$\log_a n = \frac{\log_b n}{\log_b a}.$$

This equation explains why you'll often notice people writing bounds like $O(\log n)$ without indicating the base of the logarithm. This is not sloppy usage: the identity above says that $\log_a n = \frac{1}{\log_b a} \cdot \log_b n$, so the point is that $\log_a n = \Theta(\log_b n)$, and the base of the logarithm is not important when writing bounds using asymptotic notation.

Exponentials

- Exponential functions are functions of the form **f(n) = rn** for some constant **base r**. Here we will be concerned with the case in which **r > 1**, which results in a very fast-growing function.
- In particular, where polynomials raise n to a fixed exponent, exponentials raise a fixed number to **n** as a power; this leads to much faster rates of growth.
- One way to summarize the relationship between polynomials and exponentials is as follows.

(2.9) *For every* $r > 1$ *and every* $d > 0$, *we have* $n^d = O(r^n)$.

- Taken together, then, l**ogarithms, polynomials, and exponentials** serve as useful landmarks in the range of possible functions that you encounter when **analyzing running times.**
- Logarithms grow more slowly than polynomials, and polynomials grow more slowly than exponentials.

A Survey of Common Running Times

we noted that a unifying theme in algorithm design is the search for algorithms whose performance is more efficient than a brute-force enumeration of this search space.

2 kinds of bounds:

1. the running time you hope to achieve
2. the size of the problem’s natural search space

Linear Time - O(n)

An algorithm that runs in **O(n)**, its running time is at most a constant factor times the **size of the input**. process the input in a single pass, spending a constant amount of time on each item of input encountered.

Computing the maximum of n numbers,

Suppose the numbers are provided as input in either a list or an array. We process the numbers **a1, a2,..., an**, keeping a running estimate of the maximum as we go.

Each time we encounter a number **ai**, we check whether **ai** is larger than our **current estimate,** and if so we update the estimate to **ai.**

```
$max = a_1$
For $i = 2$ to $n$
  If $a_i > max$ then
    set $max = a_i$
  Endif
Endfor
```

In this way, we do constant work per element, for a total running time of **O(n).**

Merging Two Sorted Lists

Often, an algorithm has a running time of **O(n),** but the reason is more complex.

- Suppose we are given two lists of n numbers each, **a1, a2,..., an** and **b1, b2,..., bn**, and each is already arranged in ascending order.
- We'd like to merge these into a single list **c1, c2,..., c2n** that is also arranged in ascending order.
- **For example**, merging the lists List 1: **[2, 5, 8]** List 2: **[1, 3, 7]** results in the output is **[1, 2, 3, 5, 7, 8]**

Understanding the Merging Process

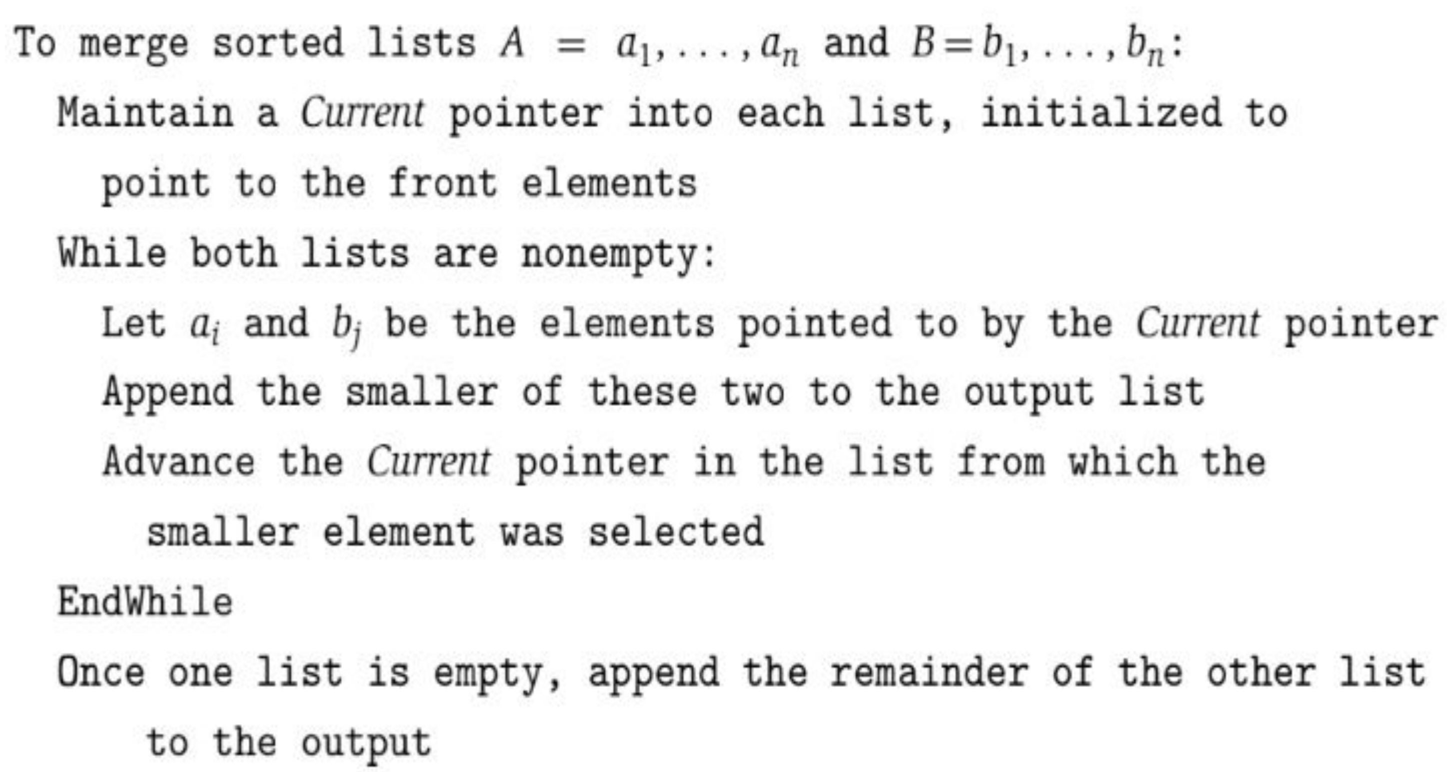
Imagine you have **two sorted lists** and you want to merge into one **sorted** list.

Ex : 2, 3, 11, 19 and 4, 9, 16, 25 results in the output ?.

Step-by-step approach:

1. **Start with the first element** of both lists.
2. **Compare** the first element of both lists.
3. **Take the smaller** element and place it into the new (merged) list.
4. **Remove** the selected element from its original list.
5. **Repeat** steps 2–4 until one of the lists is empty.
6. **Append the remaining elements** from the non-empty list to the merged list.

Algorithm



```
To merge sorted lists $A = a_1, \ldots, a_n$ and $B = b_1, \ldots, b_n$:
  Maintain a Current pointer into each list, initialized to
    point to the front elements
  While both lists are nonempty:
    Let $a_i$ and $b_j$ be the elements pointed to by the Current pointer
    Append the smaller of these two to the output list
    Advance the Current pointer in the list from which the
      smaller element was selected
  EndWhile
  Once one list is empty, append the remainder of the other list
      to the output
```

Now, it is true that each element can be involved in at most **O(n)**

O(n log n) Time

- It is the running time of any algorithm that **splits its input into two equal-sized pieces, solves each piece recursively, and then combines the two solutions in linear time**.

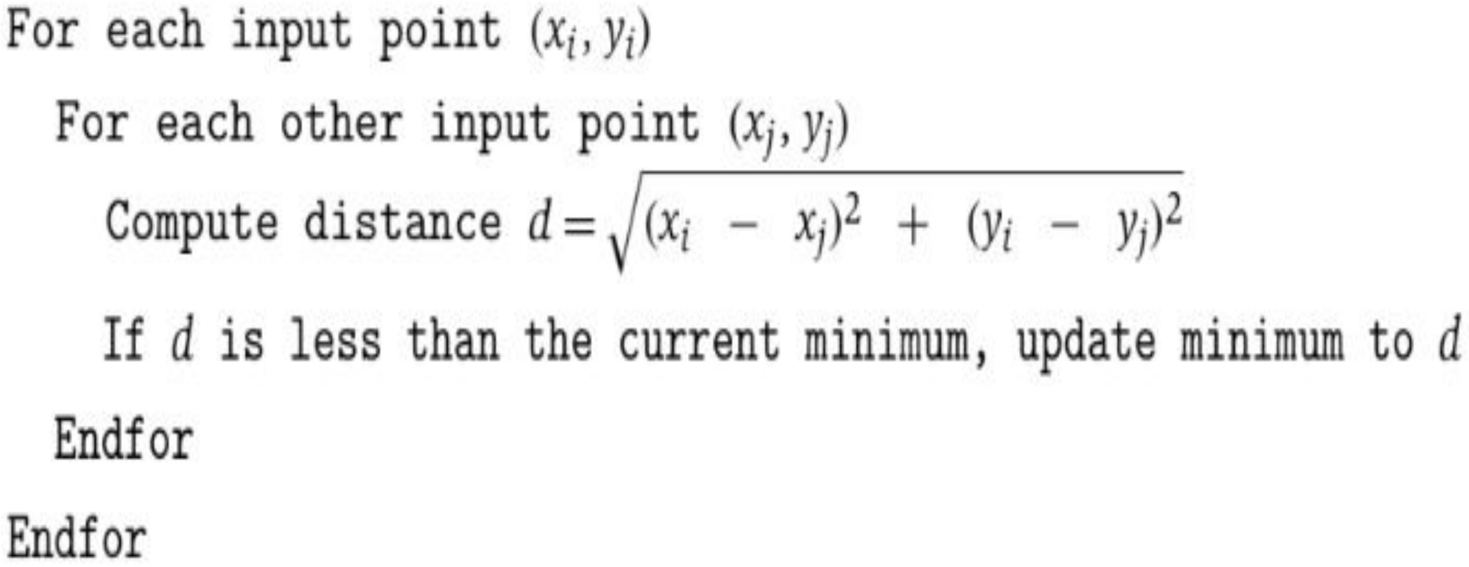
  Ex: Merge sort algorithm

- Note that this algorithm requires O(n log n) time to sort the numbers, and then it spends constant work on each number in ascending order.
- One also frequently encounters **O(n log n)** as a running time simply because there are many algorithms whose most expensive step is to **sort the input.**

Quadratic Time

- Suppose you are given **n points** in the plane, each specified by **(x, y) coordinates**, and you'd like to find the **pair of points** that are **closest together.**
- The distance between points (xi, yi) and (xj, yj) can be computed by the formula $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ in constant time , so the overall running time is **O(n2).**
- Quadratic time also arises naturally from a **pair of nested loops:** An algorithm consists of a loop with **O(n) iterations**, and each iteration of the loop launches an internal loop that **takes O(n) time**. Multiplying these 2 factors of **n** together gives the running time.

The brute-force algorithm for finding the closest pair of points can be written in an equivalent way with 2 nested loops:



```
For each input point $(x_i, y_i)$
  For each other input point $(x_j, y_j)$
    Compute distance $d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$
    If $d$ is less than the current minimum, update minimum to $d$
  Endfor
Endfor
```

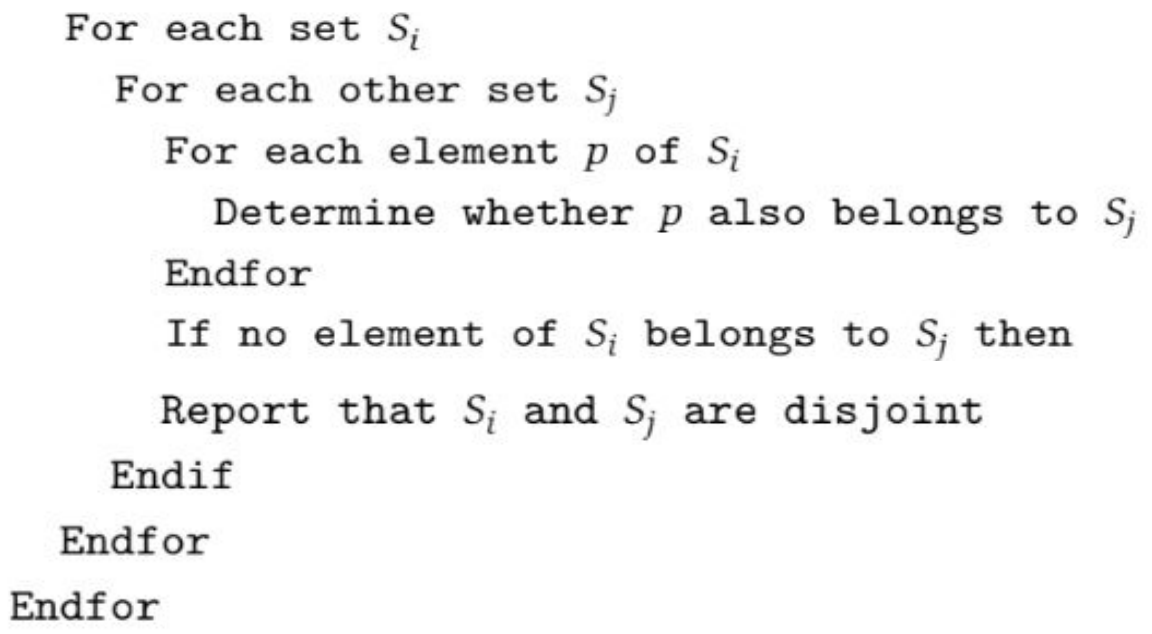
Cubic Time

More elaborate sets of nested loops often lead to algorithms that run in O(n3) time.

Example:

Given sets S1, S2,..., Sn, each of which is a subset of {1, 2, . . . , n}, and we would like to know whether some pair of these sets is disjoint—in other words, has **no elements in common**.

What is the running time needed to solve this problem? Let's suppose that each set Si is represented in such a way that the elements of Si can be listed in constant time per element, and we can also check in constant time whether a given number p belongs to Si.



```
   For each set $S_i$
     For each other set $S_j$
       For each element $p$ of $S_i$
         Determine whether $p$ also belongs to $S_j$
       Endfor
       If no element of $S_i$ belongs to $S_j$ then
       Report that $S_i$ and $S_j$ are disjoint
     Endif
   Endfor
 Endfor
```

Each of the sets has maximum size O(n), so the innermost loop takes time **O(n).** Looping over the sets **Sj** involves **O(n)** iterations around this innermost loop; and looping over the sets **Si** involves **O(n)** iterations around this.

Multiplying these 3 factors of n together, we get the running time of **O(n3)**.

$O(n^k)$ Time

Any algorithm where the number of operations is proportional to **n raised to a constant power (k).**

```
For each subset $S$ of $k$ nodes
  Check whether $S$ constitutes an independent set
  If $S$ is an independent set then
    Stop and declare success
  Endif
Endfor
If no $k$-node independent set was found then
  Declare failure
Endif
```

Beyond Polynomial Time

```
For each subset $S$ of nodes
  Check whether $S$ constitutes an independent set
  If $S$ is a larger independent set than the largest seen so far then
    Record the size of $S$ as the current maximum
  Endif
Endfor
```

The total number of subsets of an n-element set is **2^n**, and so the outer loop in this algorithm will run for **2^n** iterations as it tries all these subsets. Inside the loop, we are checking all pairs from a set S that can be as large as n nodes, so each iteration of the loop takes at most **O(n^2) time**. Multiplying these two together, we get a running time of $O(n^2 2^n)$.

Sublinear Time

Since it takes linear time just to read the input, these situations tend to arise in a model of computation where the input can be “queried” indirectly rather than read completely, and the goal is to minimize the amount of querying that must be done.

Ex: Binary search

- The running time of binary search is **O(log n),** because of this successive **shrinking of the search region.**
- In general, O(log n) arises as a time bound whenever we’re dealing with an algorithm that does a constant amount of work in order to throw away a constant fraction of the input.

Recurrence Relations Substitution Method

Solve the following recurrence relations using substitution method

(i) $x(n)=x(n-1)+5$ for $n>1$, $x(1)=0$

$x(2)=x(1)+5=0+5=5$

$x(3)=x(2)+5=5+5=10$

$x(4)=x(3)+5=10+5=15$

Observing the pattern, we can generalize:

$x(n)=x(1)+5(n-1)=0+5(n-1)=5(n-1)$

Correct formula: $x(n)=5(n-1)$

A First Problem: Stable Matching

The Problem

The Stable Matching Problem originated in **1962**, when David Gale and Lloyd Shapley, two mathematical economists.

Example, that your friend Raj has just accepted a summer job at the large telecommunications company **CluNet**. A few days later, the small start-up company **WebExodus,** which had been dragging its feet on making a few final decisions, calls up Raj and **offers him a summer job as well.** Now, Raj actually prefers **WebExodus** to CluNet.

So, he decides to **switch jobs**. CluNet, now needing a **replacement**, hires someone from its waitlist, who in turn backs out of a job at **Babelsoft**. **This creates a chain reaction where multiple job acceptances and rejections start affecting different companies.**

- This creates instability because companies and applicants keep changing their choices based on new information, leading to **confusion** and **dissatisfaction**.
- Since no one wants to switch, the system is **stable**, meaning no further changes or reassignments will happen.

What is Stable Matching ?

- Problem of finding a stable matching between 2 sets of elements given an ordered set of preferences for an each element
- A mapping from the elements of one set to the elements of the other.
- A matching is stable when the following conditions are both false:

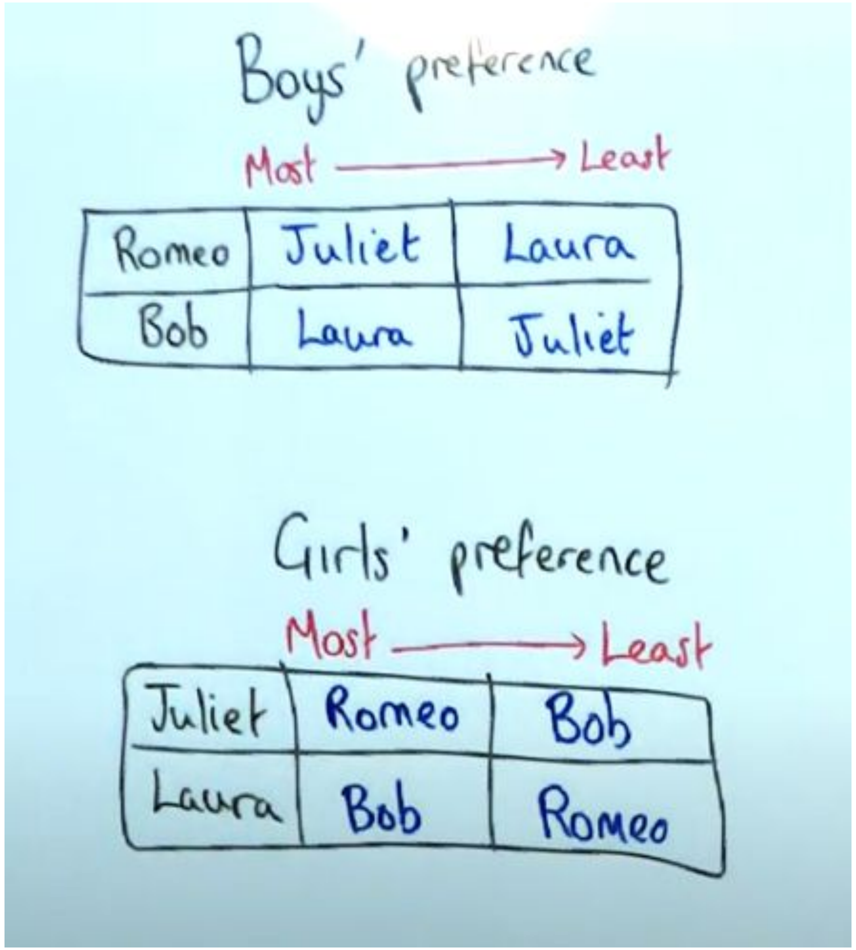
a) An element (A) from the first set prefers an element (B) from the second set over its currently matched pair

b) B also prefer A over its currently matched pair.

The Gale-Shapley algorithm, also called the Stable Matching Algorithm, is used for solving the stable marriage problem.

it applies to many real-world scenarios, such as job hiring, student admissions.

Example: Stable Matching in University Admissions



Boys' preference

Most $\longrightarrow$ Least

Romeo	Juliet	Laura
Bob	Laura	Juliet

Girls' preference

Most $\longrightarrow$ Least

Juliet	Romeo	Bob
Laura	Bob	Romeo

Formulating the Problem

- The problem is simplified by assuming that each applicant applies to every company, and each company selects **only one applicant.** This removes complications like varying numbers of applicants and job slots.
- **Ex:** consider a set **M** = {m1,..., mn} of n men, and a set **W** = {w1,..., wn} of n women. Let M × W denote the set of **all possible ordered pairs of the form (m, w)**, where m ∈ M and w ∈ W. A matching **S is a set of ordered pairs,** each from M × W, with the property that each member of M and each member of W appears in at most one pair in **S.** A perfect matching **S'** is a matching with the property that each member of M and each member of W appears in exactly one pair in **S'.**

There are two pairs **(m, w)** and **(m' , w')** in **S** with the property that **m** prefers **w' to w**, and **w'** prefers **m to m'**. In this case, there's nothing to stop **m and w'** from abandoning their current partners and heading off together; the set of marriages is not self-enforcing. We'll say that such a pair **(m, w')** is an **instability with respect to S**:(m, w') does not belong to **S,** but each of **m** and **w** prefers the other to their partner in **S.**

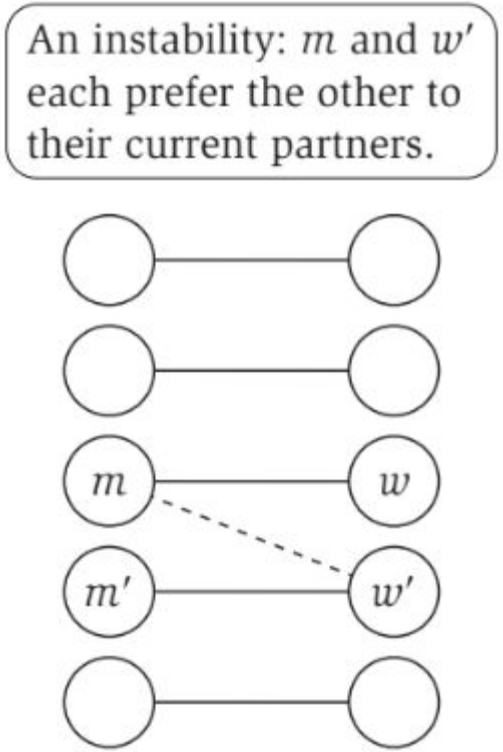

An instability: m and w' each prefer the other to their current partners.
m
w
m'
w'

Figure 1.1 Perfect matching S with instability (m, w').

Our goal, then, is a set of marriages with no instabilities. We'll say that a matching S is stable if (i) it is perfect, and (ii) there is no instability with respect to S.

The two key questions are:

1. **Does a stable matching always exist?** – Yes, there is always at least one stable matching.
2. **Can we efficiently find one?** – Yes, the **Gale-Shapley algorithm** (or Deferred Acceptance algorithm) efficiently finds a stable matching in a step-by-step manner.

Designing the Gale - Shapley Algorithm

we will give an efficient algorithm that takes the preference lists and constructs a **stable matching**.

- Initially, everyone is unmarried. Suppose an unmarried man **m** chooses the woman **w** who ranks highest on his preference list and proposes to her. Can we declare immediately that **(m, w)** will be one of the pairs in our final stable matching? Not necessarily: at some point in the future, a man **m’** whom **w** prefers may propose to her. It would be dangerous for **w** to reject **m** right away; she may never receive a proposal from someone she ranks as highly as **m**. So a natural idea would be to have the pair **(m, w)** enter an **intermediate state engagement.**

- Suppose we are now at a state in which some men and women are free not engaged and some are engaged. An arbitrary free man **m** chooses the highest-ranked woman **w** to whom he has not yet proposed, and he proposes to her. If **w** is also free, then **m** and **w** become engaged. Otherwise, **w** is already engaged to some other man **m’** . In this case, she determines which of **m** or **m’** ranks higher on her preference list; this man becomes engaged to **w** and the other becomes free.
- Finally, the algorithm will terminate when no one is free; at this moment,all engagements are declared final, and the resulting **perfect matching is returned.**

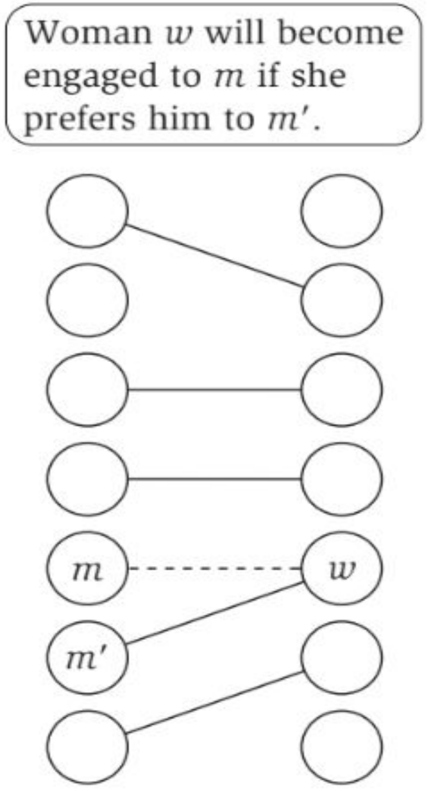

Woman w will become engaged to m if she prefers him to m'.
m
w
m'

Figure 1.2 An intermediate state of the G-S algorithm when a free man m is proposing to a woman w.

The Gale-Shapley algorithm

1. Each proposer proposes to their most preferred receiver who has not yet rejected them.
2. Each receiver evaluates the proposals and:
   a) Accepts the most preferred proposer (tentatively).
   b) Rejects the others.
3. Rejected proposers move to their next preferred choice.
4. The process repeats until all proposers are matched.

Time Complexity: In the worst case, the overall complexity is **O(n2)**, where n is the number of proposers

```
Initially all $m \in M$ and $w \in W$ are free
While there is a man $m$ who is free and hasn't proposed to
every woman
   Choose such a man $m$
   Let $w$ be the highest-ranked woman in $m$'s preference list
      to whom $m$ has not yet proposed
   If $w$ is free then
      $(m, w)$ become engaged
   Else $w$ is currently engaged to $m'$
      If $w$ prefers $m'$ to $m$ then
         $m$ remains free
      Else $w$ prefers $m$ to $m'$
         $(m, w)$ become engaged
         $m'$ becomes free
      Endif
   Endif
Endwhile
Return the set $S$ of engaged pairs
```

Analyzing the Algorithm

First consider the view of a woman **w** during the execution of the algorithm. For a while, no one has proposed to her, and she is free. Then a man **m** may propose to her, and she becomes engaged. As time goes on, she may receive additional proposals, accepting those that increase the rank of her partner. So we discover the following.

(1.1) w remains engaged from the point at which she receives her first proposal; and the sequence of partners to which she is engaged gets better and better (in terms of her preference list).

(1.2) The sequence of women to whom m proposes gets worse and worse (in terms of his preference list).

(1.3) The G-S algorithm terminates after at most n2 iterations of the While loop.

(1.4) If m is free at some point in the execution of the algorithm, then there is a woman to whom he has not yet proposed.

(1.5) The set S returned at termination is a perfect matching.

(1.6) Consider an execution of the G-S algorithm that returns a set of pairs S. The set S is a stable matching.

(1.7) Every execution of the G-S algorithm results in the set S∗.

(1.8) In the stable matching S∗, each woman is paired with her worst valid Partner.

Thus, we find that our simple example above, in which the men’s preferences clashed with the women’s, hinted at a very general phenomenon: for any input, the side that does the proposing in the **G-S algorithm** ends up with the best possible stable matching (from their perspective), while the side that does not do the proposing correspondingly ends up with the worst possible stable matching.

https://youtu.be/uigz45tk2-s?si=r9nbnLLF0qTNOEqL