Introduction to complexity theory

Why Computational Efficiency Still Matters?

- Computers execute tasks extremely fast, but real-world problems can be too large to handle.
- As computers advance, we attempt to solve bigger problems, keeping us limited by capacity.
- Early computers were large, but had far less computational power than modern smartphones.
- Despite advancements, computational and memory constraints remain.

Hence, we need to analyze the **efficiency of our program** using “Complexity Theory”

Importance of Program Efficiency

- Efficient programming is crucial due to limited computational resources.
- **In classroom settings, we almost never encounter this issue** because the inputs to our problems are **<u>quite small.</u>**
- As a result, even an i**nefficient program** can process them **very fast** (in computer terms).
- However, real-world problems involve **<u>massive inputs,</u>** making **<u>efficiency critical</u>**.

- **Poorly optimized programs** can lead to
  - Unacceptable execution times
  - High memory consumption

Example – Matrix Multiplication Complexity

- Matrix multiplication: C = A × B, where C’s elements are computed using row-column multiplication.

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} = \begin{bmatrix} (aj + bm + cp) & (ak + bn + cq) & (al + bo + cr) \\ (dj + em + fp) & (dk + en + fq) & (dl + eo + fr) \\ (gj + hm + ip) & (gk + hn + iq) & (gl + ho + ir) \end{bmatrix}$$

Example – Matrix Multiplication Complexity

- Matrix multiplication: C = A × B, where C's elements are computed using row-column multiplication.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} = \begin{bmatrix} (aj+bm+cp) & (ak+bn+cq) & (al+bo+cr) \\ (dj+em+fp) & (dk+en+fq) & (dl+eo+fr) \\ (gj+hm+ip) & (gk+hn+iq) & (gl+ho+ir) \end{bmatrix}$$

- If A and B are 3 × 3 matrices:
  - Each C element requires 3 multiplications, C has 9 elements
  - Total multiplications = 3 × 9 = 27
  - At 2 GHz CPU speed (~1 operation per nanosecond), takes ~27 nanoseconds.
  - 27 × 10^(-9) seconds = 27 nanoseconds.

Example – Matrix Multiplication Complexity

- Matrix multiplication: C = A × B, where C’s elements are computed using row-column multiplication.

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

- If A and B are 10,000 × 10,000 matrices:
  - Each C element requires 10,000 multiplications.
  - Total multiplications = 10^4×10^4×10^4=10^12
  - At 2 GHz CPU speed (~1 operation per nanosecond), takes **~16.67 minutes**.
    - 1012* 10^-9 seconds = 1000 seconds =16.67 minutes
- <u>Larger matrices drastically increase execution time.</u>

Example – Memory Consumption Issue

- **Memory efficiency is crucial** to avoid excessive resource usage.

- **For example:**
    - Bangladesh NID database has **50 million entries** (~5×10^7).
    - **Each name: 20 characters** → **100 MB total** (5 X 10^7 X 20=10^9 bytes).
    - **If additional data is stored**, memory usage can <u>exceed **1 GB**.</u>

- **Problem:** A simple program that loads the entire file into memory **consumes at least 100 MB** and <u>could exceed **1 GB**.</u>

- **Concern:** Personal computers have **8-16 GB RAM**
    - —a single inefficient program can **severely impact performance**.

Understanding Time & Space Complexity

- **CPU time and memory space** are **scarce resources** that must be used efficiently.

- **Two key measures of efficiency:**
  - **Time Complexity** → How execution time grows with input size.
  - **Space Complexity** → How much memory the program consumes.

- Performance varies based on input, but we focus on the **worst-case scenario**:
  - **Worst-case complexity** determines
    1) **maximum resource needs** and 2) **execution time limits**.

- While best-case and average-case complexities exist, **this book focuses only on worst-case complexity**.

Improving Worst-Case Performance of a Program

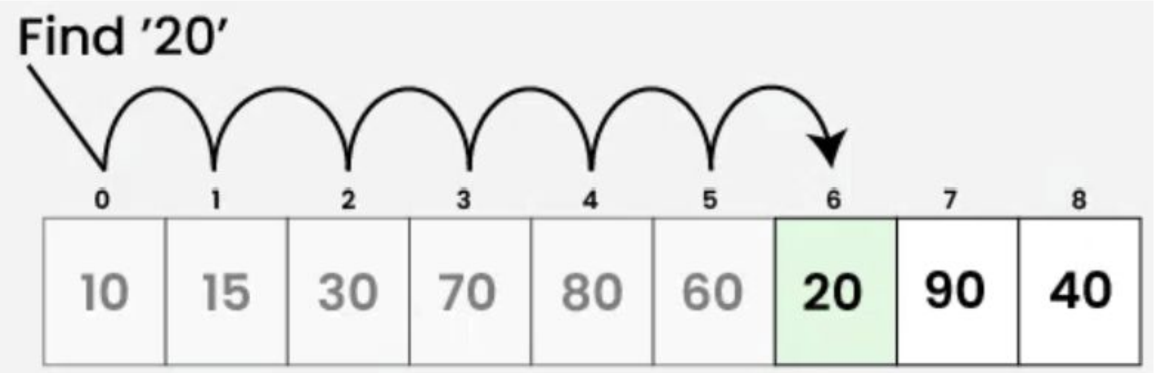
- Small modifications can **drastically improve performance**.

- **Example:** Searching in the NID database
  - **Naïve approach**: Load the <u>entire file</u> into memory
    - **High memory usage** (~1 GB).
  - **Optimized approach**: Read <u>one line at a time</u>, compare, then discard if not a match
    - **Only 20 bytes needed**.
  - **Performance gain**: Stops searching early if a match is found,
    - <u>reducing execution time.</u>

- Most performance improvements require **clever algorithms and data structures**.
- Data Structures & Algorithms (DSA) are core areas of computer science for this reason.

Linear Search

```
Function linearSearch (S, x) {
      L = length(S)
      for (i = 0; i < L; i = i + 1) {
            if (S[i] == x) then return i
      }
      return -1
}
```



Binary Search [data sorted in ascending]

- Binary Search is a searching algorithms which works on sorted array
- At every iteration it finds the middle element of the array using

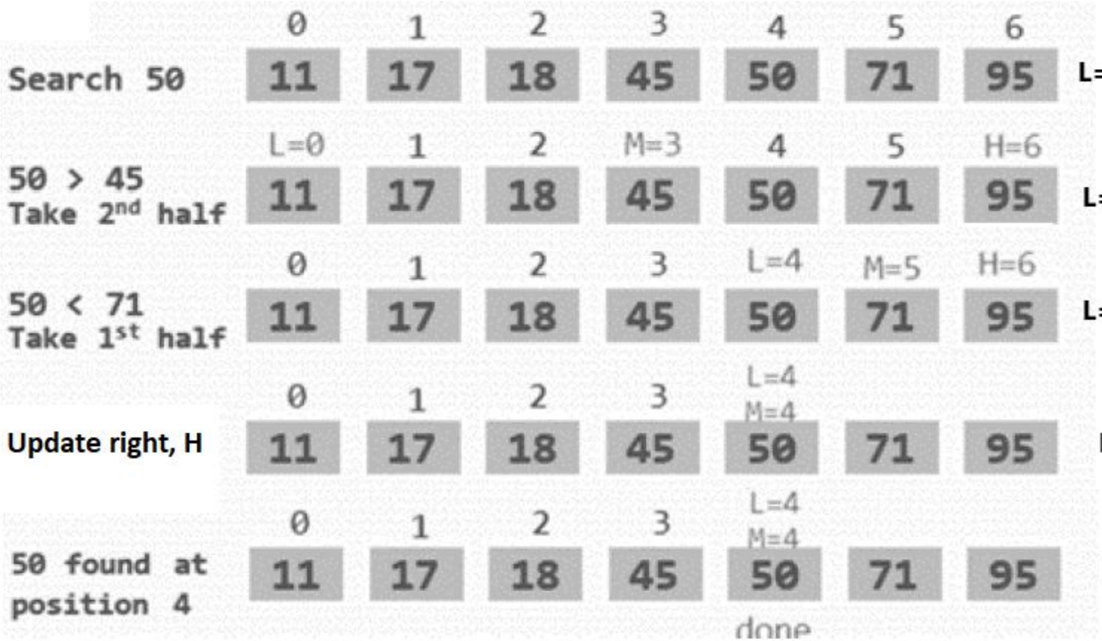
$$mid = \frac{(low + high)}{2}$$

- If middle element is equal to the key, search stops.
- If the middle element is greater than the key, search continues in the first half. Otherwise search continues in the second half.
- It has best case complexity of O(1), average and worst case complexity of O(log n)

Binary Search [data sorted in ascending]

```
Function binarySearch (S, x) {
      Length = length(S)
      L = 0
      H = Length - 1
      while (L <= H) {
            M = (L + H) / 2
            if (S[M] == x) then return M
            else {
                  if (S[M] < x) then L = M + 1
                  else H = M - 1
            }
      }
      return -1
}
```

```
Function binarySearch (S, x) {
      Length = length(S)
      L = 0
      H = Length - 1
      while (L <= H) {
            M = (L + H) / 2
            if (S[M] == x) then return M
            else {
                  if (S[M] < x) then L = M + 1
                  else H = M - 1
            }
      }
      return -1
}
```



L=0, H=6, M= (0+6)/2=3

L=4 (update), H=6

L=4, H=6, M=(4+6)/2=5

L=4, H=4 (update),

L=4, H=4, M=(4+4)/2=4
(Found)

Comparing Linear Search vs. Binary Search

Linear Search: O(N)

- Directly checks each element.
- **Example:** If N=1024, it takes **1024 comparisons**.

Binary Search: $O(\log_2 N)$

- Cuts the search space in half each step.
- **Example:** If N=1024, it only takes **10 comparisons** ($\log_2 1024 = \log_2 2^{10} = 10$).

Key Takeaway:

- **Binary search is exponentially faster** than linear search for **large inputs.**
- Choosing the right algorithm **can drastically improve performance**.

Understanding Asymptotic Complexity

- **Asymptotic complexity** describes how a program's **resource usage grows** with **input size N.**

- **Example:** If complexity is O(N^2), the actual time/space usage is **C · N^2** for some constant C.
  - The constant C depends on:
    - Code implementation details.
    - CPU architecture and operation costs.
    - Data size and structure.

- **Key takeaway:** We focus on how complexity **scales** rather than **exact execution time**.

Example – Linear Search Complexity

```
Function linearSearch (S, x) {
      L = length(S)
      for (i = 0; i < L; i = i + 1) {
            if (S[i] == x) then return i
      }
      return -1
}
```

- **Operations in worst-case for input size N:**

    1) Compute length → **1 operation**.
    2) Increment loop index → **N additions**.
    3) Compare index with length → **N comparisons**.
    4) Load and compare elements → N+ N = **2N operations**.
    5) Return statement → **1 operation**.

- **Total:** 1+ N + N + 2N+ 1 = 4N+2 operations.

- If another implementation takes **3N+2**,
    - It's **better**, but constant differences **don't matter** much.

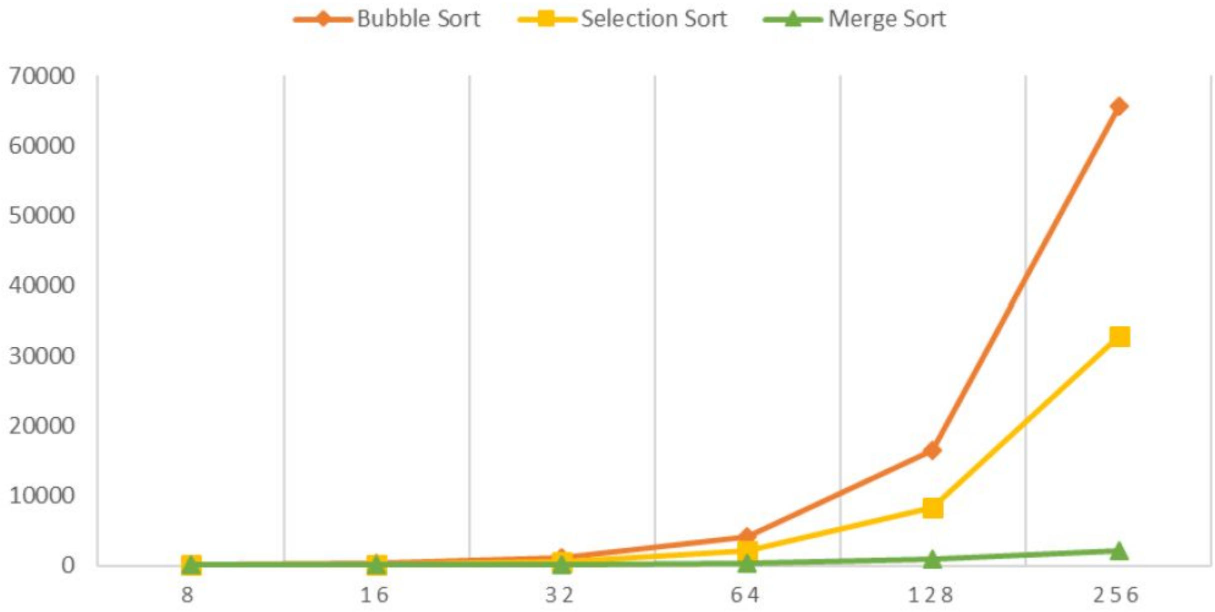
Why Ignore Constant Factors in Complexity?

- Compilers optimize execution, making small differences **negligible.**
- Different CPU operations have varying execution costs, making exact timing unpredictable.
- **Hardware improvements** make constant factor differences irrelevant over time.

- **Key takeaway:** What matters is **how complexity grows with input size** rather than small efficiency differences.

Comparing Sorting Algorithm Efficiency

- **Selection Sort:** O(n^2), runs in (n(n+1))/2 time.
- **Bubble Sort:** O(n^2), similar growth as selection sort.
- **Merge Sort:** O(nlogn) significantly faster for large inputs.

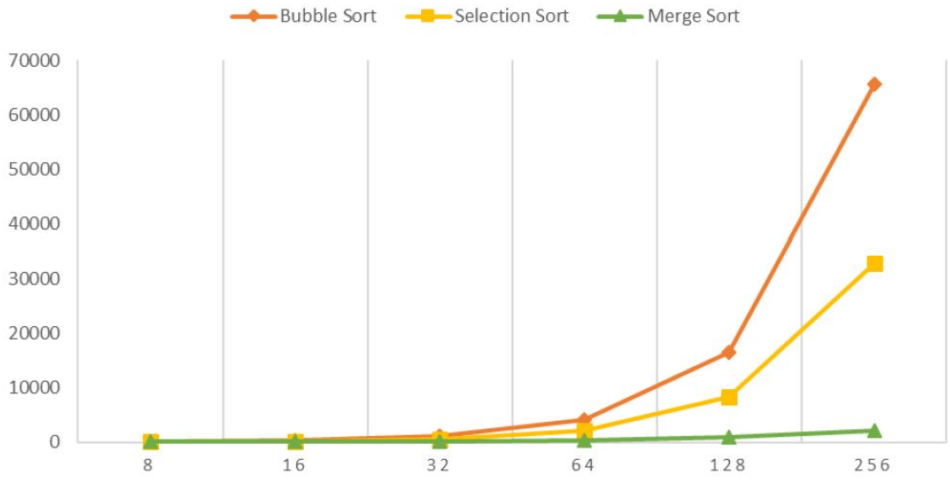


Comparing Sorting Algorithm Efficiency

- **Selection Sort:** O(n^2), runs in (n(n+1))/2 time.
- **Bubble Sort:** O(n^2), similar growth as selection sort.
- **Merge Sort:** O(nlogn) significantly faster for large inputs.

- **Key Takeaways:**
  - ✅ For small inputs, all three perform similarly.
  - ✅ As **N** grows, merge sort outperforms both selection and bubble sort.
  - ✅ Constant factors are negligible; the dominant term determines efficiency.

Input Size	Bubble Sort	Selection Sort	Merge Sort
8	64	28	24
16	256	136	64
32	1024	528	160
64	4096	2080	384
128	16384	8256	896
256	65536	32896	2048
512	262144	131328	4608
1024	1048576	524800	10240



Space Complexity of Bubble, Selection, and Merge Sort

- **Bubble Sort** and **Selection Sort** are in-place sorting algorithms.
  - They only require a small, constant amount of extra memory for **swaps.**
  - Therefore, their **space complexity** is **O(n)**.

- **Merge Sort**, though also an in-place algorithm, requires **extra memory** for temporary subarrays during the merging process.
  - The space needed for the **subarrays** is proportional to the **input size**.
  - Thus, the **space complexity** of **Merge Sort** is also **O(n)**.

- All three sorting algorithms have **O(n)** space complexity
  - Despite the different mechanisms used in the algorithms,
  - The additional memory used is proportional to the input size,

Understanding Big-O Notation

What is Big-O Notation?

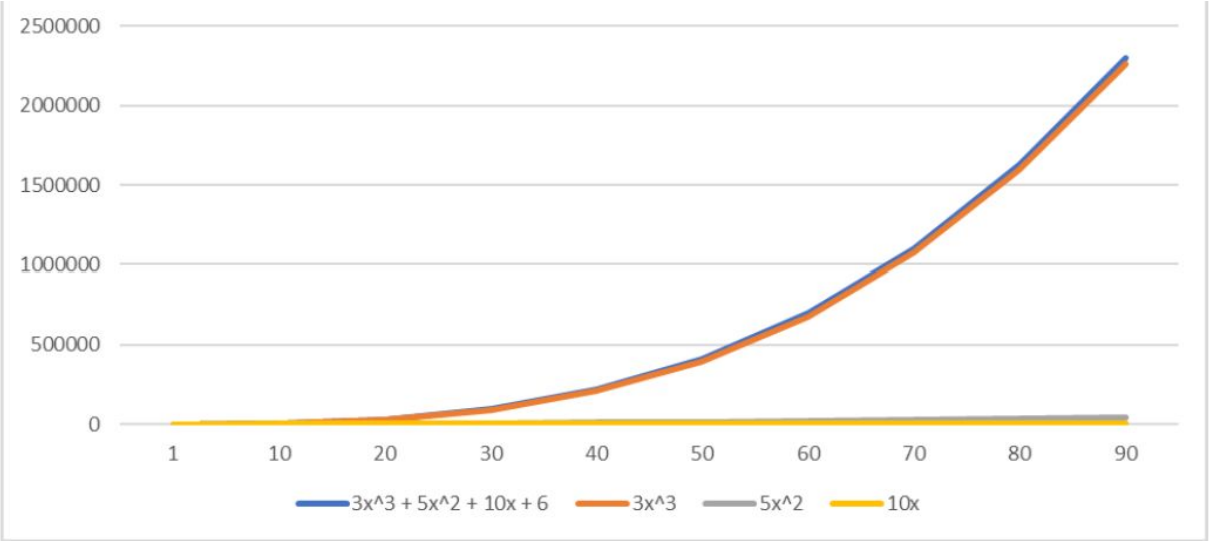
- It describes the **worst-case** time or space complexity of an algorithm as input size **N** grows to **infinity.**
- It helps compare algorithms by focusing on their **growth rate**, ignoring
  - 1) constant factors (Multiplicative & Additive)
  - 2) lower-order terms

- Example1 (constant factors): **O(N)**
  - Linear search algorithm takes T(N) = 4N + 2 operations in the worst case.
    - Multiplicative constants (4) are ignored because they don't affect growth rate.
    - Additive constants (+2) are insignificant for large N.

- Example2 (lower-order terms):
  - If time complexity is an^3 + bn^2 + cn + d,
    - we write **O(N^3)** because N^3 dominates for large N.

Understanding Big-O Notation

- Ignores lower-order terms:
  - 3x^3+5x^2+10x+6 and 3x^3 have same time complexity
  - 5x^2 is 2nd lowest
  - 10x has the lowest



How to Read Big-O Notation

- **O(1)**→ "Order 1" (Constant time)
- **O(logN)**→ "Order log N" (Logarithmic time)
- **O(N)** → "Order N" (Linear time)
- **O(NlogN)**→ "Order N log N" (Log-linear time)
- **O(N^2)** → "Order N squared" (Quadratic time)
- **O(N^3)** → "Order N cubed" (Cubic time)
- **O(2^N)**→ "Order 2 to the power of N" (Exponential time)
- **O(N!)**→ "Order N factorial" (Factorial time)

  - Bubble/Selection Sort → **O(N^2)**(quadratic).
  - Merge Sort → **O(NlogN)** (log-linear).

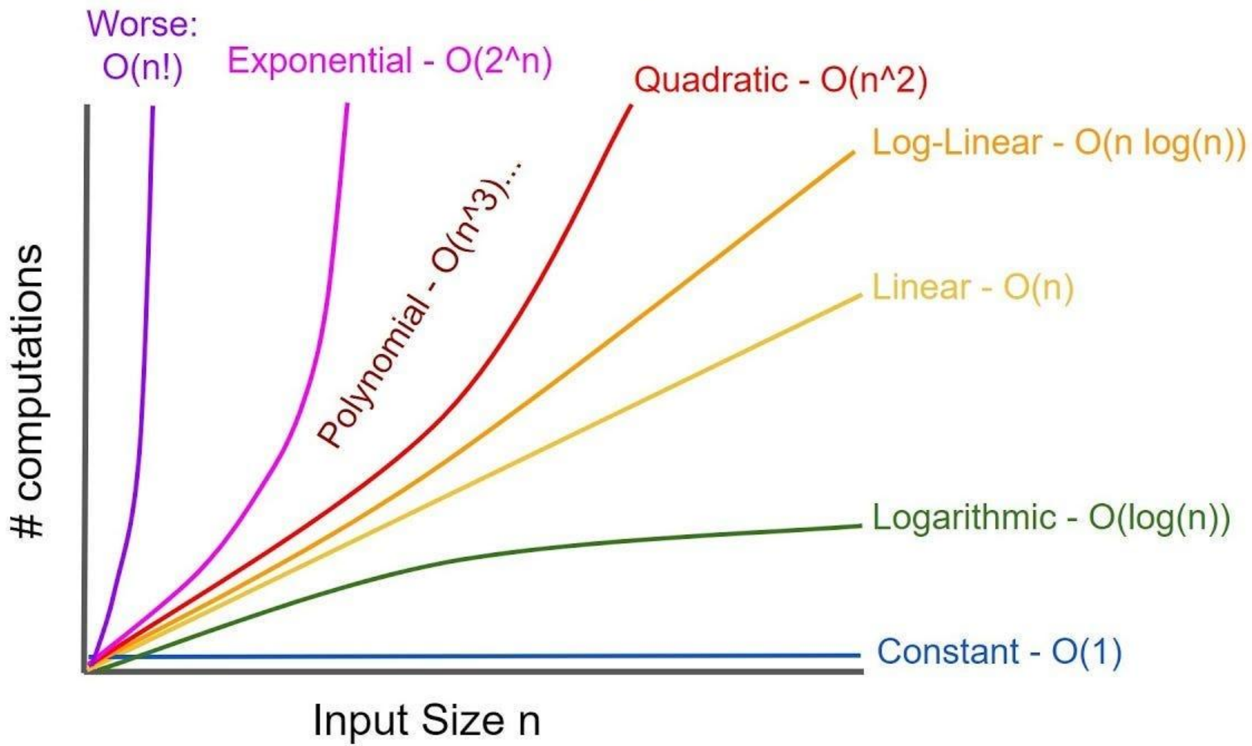
Typical Running Time Functions

- 1 (constant running time):
  - Instructions are executed once or a few times
- logN (logarithmic)
  - A big problem is solved by cutting the original problem in smaller sizes, by a constant fraction at each step
  - Shortcut: the number that is used for division or multiplication will be the base of log
- N (linear)
  - A small amount of processing is done on each input element
- N logN
  - A problem is solved by dividing it into smaller problems, solving them independently and combining the solution

Typical Running Time Functions

- N^2 (quadratic)
  - Typical for algorithms that process all pairs of data items (**double nested loops**)
- N^3 (cubic)
  - Processing of triples of data (**triple nested loops**)
- N^K (polynomial), 2^N (exponential)
  - Few exponential algorithms are appropriate for practical use

Typical Running Time Functions



Typical Running Time Functions

BEST

$O(1)$	CONSTANT
$O(\log n)$	LOGARITHMIC
$O(n)$	LINEAR
$O(n^2)$	QUADRATIC
$O(n^3)$	CUBIC
$O(n^k)$	POLYNOMIAL
$O(2^n)$	EXPONENTIAL
$O(n!)$	FACTORIAL

WORST

Complexity	Description	Examples
1	*Constant* algorithm does not depend on the input size. Execute one instruction a fixed number of times	Arithmetic operations (+, -, *, /, %) Comparison operators (<, >, ==, !=) Variable declaration Assignment statement Invoking a method or function
log N	*Logarithmic* algorithm gets slightly slower as N grows. Whenever N doubles, the running time increases by a constant.	Bits in binary representation of N Binary search Insert, delete into heap or BST
N	*Linear* algorithm is optimal if you need to process N inputs. Whenever N doubles, then so does the running time.	Iterate over N elements Allocate array of size N Concatenate two string of length N
N log N	*Linearithmic* algorithm scales to huge problems. Whenever N doubles, the running time more (but not much more) than doubles.	Quicksort Mergesort FFT
N^2	*Quadratic* algorithm practical for use only on relatively small problems. Whenever N doubles, the running time increases fourfold.	All pairs of N elements Allocate N-by-N array
N^3	*Cubic* algorithm is practical for use on only small problems. Whenever N doubles, the running time increases eightfold.	All triples of N elements N-by-N matrix multiplication
2^N	*Exponential* algorithm is not usually appropriate for practical use. Whenever N doubles, the running time squares!	Number of N-bit integers All subsets of N elements Discs moved in Towers of Hanoi
N!	*Factorial* algorithm is worse than exponential. Whenever N increases by 1, the running time increases by a factor of N	All permutations of N elements

Multi-Parameter Complexity [examples in analysis 11, 12]

- When input has multiple attributes, complexity depends on **all relevant factors**.

- **Example:** Traversing a graph with N vertices and M edges → **O(N+M)**.

- **Key takeaway:** Some problems require expressing complexity in terms of **multiple input sizes**.
  - We will see details in the graph

Example01

Code:

```
a = b;
```

Time complexity: ?

Space complexity: ?

Example01

Code:

```
a = b;
```

Time complexity: O(1)

Space complexity: O(1) — Since it's a simple value assignment

Example02

Code:

```
sum = 0;
for (i=1; i <=n; i++)
      sum += n;
```

Time complexity: ?
Space complexity: ?

Example02

Code:

```
sum = 0; O (1)
for (i=1; i <=n; i++) // O (n)
      sum += n; O (1)
```

Time complexity: O (1) + O (n) X O (1) = O (1) + O (n) = O (n)

Space complexity: O(1) — The loop runs O(n) times, but no extra space is used apart from a few variables, making it constant space.

Example03

Code:

```
sum1 = 0;
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
            sum1++;


Time Complexity: ?
Space complexity:??
```

Example03

Code:

```
sum1 = 0;O(1)
for (i=1; i<=n; i++) O(n)
    for (j=1; j<=n; j++) O(n)
            sum1++; O(1)
```

Time complexity: $O(1) + O(n) \times O(n) = O(n^2)$

Space complexity: **O(1) – only a few integer variables are used, requiring constant space.**

Example04

Code:

```
sum1 = 0;
for (i=1; i<=n; i++)
   for (j=1; j<=2*n; j++)
         sum1++;

Time complexity:??
Space complexity:??
```

Example04

Code:

```
sum1 = 0;  O(1)
for (i=1; i<=n; i++)  O(n)
    for (j=1; j<=2*n; j++)  O(2n)= O(n)
          sum1++;  O(1)
```

`Time complexity:` $O(1) + O(n) \times O(n) = O(n^2)$

`Space complexity:` **O(1)**

Example05

Code:

```
sum = 0;
for (j=1; j<=n; j++)
    for (i=1; i<=j; i++)
            Sum++;

for (k=0; k<n; k++)
    s = k;
```

Time complexity: ?

Space complexity:??

Example 05

Code:

```
sum = 0; O(1)
for (j=1; j<=n; j++)    O(n)
     for (i=1; i<=j; i++)    O(n)
                Sum++;   O(1)

for (k=0; k<n; k++) O(n)
     s = k; O(1)
```

$O(n)* O(n) = O(n^2)$

$O(n)* O(1) = O(n)$

Time complexity: $O(1) + O(n^2) + O(n) =$ **$O(n^2)$**

Space complexity: **O(1)**

Example06

Code:

```
sum1 = 0;
n = 100;
for (k=1; k<=n; k*=2) // k = k*2
       for (j=1; j<=n; j++)
           Sum1++;
```

```
Time complexity: ??
Space complexity:??
```

Example06

Code:

```
sum1 = 0; O(1)
n = 100; O(1)
for (k=1; k<=n; k*=2) //k starts at 1 and doubles each time
                      //(runs for half of n time) = O(log2n)
      for (j=1; j<=n; j++)//j runs from 1 to n = O(n)
          Sum1++; O(1)
```

Time complexity: $O(1) + O(1) + O(\log_2 n) \times O(n) = O(n\log_2 n)$

Space complexity: $O(1)$

Example06

Code:

$\log_2(100)$=6.67= approx 7

```
sum1 = 0;
n = 100;
for (k=1; k<=n; k*=2) // k=1 2 4 8 16 32 64 = total 7 times
      for (j=1; j<=n; j++)
            sum1++;
```

Time complexity: **$O(\log_2 n)$ x $O(n)$ = $O(n\log_2 n)$**

Example07

Code:

```
sum1 = 0;
n = 100;
for (i=n; i>=1; i=i/2)
        for (j=1; j<=n; j++)
            Sum1= Sum1 + i ;
```

Time complexity: **?**

Space complexity:???

Example07

Code:

```
sum1 = 0; O(1)
n = 100; O(1)
for (i=n; i>=1; i=i/2)  //i starts at n and is halved each time
                        //= O(log2n)
        for (j=1; j<=n; j++) //j runs from 1 to n = O(n)
            Sum1= Sum1 + i ; O(1)
```

Time complexity: $O(1) + O(1) + O(\log_2 n) \times O(n) = O(n\log_2 n)$

Space complexity: $O(1)$

Example07

$\log_2(100)=6.67=$ **approx 7**

Code:

```
sum1 = 0;
n = 100;
for (i=n; i>=1; i=i/2) // i=100 50 25 12 6 3 1  = total 7 times
        for (j=1; j<=n; j++)
            Sum1= Sum1 + i ; O(1)
```

Time complexity: $O(\log_2 n) \times O(n) = O(n\log_2 n)$

Example08

Code:

```
a=0
N= 5
i=N

while (i>0):
     a = a+ i;
     i = i/2


Time complexity: ??
Space complexity:??
```

Example08

Code:

```
a=0
N= 5
i=N

while (i>0):
     a = a+ i; O(1)
     i = i/2 //i starts at N and is halved each time = O(log2n)
```

Time complexity: $O(\log_2 n) \times O(1) = O(\log_2 n)$

Space complexity: $O(1)$

Example09

Code:

```
sum1 = 0;
n = 100;
for (i=n; i>=1; i=i/2)
        for (k=1; k<=n; k*=2)
            Sum1= Sum1 + i ;
```

Time complexity: **??**

Space complexity:??

Example09

Code:

```
sum1 = 0;
n = 100;
for (i=n; i>=1; i=i/2)  O(log2n)
        for (k=1; k<=n; k*=2) O(log2n)
            Sum1= Sum1 + i ; O(1)
```

Time complexity: $O(\log_2 n) \times O(\log_2 n) \times O(1) = O((\log_2 n)^2)$

Space complexity: $O(1)$

Example10

```
sum1 = 0;
n = 5;
for (i=0; i<=n; i=i+1)
        for (k=0; k<=i*n; k=k+1)
            Sum1= Sum1 + i ;

for (i=0; i<=n+n; i=i+1)
          Sum1= Sum1 + i
```

```
Time complexity: ???
Space complexity:??
```

Example10

```
sum1 = 0; O(1)
n = 5; O(1)
for (i=0; i<=n; i=i+1)   O(n)
        for (k=0; k<=i*n; k=k+1) O(n²), max value of i is n
            Sum1= Sum1 + i ; O(1)

for (i=0; i<=n+n; i=i+1)   O(2n)= O(n)
          Sum1= Sum1 + i ; O(1)

Time complexity: O(1) + O(1) + O(n³)+ O(n) = O(n³)
Space complexity: O(1)
```

$O(n)* O(n^2) = O(n^3)$

Example11

Code:

```
sum1 = 0;
n = 100;
m = 10;

for (i=n; i>=1; i=i-1):
        for (k=1; k<=m; k=k+2):
            Sum1= Sum1 + i + k ;

for (i=1; i<=m; i=i+1):
            print(i) ;
```

Time complexity: **??**

Space complexity:???

Example11

Code:

```
sum1 = 0;
n = 100;
m = 10;

for (i=n; i>=1; i=i-1): O(n)
         for (k=1; k<=m; k=k+2): O(m/2)= O(m)
             Sum1= Sum1 + i + k ; O(1)

for (i=1; i<=m; i=i+1):  O(m)
             print(i) ; O(1)
```

Time complexity: **O(n)x O(m)x O(1) + O(m)x O(1)**
= O(n x m) + O(m)
= O(n x m)

Space complexity: **O(1)**

Example12

Code:

```
arr=[];
counter=0;
N=100, M=40;
for (i=1; i<=N; i=i+1):
    arr.append(i)
for (i=1; i<=M; i=i+1):
    counter+=1
```

Time complexity: **????**

Space complexity:???

Example12

Code:

```
arr=[]; O(1)
counter=0; O(1)
N=100, M=40; O(2)
for (i=1; i<=N; i=i+1): O(N)
    arr.append(i) O(1)
for (i=1; i<=M; i=i+1): O(M)
    counter+=1 O(1)
```

Time complexity: **O(1)+O(1)+O(2) + O(N)xO(1)+ O(M)x O(1)**

= O(4)+ O(N) + O(M)
= O(N) + O(M)
= O(N+M)

Example12

Code:

```
arr=[];
counter=0;
N=100, M=100;
for (i=1; i<=N; i=i+1):
    arr.append(i)
for (i=1; i<=M; i=i+1):
    counter+=1
```

`Space complexity:` **????**

Example12

Code:

```
arr=[];
counter=0;
N=100, M=100;

for (i=1; i<=N; i=i+1):
    arr.append(i) O(N) because N number of elements are being inserted.
                        So, space requirement is N

for (i=1; i<=M; i=i+1): O(1): Space needed for local variables only
    counter+=1
```

Space complexity: **O(N)**

Example13

Code:

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
    for (j = 2; j <= n; j = j * 2) {
        k = k + n / 2;
    }
}
```

```
Time complexity: ???
```

Example13

Code:

```
int i, j, k = 0; O(1)
for (i = n / 2; i <= n; i++) { O(n/2)=O(n)
    for (j = 2; j <= n; j = j * 2) { O(log2n)
        k = k + n / 2; O(1)
    }
}
```

Time complexity: $O(n) \times O(\log_2 n) \times O(1) = O(n\log_2 n)$

Example14

Code:

```
int i, j;
if (condition) {                    // Suppose condition is true in some cases
    for (i = 0; i < n; i++) {
        a[i] = i + 2;
    }
}
else {
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            a[i] = a[j] + 1;
        }
    }
}
```

Time complexity: **???**

Example14

Code:

```
int i, j;                          // O(1)
if (condition) {                   // Suppose condition is true in some cases
    for (i = 0; i < n; i++) {   // O(n)
        a[i] = i + 2;           // O(1)
    }
}
else {
    for (i = 0; i < n; i++) {       // O(n)
        for (j = 0; j < n; j++) {   // O(n)
            a[i] = a[j] + 1;        // O(1)
        }
    }
}
```

Time complexity: **$O(n) + O(n^2) = O(n^2)$**

Practical Use of Asymptotic Complexity

- **Ignoring Constants:**
  - Asymptotic complexity ignores constant multipliers and additive factors.
  - However, constant coefficients of <u>different powers of **N** may matter</u> in real-world applications.

- **Estimating Resource Requirements**
  - Used to estimate the <u>**time** and **space** costs</u> of a program for **large input sizes.**
  - Helps in selecting the **most efficient implementation.**

Practical Use of Asymptotic Complexity

- **Example: Space Complexity Limitation**
  - Two implementations have **C_1 * n $\log_2 n$** and **C_2 * n $\log_2 n$** same time complexity (**Same**).
  - Assume:
    - **RAM size** = 32GB
    - **Understanding the Input Size:**
      - The maximum input size is **512M (512 million) = 512M = 2^{29}** entries.
      - Each entry is **1 byte**, so storing all entries would take **2^{29} bytes (≈ 512MB)**

    - **Computing Memory Requirement for Processing:**
      - The time complexity given is **n $\log_2$n.**
      - For **n = 2^{29}**, we compute: n**$\log_2$n**= (2^29)×(**30)**
      - Here, **$\log_2(2^{29})$ = 29**, so we assume an additional small factor (≈ 30) for overhead.
      - **Approximate: 2^{29}**×30 ≈ **2^{30}**×15
      - Since **2^{30}** bytes = 1GB, Memory requirement is 15GB

Practical Use of Asymptotic Complexity

- **Example: Space Complexity Limitation**
  - Two implementations have **C_1 * n $\log_2 n$** and **C_2 * n $\log_2 n$** time complexity (**Same**).
  - Assume:
    - **RAM size** = 32GB
    - **Max input size** = 512M (≈ **2^{29}**) entries of 1 byte each
    - **n $\log_2 n$** = **2^{29} * 30** ≈ 15GB

    - **Impact of Constants (C_1 and C_2):**
      - If **C_1 = 2**, the required space is = 2×15 GB=**30 GB**
        - This **fits within** the available 32GB RAM
      - If **C_2 = 3**, the required space is = 3×15GB=**45 GB**
        - This **exceeds** the available 32GB RAM, making execution **impossible**

First implementation is the only viable option.

Practical Use of Asymptotic Complexity

- **Time vs. Space Tradeoff**
  - **Optimization dilemma:** Faster execution often requires more space, and vice versa.
    - Faster execution often requires more memory due to additional data storage or complex operations.
    - Reducing memory usage can slow down execution due to less efficient algorithms or extra resource management steps.
  - **Real-world impact:** Some problems require balancing time and space efficiency **based on constraints.**

The End