Design and Analysis of Algorithms

Lecture - 3

Success is always inevitable with Hard Work and Perseverance

N. Ravitha Rajalakshmi

Learning Objective

- Understand the Analysis Framework used for measuring time efficiency of an algorithm.
- Introduce the Asymptotic Notations..

Formal Framework to compute runtime

- Framework that helps in identifying efficient algorithms
    - Finding actual runtime is difficult
    - It either increases (or) decreases runtime computation by a constant factor.

- Need for a measure of runtime that ignore the constant multiples

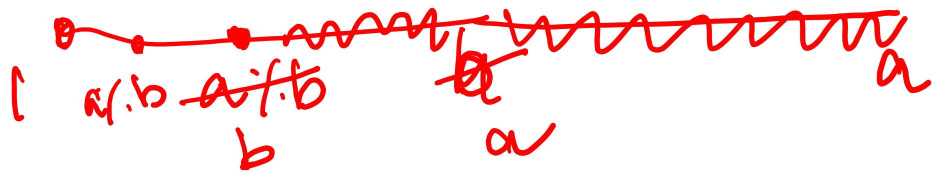
"Measure the runtime with respect to the growth of the input size"

Input size

Time Complexity (or) Time efficiency = $f\,(input\ size^{*})$

*Choice of input size is influenced by the key operation (basic operation)

Problem	Input Size
Find smallest element in an array	Number of elements in array
Multiplying two polynomial	Degree of the polynomial
Spell Checker	Number of words (or) characters
Matrix Multiplication	Order of the matrix
GCD of two numbers	Largest of the two number



Basic Operation

Time Complexity (or) Time efficiency = $f(input\ size^*)$

$\cong C_{op}C(n)$ [Approximate]

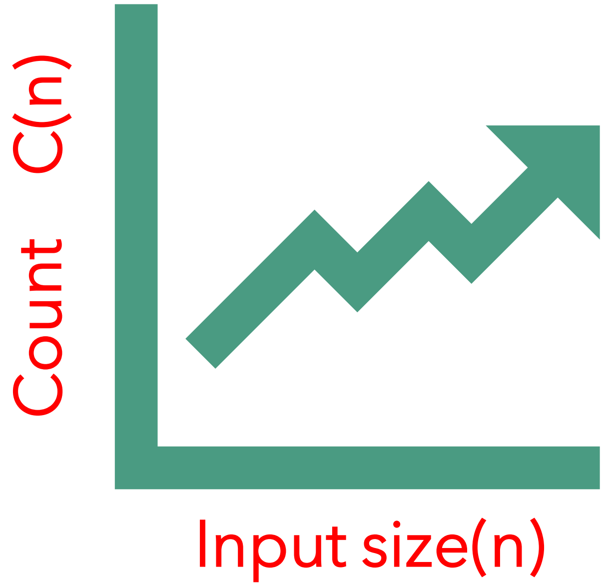
C_{op} - Cost of basic operation

$C(n)$ - Count of basic operation

Problem	Basic Operation
Find smallest element in an array	Comparison with array elements
Spell Checker	Comparison with words in dictionary {wlak} – {walk, flak} - no of characters misplaced and return result
GCD of two numbers	Comparison with array elements

Approximate Notion

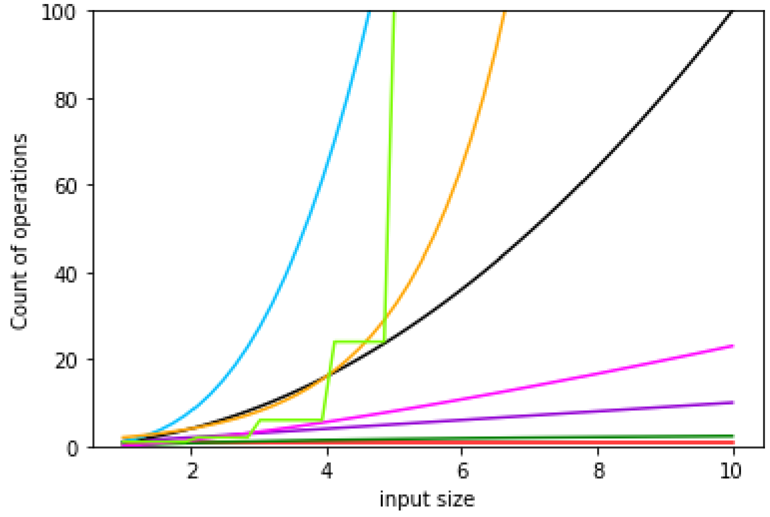
- Is it realistic to obtain C_{op}
  - No, its machine dependent!
- T(n) $\cong C(n)$
- Do we need exact C(n) ?
  - No, Interested in Order of growth of function



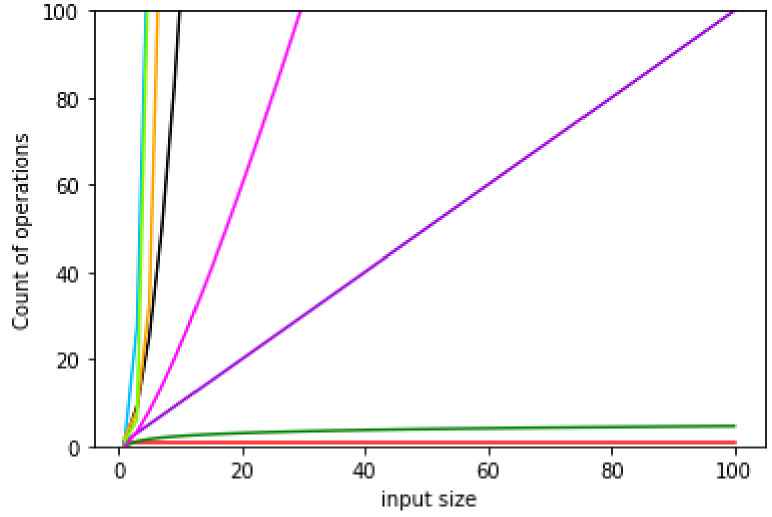
Common Order of growth functions

- Growth of number of basic operation w.r.t to increase in the input size

$$1 < \log n < n < n\ log n < n^2 < n^3 < 2^n < n!$$



Smaller inputs



Larger inputs

Pause & Think

- Consider the problem of finding factorial of a number (Assume iterative version) , what would be the input size and basic operation?

Function Factorial (n)

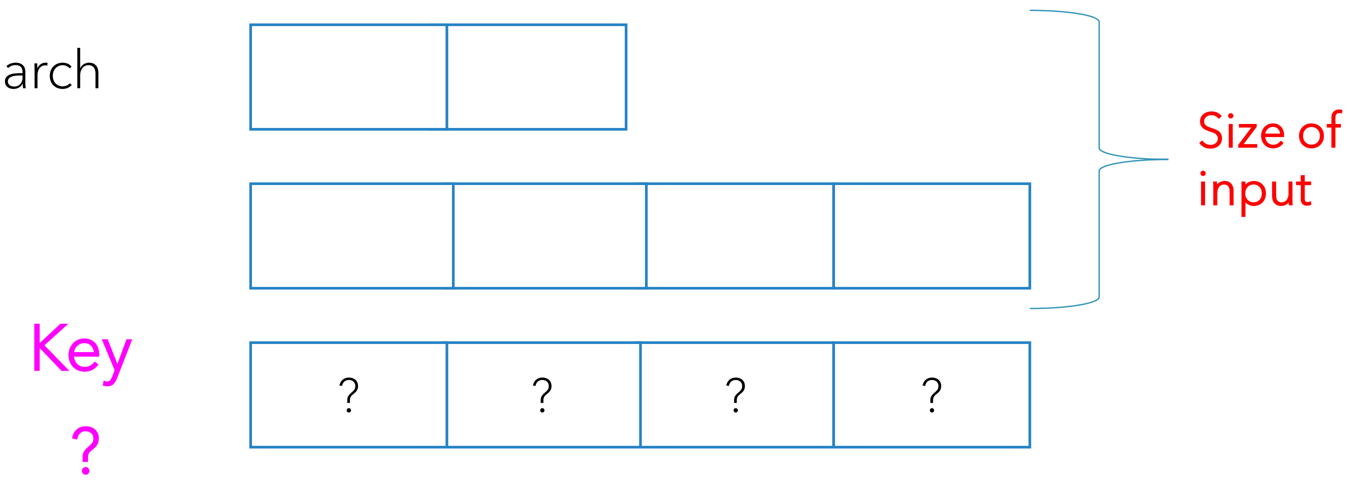
```
# n – number n>=1
fact = 1;
for (int i = 1; i<=n; i++){
        fact = fact * i;
}
return fact
```

n – Input size
Key Operation - Multiplication

Algorithm Efficiency

- Will the count vary based on the specifics of an input?
    - In other terms, with actual value of input does the count differ?

Take the example of linear search



Efficiency w.r.t Good, Bad Inputs

- Some input cases, count is going to be smaller
  - Best case complexity
- Some input cases, count is going to be larger
  - Worst case complexity
- It is not possible to inspect every possible test case
  - What is the reasonable time taken by the algorithm on any random input for its computation
    - Average case complexity

Linear Search

Input: An array A with n elements. A key k. .

Output: An index, i, where A[i] = k. If there is no such i, then NOT_FOUND.

Index	1	2	3	4
Array Elements	80	100	30	60

K

Linear Search
Best Case

Input: An array A with n elements. A key k. .

Output: An index, i, where A[i] = k. If there is no such i, then NOT_FOUND.

Index	1	2	3	4
Array Elements	80	100	30	60

80

Element is present at the first position

$C(n) = 1$

Linear Search Worst Case

Input: An array A with n elements. A key k. .

Output: An index, i, where A[i] = k. If there is no such i, then NOT_FOUND.

Index	1	2	3	4
Array Elements	80	100	30	60

10

Element is not present in the array

$C(n) = n$

Linear Search Average Case

Element can be present at any random position

Input: An array A with n elements. A key k. .

Output: An index, i, where A[i] = k. If there is no such i, then NOT_FOUND.

Let p denote probability of a successful search

(1- p) denote the probability of unsuccessful search

There are n different possibilities if the search was successful . Prob (every successful search) = 1/n

= Prob of successful + Prob of unsuccessful

Linear Search
Average Case

= Prob of successful + Prob of unsuccessful

= p * [1/n * 1 + 1/n * 2 + ... 1/n * n] + (1-p) * n

= p /n * $\sum_{i=1}^{n} i$ + (1- p) * n

= p / n * n(n+1)/2 + (1-p) * n

= p * [(n+1)/2] + (1-p) * n

Pause & Think

- In the average case formulation for time complexity, what happens if p is set to a value of 0?
  - If p = 0 , it indicates unsuccessful search
  - T(n) = n

$$T(n) = p * [(n+1)/2] + (1-p) * n$$

- What happens if its set to a value of 1?
  - If p=1 , it indicates successful search
  - T(n) = (n+1) /2 , nearly half of the elements are searches on an average case

Asymptotic Notations

- To rank and compare order of growth
- O (Big Oh), θ (Big theta), Ω (Big omega)

Definition O (Big Oh) Notation

$f(n) = O(g(n))$ (f is Big-O of g) or $f \preceq g$ if there exist constants N and c so that for all $n \geq N$, $f(n) \leq c \cdot g(n)$.

Larger inputs

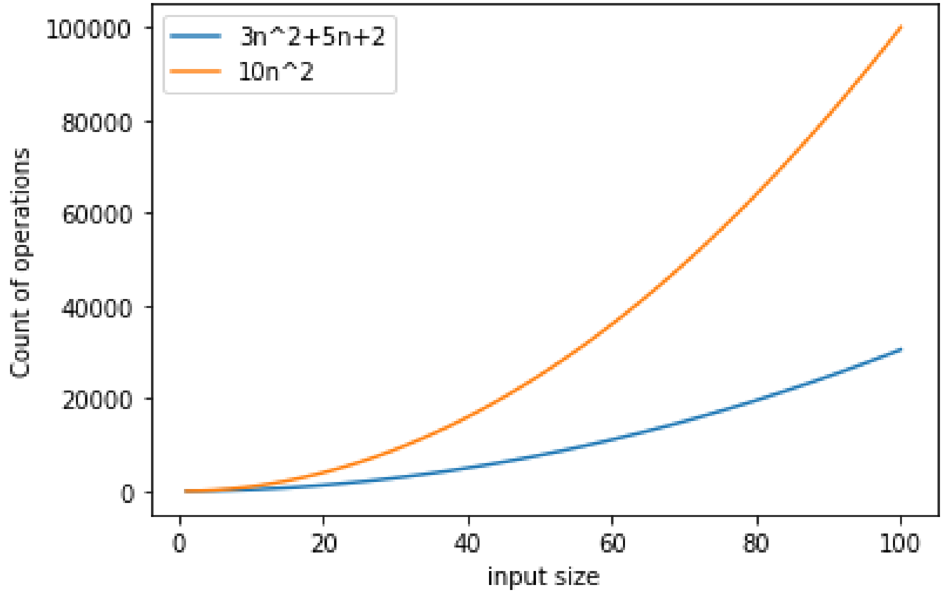
f is bounded by g

Asymptotic Notation

Actual Runtime $\mathrm{f(n)} = 3n^2 + 5n + 2$

Can we represent f (n) using a function $\mathrm{g(n)} = n^2$

- $3n^2 + 5n + 2 =$ **$O(n^2)$** since if n ≥ 1,
- $3n^2 + 5n + 2 \leq 3n^2 + 5n^2 + 2n^2 = \mathbf{10n^2}$.



Summary

- Discussed basic analysis framework used for iterative algorithm.
- Use of Asymptotic notations and their necessity

Thank You
Happy Learning

Success is always inevitable with Hard Work and Perseverance