CS2x1: Data Structures and Algorithms

Koteswararao Kondepu

k.kondepu@iitdh.ac.in

List of Topics [C201]

- **Introduction:**
  - ***Data structures***
  - ***Abstract data types***
- **Creation and manipulation of linear data structures:**
  - ***Arrays; Stacks; Queues; Circular Queues; Singly Linked lists; Circular Singly Linked List; Doubly Linked List; Circular Doubly Linked List***

- Introduction to Algorithms

- Creation and manipulation of non-linear data structures:
  - *Trees; Heaps; Hash tables; Balanced trees; Tries; Graphs.*

- Algorithms for sorting and searching, depth-first and breadth-first search, shortest paths and minimum spanning tree.

What is an Algorithm?

Google

define: algorithm

algorithm

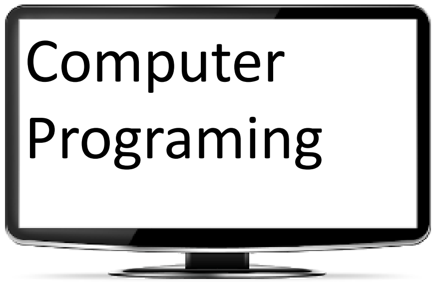
/ˈalɡərɪð(ə)m/

noun

a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

Define: Algorithm

- Algorithms



- Algorithm: well-defined computation procedure, step-by-step instructions

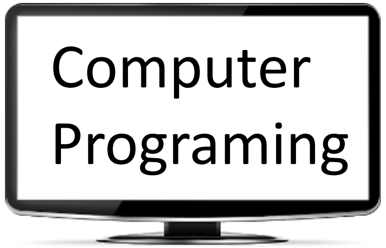
Some values
(or)
Set of values
Input

?????

Some values
(or)
Set of values
Output

Solution: Algorithm

- Algorithms



- Algorithm: well-defined computation procedure, step-by-step instructions

Some values
(or)
Set of values
Input

Algorithm
Step1
Step2
....
Stepk

Some values
(or)
Set of values
Output

- **Algorithm**: A sequence of computational steps that transform the ***input*** into the **output**

Example: Algorithm

- **Input:** A sequence of n number $<a_1, a_2, \dots a_n>$
- **Output:** The reordering or rearranging of input sequence $\rightarrow a_1 \le a_2 \le \dots \le a_n$

<31, 41, 59, 26, 41, 58> → **Sorting Algorithm** → <26, 31, 41, 41, 58, 59>

instance

Instance of the problem

- Infinitely many ***correct*** algorithms for the same algorithmic problem

Correctness: Algorithm

- Algorithm: Must prove that it always returns the desired output for all correct instances of the problem.
- For sorting:

  (i) Instance of the problem contains repeated elements

  <31, 26, 41, 26, 41, 31> → **Sorting Algorithm** → <26, 26, 31, 31, 41, 41>

  (ii) The input already might be sorted

  <26, 26, 31, 31, 41, 41> → **Sorting Algorithm** → <26, 26, 31, 31, 41, 41>

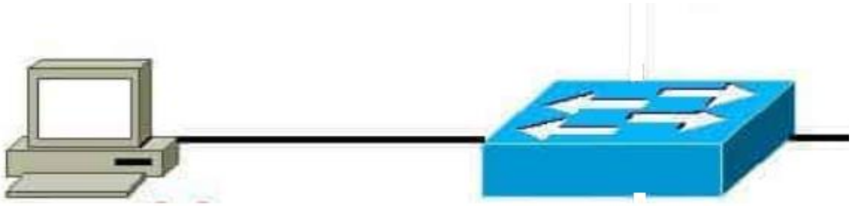
- Algorithm is said to be ***correct*** if it halts with the correct output for every input instance!
- An ***incorrect*** algorithm might not halt at all on some input sequence problems
- **Correctness** is not obvious in many optimization problems!

Expressing Algorithms

- **Algorithm:** Design the problem
  Sequence of steps

  Hardware design
  English
  Flowchart
  Pseudocode

  Hardware and Software systems ***independent***



Hardware and Software systems ***dependent***

- **Program:** An <u>implementation</u> of an Algorithm in any given programming language

What is a Good Algorithm?

- ***Efficiency:***
  - *(i) Running time*
  - *(ii) Space consumed*

✓ ***<u>Runtime</u>*** is determined by the primitive operations carried out during the execution of the algorithm (in compiled code, by the interpreter, etc.)

✓ Different algorithms are devised to solve the same problem and often differ in their efficiency

- *Efficiency as a function of input size:*
  - *(i) Insertion of an element at the end in a singly linked*
  - *(ii) Insertion of an element at the end in a circular doubly linked list*

- ***Input size*** *might be: -*
  - *The number of items in the input (e.g., as in a list)*
  - *An algorithm may also be dependent on more than one input*

Measuring the Running Time

- How should we measure the running time of an **algorithm**?

Experimental Study:

- Write a program that implements an algorithm
- Run the program by varying the input size
- We can use *clock_t* in the case of C for timestamping

Example: Measuring the Running Time

```
#include<stdio.h>
#include <time.h>
long int fact(int n);
void main()
{
long int i;
clock_t begin, end;
long int fact_var;
for (i=1; i<1000000; i=i+19999)
 {
    begin = clock();
    fact_var=fact(i);
    end = clock();
    long double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    printf("Runtime of Fact (%ld)= %Lf\n", i, time_spent);
  }
}
long int fact(int n) {
    if (n>=1) return n*fact(n-1);
    else return 1;
}
```

Output:

```
Runtime of Fact (1)= 0.000002
Runtime of Fact (20000)= 0.001043
Runtime of Fact (39999)= 0.001193
Runtime of Fact (59998)= 0.001442
Runtime of Fact (79997)= 0.001597
Runtime of Fact (99996)= 0.002272
Runtime of Fact (119995)= 0.002640
Runtime of Fact (139994)= 0.002815
Runtime of Fact (159993)= 0.003231
```

Limitations on Experimental Measurements:

- It is always necessary to <u>implement</u> and test the algorithm to determine its running time

- Experiments can be done only on a limited set of inputs, and it <u>may not be indicative of the running time on the other inputs</u>

- When required to compare algorithms → the same *Hardware and Software* environments should be used

Example: Measuring the Running Time

```c
#include<stdio.h>
#include <time.h>
long int fact(int n);
void main()
{
long int i;
clock_t begin, end;
long int fact_var;
for (i=1; i<1000000; i=i+999){
    begin = clock();
    fact_var=fact(i);
    end = clock();
    long double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    printf("Runtime of Fact (%ld)= %Lf\n", i, time_spent);
  }
}
long int fact(int n) {
    if (n>=1) return n*fact(n-1);
    else return 1;
}
```

Output:

```
Runtime of Fact (34966)= 0.000427
Runtime of Fact (35965)= 0.000603
Runtime of Fact (36964)= 0.000573
Runtime of Fact (37963)= 0.000473
Runtime of Fact (38962)= 0.000624
Runtime of Fact (39961)= 0.000568
Runtime of Fact (40960)= 0.000678
Runtime of Fact (41959)= 0.000652
Runtime of Fact (42958)= 0.000570
Runtime of Fact (43957)= 0.000567
Runtime of Fact (44956)= 0.000664
```

Measure Beyond Experimental

- Develop → <u>Generic methodology</u> for analyzing the running time of an algorithm

    - Use high-level description → instead of testing its implementations
    - Consider all possible inputs
    - Evaluate the efficiency of the algorithm in a way that it is independent of the hardware and software

Algorithmic Time Complexity

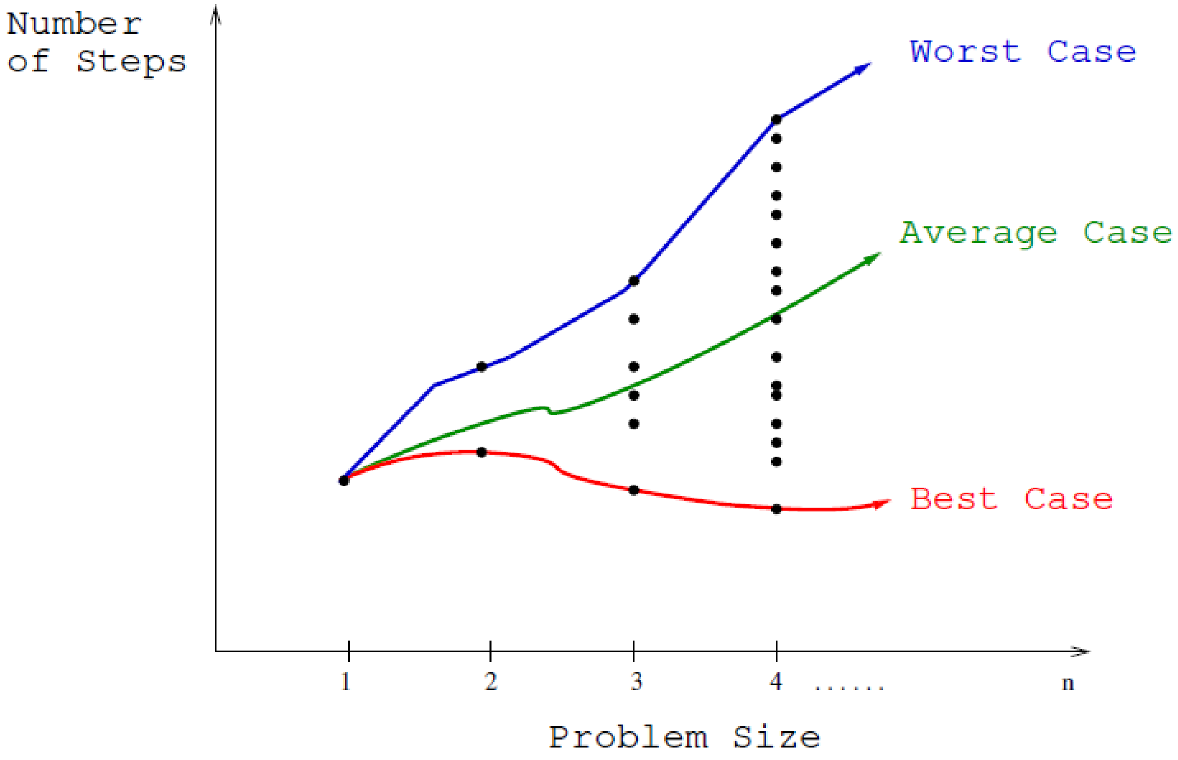
- Important to know → analyzing algorithmic time complexity
  - Each "simple" operation (+, -, =, if, call) takes 1 step
  - Loops (program constructs) → are not simple operations, hence they depend upon the size of the *input size*
  - Methods (subroutines) → For example, "fact" is not a single-step operation
  - Primitive operations → data movement (e.g., assign), control (e.g., return) takes 1 step

- If the code is small, by inspecting the pseudo-code → measure the run time of an algorithms by counting the number of steps

Different Time Complexities

- The worst-case complexity of an algorithm → the maximum number of steps taken on any instance of size *n.*
- The average-case complexity of an algorithm → the average number of steps taken on any instance of size *n.*
- The best-case complexity of an algorithm → the minimum number of steps taken on any instance of size *n.*

Function → Time vs. Size



Credit to: Analysis of Algorithms Lectures by Stevn S. Skiena

Algorithm Time Complexities (1)

Constant Time complexity: O (1)

```
void display_elemet(int *arr, int index){

printf("array[%d]: %d\n", index, arr[index]);

}
```

Linear Time complexity: O (n)

```
void display_elemet(int *arr){

for (index=0; index< n; index++){

    printf("array[%d]: %d\n", index, arr[index]);
  }
}
```

Algorithm Time Complexities (2)

Quadratic Time complexity: O (n^2)

```
void display_elemet(int *arr, int n){
int i, j, size;
for (i=0; i< n; i++){
   for (j=0; j < n; j++){
     printf("array[%d]: %d\n", size, A[j]);
  }
}
```

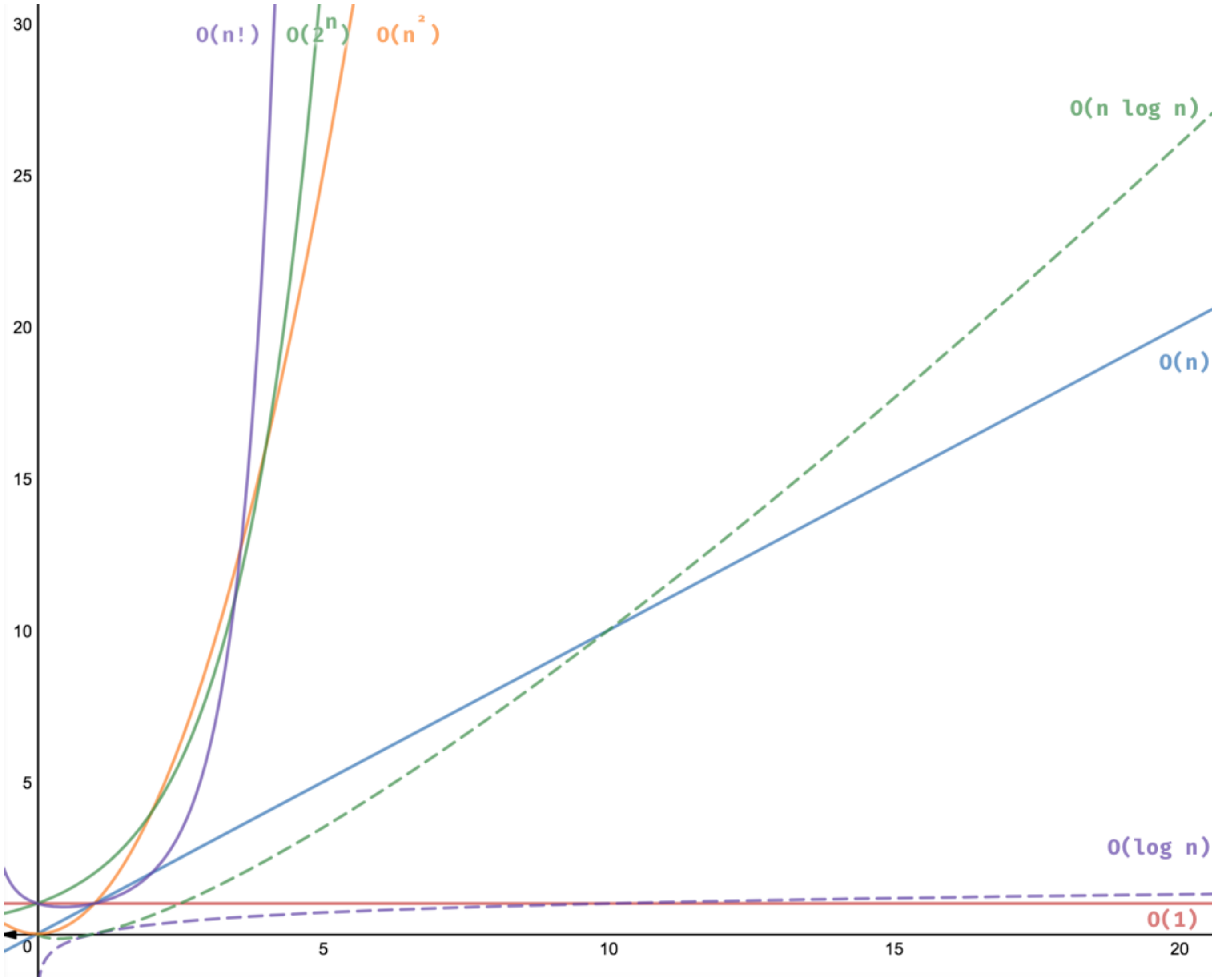
Algorithm Time Complexities (3)

Logarithmic Time complexity: O (logn)

```
int binarySearch(int A, int l, int r, int find){
if (r >= l)
{
int mid = l + (r - l)/2;
if (A[mid] == find) return mid;

if (A[mid] > x)
     return binarySearch(A, l, mid-1, find);
else
return binarySearch(arr, mid+1, r, x);
}
return -1;
}
int main(void)
{
int A[10]; int N=10;
int result = binarySearch(A, 0, n-1, find);
(result == -1)? printf("Not Found"): printf("Element at index %d", result);
return 0;}
```

Algorithm Time Complexities (4)



Assignment#3: *Circular Queue* using *Singly Linked List*

Objective: To Implement *Circular Queue* using *Singly Linked List*

Inputs: The input file will be a text file where each line represents an operation to be performed namely (*enqueue, dequeue, display*).

- Output: A file (e.g., ***output.txt***)
  - What the output file should contain?
    - The output file should contain the corresponding operations performed for each line provided in the input file

Assignment#3: *Circular Queue* using *Singly Linked List (1)*

Objective: To Implement *Circular Queue* using *Singly Linked List*

Inputs: The input file will be a text file where each line represents an operation to be performed namely (*enqueue, dequeue, display*).

- Output: A file (e.g., ***output.txt***)
  - What the output file should contain?
    - The output file should contain the operations performed for each line provided in the input file

```
enqueue 10
enqueue 15
enqueue -11
display
dequeue
display
```

input.txt

```
Inserted value: 10
Inserted value: 15
Inserted value: -11
Elements of the queue: 10 15 -11
deleted value: 10
Elements of the queue: 15 -11
```

output.txt

thank you!

email:
k.kondepu@iitdh.ac.in

NEXT Class: 09/05/2023