CS 203

Design & Analysis of Algorithms

Instructors: Dr. Ashish Phophalia
ashish_p@iiitvadodara.ac.in

General Info about CS 203

LTPC: 3-0-0-3

Lecture: as per email shared

Office Hours: Drop email for anything else

Course Grading:

- Mid-Sem Exam – 30%
- End Sem Exam – 45%
- Quizzes & Assignments – 25%
- Attendance: as per Institute Rule

General Info about CS 263

LTPC: 0-0-3-2

Lab: as per email shared

Office Hours: Drop email for anything else

Teaching Assistants: TBA

Course Grading:

- Continuous Lab Evaluation – 50%
- Continuous Lab Quizzes – 20%
- Mid Sem Viva - 10%
- Final Lab Exam & Viva – 20%

Reference Books:

1. T.H.Cormen, C.E.Leiserson, R.L.Rivest, C. Stein, Introduction to Algorithms, 3rd Ed., Prentice Hall India, 2010.
2. Jon Kleinberg & Eva Tardos, Algorith Design, Pearson Education, 1st Ed. 2006
3. D.E.Knuth, The Art of Computer Programming, Vols. 1-4, Addison Wesley, 1998.
4. A.V.Aho, J.E.Hopcroft, J.D.Ullman, Design and Analysis of Algorithms, Addison Wesley, 1976.
5. E.Horowitz, S.Sahani, Fundamentals of Computer Algorithms, Galgotia Publishers, 1984.
6. K.Melhorn, Data Structures and Algorithms, Vols.1 and 2, Springer Verlag, 1984.
7. P.W.Purdom, Jr. and C.A.Brown, The Analysis of Algorithms, Holt Rhinehart and Winston, 1985.
8. http://jeffe.cs.illinois.edu/teaching/algorithms/#book

Course Resources

- Offered as first formal course on algorithm around the globe in CS stream
- Available on leading platform like MIT OCW, Coursera…
- You are free to refer course website of other reputed universities/faculties

Why this course?

- Core course – can't escape
- Introduction to more complex and real-world problems with detailed analysis
- It will help to innovate and analyze new algorithms/procedures while doing research

Course Content

Available on website – B.Tech. 2018 Curriculum

Introduction and asymptotic notations: The role of algorithms in computing, Insertion sort, Analysing algorithms (Random-access machine (RAM) model), Designing algorithms, Asymptotic notations.

Divide and Conquer Techniques: Divide and conquer algorithms such as the maximum-subarray problem, Strassen's algorithm for matrix multiplication, etc., Solving recurrences -- The substitution method, The recursion-tree method, The master method, Proof of the master theorem.

Heapsort and Quicksort: Heaps, Maintaining the heap property, Building a heap, the heapsort algorithm, Priority queues, Quicksort.

Dynamic Programming: Dynamic programming algorithms such as the rod cutting, matrix-chain multiplication, Longest common subsequence, etc., Elements of dynamic programming – Optimal Substructure, Overlapping sub-problems.

Greedy Algorithms: Greedy algorithms such as activity-selection problem, huffman codes, etc., Elements of the greedy strategy - Optimal Substructure, Greedy choice property.

Graph Algorithms: Representations of graphs, Depth First search, Breadth First Search, Topological sort, Minimum Spanning Trees - The algorithms of Kruskal and Prim, Shortest Paths - The Bellman-Ford algorithm, Dijkstra's algorithm.

NP-Completeness and Approximation Algorithms – Introduction to NP-Completeness, Approximation algorithms such as the traveling-salesman problem, the subset-sum problem, etc.

Algorithm

1. What is an algorithm?
2. How to define it precisely?
3. How it differs from procedures & functions?
4. How it is different than a program?
5. I am best at programming then why I should study this course?

Change your viewpoint and solve problems not simply code...

Design an Algorithm

1. Why to Design?
2. What if already existing?
3. Who is going to use it?
4. Does it beneficial to end user?

Designing of an algorithm is art and science

Analyzing an Algorithm

1. Why to Analyze?

2. What if it fails?

3. How to make inference from such analysis?

4. What if my problem is scalable?

With Design, proof of correctness and complexity analysis is MUST!! Since you are the creator of it.

Additional Questions

1. It has to be always Unique?

2. If not, can we improvise it? Or where is the Optimal solution?

3. How to implement it? What about data structures?

4. How to ensure it working as desired?

Timing Analysis

Timing Analysis

- Timing analysis is indispensable for acceptance of an algorithm.
- Approach I: Exact Running Time (by implementing the algorithm) Analysis
- Approach II: Timing Analysis with Machine Constants
- Approach III: Order of Growth, for large inputs

Approach I:-Exact Running Time Analysis

- Compare algorithms based on their runtime
- Issues
  - Firstly, implement the algorithms
  - Running time is not absolute, determined by
    - Hardware (Machine)
    - Input
    - Programming Skill and Support (Compiler etc.)

APPROACH I:-EXACT RUNNING TIME ANALYSIS

main.cpp

```
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <chrono>

using namespace std;

int main(int argc, char **argv)
{
    auto start = std::chrono::steady_clock::now();
    
    int arr[100000];
    for(int i = 0; i < 100000; i++)
        arr[i] = rand();

    //find max
    int max_val = 0;
    
    for(int i = 0; i < 100000; i++)
        if (max_val < arr[i])
            max_val = arr[i];
    
    auto end = std::chrono::steady_clock::now();
    std::chrono::duration<double> elapsed_seconds = end-start;
    std::cout << "max value\t" << max_val << "\telapsed time: " << elapsed_seconds.count() << "s\n";
    
    return 0;
}
```

Analysis Running Time with Each Execution

- Running time is specific to each execution (run) of the program

```
jmayank@jmayank-VivoBook-ASUSLaptop-X430FA-S430FA:~/Desktop/Code$ g++ main.cpp
jmayank@jmayank-VivoBook-ASUSLaptop-X430FA-S430FA:~/Desktop/Code$ ./a.out
max value       2147469841      elapsed time0.00146074s
jmayank@jmayank-VivoBook-ASUSLaptop-X430FA-S430FA:~/Desktop/Code$ ./a.out
max value       2147469841      elapsed time0.00530637s
jmayank@jmayank-VivoBook-ASUSLaptop-X430FA-S430FA:~/Desktop/Code$ ./a.out
max value       2147469841      elapsed time0.00274776s
jmayank@jmayank-VivoBook-ASUSLaptop-X430FA-S430FA:~/Desktop/Code$ ./a.out
max value       2147469841      elapsed time0.00220558s
jmayank@jmayank-VivoBook-ASUSLaptop-X430FA-S430FA:~/Desktop/Code$ ./a.out
max value       2147469841      elapsed time0.00175753s
jmayank@jmayank-VivoBook-ASUSLaptop-X430FA-S430FA:~/Desktop/Code$ ./a.out
max value       2147469841      elapsed time0.00301175s
jmayank@jmayank-VivoBook-ASUSLaptop-X430FA-S430FA:~/Desktop/Code$
```

MACHINE DEPENDENT MODEL OF COMPUTATION

- Given two programs P1 and P2, we have machine M1 and M2.
  - In M1:- P1 is better than P2
  - In M2:- P2 is better than P1

We can't say anything, which program (Algorithm) is better.

TIMING ANALYSIS

- Timing analysis is indispensable for acceptance of an algorithm.

Running time as a function of the input size

- Input Size:-carefully defined

    -- can be #input

    -- #bits to represent input

    -- graph => #nodes, #edges

TIMING ANALYSIS

- Running time as a function of the input size
- Types of Analysis
  - Worst-case
    - Provides an upper bound on running time
    - We must know the case that causes a maximum number of operations to be executed.
    - An absolute guarantee that the algorithm would not run longer for any input
  -

Timing Analysis

- Average-case
  - Provides a prediction about the running time
  - Assumes that the input is random
  - Expected
- Best-case
  - Not much useful
  - Provides a lower bound on running time
  - Input is the one for which the algorithm runs the fastest, applicable for some input
  - We must know the case that causes a minimum number of operations to be executed.

LINEAR SEARCH

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
  - If x matches with an element, return the index.
  - If x doesn't match with any of elements, return -1.

10	13	18	25	32	24	31

arr[]

Worst , Average and Best Case Analysis?

Approach II: Timing analysis with Machine Independent Model and Machine Constants

Machine Independent Model of Computation

- We have to make abstract model.
- Consider M1 and M2 machines
  - Same instruction set
    - Count number of instructions steps
    - If they have faster clock cycle then one will execute faster than another. Only they differ in ratio.
  - Different Instruction set
    - Assume every instructions take constant of time.
    - Addition, Subtraction, Comparison, Multiplication,…

STEP COUNT METHOD: EXAMPLE 1

- Sum of array elements
- Algorithm Sum (A, n)

```
{
   S=0;
   for(i=0; i<n; i++)
   {
   S= S+A[i];
   }
   Return S
}
```

What would be the time and space complexity?

Step Count Method: Example 2

- Addition of two matrices
- Algorithm Sum (A, B, n)

```
{
   for(i=0; i<n; i++)
   {
        for(j=0; j<n; j++)
        {
        C[i, j]= A[i, j]+B[i, j]
        }
   }
   Return C
}
```

What would be the time and space complexity?

Step Count Method: Example 2

- Addition of multiplication of matrices
- Algorithm Multiply (A, B, n)

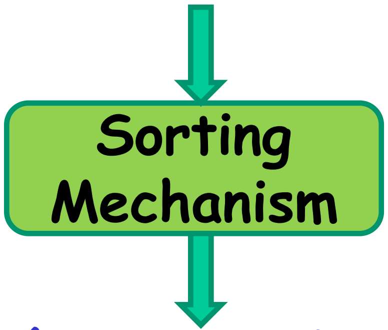
```
{
    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
        {
            C[i, j]=0;
            for(k=0; k<n; k++)
            {
                C[i, j]+= A[i, k]xB[k, j]
            }
        }
    Return C
}
```

What would be the time and space complexity?

TIMING ANALYSIS WITH STEP COUNT METHOD: INSERTION SORT

An example analysis of a sorting algorithm

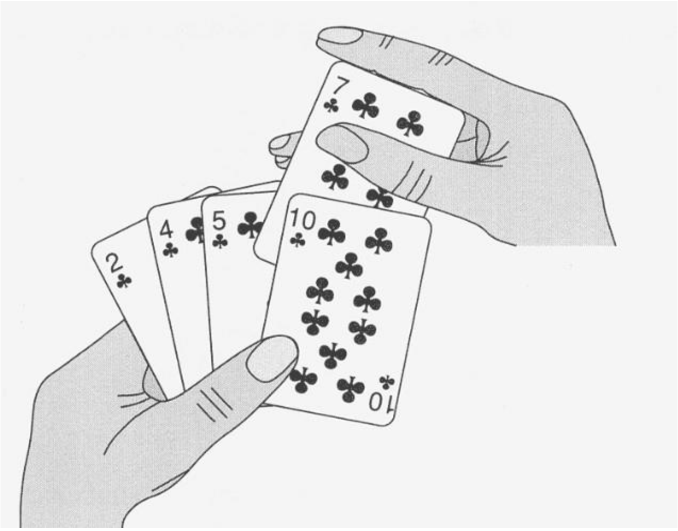
- **Sorting**
  - **Input:-** $\langle a_1,\ a_2 \ldots\ldots,\ a_n \rangle$



  - **Output:-** A permutation of $\langle a_1,\ a_2 \ldots\ldots,\ a_n \rangle$ such that

    $a_i \le a_{i+1}$, $0 \le i \le n-1$

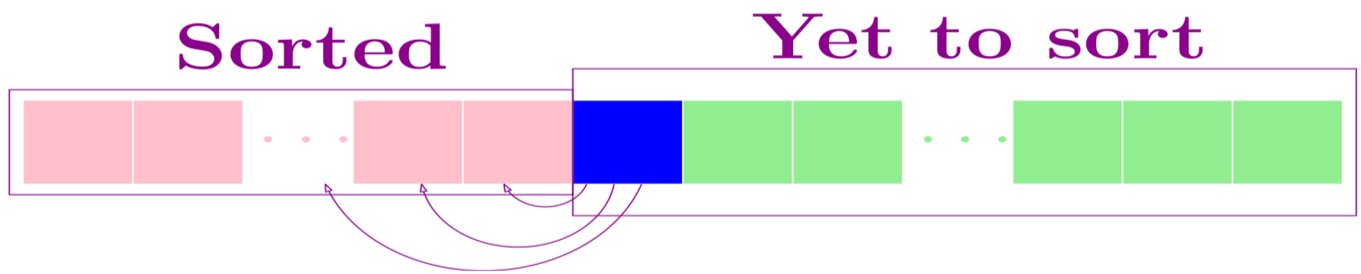
IDEA



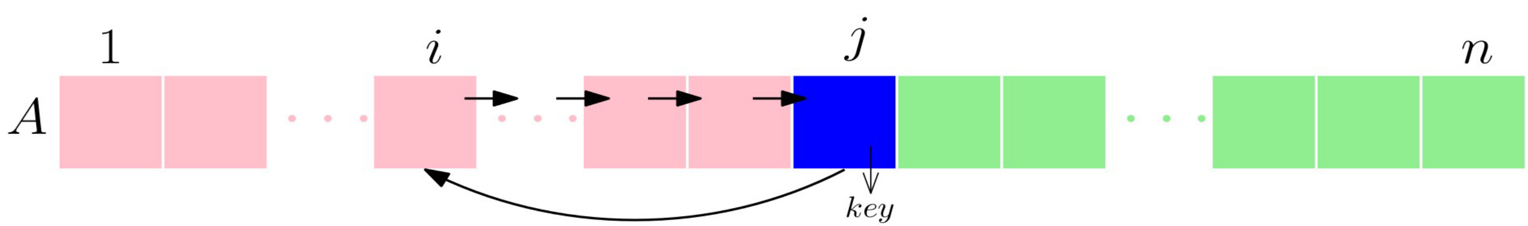
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
  - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted

IDEA

- Place (insert) the first (blue) unsorted element in the sorted (pink) subarray



- for *j* = 2 to n
  - place (insert) *A*[*j*] in the sorted subarray *A*[1:*j*-1]



Example

- 13 34 6 57 63 7

Insertion Sort Analysis: Step Count Method

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

t_j: # of times the while statement is executed at iteration j

Insertion Sort Analysis: Step Count Method

- Best Case [Sorted]

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

- T(n) = dn+e

INSERTION SORT ANALYSIS: STEP COUNT METHOD

- Worst Case [Reverse Sorted]

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8) .
\end{aligned}
$$

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

- $T(n) = an^2 + bn+c$

Analysis of Algorithm

- In general, we are not so much interested in the time and space complexity for small inputs.

- For example, while the difference in time complexity between linear and binary search is meaningless for a sequence with $n = 10$, but it is significant for $n = 2^{30}$

EXAMPLE

Consider two algorithms A and B that solve the same class of problems.

- The time complexity of A is 5,000n, the one for B is $\lceil 1.1^{n} \rceil$ for an input with n elements.
- For n = 10, A requires 50,000 steps, but B only 3, so B seems to be superior to A.
- For n = 1000, however, A requires 5,000,000 steps, while B requires 2.5×10^{41392}

Input Size (n)	Algorithm A=(5000n)	Algorithm B = $\lceil 1.1^{n} \rceil$
10	50000	3
100	5×10^{5}	13,781
1000	5×10^{6}	2.5×10^{41}
1000 000	5×10^{9}	2.5×10^{41392}

ANALYSIS OF ALGORITHM

- During design we are interested to measure the (relative) performance of algorithms for sufficiently larger input size
- Try to approximate the growth of running time as input size increases
  - More specifically, $n \rightarrow \infty$

Asymptotic Analysis

Asymptotic Analysis

Why not precise computation time analysis?

- Need to implement
- Machine/Input/Programming Support specific

WHY NOT STEP COUNT METHOD?

- Consider Linear Search O(an) and Binary Search (b log n).
- We run the Linear Search on a fast computer ***A*** and Binary Search on a slow computer **B**.
- Let's say the constant for ***A*** is 0.2 and constant for **B** is 1000.

n	0.2*n	1000 log n
10	2 sec	~38 min
100	20 sec	~1 hr
10^6	5.5 hr	~4 hr
10^9	6.3 years	~5 hr

- Lower order terms and constant terms does not impact much for sufficiently large input
- Overhead of considering all the terms

Concepts of order of growth and Asymptotic Notations are essential to understand.

ORDER OF GROWTH

- **Focus on the dominating terms**
  - Ignore lower order terms:- Does not matter much for significantly large input
  - Ignore constant multiplier:- Exact value differs by a constant factor

- **For insertion sort:** $an^2 + bn+c$
  - Ignore lower order terms=> an^2
  - Ignore constant multiplier=> n^2
- Meaningful (but inexact) analysis
- Specifically, worst-case running time ($an^2 + bn+c$) is not equal to n^2, rather it grows like n^2
- Running time is n^2 captures the notion that the order of growth is n^2
- Efficient way of analyzing (in fact, comparing the relative) performance of an algorithm

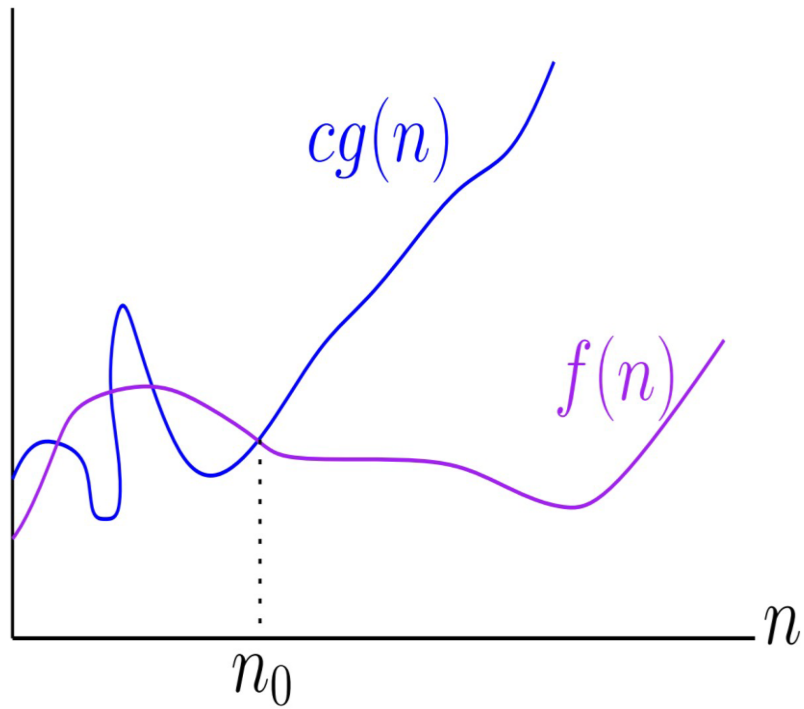
ASYMPTOTIC ANALYSIS

- Considering the order of growth for the larger input, we are studying the ***asymptotic*** efficiency of algorithms.
- It measure of the efficiency of algorithms that don't depend on machine-specific constants and doesn't require algorithms to be implemented and time taken by programs to be compared.

- How the running time of an algorithm increases with the size of the input.

Asymptotic Notations: O (Big Oh) Notation

- Asymptotic Upper Bound => Asymptotic "less than or equal to"
  - f(n) = O(g(n)) => f(n) "≤" g(n)
- O(g(n)) = {f(n) : there exist **positive** constants c and n0such that **0 ≤ f (n) ≤ cg(n)** for all **n≥n0**}



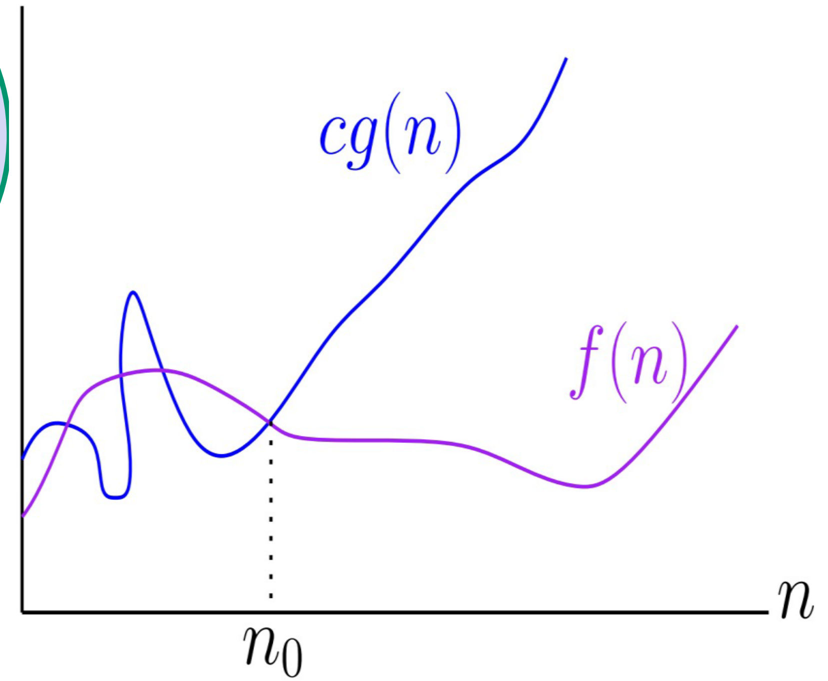
g(n) is an asymptotic upper bound of f (n)

O (Big Oh) Notation

$O(n^2)$
$2n^2-3n+4$
$3n+5$
30

$O(1)$
20
1000
10

$O(n)$
$3n-2$
$5n$
10



f grows not faster than g
f does not exceed g

- $3n^2 = O(n^3)$:
  - $3n^2 \leq cn^3 \Rightarrow 3 \leq cn \Rightarrow c = 1$ and $n0 = 3$

    (Also $c = 3$ and $n0 = 1$ or $c = 3.5$ and $n0 = 1$)
- $n^2 = O(n^2)$:
  - $n^2 \leq cn^2 \Rightarrow c \geq 1 \Rightarrow c = 1$ and $n0 = 1$
- $150n^2 + 200n = O(n^2)$:
  - $150n^2 + 200n \leq 150n^2 + n^2 = 151n^2$ (for $n \geq 200$)

  $\Rightarrow c = 151$ and $n0 = 200$
- $3n = O(n^2)$:
  - $3n \leq cn^2 \Rightarrow cn \geq 3 \Rightarrow c = 1$ and $n0 = 3$
    -

- **no unique pair of *n0* and *c***
- **To prove upper bound, find some *n0* and *c***