CS2102 Analysis and Design of Algorithms

Course Overview

M.Sc. Bryan Gonzales Vega
bgonzales.vega@gmail.com

University of Engineering and Technology

Lecture Content

1. Course Overview
    - Course Roadmap
    - Course Resources
    - Grading
    - Communication and Tools

2. Introduction
    - Asymptotic Analysis
    - Asymptotic Notation

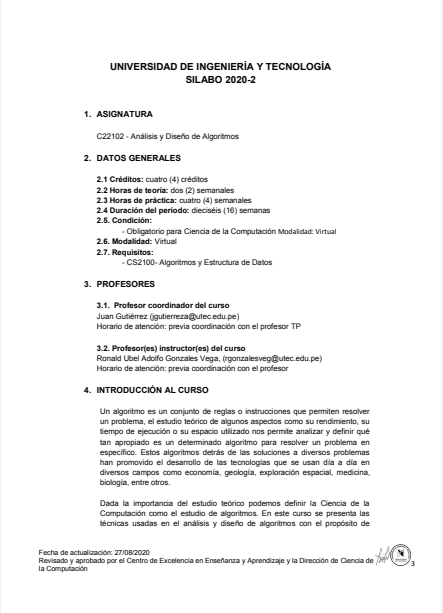
Course Overview

Roadmap

Course Foundation

- Elemental Data Structures
- Computer Theory
- Discrete Mathematics
- Probabilities
- Programming Paradigms

CS2102 - Analysis and Design of Algorithms [1]



UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA
SILABO 2020-2

1. ASIGNATURA

C22102 - Análisis y Diseño de Algoritmos

2. DATOS GENERALES

2.1 Créditos: cuatro (4) créditos
2.2 Horas de teoría: dos (2) semanales
2.3 Horas de práctica: cuatro (4) semanales
2.4 Duración del período: dieciséis (16) semanas
2.5. Condición:
- Obligatorio para Ciencia de la Computación Modalidad: Virtual
2.6. Modalidad: Virtual
2.7. Requisitos:
- CS2100- Algoritmos y Estructura de Datos

3. PROFESORES

3.1. Profesor coordinador del curso
Juan Gutiérrez (jgutierreza@utec.edu.pe)
Horario de atención: previa coordinación con el profesor TP

3.2. Profesor(es) instructor(es) del curso
Ronald Ubel Adolfo Gonzales Vega, (rgonzalesveg@utec.edu.pe)
Horario de atención: previa coordinación con el profesor

4. INTRODUCCIÓN AL CURSO

Un algoritmo es un conjunto de reglas o instrucciones que permiten resolver un problema, el estudio teórico de algunos aspectos como su rendimiento, su tiempo de ejecución o su espacio utilizado nos permite analizar y definir qué tan apropiado es un determinado algoritmo para resolver un problema en específico. Estos algoritmos detrás de las soluciones a diversos problemas han promovido el desarrollo de las tecnologías que se usan día a día en diversos campos como economía, geología, exploración espacial, medicina, biología, entre otros.

Dada la importancia del estudio teórico podemos definir la Ciencia de la Computación como el estudio de algoritmos. En este curso se presenta las técnicas usadas en el análisis y diseño de algoritmos con el propósito de

Fecha de actualización: 27/08/2020
Revisado y aprobado por el Centro de Excelencia en Enseñanza y Aprendizaje y la Dirección de Ciencia de la Computación

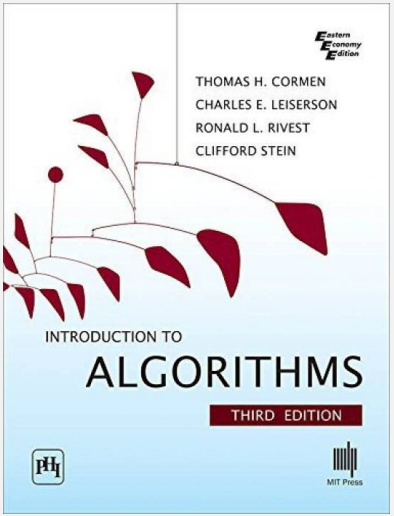
[1] Syllabus can be downloaded from Canvas

Books [II]

- Introduction to Algorithms [Cormen et al., 2009]
- The Art of Computer Programming [Knuth, 1997]
- Concrete Mathematics [Graham et al., 1989]

MIT Open Course Ware [III]

- Design and Analysis of Algorithms [Demaine et al., 2015]
- Introduction to Algorithms [Leiserson and Demaine, 2005]



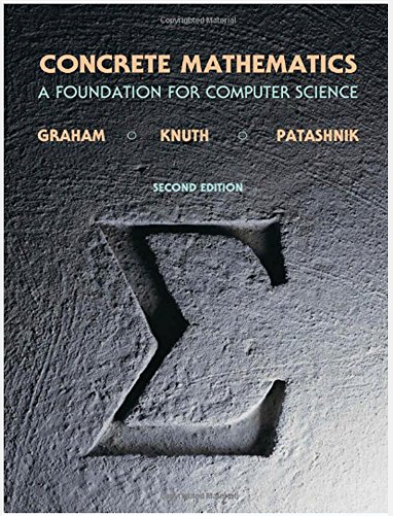
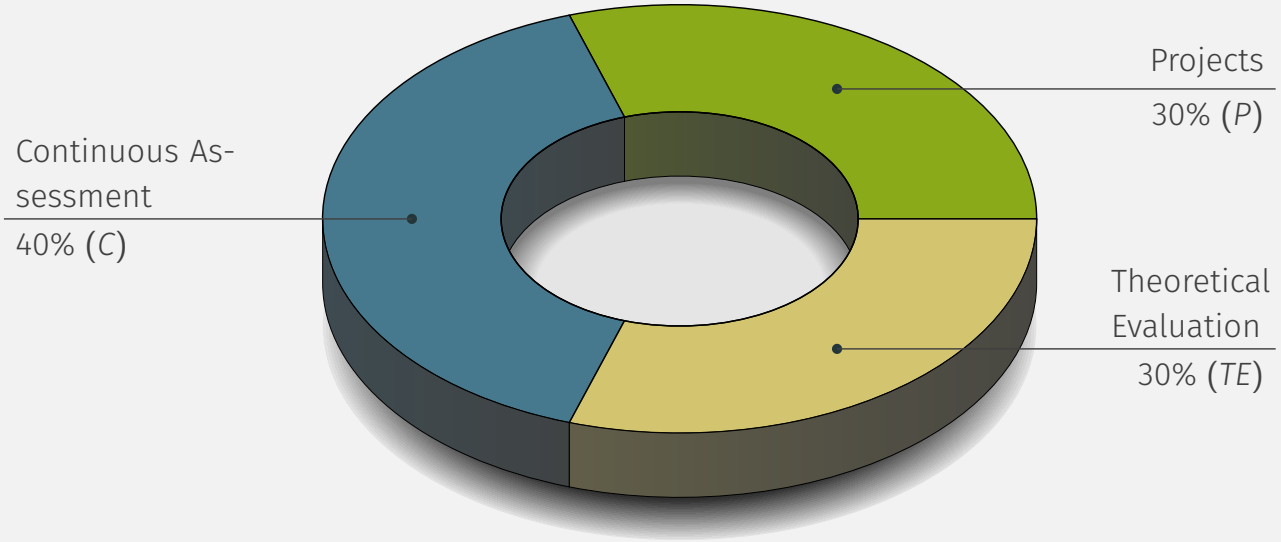


Figure 1: Algorithms (left) and mathematical foundation (right) references used in this course

[II] Complete list of books are detailed in the Syllabus. Additionally, you may find `https://teachyourselfcs.com` useful.

[III] Resources can also be obtained from other sources.

Grading



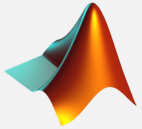
$$\textbf{Final Grade} = \underbrace{C_1(0.20) + P_1(0.15)}_{\text{1st half}} + \underbrace{C_2(0.20) + P_2(0.15)}_{\text{2nd half}} + TE(0.30)$$

Communication and Tools

Slack
https://utec-cs2102.slack.com

VSCode
Live Share Extension

Zoom
https://zoom.us [IV]



Matlab
Simulation and visualization

Mail
bgonzales.vega@gmail.com

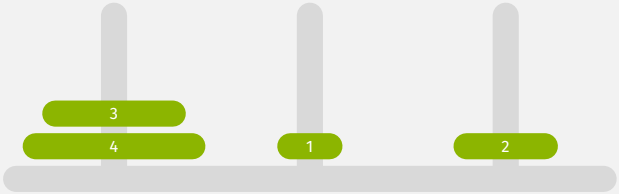
GitHub
Classroom, Automated Grading

BLACKOUT: Sunday, Monday and Tuesday

[IV] Specific links to each lecture will be posted on Canvas before the session.

Introduction

- Only one disk can be moved among the towers at any given time.
- Only the top disk can be removed.
- No large disk can sit over a small disk.



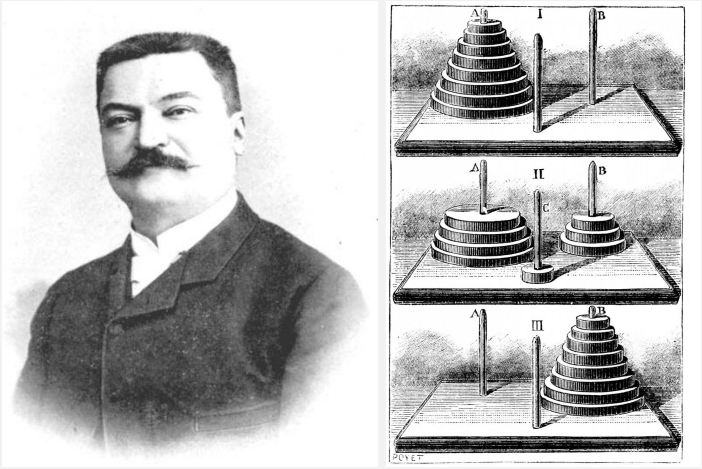


Figure 2: Edouard Lucas (left) and Tower of Hanoi (right) with 8 disks in a peg. Tower of Brahma generalization uses 64 disks.

How can we use **induction** to prove the solution of the Tower of Hanoi puzzle?

T_k = min # movements to move k disk

$$T_k = 2T_{k-1} + 1$$

$$\boxed{T_k = 2^k - 1}$$ assume as true

base case

$$T_k = \begin{cases} 0 & , k = 0 \\ 2T_{k-1} + 1 & , k > 0 \end{cases}$$

↳ recurrence equation

$$T_k = 2T_{k-1} + 1$$

$$= 2\left[2^{k-1} - 1\right] + 1$$

$$= 2^{k-1+1} - 2 + 1$$

$$\boxed{T_k = 2^k - 1}$$

Asymptotic Concerned about how the running time of an algorithm increases with the size of the input in the limit. Additionally, the **order of growth** or **rate of growth** allow us to compare the relative performance between algorithms.

i.e. $5n^2 + 4n + 5 = \Theta(n^2)$

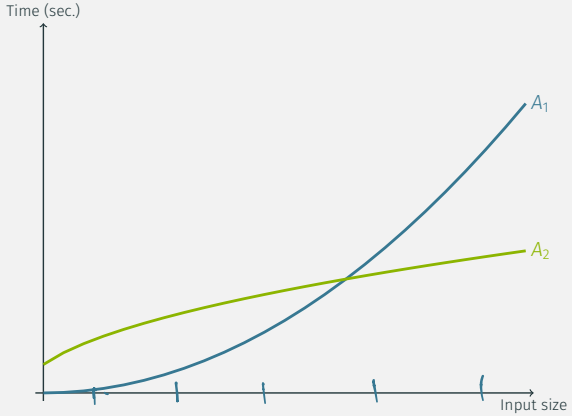


Figure 3: Empirical analysis.[V] Given 2 algorithms A_1 and A_2 we can run a set of simulations varying the input size and compare the running times of both algorithms.

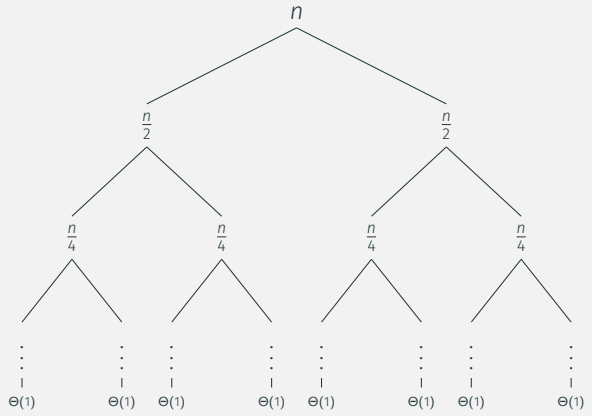


Figure 4: Theoretical analysis. Instead of running the algorithms in a machine, we can foresee the running time of the algorithms given an specific input.

[V] Not restricted to only 2 algorithms

Incremental Insertion sort implementation is an algorithm that can be analyzed calculating the total number of times each line is executed, isolating the machine-dependent constants.

Algorithm 1 Insertion sort with an array of *numbers* of size n

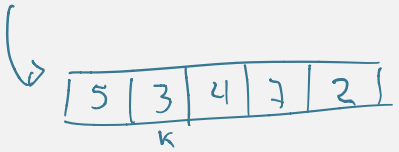
```
1: for i = 2 to n do
2:     key = numbers[i]
3:     j = i − 1
4:     while j > 0 and numbers[j] > key do
5:         numbers[j + 1] = numbers[j]
6:         j = j − 1
7:     end while
8:     numbers[j + 1] = key
9: end for
```

Loop Invariant

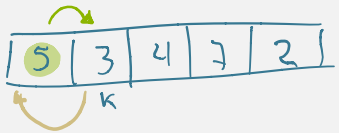
Initialization	It is true prior the first iteration of the loop
Maintenance	If it is true before an iteration of the loop, it remains true before the next iteration
Termination	When the loop terminates, the invariant help us to show the algorithm is correct

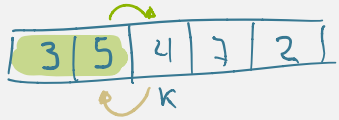
In every analysis we should consider and inspect the different input distributions specific to a particular problem or algorithm.

Cont'd - How insertion sort works

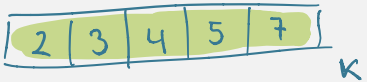


=> invariant is true before the loop





invariant is true during the loop

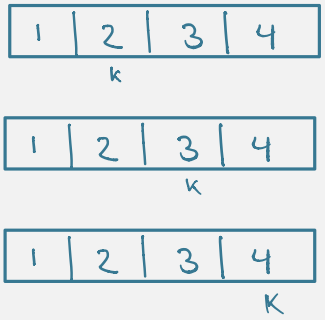


=> invariant is true after the loop

Loop invariant:

all elements before k are sorted

Best Case Analysis



Algorithm 1 Insertion sort with an array of *numbers* of size n

$C_1 \cdot n$	1:	**for** $i = 2$ to n **do**
$C_2 \cdot n-1$	2:	$key = numbers[i]$
$C_3 \cdot n-1$	3:	$j = i - 1$
$C_4 \cdot n-1$	4:	**while** $j > 0$ **and** $numbers[j] > key$ **do**
0	5:	$numbers[j + 1] = numbers[j]$
0	6:	$j = j - 1$
	7:	**end while**
$C_5 \cdot n-1$	8:	$numbers[j + 1] = key$
	9:	**end for**

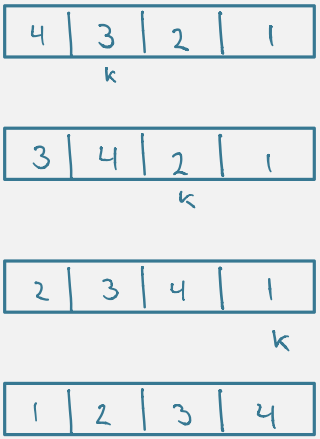
$$B(n) = C_1 \cdot n + C_2(n-1) + C_3(n-1) + C_4(n-1) + C_5(n-1)$$

$$B(n) = C_1 n + C_2 n - C_2 + C_3 n - C_3 + C_4 n - C_4 + C_5 n - C_5$$

$$B(n) = an + b = \Theta(n)$$

Cont'd - Insertion Sort Worst Case

Worst Case Analysis



Algorithm 1 Insertion sort with an array of *numbers* of size n

```
1: for i = 2 to n do
2:     key = numbers[i]
3:     j = i − 1
4:     while j > 0 and numbers[j] > key do
5:         numbers[j + 1] = numbers[j]
6:         j = j − 1
7:     end while
8:     numbers[j + 1] = key
9: end for
```

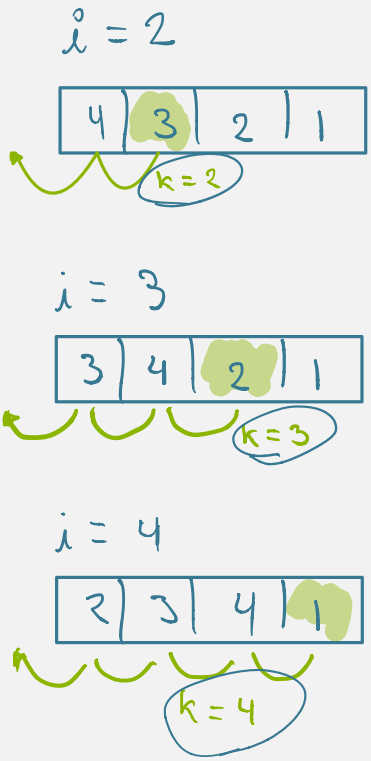
$c_1 \cdot n$
$c_2 \cdot n-1$
$c_3 \cdot n-1$
$\sum_{i=2}^{n} t_i$
$\sum_{i=2}^{n} t_{i-1}$
$c_7 \cdot n-1$

$\sum_{i=2}^{n} t_i \Rightarrow \frac{n(n+1)}{2} - 1$

$\sum_{i=2}^{n} t_{i-1} \Rightarrow \frac{n(n-1)}{2}$

$$W(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left[\frac{n(n+1)}{2} - 1\right] + c_5\left[\frac{n(n-1)}{2}\right] + c_6\left[\frac{n-1(n)}{2}\right] + c_7(n-1)$$

$$w(n) = \underline{a n^2 + b n + c} = \underline{\Theta(n^2)}$$



Algorithm 1 Insertion sort with an array of *numbers* of size n

```
1: for i = 2 to n do
2:     key = numbers[i]
3:     j = i − 1
4:     while j > 0 and numbers[j] > key do
5:         numbers[j + 1] = numbers[j]
6:         j = j − 1
7:     end while
8:     numbers[j + 1] = key
9: end for
```

$n-1$ $n-1$ $\sum_{i=2}^{n} t_i$ $\sum_{i=2}^{n} t_{i-1}$ $n-1$

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

$$\sum_{i=2}^{n} t_{i-1} = \sum_{i=2}^{n} i - 1 = \sum_{i=1}^{n-1} i \Rightarrow \frac{(n-1)(n-1+1)}{2}$$

$$\Rightarrow n(n-1)/2$$

Recursive merge sort implementation can be represented through a recurrence equation (a.k.a recurrence relation or recursion relation) and one way to solve it is guessing the solution and prove if that guess is correct, maybe through **mathematical induction**.

mergesort(n) ⟶ T(n)

Algorithm 2 Merge-Sort with an array of numbers A of size n from index p to r

1: **if** $p < r$ **then** Θ(1)
2: $q = \lfloor (p + r)/2 \rfloor$ Θ(1)
3: Merge-Sort(A,p,q) T(n/2)
4: Merge-Sort(A,$q + 1$,r) T(n/2)
5: Merge(A,p,q) Θ(n)
6: **end if**

Recurrence equation

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 2T(n/2) + \Theta(n), & n > 1 \end{cases} \quad (1)$$

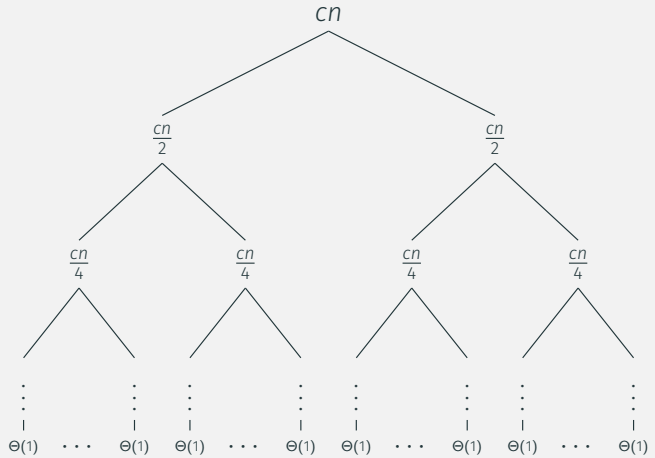
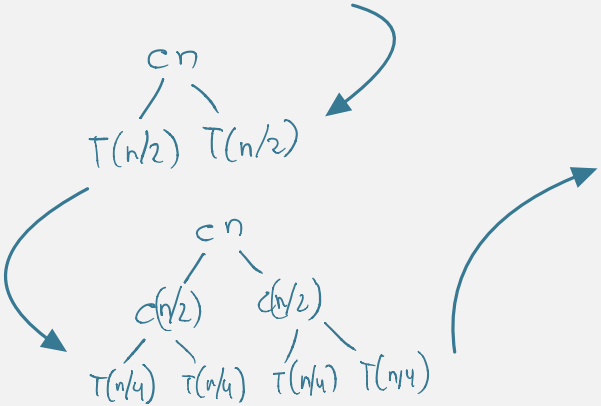


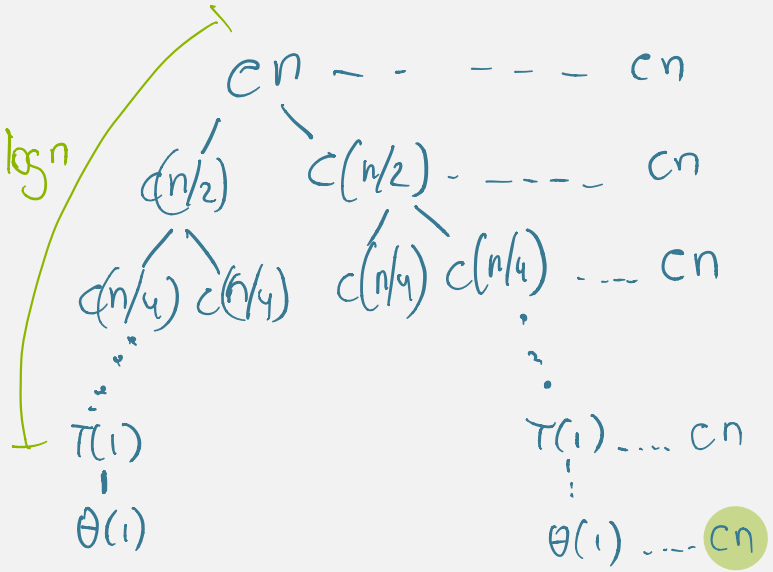
Figure 5: Recursion Tree. Graphical representation of the recursive calls of a recursive algorithm. It help us to see the big picture and approximate the solution but it shouldn't be used as a proof.

Cont'd - Recursion Tree

$T(n) = 2T(n/2) + \Theta(n)$

$T(n) = 2T(n/2) + c \cdot n$





$$T(n) = c \cdot n \log_2 n + c \cdot n = \Theta(n \log n)$$

The 3 notations presented below should be treated as descriptive representations instead of operations even if there is an equal sign.

Big-O Notation ($\mathcal{O}$)

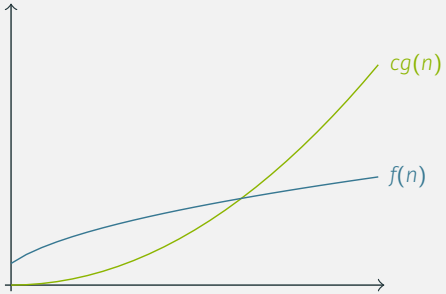


Figure 6: An upper bound is represented by $\mathcal{O}(g(n))$ where $f(n) : 0 \leq f(n) \leq cg(n)$.

$$f(n) = \mathcal{O}(g(n))$$

Theta Notation (Θ)

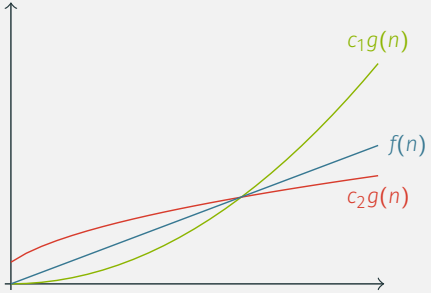


Figure 7: A Tight bound is represented by $\Theta(g(n))$ where $f(n) : 0 \leq c_2g(n) \leq f(n) \leq c_1g(n)$.

$$f(n) = \Theta(g(n))$$

Big-Omega Notation (Ω)

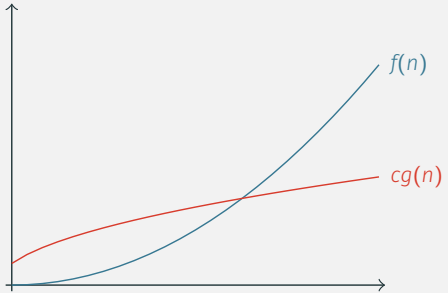


Figure 8: A Lower bound is represented by $\Omega(g(n))$ where $f(n) : 0 \leq cg(n) \leq f(n)$.

$$f(n) = \Omega(g(n))$$

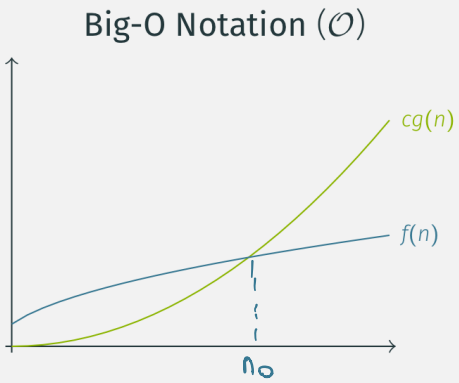


Figure 6: An upper bound is represented by $\mathcal{O}(g(n))$ where $f(n) : 0 \leq f(n) \leq cg(n)$.

$$f(n) = \mathcal{O}(g(n))$$

Sets definition :

$$O(g(n)) = \{ f(n), c>0, n \geq n_0$$
$$0 \leq f(n) \leq c \cdot g(n) \forall n \geq n_0 \}$$

$$n^2 \in O(n^3)$$
$$\in \downarrow$$
$$c \cdot n^3$$

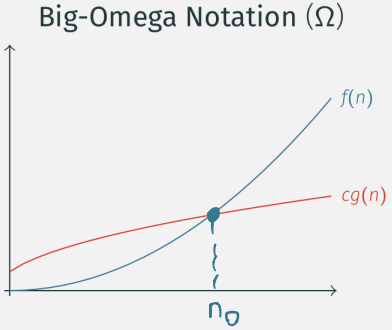


Figure 8: A Lower bound is represented by $\Omega(g(n))$ where $f(n) : 0 \leq cg(n) \leq f(n)$.

$$f(n) = \Omega(g(n))$$

Sets definition :

$$\Omega(g(n)) = \{ f(n), \exists c > 0, n_0 > 0 \quad 0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0 \}$$

$$\sqrt{n} = \Omega(\log n)$$

Theta Notation :

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

$$T(n) = \begin{cases} 4T(n/2) + n & , n > 1 \\ \Theta(1) & , n = 1 \end{cases}$$

$T(n) = \mathcal{O}(n^2)$ // guess

$T(k) \leq c \cdot k^2 \;, k < n$ // assume

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\leq 4\left[c\left(\frac{k}{2}\right)^2\right] + n \\ &\leq 4c\left(\frac{k}{2}\right)^2 + n \\ &\leq 4\frac{ck^2}{4} + n \\ &\leq ck^2 + n \\ &\leq ck^2 - (-n) \end{aligned}$$

$-n \geq 0$

doesn't close the induction $\Leftarrow$

Based on the topics presented in this lecture about asymptotic analysis and asymptotic notation, answer the following questions:

I. Why the $\Theta(n^2)$ bound on the worst-case running time of insertion sort doesn't imply a $\Theta(n^2)$ bound on the running time on every input?

II. Consequently with the question above, why is not precise to declare that the running time of insertion sort is $\mathcal{O}(n)$?

III. About the Brahma Tower generalization, which has 64 disks, what is the amount of time it takes to move the disks from one peg to other if every movement takes 1 second?

IV. Since asymptotic analysis is about the behavior in the limit, what is the relation between the mathematical limit and the asymptotic notations seen so far?

Questions?

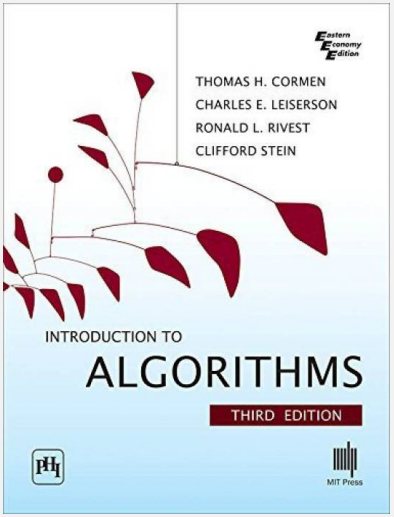
Next Lectures

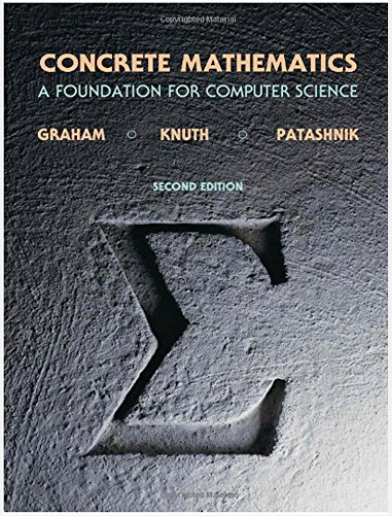
Introduction to Algorithms
[Cormen et al., 2009]

- Chap 02: Getting Started
- Chap 03: Growth of functions

Concrete Mathematics
[Graham et al., 1989]

- Chap 01: Recurrent Problems
- Chap 02: Sums





Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009).
Introduction to algorithms.
MIT press.

Demaine, E., Devadas, S., and Lynch, N. (2015).
Design and Analysis of Algorithms.
https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015.
[Online; accessed Jan 21, 2020].

Graham, R., Graham, R., Knuth, D., Knuth, D., and Patashnik, O. (1989).
Concrete Mathematics: A Foundation for Computer Science.
A foundation for computer science. Addison-Wesley.

Knuth, D. E. (1997).
The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms.
Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.


Leiserson, C. and Demaine, E. (2005).
Introduction to Algorithms (SMA 5503).
https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005.
[Online; accessed Jan 21, 2020].

Cont’d