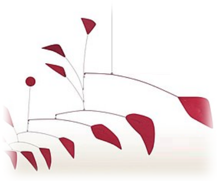


EECS 3101 - Design and Analysis of Algorithms

Shahin Kamali

Topic 1 - Introductions

York University

Picture is from the cover of the textbook CLRS.

Introduction

In a Glance . . .

- Algorithms are
  - Practical
  - Diverse
  - Fun (really!)
- Let's 'learn & play' algorithms and enjoy ...

Formalities

Textbook

- The main reference (optional):
  - Introduction to Algorithms, forth edition, by Cormen, Leiserson, Rivest, and Stein, MIT Press, 2024.
- Optional optional textbooks:
  - Algorithms and Data Structures, by Mehlhorn and Sanders, Springer, 2008.
  - The Algorithm Design Manual, second edition, by Skiena, Springer, 2008.
  - Advanced Data Structures, by Brass, Cambridge, 2008.

Grading

- There will be:
  - Five assignments
  - Two quizzes
  - A midterm exam
  - A final exam

Theorem

The focus of this course is on learning, practising, and discovering.

Corollary

Having fun in the process is important.

Grading (cntd.)

- Five assignments:
  - 5 to 10 percent extra for bonus questions.
  - submit only pdf files (preferably use LaTeX) on Crowdmark (`https://www.crowdmark.com/`).
- Quizzes, Midterm & Final exams:
  - there will be extra for bonus questions in midterm and final.
  - all are closed-book.
  - sample exams will be provided for practice for midterm and final.

Prerequisites

- What I have learned from previous courses?
- Basic sorting algorithms, e.g., quick sort and merge sort.
- Asymptotic notations, e.g., big O, Ω, etc.
- Basic abstract data types (ADTs) and data structures
    - Stacks, queues, dictionaries, binary search trees, hash tables, graphs.
- Basic algorithm families
    - Greedy algorithms, divide & conquer (d&c)
- Analysis techniques
    - E.g., how to analyse time complexity of a d&c algorithm?
    - Solving recursions, Master theorem, etc.

Algorithms

Algorithms

- What is an algorithm?

Definition

An algorithm is a computational procedure formed by a sequence of instructions (steps) to solve a problem.

  - The problem has an **input** and often requires an **output**.
  - Transition from one step to another can be **deterministic** or **randomized**.
    - The algorithm is deterministic if it never uses randomization; otherwise, it is a randomized algorithm.
  - Solving the problem requires the algorithm to **terminate**.
    - **Time complexity** concerns the number of steps that it takes for the algorithm to terminate (often on the worst-case input).

Abstract Data Type

- What is an Abstract Data Type (ADT)?

Definition

An abstract data type is formed by I) a set of values (data items) and II) a set of operations allowed on these items.

- Stack is an ADT. Data items can be anything and operations are *push* and *pop*.
- An ADT is abstract way of looking at data (no implementation is prescribed).
- An ADT is the way data 'looks' from the view point of user.

Data Structure

- What is a Data Structure?

Definition

A data structure is a concrete representation of data, including how data is organized, stored, and accessed on a computer.

- A linked-list is a data structure.
- Data structures are **implementations** of ADTs.
- A data structure is the way data ‘looks’ from the view point of implementer.

ADTs vs Data Structures

- ADTs: Stacks, queues, priority queues, dictionaries
- Data structures array, linked-list, binary-search-tree, binary-heap hash-table-using-probing, hash-table-using-chaining, adjacency list, adjacency matrix, etc.

Asymptotic Analysis

Algorithms (review)

- An **algorithm** is a step-by-step procedure carrying out a computation to solve an arbitrary instance of a problem.
  - E.g., sorting is a problem; a set of numbers form an instance of that and 'solving' involves creating a sorted output.
- A **program** is an implementation of an algorithm using a specific programming language.
- For a given problem (e.g., sorting) there can be several algorithms (e.g., Quicksort, Mergesort), and for a given algorithm (e.g., Quicksort) there can be several programs.
  - Our focus in this course is on algorithms (not programs).
  - How to implement a given algorithm relates to the art of **performance engineering** (writing a fast code)!

Algorithms Design & Analysis

- Given a problem P, we need to
  - Design an algorithm A that solves P (**Algorithm Design**).
  - Verify **correctness** and **efficiency** of the algorithm (**Algorithm Analysis)**.
  - If the algorithm is correct and efficient, **implement** it.
    - If you implement something that is not necessarily correct or efficient in all cases, that would be a **heuristic**.

Algorithm Evaluation

- How should we evaluate different algorithms for solving a problem?
    - In this course we are mainly concerned with amount of **time** it takes to solve a problem (this is called **running time**).
    - We can think of other measures such as the amount of **memory** that is required by the algorithm.
    - Other measures include amount of data movement, network traffic generated, etc.
- The amount of time/memory/traffic required by an algorithm depend on the **size** of the problem.
    - Sorting a larger set of numbers takes more time!

Running Time of Algorithms

- How to assess the running time of an algorithm?
- **Experimental analysis:**
  - Implement the algorithm in a program.
  - Run the program with inputs of different sizes.
  - Experimentally measure the actual running time (e.g., using *clock()* from time.h).
- Shortcomings of experimental studies:
  - We need to implement the program (what if we are lazy and those engineers are hard to employ?)
  - We cannot test all input instances for the problem. What are the good samples? (remember the Morphy's law)
  - Many factors have impact on experimental timing, e.g., hardware (processor, memory), software environment (operating system, compiler, programming language), and human factors (how good was the programmer?)

Computational Models

- We need to assess time/memory requirement of algorithms using models that
  - take into account all input instances.
  - do not require implementation of the algorithms.
  - are independent of hardware/software/programmer.
- In order to achieve this, we:
  - Express algorithms using **pseudo-codes** (don't worry about implementation).
  - Instead of measuring time in seconds, count the number of **primitive operations**.
    - This requires an abstract **model of computation**.

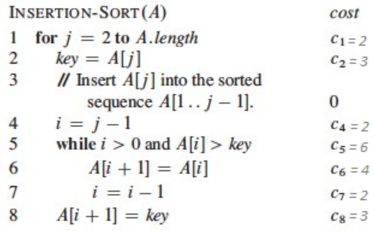
Random Access Machine (RAM) Model

- The **random access machine** (RAM):
  - Has a set of memory cells, each storing one ‘word’ of data.
  - Any **access to a memory location** takes constant time.
  - Any **primitive operation** takes constant time.
  - The **running time** of a program can be computed to be the number of memory accesses plus the number of primitive operations.
- Word-RAM is a RAM machine with the extra assumption that all values in our problem can ‘fit’ in a constant number of words (values are not too big).
- We often use Word-RAM model for analysis of algorithms.

Observation

RAM is a simplified model which only provides an approximation of a ‘real’ computer.

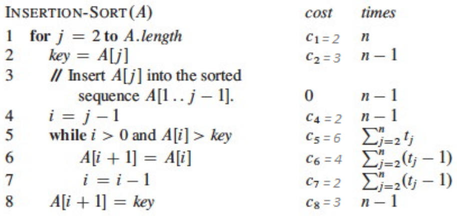
Analysis of Insertion Sort under RAM



```
INSERTION-SORT(A)                                  cost
1  for $j = 2$ to $A.length$                        $c_1 = 2$
2      $key = A[j]$                                 $c_2 = 3$
3      // Insert $A[j]$ into the sorted
           sequence $A[1 \,..\, j-1]$.              0
4      $i = j - 1$                                  $c_4 = 2$
5      while $i > 0$ and $A[i] > key$               $c_5 = 6$
6          $A[i+1] = A[i]$                          $c_6 = 4$
7          $i = i - 1$                              $c_7 = 2$
8      $A[i+1] = key$                               $c_8 = 3$
```

- First, calculate the ‘cost’ (sum of memory accesses and primitive operations) for each line.
    - E.g., in line 5, there are 3 memory accesses and 3 primitive operations.

Analysis of Insertion Sort under RAM



INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	$c_1 = 2$	n
2 $\quad key = A[j]$	$c_2 = 3$	$n-1$
3 $\quad$ // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $\quad i = j - 1$	$c_4 = 2$	$n-1$
5 $\quad$ **while** $i > 0$ and $A[i] > key$	$c_5 = 6$	$\sum_{j=2}^{n} t_j$
6 $\qquad A[i+1] = A[i]$	$c_6 = 4$	$\sum_{j=2}^{n} (t_j - 1)$
7 $\qquad i = i - 1$	$c_7 = 2$	$\sum_{j=2}^{n} (t_j - 1)$
8 $\quad A[i+1] = key$	$c_8 = 3$	$n-1$

- Next, find the number of times each line is executed.
  - This depends on the input, we may consider best or worst case input.
  - Let t_j be number of times the *while* loop is executed for inserting the j'th item.
    - In the best case, $t_j = 1$ and in the worst case $t_j = j$.
  - Summing up all costs, in the best case we have $T(n) = an + b$ for constant a and b.
  - In the worst case, we have $T_n = \alpha n^2 + \beta n + \gamma$ for constant α, β, γ.

Primitive Operations

- RAM model implicitly assumes primitive operations have fairly similar. running time
- Primitive operations:
    - basic integer arithmetic (addition, subtraction, multiplication, division, and modulo)
    - bitwise logic and bit shifts (logical AND, OR, exclusive-OR, negation, left shift, and right shift)
- Non-primitive operations:
    - exponentiation, radicals (square roots), logarithms, trigonometric, functions (sine, cosine, tangent), etc.

Asymptotic Notations

Statement

So, we can express the cost (running time) of an algorithm A for a problem of size n as a function $T_A(n)$.

- How do we compare two different algorithms? say $T_A(n) = \frac{1}{1000}n^3$ and $T_B(n) = 1000n^2 + 500n + 200$.
- Summarize the time complexity using asymptotic notations!
- Idea: assume the size of input grows to infinity; identify which component of $T_A(n)$ contributes most to the grow of $T_A(n)$.
- As n grows:
  - constants don't matter (e.g., $T_A(n) \approx n^3$).
  - low-order terms don't matter (e.g., $T_B(n) \approx 1000n^2$).

Asymptotic Notations

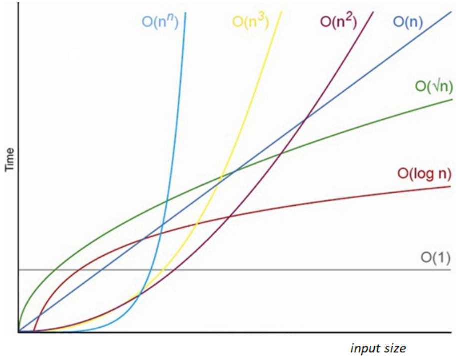
- Informally $T_B(n) = O(T_A(n))$ means T_B is **asymptotically smaller than or equal** to T_A.
- Is it sufficient to define O so that we have $T_B(n) < T_A(n)$?
  - No because the inequality might not hold for small values of n which we don't care about.
  - The two function might have constants we would prefer to ignore.

Definition

$$\mathrm{f(n)} \in \mathrm{O(g(n))} \Leftrightarrow$$

$$\exists M > 0, \exists n_0 > 0 \text{ s.t. } \underbrace{\forall n > n_0}_{\text{ignore low-order terms}}, \underbrace{f(n) \leq M \cdot g(n)}_{\text{ignore constants}}$$

Big Oh Illustration



https://apelbaum.wordpress.com/2011/05/05/big-o/

- Let $f(n) = 1000n^2 + 1000n$ and $g(n) = n^3$. Prove $f(n) \in O(g(n))$

Review

- To analyze running time of an algorithm (under RAM model) we sum the number of primitive operations and memory accesses of the algorithm.
- The cost (running time) of algorithm A for a problem of size n would be a function $T_A(n)$.
- How do we compare two different algorithms? say $T_A(n) = \frac{1}{1000}n^3$ and $T_B(n) = 1000n^2 + 500n + 200$.
- Summarize the time complexity using asymptotic notations!
- Idea: assume the size of input grows to infinity; identify which component of $T_A(n)$ contributes most to the grow of $T_A(n)$.
- As n grows:
    - constants don't matter.
    - low-order terms don't matter.

Big O Notations

- Informally $f(n) = O(g(n))$ means f is **asymptotically smaller than or equal** to g.

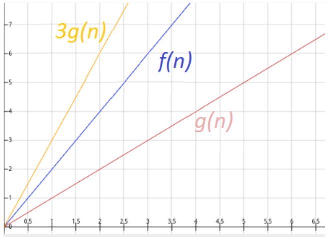
Definition

$$\mathrm{f(n)} \in \mathrm{O(g(n))} \Leftrightarrow$$

$$\exists M > 0, \exists n_0 > 0 \text{ s.t. } \underbrace{\forall n > n_0}_{\text{ignore low-order terms}}, \underbrace{f(n) \leq M \cdot g(n)}_{\text{ignore constants}}$$

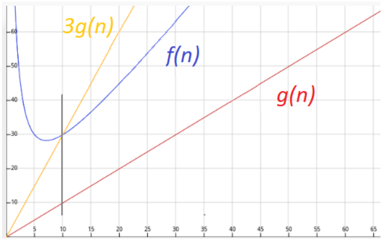
Big O Notations

- E.g., $f(n) = 2n$, $g(n) = n$. Is it that $f(n) \in O(g(n))$?
  - Yes, $f(n)$ is asymptotically smaller than or equal (equal) to $g(n)$.
  - To prove, we should show $\exists M > 0, \exists n_0 > 0$ s.t.$\forall n > n_0, f(n) \leq M \cdot g(n)$
  - It suffices to define $n_0 = 1$ and $M = 3$, we have $\forall n > 1, 2n \leq 3n$.
  - M could be any number larger than or equal to 2, and n_0 could be any number.
- We require specific values of M (not all choices for M work).



Big O Notations

- E.g., $f(n) = 2n + 100/n$, $g(n) = n$. Is it that $f(n) \in O(g(n))$?
    - Yes, again, $f(n)$ is asymptotically smaller than or equal (equal) to $g(n)$.
    - To prove, we should show $\exists M > 0, \exists n_0 > 0 \text{ s.t.} \forall n > n_0, f(n) \leq M \cdot g(n)$.
    - It suffices to define $n_0 = 10$ and $M = 3$, we have $\forall n > 10, 2n + 100/n \leq 3n$.
- We require specific values of M and n_0 (not all choices work).



Big O Notation

- Let $f(n) = 2023n^2 + 1402n$ and $g(n) = n^3$. Prove $f(n) \in O(g(n))$.
- We should define M and n_0 s.t. $\forall n > n_0$ we have $2019n^2 + 1397n \leq Mn^3$. This is equivalent to $2023n + 1402 \leq Mn^2$.
- We have $2023n + 1402 \leq 2023n + 1402n = 3425n$. So, to prove $2023n + 1402 \leq Mn^2$, it suffices to prove $3425n \leq Mn^2$, i.e., $3425 \leq Mn$. This is always true assuming $M = 1$ and $n \geq 3425$ ($n_0 = 3425$).
- Setting $M = 3426$ and $n_0 = 1$ also work!

Little o Notations

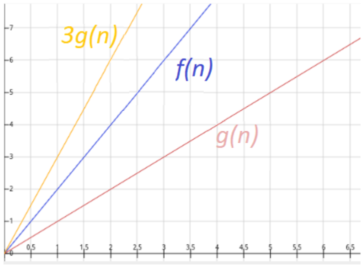
- Informally $f(n) = o(g(n))$ means f is **asymptotically smaller than** g.

Definition

$$\mathrm{f(n)} \in \mathrm{o(g(n))} \Leftrightarrow$$
$$\forall M > 0, \exists n_0 > 0 \text{ s.t. } \underbrace{\forall n > n_0}_{\text{ignore low-order terms}}, f(n) < M \cdot g(n)$$

Little o Notations

- E.g., $f(n) = 2n$, $g(n) = n$. Is it that $f(n) \in o(g(n))$?
  - No because for $M = 1$, it is not true that $f(n) < Mg(n)$ (i.e., $2n < n$) for large values of n.



Little o Notation

- Prove that $n^2 \sin(n) + 1984n + 2016 \in o(n^3)$.
    - We have to prove that for all values of M there is an n_0 so that for $n > n_0$ we have $n^2 \sin(n) + 1984n + 2016 < Mn^3$.
    - We know $n^2 \sin(n) \leq n^2$, $1984n \leq 1984n^2$ and $2016 \leq 2016n^2$. So, $n^2 \sin(n) + 1984n + 2016 \leq (1 + 1984 + 2016)n^2 = 4001n^2$.
    - So, to prove $n^2 \sin(n) + 1984n + 2016 < Mn^3$ it suffices to prove $4001n^2 < Mn^3$, i.e., $4001/M < n$, so, we can define n_0 to be any value larger than $4001/M$.
- For little o, n_0 is often defined as a function of M.

Big Ω Notation

- $f(n) = \Omega(g(n))$ means f is **asymptotically larger than or equal to** g.

Definition

$$f(n) \in \Omega(g(n)) \Leftrightarrow \exists M > 0, \exists n_0 > 0 \text{ s.t.} \forall n > n_0, f(n) \geq M \cdot g(n)$$

- Let $f(n) = n/2020$ and $g(n) = log(n)$. Prove $f(n) \in \Omega(g(n))$.
  - We need to provide M and n_0 so that for all $n \geq n_0$ we have $n/2020 \geq M \log(n)$, i.e., $n \geq 2020M \log(n)$.
  - We know $\log(n) < n$ (assuming $n > 1$). So, in order to show $2020M \log(n) \leq n$, it suffices to have $2020M \leq 1$, i.e., M can be any value smaller than $1/2020$ (and n_0 can be 1 or any other positive integer).

Little ω Notation

- $f(n) = \omega(g(n))$ means f is **asymptotically larger than** g.

Definition

$$\mathrm{f(n)} \in \omega(\mathrm{g(n)}) \Leftrightarrow \forall M > 0, \exists n_0 > 0 \text{ s.t.} \forall n > n_0, f(n) > M \cdot g(n)$$

- Let $f(n) = n/2020$ and $g(n) = log(n)$. Prove $f(n) \in \omega(g(n))$.
  - For any constant M we need to provide n_0 so that for all $n \geq n_0$ we have $n/2020 > M \log(n)$, i.e., $n > 2020M \log(n)$.
  - We know $log(n) < \sqrt{n}$ (assuming $n > 16$). So, in order to show $2020M \log(n) < n$, it suffices to have $2020M\sqrt{n} < n$, i.e., $2020M < \sqrt{n}$. For that, it suffices to have $(2020M)^2 < n$, i.e., n_0 can be defined as $\max\{16, (2020M)^2\}$.
- Similarly to little o, for ω, we often need to define n_0 as a function of M.

Θ Notation

- Informally $f(n) = \Theta(g(n))$ means f is **asymptotically equal to** g.

Definition

$\mathrm{f(n)} \in \Theta(g(n)) \Leftrightarrow$

$\exists M_1, M_2 > 0, \exists n_0 > 0 \text{ s.t.} \forall n > n_0, M_1 \cdot g(n) \leq f(n) \leq M_2 \cdot g(n)$

- Let $f(n) = n$ and $g(n) = n/2020$. Prove $f(n) \in \Theta(g(n))$.
  - We need to provide M_1, M_2, n_0 so that for all $n \geq n_0$ we have $M_1 \; n/2020 \leq n \leq M_2 \; n/2020$.
  - For the first inequality, we can have $M_1 = 1$ and for all n we have $n/2020 \leq n$.
  - For the second inequality, we let M_2 to be any constant larger than 2020 which gives $M_2/2020 \geq 1$.
  - n_0 can be any value, e.g., $n_0 = 1$.

Asymptotic Notations in a Nutshell

Definition

f(n) $\in$ O(g(n)) $\Leftrightarrow \exists M > 0, \exists n_0 > 0$ s.t.$\forall n > n_0, f(n) \leq M \cdot g(n)$

Definition

f(n) $\in$ o(g(n)) $\Leftrightarrow \forall M > 0, \exists n_0 > 0$ s.t.$\forall n > n_0, f(n) < M \cdot g(n)$

Definition

f(n) $\in \Omega(g(n)) \Leftrightarrow \exists M > 0, \exists n_0 > 0$ s.t.$\forall n > n_0, f(n) \geq M \cdot g(n)$

Definition

f(n) $\in \omega$(g(n)) $\Leftrightarrow \forall M > 0, \exists n_0 > 0$ s.t.$\forall n > n_0, f(n) > M \cdot g(n)$

Definition

f(n) $\in \Theta(g(n)) \Leftrightarrow \exists M_1, M_2 > 0, \exists n_0 > 0$ s.t.
$\forall n > n_0, M_1 \cdot g(n) \leq f(n) \leq M_2 \cdot g(n)$

Common Growth Rates

- $\Theta(1) \rightarrow$ constant complexity
  - e.g., an algorithms that only samples a constant number of inputs
- $\Theta(\log n) \rightarrow$ logarithmic complexity
  - Binary search
- $\Theta(n) \rightarrow$ linear complexity
  - Most practical algorithms :)
- $\Theta(n \log n) \rightarrow$ pseudo-linear complexity
  - Optimal comparison based sorting algorithms, e.g., merge-sort
- $\Theta(n^2) \rightarrow$ Quadratic complexity
  - naive sorting algorithms (bubble sort, insertion sort)
- $\Theta(n^3) \rightarrow$ Cubic Complexity
  - naive matrix multiplication
- $\Theta(2^n) \rightarrow$ Exponential Complexity
  - The 'algorithm' terminates but the universe is likely to end much earlier even if $n \approx 1000$.

Techniques for Comparing Growth Rates

- Assume the running time of two algorithms are given by functions $f(n)$ and $g(n)$ and let

$$L = \lim_{n\to\infty} \frac{f(n)}{g(n)}.$$

Then

$$f(n) \in \begin{cases} o(g(n)) & \text{if } L = 0 \\ \Theta(g(n)) & \text{if } 0 < L < \infty \\ \omega(g(n)) & \text{if } L = \infty \end{cases}$$

.

  - If the limit is not defined, we need another method.

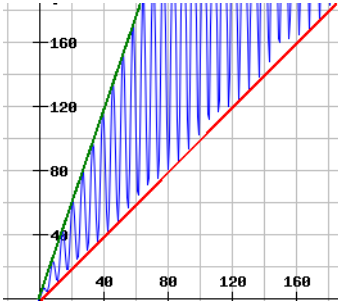
- Note that we cannot compare two algorithms using big O and Ω notations.
  - E.g., algorithm A can have complexity $O(n^2)$ and algorithm B has complexity $O(n^3)$. We **cannot** state that A is faster than B (why?)

Fun with Asymptotic Notations

- Compare the grow-rate of $\log n$ and n^r where r is a positive real number.

Fun with Asymptotic Notations

- Prove that $n(sin(n) + 2)$ is $\Theta(n)$.
- Use the definition since the limit does not exist.
  - Define n_0, M_1, M_2 so that $\forall n > n_0$ we have $M_1 n(sin(n) + 2) \leq n \leq qM_2 n(sin(n) + 2)$.
  - $M_1 = 1/3, M_2 = 1, n_0 = 1$ work!



Fun with Asymptotic Notations

- The same relationship that holds for relative values of numbers hold for asymptotic.
  - E.g., if $f(n) \in O(g(n))$ [$f(n)$ is asymptotically smaller than or equal to $g(n)$], then we have $g(n) \in \Omega(f(n))$ [$g(n)$ is asymptotically larger than or equal to $f(n)$].

    we know $\exists M', n_0$ s.t., $f(n) \leq M'g(n)$ for $n \geq n_0$, i.e., $g(n) \geq 1/M' \times f(n)$ (select the same n_0 and $M = 1/M'$).
- In order to prove $f(n) \in \Theta(g(n))$, we often show that $f(n) \in O(n)$ and $f(n) \in \Omega(g(n))$.

  suppose $\exists M_1, n_0'$ s.t., $f(n) \leq M_1 g(n)$ for $n \geq n_0'$. Also, $\exists M_2, n_0''$ s.t., $f(n) \geq M_2 g(n)$ for $n \geq n_0''$. Select, $n_0 = \max\{n_0', n_0''\}$ and we have $M_2 g(n) \leq f(n) \leq M_1 g(n)$.

Fun with Asymptotic Notations

- We have **transitivity** in asymptotic notations: if $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, we have $f(n) \in O(h(n))$.

  We know $\exists M_1, n_0'$ s.t., $f(n) \leq M_1 g(n)$ for $n \geq n_0'$. Also, $\exists M_2, n_0''$ s.t., $g(n) \leq M_2 h(n)$ for $n \geq n_0''$. For $n \geq n_0$ with $n_0 = \max\{n_0', n_0''\}$, it holds that $f(n) \leq M_1 M_2 h(n)$ (select $M = M_1 M_2$).

- **Max rule**: $f(n) + g(n) \in \Theta(\max\{f(n), g(n)\})$.
  - E.g., $2n^3 + 8n^2 + 16n\log n \in \Theta(\max\{2n^3, 8n^2, 16n\log n\}) = \Theta(n^3)$.

  it holds that $\max\{f(n), g(n)\} \leq f(n) + g(n) \leq 2\max\{f(n), g(n)\}$ for $n \geq 1$. (select $n_0 = 1$, $M_1 = 1$ and $M_2 = 2$).

Fun with Asymptotic Notations

- What is the time complexity of **arithmetic sequences**?
  - $\sum_{i=0}^{n-1}(a+di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2)$
- What about **geometric sequence**?
  - $\sum_{i=0}^{n-1} ar^i = \begin{cases} a\frac{1-r^n}{1-r} \in \Theta(1) & \text{if } 0 < r < 1 \\ na \in \Theta(n) & \text{if } r = 1 \\ a\frac{r^n-1}{r-1} \in \Theta(r^n) & \text{if } r > 1 \end{cases}$
- What about **Harmonic sequence**?
  - $H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln(n) + \gamma \in \Theta(\log n)$ (γ is a constant≈ 0.577)

Loop Analysis

- Identify **elementary operations** that require constant time.
- The complexity of a loop is expressed as the **sum** of the complexities of each iteration of the loop.
- Analyse independent loops separately, and then **add** the results (use "maximum rules" and simplify when possible).
- If loops are nested, start with the innermost loop and proceed outwards.

Example of Loop Analysis

```
Algo1 (n)
1.  $A \leftarrow 0$
2.  for $i \leftarrow 1$ to $n$ do
3.      for $j \leftarrow i$ to $n$ do
4.          $A \leftarrow A/(i-j)^2$
5.          $A \leftarrow A^{100}$
6.  return sum
```

$$O(1)+\Sigma_{i=1}^{n}\Sigma_{j=i}^{n}c = O(1)+\Sigma_{i=1}^{n}(n-i+1)c = O(1)+\Sigma_{p=1}^{n}pc = \Theta(n^2)$$

Example of Loop Analysis

```
Algo2 $(A, n)$
1.  $max \leftarrow 0$
2.  for $i \leftarrow 1$ to $n$ do
3.      for $j \leftarrow i$ to $n$ do
4.          $X \leftarrow 0$
5.          for $k \leftarrow i$ to $j$ do
6.              $X \leftarrow A[k]$
7.              if $X > max$ then
8.                  $max \leftarrow X$
9.  return $max$
```

$$\sum_{i=1}^{n}\sum_{j=i}^{n}(O(1)+\sum_{k=i}^{j}c) = \Theta(n^3)$$

Example of Loop Analysis

```
Algo3 ($n$)
1.  $X \leftarrow 0$
2.  for $i \leftarrow 1$ to $n^2$ do
3.      $j \leftarrow i$
4.      while $j \geq 1$ do
5.          $X \leftarrow X + i/j$
6.          $j \leftarrow \lfloor j/2 \rfloor$
7.  return $X$
```

- The while loop takes $O(\log i)$; note that $\log(x!) = \Theta(x \log x)$.
- The time complexity is asymptotically equal to

$$\sum_{i=1}^{n^2} \log i = \log 1 + \log 2 + \ldots \log n^2 = \log(1 \times 2 \times \ldots \times n^2) = \log(n^2!)$$

$$= \Theta(n^2 \log(n^2)) = \Theta(2n^2 \log(n^2)) = \Theta(n^2 \log n)$$

MergeSort

Sorting an array A of n numbers

- **Step 1:** We split A into two subarrays: A_L consists of the first $\lceil \frac{n}{2} \rceil$ elements in A and A_R consists of the last $\lfloor \frac{n}{2} \rfloor$ elements in A.
- **Step 2:** Recursively run *MergeSort* on A_L and A_R.
- **Step 3:** After A_L and A_R have been sorted, use a function *Merge* to merge them into a single sorted array. This can be done in time $\Theta(n)$.

MergeSort

```
$MergeSort(A, n)$
1.  if $n = 1$ then
2.      $S \leftarrow A$
3.  else
4.      $n_L \leftarrow \lceil \frac{n}{2} \rceil$
5.      $n_R \leftarrow \lfloor \frac{n}{2} \rfloor$
6.      $A_L \leftarrow [A[1], \ldots, A[n_L]]$
7.      $A_R \leftarrow [A[n_L + 1], \ldots, A[n]]$
8.      $S_L \leftarrow MergeSort(A_L, n_L)$
9.      $S_R \leftarrow MergeSort(A_R, n_R)$
10.     $S \leftarrow Merge(S_L, n_L, S_R, n_R)$
11. return $S$
```

Analysis of MergeSort

- The following is the corresponding **sloppy recurrence** (it has floors and ceilings removed):

$$T(n) = \begin{cases} 2\,T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \\ d & \text{if } n = 1. \end{cases}$$

- The exact and sloppy recurrences are identical when n is a power of 2.
- The recurrence can easily be solved by various methods when $n = 2^j$. The solution has growth rate $T(n) \in \Theta(n \log n)$.
- It is possible to show that $T(n) \in \Theta(n \log n)$ for all n by analyzing the exact recurrence.

Analysis of Recursions

- The sloppy recurrence for time complexity of merge sort:

$$T(n) = \begin{cases} 2\,T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \\ d & \text{if } n = 1. \end{cases}$$

- We can find the solution using **alternation method**:

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 2(2T(n/4) + cn/2) + cn = 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn = 8T(n/8) + 3cn \\ &= \dots \\ &= 2^k T(n/2^k) + kcn \\ &= 2^{\log n} T(1) + \log ncn = \Theta(n \log n) \end{aligned}$$

Substitution method

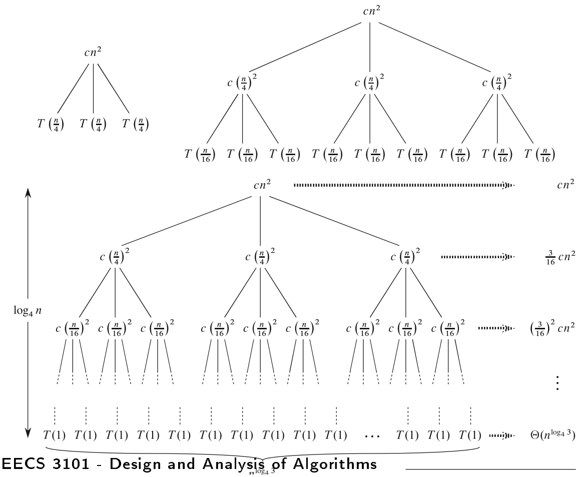
- **Guess** the growth function and prove an upper bound for it using induction.
  - For merge-sort, prove $T(n) < Mn \log n$ for some value of M (that we choose).
  - This holds for $n = 2$ since we have $T(2) = 2d + 2c$, which is less than $2M$ as long as $M \geq c + d$ (base of induction).
  - Fix a value of n and assume the inequality holds for smaller values. we have $T(n) = 2T(n/2) + cn \leq 2M(n/2(\log n/2)) + cn = Mn(\log n/2) + cn = Mn \log n - Mn + cn \leq Mn \log n$ as long as M is selected to be at least c (the inequality comes from the induction hypothesis).
- This shows $T(n) \in O(n \log n)$

Recursion Tree

- Suppose we want to solve the following recursion:

$$T(n) = \begin{cases} \mathbf{3}\ T\left(\frac{n}{\mathbf{4}}\right) + cn^{\mathbf{2}} & \text{if } n > \mathbf{1} \\ d & \text{if } n = \mathbf{1}. \end{cases}$$

- Let's form a **recursion tree**:



cn^2
$T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$
$c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$
$\log_4 n$
cn^2
$\frac{3}{16}cn^2$
$c\left(\frac{n}{16}\right)^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$T(1)$ $T(1)$ $T(1)$ $\cdots$ $T(1)$
$\Theta(n^{\log_4 3})$
$n^{\log_4 3}$

- The total work in **internal nodes** is $cn^2(1 + 3/16 + (3/16)^2 + \ldots) = \Theta(n^2)$.
- The total work in **leaves** is $n^{\log_3 4}$.
- The max rule indicates that $T(n) = \Theta(n^2)$.

Master theorem

$$T(n) = \begin{cases} a\,T\left(\frac{n}{b}\right) + f(n) & \text{if } n > 1 \\ d & \text{if } n = 1. \end{cases}$$

$(a \geq 1,\ b > 1$, and $f(n) > 0)$

- Compare $f(n)$ and $n^{\log_b a}$
- Case 1: if $f(n) \in O(n^{\log_b a-\epsilon})$, then $T(n) \in \Theta(n^{\log_b a})$
- Case 2: if $f(n) \in \Theta(n^{\log_b a}(\log n)^k)$ for some non-negative k then $T(n) \in \Theta(f(n)\log n) = \Theta(n^{\log_b a}(\log n)^{k+1})$
- Case 3: if $f(n) \in \Omega(n^{\log_b a+\epsilon})$ and if $af(n/b) \leq cf(n)$ for **some constant** $c < 1$ (regularity condition), then $T(n) \in \Theta(f(n))$

Master theorem examples

- $T(n) = 2T(n/2) + \log n$? case 1: $T(n) \in \Theta(n)$
- $T(n) = 4T(n/4) + 100n$? case 2: $T(n) \in \Theta(n \log n)$
- $T(n) = 3T(n/2) + n^2$?
  - Case 3, check whether regularity condition holds, i.e., whether $af(n/b) \leq cf(n)$ for some $c < 1$. Since we have $3(n/2)^2 = 3/4n^2$ the regularity condition holds (c can be any value in the range $(3/4, 1)$, i.e., $T(n) \in \Theta(n^2)$
- $T(n) = T(n/2) + n(2 - cos(n))$?
  - Case 3, check whether regularity condition holds.
  - For $n = 2k\pi$, we have $cos(n/2) = -1$ and $cos(n) = 1$; we have $af(n/b) = n/2(2 - cos(n/2)) = 3n/2$, which is not within a factor $c < 1$ of $f(n) = n(2 - 1) = n$ [i.e., we cannot say $3n/2 \leq cn$ for any $c < 1$]. So we cannot get any conclusion from Master theorem.
- $T(n) = 2T(n/2) + n(\log n)^3$? Case 2, we have $f(n) = \Theta(n^{\log_b a}(\log n)^k)$ for $k = 3$. We have $T(n) = \Theta(n(\log n)^4)$.