

CCES – College of Computer Engineering and Sciences

Prince Sattam Bin Abdulaziz University

CS3401 – Algorithm Design and Analysis

Chapter 5

Transform & Conquer

Academic Year 2026 – 2027



Dr. Khalil Chebil

A white capital letter 'E' centered within a white circle.

Dr. Esraa Eldesouky

Chapter Overview

Introduction

Instance Simplification — Presorting

Representation Change — Balanced Trees

Representation Change — Heaps & Heapsort

Problem Reduction

Linear Programming

1

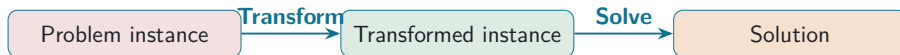
Section 1

Introduction

The Transform & Conquer Paradigm

Core idea

Instead of solving a problem directly, *transform* it into a different (simpler or better-understood) form, then solve.



1. Instance simplification

Transform to a simpler/easier instance of the *same* problem.

Example: presort before searching

2. Representation change

Transform to a different *data structure* for the same instance.

Example: AVL tree, heap

3. Problem reduction

Transform to a *different* problem with a known algorithm.

Example: LCM via GCD

Section 2

Instance Simplification — Presorting

Presorting

Idea

Many problems become *much easier* on a sorted list. Sort once in $\Theta(n \log n)$, then exploit order repeatedly.

Example 1: Element uniqueness

- ▶ Brute force: compare all pairs $\rightarrow \mathcal{O}(n^2)$
- ▶ Presort + scan adjacent pairs $\rightarrow \Theta(n \log n)$

```
1: sort A
2: for  $i \leftarrow 0$  to  $n - 2$ 
3:   if  $A[i] = A[i + 1]$  then return false
4:   end if
5: end for
6: return true
```

Problem	Brute force	After presort
Uniqueness	$\mathcal{O}(n^2)$	$\Theta(n \log n)$
Mode	$\mathcal{O}(n^2)$	$\Theta(n \log n)$
Search/query	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$

Example 2: Mode (most frequent element)

- ▶ Presort, then scan runs to find the longest

Example 3: Searching with many queries

- ▶ Presort once in $\Theta(n \log n)$
- ▶ Each binary search query: $\mathcal{O}(\log n)$
- ▶ Break-even vs. sequential: $\approx \log n$ queries

Key question

Is the $\Theta(n \log n)$ presorting cost worth paying?

Yes if you execute *many queries* on the same data.

No if you only need one answer.

Section 3

Representation Change — Balanced Trees

Binary Search Trees — The Problem

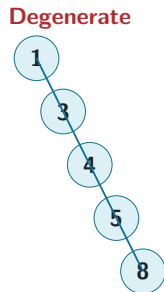
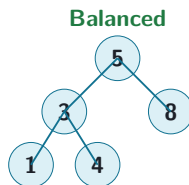
BST operations (search, insert, delete) depend on *height*:

- ▶ Best (balanced): $h = \lfloor \log_2 n \rfloor \rightarrow \Theta(\log n)$
- ▶ Worst (sorted input): $h = n - 1 \rightarrow \Theta(n)$

The problem

Inserting sorted data into a BST produces a *linear chain* — as slow as sequential search!

Solution: Keep the tree *balanced* after every insertion/deletion.



AVL Trees

Definition

An **AVL tree** is a BST where for every node, the *balance factor* (height of left subtree minus height of right subtree) is -1 , 0 , or $+1$.

Rebalancing rotations (when a new node violates balance):

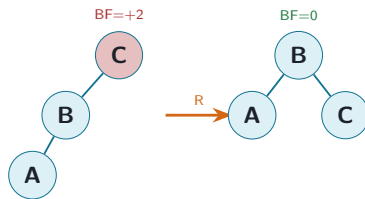
- ▶ **Single R-rotation**: left-left imbalance
- ▶ **Single L-rotation**: right-right imbalance
- ▶ **Double LR-rotation**: left-right imbalance
- ▶ **Double RL-rotation**: right-left imbalance

Guaranteed height:

$$h \leq 1.44 \log_2(n + 2) - 1.33$$

All operations: $\mathcal{O}(\log n)$ worst case.

Single R-rotation example:



Construction: insert 5,6,8,3,2,4,7
(worked trace: 4 rotations needed)

2-3 Trees

Definition

A **2-3 tree** is a balanced search tree where:

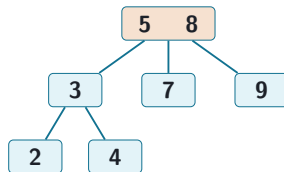
- ▶ Every node has 2 or 3 children (internal nodes)
- ▶ All leaves are at the *same level*

Insertion: insert at a leaf; if a 3-node overflows, *split* it and push the middle key up.

Height bounds:

$$\log_3(n+1) - 1 \leq h \leq \log_2(n+1) - 1$$

All operations: $\Theta(\log n)$



Generalisations: 2-3-4 trees, B-trees
(used in databases and file systems for disk-block efficiency)

Section 4

Representation Change — Heaps & Heapsort

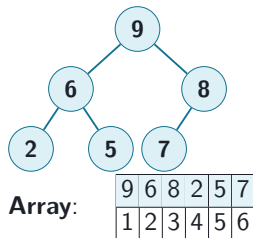
Heaps

Definition

A **heap** is a complete binary tree where every node's key is $\geq$ its children's keys (*max-heap*).

Key properties:

- ▶ Height: $\lfloor \log_2 n \rfloor$ — always balanced
- ▶ Root holds the *maximum* key
- ▶ Stored as an array (left-to-right, level by level):
 - Left child of j : index $2j$
 - Right child of j : index $2j + 1$
 - Parent of j : index $\lfloor j/2 \rfloor$



Heapsort

Stage 1 — Build heap (bottom-up): Start from last parental node; sift down each node.
Complexity: $\Theta(n)$

Stage 2 — Sort (repeated max deletion):

1. Swap root (max) with last element
2. Shrink heap by 1
3. Sift down new root
4. Repeat $n - 1$ times

Complexity: $\Theta(n \log n)$

Total: $\Theta(n) + \Theta(n \log n) = \Theta(n \log n)$

Sorting [2, 9, 7, 6, 5, 8]:

After build-heap:

9 6 8 2 5 7

After extraction 1:

7 6 8 2 5 9

After sift-down:

8 6 7 2 5 9

... continues ...

2 5 6 7 8 9

In-place: **Yes** Stable: **No**



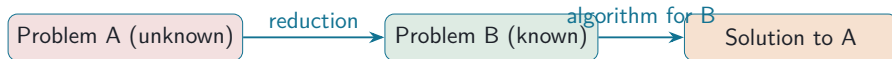
Section 5

Problem Reduction

Problem Reduction — Concept

Idea

Solve problem A by transforming it into problem B for which an efficient algorithm is already known.



Classic examples:

- ▶ $\text{lcm}(m, n) = mn / \text{gcd}(m, n)$ — reduce LCM to GCD
- ▶ Paths of length k in graph: compute A^k (matrix power)
- ▶ Minimisation $\rightarrow$ maximisation: negate objective

Requirement for usefulness

The combined cost of the reduction *plus* solving B must be less than the best direct algorithm for A.

LCM via GCD

Theorem

For any positive integers m, n :

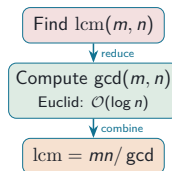
$$\text{lcm}(m, n) = \frac{m \cdot n}{\text{gcd}(m, n)}$$

Algorithm:

1. Compute $\text{gcd}(m, n)$ using Euclid's algorithm — $\mathcal{O}(\log n)$
2. Return $m / \text{gcd}(m, n) \cdot n$ — $\mathcal{O}(1)$

Example: $\text{lcm}(24, 60)$

- ▶ $\text{gcd}(24, 60) = 12$
- ▶ $\text{lcm}(24, 60) = 24 \times 60 / 12 = 120$



Alternative (brute force): $24 = 2^3 \cdot 3$, $60 = 2^2 \cdot 3 \cdot 5$

$$\text{lcm} = 2^3 \cdot 3 \cdot 5 = 120$$

Requires prime factorisation — much slower.

Lesson

Reduction to GCD turns an apparently hard problem (LCM) into a one-liner with $\mathcal{O}(\log n)$ complexity.

Section 6

Linear Programming

Linear Programming — Introduction

Definition

A **Linear Program (LP)** maximises (or minimises) a *linear* objective function subject to *linear* inequality/equality constraints.

maximise/minimise $c_1x_1 + \dots + c_nx_n$ subject to $a_{i1}x_1 + \dots + a_{in}x_n \leq / \geq / = b_i, x_j \geq 0$

Components:

- ▶ **Decision variables:** $x_1, \dots, x_n$
- ▶ **Objective function:** linear in x_j
- ▶ **Constraints:** linear inequalities/equalities
- ▶ **Non-negativity:** $x_j \geq 0$

Properties:

- ▶ *Feasible region:* convex polygon/polyhedron
- ▶ *Optimal solution:* always at a vertex
- ▶ Solvable in polynomial time (ellipsoid method, interior-point methods)
- ▶ Simplex: efficient in practice

LP Example: Production Planning

Problem: A factory produces chocolates A and B. Resources (per unit): A needs 1 Milk + 3 Choco; B needs 1 Milk + 2 Choco. Available: 5 Milk, 12 Choco. Profit: 6 SAR/unit A, 5 SAR/unit B. Maximise profit.

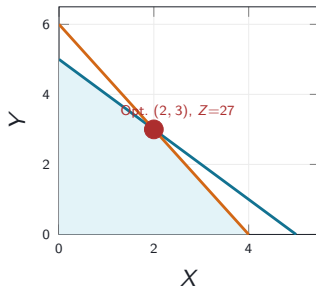
LP formulation:

$$\max \quad 6X + 5Y \quad (1)$$

$$\text{s.t.} \quad X + Y \leq 5 \quad (2)$$

$$3X + 2Y \leq 12 \quad (3)$$

$$X, Y \geq 0 \quad (4)$$



Reducing the 0-1 Knapsack to an LP

0-1 Knapsack as an Integer Linear Program (ILP):

$$\max \sum_{j=1}^n v_j x_j \quad \text{s.t.} \quad \sum_{j=1}^n w_j x_j \leq W, \quad x_j \in \{0, 1\}$$

Example ($W = 10$):

	w	v	x^*
Item 1	7	42	0
Item 2	3	12	0
Item 3	4	40	1
Item 4	5	25	1

Optimal:

items 3+4, value = 65, weight = 9

ILP is harder than LP

Adding integrality constraints makes the problem NP-hard. Solving it exactly requires branch-and-bound or dynamic programming.

LP relaxation

Remove $x_j \in \{0, 1\}$; allow $0 \leq x_j \leq 1$. This gives an upper bound and is solvable in polynomial time. Rounding the LP solution gives a heuristic.

Chapter 5 Summary

Three flavours of Transform & Conquer:

1. **Presorting** — $\Theta(n \log n)$ once, then fast queries
2. **Representation change**
 - AVL trees: $\mathcal{O}(\log n)$ all ops
 - 2-3 trees: $\Theta(\log n)$ all ops
 - Heaps: $\mathcal{O}(\log n)$ insert/delete-max
 - Heapsort: $\Theta(n \log n)$, in-place
3. **Problem reduction**
 - LCM $\rightarrow$ GCD, graph paths $\rightarrow$ matrix power
 - LP as a reduction target for optimisation

Next chapter

Chapter 6: Dynamic Programming — solving problems with *overlapping sub-problems* by filling a table bottom-up (or top-down with memoisation).

Self-check

1. When is presorting beneficial?
2. What rotation fixes a left-right imbalance in an AVL tree?
3. What is the difference between LP and ILP?