Transform & conquer

- transform-and-conquer approach
- presorting
- balanced search trees, heaps
- Horner's Rule
- problem reduction

The idea behind transform-and-conquer is to transform the given problem into a slightly different problem that suffices.

e.g., presorting data in a list can simplify many algorithms

- suppose we want to determine if a list contains any duplicates

  BRUTE FORCE: compare each item with every other item
  \((N-1) + (N-2) + \ldots + 1 = (N-1)N/2 \rightarrow O(N^2)\)

  TRANSFORM & CONQUER: first sort the list, then make a single pass through the list and check for adjacent duplicates
  \(O(N \log N) + O(N) \rightarrow O(N \log N)\)

- finding the mode of a list, finding closest points, …
Balanced search trees

recall binary search trees – we need to keep the tree balanced to ensure \( O(N \log N) \) search/add/remove

- OR DO WE?
  - it suffices to ensure \( O(\log N) \) height, not necessarily minimal height

transform the problem of "tree balance" to "relative tree balance"

several specialized structures/algorithms exist:
- AVL trees
- 2-3 trees
- red-black trees

AVL trees

an AVL tree is a binary search tree where

- for every node, the heights of the left and right subtrees differ by at most 1

- first self-balancing binary search tree variant
- named after Adelson-Velskii & Landis (1962)

AVL property is weaker than perfect balance, but sufficient

height of AVL tree with \( N \) nodes < 2 \( \log(N+2) \)

\( \rightarrow \) searching is \( O(\log N) \)
Inserting/removing from AVL tree

when you insert or remove from an AVL tree, imbalances can occur

- if an imbalance occurs, must rotate subtrees to retain the AVL property

AVL tree rotations

there are two possible types of rotations, depending upon the imbalance caused by the insertion/removal

- worst case, inserting/removing requires traversing the path back to the root and rotating at each level
  - each rotation is a constant amount of work → inserting/removing is $\mathcal{O}(\log N)$
Red-black trees

A red-black tree is a binary search tree in which each node is assigned a color (either red or black) such that

1. The root is black
2. A red node never has a red child
3. Every path from root to leaf has the same number of black nodes

- Add & remove preserve these properties (complex, but still $O(\log N)$)
- Red-black properties ensure that tree height $< 2 \log(N+1) \rightarrow O(\log N)$ search

TreeSets & TreeMaps

- `java.util.TreeSet` uses red-black trees to store values
  - $O(\log N)$ efficiency on add, remove, contains

- `java.util.TreeMap` uses red-black trees to store the key-value pairs
  - $O(\log N)$ efficiency on put, get, containsKey

Thus, the original goal of an efficient tree structure is met

- Even though the subgoal of balancing a tree was transformed into "relatively balancing" a tree
Scheduling & priority queues

many real-world applications involve optimal scheduling
- balancing transmission of multiple signals over limited bandwidth
- selecting a job from a printer queue
- selecting the next disk sector to access from among a backlog
- multiprogramming/multitasking

a priority queue encapsulates these three optimal scheduling operations:
- add item (with a given priority)
- find highest priority item
- remove highest priority item

- can be implemented as an unordered list
  → add is O(1), findHighest is O(N), removeHighest is O(N)
- can be implemented as an ordered list
  → add is O(N), findHighest is O(1), removeHighest is O(1)

Heaps

Java provides a `java.util.PriorityQueue` class
- the underlying data structure is not a list or queue at all
- it is a tree structure called a heap

a complete tree is a tree in which
- all leaves are on the same level or else on 2 adjacent levels
- all leaves at the lowest level are as far left as possible
- note: a complete tree with N nodes will have minimal height = \( \lceil \log_2 N \rceil + 1 \)

a heap is complete binary tree in which
- for every node, the value stored is ≤ the values stored in both subtrees
  (technically, this is a min-heap -- can also define a max-heap where the value is ≥)
Inserting into a heap

to insert into a heap
  - place new item in next open leaf position
  - if new value is smaller than parent, then swap nodes
  - continue up toward the root, swapping with parent, until smaller parent found

note: insertion maintains completeness and the heap property
  - worst case, if add smallest value, will have to swap all the way up to the root
  - but only nodes on the path are swapped \(\text{O(height)} = \text{O(log N)}\) swaps

Finding/removing from a heap

finding the min value in a heap is \(\text{O(1)}\) – it’s in the root

removing the min value requires some work
  - replace root with last node on bottom level
  - if new root value is greater than either child, swap with smaller child
  - continue down toward the leaves, swapping with smaller child, until smallest

note: removing root maintains completeness and the heap property
  - worst case, if last value is largest, will have to swap all the way down to leaf
  - but only nodes on the path are swapped \(\text{O(height)} = \text{O(log N)}\) swaps
Implementing a heap

a heap provides for $O(1)$ find min, $O(\log N)$ insertion and min removal

- also has a simple, List-based implementation
- since there are no holes in a heap, can store nodes in an ArrayList, level-by-level

- root is at index 0
- last leaf is at index $\text{size()} - 1$
- for a node at index $i$, children are at $2i+1$ and $2i+2$
- to add at next available leaf, simply add at end

Heap sort

the priority queue nature of heaps suggests an efficient sorting algorithm

- start with the ArrayList to be sorted
- construct a heap out of the elements
- repeatedly, remove min element and put back into the ArrayList

- $N$ items in list, each insertion can require $O(\log N)$ swaps to reheapify
  $\rightarrow$ construct heap in $O(N \log N)$
- $N$ items in heap, each removal can require $O(\log N)$ swap to reheapify
  $\rightarrow$ copy back in $O(N \log N)$

thus, overall efficiency is $O(N \log N)$, which is as good as it gets!
- can also implement so that the sorting is done in place, requires no extra storage
Horner's rule

Polynomials are used extensively in mathematics and algorithm analysis

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots + a_1 x + a_0 \]

How many multiplications would it take to evaluate this function for some value of \( x \)?

W.G. Horner devised a new formula that transforms the problem

\[ p(x) = (((a_n x + a_{n-1}) x + a_{n-2}) x + \ldots + a_1) x + a_0 \]

can evaluate in only \( n \) multiplications and \( n \) additions

Problem reduction

In CSC321, we looked at a number of examples of reducing a problem from one form to another

- \( \text{e.g., generate the powerset (set of all subsets) of an N element set} \)
  \[ S = \{ x_1, x_2, x_3, x_4 \} \quad \text{powersets} = \{ \emptyset, \{ x_1 \}, \{ x_2 \}, \{ x_3 \}, \{ x_4 \}, \{ x_1, x_2 \}, \{ x_1, x_3 \}, \{ x_1, x_4 \}, \{ x_2, x_3 \}, \{ x_2, x_4 \}, \{ x_3, x_4 \}, \{ x_1, x_2, x_3 \}, \{ x_1, x_2, x_4 \}, \{ x_1, x_3, x_4 \}, \{ x_2, x_3, x_4 \}, \{ x_1, x_2, x_3, x_4 \} \} \]

- \( \text{PROBLEM REDUCTION: simplify by reducing it to a problem about bit sequences} \)
  can map each subset into a sequence of N bits: \( b_i = 1 \rightarrow x_i \) in subset
  \[ \{ x_1, x_2, x_3 \} \leftrightarrow 10011000\ldots0 \]

much simpler to generate all possible N-bit sequences

\[ \{ 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111 \} \]
lcm & gcd

consider calculating the least common multiple of two numbers m & n

- BRUTE FORCE: reduce each number to its prime factors
  then multiply (factors in both m & n) (factors only in m) (factors only in n)

\[
\begin{align*}
24 & = 2 \cdot 2 \cdot 2 \cdot 3 \\
60 & = 2 \cdot 2 \cdot 3 \cdot 5 \\
lcm(24, 60) & = (2 \cdot 2 \cdot 3) \cdot (2) \cdot (5) = 12 \cdot 2 \cdot 5 = 120
\end{align*}
\]

- PROBLEM REDUCTION: can recognize a relationship between lcm & gcd

\[
lcm(m, n) = \frac{m \times n}{\gcd(m, n)}
\]

\[
lcm(24, 60) = \frac{24 \times 60}{12} = 2 \cdot 60 = 120
\]

gcd can be calculated efficiently using Euclid’s algorithm:

\[
gcd(a, 0) = a \\
gcd(a, b) = gcd(b, a \% b)
\]