Solution Construction Paradigms

Control flow paradigms categorize algorithms based on their repetitive structure.

- loops
- recursion

Solution construction paradigms categorize algorithms based on how solutions are built.

- *decomposition* – solve the problem by breaking it into smaller problems and combining the subproblem solutions
- *series of choices* – build the solution incrementally by making a series of decisions

Divide and Conquer

- *decomposition* approach + *recursive* control flow

Recursive Patterns

Characterized by the number and size of subproblems –

- 2+ friends
- *divide-and-conquer* – split into $b \geq 2$ subproblems of size n/b (b is typically 2)
  - *case analysis* – each friend gets a subproblem resulting from a different alternative

How to Design Divide-and-Conquer Algorithms

- recursive, so follow the recursive process
  - establish the problem
    - specifications
    - examples
  - identify avenues of attack
    - know targets
    - patterns and paradigms
  - define the algorithm
    - generalize/define subproblems
    - base case(s)
    - main case
    - top level: initial subproblem, setup, wrapup
    - special cases
    - algorithms
  - show termination and correctness
    - termination: size, making progress, the end is reached
    - correctness: establish the base case(s), show the main case, final answer
  - determine efficiency
    - implementation
    - time and space
    - room for improvement

How to Design Divide-and-Conquer Algorithms

- additions/specializations for divide-and-conquer

- *Known targets.*
  Divide-and-conquer algorithms often seek to improve on a polynomial-time iterative brute-force running time. If there aren't explicit targets stated for the problem, identify the brute force algorithm and its running time.
- *Patterns.*
  Consider specifically the divide-and-conquer patterns defined in section 5.1 — can they be applied to this problem? What would such an approach look like?

  - *easy split* – split input in straightforward way, then do work combining subproblem solutions
    - e.g. mergesort
  - *easy merge* – do work creating the subproblem instances, then just append subproblem solutions
    - e.g. quicksort

- *Room for improvement.*
  For divide-and-conquer, there are two common strategies for trying to improve the running time. Can you leverage the work the friends are doing and have them hand back more in order to reduce your computation? Do you need to pass all of the elements off to friends or can some be eliminated?