Chapter 5

Divide and Conquer

Decomposition as a solution-construction strategy refers to solving a problem by breaking it into smaller subproblems. *Divide-and-conquer* is an algorithmic paradigm involving decomposition and recursion — to solve a problem, divide it into b subproblems of size n/b where $b \geq 2$ (typically 2).

5.1 Divide-and-Conquer Patterns

Divide-and-conquer algorithms often take one of the following forms:

- **easy split**, where the input is divided in half in a straightforward way (such as "first half" and "second half"); the work in the main case is primarily in combining the results from the friends to produce the solution
- **easy merge**, where each friend produces some of the output (typically one friend produces the first part of the output and the other friend produces the second part); the work in the main case is primarily in splitting the input

5.2 How to Design Divide-and-Conquer Algorithms

Divide-and-conquer algorithms are recursive and thus follow the template in section 4.3 with a few additions/modifications:

- *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?
- *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?

The full process is given below.

A vertical rule in the left margin (as illustrated here) is used to highlight the elements of the process specific to divide-and-conquer algorithms.

Establish the problem. Defining — and understanding — the precise task to be solved is an essential prerequisite for algorithm development.

- *Specifications.*
  State complete specifications for the problem. What is the problem? What do you start with (input) and what is the end result (output)? What are the legal input instances and the required output for each?
- *Examples.*
  If needed, give examples (specific inputs and the corresponding outputs) of typical and special cases to clarify the specifications.

Identify avenues of attack. Determine the algorithmic approach(es) to try. In real problem solving the appropriate paradigm is not always clear at the start, so several possibilities may need to be considered. In assigned problems the paradigm is often specified or implied, allowing this step to be abbreviated or skipped.

- *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.

- *Paradigams and 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?

Define the algorithm. Assemble the pieces of the algorithm according to the template for the particular paradigm and pattern, culminating in a statement of the algorithm itself.

- *Generalize / define subproblems.*
  Define the subproblem — the task, the input, and the output. Write this as a function definition where parameters and return values are explicitly identified.
- *Base case(s).*
  Address how to solve the smallest problem(s). This is often trivial, or is solved via brute force.
- *Main case.*
  Address how to solve a typical large problem instance (one that is not a base case). Specify how many friends are needed and what subproblem is handed to each friend, as well as how the results the friends hand back are combined to solve the problem.
- *Top level.*
  The top level puts the context around the recursion.
  - *Initial subproblem.*
    Specify the inputs and parameters for the initial subproblem — the one whose solution solves the original problem.
  - *Setup.*
    Whatever must happen before the initial subproblem is solved.
  - *Wrapup.*
    Whatever must happen to get the final answer after the solution for the initial subproblem is obtained.

- *Special cases.*
  Make sure the algorithm works for all legal inputs — identify the cases that need to be handled and address how that handling is incorporated into the previous steps (if not already accounted for).

- *Algorithm.*
  State the whole algorithm. There shouldn't be new elements here, instead bring together the base case(s), main case, and top level along with any handling needed for special cases and state the whole algorithm.

  The algorithm should be specified at as high a level as possible but with enough detail to clearly convey the steps and to be able to show correctness and (with the addition of implementation details later) address running time. Can a person familiar with standard data structures and algorithms carry out the algorithm by hand based on what is written on the page?

Show termination and correctness. Show that the algorithm produces a correct solution.

- *Termination.* Show that the algorithm produces a correct solution.
  - *Size.*
    What is the size of the problem?
  - *Making progress.*
    Explain why what each of your friends get is a smaller instance of the problem.
  - *The end is reached.*
    Explain why a base case is always reached.

- *Correctness.* Show that the algorithm is correct.
  - *Establish the base case(s).*
    Explain why the solution is correct for each base case.
  - *Show the main case.*
    Assume that the friends return the correct results for their subproblem, and explain why the correct answer is then produced from those results.
  - *Final answer.*
    Explain why the top level — the setup plus a correct solution to the initial subproblem followed by the wrapup — means that the final result is a correct answer to the problem.

Determine efficiency. Evaluate the running time and space requirements of the algorithm.

- *Implementation.*
  Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic details that are needed in order to establish the running time.

- *Time and space.*
  Assess the running time and space requirements of the algorithm given the implementation identified.

- *Room for improvement.*
  Are the targets met? Is it necessary to do better? If improvements in running time and/or space are needed, identify possible avenues 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?