Chapter 2

Introduction

2.1 Paradigms

Algorithms are often categorized by their approach to problem-solving. Common algorithmic paradigms include:

- *divide and conquer*, where a problem is solved by breaking it into smaller instances of the same problem and then combining the subproblem solutions,
- *greedy*, where a locally optimal choice is made at each step,
- *backtracking*, where legal pathways to a solution are systematically explored to find a solution,
- *branch and bound*, an extension to backtracking where unnecessary paths leading to suboptimal solutions are eliminated, and
- *dynamic programming*, where repeated subproblems are leveraged to reduce computation time.
- *randomized algorithms*, where randomness is incorporated in order to improve efficiency or simplify the implementation, and
- *approximation algorithms*, where near-optimal solutions are sought when optimal solutions are too computationally expensive.

Each of these paradigms not only encompasses a particular methodology for solving computational problems, but also gives rise to a set of tactics that can be employed for developing algorithms of that type.

2.2 Structures and Approaches

There are also broader commonalities between algorithms that span multiple paradigms and which can be exploited when considering algorithm development strategies. At the most fundamental structural level, there are only two types of algorithms —

- those that do repetition using loops, and
- those that do repetition using recursion.

Algorithms can also be categorized by their approach to generating a solution rather than by the programming constructs used in their implementation. We'll focus primarily on two such approaches:

- *divide and conquer*, where a problem is solved by breaking it into smaller instances of the same problem and then combining the subproblem solutions, and
- *series of choices*, where the solution is built up incrementally by making a series of choices or decisions.

2.3 How to Design Algorithms

Our process[1] for designing algorithms involves five main steps: establish the problem, identify avenues of attack, define the algorithm, show termination and correctness, and determine efficiency. An outline of these steps is given below; more specific versions for particular algorithmic approaches and paradigms will be discussed in chapters 3 to 10.

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. The approach and paradigm steps are intended for when a paradigm is not pre-determined; skip them if the paradigm is already decided.

- *Targets.*
  Identify any applicable time and space requirements for your solution. This might be stated as part of the problem ("find an $O(n \log n)$ algorithm"), come from having an algorithm you are trying to improve on, or stem from the expected input size (e.g. you need to work with very large inputs so you need $O(n \log n)$ or better).
- *Approach.*
  Identify applicable approach(es): divide and conquer or series of choices? For each, consider briefly what that approach would look like for this problem — does it even make sense?
- *Paradigms and patterns.*
  Based on the targets and approach(es) identified, identify the applicable paradigm(s) and specific pattern(s) within each paradigm. For each, consider briefly what that paradigm and pattern would look like for this problem — does it even make sense?

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.

- *Algorithm.*
  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.*
  Explain why the algorithm always terminates i.e. it always eventually produces a solution.
- *Correctness.*
  Explain why the solution produced is the correct solution for every valid input instance.

[1] adapted from Jeff Edmonds, "How to Think About Algorithms"

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.