Chapter 2

Introduction

2.1 Paradigms

Algorithm design is often best understood in terms of recurring problem-solving patterns, or *paradigms*. Instead of inventing a completely new strategy for each problem, many algorithms can be developed by recognizing that a problem fits an existing approach and applying that pattern. Organizing algorithms around paradigms provides a useful framework for understanding how algorithms are structured and how new ones can be developed — each paradigam 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.

One way to classify algorithms is by *control flow*. At the most fundamental structural level, there are only two types of algorithms —

- *iterative*, which do repetition using loops, and
- *recursive*, which do repetition using recursion.

Algorithms can also be classified by *how solutions are constructed*. Two such approaches include

- *decomposition*, where a problem is solved by breaking it into independent subproblems, and
- *series of choices*, where the solution is built up step-by-step by selecting among possible alternatives.

A decomposition approach leads directly to paradigms such as *divide and conquer*, where a problem is solved by breaking it into smaller instances of the same problem and then combining the subproblem solutions.

The series of choices approach encompasses several different paradigms. In some cases the correct decision for each choice can be determined immediately, leading to *greedy algorithms*. In others, the algorithm must systematically explore many possibilities, leading to *recursive backtracking*. To make this search process practical, it is often necessary to incorporate improvements such as eliminating choices that cannot lead to a solution and avoiding redundant work, techniques that underlie paradigms such as *branch and bound* and *dynamic programming*.

Other paradigms also appear in algorithm design. *Randomized algorithms* incorporate randomness to simplify implementation or improve expected efficiency. *Approximation algorithms* seek near-optimal solutions when computing an exact optimal solution would be too computationally expensive. These approaches also play an important role in the broader study of algorithms, but will not be discussed in depth in this course.

2.2 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 later chapters.

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.*
  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 solution-construction approaches: decomposition 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.