Chapter 3

Iterative Algorithms

Iterative algorithms involve loops. There are two main concerns when developing an iterative algorithm —

- figuring out what the repeated step is, and
- establishing the algorithm's correctness.

3.1 Iterative Patterns

Iterative algorithms solve problems by moving towards the solution one iteration at a time. This observation allows for a categorization of iterative algorithms based on the focus of each iteration, including:

- **process input**, where each iteration processes the next input element
- **produce output**, where each iteration produces the next piece of the output
- **narrow the search space**, where each iteration gets closer to the answer by eliminating non-answers

Insertion sort is an example of the process input pattern — each input element is taken in turn and inserted into the group of sorted elements. Selection sort illustrates the produce output pattern — the output is the sorted list, and selection sort finds the smallest remaining element to append to the sorted list so far. Binary search is a prime example of narrowing the search space — each iteration reduces the range within the array where an element might be by eliminating the half where the element isn't.

3.2 Key Elements of Iterative Algorithms

3.2.1 Main Loop

The core of an iterative algorithm is the main loop — the loop body and the exit condition. The iterative pattern shapes the form that these elements take:

pattern	loop structure	exit condition
process input	for each input element, process that element and incorporate it into the solution so far	when all of the input elements have been processed
produce output	repeatedly produce the next output element repeatedly produce the next piece of the solution	when all of the output elements have been produced when the solution is complete
narrow the search space	repeatedly eliminate some non-solutions	when the solution has found or there are no solutions left

3.2.2 Loop Invariants

A *loop invariant* is a statement about the current program state that is true each time the loop condition is tested, including the last time when the exit condition is reached. Good loop invariants need to be strong enough to be able to use them to demonstrate the correctness of the whole algorithm. Sometimes this can be a direct statement, and the invariant takes the form of asserting that the solution computed so far satisfies the same properties that the solution is expected to have. Other times it necessary to take a more indirect route, and the invariant establishes properties about the loop of the result that can then be used to conclude that the solution is a correct solution.

Showing that a loop invariant holds is based on the proof technique of induction. This has two parts: showing the the loop invariant is true at the beginning and showing that if the loop invariant is true at the beginning of one iteration, it continues to hold at the beginning of the next iteration. Since the loop invariant is often trivially true at the beginning of the first iteration, it is useful to include the whole first iteration in the notion of "at the beginning" and show that the invariant holds at the beginning of both the first and second iterations.

3.2.3 Termination and Correctness

A loop terminates when the exit condition is reached. As a result, we must be making progress towards the exit condition with every iteration. This can be done by identifying a *measure of progress* connected to the exit condition and showing that each iteration advances this value towards eventually making the exit condition true.

These elements often take standard forms based on the iterative pattern:

pattern	measure of progress	making progress	termination argument
process input	number of input elements processed	each iteration processes one more element	repeatedly processing one more input element means that eventually all will have been processed
produce output	number of elements in the solution	each iteration produces one more element	repeatedly producing one more output element or one more piece of the solution means that eventually all will have been produced
narrow the search space	size of the current range or (alternatively) the number of solutions still in the current range	each iteration reduces the size of the search space	repeatedly reducing the size of the current range or the number of solutions still in the current range means that eventually there will be no solutions left if the solution hasn't been found

Showing correctness for iterative algorithms is based on the loop invariant. In the simplest form, the loop invariant is a statement that the solution so far is correct — then the combination of the loop invariant being true (the solution so far is true) when the exit condition is reached (the solution is complete) yields the desired result (the complete solution is correct).

The loop invariant often takes a standard form based on the iterative pattern:

pattern	loop invariant
process input	have a correct solution for the first k input elements, or (alternatively) haven't gone wrong yet (solution so far is consistent with a solution for the whole problem)
produce output	have produced the first k elements of the correct output
narrow the search space	either the element is within the current search space / set of solutions or it was never present at all, or (alternatively) not all of the solutions (if there are any) have been eliminated

3.3 How to Design Iterative Algorithms

The steps outlined in section 2.2 apply broadly to all types of algorithms. Putting those together with the elements and patterns applicable to iterative algorithms results in the design process below.

A vertical rule in the left margin (as illustrated here) is used to highlight the elements of the process specific to iterative 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.*
  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).
- *Paradigms and patterns.*

  Consider the iterative patterns defined in section 3.1 — can they be applied to this problem? What would such an approach look like?

Define the algorithm.

The core of an iterative algorithm is defining the loop.

- *Main steps.*
  This is the core of the algorithm — the loop body. What's being repeated?
- *Exit condition.*
  When does the loop end?

Complete the algorithm by identifying anything that happens before or after the loop. Also consider special cases that need handling.

- *Setup.*
  Whatever must happen before the loop begins.
- *Wrapup.*
  Whatever must happen to get the final answer after the loop ends.
- *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 any new pieces — assemble the elements from the previous steps (the main steps, exit condition, setup, and wrapup 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.*
  Explain why the algorithm always terminates i.e. it always eventually produces a solution.

  For iterative algorithms, show that the loop's exit condition is always reached (and thus the loop always terminates).

  - *Measure of progress.*
    Identify a quantity and the direction of change that leads towards the exit condition.
  - *Making progress.*
    Explain why every iteration of the loop advances the measure of progress towards the exit condition.
  - *The end is reached.*
    Explain why making progress ensures that the exit condition is always reached.

- *Correctness.*
  Explain why the solution produced is the correct solution for every valid input instance.

  - *Loop invariant.*
    State a loop invariant.
  - *Establish the loop invariant.*
    Explain why the loop invariant holds at the beginning of the first and second iterations of the loop (i.e. before and after the first iteration).
  - *Maintain the loop invariant.*
    Explain why the loop invariant continues to be true after each iteration — assuming that it holds after iteration k (just before iteration k+1), explain why it also holds after iteration k+1 (just before iteration k+2).
  - *Final answer.*
    Explain why the whole algorithm — setup, loop, 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 iterative algorithms, the primary avenues for improvement are to reduce the number of iterations or to reduce the work per iteration.