Chapter 2

Iterative Algorithms

Iterative algorithms involve loops, and proceed forward towards the solution one step at a time. There are two main concerns when developing an iterative algorithm –

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

2.1 15 Steps to an Iterative Algorithm

Establishing the problem. The goal of these steps is to define the task to be solved.

1. *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? For optimization problems, distinguish between legal solutions and optimal ones.

2. *Examples.*
   If needed, give examples (specific inputs and the corresponding outputs) of typical and special cases to clarify the specifications.

Brainstorming ideas. The goal of these steps is to identify avenues that may lead to a solution.

3. *Targets.*
   What are the 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 even $O(n^2)$ would likely be too slow).

4. *Tactics.*
   The time and space constraints may narrow down the algorithmic options and/or may guide you in particular directions. Consider both things you can and can't do.

5. *Approaches.*
   Consider specific iterative patterns — what would they look like if applied to this problem? (You're not yet actually coming up with an algorithm, but rather exploring whether a given pattern might be applicable and what the framework the pattern gives you would look like.)

Definining the algorithm. The goal of these steps is to define the algorithm.

Specify these steps in sufficient detail to support the level at which the algorithm's correctness is to be established — are you arguing correctness of the idea or of a specific implementation? The latter will need

more detailed pseudocode to the level of identifying variables because correctness requires that the variables have the right values.

6. *Main steps.*
   This is the core of the algorithm — the main loop, focusing on the loop body. What's being repeated?

7. *Exit condition.*
   Identify when the loop ends. (This is a boolean condition about the state.)

8. *Setup.*
   Whatever must happen before the loop starts (initialization, etc).

9. *Wrapup.*
   Whatever must happen after the loop ends to produce the final solution.

10. *Special cases.*
    Make sure the algorithm works for all legal inputs — revise the previous steps to add handling for those cases as needed.

11. *Algorithm.*
    Assemble the algorithm from the previous steps and state it.

    There shouldn't be new steps here, instead bring together steps 6–10 and state the whole algorithm.

Showing correctness. The goal of these steps is to show that the algorithm correctly solves the problem.

12. *Termination.* Show that the loop eventually terminates.

    (a) *Measure of progress.*
    What metric can be used to tell you that you are getting closer to the solution?

    (b) *Making progress.*
    Explain how each iteration of the loop changes the value of the measure of progress.

    (c) *Reaching the end.*
    Explain why making progress means that eventually the exit condition will be satisfied.

13. *Correctness.* Show that the algorithm is correct.

    (a) *Loop invariant.*
    A loop invariant is a boolean statement about the state at the start of the loop body. It purpose is to ensure that the algorithm remains on track towards the desired solution. It may be a direct statement of the correctness of the partial solution, or, especially for optimization problems, an indirect statement about a property of the partial solution that can be utilized to show that, when complete, the complete solution is optimal.

    (b) *Establish the loop invariant.*

    - Explain why the invariant is true after the setup and before the first iteration of the loop.
    - Assuming the invariant is true before the first iteration, explain why it is still true after the first iteration and before the second.

    (c) *Maintain the loop invariant.*
    Assume that the loop invariant is true at the start of an iteration, and explain why the invariant is still true at the end of that iteration.

    (d) *Final answer.*
    Explain why, with the loop invariant being true and the exit condition being false when the loop completes, the wrapup steps lead to a correct result.

Determining efficiency. The goal of these steps is to determine the running time and space requirements of the algorithm.

14. *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.

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

2.2 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 how that progress towards the solution is made. Many iterative algorithms follow one of the following three patterns:

- **process input**, where the tactic is to repeatedly process the next input item
- **produce output**, where the tactic is to repeatedly produce the next output item
- **narrowing the search space**, where the tactic is to repeatedly eliminate non-answers

Insertion sort is an example of process input — each input item is taken in turn and inserted into the group of sorted elements. Selection sort illustrates produce output — repeatedly finding the smallest remaining element produces the sorted elements in order. Binary search is a prime example of narrowing the search space.

Identifying a pattern to use provides additional direction for filling in some of the 15 steps. In particular:

Main steps. Typical forms:

- process input: for each input item, incorporate into the solution to obtain a solution for one more element
- produce output: repeatedly do the steps to produce the next output item
- narrowing the search space: repeatedly eliminate one or more elements that aren't what you are looking for

Exit condition. Typical forms:

- process input: when all of the input items have been processed
- produce output: when all of the output items have been produced
- narrowing the search space: when the item has been found or the search space is empty

Measure of progress. Typical forms:

- process input: number of input items processed
- produce output: number of output items produced
- narrowing the search space: size of the current search space

Making progress. Typical forms:

- process input: each iteration processes one more input item
- produce output: each iteration produces one more output item
- narrowing the search space: each iteration reduces the size of the search space

Termination. Typical forms:

- process input: repeatedly processing one more input item means that eventually all of the input will have been processed
- produce output: repeatedly producing one more output item means that eventually all of the output will have been produced
- narrowing the search space: repeatedly reducing the size of the search space means that it will eventually become empty (if the target hasn't been found)

Loop invariant. Typical forms:

- process input: we have a correct solution for the first k input items, or [alternatively] we haven't gone wrong yet (the solution so far is still consistent with a solution for the whole problem)
- produce output: we have produced the first k items of the (correct) output
- narrowing the search space: either the element is within the current search space or it was never present at all, or [alternatively] not all of the solutions (if there are any) have been eliminated

2.3 Example

Let's consider sorting.

> A vertical line in the margin on the left side of the page indicates what would actually be written in a solution, to distinguish it from commentary about the example.

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? For optimization problems, distinguish between legal solutions and optimal ones.

> Given a list of n elements, arrange them in non-decreasing order.
>
> Input: a list of n elements
>
> Output: the contents of the list in non-decreasing order

Examples. If needed, give examples (specific inputs and the corresponding outputs) of typical and special cases to clarify the specifications.

> $1, 6, 3, 4, 2 \rightarrow 1, 2, 3, 4, 6$
> $1, 6, 3, 4, 3 \rightarrow 1, 3, 3, 4, 6$

An example with duplicate elements is included because there is not a stated requirement that elements be unique (and "non-decreasing order" rather than "increasing order" includes the possibility of duplicates).

Targets. What are the time and space requirements for your solution?

> No time or space requirements given. We might expect to find something between $O(n)$ — it seems unlikely that n elements could be sorted without looking at each element at least once — and $O(n^2)$ — it is plausible that every element would need to be compared to every other to order them, but each pairwise comparison shouldn't have to happen more than a constant (and not dependent on n) number of times.

Tactics. The time and space constraints may narrow down the algorithmic options and/or may guide you in particular directions. Consider both things you can and can't do.

> $O(n^2)$ allows for all-pairs comparison, assuming $O(1)$ for one comparison. $O(n)$ means that each element can only be compared to a fixed number of others.
>
> Comparing two elements to determine how they should be ordered is $O(1)$ for integers, but could be larger for other types (e.g. comparing strings) and potentially needs to be factored in

— expensive comparisons may limit the number of comparisons further or dictate an approach other than comparison-based sorting.

Approaches. Consider specific iterative patterns — what would they look like if applied to this problem?

Process input — consider each element in turn and insert it into the correct place amongst those sorted so far.

Produce output — find the next smallest element and put it at the end of those sorted so far.

This isn't a search problem, so narrowing the search space doesn't apply.

Main steps. This is the core of the algorithm — the main loop, focusing on the loop body. What's being repeated?

Both the process input and produce output approaches seem viable for this problem, and neither stands out as clearly simpler or more complex that the other. Let's try process input. This pattern is "for each input item, incorporate it into the solution to obtain a solution for one more element". A solution is a sorted list of elements, so:

```
for each element
  add the element in the proper position in elements-sorted-so-far
```

Exit condition. Identify when the loop ends. (This is a boolean condition about the state.)

When all of the input items have been added to elements-so-far. (This is already incorporated in the "for each element" in the main steps.)

(This comes straight from the process input pattern.)

Setup. Whatever must happen before the loop starts (initialization, etc).

Elements-sorted-so-far is initialized to be empty.

Wrapup. Whatever must happen after the loop ends to produce the final solution.

Elements-sorted-so-far should contain all of the elements in sorted order when the loop finishes, so there is no need to do anything else.

Special cases. Make sure the algorithm works for all legal inputs — revise the previous steps to add handling for those cases as needed.

Duplicate elements is a potential special case — it's sort of covered with "add in the proper position" but it is worth clarifying what "proper position" means: Element x belongs between consecutive elements that are less than x and not less than x, respectively. x goes at the beginning if it is not bigger than the first thing and at the end if it is bigger than the last thing.

Algorithm. Assemble the algorithm from the previous steps and state it.

```
for each element
  add the element to elements-sorted-so-far, putting it at the beginning if it is
   not bigger than the first element, at the end if it is bigger than the last
   element, and otherwise between consecutive elements less than and not less than
   it, respectively
```

Termination. Show that the loop eventually terminates.

These steps often come directly from the iterative pattern.

Measure of progress. What metric can be used to tell you that you are getting closer to the solution?

The number of input elements that have been iterated through.

Making progress. Explain how each iteration of the loop changes the value of the measure of progress.

Each iteration involves a new element, increasing the number of elements that have been iterated through by 1.

Termination. Explain why making progress means that eventually the exit condition will be satisfied.

Each iteration involves a new element, so eventually there won't be any new items left.

Correctness . Show that the algorithm is correct.

Loop invariant. A loop invariant is a boolean statement about the state at the start of the loop body.

The loop invariant is involved in proving correctness, so it should be about something that will help with arguing that the solution is correct. Since what we want in the end is that elements-sorted-so-far (a) is correctly sorted and (b) contains all of the input elements, a useful loop invariant is

`elements-sorted-so-far contains the first k input elements in non-decreasing order`

where $k = 0, 1, 2, ...$ is the number of iterations completed. $k = 0$ at the beginning of the first iteration.

Establish the loop invariant. Explain why the loop invariant is true at the beginning of the first iteration, and why, if it is true at the beginning of the first iteration, why it is also true at the end of the iteration / beginning of the second iteration.

Before the first iteration, $k = 0$ input elements have been processed. Elements-sorted-so-far is initialized to an empty list, which certainly contains $k = 0$ elements in non-decreasing order.

The first iteration puts the first element into its place in the sorted list. Since the sorted list is empty so far, this just adds the element to the list. One element is certainly in non-decreasing order.

Maintain the loop invariant. Assume that the loop invariant is true at the start of an iteration, and explain why the invariant is still true at the end of that iteration.

Assume elements-sorted-so-far contains the first k input elements in non-decreasing order. Show that as a result of one more iteration of the loop body, the first $k + 1$ input elements are in non-decreasing order.

This iteration takes the next input element x and puts it into elements-sorted-so-far:

- before the first element-sorted-so-far, if element x is not bigger than it
- after the last element-sorted-so-far, if element x is bigger than it
- between two consecutive elements less than x and not less than x otherwise

All of these cases maintain the non-decreasing order: x is put before something $\geq x$ and and after something $< x$. It is also necessary to show that one of these cases always applies — the third case must always be possible if neither of the first two apply, because if the first element-sorted-so-far is smaller than x and the last element-sorted-so-far is not bigger than x, there's at least one of elements-sorted-so-far less than x and another one not less than x and in non-decreasing

order, there's a pair adjacent to each other.

Final answer. Explain why, with the loop invariant being true and the exit condition being false when the loop completes, the wrapup steps lead to a correct result.

We need to show that at the end, elements-sorted-so-far contains all n elements in non-decreasing order.

> When the loop exits, the exit condition means that all n elements have been processed and thus added to elements-sorted-so-far. The loop invariant states that the k elements in elements-sorted-so-far are in non-decreasing order. So, with $k = n$, we have all n elements in non-decreasing order.

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.

> The input is described only as "a list". For traversal, array vs linked list doesn't matter, but for inserting an element into elements-sorted-so-far, an array requires shifting. However, the insertion step follows finding the insertion location, which requires traversal for a linked list so the combination of find insertion spot and insert will be $O(n)$ for both array and linked list.

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

> The loop repeats n times (once for each input element). Inserting one element in the proper sorted order takes time $O(k)$ where k is the number of elements processed so far — $O(\log k)$ to search and $O(k)$ to insert for an array and $O(k)$ to search and $O(1)$ to insert for a linked list. Summing $O(k)$ for $k = 0..n\text{-}1$ results in $\Theta(n^2)$ time.
>
> Additional space is $O(1)$ because the sort can be implemented in-place for an array (divide it into an unsorted portion or a sorted portion) or by simply relinking nodes for a linked list.