Chapter 1

Introduction

Our strategy for algorithm development will be to identify general algorithmic strategies and try to apply a template for developing algorithms of a given type. Broadly speaking, there are two types of algorithms —

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

We will first consider strategies for developing iterative and recursive algorithms in general, and then focus on two particular algorithmic structures — divide-and-conquer, and a series of choices.

In all cases, the process will involve five main steps: establishing the problem, brainstorming ideas, defining the algorithm, showing correctness, determining efficiency.

Chapter 2

Recursive Algorithms

Iterative algorithms proceed forward towards the solution one step at a time. Recursive algorithms, by contrast, often work backwards, combining solutions from one or more subproblems to solve the whole problem. Recursive algorithms can be thought of as getting help from your friends — you hand off smaller problems to your friends (who somehow solve those problems) and then use the solutions provided by your friends to produce a solution to your problem.

2.1 16 Steps to a Recursive 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.

3. *Size.*
   What is the size of the problem, and what constitutes a smaller problem? What is the simplest/smallest instance of the problem?

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

4. *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).

5. *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.

6. *Approaches.*
   Consider specific recursive 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.

7. *Generalize / define subproblems.*
   Friends get smaller versions of the original problem, which often takes the form of a generalized version of the original problem. (For example, doing the original task on a portion of the original input is a generalized version of the original problem — the specific task is to work with all of the input, while the generalized version works with any portion of the input, including all of it.) Define the generalized problem, its input, and its output along with pre- and postconditions. Make sure that everything the friend needs or hands back should be covered by the input(s) and output(s) — avoid global variables and global effects.

8. *Base case(s).*
   Address how to solve the smallest problem(s). This is often trivial, or is solved via brute force.

9. *Main case.*
   The main case addresses how to solve a typical large problem instance. Specify how many friends are needed and what is handed to each friend, as well as how to generate what is handed to the friends and how the results the friends hand back are combined to solve the problem.

10. *Top level.*
    The top level puts the context around the recursion.

    (a) *Initial subproblem.*
    Specify the inputs and parameters for the initial subproblem — the one whose solution solves the original problem.

    (b) *Setup.*
    Whatever must happen before the initial subproblem is solved.

    (c) *Wrapup.*
    Whatever must happen to get the final answer after the solution for the initial subproblem is obtained.

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

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

    There shouldn't be new steps here, instead bring together steps 8–11 and state the whole algorithm.

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

13. *Termination.* Show that the recursion always terminates.

    (a) *Making progress.*
    Explain why what each of your friends get is a smaller instance of the problem.

    (b) *Reaching the end.*
    Explain why a base case is always reached.

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

(a) *Establish the base case(s).*
Explain why the solution is correct for each base case.

(b) *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.

(c) *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.

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

15. *Implementation.*
Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic steps that haven't been specified.

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

2.2 Recursive Patterns

Recursive algorithms can be categorized by how many subproblems are generated and what each friend gets.

- **1-friend solutions**, where there is only one subproblem at each level
  - **constant amount**
    The subproblem is smaller by a fixed number of elements, typically 1. For example, computing a^n as aa^{n-1} and $n!$ as $n(n-1)!$.
  - **constant factor**
    The subproblem is a fixed fraction (typically 1/2) of the size of the original. For example, binary search (splits in half) and computing a^n as $(a^{n/2})^2$ if n is even and $a(a^{(n-1)/2})^2$ if n is odd.
  - **variable factor**
    The subproblem is always smaller, but the size reduction varies from one step to the next. For example, computing $\gcd(m,n)$ as $\gcd(n, m \mod n)$.

  1-friend recursive algorithms are also sometimes called "decrease and conquer" algorithms. They can often be written more simply and efficiently as an iterative algorithm.

- **2+-friend solutions**, where there is more than one subproblem at each level
  - **divide-and-conquer**
    The problem is split into b subproblems of size n/b where $b \geq 2$ (typically 2).
  - **case analysis**
    Each friend considers a different choice.

2.3 Running Time

A *recurrence relation* is an equation which is defined in terms of itself; they arise naturally when analyzing recursive algorithms. In particular, recursive algorithms tend to lead to recurrence relations in one of two forms: splitting off b elements or dividing into subproblems of size n/b.

Split off b elements. For recurrence relations of the form $T(n) = aT(n-b) + f(n)$ where $f(n) = 0$ or $f(n) = \Theta(n^c \log^d n)$:

a	$f(n)$	**behavior**	**solution**
> 1	any	base case dominates	$T(n) = \Theta(a^{n/b})$
1	≥ 1	all levels are important	$T(n) = \Theta(nf(n))$

The solution cases are based on the number of subproblems and $f(n)$. There are two behaviors:

- Base case dominates — there are too many leaves.
- All levels are important — there are $O(n)$ steps to get to the base case, and roughly the same amount of work in each level.

Divide into subproblems of size n/b. For recurrence relations of the form $T(n) = aT(n/b) + f(n)$ where $f(n) = \Theta(n^c \log^d n)$:

$(\log a)/(\log b)$ **vs** c	d	**behavior**	**solution**
$<$	any	top level dominates	$T(n) = \Theta(f(n))$
$=$	> -1	all levels are important	$T(n) = \Theta(f(n) \log n)$
$=$	< -1	base cases dominate	$T(n) = \Theta(n^{(\log a)/(\log b)})$
$>$	any	base cases dominate	$T(n) = \Theta(n^{(\log a)/(\log b)})$

The solution cases are based on the relationship between the number of subproblems, the problem size, and $f(n)$. There are three behaviors:

- Top level dominates — there is more work splitting/combining than in subproblems, making the root too expensive.
- All levels are important — there are $\log n$ steps to get to the base case, and roughly the same amount of work in each level.
- Base cases dominate — there are so many subproblems that taking care of all the base cases is more work than splitting/combining (too many leaves).

Part II

Divide-and-Conquer

Chapter 3

Divide and Conquer

Divide-and-conquer algorithms are recursive algorithms where a problem is divided into b subproblems of size n/b where $b \geq 2$ (typically 2).

3.1 Patterns

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

- **Process input**, 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
- **Produce output**, 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

For search problems, where the goal is to find an element in a collection, another pattern is applicable:

- **Narrow the search space**, where each friend searches a different part of the search space

3.2 Developing Divide-and-Conquer Algorithms

Divide-and-conquer algorithms are recursive, so the 15-step recursive algorithm development template can be used. However, there are some patterns specific to divide-and-conquer algorithms that can help guide the process.

Targets. Divide-and-conquer algorithms often seek to improve on a polynomial-time iterative brute-force running time. Identify the brute force algorithm and its running time as part of this step step. (It is possible to go through the iterative process to develop this algorithm and prove it correct, but generally correctness is apparent i.e. all possibilities are exhaustively checked.)

Approaches. Consider the divide-and-conquer patterns (process input, produce output, narrow the search space) — of those that are applicable, what do they look like in terms of this problem?

3.3 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.

Task: Sort the elements of an array.
Input: an array A of n elements
Output: the array with the elements in non-decreasing order

Size. What is the size of the problem, and what constitutes a smaller problem? What is the simplest/smallest instance of the problem?

The problem size is the number of elements to sort. The smallest problem is 1 element.

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

Iterative sorting algorithms like insertion sort, selection sort, and others are $O(n^2)$.

Tactics.

Beating $O(n^2)$ means that it is not possible to compare every pair of elements directly — we must be able to effectively compare one element to multiple elements with a single comparison.

Approaches.

A process input approach would mean splitting the input in half and sorting each half, then combining the two sorted halves. A produce output approach would mean generating a sorted list of the smaller values and a sorted list of the larger values, so the work is separating the smaller and larger elements in the input.

Generalize / define subproblems. Friends get smaller versions of the original problem, which often takes the form of a generalized version of the original problem.

A smaller problem means fewer elements — so sorting part of the array instead of the whole thing.

Generalized task: sort a section of the array
Input: array `A` and indexes `low` and `high` defining the region to sort
Output: elements `A[low..high]` arranged in non-decreasing order

Base case(s). Address how to solve the smallest problem(s).

The base case occurs when `low` = `height`. In that case, there is only one element and it is sorted, so there's nothing to do.

Main case. The main case addresses how to solve a typical large problem instance. Specify how many friends are needed and what is handed to each friend, as well as how to generate what is handed to the friends and how the results the friends hand back are combined to solve the problem.

A process input approach means to split the input in half in the most straightforward way, have the friends solve the resulting problems, and then figure out how to use the friends' results to get our answer. Thus

```
split A[low..high] in half
have a friend sort each half: A[low..(low+high)/2] and A[(low+high)/2+1..high]
merge the sorted halves into a single sorted array
```

where "merge" means to move through the two halves, each time taking the smaller of the two current elements for the sorted result.

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.

Sort `A[0..n-1]`.

Setup. Whatever must happen before the initial subproblem is solved.

Nothing else is needed.

Wrapup. Whatever must happen to get the final answer after the solution for the initial subproblem is obtained.

Nothing else is needed — the sorted array is the answer.

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

$n = 0$ is not addressed; 0 elements are already sorted so there is nothing to do.

Duplicate elements are not ruled out in the input. That means there may not be a smaller of the current elements during the merging step, but if the values are the same, either may be taken as the next sorted element.

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

```
sort A[0..n-1]
```

where sort `A[low..high]` is

```
if ( low < high )
  split A[low..high] in half
  have a friend sort each half: A[low..(low+high)/2] and A[(low+high)/2+1..high]
  merge the sorted halves into a single sorted array
```

(Note that with `A` itself being sorted, there's nothing to do in the base case — that entry is already sorted.)

Termination. Show that the recursion always terminates.

Making progress. Explain why what each of your friends get is a smaller instance of the problem.

Observe that if low < high (i.e. not a base case), low ≤ (low+high)/2 < high. Thus the range low..(low+high)/2 doesn't contain 'high' and the range (low+high)/2+1..high doesn't contain 'low', so each friend's problem is at least one smaller than ours.

Reaching the end. Explain why a base case is always reached.

Starting with $n \geq 2$ and dividing in half each time (integer division) means we can't skip 1 — if n is even, then $n = 2k$ for some integer k and $n/2 = 2k/2 = k$; if n is odd, then $n = 2k + 1$ for some integer k and $\lfloor n/2 \rfloor = \lfloor (2k + 1)/2 \rfloor = k$. With $k \geq 1$ because $n \geq 2$, there's at least 1 element in each half.

Establish the base case(s). Explain why the solution is correct for each base case.

The base case is a single element, which is always sorted.

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.

Every element is given to exactly one of the friends, so nothing is left out or double-counted.

The friends return their elements in sorted order; the merge process combines the two groups into a single sorted order.

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.

There isn't any setup or wrapup, and the initial subproblem `A[0..n-1]` includes all of the original elements, so sorting those elements sorts all of `A`.

Implementation. Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic steps that haven't been specified.

The merge step: start a current index at the beginning of each half, then repeatedly take the smaller of the two current elements and advance that index.

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

The base case takes O(1) time. For the main case (n elements), consider each step:

- Dividing the array is O(1) (to compute (low+high)/2).
- Two problems of size $n/2$ are generated.
- Merging two sorted arrays with $n/2$ elements each takes O(n) time — one of the current indexes is advanced in each iteration, so there are a total of n iterations.

This leads to the recurrence relation $T(n) = 2T(n/2) + n$. With T(1) = 1, this yields $T(n) = \Theta(n \log n)$.

Space requirements are O(1) for a few temporary variables such as two current indexes used in merging and O(n) for a second array for the merged results.