Chapter 4

Recursive Algorithms

Recursive algorithms rely on friends: to solve a problem, one or more smaller problems are identified and handed off to others (your friends) to be solved (somehow), and then those subproblem solutions are used to produce a solution for the current problem.

4.1 Recursive Patterns

Recursive algorithms can be categorized by the number and size of the subproblems.

- **1-friend solutions**, where there is only one subproblem at each level
  - **constant amount**
    The subproblem is smaller by a fixed amount, 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 $\gcd(n, m \mod n)$.
- **2+-friend solutions**, where there is more than one subproblem at each level
  - **divide and conquer**
    Each friend gets a portion of the input: the problem is split into b subproblems of size n/b where $b \geq 2$ (typically 2).
  - **case analysis**
    Each friend gets a subproblem resulting from a different alternative for the next step of the solution: the problem is split into a subproblems of size $n-b$ where a is the number of alternatives and b is typically 1.

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.

Recursive algorithms can also be categorized by the nature of the subproblem. Two forms are common:

- *solve a smaller problem*, where the subproblem is treated as a standalone instance of the problem to be solved, and
- *solve the rest of the problem*, where the subproblem is solved in the context of actions taken up to this point — solving the subproblem completes a partial solution already begun.

The 1-friend and divide-and-conquer patterns typically utilize the "solve a smaller problem" form, while case analysis patterns utilize the "solve the rest of the problem" form.

4.2 Elements of Recursive Algorithms

4.2.1 Size

Recursion works by solving smaller instances of the original problem, then combining or merging the solutions from the smaller instances to obtain the solution for the current problem instance. "Size" refers to some measure of the input that leads to a simpler problem instance when decreased, for example, when the input is a collection of things, the size is typically the number of elements. For strings or numbers, size is often the length of the string or the magnitude (value) of the number. For trees, size may be the number of nodes or the height of the tree. For graphs, size may be the number of vertices or edges.

4.2.2 Subproblems

Defining subproblems means specifying the task, the input, and the output.

A subproblem's task is a generalized version of the original task — for "solve a smaller problem", the original task is to solve the problem for the whole input while the generalized version is to solve the problem for part of the input. For "solve the rest of the problem", the original task is to solve the problem from scratch while the generalized version is to solve the problem in light of the solution so far.

For the subproblem's input, prefer adding what is needed to specify which part of the input the subproblem is to work with rather than assuming the subproblem works with the whole of a smaller instance. In addition, the input for "rest of the problem" subproblem typically includes at least some aspects of the partial solution so far.

The subproblem's output needs to include what is needed for the original problem, but can include additional things. Every subproblem must returns the same type of output, however, so the additional things must be appropriate for all subproblems.

4.2.3 Termination and Correctness

Recursion terminates when a base case is reached. As a result, we must be making progress towards a solution with every handoff to a friend — since the base case is the smallest problem instance, every subproblem handed to a friend must be smaller than our subproblem. We also have to actually reach a base case — it can't be possible to skip over a base case.

Showing correctness for recursive algorithms is based on the proof technique of induction. For induction it is necessary to show that the correct answer is produced for the base cases and that given correct solutions for the subproblems, the solution produced for the current instance is correct.

4.2.4 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).

4.2.5 Room For Improvement

The running time for recursive algorithms comes from the number and size of the subproblems as well as the amount of work done in addition to the recursive calls themselves. Strategies for reducing the additional work done include seeking a more efficient implementation and leveraging the work the friends are already doing by having them hand back something additional that saves us computation. For divide and conquer, reducing the number and size of the subproblems means a decrease-and-conquer approach — can a guaranteed number of elements *not* be passed along to a friend? For case analysis, reducing the number of subproblems means being able to eliminate some cases from consideration because they are redundant or won't lead to a viable solution.

4.3 How to Design Recursive 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 recursive 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 recursive 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 recursive patterns defined in section 4.1 — can they be applied to this problem? What would such an approach look like?

Define the algorithm.

The core of a recursive algorithm is defining the subproblems.

- *Generalize / define subproblems.*
  Define the subproblem — the task, the input, and the output. Write this as a function definition where parameters and return values are explicitly identified.
- *Base case(s).*
  Address how to solve the smallest problem(s). This is often trivial, or is solved via brute force.
- *Main case.*
  Address how to solve a typical large problem instance (one that is not a base case). Specify how many friends are needed and what subproblem is handed to each friend, as well as how the results the friends hand back are combined to solve the problem.
- *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.
  - *Setup.*
    Whatever must happen before the initial subproblem is solved.
  - *Wrapup.*
    Whatever must happen to get the final answer after the solution for the initial subproblem is obtained.
- *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 new elements here, instead bring together the base case(s), main case, and top level 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 recursive algorithms, show that the base case is always reached (and thus the recursive always terminates).

- *Size.*
  What is the size of the problem?
- *Making progress.*
  Explain why what each of your friends get is a smaller instance of the problem.
- *The end is reached.*
  Explain why a base case is always reached.

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

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

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.

  Common avenues for improvement for recursive algorithms are discussed in section 4.2.5.