Chapter 1

Problem Solving Patterns

> It's not that I'm so smart, it's just that I stay with problems longer.
>
> — A. Einstein

Developing problem solving skills is like learning to play a musical instrument—books and teachers can point you in the right direction, but only your hard work will take you there. Just like a musician, although you need to know underlying concepts, theory is no substitute for practice. For this reason, EPI consists primarily of problems.

Great problem solvers have skills that cannot be rigorously formalized. Still, when faced with a challenging programming problem, it is helpful to have a small set of "patterns"—general reusable solutions to commonly occurring problems—that may be applicable.

We now introduce a number of patterns, and illustrate them with examples. We have classified these patterns into three categories:

- data structure patterns,
- algorithm design patterns, and
- abstract analysis patterns.

These patterns are summarized in Table 1.1 on Page 9, Table 1.2 on Page 14, and Table 1.3 on Page 22, respectively.

The notion of patterns is very general; in particular, there are many patterns that arise in the context of software design—the builder pattern, composition, publish-subscribe, etc. These are more suitable to large-scale systems, and as such are outside of the scope of EPI, which is focused on smaller programs that can be solved in an interview.

Data structure patterns

A data structure is a particular way of storing and organizing related data items so that they can be manipulated efficiently. Usually, the correct selection of data structures is key to designing a good algorithm. Different data structures are suited to different of applications; some are highly specialized. For example, heaps are particularly well-suited for algorithms that merge different sorted data streams, while compiler implementations usually use hash tables to look up identifiers.

A solution will often require a combination of data structures. For example, our solution to the problem of tracking the most visited pages on a website (Solution 9.12 on Page 82) involves a combination of a heap, a queue, a BST, and a hash table.

Primitive types

You should be comfortable with the basic types (chars, integers, doubles, etc.), their variants (unsigned, long, etc.), and operations on them (bitwise operators, comparison, etc.). Don't forget that the basic types differ between programming languages. For example, there are no unsigned integers in Java, and the number of bits in an integer is compiler- and machine-dependent in C.

A common problem related to basic types is computing the number of bits set to 1 in an integer-valued variable x. To solve this problem, you need to know how to manipulate individual bits in an integer. One straightforward approach is to iteratively test individual bits, using an unsigned integer variable m initialized to 1. Iteratively identify bits of x that are set to 1 by examining the bitwise AND of m with x, shifting m left by one bit at a time. The overall complexity is $O(n)$ where n is the length of the integer.

Another approach, that may run faster on some inputs, is based on computing $y = x\&(x-1)$, where & is the bitwise AND operator. This is 1 at exactly the rightmost bit of x. Consequently, this bit can be removed from x by computing $x \oplus y$. The time complexity is $O(s)$, where s is the number of bits set to 1 in x.

In practice, if the computation is performed repeatedly, the most efficient approach would be to create a lookup table. For example, we could use a 256 entry integer-valued array P such that $P[i]$ is the number of bits set to 1 in i. If x is 32 bits, the result can be computed by decomposing x into 4 disjoint bytes, $b3, b2, b1,$ and $b0$. The bytes are computed using bitmasks and shifting, e.g., $b1$ is `(x & 0xff00) » 8`. The final result is $P[b3] + P[b2] + P[b1] + P[b0]$. Computing the parity of an integer is closely related to counting the number of bits set to 1, and we present a detailed analysis of the parity problem in Solution 2.1 on Page 171.

Arrays

Conceptually, an array maps integers in the range $[0, n-1]$ to objects of a given type, where n is the number of objects in this array. Array lookup and insertion are very fast, making arrays suitable for a variety of applications. Reading past the last element of an array is a common error, invariably with catastrophic consequences.

Table 1.1: Data structure patterns.

Data structure	Key points
Primitive types	Know how `int`, `char`, `double`, etc. are represented in memory and the primitive operations on them.
Arrays	Fast access for element at an index, slow lookups (unless sorted) and insertions. Be comfortable with notions of iteration, resizing, partitioning, merging, etc.
Strings	Know how strings are represented in memory. Understand basic operators such as comparison, copying, matching, joining, splitting, etc.
Lists	Understand trade-offs with respect to arrays. Be comfortable with iteration, insertion, and deletion within singly and doubly linked lists. Know how to implement a list with dynamic allocation, and with arrays.
Stacks and queues	Understand insertion and deletion. Know array and linked list implementations.
Binary trees	Use for representing hierarchical data. Know about depth, height, leaves, search path, traversal sequences, successor/predecessor operations.
Heaps	Key benefit: $O(1)$ lookup find-min, $O(\log n)$ insertion, and $O(\log n)$ deletion of min. Node and array representations. Max-heap variant.
Hashes	Key benefit: $O(1)$ insertions, deletions and lookups. Key disadvantages: not suitable for order-related queries; need for resizing; poor worst case performance. Understand implementation using array of buckets and collision chains. Know hash functions for integers, strings, objects. Understand importance of equals function. Variants such as Bloom filters.
BSTs	Key benefit: $O(\log n)$ insertions, deletions, lookups, find-min, find-max, successor, predecessor when tree is balanced. Understand implementation using nodes and pointers. Be familiar with notion of balance, and operations maintaining balance. Know how to augment a BST, e.g., interval trees and dynamic order statistics.

The following problem arises when optimizing quicksort: given an array A whose elements are comparable, and an index i, reorder the elements of A so that the initial elements are all less than $A[i]$, and are followed by elements equal to $A[i]$, which in turn are followed by elements greater than $A[i]$, using $O(1)$ space.

The key to the solution is to maintain two regions on opposite sides of the array that meet the requirements, and grow these regions one element at a time. Details are given in Solution 3.1 on Page 181.

Strings

Strings are ubiquitous in programming today—scripting, web development, bioinformatics all make extensive use of strings. The following problem illustrates some of the intricacies of string manipulation.

Suppose you are asked to write a function that takes as input a string s over the letters {"a", "b", "c", "d"}, and replaces each "a" by "dd" and deletes each "b". It is straightforward to implement such a function if it can allocate $O(|s|)$ additional storage. However, if you are allowed to use only $O(1)$ additional storage, the problem becomes more challenging. It is given that s is stored in an array that has enough space for the final result.

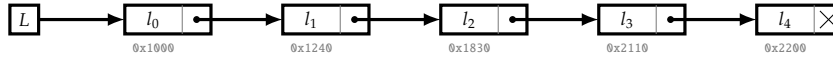
One approach is to make a first pass through s in which we delete each "b" and count the number of "a"s. We then make a second pass working backwards from the end of the current string, copying characters to the end of the result string (whose size we know from the number of "a"s), replacing each "a", by "dd". Details are given in Solution 3.10 on Page 189.

Lists

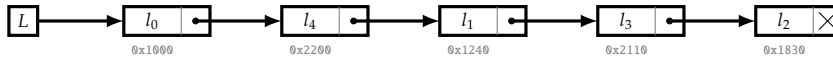
An abstract data type (ADT) is a mathematical model for a class of data structures that have similar functionality. Strictly speaking, a list is an ADT, and not a data structure. It implements an ordered collection of values, which may be repeated. In the context of this book, we view a list as a sequence of nodes with links to the next node in the sequence. In a doubly linked list each node additionally has links to the prior node.

A list is similar to an array in that it contains objects in a linear order. The key difference is that inserting and deleting elements has time complexity $O(1)$; obtaining the k-th element in a list is expensive, having $O(n)$ time complexity. Lists are usually building blocks of more complex data structures. However, they can be the subject of tricky problems in their own right, as illustrated by the following:

Given a singly linked list $\langle l_0, l_1, l_2, \ldots, l_{n-1} \rangle$, assuming n is even, define the "zip" of the list to be $\langle l_0, l_{n-1}, l_1, l_{n-2}, \ldots, l_{\frac{n}{2}-1}, l_{\frac{n}{2}} \rangle$. Suppose you were asked to write a function that computes the zip of a list, with the constraint that it should not use any additional storage (either by explicit allocation on the heap, or via the program stack) beyond a few words.



(a) List before zipping. The number in hex below each node represents its address in memory.



(b) List after zipping. Note that nodes are reused—no memory has been allocated.

Figure 1.1:

The solution is based on an appropriate iteration combined with "pointer swapping", i.e., updating next and previous fields for each node. Refer to Solution 4.11 on Page 213 for details.

Stacks and queues

Stacks support last-in, first-out semantics for inserts and deletes, whereas queues are first-in, first-out. Both are ADTs, and are commonly implemented using linked lists or arrays. Like lists, they are usually building blocks in a more complex setting, but can make for interesting problems in their own right.

As an example, consider the problem of evaluating Reverse Polish notation expressions, i.e., expressions of the form "$3, 4, \times, 1, 2, +, +$", "$1, 1, +, -2, \times$", or "$4, 6, /, 2, /$". A stack is ideal for this purpose—operands are pushed on the stack, and popped as operators are processed, with intermediate results being pushed back onto the stack. Details are given in Solution 5.2 on Page 216.

Binary trees

A binary tree is a data structure that can represent hierarchical relationships. Binary trees most commonly occur in the context of binary search trees, wherein keys are stored in a sorted fashion. However, there are many other applications of binary trees. For example, consider a set of resources organized as nodes in a binary tree. Processes need to be able to lock resource nodes. A node can be locked if and only if none of its descendants and ancestors are locked. Your task is to design and implement an application programming interface (API) for locking.

A reasonable API is one with `isLock()`, `lock()`, and `unLock()` methods. Naïvely implemented the time complexity for these methods is $O(n)$, where n is the number of nodes. However, these can be made to run in time $O(1)$, $O(h)$, and $O(h)$ respectively, if nodes have a parent field. Details are given in Solution 6.4 on Page 231.

Heaps

A heap is a data structure based on a binary tree. It efficiently implements an ADT called a priority queue. A priority queue resembles a queue, with one difference:

each element has a "priority" associated with it, and deletion removes the element with the highest priority.

Suppose you are given a set of files, each containing stock trade information. Each trade appears as a separate line containing information about that trade. Lines begin with an integer-valued timestamp, and lines within a file are sorted in increasing order of timestamp. Suppose you were asked to design an algorithm that combines these trades into a single file R in which trades are sorted by timestamp.

This problem can be solved by a multistage merge process, but there is a trivial solution based on a min-heap data structure. Entries are trade-file pairs and are ordered by the timestamp of the trade. Initially the min-heap contains the first trade from each file. Iteratively delete the minimum entry $e = (t, f)$ from the min-heap, write t to R, and add in the next entry in the file f. Details are given in Solution 7.1 on Page 243.

Hashes

A hash is a data structure used to store keys, optionally with corresponding values. It implements constant time inserts, deletes and lookups. One caveat is that these operations require a good hash function—a mapping from the set of all possible keys to the integers that is similar to a uniform random assignment. Another is that if the number of keys that is to be stored is not known in advance then the hash needs to be periodically resized, which, depending on how the resizing is implemented, can lead to some updates having $\Theta(n)$ complexity.

Suppose you were asked to write an application that compares n programs for plagiarism. Specifically, your application is to break every program into overlapping character strings, each of length 100, and report on the number of strings that are in common to the two programs. Hashing can be used to perform this check very efficiently if the right hash function is used. Details are given in Solution 9.14 on Page 284.

Binary search trees

Binary search trees (BSTs) are used to store objects which are comparable. The underlying idea is to organize the objects in a tree in which the nodes satisfy the BST property: the key stored at any node is greater than or equal to the keys stored in its left subtree and less than or equal to the keys stored in its right subtree. Insertion and deletion can be implemented so that the height of the BST is $O(\log n)$, leading to fast ($O(\log n)$) lookup and update times. AVL trees and red-black trees are BST implementations that support this form of insertion and deletion.

BSTs are a workhorse of data structures and can be used to solve almost every data structures problem reasonably efficiently. It is common to augment the BST to make it possible to manipulate more complicated data, e.g., intervals, and efficiently support more complex queries, e.g., the number of elements in a range.

As an example application of BSTs, consider the following problem. You are given a set of line segments. Each segment is a closed interval $[l_i, r_i]$ of the x-axis, a color, and a height. For simplicity, assume no two segments whose intervals overlap have the same height. When the x-axis is viewed from above, the color at point x on the x-axis is the color of the highest segment that includes x. (If there is no segment containing x, the color is blank.) You are to implement a function that computes the sequence of colors as seen from the top.

The key idea is to sort the endpoints of the line segments and do a sweep from left-to-right. As we do the sweep, we maintain a list of line segments that intersect the current position as well as the highest line and its color. To quickly lookup the highest line in a set of intersecting lines, we keep the current set in a BST, with the interval's height as its key. Details are given in Solution 11.15 on Page 319.

Other data structures

The data structures described above are the ones most commonly used. There are many other data structures that have more specialized applications. Some examples include:

- Skip lists, which store a set of comparable items using a hierarchy of sorted linked lists. Lists higher in the hierarchy consist of increasingly smaller subsequences of the items. Skip lists implement the same functionality as balanced BSTs, but are simpler to code and faster, especially when used in a concurrent context.
- Treaps, which are a combination of a BST and a heap. When an element is inserted into a treap, it is assigned a random key that is used in the heap organization. The advantage of a treap is that it is height balanced with high probability and the insert and delete operations are considerably simpler than for deterministic height balanced trees such as AVL and red-black trees.
- Fibonacci heaps, which consist of a series of trees. Insert, find minimum, decrease key, and merge (union) run in constant amortized time; delete and delete-minimum take $O(\log n)$ time. In particular Fibonacci heaps can be used to reduce the time complexity of Dijkstra's shortest path algorithm from $O((|E|+|V|)\log|V|)$ to $O(|E|+|V|\log|V|)$.
- Disjoint-set data structures, which are used to manipulate subsets. The basic operations are union (form the union of two subsets), and find (determine which set an element belongs to). These are used in a number of algorithms, notably in computing the strongly connected components of an undirected graph (Solution 13.5 on Page 371).
- Tries, which are a tree-based data structure used to store strings. Unlike BSTs, nodes do not store keys; instead, the node's position in the tree determines the key it is associated with. Tries can have performance advantages with respect to BSTs and hash tables; they can also be used to solve the longest matching prefix problem (Solution 16.3 on Page 406).

Table 1.2: Algorithm design patterns.

Technique	Key points
Sorting	Uncover some structure by sorting the input.
Divide and conquer	Divide the problem into two or more smaller independent subproblems and solve the original problem using solutions to the subproblems.
Recursion	If the structure of the input is defined in a recursive manner, design a recursive algorithm that follows the input definition.
DP	Compute solutions for smaller instances of a given problem and use these solutions to construct a solution to the problem.
Incremental improvement	Quickly build a feasible solution and improve its quality with small, local updates.
Elimination	Identify and rule out potential solutions that are suboptimal or dominated by other solutions.
Parallelism	Decompose the problem into subproblems that can be solved independently on different machines.
Caching	Store computation and later look it up to save work.
Randomization	Use randomization within the algorithm to reduce complexity.
Approximation	Efficiently compute a suboptimum solution that is of acceptable quality.
State	Identify an appropriate notion of state.

Algorithm design patterns

Sorting

Certain problems become easier to understand, as well as solve, when the input is sorted. The solution to the calendar rendering problem (Problem 10.10 on Page 88) entails taking a set of intervals and computing the maximum number of intervals whose intersection is nonempty. Naïve strategies yield quadratic run times. However, once the interval endpoints have been sorted, it is easy to see that a point of maximum overlap can be determined by a linear time iteration through the endpoints.

Often it is not obvious what to sort on—for example, we could have sorted the intervals on starting points rather than endpoints. This sort sequence, which in some respects is more natural, does not work. However, some experimentation with it will likely lead to the correct criterion.

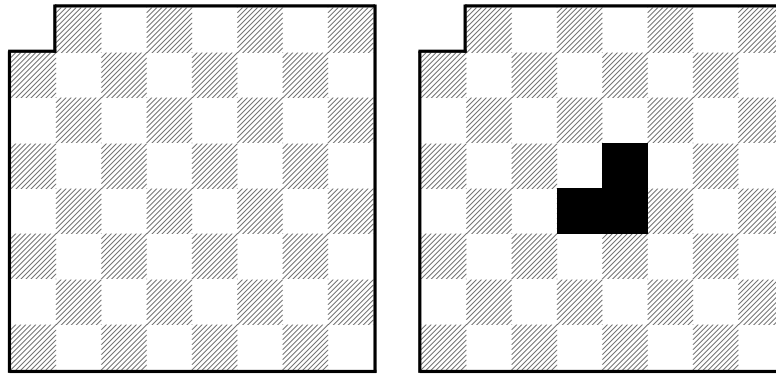
Sorting is not appropriate when an $O(n)$ (or better) algorithm is possible, e.g., determining the k-th largest element (Problem 8.13 on Page 75). Furthermore, sorting can obfuscate the problem. For example, given an array A of numbers if we are to

determine the maximum of $A[i] - A[j]$, for $i < j$, sorting destroys the order and complicates the problem.

Divide and conquer

A divide and conquer algorithm works by decomposing a problem into two or more smaller independent subproblems, until it gets to instances that are simple enough to be solved directly; the results from the subproblems are then combined. More details and examples are given in Chaper 12 on Page 99; we illustrate the basic idea below.

A triomino is formed by joining three unit-sized squares in an L-shape. A mutilated chessboard (henceforth 8×8 Mboard) is made up of 64 unit-sized squares arranged in an 8×8 square, minus the top left square, as shown in Figure 1.2(a). Suppose you are asked to design an algorithm which computes a placement of 21 triominoes that covers the 8×8 Mboard. Since there are 63 squares in the 8×8 Mboard and we have 21 triominoes, a valid placement cannot have overlapping triominoes or triominoes which extend out of the 8×8 Mboard.



(a) An 8×8 Mboard. (b) Four 4×4 Mboards.

Figure 1.2: Mutilated chessboards.

Divide and conquer is a good strategy to attack this problem. Instead of the 8×8 Mboard, let's consider an $n \times n$ Mboard. A 2×2 Mboard can be covered with one triomino since it is of the same exact shape. You may hypothesize that a triomino placement for an $n \times n$ Mboard with the top left square missing can be used to compute a placement for an $(n + 1) \times (n + 1)$ Mboard. However you will quickly see that this line of reasoning does not lead you anywhere.

Another hypothesis is that if a placement exists for an $n \times n$ Mboard, then one also exists for a $2n \times 2n$ Mboard. This approach does work. Take four $n \times n$ Mboards and arrange them to form a $2n \times 2n$ square in such a way that three of the Mboards have their missing square set towards the center and one Mboard has its missing

square outward to coincide with the missing corner of a $2n \times 2n$ Mboard, as shown in Figure 1.2(b) on the previous page. The gap in the center can be covered with a triomino and, by hypothesis, we can cover the four $n \times n$ Mboards with triominoes as well. Hence a placement exists for any n that is a power of 2. In particular, a placement exists for the $2^3 \times 2^3$ Mboard; the recursion used in the proof directly yields the placement.

In addition to divide and conquer, we used the generalization principle above. The idea behind generalization is to find a problem that subsumes the given problem and is easier to solve. We used it to go from the 8×8 Mboard to the $2^n \times 2^n$ Mboard.

Other examples of divide and conquer include counting the number of pairs of elements in an array that are out of sorted order (Solution 12.2 on Page 325) and computing the closest pair of points in a set of points in the plane (Solution 12.3 on Page 326).

Recursion

A recursive function consists of base cases, and calls to the same function with different arguments. A recursive algorithm is appropriate when the input data is naturally implemeted using recursive functions. Divide and conquer is usually implemented using recursion. However, the two concepts are not synonymous. Recursion is more general; there is no concept of the subproblems being of the same form. Indeed, in theory, all computation can be defined using recusion.

String matching exemplifies the use of recursion. Suppose you were asked to write a Boolean-valued function which takes a string and a matching expression, and returns `true` iff the string "matches" the matching expression. Specifically, the matching expression is itself a string, and could be

- x where x is a character, for simplicity assumed to be a lower-case letter (matches the string "x").
- . (matches any string of length 1).
- $x*$ (matches the string consisting of zero or more occurrences of the character x).
- $.*$ (matches the string consisting of zero or more of any characters).
- r_1r_2 where r_1 and r_2 are regular expressions of the given form (matches any string that is the concatenation of strings s_1 and s_2, where s_1 matches r_1 and s_2 matches r_2).

This problem can be solved by checking a number of cases based on the first one or two characters of the matching expression, and recursively matching the rest of the string. Details are given in Solution 3.13 on Page 192.

Dynamic Programming

Dynamic Programming (DP) is applicable when the problem has the "optimal substructure property", that is, it is possible to reconstruct a solution to the given instance from solutions to subinstances of smaller problems of the same type. A key aspect

of DP is maintaining a cache of solutions to subinstances. DP can be implemented recursively (in which case the cache is typically a dynamic data structure such as a hash or a BST), or iteratively (in which case the cache is usually a one- or multi-dimensional array). It is most natural to design a DP algorithm using recursion. Usually, but not always, it is more efficient to implement it using iteration.

As an example of the power of DP, consider the problem of determining the number of combinations of 2, 3, and 7 point plays that can generate a score of 222. Let $C(s)$ be the number of combinations that can generate a score of s. Then $C(222) = C(222-7) + C(222-3) + C(222-2)$, since a combinations ending with a 2 point play is different from one ending with a 3 point play, a combinations ending with a 3 point play is different from one ending with a 7 point play, etc.

The recursion breaks down for small scores. Specifically, there are two boundary conditions: (1.) $s < 0 \implies C(s) = 0$, and (2.) $s = 0 \implies C(s) = 1$.

Implementing the recursion naïvely results in multiple calls to the same subinstance. For example, let $C(a) \to C(b)$ indicate that a call to C with input a directly calls C with input b. Then $C(213)$ will be called in the order $C(222) \to C(222-7) \to C((222-7)-2)$, as well as $C(222) \to C(222-3) \to C((222-3)-3) \to C(((222-3)-3)-3)$.

This phenomenon results in the run time increasing exponentially with the size of the input. The solution is to store previously computed values of C in an array of length 223. Details are given in Solution 12.15 on Page 344.

Sometimes, it is profitable to study the set of partial solutions. Specifically, it may be possible to "prune" dominated solutions, i.e., solutions which cannot be better than previously explored solutions. The candidate solutions are referred to as the "efficient frontier" that is propagated through the computation.

For example, if we are to implement a stack that supports a max operation, which returns the largest element stored in the stack, we can record for each element in the stack what the largest value stored at or below that element is by comparing the value of that element with the value of the largest element stored below it. Details are given in Solution 5.1 on Page 215. The largest rectangle under the skyline (Problem 12.8 on Page 334) provides a more sophisticated example of the efficient frontier concept.

Another consideration is how the partial solutions are organized. For example, in the solution to the longest nondecreasing subsequence problem 12.6 on Page 330, it is better to keep the efficient frontier sorted by length of each subsequence rather than its final index.

Incremental improvement

When you are faced with the problem of computing an optimum solution, it is often easy to come up with a candidate solution. This solution can be incrementally updated to make it optimum. This is especially true when a solution has to satisfy a set of constraints.

As an example, consider a department with n graduate students and n professors. Each student begins with a rank ordered preference list of the professors based on

how keen he is to work with each of them. Each professor has a similar preference list of students. Suppose you were asked to devise an algorithm which takes as input the preference lists and outputs a one-to-one pairing of students and advisers in which there are no student-adviser pairs $(s0, a0)$ and $(s1, a1)$ such that $s0$ prefers $a1$ to $a0$ and $a1$ prefers $s0$ to $s1$.

Here is an algorithm for this problem in the spirit of incremental improvement. Each student who does not have an adviser "proposes" to the most-preferred professor to whom he has not yet proposed. Each professor then considers all the students who have proposed to him and says to the student in this set he most prefers "I accept you"; he says "no" to the rest. The professor is then provisionally matched to a student. In each subsequent round, each student who does not have an adviser proposes to the professor to whom he has not yet proposed who is highest on his preference list. He does this regardless of whether the professor has already been matched with a student. The professor once again replies with a single accept, rejecting the rest. In particular, he may leave a student with whom he is currently paired. That this algorithm is correct is nontrivial—details are presented in Solution 18.17 on Page 445.

Many other algorithms are in this spirit: the standard algorithms for bipartite matching (Solution 18.18 on Page 447), maximum flow (Solution 18.20 on Page 448), and computing all pairs of shortest paths in a graph (Solutions 13.12 on Page 379 and 13.11 on Page 377) use incremental improvement. Other famous examples include the simplex algorithm for linear programming, and Euler's algorithm for computing a path in a graph which covers each edge once.

Sometimes it is easier to start with an infeasible solution that has a lower cost than the optimum solution, and incrementally update it to get to a feasible solution that is optimum. The standard algorithms for computing a minimum spanning tree (Solution 14.6 on Page 385) and shortest paths in a graph from a designated vertex (Solution 13.9 on Page 375) proceed in this fashion.

It is noteworthy that naïvely applying incremental improvement does not always work. For the professor-student pairing example above, if we begin with an arbitrary pairing of professors and students, and search for pairs p and s such that p prefers s to his current student, and s prefers p to his current professor and reassign such pairs, the procedure will not always converge.

Incremental improvement is often useful when designing heuristics, i.e., algorithms which are usually faster and/or simpler to implement than algorithms which compute an optimum result, but may return a suboptimal result. The algorithm we present for computing a tour for a travelling salesman (Solution 14.6 on Page 385) is in this spirit.

Elimination

One common approach to getting an efficient algorithm is to use elimination—that is to identify and rule out potential solutions that are suboptimal or dominated by other

solutions. Binary search, which is the subject of a number of problems in Chapter 8, uses elimination. Solution 8.9 on Page 265, where we use elimination to compute the square root of a real number, is especially instructive. Below we consider a fairly sophisticated application of elimination.

Suppose you have to build a distributed storage system. A large number, n, of users will share data on your system. There are m servers, numbered from 0 to $m-1$. One way to distribute users across servers is to assign the user with login id l to the server $h(l) \bmod m$, where $h()$ is a hash function. If the hash function does a good job, this approach distributes users uniformly across servers. However, if certain users require much more storage than others, some servers may be overloaded while others idle.

Let b_i be the number of bytes of storage required by user i. We will use values $k_0 < k_1 < \cdots < k_{m-2}$ to partition users across the m servers—a user with hash code c gets assigned to the server with the lowest id i such that $c \leq k_i$, or to server $m-1$ if no such i exists. We would like to select $k_0, k_1, \ldots, k_{m-2}$ to minimize the maximum number of bytes stored at any server.

The optimum values for $k_0, k_1, \ldots, k_{m-2}$ can be computed via DP—the essence of the program is to add one server at a time. The straightforward formulation has an $O(nm^2)$ time complexity.

However, there is a much faster approach based on elimination. The search for values $k_0, k_1, \ldots, k_{m-2}$ such that no server stores more than b bytes can be performed in $O(n)$ time by greedily selecting values for the k_is. We can then perform binary search on b to get the minimum b and the corresponding values for $k_0, k_1, \ldots, k_{m-2}$. The resulting time complexity is $O(n \log W)$, where $W = \sum_{i=0}^{m-1} b_i$

For the case of 10000 users and 100 servers, the DP algorithm took over an hour; the approach using binary search for b with greedy assignment took 0.1 seconds. Details are given in Solution 12.24 on Page 355.

The takeaway is that there may be qualitatively different ways to search for a solution, and that it is important to look for ways in which to eliminate candidates. The efficient frontier concept, described on Page 17, has some commonalities with elimination.

Parallelism

In the context of interview questions, parallelism is useful when dealing with scale, i.e., when the problem is so large that it would be impossible to solve it on a single machine or would take an unacceptable long time. The key insight you need to display is that you know how to decompose the problem so that

1. each subproblem can be solved relatively independently, and
2. the solution to the original problem can be efficiently constructed from solutions to the subproblems.

Efficiency is typically measured in terms of central processing unit (CPU) time, random access memory (RAM), network bandwidth, number of memeory and database accesses, etc.

Consider the problem of sorting a petascale integer array. If we know the distribution of the numbers, the best approach would be to define equal-sized ranges of integers and send one range to one machine for sorting. The sorted numbers would just need to be concatenated in the correct order. If the distribution is not known then we can send equal-sized arbitrary subsets to each machine and then merge the sorted results, e.g., using a min-heap. Details are given in Solution 10.4 on Page 290.

Caching

Caching is a great tool whenever there is a possibility of repeating computations. For example, the central idea behind dynamic programming is caching results from intermediate computations. Caching is also extremely useful when implementing a service that is expected to respond to many requests over time, and there are many repeated requests. Workloads on web services exhibit this property. Solution 15.1 on Page 394 sketches the design of a servlet which implements an online spell correction service; one of the key issues is performing cache updates in the presence of concurrent requests.

Randomization

Suppose you were asked to write a routine that takes an array A of n elements and an integer k between 1 and n, and returns the k-th largest element in A.

This problem can be solved by first sorting the array, and returning the element at index k in the sorted array. The time complexity of this approach is $O(n \log n)$. However, sorting performs far more work than is needed. A better approach is to eliminate parts of the array. For example, we could use the median to determine the $n/2$ largest elements of A; if $n/2 \geq k$, the desired element is in this set, otherwise we search for the $(n/2 - k)$-th largest element in the $n/2$ smallest elements.

It is possible, though nontrivial, to compute the median in time $O(n)$ without using randmization. However, an approach that works well is to select an index r at random and reorder the array so that elements greater than or equal to $A[r]$ appear first, followed by $A[r]$, followed by elements less than or equal to $A[r]$. If $A[r]$ is the k-th element in A after reordering, it is the desired element. If there are more than or equal to k elements before $A[r]$, we can focus our attention on that set, otherwise we search the elements after $A[r]$. This approach is fastest if $A[r]$ is close to the true median. A formal analysis shows that the probability of random index selection repeatedly resulting in unbalanced splits exponentially drops off with n. Details are given in Solution 8.13 on Page 268.

Randomization can also be used to create "signatures" to reduce the complexity of search, analogous to hashing. For example, consider the problem of determining whether an $m \times m$ array S of integers is a subarray of an $n \times n$ array T. Formally, we say S is a subarray of T iff there are p, q such that $S[i][j] = T[p + i][q + j]$, for all $0 \leq i, j \leq n - 1$. The brute-force approach to checking whether S is a subarray of T has complexity $O(n^2m^2)$—$O(n^2)$ individual checks, each of complexity $O(m^2)$. We

can improve the complexity to $O(n^2m)$ by computing a hash code for S and then computing the hash codes for $m \times m$ subarrays of T. The latter hash codes can be computed incrementally in $O(m)$ time if the hash function is chosen appropriately. For example, if the hash code is simply the XOR of all the elements of the subarray, the hash code for a subarray shifted over by one column can be computed by XORing the new elements and the removed elements with the previous hash code. A similar approach works for more complex hash functions, specifically for those that are in the form of a polynomial.

Approximation

In the real-world, it is routine to be given a problem that is difficult to solve exactly, either because of its intrinsic complexity, or the complexity of the code required. Developers need to recognize such problems, and be ready to discuss alternatives with the author of the problem. In practice, a solution that is "close" to the optimum solution is usually perfectly acceptable.

For example, let $\{A_0, A_1, \ldots, A_{n-1}\}$ be a set of n cities. Suppose we need to choose a subset of A to locate warehouses. Specifically, we want to choose k cities in such a way that cities are close to the warehouses. Let's say we define the cost of a warehouse assignment to be the maximum distance of any city to a warehouse.

The problem of finding a warehouse assignment that has the minimum cost is known to be NP-complete. However, consider the following algorithm for computing k cities. We pick the first warehouse to be the city for which the cost is minimized—this takes $\Theta(n^2)$ time since we try each city one at a time and check its distance to every other city. Now let's say we have selected the first $i-1$ warehouses $\{c_1, c_2, \ldots, c_{i-1}\}$ and are trying to choose the i-th warehouse. A reasonable choice for c_i is the city that is the farthest from the $i-1$ warehouses already chosen. This city can be computed in $O(ni)$ time. This greedy algorithm yields a solution whose cost is no more than $2\times$ that of the optimum solution; some heuristic tweaks can be used to further improve the quality. Details are given in Solution 14.7 on Page 385.

As another example of approximation, consider the problem of determining the k most frequent elements of a very large array. The direct approach of maintaining counts for each element may not be feasible because of space constraints. A natural approach is to *sample* the set to determine a set of candidates, exact counts for which are then determined in a second pass. The size of the candidate set depends on the distribution of the elements.

State

Formally, the state of a system is information that is sufficient to determine how that system evolves as a function of future inputs. Identifying the right notion of state can be critical to coming up with an algorithm that is time and space efficient, as well as easy to implement and prove correct.

Table 1.3: Abstract analysis techniques.

Analysis principle	Key points
Case analysis	Split the input/execution into a number of cases and solve each case in isolation.
Small examples	Find a solution to small concrete instances of the problem and then build a solution that can be generalized to arbitrary instances.
Iterative refinement	Most problems can be solved using a brute-force approach. Find such a solution and improve upon it.
Reduction	Use a well known solution to some other problem as a subroutine.
Graph modeling	Describe the problem using a graph and solve it using an existing algorithm.
Write an equation	Express relationships in the problem in the form of equations (or inequalities).
Variation	Solve a slightly different (possibly more general) problem and map its solution to the given problem.
Invariants	Find a function of the state of the given system that remains constant in the presence of (possibly restricted) updates to the state. Use this function to design an algorithm, prove correctness, or show an impossibility result.

There may be multiple ways in which state can be defined, all of which lead to correct algorithms. For example, when computing the max-difference (Problem 3.3 on Page 40), we could use the values of the elements at all prior indices as the state when we iterate through the array. Of course, this is inefficient, since all we really need is the minimum value.

One solution to computing the Levenshtein distance between two strings (Problem 12.11 on Page 105) entails creating a 2D array, whose dimensions are $(m+1)(n+1)$, where m and n are the lengths of the strings being compared. For large strings, this size may be unacceptably large. The algorithm iteratively fills rows of the array. It reads values from the current row and the previous row. This observation can be used to reduce the memory needed to two rows. A more careful implementation can reduce the memory required to just one row.

More generally, the efficient frontier concept on Page 17 demonstrates how an algorithm can be made to run faster and with less memory if state is chosen carefully. Other examples illustrating the benefits of careful state selection include string matching (Problem 3.9 on Page 41) and lazy initialization (Problem 3.2 on Page 39).

Abstract analysis patterns

Case analysis

In case analysis a problem is divided into a number of separate cases, and analyzing each such case individually suffices to resolve the initial problem. Cases do not have to be mutually exclusive; however, they must be exhaustive, that is cover all possibilities. For example, to prove that for all n, $n^3 \bmod 3$ is $0, 1$, or 8, we can consider the cases $n = 3m$, $n = 3m + 1$, and $n = 3m + 2$. These cases are individually easy to prove, and are exhaustive. Case analysis is commonly used in mathematics and games of strategy. Here we consider an application of case analysis to algorithm design.

Suppose you are given a set S of 25 distinct integers and a CPU that has a special instruction, `SORT5`, that can sort five integers in one cycle. Your task is to identify the largest, second-largest, and third-largest integers in S using `SORT5` to compare and sort subsets of S; furthermore, you must minimize the number of calls to `SORT5`.

If all we had to compute was the largest integer in the set, the optimum approach would be to form five disjoint subsets $S_1, \ldots, S_5$ of S, sort each subset, and then sort $\{\max S_1, \ldots, \max S_5\}$. This takes six calls to `SORT5` but leaves ambiguity about the second and third largest integers.

It may seem like many additional calls to `SORT5` are still needed. However if you do a careful case analysis and eliminate all $x \in S$ for which there are at least three integers in S larger than x, only five integers remain and hence just one more call to `SORT5` is needed to compute the result. Details are given in Solution 18.2 on Page 437.

Small examples

Problems that seem difficult to solve in the abstract can become much more tractable when you examine small concrete instances. For instance, consider the following problem. There are 500 closed doors along a corridor, numbered from 1 to 500. A person walks through the corridor and opens each door. Another person walks through the corridor and closes every alternate door. Continuing in this manner, the i-th person comes and toggles the state (open or closed) of every i-th door starting from Door i. You are to determine exactly how many doors are open after the 500-th person has walked through the corridor.

It is very difficult to solve this problem using an abstract approach, e.g., introducing Boolean variables for the state of each door and a state update function. However if you try the same problem with $1, 2, 3, 4, 10$, and 20 doors, it takes a short time to see that the doors that remain open are $1, 4, 9, 16 \ldots$, regardless of the total number of doors. The 10 doors case is illustrated in Figure 1.3 on the following page. Now the pattern is obvious—the doors that remain open are those corresponding to the perfect squares. Once you make this connection, it is easy to prove it for the general case. Hence the total number of open doors is $\lfloor \sqrt{500} \rfloor = 22$. Solution 18.1 on Page 436 develops this analysis in more detail.

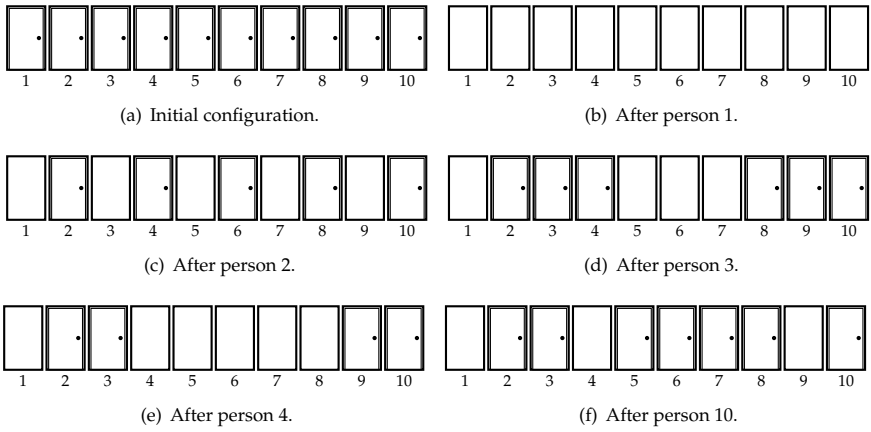


Figure 1.3: Progressive updates to 10 doors.

Optimally selecting a red card (Problem 17.18 on Page 146) and avoiding losing at the alternating coin pickup game (Problem 18.19 on Page 154) are other problems that benefit from use of the "small example" principle.

Iterative refinement of a brute-force solution

Many problems can be solved optimally by a simple algorithm that has a high time/space complexity—this is sometimes referred to as a brute-force solution. (Other terms are *enumerative search* and *generate-and-test*.) Often, this algorithm can be refined to one that is faster. At the very least, it may offer hints into the nature of the problem.

As an example, suppose you were asked to write a function that takes an array A of n numbers, and rearranges A's elements to get a new array B having the property that $B[0] \leq B[1] \geq B[2] \leq B[3] \geq B[4] \leq B[5] \geq \cdots$.

One straightforward solution is to sort A and interleave the bottom and top halves of the sorted array. Alternately, we could sort A and then swap the elements at the pairs $(A[1], A[2]), (A[3], A[4]), \ldots$ Both these approaches have the same time complexity as sorting, namely $O(n \log n)$.

You will soon realize that it is not necessary to sort A to achieve the desired configuration—you could simply rearrange the elements around the median, and then perform the interleaving. Median finding can be performed in time $O(n)$ (Solution 8.13 on Page 268), which is the overall time complexity of this approach.

Finally, you might notice that the desired ordering is very local, and realize that it is not necessary to find the median. Simply iterating through the array and setting $(A[i], A[i+1])$ to $(\min(A[i], A[i+1]), \max(A[i], A[i+1]))$ for even i and to $(\max(A[i], A[i+1]), \min(A[i], A[i+1]))$ for odd i achives the desired configuration. In code:

```
template <typename T>
void rearrange(vector<T> &A) {
  for (int i = 0; i < A.size() - 1; ++i) {
    T pair_min = min(A[i], A[i + 1]), pair_max = max(A[i], A[i + 1]);
    if (i & 1) { // odd one
      A[i] = pair_max, A[i + 1] = pair_min;
    } else {
      A[i] = pair_min, A[i + 1] = pair_max;
    }
  }
}
```

This approach has time complexity $O(n)$, which is the same as the approach based on median finding. However, it is much easier to implement, and operates in an online fashion, i.e., it never needs to store more than two elements in memory or read a previous element.

As another example of iterative refinement, consider the problem of string search (Problem 3.9 on Page 41): given two strings s (search string) and t (text), find all occurrences of s in t. Since s can occur at any offset in t, the brute-force solution is to test for a match at every offset. This algorithm is perfectly correct; its time complexity is $O(nm)$, where n and m are the lengths of s and t.

After trying some examples, you may see that there are several ways in which to improve the time complexity of the brute-force algorithm. For example, if the character $t[i]$ is not present in s you can advance the matching by n characters. Furthermore, this skipping works better if we match the search string from its end and work backwards. These refinements will make the algorithm very fast (linear time) on random text and search strings; however, the worst case complexity remains $O(nm)$.

You can make the additional observation that a partial match of s which does not result in a full match implies other offsets which cannot lead to full matches. For example, if s = *abdabcabc* and if, starting backwards, we have a partial match up to *abcabc* that does not result in a full match, we know that the next possible matching offset has to be at least three positions ahead (where we can match the second *abc* from the partial match).

By putting together these refinements you will have arrived at the famous Boyer-Moore string search algorithm—its worst-case time complexity is $O(n + m)$ (which is the best possible from a theoretical perspective); it is also one of the fastest string search algorithms in practice.

Many other sophisticated algorithms can be developed in this fashion. As another example, the brute-force solution to computing the maximum subarray sum for an integer array of length n is to compute the sum of all subarrays, which has $O(n^3)$ time complexity. This can be improved to $O(n^2)$ by precomputing the sums of all the prefixes of the given arrays; this allows the sum of a subarray to be computed in constant time. The natural divide and conquer algorithm has an $O(n \log n)$ time complexity. Finally, one can observe that a maximum subarray must end at one of

n indices, and the maximum subarray sum for a subarray ending at index i can be computed from previous maximum subarray sums, which leads to an $O(n)$ algorithm. Details are presented on Page 102.

Reduction

Consider the problem of finding if one string is a rotation of the other, e.g., "car" and "arc" are rotations of each other. A natural approach may be to rotate the first string by every possible offset and then compare it with the second string. This algorithm would have quadratic time complexity.

You may notice that this problem is quite similar to string search which can be done in linear time, albeit using a somewhat complex algorithm. Therefore it is natural to try to reduce this problem to string search. Indeed, if we concatenate the second string with itself and search for the first string in the resulting string, we will find a match iff the two original strings are rotations of each other. This reduction yields a linear time algorithm for our problem.

The reduction principle is also illustrated in the problem of checking whether a road network is resilient in the presence of blockages (Problem 13.4 on Page 118) and the problem of finding the minimum number of pictures needed to photograph a set of teams (Problem 13.7 on Page 121).

Usually you try to reduce the given problem to an easier problem. Sometimes, however, you need to reduce a problem known to be difficult to the given problem. This shows the given problem is difficult, which justifies heuristics and approximate solutions. Such scenarios are described in more detail in Chapter 14.

Graph modeling

Drawing pictures is a great way to brainstorm for a potential solution. If the relationships in a given problem can be represented using a graph, quite often the problem can be reduced to a well-known graph problem. For example, suppose you are given a set of exchange rates between currencies and you want to determine if an arbitrage exists, i.e., there is a way by which you can start with one unit of some currency C and perform a series of barters which results in having more than one unit of C.

Table 1.4 on the next page shows a representative example. An arbitrage is possible for this set of exchange rates: 1 USD $\rightarrow 1 \times 0.8123 = 0.8123$ EUR $\rightarrow 0.8123 \times 1.2010 = 0.9755723$ CHF $\rightarrow 0.9755723 \times 80.39 = 78.426257197$ JPY $\rightarrow 78.426257197 \times 0.0128 = 1.00385609212$ USD.

We can model the problem with a graph where currencies correspond to vertices, exchanges correspond to edges, and the edge weight is set to the logarithm of the exchange rate. If we can find a cycle in the graph with a positive weight, we would have found such a series of exchanges. Such a cycle can be solved using the Bellman-Ford algorithm, as described in Solution 13.12 on Page 379.

Table 1.4: Exchange rates for seven major currencies.

Symbol	USD	EUR	GBP	JPY	CHF	CAD	AUD
USD	1	0.8148	0.6404	78.125	0.9784	0.9924	0.9465
EUR	1.2275	1	0.7860	96.55	1.2010	1.2182	1.1616
GBP	1.5617	1.2724	1	122.83	1.5280	1.5498	1.4778
JPY	0.0128	0.0104	0.0081	1	1.2442	0.0126	0.0120
CHF	1.0219	0.8327	0.6546	80.39	1	1.0142	0.9672
CAD	1.0076	0.8206	0.6453	79.26	0.9859	1	0.9535
AUD	1.0567	0.8609	0.6767	83.12	1.0339	1.0487	1

Write an equation

Some problems can be solved by expressing them in the language of mathematics. For example, suppose you were asked to write an algorithm that computed binomial coefficients, $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

The problem with computing the binomial coefficient directly from the definition is that the factorial function grows very quickly and can overflow an integer variable. If we use floating point representations for numbers, we lose precision and the problem of overflow does not go away. These problems potentially exist even if the final value of $\binom{n}{k}$ is small. One can try to factor the numerator and denominator and try and cancel out common terms but factorization is itself a hard problem.

The binomial coefficients satisfy the *addition formula*:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

This identity leads to a straightforward recursion for computing $\binom{n}{k}$ which avoids the problems described above. DP has to be used to achieve good time complexity—details are in Solution 12.14 on Page 344.

Variation

The idea of the variation pattern is to solve a slightly different (possibly more general) problem and map its solution to your problem.

Suppose we were asked to design an algorithm which takes as input an undirected graph and produces as output a black or white coloring of the vertices such that for every vertex, at least half of its neighbors differ in color from it.

We could try to solve this problem by assigning arbitrary colors to vertices and then flipping colors wherever constraints are not met. However this approach may lead to the number of vertices that do not satisfy the constraint increasing.

It turns out we can define a slightly different problem whose solution will yield the desired coloring. Define an edge to be *diverse* if its ends have different colors. It is easy to verify that a coloring that maximizes the number of diverse edges also satisfies the constraint of the original problem, so there always exists a coloring satisfying the constraint.

It is not necessary to find a coloring that maximizes the number of diverse edges. All that is needed is a coloring in which the set of diverse edges is maximal with respect to single vertex flips. Such a coloring can be computed efficiently; details are given in Problem 12.29 on Page 114.

Invariants

The following problem was popular at interviews in the early 1990s. You are given an 8×8 square with two unit sized squares at the opposite ends of a diagonal removed, leaving 62 squares, as illustrated in Figure 1.4. You are given 31 rectangular dominoes. Each can cover exactly two squares. How would you cover all the 62 squares with the dominoes?

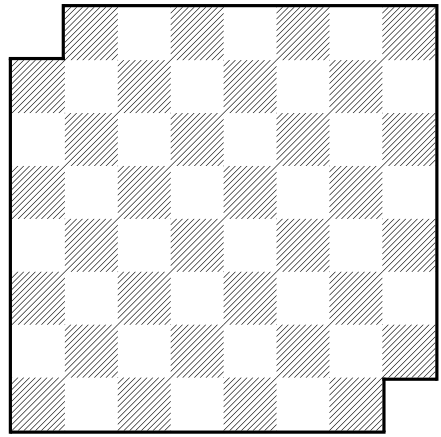


Figure 1.4: A chessboard, with two diagonally opposite corners removed.

It is easy to spend hours trying unsuccessfully to find such a covering. This experience will teach you that a problem may be intentionally worded to mislead you into following a futile path.

There is a simple argument that no covering exists. Think of the 8×8 square as a chessboard. Then the two removed squares will always have the same color, so there will be either 30 black and 32 white squares to be covered or 32 black and 30 white squares to be covered. Each domino will cover one black and one white square, so the number of black and white squares covered as you successively put down the dominoes is equal. Hence it is impossible to cover the given chessboard.

This proof of impossibility is an example of invariant analysis. An invariant is a function of the state of a system being analyzed that remains constant in the presence of (possibly restricted) updates to the state. Invariant analysis is particularly powerful at proving impossibility results as we just saw with the chessboard tiling problem. The challenge is finding a simple invariant.

The argument above also used the "auxiliary elements" pattern, in which we added a new element to our problem to get closer to a solution. The original problem did not talk about the colors of individual squares; adding these colors made it easy to prove impossibility.

It is possible to prove impossibility without appealing to square colors. Specifically, orient the board with the missing pieces on the lower right and upper left. There is an impossibility proof that uses a case-analysis for each column on the height of the highest domino that is parallel to the base. However, the proof given above is much simpler.

Invariant analysis can be used to design algorithms, as well as prove impossibility results. In the coin selection problem, there are fifty coins in a line, as in Figure 12.8 on Page 108. Two players, F and S, take turns at choosing one coin each—they can only choose from the two coins at the ends of the line. Player F goes first. The game ends when all the coins have been picked up. The player whose coins have the higher total value wins.

The optimum strategy for F can be computed using Dynamic Programming (Problem 12.19 on Page 108). However, if F's goal is simply to ensure he does not do worse than S, he can achieve this goal with much less computation. Specifically, he can number the coins from 1 to 50 from left-to-right, and compute the sum of the even-index coins and the sum of the odd-index coins. Suppose the odd-index sum is larger. Then F can force S to always select an even-index coin by selecting the odd-index coins when it is his own turn, ensuring that S cannot win. The same principle holds when the even-index sum is larger, or the sums are equal.

Invariant analysis can be used with symmetry to solve very difficult problems, sometimes in less than intuitive ways. For example, in the game of chomp, Player F and Player S alternately take bites from a chocolate bar. The chocolate bar is an $n \times n$ rectangle; a bite must remove a square and all squares above and to the right in the chocolate bar. The first player to eat the lower leftmost square, which is poisoned, loses. Player F can force a win by first selecting the square immediately above and to the right of the poisoned square, leaving the bar shaped like an L, with equal vertical and horizontal sides. Now whatever move S makes, F can play a symmetric move about the line bisecting the chocolate bar through the poisoned square to recreate the L shape (this is the invariant), which forces S to be the first to consume the poisoned square. Details are given in Solution 18.6 on Page 438.

Algorithm design using invariants is also illustrated in Solution 9.7 on Page 278 (can the characters in a string be permuted to form a palindrome?) and in Solution 10.14 on Page 299 (are there three elements in an array that sum to a given number?).

Complexity Analysis

The run time of an algorithm depends on the size of its input. One common approach to capture the run time dependency is by expressing asymptotic bounds

on the worst-case run time as a function of the input size. Specifically, the run time of an algorithm on an input of size n is $O(f(n))$ if, for sufficiently large n, the run time is not more than $f(n)$ times a constant. The big-O notation simply indicates an upper bound; if the run time is asymptotically proportional to $f(n)$, the complexity is written as $\Theta(f(n))$. (Note that the big-O notation is widely used where Θ is more appropriate.) The notation $\Omega(f(n))$ is used to denote an asymptotic lower bound of $f(n)$ on the time complexity of an algorithm

As an example, searching an unsorted array of integers of length n, for a given integer, has an asymptotic complexity of $\Theta(n)$ since in the worst-case, the given integer may not be present. Similarly, consider the naïve algorithm for testing primality which tries all numbers from 2 to the square root of the input number n. What is its complexity? In the best case, n is divisible by 2. However in the worst-case, the input may be a prime, so the algorithm performs $\sqrt{n}$ iterations. Furthermore, since the number n requires $\lg n$ bits to encode, this algorithm's complexity is actually exponential in the size of the input.

Generally speaking, if an algorithm has a run time that is a polynomial, i.e., $O(n^k)$ for some fixed k, where n is the size of the input, it is considered to be efficient; otherwise, it is inefficient. Notable exceptions exist—for example, the simplex algorithm for linear programming is not polynomial but works very well in practice; the AKS primality testing algorithm has polynomial runtime but the degree of the polynomial is too high for it to be competitive with randomized algorithms for primality testing.

Complexity theory is applied as a similar way when analyzing the space requirements of an algorithm. Usually, the space needed to read in an instance is not included; otherwise, every algorithm would have $\Omega(n)$ space complexity.

Several of our problems call for an algorithm that uses $O(1)$ space. Conceptually, the memory used by such an algorithm should not depend on the size of the input instance. Specifically, it should be possible to implement the algorithm without dynamic memory allocation (explicitly, or indirectly, e.g., through library routines). Furthermore, the maximum depth of the function call stack should also be a constant, independent of the input. The standard algorithm for depth-first search of a graph is an example of an algorithm that does not perform any dynamic allocation, but uses the function call stack for implicit storage—its space complexity is not $O(1)$.

A streaming algorithm is one in which the input is presented as a sequence of items and is examined in only a few passes (typically just one). These algorithms have limited memory available to them (much less than the input size) and also limited processing time per item. Algorithms for computing summary statistics on log file data often fall into this category.

As a rule, algorithms should be designed with the goal of reducing the worst-case complexity rather than average-case complexity for several reasons:

1. It is very difficult to define meaningful distributions on the inputs.
2. Pathological inputs are more likely than statistical models may predict. For example, a worst-case input for a naïve implementation of quicksort is one where all entries are the same, which is not unlikely in a practical setting.

3. Malicious users may exploit bad worst-case performance to create denial-of-service attacks.

Conclusion

In addition to developing intuition for which patterns may apply to a given problem, it is also important to know when your approach is not working. In an interview setting, even if you do not end up solving the problem entirely, you will get credit for approaching problems in a systematic way and clearly communicating your approach to the problem. We cover nontechnical aspects of problem solving in Chapter 20.