Elements of Programming Interviews

The Insiders' Guide

Adnan Aziz
Tsung-Hsien Lee
Amit Prakash

This document is a sampling of our book, Elements of Programming Interviews (EPI). Its purpose is to provide examples of EPI's organization, content, style, topics, and quality. The sampler focuses solely on problems; in particular, it does not include three chapters on the nontechnical aspects of interviewing. We'd love to hear from you—we're especially interested in your suggestions as to where the exposition can be improved, as well as any insights into interviewing trends you may have.

You can buy EPI with at Amazon.com.

http://ElementsOfProgrammingInterviews.com

Adnan Aziz is a professor at the Department of Electrical and Computer Engineering at The University of Texas at Austin, where he conducts research and teaches classes in applied algorithms. He received his Ph.D. from The University of California at Berkeley; his undergraduate degree is from Indian Institutes of Technology Kanpur. He has worked at Google, Qualcomm, IBM, and several software startups. When not designing algorithms, he plays with his children, Laila, Imran, and Omar.

Tsung-Hsien Lee is a Software Engineer at Google. Previously, he worked as a Software Engineer Intern at Facebook. He received both his M.S. and undergraduate degrees from National Tsing Hua University. He has a passion for designing and implementing algorithms. He likes to apply algorithms to every aspect of his life. He takes special pride in helping to organize Google Code Jam 2014.

Amit Prakash is a co-founder and CTO of ThoughtSpot, a Silicon Valley startup. Previously, he was a Member of the Technical Staff at Google, where he worked primarily on machine learning problems that arise in the context of online advertising. Before that he worked at Microsoft in the web search team. He received his Ph.D. from The University of Texas at Austin; his undergraduate degree is from Indian Institutes of Technology Kanpur. When he is not improving business intelligence, he indulges in his passion for puzzles, movies, travel, and adventures with Nidhi and Aanya.

Elements of Programming Interviews: The Insiders' Guide
by Adnan Aziz, Tsung-Hsien Lee, and Amit Prakash

We typeset this book using LaTeX and the Memoir class. We used TikZ to draw figures. Allan Ytac created the cover, based on a design brief we provided.

The companion website for the book includes contact information and a list of known errors for each version of the book. If you come across an error or an improvement, please let us know.

Version 1.4.10

Website: http://ElementsOfProgrammingInterviews.com

Table of Contents

I Problems 1

1 Primitive Types 2
- 1.1 Compute parity . . . 2
- 1.2 Convert base . . . 2

2 Arrays 4
- 2.1 Compute the max difference . . . 4

3 Strings 5
- 3.1 Interconvert strings and integers . . . 5
- 3.2 Reverse all the words in a sentence . . . 5

4 Linked Lists 6
- 4.1 Test for cyclicity . . . 7

5 Stacks and Queues 8
- 5.1 Implement a stack with max API . . . 8
- 5.2 Print a binary tree in order of increasing depth . . . 9

6 Binary Trees 10
- 6.1 Test if a binary tree is balanced . . . 12

7 Heaps 13
- 7.1 Merge sorted files . . . 13

8 Searching 15
- 8.1 Search a sorted array for first occurrence of k . . . 17

9 Hash Tables 18
- 9.1 Test if an anonymous letter is constructible . . . 19

10 Sorting 20

10.1 Compute the intersection of two sorted arrays 21
10.2 Render a calendar 21

11 Binary Search Trees **23**
11.1 Test if a binary tree satisfies the BST property 23

12 Recursion **25**
12.1 Enumerate the power set 25

13 Dynamic Programming **27**
13.1 Count the number of ways to traverse a 2D array 29

14 Greedy Algorithms and Invariants **30**
14.1 The 3-sum problem 30

15 Graphs **31**
15.1 Paint a Boolean matrix 34

16 Parallel Computing **35**
16.1 Implement a Timer class 36

17 Design Problems **37**
17.1 Design a system for detecting copyright infringement 39

II Hints **40**

III Solutions **42**

IV Notation and Index **71**

Index of Terms **74**

Part I

Problems

Chapter

1 Primitive Types

Representation is the essence of programming.

— "*The Mythical Man Month*,"
F. P. Brooks, 1975

A program updates variables in memory according to its instructions. The variables are classified according to their type—a type is a classification of data that spells out possible values for that type and the operations that can be performed on it.

Types can be primitive, i.e., provided by the language, or defined by the programmer. The set of primitive types depends on the language. For example, the primitive types in C++ are `bool`, `char`, `short`, `int`, `long`, `float`, and `double`; integer types and `char`s have unsigned versions too. The primitive types in Java are `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. A programmer can define a complex number type as a pair of doubles, one for the real and one for the imaginary part. The width of a primitive-type variable is the number of bits of storage it takes in memory. For example, most implementations of C++ use 32 or 64 bits for an `int`. In Java an `int` is always 32 bits.

Problems involving manipulation of bit-level data are often asked in interviews. It is easy to introduce errors in code that manipulates bit-level data—when you play with bits, expect to get bitten.

1.1 Compute parity

The parity of a binary word is 1 if the number of 1s in the word is odd; otherwise, it is 0. For example, the parity of 1011 is 1, and the parity of 10001000 is 0. Parity checks are used to detect single bit errors in data storage and communication. It is fairly straightforward to write code that computes the parity of a single 64-bit word.

Problem 1.1: How would you compute the parity of a very large number of 64-bit words? *pg. 43*

1.2 Convert base

In the decimal system, the position of a digit is used to signify the power of 10 that digit is to be multiplied with. For example, "314" denotes the number $3 \times 100 + 1 \times 10 + 4 \times 1$. (Note that zero, which is not needed in other systems, is essential in the decimal system, since a zero can be used to skip a power.) The base b number

system generalizes the decimal number system: the string "$a_{k-1}a_{k-2}\ldots a_1a_0$", where $0 \leq a_i < b$, denotes in base-b the integer $a_0 \times b^0 + a_1 \times b^1 + a_2 \times b^2 + \cdots + a_{k-1} \times b^{k-1}$.

Problem 1.2: Write a function that performs base conversion. Specifically, the input is an integer base b_1, a string s, representing an integer x in base b_1, and another integer base b_2; the output is the string representing the integer x in base b_2. Assume $2 \leq b_1, b_2 \leq 16$. Use "A" to represent 10, "B" for 11, ..., and "F" for 15. ***pg. 45***

Chapter

2 Arrays

The machine can alter the scanned symbol and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine.

— "*Intelligent Machinery*,"
A. M. Turing, 1948

The simplest data structure is the *array*, which is a contiguous block of memory. Given an array A which holds n objects, $A[i]$ denotes the $(i+1)$-th object stored in the array. Retrieving and updating $A[i]$ takes $O(1)$ time. However, the size of the array is fixed, which makes adding more than n objects impossible. Deletion of the object at location i can be handled by having an auxiliary Boolean associated with the location i indicating whether the entry is valid.

Insertion of an object into a full array can be handled by allocating a new array with additional memory and copying over the entries from the original array. This makes the worst-case time of insertion high but if the new array has, for example, twice the space of the original array, the average time for insertion is constant since the expense of copying the array is infrequent.

2.1 Compute the max difference

A robot needs to travel along a path that includes several ascents and descents. When it goes up, it uses its battery to power the motor and when it descends, it recovers the energy which is stored in the battery. The battery recharging process is ideal: on descending, every Joule of gravitational potential energy converts to a Joule of electrical energy which is stored in the battery. The battery has a limited capacity and once it reaches this capacity, the energy generated in descending is lost.

Problem 2.1: Design an algorithm that takes a sequence of n three-dimensional coordinates to be traversed, and returns the minimum battery capacity needed to complete the journey. The robot begins with the battery fully charged. ***pg. 45***

Chapter

3 Strings

String pattern matching is an important problem that occurs in many areas of science and information processing. In computing, it occurs naturally as part of data processing, text editing, term rewriting, lexical analysis, and information retrieval.

— "Algorithms For Finding Patterns in Strings,"
A. V. Aho, 1990

Strings are ubiquitous in programming today—scripting, web development, and bioinformatics all make extensive use of strings. You should know how strings are represented in memory, and understand basic operations on strings such as comparison, copying, joining, splitting, matching, etc. We now present problems on strings which can be solved using elementary techniques. Advanced string processing algorithms often use hash tables (Chapter 9) and dynamic programming (Page 27).

3.1 Interconvert strings and integers

A string is a sequence of characters. A string may encode an integer, e.g., "123" encodes 123. In this problem, you are to implement methods that take a string representing an integer and return the corresponding integer, and vice versa. Your code should handle negative integers. You cannot use library functions like `stoi` in C++ and `parseInt` in Java.

Problem 3.1: Implement string/integer inter-conversion functions. *pg. 46*

3.2 Reverse all the words in a sentence

Given a string containing a set of words separated by whitespace, we would like to transform it to a string in which the words appear in the reverse order. For example, "Alice likes Bob" transforms to "Bob likes Alice". We do not need to keep the original string.

Problem 3.2: Implement a function for reversing the words in a string s. *pg. 47*

Chapter

4 Linked Lists

The S-expressions are formed according to the following recursive rules.

1. *The atomic symbols p_1, p_2, etc., are S-expressions.*
2. *A null expression Λ is also admitted.*
3. *If e is an S-expression so is (e).*
4. *If e_1 and e_2 are S-expressions so is (e_1, e_2).*

— *"Recursive Functions Of Symbolic Expressions,"* J. McCarthy, 1959

A *singly linked list* is a data structure that contains a sequence of nodes such that each node contains an object and a reference to the next node in the list. The first node is referred to as the *head* and the last node is referred to as the *tail*; the tail's next field is a `null` reference. The structure of a singly linked list is given in Figure 4.1. There are many variants of linked lists, e.g., in a *doubly linked list*, each node has a link to its predecessor; similarly, a sentinel node or a self-loop can be used instead of `null`. The structure of a doubly linked list is given in Figure 4.2.

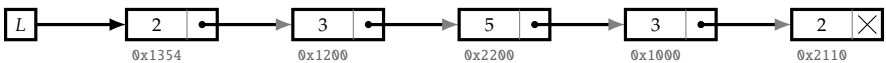


Figure 4.1: Example of a singly linked list. The number in hex below a node indicates the memory address of that node.

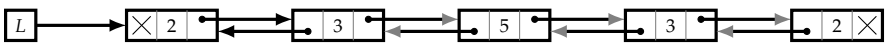


Figure 4.2: Example of a doubly linked list.

For all problems in this chapter, unless otherwise stated, L is a singly linked list, and your solution may not use more than a few words of storage, regardless of the length of the list. Specifically, each node has two entries—a `data` field, and a `next` field, which points to the next node in the list, with the next field of the last node being `null`. Its prototype is as follows:

```
1 template <typename T>
2 struct ListNode {
```

```
3   T data;
4   shared_ptr<ListNode<T>> next;
5 };
```

4.1 Test for cyclicity

Although a linked list is supposed to be a sequence of nodes ending in a `null`, it is possible to create a cycle in a linked list by making the `next` field of an element reference to one of the earlier nodes.

Problem 4.1: Given a reference to the head of a singly linked list, how would you determine whether the list ends in a `null` or reaches a cycle of nodes? Write a function that returns `null` if there does not exist a cycle, and the reference to the start of the cycle if a cycle is present. (You do not know the length of the list in advance.) *pg. 48*

Chapter

5 Stacks and Queues

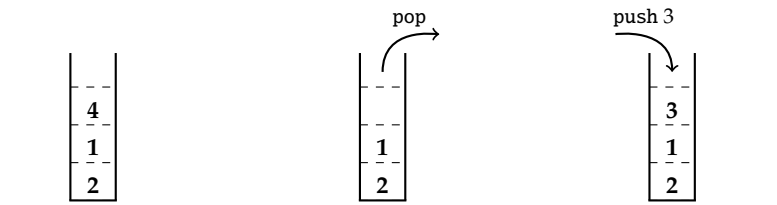
Linear lists in which insertions, deletions, and accesses to values occur almost always at the first or the last node are very frequently encountered, and we give them special names . . .

— "*The Art of Computer Programming, Volume* 1,"
D. E. Knuth, 1997

Stacks

A *stack* supports two basic operations—`push` and `pop`. Elements are added (pushed) and removed (popped) in last-in, first-out order, as shown in Figure 5.1. If the stack is empty, pop typically returns a `null` or throws an exception.

When the stack is implemented using a linked list these operations have $O(1)$ time complexity. If it is implemented using an array, there is maximum number of entries it can have—`push` and `pop` are still $O(1)$. If the array is dynamically resized, the amortized time for both `push` and `pop` is $O(1)$. A stack can support additional operations such as `peek` (return the top of the stack without popping it).



(a) Initial configuration. (b) Perform pop on (a). (c) Perform push 3 on (b).

Figure 5.1: Operations on a stack.

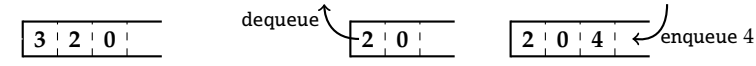
5.1 Implement a stack with max API

Problem 5.1: Design a stack that supports a `max` operation, which returns the maximum value stored in the stack. *pg. 50*

Queues

A *queue* supports two basic operations—`enqueue` and `dequeue`. (If the queue is empty, `dequeue` typically returns a `null` or throws an exception.) Elements are added (enqueued) and removed (dequeued) in first-in, first-out order.

A queue can be implemented using a linked list, in which case these operations have $O(1)$ time complexity. Other operations can be added, such as `head` (which returns the item at the start of the queue without removing it), and `tail` (which returns the item at the end of the queue without removing it).



(a) Initial configuration. (b) Queue (a) after dequeue. (c) Queue (b) after `enqueue(4)`.

Figure 5.2: Examples of enqueue and dequeue.

A *deque*, also sometimes called a double-ended queue, is a doubly linked list in which all insertions and deletions are from one of the two ends of the list, i.e., at the head or the tail. An insertion to the front is called a `push`, and an insertion to the back is called an `inject`. A deletion from the front is called a `pop`, and a deletion from the back is called an `eject`.

5.2 Print a binary tree in order of increasing depth

Binary trees are formally defined in Chapter 6. In particular, each node in a binary tree has a depth, which is its distance from the root.

Problem 5.2: Given the root of a binary tree, print all the keys at the root and its descendants. The keys should be printed in the order of the corresponding nodes' depths. For example, you could print

```
314
6 6
271 561 2 271
28 0 3 1 28
17 401 257
641
```

for the binary tree in Figure 6.1 on the next page. *pg. 52*

Chapter

6 Binary Trees

The method of solution involves the development of a theory of finite automata operating on infinite trees.

— "*Decidability of Second Order Theories and Automata on Trees*," M. O. Rabin, 1969

A *binary tree* is a data structure that is useful for representing hierarchy. Formally, a binary tree is either empty, or a *root* node r together with a left binary tree and a right binary tree. The subtrees themselves are binary trees. The left binary tree is sometimes referred to as the *left subtree* of the root, and the right binary tree is referred to as the *right subtree* of the root.

Figure 6.1 gives a graphical representation of a binary tree. Node A is the root. Nodes B and I are the left and right children of A.

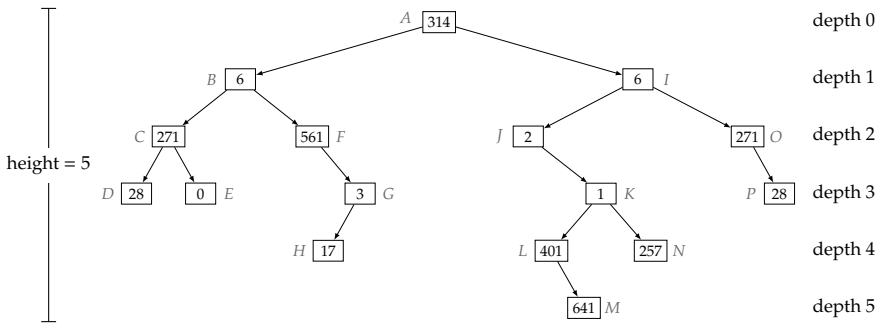


Figure 6.1: Example of a binary tree. The node depths range from 0 to 5. Node M has the highest depth (5) of any node in the tree, implying the height of the tree is 5.

Often the root stores additional data. Its prototype is listed as follows:

```
1 template <typename T>
2 struct BinaryTreeNode {
3   T data;
4   unique_ptr<BinaryTreeNode<T>> left, right;
5 };
```

Each node, except the root, is itself the root of a left subtree or a right subtree. If l is the root of p's left subtree, we will say l is the *left child* of p, and p is the *parent* of l;

the notion of *right child* is similar. If a node is a left or a right child of p, we say it is a *child* of p. Note that with the exception of the root, every node has a unique parent. Usually, but not universally, the node object definition includes a parent field (which is `null` for the root). Observe that for any node there exists a unique sequence of nodes from the root to that node with each node in the sequence being a child of the previous node. This sequence is sometimes referred to as the *search path* from the root to the node.

The parent-child relationship defines an ancestor-descendant relationship on nodes in a binary tree. Specifically, a node is an *ancestor* of d if it lies on the search path from the root to d. If a node is an ancestor of d, we say d is a *descendant* of that node. Our convention is that x is an ancestor and descendant of itself. A node that has no descendants except for itself is called a *leaf*.

The *depth* of a node n is the number of nodes on the search path from the root to n, not including n itself. The *height* of a binary tree is the maximum depth of any node in that tree. A *level* of a tree is all nodes at the same depth. See Figure 6.1 on the preceding page for an example of the depth and height concepts.

As concrete examples of these concepts, consider the binary tree in Figure 6.1 on the facing page. Node I is the parent of J and O. Node G is a descendant of B. The search path to L is $\langle A, I, J, K, L\rangle$. The depth of N is 4. Node M is the node of maximum depth, and hence the height of the tree is 5. The height of the subtree rooted at B is 3. The height of the subtree rooted at H is 0. Nodes $D, E, H, M, N,$ and P are the leaves of the tree.

A *full binary tree* is a binary tree in which every node other than the leaves has two children. A *perfect binary tree* is a full binary tree in which all leaves are at the same depth, and in which every parent has two children. A *complete binary tree* is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. (This terminology is not universal, e.g., some authors use complete binary tree where we write perfect binary tree.) It is straightforward to prove using induction that the number of nonleaf nodes in a full binary tree is one less than the number of leaves. A perfect binary tree of height h contains exactly $2^{h+1} - 1$ nodes, of which 2^h are leaves. A complete binary tree on n nodes has height $\lfloor \lg n \rfloor$.

A key computation on a binary tree is *traversing* all the nodes in the tree. (Traversing is also sometimes called *walking*.) Here are some ways in which this visit can be done.

- Traverse the left subtree, visit the root, then traverse the right subtree (an *inorder* traversal). An inorder traversal of the binary tree in Figure 6.1 on the preceding page visits the nodes in the following order: $\langle D, C, E, B, F, H, G, A, J, L, M, K, N, I, O, P\rangle$.
- Visit the root, traverse the left subtree, then traverse the right subtree (a *preorder* traversal). A preorder traversal of the binary tree in Figure 6.1 on the facing page visits the nodes in the following order: $\langle A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P\rangle$.

- Traverse the left subtree, traverse the right subtree, and then visit the root (a *postorder* traversal). A postorder traversal of the binary tree in Figure 6.1 on Page 10 visits the nodes in the following order: $\langle D, E, C, H, G, F, B, M, L, N, K, J, P, O, I, A \rangle$.

Let T be a binary tree on n nodes, with height h. Implemented recursively, these traversals have $O(n)$ time complexity and $O(h)$ additional space complexity. (The space complexity is dictated by the maximum depth of the function call stack.) If each node has a parent field, the traversals can be done with $O(1)$ additional space complexity.

The term tree is overloaded, which can lead to confusion; see Page 33 for an overview of the common variants.

6.1 Test if a binary tree is balanced

A binary tree is said to be balanced if for each node in the tree, the difference in the height of its left and right subtrees is at most one. A perfect binary tree is balanced, as is a complete binary tree. A balanced binary tree does not have to be perfect or complete—see Figure 6.2 for an example.

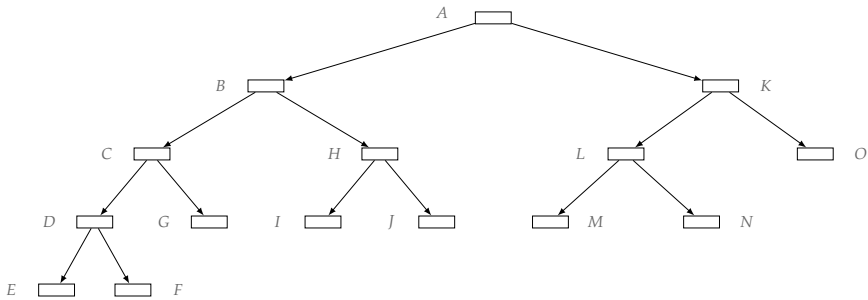


Figure 6.2: A balanced binary tree of height 4.

Problem 6.1: Write a function that takes as input the root of a binary tree and checks whether the tree is balanced. *pg. 54*

Chapter

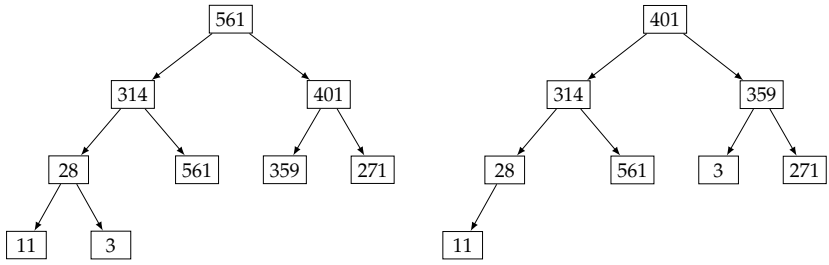
7 Heaps

Using F-heaps we are able to obtain improved running times for several network optimization algorithms.

— "*Fibonacci heaps and their uses*,"
M. L. Fredman and R. E. Tarjan, 1987

A *heap* is a specialized binary tree, specifically it is a complete binary tree. It supports $O(\log n)$ insertions, $O(1)$ time lookup for the max element, and $O(\log n)$ deletion of the max element. The extract-max operation is defined to delete and return the maximum element. (The *min-heap* is a completely symmetric version of the data structure and supports $O(1)$ time lookups for the minimum element.)

A max-heap can be implemented as an array; the children of the node at index i are at indices $2i+1$ and $2i+2$. Searching for arbitrary keys has $O(n)$ time complexity. Anything that can be done with a heap can be done with a balanced BST with the same or better time and space complexity but with possibly some implementation overhead. There is no relationship between the heap data structure and the portion of memory in a process by the same name.



(a) A max-heap where 561 is the max.

(b) After the deletion of max of the heap in (a).

Figure 7.1: An example of max-heap.

7.1 Merge sorted files

You are given 500 files, each containing stock trade information for an S&P 500 company. Each trade is encoded by a line as follows:

```
1232111,AAPL,30,456.12
```

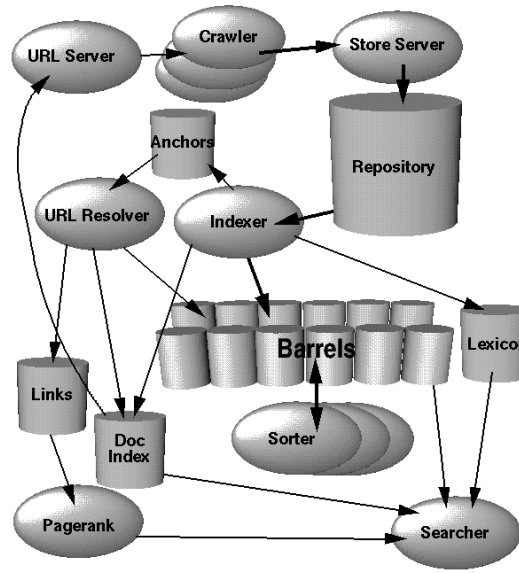
The first number is the time of the trade expressed as the number of milliseconds since the start of the day's trading. Lines within each file are sorted in increasing order of time. The remaining values are the stock symbol, number of shares, and price. You are to create a single file containing all the trades from the 500 files, sorted in order of increasing trade times. The individual files are of the order of 5–100 megabytes; the combined file will be of the order of five gigabytes.

Problem 7.1: Design an algorithm that takes a set of files containing stock trades sorted by increasing trade times, and writes a single file containing the trades appearing in the individual files sorted in the same order. The algorithm should use very little RAM, ideally of the order of a few kilobytes. *pg. 55*

Chapter

8

Searching



— "*The Anatomy of A Large-Scale Hypertextual Web Search Engine*," S. M. Brin and L. Page, 1998

Search algorithms can be classified in a number of ways. Is the underlying collection static or dynamic, i.e., inserts and deletes are interleaved with searching? Is worth spending the computational cost to preprocess the data so as to speed up subsequent queries? Are there statistical properties of the data that can be exploited? Should we operate directly on the data or transform it?

In this chapter, our focus is on static data stored in sorted order in an array. Data structures appropriate for dynamic updates are the subject of Chapters 7, 9, and11.

The first collection of problems in this chapter are related to binary search. The second collection pertains to general search.

Binary search

Binary search is at the heart of more interview questions than any other single algorithm. Given an arbitrary collection of n keys, the only way to determine if a search key is present is by examining each element. This has $O(n)$ time complexity.

Fundamentally, binary search is a natural elimination-based strategy for searching a sorted array. The idea is to eliminate half the keys from consideration by keeping the keys in sorted order. If the search key is not equal to the middle element of the array, one of the two sets of keys to the left and to the right of the middle element can be eliminated from further consideration.

Questions based on binary search are ideal from the interviewers perspective: it is a basic technique that every reasonable candidate is supposed to know and it can be implemented in a few lines of code. On the other hand, binary search is much trickier to implement correctly than it appears—you should implement it as well as write corner case tests to ensure you understand it properly.

Many published implementations are incorrect in subtle and not-so-subtle ways—a study reported that it is correctly implemented in only five out of twenty textbooks. Jon Bentley, in his book "*Programming Pearls*" reported that he assigned binary search in a course for professional programmers and found that 90% failed to code it correctly despite having ample time. (Bentley's students would have been gratified to know that his own published implementation of binary search, in a column titled "Writing Correct Programs", contained a bug that remained undetected for over twenty years.)

Binary search can be written in many ways—recursive, iterative, different idioms for conditionals, etc. Here is an iterative implementation adapted from Bentley's book, which includes his bug.

```
1  int bsearch(int t, const vector<int>& A) {
2    int L = 0, U = A.size() - 1;
3    while (L <= U) {
4      int M = (L + U) / 2;
5      if (A[M] < t) {
6        L = M + 1;
7      } else if (A[M] == t) {
8        return M;
9      } else {
10       U = M - 1;
11     }
12   }
13   return -1;
14 }
```

The error is in the assignment `M = (L + U) / 2` in Line 4, which can lead to overflow. A common solution is to use `M = L + (U - L) / 2`.

However, even this refinement is problematic in a C-style implementation. *The C Programming Language (2nd ed.)* by Kernighan and Ritchie (Page 100) states: "If one is sure that the elements exist, it is also possible to index backwards in an array; `p[-1]`, `p[-2]`, etc. are syntactically legal, and refer to the elements that immediately precede `p[0]`." In the expression `L + (U - L) / 2`, if `U` is a sufficiently large positive integer and `L` is a sufficiently large negative integer, `(U - L)` can overflow, leading to out of bounds array access. The problem is illustrated below:

```
1 #define N 3000000000
2 char A[N];
```

```
char* B = (A + 1500000000);
int L = -1499000000;
int U = 1499000000;
// On a 32-bit machine (U - L) = -1296967296 because the actual value,
// 2998000000 is larger than 2^31 - 1. Consequently, the bsearch function
// called below sets m to -2147483648 instead of 0, which leads to an
// out-of-bounds access, since the most negative index that can be applied
// to B is -1500000000.
int result = BinarySearch(key, B, L, U);
```

The solution is to check the signs of `L` and `U`. If `U` is positive and `L` is negative, `M = (L + U) / 2` is appropriate, otherwise set `M = L + (U - L) / 2`.

In our solutions that make use of binary search, `L` and `U` are nonnegative and so we use `M = L + (U - L) / 2` in the associated programs.

The time complexity of binary search is given by $T(n) = T(n/2) + c$, where c is a constant. This solves to $T(n) = O(\log n)$, which is far superior to the $O(n)$ approach needed when the keys are unsorted. A disadvantage of binary search is that it requires a sorted array and sorting an array takes $O(n \log n)$ time. However, if there are many searches to perform, the time taken to sort is not an issue.

Many variants of searching a sorted array require a little more thinking and create opportunities for missing corner cases.

8.1 Search a sorted array for first occurrence of k

Binary search commonly asks for the index of any element of a sorted array A that is equal to a given element. The following problem has a slight twist on this.

-14	-10	2	108	108	243	285	285	285	401
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$

Figure 8.1: A sorted array with repeated elements.

Problem 8.1: Write a method that takes a sorted array A and a key k and returns the index of the *first* occurrence of k in A. Return -1 if k does not appear in A. For example, when applied to the array in Figure 8.1 your algorithm should return 3 if $k = 108$; if $k = 285$, your algorithm should return 6. *pg. 56*

Chapter

9 Hash Tables

The new methods are intended to reduce the amount of space required to contain the hash-coded information from that associated with conventional methods. The reduction in space is accomplished by exploiting the possibility that a small fraction of errors of commission may be tolerable in some applications.

— "Space/time trade-offs in hash coding with allowable errors,"
B. H. Bloom, 1970

The idea underlying a *hash table* is to store objects according to their key field in an array. Objects are stored in array locations based on the "hash code" of the key. The hash code is an integer computed from the key by a hash function. If the hash function is chosen well, the objects are distributed uniformly across the array locations.

If two keys map to the same location, a "collision" is said to occur. The standard mechanism to deal with collisions is to maintain a linked list of objects at each array location. If the hash function does a good job of spreading objects across the underlying array and take $O(1)$ time to compute, on average, lookups, insertions, and deletions have $O(1 + n/m)$ time complexity, where n is the number of objects and m is the length of the array. If the "load" n/m grows large, rehashing can be applied to the hash table. A new array with a larger number of locations is allocated, and the objects are moved to the new array. Rehashing is expensive ($O(n + m)$ time) but if it is done infrequently (for example, whenever the number of entries doubles), its amortized cost is low.

A hash table is qualitatively different from a sorted array—keys do not have to appear in order, and randomization (specifically, the hash function) plays a central role. Compared to binary search trees (discussed in Chapter 11), inserting and deleting in a hash table is more efficient (assuming rehashing is infrequent). One disadvantage of hash tables is the need for a good hash function but this is rarely an issue in practice. Similarly, rehashing is not a problem outside of realtime systems and even for such systems, a separate thread can do the rehashing.

A hash function has one hard requirement—equal keys should have equal hash codes. This may seem obvious, but is easy to get wrong, e.g., by writing a hash function that is based on address rather than contents, or by including profiling data.

A softer requirement is that the hash function should "spread" keys, i.e., the hash codes for a subset of objects should be uniformly distributed across the underlying array. In addition, a hash function should be efficient to compute.

Now we illustrate the steps in designing a hash function suitable for strings. First, the hash function should examine all the characters in the string. (If this seem obvious, the string hash function in the original distribution of Java examined at most 16 characters, in an attempt to gain speed, but often resulted in very poor performance because of collisions.)

It should give a large range of values, and should not let one character dominate (e.g., if we simply cast characters to integers and multiplied them, a single 0 would result in a hash code of 0). We would also like a rolling hash function, one in which if a character is deleted from the front of the string, and another added to the end, the new hash code can be computed in $O(1)$ time. The following function has these properties:

```
int StringHash(const string& str, int modulus) {
  const int kMult = 997;
  int val = 0;
  for (const char& c : str) {
    val = (val * kMult + c) % modulus;
  }
  return val;
}
```

A hash table is a good data structure to represent a dictionary, i.e., a set of strings. In some applications, a trie, which is an tree data structure that is used to store a dynamic set of strings. Unlike a BST, nodes in the tree do not store a key. Instead, the node's position in the tree defines the key which it is associated with.

9.1 Test if an anonymous letter is constructible

A hash table can be viewed as a dictionary. For this reason, hash tables commonly appear in string processing.

Problem 9.1: You are required to write a method which takes an anonymous letter L and text from a magazine M. Your method is to return `true` if and only if L can be written using M, i.e., if a letter appears k times in L, it must appear at least k times in M. *pg. 57*

Chapter

10 Sorting

PROBLEM 14 *(Meshing). Two monotone sequences S, T, of lengths n, m, respectively, are stored in two systems of $n(p+1)$, $m(p+1)$ consecutive memory locations, respectively: $s, s+1, \ldots, s+n(p+1)-1$ and $t, t+1, \ldots, t+m(p+1)-1$. . . . It is desired to find a monotone permutation R of the sum $[S, T]$, and place it at the locations $r, r+1, \ldots, r+(n+m)(p+1)-1$.*

— *"Planning And Coding Of Problems For An Electronic Computing Instrument,"* H. H. Goldstine and J. von Neumann, 1948

Sorting—rearranging a collection of items into increasing or decreasing order—is a common problem in computing. Sorting is used to preprocess the collection to make searching faster (as we saw with binary search through an array), as well as identify items that are similar (e.g., students are sorted on test scores).

Naive sorting algorithms run in $O(n^2)$ time. A number of sorting algorithms run in $O(n \log n)$ time—heapsort, merge sort, and quicksort are examples. Each has its advantages and disadvantages: for example, heapsort is in-place but not stable; merge sort is stable but not in-place; quicksort runs $O(n^2)$ time in worst case. (An in-place sort is one which uses $O(1)$ space; a stable sort is one where entries which are equal appear in their original order.)

A well-implemented quicksort is usually the best choice for sorting. We briefly outline alternatives that are better in specific circumstances.

For short arrays, e.g., 10 or fewer elements, insertion sort is easier to code and faster than asymptotically superior sorting algorithms. If every element is known to be at most k places from its final location, a min-heap can be used to get an $O(n \log k)$ algorithm. If there are a small number of distinct keys, e.g., integers in the range [0..255], counting sort, which records for each element, the number of elements less than it, works well. This count can be kept in an array (if the largest number is comparable in value to the size of the set being sorted) or a BST, where the keys are the numbers and the values are their frequencies. If there are many duplicate keys we can add the keys to a BST, with linked lists for elements which have the same key; the sorted result can be derived from an in-order traversal of the BST.

Most sorting algorithms are not stable. Merge sort, carefully implemented, can be made stable. Another solution is to add the index as an integer rank to the keys to break ties.

Most sorting routines are based on a compare function that takes two items as input and returns −1 if the first item is smaller than the second item, 0 if they are equal and 1 otherwise. However, it is also possible to use numerical attributes directly, e.g., in radix sort.

The heap data structure is discussed in detail in Chapter 7. Briefly, a max-heap (min-heap) stores keys drawn from an ordered set. It supports $O(\log n)$ inserts and $O(1)$ time lookup for the maximum (minimum) element; the maximum (minimum) key can be deleted in $O(\log n)$ time. Heaps can be helpful in sorting problems, as illustrated by Problem 7.1 on Page 13.

10.1 Compute the intersection of two sorted arrays

A natural implementation for a search engine is to retrieve documents that match the set of words in a query by maintaining an inverted index. Each page is assigned an integer identifier, its *document-ID*. An inverted index is a mapping that takes a word w and returns a sorted array of page-ids which contain w—the sort order could be, for example, the page rank in descending order. When a query contains multiple words, the search engine finds the sorted array for each word and then computes the intersection of these arrays—these are the pages containing all the words in the query. The most computationally intensive step of doing this is finding the intersection of the sorted arrays.

Problem 10.1: Given sorted arrays A and B of lengths n and m respectively, return an array C containing elements common to A and B. The array C should be free of duplicates. How would you perform this intersection if—(1.) $n \approx m$ and (2.) $n \ll m$?
pg. 58

10.2 Render a calendar

Consider the problem of designing an online calendaring application. One component of the design is to render the calendar, i.e., display it visually.

Suppose each day consists of a number of events, where an event is specified as a start time and a finish time. Individual events for a day are to be rendered as nonoverlapping rectangular regions whose sides are parallel to the x- and y-axes. Let the x-axis correspond to time. If an event starts at time b and ends at time e, the upper and lower sides of its corresponding rectangle must be at b and e, respectively. Figure 10.1 on the following page represents a set of events.

Suppose the y-coordinates for each day's events must lie between 0 and L (a prespecified constant), and the rectangle for each event has the same "height", which is the distance between the sides parallel to the x-axis is fixed. Your task is to compute the maximum height an event rectangle can have. In essence, this is equivalent to the following problem.

Problem 10.2: Given a set of n events, how would you determine the maximum number of events that take place concurrently?
pg. 59

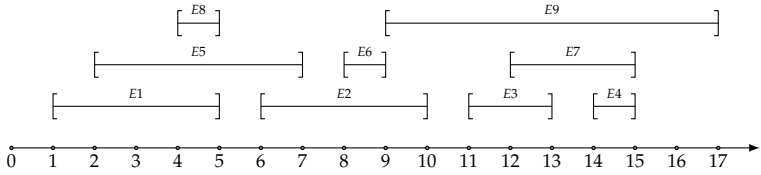


Figure 10.1: A set of nine events. The earliest starting event begins at time 1; the latest ending event ends at time 17. The maximum number of concurrent events is 3, e.g., $\{E1, E5, E8\}$ as well as others.

Chapter

11 Binary Search Trees

The number of trees which can be formed with $n+1$ given knots $\alpha, \beta, \gamma, \ldots = (n+1)^{n-1}$.

— "*A Theorem on Trees*," A. Cayley, 1889

Adding and deleting elements to an array is computationally expensive, particularly when the array needs to stay sorted. BSTs are similar to arrays in that the keys are in a sorted order. However, unlike arrays, elements can be added to and deleted from a BST efficiently. BSTs require more space than arrays since each node stores two pointers, one for each child, in addition to the key.

A BST is a binary tree as defined in Chapter 6 in which the nodes store keys drawn from a totally ordered set. The keys stored at nodes have to respect the BST property—the key stored at a node is greater than or equal to the keys stored at the nodes of its left subtree and less than or equal to the keys stored in the nodes of its right subtree. Figure 11.1 on the next page shows a BST whose keys are the first 16 prime numbers.

Key lookup, insertion, and deletion take time proportional to the height of the tree, which can in worst-case be $O(n)$, if insertions and deletions are naively implemented. However, there are implementations of insert and delete which guarantee the tree has height $O(\log n)$. These require storing and updating additional data at the tree nodes. Red-black trees are an example of balanced BSTs and are widely used in data structure libraries, e.g., to implement `set` and `map` in the C++ Standard Template Library (STL).

The BST prototype is as follows:

```
1 template <typename T>
2 struct BSTNode {
3   T data;
4   unique_ptr<BSTNode<T>> left, right;
5 };
```

11.1 Test if a binary tree satisfies the BST property

Problem 11.1: Write a function that takes as input the root of a binary tree whose nodes have a key field, and returns `true` if and only if the tree satisfies the BST property. ***pg. 60***

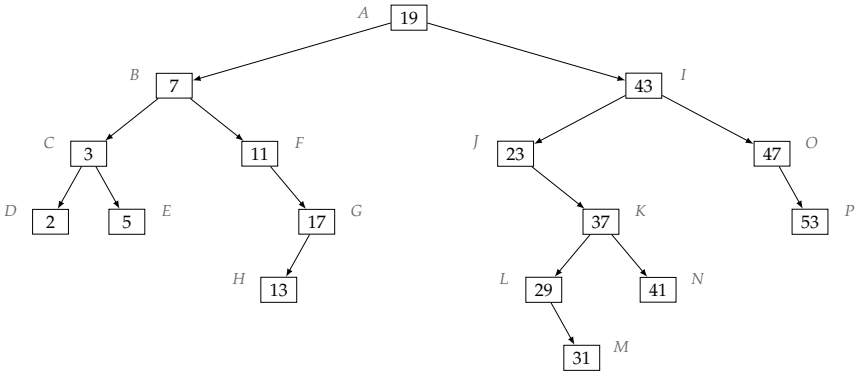


Figure 11.1: An example BST.

Chapter

12 Recursion

The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions.

— "*Algorithms + Data Structures = Programs*,"
N. E. Wirth, 1976

Recursion is a method where the solution to a problem depends partially on solutions to smaller instances of related problems. Two key ingredients to a successful use of recursion are identifying the base cases, which are to be solved directly, and ensuring progress, that is the recursion converges to the solution.

Both backtracking and branch-and-bound are naturally formulated using recursion.

Chapter 13 describes dynamic programming, which conceptually is based on recursion augmented with a cache to avoid solving the same problem multiple times.

12.1 Enumerate the power set

The power set of a set S is the set of all subsets of S, including both the empty set $\emptyset$ and S itself. The power set of $\{A, B, C\}$ is graphically illustrated in Figure 12.1.

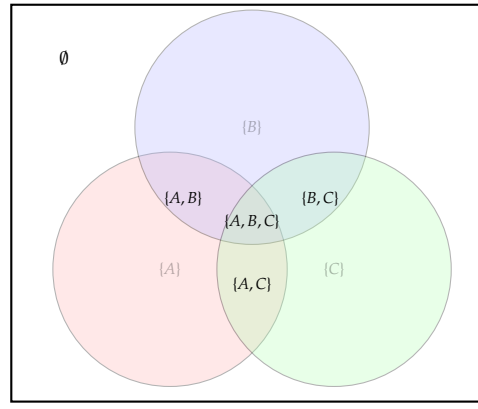


Figure 12.1: The power set of $\{A, B, C\}$ is $\{\emptyset, \{A\}, \{B\}, \{C\}, \{A, B\}, \{B, C\}, \{A, C\}, \{A, B, C\}\}$.

Problem 12.1: Implement a method that takes as input a set S of n distinct elements, and prints the power set of S. Print the subsets one per line, with elements separated by commas. *pg. 63*

Chapter

13 Dynamic Programming

The important fact to observe is that we have attempted to solve a maximization problem involving a particular value of x and a particular value of N by first solving the general problem involving an arbitrary value of x and an arbitrary value of N.

— "Dynamic Programming,"
R. E. Bellman, 1957

DP is a general technique for solving complex optimization problems that can be decomposed into overlapping subproblems. Like divide-and-conquer, we solve the problem by combining the solutions of multiple smaller problems but what makes DP different is that the subproblems may not be independent. A key to making DP efficient is reusing the results of intermediate computations. (The word "programming" in dynamic programming does not refer to computer programming—the word was chosen by Richard Bellman to describe a program in the sense of a schedule.) Problems which are naturally solved using DP are a popular choice for hard interview questions.

To illustrate the idea underlying DP, consider the problem of computing Fibonacci numbers defined by $F_n = F_{n-1} + F_{n-2}$, $F_0 = 0$ and $F_1 = 1$. A function to compute F_n that recursively invokes itself to compute F_{n-1} and F_{n-2} would have a time complexity that is exponential in n. However, if we make the observation that recursion leads to computing F_i for $i \in [0, n-1]$ repeatedly, we can save the computation time by storing these results and reusing them. This makes the time complexity linear in n, albeit at the expense of $O(n)$ storage. Note that the recursive implementation requires $O(n)$ storage too, though on the stack rather than the heap and that the function is not tail recursive since the last operation performed is + and not a recursive call.

The key to solving any DP problem efficiently is finding the right way to break the problem into subproblems such that

- the bigger problem can be solved relatively easily once solutions to all the subproblems are available, and
- you need to solve as few subproblems as possible.

In some cases, this may require solving a slightly different optimization problem than the original problem. For example, consider the following problem: given an array of integers A of length n, find the interval indices a and b such that $\sum_{i=a}^{b} A[i]$ is maximized. As a concrete example, the interval corresponding to the maximum subarray sum for the array in Figure 13.1 on the next page is $[0, 3]$.

904	40	523	12	-335	-385	-124	481	-31
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$

Figure 13.1: An array with a maximum subarray sum of 1479.

The brute-force algorithm, which computes each subarray sum, has $O(n^3)$ time complexity—there are $\frac{n(n-1)}{2}$ subarrays, and each subarray sum can be computed in $O(n)$ time. The brute-force algorithm can be improved to $O(n^2)$ by first computing sums $S[i]$ for subarrays $A[0 : i]$ for each $i < n$; the sum of subarray $A[i : j]$ is $S[j] - S[i-1]$, where $S[-1]$ is taken to be 0.

Here is a natural divide-and-conquer algorithm. We solve the problem for the subarrays $L = A[0 : \lfloor \frac{n}{2} \rfloor]$ and $R = A[\lfloor \frac{n}{2} \rfloor + 1 : n-1]$. In addition to the answers for each, we also return the maximum subarray sum l for any subarray ending at the last entry in L, and the maximum subarray sum r for any subarray starting at 0 for R. The maximum subarray sum for A is the maximum of $l + r$, the answer for L, and the answer for R. The time complexity analysis is similar to that for quicksort, which leads to an $O(n \log n)$.

Now we will solve this problem by using DP. A natural thought is to assume we have the solution for the subarray $A[0 : n-2]$. However, even if we knew the largest sum subarray for subarray $A[0 : n-2]$, it does not help us solve the problem for $A[0 : n-1]$. A better approach is to iterate through the array. For each index j, the maximum subarray ending at j is equal to $S[j] - \min_{i \leq j} S[i]$. During the iteration, we cache the minimum subarray sum we have visited and compute the maximum subarray for each index. The time spent per index is constant, leading to an $O(n)$ time and $O(1)$ space solution. The code below returns a pair of indices (i, j) such that $A[i : j-1]$ is a maximum subarray. It is legal for all array entries to be negative, or the array to be empty. The algorithm handles these input cases correctly. Specifically, it returns equal indices, which denote an empty subarray.

```
pair<int, int> FindMaximumSubarray(const vector<int>& A) {
  // A[range.first : range.second - 1] will be the maximum subarray.
  pair<int, int> range(0, 0);
  int min_idx = -1, min_sum = 0, sum = 0, max_sum = 0;
  for (int i = 0; i < A.size(); ++i) {
    sum += A[i];
    if (sum < min_sum) {
      min_sum = sum, min_idx = i;
    }
    if (sum - min_sum > max_sum) {
      max_sum = sum - min_sum, range = {min_idx + 1, i + 1};
    }
  }
  return range;
}
```

Here are two variants of the subarray maximization problem that can be solved

with ideas that are similar to the above approach: find indices a and b such that $\sum_{i=a}^{b} A[i]$ is—(1.) closest to 0 and (2.) closest to t. (Both entail some sorting, which increases the time complexity to $O(n \log n)$.) Another good variant is finding indices a and b such that $\prod_{i=a}^{b} A[i]$ is maximum when the array contains both positive and negative integers.

A common mistake in solving DP problems is trying to think of the recursive case by splitting the problem into two equal halves, *a la* quicksort, i.e., somehow solve the subproblems for subarrays $A[0 : \lfloor \frac{n}{2} \rfloor]$ and $A[\lfloor \frac{n}{2} \rfloor + 1 : n]$ and combine the results. However, in most cases, these two subproblems are not sufficient to solve the original problem.

13.1 Count the number of ways to traverse a 2D array

Suppose you start at the top-left corner of an $n \times m$ 2D array A and want to get to the bottom-right corner. The only way you can move is by either going right or going down. Three legal paths for a 5×5 2D array are given in Figure 13.2.

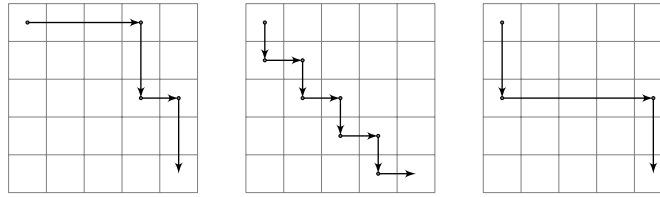


Figure 13.2: Paths through a 2D array.

Problem 13.1: How many ways can you go from the top-left to the bottom-right in an $n \times m$ 2D array? How would you count the number of ways in the presence of obstacles, specified by an $n \times m$ Boolean 2D array B, where a `true` represents an obstacle. *pg. 64*

Chapter

14 Greedy Algorithms and Invariants

The intended function of a program, or part of a program, can be specified my making general assertions about the values which the relevant variables will take after *execution of the program.*

— "*An Axiomatic Basis for Computer Programming*," C. A. R. Hoare, 1969

Invariants

An invariant is a condition that is true during execution of a program. Invariants can be used to design algorithms as well as reason about their correctness.

The divide and conquer algorithms seen previously, e.g., binary search, maintain the invariant that the space of candidate solutions contains all possible solutions as the algorithms execute.

Sorting algorithms nicely illustrates algorithm design using invariants. For example, intuitively, selection sort is based on finding the smallest element, the next smallest element, etc. and moving them to their right place. More precisely, we work with successively larger subarrays beginning at index 0, and preserve the invariant that these subarrays are sorted and their elements are less than or equal to the remaining elements.

As another example, suppose we want to find two elements in a sorted array A summing to a specified K. Let n denote the length of A. We start by considering $s_{0,n-1} = A[0] + A[n-1]$. If $s_{0,n-1} = K$, we are done. If $s_{0,n-1} < K$, then we can restrict our attention to solving the problem on the subarray $A[1 : n-1]$, since $A[0]$ can never be one of the two elements. Similarly, if $s_{0,n-1} > K$, we restrict the search to $A[0 : n-2]$. The invariant is that if two elements which sum to K exist, they must lie within the subarray currently under consideration.

14.1 The 3-sum problem

Let A be an array of n numbers. Let t be a number, and k be an integer in $[1, n]$. Define A to k-create t if and only if there exists k indices $i_0, i_1, \ldots, i_{k-1}$ (not necessarily distinct) such that $\sum_{j=0}^{k-1} A[i_j] = t$.

Problem 14.1: Design an algorithm that takes as input an array A and a number t, and determines if A 3-creates t. ***pg. 66***

Chapter

15 Graphs

Concerning these bridges, it was asked whether anyone could arrange a route in such a way that he would cross each bridge once and only once.

— "The solution of a problem relating to the geometry of position," L. Euler, 1741

Informally, a graph is a set of vertices and connected by edges. Formally, a directed graph is a tuple (V, E), where V is a set of *vertices* and $E \subset V \times V$ is the set of edges. Given an edge $e = (u, v)$, the vertex u is its *source*, and v is its *sink*. Graphs are often decorated, e.g., by adding lengths to edges, weights to vertices, and a start vertex. A directed graph can be depicted pictorially as in Figure 15.1.

A *path* in a directed graph from u to vertex v is a sequence of vertices $\langle v_0, v_1, \ldots, v_{n-1} \rangle$ where $v_0 = u$, $v_{n-1} = v$, and $(v_i, v_{i+1}) \in E$ for $i \in \{0, \ldots, n-2\}$. The sequence may contain a single vertex. The *length* of the path $\langle v_0, v_1, \ldots, v_{n-1} \rangle$ is $n-1$. Intuitively, the *length* of a path is the number of edges it traverses. If there exists a path from u to v, v is said to be *reachable* from u.

For example, the sequence $\langle a, c, e, d, h \rangle$ is a path in the graph represented in Figure 15.1.

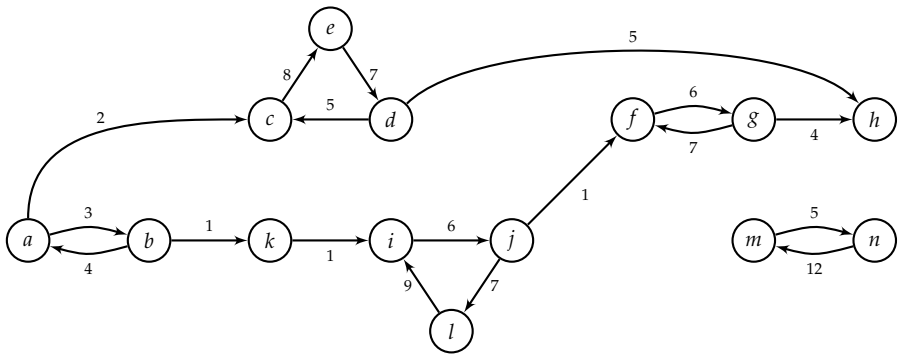


Figure 15.1: A directed graph with weights on edges.

A *directed acyclic graph* (*DAG*) is a directed graph in which there are no *cycles*, i.e., paths of the form $\langle v_0, v_1, \ldots, v_{n-1}, v_0 \rangle, n \geq 1$. See Figure 15.2 on the next page for an example of a directed acyclic graph. Vertices in a DAG which have no incoming

edges are referred to as *sources*; vertices which have no outgoing edges are referred to as *sinks*. A *topological ordering* of the vertices in a DAG is an ordering of the vertices in which each edge is from a vertex earlier in the ordering to a vertex later in the ordering.

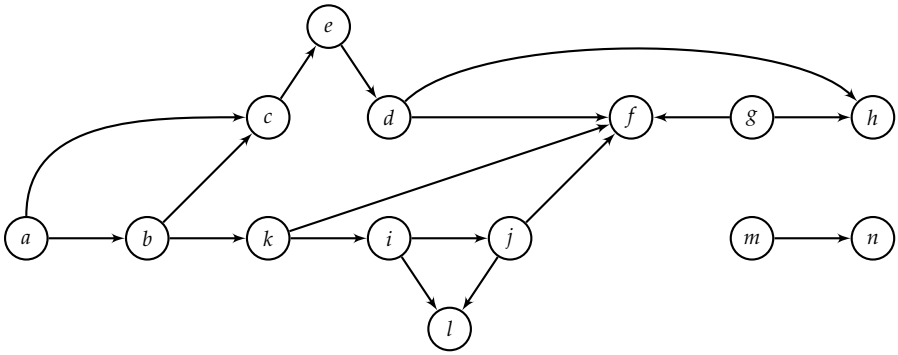


Figure 15.2: A directed acyclic graph. Vertices a, g, m are sources and vertices l, f, h, n are sinks. The ordering $\langle a, b, c, e, d, g, h, k, i, j, f, l, m, n \rangle$ is a topological ordering of the vertices.

An undirected graph is also a tuple (V, E); however, E is a set of unordered pairs of V. Graphically, this is captured by drawing arrowless connections between vertices, as in Figure 15.3.

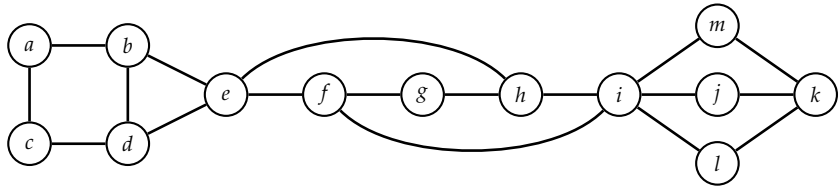


Figure 15.3: An undirected graph.

If G is an undirected graph, vertices u and v are said to be *connected* if G contains a path from u to v; otherwise, u and v are said to be *disconnected*. A graph is said to be connected if every pair of vertices in the graph is connected. A *connected component* is a maximal set of vertices C such that each pair of vertices in C is connected in G. Every vertex belongs to exactly one connected component.

A directed graph is called *weakly connected* if replacing all of its directed edges with undirected edges produces a connected undirected graph. It is *connected* if it contains a directed path from u to v or a directed path from v to u for every pair of vertices u and v. It is *strongly connected* if it contains a directed path from u to v and a directed path from v to u for every pair of vertices u and v.

Graphs naturally arise when modeling geometric problems, such as determining connected cities. However, they are more general, and can be used to model many kinds of relationships.

A graph can be implemented in two ways—using *adjacency lists* or an *adjacency matrix*. In the adjacency list representation, each vertex v, has a list of vertices to which it has an edge. The adjacency matrix representation uses a $|V| \times |V|$ Boolean-valued matrix indexed by vertices, with a 1 indicating the presence of an edge. The time and space complexities of a graph algorithm are usually expressed as a function of the number of vertices and edges.

A *tree* (sometimes called a free tree) is a special sort of graph—it is an undirected graph that is connected but has no cycles. (Many equivalent definitions exist, e.g., a graph is a free tree if and only if there exists a unique path between every pair of vertices.) There are a number of variants on the basic idea of a tree. A rooted tree is one where a designated vertex is called the root, which leads to a parent-child relationship on the nodes. An ordered tree is a rooted tree in which each vertex has an ordering on its children. Binary trees, which are the subject of Chapter 6, differ from ordered trees since a node may have only one child in a binary tree, but that node may be a left or a right child, whereas in an ordered tree no analogous notion exists for a node with a single child. Specifically, in a binary tree, there is position as well as order associated with the children of nodes.

As an example, the graph in Figure 15.4 is a tree. Note that its edge set is a subset of the edge set of the undirected graph in Figure 15.3 on the facing page. Given a graph $G = (V, E)$, if the graph $G' = (V, E')$ where $E' \subset E$, is a tree, then G' is referred to as a spanning tree of G.

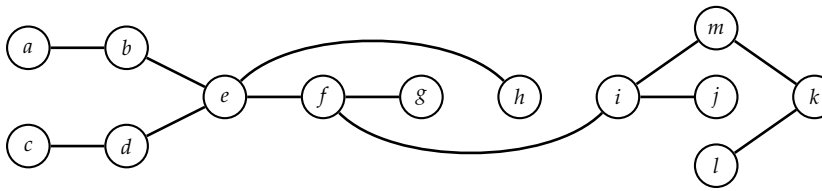


Figure 15.4: A tree.

Graph search

Computing vertices which are reachable from other vertices is a fundamental operation which can be performed in one of two idiomatic ways, namely depth-first search (DFS) and breadth-first search (BFS). Both have linear time complexity—$O(|V| + |E|)$ to be precise. In a worst case there is a path from the initial vertex covering all vertices without any repeats, and the DFS edges selected correspond to this path, so the space complexity of DFS is $O(|V|)$ (this space is implicitly allocated on the function call stack). The space complexity of BFS is also $O(|V|)$, since in a worst case there is an edge from the initial vertex to all remaining vertices, implying that they will all be in the BFS queue simultaneously at some point.

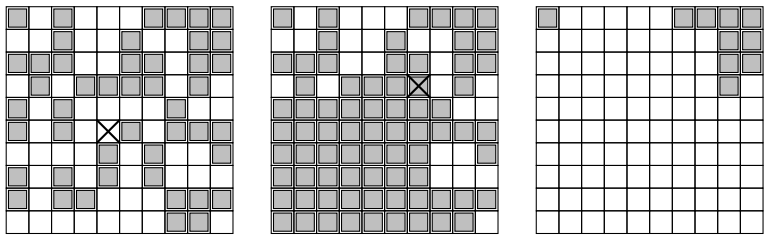
DFS and BFS differ from each other in terms of the additional information they provide, e.g., BFS can be used to compute distances from the start vertex and DFS can be used to check for the presence of cycles. Key notions in DFS include the concept of *discovery time* and *finishing time* for vertices.

15.1 Paint a Boolean matrix

Let A be a $D \times D$ Boolean 2D array encoding a black-and-white image. The entry $A(a, b)$ can be viewed as encoding the color at location (a, b). Define a path from entry (a, b) to entry (c, d) to be a sequence of entries $\langle (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \rangle$ such that

- $(a, b) = (x_1, y_1)$, $(c, d) = (x_n, y_n)$, and
- for each i, $1 \leq i < n$, we have $|x_i - x_{i+1}| + |y_i - y_{i+1}| = 1$.

Define the region associated with a point (i, j) to be all points (i', j') such that there exists a path from (i, j) to (i', j') in which all entries are the same color. In particular this implies (i, j) and (i', j') must be the same color.



(a) (b) (c)

Figure 15.5: The color of all squares associated with the first square marked with a $\times$ in (a) have been recolored to yield the coloring in (b). The same process yields the coloring in (c).

Problem 15.1: Implement a routine that takes a $n \times m$ Boolean array A together with an entry (x, y) and flips the color of the region associated with (x, y). See Figure 15.5 for an example of flipping. *pg. 67*

Chapter

16

Parallel Computing

The activity of a computer must include the proper reacting to a possibly great variety of messages that can be sent to it at unpredictable moments, a situation which occurs in all information systems in which a number of computers are coupled to each other.

— "*Cooperating sequential processes*,"
E. W. Dijkstra, 1965

Parallel computation has become increasingly common. For example, laptops and desktops come with multiple processors which communicate through shared memory. High-end computation is often done using clusters consisting of individual computers communicating through a network.

Parallelism provides a number of benefits:

- High performance—more processors working on a task (usually) means it is completed faster.
- Better use of resources—a program can execute while another waits on the disk or network.
- Fairness—letting different users or programs share a machine rather than have one program run at a time to completion.
- Convenience—it is often conceptually more straightforward to do a task using a set of concurrent programs for the subtasks rather than have a single program manage all the subtasks.
- Fault tolerance—if a machine fails in a cluster that is serving web pages, the others can take over.

Concrete applications of parallel computing include graphical user interfaces (GUI) (a dedicated thread handles UI actions while other threads are, for example, busy doing network communication and passing results to the UI thread, resulting in increased responsiveness), Java virtual machines (a separate thread handles garbage collection which would otherwise lead to blocking, while another thread is busy running the user code), web servers (a single logical thread handles a single client request), scientific computing (a large matrix multiplication can be split across a cluster), and web search (multiple machines crawl, index, and retrieve web pages).

The two primary models for parallel computation are the shared memory model, in which each processor can access any location in memory, and the distributed memory model, in which a processor must explicitly send a message to another processor to access its memory. The former is more appropriate in the multicore setting and the latter is more accurate for a cluster. The questions in this chapter are

mostly focused on the shared memory model. We cover a few problems related to the distributed memory model, such as leader election and sorting large data sets, at the end of the chapter.

Writing correct parallel programs is challenging because of the subtle interactions between parallel components. One of the key challenges is races—two concurrent instruction sequences access the same address in memory and at least one of them writes to that address. Other challenges to correctness are

- starvation (a processor needs a resource but never gets it),
- deadlock (Thread A acquires Lock $L1$ and Thread B acquires Lock $L2$, following which A tries to acquire $L2$ and B tries to acquire $L1$), and
- livelock (a processor keeps retrying an operation that always fails).

Bugs caused by these issues are difficult to find using testing. Debugging them is also difficult because they may not be reproducible since they are usually load dependent. It is also often true that it is not possible to realize the performance implied by parallelism—sometimes a critical task cannot be parallelized, making it impossible to improve performance, regardless of the number of processors added. Similarly, the overhead of communicating intermediate results between processors can exceed the performance benefits.

16.1 Implement a Timer class

Consider a web-based calendar in which the server hosting the calendar has to perform a task when the next calendar event takes place. (The task could be sending an email or a Short Message Service (SMS).) Your job is to design a facility that manages the execution of such tasks.

Problem 16.1: Develop a `Timer` class that manages the execution of deferred tasks. The `Timer` constructor takes as its argument an object which includes a `Run` method and a `name` field, which is a string. `Timer` must support—(1.) starting a thread, identified by name, at a given time in the future; and (2.) canceling a thread, identified by name (the cancel request is to be ignored if the thread has already started). ***pg. 69***

Chapter

17

Design Problems

We have described a simple but very powerful and flexible protocol which provides for variation in individual network packet sizes, transmission failures, sequencing, flow control, and the creation and destruction of process- to-process associations.

— "*A Protocol for Packet Network Intercommunication*," V. G. Cerf and R. E. Kahn, 1974

You may be asked in an interview how to go about creating a set of services or a larger system, possibly on top of an algorithm that you have designed. These problems are fairly open-ended, and many can be the starting point for a large software project.

In an interview setting when someone asks such a question, you should have a conversation in which you demonstrate an ability to think creatively, understand design trade-offs, and attack unfamiliar problems. You should sketch key data structures and algorithms, as well as the technology stack (programming language, libraries, OS, hardware, and services) that you would use to solve the problem.

The answers in this chapter are presented in this context—they are meant to be examples of good responses in an interview and are not comprehensive state-of-the-art solutions.

We review patterns that are useful for designing systems in Table 17.1. Some other things to keep in mind when designing a system are implementation time, extensibility, scalability, testability, security, internationalization, and IP issues.

Table 17.1: System design patterns.

Design principle	Key points
Decomposition	Split the functionality, architecture, and code into manageable, reusable components.
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.

Decomposition

Good decompositions are critical to successfully solving system-level design problems. Functionality, architecture, and code all benefit from decomposition.

For example, in our solution to designing a system for online advertising(Problem **??** on Page ??) , we decompose the goals into categories based on the

stake holders. We decompose the architecture itself into a front-end and a back-end. The front-end is divided into user management, web page design, reporting functionality, etc. The back-end is made up of middleware, storage, database, cron services, and algorithms for ranking ads. The design of TEX (Problem **??** on Page ??) and Connexus (Problem **??** on Page ??) also illustrate such decompositions.

Decomposing code is a hallmark of object-oriented programming. The subject of design patterns is concerned with finding good ways to achieve code-reuse. Broadly speaking, design patterns are grouped into creational, structural, and behavioral patterns. Many specific patterns are very natural—strategy objects, adapters, builders, etc., appear in a number of places in our codebase. Freeman *et al.*'s *"Head First Design Patterns"* is, in our opinion, the right place to study design patterns.

Parallelism

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

- each subproblem can be solved relatively independently, and
- 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 memory 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 **??** on Page ??.

The solutions to Problems **??** on Page ?? and **??** on Page ?? also illustrate the use of parallelism. The solution to Problem **??** on Page ?? also illustrates the use of parallelism.

Caching

Caching is a great tool whenever computations are repeated. 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 many requests are repeated. Workloads on web services exhibit this property. Solution **??** on Page ?? sketches the design of an online spell correction service; one of the key issues is performing cache updates in the presence of concurrent requests. Solution **??** on Page ?? shows how multithreading combines with caching in code which tests the Collatz hypothesis.

17.1 Design a system for detecting copyright infringement

YouTV.com is a successful online video sharing site. Hollywood studios complain that much of the material uploaded to the site violates copyright.

Problem 17.1: Design a feature that allows a studio to enter a set V of videos that belong to it, and to determine which videos in the YouTV.com database match videos in V. ***pg. 70***

Part II

Hints

Hints

When I interview people, and they give me an immediate answer, they're often not thinking. So I'm silent. I wait. Because they think they have to keep answering. And it's the second train of thought that's the better answer.

— R. Leach

Use a hint after you have made a serious attempt at the problem. Ideally, the hint should give you the flash of insight needed to complete your solution.

Usually, you will receive hints only after you have shown an understanding of the problem, and have made serious attempts to solve it.

1.1: Use a lookup table, but don't use 2^{64} entries!

1.2: What base can you easily convert to and from?

2.1: Identifying the minimum and maximum heights is not enough since the minimum height may appear after the maximum height. Focus on valid height differences.

3.1: Build the result one digit at a time.

3.2: It's difficult to solve this with one pass.

4.1: Consider using two pointers, one fast and one slow.

5.1: Use additional storage to track the maximum value.

5.2: First think about solving this problem with a pair of queues.

6.1: Think of a classic binary tree algorithm that runs in $O(h)$ additional space.

7.1: Which portion of each file is significant as the algorithm executes?

8.1: Don't stop after you reach the first k. Think about the case where every entry equals k.

9.1: Count.

10.1: Solve the problem if n and m differ by orders of magnitude. What if $n \approx m$?

10.2: Focus on endpoints.

11.1: Is it correct to check for each node that its key is greater than or equal to the key at its left child and less than or equal to the key at its right child?

12.1: There are 2^n subsets for a given set S of size n. There are 2^k k-bit words.

13.1: If $i > 0$ and $j > 0$, you can get to (i, j) from $(i - 1, j)$ or $(j - 1, i)$.

14.1: How would you check if $A[i]$ is part of a triple that 3-creates t?

15.1: Solve this conceptually, then think about implementation optimizations.

16.1: There are two aspects—data structure design and concurrency.

17.1: Normalize the video format and create signatures.

Part III

Solutions

C++11

C++11 adds a number of features that make for elegant and efficient code. The C++11 constructs used in the solution code are summarized below.

- The `auto` attribute assigns the type of a variable based on the initializer expression.
- The enhanced range-based for-loop allows for easy iteration over a list of elements.
- The `emplace_front` and `emplace_back` methods add new elements to the beginning and end of the container. They are more efficient than `push_front` and `push_back`, and are variadic, i.e., takes a variable number arguments. The `emplace` method is similar and applicable to containers where there is only one way to insert (e.g., a stack or a map).
- The `array` type is similar to ordinary arrays, but supports `.size()` and boundary checking. (It does not support automatic and dynamic resizing.)
- The `tuple` type implements an ordered set.
- Anonymous functions ("lambdas") can be written via the `[]` notation.
- An initializer list uses the `{}` notation to avoid having to make explicit calls to constructors when building list-like objects.

Problem 1.1, *pg.* 2: *How would you compute the parity of a very large number of* 64*-bit words?*

Solution 1.1: The brute-force algorithm iteratively tests the value of each bit, and keeps the running XOR of the bits processed.

```
short Parity(unsigned long x) {
  short result = 0;
  while (x) {
    result ^= (x & 1);
    x >>= 1;
  }
  return result;
}
```

In general, if the word size is n bits, the time complexity of the algorithm above is $O(n)$.

There is a neat trick that erases the lowest set bit in a word in a single operation which can be used to improve performance in the best and average cases.

```
short Parity(unsigned long x) {
  short result = 0;
  while (x) {
    result ^= 1;
    x &= (x - 1);  // Drops the lowest set bit of x.
  }
  return result;
}
```

Let k be the number of bits set to 1 set in a particular word. (For example, for 1011, $k = 3$.) Then time complexity of the algorithm above is $O(k)$.

The problem statement refers to computing the parity for a very large number of words. When you have to perform a large number of parity computations, and, more generally, any kind of bit fiddling computations, an excellent approach is to cache computations in an array-based lookup table.

Clearly, we cannot cache the parity of every 64-bit integer. However, when XORing a collection of bits, it does not matter how we group those bits, i.e., the XOR computation is associative. Therefore, we can compute the parity of a 64-bit integer by grouping its bits into four nonoverlapping 16 bit words, computing the parity of each subset, and XORing these four results.

It is feasible to cache the parity of all possible 16-bit words using an array; its length is $2^{16} = 65536$. The array can be statically initialized, or we can use lazy initialization, with a separate flag bit used to indicate whether a particular entry is valid.

The following implementation of parity uses this approach.

```
short Parity(unsigned long x) {
  const int kWordSize = 16;
  const int kBitMask = 0xFFFF;
  return precomputed_parity[x >> (3 * kWordSize)] ^
         precomputed_parity[(x >> (2 * kWordSize)) & kBitMask] ^
         precomputed_parity[(x >> kWordSize) & kBitMask] ^
         precomputed_parity[x & kBitMask];
}
```

The time complexity is a function of the size of the keys used to index the lookup table. Let L be the width of the words for which we cache the results. Since there are n/L terms, the time complexity for n-bit words is $O(n/L)$, assuming word-level operations, such as shifting, take $O(1)$ time. (This does not include the time for initialization of the lookup table.)

The above solution exploits use of the fact that the XOR is associative. We now describe another solution, with a much smaller lookup table. It makes use of the property that the order in which we perform XOR does not change the result, e.g., XOR commutes, to use the CPU's XOR instruction which operates in parallel on words.

For example, the parity of $\langle b_{63}, b_{62}, \ldots, b_3, b_2, b_1, b_0 \rangle$ equals the XOR of the parity of $\langle b_{63}, b_{62}, \ldots, b_{32} \rangle$ with the parity of $\langle b_{31}, b_{30}, \ldots, b_0 \rangle$. The XOR of these two 32-bit values can be computed with a single shift and a single 32-bit XOR instruction. Conceptually, the final step in the algorithm below entails a lookup into a table indexed by a 4-bit quantity. We could instead perform two more shift and XOR steps, and completely avoided a lookup-table.

```
short Parity(unsigned long x) {
  x ^= x >> 32;
  x ^= x >> 16;
  x ^= x >> 8;
  x ^= x >> 4;
  x &= 0xf; // Only wants the last 4 bits of x.
  // Return the LSB, which is the parity.
```

```
  return FourBitParityLookup(x) & 1;
}

// The LSB of kFourBitParityLookupTable is the parity of 0,
// next bit is parity of 1, followed by the parity of 2, etc.

const int kFourBitParityLookupTable = 0x6996; // = 0b0110100110010110.

short FourBitParityLookup(int x) {
  return kFourBitParityLookupTable >> x;
}
```

Problem 1.2, *pg.* 2: *Write a function that performs base conversion. Specifically, the input is an integer base b_1, a string s, representing an integer x in base b_1, and another integer base b_2; the output is the string representing the integer x in base b_2. Assume $2 \leq b_1, b_2 \leq 16$. Use "A" to represent 10, "B" for 11, ..., and "F" for 15.*

Solution 1.2: We can convert the base-b_1 string s to a variable x of integer type, and then convert x to a base-b_2 string a. For example, if the string is "615", $b_1 = 7$ and $b_2 = 13$, then in decimal x is 306, and the final result is "1A7".

```
string ConvertBase(const string& s, int b1, int b2) {
  bool is_negative = s.front() == '-';
  int x = 0;
  for (size_t i = (is_negative == true ? 1 : 0); i < s.size(); ++i) {
    x *= b1;
    x += isdigit(s[i]) ? s[i] - '0' : s[i] - 'A' + 10;
  }

  string result;
  do {
    int reminder = x % b2;
    result.push_back(reminder >= 10 ? 'A' + reminder - 10 : '0' + reminder);
    x /= b2;
  } while (x);

  if (is_negative) {  // s is a negative number.
    result.push_back('-');
  }
  reverse(result.begin(), result.end());
  return result;
}
```

The time complexity is $O(n(1 + \log_{b_2} b_1))$, where n is the length of s. The reasoning is as follows. First, we perform n multiply-and-adds to get x from s. Then we perform $\log_{b_2} x$ multiply and adds to get the result. The value x is upper-bounded by ${b_1}^n$, and $\log_{b_2}({b_1}^n) = n \log_{b_2} b_1$.

Problem 2.1, *pg.* 4: *Design an algorithm that takes a sequence of n three-dimensional coordinates to be traversed, and returns the minimum battery capacity needed to complete the journey. The robot begins with the battery fully charged.*

Solution 2.1: Suppose the three dimensions correspond to the X, Y, and Z axes, with z being the vertical dimension. Since energy usage depends on the change in altitude of the robot, we can ignore the x and y coordinates. Suppose the points where the robot goes in successive order have z coordinates $z_0, \ldots, z_{n-1}$.

Assume that the battery capacity is such that with the fully charged battery, the robot can climb B meters. The robot will run out of energy if and only if there are points i and j such that to go from Point i to Point j, the robot has to climb more than B meters. Therefore, we would like to pick the smallest B such that for any $i < j$, we have $B \geq z_j - z_i$.

Here is an implementation.

```
int FindBatteryCapacity(const vector<int>& h) {
  int min_height = numeric_limits<int>::max(), capacity = 0;
  for (const int &height : h) {
    capacity = max(capacity, height - min_height);
    min_height = min(min_height, height);
  }
  return capacity;
}
```

Variant 2.1.1: Let A be an array of integers. Find the length of a longest subarray all of whose entries are equal.

Problem 3.1, *pg. 5*: *Implement string/integer inter-conversion functions.*

Solution 3.1: Let's consider the integer to string problem first. If the number to convert is a single digit, i.e., it is between 0 and 9, the result is easy to compute: it is the string consisting of the single character encoding that digit.

If the number has more than one digit, it is natural to perform the conversion digit-by-digit. The key insight is that for any positive integer x, the least significant digit in the decimal representation of x is $x \mod 10$, and the remaining digits are $x/10$. This approach computes the digits in reverse order, e.g., if we begin with 423, we get 3 and are left with 42 to convert. Then we get 2, and are left with 4 to convert. Finally, we get 4 and there are no digits to convert. There are multiple ways in which we can form a string from the reverse order of the digits, but a particularly simple and memory-efficient way is to add the digits to the end of a string, and eventually reverse it.

If x is negative, we record that, negate x, and then add a '-' before reversing. If x is 0, our code breaks out of the iteration without writing any digits, in which case we need to explicitly set a 0.

To convert from a string to an integer we recall the basic working of a positional number system. A base-10 number $d_2d_1d_0$ encodes the number $10^2 \times d_2 + 10^1 \times d_1 + d_0$. A brute-force algorithm then is to begin with the rightmost digit, and iteratively add $10^i \times d_i$ to a cumulative sum. The efficient way to compute 10^{i+1} is to use the existing value 10^i and multiply that by 10.

A more elegant solution is to begin from the leftmost digit and with each succeeding digit, multiply the partial result by 10 and add that digit. For example, to convert "314" to an integer, we initial the partial result r to 0. In the first iteration, $r = 3$, in the second iteration $r = 3 \times 10 + 1 = 31$, and in the third iteration $r = 31 \times 10 + 4 = 314$, which is the final result.

Negative numbers are handled by recording the sign and negating the result.

```
string IntToString(int x) {
  bool is_negative = false;
  if (x < 0) {
    x = -x, is_negative = true;
  }

  string s;
  while (x) {
    s.push_back('0' + x % 10);
    x /= 10;
  }
  if (s.empty()) {
    return {"0"};  // x is 0.
  }

  if (is_negative) {
    s.push_back('-');  // Adds the negative sign back.
  }
  reverse(s.begin(), s.end());
  return s;
}

int StringToInt(const string& s) {
  bool is_negative = s[0] == '-';
  int r = 0;
  for (int i = s[0] == '-' ? 1 : 0; i < s.size(); ++i) {
    int digit = s[i] - '0';
    r = r * 10 + digit;
  }
  return is_negative ? -r : r;
}
```

Problem 3.2, *pg. 5*: *Implement a function for reversing the words in a string s.*

Solution 3.2: The code for computing the position for each character in the final result in a single pass is intricate.

However, for the special case where each word is a single character, the desired result is simply the reverse of s.

For the general case, reversing s gets the words to their correct relative positions. However, for words that are longer than one character, their letters appear in reverse order. This situation can be corrected by reversing the individual words.

For example, "ram is costly" reversed yields "yltsoc si mar". We obtain the final result by reversing each word to obtain "costly is ram".

```
void ReverseWords(string* s) {
  // Reverses the whole string first.
  reverse(s->begin(), s->end());

  size_t start = 0, end;
  while ((end = s->find(" ", start)) != string::npos) {
    // Reverses each word in the string.
    reverse(s->begin() + start, s->begin() + end);
    start = end + 1;
  }
  // Reverses the last word.
  reverse(s->begin() + start, s->end());
}
```

Since we spend $O(1)$ per character, the time complexity is $O(n)$, where n is the length of s.

Problem 4.1, *pg. 7*: *Given a reference to the head of a singly linked list, how would you determine whether the list ends in a* `null` *or reaches a cycle of nodes? Write a function that returns* `null` *if there does not exist a cycle, and the reference to the start of the cycle if a cycle is present. (You do not know the length of the list in advance.)*

Solution 4.1: This problem has several solutions. If space is not an issue, the simplest approach is to explore nodes via the next field starting from the head and storing visited nodes in a hash table—a cycle exists if and only if we visit a node already in the hash table. If no cycle exists, the search ends at the tail (often represented by having the next field set to `null`). This solution requires $O(n)$ space, where n is the number of nodes in the list.

A brute-force approach that does not use additional storage and does not modify the list is to traverse the list in two loops—the outer loop traverses the nodes one-by-one, and the inner loop starts from the head, and traverses as many nodes as the outer loop has gone through so far. If the node being visited by the outer loop is visited twice, a loop has been detected. (If the outer loop encounters the end of the list, no cycle exists.) This approach has $O(n^2)$ time complexity.

This idea can be made to work in linear time—use a slow iterator and a fast iterator to traverse the list. In each iteration, advance the slow iterator by one and the fast iterator by two. The list has a cycle if and only if the two iterators meet. The reasoning is as follows: if the fast iterator jumps over the slow iterator, the slow iterator will equal the fast iterator in the next step.

Now, assuming that we have detected a cycle using the above method, we can find the start of the cycle, by first calculating the cycle length C. Once we know there is a cycle, and we have a node on it, it is trivial to compute the cycle length. To find the first node on the cycle, we use two iterators, one of which is C ahead of the other. We advance them in tandem, and when they meet, that node must be the first node on the cycle.

The code to do this traversal is quite simple:

```
shared_ptr<ListNode<int>> HasCycle(const shared_ptr<ListNode<int>>& head) {
```

```
  shared_ptr<ListNode<int>> fast = head, slow = head;

  while (fast && fast->next && fast->next->next) {
    slow = slow->next, fast = fast->next->next;
    if (slow == fast) {
      // There is a cycle, so now let's calculate the cycle length.
      int cycle_len = 0;
      do {
        ++cycle_len;
        fast = fast->next;
      } while (slow != fast);

      // Finds the start of the cycle.
      auto cycle_len_advanced_iter = head;
      while (cycle_len--) {
        cycle_len_advanced_iter = cycle_len_advanced_iter->next;
      }

      auto iter = head;
      // Both iterators advance in tandem.
      while (iter != cycle_len_advanced_iter) {
        iter = iter->next;
        cycle_len_advanced_iter = cycle_len_advanced_iter->next;
      }
      return iter;  // iter is the start of cycle.
    }
  }
  return nullptr;  // No cycle.
}
```

Let F be the number of nodes to the start of the cycle, C the number of nodes on the cycle, and n the total number of nodes. Then the time complexity is $O(F) + O(C) = O(n)$—$O(F)$ for both pointers to reach the cycle, and $O(C)$ for them to overlap once the slower one enters the cycle.

Variant 4.1.1: The following program purports to compute the beginning of the cycle without determining the length of the cycle; it has the benefit of being more succinct than the code listed above. Is the program correct?

```
shared_ptr<ListNode<int>> HasCycle(const shared_ptr<ListNode<int>>& head) {
  shared_ptr<ListNode<int>> fast = head, slow = head;

  while (fast && fast->next && fast->next->next) {
    slow = slow->next, fast = fast->next->next;
    if (slow == fast) {  // There is a cycle.
      // Tries to find the start of the cycle.
      slow = head;
      // Both pointers advance at the same time.
      while (slow != fast) {
        slow = slow->next, fast = fast->next;
      }
      return slow;  // slow is the start of cycle.
    }
```

```
  }
  return nullptr;  // No cycle.
}
```

Problem 5.1, *pg. 8*: *Design a stack that supports a* `max` *operation, which returns the maximum value stored in the stack.*

Solution 5.1: The simplest way to implement `max()` is to consider each element in the stack, e.g., by iterating through the underlying array for an array-based stack. The time complexity is $O(n)$ and the space complexity is $O(1)$, where n is the number of elements currently in the stack.

The time complexity can be reduced to $O(\log n)$ using auxiliary data structures, specifically, a heap or a BST, and a hash table. The space complexity increases to $O(n)$ and the code is quite complex.

Suppose we use a single auxiliary variable, M, to record the element that is maximum in the stack. Updating M on pushes is easy: $M = \max(M, e)$, where e is the element being pushed. However, updating M on pop is very time consuming. If M is the element being popped, we have no way of knowing what the maximum remaining element is, and are forced to consider all the remaining elements.

We can dramatically improve on the time complexity of popping by caching, in essence, trading time for space. Specifically, for each entry in the stack, we cache the maximum stored at or below that entry. Now when we pop, we evict the corresponding cached value.

```
class Stack {
 public:
  bool Empty() const { return element_with_cached_max_.empty(); }

  int Max() const {
    if (!Empty()) {
      return element_with_cached_max_.top().second;
    }
    throw length_error("Max(): empty stack");
  }

  int Pop() {
    if (Empty()) {
      throw length_error("Pop(): empty stack");
    }
    int pop_element = element_with_cached_max_.top().first;
    element_with_cached_max_.pop();
    return pop_element;
  }

  void Push(int x) {
    element_with_cached_max_.emplace(
        x, std::max(x, Empty() ? x : element_with_cached_max_.top().second));
  }

 private:
```

```
  // Stores (element, cached maximum) pair.
  stack<pair<int, int>> element_with_cached_max_;
};
```

Each of the specified methods has time complexity $O(1)$. The additional space complexity is $O(n)$, regardless of the stored keys.

We can improve on the best-case space needed by observing that if an element e being pushed is smaller than the maximum element already in the stack, then e can never be the maximum, so we do not need to record it. We cannot store the sequence of maximum values in a separate stack because of the possibility of duplicates. We resolve this by additionally recording the number of occurrences of each maximum value. See Figure 17.1 on the following page for an example.

```
class Stack {
 public:
  bool Empty() const { return element_.empty(); }

  int Max() const {
    if (!Empty()) {
      return cached_max_with_count_.top().first;
    }
    throw length_error("Max(): empty stack");
  }

  int Pop() {
    if (Empty()) {
      throw length_error("Pop(): empty stack");
    }
    int pop_element = element_.top();
    element_.pop();
    const int kCurrentMax = cached_max_with_count_.top().first;
    if (pop_element == kCurrentMax) {
      int& max_frequency = cached_max_with_count_.top.second;
      --max_frequency;
      if (max_frequency == 0) {
        cached_max_with_count_.pop();
      }
    }
    return pop_element;
  }

  void Push(int x) {
    element_.emplace(x);
    if (cached_max_with_count_.empty()) {
      cached_max_with_count_.emplace(x, 1);
    } else {
      const int kCurrentMax = cached_max_with_count_.top().first;
      if (x == kCurrentMax) {
        int& max_frequency = cached_max_with_count_.top.second;
        ++max_frequency;
      } else if (x > kCurrentMax) {
        cached_max_with_count_.emplace(x, 1);
      }
    }
```

```
42   }
43
44  private:
45   stack<int> element_;
46   // Stores (maximum value, count) pair.
47   stack<pair<int, int>> cached_max_with_count_;
48 };
```

The worst-case additional space complexity is $O(n)$, which occurs when each key pushed is greater than all keys in the primary stack. However, when the number of distinct keys is small, or the maximum changes infrequently, the additional space complexity is less, $O(1)$ in the best case. The time complexity for each specified method is still $O(1)$.

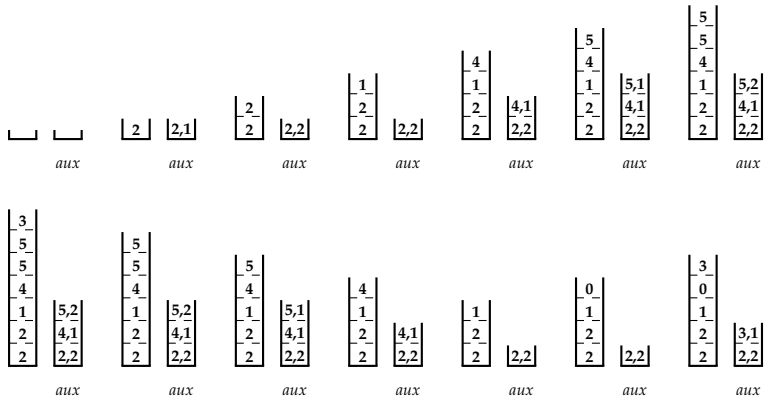


Figure 17.1: The primary and auxiliary stacks for the following operations: `push(2)`, `push(2)`, `push(1)`, `push(4)`, `push(5)`, `push(5)`, `push(3)`, `pop()`, `pop()`, `pop()`, `pop()`, `push(0)`, `push(3)`. Both stacks are initially empty, and their progression is shown from left-to-right, then top-to-bottom. The top of the auxiliary stack holds the maximum element in the stack, and the number of times that element occurs in the stack. The auxiliary stack is denoted by *aux*.

Problem 5.2, *pg. 9*: *Given the root of a binary tree, print all the keys at the root and its descendants. The keys should be printed in the order of the corresponding nodes' depths. For example, you could print*

314
6 6
271 561 2 271
28 0 3 1 28
17 401 257
641

for the binary tree in Figure 6.1 on Page 10.

Solution 5.2: The brute-force approach is to keep an array A of lists. The list at $A[i]$ is the set of nodes at depth i. We initialize $A[0]$ to the root. To get $A[i+1]$ from $A[i]$,

we traverse $A[i]$ and add children to $A[i+1]$. The time and space complexities are both $O(n)$, where n is the number of nodes in the tree.

We can improve on the space complexity by recycling space. Specifically, we do not need $A[i]$ after $A[i+1]$ is computed, i.e., two lists suffice.

As an alternative, we can maintain a queue of nodes to process. Specifically the queue contains nodes at depth i followed by nodes at depth $i+1$. After all nodes at depth i are processed, the head of the queue is a node at depth $i+1$; processing this node introduces nodes from depth $i+2$ to the end of the queue.

We use a count variable that records the number of nodes at the depth of the head of the queue that remain to be processed. When all nodes at depth i are processed, the queue consists of exactly the set of nodes at depth $i+1$, and the count is updated to the size of the queue.

```
void PrintBinaryTreeDepthOrder(const unique_ptr<BinaryTreeNode<int>>& root) {
  queue<BinaryTreeNode<int>*> processing_nodes;
  processing_nodes.emplace(root.get());
  size_t num_nodes_current_level = processing_nodes.size();
  while (!processing_nodes.empty()) {
    auto curr = processing_nodes.front();
    processing_nodes.pop();
    --num_nodes_current_level;
    if (!curr) {
      continue;
    }
    cout << curr->data << ' ';

    // Defer the null checks to the null test above.
    processing_nodes.emplace(curr->left.get());
    processing_nodes.emplace(curr->right.get());
    // Done with the nodes at the current depth.
    if (!num_nodes_current_level) {
      cout << endl;
      num_nodes_current_level = processing_nodes.size();
    }
  }
}
```

Since each node is enqueued and dequeued exactly once, the time complexity is $O(n)$. The space complexity is $O(m)$, where m is the maximum number of nodes at any single depth.

Variant 5.2.1: Write a function which takes as input a binary tree where each node is labeled with an integer and prints all the node keys in top down, alternating left-to-right and right-to-left order, starting from left-to-right. For example, you should print

```
314
6 6
271 561 2 271
28 1 3 0 28
```

```
17 401 257
641
```
for the binary tree in Figure 6.1 on Page 10.

Variant 5.2.2: Write a function which takes as input a binary tree where each node is labeled with an integer and prints all the node keys in a bottom up, left-to-right order. For example, if the input is the tree in Figure 6.1 on Page 10, your function should print
```
641
17 401 257
28 0 3 1 28
271 561 2 271
6 6
314
```

Problem 6.1, *pg. 12*: *Write a function that takes as input the root of a binary tree and checks whether the tree is balanced.*

Solution 6.1:

Here is a brute-force algorithm. Compute the height for the tree rooted at each node x recursively. The basic computation is to compute the height for each node starting from the leaves, and proceeding upwards. For each node, we check if the difference in heights of the left and right children is greater than one. We can store the heights in a hash table, or in a new field in the nodes. This entails $O(n)$ storage and $O(n)$ time, where n is the number of nodes of the tree.

We can solve this problem using less storage by observing that we do not need to store the heights of all nodes at the same time. Once we are done with a subtree, all we need is whether it is balanced, and if so, what its height is—we do not need any information about descendants of the subtree's root.

```
1  bool IsBalancedBinaryTree(const unique_ptr<BinaryTreeNode<int>>& tree) {
2    return CheckBalanced(tree).first;
3  }
4
5  // First value of the return value indicates if tree is balanced, and if
6  // balanced the second value of the return value is the height of tree.
7  pair<bool, int> CheckBalanced(const unique_ptr<BinaryTreeNode<int>>& tree) {
8    if (tree == nullptr) {
9      return {true, -1};  // Base case.
10   }
11
12   auto left_result = CheckBalanced(tree->left);
13   if (!left_result.first) {
14     return {false, 0};  // Left subtree is not balanced.
15   }
16   auto right_result = CheckBalanced(tree->right);
17   if (!right_result.first) {
18     return {false, 0};  // Right subtree is not balanced.
19   }
```

```
20
21   bool is_balanced = abs(left_result.second - right_result.second) <= 1;
22   int height = max(left_result.second, right_result.second) + 1;
23   return {is_balanced, height};
24 }
```

The program implements a postorder traversal with some calls possibly being eliminated because of early termination. Specifically, if any left subtree is unbalanced we do not need to visit the corresponding right subtree. The function call stack corresponds to a sequence of calls from the root through the unique path to the current node, and the stack height is therefore bounded by the height of the tree, leading to an $O(h)$ space bound. The time complexity is the same as that for a postorder traversal, namely $O(n)$.

Variant 6.1.1: Write a function that returns the size of the largest subtree that is complete.

Problem 7.1, *pg. 13*: *Design an algorithm that takes a set of files containing stock trades sorted by increasing trade times, and writes a single file containing the trades appearing in the individual files sorted in the same order. The algorithm should use very little RAM, ideally of the order of a few kilobytes.*

Solution 7.1: In the abstract, we are trying to merge k sequences sorted in increasing order. One way to do this is to repeatedly pick the smallest element amongst the smallest remaining elements of each of the k sequences. A min-heap is ideal for maintaining a set of elements when we need to insert arbitrary values, as well as query for the smallest element. There are no more than k elements in the min-heap. Both extract-min and insert take $O(\log k)$ time. Hence, we can do the merge in $O(n \log k)$ time, where n is the total number of elements in the input. The space complexity is $O(k)$ beyond the space needed to write the final result. The implementation is given below. Note that for each element we need to store the sequence it came from. For ease of exposition, we show how to merge sorted arrays, rather than files. The only difference is that for the file case we do not need to explicitly maintain an index for next unprocessed element in each sequence—the file I/O library tracks the first unread entry in the file.

```
1 struct Compare {
2   bool operator()(const pair<int, int>& lhs, const pair<int, int>& rhs) {
3     return lhs.first > rhs.first;
4   }
5 };
6
7 vector<int> MergeArrays(const vector<vector<int>>& S) {
8   priority_queue<pair<int, int>, vector<pair<int, int>>, Compare> min_heap;
9   vector<int> S_idx(S.size(), 0);
10
11  // Every array in S puts its smallest element in heap.
12  for (int i = 0; i < S.size(); ++i) {
13    if (S[i].size() > 0) {
```

```
      min_heap.emplace(S[i][0], i);
      S_idx[i] = 1;
    }
  }

  vector<int> ret;
  while (!min_heap.empty()) {
    pair<int, int> p = min_heap.top();
    ret.emplace_back(p.first);
    // Add the smallest element into heap if possible.
    if (S_idx[p.second] < S[p.second].size()) {
      min_heap.emplace(S[p.second][S_idx[p.second]++], p.second);
    }
    min_heap.pop();
  }
  return ret;
}
```

Problem 8.1, *pg. 17*: *Write a method that takes a sorted array A and a key k and returns the index of the* first *occurrence of k in A. Return -1 if k does not appear in A. For example, when applied to the array in Figure 8.1 on Page 17 your algorithm should return 3 if $k = 108$; if $k = 285$, your algorithm should return 6.*

Solution 8.1: The key idea is to search for k. However, even if we find k, after recording this we continue the search on the left subarray.

```
int SearchFirst(const vector<int>& A, int k) {
  int left = 0, right = A.size() - 1, result = -1;
  while (left <= right) {
    int mid = left + ((right - left) / 2);
    if (A[mid] > k) {
      right = mid - 1;
    } else if (A[mid] == k) {
      // Records the solution and keep searching the left part.
      result = mid, right = mid - 1;
    } else {  // A[mid] < k.
      left = mid + 1;
    }
  }
  return result;
}
```

The complexity bound is still $O(\log n)$—this is because each iteration reduces the size of the subarray being searched by half.

Variant 8.1.1: Let A be an unsorted array of n integers, with $A[0] \geq A[1]$ and $A[n-2] \leq A[n-1]$. Call an index i a *local minimum* if $A[i]$ is less than or equal to its neighbors. How would you efficiently find a local minimum, if one exists?

Variant 8.1.2: A sequence is said to be ascending if each element is greater than or equal to its predecessor; a descending sequence is one in which each element is less than or equal to its predecessor. A sequence is strictly ascending if each element is greater than its predecessor. Suppose it is known that an array A consists of an ascending sequence followed by a descending sequence. Design an algorithm for finding the maximum element in A. Solve the same problem when A consists of a strictly ascending sequence, followed by a descending sequence.

Problem 9.1, *pg. 19*: *You are required to write a method which takes an anonymous letter L and text from a magazine M. Your method is to return* `true` *if and only if L can be written using M, i.e., if a letter appears k times in L, it must appear at least k times in M.*

Solution 9.1: Assume the string encoding the magazine is as long as the string encoding the letter. (If not, the answer is `false`.) We build a hash table H_L for L, where each key is a character in the letter and its value is the number of times it appears in the letter. Consequently, we scan the magazine character-by-character. When processing c, if c appears in H_L, we reduce its frequency count by 1; we remove it from H_L when its count goes to zero. If H_L becomes empty, we return `true`. If it is nonempty when we get to the end of M, we return `false`.

```
1  bool AnonymousLetter(const string& L, const string& M) {
2    unordered_map<char, int> hash;
3    // Inserts all chars in L into a hash table.
4    for_each(L.begin(), L.end(), [&hash](const char & c) { ++hash[c]; });
5
6    // Checks characters in M that could cover characters in a hash table.
7    for (const char& c : M) {
8      auto it = hash.find(c);
9      if (it != hash.cend()) {
10       if (--it->second == 0) {
11         hash.erase(it);
12         if (hash.empty()) {
13           return true;
14         }
15       }
16     }
17   }
18   // No entry in hash means L can be covered by M.
19   return hash.empty();
20 }
```

In the worst case, the letter is not constructible or the last character of M is essentially required. Therefore, the time complexity is $O(n_M)$ where n_M is the length of M The space complexity is $O(c_L)$, where c_L is the number of distinct characters appearing L.

If the characters are coded in ASCII, we could do away with H_L and use a 256 entry integer array A, with $A[i]$ being set to the number of times the character i appears in the letter.

Problem 10.1, *pg. 21*: *Given sorted arrays A and B of lengths n and m respectively, return an array C containing elements common to A and B. The array C should be free of duplicates.*

How would you perform this intersection if—(1.) $n \approx m$ and (2.) $n \ll m$?

Solution 10.1: The brute-force algorithm is a "loop join", i.e., traversing through all the elements of one array and comparing them to the elements of the other array. This has $O(mn)$ time complexity, regardless of whether the arrays are sorted or unsorted:

```
vector<int> IntersectTwoSortedArrays(const vector<int>& A,
                                     const vector<int>& B) {
  vector<int> intersect;
  for (int i = 0; i < A.size(); ++i) {
    if (i == 0 || A[i] != A[i - 1]) {
      for (int b : B) {
        if (A[i] == b) {
          intersect.emplace_back(A[i]);
          break;
        }
      }
    }
  }
  return intersect;
}
```

However, since both the arrays are sorted, we can make some optimizations. First, we can scan array A and use binary search in array B, find whether the element is present in B.

```
vector<int> IntersectTwoSortedArrays(const vector<int>& A,
                                     const vector<int>& B) {
  vector<int> intersect;
  for (int i = 0; i < A.size(); ++i) {
    if ((i == 0 || A[i] != A[i - 1]) &&
        binary_search(B.cbegin(), B.cend(), A[i])) {
      intersect.emplace_back(A[i]);
    }
  }
  return intersect;
}
```

The time complexity is $O(n \log m)$.

We can further improve our run time by choosing the shorter array for the outer loop since if $n \ll m$ then $m \log(n) \gg n \log(m)$.

This is the best solution if one set is much smaller than the other. However, it is not the best when the array lengths are similar because we are not exploiting the fact that both arrays are sorted. In this case, iterating in tandem through the elements of each array in increasing order will work best as shown in this code.

```
vector<int> IntersectTwoSortedArrays(const vector<int>& A,
                                     const vector<int>& B) {
  vector<int> intersect;
  int i = 0, j = 0;
  while (i < A.size() && j < B.size()) {
    if (A[i] == B[j] && (i == 0 || A[i] != A[i - 1])) {
      intersect.emplace_back(A[i]);
```

```
      ++i, ++j;
    } else if (A[i] < B[j]) {
      ++i;
    } else {  // A[i] > B[j].
      ++j;
    }
  }
  return intersect;
}
```

The run time for this algorithm is $O(m + n)$.

Problem 10.2, *pg. 21*: *Given a set of n events, how would you determine the maximum number of events that take place concurrently?*

Solution 10.2: Each event corresponds to an interval $[b, e]$; let b and e be the earliest starting time and last ending time. Define the function $c(t)$ for $t \in [b, e]$ to be the number of intervals containing t. Observe that $c(\tau)$ does not change if τ is not the starting or ending time of an event.

This leads to an $O(n^2)$ brute-force algorithm: for each interval, for each of its two endpoints, determining how many intervals contain that point. The total number of endpoints is $2n$ and each check takes $O(n)$ time, since checking whether an interval $[b_i, e_i]$ contains a point t takes $O(1)$ time (simply check if $b_i \leq t \leq e_i$).

We can improve the run time to $O(n \log n)$ by sorting the set of all the endpoints in ascending order. If two endpoints have equal times, and one is a start time and the other is an end time, the one corresponding to a start time comes first. (If both are start or finish times, we break ties arbitrarily.)

We initialize a counter to 0, and iterate through the sorted sequence S from smallest to largest. For each endpoint that is the start of an interval, we increment the counter by 1, and for each endpoint that is the end of an interval, we decrement the counter by 1. The maximum value attained by the counter is maximum number of overlapping intervals.

```
struct Interval {
  int start, finish;
};

struct Endpoint {
  bool operator<(const Endpoint& e) const {
    return time != e.time ? time < e.time : (isStart && !e.isStart);
  }

  int time;
  bool isStart;
};

int FindMaxConcurrentEvents(const vector<Interval>& A) {
  // Builds the endpoint array.
  vector<Endpoint> E;
  for (const Interval& i : A) {
    E.emplace_back(Endpoint{i.start, true});
```

```
    E.emplace_back(Endpoint{i.finish, false});
  }
  // Sorts the endpoint array according to the time.
  sort(E.begin(), E.end());

  // Finds the maximum number of events overlapped.
  int max_count = 0, count = 0;
  for (const Endpoint& e : E) {
    if (e.isStart) {
      max_count = max(++count, max_count);
    } else {
      --count;
    }
  }
  return max_count;
}
```

Sorting the endpoint array takes $O(n \log n)$ time; iterating through the sorted array takes $O(n)$ time, yielding an $O(n \log n)$ time complexity. The space complexity is $O(n)$, which is the size of the endpoint array.

Variant 10.2.1: Users $1, 2, \ldots, n$ share an Internet connection. User i uses b_i bandwidth from time s_i to f_i, inclusive. What is the peak bandwidth usage?

Problem 11.1, *pg.* 23: *Write a function that takes as input the root of a binary tree whose nodes have a key field, and returns* `true` *if and only if the tree satisfies the BST property.*

Solution 11.1: Several solutions exist, which differ in terms of their space and time complexity, and the effort needed to code them.

The simplest is to start with the root r, and compute the maximum key *r.left.max* stored in the root's left subtree, and the minimum key *r.right.min* in the root's right subtree. Then we check that the key at the root is greater than or equal to *r.right.min* and less than or equal to *r.left.max*. If these checks pass, we continue checking the root's left and right subtree recursively.

Computing the minimum key in a binary tree is straightforward: we compare the key stored at the root with the minimum key stored in its left subtree and with the minimum key stored in its right subtree. The maximum key is computed similarly. (Note that the minimum may be in either subtree, since the tree may not satisfy the BST property.)

The problem with this approach is that it will repeatedly traverse subtrees. In a worst case, when the tree is BST and each node's left child is empty, its complexity is $O(n^2)$, where n is the number of nodes. The complexity can be improved to $O(n)$ by caching the largest and smallest keys at each node; this requires $O(n)$ additional storage.

We now present two approaches which have $O(n)$ time complexity and $O(h)$ additional space complexity.

The first, more straightforward approach, is to check constraints on the values for each subtree. The initial constraint comes from the root. Each node in its left (right)

child must have a value less than or equal (greater than or equal) to the value at the root. This idea generalizes: if all nodes in a tree rooted at t must have values in the range $[l, u]$, and the value at t is $w \in [l, u]$, then all values in the left subtree of t must be in the range $[l, w]$, and all values stored in the right subtree of t must be in the range $[w, u]$. The code below uses this approach.

```
bool IsBST(const unique_ptr<BinaryTreeNode<int>>& root) {
  return IsBSTHelper(root, numeric_limits<int>::min(),
                     numeric_limits<int>::max());
}

bool IsBSTHelper(const unique_ptr<BinaryTreeNode<int>>& root, int lower,
                 int upper) {
  if (!root) {
    return true;
  } else if (root->data < lower || root->data > upper) {
    return false;
  }

  return IsBSTHelper(root->left, lower, root->data) &&
         IsBSTHelper(root->right, root->data, upper);
}
```

The second approach is to perform an inorder traversal, and record the value stored at the last visited node. Each time a new node is visited, its value is compared with the value of the previous visited node; if at any step, the value at the previously visited node is greater than the node currently being visited, we have a violation of the BST property. In principle, this approach can use the existence of an $O(1)$ space complexity inorder traversal to further reduce the space complexity.

```
bool IsBST(const unique_ptr<BinaryTreeNode<int>>& root) {
  auto* n = root.get();
  // Stores the value of previous visited node.
  int last = numeric_limits<int>::min();
  bool result = true;

  while (n) {
    if (n->left.get()) {
      // Finds the predecessor of n.
      auto* pre = n->left.get();
      while (pre->right.get() && pre->right.get() != n) {
        pre = pre->right.get();
      }

      // Processes the successor link.
      if (pre->right.get()) {  // pre->right == n.
        // Reverts the successor link if predecessor's successor is n.
        pre->right.release();
        if (last > n->data) {
          result = false;
        }
        last = n->data;
        n = n->right.get();
```

```
      } else {  // If predecessor's successor is not n.
        pre->right.reset(n);
        n = n->left.get();
      }
    } else {
      if (last > n->data) {
        result = false;
      }
      last = n->data;
      n = n->right.get();
    }
  }
  return result;
}
```

The approaches outlined above all explore the left subtree first. Therefore, even if the BST property does not hold at a node which is close to the root (e.g., the key stored at the right child is less than the key stored at the root), their time complexity is still $O(n)$.

We can search for violations of the BST property in a BFS manner to reduce the time complexity when the property is violated at a node whose depth is small, specifically much less than n.

The code below uses a queue to process nodes. Each queue entry contains a node, as well as an upper and a lower bound on the keys stored at the subtree rooted at that node. The queue is initialized to the root, with lower bound $-\infty$ and upper bound $+\infty$.

Suppose an entry with node n, lower bound l and upper bound u is popped. If n's left child is not `null`, a new entry consisting of *n.left*, upper bound *n.key* and lower bound l is added. A symmetric entry is added if n's right child is not `null`. When adding entries, we check that the node's key lies in the range specified by the lower bound and the upper bound; if not, we return immediately reporting a failure.

We claim that if the BST property is violated in the subtree consisting of nodes at depth d or less, it will be discovered without visiting any nodes at depths $d + 1$ or more. This is because each time we enqueue an entry, the lower and upper bounds on the node's key are the tightest possible. A formal proof of this is by induction; intuitively, it is because we satisfy all the BST requirements induced by the search path to that node.

```
struct QNode {
  const unique_ptr<BinaryTreeNode<int>>& node;
  int lower, upper;
};

bool IsBST(const unique_ptr<BinaryTreeNode<int>>& root) {
  queue<QNode> q;
  q.emplace(
      QNode{root, numeric_limits<int>::min(), numeric_limits<int>::max()});

  while (!q.empty()) {
    if (q.front().node.get()) {
```

```
      if (q.front().node->data < q.front().lower ||
          q.front().node->data > q.front().upper) {
        return false;
      }

      q.emplace(QNode{q.front().node->left, q.front().lower,
                      q.front().node->data});
      q.emplace(QNode{q.front().node->right, q.front().node->data,
                      q.front().upper});
    }
    q.pop();
  }
  return true;
}
```

Problem 12.1, *pg. 25*: *Implement a method that takes as input a set S of n distinct elements, and prints the power set of S. Print the subsets one per line, with elements separated by commas.*

Solution 12.1: One way to solve this problem is realizing that for a given ordering of the elements of S, there exists a one-to-one correspondence between the 2^n bit arrays of length n and the set of all subsets of S—the 1s in the n-length bit array v indicate the elements of S in the subset corresponding to v.

For example, if $S = \{g, l, e\}$ and the elements are ordered $g < l < e$, the bit array $\langle 0, 1, 1 \rangle$ denotes the subset $\{l, e\}$.

If n is less than or equal to the width of an integer on the architecture (or language) we are working on, we can enumerate bit arrays by enumerating integers in $[0, 2^n - 1]$ and examining the indices of bits set in these integers. These indices are determined by first isolating the lowest set bit by computing $y = x\&\sim(x-1)$, and then getting the index by computing $\lg y$.

```
void GeneratePowerSet(const vector<int>& S) {
  for (int i = 0; i < (1 << S.size()); ++i) {
    int x = i;
    while (x) {
      int tar = log2(x & ~(x - 1));
      cout << S[tar];
      if (x &= x - 1) {
        cout << ',';
      }
    }
    cout << endl;
  }
}
```

Since each set takes $O(n)$ time to print, the time complexity is $O(n2^n)$. In practice, it would likely be faster to iterate through all the bits in x, one at a time.

Alternately, we can use recursion. The most natural recursive algorithm entails recursively computing all subsets U of $S - \{x\}$, i.e., S without the element x (which could be any element), and then adding x to each subset in U to create the set of

subsets V. (The base case is when S is empty, in which case we return $\{\{\}\}$.) The subsets in U are distinct from those in V, and the final result is just $U \cup V$, which we can then print. The number of recursive calls for a set of n elements, $T(n)$ satisfies $T(n) = 2T(n-1)$, with $T(0) = 1$, which solves to $T(n) = 2^n$. Since each call takes time $O(n)$, the total time complexity is $O(n2^n)$.

The problem with the approach just described is that it uses $O(n2^n)$ space. The fact that we only need to print the subsets, and not return them, suggests that a more space optimal approach would be to compute the subsets incrementally. Conceptually, the algorithm shown below passes two additional parameters, m and `subset`; the latter indicates which of the first m elements of S must be part of the subsets begin created from the remaining $n - m$ elements. It iteratively prints `subset`, and then prints all subsets of the remaining elements, with and without the $(m+1)$-th element.

```
1  void GeneratePowerSet(const vector<int>& S) {
2    vector<int> subset;
3    GeneratePowerSetHelper(S, 0, &subset);
4  }
5
6  void GeneratePowerSetHelper(const vector<int>& S, int m,
7                              vector<int>* subset) {
8    if (!subset->empty()) {
9      // Print the subset.
10     cout << subset->front();
11     for (int i = 1; i < subset->size(); ++i) {
12       cout << "," << (*subset)[i];
13     }
14   }
15   cout << endl;
16
17   for (int i = m; i < S.size(); ++i) {
18     subset->emplace_back(S[i]);
19     GeneratePowerSetHelper(S, i + 1, subset);
20     subset->pop_back();
21   }
22 }
```

The number of recursive calls, $T(n)$ satisfies the recurrence $T(n) = T(n-1) + T(n-2) + \cdots + T(1) + T(0)$, which solves to $T(n) = O(2^n)$. Since we spend $O(n)$ time within a call, the time complexity is $O(n2^n)$. The space complexity is $O(n)$ which comes from the maximum stack depth as well as the maximum size of a subset.

Variant 12.1.1: Solve Problem 12.1 on Page 25 when the input array may have duplicates, i.e., denotes a multiset. You should not repeat any multiset. For example, if $A = \langle 1, 2, 3, 2 \rangle$, then you should return $\Big\langle \langle\rangle, \langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 2\rangle, \langle 2, 3\rangle, \langle 1, 2, 2\rangle, \langle 1, 2, 3\rangle, \langle 2, 2, 3\rangle, \langle 1, 2, 2, 3\rangle \Big\rangle$.

Problem 13.1, *pg. 29*: *How many ways can you go from the top-left to the bottom-right in an $n \times m$ 2D array? How would you count the number of ways in the presence of obstacles, specified by an $n \times m$ Boolean 2D array B, where a* `true` *represents an obstacle.*

Solution 13.1: This problem can be solved using a straightforward application of DP: the number of ways to get to (i, j) is the number of ways to get to $(i-1, j)$ plus the number of ways to get to $(i, j-1)$. (If $i = 0$ or $j = 0$, there is only one way to get to (i, j).) The matrix storing the number of ways to get to (i, j) for the configuration in Figure 13.2 on Page 29 is shown in Figure 17.2. To fill in the i-th row we do not need values from rows before $i-1$. Consequently, we do not need an $n \times m$ 2D array. Instead, we can use two rows of storage, and alternate between them. With a little more care, we can get by with a single row. By symmetry, the number of ways to get to (i, j) is the same as to get to (j, i), so we use the smaller of m and n to be the number of columns (which is the number of entries in a row). This leads to the following algorithm.

```
int NumberOfWays(int n, int m) {
  if (n < m) {
    swap(n, m);
  }
  vector<int> A(m, 1);
  for (int i = 1; i < n; ++i) {
    int prev_res = 0;
    for (int j = 0; j < m; ++j) {
      A[j] = A[j] + prev_res;
      prev_res = A[j];
    }
  }
  return A[m - 1];
}
```

The time complexity is $O(mn)$, and the space complexity is $O(\min(m, n))$.

1	**1**	**1**	**1**	**1**
1	**2**	**3**	**4**	**5**
1	**3**	**6**	**10**	**15**
1	**4**	**10**	**20**	**35**
1	**5**	**15**	**35**	**70**

Figure 17.2: The number of ways to get from $(0, 0)$ to (i, j) for $0 \leq i, j \leq 4$.

Another way of deriving the same answer is to use the fact that each path from $(0, 0)$ to $(n-1, m-1)$ is a sequence of $m-1$ horizontal steps and $n-1$ vertical steps. There are $\binom{n+m-2}{n-1} = \binom{n+m-2}{m-1} = \frac{(n+m-2)!}{(n-1)!(m-1)!}$ such paths.

Our first solution generalizes trivially to obstacles: if there is an obstacle at (i, j) there are zero ways of getting from $(0, 0)$ to (i, j).

```
// Given the dimensions of A, n and m, and B, return the number of ways
// from A[0][0] to A[n - 1][m - 1] considering obstacles.
int NumberOfWaysWithObstacles(int n, int m, const vector<deque<bool>>& B) {
  vector<vector<int>> A(n, vector<int>(m, 0));
  if (B[0][0]) {  // No way to start from (0, 0) if B[0][0] == true.
    return 0;
```

```
  }

  A[0][0] = 1;
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
      if (B[i][j] == 0) {
        A[i][j] += (i < 1 ? 0 : A[i - 1][j]) + (j < 1 ? 0 : A[i][j - 1]);
      }
    }
  }
  return A.back().back();
}
```

The time complexity is $O(nm)$.

Variant 13.1.1: A decimal number is a sequence of digits, i.e., a sequence over $\{0, 1, 2, \ldots, 9\}$. The sequence has to be of length 1 or more, and the first element in the sequence cannot be 0. Call a decimal number D *monotone* if $D[i] \leq D[i+1], 0 \leq i < |D|$. Write a function which takes as input a positive integer k and computes the number of decimal numbers of length k that are monotone.

Variant 13.1.2: Call a decimal number D, as defined above, *strictly monotone* if $D[i] < D[i+1], 0 \leq i < |D|$. Write a function which takes as input a positive integer k and computes the number of decimal numbers of length k that are strictly monotone.

Problem 14.1, *pg. 30*: *Design an algorithm that takes as input an array A and a number t, and determines if A 3-creates t.*

Solution 14.1: First, we consider the problem of computing a pair of entries which sum to K. Assume A is sorted.We start with the pair consisting of the first element and the last element: $(A[0], A[n-1])$. Let $s = A[0]+A[n-1]$. If $s = K$, we are done. If $s < K$, we increase the sum by moving to pair $(A[1], A[n-1])$. We need never consider $A[0]$; since the array is sorted, for all i, $A[0]+A[i] \leq A[0]+A[n-1] = K < s$. If $s > K$, we can decrease the sum by considering the pair $(A[0], A[n-2])$; by analogous reasoning, we need never consider $A[n-1]$ again. We iteratively continue this process till we have found a pair that sums up to K or the indices meet, in which case the search ends. This solution works in $O(n)$ time and $O(1)$ space in addition to the space needed to store A.

Now we describe a solution to the problem of finding three entries which sum to t. We sort A and for each $A[i]$, search for indices j and k such that $A[j]+A[k] = t-A[i]$. The additional space needed is $O(1)$, and the time complexity is the sum of the time taken to sort, $O(n \log n)$, and then to run the $O(n)$ algorithm described in the previous paragraph n times (one for each entry), which is $O(n^2)$ overall. The code for this approach is shown below.

```
bool HasThreeSum(vector<int> A, int t) {
  sort(A.begin(), A.end());

```

```
  for (int a : A) {
    // Finds if the sum of two numbers in A equals to t - a.
    if (HasTwoSum(A, t - a)) {
      return true;
    }
  }
  return false;
}

bool HasTwoSum(const vector<int>& A, int t) {
  int j = 0, k = A.size() - 1;

  while (j <= k) {
    if (A[j] + A[k] == t) {
      return true;
    } else if (A[j] + A[k] < t) {
      ++j;
    } else {  // A[j] + A[k] > t.
      --k;
    }
  }
  return false;
}
```

Surprisingly, it is possible, in theory, to improve the time complexity when the entries in A are nonnegative integers in a small range, specifically, the maximum entry is $O(n)$. The idea is to determine all possible 3-sums by encoding the array as a polynomial $P_A(x) = \sum_{i=0}^{n-1} x^{A[i]}$. The powers of x that appear in the polynomial $P_A(x) \times P_A(x)$ corresponds to sums of pairs of elements in A; similarly, the powers of x in $P_A(x) \times P_A(x) \times P_A(x)$ correspond to sums of triples of elements in A. Two n-degree polynomials can be multiplied in $O(n \log n)$ time using the fast Fourier Transform (FFT). The details are long and tedious, and the approach is unlikely to do well in practice.

Variant 14.1.1: Solve the same problem when the three elements must be distinct. For example, if $A = \langle 5, 2, 3, 4, 3 \rangle$ and $t = 9$, then $A[2] + A[2] + A[2]$ is not acceptable, $A[2] + A[2] + A[4]$ is not acceptable, but $A[1] + A[2] + A[3]$ and $A[1] + A[3] + A[4]$ are acceptable.

Variant 14.1.2: Solve the same problem when k is an additional input.

Variant 14.1.3: Write a function that takes as input an array of integers A and an integer T, and returns a 3-tuple $(A[p], A[q], A[r])$ where p, q, r are all distinct, minimizing $|T - (A[p] + A[q] + A[r])|$, and $A[p] \leq A[r] \leq A[s]$.

Problem 15.1, *pg. 34*: *Implement a routine that takes a $n \times m$ Boolean array A together with an entry (x, y) and flips the color of the region associated with (x, y). See Figure 15.5 on Page 34 for an example of flipping.*

Solution 15.1: Conceptually, this problem is very similar to that of exploring an undirected graph. Entries can be viewed as vertices, and neighboring vertices are connected by edges.

For the current problem, we are searching for all vertices whose color is the same as that of (x, y) that are reachable from (x, y). Breadth-first search is natural when starting with a set of vertices. Specifically, we can use a queue to store such vertices. The queue is initialized to (x, y). The queue is popped iteratively. Call the popped point p. First, we record p's initial color, and then flip its color. Next we examine p neighbors. Any neighbor which is the same color as p's initial color is added to `q`. The computation ends when `q` is empty. Correctness follows from the fact that any point that is added to the queue is reachable from (x, y) via a path consisting of points of the same color, and all points reachable from (x, y) via points of the same color will eventually be added to the queue.

```
void FilpColor(int x, int y, vector<deque<bool>> *A) {
  const array<array<int, 2>, 4> dir = {{{{0, 1}}, {{0, -1}},
                                        {{1, 0}}, {{-1, 0}}}};
  const bool color = (*A)[x][y];

  queue<pair<int, int>> q;
  (*A)[x][y] = !(*A)[x][y];  // Flips.
  q.emplace(x, y);
  while (!q.empty()) {
    auto curr(q.front());
    for (const auto& d : dir) {
      const pair<int, int> next(curr.first + d[0], curr.second + d[1]);
      if (next.first >= 0 && next.first < A->size() &&
          next.second >= 0 && next.second < (*A)[next.first].size() &&
          (*A)[next.first][next.second] == color) {
        // Flips the color.
        (*A)[next.first][next.second] = !(*A)[next.first][next.second];
        q.emplace(next);
      }
    }
    q.pop();
  }
}
```

The time complexity is the same as that of BFS, i.e., $O(mn)$. The space complexity is a little better than the worst case for BFS, since there are at most $O(m + n)$ vertices that are at the same distance from a given entry.

We also provide a recursive solution which is in the spirit of DFS. It does not need a queue but implicitly uses a stack, namely the function call stack.

```
void FilpColor(int x, int y, vector<deque<bool>> *A) {
  const array<array<int, 2>, 4> dir = {{{{0, 1}}, {{0, -1}},
                                        {{1, 0}}, {{-1, 0}}}};
  const bool color = (*A)[x][y];

  (*A)[x][y] = !(*A)[x][y];  // Flips.
  for (const auto& d : dir) {
    const int nx = x + d[0], ny = y + d[1];
```

```
9       if (nx >= 0 && nx < A->size() && ny >= 0 && ny < (*A)[nx].size() &&
10          (*A)[nx][ny] == color) {
11        FilpColor(nx, ny, A);
12      }
13    }
14  }
```

The time complexity is the same as that of DFS.

Both the algorithms given above differ slightly from traditional BFS and DFS algorithms. The reason is that we have a color field already available, and hence do not need the auxiliary color field traditionally associated with vertices BFS and DFS. Furthermore, since we are simply determining reachability, we only need two colors, whereas BFS and DFS traditionally use three colors to track state. (The use of an additional color makes it possible, for example, to answer questions about cycles in directed graphs, but that is not relevant here.)

Variant 15.1.1: Design an algorithm for computing the black region that contains the most points.

Variant 15.1.2: Design an algorithm that takes a point (a, b), sets $A(a, b)$ to black, and returns the size of the black region that contains the most points. Assume this algorithm will be called multiple times, and you want to keep the aggregate run time as low as possible.

Problem 16.1, *pg. 36*: *Develop a* `Timer` *class that manages the execution of deferred tasks. The* `Timer` *constructor takes as its argument an object which includes a* `Run` *method and a* `name` *field, which is a string.* `Timer` *must support—(1.) starting a thread, identified by name, at a given time in the future; and (2.) canceling a thread, identified by name (the cancel request is to be ignored if the thread has already started).*

Solution 16.1: The two aspects to the design are the data structures and the locking mechanism.

We use two data structures. The first is a min-heap in which we insert key-value pairs: the keys are run times and the values are the thread to run at that time. A dispatch thread runs these threads; it sleeps from call to call and may be woken up if a thread is added to or deleted from the pool. If woken up, it advances or retards its remaining sleep time based on the top of the min-heap. On waking up, it looks for the thread at the top of the min-heap—if its launch time is the current time, the dispatch thread deletes it from the min-heap and executes it. It then sleeps till the launch time for the next thread in the min-heap. (Because of deletions, it may happen that the dispatch thread wakes up and finds nothing to do.)

The second data structure is a hash table with thread ids as keys and entries in the min-heap as values. If we need to cancel a thread, we go to the min-heap and delete it. Each time a thread is added, we add it to the min-heap; if the insertion is to the top of the min-heap, we interrupt the dispatch thread so that it can adjust its wake up time.

Since the min-heap is shared by the update methods and the dispatch thread, we need to lock it. The simplest solution is to have a single lock that is used for all read and writes into the min-heap and the hash table.

Problem 17.1, *pg. 39*: *Design a feature that allows a studio to enter a set V of videos that belong to it, and to determine which videos in the YouTV.com database match videos in V.*

Solution 17.1: Videos differ from documents in that the same content may be encoded in many different formats, with different resolutions, and levels of compression.

One way to reduce the duplicate video problem to the duplicate document problem is to re-encode all videos to a common format, resolution, and compression level. This in itself does not mean that two videos of the same content get reduced to identical files—the initial settings affect the resulting videos. However, we can now "signature" the normalized video.

A trivial signature would be to assign a 0 or a 1 to each frame based on whether it has more or less brightness than average. A more sophisticated signature would be a 3 bit measure of the red, green, and blue intensities for each frame. Even more sophisticated signatures can be developed, e.g., by taking into account the regions on individual frames. The motivation for better signatures is to reduce the number of false matches returned by the system, and thereby reduce the amount of time needed to review the matches.

The solution proposed above is algorithmic. However, there are alternative approaches that could be effective: letting users flag videos that infringe copyright (and possibly rewarding them for their effort), checking for videos that are identical to videos that have previously been identified as infringing, looking at meta-information in the video header, etc.

Variant 17.1.1: Design an online music identification service.

Part IV

Notation and Index

Notation

To speak about notation as the only way that you can guarantee structure of course is already very suspect.

— E. S. Parker

We use the following convention for symbols, unless the surrounding text specifies otherwise:

i, j, k	nonnegative array indices
f, g, h	function
A	k-dimensional array
L	linked list or doubly linked list
S	set
T	tree
G	graph
V	set of vertices of a graph
E	set of edges of a graph
u, v	vertex-valued variables
e	edge-valued variable
m, n	number of elements in a collection
x, y	real-valued variables
σ	a permutation

Symbolism	Meaning
$(d_{k-1} \ldots d_0)_r$	radix-r representation of a number, e.g., $(1011)_2$
$\log_b x$	logarithm of x to the base b
$\lg x$	logarithm of x to the base 2
$\lvert S \rvert$	cardinality of set S
$S \setminus T$	set difference, i.e., $S \cap T'$, sometimes written as $S - T$
$\lvert x \rvert$	absolute value of x
$\lfloor x \rfloor$	greatest integer less than or equal to x
$\lceil x \rceil$	smallest integer greater than or equal to x
$\langle a_0, a_1, \ldots, a_{n-1} \rangle$	sequence of n elements
a^k, $a = \langle a_0, \ldots, a_{n-1} \rangle$	the sequence $\langle a_k, a_{k+1}, \ldots, a_{n-1} \rangle$
$\sum_{R(k)} f(k)$	sum of all $f(k)$ such that relation $R(k)$ is true
$\prod_{R(k)} f(k)$	product of all $f(k)$ such that relation $R(k)$ is true
$\min_{R(k)} f(k)$	minimum of all $f(k)$ such that relation $R(k)$ is true
$\max_{R(k)} f(k)$	maximum of all $f(k)$ such that relation $R(k)$ is true

$\sum_{k=a}^{b} f(k)$	shorthand for $\sum_{a \leq k \leq b} f(k)$
$\prod_{k=a}^{b} f(k)$	shorthand for $\prod_{a \leq k \leq b} f(k)$
$\{a \mid R(a)\}$	set of all a such that the relation $R(a)$ = true
$[l, r]$	closed interval: $\{x \mid l \leq x \leq r\}$
(l, r)	open interval: $\{x \mid l < x < r\}$
$[l, r)$	$\{x \mid l \leq x < r\}$
$(l, r]$	$\{x \mid l < x \leq r\}$
$\{a, b, \ldots\}$	well-defined collection of elements, i.e., a set
A_i or $A[i]$	the i-th element of one-dimensional array A
$A[i : j]$	subarray of one-dimensional array A consisting of elements at indices i to j inclusive
$A[i][j]$ or $A[i, j]$	the element in i-th row and j-th column of 2D array A
$A[i_1 : i_2][j_1 : j_2]$	2D subarray of 2D array A consisting of elements from i_1-th to i_2-th rows and from j_1-th to j_2-th column, inclusive
$\binom{n}{k}$	binomial coefficient: number of ways of choosing k elements from a set of n items
$n!$	n-factorial, the product of the integers from 1 to n, inclusive
$O\big(f(n)\big)$	big-oh complexity of $f(n)$, asymptotic upper bound
x mod y	mod function
$x \oplus y$	bitwise-XOR function
$x \approx y$	x is approximately equal to y
`null`	pointer value reserved for indicating that the pointer does not refer to a valid address
$\emptyset$	empty set
∞	infinity: Informally, a number larger than any number.
$\mathcal{Z}$	the set of integers $\{\ldots, -2, -1, 0, 1, 2, 3, \ldots\}$
$\mathcal{Z}^+$	the set of nonnegative integers $\{0, 1, 2, 3, \ldots\}$
$\mathcal{Z}_n$	the set $\{0, 1, 2, 3, \ldots, n-1\}$
$x \ll y$	much less than
$x \gg y$	much greater than
$\Rightarrow$	logical implication

Index of Terms

2D array, 29, 34, 64, 65, 73
2D subarray, 73
$O(1)$ space, 12, 20, 28, 52, 61, 66

adjacency list, 33, **33**
adjacency matrix, 33, **33**
amortized, 8
amortized analysis, 18
array, 4, **4**, 8, 16–18, 20, 21, 23, 27, 28, 30, 34, 38, 44, 56–58, 66, 67
    bit, *see* bit array
    deletion from, 4

balanced BST, 13
BFS, 33, **33**, 62, 68, 69
binary search, 15, **15**, 16, 17, 20, 30, 58
binary search tree, 18
    height of, 23
    red-black tree, 23
binary tree, 9–13, 23, 33, 52–54, 60
    complete, 11, 13
    full, 11
    height of, 11, 12, 54, 55
    perfect, 11
binomial coefficient, 73
bit array, 63
breadth-first search, *see* BFS
BST, 19, 20, 23, 60–62

caching, **37**, 38
central processing unit, *see* CPU
child, 11, 23, 33
closed interval, **73**
coloring, 34
complete binary tree, 11, **11**, 12, 13
    height of, 11
complex number, 2
connected component, 32, **32**
connected directed graph, **32**
connected undirected graph, 32, **32**
connected vertices, 32, **32**
constraint, 60
counting sort, 20
CPU, **38**

DAG, 31, **31**, 32
database, 38, 39, 70
deadlock, **36**
decomposition, 37, 38
degree
    of a polynomial, 67
deletion
    from arrays, 4
    from doubly linked lists, 9
    from hash tables, 18
    from max-heaps, 13
depth
    of a node in a binary search tree, 62
    of a node in a binary tree, 11, **11**, 12
    of the function call stack, 64
depth-first search, *see* DFS
deque, **9**
dequeue, 9, 53
DFS, 33, **33**, 68, 69
directed acyclic graph, *see* DAG, 32
directed graph, 31, *see also* directed acyclic graph, *see also* graph, **31**, 32, 69
    connected directed graph, 32
    weakly connected graph, 32
discovery time, **33**
distributed memory, **35**, 36
distribution
    of the numbers, 38
divide-and-conquer, 27, 28
double-ended queue, *see* deque
doubly linked list, 6, **6**, 9, 72
    deletion from, 9
DP, 27–29, 65
dynamic programming, 27

edge, 31, **31**, 32, 33, 72
edge set, 33

elimination, 16
enqueue, 9, 53, 62
extract-max, 13
extract-min, 55

fast Fourier Transform, *see* FFT
FFT, **67**
Fibonacci number, 27
finishing time, **33**
first-in, first-out, 9
free tree, 33, **33**
full binary tree, 11, **11**

garbage collection, 35
graph, 31, *see also* directed graph, **31**, 32, 33, *see also* tree
graphical user interfaces, *see* GUI
GUI, **35**

hash code, 18, **18**, 19
hash function, 18, 19
hash table, 5, 18, 19, 48, 54, 57, 69, 70
    deletion from, 18
    lookup of, 18
head
    of a deque, 9
    of a linked list, 6, 7, 48
    of a queue, 9, 53
heap, 13, **13**, 21, 27
    max-heap, 13, 21
    min-heap, 13, 21
heapsort, 20
height
    of a binary search tree, 23
    of a binary tree, 11, **11**, 12, 54, 55
    of a complete binary tree, 11
    of a event rectangle, 21
    of a perfect binary tree, 11
    of a stack, 55

I/O, 55
in-place sort, **20**
invariant, 30
inverted index, 21

last-in, first-out, 8
leaf, 11
left child, 10, 11, 33, 54, 60, 62
left subtree, 10–12, 23, 60–62
level
    of a tree, **11**
linked list, 6, 72
list, *see also* singly linked list, 7–9, 18, 20, 48
livelock, **36**
load
    of a hash table, 18
lock
    deadlock, 36
    livelock, 36

matching
    of strings, 5
matrix, 33, 65
    adjacency, 33
    multiplication of, 35
matrix multiplication, 35
max-heap, 13, 21
    deletion from, 13
merge sort, 20
min-heap, 13, 20, 21, 38, 55, 69, 70
multicore, 35

network, 35
    network bandwidth, 38
network bandwidth, 38
node, 10–12, 19, 23, 33, 41, 53–55, 60–62

open interval, **73**
ordered tree, 33, **33**
OS, 37
overflow
    integer, 16
overlapping intervals, 59

parallelism, 35–38
parent-child relationship, 11, 33
path, **31**
perfect binary tree, 11, **11**, 12
    height of, 11
permutation
    random, 4
power set, 25, **25**, 26, 63
prime, 23

queue, 9, **9**, 53, 62, 68
quicksort, 20, 28, 29

race, 36
radix sort, 21
RAM, 14, **38**, 55
random access memory, *see* RAM
random permutation, 4
randomization, 18
reachable, **31**, 33
recursion, 27
red-black tree, **23**
rehashing, 18
right child, 10, 11, 33, 54, 60, 62
right subtree, 10–12, 23, 60, 61
rolling hash, 19

root, 9–12, 23, 33, 52, 54, 55, 60–62
rooted tree, 33, **33**

searching
    binary search, *see* binary search
shared memory, 35, **35**, 36
Short Message Service, *see* SMS
signature, 70
singly linked list, 6, **6**, 7, 48
sinks, **32**
SMS, **36**
sorting, 17, 20, 21, 38, 59
    counting sort, 20
    heapsort, 20
    in-place, 20
    in-place sort, 20
    merge sort, 20
    quicksort, 20, 28, 29
    radix sort, 21
    stable, 20
    stable sort, 20
sources, **32**
spanning tree, 33, *see also* minimum spanning tree
stable sort, **20**
stack, 8, **8**, 50, 52, 68
    height of, 55
Standard Template Library, *see* STL
starvation, **36**
STL, **23**
string, 3, 5, 19, 36, 45–47, 69
string matching, 5
strongly connected directed graph, **32**
subarray, 28, 29, 56
subtree, 11, 55, 60, 62

tail
    of a deque, 9
    of a linked list, 6, 48
    of a queue, 9
tail recursive, 27
topological ordering, **32**
tree, 33, **33**
    binary, *see* binary tree
    free, 33
    ordered, 33
    red-black, 23
    rooted, 33

UI, 35
undirected graph, 32, **32**, 33, 68

vertex, 31, **31**, 32, 33, 72
    connected, 32
weakly connected graph, **32**
width, **2**

Acknowledgments

Several of our friends, colleagues, and readers gave feedback. We would like to thank Taras Bobrovytsky, Senthil Chellappan, Yi-Ting Chen, Monica Farkash, Dongbo Hu, Jing-Tang Keith Jang, Matthieu Jeanson, Gerson Kurz, Danyu Liu, Hari Mony, Shaun Phillips, Gayatri Ramachandran, Ulises Reyes, Kumud Sanwal, Tom Shiple, Ian Varley, Shaohua Wan, Don Wong, and Xiang Wu for their input.

I, Adnan Aziz, thank my teachers, friends, and students from IIT Kanpur, UC Berkeley, and UT Austin for having nurtured my passion for programming. I especially thank my friends Vineet Gupta, Tom Shiple, and Vigyan Singhal, and my teachers Robert Solovay, Robert Brayton, Richard Karp, Raimund Seidel, and Somenath Biswas, for all that they taught me. My coauthor, Tsung-Hsien Lee, brought a passion that was infectious and inspirational. My coauthor, Amit Prakash, has been a wonderful collaborator for many years—this book is a testament to his intellect, creativity, and enthusiasm. I look forward to a lifelong collaboration with both of them.

I, Tsung-Hsien Lee, would like to thank my coauthors, Adnan Aziz and Amit Prakash, who give me this once-in-a-life-time opportunity. I also thank my teachers Wen-Lian Hsu, Ren-Song Tsay, Biing-Feng Wang, and Ting-Chi Wang for having initiated and nurtured my passion for computer science in general, and algorithms in particular. I would like to thank my friends Cheng-Yi He, Da-Cheng Juan, Chien-Hsin Lin, and Chih-Chiang Yu, who accompanied me on the road of savoring the joy of programming contests; and Kuan-Chieh Chen, Chun-Cheng Chou, Ray Chuang, Wen-Sao Hong, Wei-Lun Hung, Nigel Liang, Huan-Kai Peng, and Yu-En Tsai, who give me valuable feedback on this book. Last, I would like to thank all my friends and colleagues at Facebook, National Tsing Hua University, and UT Austin for the brain-storming on puzzles; it is indeed my greatest honor to have known all of you.

I, Amit Prakash, have my coauthor and mentor, Adnan Aziz, to thank the most for this book. To a great extent, my problem solving skills have been shaped by Adnan. There have been occasions in life when I would not have made it through without his help. He is also the best possible collaborator I can think of for any intellectual endeavor. I have come to know Tsung-Hsien through working on this book. He has been a great coauthor. His passion and commitment to excellence can be seen everywhere in this book. Over the years, I have been fortunate to have had great teachers at IIT Kanpur and UT Austin. I would especially like to thank my teachers Scott Nettles, Vijaya Ramachandran, and Gustavo de Veciana. I would also like to thank my friends and colleagues at Google, Microsoft, IIT Kanpur, and UT Austin for many stimulating conversations and problem solving sessions. Finally, and most importantly, I want to thank my family who have been a constant source of support, excitement, and joy all my life and especially during the process of writing this book.

Adnan Aziz — *Austin, Texas*
Tsung-Hsien Lee — *Mountain View, California*
Amit Prakash — *Belmont, California*
October, 2012