Elements of Programming Interviews

The Insiders' Guide

Adnan Aziz
Tsung-Hsien Lee
Amit Prakash

This document is a sampling of the C++ version 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 material 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 at Amazon (paperback) and Google Play (Ebook). Visit our website for details.

http://ElementsOfProgrammingInterviews.com

Table of Contents

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. Variables come in types—a type is a classification of data that spells out possible values for that type and the operations that can be performed on it. A type can be provided by the language or defined by the programmer. Many languages provide types for Boolean, integer, character and floating point data. Often, there are multiple integer and floating point types, depending on signedness and precision. The width of these types is the number of bits of storage a corresponding variable 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.

Primitive types boot camp

Writing a program to count the number of bits that are set to 1 in an integer is a good way to get up to speed with primitive types. The following program tests bits one-at-a-time starting with the least-significant bit. It illustrates shifting and masking; it also shows how to avoid hard-coding the size of the integer word.

```
short CountBits(unsigned int x) {
  short num_bits = 0;
  while (x) {
    num_bits += x & 1;
    x >>= 1;
  }
  return num_bits;
}
```

Since we perform $O(1)$ computation per bit, the time complexity is $O(n)$, where n is the number of bits in the integer word. Note that, by definition, the time complexity is for the worst case input, which for this example is the word $(111\ldots11)_2$. In the best case, the time complexity is $O(1)$, e.g., if the input is 0. The techniques in Solution 1.1 on the facing page, which is concerned with counting the number of bits modulo 2, i.e., the parity, can be used to improve the performance of the program given above.

Know your primitive types

You should know the primitive types very intimately, e.g., sizes, ranges, signedness properties, and operators. You should also know the utility methods for primitive types in `cmath`. The `random` library is also very helpful, especially when writing tests.

- Be very familiar with the bit-wise operators such as `6&4`, `1|2`, `8>>1`, `1<<10`, `~0`, `15^x`.

Be very comfortable with the **bitwise operators**, particularly XOR.

Understand how to use **masks** and create them in an **machine independent** way.

Know fast ways to **clear the lowermost set bit** (and by extension, set the lowermost 0, get its index, etc.)

Understand **signedness** and its implications to **shifting**.

Consider using a **cache** to accelerate operations by using it to brute-force small inputs.

Be aware that **commutativity** and **associativity** can be used to perform operations in **parallel** and **reorder** operations.

- Know the constants denoting the maximum and minimum values for numeric types, etc., e.g., `numeric_limits<int>::min()`, `numeric_limits<float>.max()`, `numeric_limits<double>::infinity()`.
- Take extra care when comparing floating point values.
- Key methods in `cmath` are `abs(-34)`, `fabs(-3.14)`, `ceil(2.17)`, `floor(3.14)`, `min(x, -4)`, `max(3.14, y)`, `pow(2.71, 3.14)`, `log(7.12)`, and `sqrt(225)`.
- Know how to interconvert integers, characters, and strings, e.g., `x - '0'` to convert a digit character to an integer.
- Key methods in `random` are `uniform_int_distribution<> dis(1, 6)` (which returns an integer value in $[1, 6]$), `uniform_real_distribution<double> dis(1.3, 2.9)` (which returns a floating point number in $[1.3, 2.9]$), `generate_canonical<double, 10>` (which returns a value in $[0, 1)$).

1.1 Computing the parity of a word

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.

How would you compute the parity of a very large number of 64-bit words?

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

Solution: The brute-force algorithm iteratively tests the value of each bit while tracking the number of 1s seen so far. Since we only care if the number of 1s is even or odd, we can store the number modulo 2.

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

```
}
```

The time complexity is $O(n)$, where n is the word size.

If we know how to erase the lowest set bit in a word in a signle operation by using XOR. This 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 in a particular word. (For example, for 10001010, $k = 3$.) 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, two keys to performance are processing multiple bits at a time and caching results in an array-based lookup table.

First we demonstrate caching. Clearly, we cannot cache the parity of every 64-bit integer—we would need 2^{64} bits of storage, which is of the order of ten trillion exabytes. However, when computing the parity of a collection of bits, it does not matter how we group those bits, i.e., the computation is associative. Therefore, we can compute the parity of a 64-bit integer by grouping its bits into four nonoverlapping 16 bit subwords, computing the parity of each subwords, and then computing the parity of these four subresults. We choose 16 since $2^{16} = 65536$ is relatively small, which makes it feasible to cache the parity of all 16-bit words using an array. Furthermore, since 16 evenly divides 64, the code is simpler than if we were, for example, to use 10 bit subwords.

We illustrate the approach with a lookup table for 2-bit words. The cache is $\langle 0, 1, 1, 0 \rangle$—these are the parities of (00), (01), (10), (11), respectively. To compute the parity of (11001010) we would compute the parities of (11), (00), (10), (10). By table lookup we see these are $0, 0, 1, 1$, respectively, so the final result is the parity of $0, 0, 1, 1$ which is 0.

To lookup the parity of the first two bits in (11101010), we right shift by 6, to get (00000011), and use this as an index into the cache. To lookup the parity of the next two bits, i.e., (10), we right shift by 4, to get (10) in the two least-significant bit places. The right shift does not remove the leading (11)—it results in (00001110). We cannot index the cache with this, it leads to an out-of-bounds access. To get the last two bits after the right shift by 4, we bitwise-AND (00001110) with (00000011) (this is the "mask" used to extract the last 2 bits). The result is (00000010). Similar masking is needed for the two other 2-bit lookups.

```
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, and n the word size. Since there are n/L terms, the time complexity 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 XOR of two bits is 0 if both bits are 0 or both bits are 1; otherwise it is 1. XOR has the property of being associative (as previously described), as well as commutative, i.e., the order in which we perform the XORs does not change the result. The XOR of a group of bits is its parity. We can exploit this fact to use the CPU's word-level XOR instruction to process multiple bits at a time.

For example, the parity of $\langle b_{63}, b_{62}, \ldots, b_3, b_2, b_1, b_0 \rangle$ equals the parity of the XOR of $\langle b_{63}, b_{62}, \ldots, b_{32} \rangle$ and $\langle b_{31}, b_{30}, \ldots, b_0 \rangle$. The XOR of these two 32-bit values can be computed with a single shift and a single 32-bit XOR instruction. We repeat the same operation on 32-, 16-, 8-, 4-, 2-, and 1-bit operands to get the final result. Note that the leading bits are not meaningful, and we have to explicitly extract the result from the least-significant bit.

We illustrate the approach with an 8-bit word. The parity of (11010111) is the same as the parity of (1101) XORed with (0111), i.e., of (1010). This in turn is the same as the parity of (10) XORed with (10), i.e., of (00). The final result is the XOR of (0) with (0), i.e., 0. Note that the first XOR yields (11011010), and only the last 4 bits are relevant going forward. The second XOR yields (11101100), and only the last 2 bits are relevant. The third XOR yields (10011010). The last bit is the result, and to extract it we have to bitwise-AND with (00000001).

```
short Parity(unsigned long x) {
  x ^= x >> 32;
  x ^= x >> 16;
  x ^= x >> 8;
  x ^= x >> 4;
  x ^= x >> 2;
  x ^= x >> 1;
  return x & 0x1;
}
```

The time complexity is $O(\log n)$, where n is the word size.

Note that we could have combined caching with word-level operations, e.g., by doing a lookup once we get to 16 bits. The actual runtimes depend on the input data, e.g., the refinement of the brute-force algorithm is very fast on sparse inputs. However, for random inputs, the refinement of the brute-force is roughly 20% faster than the brute-force algorithm. The table-based approach is four times faster still, and using associativity reduces run time by another factor of two.

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. It is usually used to represent sequences. Given an array A, $A[i]$ denotes the $(i+1)$th object stored in the array. Retrieving and updating $A[i]$ takes $O(1)$ time. Insertion into a full array can be handled by resizing, i.e., allocating a new array with additional memory and copying over the entries from the original array. This increases the worst-case time of insertion, but if the new array has, for example, a constant factor larger than the original array, the average time for insertion is constant since resizing is infrequent. Deleting an element from an array entails moving all successive elements one over to the left to fill the vacated space. For example, if the array is $\langle 2, 3, 5, 7, 9, 11, 13, 17\rangle$, then deleting the element at index 4 results in the array $\langle 2, 3, 5, 7, 11, 13, 17, 0\rangle$. (We do not care about the last value.) The time complexity to delete the element at index i from an array of length n is $O(n-i)$.

Array boot camp

The following problem gives good insight into working with arrays: Your input is an array of integers, and you have to reorder its entries so that the even entries appear first. This is easy if you use $O(n)$ space, where n is the length of the array. However, you are required to solve it without allocating additional storage.

When working with arrays you should take advantage of the fact that you can operate efficiently on both ends. For this problem, we can partition the array into three subarrays: Even, Unclassified, and Odd, appearing in that order. Initially Even and Odd are empty, and Unclassified is the entire array. We iterate through Unclassified, moving its elements to the boundaries of the Even and Odd subarrays via swaps, thereby expanding Even and Odd, and shrinking Unclassified.

```
void EvenOdd(vector<int>* A_ptr) {
  vector<int>& A = *A_ptr;
  int next_even = 0, next_odd = A.size() - 1;
  while (next_even < next_odd) {
    if (A[next_even] % 2 == 0) {
      ++next_even;
    } else {
      swap(A[next_even], A[next_odd--]);
    }
  }
}
```

The additional space complexity is clearly $O(1)$—a couple of variables that hold indices, and a temporary variable for performing the swap. We do a constant amount of processing per entry, so the time complexity is $O(n)$.

Array problems often have simple brute-force solutions that use $O(n)$ space, but subtler solutions that **use the array itself** to **reduce space** complexity to $O(1)$.

Filling an array from the front is slow, so see if it's possible to **write values from the back**.

Instead of deleting an entry (which requires moving all entries to is right), consider **overwriting** it.

When dealing with integers encoded by an array consider reversing the array so the **least-significant digit is the first entry**.

Be comfortable with writing code that operates on **subarrays**.

It's incredibly easy to make **off-by-1** errors when operating on arrays.

Don't worry about preserving the **integrity** of the array (sortedness, keeping equal entries together, etc.) until it is time to return.

An array can serve as a good data structure when you know the distribution of the elements in advance. For example, a Boolean array of length W is a good choice for representing a **subset of** $\{0, 1, \ldots, W-1\}$. (When using a Boolean array to represent a subset of $\{1, 2, 3, \ldots, n\}$, allocate an array of size $n+1$ to simplify indexing.) .

When operating on $2D$ arrays, **use parallel logic** for rows and for columns.

Sometimes it's easier to **simulate the specification**, than to analytically solve for the result. For example, rather than writing a formula for the i-th entry in the spiral order for an $n \times n$ matrix, just compute the output from the beginning.

Know your array libraries

You should know the C++ `array` and `vector` classes very intimately. Note that `array` is fixed-size, and `vector` is dynamically-resized. There are a number of very useful methods in the `algorithm` library.

- Know the syntax for allocating and instantiating an `array` or a `vector`, i.e., `array<int, 3> A = {1, 2, 3}`, `vector<int> A = {1, 2, 3}`. To construct a subarray from an array, you can use `vector<int> subarray_A(A.begin() + i, A.begin() + j)`—this sets `subarray_A` to be $A[i : j-1]$.
- Understand how to instantiate a $2D$ array—`array<array<int, 2>, 3> A = {{1, 2}, {3, 4}, {5, 6}}` and `vector<vector<int>> A = {{1, 2}, {3, 4}, {5, 6}}` both create an array which will hold three rows where each column holds two elements.
- Since `vector` is dynamically sizable, `push_back(42)` (or `emplace_back(42)`) are frequently used to add values to the end.

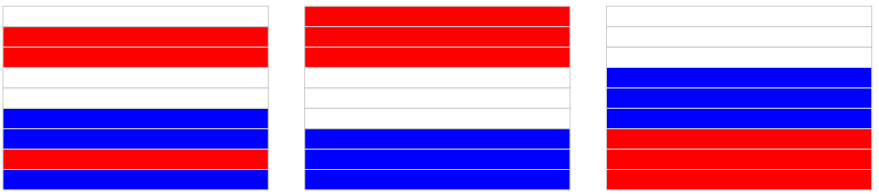
- Understand what "deep" means when checking equality of arrays, and hashing them.
- Key methods in `algorithms` include: `binary_search(A.begin(), A.end(), 42)`, `lower_bound(A.begin(), A.end(), 42)`, `upper_bound(A.begin(), A.end(), 42)`, `fill(A.begin(), A.end(), 42)`, `swap(x, y)`, `min_element(A.begin(), A.end())`, `max_element(A.begin(), A.end())`, `reverse(A.begin(), A.end())`, `rotate(A.begin(), A.begin() + shift, A.end())`, and `sort(A.begin(), A.end())`.
- Understand the variants of these methods, e.g., how to create a copy of a subarray.

2.1 The Dutch national flag problem

The quicksort algorithm for sorting arrays proceeds recursively—it selects an element (the "pivot"), reorders the array to make all the elements less than or equal to the pivot appear first, followed by all the elements greater than the pivot. The two subarrays are then sorted recursively.

Implemented naively, quicksort has large run times and deep function call stacks on arrays with many duplicates because the subarrays may differ greatly in size. One solution is to reorder the array so that all elements less than the pivot appear first, followed by elements equal to the pivot, followed by elements greater than the pivot. This is known as Dutch national flag partitioning, because the Dutch national flag consists of three horizontal bands, each in a different color.

As an example, assuming that black precedes white and white precedes gray, Figure 2.1(b) is a valid partitioning for Figure 2.1(a). If gray precedes black and black precedes white, Figure 2.1(c) is a valid partitioning for Figure 2.1(a).



(a) Before partitioning.

(b) A three-way partitioning resembling the Dutch national flag.

(c) Another three-way partitioning: the Russian national flag.

Figure 2.1: Illustrating the Dutch national flag problem.

Write a program that takes an array A and an index i into A, and rearranges the elements such that all elements less than $A[i]$ (the "pivot") appear first, followed by elements equal to the pivot, followed by elements greater than the pivot.

Hint: Think about the partition step in quicksort.

Solution: The problem is trivial to solve with $O(n)$ additional space, where n is the length of A. We form three lists, namely, elements less than the pivot, elements equal to the pivot, and elements greater than the pivot. Consequently, we write these values into A. The time complexity is $O(n)$.

We can avoid using $O(n)$ additional space at the cost of increased time complexity as follows. In the first stage, we iterate through A starting from index 0, then index 1, etc. In each iteration, we seek an element smaller than the pivot—as soon as we find it, we move it to the subarray of smaller elements via an exchange. This moves all the elements less than the pivot to the start of the array. The second stage is similar to the first one, the difference being that we move elements greater than the pivot to the end of the array. Code illustrating this approach is shown below.

```
typedef enum { RED, WHITE, BLUE } Color;

void DutchFlagPartition(int pivot_index, vector<Color>* A_ptr) {
  vector<Color>& A = *A_ptr;
  Color pivot = A[pivot_index];
  // First pass: group elements smaller than pivot.
  for (int i = 0; i < A.size(); ++i) {
    // Look for a smaller element.
    for (int j = i + 1; j < A.size(); ++j) {
      if (A[j] < pivot) {
        swap(A[i], A[j]);
        break;
      }
    }
  }
  // Second pass: group elements larger than pivot.
  for (int i = A.size() - 1; i >= 0 && A[i] >= pivot; --i) {
    // Look for a larger element. Stop when we reach an element less
    // than pivot, since first pass has moved them to the start of A.
    for (int j = i - 1; j >= 0 && A[j] >= pivot; --j) {
      if (A[j] > pivot) {
        swap(A[i], A[j]);
        break;
      }
    }
  }
}
```

The additional space complexity is now $O(1)$, but the time complexity is $O(n^2)$, e.g., if $i = n/2$ and all elements before i are greater than $A[i]$, and all elements after i are less than $A[i]$. Intuitively, this approach has bad time complexity because in the first pass when searching for each additional element smaller than the pivot we start from the beginning. However, there is no reason to start from so far back—we can begin from the last location we advanced to. (Similar comments hold for the second pass.)

To improve time complexity, we make a single pass and move all the elements less than the pivot to the beginning, In the second pass we move the larger elements to the end. It is easy to perform each pass in a single iteration, moving out-of-place elements as soon as they are discovered.

```
typedef enum { RED, WHITE, BLUE } Color;

void DutchFlagPartition(int pivot_index, vector<Color>* A_ptr) {
  vector<Color>& A = *A_ptr;
  Color pivot = A[pivot_index];
  // First pass: group elements smaller than pivot.
  int smaller = 0;
  for (int i = 0; i < A.size(); ++i) {
    if (A[i] < pivot) {
      swap(A[i], A[smaller++]);
    }
  }
  // Second pass: group elements larger than pivot.
  int larger = A.size() - 1;
  for (int i = A.size() - 1; i >= 0 && A[i] >= pivot; --i) {
    if (A[i] > pivot) {
```

```
      swap(A[i], A[larger--]);
    }
  }
}
```

The time complexity is $O(n)$ and the space complexity is $O(1)$.

The algorithm we now present is similar to the one sketched above. The main difference is that it performs classification into elements less than, equal to, and greater than the pivot in a single pass. This reduces runtime, at the cost of a trickier implementation. We do this by maintaining four subarrays: *bottom* (elements less than pivot), *middle* (elements equal to pivot), *unclassified*, and *top* (elements greater than pivot). Initially, all elements are in *unclassified*. We iterate through elements in *unclassified*, and move elements into one of *bottom*, *middle*, and *top* groups according to the relative order between the incoming unclassified element and the pivot.

As a concrete example, suppose the array is currently $A = \langle -3, 0, -1, 1, 1, ?, ?, ?, 4, 2 \rangle$, where the pivot is 1 and ? denotes unclassified elements. There are three possibilities for the first unclassified element, $A[5]$.

- $A[5]$ is less than the pivot, e.g., $A[5] = -5$. We exchange it with the first 1, i.e., the new array is $\langle -3, 0, -1, -5, 1, 1, ?, ?, 4, 2 \rangle$.
- $A[5]$ is equal to the pivot, i.e., $A[5] = 1$. We do not need to move it, we just advance to the next unclassified element, i.e., the array is $\langle -3, 0, -1, 1, 1, 1, ?, ?, 4, 2 \rangle$.
- $A[5]$ is greater than the pivot, e.g., $A[5] = 3$. We exchange it with the last unclassified element, i.e., the new array is $\langle -3, 0, -1, 1, 1, ?, ?, 3, 4, 2 \rangle$.

Note how the number of unclassified elements reduces by one in each case.

```
typedef enum { RED, WHITE, BLUE } Color;

void DutchFlagPartition(int pivot_index, vector<Color>* A_ptr) {
  vector<Color>& A = *A_ptr;
  Color pivot = A[pivot_index];
  /**
   * Keep the following invariants during partitioning:
   * bottom group: A[0 : smaller - 1].
   * middle group: A[smaller : equal - 1].
   * unclassified group: A[equal : larger - 1].
   * top group: A[larger : A.size() - 1].
   */
  int smaller = 0, equal = 0, larger = A.size();
  // Keep iterating as long as there is an unclassified element.
  while (equal < larger) {
    // A[equal] is the incoming unclassified element.
    if (A[equal] < pivot) {
      swap(A[smaller++], A[equal++]);
    } else if (A[equal] == pivot) {
      ++equal;
    } else {  // A[equal] > pivot.
      swap(A[equal], A[--larger]);
    }
  }
}
```

Each iteration decreases the size of *unclassified* by 1, and the time spent within each iteration is $O(1)$, implying the time complexity is $O(n)$. The space complexity is clearly $O(1)$.

Variant: Assuming that keys take one of three values, reorder the array so that all objects with the same key appear together. The order of the subarrays is not important. For example, both Figures 2.1(b) and 2.1(c) on Page 8 are valid answers for Figure 2.1(a) on Page 8. Use $O(1)$ additional space and $O(n)$ time.

Variant: Given an array A of n objects with keys that takes one of four values, reorder the array so that all objects that have the same key appear together. Use $O(1)$ additional space and $O(n)$ time.

Variant: Given an array A of n objects with Boolean-valued keys, reorder the array so that objects that have the key false appear first. Use $O(1)$ additional space and $O(n)$ time.

Variant: Given an array A of n objects with Boolean-valued keys, reorder the array so that objects that have the key false appear first. The relative ordering of objects with key true should not change. Use $O(1)$ additional space and $O(n)$ time.

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 61).

Strings boot camp

A palindromic string is one which reads the same when it is reversed. The program below checks whether a string is palindromic. Rather than creating a new string for the reverse of the input string, it traverses the input string forwards and backwards, thereby saving space. Notice how it uniformly handles even and odd length strings.

```
bool IsPalindromic(const string& s) {
  for (int i = 0, j = s.size() - 1; i < j; ++i, --j) {
    if (s[i] != s[j]) {
      return false;
    }
  }
  return true;
}
```

The time complexity is $O(n)$ and the space complexity is $O(1)$, where n is the length of the string.

Similar to arrays, string problems often have simple brute-force solutions that use $O(n)$ space solution, but subtler solutions that uses the string itself to **reduce space complexity** to $O(1)$.

Understand the **implications** of a string type that's **immutable**, e.g., the need to allocate a new string when concatenating immutable strings. Know **alternatives** to immutable strings, e.g., an array of characters or a `StringBuilder` in Java.

Updating a mutable string from the front is slow, so see if it's possible to **write values from the back**.

Know your string libraries

When manipulating C++ strings, you need to know the `string` class well.

- The basic methods are `append("Gauss")`, `push_back('c')`, `pop_back()`, `insert(s.begin() + shift, "Gauss")`, `substr(pos, len)`, and `compare("Gauss")`.
- Remember a string is organized like like an array. It performs well for operations from the back, e.g., `push_back('c')` and `pop_back()`, but poorly in the middle of a string, e.g., `insert(A.begin() + middle, "Gauss")`.
- The comparison operators <, <=, >, >=, and == can be applied to strings, with == testing logical equality, rather than pointer equality.

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.

Implement string/integer inter-conversion functions.

Hint: Build the result one digit at a time.

Solution: 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. The natural algorithm would be to prepend digits to the partial result. However, adding a digit to the beginning of a string is expensive, since all remaining digit have to be moved. A more time efficient approach is to add each computed digit to the end, and then reverse the computed sequence.

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;
  do {
    s += '0' + x % 10;
    x /= 10;
  } while (x);

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

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

3.2 Base conversion

In the decimal number 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 $3 \times 100 + 1 \times 10 + 4 \times 1$. 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 $a_0 \times b^0 + a_1 \times b^1 + a_2 \times b^2 + \cdots + a_{k-1} \times b^{k-1}$.

Write a program that performs base conversion. The input is a string, an integer b_1, and another integer b_2. The string represents be an integer in base b_1. The output should be the string representing the integer 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. (For example, if the string is "615", b_1 is 7 and b_2 is 13, then the result should be "1A7", since $6 \times 7^2 + 1 \times 7 + 5 = 1 \times 13^2 + 10 \times 13 + 7$.)

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

Solution: A brute-force approach might be to convert the input to a unary representation, and then group the 1s as multiples of b_2, b_2^2, b_2^3, etc. For example, $(102)_3 = (1111111111)_1$. To convert to base 4, there are two groups of 4 and with three 1s remaining, so the result is $(23)_4$. This approach is hard to implement, and has terrible time and space complexity.

The insight to a good algorithm is the fact that all languages have an integer type, which supports arithmetical operations like multiply, add, divide, modulus, etc. These operations make the conversion much easier. Specifically, we can convert a string in base b_1 to integer type using a sequence of multiply and adds. Then we convert that integer type to a string in base b_2 using a sequence of modulus and division operations. For example, for the string is "615", $b_1 = 7$ and $b_2 = 13$, then the integer value, expressed in decimal, is 306. The least significant digit of the result is 306 mod 13 = 7, and we continue with 306/13 = 23. The next digit is 23 mod 13 = 10, which we denote by 'A'. We continue with 23/13 = 1. Since 1 mod 13 = 1 and 1/13 = 0, the final digit is 1, and the overall 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;
  }
  return (is_negative ? "-" : "") + (x == 0 ? "0" : ConstructFromBase(x, b2));
}

string ConstructFromBase(int x, int base) {
  return x == 0 ? "" : ConstructFromBase(x / base, base) +
                       (char)(x % base >= 10 ? 'A' + x % base - 10
                                             : '0' + x % base);
}
```

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 $\log_{b_2}(b_1^n) = n \log_{b_2} b_1$.

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 null. 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.

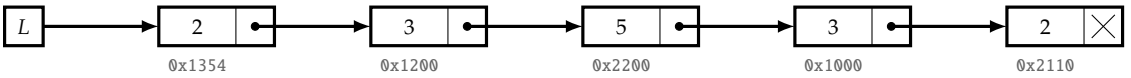


Figure 4.1: Example of a singly linked list. The number in hex below a node indicates the memory address of that node.

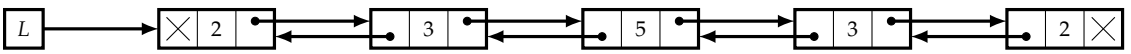


Figure 4.2: Example of a doubly linked list.

For all problems in this chapter, unless otherwise stated, 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:

```
template <typename T>
struct ListNode {
  T data;
  shared_ptr<ListNode<T>> next;
};
```

Linked lists boot camp

There are two types of list-related problems—those where you have to implement your own list, and those where you have to exploit the standard list library. We will review both these aspects here, starting with implementation, then moving on to list libraries.

Implementing a basic list API—search, insert, delete—for singly linked lists is an excellent way to become comfortable with lists.

Search for a key:

```
shared_ptr<ListNode<int>> SearchList(shared_ptr<ListNode<int>> L, int key) {
  while (L && L->data != key) {
    L = L->next;
  }
  // If key was not present in the list, L will have become null.
  return L;
}
```

Insert a new node after a specified node:

```
// Insert new_node after node.
void InsertAfter(const shared_ptr<ListNode<int>>& node,
                 const shared_ptr<ListNode<int>>& new_node) {
  new_node->next = node->next;
  node->next = new_node;
}
```

Delete a node:

```
// Delete the node past this one. Assume node is not tail.
void DeleteAfter(const shared_ptr<ListNode<int>>& node) {
  node->next = node->next->next;
}
```

Insert and delete are local operations and have $O(1)$ time complexity. Search requires traversing the entire list, e.g., if the key is at the last node or is absent, so its time complexity is $O(n)$, where n is the number of nodes.

List problems often have a simple brute-force solution that uses $O(n)$ space, but a subtler solution that uses the **existing list nodes** to reduce space complexity to $O(1)$.

Very often, a problem on lists is conceptually simple, and is more about **cleanly coding what's specified**, rather than designing an algorithm.

Consider using a **dummy head** (sometimes referred to as a sentinel) to avoid having to check for empty lists. This simplifies code, and makes bugs less likely.

It's easy to forget to **update next** (and previous for double linked list) for the head and tail.

Algorithms operating on singly linked lists often benefit from using **two iterators**, one ahead of the other, or one advancing quicker than the other.

Know your linked list libraries

We now review the standard linked list library, with the reminder that many interview problems that are directly concerned with lists require you to write your own list class. When manipulating C++ lists, you need to know the `list` and `forward_list` classes well. The `list` class is a doubly-linked list; `forward_list` is a singly-linked list.[1]

For doubly-linked lists, i.e., `list`, here are the functions you should know well.

- The functions to insert and delete elements in list are `push_front(42)` (or `emplace_front(42)`), `pop_front()`, `push_back()` (or `emplace_back()`, and `pop_back()`).

[1] A singly-linked list is more space efficient than doubly-linked list because nodes do not contain a previous field. The trade-off is the inability to do bidirectional iteration.

- The `splice(L1.end(), L2)`, `reverse()`, and `sort()` functions are analogous to those on `forward_list`.

For singly-linked lists, i.e., `forward_list`, here are the functions you should know well.

- The functions to insert and delete elements in list are `push_front(42)` (or `emplace_front(42)`), `pop_front()`, `insert_after(L.end(), 42)` (or `emplace_after(L.end(), 42)`, and `erase_after(A.begin())`).
- To transfer elements from list to another used `splice_after(L1.end(), L2)`.
- Reverse the order of the elements with `reverse()`.
- Use `sort()` to sort lists, and save yourself a great deal of pain.

4.1 Test for cyclicity

Although a linked list is supposed to be a sequence of nodes ending in 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.

Write a program that takes the head of a singly linked list and returns null if there does not exist a cycle, and the node at the start of the cycle, if a cycle is present. (You do not know the length of the list in advance.)

Hint: Consider using two iterators, one fast and one slow.

Solution: 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) {
    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: 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.
}
```

Chapter

5 Stacks and Queues

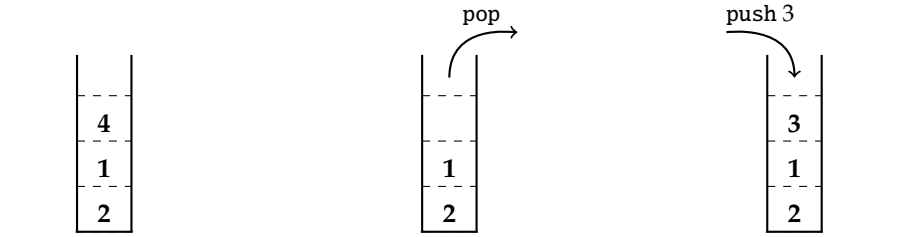
Linear lists in which insertions, deletions, and accesses to values occur almost always at the first or the last node are very frequently encountered, and we give them special names . . .

— "*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 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, which returns the top of the stack without popping it.



(a) Initial configuration. (b) Popping (a). (c) Pushing 3 on to (b).

Figure 5.1: Operations on a stack.

Stacks boot camp

The last-in, first-out semantics of a stack make it very useful for creating reverse iterators for sequences which are stored in a way that would make it difficult or impossible to step back from a given element. This a program uses a stack to print the entries of a singly-linked list in reverse order.

```
void PrintLinkedListInReverse(shared_ptr<ListNode<int>> head) {
  stack<int> nodes;
  while (head) {
    nodes.push(head->data);
```

```
    head = head->next;
  }
  while (!nodes.empty()) {
    cout << nodes.top() << endl;
    nodes.pop();
  }
}
```

The time and space complexity are $O(n)$, where n is the number of nodes in the list.

As an alternative, we could form the reverse of the list and then iterate through the list printing entries, then perform another reverse to recover the list—this would have $O(n)$ time complexity and $O(1)$ space complexity.

> Learn to recognize when the stack **LIFO** property is **applicable**. For example, **parsing** typically benefits from a stack.
>
> Consider **augmenting** the basic stack or queue data structure to support additional operations, such as finding the maximum element.

Know your stack libraries

When manipulating C++ stacks, you need to know the `stack` class well. The key functions in the `stack` class are `top()`, `push(42)` (or `emplace(42)`), and `pop()`. When called from an empty stack, `top()` and `pop()` throw exceptions.

- `push(e)` pushes an element onto the stack. Not much can go wrong with a call to push.
- `top()` will retrieve, but does not remove, the element at the top of the stack.
- `pop()` will remove and the element at the top of the stack but does not return. To avoid the exception, first test with `empty()`.
- `empty()` tests if the stack is empty.

5.1 Implement a stack with max API

Design a stack that includes a max operation, in addition to push and pop. The max method should return the maximum value stored in the stack.

Hint: Use additional storage to track the maximum value.

Solution: The simplest way to implement a max operation 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()) {
      throw length_error("Max(): empty stack");
    }
    return element_with_cached_max_.top().max;
  }

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

  void Push(int x) {
    element_with_cached_max_.emplace(
        ElementWithCachedMax{x, max(x, Empty() ? x : Max())});
  }

 private:
  struct ElementWithCachedMax {
    int element, max;
  };
  stack<ElementWithCachedMax> 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 5.2 on Page 24 for an example.

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

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

```
  int Pop() {
    if (Empty()) {
      throw length_error("Pop(): empty stack");
    }
    int pop_element = element_.top();
    element_.pop();
    const int current_max = cached_max_with_count_.top().max;
    if (pop_element == current_max) {
      int& max_frequency = cached_max_with_count_.top().count;
      --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(MaxWithCount{x, 1});
    } else {
      const int current_max = cached_max_with_count_.top().max;
      if (x == current_max) {
        int& max_frequency = cached_max_with_count_.top().count;
        ++max_frequency;
      } else if (x > current_max) {
        cached_max_with_count_.emplace(MaxWithCount{x, 1});
      }
    }
  }

 private:
  stack<int> element_;

  struct MaxWithCount {
    int max, count;
  };
  stack<MaxWithCount> cached_max_with_count_;
};
```

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 $O(1)$.

Queues

A *queue* supports two basic operations—enqueue and dequeue. (If the queue is empty, dequeue typically returns null or throws an exception.) Elements are added (enqueued) and removed (dequeued) in first-in, first-out order. The most recently inserted element is referred to as the tail or back element, and the item that was inserted least recently is referred to as the head or front element.

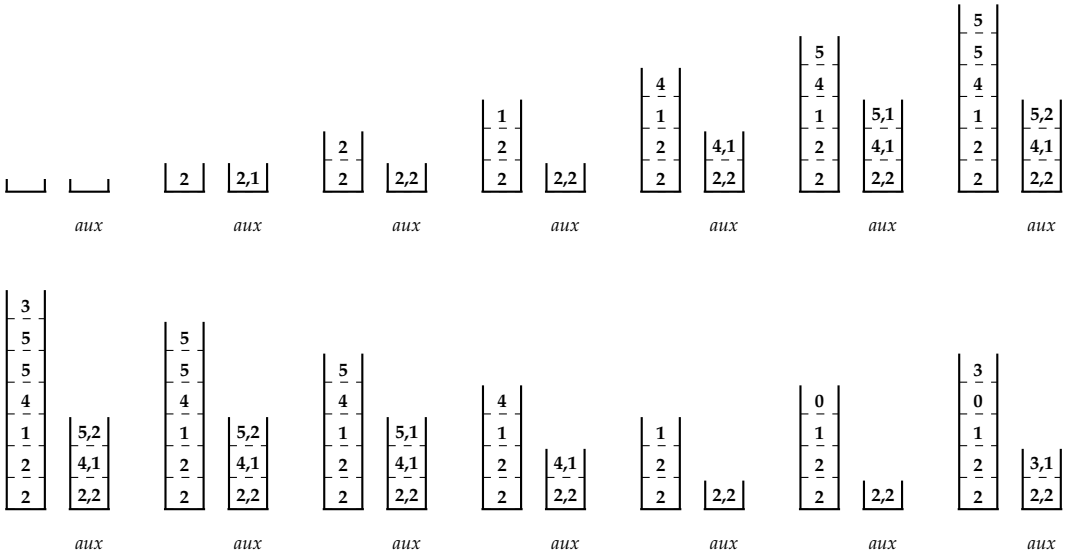
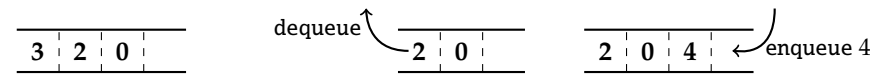


Figure 5.2: The primary and auxiliary stacks for the following operations: push 2, push 2, push 1, push 4, push 5, push 5, push 3, pop, pop, pop, pop, push 0, push 3. Both stacks are initially empty, and their progression is shown from left-to-right, then top-to-bottom. The top of the auxiliary stack 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*.

A queue can be implemented using a linked list, in which case these operations have $O(1)$ time complexity. The queue API often includes other operations, e.g., a method that returns the item at the head of the queue without removing it, a method that returns the item at the tail of the queue without removing it, etc.



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

Figure 5.3: Examples of enqueuing and dequeuing.

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 commonly called a push, and an insertion to the back is commonly called an inject. A deletion from the front is commonly called a pop, and a deletion from the back is commonly called an eject. (Different languages and libraries may have different nomenclature.)

Queues boot camp

In the following program, we implement the basic queue API—enqueue and dequeue—as well as a max-method, which returns the maximum element stored in the queue. The basic idea is to use composition: add a private field that references a library queue object, and forward existing methods (enqueue and dequeue in this case) to that object.

```
class Queue {
 public:
  void Enqueue(int x) { data_.emplace_back(x); }
```

```
  int Dequeue() {
    if (data_.empty()) {
      throw length_error("empty queue");
    }
    int val = data_.front();
    data_.pop_front();
    return val;
  }

  int Max() const {
    if (data_.empty()) {
      throw length_error("Cannot perform Max() on empty queue.");
    }
    return *max_element(data_.begin(), data_.end());
  }

 private:
  list<int> data_;
};
```

The time complexity of enqueue and dequeue are the same as that of the library queue, namely, $O(1)$. The time complexity of finding the maximum is $O(n)$, where n is the number of entries.

Learn to recognize when the queue FIFO property is applicable. For example, queues are ideal when order needs to be preserved.

Know your queue libraries

When manipulating C++ queues, you need to know the `queue` class well. The key functions in the `queue` class are `front()`, `back()`, `push(42)` (or `emplace(42)`, and `pop()`). When called on an empty queue, `front()`, `back()`, and `pop()` throw exceptions.

- `push(e)` pushes an element onto the queue. Not much can go wrong with a call to push.
- `front()` will retrieve, but does not remove, the element at the front of the queue. Similarly, `back()` will retrieve, but also does not remove, the element at the back of the queue.
- `pop()` will remove and the element at the top of the queue but does not return.

`deque` implements the double-ended queue in C++, and `push_back(123)` (or `emplace_back(123)`), `push_front()` (or `emplace_front(123)`), `pop_back()`, `pop_front()`, `front()`, and `back()` are the key functions.

5.2 Compute binary tree nodes 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.

Given a binary tree, return an array consisting of the keys at the same level. Keys should appear in the order of the corresponding nodes' depths, breaking ties from left to right. For example, you should return $\langle\langle 314\rangle, \langle 6, 6\rangle, \langle 271, 561, 2, 271\rangle, \langle 28, 0, 3, 1, 28\rangle, \langle 17, 401, 257\rangle, \langle 641\rangle\rangle$ for the binary tree in Figure 6.1 on Page 28.

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

Solution: 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 to record 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.

```
vector<vector<int>> BinaryTreeDepthOrder(
    const unique_ptr<BinaryTreeNode<int>>& tree) {
  queue<BinaryTreeNode<int>*> processing_nodes;
  processing_nodes.emplace(tree.get());
  int num_nodes_to_process_at_current_level = processing_nodes.size();
  vector<vector<int>> result;
  vector<int> one_level;

  while (!processing_nodes.empty()) {
    auto curr = processing_nodes.front();
    processing_nodes.pop();
    --num_nodes_to_process_at_current_level;
    if (curr) {
      one_level.emplace_back(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_to_process_at_current_level) {
      num_nodes_to_process_at_current_level = processing_nodes.size();
      if (!one_level.empty()) {
        result.emplace_back(move(one_level));
      }
    }
  }
  return result;
}
```

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: Write a program which takes as input a binary tree and returns the keys in top down, alternating left-to-right and right-to-left order, starting from left-to-right. For example, if the input is the tree in Figure 6.1 on Page 28, your program should return $\langle\langle 314\rangle, \langle 6, 6\rangle, \langle 271, 561, 2, 271\rangle, \langle 28, 1, 3, 0, 28\rangle, \langle 17, 401, 257\rangle, \langle 641\rangle\rangle$.

Variant: Write a program which takes as input a binary tree and returns the keys in a bottom up, left-to-right order. For example, if the input is the tree in Figure 6.1 on Page 28, your program should return $\langle\langle 641\rangle, \langle 17, 401, 257\rangle, \langle 28, 0, 3, 1, 28\rangle, \langle 271, 561, 2, 271\rangle, \langle 6, 6\rangle, \langle 314\rangle\rangle$.

Variant: Write a program which takes as input a binary tree with integer keys, and returns the average of the keys at each level. For example, if the input is the tree in Figure 6.1 on the following page, your program should return $\langle 314, 6, 276.25, 12, 225, 641 \rangle$.

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.

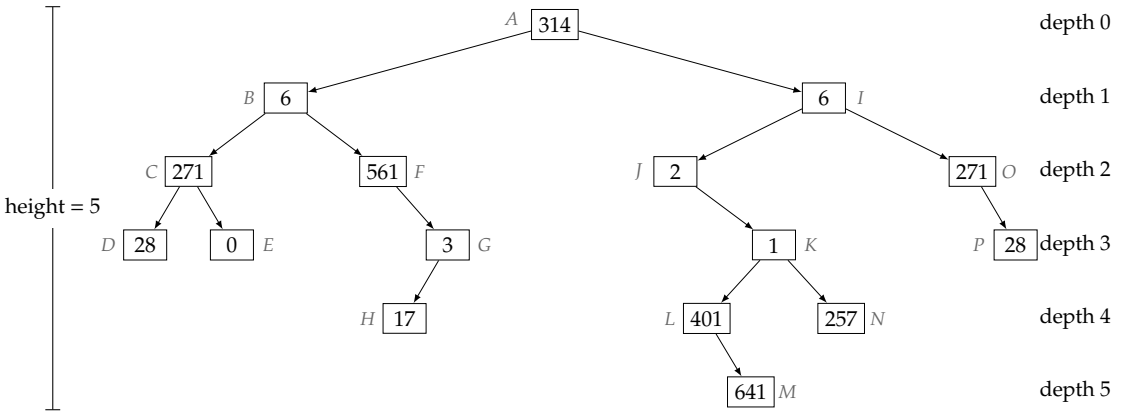


Figure 6.1: Example of a binary tree. The node depths range from 0 to 5. Node M has the highest depth (5) of any node in the tree, implying the height of the tree is 5.

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

```
template <typename T>
struct BinaryTreeNode {
  T data;
  unique_ptr<BinaryTreeNode<T>> left, right;
};
```

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 a node 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 facing 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 preceding 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 left-skewed tree is a tree in which no node has a right child; a right-skewed tree is a tree in which no node has a left child. In either case, we refer to the binary tree as being skewed.

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 facing 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 preceding 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 the facing page 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 of 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 72 for an overview of the common variants.

Binary trees boot camp

A good way to get up to speed with binary trees is to implement the three basic traversals—inorder, preorder, and postorder.

```
void TreeTraversal(const unique_ptr<BinaryTreeNode<int>>& root) {
  if (root) {
    // Preorder: Processes the root before the traversals of left and right
    // children.
    cout << "Preorder: " << root->data << endl;
    TreeTraversal(root->left);
    // Inorder: Processes the root after the traversal of left child and
    // before
    // the traversal of right child.
    cout << "Inorder: " << root->data << endl;
    TreeTraversal(root->right);
    // Postorder: Processes the root after the traversals of left and right
    // children.
    cout << "Postorder: " << root->data << endl;
  }
}
```

The time complexity of each approach is $O(n)$, where n is the number of nodes in the tree. Although no memory is explicitly allocated, the function call stack reaches a maximum depth of h, the height of the tree. Therefore, the space complexity is $O(h)$. The minimum value for h is $\lg n$ (complete binary tree) and the maximum value for h is n (skewed tree).

Recursive algorithms are well-suited to problems on trees. Remember to include space implicitly allocated on the **function call stack** when doing space complexity analysis.

Some tree problems have simple brute-force solutions that use $O(n)$ space solution, but subtler solutions that uses the **existing tree nodes** to reduce space complexity to $O(1)$.

Consider **left- and right-skewed trees** when doing complexity analysis. Note that $O(h)$ complexity, where h is the tree height, translates into $O(\log n)$ complexity for balanced trees, but $O(n)$ complexity for skewed trees.

If each node has a **parent field**, use it to make your code simpler, and to reduce time and space complexity.

It's easy to make the **mistake** of treating a node that has a `single child` as a leaf.

6.1 Test if a binary tree is height-balanced

A binary tree is said to be height-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 height-balanced, as is a complete binary tree. A height-balanced binary tree does not have to be perfect or complete—see Figure 6.2 on the facing page for an example.

Write a program that takes as input the root of a binary tree and checks whether the tree is height-balanced.

Hint: Think of a classic binary tree algorithm.

Solution: 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

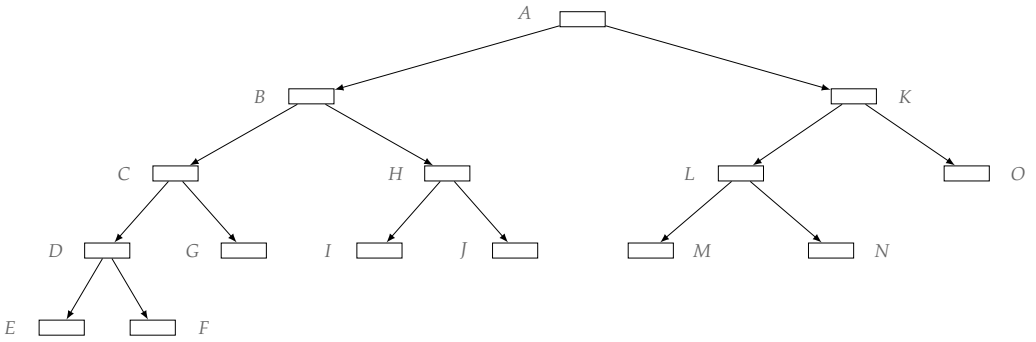


Figure 6.2: A height-balanced binary tree of height 4.

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 height-balanced, and if so, what its height is—we do not need any information about descendants of the subtree's root.

```
struct BalancedStatusWithHeight {
  bool balanced;
  int height;
};

bool IsBalanced(const unique_ptr<BinaryTreeNode<int>>& tree) {
  return CheckBalanced(tree).balanced;
}

// First value of the return value indicates if tree is balanced, and if
// balanced the second value of the return value is the height of tree.
BalancedStatusWithHeight CheckBalanced(
    const unique_ptr<BinaryTreeNode<int>>& tree) {
  if (tree == nullptr) {
    return {true, -1};  // Base case.
  }

  auto left_result = CheckBalanced(tree->left);
  if (!left_result.balanced) {
    return {false, 0};  // Left subtree is not balanced.
  }
  auto right_result = CheckBalanced(tree->right);
  if (!right_result.balanced) {
    return {false, 0};  // Right subtree is not balanced.
  }

  bool is_balanced = abs(left_result.height - right_result.height) <= 1;
  int height = max(left_result.height, right_result.height) + 1;
  return {is_balanced, height};
}
```

The program implements a postorder traversal with some calls possibly being eliminated because of early termination. Specifically, if any left subtree is not height-balanced 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: Write a program that returns the size of the largest subtree that is complete.

Variant: Define a node in a binary tree to be k-balanced if the difference in the number of nodes in its left and right subtrees is no more than k. Design an algorithm that takes as input a binary tree and positive integer k, and returns a node in the binary tree such that the node is not k-balanced, but all of its descendants are k-balanced. For example, when applied to the binary tree in Figure 6.1 on Page 28, if $k = 3$, your algorithm should return Node J.

Chapter

7 Heaps

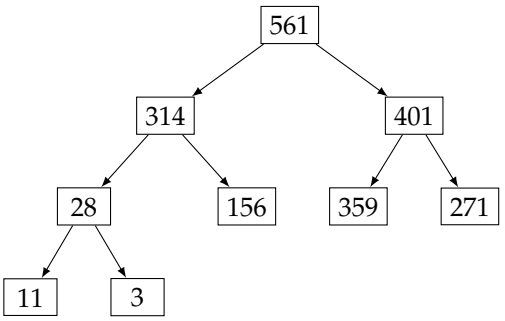
Using F-heaps we are able to obtain improved running times for several network optimization algorithms.

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

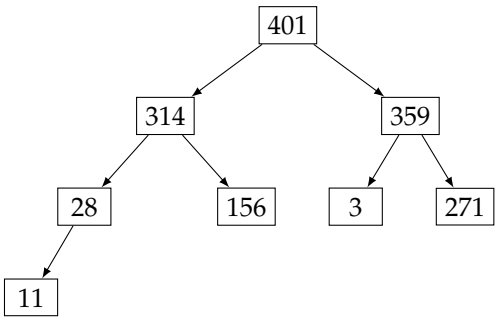
A *heap* (also referred to as a priority queue) is a specialized binary tree. Specifically, it is a complete binary tree as defined on Page 29. The keys must satisfy the *heap property*—the key at each node is at least as great as the keys stored at its children. See Figure 7.1(a) for an example of a max-heap. A max-heap can be implemented as an array; the children of the node at index i are at indices $2i+1$ and $2i+2$. The array representation for the max-heap in Figure 7.1(a) is $\langle 561, 314, 401, 28, 156, 359, 271, 11, 3\rangle$.

A max-heap 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. See Figure 7.1(b) for an example of deletion of the max element. Searching for arbitrary keys has $O(n)$ time complexity.

The *min-heap* is a completely symmetric version of the data structure and supports $O(1)$ time lookups for the minimum element.



(a) A max-heap. Note that the root hold the maximum key, 561.



(b) After the deletion of max of the heap in (a). Deletion is performed by replacing the root's key with the key at the last leaf and then recovering the heap property by repeatedly exchanging keys with children.

Figure 7.1: A max-heap and deletion on that max-heap.

Heaps boot camp

Suppose you were asked to write a program which takes a sequence of strings presented in "streaming" fashion: you cannot back up to read an earlier value. Your program must compute the k longest strings in the sequence. All that is required is the k longest strings—it is not required to order these strings.

As we process the input, we want to track the k longest strings seen so far. Out of these k strings, the string to be evicted when a longer string is to be added is the shortest one. A min-heap (not a max-heap!) is the right data structure for this application, since it supports efficient find-min, remove-min, and insert. In the program below we use a heap with a custom compare function, wherein strings are ordered by length.

```
vector<string> TopK(int k, istringstream* stream) {
  priority_queue<string, vector<string>, function<bool(string, string)>>
  min_heap([](const string& a, const string& b) -> bool {
    return a.size() >= b.size();
  });
  string next_string;
  while (*stream >> next_string) {
    min_heap.emplace(next_string);
    if (min_heap.size() > k) {
      // Remove the shortest string. Note that the comparison function above
      // will order the strings by length.
      min_heap.pop();
    }
  }
  vector<string> result;
  while (!min_heap.empty()) {
    result.emplace_back(min_heap.top());
    min_heap.pop();
  }
  return result;
}
```

Each string is processed in $O(\log k)$ time, which is the time to add and to remove the minimum element from the heap. Therefore, if there are n strings in the input, the time complexity to process all of them is $O(n \log k)$.

We could improve best-case time complexity by first comparing the new string's length with the length of the string at the top of the heap (getting this string takes $O(1)$ time) and skipping the insert if the new string is too short to be in the set.

> Use a heap when **all you care about** is the **largest** or **smallest** elements, and you **do not need** to support fast lookup, delete, or search operations for arbitrary elements.
>
> A heap is a good choice when you need to compute the k **largest or** k **smallest** elements in a collection. For the former, use a min-heap, for the latter, use a max-heap.

Know your heap libraries

The implementation of heaps in C++ is referred to as a priority queue; the class is `priority_queue`. The key functions are `push("Gauss")` (or `emplace("Gauss")`), `top()`, and `pop()`. Calling `top()` and `pop()` on an empty stack throws an exception. It is possible to specify a custom comparator in the heap constructor, as illustrated on on this page.

7.1 Merge sorted files

This problem is motivated by the following scenario. You are given 500 files, each containing stock trade information for an S&P 500 company. Each trade is encoded by a line in the following format: `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. In the abstract, we are trying to solve the following problem.

Write a program that takes as input a set of sorted sequences and computes the union of these sequences as a sorted sequence. For example, if the input is $\langle 3, 5, 7\rangle$, $\langle 0, 6\rangle$, and $\langle 0, 6, 28\rangle$, then the output is $\langle 0, 0, 3, 5, 6, 6, 7, 28\rangle$.

Hint: Which part of each sequence is significant as the algorithm executes?

Solution: A brute-force approach is to concatenate these sequences into a single array and then sort it. The time complexity is $O(n \log n)$, assuming there are n elements in total.

The brute-force approach does not use the fact that the individual sequences are sorted. We can take advantage of this fact by restricting our attention to the first remaining element in each sequence. Specifically, we repeatedly pick the smallest element amongst the first element of each of the remaining part of each of the sequences.

A min-heap is ideal for maintaining a collection of elements when we need to add arbitrary values and extract the smallest element.

For ease of exposition, we show how to merge sorted arrays, rather than files. As a concrete example, suppose there are three sorted arrays to be merged: $\langle 3, 5, 7\rangle$, $\langle 0, 6\rangle$, and $\langle 0, 6, 28\rangle$. For simplicity, we show the min-heap as containing entries from these three arrays. In practice, we need additional information for each entry, namely the array it is from, and its index in that array. (In 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.)

The min-heap is initialized to the first entry of each array, i.e., it is $\{3, 0, 0\}$. We extract the smallest entry, 0, and add it to the output which is $\langle 0\rangle$. Then we add 6 to the min-heap which is $\{3, 0, 6\}$ now. (We chose the 0 entry corresponding to the third array arbitrarily, it would be a perfectly acceptable to choose from the second array.) Next, extract 0, and add it to the output which is $\langle 0, 0\rangle$; then add 6 to the min-heap which is $\{3, 6, 6\}$. Next, extract 3, and add it to the output which is $\langle 0, 0, 3\rangle$; then add 5 to the min-heap which is $\{5, 6, 6\}$. Next, extract 5, and add it to the output which is $\langle 0, 0, 3, 5\rangle$; then add 7 to the min-heap which is $\{7, 6, 6\}$. Next, extract 6, and add it to the output which is $\langle 0, 0, 3, 5, 6\rangle$; assuming 6 is selected from the second array, which has no remaining elements, the min-heap is $\{7, 6\}$. Next, extract 6, and add it to the output which is $\langle 0, 0, 3, 5, 6, 6\rangle$; then add 28 to the min-heap which is $\{7, 28\}$. Next, extract 7, and add it to the output which is $\langle 0, 0, 3, 5, 6, 6, 7\rangle$; the min-heap is $\{28\}$. Next, extract 28, and add it to the output which is $\langle 0, 0, 3, 5, 6, 6, 7, 28\rangle$; now, all elements are processed and the output stores the sorted elements.

```
struct IteratorCurrentAndEnd {
  bool operator>(const IteratorCurrentAndEnd& that) const {
    return *current > *that.current;
  }

  vector<int>::const_iterator current;
  vector<int>::const_iterator end;
};

vector<int> MergeSortedArrays(const vector<vector<int>>& sorted_arrays) {
```

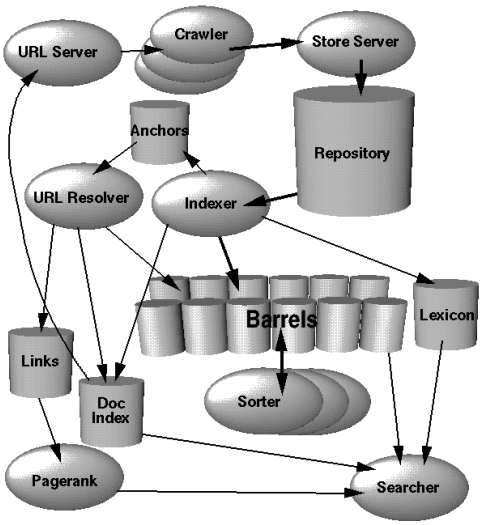
```
  priority_queue<IteratorCurrentAndEnd, vector<IteratorCurrentAndEnd>,
                 greater<>> min_heap;

  for (const vector<int>& sorted_array : sorted_arrays) {
    if (!sorted_array.empty()) {
      min_heap.emplace(
          IteratorCurrentAndEnd{sorted_array.cbegin(), sorted_array.cend()});
    }
  }

  vector<int> result;
  while (!min_heap.empty()) {
    auto smallest_array = min_heap.top();
    min_heap.pop();
    if (smallest_array.current != smallest_array.end) {
      result.emplace_back(*smallest_array.current);
      min_heap.emplace(IteratorCurrentAndEnd{next(smallest_array.current),
                                             smallest_array.end});
    }
  }
  return result;
}
```

Let k be the number of input sequences. Then 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. The space complexity is $O(k)$ beyond the space needed to write the final result. In particular, if the data comes from files and is written to a file, instead of arrays, we would need only $O(k)$ additional storage.

Alternatively, we could recursively merge the k files, two at a time using the merge step from merge sort. We would go from k to $k/2$ then $k/4$, etc. files. There would be $\log k$ stages, and each has time complexity $O(n)$, so the time complexity is the same as that of the heap-based approach, i.e., $O(n \log k)$. The space complexity of any reasonable implementation of merge sort would end up being $O(n)$, which is considerably worse than the heap based approach when $k \ll n$.

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 it 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, and 11.

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

Binary search

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.

```
int bsearch(int t, const vector<int>& A) {
  int L = 0, U = A.size() - 1;
  while (L <= U) {
    int M = (L + U) / 2;
    if (A[M] < t) {
      L = M + 1;
    } else if (A[M] == t) {
      return M;
    } else {
      U = M - 1;
    }
  }
  return -1;
}
```

The error is in the assignment `M = (L + U) / 2` in Line 4, which can potentially lead to overflow. This overflow can be avoided by using `M = L + (U - L) / 2`.

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.

Searching boot camp

When objects are comparable, they can be sorted and searched for using library search functions. Typically, the language knows how to compare built-in types, e.g., integers, strings, library classes for date, URLs, SQL timestamps, etc. However, user-defined types used in sorted collections must explicitly implement comparision, and ensure this comparision has basic properties such as transitivity. (If the comparision is implemented incorrectly, you may find a lookup into a sorted collection fails, even when the item is present.)

Suppose we are given as input an array of students, sorted by descending GPA, with ties broken on name. In the program below, we show how to use the library binary search routine to perform

fast searches in this array. In particular, we pass binary search a custom comparator which compares students on GPA (higher GPA comes first), with ties broken on name.

```
struct Student {
  string name;
  double grade_point_average;
};

const static function<bool(const Student&, const Student&)> CompGPA = [](
    const Student& a, const Student& b) -> bool {
  if (a.grade_point_average != b.grade_point_average) {
    return a.grade_point_average > b.grade_point_average;
  }
  return a.name < b.name;
};

bool SearchStudent(
    const vector<Student>& students, const Student& target,
    const function<bool(const Student&, const Student&)>& comp_GPA) {
  return binary_search(students.begin(), students.end(), target, comp_GPA);
}
```

Assuming the i-th element in the sequence can be accessed in $O(1)$ time, the time complexity of the program is $O(\log n)$.

Binary search is an effective search tool. It is applicable to more than just searching in **sorted arrays**, e.g., it can be used to search an **interval of real numbers or integers**.

If your solution uses sorting, and the computation performed after sorting is faster than sorting, e.g., $O(n)$ or $O(\log n)$, **look for solutions that do not perform a complete sort**.

Consider **time/space tradeoffs**, such as making multiple passes through the data.

Know your searching libraries

Searching is a very broad concept, and it is present in many data structures. For example `find(A.begin(), A.end(), target)` in `algorithm` header finds the first element in a STL container. Here we focus on binary search in a sorted STL container.

- To check a targeted value is presented, use `binary_search(A.begin(), A.end(), target)`. Note that it returns a boolean about the status of existence instead of the location.
- To find the first element that is not less than a targeted value, use `lower_bound(A.begin(), A.end(), target)`. In other words, it find the first element that is greater than or equal to the targeted value.
- To find the first element that is greater than a targeted value, use `upper_bound(A.begin(), A.end(), target)`.

All three functions above have a time complexity of $O(\log n)$ in a sorted STL container containing n elements. In an interview, if it is allowed, use the above functions, instead of implementing your own binary search.

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 that is equal to a specified 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.

Write a method that takes a sorted array and a key and returns the index of the *first* occurrence of that key in the array. For example, when applied to the array in Figure 8.1 your algorithm should return 3 if the given key is 108; if it is 285, your algorithm should return 6.

Hint: What happens when every entry equals k? Don't stop when you first see k.

Solution: A naive approach is to use binary search to find the index of any element equal to the key, k. (If k is not present, we simply return -1.) After finding such an element, we traverse backwards from it to find the first occurrence of that element. The binary search takes time $O(\log n)$, where n is the number of entries in the array. Traversing backwards takes $O(n)$ time in the worst-case—consider the case where entries are equal to k.

The fundamental idea of binary search is to maintain a set of candidate solutions. For the current problem, if we see the element at index i equals k, although we do not know whether i is the first element equal to k, we do know that no subsequent elements can be the first one. Therefore we remove all elements with index $i + 1$ or more from the candidates.

Let's apply the above logic to the given example, with $k = 108$. We start with all indices as candidates, i.e., with $[0, 9]$. The midpoint index, 4 contains k. Therefore we can now update the candidate set to $[0, 3]$, and record 4 as an occurrence of k. The next midpoint is 1, and this index contains -10. We update the candidate set to $[2, 3]$. The value at the midpoint 2 is 2, so we update the candidate set to $[3, 3]$. Since the value at this midpoint is 108, we update the first seen occurrence of k to 3. Now the interval is $[3, 2]$, which is empty, terminating the search—the result is 3.

```
int SearchFirstOfK(const vector<int>& A, int k) {
  int left = 0, right = A.size() - 1, result = -1;
  // [left : right] is the candidate set.
  while (left <= right) {
    int mid = left + ((right - left) / 2);
    if (A[mid] > k) {
      right = mid - 1;
    } else if (A[mid] == k) {
      result = mid;
      // Nothing to the right of mid can be the first occurrence of k.
      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 candidate set by half.

Variant: Design an efficient algorithm that takes a sorted array and a key, and finds the index of the *first* occurrence of an element greater than that key. For example, when applied to the array

in Figure 8.1 on the facing page your algorithm should return 9 if the key is 285; if it is -13, your algorithm should return 1.

Variant: 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: Write a program which takes a sorted array A of integers, and an integer k, and returns the interval enclosing k, i.e., the pair of integers L and U such that L is the first occurrence of k in A and U is the last occurrence of k in A. If k does not appear in A, return $[-1, -1]$. For example if $A = \langle 1, 2, 2, 4, 4, 4, 7, 11, 11, 13 \rangle$ and $k = 11$, you should return $[7, 8]$.

Variant: Write a program which tests if p is a prefix of a string in an array of sorted strings.

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 ("slots") 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.

A common mistake with hash tables is that a key that's present in a hash table will be updated. The consequence is that a lookup for that key will now fail, even though it's still in the hash table. If you have to update a key, first remove it, then update it, and finally, add it back—this ensures it's moved the the correct array location. As a rule, you should avoid using mutable objects as keys.

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. 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 (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 a tree data structure that is used to store a dynamic set of strings, has computational advantages. 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.

Hash tables boot camp

We introduce hash tables with two examples—one is an application that benefits from the algorithmic advantages of hash tables, and the other illustrates the design of a class that can be used in a hash table.

An application of hash tables

Anagrams are popular word play puzzles, whereby rearranging letters of one set of words, you get another set of words. For example, "eleven plus two" is an anagram for "twelve plus one". Crossword puzzle enthusiasts and Scrabble players benefit from the ability to view all possible anagrams of a given set of letters.

Suppose you were asked to write a program that takes as input a set of words and returns groups of anagrams for those words. Each group must contain at least two words.

For example, if the input is "debitcard", "elvis", "silent", "badcredit", "lives", "freedom", "listen", "levis", "money" then there are three groups of anagrams: (1.) "debitcard", "badcredit"; (2.) "elvis", "lives", "levis"; (3.) "silent", "listen". (Note that "money" does not appear in any group, since it has no anagrams in the set.)

Let's begin by considering the problem of testing whether one word is an anagram of another. Since anagrams do not depend on the ordering of characters in the strings, we can perform the test by sorting the characters in the string. Two words are anagrams if and only if they result in equal strings after sorting. For example, sort("logarithmic") and sort("algorithmic") are both "acghiilmort", so "logarithmic" and "algorithmic" are anagrams.

We can form the described grouping of strings by iterating through all strings, and comparing each string with all other remaining strings. If two strings are anagrams, we do not consider the second string again. This leads to an $O(n^2 m \log m)$ algorithm, where n is the number of strings and m is the maximum string length.

Looking more carefully at the above computation, note that the key idea is to map strings to a representative. Given any string, its sorted version can be used as a unique identifier for the anagram group it belongs to. What we want is a map from a sorted string to the anagrams it corresponds to. Anytime you need to store a set of strings, a hash table is an excellent choice. Our

final algorithm proceeds by adding sort(s) for each string s in the dictionary to a hash table. The sorted strings are keys, and the values are arrays of the corresponding strings from the original input.

```
vector<vector<string>> FindAnagrams(const vector<string>& dictionary) {
  unordered_map<string, vector<string>> sorted_string_to_anagrams;
  for (const string& s : dictionary) {
    // Sorts the string, uses it as a key, and then appends
    // the original string as another value into hash table.
    string sorted_str(s);
    sort(sorted_str.begin(), sorted_str.end());
    sorted_string_to_anagrams[sorted_str].emplace_back(s);
  }

  vector<vector<string>> anagram_groups;
  for (const auto& p : sorted_string_to_anagrams) {
    if (p.second.size() >= 2) {  // Found anagrams.
      anagram_groups.emplace_back(p.second);
    }
  }
  return anagram_groups;
}
```

The computation consists of n calls to sort and n insertions into the hash table. Sorting all the keys has time complexity $O(nm \log m)$. The insertions add a time complexity of $O(nm)$, yielding $O(nm \log m)$ time complexity in total.

Design of a hashable class

Consider a class that represents contacts. For simplicity, assume each contact is a string. Suppose it is a hard requirement that the individual contacts are to be stored in a list and it's possible that the list contains duplicates. Two contacts should be equal if they contain the same set of strings, regardless of the ordering of the strings within the underlying list. Multiplicity is not important, i.e., three repetitions of the same contact is the same as a single instance of that contact. In order to be able to store contacts in a hash table, we first need to explicitly define equality, which we can do by forming sets from the lists and comparing the sets.

In our context, this implies that the hash function should depend on the strings present, but not their ordering; it should also consider only one copy if a string appears in duplicate form. The hash function below forms a set from the contact list, which removes duplicates. It then XORs hashcodes for individual entries. Since XOR is commutative, the result does not depend on the order in which entries are processed. It should be pointed out that the hash function and equals methods below are very inefficient. In practice, it would be advisable to cache the underlying set and the hash code, remembering to void these values on updates.

```
struct ContactList {
  // Equal function for hash.
  bool operator==(const ContactList& that) const {
    return unordered_set<string>(names.begin(), names.end()) ==
           unordered_set<string>(that.names.begin(), that.names.end());
  }

  vector<string> names;
};
```

```
// Hash function for ContactList.
struct HashContactList {
  size_t operator()(const ContactList& contacts) const {
    size_t hash_code = 0;
    for (const string& name : unordered_set<string>(contacts.names.begin(),
                                                    contacts.names.end())) {
      hash_code ^= hash<string>()(name);
    }
    return hash_code;
  }
};

vector<ContactList> MergeContactLists(const vector<ContactList>& contacts) {
  unordered_set<ContactList, HashContactList> unique_contacts(
      contacts.begin(), contacts.end());
  return {unique_contacts.begin(), unique_contacts.end()};
}
```

The time complexity of computing the hash is $O(n)$, where n is the number of strings in the contact list. Hash codes are often cached for performance, with the caveat that the cache must be cleared if object fields that are referenced by the hash function are updated.

Hash tables have the **best theoretical and real-world performance** for lookup, insert and delete. Each of these operations has $O(1)$ time complexity. The $O(1)$ time complexity for insertion is for the average case—a single insert can take $O(n)$ if the hash table has to be resized.

Consider using a hash code as a **signature** to enhance performance, e.g., to filter out candidates.

Consider using a precomputed `lookup table` instead of boilerplate if-then code for mappings, e.g., from character to value, or character to character.

When defining your own type that will be put in a hash table, be sure you understand the relationship between **logical equality** and the fields the hash function must inspect. Specifically, anytime equality is implemented, it is imperative that the correct hash function is also implemented, otherwise when objects are placed in hash tables, logically equivalent objects may appear in different buckets, leading to lookups returning false, even when the searched item is present.

Sometimes you'll need a `multimap`, i.e., a map that contains multiple values for a single key, or a bi-directional map. If the language's standard libraries do not provide the functionality you need, learn how to implement a multimap using lists as values, or find a **third party library** that has a multimap.

Know your hash table libraries

There are two hash table-based data structures commonly used in C++—`unordered_set` and `unordered_map`. The difference between the two is that the latter stores key-value pairs, whereas the former simply stores keys. Both have the property that they do not allow for duplicate keys, unlike, for example, `list` and `priority_queue`.

The most important functions for `unordered_set` are `insert(42)` (or `emplace(42)`), `erase(42)`, `find(42)`, and `size()`.

- `insert(val)` inserts new element and returns a pair of iterator and boolean where the iterator points to the newly inserted element or the element whose key is equivalent, and the boolean indicating if the element was added successfully, i.e., was not already present.
- `find(k)` returns the iterator to the element if it was present; otherwise, a special iterator `end()` is returned.
- The order in which keys are traversed by the iterator returned by `begin()` is unspecified; it may even change with time.

The most important methods for `unordered_map` are `insert({42, "Gauss"})` (or `emplace({42, "Gauss"})`), `erase(42)`, `find(42)`, and `size()`. Those functions are analogous to the ones in `unordered_set`. The `pair<key, value>` type is a key-value pair that's useful when iterating over the map. The iteration order is not fixed, though iterations over the entry set, the key set, and the value set do agree.

The `hash()` method in `functional` header provides convenient functions for hashing the basic classes in C++, e.g., `int`, `bool`, `string`, `unique_ptr`, `shared_ptr`, etc. We show how to design a hash function for a custom class on Page 44.

9.1 Is an anonymous letter constructible?

Write a program which takes text for an anonymous letter and text for a magazine and determines if it is possible to write the anonymous letter using the magazine. The anonymous letter can be written using the magazine if for each character in the anonymous letter, the number of times it appears in the anonymous letter is no more than the number of times it appears in the magazine.

Hint: Count the number of distinct characters appearing in the letter.

Solution: A brute force approach is to count for each character in the character set the number of times it appears in the letter and in the magazine. If any character occurs more often in the letter than the magazine we return false, otherwise we return true. This approach is potentially slow because it iterates over all characters, including those that do not occur in the letter or magazine. It also makes multiple passes over both the letter and the magazine—as many passes as there are characters in the character set.

A better approach is to make a single pass over the letter, storing the character counts for the letter in a single hash table—keys are characters, and values are the number of times that character appears. Next, we make a pass over the magazine. When processing a character c, if c appears in the hash table, we reduce its count by 1; we remove it from the hash when its count goes to zero. If the hash becomes empty, we return true. If we reach the end of the letter and the hash is nonempty, we return false—each of the characters remaining in the hash occurs more times in the letter than the magazine.

```
bool IsLetterConstructibleFromMagazine(const string& letter_text,
                                       const string& magazine_text) {
  unordered_map<char, int> char_frequency_for_letter;
  // Compute the frequencies for all chars in letter_text.
  for (char c : letter_text) {
    ++char_frequency_for_letter[c];
  }

  // Check if the characters in magazine_text can cover characters
  // in char_frequency_for_letter.
```

```
  for (char c : magazine_text) {
    auto it = char_frequency_for_letter.find(c);
    if (it != char_frequency_for_letter.cend()) {
      --it->second;
      if (it->second == 0) {
        char_frequency_for_letter.erase(it);
        if (char_frequency_for_letter.empty()) {
          // All characters for letter_text are matched.
          break;
        }
      }
    }
  }
  // Empty char_frequency_for_letter means every char in letter_text can be
  // covered by a character in magazine_text.
  return char_frequency_for_letter.empty();
}
```

In the worst-case, the letter is not constructible or the last character of the magazine is essentially required. Therefore, the time complexity is $O(m + n)$ where m and n are the number of characters in the letter and magazine, respectively. The space complexity is the size of the hash table constructed in the pass over the letter, i.e., $O(L)$, where L is the number of distinct characters appearing in the letter.

If the characters are coded in ASCII, we could do away with the hash table 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.

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 34.

Sorting boot camp

It's important to know how to use effectively use the sort functionality provided by your language's library. Let's say we are given a student class that implements a compare method that compares students by name. To sort an array of students by GPA, we pass an explicit compare function to the sort function.

```
struct Student {
  bool operator<(const Student& that) const { return name < that.name; }

  string name;
  double grade_point_average;
};

void SortByGPA(vector<Student>* students) {
  sort(students->begin(), students->end(),
       [](const Student& a, const Student& b) -> bool {
         return a.grade_point_average >= b.grade_point_average;
       });
}

void SortByName(vector<Student>* students) {
  // Rely on the operator< defined in Student.
  sort(students->begin(), students->end());
}
```

The time complexity of any reasonable library sort is $O(n \log n)$ for an array with n entries. Most library sorting functions are based on quicksort, which has $O(1)$ space complexity.

Sorting problems come in two flavors: (1.) **use sorting to make subsequent steps in an algorithm simpler**, and (2.) design a **custom sorting routine**. For the former, it's fine to use a library sort function, possibly with a custom comparator. For the latter, use a data structure like a BST, heap, or array indexed by values.

The most natural reason to sort is if the inputs have a **natural ordering**, and sorting can be used as a preprocessing step to **speed up searching**.

For `specialized input`, e.g., a very small range of values, or a small number of values, it's possible to sort in $O(n)$ time rather than $O(n \log n)$ time.

It's often the case that sorting can be implemented in **less space** than required by a brute-force approach.

Know your sorting libraries

To sort an array, use `sort()` in the `algorithm` header, and to sort a list use the member function `list::sort()`.

- The time complexity of sorting is $O(n \log n)$, where n is length of the array. The standard gives no guarantees as to the space complexity. In practice, it's most commonly a variant of quicksort, which does not allocate additional memory, but uses $O(\log n)$ space on the function call stack.
- Both `sort()` in `algorithm` and `list::sort()` operate on arrays and lists of objects that implement `operator<()`.

- Both `sort()` in `algorithm` and `list::sort()` have the provision of sorting according to an explicit comparator object, as shown on on the previous page.

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.

Write a program which takes as input two sorted arrays, and returns a new array containing elements that are present in both of the input arrays. The input arrays may have duplicate entries, but the returned array should be free of duplicates. For example, the input is $\langle 2, 3, 3, 5, 5, 6, 7, 7, 8, 12\rangle$ and $\langle 5, 5, 6, 8, 8, 9, 10, 10\rangle$, your output should be $\langle 5, 6, 8\rangle$.

Hint: Solve the problem if the input array lengths differ by orders of magnitude. What if they are approximately equal?

Solution: 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. Let m and n be the lengths of the two input arrays.

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

The brute-force algorithm has $O(mn)$ time complexity.

Since both the arrays are sorted, we can make some optimizations. First, we can iterate through the first array and use binary search in array to test if the element is present in the second array.

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

```
}
```

The time complexity is $O(m \log n)$, where m is the length of the array being iterated over. We can further improve our run time by choosing the shorter array for the outer loop since if n is much smaller than m, then $n \log(m)$ is much smaller than $m \log(n)$.

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. We can achieve linear runtime by simultaneously advancing through the two input arrays in increasing order. At each iteration, if the array elements differ, the smaller one can be eliminated. If they are equal, we add that value to the intersection and advance both. (We handle duplicates by comparing the current element with the previous one.) For example, if the arrays are $A = \langle 2, 3, 3, 5, 7, 11 \rangle$ and $B = \langle 3, 3, 7, 15, 31 \rangle$, then we know by inspecting the first element of each that 2 cannot belong to the intersection, so we advance to the second element of A. Now we have a common element, 3, which we add to the result, and then we advance in both arrays. Now we are at 3 in both arrays, but we know 3 has already been added to the result since the previous element in A is also 3. We advance in both again without adding to the intersection. Comparing 5 to 7, we can eliminate 5 and advance to the fourth element in A, which is 7, and equal to the element that B's iterator holds, so it is added to the result. We then eliminate 11, and since no elements remain in A, we return $\langle 3, 7 \rangle$.

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

Since we spend $O(1)$ time per input array element, the time complexity for the entire algorithm is $O(m + n)$.

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 pre-specified constant), and each event's rectangle must have the same "height" (distance between the sides

parallel to the X-axis). Your task is to compute the maximum height an event rectangle can have. In essence, this is equivalent to the following problem.

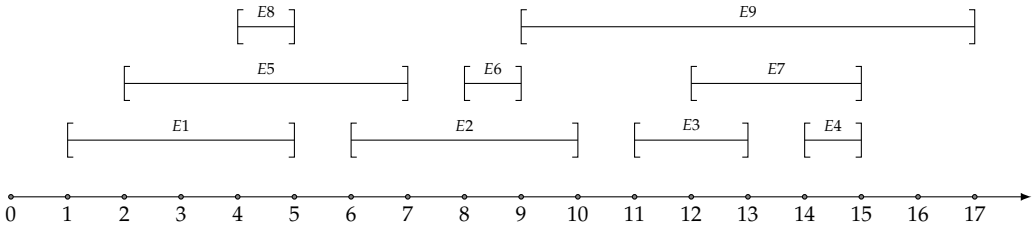


Figure 10.1: A set of nine events. The earliest starting event begins at time 1; the latest ending event ends at time 17. The maximum number of concurrent events is 3, e.g., $\{E1, E5, E8\}$ as well as others.

Write a program that takes a set of events, and determines the maximum number of events that take place concurrently.

Hint: Focus on endpoints.

Solution: The number of events scheduled for a given time changes only at times that are start or end times of an event. This leads the following brute-force algorithm. For each endpoint, compute the number of events that contain it. The maximum number of concurrent events is the maximum of this quantity over all endpoints. If there are n intervals, the total number of endpoints is $2n$. Computing the number of events containing an endpoint takes $O(n)$ time, since checking whether an interval contains a point takes $O(1)$ time. Therefore, the overall time complexity is $O(2n \times n) = O(n^2)$.

The inefficiency in the brute-force algorithm lies in the fact that it does not take advantage of locality, i.e., as we move from one endpoint to another. Intuitively, we can improve the run time 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.)

Now as we proceed through endpoints we can incrementally track the number of events taking place at that endpoint using a counter. 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.

For the example in Figure 10.1, the first seven endpoints are 1(start), 2(start), 4(start), 5(end), 5(end), 6(start The counter values are updated to 1, 2, 3, 2, 1, 2, 1.

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

struct Endpoint {
  bool operator<(const Endpoint& e) const {
    // If times are equal, an endpoint that starts an interval comes first.
    return time != e.time ? time < e.time : (isStart && !e.isStart);
  }

  int time;
  bool isStart;
};

int FindMaxSimultaneousEvents(const vector<Event>& A) {
```

```
  // Builds an array of all endpoints.
  vector<Endpoint> E;
  for (const Event& event : A) {
    E.emplace_back(Endpoint{event.start, true});
    E.emplace_back(Endpoint{event.finish, false});
  }
  // Sorts the endpoint array according to the time, breaking ties
  // by putting start times before end times.
  sort(E.begin(), E.end());

  // Track the number of simultaneous events, and record the maximum
  // number of simultaneous events.
  int max_num_simultaneous_events = 0, num_simultaneous_events = 0;
  for (const Endpoint& endpoint : E) {
    if (endpoint.isStart) {
      ++num_simultaneous_events;
      max_num_simultaneous_events =
          max(num_simultaneous_events, max_num_simultaneous_events);
    } else {
      --num_simultaneous_events;
    }
  }
  return max_num_simultaneous_events;
}
```

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: 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?

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, when the array needs to stay sorted. BSTs are similar to arrays in that the stored values (the "keys") are stored in a sorted order. BSTs offer the ability to search for a key as well as find the *min* and *max* elements, look for the successor or predecessor of a search key (which itself need not be present in the BST), and enumerate the keys in a range in sorted order. However, unlike with a sorted array, keys can be added to and deleted from a BST efficiently.

A BST is a binary tree as defined in Chapter 6 in which the nodes store keys that are comparable, e.g., integers or strings. 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 facing 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 that 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 height-balanced BSTs and are widely used in data structure libraries.

A common mistake with BSTs is that an object that's present in a BST is be updated. The consequence is that a lookup for that object will now fail, even though it's still in the BST. When a mutable object that's in a BST is to be updated, first remove it from the tree, then update it, then add it back. (As a rule, avoid putting mutable objects in a BST.)

The BST prototype is as follows:

```
template <typename T>
struct BSTNode {
  T data;
  unique_ptr<BSTNode<T>> left, right;
};
```

Binary search trees boot camp

Searching is the single most fundamental application of BSTs. Unlike a hash table, a BST offers the ability to find the min and max elements, and find the next largest/next smallest element. These operations, along with lookup, delete and find, take time $O(\log n)$ for library implementations of BSTs. Both BSTs and hash tables use $O(n)$ space—in practice, a BST uses slightly more space.

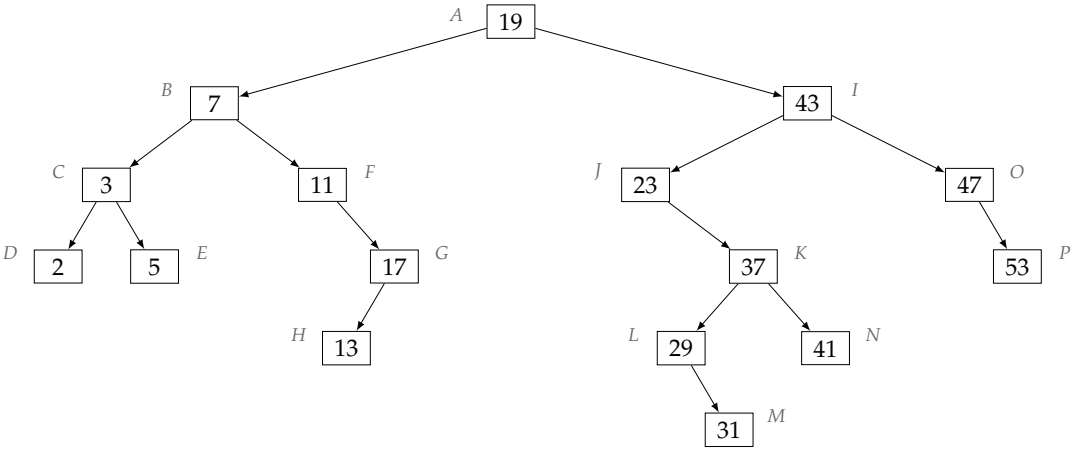


Figure 11.1: An example of a BST.

The following program demonstrates how to check if a given value is present in a BST. It is a nice illustration of the power of recursion when operating on BSTs.

```
BSTNode<int>* SearchBST(const unique_ptr<BSTNode<int>>& tree, int key) {
  if (tree == nullptr) {
    return nullptr;
  }
  if (tree->data == key) {
    return tree.get();
  }
  return key < tree->data ? SearchBST(tree->left, key)
                          : SearchBST(tree->right, key);
}
```

Since the program descends tree with in each step, and spends $O(1)$ time per level, the time complexity is $O(h)$, where h is the height of the tree.

With a BST you can **iterate** through elements in **sorted order** in time $O(n)$ (regardless of whether it is balanced).

Some problems need a **combination of a BST and a hashtable**. For example, if you insert student objects into a BST and entries are ordered by GPA, and then a student's GPA needs to be updated and all we have is the student's name and new GPA, we cannot find the student by name without a full traversal. However, with an additional hash table, we can directly go to the corresponding entry in the tree.

Sometimes, it's necessary to **augment** a BST, e.g., the number of nodes at a subtree in addition to its key, or the range of values sorted in the subtree.

The BST property is a **global property**—a binary tree may have the property that each node's key is greater than the key at its left child and smaller than the key at its right child, but it may not be a BST.

Know your binary search tree libraries

There are two BST-based data structures commonly used in C++—`set` and `map`. Similar to `unordered_set` and `unordered_map` on Page 45, `set` stores keys, and `map` stores key-value pairs.

Below, we describe the functionalities added by `set` that go beyond what's in `unordered_set`. The functionalities added by `map` are similar.

- The iterator returned by `begin()` traverses keys in ascending order. (To iterate over keys in descending order, use `rbegin()`.)
- `*begin()`/`*rbegin()` yield the smallest and largest keys in the BST.
- `lower_bound(12)`/`upper_bound(3)` return the first element that is greater than or equal to/-greater than the argument. For example, if the vector is ⟨1010102020203030⟩, then `lower_bound (v.begin(), v.end(), 20)` returns 3 and `upper_bound (v.begin(), v.end(), 20)` returns 6.
- `equal_range(10)` return the range of values equal to the argument.

It's particularly important to note that these operations take $O(\log n)$, since they are backed by the underlying tree.

The `set` and `map` constructors permit for an explicit comparator object that is used to order keys, and you should be comfortable with the syntax used to specify this object. (See on Page 49 for an example of comparator syntax.)

11.1 Test if a binary tree satisfies the BST property

Write a program that takes as input a binary tree and checks if the tree satisfies the BST property.

Hint: 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?

Solution: A direct approach, based on the definition of a BST, is to begin with the root, and compute the maximum key stored in the root's left subtree, and the minimum key in the root's right subtree. We check that the key at the root is greater than or equal to the maximum from the left subtree and less than or equal to the minimum from the right subtree. If both these checks pass, we recursively check the root's left and right subtrees. If either check fails, we return false.

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

The problem with this approach is that it will repeatedly traverse subtrees. In the worst-case, when the tree is a BST and each node's left child is empty, the 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 for the cache.

We now present two approaches which have $O(n)$ time complexity and $O(h)$ additional space complexity, where h is the height of the tree.

The first approach is to check constraints on the values for each subtree. The initial constraint comes from the root. Every node in its left (right) subtree must have a key less than or equal (greater than or equal) to the key at the root. This idea generalizes: if all nodes in a tree must have keys in the range $[l, u]$, and the key at the root is w (which itself must be between $[l, u]$, otherwise the requirement is violated at the root itself), then all keys in the left subtree must be in the range $[l, w]$, and all keys stored in the right subtree must be in the range $[w, u]$.

As a concrete example, when applied to the BST in Figure 11.1 on Page 55, the initial range is $[-\infty, \infty]$. For the recursive call on the subtree rooted at B, the constraint is $[-\infty, 19]$; the 19 is the upper bound required by A on its left subtree. For the recursive call starting at the subtree rooted at F, the constraint is $[7, 19]$. For the recursive call starting at the subtree rooted at K, the constraint is $[23, 43]$. The binary tree in Figure 6.1 on Page 28 is identified as not being a BST when the recursive call reaches C—the constraint is $[-\infty, 6]$, but the key at F is 271, so the tree cannot satisfy the BST property.

```
bool IsBinaryTreeBST(const unique_ptr<BinaryTreeNode<int>>& tree) {
  return AreKeysInRange(tree, numeric_limits<int>::min(),
                        numeric_limits<int>::max());
}

bool AreKeysInRange(const unique_ptr<BinaryTreeNode<int>>& tree,
                    int low_range, int high_range) {
  if (tree == nullptr) {
    return true;
  } else if (tree->data < low_range || tree->data > high_range) {
    return false;
  }

  return AreKeysInRange(tree->left, low_range, tree->data) &&
         AreKeysInRange(tree->right, tree->data, high_range);
}
```

Alternatively, we can use the fact that an inorder traversal visits keys in sorted order. Furthermore, if an inorder traversal of a binary tree visits keys in sorted order, then that binary tree must be a BST. (This follows directly from the definition of a BST and the definition of an inorder walk.) Thus we can check the BST property by performing an inorder traversal, recording the key stored at the last visited node. Each time a new node is visited, its key is compared with the key of the previously visited node. If at any step in the walk, the key at the previously visited node is greater than the node currently being visited, we have a violation of the BST property.

All these approaches 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, thereby reducing the time complexity when the property is violated at a node whose depth is small.

Specifically, we use a queue, where 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 ∞. We iteratively check the constraint on each node. If it violates the constraint we stop—the BST property has been violated. Otherwise, we add its children along with the corresponding constraint.

For the example in Figure 11.1 on Page 55, we initialize the queue with $(A, [-\infty, \infty])$. Each time we pop a node, we first check the constraint. We pop the first entry, $(A, [-\infty, \infty])$, and add its children, with the corresponding constraints, i.e., $(B, [-\infty, 19])$ and $(I, [19, \infty])$. Next we pop $(B, [-\infty, 19])$, and add its children, i.e., $(C, [-\infty, 7])$ and $(D, [7, 19])$. Continuing through the nodes, we check that all nodes satisfy their constraints, and thus verify the tree is a BST.

If the BST property is violated in a subtree consisting of nodes within a particular depth, the violation will be discovered without visiting any nodes at a greater depth. This is because each time we enqueue an entry, the lower and upper bounds on the node's key are the tightest possible.

```
struct QueueEntry {
  const unique_ptr<BinaryTreeNode<int>>& tree_node;
  int lower_bound, upper_bound;
};

bool IsBinaryTreeBST(const unique_ptr<BinaryTreeNode<int>>& tree) {
  queue<QueueEntry> BFS_queue;
  BFS_queue.emplace(QueueEntry{tree, numeric_limits<int>::min(),
                               numeric_limits<int>::max()});

  while (!BFS_queue.empty()) {
    if (BFS_queue.front().tree_node.get()) {
      if (BFS_queue.front().tree_node->data < BFS_queue.front().lower_bound ||
          BFS_queue.front().tree_node->data > BFS_queue.front().upper_bound) {
        return false;
      }

      BFS_queue.emplace(QueueEntry{BFS_queue.front().tree_node->left,
                                   BFS_queue.front().lower_bound,
                                   BFS_queue.front().tree_node->data});
      BFS_queue.emplace(QueueEntry{BFS_queue.front().tree_node->right,
                                   BFS_queue.front().tree_node->data,
                                   BFS_queue.front().upper_bound});
    }
    BFS_queue.pop();
  }
  return true;
}
```

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

Recursion boot camp

The Euclidean algorithm for calculating the greatest common divisor (GCD) of two numbers is a classic example of recursion. The central idea is that if $y > x$, the GCD of x and y is the GCD of x and $y - x$. For example, GCD(156, 36) = GCD((156 − 36) = 120, 36). By extension, this implies that the GCD of x and y is the GCD of x and y mod x, i.e., GCD(156, 36) = GCD((156 mod 36) = 12, 36) = GCD(12, 36 mod 12 = 0) = 12.

```
long long GCD(long long x, long long y) { return y == 0 ? x : GCD(y, x % y); }
```

Since with each recursive step one of the arguments is at least halved, it means that the time complexity is $O(\log \max(x, y))$. Put another way, the time complexity is $O(n)$, where n is the number of bits needed to represent the inputs. The space complexity is also $O(n)$, which is the maximum depth of the function call stack. (The program is easily converted to one which loops, thereby reducing the space complexity to $O(1)$.)

11.2 Generate 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 $\{0, 1, 2\}$ is graphically illustrated in Figure 11.2 on the facing page.

Write a function that takes as input a set and returns its power set.

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

Recursion is especially suitable when the **input is expressed using recursive rules**.

Recursion is a good choice for **search**, **enumeration**, and **divide-and-conquer**.

Use recursion as **alternative to deeply nested iteration** loops. For example, recursion is much better when you have an undefined number of levels, such as the IP address problem generalized to k substrings.

If you are asked to **remove recursion** from a program, consider mimicking call stack with the **stack data structure**. bt-traversal

Recursion can be easily removed from a **tail-recursive** program by using a while-loop—no stack is needed. (Optimizing compilers do this.)

If a recursive function may end up being called with the **same arguments** more than once, **cache** the results—this is the idea behind Dynamic Programming (Chapter 12).

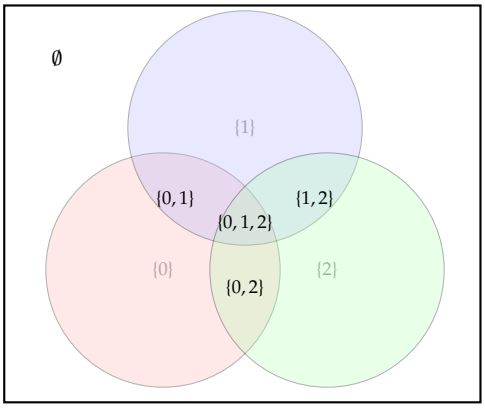


Figure 11.2: The power set of $\{0, 1, 2\}$ is $\{\emptyset, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{1, 2\}, \{0, 2\}, \{0, 1, 2\}\}$.

Solution: A brute-force way is to compute all subsets U that do not include a particular element (which could be any single element). Then we compute all subsets V which do include that element. Each subset set must appear in U or in V, so the final result is just $U \cup V$. The construction is recursive, and the base case is when the input set is empty, in which case we return $\{\{\}\}$.

As an example, let $S = \{0, 1, 2\}$. Pick any element, e.g., 0. First, we recursively compute all subsets of $\{1, 2\}$. To do this, we select 1. This leaves us with $\{2\}$. Now we pick 2, and get to a base case. So the set of subsets of $\{2\}$ is $\{\}$ union with $\{2\}$, i.e., $\{\{\}, \{2\}\}$. The set of subsets of $\{1, 2\}$ then is $\{\{\}, \{2\}\}$ union with $\{\{1\}, \{1, 2\}\}$, i.e., $\{\{\}, \{2\}, \{1\}, \{1, 2\}\}$. The set of subsets of $\{0, 1, 2\}$ then is $\{\{\}, \{2\}, \{1\}, \{1, 2\}\}$ union with $\{\{0\}, \{0, 2\}, \{0, 1\}, \{0, 1, 2\}\}$, i.e., $\{\{\}, \{2\}, \{1\}, \{1, 2\}, \{0\}, \{0, 2\}, \{0, 1\}, \{0, 1, 2\}\}$,

```
vector<vector<int>> GeneratePowerSet(const vector<int>& input_set) {
  vector<vector<int>> power_set;
  DirectedPowerSet(input_set, 0, new vector<int>, &power_set);
  return power_set;
}

// Generate all subsets whose intersection with input_set[0], ...,
// input_set[to_be_selected - 1] is exactly selected_so_far.
void DirectedPowerSet(const vector<int>& input_set, int to_be_selected,
                      vector<int>* selected_so_far,
```

```
                      vector<vector<int>>* power_set) {
  if (to_be_selected == input_set.size()) {
    power_set->emplace_back(*selected_so_far);
    return;
  }
  // Generate all subsets that contain input_set[to_be_selected].
  selected_so_far->emplace_back(input_set[to_be_selected]);
  DirectedPowerSet(input_set, to_be_selected + 1, selected_so_far, power_set);
  // Generate all subsets that do not contain input_set[to_be_selected].
  selected_so_far->pop_back();
  DirectedPowerSet(input_set, to_be_selected + 1, selected_so_far, power_set);
}
```

The number of recursive calls, $C(n)$ satisfies the recurrence $C(n) = 2C(n-1)$, which solves to $C(n) = O(2^n)$. Since we spend $O(n)$ time within a call, the time complexity is $O(n2^n)$.

A drawback of the above approach is that its space complexity is $O(2^n)$, even if we just want to print the subsets, rather than returning all of them. A more incremental way to enumerate the subsets is to recognize 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 $S = \{a, b, c, d\}$, the bit array $\langle 1, 0, 1, 1\rangle$ denotes the subset $\{a, c, d\}$.

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 $\lg y$.

```
vector<vector<int>> GeneratePowerSet(const vector<int>& input_set) {
  vector<vector<int>> power_set;
  for (int int_for_subset = 0; int_for_subset < (1 << input_set.size());
       ++int_for_subset) {
    int bit_array = int_for_subset;
    vector<int> subset;
    while (bit_array) {
      subset.emplace_back(input_set[log2(bit_array & ~(bit_array - 1))]);
      bit_array &= bit_array - 1;
    }
    power_set.emplace_back(subset);
  }
  return power_set;
}
```

Since each set takes $O(n)$ time to compute, the time complexity is $O(n2^n)$. In practice, this approach is very fast. Furthermore, its space complexity is $O(n)$ when we want to just enumerate subsets, e.g., to print them, rather that to return a list of all subsets.

Variant: Solve this problem 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 $\Big\langle\langle\rangle, \langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 2\rangle, \langle 2, 3\rangle, \langle 1, 2, 2\rangle, \langle 1, 2, 3\rangle, \langle 2, 2, 3\rangle, \langle 1, 2, 2, 3\rangle\Big\rangle$.

Chapter

12 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 optimization, search, and counting problems that can be decomposed into subproblems. You should consider using DP whenever you have to make choices to arrive at the solution, specifically, when the solution relates to subproblems.

Like divide-and-conquer, DP solves the problem by combining the solutions of multiple smaller problems, but what makes DP different is that the same subproblem may reoccur. Therefore, a key to making DP efficient is caching the results of intermediate computations. Problems whose solutions use DP are a popular choice for hard interview questions.

To illustrate the idea underlying DP, consider the problem of computing Fibonacci numbers. The first two Fibonacci numbers are 0 and 1. Successive numbers are the sums of the two previous numbers. The first few Fibonacci numbers are $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$. The Fibonacci numbers arise in many diverse applications—biology, data structure analysis, and parallel computing are some examples.

Mathematically, the nth Fibonacci number $F(n)$ is given by the equation $F(n) = F(n-1) + F(n-2)$, with $F(0) = 0$ and $F(1) = 1$. A function to compute $F(n)$ that recursively invokes itself has a time complexity that is exponential in n. This is because the recursive function computes some $F(i)$s repeatedly. Figure 12.1 on the next page graphically illustrates how the function is repeatedly called with the same arguments.

Caching intermediate results makes the time complexity for computing the nth Fibonacci number linear in n, albeit at the expense of $O(n)$ storage. The program below computes $F(n)$ via iteration in $O(n)$ time. Compared to a recursive program with caching, the iterative program fills in the cache in a bottom-up fashion, and reuses storage to reduce space complexity to $O(1)$.

```
unordered_map<int, int> cache;

int Fibonacci(int n) {
  if (n <= 1) {
    return n;
  } else if (!cache.count(n)) {
    cache[n] = Fibonacci(n - 1) + Fibonacci(n - 2);
  }
  return cache[n];
}
```

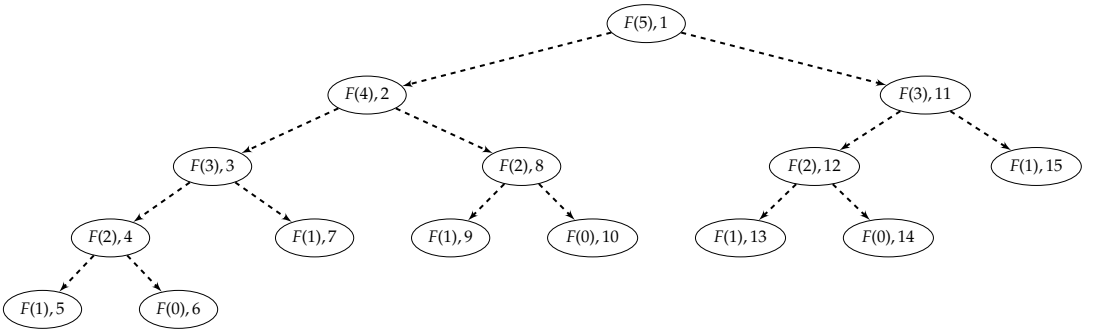


Figure 12.1: Tree of recursive calls when naively computing the 5th Fibonacci number, $F(5)$. Each node is a call: $F(x)$ indicates a call with argument x, and the italicized numbers on the right are the sequence in which calls take place. The children of a node are the subcalls made by that call. Note how there are 2 calls to $F(3)$, and 3 calls to each of $F(2)$, $F(1)$, and $F(0)$.

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

- the original problem can be solved relatively easily once solutions to the subproblems are available, and
- these subproblem solutions are cached.

Usually, but not always, the subproblems are easy to identify.

Here is a more sophisticated application of DP. Consider the following problem: find the maximum sum over all subarrays of a given array of integer. As a concrete example, the maximum subarray for the array in Figure 12.2 starts at index 0 and ends at index 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 12.2: 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 takes $O(n)$ time to compute. The brute-force algorithm can be improved to $O(n^2)$, at the cost of $O(n)$ additional storage, by first computing $S[k] = \sum A[0:k]$ for all k. The sum for $A[i:j]$ is then $S[j] - S[i-1]$, where $S[-1]$ is taken to be 0.

Here is a natural divide-and-conquer algorithm. Take $m = \lfloor \frac{n}{2} \rfloor$ to be the middle index of A. Solve the problem for the subarrays $L = A[0:m]$ and $R = A[m+1:n-1]$. In addition to the answers for each, we also return the maximum subarray sum l for a subarray ending at the last entry in L, and the maximum subarray sum r for a subarray starting at the first entry of 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, and the time complexity is $O(n \log n)$. Because of off-by-one errors, it takes some time to get the program just right.

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 $A[0:n-1]$. Instead we need to know the subarray amongst all subarrays $A[0:i]$, $i < n-1$, with the smallest subarray sum. The desired value is $S[n-1]$ minus this subarray's sum. We compute this value by iterating through

the array. For each index j, the maximum subarray sum ending at j is equal to $S[j] - \min_{k \le j} S[k]$. During the iteration, we track the minimum $S[k]$ we have seen so far and compute the maximum subarray sum for each index. The time spent per index is constant, leading to an $O(n)$ time and $O(1)$ space solution. It is legal for all array entries to be negative, or the array to be empty. The algorithm handles these input cases correctly.

```
int FindMaximumSubarray(const vector<int>& A) {
  int 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;
    }
    if (sum - min_sum > max_sum) {
      max_sum = sum - min_sum;
    }
  }
  return max_sum;
}
```

Here are some common mistakes made when applying DP.

- 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., 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.
- Make sure that **combining solutions** to the subproblems does **yield a solution** to the original problem. For example, if the longest path without repeated cities from City 1 to City 2 passes through City 3, then the subpaths from City 1 to City 3 and City 3 to City 2 may not be individually longest paths without repeated cities.

Dynamic programming boot camp

The programs presented in the introduction for computing the Fibonacci numbers and the maximum subarray sum are good illustrations of DP.

Consider using DP whenever you have to **make choices** to arrive at the solution, specifically, when the solution relates to subproblems.

In addition to optimization problems, DP is also **applicable** to **counting and decision problems**—any problem where you can express a solution recursively in terms of the same computation on smaller instances.

Although conceptually DP involves recursion, often for efficiency the cache is **built "bottom-up"**, i.e., iteratively.

To save space, `cache space` may be `recycled` once it is know that a set of entries will not be looked up again.

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

In this problem you are to count the number of ways of starting at the top-left corner of a 2D array and getting to the bottom-right corner. All moves must either go right or down. For example, we

show three ways in a 5×5 2D array in Figure 12.3. (As we will see, there are a total of 70 possible ways for this example.)

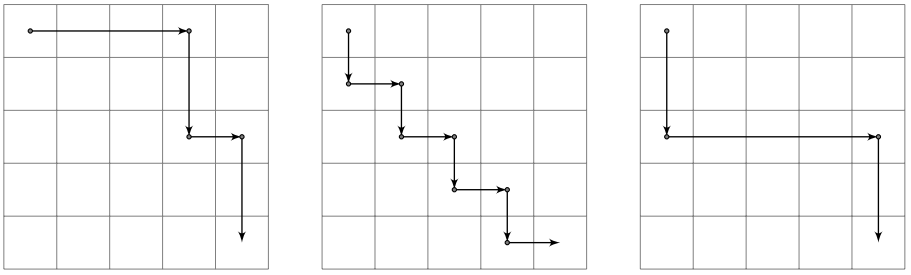


Figure 12.3: Paths through a 2D array.

Write a program that counts how many ways you can go from the top-left to the bottom-right in a 2D array.

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

Solution: A brute-force approach is to enumerate all possible paths. This can be done using recursion. However, there is a combinatorial explosion in the number of paths, which leads to huge time complexity.

The problem statement asks for the number of paths, so we focus our attention on that. A key observation is that because paths must advance down or right, the number of ways to get to the bottom-right entry is the number of ways to get to the entry immediately above it, plus the number of ways to get to the entry immediately to its left. Let's treat the origin $(0, 0)$ as the top-left entry. Generalizing, 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) from the origin.) This is the basis for a recursive algorithm to count the number of paths. Implemented naively, the algorithm has exponential time complexity—it repeatedly recurses on the same locations. The solution is to cache results. For example, the number of ways to get to (i, j) for the configuration in Figure 12.3 is cached in a matrix as shown in Figure 12.4 on the next page.

```
int NumberOfWays(int n, int m) {
  vector<vector<int>> number_of_ways(n, vector<int>(m, 0));
  return ComputeNumberOfWaysToXY(n - 1, m - 1, &number_of_ways);
}

int ComputeNumberOfWaysToXY(int x, int y,
                            vector<vector<int>>* number_of_ways_ptr) {
  if (x == 0 && y == 0) {
    return 1;
  }

  vector<vector<int>>& number_of_ways = *number_of_ways_ptr;
  if (number_of_ways[x][y] == 0) {
    int ways_top =
        x == 0 ? 0 : ComputeNumberOfWaysToXY(x - 1, y, number_of_ways_ptr);
    int ways_left =
        y == 0 ? 0 : ComputeNumberOfWaysToXY(x, y - 1, number_of_ways_ptr);
    number_of_ways[x][y] = ways_top + ways_left;
  }
  return number_of_ways[x][y];
```

```
}
```

The time complexity is $O(nm)$, and the space complexity is $O(nm)$, where n is the number of rows and m is the number of columns.

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

A more analytical way of solving this problem 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.

Variant: Solve the same problem using $O(\min(n, m))$ space.

Variant: Solve the same problem in the presence of obstacles, specified by a Boolean 2D array, where the presence of a true value represents an obstacle.

Variant: A fisherman is in a rectangular sea. The value of the fish at point (i, j) in the sea is specified by an $n \times m$ 2D array A. Write a program that computes the maximum value of fish a fisherman can catch on a path from the upper leftmost point to the lower rightmost point. The fisherman can only move down or right, as illustrated in Figure 12.5.

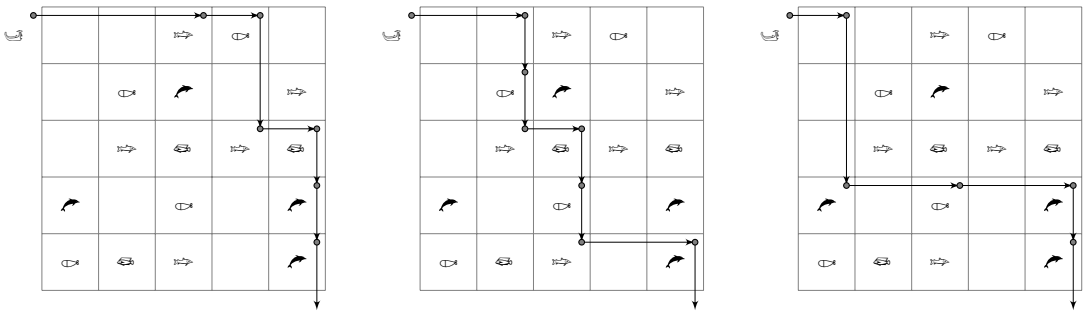


Figure 12.5: Alternate paths for a fisherman. Different types of fish have different values, which are known to the fisherman.

Variant: Solve the same problem when the fisherman can begin and end at any point. He must still move down or right. (Note that the value at (i, j) may be negative.)

Variant: 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 program which takes as input a positive integer k and computes the number of decimal numbers of length k that are monotone.

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

Chapter

13

Greedy Algorithms and Invariants

The intended function of a program, or part of a program, can be specified by 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

Greedy algorithms boot camp

For US currency, wherein coins take values 1, 5, 10, 25, 50, 100 cents, the greedy algorithm for making change results in the minimum number of coins. Here is an implementation of this algorithm. Note that once it selects the number of coins of a particular value, it never changes that selection; this is the hallmark of a greedy algorithm.

```
int ChangeMaking(int cents) {
  const array<int, 6> kCoins = {100, 50, 25, 10, 5, 1};
  int num_coins = 0;
  for (int i = 0; i < kCoins.size(); i++) {
    num_coins += cents / kCoins[i];
    cents %= kCoins[i];
  }
  return num_coins;
}
```

We perform 6 iterations, and each iteration does a constant amount of computation, so the time complexity is $O(1)$.

A greedy algorithm is often the right choice for an **optimization problem** where there's a natural set of **choices to select from**.

It's often easiest to conceptualize a greedy algorithm recursively, and then **implement** it using iteration for higher performance.

Even if the greedy approach does not yield an optimum solution, it can give insights into the optimum algorithm, or serve as a heuristic.

Sometimes the right greedy choice is **not obvious**.

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. For example, binary search, maintains

the invariant that the space of candidate solutions contains all possible solutions as the algorithm executes.

Sorting algorithms nicely illustrate 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, their elements are less than or equal to the remaining elements, and the entire array remains a permutation of the original array.

Invariants boot camp

Suppose you were asked to write a program that takes as input a sorted array and a given value and determines if there are two entries in the array that add up to that value. For example, if the array is $\langle -2, 1, 2, 4, 7, 11 \rangle$, then there are entries adding to 6 and to 10, but not to 0 and 13.

There are several ways to solve this problem: iterate over all pairs, or for each array entry search for the given value minus that entry. The most efficient approach uses invariants: maintain a subarray that is guaranteed to hold a solution, if it exists. This subarray is initialized to the entire array, and iteratively shrunk from one side or the other. The shrinking makes use of the sortedness of the array. Specifically, if the sum of the leftmost and the rightmost elements is less than the target, then the leftmost element can never be combined with some element to obtain the target. A similar observation holds for the rightmost element.

```
bool HasTwoSum(vector<int> A, int t) {
  int i = 0, j = A.size() - 1;
  while (i <= j) {
    if (A[i] + A[j] == t) {
      return true;
    } else if (A[i] + A[j] < t) {
      ++i;
    } else {  // A[i] + A[j] > t.
      --j;
    }
  }
  return false;
}
```

The time complexity is $O(n)$, where n is the length of the array. The space complexity is $O(1)$, since the subarray can be represented by two variables.

> The key strategy to determine whether to use an invariant when designing an algorithm is to work on **small examples** to hypothesize the invariant.
>
> Often, the invariant is a subset of the set of input space, e.g,. a subarray.

13.1 The 3-sum problem

Design an algorithm that takes as input an array and a number, and determines if there are three entries in the array (not necessarily distinct) which add up to the specified number. For example, if the array is $\langle 11, 2, 5, 7, 3 \rangle$ then there are three entries in the array which add up to 21 ($3, 7, 11$ and $5, 5, 11$). (Note that we can use 5 twice, since the problem statement said we can use the same entry more than once.) However, no three entries add up to 22.

Hint: How would you check if a given array entry can be added to two more entries to get the specified number?

Solution: 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;
}
```

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 to find a pair in a sorted array that sums to a specified value, which is $O(n^2)$ overall.

Variant: 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: Solve the same problem when k, the number of elements to sum, is an additional input.

Variant: Write a program 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 $A[p] \leq A[r] \leq A[s]$.

Variant: Write a program that takes as input an array of integers A and an integer T, and returns the number of 3-tuples (p, q, r) such that $A[p] + A[q] + A[r] \leq T$ and $A[p] \leq A[q] \leq A[r]$.

Chapter

14 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 set V of *vertices* and a set $E \subset V \times V$ 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, a start vertex, etc. A directed graph can be depicted pictorially as in Figure 14.1.

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 each (v_i, v_{i+1}) is an edge. The sequence may consist of 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 14.1.

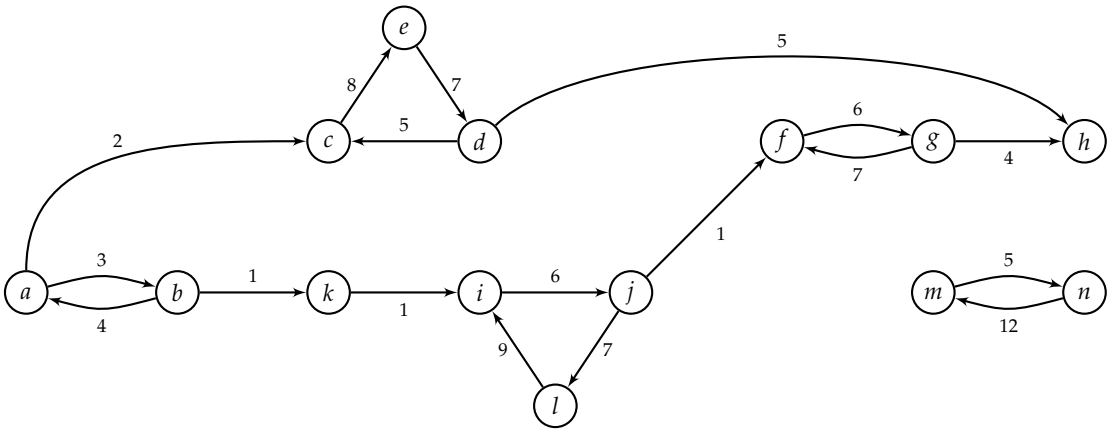


Figure 14.1: A directed graph with weights on edges.

A *directed acyclic graph* (*DAG*) is a directed graph in which there are no *cycles*, i.e., paths which contain one or more edges and which begin and end at the same vertex. See Figure 14.2 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.

An undirected graph is also a tuple (V, E); however, E is a set of unordered pairs of vertices. Graphically, this is captured by drawing arrowless connections between vertices, as in Figure 14.3.

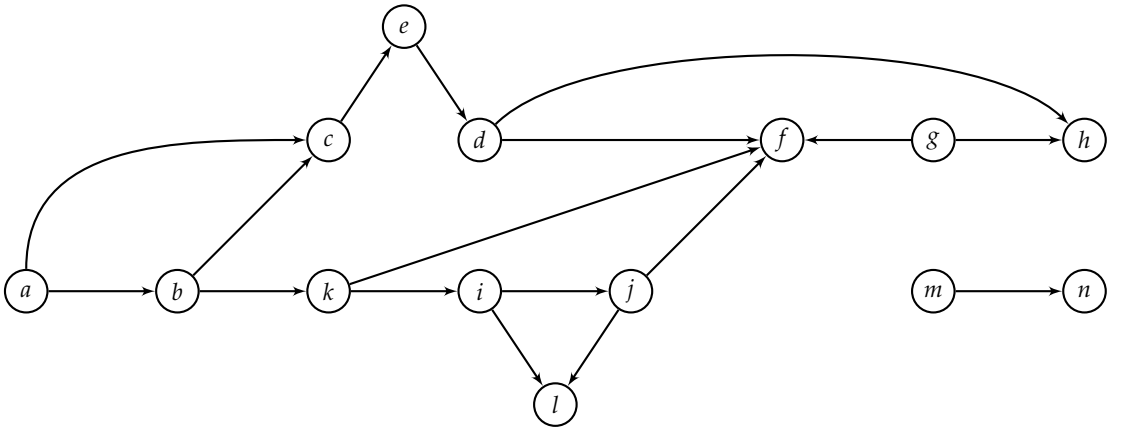


Figure 14.2: A directed acyclic graph. Vertices a, g, m are sources and vertices l, f, h, n are sinks. The ordering $\langle a, b, c, e, d, g, h, k, i, j, f, l, m, n \rangle$ is a topological ordering of the vertices.

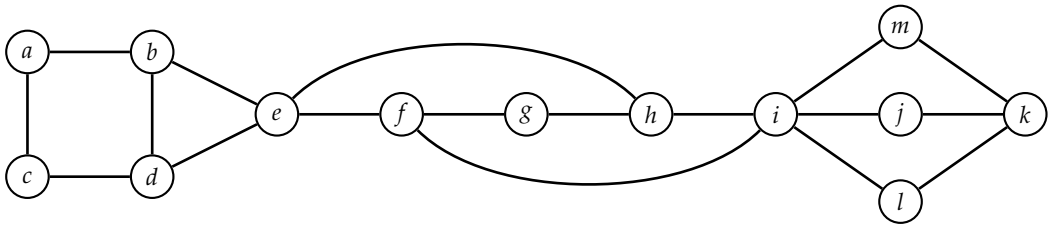


Figure 14.3: An undirected graph.

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.

For example, the graph in Figure 14.3 is connected, and it has a single connected component. If edge (h, i) is removed, it remains connected. If additionally (f, i) is removed, it becomes disconnected and there are two connected components.

A directed graph is called *weakly connected* if replacing all of its directed edges with undirected edges produces an undirected graph that is connected. 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 14.4 is a tree. Note that its edge set is a subset of the edge set of the undirected graph in Figure 14.3 on the previous 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.

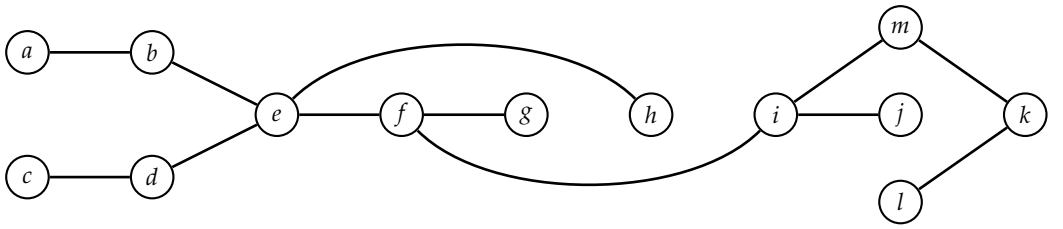


Figure 14.4: A tree.

Graphs boot camp

Graphs are ideal for modeling and analyzing relationships between pairs of objects. For example, suppose you were given a list of the outcomes of matches between pairs of teams, with each outcome being a win or loss. A natural question is as follows: given teams A and B, is there a sequence of teams starting with A and ending with B such that each team in the sequence has beaten the next team in the sequence?

A slick way of solving this problem is to model the problem using a graph. Teams are vertices, and an edge from one team to another indicates that the team corresponding to the source vertex has beaten the team corresponding to the destination vertex. Now we can apply graph reachability to perform the check. Both DFS and BFS are reasonable approaches—the program below uses DFS.

```
struct MatchResult {
  string winning_team, losing_team;
};

bool CanTeamABeatTeamB(const vector<MatchResult>& matches,
                       const string& team_a, const string& team_b) {
  return IsReachableDFS(BuildGraph(matches), team_a, team_b,
                        make_unique<unordered_set<string>>().get());
}

unordered_map<string, unordered_set<string>> BuildGraph(
    const vector<MatchResult>& matches) {
  unordered_map<string, unordered_set<string>> graph;
  for (const MatchResult& match : matches) {
    graph[match.winning_team].emplace(match.losing_team);
  }
  return graph;
}
```

```
bool IsReachableDFS(const unordered_map<string, unordered_set<string>>& graph,
                    const string& curr, const string& dest,
                    unordered_set<string>* visited_ptr) {
  unordered_set<string>& visited = *visited_ptr;
  if (curr == dest) {
    return true;
  } else if (visited.find(curr) != visited.end() ||
             graph.find(curr) == graph.end()) {
    return false;
  }
  visited.emplace(curr);
  for (const string& team : graph.at(curr)) {
    if (IsReachableDFS(graph, team, dest, visited_ptr)) {
      return true;
    }
  }
  return false;
}
```

The time complexity and space complexity are both $O(E)$, where E is the number of outcomes.

It's natural to use a graph when the problem involves **spatially connected** objects, e.g., road segments between cities.

More generally, consider using a graph when you need to analyze **any binary relationship**, between objects, such as interlinked webpages, followers in a social graph, etc.

Some graph problems entail **analyzing structure**, e.g., looking for cycles or connected components. **DFS** works particularly well for these applications.

Some graph problems are related to **optimization**, e.g., find the shortest path from one vertex to another. **BFS, Dijkstra's shortest path algorithm, and minimum spanning tree** are examples of graph algorithms appropriate for optimization problems.

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 the 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.

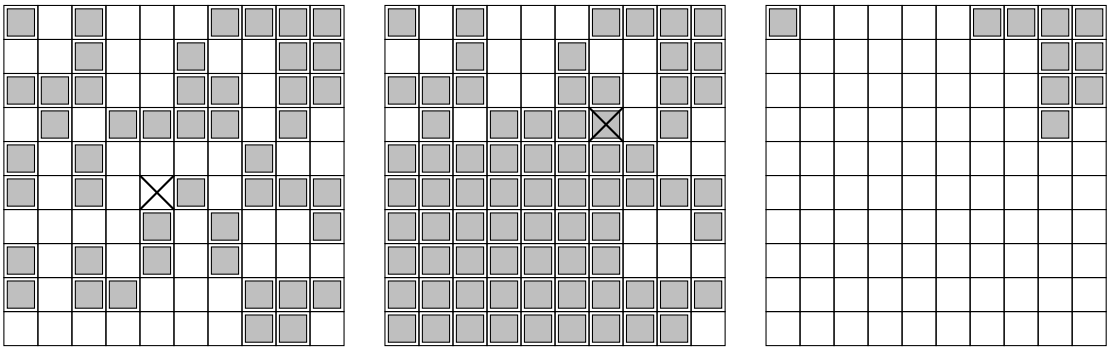
DFS and BFS differ from each other in terms of the additional information they provide, e.g., BFS can be used to compute distances from the start vertex and DFS can be used to check for the presence of cycles. Key notions in DFS include the concept of *discovery time* and *finishing time* for vertices.

14.1 Paint a Boolean matrix

Let A be a Boolean 2D array encoding a black-and-white image. The entry $A(a, b)$ can be viewed as encoding the color at entry (a, b). Call two entries adjacent if one is to the left, right, above or

below the other. Note that the definition implies that an entry can be adjacent to at most four other entries, and that adjacency is symmetric, i.e., if $e0$ is adjacent to entry $e1$, then $e1$ is adjacent to $e0$.

Define a path from entry $e0$ to entry $e1$ to be a sequence of adjacent entries, starting at $e0$, ending at $e1$, with successive entries being adjacent. 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 14.5: 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).

Implement a routine that takes an $n \times m$ Boolean array A together with an entry (x, y) and flips the color of the region associated with (x, y). See Figure 14.5 for an example of flipping.

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

Solution: Graph search can overcome the complexity of enumerative and random search solutions. Specifically, entries can be viewed as vertices, with vertices corresponding to adjacent entries begin 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 the queue. The computation ends when the queue 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 FlipColor(int x, int y, vector<deque<bool>>* A_ptr) {
  vector<deque<bool>>& A = *A_ptr;
  const array<array<int, 2>, 4> kDirs = {
      {{{0, 1}}, {{0, -1}}, {{1, 0}}, {{-1, 0}}}};
  const bool color = A[x][y];

  struct Coordinate {
    int x, y;
  };
  queue<Coordinate> q;
  A[x][y] = !color;  // Flips.
```

```
  q.emplace(Coordinate{x, y});
  while (!q.empty()) {
    Coordinate curr = q.front();
    for (const array<int, 2>& dir : kDirs) {
      const int next_x = curr.x + dir[0], next_y = curr.y + dir[1];
      if (next_x >= 0 && next_x < A.size() && next_y >= 0 &&
          next_y < A[next_x].size() && A[next_x][next_y] == color) {
        // Flips the color.
        A[next_x][next_y] = !color;
        q.emplace(Coordinate{next_x, next_y});
      }
    }
    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 FlipColor(int x, int y, vector<deque<bool>>* A_ptr) {
  vector<deque<bool>>& A = *A_ptr;
  const array<array<int, 2>, 4> kDirs = {
      {{{0, 1}}, {{0, -1}}, {{1, 0}}, {{-1, 0}}}};
  const bool color = A[x][y];

  A[x][y] = !color;  // Flips.
  for (const array<int, 2>& dir : kDirs) {
    const int next_x = x + dir[0], next_y = y + dir[1];
    if (next_x >= 0 && next_x < A.size() && next_y >= 0 &&
        next_y < A[next_x].size() && A[next_x][next_y] == color) {
      FlipColor(next_x, next_y, A_ptr);
    }
  }
}
```

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

Variant: 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.

Chapter

15 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 focused on the shared memory model.

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.

The problems in this chapter focus on thread-level parallelism. Problems concerned with parallelism on distributed memory architectures, e.g., cluster computing, are usually not meant to be coded; they are popular design and architecture problems.

Parallel computing boot camp

A semaphore is a very powerful synchronization construct. Conceptually, a semaphore maintains a set of permits. A thread calling *acquire()* on a semaphore waits, if necessary, until a permit is available, and then takes it. A thread calling *release()* on a semaphore adds a permit and notifies threads waiting on that semaphore, potentially releasing a blocking acquirer. The program below shows how to implement a semaphore in C++, using language primitives. (The C++ concurrency library provides a full-featured implementation of semaphores which should be used in practice.)

```
class Semaphore {
 public:
  Semaphore(int max_available) : max_available_(max_available), taken_(0) {}

  void Acquire() {
    unique_lock<mutex> lock(mx_);
    while (taken_ == max_available_) {
      cond_.wait(lock);
    }
    ++taken_;
  }

  void Release() {
    lock_guard<mutex> lock(mx_);
    --taken_;
    cond_.notify_all();
  }

 private:
  int max_available_;
  int taken_;
  mutex mx_;
  condition_variable cond_;
};
```

Start with an algorithm that **locks aggressively** and is easily seen to be correct. Then **add back concurrency**, while ensuring the **critical** parts are locked.

When analyzing parallel code, assume a worst-case thread scheduler. In particular, it may choose to schedule the same thread repeatedly, it may alternate between two threads, it may starve a thread, etc.

Try to work at a **higher level of abstraction**. In particular, know the **concurrency libraries**—don't implement your own **semaphores, thread pools, deferred execution**, etc. (You should know how these features are implemented, and implement them if asked to.)

15.1 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.

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 string-valued name field. The class 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).

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

Solution: 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.

Part II

Domain Specific Problems

Chapter

16

Design Problems

Don't be fooled by the many books on complexity or by the many complex and arcane algorithms you find in this book or elsewhere. Although there are no textbooks on simplicity, simple systems work and complex don't.

— "Transaction Processing: Concepts and Techniques," J. Gray, 1992

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 16.1. Some other things to keep in mind when designing a system are implementation time, extensibility, scalability, testability, security, internationalization, and IP issues.

Table 16.1: System design patterns.

Design principle	Key points
Algorithms and Data Structures	Identify the basic algorithms and data structures
Decomposition	Split the functionality, architecture, and code into manageable, reusable components.
Scalability	Break the problem into subproblems that can be solved relatively independently on different machines. Shard data across machines to make it fit. Decouple the servers that handle writes from those that handle reads. Use replication across the read servers to gain more performance. Consider caching 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, 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.

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.

Scalability

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.

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.

16.1 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.

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.

Hint: Normalize the video format and create signatures.

Solution: 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: Design an online music identification service.

Chapter

17 Language Questions

The limits of my language means the limits of my world.

— L. Wittgenstein

We now present some commonly asked problems on the C++ language. This set is not meant to be comprehensive—it's a test of preparedness. If you have a hard time with them, we recommend that you spend more time reviewing language.

17.1 Smart pointers

What is a smart pointer? What are the three commonly used smart pointer types? What is the appropriate use case for each one?

A smart pointer is a class which encapsulates an actual pointer to a dynamically allocated object, keeps track of its usage, and ensures that memory is deallocated when it is appropriate so that there is no memory/resources leak. The standard library includes these smart pointer classes:

- `unique_ptr`: destroys the object when the variable goes out of scope. May transfer ownership to another unique_ptr, e.g., when passing by value to a function. At all times there will be only one pointer referring to an object.
- `shared_ptr`: is freely copyable so that many variables can refer to the same object. Keeps track of the reference count and destroys the object when it is no longer in use.
- `weak_ptr`: refers to an object which is held by a `shared_ptr` but does not participate in reference counting so the object may be destroyed even if a `weak_ptr` refers to it. A `weak_ptr` is needed to solve the problem of reference cycles—if `shared_ptr` A points to B and `shared_ptr` B points to A, then neither will ever be destroyed.

Here is an example illustrating the difference between `shared_ptr` and `weak_ptr`.

```
struct Cycle1 {
  shared_ptr<Cycle2> next;
};

struct Cycle2 {
  shared_ptr<Cycle1> next;
};

auto head = make_shared<Cycle1>();  // head's reference count is now 1.
auto tail = make_shared<Cycle2>();  // tail's reference count is now 1.
head->next = tail;  // tail's reference count is now 2.
tail->next = head;  // head's reference count is now 2.
// On destruction of head and tail, the reference counts go to 1 and stay
// there.
```

When `head` and `tail` go out of scope the reference counts decrease by 1 so both ref counts will be 1. This is a leak, since `head` and `tail` can only be destroyed when their ref counts become 0. We can fix this leak by declaring `tail` to be of type `weak_ptr<Cycle2>`.

Chapter

18 Object-Oriented Design

One thing expert designers know not to do is solve every problem from first principles.

— "*Design Patterns: Elements of Reusable Object-Oriented Software*," E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides, 1994

A class is an encapsulation of data and methods that operate on that data. Classes match the way we think about computation. They provide encapsulation, which reduces the conceptual burden of writing code, and enable code reuse, through the use of inheritance and polymorphism. However, naive use of object-oriented constructs can result in code that is hard to maintain.

A design pattern is a general repeatable solution to a commonly occurring problem. It is not a complete design that can be coded up directly—rather, it is a description of how to solve a problem that arises in many different situations. In the context of object-oriented programming, design patterns address both reuse and maintainability. In essence, design patterns make some parts of a system vary independently from the other parts.

Adnan's Design Pattern course material, available freely online, contains lecture notes, homeworks, and labs that may serve as a good resource on the material in this chapter.

18.1 Creational Patterns

Explain what each of these creational patterns is: builder, static factory, factory method, and abstract factory.

Solution: The idea behind the builder pattern is to build a complex object in phases. It avoids mutability and inconsistent state by using an mutable inner class that has a build method that returns the desired object. Its key benefits are that it breaks down the construction process, and can give names to steps. Compared to a constructor, it deals far better with optional parameters and when the parameter list is very long.

A static factory is a function for construction of objects. Its key benefits are as follow: the function's name can make what it's doing much clearer compared to a call to a constructor. The function is not obliged to create a new object—in particular, it can return a flyweight. It can also return a subtype that's more optimized, e.g., it can choose to construct an object that uses an integer in place of a Boolean array if the array size is not more than the integer word size.

A factory method defines interface for creating an object, but lets subclasses decided which class to instantiate. The classic example is a maze game with two modes—one with regular rooms, and one with magic rooms. The program below uses a template method to combine the logic common to the two versions of the game.

```
class MazeGameCreator {
 public:
```

```
  virtual Room* MakeRoom() = 0;
  // This factory method is a template method for creating MazeGame objects.
  // MazeGameCreator's subclasses implement MakeRoom() as appropriate for
  // the type of room being created.
  MazeGame* FactoryMethod() {
    MazeGame* mazeGame = new MazeGame();
    Room* room1 = MakeRoom();
    Room* room2 = MakeRoom();
    room1->Connect(room2);
    mazeGame->AddRoom(room1);
    mazeGame->AddRoom(room2);
    return mazeGame;
  }
};
```

This snippet implements the regular rooms.

```
class OrdinaryMazeGameCreator : public MazeGameCreator {
  Room* MakeRoom() override { return new OrdinaryRoom(); }
};
```

This snippet implements the magic rooms.

```
class MagicMazeGameCreator : public MazeGameCreator {
  Room* MakeRoom() override { return new MagicRoom(); }
};
```

Here's how you use the factory to create regular and magic games.

```
MazeGame* ordinaryMazeGame =
    (new OrdinaryMazeGameCreator())->FactoryMethod();
MazeGame* magicMazeGame = (new MagicMazeGameCreator())->FactoryMethod();
```

A drawback of the factory method pattern is that it makes subclassing challenging.

An abstract factory provides an interface for creating families of related objects without specifying their concrete classes. For example, a class `DocumentCreator` could provide interfaces to create a number of products, such as `createLetter()` and `createResume()`. Concrete implementations of this class could choose to implement these products in different ways, e.g., with modern or classic fonts, right-flush or right-ragged layout, etc. Client code gets a `DocumentCreator` object and calls its factory methods. Use of this pattern makes it possible to interchange concrete implementations without changing the code that uses them, even at runtime. The price for this flexibility is more planning and upfront coding, as well as and code that may be harder to understand, because of the added indirections.

Chapter

19

Common Tools

UNIX is a general-purpose, multi-user, interactive operating system for the Digital Equipment Corporation PDP-11/40 and 11/45 computers. It offers a number of features seldom found even in larger operating systems, including: (1) a hierarchical file system incorporating demountable volumes; (2) compatible file, device, and inter-process I/O; (3) the ability to initiate asynchronous processes; (4) system command language selectable on a per-user basis; and (5) over 100 subsystems including a dozen languages.

— "*The UNIX TimeSharing System*,"
D. Ritchie and K. Thompson, 1974

The problems in this chapter are concerned with tools: version control systems, scripting languages, build systems, databases, and the networking stack. Such problems are not commonly asked—expect them only if you are interviewing for a specialized role, or if you claim specialized knowledge, e.g., network security or databases. We emphasize these are vast subjects, e.g., networking is taught as a sequence of courses in a university curriculum. Our goal here is to give you a flavor of what you might encounter. Adnan's Advanced Programming Tools course material, available freely online, contains lecture notes, homeworks, and labs that may serve as a good resource on the material in this chapter.

Version control

Version control systems are a cornerstone of software development—every developer should understand how to use them in an optimum way.

19.1 Merging in a version control system

What is merging in a version control system? Specifically, describe the limitations of line-based merging and ways in which they can be overcome.

Solution:

Modern version control systems allow developers to work concurrently on a personal copy of the entire codebase. This allows software developers to work independently. The price for this merging: periodically, the personal copies need to be integrated to create a new shared version. There is the possibility that parallel changes are conflicting, and these must be resolved during the merge.

By far, the most common approach to merging is text-based. Text-based merge tools view software as text. Line-based merging is a kind of text-based merging, specifically one in which each line is treated as an indivisible unit. The merging is "three-way": the lowest common ancestor in the revision history is used in conjunction with the two versions. With line-based merging of text files, common text lines can be detected in parallel modifications, as well as text lines that have been

inserted, deleted, modified, or moved. Figure 19.1 on the facing page shows an example of such a line-based merge.

One issue with line-based merging is that it cannot handle two parallel modifications to the same line: the two modifications cannot be combined, only one of the two modifications must be selected. A bigger issue is that a line-based merge may succeed, but the resulting program is broken because of syntactic or semantic conflicts. In the scenario in Figure 19.1 on the next page the changes made by the two developers are made to independent lines, so the line-based merge shows no conflicts. However, the program will not compile because a function is called incorrectly.

In spite of this, line-based merging is widely used because of its efficiency, scalability,and accuracy. A three-way, line-based merge tool in practice will merge 90% of the changed files without any issues. The challenge is to automate the remaining 10% of the situations that cannot be merged automatically.

Text-based merging fails because it does not consider any syntactic or semantic information.

Syntactic merging takes the syntax of the programming language into account. Text-based merge often yields unimportant conflicts such as a code comment that has been changed by different developers or conflicts caused by code reformatting. A syntactic merge can ignore all these: it displays a merge conflict when the merged result is not syntactically correct.

We illustrate syntactic merging with a small example:

```
if (n%2 == 0) then m = n/2;
```

Two developers change this code in a different ways, but with the same overall effect. The first updates it to

```
if (n%2 == 0) then m = n/2 else m = (n-1)/2;
```

and the second developer's update is

```
m = n/2;
```

Both changes to the same thing. However, a textual-merge will likely result in

```
m := n div 2 else m := (n-1)/2
```

which is syntactically incorrect. Syntactic merge identifies the conflict; it is the integrator's responsibility to manually resolve it, e.g., by removing the `else` portion.

Syntactic merge tools are unable to detect some frequently occurring conflicts. For example, in the merge of Versions *1a* and *1b* in Figure 19.1 on the facing page will not compile, since the call to `sum(10)` has the wrong signature. Syntactic merge will not detect this conflict since the program is still syntactically correct. The conflict is a semantic conflict, specifically, a static semantic conflict, since it is detectable at compile-time. (The compile will return something like "function argument mismatch error".)

Technically, syntactic merge algorithms operate on the parse trees of the programs to be merged. More powerful static semantic merge algorithms have been proposed that operate on a graph representation of the programs, wherein definitions and usage are linked, making it easier to identify mismatched usage.

Static semantic merging also has shortcomings. For example, suppose `Point` is a class representing 2*D*-points using Cartesian coordinates, and that this class has a distance function that return $\sqrt{x^2 + y^2}$. Suppose Alice checks out out the project, and subclasses `Point` to create a class that supports polar coordinates, and uses `Points`'s distance function to return the radius. Concurrently, Bob checks out the project and changes `Point`'s distance function to return $|x| + |y|$. Static semantic

merging reports no conflicts: the merged program compiles without any trouble. However the behavior is not what was expected.

Both syntactic merging and semantic merging greatly increase runtimes, and are very limiting since they are tightly coupled to specific programming languages. In practice, line-based merging is used in conjunction with a small set of unit tests (a "smoke suite") invoked with a pre-commit hook script. Compilation finds syntax and static semantics errors, and the tests (hopefully) identify deeper semantic issues.

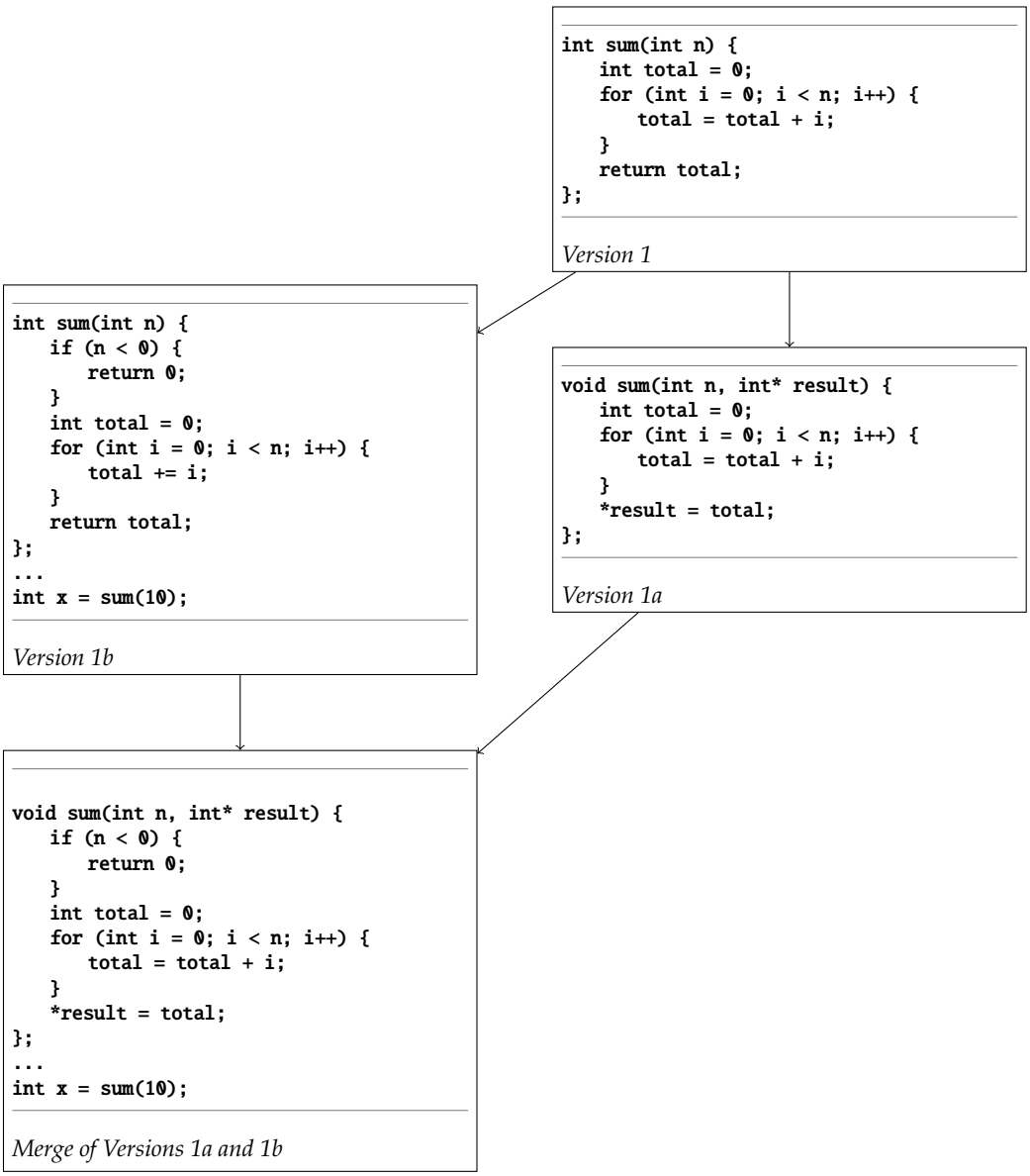


Figure 19.1: An example of a 3-way line based merge.

Build systems

A program typically cannot be executed immediately after change: intermediate steps are required to build the program and test it before deploying it. Other components can also be affected by

changes, such as documentation generated from program text. Build systems automate these steps.

Database

Most software systems today interact with databases, and it's important for developers to have basic knowledge of databases.

19.2 Normalization

What is database normalization? What are its advantages and disadvantages?
Solution: Database normalization is the process of organizing the columns and tables of a relational database to minimize data redundancy. Specifically, normalization involves decomposing a table into less redundant tables without losing information, thereby enforcing integrity and saving space.

The central idea is the use of "foreign keys". A foreign key is a field (or collection of fields) in one table that uniquely identifies a row of another table. In simpler words, the foreign key is defined in a second table, but it refers to the unique key in the first table. For example, a table called Employee may use a unique key called employee_id. Another table called Employee Details has a foreign key which references employee_id in order to uniquely identify the relationship between both the tables. The objective is to isolate data so that additions, deletions, and modifications of an attribute can be made in just one table and then propagated through the rest of the database using the defined foreign keys.

The primary drawback of normalization is performance—often, a large number of joins are required to recover the records the application needs to function.

Networking

Most software systems today interact with the network, and it's important for developers even higher up the stack to have basic knowledge of networking. Here are networking questions you may encounter in an interview setting.

Part III

The Honors Class

Chapter

20 Honors Class

The supply of problems in mathematics is inexhaustible, and as soon as one problem is solved numerous others come forth in its place.

— "*Mathematical Problems*,"
D. Hilbert, 1900

This chapter contains problems that are more difficult to solve than the ones presented earlier. Many of them are commonly asked at interviews, albeit with the expectation that the candidate will not deliver the best solution.

There are several reasons why we included these problems:

- Although mastering these problems is not essential to your success, if you do solve them, you will have put yourself into a very strong position, similar to training for a race with ankle weights.
- If you are asked one of these questions or a variant thereof, and you successfully derive the best solution, you will have strongly impressed your interviewer. The implications of this go beyond being made an offer—your salary, position, and status will all go up.
- Some of the problems are appropriate for candidates interviewing for specialized positions, e.g., optimizing large scale distributed systems, machine learning for computational finance, etc. They are also commonly asked of candidates with advanced degrees.
- Finally, if you love a good challenge, these problems will provide it!

You should be happy if your interviewer asks you a hard question—it implies high expectations, and is an opportunity to shine, compared, for example, to being asked to write a program that tests if a string is palindromic.

20.1 Compute the greatest common divisor

The greatest common divisor (GCD) of positive integers x and y is the largest integer d such that d divides x evenly, and d divides y evenly, i.e., $x \bmod d = 0$ and $y \bmod d = 0$.

Design an efficient algorithm for computing the GCD of two numbers without using multiplication, division or the modulus operators.

Hint: Use case analysis: both even; both odd; one even and one odd.

Solution: The straightforward algorithm is based on recursion. If $x = y$, $\text{GCD}(x, y) = x$; otherwise, assume without loss of generality, that $x > y$. Then $\text{GCD}(x, y)$ is the $\text{GCD}(x - y, y)$.

The recursive algorithm based on the above does not use multiplication, division or modulus, but for some inputs it is very slow. As an example, if the input is $x = 2^n$, $y = 2$, the algorithm

makes 2^{n-1} recursive calls. The time complexity can be improved by observing that the repeated subtraction amounts to division, i.e., when $x > y$, $\text{GCD}(x, y) = \text{GCD}(y, x \bmod y)$, but this approach uses integer division which was explicitly disallowed in the problem statement.

Here is a fast solution, which is also based on recursion, but does not use general multiplication or division. Instead it special-cases division to division by 2.

An example is illustrative. Suppose we were to compute the GCD of 24 and 300. Instead of repeatedly subtracting 24 from 300, we can observe that since both are even, the result is $2 \times \text{GCD}(12, 150)$. Dividing by 2 is a right shift by 1, so we do not need a general division operation. Since 12 and 150 are both even, $\text{GCD}(12, 150) = 2 \times \text{GCD}(6, 75)$. Since 75 is odd, the GCD of 6 and 75 is the same as the GCD of 3 and 75, since 2 cannot divide 75. The GCD of 3 and 75 is the GCD of 3 and $75 - 3 = 72$. Repeatedly applying the same logic, $\text{GCD}(3, 72) = \text{GCD}(3, 36) = \text{GCD}(3, 18) = \text{GCD}(3, 9) = \text{GCD}(3, 6) = \text{GCD}(3, 3) = 3$. This implies $\text{GCD}(24, 300) = 2 \times 2 \times 3 = 12$.

More generally, the base case is when the two arguments are equal. Otherwise, we check if none, one, or both numbers are even. If both are even, we compute the GCD of the halves of the original numbers, and return that result times 2; if exactly one is even, we half it, and return the GCD of the resulting pair; if both are odd, we subtract the smaller from the larger and return the GCD of the resulting pair. Multiplication by 2 is trivially implemented with a single left shift. Division by 2 is done with a single right shift.

```
long long GCD(long long x, long long y) {
  if (x == y) {
    return x;
  } else if (!(x & 1) && !(y & 1)) {  // x and y are even.
    return GCD(x >> 1, y >> 1) << 1;
  } else if (!(x & 1) && y & 1) {  // x is even, and y is odd.
    return GCD(x >> 1, y);
  } else if (x & 1 && !(y & 1)) {  // x is odd, and y is even.
    return GCD(x, y >> 1);
  } else if (x > y) {  // Both x and y are odd, and x > y.
    return GCD(x - y, y);
  }
  return GCD(x, y - x);  // Both x and y are odd, and x <= y.
}
```

Note that the last step leads to a recursive call with one even and one odd number. Consequently, in every two calls, we reduce the combined bit length of the two numbers by at least one, meaning that the time complexity is proportional to the sum of the number of bits in x and y, i.e., $O(\log x + \log y)$.

20.2 Compute the maximum product of all entries but one

Suppose you are given an array A of integers, and are asked to find the largest product that can be made by multiplying all but one of the entries in A. (You cannot use an entry more than once.) For example, if $A = \langle 3, 2, 5, 4 \rangle$, the result is $3 \times 5 \times 4 = 60$, if $A = \langle 3, 2, -1, 4 \rangle$, the result is $3 \times 2 \times 4 = 24$, and if $A = \langle 3, 2, -1, 4, -1, 6 \rangle$, the result is $3 \times -1 \times 4 \times -1 \times 6 = 72$.

One approach is to form the product P of all the elements, and then find the maximum of $P/A[i]$ over all i. This takes $n - 1$ multiplications (to form P) and n divisions (to compute each $P/A[i]$). Suppose because of finite precision considerations we cannot use a division-based approach; we can only use multiplications. The brute-force solution entails computing all n products of $n - 1$ elements; each such product takes $n - 2$ multiplications, i.e., $O(n^2)$ time complexity.

Given an array A of length n whose entries are integers, compute the largest product that can be made using $n-1$ entries in A. You cannot use an entry more than once. Array entries may be positive, negative, or 0. Your algorithm cannot use the division operator, explicitly or implicitly.

Hint: Consider the products of the first $i-1$ and the last $n-i$ elements. Alternatively, count the number of negative entries and zero entries.

Solution: The brute-force approach to compute $P/A[i]$ is to multiplying the entries appearing before i with those that appear after i. This leads to $n(n-2)$ multiplications, since for each term $P/A[i]$ we need $n-2$ multiplications.

Note that there is substantial overlap in computation when computing $P/A[i]$ and $P/A[i+1]$. In particular, the product of the entries appearing before $i+1$ is $A[i]$ times the product of the entries appearing before i. We can compute all products of entries before i with $n-1$ multiplications, and all products of entries after i with $n-1$ multiplications, and store these values in arrays L and R, respectively. The desired result then is the maximum over all i of $L[i] \times R[i]$.

```
int FindBiggestNMinusOneProduct(const vector<int>& A) {
  // Builds forward product L, and backward product R.
  vector<int> L, R(A.size());
  partial_sum(A.cbegin(), A.cend(), back_inserter(L), multiplies<int>());
  partial_sum(A.crbegin(), A.crend(), R.rbegin(), multiplies<int>());

  // Finds the biggest product of (n - 1) numbers.
  int max_product = numeric_limits<int>::min();
  for (int i = 0; i < A.size(); ++i) {
    int forward = i > 0 ? L[i - 1] : 1;
    int backward = i + 1 < A.size() ? R[i + 1] : 1;
    max_product = max(max_product, forward * backward);
  }
  return max_product;
}
```

The time complexity is $O(n)$; the space complexity is $O(n)$, since the solution uses two arrays, each of length n.

We now solve this problem in $O(n)$ time and $O(1)$ additional storage. The insight comes from the fact that if there are no negative entries, the maximum product comes from using all but the smallest element. (Note that this result is correct if the number of 0 entries is zero, one, or more.)

If the number of negative entries is odd, regardless of how many 0 entries and positive entries, the maximum product uses all entries except for the negative entry with the smallest absolute value, i.e., the greatest negative entry.

Going further, if the number of negative entries is even, the maximum product again uses all but the smallest nonnegative element, assuming the number of nonnegative entries is greater than zero. (This is correct, even in the presence of 0 entries.)

If the number of negative entries is even, and there are no nonnegative entries, the result must be negative. Since we want the largest product, we leave out the entry whose magnitude is largest, i.e., the least negative entry.

This analysis yields a two-stage algorithm. First, determine the applicable scenario, e.g., are there an even number of negative entries? Consequently, perform the actual multiplication to get the result.

```
int FindBiggestNMinusOneProduct(const vector<int>& A) {
  int least_nonnegative_idx = -1;
```

```
  int number_of_negatives = 0, greatest_negative_idx = -1,
      least_negative_idx = -1;

  // Identify the least negative, greatest negative, and least nonnegative
  // entries.
  for (int i = 0; i < A.size(); ++i) {
    if (A[i] < 0) {
      ++number_of_negatives;
      if (least_negative_idx == -1 || A[least_negative_idx] < A[i]) {
        least_negative_idx = i;
      }
      if (greatest_negative_idx == -1 || A[i] < A[greatest_negative_idx]) {
        greatest_negative_idx = i;
      }
    } else {  // A[i] >= 0.
      if (least_nonnegative_idx == -1 || A[i] < A[least_nonnegative_idx]) {
        least_nonnegative_idx = i;
      }
    }
  }

  int product = 1;
  int idx_to_skip =
      number_of_negatives % 2
          ? least_negative_idx
          // Check if there are any nonnegative entry.
          : (least_nonnegative_idx != -1 ? least_nonnegative_idx
                                         : greatest_negative_idx);
  for (int i = 0; i < A.size(); ++i) {
    if (i != idx_to_skip) {
      product *= A[i];
    }
  }
  return product;
}
```

The algorithm performs a traversal of the array, with a constant amount of computation per entry, a nested conditional, followed by another traversal of the array, with a constant amount of computation per entry. Hence, the time complexity is $O(n) + O(1) + O(n) = O(n)$. The additional space complexity is $O(1)$, corresponding to the local variables.

Variant: Let A be as above. Compute an array B where $B[i]$ is the product of all elements in A except $A[i]$. You cannot use division. Your time complexity should be $O(n)$, and you can only use $O(1)$ additional space.

Variant: Let A be as above. Compute the maximum over the product of all triples of distinct elements of A.

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:

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

Symbolism	Meaning
$(d_{k-1} \dots d_0)_r$	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
$\|S\|$	cardinality of set S
$S \setminus T$	set difference, i.e., $S \cap T'$, sometimes written as $S - T$
$\|x\|$	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
$\langle a_0, a_1, \dots, a_{n-1} \rangle$	sequence of n elements
$\sum_{R(k)} f(k)$	sum 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 $\sum_{a \le k \le b} f(k)$
$\{a \mid R(a)\}$	set of all a such that the relation $R(a)$ = true
$[l, r]$	closed interval: $\{x \mid l \le x \le r\}$
$[l, r)$	left-closed, right-open interval: $\{x \mid l \le x < r\}$
$\{a, b, \dots\}$	well-defined collection of elements, i.e., a set
A_i or $A[i]$	the ith 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 ith row and jth 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_1th to i_2th rows and from j_1th to j_2th 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
$x \bmod y$	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.
$x \ll y$	much less than
$x \gg y$	much greater than
$\Rightarrow$	logical implication

Index of Terms