```
  if (ans.empty()) {
    ans.push_back('0');
  }
  if (neg) {
    ans.push_back('-');
  }
  reverse(ans.begin(), ans.end());
  return ans;
}
```

Problem 5.8, *pg.49*: *Write a function that converts Excel column ids to the corresponding integer, with "A" corresponding to 1. The function signature is* `int ssDecodeColID(string)`*; you may ignore error conditions, such as* `col` *containing characters outside of* $[A, Z]$*. How would you test your code?*

Solution 5.8: This problem is similar to the problem of converting a string representing a base-26 number to the corresponding integer, except that "A" corresponds to 1 not 0.

```
int ssDecodeColID(const string &col) {
  int ret = 0;
  for (const char &c : col) {
    ret = ret * 26 + c - 'A' + 1;
  }
  return ret;
}
```

Good test cases are around boundaries, e.g., "A", "B", "Y", "Z", "AA", "AB", "ZY", "ZZ", and some random strings, e.g., "M", "BZ", "CCC".

Problem 5.9, *pg.50*: *Let A be an array of n integers. Write an encode function that returns a string representing the concatenation of the Elias gamma codes for $\langle A[0], A[1], \ldots, A[n-1] \rangle$ in that order, and a decode function that takes a string s assumed to be generated by the encode function, and returns the array that was passed to the encode function.*

Solution 5.9: The code follows in a straightforward way from the above specifications. Each encoded number starts with one fewer 0s than bits in the number, which allows us to uniquely determine the length of the result.

```
string trans_int_to_binary(int decimal) {
  string ret;
  while (decimal) {
    ret.insert(0, 1, '0' + (decimal & 1));
    decimal >>= 1;
  }
  return ret;
}

string encode(const vector<int> &A) {
  string ret = "";
  for (const int &a : A) {
```

```
    string binary = trans_int_to_binary(a);
    binary.insert(0, binary.size() - 1, '0');  // prepend 0s
    ret += binary;
  }
  return ret;
}

int trans_binary_to_int(const string &binary) {
  int ret = 0;
  for (const char &c : binary) {
    ret = (ret << 1) + c - '0';
  }
  return ret;
}

vector<int> decode(const string &s) {
  vector<int> ret;
  int idx = 0;
  while (idx < s.size()) {
    // Count the number of consecutive 0s
    int zero_idx = idx;
    while (zero_idx < s.size() && s[zero_idx] == '0') {
      ++zero_idx;
    }

    int len = zero_idx - idx + 1;
    ret.emplace_back(trans_binary_to_int(s.substr(zero_idx, len)));
    idx = zero_idx + len;
  }
  return ret;
}
```

Problem 5.10, ***pg. 50***: *Design an algorithm for computing the GCD of two numbers without using multiplication, division or the modulus operators.*

Solution 5.10: The idea is to use recursion, the base case being where one of the arguments is 0. Otherwise, we check if none, one or both numbers are even. If both are even, we compute the GCD of these numbers divided by 2, and return that result times 2; if 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.

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 run time complexity is proportional to the sum of the lengths of the arguments.

Languages such as Java and Python include libraries for manipulating integers of arbitrary length, making them ideally suited for our application.

```
private static BigInteger TWO = new BigInteger("2");

```

```
private static boolean isOdd(BigInteger x) {
  return x.testBit(0);
}

private static boolean isEven(BigInteger x) {
  return !isOdd(x);
}

public static BigInteger GCD(BigInteger x, BigInteger y) {
  if (x.equals(BigInteger.ZERO)) {
    return y;
  } else if (y.equals(BigInteger.ZERO)) {
    return x;
  } else if (isEven(x) && isEven(y)) {
    x = x.shiftRight(1);
    y = y.shiftRight(1);
    return TWO.multiply(GCD(x, y));
  } else if (isOdd(x) && isEven(y)) {
    return GCD(x, y.shiftRight(1));
  } else if (isOdd(y) && isEven(x)) {
    return GCD(y, x.shiftRight(1));
  } else if (x.compareTo(y) <= 0) {
    return GCD(x, y.subtract(x));
  } else {
    return GCD(y, x.subtract(y));
  }
}
```

Problem 5.11, *pg.50*: *Write a function that takes a single positive integer argument n ($n \geq 2$) and return all the primes between 1 and n.*

Solution 5.11: We use a bit-vector `is_prime` of length $n+1$ to encode the set of primes. Initialize each entry to 1. The entry `is_prime[i]` will eventually be set to 0 iff i is not a prime. Set p to 2. Count in increments of p and "mark" (set the corresponding entry in `is_prime` to 0) each number greater than p in the count to be a non-prime (since it is divisible by p). Update p to the next unmarked number, and iterate.

This approach can be improved somewhat by ignoring even numbers, and not allocating entries for i less than 3. The count can also start from p^2 instead of p, since all numbers kp, where $k < p$ have already been marked. The code below reflects these optimizations.

```
// Given n, return the primes from 1 to n
vector<int> generate_primes_from_1_to_n(const int &n) {
  int size = floor(0.5 * (n - 3)) + 1;
  // is_prime[i] represents (2i + 3) is prime or not
  vector<int> primes;  // stores the primes from 1 to n
  primes.emplace_back(2);
  vector<bool> is_prime(size, true);
  for (long i = 0; i < size; ++i) {
    if (is_prime[i]) {
      int p = (i << 1) + 3;
      primes.emplace_back(p);
```

```
      // Sieving from p^2, whose index is 2i^2 + 6i + 3
      for (long j = ((i * i) << 1) + 6 * i + 3; j < size; j += p) {
        is_prime[j] = false;
      }
    }
  }
  return primes;
}
```

Problem 5.12, ***pg. 50***: *Let R and S be xy-aligned rectangles in the Cartesian plane. Write a function which tests if R and S have a nonempty intersection. If the intersection is nonempty, return the rectangle formed by their intersection.*

Solution 5.12: Let the given rectangles be $R = ((R_x, R_y), R_w, R_h)$ and $S = ((S_x, S_y), S_w, S_h)$. Observe that the rectangles definitely do not intersect if $I_x = [R_x, R_x + R_w] \cap [S_x, S_x + S_w] = \emptyset$; similarly, they definitely do not intersect if $I_y = [R_y, R_y + R_h] \cap [S_y, S_y + S_h] = \emptyset$.

Conversely, any point $p = (x, y)$ such that $x \in I_x$ and $y \in I_y$ lies in both R and S. Suppose $I_x = [a_x, b_x]$ and $I_y = [a_y, b_y]$; then the desired rectangle is $((a_x, a_y), b_x - a_x, b_y - a_y)$.

```
class Rectangle {
  public:
    int x, y, width, height;
};

bool is_intersect(const Rectangle &R, const Rectangle &S) {
  return R.x <= S.x + S.width && R.x + R.width >= S.x &&
         R.y <= S.y + S.height && R.y + R.height >= S.y;
}

Rectangle intersect_rectangle(const Rectangle &R, const Rectangle &S) {
  if (is_intersect(R, S)) {
    return {max(R.x, S.x), max(R.y, S.y),
            min(R.x + R.width, S.x + S.width) - max(R.x, S.x),
            min(R.y + R.height, S.y + S.height) - max(R.y, S.y)};
  } else {
    return {0, 0, -1, -1};  // no intersection
  }
}
```

Variant 5.12.1: Given four points in the plane, how would you check if they are the vertices of an xy-aligned rectangle?

Variant 5.12.2: How would you check if two rectangles, not necessarily xy-aligned, intersect?

Problem 5.13, ***pg. 51***: *Write a function that multiplies two unsigned positive integers. The only operators you are allowed to use are assignment and the bitwise operators, i.e., », «, |, &,*

~, ^. (In particular, you cannot use increment or decrement.) You may use loops, conditionals and functions that you write yourself; other functions are allowed.

Solution 5.13: We mimic the grade school algorithm for multiplication. Suppose we are to multiply x and y. We initialize sum to 0 and iterate through the bits of x, adding $2^k y$ to sum if bit k of x is 1.

We implement addition itself by mimicking the grade school algorithm for addition. This consists computing the sum bit-by-bit, and "rippling" the carry along. We use a bitmask that identifies the k-th bits; it also serves to tell us when all bits have been read.

```
unsigned add_no_operator(const unsigned &a, const unsigned &b) {
  unsigned sum = 0, carryin = 0, k = 1;
  while (k) {
    unsigned ak = a & k, bk = b & k;
    unsigned carryout =  (ak & bk) | (ak & carryin) | (bk & carryin);
    sum |= (ak ^ bk ^ carryin);
    carryin = carryout << 1;
    k <<= 1;
  }
  return sum;
}

unsigned multiply_no_operator(const unsigned &x, const unsigned &y) {
  unsigned sum = 0, k = 1, scaled_y = y;
  while (k) {
    // Examine the k-th bit of x
    if (x & k) {
      sum = add_no_operator(sum, scaled_y);
    }
    k <<= 1;
    scaled_y <<= 1;
  }
  return sum;
}
```

Problem 5.14, *pg. 51*: *Given two positive integers x and y, how would you compute x/y if the only operators you can use are addition, subtraction, and multiplication?*

Solution 5.14: We can use the following recursion:

$$\frac{x}{y} = \begin{cases} 0, & \text{if } x < y; \\ 1 + \frac{(x-y)}{y}, & \text{otherwise.} \end{cases}$$

This is not efficient by itself, but we can improve it by computing the largest k such that $2^k y \leq x$, in which case the recursive step is $2^k + \frac{(x-2^k y)}{y}$.

```
unsigned divide_x_y(const unsigned &x, const unsigned &y) {
  if (x < y) {
    return 0;
  }
```

```
5
6    int power = 0;
7    while ((1U << power) * y <= x) {
8      ++power;
9    }
10   unsigned part = 1U << (power - 1);
11   return part + divide_x_y(x - part * y, y);
12 }
```

Problem 6.1, *pg. 52*: *Write a function that takes an array A and an index i into A, and rearranges the elements such that all elements less than $A[i]$ appear first, followed by elements equal to $A[i]$, followed by elements greater than $A[i]$. Your algorithm should have $O(1)$ space complexity and $O(|A|)$ time complexity.*

Solution 6.1: This problem is conceptually straightforward: maintain four groups, *bottom* (elements less than `pivot`), *middle* (elements equal to `pivot`), *unclassified*, and *top* (elements greater than `pivot`). These groups are stored in contiguous order in A. To make this partitioning run in $O(1)$ space, we use `smaller`, `equal`, and `larger` pointers to track these groups in the following way:

- *bottom*: stored in subarray $A[0 : \texttt{smaller} - 1]$.
- *middle*: stored in subarray $A[\texttt{smaller} : \texttt{equal} - 1]$.
- *unclassified*: stored in subarray $A[\texttt{equal} : \texttt{larger}]$.
- *top*: stored in subarray $A[\texttt{larger} + 1 : |A| - 1]$.

We explore elements of *unclassified* in order, and classify the element into one of *bottom*, *middle*, and *top* groups according to the relative order between the incoming unclassified element and `pivot`. Each iteration decreases the size of *unclassified* group by 1, and the time spent within each iteration is constant, implying the time complexity is $\Theta(|A|)$.

The implementation is short but tricky, pay attention to the movements of pointers.

```
1  template <typename T>
2  void dutch_flag_partition(vector<T> &A, const int &pivot_index) {
3    T pivot = A[pivot_index];
4    /**
5     * Keep the following invariants during partitioning:
6     * bottom group: A[0 : smaller - 1]
7     * middle group: A[smaller : equal - 1]
8     * unclassified group: A[equal : larger]
9     * top group: A[larger + 1 : A.size() - 1]
10    */
11   int smaller = 0, equal = 0, larger = A.size() - 1;
12   // When there is any unclassified element
13   while (equal <= larger) {
14     // A[equal] is the incoming unclassified element
15     if (A[equal] < pivot) {
16       swap(A[smaller++], A[equal++]);
17     } else if (A[equal] == pivot) {
18       ++equal;
19     } else {  // A[equal] > pivot
20       swap(A[equal], A[larger--]);
```

```
    }
  }
}
```

ϵ-Variant 6.1.1: Assuming that keys take one of three values, reorder the array so that all objects of the same key appear in the same subarray. The order of the subarrays is not important. For example, both Figures 6.1(b) and 6.1(c) on Page 53 are valid answers for Figure 6.1(a) on Page 53. Use $O(1)$ additional space and $O(|A|)$ time.

ϵ-Variant 6.1.2: Given an array A of objects with keys that takes one of four values, reorder the array so that all objects that have the same key appear in the same subarray. Use $O(1)$ additional space and $O(|A|)$ time.

ϵ-Variant 6.1.3: Given an array A of objects with Boolean-valued keys, reorder the array so that all objects that have the same key appear in the same subarray. Use $O(1)$ additional space and $O(|A|)$ time.

Problem 6.2, *pg. 53*: *Design a deterministic scheme by which reads and writes to an uninitialized array can be made in $O(1)$ time. You may use $O(n)$ additional storage; reads to uninitialized entry should return* `false`.

Solution 6.2: Create an (uninitialized) array P of n pointers. The array P will maintain a pointer for each initialized entry of A to a back pointer on another array S, itself an (uninitialized) array of n integers. An integer-valued variable t indicates the first empty entry in S; initially, $t = 0$.

Each time entry i from A is to be read, we can check if that entry has been written to before by examining $P[i]$. If $P[i]$ is outside $[0, t-1]$, $A[i]$ is uninitialized. However, even if $P[i]$ is uninitialized, it may lie in $[0, t-1]$. We look at the "back pointer" stored in $S[P[i]]$ and confirm that it is indeed i.

The first time entry i is written in A, we set $S[t]$ to i, $P[i]$ to t and increment t. (We can check that the write is the first write to $A[i]$ by first performing a read and checking if the entry is uninitialized.)

The approach is illustrated in Figure 21.3 on the facing page. The first three entries written are at indices 7, 2, and 1, in that order. Checking if the entry at index 6 is initialized entails examining $P[6]$. If the value in $P[6]$ is not in $[0, 2]$, $A[6]$ is uninitialized. If it is a valid index, e.g., 1, we check if $S[1]$ is 6. For this example, $P[6] = 1$ and $S[P[6]] = 2 \neq 6$, so $A[6]$ is uninitialized.

```
template <typename ValueType, size_t N>
class Array {
  private:
    ValueType A[N];
    int P[N], S[N], t;

    const bool isValid(const size_t &i) const {
      return (0 <= P[i] && P[i] < t && S[P[i]] == i);
```

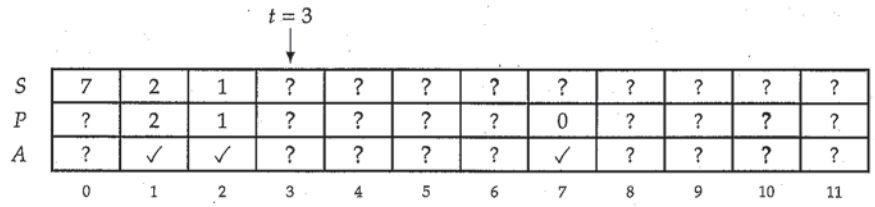


Figure 21.3: Initializing an array in $O(1)$ time.

```
  }

 public:
  Array(void) : t(0) {};

  const bool read(const size_t &i, ValueType &v) const {
    if (isValid(i)) {
      v = A[i];
      return true;
    }
    return false;
  }

  void write(const size_t &i, const ValueType &v) {
    if (!isValid(i)) {
      S[t] = i;
      P[i] = t++;
    }
    A[i] = v;
  }
};
```

Problem 6.3, *pg. 53*: *Design an algorithm that takes a sequence of n three-dimensional coordinates to be traversed, and returns the minimum battery capacity needed to complete the journey. The robot begins with a fully charged battery.*

Solution 6.3: Suppose the three-dimensions correspond to x, y, and z, with z being the vertical dimension. Since energy usage depends on the change in height of the robot, we can ignore the x and y coordinates. Suppose the points where the robot goes in successive order have z coordinates $z_0, \ldots, z_{n-1}$. Assume that the battery capacity is such that with the fully charged battery, the robot can climb B meters. The robot will run out of energy iff there exist integers i and j such that $i < j$ and $z_j - z_i > B$, i.e., to go from Point i to Point j, the robot has to climb more than B meters. Therefore, we would like to pick B such that for any $i < j$, we have $B \geq z_j - z_i$.

We developed several algorithms for this problem in the introduction. Specifically, on Page 2 we showed how to compute the minimum B in $O(n)$ time by keeping the running min as we do a sweep. In code:

```
template <typename HeightType>
```

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

Problem 6.4, pg. 54: *For each of the following, A is an integer array of length n.*

(1.) Compute the maximum value of $(A[j_0] - A[i_0]) + (A[j_1] - A[i_1])$, subject to $i_0 < j_0 < i_1 < j_1$.

(2.) Compute the maximum value of $\sum_{t=0}^{k-1}(A[j_t] - A[i_t])$, subject to $i_0 < j_0 < i_1 < j_1 < \cdots < i_{k-1} < j_{k-1}$. Here k is a fixed input parameter.

(3.) Repeat Problem (2.) when k can be chosen to be any value from 0 to $\lfloor n/2 \rfloor$.

Solution 6.4: The brute-force algorithm for (1.) has complexity $O(n^4)$. The complexity can be improved to $O(n^2)$ by applying the $O(n)$ algorithm to $A[0 : j]$ and $A[j+1 : n-1]$ for each $j \in [1, n-2]$. However, we can actually solve (1.) in $O(n)$ time by performing a forward iteration and storing the best solution for $A[0 : j]$, $j \in [1, n-1]$. We then do a reverse iteration, computing the best solution for $A[j : n-1]$, $j \in [0, n-2]$, which we combine with the result from the forward iteration. The additional space complexity is $O(n)$, which is the space used to store the best solutions for the subarrays.

Here is a straightforward algorithm for (2.). Iterate over j from 1 to k and iterate through A, recording for each index i the best solution for $A[0 : i]$ with j pairs. We store these solutions in an auxiliary array of length n. The overall time complexity will be $O(kn^2)$; by reusing the arrays, we can reduce the additional space complexity to $O(n)$.

We can improve the time complexity to $O(kn)$, and the additional space complexity to $O(k)$ as follows. Define B_i^j to be the most money you can have if you must make $j-1$ buy-sell transactions prior to i and buy at i. Define S_i^j to be the maximum profit achievable with j buys and sells with the j-th sell taking place at i. Then the following mutual recurrence holds:

$$\begin{aligned} S_i^j &= A[i] + \max_{i' < i} B_{i'}^j \\ B_i^j &= \max_{i' < i} S_{i'}^{j-1} - A[i] \end{aligned}$$

The key to achieving an $O(kn)$ time bound is the observation that computing B and S requires computing $\max_{i' < i} B_{i'}^{j-1}$ and $\max_{i' < i} S_{i'}^{j-1}$. These two quantities can be computed in constant time for each i and j with a conditional update. In code:

```
template <typename T>
T max_k_pairs_profits(const vector<T>& A, const int &k) {
  vector<T> k_sum(k << 1, numeric_limits<T>::min());
  for (int i = 0; i < A.size(); ++i) {
    vector<T> pre_k_sum(k_sum);
    for (int j = 0, sign = -1; j < k_sum.size() && j <= i; ++j, sign *= -1) {
```

```
      T diff = sign * A[i] + (j == 0 ? 0 : pre_k_sum[j - 1]);
      k_sum[j] = max(diff, pre_k_sum[j]);
    }
  }
  return k_sum.back();  // return the last selling profits as the answer
}
```

Note that the improved solution to (2.) on the preceding page specialized to $k = 2$ strictly subsumes the solution to (1.) on the facing page.

Surprisingly, (3.) on the preceding page can be solved trivially—since we can use an unlimited number of pairs, we can select all pairs $(i, i+1)$ such that $A[i+1] > A[i]$.

Problem 6.5, *pg. 54*: *Design an efficient algorithm for the* 0 mod *n-sum subset problem.*

Solution 6.5: Consider `prefix_sum[j]` $= \sum_{i=0}^{j} A[i] \bmod n$. Either each `prefix_sum[j]` is distinct, in which case for some c we have `prefix_sum[c]` $= 0$ (since `prefix_sum` takes values in $\{0, 1, \ldots, n-1\}$), or for some $a < b$ we have `prefix_sum[a]` $=$ `prefix_sum[b]`.

In the first case, the subarray $A[0 : c]$ serves as the result. In the second case, the sum $\sum_{k=a+1}^{b} A[k] \bmod n = 0$, so the subarray $A[a+1 : b]$ can be returned as the result.

```
vector<int> find_0_sum_subset(const vector<int> &A) {
  vector<int> prefix_sum(A);
  for (int i = 0; i < prefix_sum.size(); ++i) {
    prefix_sum[i] += i > 0 ? prefix_sum[i - 1] : 0;
    prefix_sum[i] %= A.size();
  }

  vector<int> table(A.size(), -1);
  for (int i = 0; i < A.size(); ++i) {
    if (prefix_sum[i] == 0) {
      vector<int> ans(i + 1);
      iota(ans.begin(), ans.end(), 0);
      return ans;
    } else if (table[prefix_sum[i]] != -1) {
      vector<int> ans(i - table[prefix_sum[i]]);
      iota(ans.begin(), ans.end(), table[prefix_sum[i]] + 1);
      return ans;
    }
    table[prefix_sum[i]] = i;
  }
}
```

Problem 6.6, *pg. 55*: *Design and implement an algorithm that takes as input an array A of n elements, and returns the beginning and ending indices of a longest increasing subarray of A.*

Solution 6.6: The brute-force algorithm is to compute for each i the length m_i of the longest increasing subarray ending at i. This is $m_{i-1} + 1$ if $i \neq 0$ and $A[i-1] < A[i]$, and 1 otherwise. The brute-force algorithm has time complexity $O(n)$, and the space complexity can be reduced to $O(1)$.

We can heuristically improve upon the brute-force algorithm by observing that if $A[i-1] \not< A[i]$ (i.e., we are starting to look for a new subarray starting at i) and the longest contiguous subarray seen up to index i has length L, we can move on to index $i+L$ and work backwards towards i; specifically, if for any $j, i \leq j < i+L$ we have $A[j] \not< A[j+1]$, we can skip the remaining indices.

This is a heuristic in that it does not improve the worst-case complexity—if the array consists of alternating 0s and 1s, we still examine each element—but the best case complexity reduces to $O(\max(n/L, L))$, where L is the length of the longest increasing subarray.

The average case time complexity depends on the probability distribution function for the input, and in general is very difficult to compute. For example, if A is a random permutation, or its entries are independent and uniform in $[0,1]$, it is known that the expected value for L is $\times(\log n / \log\log n)$. If A's entries are independent identically distributed Bernoulli random variables, the longest contiguous nondecreasing subarray has length $\times(\log n)$ in expectation. Both these facts are difficult to prove, and their implications to the average case time complexity are even harder to analyze.

```
template <typename T>
pair<int, int> find_longest_increasing_subarray(const vector<T> &A) {
  int max_len = 1;
  pair<int, int> ans(0, 0);
  int i = 0;
  while (i < A.size()) {
    // Check backwardly and skip if A[j] >= A[j + 1]
    bool is_skippable = false;
    for (int j = i + max_len - 1; j >= i; --j) {
      if (A[j] >= A[j + 1]) {
        i = j + 1;
        is_skippable = true;
        break;
      }
    }

    // Check forwardly if it is not skippable
    if (is_skippable == false) {
      i += max_len - 1;
      while (i + 1 < A.size() && A[i] < A[i + 1]) {
        ++i, ++max_len;
      }
      ans = {i - max_len + 1, i};
    }
  }
  return ans;
}
```

Problem 6.7, *pg. 55*: *How would you compute the weakest implied equivalence relation given n, A, and B? You do not have access to any data structure libraries.*

Solution 6.7: The basic idea is to start by mapping each element to itself. This

mapping is stored in an array F, and can be viewed as implementing a tree relation, with $F[i]$ being i's parent. We iterate through A and B. Since $A[i]$ and $B[i]$ are equivalent, we scan $A[i]$'s ancestors and $B[i]$'s ancestors and update $A[i]$'s (or $B[i]$'s) ancestor to the ancestor which has the smaller index. After all entries in A and B are processed, we make a last pass through F, compressing the ancestor tree, since some parent relationship may have been updated as we iterated through A and B. We return the result as array F; $F[i]$ is the element with the smallest index in the equivalence class of element i.

```
int backtrace(const vector<int> &F, int idx) {
  while (F[idx] != idx) {
    idx = F[idx];
  }
  return idx;
}

/*
 * A and B encode pairwise equivalences on a cardinality N set whose elements
 * are indexed by 0, 1, 2, ..., N-1.
 *
 * For example A[i] = 6 and B[i] = 0 indicates that the 6 and 0 are to be
 * grouped into the same equivalence class.
 *
 * We return the weakest equivalence relation implied by A and B in an array
 * F of length N; F[i] holds the smallest index of all the elements that
 * i is equivalent to.
 */
vector<int> compute_equival_classes(const int &n, const vector<int> &A,
                                    const vector<int> &B) {
  // Each element maps to itself
  vector<int> F(n);
  iota(F.begin(), F.end(), 0);

  for (int i = 0; i < A.size(); ++i) {
    int a = backtrace(F, A[i]), b = backtrace(F, B[i]);
    a < b ? F[b] = a : F[a] = b;
  }

  // Generate the weakest equivalence relation
  for (int &f : F) {
    while (f != F[f]) {
      f = F[f];
    }
  }
  return F;
}
```

Problem 6.8, ***pg.55***: *Suppose you know the permutation σ and the extract sequence $\langle i_0, i_1, \ldots, i_{m-1} \rangle$ in advance. How would you efficiently compute the order in which the m elements are removed from S?*

Solution 6.8: Our algorithm maintains a collection of subsets $\{R_0, R_1, \ldots, R_m\}$ that

partitions $\mathcal{Z}_n$. Specifically, R_k, for $1 \leq k \leq m-1$ consists of elements in σ whose indices are greater than i_{k-1} and less than or equal to i_k. Subset R_0 is all elements in σ with indices less than or equal to i_0. Subset R_m is all elements in σ with indices greater than i_{m-1}. It follows from the definition that $i_{k-1} = i_k$ implies R_k is empty.

We process each $t \in [0, n-1]$ in ascending order. For each t, we determine if it is extracted, and, if it is extracted, when it is extracted. We do this by seeing which R_k it belongs to. If $k = m$, i is never extracted. Otherwise i is removed in the k-th extraction. Consequently, we remove R_k from the partition and add all its elements to the first subset that exists in the partition such that $R_{k'}, k' > k$.

The time complexity is dominated by forming the union of disjoint-sets, and finding the set each of element belongs to. The disjoint-set data structure is ideally suited for union-find and has a run time that is essentially linear.

```
int find_set(vector<int> &set, const int &x) {
  if (set[x] != x) {
    set[x] = find_set(set, set[x]);  // path compression
  }
  return set[x];
}

void union_set(vector<int> &set, const int &x, const int &y) {
  int x_root = find_set(set, x), y_root = find_set(set, y);
  set[min(x_root, y_root)] = max(x_root, y_root);
}

vector<int> offline_minimum(const vector<int> &A, const vector<int> &E) {
  vector<int> R(A.size(), E.size());
  int pre = 0;

  // Initialize the collection of subsets
  for (int i = 0; i < E.size(); ++i) {
    for (int j = pre; j <= E[i]; ++j) {
      R[A[j]] = i;
    }
    pre = E[i] + 1;
  }

  vector<int> ret(E.size(), -1);  // stores the answer
  vector<int> set(E.size() + 1);  // the disjoint-set
  iota(set.begin(), set.end(), 0);  // initializes the disjoint-set
  for (int i = 0; i < A.size(); ++i) {
    if (find_set(set, R[i]) != E.size() && ret[find_set(set, R[i])] == -1) {
      ret[set[R[i]]] = i;
      union_set(set, set[R[i]], set[R[i]] + 1);
    }
  }
  return ret;
}
```

Problem 6.9, *pg.55*: *Write a function that takes two strings representing integers, and returns an integer representing their product.*

Solution 6.9: We mimic the grade school algorithm for multiplication, i.e., shift, multiply by a digit, and add. The number of digits required for the product is either $n+m$ or $n+m-1$ for n and m digit operands, so we allocate a string of size $n+m$ for the result; the computation determines whether the number of digits is $n+m$ or $n+m-1$. We do not store all the partial products, and then add them; rather we add each partial product into the result.

```
class BigInt {
  private:
    int sign;  // -1 or 1;
    vector<char> digits;

  public:
    BigInt(const int &capacity) : sign(1), digits(capacity) {}

    BigInt(const string &s) : sign(s[0] == '-' ? -1 : 1),
                              digits(s.size() - (s[0] == '-')) {
        for (int i = s.size() - 1, j = 0; i >= (s[0] == '-'); --i, ++j) {
          if (isdigit(s[i])) {
            digits[j] = s[i] - '0';
          }
        }
    }

    BigInt operator*(const BigInt &n) const {
      BigInt result(digits.size() + n.digits.size());
      result.sign = sign * n.sign;
      int i, j;
      for (i = 0; i < n.digits.size(); ++i) {
        if (n.digits[i]) {
          int carry = 0;
          for (j = 0; j < digits.size() || carry; ++j) {
            int n_digit = result.digits[i + j] +
                          (j < digits.size() ? n.digits[i] * digits[j] : 0) +
                          carry;
            result.digits[i + j] = n_digit % 10;
            carry = n_digit / 10;
          }
        }
      }

      // If one number is 0, the result size should be 0
      if ((digits.size() == 1 && digits.front() == 0) ||
          (n.digits.size() == 1 && n.digits.front() == 0)) {
        result.digits.resize(1);
      } else {
        result.digits.resize(i + j - 1);
      }
      return result;
    }
};
```

e-**Variant 6.9.1**: Solve the same problem when numbers are represented as lists of

digits.

Problem 6.10, ***pg. 56***: *Given an array A of n elements and a permutation Π, compute $\Pi(A)$ using only constant additional storage.*

Solution 6.10: We can use the fact that every permutation can be expressed as a composition of disjoint cycles, with the decomposition being unique up to ordering.

For example, the permutation $(2, 0, 1, 3)$ can be represented as $(0, 2, 1)(3)$, i.e., we can achieve the permutation $(2, 0, 1, 3)$ by these two moves: $0 \mapsto 2, 2 \mapsto 1, 1 \mapsto 0$, and $3 \mapsto 3$.

If the permutation is presented as a set of disjoint cycles, it can easily be applied using a constant amount of additional storage since we just need to perform rotation by one element. Therefore we want to identify the disjoint cycles that constitute the permutation.

It is straightforward to identify the set of cycles with an additional n bits. Start from any position and keep going forward (from i to $A[i]$) till the initial index is reached, at which point one of the cycles has been found. Then go to another position that is not yet part of any cycle. Finding a position that is not already a part of a cycle is trivial using one bit per array element.

One way to perform this without explicitly using additional $O(n)$ storage is to use the sign bit in the integers that constitute the permutation: Specifically, we subtract n from each entry in `perm` after it has been applied. We check if the element at index i has already been moved by seeing if `perm`$[i]$ is negative.

```
template <typename T>
void apply_permutation1(vector<int> &perm, vector<T> &A) {
  for (int i = 0; i < A.size(); ++i) {
    if (perm[i] >= 0) {
      int a = i;
      T temp = A[i];
      do {
        int next_a = perm[a];
        T next_temp = A[next_a];
        A[next_a] = temp;
        // Mark a as visited by using the sign bit
        perm[a] -= perm.size();
        a = next_a, temp = next_temp;
      } while (a != i);
    }
  }

  // Restore perm back
  size_t size = perm.size();
  for_each(perm.begin(), perm.end(), [size](T &x) { x += size; });
}
```

The code above will apply the permutation in $O(n)$ time but implicitly uses $\Theta(n)$ additional storage, even if it is borrowed from the sign bit of the entries of the `perm` array. We restore `perm` by adding n to each entry after the permutation has been applied.

We can avoid using $\Theta(n)$ additional storage by going from left-to-right and applying the cycle only if the current position is the leftmost position in the cycle. Testing whether the current position is the leftmost position, entails traversing the cycle once more, which increases the run time to $O(n^2)$.

```
template <typename T>
void apply_permutation2(vector<int> &perm, vector<T> &A) {
  for (int i = 0; i < A.size(); ++i) {
    // Traverse the cycle to see if i is the min element
    bool is_min = true;
    int j = perm[i];
    while (j != i) {
      if (j < i) {
        is_min = false;
        break;
      }
      j = perm[j];
    }

    if (is_min) {
      int a = i;
      T temp = A[i];
      do {
        int next_a = perm[a];
        T next_temp = A[next_a];
        A[next_a] = temp;
        a = next_a, temp = next_temp;
      } while (a != i);
    }
  }
}
```

Problem 6.11, ***pg. 56***: *Given an array A of integers representing a permutation Π, update A to represent Π^{-1} using only constant additional storage.*

Solution 6.11: The solution is similar to Solution 6.10 on the facing page. All that is needed is to decompose the permutation into a set of cycles and invert each cycle one step back. For example, the permutation $(2, 0, 1, 3)$ can be represented as $(0, 2, 1)(3)$. Hence the inverse can be represented as $(1, 2, 0)(3)$ which amounts to $(1, 2, 0, 3)$.

To save additional space, we can use exactly the same set of tricks as in Solution 6.10 on the preceding page.

Problem 6.12, ***pg. 56***: *Given a permutation p represented as a vector, return the vector corresponding to the next permutation under lexicographic ordering. If p is the last permutation, return empty vector. For example, if $p = (1, 0, 3, 2)$, your function should return $(1, 2, 0, 3)$.*

Solution 6.12: The key insight is that if $p[k] < p[k+1]$, and for all $i > k$, $p[i] \geq p[i+1]$, then no permutation of the elements consequent to k will lead to a permutation that is ahead of p in the lexicographic order. Therefore, we must increase $p[k]$. To

obtain the next permutation we find the largest index l such that $p[l] > p[k]$ (such an l must exist since $p[k] < p[k+1]$). Swapping $p[l]$ and $p[k]$ leaves the sequence after position k in decreasing order. Reversing this sequence after position k produces its lexicographically minimal permutation, and the lexicographic successor of the original p.

To find the previous permutation, we apply the same idea with some modifications.

```
vector<int> next_permutation(vector<int> p) {
  int k = p.size() - 2;
  while (k >= 0 && p[k] >= p[k + 1]) {
    --k;
  }
  if (k == -1) {
    return {};  // p is the last permutation
  }

  int l;
  for (int i = k + 1; i < p.size(); ++i) {
    if (p[i] > p[k]) {
      l = i;
    } else {
      break;
    }
  }
  swap(p[k], p[l]);

  // Produce the lexicographically minimal permutation
  reverse(p.begin() + k + 1, p.end());
  return p;
}
```

Variant 6.12.1: Compute the k-th permutation under lexicographic ordering, starting from the identity permutation, which is the first permutation in lexicographic ordering.

e-Variant 6.12.2: Given a permutation p represented as a vector, return the vector corresponding to the *previous* permutation of p under lexicographic ordering.

Problem 6.13, *pg. 56*: *Design a $\Theta(n)$ algorithm for rotating an array A of n elements to the right by i positions. You are allowed $O(1)$ additional storage.*

Solution 6.13: This is a special case of applying a permutation with constant additional storage (Problem 6.10 on Page 56) with the permutation corresponding to a rotation. A rotation corresponds to a set of cycles of the form $\langle c, (i+c) \bmod n, (2i+c) \bmod n, \ldots, (mi+c) \bmod n \rangle$ for a number of different values of c. For example, for the case where $n = 6$ and $i = 2$, the corresponding cycles are $(0, 2, 4)$ and $(1, 3, 5)$. When $n = 15$ and $i = 6$, the cycles are $\langle 0, 6, 12, 3, 9 \rangle$, $\langle 1, 7, 13, 4, 10 \rangle$, and $\langle 2, 8, 14, 5, 11 \rangle$.

These examples lead us to conjecture the following:

(1.) All cycles have the same length, and are a shifted version of the cycle $\langle 0, i \bmod n, 2i \bmod n, \ldots (l-1)i \bmod n \rangle$.

(2.) The number of cycles is the GCD of n and i.

These conjectures can be justified on heuristic grounds, specifically from considering the prime factorizations for i and n. See on the next page for the formal proof.

Assuming these conjectures to be correct, we can apply the rotation one cycle at a time, as follows. The first elements of the different cycles are at indices $0, 1, 2, \ldots, \text{GCD}(n,i) - 1$. For each cycle, we assign the index of first element to a temporary variable j. We iteratively move the element at j to $(j+i) \bmod n$ and update j to $(j+i) \bmod n$, stopping after $n/\text{GCD}(n,i)$ moves. This takes $O(1)$ space: a variable to track which cycle we are processing, a variable to track how many elements we have processed in the current cycle, as well as temporary variables for performing the move.

```
template <typename T>
void rotate_array(vector<T> &A, int i) {
  i %= A.size();
  int cycles = GCD(A.size(), i);  // number of cycles in this rotation
  int hops = A.size() / cycles;  // number of elements in a cycle

  for (int c = 0; c < cycles; ++c) {
    T temp = A[c];
    for (int j = 1; j < hops; ++j) {
      swap(A[(c + j * i) % A.size()], temp);
    }
    A[c] = temp;
  }
}
```

We now provide an alternative to the permutation approach. The new solution works well in practice and is considerably simpler. Assume that $A = \langle 1, 2, 3, 4, a, b \rangle$, and $i = 2$. Then in the rotated A there are two subarrays, $\langle 1, 2, 3, 4 \rangle$ and $\langle a, b \rangle$ that keep their original orders. Therefore, rotation can be seen as the exchanges of the two subarrays of A. To achieve these exchanges using only $O(1)$ space we use a `reverse` function. Using A and i as an example, we first reverse A to get A' ($\langle 1, 2, 3, 4, a, b \rangle \mapsto \langle b, a, 4, 3, 2, 1 \rangle$), then reverse the first i elements of A' ($\langle b, a, 4, 3, 2, 1 \rangle \mapsto \langle a, b, 4, 3, 2, 1 \rangle$), and reverse the remaining elements starting from the i-th element of A' ($\langle a, b, 4, 3, 2, 1 \rangle \mapsto \langle a, b, 1, 2, 3, 4 \rangle$) which yields the rotated A. Following is the code in C++:

```
template <typename T>
void rotate_array(vector<T> &A, int i) {
  i %= A.size();
  reverse(A.begin(), A.end());
  reverse(A.begin(), A.begin() + i);
  reverse(A.begin() + i, A.end());
}
```

We now prove Conjectures (1.) and (2.).

Proof:

First we prove that rotation does result in cycles. Take l_0 to be the largest integer such that the sequence $\sigma_0 = \langle 0, i \bmod n, 2i \bmod n, 3i \bmod n, \ldots, ((l_0 - 1)i) \bmod n \rangle$ does not repeat. We claim that $(l_0 i) \bmod n = 0$. Since l_0 was defined to be maximal, it must be that $(l_0 i) \bmod n$ is a value that is already in σ_0. For contradiction, suppose it equals $\sigma_0(r), 0 < r < (l_0 - 1)$. Then

$$\begin{aligned} \sigma_0(l_0 - r) \bmod n &= ((l_0 - r)i) \bmod n \\ &= ((l_0 i) \bmod n - (ri) \bmod n) \bmod n \\ &= ((l_0 i) \bmod n - \sigma_0(r) \bmod n) \bmod n \\ &= 0 \bmod n. \end{aligned}$$

Hence the sequence repeats at $\sigma_0(l_0 - r)$, contradicting the maximality of l_0.

Define l_c to be the largest integer such that the sequence $\sigma_c = \langle c, (i + c) \bmod n, (2i + c) \bmod n, (3i + c) \bmod n, \ldots, ((l_c - 1)i + c) \bmod n \rangle$ does not repeat. Conjecture (1.) on the preceding page, namely that all cycles have the same length, follows from the observation that the difference between $(ij) \bmod n$ and $(ij + c) \bmod n$ always equals $c \bmod n$.

Now we prove Conjecture (2.) on the previous page, i.e., there exist exactly GCD(n, i) cycles. Since we have just seen that all cycles have the same length, it suffices to prove that the length of the cycle containing 0 is $n/\text{GCD}(n, i)$.

Let g be the smallest integer greater than 0 that appears in the cycle which contains 0. Because of the modulus operation, g is not necessarily the number that follows 0, e.g., when $n = 15$ and $i = 6$, g is 3, even though the cycle corresponding to 0 is $\langle 0, 6, 12, 3, 9 \rangle$. The set S_0 of numbers in the cycle that 0 belongs to is $\{x \mid \exists j\ x = ij \bmod n\}$. Equivalently, $S_0 = \{x \mid \exists a \exists b\ x = (ai + bn) \bmod n\}$. It is a basic fact that the GCD of i and n is the smallest positive integer of the form $ai + bn$, with a and b being arbitrary integers. Therefore g is the GCD of n and i.

We claim that S_0 is exactly equal to the set of numbers in $\{0, 1, \ldots, n - 1\}$ that are divisible by g. Conjecture (2.) follows from the fact that exactly $n/\text{GCD}(n, i)$ numbers in $\{0, 1, \ldots, n - 1\}$ are divisible by g.

First we prove that all numbers in S_0 are divisible by g. If not, say $e = ij \bmod n$ is not divisible by g. Then $e = gq + r$, where $r \in (0, g)$ is the remainder. Since $g = ai + bn$ for some a and b, we have $r = (e - (ai + bn)) \bmod n = (e - ai) \bmod n$. Since e lies in S_0, all numbers of the form $(e + Gi) \bmod n$, where $G \geq 0$, also lie in S_0. In particular, let H be such that $Hn - a \geq 0$. Then $(e + (Hn - a)i) \bmod n$ lies in S_0. But $(e + (Hn - a)i) \bmod n = (e - ai) \bmod n = r$, which contradicts the minimality of g.

Now we show that $gI \bmod n \in S_0$ for all I. Since $g = ai + bn$, for some a and b, we have $gI \bmod n = (aiI + bnI) \bmod n = aIi \bmod n$. Let J be such that $(Jn + aI) \geq 0$. Then $(Jn + aI)i \bmod n$ is in S_0. But $(Jn + aI)i \bmod n = aIi \bmod n = gI \bmod n$, demonstrating that $gI \bmod n \in S_0$.

Problem 6.14, ***pg.57****:* *Check whether a* 9×9 *2D array representing a partially completed Sudoku is valid. Specifically, check that no row, column, and* 3×3 *2D subarray contains duplicates. A 0-value in the 2D array indicates that entry is blank; every other entry is in* $[1, 9]$.

Solution 6.14: We need to check nine row constraints, nine column constraints, and nine sub-grid constraints. We use bit arrays to test for constraint violations, that is to ensure no number in $[1, 9]$ appears more than once.

```
// Check if a partially filled matrix has any conflicts
bool is_valid_Sudoku(const vector<vector<int>> &A) {
  // Check row constraints
  for (int i = 0; i < A.size(); ++i) {
    vector<bool> is_present(A.size() + 1, false);
    for (int j = 0; j < A.size(); ++j) {
      if (A[i][j] != 0 && is_present[A[i][j]] == true) {
        return false;
      } else {
        is_present[A[i][j]] = true;
      }
    }
  }

  // Check column constraints
  for (int j = 0; j < A.size(); ++j) {
    vector<bool> is_present(A.size() + 1, false);
    for (int i = 0; i < A.size(); ++i) {
      if (A[i][j] != 0 && is_present[A[i][j]] == true) {
        return false;
      } else {
        is_present[A[i][j]] = true;
      }
    }
  }

  // Check region constraints
  int region_size = sqrt(A.size());
  for (int I = 0; I < region_size; ++I) {
    for (int J = 0; J < region_size; ++J) {
      vector<bool> is_present(A.size() + 1, false);
      for (int i = 0; i < region_size; ++i) {
        for (int j = 0; j < region_size; ++j) {
          if (A[region_size * I + i][region_size * J + j] != 0 &&
              is_present[A[region_size * I + i][region_size * J + j]]) {
            return false;
          } else {
            is_present[A[region_size * I + i][region_size * J + j]] = true;
          }
        }
      }
    }
  }
  return true;
}
```

Solution 17.8 on Page 395 describes how to solve Sudoku instances using branch and bound.

Problem 6.15, ***pg. 57****: Implement a function which takes a 2D array A and prints A in spiral order.*

Solution 6.15: The outermost elements of an $n \times n$ 2D array can be written in spiral order using four iterations: elements $(0,0)$ to $(0,n-2)$, then elements $(0,n-1)$ to $(n-2,n-1)$, followed by elements $(n-1,n-1)$ to $(n-1,1)$, and finally elements $(n-1,0)$ to $(1,0)$. After this, we are left with the problem of printing the elements of an $(n-2) \times (n-2)$ 2D array in spiral order. This leads to an iterative algorithm that prints the outermost elements of $n \times n, (n-2) \times (n-2), (n-4) \times (n-4), \ldots$ 2D arrays.

```
void print_matrix_clockwise(const vector<vector<int>> &A, const int &offset) {
  if (offset == A.size() - offset - 1) {  // for matrix with odd size
    cout << A[offset][offset];
  }

  for (int j = offset; j < A.size() - offset - 1; ++j) {
    cout << A[offset][j] << ' ';
  }
  for (int i = offset; i < A.size() - offset - 1; ++i) {
    cout << A[i][A.size() - offset - 1] << ' ';
  }
  for (int j = A.size() - offset - 1; j > offset; --j) {
    cout << A[A.size() - offset - 1][j] << ' ';
  }
  for (int i = A.size() - offset - 1; i > offset; --i) {
    cout << A[i][offset] << ' ';
  }
}

void print_matrix_in_spiral_order(const vector<vector<int>> &A) {
  for (int offset = 0; offset < ceil(0.5 * A.size()); ++offset) {
    print_matrix_clockwise(A, offset);
  }
}
```

An alternate solution in C++ writes 0 into array entries to indicate they have been processed, and a `shift` 2D array to compress the four iterations above into a single iterations parametrized by `shift`:

```
void print_matrix_spiral(vector<vector<int>> A) {
  const array<array<int, 2>, 4> shift = {0, 1, 1, 0, 0, -1, -1, 0};
  int dir = 0, x = 0, y = 0;

  for (int i = 0; i < A.size() * A.size(); ++i) {
    cout << A[x][y] << ' ';
    A[x][y] = 0;
    int nx = x + shift[dir][0], ny = y + shift[dir][1];
    if (nx < 0 || nx >= A.size() || ny < 0 || ny >= A.size() ||
        A[nx][ny] == 0) {
      dir = (dir + 1) & 3;
```

```
12        nx = x + shift[dir][0], ny = y + shift[dir][1];
13      }
14      x = nx, y = ny;
15    }
16  }
```

ϵ-Variant 6.15.1: Given a dimension d, write a program to generate a $d \times d$ 2D array which when printed in spiral order outputs the sequence $\langle 1, 2, 3, \ldots, d^2 \rangle$. For example, if $d = 3$, the result should be

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 8 & 9 & 4 \\ 7 & 6 & 5 \end{bmatrix}.$$

ϵ-Variant 6.15.2: Given a sequence of integers σ, compute a 2D array A which when printed in spiral order yields σ. (Assume $|\sigma| = n^2$ for some integer n.)

ϵ-Variant 6.15.3: Write a program to enumerate the first n pairs of integers (a, b) in spiral order, starting from $(0, 0)$ followed by $(1, 0)$. For example, if $n = 10$, your output should be $(0, 0), (1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), (-1, 1), (0, 1), (1, 1), (2, 1)$.

Problem 6.16, *pg. 58*: *Implement a routine that takes a $D \times D$ Boolean array A together with an entry (x, y) and flips the color of the region associated with (x, y). See Figure 6.5 on Page 58 for an example of flipping.*

Solution 6.16: Conceptually, we solve this problem by maintaining a queue `q` of entries to process, and a 2D Boolean array `processed` indicating whether an entry has been processed. Initially, all entries in `processed` are marked `false` and `q` contains (x, y). The queue is popped iteratively, and the neighbors of the popped element are examined. Any neighbor which is unprocessed and whose color needs to be changed is added to `q`. After its neighbors have been examined, the processed status of the element just popped is set to `true` and its color is flipped. The computation ends when `q` is empty.

In practice, we do not require the `processed` array, since there are only two colors. We only need to record the color of the initial entry, and compare new entries with that color.

```
1  void flip_color(vector<vector<bool>> &A, const int &x, const int &y) {
2    const array<array<int, 2>, 4> dir = {-1, 0, 1, 0, 0, -1, 0, 1};
3    const bool color = A[x][y];
4
5    queue<pair<int, int>> q;
6    q.emplace(x, y);
7    while (q.empty() == false) {
8      pair<int, int> curr(q.front());
9      A[curr.first][curr.second] = !A[curr.first][curr.second];  // flip color
10     for (auto &d : dir) {
```

```
      pair<int, int> next(curr.first + d[0], curr.second + d[1]);
      if (next.first >= 0 && next.first < A.size() &&
          next.second >= 0 && next.second < A[next.first].size() &&
          A[next.first][next.second] == color) {
        q.emplace(next);
      }
    }
    q.pop();
  }
}
```

We also provide a recursive solution which does not need a queue but implicitly uses a stack. This solution does not require a processed array; instead, we temporarily flip the color of the entry being processed.

```
void flip_color(vector<vector<bool>> &A, const int &x, const int &y) {
  const array<array<int, 2>, 4> dir = {-1, 0, 1, 0, 0, -1, 0, 1};
  const bool color = A[x][y];
  A[x][y] = !A[x][y];  // flip the color

  for (auto &d : dir) {
    const int nx = x + d[0], ny = y + d[1];
    if (nx >= 0 && nx < A.size() && ny >= 0 && ny < A[nx].size() &&
        A[nx][ny] == color) {
      flip_color(A, nx, ny);
    }
  }
}
```

ϵ-Variant 6.16.1: Design an algorithm for computing the black region that contains the most points.

ϵ-Variant 6.16.2: Design an algorithm that takes a point (a, b), sets $A(a, b)$ to black, and returns the size of the black region that contains the most points. Assume this algorithm will be called multiple times, and you want to keep the aggregate run time as low as possible.

Problem 6.17, *pg. 58*: *Design an algorithm that rotates a $n \times n$ 2D array by 90 degrees clockwise. Assume that $n = 2^k$ for some positive integer k. What is the time complexity of your algorithm?*

Solution 6.17: It is natural to use recursion: decompose the 2D array into four equal-sized subarrays, $A[0 : \frac{n}{2} - 1][0 : \frac{n}{2} - 1]$, $A[0 : \frac{n}{2} - 1][\frac{n}{2} : n - 1]$, $A[\frac{n}{2} : n - 1][0 : \frac{n}{2} - 1]$, and $A[\frac{n}{2} : n-1][\frac{n}{2} : n-1]$, and recursively rotate each of these. Consequently, make a copy C of $A[0 : \frac{n}{2} - 1][0 : \frac{n}{2} - 1]$, and copy $A[0 : \frac{n}{2} - 1][\frac{n}{2} : n - 1]$ into $A[0 : \frac{n}{2} - 1][0 : \frac{n}{2} - 1]$, $A[\frac{n}{2} : n - 1][\frac{n}{2} : n - 1]$ into $A[0 : \frac{n}{2} - 1][\frac{n}{2} : n - 1]$, $A[\frac{n}{2} : n - 1][0 : \frac{n}{2} - 1]$ into $A[\frac{n}{2} : n - 1][\frac{n}{2} : n - 1]$ and C into $A[\frac{n}{2} : n - 1][0 : \frac{n}{2} - 1]$. The run time satisfies the recurrence $T(n) = 4T(\frac{n}{4}) + O(n)$, which solves to $O(n \log n)$, where $n = 2^{2k}$.

```
template <typename T>
void copy_matrix(vector<vector<T>> &A, const int &A_x_s, const int &A_x_e,
                 const int &A_y_s, const int &A_y_e,
                 const vector<vector<T>> &S,
                 const int &S_x, const int &S_y) {
  for (int i = 0; i < A_x_e - A_x_s; ++i) {
    copy(S[S_x + i].cbegin() + S_y, S[S_x + i].cbegin() + S_y + A_y_e - A_y_s,
         A[A_x_s + i].begin() + A_y_s);
  }
}

template <typename T>
void rotate_matrix_helper(vector<vector<T>> &A, const int &x_s,
                          const int &x_e, const int &y_s, const int &y_e) {
  if (x_e > x_s + 1) {
    int mid_x = x_s + ((x_e - x_s) >> 1), mid_y = y_s + ((y_e - y_s) >> 1);
    // Move submatrices
    vector<vector<T>> C(mid_x - x_s, vector<T>(mid_y - y_s));
    copy_matrix(C, 0, C.size(), 0, C.size(), A, x_s, y_s);
    copy_matrix(A, x_s, mid_x, y_s, mid_y, A, mid_x, y_s);
    copy_matrix(A, mid_x, x_e, y_s, mid_y, A, mid_x, mid_y);
    copy_matrix(A, mid_x, x_e, mid_y, y_e, A, x_s, mid_y);
    copy_matrix(A, x_s, mid_x, mid_y, y_e, C, 0, 0);

    // Recursively rotate submatrices
    rotate_matrix_helper(A, x_s, mid_x, y_s, mid_y);
    rotate_matrix_helper(A, x_s, mid_x, mid_y, y_e);
    rotate_matrix_helper(A, mid_x, x_e, mid_y, y_e);
    rotate_matrix_helper(A, mid_x, x_e, y_s, mid_y);
  }
}

template <typename T>
void rotate_matrix(vector<vector<T>> &A) {
  rotate_matrix_helper(A, 0, A.size(), 0, A.size());
}
```

Alternately, we could perform the rotation using $O(1)$ additional memory in $O(n)$ time by iterating through any one of the four subarrays, and rotating elements in sets of four.

Variant 6.17.1: Suppose the underlying hardware has support for fast two-dimensional block copies. Specifically, you can copy an $m \times m$ 2D array in $O(m)$ time. How can you exploit the hardware to reduce the time complexity?

Problem 6.18, *pg. 59*: *Implement run-length encoding and decoding functions. Assume the string to be encoded consists of letters of the alphabet, with no digits, and the string to be decoded is a valid encoding.*

Solution 6.18: The decoding function entails converting a number represented in decimal to its integer equivalent; the encoding function entails the reverse. Both of these are covered in Solution 5.6 on Page 176. The remainder of the code consists of

iterating through the input string and appending to the result string.

```
string decoding(const string &s) {
  int count = 0;
  string ret;
  for (const char &c : s) {
    if (isdigit(c)) {
      count = count * 10 + c - '0';
    } else {  // isalpha
      ret.append(count, c);
      count = 0;
    }
  }
  return ret;
}

string encoding(const string &s) {
  int count = 1;
  stringstream ss;
  for (int i = 1; i < s.size(); ++i) {
    if (s[i] == s[i - 1]) {
      ++count;
    } else {
      ss << count << s[i - 1];
      count = 1;
    }
  }
  ss << count << s.back();
  return ss.str();
}
```

Problem 6.19, *pg. 59*: *Implement a function for reversing the words in a string. Your function should use $O(1)$ space.*

Solution 6.19: The code for computing the position for each character in a single pass is fairly complex. However, a two stage iteration is easy. In the first step, reverse the entire string and in the second step, reverse each word. For example, "ram is costly" transforms to "yltsoc si mar", which transforms to "costly is ram". Here is code in C++:

```
void reverse_words(string &input) {
  // Reverse the whole string first
  reverse(input.begin(), input.end());

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

Problem 6.20, ***pg. 59***: *Given two strings s (the "search string") and t (the "text"), find the first occurrence of s in t.*

Solution 6.20: Knuth-Morris-Pratt, Boyer-Moore, and Rabin-Karp. are widely taught algorithms for substring search that run in linear time. In practice Boyer-Moore is the fastest string search algorithm, because for many applications, it runs in sub-linear time.

The Boyer-Moore algorithm works by trying to match characters of s in t at a certain offset in the reverse order (last character of s matched first). If we can match all the characters in s, then we have found a match; otherwise, we stop at the first mismatch. The key idea behind the Boyer-Moore algorithm is to skip as many offsets as possible when we are done matching characters at a given offset. We do this by building two tables—the good suffix shift table and the bad character shift table.

For a given character, the bad character shift table gives us the distance of the last occurrence of that character in s to the rightmost string. If the character does not occur in s, then the entry in the table is $|s|$. Hence when we find a character in t that does not match for the current offset, we know how much we must move forward so that this character can match for the first time.

The good suffix shift table is a little more complex. Conceptually, for a given suffix x of s, it tells us what is the shortest suffix y of s that is longer than x and has x as suffix. In practice what we store is how far can we move safely, given that we have matched up to $|x|$ characters but did not match the next character.

The Rabin-Karp algorithm is based on the idea of "fingerprinting". It compute hash codes of each substring $t[i : i+|s|-1]$ for $i = 0$ to $|t|-|s|$—these are the fingerprints. If $h(t[i : i+|s|-1]) \neq h(s)$, the $|s|$ length substring beginning at i cannot equal s. A good hash function is one where the probability of collisions is low and $h(t[i : i+|s|-1])$ can be incrementally computed, that is the time to compute $h(t[i : i+|s|-1])$, given $h(t[i-1 : i+|s|-2])$, is $O(1)$. (Such a hash function is sometimes referred to as a rolling hash.) The Rabin-Karp algorithm is very simple to implement, and generalizes more easily, e.g., to two dimension pattern matching, than the Knuth-Morris-Pratt and Boyer-Moore algorithms. The expected time complexity is $O(|s| + |t|)$.

```
1  const int base = 26, mod = 997;
2
3  int rabin_karp(const string &t, const string &s) {
4    if (s.size() > t.size()) {
5      return -1;  // s is not a substring of t
6    }
7
8    int t_hash = 0, s_hash = 0;
9    for (int i = 0; i < s.size(); ++i) {
10     t_hash = (t_hash * base + t[i]) % mod;
11     s_hash = (s_hash * base + s[i]) % mod;
12   }
13
14   for(int i = s.size(); i < t.size(); ++i) {
15     // In case of hash collision, check the two substrings are actually equal
16     if (t_hash == s_hash && t.compare(i - s.size(), s.size(), s) == 0) {
```

```
      return i - s.size();  // find match
    }
    t_hash -= (t[i - s.size()] * static_cast<int>(pow(base, s.size() - 1)))
              % mod;
    if (t_hash < 0) {
      t_hash += mod;
    }
    t_hash = (t_hash * base + t[i]) % mod;
  }

  if (t_hash == s_hash && t.compare(t.size() - s.size(), s.size(), s) == 0) {
    return t.size() - s.size();
  }
  return -1;  // s is not a substring of t
}
```

Problem 6.21, *pg. 60*: *Write a function which takes as input a string s, and removes each "b" and replaces each "a" by "dd". Use $O(1)$ additional storage—assume s is stored in an array that has enough space for the final result.*

Solution 6.21: We start by making a first pass through s in which we delete each "b" by maintaining a write index, `write_idx` and a current index, `cur_idx`—we achieve the effect of deleting "b" by skipping over "b". We also count the number of "a"s. We then make a second pass working backwards from the end of the current string, copying characters to the end of the resulting string (whose size we know from the number of "a"s). For each "a", we write "dd".

```
string replace_and_remove(string s) {
  // Remove "b" and count the number of "a"
  int write_idx = 0, a_count = 0;
  for (const char &c : s) {
    if (c != 'b') {
      s[write_idx++] = c;
    }
    if (c == 'a') {
      ++a_count;
    }
  }

  // Allocate space according to the number of "a"
  s.resize(write_idx + a_count);
  // Replace "a" with "dd";
  int cur_idx = write_idx - 1;
  write_idx = s.size() - 1;
  while (cur_idx >= 0) {
    if (s[cur_idx] == 'a') {
      s[write_idx--] = s[write_idx--] = 'd';
    } else {
      s[write_idx--] = s[cur_idx];
    }
    --cur_idx;
  }
  return s;
```

```
27 }
```

We can prove that the second step correctly replaces each "a" by "dd" by induction on the length of the string n.

Proof:

For the base case, i.e., length 1 string, there are two possibilities—the string is "a" or x, where x is one of {"b", "c", "d"}. For both possibilities, induction goes through. Assume by induction that the construction is correct for all strings of length $n > 1$. Consider a string s of length $n + 1$. Let's say the length of the final result is k. If s ends in "c" or "d", we copy $s[n-1]$ over to $s[k-1]$. By the induction hypothesis, our construction correctly copies the substring s^{n-1} consisting of the first $n-1$ characters of s to the remaining $k-1$ locations. If s ends in "a", we write "d" into locations $k-2$ and $k-1$. Now we have to process s^{n-1}, which will require $k-2$ locations. By the induction hypothesis, the construction correctly writes the result into these locations, and induction goes through.

ϵ-Variant 6.21.1: You have an array C of characters. The characters may be letters, digits, blanks, and punctuation. The telex-encoding of the array C is an array T of characters in which letters, digits, and blanks appear as before, but punctuation marks are spelled out. For example, telex-encoding entails replacing the character "." by the string "DOT", the character "," by "COMMA", the character "?" by "QUESTION MARK", and the character "!" by "EXCLAMATION MARK". Design an algorithm to perform telex-encoding with $O(1)$ space.

Variant 6.21.2: Write a function which merges two sorted arrays of integers, A and B. Specifically, the final result should be a sorted array of length $|A| + |B|$. Use $O(1)$ additional storage—assume the result is stored in A, which has sufficient space. These arrays are C-style arrays, i.e., contiguous preallocated blocks of memory.

Problem 6.22, *pg. 60*: *Given a cell phone keypad (specified by a mapping M that takes individual digits and returns the corresponding set of characters) and a number sequence, return all possible character sequences (not just legal words) that correspond to the number sequence.*

Solution 6.22: Recursion is natural. Let P be an n-digit number sequence. Assume these digits are indexed starting at 0, i.e., $P[0]$ is the first digit. Let S be a character sequence corresponding to the first k digits of P. We can generate all length n character sequences corresponding to P that have S as their prefix as follows. If $k = n$, there is nothing to do. Otherwise, we recurse on each length-$k+1$ sequence of the form Sx, for each $x \in M(P[k])$.

```
1 const array<string, 10> M = {"0", "1", "ABC", "DEF", "GHI", "JKL", "MNO",
2                               "PQRS", "TUV", "WXYZ"};
3
4 void phone_mnemonic_helper(const string &num, const int &d, string &ans) {
5   if (d == num.size()) {
```

```
    cout << ans << endl;
  } else {
    for (const char &c : M[num[d] - '0']) {
      ans[d] = c;
      phone_mnemonic_helper(num, d + 1, ans);
    }
  }
}

void phone_mnemonic(const string &num) {
  string ans(num.size(), 0);
  phone_mnemonic_helper(num, 0, ans);
}
```

Problem 6.23, *pg. 60*: *Design an algorithm that takes a string s and a string r, assumed to be a well-formed ESRE, and checks if r matches s.*

Solution 6.23: The key to solving this problem is using recursion effectively.

If r starts with ^, then the remainder of r, i.e., r^1, must strictly match a prefix of s. If r ends with a $, some suffix of s must be strictly matched by r without the trailing $. Otherwise, r must strictly match some substring of s.

Call the function that checks whether r strictly matches a prefix of string s `is_match`. This function has to check several cases:

(1.) Length-0 ESREs which match everything.
(2.) An ESRE starting with ^ or ending with $.
(3.) An ESRE starting with an alphanumeric character or dot.
(4.) An ESRE starting with a * match, e.g., a*wXY or .*Wa.

Case (1.) is a base case. Case (2.) involves a check possibly followed by a recursive call to `is_match_here`. Case (3.) requires a single call to `is_match_here`. Case (4.) is handled by a walk down the string s, checking that the prefix of s thus far matches the alphanumeric character or dot until some suffix of s is matched by the remainder of the ESRE, i.e., r^2.

```
bool is_match_here(const string &r, const string &s) {
  // Case (1.)
  if (r.empty()) {
    return true;
  }

  // Case (2) : ends with '$'
  if (r.front() == '$' && r.size() == 1) {
    return s.empty();
  }

  // Case (4.)
  if (r.size() >= 2 && r[1] == '*') {
    for (int i = 0; i < s.size() && (r.front() == '.' || r.front() == s[i]);
         ++i) {
      if (is_match_here(r.substr(2), s.substr(i + 1))) {
        return true;
      }
```

```
    }
    return is_match_here(r.substr(2), s);
  }

  // Case (3.)
  return !s.empty() && (r.front() == '.' || r.front() == s.front()) &&
         is_match_here(r.substr(1), s.substr(1));
}

bool is_match(const string &r, const string &s) {
  // Case (2.) : starts with '^'
  if (r.front() == '^') {
    return is_match_here(r.substr(1), s);
  }

  for (int i = 0; i <= s.size(); ++i) {
    if (is_match_here(r, s.substr(i))) {
      return true;
    }
  }
  return false;
}
```

ϵ-Variant 6.23.1: Solve the same problem for regular expressions without the ^ and $ operators.

Problem 7.1, *pg. 63*: *Write a function that takes L and F, and returns the merge of L and F. Your code should use $O(1)$ additional storage—it should reuse the nodes from the lists provided as input. Your function should use $O(1)$ additional storage, as illustrated in Figure 7.3 on Page 63. The only field you can change in a node is* `next`.

Solution 7.1: We traverse the lists, using one pointer per list, each initialized to the list head. We compare the contents of the pointer—the pointer with the lesser contents is to be added to the end of the result and advanced. If either pointer is `null`, we add the sublist pointed to by the other to the end of the result. The add can be performed by a single pointer update—it does not entail traversing the sublist. The worst case time complexity corresponds to the case when the lists are of comparable length. In the best case, one list is much shorter than the other and all its entries appear at the beginning of the merged list.

```
template <typename T>
void append_node(shared_ptr<node_t<T>> &head, shared_ptr<node_t<T>> &tail,
                 shared_ptr<node_t<T>> &n) {
  head ? tail->next = n : head = n;
  tail = n;  // reset tail to the last node
}

template <typename T>
void append_node_and_advance(shared_ptr<node_t<T>> &head,
                             shared_ptr<node_t<T>> &tail,
                             shared_ptr<node_t<T>> &n) {
```

```
12    append_node(head, tail, n);
13    n = n->next;  // advance n
14  }
15
16  template <typename T>
17  shared_ptr<node_t<T>> merge_sorted_linked_lists(shared_ptr<node_t<T>> F,
18                                                  shared_ptr<node_t<T>> L) {
19    shared_ptr<node_t<T>> sorted_head = nullptr, tail = nullptr;
20
21    while (F && L) {
22      append_node_and_advance(sorted_head, tail, F->data < L->data ? F : L);
23    }
24
25    // Append the remaining nodes of F
26    if (F) {
27      append_node(sorted_head, tail, F);
28    }
29    // Append the remaining nodes of L
30    if (L) {
31      append_node(sorted_head, tail, L);
32    }
33    return sorted_head;
34  }
```

e-**Variant 7.1.1**: Solve the same problem when the lists are doubly linked.

Problem 7.2, *pg. 63*: *Given a reference to the head of a singly linked list L, how would you determine whether L ends in a* `null` *or reaches a cycle of nodes? Write a function that returns* `null` *if there does not exist a cycle, and the reference to the start of the cycle if a cycle is present. (You do not know the length of the list in advance.)*

Solution 7.2: 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 iff 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 $\Theta(n)$ space, where n is the number of nodes in the list.

In some languages, e.g., C, the next field is a pointer. Typically, for performance reasons related to the memory subsystem on a processor, memory is allocated on word boundaries, and (at least) two of the least significant bits in the next pointer are 0. Bit fiddling can be used to set the least significant bit on the next pointer to mark whether a node as been visited. This approach has the disadvantage of changing the data structure—these updates can be undone later.

Another approach is to reverse the linked list, in the manner of Solution 7.9 on Page 215. If the head is encountered during the reversal, it means there is a cycle; otherwise we will get to the tail. Although this approach requires no additional storage, and runs in $O(n)$ time, it does modify the list.

A naïve approach that does not use additional storage and does not modify the

list is to walk the list in two loops—the outer loop visits the nodes one-by-one, and the inner loop starts from the head, and visits m nodes, where m is the number of nodes visited in the outer loop. 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 pointer, `slow`, and a fast pointer, `fast`, to visit the list. In each iteration, advance `slow` by one and `fast` by two. The list has a cycle iff the two pointers meet.

This is proved as follows.

Proof:

Number the nodes in the cycle by assigning first node encountered the index 0. Let C be the total number of nodes in the cycle. If the fast pointer reaches the first node at iteration F, at iteration $i \geq F$, it will be at node $2(i - F) \bmod C$. If the slow pointer reaches the first node at iteration S, at iteration $i \geq S$, it will be at node $(i - S) \bmod C$. The difference between the pointer locations after the slow pointer reaches the first node in the cycle is $2(i - F) - (i - S) \bmod C = i - (2F - S) \bmod C$. As i increases by one in each iteration, the equation $(i - (2F - S)) \bmod C = 0$ has a solution.

Now, assuming that we have detected a cycle using the above method, we find the start of the cycle, by first calculating the cycle length. We do this by freezing the fast pointer, and counting the number of times we have to advance the slow pointer to come back to the fast pointer. Consequently, we set both `slow` and `fast` pointers to the head. Then we advance `fast` by the length of the cycle, then move both `slow` and `fast` one at a time. The start of the cycle is located at the node where these two pointers meet again.

The code to do this traversal is quite simple in C++:

```
template <typename T>
shared_ptr<node_t<T>> has_cycle(const shared_ptr<node_t<T>> &head) {
  shared_ptr<node_t<T>> fast = head, slow = head;

  while (slow && slow->next && fast && fast->next && fast->next->next) {
    slow = slow->next, fast = fast->next->next;
    // Found cycle
    if (slow == fast) {
      // Calculate the cycle length
      int cycle_len = 0;
      do {
        ++cycle_len;
        fast = fast->next;
      } while (slow != fast);

      // Try to find the start of the cycle
      slow = head, fast = head;
      // Fast pointer advances cycle_len first
      while (cycle_len--) {
        fast = fast->next;
      }
      // Both pointers advance at the same time
```

```
      while (slow != fast) {
        slow = slow->next, fast = fast->next;
      }
      return slow;  // the start of cycle
    }
  }
  return nullptr;  // no cycle
}
```

ϵ-Variant 7.2.1: The following program purports to compute the beginning of the cycle without determining the length of the cycle; it has the benefit of being more succinct than the code listed above. Is the program correct?

```
template <typename T>
shared_ptr<node_t<T>> has_cycle(const shared_ptr<node_t<T>> &head) {
  shared_ptr<node_t<T>> fast = head, slow = head;

  while (slow && slow->next && fast && fast->next && fast->next->next) {
    slow = slow->next, fast = fast->next->next;
    // Found cycle
    if (slow == fast) {
      // Try 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;  // means no cycle
}
```

Problem 7.3, *pg. 63*: *Write a function that takes a sorted circular singly linked list and a pointer to an arbitrary node in this linked list, and returns the median of the linked list.*

Solution 7.3: We can solve this in stages. First we find n, the number of nodes. Then we identify the first node f with the minimum element. Finally, we return the $\lfloor \frac{n}{2} \rfloor$-th element if n is odd, with f being the 0-th element, and the average of the $\frac{n}{2}$-th and $(\frac{n}{2}+1)$-th elements if n is even. One corner case to watch out for is all entries being equal, which we check for in the first stage since we cannot find the first node with the minimum element.

```
template <typename T>
double find_median_sorted_circular_linked_list(
  const shared_ptr<node_t<T>> &r_node) {
  if (!r_node) {
    return 0.0;  // no node in this linked list
  }

  // Check all nodes are identical or not and identify the start of list
```

```
  shared_ptr<node_t<T>> curr = r_node, start = r_node;
  int count = 0;
  bool is_identical = true;
  do {
    if (curr->data != curr->next->data) {
      is_identical = false;
    }
    ++count, curr = curr->next;

    // start will point to the largest element in the list
    if (start->data <= start->next->data) {
      start = start->next;
    }
  } while (curr != r_node);
  // If all values are identical, median = curr->data
  if (is_identical == true) {
    return curr->data;
  }

  // Since start point to the largest element, its next is the start of list
  start = start->next;

  // Traverse to the middle of the list and return the median
  for (int i = 0; i < (count - 1) >> 1; ++i) {
    start = start->next;
  }
  return count & 1 ? start->data : 0.5 * (start->data + start->next->data);
}
```

Problem 7.4, *pg. 64*: *Let $h1$ and $h2$ be the heads of lists $L1$ and $L2$, respectively. Assume that $L1$ and $L2$ are well-formed, that is each consists of a finite sequence of nodes. (In particular, neither list has a cycle.) How would you determine if there exists a node r reachable from both $h1$ and $h2$ by following the next fields? If such a node exists, find the node that appears earliest when traversing the lists. You are constrained to use no more than constant additional storage.*

Solution 7.4: The lists overlap iff both have the same tail node: since each node has a single next field, once the lists converge at a node, they cannot diverge at a later node. Let $|L|$ denote the number of nodes in list L. Checking overlap amounts to finding the tail nodes for each, which is easily performed in $O(|L1| + |L2|)$ time and $O(1)$ space. To find the first node, we proceed as above, and in addition we compute $|L1|$ and $|L2|$. The first node is determined by first advancing through the longer list by $||L1| - |L2||$ nodes, and then advancing through both lists in lock-step, stopping at the first common node.

```
// Count the list length till end
template <typename T>
int count_len(shared_ptr<node_t<T>> L) {
  int len = 0;
  while (L) {
    ++len, L = L->next;
```

```
  }
  return len;
}

template <typename T>
void advance_list_by_k(shared_ptr<node_t<T>> &L, int k) {
  while (k--) {
    L = L->next;
  }
}

template <typename T>
shared_ptr<node_t<T>> overlapping_no_cycle_lists(shared_ptr<node_t<T>> L1,
                                                 shared_ptr<node_t<T>> L2) {
  // Count the lengths of L1 and L2
  int L1_len = count_len<T>(L1), L2_len = count_len<T>(L2);

  // Advance the longer list
  advance_list_by_k(L1_len > L2_len ? L1 : L2, abs(L1_len - L2_len));

  while (L1 && L2 && L1 != L2) {
    L1 = L1->next, L2 = L2->next;
  }
  return L1;  // nullptr means no overlap between L1 and L2
}
```

Figure 21.4 shows an example of lists which overlap and have cycles. For this example, both A and B are acceptable answers.

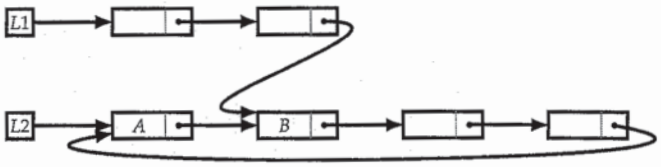


Figure 21.4: Overlapping lists.

Problem 7.5, *pg. 64*: *Solve Problem 7.4 on Page 64 for the case where L1 and L2 may each or both have a cycle. If such a node exists, return a node that appears first when traversing the lists. This node may not be unique—if L1 has a cycle $\langle n_0, n_1, \ldots, n_{k-1}, n_0 \rangle$, where n_0 is the first node encountered when traversing L1, then L2 may have the same cycle but a different first node.*

Solution 7.5: Suppose that one, or both, of the lists may have a cycle. Using the approach in Solution 7.2 on Page 208, we can determine in linear time and $O(1)$ space whether the lists have a cycle. If neither is cyclic, the lists are well-formed, and we can check overlap using the technique in Solution 7.4 on the preceding page. If one is cyclic, and the other is not, they cannot overlap.

If both are cyclic, and overlap, the cycles must be identical. Use the technique in Solution 7.2 on Page 208 to obtain nodes $a1$ and $a2$ on the cycle of $L1$ and $L2$. Visit the cycle from $a1$, stopping when $a1$ reappears. If $a2$ appears during this visit, the cycles are identical; otherwise, they are disjoint—the lists have no overlap. If there is an overlap, the problem specification allows us to return either $a1$ or $a2$. It is readily verified that the entire computation runs in $O(|L1| + |L2|)$ time and uses $O(1)$ space.

```
template <typename T>
shared_ptr<node_t<T>> overlapping_lists(shared_ptr<node_t<T>> L1,
                                        shared_ptr<node_t<T>> L2) {
  // Store the start of cycle if any
  shared_ptr<node_t<T>> s1 = has_cycle<T>(L1), s2 = has_cycle<T>(L2);

  if (!s1 && !s2) {
    return overlapping_no_cycle_lists(L1, L2);
  } else if (s1 && s2) {  // both lists have cycles
    shared_ptr<node_t<T>> temp = s2;
    do {
      temp = temp->next;
    } while (temp != s1 && temp != s2);
    return temp == s1 ? s1 : nullptr;
  }
  return nullptr;  // one list has cycle, one list has no cycle
}
```

Problem 7.6, *pg. 64*: *Write a function that takes a singly linked list L, and reorders the elements of L so that the new list represents even-odd(L). Your function should use $O(1)$ additional storage, as illustrated in Figure 7.6 on Page 65. The only field you can change in a node is* `next`.

Solution 7.6: We maintain two pointers, one iterates through the even elements, the other iterates through odd elements. We update the next field of the even pointer to the next of the odd pointer, and vice versa. Finally we update the next field of the last of the even elements to the head of the odd list. Care has to be taken to handle odd/even length lists uniformly, and to correctly process extreme cases (first and last nodes).

```
template <typename T>
shared_ptr<node_t<T>> even_odd_merge(const shared_ptr<node_t<T>> &L) {
  shared_ptr<node_t<T>> odd = L ? L->next : nullptr;
  shared_ptr<node_t<T>> odd_curr = odd;
  shared_ptr<node_t<T>> pre_even_curr = nullptr, even_curr = L;

  while (even_curr && odd_curr) {
    even_curr->next = odd_curr->next;
    pre_even_curr = even_curr;
    even_curr = even_curr->next;
    if (even_curr) {
      odd_curr->next = even_curr->next;
      odd_curr = odd_curr->next;
    }
```

```
  }

  // Odd number of nodes
  if (even_curr) {
    pre_even_curr = even_curr;
  }
  // Prevent empty list
  if (pre_even_curr) {
    pre_even_curr->next = odd;
  }
  return L;
}
```

Problem 7.7, *pg. 65*: *Let v be a node in a singly linked list L. Node v is not the tail; delete it in $O(1)$ time.*

Solution 7.7: This is more of a trick question than a conceptual one. Given the pointer to a node, it is impossible to delete it from the list without modifying its predecessor's next pointer and the only way to get to the predecessor is to traverse the list from head. However it is easy to delete the next node since it just requires modifying the next pointer of the current node. Now if we copy the value part of the next node to the current node, this would be equivalent to deleting the current node.

In practice this approach would not be acceptable, since it corrupts pointer-valued variables that point to v's successor.

```
template <typename T>
void deletion_from_list(const shared_ptr<node_t<T>> &v) {
  v->data = v->next->data;
  v->next = v->next->next;
}
```

Problem 7.8, *pg. 65*: *Given a singly linked list L and a number k, write a function to remove the k-th last element from L. Your algorithm cannot use more than a few words of storage, regardless of the length of the list. In particular, you cannot assume that it is possible to record the length of the list.*

Solution 7.8: We use two pointers, curr and ahead. First, the ahead pointer is advanced by k steps, and then curr and ahead advance in step. When ahead reaches null, curr points to the k-th last node in L, and we can remove it. Following is the code in C++:

```
template <typename T>
void remove_kth_last(shared_ptr<node_t<T>> &L, const int &k) {
  // Advance k steps first
  shared_ptr<node_t<T>> ahead = L;
  int num = k;
  while (ahead && num--) {
    ahead = ahead->next;
  }

```

```
  if (num) {
    throw length_error("not enough nodes in the list");
  }

  shared_ptr<node_t<T>> pre = nullptr, curr = L;
  // Find the k-th last node
  while (ahead) {
    pre = curr;
    curr = curr->next, ahead = ahead->next;
  }
  if (pre) {
    pre->next = curr->next;
  } else {
    L = curr->next;  // special case: delete L
  }
}
```

Problem 7.9, *pg. 65*: *Give a linear time non-recursive function that reverses a singly linked list. The function should use no more than constant storage beyond that needed for the list itself. The desired transformation is illustrated in Figure 7.7 on Page 65.*

Solution 7.9: The natural way of implementing the reversal is through recursion. However, this approach implicitly uses $\Theta(n)$ space on the stack. The function is not tail recursive, which precludes compilers from automatically converting the function to an iterative one.

Reversal can be performed iteratively—walk the list with two pointers, and update the trailing pointer's next field. It uses $O(1)$ additional storage, and has $\Theta(n)$ time complexity.

Recursive implementation, uses $\Theta(n)$ storage on the function call stack:

```
template <typename T>
shared_ptr<node_t<T>> reverse_linked_list(const shared_ptr<node_t<T>> &head) {
  if (!head || !head->next) {
    return head;
  }

  shared_ptr<node_t<T>> new_head = reverse_linked_list(head->next);
  head->next->next = head;
  head->next = nullptr;
  return new_head;
}
```

Iterative implementation:

```
template <typename T>
shared_ptr<node_t<T>> reverse_linked_list(const shared_ptr<node_t<T>> &head) {
  shared_ptr<node_t<T>> prev = nullptr, curr = head;
  while (curr) {
    shared_ptr<node_t<T>> temp = curr->next;
    curr->next = prev;
    prev = curr;
    curr = temp;
```

```
  }
  return prev;
}
```

Problem 7.10, *pg. 66*: *Write a function that determines whether a sequence represented by a singly linked list L is a palindrome. Assume L can be changed and does not have to be restored it to its original state.*

Solution 7.10: Checking if two lists represent the same sequence is straightforward. Therefore one way to check if a linked list is a palindrome is to reverse the second half of the list and compare it with the first half. The middle element can be determined by using a slow pointer and a fast pointer technique (Solution 7.2 on Page 208), and reversing a singly linked list can be performed using Solution 7.9 on the previous page.

This approach changes the list passed in, but the reversed sublist can be reversed again to restore the original list.

```
template <typename T>
bool is_linked_list_a_palindrome(shared_ptr<node_t<T>> L) {
  // Find the middle point of L
  shared_ptr<node_t<T>> slow = L, fast = L;
  while (fast) {
    fast = fast->next;
    if (fast) {
      fast = fast->next, slow = slow->next;
    }
  }

  // Compare the first half and reversed second half lists
  shared_ptr<node_t<T>> reverse = reverse_linked_list<T>(slow);
  while (reverse && L) {
    if (reverse->data != L->data) {
      return false;
    }
    reverse = reverse->next, L = L->next;
  }
  return true;
}
```

Variant 7.10.1: Solve the same problem when the list is doubly linked and you have pointers to the head and the tail.

Problem 7.11, *pg. 66*: *Write a function that takes a singly linked list L, and reorders the elements of L to form a new list representing zip(L). Your function should use $O(1)$ additional storage, as illustrated in Figure 4.1 on Page 25. The only field you can change in a node is* `next`.

Solution 7.11: The problem can be solved in a straightforward manner—find the middle of the list, reverse the second half, and then interleave the first and second

halves. The middle element can be determined by using a slow pointer and a fast pointer (Solution 7.2 on Page 208), and reversing a singly linked list can be done using Solution 7.9 on Page 215. Interleaving is performed by walking the two lists and updating next field from the first list to the corresponding element in the second list, and vice versa.

Though this algorithm is conceptually simple, corner cases abound: the empty list, lists of length 1, and even/odd lengths lists.

```
template <typename T>
void connect_a_next_to_b_advance_a(shared_ptr<node_t<T>> &a,
                                   const shared_ptr<node_t<T>> &b) {
  shared_ptr<node_t<T>> temp = a->next;
  a->next = b;
  a = temp;
}

template <typename T>
shared_ptr<node_t<T>> zipping_linked_list(const shared_ptr<node_t<T>> &L) {
  shared_ptr<node_t<T>> slow = L, fast = L, pre_slow = nullptr;

  // Find the middle point of L
  while (fast) {
    fast = fast->next;
    if (fast) {
      pre_slow = slow;
      fast = fast->next, slow = slow->next;
    }
  }

  if (!pre_slow) {
    return L;  // only contains one node in the list
  }
  pre_slow->next = nullptr;  // split the list into two lists
  shared_ptr<node_t<T>> reverse = reverse_linked_list<T>(slow), curr = L;

  // Zipping the list
  while (curr && reverse) {
    // connect curr->next to reverse, and advance curr
    connect_a_next_to_b_advance_a(curr, reverse);
    if (curr) {
      // connect reverse->next to curr, and advance reverse
      connect_a_next_to_b_advance_a(reverse, curr);
    }
  }
  return L;
}
```

Problem 7.12, *pg. 66*: *Implement a function which takes as input a pointer to the head of a postings list L, and returns a copy of the postings list. Your function should take $O(n)$ time, where n is the length of the postings list and should use $O(1)$ storage beyond that required for the n nodes in the copy. You can modify the original list, but must restore it to its initial state before returning.*

Solution 7.12: We do the copy in following three stages:

(1.) First we copy a node c_x per node x in the original list, and when we do the allocation, we set c_x's next pointer to x's next pointer, then update x's next pointer to c_x. (Note that this does not preclude us from traversing the nodes of the original list.)

(2.) Then we update the jump field for each copied node c_x; specifically, if y is x's jump field, we set c_x's jump field to c_y, which is the copied node of y. (We can do this by traversing the nodes in the original list; note that c_y is just y's next field.)

(3.) Now we set the next field for each x to its original value (which we get from c_x's next field), and the next field for each c_x to $c_{n(x)}$, where $n(x)$ is x's original next node.

These three stages are illustrated in Figures 21.5(b) to 21.5(d) on the current page.

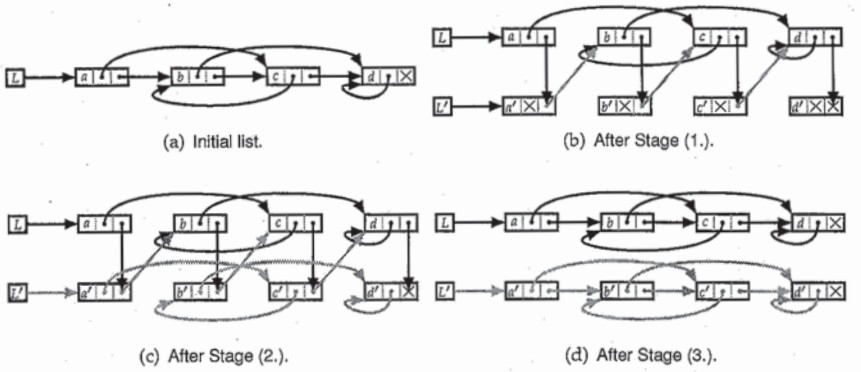


Figure 21.5: Duplicating a postings list.

Code implementing the copy is given below:

```
template <typename T>
shared_ptr<node_t<T>> copy_postings_list(const shared_ptr<node_t<T>> &L) {
  // Return empty list if L is nullptr
  if (!L) {
    return nullptr;
  }

  // 1st stage: copy the nodes from L
  shared_ptr<node_t<T>> p = L;
  while (p) {
    auto temp =
      shared_ptr<node_t<T>>(new node_t<T>{p->data, p->next, nullptr});
    p->next = temp;
    p = temp->next;
  }

  // 2nd stage: update the jump field
```

```
  p = L;
  while (p) {
    if (p->jump) {
      p->next->jump = p->jump->next;
    }
    p = p->next->next;
  }

  // 3rd stage: restore the next field
  p = L;
  shared_ptr<node_t<T>> copied = p->next;
  while (p->next) {
    shared_ptr<node_t<T>> temp = p->next;
    p->next = temp->next;
    p = temp;
  }
  return copied;
}
```

Problem 8.1, ***pg. 67****: Design a stack that supports a* **max** *operation, which returns the maximum value stored in the stack, and throws an exception if the stack is empty. Assume elements are comparable. All operations must be $O(1)$ time. You can use $O(n)$ additional space, beyond what is required for the elements themselves.*

Solution 8.1: A conceptually straightforward approach to tracking the maximum is store pairs in a stack. The first component is the key being pushed; the second is the largest value in the stack after the push is completed. When we push a value, the maximum value stored at or below any of the entries below the entry just pushed does not change. The pushed entry's maximum value is simply the larger of the value just pushed and the maximum prior to the push, which can be determined by inspecting the maximum field of the element below. Since popping does not change the values below, there is nothing special to be done for pop. Of course appropriate checks have to be made to ensure the stack is not empty.

This approach has $O(1)$ time complexity for the the specified methods. The additional space complexity is $\Theta(n)$, regardless of the stored keys.

```
template <typename T>
class Stack {
  private:
    stack<pair<T, T>> s;

  public:
    const bool empty(void) const {
      return s.empty();
    }

    const T &max(void) const {
      if (empty() == false) {
        return s.top().second;
      }
      throw length_error("empty stack");
```

```
    }

    T pop(void) {
      if (empty() == false) {
        T ret = s.top().first;
        s.pop();
        return ret;
      }
      throw length_error("empty stack");
    }

    void push(const T &x) {
      s.emplace(x, std::max(x, empty() ? x : s.top().second));
    }
};
```

Heuristically, the additional space required can be reduced by maintaining two stacks, the primary stack, which holds the keys being pushed, and an auxiliary stack, whose operation we now describe.

The top of the auxiliary stack holds a pair. The first component of the pair is the maximum key in the primary stack. The second component is the number of times that key appears in the primary stack.

Let m be the maximum key currently in the primary stack. There are three cases to consider when a key k is pushed.

1. k is smaller than m. The auxiliary stack is not updated.
2. k is equal to m. We increment the second component of the pair stored at the top of the auxiliary stack.
3. k is greater than m. The pair $(k, 1)$ is pushed onto the auxiliary stack.

There are two cases to consider when the primary stack is popped. Let k be the popped key.

1. k is less than m. The auxiliary stack is not updated.
2. k is equal to m. We decrement the second component of the top of the auxiliary stack. If its value becomes 0, we pop the auxiliary stack.

These operations are illustrated in Figure 21.6 on Page 222.

```
template <typename T>
class Stack {
  private:
    stack<T> s;
    stack<pair<T, int>> aux;

  public:
    const bool empty(void) const {
      return s.empty();
    }

    const T &max(void) const {
      if (empty() == false) {
        return aux.top().first;
      }
      throw length_error("empty stack");
```

```
    }

    T pop(void) {
      if (empty() == false) {
        T ret = s.top();
        s.pop();
        if (ret == aux.top().first) {
          --aux.top().second;
          if (aux.top().second == 0) {
            aux.pop();
          }
        }
        return ret;
      }
      throw length_error("empty stack");
    }

    void push(const T &x) {
      s.emplace(x);
      if (aux.empty() == false) {
        if (x == aux.top().first) {
          ++aux.top().second;
        } else if (x > aux.top().first) {
          aux.emplace(x, 1);
        }
      } else {
        aux.emplace(x, 1);
      }
    }
};
```

The worst-case additional space complexity is $\Theta(n)$, 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 $O(1)$.

Problem 8.2, ***pg. 68***: *Write a function that takes an arithmetical expression in RPN and returns the number that the expression evaluates to.*

Solution 8.2: Conceptually, the algorithm for evaluating an RPN expression iterates through the string from left-to-right. It tokenizes the input into numbers and operators. Numbers are pushed onto a stack. When an operator is read, if it takes k arguments, and there are fewer than k numbers on the stack, a parse error exception is thrown. Otherwise, the expression is evaluated by popping the top k elements of the stack and the operator; the result is pushed back on the stack. When no tokens are left, the only one value left on the stack is the result; otherwise a parse error is declared.

```
int eval(const string &s) {
  stack<int> eval_stack;
  stringstream ss(s);
```

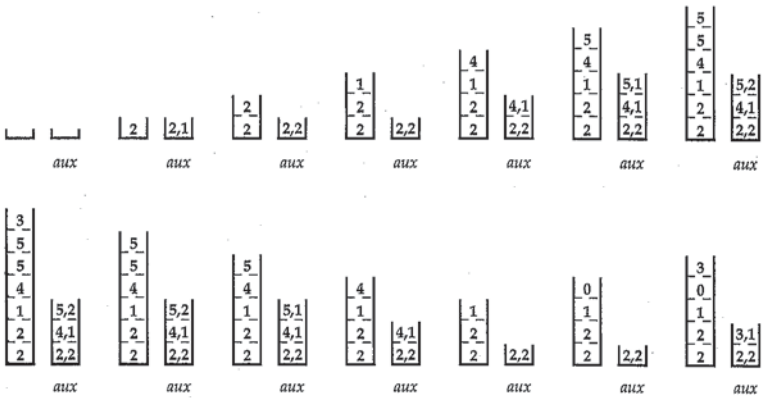


Figure 21.6: The primary and auxiliary stacks for the following operations: `push(2)`, `push(2)`, `push(1)`, `push(4)`, `push(5)`, `push(5)`, `push(3)`, `pop()`, `pop()`, `pop()`, `pop()`, `push(0)`, `push(3)`. Both stacks are initially empty, and their progression is shown from left-to-right, then top-to-bottom. 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*.

```
4    string symbol;
5
6    while (getline(ss, symbol, ',')) {
7      if (symbol == "+" || symbol == "-" || symbol == "*" || symbol == "/") {
8        int y = eval_stack.top();
9        eval_stack.pop();
10       int x = eval_stack.top();
11       eval_stack.pop();
12       switch (symbol.front()) {
13         case '+':
14           eval_stack.emplace(x + y);
15           break;
16         case '-':
17           eval_stack.emplace(x - y);
18           break;
19         case '*':
20           eval_stack.emplace(x * y);
21           break;
22         case '/':
23           eval_stack.emplace(x / y);
24           break;
25       }
26     } else {  // number
27       eval_stack.emplace(stoi(symbol));
28     }
29   }
30   return eval_stack.top();
31 }
```

ϵ-Variant 8.2.1: Solve the same problem for expressions in Polish notation, i.e., when

$A, B, \circ$ is replaced by $\circ, A, B$ in Rule (2.) on Page 68.

Problem 8.3, ***pg. 68***: *Given a BST node n, print all the keys at n and its descendants. The nodes should be printed in sorted order, and you cannot use recursion. For example, for Node I in the binary search tree in Figure 14.1 on Page 105 you should print the sequence* $\langle 23, 29, 31, 37, 41, 43, 47, 53 \rangle$.

Solution 8.3: The recursive solution is trivial—first print the left subtree, then print the root, and finally print the right subtree. This algorithm can be converted into a iterative algorithm by using an explicit stack. Several implementations are possible; the one below is noteworthy in that it pushes the current node, and not its right child, and it does not use a visited field.

```
template <typename T>
void print_BST_in_sorted_order(const shared_ptr<BinarySearchTree<T>> &n) {
  stack<shared_ptr<BinarySearchTree<T>>> s;
  shared_ptr<BinarySearchTree<T>> curr = n;

  while (!s.empty() || curr) {
    if (curr) {
      s.push(curr);
      curr = curr->left;
    } else {
      curr = s.top();
      s.pop();
      cout << curr->data << endl;
      curr = curr->right;
    }
  }
}
```

Problem 8.4, ***pg. 68***: *Write recursive and iterative routines that take a postings list, and computes the jump-first order. Assume each node has an* `order` *field, which is an integer that is initialized to* -1 *for each node.*

Solution 8.4: Recursion is natural—if the current node is unvisited, update the current node's order, visit the `jump` node, then visit the next node. The iterative solution mimics the recursive algorithm using a stack to push nodes that need to be visited. Because of a stack's last-in, first-out semantics, the `next` node is pushed first, since it is to be visited after the `jump` node. Recursive implementation:

```
template <typename T>
void search_postings_list_helper(const shared_ptr<node_t<T>> &L,
                                 int &order) {
  if (L && L->order == -1) {
    L->order = order++;
    search_postings_list_helper<T>(L->jump, order);
    search_postings_list_helper<T>(L->next, order);
  }
}

```

```
template <typename T>
void search_postings_list(const shared_ptr<node_t<T>> &L) {
  int order = 0;
  search_postings_list_helper<T>(L, order);
}
```

Iterative implementation:

```
template <typename T>
void search_postings_list(const shared_ptr<node_t<T>> &L) {
  stack<shared_ptr<node_t<T>>> s;
  int order = 0;
  s.emplace(L);
  while (!s.empty()) {
    shared_ptr<node_t<T>> curr = s.top();
    s.pop();
    if (curr && curr->order == -1) {
      curr->order = order++;
      s.emplace(curr->next);
      s.emplace(curr->jump);
    }
  }
}
```

Problem 8.5, ***pg. 69***: *Exactly n rings on P1 need to be transferred to P2, possibly using P3 as an intermediate, subject to the stacking constraint. Write a function that prints a sequence of operations that transfers all the rings from P1 to P2.*

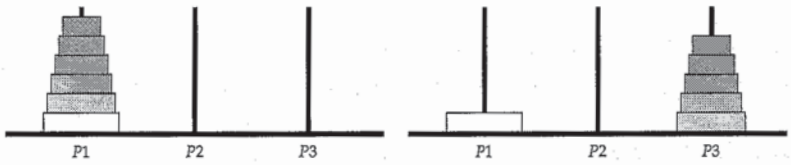
Solution 8.5: Number the n rings from 1 to n. Transfer these n rings from $P1$ to $P2$ as follows.

1. Recursively transfer $n-1$ rings from $P1$ to $P3$ using $P2$.
2. Move the ring numbered $n-1$ from $P1$ to $P2$.
3. Recursively transfer the $n-1$ rings on $P3$ to $P2$, using $P1$.

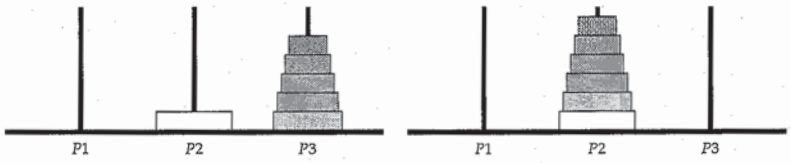
This is illustrated in Figure 21.7 on the next page. Code implementing this idea is given below.

```
void transfer(const int &n, array<stack<int>, 3> &pegs,
              const int &from, const int &to, const int &use) {
  if (n > 0) {
    transfer(n - 1, pegs, from, use, to);
    pegs[to].push(pegs[from].top());
    pegs[from].pop();
    cout << "Move from peg " << from << " to peg " << to << endl;
    transfer(n - 1, pegs, use, to, from);
  }
}

void move_tower_hanoi(const int &n) {
  array<stack<int>, 3> pegs;
  // Initialize pegs
  for (int i = n; i >= 1; --i) {
    pegs[0].push(i);
```



(a) Move all but the lowest disc from $P1$ to $P3$. (b) Move the lowest disc from $P1$ to $P2$.



(c) Move $P3$ to $P2$ using $P1$. (d) Solved!

Figure 21.7: A recursive solution to the Towers of Hanoi for $n = 6$.

```
17    }
18
19    transfer(n, pegs, 0, 1, 2);
20  }
```

ϵ-Variant 8.5.1: Find the minimum number of operations subject to the constraint that each operation must involve $P3$.

ϵ-Variant 8.5.2: Find the minimum number of operations subject to the constraint that each transfer must be from $P1$ to $P2$, $P2$ to $P3$, or $P3$ to $P1$.

ϵ-Variant 8.5.3: Find the minimum number of operations subject to the constraint that a ring can never be transferred directly from $P1$ to $P2$ (transfers from $P2$ to $P1$ are allowed).

ϵ-Variant 8.5.4: Find the minimum number of operations when the stacking constraint is relaxed to the following—the largest ring on a peg must be the lowest ring on the peg. (The remaining rings on the peg can be in any order, e.g., it is fine to have the second-largest ring above the third-largest ring.)

Variant 8.5.5: Find the minimum number of operations if you have a fourth peg, $P4$.

Problem 8.6, *pg. 69*: *Design an algorithm that processes buildings as they are presented to it and tracks the buildings that have a view of the sunset. The number of buildings is not known in advance. Buildings are given in east-to-west order and are specified by their heights. The amount of memory your algorithm uses should depend solely on the number of buildings that have a view; in particular it should not depend on the number of buildings*

processed.

Solution 8.6: We use a stack to record buildings that have a view. Each time a building b is processed, if it is taller than the building at the top of the stack, we pop the stack until the top of the stack is taller than b—all the buildings thus removed lie to the east of a taller building.

Although some individual steps may require many pops, each building is pushed and popped at most once. Therefore the run time to process n buildings is $O(n)$, and the stack always holds precisely the buildings which currently have a view.

```
template <typename T>
vector<pair<int, T>> examine_buildings_with_sunset(istringstream &sin) {
  int idx = 0;  // building's index
  T height;
  // Stores (building_idx, building_height) pair with sunset views
  vector<pair<int, T>> buildings_with_sunset;
  while (sin >> height) {
    while (buildings_with_sunset.empty() == false &&
           height >= buildings_with_sunset.back().second) {
      buildings_with_sunset.pop_back();
    }
    buildings_with_sunset.emplace_back(idx++, height);
  }

  // Returns buildings with its index and height.
  return buildings_with_sunset;
}
```

ϵ-Variant 8.6.1: Solve the problem subject to the same constraints when buildings are presented in west-to-east order.

Problem 8.7, *pg. 69*: *Design an algorithm to sort a stack S of numbers in descending order. The only operations allowed are* `push`, `pop`, `top` *(which returns the top of the stack without a* `pop`*), and* `empty`. *You cannot explicitly allocate memory outside of a few words.*

Solution 8.7: We use recursion—pop the stack and store the result in e, sort the popped stack, then insert the popped element in the right place. The insertion is also done using recursion—if e is smaller than `top`, then push e and return, else do a pop and store the result in f, insert e in the popped stack, then push f. For both the sort and the insert functions the empty stack is the base case. This implementation uses $\Theta(n)$ storage on the function call stack:

```
template <typename T>
void insert(stack<T> &S, const T &e) {
  if (S.empty() || S.top() <= e) {
    S.push(e);
  } else {
    T f = S.top();
    S.pop();
    insert(S, e);
    S.push(f);
```

```
  }
}

template <typename T>
void sort(stack<T> &S) {
  if (!S.empty()) {
    T e = S.top();
    S.pop();
    sort(S);
    insert(S, e);
  }
}
```

Problem 8.8, *pg. 69*: *Write a function which takes a path name, and returns the shortest equivalent path name. Assume individual directories and files have names that use only alphanumeric characters. Subdirectory names may be combined using forward slashes (/), the current directory (.), and parent directory (..). The formal grammar is specified as follows:*

$$\begin{aligned} \texttt{name} &= [A-Za-z0-9]+ \\ \texttt{spdir} &= \texttt{.} \mid \texttt{..} \\ \texttt{pathname} &= \texttt{name} \mid \texttt{spdir} \mid [\texttt{spdir} \mid \texttt{name} \mid \texttt{pathname}]\texttt{?/+}\,\texttt{pathname?} \end{aligned}$$

Here + denotes one or more repetitions or the preceding token, and ? denotes 0 or 1 occurrences of the preceding token. You should throw an exception on invalid path names.

Solution 8.8: The solution uses a stack which will hold the path. The candidate string is parsed from left to right, splitting on /. A leading / is pushed on the stack—this must be an absolute path name. Consequent names are pushed on the stack. Any .. causes a pop of a nonempty stack; if the stack is empty, .. is pushed onto the stack. Any . is skipped.

The error conditions are trying to pop a stack which begins with /, and a substring which is not a name, the empty string, ., or .., separated by /.

The final state of the stack directly corresponds to the shortest equivalent directory path. The argument is based on representing the directory hierarchy as a tree rooted at the root. The directory we compute is a shortest path in the tree. If the bottom of the stack is /, the path is absolute, otherwise it is relative.

```
string normalized_path_names(const string &path) {
  vector<string> s;  // Use vector as a stack
  // Special case: starts with "/", which is an absolute path
  if (path.front() == '/') {
    s.emplace_back("/");
  }

  stringstream ss(path);
  string token;
  while (getline(ss, token, '/')) {
    if (token == "..") {
```

```
      if (s.empty() || s.back() == "..") {
        s.emplace_back(token);
      } else {
        if (s.back() == "/") {
          throw invalid_argument("Path error");
        }
        s.pop_back();
      }
    } else if (token != "." && token != "") {  // name
      for (const char &c : token) {
        if (c != '.' && isalnum(c) == false) {
          throw invalid_argument("Invalid directory name");
        }
      }
      s.emplace_back(token);
    }
  }

  string normalized_path("");
  if (s.empty() == false) {
    auto it = s.cbegin();
    normalized_path += *it++;
    while (it != s.cend()) {
      if (*(it - 1) != "/") {  // previous one is not an absolute path
        normalized_path += "/";
      }
      normalized_path += *it++;
    }
  }
  return normalized_path;
}
```

Problem 8.9, *pg. 70*: *Given the root node r of a binary tree, print all the keys and levels at r and its descendants. The nodes should be printed in order of their level. You cannot use recursion. You may use a single queue, and constant additional storage. For example, you should print the sequence* $\langle 314, 6, 6, 271, 561, 2, 271, 28, 0, 3, 1, 28, 17, 401, 257, 641 \rangle$ *for the binary tree in Figure 9.1 on Page 73.*

Solution 8.9: We maintain a queue of nodes to process. Specifically the queue contains nodes at level l followed by nodes at level $l + 1$. After all nodes from level l are processed, the head of the queue is a node at level $l + 1$; processing this node introduces nodes from level $l + 2$ to the end of the queue. We use a count variable that records the number of nodes at the level of the head of the queue that remain to be processed. When all nodes at level l are processed, the queue consists of exactly the set of nodes at level $l + 1$, and count is updated to the size of the queue.

```
template <typename T>
void print_binary_tree_level_order(const shared_ptr<BinaryTree<T>> &n) {
  // Prevent empty tree
  if (!n) {
    return;
  }
```

```
7
8     queue<shared_ptr<BinaryTree<T>>> q;
9     q.emplace(n);
10    while (!q.empty()) {
11      cout << q.front()->data << ' ';
12      if (q.front()->left) {
13        q.emplace(q.front()->left);
14      }
15      if (q.front()->right) {
16        q.emplace(q.front()->right);
17      }
18      q.pop();
19    }
20  }
```

Problem 8.10, ***pg. 71****: Implement a queue API using an array for storing elements. Your API should include a constructor function, which takes as argument the capacity of the queue,* `enqueue` *and* `dequeue` *functions, a* `size` *function, which returns the number of elements stored, and implement dynamic resizing.*

Solution 8.10: We use an array of length n to store up to n elements. We resize the array by a factor of 2 each time we run out of space. The queue has a `head` field that indexes the least recently inserted element, and a `tail` field, which is the index that the next inserted element will be written to. We record the number of elements in the queue with a `count` variable. Initially, `head` and `tail` are 0. When `count` = n and a `enqueue` is attempted we resize. When `count` = 0 and a `dequeue` is attempted we throw an exception.

```
1  template <typename T>
2  class Queue {
3    private:
4      size_t head, tail, count;
5      vector<T> data;
6
7    public:
8      Queue(const size_t &cap = 8) : head(0), tail(0), count(0), data({cap}) {}
9
10     void enqueue(const T &x) {
11       // Dynamically resize due to data.size() limit
12       if (count == data.size()) {
13         data.resize(data.size() << 1);
14       }
15       // Perform enqueue
16       data[tail] = x;
17       tail = (tail + 1) % data.size(), ++count;
18     }
19
20     T dequeue(void) {
21       if (count) {
22         --count;
23         T ret = data[head];
24         head = (head + 1) % data.size();
```

```
25          return ret;
26        }
27        throw length_error("empty queue");
28      }
29
30      const size_t &size(void) const {
31        return count;
32      }
33  };
```

Alternative implementations are possible, e.g., we can avoid using `count`, and instead use the difference between `head` and `tail` to determine the number of elements. In such an implementation we cannot store more than $n-1$ elements, since otherwise there is no way to differentiate a full queue from an empty one.

Problem 8.11, ***pg. 71***: *Implement a queue using two unsigned integer-valued variables. Assume that the only elements pushed into the queue are integers in* $[0,9]$. *Your program should work correctly when* 0*s are the only elements in the queue. What is the maximum number of elements that can be stored in the queue for it to operate correctly?*

Solution 8.11: The queue state can be viewed as a sequence of digits, with the newest element corresponding to the rightmost digit. A sequence of digits uniquely represents an integer in base-10. Pushing an element corresponds to multiplying that integer by 10 and adding the new element to the result. Popping an element corresponds to identifying the most significant digit. The index i of the most significant digit d is the number of digits in the number, which is computed using $\log_{10}$; the number encoding the new queue is simply the number encoding the old queue minus $d \times 10^i$. The maximum number of elements we can store is dictated by the size of the integer. For k-bit integers the queue is limited to size $\lfloor \log_{10} 2^k \rfloor$.

```
1  class Queue {
2    private:
3      unsigned val, size;
4
5    public:
6      Queue() : val(0), size(0) {}
7
8      void enqueue(const unsigned &x) {
9        val = val * 10 + x;
10       ++size;
11     }
12
13     unsigned dequeue(void) {
14       if (size) {
15         unsigned ret = 0, d = floor(log10(val));
16         if (d + 1 == size) {
17           ret = val / pow(10.0, d);
18           val -= pow(10.0, d) * ret;
19         }
20         --size;
21         return ret;
22       }
```

```
23          throw length_error("empty queue");
24      }
25  };
```

ϵ-Variant 8.11.1: Implement a queue with a single integer-valued variable by reserving the most significant digit for the size.

Problem 8.12, *pg.71*: *How would you implement a queue given two stacks and $O(1)$ additional storage? Your implementation should be efficient—the time to do a sequence of m combined enqueues and dequeues should be $O(m)$.*

Solution 8.12: Call the two stacks A and B. A straightforward implementation of the queue is to enqueue by pushing the element to be enqueued onto A. The element to be dequeued is then the element at the bottom of A, which can be achieved by first popping all the elements of A and pushing them to B, then popping the top of B (which was the bottom-most element of A), and finally popping the remaining elements from B and pushing them to A.

The primary problem with this approach is that every dequeue takes two pushes and two pops of *each* element. (Enqueue takes $O(1)$ time.)

The statement of the problem has a hint—it says that every sequence of m combined enqueues and dequeues should take $O(m)$ time. If we could implement enqueue and dequeue each on $O(1)$, this bound would be trivially met. However, the bound can also be achieved even if individual enqueues and dequeues have high time complexity, as long as there exist enough fast enqueues and dequeues to compensate.

We can implement enqueues by always pushing onto A. Dequeues are handled as follows.

- If B is empty, e.g., we have not done any dequeues so far or all elements in B have been popped, we transfer the contents of A over to B, using pops on A and pushes onto B. Now the top of B contains the element that was enqueued earliest. We simply pop and return that. We do not transfer back from B to A.
- If B is nonempty, e.g., we had just done a dequeue as above, we do by dequeuing from the top of B.

This approach takes $O(m)$ time for m operations, which can be seen from the fact that each element is pushed no more than twice (first on enqueuing onto A and then onto B) and popped no more than twice (first from A and then on dequeuing from B). This style of complexity analysis is known as amortized analysis.

Another minor observation about the implementation is that it can always hold at least n elements (since A is of size n), and in some cases, it may hold up to $2n - 1$ elements before overflowing, e.g., if we do n enqueues, a dequeue, followed by n enqueues. However, we cannot guarantee supporting more than n elements, e.g., if we do $n + 1$ consecutive enqueues, we will be forced to use stack B, and not be able to access the element enqueued earliest for a dequeue.

```
1  template <typename T>
```

```
class Queue {
  private:
    stack<T> A, B;

  public:
    void enqueue(const T &x) {
      A.emplace(x);
    }

    T dequeue(void) {
      if (B.empty()) {
        while (!A.empty()) {
          B.emplace(A.top());
          A.pop();
        }
      }
      if (B.empty() == false) {
        T ret = B.top();
        B.pop();
        return ret;
      }
      throw length_error("empty queue");
    }
};
```

Problem 8.13, *pg. 72*: *How would you implement a queue so that any series of m combined* enqueue, dequeue, *and* max *operations can be done in $O(m)$ time?*

Solution 8.13: This problem can be solved by a combination of Solutions 8.1 on Page 219 and 8.12 on the preceding page. Build the queue by using two stacks, each of which supports the maximum operation. This queue will be able to achieve enqueue, dequeue, and max in amortized $O(1)$ time.

```
template <typename T>
class Queue {
  private:
    Stack<T> A, B;

  public:
    void enqueue(const T &x) {
      A.push(x);
    }

    T dequeue(void) {
      if (B.empty()) {
        while (A.empty() == false) {
          B.push(A.pop());
        }
      }
      if (B.empty() == false) {
        return B.pop();
      }
      throw length_error("empty queue");
```

```
    }

    const T &max(void) const {
      if (A.empty() == false) {
        return B.empty() ? A.max() : std::max(A.max(), B.max());
      } else {  // A.empty() == true
        if (B.empty() == false) {
          return B.max();
        }
        throw length_error("empty queue");
      }
    }
};
```

The solution above is fairly indirect. A more straightforward approach is based on using a deque. Suppose the queue Q consists of elements $\langle e_0, e_1, \ldots, e_{n-1} \rangle$, where e_0 is the element at the head. Call an element e_i in Q *dominated* if there is another element e_j such that $j > i$ and $e_i < e_j$. A dominated element can never become the maximum element in Q, regardless of the sequence of enqueues and dequeues. This is because e_i will be dequeued before e_j, and $e_j > e_i$. Call e_i a *candidate* if it is not dominated.

We maintain the set of candidates in a deque D. Elements in D are ordered by their position in Q, with the candidate closest to the head of Q appearing first. Observe that each candidate in D is greater than or equal to its successors. Consequently the largest element in Q appears at the head of D.

When Q is dequeued, if the element just dequeued is at the head of D, we pop D from its head, otherwise D remains unchanged. If K is enqueued into Q, we iteratively eject D from its tail till the element at D's tail is greater than or equal to K. Then we inject K onto the tail of D. These operations are illustrated in Figure 21.8 on the next page, and implemented in the following code.

```
template <typename T>
class Queue {
  private:
    queue<T> Q;
    deque<T> D;

  public:
    void enqueue(const T &x) {
      Q.emplace(x);
      while (D.empty() == false && D.back() < x) {
        D.pop_back();
      }
      D.emplace_back(x);
    }

    T dequeue(void) {
      if (Q.empty() == false) {
        T ret = Q.front();
        if (ret == D.front()) {
          D.pop_front();
        }
```

```
22          Q.pop();
23          return ret;
24        }
25        throw length_error("empty queue");
26      }
27
28      const T &max(void) const {
29        if (D.empty() == false) {
30          return D.front();
31        }
32        throw length_error("empty queue");
33      }
34  };
```

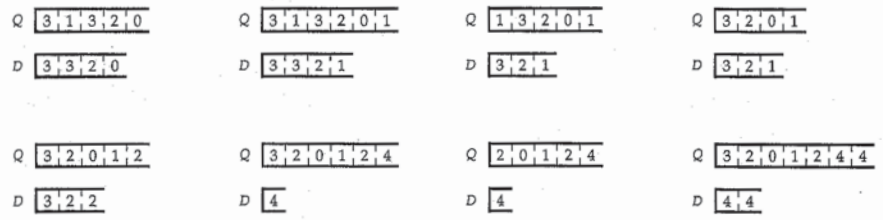


Figure 21.8: The queue with max for the following operations: `enqueue(1)`, `dequeue()`, `dequeue()`, `enqueue(2)`, `enqueue(4)`, `dequeue()`, `enqueue(4)`. The queue initially contains 3, 1, 3, 2, and 0 in that order. The deque D corresponding to queue Q is immediately below Q. The progression is shown from left-to-right, then top-to-bottom. The head of each queue and deque is on the left. Observe how the head of the deque holds the maximum element in the queue.

Each dequeue operation has time $O(1)$ complexity. A single enqueue operation may entail many ejections from D. However, the amortized time complexity of m enqueues and dequeues is $O(m)$, since an element can be added and removed from D no more than once. The max operation is $O(1)$ since it consists of returning the element at the head of D.

Problem 8.14, *pg. 72*: *Let A be an array of length n, and w the window size. Entry $A[i]$ is a pair (t_i, v_i), where t_i is the timestamp and v_i the traffic volume at that time. Assume A is sorted by increasing timestamp. Design an algorithm to compute $v_i = \max\{v_j \mid (t_i - t_j) \leq w, j \leq i\}$, for $0 \leq i \leq n-1$.*

Solution 8.14: The brute-force entails finding the maximum in the subarray consisting of elements whose timestamps line in $[A[i] - w, A[i]]$. It has time complexity $O(n\Delta_w)$, where Δ_w is the maximum number of array elements whose timestamps are contained in any length w interval.

BSTs, which are the subject of Chapter 14, can reduce the complexity to $O(n \log \Delta_w)$—one BST holds volumes in the current window, another BST implements a mapping from timestamps to corresponding nodes in the first BST.

We now describe how to use the queue with maximum data structure developed in Solution 8.13 on Page 232 to achieve an $O(n)$ time complexity, independent of w.

Initialize Q to an empty queue with maximum. Iteratively enqueue (v_i, t_i) in order of increasing i. For each i, iteratively dequeue Q until the difference of the timestamp at Q's head and t_i is less than or equal to w. The sequence of maximum values in the queue for each i is the desired result.

```
class TrafficElement {
  public:
    int time, volume;

    const bool operator<(const TrafficElement &that) const {
      return time < that.time;
    }

    const bool operator==(const TrafficElement &that) const {
      return time == that.time && volume == that.volume;
    }
};

void TrafficVolumes(const vector<TrafficElement> &A, const int &w) {
  Queue<TrafficElement> Q;
  for (int i = 0; i < A.size(); ++i) {
    Q.enqueue(A[i]);
    while (A[i].time - Q.front().time > w) {
      Q.dequeue();
    }
    cout << "Max after inserting " << i << " is " << Q.max().volume << endl;
  }
}
```

Each element is enqueued once. Each element is dequeued at most once. Since the queue with maximum data structure has an $O(1)$ amortized time complexity per operation, the overall time complexity is $O(n)$.. The additional space complexity is $O(\Delta_w)$.

Problem 9.1, *pg. 75*: *Write a function that takes as input the root of a binary tree and returns* `true` *or* `false` *depending on whether the tree is balanced. Use $O(h)$ additional storage, where h is the height of the tree.*

Solution 9.1: Without the $O(h)$ constraint the problem is trivial—we can compute the height for the tree rooted at each node x recursively. The basic computation is $x.height = \max(x.left.height, x.right.height) + 1$, and in each step we check if the difference in heights of the left and right children is greater than one. We can store the heights in a hash table, or in a new field in the nodes. This entails $O(n)$ storage, where n is the number of nodes of the tree.

We will solve this problem using $O(h)$ storage by implementing a `get_height` function which takes a node x as an argument and returns an integer. The function `get_height` returns -2 if the node is unbalanced; otherwise it returns the height of the subtree rooted at that node. The implementation of `get_height` is as follows. If x is `null`, return -1. Otherwise run `get_height` on the left child. If the returned value

l is -2, node x is not balanced; return -2. Call `get_height` on x's right child; let the returned value be r. If r is -2 or $|l - r| > 1$ return -2, otherwise return $\max(l, r) + 1$.

The function `get_height` implements a postorder walk with some calls being eliminated because of early detection of unbalance. 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.

```
template <typename T>
int get_height(const shared_ptr<BinaryTree<T>> &n) {
  if (!n) {
    return -1;  // base case
  }

  int l_height = get_height(n->left);
  if (l_height == -2) {
    return -2;  // left subtree is not balanced
  }
  int r_height = get_height(n->right);
  if (r_height == -2) {
    return -2;  // right subtree is not balanced
  }

  if (abs(l_height - r_height) > 1) {
    return -2;  // current node n is not balanced
  }
  return max(l_height, r_height) + 1;  // return the height
}

template <typename T>
bool is_balanced_binary_tree(const shared_ptr<BinaryTree<T>> &n) {
  return get_height(n) != -2;
}
```

We can improve the space complexity if we know the number of nodes n in the tree in advance. Specifically, the space complexity can be improved to $O(\log n)$ by keeping a global variable that records the maximum height m_s of the stack. Donald Knuth (*"The Art of Computer Programming, Volume 3: Sorting and Searching"*, Page 460) proves that the height of a balanced tree on n nodes is no more than $h_n = 1.4405 \lg(\frac{n}{2} + 3) - 0.3277$. The stack height is a lower bound on the height of the tree, and therefore if the stack height ever exceeds h_n, we return -2.

Variant 9.1.1: Write a function that returns the size of the largest subtree that is complete.

Problem 9.2, *pg. 75*: *Design an algorithm that takes as input a binary tree and positive integer k, and returns a node u in the binary tree such that u is not k-balanced, but all of u's descendants are k-balanced. If no such node exists, return* `null`. *For example, when applied to the binary tree in Figure 9.1 on Page 73, your algorithm should return Node J if k = 3.*

Solution 9.2: It is straightforward to compute the number of nodes in each subtree of a binary tree by a postorder traversal: for each node u, count the number of nodes in its left and the right subtrees, and add one to the sum of those counts to get the count for u.

We can extend this computation by keeping a global pointer-valued variable g that is used to record the result. Initially, g is `null`. We do the postorder traversal to compute the number of nodes in subtrees as before, as long as g is not `null`. The first time our traversal finds a node which is not k-balanced, we set g to r. In the traversal, if g is not `null`, we return. The global g holds the final result.

Since it is poor programming practice to use global variables, our implementation below uses a pointer-integer pair for the return value. The pointer plays the role of g as described above. Since each node is processed only after its descendants, we are guaranteed that the result is set correctly. The time complexity is $O(n)$, where n is the number of nodes.

```
template <typename T>
pair<shared_ptr<BinaryTree<T>>, int> find_non_k_balanced_node_helper(
    const shared_ptr<BinaryTree<T>> &n, const int &k) {
  // Empty tree
  if (!n) {
    return {nullptr, 0};
  }

  // Early return if left subtree is not k-balanced
  auto L = find_non_k_balanced_node_helper<T>(n->left, k);
  if (L.first) {
    return L;
  }
  // Early return if right subtree is not k-balanced
  auto R = find_non_k_balanced_node_helper<T>(n->right, k);
  if (R.first) {
    return R;
  }

  int node_num = L.second + R.second + 1;  // #nodes in n
  if (abs(L.second - R.second) > k) {
    return {n, node_num};
  }
  return {nullptr, node_num};
}

template <typename T>
shared_ptr<BinaryTree<T>> find_non_k_balanced_node(
    const shared_ptr<BinaryTree<T>> &n, const int &k) {
  return find_non_k_balanced_node_helper<T>(n, k).first;
}
```

Problem 9.3, ***pg.76****: Write a function that takes as input the root of a binary tree and returns* `true` *or* `false` *depending on whether the tree is symmetric.*

Solution 9.3: We present a recursive algorithm that follows directly from the definition of symmetry.

```
template <typename T>
bool is_symmetric_helper(const shared_ptr<BinaryTree<T>> &l,
                         const shared_ptr<BinaryTree<T>> &r) {
  if (!l && !r) {
    return true;
  } else if (l && r) {
    return l->data == r->data && is_symmetric_helper<T>(l->left, r->right) &&
           is_symmetric_helper<T>(l->right, r->left);
  } else {  // (l && !r) || (!l && r)
    return false;
  }
}

template <typename T>
bool is_symmetric(const shared_ptr<BinaryTree<T>> &n) {
  return (!n || is_symmetric_helper<T>(n->left, n->right));
}
```

Problem 9.4, *pg. 76*: *For a certain application, processes need to lock nodes in a binary tree. Implement a library for locking nodes in a binary tree, subject to the constraint that a node cannot be locked if any of its descendants or ancestors are locked. Specifically, write functions* `isLock()`, `lock()`, *and* `unLock()`, *with time complexities* $O(1)$, $O(h)$, *and* $O(h)$. *Here* h *is the height of the binary tree. Assume that each node has a parent field.*

Solution 9.4: Each node has a bool-valued `locked` field, indicating whether it holds a lock—this makes the `isLock()` function trivial. In addition, we use an integer-valued `numChildrenLocks` field for each node n which tracks the number of children in the subtrees rooted at n that are locked. The `lock()` function proceeds only if the number of locked children is 0; if so it checks the state of all the ancestors leading up to the root. If the node is lockable, the function increments the `numChildrenLocks` fields for each of the ancestors. The `unLock()` function simply sets `locked` to `false`, and decreases the `numChildrenLocks` field for each ancestor all the way to the root. The time complexity for `lock()` and `unLock()` is bounded by the distance of the node from the root, i.e., the height of the tree.

```
template <typename T>
class BinaryTree {
  private:
    bool locked;
    int numChildrenLocks;

  public:
    shared_ptr<BinaryTree<T>> left, right, parent;

    const bool &isLock(void) const {
      return locked;
    }

```

```
    void lock(void) {
      if (numChildrenLocks == 0 && locked == false) {
        // Make sure all parents do not lock
        shared_ptr<BinaryTree<T>> n = parent;
        while (n) {
          if (n->locked == true) {
            return;
          }
          n = n->parent;
        }

        // Lock itself and update its parents
        locked = true;
        n = parent;
        while (n) {
          ++n->numChildrenLocks;
          n = n->parent;
        }
      }
    }

    void unLock(void) {
      if (locked) {
        // Unlock itself and update its parents
        locked = false;
        shared_ptr<BinaryTree<T>> n = parent;
        while (n) {
          --n->numChildrenLocks;
          n = n->parent;
        }
      }
    }
};
```

Problem 9.5, *pg. 76*: *Let T be the root of a binary tree in which nodes have an explicit parent field. Design an iterative algorithm that enumerates the nodes inorder and uses $O(1)$ additional space. Your algorithm cannot modify the tree.*

Solution 9.5: The standard idiom for an inorder walk is visit-left, visit-root, visit-right. Accessing the left child is straightforward. Returning from a left child l to its parent entails examining l's parent field; returning from a right child r to its parent is similar.

To make this scheme work, we need to know when we take a parent pointer to node `r` if the child we completed visiting was `r`'s left child (in which case we need to visit `r` and then `r`'s right child) or a right child (in which case we have completed visiting `r`). We achieve this by storing the child in a `prev` variable before we move to the parent, `r`. We then compare `prev` with `r`'s left child and the right child.

```
template <typename T>
void inorder_traversal(const shared_ptr<BinaryTree<T>> &r) {
  // Empty tree
  if (!r) {
```

```
    return;
  }

  shared_ptr<BinaryTree<T>> prev = nullptr, curr = r, next;
  while (curr) {
    if (!prev || prev->left == curr || prev->right == curr) {
      if (curr->left) {
        next = curr->left;
      } else {
        cout << curr->data << endl;
        next = (curr->right ? curr->right : curr->parent);
      }
    } else if (curr->left == prev) {
      cout << curr->data << endl;
      next = (curr->right ? curr->right : curr->parent);
    } else {  // curr->right == prev
      next = curr->parent;
    }

    prev = curr;
    curr = next;
  }
}
```

ϵ-Variant 9.5.1: How would you perform preorder and postorder walks iteratively using $O(1)$ additional space? Your algorithm cannot modify the tree. Nodes have an explicit parent field.

Problem 9.6, *pg. 76*: *Design a function that efficiently computes the k-th node appearing in an inorder traversal. Specifically, your function should take as input a binary tree T and an integer k. Each node has a* `size` *field, which is the number of nodes in the subtree rooted at that node. What is the time complexity of your function?*

Solution 9.6: If the left child has $k-1$ children, then the root is the k-th node; if the left child has k or more children, then the k-th node is the k-th node of the left subtree; and if the left child has $l < k-1$ children, the k-th node is the $k-(l+1)$-th node of the right subtree.

```
template <typename T>
shared_ptr<BinaryTree<T>> find_kth_node_binary_tree(
    shared_ptr<BinaryTree<T>> r, int k) {
  while (k && r) {
    int left_size = r->left ? r->left->size : 0;
    if (left_size < k - 1) {
      k -= (left_size + 1);
      r = r->right;
    } else if (left_size == k - 1) {
      return r;
    } else {  // left_size > k - 1
      r = r->left;
    }
  }
```

```
15    throw length_error("no k-th node in binary tree");
16  }
```

Since we descend the tree in each iteration, the time complexity is $O(h)$, where h is the height of the tree.

Problem 9.7, ***pg. 77***: *Given an inorder traversal order, and one of a preorder or a postorder traversal order of a binary tree, write a function to reconstruct the tree.*

Solution 9.7: Suppose we are given the inorder and preorder traversal sequences. The preorder sequence gives us the key of the root node—it is the first node in the sequence. This in turn allows us to split the inorder sequence into an inorder sequence for the left subtree, followed by the root, followed by the right subtree. The left subtree inorder sequence allows us to compute the preorder sequence for the left subtree from the preorder sequence: the nodes in the left subtree appear before all the nodes in the right subtree in the preorder sequence.

For example, if the inorder sequence is $\langle B, A, D, C, E \rangle$ and the preorder sequence is $\langle A, B, C, D, E \rangle$, we know the root has key A and the left subtree consists of the single node with key B. Therefore, in an inorder visit the right subtree nodes appear as $\langle D, C, E \rangle$, and in a preorder visit the right subtree nodes appear as $\langle C, D, E \rangle$. Therefore the root of the right subtree is a node whose key is C and its left and right subtrees are the single nodes with keys D and E, respectively.

If the keys are unique, we can use the above algorithm to uniquely reconstruct a binary tree yielding the given inorder and preorder sequences, but this is not always true if duplicate keys are present. As an extreme example, if all keys are the same, all binary trees on n nodes with that key yield identical sequences.

```
1  template <typename T>
2  shared_ptr<BinaryTree<T>> reconstruct_pre_in_orders_helper(
3      const vector<T> &pre, const int &pre_s, const int &pre_e,
4      const vector<T> &in, const int &in_s, const int &in_e) {
5    if (pre_e > pre_s && in_e > in_s) {
6      auto it = find(in.cbegin() + in_s, in.cbegin() + in_e, pre[pre_s]);
7      int left_tree_size = it - (in.cbegin() + in_s);
8
9      return shared_ptr<BinaryTree<T>>(new BinaryTree<T>{
10       pre[pre_s],
11       // Recursively build the left subtree
12       reconstruct_pre_in_orders_helper<T>(
13         pre, pre_s + 1, pre_s + 1 + left_tree_size,
14         in, in_s, it - in.cbegin()),
15       // Recursively build the right subtree
16       reconstruct_pre_in_orders_helper<T>(
17         pre, pre_s + 1 + left_tree_size, pre_e,
18         in, it - in.cbegin() + 1, in_e)
19       });
20   }
21   return nullptr;
22 }
23
```

```
template <typename T>
shared_ptr<BinaryTree<T>> reconstruct_pre_in_orders(const vector<T> &pre,
                                                    const vector<T> &in) {
  return reconstruct_pre_in_orders_helper(pre, 0, pre.size(),
                                          in, 0, in.size());
}
```

We recover the tree from postorder and inorder traversal sequences similarly:

```
template <typename T>
shared_ptr<BinaryTree<T>> reconstruct_post_in_orders_helper(
    const vector<T> &post, const int &post_s, const int &post_e,
    const vector<T> &in, const int &in_s, const int &in_e) {
  if (post_e > post_s && in_e > in_s) {
    auto it = find(in.cbegin() + in_s, in.cbegin() + in_e, post[post_e - 1]);
    int left_tree_size = it - (in.cbegin() + in_s);

    return shared_ptr<BinaryTree<T>>(new BinaryTree<T>{
      post[post_e - 1],
      // Recursively build the left subtree
      reconstruct_post_in_orders_helper<T>(
        post, post_s, post_s + left_tree_size,
        in, in_s, it - in.cbegin()),
      // Recursively build the right subtree
      reconstruct_post_in_orders_helper<T>(
        post, post_s + left_tree_size, post_e - 1,
        in, it - in.cbegin() + 1, in_e)
      });
  }
  return nullptr;
}

template <typename T>
shared_ptr<BinaryTree<T>> reconstruct_post_in_orders(const vector<T> &post,
                                                     const vector<T> &in) {
  return reconstruct_post_in_orders_helper(post, 0, post.size(),
                                           in, 0, in.size());
}
```

Variant 9.7.1: Let A be an array of n distinct integers. Let the index of the maximum element of A be m. Define the max-tree on A to be the binary tree on the entries of A in which the root contains the maximum element of A, the left child is the max-tree on $A[0 : m-1]$ and the right child is the max-tree on $A[m+1 : n-1]$. Design an $O(n)$ algorithm for building the max-tree of A.

Problem 9.8, *pg. 77*: *Design an $O(n)$ time algorithm for reconstructing a binary tree from a preorder visit sequence that uses* `null` *to mark empty children. How would you modify your reconstruction algorithm if the sequence corresponded to a postorder or inorder walk?*

Solution 9.8: We traverse the sequence from right-to-left. We push nodes and `null`s on to a stack; every time we encountered a non-`null` node x, we pop the stack twice—call the first node popped `l` and the second `r`. Set x's left and right children to `l` and

r, respectively, and push x. When the sequence is exhausted, there will be a single node on the stack, which will be the root.

```
template <typename T>
shared_ptr<BinaryTree<T>> reconstruct_preorder(
  const vector<shared_ptr<T>> &preorder) {
  stack<shared_ptr<BinaryTree<T>>> s;
  for (auto it = preorder.crbegin(); it != preorder.crend(); ++it) {
    if (!(*it)) {
      s.emplace(nullptr);
    } else {  // non-nullptr
      shared_ptr<BinaryTree<T>> l = s.top();
      s.pop();
      shared_ptr<BinaryTree<T>> r = s.top();
      s.pop();
      s.emplace(new BinaryTree<T>{*(*it), l, r});
    }
  }
  return s.top();
}
```

Reconstructing from a postorder traversal is similar—we traverse from the beginning of the sequence, and when popping, the top of the stack is the right child, and the node below it is the left child.

Reconstructing from an inorder traversal is impossible, even with the null markers. This is because *every* binary tree that yields $\langle v_0, v_1, \ldots, v_{n-1} \rangle$ on an inorder walk has a modified sequence of $\langle \texttt{null}, v_0, \texttt{null}, v_1, \texttt{null}, v_2, \ldots, \texttt{null}, v_{n-1}, \texttt{null} \rangle$. An inorder traversal order is not enough to uniquely reconstruct a binary tree, so the inorder sequence with markers will also be insufficient. If all we want is a binary tree that yields the given sequence, we can simply return a completely right-skewed tree, i.e., its root is v_0, left child is empty, and right child is reconstructed recursively from $\langle v_1, v_2, \ldots, v_{n-1} \rangle$.

Problem 9.9, *pg. 78*: *Given a binary tree, write a function which forms a linked list from the leaves of the binary tree. The leaves should appear in left-to-right order. For example, when applied to the binary tree in Figure 9.1 on Page 73, your function should return* $\langle D, E, H, M, N, P \rangle$.

Solution 9.9: We use recursion, passing in the list L of leaves. If the node is a leaf, which we determine by checking if both children are null, we append it to the list L and return. Otherwise, we recurse on the left and right children, which causes the leaves on the left subtree to appear before the leaves on the right subtree. The time complexity is $O(n)$, where n is the number of nodes.

```
template <typename T>
void connect_leaves_helper(const shared_ptr<BinaryTree<T>> &n,
                           list<shared_ptr<BinaryTree<T>>> &L) {
  if (n) {
    if (!n->left && !n->right) {
      L.push_back(n);
    } else {
```

```
      connect_leaves_helper(n->left, L);
      connect_leaves_helper(n->right, L);
    }
  }
}

template <typename T>
list<shared_ptr<BinaryTree<T>>> connect_leaves(
    const shared_ptr<BinaryTree<T>> &n) {
  list<shared_ptr<BinaryTree<T>>> L;
  connect_leaves_helper(n, L);
  return L;
}
```

Problem 9.10, *pg. 78*: *Write a function that prints the nodes on the exterior of a binary tree in anti-clockwise order, i.e., print the nodes on the path from the root to the leftmost leaf in that order, then the leaves from left-to-right, then the nodes from the rightmost leaf up to the root. For example, when applied to the binary tree in Figure 9.1 on Page 73, your function should return* $\langle A, B, C, D, E, H, M, N, P, O, I \rangle$. *(By leftmost (rightmost) leaf, we mean the leaf that appears first (last) in an inorder walk.)*

Solution 9.10: One approach is to print all the nodes leading to the leftmost leaf first, using a recursive search from the root that favors a left child when available, followed by the leaves (which can be performed using the technique in Solution 9.9 on the preceding page) followed by printing all nodes from the rightmost leaf to the root, which is performed using a recursive search from the root that favors the right. The first and last functions print in preorder and postorder to ensure the right ordering of nodes.

Alternately, we can print the root, followed by all the required nodes (leftmost and leaves) from the left subtree followed by all the required nodes (leaves and rightmost) from the right subtree. The left subtree and right subtree are processed by symmetric functions. Details are given below.

```
template <typename T>
void left_boundary_b_tree(const shared_ptr<BinaryTree<T>> &n,
                          const bool &is_boundary) {
  if (n) {
    if (is_boundary || (!n->left && !n->right)) {
      cout << n->data << ' ';
    }
    left_boundary_b_tree(n->left, is_boundary);
    left_boundary_b_tree(n->right, is_boundary && !n->left);
  }
}

template <typename T>
void right_boundary_b_tree(const shared_ptr<BinaryTree<T>> &n,
                           const bool &is_boundary) {
  if (n) {
    right_boundary_b_tree(n->left, is_boundary && !n->right);
    right_boundary_b_tree(n->right, is_boundary);
```

```
    if (is_boundary || (!n->left && !n->right)) {
      cout << n->data << ' ';
    }
  }
}

template <typename T>
void exterior_binary_tree(const shared_ptr<BinaryTree<T>> &root) {
  if (root) {
    cout << root->data << ' ';
    left_boundary_b_tree(root->left, true);
    right_boundary_b_tree(root->right, true);
  }
}
```

Problem 9.11, ***pg. 78***: *Design an efficient algorithm for computing the LCA of nodes a and b in a binary tree in which nodes do not have a parent pointer.*

Solution 9.11: Let a and b be the nodes whose LCA we wish to compute. Observe that if the root is one of a or b, then it is the LCA. Otherwise, let L and R be the trees rooted at the left child and the right child of the root. If both nodes lie in L (or R), their LCA is in L (or R). Otherwise, their LCA is the root itself. This is the basis for the algorithm presented below. Its time complexity is $O(n)$, where n is the number of nodes.

```
template <typename T>
shared_ptr<BinaryTree<T>> LCA(const shared_ptr<BinaryTree<T>> &n,
                              const shared_ptr<BinaryTree<T>> &a,
                              const shared_ptr<BinaryTree<T>> &b) {
  if (!n) {  // empty subtree
    return nullptr;
  } else if (n == a || n == b) {
    return n;
  }

  auto l_res = LCA(n->left, a, b), r_res = LCA(n->right, a, b);
  if (l_res && r_res) {
    return n;  // found a and b in different subtrees
  } else {
    return l_res ? l_res : r_res;
  }
}
```

Problem 9.12, ***pg. 78***: *Given two nodes in a binary tree T, design an algorithm that computes their LCA. Assume that each node has a parent pointer. The tree has n nodes and height h. Your algorithm should run in $O(1)$ space and $O(h)$ time.*

Solution 9.12: Suppose we know the depths d_a and d_b of nodes a and b. Without loss of generality, assume $d_a \geq d_b$. Follow $d_a - d_b$ parent pointers starting from a. Let c be the resulting node. The LCA of a and b is the same as the LCA of b and c. We can

now compute the LCA of c and b by iteratively moving up the tree, from c and from b till we reach a common node l, which is the desired LCA.

The depth of a node can be computed by following its parent pointers until the root is reached. This computation has a time complexity $O(h)$, and space complexity $O(1)$. Therefore the LCA of a and b can be computed in $O(h)$ time and $O(1)$ space.

```
template <typename T>
int get_depth(shared_ptr<BinaryTree<T>> n) {
  int d = 0;
  while (n) {
    ++d, n = n->parent;
  }
  return d;
}

template <typename T>
shared_ptr<BinaryTree<T>> LCA(shared_ptr<BinaryTree<T>> a,
                              shared_ptr<BinaryTree<T>> b) {
  int depth_a = get_depth(a), depth_b = get_depth(b);
  if (depth_b > depth_a) {
    swap(a, b);
  }

  // Advance deeper node first
  int depth_diff = depth_a - depth_b;
  while (depth_diff--) {
    a = a->parent;
  }

  // Both pointers advance until they found a common ancestor
  while (a != b) {
    a = a->parent, b = b->parent;
  }
  return a;
}
```

Problem 9.13, *pg. 78*: *Design an algorithm for computing the LCA of a and b that has time complexity $O(\max(d_a - d_l, d_b - d_l))$. What is the worst-case time and space complexity of your algorithm?*

Solution 9.13: Let the sequences of nodes as we traverse parent pointers from a and b to the root be $\langle a, a_1, a_2, \ldots \rangle$ and $\langle b, b_1, b_2, \ldots \rangle$ respectively. The LCA of a and b is the first node in either sequence that is common to the two sequences. This leads to an algorithm for computing the LCA in $O(\max(d_a - d_l, d_b - d_l))$ time: interleave traversing parent pointers from a and from b, storing visited nodes in a hash table. Each time we visit a node we check to see if it has been visited before. We will revisit a node after exactly $2(\max(d_a, d_b) - d_l)$ pointers have been traversed, yielding the desired time complexity.

Note that we are trading space for time. The algorithm for Solution 9.12 on the previous page used $O(1)$ space and $O(h)$ time, whereas the algorithm presented above

uses $O(\max(d_a, d_b) - d_l)$ space and time. In the worst case, a and b are leaves whose LCA is the root, and we use $O(h)$ space and time.

```
template <typename T>
shared_ptr<BinaryTree<T>> LCA(shared_ptr<BinaryTree<T>> a,
                              shared_ptr<BinaryTree<T>> b) {
  unordered_set<shared_ptr<BinaryTree<T>>> hash;
  while (a || b) {
    if (a) {
      if (hash.emplace(a).second == false) {
        return a;  // adds a failed because a exists in hash
      }
      a = a->parent;
    }
    if (b) {
      if (hash.emplace(b).second == false) {
        return b;  // adds b failed because b exists in hash
      }
      b = b->parent;
    }
  }
  // Throw error if a and b are not in the same tree
  throw invalid_argument("a and b are not in the same tree");
}
```

Problem 9.14, ***pg. 79***: *Given a string s and a set of strings D, find the shortest prefix of s which is not a prefix of any string in D.*

Solution 9.14: A trie is a data structure for storing a set of strings based on positional trees. To be concrete, suppose the strings are over the alphabet {"a", "b", ..., "z"}. Each node has a hash table mapping each character in the alphabet to the corresponding child pointer. Some or all of the children may be `null`. A path of length l starting from the root naturally corresponds to a string of l characters. Each node has a Boolean field indicating whether the string corresponding to the path from the root is a string in the set.

Finding a shortest prefix of s that is not a prefix of any string in the represented set is simply a matter of finding the first node m on the search path from the root that does not have a child corresponding to the next character in s.

```
class Trie {
  private:
    class TrieNode {
      public:
        bool isString;
        unordered_map<char, shared_ptr<TrieNode>> l;
    };

    shared_ptr<TrieNode> root;

  public:
    Trie() : root(shared_ptr<TrieNode>(new TrieNode{false})) {}

```

```
14      bool insert(const string &s) {
15        shared_ptr<TrieNode> p = root;
16        for (const char &c : s) {
17          if (p->l.find(c) == p->l.cend()) {
18            p->l[c] = shared_ptr<TrieNode>(new TrieNode{false});
19          }
20          p = p->l[c];
21        }
22
23        // s already existed in this trie
24        if (p->isString == true) {
25          return false;
26        } else {  // p->isString == false
27          p->isString = true;  // inserts s into this trie
28          return true;
29        }
30      }
31
32      string getShortestUniquePrefix(const string &s) {
33        shared_ptr<TrieNode> p = root;
34        string prefix;
35        for (const char &c : s) {
36          prefix += c;
37          if (p->l.find(c) == p->l.cend()) {
38            return prefix;
39          }
40          p = p->l[c];
41        }
42        return {};
43      }
44  };
45
46  string find_shortest_prefix(const string &s, const unordered_set<string> &D) {
47    // Build a trie according to given dictionary D
48    Trie T;
49    for (const string &word : D) {
50      T.insert(word);
51    }
52    return T.getShortestUniquePrefix(s);
53  }
```

ϵ-Variant 9.14.1: How would you find the shortest string that is not a prefix of any string in D?

Problem 10.1, *pg. 80*: *Design an algorithm that takes a set of files containing stock trade information in sorted order, and writes a single file containing the lines appearing in the individual files sorted in sorted order. The algorithm should use very little RAM, ideally of the order of a few kilobytes.*

Solution 10.1: In the abstract, we are trying to merge k sorted files. One way to do this is to repeatedly pick the smallest element amongst the smallest remaining elements from each file. A min-heap is ideal for maintaining a set of elements when

we repeatedly insert and query for the smallest element (both extract-min and insert take $O(\log k)$ time). Hence we can do the merge in $O(n \log k)$ time, where n is the total number of elements in the input. Here is the code for this:

```
template <typename T>
class Compare {
  public:
    const bool operator()(const pair<T, int> &lhs,
                          const pair<T, int> &rhs) const {
      return lhs.first > rhs.first;
    }
};

template <typename T>
vector<T> merge_arrays(const vector<vector<T>> &S) {
  priority_queue<pair<T, int>, vector<pair<T, int>>, Compare<T>> min_heap;
  vector<int> S_idx(S.size(), 0);

  // Every array in S puts its smallest element in heap
  for (int i = 0; i < S.size(); ++i) {
    if (S[i].size() > 0) {
      min_heap.emplace(S[i][0], i);
      S_idx[i] = 1;
    }
  }

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

Alternately, we could recursively merge the k files, two at a time using the merge step from merge sort.

Problem 10.2, ***pg. 81***: *Design an efficient algorithm for sorting a k-increasing-decreasing array. You are given another array of the same size that the result should be written to, and you can use $O(k)$ additional storage.*

Solution 10.2: The first thing to note is that any array can be decomposed into a sequence of increasing and decreasing subarrays. If k is comparable to n, then the problem is equivalent to the general sorting problem.

If k is substantially smaller than n, we could first reverse the order of the decreasing subarrays. Now we can use the techniques in Solution 10.1 on the facing page to sort the array in time $O(n \log k)$ time with $O(k)$ space.

```
template <typename T>
vector<T> sort_k_increasing_decreasing_array(const vector<T> &A) {
  // Decompose A into a set of sorted arrays
  vector<vector<T>> S;
  bool is_increasing = true;  // the trend we are looking for
  int start_idx = 0;
  for (int i = 1; i < A.size(); ++i) {
    if ((A[i - 1] < A[i] && !is_increasing) ||
        (A[i - 1] >= A[i] && is_increasing)) {
      if (is_increasing) {
        S.emplace_back(A.cbegin() + start_idx, A.cbegin() + i);
      } else {
        S.emplace_back(A.crbegin() + A.size() - i,
                       A.crbegin() + A.size() - start_idx);
      }
      start_idx = i;
      is_increasing = !is_increasing;  // inverse the trend we are looking for
    }
  }
  if (start_idx < A.size()) {
    if (is_increasing) {
      S.emplace_back(A.cbegin() + start_idx, A.cend());
    } else {
      S.emplace_back(A.crbegin(), A.crbegin() + A.size() - start_idx);
    }
  }

  return merge_arrays(S);
}
```

Problem 10.3, *pg. 81*: *How would you implement a stack API using a heap and a queue API using a heap?*

Solution 10.3: The basic idea is to use an integer-valued variable `order` that keeps track of the order in which elements were added.

We mimic a stack S with a max-heap H by storing $y = (\texttt{order}, x)$ in H each time x is pushed in S, and incrementing `order`. Heap entries are compared by order. Popping is simply a matter of extracting the max element.

We mimic a queue analogously, except that we decrement `order` on inserts, thereby favoring the element that was inserted first when we do extract-max. It is straightforward to support queue inserts and deletes. Supporting a back function, which returns the element at the queue tail, is more involved. It can be performed with an additional min-heap.

```
template <typename T>
class Compare {
  public:
    bool operator()(const pair<int, T> &lhs, const pair<int, T> &rhs) const {
      return lhs.first < rhs.first;
    }
};
```

```
8
9  template <typename T>
10 class Stack :  // inherits empty(), pop(), and size() methods
11   public priority_queue<pair<int, T>, vector<pair<int, T>>, Compare<T>> {
12   private:
13     int order;
14     typedef priority_queue<pair<int, T>, vector<pair<int, T>>, Compare<T>> PQ;
15
16   public:
17     Stack() : order(0) {}
18
19     const T &top() const {
20       return PQ::top().second;
21     }
22
23     void push(const T &x) {
24       PQ::emplace(order++, x);
25     }
26 };
27
28 template <typename T>
29 class Queue :  // inherits empty(), pop(), and size() methods
30   public priority_queue<pair<int, T>, vector<pair<int, T>>, Compare<T>> {
31   private:
32     int order;
33     typedef priority_queue<pair<int, T>, vector<pair<int, T>>, Compare<T>> PQ;
34
35   public:
36     Queue() : order(0) {}
37
38     const T &front() const {
39       return PQ::top().second;
40     }
41
42     void push(const T &x) {
43       PQ::emplace(order--, x);
44     }
45 };
```

Problem 10.4, ***pg. 81***: *How would you compute the k stars which are closest to the Earth? You have only a few megabytes of RAM.*

Solution 10.4: If RAM was not a limitation, we could read the data into an array, and apply the selection algorithm from Solution 11.13 on Page 270.

It is not difficult to come up with an algorithm based on processing through the file, selecting all stars within a distance d, and sorting the result. Selecting d appropriately is difficult, and will require multiple passes with different choices of d.

A better approach is to use a max-heap H of k elements. We start by adding the first k stars to H. As we process the stars, each time we encounter a star s that is closer to the Earth than the star m in H that is furthest from the Earth (which is the star at the root of H), we delete m from H, and add s to H.

The heap-based algorithm has $O(n \log k)$ time complexity to find the k closest stars out of n candidates, independent of the order in which stars are processed and their locations. Its space complexity is $O(k)$.

```
class Star {
  public:
    int ID;
    double x, y, z;

    // The distance between this star to the Earth
    const double distance() const {
      return sqrt(x * x + y * y + z * z);
    }

    const bool operator<(const Star &s) const {
      return distance() < s.distance();
    }
};

vector<Star> find_closest_k_stars(istringstream &sin, const int &k) {
  // Use max_heap to find the closest k stars
  priority_queue<Star, vector<Star>> max_heap;
  string line;

  // Record the first k stars
  while (getline(sin, line)) {
    stringstream line_stream(line);
    string buf;
    getline(line_stream, buf, ',');
    int ID = stoi(buf);
    array<double, 3> data;  // stores x, y, and z
    for (int i = 0; i < 3; ++i) {
      getline(line_stream, buf, ',');
      data[i] = stod(buf);
    }
    Star s{ID, data[0], data[1], data[2]};

    if (max_heap.size() == k) {
      // Compare the top of heap with the incoming star
      Star far_star = max_heap.top();
      if (s < far_star) {
        max_heap.pop();
        max_heap.emplace(s);
      }
    } else {
      max_heap.emplace(s);
    }
  }

  // Store the closest k stars
  vector<Star> closest_stars;
  while (!max_heap.empty()) {
    closest_stars.emplace_back(max_heap.top());
    max_heap.pop();
  }
```

```
52    return closest_stars;
53  }
```

Problem 10.5, ***pg. 81***: *Design an $O(n \log k)$ time algorithm that reads a sequence of n elements and for each element, starting from the k-th element, prints the k-th largest element read up to that point. The length of the sequence is not known in advance. Your algorithm cannot use more than $O(k)$ additional storage.*

Solution 10.5: We use a min-heap of size k. When the first k elements have been read in, the root holds the k-th largest element. Each successive element s is compared with the minimum element m in the heap. If s is less than or equal to m, do nothing. Otherwise, we remove m, and add s. The new root (which may or may not be s) is the k-th largest element. For the first k iterations, we simply add elements to the min-heap; the root holds the smallest value.

The time complexity per element processed is dominated by the time to delete the root, i.e., $O(\log k)$. The worst-case input is one in which elements appear in increasing order; the best case input is one in which elements appear in decreasing order.

```
1  template <typename T>
2  void find_k_th_largest_stream(istringstream &sin, const int &k) {
3    priority_queue<T, vector<T>, greater<T>> min_heap;
4    // The first k elements, output the minimum element
5    T x;
6    for (int i = 0; i < k && sin >> x; ++i) {
7      min_heap.emplace(x);
8      cout << min_heap.top() << endl;
9    }
10
11   // After the first k elements, output the k-th largest one
12   while (sin >> x) {
13     if (min_heap.top() < x) {
14       min_heap.pop();
15       min_heap.emplace(x);
16     }
17     cout << min_heap.top() << endl;
18   }
19 }
```

Problem 10.6, ***pg. 82***: *The input consists of a very long sequence of numbers. Each number is at most k positions away from its correctly sorted position. Design an algorithm that outputs the numbers in the correct order and uses $O(k)$ storage, independent of the number of elements processed.*

Solution 10.6: The easiest way of looking at this problem is that we need to store the numbers in memory till all the numbers smaller than this number have arrived. Once those numbers have arrived and have been written to the output file, we can go ahead and write this number. Since we do not know precisely what order the numbers appear in, it is not possible to say when all the numbers smaller than a given number have arrived and have been written to the output. However since

we are told that no number is off by more than k positions from its correctly sorted position, if more than k numbers greater than a given number have arrived and all the numbers smaller than the given number that arrived have been written, we can be sure that there are no more other smaller numbers that are going to arrive. Hence it is safe to write the given numbers.

This essentially gives us the strategy to always keep $k+1$ numbers in a min-heap. As soon as we read a new number, we extract the min from the heap and write the output and then insert the new number.

```
template <typename T>
void approximate_sort(istringstream &sin, const int &k) {
  priority_queue<T, vector<T>, greater<T>> min_heap;
  // Firstly push k elements into min_heap
  T x;
  for (int i = 0; i < k && sin >> x; ++i) {
    min_heap.push(x);
  }

  // Extract the minimum one for every incoming element
  while (sin >> x) {
    min_heap.push(x);
    cout << min_heap.top() << endl;
    min_heap.pop();
  }

  // Extract the remaining elements in min_heap
  while (min_heap.size()) {
    cout << min_heap.top() << endl;
    min_heap.pop();
  }
}
```

Problem 10.7, *pg. 82*: *Design an $O(n)$ time algorithm to compute the k elements closest to the median of an array A.*

Solution 10.7: There exists two standard algorithms for computing the median in $O(n)$ time—one uses randomized partitioning of the array; the other uses divide and conquer, specifically, it computes the median of the medians of $\lceil n/5 \rceil$ subarrays.

Assuming that we have computed the median μ in $O(n)$ time, we can compute the k elements closest to μ by maintaining a max-heap H of elements of the array. The value associated with the i-th element $A[i]$ is its distance to the median, i.e., $|\mu - A[i]|$. We start by adding the first k elements of the array to H. Now we process the remaining elements. For $j = k$ to $n-1$, if $|\mu - A[j]|$ is larger than the maximum value stored in the heap, we ignore it; otherwise, we remove the maximum element of H, and insert $A[j]$ in its place. When all elements are processed, the heap contains the k elements closest to the median.

Another approach, which does not require the $O(k)$ additional storage entailed by the max-heap, and runs in $O(n)$ time instead of $O(n \log k)$ time is to first compute the median μ, and then use a selection algorithm.

A selection algorithm takes as inputs a set A of n numbers, and an integer $i \in [1, n]$ and returns the i-th smallest element of A. There exists a practical selection algorithm, similar to quicksort, which runs in $O(n)$ expected time.

If we take $|A[j] - \mu|$ as the value of $A[j]$, and run the selection algorithm with $i = k$, we will get an element p. Let S be the elements strictly less than p. Suppose $|S| = k-1$. Then $\{p\} \cup S$ is the result. If $|S| < k - 1$, at least $k - |S|$ duplicates of p are present, in which case the union of any $k - |S|$ elements whose value is p with S is the result.

Note that both approaches start by computing the median, which changes the original array.

```
// Promote to double to prevent precision error
template <typename T>
double find_median(vector<T> &A) {
  int half = A.size() >> 1;
  nth_element(A.begin(), A.begin() + half, A.end());
  if (A.size() & 1) {  // A has odd number elements
    return A[half];
  } else {  // A has even number elements
    T x = A[half];
    nth_element(A.begin(), A.begin() + half - 1, A.end());
    return 0.5 * (x + A[half - 1]);
  }
}

template <typename T>
class Comp {
  private:
    double m_;

  public:
    Comp(const double &m) : m_(m) {};

    const bool operator()(const T &a, const T &b) const {
      return fabs(a - m_) < fabs(b - m_);
    }
};

template <typename T>
vector<T> find_k_closest_to_median(vector<T> A, const int &k) {
  // Find the element i where |A[i] - median| is k-th smallest
  nth_element(A.begin(), A.begin() + k - 1, A.end(), Comp<T>{find_median(A)});
  return {A.cbegin(), A.cbegin() + k};
}
```

Problem 10.8, *pg. 82*: *Design an algorithm for computing the running median of a sequence. The time complexity should be $O(\log n)$ per element read in, where n is the number of values read in up to that element.*

Solution 10.8: We use two heaps, L, a max-heap, and H, a min-heap. The invariant here is that for every incoming element from the stream, we want to let L store the smaller half of the stream data so far, and let H store the bigger half. By keeping this

invariant, we can output the median easily according to the number of elements we have seen so far. Following is the implementation in C++:

```
template <typename T>
void online_median(istringstream &sin) {
  // Min-heap stores the bigger part of the stream
  priority_queue<T, vector<T>, greater<T>> H;
  // Max-heap stores the smaller part of the stream
  priority_queue<T, vector<T>, less<T>> L;

  T x;
  while (sin >> x) {
    if (L.empty() == false && x > L.top()) {
      H.emplace(x);
    } else {
      L.emplace(x);
    }
    if (H.size() > L.size() + 1) {
      L.emplace(H.top());
      H.pop();
    } else if (L.size() > H.size() + 1) {
      H.emplace(L.top());
      L.pop();
    }

    if (H.size() == L.size()) {
      cout << 0.5 * (H.top() + L.top()) << endl;
    } else {
      cout << (H.size() > L.size() ? H.top() : L.top()) << endl;
    }
  }
}
```

Problem 10.9, *pg. 82*: *Design an algorithm for efficiently computing the k smallest real numbers of the form $a + b\sqrt{2}$ for nonnegative integers a and b.*

Solution 10.9: We can solve this problem using a min-heap H and a set S as follows. We initialize H to contain $0 + 0\sqrt{2} = 0$, and initialize S to the empty set. (A simple list will suffice to represent S). We now iteratively do the following, stopping when S has k elements. When we perform an extract-min from H to obtain a number $a + b\sqrt{2}$, we add it to H, and compute $c_1 = (a + 1) + b\sqrt{2}$ and $c_2 = a + (b + 1)\sqrt{2}$ which we add to H.

Suppose for the sake of contradiction that S is not the desired set. Since $|S| = k$, there has to be at least one number in the desired set that is not in S. Let the smallest such number be $m = p + q\sqrt{2}$. Note that p and q cannot both be 0. Similarly, there must be a number l that is in S and is greater than all numbers in S_2^k. If $p > 0$, consider the number $n = (p - 1) + q\sqrt{2}$. It is less than m, and greater than 0, so it must be in S, since S contains all numbers in the desired set that are smaller than m. But then when we processed n to put it in S, we would have added n to H. This contradicts our adding l to S—the heap would always return n before l.

It is possible for a number to be inserted twice into the heap. For example, both $1 + 2\sqrt{2}$ and $2 + \sqrt{2}$ produce $2 + 2\sqrt{2}$. No number can be inserted more than twice: the irrationality of $\sqrt{2}$ implies that $a + b\sqrt{2} = c + d\sqrt{2}$ iff $a = b$ and $c = d$. We can check for duplicates when we perform extract-min.

```
class Num {
  public:
    int a_, b_;
    double val_;

    Num(const int &a, const int &b) : a_(a), b_(b), val_(a + b * sqrt(2)) {}

    const bool operator<(const Num &n) const {
      return val_ > n.val_;
    }

    // Equal function for hash
    const bool operator==(const Num &n) const {
      return a_ == n.a_ && b_ == n.b_;
    }
};

// Hash function for Num
class HashNum {
  public:
    const size_t operator()(const Num &n) const {
      return hash<int>()(n.a_) ^ hash<int>()(n.b_);
    }
};

vector<Num> generate_first_k(const int &k) {
  priority_queue<Num, vector<Num>> min_heap;
  vector<Num> smallest;
  unordered_set<Num, HashNum> hash;

  // Initial for 0 + 0 * sqrt(2)
  min_heap.emplace(0, 0);
  hash.emplace(0, 0);

  while (smallest.size() < k) {
    Num s(min_heap.top());
    smallest.emplace_back(s);
    hash.erase(s);
    min_heap.pop();

    // Add the next two numbers derived from s
    Num c1(s.a_ + 1, s.b_), c2(s.a_, s.b_ + 1);
    if (hash.emplace(c1).second) {
      min_heap.emplace(c1);
    }
    if (hash.emplace(c2).second) {
      min_heap.emplace(c2);
    }
  }
```

```
  return smallest;
}
```

Problem 10.10, *pg. 83*: *Design an $O(k)$ time algorithm for determining whether the k-th largest element in a max-heap is smaller than, equal to, or larger than a given x. The max-heap is represented using an array. Your algorithm's time complexity should be independent of the number of elements in the max-heap, and may use $O(k)$ additional storage. It cannot make any changes to the max-heap, and should handle the possibility of duplicate entries.*

Solution 10.10: We count the number of elements that are greater than or equal to x. The key to achieving an $O(k)$ time complexity is visiting max-heap nodes in best-first order, and stopping the computation as soon as we have found more than k nodes greater than x.

We use two integer variables, `equal` and `larger`, which are initialized to 0 and passed by reference to recursive calls of the check function. If the element r at the root is smaller than x, we know there are no elements in the max-heap larger than x, so we return right away. Otherwise, if $r = x$ we increment the `equal` count by 1; if $r > x$ we increment the `larger` count by 1. We then recurse on the left child and the right child. At any stage, if we determine there are more than k keys greater than x or more than k keys equal to x we return. Within each call we do constant work, and for each recursive call, either we increment `equal` or `larger`, or we came from a call that performed such an increment, implying the number of recursive calls is $O(k)$.

```
template <typename T>
void compare_k_th_largest_heap_helper(const vector<T> &max_heap, const int &k,
                                      const T &x, const int &idx, int &larger,
                                      int &equal) {
  if (idx < max_heap.size()) {
    if (max_heap[idx] < x) {
      return;
    } else if (max_heap[idx] == x) {
      ++equal;
    } else {  // max_heap[idx] > x
      ++larger;
    }

    if (equal < k && larger < k) {
      compare_k_th_largest_heap_helper(max_heap, k, x, (idx << 1) + 1, larger,
                                       equal);
      compare_k_th_largest_heap_helper(max_heap, k, x, (idx << 1) + 2, larger,
                                       equal);
    }
  }
}

// -1 means smaller, 0 means equal, and 1 means larger
template <typename T>
int compare_k_th_largest_heap(const vector<T> &max_heap, const int &k,
                              const T &x) {
  int larger = 0, equal = 0;
```

```
28    compare_k_th_largest_heap_helper(max_heap, k, x, 0, larger, equal);
29    return larger >= k ? 1 : (larger + equal >= k ? 0 : -1);
30  }
```

Problem 11.1, ***pg. 86*****:** *Write a method that takes a sorted array A and a key k and returns the index of the* first *occurrence of k in A. Return −1 if k does not appear in A. For example, when applied to the array in Figure 11.1 on Page 86 your algorithm should return 3 if k = 108; if k = 285, your algorithm should return 6.*

Solution 11.1: The key idea is to search for k. However, even if we find k, after recording this we continue the search on the left subarray. The complexity bound is still $O(\log n)$—this is because each iteration reduces the size of the subarray being searched by half. In C++ code:

```
1  template <typename T>
2  int search_first(const vector<T> &A, const T &k) {
3    int l = 0, r = A.size() - 1, res = -1;
4    while (l <= r) {
5      int m = l + ((r - l) >> 1);
6      if (A[m] > k) {
7        r = m - 1;
8      } else if (A[m] == k) {
9        // Record the solution and keep searching the left part
10       res = m, r = m - 1;
11     } else {  // A[m] < k
12       l = m + 1;
13     }
14   }
15   return res;
16 }
```

ϵ-Variant 11.1.1: Let A be an unsorted array of n integers, with $A[0] \geq A[1]$ and $A[n-2] \leq A[n-1]$. Call an index i a *local minimum* if $A[i]$ is less than or equal to its neighbors. How would you efficiently find a local minimum, if one exists?

ϵ-Variant 11.1.2: A sequence is said to be ascending if each element is greater than or equal to its predecessor; a descending sequence is one in which each element is less than or equal to its predecessor. A sequence is strictly ascending if each element is greater than its predecessor. Suppose it is known that an array A consists of an ascending sequence followed by a descending sequence. Design an algorithm for finding the maximum element in A. Solve the same problem when A consists of a strictly ascending sequence, followed by a descending sequence.

Problem 11.2, ***pg. 86*****:** *Design an efficient algorithm that takes a sorted array A and a key k, and finds the index of the* first *occurrence an element larger than k; return −1 if every element is less than or equal to k. For example, when applied to the array in Figure 11.1 on Page 86 your algorithm should return −1 if k = 500; if k = 101, your algorithm should return 3.*

Solution 11.2: The naïve approach is to look for k via a binary search and then, if k is found, walk the array forward until either the first element larger than k is encountered or the end of the array is reached. If k is not found, binary search will end up pointing to either the first value greater than k in the array, in which case no further action is required or the last value smaller than k in which case the next element, if it exists, is the value that we are looking for. The worst-case run time of this algorithm is $\Theta(n)$—an array all of whose values equal k, except for the last one (which is greater than k), is the worst-case.

A better approach is to use binary search to eliminate half the candidates at each iteration: if we encounter an element larger than k, we record that element, and continue the search in the candidates on the left; otherwise, we continue searching in the candidates on the right. This algorithm has an $O(\log n)$ time complexity.

```
template <typename T>
int search_first_larger_k(const vector<T> &A, const T &k) {
  int l = 0, r = A.size() - 1, res = -1;
  while (l <= r) {
    int m = l + ((r - l) >> 1);
    if (A[m] > k) {
      // Record the solution and keep searching the left part
      res = m, r = m - 1;
    } else {  // A[m] <= k
      l = m + 1;
    }
  }
  return res;
}
```

Remark: As with Problem 11.1 on Page 86, the same problem can be posed for BSTs, and again, the solution is analogous to the one given above.

Problem 11.3, *pg. 86*: *Design an efficient algorithm that takes a sorted array A of distinct integers, and returns an index i such that $A[i] = i$ or indicate that no such index exists by returning -1. For example, when the input is the array shown in in Figure 11.1 on Page 86, your algorithm should return 2.*

Solution 11.3: Since the array contains distinct integers and is sorted, for any $i > 0$, $A[i] \geq A[i-1]+1$. Therefore $B[i] = A[i]-i$ is sorted. It follows that we can do a binary search for 0 in B to find an index such that $A[i] = i$. (We do not need to actually create B, we can simply use $A[i] - i$ wherever $B[i]$ is referenced.)

```
int search_index_value_equal(const vector<int> &A) {
  int l = 0, r = A.size() - 1;
  while (l <= r) {
    int m = l + ((r - l) >> 1);
    int val = A[m] - m;
    if (val == 0) {
      return m;
    } else if (val > 0) {
      r = m - 1;
    } else {  // val < 0
```

```
      l = m + 1;
    }
  }
  return -1;
}
```

Variant 11.3.1: Solve the same problem when A is sorted but may contain duplicates.

Problem 11.4, *pg. 87*: *Design an algorithm that takes an abs-sorted array A and a number k, and returns a pair of indices of elements in A that sum up to k. For example, if the input to your algorithm is the array in Figure 11.2 on Page 87 and $k = 167$, your algorithm should output* $(3, 7)$. *Output* $(-1, -1)$ *if there is no such pair.*

Solution 11.4: First consider the case where the array is sorted in the conventional sense. In this case we can 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.

This approach will not work when the array entries are sorted by absolute value. In this instance, we need to consider three cases:

(1.) Both the numbers in the pair are negative.

(2.) Both the numbers in the pair are positive.

(3.) One is negative and the other is positive.

For Cases (1.) and (2.), we can run the above algorithm separately by just limiting ourselves to either positive or negative numbers. For Case (3.), we can use the same approach where we have one index for positive numbers, one index for negative numbers, and they both start from the highest possible index and then go down.

```
template <typename T, typename Comp>
pair<int, int> find_pair_using_comp(const vector<T> &A, const T &k,
                                    Comp comp) {
  pair<int, int> ret(0, A.size() - 1);
  while (ret.first < ret.second && comp(A[ret.first], 0)) {
    ++ret.first;
  }
  while (ret.first < ret.second && comp(A[ret.second], 0)) {
    --ret.second;
  }

  while (ret.first < ret.second) {
    if (A[ret.first] + A[ret.second] == k) {
      return ret;
    } else if (comp(A[ret.first] + A[ret.second], k)) {
```

```
      do {
        ++ret.first;
      } while (ret.first < ret.second && comp(A[ret.first], 0));
    } else {
      do {
        --ret.second;
      } while (ret.first < ret.second && comp(A[ret.second], 0));
    }
  }
  return {-1, -1};  // no answer
}

template <typename T>
pair<int, int> find_pos_neg_pair(const vector<T> &A, const T &k) {
  // ret.first for positive, and ret.second for negative
  pair<int, int> ret(A.size() - 1, A.size() - 1);
  // Find the last positive or zero
  while (ret.first >= 0 && A[ret.first] < 0) {
    --ret.first;
  }

  // Find the last negative
  while (ret.second >= 0 && A[ret.second] >= 0) {
    --ret.second;
  }

  while (ret.first >= 0 && ret.second >= 0) {
    if (A[ret.first] + A[ret.second] == k) {
      return ret;
    } else if (A[ret.first] + A[ret.second] > k) {
      do {
        --ret.first;
      } while (ret.first >= 0 && A[ret.first] < 0);
    } else {  // A[ret.first] + A[ret.second] < k
      do {
        --ret.second;
      } while (ret.second >= 0 && A[ret.second] >= 0);
    }
  }
  return {-1, -1};  // no answer
}

template <typename T>
pair<int, int> find_pair_sum_k(const vector<T> &A, const T &k) {
  pair<int, int> ret = find_pos_neg_pair(A, k);
  if (ret.first == -1 && ret.second == -1) {
    return k >= 0 ? find_pair_using_comp(A, k, less<T>()) :
           find_pair_using_comp(A, k, greater_equal<T>());
  }
  return ret;
}
```

A simpler solution is based on a hash table (Chapter 12) to store all the numbers and then for each number x in the array, look up $k - x$ in the hash table. If the

hash function does a good job of spreading the keys, the time complexity for this approach is $O(n)$. However, it requires $O(n)$ additional storage. If the array is sorted on elements (and not absolute values), for each $A[i]$ we can use binary search to find $k - A[i]$. This approach uses $O(1)$ additional space and has time complexity $O(n \log n)$. However, it is strictly inferior to the two pointer technique described at the beginning of the solution.

Variant 11.4.1: Design an algorithm that takes as input an array of integers A, and an integer k, and returns a pair of indices i and j such that $A[j] - A[i] = k$, if such a pair exists.

Problem 11.5, *pg. 87*: *Design an $O(\log n)$ algorithm for finding the position of the smallest element in a cyclically sorted array. Assume all elements are distinct. For example, for the array in Figure 11.3 on Page 87, your algorithm should return 4.*

Solution 11.5: We make use of the decrease and conquer principle. Specifically, we maintain an interval of candidate indices, and iteratively eliminate a constant fraction of the indices in this interval. Let $I = [l_I, r_I]$ be the set of indices being considered, and m_I be the midpoint of I, i.e., $l_I + \lfloor \frac{r_I - l_I}{2} \rfloor$. If $A[m_I] > A[r_I]$ then $[l_I, m_I]$ cannot contain the index of the minimum element. Therefore we can restrict the search to $[m_I + 1, r_I]$. If $A[m_I] < A[r_I]$ we restrict our attention to $[l_I, m_I]$. We start with $I = [0, n-1]$, and end when the interval has one element.

```
template <typename T>
int search_smallest(const vector<T> &A) {
  int l = 0, r = A.size() - 1;
  while (l < r) {
    int m = l + ((r - l) >> 1);
    if (A[m] > A[r]) {
      l = m + 1;
    } else {  // A[m] <= A[r]
      r = m;
    }
  }
  return l;
}
```

Note that this problem cannot be solved in less than linear time when elements may be repeated. For example, if A consists of $n-1$ 1s and a single 0, that 0 cannot be detected in the worst case without inspecting every element. Following is the C++ code for the scenario when elements may be repeated:

```
template <typename T>
int search_smallest_helper(const vector<T> &A, const int &l, const int &r) {
  if (l == r) {
    return l;
  }

  int m = l + ((r - l) >> 1);
  if (A[m] > A[r]) {
```

```
    return search_smallest_helper(A, m + 1, r);
  } else if (A[m] < A[r]) {
    return search_smallest_helper(A, l, m);
  } else {  // A[m] == A[r]
    // Smallest element must exist in either left or right side
    int l_res = search_smallest_helper(A, l, m);
    int r_res = search_smallest_helper(A, m + 1, r);
    return A[r_res] < A[l_res] ? r_res : l_res;
  }
}

template <typename T>
int search_smallest(const vector<T> &A) {
  return search_smallest_helper(A, 0, A.size() - 1);
}
```

Variant 11.5.1: Design an $O(\log n)$ algorithm for finding the position of an element k in a cyclically sorted array.

Problem 11.6, *pg. 87*: *Let A be a sorted array. The length of A is not known in advance; accessing $A[i]$ for i beyond the end of the array throws an exception. Design an algorithm that takes A and a key k and returns an index i such that $A[i] = k$; return -1 if k does not appear in A.*

Solution 11.6: The key idea here is to simultaneously do a binary search for the end of the array as well as the key. We examine $A[2^p - 1]$ in the p-th step till we hit an exception or an entry greater than or equal to k. Then we do a conventional binary search for k in the range $[2^{p-1} + 1, 2^p - 2]$. The run time of the search algorithm is $O(\log n)$, where n is the length of A. In code:

```
template <typename T>
int binary_search_unknown_len(const vector<T> &A, const T &k) {
  // Find the possible range where k exists
  int p = 0;
  while (true) {
    try {
      T val = A.at((1 << p) - 1);
      if (val == k) {
        return (1 << p) - 1;
      } else if (val > k) {
        break;
      }
    }
    catch (exception& e) {
      break;
    }
    ++p;
  }

  // Binary search between indices 2^(p - 1) + 1 and 2^p - 2
  int l = (1 << (p - 1)) + 1, r = (1 << p) - 2;
  while (l <= r) {
```

```
    int m = l + ((r - l) >> 1);
    try {
      T val = A.at(m);
      if (val == k) {
        return m;
      } else if (val > k) {
        r = m - 1;
      } else {  // A[m] < k
        l = m + 1;
      }
    }
    catch (exception& e) {
      r = m - 1;  // search the left part if out of boundary
    }
  }
  return -1;  // nothing matched k
}
```

Problem 11.7, ***pg. 88***: *Let A be an array of n nonnegative real numbers and S' be a nonnegative real number less than $\sum_{i=0}^{n-1} A[i]$. Design an efficient algorithm for computing σ such that $\sum_{i=0}^{n-1} \min(A[i], \sigma) = S'$, if such a σ exists.*

Solution 11.7: Define $F(\sigma) = \sum_{i=0}^{n-1} \min(A[i], \sigma)$. We are looking for a value of σ such that $F(\sigma) = S'$. Clearly, F is a continuous function of σ. Since $0 \leq S' \leq \sum_{i=0}^{n-1} A[i]$, by the intermediate value theorem of calculus, there must exist a value of σ in $[0, \max_{i=0}^{n-1} A[i]]$ such that $F(\sigma) = S'$. Furthermore, since F monotonically increases with σ, we can perform binary search on the interval $[0, \max_{i=0}^{n-1} A[i]]$ to find the correct value of σ.

Assume that A is already sorted, i.e., for all i, $A[i] \leq A[i+1]$. Compute the prefix sum $z_k = \sum_{i=0}^{k-1} A[i]$. Now, suppose $A[k-1] \leq \sigma \leq A[k]$. Consequently, $F(\sigma) = (n-k)\sigma + z_k$.

Using the above expression, we can search for the value of k such that $F(A[k]) \leq S' \leq F(A[k+1])$ by performing binary search for k. (Since we sort A, the run time of our solution is already $O(n \log n)$, implying we could do a linear search without changing the time complexity.) Once we have found the right value of k, we can compute the value of σ by simply solving the equation for $F(\sigma)$ above.

The most expensive operation for this entire solution is sorting A, hence the run time is $O(n \log n)$. If we are given A sorted in advance and its prefix sums, then for each value of S', the search would have time complexity $O(\log n)$.

```
double completion_search(vector<double> &A, const double &budget) {
  sort(A.begin(), A.end());
  // Calculate the prefix sum for A
  vector<double> prefix_sum;
  partial_sum(A.cbegin(), A.cend(), back_inserter(prefix_sum));
  // costs[i] represents the total payroll if the cap is A[i]
  vector<double> costs;
  for (int i = 0; i < prefix_sum.size(); ++i) {
    costs.emplace_back(prefix_sum[i] + (A.size() - i - 1) * A[i]);
  }

```

```
  auto lower = lower_bound(costs.cbegin(), costs.cend(), budget);
  if (lower == costs.cend()) {
    return -1.0;  // no solution since budget is too large
  }

  if (lower == costs.cbegin()) {
    return budget / A.size();
  }
  int idx = lower - costs.cbegin() - 1;
  return A[idx] + (budget - costs[idx]) / (A.size() - idx - 1);
}
```

ϵ-Variant 11.7.1: Solve the same problem using only $O(1)$ space.

Problem 11.8, *pg. 88*: *You are given two sorted arrays A and B of lengths m and n, respectively, and a positive integer $k \in [1, m+n]$. Design an algorithm that runs in $O(\log k)$ time for computing the k-th smallest element in array formed by merging A and B. Array elements may be duplicated within and between A and B.*

Solution 11.8: Suppose the first k elements of the union of A and B consist of the first x elements of A and the first $k-x$ elements of B. We'll use binary search to determine x.

Specifically, we will maintain an interval $[l, u]$ that contains x, and use binary search to iteratively half the size of the interval. At each iteration we try $x = \lfloor \frac{l+u}{2} \rfloor$. If $l = u$, we simply return the larger of $A[x-1]$ and $B[k-x-1]$. (If $A[x-1] = B[k-1-x]$, we arbitrarily return either.) If $l \neq u$ but $A[x-1] = B[k-1-x]$, we return $A[x-1]$, since the first x elements of A and the first $k-x$ elements of B when sorted end in $A[x-1]$ or $B[k-1-x]$, which are equal.

Otherwise, if $A[x] < B[k-x-1]$, then $A[x]$ must be in the first k elements of the union, so we can update l to $x+1$. Conversely, if $B[k-x] < A[x-1]$ then $A[x-1]$ cannot be in the first k elements, so we can update u to $x-1$.

The initial values for l and u need to be chosen carefully. Naïvely setting $l = 0, u = k$ does not work, since this choice may lead to x lying outside the range of valid indices for B, i.e., outside $[0, n-1]$. Setting $l = \max(0, k-n)$ and $u = \min(m, k)$ resolves this problem.

```
template <typename T>
T find_kth_in_two_sorted_arrays(const vector<T> &A, const vector<T> &B,
                                const int &k) {
  // Lower bound of elements we will choose in A
  int l = max(0, static_cast<int>(k - B.size()));
  // Upper bound of elements we will choose in A
  int u = min(static_cast<int>(A.size()), k);

  while (l < u) {
    int x = l + ((u - l) >> 1);
    T A_x_1 = (x <= 0 ? numeric_limits<T>::min() : A[x - 1]);
    T A_x = (x >= A.size() ? numeric_limits<T>::max() : A[x]);
    T B_k_x_1 = (k - x <= 0 ? numeric_limits<T>::min() : B[k - x - 1]);
```

```
    T B_k_x = (k - x >= B.size() ? numeric_limits<T>::max() : B[k - x]);

    if (A_x < B_k_x_1) {
      l = x + 1;
    } else if (A_x_1 > B_k_x) {
      u = x - 1;
    } else {
      // B[k - x - 1] <= A[x] && A[x - 1] < B[k - x]
      return max(A_x_1, B_k_x_1);
    }
  }

  T A_l_1 = l <= 0 ? numeric_limits<T>::min() : A[l - 1];
  T B_k_l_1 = k - l - 1 < 0 ? numeric_limits<T>::min() : B[k - l - 1];
  return max(A_l_1, B_k_l_1);
}
```

Problem 11.9, ***pg. 88***: *Implement a function which takes as input a floating point variable x and returns $\sqrt{x}$.*

Solution 11.9: One of the fastest ways to invert a fast-growing monotone function (such as the square function) is to do a binary search. Given x, we start with a lower bound l and an upper bound u on $\sqrt{x}$. We iteratively check if the square of midpoint m of $[l, u]$ is smaller than, greater than, or equal to x. In the first case, we update the lower bound to m; in the second case, we update the upper bound to m; in the third case, we return m.

When checking for equality, we use a notion of tolerance, eps, since floating point arithmetic is not exact. This tolerance is user-specified.

Trivial choices for the initial lower bound and upper bound are 0 and the largest floating point number that is representable. If $x \geq 1.0$, we can tighten the lower and upper bounds to 1.0 and x, since $x \geq 1.0 \Rightarrow x^2 \geq x$. If $x < 1.0 \Rightarrow x^2 < x$, the previous choice of l and u is incorrect; instead, we can use x and 1.0. The time complexity is $O(\log \frac{x}{\text{eps}})$ since the number of iterations is affected by the choice of eps. Care has to be taken to ensure the compare function is resilient to finite precision effects.

```
// 0 means equal, -1 means smaller, 1 means larger
int compare(const double &a, const double &b) {
  // Use normalization for precision problem
  double diff = (a - b) / b;
  return diff < -numeric_limits<double>::epsilon() ?
         -1 : diff > numeric_limits<double>::epsilon();
}

double square_root(const double &x) {
  // Decide the search range according to x
  double l, r;
  if (compare(x, 1.0) < 0) {  // x < 1.0
    l = x, r = 1.0;
  } else {  // x >= 1.0
    l = 1.0, r = x;
  }
```

```
17
18    // Keep searching if l < r
19    while (compare(l, r) == -1) {
20      double m = l + 0.5 * (r - l);
21      double square_m = m * m;
22      if (compare(square_m, x) == 0) {
23        return m;
24      } else if (compare(square_m, x) == 1) {
25        r = m;
26      } else {
27        l = m;
28      }
29    }
30    return l;
31  }
```

Variant 11.9.1: Given two positive floating point numbers x and y, how would you compute $\frac{x}{y}$ to within a specified tolerance ϵ if the division operator cannot be used? You cannot use any library functions, such as *log* and *exp*; addition and multiplication are acceptable.

Problem 11.10, *pg. 88*: *Let A be an $n \times n$ 2D array where rows and columns are sorted in increasing sorted order. Design an efficient algorithm that decides whether a number x appears in A. How many entries of A does your algorithm inspect in the worst-case? Can you prove a tight lower bound that any such algorithm has to consider in the worst-case?*

Solution 11.10: One approach is to start by comparing x to $A[0][n-1]$. If $x = A[0][n-1]$, stop. Otherwise:

- $x > A[0][n-1]$, in which case x is greater than all elements in Row 0.
- $x < A[0][n-1]$, in which case x is less than all elements in Column $n-1$.

In either case, we have a 2D array with n fewer elements to search. In each iteration, we remove a row or a column, which means we inspect at most $2n-1$ elements.

```
1  template <typename T>
2  bool matrix_search(const vector<vector<T>> &A, const T &x) {
3    int r = 0, c = A[0].size() - 1;
4    while (r < A.size() && c >= 0) {
5      if (A[r][c] == x) {
6        return true;
7      } else if (A[r][c] < x) {
8        ++r;
9      } else {  // A[r][c] > x
10       --c;
11     }
12   }
13   return false;
14 }
```

For a tight lower bound, let x be any input. Define A to be

$$\begin{bmatrix} & & & & & x-1 \\ & & & & & x+1 \\ & & & & \cdots & \\ & & & x-1 & & \\ & & x-1 & x+1 & & \\ & x-1 & x+1 & & & \\ x-1 & x+1 & & & & \\ x+1 & & & & & \end{bmatrix}$$

where entries not shown are chosen so that the matrix is sorted by rows and by columns. We claim that any algorithm that solves the 2D array search problem will have to compare x with each of the $2n-1$ elements shown (i.e., the diagonal elements and the elements immediately below them). Call these elements the Δ elements.

Proof:

Comparing x with other elements does not eliminate any of the Δ elements. Suppose an algorithm did not compare x with one of the Δ elements. Then we could make that element x (instead of $x-1$ or $x+1$) and the algorithm would behave exactly as before and hence return the wrong result. Therefore at least $2n-1$ compares are necessary which means that the algorithm we designed is optimum.

Problem 11.11, ***pg. 89***: *How would you organize a tournament with 128 players to minimize the number of matches needed to find the best player? How many matches do you need to find the best and the second best player?*

Solution 11.11: First, we consider the problem of finding the best player. Each game eliminates one player and there are 128 players; so, 127 matches are necessary and also sufficient.

To find the second best, we note that the only candidates are the players who are beaten by the player who is eventually determined to be the best—everyone else lost to someone who is not the best.

To find the best player, the order in which we organize the matches is inconsequential—we just pick pairs from the set of candidates and whoever loses is removed from the pool of candidates. However if we proceed in an arbitrary order, we might start with the best player, who defeats 127 other players and then the players who lost need to play 126 matches amongst themselves to find the second best.

To find the second best player, we can do much better by organizing the matches as a complete binary tree. Specifically, we pair off all the players arbitrarily to form 64 matches. After these matches, we are left with 64 candidates; we pair them off again arbitrarily and they play 32 matches. Proceeding in this fashion, we organize the 127 matches needed to find the best player and the winner will have played 7 matches. Therefore we can find the second best player by organizing 6 matches between the 7

players who lost to the best player, for a total of $127 + (7 - 1) = 133$ matches.

Problem 11.12, ***pg. 89***: *Find the min and max elements from an array of n elements using no more than* $\lceil 3n/2 \rceil - 2$ *comparisons.*

Solution 11.12: If $n = 1$, no comparisons are needed. Suppose $n > 1$. Find the min m and the max M of the first two numbers; this requires a single comparison. Now process the remaining elements two at a time. Let (x, y) be such a pair. If $\min(x, y) < m$, update m; if $\max(x, y) > M$, update M. This entails three comparisons for each pair. If n is odd, the last update entails two comparisons, namely comparing the last element with m and M.

```
// Return (min, max) pair of elements in A
template <typename T>
pair<T, T> find_min_max(const vector<T> &A) {
  if (A.size() <= 1) {
    return {A.front(), A.front()};
  }

  // Initialize the min and max pair
  pair<T, T> min_max_pair = minmax(A[0], A[1]);
  for (int i = 2; i + 1 < A.size(); i += 2) {
    pair<T, T> local_pair = minmax(A[i], A[i + 1]);
    min_max_pair = {min(min_max_pair.first, local_pair.first),
                    max(min_max_pair.second, local_pair.second)};
  }
  // Special case: if there is odd number of elements in the array, we still
  // need to compare the last element with the existing answer.
  if (A.size() & 1) {
    min_max_pair = {min(min_max_pair.first, A.back()),
                    max(min_max_pair.second, A.back())};
  }
  return min_max_pair;
}
```

Variant 11.12.1: What is the least number of comparisons required to find the min and the max in the worst case?

Problem 11.13, ***pg. 89***: *Design an algorithm for computing the k-th largest element in an array A that runs in* $O(n)$ *expected time.*

Solution 11.13: The basic idea is to use decrease and conquer. We pick a random index r in the array A. Let $A[r] = x$. Reorder the elements in A in such a way that all elements that appear before index p are greater than x, and all elements that appear after p are less than or equal to x. Call the reordered array A'.

If $p = k - 1$, we are done—$A'[p]$ is the k-th largest element. Otherwise if $p > k - 1$, the element we are looking for is the k-th largest element of the subarray $A'[0 : p - 1]$. Finally, if $p < k - 1$, the element we seek is the $(k - (p + 1))$-th largest element of the subarray $A'[p + 1 : n - 1]$. Each of the two latter cases can be solved recursively.

Since we expect to split the array into roughly equal halves on average, intuitively, the expected time complexity $T(n)$ should satisfy $T(n) = O(n) + T(n/2)$. This solves to $T(n) = O(n)$. A more formal analysis requires the use of indicator random variables X_i, for $0 \leq i \leq k-1$, one per choice of t, and leads to the same conclusion.

```
// Partition A according pivot, return its index after partition
template <typename T>
int partition(vector<T> &A, const int &l, const int &r, const int &pivot) {
  T pivot_value = A[pivot];
  int larger_index = l;

  swap(A[pivot], A[r]);
  for (int i = l; i < r; ++i) {
    if (A[i] > pivot_value) {
      swap(A[i], A[larger_index++]);
    }
  }
  swap(A[r], A[larger_index]);
  return larger_index;
}

template <typename T>
T find_k_th_largest(vector<T> A, const int &k) {
  int l = 0, r = A.size() - 1;

  while (l <= r) {
    default_random_engine gen((random_device())());
    uniform_int_distribution<int> dis(l, r);  // generate random int in [l, r]
    int p = partition(A, l, r, dis(gen));
    if (p == k - 1) {
      return A[p];
    } else if (p > k - 1) {
      r = p - 1;
    } else {  // p < k - 1
      l = p + 1;
    }
  }
}
```

ϵ-Variant 11.13.1: Design an algorithm for finding the k-th largest element of A in the presence of duplicates. The k-th largest element is defined to be $A[k-1]$ after A has been sorted in a stable manner, i.e., if $A[i] = A[j]$ and $i < j$ then $A[i]$ must appear before $A[j]$ after stable sorting.

Problem 11.14, *pg. 89*: *Design an algorithm for computing the k-th largest element in a sequence of elements. It should run in $O(n)$ expected time where n is the length of the sequence, which is not known in advance. The value k is known in advance. Your algorithm should print the k-th largest element after the sequence has ended. It should use $O(k)$ additional storage.*

Solution 11.14: The natural approach is to use a min-heap containing the k largest

elements seen thus far. As each new element e is read, it is compared with the value of the smallest element m in the min-heap: if $e \leq m$, we continue; otherwise we delete m and insert e. This approach has time complexity $O(n \log k)$, since inserts and deletes take $O(\log k)$ time, and when elements are presented in ascending order, each new element requires an insert and a delete.

A better approach is to keep the k largest elements in an array M of length $2k - 1$. We add elements to M, and each time M is full, we find the k largest elements using the selection algorithm in Solution 11.13 on Page 270. The smaller elements are discarded, and we continue. The selection algorithm takes $O(k)$ time and is run every k elements, implying an $O(n)$ time complexity.

```
template <typename T>
T find_k_th_largest_unknown_length(istringstream &sin, const int &k) {
  vector<T> M;
  T x;
  while (sin >> x) {
    M.emplace_back(x);
    if (M.size() == (k << 1) - 1) {
      // Keep the k largest elements and discard the small ones
      nth_element(M.begin(), M.begin() + k - 1, M.end(), greater<T>());
      M.resize(k);
    }
  }
  nth_element(M.begin(), M.begin() + k - 1, M.end(), greater<T>());
  return M[k - 1];  // return the k-th largest one
}
```

Problem 11.15, ***pg. 90***: *Suppose you were given a file containing roughly one billion Internet Protocol (IP) addresses, each of which is a 32-bit unsigned integer. How would you programmatically find an IP address that is not in the file? Assume you have unlimited drive space but only two megabytes of RAM at your disposal.*

Solution 11.15: In the first step, we build an array of 2^{16} 32-bit unsigned integers that is initialized to 0 and for every IP address in the file, we take its 16 most significant bits to index into this array and increment the count of that number. Since the file contains fewer than 2^{32} numbers, there must be one entry in the array that is less than 2^{16}. This tells us that there is at least one IP address which has those upper bits and is not in the file. In the second pass, we can focus only on the addresses that match this criterion and use a bit array of size 2^{16} to identify one of the missing numbers.

```
int find_missing_element(ifstream &ifs) {
  vector<size_t> counter(1 << 16, 0);
  unsigned int x;
  while (ifs >> x) {
    ++counter[x >> 16];
  }

  for (int i = 0; i < counter.size(); ++i) {
    // Find one bucket contains less than (1 << 16) elements
    if (counter[i] < (1 << 16)) {
```

```
      bitset<(1 << 16)> bit_vec;
      ifs.clear();
      ifs.seekg(0, ios::beg);
      while (ifs >> x) {
        if (i == (x >> 16)) {
          bit_vec.set(((1 << 16) - 1) & x);  // gets the lower 16 bits of x
        }
      }
      ifs.close();

      for (int j = 0; j < (1 << 16); ++j) {
        if (bit_vec.test(j) == 0) {
          return (i << 16) | j;
        }
      }
    }
  }
}
```

Problem 11.16, ***pg. 90***: *Let A be an array of n integers in $\mathcal{Z}_n$, with exactly one element t appearing twice. This implies exactly one element $m \in \mathcal{Z}_n$ is missing from A. How would you compute t and m in $O(n)$ time and $O(1)$ space?*

Solution 11.16: Let $\text{Sum}(\mathcal{Z}_n)$ be the sum of the elements in $\mathcal{Z}_n$, and $\text{Sqr}(\mathcal{Z}_n)$ be the sum of the squares of the elements in $\mathcal{Z}_n$. The sum of the elements in A is exactly $\text{Sum}(\mathcal{Z}_n) + t - m$, and the sum of the squares of the elements in A is exactly $\text{Sqr}(\mathcal{Z}_n) + t^2 - m^2$. It is straightforward to compute $m - t$: initialize sum to 0, and add $(i - A[i])$ to sum for each index i. A similar computation yields $m^2 - t^2$ (i.e., square_sum in code). Factoring and canceling the expression $\frac{m^2-t^2}{m-t}$ yields $m + t$, to which we add $m - t$ to obtain $2m$ and subtract $m - t$ to obtain $2t$. Details are given below:

```
// Return pair<int, int>(duplicate, missing)
pair<int, int> find_duplicate_missing(const vector<int> &A) {
  int sum = 0, square_sum = 0;
  for (int i = 0; i < A.size(); ++i) {
    sum += i - A[i], square_sum += i * i - A[i] * A[i];
  }
  return {(square_sum / sum - sum) >> 1, (square_sum / sum + sum) >> 1};
}
```

The problem with the approach above is that it can lead to overflow. A substantially better approach is to compute the XOR of all the elements in $\mathcal{Z}_n$ and A—this yields $m \oplus t$. Since $m \neq t$, there must be some bit in $m \oplus t$ that is set to 1, i.e., m and t differ in that bit. We then compute the XOR of all the elements in $\mathcal{Z}_n$ and in A in which that bit is 1. Let this XOR be h. By the logic described in the problem statement, h must be one of m or t. We can scan through A to determine if h is the duplicate or the missing element. This approach is simpler and, since it requires no arithmetic, it cannot result in an overflow. A disadvantage is that it requires three passes through A.

```
// Return pair<int, int>(duplicate, missing)
pair<int, int> find_duplicate_missing(const vector<int> &A) {
  int miss_XOR_dup = 0;
  for (int i = 0; i < A.size(); ++i) {
    miss_XOR_dup ^= i ^ A[i];
  }

  int differ_bit = miss_XOR_dup & (~(miss_XOR_dup - 1)), miss_or_dup = 0;
  for (int i = 0; i < A.size(); ++i) {
    if (i & differ_bit) {
      miss_or_dup ^= i;
    }
    if (A[i] & differ_bit) {
      miss_or_dup ^= A[i];
    }
  }

  for (const int &A_i : A) {
    if (A_i == miss_or_dup) {   // find duplicate
      return {miss_or_dup, miss_or_dup ^ miss_XOR_dup};
    }
  }
  // miss_or_dup is missing element
  return {miss_or_dup ^ miss_XOR_dup, miss_or_dup};
}
```

Problem 11.17, *pg. 90*: *Given an array A, in which each element of A appears three times except for one element e that appears once, find e in $O(1)$ space and $O(n)$ time.*

Solution 11.17: One way to view Solution 11.16 on the preceding page is that it counts modulo 2 for each bit-position the number of entries in which the bit in that position is 1. Specifically, the XOR of elements at indices $[0, i-1]$, determines exactly which bit-positions have been 1 an odd number of times in elements of A whose indices are in $[0, i-1]$.

The analogous approach for the current problem is to count modulo 3 for each bit-position the number of times the bit in that position has been 1. The effect of counting modulo 3 is to eliminate the elements that appear three times, and so the bit-positions which have a count of 1 are precisely those bit-positions in e which are set to 1.

Representing a number modulo 3 requires two bits. We use two integer-valued variables, `ones` and `twos`, to do the counting. The variable `ones` denotes whether a bit-position has been set once (modulo 3) so far; the variable `twos` denotes whether a bit-position has been set twice (modulo 3) so far. When a bit-position has a count of 2 (modulo 3) and another 1 is observed, we reset the `ones` and `twos` variables.

Suppose `ones` and `twos` have been set appropriately after reading in the first $i-1$ elements. After reading A_{i-1}, bit-position j has a count of 1 modulo 3 iff it had a count of 1 modulo 3 and the j-th bit in A_i is a zero or the count was 0 modulo 3 and the j-th bit in A_i is a one. This gives us the update equation for `ones`.

After reading A_{i-1}, bit-position j has a count of 2 modulo 3 iff it had a count of 2 modulo 3 and the j-th bit in A_{i-1} is a zero or the count was 1 modulo 3 and the j-th bit in i is a one. This gives us the update equation for `twos`.

The code below implements the update equations; for the reasons described above, the final result is `ones`.

```
int find_element_appears_once(const vector<int> &A) {
  int ones = 0, twos = 0;
  int next_ones, next_twos;
  for (const int &i : A) {
    next_ones = (~i & ones) | (i & ~ones & ~twos);
    next_twos = (~i & twos) | (i & ones);
    ones = next_ones, twos = next_twos;
  }
  return ones;
}
```

Problem 11.18, ***pg. 91***: *Design an efficient algorithm that takes a close array A, and a key k and searches for any index j such that $A[j] = k$. Return -1 if no such index exists. For example, for the array in Figure 11.4 on Page 91, if $k = 2$, your algorithm should return an index in $\{4, 5, 7\}$.*

Solution 11.18: The close property allows us to skip indices: if $|A[i] - k| = l$, then for no index $i' \in (i - l, i + l)$ can $A[i'] = k$. We use this test to speedup the basic iterative search through an array in the code given below.

```
int close_search(const vector<int> &A, const int &k) {
  int idx = 0;
  while (idx < A.size() && A[idx] != k) {
    idx += abs(A[idx] - k);
  }
  return idx < A.size() ? idx : -1;  // -1 means no result
}
```

A worst-case input is one where all elements have value $k - 1$, in which case the algorithm is forced to inspect all elements. Hence its time complexity is $O(n)$, where n is the length of the close array. (Recall time complexity measures performance on worst-case inputs.) In the best case, our algorithm inspects $1/k$-th of the array, e.g., if all elements are 0.

Problem 11.19, ***pg. 91***: *You are reading a sequence of words from a very long stream. You know* a priori *that more than half the words are repetitions of a single word w (the "majority element") but the positions where w occurs are unknown. Design an algorithm that makes a single pass over the stream and uses only a constant amount of memory to identify w.*

Solution 11.19: The following observation leads to an elegant solution. If you take any two distinct elements from the stream and discard them away, the majority element remains the majority of the remaining elements. (This hinges on the assumption that there exists a majority element to begin with). The reasoning is follows.

Proof:

Let's say the majority element occurred m times out of n elements in the stream such that $\frac{m}{n} > \frac{1}{2}$. The two distinct elements that are discarded can have at most one of the majority elements. Hence after discarding them, the ratio of the previously majority element to the total number of elements is either $\frac{m}{(n-2)}$ or $\frac{(m-1)}{(n-2)}$. It is simple to verify that if $\frac{m}{n} > \frac{1}{2}$, then $\frac{m}{(n-2)} > \frac{(m-1)}{(n-2)} > \frac{1}{2}$.

Now, as we read the stream from beginning to the end, as soon as we encounter more than one distinct element, we can discard one instance of each element and what we are left with in the end must be the majority element.

```
string majority_search(istringstream &sin) {
  string candidate, buf;
  int count = 0;
  while (sin >> buf) {
    if (count == 0) {
      candidate = buf;
      count = 1;
    } else if (candidate == buf) {
      ++count;
    } else {
      --count;
    }
  }
  return candidate;
}
```

The code above assumes a majority word exists in the sequence. If no word has a strict majority, it still returns a word from the stream, albeit without any meaningful guarantees on how common that word is. We could check with a second pass whether the returned word was a majority. Similar ideas can be used to identify words that appear more than n/k times in the sequence, as discussed in Problem 12.11 on Page 96.

Problem 12.1, *pg. 92*: *Design a hash function that is suitable for words in a dictionary.*

Solution 12.1: First, the hash function should examine all the characters in each word. (If this seem obvious, the string hash function in the original distribution of Java examined at most 16 characters, in an attempt to gain speed, but often resulted in very poor performance because of collisions.) It should give a large range of values, and should not let one character dominate (e.g., if we simply cast characters to integers and multiplied them, a single 0 would result in a hash code of 0). We would also like a rolling hash function, one in which if a character is deleted from the front of the string, and another added to the end, the new hash code can be computed in $O(1)$ time (see Solution 12.13 on Page 286). The following function has these properties:

```
int string_hash(const string &str, const int &modulus) {
  const int MULT = 997;
  int val = 0;
  for (const char &c : str) {
```

```
    val = (val * MULT + c) % modulus;
  }
  return val;
}
```

Problem 12.2, ***pg. 93*****:** *Design a hash function for chess game states. Your function should take a state and the hash code for that state, and a move, and efficiently compute the hash code for the updated state.*

Solution 12.2: A straightforward hash function is to treat the 4×64 bits that constitute the board as a sequence of 64 digits in base 13, and use the hash function $\sum_{i=0}^{63} c_i p^i$, where c_i is the digit in location i, and p is a prime (see Solution 12.1 on the facing page).

This hash function does have ability to be updated incrementally—if, for example, a black knight on one square is replaced by a white bishop, the hash code update simply requires subtracting $c_k^B p^{i_1}$ and $c_b^W p^{i_2}$, and adding $c_b^W p^{i_1}$ and $c_\emptyset p^{i_2}$, where i_1 and i_2 are the initial locations of the knight and the bishop, respectively, and $c_k^B, c_b^W, c_\emptyset$ are the codes for black knight, white bishop, and empty space, respectively.

A more efficient hash function is based on creating a random 64-bit integer code for each of the 4×64 assignments of pieces to squares. The hash code for the state of the chessboard is the XOR of the code for each piece. Updates are trivial—for the example above, we XOR the code for black knight on i_1, white bishop on i_2, white bishop on i_1, and blank on i_2.

Problem 12.3, ***pg. 93*****:** *Let s be an array of strings. Write a function which finds a closest pair of equal entries. For example, if s = ["All", "work", "and", "no", "play", "makes", "for", "no", "work", "no", "fun", "and", "no", "results"], then the second and third occurrences of "no" is the closest pair.*

Solution 12.3: We make a scan through the array. For each i, we determine the index j of the most recent occurrence of $s[i]$. If $i - j$ is less than the difference of the closest duplicate pair seen so far, we update that difference to $i - j$. The most recent occurrence of $s[i]$ is computed through a hash table lookup. The time complexity is $O(n)$, since we perform a constant amount of work per entry. The space complexity is $O(d)$, where d is the number of distinct strings in the array.

```
int find_nearest_repetition(const vector<string> &s) {
  unordered_map<string, int> string_to_location;
  int closest_dis = numeric_limits<int>::max();
  for (int i = 0; i < s.size(); ++i) {
    auto it = string_to_location.find(s[i]);
    if (it != string_to_location.end()) {
      closest_dis = min(closest_dis, i - it->second);
    }
    string_to_location[s[i]] = i;
  }
  return closest_dis;
}
```

Problem 12.4, pg. 94: *Given a set of binary trees $A_1, \ldots, A_n$ how would you compute a new set of binary trees $B_1, \ldots, B_n$ such that for each i, $1 \leq i \leq n$, A_i and B_i are isomorphic, and no pair of isomorphic nodes exists in the set of nodes defined by $B_1, \ldots, B_n$. (This is sometimes referred to as the canonical form.) Assume nodes are not shared in $A_1, \ldots, A_n$. See Figure 12.2 on Page 94 for an example.*

Solution 12.4: We will refer to $B_1, \ldots, B_n$ as the *canonical form* for $A_1, \ldots, A_n$. We can greatly accelerate the computation of the canonical form by caching. Specifically, we will cache the hash code for canonical nodes. Also, to compute the canonical node for a node n in some A_i, we will first compute the canonical nodes for n's children.

We need to define a hash function and an equality function for nodes. The hash function must have the property that isomorphic nodes are mapped to identical hash codes. The equality function should implement the isomorphism check.

The equality function can be implemented directly from the definition of isomorphism. The hash function can also be implemented fairly easily, e.g., $h(\texttt{null}) = 1$ and $h(x) = 3h(x.key) + 5h(x.left) + 7h(x.right)$.

```
static class BinaryTreeNode {
  int key;
  BinaryTreeNode left, right;
  Integer cachedHash;

  public BinaryTreeNode(int k, BinaryTreeNode l, BinaryTreeNode r) {
    this.key = k;
    this.left = l;
    this.right = r;
    this.cachedHash = null;
  }

  @Override
  public int hashCode() {
    if (this.cachedHash != null) {
      return this.cachedHash;
    }

    int x = 3 * key;
    int y = this.left == null ? 5 : 5 * this.left.hashCode();
    int z = this.right == null ? 7 : 7 * this.right.hashCode();
    this.cachedHash = x + y + z;
    return this.cachedHash;
  }

  @Override
  public boolean equals(Object o) {
    if (o == this) {
      return true;
    }
    if (!(o instanceof BinaryTreeNode)) {
      return false;
    }
    BinaryTreeNode n = (BinaryTreeNode)o;

```

```
    if (n == null || key != n.key) {
      return false;
    }
    // Assuming that equals is called on nodes
    // where children are already in canonical form
    return (left == n.left && right == n.right);
  }
}

static Map<BinaryTreeNode, BinaryTreeNode> nodeToCanonicalNode =
  new HashMap<BinaryTreeNode, BinaryTreeNode>();

static BinaryTreeNode getCanonical(BinaryTreeNode n) {
  BinaryTreeNode lc = (n.left == null) ? null : getCanonical(n.left);
  BinaryTreeNode rc = (n.right == null) ? null : getCanonical(n.right);
  BinaryTreeNode nc = new BinaryTreeNode(n.key, lc, rc);

  if (nodeToCanonicalNode.containsKey(nc)) {
    return nodeToCanonicalNode.get(nc);
  }
  nodeToCanonicalNode.put(nc, nc);
  return nc;
}
```

Incidentally, the implementation above illustrates what is known as the *flyweight pattern*.

ϵ-Variant 12.4.1: Design an efficient algorithm that computes the largest subtree common to two binary trees.

Problem 12.5, *pg. 94*: *You are given a sequence of users where each user has a unique 32-bit integer key and a set of attributes specified as strings. When you read a user, you should pair that user with another previously read user with identical attributes who is currently unpaired, if such a user exists. If the user cannot be paired, you should keep him in the unpaired set. How would you implement this matching process efficiently?*

Solution 12.5: Each user is associated with a set of attributes and we need to find users associated with a given set of attributes quickly. A hash table would be a perfect solution here but we need a hash function over the set of attributes. If the number of attributes is small, we can represent a subset of attributes as a bit array, where each bit represents a specific attribute. Once we have a canonical representation for sets, then we can use any hash function for bit arrays.

If the set of possible attributes is large, a better way to canonically represent a subset of attributes is to sort the attributes. (Any arbitrary ordering of attributes will work.) We can represent the sorted sequence of attributes as a string by concatenating the individual elements, and use a hash function for strings.

Problem 12.6, *pg. 95*: *Solve Problem 12.5 on Page 94 when users are grouped based on having similar attributes. The similarity between two sets of attributes A and B is* $\frac{|A \cap B|}{|A \cup B|}$.

Solution 12.6: Grouping users based on similarity makes the problem significantly more difficult. *Min-hashing* is a common approach. Essentially, we construct a set of k independent hash functions, $h_1, h_2, \ldots, h_k$. Then for the set of attributes S of each user, we define

$$M_k(S) = \min_{a_i \in S} h_k(a_i).$$

If two sets of attributes S_1 and S_2 are similar, then the probability of $M_k(S_1) = M_k(S_2)$ is high, specifically it is $|S_1 \cap S_2|/|S_1 \cap S_2|$. We map each set of attributes S to the sequence $\langle M_1(S), M_2(S), \ldots, M_k(S) \rangle$. We can use one of two criterion for identifying similar users. The first is that users who have the same sequences are potentially similar. The second is that users who have any hash code in common are candidates for being similar. The first criterion will have a higher false negative rate, whereas the second will have a higher false positive rate. In either case the problem has been reduced to hashing. The parameter k can be varied to increase or decrease the likelihood of false negatives.

Problem 12.7, *pg. 95*: *Write a function that takes as input a dictionary of English words, and returns a partition of the dictionary into subsets of words that are all anagrams of each other.*

Solution 12.7: Given a string s, let sort(s) be the string consisting of the characters in s rearranged so that they appear in sorted order. Observe that x and y are anagrams iff sort(x) = sort(y). For example, sort("logarithmic") and sort("algorithmic") are both "acghiilmort". Therefore anagrams can be identified by adding sort(s) for each string s in the dictionary to a hash table.

```
1  void find_anagrams(const vector<string> &dictionary) {
2    // Get the sorted string and then insert into hash table
3    unordered_map<string, vector<string>> hash;
4    for (const string &s : dictionary) {
5      string sorted_str(s);
6      // Use sorted string as the hash code
7      sort(sorted_str.begin(), sorted_str.end());
8      hash[sorted_str].emplace_back(s);
9    }
10
11   for (const pair<string, vector<string>> &p : hash) {
12     // Multiple strings with the same hash code => anagrams
13     if (p.second.size() >= 2) {
14       // Output all strings
15       copy(p.second.begin(), p.second.end(),
16            ostream_iterator<string>(cout, " "));
17       cout << endl;
18     }
19   }
20 }
```

Problem 12.8, *pg. 95*: *Write a program to test whether the letters forming a string s can be permuted to form a palindrome. For example, "edified" can be permuted to form "deified".*

Explore solutions that trade time for space.

Solution 12.8: If the string is of even length, a necessary and sufficient condition for it to be a palindrome is that each character in the string appear an even number of times. If the length is odd, all but one character should appear an even number of times. Both these cases are covered by testing that at most one character appears an odd number of times, which can be checked using a hash table mapping characters to frequencies.

```
bool can_string_be_a_palindrome(const string &s) {
  unordered_map<char, int> hash;
  // Insert each char into hash
  for_each(s.begin(), s.end(), [&hash](const char &c) { ++hash[c]; });

  // A string can be permuted as a palindrome if the number of odd time
  // chars <= 1
  int odd_count = 0;
  for (const pair<char, int> &p : hash) {
    if (p.second & 1 && ++odd_count > 1) {
      break;
    }
  }
  return odd_count <= 1;
}
```

When the character set is large, we can perform the check without additional storage by sorting the string—this can be done in $O(n \log n)$ time and $O(1)$ space. Then we make a pass through the string determining character frequencies. (This approach changes the string itself.)

```
bool can_string_be_a_palindrome(string s) {
  sort(s.begin(), s.end());
  int odd_count = 0, num_curr_char = 1;

  for (int i = 1; i < s.size() && odd_count <= 1; ++i) {
    if (s[i] != s[i - 1]) {
      if (num_curr_char & 1) {
        ++odd_count;
      }
      num_curr_char = 1;
    } else {
      ++num_curr_char;
    }
  }
  if (num_curr_char & 1) {
    ++odd_count;
  }

  // A string can be permuted as a palindrome if the number of odd time
  // chars <= 1
  return odd_count <= 1;
}
```

Problem 12.9, ***pg. 95***: *You are required to write a method which takes an anonymous letter L and text from a magazine M. Your method is to return* `true` *iff L can be written using M, i.e., if a letter appears k times in L, it must appear at least k times in M.*

Solution 12.9: In the problem scenario, it is likely that the string encoding the magazine is much longer than the string encoding the anonymous letter. We build a hash table H_L for L, where each key is a character in the letter and its value is the number of times it appears in the letter. Consequently, we scan the magazine character-by-character. When processing c, if c appears in H_L, we reduce its frequency count by 1; we remove it from H_L when its count goes to zero. If H_L becomes empty, we return `true`. If it is nonempty when we get to the end of M, we return `false`.

```
bool anonymous_letter(const string &L, const string &M) {
  unordered_map<char, int> hash;
  // Insert all chars in L into a hash table
  for_each(L.begin(), L.end(), [&hash](const char &c) { ++hash[c]; });

  // Check chars in M that could cover chars in a hash table
  for (const char &c : M) {
    auto it = hash.find(c);
    if (it != hash.cend()) {
      if (--it->second == 0) {
        hash.erase(it);
        if (hash.empty() == true) {
          return true;
        }
      }
    }
  }
  // No entry in hash means L can be covered by M
  return hash.empty();
}
```

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

Problem 12.10, ***pg. 95***: *Let P be a set of n points in the plane. Each point has integer coordinates. Design an efficient algorithm for computing a line that contains the maximum number of points in P.*

Solution 12.10: Every pair of distinct points defines a line. We can use a hash table H to map lines to the set of points in P that lie on them. (Each corresponding set of points itself could be stored using a hash table.)

There are $\frac{n(n-1)}{2}$ pairs of points, and for each pair we have to do a lookup in H, an insert into H if the defined line is not already in H, and two inserts into the corresponding set of points. The hash table operations are $O(1)$ time, leading to an $O(n^2)$ time bound for this part of the computation.

We finish by finding the line with the maximum number of points with a simple iteration through the hash table searching for the line with the most points in its

corresponding set.

The design of a hash function appropriate for lines is more challenging than it may seem at first. The equation of line through (x_1, y_1) and (x_2, y_2) is

$$y = \frac{y_2 - y_1}{x_2 - x_1}x + \frac{x_2y_1 - x_1y_2}{x_2 - x_1}.$$

One idea would be to compute a hash code from the slope and the y-intercept of this line as an ordered pair of doubles. Because of finite precision arithmetic, we may have three points that are collinear map to distinct buckets.

A more robust hash function treats the slope and the y-intercept as rationals. A rational is an ordered pair of integers: the numerator and the denominator. We need to bring the rational into a canonical form before applying the hash function. One canonical form is to make the denominator always nonnegative, and relatively prime to the numerator. Lines parallel to the y-axis are a special case. For such lines, we use the x-intercept in place of the y-intercept, and use $\frac{1}{0}$ as the slope.

```
class Point {
  public:
    int x, y;

    //Equal function for hash
    const bool operator==(const Point &that) const {
      return x == that.x && y == that.y;
    }
};

// Hash function for Point
class HashPoint {
  public:
    const size_t operator()(const Point &p) const {
      return hash<int>()(p.x) ^ hash<int>()(p.y);
    }
};

// Line function of two points, a and b, and the equation is
// y = x(b.y - a.y) / (b.x - a.x) + (b.x * a.y - a.x * b.y) / (b.x - a.x)
class Line {
  private:
    static pair<int, int> get_canonical_fractional(int a, int b) {
      int gcd = GCD(abs(a), abs(b));
      a /= gcd, b /= gcd;
      return b < 0 ? make_pair(-a, -b) : make_pair(a, b);
    }

  public:
    // Store the numerator and denominator pair of slope unless the line is
    // parallel to y-axis that we store 1/0
    pair<int, int> slope;
    // Store the numerator and denominator pair of the y-intercept unless
    // the line is parallel to y-axis that we store the x-intercept
    pair<int, int> intercept;
```

```
36
37      Line(const Point &a, const Point &b) :
38        slope(a.x != b.x ? get_canonical_fractional(b.y - a.y, b.x - a.x) :
39              make_pair(1, 0)),
40        intercept(a.x != b.x ?
41                  get_canonical_fractional(b.x * a.y - a.x * b.y, b.x - a.x) :
42                  make_pair(a.x, 1)) {}
43
44      // Equal function for hash
45      const bool operator==(const Line &that) const {
46        return slope == that.slope && intercept == that.intercept;
47      }
48  };
49
50  // Hash function for Line
51  class HashLine {
52    public:
53      const size_t operator()(const Line &l) const {
54        return hash<int>()(l.slope.first) ^ hash<int>()(l.slope.second) ^
55               hash<int>()(l.intercept.first) ^ hash<int>()(l.intercept.second);
56      }
57  };
58
59  Line find_line_with_most_points(const vector<Point> &P) {
60    // Add all possible lines into hash table
61    unordered_map<Line, unordered_set<Point, HashPoint>, HashLine> table;
62    for (int i = 0; i < P.size(); ++i) {
63      for (int j = i + 1; j < P.size(); ++j) {
64        Line l(P[i], P[j]);
65        table[l].emplace(P[i]), table[l].emplace(P[j]);
66      }
67    }
68
69    // Return the line with most points have passed
70    return max_element(table.cbegin(), table.cend(),
71      [](const pair<Line, unordered_set<Point, HashPoint>> &a,
72         const pair<Line, unordered_set<Point, HashPoint>> &b) {
73        return a.second.size() < b.second.size();
74      })->first;
75  }
```

Problem 12.11, *pg. 96*: *You are reading a sequence of strings separated by white space. You are allowed to read the sequence twice. Devise an algorithm that uses $O(k)$ memory to identify the words that occur at least $\lceil \frac{n}{k} \rceil$ times, where n is the length of the sequence.*

Solution 12.11: This is essentially a generalization of Problem 11.19 on Page 91. Here instead of discarding two distinct words, we discard $k+1$ distinct words at any given time and we are guaranteed that all the words that occurred more than $\frac{1}{k}$ times the length of the sequence prior to discarding continue to appear more than $\frac{1}{k}$ times in the remaining sequence. To implement this strategy, we need a hash table of the current candidates. Here is the code in C++:

```
vector<string> Search_frequent_items(istringstream &sin, const int &k) {
  // Find the candidates which may occur >= n / k times
  string buf;
  unordered_map<string, int> hash;
  int n = 0;  // count the number of strings

  while (sin >> buf) {
    ++hash[buf], ++n;
    // Detecting k + 1 items in hash, at least one of them must have exactly 1
    // in it. We will discard those k + 1 items by 1 for each.
    if (hash.size() == k + 1) {
      auto it = hash.begin();
      while (it != hash.end()) {
        if (--(it->second) == 0) {
          hash.erase(it++);  // remove the empty entry
        } else {
          ++it;
        }
      }
    }
  }

  // Reset hash for the following counting
  for (auto &it : hash) {
    it.second = 0;
  }

  // Reset the stream and read it again
  sin.clear();
  sin.seekg(0, ios::beg);
  // Count the occurrence of each candidate word
  while (sin >> buf) {
    auto it = hash.find(buf);
    if (it != hash.end()) {
      ++it->second;
    }
  }

  vector<string> ret;
  for (const pair<string, int> &it : hash) {
    // Select the word which occurs >= n / k times
    if (it.second >= static_cast<double>(n) / k) {
      ret.emplace_back(it.first);
    }
  }
  return ret;
}
```

The code may appear to take $O(nk)$ time since the inner loop may take k steps (decrementing count for all k entries) and the outer loop is called n times. However each word in the sequence can be erased only once, so the total time spent erasing is $O(n)$ and the rest of the steps inside the outer loop run in $O(1)$ time.

The first step yields a set S of not more than k words; set S is a superset of the words that occur greater than or equal to $\lceil \frac{n}{k} \rceil$ times. To get the exact set, we need to

make another pass over the sequence and count the number of times each word in S actually occurs. We return the words in S which occur greater than or equal to $\lceil \frac{n}{k} \rceil$ times.

Problem 12.12, *pg. 96*: *Design a scheme for checking membership in E that minimizes the number of disk accesses. Assume that $|R| = 10^6, |E| = 10^5$, and you can use up to 1.25×10^5 bytes of RAM.*

Solution 12.12: One approach may be to use the RAM to store E. However, we have only 1.125 bytes per word, which is not enough to store E. If we keep a subset of E in RAM, we cannot decide if a word is in R by looking at the subset of E. Alternately, we could keep a subset of R in RAM. However, R is 10× the size of E, and only a very small subset of R would fit. Potentially, we could sort R in descending order of frequency of occurrence, and fit as much as possible in RAM, but this is not effective if words occur uniformly.

A better approach is to use a *Bloom filter* to over-approximate E. We use the RAM to create a bit array A consisting of $n = 10^6$ bits, initialized to 0. We then have k hash functions $h_0, \ldots, h_{k-1}$ that we apply to each word w in E, and set bits $h_0(w) \bmod n, h_1(w) \bmod n, \ldots, h_{k-1}(w) \bmod n$ in A. When we get a new word x, we examine locations $h_0(x) \bmod n, h_1(x) \bmod n, \ldots, h_{k-1}(x) \bmod n$ in A—if any location is 0, we know $x \notin E$ and we use the rules to hyphenate it.

There may be words $x \in R$ for which locations $h_0(x) \bmod n, h_1(x) \bmod n, \ldots, h_{k-1}(x) \bmod n$ are all set to 1; however, when we go to disk we'll see that $x \notin E$, so we simply use the rules.

Since most words are in R and not in E, the intent is that A can be used to eliminate those words from R. The optimum choice for k depends on $|E|$ and n, and can be computed using simulation, or through analytical methods. It can be shown that in the optimum case, half the bits end up being set.

Problem 12.13, *pg. 96*: *A pair of strings is* k-suspicious *if they have a substring of length greater than or equal to k in common. Design an efficient algorithm that takes as input a set of strings and positive integer k, and returns all pairs of strings that are k-suspicious. Assume that most pairs will not be k-suspicious.*

Solution 12.13: Let l_i be the length of the i-th string. For each string we can compute $l_i - k + 1$ hash codes, one for *each* k length substring.

We insert these hash codes in a hash table G, recording which string each code corresponds to, and the offset the corresponding substring appears at. A collision indicates that the two length-k substrings are potentially the same.

Since we are computing hash code for each k length substring, it is important for efficiency to have a hash function which can be updated incrementally when we delete one character and add another. Solution 12.1 on Page 276 describes such a hash function.

In addition, it is important to have many slots in G, so that collisions of unequal strings is rare. A total of $\sum_{i=1}^{|S|}(l_i - k + 1)$ strings are added to the hash table. If k is

small relative to the string length and G has significantly fewer slots than the total number of characters in the strings, then we are certain to have collisions.

If it is not essential to return an exact answer, we can save storage by only considering a subset of substrings, e.g., those whose hash codes have 0s in the last b bits. This means that on average we consider $\frac{1}{2^b}$ of the total set of substrings (assuming the hash function does a reasonable job of spreading keys).

As an alternative to storing all the hash codes for all the strings in a hash table, we can record the hash codes in a Bloom filter, as in Solution 12.12 on the preceding page. This results in huge storage efficiencies, since in a Bloom filter, half the bits are set to one in an optimum configuration, whereas in the hash table approach, we want to have many more slots than elements. The negative of the Bloom filter approach is that it does not tell us where the potentially equal substring is. Based on the problem size, and the number of expected matching strings, we could create Bloom filters for subsets that partition the set of all strings, and then match each string d against the subsets, essentially using the Bloom filter to eliminate subsets of strings at a time from comparison with d.

Problem 12.14, *pg. 96*: *Let A and Q be arrays of strings. Define the subarray $A[i : j]$ to* cover *Q if for all $k \in [0, |Q| - 1]$, there exists $l \in [i, j]$, $Q[k] = A[l]$. Write a function that takes two arrays A and Q and computes a minimum length subarray $A[i : j]$ that covers Q. Suppose that A is presented in streaming fashion, i.e., elements are read one at a time, and you cannot read earlier entries. The array Q is much smaller, and can be stored in RAM. How would you modify your solution for this case?*

Solution 12.14: We keep two pointers, left and right, which mark a minimal subarray of A that contains all the words in Q. A hash table H maps strings in Q that appear in the subarray to their frequency. The left and right pointers are initialized to 0. While the subarray does not contain all of Q, which is checked by looking at the size of H, we advance the right pointer. As soon as Q is covered, we record the separation of left and right, and advance left until Q is not covered. Then we advance right till it is covered again. In this way we track all minimal subarrays of A that contain Q. Assuming A is longer than Q (if it is not, we can immediately return `false`), the complexity is $O(n)$, where n is the length of A, since for each of the two pointers we do constant work per advance, and each is advanced at most $n - 1$ times.

```
pair<int, int> find_smallest_subarray_covering_subset(
    const vector<string> &A, const vector<string> &Q) {
  unordered_set<string> dict(Q.cbegin(), Q.cend());
  unordered_map<string, int> count_Q;
  int l = 0, r = 0;
  pair<int, int> res(-1, -1);
  while (r < static_cast<int>(A.size())) {
    // Keep moving r until it reaches end or count_Q has |Q| items
    while (r < static_cast<int>(A.size()) && count_Q.size() < Q.size()) {
      if (dict.find(A[r]) != dict.end()) {
        ++count_Q[A[r]];
      }
      ++r;
```

```
    }

    if (count_Q.size() == Q.size() &&
        ((res.first == -1 && res.second == -1) ||
         r - 1 - l < res.second - res.first)) {
      res = {l, r - 1};
    }

    // Keep moving l until it reaches end or count_Q has less |Q| items
    while (l < static_cast<int>(A.size()) && count_Q.size() == Q.size()) {
      if (dict.find(A[l]) != dict.end()) {
        auto it = count_Q.find(A[l]);
        if (--(it->second) == 0) {
          count_Q.erase(it);
          if ((res.first == -1 && res.second == -1) ||
              r - 1 - l < res.second - res.first) {
            res = {l, r - 1};
          }
        }
      }
      ++l;
    }
  }
  return res;
}
```

The disadvantage of this approach is that we need to keep the subarrays in memory. We can achieve a streaming algorithm by keeping track of latest occurrences of query keywords as we process A. We use a doubly linked list L to store the last occurrence (index) of each keyword in Q, and hash table H to map each keyword in Q to the corresponding node in L. Each time a word in Q is encountered, we remove its node from L (which we find by using H), create a new node which records the current index in A, and append the new node to the end of L. We also update H. By doing this, each keyword in L is ordered by its order in A; therefore, if L has $|Q|$ words (i.e., all keywords are shown) and the current index minus the index stored in the first node in L is less than current best, we update current best. The complexity is still $O(n)$.

```
pair<int, int> find_smallest_subarray_covering_subset(
    istringstream &sin, const vector<string> &Q) {
  list<int> loc;  // tracks the last occurrence (index) of each string in Q
  unordered_map<string, list<int>::iterator> dict;
  for (const string &s : Q) {
    dict.emplace(s, loc.end());
  }

  pair<int, int> res(-1, -1);
  int idx = 0;
  string s;
  while (sin >> s) {
    auto it = dict.find(s);
    if (it != dict.end()) {  // s is in Q
      if (it->second != loc.end()) {
```

```
        loc.erase(it->second);
      }
      loc.emplace_back(idx);
      it->second = --loc.end();
    }

    if (loc.size() == Q.size() &&  // found |Q| keywords
        ((res.first == -1 && res.second == -1) ||
         idx - loc.front() < res.second - res.first)) {
      res = {loc.front(), idx};
    }
    ++idx;
  }
  return res;
}
```

Variant 12.14.1: Given an array A, find a longest subarray $A[i : j]$ such that all elements in $A[i : j]$ are distinct.

Problem 12.15, *pg. 97*: *Write a function that takes two integer-valued arrays A and Q and computes a minimum length subarray $A[i : j]$ that sequentially covers Q. Assume all elements in Q are distinct.*

Solution 12.15: We solve this with a single pass over the elements of A. We maintain three data structures:

(1.) A hash map K which maps each element of Q to its index in Q, i.e., $K(Q[j]) = j$.

(2.) An array L which maps j to the index of $Q[j]$'s most recent occurrence in A.

(3.) An array D which maps j to the length of the shortest subarray of A that ends at $L[j]$ and sequentially covers subarray $Q[0 : j]$.

When processing $A[i]$, if $A[i] = Q[j]$, we set $L[j] = i$. The update for D is based on the observation that if $A[i] = Q[j]$ then the shortest subarray of A that ends at $A[i]$ and sequentially covers subarray $Q[0 : j]$ consists of the shortest subarray that sequentially covers the nearest previous $Q[j-1]$ (which is at index $L[j-1]$) together with all the indices in $[L[j-1]+1, i]$. We use K to get j from $Q[j]$. Therefore, we can write the following equation for D:

$$D[j] = i - L[j-1] + D[j-1]$$

Processing each entry of A entails no more than a constant number of lookups and updates, leading to an $O(|A|)$ time complexity. The additional space complexity is dominated by the three hash tables, i.e., $O(|Q|)$.

```
pair<int, int> find_smallest_sequentially_covering_subset(
  const vector<string> &A, const vector<string> &Q) {
  unordered_map<string, int> K;
  vector<int> L(Q.size(), -1), D(Q.size(), numeric_limits<int>::max());

  // Initialize K
  for (int i = 0; i < Q.size(); ++i) {
```

```
    K.emplace(Q[i], i);
  }

  pair<int, int> res(-1, A.size());  // default value
  for (int i = 0; i < A.size(); ++i) {
    auto it = K.find(A[i]);
    if (it != K.cend()) {
      if (it->second == 0) {  // first one, no predecessor
        D[0] = 1;  // base condition
      } else if (D[it->second - 1] != numeric_limits<int>::max()) {
        D[it->second] = i - L[it->second - 1] + D[it->second - 1];
      }
      L[it->second] = i;

      if (it->second == Q.size() - 1 &&
          D.back() < res.second - res.first + 1) {
        res = {i - D.back() + 1, i};
      }
    }
  }
  return res;
}
```

Problem 12.16, *pg. 97*: *Implement a cache for looking up prices of books identified by their ISBN. Use the Least Recently Used (LRU) strategy for cache eviction policy.*

Solution 12.16: We use a hash table to quickly lookup price. Keys are ISBNs. Along with each key, we store the price and the most recent time a lookup was done on that key. However it takes $O(n)$ time to find the LRU item in a hash table with this scheme, where n is the number of entries in the hash table.

One way to improve performance is to use lazy garbage collection, which amortizes the cost of removing the LRU ISBNs. To be concrete, let's say we want the cache to be of size n. We do not delete any entries from the hash table until it grows to $2n$ entries. At this point we iterate through the entire hash table, and find the median age of items. Subsequently we discard everything below the median. The worst-case time to delete becomes $\Theta(n)$ but it will happen at most once every n operations. Therefore the amortized cost of deletion is $O(1)$ at the cost of doubling the memory consumption.

An alternative is to maintain a separate queue of keys. In the hash table we store for each key a reference to its location in the queue. Each time an ISBN is looked up that is found in the hash table, it is moved to the front of the queue. (This requires us to use a linked list implementation of the queue, so that items in the middle of the queue can be moved to the head.) When the queue exceeds length n, each time a lookup is performed that is not found in the hash table, the result is placed in the hash table and at the head of the queue; the item at the tail of the queue is deleted from the queue and the hash table.

```
template <typename ISBNType, typename PriceType, size_t capacity>
class LRUCache {
```

```
  private:
    typedef unordered_map<
        ISBNType, pair<typename list<ISBNType>::iterator, PriceType>> Table;
    Table cache;
    list<ISBNType> data;

    // Move the most recent accessed item to the front
    void moveToFront(const ISBNType &isbn,
                     const typename Table::iterator &it) {
      data.erase(it->second.first);
      data.emplace_front(isbn);
      it->second.first = data.begin();
    }

  public:
    const bool lookup(const ISBNType &isbn, PriceType* price) {
      auto it = cache.find(isbn);
      if (it == cache.end()) {
        return false;
      }

      *price = it->second.second;
      moveToFront(isbn, it);
      return true;
    }

    void insert(const ISBNType &isbn, const PriceType &price) {
      auto it = cache.find(isbn);
      if (it != cache.end()) {
        moveToFront(isbn, it);
      } else {
        // Remove the least recently used
        if (cache.size() == capacity) {
          cache.erase(data.back());
          data.pop_back();
        }

        data.emplace_front(isbn);
        cache.emplace(isbn, data.begin(), price);
      }
    }

    const bool erase(const ISBNType &isbn) {
      auto it = cache.find(isbn);
      if (it == cache.end()) {
        return false;
      }

      data.erase(it->second.first);
      cache.erase(it);
      return true;
    }
};
```

Problem 13.1, ***pg. 98***: *What is the most efficient sorting algorithm for each of the following situations:*

- *A large array whose entries are random numbers.*
- *A small array of numbers.*
- *A large array of numbers that is already almost sorted.*
- *A large collection of integers that are drawn from a small range.*
- *A large collection of numbers most of which are duplicates.*
- *Stability is required, i.e., the relative order of two records that have the same sorting key should not be changed.*

Solution 13.1: In general, quicksort is considered one of the most efficient sorting algorithms since it has an average case run time of $\Theta(n \log n)$ and it sorts in-place (sorted data are not copied to some other buffer). For a large set of random integers, quicksort would be our choice.

Quicksort is more nuanced than appears at a first glance. For example, in a naïve implementation, an array with many duplicate elements leads to quadratic run times (and a high likelihood of stack space being exhausted because of the number of recursive calls). This can be managed by putting all keys equal to the pivot in the correct place. Similarly, it is important to call the smaller subproblem first—this, in conjunction with tail recursion ensures that the stack depth is $O(\log n)$.

However there are cases where other sorting algorithms are preferable.

- Small set—for a small set (for example, 3–10 numbers), a simple implementation such as insertion sort is easier to code, and runs faster in practice.
- Almost sorted array—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 (Solution 10.6 on Page 253).
- Integers from a small range, or a few distinct keys—counting sort, which records for each element, the number of elements less than it. 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.
- Many duplicates—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 walk of the BST
- Stability is required—most useful 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.

Problem 13.2, ***pg. 99***: *Sort lines of a text file that has one million lines such that the average length of a line is* 100 *characters but the longest line is one million characters long.*

Solution 13.2: Almost all sorting algorithms rely on swapping records. However this becomes complicated when the record size varies. One way of dealing with this problem is to allocate for the maximum possible size for each record—this can be wasteful if there is a large variation in the sizes.

The better solution is *indirect sort*. First, build an array P of pointers to the records. Then sort the pointers using the compare function on the dereferenced pointers. Finally, iterate through P writing the dereferenced pointers.

```
template <typename T>
void indirect_sort(const string &file_name) {
  // Store file records into A
  ifstream ifs(file_name.c_str());
  vector<T> A;
  T x;
  while (ifs >> x) {
    A.emplace_back(x);
  }

  // Initialize P
  vector<T*> P;
  for (T &a : A) {
    P.emplace_back(&a);
  }

  // Indirect sort file
  sort(P.begin(), P.end(), [](const T* a, const T* b) -> bool {
                             return *a < *b;
                           });

  // Output file
  ofstream ofs(file_name.c_str());
  for (const T* p : P) {
    ofs << p << endl;
  }
}
```

Problem 13.3, ***pg. 99***: *Design a sorting algorithm that minimizes the total distance that items are moved.*

Solution 13.3: Whenever the swap operation for the objects being sorted is expensive, one of the best things to do is indirect sort, i.e., sort references to the objects first and then apply the permutation that was applied to the references in the end.

In the case of statues, we can assign increasing indices to the statues from left-to-right and then sort the pairs of statue height and index. The indices in the sorted pairs would give us the permutation to apply. While applying permutation, we would want to perform it in a way that we move each statue the minimum possible distance. We can achieve this if each statue is moved exactly to its correct destination exactly once (no intermediate swaps).

Problem 13.4, ***pg. 99***: *You are given an array of n* `Person` *objects. Each* `Person` *object has a field* `key`*. Rearrange the elements of the array so that* `Person` *objects with equal keys appear together. The order in which distinct keys appear is not important. Your algorithm must run in $O(n)$ time and $O(k)$ additional space. How would your solution change if keys have to appear in sorted order?*

Solution 13.4: We use the approach described in the introduction to the problem. However, we cannot apply it directly, since we need to write objects, not integers—two objects may have the same key but other fields may be different.

We use a hash table C to count the number of distinct occurrences of each key. We iterate over each key k in C and keep a cumulative count s which is the starting offset in the array where elements with key k are to be placed. We put the key-value pair (k, s) in a hash table M—basically M partitions the array into the subarrays holding objects with equal keys.

We then iteratively get a key k from M and swap the element e at k's current offset (which we get from M) with the location appropriate for e's key *e.key* (which we also get from M). Since e is now in its correct location, we update M by advancing the offset corresponding to *e.key*, taking care to remove *e.key* from M when all elements with key equal to *e.key* are correctly placed.

The time complexity is $O(n)$, since the first pass entails n hash table inserts, and the second pass performs a constant amount of work to move one element to the right location. (Selecting an arbitrary key from a hash table is a constant time operation.) The additional space complexity dictated by C and M, and is $O(k)$, where k is the number of distinct keys.

If the objects are additionally required to appear in sorted key order, we can store M using a BST-based map instead of a hash table. The time complexity becomes $O(n + k \log k)$, since BST insertion takes time $O(\log k)$. This should make sense, since if $k = n$, we are doing a regular sort, which is known to be $O(n \log n)$ for sorting based on comparisons.

```
template <typename KeyType>
void counting_sort(vector<Person<KeyType>> &people) {
  unordered_map<KeyType, int> key_to_count;
  for (const Person<KeyType> &p : people) {
    ++key_to_count[p.key_];
  }
  unordered_map<KeyType, int> key_to_offset;
  int offset = 0;
  for (const auto p : key_to_count) {
    key_to_offset[p.first] = offset;
    offset += p.second;
  }

  while (key_to_offset.size()) {
    auto from = key_to_offset.begin();
    auto to = key_to_offset.find(people[from->second].key_);
    swap(people[from->second], people[to->second]);
    // Use key_to_count to see when we are finished with a particular key
    if (--key_to_count[to->first]) {
      ++to->second;
    } else {
      key_to_offset.erase(to);
    }
  }
}
```

Problem 13.5, ***pg. 99*****:** *Given sorted arrays A and B of lengths n and m respectively, return an array C containing elements common to A and B. The array C should be free of duplicates. How would you perform this intersection if—(1.) $n \approx m$ and (2.) $n \ll m$?*

Solution 13.5: The simplest algorithm is a "loop join", i.e., walking through all the elements of one array and comparing them to the elements of the other array. This has $O(mn)$ time complexity, regardless of whether the arrays are sorted or unsorted:

```
template <typename T>
vector<T> intersect_arrs1(const vector<T> &A, const vector<T> &B) {
  vector<T> intersect;
  for (int i = 0; i < A.size(); ++i) {
    if (i == 0 || A[i] != A[i - 1]) {
      for (int j = 0; j < B.size(); ++j) {
        if (A[i] == B[j]) {
          intersect.emplace_back(A[i]);
          break;
        }
      }
    }
  }
  return intersect;
}
```

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

```
template <typename T>
vector<T> intersect_arrs2(const vector<T> &A, const vector<T> &B) {
  vector<T> intersect;
  for (int i = 0; i < A.size(); ++i) {
    if ((i == 0 || A[i] != A[i - 1]) &&
        binary_search(B.cbegin(), B.cend(), A[i])) {
      intersect.emplace_back(A[i]);
    }
  }
  return intersect;
}
```

Now our algorithm time complexity is $O(n \log m)$. We can further improve our run time by choosing the longer array for the inner loop since if $n \ll m$ then $m \log(n) \gg n \log(m)$.

This is the best solution if one set is much smaller than the other. However it is not optimal for cases where the set sizes are similar because we are not using the fact that both arrays are sorted to our advantage. In that case, iterating in tandem through the elements of each array in increasing order will work best as shown in this C++ code:

```
template <typename T>
vector<T> intersect_arrs3(const vector<T> &A, const vector<T> &B) {
  vector<T> intersect;
  int i = 0, j = 0;
```

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

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

Problem 13.6, *pg. 100*: *Design an algorithm that takes as input two teams and the heights of the players in the teams and checks if it is possible to place players to take the photo subject to the placement constraint.*

Solution 13.6: Let A and B be arrays. Write $A < B$ if $A[i] < B[i]$ for each i. Let $\langle x_0, x_1, \ldots, x_{n-1} \rangle$ be an array of the heights of the players in Team X and $\langle y_0, y_1, \ldots, y_{n-1} \rangle$ be an array of the heights of the players in Team Y. By the transitivity of $<$, Team X can be placed in front of Team Y iff $\text{sort}\langle x_0, \ldots, x_{n-1} \rangle < \text{sort}\langle y_0, \ldots, y_{n-1} \rangle$, so the problem reduces to sorting. Figure 21.9 shows the teams in Figure 13.1 on Page 100 sorted by their heights.

Figure 21.9: The teams from Figure 13.1 on Page 100 in sorted order.

```
template <typename HeightType>
class Player {
  public:
    HeightType height;

    const bool operator<(const Player &that) const {
      return height < that.height;
    }
};

template <typename HeightType>
class Team {
  private:
    vector<Player<HeightType>> members;

    vector<Player<HeightType>> sortHeightMembers(void) const {
      vector<Player<HeightType>> sorted_members(members);
      sort(sorted_members.begin(), sorted_members.end());
      return sorted_members;
    }
```

```
21
22   public:
23     const bool operator<(const Team &that) const {
24       vector<Player<HeightType>> this_sorted(sortHeightMembers());
25       vector<Player<HeightType>> that_sorted(that.sortHeightMembers());
26       for (int i = 0; i < this_sorted.size() && i < that_sorted.size(); ++i) {
27         if (this_sorted[i] < that_sorted[i] == false) {
28           return false;
29         }
30       }
31       return true;
32     }
33 };
```

Problem 13.7, *pg.100*: *Given a string s, print in alphabetical order each character that appears in s, and the number of times that it appears. For example, if s ="bcdacebe", output "(a, 1), (b, 2), (c, 2), (d, 1), (e, 2)"..*

Solution 13.7: Many solutions exist for this problem. In the code below, we treat the string as an array of characters and sort that array; consequently, we iterate through the sorted array and count the number of occurrences of each character.

```
1  void count_occurrences(string S) {
2    sort(S.begin(), S.end());
3
4    int count = 1;
5    for (int i = 1; i < S.size(); ++i) {
6      if (S[i] == S[i - 1]) {
7        ++count;
8      } else {
9        cout << '(' << S[i - 1] << ',' << count << "),";
10       count = 1;
11     }
12   }
13   cout << '(' << S.back() << ',' << count << ')' << endl;;
14 }
```

As an alternative, we could use an auxiliary array of integers indexed by characters, which counts the number of occurrences of each character. The sort routine itself can be based on radix sort, since the elements can be viewed as integers taking values in $[0, M-1]$ for a relatively small M. (We could also use a BST or a hash table in which keys are characters, and values are counts.)

Variant 13.7.1: The array A is an array of person objects. A person object has an age field, which is an integer in the range $[0, 150]$, and a name field, which is a string. The array is to be sorted on the age field and the sort must be stable. Design an $O(n)$ time algorithm for sorting A. Is it possible to sort in $O(n)$ time if ties are to be broken on the name field?

Problem 13.8, *pg. 100*: *Design an efficient algorithm for removing all the duplicates from an array.*

Solution 13.8: An efficient way of eliminating duplicates from any set of records, where a "less-than" operator can be defined, is to sort the records and then eliminate the duplicates in a single pass over the data.

Sorting can be done in $O(n \log n)$ time; the subsequent elimination of duplicates takes $\Theta(n)$ time. If the elimination of duplicates is done in-place, it would be more efficient than writing the unique set in a separate array since we would achieve better cache performance. Here is the code that does in-place duplicate removal:

```
template <typename T>
int eliminate_duplicate(vector<T> &A) {
  sort(A.begin(), A.end());  // makes identical elements become neighbors
  auto it = unique(A.begin(), A.end());  // removes neighboring duplicates
  A.resize(it - A.cbegin());  // truncates the unnecessary trailing part
  return it - A.cbegin();
}
```

Another efficient way is to use hash table where we store each record into a hash table as the key with no value and then write out all the keys in the hash table. Since hash table inserts can be done in $O(1)$ time and iterating over all the keys takes $O(n)$ time, this solution has much better time complexity than the sorting approach. However, in practice for small inputs, the sorting approach will likely be faster since it can be done in-place.

A counting sort (Solution 13.4 on Page 293 is appropriate if the number of distinct elements is small and the array is very large.

e-Variant 13.8.1: Given an array A with possible duplicate entries, find the k entries that occur most frequently.

Problem 13.9, *pg. 101*: *Design an efficient algorithm that takes as input an array A of even length and computes a 2-partition of A that has minimum $Q(\Pi)$.*

Solution 13.9: Consider the task l that has the longest duration, i.e., $A[l] \geq A[i]$ for all $i \neq l$. Task l must be assigned to some worker w, and that worker must do some other additional Task s. The fastest w can complete both tasks is $A[l] + A[s]$, and this is minimized for sum_w such that $A[s]$ is a minimum element of A.

We now claim that there is always an optimum assignment in which the longest duration Task l and shortest duration Task s are assigned to the same worker.

Proof:

Let α be any assignment in which l and s are not paired. Suppose α pairs l with s' and s with l', where by supposition $A[l] \geq A[l']$ and $A[s] \leq A[s']$. Consider the assignment β where l and s are paired and l' and s' are paired, with the remaining pairings unchanged from α. Observe $A[l] + A[s'] \geq A[l] + A[s]$ due to $A[s'] \geq A[s]$. Since $A[l] \geq A[l']$ we have $A[l] + A[s'] \geq A[l'] + A[s']$, which implies that the pairing of l' and s' in β is still better than the maximum delay pairing in α (which must

have duration at least $A[l] + A[s']$ or $A[l'] + A[s]$).

Note that we are not claiming that the time taken for the optimum assignment is $A[l]+A[s]$. Indeed this may not even be the case, e.g., if the two longest duration tasks are close to each other in duration and the shortest duration task takes much less time than the second shortest task. As a concrete example consider $A = \langle 1, 8, 9, 10 \rangle$ where maximum delay is $8 + 9 = 17$.

```
template <typename T>
vector<pair<T, T>> task_assignment(vector<T> A) {
  sort(A.begin(), A.end());
  vector<pair<T, T>> P;
  for (int i = 0, j = A.size() - 1; i < j; ++i, --j) {
    P.emplace_back(A[i], A[j]);
  }
  return P;
}
```

Problem 13.10, ***pg. 101***: *Given a set of events, how would you determine the maximum number of events that take place concurrently?*

Solution 13.10: Each event corresponds to an interval $[b, e]$; let b and e be the earliest starting time and last ending time. Define the function $c(t)$ for $t \in [b, e]$ to be the number of intervals containing t. Observe that $c(\tau)$ does not change if τ is not the starting or ending time of an event.

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

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

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

```
template <typename TimeType>
class Interval {
  public:
    TimeType start, finish;
};

template <typename TimeType>
class Endpoint {
  public:
```

```
    TimeType time;
    bool isStart;

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

template <typename TimeType>
int find_max_concurrent_events(const vector<Interval<TimeType>> &A) {
  // Build the endpoint array
  vector<Endpoint<TimeType>> E;
  for (const Interval<TimeType> &i : A) {
    E.emplace_back(Endpoint<TimeType>{i.start, true});
    E.emplace_back(Endpoint<TimeType>{i.finish, false});
  }
  // Sort the endpoint array according to the time
  sort(E.begin(), E.end());

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

ϵ-Variant 13.10.1: Users $1, 2, \ldots, n$ share an Internet connection. User i uses b_i bandwidth from time s_i to f_i, inclusive. What is the peak bandwidth usage?

Problem 13.11, *pg. 102*: *Design an algorithm that takes as input a set of intervals I, and outputs the union of the intervals. What is the time complexity of your algorithm as a function of the number of intervals?*

Solution 13.11: We begin with by sorting the intervals I on their left endpoints. If left endpoints a and b are equal, with a corresponding to a closed interval and b to an open interval, a comes first; otherwise, we break ties arbitrarily.

Let the sorted sequence be $\langle I_0, I_1, \ldots, I_{n-1} \rangle$. We create the result $\langle R_0, R_1, \ldots, R_m \rangle$ where $m \leq n$ by processing intervals in order; the R_is will be sorted by their left endpoints. Let t and s be interval-valued variables initialized to I_0, and I_1, respectively; we will show how to extend t to R_0. Let the left and right endpoints of $t(s)$ be $t.l(s.l)$ and $t.r(s.r)$, respectively. We have the following cases:

- $(s.l > t.r)$: R_0 is t, since no later interval can overlap or be adjacent to t.
- $(s.l = t.r)$ and (s is left open and t is right open): we set R_0 to t, since s and t cannot be merged and no later interval can overlap or be adjacent to t.

– ($s.l < t.r$) or ($s.l = t.r$ and (s is left-closed or t is right-closed)): if $s.r > t.r$ or ($s.r = t.r$ and s is right-closed) we extend t's right endpoint to $s.r$, and t is right open iff s is right open. We assign s to the next unprocessed interval and continue.

The code below implements this case analysis iteratively:

```
template <typename TimeType>
class Interval {
  private:
    class Endpoint {
      public:
        bool isClose;
        TimeType val;
    };

  public:
    Endpoint left, right;

    const bool operator<(const Interval &i) const {
      return left.val != i.left.val ?
             left.val < i.left.val : (left.isClose && !i.left.isClose);
    }
};

template <typename TimeType>
vector<Interval<TimeType>> Union_intervals(vector<Interval<TimeType>> I) {
  // Empty input
  if (I.empty()) {
    return {};
  }

  // Sort intervals according to their left endpoints
  sort(I.begin(), I.end());
  Interval<TimeType> curr(I.front());
  vector<Interval<TimeType>> uni;
  for (int i = 1; i < I.size(); ++i) {
    if (I[i].left.val < curr.right.val ||
        (I[i].left.val == curr.right.val &&
         (I[i].left.isClose || curr.right.isClose))) {
      if (I[i].right.val > curr.right.val ||
          (I[i].right.val == curr.right.val && I[i].right.isClose)) {
        curr.right = I[i].right;
      }
    } else {
      uni.emplace_back(curr);
      curr = I[i];
    }
  }
  uni.emplace_back(curr);
  return uni;
}
```

Problem 13.12, *pg. 102*: *You are given a set of n tasks modeled as closed intervals $[a_i, b_i]$, for*

$i = 0, \ldots, n-1$. *A set S of visit times* covers *the tasks if* $[a_i, b_i] \cap S \neq \emptyset$, *for* $i = 0, \ldots, n-1$. *Design an efficient algorithm for finding a minimum cardinality set of visit times that covers all the tasks.*

Solution 13.12: A covering set S must contain at least one point x such that $x \leq b_{min} = \min_{i=0}^{n-1} b_i$. Any such point covers the subset of intervals $[a_i, b_i], a_i \leq b_{min}$. Of course, b_{min} itself covers all such intervals and so there exists a minimum cardinality covering that contains b_{min} and no other points to its left. The same principle can be applied to the remaining intervals.

```
template <typename TimeType>
class Interval {
  public:
    TimeType left, right;
};

template <typename TimeType>
class LeftComp {
  public:
    const bool operator()(const Interval<TimeType> &a,
                          const Interval<TimeType> &b) const {
      return a.left != b.left ? a.left < b.left : a.right < b.right;
    }
};

template <typename TimeType>
class RightComp {
  public:
    const bool operator()(const Interval<TimeType> &a,
                          const Interval<TimeType> &b) const {
      return a.right != b.right ? a.right < b.right : a.left < b.left;
    }
};

template <typename TimeType>
vector<TimeType> find_minimum_visits(const vector<Interval<TimeType>> &I) {
  set<Interval<TimeType>, LeftComp<TimeType>> L;
  set<Interval<TimeType>, RightComp<TimeType>> R;
  for (const Interval<TimeType> &i : I) {
    L.emplace(i), R.emplace(i);
  }

  vector<TimeType> S;
  while (L.size() && R.size()) {
    TimeType b = R.cbegin()->right;
    S.emplace_back(R.cbegin()->right);

    // Remove the intervals which intersect with R.cbegin()
    auto it = L.cbegin();
    while (it != L.end() && it->left <= b) {
      R.erase(*it);
      L.erase(it++);
    }
  }
```

```
45   return S;
46 }
```

Using a balanced BST (e.g., set in C++), we can implement the search for minimum, insertion, and deletion in $O(\log n)$ time, yielding an $O(n \log n)$ algorithm.

Problem 13.13, *pg. 102*: *Let $[\theta_i, \phi_i]$, for $i = 0, \ldots, n-1$ be n arcs, where the i-th arc is the set of points on the perimeter of the unit circle that subtend an angle in the interval $[\theta_i, \phi_i]$ at the center. A ray is a set of points that all subtend the same angle to the origin, and is identified by the angle they make relative to the x-axis. A set R of rays "covers" the arcs if $[\theta_i, \phi_i] \cap R \neq \emptyset$, for $i = 0, \ldots, n-1$. Design an efficient algorithm for finding a minimum cardinality set of rays that covers all arcs.*

Solution 13.13: If there exists a point on the circle that is not contained in at least one of the n arcs, the problem is identical to Problem 13.12 on Page 102. Therefore we assume every point is contained in one of the n arcs.

Without loss of generality, we may assume that a minimum cardinality covering set S contains only right endpoints of arcs, i.e., "clockwise" right endpoints. The total number of such endpoints is n. If we choose a given right endpoint and eliminate all the arcs that are covered by it, the remaining problem is identical to that in Problem 13.12 on Page 102. This means we can solve the arc-covering problem by n calls to the algorithm in Solution 13.12 on the facing page, yielding an $O(n^2 \log n)$ algorithm.

The approach of solving a problem involving a circular array by solving a number of instances of the same problem on linear arrays is fairly common—see for example Problem 15.5 on Page 118.

Problem 13.14, *pg. 103*: *Design an algorithm that takes as input an array A and a number t, and determines if A 3-creates t.*

Solution 13.14: We consider the case where $k = 2$ and A is sorted in Problem 11.4 on Page 87. Therefore, one solution is to 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 in Solution 11.4 on Page 261 n times, which is $O(n^2)$ overall. The code for this approach is shown below.

```
1  template <typename T>
2  bool has_2_sum(const vector<T> &A, const T &t) {
3    int j = 0, k = A.size() - 1;
4
5    while (j <= k) {
6      if (A[j] + A[k] == t) {
7        return true;
8      } else if (A[j] + A[k] < t) {
9        ++j;
10     } else {  // A[j] + A[k] > t
11       --k;
12     }
```

```
13    }
14    return false;
15  }
16
17  template <typename T>
18  bool has_3_sum(vector<T> A, const T &t) {
19    sort(A.begin(), A.end());
20
21    for (const T &a : A) {
22      // Find if the sum of two numbers in A equals to t - a
23      if (has_2_sum(A, t - a)) {
24        return true;
25      }
26    }
27    return false;
28  }
```

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

ϵ-Variant 13.14.1: Solve the same problem when the three elements must be distinct. For example, if $A = [5, 2, 3, 4, 3]$ 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 13.14.2: Solve the same problem when k is an additional input.

Problem 13.15, *pg. 103*: *Develop an algorithm for computing a short sequence of flips that will sort an array A.*

Solution 13.15: The most straightforward pancake sorting algorithm is analogous to insertion sort. Observe that we can move any pancake to any position with at most two flips: the first flip brings it to the top, the second flip moves it to the desired location. We can iteratively apply this idea to move the largest pancake to the bottom, followed by moving the second largest pancake to just above the largest pancake, etc. Note that the flips do not affect the positions of the pancakes that have been sorted by previous flips. Since we can put at least one pancake in the right place with two flips, this algorithm uses at most $2n$ flips. We can tighten this bound by observing that a stack of two pancakes requires at most one flip, so the number of flips made by the algorithm is actually $2(n-2) + 1 = 2n - 3$.

Note that we are trying to minimize the number of flips, not the time complexity of the algorithm that computes the sequence of flips. The run time of the algorithm is $O(n^2)$, since reversal takes $O(n)$ and finding the maximum element in an array also takes $O(n)$, and each of these operations could be performed once per pancake.

Remark: It is known that in the worst case $\frac{15}{14}n$ flips are necessary and $\frac{18}{11}n$ flips are sufficient. In other words, the straightforward algorithm does not always yield the optimum result. The first improvement was made by Bill Gates, when he was a math undergraduate at Harvard, in collaboration with Christos Papadimitriou.

Problem 14.1, *pg. 104*: *Write a function that takes as input the root of a binary tree whose nodes have a key field, and returns* `true` *iff the tree satisfies the BST property.*

Solution 14.1: Several solutions exist, which differ in terms of their space and time complexity, and the effort needed to code them.

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

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

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

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

The first, more straightforward approach, is to check constraints on the values for each subtree. The initial constraint comes from the root. Each node in its left (right) child must have a value less than or equal (greater than or equal) to the value at the root. This idea generalizes: if all nodes in a tree rooted at t must have values in the range $[l, u]$, and the value at t is $w \in [l, u]$, then all values in the left subtree of t must be in the range $[l, w]$, and all values stored in the right subtree of t must be in the range $[w, u]$. The code below uses this approach.

```
1 template <typename T>
2 bool is_BST_helper(const shared_ptr<BinaryTree<T>> &r, const T &lower,
3                     const T &upper) {
4   if (!r) {
5     return true;
6   } else if (r->data < lower || r->data > upper) {
7     return false;
8   }
```

```
9
10    return is_BST_helper(r->left, lower, r->data) &&
11           is_BST_helper(r->right, r->data, upper);
12  }
13
14  template <typename T>
15  bool is_BST(const shared_ptr<BinaryTree<T>> &r) {
16    return is_BST_helper(r, numeric_limits<T>::min(), numeric_limits<T>::max());
17  }
```

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

```
1  template <typename T>
2  bool is_BST(shared_ptr<BinaryTree<T>> n) {
3    // Store the value of previous visited node
4    int last = numeric_limits<T>::min();
5    bool res = true;
6
7    while (n) {
8      if (n->left) {
9        // Find the predecessor of n
10       shared_ptr<BinaryTree<T>> pre = n->left;
11       while (pre->right && pre->right != n) {
12         pre = pre->right;
13       }
14
15       // Build the successor link
16       if (pre->right) {  // pre->right == n
17         // Revert the successor link if predecessor's successor is n
18         pre->right = nullptr;
19         if (last > n->data) {
20           res = false;
21         }
22         last = n->data;
23         n = n->right;
24       } else {  // if predecessor's successor is not n
25         pre->right = n;
26         n = n->left;
27       }
28     } else {
29       if (last > n->data) {
30         res = false;
31       }
32       last = n->data;
33       n = n->right;
34     }
35   }
36   return res;
37 }
```

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

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

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

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

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

```
template <typename T>
class QNode {
  public:
    shared_ptr<BinaryTree<T>> node;
    T lower, upper;
};

template <typename T>
bool is_BST(const shared_ptr<BinaryTree<T>> &n) {
  queue<QNode<T>> q;

  q.emplace(QNode<T>{n, numeric_limits<T>::min(), numeric_limits<T>::max()});
  while (!q.empty()) {
    if (q.front().node) {
      if (q.front().node->data < q.front().lower ||
          q.front().node->data > q.front().upper) {
        return false;
      }

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

```
28 }
```

Problem 14.2, *pg. 105*: *Given a node x, find the successor of x in a BST. Assume that nodes have `parent` fields, and the `parent` field of root points to `null`.*

Solution 14.2: If there is a right subtree from the given node, we return the smallest node in the right subtree. If the node does not have a right child, then we need to keep going up the tree till we find a node which is the left child of its parent, in which case that parent is the desired successor. If we reach the root then the given node is the largest node in the tree and has no successor.

```
1  template <typename T>
2  shared_ptr<BinarySearchTree<T>> find_successor_BST(
3    shared_ptr<BinarySearchTree<T>> n) {
4    if (n->right) {
5      // Find the smallest element in n's right subtree
6      n = n->right;
7      while (n->left) {
8        n = n->left;
9      }
10     return n;
11   }
12
13   // Find the first parent which is larger than n
14   while (n->parent && n->parent->right == n) {
15     n = n->parent;
16   }
17   // Return nullptr means n is the largest in this BST
18   return n->parent;
19 }
```

Problem 14.3, *pg. 105*: *Design efficient functions for inserting and removing keys in a BST. Assume that all elements in the BST are unique, and that your insertion method must preserve this property. You cannot change the contents of any node. What are the time complexities of your functions?*

Solution 14.3: Insertion is done as follows. We start by creating a new node c holding k. Observe that inserting a key k into a tree that is empty is trivial: we set c to be the root of the tree. If the tree is nonempty and k is greater than the key r stored at the root, we need to insert k into the root's right subtree. If r and k are equal, we return, since we do not want add a duplicate. Otherwise we insert into the root's left subtree. At some step, we will either return because we found a node containing a key equal to k, or encounter an empty tree. For this case, we need to set the appropriate child of the last node whose key we compared k with to c.

Deletion begins with first identifying the node d containing k. (If no such node exists, nothing needs to be done.) Suppose d has no right child and no left child. In this case, there is nothing to be done, beyond updating the appropriate child field in d's parent to `null`. Otherwise, if d has a right child, we find its successor, call it s,

which must lie in d's right subtree. We "substitute" d with s—this entails updating d's parent to point to s instead of d, and setting the children of s to the children of d. There are a couple of corner cases: if d has no parent, the parent update operation is skipped; if s is the right child of d, then the right child of s is set to `null`. If d does not have a right child, all we do is update d's parent to point to s. There is one corner case: if d has no parent, i.e., it is the root, the root is updated to s.

Both insertion and deletion times are dominated by the time taken to search for a key and to find the minimum element in a subtree. Both of these times are proportional to the height of the tree. In the worst case, the BST can grow to be very skewed. For example, if the initial tree is empty, and n successive insertions are done, where each key inserted is larger than the previous one, the height of the resulting tree is n. It is possible to modify the insertion and deletion routines to keep the tree height $O(\log n)$, where n is the number of nodes; furthermore, these insertion and deletion functions continue to have time complexity proportional to the height. AVL and red-black trees are BSTs with additional state related to height imbalance stored at each node. Specialized insertion and deletion operations do "rotations" about nodes to keep the height logarithmic in the number of nodes.

```
template <typename T>
class BinarySearchTree {
  private:
    class TreeNode {
      public:
        T data;
        shared_ptr<TreeNode> left, right;
    };

    void clear(shared_ptr<TreeNode> &n) {
      if (n) {
        clear(n->left), clear(n->right);
        n = nullptr;
      }
    }

    // Replace the link between par and child by new_link
    void replaceParentChildLink(shared_ptr<TreeNode> par,
                                shared_ptr<TreeNode> child,
                                shared_ptr<TreeNode> new_link) {
      if (!par) {
        return;
      }

      if (par->left == child) {
        par->left = new_link;
      } else {
        par->right = new_link;
      }
    }

    shared_ptr<TreeNode> root;

```

```
34  public:
35    BinarySearchTree(void) : root(nullptr) {}
36
37    ~BinarySearchTree(void) {
38      clear();
39    }
40
41    const bool empty(void) const {
42      return !root;
43    }
44
45    void clear(void) {
46      clear(root);
47    }
48
49    const bool insert(const T &key) {
50      shared_ptr<TreeNode>
51        t = shared_ptr<TreeNode>(new TreeNode{key, nullptr, nullptr}),
52        par = nullptr;
53
54      if (empty()) {
55        root = t;
56      } else {
57        shared_ptr<TreeNode> curr;
58        curr = root;
59        while (curr) {
60          par = curr;
61          if (t->data == curr->data) {
62            t = nullptr;
63            return false;  // no insertion for duplicate key
64          } else if (t->data < curr->data) {
65            curr = curr->left;
66          } else {  // t->data > curr->data
67            curr = curr->right;
68          }
69        }
70
71        // Insert key according to key and par
72        if (t->data < par->data) {
73          par->left = t;
74        } else {
75          par->right = t;
76        }
77      }
78      return true;
79    }
80
81    const bool erase(const T &key) {
82      // Find the node with key
83      shared_ptr<TreeNode> curr = root, par = nullptr;
84      while (curr && curr->data != key) {
85        par = curr;
86        curr = key < curr->data ? curr->left : curr->right;
87      }
88
```

```
89        // No node with key in this binary tree
90        if (!curr) {
91          return false;
92        }
93
94        if (curr->right) {
95          // Find the minimum of the right subtree
96          shared_ptr<TreeNode> r_curr = curr->right, r_par = curr;
97          while (r_curr->left) {
98            r_par = r_curr;
99            r_curr = r_curr->left;
100         }
101         // Move links to erase the node
102         replaceParentChildLink(par, curr, r_curr);
103         replaceParentChildLink(r_par, r_curr, r_curr->right);
104         r_curr->left = curr->left, r_curr->right = curr->right;
105
106         // Update root link if needed
107         if (root == curr) {
108           root = r_curr;
109         }
110       } else {
111         // Update root link if needed
112         if (root == curr) {
113           root = curr->left;
114         }
115         replaceParentChildLink(par, curr, curr->left);
116       }
117       curr = nullptr;
118       return true;
119     }
120 };
```

Problem 14.4, *pg. 105*: *Given a BST T, write recursive and iterative versions of a function that takes a BST T, a key k, and returns the node containing k that would appear first in an inorder walk. If k is absent, return* **null**. *For example, when applied to the BST in Figure 14.2 on Page 106, your algorithm should return Node B if* $k = 108$, *Node G if* $k = 285$, *and* **null** *if* $k = 143$.

Solution 14.4: The standard way to search for a key k in a BST is to first check if the tree is empty; if so, we return `null`. Consequently check if the root stores k, in which case we return the root, and recur on the left/right child of the root, if k is less/greater than the value at the root.

For the problem we are given, we need to find the *first* occurrence of k that would appear in an inorder walk. We achieve this by making a simple modification to the standard search—if k is matched by the root, we also check to see if it appears in the root's left subtree, in which case we return the node returned by that call. Correctness follows from the fact that the nodes in the root's left subtree all appear before the root in an inorder walk.

```
1 template <typename T>
```

```
shared_ptr<BinarySearchTree<T>> find_first_equal_k(
    const shared_ptr<BinarySearchTree<T>> &r, const T &k) {
  if (!r) {
    return nullptr;  // no match
  } else if (r->data == k) {
    // Recursively search the left subtree for first one == k
    shared_ptr<BinarySearchTree<T>> n = find_first_equal_k(r->left, k);
    return n ? n : r;
  }
  // Search left or right tree according to r->data and k
  return find_first_equal_k(r->data < k ? r->right : r->left, k);
}
```

The straightforward implementation of the standard BST search is tail recursive, and can mechanically be converted to iterative code. However, the recursive version for finding the first occurrence of k is not tail recursive, and needs to be written from scratch. The iterative code below makes use of the elimination principle.

```
template <typename T>
shared_ptr<BinarySearchTree<T>> find_first_equal_k(
    shared_ptr<BinarySearchTree<T>> r, const T &k) {
  shared_ptr<BinarySearchTree<T>> first = nullptr;
  while (r) {
    if (r->data < k) {
      r = r->right;
    } else if (r->data > k) {
      r = r->left;
    } else {  // r->data == k
      // Search for the leftmost in the left subtree
      first = r;
      r = r->left;
    }
  }
  return first;
}
```

Problem 14.5, *pg. 106*: *Write a function that takes a BST T and a key k, and returns the first entry larger than k that would appear in an inorder walk. If k is absent or no key larger than k is present, return* `null`. *For example, when applied to the BST in Figure 14.1 on Page 105 you should return 29 if $k = 23$; if $k = 32$, you should return* `null`.

Solution 14.5: This problem can be solved using a reductionist approach. First we find the node n holding k that appears last in an inorder walk of the tree, and then we find n's successor. The node n can be found using a straightforward modification of Solution 14.4 on the preceding page. If n does not exist (k is not the value stored at any node in the tree) or n has no successor (i.e., it is the last node in the inorder walk), we return `null`.

A more direct approach is to maintain a candidate node, `first`. The node `first` is initialized to `null`. Now we look for k using the standard search idiom. If the current node's key is larger than k, we update `first` to the current node and continue the

search in the left subtree. If the current node's key is smaller than k, we search in the right subtree. If the current node's key is equal to k, we set a Boolean-valued `found_k` variable to `true`, and continue search in the current node's right subtree. When the current node becomes `null`, if `found_k` is `true` we return `first`, otherwise we return `null`. Correctness follows from the fact that after `first` is assigned within the loop, the desired result is within the tree rooted at `first`. The concept of this approach is similar with Solution 11.2 on Page 259.

```
template <typename T>
shared_ptr<BinarySearchTree<T>> find_first_larger_k_with_k_exist(
    shared_ptr<BinarySearchTree<T>> r, const T &k) {
  bool found_k = false;
  shared_ptr<BinarySearchTree<T>> first = nullptr;

  while (r) {
    if (r->data == k) {
      found_k = true;
      r = r->right;
    } else if (r->data > k) {
      first = r;
      r = r->left;
    } else {  // r->data < k
      r = r->right;
    }
  }
  return found_k ? first : nullptr;
}
```

Problem 14.6, ***pg. 106***: *Write a function that takes a min-first BST T and a key k, and returns true iff T contains k.*

Solution 14.6: The algorithm proceeds in the following sequence:

1. If T is empty, or if k is less than the root, it cannot be present in T; we return false.
2. Otherwise, if k equals the key stored at T's root, we return true.
3. Otherwise, we recursively search both left and right subtrees and return true if and only if it is present in either.

```
template <typename T>
bool search_min_first_BST(const shared_ptr<BinarySearchTree<T>> &r,
                          const T &k) {
  if (!r || r->data > k) {
    return false;
  } else if (r->data == k) {
    return true;
  }
  return search_min_first_BST(r->left, k) ||
         search_min_first_BST(r->right, k);
}
```

Variant 14.6.1: Print the keys in a min-first BST in sorted order.

Variant 14.6.2: A max-first BST is defined analogously to the min-first BST, the difference being that the largest key is stored at the root. Design an algorithm that takes an n node min-BST and converts it to a max-BST in $O(n)$ time. Use as little additional space as possible.

Variant 14.6.3: Implement insert and delete functions for a min-first BST.

Problem 14.7, *pg.107*: *How would you build a BST of minimum possible height from a sorted array A?*

Solution 14.7: Intuitively, we want the subtrees to be as balanced as possible. One way of achieving this is to make the element at entry $\lfloor \frac{n}{2} \rfloor$ the root, and recursively compute minimum height BSTs for the subarrays $A[0 : \lfloor \frac{n}{2} \rfloor - 1]$ and $A[\lfloor \frac{n}{2} \rfloor + 1 : n-1]$.

```
// Build BST based on subarray A[start : end - 1]
template <typename T>
shared_ptr<BinarySearchTree<T>> build_BST_from_sorted_array_helper(
    const vector<T> &A, const int &start, const int &end) {
  if (start < end) {
    int mid = start + ((end - start) >> 1);
    return shared_ptr<BinarySearchTree<T>>(new BinarySearchTree<T>{
        A[mid],
        build_BST_from_sorted_array_helper(A, start, mid),
        build_BST_from_sorted_array_helper(A, mid + 1, end)}
    );
  }
  return nullptr;
}

template <typename T>
shared_ptr<BinarySearchTree<T>> build_BST_from_sorted_array(
    const vector<T> &A) {
  return build_BST_from_sorted_array_helper(A, 0, A.size());
}
```

Problem 14.8, *pg.107*: *Let L be a singly linked list of numbers, sorted in ascending order. Design an efficient algorithm that takes as input L, and builds a height-balanced BST on the entries in L. Your algorithm should run in $O(n)$ time, where n is the number of nodes in L. You cannot use dynamic memory allocation—reuse the nodes of L for the BST. You can update pointer fields, but cannot change node contents.*

Solution 14.8: The straightforward algorithm entails finding the midpoint m of the list, creating the root with the corresponding key, and recursing on the first half and the second half of the list. The time complexity satisfies the recurrence $T(n) = O(n) + 2T(\frac{n}{2})$—the $O(n)$ term comes from the traversal required to find the midpoint of the list. This solves to $T(n) = O(n \log n)$.

If the nodes were in an array A, we could index directly into A to obtain m, and the time complexity would satisfy $S(n) = O(1) + 2S(\frac{n}{2})$, which solves to $S(n) = O(n)$.

We can avoid the additional space required to convert the list to an array by first building the tree with empty keys, and then populating it by performing an inorder walk of the tree in conjunction with a traversal of the list.

Specifically, first we find the length n of the list, $O(n)$ operation. We then create the balanced tree recursively—create the root node, and a balanced left child on $L = \lfloor \frac{n}{2} \rfloor$, and a balanced right child on $R = n - L - 1$ nodes. This entails a $O(1)$ time spent per node, and is also an $O(1)$ operation. The inorder walk entails calling the next method of the list iterator per visited node, leading to an $O(n)$ complexity for the final population of the tree nodes.

```
// Build a BST from the (s + 1)-th to the e-th node in L
template <typename T>
shared_ptr<BinarySearchTree<T>> build_BST_from_sorted_doubly_list_helper(
    shared_ptr<node_t<T>> &L, const int &s, const int &e) {
  shared_ptr<BinarySearchTree<T>> curr = nullptr;
  if (s < e) {
    int m = s + ((e - s) >> 1);
    curr = shared_ptr<BinarySearchTree<T>>(new BinarySearchTree<T>);
    curr->left = build_BST_from_sorted_doubly_list_helper(L, s, m);
    curr->data = L->data;
    L = L->next;
    curr->right = build_BST_from_sorted_doubly_list_helper(L, m + 1, e);
  }
  return curr;
}

template <typename T>
shared_ptr<BinarySearchTree<T>> build_BST_from_sorted_doubly_list(
    shared_ptr<node_t<T>> L, const int &n) {
  return build_BST_from_sorted_doubly_list_helper(L, 0, n);
}
```

Problem 14.9, *pg.107*: *Design an algorithm that takes as input a BST B and returns a sorted doubly linked list on the same elements. Your algorithm should not allocate any new nodes. The original BST does not have to be preserved; use its nodes as the nodes of the resulting list, as shown in Figure 14.4 on Page 107.*

Solution 14.9: Here is a recursive solution. Build a list out of left subtree, append the root to it, and then append the list from the right subtree. Since we do a constant amount of work per tree node, the time complexity is $O(n)$, where n is the number of nodes in the BST. The space complexity is $\Theta(h)$, where h is the height of the BST. The worst case is for a completely left-skewed tree, i.e., a tree in which no node has a right child—n activation records are pushed on the stack.

```
// Transform a BST into a circular sorted doubly linked list in-place,
// return the head of the list
template <typename T>
shared_ptr<BinarySearchTree<T>> BST_to_doubly_list(
    const shared_ptr<BinarySearchTree<T>> &n) {
  // Empty subtree
  if (!n) {
```

```
    return nullptr;
  }

  // Recursively build the list from left and right subtrees
  auto l_head(BST_to_doubly_list(n->left));
  auto r_head(BST_to_doubly_list(n->right));

  // Append n to the list from left subtree
  shared_ptr<BinarySearchTree<T>> l_tail = nullptr;
  if (l_head) {
    l_tail = l_head->left;
    l_tail->right = n;
    n->left = l_tail;
    l_tail = n;
  } else {
    l_head = l_tail = n;
  }

  // Append the list from right subtree to n
  shared_ptr<BinarySearchTree<T>> r_tail = nullptr;
  if (r_head) {
    r_tail = r_head->left;
    l_tail->right = r_head;
    r_head->left = l_tail;
  } else {
    r_tail = l_tail;
  }
  r_tail->right = l_head, l_head->left = r_tail;

  return l_head;
}
```

Problem 14.10, *pg. 108*: *Let A and B be BSTs. Design an algorithm that merges them in $O(n)$ time. You cannot use dynamic allocation. You do not need to preserve the original trees. You can update pointer fields, but cannot change the key stored in a node.*

Solution 14.10: Our solution builds on Solution 14.9 on the previous page and Solution 14.8 on Page 314. We convert each BST into a doubly linked list using Solution 14.9 on the previous page, which runs in $O(n)$ time and $O(n)$ space. These two lists can be merged in $O(n)$ time and $O(1)$ space, as described in Solution 7.1 on Page 207. The resulting list can be converted into a BST using Solution 14.8 on Page 107, which uses $O(n)$ time and $O(\log n)$ space. Each of the three sub-routines does not explicitly allocate memory.

```
template <typename T>
void append_node(shared_ptr<BinarySearchTree<T>> &head,
                 shared_ptr<BinarySearchTree<T>> &tail,
                 shared_ptr<BinarySearchTree<T>> &n) {
  if (head) {
    tail->right = n, n->left = tail;
  } else {
    head = n;
```

```
  }
  tail = n;
}

template <typename T>
void append_node_and_advance(shared_ptr<BinarySearchTree<T>> &head,
                             shared_ptr<BinarySearchTree<T>> &tail,
                             shared_ptr<BinarySearchTree<T>> &n) {
  append_node(head, tail, n);
  n = n->right;  // advance n
}

// Merge two sorted linked lists, return the head of list
template <typename T>
shared_ptr<BinarySearchTree<T>> merge_sorted_linked_lists(
    shared_ptr<BinarySearchTree<T>> A, shared_ptr<BinarySearchTree<T>> B) {
  shared_ptr<BinarySearchTree<T>> sorted_list = nullptr, tail = nullptr;

  while (A && B) {
    append_node_and_advance(sorted_list, tail, A->data < B->data ? A : B);
  }

  // Append the remaining of A
  if (A) {
    append_node(sorted_list, tail, A);
  }
  // Append the remaining of B
  if (B) {
    append_node(sorted_list, tail, B);
  }
  return sorted_list;
}

template <typename T>
shared_ptr<BinarySearchTree<T>> merge_BSTs(
    shared_ptr<BinarySearchTree<T>> A, shared_ptr<BinarySearchTree<T>> B) {
  // Transform BSTs A and B into sorted doubly lists
  A = BST_to_doubly_list(A), B = BST_to_doubly_list(B);
  A->left->right = B->left->right = nullptr;
  A->left = B->left = nullptr;
  int len_A = count_len(A), len_B = count_len(B);
  return build_BST_from_sorted_doubly_list(merge_sorted_linked_lists(A, B),
                                           len_A + len_B);
}
```

Problem 14.11, ***pg. 108***: *Given the root of a BST and an integer k, design a function that finds the k largest elements in this BST. For example, if the input to your function is the BST in Figure 14.1 on Page 105 and k = 3, your function should return* ⟨53, 47, 43⟩.

Solution 14.11: We do a reverse inorder traversal of the tree: visit the right subtree, visit the root, then visit the left subtree. This results in nodes being visited in descending order. As soon as we visit k nodes, we can halt. Note that the time complexity of this approach is still $O(n)$ even if $k \ll n$, e.g., if the BST is of the form

of a list. The code below uses a vector to store the desired keys; as soon as the vector has k elements, we return.

```
template <typename T>
void find_k_largest_in_BST_helper(const shared_ptr<BinarySearchTree<T>> &r,
                                  const int &k, vector<T> &k_elements) {
  // Perform reverse inorder traversal
  if (r && k_elements.size() < k) {
    find_k_largest_in_BST_helper(r->right, k, k_elements);
    if (k_elements.size() < k) {
      k_elements.emplace_back(r->data);
      find_k_largest_in_BST_helper(r->left, k, k_elements);
    }
  }
}

template <typename T>
vector<T> find_k_largest_in_BST(const shared_ptr<BinarySearchTree<T>> &root,
                                const int &k) {
  vector<T> k_elements;
  find_k_largest_in_BST_helper(root, k, k_elements);
  return k_elements;
}
```

Problem 14.12, ***pg. 109****: Which traversal orders—inorder, preorder, and postorder—of a BST can be used to reconstruct the BST uniquely? Write a program that takes as input a sequence of node keys and computes the corresponding BST. Assume that all keys are unique.*

Solution 14.12: The sequence of node keys generated by an inorder traversal is not enough to reconstruct the tree. For example, the sequence $\langle 1,2,3\rangle$ corresponds to five distinct BSTs as shown in Figure 21.10.

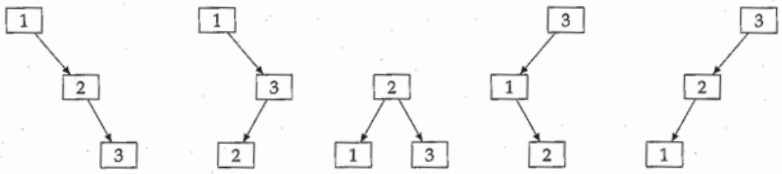


Figure 21.10: Five distinct BSTs for the traversal sequence $\langle 1,2,3\rangle$.

We claim there exists a unique BST corresponding to a sequence of nodes visited in a preorder traversal.

Proof:

We use induction on n, the number of nodes. Only one tree exists on one node, so the base case holds. Assume the claim holds for all $n < k$. Consider a preorder traversal sequence σ of length k. The first value v in σ corresponds to the root. The subsequence σ_l of σ which begins at the second element of σ and ends at the last

value less than v, corresponds to the preorder traversal of the root's left subtree. Its length is less than k. Hence by induction we can reconstruct the left subtree uniquely. The right subtree corresponds to the subsequence following σ_l, and can also be reconstructed uniquely by induction.

It is critical that the elements stored in the tree be unique. If the root contains v and the tree contains more occurrences of v, we cannot identify from the sequence whether the subsequent vs are in the left subtree or the right subtree.

The above proof can be used to rebuild the BST from a preorder sequence.

```
// Build a BST based on preorder[s : e - 1], return its root
template <typename T>
shared_ptr<BinarySearchTree<T>> rebuild_BST_from_preorder_helper(
    const vector<T> &preorder, const int &s, const int &e) {
  if (s < e) {
    int x = s + 1;
    while (x < e && preorder[x] < preorder[s]) {
      ++x;
    }
    return shared_ptr<BinarySearchTree<T>>(new BinarySearchTree<T>{
        preorder[s],
        rebuild_BST_from_preorder_helper(preorder, s + 1, x),
        rebuild_BST_from_preorder_helper(preorder, x, e)});
  }
  return nullptr;
}

// Given a preorder traversal of a BST, return its root
template <typename T>
shared_ptr<BinarySearchTree<T>> rebuild_BST_from_preorder(
    const vector<T> &preorder) {
  return rebuild_BST_from_preorder_helper(preorder, 0, preorder.size());
}
```

The worst-case input for this algorithm is the pre-order sequence corresponding to a completely left-skewed tree. The worst-case time complexity satisfies the recurrence $W(n) = W(n-1)+O(n)$, which solves to $O(n^2)$. The best-case input is a sequence corresponding to a completely right-skewed tree, and the corresponding time complexity is $O(n)$. When the sequence corresponds to a balanced BST, the time complexity is given by $B(n) = 2B(n/2) + O(n)$, which solves to $O(n \log n)$.

The implementation above potentially iterates over nodes multiple times, which is wasteful. A better approach is to reconstruct the left subtree in the same iteration as identifying the nodes which lie in it. The code shown below takes this approach. Its worst-case time complexity is $O(n)$, since it performs a constant amount of work per node.

```
template <typename T>
shared_ptr<BinarySearchTree<T>> rebuild_BST_from_preorder_helper(
    const vector<T> &preorder, int &idx, const T &min, const T &max) {
  if (idx == preorder.size()) {
    return nullptr;
```

```
  }

  T curr = preorder[idx];
  if (curr < min || curr > max) {
    return nullptr;
  }

  ++idx;
  shared_ptr<BinarySearchTree<T> > root(new BinarySearchTree<T>{curr,
      rebuild_BST_from_preorder_helper(preorder, idx, min, curr),
      rebuild_BST_from_preorder_helper(preorder, idx, curr, max)});
  return root;
}

template <typename T>
shared_ptr<BinarySearchTree<T> > rebuild_BST_from_preorder(
    const vector<T> &preorder) {
  int idx = 0;
  return rebuild_BST_from_preorder_helper(preorder, idx,
                                          numeric_limits<T>::min(),
                                          numeric_limits<T>::max());
}
```

Problem 14.13, *pg. 109*: *Design an algorithm that takes a BST T of size n and height h, nodes s and b, and returns the LCA of s and b. Assume* s.key < b.key. *For example, in Figure 14.1 on Page 105, if s is node C and b is node G, your algorithm should return node B. Your algorithm should run in $O(h)$ time and $O(1)$ space. Nodes do not have pointers to their parents.*

Solution 14.13: In Solution 9.11 on Page 245 we presented an algorithm for this problem in the context of binary trees. The idea underlying that algorithm was to do a postorder walk—the LCA is the first node visited after s and b have both been visited.

This idea can be refined for BSTs—since nodes satisfy the BST property and keys are distinct, we prune much of the exploration. Specifically, initialize x to the root. If $x.key = s.key$, $x.key = b.key$, or $((s.key < x.key)$ and $(x.key < b.key))$ then the LCA is x itself. Otherwise, if $x.key > b.key$ we set x to $x.left$ and continue the search, since the LCA must lie in $x.left$. Similarly, if $x.key < s.key$ we set x to $x.right$ and continue.

```
template <typename T>
shared_ptr<BinarySearchTree<T>> find_LCA(
    shared_ptr<BinarySearchTree<T>> x,
    const shared_ptr<BinarySearchTree<T>> &s,
    const shared_ptr<BinarySearchTree<T>> &b) {
  while (x->data < s->data || x->data > b->data) {
    if (x->data < s->data) {
      x = x->right;  // LCA must be in x's right child
    }
    if (x->data > b->data) {
      x = x->left;  // LCA must be in x's left child
    }
```

```
  }

  // x->data >= s->data && x->data <= b->data
  return x;  // x is LCA
}
```

Problem 14.14, *pg. 109*: *Let r, s, and m be distinct nodes in a BST. In this BST, nodes do not have pointers to their parents and all keys are unique. Write a function which returns* `true` *if m has both an ancestor and a descendant in the set {r, s}. For example, in Figure 14.1 on Page 105, if m is Node J, your function should return* `true` *if the given set is {A, K} and return* `false` *if the given set is {I, P}.*

Solution 14.14: There are two possibilities: m is a descendant of r and an ancestor of s, or m is an ancestor of r and a descendant of s.

Consider the first case. We can check if m is a descendant of r, and s is a descendant of m by simply doing one search for *s.key* in the subtree rooted at r, and recording whether m was encountered during the search. The search from r has time complexity $O(h)$, where h is the height of the tree, since we can use the BST property to prune one of the two children at each node. To check if m is an descendant of s, and r is a descendant of m, we do a symmetric search for *r.key* in the subtree rooted at s.

The disadvantage of performing these two searches one-after-another is that even when the distance between r and s is short, we may begin the search from the lower of the two, and incur the full $O(h)$ time complexity. We can prevent this by performing the searches from s for *r.key* and from r for *s.key* in an interleaved way; this way, if the final result returned is `true`, we will avoid performing an unsuccessful search on a large subtree.

```
template <typename T>
bool is_r_s_descendant_ancestor_of_m(
    const shared_ptr<BinarySearchTree<T>> &r,
    const shared_ptr<BinarySearchTree<T>> &s,
    const shared_ptr<BinarySearchTree<T>> &m) {
  shared_ptr<BinarySearchTree<T>> curr_r = r, curr_s = s;

  // Interleaving searches from r and s
  while (curr_r && curr_r != s && curr_s && curr_s != r) {
    if (curr_r == m || curr_s == m) {
      return true;
    }
    curr_r = curr_r->data > s->data ? curr_r->left : curr_r->right;
    curr_s = curr_s->data > r->data ? curr_s->left : curr_s->right;
  }

  // Keep searching from r
  while (curr_r && curr_r != s) {
    if (curr_r == m) {
      return true;
    }
    curr_r = curr_r->data > s->data ? curr_r->left : curr_r->right;
  }
```

```
  // Keep searching from s
  while (curr_s && curr_s != r) {
    if (curr_s == m) {
      return true;
    }
    curr_s = curr_s->data > r->data ? curr_s->left : curr_s->right;
  }
  return false;
}
```

Problem 14.15, ***pg. 109***: *How would you efficiently perform a range query on a BST? Specifically, write a function that takes as input a BST and a range* $[L, U]$ *and returns a list of all the keys that lie in* $[L, U]$*?*

Solution 14.15: We can return the list of nodes whose entries lie in $[L, U]$ by first finding the first node l whose entry is greater than or equal to L. Node l can be found by applying the technique of Solution 14.4 on Page 311. If this returns `null` (because no node has an entry equal to L), l is the first node with a key greater than L; this can be computed via a slightly modified version of Solution 14.5 on Page 312. Note that l may be `null` (if all node entries are less than L), in which case we return the empty list.

If l is not `null`, we make it the start of the result list. Now repeatedly call the successor function (Solution 14.2 on Page 308), adding successive entries to the list, stopping when the successor function returns `null` (which is the case if all node entries are less than or equal to U), or a node whose entry is greater than U.

The time complexity to find l is $O(h)$, where h is the height of the tree. Individual calls to the successor function have complexity $O(h)$, which leads to a $O(hm)$ bound overall, where m is the size of the list.

However, the bound is not tight—m successive calls to successor have time complexity $O(h + m)$. The reason is that we traverse less than or equal to $d_l + 2(m-1) + d_u$ edges, where d_l and d_u are the depths of l and the last node whose key is less than or equal to U. Both d_l and d_u are bounded by h, leading to the claimed time complexity.

```
template <typename T>
shared_ptr<BinarySearchTree<T>> find_first_larger_equal_k(
    const shared_ptr<BinarySearchTree<T>> &r, const T &k) {
  if (!r) {
    return nullptr;
  } else if (r->data >= k) {
    // Recursively search the left subtree for first one >= k
    auto n = find_first_larger_equal_k(r->left, k);
    return n ? n : r;
  }
  // r->data < k so search the right subtree
  return find_first_larger_equal_k(r->right, k);
}

template <typename T>
list<shared_ptr<BinarySearchTree<T>>> range_query_on_BST(
```

```
17      shared_ptr<BinarySearchTree<T>> n, const T &L, const T &U) {
18    list<shared_ptr<BinarySearchTree<T>>> res;
19    for (auto it = find_first_larger_equal_k(n, L);
20         it && it->data <= U;
21         it = find_successor_BST(it)) {
22      res.emplace_back(it);
23    }
24    return res;
25  }
```

This solution can be improved by computing the number of entries in a BST in a range $[L, U]$ without enumerating all the entries in that range. See Solution 14.22 on Page 331 for details.

Problem 14.16, *pg. 110*: *Design an algorithm that takes three sorted arrays A, B, and C and returns a triple (i, j, k) such that distance(i, j, k) is minimum. Your algorithm should run in $O(|A| + |B| + |C|)$ time.*

Solution 14.16: We follow an approach similar to merge sort. Specifically, we keep three index variables, one for each of A, B, and C. These variables are initialized to 0, i.e., they index the minimum elements of A, B, and C. We iteratively identify the index variable whose corresponding element is the minimum of three, breaking ties by giving preference to A over B over C, and advance that corresponding index. In each iteration we record the difference between the largest and the smallest of the three elements and track the minimum difference m seen, along with the corresponding indices.

We claim that after all elements are processed, m and its associated triple are distance minimizing.

Proof:

Clearly m is always an upper bound on the minimum distance. Let (i, j, k) be a distance minimizing triple. Without loss of generality, assume (1.) $A[i] \leq B[j] \leq C[k]$, and (2.) $A[i] < A[i+1]$ and $C[k] > C[k-1]$. Since (i, j, k) is optimum, there cannot exist i' such that $A[i'] \in (A[i], C[k])$, or k' such that $C[k'] \in (A[i], C[k])$. (There may exist one or more $j' \neq j$ such that $B[j'] \in (A[i], C[k])$.)

Our algorithm will process $A[i]$ before $C[k]$, and by the observation in the previous paragraph, no other element of A or C will be processed after $A[i]$ is processed and before $C[k]$ is processed. Since $A[i] \leq B[j] \leq C[k]$, index $B[j]$ will be processed after $A[i]$ is processed and before $C[k]$ is processed. When $C[k]$ is processed, the index variable for B must correspond to an element of B which lies in $[A[i], C[k]]$. Therefore, when $C[k]$ is processed, the minimum difference will either already be $C[k] - A[i]$, or updated to $C[k] - A[i]$, i.e., the algorithm computes the correct result.

In the following code, we implement a general purpose function which finds the minimum distance in k sorted arrays. These arrays are passed in as `arrs`. Since we need to repeatedly find the minimum among all those sorted arrays, we use a balanced BST to identify the array that contains the minimum element. The BST

also allows us to find the difference of the minimum and maximum values in the collection efficiently. The overall time complexity is $O(n \log k)$, where n is the total number of elements in the k arrays. For the special case $k = 3$ specified in the problem statement, the time complexity is $O(n \log 3) = O(n)$.

```
template <typename T>
class ArrData {
  public:
    int idx;
    T val;

    const bool operator<(const ArrData &a) const {
      if (val != a.val) {
        return val < a.val;
      } else {
        return idx < a.idx;
      }
    }
};

template <typename T>
T find_min_distance_sorted_arrays(const vector<vector<T>> &arrs) {
  // Pointers for each of arrs
  vector<int> idx(arrs.size(), 0);
  T min_dis = numeric_limits<T>::max();
  set<ArrData<T>> current_heads;

  // Each of arrs puts its minimum element into current_heads
  for (int i = 0; i < arrs.size(); ++i) {
    if (idx[i] >= arrs[i].size()) {
      return min_dis;
    }
    current_heads.emplace(ArrData<T>{i, arrs[i][idx[i]]});
  }

  while (true) {
    min_dis = min(min_dis, current_heads.crbegin()->val -
                           current_heads.cbegin()->val);
    int tar = current_heads.cbegin()->idx;
    // Return if there is no remaining element in one array
    if (++idx[tar] >= arrs[tar].size()) {
      return min_dis;
    }
    current_heads.erase(current_heads.begin());
    current_heads.emplace(ArrData<T>{tar, arrs[tar][idx[tar]]});
  }
}
```

Problem 14.17, *pg. 110*: *You are to implement methods to analyze log file data to find the most visited pages. Specifically, implement the following methods:*

- *`void add(Entry p)`—add `p.page` to the set of visited pages. It is guaranteed that if `add(q)` is called after `add(p)` then `q.timestamp` is greater than or equal*

to `p.timestamp`.

- `List<String> common(k)`—*return a list of the* `k` *most common pages.*

First solve this problem when `common(k)` *is called exactly once after all pages have been read. Then solve the problem when calls to* `common` *and* `add` *are interleaved. Assume you have unlimited RAM.*

Solution 14.17: For the first scenario, we keep a hash table H of (*page*, *count*) pairs. We process the log file entry-by-entry, inserting a page into H with a count of 1 if the page is not already present; otherwise we increment the count for the page. After all pages have been read we can compute the k most common pages by iterating through the pairs in H and using, for example, the techniques in 11.13 on Page 270 or 11.14 on Page 271.

When calls to `add` and `common` are interleaved it is more efficient to use a hash table M in conjunction with a BST B. The BST stores objects which consist of a frequency and a page—compares are made based on the frequency field, with ties being broken by the page field. (The tie-breaker is needed to ensure two pages with the same frequency have distinct entries in B.) The hash table M maps each page to its corresponding object in B. To add a page p we first do a find in M. If p is present, the frequency field in the corresponding object in B is updated. (The simplest way to update B is to do a delete followed by an insert.) Otherwise a new object with frequency equal to 1 and page equal to p is added to B. The time complexity of `add` is dominated by the BST update, which is $O(\log n)$, where n is the number of distinct pages processed so far.

The `common` method is implemented by finding the maximum element in B and making $k-1$ calls to the predecessor function. If B is balanced, the time complexity of $k-1$ calls to predecessor is $O(k+\log n)$. For $k \ll n$ this compares very favorably with having to iterate through the entire collection as we did in the first scenario. The penalty is the added overhead of the BST B, specifically the increased time to perform each add.

Problem 14.18, *pg. 110*: *Implement the API in Problem 14.17 on Page 110. If* `common` *is called after processing the i-th entry,* `common` *should return the k most visited pages whose timestamp is in* $[t_i - W, t_i]$. *Here* t_i *is the timestamp of the i-th entry and W is specified by the client before any pages are read and does not change. RAM is limited—in particular you cannot keep a map containing all pages. Maximize time efficiency assuming calls to* `add` *and* `common` *may be interleaved and* `common` *is frequently called.*

Solution 14.18: Define the current window when entry i is being processed to be the time interval $[t_i - W, t_i]$.

We use three data structures:

1. A queue Q containing the entries whose timestamp is in the current window.
2. A BST B containing page-frequency pairs, where elements are ordered by frequency.
3. A hash table M mapping pages to entries in the BST B.

The k largest elements in B are the desired pages for the current interval.

The add function enqueues p in Q and increments the corresponding page's frequency in B. The timestamp on p may result in the current window changing. We examine the head of the queue Q, and iteratively remove entries whose timestamp is outside the current window. For each entry that is removed, we reduce the frequency of the corresponding page in B. To do this, we use the hash table M to go from p's *visited_page* field to the corresponding entry in B.

The common function is implemented exactly as in Solution 14.17 on the preceding page.

The time complexity for adding pages is dominated by updates to the BST. In the worst case, every page is unique and all appear in the window, leading to a $O(n \log n)$ time complexity, where n is the number of log entries. The space complexity is $O(n)$.

In practical settings, the maximum number of pages in a window, and hence in B, will likely be much less than n. If the number of entries in a window is bounded by c, then the time complexity for n calls to add is $O(n \log c)$—the $\log c$ term corresponds to the time needed to perform BST updates. The space complexity is $O(c)$.

Problem 14.19, *pg. 111*: *Write a function that takes a single integer argument n and computes all the Gaussian integers $a + bi$, for $-n \leq a, b \leq n$ that are Gaussian primes.*

Solution 14.19: The *modulus* of a complex number $z = a + bi$ is by definition $\sqrt{a^2 + b^2}$, and is commonly denoted by $|z|$.

It is straightforward to see that $|z_1 z_2| = |z_1||z_2|$. Therefore, one approach to computing the Gaussian primes in the desired range is to sort the numbers in the range based on their modulus. The only numbers with modulus 1 are the units. The next smallest modulus is $\sqrt{2}$, e.g., $1 + i$. All numbers whose modulus is $\sqrt{2}$ must be Gaussian primes; this follows from the primality of 2 in the conventional integers. We can eliminate all multiples of such numbers by nonunits, and then examine the remaining candidates for the one with the smallest modulus, and continue. We maintain the set of candidates in a BST using the modulus to order numbers, breaking ties lexicographically. This approach is the analog of the sieve method for ordinary primes, described in Solution 5.11, extended to two dimensions.

```
bool is_unit(const complex<int> &z) {
  return (z.real() == 1 && z.imag() == 0) ||
         (z.real() == -1 && z.imag() == 0) ||
         (z.real() == 0 && z.imag() == 1) ||
         (z.real() == 0 && z.imag() == -1);
}

class ComplexCompare {
  public:
    const bool operator()(const complex<double> &lhs,
                          const complex<double> &rhs) const {
      if (norm(lhs) != norm(rhs)) {
        return norm(lhs) < norm(rhs);
      } else if (lhs.real() != rhs.real()) {
        return lhs.real() < rhs.real();
      } else {
        return lhs.imag() < rhs.imag();
```

```
      }
    }
};

vector<complex<int>> generate_Gaussian_primes(const int &n) {
  set<complex<double>, ComplexCompare> candidates;
  vector<complex<int>> primes;

  // Generate all possible Gaussian prime candidates
  for (int i = -n; i <= n; ++i) {
    for (int j = -n; j <= n; ++j) {
      if (is_unit({i, j}) == false && abs(complex<double>(i, j)) != 0) {
        candidates.emplace(i, j);
      }
    }
  }

  while (candidates.empty() == false) {
    complex<double> p = *(candidates.begin());
    candidates.erase(candidates.begin());
    primes.emplace_back(p);
    int max_multiplier = n / floor(sqrt(norm(p))) + 1;

    for (int i = max_multiplier; i >= -max_multiplier; --i) {
      for (int j = max_multiplier; j >= -max_multiplier; --j) {
        complex<double> x = {i, j};
        if (is_unit(x) == false) {
          candidates.erase(x * p);
        }
      }
    }
  }
  return primes;
}
```

Problem 14.20, ***pg. 111***: *Implement a function that computes the view from above. Your input is a sequence of line segments, each specified as a 4-tuple $\langle l, r, c, h \rangle$, where l and r are the left and right endpoints, respectively, c encodes the color, and h are the height. The output should be in the same format. No two segments whose intervals overlap have the same height.*

Solution 14.20: First observe that the left endpoint and the right endpoint of each segment in the view from above is the left or right endpoint of an input segment.

This observation leads to the following algorithm. Sort the endpoints of the segments and then do a sweep from left-to-right. As we sweep, we maintain the set of segments that intersect the current position; this set is stored in a BST with the height being the key. The color is determined by the highest segment. When we encounter a left endpoint, we add the corresponding segment in a BST. When we encounter a right endpoint, we remove the corresponding segment from the BST. We use the height field as a proxy for the segment, since the problem statement guarantees that the height uniquely determines the segment.

```
template <typename XaxisType, typename ColorType, typename HeightType>
class LineSegment {
  public:
    XaxisType left, right;  // specifies the interval
    ColorType color;
    HeightType height;

    const bool operator<(const LineSegment &that) const {
      return height < that.height;
    }
};

template <typename XaxisType, typename ColorType, typename HeightType>
class Endpoint {
  public:
    bool isLeft;
    const LineSegment<XaxisType, ColorType, HeightType>* l;

    const bool operator<(const Endpoint &that) const {
      return val() < that.val();
    }

    const XaxisType &val(void) const {
      return isLeft ? l->left : l->right;
    }
};

template <typename XaxisType, typename ColorType, typename HeightType>
void calculate_view_from_above(
    const vector<LineSegment<XaxisType, ColorType, HeightType>> &A) {
  vector<Endpoint<XaxisType, ColorType, HeightType>> E;
  for (int i = 0; i < A.size(); ++i) {
    E.emplace_back(Endpoint<XaxisType, ColorType, HeightType>{true, &A[i]});
    E.emplace_back(Endpoint<XaxisType, ColorType, HeightType>{false, &A[i]});
  }
  sort(E.begin(), E.end());

  XaxisType prev_xaxis = E.front().val();  // the first left end point
  shared_ptr<LineSegment<XaxisType, ColorType, HeightType>> prev = nullptr;
  map<HeightType, const LineSegment<XaxisType, ColorType, HeightType>*> T;
  for (const Endpoint<XaxisType, ColorType, HeightType> &e: E) {
    if (T.empty() == false && prev_xaxis != e.val()) {
      if (prev == nullptr) {  // found first segment
        prev = shared_ptr<LineSegment<XaxisType, ColorType, HeightType>>(
          new LineSegment<XaxisType, ColorType, HeightType>{
            prev_xaxis, e.val(), T.crbegin()->second->color,
            T.crbegin()->second->height});
      } else {
        if (prev->height == T.crbegin()->second->height &&
            prev->color == T.crbegin()->second->color) {
          prev->right = e.val();
        } else {
          cout << "[" << prev->left << ", " << prev->right << "]"
               << ", color = " << prev->color << ", height = "
```

```
55                  << prev->height << endl;
56              *prev = {prev_xaxis, e.val(), T.crbegin()->second->color,
57                       T.crbegin()->second->height};
58            }
59          }
60        }
61        prev_xaxis = e.val();
62
63        if (e.isLeft == true) {  // left end point
64          T.emplace(e.l->height, e.l);
65        } else {  // right end point
66          T.erase(e.l->height);
67        }
68      }
69
70      // Output the remaining segment if any
71      if (prev) {
72        cout << "[" << prev->left << ", " << prev->right << "]"
73             << ", color = " << prev->color << ", height = "
74             << prev->height << endl;
75      }
76    }
```

ϵ-Variant 14.20.1: Solve the same problem when multiple segments may have the same height. Break ties arbitrarily.

ϵ-Variant 14.20.2: Design an efficient algorithm for computing the length of the union of a set of closed intervals.

Variant 14.20.3: Design an efficient algorithm for computing the area of a set of rectangles whose sides are aligned with the X and Y axes.

Variant 14.20.4: Runners $R_1, R_2, \ldots, R_n$ race on a track of length L. Runner R_i begins at an offset s_i from the start of the track, and runs at speed v_i. Compute the set of runners who lead the race at some time.

Variant 14.20.5: Given a set H of nonintersecting horizontal line segments in the 2D plane, and a set V of nonintersecting vertical line segments in the 2D plane, determine if any pair of line segments intersect.

Problem 14.21, *pg. 112*: *Design a data structure that implements the following methods:*

- `insert(s,c)`, *which adds client s with credit c, overwriting any existing entry for s.*
- `remove(s)`, *which removes client s.*
- `lookup(s)`, *which returns the number of credits associated with client s, or* -1 *if s is not present.*
- `addAll(C)`, *the effect of which is to increment the number of credits for each client currently present by C.*

– `max()`, *which returns the client with the highest number of credits.*

The `insert(s,c)`, `remove(s)`, *and* `lookup(s)` *methods should run in time* $O(\log n)$, *where* n *is the number of clients. The remaining methods should run in time* $O(1)$.

Solution 14.21: We use one hash table, `credits` and one BST, `inverse_credits`. We also use an integer-valued variable `offset` which is initialized to 0. A call to `addAll(C)` increments `offset` by `C`.

The hash table `credits` consists of key-value pairs, where the key is the client string and value is an integer. It gives us the ability to do lookup in $O(1)$ time. The `lookup(s)` method returns the sum of `offset` and the value associated with `s` in `credits`. For `lookup(s)` to work correctly, we must subtract `offset` from `c` when performing `insert(s, c)`—this way a client's credits include only the credits added via `addAll` after it was inserted.

The BST `inverse_credits` is used to implement `max()` in $O(1)$ time. The entries in `inverse_credits` are key-value pairs, where each key is a value v from `credits`; its associated value is a hash table of the client strings which v is associated with in `credits`. The `insert(s, c)` and `remove(s)` methods entail updating `inverse_credits`. Insertion consists of a lookup in `inverse_credits` followed by either adding a new pair to `inverse_credits` (if `c - offset` is not associated with any key in `credits`) or adding a string to the hash table associated with `c - offset`. The time complexity for updating `inverse_credits` when `insert(s, c)` is called is $O(\log n)$ for the BST lookup plus $O(1)$ for the hash table creation or update, i.e., $O(\log n)$. Similarly, the time complexity for updating `inverse_credits` when `remove(s)` is called is $O(\log n)$. The detailed implementation is given below. We take advantage of STL's implementation of BST, in which the minimum and maximum entries are computed in constant time, to make `max()` an $O(1)$ time operation.

```
1  class ClientsCreditsInfo {
2    private:
3      int offset;
4      unordered_map<string, int> credits;
5      map<int, unordered_set<string>> inverse_credits;
6
7    public:
8      ClientsCreditsInfo(void) : offset(0) {}
9
10     void insert(const string &s, const int &c) {
11       credits.emplace(s, c - offset);
12       inverse_credits[c - offset].emplace(s);
13     }
14
15     void remove(const string &s) {
16       auto credits_it = credits.find(s);
17       if (credits_it != credits.end()) {
18         inverse_credits[credits_it->second].erase(s);
19         credits.erase(credits_it);
20       }
21     }
22
23     int lookup(const string &s) const {
```

```
24        auto it = credits.find(s);
25        return it == credits.cend() ? -1 : it->second + offset;
26      }
27
28      void addAll(const int &C) {
29        offset += C;
30      }
31
32      string max(void) const {
33        auto it = inverse_credits.crbegin();
34        return it == inverse_credits.crend() || it->second.empty() ?
35               "" : *it->second.cbegin();
36      }
37  };
```

Problem 14.22, ***pg. 112***: *Suppose each node in a BST has a size field, which denotes the number of nodes at the subtree rooted at that node, inclusive of the node. How would you efficiently compute the number of nodes that lie in a given range? Can the size field be updated efficiently on insert and on delete?*

Solution 14.22: We can find the number of nodes whose entries lie in $[L, U]$ by using the approach of Solution 14.15 on Page 322, and simply counting nodes, rather than inserting them into a list. This leads to the same time complexity. However, we can do better by exploiting the size field.

For example, suppose we want to find the number of entries that are less than a given value v. Initialize the count to 0. We search for the first occurrence of v, and each time we take a left child, we leave count unchanged; each time we take a right child, we add one plus the size of the corresponding left child. If v is present, when we reach the first occurrence of v, we add the size of v's left child. The same approach can be used to find the number of entries that are greater than v.

The time bound for these computations is $O(h)$, since the search always descends the tree. We can compute the number of nodes in the final result by first computing the number of nodes less than L and the number of nodes greater than H, and subtracting that from the total number of nodes (which is the size stored at the root).

The size field can be updated on insert and delete without changing the $O(h)$ time complexity of both. Essentially, the only nodes whose size field change are those on the search path to the added/deleted node. Some conditional checks are needed for each such node, but these add constant time per node, leaving the $O(h)$ time complexity unchanged.

Variant 14.22.1: Define the "Markowitz bullet" of a set of points P in the upper right quadrant of the Cartesian plane to be those points which are not below and to the right of any other point in P. Design a data structure for representing the Markowitz bullet. Specifically, it should be possible to efficiently check if a new point is below and to the right of some point in the Markowitz bullet, and to add a point to the Markowitz bullet (which may result in other points being removed from the bullet).

Problem 14.23, *pg. 113*: *Design a data structure that stores closed intervals and can efficiently return the complete set of intervals that intersect a specified range $[L, U]$. Your data structure must also support efficient insertions and deletions.*

Solution 14.23: The solution to this problem is based on the notion of an interval tree. This is a BST in which entries are intervals, e.g., [213, 455]. For simplicity we assume closed intervals throughout. Given an interval $I = [L, U]$, we will refer to L as the left endpoint of I, and U as the right endpoint of I. We use the left endpoint of the interval as the BST key.

Each node a stores an interval $[l_a, u_a]$, and also a max field m_a, which is the largest right endpoint amongst the intervals stored in the subtree rooted at a. It is fairly straightforward to show that the max field can be updated through inserts and deletes without changing the time complexity of these operations.

Searching for a node that intersects $I = [L, U]$ is done as follows. If the root r is `null` or $[l_r, u_r]$ has a nonempty intersection with I, return r; otherwise, if r's left child *r.left* is not `null` and the max field of *r.left* is greater than or equal to L, recurse on *r.left*; otherwise recurse on r's right child.

Since it descends the tree at each step, and performs constant work within a step, the procedure has time complexity $O(h)$ where h is the height of the tree. Its correctness follows from basic facts about intervals. The only tricky case is the justification that we do not need to search r's right subtree when the max field M of *r.left* is greater than or equal to L. The reasoning is as follows. Suppose no interval in *r.left* overlaps with I. We know there must be at least one interval of the form $[m, M]$ in *r.left*. Since I does not intersect any interval in *r.left*, it must be that $U < m$ or $L > M$. The latter is not possible, because the max field M is greater or equal to than L. Since the tree satisfies the BST property on the left end points, we know that for each $[p, q]$ in r's right subtree, $U < p$, and hence cannot intersect any interval in that subtree.

The problem statement asked for all intervals in the tree that intersect the given interval. We can do this by iteratively finding and removing the node returned in the procedure above, and later putting the nodes back in the tree. The time complexity is $O(mh)$, where m is the number of nodes in the result.

Variant 14.23.1: Solve the same problem without modifying the tree.

Problem 15.1, *pg. 115*: *Design an efficient algorithm for computing the skyline.*

Solution 15.1: The simplest solution is to compute the skyline incrementally. For one building, the skyline is trivial. Suppose we know the skyline for $n - 1$ buildings, and need to compute the new skyline when the n-th building (L_n, R_n, H_n) is added. We now iterate through the existing skyline from left to right to see where L_n should be added. Next we move through the existing skyline and increase any heights that are less than H_n to H_n until we reach R_n.

This algorithm is simple, but has $\Theta(n^2)$ complexity, since adding the n-th building may entail $\Theta(n)$ comparisons. A better solution is to use divide and conquer: we compute skylines for the first $\frac{n}{2}$ buildings and the last $\frac{n}{2}$ buildings, and merge the

results. The merge is similar to the procedure for adding a single building, described above, and can be performed in $O(n)$ time. Basically, we iterate through the two skylines together from left-to-right, matching their left and right coordinates, and adjusting heights appropriately.

```
template <typename CoordType, typename HeightType>
class Skyline {
  public:
    CoordType left, right;
    HeightType height;
};

template <typename CoordType, typename HeightType>
void merge_intersect_skylines(vector<Skyline<CoordType, HeightType>> &merged,
                              Skyline<CoordType, HeightType> &a, int &a_idx,
                              Skyline<CoordType, HeightType> &b, int &b_idx) {
  if (a.right <= b.right) {
    if (a.height > b.height) {
      if (b.right != a.right) {
        merged.emplace_back(a), ++a_idx;
        b.left = a.right;
      } else {
        ++b_idx;
      }
    } else if (a.height == b.height) {
      b.left = a.left, ++a_idx;
    } else {  // a.height < b.height
      if (a.left != b.left) {
        merged.emplace_back(
          Skyline<CoordType, HeightType>{a.left, b.left, a.height});
      }
      ++a_idx;
    }
  } else {  // a.right > b.right
    if (a.height >= b.height) {
      ++b_idx;
    } else {
      if (a.left != b.left) {
        merged.emplace_back(
          Skyline<CoordType, HeightType>{a.left, b.left, a.height});
      }
      a.left = b.right;
      merged.emplace_back(b), ++b_idx;
    }
  }
}

template <typename CoordType, typename HeightType>
vector<Skyline<CoordType, HeightType>> merge_skylines(
    vector<Skyline<CoordType, HeightType>> &L,
    vector<Skyline<CoordType, HeightType>> &R) {
  int i = 0, j = 0;
  vector<Skyline<CoordType, HeightType>> merged;

```

```
  while (i < L.size() && j < R.size()) {
    if (L[i].right < R[j].left) {
      merged.emplace_back(L[i++]);
    } else if (R[j].right < L[i].left) {
      merged.emplace_back(R[j++]);
    } else if (L[i].left <= R[j].left) {
      merge_intersect_skylines(merged, L[i], i, R[j], j);
    } else {  // L[i].left > R[j].left
      merge_intersect_skylines(merged, R[j], j, L[i], i);
    }
  }

  copy(L.cbegin() + i, L.cend(), back_inserter(merged));
  copy(R.cbegin() + j, R.cend(), back_inserter(merged));
  return merged;
}

template <typename CoordType, typename HeightType>
vector<Skyline<CoordType, HeightType>> drawing_skylines_helper(
    vector<Skyline<CoordType, HeightType>> &skylines,
    const int &start, const int &end) {
  if (end - start <= 1) {  // 0 or 1 skyline, just copy it
    return {skylines.cbegin() + start, skylines.cbegin() + end};
  }
  int mid = start + ((end - start) >> 1);
  auto L = drawing_skylines_helper(skylines, start, mid);
  auto R = drawing_skylines_helper(skylines, mid, end);
  return merge_skylines(L, R);
}

template <typename CoordType, typename HeightType>
vector<Skyline<CoordType, HeightType>> drawing_skylines(
    vector<Skyline<CoordType, HeightType>> skylines) {
  return drawing_skylines_helper(skylines, 0, skylines.size());
}
```

Variant 15.1.1: Compute the skyline problem when each building has the shape of an isosceles triangle with a 90 degree angle at its apex.

Problem 15.2, *pg. 115*: *Design an efficient algorithm that takes an array A of n numbers and returns the number of inverted pairs of indices.*

Solution 15.2: The brute-force algorithm examines all $i \in [0, n-1]$ and all $j \in [i+1, n-1]$, and has an $O(n^2)$ complexity. A more efficient approach is to use merge sort. Suppose we have counted the number of inversions in the left half L and the right half R of A. What are the inversions that remain to be counted? Sort the left and right half arrays, and merge the two halves. For any (i, j) pair, if $L[i] > R[j]$, then for all $i' \geq i$ we must have $L[i'] > R[j]$, and we need to add $m - i$ to the inversion count, where m is the length of L. The time complexity is identical to that for merge sort, i.e., $O(n \log n)$.

```
template <typename T>
int merge(vector<T> &A, const int &start, const int &mid, const int &end) {
  vector<T> sorted_A;
  int left_start = start, right_start = mid, inver_count = 0;

  while (left_start < mid && right_start < end) {
    if (A[left_start] <= A[right_start]) {
      sorted_A.emplace_back(A[left_start++]);
    } else {
      // A[left_start:mid - 1] will be the inversions
      inver_count += mid - left_start;
      sorted_A.emplace_back(A[right_start++]);
    }
  }
  copy(A.begin() + left_start, A.begin() + mid, back_inserter(sorted_A));
  copy(A.begin() + right_start, A.begin() + end, back_inserter(sorted_A));

  // Update A with sorted_A
  copy(sorted_A.begin(), sorted_A.end(), A.begin() + start);
  return inver_count;
}

template <typename T>
int count_inversions_helper(vector<T> &A, const int &start, const int &end) {
  if (end - start <= 1) {
    return 0;
  }

  int mid = start + ((end - start) >> 1);
  return count_inversions_helper(A, start, mid) +
         count_inversions_helper(A, mid, end) + merge(A, start, mid, end);
}

template <typename T>
int count_inversions(vector<T> A) {
  return count_inversions_helper(A, 0, A.size());
}
```

Problem 15.3, ***pg.116***: *You are given a list of pairs of points in the two-dimensional Cartesian plane. Each point has integer x and y coordinates. How would you find the two closest points?*

Solution 15.3: The brute-force solution is to consider all pairs of points: this yields an $O(n^2)$ algorithm.

To improve upon the brute-force solution, it is instructive to consider the one-dimensional case. The obvious solution for the one-dimensional case is to iterate through the points in sorted order, comparing the distance between successive points with the running minimum. However, this does not generalize to the two-dimensional case, since there is no natural total ordering of the points. Another approach for the one-dimensional case is divide and conquer: partition the set about the median, solve the problem for the left and right partitions, and combine the

results. The last step entails finding points closest to the median from the left and right partitions.

The complexity of the partitioning approach is the same as that of the approach based on sorting—$O(n \log n)$, where n is the number of points. However the partitioning approach is applicable in more than one dimension. Specifically, we can split the points into two equal-sized sets using a line $x = P$ parallel to the y-axis. (Such a line can be found by computing the median of the values for the x coordinates—this calculation can be performed using the algorithm described in Solution 11.13 on Page 270.)

We can then compute the closest pair of points recursively on the two sets; let the closest pair of points on the left of P be d_l apart and the closest pair of points to the right of P be d_r apart. Let $d = \min(d_l, d_r)$.

Now, all we need to look at is points which are in the band $[P - d, P + d]$. In degenerate situations, all points may be within this band. If we compare all the pairs, the complexity becomes quadratic again. However we can sort the points in the band on their y coordinates and iterate through the sorted list, looking for points d or less distance from the point being processed.

Intuitively, there cannot be many such points since otherwise, the closest pair in the left and right partitions would have to be less than d apart. This intuition can be analytically justified—Shamos and Hoey's famous 1975 paper *"Closest-point problems"* shows that no more than six points can be within d distance of any point which leads to an $O(n \log n)$ algorithm. (The time is dominated by the need to sort.)

The recursion can be sped up by switching to brute-force when a small number of points remain.

```
class Point {
  public:
    int x, y;
};

double distance(const Point &a, const Point &b) {
  return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}

// Return the closest two points and its distance as a tuple
tuple<Point, Point, double> brute_force(const vector<Point> &P, const int &s,
                                        const int &e) {
  tuple<Point, Point, double> ret;
  get<2>(ret) = numeric_limits<double>::max();
  for (int i = s; i < e; ++i) {
    for (int j = i + 1; j < e; ++j) {
      double dis = distance(P[i], P[j]);
      if (dis < get<2>(ret)) {
        ret = {P[i], P[j], dis};
      }
    }
  }
  return ret;
}
```

```
25
26 // Return the closest two points and its distance as a tuple
27 tuple<Point, Point, double> find_closest_pair_in_remain(vector<Point> &P,
28                                                         const double &d) {
29   sort(P.begin(), P.end(), [](const Point &a, const Point &b) -> bool {
30                              return a.y < b.y;
31                            });
32
33   // At most six points in P
34   tuple<Point, Point, double> ret;
35   get<2>(ret) = numeric_limits<double>::max();
36   for (int i = 0; i < P.size(); ++i) {
37     for (int j = i + 1; j < P.size() && P[j].y - P[i].y < d; ++j) {
38       double dis = distance(P[i], P[j]);
39       if (dis < get<2>(ret)) {
40         ret = {P[i], P[j], dis};
41       }
42     }
43   }
44   return ret;
45 }
46
47 // Return the closest two points and its distance as a tuple
48 tuple<Point, Point, double> find_closest_pair_points_helper(
49     const vector<Point> &P, const int &s, const int &e) {
50   if (e - s <= 3) {  // brute-force to find answer if there are <= 3 points
51     return brute_force(P, s, e);
52   }
53
54   int mid = (e + s) >> 1;
55   auto l_ret = find_closest_pair_points_helper(P, s, mid);
56   auto r_ret = find_closest_pair_points_helper(P, mid, e);
57   auto min_l_r = get<2>(l_ret) < get<2>(r_ret) ? l_ret : r_ret;
58   vector<Point> remain;  // stores the points whose x-dis < min_d
59   for (const Point &p : P) {
60     if (abs(p.x - P[mid].x) < get<2>(min_l_r)) {
61       remain.emplace_back(p);
62     }
63   }
64
65   auto mid_ret = find_closest_pair_in_remain(remain, get<2>(min_l_r));
66   return get<2>(mid_ret) < get<2>(min_l_r) ? mid_ret : min_l_r;
67 }
68
69 pair<Point, Point> find_closest_pair_points(vector<Point> P) {
70   sort(P.begin(), P.end(), [](const Point &a, const Point &b) -> bool {
71                              return a.x < b.x;
72                            });
73   auto ret = find_closest_pair_points_helper(P, 0, P.size());
74   return {get<0>(ret), get<1>(ret)};
75 }
```

Problem 15.4, *pg. 116*: *Design an efficient algorithm to compute the diameter of a tree.*

Solution 15.4: We can compute the diameter by running BFS, described on Page 132, from each node and recording the maximum value of the shortest path distances computed. This has $O(|V|(|V| + |E|)) = O(|V|^2)$ time complexity since $|E| = |V| - 1$ in a tree.

We can achieve better time complexity by using divide and conquer. First we define some notation. If T is a nonempty tree, let root(T) denote the node at the root of T. Let $l_{u,v}$ be the length of the edge (u, v). The *degree* of a node u in a rooted tree is the number of its children. Define the *weighted height* h_u of a tree rooted at u to be 0 if u is a leaf and $\max_{1 \leq i \leq n}(l_{u,\text{root}(T_i)} + h_{\text{root}(T_i)})$, where $T_1, T_2, \ldots, T_n$ are the subtrees rooted at u's children.

Let T be a tree whose root is r. Suppose r has degree m. For now, assume $m \geq 2$. Let $d_1, d_2, \ldots, d_m$ be the diameters and $h_1, h_2, \ldots, h_n$ the weighted heights of the subtrees.

Let λ be a longest path in T. Either it passes through r or it does not. If λ does not pass through r, it must be entirely within one of the m subtrees and hence the longest path length in T is the maximum of $d_1, d_2, \ldots, d_m$. If it does pass through r, it must be between a pair of nodes in distinct subtrees that are farthest from r. The distance from r to the node in T_i that is farthest from it is simply $f_i = h_i + l_{r,i}$, where $l_{u,v}$ denotes the length of the edge (u, v). Therefore the longest length path in T is the larger of the maximum of $d_1, d_2, \ldots, d_m$ and the sum of the two largest f_is.

Now we consider the cases $m = 0$ and $m = 1$. If $m = 0$ the subtree rooted at t is just the node t and the length of the longest path is 0. If $m = 1$ the length of the longest path in t is $\max(h_1 + l_{r,1}, d_1)$.

The following algorithm computes the tree diameter. Process the tree in bottom-up fashion. For each node we process its subtrees one at a time. We update the maximum tree diameter based on the subtree weighted heights, diameters, and edge weights, using the observations above. The time complexity is proportional to the size of the tree, i.e., $O(|V|)$.

```
class TreeNode {
  public:
    vector<pair<shared_ptr<TreeNode>, double>> edges;
};

// Return (height, diameter) pair
pair<double, double> compute_height_and_diameter(
    const shared_ptr<TreeNode> &r) {
  double diameter = numeric_limits<double>::min();
  array<double, 2> height = {0.0, 0.0};  // store the max 2 heights
  for (const pair<shared_ptr<TreeNode>, double> &e : r->edges) {
    pair<double, double> h_d = compute_height_and_diameter(e.first);
    if (h_d.first + e.second > height[0]) {
      height[1] = height[0];
      height[0] = h_d.first + e.second;
    } else if (h_d.first + e.second > height[1]) {
      height[1] = h_d.first + e.second;
    }
    diameter = max(diameter, h_d.second);
  }
  return {height[0], max(diameter, height[0] + height[1])};
```

```
}

double compute_diameter(const shared_ptr<TreeNode> &T) {
  return T ? compute_height_and_diameter(T).second : 0.0;
}
```

Problem 15.5, ***pg. 118***: *Given a circular array A, compute its maximum subarray sum in $O(n)$ time, where n is the length of A. Can you devise an algorithm that takes $O(n)$ time and $O(1)$ space?*

Solution 15.5: First recall the standard algorithm for the conventional maximum subarray sum problem. This proceeds by computing the maximum subarray sum $S[i]$ when the subarray ends at i, which is $\max(S[i-1] + A[i], A[i])$. Its running time is $O(n)$, where n is the length of the array.

One approach for the maximum circular subarray is to break the problem into two separate instances. The first instance is the noncircular one, and is solved as described above.

The second instance entails looking for the maximum subarray that cycles around. Naïvely, this entails finding the maximum subarray that starts at index 0, the maximum subarray ending at index $n-1$, and adding their sums. However, these two subarrays may overlap, and simply subtracting out the overlap does not always give the right result (consider the array $\langle 10, -4, 5, -4, 10 \rangle$).

Instead, we compute for each i the maximum subarray sum S_i for the subarray that starts at 0 and ends at or before i, and the maximum subarray E_i for the subarray that starts after i and ends at the last element. Then the maximum subarray sum for a subarray that cycles around is the maximum over all i of $S_i + E_i$.

```
// Calculate the non-circular solution
template <typename T>
T find_max_subarray(const vector<T> &A) {
  T maximum_till = 0, maximum = 0;
  for (const T &a : A) {
    maximum_till = max(a, a + maximum_till);
    maximum = max(maximum, maximum_till);
  }
  return maximum;
}

// Calculate the solution which is circular
template <typename T>
T find_circular_max_subarray(const vector<T> &A) {
  // Maximum subarray sum starts at index 0 and ends at or before index i
  vector<T> maximum_begin;
  T sum = A.front();
  maximum_begin.emplace_back(sum);
  for (int i = 1; i < A.size(); ++i) {
    sum += A[i];
    maximum_begin.emplace_back(max(maximum_begin.back(), sum));
  }

```

```
  // Maximum subarray sum starts at index i + 1 and ends at the last element
  vector<T> maximum_end(A.size());
  maximum_end.back() = 0;
  sum = 0;
  for (int i = A.size() - 2; i >= 0; --i) {
    sum += A[i + 1];
    maximum_end[i] = max(maximum_end[i + 1], sum);
  }

  // Calculate the maximum subarray which is circular
  T circular_max = 0;
  for (int i = 0; i < A.size(); ++i) {
    circular_max = max(circular_max, maximum_begin[i] + maximum_end[i]);
  }
  return circular_max;
}

template <typename T>
T max_subarray_sum_in_circular(const vector<T> &A) {
  return max(find_max_subarray(A), find_circular_max_subarray(A));
}
```

Alternately, the maximum subarray that cycles around can be determined by computing the minimum subarray—the remaining elements yield a subarray that cycles around. (One or both of the first and last elements may not be included in this subarray, but that is fine.) This approach uses $O(1)$ space and $O(n)$ time; code for it is given below.

```
template <typename T>
T find_optimum_subarray_using_comp(const vector<T> &A,
                                   const T&(*comp)(const T&, const T&)) {
  T till = 0, overall = 0;
  for (const T &a : A) {
    till = comp(a, a + till);
    overall = comp(overall, till);
  }
  return overall;
}

template <typename T>
T max_subarray_sum_in_circular(const vector<T> &A) {
  // Find the max in non-circular case and circular case
  return max(find_optimum_subarray_using_comp(A, max),  // non-circular case
             accumulate(A.cbegin(), A.cend(), 0) -
             find_optimum_subarray_using_comp(A, min));  // circular case
}
```

Problem 15.6, ***pg.119***: *Given an array A of n numbers, find a longest subsequence $\langle i_0, \ldots, i_{k-1} \rangle$ such that $i_j < i_{j+1}$ and $A[i_j] \leq A[i_{j+1}]$ for any $j \in [0, k-2]$.*

Solution 15.6: We present two solutions, an $O(n^2)$, and an $O(n \log n)$ one.

We first describe the $O(n^2)$ solution. Let s_i be the length of the longest nondecreasing subsequence of A that ends at $A[i]$ (specifically, $A[i]$ is included in this

subsequence). Then we can write the following recurrence:

$$s_i = \max\left(\max_{j\in[0,i-1]}\begin{cases} s_j+1, & \text{if } A[j] \le A[i]; \\ 1, & \text{otherwise.} \end{cases}, 1\right)$$

We use this recurrence to fill up a table for s_i. The time complexity of this algorithm is $O(n^2)$. If we want the sequence as well, for each i, in addition to storing the length of the sequence, we store the index of the last element of sequence that we extended to get the current sequence. Here is an implementation of this algorithm:

```
template <typename T>
vector<T> longest_nondecreasing_subsequence(const vector<T> &A) {
  // Empty array
  if (A.empty() == true) {
    return A;
  }

  vector<int> longest_length(A.size(), 1), previous_index(A.size(), -1);
  int max_length_idx = 0;
  for (int i = 1; i < A.size(); ++i) {
    for (int j = 0; j < i; ++j) {
      if (A[i] >= A[j] && longest_length[j] + 1 > longest_length[i]) {
        longest_length[i] = longest_length[j] + 1;
        previous_index[i] = j;
      }
    }
    // Record the index where longest subsequence ends
    if (longest_length[i] > longest_length[max_length_idx]) {
      max_length_idx = i;
    }
  }

  // Build the longest nondecreasing subsequence
  int max_length = longest_length[max_length_idx];
  vector<T> ret(max_length);
  while (max_length > 0) {
    ret[--max_length] = A[max_length_idx];
    max_length_idx = previous_index[max_length_idx];
  }
  return ret;
}
```

We now describe a subtler algorithm that has $O(n \log n)$ complexity. Let $M_{i,j}$ be the smallest possible tail value for any nondecreasing subsequence of length j using array elements $A[0], A[1], \ldots, A[i]$. Note that for any i, we must have $M_{i,1} \le M_{i,2} \le \cdots \le M_{i,j}$.

We process A's elements iteratively. When processing $A[i+1]$, we look for the largest j such that $M_{i,j} \le A[i+1]$. First, assume such a j exists. Then we can construct a $j+1$ length subsequence that ends at $A[i+1]$. If no length $j+1$ nondecreasing subsequence exists in $A[0], A[1], \ldots, A[i]$, then $M_{i+1,j+1}$ must be $A[i+1]$, otherwise it remains equal to $M_{i,j+1}$. Furthermore, $M_{i+1,j'}$ remains unchanged for all $j' \le j$.

Now suppose there does not exist j such that $M_{i,j} \le A[i+1]$. This can only be true

if $A[i+1]$ is the unique smallest element in $A[0:i+1]$. Therefore we set $M_{i+1,1}$ to $A[i+1]$.

Therefore processing $A[i+1]$ entails a binary search for j and then an update to $M_{i+1,j+1}$ if possible, leading to an $O(n \log n)$ time complexity.

Code implementing this procedure is given below; the appropriate entries from M are maintained in the `tail_values` vector.

```
template <typename T>
int longest_nondecreasing_subsequence(const vector<T> &A) {
  vector<T> tail_values;
  for (const T &a : A) {
    auto it = upper_bound(tail_values.begin(), tail_values.end(), a);
    if (it == tail_values.end()) {
      tail_values.emplace_back(a);
    } else {
      *it = a;
    }
  }
  return tail_values.size();
}
```

ϵ-Variant 15.6.1: Define a sequence of numbers $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ to be *alternating* if $a_i < a_{i+1}$ for even i and $a_i > a_{i+1}$ for odd i. Given an array of numbers A of length n, find a longest subsequence $\langle i_0, \ldots, i_{k-1} \rangle$ such that $\langle A[i_0], A[i_1], \ldots, A[i_{k-1}] \rangle$ is alternating.

ϵ-Variant 15.6.2: Define a sequence of numbers $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ to be *weakly alternating* if no three consecutive terms in the sequence are increasing or decreasing. Given an array of numbers A of length n, find a longest subsequence $\langle i_0, \ldots, i_{k-1} \rangle$ such that $\langle A[i_0], A[i_1], \ldots, A[i_{k-1}] \rangle$ is weakly alternating.

ϵ-Variant 15.6.3: Define a sequence of numbers $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ to be *convex* if $a_i < \frac{a_{i-1}+a_{i+1}}{2}$, for $1 \leq i \leq n-2$. Given an array of numbers A of length n, find a longest subsequence $\langle i_0, \ldots, i_{k-1} \rangle$ such that $\langle A[i_0], A[i_1], \ldots, A[i_{k-1}] \rangle$ is convex.

ϵ-Variant 15.6.4: Define a sequence of numbers $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ to be *bitonic* if there exists k such that $a_i < a_{i+1}$, for $0 \leq i < k$ and $a_i > a_{i+1}$, for $k \leq i < n-1$. Given an array of numbers A of length n, find a longest subsequence $\langle i_0, \ldots, i_{k-1} \rangle$ such that $\langle A[i_0], A[i_1], \ldots, A[i_{k-1}] \rangle$ is bitonic.

ϵ-Variant 15.6.5: Define a sequence of points in the plane to be *ascending* if each point is above and to the right of the previous point. How would you find a maximum ascending subset of a set of points in the plane?

Problem 15.7, *pg. 119*: *Design an algorithm that takes as input an array A of n numbers and a key k, and returns a longest subarray of A for which the subarray sum is less than or equal to k.*

Solution 15.7: Let R be the prefix sum array for A, i.e., $R[i] = \sum_{k=0}^{i} A[k]$; R can be computed in a single iteration over A. Note that the sum of the elements in the subarray $A[i:j]$ is $R[j] - R[i-1]$ (for convenience, take $R[-1] = 0$).

We can now use the idea of the "efficient frontier" of candidate starting indices: we will never choose i as the starting index if there exists i' such that $i' < i$ and $R[i'] \geq R[i]$. The frontier can be captured by the array $T[i] = \min_{k=i}^{n-1}(R[k])$, which can be computed by a single iteration over R, starting from the end of R and working backwards. Observe that the entries in T are monotonically increasing, i.e., for all $i, T[i] \leq T[i+1]$.

Consider any index i. Let j be the largest index such that $T[j] \leq k + R[i]$. We claim the longest subarray starting at $i+1$ that has a sum less than or equal to k must end at j, inclusive.

Proof:

First note that by definition of T there exists $j' \geq j$ such that $R[j'] = T[j]$. Hence there is a subarray starting at $i+1$ that has length at least $j - i$ that satisfies the subarray sum constraint (i.e., $R[j] - R[i] \leq k$).

Now suppose for contradiction that there exists a longer subarray satisfying the subarray sum constraint starting at $i+1$ and ending at j'' where $j'' > j$. Since this subarray satisfies the subarray sum constraint, we have $R[j''] - R[i] \leq k$, i.e., $R[j''] \leq k + R[i]$. Now j was chosen to be the largest index such that $T[j] \leq k + R[i]$, and by hypothesis $j < j''$ implies $T[j''] \leq R[j''] \leq k + R[i]$. This contradicts the maximality of j, so j is indeed the ending index of the longest subarray starting at i that satisfies the subarray sum constraint.

We have previously shown how T can be computed in $O(n)$ time. Given an index i we can compute the corresponding j using the variant of binary search that finds the greatest entry less than the search key. Specifically, Solution 11.2 on Page 259 shows how to find the smallest element larger than the search key in $O(\log n)$. A symmetric algorithm solves the problem of finding the greatest element less than the search key.

We do this once for each i, and record the longest subarray seen thus far, leading to an overall time complexity of $O(n \log n)$.

```
template <typename T>
pair<int, int> find_longest_subarray_less_equal_k(const vector<T> &A,
                                                  const T &k) {
  // Build the prefix sum according to A
  vector<T> prefix_sum;
  partial_sum(A.cbegin(), A.cend(), back_inserter(prefix_sum));

  vector<T> min_prefix_sum(prefix_sum);
  for (int i = min_prefix_sum.size() - 2; i >= 0; --i) {
    min_prefix_sum[i] = min(min_prefix_sum[i], min_prefix_sum[i + 1]);
  }

  pair<int, int> arr_idx(0, upper_bound(min_prefix_sum.cbegin(),
                                        min_prefix_sum.cend(), k) -
                            min_prefix_sum.cbegin() - 1);
  for (int i = 0; i < prefix_sum.size(); ++i) {
```

```
    int idx = upper_bound(min_prefix_sum.cbegin(), min_prefix_sum.cend(),
                          k + prefix_sum[i]) - min_prefix_sum.cbegin() - 1;
    if (idx - i - 1 > arr_idx.second - arr_idx.first) {
      arr_idx = {i + 1, idx};
    }
  }
  return arr_idx;
}
```

Variant 15.7.1: Design an algorithm for finding the longest subarray of a given array such that the average of the subarray elements is $\leq k$.

Problem 15.8, *pg. 120*: *Let A be an array of n numbers encoding the heights of adjacent buildings of unit width. Design an algorithm to compute the area of the largest rectangle contained in this skyline, i.e., compute* $\max_{i<j}((j-i+1) \times \min_{k=i}^{j} A[k])$.

Solution 15.8: A brute-force approach is to take each (i, j) pair, find the minimum of subarray $A[i:j]$, and multiply that by $j-i+1$. This has time complexity $O(n^3)$, which can improved to $O(n^2)$ by iterating over i and then $j \geq i$ and tracking the minimum height of buildings from i to j, inclusive.

Another approach is to iterate over the buildings in the following fashion: when processing the i-th building, we find the largest rectangle which includes that building and has height at least $A[i]$. Implemented naïvely, this approach has time complexity $O(n^2)$. However, we can improve upon it greatly with the following observation: if $k < k' \leq i$ and $A[k] > A[k']$ then there is no reason to consider Building k, when processing Building i. Therefore we can maintain an "efficient frontier" of candidates from Building 0 to Building i. This consists of a stack of buildings: a building is pushed into the stack iff its height is greater than the height of the building on the top of the stack. Observe that the largest rectangle under Building i that has height $A[i]$ extends out to the last building to the left that is in the stack and has height greater than or equal to $A[i]$. Before adding Building i to the stack, we remove buildings from the stack until the stack is empty or the top building has height less than $A[i]$. A similar stack is used to determine the largest rectangle to the right under Building i.

For each building, it will at most be pushed and popped from each of the two stacks at most once, which leads to an amortized $O(1)$ time complexity per building for stack updates, and an overall $O(n)$ time complexity. Following is the implementation in C++:

```
template <typename T>
T calculate_largest_rectangle(const vector<T> &A) {
  // Calculate L
  stack<int> s;
  vector<int> L;
  for (int i = 0; i < A.size(); ++i) {
    while (!s.empty() && A[s.top()] >= A[i]) {
      s.pop();
    }
    L.emplace_back(s.empty() ? -1 : s.top());
```

```
    s.emplace(i);
  }

  // Clear stack for calculating R
  while (!s.empty()) {
    s.pop();
  }
  vector<int> R(A.size());
  for (int i = A.size() - 1; i >= 0; --i) {
    while (!s.empty() && A[s.top()] >= A[i]) {
      s.pop();
    }
    R[i] = s.empty() ? A.size() : s.top();
    s.emplace(i);
  }

  // For each A[i], find its maximum area include it
  T max_area = 0;
  for (int i = 0; i < A.size(); ++i) {
    max_area = max(max_area, A[i] * (R[i] - L[i] - 1));
  }
  return max_area;
}
```

ϵ-Variant 15.8.1: Find the largest square under the skyline.

Problem 15.9, *pg. 120*: *Let A be an $n \times m$ Boolean 2D array. Design efficient algorithms for the following two problems:*

- *What is the largest 2D subarray containing only 1s?*
- *What is the largest square 2D subarray containing only 1s?*

What are the time and space complexities of your algorithms as a function of n and m?

Solution 15.9: A brute-force approach is to examine all 2D subarrays. Since a 2D subarray is characterized by two diagonally opposite corners the total number of such arrays is $O(m^2n^2)$ Each 2D subarray can be checked by examining the corresponding entries, so the overall complexity is $O(m^3n^3)$. This can be easily reduced to $O(m^2n^2)$ by processing 2D subarrays by size, and reusing results—the 2D subarray $A[i : i + a][j : j + b]$ is feasible iff the 2D subarrays $A[i : i + a - 1][j : j + b - 1]$, $A[i + a : i + a][j : j + b - 1]$, $A[i : i + a - 1][j + b : j + b]$, and $A[i + a : i + a][j + b : j + b]$ are feasible. This is a $O(1)$ time operation, assuming that feasibility of the smaller 2D subarrays has already been computed and stored. (Note that this solution requires $O(m^2n^2)$ storage.)

The following approach lowers the time and space complexity. For each feasible entry $A[i][j]$ we record $(h_{i,j}, w_{i,j})$, where $h_{i,j}$ is the largest L such that all the entries in $A[i : i + L - 1][j : j]$ are feasible, and $w_{i,j}$ is the largest L such that all the entries in $A[i : i][j : j + L - 1]$ are feasible. This computation can be performed in $O(mn)$ time, and requires $O(mn)$ storage.

Now for each feasible entry $A[i][j]$ we calculate the largest 2D subarray that has $A[i][j]$ as its bottom-left corner. We do this by processing each entry in $A[i :$

$i+h_{i,j}-1][j:j]$. As we iterate through the entries in vertical order, we update w to the smallest $w_{i,j}$ amongst the entries processed so far. The largest 2D subarray that has $A[i][j]$ as its bottom-left corner and $A[i'][j]$ as its top-left corner has area $(i'-i+1)w$. We track the largest 2D subarray seen so far across all $A[i][j]$ processed.

The time complexity per $A[i][j]$ is proportional to the number of rows, i.e., $O(n)$, yielding an overall time complexity of $O(mn^2)$, and space complexity of $O(mn)$.

```
class MaxHW {
  public:
    int h, w;
};

int max_rectangle_submatrix(const vector<vector<bool>> &A) {
  // DP table stores (h, w) for each (i, j)
  vector<vector<MaxHW>> table(A.size(), vector<MaxHW>(A.front().size()));

  for (int i = A.size() - 1; i >= 0; --i) {
    for (int j = A[i].size() - 1; j >= 0; --j) {
      // Find the largest h such that (i, j) to (i + h - 1, j) are feasible
      // Find the largest w such that (i, j) to (i, j + w - 1) are feasible
      table[i][j] = A[i][j] ?
                    MaxHW{i + 1 < A.size() ? table[i + 1][j].h + 1 : 1,
                          j + 1 < A[i].size() ? table[i][j + 1].w + 1 : 1} :
                    MaxHW{0, 0};
    }
  }

  int max_rect_area = 0;
  for (int i = 0; i < A.size(); ++i) {
    for (int j = 0; j < A[i].size(); ++j) {
      // Process (i, j) if it is feasible and is possible to update
      // max_rect_area
      if (A[i][j] && table[i][j].w * table[i][j].h > max_rect_area) {
        int min_width = numeric_limits<int>::max();
        for (int a = 0; a < table[i][j].h; ++a) {
          min_width = min(min_width, table[i + a][j].w);
          max_rect_area = max(max_rect_area, min_width * (a + 1));
        }
      }
    }
  }
  return max_rect_area;
}
```

If we are looking for the largest feasible square region, we can improve the complexity as follows—we compute the $(h_{i,j}, w_{i,j})$ values as before. Suppose we know the length s of the largest square region that has $A[i+1][j+1]$ as its bottom-left corner. Then the length of the side of the largest square with $A[i, j]$ as its bottom-left corner is at most $s+1$, which occurs iff $h_{i,j} \geq s+1$ and $w_{i,j} \geq s+1$. The general expression for the length is $\min(s+1, h_{i,j}, w_{i,j})$. Note that this is a $O(1)$ time computation. In total, the run time is $O(mn)$, a factor of n better than before.

The calculations above can be sped up by intelligent pruning. For example, if

we already have a feasible 2D subarray of dimensions $H \times W$, there is no reason to process an entry $A[i, j]$ for which $h_{i,j} \leq H$ and $w_{i,j} \leq W$.

```
class MaxHW {
  public:
    int h, w;
};

int max_square_submatrix(const vector<vector<bool>> &A) {
  // DP table stores (h, w) for each (i, j)
  vector<vector<MaxHW>> table(A.size(), vector<MaxHW>(A.front().size()));

  for (int i = A.size() - 1; i >= 0; --i) {
    for (int j = A[i].size() - 1; j >= 0; --j) {
      // Find the largest h such that (i, j) to (i + h - 1, j) are feasible
      // Find the largest w such that (i, j) to (i, j + w - 1) are feasible
      table[i][j] = A[i][j] ?
                    MaxHW{i + 1 < A.size() ? table[i + 1][j].h + 1 : 1,
                          j + 1 < A[i].size() ? table[i][j + 1].w + 1 : 1} :
                    MaxHW{0, 0};
    }
  }

  // A table stores the length of largest square for each (i, j)
  vector<vector<int>> s(A.size(), vector<int>(A.front().size(), 0));
  int max_square_area = 0;
  for (int i = A.size() - 1; i >= 0; --i) {
    for (int j = A[i].size() - 1; j >= 0; --j) {
      int side = min(table[i][j].h, table[i][j].w);
      if (A[i][j]) {
        // Get the length of largest square with bottom-left corner (i, j)
        if (i + 1 < A.size() && j + 1 < A[i + 1].size()) {
          side = min(s[i + 1][j + 1] + 1, side);
        }
        s[i][j] = side;
        max_square_area = max(max_square_area, side * side);
      }
    }
  }
  return max_square_area;
}
```

The largest 2D subarray can be found in $O(nm)$ time and space using a qualitatively different approach. Essentially, we reduce our problem to n instances of the largest rectangle under the skyline problem described in Problem 15.8 on Page 120. First, for each $A[i][j]$ we determine the largest $h_{i,j}$ such that $A[i : i + h_{i,j} - 1][j : j]$ is feasible. (If $A[i][j] = 0$ then $h_{i,j} = 0$.) The h values can be computed in $O(nm)$ time by using DP. Then for each of the n rows, we compute the largest 2D subarray whose bottom edge is on that row in time $O(m)$, using Solution 15.8 on Page 344. This computation can be performed in time $O(n)$ once the $h_{i,j}$ values have been computed. The final solution is the maximum of the n instances.

```
int max_rectangle_submatrix(const vector<vector<bool>> &A) {
```