```
  vector<vector<int>> table(A.size(), vector<int>(A.front().size()));

  for (int i = A.size() - 1; i >= 0; --i) {
    for (int j = A[i].size() - 1; j >= 0; --j) {
      table[i][j] = A[i][j] ? i + 1 < A.size() ? table[i + 1][j] + 1 : 1 : 0;
    }
  }

  // Find the max among all instances of the largest rectangle
  int max_rect_area = 0;
  for (const vector<int> &t : table) {
    max_rect_area = max(max_rect_area, calculate_largest_rectangle(t));
  }
  return max_rect_area;
}
```

The largest square 2D subarray containing only 1s can be computed similarly, with a minor variant on the algorithm in Solution 15.8 on Page 344.

Problem 15.10, *pg. 120*: *Design an algorithm that takes as arguments a 2D array A and a 1D array S, and determines whether S appears in A. If S appears in A, print the sequence of entries where it appears.*

Solution 15.10: We solve this problem using recursion. The Boolean-valued `matchHelper` function takes A, S, a pair of integers (i, j) encoding an entry in A, and a third integer k encoding the offset into S that needs to be matched from (i, j). The function returns `true` if $k = |S|$ or if $S[k] = A[i][j]$ and an entry adjacent to (i, j) matches S with an offset of $k + 1$.

To avoid repeated calls to the same function with the same argument, we cache results. This reduces the complexity to $O(nml)$, where n and m are the dimensions of A, and l is the length of S—we do a constant amount of work within each call to `match_helper` and the number of calls is bounded by the size of the cache. When we find a match, we print it by printing the arguments to the sequence of calls that led to the success.

```
class HashTuple {
  public:
    size_t operator()(const tuple<int, int, int> &t) const {
      return hash<int>()(get<0>(t)) ^ hash<int>()(get<1>(t)) ^
             hash<int>()(get<2>(t));
    }
};

bool match_helper(const vector<vector<int>> &A, const vector<int> &S,
                  unordered_set<tuple<int, int, int>, HashTuple> cache,
                  int i, int j, int len) {
  if (S.size() == len) {
    return true;
  }

  if (i < 0 || i >= A.size() || j < 0 || j >= A[i].size() ||
      cache.find({i, j, len}) != cache.cend()) {
```

```
    return false;
  }

  if (A[i][j] == S[len] &&
      (match_helper(A, S, cache, i - 1, j, len + 1) ||
       match_helper(A, S, cache, i + 1, j, len + 1) ||
       match_helper(A, S, cache, i, j - 1, len + 1) ||
       match_helper(A, S, cache, i, j + 1, len + 1))) {
    return true;
  }
  cache.insert({i, j, len});
  return false;
}

bool match(const vector<vector<int>> &A, const vector<int> &S) {
  unordered_set<tuple<int, int, int>, HashTuple> cache;
  for (int i = 0; i < A.size(); ++i) {
    for (int j = 0; j < A[i].size(); ++j) {
      if (match_helper(A, S, cache, i, j, 0)) {
        return true;
      }
    }
  }
  return false;
}
```

Problem 15.11, ***pg. 120***: *Given two strings, represented as arrays of characters A and B, compute the minimum number of edits needed to transform the first string into the second string.*

Solution 15.11: Let the Levenshtein distance between the two strings A and B be represented by $E(A, B)$. Let's say that a and b are, respectively, the length of strings A and B. We now make two claims:

- If $A[a-1] = B[b-1]$, i.e., the last character of A and B are the same, then $E(A, B) = E(A[0:a-2], B[0:b-2])$. This is because $E(A[0:a-2], B[0:b-2])$ is an upper bound and a lower bound on $E(A, B)$. It is an upper bound, since one way to transform A to B is to transform $A[0:a-2]$ to $B[0:b-2]$. It is a lower bound since we can take a transformation of A to B, and reorder the operations in it to get a transformation of $A[0:a-2]$ into $B[0:b-2]$ that is no longer than the original transformation.
- If $A[a-1] \neq B[b-1]$, i.e., the last two characters of the strings do not match, then

$$E(A, B) = 1 + \min \begin{pmatrix} E(A[0:a-2], B[0:b-2]), \\ E(A[0:a-2], B), \\ E(A, B[0:b-2]) \end{pmatrix}$$

  Clearly, the expression on the right hand side is an upper bound on $E(A, B)$. To show that it is a lower bound, if a smaller sequence transforms A into B, there must be a step where the last character of A becomes the same as the last character of B. This could happen either by inserting a new character at

the end, deleting the last character, or substituting the last character of A with the last character of B. We can reorder the sequence such that this operation happens at the end. The length of the sequence would remain the same and we would still end up with B in the end. If this operation was a "delete", then by deleting this operation, we get a sequence of operations that turn $A[0, a-2]$ into B. If this operation was an "insert", then by dropping this operation, we would have a set of transformations that turn A into $B[0, b-2]$. If this operation was a "substitute", then by discarding this operation, we would have a set of transformations that turn $A[0, a-2]$ into $B[0, b-2]$. In any of those cases, it would be a contradiction if there was a sequence of operations that turned A into B which is smaller than $\min(E(A[0:a-2], B[0:b-2]), E(A[0:a-2], B), E(A, B[0:b-2])) + 1$.

We use the above claims to compute $E(A, B)$. Specifically, we tabulate the values of $E\big(A[0:k], B[0:l]\big)$ for all values of $k < a$ and $l < b$. This takes $O(ab)$ time. We can implement this algorithm using $O(\min(a, b))$ space by reusing space, since we never need more than one row of prior solution at a time. Following is the code in C++.

```
int Levenshtein_distance(string A, string B) {
  // Try to reduce the space usage
  if (A.size() < B.size()) {
    swap(A, B);
  }

  vector<int> D(B.size() + 1);
  // Initialization
  iota(D.begin(), D.end(), 0);

  for (int i = 1; i <= A.size(); ++i) {
    int pre_i_1_j_1 = D[0];  // stores the value of D[i - 1][j - 1]
    D[0] = i;
    for (int j = 1; j <= B.size(); ++j) {
      int pre_i_1_j = D[j];  // stores the value of D[i -1][j]
      D[j] = A[i - 1] == B[j - 1] ?
             pre_i_1_j_1 : 1 + min(pre_i_1_j_1, min(D[j - 1], D[j]));
      // Previous D[i - 1][j] will become the next D[i - 1][j - 1]
      pre_i_1_j_1 = pre_i_1_j;
    }
  }
  return D.back();
}
```

Figure 21.11 on the next page shows the E values for the strings "Carthorse" and "Orchestra". Upper-case and lower-case characters are treated as being different. The Levenshtein distance for this two strings is 8. The longest subsequence which is present in both strings is $\langle r, h, s \rangle$.

ϵ-Variant 15.11.1: Given A and B as above, compute a longest sequence of characters that is a subsequence of A and of B.

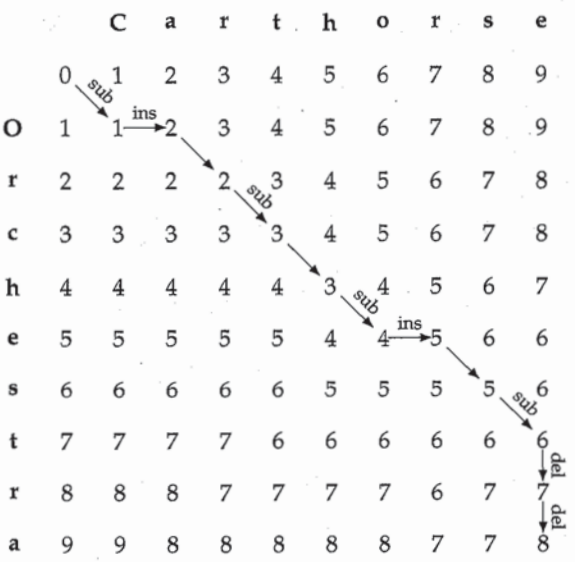


Figure 21.11: The E table for "Carthorse" and "Orchestra".

ϵ-Variant 15.11.2: Given a string A, compute the minimum number of characters you need to delete from A to make the resulting string a palindrome.

Variant 15.11.3: Given a string A and a regular expression r, what is the string in the language of the regular expression r that is closest to A? The distance between strings is the Levenshtein distance specified above.

Problem 15.12, ***pg. 121***: *Given a dictionary and a string s, design an efficient algorithm that checks whether s is the concatenation of a sequence of dictionary words. If such a concatenation exists, your algorithm should output it.*

Solution 15.12: This is a straightforward DP problem. If the input string s has length n, we build a table T of length n such that $T[k]$ is a Boolean indicating whether the substring $s(0, k)$ can be decomposed into a sequence of valid words.

We can build a hash table of all the valid words to determine if a string is a valid word in $O(1)$ time. Then $T[k]$ holds iff one of the following two conditions is true:

1. There exists a $j \in [0, k-1]$ such that $T[j]$ is true and $s(j+1, k)$ is a valid word.
2. Substring $s(0, k)$ is a valid word.

This tells us if we can break a given string into valid words, but does not yield the words themselves. We can obtain the words with a little more book-keeping. In table T, along with the Boolean value, we also store the length of the last word in the string.

```
1 vector<string> word_breaking(const string &s,
2                              const unordered_set<string> &dict) {
```

```
  // T[i] stores the length of the last string which composed of s(0, i)
  vector<int> T(s.size(), 0);
  for (int i = 0; i < s.size(); ++i) {
    // Set T[i] if s(0, i) is a valid word
    if (dict.find(s.substr(0, i + 1)) != dict.cend()) {
      T[i] = i + 1;
    }

    // Set T[i] if T[j] != 0 and s(j + 1, i) is a valid word
    for (int j = 0; j < i && T[i] == 0; ++j) {
      if (T[j] != 0 && dict.find(s.substr(j + 1, i - j)) != dict.cend()) {
        T[i] = i - j;
      }
    }
  }

  vector<string> ret;
  // s can be assembled by valid words
  if (T.back()) {
    int idx = s.size() - 1;
    while (idx >= 0) {
      ret.emplace_back(s.substr(idx - T[idx] + 1, T[idx]));
      idx -= T[idx];
    }
    reverse(ret.begin(), ret.end());
  }
  return ret;
}
```

If we want all possible decompositions, we can store all possible values of j that gives us a correct break with each position. However the number of possible decompositions can be exponential here. This is exemplified by the string "itsitsitsits...".

Problem 15.13, *pg. 121*: *Given text, i.e., a string of words separated by single blanks, decompose the text into lines such that no word is split across lines and the messiness of the decomposition is minimized. Each line can hold no more than L characters. How would you change your algorithm if the messiness is the sum of the messinesses of all but the last line?*

Solution 15.13: Let the text W consist of words $\langle w_0, w_1, \ldots, w_{n-1} \rangle$. Let l_k be the length of w_k. Suppose we know the optimum messiness $M(i)$ for each subtext of the form $W_i = \langle w_0, w_1, \ldots, w_i \rangle$. We find the optimum decomposition for $W_{i+1} = \langle w_0, w_1, \ldots, w_{i+1} \rangle$ as follows. Consider the last line. It will be of the form $W_{j:i+1} = \langle w_j, w_{j+1}, \ldots, w_{i+1} \rangle$, where j could be $i+1$ (word w_{i+1} lies on a line by itself). The optimum messiness for a decomposition is then $M(j-1) + 2^{L - l_{i+1} - \sum_{k=j}^{i}(l_k+1)}$.

We can compute an optimum messiness for W_{i+1} by iterating from $j = i+1$ down to the first f such that $l_{i+1} + \sum_{k=f}^{i}(l_k + 1) > L$. For each value of j we perform constant work to compute the optimum messiness, assuming we form $\sum_{k=j}^{i}(l_k + 1)$ incrementally. Since each line is constrained to L characters, we will not examine more than L words for each word processed, leading to an $O(nL)$ time complexity. Naïvely, the space complexity is $O(n)$ for storing M; however, since we never examine

more than L previous words on the line that w_{i+1} is on, we can reuse space and reduce the additional storage to $O(L)$.

Now we consider the case where the messiness of a decomposition does not include the messiness of the final line. First, we compute M as above. If the final line is $\langle w_j, w_{j+1}, \ldots, w_{n-1} \rangle$, the optimum messiness will be $M(j-1)$. Not more than L (actually $\lfloor \frac{L}{2} \rfloor + 1$) possibilities exist for the final line, and we can compute the optimum messiness once M is computed by considering all the possibilities of the final line. The time and space bounds are the same as before.

```
int find_pretty_printing(const vector<string> &W, const int &L) {
  // Calculate M(i)
  vector<long> M(W.size(), numeric_limits<long>::max());
  for (int i = 0; i < W.size(); ++i) {
    int b_len = L - W[i].size();
    M[i] = min((i - 1 < 0 ? 0 : M[i - 1]) + (1 << b_len), M[i]);
    for (int j = i - 1; j >= 0; --j) {
      b_len -= (W[j].size() + 1);
      if (b_len < 0) {
        break;
      }
      M[i] = min((j - 1 < 0 ? 0 : M[j - 1]) + (1 << b_len), M[i]);
    }
  }

  // Find the minimum cost without considering the last line
  long min_mess = (W.size() >= 2 ? M[W.size() - 2] : 0);
  int b_len = L - W.back().size();
  for (int i = W.size() - 2; i >= 0; --i) {
    b_len -= (W[i].size() + 1);
    if (b_len < 0) {
      return min_mess;
    }
    min_mess = min(min_mess, (i - 1 < 0 ? 0 : M[i - 1]));
  }
  return min_mess;
}
```

Variant 15.13.1: Suppose the messiness of a line ending with b blank characters is defined to be b. Can you solve the messiness minimization problem in $O(n)$ time and $O(1)$ space?

Problem 15.14, ***pg.122:*** *Design an efficient algorithm for computing $\binom{n}{k}$ which has the property that it never overflows if $\binom{n}{k}$ can be represented as a 32-bit integer; assume n and k are integers.*

Solution 15.14: It is tempting to proceed by pairing terms in the numerator and denominator for the expression for $\binom{n}{k}$ that have common factors and cancel them out. This approach is unsatisfactory because of the need to have factorizations.

The binomial coefficients satisfy several identities, the most basic of which is the

addition formula:

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

Various proofs exist of this identity, ranging from the combinatorial interpretation to induction and, finally, direct manipulation of the expressions.

This identity yields a straightforward recursion for $\binom{n}{k}$. The base cases are $\binom{r}{r}$ and $\binom{r}{0}$, both of which are 1. The individual results from the subcalls are 32-bit integers and if $\binom{n}{k}$ can be represented by a 32-bit integer, they can too; so, overflow is not a concern.

The recursion can lead to repeated subcalls and consequently exponential run times, which can be avoided by caching intermediate results as in DP. The number of subproblems is $O(n^2)$ and the results can be combined in $O(1)$ time, yielding an $O(n^2)$ complexity bound.

```
int compute_binomial_coefficients(const int &n, const int &k) {
  vector<vector<int>> table(n + 1, vector<int>(k + 1));
  // Basic case: C(i, 0) = 1
  for (int i = 0; i <= n; ++i) {
    table[i][0] = 1;
  }
  // Basic case: C(i, i) = 1
  for (int i = 1; i <= k; ++i) {
    table[i][i] = 1;
  }

  // C(i, j) = C(i - 1, j) + C(i - 1, j - 1)
  for (int i = 2; i <= n; ++i) {
    for (int j = 1; j < i && j <= k; ++j) {
      table[i][j] = table[i - 1][j] + table[i - 1][j - 1];
    }
  }
  return table[n][k];
}
```

Problem 15.15, *pg. 122*: *You have an aggregate score s and W which specifies the points that can be scored in an individual play. How would you find the number of combinations of plays that result in an aggregate score of s? How would you compute the number of distinct sequences of individual plays that result in a score of s?*

Solution 15.15: Let $W = \{w_0, w_1, \ldots, w_{n-1}\}$ be the possible scores for individual plays. Let X be the set $\{\langle x_0, x_1, \ldots, x_{n-1}\rangle \mid \sum_{i=0}^{n-1} w_i x_i = s\}$. We want to compute $|X|$. Observe that x_0 can take any value in $[0, \left\lfloor \frac{s}{w_0} \right\rfloor]$. Therefore, we can partition X into subsets of vectors of the form $\{\langle x_0, x_1, \ldots, x_{n-1}\rangle\}$, where $0 \leq x_0 \leq \lfloor \frac{s}{w_0} \rfloor$. We can determine the size of each of these subsets by solving the same problem in one fewer dimension—specifically for each x_0 we count the number of combinations in which $s - x_0 w_0$ can be achieved using plays $\{w_1, w_2, \ldots, w_{n-1}\}$. The base case corresponds to computing the number of ways in which a score $t \leq s$ can be formed with the w_{n-1}-score plays, which is 1 or 0, depending on whether w_{n-1} evenly divides t.

The algorithm outlined above has exponential complexity. We can use DP to reduce its complexity—for each $t \leq s$ and $d \in [1, n-1]$ we cache the number of combinations of ways in which $w_d, \ldots, w_{n-1}$ can be used to achieve t. By iterating first over W and then over t, we can reuse space. This is the approach given below.

```
int count_combinations(const int &k, const vector<int> &score_ways) {
  vector<int> combinations(k + 1, 0);
  combinations[0] = 1;  // 1 way to reach 0
  for (const int &score : score_ways) {
    for (int j = score; j <= k; ++j) {
      combinations[j] += combinations[j - score];
    }
  }
  return combinations[k];
}
```

We can compute the number of permutations of scores which lead to an aggregate score of s using recursion. Suppose we know for all $u < v$ the number of permutations of ways in which u can be achieved. We can achieve v points by first scoring $v - w_i$ points followed by w_i. Observe each of these is a distinct permutation. The recursion can be converted to DP by caching the number of permutations yielding t for each $t < s$.

```
int count_permutations(const int &k, const vector<int> &score_ways) {
  vector<int> permutations(k + 1, 0);
  permutations[0] = 1;  // 1 way to reach 0
  for (int i = 0; i <= k; ++i) {
    for (const int &score : score_ways) {
      if (i >= score) {
        permutations[i] += permutations[i - score];
      }
    }
  }
  return permutations[k];
}
```

Variant 15.15.1: Suppose the final score is given in the form (s, s'), i.e., Team 1 scored s points and Team 2 scored s' points. How would you compute the number of distinct scoring sequences which result in this score? For example, if the final score is $(6, 3)$ then Team 1 scores 3, Team 2 scores 3, Team 1 scores 3 is a scoring sequence which results in this score.

Variant 15.15.2: Suppose the final score is (s, s'). How would you compute the maximum number of times the team that lead could have changed? For example, if $s = 10$ and $s' = 6$, the lead could have changed 4 times: Team 1 scores 2, then Team 2 scores 3 (lead change), then Team 1 scores 2 (lead change), then Team 2 scores 3 (lead change), then Team 1 scores 3 (lead change) followed by 3.

Problem 15.16, ***pg. 122*****:** *How many ways can you go from the top-left to the bottom-right in an $n \times m$ 2D array? How would you count the number of ways in the presence of obstacles, specified by an $n \times m$ Boolean 2D array B, where a* `true` *represents an obstacle.*

Solution 15.16: This problem can be solved using a straightforward application of DP: the number of ways to get to (i, j) is the number of ways to get to $(i-1, j)$ plus the number of ways to get to $(i, j-1)$. (If $i = 0$ or $j = 0$, there is only one way to get to (i, j).) The matrix storing the number of ways to get to (i, j) for the configuration in Figure 15.8 on Page 122 is show in Figure 21.12.

```
int number_of_ways(const int &n, const int &m) {
  vector<vector<int>> A(n, vector<int>(m, 0));
  A[0][0] = 1;  // 1 way to start from (0, 0)
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
      A[i][j] += (i < 1 ? 0 : A[i - 1][j]) + (j < 1 ? 0 : A[i][j - 1]);
    }
  }
  return A.back().back();
}
```

We can improve on the above by noting that we do not need an $n \times m$ 2D array, since to fill in the i-th row we do not need values from rows before $i-1$.

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

Figure 21.12: The number of ways to get from $(0, 0)$ to (i, j) for $0 \leq i, j \leq 4$.

An even better solution is based on the fact that each path from $(0, 0)$ to $(n-1, m-1)$ is a sequence of $m-1$ horizontal steps and $n-1$ vertical steps. There are $\binom{n+m-2}{n-1} = \binom{n+m-2}{m-1} = \frac{(n+m-2)!}{(n-1)!(m-1)!}$ such paths. This value can be efficiently computed without division by using DP; refer to Solution 15.14 on Page 353 for details.

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

```
// Given the dimensions of A, n and m, and B, return the number of ways
// from A[0][0] to A[n - 1][m - 1] considering obstacles
int number_of_ways_with_obstacles(const int &n, const int &m,
                                  const vector<vector<bool>> &B) {
  vector<vector<int>> A(n, vector<int>(m, 0));
  // No way to start from (0, 0) if B[0][0] == true
  A[0][0] = !B[0][0];
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
      if (B[i][j] == 0) {
        A[i][j] += (i < 1 ? 0 : A[i - 1][j]) + (j < 1 ? 0 : A[i][j - 1]);
```

```
12        }
13      }
14    }
15    return A.back().back();
16  }
```

Variant 15.16.1: A decimal number is a sequence of digits, i.e., a sequence over $\{0, 1, 2, \ldots, 9\}$. The sequence has to be of length 1 or more, and the first element in the sequence cannot be 0. Call a decimal number D *monotone* if $D[i] \leq D[i+1], 0 \leq i < |D|$. Write a function which takes as input a positive integer k and computes the number of decimal numbers of length k that are monotone.

Variant 15.16.2: Call a decimal number D, as defined above, *strictly monotone* if $D[i] < D[i+1], 0 \leq i < |D|$. Write a function which takes as input a positive integer k and computes the number of decimal numbers of length k that are strictly monotone.

Problem 15.17, *pg. 123*: *Write a program that computes the maximum value of fish a fisherman can catch on a path from the upper leftmost point to the lower rightmost point. The fisherman can only move down or right, as illustrated in Figure 15.9 on Page 123.*

Solution 15.17: The maximum value that the fisherman can obtain while getting to (i, j) is the larger of the maximum value he can catch on his way to $(i-1, j)$ or $(i, j-1)$ plus the value of the fish at (i, j). This is the basis for the DP algorithm shown below. (If $i = 0$ or $j = 0$, there is a unique path to (i, j) from $(0, 0)$.)

```
1  template <typename T>
2  T maximize_fishing(vector<vector<T>> A) {
3    for (int i = 0; i < A.size(); ++i) {
4      for (int j = 0; j < A[i].size(); ++j) {
5        A[i][j] += max(i < 1 ? 0 : A[i - 1][j], j < 1 ? 0 : A[i][j - 1]);
6      }
7    }
8    return A.back().back();
9  }
```

ϵ-Variant 15.17.1: Solve the same problem when the fisherman can begin and end at any point. He must still move down or right. (Note that the value at (i, j) may be negative.)

Problem 15.18, *pg. 123*: *Design an efficient algorithm for computing the maximum margin of victory for the starting player in the pick-up-coins game.*

Solution 15.18: First we note that maximizing the margin of victory for Player F is the same as maximizing the value of the coins picked up by Player F—this follows from the fact that the sum of the coins picked by the two players is equal to the total sum of all the coins.

Call the sum of the coins selected by a player his revenue. Let $f(a,b)$ be the maximum revenue a player can get when it is his turn to play, and the coins remaining on the table are at indices a to b, inclusive, where $a \leq b$. Then $f(a,b)$ satisfies the following equations:

$$f(a,b) = \begin{cases} \max\left(\begin{array}{l} C[a] + \min\left(\begin{array}{l} f(a+2,b), \\ f(a+1,b-1) \end{array} \right), \\ C[b] + \min\left(\begin{array}{l} f(a+1,b-1), \\ f(a,b-2) \end{array} \right) \end{array} \right), & \text{if } a \leq b; \\ 0, & \text{otherwise.} \end{cases}$$

The logic is that the second player will choose the coin that maximizes his profit, which is equivalent to minimizing the profit that the first player will make after the second player makes his move.

We can solve for f using DP—there are $\frac{n(n+1)}{2}$ possible arguments for $f(a,b)$ where n is the number of coins, and the work required to compute f from previously computed values is constant. Hence f can be computed in $O(n^2)$ time.

```
template <typename CoinType>
CoinType pick_up_coins_helper(const vector<CoinType> &C, const int &a,
                              const int &b, vector<vector<CoinType>> &T) {
  if (a > b) {
    return 0;  // base condition
  }

  if (T[a][b] == -1) {
    T[a][b] = max(C[a] + min(pick_up_coins_helper(C, a + 2, b, T),
                             pick_up_coins_helper(C, a + 1, b - 1, T)),
                  C[b] + min(pick_up_coins_helper(C, a + 1, b - 1, T),
                             pick_up_coins_helper(C, a, b - 2, T)));
  }
  return T[a][b];
}

template <typename CoinType>
CoinType pick_up_coins(vector<CoinType> &C) {
  vector<vector<CoinType>> T(C.size(), vector<int>(C.size(), -1));
  return pick_up_coins_helper(C, 0, C.size() - 1, T);
}
```

The DP algorithm applied to the configuration in Figure 4.6 on Page 44, shows a maximum gain for F is 140¢, i.e., whatever strategy S plays, F can guarantee a gain of at least 140¢.

ϵ-Variant 15.18.1: You are given two fixed arrays of numbers A and B, each of length k, and another array C, also of length k. You can assign $C[i]$ to one of $A[i]$, $B[i]$, or 0, subject to the constraint that if $C[i] = A[i]$ then $C[i-1]$ must be assigned 0 if $i > 0$. Design an algorithm that computes an assignment to C that maximizes the sum of the elements in C.

Problem 15.19, *pg. 123*: *Design an algorithm for minimizing power that takes as input a rooted tree and assigns each node to a low or high voltage, subject to the design constraint.*

Solution 15.19: Let $l(r)$ and $h(r)$ be the power consumption of node r under low and high voltages, respectively. Let $L(r)$ be the minimum possible power that can be achieved when we assign a low voltage to r. Let $H(r)$ be the minimum possible power that can be achieved when r is assigned a high voltage.

Denote the set of all nodes that are inputs to r by $I(r)$. Then the following recurrence relationships must hold for L and H:

$$\begin{aligned} L(r) &= l(r) + \sum_{c \in I(r)} H(c), \\ H(r) &= h(r) + \sum_{c \in I(r)} \min\big(L(c), H(c)\big). \end{aligned}$$

Using these equations, we can tabulate the values of L and H for all nodes. The desired solution is the minimum of the values of L and H for the root of the tree. Since we do a constant number of operations per node, the overall complexity is $O(n)$, where n is the number of nodes.

Problem 15.20, *pg. 124*: *Implement cutpoint selection to minimize the number of nodes in the two-dimensional tree representing an image.*

Solution 15.20: We maintain a cache mapping each subrectangle of the image to the minimum number of nodes needed when representing it as a two-dimensional tree. Computing a cache entry entails first checking if the corresponding image is monochromatic. If it is not monochromatic the algorithm tries all possible choices for the cutpoint. Given a cutpoint, the optimum tree is determined by making four lookups into the cache.

The cache has no more than $\binom{(m+1)\times(n+1)}{2} = O(n^2m^2)$ entries, which is an upper bound on the number of recursive calls. The time spent within a call is dominated by iterating through the different choices of the cutpoint, i.e., $O(mn)$, leading to a $O(m^3n^3)$ time complexity for the entire algorithm.

If large monochromatic regions are present in the image, the run time can be sped up by computing the number of 1s in $P[0 : i, 0 : j]$, for $0 \leq i < m, 0 \leq j < n$. Call this quantity $N[i, j]$. It can be computed in $O(nm)$ time by iterating through rows in ascending order. Precomputing N accelerates checking subrectangles in P for monochromaticity: the number of 1s in $P[i : i', j : j']$ is $N[i', j'] - N[i-1, j'] - N[i', j-1] + N[i-1, j-1]$.

```
1 class Point {
2   public:
3     int i, j;
4
5     const bool operator>(const Point &that) const {
6       return i > that.i || j > that.j;
7     }
8
```

```
    // Equal function for hash
    const bool operator==(const Point &that) const {
      return i == that.i && j == that.j;
    }
};

// Hash function for Point
class HashPoint {
  public:
    const size_t operator()(const Point &p) const {
      return hash<int>()(p.i) ^ hash<int>()(p.j);
    }
};

class TreeNode {
  public:
    int node_num;  // stores the number of node in its subtree

    Point lowerLeft, upperRight;

    // Store the SW, NW, NE, and SE rectangles if color is mixed
    vector<shared_ptr<TreeNode>> children;
};

bool is_monochromatic(const vector<vector<int>> &image_sum,
                      const Point &lower_left, const Point &upper_right) {
  int pixel_sum = image_sum[upper_right.i][upper_right.j];
  if (lower_left.i >= 1) {
    pixel_sum -= image_sum[lower_left.i - 1][upper_right.j];
  }
  if (lower_left.j >= 1) {
    pixel_sum -= image_sum[upper_right.i][lower_left.j - 1];
  }
  if (lower_left.i >= 1 && lower_left.j >= 1) {
    pixel_sum += image_sum[lower_left.i - 1][lower_left.j - 1];
  }
  return pixel_sum == 0 ||  // totally white
         pixel_sum == (upper_right.i - lower_left.i + 1) *  // totally black
                      (upper_right.j - lower_left.j + 1);
}

shared_ptr<TreeNode> calculate_optimal_2D_tree_helper(
    const vector<vector<int>> &image, const vector<vector<int>> &image_sum,
    const Point &lower_left, const Point &upper_right,
    unordered_map<Point,
                  unordered_map<Point, shared_ptr<TreeNode>, HashPoint>,
                  HashPoint> &table) {
  // Illegal rectangle region, returns empty node
  if (lower_left > upper_right) {
    return shared_ptr<TreeNode>(new TreeNode{0, lower_left, upper_right});
  }

  if (table[lower_left].find(upper_right) == table[lower_left].cend()) {
    if (is_monochromatic(image_sum, lower_left, upper_right)) {
      shared_ptr<TreeNode> p(new TreeNode{1, lower_left, upper_right});
```

```
      table[lower_left][upper_right] = p;
    } else {
      shared_ptr<TreeNode>
        p(new TreeNode{numeric_limits<int>::max(), lower_left, upper_right});
      for (int s = lower_left.i; s <= upper_right.i + 1; ++s) {
        for (int t = lower_left.j; t <= upper_right.j + 1; ++t) {
          if ((s != lower_left.i && s != upper_right.i + 1) ||
              (t != lower_left.j && t != upper_right.j + 1)) {
            vector<shared_ptr<TreeNode>> children = {
              // SW rectangle
              calculate_optimal_2D_tree_helper(image, image_sum, lower_left,
                                               Point{s - 1, t - 1}, table),
              // NW rectangle
              calculate_optimal_2D_tree_helper(image, image_sum,
                                               Point{lower_left.i, t},
                                               Point{s - 1, upper_right.j},
                                               table),
              // NE rectangle
              calculate_optimal_2D_tree_helper(image, image_sum, Point{s, t},
                                               upper_right, table),
              // SE rectangle
              calculate_optimal_2D_tree_helper(image, image_sum,
                                               Point{s, lower_left.j},
                                               Point{upper_right.i, t - 1},
                                               table)};

            int node_num = 1;  // itself
            for (shared_ptr<TreeNode> &child : children) {
              node_num += child->node_num;
              // Remove the child contains no node
              if (child->node_num == 0) {
                child = nullptr;
              }
            }
            if (node_num < p->node_num) {
              p->node_num = node_num, p->children = children;
            }
          }
        }
      }
      table[lower_left][upper_right] = p;
    }
  }
  return table[lower_left][upper_right];
}

shared_ptr<TreeNode> calculate_optimal_2D_tree(
    const vector<vector<int>> &image) {
  vector<vector<int>> image_sum(image);
  for (int i = 0; i < image.size(); ++i) {
    partial_sum(image_sum[i].cbegin(), image_sum[i].cend(),
                image_sum[i].begin());
    for (int j = 0; i > 0 && j < image[i].size(); ++j) {
      image_sum[i][j] += image_sum[i - 1][j];
    }
```

```
  }

  unordered_map<Point,
                unordered_map<Point, shared_ptr<TreeNode>, HashPoint>,
                HashPoint> table;
  return calculate_optimal_2D_tree_helper(image, image_sum, Point{0, 0},
                                          Point{static_cast<int>(
                                            image.size() - 1),
                                            static_cast<int>(
                                            image[0].size() - 1)},
                                          table);
}
```

Variant 15.20.1: Define the intersection of two images that have the same dimension to be the image that is white wherever either image is white and black otherwise. Write a function that takes two two-dimensional trees representing images, and returns a two-dimensional tree that represents their intersection.

Problem 15.21, ***pg. 125:*** *Given n queries, compute an order in which to process queries that minimizes the total waiting time.*

Solution 15.21: Consider a schedule in which the i-th client is the c_i-th one to be processed; the j-th client processed corresponds to client with ID d_j. Then the waiting time for the i-th client is $\sum_{j<c_i} t_{d_j}$. Hence sum of all the wait times would be

$$\sum_{i=1}^{n} T_i = \sum_{i=1}^{n} \sum_{j<c_i} t_{d_j} = \sum_{i=1}^{n} t_i(n - c_i).$$

Since we want to minimize the total wait time for all the queries and each c_i takes a value between 1 and n, it follows that the queries that take the smallest time must get served first. Hence we should sort the queries by their service time and then process them in the order of non-decreasing service time.

```
template <typename T>
T minimum_waiting_time(vector<T> service_time) {
  // Sort the query time in increasing order
  sort(service_time.begin(), service_time.end());

  T waiting = 0;
  for (int i = 0; i < service_time.size(); ++i) {
    waiting += service_time[i] * (service_time.size() - (i + 1));
  }
  return waiting;
}
```

Problem 15.22, ***pg. 125:*** *Design an algorithm that computes the least number of tutors needed to schedule a set of requests.*

Solution 15.22: We schedule tutors greedily: as soon as there is a request that cannot be handled by the previously assigned tutors, we choose a new tutor.

This scheme is simple to implement, but it is not completely trivial to prove that it is optimum.

We will use the notion of slack to prove optimality. Suppose the last tutor scheduled begins at time t, and he completes the last lesson he was assigned at time t'. Then the *slack* in the schedule is defined to be $t + 120 - t'$ minutes.

Consider a set of requests $j_1, \ldots, j_n$ such that the requests are ordered by the time they need to be completed. We claim that greedy scheduling is optimum.

Proof:

We use induction on the number of requests. For our induction hypothesis, in addition to the claim that the number of tutors is minimized, we also claim that the schedule maximizes the slack. For $n = 1$, the greedy algorithm sends exactly one tutor at the start time of the request. Clearly this is the strategy that uses the minimum number of tutors and no more slack is possible.

Assume the induction hypothesis holds for all $n \leq k$. We will now prove it for $n = k+1$. Consider the requests $j_1, \ldots, j_k$ sorted by their start time. When the next request j_{k+1} is added to the list, either it can be covered by an existing tutor, which is determined by the slack, or it may require a new tutor.

If the new request can be covered by the slack, using an existing tutor must be optimum with respect to the number of tutors. If we needed at least m tutors to cover the first k requests, we cannot cover the $k + 1$ requests with fewer tutors. Also, since the schedule for the first k requests maximized the slack, we cannot have a schedule with m tutors that covers all $k + 1$ requests and has more slack.

If an additional tutor is needed for the $k + 1$-th request, it must be that the m-th tutor did not have the slack to cover the last request. If there exists another way to cover the requests with m or less tutors, then we can use the same set of tutors to cover the first k requests and get a bigger slack, which contradicts the induction hypothesis. Since the $(m+1)$-th tutor will start exactly when the last request starts, this schedule is slack maximizing.

Problem 15.23, *pg. 126*: *Design an algorithm that takes as input a pair of arrays specifying jobs per task and server capacities, and returns an assignment of jobs to servers for which all tasks complete within one unit time. No server may process more than one job for a given task. If no such assignment exists, your algorithm should indicate that.*

Solution 15.23: Let X be an $m \times n$ Boolean 2D array. Define X to *realize* (T, S) if for all i, $T[i] = \sum_{j=0}^{n-1} X[i, j]$ and for all j, $S[j] \geq \sum_{i=0}^{m-1} X[i, j]$. Clearly X can be viewed as an assignment of jobs to servers; since X is Boolean, we never have more than one job from the same task assigned to the same server.

Let s be the index of a maximal element S, i.e., a server with maximum capacity. We claim that (T, S) is realizable iff it can be realized by a 2D array X in which server s has the greatest load, i.e., Column s has the maximum number of 1s out of all the columns of X.

Proof:

If in some legal assignment the server s is not a most loaded server, look at a server s' that is most loaded. There must exist a task t which utilizes s' but not s, since otherwise s would be a most loaded server. We can then move that job from t to s without violating any constraints. (Note that s must have residual capacity, since otherwise it would not be the server with maximum capacity.)

The above reasoning can be used to develop an algorithm for building the 2D array X that implements the assignment, if a valid assignment exists: initialize X to all 0s. Sort servers in decreasing order of capacity. Starting with a server with maximum capacity, greedily assign jobs from tasks to that server. Iterate until there all tasks have been assigned or no servers remain. In the latter case, the above theorem guarantees that no assignment of jobs to servers satisfying the constraints.

In the code listed below, we use a straightforward implementation of the greedy algorithm. Specifically, we iterate over the servers in an outer loop and tasks in an inner loop. The time complexity is $O(nm)$. The code includes some heuristics for early detection of infeasibility, such as a single task requiring more servers than are available, or the aggregated tasks exceeding the total available server capacity.

```
const bool comp(const pair<int, int> &a, const pair<int, int> &b) {
  return a.second > b.second;
}

vector<vector<bool>> find_feasible_job_assignment(const vector<int> &T,
                                                  const vector<int> &S) {
  int T_total = accumulate(T.cbegin(), T.cend(), 0),  // aggregated work units
      S_total = accumulate(S.cbegin(), S.cend(), 0,
                           [&T](const int &x, const int &y) -> int {
                             return x + min(y, static_cast<int>(T.size()));
                           });  // tighter bound of server capacity
  if (T_total > S_total || *max_element(T.cbegin(), T.cend()) > S.size()) {
    return {};  // too many jobs or one task needs too many servers
  }

  vector<pair<int, int>> T_idx_data, S_idx_data;
  for (int i = 0; i < T.size(); ++i) {
    T_idx_data.emplace_back(i, T[i]);
  }
  for (int j = 0; j < S.size(); ++j) {
    S_idx_data.emplace_back(j, S[j]);
  }

  sort(S_idx_data.begin(), S_idx_data.end(), comp);
  vector<vector<bool>> X(T.size(), vector<bool>(S.size(), false));
  for (int j = 0; j < S_idx_data.size(); ++j) {
    if (S_idx_data[j].second < T_idx_data.size()) {
      nth_element(T_idx_data.begin(),
                  T_idx_data.begin() + S_idx_data[j].second, T_idx_data.end(),
                  comp);
    }

    // Greedily assign jobs
```

```
    int size = min(static_cast<int>(T_idx_data.size()), S_idx_data[j].second);
    for (int i = 0; i < size; ++i) {
      if (T_idx_data[i].second) {
        X[T_idx_data[i].first][S_idx_data[j].first] = true;
        --T_idx_data[i].second;
        --T_total;
      }
    }
  }
  if (T_total) {
    return {};  // still some jobs remain, no feasible assignment
  }
  return X;
}
```

Problem 15.24, *pg. 126*: *You have n users with unique hash codes h_0 through h_{n-1}, and m servers. The hash codes are ordered by index, i.e., $h_i < h_{i+1}$ for $i \in [0, n-2]$. User i requires b_i bytes of storage. The values $k_0 < k_1 < \cdots < k_{m-2}$ are used to assign users to servers. Specifically, the user with hash code c gets assigned to the server with the lowest ID i such that $c \leq k_i$, or to server $m-1$ if no such i exists. The load on a server is the sum of the bytes of storage of all users assigned to that server. Compute values for $k_0, k_1, \ldots, k_{m-1}$ that minimizes the load on the most heavily loaded server.*

Solution 15.24: Let $L(p, q)$ be the maximum load on a server when users with hash codes h_0 through h_p are assigned to Servers 0 through q in an optimum way, i.e., when the maximum load is minimized. Then following recurrence holds:

$$L(p, q) = \min_{x \in [0,p]} \left(\max \left(L(x, q-1), \sum_{i=x+1}^{p} b_i \right) \right)$$

In other words, to find the optimum assignment of users with hash codes $\{h_0, h_1, \ldots, h_p\}$ to q servers, we find x such that if we assign the first $x+1$ users optimally to $q-1$ servers and the remainder to Server q, the maximum load on a given server is minimized.

We can use the recurrence to tabulate the values in L till we get $L(n-1, m-1)$. The base case corresponds to entries of the form $L(p, 0)$, in which case the maximum load is $\sum_{i=0}^{p} b_i$. The time complexity to compute each $L(i, j)$ is $O(n)$, so the overall complexity to compute $L(n-1, m-1)$ is $O(n^2 m)$.

A qualitatively different approach, based on the greedy method, is to check whether $k_0, k_1, \ldots, k_{m-1}$ can be chosen so as to ensure that no server stores more than b bytes. For a given b, this can easily be done—iterate through the n users in the order of their hash codes, and assign them to the servers greedily, i.e., assign users to servers, moving on the next server when the capacity of the current server is exceeded. We can perform binary search to get the minimum b, and the corresponding values for $k_0, k_1, \ldots, k_{m-1}$. The time complexity of the approach is $O(n \log W)$, where W is the total number of bytes that are to be stored, i.e., $W = \sum_{i=0}^{m-1} b_i$.

This approach is much faster in practice: when $n = 10000$, $m = 100$, and loads are uniform integer random variables in the range $[1, 100]$, the $O(n^2m)$ DP algorithm takes over an hour on our machine. In contrast, binary search for w took 0.1 seconds. Furthermore, binary search requires no additional storage beyond that needed to store the final result. The complexity of the code is also greatly reduced compared to the DP algorithm.

```
bool greedy_assignment(const vector<int> &user_file_size,
                       const int &server_num, const int &limit,
                       vector<int> &assign_res) {
  int server_idx = 0;
  for (const int &file : user_file_size) {
    while (server_idx < server_num && file + assign_res[server_idx] > limit) {
      ++server_idx;
    }

    if (server_idx >= server_num) {
      return false;
    } else {
      assign_res[server_idx] += file;
    }
  }
  return true;
}

vector<int> decide_load_balancing(vector<int> user_file_size,
                                  const int &server_num) {
  // Uses binary search to find the assignment with minimized maximum load
  int l = 0,
      r = accumulate(user_file_size.cbegin(), user_file_size.cend(), 0);
  vector<int> feasible_assignment;
  while (l <= r) {
    int m = l + ((r - l) >> 1);
    vector<int> assign_res(server_num, 0);
    bool is_feasible = greedy_assignment(user_file_size, server_num, m,
                                         assign_res);
    if (is_feasible) {
      feasible_assignment = assign_res;
      r = m - 1;
    } else {
      l = m + 1;
    }
  }
  return feasible_assignment;
}
```

Problem 15.25, ***pg. 126***: *Implement first-fit to run in $O(n \log n)$ time.*

Solution 15.25: This can be trivially done in $O(n^2)$ time if we do a linear scan through the boxes for each new object to find the first box where it would fit.

To speed things up, we maintain a *"tournament tree"* data structure which captures remaining capacities as well as the box sequence. The tournament tree is organized

as a complete binary tree. Each leaf corresponds to a box, and the leaf order from left-to-right is the box sequence.

For simplicity, we assume $n = 2^k$ for some k. We start with as many boxes as items. For each internal node v, we record the largest remaining capacity $v.max$ that exists amongst the boxes corresponding to leaves at the subtree rooted at v. Finding the first box that has capacity c can be done recursively. Let r be the root: if $r.left.max \geq c$, we search the root's left child, otherwise, we must use a box on the right side. (Note that $r.max \geq c$ always holds, since we have n boxes and n items.)

After an item is placed in a box, the remaining capacity of the box changes. The only internal nodes v whose $v.max$ changes are those that are ancestors of the leaf corresponding to that box. Updating these nodes consists of simply updating their $v.max$ to the maximum of the $v.max$ of their children in a bottom-up order.

A complete binary tree on n leaves has exactly $n-1$ internal nodes. The height of a complete binary tree on $2n-1$ nodes is $\lceil \log(2n-1) \rceil$, implying the update runs in $O(\log n)$ time.

Building the initial tournament tree has time complexity $O(n)$. Each item takes $O(\log n)$ time to process, leading to the desired $O(n \log n)$ time bound.

```
template <typename ItemType, typename CapacityType>
class TournamentTree {
  private:
    class TreeNode {
      public:
        CapacityType cap;  // leaf: remaining capacity in the box
                           // non-leaf: max remaining capacity in the subtree
        vector<ItemType> items;  // stores the items in the leaf node
    };

    // Store the complete binary tree. For tree[i],
    // left subtree is tree[2i + 1], and right subtree is tree[2i + 2].
    vector<TreeNode> tree;

    // Recursively inserts item in tournament tree
    void insertHelper(const int &idx, const ItemType &item,
                      const CapacityType &cap) {
      int left = (idx << 1) + 1, right = (idx << 1) + 2;
      if (left < tree.size()) {  // internal node
        insertHelper(tree[left].cap >= cap ? left : right, item, cap);
        tree[idx].cap = max(tree[left].cap, tree[right].cap);
      } else {  // leaf node
        tree[idx].cap -= cap, tree[idx].items.emplace_back(item);
      }
    }

  public:
    // n items, and each box has unit_cap
    TournamentTree(int n, const CapacityType &unit_cap) :
      // Complete tree with n leafs has 2n - 1 nodes
      tree(vector<TreeNode>((n << 1) - 1, {unit_cap})) {}

    void insert(const ItemType &item, const CapacityType &item_cap) {
```

```
34        insertHelper(0, item, item_cap);
35      }
36  };
```

The approach is illustrated in Figure 21.13. Assume sizes have been normalized with respect to the box capacity. In Figure 21.13(a), the tournament tree depicted corresponds to six boxes, and five insertions of items of normalized sizes 0.6, 0.6, 0.55, 0.8, and 0.5 in that order. Now if an item v of size 0.45 is to be inserted, we first see that the root node a has a capacity at least 0.45, indicating it is possible to fit that item. We then check a's left child, node b. It has capacity 0.45, which means we can fit v in a box in the subtree rooted at b. Since node b's left child has capacity 0.4, we are forced to go right to c. We first examine c's left child, d, which corresponds to a box with capacity 0.45, meaning we can pack v in d. We then update the tournament tree as shown Figure 21.13(b).

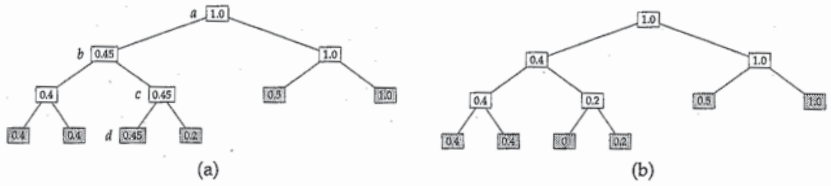


Figure 21.13: A tournament tree before and after inserting an item of size 0.45.

Problem 15.26, ***pg. 127***: *Given a set of symbols with corresponding frequencies, find a code book that has the smallest average code length.*

Solution 15.26: Huffman coding yields an optimum solution to this problem. (There may be other optimum codes as well.) Huffman coding proceeds in three steps:

(1.) Sort characters in increasing order of frequencies and create a binary tree node for each character. Denote the set just created by S.

(2.) Create a new node n whose children are the two nodes with smallest frequencies and assign n's frequency to be the sum of the frequencies of its children.

(3.) Remove the children from S and add n to S. Repeat from Step (2.) till S consists of a single node, which is the root.

Mark all the left edges with 0 and the right edges with 1. The path from the root to a leaf node yields the bit string encoding the corresponding character.

We use a min-heap of candidate nodes to represent S. Since each invocation of Steps (2.) and (3.) requires two *extract-min* and one *insert* operation, we can find the Huffman codes in $O(n \log n)$ time. Here is an implementation of Huffman coding.

```
1  class Symbol {
2    public:
3      char c;
4      double prob;
5      string code;
6  };
```

```
7
8  class BinaryTree {
9    public:
10     double prob;
11     shared_ptr<Symbol> s;
12     BinaryTree *left, *right;
13 };
14
15 class Compare {
16   public:
17     const bool operator()(const BinaryTree* lhs,
18                           const BinaryTree* rhs) const {
19       return lhs->prob > rhs->prob;
20     }
21 };
22
23 // Traverse tree and assign code
24 void assign_huffman_code(const BinaryTree* r, const string &s) {
25   if (r) {
26     // This node (i.e.,leaf) contains symbol
27     if (r->s) {
28       r->s->code = s;
29     } else {  // non-leaf node
30       assign_huffman_code(r->left, s + '0');
31       assign_huffman_code(r->right, s + '1');
32     }
33   }
34 }
35
36 void Huffman_encoding(vector<Symbol> &symbols) {
37   // Initially assign each symbol into min->heap
38   priority_queue<BinaryTree*, vector<BinaryTree*>, Compare> min_heap;
39   for (Symbol &s : symbols) {
40     min_heap.emplace(new BinaryTree{s.prob, shared_ptr<Symbol>(&s),
41                                     nullptr, nullptr});
42   }
43
44   // Keep combining two nodes until there is one node left
45   while (min_heap.size() > 1) {
46     BinaryTree* l = min_heap.top();
47     min_heap.pop();
48     BinaryTree* r = min_heap.top();
49     min_heap.pop();
50     min_heap.emplace(new BinaryTree{l->prob + r->prob, nullptr, l, r});
51   }
52
53   // Traverse the binary tree and assign code
54   assign_huffman_code(min_heap.top(), string());
55 }
```

Applying this algorithm to the frequencies for English characters presented in Table 15.1 on Page 127 yields the Huffman tree in Figure 21.14 on the following page. The path from root to leaf yields that character's Huffman code, which is listed in Table 21.1 on the next page. For example, the codes for t, e, and z are 000, 100, and

001001000, respectively.

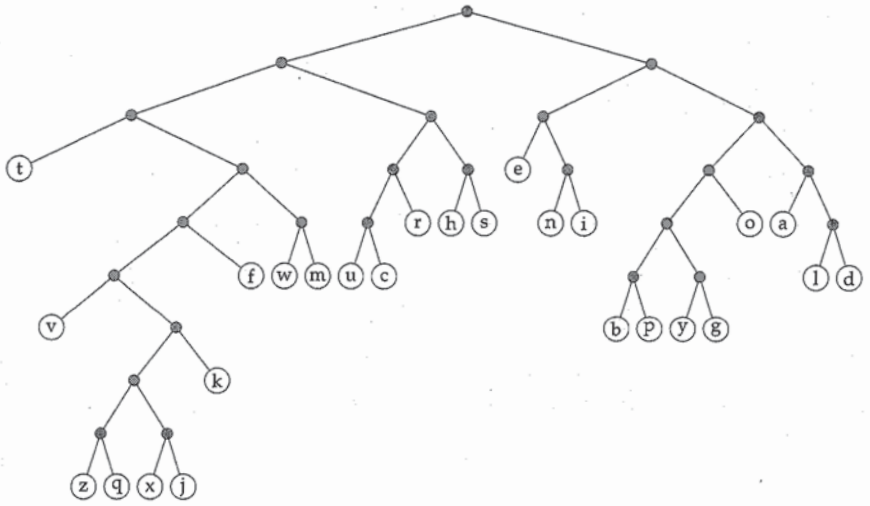


Figure 21.14: A Huffman tree for the English characters, assuming the frequencies given in Table 15.1 on Page 127.

Table 21.1: Huffman codes for English characters, assuming the frequencies given in Table 15.1 on Page 127.

Character	Huffman code	Character	Huffman code	Character	Huffman code
a	1110	j	001001011	s	0111
b	110000	k	0010011	t	000
c	01001	l	11110	u	01000
d	11111	m	00111	v	001000
e	100	n	1010	w	00110
f	00101	o	1101	x	001001010
g	110011	p	110001	y	110010
h	0110	q	001001001	z	001001000
i	1011	r	0101		

The codebook is explicitly given in Table 21.1. The average code length for this coding is 4.205. In contrast, the trivial coding takes $\lceil \log 26 \rceil = 5$ bits for each character.

Although it is unlikely that a rigorous proof of optimality would be asked in an interview setting, we give a proof by induction on the number of characters.

Proof:

For a single character, Huffman codes are trivially optimum. Let's say that for any distribution of frequencies among n characters, Huffman codes are optimum. Given this assumption, we will prove it is true for $n+1$ characters. We denote the frequency of character c by $f(c)$.

Suppose there exists an encoding that has a smaller average length of code for some frequency distribution for $n+1$ characters.

For any encoding, we can map the codes to a binary tree by identifying the

null string with root and adding a left edge for each 0 and a right edge for each 1.

We make several observations about a tree T corresponding to an optimum encoding:

- Each character must map to a leaf node; otherwise, the coding will violate our requirements on code prefixes.
- There cannot be a non-leaf node in T that has fewer than two children (otherwise, we can delete that node, bring its child one level up, and hence reduce the average code length).
- If we sort the leaves of T by their depths, the two deepest leaves must have the same depth (since the parent of the leaf with the longest path must have another child).
- The two deepest leaves in T must be assigned to two characters with smallest frequencies (otherwise, we can swap characters and achieve smaller average code length).
- Suppose we remove the two smallest frequency characters s and t and replace them with a new character u that has its frequency equal to $f(s) + f(t)$. Then the optimum prefix coding for this set C' of characters and corresponding frequencies must have the same average code length as the tree T' that results from deleting the two lowest frequency leaves in T (which as previously argued can be taken as siblings) and assigning their parent's frequency to be the sum of these two frequencies. Otherwise, if a tree S' for C' has lower average code length than T', we can create a tree S from S' by replacing the leaf corresponding to u with an internal node with two children corresponding to s and t. The average code length for the tree S is $f(s) + f(t)$ plus the average code length of the tree S', which, by hypothesis, has a lower average code length than T'. Since by construction, the average code length of T is $f(s) + f(t)$ plus the average code length of T', this contradicts the assumed optimality of T.

Now suppose the characters have frequencies $p_1 \geq p_2 \geq \cdots \geq p_{n+1}$. Let $A(p_1, \ldots, p_{n+1})$ be the optimum average code length for this frequency distribution and $\mathcal{H}(p_1, \ldots, p_{n+1})$ be the average code length for Huffman coding.

From the above observations, it follows that

$$A(p_1, \ldots, p_{n+1}) = A(p_1, \ldots, p_{n-1}, p_n + p_{n+1}) + (p_n + p_{n+1}).$$

From the construction of Huffman codes we know that

$$\mathcal{H}(p_1, \ldots, p_{n+1}) = \mathcal{H}(p_1, \ldots, p_{n-1}, p_n + p_{n+1}) + (p_n + p_{n+1}).$$

By our inductive assumption, $\mathcal{H}(p_1, \ldots, p_{n-1}, p_n + p_{n+1}) = A(p_1, \ldots, p_{n-1}, p_n + p_{n+1})$. Hence $\mathcal{H}(p_1, \ldots, p_{n+1}) = A(p_1, \ldots, p_{n+1})$. In other words, Huffman coding is optimum for $n + 1$ characters if it is optimum for n characters.

Problem 15.27, pg. 127: *How would you efficiently assign to each node u a new weight $w'(u)$ such that (1.) each root-to-leaf path has the same weight W^*, (2.) for all nodes u, $w'(u) \geq w(u)$, and (3.) $\sum_{u \in nodes(T)} w'(u)$ is minimum? See Figure 15.11 on Page 128 for an example.*

Solution 15.27: Let $\mu(u)$ be the weight of a u-to-leaf path with maximum weight under the weighing function w. Since $\mu(u) = w(u) + \max_{v \in children} \mu(v)$, we can compute μ for all vertices in the tree in a single pass in $O(n)$ time.

Observe that the root r has the largest μ-value. We will show how to construct new weights in which *every* root-to-leaf path has weight $\mu(r)$, while satisfying the constraints. Since we seek to minimize the total weight, we would never use a value greater than $\mu(r)$ for W^*.

We compute the new weights with a top-down traversal. We use a global variable s that records the updated weight of the path from the root up to, but not including the current node u. We assign the new weight of u, $w'(u)$, to $w(u) + \mu(r) - (s + \mu(u))$, update s, and recurse on the children. When we finish processing a node, we return to the parent, again updating the value of s.

Now we justify the algorithm. The term $\mu(r) - (s + \mu(u))$ is the "slack" at the node u. Basically, it tells us how much weight needs to be added to the heaviest u-to-leaf path to make its weight $\mu(r)$. Optimality cannot be lost in updating $w'(u)$ by the slack amount, since otherwise the slack will have to be added across u's subtrees, which will increase $\sum_{u \in nodes(T)} w'(u)$ if u has more than one child, and not change it if it has a single child.

ϵ-Variant 15.27.1: Compute a weighing function w' which minimizes $\sum_{u \in nodes(T)} w'(u)$ subject to the constraint that for each leaf l the weight of the path from root to l under w' equals the weight of the path from root to l under w.

Problem 15.28, pg. 128: *Devise an efficient algorithm that takes as input a set P of people and a set $F \subset P \times P$ of pairs of people and returns a largest subset of P within which each individual knows three or more other members of P and does not know three or more other members of P. The "knows" relation is not necessarily symmetric or transitive.*

Solution 15.28: We compute the optimum invitation list by iteratively removing people who cannot meet Leona's constraints until there is no one left to remove—the remaining set is the unique maximum set of people that Leona can invite.

Specifically, we iteratively remove anyone who knows fewer than three people in the current set and anyone who has fewer than three people they do not know in the current set. The process must converge since we start with a finite number of people and remove at least one person in each iteration. The remaining set satisfies Leona's constraints by construction.

It remains to show that the remaining set is maximum. We do this by proving that people who are removed could never be in a set that satisfies the constraints. We use induction on the k-th person removed.

Proof:

Let p_1 be the first person to be removed. Either p_1 knows fewer than three

people in the entire set or p_1 does not know fewer than three people in the entire set. Clearly p_1 cannot belong to any set that satisfies the constraints.

Inductively assume the first $i-1$ persons removed could not belong to any set that satisfies the constraints. Consider p_i, the i-th person removed. It must be that either fewer than three people know p_i in the current set or p_i does not know fewer than three people in the current set. But by induction, the current set includes any set satisfying Leona's constraints, so p_i cannot belong to a set satisfying Leona's constraints, and induction goes through.

Problem 15.29, *pg. 128*: *Let $G = (V, E)$ be an undirected graph. A two-coloring of G is a function assigning each vertex of G to* black *or* white. *Call a two-coloring* diverse *if each vertex has at least half its neighbors opposite in color to itself. Does every graph have a diverse coloring? How would you compute a diverse coloring, if it exists?*

Solution 15.29:

Here is a simple greedy algorithm for computing a diverse coloring. Start with an arbitrary two-coloring. If it is diverse, we are done. Define a vertex to be diverse under a coloring if at least half its neighbors are opposite in color to itself. Search for a non-diverse vertex v and flip v's color. Stop when the coloring is diverse.

It is implicit in the above description that every graph has a diverse coloring. Proving that the algorithm converges is slightly tricky, since a flip may increase the number of non-diverse vertices—see Figures 21.15(a) on the following page and 21.15(b) on the next page for an example. Hence we cannot prove that the greedy algorithm convergence by showing the number of non-diverse vertices keeps decreasing.

The key to proving convergence of the greedy algorithm is to focus on edges instead of vertices. For a given coloring, define an edge to be diverse if it connects vertices of different colors. We prove that the greedy approach does converge to a diverse coloring by counting the number of diverse edges.

Proof:

Suppose x is not diverse. Without loss of generality, suppose x is white. Then by changing x's color to black, the number of diverse edges strictly increases (since x had more white neighbors than black neighbors, and the diversity of other edges is unchanged).

Therefore the greedy algorithm must eventually converge, since the number of edges is finite. When it stops, all vertices are diverse.

This process is illustrated in Figure 21.15 on the following page. The coloring in Figure 21.15(a) on the next page is not diverse, since D is colored white, and has three white and two black neighbors; it is the only non-diverse vertex. Flipping D's color makes E non-diverse (Figure 21.15(b) on the following page). However, the number of non-diverse edges reduces from five to four. Flipping E's color reduces the number of non-diverse edges to three, and results in a diverse coloring (Figure 21.15(c) on the next page).

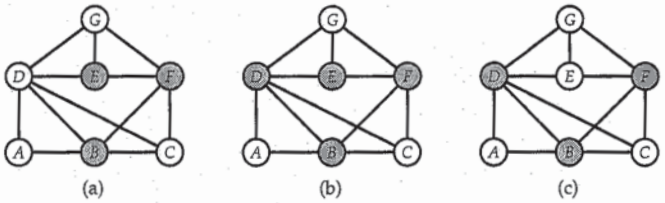


Figure 21.15: Computing a diverse coloring.

Problem 16.1, *pg. 132*: *Given a 2D array of black and white entries representing a maze with designated entrance and exit points, find a path from the entrance to the exit, if one exists.*

Solution 16.1: Model the maze as an undirected graph. Each vertex corresponds to a white pixel. We will index the vertices based on the coordinates of the corresponding pixel; so, vertex $v_{i,j}$ corresponds to the 2D array entry (i, j). Use edges to model adjacent pixels: $v_{i,j}$ is connected to vertices $v_{i+1,j}$, $v_{i,j+1}$, $v_{i-1,j}$, and $v_{i,j-1}$, assuming these vertices exist—vertex $v_{a,b}$ does not exist if the corresponding pixel is black or the coordinates (a, b) lie outside the image.

Now, run a DFS starting from the vertex corresponding to the entrance. If at some point, we discover the exit vertex in the DFS, then there exists a path from the entrance to the exit. If we implement recursive DFS then the path would consist of all the vertices in the call stack corresponding to previous recursive calls to the DFS routine.

This problem can also be solved using BFS from the entrance vertex on the same graph model. The BFS tree has the property that the computed path will be a shortest path from the entrance. However BFS is more difficult to implement than DFS since in DFS, the compiler implicitly handles the DFS stack, whereas in BFS, the queue has to be explicitly coded. Since the problem did not call for a shortest path, it is better to use DFS.

```
class Coordinate {
  public:
    int x, y;

    const bool operator==(const Coordinate& that) const {
      return (x == that.x && y == that.y);
    }
};

// Check cur is within maze and is a white pixel
bool is_feasible(const Coordinate &cur, const vector<vector<int>> &maze) {
  return cur.x >= 0 && cur.x < maze.size() &&
         cur.y >= 0 && cur.y < maze[cur.x].size() && maze[cur.x][cur.y] == 0;
}

// Perform DFS to find a feasible path
bool search_maze_helper(vector<vector<int>> &maze, const Coordinate &cur,
```

```
                        const Coordinate &e, vector<Coordinate> &path) {
  if (cur == e) {
    return true;
  }

  const array<array<int, 2>, 4> shift = {0, 1, 0, -1, 1, 0, -1, 0};
  for (const array<int, 2> &s : shift) {
    Coordinate next{cur.x + s[0], cur.y + s[1]};
    if (is_feasible(next, maze)) {
      maze[next.x][next.y] = 1;
      path.emplace_back(next);
      if (search_maze_helper(maze, next, e, path)) {
        return true;
      }
      path.pop_back();
    }
  }
  return false;
}

vector<Coordinate> search_maze(vector<vector<int>> maze, const Coordinate &s,
                               const Coordinate &e) {
  vector<Coordinate> path;
  maze[s.x][s.y] = 1;
  path.emplace_back(s);
  if (search_maze_helper(maze, s, e, path) == false) {
    path.pop_back();
  }
  return path;  // empty path means no path between s and e
}
```

Problem 16.2, *pg. 133*: *Given a dictionary D and two strings s and t, write a function to determine if s produces t. Assume that all characters are lowercase alphabets. If s does produce t, output the length of a shortest production sequence; otherwise, output -1.*

Solution 16.2: Define the undirected graph $G = (D, E)$ by $(u, v) \in E$ iff $|u| = |v|$, and u and v differ in one character. (Note that the relation "differs in one character" is symmetric, which is why the graph is undirected.)

A production sequence is simply a path in G, so what we need is a shortest path from s to t in G. Shortest paths in an undirected graph are naturally computed using BFS. We use a queue and a hash table of vertices (which indicates if a vertex has already been visited). We enumerate neighbors of a vertex v by an outer loop that iterates over each position in v and an inner loop that iterates over each choice of character for that position.

```
// Use BFS to find the least steps of transformation
int transform_string(unordered_set<string> D, const string &s,
                     const string &t) {
  queue<pair<string, int>> q;
  D.erase(s);  // mark s as visited by erasing it in D
  q.emplace(s, 0);

```

```
  while (!q.empty()) {
    pair<string, int> f(q.front());
    // Return if we find a match
    if (f.first == t) {
      return f.second;  // number of steps reaches t
    }

    // Try all possible transformations of f.first
    string str = f.first;
    for (int i = 0; i < str.size(); ++i) {
      for (int j = 0; j < 26; ++j) {  // iterates through 'a' ~ 'z'
        str[i] = 'a' + j;  // change the (i + 1)-th char of str
        auto it(D.find(str));
        if (it != D.end()) {
          D.erase(it);  // mark str as visited by erasing it
          q.emplace(str, f.second + 1);
        }
      }
      str[i] = f.first[i];  // revert the change of str
    }
    q.pop();
  }

  return -1;  // cannot find a possible transformations
}
```

Problem 16.3, *pg. 133*: *Design an algorithm that takes a set of pins and a set of wires connecting pairs of pins, and determines if it is possible to place some pins on the left half of a PCB, and the remainder on the right half, such that each wire is between left and right halves. Return such a division, if one exists. For example, the light vertices and dark vertices in Figure 16.5 on Page 133 are such division.*

Solution 16.3: Assuming the pins are numbered from 0 to $p-1$, create an undirected graph G on p vertices $v_0, \ldots, v_{p-1}$. Add an edge between v_i to v_j if Pin i and Pin j are connected by a wire. For simplicity, assume G is connected; if not, the connected components can be analyzed independently.

Run BFS on G starting with v_0. Assign v_0 arbitrarily to lie on the left half. All vertices at an odd distance from v_0 are assigned to the right half.

When performing BFS on an undirected graph, all newly discovered edges will either be from vertices which are at a distance d from v_0 to undiscovered vertices (which will then be at a distance $d+1$ from v_0) or from vertices which are at a distance d to vertices which are also at a distance d. First, assume we never encounter an edge from a distance k vertex to a distance k vertex. In this case, each wire is from a distance k vertex to a distance $k+1$ vertex, so all wires are between the left and right halves.

If any edge is from a distance k vertex to a distance k vertex, we stop—the pins cannot be partitioned into left and right halves as desired. The reason is as follows. Let u and v be such vertices. Consider the first common ancestor a in the BFS tree of u and v (such an ancestor must exist since the search started at v_0). The paths $p_{a,u}$ and

$p_{a,v}$ in the BFS tree from a to u and v are of equal length; therefore the cycle formed by going from a to u via $p_{a,u}$, then through the edge (u, v), and then back to a from v via $p_{a,v}$ has an odd length. A cycle in which the vertices can be partitioned into two sets must have an even number of edges—it has to go back and forth between the sets and terminate at the starting vertex, and each back and forth adds two edges. Therefore the vertices in an odd length cycle cannot be partitioned into two sets such that all edges are between the sets.

```
class GraphVertex {
  public:
    int d;
    vector<GraphVertex*> edges;

    GraphVertex(void) : d(-1) {}
};

bool BFS(GraphVertex* s) {
  queue<GraphVertex*> q;
  q.emplace(s);

  while (q.empty() == false) {
    for (GraphVertex* &t : q.front()->edges) {
      if (t->d == -1) {  // unvisited vertex
        t->d = q.front()->d + 1;
        q.emplace(t);
      } else if (t->d == q.front()->d) {
        return false;
      }
    }
    q.pop();
  }
  return true;
}

bool is_any_placement_feasible(vector<GraphVertex> &G) {
  for (GraphVertex &v : G) {
    if (v.d == -1) {  // unvisited vertex
      v.d = 0;
      if (BFS(&v) == false) {
        return false;
      }
    }
  }
  return true;
}
```

ϵ-Variant 16.3.1: Design an algorithm that checks if a graph is bipartite.

ϵ-Variant 16.3.2: Design an algorithm that checks if a graph is 2-colorable.

Problem 16.4, *pg. 133*: *Let $G = (V, E)$ be a connected undirected graph. How would you*

efficiently check if G is $2\exists$-connected? Can you make your algorithm run in $O(|V|)$ time? How would you check if G is $2\forall$-connected?

Solution 16.4: First, we consider the problem of checking if G is $2\exists$-connected. If $G' = (V, E \setminus \{(u, v)\})$ is connected, it must be that a path exists between u and v. This is possible iff u and v lie on a cycle in G. Thus we have reduced the problem of checking if G is $2\exists$-connected to the checking if there exists a cycle in G.

We can check for the existence of a cycle in G by running DFS on G. Recall DFS maintains a color for each vertex. Initially, all vertices are white. When a vertex is first discovered, it is colored gray. When DFS finishes processing a vertex, that vertex is colored black.

As soon as we discover an edge from a gray vertex back to a gray vertex which is not its immediate predecessor in the search, a cycle exists in G and we can stop.

In general, the time complexity of DFS is $O(|V| + |E|)$. However the algorithm described above has time complexity $O(|V|)$. This is because an undirected graph with no cycles can have at most $|V| - 1$ edges.

```
class GraphVertex {
  public:
    enum Color {white, gray, black} color;
    vector<GraphVertex*> edges;
};

bool DFS(GraphVertex* cur, const GraphVertex* pre) {
  // Visiting a gray vertex means a cycle
  if (cur->color == GraphVertex::gray) {
    return true;
  }

  cur->color = GraphVertex::gray;  // marks current vertex as a gray one
  // Traverse the neighbor vertices
  for (GraphVertex* &next : cur->edges) {
    if (next != pre && next->color != GraphVertex::black) {
      if (DFS(next, cur)) {
        return true;
      }
    }
  }
  cur->color = GraphVertex::black;  // marks current vertex as black
  return false;
}

bool is_graph_2_exist(vector<GraphVertex> &G) {
  if (G.empty() == false) {
    return DFS(&G.front(), nullptr);
  }
  return false;
}
```

Now, we consider the problem of checking if G is $2\forall$-connected. Clearly, G is not $2\forall$-connected iff there exists an edge e such that $G' = (V, E \setminus \{e\})$ is disconnected. The latter condition holds iff no cycle includes edge e.

We can find an edge (u, v) that is not on a cycle with DFS. Without loss of generality, assume u is discovered first. Observe that the removal of (u, v) disconnects G iff no back edges exist between v or v's descendants and u or u's ancestors. Define $l(v)$ to be the minimum of the discovery time $d(v)$ of v and $d(w)$ of w such that (t, w) is a back edge from t, where t is a descendant of v, and w is an ancestor of v.

We claim $l(v) < d(v)$ iff a back edge exists between v or one of v's descendants to u or one of u's ancestors. If $l(v) < d(v)$, then there exists a path from v through one of its descendants to an ancestor of v, i.e., v lies on a cycle. For every vertex v, except the root, if $l(v) = d(v)$, it is not possible to get from v back to u; hence removal of (u, v) disconnects u and v. If the root is connected to an edge whose removal disconnects G, the other vertex on that edge will be identified by the technique above, meaning we do not need to consider the root at all.

Now, we show how to compute $l(v)$ efficiently. Once we have processed all of v's children, then $l(v) = \min(d(v), \min_{x \in \text{children}(v)} l(x))$. This computation does not add to the asymptotic complexity of DFS since it is just constant additional work per edge, so we can check 2V-connectedness in $O(|V| + |E|)$ time.

```
class GraphVertex {
  public:
    int d, l;  // discovery and leaving time
    vector<GraphVertex*> edges;
};

bool DFS(GraphVertex* cur, GraphVertex* pre, int time) {
  cur->d = ++time, cur->l = numeric_limits<int>::max();
  for (GraphVertex* &next : cur->edges) {
    if (next != pre) {
      if (next->d != 0) {  // back edge
        cur->l = min(cur->l, next->d);
      } else {  // forward edge
        if (DFS(next, cur, time) == false) {
          return false;
        }
        cur->l = min(cur->l, next->l);
      }
    }
  }
  return (pre == nullptr || cur->l < cur->d);
}

bool is_graph_2_for_all(vector<GraphVertex> &G) {
  if (G.empty() == false) {
    return DFS(&G.front(), nullptr, 0);
  }
  return true;
}
```

ϵ-Variant 16.4.1: Let G be a connected undirected graph. A vertex of G is an *articulation point* if its removal disconnects G. An edge of G is a *bridge* if its removal disconnects G. A *biconnected component* (BCC) of G is a maximal set of edges having

the property that any two edges in the set lie on common simple cycle. Design algorithms for computing articulation points, bridges, and BCCs.

Variant 16.4.2: Solve the 2∃-connectedness problem for an undirected graph using only two colors per vertex. (Do not use auxiliary data structures such as hash tables to mimic the third color.)

Problem 16.5, *pg. 134*: *Devise an efficient algorithm which takes a social network and computes for each individual his extended contacts.*

Solution 16.5: It is natural to model the network as a graph: vertices correspond to individuals, and an edge exists from A to B if B is a contact of A.

For an individual x, we can compute the set of x's contacts by running graph search (DFS or BFS) from x. Running graph search for each individual leads to a $O(|V|(|V| + |E|))$ algorithm for transitive closure.

```
class GraphVertex {
  public:
    int visitTime;
    vector<GraphVertex*> edges, extendedContacts;
};

void DFS(GraphVertex* cur, const int &time, vector<GraphVertex*> &contacts) {
  for (GraphVertex* &next : cur->edges) {
    if (next->visitTime != time) {
      next->visitTime = time;
      contacts.emplace_back(next);
      DFS(next, time, contacts);
    }
  }
}

void transitive_closure(vector<GraphVertex> &G) {
  // Build extended contacts for each vertex
  for (int i = 0; i < G.size(); ++i) {
    if (G[i].visitTime != i) {
      G[i].visitTime = i;
      DFS(&G[i], i, G[i].extendedContacts);
    }
  }
}
```

Another approach which has complexity $O(|V|^3)$ but which may be more efficient in practice for dense graphs, i.e., graphs in which $|E| = \Theta(|V|^2)$ is to run an all pairs shortest paths algorithm with edge weights of 1. If a path exists from u to v, the shortest path distance from u to v will be finite; otherwise, it will be ∞. We can improve this shortest path calculation by simply recording whether there is a path from u to v. In this way, we need a Boolean 2D array rather than an integer 2D array encoding the distances between the vertices. Although from an asymptotic perspective, the approach based on the all pairs shortest paths algorithm has the

same complexity as running multiple graph searches, it is likely to be more efficient in practice because of the use of Boolean arrays.

Problem 16.6, *pg. 134*: *Design an efficient algorithm that takes as input a collection of equality and inequality constraints and decides whether the constraints can be satisfied simultaneously.*

Solution 16.6: Let ϕ be a set of equality and inequality constraints on variables $x_0, \ldots, x_{n-1}$. Create an undirected graph G_ϕ on vertices $x_0, \ldots, x_{n-1}$; for each equality $x_i = x_j$, add the edge (x_i, x_j).

Now examine the connected components of G_ϕ. By the transitivity of equality, we can infer that $x_i = x_j$ for all vertices x_i and x_j in a common connected component.

Therefore if for some inequality $x_p \neq x_q$, vertices x_p and x_q lie in the same connected component, the set of constraints ϕ is not satisfied.

Conversely, let there be k connected components $C_0, \ldots, C_{k-1}$. Assign the variables in C_i to the value i. This satisfies all the equality constraints and since all the inequality constraints involve variables from different connected components, all inequality constraints are satisfied too.

```
1  class Constraint {
2    public:
3      int a, b;
4  };
5
6  class GraphVertex {
7    public:
8      int group;  // represents the connected component it belongs
9      vector<GraphVertex*> edges;
10
11     GraphVertex() : group(-1) {}
12 };
13
14 void DFS(GraphVertex &u) {
15   for (GraphVertex* &v : u.edges) {
16     if (v->group == -1) {
17       v->group = u.group;
18       DFS(*v);
19     }
20   }
21 }
22
23 bool are_constraints_satisfied(
24   const vector<Constraint> &E,     // Equality constraints
25   const vector<Constraint> &I) {  // Inequality constraints
26   unordered_map<int, GraphVertex> G;
27   // Build graph G according to E
28   for (const Constraint &e : E) {
29     G[e.a].edges.emplace_back(&G[e.b]), G[e.b].edges.emplace_back(&G[e.a]);
30   }
31
32   // Assign group index for each connected component
33   int group_count = 0;
```

```
  for (pair<int, GraphVertex> vertex : G) {
    if (vertex.second.group == -1) {  // is a unvisited vertex
      vertex.second.group = group_count++;  // assigns a group index
      DFS(vertex.second);
    }
  }

  // Examine each inequality constraint to see if there is a violation
  for (const Constraint &i : I) {
    if (G[i.a].group == G[i.b].group) {
      return false;
    }
  }
  return true;
}
```

Problem 16.7, *pg.135*: *How would you generalize your solution to Problem 13.6 on Page 100. to determine the largest number of teams that can be photographed simultaneously subject to the same constraints?*

Solution 16.7: Let G be the DAG with vertices corresponding to the teams as follows and edges from vertex X to Y iff $\text{sort}(X) < \text{sort}(Y)$.

Every sequence of teams where a team can be placed behind its predecessor corresponds to a path in G. To find the longest such sequence, we simply need to find the longest path in the DAG G. We can do this, for example, by topologically ordering the vertices in G; the longest path terminating at vertex v is the maximum of the longest paths terminating at v's fan-ins concatenated with v itself.

The topological ordering computation is $O(|V| + |E|)$ and dominates the computation time.

```
class GraphVertex {
  public:
    vector<GraphVertex*> edges;
    int maxDistance;
    bool visited;

    GraphVertex(void) : maxDistance(1), visited(false) {};
};

void DFS(GraphVertex* cur, stack<GraphVertex*> &vertex_order) {
  cur->visited = true;
  for (GraphVertex* &next : cur->edges) {
    if (next->visited == false) {
      DFS(next, vertex_order);
    }
  }
  vertex_order.emplace(cur);
}

stack<GraphVertex*> build_topological_ordering(vector<GraphVertex> &G) {
  stack<GraphVertex*> vertex_order;
  for (GraphVertex &g: G) {
```

```
23      if (g.visited == false) {
24        DFS(&g, vertex_order);
25      }
26    }
27    return vertex_order;
28  }
29
30  int find_longest_path(stack<GraphVertex*> &vertex_order) {
31    int max_distance = 0;
32    while (vertex_order.empty() == false) {
33      GraphVertex* u = vertex_order.top();
34      max_distance = max(max_distance, u->maxDistance);
35      for (GraphVertex* &v : u->edges) {
36        v->maxDistance = max(v->maxDistance, u->maxDistance + 1);
37      }
38      vertex_order.pop();
39    }
40    return max_distance;
41  }
42
43  int find_largest_number_teams(vector<GraphVertex> &G) {
44    stack<GraphVertex*> vertex_order(build_topological_ordering(G));
45    return find_longest_path(vertex_order);
46  }
```

Problem 16.8, ***pg. 136***: *Given an instance of the task scheduling problem, compute the least amount of time in which all the tasks can be performed, assuming an unlimited number of servers. Explicitly check that the system is feasible.*

Solution 16.8: This problem is naturally modeled using a directed graph. Vertices correspond to tasks, and an edge from u to v indicates that u must be completed before v can begin. The system is infeasible iff a cycle is present in the derived graph.

We can check the presence of a cycle by performing a DFS. If no cycle is present, the DFS numbering yields a topological ordering of the graph, i.e., an ordering of the vertices such that v follows u whenever an edge is present from u to v. Specifically, the DFS finishing time gives a topological ordering in reverse order. Therefore both testing for a cycle and computing a topological ordering can be performed in $O(n+m)$ time, where n and m are the number of vertices and edges in the graph, respectively.

Since the number of servers is unlimited, T_i can be completed τ_i time after all the tasks it depends on have completed. Therefore we can compute the soonest each task can complete by processing tasks in topological order, starting from the tasks that depend on no other tasks. If no such tasks exist, there must be a sequence of tasks starting and ending at the same task, such that each task requires the previous task to be completed before it can be started, i.e., the system is infeasible.

When the number of servers is limited, the problem becomes NP-complete. An equivalent problem with limited resources is the subject of Problem 17.13 on Page 143.

Problem 16.9, ***pg. 136***: *Design an algorithm which takes as input a graph $G = (V, E)$, directed or undirected, a nonnegative cost function on E, and vertices s and t; your algorithm*

should output a path with the fewest edges amongst all shortest paths from s to t.

Solution 16.9: Dijkstra's shortest path algorithm uses scalar values for edge length. However it can easily be modified to the case where the edge weight is a pair if *addition* and *comparison* can be defined over these pairs. In this case, if the edge cost is c, we say the length of the edge is given by the pair $(c, 1)$. We define addition to be just component-wise addition. Hence if we sum up the edge lengths over a path, we essentially get the total cost and the number of edges in the path. The compare function is lexicographic, first the total cost, then the number of edges. We can run Dijkstra's shortest path algorithm with this compare function and find the shortest path that requires the least number of edges.

Since a heap does not support efficient updates, it is more convenient to use a BST than a heap to implement the algorithm.

```
template <typename DistanceType>
class GraphVertex {
  public:
    pair<DistanceType, int> distance;  // stores (dis, #edges) pair
    // stores (vertex, dis) pair
    vector<pair<GraphVertex<DistanceType>*, DistanceType>> edges;
    int id;  // stores the id of this vertex
    GraphVertex* pred;  // stores the predecessor in the shortest path
    bool visited;

    GraphVertex(void) :
      distance(numeric_limits<DistanceType>::max(), 0),
      pred(nullptr),
      visited(false) {}
};

template <typename DistanceType>
class Comp {
  public:
    const bool operator()(const GraphVertex<DistanceType>* lhs,
                          const GraphVertex<DistanceType>* rhs) const {
      return lhs->distance.first < rhs->distance.first ||
             (lhs->distance.first == rhs->distance.first &&
                 lhs->distance.second < rhs->distance.second);
    }
};

template <typename DistanceType>
void output_shortest_path(GraphVertex<DistanceType>* &v) {
  if (v) {
    output_shortest_path(v->pred);
    cout << v->id << " ";
  }
}

template <typename DistanceType>
void Dijkstra_shortest_path(vector<GraphVertex<DistanceType>> &G,
                            GraphVertex<DistanceType>* s,
                            GraphVertex<DistanceType>* t) {
```

```
  // Initialization the distance of starting point
  s->distance = {0, 0};
  set<GraphVertex<DistanceType>*, Comp<DistanceType>> node_set;
  node_set.emplace(s);

  do {
    GraphVertex<DistanceType>* u = nullptr;
    // Extract the minimum distance vertex from heap
    while (node_set.empty() == false) {
      u = *node_set.cbegin();
      node_set.erase(node_set.cbegin());
      if (u->visited == false) {  // found an unvisited node
        break;
      }
    }

    if (u) {  // u is a valid vertex
      u->visited = true;  // mark u as visited
      // Relax neighboring vertices of u
      for (const auto &v : u->edges) {
        DistanceType v_distance = u->distance.first + v.second;
        int v_num_edges = u->distance.second + 1;
        if (v.first->distance.first > v_distance ||
            (v.first->distance.first == v_distance &&
             v.first->distance.second > v_num_edges)) {
          node_set.erase(v.first);
          v.first->pred = u;
          v.first->distance = {v_distance, v_num_edges};
          node_set.emplace(v.first);
        }
      }
    } else {  // u is not a valid vertex
      break;
    }
  } while (t->visited == false);  // until t is visited

  // Output the shortest path with fewest edges
  output_shortest_path(t);
}
```

ϵ-Variant 16.9.1: Solve the same problem when edge weights are integers in $(-\infty, \infty)$. You may modify the graph, but must use an unmodified shortest path algorithm.

Problem 16.10, *pg. 136*: *Given a time-table, a starting city, a starting time, and a destination city, how would you compute the soonest you could get to the destination city? Assume all flights start and end on time, and that you need 60 minutes between flights.*

Solution 16.10: We use Dijkstra's single-source shortest path algorithm with a minor variation. At each iteration we add a new city c to the set of cities to which we know the fastest route. After c is identified, we relax edges out of c by searching for all flights out of c that depart 60 minutes or later from the earliest we can get to c. We do

not need to relax an edge from c to c' more than once, e.g., if two flights from c to c' that satisfy the inter-flight time constraint, we only need to relax the one that arrives in c' sooner.

The time complexity is identical to that of Dijkstra's algorithm with the flights playing the role of edges, and the cities playing the role of vertices.

Problem 16.11, ***pg. 136***: *Devise an efficient algorithm which takes the existing highway network (specified as a set of highway sections between pairs of cities) and proposals for new highway sections, and returns a proposed highway section which minimizes the shortest driving distance between El Paso and Corpus Christi.*

Solution 16.11: Note that we cannot add more than one proposal to the existing network and run a shortest path algorithm—we may end up with a shortest path which uses multiple proposals.

Let there be n cities, m existing highway sections, and k proposed sections. One approach is to simply run a single-source shortest path algorithm from El Paso k times, one for each of the proposed sections. Dijkstra's single-source shortest path algorithm can be made to run in $O(n \log n + m)$, leading to an overall time complexity of $O(k(n \log n + m))$.

Since $m+k$ can be as large as $\frac{n(n-1)}{2}$, the above approach can have a time complexity as high as $O(n^4)$. We can improve upon this by first running an all pairs shortest paths algorithm, such as the Floyd-Warshall algorithm which will compute all pairs of shortest paths distances in $O(n^3)$ time.

Let $S(u, v)$ be the 2D array of shortest path distances for each pair of cities. Each proposal p is a pair of cities (x, y). The best we can do by using proposal p is $\min(S(a, b), S(a, x) + d(x, y) + S(y, b))$ where $d(x, y)$ is the distance of the proposed highway p between x and y, and a and b are El Paso and Corpus Christi, respectively. This computation is $O(1)$ time, so we can evaluate all the proposals in time proportional to the number of proposals after we have computed the shortest path between each pair of cities. This results in an $O(n^3 + k)$ time complexity; since $k \leq \frac{n(n-1)}{2}$, the time complexity is $O(n^3)$.

```
template <typename DistanceType>
class HighwaySection {
  public:
    int x, y;
    DistanceType distance;
};

template <typename DistanceType>
void Floyd_Warshall(vector<vector<DistanceType>> &G) {
  for (int k = 0; k < G.size(); ++k) {
    for (int i = 0; i < G.size(); ++i) {
      for (int j = 0; j < G.size(); ++j) {
        if (G[i][k] != numeric_limits<DistanceType>::max() &&
            G[k][j] != numeric_limits<DistanceType>::max() &&
            G[i][j] < G[i][k] + G[k][j]) {
          G[i][j] = G[i][k] + G[k][j];
        }
```

```
      }
    }
  }
}

template <typename DistanceType>
HighwaySection<DistanceType> find_best_proposals(
    const vector<HighwaySection<DistanceType>> &H,
    const vector<HighwaySection<DistanceType>> &P,
    const int &a, const int &b, const int &n) {
  // G stores the shortest path distance between all pairs
  vector<vector<DistanceType>>
    G(n, vector<DistanceType>(n, numeric_limits<DistanceType>::max()));

  // Build graph G based on existing highway sections H
  for (const HighwaySection<DistanceType> &h : H) {
    G[h.x][h.y] = G[h.y][h.x] = h.distance;
  }
  // Perform Floyd Warshall to build the shortest path between vertices
  Floyd_Warshall(G);

  // Examine each proposal for shorter distance between a and b
  DistanceType min_dis_a_b = G[a][b];
  HighwaySection<DistanceType> best_proposal;
  for (const HighwaySection<DistanceType> &p : P) {
    if (G[a][p.x] != numeric_limits<DistanceType>::max() &&
        G[p.y][b] != numeric_limits<DistanceType>::max()) {
      if (min_dis_a_b < G[a][p.x] + p.distance + G[p.y][b]) {
        min_dis_a_b = G[a][p.x] + p.distance + G[p.y][b];
        best_proposal = p;
      }
      if (min_dis_a_b < G[a][p.y] + p.distance + G[p.x][b]) {
        min_dis_a_b = G[a][p.y] + p.distance + G[p.x][b];
        best_proposal = p;
      }
    }
  }
  return best_proposal;
}
```

Problem 16.12, *pg. 137*: *Design an efficient algorithm to determine whether there exists an arbitrage—a way to start with a single unit of some commodity C and convert it back to more than one unit of C through a sequence of exchanges.*

Solution 16.12: We define a weighted directed graph $G = (V, E = V \times V)$, where V corresponds to the set of commodities. The weight $w(e)$ of edge $e = (u, v)$ is the amount of commodity v we can buy with one unit of commodity u. Observe that an arbitrage exists iff there exists a cycle in G whose edge weights multiply out to more than 1.

Create a new graph $G' = (V, E)$ with weight function $w'(e) = -\lg w(e)$. Since $\lg(a \times b) = \lg a + \lg b$, there exists a cycle in G whose edge weights multiply out to more than 1 iff there exists a cycle in G' whose edge weights sum up to less than

$\lg 1 = 0$. (This property is true for logarithms to any base, so if it is more efficient for example to use base-e, we can do so.)

The Bellman-Ford algorithm, which takes $O(|V||E|)$ time, detects negative-weight cycles. Usually finding a negative-weight cycle is done by adding a dummy vertex s with 0-weight edges to each vertex in the given graph and running the Bellman-Ford single-source shortest path algorithm from s. However, for the arbitrage problem, the graph is complete. Hence we can run Bellman-Ford algorithm from any single vertex, and get the right result.

```
bool Bellman_Ford(const vector<vector<double>> &G, const int &source) {
  vector<double> dis_to_source(G.size(), numeric_limits<double>::max());
  dis_to_source[source] = 0;

  for (int times = 1; times < G.size(); ++times) {
    bool have_update = false;
    for (int i = 0; i < G.size(); ++i) {
      for (int j = 0; j < G[i].size(); ++j) {
        if (dis_to_source[i] != numeric_limits<double>::max() &&
            dis_to_source[j] > dis_to_source[i] + G[i][j]) {
          have_update = true;
          dis_to_source[j] = dis_to_source[i] + G[i][j];
        }
      }
    }

    // No update in this iteration means no negative cycle
    if (have_update == false) {
      return false;
    }
  }

  // Detect cycle if there is any further update
  for (int i = 0; i < G.size(); ++i) {
    for (int j = 0; j < G[i].size(); ++j) {
      if (dis_to_source[i] != numeric_limits<double>::max() &&
          dis_to_source[j] > dis_to_source[i] + G[i][j]) {
        return true;
      }
    }
  }
  return false;
}

bool is_Arbitrage_exist(vector<vector<double>> G) {
  // Transform each edge in G
  for (vector<double> &edge_list : G) {
    for (double &edge : edge_list) {
      edge = -log10(edge);
    }
  }

  // Use Bellman-Ford to find negative weight cycle
  return Bellman_Ford(G, 0);
```

```
45 }
```

Problem 16.13, *pg. 137*: *Let $G = (V, E)$ be an undirected graph with edge weight function $w : E \mapsto \mathcal{Z}^+$. You are given $T \subset E$, an MST of G. Let e be an edge. Design efficient algorithms for computing the MST when (1.) $w(e)$ decreases , and (2.) $w(e)$ increases.*

Solution 16.13: We make use of two key facts about an MST. The first is that if an edge e is the unique heaviest weight edge on some cycle, it cannot lie in the MST. The second is that if an edge is the unique lightest weight edge in some set of edges that disconnect the graph, it must lie in the MST.

For the first case, if $e \in T$, then T remains unchanged. Otherwise, let $e = (u, v)$. We search for the unique path π in T between u and v. If $w(e)$ remains greater than or equal to the weights of the edges on this path, T is unchanged. Otherwise, we obtain the new MST by deleting the edge in π which has maximum weight, and adding e.

For the second case, if $e \notin T$, then T will not change. Otherwise, let $e = (u, v)$. Removing e from T leaves us with two components. We want to replace e by the lowest-weight edge between these components. We can find this edge by first finding the two components, e.g., by DFS through $T \setminus \{e\}$ from u and from v, and then enumerating all the edge in G.

In both cases, the time complexity is dominated by the need to do graph search, i.e., $O(|V| + |E|)$.

Problem 17.1, *pg. 139*: *How would you programmatically determine if a tie is possible in a presidential election with two candidates, R and D?*

Solution 17.1: We need to determine if there exists a subset of states whose Electoral College votes add up to $\frac{538}{2} = 269$. This is an instance of the subset sum problem, and is known to be NP-complete. It is a specialization of the 0-1 knapsack problem described in Problem 17.2 on Page 139 and the DP solution to that problem can be used. Following is the code in C++:

```
1  // V contains the number of votes for each state
2  long ties_election(const vector<int> &V) {
3    int total_votes = accumulate(V.cbegin(), V.cend(), 0);
4
5    // No way to tie if the total number of votes is odd
6    if (total_votes & 1) {
7      return 0;
8    }
9
10   vector<vector<long>> table(V.size() + 1, vector<long>(total_votes + 1, 0));
11   table[0][0] = 1;  // base condition: 1 way to reach 0
12   for (int i = 0; i < V.size(); ++i) {
13     for (int j = 0; j <= total_votes; ++j) {
14       table[i + 1][j] = table[i][j] + (j >= V[i] ? table[i][j - V[i]] : 0);
15     }
16   }
17   return table[V.size()][total_votes >> 1];
18 }
```

Problem 17.2, *pg. 139*: *Design an algorithm for the knapsack problem that selects a subset of items that has maximum value and weighs at most w ounces. All items have integer weights and values.*

Solution 17.2: Let $V[i, w]$ be the maximum value that can be packed with weight less than or equal to w using the first i clocks. Then $V[i, w]$ satisfies the following recurrence:

$$V[i,w] = \begin{cases} \max(V[i-1,w], V[i-1,w-w_i]+v_i), & \text{if } w_i \le w; \\ V[i-1,w], & \text{otherwise.} \end{cases}$$

For $i = 0$ or $w = 0$, we set $V[i, w] = 0$. This DP procedure computes $V[n, w]$ in $O(nw)$ time, and uses $O(nw)$ space. Note that the space complexity can be improved to $O(w)$ by using a one-dimensional array to store the current optimal result and rewriting the next step result back to this array. Following is the code in C++:

```
template <typename ValueType>
ValueType knapsack(const int &w, const vector<pair<int, ValueType>> &items) {
  vector<ValueType> V(w + 1, 0);
  for (int i = 0; i < items.size(); ++i) {
    for (int j = w; j >= items[i].first; --j) {
      V[j] = max(V[j], V[j - items[i].first] + items[i].second);
    }
  }
  return V[w];
}
```

Variant 17.2.1: Solve the knapsack problem when the thief can take a fractional amount of an item.

Problem 17.3, *pg. 140*: *Let array A be an array of n positive integers. Entry A[i] is the value of the i-th stolen item. Design an algorithm that computes a subset $S \subset \mathcal{Z}_n = \{0, 1, 2, \ldots, n-1\}$ such that $|\sum_{i \in S} A[i] - \sum_{j \in \mathcal{Z}_n \setminus S} A[j]|$ is minimized.*

Solution 17.3: We first compute a Boolean-valued array *is_Ok*, indexed from 0 to sum $= \sum_{i=0}^{n-1} A[i]$, inclusive. The array *is_Ok* encodes whether a given i is the sum of the elements in some subset of stolen items. The *is_Ok* array entries are initialized to `false` for each index greater than 0; *is_Ok*[0] is initialized to `true`. We assign values to *is_Ok* as follows. Set *is_Ok*[A[0]] to `true`. Then set *is_Ok*[A[1]] and *is_Ok*[A[0] + A[1]] to `true`. Then set *is_Ok*[A[2]], *is_Ok*[A[0] + A[2]], *is_Ok*[A[1] + A[2]], and *is_Ok*[A[0] + A[1] + A[2]] to `true`. Generalizing, the set of values corresponding to subsets of $\{0, 1, \ldots, i-1, i\}$ is the union of the set of values $\{v_0, v_1, \ldots, v_{j-1}\}$ corresponding to subsets of $\{0, 1, \ldots, i-1\}$ and the set $\{v_0 + A[i], v_1 + A[i], \ldots, v_{j-1} + A[i]\}$. This new set of values can be computed by iterating over values v in [sum, $A[i]$], setting *is_Ok*[v] to `true` wherever *is_Ok*[$v - A[i]$] is `true`. The time complexity is $O(n \cdot \text{sum})$.

After obtaining *is_Ok* we find the best partition by iterating over i from $\frac{\text{sum}}{2}$ downwards until we first hit an entry i such that *is_Ok*[i] is `true`; this is the closest we can get to an equal split. Note that *is_Ok* is symmetric about its midpoint since if $S \subset \{0, 1, 2, \ldots, n-1\}$ has a value w, then $S' = \mathcal{Z}_n \setminus S$ has a value $\texttt{sum} - w$, so we do not need to search *is_Ok* for indices greater than $\frac{\text{sum}}{2}$.

```
int minimize_difference(const vector<int> &A) {
  int sum = accumulate(A.cbegin(), A.cend(), 0);

  unordered_set<int> is_Ok;
  is_Ok.emplace(0);
  for (const int &item : A) {
    for (int v = sum >> 1; v >= item; --v) {
      if (is_Ok.find(v - item) != is_Ok.cend()) {
        is_Ok.emplace(v);
      }
    }
  }

  // Find the first i from middle where is_Ok[i] == true
  for (int i = sum >> 1; i > 0; --i) {
    if (is_Ok.find(i) != is_Ok.cend()) {
      return (sum - i) - i;
    }
  }
  return sum;   // one thief takes all
}
```

Variant 17.3.1: Solve the same problem with the additional constraint that the thieves have the same number of items.

Problem 17.4, *pg. 140*: *Write a program that determines a sequence of steps by which the required amount of milk can be obtained using the worn-out jugs. The milk is being added to a large mixing bowl, and hence cannot be removed from the bowl. Furthermore, it is not possible to pour one jug's contents into another. Your scheme should always work, i.e., return between* 2100 *and* 2300 *mL of milk, independent of how much is chosen in each individual step, as long as that quantity satisfies the given constraints.*

Solution 17.4: It is natural to solve this problem using recursion—if we use jug A for the last step, we need to correctly measure a volume of milk that is at least 2100 − 230 = 1870 mL—the last measurement may be as little as 230 mL, and anything less than 1870 mL runs the risk of being too little. Similarly, the volume must be at most 2300 − 240 = 2060 mL. The volume is not achievable if it is not achievable with any of the three jugs as ending points. We cache intermediate computations to reduce the number of recursive calls.

In the following code, we implement a general purpose function which finds the feasibility among n jugs; those arrays are passed in as `jugs`.

```
class Jug {
```

```
  public:
    int low, high;
};

class PairEqual {
  public:
    const bool operator()(const pair<int, int> &a,
                          const pair<int, int> &b) const {
      return a.first == b.first && a.second == b.second;
    }
};

class HashPair {
  public:
    const size_t operator()(const pair<int, int> &p) const {
      return hash<int>()(p.first) ^ hash<int>()(p.second);
    }
};

bool check_feasible_helper(const vector<Jug> &jugs, const int &L,
                           const int &H, unordered_set<pair<int, int>,
                                                       HashPair,
                                                       PairEqual> &c) {
  if (L > H || c.find({L, H}) != c.cend() || (L < 0 && H < 0)) {
    return false;
  }

  // Check the volume for each jug to see if it is possible
  for (const Jug &j : jugs) {
    if ((L <= j.low && j.high <= H) ||  // base case: j is contained in [L, H]
      check_feasible_helper(jugs, L - j.low, H - j.high, c)) {
      return true;
    }
  }
  c.emplace(L, H);  // marks this as impossible
  return false;
}

bool check_feasible(const vector<Jug> &jugs, const int &L, const int &H) {
  unordered_set<pair<int, int>, HashPair, PairEqual> cache;
  return check_feasible_helper(jugs, L, H, cache);
}
```

Variant 17.4.1: Suppose Jug i can be used to measure any quantity in $[l_i, u_i]$ exactly. Determine if it is possible to measure a quantity of milk between L and U.

Problem 17.5, *pg. 141*: *Given a graph $G = (V, E)$, with cost function $c : E \mapsto \mathcal{Z}^+$, delay function $d : E \mapsto \mathcal{Z}^+$, designated vertices s and t, and a delay constraint $\Delta \in \mathcal{Z}^+$, find a path from s to t with minimum cost, subject to the constraint that the delay of the path is no more than Δ. Costs are additive—the cost of a path is the sum of the costs of the individual edges; the same holds for delays.*

Solution 17.5: The delay-constrained shortest-path problem is known to be NP-complete. It can be solved in pseudo-polynomial time using DP. For each vertex v and delay $\delta \in [0, \Delta]$ we compute $M(v, \delta)$, the minimum cost path from s to v that has a delay less than or equal to δ. The function $M(v, \delta)$ satisfies the following recurrence:

$$M(v, \delta) = \min_{u \in \text{fan-in}(v)} (c(u, v) + M(u, \delta - d(u, v)))$$

Base cases include $M(s, \delta) = 0$ and $M(v, \delta) = \infty$, for $\delta < 0$.

By caching values, $M(t, \Delta)$ can be computed in $O(|E|\Delta)$ time with $O(|V|\Delta)$ space for the cache.

Alternately, we can use branch and bound. We can obtain good lower bounds using Lagrangian relaxation. The general idea behind Lagrangian relaxation is to solve constrained optimization problems by adding the constraint as a penalty term in the objective function. In our specific setting we can use a multiplier $\lambda \in \mathcal{R}^+$ to compute a new cost $c(u, v) + \lambda d(u, v)$ for each edge. Now we run a conventional shortest path algorithm. The effect of the λ term is to avoid paths that incur a large delay—the larger λ is, the more we avoid delay.

Let c_{opt} be the optimum cost for delay feasible paths, and d_{opt} the delay on an optimum path. Suppose the shortest path for a given λ has cost $c^* + \lambda d^*$. Observe that if this path is infeasible, i.e., $d^* > \Delta$, then $c_{\text{opt}} > c^*$, since otherwise it would be impossible for $c_{\text{opt}} + \lambda d_{\text{opt}} \geq c^* + \lambda d^*$.

Note that each $c^* + \lambda d^*$ where $d^* > \Delta$ yields a lower bound. The greatest lower bound across all λ may not be c_{opt}. This phenomenon is known as a duality gap. Furthermore, if $d^* > \Delta$, then we can say nothing about the relationship between c^* and c_{opt}.

Problem 17.6, *pg. 141*: *Suppose you are given a set of cities in the Cartesian plane, as shown in Figure 4.3 on Page 36. The cost of traveling from one city to another is a constant multiple of the distance between the cities. Give an efficient procedure for computing a tour whose cost is no more than two times the cost of an optimum tour.*

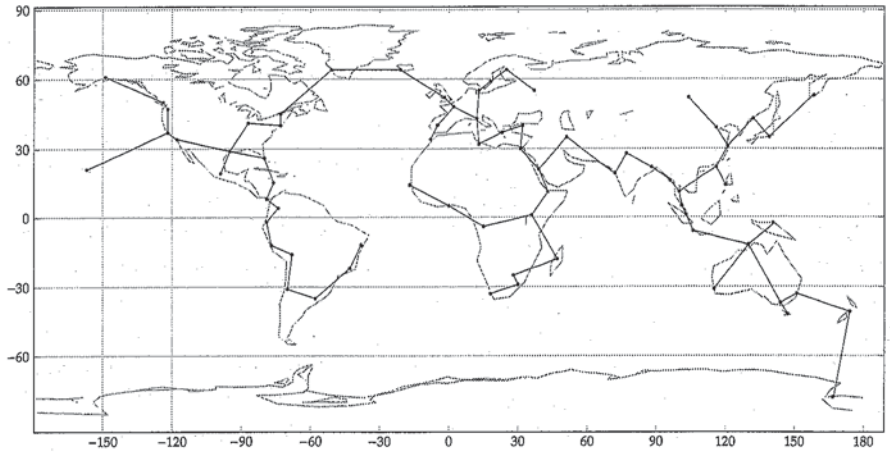
Solution 17.6: A good way to approach this problem is to think of a similar problem that can be solved exactly efficiently. The MST problem has an efficient algorithm, and it yields a way of visiting each city exactly twice—start at any city c and do an in-order walk in the MST with c as the root. This traversal leads to a path in which each edge is visited exactly twice.

Consider any tour. If we drop the edge back to the starting city, the remaining set of edges constitute a tree. Therefore the cost of the optimum tour is at least as great as the cost of the MST.

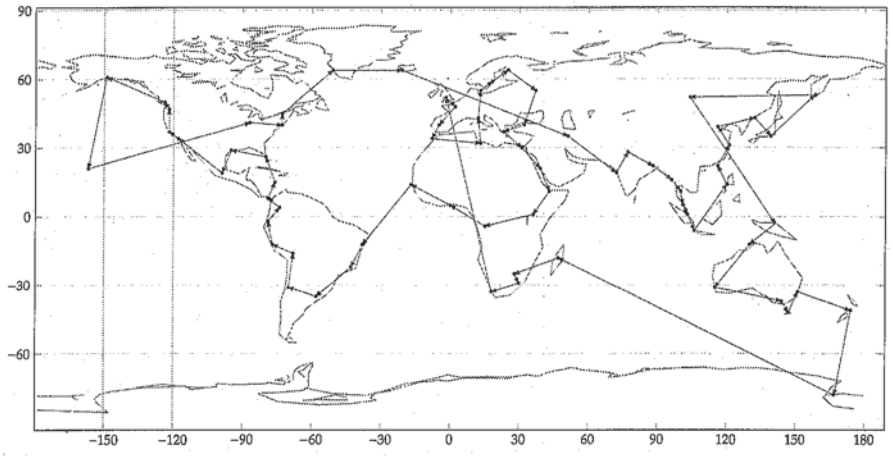
Now we make use of the fact that the distances between cities satisfies the triangle inequality to build a tour from the MST whose cost is no greater than the MST. When we perform our in-order walk, we simply skip over cities we have already visited—the direct distance from u to v cannot be more than the sum of distances on a path from u to v.

Hence we have a true tour costing at most twice the cost of the MST which itself was a lower bound on the cost of the traveling salesman problem, i.e., the tour has a cost that is at most twice the cost of an optimum tour.

The application of the algorithm described above to the cities in Figure 4.3 on Page 36 is shown in Figure 21.16.



(a) A minimum spanning tree for the cities in Figure 4.3 on Page 36.



(b) A tour derived from the minimum spanning tree in (a).

Figure 21.16: The two key steps of the approximation algorithm for the TSP.

Problem 17.7, *pg. 142*: *Design a fast algorithm for selecting k warehouse locations that is provably within a constant factor of the optimum solution.*

Solution 17.7: A natural approach to this problem is to build the assignment one warehouse at a time. We can pick the first warehouse to be the city for which the cost

is minimized—this takes $\Theta(n^2)$ time since we try each city one at a time and check its distance to every other city.

Let's say we have selected the first $i-1$ warehouses $\{w_1, w_2, \ldots, w_{i-1}\}$ and are trying to choose the i-th warehouse. A reasonable choice for the i-th warehouse is the one that is the farthest from the $i-1$ warehouses already chosen. This city can be computed in $O(ni)$ time.

We use the computation above to select k warehouses. Let the maximum distance from the remaining cities to the k warehouses be d_m. Then the cost of this assignment is d_m. Let e be a city which is d_m distance from the warehouse it is closest to. Note that the k warehouse cities are all at least d_m apart; otherwise, we would have chosen e to locate a warehouse at in one of the first k iterations.

By the pigeonhole principle, at least two of the $k+1$ cities $\{w_1, w_2, \ldots, w_k, e\}$ must have the same closest warehouse in an optimum assignment. Let p and q be two such cities and w be the warehouse city closest to p and q in an optimum assignment of warehouses. By the triangle inequality, $d(p,q) \leq d(w,p) + d(w,q)$. Since $d_m \leq d(p,q)$, it follows that at least one of $d(w,p)$ or $d(w,q)$ is greater than or equal to $\frac{d_m}{2}$. Hence the cost of this optimum assignment is at least $\frac{d_m}{2}$, implying our greedy heuristic produced an assignment that is within a factor of two of the cost of the optimum assignment.

Note that the initial selection of a warehouse is immaterial for the argument to work but heuristically, it is better to choose a central city as a starting point.

Problem 17.8, *pg. 142*: *Implement a Sudoku solver. Your program should read an instance of Sudoku from the command line. The command line argument is a sequence of 3-digit strings, each encoding a row, a column, and a digit at that location.*

Solution 17.8: We use a straight-forward application of the backtracking principle. We traverse the 2D array entries one at a time. If the entry is empty, we try each value for the entry, and see if the updated 2D array is still valid; if it is we recurse. If all the entries have been filled, the search is successful.

In practice it is more efficient to see if a conflict results on adding a new entry before adding it rather than adding it and seeing if a conflict is present. See the code for details.

```
bool valid_to_add(const vector<vector<int>> &A, const int &i, const int &j,
                  const int &val) {
  // Check row constraints
  for (int k = 0; k < A.size(); ++k) {
    if (val == A[k][j]) {
      return false;
    }
  }

  // Check column constraints
  for (int k = 0; k < A.size(); ++k) {
    if (val == A[i][k]) {
      return false;
    }
```

```
  }

  // Check region constraints
  int region_size = sqrt(A.size());
  int I = i / region_size, J = j / region_size;
  for (int a = 0; a < region_size; ++a) {
    for (int b = 0; b < region_size; ++b) {
      if (val == A[region_size * I + a][region_size * J + b]) {
        return false;
      }
    }
  }
  return true;
}

bool solve_Sudoku_helper(vector<vector<int>> &A, int i, int j) {
  if (i == A.size()) {
    i = 0;  // start a new row
    if (++j == A[i].size()) {
      return true;  // entire matrix has been filled without conflict
    }
  }

  // Skip nonempty entries
  if (A[i][j] != 0) {
    return solve_Sudoku_helper(A, i + 1, j);
  }

  for (int val = 1; val <= A.size(); ++val) {
    // Note: practically, it's substantially quicker to check if entryval
    // conflicts with any of the constraints if we add it at (i,j) before
    // adding it, rather than adding it and then calling is_valid_Sudoku.
    // The reason is that we know we are starting with a valid configuration,
    // and the only entry which can cause a problem is entryval at (i,j).
    if (valid_to_add(A, i, j, val)) {
      A[i][j] = val;
      if (solve_Sudoku_helper(A, i + 1, j)) {
        return true;
      }
    }
  }

  A[i][j] = 0;  // undo assignment
  return false;
}

// 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;
}

bool solve_Sudoku(vector<vector<int>> &A) {
  if (is_valid_Sudoku(A) == false) {
    cout << "Initial configuration violates constraints." << endl;
    return false;
  }

  if (solve_Sudoku_helper(A, 0, 0)) {
    for (int i = 0; i < A.size(); ++i) {
      copy(A[i].begin(), A[i].end(), ostream_iterator<int>(cout, " "));
      cout << endl;
    }
    return true;
  } else {
    cout << "No solution exists." << endl;
    return false;
  }
}
```

Variant 17.8.1: Compute a placement of eight queens on an 8×8 chessboard in which no two queens attack each other.

Variant 17.8.2: Compute a placement of 32 knights, or 14 bishops, 16 kings or eight rooks on an 8×8 chessboard in which no two pieces attack each other.

Variant 17.8.3: Compute the smallest number of queens that can be placed to attack each uncovered square.

Problem 17.9, *pg. 142*: *Given an array of digits A and a nonnegative integer k, intersperse multiplies ($\times$) and adds ($+$) with the digits of A such that the resulting arithmetical expression evaluates to k. For example, if A is $\langle 1,2,3,2,5,3,7,8,5,9\rangle$ and k is 995, then k can be realized by the expression* "$123 + 2 + 5 \times 3 \times 7 + 85 \times 9$".

Solution 17.9: Let A be the array of n digits and k the target sum. We want to intersperse $\times$ and $+$ operations among these characters in such a way that the resulting expression equals k.

For each pair of characters, $(A[i], A[i+1])$, we can choose to insert a $\times$, a $+$, or no operator. The number of such locations is $n-1$, implying we can encode the choice with an array of length $n-1$. Each entry is one of three values—$\times$, $+$, and $\sqcup$ (which indicates no operator is added at that location). Exactly 3^{n-1} such arrays exist, so a brute-force solution is to systematically enumerate all arrays. For each enumerated array, we compute the resulting expression, and return as soon as we evaluate to k. The time complexity is $O(n3^n)$, since each expression takes time $O(n)$ to evaluate.

The performance can be improved heuristically using pruning. For example, if we have a partial assignment that inserts $+$ between i and $i+1$, and an assignment of operators to $A[0:i]$ yields an expression that evaluates to k', then we need to search for an assignment of operators to $A[i+1:n-1]$ that yields $k-k'$. (Note we cannot do similar pruning with $\times$ because $\times$ has a higher precedence.) Additional pruning can be based on the observation that the maximum value of an expression corresponds to the case where no operators are inserted. For the previous example, if $k-k'$ is greater than the integer encoded by digits in $A[i+1:n-1]$, we can stop searching. Here is an implementation in C++; it makes heavy use of the STL.

```
1  int evaluate(list<int> operand_list, const list<char> &oper_list) {
2    // Evaluate '*' first
3    auto operand_it = operand_list.begin();
4    for (const char &oper : oper_list) {
5      if (oper == '*') {
6        int product = *operand_it;
7        operand_it = operand_list.erase(operand_it);
8        product *= *operand_it;
9        *operand_it = product;
10     } else {
11       ++operand_it;
12     }
13   }
14
```

```
  // Evaluate '+' second
  return accumulate(operand_list.cbegin(), operand_list.cend(), 0);
}

bool exp_synthesis_helper(const vector<int> &A, const int &k,
                          list<int> &operand_list, list<char> &oper_list,
                          int cur, const int &level) {
  cur = cur * 10 + A[level] - '0';
  if (level == A.size() - 1) {
    operand_list.emplace_back(cur);
    if (evaluate(operand_list, oper_list) == k) {
      auto operand_it = operand_list.cbegin();
      cout << *operand_it++;
      for (const char &oper : oper_list) {
        cout << ' ' << oper << ' ' << *operand_it++;
      }
      cout << " = " << k << endl;
      return true;
    }
    operand_list.pop_back();
  } else {
    // No operator
    if (exp_synthesis_helper(A, k, operand_list, oper_list, cur, level + 1)) {
      return true;
    }

    // Add operator '+'
    operand_list.emplace_back(cur);
    if (k - evaluate(operand_list, oper_list) <=
        stoi(string(A.cbegin() + level + 1, A.cend()))) {  // pruning
      oper_list.emplace_back('+');
      if (exp_synthesis_helper(A, k, operand_list, oper_list, 0, level + 1)) {
        return true;
      }
      oper_list.pop_back();  // revert
    }
    operand_list.pop_back();  // revert

    // Add operator '*'
    operand_list.emplace_back(cur), oper_list.emplace_back('*');
    if (exp_synthesis_helper(A, k, operand_list, oper_list, 0, level + 1)) {
      return true;
    }
    operand_list.pop_back(), oper_list.pop_back();  // revert
  }
  return false;
}

void exp_synthesis(const vector<int> &A, const int &k) {
  list<char> oper_list;
  list<int> operand_list;
  if (exp_synthesis_helper(A, k, operand_list, oper_list, 0, 0) == false) {
    cout << "no answer" << endl;
  }
}
```

Problem 17.10, *pg. 142*: *Given a positive integer n, how would you determine the minimum number of multiplications to evaluate x^n?*

Solution 17.10: It is natural to try divide and conquer, e.g., determine the minimum number of multiplications for each of x^k and $x^{\frac{n}{k}}$, for different values of k. This does not work because the subproblems are not independent—we cannot just add the minimum number of multiplications to compute x^5 and x^6 since both may use x^3.

Instead we resort to branch and bound: we maintain a set of partial solutions which we try to extend to the final solution. The key to efficiency is pruning out partial solutions efficiently.

In our context, a partial solution is a list of exponents that we have already computed. Note that in a minimum solution, we will never have an element repeated in the list. In addition, it suffices to consider partial solutions in which the exponents occur in increasing order since if $k > j$ and x^k occurs before x^j in the chain, then x^k could not be used in the derivation of x^j. Hence we lose nothing by advancing the position of x^k.

Here is code that solves the problem:

```
list<int> get_minimum_expression(const int &n) {
  list<int> init_list;
  init_list.emplace_back(1);

  list<list<int>> exp_lists;
  exp_lists.emplace_back(init_list);
  list<int> min_exp;
  int shortest_size = numeric_limits<int>::max();

  while (exp_lists.empty() == false) {
    list<int> exp = exp_lists.front();
    exp_lists.pop_front();
    // Try all possible combinations in a list
    for (const int &i : exp) {
      for (const int &j : exp) {
        int sum = i + j;
        if (shortest_size > exp.size() + 1) {
          if (sum == n) {
            min_exp = exp;
            min_exp.emplace_back(sum);
            shortest_size = exp.size() + 1;
          } else if (sum < n && sum > exp.back()) {
            list<int> ext = exp;
            ext.emplace_back(sum);
            exp_lists.emplace_back(ext);
          }
        }
      }
    }
  }
  return min_exp;
```

```
32 }
```

If $n = 30$, the code runs in a fraction of a second. It reports $\langle x, x^2, x^3, x^5, x^{10}, x^{15}, x^{30} \rangle$. In all, 7387 partial solutions are examined.

Other bounding techniques are possible. For example, from the binary representation of 30 (11110), we know that seven multiplications suffice (compute x^2, x^4, x^8, x^{16}, and then multiply these together).

The code could avoid considering all pairs i, j and focus on pairs that just involve the last element since other pairs will have been considered previously. More sophisticated bounding can be applied: a chain like $\langle x, x^2, x^3, x^6, x^7 \rangle$ will require at least three more multiplications. The reason is k multiplications starting at a^x yield a maximum of $a^{x2^k} = a^{x \cdot 2^k}$, and $a^{x \cdot 2^k} \geq a^y$ iff $x \cdot 2^k \geq y$. Dividing by x and taking logs, we see $k \geq \lg \frac{y}{x}$. In particular, $\lg \frac{30}{7} > 2.099$ and so this chain can be safely pruned. When selecting a partial solution to continue searching from, we could choose one that is promising, e.g., the shortest solution—this might lead to better solutions faster and therefore more bounding on other search paths.

For hand calculations, these techniques are important but are trickier to code and our original code solves the given problem reasonably quickly.

Problem 17.11, *pg. 143*: *Design an algorithm for checking if a CNF expression is satisfiable.*

Solution 17.11: A reasonable way to proceed is to use backtracking. We choose a variable v, see if there exists a satisfying assignment when $v = 0$. If no such assignment exists, we try $v = 1$. If no satisfying assignment exists for $v = 0$ and for $v = 1$, the expression is not satisfiable.

Once we choose a variable and set its value, the expression simplifies—we need to remove clauses where v appears if we set $v = 1$ and remove clauses where v' appears when we set $v = 0$. In addition, whenever we get to a unit clause—one where a single literal appears—we know that in any satisfying assignment for the current expression, that literal must be set to true; this rule leads to additional simplification. Conversely, if all the clauses are true, we do not need to proceed further—every assignment to the remaining variables makes the expression true.

Variables may be chosen in various ways. One natural choice is to pick the variable which appears the most times in clauses with two literals since it leads to the most unit clauses on simplification. Another choice is to pick the variable which is the most binate—i.e., it appears the most times in negated and non-negated forms.

Problem 17.12, *pg. 143*: *How would you test the Collatz conjecture for the first n positive integers?*

Solution 17.12: Often interview questions are open-ended with no definite good solution—all you can do is provide a good heuristic and code it well. For the Collatz conjecture, the general idea is to iterate through all numbers and for each number repeatedly apply the rules till you reach 1. Here are some of the ideas that you can try to accelerate the check:

- Reuse computation by storing all the numbers you have already proven to converge to 1; that way, as soon as you reach such a number, you can assume it would reach 1.
- To save space, restrict the hash table to odd numbers.
- If you have tested every number up to k, you can stop the chain as soon as you reach a number that is less than or equal to k. You do not need to store the numbers below k in the hash table.
- If multiplication and division are expensive, use bit shifting and addition.
- Partition the search set and use many computers in parallel to explore the subsets, as show in Solution 18.11 on Page 410.

Since the numbers in a sequence may grow beyond 32 bits, you should use 64-bit integer and keep testing for overflow; alternately, you can use arbitrary precision integers.

```
bool test_Collatz_conjecture(const int &n) {
  // Stores the odd number that converges to 1
  unordered_set<long> table;

  // Start from 2 since we don't need to test 1
  for (int i = 2; i <= n; ++i) {
    unordered_set<long> sequence;
    long test_i = i;
    while (test_i != 1 && test_i >= i) {
      // A cycle means Collatz fails.
      if (sequence.emplace(test_i).second == false) {
        return false;
      }

      if (test_i & 1) {  // odd number
        if (table.emplace(test_i).second == false) {
          break;  // this number have already be proven to converge to 1
        }
        long next_test_i = 3 * test_i + 1;  // 3n + 1
        if (next_test_i <= test_i) {
          throw overflow_error("test process overflow");
        }
        test_i = next_test_i;
      } else {  // even number
        test_i >>= 1;  // n / 2
      }
    }
    table.erase(i);  // removes i from table
  }
  return true;
}
```

Problem 17.13, *pg. 143*: *You need to schedule n lectures in m classrooms. Some of those lectures are prerequisites for others. All lectures are one hour-long and start on the hour. How would you choose when and where to hold lectures to finish all the lectures as soon as possible?*

Solution 17.13: We are given a set of n unit duration lectures and m classrooms. The lectures can be held simultaneously as long as no two lectures need to happen in the same classroom at the same time and all the precedence constraints are met.

The problem of scheduling these lectures to minimize the time taken to completion is known to be NP-complete. (The same problem with an unlimited number of classrooms can be solved in polynomial time; it is the subject of Problem 16.8 on Page 136.)

This problem is naturally modeled using graphs. We model lectures as vertices, with an edge from vertex u to vertex v if u is a prerequisite for v. Clearly, the graph must be acyclic for the precedence constraints to be satisfied.

If only one lecture room exists, we can simply hold the lectures in topological order and complete the n lectures in n time (assuming each lecture is of unit duration).

We can develop heuristics using the following observation. Suppose at a given time S is a set of lectures whose precedence constraints have been satisfied. If the cardinality of S is less than or equal to m, we can schedule all the lectures in S; otherwise, we need to schedule a subset.

Subset selection can be based on several heuristic criterion.

- Rank order lectures based on the length of the longest dependency chain that they are at the start of.
- Rank order lectures based on the number of lectures that they are immediate prerequisites for.
- Rank order lectures based on the total number of lectures that they are direct or indirect prerequisites for.

We can also use combinations of these criteria to order the lectures that are currently schedulable.

If the candidate set is less than size m, we schedule all the lectures; otherwise, we choose the m most critical lectures and schedule those—the idea is that they should be scheduled sooner since they are at the start of longer dependency chains.

Problem 18.1, *pg. 145*: *Design an online spell correction system. It should take as input a string s and return an array of entries in its dictionary which are closest to the string using the Levenshtein distance specified in Problem 15.11 on Page 120. Cache the most recently computed result.*

Solution 18.1: The naïve solution would be:

```
1  public class S1 extends SpellCheckService {
2    static String wLast = null;
3    static String [] closestToLastWord = null;
4
5    public static void service(ServiceRequest req, ServiceResponse resp) {
6      String w = req.extractWordToCheckFromRequest();
7      if (!w.equals(wLast)) {
8        wLast = w;
9        closestToLastWord = Spell.closestInDictionary(w);
10     }
11     resp.encodeIntoResponse(closestToLastWord);
```

```
12    }
13  }
```

This solution has a race condition. Suppose Threads A and B run the service. Suppose Thread A updates `wLast`, and then Thread B is scheduled. Now Thread B reads `wLast` and `closestToLastWord`. Since Thread A has not updated `closestToLastWord`, if `wLast` equals the check string `w` passed to B, the cached `closestToLastWord` B returns corresponds to the previous value of `wLast`. The call to `closestToLastWord` could take quite long or be very fast, depending on the length and contents of `checkWord`. Hence it is entirely possible that Thread B reads both `wLast` and `closestToLastWord` between Thread A's updates them.

A thread-safe solution would be to declare `service` to be synchronized; in this case, only one thread could be executing the method and there is no race between write to `wLast` and `closestToLastWord`. This leads to poor performance—only one thread can be executing at a time.

The solution is to lock just the part of the code that operates on the cached values—specifically, the check on the cached value and the updates to the cached values:

```
1  public class S2 extends SpellCheckService {
2    static String wLast = null;
3    static String [] closestToLastWord = null;
4
5    public static void service(ServiceRequest req, ServiceResponse resp) {
6      String w = req.extractWordToCheckFromRequest();
7      String [] result  = null;
8      synchronized (S2.class) {
9        if (w.equals(wLast)) {
10         result = Arrays.copyOf(closestToLastWord, closestToLastWord.length);
11       }
12     }
13     if (result == null) {
14       result = Spell.closestInDictionary(w);
15       synchronized (S2.class) {
16         wLast = w;
17         closestToLastWord = result;
18       }
19     }
20     resp.encodeIntoResponse(result);
21   }
22 }
```

In the above code, multiple threads can be in their call to `closestInDictionary` which is good because the call may take a long time. Locking ensures that the read assignment on a hit and write assignment on completion are atomic. Note that we have to clone `closestToLastWord` when assigning to `result` since otherwise, `closestToLastWord` might change before we encode it into the response.

Variant 18.1.1: Threads 1 to n execute a method called `critical`. Before this, they execute a method called `rendezvous`. The synchronization constraint is that only one thread can execute `critical` at a time, and all threads must have completed

executing `rendezvous` before `critical` can be called. You can assume n is stored in a variable n that is accessible from all threads. Design a synchronization mechanism for the threads. All threads must execute the same code. Threads may call `critical` multiple times, and you should ensure that a thread cannot call `critical` a $(k+1)$-th time until all other threads have completed their k-th calls to `critical`.

Variant 18.1.2: In this problem you are to design a synchronization mechanism for a pool. This is a data structure that combines requests. Specifically, requests come from two types of threads. The pool has a capacity of four requests. A thread cannot have more than one request in the pool. When the pool is full, it must be the case that requests from both types of threads are present. Exactly one of the requesting threads must call the `launch` function when four requests are in the pool. Each thread corresponding to a request in the pool should invoke a `flush` function before `launch` is executed. Threads should call `flush` as late as possible.

Problem 18.2, ***pg.145***: *Suppose you find that the* `SimpleWebServer` *has poor performance because* `processReq` *frequently blocks on I/O. What steps could you take to improve* `SimpleWebServer`*'s performance?*

Solution 18.2: The first attempt to solve this problem might be to have `main` launch a new thread per request rather than process the request itself:

```
1  class ThreadPerTaskWebServer {
2    private static final int SERVERPORT = 8080;
3    public static void main(String [] args) throws IOException {
4     final ServerSocket serversocket = new ServerSocket(SERVERPORT);
5      while (true) {
6        final Socket connection = serversocket.accept();
7        Runnable task = new Runnable() {
8          public void run() {
9            Worker.handleRequest(connection);
10         }
11       };
12       new Thread(task).start();
13     }
14   }
15 }
```

The problem with this approach is that we do not control the number of threads launched. A thread consumes a nontrivial amount of resources, such as the time taken to start and end the thread and the memory used by the thread. For a lightly-loaded server, this may not be an issue but under load, it can result in exceptions that are challenging, if not impossible, to handle.

The right trade-off is to use a *thread pool*. As the name implies, this is a collection of threads, the size of which is bounded. Java provides thread pools through the `Executor` framework.

```
1  class TaskExecutionWebServer {
2    private static final int NTHREADS = 100;
```

```
  private static final int SERVERPORT = 8080;
  private static final Executor exec = Executors.newFixedThreadPool(NTHREADS);

  public static void main(String[] args) throws IOException {
    ServerSocket serversocket = new ServerSocket(SERVERPORT);
    while (true) {
      final Socket connection = serversocket.accept();
      Runnable task = new Runnable() {
        public void run() {
          Worker.handleRequest(connection);
        }
      };
      exec.execute(task);
    }
  }
}
```

Problem 18.3, ***pg. 146****: Implement a* `Requester` *class. The* `Execute` *method may take an indeterminate amount of time to return; it may never return. You need to have a time-out mechanism for this. Assume* `Requester` *objects have an* `Error` *method that you can invoke.*

Solution 18.3: Our strategy is to launch a thread T per `Requester` object. Thread T in turn launches another thread, S, which calls `execute` and `ProcessResponse`. The call to `execute` in S is wrapped in a try-catch `InterruptedException` loop; if `execute` completes successfully, `ProcessResponse` is called on the result.

After launching S, T sleeps for the timeout interval—when it wakes up, it interrupts S. If S has completed, nothing happens; otherwise, the try-catch `InterruptedException` calls `error`.

Code for this is given below:

```
public String execute(String req, long delay) {
  try {
    // simulate the time taken to perform a computation
    Thread.sleep(delay);
  } catch (InterruptedException e) {
    return error(req);
  }
  return execute(req);
}
public static void Dispatch(final Requestor r, final String request,
                            final long delay) {
  Runnable task = new Runnable() {
    public void run() {
      Runnable actualTask = new Runnable() {
        public void run() {
          String response = r.execute(request, delay);
          r.ProcessResponse(response);
        }
      };
      Thread innerThread = new Thread(actualTask);
      innerThread.start();
```

```
22        try {
23          Thread.sleep(TIMEOUT);
24          innerThread.interrupt();
25        } catch(InterruptedException e) {
26          e.printStackTrace();
27        }
28      }
29    };
30    new Thread(task).start();
31  }
```

Problem 18.4, ***pg. 146***: *Develop a* `Timer` *class that manages the execution of deferred tasks. The* `Timer` *constructor takes as its argument an object which includes a* `Run` *method and a* `name` *field, which is a string.* `Timer` *must support—(1.) starting a thread, identified by name, at a given time in the future; and (2.) canceling a thread, identified by name (the cancel request is to be ignored if the thread has already started).*

Solution 18.4: The two aspects to the design are the data structures and the locking mechanism.

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

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

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

Problem 18.5, ***pg. 146***: *Implement a synchronization mechanism for the first readers-writers problem.*

Solution 18.5: We want to indicate whether the string is being read as well as whether the string is being written to. We achieve this with a pair of locks—`LR` and `LW` and a read counter locked by `LR`.

A reader proceeds as follows. It locks `LR`, increments the counter, and releases `LR`. After it performs its reads, it locks `LR`, decrements the counter, and releases `LR`. A writer locks `LW`, then performs the following in an infinite loop. It locks `LR`, checks

to see if the read counter is 0; if so, it performs its write, releases LR, and breaks out of the loop. Finally, it releases LW. In the code below we use the Java `wait()` and `notify()` primitives to avoid the CPU cycles wasted in a busy wait.

```
// LR and LW are static members of type Object in the RW class.
// They serve as read and write locks. The static integer
// variable readCount in RW tracks the number of readers.
class Reader extends Thread {
  public void run() {
    while (true) {
      synchronized (RW.LR) {
        RW.readCount++;
        RW.LR.notify();
      }
      System.out.println(RW.data);
      synchronized (RW.LR) {
        RW.readCount--;
        RW.LR.notify();
      }
      Task.doSomeThingElse();
    }
  }
}

class Writer extends Thread {
  public void run() {
    while (true) {
      synchronized (RW.LW) {
      boolean done = false;
      while (!done) {
          synchronized (RW.LR) {
            if (RW.readCount == 0) {
              RW.data = new Date().toString();
              done = true;
            } else {
              // use wait/notify to avoid busy waiting
              try {
                RW.LR.wait();
              } catch (InterruptedException e) {
                System.out.println("InterruptedException in Writer wait");
              }
            }
            RW.LR.notify();
          }
        }
      }
      Task.doSomeThingElse();
    }
  }
}
```

Problem 18.6, ***pg. 147****:* *Implement a synchronization mechanism for the second readers-writers problem.*

Solution 18.6: We want to give writers the preference. We achieve this by modifying Solution 18.5 on Page 407 to have a reader start by locking `LW` and then immediately releasing `LW`. In this way, a writer who acquires the `LW` lock is guaranteed to be ahead of the subsequent readers.

Problem 18.7, *pg. 147*: *Implement a synchronization mechanism for the third readers-writers problem.*

Solution 18.7: We can achieve fairness between readers and writers by having a bit which indicates whether a read or a write was the last operation done. If the last operation done was a read, a reader on acquiring a lock must release the lock and retry—this gives writers priority in acquiring the lock; a similar operation is done by writers.

Note that this solution entails readers and writers having to wait longer than is absolutely necessary. Specifically, readers may wait even if s is opened for read and writers may wait even if no one else has a lock on s.

Variant 18.7.1: Categorical starvation refers to a phenomenon in which one category of threads make another category of threads wait indefinitely. Both Solutions 18.5 on Page 407 and 18.6 on the preceding page exhibit categorical starvation, with the readers and writers constituting the categories. Solution 18.7 on this page guarantees no categorical starvation. Thread starvation refers to a phenomenon in which a specific thread waits indefinitely while others proceed. Solve Problem 18.7 on Page 147 with the added constraint that it is free of thread starvation.

Problem 18.8, *pg. 147*: *Design a synchronization mechanism for A which ensures that P does not try to add a string into the array if it is full and C does not try to remove data from an empty buffer.*

Solution 18.8: This problem can be solved for a single producer and a single consumer with a pair of semaphores—*fillCount* is incremented and *emptyCount* is decremented whenever an item is added to the buffer. If the producer wants to decrement *emptyCount* when its count is zero, the producer sleeps. The next time an item is consumed, *emptyCount* is incremented and the producer is woken up. The consumer operates analogously. The Java methods, `wait` and `notify`, can be used to implement the desired functionality.

In the presence of multiple producers and consumers, the solution above has two races—two producers can try writing to the same slot and two consumers can read from the same slot. These races can be removed by adding mutexes around the insert and delete calls.

Problem 18.9, *pg. 147*: *Model the barber shop using semaphores and mutexes to ensure correct behavior. Each customer is a thread, as is the barber.*

Solution 18.9: A casual implementation is susceptible to races. For example, a new customer may see the barber cutting hair and go to the waiting room. Before this

customer gets to there, the barber may complete the haircut, check the waiting room, observe it to be empty, and go back to his chair to sleep. This is a form of livelock—the barber and the customer are both idle, waiting for each other. As another example, in the absence of appropriate locking, two customers may arrive simultaneously, see the barber cutting hair, and a single vacant seat in the waiting room, and go to the waiting room to occupy the single chair.

One way to achieve correct operation is to have a single mutex which allows only one person to change state at a time. The barber must acquire the mutex before checking for customers; he must release it when he either begins to sleep or begins to cut hair. A customer must acquire the mutex before entering the shop; he must release it when he sits in either a waiting room chair or the barber chair.

For a complete solution, in addition to the mutex, we need event semaphores to record the number of customers in the waiting room and the number of people getting their hair cut. The event semaphore recording the number of customers in the waiting room is used to wake up the barber when a customer enters; the event semaphore recording the number of customers getting a haircut is used to wake up waiting customers.

Problem 18.10, ***pg. 148***: *Implement a synchronization mechanism for the dining philosophers problem.*

Solution 18.10: The natural solution is for each resource to have a lock. The problem arises when each thread i first requests lock i and then lock $i + 1 \bmod n$. Since all locks have already been acquired, the thread deadlocks.

One approach is to have a central controller, which knows exactly which resources are in use and arbitrates conflicting requests. If resources are not available for a thread, the controller can reject its request.

A general principle for avoiding livelock is to order the resources and require that resources be acquired in increasing order and released in decreasing order. For example, if all threads request simultaneously, Resource $n-1$ will be left unrequested (since Thread $n-1$ will request 0 first, and then $n-1$). Thread $n-2$ will then succeed at acquiring Resource $n-1$ since Thread $n-1$ will block on Resource 0.

This solution is not starvation-free, e.g., Thread 2 can wait forever while Threads 1 and 3 alternate. To guarantee that no thread starves, track of the number of times a thread cannot execute when its neighbors release their locks. If this number exceeds some limit, the state of the thread could change to `starving` and the decision procedure to enter the critical section is supplemented to require that none of the neighbors are starving. A philosopher that cannot pick up locks because a neighbor is starving is effectively waiting for the neighbor's neighbor to finish eating. This additional dependency reduces concurrency—raising the threshold for transition to the `starving` state reduces this effect.

Problem 18.11, ***pg. 148***: *Design a multi-threaded program for checking the Collatz conjecture. Make full use of the cores available to you. To keep your program from overloading the system, you should not have more than n threads running at a time.*

Solution 18.11: Heuristics for pruning checks on individual integers are discussed in Solution 17.12 on Page 401. The focus on this problem is on implementing a multi-threaded checker. We could have a master thread launch n threads, one per number, starting with $1, 2, \ldots, x$. The master thread would keep track of what number needs to be processed next, and when a thread returned, it could re-assign it the next unchecked number.

The problem with this approach is that the time spent executing the check in an individual thread is very small compared to the overhead of communicating with the thread. The natural solution is to have each thread process a subrange of $[1, U]$. We could do this by dividing $[1, U]$ into n equal sized subranges, and having Thread i handle the i-th subrange.

The heuristics for checking the Collatz conjecture take longer on some integers than others, and in the strategy above there is the potential of a situation arising where one thread takes much longer to complete than the others, which leads to most of the cores being idle.

A good compromise is to have threads handle smaller intervals, which are still large enough to offset the thread overhead. We can maintain a work-queue consisting of unprocessed intervals, and assigning these to returning threads. The Java Executor framework is ideally suited to implementing this, and an implementation is given in the code below:

```
1  // Performs basic unit of work
2  class MyRunnable implements Runnable {
3    public int lower;
4    public int upper;
5
6    MyRunnable(int lower, int upper) {
7      this.lower = lower;
8      this.upper = upper;
9    }
10
11   @Override
12   public void run() {
13     for (int i = lower; i <= upper; ++i) {
14       Collatz.CollatzCheck(i, new HashSet<BigInteger>());
15     }
16     System.out.println("(" + lower + "," + upper + ")" );
17   }
18 }
19
20 public class Collatz {
21   // Checks an individual number
22   public static boolean CollatzCheck(BigInteger x, Set<BigInteger> visited) {
23     if (x.equals(BigInteger.ONE)) {
24       return true;
25     } else if (visited.contains(x)) {
26       return false;
27     }
28     visited.add(x);
29     if (x.getLowestSetBit() == 1) {  // odd number
```

```
30        return CollatzCheck(
31            (new BigInteger("3")).multiply(x).add(BigInteger.ONE), visited);
32      } else {  // even number
33        return CollatzCheck(x.shiftRight(1), visited); // divide by 2
34      }
35    }
36
37    public static boolean CollatzCheck(int x, Set<BigInteger> visited) {
38      BigInteger b = new BigInteger(new Integer(x).toString());
39      return CollatzCheck(b, visited);
40    }
41
42    public static ExecutorService execute() {
43      // Uses the Executor framework for task assignment and load balancing
44      List<Thread> threads = new ArrayList<Thread>();
45      ExecutorService executor = Executors.newFixedThreadPool(NTHREADS);
46      for (int i = 0 ; i < (N / RANGESIZE); ++i) {
47        Runnable worker = new MyRunnable(i * RANGESIZE + 1,
48                                         (i + 1) * RANGESIZE);
49        executor.execute(worker);
50      }
51      executor.shutdown();
52      return executor;
53    }
54  }
```

Problem 18.12, ***pg. 148***: *Design an algorithm that computes the sequence of transfers that minimizes the time taken to transfer a message from the root to all the nodes in the tree.*

Solution 18.12: We solve this problem using a straightforward bottom-up recursion. Let $T(u)$ denote the minimum number of seconds to propagate the message to the subtree rooted at u. Note that $T(u) = 0$ if u is a leaf.

Suppose u is not a leaf. Order u's children $v_0, v_1, \ldots, v_{k-1}$ by decreasing transfer times. It is straightforward to see it is optimum for u to send the messages in this order. Therefore $T(u) = \max_{0 \leq i \leq n-1}(T(v_i) + i + 1)$. The run time is dominated by the time to sort results from child nodes, leading to an $O(n \log n)$ time bound, where n is the number of nodes in the tree.

Problem 18.13, ***pg. 148***: *Devise a protocol by which hosts can elect a leader from the set of all hosts participating in the protocol. The protocol should be fast, in that it converges quickly; it should be efficient, in that it should use few connections and small messages.*

Solution 18.13: Think of the hosts as being vertices in a directed graph with an edge from A to B, if A initially know B's IP address.

We will study variants of this problem—synchronized or unsynchronized hosts, and known or unknown bounds on the number of hosts. We will compare solutions with respect to convergence time, message size, and the number of messages. We assume the graph is weakly connected (otherwise the problem is unsolvable).

First, assume that the hosts are all synchronized to a common clock (there are

standard protocols which can allow computers to synchronize within a few tens of milliseconds; alternately, Global Positioning System (GPS) signals can be used to achieve even tighter synchronization).

We first consider the case where the number of hosts n and the diameter D of the network is known to all the hosts. Our algorithm will elect the host with the highest IP address as the leader.

Since hosts are synchronized, we can proceed in rounds. The simplest algorithm for leader election is flooding—each host keeps track of the highest IP address it knows about; the highest IP address is initialized to its own IP address. In each round, host propagates the highest IP address it knows of to each of its (initial) neighbors. After D rounds, if the highest IP address a host knows of is its own, it declares itself the leader.

Here is a small improvement to this algorithm to reduce the number of messages sent—a host sends out an update only when the highest IP address it knows about changes.

It takes D rounds to converge and Dm messages are communicated, where m is the number of edges in the graph. The number of iterations to convergence can be reduced to $\lg D$ by having each host send the set of hosts it has discovered in each iteration to each host it knows about. This leads to faster convergence since the distance to the frontier of undiscovered hosts doubles in each iteration. However it requires much more communication—the final round involves n hosts sending n messages and each message has the ids of n hosts. Furthermore, unlike the original algorithm, this variant requires messages to potentially traverse longer routes (in the original algorithm, a host communicated only with the hosts it knew about initially). The algorithm works correctly even if D is just an upper bound on the true diameter.

When n and D are completely unknown, leader election can be performed through a distributed BFS. Each host starts by sending out a search message to all of its outgoing neighbors. In any round, if a host receives a search message, it chooses one of the hosts from which it received a search as its parent and informs its parent about its selection. (Since we are assuming an IP network, a child can directly communicate its selection back to its parent.)

This procedure constructs a BFS tree for each host. Completion can be detected by having hosts respond to search messages with both a parent or non-parent message as well as a notification of completion from its children. When BFS completes, each host has complete knowledge of the graph and can determine the leader.

Now, we consider the asynchronous case. The flooding algorithms we considered earlier cannot be directly generalized to asynchronous hosts because there is no notion of a round. However we can simulate rounds by having hosts tag their messages with the round number. A host waits to receive all round r messages from all its neighbors before performing its round r update. This algorithm cannot avoid sending messages if the highest IP it knows about does not change in round r since the neighbors depend on receiving all their round r messages before they can advance.

ϵ-Variant 18.13.1: Devise a protocol by which a collection of hosts on the Internet can discover each other.

Variant 18.13.2: A set of soldiers is arranged in a line. All soldiers who are not at the two ends have a copy of the same finite state machine; the soldiers at the ends may have different finite state machines. The finite state machines operate in lock-step, i.e., they all update on a common clock. Each of these finite state machines has at least three states—quiescent, excited, and firing. Design the finite state machines so that when all soldiers except for one at an end begin in a quiescent state, and the remaining soldier is in the excited state, all soldiers will enter the firing state at the same instant at some time in the future.

Problem 18.14, *pg. 149*: *How would you sort a billion* 1000 *byte strings? How about a trillion* 1000 *byte strings?*

Solution 18.14: A billion 1000 byte strings cannot fit in the RAM of a single machine, but can fit on the hard drive of a single machine. Therefore, one approach is to partition the data into smaller blocks that fit in RAM, sort each block individually, write the sorted block to disk, and then combine the sorted blocks. The sorted blocks can be merged using for example Solution 10.1 on Page 248. The UNIX `sort` program uses these principles when sorting very large files, and is faster than direct implementations of the merge-based algorithm just described.

If the data consists of a trillion 1000 byte strings, it cannot fit on a single machine—it must be distributed across a cluster of machines. The most natural interpretation of sorting in this scenario is to organize the data so that lookups can be performed via binary search. Sorting the individual datasets is not sufficient, since it does not achieve a global ordering—lookups entail a binary search on each machine. The straightforward solution is to have one machine merge the sorted datasets, but then that machine will become the bottleneck.

A solution which does away with the bottleneck is to first reorder the data so that the i-th machine stores strings in a range, e.g., Machine 3 is responsible for strings that lie between *daily* and *ending*. The range-to-machine mapping R can be computed by sampling the individual files and sorting the sampled values. If the sampled subset is small enough, it can be sorted by a single machine. The techniques in Solution 11.13 on Page 89 can be used to determine the ranges assigned to each machine. Specifically, let A be the sorted array of sampled strings. Let there be n machines. Define $r_i = iA[|A|/n]$. Then Machine i is responsible for strings in the range $[r_i, r_{i+1})$. If the distribution of the data is known *a priori*, e.g., it is uniform, the sampling step can be skipped.

The reordering can be performed in a completely distributed fashion by having each machine route the strings it begins with to the responsible machines.

After reordering, each machine sorts the strings it stores. Consequently queries such as lookups can be performed by using R to determine which individual machine to forward the lookup to.

Problem 18.15, *pg. 149*: *Implement crawling under the constraint that in any given minute your crawlers do not request more than b bytes from any website.*

Solution 18.15: This problem, as posed, is ambiguous.

- Since we usually download one file in one request, if a file is greater than b bytes, there is no way we can meet the constraint of serving fewer than b bytes every minute, unless we can work with the lower layers of networking stack such as the transport layer or the network layer. Often the system designer could look at the distribution of file sizes and conclude that this problem happens so infrequently that we do not care. Alternately, we may choose to download no more than the first b bytes of any file.
- Given that the host's bandwidth is a resource for which there could be contention, one important design choice to be made is how to resolve a contention. Do we let requests get served in first-come first-served order or is there a notion of priority? Often crawlers have a built-in notion of priority based on how important the document is to the users or how fresh the current copy is.

One way of doing this could be to maintain a permission server with which each crawler checks to see if it is okay to hit a particular host. The server can keep an account of how many bytes have been downloaded from the server in the last minute and not permit any crawler to hit the server if we are already close to the quota. If we do not care about priority, then we can keep the interface synchronous where a server requests permission to download a file and it immediately gets approved or denied. If we care about priorities, then the server may enqueue the request and inform the crawler when it is alright to download the file. The queues at the permission server may be based on priorities.

If the permission server becomes a bottleneck, we can use multiple permission servers such that the responsibility of a given host is decided by applying a hash function to the host name and assigning it to a particular server based on the hash code.

Problem 19.1, *pg. 150*: *Design a program that produces high quality mosaics with minimal compute time.*

Solution 19.1: A good way to begin is to partition the image into $s \times s$-sized squares, compute the average color of each such image square, and then find the tile that is closest to it in the color space. Distance in the color space can be the $L2$-distance over the Red-Green-Blue (RGB) intensities for the color. As you look more carefully at the problem, you might conclude that it would be better to match each tile with an image square that has a similar structure. One way could be to perform a coarse pixelization (2×2 or 3×3) of each image square and finding the tile that is "closest" to the image square under a distance function defined over all pixel colors. In essence, the problem reduces to finding the closest point from a set of points in a k-dimensional space.

Given m tiles and an image partitioned into n squares, then a brute-force approach would have $O(mn)$ time complexity. You could improve on this by first indexing the

tiles using an appropriate search tree. You can also run the matching in parallel by partitioning the original image into subimages and searching for matches on the subimages independently.

Problem 19.2, *pg. 150*: *Given a million documents with an average size of* 10 *kilobytes, design a program that can efficiently return the subset of documents containing a given set of words.*

Solution 19.2: The predominant way of doing this is to build inverted indices. In an inverted index, for each word, we store a sequence of locations where the word occurs. The sequence itself could be represented as an array or a linked list. Location is defined to be the document ID and the offset in the document. The sequence is stored in sorted order of locations (first ordered by document ID, then by offset). When we are looking for documents that contain a set of words, what we need to do is find the intersection of sequences for each word. Since the sequences are already sorted, the intersection can be done in time proportional to the aggregate length of the sequences. We list a few optimizations below.

- *Compression*—compressing the inverted index helps both with the ability to index more documents as well as memory locality (fewer cache misses). Since we are storing sorted sequences, one way of compressing is to use delta compression where we only store the difference between the successive entries. The deltas can be represented in fewer bits.
- *Caching*—the distribution of queries is usually fairly skewed and it helps a great deal to cache the results of some of the most frequent queries.
- *Frequency-based optimization*—since search results often do not need to return every document that matches (only top ten or so), only a fraction of highest quality documents can be used to answer most of the queries. This means that we can make two inverted indices, one with the high quality documents that stays in RAM and one with the remaining documents that stays on disk. This way if we can keep the number of queries that require the secondary index to a small enough number, then we can still maintain a reasonable throughput and latency.
- *Intersection order*—since the total intersection time depends on the total size of sequences, it would make sense to intersect the words with smaller sets first. For example, if we are looking for "USA GDP 2009", it would make sense to intersect the lists for GDP and 2009 before trying to intersect the sequence for USA.

We could also build a multilevel index to improve accuracy on documents. For a high priority web page, we can decompose the page into paragraphs and sentences, which are indexed individually. That way the intersections for the words might be within the same context. We can pick results with closer index values from these sequences. See the sorted array intersection problem 13.5 on Page 99 and digest problem 12.14 on Page 96 for related issues.

Problem 19.3, *pg. 151*: *You are given a large set of strings S. Given a query string Q, how*

would you design a system that can quickly identify the longest string $p \in S$ that is a prefix of Q?

Solution 19.3: This is a well studied problem because of its implications to building a high speed Internet backbone. A number of approaches have been proposed and used in IP routers. One simple approach is to build a trie data structure such that we can traverse the trie for an IP address till we hit a node that has a label. This essentially requires one pointer indirection per bit of input. The lookup speed can be improved a little at the cost of memory by making fatter nodes in the trie that consume multiple bits at a time. See Solution 9.14 on Page 247 for more details.

A number of approaches have been tried in software and hardware to speed the lookup process:

- Binary search on hash tables—we can have one hash table for each possible prefix length and then do a search for the longest matching prefix by searching through the hash tables. This can take as many as 32 hash table lookups in the worst case. One way to reduce the number of lookups is to perform binary search for the longest matching prefix. For binary search to work, we have to insert additional prefixes in the hash tables to ensure that if a longer prefix exists, binary search does not terminate early. This can be done by performing a binary search for each prefix and inserting additional dummy entries wherever the binary search terminates early. This could inflate the size of hash tables by a factor of $\lg 32 = 5$; in practice the blowup is much smaller.
- Ternary Content Addressable Memory (TCAM). In a conventional RAM, the user supplies an address, and the RAM outputs the data stored at that address. In a Content Addressable Memory (CAM), the user supplies a key, and the CAM returns a Boolean indicating if the key is stored at any address. Depending on the CAM, it may also return the lowest address that stores that key, and possibly a corresponding value. A TCAM is a specialized CAM, where instead of storing 0s and 1s, a single unit of memory can also store a third state called the "don't care" state. The contents of memory can be addressed by partial contents of the memory. A TCAM where each memory location stores 32 ternary values can be used to store prefixes. Each prefix is padded with "don't care" bits to make it 32 ternary values. This way, when we use an IP address to address the TCAM, we get all the matching prefixes. If we store longer prefixes at lower memory locations, the TCAM will return the longest matching prefixes.

Problem 19.4, *pg. 151*: *How would you build a spelling correction system?*

Solution 19.4: The basic idea behind most spelling correction systems is that the misspelled word's Levenshtein distance from the intended word tends to be very small (one or two edits). Hence if we keep a hash table for all the words in the dictionary and look for all the words that are within two Levenshtein distances of the text, most likely, the intended word will be found in this set. If the alphabet has m characters and the search text has n characters, we would need to perform roughly nm^2 hash table lookups. The intersection of set of all strings at a distance of two or

less from a word and the set of dictionary words may be large. It is important to provide a ranked list of suggestions to the users, with the most likely candidates are at the beginning of the list. There are several ways to achieve this.

- Typing errors model—often spelling mistakes are a result of typing errors. Typing errors can be modeled based on keyboard layouts.
- Phonetic modeling—a big class of spelling errors happen when the person spelling it knows how the words sounds but does not know the exact spelling. In such cases, it helps to map the text to phonemes and then find all the words that map to the same phonetic sequence.
- History of refinements—often users themselves provide a great amount of data about the most likely misspellings by first entering a misspelled word and then correcting it. This historic data is often immensely valuable for spelling correction.
- Stemming—often the size of a dictionary can be reduced by keeping only the stemmed version of each word. (This entails stemming the query text.)

Problem 19.5, ***pg. 151***: *Design a stemming algorithm that is fast and effective.*

Solution 19.5: Stemming is a large topic. Here we mention some basic ideas related to stemming, however this is in no way a comprehensive discussion on stemming approaches.

Most stemming systems are based on simple rewrite rules, e.g., remove suffixes of the form "es", "s", and "ation". Suffix removal does not always work. For example, wolves should be stemmed to wolf. To cover this case, we may have a rule that replaces the suffix "ves" with "f".

Most rules amount to matching a set of suffixes and applying the corresponding transformation to the string. One way of efficiently performing this is to build a finite state machine based on all the rules.

A more sophisticated system might have exceptions to the broad rules based on the stem matching some patterns. The Porter stemmer, developed by Martin Porter, is considered to be one of the most authoritative stemming algorithms in the English language. It defines several rules based on patterns of vowels and consonants.

Other approaches include the use of stochastic methods to learn rewrite rules and n-gram based approaches where we look at the surrounding words to determine the correct stemming for a word.

Problem 19.6, ***pg. 152***: *How would you implement TEX?*

Solution 19.6: Note that the problem does not ask for the design of TEX, which itself is a complex problem involving feature selection, and language design. There are a number of issues common to implementing any such program: programming language selection, lexing and parsing input, error handling, macros, and scripting.

Two key implementation issues specific to TEX are a specifying fonts and symbols (e.g., $\mathtt{A}, \mathtt{b}, f, \sum, \oint, \uplus$), and assembling a document out of components.

Focusing on the second aspect, a reasonable abstraction is to use a rectangular bounding box to describe components. The description is hierarchical: each individual symbol is a rectangle, lines and paragraphs are made out of these rectangles and are themselves rectangles, as are section titles, tables and table entries, and included images. A key algorithmic problem is to assemble these rectangles, while preserving hard constraints on layout, and soft constraints on aesthetics. See also Problem 15.13 on Page 121 for an example of the latter.

Turning our attention to symbol specification, the obvious approach is to use a 2D array of bits to represent each symbol. This is referred to as a bit-mapped representation. The problem with bit-mapped fonts is that the resolution required to achieve acceptable quality is very high, which leads to huge documents and font-libraries. Different sizes of the same symbol need to be individually mapped, as do italicized and bold-face versions.

A better approach is to define symbols using mathematical functions. A reasonable approach is to use a language that supports quadratic and cubic functions, and elementary graphics transformations (rotation, interpolation, and scaling). This approach overcomes the limitations of bit-mapped fonts—parameters such as aspect ratio, font slant, stroke width, serif size, etc. can be programmed.

Other implementation issues include enabling cross-referencing, automatically creating indices, supporting colors, and outputting standard page description formats (e.g., PDF)).

Donald Knuth's book "*Digital Typography*" describes in great detail the design and implementation of TEX.

Problem 19.7, *pg. 152*: *Implement the UNIX* `tail` *command.*

Solution 19.7: The natural approach to this problem is to read the input one line at a time. Each line can be stored in a queue—when the queue size is equal to the number of desired lines, each additional line is inserted at the tail, and the line at the head is deleted. (A circular buffer is a particularly appropriate implementation for this application.) The drawback of this approach is that it entails reading the entire file which could be huge.

The UNIX OS provides the ability to perform random access on a file, essentially allowing us to treat the file as an array of characters, albeit with much slower access times. This capability is exposed to C++ programmers through the `seekg` ("seek get") function in the `istream` library. In the code below, we use `seekg` to process the file in reverse order starting the end of the file. We store the characters in a string, stopping when the specified number of lines have been read.

```
1 string tail(const string &file_name, const int &tail_count) {
2   fstream file_ptr(file_name.c_str());
3
4   file_ptr.seekg(0, ios::end);
5   int file_size = file_ptr.tellg(), newline_count = 0;
6   string output;  // stores the last tail_count lines
7   // Reads file in reverse looking for '\n'
8   for (int i = 0; i < file_size; ++i) {
```

```
    file_ptr.seekg(file_size - i - 1, ios::beg);
    char c;
    file_ptr.get(c);
    if (c == '\n') {
      ++newline_count;
      if (newline_count > tail_count) {
        break;
      }
    }
    output.push_back(c);
  }
  // Reverse the output string using the reverse function
  // from the <algorithm> library in STL. The arguments
  // are iterators to the start and end of String object.
  reverse(output.begin(), output.end());
  return output;
}
```

Problem 19.8, *pg. 152*: *Design a feature that allows a studio to enter a set V of videos that belong to it, and to determine which videos in the YouTV.com database match videos in V.*

Solution 19.8: If we replaced videos everywhere with documents, we could use the techniques in Solution 12.13 on Page 286, where we looked for near duplicate documents by computing hash codes for each length-k substring.

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

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

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

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

Variant 19.8.1: Design an online music identification service.

Problem 19.9, *pg. 152*: *Design a system that can compute the ranks of ten billion web pages*

in a reasonable amount of time.

Solution 19.9: Since the web graph can have billions of vertices and it is mostly a sparse graph, it is best to represent the graph as an adjacency list. Building the adjacency list representation of the graph may require a significant amount of computation, depending upon how the information is collected. Usually, the graph is constructed by downloading the pages on the web and extracting the hyperlink information from the pages. Since the URL of a page can vary in length, it is often a good idea to represent the URL by a hash code.

The most expensive part of the PageRank algorithm is the repeated matrix multiplication. Usually, it is not possible to keep the entire graph information in a single machine's RAM. Two approaches to solving this problem are described below.

- Disk-based sorting—we keep the column vector X in memory and load rows one at a time. Processing Row i simply requires adding $A_{i,j}X_j$ to X_j for each j such that $A_{i,j}$ is not zero. The advantage of this approach is that if the column vector fits in RAM, the entire computation can be performed on a single machine. This approach is slow because it uses a single machine and relies on the disk.
- Partitioned graph—we use n servers and partition the vertices (web pages) into n sets. This partition can be computed by partitioning the set of hash codes in such a way that it is easy to determine which vertex maps to which machine. Given this partitioning, each machine loads its vertices and their outgoing edges into RAM. Each machine also loads the portion of the PageRank vector corresponding to the vertices it is responsible for. Then each machine does a local matrix multiplication. Some of the edges on each machine may correspond to vertices that are owned by other machines. Hence the result vector contains nonzero entries for vertices that are not owned by the local machine. At the end of the local multiplication it needs to send updates to other hosts so that these values can be correctly added up. The advantage of this approach is that it can process arbitrarily large graphs.

PageRank runs in minutes on a single machine on the graph consisting of the six million pages that constitute Wikipedia. It takes roughly 70 iterations to converge on this graph. Anecdotally, PageRank takes roughly 200 iterations to converge on the web graph.

Problem 19.10, *pg. 152*: *Design a system for maintaining a set of prioritized jobs that implements the following API:*

1. *Insert a new job with a given priority.*
2. *Delete a job.*
3. *Find the highest priority job.*

Each job has a unique ID. Assume the set cannot fit into a single machine's memory.

Solution 19.10: If we have enough RAM on a single machine, the most simple solution would be to maintain a min-heap where entries are ordered by their priority. An additional hash table can be used to map jobs to their corresponding entry in the min-heap to make deletions fast.

A more scalable solution entails partitioning the problem across multiple machines. One approach is to apply a hash function to the job ids and partition the resulting hash codes into ranges, one per machine. Insert as well as delete require communication with just one server. To do extract-min, we send a lookup minimum message to all the machines, infer the min from their responses, and then delete it.

At a given time many clients may be interested in the the highest priority event, and it is challenging to distribute this problem well. If many clients are trying to do this operation at the same time, we may run into a situation where most clients will find that the min event they are trying to extract has already been deleted. If the throughput of this service can be handled by a single machine, we can make one server solely responsible for responding to all the requests. This server can prefetch the top hundred or so events from each of the machines and keep them in a heap.

In many applications, we do not need strong consistency guarantees. We want to spend most of our resources taking care of the highest priority jobs. In this setting, a client could pick one of the machines at random, and request the highest priority job. This would work well for the distributed crawler application. It is not suited to event-driven simulation because of dependencies.

Problem 19.11, ***pg. 153***: *You have guaranteed your clients that 99% of their requests will be serviced in less than one second. How would you design a system to meet this requirement with minimal cost?*

Solution 19.11: Suppose at a given time no more than a fixed number of requests can be served concurrently; pending requests must wait for a slot to open up before they can be served. It is important to queue requests in such a way that the requests that take a long time to serve do not block a large number of short requests behind them.

Suppose the time it takes for the server to process a request is a known easy-to-compute function of the internals of the request, and the service time follows a Pareto distribution. In such cases, the 99-th percentile latency is dramatically reduced by maintaining a short-request queue and a long-request queue. A threshold is used to assign requests to queues. Requests that take longer than the threshold go to the long-request queue; the remainder are assigned to the short-request queue. We pick the threshold to make most requests go to the queue of short requests queue. The requests in the short-request queue are never blocked behind a long-running request. The longer requests do have longer to wait, but overall this strategy is extremely effective at reducing the 99-th percentile latency.

Often the system designer does not know how long a given request will take. Even in this case, it is advantageous to keep two queues. When a request comes in, it is put in the short-request queue. If it takes longer than a certain threshold T_c, it is canceled and added to the long-request queue. Simulation studies and experimentation can be used to derive a suitable choice for T_c.

Problem 19.12, ***pg. 153***: *Jingle, a search engine startup, wants to monetize its search results by displaying advertisements alongside search results. Design an online advertising system for Jingle.*

Solution 19.12: Reasonable goals for such a system include

- providing users with the most relevant ads,
- providing advertisers the best possible return on their investment, and
- minimizing the cost and maximizing the revenue to Jingle.

Two key components for such a system are:

- The front-facing component, which advertisers use to create advertisements, organize campaigns, limit when and where ads are shown, set budgets, and create performance reports.
- The ad-serving system, which selects which ads to show on the searches.

The front-facing system can be a fairly conventional web application, i.e., a set of web pages, middleware that responds to user requests, and a database. Key features include:

- User authentication—a way for users to create accounts and authenticate themselves. Alternately, use an existing single sign-on login service, e.g., Facebook or Google.
- User input—a set of form elements to let advertisers specify ads, advertising budget, and search keywords to bid on.
- Performance reports—a way to generate reports on how the advertiser's money is being spent.
- Customer service—even the best of automated systems require occasional human interaction, e.g., ways to override limits on keywords. This requires an interface for advertisers to contact customer service representatives, and an interface for those representatives to interact with the system.

The whole front-end system can be built using, for example, HyperText Markup Language (HTML) and JavaScript. A commonly used approach is to use a LAMP stack on the server-side: Linux as the OS, Apache as the HTTP server, MySQL as the database software, and PHP for the application logic.

The ad-serving system is less conventional. The ad-serving system would build a specialized data structure, such as a decision tree, from the ads database. It chooses ads from the database of ads based on their "relevance" to the search. In addition to keywords, the ad-serving systems can use knowledge of the user's search history, how much the advertiser is willing to pay, the time of day, user locale, and type of browser. Many strategies can be envisioned here for estimating relevance, such as, using information retrieval or machine learning techniques that learn from past user interactions.

The ads could be added to the search results by embedding JavaScript in the results page. This JavaScript pulls in the ads from the ad-serving system directly. This helps isolate the latency of serving search results from the latency of serving ad results.

Problem 19.13, *pg. 153*: *Design a system that automatically generates a sidebar of related articles.*

Solution 19.13: The key technical challenge in this problem is to come up with the list of articles—the code for adding these to a sidebar is trivial.

One suggestion might be to add articles that have proven to be popular recently. Another is to have links to recent news articles. A human reader at Jingle could tag articles which he believes to be significant. He could also add tags such as finance, sports, and politics, to the articles. These tags could also come from the HTML meta-tags or the page title.

We could also provide randomly selected articles to a random subset of readers and see how popular these articles prove to be. The popular articles could then be shown more frequently.

On a more sophisticated level, Jingle could use automatic textual analysis, where a similarity is defined between pairs of articles—this similarity is a real number and measures how many words are common to the two. Several issues come up, such as the fact that frequently occurring words such as "for" and "the" should be ignored and that having rare words such as "arbitrage" and "diesel" in common is more significant than having say, "sale" and "international".

Textual analysis has problems, such as the fact that two words may have the same spelling but completely different meanings (anti-virus means different things in the context of articles on acquired immune deficiency syndrome (AIDS) and computer security). One way to augment textual analysis is to use collaborative filtering—using information gleaned from many users. For example, by examining cookies and timestamps in the web server's log files, we can tell what articles individual users have read. If we see many users have read both A and B in a single session, we might want to recommend B to anyone reading A. For collaborative filtering to work, we need to have many users.

Problem 19.14, ***pg. 154***: *Design a driving directions service with a web interface.*

Solution 19.14: At its core, a driving directions service needs to store the map as a graph, where each intersection and street address is a vertex and the roads connecting them are edges. When a user enters a starting address and an ending address, it finds the corresponding vertices and finds the shortest path connecting the two vertices (for some definition of shortest). Issues include:

- Address normalization—a given address may be expressed by the user in different ways, for example, "street" may be shortened to "st", there may not be a city and state, just zip code or vice versa. We need a way to normalize the addresses to a standard format. Sometimes an underspecified address may need to be mapped to some concrete address, for example, a city name to the city center.
- Definition of shortest—different users may have different preferences for routing, for example, shortest distance or fastest path (considering average speed on the road), and avoiding use of freeways. Each of these preferences can be captured by some notion of edge length.
- Approximate shortest distance—given the enormity of a graph representing all the roads in a large country, it would be fairly difficult for a single server to compute the shortest path using standard shortest path algorithms and return in a reasonable amount of time. However using the knowledge that most long

paths go through a standard system of highways and the fact that the vertices and edges in the graph represent points in Euclidean space, we can devise some clever approximation algorithms that run much faster.

The imagery displayed on the web UI can be made out of "tiles"—smaller individual images. JavaScript handlers pull in more tiles when the user requests a zoom or moves to a neighboring region. Tiles can be pre-fetched to improve perceived responsiveness.

Problem 19.15, *pg. 154*: *Design an efficient way of copying one thousand files each* 100 *kilobytes in size from a single lab server to each of* 1000 *servers in a distant data center.*

Solution 19.15: Assume that the bandwidth from the lab machine is a limiting factor. It is reasonable to first do trivial optimizations, such as combining the articles into a single file and compressing this file.

Opening 1000 connections from the lab server to the 1000 machines in the data center and transferring the latest news articles is not feasible since the total data transferred will be approximately 100 gigabytes (without compression).

Since the bandwidth between machines in a data center is very high, we can copy the file from the lab machine to a single machine in the data center and have the machines in the data center complete the copy. Instead of having just one machine serve the file to the remaining 999 machines, we can have each machine that has received the file initiate copies to the machines that have not yet received the file. In theory, this leads to an exponential reduction in the time taken to do the copy.

Several additional issues have to be dealt with. Should a machine initiate further copies before it has received the entire file? (This is tricky because of link or server failures.) How should the knowledge of machines which do not yet have copies of the file be shared? (There can be a central repository or servers can simply check others by random selection.) If the bandwidth between machines in a data center is not a constant, how should the selections be made? (Servers close to each other, e.g., in the same rack, should prefer communicating with each other.)

Finally, it should be mentioned that there are open source solutions to this problem, e.g., Unison and BitTorrent, which would be a good place to start.

Problem 19.16, *pg. 154*: *Design an online poker playing service for Clump Enterprises. Describe both the system architecture and a set of classes.*

Solution 19.16: An online poker playing service would have a front-end system which users interact with and a back-end system which runs the games, manages money, and looks for fraud.

The front-end system would entail a UI for account management—this would cover first-time registration, logging-in, managing online persona, and sending or receiving money. In addition, there would be the game playing UI—this could be as simple as some HTML rendering of the state of the game (cards in hand, cards on the table, bets) and a form to enter a bet. A more sophisticated UI might use JavaScript

to animate the dealing of cards, to change the expression on player's images, and to display status messages.

The back-end needs to form tables of players, shuffle in a truly random manner, deal correctly, check if the player's moves are legal, and update player's finances. It can be implemented using, for example, a Java servlet engine which receives HTTP requests, sends appropriate responses, and updates the database appropriately.

One of the big challenges in such a system is fault-tolerance. On the server side, there exist standard techniques for fault-tolerance, such as replication.

On the client side, there exists the possibility that a player may realize he is in a poor situation and claim that his Internet connection went down. This can be resolved by having a rule that the server will bid on the player's behalf if the player does not respond quickly enough. Another possibility is having the server treat the disconnected player as being in the game but not requiring any more betting of him. This clearly can be abused by the player, so the server needs to record how often a player's connection hangs in a way that is favorable to him.

Collusion among players is another serious problem. Again, the server logs can be mined for examples of players working together to share knowledge of their cards or squeeze other players out. In addition, players can themselves flag suspicious play and customer service representatives can investigate further.

Random number generation is an intensely studied problem but is still often done wrong. A fairly frequent problem is using the ID of the process to seed the random number generator, which means that there are not more than 32768 possible sequences of shuffles on most UNIX systems. This is much less than the 52! possible card sequences.

Now we turn our attention to class design. We begin with the simplest classes: a card class and a deck of cards class. The first has two data members, the suit and the rank, both of which should be enumerated types. The second has a single data field, namely a list of cards; appropriate member functions would be a shuffling routine and a deal n cards function.

Players are modeled by a class that specifies their current hand, and the amount of money they have. Attributes such as name, picture, and location, are best inherited from a user class. Methods include actions to bet/fold, and request cards.

By the rules of poker, a player can bet all that he has left and remain in the game, getting the corresponding share of the pot if he wins. Therefore the pot class should specify how much money has been bet, and who it has been bet by.

The game class itself consists of a deck, a list of players, and a pot. It is responsible for ensuring that players take an appropriate amount of time to play, and that all moves are legal.

Problem 19.17, *pg. 154*: *Design the World Wide Web. Specifically, describe what happens when you enter a URL in a browser address bar, and press return.*

Solution 19.17: At the network level, the browser extracts the domain name component of the URL, and determines the IP address of the server, e.g., through a call to a Domain Name Server (DNS), or a cache lookup. It then communicates using

the HTTP protocol with the server. HTTP itself is built on top of TCP/IP, which is responsible for routing, reassembling, and resending packets, as well as controlling the transmission rate.

The server determines what the client is asking for by looking at the portion of the URL that comes after the domain name, and possibly also the body of the HTTP request. The request may be for something as simple a file, which is returned by the webserver; HTTP spells out a format by which the type of the returned file is specified. For example, the URL http://go.com/imgs/abc.png may encode a request for the file whose hierarchical name is imgs/abc.png relative to a base directory specified at configuration to the web server.

The URL may also encode a request to a service provided by the web server. For example, http://go.com/lookup/flight?num=UA37,city=AUS is a request to the lookup/flight service, with an argument consisting of two attribute-value pair. The service could be implemented in many ways, e.g., Java code within the server, or a Common Gateway Interface (CGI) script written in Perl. The service generates a HTTP response, typically HTML, which is then returned to the browser. This response could encode data which is used by scripts running in the browser. Common data formats include JavaScript Object Notation (JSON) and Extensible Markup Language (XML).

The browser is responsible for taking the returned HTML and displaying it on the client. The rendering is done in two parts. First, a parse tree (the Document Object Model (DOM)) is generated from the HTML, and then a rendering library "paints" the screen. The returned HTML may include scripts written in JavaScript. These are executed by the browser, and they can perform actions like making requests and updating the DOM based on the responses—this is how a live stock ticker is implemented. Styling attributes (Cascading Style Sheets (CSS)) are commonly used to customize the look of a page.

Many more issues exist on both the client and server side: security, cookies, HTML form elements, HTML styling, and handlers for multi-media content, to name a few.

Problem 20.1, *pg. 156*: *Does the following process yield a uniformly random permutation of A? "For $i \in \{0, 1, \ldots, n-1\}$, swap $A[i]$ with a randomly chosen element of A." (The randomly chosen element could be i itself.)*

Solution 20.1: It does not yield all permutations with equal probability. One way to see this is to consider the case $n = 3$. The number of permutations is $3! = 6$. The total number of ways in which we can choose the elements to swap is $3^3 = 27$ and all are equally likely. Since 27 is not divisible by 6, some permutations correspond to more ways than others, ergo not all permutations are equally likely.

The process can be fixed by selecting elements at random and moving them to the end, similar to how we proceeded in Problems 20.2 on Page 156 and 20.7 on Page 157.

Problem 20.2, *pg. 156*: *Let A be an array of n distinct elements. Design an algorithm that returns a subset of k elements of A. All subsets should be equally likely. Use as few calls to the random number generator as possible and use $O(1)$ additional storage. You can return the result in the same array as input.*

Solution 20.2: The problem is trivial when $k = 1$—we simply make one call to the random number generator, take the returned r value mod n. We can swap $A[n-1]$ with $A[r]$; $A[n-1]$ then holds the result.

For $k > 1$, we start by choosing one element at random as above and we now repeat the same process with the $n-1$ element subarray $A[0 : n-2]$. Eventually, the random subset occupies the slots $A[n-k : n-1]$ and the remaining elements are in the first $n-k$ slots.

The algorithm clearly runs in $O(1)$ space. To show that all the subsets are equally likely, we prove something stronger, namely that all permutations of size k are equally likely.

Formally, an m-permutation of a set S of cardinality n is a sequence of m elements of S with no repetitions. It is easily verified that the number of m-permutations is $\frac{n!}{(n-m)!}$.

The induction hypothesis now is that after iteration m, the subarray $A[n-m : n-1]$ contains each possible m-permutation with probability $\frac{(n-m)!}{n!}$.

The base case holds since for $m = 1$, any element is equally likely to be selected.

Suppose the inductive hypothesis holds for $m = l$. Now we study $m = l+1$. Consider a particular $(l+1)$-permutation, say $\langle \alpha_1, \ldots, \alpha_{l+1} \rangle$. This consists of a single element α_1 followed by the l-permutation $\langle \alpha_2, \ldots, \alpha_{l+1} \rangle$. Let E_1 be the event that α_1 is selected in iteration $l+1$ and E_2 be the event that the first l iterations produced $\langle \alpha_2, \ldots, \alpha_{l+1} \rangle$. The probability of $\langle \alpha_1, \ldots, \alpha_{l+1} \rangle$ resulting after iteration $l+1$ is simply $\Pr(E_1 \cap E_2) = \Pr(E_1 \mid E_2)\Pr(E_2)$. By the inductive hypothesis, the probability of permutation $\langle \alpha_2, \ldots, \alpha_{l+1} \rangle$ is $\frac{(n-l)!}{n!}$. The probability $\Pr(E_1|E_2) = \frac{1}{n-l}$ since the algorithm selects from elements in the subarray $A[0 : n-l-1]$ with equal probability. Therefore

$$\Pr(E_1 \cap E_2) = \Pr(E_1|E_2)\Pr(E_2) = \frac{1}{n-l}\frac{(n-l)!}{n!} = \frac{(n-l-1)!}{n!}$$

and induction goes through.

The algorithm generates all random k-permutations with equal probability, from which it follows that all subsets of size k are equally likely.

The algorithm just described makes k calls to the random number generator. When k is bigger than $\frac{n}{2}$, we can optimize by computing a subset of $n-k$ elements to remove from the set. For example, when $k = n-1$, this replaces $n-1$ calls to the random number generator with a single call. Of course, while all subsets are equally likely with this optimization, all permutations are not. Following is the code in C++:

```
template <typename T>
vector<T> offline_sampling(vector<T> A, const int &k) {
  for (int i = 0; i < k; ++i) {
    default_random_engine gen((random_device())());  // random num generator
    // Generate random int in [i, A.size() - 1]
    uniform_int_distribution<int> dis(i, A.size() - 1);
    swap(A[i], A[dis(gen)]);
  }
  A.resize(k);
  return A;
```

```
11 }
```

Variant 20.2.1: The `rand()` function in the standard C library returns a uniformly random number in $[0, \texttt{RAND\_MAX} - 1]$. Does `rand()` mod n generate a number uniformly distributed $[0, n-1]$?

Problem 20.3, *pg. 156*: *Design an algorithm that creates uniformly random permutations of $\{0, 1, \ldots, n-1\}$. You are given a random number generator that returns integers in the set $\{0, 1, \ldots, n-1\}$ with equal probability; use as few calls to it as possible.*

Solution 20.3: Solution 20.2 on Page 427 can be used with $k = n$. Although the subset that is returned is unique (it will be $\{0, 1, \ldots, n-1\}$), all $n!$ possible orderings of the elements in the set occur with equal probability. (Note that we cannot use the trick to reduce the number of calls to the random number generator at the end of Solution 20.2 on Page 427.)

Problem 20.4, *pg. 156*: *How would you implement a random number generator that generates a random integer i in $[a, b]$, given a random number generator that produces either zero or one with equal probability? All generated values should have equal probability. What is the run time of your algorithm, assuming each call to the given random number generator takes $O(1)$ time?*

Solution 20.4: Basically, we want to produce a random integer in $[0, b-a]$. Let $l = b - a + 1$. We can produce a random integer in $[0, l-1]$, as follows. Let i be the least integer such that $l \leq 2^i$.

If l is a power of 2, say $l = 2^i$, then all we need are i calls to the 0-1 valued random number generator—the i bits from the calls encode an i bit integer in $[0, l-1]$, and all such numbers are equally likely; so, we can use this integer.

If l is not a power of 2, the i calls may or may not encode an integer in the range 0 to $l-1$. If the number is in the range, we return it; since all the numbers are equally likely, the result is correct.

If the number is outside the range $[0, l-1]$, we try again. The probability of having to try again is less than $\frac{1}{2}$ since $l > 2^{i-1}$. Therefore the probability that we take exactly k steps before succeeding is at most $\frac{1}{2}(1-\frac{1}{2})^{k-1} = \frac{1}{2^k}$. This implies the expected number of trials is less than $1\frac{1}{2} + 2(\frac{1}{2})^2 + 3(\frac{1}{2})^3 + \cdots$. Differentiating the identity $\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots$, yields the identity $\frac{1}{(1-x)^2} = 1 + 2x + 3x^2 + 4x^3 + \cdots$. Multiplying both sides by x demonstrate that $\frac{x}{(1-x)^2} = x + 2x^2 + 3x^3 + 4x^4 + \cdots$. Substituting $\frac{1}{2}$ for x in this last identity proves that $1(\frac{1}{2}) + 2(\frac{1}{2})^2 + 3(\frac{1}{2})^3 + \cdots = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$. Therefore the expected number of trials is less than 2.

```
1 int uniform_random_a_b(const int &a, const int &b) {
2   int l = b - a + 1, res;
3   do {
4     res = 0;
5     for (int i = 0; (1 << i) < l; ++i) {
```

```
6       // zero_one_random is the system-provided random number generator
7       res = (res << 1) | zero_one_random();
8     }
9   } while (res >= 1);
10  return res + a;
11 }
```

Problem 20.5, *pg. 157*: *You are given a set T of n nonnegative real numbers $\{t_0, t_1, \ldots, t_{n-1}\}$ and probabilities $p_0, p_1, \ldots, p_{n-1}$, where $\sum_{i=0}^{n-1} p_i = 1$. Assume that $t_0 < t_1 < \cdots < t_{n-1}$. Given a random number generator that produces values in $[0,1]$ uniformly, how would you generate a value X from T according to the specified probabilities?*

Solution 20.5: Let $F_X(\alpha)$ be the probability that $X \leq \alpha$, i.e., $F_X(\alpha) = \sum_{t_i \leq \alpha} p_i$. We do the following operation to generate a random value according to X's distribution. Select a number r uniformly at random in the unit interval, $[0,1]$, then project back from F_X to obtain a value s for X. More specifically, we return the largest t_i such that $F_X(t_i) \leq r$.

By construction, the probability that the value s we return is less than or equal to α is $F_X(\alpha)$, so the distribution for s is exactly the same as that for X.

ϵ-Variant 20.5.1: Given a random number generator that produces values in $[0,1]$ uniformly, how would you generate a value X from T according to a continuous probability distribution, such as the exponential distribution?

Problem 20.6, *pg. 157*: *Design an algorithm that reads a sequence of packets and maintains a uniform random subset of size k of the read packets when the $n \geq k$-th packet is read.*

Solution 20.6: We store the first k packets. Consequently, we select the n-th packet to add to our subset with probability $\frac{k}{n}$. If we do choose it, we select an element uniformly at random to eject from the subset.

To prove correctness, we use induction on the number of packets that have been read. Specifically, the inductive hypothesis is that all k-sized subsets are equally likely after $n \geq k$ packets have been read.

The number of k-size subsets is $\binom{n}{k}$, implying the probability of any k-size subset should be $\frac{1}{\binom{n}{k}}$.

For the base case, $n = k$, there is exactly one subset of size k which is what the algorithm computes.

Assume the induction hypothesis holds for $n > k$. Consider the $(n+1)$-th packet. The probability of a k-size subset that does not include the $(n+1)$-th packet is the probability that the k-size subset was selected after reading the n-th packet and the $(n+1)$-th packet was not selected. These two events are independent, which means the probability of selecting such a subset is

$$\frac{1}{\binom{n}{k}}\left(1 - \frac{k}{n+1}\right) = \frac{k!(n-k)!}{n!}\left(\frac{n+1-k}{n+1}\right) = \frac{k!(n+1-k)!}{(n+1)!}.$$

This simplifies to $\frac{1}{\binom{n+1}{k}}$, so induction goes for subsets excluding the $n+1$ element.

The probability of a k-size subset H that includes the $(n+1)$-th packet p_{n+1} can be computed as follows. Let G be a k-size subset of the first n packets. The only way we can get from G to H is if G contains $H \setminus \{p_{n+1}\}$. Let G^* be such a subset; let $\{q\} = G \setminus G^*$.

The probability of going from G to H is the probability of selecting p_{n+1} and dropping q, which is equal to $\frac{k}{n+1}\frac{1}{k}$. There exist $-(k-1)$ candidate subsets for G^*, each with probability $\frac{1}{\binom{n}{k}}$ (by the inductive hypothesis) which means that the probability of H is given by

$$\frac{k}{n+1}\frac{1}{k}(n+(k-1))\frac{1}{\binom{n}{k}} = \frac{(n+1-k)(n-k)!k!}{(n+1)n!} = \frac{1}{\binom{n+1}{k}},$$

so induction goes through for subsets including the $(n+1)$-th element. Following is the code in C++:

```
template <typename T>
vector<T> reservoir_sampling(istringstream &sin, const int &k) {
  T x;
  vector<T> R;
  // Store the first k elements
  for (int i = 0; i < k && sin >> x; ++i) {
    R.emplace_back(x);
  }

  // After the first k elements
  int element_num = k + 1;
  while (sin >> x) {
    default_random_engine gen((random_device())());  // random num generator
    // Generate random int in [0, element_num]
    uniform_int_distribution<int> dis(0, element_num++);
    int tar = dis(gen);
    if (tar < k) {
      R[tar] = x;
    }
  }
  return R;
}
```

Problem 20.7, *pg. 157*: *Design an algorithm that computes an array of size k consisting of distinct integers in the set $\{0,1,\ldots,n-1\}$. All subsets should be equally likely and, in addition, all permutations of elements of the array should be equally likely. Your time should be $O(k)$. Your algorithm should use $O(k)$ space in addition to the k element array holding the result. You may assume the existence of a subroutine that returns integers in the set $\{0,1,\ldots,n-1\}$ with uniform probability.*

Solution 20.7: We maintain a hash table H which maps a subset of $\{0,1,\ldots,n-1\}$ to $\{0,1,\ldots,k-1\}$. Initially H is empty. The final result is stored in an array R of length k. We do k iterations of the following. Choose a random integer r in $[i,n-1]$, where i is the current iteration count, starting at 0. If r does not lie in H, we set $R[i]$ to r and

add (r, i) to H. If r does lie in H, say $(r, j) \in H$, we set $R[i] = j$, remove (r, j) from H, and add (r, i) to H.

This approach is correct because it mimics the offline sampling algorithm described in Solution 20.2 on Page 427. We simulate the array used in that algorithm with H; when $k \ll n$, this results in a huge saving in space, since the array is of length n, and most of it is unchanged.

```
vector<int> online_sampling(const int &n, const int &k) {
  unordered_map<int, int> table;
  vector<int> res;
  for (int i = 0; i < k; ++i) {
    default_random_engine gen((random_device())());  // random num generator
    // Generate random int in [i, n - 1]
    uniform_int_distribution<int> dis(i, n - 1);
    int r = dis(gen);
    auto it = table.find(r);
    if (it == table.end()) {  // r is not in table
      res.emplace_back(r);
      table.emplace(r, i);
    } else {  // r is in table
      res.emplace_back(it->second);
      it->second = i;
    }
  }
  return res;
}
```

Problem 20.8, ***pg.157***: *Assuming elections are statistically independent and that the probability of a Republican winning Election i is p_i, how would you compute the probability of a Republican majority?*

Solution 20.8: Number the individual elections from 1 to 435. Let p_n be the probability that the Republican candidate wins Election n. Let $\Pr(r, n)$ be the probability that exactly r Republicans win in elections $\{1, 2, \ldots, n\}$.

Exactly r Republicans win in elections $\{1, 2, \ldots, n\}$ if (1.) r Republicans win in elections $\{1, 2, \ldots, n-1\}$ and the Republican candidate loses election n, or (2.) $r-1$ Republicans win in elections $\{1, 2, \ldots, n-1\}$ and the Republican candidate wins election n.

Since these events are disjoint, $\Pr(r, n)$ is the sum of the probabilities of these two events. To be precise,

$$\Pr(r, n) = \Pr(r-1, n-1)p_n + \Pr(r-1, n)(1-p_n).$$

Therefore P can be computed using DP; the base cases for the recursion are $\Pr(0, 0) = 1$ and $\Pr(r, n) = 0$, for $r > n$.

The probability of a Republican majority is $\sum_{k=\lceil \frac{435}{2} \rceil}^{435} \Pr(k, 435)$. Since both r and n take values from 0 to the total number of elections and computing $\Pr(r, n)$ from

earlier values takes $O(1)$ time, the complexity of computing $\{\Pr(i, j) \mid 0 \leq i \leq j \leq 435\}$ is proportional to the square of the number of elections.

```
// prob is the probability that each Republican wins.
// r is the number of Republicans wins, and n is the number of elections.
double house_majority_helper(const vector<double> &prob, const int &r,
                             const int &n, vector<vector<double>> &P) {
  if (r > n) {
    return 0.0;  // base case: not enough Republicans
  } else if (r == 0 && n == 0) {
    return 1.0;  // base case
  } else if (r < 0) {
    return 0.0;
  }

  if (P[r][n] == -1.0) {
    P[r][n] = house_majority_helper(prob, r - 1, n - 1, P) * prob[n - 1] +
              house_majority_helper(prob, r, n - 1, P) * (1.0 - prob[n - 1]);
  }
  return P[r][n];
}

double house_majority(const vector<double> &prob, const int &n) {
  // Initialize DP table
  vector<vector<double>> P(n + 1, vector<double>(n + 1, -1.0));

  // Accumulate the probabilities of majority cases
  double prob_sum = 0.0;
  for (int r = ceil(0.5 * n); r <= n; ++r) {
    prob_sum += house_majority_helper(prob, r, n, P);
  }
  return prob_sum;
}
```

ϵ-Variant 20.8.1: Compute the probability of a Republican majority given the outcomes of a subset of the races.

ϵ-Variant 20.8.2: Richard and Leopold are playing a series of tennis games in which Richard wins an individual game with probability 0.6. The outcomes of successive games are independent. The first player to win 11 games wins the series. The prize for winning the series is \$100; the winner gets all the prize money. Because of the weather, the series is stopped with Richard leading 7 games to 5. Divide the prize money fairly between Richard and Leopold.

Variant 20.8.3: Consider the following three events: getting one or more sixes when six dice are rolled, getting two or more sixes when 12 dice are rolled, and getting three or more sixes when 18 dice are rolled. Which, if any, of these three events is most probable?

Problem 20.9, *pg. 158*: *You select a coin at random from the bag and toss it five times.*

It comes up heads three times. What is the probability that it was the coin that was biased towards tails? How many times do you need to toss the coin that is biased towards tails before it comes up with a majority of tails with probability greater than $\frac{99}{100}$?

Solution 20.9: Let L be the event that the selected coin is tail-biased, U be the event that the selected coin is head-biased, and $3H5$ be the event that a coin chosen at random from the bag comes up heads 3 times out of 5 tosses.

We want to compute $\Pr(L \mid 3H5)$. By Bayes' rule, this is $\Pr(L \cap 3H5)/\Pr(3H5)$. Applying Bayes' rule again, this probability equals

$$\begin{aligned}
&\frac{\Pr(3H5 \mid L)\Pr(L)}{\Pr\Big(3H5 \cap (L \cup U)\Big)} \\
&= \frac{\Pr(3H5 \mid L)\Pr(L)}{\Pr(3H5 \cap L) + \Pr(3H5 \cap U)} \\
&= \frac{\Pr(3H5 \mid L)\Pr(L)}{\Pr(3H5 \mid L)\Pr(L) + \Pr(3H5 \mid U)\Pr(U)} \\
&= \frac{\binom{5}{3} \times 0.4^3 \times 0.6^2 \times 0.5}{\binom{5}{3} \times 0.4^3 \times 0.6^2 \times 0.5 + \binom{5}{3} \times 0.4^2 \times 0.6^3 \times 0.5} \\
&= 0.4
\end{aligned}$$

For the second part, we can use the Chebyshev inequality to compute the number of trials we need for a majority of n tosses of the tail-biased coin to be heads with probability $\frac{1}{100}$. Let L_i be the event that the i-th toss of the tail-biased coin comes up heads. It will be convenient to use a Bernoulli random variable X_i to encode this event, with a 1 indicating heads and 0 indicating tails.

The mean μ of the sum X of n Bernoulli random variables which are independent and identically distributed (IID) with probability p is $n \times p$; the standard deviation σ is $\sqrt{np(1-p)}$. In our context, $\mu = 0.4n$ and $\sigma = \sqrt{6n/25}$.

The Chebyshev inequality gives us an upper bound on the probability of a random variable being far from its mean. Specifically, $\Pr\Big(|X - \mu| \geq k\sigma\Big) \leq \frac{1}{k^2}$. Note that the event $|X - \mu| \geq k\sigma$ is a superset of the event $X - \mu \geq k\sigma$.

For the majority of n tosses to not be tails, it is necessary that the sum of the n coin tosses is greater than or equal to $0.5n$. To bound this probability by $\frac{1}{100}$ we take $k = 10$ in the Chebyshev inequality. Specifically, we need to solve for n such that $0.5n - 0.4n \geq 10 \times \sqrt{6n/25}$, i.e., $0.1n \geq 10 \times \sqrt{6n/25}$, which is satisfied for $n \geq 2400$.

The Chebyshev inequality holds for all random variables if they have a variance. We can obtain a tighter bound by applying a Chernoff bound, which is specific to the sums of Bernoulli random variables. Specifically, Chernoff bounds tell us that $\Pr(X \geq (1+\delta)\mu) \leq e^{\frac{-\mu\delta^2}{3}}$. We want to bound $\Pr(X \geq 0.5n = (1 + 0.25)(0.4n))$, hence $\delta = 0.25$. Thus we want $e^{\frac{-0.4n(0.25)^2}{3}} < 0.01$; taking natural logs we obtain $-\frac{0.4n(0.25)^2}{3} < -\ln 100 = -4.6$, which holds for $n > 552$.

The Chernoff bound is also pessimistic. Through a simulation, the code for which is attached below, we determined that when $n = 553$, only 17 times in 10^7 trials did

we observe a majority of tails. When $n = 148$, tails was not a majority in 0.88% of the trials.

```
// Return the number of fail trails
int simulate_biased_coin(const int &n, const int &trails) {
  default_random_engine gen((random_device())()); // random num generator
  // Generate random double in [0.0, 1.0]
  uniform_real_distribution<double> dis(0.0, 1.0);
  const double bias = 0.4;
  int fails = 0;
  for (int i = 0; i < trails; ++i) {
    int biased_num = 0;
    for (int j = 0; j < n; ++j) {
      biased_num += (dis(gen) >= bias);
    }

    if (biased_num < (n >> 1)) {
      ++fails;
    }
  }
  return fails;
}
```

Problem 20.10, *pg. 158*: *If m balls are thrown into n bins uniformly randomly and independently, what is the expected number of bins that do not have any balls?*

Solution 20.10: The probability that a given ball does not land in a given bin is $\frac{n-1}{n}$. Since throws are independent, the probability that no ball lands in that bin is $\left(\frac{n-1}{n}\right)^m$. Hence the expected number of empty bins is $n\left(\frac{n-1}{n}\right)^m$. This is closely approximated by $n \times e^{-m/n}$. Hence if on an average, each server is handling significantly more than one client, there should be very few idle servers. If $n = m$, then the expected number of empty bins tends to $1/e$ times the total number of bins, which is a classical result.

We used the linearity of expectation in an essential way. Linearity of expectation does not require the individual random variables to be independent. This is crucial, since bins are not independent, e.g., it is impossible for all bins to be empty.

Variant 20.10.1: David is the first passenger to board a flight. He has lost his boarding card, and selects a seat to sit in uniformly randomly. Successive passengers either sit in their assigned seat, or, if someone is already in their seat, select another seat uniformly randomly from the set of remaining empty seats. Selections are done independently. Henri is the first to board a different flight. He has also lost his boarding card, and his flight fills up the same way as David's flight. There are 100 seats on David's flight, and 200 seats on Henri's flight. Both flights are full. Let L_A and L_B be the last passengers to board David's flight and Henri's flight, respectively. Which of L_A and L_B are more likely to get their assigned seat?

Variant 20.10.2: How many people need to be at a party before the probability of two people at the party having a common birthday exceeds 0.5? How many people

need to be at a party before the probability of one of them having your birthday exceeds 0.5? (Assume birthdays are uniformly independently distributed across 365 days of the year; nobody is born on February 29.)

Problem 20.11, ***pg.158***: *What is the expected number of fixed points of a uniformly random permutation $\sigma : \{0, 1, \ldots, n-1\} \mapsto \{0, 1, \ldots, n-1\}$, i.e., the expected cardinality of $\{i \mid \sigma(i) = i\}$? What is the expected length of the longest increasing sequence starting at $\sigma(0)$, i.e., if k is the first index such that $\sigma(k) < \sigma(k-1)$, what is the expected value of k?*

Solution 20.11: Let X_i be the random variable, which is 1 if $\sigma(i) = i$ and 0 otherwise. Such a random variable is often referred to as an "indicator random variable". The number of fixed points is equal to $X_0 + X_1 + \cdots + X_{n-1}$. Expectation is linear, i.e., the expected value of a sum of random variables is equal to the sum of the expected values of the individual random variables. The expected value of X_i is $0 \times \frac{n-1}{n} + 1 \times \frac{1}{n}$ (since an element is equally likely to be mapped to any other element). Therefore the expected number of fixed points is $n \times \frac{1}{n} = 1$.

We can compute the expected value of k by defining indicator random variables $Y_0, Y_1, \ldots Y_{n-1}$, where $Y_i = 1$ iff for all $j < i$ we have $\sigma(j) < \sigma(i)$. Observe that k is simply the sum of the Y_is. The expected value of Y_i is $\frac{1}{i+1}$, since for all $j < i$ we have $\sigma(j) < \sigma(i)$ iff the largest of the first $i+1$ elements is at position i, which has probability $\frac{1}{i+1}$ since all the permutations are equally likely. Therefore the expected value for k is $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$, which tends to $\log_e n$.

For both parts of the problem, we used the linearity of expectation which does not require the individual random variables to be independent. This property of expectation is crucial since the X_is and the Y_js are not independent—for example, if the first $n-1$ elements get mapped to themselves, then the n-th element must also map to itself.

Problem 20.12, ***pg.158***: *Gottfried repeatedly rolls an unbiased six-sided die. He stops when he has rolled all the six numbers on the die. How many rolls will it take, on an average, for Gottfried to see all the six numbers?*

Solution 20.12: First we prove that if $\langle X_0, X_1, \ldots \rangle$ is a sequence of Bernoulli IID random variables, with $p(X_i = 1) = p$, then the expected time to see the first 1 is $\frac{1}{p}$. The reasoning is as follows. Define F_i to be the event that the first 1 comes on the i-th trial. Then $\Pr(F_i) = (1-p)^{i-1}p$. Hence the expected time is $S = \sum_{i=1}^{\infty} i(1-p)^{i-1}p$. This sum simplifies to $\frac{1}{p}$ (multiply both sides by p, subtract, and sum the infinite geometric series on the right).

Now, we consider the problem of die rolls. The key is to determine the expected time to see the k-th new value. Clearly, the expected time to see the first new value is just 1. The time to see the second new value from the first new value is $\frac{1}{5/6}$ since the probability of seeing a new value, given that one value has already been seen, is $\frac{5}{6}$. In this way, the time taken to see the third new value, given that two values have already been seen, is $\frac{1}{4/6}$. Generalizing this idea, the time taken to see the k-th new value, given that $k-1$ values have already been seen, is $\frac{1}{(6-(k-1))/6}$. Hence the expected

time to see the sixth new value is $\frac{6}{6} + \frac{6}{5} + \frac{6}{4} + \frac{6}{3} + \frac{6}{2} + \frac{6}{1} \approx 14.7$.

Variant 20.12.1: On average, how many cards on average have to be dealt from a well-shuffled deck before an ace appears?

Problem 20.13, *pg. 159*: *What is the probability that these three segments can be assembled into a triangle?*

Solution 20.13: The first thing to note is that three segments can make a triangle iff no one segment is longer than the sum of the other two. The "only if" follows from the triangle inequality and the "if" follows from a construction—take a segment and draw circles at the endpoints with radius equal to the lengths of the other circles.

For the case we are considering the three segment lengths add up to 1. Therefore, there exists a segment that is longer than the sum of the other two iff there exists a segment that is longer than $\frac{1}{2}$.

Let $l = \min(u1, u2)$, $m = \max(u1, u2) - \min(u1, u2)$, and $u = 1 - \max(u1, u2)$; these are the lengths of the first, second, and third segments, from left-to-right. If one segment is longer than 0.5, then none of the others can be longer than 0.5, implying the events $l > 0.5$, $m > 0.5$, and $u > 0.5$ are disjoint.

Observe that $l > 0.5$ iff both $u1$ and $u2$ are greater than 0.5; the probability of this event is $\frac{1}{2} \times \frac{1}{2}$ because $u1$ and $u2$ are chosen independently. Similarly $u > 0.5$ iff both $u1$ and $u2$ are less than 0.5, which has probability $\frac{1}{2} \times \frac{1}{2}$.

To compute the probability of $m > 0.5$, first we consider the case that $u1 < u2$. For $m > 0.5$, we need $u1$ to be in $[0, 0.5]$ and $u2$ to be in $[0.5 + u1, 1]$. This probability can be expressed by the integral

$$\int_{u1=0}^{0.5} \int_{u2=u1+0.5}^{1} 1 \times du1 \times du2$$

which evaluates to $\frac{1}{8}$.

By symmetry, the probability of $m > 0.5$ when $u1 > u2$ is also $\frac{1}{8}$. Hence the probability of a segment being longer than $\frac{1}{2}$ is $\frac{1}{4} + \frac{1}{4} + \frac{1}{4} = \frac{3}{4}$. Therefore the probability of being able to make a triangle out of the segments is $1 - \frac{3}{4} = \frac{1}{4}$.

***e*-Variant 20.13.1**: If a stick is broken into two pieces uniformly randomly, what is the expected length of the longer piece? What is the expected value of the ratio of the length of the longer piece to the length of the smaller piece?

***e*-Variant 20.13.2**: If a stick is broken into three pieces uniformly randomly, what is the expected length of the longest, shortest and middle-sized pieces?

Problem 20.14, *pg. 159*: *Solve Problem 20.13 on Page 159 when $u1$ is uniformly randomly in $[0, 1]$ and $u2$ is subsequently chosen uniformly randomly in $[u1, 1]$. Can you determine which of these two approaches is more likely to produce a triangle without computing the exact probabilities?*

Solution 20.14: We fail to make a triangle in case $u1 > 0.5$, $u2 - u1 > 0.5$, or $1 - u2 > 0.5$. The first probability is simply $\frac{1}{2}$.

The second probability is given by the integral

$$\int_{u1=0}^{0.5} \int_{u2=u1+0.5}^{1} \frac{1}{(1-u1)} \times du2 \times du1.$$

Note that the probability density function for $u2$ is different from the previous case since $u2$ is uniform in $[u1, 1]$, not $[0, 1]$. This integral evaluates to $\frac{1+\log_e \frac{1}{2}}{2}$. The third probability can also be computed using an integral but by symmetry, it must be the same as the second probability. Hence the final probability is $\frac{1}{2} + 2 \times \frac{1+\log_e \frac{1}{2}}{2} \approx 0.807$.

Intuitively, the second formulation leads to a higher probability of a long line segment (which implies that we cannot form a triangle) because there is less diversity in the points. For the first case, the points are spread randomly; for the second, there is a 0.5 probability that the first point itself precludes us from building the triangle. Another way to think of it is that if we put down many points, the first method will lead to short segments with little variation in lengths but the second method will give us a skewed distribution and the first few segments will be considerably longer.

Problem 20.15, ***pg. 159***: *Design a strategy that selects the best secretary with a probability greater than 0.25, regardless of n.*

Solution 20.15: The only reasonable interpretation of random is that all permutations are equally likely. Therefore, if we always select the first secretary, we have a $\frac{1}{n}$ chance of selecting the best secretary.

One way to do better is to skip the first $\lfloor \frac{n}{2} \rfloor$ secretaries and then choose the first one in the remaining set that is superior to the best secretary interviewed in the first $\lfloor \frac{n}{2} \rfloor$ secretaries. Roughly, the probability of selecting the best secretary with this strategy is at least $\frac{1}{4}$ since the probability that the second best secretary lies in the first set and the best secretary is in the second set is at least $\frac{1}{4}$.

The probability is actually greater than $\frac{1}{4}$ since if the second best secretary is in the first set, there is a higher than 0.5 probability that the best secretary is in the second set. When n is even, say $n = 2m$, the probability that the strategy selects the best secretary is at least $\frac{m}{2m} \frac{m+1}{2m} > \frac{1}{4}$. When n is odd, say $n = 2m+1$, the probability that the strategy selects the best secretary is at least $\frac{m}{2m+1} \frac{m+1}{2m} > \frac{1}{4}$. Note that even if the second best secretary is not in the first set, our strategy may still select the best secretary, e.g., if the best secretary is the first to appear.

It is known that if we follow a strategy of skipping the first s secretaries and selecting the first secretary who is superior to all others so far, the probability of selecting the best secretary is maximized for s closest to $\frac{n}{e}$, and this probability tends to $\frac{1}{e}$.

Problem 20.16, ***pg. 159***: *What is the value of w such that Once-or-Twice is a fair game, i.e., for a rational player, the expected gain is 0?*

Solution 20.16: If the probability of winning is p, then the expected gain is $-1+p\times w$. Hence for a fair game, $w=\frac{1}{p}$.

The face value of the card can be any number between 1 and 13. For the dealer, all values are equally likely. Hence if the player's card has a face value i, then the probability of winning for the player is $\frac{i-1}{13}$. If the player always takes only one random card, his probability of winning is $\frac{1}{13}\sum_{i=1}^{13}\frac{i-1}{13}=\frac{6}{13}$. Hence it makes sense to ask for the next card only if the first card yields a probability less than $\frac{6}{13}$, i.e., the face value of the first card is less than 7. If the face value of the first card is 7 or more, then the probability of winning is $\frac{1}{7}\sum_{i=7}^{13}\frac{i-1}{13}=\frac{9}{13}$; otherwise, we pick again, in which case the probability of winning is $\frac{6}{13}$. Hence the overall probability of winning is $\frac{7}{13}\frac{9}{13}+\frac{6}{13}\frac{6}{13}=\frac{99}{169}$. Thus the fair value is $\frac{169}{99}\approx 1.707$.

Problem 20.17, ***pg. 159***: *Suppose you are playing the multibet card color game and are restricted to bet in penny increments. Compute a tight lower bound on the amount that you can guarantee to win under this restriction.*

Solution 20.17: A good way to begin is to devise a strategy that guarantees a positive return. It is possible to guarantee a 2× return by waiting till the last card and betting the entire amount on the last card whose color is uniquely determined by the 51 cards that have already been seen.

To do better than a 2× return, consider the case of a deck of 4 cards with 2 red cards and 2 black cards. If we do not bet on the first card, there will be three remaining cards. Assume, without loss of generality, that two cards are black and one is red. If we bet $\$\frac{1}{3}$ on the next card being black and are successful, then we have $\$\frac{4}{3}$ which we can double on the last card for a $\frac{8}{3}>2$ return. If we lose, then the two remaining cards are black, in which case we can double our remaining money twice, i.e., achieve a $\frac{2}{3}\times 2\times 2=\frac{8}{3}>2$ return. Note that this analysis assumes we can bet arbitrary fractions of the money we possess.

Now, we consider the case where we can only bet in penny increments. Let $Q(c,r,t)$ be the maximum we can guarantee, when we have c cents to gamble with and there are r red cards remaining out of a total of t cards. We can bet b cents, for $0\leq b\leq c$ on the next card. Since we have to design a strategy that maximizes the worst-case payoff, the maximum amount we can make on betting on red cards is given by

$$Q_R(c,r,t) = \max_{0\leq b\leq c}\Big(\min\Big(Q(c+b,r-1,t-1),Q(c-b,r,t-1)\Big)\Big).$$

The maximum we can make by betting on black cards is

$$Q_B(c,r,t) = \max_{0\leq b\leq c}\Big(\min\Big(Q(c+b,r,t-1),Q(c-b,r-1,t-1)\Big)\Big).$$

Hence $Q(c,r,t)=\max\Big(Q_R(c,r.t),Q_B(c,r,t)\Big)$ which yields a DP algorithm for computing the maximum payoff—base cases are of the form $Q(c,0,t)$ and $Q(c,t,t)$, both of which are $c\times 2^t$.

However if we directly try and compute $Q(100, 26, 52)$, the algorithm runs for an unacceptably long time. This is because we will be exploring paths for which c grows very large. Since we are given the maximum payoff on a dollar when fractional amounts can be bet is less than 9.09, we can prune computations for $Q(c, r, t)$ when $c \geq 909$. The following code implements the DP algorithm with this pruning; it computes the maximum payoff, 808, in two minutes.

```
double compute_best_payoff_helper(
    unordered_map<int,
                  unordered_map<int, unordered_map<int, double>>> &cache,
    const double &upper_bound, const int &cash, const int &num_red,
    const int &num_cards) {
  if (cash >= upper_bound) {
    return cash;
  }

  if (num_red == num_cards || num_red == 0) {
    return cash * pow(2, num_cards);
  }

  if (cache[cash][num_red].find(num_cards) == cache[cash][num_red].end()) {
    double best = numeric_limits<double>::min();
    for (int bet = 0; bet <= cash; ++bet) {
      double red_lower_bound = min(
        compute_best_payoff_helper(cache, upper_bound, cash + bet,
                                   num_red - 1, num_cards - 1),
        compute_best_payoff_helper(cache, upper_bound, cash - bet,
                                   num_red, num_cards - 1));

      double black_lower_bound = min(
        compute_best_payoff_helper(cache, upper_bound, cash - bet,
                                   num_red - 1, num_cards - 1),
        compute_best_payoff_helper(cache, upper_bound, cash + bet,
                                   num_red, num_cards - 1));
      best = max(best, max(red_lower_bound, black_lower_bound));
    }
    cache[cash][num_red][num_cards] = best;
  }
  return cache[cash][num_red][num_cards];
}

double compute_best_payoff(const int &cash) {
  double upper_bound = 9.09 * cash;
  unordered_map<int, unordered_map<int, unordered_map<int, double>>> cache;
  return compute_best_payoff_helper(cache, upper_bound, cash, 26, 52);
}
```

Here is a sketch of the proof that $2^{52}/\binom{52}{26}$ is the maximum amount that you can guarantee you will win when arbitrary fractions of the stake can be bet.

Let $M(r, b)$ be the maximum amount you can guarantee you will win starting with \$1 when there are r red cards and b black cards remaining, under the assumption that you can bet any fraction of your current stake. Let $f(r, b) \in [-1, 1]$ be the optimum amount to bet on red; $f(r, b)$ could be negative, which is equivalent to betting a

positive amount on black. Then M and f satisfy the following:

$$M(r,b) = \max_{f(r,b)\in[-1,1]}(\min((1+f(r,b))M(r-1,b),(1-f(r,b))M(r,b-1))).$$

Elementary algebra can be used to show that the value for $f(r,b)$ that maximizes $M(r,b)$ is the one in which $(1+f(r,b))M(r-1,b) = (1-f(r,b))M(r,b-1)$; both of these terms equal $M(r,b)$. The optimum value for $f(r,b)$ is $\frac{r-b}{r+b}$. Substituting for $f(r,b)$ in leads to the following recurrence:

$$M(r,b) = \frac{2M(r-1,b)M(r,b-1)}{M(r-1,b)+M(r,b-1)}.$$

The base cases are $M(r,0)$ and $M(0,b)$, which are 2^r and 2^b, respectively. Induction can be used to prove that $M(r,b) = 2^{r+b}/\binom{r+b}{b}$. Substituting $r = b = 26$ yields the desired result.

Problem 20.18, ***pg. 160***: *Design a strategy that maximizes the probability of winning at the one red card game.*

Solution 20.18: We can trivially achieve a probability of success of $\frac{1}{2}$ by always choosing the first card.

A natural way to proceed is to consider the probability $p_k(f)$ of winning for the optimum strategy after k cards remain, of which f are red cards. Then $p_k(f) = \max\left(\frac{f}{k}, \frac{f}{k}p_{k-1}(f-1) + (1-\frac{f}{k})p_{k-1}(f)\right)$.

The base cases for the recurrence are $p_1(1) = 1$ and $p_1(0) = 0$. Applying the recurrence, we obtain $p_2(2) = 1, p_2(1) = \frac{1}{2}, p_2(0) = 0$, and $p_3(3) = 1, p_3(2) = \frac{2}{3}, p_3(1) = \frac{1}{3}, p_3(0) = 0$. This suggests that $p_k(f) = \frac{f}{k}$, which can directly be verified from the recurrence. Therefore the best we can do, $p_{52}(26) = \frac{26}{52} = \frac{1}{2}$, is no better than simply selecting the first card.

An alternate view of this is that since the cards in the deck are randomly ordered, the probability of the topmost card being red is the same as that of the card at the bottom of the deck being red. The bottom-most card has a $\frac{f}{k}$ probability of being red when there are f red cards and k cards in total.

Variant 20.18.1: An alchemist has discovered several dryads—stones which can change lead into gold. The rate at which a dryad converts lead into gold is proportional to its weight. The alchemist also has tryads, stones which, by themselves, do nothing. However, when a tryad is heated together with a dryad, the two stones combine to form a new stone, which could be either a tryad or a dryad. The weight is conserved. The probability of the combined stone being a dryad is the weight of the initial dryad divided by the sum of the weights of the initial stones. He can also combine tryads with tryads and dryads with dryads, which results in the same stone with the sum of the constituent weights.

The alchemist wants to combine tryads with dryads to maximize the expected dryad weight. He could follow many strategies, such as combining the heaviest

dryad with the lightest tryad, or combining the dryad and tryad that are closest in weight. What strategy should he follow?

Variant 20.18.2: Isaac and Leonhard are playing a card game against each other. Isaac starts with \$2 and Leonhard starts with \$1. They bet \$1 on each game. They stop playing when one of the two runs out of money; the other is the overall winner. Suppose the probability of Leonhard winning any single game is $\frac{2}{3}$. What is the probability that he is the overall winner?

Problem 20.19, *pg. 160*: *Consider an auction for an item in which the reserve price is a random variable X uniformly distributed in $[0, 400]$. You can bid B. If your bid is greater than or equal to the reserve price, you win the auction and have to pay B. You can then sell the item for an 80% markup over the reserve price. How much should you offer for the item?*

Solution 20.19: The first question to ask is what are you trying to optimize? The objective could be to maximize expected profit, minimize loss, or maximize ratio of expected profit to variance.

Let's say we want to maximize expected profit. Let X be the random variable corresponding to the reserve price. We win the auction if the reserve price is less than or equal to our bid. The selling price is $1.8\times$ the reserve price, and our cost to buy is fixed at B. Therefore, the expected profit is $\int_{X=0}^{X=B} \frac{1}{400}(1.8X - B)dX$. This simplifies to $\frac{0.9\times B^2 - B^2}{400}$, which is negative for all $B > 0$, i.e., we should not place a bid.

In retrospect, this result is obvious since if we win the auction, we are paying twice of X in expectation and getting only $1.8X$ in return.

Problem 20.20, *pg. 160*: *Your friend at the Acme Casino has rigged their roulette wheel to make the probability of the ball landing on red $\frac{19}{37}$. You can bet on the same color exactly 100 times; after that the casino management will be alerted. You start with \$1. On each round, you can bet any amount from 0 to your entire bankroll. What should your strategy be?*

Solution 20.20: The key to solving this problem is determining what the objective is. The "obvious" criterion is expectation maximization. It is simple to see that the strategy that maximizes expectation is betting the entire bankroll on red each time. The expected payoff is $2^{100} \times \frac{19}{37}^{100} \approx 3\times 10^{16}$; however, the probability that the strategy does not result in a "bust" (the bankroll going to 0) is $\frac{19}{37}^{100} \approx 2.39 \times 10^{-14}$.

We can avoid busting by maximizing the expectation of the logarithm of the final bankroll—since $\log 0 = -\infty$, any strategy which maximizes this objective will never bet the whole amount on any one bet. Let p be the odds of winning, and $q = 1 - p$ the odds of losing. If we bet r fraction of the current bankroll B, the expectation of the logarithm of the resulting bankroll is $p \log(B + 2rB) + (1 - p) \log(B - rB)$. Simple calculus shows that the optimum choice of $r = p - q = 2p - 1$.

The above analysis was first done by John Larry Kelly, a Ph.D. from The University of Texas at Austin, who went on to work at Bell Labs. The best reference is his original paper, "*A New Interpretation of Information Rate*". William Poundstone's "*Fortune's*

Formula: The Untold Story of the Scientific Betting System That Beat the Casinos and Wall Street" is a highly readable account of Kelly's result and its impact

Variant 20.20.1: Find the optimum betting strategy when there are k outcomes, each with probability p_i and return r_i.

Variant 20.20.2: You can choose among k investment types. The return on the i-th investment type has a normal distribution with mean r_i and standard deviation ${\sigma_i}^2$. The covariance between investment types i and j is σ_{ij}. Compute a portfolio, i.e., an allocation of an investment across these k investment types that maximizes the expected return, subject to the constraint that the aggregate investment has a variance that is less than a specified constant. How does your approach handle short positions, i.e., selling an investment type that you do not own? How would you handle diversification constraints, e.g., a requirement of the form "no more than 20% of the portfolio can be in any four investment types"? How would you do the same if there is uncertainty in some of the parameters?

Variant 20.20.3: Design a strategy by which you can go to a casino with \$100, play an unrigged Roulette wheel, and leave with more than \$100 with probability greater than 0.98. The casino only accepts positive integer-valued bets.

Variant 20.20.4: Suppose you can bet a dollar on the ball falling into a pocked numbered 1 to 36. If the ball falls in the pocket you selected, you receive \$36 in return; otherwise, you lose your bet. The casino offers you insurance against losing. Specifically, for \$20 you can insure 36 one dollar bets. If you are behind after 36 bets, the casino will give you \$40, otherwise it keeps the premium. Should you buy the insurance?

Problem 20.21, *pg. 160*: *Prove that an algorithm in which the choice of the next variable to read in an L_K expression is a deterministic function of the values read up to that point must, in the worst case, read all variables to evaluate the expression. Design a randomized algorithm that reads fewer variables on an average, independent of the values assigned to the variables.*

Solution 20.21: First, we show that any deterministic algorithm must examine all the Boolean variables. The idea is that an adversary can force the value of any subexpression to be unknown till all the variables in that subexpression have been read. For example, suppose variable X is ANDed with variable Y. If the algorithm reads the value of X before Y, we return true; when Y is queried, we return false. In this way, the value of $X \wedge Y$ is determined only after both the variables are read.

This generalizes with induction: the inductive hypothesis is that an L_k expression requires all the variables to have been read before its value is determined and its final value is the value of the last variable read. For a subexpression of the form $\phi \wedge \psi$, where ϕ and ψ are L_k expressions, if all the variables from ϕ are read before

all the variables from ψ are read, the adversary chooses the last variable read from ϕ to be true, forcing the algorithm to evaluate ψ. A similar argument can be used for subexpressions of the form $\phi \vee \psi$.

Suppose we evaluate an expression by choosing one of its two subexpressions at random to evaluate first; we evaluate the other subexpression only if the expression's value is not forced by the subexpression that we evaluated first.

For example, if we are to evaluate an L_{k+1} expression of the form $((\phi_0 \wedge \phi_1) \vee (\psi_0 \wedge \psi_1))$, where the subexpressions $\phi_0, \phi_1, \psi_0, \psi_1$ are L_k expressions, we randomly choose one of $(\phi_0 \wedge \phi_1)$ and $(\psi_0 \wedge \psi_1)$ to evaluate first. If the first expression evaluated is true, we can ignore the second; otherwise, we evaluate the second. If the first expression is true, we reduced the number of variables queried by at least half. If the first expression is false, at least one of the two subexpressions is false and we have a probability of 0.5 of selecting that subexpression and avoiding evaluating the other subexpression. So, in the worst-case, we can expect to avoid one of the four subexpressions $\phi_0, \phi_1, \psi_0, \psi_1$. Therefore the expected number of variables queried to evaluate an L_{k+1} expression, $Q(k+1)$ satisfies

$$Q(k+1) \leq 3Q(k).$$

From this, $Q(k) = 3^k$. It is straightforward to use induction to show that an L_k expression contains $n = 4^k$ variables, so $Q(k) = n^{\log_4 3} = n^{0.793}$.

Problem 20.22, *pg. 161*: *For what option price is there no opportunity for arbitrage?*

Solution 20.22: Let f be the price for the option. A fair price is determined by the no-arbitrage requirement. Suppose we start with a portfolio of x shares and y options in S—x and y may be negative (which indicates that we sell stocks or sell options).

The initial value of our portfolio is $100x + yf$. On Day 100, two things may have happened:

- The stock went up and the portfolio is worth $120x + 20y$.
- The stock went down and the portfolio is worth $70x$.

If we could choose x and y in such a way that our initial portfolio has a negative value—which means that we are paid to take it on—and regardless of the movement in the stock, our portfolio takes a nonnegative value, then we will have created an arbitrage.

Therefore the conditions for an arbitrage to exist are:

$$\begin{aligned} 120x + 20y &\geq 0 \\ 70x &\geq 0 \\ 100x + yf &< 0 \end{aligned}$$

A fair price for the option is one in which no arbitrage exists. If f is less than 0, an arbitrage exists—we are paid to buy options, lose nothing if the price goes down, and make \$20 per option if the price goes up. Therefore $f \geq 0$, so we can write the third

inequality as $y < -\frac{100}{f}x$. The first equation can be rewritten as $y \geq -6x$. Combining these two inequalities, we see that an arbitrage exists if $-\frac{100}{f} < -6$, i.e., $f > \frac{100}{6}$.

In summary, there is no arbitrage for $f \in [0, \frac{100}{6}]$; there is an arbitrage for all other values of f.

For example, if $f = 19 > \frac{100}{6}$, then the option is overpriced and we should sell ("write") options. If we write b options and buy one share, we will start with a portfolio that is worth $100 + 19b$. If the stock goes down, the options are worthless and our portfolio is worth \$70. If the stock goes up, we lose \$20 on each option we wrote but see a gain on the stock we bought. We want the net gain to be nonnegative and the initial portfolio to have a negative value, i.e.,

$$\begin{aligned} 120 + 20b &\geq 0 \\ 100 + 19b &< 0 \end{aligned}$$

Combining the two inequalities, we see that any value of b in $[-6, -\frac{100}{19})$ leads to an arbitrage.

Problem 20.23, ***pg. 161***: *Consider the same problem as Problem 20.22 on Page 161, with the existence of a third asset class, namely a bond. A \$1 bond pays \$1.02 in 100 days. You can borrow money at this rate or lend it at this rate. Show there is a unique arbitrage-free price for the option and compute this price.*

Solution 20.23: Suppose our initial portfolio consists of x_0 stocks, x_1 options, and x_2 bonds.

Proceeding as above, we see the condition for an arbitrage to exist is:

$$\begin{aligned} 100x_0 + fx_1 + x_2 &< 0 \\ 120x_0 + 20x_1 + 1.02x_2 &\geq 0 \\ 70x_0 + 1.02x_2 &\geq 0 \end{aligned}$$

Writing the linear terms as Ax, if $det(A) \neq 0$, then there always exists an arbitrage since we can solve $Ax = b$, and when $b^T = (-1, 1, 1)$ there is an arbitrage.

The determinant of A equals $70(1.02f - 20) + 1.02(100 \times 20 - 120f)$. This equals 0 when $f = 640/51 \approx 12.549 = f^*$, so an arbitrage definitely exists if the option price is not equal to f^*.

Conversely, if the option is priced at f^*, $det(A) = 0$ and in particular $A_0 = 0.6275A_1 + 0.3583A_2$, where A_i denotes the i-th row of A. Since A_0 is a linear combination of A_1 and A_2 with positive weights, if $A_1x \geq 0$ and $A_2x \geq 0$, then $A_0x \geq 0$, so no arbitrage can exist.

Problem 20.24, ***pg. 162***: *Suppose the price of Jingle stock 100 days in the future is a normal random variable with mean \$300 and standard deviation \$20. What would be the fair price of an option to buy a single share of Jingle at \$300 in 100 days? (Ignore the effect of interest rates.)*

Solution 20.24: Let x be the price of the stock on Day 100. The option is worthless if $x < 300$. If the price is $x \geq 300$, the option is worth $x - 300$ dollars. The expected value of x is given by the integral

$$\int_{300}^{\infty} (x - 300) \times \frac{e^{-\frac{(x-300)^2}{2\times(20)^2}}}{\sqrt{2\pi(20)^2}} dx.$$

The integral can be evaluated in closed form—let $y = x - 300$ and let's write σ instead of 20. The expression above simplifies to

$$\int_{0}^{\infty} y \times \frac{e^{-\frac{y^2}{2\times\sigma^2}}}{\sqrt{2\pi\sigma^2}} dy.$$

The indefinite integral $\int w \times e^{-w^2} dw$ has the closed form solution $-\frac{e^{-w^2}}{2}$, which implies the definite integral equals $\sigma\sqrt{\frac{1}{2\pi}} \approx 0.39\sigma$. Therefore the expected payoff on the option on Day 100 is $0.39 \times 20 = \$7.8$.

Problem 21.1, *pg. 163*: *Which of the* 500 *doors are open after the* 500*-th person has walked through?*

Solution 21.1: As described on Page 38, analyzing a few small examples suggests that, independent of n, door k will be open iff k is a perfect square. This can be rigorously proved as follows.

Proof:

If the number of times a door's state changes is odd, it will be open; otherwise it is closed. Therefore the number of times door k's state changes equals the number of divisors of k. From the small example analysis, we are led to the conjecture that the number of divisors of k is odd iff k is a perfect square. Note that if d divides k, then k/d also divides k. Therefore we can uniquely pair off divisors of k, other than $\sqrt{k}$ (if it is an integer). Hence, when $\sqrt{k}$ is not an integer, k has an even number of divisors. When $\sqrt{k}$ is an integer, it is the only divisor of k that cannot be uniquely paired off with another divisor, implying k has an odd number of divisors. By definition, $\sqrt{k}$ is an integer iff k is a perfect square, proving the result.

This check can be performed by squaring $\lfloor \sqrt{i} \rfloor$ and comparing the result with i.

```
bool is_door_open(const int &i) {
  double sqrt_i = sqrt(i);
  int floor_sqrt_i = floor(sqrt_i);
  return floor_sqrt_i * floor_sqrt_i == i;
}
```

Variant 21.1.1: There are 25 people seated at a round table. Each person has two cards. Each card has a number from 1 to 25. Each number appears on exactly two cards. Each person passes the card with the smaller number to the person on his left.

This is done iteratively in a synchronized fashion. Show that eventually someone will have two cards with identical numbers.

Problem 21.2, ***pg. 163***: *What is the minimum number of five man time-trials needed to determine the top three cyclists from a set of* 25 *cyclists?*

Solution 21.2: Let's start with five time-trials with no cyclist being in more than one of these five initial time-trials. Let the rankings be $\langle A1, A2, A3, A4, A5\rangle$, $\langle B1, B2, B3, B4, B5\rangle$, $\langle C1, C2, C3, C4, C5\rangle$, $\langle D1, D2, D3, D4, D5\rangle$, and $\langle E1, E2, E3, E4, E5\rangle$, where the first cyclist in each sequence is the fastest. Note that we can eliminate $A4, A5, B4, B5, C4, C5, D4, D5, E4$, and $E5$ at this stage.

Now, we race the winners from each of the initial time-trials. Without loss of generality, assume the outcome is $\langle A1, B1, C1, D1, E1\rangle$. At this point, we can eliminate $D1$ and $E1$ as well as $D2$ and $D3$ and $E2$ and $E3$. Furthermore, since $C1$ was third, $C2$ and $C3$ cannot be in the top three; Similarly, $B3$ cannot be a contender.

We need to find the best and the second best from $A2, A3, B1, B2$, and $C1$, which we can determine with one more time-trial. Therefore seven time-trials are enough.

Note that we need six time-trials to determine the overall winner, and the sequence of time-trials to determine the winner is essentially unique—if some cyclists did not participate in the first five time-trials, he would have to participate in the sixth one. But then one of the winners of the first five time-trials would not participate in the sixth time-trial and he might be the overall winner. The first six time-trials do not determine the second and the third fastest cyclists, hence a seventh race is necessary.

Problem 21.3, ***pg. 164***: *Prove that there exists a place such that Albert is at that place at the same time on Sunday as he was on Saturday.*

Solution 21.3: The easiest way to prove this is to imagine another hiker (call him Max) descending the mountain on Saturday, in exactly the same fashion as Albert did on Sunday. When ascending on Saturday, Albert will pass Max at some time and place—this is the time and place which Albert will be at on Sunday.

Problem 21.4, ***pg. 164***: *How would you break a* 4×4 *bar into* 16 *pieces using as few breaks as possible?*

Solution 21.4: If the assumption is that once you have broken the bar into two pieces, they become separate problems, then it does not matter what order you do it—you will require 15 total breaks in *any* scenario, since each break increases the number of pieces by 1.

If, on the other hand, the assumption is that the whole bar stays together (as it would if you were breaking it in its wrapper, for instance), then you can do a little better. You could simply break it along all axes (say, first the vertical and then the horizontal) for a total of six breaks.

Problem 21.5, ***pg. 164***: *If you want to ensure you do not lose, would you rather be F or S?*

Solution 21.5: Number the coins from 1 to 16. Player F can choose all the even-numbered coins by first picking Coin 16 and then always picking the coin at an even index at one of the two ends. For example, if Player S chooses Coin 1, then in the next turn, Player F chooses Coin 2. If Player S chooses Coin 15, then F chooses Coin 14 in the next turn. In this fashion, F can always leave an arrangement where S can only choose from odd-numbered coins.

If the value of the coins at even indices is greater than that of the coins at odd indices, F can win by selecting the even indices and vice versa. If the values are the same, he can simply choose either and in each case, he cannot lose. For the configuration in Figure 4.6 on Page 44, the sum of the even-numbered coins is 125¢ and the sum of the odd-numbered coins is 63¢, so F can win by picking up the even-numbered coins.

The code below implements this idea. It entails a single pass through an array, and its time complexity is $O(n)$, where n is the number of coins (assumed to be even). In contrast, the DP algorithm presented in Solution 15.18 on Page 357 has time complexity $O(n^2)$. The benefit of the DP algorithm is that it yields the optimum solution—140¢ instead of 125¢ for this example.

```
// Return 0 means choosing F, and return 1 means choosing S
template <typename CoinType>
int pick_up_coins(const vector<CoinType> &C) {
  int even_sum = 0, odd_sum = 0;
  for (int i = 0; i < C.size(); ++i) {
    if (i & 1) {  // odd
      odd_sum += C[i];
    } else {  // even
      even_sum += C[i];
    }
  }
  return even_sum >= odd_sum ? 0 : 1;
}
```

Problem 21.6, *pg. 164*: *Assuming the players have infinite computational resources at their disposal, who will win $n \times n$ chomp?*

Solution 21.6: The first player can always win. The key observation in this game is that we want to force the play to be symmetrical around the diagonal, i.e., $(0,0),(1,1),\ldots,(n-1,n-1)$ with our opponent forced to be first to break the symmetry. If that is the case, we can follow each of his moves by a matching move reflected in this diagonal which will eventually force him to select the $(0,0)$ space.

The way to force this kind of play is to be the first person to select $(1,1)$—this causes the play area to be just the column $(0,[0,n-1])$ and the row $([0,n-1],0)$ (i.e., an "L" shape). At that point, the first player can successfully mirror any move that the second player makes, forcing the second player to eventually choose $(0,0)$. This strategy is formalized in pseudocode below.

```
WinChomp():
  You choose square (1,1);
```

```
3  Until you win:
4    Wait for your opponent to choose square (i,j);
5    You choose square (j,i);
```

Problem 21.7, *pg. 165*: *Solve Problem 21.6 on Page 164 if the rectangle is n long along the x-axis, and two long along the y-axis.*

Solution 21.7: Suppose the set of remaining squares are of the form of a rectangle and one additional square (which must be on the lower row) and the second player is to move. The remaining set of squares will be in the form of a rectangle (if the second player plays the lower row) or a rectangle with a set of additional squares on the lower row. In either case, the first player can recreate the state to be a rectangle and one additional square, i.e., the first player can force a win. By playing $(n-1,1)$ as his initial move, the first player can create this situation and therefore force a win.

Problem 21.8, *pg. 165*: *Solve Problem 21.6 on Page 164 if the rectangle is of dimension $n \times m$.*

Solution 21.8: Suppose the second player has a winning strategy. Suppose the first player chose $(n-1, m-1)$ as his initial choice and the second player countered with position (i, j), leaving the set S of squares. Now, it is the first player's turn and from this set, by hypothesis, the second player can force a win. However the first player could have chosen (i, j) as his initial move and the set of remaining squares would be S (since the square $(n-1, m-1)$ is above and to the right of all other squares) with the second player's turn.

This contradicts the hypothesis that the second player has a winning strategy; therefore the first player must have a winning strategy.

Note that this solution does not give an explicit strategy, unlike Solution 21.6 on the facing page and Solution 21.7 on the current page. The argument used above is sometimes called "strategy stealing." Explicit strategies for $n \times m$ chomp are known for only a few specific cases.

Problem 21.9, *pg. 165*: *Building i has r_i residents, and is at distance d_i from the beginning of the street. Devise an algorithm that computes a distance m from the beginning of the street for the mailbox that minimizes the total distance, that residents travel to get to the mailbox, i.e., minimizes $\sum_{i=0}^{n-1} r_i|d_i - m|$.*

Solution 21.9: Suppose the total number of residents $t = \sum_{i=0}^{n-1} r_i$ is odd. Sort the residents according to their distance from the beginning of the street. The optimum location for the mailbox is at the building corresponding to the resident at the median according to the sort order above. Call this resident the median resident.

The argument is as follows. Each location to the left of the building b the median resident lives in is suboptimum, since we can reduce the total distance by moving the mailbox to b (more residents see a reduced distance than see an increased distance). In the same way, each mailbox location to the right of b is suboptimum.

If the number of residents is even, any location between the buildings that the $\lfloor \frac{t}{2} \rfloor$-th and $(\lfloor \frac{t}{2} \rfloor + 1)$-th residents live in is optimum. (If they live in the same building, which is the optimum building.)

If the buildings are given sorted by distance from the beginning of the street, the median resident can be found in linear time by iterating through the offsets in increasing order, adding the corresponding r_is until the count gets to $\frac{t}{2}$. Note that the time complexity is $O(n)$ where n is the number of buildings, not $O(t)$.

If the distances are not sorted, we can sort in $O(n \log n)$, and then find the median. Alternately, we can find the median using extensions of the randomized median finding algorithm for unweighted data. In particular, the randomized median finding algorithm presented in Solution 11.13 on Page 270 can be made to work in linear time, even for the weighted case.

Variant 21.9.1: Compute a location that minimizes the sum of the squares of the distances traveled by the residents.

Variant 21.9.2: Compute a location that minimizes the maximum distance that any resident travels.

Problem 21.10, *pg. 165*: *Given an instance of the gasup problem, how would you efficiently compute an ample city if one exists?*

Solution 21.10: Consider the thought experiment of starting at an arbitrary city with sufficiently large amount of gas so that we can complete the loop. In this experiment, we note the amount of gas in the tank as the vehicle goes through the loop at each city before loading the gas kept in that city for the vehicle. Let C be a city where the amount of gas in the tank before we refuel at that city is minimum. Call this minimum amount of gas m. Now suppose we pick C as the starting point, and we have no gas. Since we never have less gas than we started with at C, we can complete the journey without running out of gas. The computation to determine C can be easily done in linear time with a single pass over all the cities.

```
template <typename T>
int find_start_city(const vector<T> &G, const vector<T> &D) {
  T carry = 0;
  pair<int, T> min(0, 0);
  for (int i = 1; i < G.size(); ++i) {
    carry += G[i - 1] - D[i - 1];
    if (carry < min.second) {
      min = {i, carry};
    }
  }
  return min.first;
}
```

Problem 21.11, *pg. 166*: *Write a function that takes a nonnegative integer x and returns, as a string, the integer closest to x whose decimal representation is a palindrome. For example,*

given 1224, *you should return* 1221.

Solution 21.11: If x is a palindrome, we simply return x. Assume x is not a palindrome. Treat x as a string of digits. For simplicity, assume x is of even length. Let t be the left half of x, and b be the right half of x. Let s be the smallest palindrome greater than x, and l be the largest palindrome smaller than x.

Conceptually, there are three candidates for the palindrome closest to x. One is the integer which agrees with the first half of x. The other two are formed from $t+1$ and $t-1$. In all three cases, the second half of the candidate is implied by the first half. For the given example, we try 1221, 1331, and 1111. The palindrome formed by mirroring $t+1$ may be strictly greater than s; if so, s is guaranteed to be the mirror of t. A parallel result holds for $t-1$.

For odd length x, we use the same approach, but take t to be the digits to the left of the center digit.

```
unsigned diff(const unsigned &a, const unsigned &b) {
  return a > b ? a - b : b - a;
}

unsigned find_closest_palindrome(const unsigned &x) {
  string str(to_string(x));
  // Make str a palindrome by mirroring the left half to the right half
  copy(str.cbegin(), str.cbegin() + (str.size() >> 1), str.rbegin());

  unsigned mirror_left = stoul(str);
  int idx = (str.size() - 1) >> 1;
  if (mirror_left >= x) {
    // Subtract one from the left half
    while (idx >= 0) {
      if (str[idx] == '0') {
        str[idx--] = '9';
      } else {
        --str[idx];
        break;
      }
    }
    if (str[0] == '0') {  // special case, make the whole string as "99...9"
      str = to_string(stoul(str));  // removes the leading 0
      fill(str.begin(), str.end(), '9');
    }
  } else {  // mirror_left < x
    // Add one to the left half
    while (idx >= 0) {
      if (str[idx] == '9') {
        str[idx--] = '0';
      } else {
        ++str[idx];
        break;
      }
    }
  }

  // Make str a palindrome again by mirroring the left half to the right half
```

```
39    copy(str.cbegin(), str.cbegin() + (str.size() >> 1), str.rbegin());
40    return diff(x, mirror_left) < diff(x, stoul(str)) ?
41           mirror_left : stoul(str);
42  }
```

ϵ-Variant 21.11.1: Find the palindrome p closest to a nonnegative integer x, subject to the constraint that $p \neq x$. For example, if $x = 999$, then $p = 1001$.

Problem 21.12, *pg.166*: *Given an array A with n elements, compute* $\max_{j=0}^{n-1} \frac{\prod_{i=0}^{n-1} A[i]}{A[j]}$ *in $O(n)$ time without using division. Can you design an algorithm that runs in $O(1)$ space and $O(n)$ time? Array entries may be positive, negative, or 0.*

Solution 21.12: Let $L_p = \prod_{i=0}^{p} A[i]$ and $R_p = \prod_{j=p}^{n-1} A[j]$. Observe computing L_p and R_p individually takes p and $(n-1)-p$ multiplications, respectively; however, we can compute all the L_p and R_p using $2(n-1)$ multiplications, since $L_p = L_{p-1}A[p]$, and $R_p = A[p]R_{p+1}$. The product of all elements except the i-th one is simply $L_{i-1}R_{i+1}$, hence we can compute these n products using n multiplications, once we have L and R computed. Finding the largest product is simply an iteration with compare and swap in the loop. The time complexity is $O(n)$ and the solution uses two arrays of length n each.

```
1  template <typename T>
2  T find_biggest_n_1_product(const vector<T> &A) {
3    // Build forward product L, and backward product R
4    vector<T> L, R(A.size());
5    partial_sum(A.cbegin(), A.cend(), back_inserter(L), multiplies<T>());
6    partial_sum(A.crbegin(), A.crend(), R.rbegin(), multiplies<T>());
7
8    // Find the biggest product of (n - 1) numbers
9    T max_product = numeric_limits<T>::min();
10   for (int i = 0; i < A.size(); ++i) {
11     T forward = i > 0 ? L[i - 1] : 1;
12     T backward = i + 1 < A.size() ? R[i + 1] : 1;
13     max_product = max(max_product, forward * backward);
14   }
15   return max_product;
16 }
```

It is possible to solve this problem with only $O(1)$ additional storage, with a sophisticated case analysis. Suppose $A[i] \neq 0$ for all i. If A contains an odd number of negative entries, the optimum product is formed when we exclude the biggest negative entry. Otherwise, suppose A contains an even number of negative entries. If some entries are positive, the optimum product is achieved when we exclude the smallest positive entry; otherwise, we exclude the smallest negative number.

Now suppose two or more zeros are present in A. Then the product of any $n-1$ entries is always 0. Suppose there is exactly one zero. If A contains an odd number of negative numbers, the optimum product is zero; otherwise it is the product of

all elements excluding the zero. The code can be readily implemented with a small number of traversals of the array, leading to an $O(n)$ time complexity.

```
template <typename T>
T find_biggest_n_1_product(const vector<T> &A) {
  int zero_count = 0, pos_count = 0, neg_count = 0;
  int zero_idx = -1, s_neg_idx = -1, b_neg_idx = -1, s_pos_idx = -1;

  for (int i = 0; i < A.size(); ++i) {
    if (A[i] < 0) {
      ++neg_count;
      if (s_neg_idx == -1 || A[i] < A[s_neg_idx]) {
        s_neg_idx = i;
      }
      if (b_neg_idx == -1 || A[b_neg_idx] < A[i]) {
        b_neg_idx = i;
      }
    } else if (A[i] == 0) {
      zero_idx = i, ++zero_count;
    } else {  // A[i] > 0
      ++pos_count;
      if (s_pos_idx == -1 || A[i] < A[s_pos_idx]) {
        s_pos_idx = i;
      }
    }
  }

  // Try to find a number whose elimination could maximize the product of
  // the remaining (n - 1) numbers.
  int x;  // stores the idx of eliminated one
  if (zero_count >= 2) {
    return 0;
  } else if (zero_count == 1) {
    if (neg_count & 1) {
      return 0;
    } else {
      x = zero_idx;
    }
  } else {
    if (neg_count & 1) {  // odd number negative
      x = b_neg_idx;
    } else {  // even number negative
      if (pos_count > 0) {
        x = s_pos_idx;
      } else {
        x = s_neg_idx;
      }
    }
  }

  T product = 1;
  for (int i = 0; i < A.size(); ++i) {
    if (i != x) {
      product *= A[i];
    }
```

```
  }
  return product;
}
```

Variant 21.12.1: Let A be as above. Compute an array B where $B[i]$ is the product of all elements in A except $A[i]$. You cannot use division. Your time complexity should be $O(n)$, and you can only use $O(1)$ additional space.

Problem 21.13, *pg. 166*: *Given c cases and d drops, what is the maximum number of floors that you can test in the worst-case?*

Solution 21.13: Let $F(c,d)$ be the maximum number of floors we can test with c identical cases and at most d drops. We know that $F(1,d) = d$. Suppose we know the value of $F(i,j)$ for all $i \leq c$ and $j \leq d$.

If we are given $c+1$ cases and d drops we can start at floor $F(c,d-1)+1$ and drop a case. If the case breaks, then we can use the remaining c cases and $d-1$ drops to determine the floor exactly, since it must be in the range $[1, F(c,d-1)]$. If the case did not break, we proceed to floor $F(c,d-1)+1+F(c+1,d-1)$.

Therefore F satisfies the recurrence

$$F(c+1,d) = F(c,d-1)+1+F(c+1,d-1).$$

We can compute F using DP as below:

```
int get_height_helper(vector<vector<int>> &F, const int &c, const int &d) {
  if (d == 0) {
    return 0;
  } else if (c == 1) {
    return d;
  } else {
    if (F[c][d] == -1) {
      F[c][d] = get_height_helper(F, c, d - 1) +
                get_height_helper(F, c - 1, d - 1) + 1;
    }
    return F[c][d];
  }
}

int getHeight(const int &c, const int &d) {
  vector<vector<int>> F(c + 1, vector<int>(d + 1, -1));
  return get_height_helper(F, c, d);
}
```

Variant 21.13.1: How would you compute the minimum number of drops needed to find the breaking point from 1 to F floors using c cases?

Variant 21.13.2: Men numbered from 1 to n are arranged in a circle in clockwise order. Every k-th man is removed, until only one man remains. What is the number

of the last man?

Problem 21.14, *pg. 166*: *Moles are numbered from 0 to $n-1$. Mole m has a set of neighboring moles. Whacking m when it is up results in it and all of its neighbors flipping state. Given a set of moles, the neighbors for each mole, and an initial assignment of up/down states for each mole, compute a sequence of whacks (if one exists) that results in each mole being in the down state.*

Solution 21.14: First we make the observation that any pair of whacks commute, that is that the state resulting from a whack to mole m_i followed by a whack to mole m_j is the same as the state resulting from whacking m_j followed by m_i. This generalizes to arbitrary sequences of whacks, which in turn implies that a state can be achieved from an initial state s_0 iff it can be achieved by whacking each mole at most once.

Introduce a Boolean variable x_i for Mole i—this variable indicates whether the mole is to be whacked. Observe that the state of Mole i after the whacks encoded by the x_is is $y_i = x_i \oplus x_{n_0^i} \oplus x_{n_1^i} \cdots \oplus x_{n_{i_{k-1}}^i}$. Therefore the problem reduces to computing an assignment to the x_is that sets each y_i to 0.

The standard approach to solving linear equations of the form $y = Ax$ is Gaussian elimination. Iteratively remove variable x_i from all equations after the i-th one. This results in an equation in one unknown, which is then solved, and iteratively substituted back into the previous equations.

```
1  void Eliminate_rows(vector<vector<bool>> &B, const int &i, const int &j) {
2    // Use B[i] to eliminate other rows' entry j
3    for (int a = 0; a < B.size(); ++a) {
4      if (i != a && B[a][j]) {
5        for (int b = 0; b < B[i].size(); ++b) {
6          B[a][b] = B[a][b] ^ B[i][b];
7        }
8      }
9    }
10 }
11
12 vector<bool> Gaussian_elimination(const vector<vector<bool>> &A,
13                                   const vector<bool> &y) {
14   vector<vector<bool>> B(A);
15   for (int i = 0; i < B.size(); ++i) {
16     B[i].push_back(y[i]);
17   }
18
19   for (int i = 0; i < B.size(); ++i) {
20     // Find the coefficient starting with 1
21     int idx = i;
22     for (int j = i + 1; j < B.size(); ++j) {
23       if (B[j][i]) {
24         idx = j;
25         break;
26       }
27     }
28     swap(B[i], B[idx]);
29
```

```
    // Perform elimination except i-th row
    if (B[i][i]) {
      Eliminate_rows(B, i, i);
    }
  }

  for (int i = B.size() - 1; i >= 0; --i) {
    if (B[i][i] == false) {
      bool have_coefficient = false;
      for (int j = i + 1; j < A.size(); ++j) {
        if (B[i][j]) {
          Eliminate_rows(B, i, j);
          have_coefficient = true;
          swap(B[i], B[j]);  // row permutation
          break;
        }
      }

      if (have_coefficient == false && B[i].back() == true) {
        cout << "No solution." << endl;
        return {};
      }
    }
  }

  vector<bool> x;
  for (int i = 0; i < B.size(); ++i) {
    x.push_back(B[i].back());
  }
  return x;
}
```

Problem 21.15, ***pg. 167***: *Let F be an $n \times n$ Boolean 2D array representing the "knows" relation for n people; $F[a][b]$ is* `true` *iff a knows b, and $F[a][a]$ is always* `false`*. Design an $O(n)$ algorithm to find the celebrity.*

Solution 21.15: We start by checking $F[0][1]$, i.e., the relation between Person 0 and Person 1. Assuming that we are checking $F[i][j]$ where $i < j$, the key idea here is that if $F[i][j]$ is `false`, we know that j is not the celebrity and i is still a possible celebrity candidate, allowing us to eliminate j from the set of celebrity candidates by advancing from $F[i][j]$ to $F[i][j+1]$. If $F[i][j]$ is `true`, we know that i is not the celebrity, and for all $j' < j$, j' is not a celebrity because $F[i][j']$ must be `false`, allowing us to advance from $F[i][j]$ to $F[j][j+1]$ since $i < j$. We eliminate one candidate each step in $O(1)$ time which gives us a $O(n)$ time algorithm.

```
int celebrity_finding(const vector<vector<bool>> &F) {
  // Start checking the relation from F[0][1]
  int i = 0, j = 1;
  while (j < F.size()) {
    if (F[i][j] == true) {
      i = j++;  // all candidates j' < j are not celebrity candidates
    } else {  // F[i][j] == false
```

```
8          ++j;  // i is still a celebrity candidate but j is not
9        }
10     }
11     return i;
12   }
```

Variant 21.15.1: Solve the same problem when the knows relation is not reflexive, nor anti-reflexive, i.e., some people may know themselves, and others may not.

Problem 21.16, *pg. 167*: *Six guests attend a party. Any two guests either know each other or do not know each other. Prove that there exists a subset of three guests who either all know each other or all do not know each other.*

Solution 21.16: This problem can be modeled using undirected graphs where vertices correspond to guests. Add an edge between each pair of guests. Color an edge between a pair of guest "blue" if they are friends, otherwise, color it "red".

Then the theorem is equivalent to the claim that in any clique on six vertices, where each edge is either blue or red, there exists a subset of three vertices, all connected by edges of the same color.

Choose any vertex v. Examine the five edges having v as an endpoint. By the pigeon-hole principle, there must be at least three edges which are of the same color c. Let $(v,\alpha),(v,\beta)$, and (v,γ) be three such edges. Now, either there is an edge colored c between α,β, and γ, in which case v and the two vertices in α,β, and γ that are connected by a c-colored edge are three vertices all connected by c-colored edges, or there is no such edge. In the latter case, α,β, and γ are themselves connected by edges that are of the same color.

Ramsey's theorem is illustrated in Figure 21.17. In each graph, the three shaded nodes are either all connected or all disconnected.

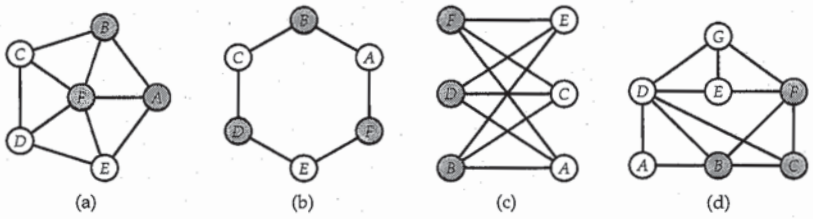


Figure 21.17: Representative graphs on six vertices.

Problem 21.17, *pg. 167*: *Prove that every tournament has a Hamiltonian path, i.e., a path that includes each vertex exactly once.*

Solution 21.17: We use induction on the number of vertices. Specifically, the induction hypothesis is that every tournament on n vertices has a Hamiltonian path.

Proof:

The base case, namely $n = 1$, is trivial. For the induction step, let T be a tournament on $n + 1$ vertices, and v_n be any vertex in T. Note that removing a vertex together with its incoming and outgoing edges from any tournament on $n + 1$ vertices leaves a tournament on n vertices.

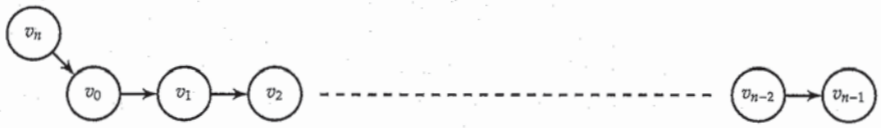
Let $\langle v_0, v_1, \ldots, v_{n-1} \rangle$ be a Hamiltonian path in the graph resulting from the removal of v_n from T. Its existence is guaranteed by the induction hypothesis. There exist three possibilities:

(1.) An edge exists from v_n to v_0. Then $\langle v_n, v_0, v_1, \ldots, v_{n-1} \rangle$ is a Hamiltonian path in T, so induction goes through.

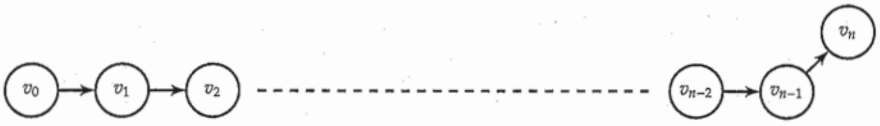
(2.) An edge exists from v_{n-1} to v_n. Then $\langle v_0, v_1, \ldots, v_{n-1}, v_n \rangle$ is a Hamiltonian path in T, so induction goes through.

(3.) No edge exists from v_n to v_0 and no edge exists from v_{n-1} to v_n. Then there must be an edge from v_0 to v_n, and an edge from v_n to v_{n-1}. Let i be the maximum over all j such that (v_j, v_n) is an edge in T. Since we have ruled out $i = n - 1$ above, there must be an edge from v_n to v_{i+1}. The path $\langle v_0, v_1, \ldots, v_i, v_n, v_{i+1}, \ldots v_{n-1} \rangle$ is a Hamiltonian path in T, so induction goes through.

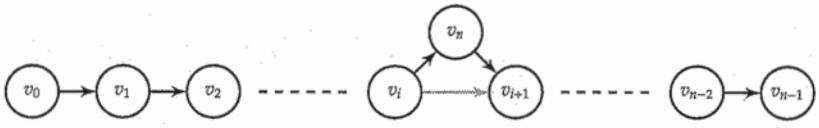
The different cases are illustrated in Figure 21.18. The proof can be shortened to



(a) Case (1.). $\langle v_n, v_0, v_1, \ldots, v_{n-1} \rangle$ is a Hamiltonian path.



(b) Case (2.). $\langle v_0, v_1, \ldots, v_{n-1}, v_n \rangle$ is a Hamiltonian path.



(c) Case (3.). $\langle v_0, v_1, \ldots, v_i, v_n, v_{i+1}, \ldots v_{n-1} \rangle$ is a Hamiltonian path.

Figure 21.18: Case analysis view graphically.

the following: choose $i \in \{0, 1, 2, \ldots, n-1\}$ to be maximal such that for every $j \leq i$ there is an edge from v_j to v_n; Then the sequence $\langle v_0, v_1, \ldots, v_i, v_n, v_{i+1}, \ldots, v_{n-1} \rangle$ is a Hamiltonian cycle.

Problem 21.18, *pg. 167*: *Design an algorithm which takes the preference lists of the students and the professors and pairs students one-to-one with professors subject to the constraint that*

there do not exist student-professor pairings $(s0, p0)$ *and* $(s1, p1)$ *such that* $s0$ *prefers* $p1$ *to* $p0$ *and* $p1$ *prefers* $s0$ *to* $s1$. *(The preferences of* $p0$ *and* $s1$ *are not important.)*

Solution 21.18: This problem can be solved using a "proposal algorithm". We will refer to the professor a student is paired with as his adviser. Each student who does not have an adviser "proposes" to the professor he likes best to whom he has not yet proposed.

Each professor then considers all the students who have sent him proposals and tells the student in this set that he most prefers "lets talk more", and he says "no" to all the other student who proposed him.

In each subsequent round, each student who does not have an adviser proposes to one professor to whom he has not yet proposed. This is done regardless of whether the professor has already been matched. The professor once again replies with a single "lets talk more", rejecting the rest. Note that a professor who has already said "lets talk more" to a student may switch to another student he prefers more, leaving the first student unmatched.

This algorithm has two key properties, which immediately yield its correctness.

- It converges to a state where everyone is paired: Once a professor begins talking to a student, he always has a student to talk to. There cannot be a professor and a student both unpaired since the student must made a request to that professor at some point—a student will eventually propose to everyone, if necessary and being unpaired, the professor would have accepted.
- The pairings are stable: Let Riemann be a student and Gauss be a professor. Suppose Gauss is talking to Dedekind and Riemann is talking to Eisenstein. Upon completion of the algorithm, it is not possible for both Riemann and Gauss to prefer each other over their current pairings. If Riemann prefers Gauss to Eisenstein, he must have asked Gauss before he asked Eisenstein. If Gauss accepted Riemann, but is not paired to Riemann at the end, he must have rejected Riemann for a student he preferred more than Riemann and therefore does not like Riemann more than Dedekind.

```
vector<pair<int, int>> find_stable_assignment(
    const vector<vector<int>> &professor_preference,
    const vector<vector<int>> &student_preference) {
  queue<int> free_student;  // stores currently free students
  for (int i = 0; i < student_preference.size(); ++i) {
    free_student.emplace(i);
  }

  // Records the professor each student have asked
  vector<int> student_pref_idx(student_preference.size(), 0);
  // Records each professor currently student choice
  vector<int> professor_choice(professor_preference.size(), -1);

  while (free_student.empty() == false) {
    int i = free_student.front();   // free student
    int j = student_preference[i][student_pref_idx[i]];   // target professor
    if (professor_choice[j] == -1) {  // this professor is free
      professor_choice[j] = i;
```

```
      free_student.pop();
    } else {  // this professor has student now
      int original_pref = find(professor_preference[j].cbegin(),
                               professor_preference[j].cend(),
                               professor_choice[j]) -
                          professor_preference[j].cbegin();
      int new_pref = find(professor_preference[j].cbegin(),
                          professor_preference[j].cend(), i) -
                     professor_preference[j].cbegin();
      if (new_pref < original_pref) {   // this professor prefers the new one
        free_student.emplace(professor_choice[j]);
        professor_choice[j] = i;
        free_student.pop();
      }
    }
    ++student_pref_idx[i];
  }

  vector<pair<int, int>> match_result;
  for (int j = 0; j < professor_choice.size(); ++j) {
    match_result.emplace_back(professor_choice[j], j);
  }
  return match_result;
}
```

Problem 21.19, ***pg. 168***: *Design an algorithm for pairing bidders with celebrities to maximize the revenue from the dance. Each celebrity cannot dance more than once, and each bidder cannot dance more than once. Assume that the set of celebrities is disjoint from the set of bidders. How would you modify your approach if all bids were for the same amount? What if celebrities and bidders are not disjoint?*

Solution 21.19: The problem can directly be mapped into the weighted bipartite matching problem. Bidders and celebrities constitute the left and right vertices; an edge exists from b to c iff b has offered money to dance with c, and the weight of an edge is the amount offered for the dance. It can be solved using specialized algorithms, flow network, or linear programming.

If the bids are all in the same amount, the problem is that of unweighted bipartite matching. If the requirement that bidders and celebrities be distinct is dropped, the problem becomes a weighted matching problem in a general graph. Both of these variants are solvable in polynomial time.

Problem 21.20, ***pg. 168***: *Suppose two squares of opposite colors are removed from a chessboard. Design an algorithm for finding a way to cover the remaining squares using* 31 *dominoes, if a covering exists.*

Solution 21.20: Model the original chessboard as a bipartite graph B, with 32 white vertices on the left and 32 black vertices on the right. Vertices correspond to squares. Put an edge between vertices whose corresponding squares are adjacent. In particular, corner vertices are connected to two edges; edge vertices are connected to three

edges; and remaining vertices are connected to four edges.

Removing a white and a black square from the chessboard corresponds to removing a left and a right vertex, together with the edges they are connected to. Call the new graph B'.

Observe that a single domino placed on the chessboard covers adjacent white and black vertices. Therefore a valid placement of dominoes corresponds to a set of edges, no two of which share a vertex. Such sets are commonly referred to as matchings.

The desired covering corresponds to a matching in B' containing 31 edges, which is the maximum possible. Hence we can solve our problem using well-known algorithms finding a maximum matching in a bipartite graph.

Alternately, it is fairly simple to construct a Hamiltonian cycle in B—a sequence $\langle v_0, v_1, \ldots, v_{63}, v_0 \rangle$, such that for all $i, 0 \leq i \leq 63$, $(v_i, v_{i+1 \bmod 64})$ is an edge. If we remove a single black and a single white vertex, we can construct a matching that covers all the remaining vertices by beginning immediately after the removed vertices and continuing along the edges in the Hamiltonian cycle. Correctness follows from the fact that there will be an even number of vertices on the subpaths in the Hamiltonian cycle that result on the deletion of the two vertices. This construction is illustrated in Figure 21.19. Therefore, regardless of the white and vertices that are removed, there will always be a suitable covering of the chessboard. (This is not apparent from the maximum matching formulation.)

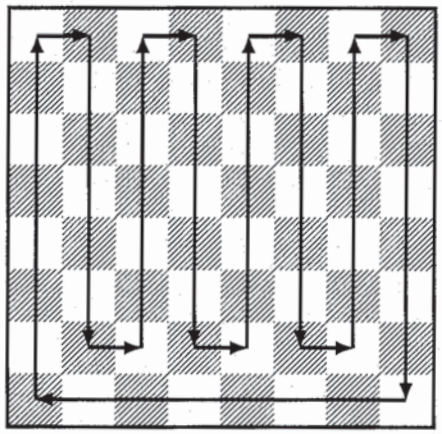


Figure 21.19: Hamiltonian cycle in a chessboard.

Variant 21.20.1: Is it true that when *any* single square is removed from a $2^k \times 2^k$ chessboard, the remaining $2^k \times 2^k - 1$ squares can be tiled with triominoes? (See also the discussion of mutilated chessboards on Page 30.)

Problem 21.21, *pg. 168*: *This problem is a continuation of Problems 13.6 on Page 100*

and 16.7 on Page 135. Design an efficient algorithm for computing the minimum number of subsets of teams so that (1.) the teams in each subset can be organized to appear in a single photograph without violating the placement constraint, and (2.) each team appears in exactly one subset.

Solution 21.21: In Solution 16.7 on Page 382, we showed how to model the problem using a DAG, with each vertex corresponding to a team. The current problem is asking for a minimum cardinality set of paths in this DAG such that each vertex appears on some path.

This is a classic problem that can be solved using maximum bipartite matching as a subroutine. Let $G = (V, E)$ be a DAG. Construct a bipartite graph B from G as follows. Each vertex $v \in V$ is represented with two vertices v and v'. For each (directed) edge $(u, v) \in E$ add an (undirected) edge (u, v') to B.

We now prove a one-to-one correspondence between matchings in B and vertex-disjoint paths that cover all vertices in G.

Proof:

Let Π be a set of vertex-disjoint paths. Initialize M to the empty set. For each $\langle u, v, w, \ldots, x, y \rangle \in \Pi$ add the edges $(u, v'), (v, w'), \ldots, (x, y')$ to M. We claim M is a matching. For two edges of the form (a, b') and (a, c') to be present in M, the edges (a, b) and (a, c) must have been present in two different paths. (The paths must be different since G is a DAG.) This contradicts the assumption that the paths in Π are vertex-disjoint. A similar argument holds for edges of the form (a', b) and (a', c).

Conversely, let M be a matching in B. Construct a set of vertex-disjoint paths Π that cover G from M as follows. Initialize S to the empty set. While $S \neq V$, choose a vertex $v \notin S$ and iteratively create a path p in G including v as follows. If v is matched in M with say a', add (v, a) to p and continue the construction with a, stopping when an unmatched vertex is encountered. Additionally, if v' is matched with b add (b, v) to p and continue the construction with b. Since M is a matching, we will never add the same vertex to two paths. Furthermore, since G is a DAG, the construction of p will always end.

Observe that $|\Pi| = |V| - |M|$, since each edge in a path corresponds to a matched edge in B. Therefore we can find a minimum vertex-disjoint set of paths that covers G by finding a maximum matching in B.

The proof shows that each unmatched vertex v corresponds to the last vertex in a path, and each unmatched vertex v' corresponds to the first vertex in a path. Therefore the number of unmatched vertices in B is twice the number of paths in the corresponding set of vertex-disjoint paths in G. Therefore, the minimum number of vertex-disjoint paths covering G can be determined by computing a maximum matching in B.

The construction is illustrated in Figure 21.20 on the facing page. Figure 21.20(a) on the next page is a DAG covered by four paths, $\langle b, c, d \rangle$, $\langle f, e, h \rangle$, $\langle a \rangle$, and $\langle g \rangle$. We use thick edges to denote edges in paths; thin edge are not part of the path covering. Different paths are shaded differently. The corresponding bipartite graph and

matching are shown in Figure 21.20(b); dark edges are the ones in the matching. The unmatched vertices on the lower row are the vertices that end paths, and the unmatched on the upper row are the vertices that path begin from. An augmenting path for a matching is a path beginning and ending at unmatched vertices which alternates between unmatched and matched edges. The existence of an augmenting path $\langle a', c, d', e, h', g \rangle$, indicates that the matching is suboptimum. The path $\langle a', c, d', e, h', g \rangle$ is an augmenting path, demonstrating that the matching in Figure 21.20(b) is suboptimum. Figure 21.20(c) illustrates the same DAG covered with three paths, $\langle b, c, a \rangle$, $\langle f, e, d \rangle$, and $\langle g, h \rangle$. The corresponding bipartite graph and matching are shown in Figure 21.20(d). The matching in Figure 21.20(d) is maximum, since there is no augmenting path for it.

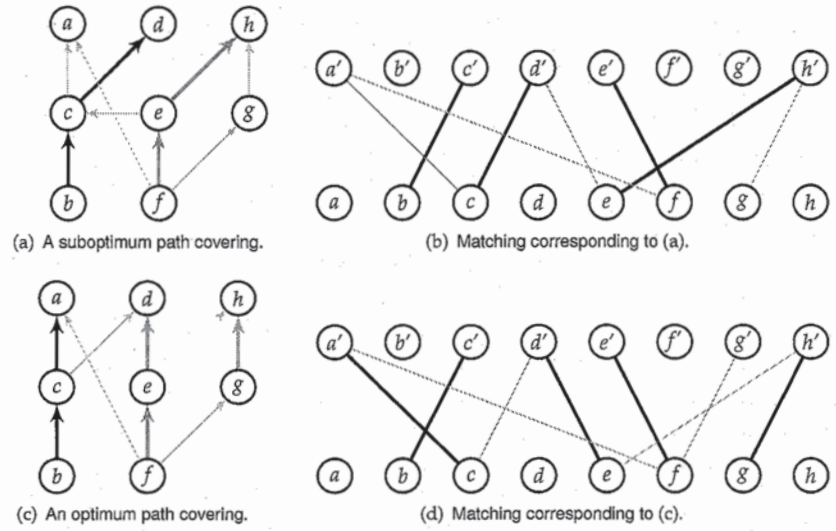


Figure 21.20: Relationship between paths and matchings.

Variant 21.21.1: Define a subset S of $\mathcal{Z}^+$ to be division-free if $\nexists x, y \in S$ such that $x \bmod y = 0$. How would you compute a maximum cardinality subset division-free subset of $\mathcal{Z}_n$?

Variant 21.21.2: Define G to be the graph on $V = \{0, 1, \ldots, m-1\} \times \{0, 1, \ldots, n-1\}$, and $E = \{((i, j), (i+1, j)) \mid 0 \leq i < m-1, 0 \leq j \leq n-1\} \cup \{((i, j), (i, j+1)) \mid 0 \leq i \leq m-1, 0 \leq j < n-1\}$. Let S be a subset of the vertices. How would you compute the minimum number of paths that begin at $(0, 0)$ and end at $(m-1, n-1)$ and include each vertex in S at least once?

Problem 21.22, *pg. 168*: *Let A and B be rooted trees. Design a polynomial time algorithm for computing a largest common rooted subtree of A and B.*

Solution 21.22: We use DP to build a map L from pairs $u \in A$ and $v \in B$ to the size of a largest subtree rooted specifically at u and at v.

Denote by $C(x)$ the set of children nodes of x. Suppose we know $L(u', v')$ for all $u' \in C(u)$ and $v' \in C(v)$. Finding the largest common subtree rooted at u and v entails finding a one-to-one mapping m from $M_u \subset C(u)$ to $M_v \subset C(v)$ that maximizes $\sum_{u' \in M_u} L(u', m(u'))$.

Form a complete bipartite graph B on $C(u) \cup C(v)$, with the weight of edge (u', v') being $L(u', v')$. The desired mapping is simply a maximum weighted bipartite matching in B.

The complexity of this procedure is polynomial, but with a high degree. Given a bipartite graph with $|V|$ vertices and $|E|$ edges, a maximum weighted matching can be computed in $O(|V|^2|E|)$ time using max-flow algorithms. This can be improved using specialized matching algorithms to $O(|V||E| + |V|^2 \log |V|)$. In our setting $|E| = \Theta(|V|^2)$, since $L(x, y) \geq 1$ for all x, y. If A and B each have n nodes, the weighted bipartite matching algorithm may be called with $|V| = \Theta(n)$, and $O(n^2)$ such calls are made, leading to a $O(n^5)$ time bound, when the best weighted bipartite matching algorithm is used. In practice it is unlikely that the matching routine will be called n^2 times with large bipartite graphs.

Variant 21.22.1: Given two binary trees, compute the largest subtree that is contained in both. Ignore node contents.

Variant 21.22.2: Given two binary trees with values stored at nodes, compute the largest subtree that is contained in both. Isomorphic nodes must have the same stored values.

Problem 21.23, *pg. 169*: *Consider a league in which teams are numbered from 0 to $n-1$. At a certain point in the season, Team i has won W_i games, and has $R_{i,j}$ games remaining with Team j. Each game will end in a win for one team and a loss for the other team. Show how the problem of determining whether Team a is mathematically eliminated can be solved using maximum flow.*

Solution 21.23: The most wins Team a can end the season with is $\mu_a = W_a + \sum_{j \neq a} R_{a,j}$. The idea is to use max-flow to check if it is not possible for all other teams to simultaneously win no more than μ_a games, in which case a is eliminated.

We define an instance of maximum flow as follows. Let $\{s\} \cup \{r_{i,j} \mid i \neq a, j \neq a, \text{ and } i < j\} \cup \{v_i \mid i \neq a\} \cup \{t\}$ be a set of vertices. Add an edge from s to each vertex of the form $r_{i,j}$ with capacity $R_{i,j}$. Add edges from $r_{i,j}$ to v_i and to v_j with capacity ∞ for each vertex of the form $r_{i,j}$. Add an edge from each vertex v_i to t with capacity $\mu_a - W_i$. This construction is illustrated in Figure 21.21 on the next page.

An integral flow f from s can be interpreted as follows. The flow through $(s, (r_{i,j}))$ is split into flow through v_i and v_j; it is the assignment of $f(s, (r_{i,j}))$ victories across Team i and Team j. The maximum number of victories that Team i can have without eliminating a is $\mu_a - W_i$; this is captured by the capacities on edges of the form (v_i, t).

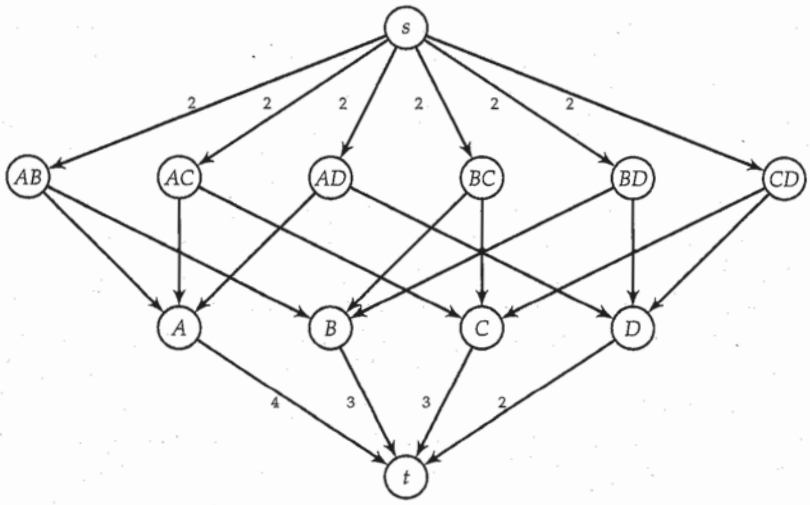


Figure 21.21: Maximum flow instance for the example described on the current page. Nodes immediately below s correspond to matches between pairs of teams $\{A, B, C, D\}$. Edges that are not labeled with a capacity have infinite capacity.

Suppose there exists a flow that saturates all the edges emanating from s, i.e., has value $\sum_{\{(i,j)|i<j \text{ and } i,j\neq a\}} R_{i,j}$. This must be a maximum flow, since it is the capacity of all the edges emanating from s. By the reasoning above, this flow assigns outcomes to all the games that remain to be played that do not involve a, and by the (v_i, t) edge capacity constraint, this assignment does not lead to the elimination of a.

Conversely, suppose a is not eliminated because of some combination of outcomes for the games. These outcomes yield a flow that has value $\sum_{\{(i,j)|i<j \text{ and } i,j\neq a\}} R_{i,j}$ which must satisfy the capacity constraints (since there are $R_{i,j}$ games remaining between Team i and Team j, and Team i wins no more than $\mu_a - W_i$ games).

Putting the two arguments together, we have proved that the network has a maximum flow of value $\sum_{\{(i,j)|i<j \text{ and } i,j\neq a\}} R_{i,j}$ iff a is not eliminated.

As a concrete example, consider five teams, A, B, C, D, and E. We want to check if Team E is eliminated. Two games remain to be played between each pair in $\{A, B, C, D\}$, and E has a total of five games to play against the remaining four teams. Currently, E has one victory less than A, two fewer victories than B and C, and three fewer victories than D.

The technique we presented above yields the maximum flow problem in Figure 21.21, and E is not eliminated iff there exists a flow that saturates all the edges coming out of s. The following flow function achieves this: $f(AB, A) = 1, f(AB, B) = 1, f(AC, A) = 1, f(AC, C) = 1, f(AD, A) = 2, f(AD, D) = 0, f(BC, B) = 0, f(BC, C) = 2, f(BD, B) = 2, f(BD, D) = 0, f(CD, C) = 0, f(CD, D) = 2$, with the remaining values for f implied by those above.

Variant 21.23.1: Let G be a DAG in which each vertex v has an associated value

$c(v) \in \mathcal{Z}$. How would you select a subset S of vertices having maximum value, subject to the constraint that if $v \in S$ and $(u, v) \in E$ then $u \in S$?

Problem 21.24, *pg. 169*: *Design an efficient algorithm for computing a rounding of a matrix, if one exists.*

Solution 21.24: Clearly, a necessary condition for a rounding of A to exist is that each row and each column of A sums to an integer. We will show that this condition is also sufficient via a constructive argument.

We formulate the problem as a variant of the max-flow problem, specifically, as an instance of the circulation with demands problem.

We define a directed graph $G = (V, E)$ as follows. The vertex set $V = \{s, t, u_0, u_1, \ldots, u_{m-1}, v_0, v_1, \ldots, v_{n-1}\}$. Add directed edges (s, u_i), (v_j, t), (u_i, v_j), and (t, s) for all $0 \leq i \leq m-1$, $0 \leq j \leq n-1$. Assign a lower bound ("demand") $L(x, y)$ and upper bound ("capacity") $U(x, y)$ to edges as follows: $L(s, u_i) = U(s, u_i) = \sum_{j=0}^{n-1} A[i, j]$, $L(v_j, t) = U(v_j, t) = \sum_{i=0}^{m-1} A[i, j]$, $L(u_i, v_j) = \lfloor A[i, j] \rfloor$, $U(u_i, v_j) = \lceil A[i, j] \rceil$, $L(t, u) = 0$, and $U(t, u) = \infty$.

We know that we can assign flows through the edges of G that satisfy the lower and upper bound constraints by simply setting the flow through each (u_i, v_j) to $A[i, j]$, with the flow through the remaining edges implied by this assignment. Hence the circulation problem is feasible. The circulation integrality theorem says that if an instance of the circulation problem has integral lower and upper bounds for each edge, then the problem is feasible iff there exists a circulation with integer flows. The proof is constructive, hence applying the circulation algorithm yields the desired rounding.

Problem 21.25, *pg. 170*: *What follows after the explorer visits the Isle of Logic? The explorer seems to have added no new knowledge since each inhabitant already knows that some inhabitants have blue eyes and some have green eyes. Why does his observation change the equilibrium?*

Solution 21.25: Consider the case where exactly one person has green eyes. The statement from the explorer would make it clear to the person with green eyes that he has green eyes since nobody else that he sees has green eyes.

Now, suppose two inhabitants have green eyes. The first day, each of these two inhabitants would see exactly one other person with green eyes. Each would see the other person on the second day too, from which they could infer that there must be two inhabitants with green eyes, the second one being themselves. Hence both of them would leave the second day.

Using induction, we can demonstrate that if there are k inhabitants with green eyes, all the green-eyed inhabitants would leave after the k-th assembly.

Proof:

We already saw the base case, $k = 1$. Suppose the induction hypothesis holds for $k - 1$. If the number of inhabitants with green eyes is k, each inhabitant with green eyes would see $k-1$ other inhabitants with green eyes. If at the k-th assembly,

they see that nobody has departed, it would indicate that they themselves have green eyes and hence all the green-eyed inhabitants would leave on the k-th day.

As for the second part of the question, for $k = 1$, it is fairly obvious that the explorer gave new knowledge to the person with green eyes. For other cases, the new information is more subtle. For $k = 2$, the green-eyed inhabitants would be able to infer the color of their eyes on the second day based on the information that everyone on the island knows that there are green-eyed inhabitants and yet no one left. For $k = 3$, they are able to infer because everyone knows that everyone knows that there are green-eyed inhabitants and yet on the second day no one left.

Suppose x is some fact and $E(x)$ represents the fact that everyone knows x to be true. In this case, let g represent the fact that there are some green-eyed inhabitants on the island. Then on the k-th day, all the green-eyed inhabitants would use $E^k(g)$, the k-iteration of the common knowledge, to infer that they have green eyes. Essentially, what the explorer did by announcing the fact in the assembly is that it became "common knowledge", i.e., $E^\infty(g)$ became true.

Problem 21.26, *pg. 170*: *Given a payoff matrix, compute values $p_0, p_1, \ldots, p_{m-1}$ for Player 1 that minimize the maximum payoff for Player 2. Assume Player 2 knows $p_0, p_1, \ldots, p_{m-1}$.*

Solution 21.26: Let $p = [p_0, p_1, \ldots, p_{m-1}]^T$ and $q = [q_0, q_1, \ldots, q_{n-1}]^T$. Denote $A[i][j]$ by $a_{i,j}$. Formally, we want to find a p that minimizes $\max_q p^T A q$, subject to the constraints that probabilities are nonnegative, and sum up to 1. When Player 2 knows $p_0, p_1, \ldots, p_{m-1}$, there always exists a q maximizing $p^T A q$ that is a vector of all 0s except for a single 1. The 1 is at an index corresponding to the maximum entry in $p^T A$. (There may be other qs which also maximize, but there is no loss of optimality in focusing on qs of the specified form.) Let e_i denote the n-dimensional vector which is 1 an index i, and 0 everywhere else.

The problem now is to find p that minimizes $\max_{0 \le i \le n-1} p^T A e_i$. Modeling this optimization problem as a linear program requires a trick to deal with the max operation. The basic idea is that the minimization problem $\min(\max_{0 \le i \le n-1}\{x_0, x_1, \ldots, x_{n-1}\})$ subject to the constraint $C(x_0, x_1, \ldots, x_{n-1})$ can be cast as a minimization problem which does not use the max operator. Specifically, this minimization problem is equivalent to minimizing t subject to $C(x_0, x_1, \ldots, x_{n-1})$ and $t \ge x_i$, for $0 \le i \le n-1$. Therefore the p we seek is the minimizer of t subject to the following constraints:

$$\begin{aligned} t &\ge p^T A e_i = \sum_{j=0}^{m-1} p_j a_{j,i}, \quad \text{for all } i,\ 0 \le i \le n-1 \\ \sum_{i=0}^{m-1} p_i &= 1 \\ p_i &\ge 0, \quad \text{for all } i,\ 0 \le i \le m-1 \end{aligned}$$

Now this optimization problem is an instance of linear programming, and can be solved using standard algorithms, e.g., simplex.

Variant 21.26.1: How would you efficiently solve for the optimum p subject to the additional constraint that the sum of the k largest values of p is no more than μ? (Adding $\binom{n}{k}$ additional constraints is not efficient.)

Variant 21.26.2: Given a subset of $\mathcal{R}^n$ defined by a set of linear inequalities $Ax \leq b$, find the radius of a largest sphere contained in the subset.

Variant 21.26.3: Suppose $C \in \mathcal{R}^{n \times n}$ and $\rho \in \mathcal{R}$. Let $\mathcal{F} = \{F \in \mathcal{R}^{n \times n} \mid F[i][j] \in [C[i][j] - \rho, C[i][j] + \rho]\}$. The constraints $Ax \leq b$ for all $A \in \mathcal{F}$ define a subset of $\mathcal{R}^n$. Find the radius of a largest sphere contained in this subset.

Part IV

Notation and Index

Notation

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

— E. S. PARKER

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

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

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

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

Index of Terms

Acknowledgments

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

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

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

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

ADNAN AZIZ — *Austin, Texas*
AMIT PRAKASH — *Belmont, California*
TSUNG-HSIEN LEE — *Mountain View, California*
October, 2012