Bitmask review

• Go over exercises 1-3
Bitmask review

- reverseBytes(x)
  - return \( (x \& 0x000000FF) \ll 24 \)
    | \( (x \& 0x0000FF00) \ll 8 \)
    | \( (x \& 0x00FF0000) \gg 8 \)
    | \( (x \& 0xFF000000) \ggg 24 \)
  - ...or Integer.reverseBytes(x) in Java
Bitmask review

• Count subsets of size at least M

  for (int mask = 0; mask < (1 << N); mask++) {
    int x = mask; int bitCount = 0;
    while (x > 0) {
      if ((x & 1) != 0) bitCount++;
      x >>= 1;
    }
    if (bitCount >= M) subsetCount++;
  }
Bitmask review

• Count subsets of size at least M
  • Or sum the binomial coefficients
    • \((N \text{ choose } M) + (N \text{ choose } (M+1)) + \ldots + (N \text{ choose } N)\)
Bitmask review

• Bit functions
What's that value?

- $2^{10}$
  - 1024 (about a thousand)
- $2^{20}$
  - 1048576 (about a million)
- 10!
  - 3628800 (about 3 million)
What's that value?

- Maximum signed integer value?
  - 2147483647, or Integer.MAX_VALUE

- How many subsets of $S$ where $|S| = N$?
  - $2^N$
Searching

• What we'll look at today:
  • Iterative: Loops, Permutations, and Subsets
  • Recursive backtracking
• Go over exercises 4-7
Searching with loops

• “Perfect square” problem
  • Math solution
    • Take the square root, determine if it is an integer
    • Is that easy?
  • Complete search solution
    • Compute the square of all numbers up to...
      • … sqrt(N) – do not need to go any further
Searching with loops

- “Perfect square” problem
- Binary search solution
Searching with loops

• “Use all digits” problem:
  • Find all pairs of 5-digit numbers that between them use the digits 0 through 9 once such that $abcde / fghij = N$
    • $2 \leq N \leq 79$
    • Each letter represents a different digit
Searching with loops

• “Use all digits” solution:
  • Rewrite the equation: \( N \times abcde = fghij \)
    • For \( X = abcde \), try all values of \( X \) between 01234 and 98765
      • Ensure the digits of \( abcde \) are unique
    • Check if \( Y = fghij \) is 5 digits and is comprise of unique digits
      • Don't forget to prepend the zero for 4-digit numbers
Searching with loops

“Use all digits” solution:

- Lots of pruning is possible
- We could approach this by trying all permutations
  - Motivation for the next problem
Searching with permutations

• “Movie seating” problem:
  • $n$ friends go to a movie and sit in a row with $n$ consecutive open seats.
  • There are $m$ seating constraints, i.e., two people $a$ and $b$ must be at most (least) $c$ seats apart
  • $0 < n \leq 8$ and $0 \leq m \leq 20$
Searching with permutations

• “Movie seating” solution:
  • Most important piece of information in this problem are the constraints
    • Up to 8 friends
    • Up to 20 seating constraints
  • Brute force / complete search is possible
    • Try all permutations and count the valid ones
      • The hard part is implementing it (in Java)
      • In C++, just use the next_permutation() function in the algorithm library
int N, validCount;
int permutation[] = new int[8]; // up to 8 friends
ArrayList<Constraint> constraints;

public static void main(String args[]) throws Exception {
    new SeatingConstraints().execute();
}

class SeatingConstraints {
    public void execute() throws Exception {
        N = 3; validCount = 0; constraints = new ArrayList<Constraint>();

        // 0 and 1 must be at most 1 seat apart
        constraints.add(new Constraint(0, 1, 1));
        // 0 and 2 must be at least 2 seats apart
        constraints.add(new Constraint(0, 2, -2));

        Arrays.fill(permutation, -1); // initialize perm array
        findPermutation(0); // recursively compute permutations
        System.out.println(validCount);
    }
}

class Constraint {
    int firstSeat, secondSeat, distance;

    public Constraint(int firstSeat, int secondSeat, int distance) {
        this.firstSeat = firstSeat;
        this.secondSeat = secondSeat;
        this.distance = distance;
    }
}

// Constraints for seating arrangement

public void findPermutation(int depth) {
    if (depth == N) { // found a full permutation
        for (Constraint c : constraints) {
            if (c.isViolated(permutation)) {
                return; // Do not count invalid perms!
            }
        }
    }
    validCount++; // Add valid perm to count
    return;
}

for (int i = 0; i < N; i++) {
    if (permutation[i] == -1) {
        permutation[i] = depth;
        findPermutation(depth+1);
        permutation[i] = -1;
    }
}
}
class Constraint {
    int a, b, dist;

    public Constraint(int a, int b, int dist) {
        this.a = a; this.b = b; this.dist = dist;
    }

    public boolean isViolated(int[] permutation) {
        int permDist = Math.abs(permutation[a] - permutation[b]);

        if (dist > 0 && permDist > dist) {
            return true; // two people are sitting too far apart
        } else if (dist < 0 && permDist < -dist) {
            return true; // two people are sitting too close together
        } else {
            return false;
        }
    }
}
Searching with permutations

• “Movie seating” solution:
  • What was the runtime of my code?
    • $O(N^N \times M)$
      • $8^8 \times 20 = 335,544,320$ – a little uncomfortably close to the limit, does not include any overhead constants
    • This is called a recursive backtracking approach
      • Traverse down the recursion tree, reach a leaf node, then travel back up, finding new leaf nodes
Searching with permutations

- Iterative approach for finding next perms

- Algorithm
  - Find largest index $i$ such that $A[i-1] < A[i]$
  - Find largest index $j$ such that $j \geq i$ and $A[j] > A[i-1]$
  - Swap $A[j]$ and $A[i-1]$
  - Reverse the suffix starting at $A[i]$

- Example
  - $A = [0, 1, 2, 5, 3, 3, 0]$
Searching with permutations

• Iterative approach for finding next perms
  • Runtime
    • It takes $O(N)$ operations to find a next perm
    • There are $N!$ permutations of a list
    • So going through all permutations in this way costs $O(N! \times N)$ – better than previous!
  • Benefits
    • Iterative
    • Lexicographical ordering
    • Does not require distinct elements in the list
• Reference, including code
Searching with combinations

• “Water gates” problem:
  • A dam has $1 \leq n \leq 20$ water gates to let out water when necessary. Using each gate has a flow rate and damage cost when used.
  • Open the gates so that a total flow rate is achieved at minimal total damage cost.
Searching with combinations

• “Water gates” solution:
  • Generate all subsets of the water gates
    • If the flow rate of the subset is more than $F$, consider it as a solution
  • How do you generate all subsets?
    • Bitmasks!
  • How many subsets are there?
    • $2^N$
  • What is the runtime of this solution?
    • $O(N \times 2^N)$
public static void main(String[] args) {
    int N = 4;
    int F = 10;

    int r[] = new int[] { 3, 2, 5, 7 }; // flow rates
    int c[] = new int[] { 4, 3, 4, 8 }; // cost of use
    int minimumCost = Integer.MAX_VALUE; // initialize with large value

    for (int mask = 0; mask < (1 << N); mask++) {
        int subsetCost = 0;
        int subsetFlow = 0;

        for (int i = 0; i < N; i++) {
            if ((mask & (1 << i)) != 0) {
                subsetCost += c[i];
                subsetFlow += r[i];
            }
        }

        if (subsetFlow >= F) {
            minimumCost = Math.min(minimumCost, subsetCost);
        }
    }

    System.out.println(minimumCost);
}
8 Queens Problem

- Problem:
  - Place 8 queens on an 8x8 chessboard and count the number of solutions
    - No queens are allowed to attack each other
8 Queens Problem

Naive solution:

- $8 \times 8 = 64$ cells, choose 8 of them and test
  - Could do this with recursive backtracking
  - But... $64 \text{ choose } 8 \approx 4 \text{ billion} = \text{ too much}$
- Complete search will not work
  - How to prune the search space?
8 Queens Problem

- Pruning the search space:
  - Two queens cannot be in the same column
  - So place one queen in each column
    - Represent this as an array of digits 0-7
      - The index of the digit is the column, the digit is the row.
    - $8^8 \sim 17$ million = better
8 Queens Problem

• Pruning the search space:
  
  • Two queens cannot be in the same row  
    • So each value in the array is unique  
    • This is now reduced to complete search over all permutations of digits 0-7  
      • $8! = 40,320 = \text{good}$
8 Queens Problem

• Pruning the search space:
  • Two queens cannot be on the same diagonal
    • Solutions built with recursive backtracking can preemptively ignore placing queens on diagonals
int queens[] = new int[8]; int a, b;

boolean isValid(int r, int c) {
    // Check previously placed queens
    for (int prev = 0; prev < c; prev++) {
        if (queens[prev] == r || (Math.abs(queens[prev] - r) == Math.abs(prev - c)))
            return false; // If here then previous queen attacks (r, c)
    }
    return true;
}

void backtrack(int c) {
    // For this column
    if (c == 8) {
        if (queens[b] == a) printSolution(queens);
        return;
    }

    for (int r = 0; r < 8; r++) {
        // Try all rows
        if (isValid(r, c)) {
            queens[c] = r; // Place a queen here
            backtrack(c + 1); // Recurse
        }
    }
}
8 Queens Problem

- Runtime of my code
  - $O(N^N)$, so $8^8 \sim 17$ million
    - But with lots of pruning, so a low constant and overall much few operations
Extra problems

- Vito's Family
- Lotto
- Citizen attention offices
- Blocks
- Marcus
- Small Factors
From this lecture

• Readings:
  • Sections 3.1 and 3.2