• **Homework 1**
  • Due tomorrow, Friday at 2pm

• **Homework 2**
  • Will be released tomorrow, Friday at 8pm and due following Friday at 2pm.

• **Midterm: Tuesday, March 8, 2016**
  • 15% of your grade
  • In class
  • There will be no homework on March 11
• Today we’ll look at: iterative loops, permutations, subsets, recursive backtracking
• For the following exercises, describe your approach to solving the problem and give its runtime
Exercise (Perfect square problem) Given an integer \( N \), determine if \( N \) is a perfect square, i.e., if \( \exists x \) such that \( x^2 = N \) and \( x \) is an integer. Constraint: \( 0 < N < 10^6 \)
• Math solution: take the square root, determine if it’s an integer
• Search solution: compute the square of numbers up to sqrt(N)
  • Binary search
Exercise  (Use all digits problem) Given a number $N$, find all pairs of 5-digit numbers that between them use the digits 0 through 9 exactly once and such that $\text{abcde}/\text{fghij} = N$ (each letter represents a unique digit $[0-9]$). Constraint: $2 < N < 79$
Solution 1: Rewrite $N \times abcde = fghij$
  - Check all $abcde$ (100k values) and check that $abcde$ and $fghij$ are different digits
  - Don’t forget to pad with 0
Solution 2: Check all permutations of $abcde$
  - You could even run all permutations of 0123456789 and divide
Exercise  (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 sit at least (or at most) \(c\) seats apart. How many possible seating configurations are there? Constraints: \(0 < N \leq 8\) and \(0 \leq M \leq 20\)

*Input:* Each case begins with a line containing two integers \(N\) and \(M\), followed by \(M\) lines containing a seating constraint, \(a, b, c\). If \(c\) is positive, \(a\) and \(b\) must be at most \(c\) seats apart. Otherwise, if \(c\) is negative, \(a\) and \(b\) must be at least \(-c\) seats apart.

<table>
<thead>
<tr>
<th>Sample input</th>
<th>Sample output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 0</td>
<td>2</td>
</tr>
<tr>
<td>3 1</td>
<td>4</td>
</tr>
<tr>
<td>0 1 1</td>
<td>8</td>
</tr>
<tr>
<td>4 2</td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>0 2 -2</td>
<td></td>
</tr>
</tbody>
</table>
• Important constraints:
  • Up to 8 friends
  • Up to 20 constraints
  • $8! \times 20$ checks --- brute force is possible!

• Solution:
  • Try all $8!$ permutations and check constraints
  • In C++, start with the lexicographically smallest (i.e., sorted) permutation and use `next_permutation`
  • In Java, you need to implement this from scratch

• How do you loop through all permutations?
  • Recursively
  • Iteratively
int permutation[] = new int[8];
bool seen[] = new bool[8];

public void findPermutation(int depth) {
    if (depth == N) {
        // found a full permutation
        return;
    }
    for (int i = 0; i < N; i++) {
        if (permutation[i] == -1 && !seen[i]) {
            permutation[i] = depth; seen[i] = true;
            findPermutation(depth+1);
            permutation[i] = -1; seen[i] = false;
        }
    }
}
• **Intuitively:**
  • We want to find the next lexicographic ordering
    • If we think of a list as a number, we want the permutation of its digits which form the smallest number larger than the original

• **Algorithm:** Given list $L = [0, 1, 2, 5, 3, 3, 0]$
  1. Let $i = \|L\| - 1$ (i.e., index of last element). Loop:
  2. If $L[i-1] < L[i]$:
     1. Let $j = i - 1$
     2. Find largest index $k$ such that $L[j] \geq L[k]$ and swap $L[j]$ and $L[k]$
     3. Reverse $L[i..\text{end}]$
  Else if no more permutations when $L$ is sorted in descending order
• Running time: each shuffle is $O(n)$, and there are $n!$ shuffles to go through all permutations.
  • However, performing all $n!$ shuffles runs in $O(n!)$ time

• Benefits:
  • Iterative
  • Lexicographic ordering
  • Allows repeated items in the list

• Reference and code
Exercise (Water gates problem) A dam has $N$ water gates to release water from the dam’s reservoir if the water level is getting too high. You are given each gate’s flow rate and cost of use. Determine the minimum total cost of use for a total flow rate of at least $F$. Constraint: $2 \leq N \leq 20$

Input: Each case begins with a line containing two integers $N$ and $F$, the number of gates and the total flow rate to achieve. Following are $N$ lines describing a gate which contains a pair of integers $r$ and $c$, the flow rate and cost of use for the gate. The sum of $r$ and the sum of $c$ will be less than $10^9$.

<table>
<thead>
<tr>
<th>Sample input</th>
<th>Sample output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 5</td>
<td>5</td>
</tr>
<tr>
<td>5 5</td>
<td>11</td>
</tr>
<tr>
<td>5 8</td>
<td></td>
</tr>
<tr>
<td>3 5</td>
<td></td>
</tr>
<tr>
<td>2 4</td>
<td></td>
</tr>
<tr>
<td>3 7</td>
<td></td>
</tr>
<tr>
<td>5 12</td>
<td></td>
</tr>
</tbody>
</table>
“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?
  - $2^N$
We’ve gone over code for:
• Permutations
• All subsets

What about combinations?
// calculates the next integer with the same number of 1-bits
unsigned int snoob(unsigned int x)
{
    unsigned int smallest, ripple, ones;
    // x = xxx0 1111 0000
    smallest = x & -x;  // 0000 0001 0000
    ripple = x + smallest;  // xxx1 0000 0000
    ones = x ^ ripple;  // 0001 1111 0000
    ones = (ones >> 2) / smallest;  // 0000 0000 0111
    return ripple | ones;  // xxx1 0000 0111
}
• Place 8 queens on an 8x8 chessboard and count the number of solutions
• No queens are allowed to attack each other
Naïve solution:
• Enumerate all possible positions on the chessboard and simulate.
• Problem? $64 \text{ choose } 8 \approx 4B$

Can you prune the search space?
• Two queens cannot be in the same column
  • 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

• Two queens cannot be in the same row
  • So each value in the array is unique
  • Now we have reduced the complete search to be over permutations of digits 0 to 7
    • $8! = 40320$

• Two queens cannot be in the same diagonal
  • If you permute using recursive backtracking, you can preemptively cut out
Let L denote an array of integers, and let f have be a function with prototype void f(int[] arr).

Exercise: How can you further reduce the search space for the 8 Queens problem?

Exercise: Give recursive code that will process every subset of L and count how many have sum k.
// Call initially with parameters (L, 0, 0)
static int countSubsets(int[] L, int idx, int sum, int k)
{
    if (idx == L.length) return sum == k ? 1 : 0;
    return countSubsets(L, idx+1, sum, k)
        + countSubsets(L, idx+1, sum+L[idx], k);
Let L denote an array of integers, and let f have be a function with prototype void f(int[] arr).

**Exercise:** Give recursive code that will call the function f one time for each permutation of L (can assume entries are distinct).
Let $L$ denote an array of integers, and let $f$ have be a function with prototype void $f$([int] arr).

**Exercise:** Give recursive code that will call the function $f$ one time for each permutation of $L$ (can assume entries are distinct).

**Exercise:** Give iterative code for calling $f$ once for each permutation of $L$ in lexicographic order.
Let L denote an array of integers, and let f have be a function with prototype void f(int[] arr).

**Exercise:** Give recursive code that will call f once for each subsequence of L of length 4.

**Exercise:** Give iterative code that will call f once for each subsequence of L of length 4.
To determine whether complete search will solve your problem, address the following:

- How can I traverse the search space?
- **How long with the complete search take?**
- How can I implement the complete search?
To determine whether a greedy algorithm will solve your problem, address the following:

- What is my greedy choice function?
- How long will the greedy algorithm take?
- Is it correct?
- How can I implement my greedy algorithm?
You are given a list of tasks, with task $i$ requiring time interval $[a_i, b_i)$, and the requirement that no two tasks can be scheduled at the same time. Give an algorithm to compute the largest number of tasks that can be completed.
Sort tasks in ascending order by $b_i$ breaking ties by ascending $a_i$-values. Always choose the earlier task in the sorted order that is feasible. Now we must show this is correct. Let $P(t)$ denote the maximum number of tasks that can be assigned using times in $[t, \infty)$. Note that $P(s) \geq P(t)$ if $s \leq t$. Suppose we are choosing between $[a_i, b_i)$ and $[a_j, b_j)$ as our earlier task with $b_i \leq b_j$. Then $P(b_i) \geq P(b_j)$ making the first interval at least as good.
You are given a list of objects, with object $i$ having weight $w_i$ and value $v_i$. Assuming you can take fractional amounts of each object, and your sack has capacity $C$, compute the largest value you can carry.
Sort tasks in descending order by $v_i/w_i$, and then use objects in order taking as much as possible of each. To show this is optimal, suppose $v_i/w_i \geq v_j/w_j$, you have taken a positive amount of object $j$, and you have not taken all of object $i$. Then trading object $j$ for object $i$ cannot disimprove the total value.
You are given a list of tasks, with task $i$ having duration $d_i$ in days. For each day before task $i$ is started, a cost $s_i$ is paid. Choose an ordering of the tasks that minimizes the total cost paid.
Sort tasks in descending order by $s_j/d_j$. To see this is optimal, suppose you have an ordering of the tasks with task $i$ immediately preceding task $j$. The change in value of swapping these tasks is $s_id_j - sjd_i$. This is non-negative precisely when $s_i/d_i \leq s_j/d_j$. 

Give a greedy algorithm for computing the minimum/maximum spanning tree
We do the minimum case. Sort the edges by weight. Only add an edge if it doesn’t create a cycle in the forest of edges already added. Suppose this algorithm is not optimal on some input. Then there must be some first edge $e$ added such that the resulting forest cannot be completed into a minimum spanning tree, but there is a MST $T$ that can be constructed from the edges added before $e$. Then adding $e$ to $T$ creates a cycle, where at least one edge in the cycle has value greater than or equal to $e$ (as adding $e$ didn’t create a cycle in our algorithm). Adding $e$ and removing this larger valued edge creates a tree that is at least as good. Implementation details will be discussed when we cover graph algorithms.
Competitive Programming 3.1, 3.2