Hackathon 3 (Feb 9, 2022)

General Instructions:

Problem 1

Solution 1:

There are at least two ways to compute m^k in time $O(\log k)$. Both of them involve repeated squaring. Here is one possible solution that uses a neat recursion:

powerRecursive(m, k)

- If $k = 1$, then return m
- Else if k is odd, then return $m \times$ power(m, $k - 1$)
- Else
  - $p \leftarrow$ power(m, $k/2$)
  - return p^2

Running time:

Informally, k gets halved repeatedly, in at most two recursive calls each time.

More formally, let $t(k)$ be the running time of powerRecursive(m, k). Induction claim on parameter k: $t(k) \leq 3\log k + 3$. Base case, $t(1) = 1$.

For k even, we have:

$$\begin{aligned} t(k) &= 2 + t(k/2) \\ &\leq 2 + 3\log(k/2) + 3 \text{ (from induction hypothesis)} \\ &= 2 + 3\log k \\ &< 3\log k + 3 \end{aligned}$$

For k odd, we have:

$$\begin{aligned} t(k) &\leq 1 + t(k-1) \\ &\leq 1 + (2 + t(k/2)) \\ &\leq 3 + t(k/2) \\ &\leq 3 + 3\log(k/2) + 3 \text{ (from induction hypothesis)} \\ &= 3 + 3\log k \end{aligned}$$

Solution 2:

Another way to compute m^k is to look at the binary representation of k to obtain a decomposition of k into a sum of powers of 2. For example, $k = 20$ has binary representation 10100 since $20 = 16 + 4$. So compute m^4 and m^{16} by repeated squaring. Return the product of these two. Here is the pseudocode:

powerIter(m, k)

- result $\leftarrow$ 1
- power $\leftarrow$ m
- While $k > 0$ do the following:
  - If k is odd, then result $\leftarrow$ result $\times$ power

    - $k \leftarrow k/2$ (Integer division! Truncates least significant bit.)
    - power $\leftarrow$ power $\times$ power
- return result

Caution! The variable *power* in the above pseudocode will end up storing a value larger than m^k. To avoid this potential overflow, give an `if` condition before squaring *power* to check if $k > 0$.

Running Time: Trivial. Left as exercise.

Problem 2

- Use `getline` for input since we do not know the length of input numbers beforehand.
- You might want to write a function very similar to `strcmp` that can tell you which of the numbers is larger.
- Then, for the algorithm, use high-school addition and subtraction.
- To store the result, observe that the sum or difference, of two numbers with n_1 and n_2 digits respectively, has at most $\max\{n_1, n_2\} + 1$ many digits. So create an array with this length to accommodate the result.

  If you created an `int` array, you will have to print it with a loop.

  Quicker - create a char array and insert `\0` after the last digit. Then print it with a standard printf %s. To avoid leading 0s, point printf to the first non-zero element in your array instead of base index.

Problem 3

This problem is very similar to the permutations problem which was solved in a recent lab session.

Let n be the input number. For each $i \in \{1, 2, \ldots, n\}$, print i followed by every possible way of writing $n - i$ as a sum of natural numbers, each of which is at least i.

A more detailed pseudocode:

printSums(n, prefix[], currentIndex, limit)

- If $n = 0$, print elements in prefix from index 1 to currentIndex
- else
    - For $i \leftarrow$ limit to n do:
        - prefix[currentIndex] $\leftarrow$ i;
        - printSums(n-i, prefix, currentIndex+1, i)
        - $i \leftarrow i + 1$

Call printSums with n, empty prefix array, currentIndex = 1, and limit = 1 to get an output that matches the problem statement.