CMPUT 403: Combinatorics and Arithmetic

Zachary Friggstad

March 4, 2016

Fast Exponentiation

Given integers a, b with $b \geq 0$, compute a^b *exactly*.

Naively, takes $b - 1$ multiplications. There is a way using only $O(\log b)$ multiplications.

Note, most applications of this have you take the answer modulo some value m so the numbers don't get too big. This is an essential subroutine in many ciphers, including RSA and Diffie-Hellman.

If $b = 2^k$ then just use repeated squaring:

$$a \to a^2 \to (a^2)^2 = a^4 \to a^8 \to \ldots \to a^{2^i} \to \ldots \to a^{2^k}.$$

Takes $k = \log_2 b$ multiplications.

In general, write $b = \sum_{i=0}^{k} c_i \cdot 2^i$ where $c_i \in \{0, 1\}$. Use repeated squaring to iteratively compute a^{2^i}. If $c_i = 1$ then multiply a^{2^i} into the answer.

```
//compute a^b mod m
int fmodexp(int a, int b, int m) {
   int ans = 1, pow2 = a; //pow2 = a^{2^i} after i iterations
   while (b) {
      if (b & 1) ans = (ans * pow2) % m; //if c_i == 1
      pow2 = (pow2*pow2) % m;
      b >>= 1;
   }
   return ans;
}
```

Linear Recurrences (yawn)

Recall the Fibonacci numbers $0, 1, 1, 2, 3, 5, 8, \ldots$.

$$fib(n) = \begin{cases} n & \text{if } n \in \{0, 1\} \\ fib(n-1) + fib(n-2) & \text{if } n \geq 2 \end{cases}$$

This is a linear recurrence. Apart from some base cases, the recurrence is just a linear combination of some previous terms.

Another example:

$$g(n) = \begin{cases} 1 & \text{if } n = 0 \\ -3 & \text{if } n = 1 \\ 7 & \text{if } n = 2 \\ 2f(n-1) - 3 \cdot f(n-3) & \text{if } n \geq 3 \end{cases}$$

Compute the n'th value of a recurrence!

Easy, just compute then in increasing order and store the results in a table. Takes $O(k \cdot n)$ time where k is the *order* of the recurrence (i.e. number of terms it reaches back).

We can do it much faster. Look!

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} fib(n) \\ fib(n+1) \end{pmatrix} = \begin{pmatrix} fib(n+1) \\ fib(n+2) \end{pmatrix}$$

Therefore,

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} fib(n) \\ fib(n+1) \end{pmatrix}$$

Use fast exponentiation to compute $fib(n)$ in $O(\log n)$ arithmetic operations!

Recall this recurrence.

$$g(n) = \begin{cases} 1 & \text{if } n = 0 \\ -3 & \text{if } n = 1 \\ 7 & \text{if } n = 2 \\ 2f(n-1) - 3 \cdot f(n-3) & \text{if } n \geq 3 \end{cases}$$

Then

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -3 & 0 & 2 \end{pmatrix}^n \cdot \begin{pmatrix} 1 \\ -3 \\ 7 \end{pmatrix} = \begin{pmatrix} g(n) \\ g(n+1) \\ g(n+2) \end{pmatrix}$$

Just use fast exponentiation again! Note, if they want the answer mod m, then make sure to liberally take mods throughout the computation.

In general, number of arithmetic operations is $O(k^3 \log n)$ where k is the order of the recurrence. For *fib*, the order is 2 and for g it is 3.

Permutations

A permutation of a set is just a rearrangement of it. Let's just talk about permutations of $\{0, \ldots, n-1\}$.

One way to express a permutation π:

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 4 & 1 & 6 & 7 & 5 & 0 & 2 \end{pmatrix}$$

Read this like "0 goes to 3" and "1 goes to 4", etc.

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 4 & 1 & 6 & 7 & 5 & 0 & 2 \end{pmatrix}$$

Can break a permutation down into cycles. Track the trajectory of an item as it repeatedly gets permuted.

$$0 \to 3 \to 6 \to 0$$

$$1 \to 4 \to 7 \to 2 \to 1$$

$$5 \to 5$$

Write this compactly as

$$\pi = (0\ 3\ 6) \cdot (1\ 4\ 7\ 2) \cdot (5)$$

$$\pi = (0\ 3\ 6) \cdot (1\ 4\ 7\ 2) \cdot (5)$$

Permutations are naturally represented as an array

```
//initialization like this possible in c++11
vector<int> pi = {3, 4, 1, 6, 7, 5, 0, 2};
```

Can find all cycles in $O(n)$ time.

```
vector<bool> seen(n, false);
vector<vector<int>> cycles;
for (auto x : pi) {
  if (seen[x]) continue;
  vector<int> cyc;
  while (!seen[x]) {
    cyc.push_back(x);
    seen[x] = true;
    x = perm[x];
  }
}
```

Wisdom from the Streets
When solving a problem involving a permutation, looking at the cycle decomposition may help!

Problem
Given a permutation π, how many times do you have to apply π before everything ends up back in its original position?

Example
Suppose you can shuffle a deck in exactly the same way each time you shuffle. How many times to you have to apply this shuffling until the deck returns to its original arrangement?

If π is just a cycle of length n, it takes n steps.

e.g. $\pi = (0\ 1\ 2\ 3\ 4\ 5)$. After 6 applications of π, every item returns back to its start location.

If π is the product of a bunch of cycles, the answer is the smallest integer m that is a multiple of all cycle lengths: the least-common multiple of all cycle lengths!

e.g. $\pi = (0\ 3\ 6) \cdot (1\ 4\ 7\ 2) \cdot (5)$.

Takes 12 applications to get every item back to its starting position.

Suppose π, σ are permutations. Get a new permutation, denoted $\pi \circ \sigma$, by first permuting according to σ and then according to π.

Example:

- $\pi = (0\ 1\ 4) \cdot (2\ 3)$
- $\sigma = (0) \cdot (1\ 2\ 4\ 3)$
- $\pi \circ \sigma = (0\ 1\ 3\ 4\ 2)$

```
vector<int> pi, sigma, comp;
for (int i = 0; i < n; ++i)
  comp[i] = pi[sigma[i]];
```

The $\circ$ operation on permutations is associative, so we can also use fast exponentiation!

i.e. computing π^k takes $O(k \log n)$ time where n is the number of items being permuted.

Finally, every permutation has an inverse.

Example

- $\pi = (0\ 1\ 4) \cdot (2\ 3)$
- $\pi^{-1} = (4\ 1\ 0) \cdot (3\ 2)$
- $\pi \circ \pi^{-1} = (0) \cdot (1) \cdot (2) \cdot (3) \cdot (4)$

```
vector<int> pi, inv;
for (int i = 0; i < n; ++i)
    inv[pi[i]] = i;
```

Finally, good old counting.

\# of permutations of n items $= n! = n \cdot (n-1) \cdot \ldots \cdot 1$.

\# of size-k subsets of a size-n set $= \binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$.

Also,

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{otherwise} \end{cases}$$

Some problems need you to calculate these binomial coefficients, usually best to build up a table using dynamic programming.