Mathematics

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

Table of Contents

- **Problem:** Calculate a^n quickly (say $n = 10^{18}$).
- Key is to use a kind of divide and conquer. Runs in $O(\log n)$ time (assuming constant time multiplication)
- Observe that $a^n = a^{n/2} \times a^{n/2}$ for even n and $a^n = a^{n/2} \times a^{n/2} \times a$ for odd n
- This is equivalent to precomputing each a^{2^m}, and combining powers according to the binary expansion of n

- Example:

$$a^9 = a^4 \times a^4 \times a$$
$$a^4 = a^2 \times a^2$$
$$a^2 = a \times a$$

- **Implementation**

```
typedef long long ll;

ll pow(ll x, ll k) {
  if (k == 0) return 1;

  ll a = pow(x, k/2);
  a = a*a;
  if (k%2 == 1) a = a*x;
  return a;
}
```

- **Complexity?** $O(\log n)$ assuming constant time multiplication.
- This is a very general idea. More generally, for a large class of functions this allows us to compute $f^{(n)}(x)$ with $O(\log n)$ overhead.
- This generalisation shows up in many graph theory, dp and math problems.
- Most useful example is probably when f is multiplication by a matrix.
- Also compare to LCA code.

- $a \equiv b \bmod m$ iff there exists some integer k such that $a = b + mk$
- $b \bmod m$ is the remainder of b when divided by m
- In C/C++, the % symbol is used for the modulo operator
- BUT: Be careful with negative numbers! In C/C++, the behaviour is:
  - $(-4)\%3 == (-4)\%(-3) == -1$
  - $(-4)\%5 == (-4)\%(-5) == -4$
  - $4\%(-3) == 1$
  - $4\%(-5) == 4$
- Technically % is not mod when negative numbers are involved.
- You can get actual mod when m is positive by doing $((a\%m) + m)\%m$.

- If $a \equiv b \bmod m$, then $a + c \equiv b + c \bmod m$
- If $a \equiv b \bmod m$, then $ac \equiv bc \bmod m$
- If $ac \equiv bc \bmod mc$, then $a \equiv b \bmod m$
- $ac \equiv bc \bmod m$ does not necessarily mean $a \equiv b \bmod m$!

- For general m, we get addition, subtraction, multiplication but not necessarily division. (e.g: $2 \cdot 1 = 2 \cdot 4 = 2 \pmod 6$). This makes numbers mod m a ring.
- However, if m is a prime then we actually do get division! (this is equivalently to saying that multiplication by non zero numbers is a bijection)
- Let p be a prime. This makes numbers mod p a *field*. Essentially, this says you have addition, subtraction, multiplication and division. Hence for the most part it's just like working in the rationals.
- How to do division by $a \mod p$ is not immediately obvious. We do it by finding a^{-1}.

- The inverse a^{-1} of a is an integer such that $a^{-1}a \equiv aa^{-1} \equiv 1 \bmod m$
- Only exists if $\gcd(a, m) = 1$. Usually m is a prime which is nice.
- **Fermat's little theorem** $a^{m-1} \equiv 1 \bmod m$ for prime m
- Hence $a^{-1} = a^{m-2} \bmod m$.
- Euler's theorem is a generalisation that works for general modulus, based on the *totient function*[1], $\phi(n)$
- Can also be computed with the Extended Euclidean algorithm for general modulus[2]
- Either way, $O(\log M)$.

[1] counts the numbers less than n which are coprime to n

[2] any modulus which is coprime to a

- Knowing these basics is useful because many combinatorics problems have answers that far exceed a long long. Hence you will usually be asked to output the answer $\bmod M$ where generally M is a prime.
- The 2 popular primes for competitions are $1,000,000,007$ and $1,000,000,009$.

- **OVERFLOWS!** Especially when M is around 1 billion and you have multiplications. Every operation you do should be modded after. One nicer way is to add helper functions `add_mod` and `multiply_mod`.
- Depending on how careful you are, you may want to mod the arguments to `multiply_mod` too.
- **Negatives** If you have subtractions, you should usually write your own `sub_mod` function that does $(((a - b)\%M) + M)\%M$.
- Mod is slow compared to add and subtract. If you are doing a lot of `add_mod` and `sub_mod` you might need to optimize down how many mods you do.

- A prime number (or a prime) is a natural number greater than 1 that has no positive divisors other than 1 and itself
- Primes are the fundamental building blocks of all of number theory
- Problems involving factorization or multiplication often reduce to looking at the prime factorization.
- Algorithm wise, the fundamental algorithms are primality testing, prime factorization and finding all factors of a number.
- Depending on the problem we will either want this for a specific number (but possibly very big) or for all numbers (up to a smaller bound).

- **Algorithm** For every possible factor F of N, check if it divides. Observing that if an $F_1 > \sqrt{N}$ is a factor, then there must be another $F_2 < \sqrt{N}$ that is also a factor, we need only check factors $F \leq \sqrt{N}$.
- We can easily tweak this to also give all prime factors.
- **Complexity** $O(\sqrt{N})$ time

Implementation

```
#include <bits/stdc++.h>
using namespace std;

bool isprime(int x) {
  if (x < 2) return false;

  for (int f = 2; f*f <= x; f++) {
    if (x % f == 0) return false;
  }
  return true;
}

// Returns prime factors in increasing order with right multiplicity.
vector<int> primefactorize(int x) {
  vector<int> factors;
  for (int f = 2; f*f <= x; f++) {
    while (x % f == 0) {
      factors.push_back(f);
      x /= f;
    }
  }
  if (x != 1) factors.push_back(x);
  return factors;
}
```

Finding all primes/factorizations

- We use the Sieve of Erastothenes.
- **Algorithm** Starting with 2, mark all multiples of 2 as composite. Then, starting with the next smallest number not marked composite (which therefore must be prime), 3, mark out all its multiples and repeat until we hit the upper bound. Every unmarked item must be a prime. This can be trivially modified to also prime factorize all numbers.

```
bool marked[N+1];
vector<int> primefactorization[N+1];
for (int i = 2; i <= N; i++) {
  if (!marked[i]) {
    primefactorization[i].push_back(i);
    for (int j = 2*i; j <= N; j += i) {
      marked[j] = true;
      int tmp = j;
      while (tmp % i == 0) {
        primefactorization[j].push_back(i);
        tmp /= i;
      }
    }
  }
}
```

- **Complexity** For some upper bound N and each prime p, we must strike out about $\frac{N}{p}$ multiples, so the amount of work we do is roughly $\frac{N}{p_1} + \frac{N}{p_2} + \ldots + \frac{N}{p_P}$. This is clearly less than $N + \frac{N}{2} + \frac{N}{3} + \ldots + \frac{N}{N}$, so by harmonic series we can say that this algorithm runs in $O(N \log N)$ time.
- More precisely, it is known that $\sum_{p<N} \frac{1}{p} = O(\log \log N)$. So the running time is $O(N \log \log N)$.
- The algorithm itself can be optimised even more, though this is not usually necessary
  - Throw away the even numbers ($2\times$ speed up)
  - For each prime, start marking at its square because the smaller ones will be marked already
  - All primes that aren't 2 or 3 are congruent to 1 or 5 mod 6.

- Two choices. Either find all prime factors then recover the full factorization (exercise). Or do it directly.
- **Algorithm** For all numbers d from 1 to N, add d as a factor to all multiples of d up to N.
- **Complexity** $\frac{N}{1} + \frac{N}{2} + \ldots = O(N \log N)$.

Useful algorithms

- **Single Primality Testing:** $O(\sqrt{N})$ is easy. Miller Rabin is better and runs in $O(12 \cdot \log N)$ for $N < 2^{64}$.
- **Single Factorization:** Generally $O(\sqrt{N})$ is good enough. Pollard's rho runs in $O(N^{1/4})$, expected.
- **Testing all primes/prime factorizing up to N:** $O(N \log \log N)$ is good enough/optimal.
- **Factorization up to N:** $O(N \log N)$ is optimal.

- **Problem statement** Goldbach's conjecture states that every even integer greater than 2 can be expressed as the sum of two odd primes. For some even integer N, find a pair of odd primes that sums to N.
- **Input** A single integer N, $3 \leq N \leq 1,000,000$
- **Output** A line containing A and B, two odd primes that sum to N, or "Goldbach's conjecture is wrong" if no such numbers exist

- **Algorithm** We know that we have to do something with primes. So let's start by generating all primes up to N, using the sieve.
- After we generate our list of primes, the remaining problem is "given an integer, find a pair from this list that sums to the integer"

- We can solve this problem in $O(n)$ time using a set with a fast membership test
- Since our elements are all small integers, we can just use a boolean array
- **Complexity** To create our list of primes, we use our $O(n \log \log n)$ time, $O(n)$ space sieve to transform this into a simpler problem which we can solve in $O(n)$ time and $O(n)$ space. So this algorithm runs in $O(n \log \log n)$ time and $O(n)$ space.

Implementation

```
#include <cstdio>

const int N = 1e6+5;
int primes[N], P, notprime[N];

int main() {
  // sieve
  notprime[0] = 1;
  notprime[1] = 1;
  for (int i = 2; i < N; i++) {
    if (notprime[i]) continue;
    // if i is prime, mark all its multiples as not prime
    primes[P++] = i;
    for (int j = i*i; j < N; j += i) notprime[j] = true;
  }

  int n;
  while (scanf("%d", &n), n) {
    // scan primes[] for pair adding to n
    for (int i = 1; i < P; i++) {
      if (!notprime[n-primes[i]]) {
        printf("%d = %d + %d\n", n, primes[i], n - primes[i]);
        break;
      }
    }
  }
  return 0;
}
```

- $\gcd(a, b)$ is the greatest integer that divides both a and b
- One of the most commonly used tools in solving number theory problems
- A few useful facts
  - $\gcd(a, b) = \gcd(a, b - a)$
  - $\gcd(a, 0) = a$
  - $\gcd(a, b)$ is the smallest positive number in $\{ax + by : x, y \in \mathbb{Z}\}$

- Can be computed with the Euclidean algorithm, which is the repeated use of the first property above:

$$\gcd(a, b) = \gcd(a, b - a)$$

- Usually, you'll use a similar rule, $\gcd(a, b) = \gcd(a, b\%a)$
- This has a complexity of $O(\log(a + b))$ because if $a < b$ then $b\%a < \frac{b}{2}$ so a number halves each time.

- **Implementation**

```
int gcd(int a, int b) {
  return b ? gcd(b, a % b) : a;
}
```

- Some versions of <algorithm> have a __gcd function already defined, but it's not always clear when it's available, so usually it's safest to just write it yourself

- As with the Euclidean algorithm, we incrementally apply the $\gcd(a, b) = \gcd(a, b - a)$ rule until we've found the GCD, but we also explicitly write the intermediate numbers as integer combinations of a and b, i.e. we find x and y where

$$ax + by = \gcd(a, b),$$

  which is called *Bézout's identity*
- This is useful for solving linear equations. We can also use it to find modular inverse.
- The generalization is CRT, the Chinese Remainder Theorem. This allows you to find a x that solves a family of equations $\{a_i x \equiv b_i \pmod{m_i}\}_{i=1}^{N}$ quickly.

- **Implementation**

```
int gcd(int a, int b, int& x, int& y) {
  if (a == 0) {
    x = 0; y = 1;
    return b;
  }
  int x1, y1;
  int d = gcd(b % a, a, x1, y1);
  x = y1 - (b / a) * x1;
  y = x1;
  return d;
}
```

- This also gives you lcm in $O(\log n)$ since

$$\operatorname{lcm}(a, b) = \frac{ab}{\gcd(a, b)}$$

Table of Contents

- 2 key tools for solving combinatorics problems. DP and math. Often need a combination of both.
- We prefer DP whenever possible. In particular, DP helps when we have to build up the structures we are counting.
- Math is helpful for determining the right thing to count and for reducing complexity.

- The binomial coefficient $\binom{n}{k}$ is the number of ways to make an unordered selection of k elements out of a set of n distinguishable elements
- One of the most widely used tools in combinatorics

- **Algorithm 1** Compute directly from the formula

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

- **Complexity** $O(\min(k, n-k))$ to compute the factorials, although parts of this can be precomputed for repeated uses. The intermediate values can become very large, however this problem can be avoided by rearranging this formula in terms of alternating multiplication and division

- **Algorithm 2** Compute from the recurrence

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

- **Complexity** There are $O(nk)$ total values of $\binom{n}{k}$, and it takes $O(1)$ time to compute each value, so this takes $O(nk)$ time

- However, we actually rarely use either of the above approaches because generally you'll be working mod a prime P.
- And math mod P is nicer because we can't overflow and don't have precision issues!

- **Problem statement** Compute $\binom{n}{k} \bmod 1,000,000,007$
- **Input** Two integers N and K, $0 \leq K \leq N \leq 1,000,000$
- **Output** A line containing $\binom{n}{k}$

- **Algorithm** We can't use the recurrence here; $O(NK)$ is too slow when N and K are each up to 1,000,000
- The only viable method is using the formula

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

- We need to be able to divide in our modulus, i.e. compute inverses
- Luckily, 1,000,000,007 is a prime (what a crazy random happenstance!)

- We can solve the problem in $O(1)$ per query after computing factorials and their inverses, using Fermat's little theorem and fast exponentiation
- We precompute every factorial and its corresponding inverse, since there are only $O(N)$ of either of these.
- **Complexity** After $O(N \log MOD)$ precomputation, we can answer each query in $O(1)$ time.

Implementation

```
typedef long long ll;

ll f[N];
// modify earlier fast exponentiation algorithm to work modulo c
ll modpow(ll a, ll b, int c);

ll inv(ll x) {
  // By Fermat's little theorem, a^(p-2) is the inverse of a mod p
  return modpow(x, MOD-2, MOD);
}

int main() {
  // factorials
  f[0] = 1;
  for (int i = 1; i < N; i++) f[i] = (i * f[i-1]) % MOD;

  int T;
  scanf("%d", &T);
  for (int i = 0; i < T; i++) {
    int n, k;
    scanf("%d%d", &n, &k);
    printf("Case %d: ", i + 1);
    ll res = (f[n] * inv(f[n-k])) % MOD;
    res = (res * inv(f[k])) % MOD;
    printf("%lld\n", res);
  }
  return 0;
}
```

- Probability and expected value problems are usually just combinatorics problems where you have to divide by something.
- Surprisingly helpful fact: Expectations are linear: for random variables X and Y and a constant c,

$$\mathbb{E}(X + c) = \mathbb{E}(X) + c$$
$$\mathbb{E}(X + Y) = \mathbb{E}(X) + \mathbb{E}(Y)$$
$$\mathbb{E}(cX) = c\mathbb{E}(X)$$

- Fun riddle: N people each throw their hats into the air. A random hat lands on each person's head. What's the expected number of people that get their own hat back?
- A: 1. The probability that each individual person get their own hat back is $1/n$, hence by linearity of expectations, the expected number of people that gets their own hat back is $1/n \cdot n = 1$.
- This is a demonstration of a more general principle: to count something, often we should break it into smaller parts which we instead count.

- **Problem Statement:** Given a tree, I am going to pick 2 vertices at random (pick the first uniformly at random then the second. The same vertex may be picked twice) What is the expected length of the unique simple path between the 2 vertices?
- **Input Format:** First line 1 integer, V, the number of vertices. $1 \leq V \leq 100k$. Followed by $V - 1$ lines describing the edges in the tree, each as a pair.

- First it is worth noting that the number of choices is just V^2. So expected value is equal to

$$\frac{\text{(sum of lengths over all paths)}}{V^2}$$

  Ignoring the denominator, this is just a combinatorics problem.
- There are 2 ways to go about this. One is to approach it directly.
- Let u be the first vertex picked and v the second. Then it suffices to find the sum of path lengths of all paths with u as an endpoint. The answer is then the sum of this over all u. In other words,

$$\text{(sum of lengths over all paths)} = \sum_{u \in V(G)} \sum_{v \in V(G)} \text{path\_length}(u, v)$$

- Consider for now just the sum over all v in the subtree of u.

$$\sum_{u \in V(G)} \sum_{v \in subtree(u)} \text{path\_length}(u, v)$$

- Then this is just the sum of depths of the subtree at u. How to calculate this quickly for all u?
- Tree DP!

Example: Path Length on Tree


```cpp
#include <bits/stdc++.h>
using namespace std;

/* As usual, assume the root is vertex 0
 * and we've run our tree representation to obtain
 * vector<int> children[MAXV]
 */

const int MAXV = 100000;
vector<int> children[MAXV];

// Size of each subtree.
int subtreeSize[MAXV];
// Sum of depths of each subtree.
long long sumOfDepths[MAXV];

void calcSubtreeSums(int c = 0) {
    subtreeSize[c] = 1;
    for (int ch : children[c]) {
        calcSubtreeSums(ch);
        subtreeSize[c] += subtreeSize[ch];
        // Each depth in ch's subtree increases by 1
        // when we move from ch to c.
        sumOfDepths[c] += subtreeSize[ch] + sumOfDepths[ch];
    }
}
```

Example: Path Length on Tree

- Common nuisance: handling subtrees is easy, but the part above u is a pain. Usually doable but more technically involved.
- An alternative fix here is to instead count over the lca, not over u. Aka:

$$\begin{aligned}&\text{(sum of lengths over all paths)}\\ &= \sum_{l \in V(G)} \sum_{(u,v)|\mathrm{lca}(u,v)=l} \mathrm{path\_length}(u,v)\end{aligned}$$

- Not hard to implement, $lca(u,v) = l$ holds exactly when u and v are in the subtree of l but are not both in the same subtree of one of the children of l.
- Next slide has code but it isn't the main point of this example. Just for completeness sakes.

Example: Path Length on Tree


```
const int MAXV = 100000;
/* Assume we have also calculated these */
vector<int> children[MAXV];
// Size of each subtree.
int subtreeSize[MAXV];
// Sum of depths of each subtree.
long long sumOfDepths[MAXV];

// Multiply the answer by 2 afterwards, atm counts unordered pairs.
long long sumOfPathLengths(int l = 0) {
    long long sumPaths = 0;
    for (int ch : children[l]) {
        sumPaths += sumOfPathLengths(ch);
    }
    // 1 to make sure we count paths starting at l.
    long long numNodesSeen = 1;
    long long sumDepthsSoFar = 0;
    for (int ch : children[l]) {
        // Consider all paths from nodes seen to nodes in this subtree.
        sumPaths += sumDepthsSoFar * subtreeSize[ch];
        // Again, we add subtreeSize[ch] since
        // we need sum of depths relative to l, not ch.
        sumPaths += numNodesSeen * (sumOfDepths[ch] + subtreeSize[ch]);
        sumDepthsSoFar += sumOfDepths[ch] + subtreeSize[ch];
        numNodesSeen += subtreeSize[ch];
    }
    return sumPaths;
}
```

- Nicer solution: Break our sum down into smaller parts. What are our parts? Natural thing that paths break down into.
- Edges.
- For each edge, we will count the number of paths containing it. Then I claim:

$$\begin{aligned}&\text{(sum of lengths over all paths)}\\ &= \sum_{e \in E(G)} \text{num\_paths\_containing\_e}\end{aligned}$$

Example: Path Length on Tree

- Question: What is a formula for number of paths containing the edge $e : \mathbf{u} \to \mathbf{v}$. Suppose **u** is the parent of **v**.
- Answer: (Number of nodes outside the subtree of v) multiplied by (Number of nodes in the subtree of v).
- Using our earlier array names, this is just

```
(V - subtree_size[v]) * subtree_size[v]
```

Example: Path Length on Tree


```
const int MAXV = 100000;
int V;
/* Assume we have also calculated these */
vector<int> children[MAXV];
// Size of each subtree.
int subtreeSize[MAXV];

// Multiply the answer by 2 afterwards, atm counts unordered pairs.
long long sumOfPathLengths(int c = 0) {
    long long sumPaths = 0;
    for (int ch : children[c]) {
        sumPaths += sumOfPathLengths(ch);
        sumPaths += (V - subtreeSize[ch]) * subtreeSize[ch];
    }
    return sumPaths;
}
```

Example: Diverse Committee

- **Problem Statement:** You are trying to form a committee again. You have already selected N people for the committee. Each person now needs a role.
  There are K possible roles. For the ith role, the committee needs between l_i and u_i people with this role (inclusive). Each person needs to be assigned exactly one role. How many ways are there to assign the roles? Two ways are different if any person is assigned a different role in the 2 ways. Output the answer modulo $1,000,000,007$.
- **Input Format:** First line, 2 integers, `N K`.
  $1 \leq N \leq 200, 1 \leq K \leq 500$.
  Next K lines each describe a role as a pair `l_i u_i`. For all i, $0 \leq l_i \leq u_i \leq N$.

- **Sample Input:**

  ```
  4 2
  1 3
  1 2
  ```

- **Sample Output:** 10

- 2 different directions we can start with.
- First we can try to do maths and find a closed form.
- One can certainly write out an equation, but it has a lot of binomial coefficients and it is very unclear how to get a closed form.
- But remember: this is *Programming* Challenges, not *Math* Challenges.
- So we try our second option. Build the answer up role by role.
- Aka: DP.

Example: Diverse Committee

- What order should we do the DP in and what is the state?
- Since restrictions are tied to roles, it makes sense to assign each role at once. Else our state will have to keep track of how many people of each role we've assigned.
- So we want to have the state $dp[r]$ which is number of different assignments using just the roles up to the rth role. Our transition is to assign role r to a set of people (we assign one role at a time, as opposed to one person at a time). Is our state big enough?
- Pretty clearly we need to store something about the people who have already been assigned roles.
- As a start, we will try an exponential DP. We can use the state $dp[r][S]$ where S is the set of people who have been assigned a role.

- Then $dp[r][S]$ will be the number of valid assignments, using exactly roles 1 to r and assigning roles to exactly the set of people S.
- It is worth being careful with definitions here.
- $dp[r][S]$ contains the answer to the problem, assuming only the first r roles exist and only the people in the set S exist. So it is the number of assignments to S of the first r roles such that, for each of the first r roles, the number of people with that role is in the set $[l_i, u_i]$.
- What are the choices/transitions?
- Transitions correspond to increasing r to $r + 1$ and hence picking a set of people to give the role $r + 1$.

- Then we need to pick a subset of people to give role $r+1$ to. This gives:

$$dp[r+1][S] = \sum_{\substack{S' \subseteq S \\ |S'| \in [l_{r+1}, u_{r+1}]}} dp[r][S \setminus S']$$

  where S' is the set of people we give role $r+1$ to.
- This is valid but so so slow.

- To speed it up we should note it does not matter which set of S people we assigned a role to. People are interchangeable (as we all know).
- So $dp[r][S1] = dp[r][S2]$ whenever $S1$ and $S2$ are sets of the same size.
- So instead we will just calculate $dp2[r][n]$ ($dp2$ just for clarity). $dp2[r][n]$ will be the number of valid assignments, for roles 1 to r where exactly n people have been assigned a role.
- We should be careful to spell out exactly what this means.

- The easiest definition is $dp2[r][n] := dp[r][\{1, \ldots, n\}]$.
- So $dp2[r][n]$ is the number of ways to assign exactly the roles 1 to r to a generic set of n people. Again, we only count assignments where for each of the first r roles, the number of people with that role is in the set $[l_i, u_i]$.
- But we do consider these n people to be different. It is really best to think of $dp2[r][n]$ as $dp[r][\{1, \ldots, n\}]$.
- This is important because otherwise we may undercount. E.g: we need to be sure we count assignment $(1, 2)$ as different from assignment $(2, 1)$.

- What is the recurrence? What are our choices/transitions?
- Again, our choices/transitions are assigning people with the role $r+1$.
- To calculate $dp[r+1][n]$ we need to consider all ways of assigning role $r+1$ to a subset of the first n people.
- That is

$$dp2[r+1][n] = \sum_{\substack{S' \subseteq \{1,\ldots,n\} \\ |S'| \in [l_{r+1}, u_{r+1}]}} dp2[r][n - |S'|]$$

(compare to previous recurrence)

- Still slow. To speed this up, we should note that it no longer matters what set S' we pick, just its size.

- Slow recurrence:

$$dp2[r+1][n] = \sum_{\substack{S' \subseteq \{1,\ldots,n\} \\ |S'| \in [l_{r+1}, u_{r+1}]}} dp2[r][n - |S'|]$$

- How many ways are there of picking a subset S' of $\{1, \ldots, n\}$ when $|S'| = s$ for given s?
- Answer: $\binom{n}{s}$
- So by grouping together all sets S' that have the same size, we get

$$dp2[r+1][n] = \sum_{s=l_{r+1}}^{u_{r+1}} \binom{n}{s} \cdot dp2[r][n-s]$$

- This recurrence is $O(N)$ assuming we precompute our binomial coefficients.
- Much better!

Example: Diverse Committee


```cpp
#include <bits/stdc++.h>
using namespace std;

const long long MOD = 1000000007;
const int MAXK(500), MAXN(200);
long long binom[MAXN+1][MAXN+1];

long long madd(long long a, long long b) {
    return (a + b) % MOD;
}

void precomp() {
    for (int i = 0; i <= MAXN; i++) {
        binom[i][0] = 1;
        for (int j = 1; j <= i; j++) {
            binom[i][j] = madd(binom[i-1][j-1], binom[i-1][j]);
        }
    }
}
```

Example: Diverse Committee


```
int N, K, l[MAXK+1], u[MAXK+1];
long long dp[MAXK+1][MAXN+1];

int main() {
    precomp();

    scanf("%d %d", &N, &K);
    // 1-indexing is nicer as now our base case corresponds to r = 0.
    for (int r = 1; r <= K; r++) {
        scanf("%d %d", &l[r], &u[r]);
    }
    // Base case:
    dp[0][0] = 1;
    for (int r = 1; r <= K; r++) {
        for (int n = 0; n <= N; n++) {
            for (int s = l[r]; s <= u[r]; s++) {
                // Careful: there are no subsets of {1,..,n} with size s > n
                    .
                if (n - s < 0) continue;
                dp[r][n] = madd(dp[r][n], binom[n][s]*dp[r-1][n-s]);
            }
        }
    }
    printf("%lld\n", dp[K][N]);
}
```

- **Complexity?** $O(NK)$ state space with $O(N)$ recurrence, hence $O(N^2K)$.
- This is common for many counting problems.
- First, you should always consider whether you can get away with DP for combinatorics. For non-simple, non-well known examples it is generally a lot easier than finding a closed form.
- Until you are used to it, it is worth starting with the exponential DP.
- Then to improve the state space and recurrence, you want to exploit symmetry. Usually the objects we are assigning are indistinguishable, so you just need to keep the count of assigned objects.
- This is why binomial coefficients show up so often in combinatorial DP problems.

- The key is to be careful with what exactly your DP state means.
- *Make sure* you understand what the DP state meant in this example and why the recurrence had binomial coefficients in it.

- $|A \cup B| = |A| + |B| - |A \cap B|$
- $|A \cup B \cup C| =$
  $|A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$
- In general,

$$|A_1 \cup A_2 \cup \ldots \cup A_n| = \sum_{I \subseteq \{1,\ldots,n\}, I \neq \emptyset} (-1)^{|I|+1} \left| \bigcap_{i \in I} A_i \right|$$

- More often you will actually want $|X \setminus \cup_{i=1}^{n} A_i|$.

Inclusion/exclusion

- In English: Suppose you have a bunch of bad properties $\{p_1, \ldots, p_n\}$. You want to count the number of elements of some set X that satisfy none of these properties. (elements for which p_1 is false AND p_2 is false AND ...).
- Inclusion Exclusion tells you you can flip this problem on its head and instead count sets of elements that DO satisfy these bad proprties.
- E.g: the number of elements that satisfy neither $\{p_1, p_2\}$ is

```
(Size of X)
- (num eles that satisfy p1)
- (num eles that satisfy p2)
+ (num eles that satisfy both)
```

- And often it is easier to count elements that satisfy properties than ones that don't!

- Concrete example: how many ways can I roll 2 die such that die 1 is ≤ 4, die 2 is ≤ 5.
- It is
  - total number of ways (36)
  - - ways where die 1 is > 4 $(2 * 6 = 12)$
  - - ways where die 2 is > 5 $(1 * 6 = 6)$
  - $+$ ways where die 1 is > 4, die 2 is > 5 $(2 * 1 = 2)$
  - $= 20$

- **Problem Statement:** Count the number of permutations of length N ($1 \leq N \leq 100,000$) such that for each of the first K ($1 \leq K \leq 15$) elements, $p_i \neq i$.
  Output the answer MOD 1,000,000,007.

Example: Careless Secretary

- Why is this hard to do directly?
- Roughly because if you assign one of the first K values to the first position then this affects the answer a lot. So you'll have to remember this.
- Using this, one can do an exponential DP. But inclusion/exclusion is cleaner here and generalizes better.

Example: Careless Secretary

- Instead of counting permutations that don't have fixed points, inclusion/exclusion tells us we should count permutations that DO!
- Formally, the bad properties we want to avoid are $\{P_1, \ldots, P_K\}$ where P_i is the property that $p_i = i$.
- So inclusion exclusion tells us we can count this by instead counting the number of permutations that satisfy some subset of the $\{P_1, \ldots, P_K\}$ and then aggregating this over all subsets.

- Why is this easier? How do we count the permutations that satisfy e.g: P_1 and P_2? What does this even mean?
- It means that $p_1 = 1$ and $p_2 = 2$. That's it.
- So the number of permutations satisfying P_1 and P_2 is just $(N-2)!$.
- So for any subset of the bad properties, $S \subseteq \{P_1, \ldots, P_K\}$, the number of permutations satisfying S is just $(N - |S|)!$.

- Hence the number of permutations avoiding all bad properties is just

$$\sum_{S \subseteq \{P_1, \ldots, P_K\}} (-1)^{|S|} (N - |S|)!$$

Example: Careless Secretary


```cpp
#include <bits/stdc++.h>
using namespace std;

using ll = long long;
const ll MOD = 1000000007;
const int MAXN = 100000;

ll madd(ll a, ll b) {
    return ((a + b) % MOD + MOD) % MOD;
}

ll fact[MAXN+1];
int main() {
    fact[0] = 1;
    for (int i = 1; i <= MAXN; i++) {
        fact[i] = (i * fact[i-1]) % MOD;
    }
    int N, K;
    scanf("%d %d", &N, &K);
    ll ans = 0;
    for (int i = 0; i < (1 << K); i++) {
        int bitcount = __builtin_popcount(i);
        int sign = bitcount % 2 ? -1 : 1;
        ans = madd(ans, sign * fact[N-bitcount]);
    }
    printf("%lld\n", ans);
    return 0;
}
```

- **Complexity?** $O(N + 2^K)$.
- Actually we can do better. In the code, note that we don't care what the exact subset is, just its size.
- So we can do all $\binom{K}{i}$ subsets of size i at once, for each value of $i \in [0, K]$.

Example: Careless Secretary


```
ll madd(ll a, ll b) {
    return ((a + b) % MOD + MOD) % MOD;
}
ll mmult(ll a, ll b) {
    return (a*b) % MOD;
}
// Add your modinv (and probably your modpow) code here
ll modinv(ll x);

ll fact[MAXN+1], invfact[MAXN+1];
ll choose(ll n, ll r) {
    return mmult(fact[n], mmult(invfact[r], invfact[n-r]));
}

int main() {
    fact[0] = invfact[0] = 1;
    for (int i = 1; i <= MAXN; i++) {
        fact[i] = (i * fact[i-1]) % MOD;
        invfact[i] = modinv(fact[i]);
    }
    int N, K;
    scanf("%d %d", &N, &K);
    ll ans = 0;
    for (int i = 0; i <= K; i++) {
        int sign = i % 2 ? -1 : 1;
        // (-1)^i * (K choose i) * (N-i)!
        ll cways = mmult(choose(K, i), fact[N-i]);
        ans = madd(ans, sign * cways);
    }
    printf("%lld\n", ans);
}
```

Example: Careless Secretary

- **Complexity?** $O(N \log MOD)$.
- Why was inclusion/exclusion helpful here? Because counting permutations with fixed points is much easier than counting permutations without fixed points.

Table of Contents

- Usually, complicated matrix operations do not come up
- Most useful applications in competitions just involve matrix multiplication
- Solving linear systems using Gaussian elimination and calculating rank is sometimes used

- Hopefully you still remember how to multiply matrices.
- If $C = A \cdot B$ then

$$C_{i,j} = \sum_{k=1}^{N} A_{i,k} B_{k,j}$$

- This gives an immediate $O(N^3)$ algorithm. Good enough for competitions.

Matrix Multiplication


```
// Implementation for square matrices.
struct Matrix {
  const static int N = 55;
  int n;
  long long v[N][N];

  Matrix(int _n) : n(_n) {
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) v[i][j] = 0;
  }
  Matrix operator*(const Matrix &o) const {
    Matrix res(n);
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
          res.v[i][j] += v[i][k] * o.v[k][j];
    return res;
  }
};
```

- The most interesting applications involve matrix exponentiation. This is the problem of calculating A^k where k might be large (e.g: $k = 10^{18}$).
- We use the same repeated squaring trick we use for calculating a^k, the only difference is in our definition of multiply.
- Complexity is $O(n^3 \log k)$ where n is the side length of the matrix.

Matrix Exponentiation


```
// Implementation for square matrices.
struct Matrix {
  const static int N = 55;
  int n;
  long long v[N][N];
  // Assume these have been implemented.
  Matrix(int _n);
  Matrix operator*(const Matrix &o) const;

  static Matrix getIdentity(int n) {
    Matrix res(n);
    for (int i = 0; i < n; i++) res.v[i][i] = 1;
    return res;
  }
  Matrix operator^(long long k) const {
    Matrix res = Matrix::getIdentity(n);
    Matrix a = *this;
    while (k) {
      if (k&1) res = res*a;
      a = a*a;
      k /= 2;
    }
    return res;
  }
};
```

- The adjacency matrix of a directed graph is a square matrix with side length V and entries $A_{i,j} =$ (num edges from $i \to j$).
- We can rephrase this to say A is the matrix that counts the number of length 1 paths between vertices.
- Then it is not hard to check (matrix multiplication is defined exactly to make the following statement true) that A^k is the matrix whose (i, j)-th entry is the number of length k paths from i to j.
- So we can find the number of length k paths from a to b in $O(V^3 \log k)$.
- We can similarly e.g: find the shortest length k path from a to b.

- One very nice application is solving linear recurrences with constant coefficients.
- These are recurrences of the form

$$a_n = c_1 a_{n-1} + \ldots + c_k a_{n-k}$$

  where c_i are constants. The question is to find a_n for a large n (say $n = 10^{18}$).
- Well known example: Fibonacci numbers.

Example: Fibonacci numbers

- Fibonacci numbers are specified by:

$$F(0) = 0$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2$$

Example: Fibonacci numbers

- You should know an $O(n)$ time, $O(n)$ memory (even $O(1)$ memory!) algorithm.
- Here is another algorithm. We can solve the recurrence (maybe you remember this from Discrete) for the closed form:

$$F(n) = \frac{\varphi^n - \psi^n}{\varphi - \psi},$$

where $\varphi = \frac{1+\sqrt{5}}{2}$ and $\psi = \frac{1-\sqrt{5}}{2}$.

  - "Constant time" - it's a little more complicated than that
  - Precision issues - the numbers in the sequence grow exponentially quickly

Example: Fibonacci numbers

- Here is a better algorithm. We can rewrite our earlier recurrence in the form:

$$\begin{pmatrix} F_{k+2} \\ F_{k+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{k+1} \\ F_k \end{pmatrix}$$

- Repeatedly multiplying this out, we get:

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

- And we can calculate the middle matrix in $O(\log n)$!
- Hence we can find F_n in $O(\log n)$ (assuming multiplication is $O(1)$, for this to be true we need to be working mod some M).

- This works more generally for any constant coefficient linear recurrence

$$a_n = c_1 a_{n-1} + \ldots + c_k a_{n-k}$$

- We get that

$$\begin{pmatrix} a_{n+k} \\ \vdots \\ a_{n+1} \end{pmatrix} = \begin{pmatrix} c_1 & c_2 & \ldots & c_{k-1} & c_k \\ 1 & 0 & \ldots & 0 & 0 \\ 0 & 1 & \ldots & 0 & 0 \\ \vdots & & & & \\ 0 & 0 & \ldots & 1 & 0 \end{pmatrix} \begin{pmatrix} a_{n+k-1} \\ \vdots \\ a_n \end{pmatrix}$$

- Exponentiating the matrix gives us a_N in $O(\log N \cdot (\text{cost of matrix multiplication})) = O(k^3 \log N)$.

Example: Freddy Frog

- **Problem Statement:** Freddy the Frog has entered a new pond. This pond has N lilypads in a row, Freddy is at lilypad 0 and wants to get to lilypad N.
  Freddy has mastered K different kinds of jumps, the ith jumping Freddy forward d_i lilypads.
  How many ways can Freddy reach the Nth lilypad? Two ways are different if the sequence of jumps Freddy performs is different. Output the answer MOD $1,000,000,007$.
- **Input Format:** First line, 2 integers `N K`. $1 \leq N \leq 10^9$, $1 \leq K \leq 100$.
  The next line has K integers, the jumps Freddy has mastered. Each integer is unique and in the range $[1, 100]$.

Example: Freddy Frog

- **Sample Input:**

  ```
  4 2
  1 2
  ```

- **Sample Output:** `5`

- There is a straight forward DP here. Let W_n be the number of different ways to reach lilypad n. What is the recurrence for W_n?
- $$W_n = \sum_{d_i} W_{n-d_i}$$

  where d_i varies over the jumps Freddy has mastered.
- This gives an $O(NK)$ solution. How do we speed it up?
- Either notice we are repeating the same operation over and over, hence exponentiation is a good idea or directly note this is a linear recurrence with constant coefficients.

Example: Freddy Frog


```
const long long MOD = 1000000007;
const int MAXJUMP = 100;

// Implementation for square matrices.
struct Matrix {
  Matrix(int _n);
  // Modify this to work mod MOD.
  Matrix operator*(const Matrix &o) const;
  static Matrix getIdentity(int n);
  Matrix operator^(long long k) const;
};

int main() {
  int N, K;
  scanf("%d %d", &N, &K);
  Matrix rec(MAXJUMP);
  for (int i = 0; i < K; i++) {
    int d; scanf("%d", &d);
    // Note the d-1 here.
    rec.v[0][d-1] = 1;
  }
  for (int i = 1; i < MAXJUMP; i++) {
    rec.v[i][i-1] = 1;
  }
  rec = rec^N;
  // The first entry of rec^N multiplied by (1,0,...,0)^T
  printf("%lld\n", rec.v[0][0]);
}
```

Summary

- Matrix exponentiation (and exponentiation in general) is useful whenever you need to find an answer for a large N (with N up to 10^{18}, though N is often around 10^9) with the answer built up repeatedly from the same small pieces.
- For example, in Freddy Frog we had to find the number of paths for a large N. But paths were built up from the same repeated building blocks, the jumps Freddy has mastered. Hence we should suspect matrix exponentiation.

Table of Contents

Further Topics

- This covered some of the critical topics in maths, though far from all of them.
- I've biased towards topics that are more algorithmic and less mathematical.
- Some further critical topics I omitted:
  - In Number Theory: Chinese Remainder Theorem. The crucial tool for solving systems of linear congruence equations.
  - In Combinatorics: Inclusion-Exclusion.
  - Fast polynomial multiplication using FFT.

Some fun stuff to look at (beyond course scope):

- Grundy numbers
- Surreal numbers
- Blue/Red/Green Hackenbush
- Chinese remainder theorem
- Burnside's lemma
- Pick's theorem
- Euler's totient function
- Simpson's rule
- Minkowski sums
- Karatsuba algorithm
- Möbius inversion formula
- Cycle Space
- Matroids
- Shank's algorithm
- Cayley's formula
- Kirchhoff's matrix tree theorem
- Catalan numbers
- Stern-Brocot tree
- Continued fractions
- AKS
- Miller-Rabin