Mathematics

Adamos Ttofari

Content

- Number Theory
  - Primes and factors
  - Primality check
  - Factors
  - Sieve of Eratosthenes
  - LCM and GCD
  - Euclid's Algorithm
  - Modular Arithmetic and Exponentiation
- Combinatorics
  - Number of ways
  - Binomial Coefficient
  - Catalan Numbers
  - Inclusion Exclusion
- Matrices
  - Basic Operations
  - Matrix Multiplication
  - Matrix Exponentiation
  - Linear Recurrences
- Probabilities
  - Basic Calculation
  - Randomized Algorithms
- Game Theory
  - Winning and Losing States

Number Theory

Number theory is a branch of mathematics that studies integers. Number theory is a fascinating field, because many questions involving integers are very difficult to solve even if they seem simple at first glance.

Primes and factors

A number a is called a **factor** or a **divisor** of a number b if a divides b. If a is a factor of b, we write a | b, and otherwise we write a ∤ b. For example, the factors of 24 are 1, 2, 3, 4, 6, 8, 12 and 24.

A number n > 1 is a **prime** if its only positive factors are 1 and n. For example, 7, 19 and 41 are primes, but 35 is not a prime, because 5 ·7 = 35. For every number n > 1, there is a unique **prime factorization**

$$\boldsymbol{n = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \ldots \cdot p_k^{\alpha_k}}$$

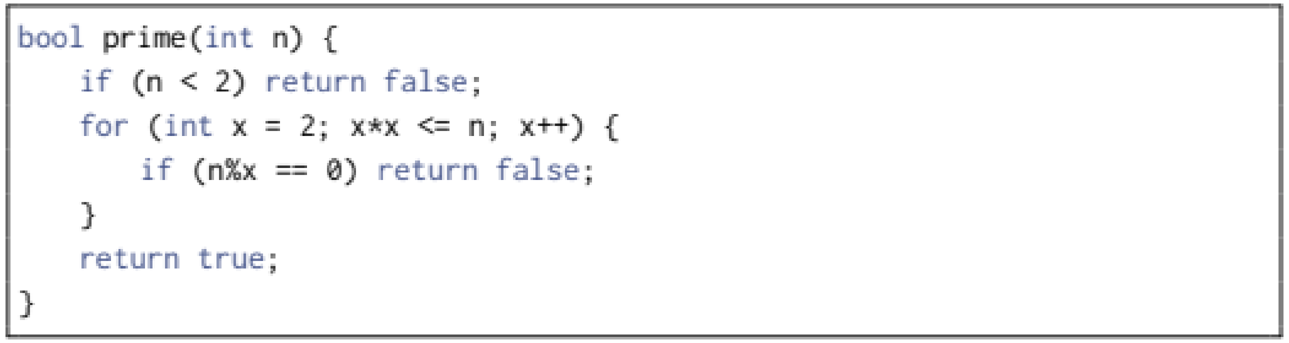
where p1, p2,..., pk are distinct primes and α1,α2,...,αk are positive numbers. For example, the prime factorization for 84 is

$$\mathbf{84 = 2^2 \cdot 3^1 \cdot 7^1}$$

Primality Check

If a number n is not prime, it can be represented as a product $a \cdot\ b$, where $a \leq \sqrt{n}$ or $b\ \leq \sqrt{n}$, so it certainly has a factor between 2 and $\sqrt{n}$. Using this observation, we can both test if a number is prime and find the prime factorization of a number in $O(\sqrt{n})$time.

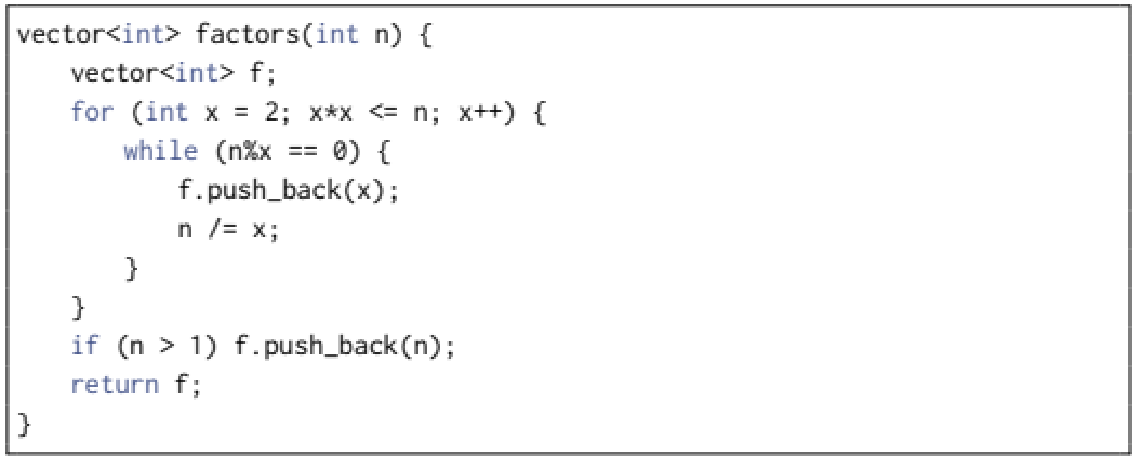
The following function prime checks if the given number n is prime. The function attempts to divide n by all numbers between 2 and $\sqrt{n}$, and if none of them divides n, then n is prime.



```
bool prime(int n) {
    if (n < 2) return false;
    for (int x = 2; x*x <= n; x++) {
        if (n%x == 0) return false;
    }
    return true;
}
```

Factors

The following function factors constructs a vector that contains the prime factorization of n. The function divides n by its prime factors and adds them to the vector. The process ends when the remaining number n has no factors between 2 and $\sqrt{n}$. If n > 1, it is prime and the last factor.



```
vector<int> factors(int n) {
    vector<int> f;
    for (int x = 2; x*x <= n; x++) {
        while (n%x == 0) {
            f.push_back(x);
            n /= x;
        }
    }
    if (n > 1) f.push_back(n);
    return f;
}
```

Note that each prime factor appears in the vector as many times as it divides the number. For example, $24 = 2^3 \cdot 3$, so the result of the function is [2,2,2,3].

Sieve of Eratosthenes

The sieve of Eratosthenes is a preprocessing algorithm that builds an array using which we can efficiently check if a given number between 2...n is prime and, if it is not, find one prime factor of the number.

The algorithm builds an array sieve whose positions 2,3,...,n are used. The value sieve[k] = 0 means that k is prime, and the value sieve[k]≠0 means that k is not a prime and one of its prime factors is sieve[k].

The algorithm iterates through the numbers 2...n one by one. Always when a new prime x is found, the algorithm records that the multiples of x (2x,3x,4x,...) are not primes, because the number x divides them.

For example, if n = 20, the array is as follows:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	2	0	3	0	2	3	5	0	3	0	7	5	2	0	3	0	5

The following code implements the sieve of Eratosthenes. The code assumes that each element of sieve is initially zero.

```
bool prime(int n) {
    if (n < 2) return false;
    for (int x = 2; x*x <= n; x++) {
        if (n%x == 0) return false;
    }
    return true;
}
```

The inner loop of the algorithm is executed n/x times for each value of x. Thus, an upper bound for the running time of the algorithm is the harmonic sum

$$\sum_{x=2}^{n} \frac{n}{x} = \frac{n}{2} + \frac{n}{3} + \frac{n}{4} + \cdots + \frac{n}{n} = O(nlgn)$$

In fact, the algorithm is more efficient, because the inner loop will be executed only if the number x is prime. It can be shown that the running time of the algorithm is only) $O(nloglogn)$, a complexity very near to $O(n)$.

Least common multiple Greatest common divisor

The greatest common divisor of numbers a and b, gcd(a,b), is the greatest number that divides both a and b, and the least common multiple of a and b, lcm(a,b), is the smallest number that is divisible by both a and b. For example, gcd(24,36) = 12 and lcm(24,36) = 72.

The greatest common divisor and the least common multiple are connected as follows:

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

Euclid’s algorithm

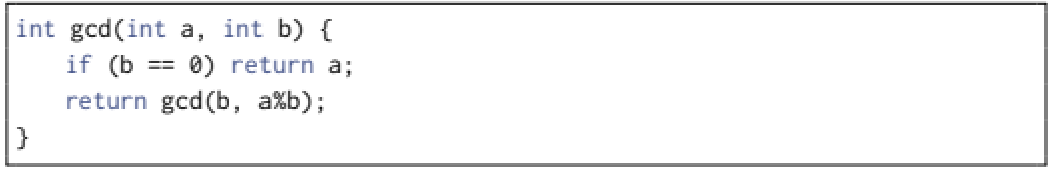
Euclid’s algorithm1 provides an efficient way to find the greatest common divisor of two numbers. The algorithm is based on the following formula:

$$\gcd(a,b) = \begin{cases} a, b = 0 \\ \gcd(b, a\ mod\ b)\ , b \neq 0 \end{cases}$$

For example,

$$\gcd(24,36) = \gcd(36,24) = \gcd(24,12) = \gcd(12,0) = 12$$

The algorithm can be implemented as follows:



```
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a%b);
}
```

It can be shown that Euclid’s algorithm works in O(logn) time, where n = min(a,b). The worst case for the algorithm is the case when a and b are consecutive Fibonacci numbers.

Modular arithmetic

In modular arithmetic, the set of numbers is limited so that only numbers 0,1,2,...,m−1 are used, where m is a constant. Each number x is represented by the number x mod m: the remainder after dividing x by m. For example, if m = 17, then 75 is represented by 75 mod 17 = 7.

Often we can take remainders before doing calculations. In particular, the following formulas hold:

$$\begin{aligned}
(x+y) \bmod m &= (x \bmod m + y \bmod m) \bmod m \\
(x-y) \bmod m &= (x \bmod m - y \bmod m) \bmod m \\
(x\cdot y) \bmod m &= (x \bmod m \cdot y \bmod m) \bmod m \\
x^n \bmod m &= (x \bmod m)^n \bmod m
\end{aligned}$$

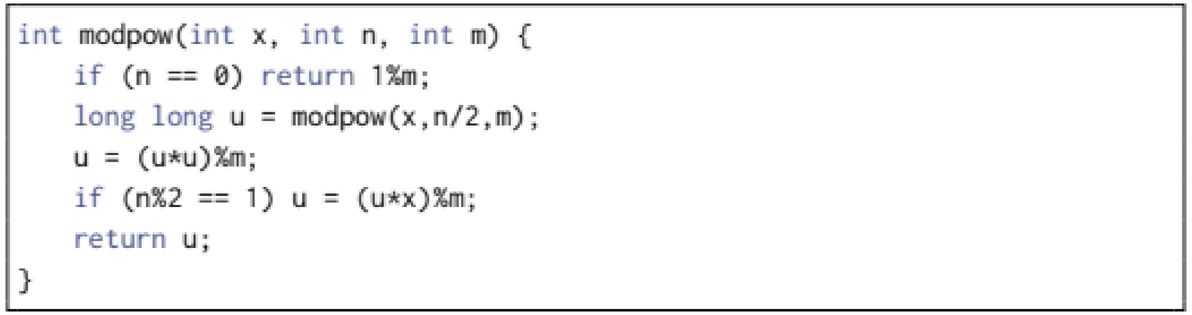
Modular exponentiation

There is often need to efficiently calculate the value of $x^n \ mod \ m$. This can be done in O(logn) time using the following recursion:

$$x^n = \begin{cases} 1, n = 0 \\ x^{n/2} \cdot x^{n/2}, n \ is \ even \\ x^{n-1} \cdot \mathrm{x}\,, n \ is \ odd \end{cases}$$

It is important that in the case of an even n, the value of x n/2 is calculated only once. This guarantees that the time complexity of the algorithm is O(logn), because n is always halved when it is even.

The following function calculates the value of $x^n \ mod \ m$:



```
int modpow(int x, int n, int m) {
    if (n == 0) return 1%m;
    long long u = modpow(x,n/2,m);
    u = (u*u)%m;
    if (n%2 == 1) u = (u*x)%m;
    return u;
}
```

Combinatorics

Combinatorics studies methods for counting combinations of objects. Usually, the goal is to find a way to count the combinations efficiently without generating each combination separately.

Number of ways

As an example, consider the problem of counting the number of ways to represent an integer n as a sum of positive integers. For example, there are 8 representations for 4:

- $1+1+1+1$
- $1+1+2$
- $1+2+1$
- $2+1+1$
- $2+2$
- $3+1$
- $1+3$
- 4

A combinatorial problem can often be solved using a recursive function. In this problem, we can define a function f (n) that gives the number of representations for n. For example, f (4) = 8 according to the above example. The values of the function can be recursively calculated as follows:

$$f(n) = \begin{cases} 1 & n = 0 \\ f(0) + f(1) + \cdots + f(n-1) & n > 0 \end{cases}$$

Sometimes, a recursive formula can be replaced with a closed-form formula. In this problem,

$$f(n) = 2^{n-1}$$

which is since there are n−1 possible positions for +-signs in the sum and we can choose any subset of them.

1 [ ] 1 [ ] 1 [ ] 1

Binomial coefficients

The binomial coefficient $\binom{n}{k}$ equals the number of ways we can choose a subset of k elements from a set of n elements. For example, $\binom{5}{3} = 10$, because the set {1,2,3,4,5} has 10 subsets of 3 elements:

{1,2,3},{1,2,4},{1,2,5},{1,3,4},{1,3,5},{1,4,5},{2,3,4},{2,3,5},{2,4,5},{3,4,5}

Binomial coefficients can be recursively calculated as follows:

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

The idea is to fix an element x in the set. If x is included in the subset, we have to choose k −1 elements from n−1 elements, and if x is not included in the subset, we have to choose k elements from n−1 elements.

The base cases for the recursion are

$$\binom{n}{0} = \binom{n}{n} = 1$$

because there is always exactly one way to construct an empty subset and a subset that contains all the elements.

Another way to calculate binomial coefficients is as follows:

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

There are n! permutations of n elements. We go through all permutations and always include the first k elements of the permutation in the subset. Since the order of the elements in the subset and outside the subset does not matter, the result is divided by k! and (n− k)!

For binomial coefficients,

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

because we actually divide a set of n elements into two subsets: the first contains k elements and the second contains n− k elements.

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n} = 2^n$$

Catalan numbers

The Catalan number C_n equals the number of valid parenthesis expressions that consist of n left parentheses and n right parentheses. For example, $C_3 = 5$, because we can construct the following parenthesis expressions using three left and right parentheses:

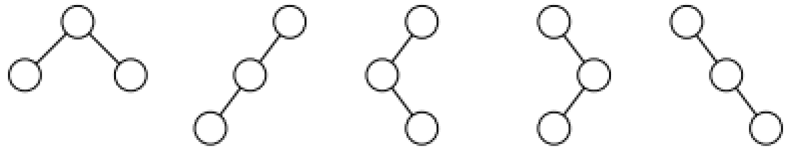
()()() • (())() • ()(()) • ((())) • (()())

A way to characterize valid parenthesis expressions is that if we choose any prefix of such an expression, it has to contain at least as many left parentheses as right parentheses. In addition, the complete expression has to contain an equal number of left and right parentheses.

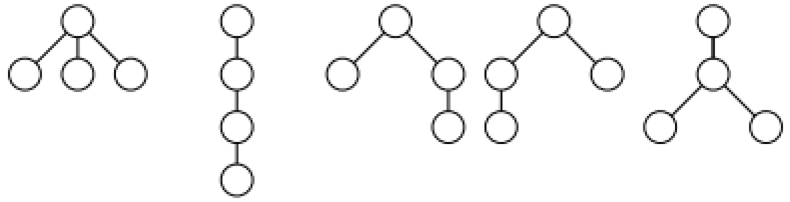
Catalan numbers are also related to trees:

- there are C_n binary trees of n nodes
- there are C_{n-1} rooted trees of n nodes

For example, for C_3 = 5, the binary trees are



and the rooted trees are



Catalan numbers can also be calculated using binomial coefficients:

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

The formula can be explained as follows:

There are a total of $\binom{2n}{n}$ ways to construct a (not necessarily valid) parenthesis expression that contains n left parentheses and n right parentheses. Let us calculate the number of such expressions that are not valid.

If a parenthesis expression is not valid, it has to contain a prefix where the number of right parentheses exceeds the number of left parentheses. The idea is to reverse each parenthesis that belongs to such a prefix. For example, the expression ())()(contains a prefix ()), and after reversing the prefix, the expression becomes)((()(.

The resulting expression consists of n + 1 left parentheses and n − 1 right parentheses. The number of such expressions is $\binom{2n}{n+1}$, which equals the number of non-valid parenthesis expressions. Thus, the number of valid parenthesis expressions can be calculated using the formula

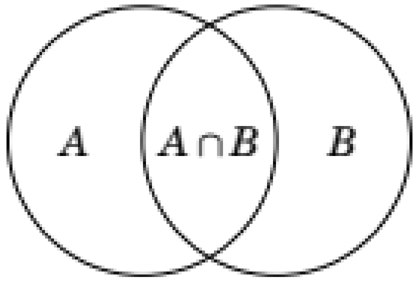
$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1}\binom{2n}{n} = \frac{1}{n+1}\binom{2n}{n}$$

Inclusion-exclusion

Inclusion-exclusion is a technique that can be used for counting the size of a union of sets when the sizes of the intersections are known, and vice versa. A simple example of the technique is the formula

$$|A \cup B| = |A| + |B| - |A \cap B|$$

where A and B are sets and $|X|$ denotes the size of X. The formula can be illustrated as follows:

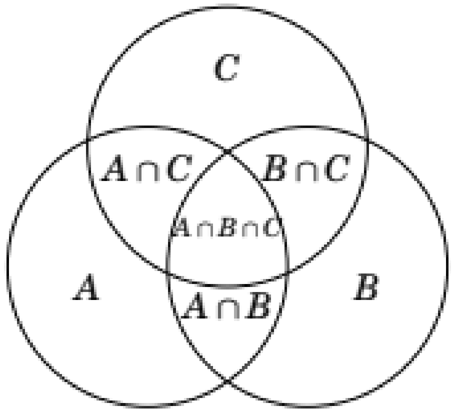


Our goal is to calculate the size of the union A ∪ B that corresponds to the area of the region that belongs to at least one circle. The picture shows that we can calculate the area of A ∪B by first summing the areas of A and B and then subtracting the area of A ∩B.

The same idea can be applied when the number of sets is larger. When there are three sets, the inclusion-exclusion formula is

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

and the corresponding picture is



Matrices

A matrix is a mathematical concept that corresponds to a two-dimensional array in programming. For example,

$$A = \begin{bmatrix} 6 & 13 & 7 & 4 \\ 7 & 0 & 8 & 2 \\ 9 & 5 & 4 & 18 \end{bmatrix}$$

is a matrix of size 3×4, i.e., it has 3 rows and 4 columns. The notation [i, j] refers to the element in row i and column j in a matrix. For example, in the above matrix, A[2,3] = 8 and A[3,1] = 9. A special case of a matrix is a vector that is a one-dimensional matrix of size n×1. For example

$$V = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

is a vector that contains three elements.

A matrix is a square matrix if it has the same number of rows and columns. For example, the following matrix is a square matrix:

$$S = \begin{bmatrix} 3 & 12 & 4 \\ 5 & 9 & 15 \\ 0 & 2 & 4 \end{bmatrix}$$

Basic Operations

The sum A + B of matrices A and B is defined if the matrices are of the same size. The result is a matrix where each element is the sum of the corresponding elements in A and B.

$$\begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} + \begin{bmatrix} 4 & 9 & 3 \\ 8 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 6+4 & 1+9 & 4+3 \\ 3+8 & 9+1 & 2+3 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 7 \\ 11 & 10 & 5 \end{bmatrix}$$

Multiplying a matrix A by a value x means that each element of A is multiplied by x.

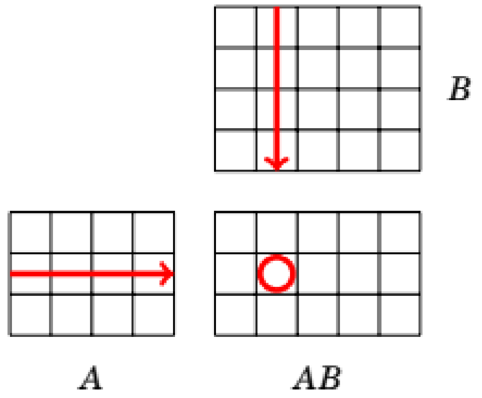
$$2 \cdot \begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} = \begin{bmatrix} 2 \cdot 6 & 2 \cdot 1 & 2 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 9 & 2 \cdot 2 \end{bmatrix} = \begin{bmatrix} 12 & 2 & 8 \\ 6 & 18 & 4 \end{bmatrix}$$

Matrix Multiplication

The product AB of matrices A and B is defined if A is of size a × n and B is of size n × b, i.e., the width of A equals the height of B. The result is a matrix of size a× b whose elements are calculated using the formula

$$AB[i,j] = \sum_{k=1}^{n} A[i,k] \cdot B[k,j].$$

The idea is that each element of AB is a sum of products of elements of A and B according to the following picture:



$$\begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 6 \\ 2 & 9 \end{bmatrix} = \begin{bmatrix} 1\cdot1+4\cdot2 & 1\cdot6+4\cdot9 \\ 3\cdot1+9\cdot2 & 3\cdot6+9\cdot9 \\ 8\cdot1+6\cdot2 & 8\cdot6+6\cdot9 \end{bmatrix} = \begin{bmatrix} 9 & 42 \\ 21 & 99 \\ 20 & 102 \end{bmatrix}$$

Matrix multiplication is associative, so A(BC) = (AB)C holds, but it is not commutative, so AB = BA does not usually hold.

Using a straightforward algorithm, we can calculate the product of two n× n matrices in $O(n^3)$ time. There are also more efficient algorithms for matrix multiplication , but they are mostly of theoretical interest and such algorithms are not necessary in competitive programming.

Matrix Exponentiation

The power A^k of a matrix A is defined if A is a square matrix. The definition is based on matrix multiplication:

$$A^k = \underbrace{A \cdot A \cdot A \cdots A}_{k \text{ times}}$$

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^3 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} = \begin{bmatrix} 48 & 165 \\ 33 & 114 \end{bmatrix}$$

In addition, A^0 is an identity matrix.

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The matrix A k can be efficiently calculated in O(n 3 logk) time using the algorithm we used for modular expomentiation

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^8 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4 \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4$$

Linear recurrences

A linear recurrence is a function f (n) whose initial values are f (0), f (1),..., f (k− 1) and larger values are calculated recursively using the formula

$$f(n) = c_1 f(n-1) + c_2 f(n-2) + \cdots + c_k f(n-k)$$

where $c_1, c_2, \ldots, c_k$ are constant coefficients.

Dynamic programming can be used to calculate any value of f (n) in O(kn) time by calculating all values of f (0), f (1),..., f (n) one after another. However, if k is small, it is possible to calculate f (n) much more efficiently in $O(k^3 logn)$ time using matrix operations.

Fibonacci numbers

A simple example of a linear recurrence is the following function that defines the Fibonacci numbers:

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2)$$

In this case, $k = 2$ and $c1 = c2 = 1$

To efficiently calculate Fibonacci numbers, we represent the Fibonacci formula as a square matrix X of size 2×2, for which the following holds:

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \end{bmatrix}$$

Thus, values f (i) and f (i +1) are given as ”input” for X, and X calculates values f (i +1) and f (i +2) from them. It turns out that such a matrix is

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

For example,

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(5) \\ f(6) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 8 \end{bmatrix} = \begin{bmatrix} 8 \\ 13 \end{bmatrix} = \begin{bmatrix} f(6) \\ f(7) \end{bmatrix}$$

Thus, we can calculate f (n) using the formula

$$\begin{bmatrix} f(n) \\ f(n+1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The value of X n can be calculated in O(logn) time, so the value of f (n) can also be calculated in O(logn) time.

General case

Let us now consider the general case where f (n) is any linear recurrence. Again, our goal is to construct a matrix X for which

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \\ \vdots \\ f(i+k) \end{bmatrix}$$

Such a matrix is

$$X = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ c_k & c_{k-1} & c_{k-2} & c_{k-3} & \cdots & c_1 \end{bmatrix}$$

In the first k −1 rows, each element is 0 except that one element is 1. These rows replace f (i) with f (i +1), f (i +1) with f (i +2), and so on. The last row contains the coefficients of the recurrence to calculate the new value f (i + k).

Now, f(n) can be calculated in O(k^3logn) time using the formula.

$$\begin{bmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{bmatrix}$$

Probabilities

A probability is a real number between 0 and 1 that indicates how probable an event is. If an event is certain to happen, its probability is 1, and if an event is impossible, its probability is 0. The probability of an event is denoted $P(\cdots)$ where the three dots describe the event.

Calculation

To calculate the probability of an event, we can either use combinatorics or simulate the process that generates the event. As an example, let us calculate the probability of drawing three cards with the same value from a shuffled deck of cards (for example, ♠8, ♣8 and ♦8).

We can calculate the probability using the formula

$$\frac{number\ of\ desired\ outcomes}{\text{total number of outcomes}}$$

In this problem, the desired outcomes are those in which the value of each card is the same. There are 13 $\binom{4}{3}$such outcomes, because there are 13 possibilities for the value of the cards and $\binom{4}{3}$ ways to choose 3 suits from 4 possible suits.

There are a total of $\binom{52}{3}$ outcomes, because we choose 3 cards from 52 cards. Thus, the probability of the event is

$$\frac{13\binom{4}{3}}{\binom{52}{3}} = \frac{1}{425}$$

Another way to calculate the probability is to simulate the process that generates the event. In this example, we draw three cards, so the process consists of three steps. We require that each step of the process is successful.

Drawing the first card certainly succeeds, because there are no restrictions. The second step succeeds with probability 3/51, because there are 51 cards left and 3 of them have the same value as the first card. In a similar way, the third step succeeds with probability 2/50.

The probability that the entire process succeeds is

$$1 \cdot \frac{3}{51} \cdot \frac{2}{50} = \frac{1}{425}$$

Randomized algorithms

Sometimes we can use randomness for solving a problem, even if the problem is not related to probabilities. A randomized algorithm is an algorithm that is based on randomness.

A **Monte Carlo algorithm** is a randomized algorithm that may sometimes give a wrong answer. For such an algorithm to be useful, the probability of a wrong answer should be small.

A **Las Vegas algorithm** is a randomized algorithm that always gives the correct answer, but its running time varies randomly. The goal is to design an algorithm that is efficient with high probability.

Order statistics

The kth order statistic of an array is the element at position k after sorting the array in increasing order. It is easy to calculate any order statistic in O(nlogn) time by first sorting the array, but is it really needed to sort the entire array just to find one element?

It turns out that we can find order statistics using a randomized algorithm without sorting the array. The algorithm, called quickselect , is a Las Vegas algorithm: its running time is usually O(n) but O(n^2) in the worst case.

The algorithm chooses a random element x of the array and moves elements smaller than x to the left part of the array, and all other elements to the right part of the array. This takes O(n) time when there are n elements. Assume that the left part contains a elements and the right part contains b elements. If a = k, element x is the kth order statistic. Otherwise, if a > k, we recursively find the kth order statistic for the left part, and if a < k, we recursively find the rth order statistic for the right part where r = k − a. The search continues in a similar way, until the element has been found.

When each element x is randomly chosen, the size of the array about halves at each step, so the time complexity for finding the kth order statistic is about

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots < 2n = O(n)$$

The worst case of the algorithm requires still O(n 2) time, because it is possible that x is always chosen in such a way that it is one of the smallest or largest elements in the array and O(n) steps are needed. However, the probability for this is so small that this never happens in practice.

Verifying matrix multiplication

Our next problem is to verify if AB = C holds when A, B and C are matrices of size n × n. Of course, we can solve the problem by calculating the product AB again (in O(n^3) time using the basic algorithm), but one could hope that verifying the answer would by easier than to calculate it from scratch.

It turns out that we can solve the problem using a Monte Carlo algorithm whose time complexity is only O(n^2). The idea is simple: we choose a random vector X of n elements, and calculate the matrices ABX and CX. If ABX = CX, we report that AB = C, and otherwise we report that AB 6= C.

The time complexity of the algorithm is O(n^2), because we can calculate the matrices ABX and CX in O(n^2) time. We can calculate the matrix ABX efficiently by using the representation A(BX), so only two multiplications of n×n and n×1 size matrices are needed.

The drawback of the algorithm is that there is a small chance that the algorithm makes a mistake when it reports that AB = C. For example,

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \neq \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix}$$

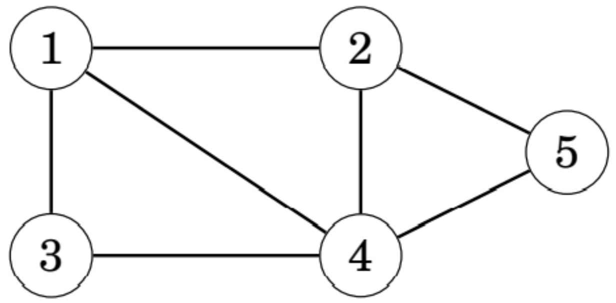
But

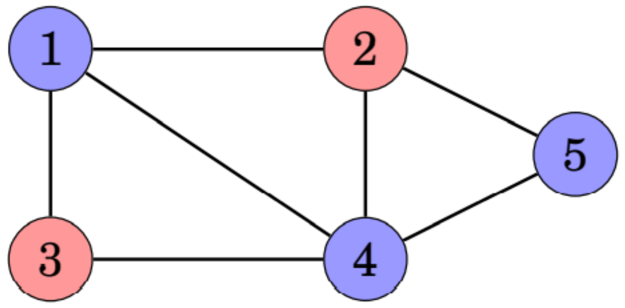
$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix}$$

However, in practice, the probability that the algorithm makes a mistake is small, and we can decrease the probability by verifying the result using multiple random vectors X before reporting that AB = C.

Graph coloring

Given a graph that contains n nodes and m edges, our task is to find a way to color the nodes of the graph using two colors so that for at least m/2 edges, the endpoints have different colors. For example, in the graph and a valid coloring is as follows:





The above graph contains 7 edges, and for 5 of them, the endpoints have different colors, so the coloring is valid.

The problem can be solved using a Las Vegas algorithm that generates random colorings until a valid coloring has been found. In a random coloring, the color of each node is independently chosen so that the probability of both colors is 1/2.

In a random coloring, the probability that the endpoints of a single edge have different colors is 1/2. Hence, the expected number of edges whose endpoints have different colors is m/2. Since it is expected that a random coloring is valid, we will quickly find a valid coloring in practice.

Game Theory

We will focus on two-player games that do not contain random elements. Our goal is to find a strategy that we can follow to win the game no matter what the opponent does, if such a strategy exists

Game states

Let us consider a game where there is initially a heap of n sticks. Players A and B move alternately, and player A begins. On each move, the player has to remove 1, 2 or 3 sticks from the heap, and the player who removes the last stick wins the game. For example, if n = 10, the game may proceed as follows:

- Player A removes 2 sticks (8 sticks left).
- Player B removes 3 sticks (5 sticks left).
- Player A removes 1 stick (4 sticks left).
- Player B removes 2 sticks (2 sticks left).
- Player A removes 2 sticks and wins.

This game consists of states 0,1,2,...,n, where the number of the state corresponds to the number of sticks left.

Winning and losing states

A winning state is a state where the player will win the game if they play optimally, and a losing state is a state where the player will lose the game if the opponent plays optimally. It turns out that we can classify all states of a game so that each state is either a winning state or a losing state.

In the above game, state 0 is clearly a losing state, because the player cannot make any moves. States 1, 2 and 3 are winning states, because we can remove 1, 2 or 3 sticks and win the game. State 4, in turn, is a losing state, because any move leads to a state that is a winning state for the opponent.

More generally, if there is a move that leads from the current state to a losing state, the current state is a winning state, and otherwise the current state is a losing state. Using this observation, we can classify all states of a game starting with losing states where there are no possible moves.

The states 0...15 of the above game can be classified as follows (W denotes a winning state and L denotes a losing state):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	*W*	*W*	*W*	*L*	*W*	*W*	*W*	*L*	*W*	*W*	*W*	*L*	*W*	*W*	*W*

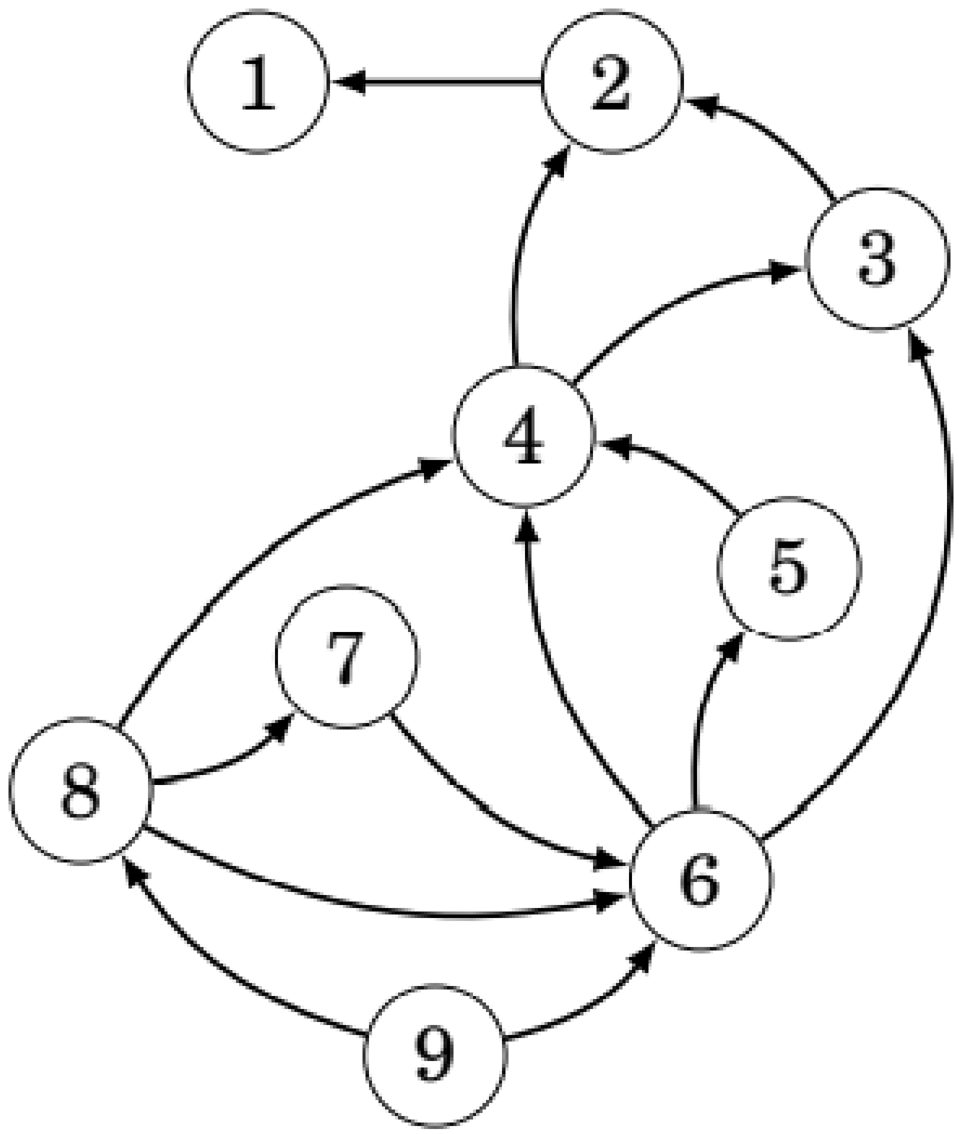
It is easy to analyze this game: a state k is a losing state if k is divisible by 4, and otherwise it is a winning state. An optimal way to play the game is to always choose a move after which the number of sticks in the heap is divisible by 4. Finally, there are no sticks left and the opponent has lost.

Of course, this strategy requires that the number of sticks is not divisible by 4 when it is our move. If it is, there is nothing we can do, and the opponent will win the game if they play optimally

State graph

Let us now consider another stick game, where in each state k, it is allowed to remove any number x of sticks such that x is smaller than k and divides k. For example, in state 8 we may remove 1, 2 or 4 sticks, but in state 7 the only allowed move is to remove 1 stick.

The following picture shows the states 1...9 of the game as a state graph, whose nodes are the states and edges are the moves between them:



The final state in this game is always state 1, which is a losing state, because there are no valid moves. The classification of states 1...9 is as follows:

1	2	3	4	5	6	7	8	9
L	*W*	*L*	*W*	*L*	*W*	*L*	*W*	*L*

Surprisingly, in this game, all even-numbered states are winning states, and all odd-numbered states are losing states.

A more general approach is to use the following recurrence relation.

$$Win(i) = \begin{cases} 0, \mathit{base\ lose\ case} \\ 1, \mathit{base\ win\ case} \\ \bigvee\{\neg Win(j) \mid j \in SucesiveStates(i)\}, \mathit{other\ cases} \end{cases}$$

"Mathematics is the gate and key to the sciences."

adamos2468@gmail.com