# Mathematics

Tómas Ken Magnússon
Bjarki Ágúst Guðmundsson

**Árangursrík forritun og lausn verkefna**

School of Computer Science
Reykjavík University

# Today we're going to cover

Basics

Number Theory

Combinatorics

Game Theory

# Basics

Computer Science $\subset$ Mathematics

- Usually at least one problem that involves solving mathematically.
- Problems often require mathematical analysis to be solved efficiently.
- Using a bit of math before coding can also shorten and simplify code.

# Finding patterns and formulas

- Some problems have solutions that form a pattern.

# Finding patterns and formulas

- Some problems have solutions that form a pattern.
- By finding the pattern, we solve the problem.
- Could be classified as mathematical ad-hoc problem.
- Requires mathematical intuition.

# Finding patterns and formulas

- Some problems have solutions that form a pattern.
- By finding the pattern, we solve the problem.
- Could be classified as mathematical ad-hoc problem.
- Requires mathematical intuition.
- Useful tricks:
    - Solve some small instances by hand.
    - See if the solutions form a pattern.

# Finding patterns and formulas

- Some problems have solutions that form a pattern.
- By finding the pattern, we solve the problem.
- Could be classified as mathematical ad-hoc problem.
- Requires mathematical intuition.
- Useful tricks:
    - Solve some small instances by hand.
    - See if the solutions form a pattern.
- Does the pattern involve some overlapping subproblem?

# Finding patterns and formulas

- Some problems have solutions that form a pattern.
- By finding the pattern, we solve the problem.
- Could be classified as mathematical ad-hoc problem.
- Requires mathematical intuition.
- Useful tricks:
  - Solve some small instances by hand.
  - See if the solutions form a pattern.
- Does the pattern involve some overlapping subproblem?
  We might need to use DP.

# Finding patterns and formulas

- Some problems have solutions that form a pattern.
- By finding the pattern, we solve the problem.
- Could be classified as mathematical ad-hoc problem.
- Requires mathematical intuition.
- Useful tricks:
  - Solve some small instances by hand.
  - See if the solutions form a pattern.
- Does the pattern involve some overlapping subproblem?
  We might need to use DP.
- Knowing reoccurring identities and sequences can be helpful.

# Arithmetic progression

- Often we see a pattern like

$$2, 5, 8, 11, 14, 17, 20, \ldots$$

# Arithmetic progression

- Often we see a pattern like

$$2, 5, 8, 11, 14, 17, 20, \ldots$$

- This is called a arithmetic progression.

$$a_n = a_{n-1} + c$$

# Arithmetic progression

- Depending on the situation we may want to get the $n$-th element

$$a_n = a_1 + (n-1)c$$

- Or the sum over a finite portion of the progression

$$S_n = \frac{n(a_1 + a_n)}{2}$$

# Arithmetic progression

- Depending on the situation we may want to get the $n$-th element

$$a_n = a_1 + (n-1)c$$

- Or the sum over a finite portion of the progression

$$S_n = \frac{n(a_1 + a_n)}{2}$$

- Remember this one?

$$1 + 2 + 3 + 4 + 5 + \ldots + n = \frac{n(n+1)}{2}$$

# Geometric progression

- Other types of pattern we often see are geometric progressions

$$1, 2, 4, 8, 16, 32, 64, 128, \ldots$$

# Geometric progression

- Other types of pattern we often see are geometric progressions

$$1, 2, 4, 8, 16, 32, 64, 128, \ldots$$

- More generally

$$a, ar, ar^2, ar^3, ar^4, ar^5, ar^6, \ldots$$

$$a_n = ar^{n-1}$$

# Geometric progression

- Sum over a finite portion

$$\sum_{i=0}^{n} ar^i = \frac{a(1 - r^n)}{(1 - r)}$$

# Geometric progression

- Sum over a finite portion

$$\sum_{i=0}^{n} ar^i = \frac{a(1 - r^n)}{(1 - r)}$$

- Or from the $m$-th element to the $n$-th

$$\sum_{i=m}^{n} ar^i = \frac{a(r^m - r^{n+1})}{(1 - r)}$$

# Little bit about logarithm

- Sometimes doing computation in logarithm can be an efficient alternative.

# Little bit about logarithm

- Sometimes doing computation in logarithm can be an efficient alternative.
- In both C++(`<cmath>`) and Java(`java.lang.Math`) we have the natural logarithm

```
double log(double x);
```

# Little bit about logarithm

- Sometimes doing computation in logarithm can be an efficient alternative.
- In both C++(`<cmath>`) and Java(`java.lang.Math`) we have the natural logarithm

  ```
  double log(double x);
  ```

  and logarithm in base 10

  ```
  double log10(double x);
  ```

# Little bit about logarithm

- Sometimes doing computation in logarithm can be an efficient alternative.
- In both C++(`<cmath>`) and Java(`java.lang.Math`) we have the natural logarithm

  ```
  double log(double x);
  ```

  and logarithm in base 10

  ```
  double log10(double x);
  ```

- And also the exponential

  ```
  double exp(double x);
  ```

# Example

- For example, what is the first power of 17 that has $k$ digits in base $b$?

# Example

- For example, what is the first power of 17 that has $k$ digits in base $b$?
- Naive solution: Iterate over powers of 17 and count the number of digits.

# Example

- For example, what is the first power of 17 that has $k$ digits in base $b$?
- Naive solution: Iterate over powers of 17 and count the number of digits.
- But the powers of 17 grow exponentially!

$$17^{16} > 2^{64}$$

- What if $k = 500$ ($\sim 1.7 \cdot 10^{615}$), or something larger?

# Example

- For example, what is the first power of 17 that has $k$ digits in base $b$?
- Naive solution: Iterate over powers of 17 and count the number of digits.
- But the powers of 17 grow exponentially!

$$17^{16} > 2^{64}$$

- What if $k = 500$ ($\sim 1.7 \cdot 10^{615}$), or something larger?
- Impossible to work with the numbers in a normal fashion.
- Why not log?

# Example

- Remember, we can calculate the length of a number $n$ in base $b$ with $\lfloor \log_b(n) \rfloor + 1$.

# Example

- Remember, we can calculate the length of a number $n$ in base $b$ with $\lfloor \log_b(n) \rfloor + 1$.
- But how do we do this with only ln or $\log_{10}$?

# Example

- Remember, we can calculate the length of a number $n$ in base $b$ with $\lfloor \log_b(n) \rfloor + 1$.
- But how do we do this with only ln or $\log_{10}$?
- Change base!
$$\log_b(a) = \frac{\log_d(a)}{\log_d(b)} = \frac{\ln(a)}{\ln(b)}$$
- Now we can at least count the length without converting bases

# Example

- We still have to iterate over the powers of 17, but we can do that in log

$$\ln(17^{x-1} \cdot 17) = \ln(17^{x-1}) + \ln(17)$$

# Example

- We still have to iterate over the powers of 17, but we can do that in log

$$\ln(17^{x-1} \cdot 17) = \ln(17^{x-1}) + \ln(17)$$

- More generally

$$\log_b(xy) = \log_b(x) + \log_b(y)$$

- For division

$$\log_b(\frac{x}{y}) = \log_b(x) - \log_b(y)$$

# Example

- We can simplify this even more.
- The solution to our problem is in mathematical terms, finding the $x$ for

$$\log_b(17^x) = k - 1$$

# Example

- We can simplify this even more.
- The solution to our problem is in mathematical terms, finding the $x$ for

$$\log_b(17^x) = k - 1$$

- One more handy identity

$$\log_b(a^c) = c \cdot \log_b(a)$$

# Example

- We can simplify this even more.
- The solution to our problem is in mathematical terms, finding the $x$ for

$$\log_b(17^x) = k - 1$$

- One more handy identity

$$\log_b(a^c) = c \cdot \log_b(a)$$

- Using this identity and the ones we've covered, we get

$$x = \left\lceil (k-1) \cdot \frac{\ln(10)}{\ln(17)} \right\rceil$$

# Base conversion

- Speaking of bases.

# Base conversion

- Speaking of bases.
- What if we actually need to use base conversion?

# Base conversion

- Speaking of bases.
- What if we actually need to use base conversion?
- Simple algorithm

```
vector<int> toBase(int base, int val) {
    vector<int> res;
    while(val) {
        res.push_back(val % base);
        val /= base;
    }
    return val;
```

- Starts from the 0-th digit, and calculates the multiple of each power.

# Working with doubles

- Comparing doubles, sounds like a bad idea.

# Working with doubles

- Comparing doubles, sounds like a bad idea.
- What else can we do if we are working with real numbers?

# Working with doubles

- Comparing doubles, sounds like a bad idea.
- What else can we do if we are working with real numbers?
- We compare them to a certain degree of precision.

# Working with doubles

- Comparing doubles, sounds like a bad idea.
- What else can we do if we are working with real numbers?
- We compare them to a certain degree of precision.
- Two numbers are deemed equal of their difference is less than some small epsilon.

```
const double EPS = 1e-9;

if (abs(a - b) < EPS) {
...
}
```

# Working with doubles

- Less than operator:

```
if (a < b - EPS) {
...
}
```

- Less than or equal:

```
if (a  < b + EPS) {
...
}
```

- The rest of the operators follow.

# Working with doubles

- This allows us to use comparison based algorithms.

# Working with doubles

- This allows us to use comparison based algorithms.
- For example `std::set<double>`.

```
struct cmp {
    bool operator(){double a, double b}{
        return a < b - EPS;
    }
};

set<double, cmp> doubleSet();
```

- Other STL containers can be used in similar fashion.

# Number Theory

# Modular arithmetic

- Problem statements often end with the sentence

  *"... and output the answer modulo $n$."*

# Modular arithmetic

- Problem statements often end with the sentence

  *"... and output the answer modulo $n$."*

- This implies that we can do all the computation with integers *modulo $n$*.

# Modular arithmetic

- Problem statements often end with the sentence

  *"... and output the answer modulo $n$."*

- This implies that we can do all the computation with integers *modulo $n$*.
- The integers, modulo some $n$ form a structure called a *ring*.

# Modular arithmetic

- Problem statements often end with the sentence

  *"... and output the answer modulo n."*

- This implies that we can do all the computation with integers *modulo $n$*.
- The integers, modulo some $n$ form a structure called a *ring*.
- Special rules apply, also loads of interesting properties.

# Modular arithmetic

Some of the allowed operations:

- Addition and subtraction modulo $n$

$$(a \bmod n) + (b \bmod n) = (a + b \bmod n)$$
$$(a \bmod n) - (b \bmod n) = (a - b \bmod n)$$

- Multiplication

$$(a \bmod n)(b \bmod n) = (ab \bmod n)$$

- Exponentiation

$$(a \bmod n)^b = (a^b \bmod n)$$

- *Note:* We are only working with integers.

# Modular arithmetic

- What about division?

# Modular arithmetic

- What about division? NO!

# Modular arithmetic

- What about division? NO!
- We could end up with a fraction!
- Division with $k$ equals multiplication with the *multiplicative inverse* of $k$.

# Modular arithmetic

- What about division? NO!
- We could end up with a fraction!
- Division with $k$ equals multiplication with the *multiplicative inverse* of $k$.
- The *multiplicative inverse* of an integer $a$, is the element $a^{-1}$ such that

$$a \cdot a^{-1} = 1 \pmod{n}$$

# Modular arithmetic

- What about logarithm?

# Modular arithmetic

- What about logarithm? YES!
  - But difficult.

# Modular arithmetic

- What about logarithm? YES!
    - But difficult.
    - Basis for some cryptography such as elliptic curve, Diffie-Hellmann.
- Google “Discrete Logarithm” if you want to know more.

# Modular arithmetic

- Prime number is a positive integer greater than 1 that has no positive divisor other than 1 and itself.
- Greatest Common Divisor of two integers $a$ and $b$ is the largest number that divides both $a$ and $b$.
- Least Common Multiple of two integers $a$ and $b$ is the smallest integer that both $a$ and $b$ divide.
- Prime factor of an positive integer is a prime number that divides it.
- Prime factorization is the decomposition of an integer into its prime factors. By the fundamental theorem of arithmetics, every integer greater than 1 has a unique prime factorization.

# Extended Euclidean algorithm

- The Euclidean algorithm is a recursive algorithm that computes the GCD of two numbers.

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

- Runs in $O(\log^2 N)$.

# Extended Euclidean algorithm

- The Euclidean algorithm is a recursive algorithm that computes the GCD of two numbers.

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

- Runs in $O(\log^2 N)$.
- Notice that this can also compute LCM

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

- See Wikipedia to see how it works and for proofs.

# Extended Euclidean algorithm

- Reversing the steps of the Euclidean algorithm we get the Bézout's identity

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

  which simply states that there always exist $x$ and $y$ such that the equation above holds.

# Extended Euclidean algorithm

- Reversing the steps of the Euclidean algorithm we get the Bézout's identity
$$\gcd(a, b) = ax + by$$
which simply states that there always exist $x$ and $y$ such that the equation above holds.
- The extended Euclidean algorithm computes the GCD and the coefficients $x$ and $y$.
- Each iteration it add up how much of $b$ we subtracted from $a$ and vice versa.

# Extended Euclidean algorithm

```
int egcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    } else {
        int d = egcd(b, a % b, x, y);
        x -= a / b * y;
        swap(x, y);
        return d;
    }
}
```

# Applications

- Essential step in the RSA algorithm.
- Essential step in many factorization algorithms.
- Can be generalized to other algebraic structures.
- Fundamental tool for proofs in number theory.
- Many other algorithms for GCD

# Primality testing

- How do we determine if a number $n$ is a prime?

# Primality testing

- How do we determine if a number $n$ is a prime?
- Naive method: Iterate over all $1 < i < n$ and check it $i \mid n$.
    - $O(N)$

# Primality testing

- How do we determine if a number $n$ is a prime?
- Naive method: Iterate over all $1 < i < n$ and check it $i \mid n$.
    - $O(N)$
- Better: If $n$ is not a prime, it has a divisor $\leq \sqrt{n}$.
    - Iterate up to $\sqrt{n}$ instead.
    - $O(\sqrt{N})$

# Primality testing

- How do we determine if a number $n$ is a prime?
- Naive method: Iterate over all $1 < i < n$ and check it $i \mid n$.
  - $O(N)$
- Better: If $n$ is not a prime, it has a divisor $\leq \sqrt{n}$.
  - Iterate up to $\sqrt{n}$ instead.
  - $O(\sqrt{N})$
- Even better: If $n$ is not a prime, it has a prime divisor $\leq \sqrt{n}$
  - Iterate over the prime numbers up to $\sqrt{n}$.
  - There are $\sim N/\ln(N)$ primes less $N$, therefore $O(\sqrt{N}/\log N)$.

# Primality testing

- Trial division is a deterministic primality test.
- Many algorithms that are probabilistic or randomized.
- Fermat test; uses Fermat's little theorem.
- Probabilistic algorithms that can only prove that a number is composite such as Miller-Rabin.
- AKS primality test, the one that proved that primality testing is in $P$.

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
    - For all numbers from 2 to $\sqrt{n}$:

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
    - For all numbers from 2 to $\sqrt{n}$:
    - If the number is not marked, iterate over every multiple of the number up to $n$ and mark them.

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
  - For all numbers from 2 to $\sqrt{n}$:
  - If the number is not marked, iterate over every multiple of the number up to $n$ and mark them.
  - The unmarked numbers are those that are not a multiple of any smaller number.
  - $O(\sqrt{N} \log \log N)$

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
  - For all numbers from 2 to $\sqrt{n}$:
  - If the number is not marked, iterate over every multiple of the number up to $n$ and mark them.
  - The unmarked numbers are those that are not a multiple of any smaller number.
  - $O(\sqrt{N} \log \log N)$

| | | | | |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

Primes:

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
    - For all numbers from 2 to $\sqrt{n}$:
    - If the number is not marked, iterate over every multiple of the number up to $n$ and mark them.
    - The unmarked numbers are those that are not a multiple of any smaller number.
    - $O(\sqrt{N} \log \log N)$

| | | | | |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

Primes:

2,

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
  - For all numbers from 2 to $\sqrt{n}$:
  - If the number is not marked, iterate over every multiple of the number up to $n$ and mark them.
  - The unmarked numbers are those that are not a multiple of any smaller number.
  - $O(\sqrt{N} \log \log N)$

| | | | | |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

Primes:

2,

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
  - For all numbers from 2 to $\sqrt{n}$:
  - If the number is not marked, iterate over every multiple of the number up to $n$ and mark them.
  - The unmarked numbers are those that are not a multiple of any smaller number.
  - $O(\sqrt{N} \log \log N)$

| | | | | |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

Primes:

2, 3,

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
    - For all numbers from 2 to $\sqrt{n}$:
    - If the number is not marked, iterate over every multiple of the number up to $n$ and mark them.
    - The unmarked numbers are those that are not a multiple of any smaller number.
    - $O(\sqrt{N} \log \log N)$

| | | | | |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

Primes:

2, 3,

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
    - For all numbers from 2 to $\sqrt{n}$:
    - If the number is not marked, iterate over every multiple of the number up to $n$ and mark them.
    - The unmarked numbers are those that are not a multiple of any smaller number.
    - $O(\sqrt{N} \log \log N)$

| | | | | |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

Primes:

2, 3, 5,

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
  - For all numbers from 2 to $\sqrt{n}$:
  - If the number is not marked, iterate over every multiple of the number up to $n$ and mark them.
  - The unmarked numbers are those that are not a multiple of any smaller number.
  - $O(\sqrt{N} \log \log N)$

| | | | | |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

Primes:

2, 3, 5,

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
    - For all numbers from 2 to $\sqrt{n}$:
    - If the number is not marked, iterate over every multiple of the number up to $n$ and mark them.
    - The unmarked numbers are those that are not a multiple of any smaller number.
    - $O(\sqrt{N} \log \log N)$

| | | | | |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

Primes:

2, 3, 5,

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
    - For all numbers from 2 to $\sqrt{n}$:
    - If the number is not marked, iterate over every multiple of the number up to $n$ and mark them.
    - The unmarked numbers are those that are not a multiple of any smaller number.
    - $O(\sqrt{N} \log \log N)$

| | | | | |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

Primes:

2, 3, 5, 7,

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
    - For all numbers from 2 to $\sqrt{n}$:
    - If the number is not marked, iterate over every multiple of the number up to $n$ and mark them.
    - The unmarked numbers are those that are not a multiple of any smaller number.
    - $O(\sqrt{N} \log \log N)$

| | | | | |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

Primes:

2, 3, 5, 7, 11,

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
  - For all numbers from 2 to $\sqrt{n}$:
  - If the number is not marked, iterate over every multiple of the number up to $n$ and mark them.
  - The unmarked numbers are those that are not a multiple of any smaller number.
  - $O(\sqrt{N} \log \log N)$

| | | | | |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

Primes:

2, 3, 5, 7, 11, 13,

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
  - For all numbers from 2 to $\sqrt{n}$:
  - If the number is not marked, iterate over every multiple of the number up to $n$ and mark them.
  - The unmarked numbers are those that are not a multiple of any smaller number.
  - $O(\sqrt{N} \log \log N)$

| | | | | |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

Primes:

2, 3, 5, 7, 11, 13, 17,

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
  - For all numbers from 2 to $\sqrt{n}$:
  - If the number is not marked, iterate over every multiple of the number up to $n$ and mark them.
  - The unmarked numbers are those that are not a multiple of any smaller number.
  - $O(\sqrt{N} \log \log N)$

| | | | | |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

Primes:

2, 3, 5, 7, 11, 13, 17, 19,

# Prime sieves

- If we want to generate primes, using a primality test is very inefficient.
- Instead, our preferred method of prime generation is the sieve of Eratosthenes.
  - For all numbers from 2 to $\sqrt{n}$:
  - If the number is not marked, iterate over every multiple of the number up to $n$ and mark them.
  - The unmarked numbers are those that are not a multiple of any smaller number.
  - $O(\sqrt{N} \log \log N)$

| | | | | |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

Primes:

2, 3, 5, 7, 11, 13, 17, 19, 23

# Sieve of Eratosthenes

```
vector<int> eratosthenes(int n){
    bool *isMarked = new bool[n+1];
    memset(isMarked, 0, n+1);
    vector<int> primes;
    int i = 2;
    for(; i*i <= n; ++i)
        if (!isMarked[i]) {
            primes.push_back(i);
            for(int j = i; j <= n; j += i)
                isMarked[j] = true;
        }
    for (; i <= n; i++)
        if (!isMarked[i])
            primes.push_back(i);
    return primes;
}
```

# Integer factorization

The fundamental theorem of arithmetic states that

- Every integer greater than 1 is a unique multiple of primes.

# Integer factorization

The fundamental theorem of arithmetic states that

- Every integer greater than 1 is a unique multiple of primes.

$$n = p_1^{e_1} p_2^{e_2} p_3^{e_3} \cdots p_k^{e_k}$$

# Integer factorization

The fundamental theorem of arithmetic states that

- Every integer greater than 1 is a unique multiple of primes.

$$n = p_1^{e_1} p_2^{e_2} p_3^{e_3} \cdots p_k^{e_k}$$

We can therefore store integers as lists of their prime powers.

# Integer factorization

The fundamental theorem of arithmetic states that

- Every integer greater than 1 is a unique multiple of primes.

$$n = p_1^{e_1} p_2^{e_2} p_3^{e_3} \cdots p_k^{e_k}$$

We can therefore store integers as lists of their prime powers.
To factor an integer $n$:

- Use the sieve of Eratosthenes to generate all the primes up $\sqrt{n}$
- Iterate over all the primes generated and check if they divide $n$, and determine the largest power that divides $n$.

```cpp
map<int, int> factor(int N) {
    vector<int> primes;
    primes = eratosthenes(static_cast<int>(sqrt(N+1)));
    map<int, int> factors;
    for(int i = 0; i < primes.size(); ++i){
        int prime = primes[i], power = 0;
        while(N % prime == 0){
            power++;
            N /= prime;
        }
        if(power > 0){
            factors[prime] = power;
        }
    }
    if (N > 1) {
        factors[N] = 1;
    }
    return factors;
}
```

# Integer factorization

The prime factors can be quite useful.

# Integer factorization

The prime factors can be quite useful.

- The number of divisors

$$\sigma_0(n) = \prod_{i=1}^{k}(e_1 + 1)$$

# Integer factorization

The prime factors can be quite useful.

- The number of divisors

$$\sigma_0(n) = \prod_{i=1}^{k} (e_1 + 1)$$

- The sum of all divisors in $x$-th power

$$\sigma_m(n) = \prod_{i=1}^{k} \frac{(p^{(e_i+1)x} - 1)}{(p_i - 1)}$$

# Integer factorization

- The Euler's totient function

$$\phi(n) = n \cdot \prod_{i=1}^{k} (1 - p)$$

# Integer factorization

- The Euler's totient function

$$\phi(n) = n \cdot \prod_{i=1}^{k} (1 - p)$$

- Euler's theorem, if $a$ and $n$ are coprime

$$a^{\phi(n)} = 1 \pmod{n}$$

Fermat's theorem is a special case when $n$ is a prime.

# Combinatorics

# Combinatorics

Combinatorics is study of countable discrete structures.

# Combinatorics

Combinatorics is study of countable discrete structures.

*Generic enumeration problem:* We are given an infinite sequence of sets $A_1, A_2, \dots A_n, \dots$ which contain objects satisfying a set of properties. Determine

$$a_n := |A_n|$$

for general $n$.

# Basic counting

- Factorial

$$n! = 1 \cdot 2 \cdot 3 \cdots n$$

- Binomial coefficient

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

# Basic counting

- Factorial

$$n! = 1 \cdot 2 \cdot 3 \cdots n$$

- Binomial coefficient

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

  Number of ways to choose $k$ objects from a set of $n$ objects, ignoring order.

# Basic counting

Properties

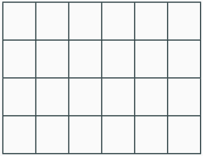
- $$\binom{n}{k} = \binom{n}{n-k}$$
- $$\binom{n}{0} = \binom{n}{n} = 1$$
- $$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Pascal triangle!

$$
\begin{array}{c}
\binom{0}{0} \\
\binom{1}{0} \quad \binom{1}{1} \\
\binom{2}{0} \quad \binom{2}{1} \quad \binom{2}{2} \\
\binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3} \\
\binom{4}{0} \quad \binom{4}{1} \quad \binom{4}{2} \quad \binom{4}{3} \quad \binom{4}{4} \\
\binom{5}{0} \quad \binom{5}{1} \quad \binom{5}{2} \quad \binom{5}{3} \quad \binom{5}{4} \quad \binom{5}{5} \\
\binom{6}{0} \quad \binom{6}{1} \quad \binom{6}{2} \quad \binom{6}{3} \quad \binom{6}{4} \quad \binom{6}{5} \quad \binom{6}{6} \\
\binom{7}{0} \quad \binom{7}{1} \quad \binom{7}{2} \quad \binom{7}{3} \quad \binom{7}{4} \quad \binom{7}{5} \quad \binom{7}{6} \quad \binom{7}{7} \\
\binom{8}{0} \quad \binom{8}{1} \quad \binom{8}{2} \quad \binom{8}{3} \quad \binom{8}{4} \quad \binom{8}{5} \quad \binom{8}{6} \quad \binom{8}{7} \quad \binom{8}{8} \\
\binom{9}{0} \quad \binom{9}{1} \quad \binom{9}{2} \quad \binom{9}{3} \quad \binom{9}{4} \quad \binom{9}{5} \quad \binom{9}{6} \quad \binom{9}{7} \quad \binom{9}{8} \quad \binom{9}{9} \\
\binom{10}{0} \quad \binom{10}{1} \quad \binom{10}{2} \quad \binom{10}{3} \quad \binom{10}{4} \quad \binom{10}{5} \quad \binom{10}{6} \quad \binom{10}{7} \quad \binom{10}{8} \quad \binom{10}{9} \quad \binom{10}{10}
\end{array}
$$

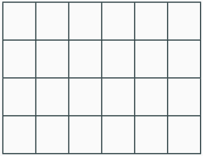
# Example

How many rectangles can be formed on a $m \times n$ grid?



# Example

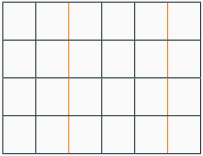
How many rectangles can be formed on a $m \times n$ grid?



- A rectangle needs 4 edges, 2 vertical and 2 horizontal.

# Example

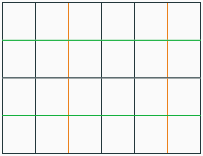
How many rectangles can be formed on a $m \times n$ grid?



- A rectangle needs 4 edges, 2 vertical and 2 horizontal.
    - 2 vertical

# Example

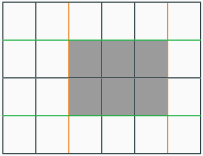
How many rectangles can be formed on a $m \times n$ grid?



- A rectangle needs 4 edges, 2 vertical and 2 horizontal.
    - 2 vertical
    - 2 horizontal

# Example

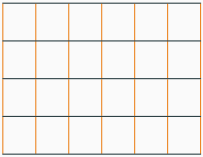
How many rectangles can be formed on a $m \times n$ grid?



- A rectangle needs 4 edges, 2 vertical and 2 horizontal.
    - 2 vertical
    - 2 horizontal

# Example

How many rectangles can be formed on a $m \times n$ grid?



- A rectangle needs 4 edges, 2 vertical and 2 horizontal.
  - 2 vertical
  - 2 horizontal
- Number of ways we can choose 2 vertical lines

$$\binom{n}{2}$$

# Example

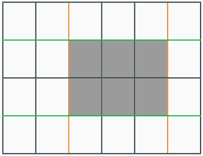
How many rectangles can be formed on a $m \times n$ grid?

- A rectangle needs 4 edges, 2 vertical and 2 horizontal.
  - 2 vertical
  - 2 horizontal
- Number of ways we can choose 2 horizontal lines

$$\binom{m}{2}$$

# Example

How many rectangles can be formed on a $m \times n$ grid?



- A rectangle needs 4 edges, 2 vertical and 2 horizontal.
    - 2 vertical
    - 2 horizontal
- Total number of ways we can form a rectangle

$$\binom{n}{2}\binom{m}{2} = \frac{n!m!}{(n-2)!(m-2)!2!2!}$$
$$= \frac{n(n-1)m(m-1)}{4}$$

# Multinomial

What if we have many objects with the same value?

# Multinomial

What if we have many objects with the same value?

- Number of permutations on $n$ objects, where $n_i$ is the number of objects with the $i$-th value.(Multinomial)

$$\binom{n}{n_1, n_2, \ldots, n_k} = \frac{n!}{n_1! n_2! \cdots n_k!}$$

# Multinomial

What if we have many objects with the same value?

- Number of permutations on $n$ objects, where $n_i$ is the number of objects with the $i$-th value.(Multinomial)

$$\binom{n}{n_1, n_2, \ldots, n_k} = \frac{n!}{n_1! n_2! \cdots n_k!}$$

- Number of way to choose $k$ objects from a set of $n$ objects with, where each value can be chosen more than once.

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

# Example

How many different ways can we divide $k$ identical balls into $n$ boxes?

# Example

How many different ways can we divide $k$ identical balls into $n$ boxes?

- Same as number of nonnegative solutions to

$$x_1 + x_2 + \ldots + x_n = k$$

# Example

How many different ways can we divide $k$ identical balls into $n$ boxes?

- Same as number of nonnegative solutions to

$$x_1 + x_2 + \ldots + x_n = k$$

- Let's imagine we have a bit string consisting only of 1 of length $n + k - 1$

$$\underbrace{1\,1\,1\,1\,1\,1\,1 \ldots 1}_{n+k-1}$$

# Example

- Choose $n-1$ bits to be swapped for 0

$$1 \ldots 1\,0\,1 \ldots 1\,0 \ldots 0\,1 \ldots 1$$

# Example

- Choose $n-1$ bits to be swapped for 0

$$\underbrace{1\ldots1}_{x_1}0\underbrace{1\ldots1}_{x_2}0\ldots0\underbrace{1\ldots1}_{x_n}$$

- Then total number of 1 will be $k$, each 1 representing an each element, and separated into $n$ groups

# Example

- Choose $n-1$ bits to be swapped for 0

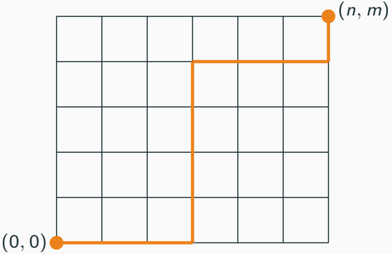
$$\underbrace{1\ldots1}_{x_1}0\underbrace{1\ldots1}_{x_2}0\ldots0\underbrace{1\ldots1}_{x_n}$$

- Then total number of 1 will be $k$, each 1 representing an each element, and separated into $n$ groups
- Number of ways to choose the bits to swap

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

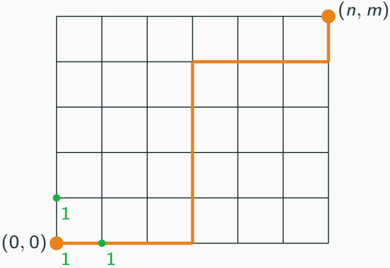
# Example

How many different lattice paths are there from $(0, 0)$ to $(n, m)$?



# Example

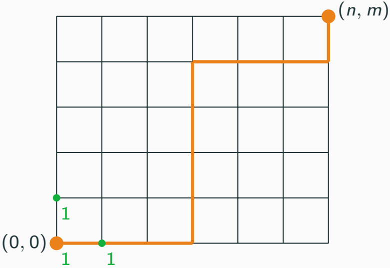
How many different lattice paths are there from $(0, 0)$ to $(n, m)$?




- There is 1 path to $(0, 0)$

# Example

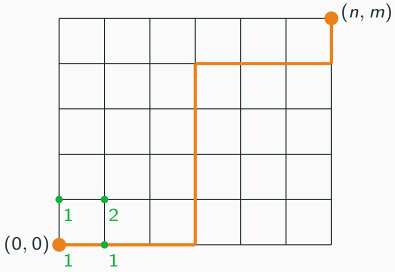
How many different lattice paths are there from $(0,0)$ to $(n,m)$?




- There is 1 path to $(0,0)$
- There is 1 path to $(1,0)$ and $(0,1)$

# Example

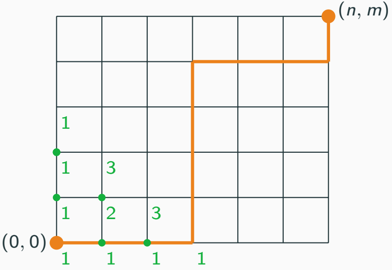
How many different lattice paths are there from $(0, 0)$ to $(n, m)$?




- There is 1 path to $(0, 0)$
- There is 1 path to $(1, 0)$ and $(0, 1)$
- Paths to $(1, 1)$ is the sum of number of paths to $(0, 1)$ and $(1, 0)$.

# Example

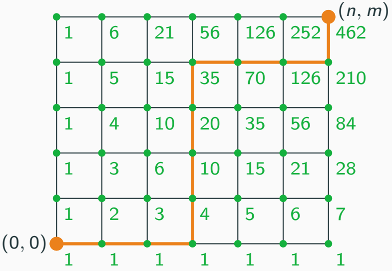
How many different lattice paths are there from $(0, 0)$ to $(n, m)$?




- There is 1 path to $(0, 0)$
- There is 1 path to $(1, 0)$ and $(0, 1)$
- Paths to $(1, 1)$ is the sum of number of paths to $(0, 1)$ and $(1, 0)$.
- Number of paths to $(i, j)$ is the sum of the number of paths to $(i - 1, j)$ and $(i, j - 1)$.

# Example

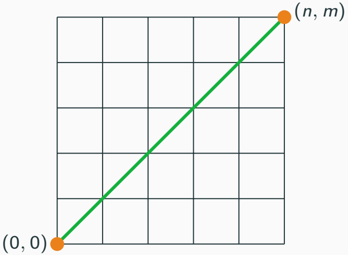
How many different lattice paths are there from $(0, 0)$ to $(n, m)$?




- There is 1 path to $(0, 0)$
- There is 1 path to $(1, 0)$ and $(0, 1)$
- Paths to $(1, 1)$ is the sum of number of paths to $(0, 1)$ and $(1, 0)$.
- Number of paths to $(i, j)$ is

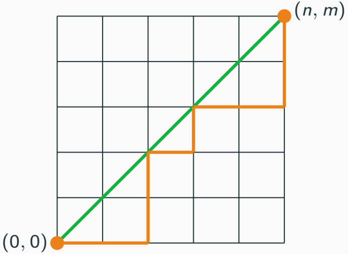
$$\binom{i+j}{i}$$

What if we are not allowed to cross the main diagonal?



# Catalan

What if we are not allowed to cross the main diagonal?

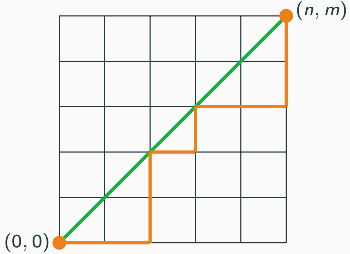



- The number of paths from $(0, 0)$ to $(n, m)$

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

# Catalan

What if we are not allowed to cross the main diagonal?




- The number of paths from $(0, 0)$ to $(n, m)$

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

- $C_n$ are known as Catalan numbers.
- Many problems involve solutions given by the Catalan numbers.

# Catalan

- Number of different ways $n + 1$ factors can be completely parenthesized.

$$((ab)c)d \quad (a(bc))d \quad (ab)(cd) \quad a((bc)d) \quad a(b(cd))$$

# Catalan

- Number of different ways $n + 1$ factors can be completely parenthesized.

$$((ab)c)d \quad (a(bc))d \quad (ab)(cd) \quad a((bc)d) \quad a(b(cd))$$

- Number of stack sortable permutations of length $n$.

# Catalan

- Number of different ways $n+1$ factors can be completely parenthesized.

$$((ab)c)d \quad (a(bc))d \quad (ab)(cd) \quad a((bc)d) \quad a(b(cd))$$

- Number of stack sortable permutations of length $n$.
- Number of different triangulations convex polygon with $n+2$ sides

# Catalan

- Number of different ways $n+1$ factors can be completely parenthesized.

$$((ab)c)d \quad (a(bc))d \quad (ab)(cd) \quad a((bc)d) \quad a(b(cd))$$

- Number of stack sortable permutations of length $n$.
- Number of different triangulations convex polygon with $n+2$ sides

- Number of full binary trees with $n+1$ leaves.

# Catalan

- Number of different ways $n+1$ factors can be completely parenthesized.

$$((ab)c)d \quad (a(bc))d \quad (ab)(cd) \quad a((bc)d) \quad a(b(cd))$$

- Number of stack sortable permutations of length $n$.
- Number of different triangulations convex polygon with $n+2$ sides

- Number of full binary trees with $n+1$ leaves.
- And aloot more.

# Fibonacci

The Fibonacci sequence is defined recursively as

$$f_1 = 1$$
$$f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

# Fibonacci

The Fibonacci sequence is defined recursively as

$$\begin{aligned} f_1 &= 1 \\ f_2 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \end{aligned}$$

Already covered how to calculate $f_n$ in $O(N)$ time with dynamic programming.

# Fibonacci

The Fibonacci sequence is defined recursively as

$$
\begin{aligned}
f_1 &= 1 \\
f_2 &= 1 \\
f_n &= f_{n-1} + f_{n-2}
\end{aligned}
$$

Already covered how to calculate $f_n$ in $O(N)$ time with dynamic programming.
But we can do even better.

# Fibonacci as matrix

The Fibonacci sequence can be represented by a vectors

$$\begin{pmatrix} f_{n+2} \\ f_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix}$$

# Fibonacci as matrix

The Fibonacci sequence can be represented by a vectors

$$\begin{pmatrix} f_{n+2} \\ f_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix}$$

Or simply as a matrix

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$$

# Fibonacci as matrix

The Fibonacci sequence can be represented by a vectors

$$\begin{pmatrix} f_{n+2} \\ f_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix}$$

Or simply as a matrix

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$$

Using fast exponentiaton, we can calculate $f_n$ in $O(\log N)$ time.

# Fibonacci as matrix

Any linear recurrence

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} \dots c_k a_{n-k}$$

can be expressed in the same way

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

# Fibonacci as matrix

Any linear recurrence

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} \dots c_k a_{n-k}$$

can be expressed in the same way

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

With a recurrence relation defined as a linear function of the $k$ preceding terms the running time will be $O(k^3 \log N)$.

# Game Theory

# Game theory

Game theory is the study of strategic decision making, but in competitive programming we are mostly interested in combinatorial games.

# Game theory

Game theory is the study of strategic decision making, but in competitive programming we are mostly interested in combinatorial games.
*Example*:

- There is a pile of $k$ matches.
- Player can remove 1, 2 or 3 from the pile and alternate on moves.
- The player who removes the last match wins.
- There are two players, and the first player starts.
- Assuming that both players play perfectly, who wins?

# Example

We can analyse these types of games with *backward induction*.

# Example

We can analyse these types of games with *backward induction*.
We call a state $N$-position if it is a winning state for the next player to move, and $P$-position if it is a winning position for the previous player.

- All terminal positions are $P$-positions.
- If every reachable state from the current one is a $N$-position then the current state is a $P$-position.
- If at least one $P$-position can be reached from the current one, then the current state is a $N$-position.
- A position is a $P$-position if all reachable states form the current one are $N$ position.

# Example

Let's analyse our previous game.

# Example

Let's analyse our previous game.

- The terminal position is a $P$-position.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P$ | | | | | | | | | | | | | |

# Example

Let's analyse our previous game.

- The terminal position is a $P$-position.
- The positions reachable from the terminal positions are $N$-positions.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P$ | $N$ | $N$ | $N$ | | | | | | | | | | |

# Example

Let's analyse our previous game.

- The terminal position is a $P$-position.
- The positions reachable from the terminal positions are $N$-positions.
- Position 4 can only reach $N$-positions, therefore a $P$ position.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P$ | $N$ | $N$ | $N$ | $P$ | | | | | | | | | |

# Example

Let's analyse our previous game.

- The terminal position is a $P$-position.
- The positions reachable from the terminal positions are $N$-positions.
- Position 4 can only reach $N$-positions, therefore a $P$ position.
- The next 3 positions can reach the $P$-position 4, therefore they are $N$-positions.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P$ | $N$ | $N$ | $N$ | $P$ | $N$ | $N$ | $N$ | | | | | | |

# Example

Let's analyse our previous game.

- The terminal position is a $P$-position.
- The positions reachable from the terminal positions are $N$-positions.
- Position 4 can only reach $N$-positions, therefore a $P$ position.
- The next 3 positions can reach the $P$-position 4, therefore they are $N$-positions.
- And so on.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P$ | $N$ | $N$ | $N$ | $P$ | $N$ | $N$ | $N$ | $P$ | $N$ | $N$ | $N$ | $P$ | ... |

# Game theory

We can see a clear pattern of the $N$ and $P$ positions in the previous game. – Easy to prove that a position is $P$ if $x \equiv 0 \pmod{4}$.

# Game theory

We can see a clear pattern of the $N$ and $P$ positions in the previous game. – Easy to prove that a position is $P$ if $x \equiv 0 \pmod 4$.

- Many games can be analyzed this way.
- Not only one dimensional games.

We can see a clear pattern of the $N$ and $P$ positions in the previous game. – Easy to prove that a position is $P$ if $x \equiv 0 \pmod 4$.

- Many games can be analyzed this way.
- Not only one dimensional games.
- What if there are $n$ piles instead of 1?

We can see a clear pattern of the $N$ and $P$ positions in the previous game. – Easy to prove that a position is $P$ if $x \equiv 0 \pmod 4$.

- Many games can be analyzed this way.
- Not only one dimensional games.
- What if there are $n$ piles instead of 1?
- What if we can remove 1, 3 or 4?

# The game called Nim

- There are $n$ piles, each containing $x_i$ chips.
- Player can remove from exactly one pile, and can remove any number of chips.
- The player who removes the last match wins.
- There are two players, and the first player starts and they alternate on moves.
- Assuming that both players play perfectly, who wins?

# The game called Nim

Nim can be analyzed with $N$ and $P$ position.

# The game called Nim

Nim can be analyzed with $N$ and $P$ position.

- Not trivial to generalize over $n$ piles.

# The game called Nim

Nim can be analyzed with $N$ and $P$ position.

- Not trivial to generalize over $n$ piles.
- Luckily someone has already done that for us.

# The game called Nim

Nim can be analyzed with $N$ and $P$ position.

- Not trivial to generalize over $n$ piles.
- Luckily someone has already done that for us.

*Buton's theorem* states that a position $(x_1, x_2, \ldots, x_n)$ in Nim is a $P$-position if and only if the xor over the number of chips is 0.

# The game called Nim

Nim can be analyzed with $N$ and $P$ position.

- Not trivial to generalize over $n$ piles.
- Luckily someone has already done that for us.

*Buton's theorem* states that a position $(x_1, x_2, \ldots, x_n)$ in Nim is a $P$-position if and only if the xor over the number of chips is 0.
This theorem extends to other sums of combinatorial games!