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 $O(\log^2 N)$.