# Mathematical Introduction

Atli Fannar Franklín, Arnar Bjarni Arnarson

**Árangursrík forritun og lausn verkefna**

School of Computer Science

Reykjavík University

# Important point

$$\text{Computer Science} \subset \text{Mathematics}$$

- Problems often require mathematical analysis to be solved efficiently.
- Using a bit of math before coding can also shorten and simplify code.
- We will now go over various bits and pieces from mathematics that are useful to know.

# Pattern finding

- Some problems have solutions that form a pattern.

# Pattern finding

- 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.

# Pattern finding

- 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.

# Pattern finding

- 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?

# Pattern finding

- 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.

# Pattern finding

- 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 an 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

$$s, sr, sr^2, sr^3, sr^4, sr^5, sr^6, \ldots$$

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

# Geometric progression

- Sum over a finite portion

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

# Geometric progression

- Sum over a finite portion

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

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

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

# Quick note on logarithms

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

# Quick note on logarithms

- 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);
```

# Quick note on logarithms

- 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);
```

# Quick note on logarithms

- 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

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

# 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

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

- 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 like in binary search.

# 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 like in binary search.
- Two numbers are deemed equal if 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.