

# 明日使えないすごいビット演算

KMC1回生 `prime(Twitter id:@_primenumber)`

# ビット演算とは

- コンピューター内で数値や文字列などのデータは2進数で記録されている
- ビット演算とは、2進数を0/1の列として操作するような演算のこと
- ビット反転 (C言語では  $\sim x$ )
  - 各ビットの0/1を反転させる

x	0	1	1	0	1	0	1	1
								
$\sim x$	1	0	0	1	0	1	0	0

# ビット演算とは

- ビット論理和 (C言語では `x|y`)
  - 各桁を比較して、少なくとも一方が1なら1

x	0	0	1	0	1	0	1	1
---	---	---	---	---	---	---	---	---


y	1	1	0	0	1	0	0	1
---	---	---	---	---	---	---	---	---



x y	1	1	1	0	1	0	1	1
-----	---	---	---	---	---	---	---	---


# ビット演算とは

- ビット論理積 (C言語では `x&y`)
  - 各桁を比較して、両方とも1なら1

x	0	0	1	0	1	0	1	1
y	1	1	0	0	1	0	0	1
								
x&y	0	0	0	0	1	0	0	1

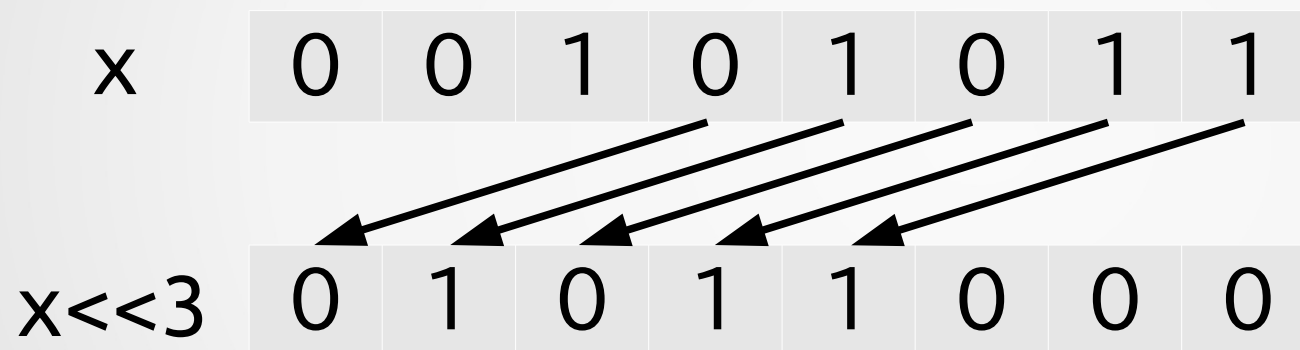
# ビット演算とは

- ビット排他的論理和 (C言語では  $x \oplus y$ )
  - 各桁を比較して、片方のみが1なら1

x	0	0	1	0	1	0	1	1
y	1	1	0	0	1	0	0	1
								
$x \oplus y$	1	1	1	0	0	0	1	0

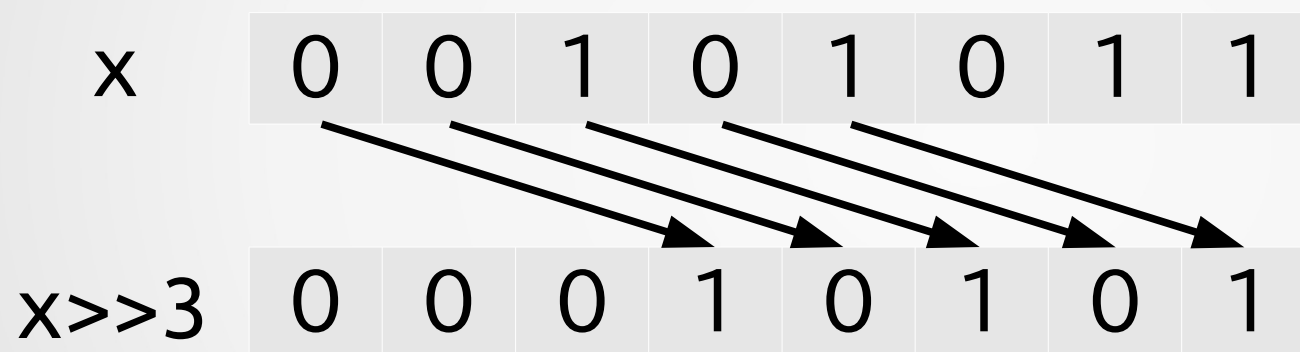
# ビット演算とは

- 左ビットシフト (C言語では  $x \ll n$ )
  - 各桁を左に指定した桁数ずらす



# ビット演算とは

- 右ビットシフト (C言語では  $x \gg n$ )
  - 各桁を右に指定した桁数ずらす



上位桁に何を詰めるかによっていくつか種類がある

- 0を詰める
- 元の最上位桁と同じ物を詰める

# ビット演算とは

- ビット演算は回路が単純になるため、とても高速なことが多い
  - とはいえ最近のCPUだと加減乗算も同じぐらい速い
  - 組み合わせて使うことも多い
- うまく使うとものすごい高速化できる
  - 単純な実装に比べて数十倍速くなることも
- 今回はビット演算を用いていろいろな操作を高速にする例を挙げます
- 数値は2の補数表現で格納されているものとします



# 明日使えないすごいビット演算

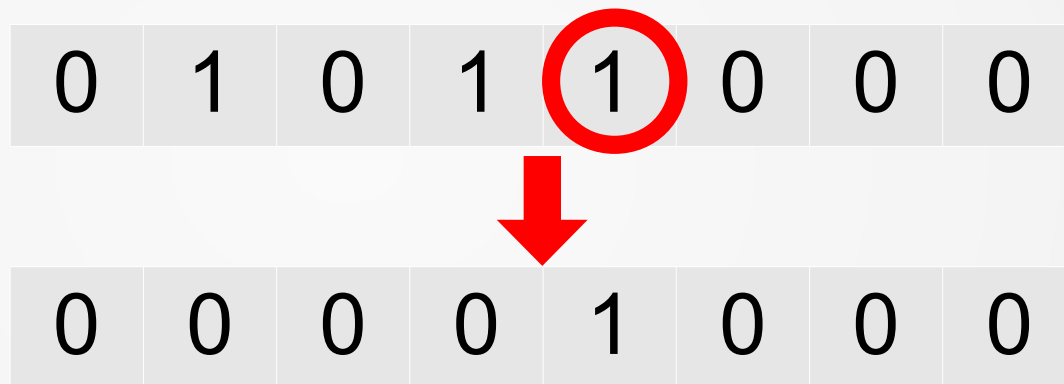
- 「1になっている一番下の桁」を取得する
  - 2の何乗で割り切れるか, みたいなことが分かったりする

data	0	1	0	1	1	0	0	0
------	---	---	---	---	---	---	---	---

## 明日使えないすごいビット演算

- 「1になっている一番下の桁」を取得する
  - 2の何乗で割り切れるか, みたいなことが分かったりする

data



## 明日使えないすごいビット演算

- 「1になっている一番下の桁」を取得する  
 $\text{data} \ \& \ (-\text{data})$

data	0	1	0	1	1	0	0	0
------	---	---	---	---	---	---	---	---

-data	1	0	1	0	1	0	0	0
-------	---	---	---	---	---	---	---	---

実は,  $-\text{data}$  は  $\sim\text{data}+1$  に等しい  
(足して0になるようにするため)

## 明日使えないすごいビット演算

- 「1になっている一番下の桁」を取得する  
 $\text{data} \ \& \ (\sim \text{data})$

data	0	1	0	1	1	0	0	0
------	---	---	---	---	---	---	---	---

-data	1	0	1	0	1	0	0	0
-------	---	---	---	---	---	---	---	---

data & (~data)	0	0	0	0	1	0	0	0
----------------	---	---	---	---	---	---	---	---

実は, -dataは $\sim \text{data} + 1$ に等しい  
(足して0になるため)

## 明日使えないすごいビット演算

- 「1になっている一番下の桁」を0にする

`data &= data-1`

data	0	1	0	1	1	0	0	0
------	---	---	---	---	---	---	---	---

data-1	0	1	0	1	0	1	1	1
--------	---	---	---	---	---	---	---	---

data & data-1	0	1	0	1	0	0	0	0
---------------	---	---	---	---	---	---	---	---

# 明日使えないすごいビット演算

- 「1になっている一番上の桁」を求める
  - 数値のだいたいの大きさを求める
  - $\log_2(n)$ の整数部分を求めるのに使える
- これは一発では行かないが、うまい方法がある

# 明日使えないすごいビット演算

- 「1になっている一番上の桁」を求める
  - 数値のだいたいの大きさを求める
  - $\log_2(n)$ の整数部分を求めるのに使える
- これは一発では行かないが、うまい方法がある
  - 二分探索!

# 明日使えないすごいビット演算

- 「1になっている一番上の桁」を求める

	0	1	0	1	1	0	0	1
0xF0	1	1	1	1	0	0	0	0
	0	1	0	1	0	0	0	0

↓ ビット論理積



# 明日使えないすごいビット演算

- 「1になっている一番上の桁」を求める

	0	1	0	1	1	0	0	1
0xF0	1	1	1	1	0	0	0	0



ビット論理積

0	1	0	1	0	0	0	0	!= 0
---	---	---	---	---	---	---	---	------

# 明日使えないすごいビット演算

- 「1になっている一番上の桁」を求める

	0	1	0	1	1	0	0	1
0xF0	1	1	1	1	0	0	0	0
					↓			
					ビット論理積			
	0	1	0	1	0	0	0	0
								!= 0

1になっている一番上の桁は上位4桁のどれか!

# 明日使えないすごいビット演算

- 「1になっている一番上の桁」を求める



# 明日使えないすごいビット演算

- 「1になっている一番上の桁」を求める



1になっている一番上の桁は上位2桁のどれか!

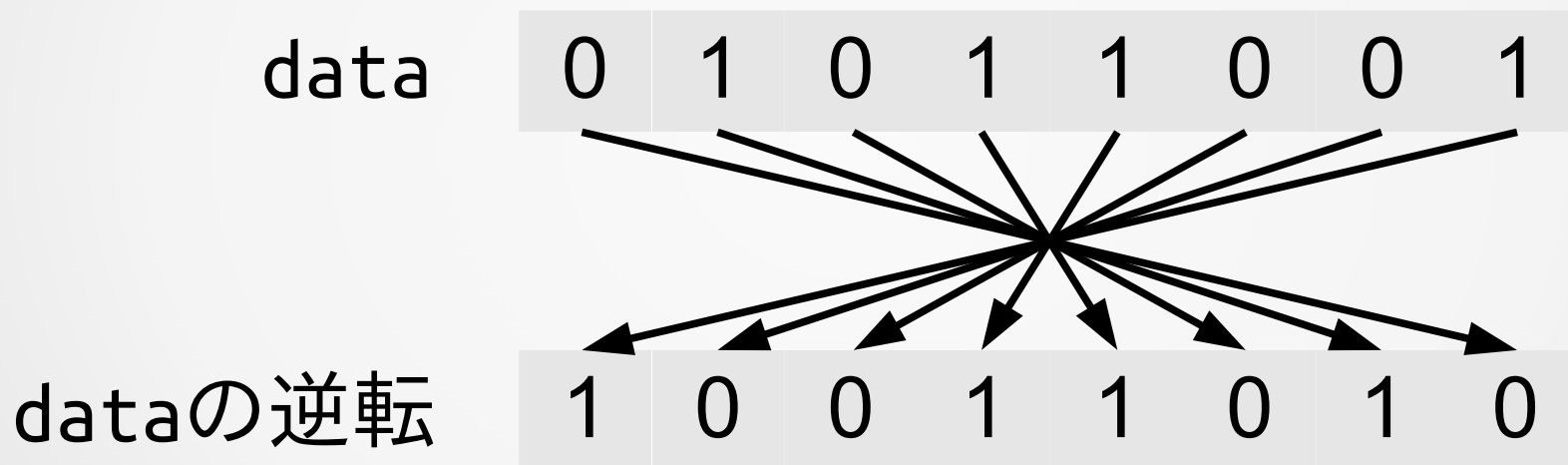
# 明日使えないすごいビット演算

- 「1になっている一番上の桁」を求める
  - サンプルコード(32ビット)

```
data = data & 0xFFFF0000 ? data & 0xFFFF0000 : data;  
data = data & 0xFF00FF00 ? data & 0xFF00FF00 : data;  
data = data & 0xF0F0F0F0 ? data & 0xF0F0F0F0 : data;  
data = data & 0xCCCCCCCC ? data & 0xCCCCCCCC : data;  
data = data & 0xAAAAAAAA ? data & 0xAAAAAAAA : data;
```

# 明日使えないすごいビット演算

- ビット列を逆転する
  - 高速フーリエ変換などで用いる



# 明日使えないすごいビット演算

- ビット列を逆転する
  - これも一気にやるのは無理

data	0	1	0	1	1	0	0	1
data&0x55		1		1		0		1
data&0xAA	0		0		1		0	

# 明日使えないすごいビット演算

- ビット列を逆転する

data	0	1	0	1	1	0	0	1
$(data \& 0x55) \ll 1$	1		1		0		1	
$(data \& 0xAA) \gg 1$		0		0		1		0



# 明日使えないすごいビット演算

- ビット列を逆転する

data	0	1	0	1	1	0	0	1
$(data \& 0x55) \ll 1$	1		1		0		1	
$(data \& 0xAA) \gg 1$		0		0		1		0
 ビット論理和								
	1	0	1	0	0	1	1	0

# 明日使えないすごいビット演算

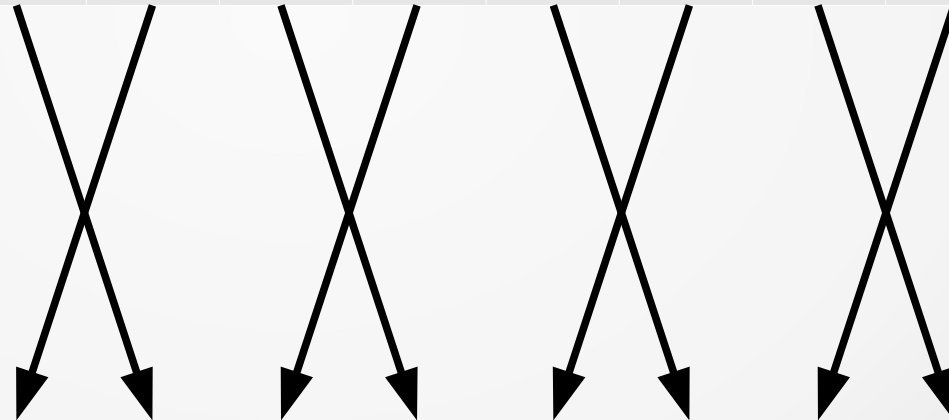
- ビット列を逆転する

変更前のdata

0	1	0	1	1	0	0	1
---	---	---	---	---	---	---	---

変更後のdata

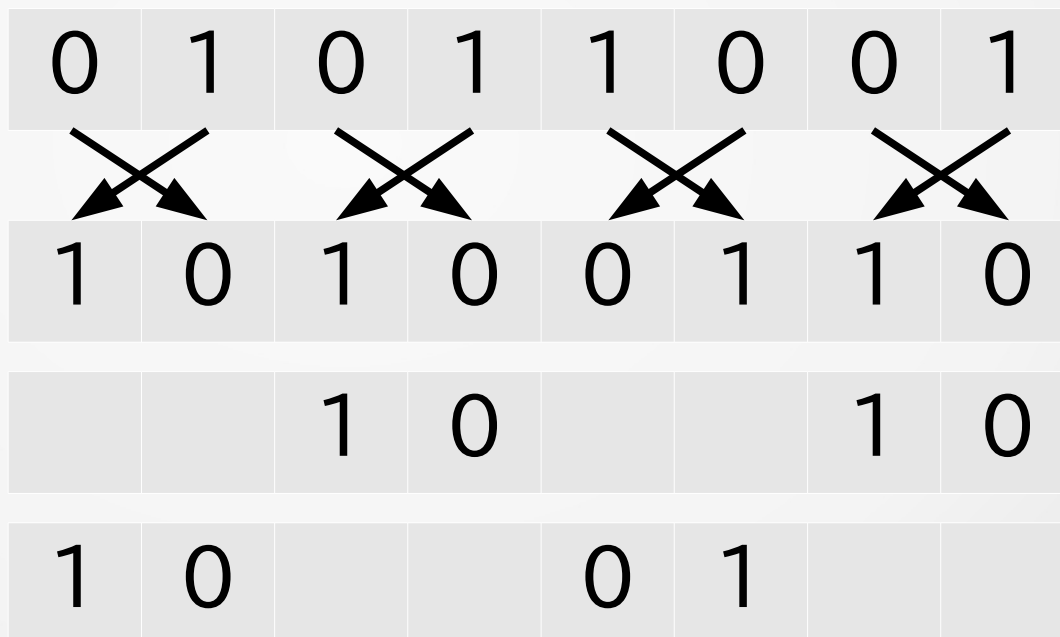
1	0	1	0	0	1	1	0
---	---	---	---	---	---	---	---



# 明日使えないすごいビット演算

- ビット列を逆転する

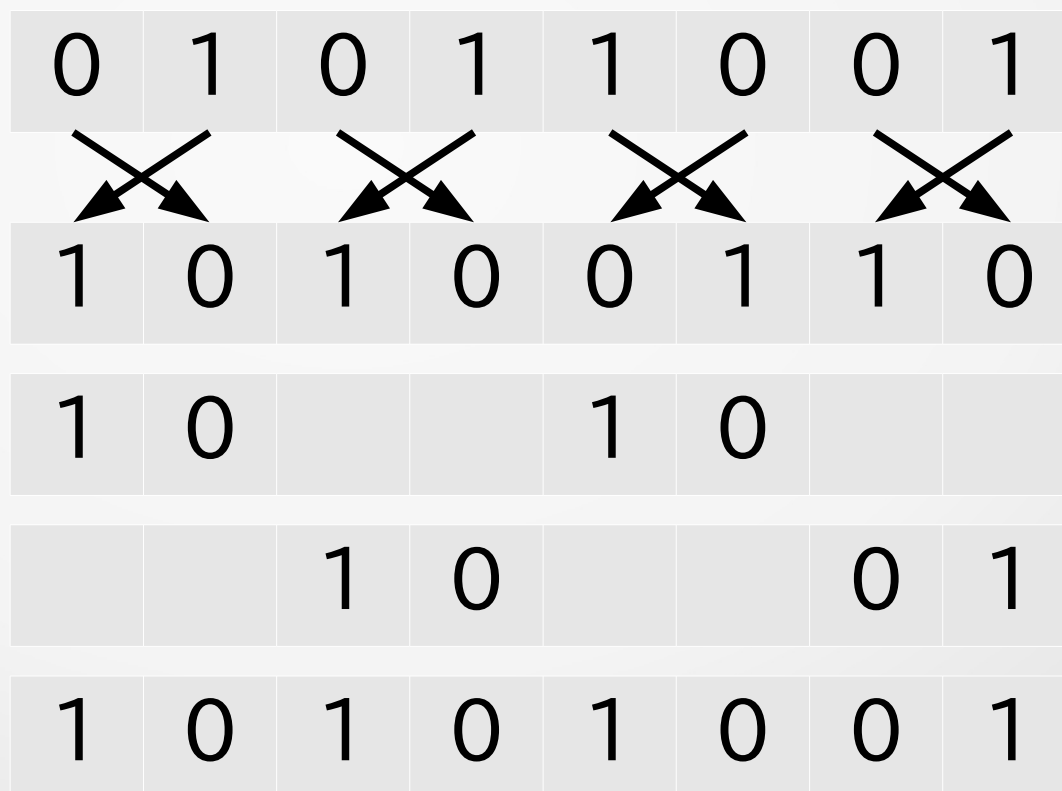
変更後のdata



# 明日使えないすごいビット演算

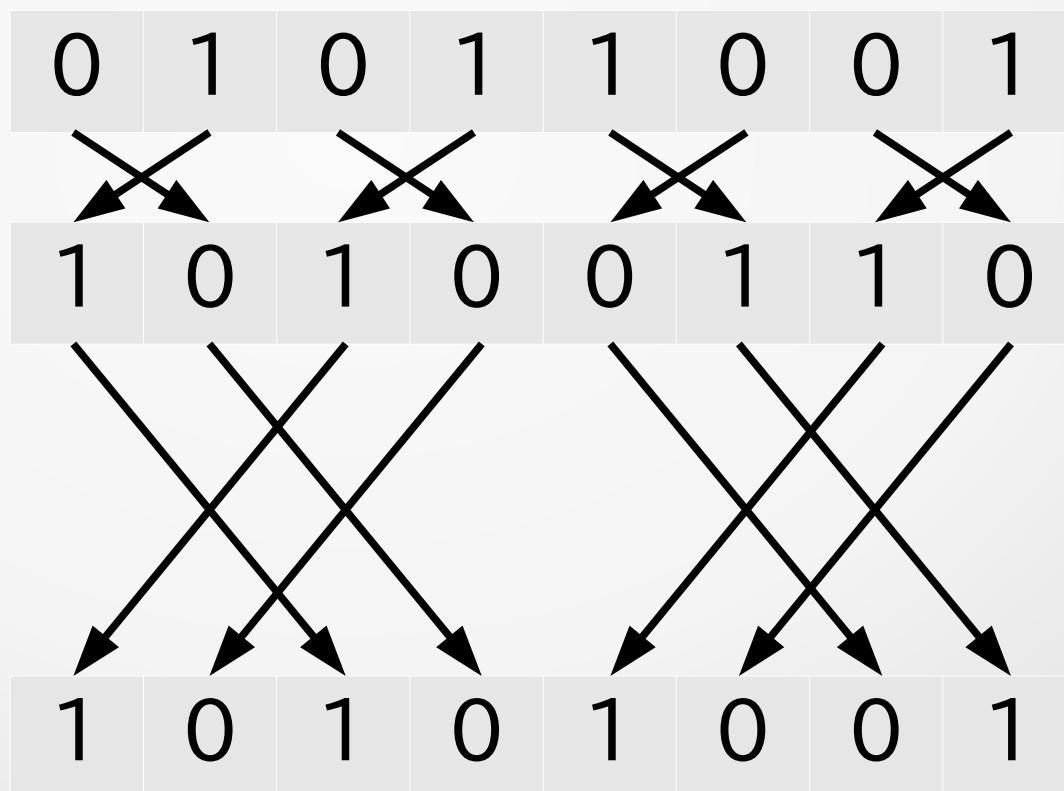
- ビット列を逆転する

変更後のdata



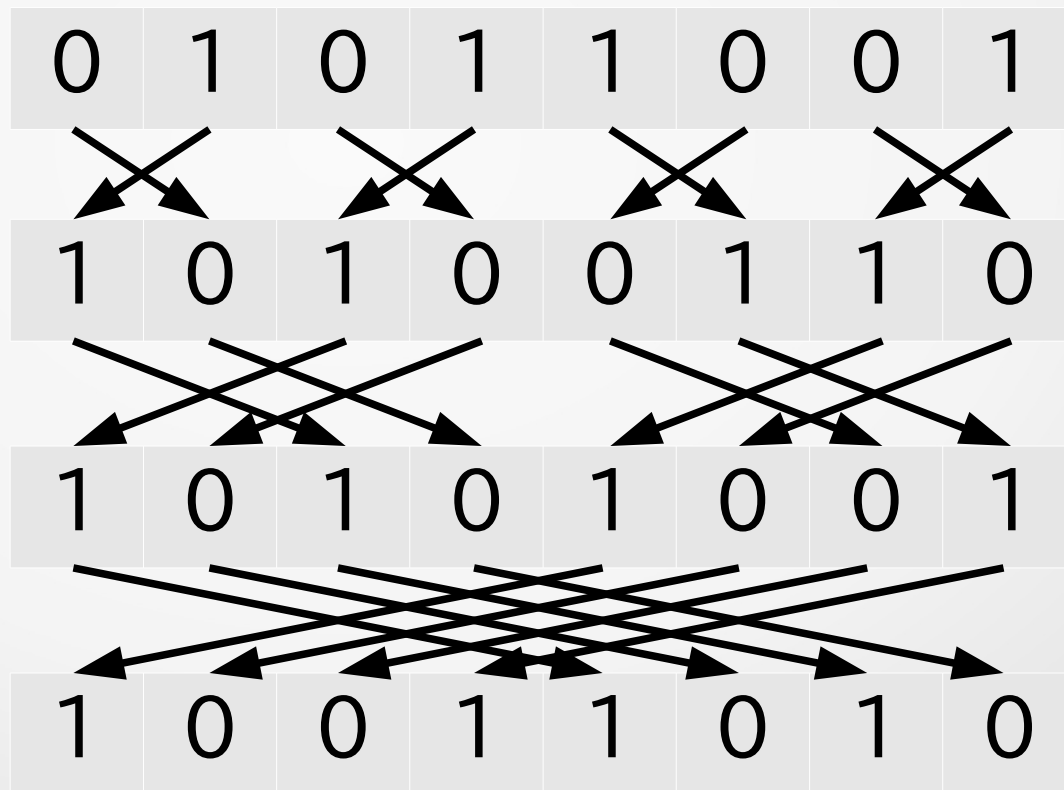
# 明日使えないすごいビット演算

- ビット列を逆転する



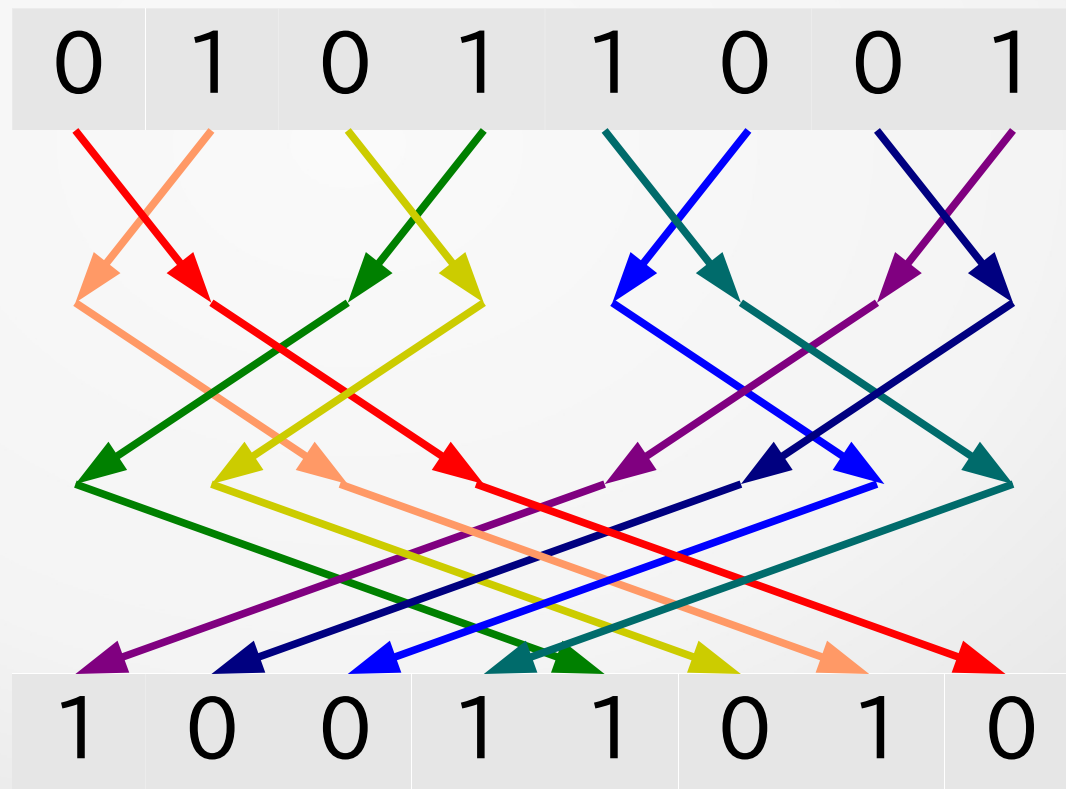
# 明日使えないすごいビット演算

- ビット列を逆転する



# 明日使えないすごいビット演算

- ビット列を逆転する



# 明日使えないすごいビット演算

- ビット列を逆転する
- dataは32ビット符号なし型とする

```
data = ((data & 0x55555555) << 1)
      | ((data & 0xAA555555) >> 1);
data = ((data & 0x33333333) << 2)
      | ((data & 0xCCCCCCCC) >> 2);
data = ((data & 0x0F0F0F0F) << 4)
      | ((data & 0xF0F0F0F0) >> 4);
data = ((data & 0x00FF00FF) << 8)
      | ((data & 0xFF00FF00) >> 8);
data = (data << 16) | (data >> 16);
```



# 明日使えないすごいビット演算

- 1になっているビットの数を数える
- ビットレベルでハミング距離を取りたい時などに使う
- 素直な実装(int型を32bitと仮定)

```
int count = 0;
for (int i = 0; i < 32; i++) {
    count += (data >> i) & 1;
}
```

# 明日使えないすごいビット演算

- 1になっているビットの数を数える
- ちょっと速い実装

```
int count = 0;
for(; data; data &= data - 1) {
    ++count;
}
```

data &= data - 1で1になっている一番小さい桁が0になる

# 明日使えないすごいビット演算

- 1になっているビットの数を数える
- けっこう速い実装

10進数	2進数	1の個数
0	00000000	0
1	00000001	1
2	00000010	1
3	00000011	2
4	00000100	1
...	...	...
255	11111111	8

あらかじめ0~255までの数について1の個数を数えて配列に入れておく

# 明日使えないすごいビット演算

- 1になっているビットの数を数える
- けっこう速い実装

```
int count = 0;  
count += table[data & 0xFF];  
count += table[(data >> 8) & 0xFF];  
count += table[(data >> 16) & 0xFF];  
count += table[(data >> 24) & 0xFF];
```

table[256] : 1の個数が入った配列

# 明日使えないすごいビット演算

- 1になっているビットの数を数える
- 配列を使った実装はけっこう速い
  - 素直な方法の20倍くらい

# 明日使えないすごいビット演算

- 1になっているビットの数を数える
- 配列を使った実装はけっこう速い
  - 素直な方法の20倍くらい
- しかし、さらに倍くらい速い実装が存在する

## 明日使えないすごいビット演算

1	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

各桁の0/1を「その桁の1の個数」と読み替えることができる

# 明日使えないすごいビット演算

1個	0個	1個	1個	0個	1個	0個	0個
----	----	----	----	----	----	----	----

各桁の0/1を「その桁の1の個数」と読み替えることができる



# 明日使えないすごいビット演算

	1個	0個	1個	1個	0個	1個	0個	0個
0x55	0	1	0	1	0	1	0	1
								
		0個		1個		1個		0個

ビット論理積

# 明日使えないすごいビット演算

	1個	0個	1個	1個	0個	1個	0個	0個
0xAA	1	0	1	0	1	0	1	0
								
	1個		1個		0個		0個	

ビット論理積

# 明日使えないすごいビット演算

1個	0個	1個	1個	0個	1個	0個	0個
----	----	----	----	----	----	----	----



&0xAA	1個		1個		0個		0個	
&0x55		0個		1個		1個		0個

## 明日使えないすごいビット演算

1個	0個	1個	1個	0個	1個	0個	0個
----	----	----	----	----	----	----	----



	1個		1個		0個		0個
	0個		1個		1個		0個

## 明日使えないすごいビット演算

1個	0個	1個	1個	0個	1個	0個	0個
----	----	----	----	----	----	----	----



	1個		1個		0個		0個
	0個		1個		1個		0個



足し算

0	1個	1	0個	0	1個	0	0個
---	----	---	----	---	----	---	----

2桁ごとに「その2桁にあった1の数」が格納された!!!

# 明日使えないすごいビット演算

1個	0個	1個	1個	0個	1個	0個	0個
----	----	----	----	----	----	----	----



0	1個	1	0個	0	1個	0	0個
---	----	---	----	---	----	---	----

&0x33

		1	0個			0	0個
--	--	---	----	--	--	---	----

&0xCC

0	1個			0	1個		
---	----	--	--	---	----	--	--

## 明日使えないすごいビット演算

1個	0個	1個	1個	0個	1個	0個	0個
----	----	----	----	----	----	----	----



0	1個	1	0個	0	1個	0	0個
---	----	---	----	---	----	---	----

		1	0個			0	0個
--	--	---	----	--	--	---	----

		0	1個			0	1個
--	--	---	----	--	--	---	----

# 明日使えないすごいビット演算

1個	0個	1個	1個	0個	1個	0個	0個
----	----	----	----	----	----	----	----



0	1個	1	0個	0	1個	0	0個
---	----	---	----	---	----	---	----

		1	0個			0	0個
--	--	---	----	--	--	---	----

		0	1個			0	1個
--	--	---	----	--	--	---	----



足し算

0	0	1	1個	0	0	0	1個
---	---	---	----	---	---	---	----



# 明日使えないすごいビット演算

1個	0個	1個	1個	0個	1個	0個	0個
----	----	----	----	----	----	----	----



0	0	1	1個	0	0	0	1個
---	---	---	----	---	---	---	----

&0x0F

				0	0	0	1個
--	--	--	--	---	---	---	----

&0xF0

0	0	1	1個				
---	---	---	----	--	--	--	--

## 明日使えないすごいビット演算

1個	0個	1個	1個	0個	1個	0個	0個
----	----	----	----	----	----	----	----



0	0	1	1個	0	0	0	1個
				0	0	0	1個
				0	0	1	1個

# 明日使えないすごいビット演算

1個	0個	1個	1個	0個	1個	0個	0個
----	----	----	----	----	----	----	----



0	0	1	1個	0	0	0	1個
				0	0	0	1個
				0	0	1	1個



足し算

0	0	0	0	0	1	0	0個
---	---	---	---	---	---	---	----

## 明日使えないすごいビット演算

- 1になっているビットの数を数える
- かなり速い実装(dataはunsigned int型)

```
data = (data & 0x55555555)
      + ((data & 0xAAAAAAAA) >> 1);
data = (data & 0x33333333)
      + ((data & 0xCCCCCCCC) >> 2);
data = (data & 0x0F0F0F0F)
      + ((data & 0xF0F0F0F0) >> 4);
data = (data & 0x00FF00FF)
      + ((data & 0xFF00FF00) >> 8);
data = (data & 0x0000FFFF)
      + ((data & 0xFFFF0000) >> 16);
```

## 明日使えないすごいビット演算

- こうして、苦勞の末我々は爆速で1になっているビットの数を数えるアルゴリズムを手に入れた!!!

## 明日使えないすごいビット演算

- こうして、苦労の末我々は爆速で1になっているビットの数を数えるアルゴリズムを手に入れた!!!
- しかし...

## 明日使えないすごいビット演算

- こうして、苦勞の末我々は爆速で1になっているビットの数を数えるアルゴリズムを手に入れた!!!
- しかし...
- IntelのSIMD拡張命令セット、SSE4.2から、ズバリ「1になっているビットの数を数える」CPU命令が追加された!(popcnt)

# 明日使えないすごいビット演算

- こうして、苦労の末我々は爆速で1になっているビットの数を数えるアルゴリズムを手に入れた!!!
- しかし・・・
- IntelのSIMD拡張命令セット、SSE4.2から、ズバリ「1になっているビットの数を数える」CPU命令が追加された!(popcnt)
  - 実際ビット演算を使ったアルゴリズムより2倍ほど速い



# 明日使えないすごいビット演算

- こうして、苦労の末我々は爆速で1になっているビットの数を数えるアルゴリズムを手に入れた!!!
- しかし・・・
- IntelのSIMD拡張命令セット、SSE4.2から、ズバリ「1になっているビットの数を数える」CPU命令が追加された!(popcnt)
  - 実際ビット演算を使ったアルゴリズムより2倍ほど速い
- 我々の努力は無駄だった!!!!!!

## まとめと注意

- ビット演算はうまく使うととても高速
- ぱっと見何してるか判りづらいのでバグを埋め込みやすい
  - ものすごい高速化をする必要のないときは使わないほうが吉

## まとめと注意

- ビット演算はうまく使うととても高速
- ぱっと見何してるか判りづらいのでバグを埋め込みやすい
  - ものすごい高速化をする必要のないときは使わないほうが吉
- CPU命令速い!!!!!!
  - 本当に高速化したいときはまずこっちを考えるべき



おわり