集中講義令和8年2月24日～26日

コンピュータシステム入門

Day 1: 情報の表現とデータ構造

講師　陳　オリビア　（大学院システム情報科学研究院）

TA　GPT-5 Thinking (OPEN AI)

Gemini 2.5 pro (Google)

CONTENTS

今日のスケジュール

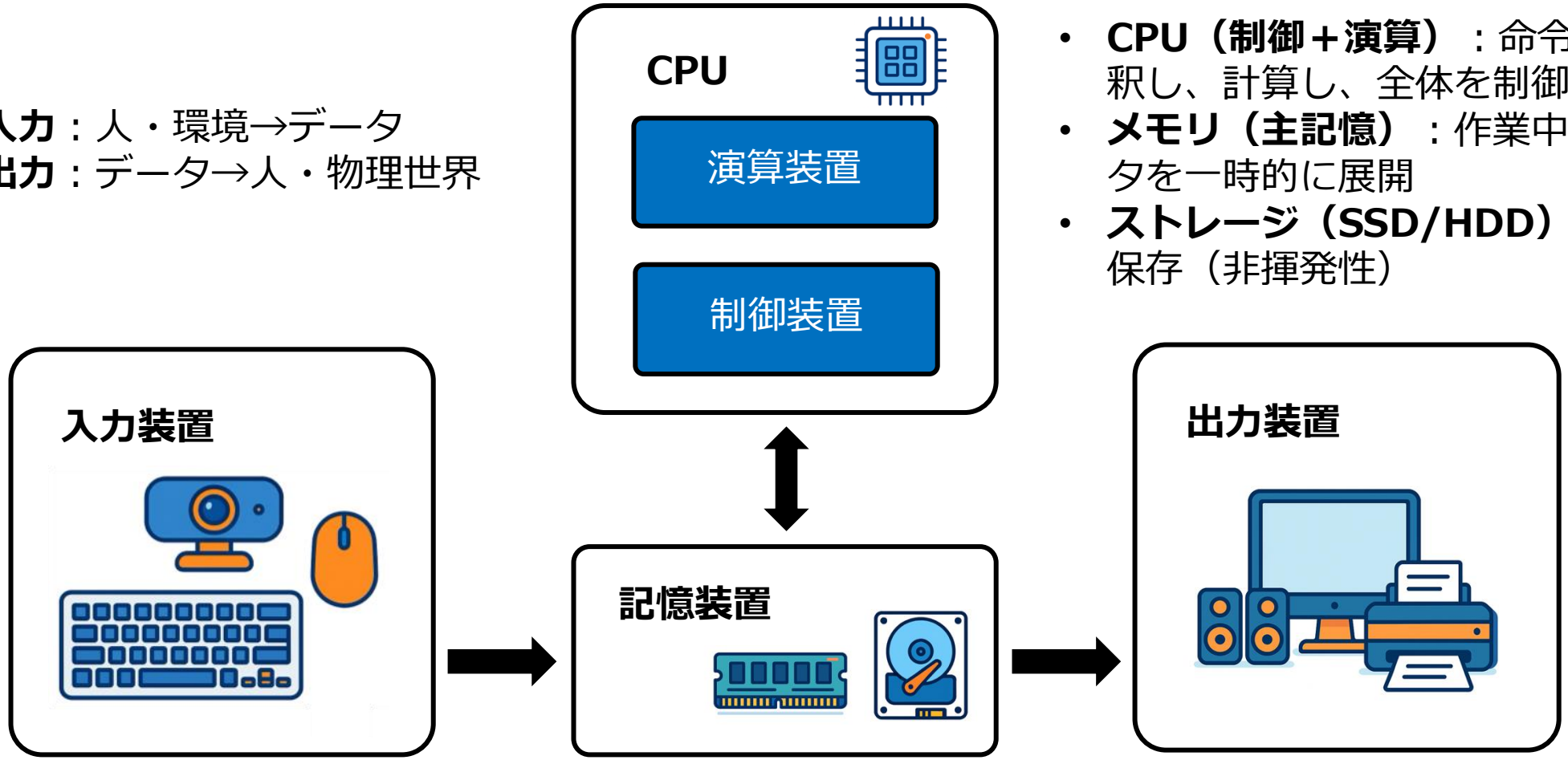
	時間帯	モジュール	タイプ
午前中	10:00 ~ 11:30	イントロダクションとコンピュータの基本構成	講義
	11:30 ~ 11:45	Coffee Break	
	11:45 ~ 12:30	コンピューターの分解	Demo
	12:30 ~ 13:30	Lunch Break	
午後	13:30 ~ 14:30	情報のデジタル表現	講義
	14:30 ~ 15:00	「デジタルの自分」を分解してみよう	演習
	15:00 ~ 15:15	Coffee Break	
	15:15 ~ 16:30	デジタル回路の基礎	講義
	16:30 ~ 17:00	論理ゲートで遊ぼう	演習

CONTENTS

モジュール２：情報の表現とデータ構造

前回の復習：コンピュータの５大装置

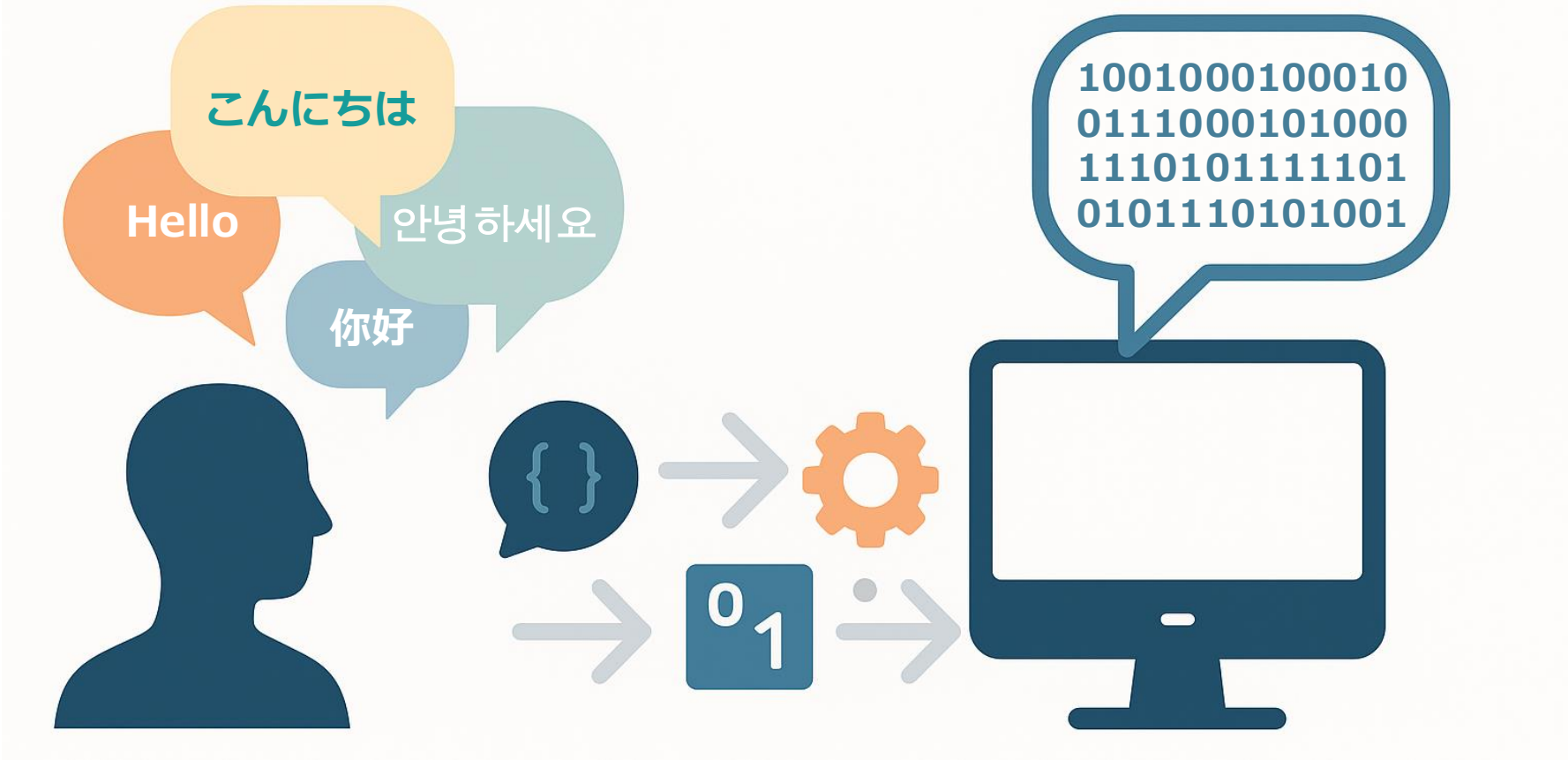
- **入力**：人・環境→データ
- **出力**：データ→人・物理世界



- **CPU（制御＋演算）**：命令を解釈し、計算し、全体を制御
- **メモリ（主記憶）**：作業中のデータを一時的に展開
- **ストレージ（SSD/HDD）**：長期保存（非揮発性）

CONTENTS

コンピュータが話す「言葉」とは？



人の言語 → 高水準言語 → コンパイラ/アセンブラ → 機械語（0/1） → CPU

↑ 抽象度（高いほど人間に近い）　← クリックで詳細

🔢 機械語

0 と 1 のビット列

CPU が直接実行できる唯一の言語。すべての命令は 2 進数（バイト列）で表現されます。人間には読みにくいため、アセンブラが自動的にアセンブリ → 機械語に変換します。

```
10110000 00001100
```

```
00000001 11000011
```

```
11101011 11111010
```

💡 x86 では LOAD AX, 12 → B0 0C（16進）になる。

↑ 抽象度（高いほど人間に近い）　← クリックで詳細

📝 アセンブリ言語

← このページで学ぶ層！

機械語と 1 対 1 に対応した人間向けの表記。`LOAD A, 12` のように英単語で書きます。CPU を直接コントロールできるため、CPU の動作が手に取るようにわかります。

```
LOAD A, 12
ADD A, B
JMP loop
HALT
```

💡 このシミュレーターの命令がまさにアセンブリ！

↑ 抽象度（高いほど人間に近い）　← クリックで詳細

⚙️ 低水準言語

C / C++ / Rust / Go …

メモリ管理やポインタを直接扱える言語。高水準言語より速く、OS やデバイスドライバの開発に使われます。コンパイル後にアセンブリ→機械語へと変換されます。

```
int *p = &x;
```

```
malloc(256)
```

```
#include <stdio.h>
```

💡 Linux カーネルは主に C 言語で書かれている。

↑ 抽象度（高いほど人間に近い）　← クリックで詳細

🐍 高水準言語

Python / Java / JavaScript / Ruby …

人間が読み書きしやすい言語。変数・関数・ループなどの抽象的な概念で書けます。そのままでは CPU が実行できないため、インタープリタやコンパイラが機械語に変換します。

```
print("Hello")
```

```
for i in range(10):
```

```
result = a + b
```

💡 このシミュレーターの Python 例がこの層に対応。

↑ 抽象度（高いほど人間に近い）　← クリックで詳細

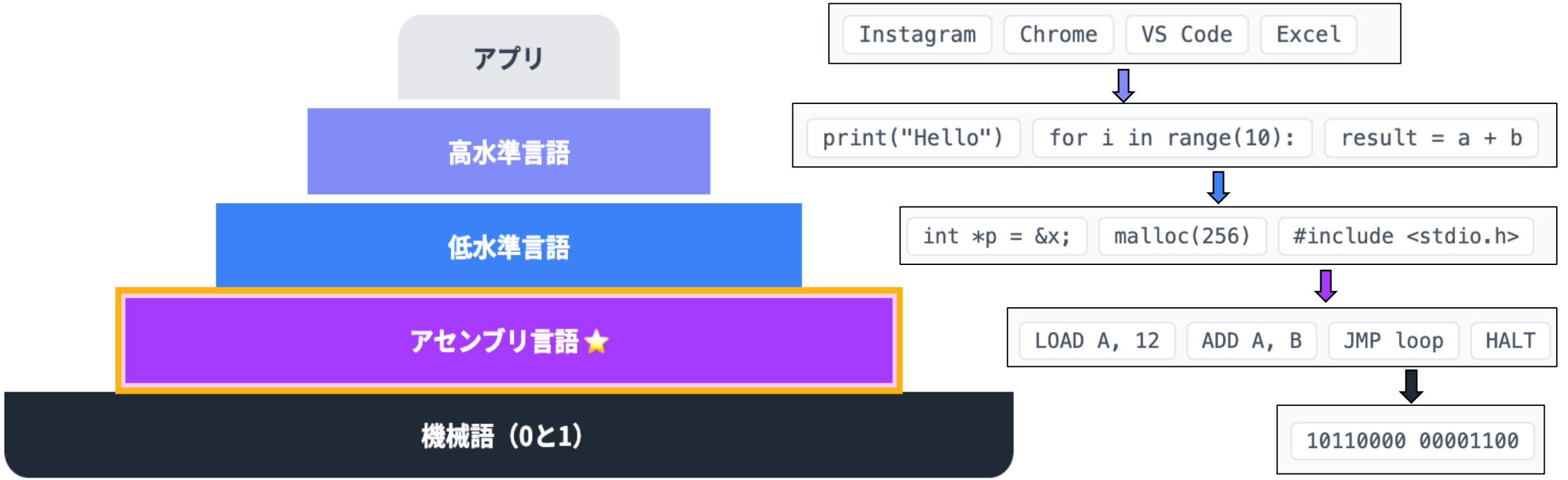
📱 アプリケーション

SNS・ゲーム・ブラウザなど

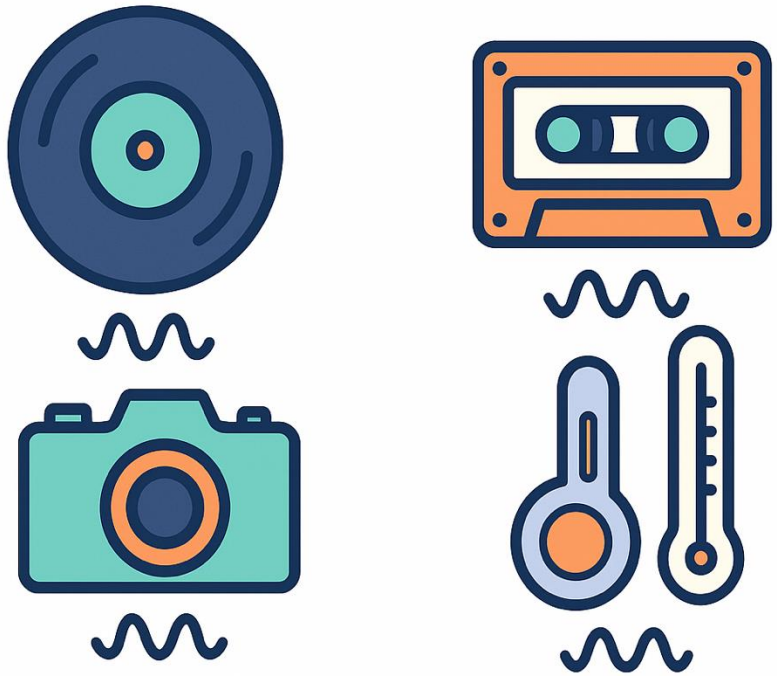
エンドユーザーが直接使うソフトウェア。Python や JavaScript などで書かれており、ハードウェアの知識がなくても開発できます。

`Instagram` `Chrome` `VS Code` `Excel`

> 💡 CPU の動作を直接意識することはほぼない。

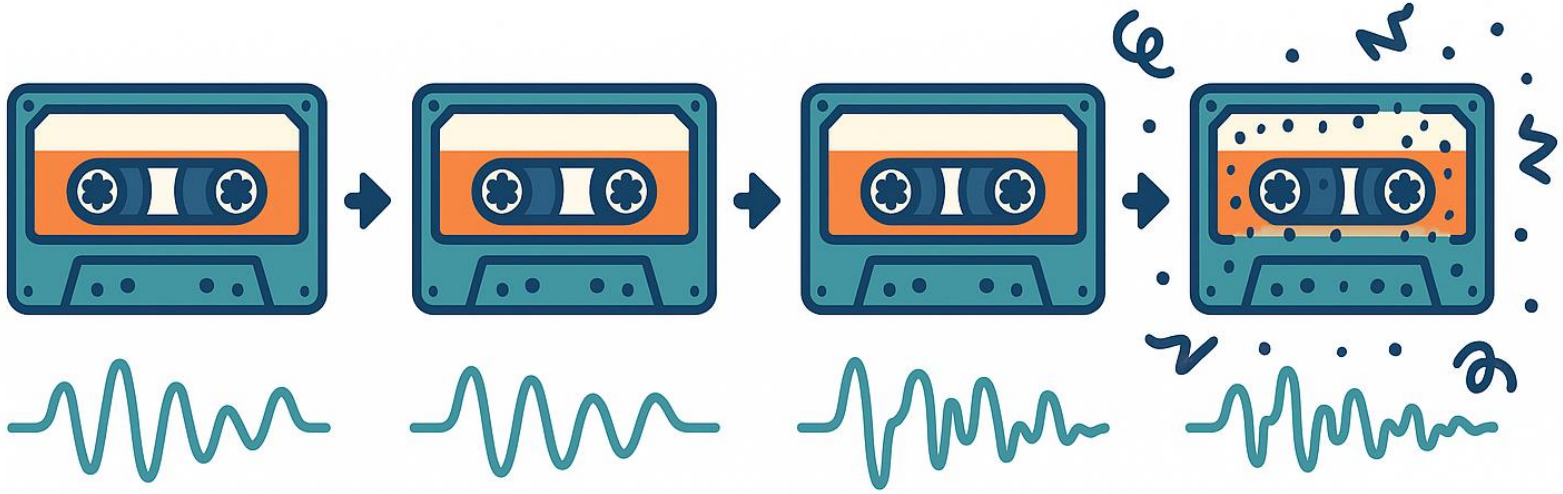

アプリ
高水準言語
低水準言語
アセンブリ言語⭐
機械語（0と1）
Instagram
Chrome
VS Code
Excel
print("Hello")
for i in range(10):
result = a + b
int *p = &x;
malloc(256)
#include <stdio.h>
LOAD A, 12
ADD A, B
JMP loop
HALT
10110000 00001100

世界は「アナログ」で満ちている



- 音の大きさ・高さ、光の明るさ、温度・圧力・位置など、多くの物理量は連続的（アナログ）に変化する
- 例：レコードの溝の起伏、カセットテープの磁束、フィルムの感光、水銀体温計の膨張
- **アナログ＝連続信号**
- **デジタル＝離散化（0/1）**

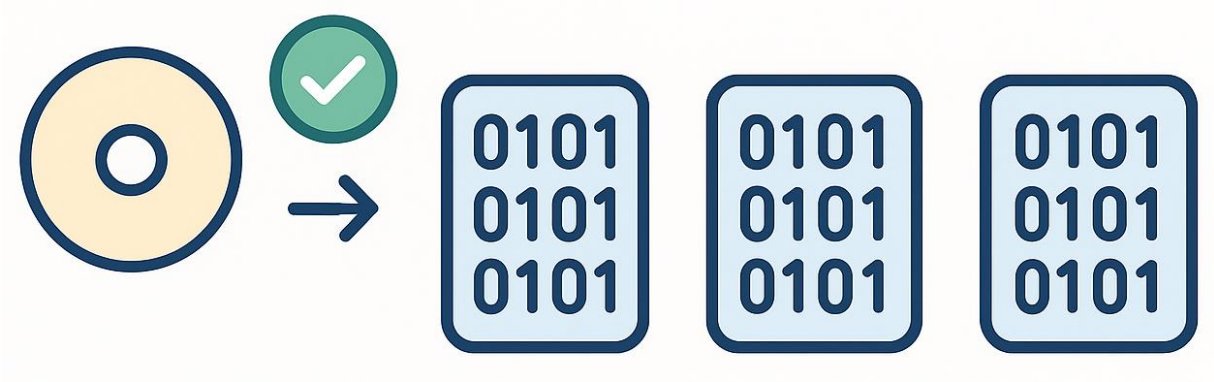
アナログの弱点：劣化



きれい　→　少しノイズ　→　さらに劣化　→ ノイズだらけ

- **S/N比の低下**：ヒスノイズが加算されていく
- **帯域の狭まり**：高音が削れ、ダイナミックレンジも縮小
- **ひずみ・タイミング誤差**：飽和、ワウ・フラッター、ヘッドずれ
- **メディア劣化**：テープ摩耗・磁性体の退行

デジタルの強み：劣化しないコピー

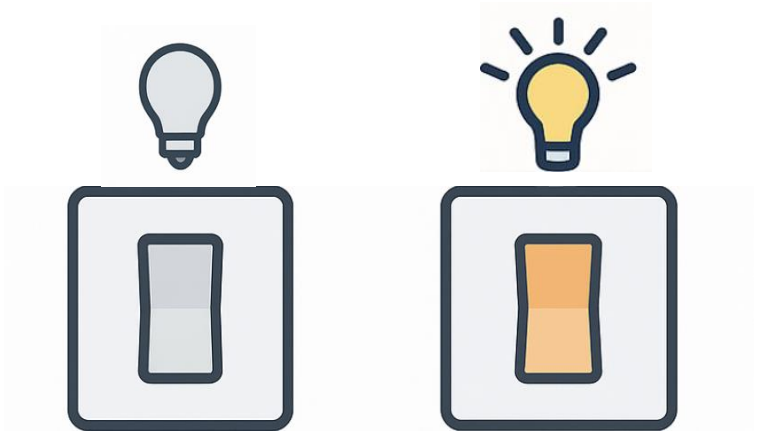


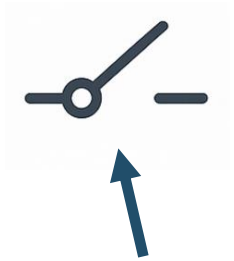
- データは**0/1の列**。
- **ビット単位で一致**すれば、何回コピーしても**完全に同じ**。
- **ファイルのコピー＝bit-perfect** → 音質・画質は変わらない。

ただし：**再圧縮（例：MP3→MP3）や形式変換**は別

コンピュータの基本は「スイッチ」

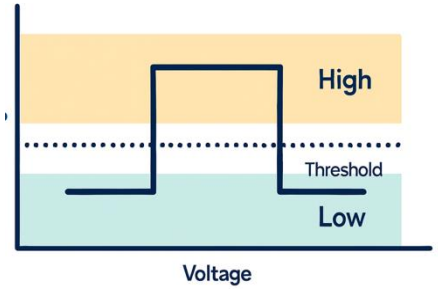
デジタルは **2値**：ON(1) / OFF(0)

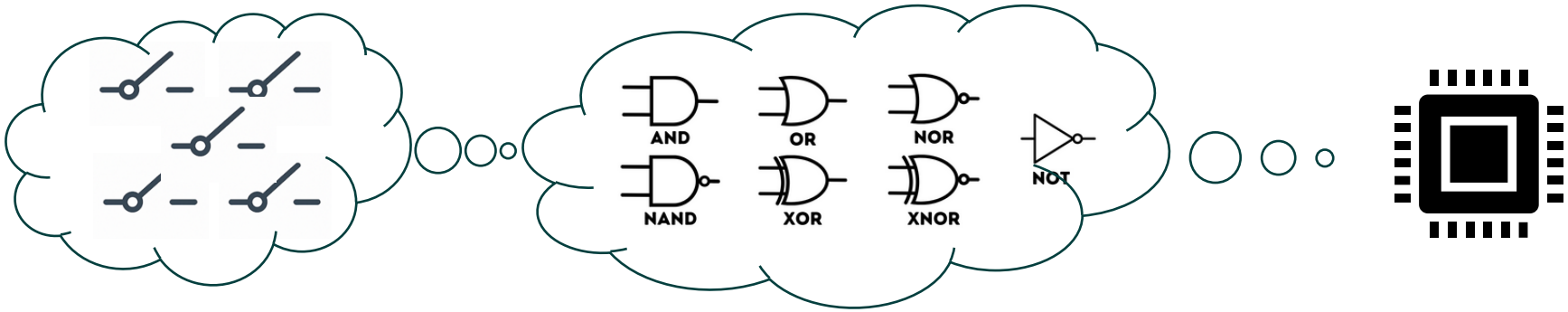




トランジスタ（MOSFET）がスイッチ役
（デジタル回路基礎で紹介）

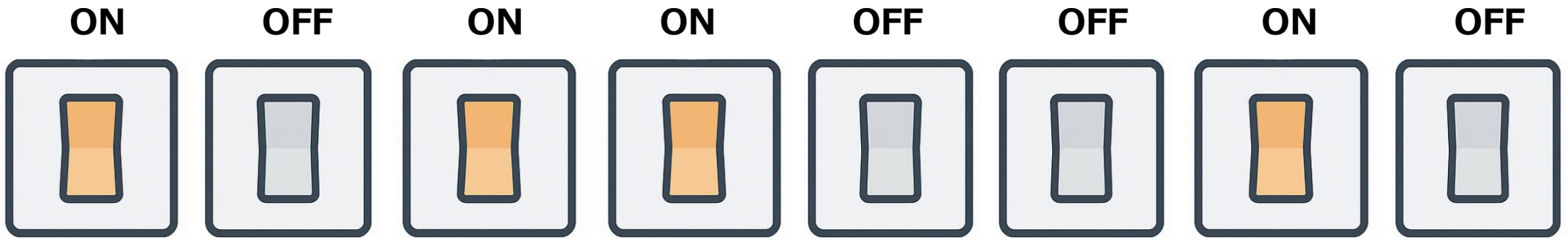
電気的には **High / Low の電圧**
（しきい値より上=1、下=0）





スイッチ　論理（デジタル）回路　CPU

「スイッチ」を「数字」に対応させる



このビット列「10110010」には、どんな“意味”がある？

数値：178, -78, 5.5625, …

グレースケール画素：明るさ ≈ 70%（178/255）

CPU命令（オペコード）：MOV DL, imm8（DLレジスタへ即値を転送する命令）

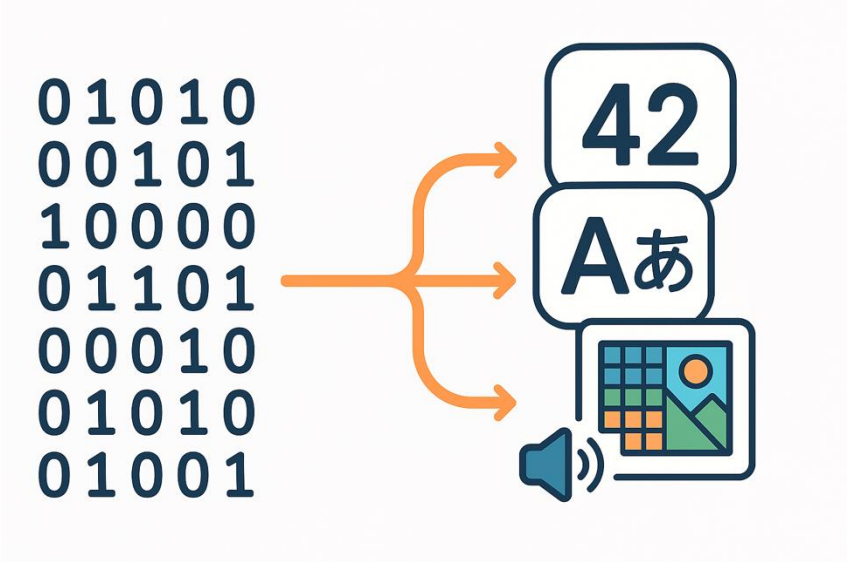
2つの状態で世界を表現する

モールス信号の「・(トン)」と「—(ツー)」で情報を表す

例：**SOS** = ・・・ ——— ・・・

重要なのは**はっきりした2状態**と**区切り**（記号間／文字間）

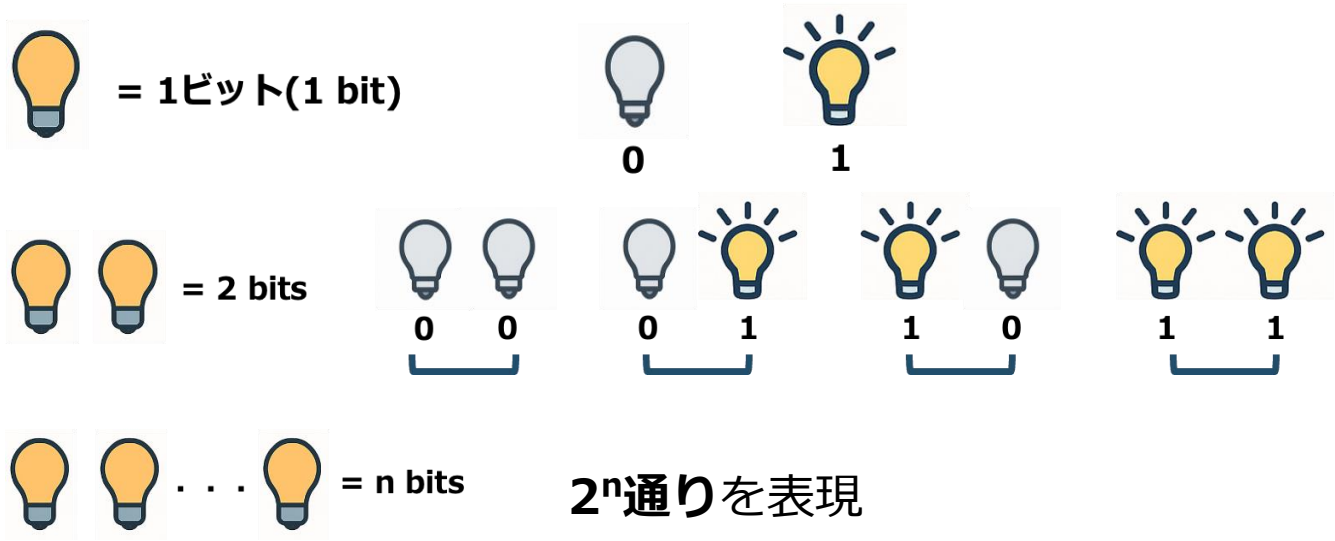
同じ発想で、**0と1**の組合せが
数・文字・画像などを表現する



CONTENTS

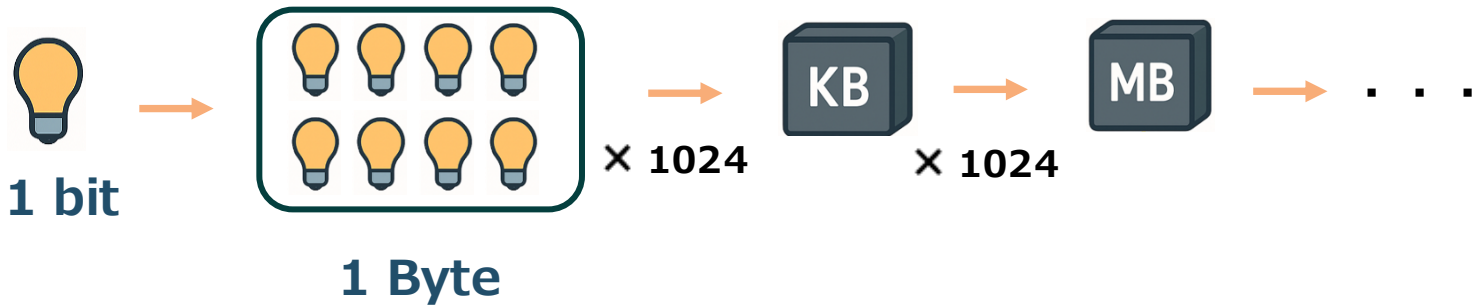
モジュール２：情報の表現とデータ構造

情報の単位「ビット」と「バイト」



2^n**通り**を表現

8ビット＝1バイト
(8 bits = 1 Byte)



2進数の「桁の重み」

10進数の重み

$$1024 = 1\times1000 + 0\times100 + 2\times10 + 4\times1$$

10^3 10^2 10^1 10^0

2進数の重み

$$1011 = 1\times8 + 0\times4 + 1\times2 + 4\times1 = \mathbf{11}$$

2^3 2^2 2^1 2^0

問題： $01010110_2 = ??_{10}$　　$178_{10} = ???_2$

2進数の足し算（繰り上がり）

1+1=10

桁上げ
Carry・キャリー

1+1+1=11

```
   1110  (14)
+  0101  (5)
---------------
```

問題発生：マイナスをどう表す？

8 − 3 = ?

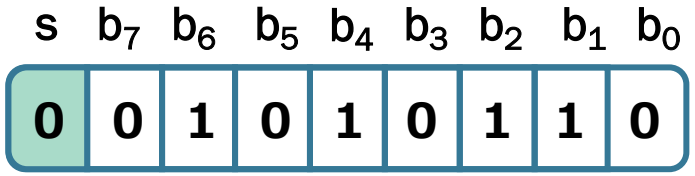
人間にとっては当たり前の引き算を、
コンピュータはどう考えているんだろう？

0/1の世界に“−”はない >> CPUはON・OFFの状態しか理解できない

うまくルールを作って、ビット並びだけで、マイナスを実現

アイデア：符号ビット

符号ビット・sign bit



+86

-86

- +0 と -0 ?
- 加算回路に使えない

賢い解決：「補数」で引き算→足し算に

A – B → A + (-B)

(-B)：**2の補数**

2の補数の求め方：
1.元のビットをすべて反転する（1の補数をとる）
2.さらに “1” を足す

例：　5 – 3 = 0101 - 0011　= 0101 + 1101 = (1)0010

0011：**2の補数 = 1101**

例：　3 – 8 = 00011 + 11000 = 11011

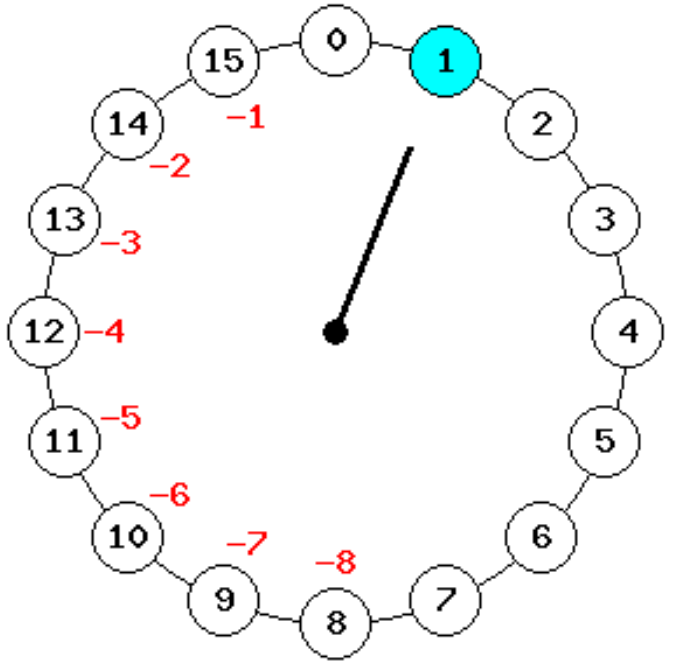
11011：**2の補数 = 00101 = 5**

補数の直感：時計のたとえ

5 – 3 ⇨ 5時のクロックを３時間戻す

＝９時間進める

「+9」は「-3」の補数（１週最大12の場合）




```
Action:    ADD 1

Bin:       0001
Hex:       1
Unsigned:  1
Signed:    1

Zero:      0
Carry:     0
Sign:      0
Overflow:  0
```

https://stackoverflow.com/questions/23990071/sign-carry-and-overflow-flag-assembly

限界：桁あふれ（オーバーフロー）

0111 (7)
+ 0101 (5)

1100 (-4 ?!)

1. より大きな「箱」を用意する (ビット数を増やす)

7 → 0000 0111
5 → 0000 0101

0000 0111 (7)
+ 0000 0101 (5)

0000 1100 (12)

⇨ 現代CPUは大きい箱（64ビット）

2. ハードウェアによる検出と、ソフトウェアによる対処

⇨ Overflow Flag (V flag)

読みやすさのために：16進数

2進数の弱点：長くて読みにくい　　1101010111101011

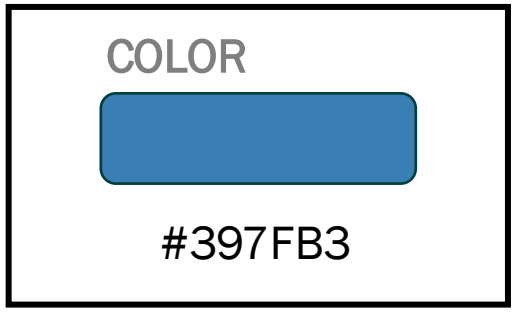
解決策：4ビットずつ区切る　　1101 0101 1110 1011

なぜ4ビット？ → 16進数

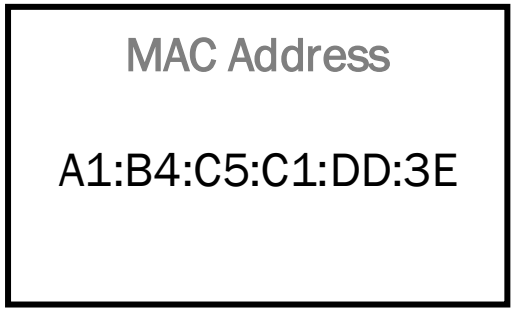
→ $2^4=16$通り表現

2進数	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
10進数	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16進数	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

16進数は身近に！



Webカラーコード



MACアドレス

メモリエラー

ゲームチートコード

CONTENTS

モジュール２：情報の表現とデータ構造

文字の表現：文字コード

コンピュータは 文字をそのまま理解できない

⇩

ルールブック（約束事）によって**数字**に変換

A → 65

ASCII（アスキー、英: **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange）

（1960年代）：**7ビット＝128文字**

ASCIIコードの文字表

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

もしルールが違うと…？

ã□“ã,“ã□«ã□¡ã□¯ æ—¥æœ¬èªž: æ–‡å—åŒ–ã□ ãƒ«ãƒ¼ãƒ«ãƒ–ãƒƒã,¯ â€œSampleâ€□â€” itâ€™s broken. □□□□?? □■□■ □□??□□

ASCIIからUnicodeへ

ただし…
英数字・記号・制御文字が中心。日本語・中国語・絵文字は対象外
インターネットと多言語化 → **文字が圧倒的に不足**
結論：世界中の文字を一意に扱うために **Unicode** が登場

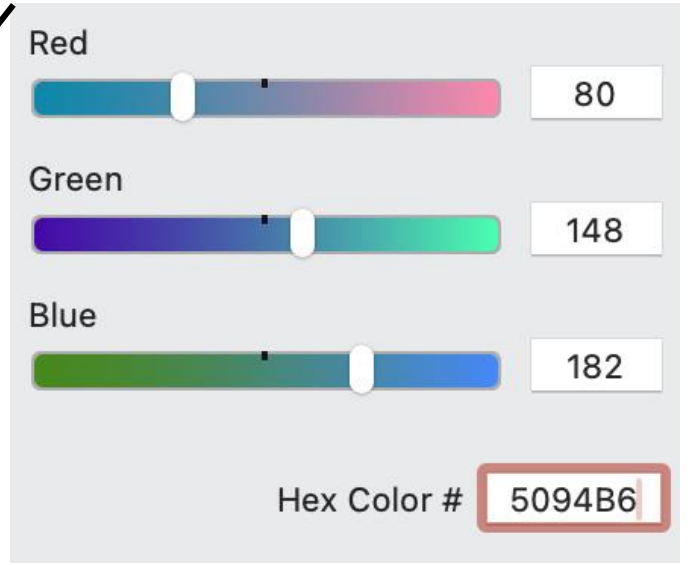
Unicodeとは？（“文字に番号”のしくみ）

- すべての文字に**コードポイント**（番号）を付与：**U+XXXX**
  例：A = **U+0041** / あ = **U+3042** / 漢 = **U+6F22** / 🙂 = **U+1F642**
- **U+0000～U+007F は ASCII と完全一致**（後方互換）
- **エンコーディング（符号化方式）**
  - 「コードポイント → ビット列」に変換する方法
    - **UTF-8**：可変長（1～4バイト） Web標準、ASCIIと互換
    - **UTF-16**：2/4バイト。Windows/Java内部など
    - **UTF-32**：4バイト固定（メモリ重め）
  - 例：　A → UTF-8: **41**（1バイト）あ → UTF-8: **E3 81 82**（3バイト）

次のうちコードポイントはどれ？
0x41 / U+3042 / E3 81 82 / 65

文字化け＝**送受のエンコーディング不一致**／フォント未対応

画像の表現：ピクセルとRGB

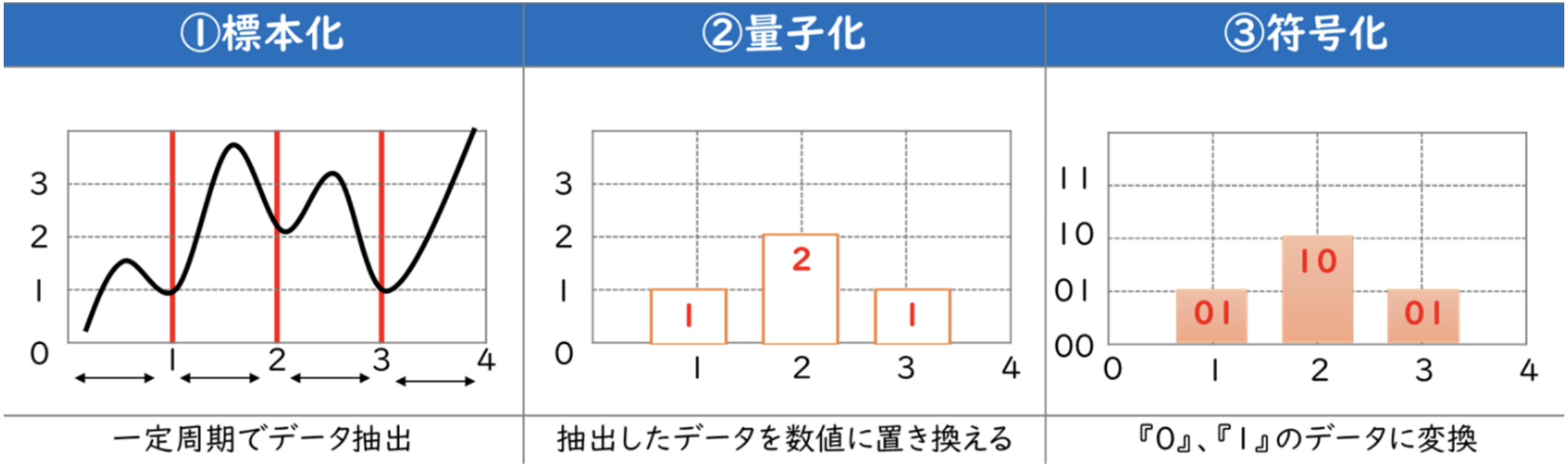


コンピュータ内部では、各色の強さを**0から255までの256段階の数字**で表すのが一般的

- **真っ赤:** R:255, G:0, B:0
- **真っ白:** R:255, G:255, B:255
- **真っ黒:** R:0, G:0, B:0

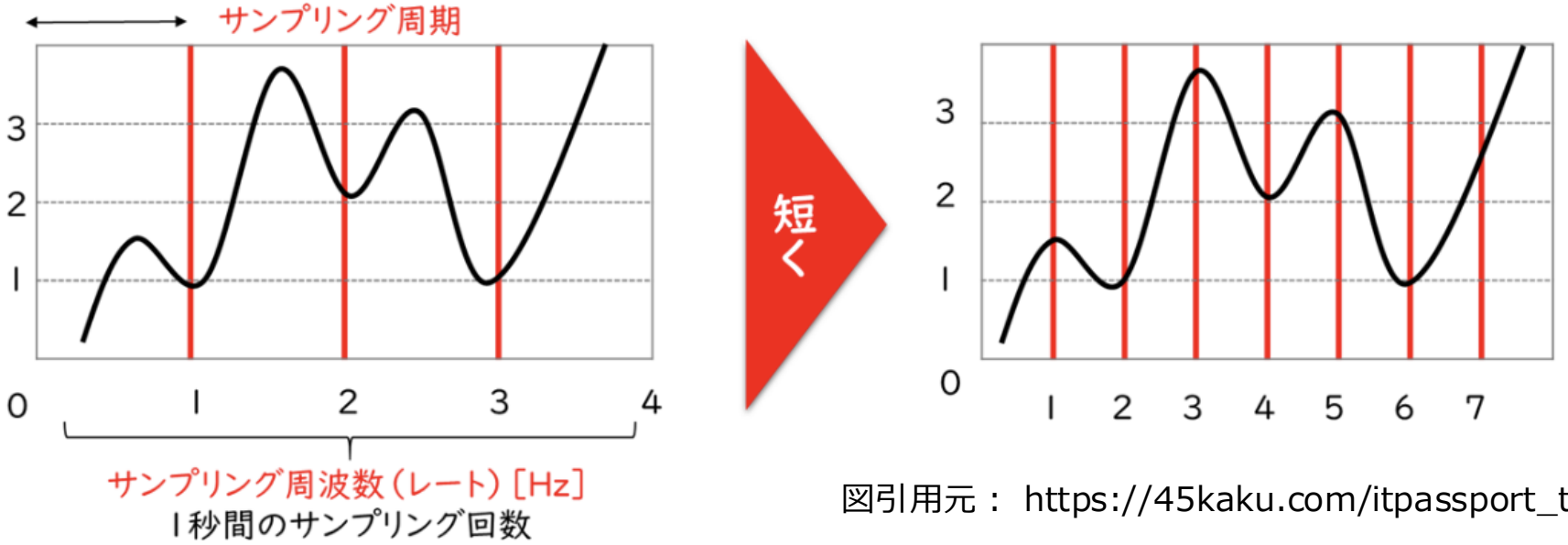
音の表現：サンプリング

アナログ波 → サンプリング → デジタル（量子化）→符号化



引用元： https://45kaku.com/itpassport_text_35-03/

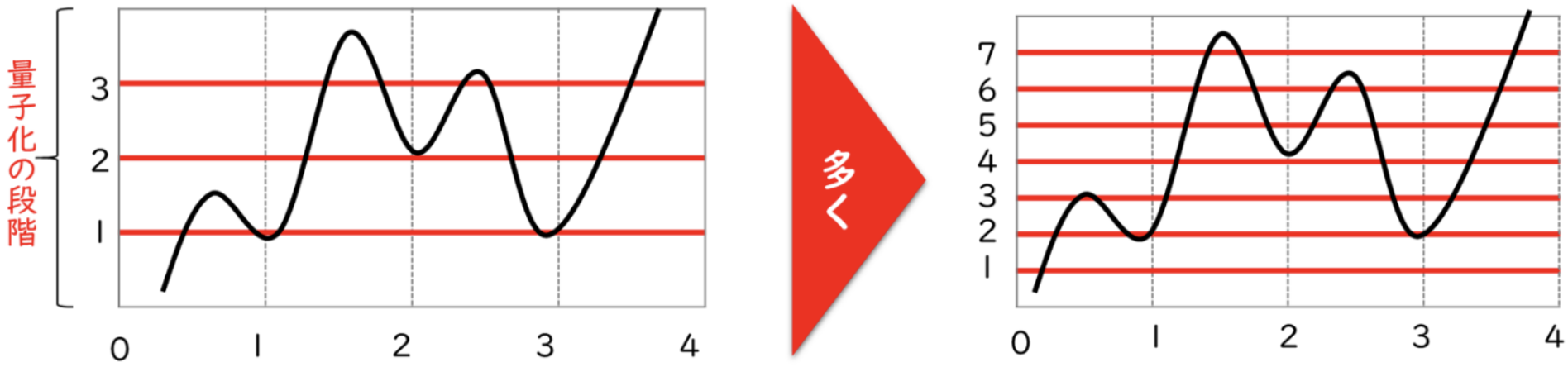
音質の要素①：標本化周波数



図引用元： https://45kaku.com/itpassport_text_35-03/z

- 測定回数が多いほど、元の音の波形を忠実に再現できる
- 音楽CDは秒毎に44,100回も測定している
- これはパラパラ漫画のコマ数が多いほど動きが滑らかになるのと同じ

音質の要素②：量子化ビット数



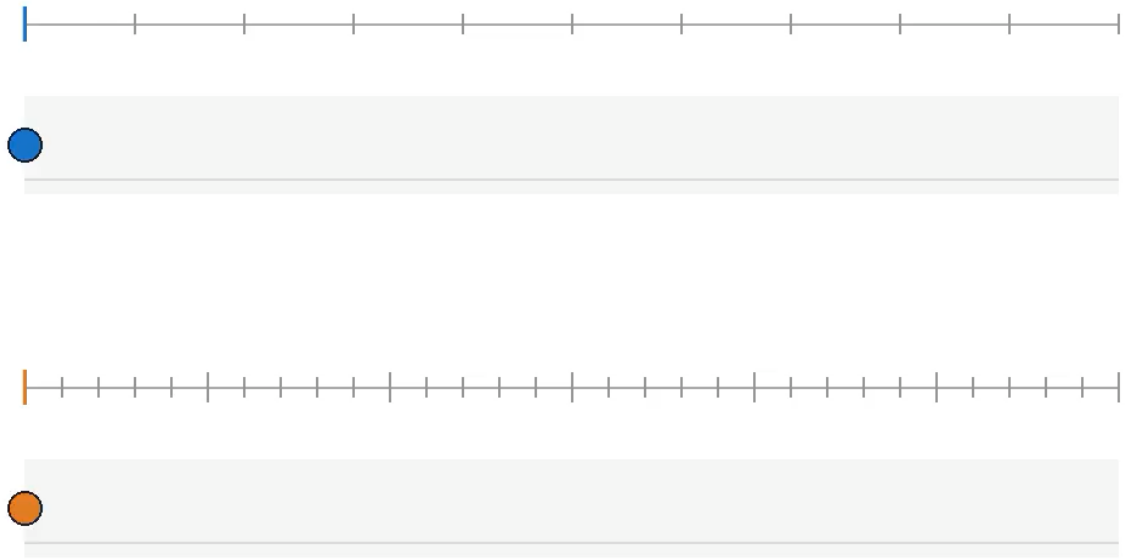
図引用元： https://45kaku.com/itpassport_text_35-03/z

- 一回一回の測定値をどれだけ細かく表現する「**量子化ビット数**」
- 音楽CDは**16**ビット、つまり音の高さを**65,536**段階で表現
- 音の大小の表現力（ダイナミックレンジ）に関わる
- ものさしの目盛りが細かいほど正確に長さを測れるのと同じ

動画の表現：パラパラ漫画の原理

- 動画の原理は**パラパラ漫画**と同じ
- **少しずつ変化する静止画**を**高速**に連続表示 → 動いて見える

動画の要素①：フレームレート

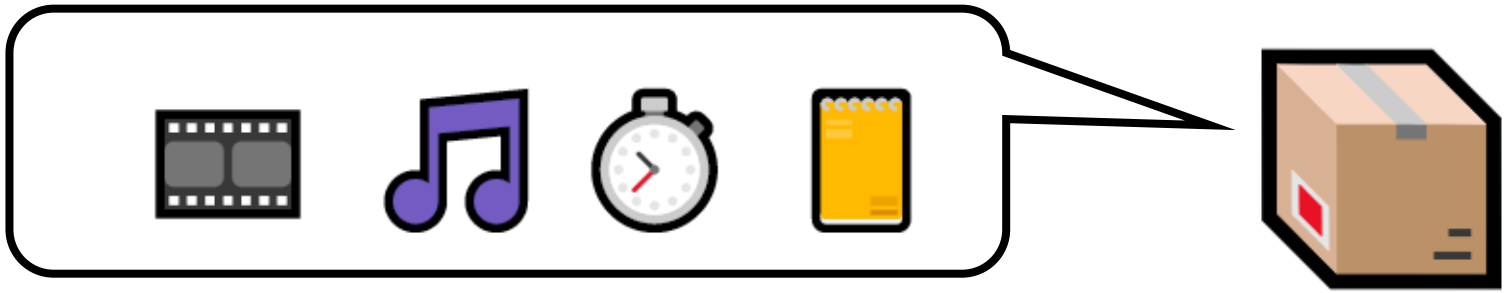


- **フレームレート（fps）＝1秒間に表示する静止画の枚数**
- 数字が大きいほど**動きが滑らか**
- 例：映画 **24fps** / テレビ **30fps**, **25fps** / ゲーム **60fps** 以上

動画の要素② ― 映像と音声

動画ファイル = 静止画の連続（映像） + 音声 + 同期（timestamp）+ 付加情報

- **映像**：1枚1枚の**フレーム**が時間順に並んだもの
- **音声**：波形データ（サンプリング＆量子化済み）
- **同期（timestamp）**：
  - 「**このフレームは何秒の位置か**」「**この音声サンプルは何秒か**」を記録
  - 映像と音が**ズレない**ようにする“時間のものさし”
- **付加情報**（メタデータ）：字幕／章情報／サムネイル／色空間 など



動画の要素② ― 映像と音声（続き）

コンテナとコーデック

- **コンテナ = 入れ物**
  - 役割：映像トラック・音声トラック・字幕・timestamp をひとつのファイルに梱包
  - 例：MP4, MKV, MOV, WebM …（拡張子はだいたい“コンテナ名”）

- **コーデック = 圧縮方式（encode/decodeのアルゴリズム）**
  - 映像コーデック：H.264/AVC, H.265/HEVC, AV1, VP9 …
  - 音声コーデック：AAC, Opus, MP3, FLAC …

よくある組み合わせ（実用）

- **MP4（コンテナ）+ H.264（映像）+ AAC（音声）** ← 最も互換性が高い
- MKV + HEVC/AV1（映像）+ Opus（音声） ← フレキシブル、配信/保存向き
- WebM + VP9/AV1（映像）+ Opus（音声） ← Web配信で多い

動画のデータ量はなぜ大きい？

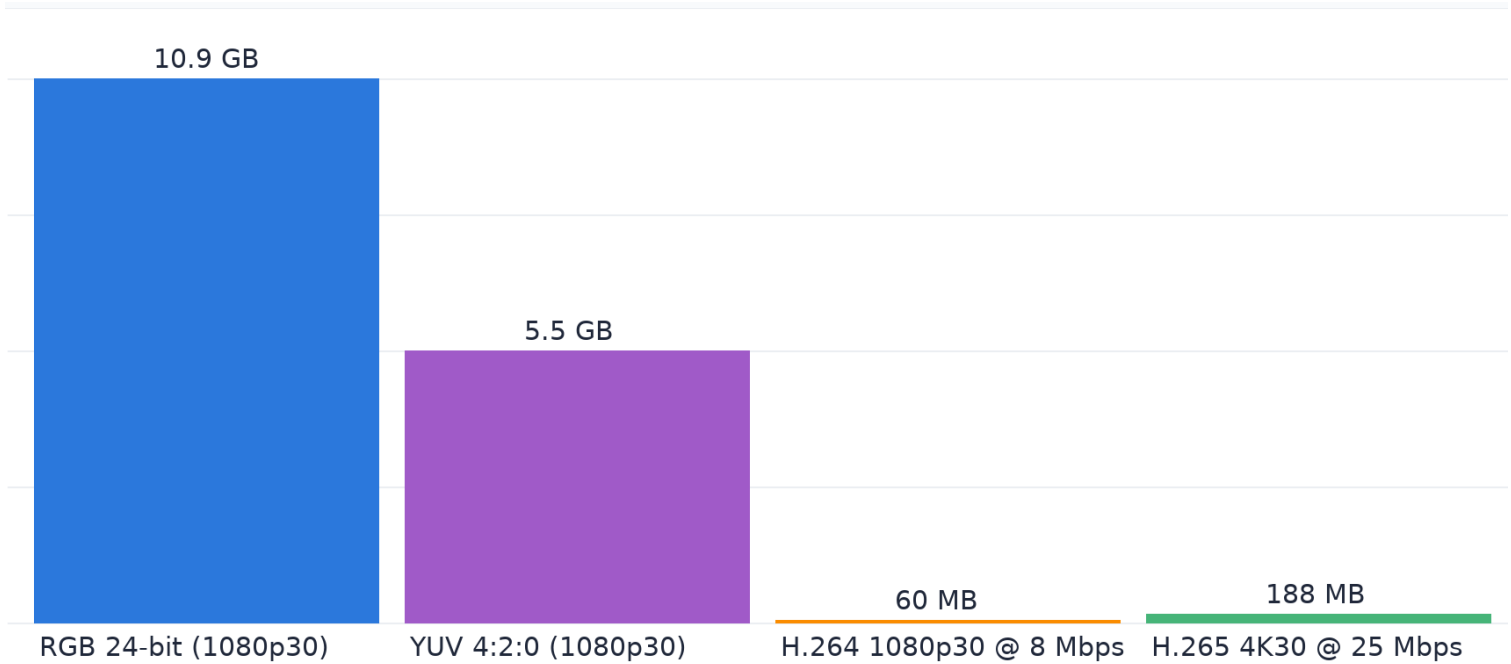
動画サイズ = (1枚のフレームのサイズ) × (fps) × (時間)　**映像サイズ**

+ サンプリング周波数 × bit深度/8 × チャンネル数 × 時間　**音声サイズ**

+ その他 （メタデータ/コンテナのオーバーヘッド数%）

例：フルHD (1920×1080)、30fps、60秒

- 非圧縮RGB（24bit）：
  1枚 ≈ 1920×1080×3 ≈ 6.2MB
  → **6.2MB × 30 × 60 ≈ 11GB**
- PCM 48kHz/16bit/2ch ≈ **11 MB**

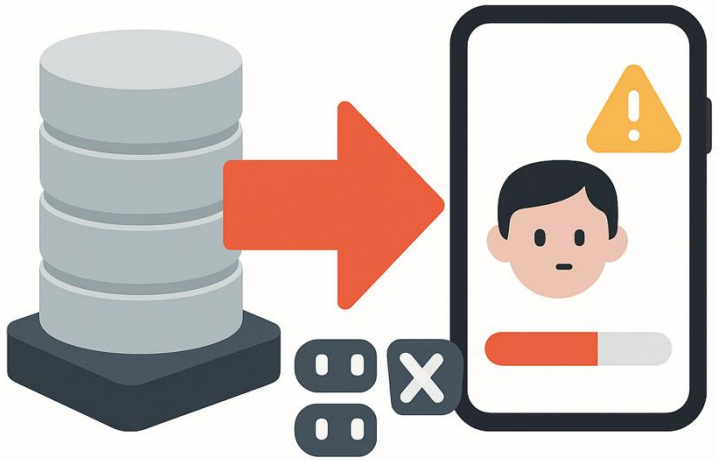


CONTENTS

モジュール２：情報の表現とデータ構造

なぜ圧縮が必要か？

60秒のHD動画サイズ
⇨11GB



iPhone 16/256GB
20分の動画でいっぱい

- さきほどの計算の通り、**生データ（未圧縮）は巨大**
- このままでは**保存も送信も現実的ではない**（スマホの容量・通信帯域がすぐ限界）
- **情報圧縮**という「**賢く小さくする**技術」が不可欠

圧縮の基本アイデア：繰り返しをなくす

データの世界では、**連続した同値** や
類似のパターン が頻出する

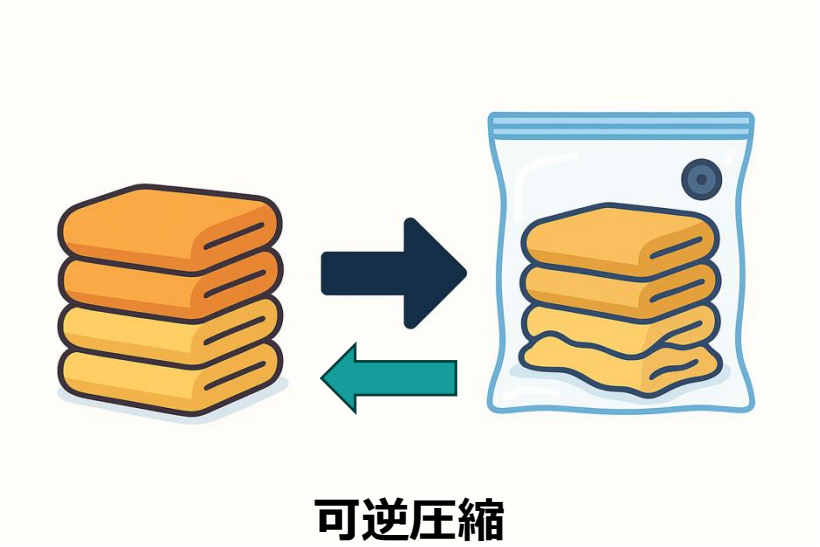
- 圧縮の基本はとてもシンプルで、**“同じものが続くなら短く表す”**
  - 例：AAAAA を **「Aが5回」** と記録すれば、文字数はずっと少なくてすむ（ランレングス符号化）

Run-Length Encoding, RLE

…あああああいいいいう…

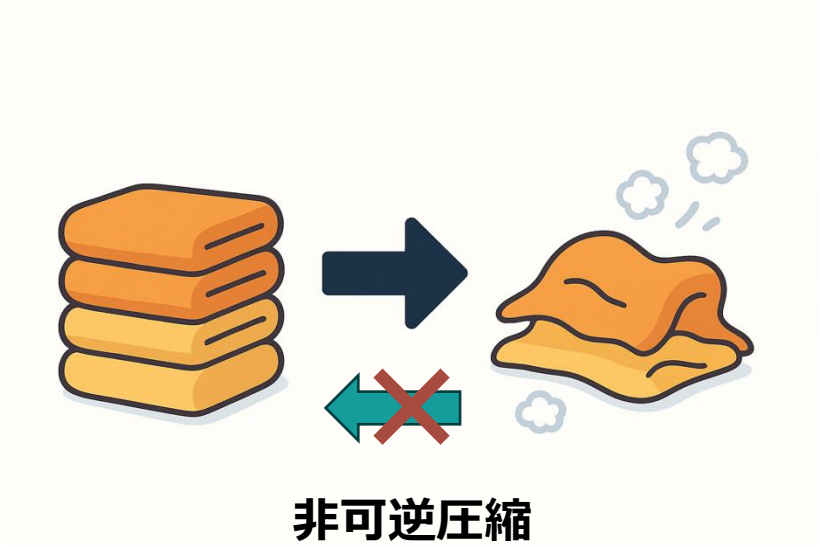
⇨ あ×５，い×４，う×１

圧縮の種類



可逆圧縮
(Lossless)

- **1ビットも失わない**圧縮
- **ZIP / PNG / FLAC**など
- テキストやプログラムのように**正確さが必須**のデータに適
- 例：ランレングス符号化



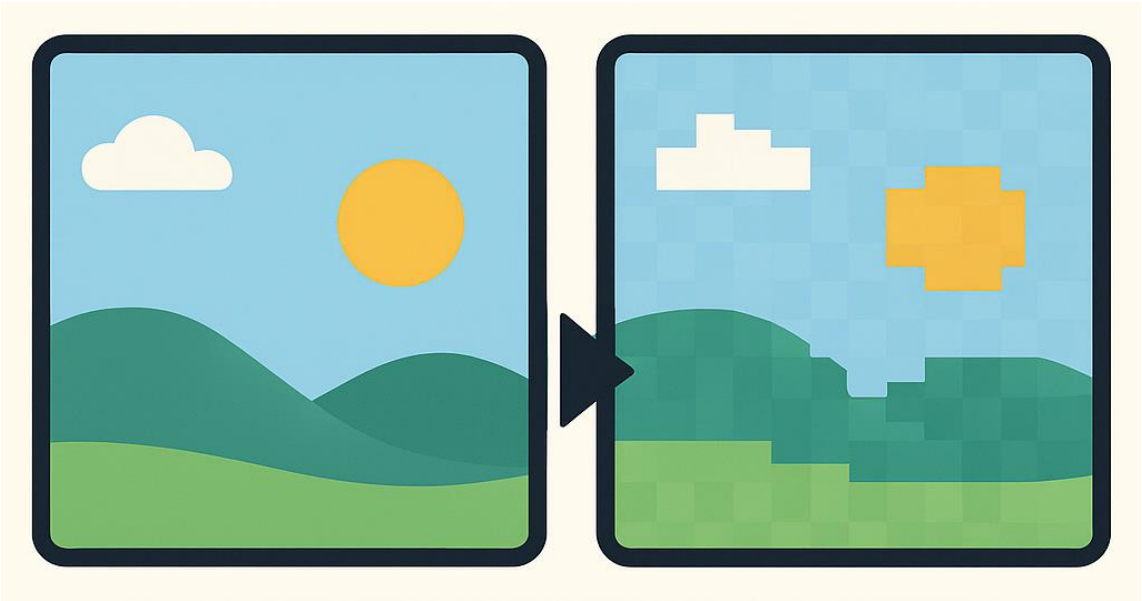
非可逆圧縮
(Lossy)

- **人間の目・耳の鈍感さ**を利用し、気づきにくい成分を間引くことで**圧倒的な圧縮率**を得る
- **JPEG / MP3 / H.264**など

非可逆圧縮の例①：JPEG

JPEGの仕組み：見た目を保ちながらサイズを減らす

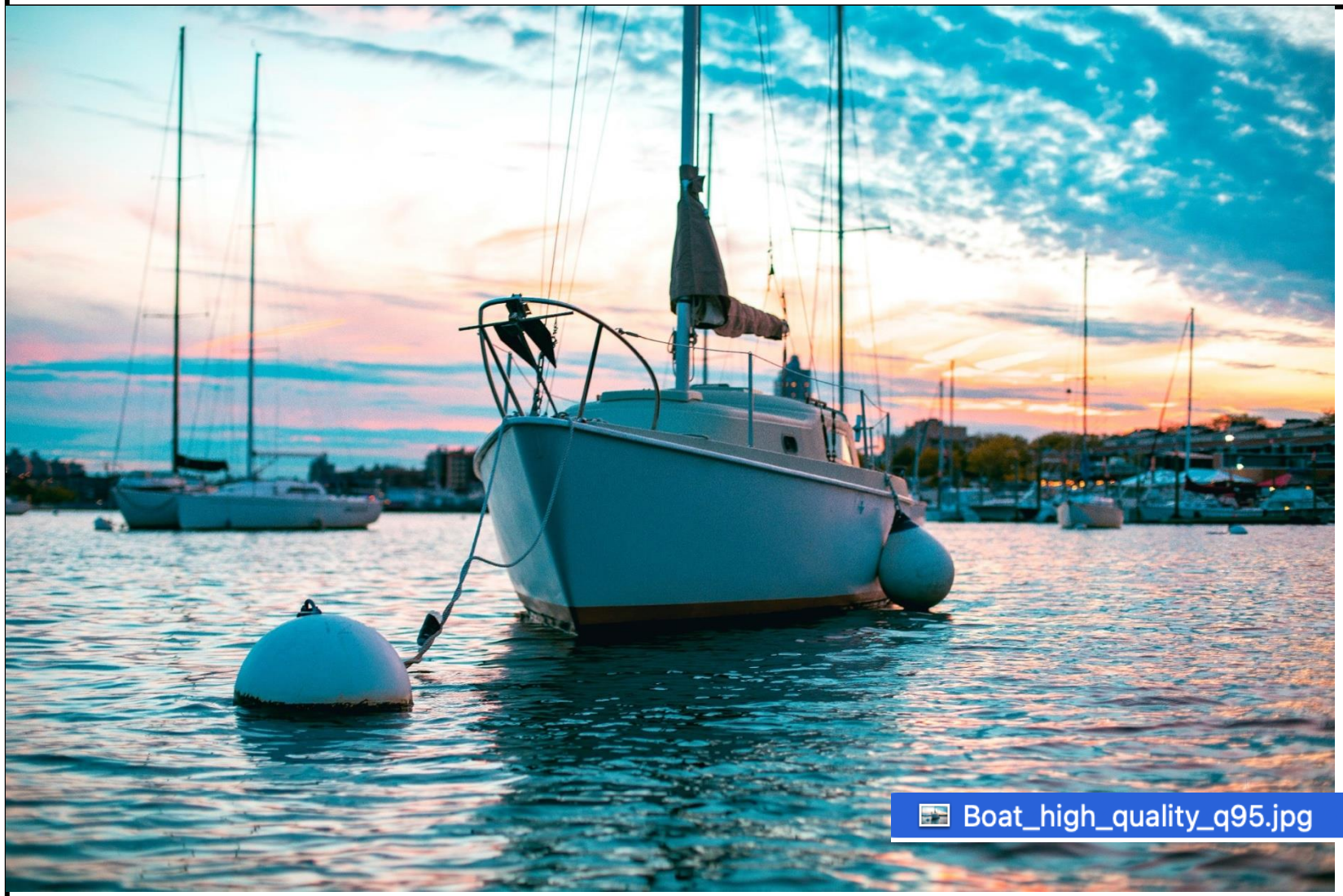
- **明るさ**はしっかり、**色の細かな変化**は少し省く
- 画面を**小さなタイル**に分け、**細かい模様**を少し平均化
- 見た目ほぼそのまま、**10倍以上**小さくできる



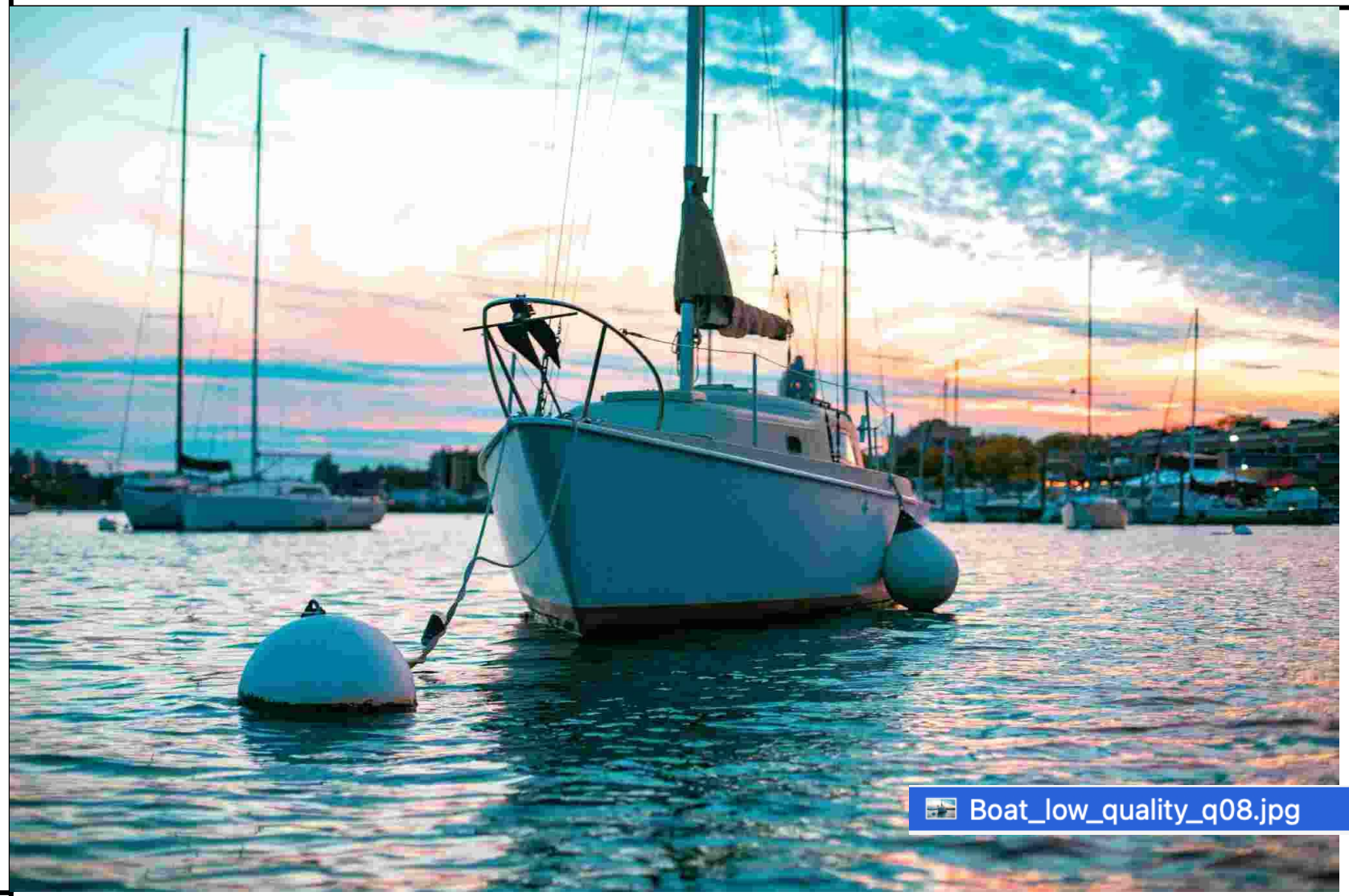
非可逆圧縮の例①：JPEG

Boat_original.jpg　4.5 MB　JPEG image

非可逆圧縮の例①：JPEG



非可逆圧縮の例①：JPEG

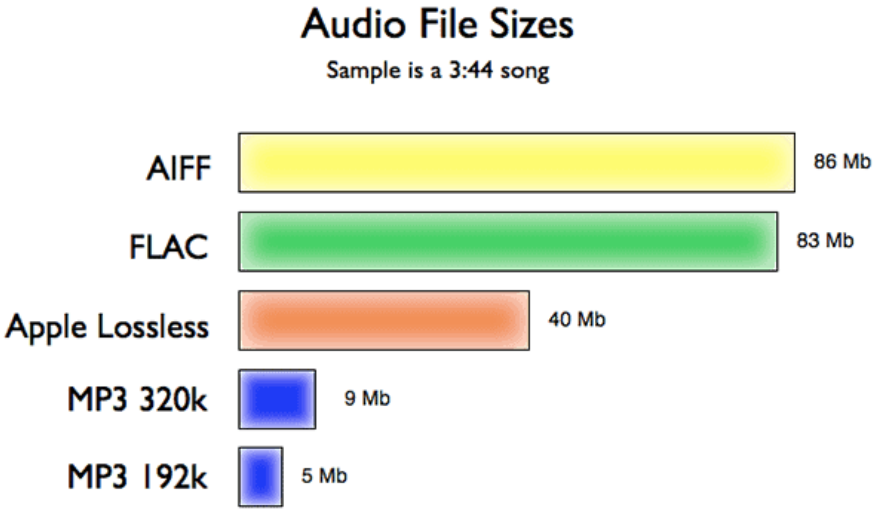


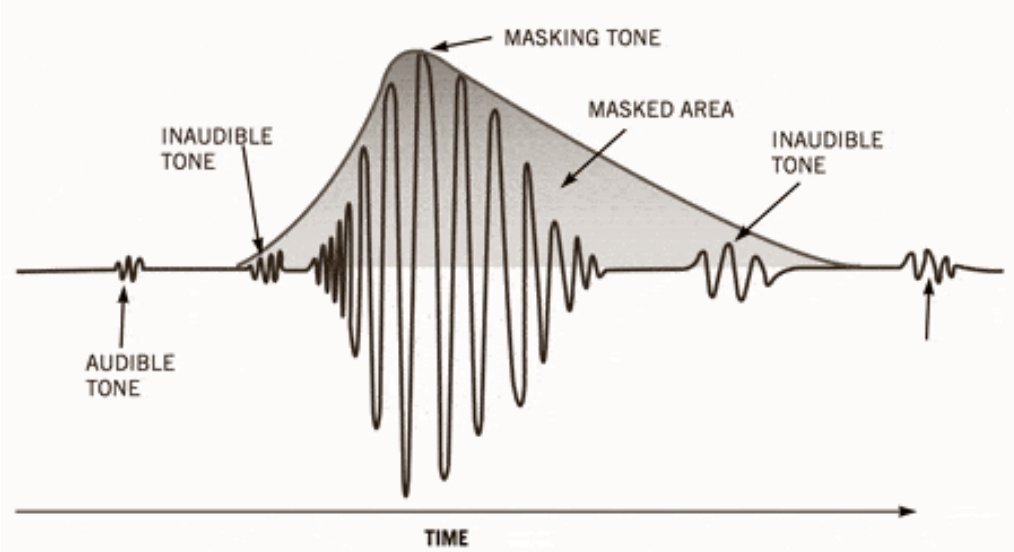
Boat_low_quality_q08.jpg　63 KB　JPEG image

非可逆圧縮の例①：JPEG

非可逆圧縮の例②：MP3と動画

音（MP3）：大きな音に隠れて**聞こえない小さな音**を省く（マスキング）





動画：**変わらない部分は使い回し、動いた差だけ**を保存

まとめ

今日の3つのキーアイデア

- 0/1（ビット）はノイズに強く、回路がシンプル
- 文字・画像・音・動画は**数字で表現**できる（文字コード／ピクセルRGB／サンプリング／フレーム）
- **圧縮＝「繰り返しを短く」＋「人の感覚の鈍感さを上手に使う」**

できるようになること

- 10進／2進／16進の**簡単な変換**ができる
- **ASCII と Unicode**の違いを例で説明できる
- **1ピクセルのRGB**とカラーコードの関係が分かる
- **音のサンプリング**（fs とビット深度）の意味を説明できる
- **可逆／非可逆**の違いを、**ZIP・PNG／JPEG・MP3**で説明できる

次回予告

次回：『0』と『1』だけで、
どうやって足し算をする？～
計算する機械の心臓部～