UTF-8

UTF-16

UCS-2

UNICODE ENCODINGS

UTF-32

UNICODE SYMBOLS INCLUDE:

3	A	;	^	丌	⇨	💩
0x0033	0x0041	0x003b	0x0302	0x110e	0x21e8	0x1f4a9

AND (SO FAR) ~144,000 MORE

WITH ~1,000,000 EMPTY SLOTS

EVERY SYMBOL HAS A CODEPOINT.

THIS IS ITS INDEX; ITS IDENTIFIER.

	HEX	DEC	BINARY
=	0x003d	61	0b111101
>	0x003e	62	0b111110
?	0x003f	63	0b111111
@	0x0040	64	0b100000
A	0x0041	65	0b100001
B	0x0042	66	0b100010
C	0x0043	67	0b100011
D	0x0044	68	0b100100
⋮			

COMPUTERS USE BINARY

THE TEXT MESSAGE "Hə 💩"

WOULD BE STORED AS

H	72
ə	97
" "	32
💩	128, 169

SO HOW DO WE STORE THIS IN

BINARY

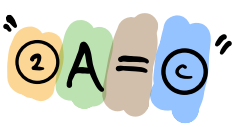
IF WE ONLY HAVE BINARY, THE "BLOB" MIGHT LOOK LIKE THIS


```
1001000 1100001 100000 1111101
00101010 01
```

BUT WHEN WE RECEIVE IT, HOW DO WE READ IT? WHERE DO SYMBOLS STOP & START?

```
1001000110000011000000111110100101010 01
```


```
1001000110000011000000111110100101010 01
```



WE NEED A FIXED CHUNK SIZE

UNICODE ALLOWS FOR CODEPOINTS AS LARGE AS 21 BITS

WHAT ABOUT 21-BIT CHUNKS?

UTF-32

AKA. UCS-4

COMPUTERS ARE GENERALLY FASTER WITH POWERS OF 2, SO LET'S USE

32 INSTEAD

Ha 

```
00000000000000000000000001001000
00000000000000000000000001100001
00000000000000000000000000100000
00000000000000001111010010101001
```

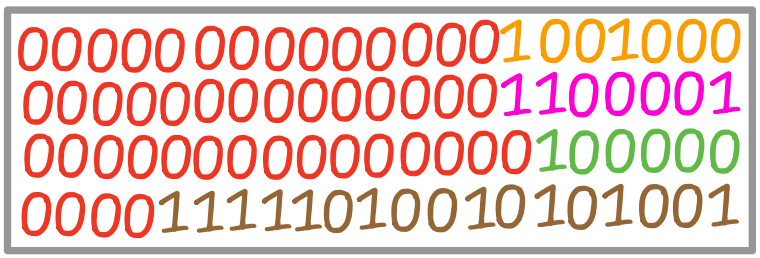
IT'S NOW EASY TO READ, BUT WE'RE WASTING

SO MUCH SPACE

(FOR COMPUTERS)

```
00000000000000000000000001001000
00000000000000000000000001100001
00000000000000000000000000100000
00000000000000001111010010101001
```

22-BIT CHUNKS AREN'T MUCH BETTER



WE NEED A WAY TO ENCODE TEXT

EFFICIENTLY

LET'S GO SMALLER THAN 32

LET'S GO SMALLER THAN 21

LET'S GO TO 16

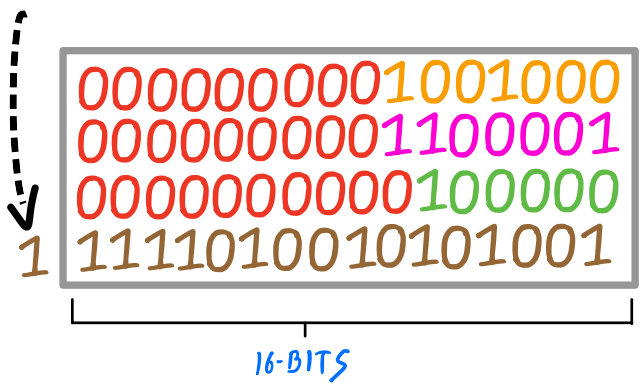
UTF-16

AND UCS-2

BEFORE THE YEAR 2000, ONLY THE FIRST ~65K CODEPOINTS WERE IN USE

WITH UCS-2, WE JUST USED 16-BIT CHUNKS

BUT TODAY... WE NEED MORE



TO REMEDY THIS, WE CAME UP WITH A PLAN.

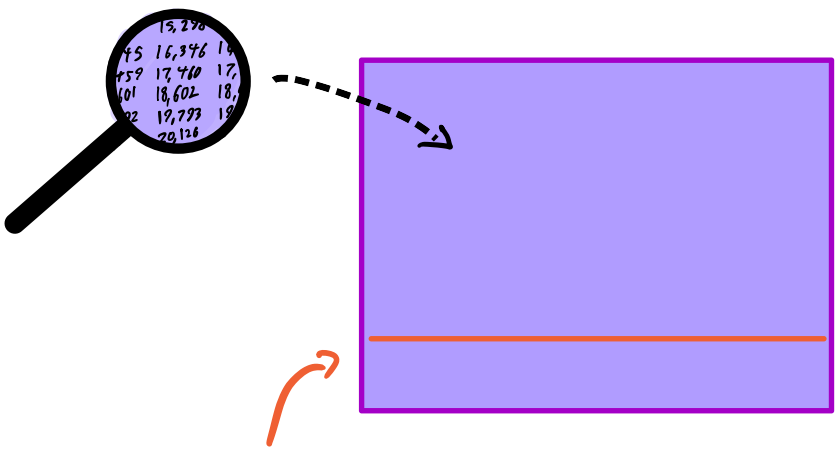
A PLAN CALLED

SURROGATE PAIRS

UTF-16 OUTLINED IT AS FOLLOWS:

PRETEND THIS BLOCK CONTAINS
THE FIRST ~65K CODEPOINTS

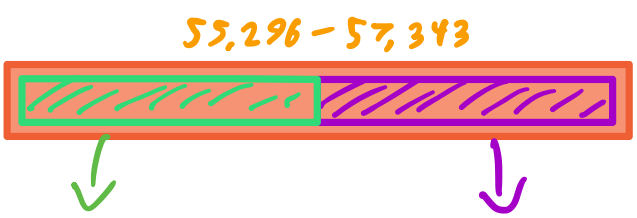
(THE ONES WE CAN CONVEY WITH 16 BITS)



THESE ARE THE SURROGATE PAIRS

(~55K → ~57K)

WE CAN SPLIT THESE ~2K CODEPOINTS
IN HALF TO MAKE 2 GROUPS



HIGH SURROGATES
55,296 - 56,319

LOW SURROGATES
56,320 - 57,343

WHY DO WE CARE?

THESE ARE RESERVED FOR UTF-16

IF A CODEPOINT IS TOO LARGE FOR 16 BITS,
WE REPRESENT IT WITH SURROGATE PAIRS

ONE HIGH, ONE LOW

LET'S RUN THROUGH THE ALGORITHM

WITH 

1. GET ITS CODEPOINT

    = 128,169 (0x1F4A9)

2. SUBTRACT 65,536 (0x10000)

   128,169 - 65,536 = 62,633

3. MOVE TO BINARY. MAKE IT 20 DIGITS.

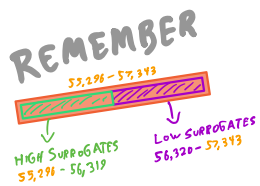
   62,633 = 0b00001111010010101001

4. GET FIRST & LAST 10 DIGITS.

   0b0000111101|0010101001

   0b0000111101 0b0010101001
   61 169

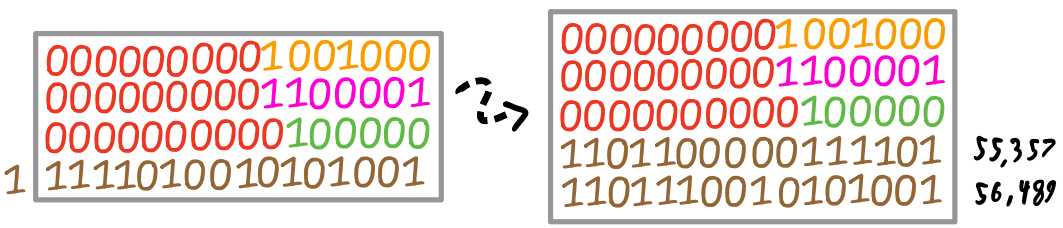
5. ADD EACH NUMBER TO THE START OF THEIR SURROGATE RANGE.



55,296 + 61 = 55,357

56,320 + 169 = 56,489

AND THERE YOU HAVE IT!



BUT... THERE STILL MIGHT BE ROOM FOR IMPROVEMENT

CHARACTERS WITH SMALL CODEPOINTS SHOW UP MUCH MORE OFTEN IN:

- CODE
- BLOGS
- ARTICLES
- PAPERS

AND ESPECIALLY WHEN USING THE

LATIN ALPHABET

UTF-8

WHAT IF WE WERE LIMITED TO

8 BITS ?

01001000
01100001
00100000
10101001

111110100

SO... WHAT NOW?

DO WE RESERVE CHARACTERS AGAIN?

THERE AREN'T ENOUGH BITS

UTF-8 WORKS BASED ON
A SERIES OF RULES/CASES

LET'S STEP THROUGH THEM

FOR ANY CHARACTER, GET ITS CODEPOINT

CASE 1

CODEPOINT < 128 (10000000)
(7-BIT MAXIMUM)

Do

PAD WITH 0 UNTIL 8 BITS

EXAMPLE

! (33) → (Expand) 00100001

g (103) → (Expand) 01100111

CASE 2

128 ≤ CODEPOINT < 2048

Do

PAD WITH 0 UNTIL 11 BITS

FIRST BYTE = 110 + FIRST 5 BITS

SECOND BYTE = 10 + NEXT 6 BITS

EXAMPLE

£ (163) → 10100011

Expand 00010100011

110 (append) 00010 10 (append) 100011

CASE 3

$2048 \leq \text{CODEPOINT} < 65,536$

Do

PAD WITH 0 UNTIL 16 BITS

FIRST BYTE = 1110 + FIRST 4 BITS

2nd & 3rd BYTE = 10 + NEXT 6 BITS

EXAMPLE

→ (65,515) → 1111111111101011

~~Expand~~ 1111|111111|101011

1st 11101111 2nd 10111111 3rd 10101011

CASE 4

$65,536 \leq \text{CODEPOINT} <$

Do

PAD WITH 0 UNTIL 21 BITS

FIRST BYTE = 11110 + FIRST 3 BITS

2nd, 3rd, & 4th BYTE = 10 + NEXT 6 BITS

EXAMPLE

(128,169) → 1111101001010101001

Expand 0000 1111 0100 101001

1st 11110000

2nd 10011111

3rd 10010010

4th 10101001

IN OTHER WORDS:

<7 bits = START WITH 0s UNTIL 8 bits.

>7 bits = START WITH AS MANY 1s AS HOW MANY BYTES YOU NEED. THEN ADD A 0.
FOR EVERY BYTE AFTER THE FIRST, START WITH 10.
ADD 0s AS NEEDED (TO FILL UP ENOUGH BITS FOR A BYTE).

SO NOW OUR MESSAGE IS

7 BYTES

01001000
01100001
00100000
11110000
10011111
10010010
10101001

0123456789