Text Search

Steven R. Bagley

Introduction

- Last lecture, looked at Text Search algorithms
- Saw how by careful constructing our algorithm
- We can massively reduce the time taken to search

String Search

- Definitions
  - The *pattern* — string to search for
  - The *text* — the text we want to search
- Problem
  - Does *pattern* occurs inside *text*

Will use these throughout the lecture

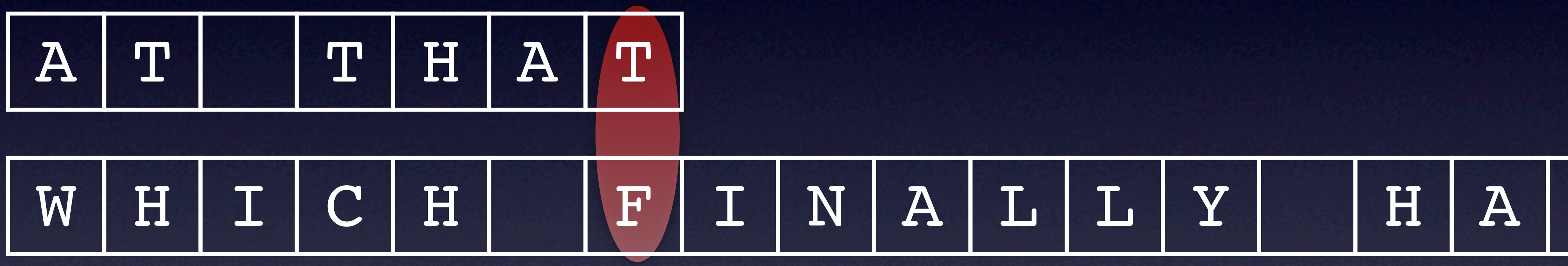
Text Search

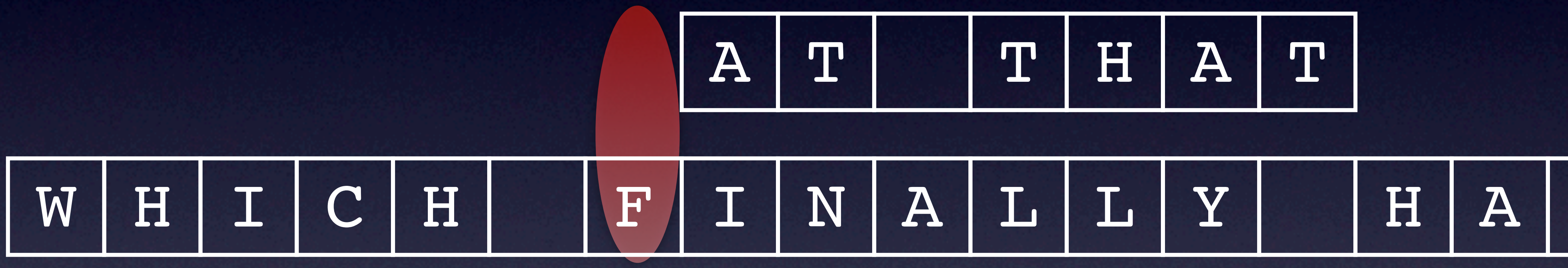
- Naive Search — Test *pattern* in every possible alignment with *text*
- Boyer-Moore — Uses information about the mismatches to skip big chunks of text

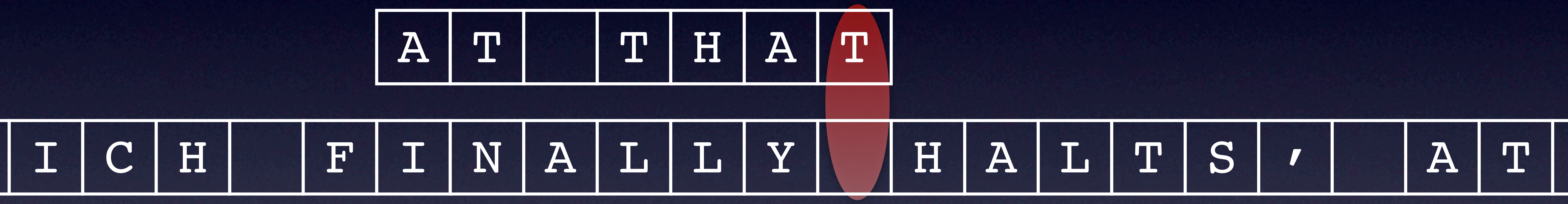
Boyer-Moore

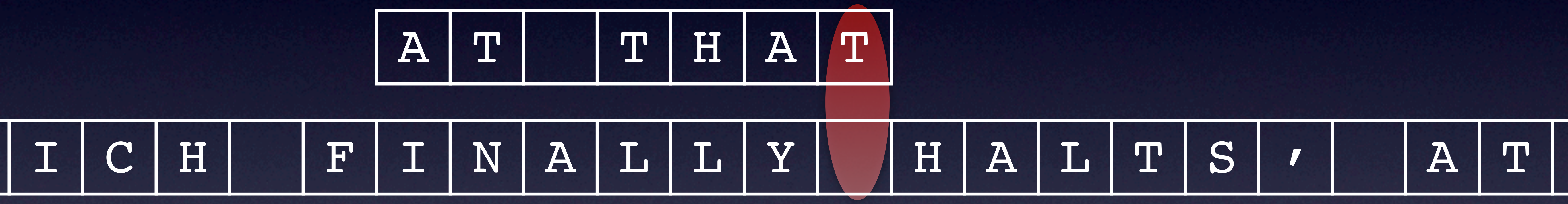
- If mismatching char not in *pattern* slide length of *pattern*
- Slide so the rightmost occurrence of char in *pattern* is aligned with char in *text*
- Slide so that a matched subpattern is aligned with the rightmost occurrence of the subpattern not preceded by this mismatched char

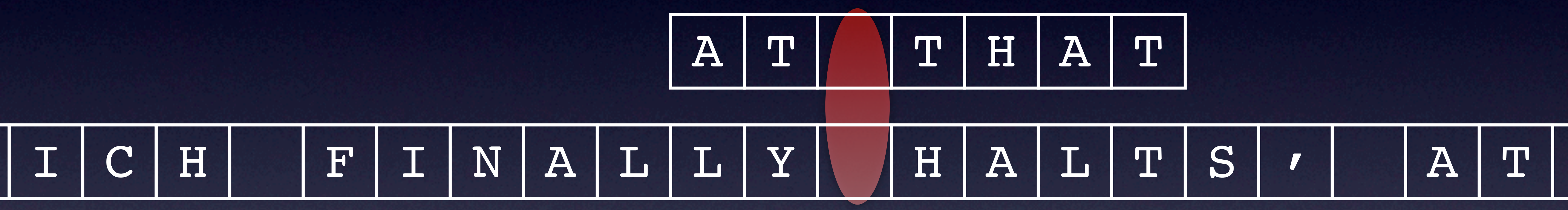
(assuming this is not backwards)


A T   T H A T
W H I C H   F I N A L L Y   H A

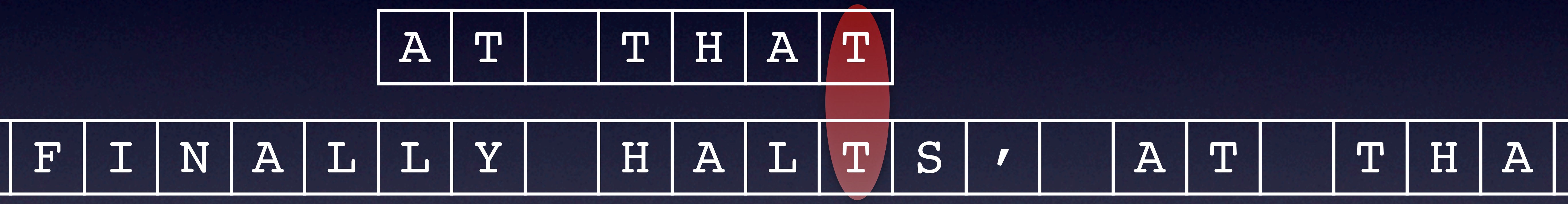

A T   T H A T
W H I C H   F I N A L L Y   H A


A T   T H A T
I C H   F I N A L L Y   H A L T S ,   A T

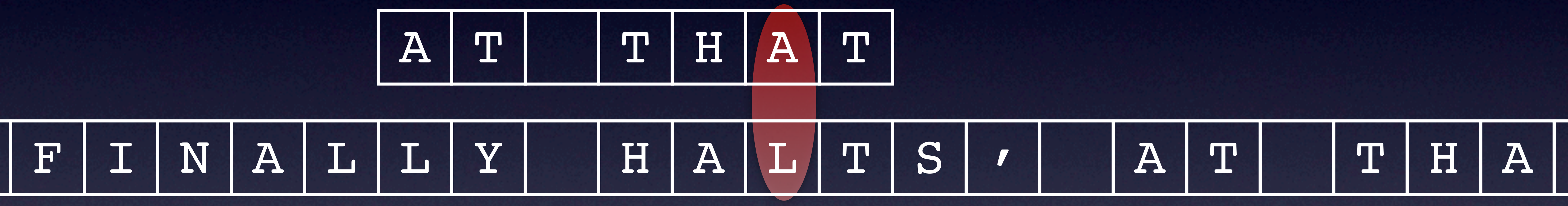

AT THAT
ICH FINALLY HALTS', AT



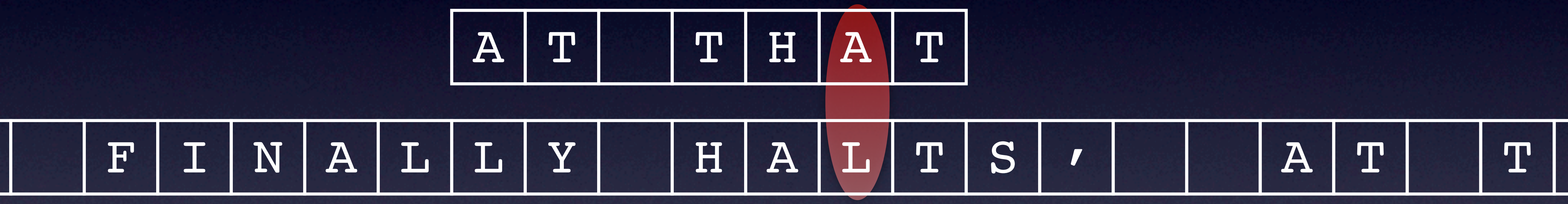
Okay, found a character that matches, step back to test the previous character



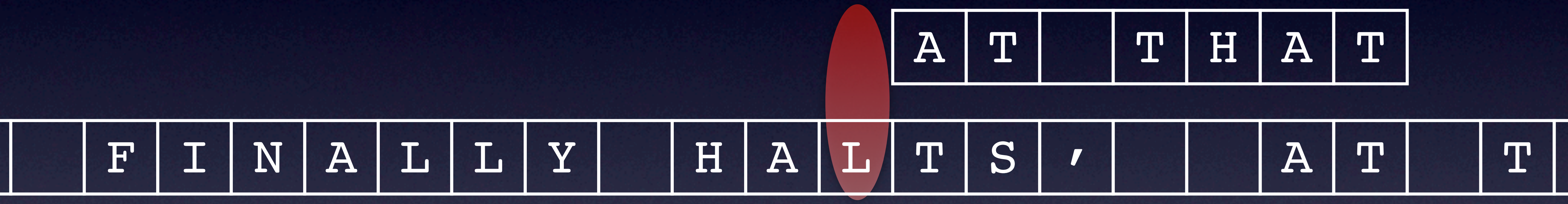
Okay, found a character that matches, step back to test the previous character



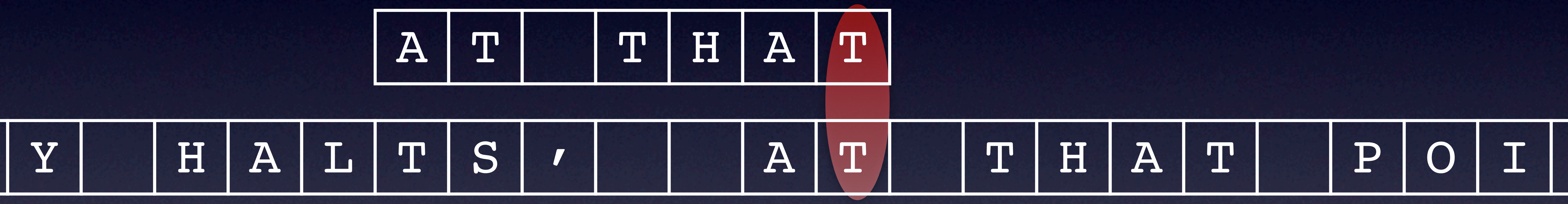
Okay, found a character that matches, step back to test the previous character



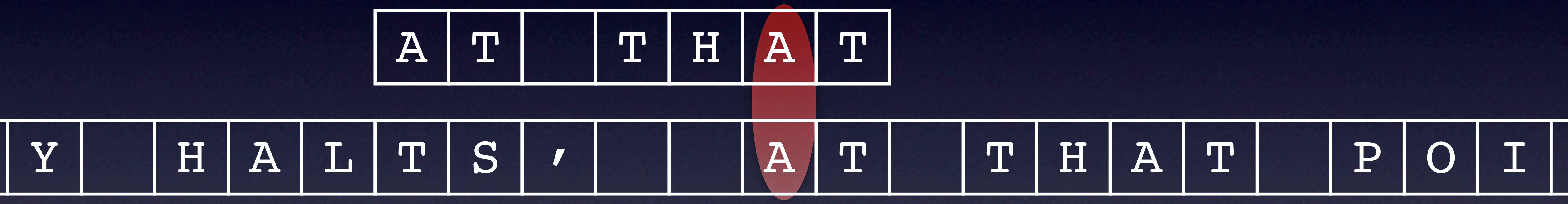
Okay, found a character that matches, step back to test the previous character



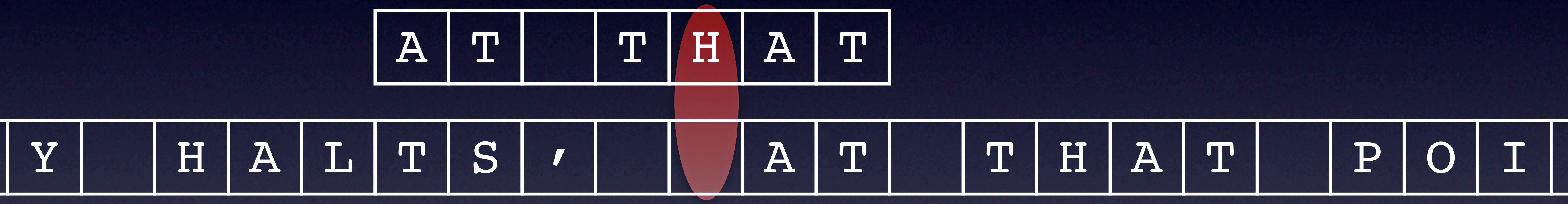
Okay, found a character that matches, step back to test the previous character



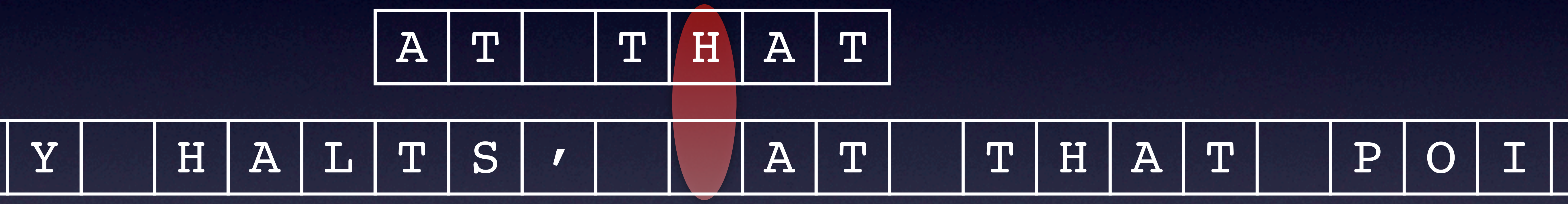
Okay, found a character that matches, step back to test the previous character



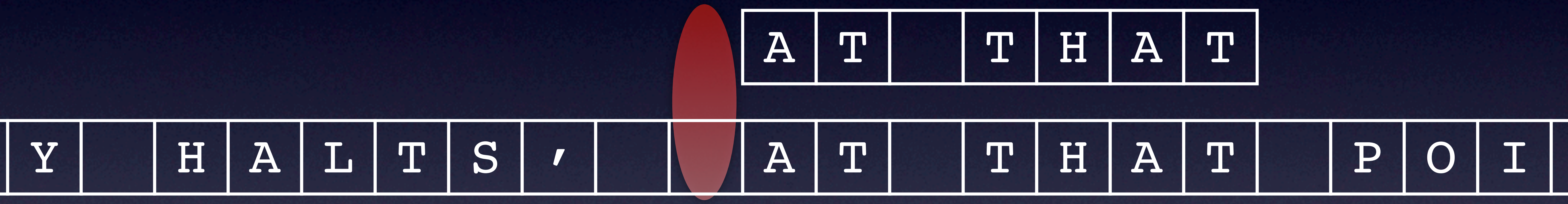
Okay, found a character that matches, step back to test the previous character



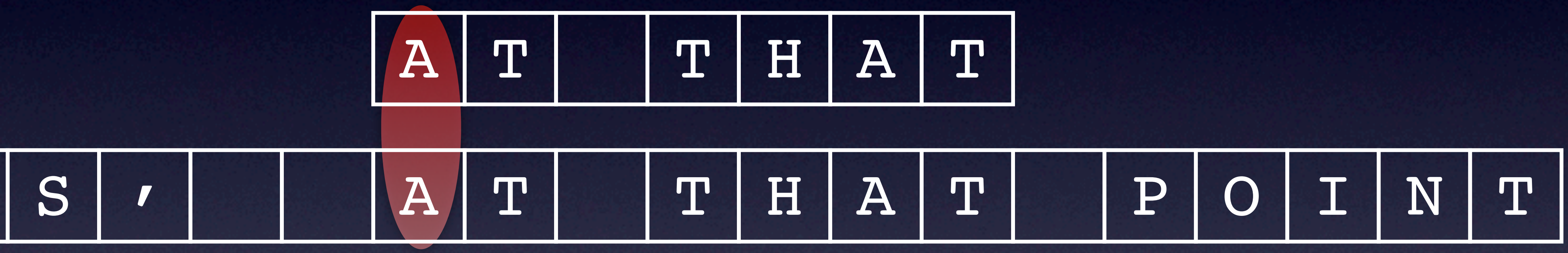
Okay, found a character that matches, step back to test the previous character



Okay, found a character that matches, step back to test the previous character



Look we’ve matched the string



Look we’ve matched the string

Boyer-Moore

- Nice, fast algorithm
- Easy to implement, although requires some largish tables
- The ‘gold standard’, other algorithms compared to this
- Longer the *pattern* the faster it tends…

Boyer-Moore

- But can break down if the alphabet is small e.g. DNA sequences – only four letters (`TACG`)
- Much more likely that the character will be found close to the right edge
- So the *pattern* will move by a much smaller amount each time

Search Algorithms

- However, other algorithms don’t suffer from this problem
- One such algorithm makes use of the Burrows-Wheeler Transform…

Burrows-Wheeler Transform

- Developed in 1994 by Mike Burrows and David Wheeler
- Takes a string and transforms it into another string
- However, this transformed string is much more compressible
- What’s compression got to do with search?

David Wheeler one of the original computer people at Cambridge
Mike Burrows went on to design Altavista and now works at Google (via MS)

Suffix Arrays

- Helps to understand BWT search if we first look at a simpler related search algorithm
- The Suffix Array is an array of all possible suffixes of *text*
- Where a suffix is a substring that ends with the last character of *text*

Gives some examples with Mississippi, e.g.
i, pi, ippi, ssippi, Mississippi are all suffixes
But Miss, and ssiss are not (because they don’t run to the end of the string)

0	MISSISSIPPI
1	ISSISSIPPI
2	SSISSIPPI
3	SISSIPPI
4	ISSIPPI
5	SSIPPI
6	SIPPI
7	IPPI
8	PPI
9	PI
10	I

All possible suffixes of Mississippi

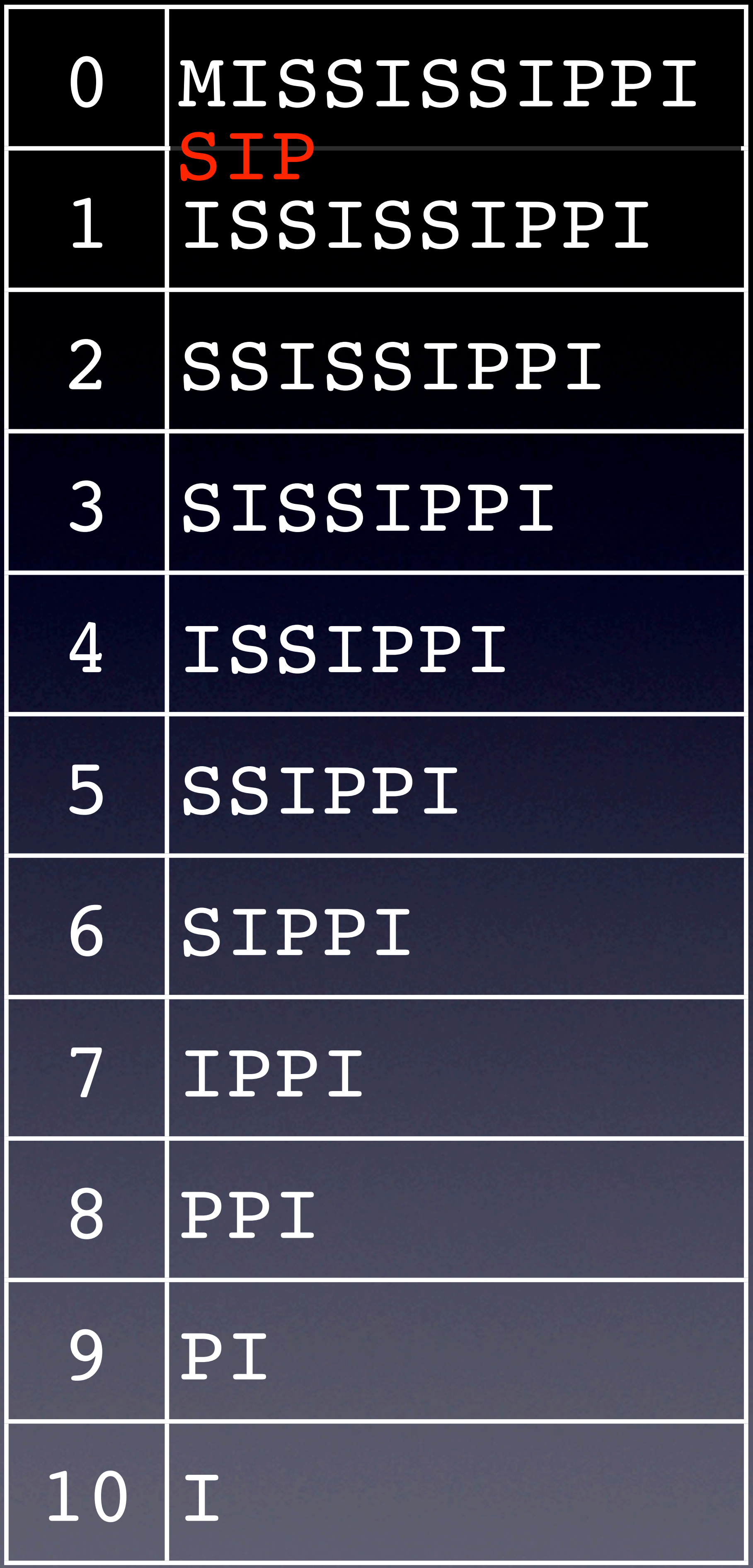
Searching Suffix Arrays

- If *pattern* exists within the text then it will be a prefix of one the suffix array entries
- Therefore, if we compare the *pattern* with the beginning of each suffix
- We can find our string
- It's exactly the same as our original naive search…

prefix, one array will start with it

0	MISSISSIPPI
1	ISSISSIPPI
2	SSISSIPPI
3	SISSIPPI
4	ISSIPPI
5	SSIPPI
6	SIPPI
7	IPPI
8	PPI
9	PI
10	I

Suppose we are search for the pattern ‘SIP’



0	MISSISSIPPI SIP
1	ISSISSIPPI
2	SSISSIPPI
3	SISSIPPI
4	ISSIPPI
5	SSIPPI
6	SIPPI
7	IPPI
8	PPI
9	PI
10	I

Suppose we are search for the pattern ‘SIP’

0	MISSISSIPPI
1	ISSISSIPPI
2	SSISSIPPI
3	SISSIPPI
4	ISSIPPI
5	SSIPPI
6	SIPPI
7	IPPI
8	PPI
9	PI
10	I

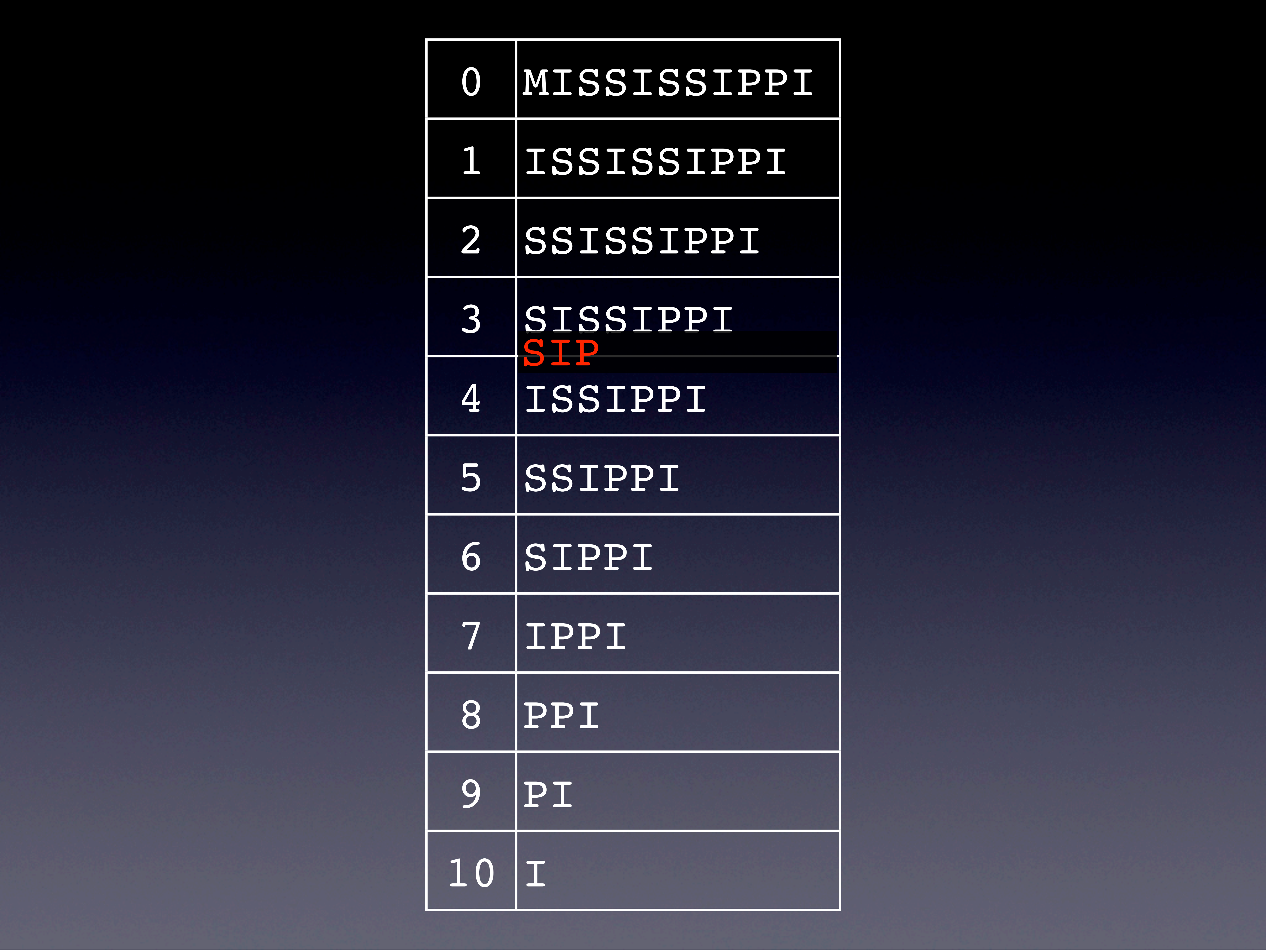
SIP

Suppose we are search for the pattern ‘SIP’

0	MISSISSIPPI
1	ISSISSIPPI
2	SSISSIPPI
3	SISSIPPI
4	ISSIPPI
5	SSIPPI
6	SIPPI
7	IPPI
8	PPI
9	PI
10	I

SIP

Suppose we are search for the pattern ‘SIP’



0	MISSISSIPPI
1	ISSISSIPPI
2	SSISSIPPI
3	SISSIPPI
	SIP
4	ISSIPPI
5	SSIPPI
6	SIPPI
7	IPPI
8	PPI
9	PI
10	I

Suppose we are search for the pattern ‘SIP’

0	MISSISSIPPI
1	ISSISSIPPI
2	SSISSIPPI
3	SISSIPPI
4	ISSIPPI

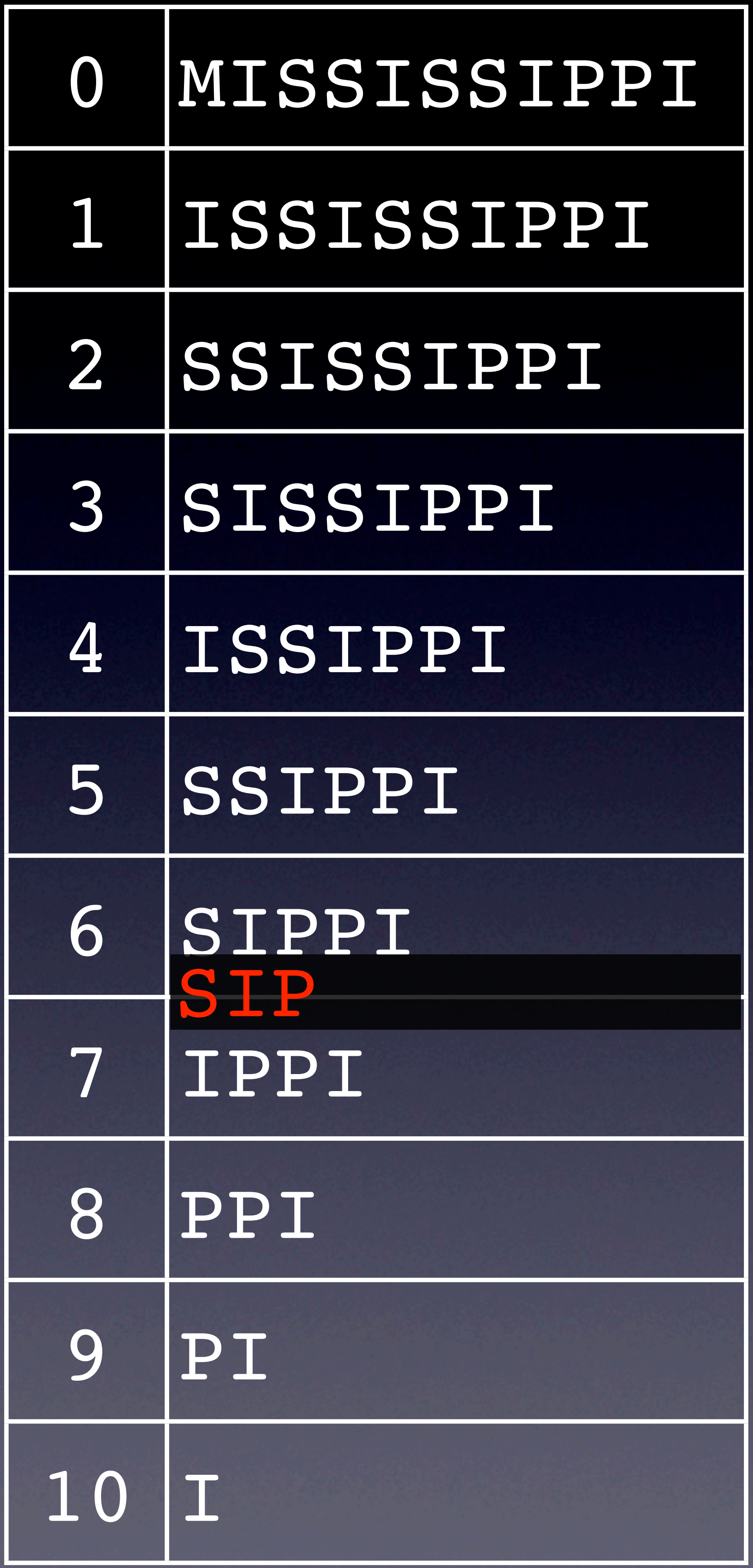
SIP

5	SSIPPI
6	SIPPI
7	IPPI
8	PPI
9	PI
10	I

Suppose we are search for the pattern ‘SIP’

0	MISSISSIPPI
1	ISSISSIPPI
2	SSISSIPPI
3	SISSIPPI
4	ISSIPPI
5	SSIPPI
	SIP
6	SIPPI
7	IPPI
8	PPI
9	PI
10	I

Suppose we are search for the pattern ‘SIP’



0	MISSISSIPPI
1	ISSISSIPPI
2	SSISSIPPI
3	SISSIPPI
4	ISSIPPI
5	SSIPPI
6	SIPPI
7	IPPI
8	PPI
9	PI
10	I

SIP

Suppose we are search for the pattern ‘SIP’

Suffix Array Search

- Just as slow as our original naive algorithm
- Also need to build up array (more RAM)
- That's true — but if you sort the suffix array, you can use binary search
- Finds *pattern* in O(`m log n`) time
- Also, finds all occurrences

more RAM than the naive approach, but probably about the same as Boyer-Moore

10	I
7	IPPI
4	ISSIPPI
1	ISSISSIPPI
0	MISSISSIPPI
9	PI
8	PPI
6	SIPPI
3	SISSIPPI
5	SSIPPI
2	SSISSIPPI

Suppose we are search for the pattern ‘SIP’
Oh look found it in two

10	I
7	IPPI
4	ISSIPPI
1	ISSISSIPPI
0	MISSISSIPPI
9	PI
	SIP
8	PPI
6	SIPPI
3	SISSIPPI
5	SSIPPI
2	SSISSIPPI

Suppose we are search for the pattern ‘SIP’
Oh look found it in two

10	I
7	IPPI
4	ISSIPPI
1	ISSISSIPPI
0	MISSISSIPPI
9	PI
8	PPI
6	SIPPI
	SIP
3	SISSIPPI
5	SSIPPI
2	SSISSIPPI

Suppose we are search for the pattern ‘SIP’
Oh look found it in two

Burrows-Wheeler

- The BWT is incredible simple to implement
- Although understand how it works is not so simple
- BWT takes a string, and produces an *NxN* matrix (where *N* is the length of the string)
- First row of the matrix is just the string

BWT == Burrows-Wheeler Transform

M	I	S	S	I	S	S	I	P	P	I

Burrows-Wheeler

- For every other row take the previous row and rotate it to the left by one…
- Putting the character shifted out the left back round into the right
- The resulting matrix is then sorted lexicographcially

M	I	S	S	I	S	S	I	P	P	I

M	I	S	S	I	S	S	I	P	P	I
I	S	S	I	S	S	I	P	P	I	M

M	I	S	S	I	S	S	I	P	P	I
I	S	S	I	S	S	I	P	P	I	M
S	S	I	S	S	I	P	P	I	M	I

M	I	S	S	I	S	S	I	P	P	I
I	S	S	I	S	S	I	P	P	I	M
S	S	I	S	S	I	P	P	I	M	I
S	I	S	S	I	P	P	I	M	I	S
I	S	S	I	P	P	I	M	I	S	S
S	S	I	P	P	I	M	I	S	S	I
S	I	P	P	I	M	I	S	S	I	S
I	P	P	I	M	I	S	S	I	S	S
P	P	I	M	I	S	S	I	S	S	I
P	I	M	I	S	S	I	S	S	I	P
I	M	I	S	S	I	S	S	I	P	P

I	M	I	S	S	I	S	S	I	P	P
I	P	P	I	M	I	S	S	I	S	S
I	S	S	I	P	P	I	M	I	S	S
I	S	S	I	S	S	I	P	P	I	M
M	I	S	S	I	S	S	I	P	P	I
P	I	M	I	S	S	I	S	S	I	P
P	P	I	M	I	S	S	I	S	S	I
S	I	P	P	I	M	I	S	S	I	S
S	I	S	S	I	P	P	I	M	I	S
S	S	I	P	P	I	M	I	S	S	I
S	S	I	S	S	I	P	P	I	M	I

Lexicogrpahically sorted version!

BWT

- The last column of this is then extracted as the product of the BWT
- This transformed string made from the original string is highly compressible

Move-To-Front Encoding, then huffman coding

I	M	I	S	S	I	S	S	I	P	P
I	P	P	I	M	I	S	S	I	S	S
I	S	S	I	P	P	I	M	I	S	S
I	S	S	I	S	S	I	P	P	I	M
M	I	S	S	I	S	S	I	P	P	I
P	I	M	I	S	S	I	S	S	I	P
P	P	I	M	I	S	S	I	S	S	I
S	I	P	P	I	M	I	S	S	I	S
S	I	S	S	I	P	P	I	M	I	S
S	S	I	P	P	I	M	I	S	S	I
S	S	I	S	S	I	P	P	I	M	I

Last column taken and used as basis for compression

```
t,<s]]meem =========-+-
<<nhiiiiix]mhiiiinx]]]te]eskt]]tiejm]iiijsc
wwewewes*e  m mte=e wwt= =+
+- ;t;cdm==mnnnt     ;;;;;tttt tt;;;;;m
wwt;;tt=,on;t;======= m=e t= tn==,,<<<<<==t
y)n};t;=        (r" ((nnnrtthkhhereydyk
mrrrrrrrrreme ntf+)f+e++tt+-+++s(tt)(stee)
("](ketx"1/ }../   ++++++++iniiixih"et]d
-jlommta]mm]trtrssssa]t**      =    =
[222555))))))))))k)]])])ste]eentste]*0]1])x
]])]0]]])*]0600000eee6e100)0e
=               >        dd  t ( TTZZZ(eee
OOFO"ed (aaa(dadds IIE ("tttttttt
WWsssBBBBBtcpetkcccccccpcpekkeeekkkktyeet]
iij]ii[[hiiiiie[[eeei]h[6h]iiii
```

Extract from a compressed Java program — lots of repeated characters

BWT

- Reversible transform – given the last column, the entire matrix can be regenerated
- Store index of the original string row
- Exploits the fact that the first column can be made by sorting the last column
- And that every character in the last column precedes the character in the first column

BWT

- The first characters of each entry of the BWT matrix are identical to the strings in the suffix array
  - Followed by the rest of the string
- Therefore, we can do a binary search in the BWT matrix to find the *pattern*

I	M	I	S	S	I	S	S	I	P	P
I	P	P	I	M	I	S	S	I	S	S
I	S	S	I	P	P	I	M	I	S	S
I	S	S	I	S	S	I	P	P	I	M
M	I	S	S	I	S	S	I	P	P	I
P	I	M	I	S	S	I	S	S	I	P
P	P	I	M	I	S	S	I	S	S	I
S	I	P	P	I	M	I	S	S	I	S
S	I	S	S	I	P	P	I	M	I	S
S	S	I	P	P	I	M	I	S	S	I
S	S	I	S	S	I	P	P	I	M	I

Last column taken and used as basis for compression

10	I
7	IPPI
4	ISSIPPI
1	ISSISSIPPI
0	MISSISSIPPI
9	PI
8	PPI
6	SIPPI
3	SISSIPPI
5	SSIPPI
2	SSISSIPPI

Suppose we are search for the pattern 'SIP'
Oh look found it in two

BWT search

- The BWT matrix will be bigger than the Suffix array
- But fortunately, there is a way to build up any line of the sorted matrix in linear time
- Using just the last line (and a couple of easily precomputed tables)

```
int sum = 0, c[] = new int[256], p[] = new int[size];
byte out = new byte[size];

for(int i = 0; i < 256; i++) c[i] = 0;

for(int i = 0; i < size; i++)
{
    p[i] = c[block[i]];
    c[block[i]] = c[block[i]] + 1;
}

for(int ch = 0; ch < 256; ch++)
{
    sum = sum + c[ch];
    c[ch] = sum - c[ch];
}

int i = x;
for(int j = size - 1; j >=0; j--)
{
    out[j] = block[i];
    i = p[i] + c[block[i]];
}
```

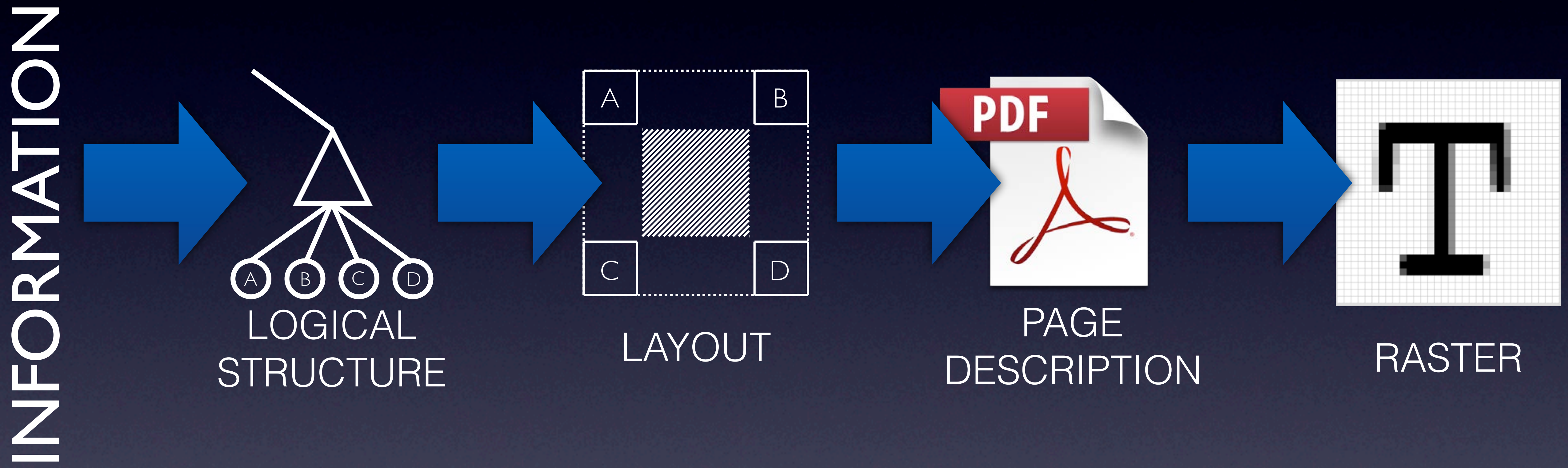
Java code to find row x based on the last column block[]
out[] contains row x
p[] and c[] are two tables used to help rebuild the row

BWT search

- Therefore, using the last line of the matrix
  - Which could well all ready exist if we've compressed *text*
- The reconstruction algorithm and a binary search
- We can find all occurrences of a *pattern* very quickly

e.g. on disk

Page Description Langauges



INFORMATION
A
B
C
D
LOGICAL STRUCTURE
A
B
C
D
LAYOUT
PDF
PAGE DESCRIPTION
RASTER

Page Description Language

- Holds a description of a ‘page’ in a form that can be converted to a viewable image
- At this level everything is in a fixed position
- Dominated now by the Adobe Graphics Model
- Used by PDF, PostScript and SVG

As well, as iOS and OS X

Early PDLs

- Before PostScript, PDLs were very basic
- Weren't necessarily driving a raster device
- Were also tied to a specific machine
- And its specific features
  - Such as units used for measurements

Technically, Warnock and Geschke had done very similar work at Xerox (Interpress) and at Evans and Sutherland to what would become PostScript
Similar to the operations you get in a typical grpahics library on a computer
Programs driving a Linotype 202 would be incompatible with Monotype machines

Early PDLs

- Aimed at *typ*esetters
- Mainly limited to text – set font, point size, position, print text
- Perhaps some very basic straight line drawing operations

Technically, Warnock and Geschke had done very similar work at Xerox (Interpress) and at Evans and Sutherland to what would become PostScript
Similar to the operations you get in a typical grpahics library on a computer

Early Software Driving

- Software would often want to drive multiple devices
- Produced an intermediate code, which would be converted to the relevant format
- Often by a separate backend program
- But the intermediate codes would be incompatible across software packages

Meaning you had lots of different backends

PostScript

- Developed by John Warnock and Chuck Geschke in early 1980s
- PostScript took a different approach
- Not aimed at driving typesetters, abstracted away from a specific device
- Text was just another mark on the page
- Based around the Adobe Graphics Model

Adobe Graphics Model

- At the heart of PDF, Postscript and SVG
- Separates the Graphics from the Rendering
- Application composes graphics on the page
- Once completed the entire page is rendered for a specific device
- Crucially, abstracts the application from device

Might be useful to know if its colour/black and white, what fonts are installed…
But that’s general metadata

Adobe Graphics Model

- Based around the idea of painting marks on a blank page
- Any mark made obscures any previous mark underneath it
- Although modern versions allow for transparency

Adobe Graphics Model

- Painted Marks may be:
  - Character shapes (glyphs)
  - Geometric shapes or lines (paths)
  - Sampled images (bitmaps)

Paths

- Constructed from a sequence of lines, curved and points
- Then filled or stroked

Adobe Graphics Model

- Painted Marks may be in
  - Colour
  - Greyscale
  - A repeating Pattern
  - Smooth transition between colours (gradient)

Adobe Graphics Model

- Painted marks may also be clipped to some other shape
- Clipping defined by a path
- As they are placed on the page
- Once its on the page it can’t be removed
- Only covered up by new marks

User Space

- AGM defines *User Space* which is what the application draws in
- Infinite, but clipped to the display…
- `XY`-coordinate system, `y` +ve upwards and `x` `+ve` to the right
- Origin is `(0,0)` — i.e. the bottom-left
- Although the user can change this

AGM defines lots of interelated spaces…

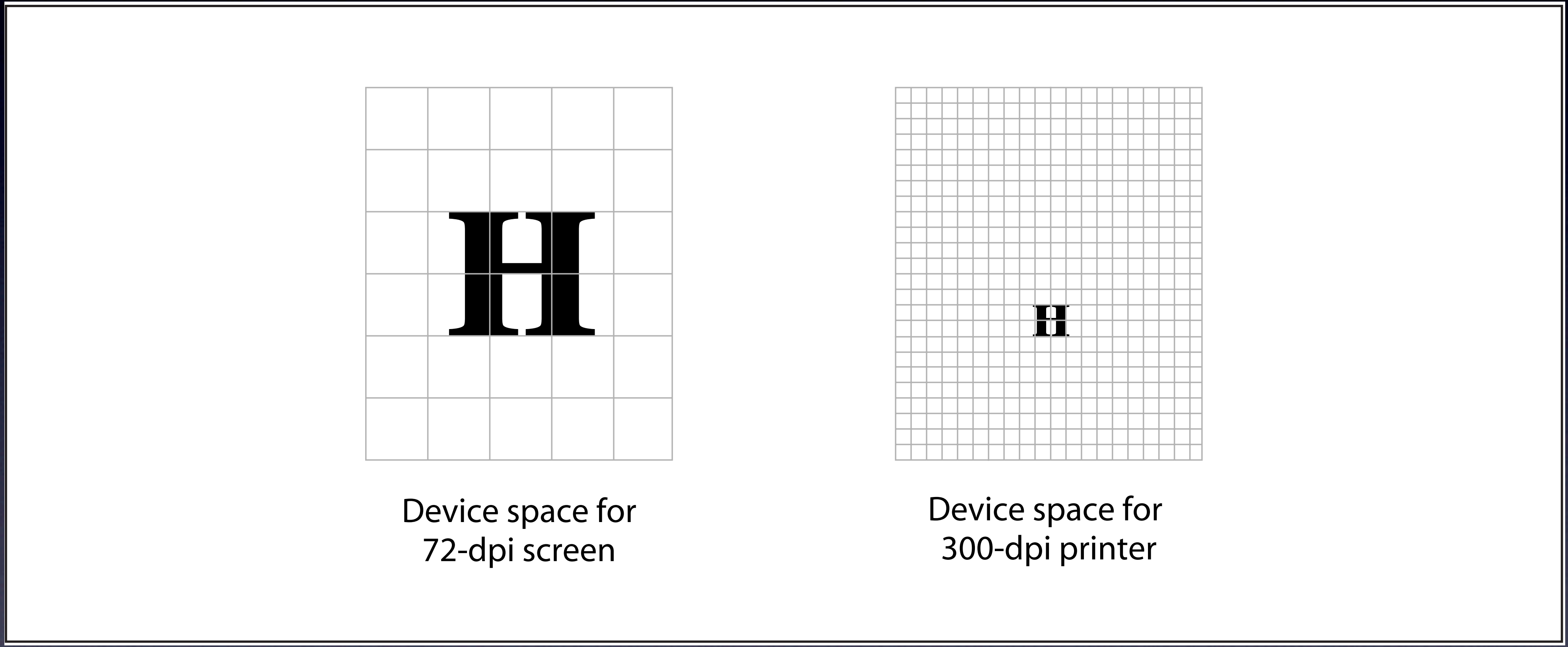
User Space

- Also defines a fixed unit for User Space
- The point — 1/72 inch
- Unrelated to the capabilities of the device
- So is a floating point value
- Before being rendered, user space coordinates are *transformed* into *Device Space*

Device Space

- Device Space is specific to the device
- Varies from device to device
- Different origin, axes, units for measurement
- Specifying graphics in *User Space* and transforming to *Device Space* gives device independence

Think about it — if you ask for a box to be 72 points high on a 72dpi machine its an inch,
On a 300dpi machine it's ~0.25 inch high

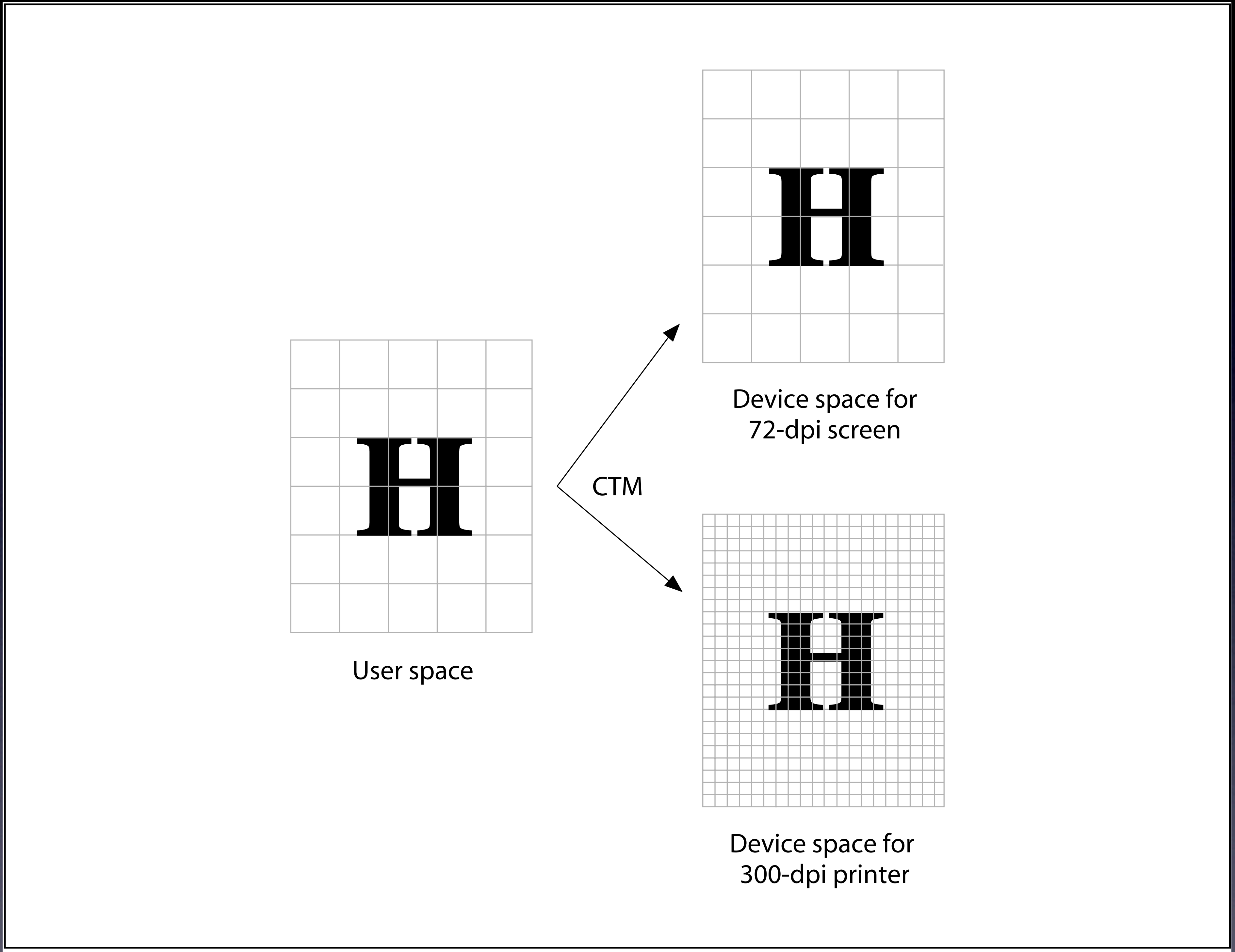


Example of difference in apparent size of object with same dimensions on different devices

Current Transformation Matrix

- Current Transformation Matrix (CTM) specifies translation
- Allows the coordinates to be scaled, translated, rotated and sheared
- Starts with a default CTM (maps User Space to a specific Device Space)
- Could be more than just scaling

May have a different origin, axis direction etc as well


Device space for
72-dpi screen
CTM
User space
Device space for
300-dpi printer