Text Search

Steven R. Bagley

Introduction

- Briefly look at how we might search for a string within a piece of text
- Good illustration of how to write good text processing algorithms
- That are efficient…

Text tends to be long… so don't really want to execute an O(n^2) algorithm

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

Naive String Search

- Easy to generate simple algorithm
- Align *pattern* with the beginning of *text*
- Compare the first character of *pattern* with the corresponding character in *text*


H e l l o
H e l l o   W o r l d

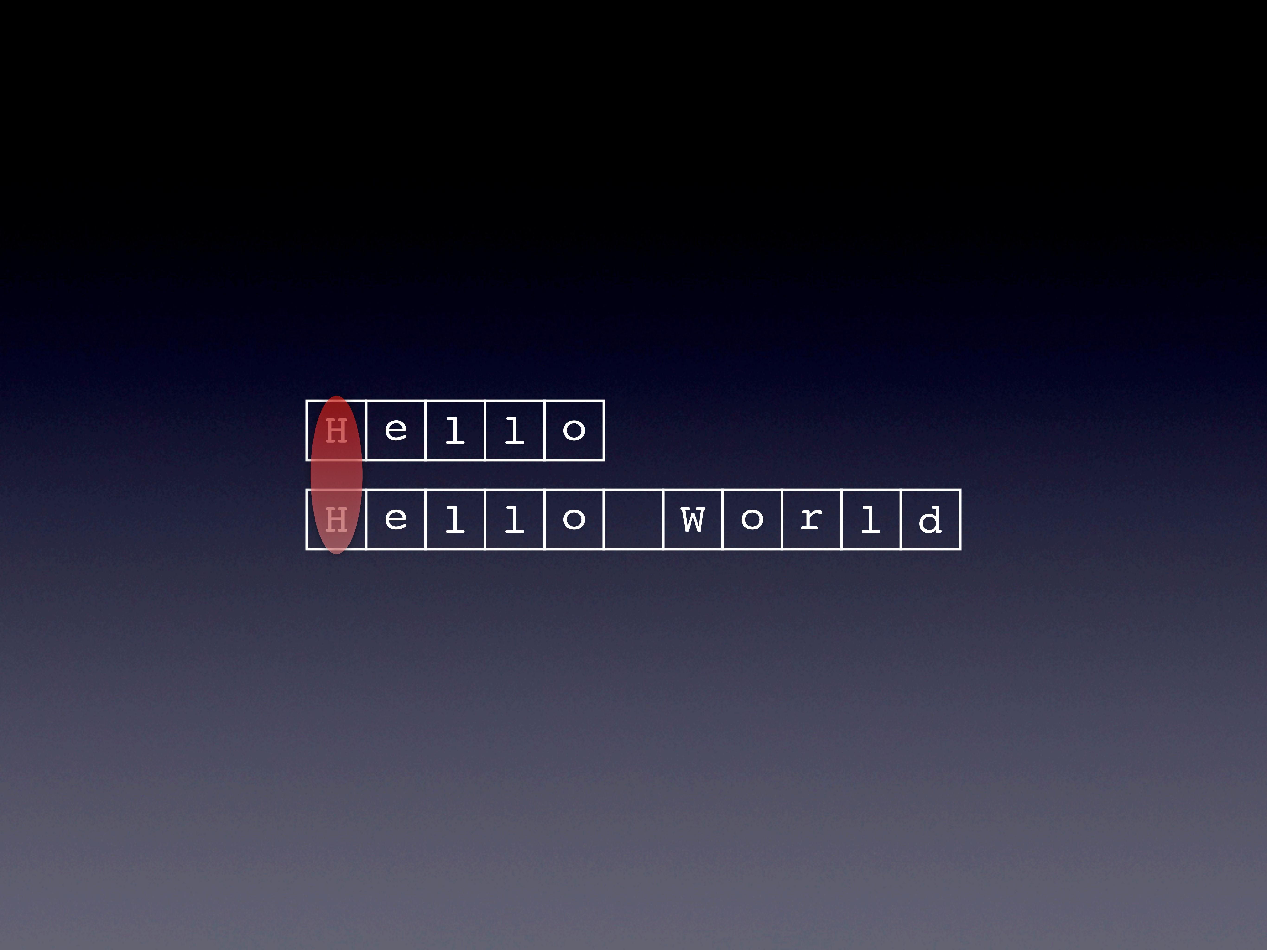
Naive String Search

- Easy to generate simple algorithm
- Align *pattern* with the beginning of *text*
- Compare the first character of *pattern* with the corresponding character in *text*
- Then…

If they match

- Compare the next character of the *pattern* with the corresponding character of *text*
- And repeat…
- If we reach the end of *pattern* then we've found *pattern* in *text*
- So can stop and return position…
- Until no match…

But what if they don't match?
Need a different pattern

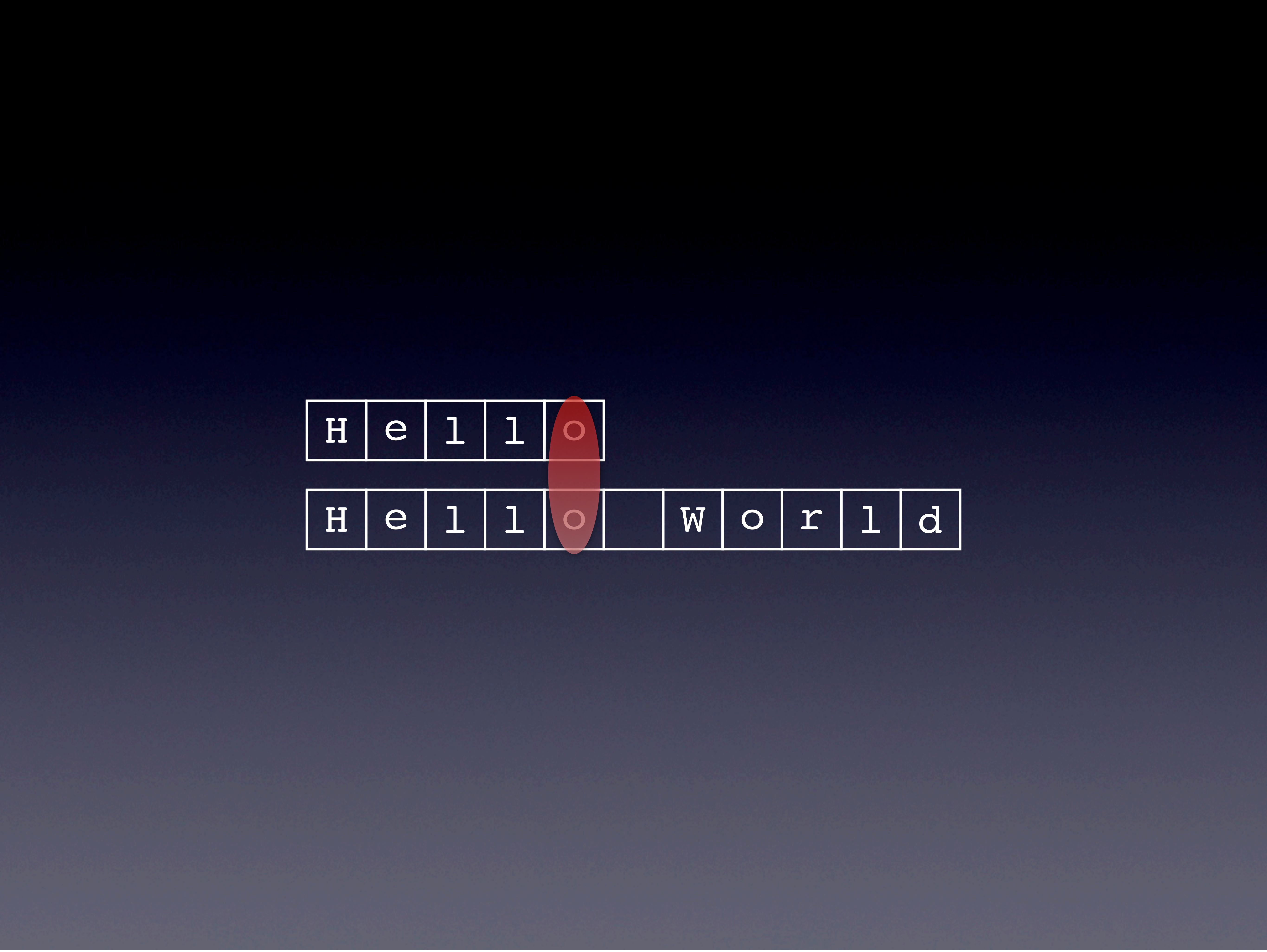


But what if they don’t match?
Need a different pattern

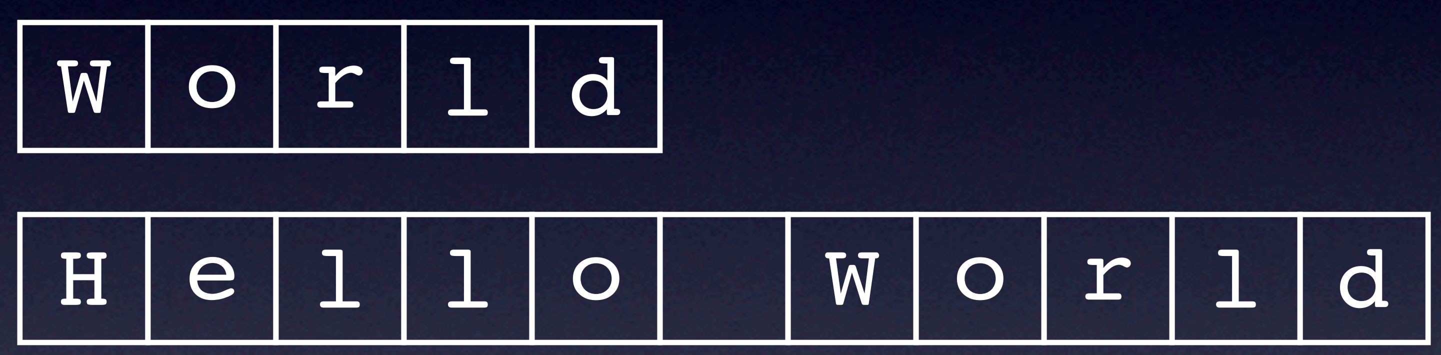
But what if they don't match?
Need a different pattern

But what if they don’t match?
Need a different pattern

But what if they don’t match?
Need a different pattern



But what if they don't match?
Need a different pattern


W o r l d
H e l l o   W o r l d


W o r l d
H e l l o   W o r l d

If they don’t match

- Slide *pattern* one character along *text*
- Compare the first character of *pattern* with corresponding character in *text*
- If they match…
- If they don’t match
- Until…

So character 0 of pattern aligns with character 1 of text

W	o	r	l	d

H	e	l	l	o		W	o	r	l	d

W	o	r	l	d

H	e	l	l	o		W	o	r	l	d


W o r l d
H e l l o W o r l d

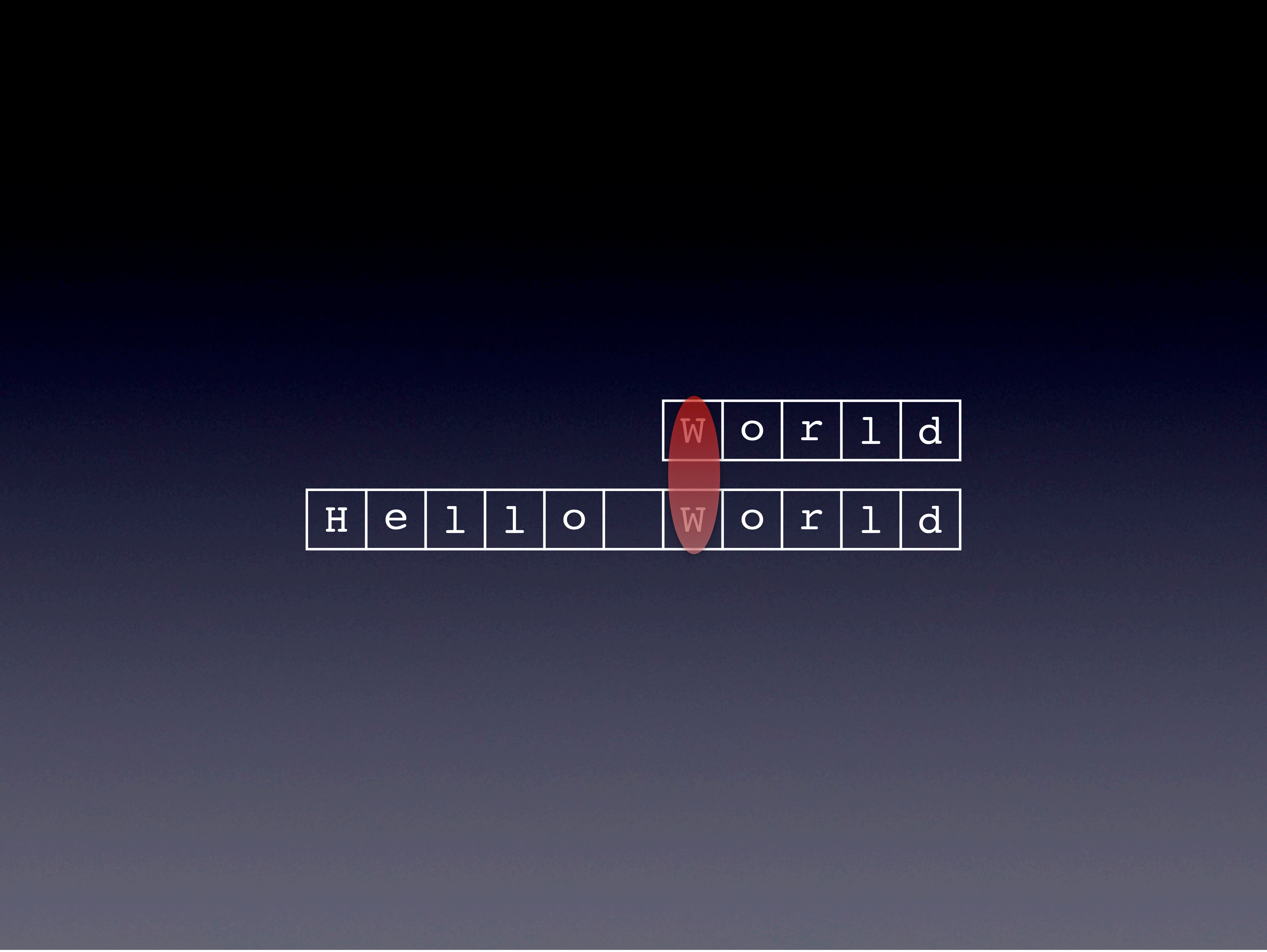
W	o	r	l	d

H	e	l	l	o		W	o	r	l	d

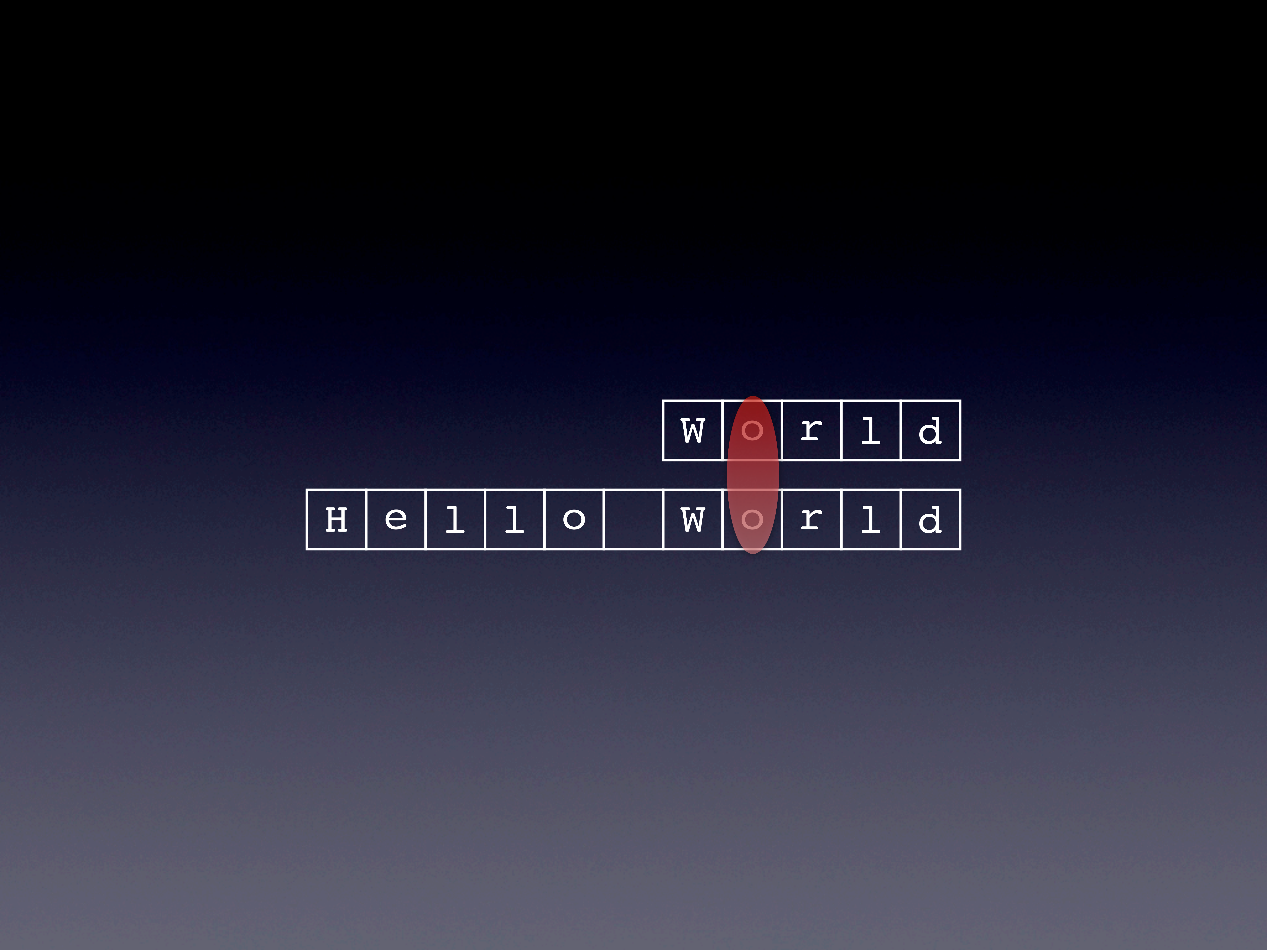

W o r l d
H e l l o W o r l d


W o r l d
H e l l o W o r l d

Finally we get a match
So can return true

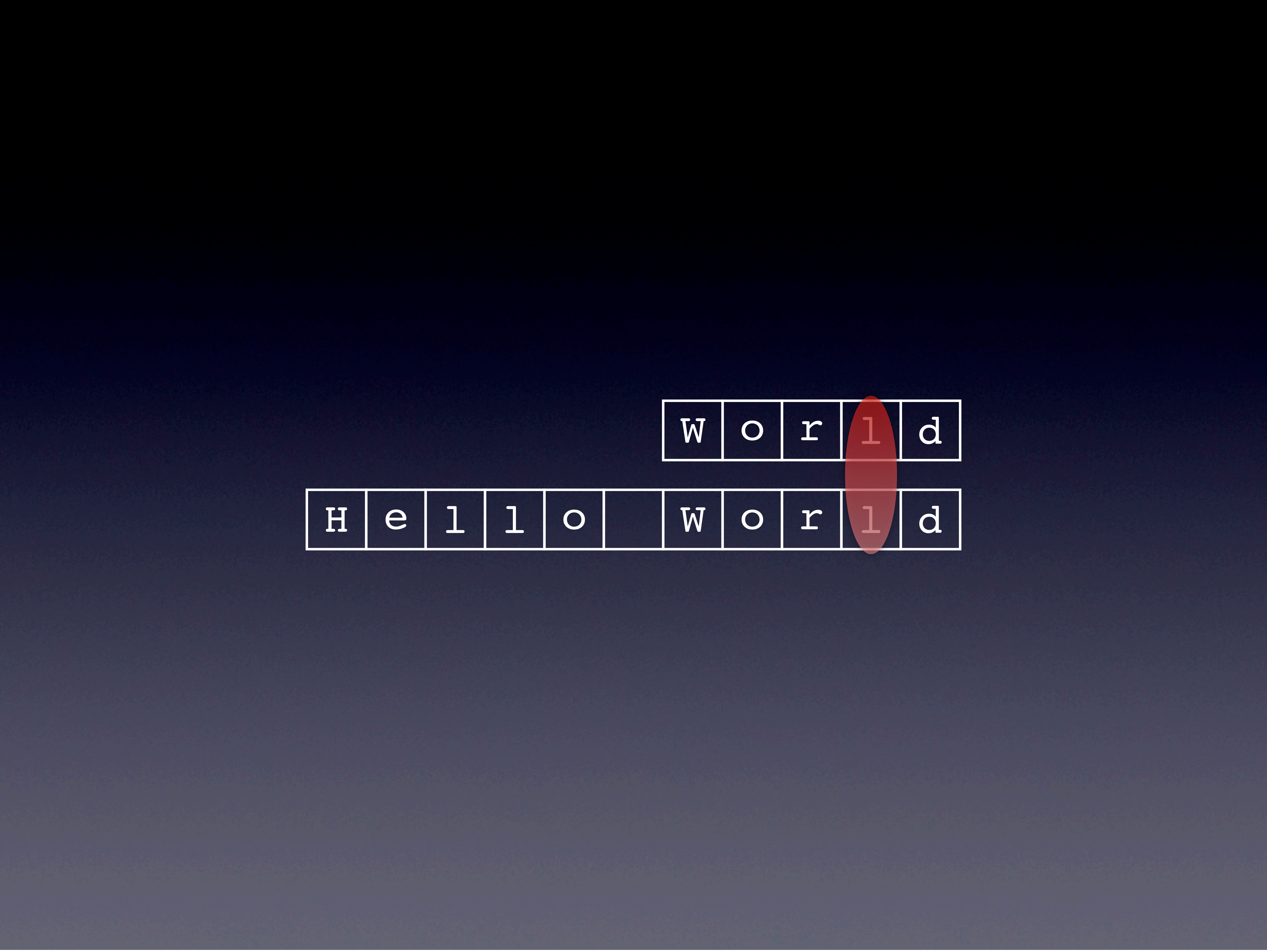


Finally we get a match
So can return true

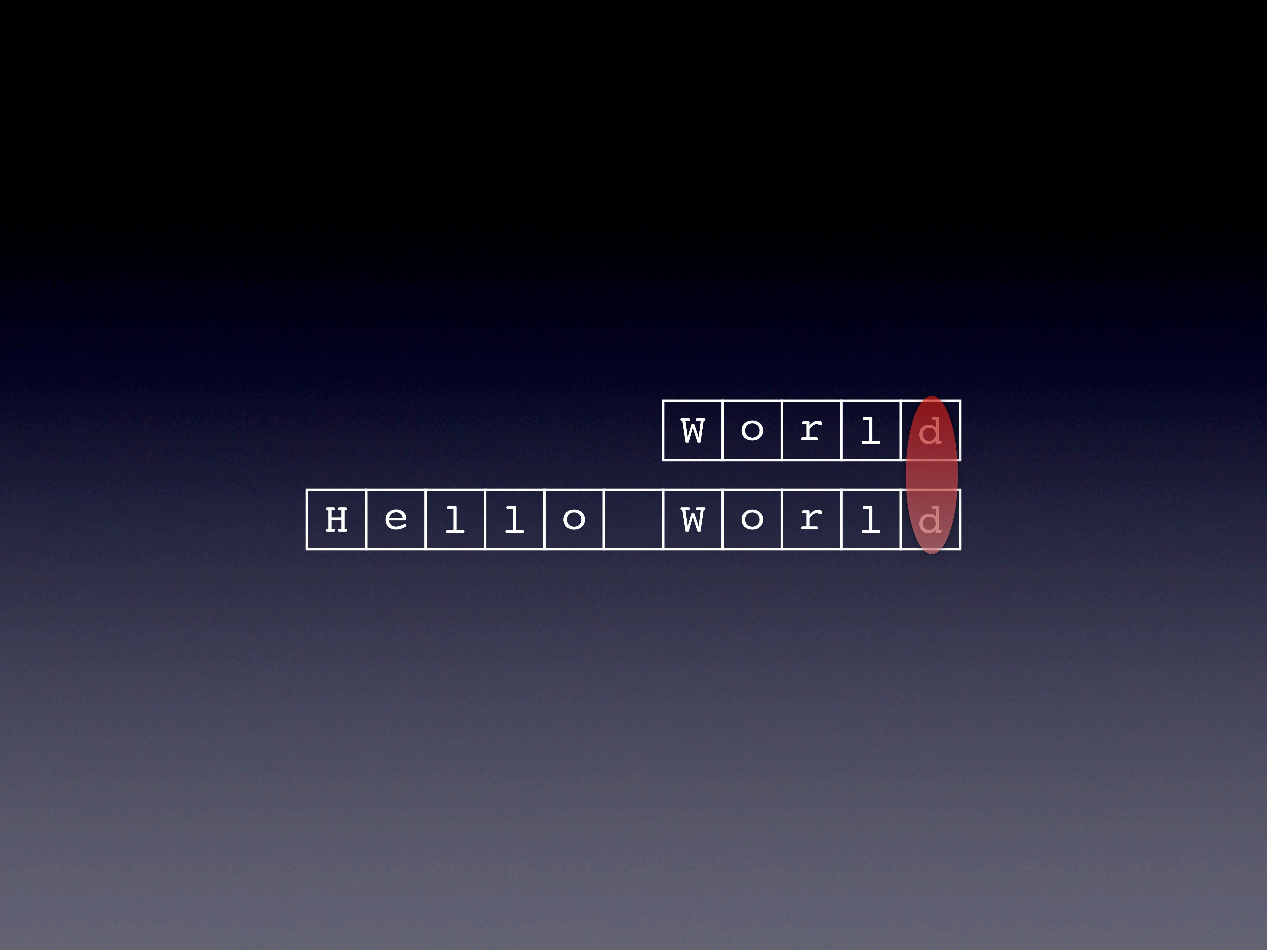


Finally we get a match
So can return true

Finally we get a match
So can return true



Finally we get a match
So can return true



Finally we get a match
So can return true

End Search

- End case is when it is no longer possible to slide *pattern* any further along *text*
- Will happen when the end of *pattern* aligns with the end of *text*
- In this case, we've not found *pattern* so we can report failure

Not matched and have reached the end of text
So string not in there

Naive String Search

- Algorithm works
- Equivalent found in C and Java libraries
- Problem is its slow
- Worst case
  - Has to compare every letter of *pattern*
  - In every alignment with *text*

strstr/String.matches/String.indexOf(String s)

Naive String Search

- Worst case has $O(nm)$ complexity
- Think about trying to find `aab` in `aaaaaaab`
- Fortunately, with real-world *patterns* and *text* its usable
- But can we do any better?

n = length of text
m = length of pattern

Slow…

- The problem with this algorithm is that if a match fails
- It always slides the match on by one character
- If we could find a way to slide by more than one character we could reduce the number of comparisons

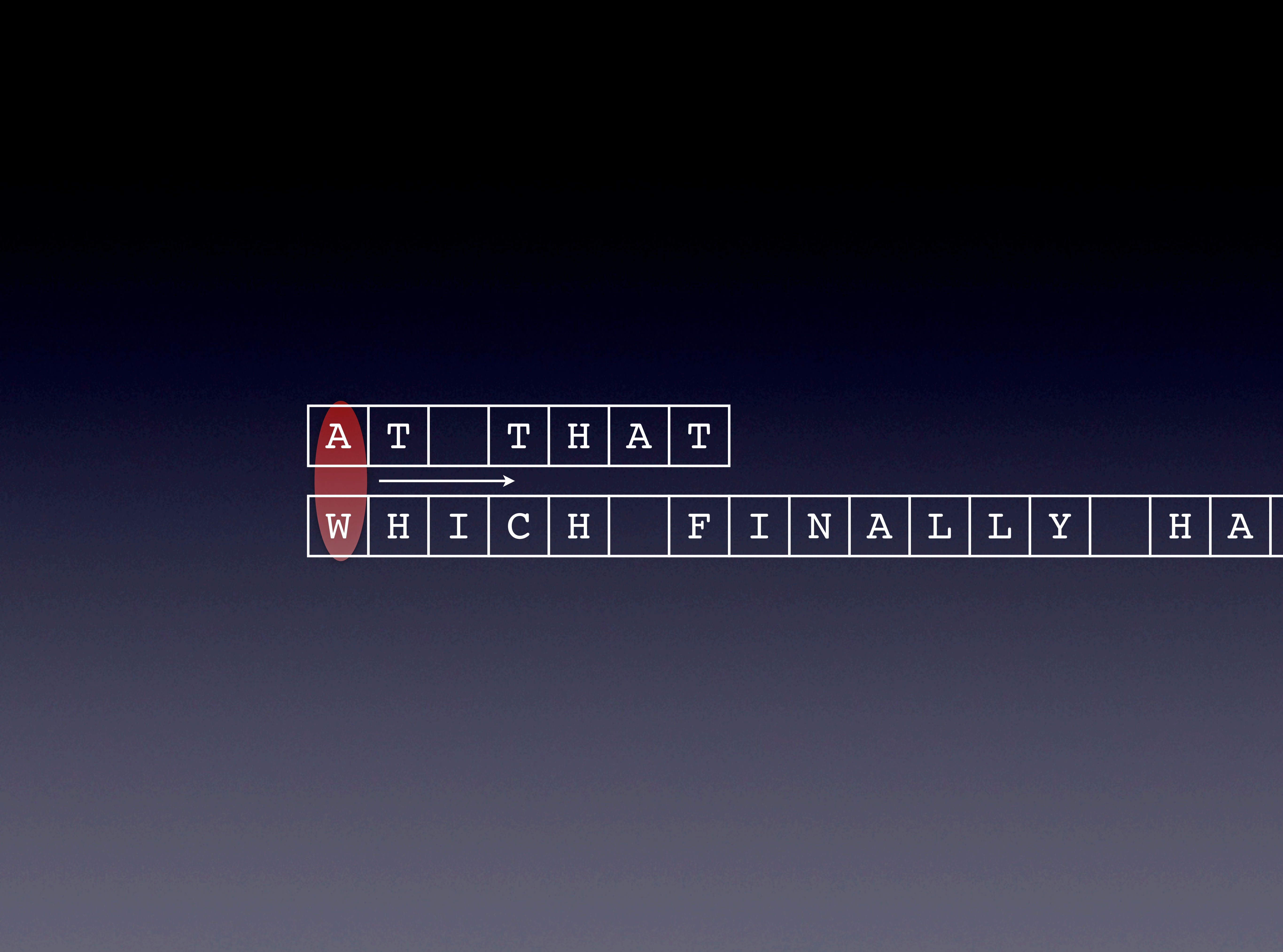
Sliding…

- Can’t just slide an arbitrary number of characters along
- Or we might skip the match…
- Can only skip characters when we know it is safe to do so…

Boyer-Moore

- One algorithm that does this is the *Boyer-Moore* Algorithm
- Sublinear algorithm
- Basic idea is that more information is obtained by scanning *pattern* from right to left than left to right

Developed in the 1970s by Robert Boyer and J. Strother Moore.
Read their paper -- the material in it is examinable!



Rather than scanning like this

Scanning like this helps us enormously…

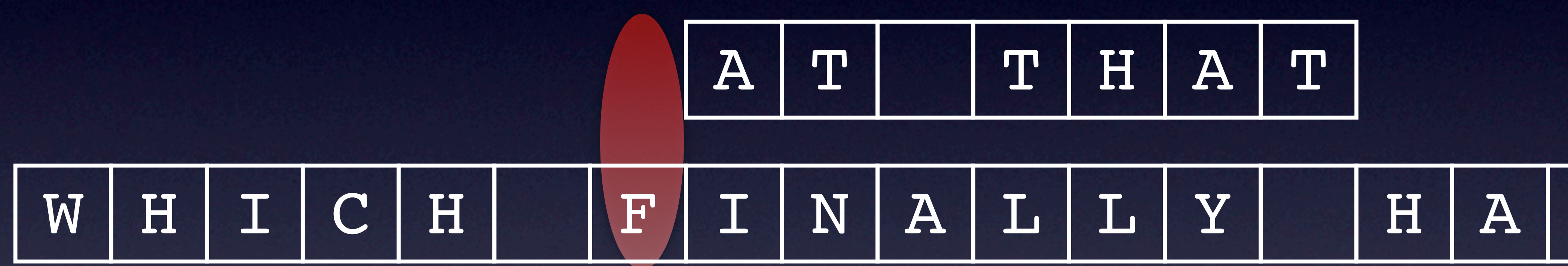
Observations

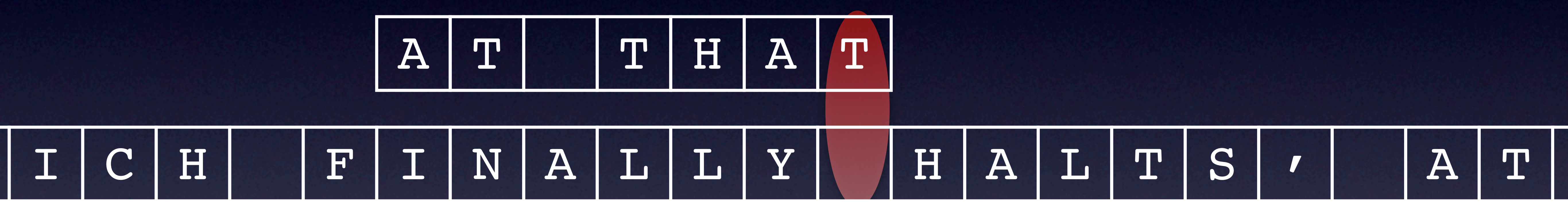
- Boyer-Moore made several observations about possible mismatchs
- These observations enable us to slide *pattern* ahead more than one character at a time

Observation One

- If the mismatching *char* in *text*, does not occur in *pattern*:
- Then we know there's no possibility of *pattern* matching at `0`, `1`, `2`, `...`, `length(pattern)`
- Since this would require the character to be part of pattern
- Can slide *pattern* down `length(pattern)` chars

AT THAT
WHICH FINALLY HA


AT THAT
WHICH FINALLY HA


AT THAT
ICH FINALLY HALTS, AT

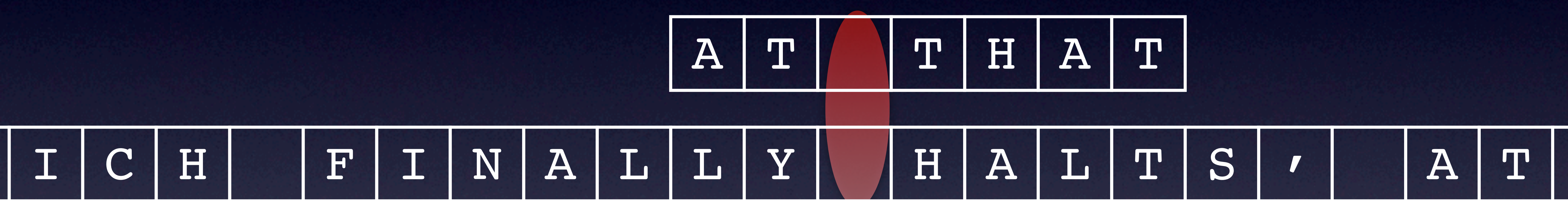
Observation Two

- More generally, even if *char* does occur in *pattern*
- We can still slide *pattern* so that the *char* aligns with the rightmost occurrence of *char* in pattern
- If we slide it any less, then it still won't be a match…

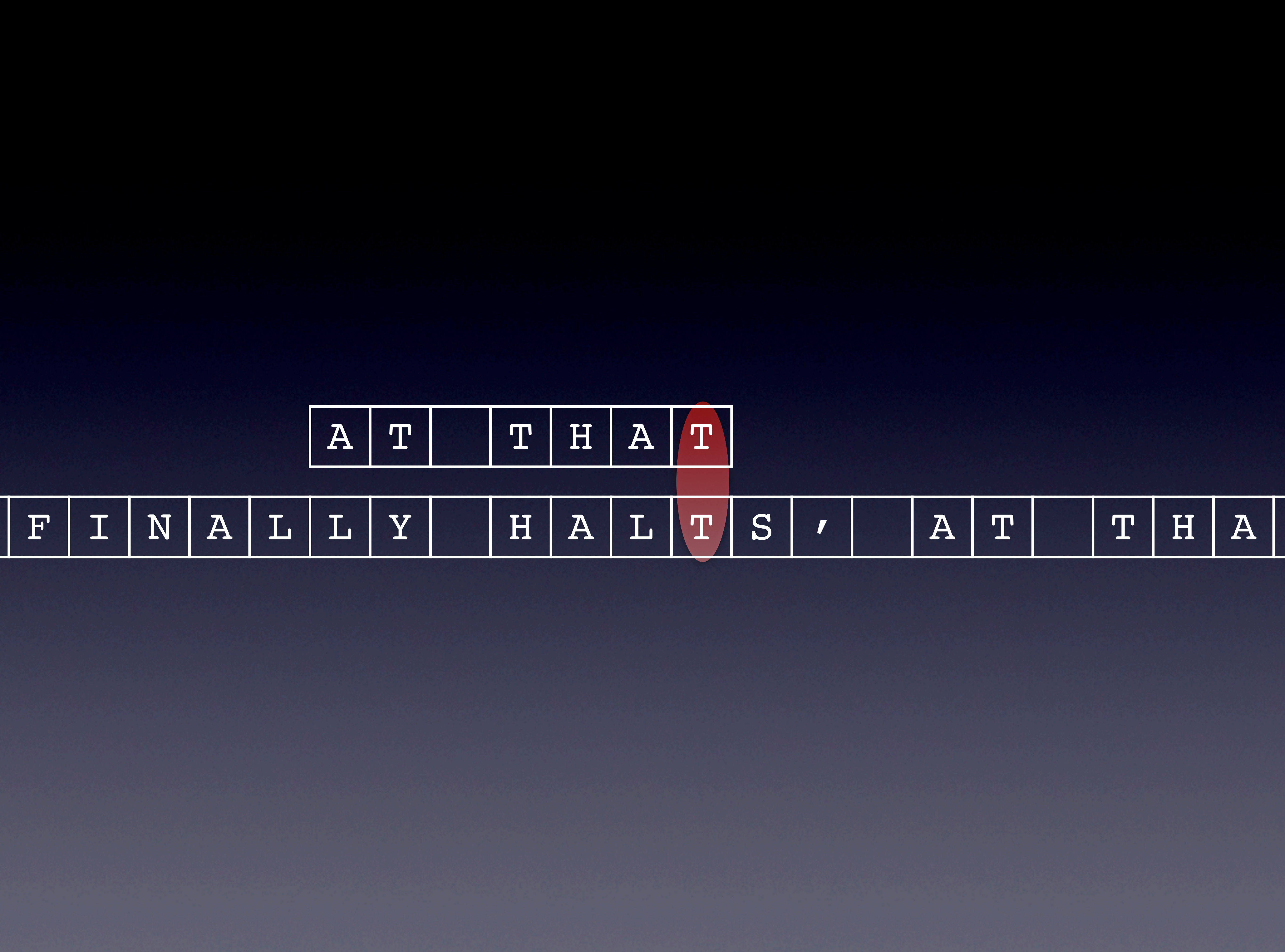
char is the mismatching char in text

AT THAT

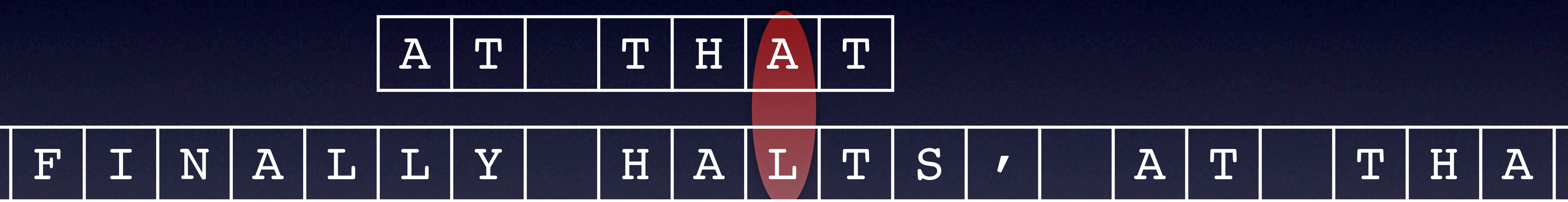
ICH FINALLY HALTS', AT



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

Observation 3a

- Third observation they made takes place when a character is matched
- Continue backing up until we match all of *pattern* — and so have found it
- Or a mismatch occurs after matching *m* characters…

Observation 3a

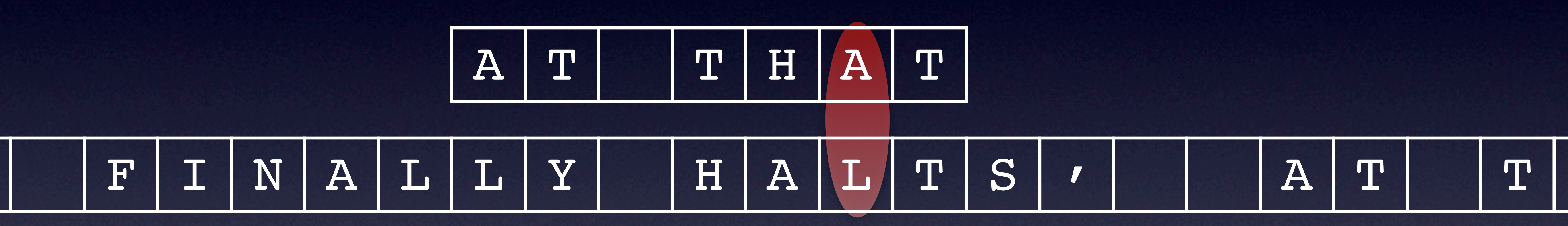
- Using the same reasoning as before, we can obtain a value *k* to slide *pattern*
- If the right-most char is to the right of the mismatch, then we'd have to slide the pattern backwards to align it
- This is worthless, so…
- In this case, *k* = 1

k is based on the rightmost occurence of char in pattern as before

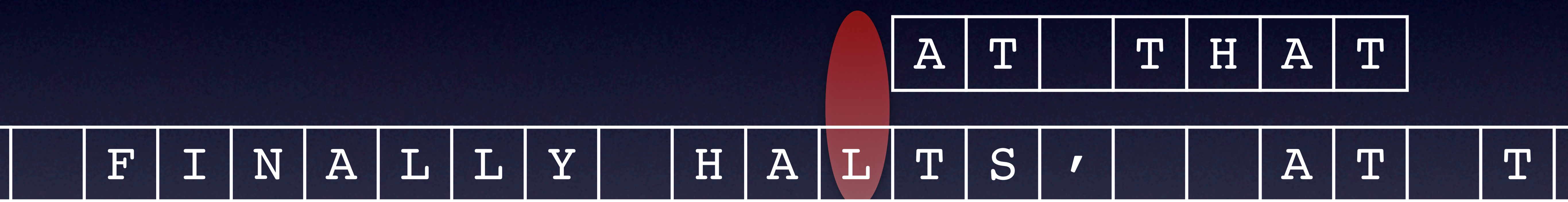
Observation 3a

- On the other hand, if it is to the left of the mismatch, then $k = delta_1 - m$
- In either case, we can slide pattern down *k* characters
- And continue from the end of pattern again

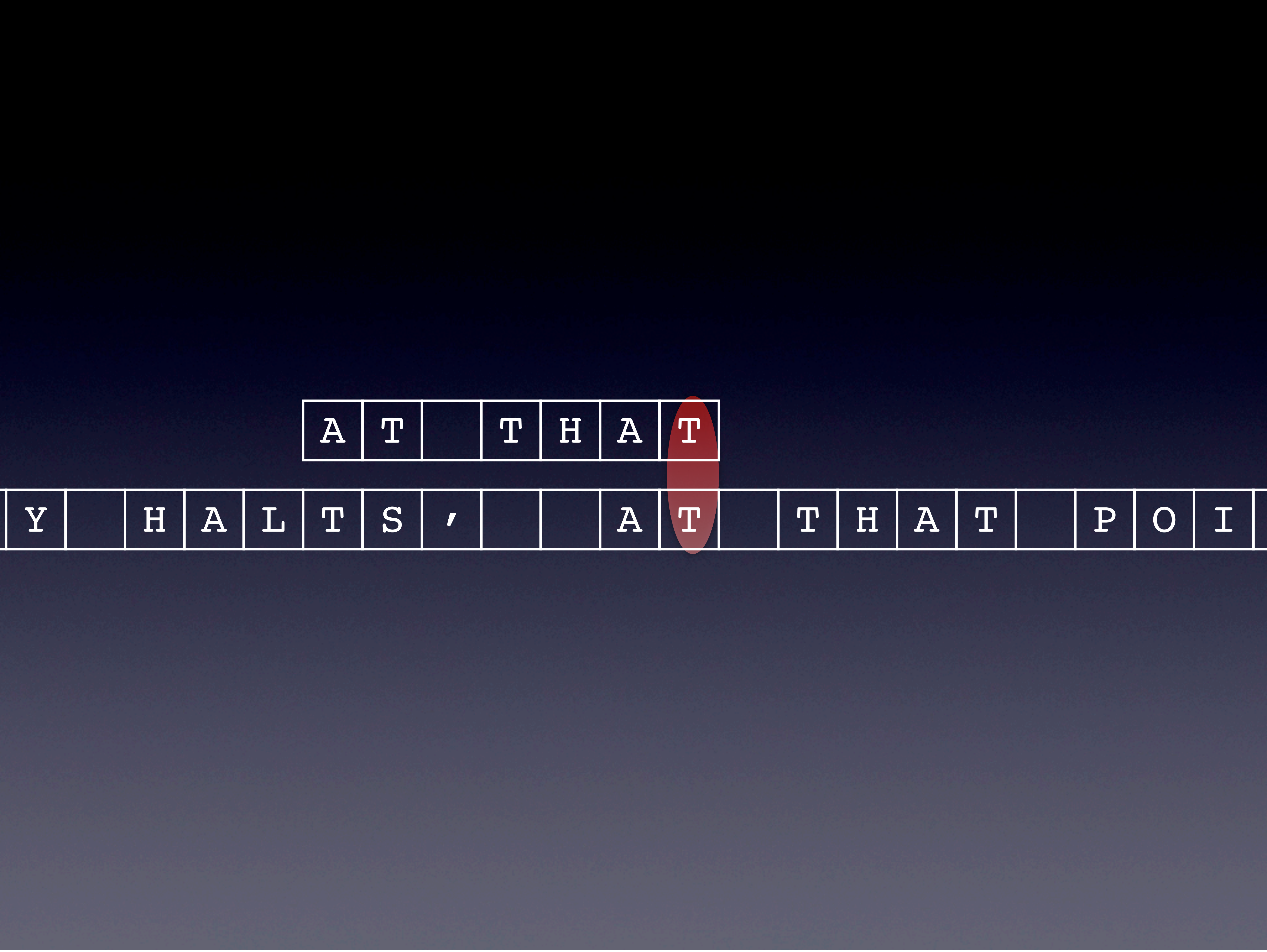
delta is the distance from the end of pattern of the rightmost occurence
m is the number of characters matched



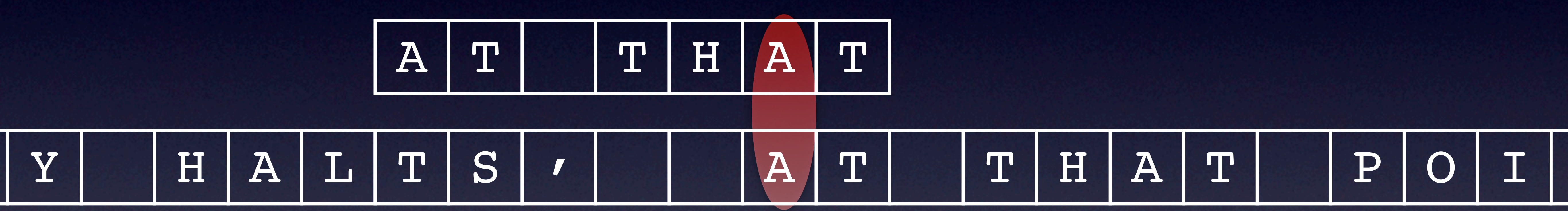
Okay, found a character that matches, step back to test the previous character



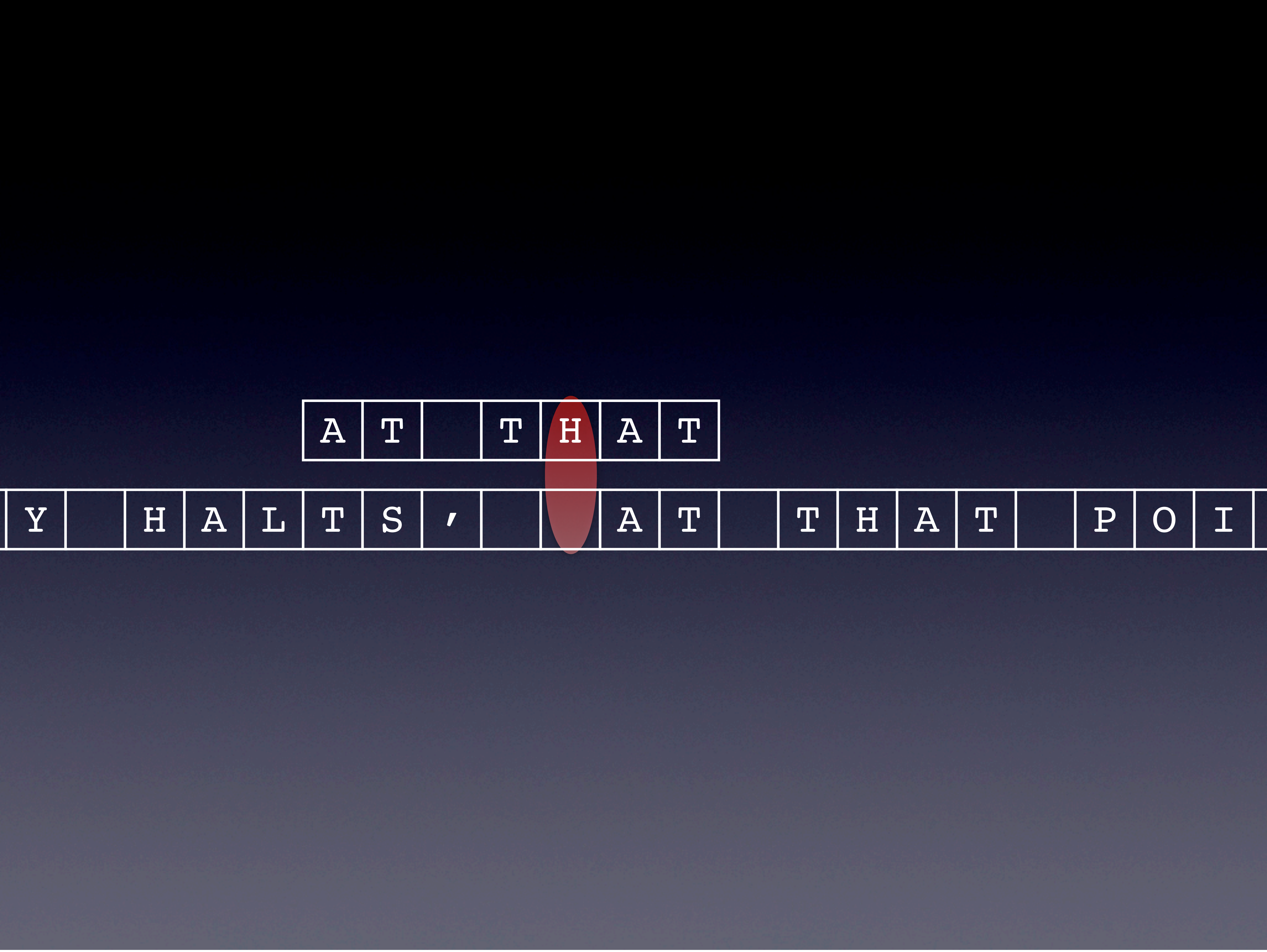
Okay, found a character that matches, step back to test the previous character



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

Observation 3b

- But we can do better than that…
- We know that the next *m* characters of *text* match the final *m* characters of *pattern*
- Call this *subpat*
- Also know that this occurrence of *subpat* is preceded by *char*

Observation 3b

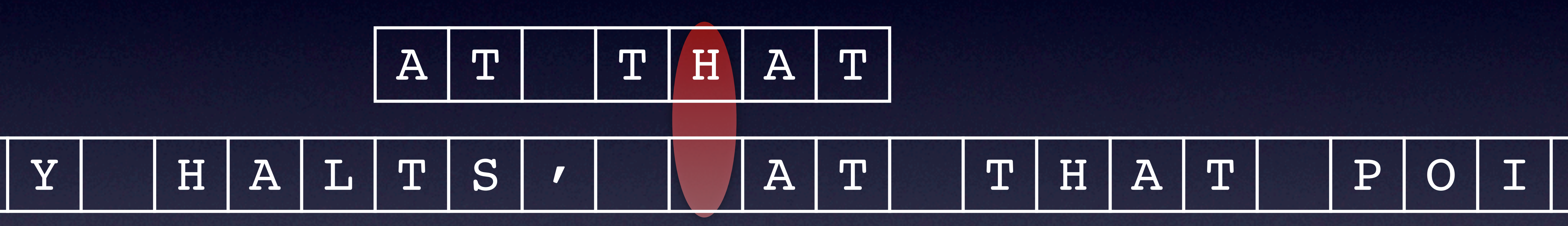
- Roughly speaking…
- Slide *pattern* down some so the discovered *subpat* is aligned by the rightmost occurrence of *subpat* in *pattern* not preceded by *char*
- Must allow the right most plausible reoccurrence of *sub*pat to fall of the left end of *pattern*

Observation 3b

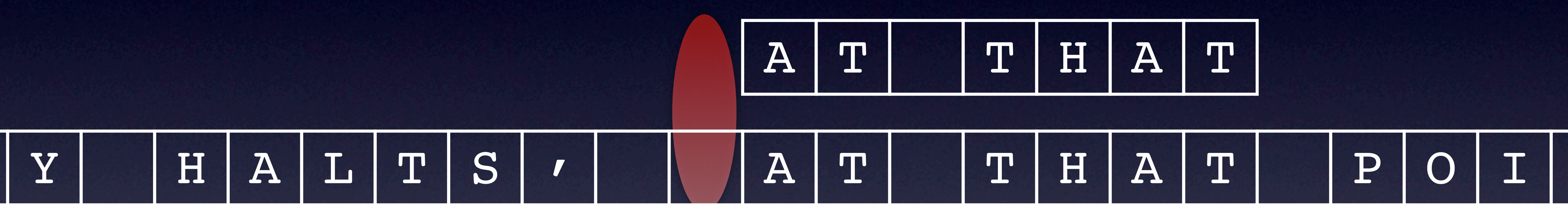
- Define a function *$delta_2(j)$* that gives the right-most occurrence of *subpat* (between *j* and the end of pattern)
- That is not preceded with the character at *j*

Observation 3

- In the case, where we have matched *m* characters we want to slide either
  - 1 character
  - $delta_1$ characters
  - $delta_2(j)$ characters
- Just chose the maximum of the three…



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

Observations

- These observations massively reduced the number of comparisons we do
- In this example, we only make 14 references to *text*
- Seven of which were verifying the final match…

Preprocessing

- However, this doesn't take into account how to find *delta*$_1$ or *delta*$_2$
- Searching for these each time would slow the program down
- Can do this by preprocessing *pattern*
- And building up two lookup tables

$Delta_1$ Lookup Table

- The first LUT maps characters to amount to slide *pattern*
- Need one entry for each possible character
- 256 entries for 8-bit chars

$Delta_1$ Lookup Table

- If *char* is in pattern
  - `length(pattern) - j`, where `j` is the index of the right-most occurrence of *char*
- Else
  - `length(pattern)`

$Delta_2$ Lookup Table

- This table maps integer positions in *pattern* to the distance we can slide *pattern* from Observation 3b
- There will be `length(pattern)` entries
- Not as easy to consturct as $delta_1$

Boyer-Moore algorithm

```
       stringlen = length of string
       i = patlen - 1

top:   if(i > stringlen) return false
       j = patlen - 1

loop:  if(j == -1) return i + 1
       if(string[j] == pat[j])
       {
            j = j - 1
            i = i - 1
            goto loop
       }
       i = i + max(delta1[string[i]], delta2[j])
       goto topl
```

In pseudo code
Dreaded goto…

Internationalization

- This algorithm works fine for Unicode too
- But the size of `delta1` will grow
- For 16-bit, characters would be `65536` entries, instead of `256`
- Interestingly, you can run it as 8-bits on UTF-8 strings because they are self-syncing

Boyer-Moore

- Gold standard in string search
- Everything else is compared to it
- But there are alternatives

BWT and Suffix Arrays

- Another approach makes use of the Burrows-Wheeler Transform
- Easier to understand by consider a related approach Suffix Arrays first

Suffix Array

- If we take *text*, we can build an array of suffixes from it
- Suffix is a substring of *text* from *i..length(text)*
- Where *0* <= *i* < *length(text)*
- This will give us an array of *length(text)* suffixes

0	ORANGE
1	RANGE
2	ANGE
3	NGE
4	GE
5	E

All possible suffixes of orange

Suffix Array

- What's the point of this?
- It's effectively the same as we were doing in our naive search algorithm
- If *pattern* is in the string, then at least one of the arrays will start with *pattern*
- If we sort the suffix array, then we can use a binary search to find the *pattern*

0	ORANGE
1	RANGE
2	ANGE
3	NGE
4	GE
5	E

All possible suffixes of orange
Suppose we are looking for the string GE in ORANGE
suffix 4 begins GE

2	ANGE
5	E
4	GE
3	NGE
0	ORANGE
1	RANGE

All possible suffixes of orange
Suppose we are looking for the string GE in ORANGE
suffix 4 begins GE

Suffix Arrays

- Binary search on a suffix array will find the string in `O(m log n)` time
- But requires us to build up the array of suffixes
- However, there's a relationship between the Burrows-Wheeler Transform and Suffix arrays

m = length of pattern
n = length of text
Think about the string we looked at before