

Advanced Algorithms:
Text Compression Algorithms

Tetsuo Shibuya

Human Genome Center, Institute of Medical Science
(Adjunct at Department of Computer Science)

University of Tokyo

<http://www.hgc.jp/~tshibuya>

- Various lossless compression algorithms for texts
 - ◆ Statistical methods
 - ▶ Huffman code
 - ▶ Tunstall
 - ▶ Golomb
 - ▶ Arithmetic code
 - ▶ PPM
 - ◆ Dictionary-based methods
 - ▶ LZ77, LZ78, LZW
 - ▶ Block sorting
 - ◆ Compression of suffix arrays

□ Two types of compression algorithms

- ◆ Lossless compression – this lecture
- ◆ *cf.* Lossy compression
 - ▶ Especially for images, audio, etc

□ Why compressible?

- ◆ Due to the 'redundancies' in information
- ◆ 'Predictable' information is somewhat redundancy
 - ▶ We do not have to remember things that can be predicted with 100% confidence



Information and Entropy

Property of information

- ◆ Non-negative
- ◆ Larger if the probability is smaller
- ◆ Information values should be addable

▶ $I(p \cdot q) = I(p) + I(q)$ (in case the two events are independent to each other)

Japan wins the world cup: 0.1%
Spain wins the world cup: 15%
Which is surprise?

Information

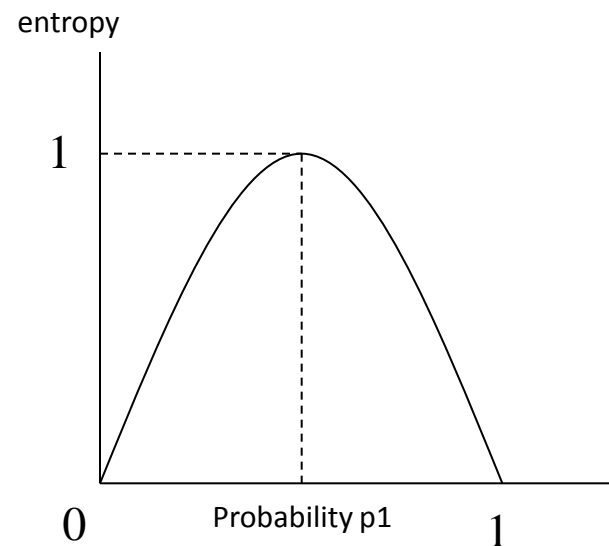
- ◆ $I(p) = -\log_2(p)$
- ◆ Unit: bit
- ▶ Probability 0.5 \rightarrow 1 bit

Entropy

- ◆ Expected information of some model
- ▶ $H(p_1, p_2, p_3, \dots, p_n) = -\sum p_i \log p_i$
- ◆ Property
- ▶ $H=0$ if $p_i=1$
- ▶ H is largest when $p_i=1/n$

Entropy = Lower bound of the compression ratio

- ◆ Impossible to estimate if we do not know the 'real' model



Entropy for 0/1 source

■ Statistical methods

- ◆ Based on the statistics of characters/words in the target text
 - ▶ Huffman code
 - ▶ Tunstall
 - ▶ Golomb
 - ▶ Arithmetic code
 - ▶ PPM
 - ▶ ...

■ Dictionary-based methods

- ◆ Based on some frequent word dictionary
 - ▶ LZ77, LZ78, LZW
 - ▶ Block sorting
 - ▶ Sequitur
 - ▶ ...

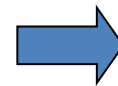
Huffman Code (1)

■ Idea

- ◆ Smaller code for more frequent characters

Model	
A	12.5%
C	12.5%
G	25%
T	50%

Code
100
101
11
0



Expected to be 1.75n bit!
(It's ideal!)

Alphabet size = 4
 $\log(\text{Alphabet size}) = 2\text{bit}$
entropy = 1.75bit

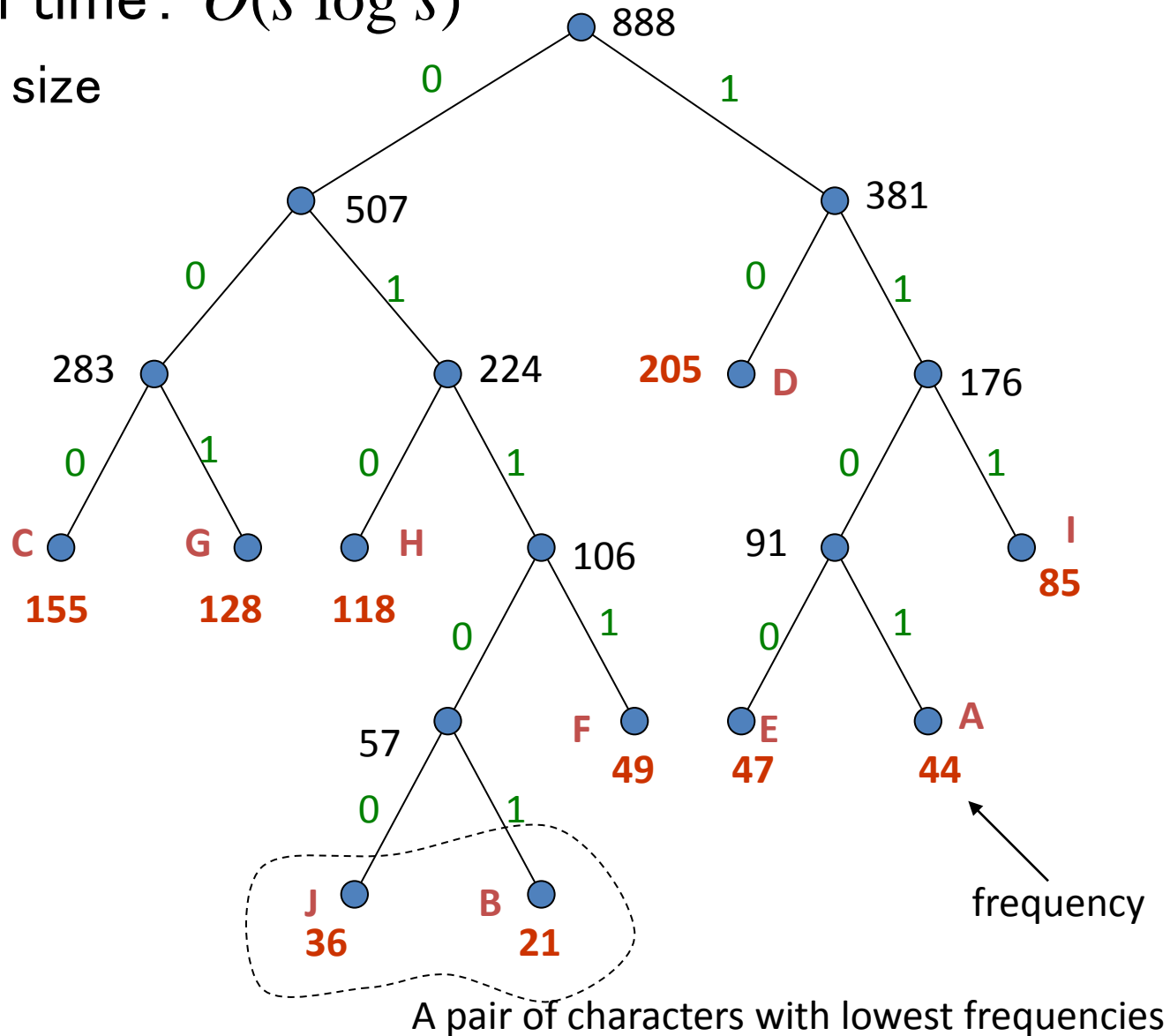
Ideal Case

Huffman Code (2)

Computation time: $O(s \log s)$

◆ s : Alphabet size

◆ Use heaps



▣ Properties

◆ Prefix code (Instantaneous code)

▶ A code is not a prefix of other code

◆ Optimal algorithm for coding characters with 0/1

◆ Entropy:

▶ $H(X) \leq L < H(X) + 1$

▣ Extension

◆ Consider a set of consecutive k characters as ONE character

▶ $H(X) \leq L < H(X) + 1/k$

AAAA 0.47%

AAAC 0.37%

AAAG 0.35%

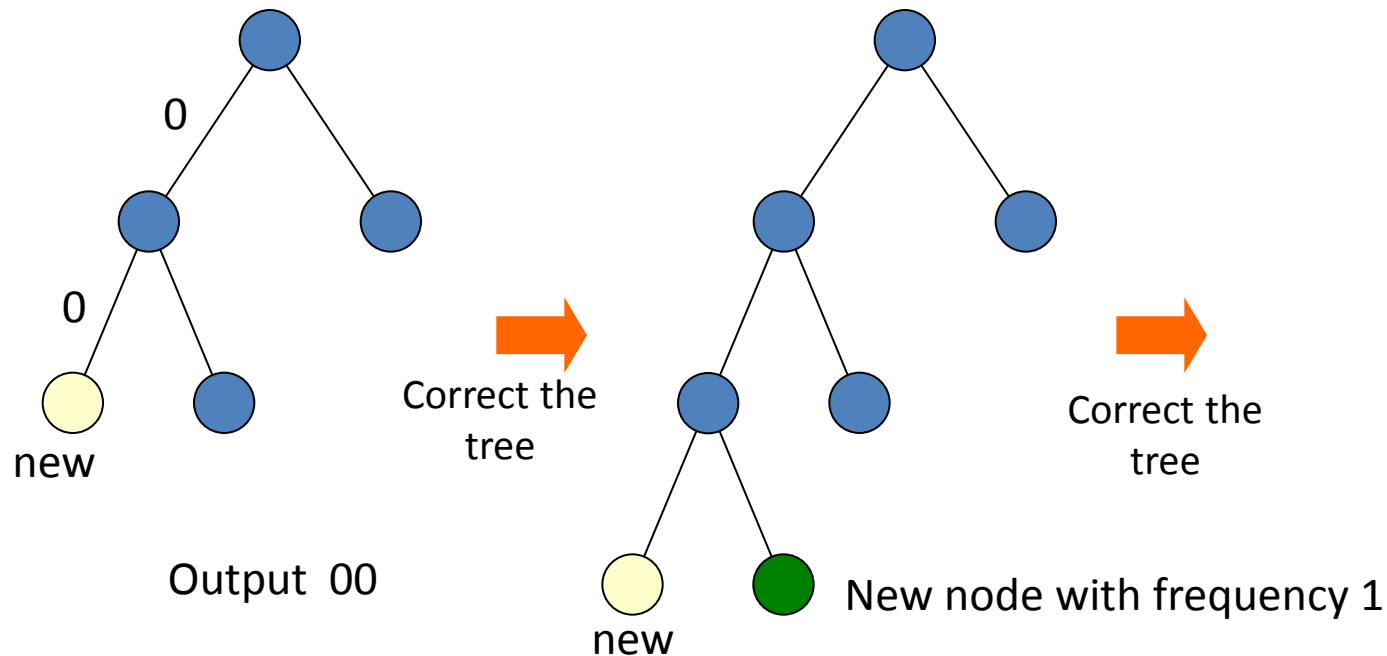
AAAT 0.41%

AACA 0.32%

...

Adaptive Coding (Feller-Gallager-Knuth)

- We must check all the frequencies of all the characters before Huffman coding
- Adaptive coding
 - ◆ Update the frequency table while coding
 - ◆ 0-leaf for the character that has never appeared yet
 - ◆ It is possible to decode online, too



■ A variation of Huffman coding

- ◆ Faster tree construction

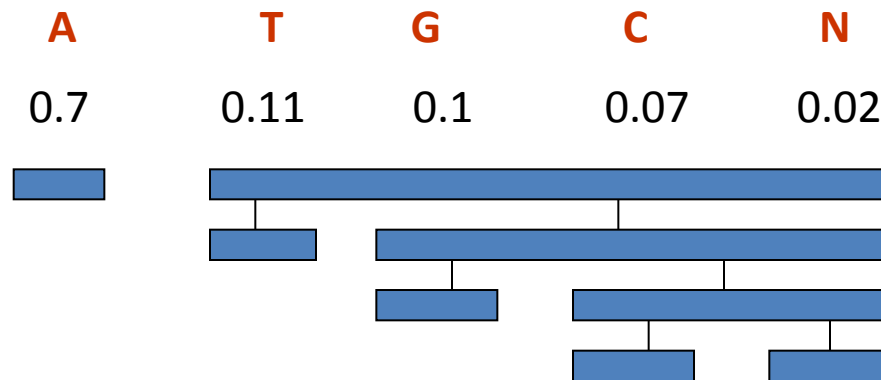
- ◆ Algorithm

 - ▶ Sort the frequency

 - ▶ Divide into two parts (which should be as equal sizes as possible)

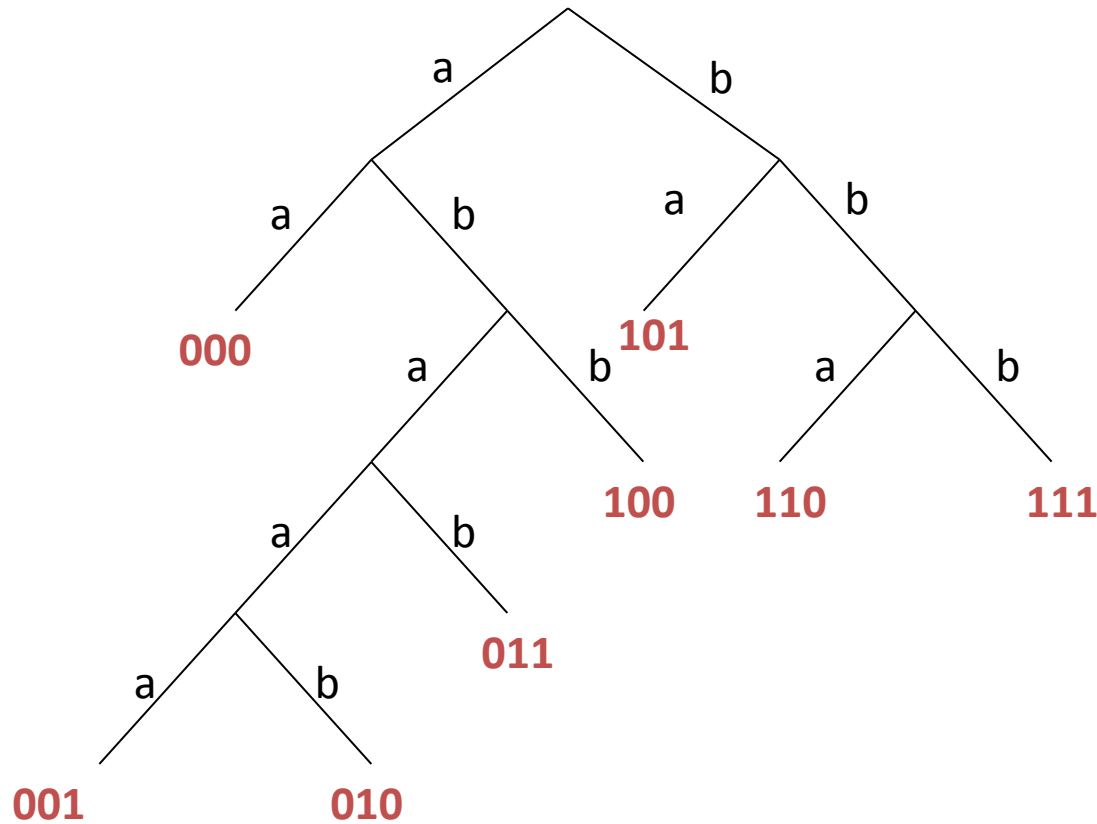
- ◆ $H(X) \leq L < H(X) + 1$

 - ▶ But no better than the Huffman coding



Tunstall Code

- Oppositely, assign a fixed-length code for different-length words



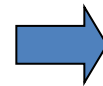
And use the Huffman against the coded strings!

Run-length coding

0101000110000010100000000100100010001
1 1 3 0 5 1 8 2 3 3

Golomb coding

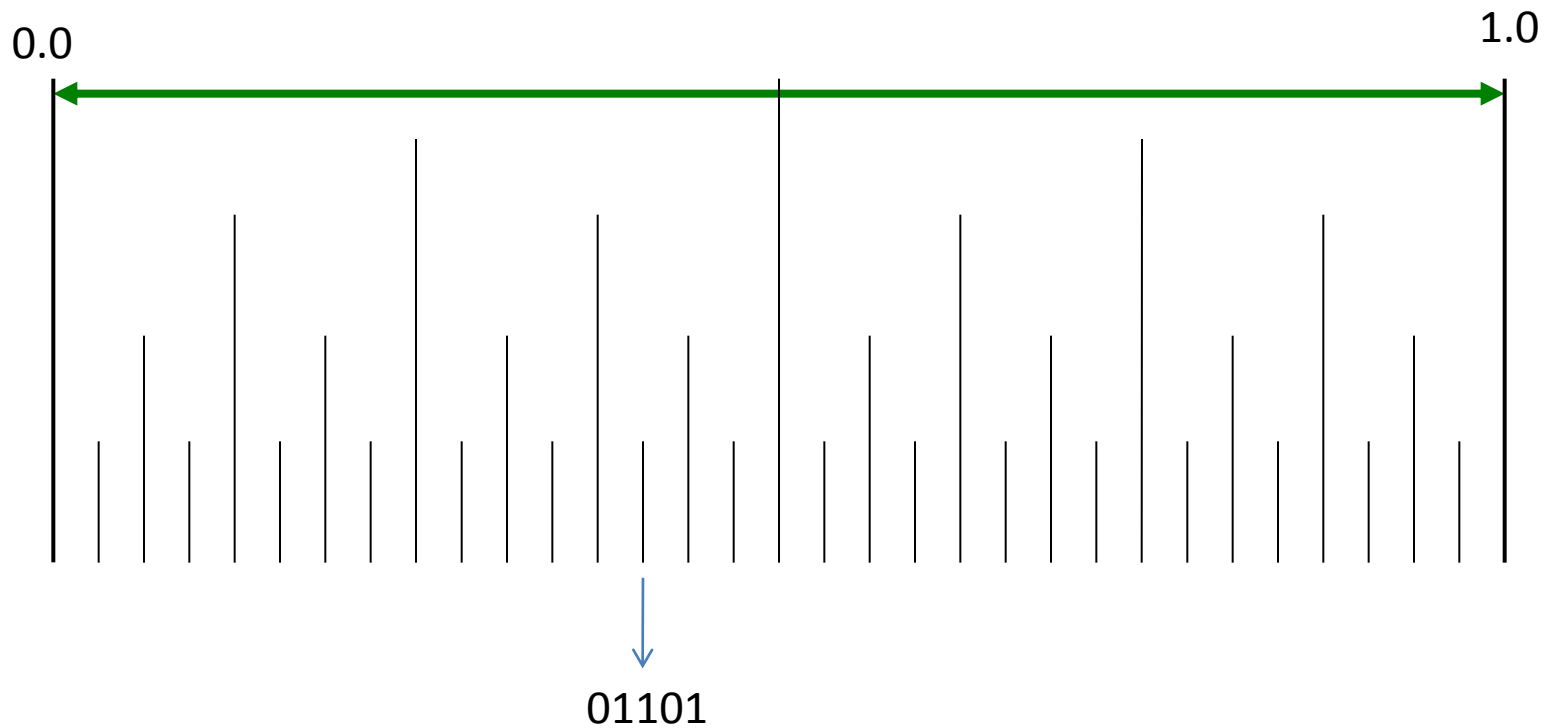
Text	Frequency
00000	121
0001	45
001	21
01	15
1	10



Huffman Coding

Arithmetic Coding (1)

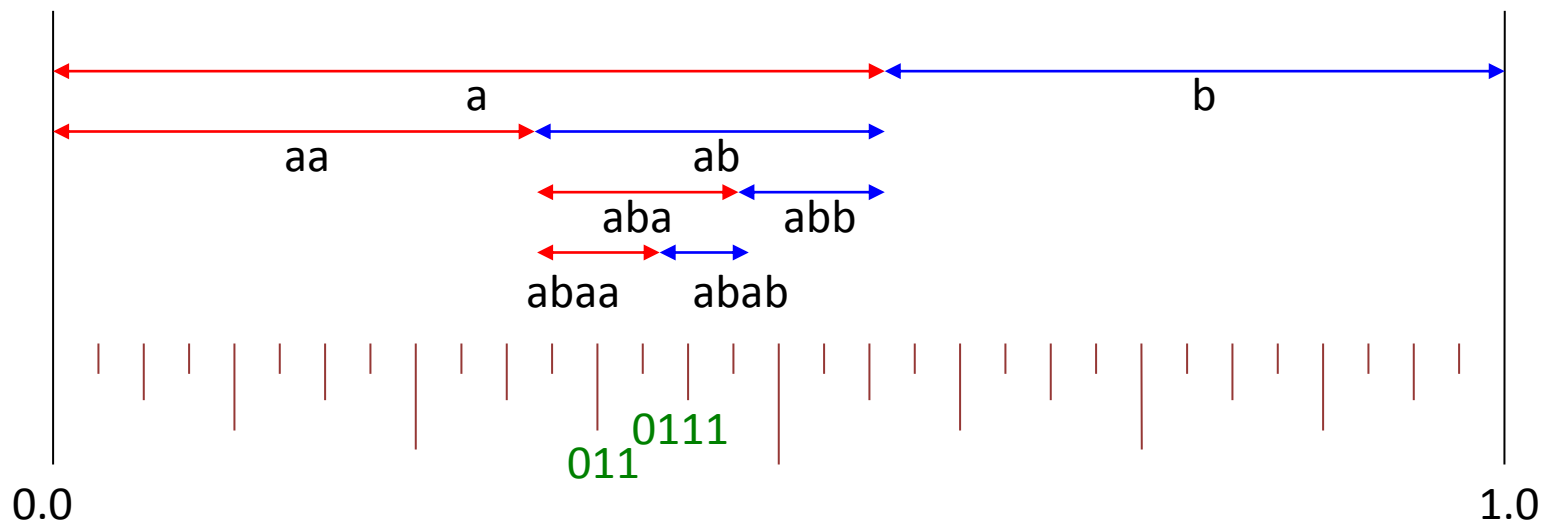
- Any $[0,1)$ real numbers can be represented by 0-1 sequence



Decoding Arithmetic Codes

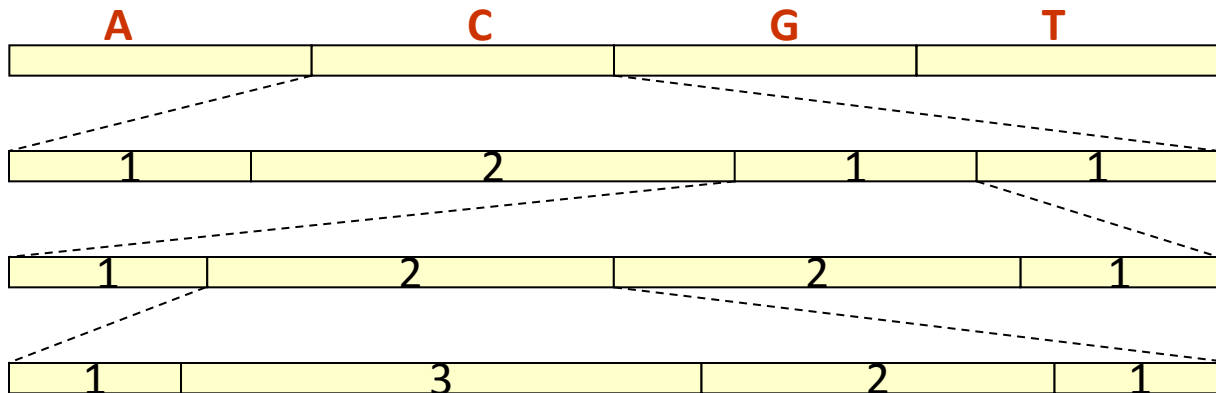
Three steps

- ◆ Comparison of the real number with thresholds
- ◆ Output a character
- ◆ Scaling the target region to $[0.1)$



Adaptive Arithmetic Coding

- Equal probabilities for all the characters at first
- Update it according to the next character
 - Same for the decoding algorithm



PPM (Prediction by Partial Matching)

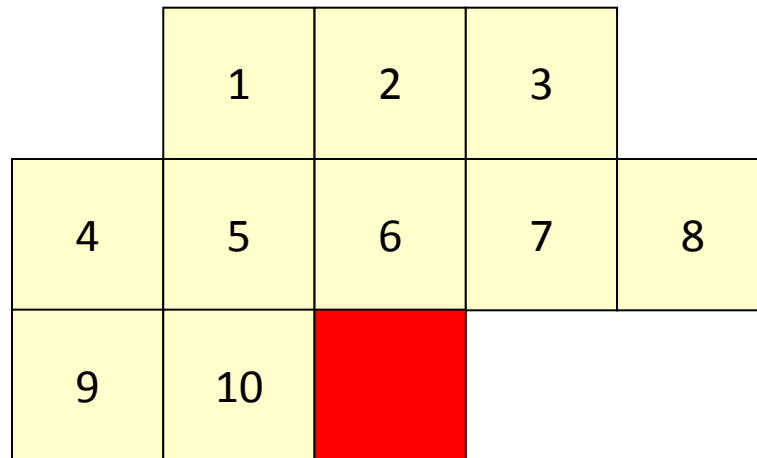
- Predict the next character probabilities based on the preceding characters
 - ◆ Output 'ESC' and use frequency table on shorter words if a new word appears
 - ▶ Let the frequency of ESC be the number of kinds of characters that have already appeared
 - ◆ Same in the decoding algorithm

Previous Word	Frequencies of the next	
⋮	⋮	
ATGCG	A: 2 C: 1	← ESC: 2
ATGCT	A: 7 C: 4 G: 21 T: 3	← ESC: 4
ATGGA	G: 2	← ESC: 1
⋮	⋮	

Frequency Table

JBIG1: Binary image compression

- Predict from 10 pixels before the pixel



 Arithmetic Coding

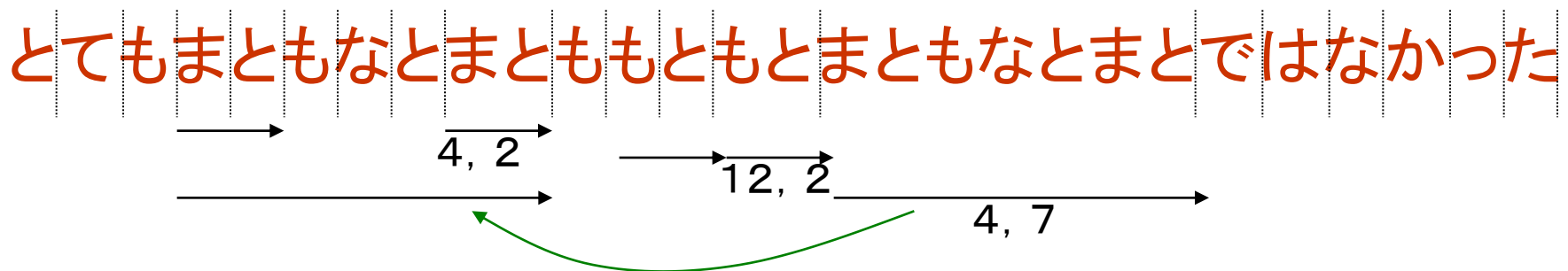
Dictionary-based compression

- ◆ Coding by previously appearing words

 - ▶ The position and the length

Computation

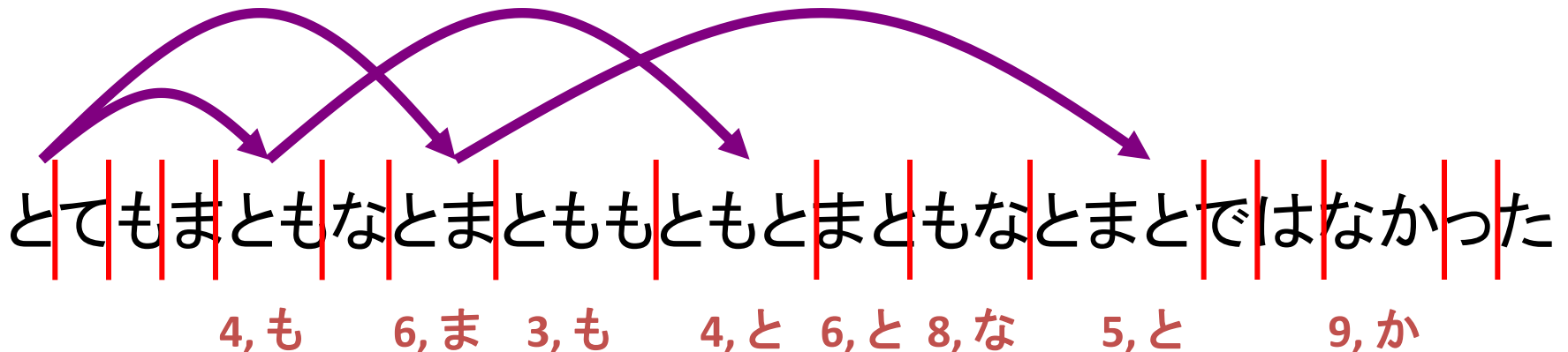
- ◆ Ideally, use suffix trees/arrays



+Huffman coding → LZH, zip, PNG

+Shanon-Fano coding → gzip

- Distance to the Position of the previous coded string + 1 character



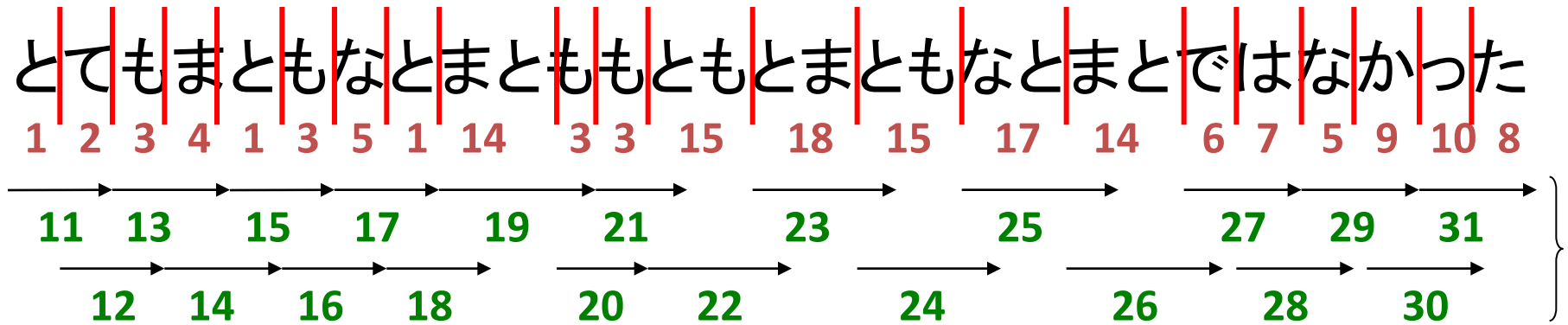
■ An implicit word table

- ◆ Initially we have just an alphabet table
- ◆ In each step
 - ▶ Find the longest word in the (implicit) table and code with it
 - ▶ Add a new code for 'the found word + next character' to the table
 - Implicitly using the position, i.e., we do not have to remember the table!

■ Very fast, but not so high compression rate

Alphabet table

1:と 2:て 3:も 4:ま 5:な 6:で 7:は 8:た 9:か 10:っ



Implicit table

Used in gif / tiff

■ Four steps

- ◆ Divide the text into blocks, and do the following for each block
 - ▶ So that we can decode faster
- ◆ BWT (Burrows–Wheeler Transform)
 - ▶ Related to the suffix arrays
- ◆ MTF (move to front) coding
- ◆ Huffman coding or arithmetic coding

cf.

```
0: mississippi$
1: ississippi$
2: ssissippi$
3: sissippi$
4: issippi$
5: sippi$
6: sippi$
7: ippi$
8: ppi$
9: pi$
10: i$
```



Sort

```
10: i$
7: ippi$
4: issippi$
1: ississippi$
0: mississippi$
9: pi$
8: ppi$
6: sippi$
3: sissippi$
5: ssippi$
2: ssissippi$
```

Suffix Array

BWT (Burrows-Wheeler Transform)

■ Suffix array $SA[0..n]$

◆ Consider cycled suffixes

▶ Same as the ordinary suffix array if we add '\$' in the end of the string

■ $BWT[i] = T[SA[i]-1]$

◆ The last characters of the sorted cycled suffixes


Cycled suffixes (with '\$')		BWT	SA
0: mississippi\$		10: i	11: \$mississippi
1: ississippi\$m		9: p	10: i\$mississipp
2: ssissippi\$mi		6: s	7: ippi\$mississ
3: sissippi\$mis		3: s	4: issippi\$miss
4: issippi\$miss		0: m	1: ississippi\$m
5: ssippi\$missi		11: \$	0: mississippi\$
6: sippi\$missis		8: p	9: pi\$mississip
7: ippi\$mississ		7: i	8: ppi\$mississi
8: ppi\$mississi		5: s	6: sippi\$missis
9: pi\$mississip		2: s	3: sissippi\$mis
10: i\$mississipp		4: i	5: ssippi\$missi
11: \$mississippi		1: s	2: ssissippi\$mi

sort

Property of the BWA

- Same characters appear nearby
 - e.g., 'T' frequently appears before 'his is'
- And it's decodable!

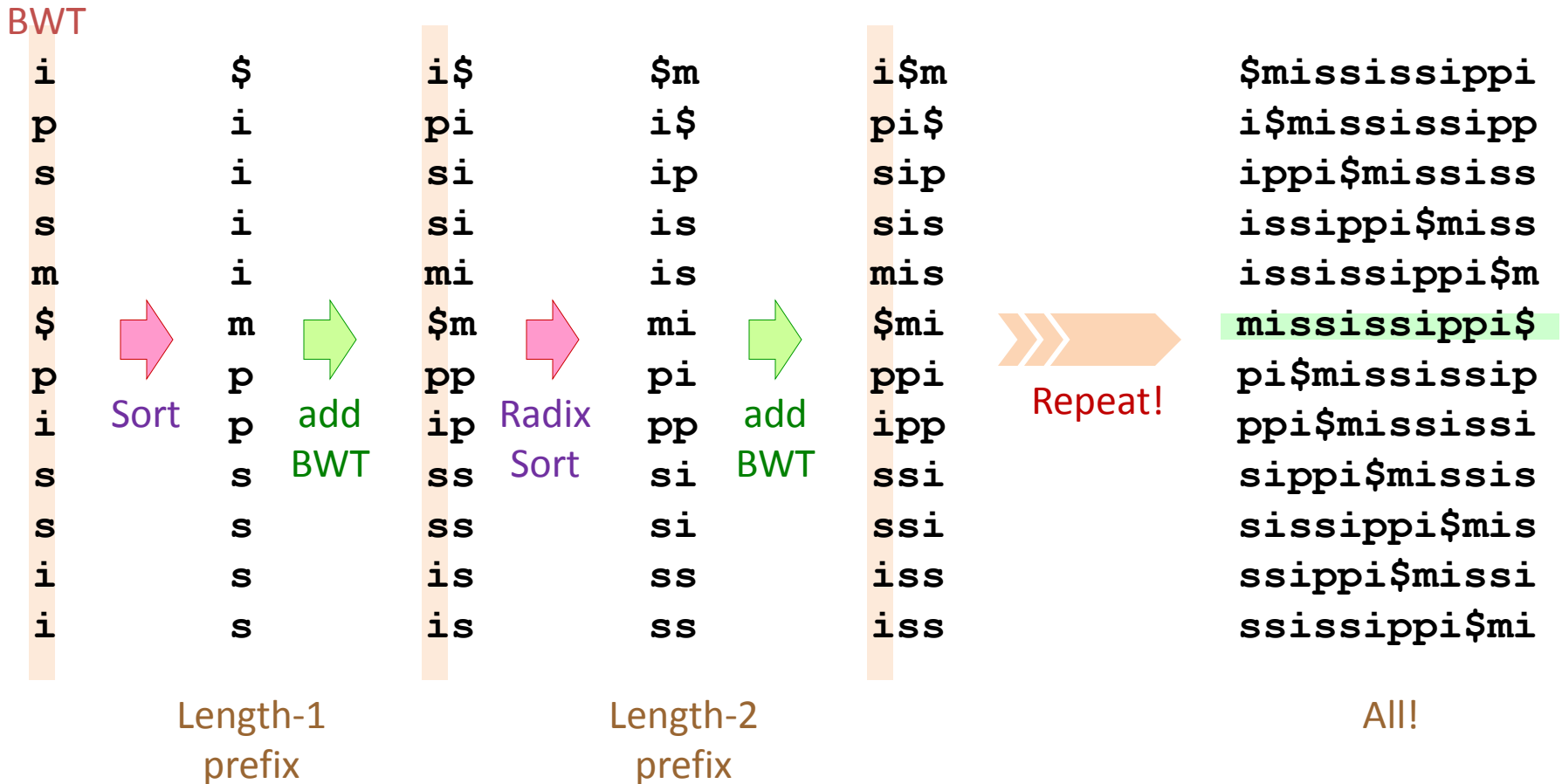
mississippi\$


decode

BWT	SA
10: i	11: \$mississippi
9: p	10: i\$mississipp
6: s	7: ippi\$mississ
3: s	4: issippi\$miss
0: m	1: ississippi\$m
11: \$	0: mississippi\$
8: p	9: pi\$mississip
7: i	8: ppi\$mississi
5: s	6: sippi\$missis
2: s	3: sissippi\$mis
4: i	5: ssippi\$missi
1: s	2: ssissippi\$mi

■ n -digit radix sort

◆ Time complexity: $O(n^2)$



MTF (move to front) Coding

- How many kinds of characters appeared after the same character appeared before?
 - $O(n \log s)$
 - Easy to decode
- MTF+BWT values are usually very small
 - i.e., small values (0,1, etc) appears very frequently
 - i.e., we can compress it with some other algorithm, like Huffman / arithmetic coding!

Text	adbdbaabcdadc								
Coded string	03211201133212								
Translation table	<table><tr><td>0</td><td>aadbdbaabcdadc</td></tr><tr><td>1</td><td>bbadbdbbabcddad</td></tr><tr><td>2</td><td>ccbaaaadddbacca</td></tr><tr><td>3</td><td>ddcccccccdbbbb</td></tr></table>	0	a adbdb a abcdadc	1	bbadbdb b abcd d ad	2	ccb a aa a ddd b acc a	3	d d cccccc c dbbbb
0	a adbdb a abcdadc								
1	bbadbdb b abcd d ad								
2	ccb a aa a ddd b acc a								
3	d d cccccc c dbbbb								
	↑ Initial table (have to be remembered)								

Compressing suffix arrays

- Still searchable!
- Compression ratio is in proportion to the entropy!

0: mississippi\$		10: i\$
1: ississippi\$		7: ippi\$
2: ssissippi\$		4: issippi\$
3: sissippi\$		1: ississippi\$
4: issippi\$		0: mississippi\$
5: sippi\$	➔	9: pi\$
6: sippi\$	Sort	8: ppi\$
7: ippi\$		6: sippi\$
8: ppi\$		3: sissippi\$
9: pi\$		5: sippi\$
10: i\$		2: ssissippi\$

Suffix Array

Ψ (Psi) Array

- The position of each following suffix for suffixes in SA
 - ◆ Opposite of the BWT
- Do not remember the original text
 - ◆ It's decodable!

Index / All suffixes

0: mississippi\$
1: ississippi\$
2: ssissippi\$
3: sissippi\$
4: issippi\$
5: sippi\$
6: sippi\$
7: ippi\$
8: ppi\$
9: pi\$
10: i\$
11: \$

Sort

Index / Suffix array / suffixes

0:11: \$
1:10: i\$
2: 7: ippi\$
3: 4: issippi\$
4: 1: ississippi\$
5: 0: mississippi\$
6: 9: pi\$
7: 8: ppi\$
8: 6: sippi\$
9: 3: sissippi\$
10:5: ssippi\$
11:2: ssissippi\$



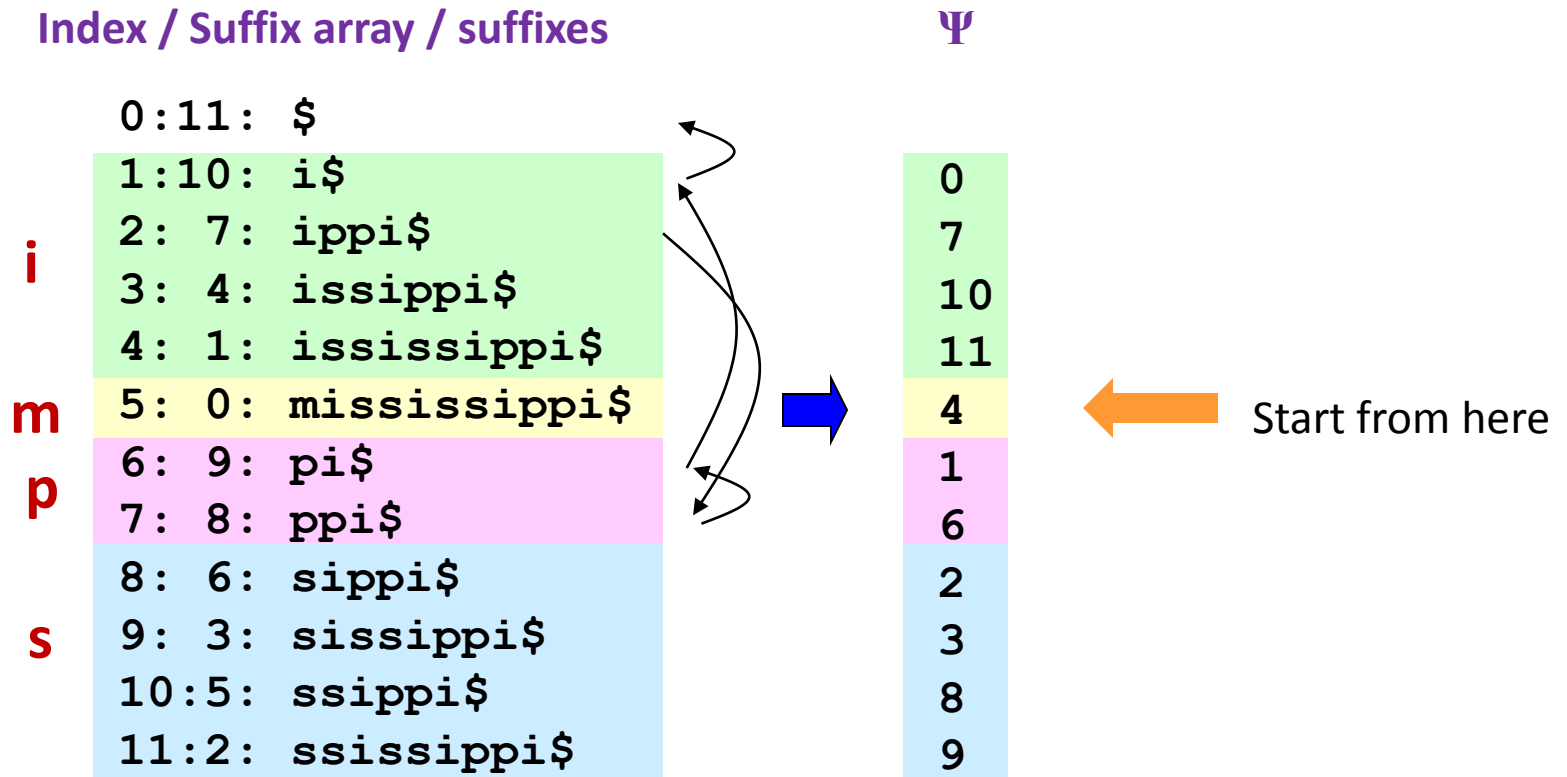
→

Ψ

0
7
10
11
4
1
6
2
3
8
9

Remember

- ◆ Which is the longest suffix
- ◆ Number of each character in the text



Compressing Ψ (1)

- Monotonically increasing within each alphabet block
 - Remember the difference between the adjacent Ψ values
 - Many small values (as in the case of MTF+BWT)
 - Many similar phrases! (e.g. $6-1 = 8-3$ below)
 - i.e., compressible!

	Index / Suffix array / suffixes	Ψ	$\Psi_i - \Psi_{i-1}$
	0:11: \$		
i	1:10: i\$	0	-
	2: 7: ippi\$	7	7
	3: 4: issippi\$	10	3
	4: 1: ississippi\$	11	1
m	5: 0: mississippi\$	4	-
	6: 9: pi\$	1	-
p	7: 8: ppi\$	6	5
	8: 6: sippi\$	2	-
s	9: 3: sissippi\$	3	1
	10:5: ssippi\$	8	5
	11:2: ssissippi\$	9	1

Compressing Ψ (2)

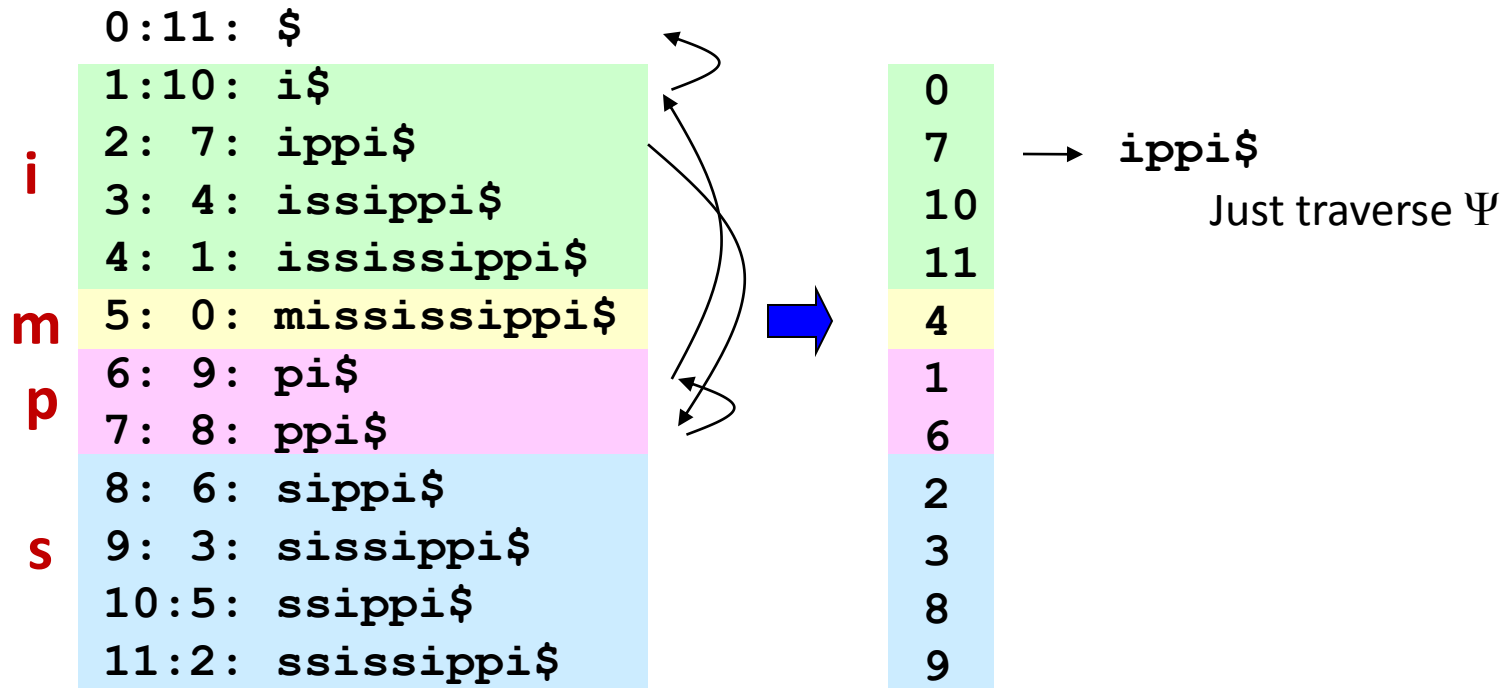
- It takes $O(n)$ time to extract the Ψ values
 - ◆ So remember some of the Ψ values to reduce it!
 - ◆ e.g. Remember $O(n/\log n)$ values and we can extract any Ψ value in $O(\log n)$ time

Original:	3	4	7	8	13	24	26	32	37	45	54	61	70	...
Difference:	-	1	4	1	5	9	2	6	5	8	9	7	9	...

- ▣ Decodable from anywhere on Ψ
- ▣ Binary search!
 - ◆ $O(m \log n)$

Index / Suffix array / suffixes

Ψ



But where is it on the original text?

■ We must know SA[i]

- ◆ If we traverse Ψ , we can extract it, but it takes $O(n)$ time
- ◆ So remember some of the SA[i] values, too!
 - ▶ If we remember $O(n / \log n)$ values, we can extract any SA[i] value in $O(\log n)$ time.

	Index / Suffix array / suffixes	Ψ
	0:11: \$	
i	1:10: i\$	0
	2: 7: ippi\$	7
	3: 4: issippi\$	10
	4: 1: ississippi\$	11
m	5: 0: mississippi\$	4
p	6: 9: pi\$	1
	7: 8: ppi\$	6
s	8: 6: sippi\$	2
	9: 3: sissippi\$	3
	10:5: ssippi\$	8
	11:2: ssissippi\$	9

▣ Various compression algorithms

- ◆ Huffman coding
- ◆ Shannon–Fano coding
- ◆ Tunstall coding
- ◆ Run–length coding
- ◆ Golomb coding
- ◆ Adaptive Huffman coding
- ◆ Arithmetic coding
- ◆ PPM
- ◆ LZ77, LZ78, LZW
- ◆ Block sorting (BWT+MTF)
- ◆ Compressed suffix array

Homework

Submit the report on one of the following problems

- ◆ in English or in Japanese
- ◆ to the report box in front of the office of the department of computer science (1F, Fac. Sci. Bldg. 7)
- ◆ until 5th August

1. Construct an example for which Horspool algorithm requires fewer comparison than Boyer–Moore algorithm. Discuss why it happens.
2. Discuss how to use Aho–Corasick algorithm when wild cards are permitted only in the text / only in the query / in both the text and query.
3. Describe any algorithm (on any problem) which achieves linear time by utilizing the strategy: $f(n)=c_1 \cdot f(c_2 \cdot n)+O(n)$ ($f(n)$ is the time complexity of the overall algorithm against an input of size n).
4. Pick up two compression algorithms and compare them. Discuss their good points and bad points.
5. Make a scribe note on any of my 3 lectures (in TeX format and send it to me via e-mail)

with some impressions on my lecture if you have

- ◆ Optional, *i.e.*, unrelated to grading