Strings: String Sorts

- The lower bound for comparison based sorting is $O(N \log N)$
- A string is a sequence of finite letters drawn from some alphabet
  - ↳ a word in the English language
  - ↳ a text file; where letters are ASCII characters
  - ↳ the binary code over $\Sigma = \{0, 1\}$
  - ↳ DNA sequence over $\Sigma = \{A, C, G, T\}$
- Pages are strings; an entire book is also a string
- The length of the string is written as $|w|$, and Σ is the alphabet
  - ↳ If $|w| = 0$, then the string "w" is empty string and written as "ϵ", which stands for epsilon
- If $w = uxv$ for a string, then:
  - ↳ "u" is called a prefix
  - ↳ "x" is called a substring
  - ↳ "v" is called a suffix of "w"
  - The string itself can be considered a prefix/suffix
- Example: Given a string $w = ababaabb$
  - ↳ "ab" is a prefix of w
  - ↳ "aba" is a substring of w
  - ↳ "abb" is a suffix of w

- When comparing strings, the running time is proportional to length of longest common prefix
  - ↳ proportional to $|w|$ in worst case
    - ↳ But, often sublinear in w
- Radix: Number of characters, R, in the alphabet
  - ↳ i.e. $R = 2$ for Binary, where $\Sigma = [0, 1]$
- Log R: Number of bits required to represent a character in the alphabet
  - ↳ i.e. $\log R = 1$ for Binary
- Characters in the alphabet set need to be ordered, so we can compare strings
- Summary of the performance of sorting algorithms

Algorithm	Guarantee	Random	Extra Space	Stable
Insertion Sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	✓
Merge Sort	$N \lg N$	$N \lg N$	N	✓
Quick Sort	$1.39\ N \lg N$	$1.39\ N \lg N$	$c \lg N$	
Heap Sort	$2N \lg N$	$2N \lg N$		

- The Key counting sort algorithm does not compare keys
  - ↳ it is a stable algorithm
- A stable algorithm maintains previous order of (equal) elements

[redacted]

- Key indexed counting takes time proportional to $(N + R)$
  - ↳ uses extra space proportional to $(N + R)$
- Stable sorting: Previous ordering is still maintained, even though the key-values are all the same
- LSD string (radix) sort
  - ↳ LSD = least significant digit
  - ↳ Consider characters from right to left
  - ↳ Stably sort using d^{th} character as the key (using key-indexed counting)
- LSD sorts fixed-length strings in ascending order
- LSD [redacted] sort is stable because key-indexed counting is stable
- LSD string sort takes time proportional to $W(N + R)$, where W is the size of the fixed length strings
- Reverse LSD does not work
  - ↳ sorting characters from left to right messes up the table
- MSD String (radix) Sort
  - ↳ MSD = most significant digit
  - ↳ Uses key-indexed counting; just like LSD
  - ↳ Partition array into "R" pieces according to first character
  - ↳ Recursively sort all strings that start with each character
- For variable length strings, you add an extra character at the end, which represents the terminating character
- In MSD, you move from left to right
  - ↳ In LSD you move from right to left
- LSD and MSD heavily depend on the characters in the keys
- MSD is too slow for small subarrays
  - ↳ Each function call requires its own "count[]" array
  - ↳ Solution: Switch to insertion sort for small sub-arrays (cut-off at 10 - 15)
- MSD's best case time complexity is similar to key counting sort
  - ↳ worst case is LSD sort
    - ↳ occurs when strings are same length
- MSD string sort takes anywhere between $O(N + R)$ and $O(W(N + R))$ where W is the average size of the fixed length strings.

Strings: Tries

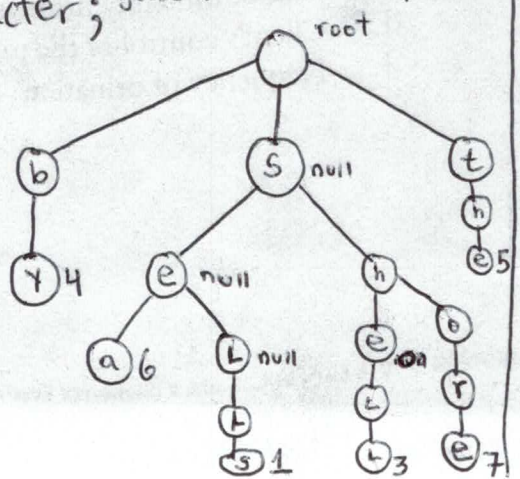
- String tries is a data structure that stores key-value pairs in a different manner than RBTs
- The trie data structure is ideal when using strings
- Summary of the performance of symbol table implementations:

Implementation	Typical Case			Ordered operations	Operations on keys
	Search	Insert	Delete		
Red-Black BSTs	log N	log N	log N	Yes	compareTo
Hash Table	1*	1*	1*	NO△	equals hashCode

*Under Uniform Hashing Assumption
△Hashing is not suitable for ordered sets

- Tries
  - → Store characters in nodes; not keys
  - → Each node has "R" children, one for each possible character; good for small alphabets

Key	Value
by	4
sea	6
sells	1
she	0
shells	3
shore	7
the	5



- In a trie, a search hit is when the node, where search ends, has a non-null value (i.e. get("sea"))
  - → search miss is when you reach a null link or node where search ends has null value (i.e. get("sell")), (i.e. get("shelter")
- Trie Performance
  - → Search hit: Need to examine all characters in the key to test for equality
  - → Search miss:
    - → Could have mismatch character on first try
    - → Typical case: Examine only a few characters (sublinear)
  - → The average number of nodes examined for search miss in a trie built from N random keys over an alphabet of size R is: $\sim \log_R N$
    - → Search miss does not depend on the key length
- The R-way trie contains R links at each node
- The number of links in a trie is between "RN" and "RNw", where "w" is the average key length
  - → space requirement ranges from O(RN) to O(RNw)
- ■ Tries give fast search hit and even faster search miss, but wastes space.

Strings: Substring Search

- The goal of substring search is to find pattern of length "M" in a text of length "N"
  - ↳ typically N >> M
  - → also referred to as pattern matching
  - ↳ i.e. pattern = NEEDLE
    text = IN A HAYSTACK NEEDLE
- Brute force is one approach to pattern matching
  - → it involves aligning the pattern at each index position of the text
  - ↳ worst case: $O(N \cdot M)$
- Knuth-Morris-Pratt (KMP) Algorithm
  - → most famous pattern matching algorithm
  - → takes advantage of previous matchings, and avoids repeating those redundant matchings
    - ↳ computes the longest border of every prefix of "p" to achieve this
  - → runs in $O(N+M)$ time, where "N" is the length of the text and "M" is the length of the pattern
  - → not very fast in practice
- A string "v" is said to be a border of a string "w" if it is both a prefix and a suffix of "w", and of length < |w|
  - ↳ i.e. v = aba is a border of w = abacaba; in fact, "w" has two borders: "a", "aba"
- The borders of a string can overla
- If a string is of length "N", then the longest possible length of a border is "N - 1"
- A string "w" can have at most |w borders, including the empty border = ϵ
  - → and at most |w-1| non-empty borde
  - ↳ i.e. the string w = abacaba has 3 borders: "a", "aba", ϵ
- A border array β_x of x is an integer array of length "n", where the i-th element of the array is equal to the length of the longest border of $x[1..i]$
  - ↳ i.e. The border array β_w of w = abacaba is shown below:

	1	2	3	4	5	6	7
w	a	b	a	c	a	b	a
β_w	0	0	1	0	1	2	3

- $\beta_x[1]$ is always 0 b/c there is only 1 character in the string
- To compute the Border array $\beta_x[i]$, first check if the current longest border, $\beta_x[i-1]$, can be extended
  - ↳ only possible when $x[i+1] = x[b+1]$
- Computing the border array requires $\Theta(n)$ time and constant additional space
- If "b" is a border of "b", and "b" is a border of "x", then "b" is a border of "x"
- The KMP algorithm uses the border array to avoid repeating redundant matchings

- KMP Algorithm Explanation
  - Each time there is a match, increment the current indices
  - If there is a mismatch, and progress in "p" has been made, then consult the border array of "p" to determine the new index in "p", where we need to continue checking "p" against "T"
  - Repeat this process until a match of "p" in "T" is found, or the index for "T" reaches "n"
- The KMP Algorithm runs in $O(N+M)$ time
  - "M" is the time required to compute "β_x"
- Boyer Moore Algorithm
  - Scans text from left to right; just like KMP
    - But, scan characters in pattern, for matches, from right to left
    - Perform shift = max. shift by applying the bad character rule
  - BM is better than KMP, in terms of performance
    - BM is used in a lot of cases for pattern matching and string searching
- The bad character rule states that upon mismatch, skip alignments until:
  a) mismatch becomes a match, or
  b) the pattern moves past the mismatched character; this is the bad character

CSP3-2

- Boyer Moore allows you to skip as many as "M" text characters when finding one not in the pattern
- If the Boyer Moore algorithm only uses the bad character rule, then it can perform as bad as the Brute Force algorithm.
  - This is because the bad character rule does not help in all cases
  - runtime is $O(N \cdot M)$
- The skip table maintains information about the index of the rightmost* occurrence of character "c" in a given prefix of the pattern
  - skiptable[i,j] = the index of rightmost* occurrence of character "i" in prefix of length (j-1) in the pattern

* The rightmost occurrence is relative to the index at 0

- Skip Table Example:

		0	1	2	3	4	5
		N	E	E	D	L	E
0-2	A-C	-1	-1	-1	-1	-1	-1
3	D	-1	-1	-1	-1	3	3
4	E	-1	-1	1	2	2	2
5-10	F-K	-1	-1	-1	-1	-1	-1
11	L	-1	-1	-1	-1	-1	4
12	M	-1	-1	-1	-1	-1	-1
13	N	-1	0	0	0	0	0
14-25	O-Z	-1	-1	-1	-1	-1	-1

Skip table for "NEEDLE"

- The values in the skip table tell us how many characters we can shift
- Boyer Moore performs really well on English text / patterns
  - because not all text characters are examined
  - BM is usually faster than KMP for structured texts
- Boyer-Moore with bad character rule takes about ~(N/M) character compares to search for a pattern of length "M" in a text of length "N"
  - worst case can be as bad as (M·N)
    - BM with good suffix rule can improve worst case to ~3N character compares by adding a KMP-like rule to guard against repetitive patterns
- The Rabin-Karp Algorithm uses hashing for pattern matching
  - compute a hash of pattern [0..M-1]
  - for each $1 \le i \le n$, compute a hash of txt[i..M+i-1]
    - If hash of pattern [0..M-1] = txt[i..M+i-1] hash, then align pattern at index "i" and perform a brute force comparison for each match, checking character by character.
  - If hash(x) ≠ hash(y), then x ≠ y
  - Works on numerical values and any character based strings
    - used for checking plagiarism
- The hash value has to be computed at each alignment, from 0 to N-1
  - efficiently done using modular hashing and Horner's method

CSP3-2

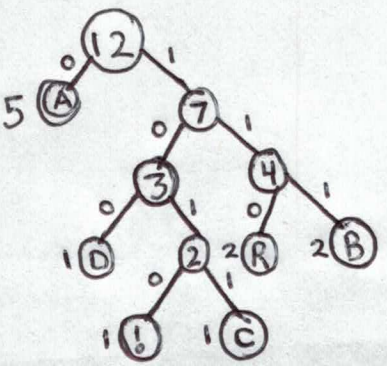
- To prevent overflow and keep numbers small, the MOD of intermediate numbers is computed
  - $(a+b)\ \text{MOD}\ Q = [(a\ \text{MOD}\ Q) + (b\ \text{MOD}\ Q)]\ \text{MOD}\ Q$
  - $(a \times b)\ \text{MOD}\ Q = [(a\ \text{MOD}\ Q) + (b\ \text{MOD}\ Q)]\ \text{MOD}\ Q$
- Rolling hash is used to efficiently compute X_{i+1} from X_i in constant time
- For the first "R" entries, Horner's Rule is used
  - for the remaining entries, rolling hash and MOD (to avoid overflow) are used
- If there is a hash match, then check character-by-character
  - Brute-force checking of patterns
- In the worst case, Rabin-Karp takes O(N·M)
  - same as brute force algorithm
  - worst case occurs when there is a hash match at every alignment, causing a brute-force checking / comparisons for the strings
- Time complexity is mostly dependent on the hash function
  - trick is to have the least amounts of hash hits / match as possible
  - good hash functions reduce the amount of hash hits, thus many alignments can be skipped
- Rabin-Karp is a fingerprint search, because it uses a single numerical value to represent a (potentially very large) pattern. Then, it looks for this fingerprint (hash value) in the text. If there is a match, it does a brute-force check

Strings: Data Compression

- Compression reduces the size of files
  - → saves space when storing it
  - → saves time when transmitting it
  - → most files have a lot of redundancy
    - → allows for lossless compression
- Strings that only contain unique characters cannot be compressed
  - → because compression relies on the fact that there is redundancy in the data
- Lossy compression is mostly used for pictures and videos
  - → with lossy compression, you cannot reconstruct the original file
- All data on a computer is ultimately represented in Binary
  - → the alphabet of binary = [0, 1]
- Compression Ratio = $\left[\frac{\text{bits in compressed message}}{\text{bits in original message}}\right]$
- No algorithm can compress every bit string
  - → there is a limit to how much something can be compressed
- The English language has a lot of redundancy
  - → redundancy allows for compression
- Run-length Encoding
  - → simple type of encoding
  - → relies on long runs of repeated bit
  - → i.e. AAAAA A BBBB → 6A4B
  - → i.e. 000000011111 → 75
    - → there are 7 zeros and 5 ones
  - → Applications: JPEG, bioinformatics etc.
- Variable-length codes use different number of bits to encode different characters
  - → allows us to use the fewest number of bits on the most-frequently used characters, and the least-frequently used characters use the most number of bits
  - → Since the codes vary in length, they need to be unique so there is no ambiguity during decompression
    - → a codeword cannot be the prefix of another
      - → i.e. A=01, B=011 is not valid
    - → 3 ways to accomplish this:
      - → fixed-length code
      - → append special stop character to each codeword
      - → general prefix-free code
- IF there is a wide variance in character frequencies, the (space) savings produced by a variable-length prefix code can be significant
- Decoding is done via the greedy strategy; for variable-length prefix codes.

- Binary trie is used to represent prefix-free code
  - ↳ all values in a trie are unique
- Expanding a Binary trie
  - ↳ Start at root
  - ↳ go left if bit is 0; go right if 1
  - ↳ If leaf node, print char and return to root
- Huffman Coding
  - ↳ saves a substantial amount of space in files with a lot of redundancy
    - ↳ particularly effective for natural language files
  - ↳ uses shorter binary codes to represent each character, using fewer bits
  - ↳ each message/data stream has its own custom prefix-free code
    - ↳ must send encoded table along with compressed message, so the receiver can decode it
    - ↳ known as dynamic encoding
- Huffman encoding is not unique
  - ↳ a message can have different encodings that are equally optimal
- When building a Binary trie, always
  - Merge the minimum weight tries into single tries
    - ↳ if multiple tries have the same weight, pick whichever when you want b/c it does not make a difference
- All optimal binary tries will result in the same size for the compressed message/data stream.
- Summary: The Huffman algorithm produces an optimal prefix-free code
- A Huffman compressed message cannot be decoded without the trie
  - ↳ the cost of the trie must be included in the compressed output
- Huffman compressed output = [compressed bit string] + [trie]
- Tries are written using preorder traversal, and this is sent as a bitstream
- Huffman code example:
  - ↳ input string: ABRACADABRA
  - ↳ trie table:

char	freq	encoding
A	5	0
B	2	111
C	1	1011
D	1	100
R	2	110
!	1	1010



- Note: For all tries, the left side is 0 and the right side is 1
- Preorder Traversal:
  1A23D4!C5RB

- Different encoding models:
  - static model: same for all texts
    - fast
    - not optimal because different texts have different statistical properties
    - i.e. ASCII, morse code
  - dynamic model: generate model based on text
    - preliminary pass/scan of the text needed to generate model
    - must transmit the model/table
    - i.e. Huffman code
  - adaptive model: progressively learn and update model as you read text
    - more accurate modeling produces better compression
    - i.e. Lempel-Ziv-Welch (LZW)

- Lempel-Ziv-Welch (LZW) compression:
  - one of the most widely used compression methods
    - easy to implement and works well for a variety of file types
    - i.e. used in gif and unix files
  - the trick is to encode substrings to a single codeword of fixed length
    - codewords are added to a symbol table. However, this ST is not sent with the compressed message / data stream.

- LZW Compression Algorithm
  - the symbol table is initialized with 128 possible single character keys
  - hexadecimal is used to refer to codeword values
    - 41 is the codeword for ASCII-A
    - 52 is the codeword for ASCII-R
    - codeword "80" is reserved to signify end of file
    - codeword values 81 through FF are assigned to various substrings of the input

- LZW Compression Example:
  - input string: ABRACADABRABRABRA (unique, repeating)

Symbol Table	
A	41
B	42
C	43
D	44
R	52
AB	81
BR	82
RA	83

input:	A	B	R	A	C	A	D	A	B	R	A	B	R	A
matches:	A	B	R	A	C	A	D	AB			BR			RA
output:	41	42	52	41	43	41	44	81			82			83

AC	84
CA	85
AD	86
DA	87
ABR	88
BRA	89

- The encoded string is:
  41 42 52 41 43 ... 80
- Remember to append "80" to the end of the encoded string

- LZW would not work very well if there is not a lot of repeating substrings in the input
- The LZW compression symbol table is represented with a trie
- LZW expansion works in a similar way to LZW compression; uses an inverse symbol table
  - maintain a symbol table that associates strings of characters with codeword values
    - the first few entries in the table are filled with one-character strings
  - read the encoded message and build upon the symbol table until you reach "80".
- The LZW expansion algorithm has a tricky case
  - happens when the codeword is the same as the table entry to be ~~completed~~ completed