

simple string processing

NOTE: in general avoid many of these unsafe C-string functions (see CERT)

glibc docs explicitly state that usage of strcat is the sign of a lazy programmer

strlen(a)

```
int i;
for (i = 0; a[i] != '\0'; i++) ;
return i;

// pointers
b = a;
while (*b++) ;
return b - a - 1;
```

strcpy(a, b)

```
int i;
for (i = 0; (a[i] = b[i]) != 0; i++); // assignment like this returns the value of

// pointers
while (*a++ = *b++) ;
```

strcmp(a, b)

```
for (i = 0; a[i] == b[i]; i++) {
    if (a[i] == '\0') {
        return 0;
    }
}
return a[i] - b[i];

// pointers
while (*a++ == *b++) {
    if (*(a - 1) == '\0') {
        return 0;
    }
}
```

```
}  
}  
return *(a - 1) - *(b - 1);
```

strcat(a, b)

```
strcpy(a + strlen(a), b);
```

Strings (from Algorithms)

- uses:
 - information processing
 - genomics
 - communications systems
 - programming systems
- string characteristics:
 - characters
 - generally 2^{16} (16 bit Unicode)
 - previously 7 bit or extended 8 bit Unicode
 - immutability (string literal pool)
 - indexing (0 indexed)
 - length
 - substrings in constant time (review how)
 - concatenation in linear time
 - character arrays
 - alphabets (collections of character sets)
 - binary
 - DNA
 - octal
 - decimal
 - hexadecimal
 - protein
 - lowercase
 - uppercase
 - base 64
 - ascii
 - extended ascii
 - unicode 16
 - numbers

String Sorts

- two general methods:
 - LSD - examines characters from right-to-left
 - MSD - examines characters from left-to-right
- affected by the number of characters in the alphabet (256 for 8 bit, 65,536 for 16 bit Unicode)

Key-indexed counting

1. compute frequency counts
 - use keys of small integers
 - maintain and increment the index of the key + 1 in an array (IMPORTANT!!!)
 - use the counts as indices
 2. move items to auxiliary array
 - place items in the array based on the count as index
 - maintain relative ordering for elements with equivalent keys
 3. copy elements from the auxiliary array back to the original
- properties:
 - key-indexed counting uses $2N + 3R + 1$ array accesses to stably sort N items whose keys are integers between 0 and $R - 1$
 - effective for keys of small integers (breaks $N \log(N)$ lower bound)

LSD string sort

- properties:
 - LSD string sort stably sorts fixed-length strings
 - LSD string sort uses about $7WN + 3WR$ array accesses and extra space proportional to $N + R$ to sort N items whose keys are W -character strings taken from an R -character alphabet
- requires fixed-length strings
- perform key-indexed counting moving from right to left
- on each pass move the sorted elements into the auxiliary array and then copy back
- LSD radix sorting was used by old (card punch) computers
- interesting because it is linear time sort for typical applications

MSD string sort

- general purpose where strings are not always the same length

- strings are sorted using key-indexed counting from left-to-right recursively
- need to ignore parts of the array for which end-of-string has already been reached (and thus is sorted) by returning -1 from a function that returns the char as integer if the access is past the end of the string
- with this, the code is basically the same as LSD sort
- can gain performance for smaller alphabets by replacing the charAt function to accessing the index of the char within the alphabet
- for large character sets, small subarrays are essential and MSD must be switched to insertion sort
- for subarrays containing large numbers of equal keys, key-indexed counting still needs to be performed, but this causes the worst-case performance for MSD
- performance:
 - for random inputs, MSD is sublinear
 - for non-random inputs, MSD can still be sublinear, but will need more character checks
 - worst-case is input with all equal strings and is linear
- properties:
 - to sort N random strings from an R-character alphabet, MSD string sort examines about $N \cdot \log_{\text{sub } R} (N)$ characters on average
 - MSD string sort uses between $8 \cdot N + 3 \cdot R$ and about $7 \cdot w \cdot N + 3 \cdot w \cdot R$ array accesses to sort N strings taken from an R-character alphabet, where w is the average string length
 - to sort N strings taken from an R-character alphabet, the amount of space needed by MSD string sort is proportional to R times the length of the longest string (plus N) in the worst case

Three-way string quicksort

- uses 2-way partitioning on the leading character of a string to perform quicksort and does this recursively
- use small subarray optimization of insertion sort to skip characters known to be equal
- use the same small alphabet optimization as above

- properties:
 - to sort an array of N random strings, 3-way string quicksort uses about $2N \ln(N)$ characters compares on average

Choosing a string sort

algorithm	stable	inplace	time	space
insertion sort	X	X	$N \rightarrow N^2$	1
quicksort		X	$N \log^2(N)$	$\log(N)$
mergesort	X		$N \log^2(N)$	N
3-way quicksort		X	$N \rightarrow N \log(N)$	$\log(N)$
LSD string sort	X		NW	N
MSD string sort	X		$N \rightarrow Nw$	$N + WR$
3-way string qsort		X	$N \rightarrow Nw$	$W + \log(N)$

Tries

- aka R-way tries
- use a symbol table implementation with the following additions:
 - longest prefix of:
 - takes a string as an argument and returns the longest key in the symbol table that is a prefix of that string
 - keys with prefix:
 - takes a string as an argument and returns all of the keys in the symbol table having that string as a prefix
 - keys that match:
 - takes a string as an argument and returns all of the keys in the symbol table that match that string (using a regex or simplified regex)
- the name comes from retrieval
- basic properties:
 - tree-like structure (nodes with references to nodes that are null or nodes)
 - each (potentially) has R links, where R is the alphabet size
 - words appear as full paths in a tree, with the first level corresponding to the first characters of all words in the symbol table, second level to second characters, etc
 - store the value associated with each key in the node corresponding to its last character
- search (find value associated with a given key):
 - start at root and follow links according to each successive character in the key string
 - if the value in the node associated with the last letter of the search key is not null, result is a search hit

- if it is null, the key is not in the table
- if the search terminates with a null link, it is a search miss
- trie:
 - stores radix of alphabet (size)
 - defines a node as a node array of size R and a value
 - starts with a dummy root node
 - size (three options):
 - maintain number of keys in outer abstraction
 - maintain number of keys in each node and outer abstraction
 - lazily compute the size recursively
 - collecting keys
 - wildcard match
 - longest prefix
 - deletion
 - alphabet
- properties:
 - the structure of a trie is independent of key insertion order, i.e. there is a unique trie for each set of keys
 - the number of array accesses when searching in a trie or inserting a key into a trie is at most 1 plus the length of the key
 - the average number of nodes examined for search miss in a trie built from N random keys over an alphabet of size R is about $\log_{\text{sub } R} N$
 - the number of links in a trie is between RN and RNw, where w is the average key length
- may use one-way branching:
 - store keys in the leaf node at the point where the key becomes distinct (i.e. to eliminate long tails of single nodes in a one-way branch)
- do not try to use tries for large numbers of long keys taken from large alphabets (especially many dissimilar keys) because it will require space proportional to R times the total number of key characters
- if the space of a trie is affordable, the performance is difficult to improve upon

Ternary search tries (TSTs)

- used to avoid the space cost of R-way tries
- each node has:
 - a character

- three links
 - less than
 - equal to
 - greater than
- a value
- properties:
 - the number of links in a TST built from N string keys of average length w is between 3N and 3Nw
 - a search miss in a TST built from N random sting keys requires about $\ln(N)$ character compares, on average. A search hit or an insertion in a TST uses a character compare for each character in the search key
- may use one-way branching:
 - store keys in the leaf node at the point where the key becomes distinct (i.e. to eliminate long tails of single nodes in a one-way branch)

Choosing a string symbol table

for N strings from an R-character alphabet (average length w)

algorithm	chars for miss	memory usage	sweet spot
BST	$c \cdot \lg(N)^2$	64N	randomly order
red-black/2-3	$c \cdot \lg(N)^2$	64N	guaranteed per
linear probing	w	32N → 128N	built-in types
R-way trie	$\log \cdot R(N)$	$(8R+56)N \rightarrow (8R+56)Nw$	short keys, sm
TST	$1.39 \lg(N)$	64N → 64Nw	nonrandom keys

- if space is available, R-way tries provide the fastest search
- for large alphabets without adequate space, TSTs are preferable
- hashing is competitive but does not work well for extended character APIs

Substring Search

- find an occurrence of a string, length M, within a larger string, length N
- common brute force algorithm runs in MN worst-case time, with M+N general time
- Cook → Knuth & Pratt → Knuth, Morris, Pratt in 1976
- Boyer, Moore around the same time
- both KMP and Boyer-Moore require complex processing

- Rabin-Karp 1980, $M+N$, simpler processing using hashing

Brute-force substring search

- scan through the indices $0 - (N-M)$ ($N-M$ because inner loop's final check will be from index $N-M$ to N , which is a check of length M)
- at each index scan from current index, to current index + M , checking each character at current index plus inner index for equivalence, and breaking if not
- if inner index is equal to M , return the value of the outer index, i.e. the index of the start of the substring match
- properties:
 - brute-force substring search requires about NM character compares to search for a pattern of length M in a text of length N , in the worst case

Knuth-Morris-Pratt substring search

- use the fact that, when a mismatch is detected at any given index, the inner loop might have already scanned forward so additional characters may be known
- additionally, can increment the outer index to any matching start character to run the next scan
- the KMP algorithm precisely defines a deterministic finite-state automaton (DFA) on text

```
// m == size of outer search string
int search(std::string txt)
{
    int i;
    int j;
    int n = txt.size();
    for (i = 0, j = 0; i < n && j < m; i++) {
        j = dfa[txt[i]][j];
    }
    if (j == m) {
        return i - m; // found
    } else {
        return n;
    }
}
```

- to construct the dfa (using a two dimensional character array, first dimension of R or full alphabet:
 - copy `dfa[][x]` to `dfa[][j]` for mismatch cases
 - set `dfa[pat[j]][j]` to $j + 1$ for the match case
 - update X


```

dfa[pat.charAt(0)][0] = 1;
for (int X = 0, j = 1; j < M; j++)
{ // Compute dfa[][j].
    for (int c = 0; c < R; c++) {
        dfa[c][j] = dfa[c][X];
    }
    dfa[pat[j]][j] = j+1;
    X = dfa[pat[j]][X];
}

```

- properties:
 - Knuth-Morris-Pratt substring search accesses no more than $M+N$ characters to search for a pattern of length M in a text of length N

Boyer-Moore substring search

- when backup is not a problem, right-to-left string scan can speed substring search
- mismatched character heuristic:
 - when scanning from right to left, can check for a final character match, then check for initial character equivalence at a distance of substring length to the left, then know we can skip left substring length + 1 if there is a mismatch
- process:
 - generate a skip table of the rightmost occurrence of each character in the search pattern
 - have an index i moving from left to right and an index j moving from right to left
 - inner loop tests for a pattern match at position i , from right to left
 - for a mismatch, do one of the following:
 - if character causing mismatch is not found in the pattern, slide the pattern $j + 1$ positions to the right
 - if it is found, use the skip table to line the pattern up with the character's rightmost occurrence
 - if this computation would not increase i , increase i by 1 to ensure the pattern always slides at least 1 to the right
- on typical inputs, substring search with the Boyer-Moore mismatched character heuristic uses about N/M character compares to search for a pattern of length M in a text of length N

Rabin-Karp fingerprint search

- substring search that uses hashing, i.e. compute a hash function for the pattern and look for the match using the same hash function for each possible M -length substring of the input text
- general process:
 - a string of length M corresponds to an M -digit base- R number

- for a hash table of size Q , convert this number to an int between 0 and $Q-1$
- use Horner's method to compute the hash
- Rabin-Karp rests on the idea that Horner's method enables constant time right shift of computing the substring hash for substring size M
- to avoid a hash collision, use a hash table of size Q larger than necessary
- use Monte Carlo correctness, i.e. a large hash table size Q such as 10^{20} or 10^{40}
- properties:
 - the Monte Carlo version of Rabin-Karp substring search is linear time and extremely likely to be correct, and the Las Vegas version of Rabin-Karp is correct and extremely likely to be linear time
- known as a fingerprint because it uses a small amount of information to represent a potentially large amount of information

Substring search summary

Cost summary for substring search implementations

algorithm	version	guarantee	typical	backup	co
brute force		MN	$1.1N$	yes	ye
KMP	full DFA	$2N$	$1.1N$	no	ye
KMP	mismatch transitions	$3N$	$1.1N$	no	ye
KMP	full algorithm	$3N$	N/M	yes	ye
Boyer-Moore	mismatched char	MN	N/M	yes	ye
Rabin-Karp	Monte Carlo	$7N$	$7N$	no	ye
Rabin-Karp	Las Vegas	$7N^*$	$7N$	yes	ye

* probabilistic guarantee, with uniform and independent hash function

Regular Expressions

- substring search with less than complete information about the string to be found

Describing patterns with regular expressions

- concatenation - joining character string patterns $\rightarrow \{AB\}$
- or - matching on one pattern or another $\rightarrow A|B$
- closure - allows parts of the pattern to be repeated arbitrarily (higher precedence than concatenation) $\rightarrow A^*$
- parentheses - used to override the default precedence rules
- definition:
 - a regular expression is either:
 - empty

- a single character
- an RE enclosed in parentheses
- two or more concatenated regular expressions
- two or more regular expressions separated by the or operator (|)
- a regular expression followed by the closure operator (*)
- the empty RE represents a set of strings, defined as follows:
 - the empty RE represents the empty set of strings with 0 elements
 - a character represents the set of strings with one element, itself
 - an RE enclosed in parentheses represents the same RE not enclosed in parentheses
 - the RE consisting of two concatenated REs represents the cross product of the sets of string represented by the individual components
 - the RE consisting of two OR'ed REs represents the union of the sets of individual components
 - the RE consisting of the closure of an RE represents the E (empty string) or the union of the sets represented by any number of copies of the RE
- many additions to this set in general RE usage are just shortcuts for describing some addition of sequences of REs
- uses:
 - substring search
 - validity checking
 - programmer's toolbox
 - genomics
 - search
 - possibilities
 - limitations

Nondeterministic finite-state automata

- any given state transition can be branched, i.e. two or more possibilities that offer a choice of transitions (each of which is tested in application)
- Kleene's theorem: there is an NFA corresponding to any given RE
- use a digraph to represent an NFA
 - start at state 0
 - read characters at input and iterate through state transitions until:
 - the set of possible states contains the accept state
 - the set of possible states does not contain the accept state

- properties:
 - determining whether an N-character text string is recognized by the NFA corresponding to an M-character RE takes time proportional to NM in the worst case

Building an NFA corresponding to an RE

- read in the RE expressed as a string
- use a character stack to handle parens and the or operator
- for each character i to length of re:
 - if '(' or '|':
 - push
 - else if ')'
 - pop or
 - if or == '|'
 - pop lp
 - add edge lp-or+1
 - add edge or-i
 - else lp = or
 - if i is less than length - 1 and the next character is '*'
 - add edge lp-i+1
 - add edge i+1-lp
 - if '(' or '*' or ')'
 - add edge i-i+1
- properties:
 - building an NFA corresponding to an M-character RE takes time and space proportional to M in the worst case
- TODO: review to describe
- checking for recognition:
 - use a bag and construct a directed DFS of the RE digraph
 -