```
// Copyright 2011 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

/*
Generating random text: a Markov chain algorithm

Based on the program presented in the "Design and Implementation" chapter
of The Practice of Programming (Kernighan and Pike, Addison-Wesley 1999).
See also Computer Recreations, Scientific American 260, 122 - 125 (1989).

A Markov chain algorithm generates text by creating a statistical model of
potential textual suffixes for a given prefix. Consider this text:

      I am not a number! I am a free man!

Our Markov chain algorithm would arrange this text into this set of prefixes
and suffixes, or "chain": (This table assumes a prefix length of two words.)

      Prefix       Suffix

      "" ""        I
      "" I         am
      I am         a
      I am         not
      a free       man!
      am a         free
      am not       a
      a number!    I
      number! I    am
      not a        number!

To generate text using this table we select an initial prefix ("I am", for
example), choose one of the suffixes associated with that prefix at random
with probability determined by the input statistics ("a"),
and then create a new prefix by removing the first word from the prefix
and appending the suffix (making the new prefix is "am a"). Repeat this process
until we can't find any suffixes for the current prefix or we exceed the word
limit. (The word limit is necessary as the chain table may contain cycles.)

Our version of this program reads text from standard input, parsing it into a
Markov chain, and writes generated text to standard output.
The prefix and output lengths can be specified using the -prefix and -words
flags on the command-line.
*/
package main

import (
      "bufio"
      //"flag"
      "fmt"
      "io"
      "io/ioutil" //use for reading frequency map into string
      "math/rand"
```

```
	"os"
	"strconv"
	"strings"
	"time"
)

// Prefix is a Markov chain prefix of one or more words.
type Prefix []string

// String returns the Prefix as a string (for use as a map key).
func (p Prefix) String() string {
	return strings.Join(p, " ")
}

// Shift removes the first word from the Prefix and appends the given word.
func (p Prefix) Shift(word string) {
	copy(p, p[1:])
	p[len(p)-1] = word
}

// Chain contains a map ("chain") of prefixes to a list of suffixes.
// A prefix is a string of prefixLen words joined with spaces.
// A suffix is a single word. A prefix can have multiple suffixes.
type Chain struct {
	chain     map[string][]string
	prefixLen int
}

// NewChain returns a new Chain with prefixes of prefixLen words.
func NewChain(prefixLen int) *Chain {
	return &Chain{make(map[string][]string), prefixLen}
}

// Build reads text from the provided Reader and
// parses it into prefixes and suffixes that are stored in Chain.
func (c *Chain) Build(r io.Reader) {
	br := bufio.NewReader(r)
	p := make(Prefix, c.prefixLen)
	for {
		var s string
		if _, err := fmt.Fscan(br, &s); err != nil {
			break
		}
		key := p.String()
		c.chain[key] = append(c.chain[key], s)
		p.Shift(s)
	}
}

//record the frequencies of words
//store the prefix-suffix in the chain map
func (c *Chain) RecordWordFreq(context []string) {
	//prefix is a string, all prefixes can be stored as a slice of string
	//except the first line which donated to N
```

```
	prefixes := make([]string, 0)

	for i := 1; i < len(context)-1; i++ {
		//words are seperated by space
		//skip first line(index 0)
		words := strings.Split(context[i], " ")

		word := make([]string, 0) //simulate the template build() method

		//append number of prefix no more than prefixLen
		for j := 0; j < c.prefixLen; j++ {
			//append targeted word
			word = append(word, words[j])
		}
		prefixes = append(prefixes, strings.Join(word, " "))
	}

	//treat "" as empty word, not prefixes
	for i := 0; i < len(prefixes); i++ {
		prefixes[i] = strings.Replace(prefixes[i], "\"\"", "", -1)
	}

	//suffix is a slice of string, all suffixes can be stored as a slice of
slices of string
	suffixes := make([][]string, len(context))
	//first split words
	for i := 1; i < len(context)-1; i++ {
		//words are seperated by space
		words := strings.Split(context[i], " ")
		//suffixes are slice of string
		suffix := make([]string, 0)
		//except prefixes
		for j := c.prefixLen; j < len(words)-1; j += 2 {
			suffix = append(suffix, words[j])
		}
		//then we find the same suffixes
		for j := c.prefixLen + 1; j < len(words); j++ {
			limit, _ := strconv.Atoi(words[j])
			for k := 0; k < limit-1; k++ {
				suffix = append(suffix, words[j-1])
			}
		}
		suffixes[i-1] = suffix
	}
	//save our results to chian
	for i := 0; i < len(context)-2; i++ {
		c.chain[prefixes[i]] = suffixes[i]
	}
}

func (c *Chain) Write(filename string) {
	//write()function takes input file and save frequency table
	//so first we initialize a frequency table to store results
```

```
	FreqTable := make(map[string]map[string]int)
	//each prefix correspond to several suffix
	//each chain has a map of [prefix]:[]suffixes
	for prefix, _ := range c.chain {
		//make a new map for suffixes
		FreqTable[prefix] = make(map[string]int)
	}
	//now we save suffixes
	//[prefix]:[]suffixes
	for prefix, suffixes := range c.chain {
		//loop over the slice: []suffixes
		for _, suffix := range suffixes {
			FreqTable[prefix][suffix] += 1
		}
	}
	//save the frequency table to output file
	out, err := os.Create(filename)
	if err != nil {
		fmt.Println("ErrorÔ°öcouldn't create", filename)
		os.Exit(1)
	}
	//first line is N
	fmt.Fprintf(out, "%d\n", c.prefixLen)
	//now store prefix and suffixes
	for prefix, suffixes := range FreqTable {
		//treat "" as empty word
		if strings.TrimPrefix(prefix, " ") != prefix {
			//correct prefix, treat " " as string space
			for strings.TrimPrefix(prefix, " ") != prefix {
				fmt.Fprintf(out, "\"\" ")
				prefix = strings.TrimPrefix(prefix, " ")
			}
			//replace "" with string ("")
			//be careful about seperating words by space
			if prefix == "" {
				fmt.Fprintf(out, "\"\"")
			}
			//format output
			fmt.Fprintf(out, "%v ", prefix)
		} else {
			fmt.Fprintf(out, "%v ", prefix)
		}
		// we need to record the suffix and frequency
		for suffix, frequency := range suffixes {
			fmt.Fprintf(out, "%v ", suffix)
			fmt.Fprintf(out, "%d ", frequency)
		}
		fmt.Fprintln(out)
	}
}

// Generate returns a string of at most n words generated from Chain.
func (c *Chain) Generate(n int) string {
	p := make(Prefix, c.prefixLen)
```

```
	var words []string
	for i := 0; i < n; i++ {
		choices := c.chain[p.String()]
		if len(choices) == 0 {
			break
		}
		next := choices[rand.Intn(len(choices))]
		words = append(words, next)
		p.Shift(next)
	}
	return strings.Join(words, " ")
}

func main() {
	// Register command-line flags.
	//I don't understand how flag works so comment them out
	//numWords := flag.Int("words", 100, "maximum number of words to print")
	//prefixLen := flag.Int("prefix", 2, "prefix length in words")
	//we need to read the command line argument based on different command
	if os.Args[1] == "read" {
		//if command is "read", then next argument is N
		//N indicated the number of words to use
		//N should read length>=1,
		prefixLen, err := strconv.Atoi(os.Args[2])
		//first we check if the argument is valid
		if err != nil {
			fmt.Println("Error: prefixLen must be integer")
			os.Exit(1)
		} else if prefixLen <= 0 { //able to handle N>=1
			fmt.Println("Error: prefixLen must be equal to or greater than
1")
			os.Exit(1)
		}
		//mark,read,N,outfilename,infile1...
		outfilename := os.Args[3]

		//now we build a chain
		c := NewChain(prefixLen)
		//read the infile,starts from the 4th to the end
		for i := 4; i < len(os.Args); i++ {
			infile, err := os.Open(os.Args[i])
			if err != nil {
				fmt.Println("Error: something went wrong opening the
file")
				fmt.Println("Probably you gave the wrong filename")
				os.Exit(1)
			}
			//build() will parse the text and stored them in chain
			c.Build(infile)
		}
		//read the input file and save the frequency table to outfilename
		c.Write(outfilename)
	} else if os.Args[1] == "generate" { //another command
		//this will take mark generate modelfile n
```

```
            numWords, err := strconv.Atoi(os.Args[3])
            if err != nil {
                  fmt.Println("Error: numWords must be integer")
                  os.Exit(1)
            } else if numWords < 0 {
                  fmt.Println("Error: numWords must be positive number")
                  os.Exit(1)
            }

            //flag.Parse()                      // Parse command-line flags.
            rand.Seed(time.Now().UnixNano()) // Seed the random number
generator.

            //read the frequency table in the model file to string
            modelfile, err := ioutil.ReadFile(os.Args[2])
            //check if it's a valid file
            if err != nil {
                  fmt.Println(err)
                  os.Exit(1)
            }
            str := string(modelfile)
            //to generate a frequency map, we need to split the file
            context := strings.Split(str, "\n") //split strings line by line
            //based on point 3.4 in the problem, the first line is N, the
prefixLen
            prefixLen, _ := strconv.Atoi(context[0])

            c := NewChain(prefixLen) // Initialize a new Chain.
            c.RecordWordFreq(context)
            //c.Build(os.Stdin)            // Build chains from standard input.
            text := c.Generate(numWords) // Generate text.

            fmt.Println(text) // Write text to standard output.
      }
}
```