```
    1: all: TextGenerator
    2:
    3: TextGenerator: TextGenerator.o MarkovModel.o
    4:         g++ TextGenerator.o MarkovModel.o -o TextGenerator -lboost_unit_test
_framework -g
    5:
    6: TextGenerator.o: TextGenerator.cpp MarkovModel.hpp
    7:         g++ -c TextGenerator.cpp -Wall -Werror -ansi -pedantic -lboost_unit_
test_framework -g
    8:
    9: MarkovModel.o: MarkovModel.cpp MarkovModel.hpp
   10:         g++ -c MarkovModel.cpp -Wall -Werror -ansi -pedantic -lboost_unit_te
st_framework -g
   11:
   12: clean:
   13:         rm *.o TextGenerator *~
```

```
 1: //  Copyright 2015 Zheondre Calcano
 2: //  PS6
 3: #include <map>
 4: #include <exception>
 5: #include <stdexcept>
 6: #include <string>
 7: #include <iostream>
 8: #include <vector>
 9: #include <algorithm>
10: #include "MarkovModel.hpp"
11:
12: int main(int argc, char *argv[]) {
13: std::string a;
14: std::cin >> a;
15: MarkovModel temp(a, 2);
16: temp.gen("ag", 10);
17: std::cout << temp << std::endl;
18: }
```

```
 1: #ifndef _MarkovModel_
 2: #define _MarkovModel_
 3:
 4: //  Copyright 2015 Z'heondre Calcano
 5: //  PS6
 6: /*
 7:  * MarkovModel.hpp
 8:  * Copyright Fred Martin, fredm@cs.uml.edu
 9:  * Tue Apr  7 21:54:53 2015
10:  */
11: #include <exception>
12: #include <stdexcept>
13: #include <string>
14: #include <iostream>
15: #include <vector>
16: #include <algorithm>
17: #include <map>
18:
19: class MarkovModel {
20:   int _order;
21:   std::map< std::string, int> _kgrams;
22:   std::string _alphabet;
23:   std::vector< std::string > _s;
24:  public:
25:   MarkovModel(std::string text, int k);
26:   ~MarkovModel() {
27:     _kgrams.clear();
28:   }
29:   int order();
30:   int freq(std::string kgram);
31:   int freq(std::string kgram, char c);
32:   char randk(std::string kgram);
33:   std::string gen(std::string kgram, int T);
34:   void sets();
35:   void findAmount(std::string x);
36:   void printmap(std::ostream &out);
37:   friend std::ostream& operator<< (std::ostream &out, MarkovModel &mm){
38:     out << mm._order << "\n" << mm._alphabet << "\n";
39:     mm.printmap(out);
40:     return out;
41:   }
42: };
43: #endif
```

```
 1: //  Copyright 2015 Zheondre Calcano
 2: //  PS6
 3: #include <map>
 4: #include <exception>
 5: #include <stdexcept>
 6: #include <string>
 7: #include <iostream>
 8: #include <vector>
 9: #include <algorithm>
10: #include "MarkovModel.hpp"
11:
12: std::string MarkovModel::gen(std::string kgram, int T) {
13:   if (kgram.size() != (unsigned)_order)
14:     throw
15:       std::runtime_error(" Kgram is not of length  k");
16:   if (kgram.size() > (unsigned)T)
17:     throw
18:       std::runtime_error(" T s less than K");
19:   int i;
20:   std::string tempst = kgram;
21:   for (i = 0; i < T; i++)
22:     tempst += randk(kgram);
23:   return tempst;
24: }
25:
26: void MarkovModel::printmap(std::ostream &out) {
27:   for (std::map< std::string, int >::iterator it = _kgrams.begin();
28:       it != _kgrams.end(); it++)
29:     out << it->first << " "<< it->second << "\n";
30: }
31:
32: int MarkovModel::freq(std::string kgram) {
33:   if (kgram.size() != (unsigned)_order)
34:     throw
35:       std::runtime_error("Kgram is not of length  k");
36:   if (kgram.empty())
37:     return _alphabet.size();
38:   return _kgrams[kgram];
39: }
40:
41: int MarkovModel::freq(std::string kgram, char c) {
42:   if (kgram.size() != (unsigned)_order)
43:     throw
44:       std::runtime_error(" Kgram is not of length  k");
45:   std::string kplus1;
46:   kplus1 = kgram;
47:   kgram.push_back(c);
48:   return _kgrams[kgram];
49: }
50:
51: char MarkovModel::randk(std::string kgram) {
52:   int stsize = (unsigned)_s.size();
53:   if (kgram.size() != (unsigned)_order)
54:     throw
55:       std::runtime_error(" Kgram is not of length  k");
56:   if (_kgrams.find(kgram) == _kgrams.end())
57:     throw
58:       std::runtime_error(" Kgram doesn't exist in map");
59:   std::string check; std::vector< char > pbvc;
60:   int amount, i, totalfreq, j, randpos;
61:   totalfreq = amount = 0;
```

```
   62:   for (std::map< std::string, int>::iterator it = _kgrams.begin();
   63:        it != _kgrams.end(); ++it) {
   64:     if (it->first == kgram) {
   65:       for (i = 0; i < stsize; i++) {
   66:         check = kgram + _s[i];
   67:         amount = _kgrams[check];
   68:         totalfreq += amount;
   69:         for (j = 0; j < amount; j++)
   70:           pbvc.push_back(check[check.size()-1]);
   71:       }
   72:       randpos = rand() % totalfreq; //NOLINT
   73:       return pbvc[randpos];
   74:     }
   75:   }
   76:   return 0;
   77: }
   78:
   79: void MarkovModel::findAmount(std::string x) {
   80:   if (_kgrams.find(x) == _kgrams.end()) {
   81:     _kgrams[x] = 1;  // not found
   82:   } else {
   83:     _kgrams[x] += 1;  // found
   84:   }
   85: }
   86:
   87: int MarkovModel::order() {
   88:   return _order;
   89: }
   90:
   91: void MarkovModel::sets() {
   92:   int i, j, duplicate, stsize; std::string a, b, temp;
   93:   stsize = (unsigned)_s.size();
   94:   temp = _alphabet;
   95:   int sizeoftemp = temp.size();
   96:   for (i = 0; i < sizeoftemp; i++) {
   97:     duplicate = 0;
   98:     a = temp.substr(i, 1);
   99:     for (j = 0; j < stsize; j++) {
  100:       b = _s[j];
  101:       if (a == b)
  102:         duplicate = 1;
  103:     }
  104:     if (duplicate == 0) {
  105:       _s.push_back(a);
  106:       duplicate = 0;
  107:     }
  108:   }
  109: }
  110:
  111: MarkovModel::MarkovModel(std::string input, int k) {
  112:   int i, kk, pos, szofinpt; std::string x;
  113:   szofinpt = (unsigned)input.size();
  114:   _alphabet = input;
  115:   _order = k;
  116:   if (k == 0) {
  117:     for (i = 0; i < szofinpt; i++)
  118:       findAmount(input.substr(i, 1));
  119:   } else {
  120:     kk = k;
  121:     pos = szofinpt - k;
  122:     for (i = 0; i < szofinpt-k; i++) {
```

```
  123:       findAmount(input.substr(i, k));
  124:       findAmount(input.substr(i, k+1));
  125:     }
  126:     int upstlen = 0;
  127:     while (kk > 0) {
  128:       findAmount(input.substr(pos, kk) + input.substr(0, upstlen));
  129:       upstlen++;
  130:       findAmount(input.substr(pos, kk) + input.substr(0, upstlen));
  131:       kk--; pos++;
  132:     }
  133:     std::string check;
  134:     sets();
  135:     int sof_s = (unsigned)_s.size();
  136:     for (std::map< std::string, int>::iterator it = _kgrams.begin();
  137:          it != _kgrams.end(); ++it) {
  138:       if ((unsigned)it->first.size() == (unsigned)k) {
  139:         x = it->first;
  140:         for (i = 0; i < sof_s; i++) {
  141:           check = it->first + _s[i];
  142:           if (_kgrams.find(check) == _kgrams.end()) {
  143:              _kgrams[check] = 0;
  144:           }
  145:         }
  146:       }
  147:     }
  148:   }
  149: }
```

```
 1: // Angel Zheondre Calcano
 2: // PS6
 3:
 4: #include <iostream>
 5: #include <string>
 6: #include <exception>
 7: #include <stdexcept>
 8:
 9: #include "MarkovModel.hpp"
10:
11: #define BOOST_TEST_DYN_LINK
12: #define BOOST_TEST_MODULE Main
13: #include <boost/test/unit_test.hpp>
14:
15: using namespace std;
16:
17: BOOST_AUTO_TEST_CASE(order0) {
18:   // normal constructor
19:   BOOST_REQUIRE_NO_THROW(MarkovModel("gagggagaggcgagaaa", 0));
20:
21:   MarkovModel mm("gagggagaggcgagaaa", 0);
22:
23:   BOOST_REQUIRE(mm.order() == 0);
24:   BOOST_REQUIRE(mm.freq("") == 17); // length of input in constructor
25:   BOOST_REQUIRE_THROW(mm.freq("x"), std::runtime_error);
26:
27:   BOOST_REQUIRE(mm.freq("", 'g') == 9);
28:   BOOST_REQUIRE(mm.freq("", 'a') == 7);
29:   BOOST_REQUIRE(mm.freq("", 'c') == 1);
30:   BOOST_REQUIRE(mm.freq("", 'x') == 0);
31:
32: }
33:
34: BOOST_AUTO_TEST_CASE(order1) {
35:   // normal constructor
36:   BOOST_REQUIRE_NO_THROW(MarkovModel("gagggagaggcgagaaa", 1));
37:
38:   MarkovModel mm("gagggagaggcgagaaa", 1);
39:
40:   BOOST_REQUIRE(mm.order() == 1);
41:   BOOST_REQUIRE_THROW(mm.freq(""), std::runtime_error);
42:   BOOST_REQUIRE_THROW(mm.freq("xx"), std::runtime_error);
43:
44:   BOOST_REQUIRE(mm.freq("a") == 7);
45:   BOOST_REQUIRE(mm.freq("g") == 9);
46:   BOOST_REQUIRE(mm.freq("c") == 1);
47:
48:   BOOST_REQUIRE(mm.freq("a", 'a') == 2);
49:   BOOST_REQUIRE(mm.freq("a", 'c') == 0);
50:   BOOST_REQUIRE(mm.freq("a", 'g') == 5);
51:
52:   BOOST_REQUIRE(mm.freq("c", 'a') == 0);
53:   BOOST_REQUIRE(mm.freq("c", 'c') == 0);
54:   BOOST_REQUIRE(mm.freq("c", 'g') == 1);
55:
56:   BOOST_REQUIRE(mm.freq("g", 'a') == 5);
57:   BOOST_REQUIRE(mm.freq("g", 'c') == 1);
58:   BOOST_REQUIRE(mm.freq("g", 'g') == 3);
59:
60:   BOOST_REQUIRE_NO_THROW(mm.randk("a"));
61:   BOOST_REQUIRE_NO_THROW(mm.randk("c"));
```

```
   62:   BOOST_REQUIRE_NO_THROW(mm.randk("g"));
   63:
   64:   BOOST_REQUIRE_THROW(mm.randk("x"), std::runtime_error);
   65:
   66:   BOOST_REQUIRE_THROW(mm.randk("xx"), std::runtime_error);
   67:
   68: }
   69:
   70: BOOST_AUTO_TEST_CASE(order2) {
   71:   // normal constructor
   72:   BOOST_REQUIRE_NO_THROW(MarkovModel("gagggagaggcgagaaa", 2));
   73:
   74:   MarkovModel mm("gagggagaggcgagaaa", 2);
   75:
   76:   BOOST_REQUIRE(mm.order() == 2);
   77:
   78:   BOOST_REQUIRE_THROW(mm.freq(""), std::runtime_error);
   79:   BOOST_REQUIRE_THROW(mm.freq("x"), std::runtime_error);
   80:   BOOST_REQUIRE_NO_THROW(mm.freq("xx"));
   81:   BOOST_REQUIRE_THROW(mm.freq("", 'g'), std::runtime_error); // kgram is wro
ng length
   82:   BOOST_REQUIRE_THROW(mm.freq("x", 'g'), std::runtime_error); // kgram is wr
ong length
   83:   BOOST_REQUIRE_THROW(mm.freq("xxx", 'g'), std::runtime_error); // kgram is
wrong length
   84:
   85:
   86:   BOOST_REQUIRE(mm.freq("aa") == 2);
   87:   BOOST_REQUIRE(mm.freq("aa", 'a') == 1);
   88:   BOOST_REQUIRE(mm.freq("aa", 'c') == 0);
   89:   BOOST_REQUIRE(mm.freq("aa", 'g') == 1);
   90:
   91:   BOOST_REQUIRE(mm.freq("ag") == 5);
   92:   BOOST_REQUIRE(mm.freq("ag", 'a') == 3);
   93:   BOOST_REQUIRE(mm.freq("ag", 'c') == 0);
   94:   BOOST_REQUIRE(mm.freq("ag", 'g') == 2);
   95:
   96:   BOOST_REQUIRE(mm.freq("cg") == 1);
   97:   BOOST_REQUIRE(mm.freq("cg", 'a') == 1);
   98:   BOOST_REQUIRE(mm.freq("cg", 'c') == 0);
   99:   BOOST_REQUIRE(mm.freq("cg", 'g') == 0);
  100:
  101:   BOOST_REQUIRE(mm.freq("ga") == 5);
  102:   BOOST_REQUIRE(mm.freq("ga", 'a') == 1);
  103:   BOOST_REQUIRE(mm.freq("ga", 'c') == 0);
  104:   BOOST_REQUIRE(mm.freq("ga", 'g') == 4);
  105:
  106:   BOOST_REQUIRE(mm.freq("gc") == 1);
  107:   BOOST_REQUIRE(mm.freq("gc", 'a') == 0);
  108:   BOOST_REQUIRE(mm.freq("gc", 'c') == 0);
  109:   BOOST_REQUIRE(mm.freq("gc", 'g') == 1);
  110:
  111:   BOOST_REQUIRE(mm.freq("gg") == 3);
  112:   BOOST_REQUIRE(mm.freq("gg", 'a') == 1);
  113:   BOOST_REQUIRE(mm.freq("gg", 'c') == 1);
  114:   BOOST_REQUIRE(mm.freq("gg", 'g') == 1);
  115:
  116: }
```