Teoria Informacji i Kodowania

Implementacja kodera/dekodera słownikowego. Metoda: LZ77

Arkadiusz Ostrzyżek WCY22KY2S1

Podejście demonstracyjne

Analiza zadania

Algorytm skanuje dane i szuka wcześniejszych wystąpień ciągów bajtów, zastępując je odniesieniami do wcześniejszych pozycji w tekście. Dzięki temu, że odniesienia zajmują mniej miejsca niż oryginalne ciągi, algorytm efektywnie redukuje wielkość danych, zachowując przy tym ich pełną treść.

Na zajęciach oraz w książce został on zademonstrowany przy użyciu liter separowanych kreskami. Z tego powodu na początku zdecydowałem się na automatyzację tego procesu, aby móc zobaczyć graficznie działanie tego szyfrowania na dowolnym pliku.

Założenia

Zakładam, że program zostanie użyty na dowolnym pliku składających się z podstawowych char'ów. W pliku po zakodowaniu nie zajdzie kompresja, ze względu na sposób zapisu do niego. Plik nie będzie większy niż 1 MB.

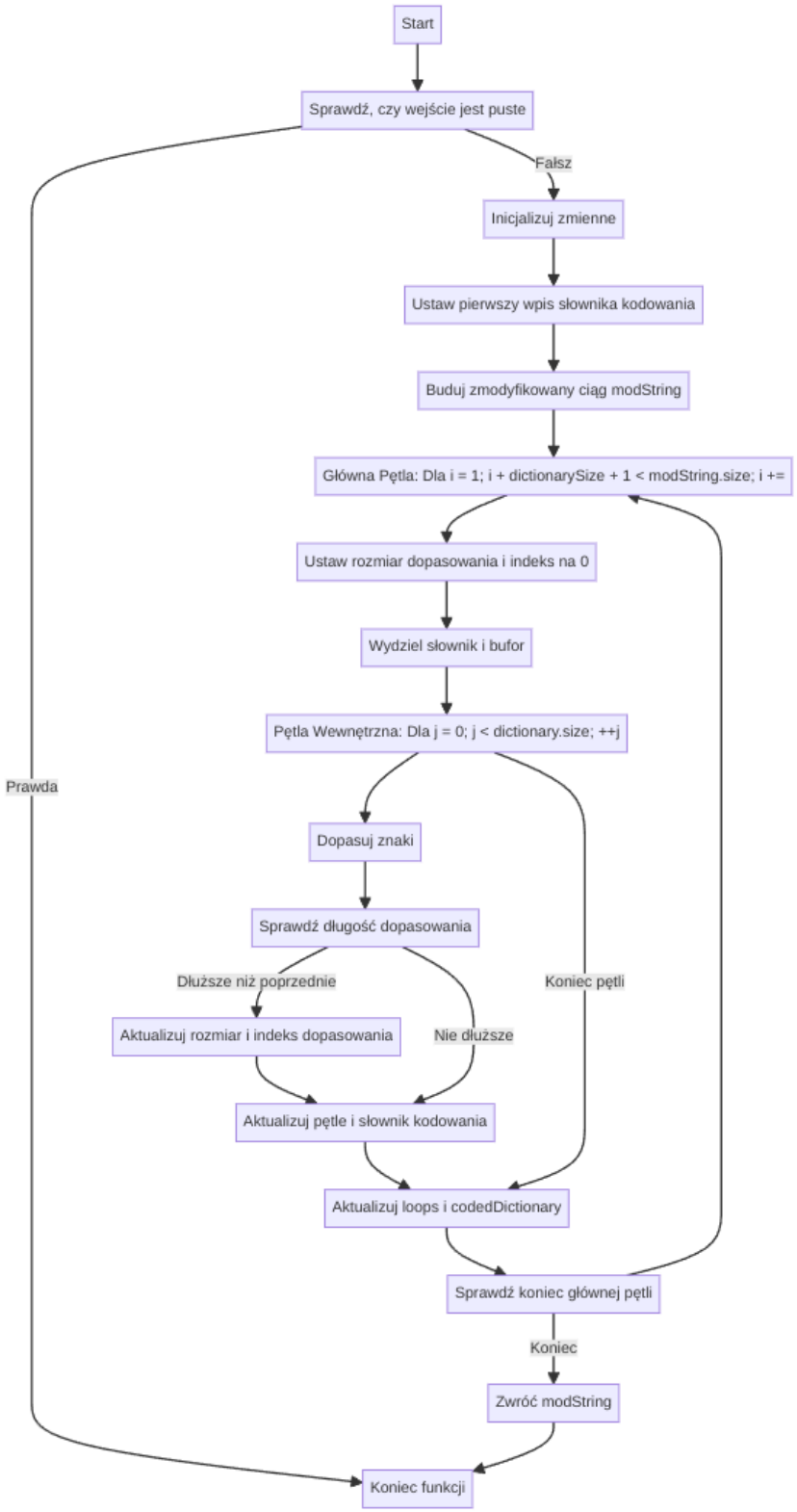
Algorytm

Kodowanie

Opis działania

Program wczytuje całość pliku do stringa. Następnie rozpoczynane jest jego szyfrowanie, a więc słownik jest wypełniany pierwszym znakiem, po czym porównujemy ciągi znaków. Najdłuższy pokrywający się string jest zapisywany zgodnie ze sposobem kodowania LW77. Następnie, słownik i buffor przesuwają. Proces powtarza się aż osiągniemy koniec pliku.

Flowchart


Start
Sprawdź, czy wejście jest puste
Prawda
Fałsz
Inicjalizuj zmienne
Ustaw pierwszy wpis słownika kodowania
Buduj zmodyfikowany ciąg modString
Główna Pętla: Dla i = 1; i + dictionarySize + 1 < modString.size; i +=
Ustaw rozmiar dopasowania i indeks na 0
Wydziel słownik i bufor
Pętla Wewnętrzna: Dla j = 0; j < dictionary.size; ++j
Dopasuj znaki
Sprawdź długość dopasowania
Dłuższe niż poprzednie
Aktualizuj rozmiar i indeks dopasowania
Nie dłuższe
Koniec pętli
Aktualizuj pętle i słownik kodowania
Aktualizuj loops i codedDictionary
Sprawdź koniec głównej pętli
Koniec
Zwróć modString
Koniec funkcji

Kod

```
1  string dictionaryCoding(string* codedDictionary, const string&
       input) {
2
3      if (input.empty()) return "";
4
5      int currentMatching, matchingSize, matchingIndex, loops =
           0;
6      string modString, dictionary, buffer;
7
8      codedDictionary[0] =     to_string(0) + "|" +
9                                   to_string(0) + "|" +
10                                  input[0];
11
12     for (int i = 0; i < dictionarySize; i++) {
13         modString += input[0];
14     }
15
16     modString += input;
17
18     for (int i = 1; i + dictionarySize + 1 < modString.size();
           i += (matchingSize + 1)) {
19
20         matchingSize = 0;
21         matchingIndex = 0;
22
23         dictionary = modString.substr(i, dictionarySize);
24         buffer = modString.substr(i + dictionarySize,
               bufferSize);
25
26         for (int j = 0; j < dictionary.size(); j++) {
27
28             currentMatching = 0;
29             while    (dictionary[j + currentMatching] == buffer[
                   currentMatching] &&
30                       dictionary[j + currentMatching] != '\0' &&
31                       buffer[currentMatching] != '\0') {
32                 currentMatching++;
33             }
34
35             if (matchingSize < currentMatching) {
36                 matchingSize = currentMatching;
37                 matchingIndex = j;
```

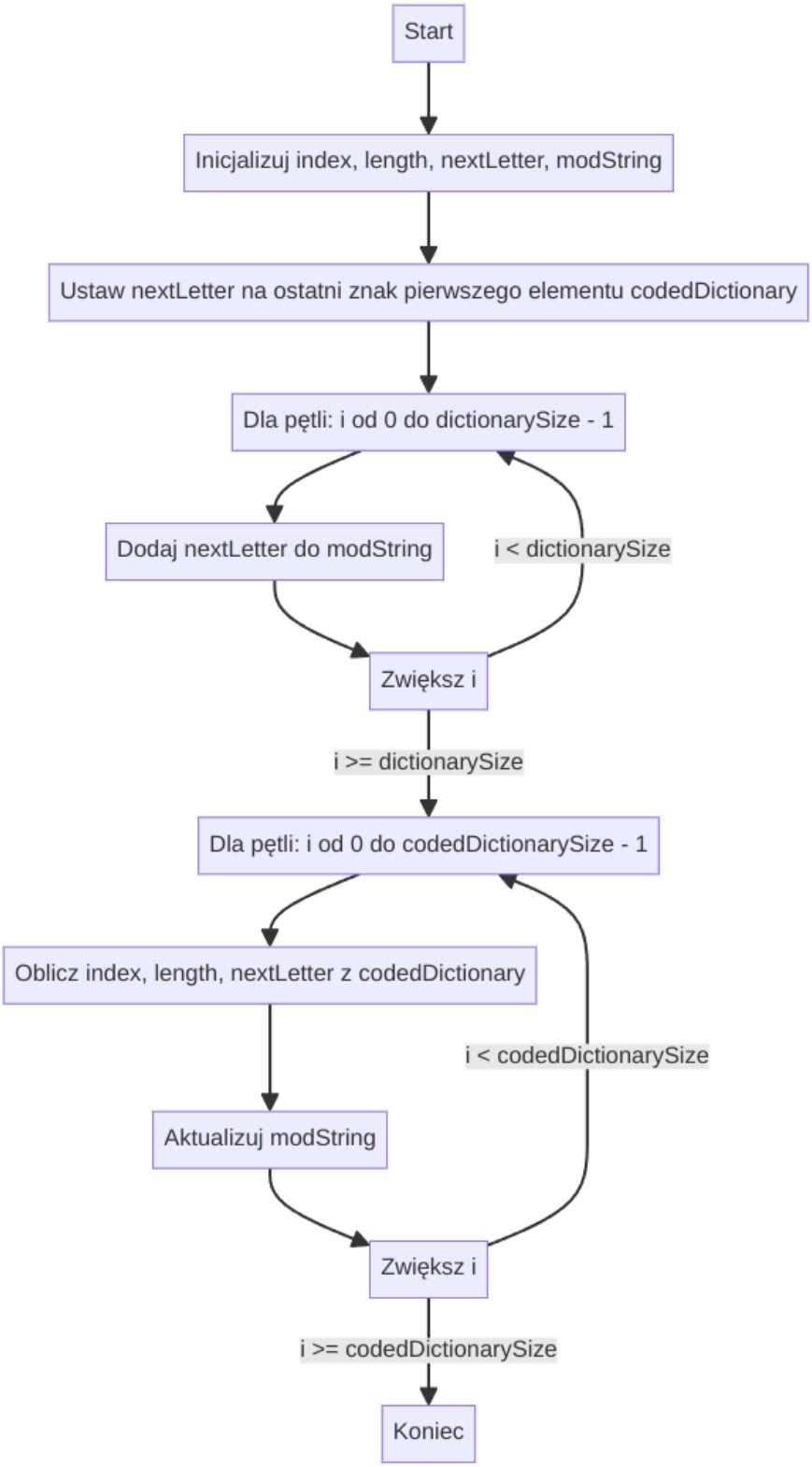
```
38                  }
39              }
40
41              loops += 1;
42          }
43          return modString;
44      }
```

Dekodowanie

Opis działania

Ze względu na sposób przetrzymywania danych w pliku, jesteśmy w stanie odtworzyć operacje dekodowania dokładnie w ten sam sposób co na papierze. W trakcie tego procesu, dekoder czyta zakodowane odniesienia, które zawierają informacje o pozycji i długości wcześniej występujących ciągów danych wraz z ewentualnym nowym bajtem danych. Następnie, wykorzystując te odniesienia, dekoder odbudowuje oryginalne ciągi bajtów, kopiując je z wcześniejszych fragmentów już zdekodowanych danych, a w przypadku nowych bajtów, dodaje je bezpośrednio

Flowchart


Start
Inicjalizuj index, length, nextLetter, modString
Ustaw nextLetter na ostatni znak pierwszego elementu codedDictionary
Dla pętli: i od 0 do dictionarySize - 1
Dodaj nextLetter do modString
i < dictionarySize
Zwiększ i
i >= dictionarySize
Dla pętli: i od 0 do codedDictionarySize - 1
Oblicz index, length, nextLetter z codedDictionary
i < codedDictionarySize
Aktualizuj modString
Zwiększ i
i >= codedDictionarySize
Koniec

Kod

```
string dictionaryDecoding(const string* codedDictionary) {

    int index, length;
    char nextLetter;
    string modString;

    nextLetter = codedDictionary[0].substr(codedDictionary[0].
       find_last_of('|') + 1, 1)[0];

    for (int i = 0; i < dictionarySize; i++) {
        modString += nextLetter;
    }

    for (int i = 0; i < codedDictionarySize; i++) {

        if (codedDictionary[i].empty()) {
            continue;
        }

        index = stoi(codedDictionary[i].substr(0,
           codedDictionary[i].find('|')));
        length = stoi(codedDictionary[i].substr(codedDictionary
           [i].find('|') + 1,
                                                codedDictionary
                                                   [i].
                                                   find_last_of
                                                   ('|') -
                                                codedDictionary
                                                [i].find('|'
                                                ) - 1));
        nextLetter = codedDictionary[i].substr(codedDictionary[
           i].find_last_of('|') + 1, 1)[0];
        modString += modString.substr(modString.size() -
           dictionarySize + index, length) + nextLetter;
    }
}
```

Testowanie

Przykład zakodowanego pliku

```
0|0|a
0|50|b
998|2|s
996|1|b
998|2|b
995|5|b
989|10|a
977|1|d
974|3|d
969|3|d
988|4|h
989|2|s
0|0|j
0|0|f
994|2|g
995|1|s
989|1|l
974|2|f
992|1|h
972|2|f
995|2|s
995|4|b
958|1|;
977|1|b
0|0|n
0|0|o
0|0|u
```

Jeżeli plik będzie się składać z mocno poważających się ciągów słów, nawet używając tego sposobu zapisu będziemy w stanie znacząco zmniejszyć plik. Najgorsze osiągi będą jednak dla zupełnie losowego pliku. Przy początkowych pomiarach ustawiłem wielkości słownika na 1000, bufora na 200.

Po zakodowaniu pliku który składał się ze zbliżonych do siebie ciągów znaków, jesteśmy w stanie zmniejszyć jego wielkość z 15 kB do 8.2 kB.

Dla zupełnie losowego pliku, nasz zakodowany plik zwiększył swoją wielkość z 10 kB do 33 kB. Plik został wygenerowany używając poniższej komendy.

```
1 tr -dc "A-Za-z 0-9" < /dev/urandom | fold -w100|head -n 100 >
    input.txt
```

Oczywiście wielkość zakodowanego pliku jest także zależna od wielkości słownika oraz buforu. Jeżeli zwiększymy jego wielkość z 1000 do 10000, to wielkość pliku będzie wynosić 2,9 kB. Jeżeli zwiększymy wielkość bufora z 200 do 800, to wielkość pliku będzie wynosić 8,1 kB. Jeżeli zwiększymy słownik i bufor, to plik będzie miał wielkość 2,6 kB.

Wnioski

Kompresja LZ77 jest bardzo efektywna dla plików z często powtarzającymi się wzorami. Gdy jednak plik składa się z wielu nowych znaków oraz nie zawierają się w nim żadne wzorce, będzie ona miała znikome skutki, lub będzie nawet prowadzić do utworzenia większego pliku niż orginał.

Podejście praktyczne

Założenia

Ta wersja będzie w stanie efektywnie kompresować wszystkie pliki, nie będzie ograniczona także przez wielkość pliku. Dane zostaną zakodowane binarnie, a więc nie będzie możliwe odczytanie ich gołym okiem.

Algorytm

Algorytm działa w ten sam sposób co poprzednio, jednak przy kodowaniu program nie wczytuje danych bezpośrednio, tylko ciągle czyta plik i porównuje zawarte w nim bajty. Ze względu na to, powinien on działać dla plików o dowolnej wielkości.

Testowanie

W ramach testów została zakodowana Berean's Reader's Bible . Orginalnie ma ona 3.9 MB, jednak po zakodowaniu zmniejsza się ona do 2.2 MB. Tak dobra kompresja jest spowodowana często powtarzającymi się zwrotami zawartymi w tym tekście.

Dla losowego pliku o wielkości 10 MB utworzonego tą samą komendą co ostatnio, zakodowany plik osiąga wielkość 15,7 MB, co było oczekiwane.

Wnioski

Kodowanie LZ77 można zaimplementować w sposób, który nie wymaga wczytywania większych ilości danych na raz oraz można kodować go zapisując bajty do pliku.

Kod źródłowy

Program demonstracyjny

```
1
2  #include <iostream>
3  #include <string>
4  #include <fstream>
5  #include <sstream>
6  #include <bitset>
7
8  using namespace std;
9
10 constexpr int dictionarySize = 10000;
11 constexpr int bufferSize = 2000;
12 constexpr int codedDictionarySize = 190000;
13
14 void printState(const string& dictionary, const string& buffer,
       const int& loops) {
15     cout << " /// STATE " << loops << " ///" << endl;
16     cout << "Dictionary: " << dictionary << endl;
17     cout << "Buffer: " << buffer << endl;
18 }
19 void printCode(const int& index, const int& length, char
      nextLetter) {
20     cout << "<" << index << ", " << length << ", " <<
          nextLetter << ">" << endl;
21 }
22 void printCodedDictionary(const string* codedDictionary) {
23     for (int i = 0; i < codedDictionarySize; i++) {
24
25         if (codedDictionary[i] == "\n") {
26             cout << "\n";
27             continue;
28         }
```

```
29
30          if (codedDictionary[i].empty()) {
31              continue;
32          }
33
34          cout << codedDictionary[i] << endl;
35
36      }
37  }
38
39  string readFileIntoString(const string& filename) {
40      ifstream file(filename);
41      stringstream buffer;
42      buffer << file.rdbuf();
43      return buffer.str();
44  }
45  void writeStringToFile(const string& filename, const string&
        str){
46
47      ofstream outputFile;
48      outputFile.open(filename);
49
50      if(outputFile.is_open()){
51          outputFile << str;
52          outputFile.close();
53      } else {
54          cout << "Unable to open file";
55      }
56  }
57
58  void saveToCodedFile(const string& filename, const string*
        codedDictionary) {
59      ofstream file;
60      file.open(filename);
61      for (int i = 0; i < codedDictionarySize; i++) {
62          if (!codedDictionary[i].empty()) {
63              file << codedDictionary[i] << endl;
64          } else {
65              break;
66          }
67      }
68      file.close();
69  }
```

```
70 void readFromCodedFile(const string& filename, string*
       codedDictionary) {
71     ifstream file;
72     file.open(filename);
73     string line;
74     int i = 0;
75     while (getline(file, line)) {
76         codedDictionary[i] = line;
77         i++;
78     }
79     file.close();
80 }
81
82 string dictionaryCoding(string* codedDictionary, const string&
       input) {
83
84     if (input.empty()) return "";
85
86     int currentMatching, matchingSize, matchingIndex, loops =
           0;
87     string modString, dictionary, buffer;
88     codedDictionary[0] =     to_string(0) + "|" +
89                                  to_string(0) + "|" +
90                                  input[0];
91
92     for (int i = 0; i < dictionarySize; i++) {
93         modString += input[0];
94     }
95
96     modString += input;
97
98     for (int i = 1; i + dictionarySize + 1 < modString.size();
           i += (matchingSize + 1)) {
99
100        matchingSize = 0;
101        matchingIndex = 0;
102
103        dictionary = modString.substr(i, dictionarySize);
104        buffer = modString.substr(i + dictionarySize,
               bufferSize);
105
106        for (int j = 0; j < dictionary.size(); j++) {
107
108            // sprawdza łśćdugo dopasowania
```

```
109             currentMatching = 0;
110             while   (dictionary[j + currentMatching] == buffer[
                    currentMatching] &&
111                     dictionary[j + currentMatching] != '\0' &&
112                     buffer[currentMatching] != '\0') {
113                 currentMatching++;
114             }
115
116             // sprawdza, czy łśćdugo dopasowania jest ęwiksza od
                    poprzedniej
117             if (matchingSize < currentMatching) {
118                 matchingSize = currentMatching;
119                 matchingIndex = j;
120             }
121         }
122
123         loops += 1;
124     }
125     return modString;
126 }
127 string dictionaryDecoding(const string* codedDictionary) {
128
129     int index, length;
130     char nextLetter;
131     string modString;
132
133     nextLetter = codedDictionary[0].substr(codedDictionary[0].
            find_last_of('|') + 1, 1)[0];
134
135     for (int i = 0; i < dictionarySize; i++) {
136         modString += nextLetter;
137     }
138
139     for (int i = 0; i < codedDictionarySize; i++) {
140
141         if (codedDictionary[i].empty()) {
142             continue;
143         }
144
145         index = stoi(codedDictionary[i].substr(0,
                codedDictionary[i].find('|')));
146         length = stoi(codedDictionary[i].substr(codedDictionary
                [i].find('|') + 1,
```

```
147                                                     codedDictionary
                                                            [i].
                                                            find_last_of
                                                            ('|') -
                                                            codedDictionary
                                                            [i].find('|'
                                                            ) - 1));
148             nextLetter = codedDictionary[i].substr(codedDictionary[
                    i].find_last_of('|') + 1, 1)[0];
149             modString += modString.substr(modString.size() -
                    dictionarySize + index, length) + nextLetter;
150         }
151
152         // zamiana znaków '\0' na '\n'
153         size_t pos = 0;
154         while ((pos = modString.find('\0', pos)) != string::npos) {
155             modString.replace(pos, 1, "\n");
156             pos += 1;
157         }
158
159         return modString.substr(dictionarySize, modString.size() -
                dictionarySize);
160 }
161
162 void codeFile(const string& filename1, const string& filename2)
        {
163         string codedDictionary[codedDictionarySize];
164         string input = readFileIntoString(filename1);
165         string output = dictionaryCoding(codedDictionary, input);
166         saveToCodedFile(filename2, codedDictionary);
167 }
168 void decodeFile(const string& filename1, const string&
        filename2) {
169         string codedDictionary[codedDictionarySize];
170         readFromCodedFile(filename1, codedDictionary);
171         string output = dictionaryDecoding(codedDictionary);
172         writeStringToFile(filename2, output);
173 }
174
175 int main() {
176
177         codeFile("test.txt", "coded.txt");
178         decodeFile("coded.txt", "decoded.txt");
179
```

```
180     return 0;
181 }
```

Program praktyczny

```
1  #include <iostream>
2  #include <fstream>
3  #include <vector>
4
5  // max dictionary and buffer size is 2^16 = 65536
6  #define DICTIONARY_SIZE 65536
7  #define BUFFER_SIZE 65536
8
9  using namespace std;
10
11 ifstream::pos_type getFileSize(const char* filename) {
12     ifstream in(filename, ifstream::ate | ifstream::binary);
13     return in.tellg();
14 }
15
16 void lz77coding(const string& input, const string& output) {
17     ifstream inputFile(input, ios::binary);
18     ofstream outputFile(output, ios::binary);
19
20     if (!inputFile.is_open()) {
21         cerr << "Failed to open file: " << input << endl;
22         exit(EXIT_FAILURE);
23     }
24
25     vector<char> fileData((istreambuf_iterator<char>(inputFile)
           ), istreambuf_iterator<char>());
26     long fileSize = fileData.size();
27     int k = 0;
28
29     while (k < fileSize) {
30         int distance = 0;
31         int length = 0;
32         char nextChar = k + length < fileSize ? fileData[k +
               length] : 0;
33
34         for (int i = 1; i <= min(k, DICTIONARY_SIZE); i++) {
35             int matchLength = 0;
36             int dictPos = k - i;
37             int bufferPos = k;
```

```
38
39              while (matchLength < BUFFER_SIZE && bufferPos <
                    fileSize) {
40                  if (fileData[dictPos + matchLength] != fileData
                        [bufferPos + matchLength]) {
41                      break;
42                  }
43                  matchLength++;
44              }
45
46              if (matchLength > length) {
47                  length = matchLength;
48                  distance = i;
49                  nextChar = bufferPos + matchLength < fileSize ?
                        fileData[bufferPos + matchLength] : 0;
50              }
51          }
52
53          auto distance16 = static_cast<uint16_t>(distance);
54          auto length16 = static_cast<uint16_t>(length);
55
56          outputFile.write((char *)&distance16, sizeof(distance16
                ));
57          outputFile.write((char *)&length16, sizeof(length16));
58          outputFile.write(&nextChar, sizeof(nextChar));
59
60          k += length + 1;
61      }
62
63      outputFile.flush();
64      inputFile.close();
65      outputFile.close();
66  }
67
68  void lz77decoding(const string& input, const string& output) {
69      ifstream inputFile(input, ios::binary);
70      ofstream outputFile(output, ios::binary);
71
72      if (!inputFile.is_open()) {
73          cerr << "Failed to open file: " << input << endl;
74          exit(EXIT_FAILURE);
75      }
76
77      vector<char> text;
```

```
78      uint16_t distance, length;
79      char nextChar;
80
81      while (inputFile.read((char *)&distance, sizeof(distance))
           &&
82              inputFile.read((char *)&length, sizeof(length)) &&
83              inputFile.read(&nextChar, sizeof(nextChar))) {
84
85          if (distance == 0 && length == 0) {
86              text.push_back(nextChar);
87          } else {
88              int start = text.size() - distance;
89              for (int i = 0; i < length; ++i) {
90                  if (start + i >= text.size()) {
91                      text.push_back(0);
92                  } else {
93                      text.push_back(text[start + i]);
94                  }
95              }
96              text.push_back(nextChar);
97          }
98      }
99
100
101     outputFile.write(text.data(), text.size());
102
103     outputFile.flush();
104     inputFile.close();
105     outputFile.close();
106 }
107
108 int main() {
109
110     string input    =   "input.txt";
111     string encoded  =   "encoded.bin";
112     string decoded  =   "decoded.txt";
113
114     lz77coding(input, encoded);
115     lz77decoding(encoded, decoded);
116
117     return 0;
118 }
```