Exercice : Code de Huffman

```cpp
#include <iostream>
#include <utility>
#include <vector>
#include <string>
using namespace std;

// ============================================================================
struct Mot_de_code {
  string entree;
  double proba;
  string code;
};

// ============================================================================
typedef vector<Mot_de_code> Code;
/* Note : les vector ne sont pas la bonne structure de données pour faire
 *   cela (associer une info à un mot en entrée) car la recherche y
 *   est linéraire.
 *   Il vaudrait mieux utiliser des hash-tables.
 *   Mais celles-ci n'ont pas encore été présentées en cours (arrivent à
 *   la fin du 2e semestre).
 */

// ============================================================================
void affiche_code(Code const& c) {
  for (auto el : c) {
    cout << '"' << el.entree << "\" --> \"" << el.code << "\" (" << el.proba << ')' << e
  }
}

/* ============================================================================
 * ajoute un mot au code ou incrémente son compte si déjà présent.
 */
void add(string const& mot, Code& code) {
  // recherche le mot
  for (auto& el : code) {
    if (el.entree == mot) {
      // le mot y est déjà : on incrémente son compte et on quitte
      ++el.proba;
      return;
    }
  }

  // On n'a pas trouvé le mot => on l'ajoute (avec un compte de 1)
  code.push_back({ mot, 1.0, "" });
}

// ============================================================================
void normalise(Code& code) {
  double somme(0.0);
  for (auto el : code) {
    somme += el.proba;
  }
  if (somme > 0.0) {
    for (auto& el : code) {
      el.proba /= somme;
    }
  }
}

// ============================================================================
Code calcule_probas(string const& texte, size_t tranche = 1)
{
  Code code;
  for (size_t i(0); i < texte.size(); i += tranche) {
```

```
    string mot(texte.substr(i, tranche));

    // remplir d'espaces à la fin si nécessaire
    while (mot.size() < tranche) mot += ' ';

    add(mot, code);
  }

  // normalise les probabilités
  normalise(code);

  return code;
}

// ============================================================================
struct Entite {
  vector<size_t> indices;
  double proba;
};

// ============================================================================
Entite fusionne(Entite const& L1, Entite const& L2) {
  Entite L;
  L.proba = L1.proba + L2.proba;
  size_t i(0);
  size_t j(0);
  while ((i < L1.indices.size()) and (j < L2.indices.size())) {
    if (L1.indices[i] < L2.indices[j]) {
      L.indices.push_back(L1.indices[i]);
      ++i;
    } else {
      L.indices.push_back(L2.indices[j]);
      ++j;
    }
  }
  while (i < L1.indices.size()) {
    L.indices.push_back(L1.indices[i]);
    ++i;
  }
  while (j < L2.indices.size()) {
    L.indices.push_back(L2.indices[j]);
    ++j;
  }
  return L;
}

// ============================================================================
vector<Entite> listes_initiales(Code const& c) {
  vector<Entite> v(c.size());

  for (size_t i(0); i < c.size(); ++i) {
    v[i].indices.push_back(i);
    v[i].proba = c[i].proba;
  }
  return v;
}

// ============================================================================
void deux_moins_probables(vector<Entite> tab, size_t& prems, size_t& deuz)
{
  if (tab.size() < 2) return;
  double min1(tab[0].proba); prems = 0;
  double min2(tab[1].proba); deuz  = 1;
  if (min1 > min2) {
    swap(min1, min2);
    swap(prems, deuz);
  }

  for (size_t i(2); i < tab.size(); ++i) {
```

```
    if (tab[i].proba < min1) {
      min2 = min1;
      deuz = prems;
      min1 = tab[i].proba;
      prems = i;
    } else if (tab[i].proba < min2) {
      min2 = tab[i].proba;
      deuz = i;
    }
  }
}

// ============================================================================
void ajoute_symbole(char bit, Code& c, Entite l)
{
  for(auto i : l.indices) {
    c[i].code = bit + c[i].code;
  }
}

// ============================================================================
void Huffman(Code& c)
{
  vector<Entite> arbre(listes_initiales(c));

  while (arbre.size() > 1) {
    size_t max_1(0), max_2(0);

    // recherche les deux moins probables
    deux_moins_probables(arbre, max_1, max_2);

    // ajoute respectivement 0 et 1 à leur code
    ajoute_symbole('0', c, arbre[max_1]);
    ajoute_symbole('1', c, arbre[max_2]);

    // les fusionne (en écrasant max_1)
    arbre[max_1] = fusionne(arbre[max_1], arbre[max_2]);

    // et réduit l'arbre (= « monte » dans l'arbre) en supprimant max_2
    arbre[max_2] = arbre.back();
    arbre.pop_back();
  }
}

// ============================================================================
string recherche_entree(string const& texte, size_t& pos, Code const& c)
{
  for (auto el : c) {
    size_t i(0), j(pos);
    while ((i < el.entree.size()) and (j < texte.size()) and (el.entree[i] == texte[j]))
      ++i; ++j;
    }
    if ((i == el.entree.size()) and (j <= texte.size())) { // match with el
      pos = j; // update pos to point to the rest of texte
      return el.code;
    }
    // special case for end of texte
    if ((j == texte.size()) and (el.entree[i] == ' ')) {
      while ((i < el.entree.size()) and (el.entree[i] == ' ')) ++i;
      if (i == el.entree.size()) { // match with el (ending with whitespaces
        pos = j;
        return el.code;
      }
    }
  }
  return string();
}

// ============================================================================
```

```
string recherche_mot_de_code(string const& texte, size_t& pos, Code const& c)
{
  for (auto el : c) {
    size_t i(0), j(pos);
    while ((i < el.code.size()) and (j < texte.size()) and (el.code[i] == texte[j])) {
      ++i; ++j;
    }
    if ((i == el.code.size()) and (j <= texte.size())) { // match with el
      pos = j; // update pos to point to the rest of texte
      return el.entree;
    }
  }
  return string();
}

// ============================================================================
string encode(string const& texte, Code const& c)
{
  string code;
  for (size_t i(0); i < texte.size(); // c'est recherche() qui va mettre i à jour
      ) {
    code += recherche_entree(texte, i, c);
  }
  return code;
}

// ============================================================================
string decode(string const& texte_code, Code const& c)
{
  string texte;
  for (size_t i(0); i < texte_code.size(); // c'est recherche() qui va mettre i à jour
      ) {
    texte += recherche_mot_de_code(texte_code, i, c);
  }
  return texte;
}

// ============================================================================
void test(string const& texte, size_t tranche)
{
  cout << "==== Par tranches de taille " << tranche << endl;
  Code c(calcule_probas(texte, tranche));
  //affiche_code(c);
  Huffman(c);
  affiche_code(c);
  string tc(encode(texte, c));
  cout << "taille orig. = " << texte.size() << " lettres/octets" << endl;
  cout << "taille codee = " << tc.size() << " bits = " << 1 + (tc.size() - 1) / 8 << " c
  cout << "long. moyenne  = " << double(tc.size()) /  double(texte.size()) << " bits" <<
  cout << "compression  = " << 100.0 - 100.0 * tc.size() / (8.0 * texte.size()) << '%' <
  cout << "texte codé : " << tc << endl;
  cout << "check : " << decode(tc, c) << endl;
}

// ============================================================================
int main()
{
   string texte = "Un petit texte à compresser : le but de cet exercice est de réaliser c

  test(texte, 1);
  test(texte, 2);
  test(texte, 3);
  test(texte, 5);

  cout << "Entrez une chaîne : ";
  cin >> texte;
  test(texte, 1);

  return 0;
```

```
}
```