EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE – LAUSANNE
POLITECNICO FEDERALE – LOSANNA
SWISS FEDERAL INSTITUTE OF TECHNOLOGY – LAUSANNE

Faculté Informatique et Communications
Cours d'Information, Calcul et Communication, sections MA et PH
Chappelier J.-C.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Information, Calcul et Communication (SMA/SPH) : Examen II

22 novembre 2018

INSTRUCTIONS (à lire attentivement)

IMPORTANT ! Veuillez suivre les instructions suivantes à la lettre sous peine de voir votre examen annulé dans le cas contraire.

1. Vous disposez d'une heure quarante-cinq minutes pour faire cet examen (9h15 – 11h00).
2. Vous devez **écrire à l'encre noire ou bleu foncée**, pas de crayon ni d'autre couleur.
   N'utilisez **pas non plus de stylo effaçable** (perte de l'information à la chaleur).
3. Vous avez droit à toute documentation papier.
   En revanche, vous ne pouvez pas utiliser d'ordinateur personnel, ni de téléphone portable, ni aucun autre matériel électronique.
4. Répondez aux questions directement sur la donnée, **MAIS** ne mélangez pas les réponses des deux exercices !
   Ne joignez aucune feuilles supplémentaires ; **seul ce document sera corrigé**.
5. Lisez attentivement et *complètement* les questions de façon à ne faire que ce qui vous est demandé. Si l'énoncé ne vous paraît pas clair, ou si vous avez un doute, demandez des précisions à l'un des assistants.
6. L'examen comporte deux exercices indépendants, qui peuvent être traités dans n'importe quel ordre, mais qui ne rapportent pas la même chose (les points sont indiqués, le total est de 56). Toutes les questions comptent pour la note finale.

Question 1 — Moyenne et écart-type [sur 28 points]

On s'intéresse dans cette question à écrire un programme qui calcule la dispersion statistique de certaines lettres d'un tableau de chaînes de caractères autour de leur moyenne d'apparition (on parle d'« écart-type », expliqué plus loin). Typiquement, on souhaiterait avoir un résultat similaire à cet exemple :

```
Entrez une chaîne de caractères :  court
  Entrez une chaîne d'au moins 8 caractères, ou la chaîne vide pour terminer.
Entrez une chaîne de caractères :  ok j'ai compris !
Entrez une chaîne de caractères :  mais ce n'est pas si simple de trouver \
    des exemples pertinents.
Entrez une chaîne de caractères :  surtout avec un seul y en tout !
Entrez une chaîne de caractères :  et un seul c par phrase ;-)
Entrez une chaîne de caractères :
Pour a, moyenne = 1.5, ecart-type = 0.57735
Pour c, moyenne = 1, ecart-type = 0
Pour e, moyenne = 4.25, ecart-type = 4.71699
Pour i, moyenne = 1.5, ecart-type = 1.91485
Pour l, moyenne = 1, ecart-type = 0.816497
Pour m, moyenne = 1, ecart-type = 1.41421
Pour n, moyenne = 1.5, ecart-type = 1.29099
Pour o, moyenne = 1.25, ecart-type = 0.957427
Pour p, moyenne = 1.75, ecart-type = 1.70783
Pour t, moyenne = 2.25, ecart-type = 2.06155
Pour u, moyenne = 2, ecart-type = 2.16025
Pour y, moyenne = 0.25, ecart-type = 0.5
```

Le programme complet visé calculera, pour chaque caractère appartenant à la liste (a, c, e, i, l, m, n, o, p, t, u, y), la moyenne et l'écart-type de son nombre d'apparition par phrase (expliqués plus loin) sur un ensemble de phrases entrées par l'utilisateur ; puis en affichera le résultat comme dans l'exemple ci-dessus.

Ce programme est constitué de la fonction `main()` suivante :

```
int main()
{
  print_stats(get_strings(), "aceilmnoptuy");
  return 0;
}
```

On vous demande ici de compléter *certaines parties* de ce programme, en utilisant le langage C++ et une approche « procédurale » avec le moins de duplication de code possible. En plus des fonctions et types de données demandées dans cet énoncé, vous êtes libres d'ajouter vos propres fonctions ou types de données intermédiaires si nécessaire.

Question 1.1 – Comptes [sur 9.5 points]

Pour commencer, on vous demande d'écrire ici :

- une fonction `char_count()` qui, pour un caractère et une chaîne de caractères passés en paramètres, calcule le nombre de fois que ce caractère est contenu dans la chaîne ; p.ex. `char_count('a', "blabla")` retourne 2 ;
- une *autre* fonction `char_count()` qui, pour un caractère et *un tableau de* chaînes de caractères passés en paramètres, calcule le nombre de fois que ce caractère est contenu dans chacune des chaînes du tableau et retourne le tableau d'entiers correspondant (dans le même ordre) ; p.ex. `char_count('a', { "blabla", "toc-toc", "tic-tac" })` retourne $(2, 0, 1)$.

Voici une version minimale suffisante :

```
int char_count(char c, string L)
{
  int retour(0);
  for (auto a : L) {
    if (a == c) ++retour;
  }
  return retour;
}

vector<int> char_count(char c, vector<string> t)
{
  vector<int> rez;
  for (auto s : t) {
    rez.push_back(char_count(c, s));
  }
  return rez;
}
```

Question 1.2 – Moyenne [sur 4.5 points]

On s'intéresse maintenant à calculer la moyenne d'une collection de n valeurs x_i $(1 \leq i \leq n)$ suivant la formule

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i.$$

Écrivez ici une fonction `mean()` qui calcule la moyenne d'un tableau de nombres entiers reçu en argument.
Note : la moyenne d'une collection de nombres entiers n'est pas nécessairement un entier.

Voici une version minimale suffisante :

```
double mean(vector<int> t)
{
  double rez(0.0);
```

```
  for (auto x : t) {
    rez += x;
  }
  if (t.size() > 0) rez /= t.size();
  return rez;
}
```

Note : la valeur moyenne d'une liste vide n'est pas spécifiée. Ici nous retournons 0, mais n'importe quelle autre valeur est acceptée. Ce qui compte c'est de ne pas faire de division par 0.

Question 1.3 – Écart-type [sur 8 points]

La formule pour calculer l'écart-type σ_x d'une collection de n valeurs x_i $(1 \leq i \leq n)$ est la suivante :

$$\sigma_x = \sqrt{\frac{1}{n-1}\left(\sum_{i=1}^{n} {x_i}^2\right) - \frac{n}{n-1}\,\overline{x}^{\,2}}$$

où $\overline{x}$ représente la moyenne calculée précédemment.

Écrivez sur la page suivante une fonction `std_dev()` qui calcule l'écart-type et la moyenne d'un tableau de nombres reçu en argument. Cette fonction devra (d'une façon ou d'une autre) « retourner » l'écart-type *et* la moyenne. Vous devez de plus utiliser *exactement* la formule donnée ci-dessus (et non pas une autre formule équivalente).

Voici une version minimale suffisante :

```
void std_dev(vector<int> t, double & moyenne, double & ecarttype)
{
  ecarttype = 0.0;
  moyenne   = mean(t);

  if (t.size() > 1) {
    for (auto x : t) {
      ecarttype += x * x;
    }
    ecarttype /= (t.size() - 1);

    ecarttype = sqrt(ecarttype - moyenne * moyenne * t.size() / (t.size() - 1));
  }
}
```

Même remarque que précédemment pour les cas de liste de vide et liste réduite à 1 seul élément nous retournons ici, mais n'importe quelle autre valeur est acceptée (le but n'est pas de vous tester en statistiques mais ce qui compte c'est de ne pas faire de division par 0).

Concernant la valeur de retour : la solution proposée ici passe par référence. Une alternative consiste à déclarer une structure contenant deux champs (moyenne, écart-type) et à la retourner. Cette alternative est tout aussi valable.

Question 1.4 – Affichage des résultats [sur 6 points]

Pour finir, nous vous demandons d'écrire ici

— une fonction `print_stats()` qui prend en paramètres un caractère et un tableau de chaînes de caractères et qui affiche sur la sortie standard la moyenne et l'écart-type du nombre de fois que ce caractère est présent dans les chaînes du tableau ;
p.ex. `print_stats('a', { "blabla", "toc-toc", "tic-tac" });` affiche

```
Pour a, moyenne = 1, ecart-type = 1
```

puisque la moyenne du tableau $(2, 0, 1)$ vaut 1 et son écart-type vaut 1 aussi ;

— une *autre* fonction `print_stats()` qui généralise la fonction précédente à une chaîne de caractères en appliquant cette dernière tour à tour à chacun des caractères de la chaîne de caractères fournie ; voir la fonction `main()` et l'exemple de déroulement fournis en début d'exercice.

Voici une version minimale suffisante :

```
void print_stats(vector<string> ts, char c)
{
  double m;
  double sigma;

  std_dev(char_count(c, ts), m, sigma);

  cout << "Pour " << c << ", moyenne = " << m
       << ", ecart-type = " << sigma << endl;
}

void print_stats(vector<string> ts, string s)
{
  for (auto c : s) print_stats(ts, c);
}
```

Question 2 — Un peu de musique [sur 28 points]

Dans cet exercice, on aimerait réaliser une application permettant de gérer des playlists, listes de fichiers de musique. Chaque fichier de musique possède plusieurs caractéristiques :

- le « nom » du fichier : chemin absolu où est il stocké sur le disque dur ;
- la durée de la chanson (en nombre entier de secondes, p.ex. 321) ;
- la taille du fichier (nombre entier de Ko, p.ex. 65432) ;
- son taux de compression (pourcentage, p.ex. 0.33).

On vous demande ici de compléter *certaines parties* d'un tel programme, en utilisant le langage C++ et une approche « procédurale » avec le moins de duplication de code possible. En plus des fonctions et types de données demandées dans cet énoncé, vous êtes libres d'ajouter vos propres fonctions ou types de données intermédiaires si nécessaire.

Question 2.1 – Structures de données [sur 3 points]

Commencez par définir (ici) les types suivants :

- `Morceau`, pour représenter un fichier de musique tel que présenté ci-dessus ;
- `collectionMusique`, pour stocker un ensemble de fichiers de musique (playlist).

```
struct Morceau {
  string chemin;
  unsigned int duree;
  unsigned long taille;
  double taux;
};
```

```
typedef vector<Morceau> collectionMusique;
```

Question 2.2 – Maxima [sur 3.5 points]

On souhaite avoir une fonction `max()` qui, pour une playlist donnée en paramètre, sort un `Morceau` dont le champ « nom du fichier » est vide et dont chacun des autres champs contient le maximum de la valeur correspondante dans la playlist reçue en paramètre (p.ex. la durée de ce « morceau de musique » artificiel est la plus grande des durées de la playlist, etc.).

Écrivez sur la page suivante (au dos de cette feuille) le code d'une telle fonction.

Code de la fonction `max()` :

Voici une version minimale suffisante :

```
Morceau max(collectionMusique c) // ou ref. const.
{
  Morceau retour({ "", 0, 0, 0.0 });
  for (auto m : c) {
    if (m.duree  > retour.duree ) retour.duree  = m.duree ;
    if (m.taux   > retour.taux  ) retour.taux   = m.taux  ;
    if (m.taille > retour.taille) retour.taille = m.taille;
  }
  return retour;
}
```

Question 2.3 – Fréquence d'échantillonage maximale [sur 4.5 points]

Codez une fonction `ratio_max()` qui, pour chaque fichier de musique d'une playlist passée en paramètre, calcule le ratio défini mathématiquement par (taille/durée)/taux_compression et retourne son maximum sur toute la playlist.

On adoptera la convention que ce ratio est nul si le taux de compression ou la durée est nul(le).

Voici une version minimale suffisante :

```
double ratio_max (const collectionMusique& fichiers_musicaux) {
  double r_max(0.0);
  for (auto m : fichiers_musicaux) {
    if ((m.duree != 0) and (m.taux != 0.0)) {
      /* ATTENTION à la division entière ici : ne pas diviser la
       *   taille par la durée sans changement de type !          */
      double r( m.taille / ( m.duree * m.taux) );
      if (r > r_max) r_max = r;
    }
  }
  return r_max;
}
```

Question 2.4 – Affichage [sur 3.5 points]

Écrivez une fonction d'affichage pour les playlists. Cette fonction affichera (voir exemple ci-dessous) :

- le nombre de morceaux de la playlist ;
- les durée, taille, taux de compression et ratio maximaux ;
- la liste des morceaux avec leurs caractéristiques propres.

Par *exemple* (il n'est pas demandé de reproduire exactement ce format, seules les *informations* affichées nous importent) :

```
Playlist de 5 morceaux :
  durée  max. : 189 s
  taille max. : 1500 Ko
  taux compr. max. : 0.48
  ratio  max. : 35.8878
  chemin                    duree taille  taux
- /home/chaps/Musique/a.mp3   121   1433  0.33
- /home/chaps/Musique/b.ogg   189   1500  0.42
- /home/chaps/Musique/c.wav    73    512  0.21
- /home/chaps/Musique/d.flac  104   1000  0.48
- /home/chaps/Musique/e.sox   176    999  0.19
```

```
void affiche(collectionMusique c) // ou ref. const.
{
  cout << "Collection de " << c.size() << " morceaux" << endl;
  const Morceau m(max(c));
  cout << "  durée  max. : " << m.duree  << " s" << endl;
  cout << "  taille max. : " << m.taille << " Ko" << endl;
  cout << "  taux compr. max. : " << m.taux << endl;
  cout << "  ratio  max. : " << ratio_max(c) << endl;

  for (auto x : c) {
    cout << "\t- " << x.chemin << "\t" << x.duree
         << "\t" << x.taille << "\t" << x.taux << endl;
  }
}
```

Question 2.5 – Suppression de gros fichiers [sur 8.5 points]

Créez une fonction `nettoyer()` qui supprime d'une playlist passée en paramètre tous les fichiers de musique dont la taille est strictement supérieure à une valeur donnée en paramètre.

Par exemple, sur la playlist affichée dans la sous-question précédente, l'appel `nettoyer(playlist, 1000)` supprimerait les deux premiers fichiers (`a.mp3` et `b.ogg`) et garderait les autres, pas nécessairement dans le même ordre (nos playlist ne sont pas ordonnées).

Il n'est pas spécifié si la fonction `nettoyer()` doit travailler sur place (modifier son argument) ou retourner une nouvelle liste. Comme discuté en cours, les deux sont possibles, c'est une question de point de vue.

Voici *une* version minimale suffisante avec modification de la liste reçue :

```
void supprime(collectionMusique& v, size_t place) {
  /* La collection n'est pas supposée être ordonnée,
   * voici donc une suppression en O(1).              */
  if ((v.size() > 0) and (place < v.size())) {
    const size_t optimiz(v.size()-1);
    if (place < optimiz) {
```

```
      v[place] = v[optimiz];   // ou plus simplement swap(v[place], v.back());
    }
    v.pop_back();
  }
}

void nettoyer(collectionMusique& fichiers_musicaux, unsigned long int taille) {
  for (size_t i(0); i < fichiers_musicaux.size(); ++i) {
    if (fichiers_musicaux[i].taille > taille)  {
      supprime(fichiers_musicaux, i); // BONUS pour la modularisation
      --i; // pour ne pas rater le suivant !!
    }
  }
}
```

Question 2.6 – Intersection de playlists [sur 5 points]

On souhaite enfin créer l'intersection de deux playlists. On considérera que deux fichiers de musique sont égaux s'ils ont simplement le même nom de fichier.

Coder ici une fonction `intersection()` réalisant cette fonctionnalité.

L'intersection de *listes* n'ayant pas été définie formellement, son interprétation en terme d'ensembles (sans répétition) ou en terme de liste (avec autant de répétitions que dans l'une des deux ; pas forcément symétrique, du coup) sont toutes les deux acceptées.

Voici donc *une* version minimale suffisante (intersection au sens des listes, la première étant prioritaire) :

```
bool est_dans(Morceau x, collectionMusique const& c)
{
  for (auto y : c) {
    if (x.chemin == y.chemin) return true;
  }
  return false;
}

collectionMusique intersection(collectionMusique const& c1,
                               collectionMusique const& c2)
{
  collectionMusique resultat;

  for (auto x : c1) {
    if (est_dans(x, c2)) {  // BONUS pour la modularisation
      resultat.push_back(x);
    }
  }
```

```
  return resultat;
}
```