MagmaDecrypt

Table des matières

1. Introduction

Ce projet propose de décrypter un motif codé par une grille de symboles alphanumériques appartenant à un ensemble fini de **nbS** symboles. Ces symboles peuvent se combiner conformément à une table similaire à celle indiquée en **Fig 1a** avec les règles suivantes. Tout d'abord, le résultat de la combinaison doit être l'un des **nbS** symboles. Ensuite, nous posons que le premier symbole agit comme un élément neutre (cf première ligne et première colonne des résultats). De plus l'opération de combinaison est commutative (la table est symétrique). Enfin cette opération est idempotente (tout symbole combiné avec lui-même donne ce symbole). Par contre, nous posons qu'il n'y a pas de garantie que l'associativité soit vérifiée, c'est-à-dire que, ayant 3 variables **a**, **b** et **c** avec comme valeur l'un des **nbS** symboles, alors le résultat de **(ab)c** peut être différent de **a(bc)**. En conséquence les valeurs *sx*, *sy* et *sz* de la table de la Fig 1a peuvent être n'importe quelles valeurs parmi les **nbS** symboles[1].

La figure 1b du haut montre une grille à décrypter. Le décryptage remplace chaque bloc de 2x2 symboles en un seul symbole à l'aide d'une clef de décryptage qui indique l'ordre des combinaisons (détails en section 3). Le décryptage s'arrête dès que la grille résultante a un nombre impair de lignes ou de colonnes.

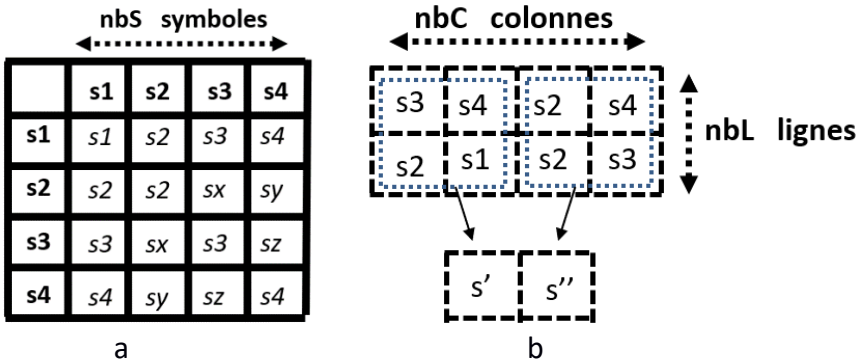


Figure 1 : (a) table de composition des **nbS** éléments d'un ensemble fini de symboles alphanumériques. (b-haut) grille de **nbL** lignes sur **nbC** colonnes symboles appartenant à cet ensemble ; (b-bas) décryptage par bloc de 2x2 symboles ; la clef du décryptage indique l'ordre des combinaisons des éléments du bloc (section 3) ; chaque résultat s' et s'' est l'un des **nbS** symboles. Le décryptage s'arrête dès que le nouveau nombre de lignes ou de colonne devient impair, ce qui est le cas ici.

Votre programme devra d'abord vérifier la validité des données du problème, puis afficher les informations suivantes : la table complète de composition des **nbS** symboles puis la grille à décrypter et chaque étape du décryptage (une seule étape pour la fig 1b). Tout cela est détaillé en section 3.

[1] C'est pourquoi cet ensemble fini de symboles est appelé un *magma* plutôt qu'un *groupe* au sens mathématique de ces termes.

2. Méthode de travail

Avertissement IA : des études très récentes montrent que l'usage de l'IA peut être bénéfique mais seulement pour les personnes ayant de bonnes bases car elles savent évaluer les propositions de l'IA. Par contre, les autres, qui commencent la programmation, peuvent perdre du temps à cause de bugs non détectés (comme l'ont indiqué les rapports du projet de l'an dernier). De plus ces personnes débutantes présentent ensuite de grosses lacunes au moment de l'examen écrit (sans IA ni machine). Donc, l'usage de IA est autorisé mais présente des risques et devra être documenté (section 6.1).

2.1 Mise en oeuvre des grands principes

Le projet représente une quantité de travail bien supérieure aux exercices proposés dans les séries. La mise en oeuvre des principes ***d'abstraction*** et de ***ré-utilisation*** est un élément central dans la stratégie de résolution.

En effet, certaines tâches correspondent à un sous-problème (*abstraction*) dont la solution sous forme d'une fonction peut être utilisée pour résoudre une tâche plus complexe. C'est tout à fait normal qu'une telle fonction ne soit appelée qu'une seule fois car le but est la structuration de la solution d'une manière claire et lisible (un critère fréquent est que la taille maximum d'une fonction ne doit pas dépasser une page écran). Dans certains contextes la conception d'une fonction peut être ajustée à l'aide de paramètres pour pouvoir être *ré-utilisée* à plusieurs endroits dans le code.

2.2 Faire une analyse papier-crayon et pseudocode AVANT de vous lancer dans le codage

Le problème est décrit d'une manière indépendante d'un langage de programmation en section 3 ; c'est ce qu'on appelle les *spécifications*. Une telle description permet de réfléchir, *sans programmer*, à la décomposition du problème en un ensemble de sous-problèmes qui seront réalisés par des fonctions ; c'est ce qu'on appelle la phase *d'analyse*. Cette phase est typiquement faite avec un papier-crayon en pseudocode comme support pour organiser vos idées et faciliter le dialogue avec les assistant.e.s.

Le résultat de cette phase d'analyse est un ensemble de fonctions dont le *but* de chaque fonction est clairement identifié : sur quelles données travaille-t-elle ? Quel(s) résultat(s) fournit-elle ?

Ensuite seulement on peut passer à une méthode de codage rigoureuse qui va vous garantir une progression régulière dans la mise au point du projet. Elle est décrite ci-dessous.

2.3 Vérification précoce par les tests (*scaffolding*)

La clef du succès du codage est de ***vérifier*** que chaque fonction réalise bien son but avec un *solide éventail de tests pour lesquels on connait les résultats attendus*. Grâce à cette méthode un sous-problème est résolu une fois pour toute dès le niveau le plus élémentaire.

L'approche de vérification par des tests implique d'*écrire du code supplémentaire dédié à ces tests* AVANT de commencer à traiter les niveaux supérieurs du projet. Vous serez ainsi amené à écrire un certain nombre de petits programmes dédiés à ces tests. Cette méthode de travail est appelée *scaffolding* (échafaudage) et cherche à rendre votre code robuste à la grande variété des cas possibles.

En effet nous disposons d'outils tels que l'éditeur et le compilateur pour détecter certaines erreurs mais ils ne permettent pas de toutes les trouver. Nous savons déjà qu'il est

recommandé de *recompiler très fréquemment* afin réduire au minimum le temps de recherche des ***erreurs syntaxiques*** (fautes d'orthographe/grammaire du langage C++). Le raisonnement est qu'une erreur se trouve dans la portion de code écrite depuis la dernière compilation avec succès, ce qui permet de réduire à très peu de lignes de codes l'espace de recherche de cette erreur. La méthode de l'échafaudage (scaffolding) est destinée à trouver les ***erreurs sémantiques*** que le compilateur ne trouve pas car le programme respecte la syntaxe du langage ; les erreurs sémantiques vont produire un comportement incorrect du programme. Le point essentiel dans cette méthode est qu'il faut tester *chaque fonction* ***individuellement*** pour *vérifier* qu'elle produit le résultat attendu AVANT de l'utiliser ailleurs. Là aussi cette approche vous rend plus efficace dans le développement de code car l'erreur sémantique se trouve normalement dans la dernière fonction en cours de test car les autres fonctions qu'elle appelle doivent être déjà validées.

2.4 Bottom-up ou top-down ?

La méthode présentée dans la section précédente suggère de vérifier d'abord les fonctions de bas-niveaux avant celles qui les utilisent. C'est l'approche ***bottom-up*****.** C'est en général ce que nous recommandons pour un projet.

A l'inverse, il existe aussi une approche ***top-down*** pour la mise au point des fonctions ; il peut être légitime de vouloir tester une fonction **f()** qui appelle une fonction **g()** avant que **g()** soit écrite en détail. Cette approche *top-down* est possible si le résultat de **g()** est facile à définir, par exemple si elle renvoie `true` ou `false`, car la seule chose utilisée par **f()** est ce résultat. On peut ainsi vérifier **f()** à l'aide d'une *forme minimale de la fonction* **g()** que l'on appelle un ***stub***. Cette forme minimale respecte le prototype prévu pour **g()** en terme de paramètres et de type de la valeur de retour ; par contre sa définition peut se restreindre à un corps vide si aucune valeur n'est retournée ou très peu de chose, par exemple une instruction return de *true* ou *false* si **g()** est supposée renvoyer un booléen.

2.5 Redirection des entrées-sorties pour automatiser les tests d'un programme

On utilisera **exclusivement** l'entrée standard (*clavier*) pour fournir les données et la sortie standard (*terminal*) pour afficher les résultats. Il ne faut pas écrire de code de lecture-écriture de fichier dans ce projet.

A la place, le mécanisme de *redirection* des entrées-sorties est vu en TP (semaine6) et permet de travailler avec des fichiers de test *sans avoir à changer une seule ligne de code*. En effet les données d'un test peuvent être éditées avec l'éditeur de programme et mémorisées dans un fichier de test. Ensuite il suffit de préciser le nom du fichier de test sur la ligne de commande qui lance l'exécution du programme et celui-ci obtient les données du fichier comme si elles venaient du clavier.

Exemple : si le nom de l'exécutable de votre projet est **proj** et que le nom d'un fichier de test est **t00.txt**, alors vous pouvez lancer votre exécutable dans le terminal de la façon suivante :

```
./proj < t00.txt
```

où le contenu de **t00.txt** est envoyé sur l'entrée standard pour cette exécution de **proj**.

Cette méthode de test est recommandée surtout pour relancer fréquemment et efficacement un ensemble de tests et vérifier que votre programme est toujours correct.

C'est aussi en redirigeant des fichiers de test sur l'entrée standard que nous noterons votre programme (précisions en section 4.4.2). Il faut donc strictement respecter cette consigne.

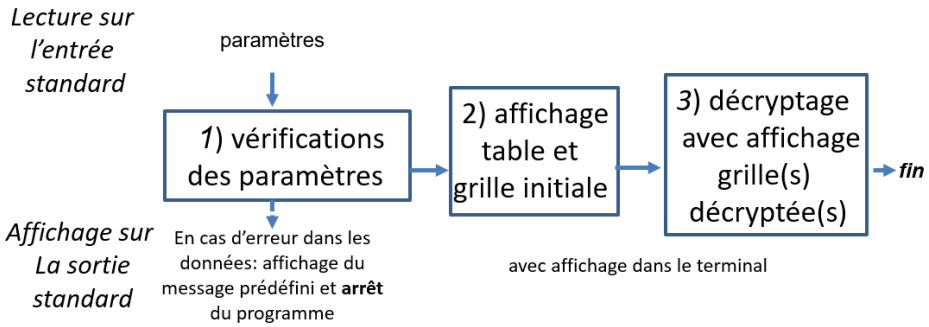


Figure 2: Tâches à effectuer par le programme : tout d'abord la vérification des paramètres avec arrêt en cas de détection d'erreur. Ensuite, on affiche la table de composition des symboles et la grille initiale à décrypter. Enfin une boucle décrypte la grille et affiche le résultat tant que son nombre de lignes et de colonnes sont pairs

Remarque : la lecture s'effectue sur l'entrée standard et les affichages sont effectués sur la sortie standard.

3. Spécifications détaillées

Cette section approfondit les éléments fournis en section 1 en cherchant à identifier des structures de données et des fonctions pouvant être ré-utilisées.

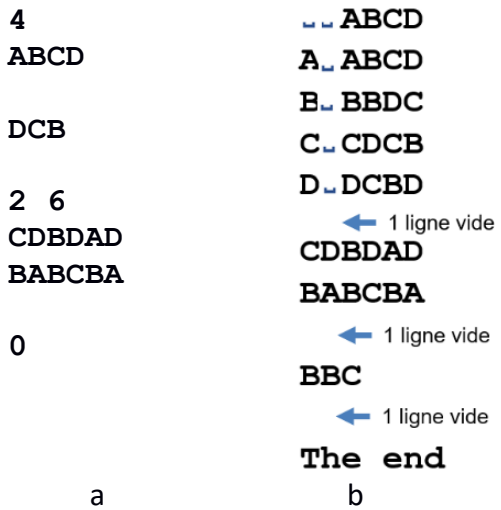
3.1 Buts des tâches

Le programme est une suite de trois tâches (Fig 2): d'abord la lecture et la vérification des données, ensuite l'affichage de la table complète des combinaisons et la grille initiale, enfin le décryptage avec affichage de(s) grille(s) décryptée(s).

3.1.1 Tâche 1 : lecture et vérification des paramètres

Plusieurs types d'erreurs doivent être détectés, dès la lecture de chaque paramètre ; un exemple de jeu de paramètres est visible dans la Fig 3a. Par souci de cohérence avec le programme du cours de C++, ce projet suppose que *les valeurs fournies en entrée sont du bon type* car le cours traitant de cette classe d'erreur n'est donné qu'en fin de semestre. Par contre il reste à vérifier que les valeurs fournies en entrées appartiennent au domaine de valeurs autorisées. La suite de cette section décrit cette classe de vérifications à effectuer.

Dès qu'une de ces vérifications détecte une erreur, le programme doit afficher un message d'erreur prédéfini puis s'arrêter. Nous désignons les messages d'erreur par une expression en majuscules dans la suite de cette section ; la manière de les afficher est fournie en section 4.4. En particulier, vous devez utiliser le fichier fourni **useful_stuff.txt** pour le recopier dans votre programme.



␣ espace

Figure 3: contenu du fichier de test t00.txt (a) ; affichage obtenu pour le fichier t00.txt lorsqu'il est redirigé sur l'entrée standard (b) ; l'espace est matérialisé par un petit dessin (ci-dessus).

Lecture des données : les données sont fournies sur l'entrée standard (stdin = clavier) sur des lignes distinctes selon l'ordre indiqué sur l'exemple de la Figure 3a. Les données d'une même ligne peuvent être précédées, suivies et séparées par un ou plusieurs espaces ou une ou plusieurs tabulations. Les lignes des données peuvent être séparées par zéro ou plusieurs lignes vides. Les paragraphes suivants décrivent les données fournies au programme :

Valeur de **nbS** : doit être strictement supérieure à 2 et inférieure ou égale à 64 (détection d'erreur: BAD_SYMBOL_NB).

Une unique chaine de caractères contenant les **nbS** symboles : doivent être des caractères imprimables dont la valeur du code ASCII appartient à l'intervalle [0x21, 0x7D]. Les erreurs à détecter sont : 1)nombre de caractères différent de **nbS**: WRONG_SYMBOL_NB, 2) valeur incorrecte d'un caractère : WRONG_SYMBOL_VALUE, et 3) il doivent être distincts les uns des autres : DUPLICATED_SYMBOL_VALUE.

Une unique chaine de caractères contenant les **nbt** symboles permettant de compléter la table des combinaisons : en effet les propriétés d'élément neutre, idempotence et commutativité permettent de déterminer automatiquement **nba=(nbS+2*(nbS-1))** combinaisons. Il suffit de donner **nbt=(nbS^2-nba)/2** caractères pour compléter le triangle supérieur de la table, de gauche à droite et de haut en bas, avec des caractères appartenant à l'ensemble des **nbS** caractères lus à l'étape précédente ; par exemple pour la Fig1 ils sont donnés dans l'ordre : *sx* puis *sy* puis *sz*. Les erreurs à détecter sont : 1)nombre de caractères différent de **nbt**: WRONG_COMPL_SYMBOL_NB, et 2) valeur incorrecte d'un caractère : WRONG_COMPL_SYMBOL_VALUE.

Nombre de lignes (**nbL**) et nombre de colonnes (**nbC**) de la grille initiale à décrypter : doivent être supérieurs ou égal à 2 et inférieur ou égal à 64. Détection d'erreur: BAD_GRID_SIZE.

nbL chaines de caractères sur des lignes distinctes, chacune contenant **nbC** symboles : il faut vérifier que chaque chaine contient bien **nbC** symboles (détection d'erreur : WRONG_GRID_LINE_SIZE) et que ces symboles appartiennent à l'ensemble des nbS symboles lus (détection d'erreur : WRONG_GRID_SYMBOL_VALUE). Par souci de simplification, tous nos fichiers de test contiendront le bon nombre **nbL** de chaînes ; nous ne demandons pas de tester ce type d'erreur.

Valeur de la clef de décryptage : *un seul* caractère alphanumérique dont le motif binaire est analysé pour déterminer comment condenser un bloc de 2x2 caractères en un seul caractère à l'aide de la table des combinaisons. Les détails de cette analyse sont détaillés en section 3.1.3 pour savoir si le caractère fourni est valide. Détection d'erreur de valeur invalide: BAD_KEY_VALUE. Par souci de simplification, tous nos fichiers de test contiendront un seul caractère ; nous ne demandons pas de tester ce type d'erreur.

3.1.2 Tâche 2 : Affichage de l'état initial : table et grille

Si aucune erreur n'est détectée le programme continue avec la tâche2 qui consiste à afficher l'état initial (Fig 3b).

On affiche d'abord la table complète des combinaisons (La Fig3b illustre les espaces à ajouter pour la lisibilité) suivie par une ligne vide, puis la grille initiale de nbL x nbC caractères, suivie par une ligne vide.

3.1.3 Tâche 3 : Algorithme de décryptage

Si **nbL** et **nbC** sont paires alors le décryptage consiste à remplacer chaque bloc de 2x2 caractères de la grille initiale par un seul caractère ; la grille décryptée est alors affichée, suivie par une ligne vide. Le décryptage continue tant que les nouveaux nombres de lignes et de colonnes sont pairs. Lorsque cette tâche est terminée le programme se termine après l'affichage du message THE_END prédéfini dans **useful_stuff.txt**.

Comment utiliser la clef de décryptage :.

Un magma commutatif ne garantit pas la propriété d'associativité ; c'est pourquoi l'ordre de combinaison des éléments d'un bloc 2x2 doit être indiqué sans ambiguïté. C'est le rôle de la clef de décryptage de définir cet ordre. Pour illustrer cela, nous nommons les 4 éléments d'un bloc respectivement **a**, **b**, **c** et **d** comme sur la Fig 4

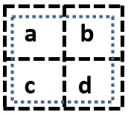


Fig 4 : les 4 variables d'un bloc 2x2 d'une grille

Il suffit de distinguer ces deux familles de combinaisons :

- 12 séquences : ((ab)c)d ; ((ab)d)c ; ((ac)b)d) ; ((ac)d)b) ; ((ad)b)c) ; ((ad)c)b) ; ((bc)a)d) ; ((bc)d)a) ; ((bd)a)c) ; ((bd)c)a) ; ((cd)a)b) ; ((cd)b)a).
- 3 combinaisons hiérarchiques : (ab)(cd) ; (ac)(bd) ; (ad)(bc)

Nous codons ces possibilités avec un caractère alphanumérique *imprimable* comme suit :

- Le bit de poids fort associé à 2^7 est toujours nul
- Séquence : le bit associé à 2^6 vaut toujours 1 ; ensuite il reste 6 bits de poids faibles, c'est-à-dire 3x2bits, pour coder la séquence des combinaisons avec les 3 premières variables de la séquence (la quatrième et dernière variable s'en déduit automatiquement). Nous associons aux variables **a**, **b**, **c** et **d** les motifs binaires **00**, **01**, **10**, et **11**. Exemple :
  - Le caractère **X** de motif **01011000** code une séquence **((bc)a)d)** car le premier motif binaire à droite de 2^6 vaut **01** qui code **b**, ensuite vient **10** qui code **c**, puis **00** qui code **a** ; le dernier opérande est donc **d** car il n'est pas encore utilisé.
- Combinaison hiérarchique : le bit associé à 2^6 vaut toujours 0 ; ensuite nous utilisons respectivement les codes ASCII de **0**, **1** et **2** pour les trois combinaisons hiérarchiques. L'exemple de la Fig3 utilise **0** ; c'est donc la combinaison **(ab)(cd)** qu'il faut utiliser pour l'évaluation de tous les blocs 2x2. On évalue d'abord les sous-expressions **ab** et **cd** puis on combine leur résultat.

4. Implémentation en langage C++

4.1 Contraintes spécifiques à ce projet

Indépendamment de ce que doit faire le programme (les spécifications décrites plus haut) nous imposons les contraintes suivantes de mise en oeuvre :

- Ecriture en C++ : votre code sera compilé avec l'option **`-std=c++17`**
- Tout votre code doit être écrit dans un seul fichier source.
- Il faut utiliser le type **vector** pour mémoriser la table et les grilles car leur taille n'est pas connue au moment de l'écriture du code.

4.2 Clarté et structuration du code avec des fonctions

La clarté de votre code sera prise en compte. Un examen manuel de votre code source sera effectué par une personne chargée d'évaluer le respect des conventions de programmation utilisées dans ce cours. Par souci d'efficacité seuls les codes indiqués dans nos conventions vous seront communiqués avec les numéros des lignes concernées dans votre fichier source.

4.3 Variables locales ou globales ?

4.3.1 Où déclarer les variables et les tableaux ?

La règle de base est qu'une variable (ou un tableau) n'est déclarée que ***localement***, là où elle est utilisée, le plus bas possible dans la hiérarchie des appels de fonctions. **Si et seulement si** une variable **x** (ou un tableau) déclarée dans une fonction **h()** est ***nécessaire*** pour une autre fonction **f(),** alors elle est transmise en paramètre à cette fonction[2] au moment de l'appel **f(x)** avec transmission par valeur ou par référence selon vos besoins**.**

Conséquence: si deux fonctions **f()** et **g()** indépendantes l'une de l'autre doivent travailler sur la même variable **x** (ou un tableau) alors une fonction de niveau supérieur **h()** doit être écrite qui va appeler **f(x)** et **g(x)** avec transmission par valeur ou par référence selon vos besoins.

4.3.2 Qu'en est-il des ***constantes*** ?

Voici les règles que nous nous donnons, en conformité avec nos conventions de programmation :

- Si une constante n'est utilisée qu'à l'intérieur d'*une seule fonction*, alors la déclaration d'une variable avec **constexpr** peut être faite dans cette fonction ou globalement.
- Si la constante est utilisée dans *plus d'une fonction*, alors la déclaration d'une variable avec **constexpr** *doit* être faite de manière globale, en début de fichier comme décrit dans les conventions de programmation.
- L'alternative de la déclaration de symboles avec #define est autorisée pour des constantes utilisées dans plusieurs fonctions. Elles sont aussi déclarées en début de fichier comme décrit dans les conventions de programmation.

4.4 Fichiers de test, messages d'erreur :

Nous vous fournissons des fichiers de tests dans le folder du projet sur moodle (section 4.4.2).

4.4.1 Message d'erreur

Le fichier **useful_stuff.txt** contient les définitions C++ des constantes, dont celles des messages d'erreur et une fonction **print_error** qu'il faut recopier dans votre fichier source (cf conventions). La fonction **print_error** se charge de quitter le programme en appelant **exit(0)**.

Exemple d'utilisation : si votre code détecte l'erreur BAD_SYMBOL_NB il suffit de faire un appel en passant seulement la constante du message: **print_error(BAD_SYMBOL_NB).**

4.4.2 Vérification anticipée de l'exécution de votre projet

Pour chaque fichier de test public, noté **`txx.txt`**, nous fournissons le fichier texte de la sortie attendue, noté **`outxx.txt`**. Vous pouvez ainsi valider les tests publics en lui comparant votre propre sortie, notée par exemple **`affxx.txt`.** Tout d'abord, voici la commande qui

[2] Nous n'acceptons pas l'approche qui consiste à transmettre systématiquement un grand nombre de paramètres à la fonction **f()** et qui restent ensuite inutilisés dans cette fonction car cela la rend moins lisible. Conclusion : *chaque paramètre doit avoir sa justification*.

mémorise votre affichage pour le fichier de test **txx.txt** :

```
./proj < txx.txt > affxx.txt
```

Ensuite, toujours dans le terminal, lancez la commande **diff** qui affiche les différences entre les deux fichiers qui suivent la commande

```
diff -s  outxx.txt affxx.txt
```

Le test est validé si elle ne trouve aucune différence, en affichant le message suivant :

```
Files outxx.txt and affxx.txt are identical
```

Nous automatiserons la vérification de l'exécution de votre projet de la même manière, en comparant toutes les sorties attendues avec celle obtenues avec votre projet (après compilation par nos soins sur la VM). Nous utiliserons aussi des fichiers de tests privés, non-fournis, pour vous stimuler à créer d'autres scénarios de tests représentatifs des différents scénarios d'exécution du programme. Il suffit d'ouvrir votre éditeur de code source préféré pour écrire les données et de sauvegarder avec l'extension **.txt** .

5 Conseils et proposition d'ACTIONS pour mettre en œuvre votre projet

Les sous-sections « ACTIONS » sont seulement des suggestions pour organiser votre travail. Il est évident que chacune des fonctions proposées doit être testée avec les tests fournis et ceux que vous aurez créés vous-même, avant de l'utiliser dans une autre ACTION. Pour être efficace, utilisez la redirection des entrées-sorties (série TP_s4.2).

5.1 ACTION1 Bottom-Up d'affichage de la table des combinaisons et d'une grille

Comme indiqué en section 4.1, les entités de table et de grille sont de type **vector** ; la nature de leurs éléments peut être, au choix, soit un **vector** de **char**, soit un **string**. Dans les deux cas, on peut facilement accéder à un caractère individuel de type **char** avec l'expression **tab[i][j]** où **tab** est le nom du **vector** au niveau table/grille, **i** est l'indice de la ligne et **j** est l'indice de colonne.

Dans une approche bottom-up écrivez et testez 2 fonctions recevant une référence constante sur ce type de **vector** et affichant leur contenu comme illustré sur la Fig 3b. Ensuite ces fonctions pourront être ré-utilisées dans d'autres étapes de test et bien sûr dans le programme plus complet.

5.2 ACTION2 Bottom-Up d'analyse de la clef de décryptage et traitement d'un bloc 2x2

Comme indiqué en section 3.1.3 le caractère lu pour la clef de décryptage doit être analysé pour déterminer 1) le type de famille de combinaisons (séquence/hiérarchique) avec le bit associé à 2^6, et selon ce type 2) l'ordre des opérations. Une approche simple est de définir (12+3) constantes symboliques de Combinaison, une par type de traitement d'un bloc 2x2. Cette constante de Combinaison est renvoyée par la fonction.

Au niveau supérieur, une fonction devrait être dédiée au traitement d'un bloc 2x2. Une telle fonction reçoit en entrée le bloc 2x2 et la constante symbolique de Combinaison suggérée ci-dessus pour piloter un **switch** ; l'ensemble des **case** va mettre en œuvre les combinaisons des 4 variables du bloc pour obtenir un seul caractère qui est renvoyé.

5.3 ACTION3 : bottom-up fonction réalisant une étape de traitement de la grille

Une telle fonction reçoit une grille pour laquelle on veut effectuer un passage de décryptage

(la constante de Combinaison de 5.2 lui est aussi passée). Le but de cette fonction est de traiter tous les blocs 2x2 et de remplacer chacun par un seul caractère avec l'autre fonction validée en 5.2. Une stratégie efficace consiste à modifier la grille fournie plutôt que de créer une nouvelle grille. Ré-utilisez la fonction d'affichage de 5.1 pour vérifier vos résultats.

5.4 ACTION4 Top-down de test de la lecture des paramètres

Ici l'idée est au contraire de considérer la fonction principale **main()** comme une « table des matières » de haut-niveau qui contient essentiellement des appels de fonctions réalisant les différentes tâches du projet. Commencez par définir comment déléguer la tâche de lecture (et de vérifications) à une ou plusieurs fonctions. L'ordre de la lecture est donné en 3.1.1.

Pour que cela fonctionne correctement **main()** doit déclarer les structures de données qui vont recevoir les données lues. Ces structures de données doivent être passées aux fonctions à qui on délègue la tâche de lecture (avec vérification) ; on doit effectuer un passage de paramètre qui permet la modification des structures de données par la ou les fonctions de lecture/vérification. Utilisez les fonctions d'affichage de 5.1 pour vérifier vos résultats.

6. Rendu :

Noms des fichiers : tous les fichiers ont le même nom qui doit être votre numéro de SCIPER ; ils diffèrent seulement par leur extension :

.cc pour le fichier source, **.pdf** pour le rapport, **.zip** pour le fichier archive.

Par exemple pour une personne X de numéro SCIPER **123456**, le nom de fichier source est **123456.cc** son rapport est **123456.pdf** et ces deux fichiers sont dans le fichier archive **123456.zip** .

Téléversement du fichier archive : le fichier archive contient seulement l'unique **fichier source** et le **rapport**. Il DOIT obligatoirement être de type **.zip** ; la VM met à disposition cet outil de compression. Il faudra le téléverser (upload), au plus tard le **19/12 à 17h,** à l'aide du lien sur **moodle** (Topic 14) ; le site de téléversement sera ouvert plusieurs semaines avant la date limite sur moodle. Vous êtes responsables de *vérifier* que l'upload s'est bien passé en le téléchargeant (download) dans votre compte et en examinant que tout est bien présent. Vous pouvez toujours faire un nouvel upload jusqu'à la date limite.

Votre code source doit respecter les conventions de programmation du cours dont : au maximum 87 caractères par ligne (commentaire compris) et 40 lignes par fonction.

6.1 Rapport

Le Rapport est d'au maximum **2 pages** écrites avec un traitement de texte. Il ne contient PAS de page de titre, ni de table des matières. Le Rapport contient :

a) une déclaration concernant l'usage de ces deux types d'outils :

a1) **Edstem** : indiquer en max 2 lignes comment vous vous êtes servis de cet outil par rapport à la discussion avec d'autres personnes et avec des assistants. Avez-vous consulté / posé des questions / contribué à des commentaires/des réponses ?

a2) **IA générative** : cet usage n'est pas interdit mais doit être documenté sur max 3-4 lignes. Ecrire soit « *Aucun usage d'IA générative* » ou préciser l'usage comme suit:

1) nom du/des outil(s)/éditeur(s). Ex : *ChatGPT4.0/OpenAI*, *Copilot/Microsoft*.

2) liste des tâches demandées : par exemple, *explication de concept, explication de message d'erreur, déverminage, production de code, reformulation de texte*…
3) Si vous avez utilisé l'IA, cela vous a-t-il fait gagner du temps ? (ou l'inverse ? dans ce cas, un bref exemple est bienvenu).

b) Résultat de la phase **d'analyse** (max une demi-page, police de taille 11) :
Décrire la structure générale du programme en faisant ressortir, avec concision (Fig 5), la mise en oeuvre des principes *d'abstraction* et de *ré-utilisation* dans votre projet.

c) Fournir votre **Pseudocode** de la tâche 3: cette tâche reçoit en entrée la table de combinaisons, la grille initiale et une constante indiquant quel traitement effectuer sur chaque bloc 2x2. Le pseudocode doit tenir en entier sur une seule page (soit sur la p 1 ou la p 2 mais pas entre les deux). Utilisez des fonctions, par exemple pour l'affichage d'une grille. Pour ce projet, on autorise de s'aligner sur l'usage du C++ pour l'usage des indices de liste.

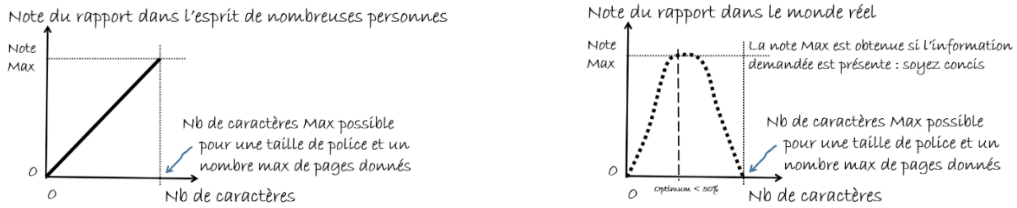


Figure 5 : un mauvais rapport dilue l'information utile jusqu'à atteindre le nombre maximum de caractères possibles sur la page (à cause de la croyance populaire de la figure de gauche) : dans la réalité ce type de rapport très compact sera très pénalisé parce qu'il est peu lisible (figure de droite) ; un bon rapport est celui qui fournit les informations demandées avec concision avec une mise en page aérée et lisible

6.2 Barème indicatif (12pts):

Ce barème est indicatif et provisoire. Il sera éventuellement modifié par la suite.

(2pt) rapport : soyez concis avec une mise en page claire (Fig 5)
(4pt) Lisibilité, structuration du code et conventions de programmation du cours
(3pt) votre programme fonctionne correctement avec les fichiers publics
(3pt) votre programme fonctionne correctement avec les fichiers non-publics

Dernier rappel : le projet d'automne est INDIVIDUEL ; vous pouvez seulement utiliser des outils de type copilot de VSCode ou chatGPT etc. Il est interdit de sous-traiter tout ou partie du travail à un tiers. De plus le détecteur de plagiat sera utilisé selon les recommandations du SAC. Le plagiat inclut la copie de code disponible sur internet. Le détecteur est efficace.