ANAGRAMMES <=> SAME GRANMA

Trouver les anagrammes d'un ensemble de mots à l'aide d'un dictionnaire

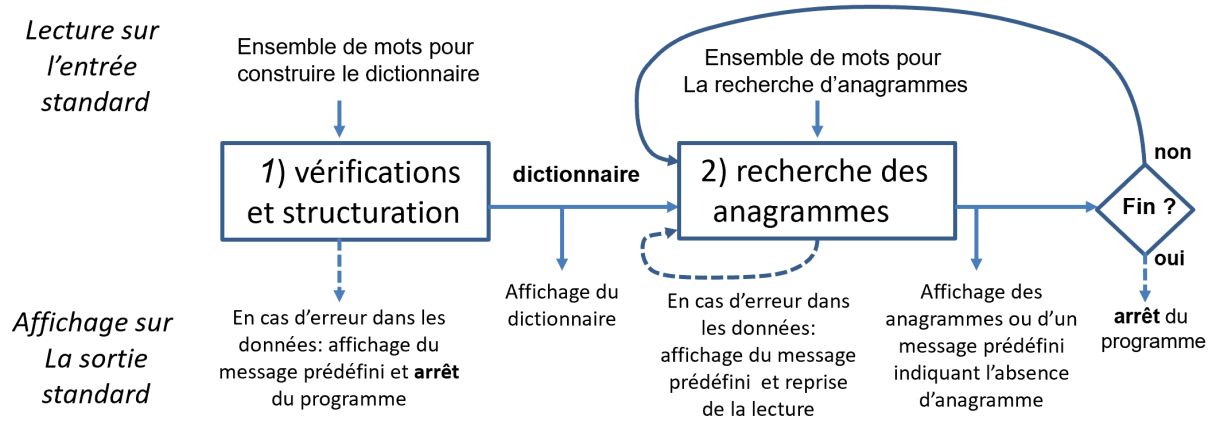


Figure 1: vue d'ensemble de la suite des deux opérations à effectuer par le programme : tout d'abord la construction du dictionnaire (étape 1) puis la boucle de recherche des anagrammes (étape 2). Les données fournies au programme sont indiquées sur la ligne du haut. Les affichages dans le terminal sont indiqués sur la ligne du bas.

1. Introduction

Ce projet cherche à afficher l'ensemble des anagrammes[1] possibles pour un ensemble de mots (un mot au minimum). Ces anagrammes peuvent eux-mêmes être constitués d'un ou de plusieurs mots.

Exemple : le titre du projet montre que les lettres du mot ANAGRAMMES peuvent être recombinées en « *SAME GRANMA* ». Il existe peut-être d'autres solutions, ce qui requiert d'apporter la contrainte suivante pour simplifier le projet :

> seul l'ensemble des mots fourni au début du programme peut être utilisé pour la recherche d'anagrammes ; dans la suite du projet, cet ensemble de mots est appelé le **dictionnaire**.

La figure 1 montre les étapes principales du fonctionnement attendu pour ce programme (détails en section3) :

1. Lecture et structuration du **dictionnaire** à l'aide de plusieurs critères de tri afin de faciliter la recherche des anagrammes et l'affichage des résultats. La lecture du caractère '.' signale la fin des mots du dictionnaire et produit un affichage de ce dictionnaire avant de passer à la seconde étape.

2. Boucle de recherche des anagrammes possibles pour le ou les mots fournis en entrée jusqu'au caractère '.' . Après l'affichage des résultats, le programme lit si le caractère '*' est fourni. Si oui, on quitte le programme et sinon on recommence la seconde étape en ré-utilisant le même dictionnaire.

[1] https://fr.wikipedia.org/wiki/Anagramme

2. Méthode du travail

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. Il est possible mais pas obligatoire qu'une fonction de bas niveau puisse 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 (tablette/stylet) comme support pour organiser vos idées. Les ébauches d'algorithmes s'expriment en pseudocode pour faciliter le dialogue avec les assistants.

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 **test1.txt**, alors vous pouvez lancer votre exécutable dans le terminal de la façon suivante :

```
proj < test1.txt
```

le contenu du fichier **test1.txt** est envoyé sur l'entrée standard pour cette exécution de **proj**.

Cette méthode de test est recommandée surtout pour relancer de manière très efficace 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 ; nous redirigerons aussi l'affichage dans un fichier que nous comparerons avec le résultat obtenu avec notre version du programme. Votre programme devra veiller à ne rien afficher de plus que les résultats demandés dans les formats précisés en section 3.

3. Spécifications détaillées

3.1 Buts des tâches

Cette section approfondit les éléments fournis en section 1 en cherchant à identifier des structures de données et des fonctions pouvant être ré-utilisées. La structure générale du programme est séquentielle avec une suite de deux tâches : d'abord la lecture des mots servant à construire et structurer le dictionnaire, puis une boucle qui recherche l'ensemble des anagrammes pour un ensemble de mots, en utilisant le dictionnaire préparé à la première étape (Fig 1).

3.1.1 Etape 1 : Construction et structuration du dictionnaire

La **tâche 1** lit les données entrées au clavier pour initialiser le dictionnaire (détails techniques en section 4).

Définition : Le **dictionnaire** est un ensemble **non-vide** de mots qui doivent respecter les propriétés suivantes :

- Chaque mot est **unique** dans le dictionnaire
- Un mot est fourni en lettres **MAJUSCULES** du code ASCII

Vous devez trier le dictionnaire selon quatre critères détaillés en section 3.1.1.1.

Lecture des mots : L'ensemble des mots peut être fourni sur plusieurs lignes. Les mots peuvent être séparés de un ou plusieurs séparateurs (espace, tabulation, passage à la ligne). La **détection du caractère '.'** *après au moins un séparateur* signale la fin des mots à utiliser pour le dictionnaire.

Détection d'erreur, affichage et arrêt du programme : Le non-respect des propriétés indiqués dans la définition du dictionnaire doit produire l'affichage d'un message prédéfini suivi de l'arrêt immédiat du programme. Nous mettons à votre disposition le fichier **error.txt** qui contient les trois messages prédéfinis à afficher (exemple en section 4.4 et Fig 3).

3.1.1.1 Structuration du dictionnaire

Le coût calcul de la recherche d'anagrammes dépend en particulier de **nbM**, le nombre de mots dans le dictionnaire. C'est pourquoi il est pertinent de structurer ce dictionnaire pour réduire autant que possible le nombre de tests à effectuer.

Propriétés des mots :
les propriétés suivantes de chaque mot devront être déterminées afin mettre en oeuvre la structuration du dictionnaire et faciliter la détection d'anagrammes :

- **nbT** : le nombre total de caractères du mot
- **nbD** : le nombre de caractères différents du mot
- **alpha** : cette chaine de caractère est initialisée avec le mot et ses caractères sont ensuite triés dans l'ordre alphabétique. Pourquoi ? Car il y a égalité de **alpha** pour deux mots qui sont anagrammes l'un de l'autre. Exemple : les mots MANOIR et ROMAIN sont des anagrammes car leur valeur de **alpha** est la même => AIMNOR.

Le dictionnaire est trié selon 4 critères hiérarchiques pour deux raisons :

- Pour que l'affichage demandé à la fin de la première étape soit le même pour tous, indépendamment de l'ordre dans lequel les mots sont lus ou traités.
- Pour accélérer autant que possible la seconde étape grâce aux relation d'ordre définies par les 3 premier critères.

Le tri hiérarchique est effectué de cette manière :

1. Ordre croissant du **nombre de caractères total** du mot **nbT**
2. Pour les mots ayant le même nombre total de caractères, Il faut trier dans l'ordre croissant du **nombre de caractères différents** du mot **nbD**.
3. Pour les mots ayant le même nombre de caractères différents, il faut trier dans l'ordre *alphabétique* de leur valeur **alpha**.
4. Pour les mots ayant la même valeur **alpha**, il faut trier les mots dans l'ordre *alphabétique*. Ensuite on n'a plus d'ambiguïté car les doublons sont interdit dans le dictionnaire.

Exemples : voici quelques exemples de dictionnaires non-triés et triés avec indication du ou des critères qui ont eu une influence dans l'ordre final pour cet exemple :

Critère(s)	Non-trié	Trié
nbT	{ACCENT, AGIR, FROID}	{AGIR, FROID, ACCENT}
nbD et **alpha**	{ABRI, ACCU, JAZZ, MAMA}	{MAMA, ACCU, JAZZ, ABRI}
nbT, **alpha** et alphabétique	{ACTE, CARTE, CARIE, ACIER	{ACTE, ACIER, CARIE, CARTE}
alphabétique	{CARIE, ACIER, CRAIE}	{ACIER, CARIE,CRAIE}

Cadre de la mise en oeuvre algorithmique :
Nous demandons d'écrire vous-même l'algorithme de tri (on ne peut pas utiliser <algorithme> du langage C++). Cela dit, nous n'imposons pas de mettre en oeuvre une solution ayant le coût calcul minimal. L'important est d'avoir un algorithme qui fonctionne en temps fini (nous aurons néanmoins un timeout dans l'autograder).

Nous recommandons de commencer l'analyse du problème à partir du pseudocode du tri vu en cours avec le premier critère **nbT** seulement. Testez ce pseudocode à la main comme en cours avec les exemples les plus simples possibles. Ensuite ajoutez le second critère **nbD**, testez votre solution à la main avec les exemples les plus simples possibles. Et ainsi de suite.

Structuration des données :

1. Stratégie avec une liste par type de donnée : pour **nbM** mots, la liste des mots, que nous notons **dico**, est complétés par une liste **nbT**, une liste **nbD** et une liste **alpha** de même taille est rangées dans le même ordre que **dico.** Grâce à cette stratégie, le mot **dico[i]** à une longueur totale **nbT[i],** un nombre de caractères différents **nbD[i]** et une chaîne **alpha[i].** L'inconvénient est que l'opération de tri doit être faite sur toutes les listes pour préserver la cohérence du problème.

2. Stratégie avec une liste de structures : l'alternative recommandée est de regrouper toutes les informations d'un mot dans une structure comme proposé dans le guide de

conception d'algorithme. Ensuite une liste de structures donne accès à chacune des informations ; si nous appelons **dico** cette liste et **mot** le nom du champ mémorisant la chaine de caractère d'un mot du dictionnaire, on accède ainsi au mot d'indice i avec avec **dico[i].mot** ainsi qu'à ses autres informations avec **dico[i].nbT, dico[i].nbD** et **dico[i].alpha.** L'avantage est qu'il suffit de trier cette unique liste.

3.1.1.2 Affichage du dictionnaire

L'affichage du dictionnaire est effectué avec un seul mot par ligne dans le terminal. La redirection de l'affichage vers un fichier permet de vérifier cet affichage dans un éditeur tel que geany. Exemple : si le nom de l'exécutable de votre projet est **proj**, que le nom d'un fichier de test contenant les données lues est **test1.txt** et que vous voulez appeler le fichier contenant la sortie **output.txt**, alors vous pouvez lancer votre exécutable dans le terminal de la façon suivante :

```
proj < test1.txt > output.txt
```

Si le fichier **output.txt** existait déjà, son ancien contenu est remplacé par celui produit par cette exécution du programme.

3.1.2 Etape 2 : Recherche des anagrammes

Cette étape obtient le dictionnaire structuré à l'étape 1 et l'utilise sur un ensemble éventuellement vide de mots pour trouver tous les anagrammes possibles de cet ensemble de mots.

Lecture : L'ensemble des mots à anagrammer est appelé le **message**, la lecture du caractère '**.**' *après au moins un séparateur* signale la fin des mots. Les mots peuvent être séparés de un ou plusieurs séparateurs de type espace, tabulation ou passage à la ligne. On autorise les doublons.

Affichage : Un **saut à la ligne** est affiché avant les anagrammes de chaque **message**. La ou les anagrammes sont affichées selon ce format : chaque MOT de l'anagramme est suivi par UN SEUL espace, SAUF le dernier mot qui est SEULEMENT suivi par un passage à la ligne. S'il n'y a aucune anagramme, le message prédéfini NO_ANAGRAM disponible dans **error.txt** doit être affiché (cf section 4.4 et Fig 3).

Fin du programme : après l'affichage, la lecture du caractère '*' isolé produit la fin du programme. Sinon on reprend à l'étape de lecture du **message**.

Détection d'erreur, affichage et reprise de la lecture : il faut que toutes les lettres soient en majuscule. Les séparateurs espace, tabulations et passage à la ligne sont autorisés. Le message d'erreur standardisé est disponible dans le fichier **error.txt**. On reprend la lecture d'un nouveau **message** après l'affichage du message d'erreur (cf section 4.4 et Fig 3).

3.1.2.1 Algorithme de détection d'anagramme

Tout d'abord il faut déterminer les propriétés globales du **message** que nous notons **nbT_m**, **nbD_m** et **alpha_m**.

La recherche dans le dictionnaire sera limité aux mots jusqu'à une taille **nbT_m** et de variété jusquà **nbD_m**. La valeur de **alpha_m** sert ensuite à guider la recherche de mots du

dictionnaire qui correspondent à cet ensemble de lettres car l'égalité des valeurs de **alpha** indiquent que deux mots sont des anagrammes l'un de l'autre.

La principale difficulté du projet provient du fait qu'on ne cherche pas un mot *unique* ayant les propriétés **nbT_m, nbD_m** et **alpha_m** mais que plusieurs mots du dictionnaire peuvent se combiner pour obtenir ces valeurs. L'algorithme doit donc construire les ensembles de mots dont la valeur combinée de leur **alpha** est égale à **alpha_m**. Nous simplifions la recherche de ces combinaisons de la manière suivante :

- Un mot ne peut se trouver qu'une seule fois dans une anagramme du message

Même avec cette restriction, il peut exister plusieurs combinaisons de mots qui sont des anagrammes du message.

Exemple A : soit le dictionnaire {DUR, DURE, RUDE, RATION, NOTAIRE} et le message « ORDINATEUR ». L'algorithme doit au moins trouver les combinaisons suivantes : « DUR NOTAIRE », « RATION RUDE » et « RATION DURE ».

De plus, si l'algorithme trouve l'une de ces anagrammes, il n'y a aucune garantie qu'elle soit grammaticalement correcte ou ait un sens. Nous faisons même l'hypothèse que la grande majorité des résultats ne voudra rien dire.

Afin d'être systématique, l'algorithme devra *afficher toutes les permutations des mots de l'anagramme.* Cela veut dire que s'il trouve une anagramme comportant **nbA** mots alors il doit être capable de trouver les **nbA !** permutations de cette anagramme.

Exemple B (Fig 2) : soit le dictionnaire {JE, SUIS, VOLDEMORT} et le message « TOM ELVIS JEDUSOR ». Si l'algorithme trouve l'anagramme « VOLDEMORT SUIS JE» alors il doit être aussi capable de trouver les autres permutations « VOLDEMORT JE SUIS », « JE VOLDEMORT SUIS », « JE SUIS VOLDEMORT», « SUIS JE VOLDEMORT » et « SUIS VOLDEMORT JE ».

Elément1 de la solution : la première clef du fonctionnement de cet algorithme est la fonction `message_contient_mot`() qui indique si la valeur du paramètre **alpha** du mot du dictionnaire *est incluse* dans la valeur du paramètre **alpha_m**. Cette fonction renvoie un booléen **vrai** si c'est le cas est **faux** sinon. Pour renvoyer **vrai** il faut que le nombre de chaque lettre de **alpha** soit *inférieur ou égal* au nombre de cette lettre dans **alpha_m**.

Suite de l'exemple B : soit le dictionnaire {JE, SUIS, VOLDEMORT} et le message « TOM ELVIS JEDUSOR ». La valeur **alpha_m** est donc «DEEIJLMOORSSTUV ». La fonction renvoie vrai pour la valeur **alpha** « EJ » du mot « JE » du dictionnaire car cette valeur est incluse dans **alpha_m**.

Elément2 de la solution : si `message_contient_mot`() renvoie **vrai** pour les paramètres **alpha** et **alpha_m** alors la seconde clef du fonctionnement de l'algorithme est la fonction `message_soustraire_mot`() qui enlève les lettres de **alpha** du paramètre **alpha_m**. Ceci est nécessaire afin de poursuivre cette recherche de mots du dictionnaire dans ce qui reste du message.

Suite de l'exemple B : Si on enlève « EJ » de «DEEIJLMOORSSTUV » on obtient «DEILMOORSSTUV ».

Les deux éléments ci-dessus sont des briques de base d'une approche bottom-up (section 2). L'élément de nous fournissons maintenant est de nature top-down ; on s'intéresse à la structure de l'algorithme principal. Il s'agit de déterminer de manière systématique toutes les combinaisons du dictionnaire qui correspondent au message et de les faire afficher.

Elément3 de la solution:
La recherche de solution réduit la taille du dictionnaire pour chaque mot trouvé dans le message et poursuit la recherche sur le reste des mots et la valeur réduite de **alpha_m**.

Important : le traitement de chaque mot du dictionnaire doit être ***indépendant*** de celui des autres mots ; c'est pourquoi on crée de nouvelles variables (se terminant par **_next**) pour poursuivre la recherche de manière récursive. En voici le pseudocode :

```
---------------------------------------------------------------------
rech_anagramme
// entrée :
// dico : liste du dictionnaire trié (complet au premier appel)
// alpha_m : chaine de caractère triée (du message au premier appel)
// anagramme : liste des mots constituant un début de solution (vide au premier appel)
// sortie : booléen de succès ou d'échec de la recherche d'anagramme

taille_dico  <- taille(dico)
Si taille_dico=0
    Sortir  faux // fin de recherche avec échec
Succes <- faux
Pour i de 1 à taille_dico
   Si message_contient_mot(dico[i].alpha, alpha_m)
      anag_next         <- ajouter(anagramme, dico[i].mot)
      alpha_m_next    <- message_soustraire_mot(dico[i].alpha, alpha_m)
      Si taille(alpha_m_next) = 0
         Afficher(anag_next)
         succes <- vrai
      Sinon
         dico_next <- enlever_mot(dico, i)
         Si rech_anagramme(dico_next, alpha_m_next, anag_next) = vrai
            succes <- vrai
Sortir succes
------------------
```

Remarque : il est pertinent d'effectuer le premier appel de cette fonction récursive avec un dictionnaire dont on a enlevé tous les mots de taille supérieure ou égale à **nbT_m**. Vous pourrez mettre en place d'autres optimisations mais le barème porte seulement sur le fonctionnement correct de la version la plus simple dans le temps alloué par l'autograder.

Exemple : poursuivons avec le dico {JE, SUIS, VOLDEMORT} et le message « TOM ELVIS JEDUSOR » dont la valeur **alpha_m** est «DEEIJLMOORSSTUV ». La figure de la page suivante détaille tous les appels récursifs avec leur paramètres (de la gauche vers la droite).

Le premier appel est effectué avec une liste vide pour le paramètre anagramme. Dans cet exemple les 3 mots sont contenus dans le message et produisent une condition vraie ; toutes

les permutations sont trouvées par l'algorithme en trois niveaux d'appels. Les affichages sont effectués au troisième niveau quand **alpha_m** devient vide.

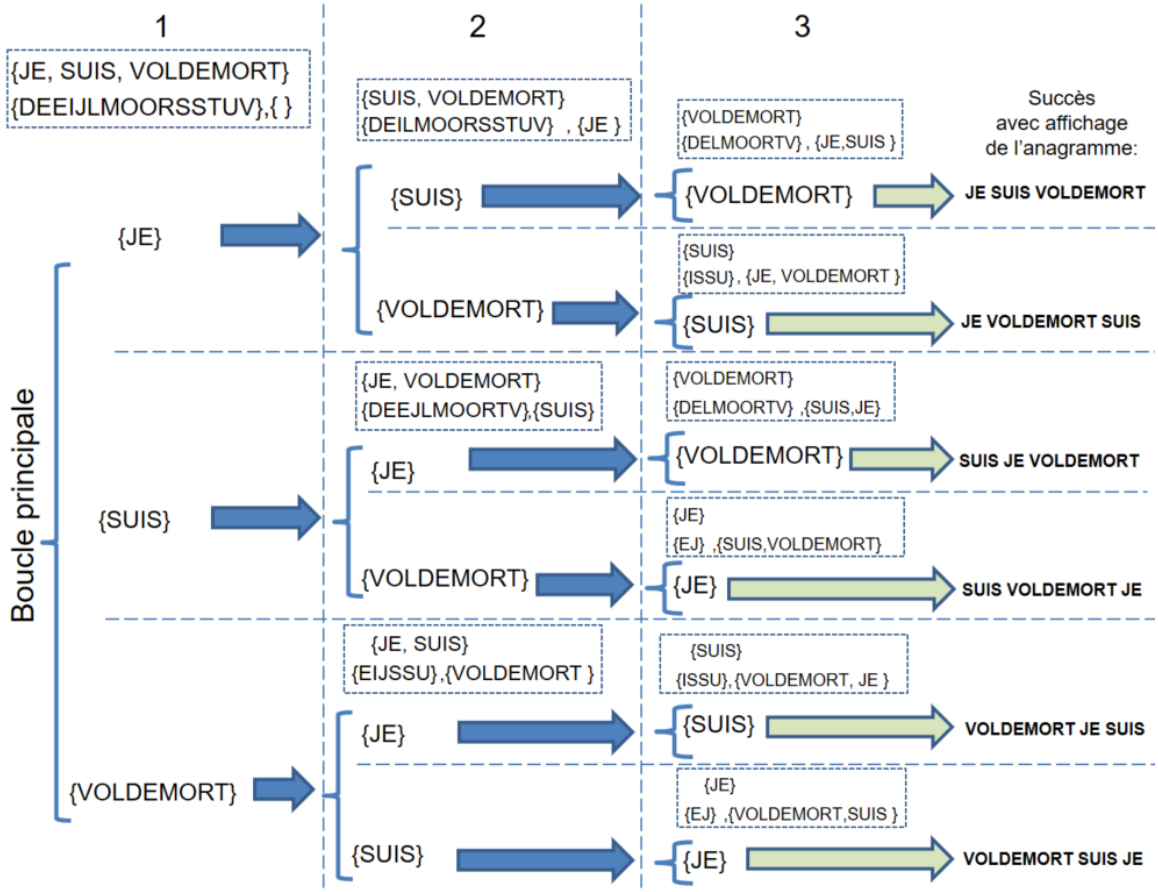


Figure 2 : exécution du pseudocode sur l'exemple B ; la colonne de gauche décrit le premier appel ; ensuite chaque appel récursif est indiqué par une flèche bleue. Les flèches vertes indiquent un succès dans la recherche d'anagrammes.

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++11
  - Votre code sera compilé avec l'option **`-std=c++11`**
- Tout votre code doit être écrit dans un seul fichier source.
- Il faut utiliser **string** pour les manipulations des mots
- Il faut utiliser **vector** pour les tableaux/listes de structures/mots
  - On ne peut **PAS** utiliser les méthodes **insert, erase** de **vector** ; il faut écrire les fonctions correspondantes vous-même si vous en avez besoin.
  - On ne peut **PAS** utiliser **qsort** ni les méthodes de **<algorithm>**

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 et autograder:

La donnée identifie plusieurs phases de tests (sections 4.5) pour valider votre programme. Nous vous fournissons plusieurs fichiers de tests téléchargeables depuis le site de l'autograder (section 4.4.2). A vous de créer vos propres fichiers plus élaborés ou juste différents. Il suffit d'ouvrir geany pour écrire les données et de sauvegarder avec l'extension **.txt** .

4.4.1 Message d'erreur (Fig 3)

Le fichier **error.txt** contient les définitions des constantes des messages d'erreur. Il faut les copier en début de votre fichier source avant la première définition de fonction. Nous testerons seulement ces erreurs sur vos projets : *dico vide, dico avec mot en double* et *lettres pas en majuscule dans dico ou dans le message*. Exemple : supposons qu'on ait trouvé qu'un mot lu pour le dictionnaire ou pour le **message** n'est pas totalement en majuscules. L'instruction d'affichage doit être : (où **`NOT_IN_CAPITAL_LETTERS`** est recopié de **error.txt**)

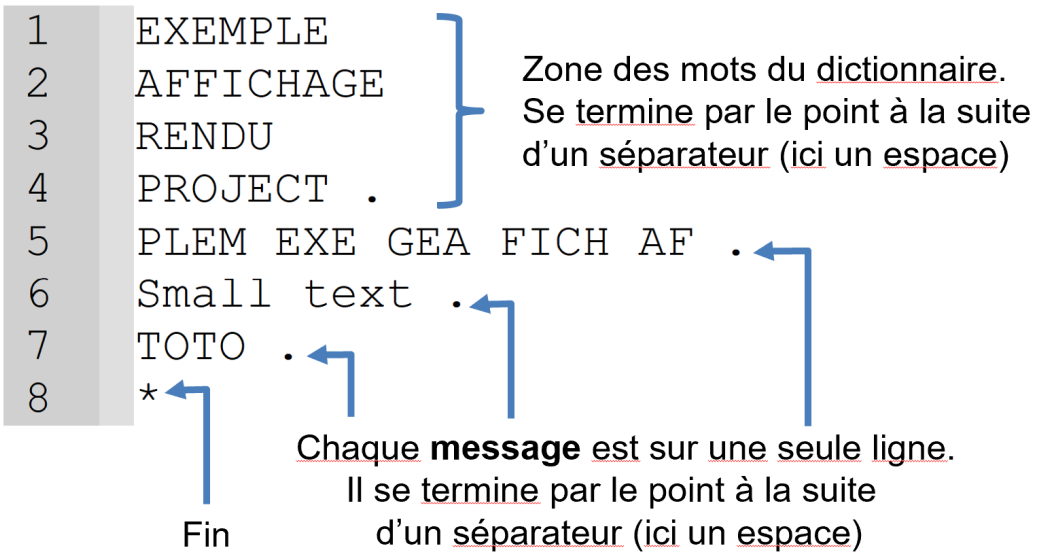
```
cout << NOT_IN_CAPITAL_LETTERS << endl;
```

Si l'erreur est dans le dictionnaire, il faut ensuite quitter le programme avec : **`exit(0);`**

Le cas où il n'y a aucune anagramme dans **message** sera aussi testé. On doit alors afficher NO_ANAGRAM :

```
cout << NO_ANAGRAM << endl ;
```

[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*.



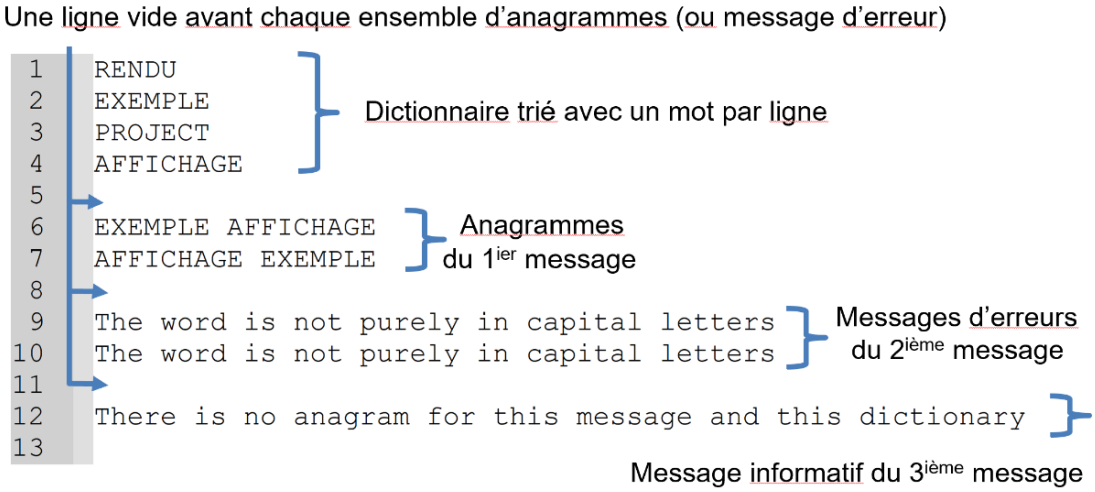


Fig 3 : (a) exemple de Fichier de données fournies en entrée
(b) affichage attendu pour ces données

4.4.2 Vérification anticipée de l'exécution de votre projet avec l'***autograder***

Un outil de type *autograder* a été mis au point pour vous permettre d'évaluer, avant la date du rendu, si votre projet s'exécute comme demandé sur l'ensemble des fichiers de test publics et non-publics qui seront utilisés pour la notation du projet. Le lien et son mode d'emploi sont visibles sur moodle. Vous y trouverez aussi **error.txt** et le lien vers le site officiel du téléversement final sur moodle.

Attention : l'autograder efface systématiquement tous les fichiers archive qui lui sont soumis. De ce fait il ne peut pas constituer une preuve que votre projet a été effectué. Votre projet devra impérativement être téléversé sur moodle comme expliqué dans la section 5.

L'autograder sera utilisé pour noter l'exécution de votre rendu de projet.

4.5 ACTIONS proposées pour organiser votre travail

Les sous-sections « ACTIONS » sont seulement des suggestions pour organiser votre travail.

4.5.1 Tâche1 : Lecture sur l'entrée standard pour construire le dictionnaire trié

Le nombre de mots du dictionnaire n'est pas connu à l'avance (section 3.1.1). Les mots sont ajoutés au dictionnaire un à un, s'ils sont entièrement en majuscule et qu'ils n'y sont pas déjà présents. Il faut afficher le message d'erreur prédéfini que vous trouvez dans **error.txt** puis quitter le programme dès le premier mot qui ne respecte pas ces conditions.

4.5.1.1 ACTIONS 1 : test de la lecture des mots du dictionnaire puis affichage

Concevez une première version du programme qui gère seulement la lecture et la détection d'erreur sur les mots jusqu'au signal de fin de lecture (caractère '.' isolé). On ne fait pas le tri du dictionnaire à cette étape de test ; testez seulement s'il est vide.

Dans ce premier exercice d'abstraction, rappelez-vous que la fonction principale **main()** joue seulement le role de « table des matières » et est constituée essentiellement d'appels de fonctions. Vous devez donc écrire une fonction qui réalise cette tâche de lecture.

Concernant les déclarations de variables, seules les variables stratégiques, telles que le dictionnaire, ont le droit d'exister au niveau de **main().** Les variables utilisées temporairement par une tâche doivent être définies localement dans la fonction qui gère cette tâche. Une fonction peut modifier un paramètre passé par référence ou renvoyer une valeur lue.

Cette version minimal de votre programme pourrait aussi appeler une fonction **d'affichage** du dictionnaire (un mot par ligne) après l'appel de la fonction de lecture dans main(). Le dictionnaire initialisé par la lecture est transmis à cette fonction. Testez cette première version en tirant parti de la redirection des entrées-sorties (série 6) pour couvrir une grande variété de cas (liste des mots avec différentes sortes de séparateurs). Créez vos fichiers tests avec geany.

4.5.1.2 ACTIONS 2 : test du tri du dictionnaire

La fonction réalisant cette tâche de tri utilise le dictionnaire construit par la tâche 1. Elle est appelée dans main() entre la fonction de lecture et celle d'affichage déjà réalisées. Rappelons que pour ce projet d'automne, on ne peut pas utiliser qsort ni le tri de <algorithme>.
Nous recommandons d'utiliser un **vector de structures** pour représenter le dictionnaire car le tri utilise 4 propriétés distinctes : longueur du mot **nbT**, variété des lettres du mot **nbD**, lettres triée dans **alpha** et le **mot** lui-même.

Le tri par insertion vu en cours peut servir de point de départ même si son ordre de complexité est en $O(nbM^2)$ car ce coût n'est pas le coût dominant du projet.

La relation d'ordre des 4 critères de tri (section 3.1.1.1 p5) est aussi mise en oeuvre avec les opérateurs *inférieur* ou *supérieur* pour obtenir l'ordre alphabétique sur des chaînes de caractères. Par exemple, si la **string s1** contient CAMION et la **string s2** contient TRAIN, alors **s1 < s2** vaut **vrai** car CAMION se trouve avant TRAIN dans l'ordre alphabétique.

Vérifiez d'abord que votre algorithme de tri fonctionne correctement avec un seul critère à la fois. Puis combinez deux critères, etc. Dans cette phase *de mise au point*, faites afficher toutes

les variables telles que **nbT, nbD, alpha** en plus du mot lui-même. Le plus important est de tester une grande variété des cas (bien plus que ceux de la page 5).

4.5.2 Tâche2 : Détection d'anagramme

4.5.2.1 ACTIONS 3 : test des éléments 1 et 2 de la solution

Le test des éléments 1 et 2 de la solution est facile car ils sont de bas niveau et travaillent sur des informations élémentaires de type string.

4.5.2.2 ACTIONS 4 : détection d'anagramme

Le pseudocode utilise deux fonctions qui ne sont pas expliquées en détail car elles parlent d'elles-mêmes. L'ajout d'un mot à l'anagramme se fait avec **push_back**() sur le **vector** associé. Par contre le retrait d'un mot du dictionnaire doit être mis en oeuvre par vous-même car pour ce projet d'automne nous n'autorisons pas l'usage de **erase**() sur un vector. Le site wikipedia sur les anagrammes vous donne de nombreux exemples pour tester cette fonction.

4.5.2.3 ACTIONS 5 : boucle de recherche d'anagramme

Après la recherche d'anagrammes sur le **message**, la lecture du caractère '*' comme caractère isolé produit la fin du programme. S'il n'est pas lu, il faut lire un nouveau **message** à analyser. S'il y a détection d'un mot qui n'est pas en majuscule, il faut reprendre la lecture du **message** depuis le début. La mise au point de cette boucle doit prendre soin de ré-initialiser les variables utilisées pour la recherche d'anagrammes.

5. 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 Monsieur 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 **24/12 à midi,** à l'aide du lien sur **moodle** (Topic 14) ; le site de téléversement sera ouvert plusieurs semaines avant la date limite. 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.

5.1 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) 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 4), la mise en oeuvre des principes *d'abstraction* et de *ré-utilisation* dans votre projet. Concentrez-vous

sur la décomposition du *tri hiérarchique* puisque vous disposez déjà de la décomposition de la recherche d'anagramme avec le pseudocode (p8).

b) **Pseudocode** de votre **algorithme** effectuant le tri hiérarchique du dictionnaire (PAS de code source). Le pseudocode doit être en entier sur une seule page (soit sur la p 1 ou la p 2 mais pas entre les deux).

c) Quel est l'ordre de complexité de l'algorithme de détection d'anagramme (pseudocode p8) en fonction de **nbM**, le nombre total de mots du dictionnaire ? Pourquoi ? Soyez bref.

d) A partir des temps d'exécution « **user** » (série 6) sur les fichiers publics dont le nom contient **perf1.txt, perf2.txt, perf3.txt,** fournir le graphique de ces temps (abscisse linéaire, ordonnée logarithmique) en fonction de **nbM**. Est-ce cohérent avec la réponse précédente ?

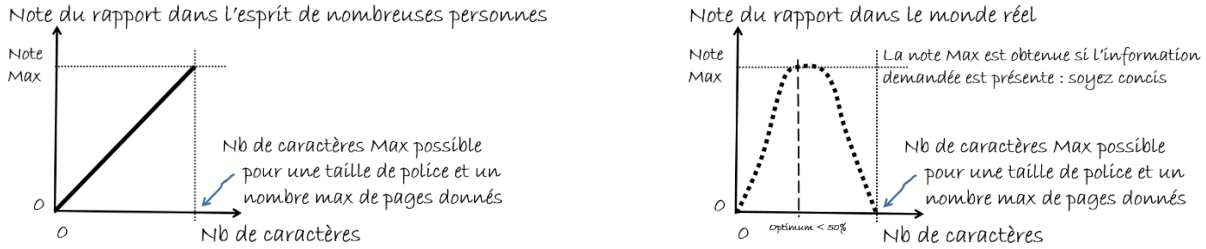


Figure 4 : un mauvais rapport dilue l'information utile jusqu'à atteindre le nombre maximum de caractères possibles sur la page (à cause de la croyance populaire de la figure de gauche) : dans la réalité ce type de rapport très compact sera très pénalisé parce qu'il est peu lisible (figure de droite) ; un bon rapport est celui qui fournit les informations demandées avec concision avec une mise en page aérée et lisible

6. Barème indicatif (12pts):

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

(2pt) rapport : soyez concis (Fig 4)
(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 ; 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.