

Algorithmique 2

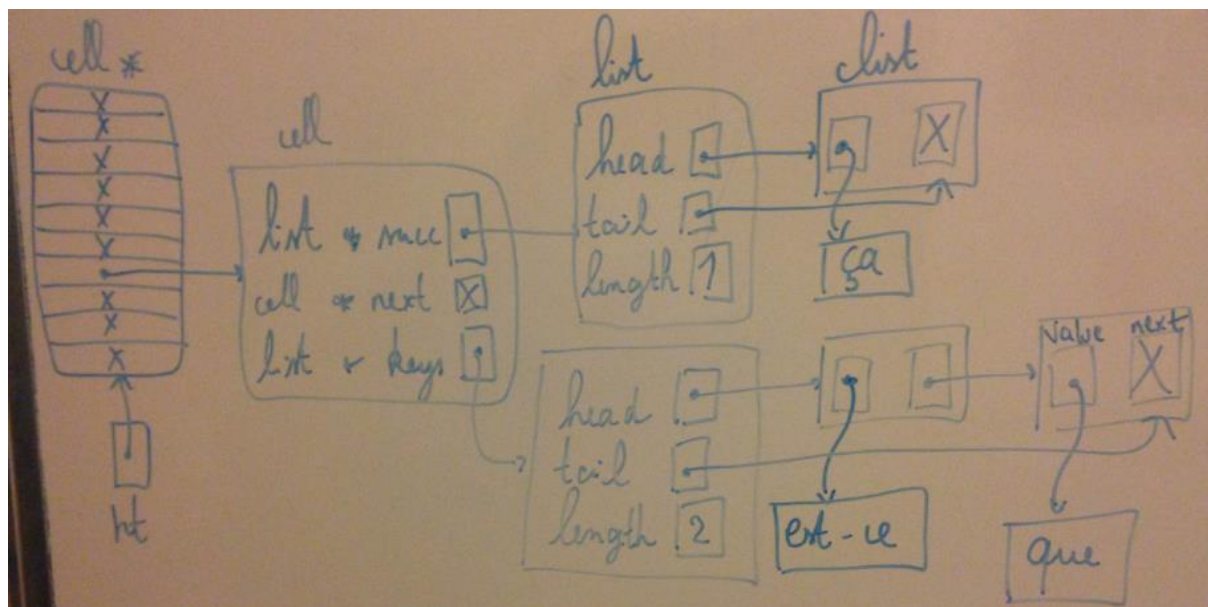
GENEREUR DE TEXTE ALEATOIRE

OUATTARA UMM-HABIBAH

L'objectif de ce projet est de générer à partir d'un texte source un texte aléatoire. Pour ce faire, j'ai décidé d'utiliser une liste simplement chaînée. J'ai opté pour cette solution car j'ai trouvé que c'était plus facile à implanter et peut être aussi plus logique au vu de l'algorithme donné dans le sujet. Cependant, niveau performance ce n'est pas vraiment la meilleure des solutions surtout pour des textes assez volumineux.

Structure

Ma structure de donnée se présente ainsi :



➤ hashtable :

Structure prise en charge par le module hashtable.h opte pour une implantation par chaînage séparée.

A l'initialisation de la table de hachage, la fonction de comparaison, premier paramètre de la fonction *hashtable_empty* est *lists_compar*. Elle donne la possibilité de comparer les éléments de deux listes. Nécessaire au bon fonctionnement du programme, elle a été modifiée pour qu'elle puisse être utilisée pour les listes. La version originale¹ ne s'intéressant qu'aux chaînes de caractères. Cependant, elle reprend le même principe de comparaison usuel. *Lists_value_hashfun*, deuxième paramètre, est une autre fonction importante dans le projet, car c'est la fonction de pré-hachage, conseillée par Kernighan et Pike. Elle a été modifiée² aussi pour être appliquée aux listes.

Quand une a été trouvée, on crée une structure de type cell, qui contient une liste associée aux successeurs et une autre liée aux clés. La valeur renvoyée au moment de

¹ Version de l'enseignement et du TP n°8

² Version originale sur le support de cours p.92

la recherche d'une clé est stockée dans la liste des successeurs. Au moment de l'ajout dans la table de hachage, on passe en paramètre de la fonction *hashtable_add*, la table de hachage correspondante, la clé et sa valeur.

➤ **lists** :

Cette structure permet la gestion des listes simplement chaînées de successeurs et de clés. Elle est prise en charge par le module *lists.h*.

Lors de l'initialisation de la table de hachage, on alloue une structure de type *lists*, dans laquelle le pointeur de tête et de queue valent NULL et où la longueur vaut 0. Cette structure, va servir à la gestion des successeurs. La liste concernant les clés, est quant à elle gérée par la fonction *lists_key_empty*. Elle donne la possibilité d'avoir une liste d'une certaine longueur, n , et de contenu quelconque. Comme je n'ai pas utilisé de tableau, la longueur de la liste doit être modifiée à chaque opération d'ajout et de suppression d'objet. C'est peut-être, le plus gros désavantage d'une liste comparé au tableau, car alloué dynamiquement, sa longueur est automatiquement modifiée.

En ce qui concerne l'affichage. Je ne me préoccupe simplement que de savoir si ma tête de la liste est NULL ou pas. Grâce à une boucle while sur la tête, j'affiche le champ value entre crochet suivi d'une tabulation. A chaque tour de boucle, la tête pointera vers le champ next de la cellule suivante. Le jour où on sort de la boucle, c'est que le next vaut NULL et que c'est la fin de la liste.

Pour pouvoir remplacer la liste définie par u_0, u_1, \dots, u_{n-1} par u_1, \dots, u_{n-1}, v , j'ai eu l'idée de définir une fonction qui duplique une liste. A la nouvelle liste ainsi obtenue, on lui supprime l'élément de tête, donc le u_0 et on lui ajoute un objet en queue, le v . Je pense qu'il y avait peut-être une autre solution, que je n'ai malheureusement pas trouvée, car elle est assez lourde pour le programme.

➤ **holdall** :

Ce module, qu'on appelle fourre-tout, pris en charge par le module *holdall.h*³, est une implantation par liste simplement chaînée. Permet essentiellement de garder toutes les allocations faites et de les libérer à la fin de l'exécution du programme. C'est pour cela que dans le programme j'utilise 3 listes de type *holdall*.

- *hashword* : concerne tous les mots lus sur l'entrée
- *hashkey* : pour la liste de clé non répétitives
- *hahslist* : pour la liste de successeur et les clés répétées

Ces trois listes sont créées avec la fonction *holdall_empty*. De cette interface, je n'utilise que les fonctions *holdall_put*, *holdall_apply_context* et *holdall_dispose*. Elles

³ Donner lors de l'enseignement

se chargent respectivement d'ajouter au fourre-tout dans les listes plus haut, des objets en tête de ces listes, d'exécuter une fonction sur le fourre-tout hashkey (fonction d'affichage dans le programme) et enfin de libérer l'espace pointé par les objets holdall quand le programme se termine.

Modules

Chaque structure a un certain nombre de fonctions utiles et utilisées. Les fonctions présentées ci-dessous sont celles qui j'ai été utilisées lors du projet.

➤ hashtable.h :

Gère la table de hachage. On y retrouve les fonctions suivantes :

- hashtable_empty : crée la structure de données de la table de hachage vide, la fonction de comparaison est compar et hashfun, la fonction de hachage.
- hashtable_add : ajoute dans la table de hachage le couple (clé/valeur) si elle n'existe pas sinon remplace la valeur de la clé par la nouvelle valeur
- hashtable_value : recherche dans la table si une clé est égale au sens de la fonction de comparaison à la clé passé en paramètre
- hashtable_dispose : libère l'espace pointé par la variable qui gère la table de hachage

➤ holdall.h :

Gère le fourre-tout avec implantation d'une liste simplement chaînée et insertion en tête. Crée une structure avec la fonction holdall_empty. On y retrouve la fonction utilisées dans le programme :

- holdall_dispose : qui libère les espaces alloués
- holdall_put : qui ajoute en tête un objet à la liste

➤ lists.h :

Se charge de la structure de données associées aux listes de successeurs et de clés. L'implantation est simplement chaînée. Sa création est donnée par lists_empty. On y retrouve les fonctions :

- lists_is_empty : renvoie true ou false selon que la liste pointée par s est vide ou non
- lists_length : renvoie la longueur de la liste associée à s
- lists_key_empty : qui crée une liste vide de longueur n et de valeur pointé par ptr, elle renvoie un pointeur vers la liste créée ou NULL en cas d'échec
- lists_insert_tail : qui insère en queue une copie de ptr dans la liste pointé par s, elle renvoie 0 si l'opération c'est bien passé ou sinon une valeur négative

- lists_compar : permet de comparer les éléments de la liste pointée par s1 avec ceux de la liste pointée par s2. 0 si les éléments sont égaux, une valeur négative si le premier est inférieur au second et une valeur positive s'il est supérieur
- lists_display : fonction d'affichage ; affiche sur le flot pointé par stream les valeurs de la liste associée à s
- lists_value_hashfun : adaptation de la fonction de pré-hachage de Pike aux listes
- lists_index_to_value : permet d'avoir la valeur de l'élément d'indice k de la liste associée à s
- lists_new_createdlist : cette fonction me permet de créer une nouvelle liste en dupliquant la liste pointée par s, elle supprime le premier élément de la liste dupliqué et lui ajoute un objet ptr en queue.

Sans oublier la fonction dispose qui libère les espaces alloués.

Après avoir défini toutes les fonctions nécessaires du module lists.h, j'ai pu implanter le main.c

Implantation

➤ aide :

Pour implanter l'aide, j'ai décidé de définir une structure énumérée, nommée option, dans laquelle sont stockées les options demandées : -h, -t, -l, -n et -s. Elles permettent respectivement d'afficher l'aide, d'afficher sous la forme d'une table le couple (clés/valeur), de choisir le nombre de mots à produire, de changer l'ordre de la génération des clés et enfin de modifier le germe de la génération pseudo-aléatoire.

```
typedef enum {
    OPT_HELP, OPT_TABLE, OPT_LIMIT, OPT_ORDER, OPT_SEED,
    OPT__NBR
} option;
```

Le paramètre OPT__NBR permet de savoir combien d'option sont possibles d'avoir. Elle permet de pouvoir vérifier la validité de l'option choisit. En effet, en définissant une autre structure contenant un tableau de nom, opt_value, pour chaque option, et de dimension OPT__NBR, qui prend en paramètre un label et une description

```
struct {
    char *label;
    char *descr;
} opt_value[OPT__NBR] = {
    [OPT_HELP] = {
        .label = "-h",
        .descr = "display the help for this program"
    },
    ,
```

j'ai pu grâce à une boucle while, comparer l'argument, définit en ligne de commande et le label de la structure.

```
while (k < OPT__NBR && strcmp(a, opt_value[k].label) != 0)
```

A noter que $a = \text{argv}[k]$, où $\text{argv}[k]$ est le tableau de caractère contenant les options. Si on ne trouve pas l'option écrite, on affiche une erreur disant que l'option voulu n'existe pas et une aide.

Pour afficher l'aide à proprement parler, on parcourt le tableau de chaîne de caractère pointé par argv , et on compare à l'aide de la fonction de comparaison `strcmp`, si on a une chaîne de caractère de la forme « -h », si c'est le cas on affiche l'aide grâce à la fonction `display_help`. Pour ce qui est de l'affichage de la table du couple (clé/valeur), j'ai une variable booléenne initialisée à false au début du programme et qui quand la fonction de comparaison `strcmp(argv[k], "-t")` vaut 0, elle passe à true.

La gestion des autres options « -l », « -n » et « -s » est un peu différente, car elles prennent aussi un entier. Elles utilisent le même principe, donc l'explication suivante donner pour « -l » vaut aussi pour les deux autres.

Prenons par exemple, « -l », qui permet de changer le nombre de mots à produire. J'ai décidé de produire 230 mots au lieu des 100 par défaut. Pour cela, j'ai fait en sorte que le « 230 » soit une commande à part entière c'est-à-dire qu'il faut que je valide le fait qu'o est bien un entier. Par conséquent, j'ai utilisé la fonction de conversion d'un entier dans une base `strtol` qui prend en paramètre un pointeur, donc ici $\text{argv}[k+1] = 230$, un pointeur de pointeur, ici `char *endptr` et la base, ici 10. Le résultat sera stocké dans une variable de type long int : x .

```
x = strtol(argv[k+1], &endptr, 10);
```

Si on a réussi la conversion on affecte à la variable correspondant au nombre de mots à afficher x

```
m = (int) x;
```

Ce qui est dommage, c'est que je n'ai pas réussi à factoriser le code de la gestion de ces trois options.

➤ fonction principale :

1. initialisation des variables nécessaire au programme

```
hashtable *ht = hashtable_empty((int (*)(const void *, const void *))
    lists_compar, (size_t (*)(const void *)) lists_value_hashfun);
holdall *hashword = holdall_empty();
holdall *hashkey = holdall_empty();
holdall *hashlist = holdall_empty();
// d = u0,u1,...,un-1
char *d = " ";
// keys : liste de n-uplet de mots initialisé à vide
lists *keys = lists_key_empty((size_t) n, d);
```

```

char string[STRING_LENGTH_MAX + 1];
// v : mot qui sera utilisé pour former la clé suivante dans la liste
// associée à la clé(u0,u1,...,un-1)
char *v;
// succ : liste associée à celle des clés(u0,u1,...,un-1)
lists *succ;

```

Si une des allocations ne s'est pas bien effectué, on va directement à l'étiquette `error_capacity` et on libère avec `dispose` les éventuelles espaces alloués.

```

if (ht == NULL || hashword == NULL || keys == NULL || hashkey == NULL ||
    hashlist == NULL) {
    goto error_capacity;
}

```

2. On vérifie si une clé existe déjà.

```

succ = (lists *) hashtable_value(ht, keys);

```

Si la clé n'est pas dans la table alors `succ` prendra la valeur `NULL` et dans ce cas on doit créer une liste de successeur à vide. On prend bien le soin de vérifier si on n'a pu la créer si ce n'est pas le cas on libère la clé pointé par `keys` et on saute à l'étiquette `error_capacity`. Dans le même temps, il faut aussi ajouter dans les fourre-tout adéquat les listes de type `holdall` et toujours vérifier qu'on a pu les ajouter sinon on saute aux étiquettes correspondantes aux erreurs. Si `succ` vaut `NULL` on ajoute dans la table de hachage le couple (clé/valeur).

3. On vérifie si on a encore des mots à lire, si ce n'est pas le cas on donne à la valeur pointé par `v`, qui correspond au mot lu sur l'entrée, vide.

```

if (scanf("%" XSTR(STRING_LENGTH_MAX) "s", string) != 1) {

```

Sinon on continue de lire les mots et on les affecte à `v`. On alloue un espace de la taille d'un mot lu + 1, encore et toujours vérifié si l'allocation s'est bien faite sinon c'est une erreur. On copie avec `strcpy` dans la chaîne pointée par `v` le contenu pointé par `string`. Si on n'a pas pu ajouter dans le fourre-tout associé aux mots lu, on libère l'espace pointé par `v` et on termine le programme avec une erreur.

```

else {
    v = malloc(strlen(string) + 1);
    if (v == NULL) {
        goto error_capacity;
    }
    strcpy(v, string);
    if (holdall_put(hashword, v) != 0) {
        free(v);
        goto error_capacity;
    }
}

```

Comme le mot v est utilisé pour former la clé suivante dans la liste des clés, on remplace dans cette liste u_0, u_1, \dots, u_{n-1} par u_1, \dots, u_{n-1}, v et toujours prendre soin de vérifier qu'on a bien effectué le remplacement

```
keys = lists_new_createdlist(keys, v);
if (keys == NULL) {
    goto error_replaced;
}
```

4. On ajout dans la liste des successeurs le mot v

```
if (lists_insert_tail(succ, v) != 0) {
    goto error_insert;
}
```

On recommence à partir du point 2 tant que le mot v n'est pas vide. Ceci était la première partie de l'algorithme. La deuxième celle qui se gère de la génération aléatoire donne ceci :

1. On choisit aléatoirement un mot dans la liste liée aux successeurs et on l'affecte à v .

```
v = random_select_word(succ, ht, keys);
```

La fonction `random_select_word` permet justement de choisir un mot aléatoire dans la liste en fonction de sa longueur. Elle prend en paramètre la liste des successeurs, la table de hachage et la liste des clés. Elle utilise aussi une autre fonction, qui permet d'avoir la valeur de l'élément d'une liste situé à un certain indice.

```
char *v = (char *) lists_index_to_value(succ, rand() % (int) lists_length(succ));
```

2. Tant que le mot v n'est pas vide et que le nombre de mot produit n'excède pas le nombre de mot par défaut

```
// i == nombre de mots produits
int i = 0;
while ((strcmp(v, d) != 0) && i < m) {
```

Alors on peut afficher les mots aléatoires

```
if (fprintf(stdout, "%s", v) < 0) {
```

3. On refait une duplication de la liste des clés pour pouvoir former une autre clé et on choisit encore une fois un mot aléatoire dans la liste associée à la clé

```
keys = lists_new_createdlist(keys, v);
if (keys == NULL) {
    goto error_replaced;
}
if (holdall_put(hashlist, keys) != 0) {
    lists_dispose(&keys);
    goto error_capacity;
}
```

```
v = random_select_word(succ, ht, keys);
```

4. On n'oublie pas d'incrémenter le nombre de mot à produire

```
++i;
```


La boucle tant que s'arrête dès que les deux conditions de la boucle ne sont pas satisfaites.

- Et on retourne `r`, qui vaut `EXIT_FAILURE` en cas d'erreur ou `EXIT_SUCCESS` si tout c'est bien passé. S'il y a une erreur on va directement à l'étiquette `dispose` ce qui permet de libérer les espaces éventuellement alloués au début du programme : `ht` qui a servi de pointeur vers la table de hachage

dispose:

```
hashtable_dispose(&ht);
```

Pour désallouer les fourre-tout on doit appliquer la fonction `rfree` et `lists_rfree` qui respectivement vont supprimer les valeurs dynamiques pointées par les objets de type `holdall`

```
if (hashword != NULL) {
    holdall_apply(hashword, rfree);
}
if (hashkey != NULL) {
    holdall_apply(hashkey, lists_rfree);
}
if (hashlist != NULL) {
    holdall_apply(hashlist, lists_rfree);
}
```

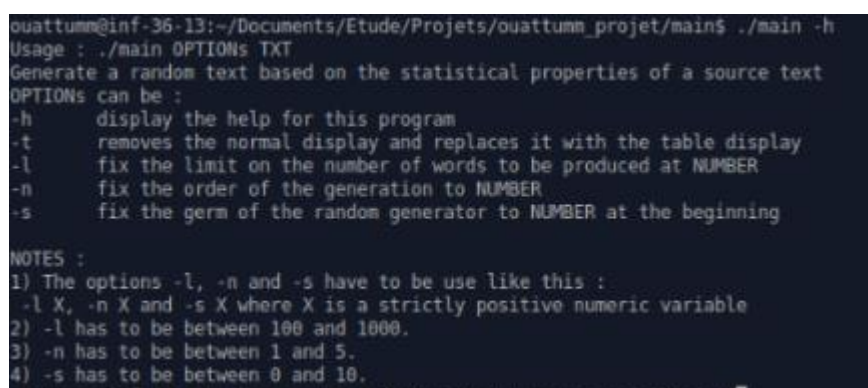
Ensuite on désalloue le tout avec `holdall_dispose`

```
holdall_dispose(&hashword);
holdall_dispose(&hashkey);
holdall_dispose(&hashlist);
```

Exemples

J'ai choisi de montrer un exemple pour chaque option, sur des textes plus ou moins long. Les exemples ont été testés avec `valgrind` sauf l'affichage de l'aide, car je ne pense pas que ça soit nécessaire.

Le premier exemple concerne l'affichage de l'aide. Il n'y a pas grand-chose à dire, l'image parle d'elle-même



```
ouattamm@inf-36-13:~/Documents/Etude/Projets/ouattamm_projet/main$ ./main -h
Usage : ./main OPTIONS TXT
Generate a random text based on the statistical properties of a source text
OPTIONS can be :
-h      display the help for this program
-t      removes the normal display and replaces it with the table display
-l      fix the limit on the number of words to be produced at NUMBER
-n      fix the order of the generation to NUMBER
-s      fix the germ of the random generator to NUMBER at the beginning

NOTES :
1) The options -l, -n and -s have to be use like this :
   -l X, -n X and -s X where X is a strictly positive numeric variable
2) -l has to be between 100 and 1000.
3) -n has to be between 1 and 5.
4) -s has to be between 0 and 10.
```

Le deuxième exemple effectué sur le fichier abcd.txt, se consacre à l’affichage sous forme de table des couples (clés/valeurs). J’ai choisi ce texte pour avoir un affichage assez lisible. On voit bien que cela correspond à la spécification demandé, à savoir une tabulation entre chaque mot. On remarque également que l’ordre de la génération est de 3 et il y a bien 3 clés. Comme j’ai choisi les listes, le nombre d’allocation est déjà assez conséquent : 220. Mais le temps d’exécution est assez convenable je trouve. A noter également qu’il n’y a pas d’erreur, toutes les allocations faites et ont été libérer.

```

ouattumm@inf-36-13:~/Documents/Etude/Projets/ouattumm_projet/main$ time valgrind
./main -t -n 3 < ../textes/abcd.txt
==2659== Memcheck, a memory error detector
==2659== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2659== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2659== Command: ./main -t -n 3
==2659==

[b]    [b]    [a]    [c]
[a]    [b]    [b]    [a]
[c]    [a]    [b]    [b]
[d]    [a]    [c]    [a]
[b]    [d]    [a]    [c]
[a]    [b]    [d]    [a]
[a]    [a]    [a]    [b]
[c]    [a]    [a]    [a]
[a]    [c]    [a]    [a]    [b]    [ ]
[b]    [a]    [c]    [a]    [a]
[a]    [b]    [a]    [c]
[a]    [a]    [b]    [a]    [d]
[ ]    [a]    [a]    [b]
[ ]    [ ]    [a]    [a]
[ ]    [ ]    [ ]    [a]

==2659==
==2659== HEAP SUMMARY:
==2659==   in use at exit: 0 bytes in 0 blocks
==2659== total heap usage: 220 allocs, 220 frees, 13,204 bytes allocated
==2659==
==2659== All heap blocks were freed -- no leaks are possible
==2659==
==2659== For counts of detected and suppressed errors, rerun with: -v
==2659== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

real    0m0.728s
user    0m0.682s
sys      0m0.044s

```

Le deuxième exemple a été effectué sur le fichier assez volumineux, Les Misérables, de Victor Hugo. J’ai choisi d’afficher le texte aléatoire contenant 112 mots, avec l’ordre de la génération modifié à 4 et un germe pseudo-aléatoire à 7. Ce qui donne la capture suivante.

```

ouattumm@inf-36-13:~/Documents/Etude/Projets/ouattumm_projet/main$ time valgrind
./main -l 112 -s 7 -n 4 < ../textes/lesmiserables.txt
==2784== Memcheck, a memory error detector
==2784== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2784== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2784== Command: ./main -l 112 -s 7 -n 4
==2784==
Tome I -- Fantine Livre premier -- Un juste Chapitre I Monsieur Myriel En 1815,
M. Charles-Francois-Bienvenu Myriel etait eveque de Digne. C'etait un vieillard
d'environ soixante-quinze ans ; il occupait le siege de Digne depuis 1806. Quoi
que ce detail ne touche en aucune maniere au fond meme de ce que nous disons ici
. Elle a encore, apres son 1688 et notre 1789, l'illusion feodale. Elle croit a
l'heredite et a la hierarchie. Ce peuple, qu'aucun ne depasse en puissance et en
gloire, s'estime comme nation, non comme peuple. En tant que peuple, il se subo
rdonne volontiers et prend un lord pour une tete. Workman, il se laisse dedaigne
r ; soldat, il
==2784==
==2784== HEAP SUMMARY:
==2784==    in use at exit: 0 bytes in 0 blocks
==2784== total heap usage: 6,860,195 allocs, 6,860,195 frees, 133,853,151 byte
s allocated
==2784==
==2784== All heap blocks were freed -- no leaks are possible
==2784==
==2784== For counts of detected and suppressed errors, rerun with: -v
==2784== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

real    0m24.201s
user    0m23.548s
sys     0m0.648s

```

On voit que le nombre d'allocations est exorbitant : 6, 860, 195 mais elles ont été toutes libérées. C'est pourquoi je pense que même si l'implantation par liste simplement chaînée a été assez facile, l'utilisation d'un tableau aurait été beaucoup plus judicieuse, car le nombre d'allocation serait beaucoup moindre. On peut noter aussi que les textes ne sont pas si aléatoires que ça. Soit cela vient d'une mauvaise gestion de ma part soit cela montre que c'est vraiment pseudo-aléatoire.

Pour ce dernier jeu d'exemple on revient à l'aide avec la gestion des erreurs possibles avec arrêt du programme, tester cette fois-ci avec valgrind. Ce qui correspond aux spécifications usuels pour l'aide.

```

ouattumm@inf-36-12:~/Documents/Etude/Projet_S03/ouattumm_projet/main$ valgrind .
./main -t -n 6 -l 50 < ../textes/knock.txt
==2881== Memcheck, a memory error detector
==2881== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2881== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2881== Command: ./main -t -n 6 -l 50
==2881==
*** Error: order '6' is out of the limits.
You can try « ./main -h » for more information.
==2881==
==2881== HEAP SUMMARY:
==2881==    in use at exit: 0 bytes in 0 blocks
==2881== total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==2881==
==2881== All heap blocks were freed -- no leaks are possible
==2881==
==2881== For counts of detected and suppressed errors, rerun with: -v
==2881== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Limites

- Je n'ai pas g rer si le mot lu d passe la longueur d fini par la macro constante `STRING_LENGTH_MAX`
- Je n'ai pas pris en compte la possibilit  qu'il y ait plusieurs fois la m me option.
- Les limites des options « -l », « -n » et « -s » sont d finis par des macros constantes

Difficult s rencontr es

- J'ai eu du mal   faire le lien entre la table de hachage et les (cl s/valeurs) donc du mal   avoir une structure de donn es.
- J'ai eu du mal  galement dans la mise en place de certaines fonctions dans mon module `lists.h` notamment la fonction de duplication.
- J'ai voulu au d but utiliser un tableau mais la structure que j'ai obtenue n' tait pas bonne du coup j'ai d cid  d'utiliser des listes.
- Probl me aussi au niveau de la factorisation pour les options, je n'ai pas pu le faire et c'est dommage car il y a du code redondant