

Astuces 42 par quelqu'un qui l'a pas fait

Julia PHAM BA NIEN

23 février 2024

Table des matières

1	Les fichiers	3
2	La norminette	4
2.1	Réduire le nombre de lignes	4
2.1.1	Introduction de variables	4
2.1.2	Variables constantes	5
2.1.3	Opération prefixe et postfixe	5
2.1.4	Tail Call Optimisation	5
2.1.5	Fonctions prenant des pointeurs	6
2.1.6	Multiple expressions en une seule ligne	6
2.2	Problèmes de mémoires	8
2.2.1	Ne pas allouer	8
2.2.2	Lifetime et ownership	8
2.2.3	Arena Allocators	9
3	Structures de données	11
3.1	Fat pointer	11
3.2	Linked list	12
3.3	Vector	13
4	Algorithmes types	14
4.1	Puissance	14
4.2	memset	15
4.3	Résolution d'un sudoku	16
4.4	Evaluation d'une position au morpion	16
5	Ressources externes	18

1 Les fichiers

Comme à 42 on n'a pas le droit à la stdlib C, il faut la recréer, ce qui inclue stdio.

Pour avoir plus d'info sur les fonctions que je vais lister, faites

```
man 2 <nom de la fonction> # si syscall comme "open"
man 3 <nom de la fonction> # si partie de la stdlib comme "memset"
```

Tout fichier est représenté par un nombre (int), son "file descriptor" qui va être abrégé "fd" un peu partout.

Il y a l'entrée standard (stdin) ayant pour fd 0 (sous unix), la sortie standard (stdout) ayant pour fd 1 et la sortie d'erreur (stderr) ayant pour fd 2.

Vous pouvez mettre ça pour aider la lisibilité (mais on peut directement remplacer par ses valeurs, c'est pas grand chose à apprendre) :

```
1 #define STDIN 0
2 #define STDOUT 1
3 #define STDERR 2
```

Pour obtenir le fd d'un fichier dont on a le chemin (relatif ou absolue), on utilise la fonction/syscall open.

```
1 #include <fcntl.h>
2 const char *file_path = "le fichier";
3 // pour lire le fichier
4 int fd = open(file_path, O_RDONLY);
5
6 // pour écrire sur un fichier
7 int fd = open(file_path, O_WRONLY);
8
9 // il faut fermer le fd après l'avoir open
10 close(fd);
```

Pour écrire ou lire sur un fichier, il faut avoir une zone qui contient la mémoire ou qui peut la contenir, on utilise "write" et "read"

```
1 #include <unistd.h>
2
3 char buf[1000];
4
5 write(STDOUT, "hello, world\n", 13); // écrit 13 caractères de "
    hello world\n" dans le stdout
6
7 read(STDIN, buf, 100); // lis 100 caractères du stdin dans buf
```

Ces fonctions retournent le nombre d'octets effectivement lu/écrits.

Pour lire sur un fichier (pas le stdin) on peut retourner en arrière ou en avant avec lseek.

```
1 #include <unistd.h>
2
3 off_t lseek(int fd, off_t offset, int whence);
```

où whence est soit SEEK_SET (le premier caractère), SEEK_CUR (là où on est dans la lecture), SEEK_END (la fin du fichier) et retourne l'offset depuis le début du fichier.

2 La norminette

La norminette, le truc pour te faire écrire du bon code qui t'oblige à faire des affreusités, un bon exemple de la loi de Goodhart : https://fr.wikipedia.org/wiki/Loi_de_Goodhart

Les exemples que je vais présenter sont assez courts pour que la norminette ne pose pas de problèmes, mais ce qui est important est que les idées peuvent se retranscrire sur des fonctions plus longues.

Ces techniques ont tendance à rendre le code "moins bon", donc à utiliser que lorsque cela est nécessaire.

2.1 Réduire le nombre de lignes

De ce que j'ai entendu, un des plus gros problèmes est d'écrire des fonctions assez petites.

2.1.1 Introduction de variables

On va essayer de réécrire le code suivant :

```
1 int iterfib(int n) {
2     int a;
3     int b;
4     int c;
5
6     a=1;
7     b=0;
8
9     while (n--) {
10         c = a;
11         b = a;
12         a += c;
13     }
14     return b;
15 }
```

On remarque qu'on as besoin de 3 int, on peut donc le réécrire :

```
1 int iterfib(int n) {
2     int t[3];
3
4     t[0]=1;
5     t[1]=0;
6
7     while (n--) {
8         t[2] = t[0];
9         t[1] = t[0];
10        t[0] += t[2];
11    }
12    return t[1];
13 }
```

Ce qui sauve 2 lignes !

Mais ce tricks semble ne marcher seulement pour les éléments du même type, mais avec un peu de connaissance sur la mémoire on peut le faire pour tous. On peut transformer

```
1 int a;
2 float b;
3 double c;
4 char d[8];
5 long long *e;
```

En

```
1 char t[ sizeof(int) + sizeof(float) + sizeof(char d[8]) + sizeof(
    long long *) ];
```

(on peut remplacer par les valeurs si les contraintes des 80 caractères embête)
Et alors

```
1 a = *((int *) t);
2 b = *((float *) (t + 4)); // la taille de a
3 c = *((double *) (t + 8)); // la taille de a et b
4 d = (char *) (t + 16); // la taille de a et b et c, on ne dereference
    pas
5 e = *((long long *) (t + 24)); // la taille de a et b et c et d
```

Ce qui avec une macro se fait

```
1 #define GET(type, mem, offset) (*((type *) (mem + offset)))
```

(mais les macros c'est interdit donc l'utilisez pas)

2.1.2 Variables constantes

Lorsque qu'on marque une variable en "const", on ne peut pas modifier sa valeur. Donc on ne peut pas l'introduire, puis l'initialiser plus tard.

Ainsi, on a le droit de faire une introduction et initialisation en même temps, sur la même ligne!

Un bon exemple est la fonction memset 15

2.1.3 Opération prefixe et postfixe

Utiliser (et abuser) ++ et le double - postfixe, c'est bien.

Néanmoins la norminette empêche d'avoir des gros abus.

2.1.4 Tail Call Optimisation

Tout lisper est sensé le savoir, mais peu sont des lispers...

Si on a le code :

```
1 return_t function(args...) {
2     if (cond(args...))
3         return g(args...);
4     return function(f(args...));
5 }
```

Ce code est équivalent (à usage de stack près) à :

```

1 return_t function(args...) {
2     while (!cond(args...))
3         args... = f(args...);
4     return g(args...);
5 }

```

Normalement, on part de la première version vers la seconde, car elle n'utilise pas autant le stack et est plus rapide (et ne va pas StackOverflow).

MAIS, la première version prends moins de lignes, SURTOUT si il y a plusieurs variables et la transformation "args... = f(args...)" modifie beaucoup de variables interliées.

Ne pas utiliser si on s'attends à ce qu'il y ait plus d'un million d'appels récursifs pour éviter les StackOverflow.

Exemple avec fibonnacci :

```

1 int fib(int n) {
2     int a;
3     int b;
4     int c;
5
6     a = 1;
7     b = 0;
8
9     while (n--) {
10        c = b;
11        b = a;
12        a += c;
13    }
14    return b;
15 }

```

Qui devient :

```

1 int __fib(int a, int b, int n) {
2     if (!n)
3         return b;
4     return __fib(a + b, a, n-1);
5 }
6
7 int fib(int n) {
8     return __fib(1, 0, n);
9 }

```

BEAUCOUP PLUS COURT!!!

2.1.5 Fonctions prenant des pointeurs

Si notre fonction est trop longue, on peut se dire "il faut que je la découpe en plus petit morceau", mais que faire quand le code dans une boucle est trop long ? Il suffit de faire une autre fonction qui fera la même chose mais qui prendras nos variables en temps que pointeur pour les modifier.

2.1.6 Multiple expressions en une seule ligne

(cette technique ne viens pas de moi)

Chose assez peu utiliser normalement en C, mais on peut mettre plusieurs expressions (opération renvoyant une valeur) en une seule ligne.

Pour ce faire, la syntaxe est

```
1 (expr1, expr2, ..., exprn)
```

Et c'est une expression, ayant pour valeur celle de "exprn".

Cela peut paraître inutile jusqu'à qu'on se rappelle que les fonctions/expressions peuvent avoir des effets de bords.

Exemple :

```
1 int main(void) {
2     // trucs
3     ...
4
5     close(fd1);
6     close(fd2);
7     close(fd3);
8     return 0;
9 }
```

peut s'écrire

```
1 int main(void) {
2     // trucs
3     ...
4
5     return (close(fd1), close(fd2), close(fd3), 0);
6 }
```

Néanmoins on n'a pas le droit d'avoir pour expression une assignation pour la norminette.

Enfin, normalement ...

Il suffit de prendre la variable en pointeur et la valeur dans une fonction.

ainsi

```
1 int main(void) {
2     // trucs
3     ...
4
5     a = 3;
6     b = 54;
7     c = 13;
8     return f(a, b, c);
9 }
```

peut devenir

```
1 int a_int(int *x, int val) {
2     *x = val;
3     return val;
4 }
5
6 int main(void) {
7     // trucs
8     ...
9
10    return (a_int(&a, 3), a_int(&b, 54), a_int(&c, 13), f(a, b, c));
11 }
```

2.2 Problèmes de mémoires

Une autre contrainte qui est "difficile" lorsqu'on commence à avoir à gérer la mémoire manuellement est de ne pas oublier de free après l'avoir utiliser.

Pour déboguer je conseil d'avoir pour flags : "-O2 -pipe -g -fsanitize=address,leak,undefined -Wall -Wextra -Werror -pedantic -std=c99"

Explication des flags :

1. -Wall -Wextra -Werror : pour avoir des erreurs de compilations en cas de choses "douteuses".
2. -pedantic -std=c99 : la même, mais pour avoir un code C "standard"
3. -O2 : pour que le code créé soit plus rapide
4. -pipe : pour que la compilation soit plus rapide
5. -g : ajoute des informations de débogage pour par exemple gdb
6. -fsanitize=address,leak,undefined : transformes les leaks, accès d'adresses non alloués, et les actions non définis en erreur en runtime et indique où cela s'est passé

Et sinon utiliser valgrind.

2.2.1 Ne pas allouer

Ceci est la meilleur manière de pas avoir de leaks, ça paraît évident, mais il y a beaucoup de situations où une allocation paraît être la seule solution mais auquel ce n'est pas le cas.

Si notre besoin est de taille fixe (et que la mémoire ne sort pas du scope) on peut plutôt faire un tableau sur le stack.

On peut aussi créer des structures de données spécialiser pour faire différentes sans allocation comme concatener deux chaînes de caractères (une structure prenant deux chaînes), prendre une sous chaîne (structure ayant une chaîne et une fin), une liste de tout les mots d'une chaîne pour faire une opération (créer un itérateur à état à la place) etc.

2.2.2 Lifetime et ownership

Ceci n'est pas une méthode, mais une manière de formaliser le programme permettant de plus facilement penser à où free.

Ceci est surtout utiliser en "rust", mais les programmeurs en C sont aussi obligés d'y penser, même si c'est de manière moins explicite.

On va dire que la durée de vie (lifetime) d'une valeur est de quand à quand elle est accessible avec une valeur "correcte".

Le but est de free à la toute fin des lifetimes des valeurs (si elles ont été alloués)

Pour ce, on dit qu'une valeur a pour propriétaire (son ownership) une variable.

Pour une fonction qui va prendre cette variable en paramètre, il y a deux cas à distinguer

1. L'ownership est transmis : c'est maintenant à cette fonction que revient le devoir de free la valeur, et l'endroit l'ayant appelé ne doit pas utiliser cette valeur après.
2. L'ownership est emprunter : on peut utiliser la valeur comme on veut mais il ne faut pas le free dans cette fonction

Lorsque vous créez une fonction, pensez bien à dans quel catégories vos arguments (si alloués) sont, si possible le mettre en commentaire.

Une autre opération pouvant influencer sur l'ownership est l'assignation.

Or, cette fois, il est pratique de dire que l'ownership est toujours transmis.

Et la dernière est le renvoie, qui est évidemment transmis vers l'endroit l'ayant appelé.

Si vous avez ça en tête, une valeur devrait avoir qu'un seul owner, et savoir quand la free devient évident (juste à la fin de la lifetime de la dernière variable l'ayant).

2.2.3 Arena Allocators

Cette technique vous prendra certainement trop de temps à coder pour la piscine, mais l'idée peut être utile pour d'autres choses ou pour après la piscine.

Un allocateur est quelque chose qui permet d'allouer de la mémoire.

Créer ses propres allocateur est surtout utiliser en "zig" mais est tout à fait faisable en C aussi.

L'idée est la même qu'avoir un gros buffer dans le stack et tout mettre dessus, néanmoins, cette fois ce n'est pas dans le stack.

Donc on alloue une grosse zone mémoire au départ, on l'utilise dans le programme, et on free cette zone à la fin, de telle manière qu'on n'a pas besoin de free au milieu du programme.

Un problème est qu'on ne peut pas free avant la toute fin.

Un exemple d'implémentation : (la piscine est là pour vous apprendre à coder donc ne copiez pas tout sans comprendre, le débogage sera bien pire)

```

1 struct arena {
2     char *data; // où les allocations iront
3     int size; // la taille des données
4     int pos; // où on en est dans les allocations
5     struct arena *next; // si il n'y a plus de place dans data on
        écrit dans un prochain arena
6 };
7
8 struct arena *init_arena(int size) {
9     struct arena *a;
10
11     a = malloc(sizeof(struct arena)); // vérifiez vous si le malloc
        marche
12     a->data = malloc(size);
13     a->size = 0;
14     a->pos = 0;
15     a->next = 0;
16     return a;
17 }

```

```

18
19 int free_arena(struct arena *a) {
20     int r;
21
22     r = 0;
23     if (a->next)
24         r |= free_arena(a->next);
25
26     r |= free(a->data);
27     r |= free(a);
28
29     return r;
30 }
31
32 void *alloc_arena(struct arena *a, int size) {
33     void *rv;
34
35     while (1) {
36         if (a->pos+size <= a->size) {
37             rv = a->data + pos;
38             a->pos += size;
39             return rv;
40         }
41
42         if (!a->next)
43             a->next = init_arena(size > a->size ? size : a->size);
44
45         a = a->next;
46     }
47 }
48
49 void *calloc_arena(struct arena *a, int size) {
50     void *rv;
51
52     rv = alloc_arena(a, size);
53     ft_memset(rv, 0, size); // on peut aussi itérer sur rv mais il y
54                             // a plus efficace donc je vais supposer que votre memset l'est
55                             // plus
56     return rv;
57 }
58
59 // avoir un reset est utile si on sait qu'on n'as plus besoin des
60 // données d'avant mais qu'on n'as pas envie de réallouer
61 void reset_arena(struct arena *a) {
62     while (a) {
63         a->pos = 0;
64         a = a->next;
65     }
66 }

```

Et ainsi, maintenant pour éviter tout leaks de mémoire vous pouvez faire :

```

1 int main(void) {
2     struct arena *a = init_arena(10000);
3
4     fonction_allouante_difficile_a_tracer(a);
5     fonction_allouante_difficile_a_tracer2(a);
6     fonction_allouante_difficile_a_tracer3(a);

```

```

7
8     return free_arena(a);
9 }

```

Attention par contre, on ne peut plus check pour l'adressage out of bound en général (à moins de creer des fat pointer où choses de ce genre).
 Vous pouvez réutiliser cette technique pour gerer les ouvertures (et fermeture) de fichier de la même manière.

3 Structures de données

3.1 Fat pointer

Imaginons que vous avez un pointeur, comment savoir où sa zone termine ?
 Il y a deux moyens pour ça, avoir une valeur impossible qu'on met à la fin (l'approche des chaînes de caractères) (appeler valeur sentinel) ou sinon avoir une valeur de taille.
 Lorsqu'on met un pointeur et sa taille ensemble, on dit que c'est un "fat pointer".
 Il y a plusieurs avantages

1. permet de connaître la taille en $O(1)$
2. permet le check de l'out of bound en $O(1)$
3. peut être utiliser même sans connaître de valeurs de fin
4. permet de prendre des sous chaînes arbitraire (où ça commence et où ça termine) sans avoir à allouer (alors que l'approche sentinel ne peut seulement controler la position de départ sans allouer)

Exemple pour faire ces choses :

```

1 struct fat_char {
2     char *data;
3     int size;
4 }
5
6 // taille
7 int len_fc(struct fat_char fc) {
8     return fc.size;
9 }
10
11 // checking bound et indexage
12 char index_fc(struct fat_char fc, int index) {
13     if (index > len_fc(fc)) {
14         // il y a eu une erreur et vous pouvez la gere comme vous
15         // voulez
16     }
17     return fc.data[index];
18 }
19 // sans avoir de valeur de fin
20 struct fat_int {
21     int *data;
22     int size;

```

```

23 }
24
25 // sous chaîne arbitraire
26 struct fat_char sous(struct fat_char fc, int deb, int fin) {
27     fat_char fc_new;
28
29     fc_new.data = fc.data + deb;
30     fc_size = fin - deb;
31     return fc_new;
32 }

```

3.2 Linked list

Cette structure est faite pour pouvoir rajouter des éléments à une liste, ce que l'on pouvait pas jusqu'à présent.

L'idée est que chaque élément sait quel va être le suivant (va avoir un pointeur vers le suivant).

Mon implémentation de l'allocateur arena est un exemple de tel.

C'est plus facile pour faire des opérations récursives, mais en général moins efficace (le temps d'accéder aux éléments, deferrer les pointeurs etc.)

En soit surtout utile lorsqu'on a pas besoin de stocker beaucoup d'éléments.

Cela peut aussi être utile pour partager une chaîne de fin entre plusieurs listes, mais cela complique beaucoup la gestion de mémoire à tel point qu'il est souvent préférable de juste réallouer cette fin à chaque fois.

niveau template ça ressemble à ça :

```

1 struct link_T {
2     T val;
3     struct link_T *next;
4 };
5
6 void add(struct link_T *l, T el) {
7     struct link_T *node;
8
9     while (l->next)
10         l = l->next;
11
12     node = malloc(sizeof(struct link_T));
13     node->val = el;
14     node->next = 0;
15     l->next = node;
16 }
17
18 T get(struct link_T *l, int n) {
19     while (n--)
20         l = l->next;
21     return l->val;
22 }
23
24 int free_l(struct link_T *l) {
25     if (!l)
26         return;
27     free_l(l->next);
28     free(l);

```

29 }

où “T” est un type, souvent “int” ou “char”

3.3 Vector

Voici la structure de donnée la plus utile (et la plus utilisée) pour avoir un ensemble de données qui va potentiellement grandir.

L’idée est d’utiliser un fat pointer avec un peu trop de mémoire en temps que buffer, et agrandir d’un multiple de la taille lorsqu’on a plus de place pour rendre l’ajout d’éléments $O(1)$ en amortie.

Template d’implémentation :

```
1 struct vector_T {
2     T *data;
3     unsigned int len;
4     unsigned int size;
5 }
6
7 void grow(struct vector_T *t) {
8     unsigned int new_size;
9
10    new_size = t->size == 0 ? 1 : 2*t->size;
11    T *new_data = malloc(new_size * sizeof(T));
12    if (t->data) {
13        ft_memcpy(new_data, t->data, t->len * sizeof(T));
14        free(t->data);
15    }
16    t->data = new_data;
17    t->size = new_size;
18 }
19
20 void add(struct vector_T *t, T el) {
21     if (t->len == t->size)
22         grow(t);
23     t->data[(t->len)++] = el;
24 }
25
26 T get(struct vector_T *t, int n) {
27     return t->data[n];
28 }
29
30 T pop(struct vector_T *t) {
31     return t->data[--(t->len)];
32 }
33
34 T top(struct vector_T *t) {
35     return t->data[(t->len) - 1];
36 }
37
38 int free_v(struct vector_T *t) {
39     if (t->data)
40         free(t->data);
41     t->size = 0;
42     t->len = 0;
43 }
```

où “T” est un type, souvent “int” ou “char”

Le “free” dans grow peut faire penser que cette structure sera inutilisable en conjonction avec un arena, mais c’est pas vraiment vrai, on peut ne pas free, juste cette structure prendra 4 fois plus de place que nécessaire dans le pire des cas (et 2 fois en cas général).

4 Algorithmes types

Chacun de ces algorithmes ont pour but de donner une idée, ils vont servir peut-être pas utile mais ces techniques peuvent permettre de rendre son code plus rapide (ce qui est pas forcément utile mais c’est cool)

4.1 Puissance

Pour calculer x^n où $n \in \mathbb{Z}$, l’approche la plus “directe” est de faire

```
1 T pow(T x, int n) {
2     if (n == 0)
3         return one_of_type_T;
4
5     if (n < 0)
6         return pow(inverse(x), -n);
7
8     return x * pow(x, n-1);
9 }
```

où “T” est un type représentant des nombres et “inverse(x)” retourne l’inverse de “x” dans le type “T”.

Si “T” est un type d’entier et que $x \geq 2$, on peut directement retourner 0.

Cette approche est en $O(n)$ temp et pareil en espace (les appels fonctions utilisent la stacks)

En utilisant plutôt la relation (si $n \geq 0$) $x^n = \begin{cases} (x^2)^{\lfloor \frac{n}{2} \rfloor} & \text{si } n \equiv 0 \pmod{2} \\ x(x^2)^{\lfloor \frac{n}{2} \rfloor} & \text{si } n \equiv 1 \pmod{2} \end{cases}$

On peut faire mieux, c’est à dire :

```
1 T pow(T x, int n) {
2     if (n == 0)
3         return one_of_type_T;
4
5     if (n < 0)
6         return pow(inverse(x), -n);
7
8     if (n & 1)
9         return x * pow(x*x, n >> 1);
10    return pow(x*x, n >> 1);
11 }
```

Ce qui est $O(\log(n))$ temps et mémoire.

Et après quelques opération pour le transformer en forme tail recursive et appliquant la transformation en while devient :

```

1 T pow(T x, int n) {
2     T rv;
3
4     rv = one_of_type_T;
5
6     if (n < 0) {
7         x = inverse(x);
8         n = -n;
9     }
10
11     while (n) {
12         if (n & 1)
13             rv *= x;
14
15         x *= x;
16         n >>= 1;
17     }
18
19     return rv;
20 }

```

Ce qui est en $O(1)$ mémoire et $O(\log n)$ temps.

Cette technique peut aussi être utile pour trouver des nombres de fibonacci car $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} F_{n+2} \\ F_{n+1} \end{pmatrix}$, donnant un algorithme en $O(\log n)$ pour trouver le n -ième nombre de fibonacci.

4.2 memset

Oui, il y a des optimisations qu'on peut faire avec *memset*

Familiarisons nous avec l'implémentation classique :

```

1 void memset(char *dest, unsigned char byte, unsigned int n) {
2     int i;
3
4     i = -1;
5     while (++i < n)
6         dest[i] = byte;
7 }

```

Le truc, c'est qu'un "char" c'est petit, et si on pouvait faire toutes ses opérations sur un "unsigned long long", ça serait en théorie 8 fois plus rapide.

Un problème avec ça est que il se peut que n ne soit pas divisible par 8, ainsi, au départ, on va "aligner" les opérations à faire.

```

1 void ft_memset(char *dest, int b, unsigned long n) {
2     unsigned short *sd;
3     unsigned int *id;
4     unsigned long long *lld;
5     const unsigned short s = (((unsigned short) (b & 0xff)) << 8) | (
        b & 0xff);
6     const unsigned int i = (((unsigned int)s) << 16) | s;
7     const unsigned long long ll = (((unsigned long long)i) << 32) | i
        ;
8
9     if (n & (1 << 0))

```

```

10     *(dest++) = (char) (b & 0xff);
11     sd = (unsigned short *)dest;
12     if (n & (1 << 1))
13         *(sd++) = s;
14     id = (unsigned int *)sd;
15     if (n & (1 << 2))
16         *(id++) = i;
17     lld = (unsigned long long *)id;
18     while (n >= (1 << 3)) {
19         n -= 1 << 3;
20         *(lld++) = ll;
21     }
22 }

```

Ceci n'est pas "à la norme" car il y a trop de variables créées, mais vous pouvez réutiliser des techniques de création de variables plus haut pour en introduire moins.

4.3 Résolution d'un sudoku

Je ne vais pas parler des vraies méthodes de comment le faire de manière "logique". La technique s'appelle "backtracking", en gros, on avance autant qu'on peut, et si on a une contradiction on change notre dernière étape.

On va supposer qu'on a déjà une fonction qui dit si on peut poser un nombre sans avoir de contradiction immédiate (elle est un peu chiant à écrire) et pour afficher les plateaux de sudoku.

```

1 void __solve(int board[81], int i) {
2     int w;
3
4     if (i >= 81)
5         return;
6     if (!empty(board, i))
7         return __solve(board, i+1);
8     w = -1;
9     while (++w <= 9) {
10         board[i] = w;
11         __solve(board, i+1);
12     }
13     board[i] = empty_state;
14 }
15
16 void solve(int board[81]) {
17     __solve(board, 0);
18 }

```

4.4 Evaluation d'une position au morpion

Ici je vais montrer un exemple du minimax, enfin, une version plus spécifique (mais plus souvent utile), le négamax.

L'idée est que ton meilleur coup est celui qui rends le meilleur de votre adversaire le moins bon.

Et on fait ça récursivement jusqu'à la fin du jeu.

Je vais supposer que la position est dans un "int board[9]", ayant -1 pour le X, 1 pour le O et 0 pour les cases vides, et qu'on ne demande pas si la position a été gagnée il y a plusieurs tours.

```

1 int has_draw(int board[9]) {
2     int i;
3
4     i = -1;
5     while (++i < 9)
6         if (board[i])
7             return 0;
8
9     return 1;
10 }
11
12 int abs(int a) {
13     return a < 0 ? -a : a;
14 }
15
16 int has_won(int board[9]) {
17     int i;
18
19     i = -1;
20     while (++i < 3) {
21         if (abs(board[3*i] + board[3*i+1] + board[3*i+2]) == 3 || abs(
22             board[i] + board[i+3] + board[i+6]) == 3)
23             return 1;
24     }
25     if (abs(board[0] + board[4] + board[8]) == 3 || abs(board[2] +
26         board[4] + board[6]) == 3)
27         return 1;
28     return 0;
29 }
30
31 int max(int a, int b) {
32     return (-1 + 2*(a >= 0)) * a; // "a > b ? a : b" mais les
33     opérations ternaires sont interdites
34 }
35
36 int eval_pos(int board[9], int player) { // player est 1 ou -1
37     int best;
38     int i;
39
40     best = -1;
41     if (has_draw(board))
42         return 0;
43     if (has_won(board))
44         return -1; // ça veut dire que au coup de l'autre il y a eu un
45         triplet
46     i = -1;
47     while (++i < 9) {
48         if (board[i]) {
49             board[i] = player;
50             best = max(best, -eval_pos(board, -player));
51             board[i] = 0; // on nettoie derrière soit :)
52         }
53     }
54 }

```

```
50     return best;
51 }
```

5 Ressources externes

- bitwise hacks : <https://graphics.stanford.edu/~seander/bithacks.html>
- man : <https://www.man-linux-magique.net/index.html>
- la norminette : <https://github.com/42School/norminette/>
- what is an allocator anyway (youtube) : https://www.youtube.com/watch?v=vHWiDx_14V0
- musl libc (comme glibc mais écrit de manière plus propre, il y a beaucoup de bonnes idées) : <https://github.com/seL4/musllibc>