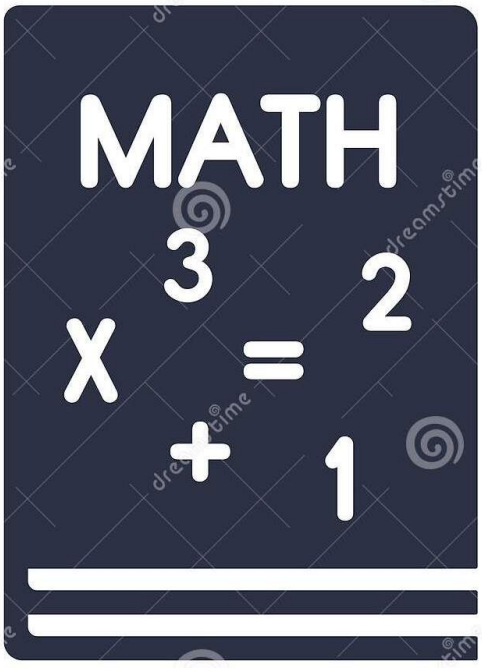
Rapport du projet N5

Algorithmique Avancé

un module pour le traitement des grands nombres entiers naturels en utilisant les listes chaînées



REMERCIEMENTS

“Avant d’entamer ce rapport, nous profitons de l’occasion pour remercier tout d’abord notre professeur Monsieur Abdelmajid Dargham qui n’a pas cessé de nous encourager pendant la durée du projet, ainsi pour sa générosité en matière de formation et d’encadrement.”

faite par : Mohamed Soulaimani Groupe 2 Nr 88

faite par : Mohamed Soulaimani Groupe 2 Nr 88

SUJET A FAIRE

Réaliser un module pour le traitement des grands nombres entiers naturels en utilisant les listes chaînées. Le module doit fournir les opérations suivantes : créer un nombre entier, afficher un nombre entier sous le format usuel, calculer la somme, la différence et le produit de deux nombres entiers, calculer le quotient et le reste de la division euclidienne de deux nombres entiers, tester si un nombre entier est divisible par un autre, tester si un nombre est premier, calculer la puissance nème d’un nombre entier.

EXÉCUTION

- - - - ✗

Pour exécuter ce programme on utilise un Makefile.

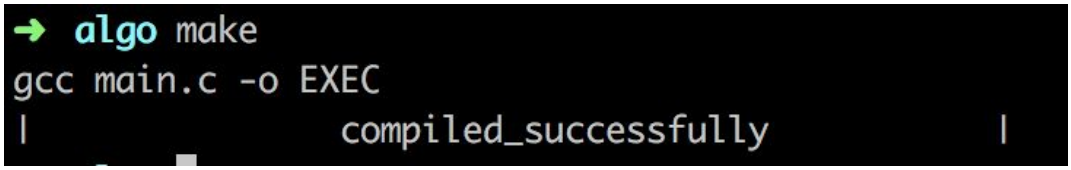
Un makefile est un fichier qui contient des paramètres pour une compilation avec gcc.

L'outil qui appelle le makefile pour compiler s'appelle “make”

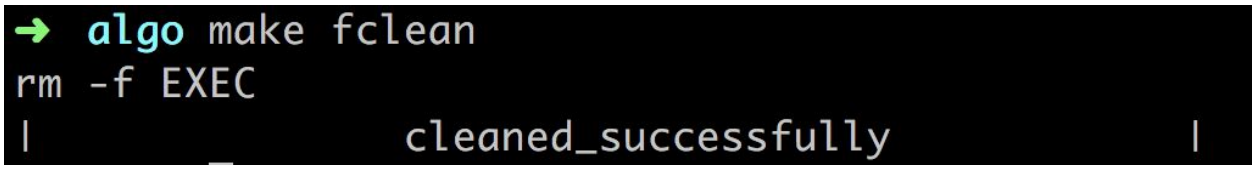
Voila le fichier Makefile avec 3 rules

```
# **************************************************************************** #
#                                                                              #
#                                                         :::      ::::::::    #
#    Makefile                                           :+:      :+:    :+:    #
#                                                     +:+ +:+         +:+      #
#    By: msoulaim <marvin@42.fr>                    +#+  +:+       +#+         #
#                                                 +#+#+#+#+#+   +#+            #
#    Created: 2020/01/10 21:43:48 by msoulaim          #+#    #+#              #
#    Updated: 2020/01/10 21:52:47 by msoulaim         ###   ########.fr        #
#                                                                              #
# **************************************************************************** #

NAME = EXEC

SRC = main.c

all: $(NAME)

$(NAME):
	gcc $(SRC) -o $(NAME)
	@echo "|        compiled_successfully       |"

fclean:
	rm -f $(NAME)
	@echo "|        cleaned_successfully        |"

re: fclean all
~
```

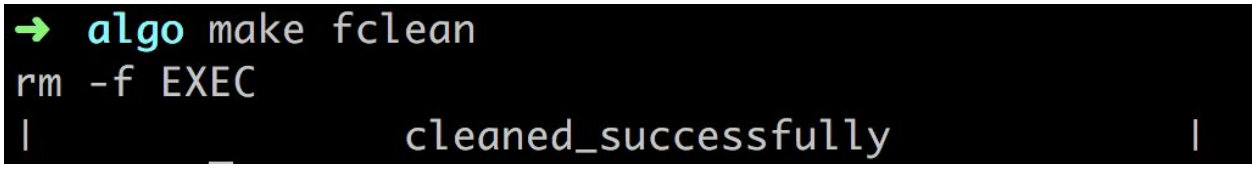
“make” , “make all” ou “make EXEC” pour compiler le projet



“make fclean” pour détruire le fichier binaire EXEC



“make re” détruire puis recompiler le projet



Structure des listes chaînées

- - - - X

-> cette list chainee va contenir a chaque de ces éléments un digit entre 0 et 9.

```
18 typedef struct  s_list
19 {
20     int c;
21     struct s_list *next;
22 }              t_list;
23
```

Lecture depuis l'entrée standard - créer un nombre entier

- - - - X

j'ai utilisé une boucle sur la fonction "read" qui vas prend zéro comme fd (file descriptor) pour lire d'entrer standard du terminal, une adresse de la variable "char c" qui va contenue un octet de l'entrée d'utilisateur, le dernier paramètre de "read" est la taille de lire qui va être 1 octet.

la boucle va stopper ou cas ou l'utilisateur entrer "ENTER" ce signal va être géré par fonction "read" comme une '\n'.

```c
t_list* new_list()
{
    t_list *first = NULL;
    t_list *temp;
    char    c = 0;
    puts("give me numbers :");
    int i = 0;
    while (read(0, &c, 1) > 0 && c != '\n')
    {
        temp = new_node(ctoi(c));
        add_end(&first, temp);
    }
    delete_zero(&first);
    return (first);
}
```

a chaque itération de "while" la fonction "raed" vas écrase la valeur enregistrée a "char c" alors en vas alloue une node de list chinée pour mettre le nouveau variable "c" avec les fonctions "new_node" "add_end()" qui vas enregistrer ce node au fin du list, mais avant le met en vas le converter de type "char" vers type "int" utilisons le tableau de code ASCII et ma fonction "ctoi" 'Char to int'.

The **decimal** set:

0	nul	1	soh	2	stx	3	etx	4	eot	5	enq	6	ack	7	bel
8	bs	9	ht	10	nl	11	vt	12	np	13	cr	14	so	15	si
16	dle	17	dc1	18	dc2	19	dc3	20	dc4	21	nak	22	syn	23	etb
24	can	25	em	26	sub	27	esc	28	fs	29	gs	30	rs	31	us
32	sp	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	del

```c
t_list* new_node(int i)
{
    t_list *new = (t_list *)malloc(sizeof(t_list));
    new->c = i;
    new->next = NULL;
    return (new);
}

int     ctoi(char c)
{
    if (c > '9' || c < '0')
    {
        puts("are u fucking with me you little shit !");
        exit(1);
    }
    c = c - '0';
    return(c);
}

void    add_end(t_list **head, t_list *new)
{
    if (*head == NULL)
    {
        *head = new;
        return ;
    }
    t_list *temp = *head;
    while (temp->next)
    {
        temp = (temp)->next;
    }
    temp->next = new;
}
```

la fonction "exit()" ici a été utilisé pour tuer le process du programme si l'utilisateur a entré une character sauf les nombres 0-9 et vas retourner '1' au terminal (1 et la retourne d'erreur par défaut des command linux comme LS).

```
void    delete_zero(t_list **head)
{
    while (*head && (*head)->c == 0)
        *head = (*head)->next;
    if (*head == NULL)
        *head = new_node(0);
}
```

la fonction "delete_zero()" vas sauter le node zero qui on vas être ajouter de l'entrée standard comme "00000012300", ce nombre va gérer comme "12300".

je vais appeler la fonction "new_list()" deux fois dans ma main pour capturer deux grand nombre "first" et "second".

la complexité de cette partie est O(n) selon n est le nombre de digit entrer

```
 t_list *first = NULL;
 t_list *second = NULL;
```

```
first = new_list();
puts("give me another one i am hungry");
second = new_list();
```

afficher un nombre entier sous le format usuel

- - - - X

-> une fonction simple qui vas affiche la list chainee du premier node vers le dernière avec l'affichage de "puts" pour afficher une '\n' après.

```
164 void    print_list(t_list *list)
165 {
166     while (list)
167     {
168         printf("%d", list->c);
169         list = list->next;
170     }
171     puts("");
172 }
```

calculer la somme de deux nombres entiers

- - - - X

-> pour faire la somme de deux nombre entier, j'ai reverser les deux nombre d'entrées "first" et "second" dans la main avant le entrer dans ma fonction addition.

-> le nombre '123456' vas gérer comme '654321'.

-> cette fonction et de complexite O(n) selon la taille de list

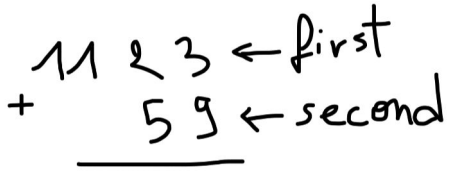
```
 94 void    reverse(t_list** head_ref)
 95 {
 96     t_list* prev = NULL;
 97     t_list* current = *head_ref;
 98     t_list* next = NULL;
 99     while (current != NULL)
100     {
101         next = current->next;
102         current->next = prev;
103         prev = current;
104         current = next;
105     }
106     *head_ref = prev;
107 }
```

-> dans la "main"

```
reverse(&first);
reverse(&second);
result = addition(first, second);
print_list(result);
```

-->dans la fonction addition()

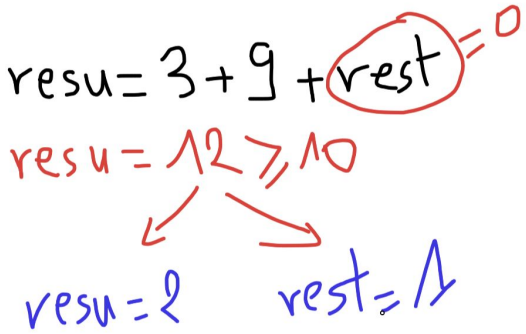
j'ai implémenter l'addition utiliser au primaire pour faire la somme de deux nombre entier naturel



au début en a deux variable intermédiaire utilise sont resu et rest sont initialisés à zéro,

et en vas considérer nos list "first" et "second" comme reverser pour accéder au nombre unité puis dizaines puis centaines ...
exemple "1123" vas gérer comme '3' -> '2' -> '1' -> '1' -> NULL

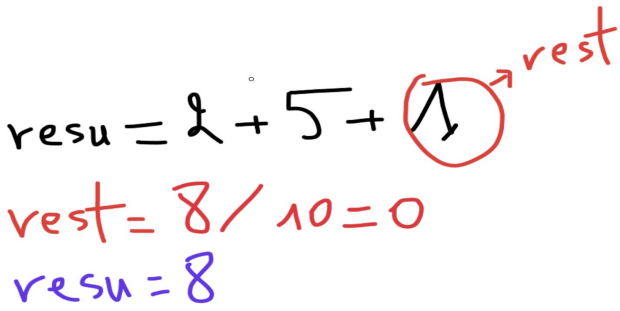
->12 est plus grand que 10 alors en peut pas l enregistrer dans un node , en vas calculer 12 % 10 pour extraire '2' de 12, et enregistrer '1' avec 12 / 10 = 1;



->utilisons "add_front()" qui vas alloue une node(2) contenue 2 et vas les adder au node result = '2' -> NULL

```
88 void    add_front(t_list **head, t_list *new)
89 {
90     new->next = *head;
91     *head = new;
92 }
```

->passons au deux nombres du first '2' et du second '5' mais cette fois ci en a un rest de la dernier addition est rest = 1.



-> result '8'->'2'->NULL

->maintenant la première boucle est finie, passons au deuxième ou au troisième selon la list qui n'est pas finie.

->dans l'exemple c'est "first" '1' -> '1' -> NULL et on a rest = 0 dans cette exemple en vas adder le rest de first a second, mais si rest != 0 en vas additionner le rest au paramètre du first jusqu'à rest == 0 apres en additionner le rest de first au result

->dans l'exemple result est '1' -> '1' -> '8' -> '2'

->avec "print_list(result)" en affiche le résultat "1182", enfin si la deuxième boucle a été fini parce que first == NULL et en a rest != 0 alors il y a un rest à additionner au result, et en peut voir ce cas si first = '2' -> NULL et second = '9' -> '9' -> '9'

```
110 t_list* addition(t_list *first, t_list *second)
111 {
112     t_list *result = NULL;
113     t_list *tmp;
114     int resu = 0;
115     int rest = 0;
116     while (first != NULL && second != NULL)
117     {
118         resu = first->c + second->c + rest;
119         rest = resu / 10;
120         if (resu >= 10)
121         {
122             resu = resu % 10;
123         }
124         add_front(&result, new_node(resu));
125         first = first->next;
126         second = second->next;
127     }
128     while (first)
129     {
130         resu = rest + first->c;
131         rest = resu / 10;
132         if (resu >= 10)
133         {
134             resu = resu % 10;
135         }
136         add_front(&result, new_node(resu));
137         first = first->next;
138     }
139     while (second)
140     {
141         resu = rest + second->c;
142         rest = resu / 10;
143         if (resu >= 10)
144         {
145             resu = resu % 10;
146         }
147         add_front(&result, new_node(resu));
148         second = second->next;
149     }
150     if (rest != 0 && result != NULL)
151     {
152         add_front(&result, new_node(rest));
153     }
154     return (result);
155 }
156
```

calculer la différence de deux nombres entiers

- - - - X

-->ce algorithme a O(n), avec n est la taille du long list first/second.
-> pour calculer la différence en utilise une fonction qui définit le grand nombre entre first and second, avec cela en décide si le résultat est positive ou négative, et afficher le '-' avant affiche le résultat.
-->dans la "main"

```
423
424     reverse(&first);
425     reverse(&second);
426     if (ret_lekbir(first, second) == 2)
427     {
428         printf("-");
429         print_list(subs_list(second, first));
430     }
431     else
432         print_list(subs_list(first, second));
433
```

-->ret_lekbir(first, second) si first > second else ret_lekbir(second, first)

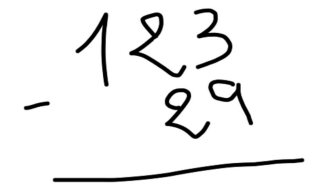
->list_lenth(t_list *list) retourne la taille du list chainee, si la longueur de list first plus que second alors la première nombre et plus grand.
->si longueur du deux list sont égale.
->en vas comparer digit par digit jusqu'à en ai un est plus grand
->sinon les deux nombre sont égaux, return 0.
-->complexité = O(n), n = le nombre de digit du petit nombre

```
166
167 int     list_lengh(t_list *list)
168 {
169     if (list == NULL)
170         return (0);
171     return (1 + list_lengh(list->next));
172 }
173
174 int     ret_lekbir(t_list *first, t_list *second)
175 {
176     if (list_lengh(first) > list_lengh(second))
177         return(1);
178     if (list_lengh(first) < list_lengh(second))
179         return(2);
180     while (first && second)
181     {
182         if (first->c > second->c)
183             return (1);
184         if (first->c < second->c)
185             return (2);
186         first = first->next;
187         second = second->next;
188     }
189     return (0);
190 }
191
```

->passons au algorithme de soustraction

```
191
192 t_list*     subs_list(t_list *first, t_list *second)
193 {
194     t_list *result = NULL;
195     int resu = 0;
196     int rest = 0;
197     while (second)
198     {
199         resu = first->c - second->c - rest;
200         if (resu < 0)
201         {
202             resu = resu + 10;
203             rest = 1;
204         }
205         else
206         {
207             rest = 0;
208         }
209         add_front(&result, new_node(resu));
210         second = second->next;
211         first = first->next;
212     }
213     while (first)
214     {
215         resu = first->c - rest;
216         if (resu < 0)
217         {
218             resu = resu + 10;
219             rest = 1;
220         }
221         else
222         {
223             rest = 0;
224         }
225         add_front(&result, new_node(resu));
226         first = first->next;
227     }
228     delete_zero(&result);
229     return (result);
230 }
231
```

-> comme la somme , en utiliser deux variable resu pour résultat , rest pour le rest and "add_front()" pour ajouter le nouveau resu au list chainee result comme une tête du list.
->c'est que subtraction du primaire.



resu = 3-9 = -6 < 0
resu = -6 + 10 = 4
rest = 1

-> en vas enregistrer 4 au list result '4' -> NULL.

resu = 2-2-1 = -1 < 0
resu = -1+10 = 9
rest = 1

-> en vas enregistrer 9 dans result '9' -> '4' -> NULL.
-> ici second est égale a NULL alors la boucle sort
-> mais la boucle first start

rest = 1
resu = 1-1 = 0

->ici en ajoute un nouveau node au result avec 0, et on enfin result = '0'->'9'->'4', c'est le rôle de delete zero to produire '9'->'4'.

```c
void    delete_zero(t_list **head)
{
    while (*head && (*head)->c == 0)
        *head = (*head)->next;
    if (*head == NULL)
        *head = new_node(0);
}
```

calculer le produit de deux nombres entiers

- - - - x

->dans la "main"

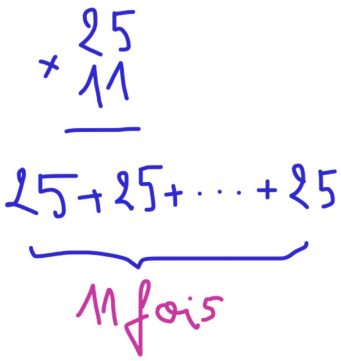
```c
432
433     reverse(&first);
434     reverse(&second);
435     if (ret_lekbir(first, second) == 1)
436         result = multiply_list(first, second);
437     else
438     {
439         result = multiply_list(second, first);
440     }
441     print_list(result);
442
```

->j'ai utilisé deux algorithme l'un avec une complexité O(n^2)

->le deuxième avec O(n) complexité, avec n la taille de la plus court list.

->le premier algorithme de O(n).

->utilisons la fonction soustraction la boucle doit répéter n fois, avec n le plus court nombre entre first et second, alors faire la somme de plus grand n fois.




```
233
234 t_list* multiply_list(t_list *first, t_list *second)
235 {
236     t_list *tmp;
237     t_list *result = new_node(0);
238
239     tmp = new_node(1);
240     int i = 0;
241     while (!(list_lengh(second) == 1 && second->c == 0))
242     {
243         result = addition(result, first);
244         reverse(&result);
245         second = subs_list(second, tmp);
246         reverse(&second);
247         i++;
248     }
249     reverse(&result);
250     return (result);
251 }
```

--->le deuxième algorithme de complexité O(n).

utilisons une fonction "add_zero()" qui vas ajouter des zéro au fin de list. 123 devient 12300 si 'i = 2'.

la multiplication ici se base au multiplication du primaire.

```
331
332 void    add_zeros(t_list *first, int i)
333 {
334     if (first == NULL)
335         return ;
336     while(first->next)
337         first = first->next;
338     while (i--)
339     {
340         first->next = (t_list *)malloc(sizeof(t_list));
341         first->next->c = 0;
342         first = first->next;
343     }
344     first->next = NULL;
345 }
346
347 t_list* new_multip(t_list *first, t_list *second)
348 {
349     t_list *tmp2 = NULL;
350     t_list *tmp = NULL;
351     t_list *tmp3 = new_node(0);
352     int i = 0;
353     while (first)
354     {
355         tmp = new_node(first->c);
356         tmp2 = multiply_list(tmp, second);
357         add_zeros(tmp2, i);
358         reverse(&tmp2);
359         tmp3 = addition(tmp3, tmp2);
360         reverse(&tmp3);
361         first = first->next;
362         i++;
363     }
364     reverse(&tmp3);
365     return tmp3;
366 }
367
```

-> en fait une multiplication de ce form.

25 * 2 + (25 * 1) * 10 = 300.

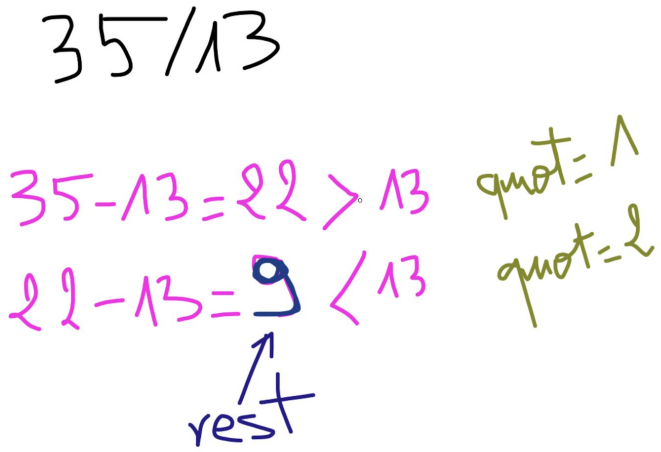
calculer le quotient et le reste de la division euclidienne de deux nombres entiers

- - - - X

→ pour calculer le quotient et le rest en vas utiliser le principe de soustraction

```
252
253 t_list* quot_rst(t_list *first, t_list *second, t_list **rest)
254 {
255     t_list *quot = new_node(0);
256     t_list *one = new_node(1);
257     if (list_lengh(second) == 1 && second->c == 1)
258     {
259         *rest = new_node(0);
260         reverse(&first);
261         return (first);
262     }
263     if (list_lengh(second) == 1 && second->c == 0)
264     {
265         printf("en peut pas diviser par zero ?");
266         exit(1);
267     }
268     if (ret_lekbir(first, second) == 2)
269     {
270         *rest = first;
271         return (quot);
272     }
273     while (ret_lekbir(first, second) == 1 || ret_lekbir(first, second) == 0)
274     {
275         first = subs_list(first, second);
276         reverse(&first);
277         quot = addition(quot, one);
278         reverse(&quot);
279     }
280     reverse(&first);
281     print_list(first);
282     reverse(&quot);
283     *rest = first;
284     return (quot);
285 }
286
```

---->si second est égal a '1' -> NULL, alors le quot est first et le rest est '0' -> NULL.
---->si second est égal a '0'->NULL, alors l'opération est impossible.
---->si second > first alors le quot est '0'->NULL et le rest est second.
---->sinon c'est un simple algorithme de complexité O(n^2), avec n le quotient de first/second.



--> dans la "main"

```
437
438     t_list *rest = NULL;
439     reverse(&first);
440     reverse(&second);
441     result = quot_rst(first, second, &rest);
442     puts("le rest de cette division est :");
443     print_list(rest);
444     puts("le quotient de cette division est :")
445     print_list(result);
446
```

tester si un nombre entier est divisible par un autre

- - - - X

-->dans la "main"
utilisons la fonction qui calcule le rest de division, est tester si ce rest est égale a '0'->NULL avec la fonction ret_lekbir().
-> la complexite ici est O(n^2), n le quotient de first/second.

```
437
438     t_list *rest = NULL;
439     reverse(&first);
440     reverse(&second);
441     result = quot_rst(first, second, &rest);
442     if (ret_lekbir(rest, new_node(0)) == 0)
443         printf ("le premier nombre est divisible par le deuxieme");
444     else
445         printf ("le premier nombre n'est pas divisible par le deuxieme");
446
```

tester si un nombre est premier

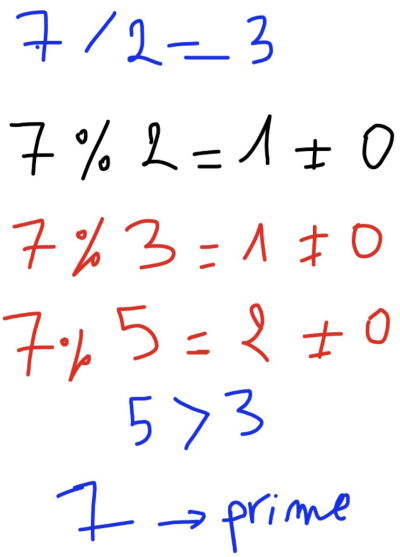
- - - - X

->la complexité de cette algorithme est O(n/4), avec n le nombre first.

-> premièrement, en test si le nombre égale à '2'->NuLL avec la fonction ret_lekbir().

->deuxièmement, en divise le nombre par '2'->NuLL pour tester si il est pair, si le rest est égale a il est divisible par '2'->NuLL.

->enfin, si le nombre ni 2 ni pair, en doit diviser ce nombre first, en tout les nombres impaire inférieure ou égale à la quotient de first a 2 jusqu'à un nombre est divisible (rest de division == 0) alors le nombre et pas prime et en affiche le nombre qui le divise, sinon si le nombre three(la suite impaire 3,5,7,9,11,13...) dépasse first/2=limit first est prime.



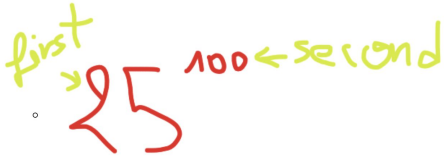
```c
void    is_prime(t_list *first)
{
    t_list *div = new_node(2);
    t_list *rest = NULL;
    t_list *limit = quot_rst(first, div, &rest);
    reverse(&limit);
    if (ret_lekbir(first, new_node(2)) == 0)
    {
        puts("nice try he is 2 so he is prime");
        return ;
    }
    if (ret_lekbir(rest, new_node(0)) == 0)
    {
        puts("this number is divisibl on two");
        return ;
    }
    t_list *one = new_node(1);
    t_list *two = new_node(2);
    t_list *three = new_node(3);
    while(ret_lekbir(three, limit) != 1)
    {
        quot_rst(first, three, &rest);
        if (rest->c == 0 && list_lengh(rest) == 1)
        {
            puts("that num is not prime he is devisible by");
            print_list(three);
            return ;
        }
        three = addition(three, two);
        printf("three == ");
        print_list(three);
        puts("");
        reverse(&three);
    }
    puts("priiiiime");
}
```

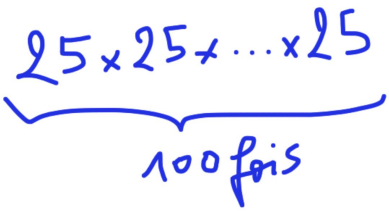
calculer la puissance n-ème d'un nombre entier

- - - - X

```
365 t_list* puissance(t_list* first, t_list *second)
366 {
367     t_list *zero = new_node(0);
368     t_list *result = new_node(1);
369     t_list *one = new_node(1);
370     while (ret_lekbir(second, zero) != 0)
371     {
372         result = new_multip(result, first);
373         reverse(&result);
374         second = subs_list(second, one);
375         reverse(&second);
376     }
377     reverse(&result);
378     return(result);
379 }
```

-->la complexité de ce algorithme est O(n^m), m longueur de list second et n longueur de list first.
-->c'est assez simple nous multiplions le nombre first a first , un nombre égale a second.





voilà le repo de mon projet sur github :
https://github.com/mohaslimani/Ensa/tree/master/TC