1.) Les exceptions.

Si on définit la fonction suivante :

```
#let f a b = -b / a;;
f : int -> int -> int = <fun>
```

et qu'on applique cette fonction de la manière suivante :

```
#f 0 5;;
#Exception non rattrapée: Division_by_zero
```

Caml renvoie un message d'erreur. Cela s'appelle une **exception**.

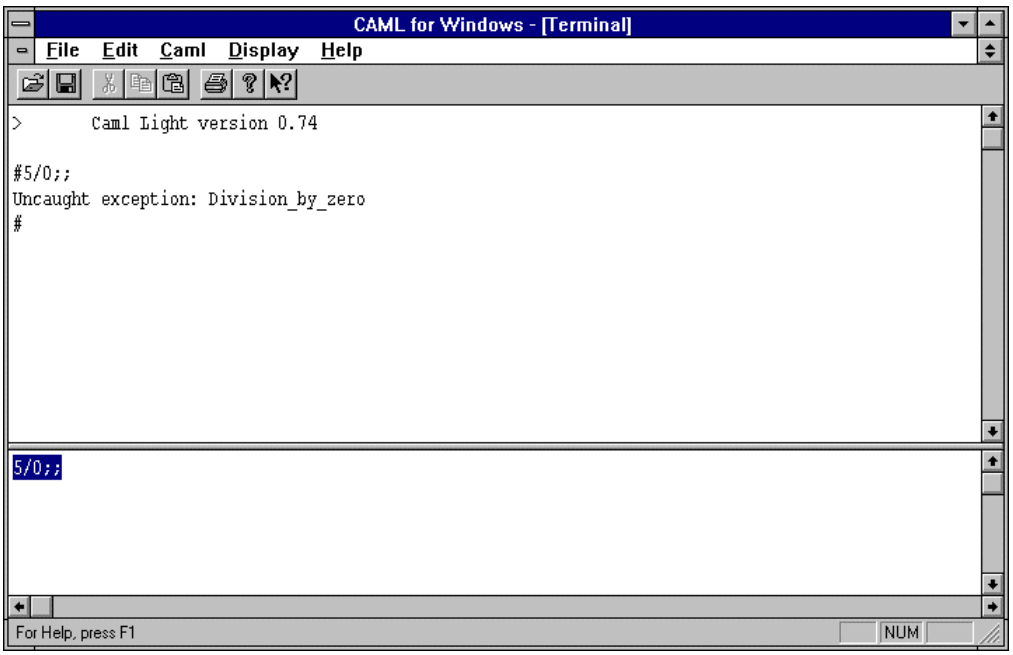


Figure 1 : Exception.

1.1.) Gestion des exceptions.

En Caml on peut :

- Utiliser des exceptions prédéfinies.
- En créer de nouvelles.

Voici quelques-unes unes des exceptions prédéfinies en Caml :

Exception	Signification
`Out_of_memory`	Envoyée par le garbage collector, quand il n'y a plus assez de mémoire pour finir le calcul.
`Invalid_argument of string`	Envoyée pour signaler que les arguments d'une fonction sont inutilisables.
`Failure of string`	Utilisé pour signaler un incident quelconque.
`Not_found`	Envoyé par les fonctions de recherche pour signaler que la recherche a échoué.
`Exit`	Cette exception est fournie pour être utilisée dans les programmes.

Le déclenchement d'une exception se fait avec le mot clé **`raise`**. Exemples :

```
let f = fun
| 0 _ -> raise (Failure "argument incorrect")
| a b -> -b / a;;
f : int -> int -> int = <fun>
```

```
f 0 3;;
Exception non
rattrapée: Failure
"argument
incorrect"
```

```
let f = fun
| 0 _ -> raise (Invalid_argument "a <> 0 svp ")
| a b -> -b / a;;
f : int -> int -> int = <fun>
```

```
f 0 3;;
Exception non
rattrapée:
Invalid_argument "a
<> 0 svp "
```

Les commandes **`raise Invalid_argument`** et **`raise Failure`** ont des abréviations :

```
let f = fun
| 0 _ -> invalid_arg "a <> 0 svp "
| a b -> -b / a;;
f : int -> int -> int = <fun>
```

```
f 0 3;;
Exception non
rattrapée:
Invalid_argument "a
<> 0 svp "
```

```
let f = fun
| 0 _ -> failwith "argument invalide "
| a b -> -b / a;;
f : int -> int -> int = <fun>
```

```
f 0 3;;
Exception non
rattrapée: Failure
"argument invalide"
```

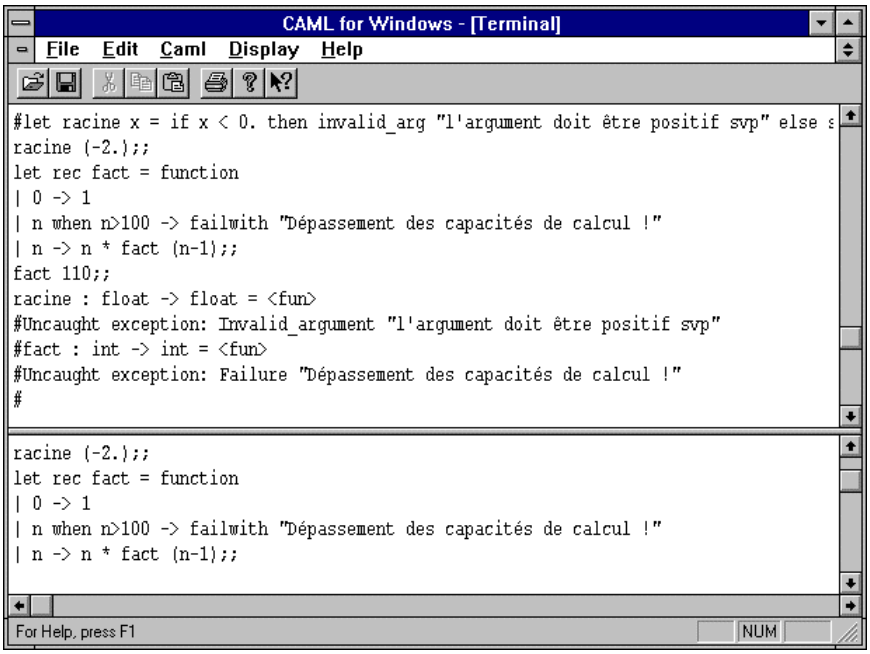


Figure 2 : Déclenchement d'exceptions.

Pour créer de nouvelles exceptions, on se sert du mot clé **`exception`**. Exemples :

```
exception Division_par_zero;;
L'exception Division_par_zero est définie.
let f = fun
| 0 _ -> raise Division_par_zero
| a b -> -b / a;;
f : int -> int -> int = <fun>
f 0 3;;
Exception non rattrapée: Division_par_zero
```

On peut aussi définir des exceptions paramétrées :

```
exception Mauvais_parametre of string;;
L'exception Mauvais_parametre est définie.
let f = fun
| 0 _ -> raise (Mauvais_parametre "a<>0 svp")
| a b -> -b / a;;
f : int -> int -> int = <fun>
f 0 3;;
Exception non rattrapée: Mauvais_parametre "a<>0 svp"
```

1.2.) Intercepter des exceptions.

On peut intercepter une exception pour "récupérer" la situation à l'aide de la construction **`try`** ***`expression`*** **`with`** ***`attitude à adopter`***.

```
try   expr
with  pattern1  -> expr1
    |  ...
    |  patternn  -> exprn
```

Exemple :

```
exception Mauvais_parametre of string;;
L'exception Mauvais_parametre est définie.
let f = fun
| 0 _ -> raise (Mauvais_parametre "a<>0 svp")
| a b -> -b / a;;
f : int -> int -> int = <fun>
#let g a b =
try f a b
with
| Mauvais_parametre "a<>0 svp" -> 0
| Mauvais_parametre "" -> (-1);;
g : int -> int -> int = <fun>
g 0 3;;
#- : int = 0
```

2.) Conception ascendante / descendante.

2.1.) Conception descendante.

On reprend le calcul de la date du lendemain. On a vu que pour calculer le lendemain d'une date, on avait besoin de connaître le nombre de jours dans un mois. Il fallait également savoir si l'année était bissextile. On a ainsi décomposé le problème en problèmes plus petits : trouver le nombre de jours dans le mois, si on est en février regarder si c'est une année bissextile, ajouter 1 au jour, si on dépasse le nombre de jours dans le mois ajouter un au mois... Cette façon de programmer a été appelée programmation descendante, simple application à l'informatique de la méthode de Descartes.

2.2.) Conception ascendante.

Il arrive qu'un problème ne se laisse pas décomposer en problèmes plus petits. Supposons que l'on veuille mettre en ordre croissant une suite de nombres. On ne voit aucune façon de décomposer ce problème en sous-problèmes. On va alors essayer de le résoudre de proche en proche. Supposons que l'on ait pu trouver le plus grand nombre de la suite et qu'on est pu le mettre au bout. On est parti de la suite :

5	2	4	6	1	3

et on a mis le plus grand au bout :

5	2	4	3	1	6

Comment avancer ? Caractérisons d'abord la situation atteinte. Il y a au bout une partie triée, en place. Le début est en désordre. Il faut réduire la longueur de la partie en désordre et étendre la partie triée en place. Quel élément lui ajouter ? Le plus grand élément de la première partie. Il faut donc chercher le plus grand élément de la première partie de la liste et l'amener au bout de cette partie.

1	2	4	3	5	6

La partie en désordre perd un élément, la partie triée au bout en gagne un. Quand il n'y aura plus qu'un élément dans la partie en désordre, il n'y aura plus de désordre ; ce sera fini.

Il faut bien comprendre l'esprit de cette méthode. Le point de départ est la proposition d'une situation intermédiaire : « Supposons que j'ai pu faire une partie du travail, donnant la situation que voici. » C'est ce qu'en mathématique on appelle une hypothèse de récurrence. On cherche alors comment passer d'un cas au suivant, puis comment démarrer, c'est-à-dire atteindre une première fois, sans trop de travail, la situation de récurrence. La récurrence est le fondement de la programmation impérative. La créativité s'exerce dans le choix d'une hypothèse, choix que l'on peut modifier ou retoucher s'il s'avère qu'il était mal venu. Le programme étant construit à partir des situations qu'il engendre, on sait qu'il fournira le bon résultat.

3.) Complexité des algorithmes.

Quand on tente de résoudre un problème, la question se pose souvent du choix d'un algorithme. Quels critères peuvent guider ce choix ? Deux besoins contradictoires sont fréquemment en présence : l'algorithme doit :

1. Etre simple à comprendre, à mettre en œuvre, à mettre au point.
2. Mettre intelligemment à contribution des ressources de l'ordinateur et, plus précisément, il doit s'exécuter le plus rapidement possible.

Si un algorithme ne doit servir qu'un petit nombre de fois, le premier critère est le plus important. Par contre, s'il doit être employé souvent, le temps d'exécution risque d'être un facteur plus déterminant que le temps passé à l'écriture.

3.1.) Mesure de la complexité.

Le temps d'exécution d'un programme dépend des facteurs suivants :

1. Les données entrant dans le programme.
2. La qualité du code généré par le compilateur.
3. La nature et la vitesse d'exécution des instructions du microprocesseur.
4. La complexité algorithmique du programme.

En général, la longueur des données est une unité de mesure adéquate. Il est alors habituel de parler d'un temps d'exécution $T(n)$ pour un programme portant sur des données de taille n. A titre d'exemple, les programmes peuvent avoir une complexité de $T(n) = cn^2$ où c est une constante. Les unités choisies pour T(n) restent à définir mais il est possible de se représenter T(n) comme le nombre d'instructions "élémentaires" exécutées par une machine formelle.

Dans de nombreux programmes, $T(n)$ est fonction de la nature des données et non seulement de leur taille. On définit alors $T(n)$ comme la complexité dans le pire des cas, c'est à dire la mesure de la complexité maximum, sur tous les ensembles de données possibles de taille n, pour un programme donné. Il est également possible de définir une complexité en moyenne, $T_{moy}(n)$ sur tous les ensembles de données de taille n. Cependant, la notion de complexité moyenne est bien plus délicate à définir que la complexité dans le pire des cas, d'une part à cause des difficultés rencontrées lors de la formulation de l'analyse mathématique de ce calcul, et d'autre part parce que la notion de données *moyennes* n'a fréquemment aucun sens précis.

Nous avons vu que le temps d'exécution d'un programme dépend également du compilateur et de la machine utilisée pour exécuter le programme. De ce fait, il n'est pas intéressant d'exprimer la complexité d'un programme en unités standard comme la seconde. Il est préférable de dire que la complexité de tel algorithme est proportionnelle à n^2, la constante de proportionnalité n'étant plus précisée.

3.2.) Notation O.

Lors de l'étude d'une suite ou d'une fonction dont la nature est compliquée, certaines questions ne nécessitent que des renseignements d'ordre qualitatif tels que $f(x) \to 0$ ou $f(x) \to +\infty$ pour $x \to +\infty$. D'autres exigent un contrôle quantitatif très précis, défini par des inégalités explicites. Les comportements asymptotiques relèvent d'une préoccupation intermédiaire : dans de nombreux problèmes, on remplace la quantité étudiée par une autre plus simple sans que, « à la limite », le résultat en soit modifié.

Pour décrire les croissances asymptotiques des fonctions, on utilise la notation de Landau "O". D'une manière analogue, quand on parle d'une complexité algorithmique en *O(n2)* pour un programme donné, on veut dire qu'il existe deux constantes positives *c* et n_0 telles que pour n supérieur ou égal à n_0 on a $T(n) \leq cn^2$. Il doit être clair que cette notation est ensembliste, et que la fonction T est décrite par la formulation précédente comme appartenant à la "classe" de toutes les fonctions satisfaisant cette condition. Par abus de langage on dit plus souvent que "T(n) est en O(n)" voire "T(n)=O(n)".

Par la suite, nous supposerons que l'ensemble des fonctions de complexité est défini sur l'ensemble N des entiers naturels et qu'elles prennent leur valeur dans R^+, l'ensemble des nombres réels positifs ou nuls. On dit alors que T(n) est en O(f(n)) s'il existe deux constantes c et n_0 telles que $T(n) \leq cf(n)$ chaque fois que n >= n_0. Un programme dont la complexité est O(f(n)) est dit appartenir à la classe de complexité O(f(n)) ou encore avoir une croissance en complexité de f(n).

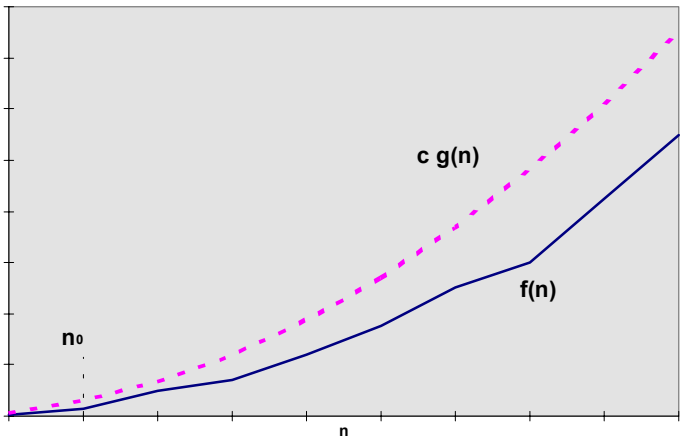


Figure 3 Exemple graphique de la notation O.

3.3.) Notation Θ.

On dit qu'une fonction $f(n)$ est en $\Theta(g(n))$ s'il existe deux constantes strictement positives c_1 et c_2 et un n_0 tels que $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ pour tout $n \ge n_0$.

La notation Θ borne donc une fonction entre 2 facteurs constants.

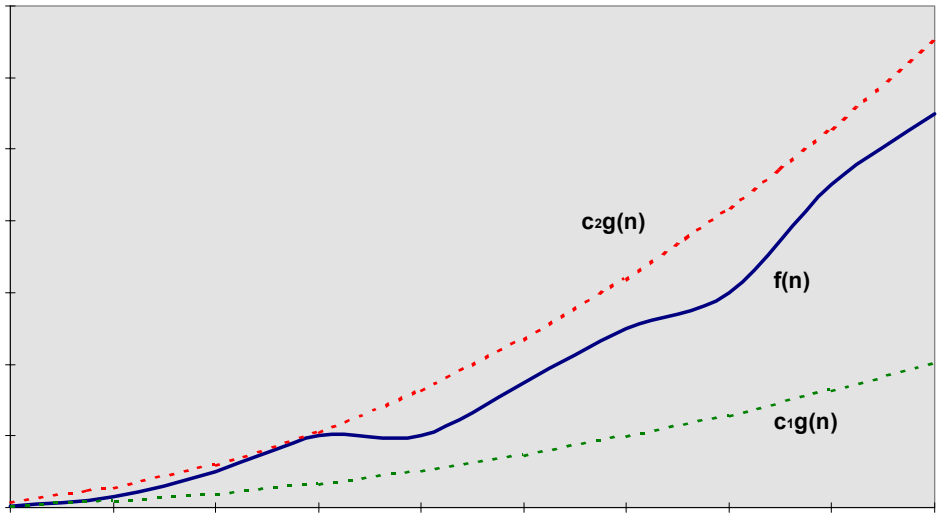


Figure 4 Exemple graphique de la notation Θ

3.4.) Notation Ω.

On dit qu'une fonction $f(n)$ est en $\Omega(g(n))$ s'il existe une constante strictement positive c et un n_0 tels que $f(n) \ge c\ g(n)$ pour tout $n \ge n_0$.

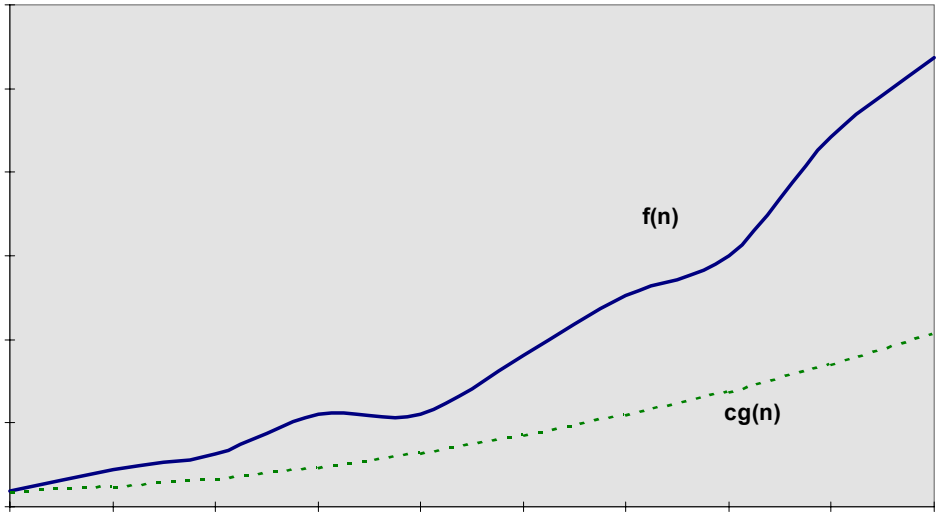
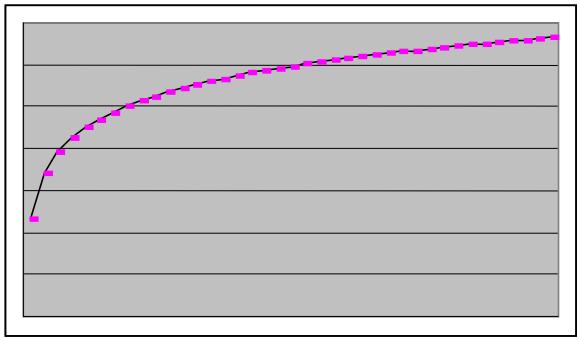


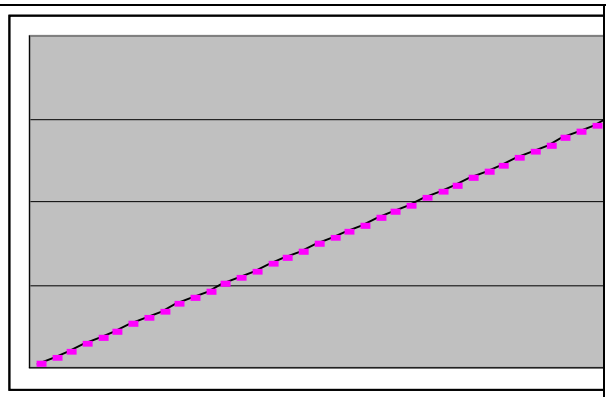
Figure 5 Exemple graphique de la notation Ω

3.5.) Les principales classes de complexité.

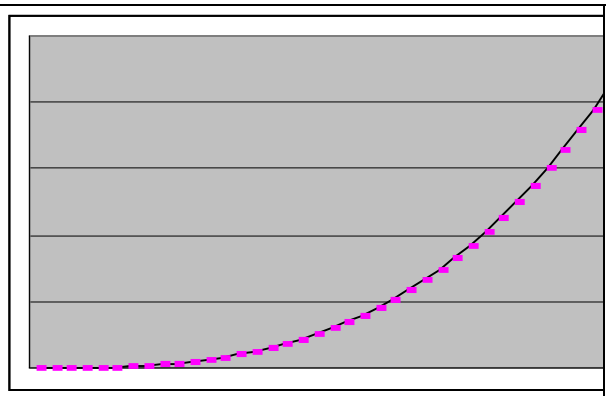
3.5.1.) COMPLEXITE LOGARITHMIQUE.

Le coût est un O(ln n). Par exemple, la recherche dichotomique dans un vecteur trié à n composantes.	

3.5.2.) COMPLEXITE LINEAIRE.

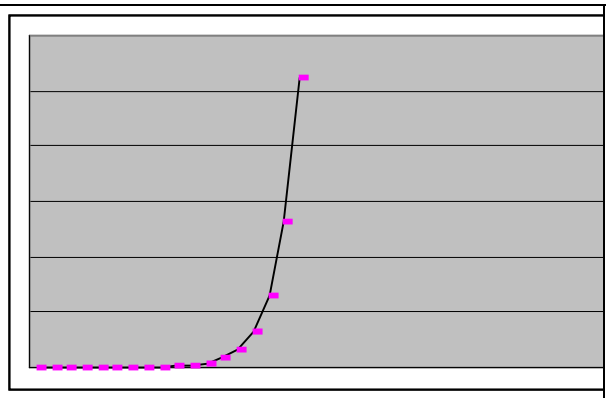
Le coût est O(n). Par exemple, la recherche linéaire dans un vecteur.	

3.5.3.) COMPLEXITE POLYNOMALE.

Le coût est $O(n^k)$. Le coût d'un tri par insertion d'un vecteur de n éléments est un $O(n^2)$.	

3.5.4.) COMPLEXITE EXPONENTIELLE.

Le coût est $O(a^n)$ pour a>1. Par exemple, la prévision des n coûts possibles dans une partie d'échecs ou la recherche de toutes les combinaisons pour le remplissage d'un carré magique d'ordre n.



Le tableau ci-dessous illustre le comportement des classes de complexité sur quelques tailles de données :

n	1	100	1000	10^6	10^9
ln n	1	<6	<8	<15	<22
n ln n	1	<50	<70	<150	<300
n^2	1	10^4	10^6	10^{12}	10^{18}
n^3	1	10^6	10^9	10^{18}	10^{27}
2^n	1	10^{30}	$>10^{300}$	$>10^{10^5}$	$>10^{10^8}$

On estime que l'âge de l'univers est inférieur à 10^{18} secondes !

De la même manière, on a 4 algorithmes de complexité respectives O(n), $O(n^2)$, $O(2^n)$ et O(ln n). On suppose que le temps de traitement d'une donnée est de 1 seconde. Le tableau suivant donne les nombre de données maximal pouvant être traité en 1000 secondes et en 10000 secondes.

	Taille maxi pour 1 seconde.	Taille maxi pour 10^3 secondes.	Taille maxi pour 10^4 secondes.	Augmentation en taille maximum des problèmes
n	1	1000	10000	10
n^2	1	32	100	3,125
2^n	1	<11	<15	1,3
ln n	1	$> 10^{998}$	$> 10^{9988}$	10^{8990}

3.6.) Calcul de la complexité.

Le calcul précis de la complexité d'un algorithme peut se révéler être un problème mathématique assez complexe. Cependant, dans la pratique, on se contente d'utiliser quelques principes fondamentaux :

3.6.1.) REGLE DE LA SOMME.

Supposons que 2 modules P_1 et P_2 aient des complexité $T_1(n)$ et $T_2(n)$ et que $T_1(n)$ est en O(f(n)) et $T_2(n)$ en O(g(n)). Alors $T_1(n)+T_2(n)$, la complexité de P_1 suivi de P_2 est en O(max(f(n),g(n)).

3.6.2.) REGLE DU PRODUIT.

Si $T_1(n)$ et $T_2(n)$ sont en O(f(n)) et O(g(n)) alors $T_1(n)T_2(n)$ est en O(f(n)g(n)).

3.6.3.) GRANDES LIGNES DE L'ANALYSE D'UN ALGORITHME.

1. La complexité de toute opération de lecture ou d'écriture peut en général se mesurer par O(1).
2. La complexité d'une suite d'instructions est déterminée par la règle de la somme. Autrement dit, elle est égale, à une constante multiplicative près, à la complexité la plus forte parmi toutes les instructions de la suite.
3. La complexité d'une instruction conditionnelle (if..then...else..) est égale à celle des instructions exécutées dans la condition plus celle de l'évaluation de la condition. On prendra toujours la plus grande entre celle dans le cas où la condition est vraie et celle où la condition est fausse.

3.7.) Exercices de calcul de complexité.

1	`let f n =`	les lignes 2,3,4,5 sont en O(1) la boucle est en O(n)
2	`print_string "Calcul de complexité";`	
3	`print_newline();`	
4	`print_string "encore 1 ligne en O(1)";`	
5	`print_newline();`	
6	`for i = 1 to n do`	
7	`  print_newline();`	
8	`done;;`	

La complexité de `f(n)` est en O(n).

1	`let majuscule s =`	les lignes 1,2 sont en O(1) la boucle est en O(string_length s) pour la conditionnelle, on prend toujours le pire des cas. les lignes 7,8,9 sont en O(1).
2	`let tmp = ref 0 in`	
3	`for i = 0 to (string_length s - 1) do`	
4	`  tmp := int_of_char s.[i];`	
5	`  if (!tmp>=97) && (!tmp<=122) then`	
6	`  begin`	
7	`  tmp :=!tmp - 32;`	
8	`  s.[i]<-char_of_int !tmp ;`	
9	`  end;`	
10	`  done;`	
11	`s;;`	

La complexité de `majuscule(s)` est en O(`string_length` s).

1	`let g n =`	la première boucle est en O(n) la seconde est en O(n)
2	`for i = 1 to n do`	
3	`  for k = 1 to n do`	
4	`  print_newline();`	
5	`  done;`	
6	`done;;`	

La complexité de g(n) est en $O(n^2)$.

1	`let tri_insertion vecteur =`	
2	`let j = ref 0 in`	instruction en O(1)
3	`for i=1 to (vect_length(vecteur)-1) do`	boucle en O(n)
4	`begin`	
5	`j := i - 1;`	instruction en O(1)
6	`while (!j >= 0) && (vecteur.(!j) > vecteur.(!j+1)) do`	Dans le pire des cas, boucle en O(n-1)
7	`begin`	
8	`echange vecteur !j (!j+1);`	instruction en O(1)
9	`j := !j - 1;`	instruction en O(1)
10	`end;`	
11	`done;`	
12	`end`	
13	`done;;`	

Dans le pire des cas, les parcours complet successifs du vecteur imposent d'effectuer $(n-1)+(n-2)+(n-3)...+1$ comparaisons. Le nombre total des comparaisons est donc majoré par $\frac{n(n-1)}{2}$.

La complexité dans le pire des cas de `tri_insertion` est en $\mathbf{O}\left(\frac{1}{2}n^2\right)$.

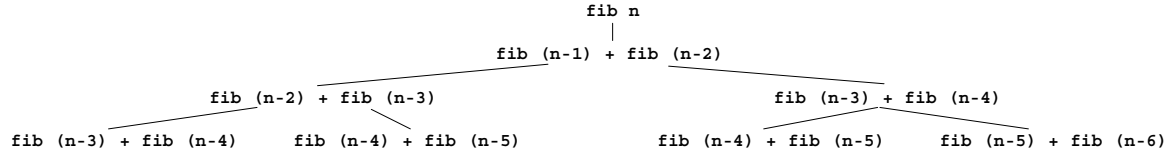
Dans le meilleur des cas, le vecteur ne sera parcouru qu'une seule fois. La complexité dans le meilleur des cas de `tri_insertion` est donc en $\mathbf{\Omega}(n)$.

1	`let rec fact = function`	
2	`| 0 -> 1`	Instruction en O(1)
3	`| n -> n * fact (n-1);;`	Appel récursif avec n-1. La complexité de cet appel est en O(n).

La complexité de « `fact n` » est en O(n).

1	`let rec fib n =`	
2	`if n<2 then 1`	Instruction en O(1).
3	`else fib(n - 1) + fib(n-2);;`	

Pour n>1, chaque appel de `fib n` engendre un appel de `fib (n-1)` et un appel de `fib (n-2)`.



La complexité de « `fib n` » est en $O(2^n)$.

1	`let bisextile an =`	Toutes les lignes sont en O(1).
2	`if ((an mod 4) = 0) then`	
3	`if ((an mod 100) <> 0) || ((an mod 400)=0) then`	
4	`if ((an mod 4000) <> 0) then true else false`	
5	`else false`	
6	`else false;;`	

La complexité de « `bisextile an` » est en O(1).

1	`let rec puissance a = function`	
2	`| 0 -> 1`	Ligne en O(1).
3	`| n -> a * puissance a (n - 1);;`	Appel récursif avec n-1. La complexité de cet appel est en O(n).

La complexité de « `puissance a n` » est en O(n).

1	`let rec premier x n =`	Dans le pire des cas, la complexité de `premier` est en O(n).
2	`if n=1 then true`	
3	`else`	
4	`if (x mod n) = 0 && (n<x) then false`	
5	`else premier x (n-1);;`	
6	`let lst_premier x =`	
7	`  for i=1 to x do`	
8	`  if (premier i i) then`	On appelle premier avec i, i-1, i-2...
9	`  begin`	
10	`    print_int i;`	
11	`    print_newline();`	
12	`  end;`	
13	`  else();`	
14	`done;;`	

La complexité de « `lst_premier x` » est exponentielle (en $O(2^n)$).

4.) Notions fondamentales à retenir.

4.1.) Langage Caml.

4.1.1.) DEFINITION D'EXCEPTIONS.

```
exception Exception_Constante;;
exception Exception_avec_argument of expression_de_type;;
```

4.1.2.) RATTRAPAGE D'EXCEPTIONS.

```
try expression with filtrage
```

4.1.3.) LANCEMENT D'EXCEPTIONS.

```
raise exception_constante ou
raise (exception_avec_argument expression)
```

4.2.) Algorithmique.

Notation de Landau.
Complexité dans le pire des cas.

5.) Exercices.

5.1.) Exercices.

EXERCICES 1.

Retrouver les nombres premiers inférieurs à 100 à l'aide du crible d'Eratosthène. Le principe de cet algorithme est le suivant : on écrit les entiers supérieurs à 1 dans l'ordre et on barre les multiples du premier nombre entier non barré, puis ceux du second etc...

Exemple avec les nombres <= 10.

1) On écrit les entiers dans l'ordre :

2	3	4	5	6	7	8	9	10

2) On barre les multiples de 2 :

2	3		5		7		9	

3) On barre les multiples de 3 :

2	3		5		7			

4) On barre les multiples de 5 :

2	3		5		7			

5) On barre les multiples de 7 :

2	3		5		7			

Les nombres 2,3,5,et 7 sont premiers.

EXERCICES 2.

L'algorithme de Salamin a été utilisé pour calculer π avec 16 millions de décimales. En voici une version en Basic :

```
 10  LET A = 1
 20  LET X = 1
 30  LET B= 1 / SQRT(2)
 40  LET C = 1/4
 50  LET Y=A
 60  LET A = (A+B)/2
 70  LET B = SQRT(B*Y)
 80  LET C=C-X*(A-Y)*(A-Y)
 90  LET X = Z*X
100  WRITE((A+B)*(A+B)/(4*C))
110  GOTO 50
```

Ecrivez en Caml la fonction *PI(n)* qui calcule π en *n* itérations.

EXERCICES 3.

L'amortissement se défini comme « *la constation comptable d'un amoindrissement de la valeur d'un élément d'actif résultant de l'usage, du temps, de changement de technique et de toute autre cause dont les effets sont irréversibles* ». L'amortissement linéaire est l'un des modes de calcul autorisé par l'administration fiscale. Son principe est le suivant :

- On fixe une durée probable d'utilisation, soit n.
- La valeur résiduelle à la fin de cette période est considérée comme nulle.
- L'annuité d'amortissement est donc égale à $a = \frac{valeur\ \ d'origine}{n}$
- Le taux d'amortissement sera donc égal à $t = \frac{100\%}{n}$

Ecrivez la fonction « `amortissement` *`valeur_origine duree`* » qui construit le tableau d'amortissement. La durée sera exprimée en années. Exemple :

```
amortissement 140000. 5;;
Années   Valeur en début d'année   Amortissement   valeur finale
1 / 5          140000.0                28000.0          112000.0
2 / 5          112000.0                28000.0          84000.0
3 / 5          84000.0                 28000.0          56000.0
4 / 5          56000.0                 28000.0          28000.0
5 / 5          28000.0                 28000.0          0.0
- : unit = ()
```