Algo, Introduction 1 : Généralités

Livres :

Froidevaux, Gaudel, Soria : Algorithmique et strucures de données. Ecrit par des locales de PSUD ⇒ se trouve à la BU. Couvre un peu plus que les cours d'algo de L3 et de M1.

Cormen, Leiserson, Rivest : Introduction to algorithms. Une bible format dictionnaire. Très bien écrit et très complet. Existe en VF.

Al-Khwârizmî, perse qui vers 830, publia un livre sur des méthodes de calcul sur les entiers et les polynômes. Son nom a été transformé et a donné le mot "algorithme".

Un algorithme, définition

Un algorithme résoud un problème. Un problème P(D) prend D en entrée et attend un résultat. Exemples :

• D = deux entiers naturels, P(D) = trouver leur somme. Autre problème : trouver leur produit.

• D = un tableau, P(D) = Trier ce tableau - on en reparlera durant le cours de L3.

• D = un fichier Prog, P(D) = dire si D est un programme Pascal (ou C, Java, Caml, Python,...) correct, et si oui, fournir l'exécutif correspondant - C'est le problème de la compilation - ce n'est pas si simple, mais ça se fait (!). Un cours y est dédié en M1 informatique.

• D = (p, d), un ficher p contenant un programme Pascal (ou .... ou une machine de Turing) et un fichier d dit "donnée de p". P(D) = dire si le programme p avec d en entrée *s'arrête*, en imaginant que les entiers sont illimités (pas de MAXINT), que la mémoire disponible est illimitée (malloc trouvera toujours de l'espace disponible) - C'est le problème de l'arrêt - on en reparlera dans la suite de cette introduction.

◦ Exemple : Soit le code fact(n) : si n == 0 alors rendre 0 sinon rendre n*fact(n-1). Il s'arrête sur l'entrée 10 mais pas sur l'entrée -10

◦ Autre exemple : Entrée d vide. Programme p qui génère tous les quadruplets d'entiers (x, y, z, n) avec $x, y, z > 0$ et $n > 2$, et teste pour chacun d'eux si $x^n + y^n = z^n$. Si p trouve un tel quadruplet, alors il arrête tout. Y a-t-il arrêt ?

• D = un polynôme à coefficients entiers, P(D) = dire si D possède des racines dans Z (non pour $x^2 - 2$, mais oui pour $x^2y - xz - 4yz^3 + 3$: (5,-1,2)) - C'est le 10e problème de Hilbert (sur 23).

Définition : un **algorithme** est un procédé automatique et effectif de calcul permettant de résoudre un problème P(D), et qui s'arrête sur toute donnée D.

Avez-vous déjà fait de l'algorithmique ? Oui, quelques exemples :

• Le pivot de Gauss • L'algo d'Euclide

• pour additionner 2 entiers n et m :

◦ L'ago du primaire à partir de n et m écrits en base 10 (7 et 5 font 12, je pose 2 et je retiens 1...) - ce que vous faites en temps normal (en base 10) et ce que font les machines (en base 2)

◦ dessiner n bâtons, puis m bâtons, compter les bâtons (des bâtons ou bien des doigts...)

On a envie de dire que le dernier algo est bien moins bon que l'autre (c'est bien le cas). C'est l'objet de la complexité que de fournir une mesure de l'efficacité d'un algo et de permettre des comparaisons.

Y a-t-il mieux que l'algo du primaire pour l'addition ? (non) Et pour la multiplication ? (oui !!)

Quelques remarques :

• *qui s'arrête pour toute donnée D* : Je propose de résoudre le problème de l'arrêt comme suit : Lancer l'exécution et si ça s'arrête, dire oui. Pbm : On attendra le résultat indéfiniment sur les programmes qui ne s'arrêtent pas. Donc ce n'est pas un algo (on dit ici que l'on a un demi-algo : quand la réponse est oui, ça répond oui, quand la réponse est non, ça répond non ou ça ne répond pas)

• *Automatique* : Pour faire des additions, pas besoin de comprendre pourquoi l'algo du primaire calcule la somme (ni même d'avoir compris ce qu'est une somme !)

• *Effectif* : on doit pouvoir faire le calcul :

◦ "prendre 0 si dieu existe, 1 sinon" n'est pas effectif

◦ "prendre 1 si le polynôme D (en entrée) a des racines entières, 0 sinon". Pbm : comment savoir s'il y a de telles racines ? Ca ne sera effectif que si vous arrivez à le faire.

◦ Deuxième tentative pour le problème de l'arrêt : Soit la fonction TM (TempsMax) suivante :

$TM : n \to max\{$temps d'exécution de p sur $d \,|\, |p| + |d| = n \wedge p$ s'arrête sur $d\}$

L'algo (?) est alors le suivant : Lancer l'exécution. Si ça s'arrête, dire oui. Si ça ne s'est pas arrêté au bout du temps $1 + TM(|p| + |d|)$, dire non.

Pbm : TM est parfaitement définie, mais comment la calculer ? Ca ne sera effectif que si on y arrive.

• "Automatique + Effectif" $\Rightarrow$ Exécutable par une machine.

• "Ordinateur" est absent de la définition. D'ailleurs, si l'on a appris un algo d'addition à mon fils, ce n'est pas pour qu'il l'implémente. Et puis Al Khwarismi, Euclide et Gauss n'avaient pas d'ordinateur.

Peut-on trouver un algorithme ?

Vous avez un problème P et vous vous apprêtez à chercher un *bon* algorithme, qui pourra s'exécuter dans des conditions raisonnables de temps et d'espace. Mais avant cela, êtes-vous sûr qu'il existera un algorithme ? Malheureusement, il existe des problèmes pour lesquels il y a aucun algorithme ! (Même en imaginant disposer d'un temps et d'une mémoire illimités). 2 preuves :

• Pour chaque fonction f de N dans $\{0,1\}$, on a le problème P_f: prendre n en entrée et calculer $f(n)$.

Il y a une infinité *non-dénombrable* de telles fonctions (A chaque réel x de $[0,1[$, associez la fonction $f_x : n \to x_n$ dans laquelle $0, x_0x_1x_2x_3...$ est l'écriture en base 2 de x).

Il y a une infinité *dénombrable* d'algorithmes (Chacun peut se coder en C par un programme de longueur finie, qui sont en quantité dénombrable).

Il y a donc des fonctions f non calculables (Il n'y a même presque que ça !) •

◦ Plusieurs raisons poussent à chercher une autre démonstration (outre le fait qu'elle est trop matheuse pour certains...) :

(1) on a montré qu'il existait des problèmes sans algorithme, mais on aimerait en citer un.

(2) Pour citer un problème, il faut le décrire, or les descriptions sont en nombre *dénombrable*, donc il se pourrait qu'en fait, les problèmes sans algoritme ne soient pas descriptibles. Est-ce le cas ? Non :

Théorème : Il n'y a pas d'algorithme pour le problème de l'arrêt. On dira que l'arrêt est indécidable.

• Par l'absurde, supposons que le programme A décide de l'arrêt, ie qu'il prend en entrée (p, d) et dit oui si p s'arrête sur d et non sinon.

Construisons alors le programme B comme suit : il prend p en entrée, lance A avec (p, p) en entrée. Si A répond non, alors B fait STOP. Si A répond oui, alors B ne ne stoppe plus jamais.

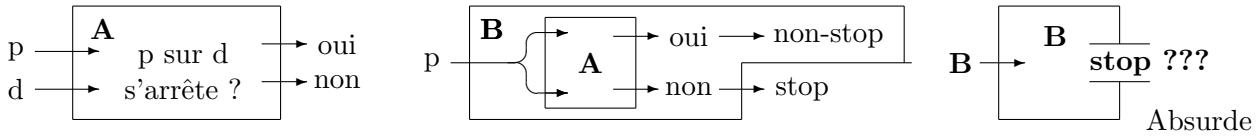
Le programme B sur l'entrée p, regarde si p s'arrête quand on lui donne en entrée son propre code, et adopte le comportement inverse en terme d'arrêt (Il se peut que p ne prenne pas du code en entrée, il y aura alors arrêt quand on lancera p sur p, avec message "erreur de type"). Bref :

B sur p ne s'arrête pas si et seulement p sur p s'arrête

En particulier, aveci l'entrée p qui est B lui-même, on obtient :

B sur B ne s'arrête pas si et seulement B sur B s'arrête

ce qui est absurde, donc le programme A n'existe pas ! •



◦ Corollaire : La fonction TM est non-calculable, car si elle l'était, on pourrait décider de l'arrêt.

Le problème de l'arrêt pouvait sembler très ardu, certains ne sont peut-être pas si étonné d'apprendre qu'il est indécidable. Mais il existe des problèmes d'apparence beaucoup plus anodine qui sont indécidables, par exemple le probleme de Post : Post prend en entrée des paires de mots, par exemple (aa,a) (ba,abaa) (bb,aa) et demande s'il existe une suite finie non vide de paires telle que les mots obtenus en concaténant les premiers, respectivement les seconds mots de chaque paire, soient les même. Pour l'exemple, il y a une solution : (aa,a) (ba,abaa) (aa,a), en effet $aa.ba.aa = a.abaa.a$.

Le problème de Post est indécidable

Le 10e problème de Hilbert est lui aussi indécidable.

La suite de ce cours ne s'intéressera qu'à des problèmes décidables.

Aparté : Cela n'etait pas le sujet de cette démonstration, mais peut-il être intéressant de lancer un programme avec son propre code en entrée ? Par exemple on peut écrire un compilateur de C écrit en C, puis le compiler et l'exécuter avec son propre code en entrée. Tout cela a-t-il un intérêt ? Et bien oui ! Vous avez un vieux compilo C BadC.exec : il est long à compiler et il produit un exécutable non performant. Alors vous écrivez en C un super compilo C GoodC.c. Vous compilez GoodC.c avec

BadC.exec, vous avez alors un compilo MediumC.exec, qui produit de bons exécutables mais est lent à compiler. Vous recompilez alors GoodC.c avec MediumC.exec et vous avez alors un compilo GoodC.exec, qui produit de bons exécutables et qui compile vite.

Complexité, premier préliminaire : les comparaisons de fonctions

Soit f et g deux fonctions (strictement positives sinon les définitions ci-dessous sont incorrectes) :

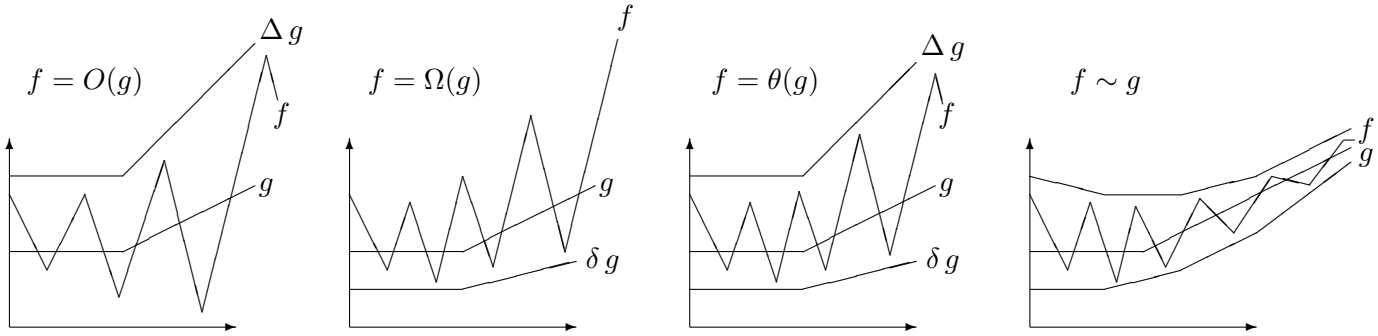
On dit que $f = O(g)$, "f est grand O de g", ssi il existe un nombre Δ tel que (à partir d'un certain rang), $f \leq \Delta\, g$, ce qui revient à dire que f/g est majoré. On peut dire que f est au maximum de l'ordre de grandeur de g (mais il peut éventuellement être bien plus petit).

On dit que $f = \Omega(g)$, "f est omega de g", ssi il existe un nombre δ (non nul) tel que (à partir d'un certain rang), $f \geq \delta\, g$, ce qui revient à dire que f/g est minoré. On peut dire que f est au minimum de l'ordre de grandeur de g (mais il peut éventuellement être bien plus grand)

On dit que $f = \theta(g)$, "f est theta de g", ssi $f = O(g)$ et $f = \Omega(g)$. On dit aussi que f est de l'ordre de grandeur de g. Notez que θ est une relation d'équivalence.

On dit que $f \sim g$, "f est équivalent à g", ssi f/g tend vers 1. C'est aussi une relation d'équivalence.

Si $f(n) = 4n^3 + 5n - 7$, alors $f = O(n^3)$, $f = O(n^7)$, $f = \Omega(n^3)$, $f = \Omega(n^2)$, $f = \theta(n^3)$, $f \sim 4\,n^3$,



Commplexité, deuxième préliminaire : taille des objets

◦ La taille d'un objet, c'est la quantité d'espace mémoire pour le coder, d'où une première définition :

• La bit-taille d'un objet est le nombre de bits utilisés pour le coder.

◦ La bit-taille mesure bien la quantité d'espace mémoire, mais est trop technique pour être pratique à utiliser, on va donc la remplacer par d'autres grandeurs. Toute grandeur convient-elle ? Non, il faut qu'elle mesure "raisonnablement" l'espace, et nous pouvons à présent définir le "raisonnable" :

• Une grandeur g sur des objets de type Truc est une taille ssi $g = \theta$(bit-taille)

◦ Exemple : Truc, ce sont les textes en français dans lesquels le nombre de caractères consécutifs autres que des lettres est limité a 6. Les paragraphes peuvent être arbitrairement longs. On peut alors considérer la bit-taille BT, le nombre de caractères NC, le monbre de mots NM, le nombre de paragraphes NP. La grandeur BT est une taille par définition. Puis NC est une taille car $8.NC = BT$ ie $8.NC \leq BT \leq 8.NC$. Puis les hypothèses sur la nature du texte font que NM est une taille car $NM \leq NC \leq (25+6)NM$, remarquez au passage que NC/NM est variable et dépend de la nature du texte, on ne peut pas dire qu'il y a une constante K telle que NC est environ $K * NM$. Enfin, NP *n'est pas* une taille, on a $NP \leq NM$ mais il manque une inégalité dans l'autre sens.

• Taille des entiers : Si n s'écrit avec k bits (en base 2), alors $2^{k-1} \leq n < 2^k - 1$, on a donc $k = 1 + \lfloor ln_2(n) \rfloor$. On retiendra que $k \sim ln_2(n)$. Donc la bit-taille d'un entier n, ce n'est pas n mais son log à base 2. Tout autre log (par exemple le log à base 10, nombre de chiffres pour écrire n en base 10) est une taille aussi car les log sont proportionnels, donc θ les uns des autres.

En pratique, très fréquemment, on ne s'intéresse pas à la variabilité du temps passé à faire une addtition, une multiplication ou une comparaison, et puis les entiers prennent de fait un espace fixe, par exemple 1, 4 ou 8 octets. On fera donc semblant que les entiers sont de taille $\theta(1)$. Mais c'est un abus. Abus nécessaire si l'on veut mettre de côté des considérations certes rigoureuses, mais souvent sans grand intérêt pour l'étude de nos algos.

• Taille d'une liste (par exemple un tableau) de n trucs. Formellement, c'est $n + \sum_1^n BT(e_i)$, mais là encore, on va souvent faire comme si un objet prenait un espace fixe et dire que la taille de la liste (du tableau) est n. C'est de nouveau un abus.

Si Truc = les booléens, tout va bien, il n'y a pas d'abus.

Si Truc = les entiers, c'est un abus tolérable que l'on fera souvent.

Si Truc = les listes d'entiers, alors stop, on ne fera pas l'abus. Difficile de dire que $[[1, 2, 3, ..., 1000000]]$ est de taille 1 !

◦ Qui est plus grand, 9^{9^9} ou 54826714590253845 71 ? Et si on disait que c'est le premier car il est plus court à écrire ? Pas naturel mais pourquoi pas (et c'est de fait l'idée de la très sérieuse notion de "complexité de Kolmogorov"). Finalement, on parle du nombre de bits pour coder un objet, mais avec quel codage ? Le codage "naturel", ce qui ne veut rien dire en théorie, même si l'on sait ce que c'est en pratique. Quoique... c'est quoi le codage "naturel" d'un graphe ? Il y en a plusiseurs, quel est le bon ?

Complexité

La complexité permet de mesurer et de comparer les performances des algorithmes. Les définitions donnés ici sont celles de la complexité en temps. Mais il y en a d'autres : complexité en espace notamment. Tentons une première définition :

• La complexité d'un algorithme est la fonction qui a une entrée d associe le temps d'exécution de l'algorithme sur cette entrée d.

◦ Cette définition rend bien ce que l'on souhaite mais n'est pas satisfaisante :

∗ On travaille donnée après donnée, ce qui n'est ni manipulable, ni parlant et ne donne aucun recul.

∗ Le temps précis est très technique, il va dépendre du langage, du codage, du compilateur, de la machine. On rentre ici dans des aspects qui sont bien trop en aval pour l'algorithmicien. Si un algo est violemment meilleur qu'un autre sur une implémentation, il le sera probablement aussi sur une autre. S'il est 10 fois plus rapide sur une implémentation, il devrait être entre 5 et 20 fois plus rapide sur une autre. Il faudrait pouvoir le dire sans entrer dans le détail de l'implémentation. On devine bien qu'en fait, les temps sur deux implémentations sont θ l'un de l'autre. On devrait donc pouvoir faire abstraction de l'implémentation (quoique l'algorithmique parle aussi de certains aspects de l'implémentation).

◦ Nous allons donc faire plusieurs choses :

(1) Introduire la complexité en nombre d'opérations (comparaisons, appels récursifs, etc.) :

• La complexité d'un algorithme en opération X est est la fonction qui a une entrée d associe le nombre d'opération X effectuées lors de l'exécution de l'algorithme sur cette entrée d.

Et une opération X sera dite "opération fondamentale" si la complexité en opération X est θ de la complexité en temps.

(2) Ne pas travailler sur les donnée l'une après l'autre, mais en fonction de la taille de la donnée :

• La complexité au pire d'un algorithme est la fonction qui à n associe le MAXIMUM sur toutes les données d de taille n, de la complexité sur d

• La complexité en moyenne d'un algorithme est la fonction qui à n associe la MOYENNE sur toutes les données d de taille n, de la complexité sur d.

◦ Notez que la complexité en moyenne nécessite une probabilité sur les entrées. Ce n'est pas toujours si naturel. Quelle probabilité mettre sur les arbres ou sur les graphes ? Pour les algos de tris, on va considérer que toutes les permutations sont équiprobables, ce qui n'est peut-être pas toujours si réaliste que ça, et puis surtout cela part du principe que tous les éléments à trier sont différents, alors qu'il peut y avoir des éléments en double, voire qu'ils sont tous égaux. Avec quelle probabilité cela arrive-t-il ?

◦ Parfois, la complexité sera donnée en fonction d'une grandeur autre qu'une taille (C'est assez courant pour les algos sur les graphes)

(3) Utiliser le vocabulaire O, Ω, θ, $\sim$. Notez que $\sim$ est plus précis que θ. Notez aussi que O ne donne qu'une majoration, surestimant éventuellement la complexité effective, ce qui n'est pas le cas pour θ et $\sim$.

◦ Exemples, regardons les complexités en fonction de n de :

```
fonction Compte_Doublons (T[1..n]) : int
      cpt = 0
      pour i de 1 a n
             pour j de i+1 a n
                    si T[i] == T[j] alors cpt ++
      rendre cpt
```

La complexité en test "T[i] == T[j]" est $n(n-1)/2 \sim n^2/2 = \theta(n^2)$. C'est bien un θ car il y en a toujours de l'ordre de n^2. On "voit" que cette complexité représente bien le temps passé. C'est une opération fondamentale, et l'algorithme ci-dessus est quadratique.

La complexité en $cpt{+}{+}$ est $O(n^2)$. Elle est effectivement O car majorée par $n(n+1)/2$, mais ce n'est pas un θ car elle peut être plus petite, elle peut même être nulle, et donc ce n'est pas une opération fondamentale. Par contre, la complexité au PIRE en $cpt{+}{+}$ est $\theta(n^2)$.

La complexité en changement de j est $\theta(n^2)$, et c'est une opération fondamentale.

La complexité en changement de i est $\theta(n)$, ce qui n'est pas pertinent pour la complexité de l'algo, ce n'est pas une opération fondamentale, elle est faite à intervalles trop éloignés pour pouvoir être représentative du temps passé par l'algo.

Vocabulaire : Un algo est dit ... si sa complexité C est ...

constant	$\theta(1)$
logarithmique	$\theta(ln(n))$
linéaire	$\theta(n)$
quasi-linéaire	$\theta(n\,ln^k(n))$ pour un certain $k>0$ (en general $k=1$)
quadratique	$\theta(n^2)$
cubique	$\theta(n^3)$
polynomial	$O(n^p)$ pour un certain p
exponentiel	s'il n'est pas polynomial mais que $ln(C)=O(n^p)$ pour un certain p
2-exponentiel	s'il n'est ni polynomial ni exponentiel mais que $ln(ln(C))=O(n^p)$ pour un certain p

∘ Les algos constants, logarithmiques., linéaires, quasi-linéaires, en $\theta(\sqrt{n})$, cubiques sont polynomiaux.

Si la complexité est en n^{1000}, la complexité est atroce mais reste polynomiale.

∘ Les algos exponentiels sont souvent de complexié $\theta(\lambda^n)$ pour un certain $\lambda>1$, mais il y en a aussi de complexité plus petite, par exemple en $n^{ln(n)}=e^{ln^2(n)}$, ou plus grande, par exemple en n^n, en 2^{n^2}.

La factorielle $n!$ (qui apparaît fréquemment comme complexité) s'encadre facilement par des exponentielles : $2^{n-1}\leq n!\leq n^n$, et est donc exponentielle. Pour plus de précision, demandez aux matheux qui vont diront que $n!\sim\sqrt{2\pi n}(n/e)^n$ (formule de Stirling). Nous sommes dans les environs de n^n.

Si la complexité est en 2^{2^n}, la complexité est trop grande pour être exponentielle, c'est 2-exponentiel.

Si on a un algo dont la complexité est d'un certain ordre de grandeur, vaut-il mieux optimiser pour gagner 10% de temps, ou changer d'algo pour gagner en ordre de grandeur ?

Nous ne nous prendrons pas la tête si les données sont petites, nous ne coderons pas le tri par tas pour trier 5 éléments. Ni si l'algo "passe" et tournera peu, nous ne coderons pas 5 heures pour gagner 10 minutes d'exécution sur un programme qui s'exécutera une seule fois.

Si l'algo traite de grosses données ou s'il a une grosse complexité, il est rentable de gagner 10%, et si les 10% s'accumulent, cela finit par changer la donne, mais il restera toujours préférable de changer d'ordre de grandeur.

Dans ce premier tableau, on a en abcisse n, la taille des données, et en ordonnée $f(n)$, le nombre d'opérations à effectuer en fonction du nombre de données. Le tableau donne alors le temps d'exécution sur un Cray 1 qui exécutait environ 10^5 opérations par seconde. Le temps est en secondes par défaut, m=minute(s), h=heure(s), j=jour(s), a=année(s), les gros chiffres sont en années. Une * est mise si c'est moins d'un millième de seconde.

$Cray1$	10	20	50	100	1000	10^6	10^9	10^{12}	10^{15}	10^{18}
n	$*$	$*$	$*$	$*$	.01	10	$2.8h$	$115j$	$317a$	3.10^5
$n*ln_2n$	$*$	$*$	$*$	$*$	.1	$3m$	$3.5j$	$12a$	16000	2.10^7
n^2	$*$	$*$	.025	.1	10	$115j$	3.10^5	3.10^{11}	3.10^{17}	3.10^{23}
n^3	.01	.08	1.25	10	$3h$	3.10^5	3.10^{14}	3.10^{23}	3.10^{32}	3.10^{41}
n^{10}	$1j$	$3a$	30000	3.10^7	3.10^{17}	3.10^{47}	3.10^{77}	3.10^{107}	3.10^{137}	3.10^{167}
2^n	.01	10	$317a$	3.10^{17}	10^{288}	$10^{\sim 3.10^5}$	$10^{\sim 3.10^8}$	$10^{\sim 3.10^{11}}$	$10^{\sim 3.10^{14}}$	$10^{\sim 3.10^{17}}$
3^n	.5	$10h$	2.10^{11}	2.10^{35}	4.10^{464}	$10^{\sim 5.10^5}$	$10^{\sim 5.10^8}$	$10^{\sim 5.10^{11}}$	$10^{\sim 5.10^{14}}$	$10^{\sim 5.10^{17}}$
$n!$	30	8.10^5	10^{51}	3.10^{144}	10^{2554}	$10^{\sim 6.10^6}$	$10^{\sim 8.10^9}$	$10^{\sim 10^{13}}$	$10^{\sim 10^{16}}$	$10^{\sim 10^{20}}$
2^{2^n}	10^{295}	$10^{\sim 3.10^5}$	$10^{\sim 10^{14}}$	$10^{\sim 10^{30}}$	$10^{\sim 10^{300}}$	$10^{\sim 10^{3000}}$				

Le Cray 1, c'était le super ordinateur de mes 10 ans, en 1975. Aujourd'hui (début des années 2020), un PC fait de l'ordre de 10^{13} flops et un superordinateur de l'ordre de 10^{18} flops.

$PC2020$	10	20	50	100	1000	10^6	10^9	10^{12}	10^{15}	10^{18}
n	$*$	$*$	$*$	$*$	$*$	$*$	$*$	0.1	$1m40$	$1j$
$n*ln_2n$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	4	$1h30$	$60j$
n^2	$*$	$*$	$*$	$*$	$*$	0.1	$1j$	3000	3.10^9	3.10^{15}
n^3	$*$	$*$	$*$	$*$	$*$	$1j$	3.10^6	3.10^{15}	3.10^{24}	3.10^{33}
n^{10}	$*$	1	$3h$	$100j$	3.10^9	3.10^{39}	3.10^{69}	3.10^{99}	3.10^{129}	3.10^{159}
2^n	$*$	$*$	$*$	3.10^9	10^{280}	$10^{\sim 3.10^5}$	$10^{\sim 3.10^8}$	$10^{\sim 3.10^{11}}$	$10^{\sim 3.10^{14}}$	$10^{\sim 3.10^{17}}$
3^n	$*$	$*$	2000	2.10^{27}	4.10^{456}	$10^{\sim 5.10^5}$	$10^{\sim 5.10^8}$	$10^{\sim 5.10^{11}}$	$10^{\sim 5.10^{14}}$	$10^{\sim 5.10^{17}}$
$n!$	$*$	$3j$	10^{43}	3.10^{136}	10^{2546}	$10^{\sim 6.10^6}$	$10^{\sim 8.10^9}$	$10^{\sim 10^{13}}$	$10^{\sim 10^{16}}$	$10^{\sim 10^{20}}$
2^{2^n}	10^{287}	$10^{\sim 3.10^5}$	$10^{\sim 10^{14}}$	$10^{\sim 10^{30}}$	$10^{\sim 10^{300}}$	$10^{\sim 10^{3000}}$				

$SO2020$	10	20	50	100	1000	10^6	10^9	10^{12}	10^{15}	10^{18}
n	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	1
$n*ln_2n$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$1m$
n^2	$*$	$*$	$*$	$*$	$*$	$*$	1	$12j$	30000	3.10^{10}
n^3	$*$	$*$	$*$	$*$	$*$	1	$30a$	3.10^{10}	3.10^{19}	3.10^{28}
n^{10}	$*$	$*$	0.1	$1m40$	30000	3.10^{34}	3.10^{64}	3.10^{94}	3.10^{124}	3.10^{154}
2^n	$*$	$*$	$*$	30000	10^{275}	$10^{\sim 3.10^5}$	$10^{\sim 3.10^8}$	$10^{\sim 3.10^{11}}$	$10^{\sim 3.10^{14}}$	$10^{\sim 3.10^{17}}$
3^n	$*$	$*$	$8j$	2.10^{22}	4.10^{451}	$10^{\sim 5.10^5}$	$10^{\sim 5.10^8}$	$10^{\sim 5.10^{11}}$	$10^{\sim 5.10^{14}}$	$10^{\sim 5.10^{17}}$
$n!$	$*$	3	10^{38}	3.10^{131}	10^{2541}	$10^{\sim 6.10^6}$	$10^{\sim 8.10^9}$	$10^{\sim 10^{13}}$	$10^{\sim 10^{16}}$	$10^{\sim 10^{20}}$
2^{2^n}	10^{282}	$10^{\sim 3.10^5}$	$10^{\sim 10^{14}}$	$10^{\sim 10^{30}}$	$10^{\sim 10^{300}}$	$10^{\sim 10^{3000}}$				

Certains parleront de la loi de Moore "la rapidité des ordinateurs s'améliore d'un facteur 2 tous les 18 mois", qui marche plutôt bien jusqu'à ce jour. Et ils diront que si le calcul est trop gros pour tourner aujourd'hui, et bien il suffit d'attendre que, grâce à la loi de Moore, les ordinateurs soient devenus assez puissants pour que le calcul passe. Sauf que la loi de Moore ne pourra pas fonctionner éternellement, il va finir par être impossible de descendre plus bas pour des raisons physiques :

Un ordinateur O_1 qui effectuerait une opération en un temps égal à celui qui est nécessaire à la lumière pour parcourir (3.10^8 m/s) la distance égale au diamètre d'un éléctron (10^{-22} m), et qui aurait été lancé au big bang (5.10^{17} s), n'aurait pas fini les calculs pour lesquels le tableau ci-dessus donne $t > 10^{30}$.

O_2, un ensemble d'ordinateurs travaillant en parallèle dans ces conditions, en nombre égal au nombre de particules de l'univers connu (10^{80}), n'aurait pas fini les calculs pour lesquels le tableau ci-dessus donne $t > 10^{110}$.

Dans ce second tableau, le temps disponible t est en abcisse. Le tableau donne alors la taille n des données que l'on peut traiter dans le temps t en fonction du nombre d'opérations $f(n)$, en ordonnée, à calculer. Dans les deux dernières colonnes, ce sont les valeurs maximales de n traitées par respectivement O_1 et O_2, qui ont été lancés au big bang.

	1 seconde	1 minute	1 heure	1 siècle	O_1	O_2
n	1.000.000	60.000.000	3.600.000.000	3.150.000.000.000.000	10^{43}	10^{103}
$n*lnn$	90.000	4.000.000	178.000.000	100.000.000.000.000	10^{41}	10^{101}
n^2	1.000	7.700	60.000	56.000.000	10^{22}	10^{52}
n^3	100	390	1530	145.000	10^{15}	10^{35}
n^5	15	34	77	1.250	100.000.000	10^{20}
2^n	20	25	31	51	142	341
3^n	12	16	20	32	90	216
$n!$	9	11	13	18	37	72
2^{2^n}	4	4	5	5	6	8

Algo, Introduction 2 : Aspects techniques

Récursivité

Une fonction ou une procédure récursive est une fonctionnalité qui s'appelle elle-même.
(parfois indirectement, il y a récursivité si f appelle g et g appelle f.)

Exemple :

```
Fact (int n) : int
si n == 0 alors rendre 1
        sinon rendre n * Fact(n-1)
```

Comment ça fonctionne ? Comment gérer la présence de plusieurs appels de la même fonctionnalité ? Chaque appel a une variable n, est-ce la même, et comment s'y retrouver sinon ? Essayons d'effectuer manuellement un calcul récursif, par exemple un appel à Fact(5). Pour cela, vous disposez de papier et d'une ardoise avec plusieurs lignes.

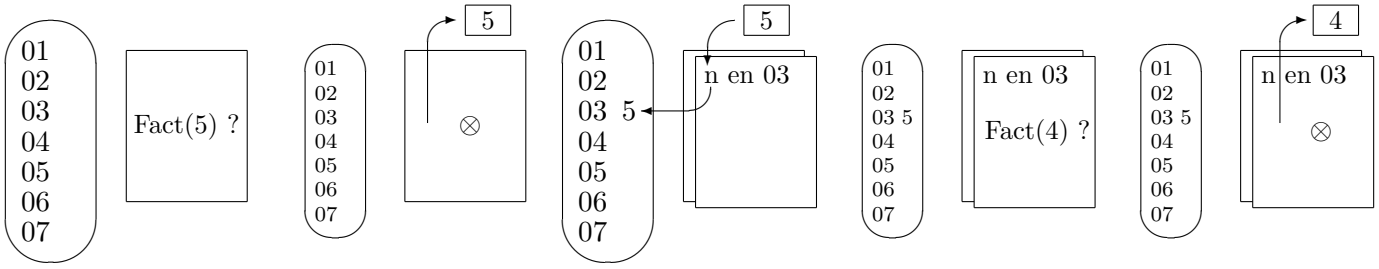
◦ Un calcul est en cours sur une feuille de papier, et il y a un appel à Fact(5).

◦ Vous notez la valeur de l'argument, 5, sur un Post-It, vous mettez une petite croix sur la feuille pour savoir où reprendre le calcul plus tard,.

◦ Vous prenez une nouvelle feuille que vous posez sur la première. Vous cherchez une ligne inutilisée sur l'ardoise, vous trouvez que la 3ème ligne est libre. Vous notez sur la feuille "n est en ligne 03", puis vous prenez le Post-It, vous y lisez 5 que vous écrivez en n, ie sur la ligne 03 de l'ardoise.

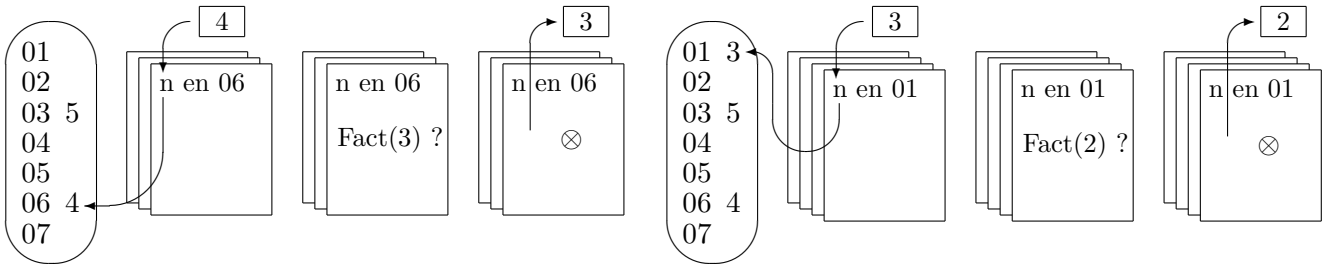
◦ Puis vous commencez à exécuter le code : n vaut-il 0 ? non, donc, on calcule $n-1$ (c.-à-d. "contenu de la ligne 03 de l'ardoise" moins 1), c'est 4, et maintenant, il faut donc calculer Fact(4)

◦ Rebelote : Vous notez la valeur de l'argument, 4, sur un Post-It, vous mettez une petite croix sur la feuille pour savoir où reprendre le calcul plus tard.



◦ Vous prenez une nouvelle feuille que vous posez sur les deux premières. Vous cherchez une ligne inutilisée sur l'ardoise, vous trouvez que la 6ème ligne est libre. Vous notez sur la feuille "n est en ligne 06", puis vous prenez le Post-It, vous y lisez 4 que vous écrivez en n, ie sur la ligne 06 de l'ardoise.

◦ Le calcul continue ainsi ...

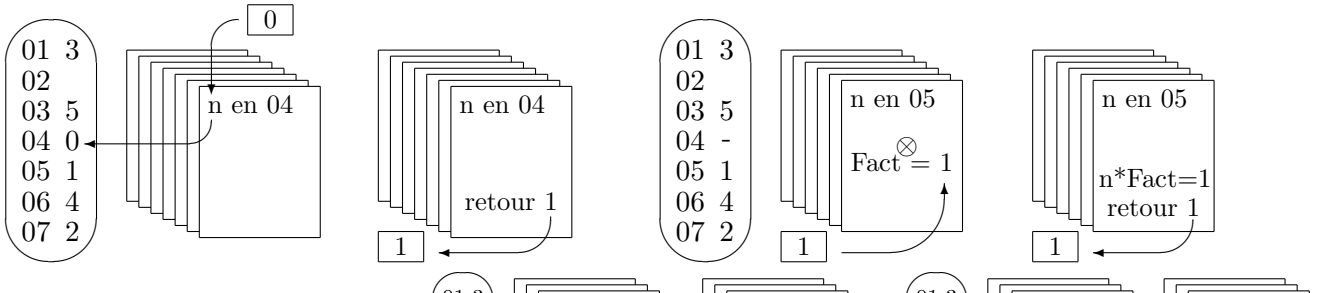


◦ ... jusqu'à ce qu'une feuille recoive la valeur 0.

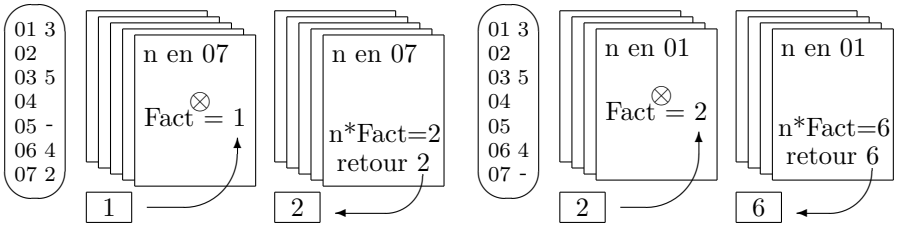
◦ Après avoir testé si n == 0, le code dit de rendre 1. On met 1 sur un Post-It.

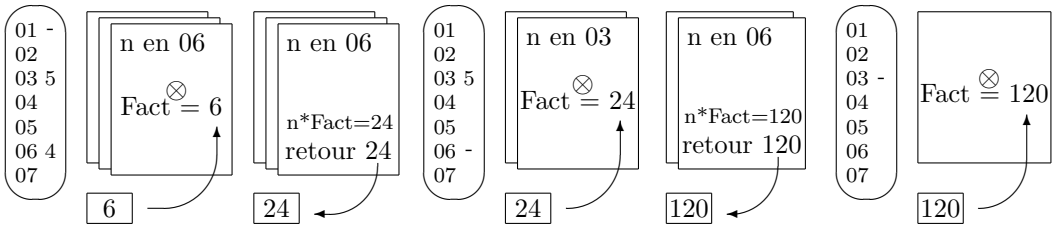
◦ Sur l'ardoise, la ligne 04 redevient libre, la feuille du dessus de pile est enlevée (elle est jetée, ou mieux : elle est mise au recyclage...), et le calcul de la feuille en dessous se prépare à repartir au niveau de la croix. On y attendait un résultat, on le trouve sur le Post-It, c'est 1.

◦ Le calcul repart, on effectue une multiplication par n (que l'on trouve sur la ligne 05 de l'ardoise) et on obtient 1, que l'on doit rendre, donc on le met sur un Post-It.



◦ Les fins de code s'effectuent et se ferment de la même manière sur les feuilles en dessous juqu'à ce que l'appelant récupère la valeur 120 :





Dans un ordinateur, la calcul se déroule de façon très similaire. L'ardoise, c'est la mémoire. Les croix sur les feuilles, ce sont des numéros de ligne de code où un calcul a été interrompu et d'où il repartira. Ce ne sont pas des piles de feuilles, mais une PILE de numéros de lignes de code et une PILE de références mémoire (paire (n,08) pour "n est en 08").

A chaque appel de fonctionnalité (récursive ou non) : On calcule les informations nécessaires (valeur ou adresse) des arguments que l'on met de côté. On EMPILE le numéro de ligne de code courant. On fait les ALLOCATIONS mémoire nécessaire pour la fonctionnalité appelée, pour ses arguments par valeur et pour ses variables locales. On EMPILE toutes les références mémoire nouvelles (adresses des arguments par variable et adresses attribuées par les allocations demandées), on enregistre dans les mémoires correspondantes les valeurs des arguments par valeur. Et enfin, le calcul reprend à la première ligne de code de la fonctionnalité appelée.

Quand la fonctionnalité se ferme, on mémorise le résultat si c'est une fonction, on DESALLOUE les mémoires allouées à l'ouverture, on DEPILE les références mémoires empilées à l'ouverture. On DEPILE une ligne de code et on reprend le calcul à partir de la ligne de code dépilée.

Fact(-1) ?

Que se passe-t-il si on lance Fact sur -1 ? (Un matheux vous répondra que Factorielle se prolonge en la fonction Γ qui justement n'est pas définie en -1, mais là n'est pas la question, nous avons un code que nous exécutons, point barre)

Fact(-1) lance Fact(-2) qui lance Fact(-3) qui lance ...

Une première réponse est de dire que l'algo ne termine donc pas.

Mais en pratique, si, il va terminer : Chaque empilement d'un nouvel appel nécessite de l'espace mémoire, Et Fact(-1) passe son temps à empiler de nouveaux appels et donc consomme de la mémoire en permanence. Jusqu'à épuisement de la mémoire, et le programme s'arrête alors brutalement avec message de type "out of memory".

Note : il peut en fait se passer autre chose : vous aurez peut-être codé vos entiers sur des int de 4 octets. Si vous avez suffisament de mémoire pour empiler quelques milliards d'appels, $n = -2^{31}$ va être atteint, ce qui peut générer un arrêt brutal pour dépassement de MAXINT.

Comment penser en récursif ?

C'est un peu comme les récurrences en math (Cf l'étymologie commune),

Si vous êtes sur un cas élémentaire (cas de base) alors vous le résolvez tout seul.

Sinon il est plus complexe et vous allez faire faire une sous-partie du problème à un clône sous-traitant (qui fera sans doute lui-même appel à un sous-traitant pour une sous-sous-partie, etc.), et vous complétez son résultat pour avoir le vôtre.

En math, on peut être amené à se demander si une récurrence est bien définie. En info, on peut toujours écrire un code récursif et lancer un calcul mais la question des matheux restera et se posera sous la forme "Est-ce que le calcul termine ?" Et s'il termine, se posera la question supplémentaire de la complexité du calcul.

Par exemple, pour factorielle, si vous êtes sur un cas élémentaire, $n = 0$, alors vous pouvez rendre 1 tout seul, sinon vous pouvez toujours demander à un ami de vous calculer factorielle(n-1), ce qui vous sera utile puisque vous n'aurez plus qu'à multiplier sa réponse par n pour avoir le résultat.

Jean doit calculer factorielle de 4, il demande à Jacques combien vaut factorielle de 3.
Jacques doit calculer factorielle de 3, il demande à Marcel combien vaut factorielle de 2.
Marcel doit calculer factorielle de 2, il demande à Maurice combien vaut factorielle de 1.
Maurice doit calculer factorielle de 1, il demande à Paul combien vaut factorielle de 0.
Paul doit calculer factorielle de 0, cas élémentaire, c'est 1, il le dit à Maurice.
Maurice multiplie son 1 par le 1 reçu de Paul, ça fait 1, il le dit à Marcel.
Marcel multiplie son 2 par le 1 reçu de Maurice, ça fait 2, il le dit à Jacques.
Jacques multiplie son 3 par le 2 reçu de Marcel, ça fait 6, il le dit à Jean.
Jean multiplie son 4 par le 6 reçu de Jacques, ça fait 24, il le dit à son commanditaire

Exemple : savoir si un élément x apparait dans une liste L

Cas élémentaire : Si la liste est vide, je peux répondre non.

Sinon ? Je peux toujours demander à un ami si x est dans $suite(L)$, la liste L sans son premier élément. L'info est pertinente, et il me faudra par ailleurs demander si le premier de la liste ne serait pas x, puis que je fasse un OU des réponses aux deux questions. Quelle question je pose en premier ? Les deux sont possibles et conduisent aux deux pseudo-codes suivants :

```
EstDans_1 (Liste L ; element x) : booleen
si L est vide
alors rendre faux
sinon si x == premier(L)
      alors rendre vrai
      sinon rendre EstDans_1(suite(L), x)
```

```
EstDans_2 (Liste L ; element x) : booleen
si L est vide
alors rendre faux
sinon si EstDans_2(suite(L), x )
      alors rendre vrai
      sinon rendre x == premier(L)
```

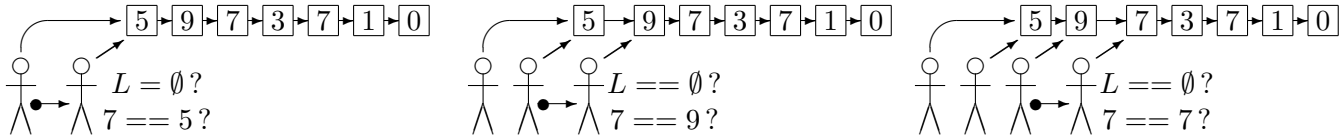
Les deux versions sont correctes mais le déroulement ne sera pas le même :

Regardons si 7 est dans la liste [5,9,7,3,7,1,0]

• En utilisant EstDans_1 (ED1) : Je lance un appel à ED1 sur la liste entière. ce premier appel ED1 teste si la liste est vide, puis regarde si "x==prem(L)", i.e. si 7==5 ...

... La réponse est non, donc il lance un 2e appel avec "suite(L)", liste privée de son premier élément, i.e. la portion de la liste à partir de 9. Ce deuxième appel ED1 teste si la liste est vide puis regarde si "x==prem(L)", i.e. si 7==9 ...

... La réponse est non, donc il lance un 3e appel avec "suite(L)", L étant ici la liste du 2e appel, donc le 3e appel travaille avec la portion de liste qui commence avec la première occurrence de 7. Ce troisième appel ED1 teste si la liste est vide puis regarde si "x==prem(L)", i.e. si 7==7...

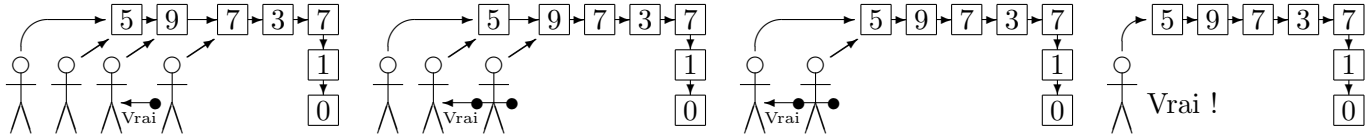


... La réponse est oui, donc le 3e appel renvoie "vrai" au 2e appel ...

... Tout ce que le 2e appel a à faire, c'est réceptionner cette réponse et la transmettre sans rien faire d'autre, car l'exécution de son code termine par "rendre appel récursif". Ce qu'il fait ...

... De la même manière, le premier appel me transmet cette réponse, sans rien à faire d'autre...

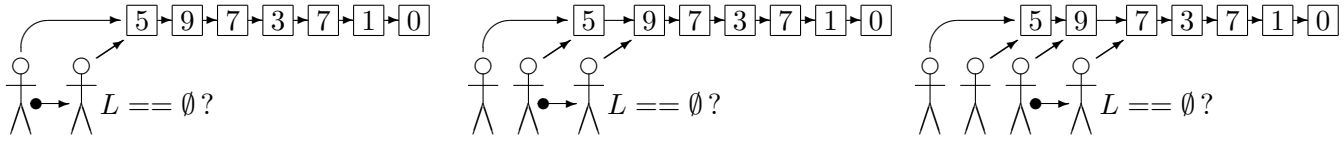
... Je réceptionne cette réponse "vrai" et c'est fini.



• En utilisant EstDans_2 (ED2) : Je lance un appel à ED2 sur la liste entière. ce premier appel ED2 teste si la liste est vide (mais pas "x==prem(L)" pour l'instant), la réponse est non ...

... donc il lance un 2e appel qui teste si la liste est vide (mais pas ...), c'est non ...

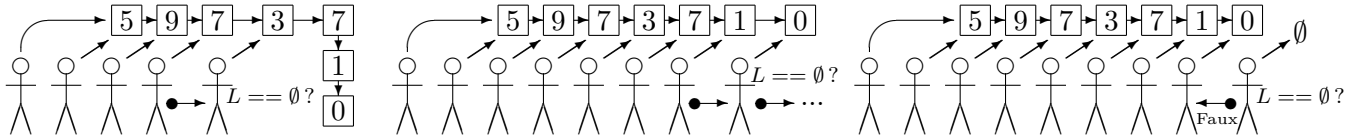
... donc il lance un 3e appel qui teste si la liste est vide. Mais pas "x==prem(L)" pour l'instant. Il ne voit donc pas pour l'instant qu'il est face à une occurence de 7. Sa liste n'est pas vide ...



... donc il lance un 4e appel qui ... Et Caetera ...

... un 7e appel sur la liste singleton dernier élément, teste si la liste est vide. Non donc il lance ...

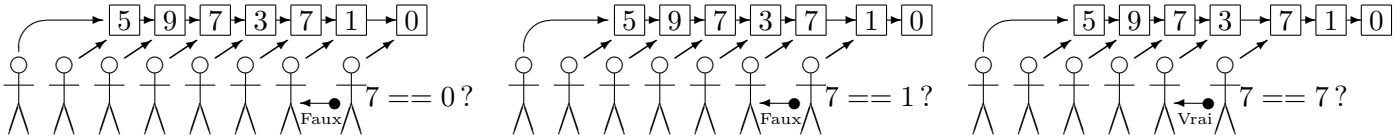
... un 8e appel en bout de liste, qui teste si la liste est vide. C'est le cas, donc ce 8e appel renvoie "faux" au 7e appel.



... Le 7e appel reçoit "Faux", donc il teste si "x==premier(L)". Non, il renvoie "Faux" au 6e appel ...

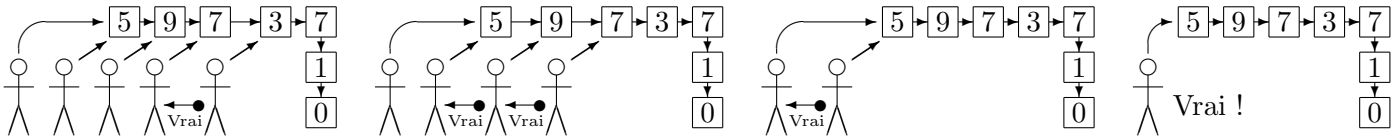
... Le 6e appel reçoit "Faux", donc il teste si "x==premier(L)". Non, il renvoie "Faux" au 5e appel ...

... Le 5e appel reçoit "Faux", donc il teste si "x==premier(L)". Cette fois, c'est OUI, donc il renvoie "Vrai" au 4e appel ...



... Le 4e appel reçoit vrai, donc il renvoie lui aussi vrai au 3e appel sans avoir à tester "x == premier(L)"...

... Les 3e et 2e appels font de même ... puis le 1er ... et je reçois Vrai

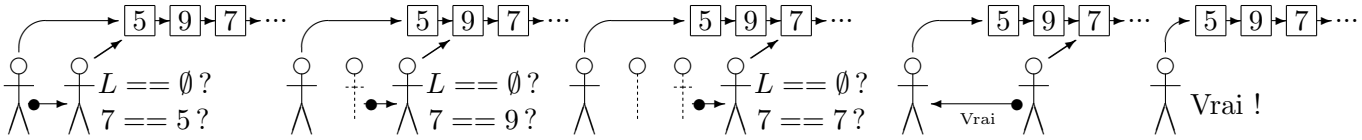


• La version ED1 s'arrête à la première occurence, elle est donc de complexité $O(L)$ tandis que la version ED2 va systématiquement jusqu'au bout de la liste et est de complexité $\theta(L)$. Si l'on souhaite simplement calculer EstDans, la première version est donc meilleure.

• Si x apparaît plusieurs fois dans la liste, ED1 trouvera la première occurence tandis que ED2 trouvera la dernière. Cela n'a pas d'importance pour EstDans, mais cela fait une différence quand vous devrez calculer par exemple la position de x. Vous demanderez alors ce que vous devez répondre si x apparaît plusieurs fois. Si on vous répond la position de la première occurence, vous vous inspirerez de ED1. Si on vous répond la position de la dernière occurence, vous vous inspirerez de ED2. Et donc les deux versions sont intéressantes à considérer.

• récursif terminal :

Pourquoi les 1e et 2e appels de ED1 attendent-ils après avoir déclenché les appels récursifs ? Pour réceptionner la réponse de leur appelé et la transmettre à leur appelant, et c'est tout, il ne servent plus qu'à faire l'intermédiaire. Mais finalement, on pourrait imaginer une autre exécution, plus économe en espace mémoire, dans laquelle un tel appel qui sait qu'il ne servira plus que d'intermédiaire, décide plutôt de s'en aller en laissant la réponse passer directement de devant lui à derrière lui.



Par contre, on ne peut pas procéder ainsi avec ED2, car après la réception de la réponse de l'appel, ED2 doit regarder cette réponse et selon le cas, renvoyer vrai ou regarder si $x == premier(L)$.

Le phénomène Fibonacci

Les nombres de Fibonacci sont définis par $\begin{cases} F_0 = F_1 = 1 \\ \forall i \geq 2,\ F_i = F_{i-1} + F_{i-2} \end{cases}$ ce qui conduit naturellement a l'écriture du code récursif:

```
fonction fibo (n entier) : entier
        si n == 0 ou n == 1                          // Code non efficace
        alors rendre 1
        sinon rendre fibo(n-1) + fibo(n-2)
```

Ce code vous calcule correctement F_0, F_1, F_5, F_{20}. Puis vous lancez F_{100}. Le résultat ne vient pas, vous attendez une minute, un jour, un siècle, rien à faire, la machine tourne sans rendre de résultat Cela semble surprenant, car avec de la persévérance, un humain est capable de calculer F_{100} et la machine est supposée aller beaucoup plus vite qu'un humain. Pour comprendre ce qui se passe, regardons comment

se calcule F_6. Pour cela, dessinons "l'arbre des appels"

J'appelle $Fibo(6)$, qui teste si $6 == 0$ ou $6 == 1$, c'est non, donc cet appel lance $Fibo(5)$ qui procède de la même manière et lance $Fibo(4)$ qui lance $Fibo(3)$ qui lance $Fibo(2)$ qui lance $Fibo(1)$ ...

... Ce dernier appel voit que son argument est 0 ou 1, donc il rend 1. L'appel $Fibo(2)$ réceptionne ce 1 et continue son code, donc il lance un autre appel $Fibo(0)$ ...

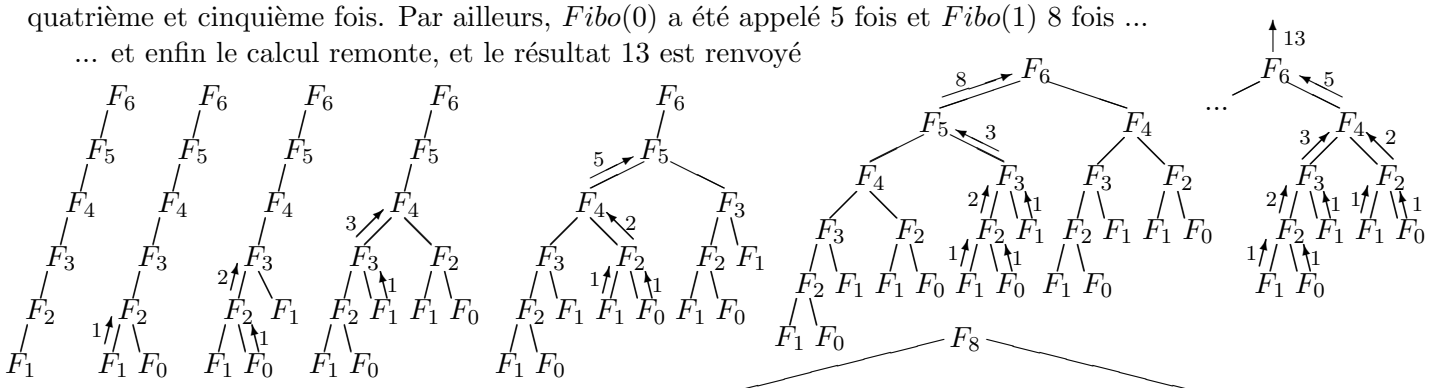
... qui voit que son argument est 0 ou 1, donc il rend 1. L'appel $Fibo(2)$ a lancé deux appels, il a reçu 1 et 1, il en fait la somme 2 qu'il rend à son appelant $Fibo(3)$. Ce dernier réceptionne cette réponse et lance un deuxième appel récursif $Fibo(1)$. Notez que l'on demande ce que vaut $Fibo(1)$ pour la seconde fois ...

... les calculs remontent jusqu'à l'appel $Fibo(4)$ qui lance un autre appel à $Fibo(2)$. On notera que $Fibo(2)$ a déjà été calculé mais peu importe, le code le refait calculer ...

... les calculs remontent jusqu'à l'appel $Fibo(5)$ qui lance un autre appel à $Fibo(3)$. On notera que $Fibo(3)$ a déjà été calculé et que cela fait recalculer $Fibo(2)$ pour la troisième fois ...

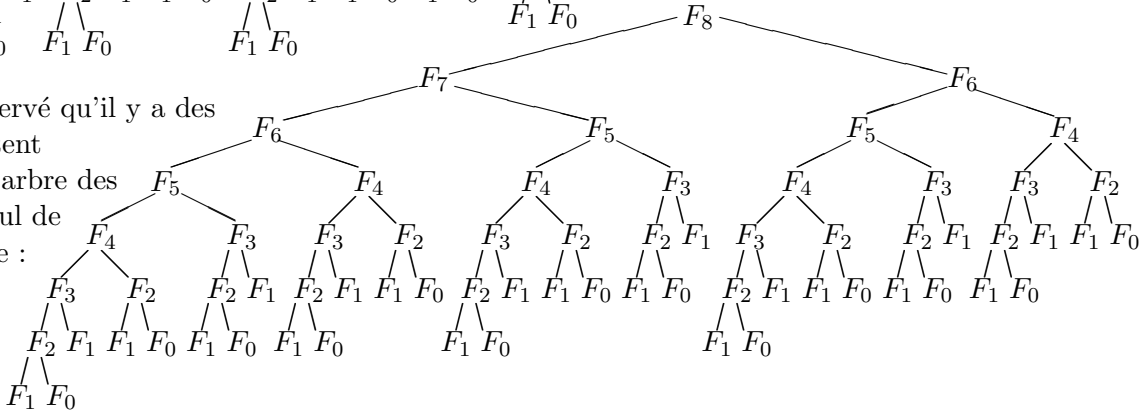
... les calculs remontent jusqu'à l'appel $Fibo(6)$ qui lance un autre appel à $Fibo(4)$, qui lui aussi est calculé pour la seconde fois. Cela fait recalculer $Fibo(3)$ pour la troisième fois, et $Fibo(2)$ pour les quatrième et cinquième fois. Par ailleurs, $Fibo(0)$ a été appelé 5 fois et $Fibo(1)$ 8 fois ...

... et enfin le calcul remonte, et le résultat 13 est renvoyé

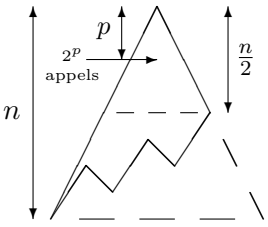


Vous aurez observé qu'il y a des calculs qui se répètent

Si on regarde l'arbre des appels pour le calcul de $Fibo(8)$, cela donne :



Le nombre d'appels répétés semble croître rapidement avec n. Combien y a-t-il par exemple d'appels finals $Fibo(0)$ ou $Fibo(1)$ quand on calcule $Fibo(n)$? Appelons X_n ce nombre.

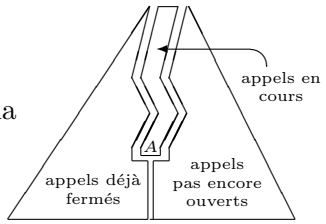


L'arbre des appels a 1 appel racine, 2 appels fils, 4 appels petit-fils, 2^k appels à profondeur k, du moins si le k^{ieme} niveau est plein. L'appel final le plus profond est tout à gauche, quand les indices ne descendent que de 1 en 1, et il est de profondeur n environ. L'appel final le moins profond est tout à droite, quand les indices descendent de 2 en 2, et il est de profondeur $n/2$ environ. Notre arbre des appels a donc plus d'appels finals que si ils étaient tous à profondeur $n/2$ et moins d'appels que si ils étaient tous à profoncdeur n. On a donc l'encadrement $\sqrt{2}^n = 2^{n/2} \leq X_n \leq 2^n$, ce qui démontre que X_n est exponentiel en n, et explique la lenteur du programme sur des n tels que 100.

PS : Combien vaut vraiment X_n ? On soupçonne que c'est du λ^n pour un certain $\sqrt{2} \leq \lambda \leq 2$. Avec quel λ ? Si $n = 0$ ou $n = 1$, l'appel principal est final, sinon il ne l'est pas et il y aura les appels finals des deux appels récursifs. Donc $X_0 = X_1 = 1$ et $\forall n \geq 2, X_n = X_{n-1} + X_{n-2}$, et donc $X_n = Fibo_n$. Combien vaut $Fibo_n$? On a $Fibo_n = \theta((\frac{1+\sqrt{5}}{2})^n)$. Plus exactement, il existe A et B tels que $Fibo_n = A(\frac{1+\sqrt{5}}{2})^n + B(\frac{1-\sqrt{5}}{2})^n$. Pourquoi ? On peut donner 3 méthodes : (1) L'ensemble des suites vérifiant $\forall n \geq 2, r_n = r_{n-1} + r_{n-2}$ est un espace vectoriel de dimension 2 ($(r_n)_{n\in\mathbb{N}} \to (r_0, r_1)$ est un isomorphisme vers $\mathbb{R}^2$). Y-a-t-il des solutions de la forme λ^n ? Cela marche ssi $\lambda^2 = \lambda + 1$ i.e. ssi $\lambda = \frac{1\pm\sqrt{5}}{2}$. Il suffit alors de remarquer que toute suite de cet espace, dont $Fibo$, est combinaison linéaire des deux solutions indépendantes $(\frac{1+\sqrt{5}}{2})^n$ et $(\frac{1-\sqrt{5}}{2})^n$. (2) Soit V_n le vecteur $\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$. On a $V_n = M \times V_{n-1}$, avec $M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. D'où $V_n = M^n V_0$. On souhaite donc calculer M^n. Pour cela, on souhaite diagonaliser M, et donc s'intéresse à ses valeurs propres qui s'avèrent être $\frac{1\pm\sqrt{5}}{2}$ ... (3) On a $\forall n \in \mathbb{N}, F_{n+2} = F_{n+1} + F_n$, et donc $\sum_{n\geq 0} t^{n+2} F_{n+2} = \sum_{n\geq 0} t^{n+2} F_{n+1} + \sum_{n\geq 0} t^{n+2} F_n$, ce qui revient à dire que la fonction $F(t) = \sum_{n\geq 0} t^n F_n$ vérifie

$F(t) - 1 - t = t(F(t) - 1)) + t^2 F(t)$, c'est donc $\frac{1}{1+t+t^2}$. Les F_n sont donc les coefficients du développement en série entière de cette fraction que l'on trouve en la décomposant en éléments simples, qui fait intervenir les racines du dénominateur à savoir $1/\frac{1\pm\sqrt{5}}{2}$...

Fibo(100) génère-t-il un arrêt "out of memory" ?



Quand on est en train de travailler un certain appel A, tous les appels entre la racine et A dans l'arbre sont en cours, il sont ouverts mais pas encore fermés, et ils sont dans la pile d'appels. Les appels à droite n'ont pas encore été ouverts, ils ne sont pas encore dans la pile d'appels. Les appels à gauche ont été ouverts mais aussi fermés, ils ne sont plus dans la pile d'appels. Pour ceux en dessous, cela dépend de où on en est du travail de A, mais là encore, soit ils ne sont pas encore dans la pile, soit ils n'y sont plus. La pile des appels ne contient que les appels sur la branche de l'abre de la racine vers A, soit 100 appels au max, pas de quoi faire exploser la mémoire, donc non il n'y a pas d'arrêt "out of memory" sur Fibo(100).

Il y a bien sûr moyen de calculer $Fibo(n)$ en un temps $\theta(n)$, pour cela, il suffit de calculer les nombres de Fibonacci successifs en ne retenant à chaque étape que les deux derniers. Cela donne le code itératif ci-dessous à gauche que l'on peut transformer en un code récursif, ci-dessous à droite.

```
fonction Fibo (int n ) : int       |   fonction Fibobis (int n, p, fibo, pred) : int
  fibo = 1          // Fibo(0)     |            // donne Fibo(n) en supposant que p <= n,
  pred = 0          // Fibo(-1)    |            // Fibo(p)=fibo et Fibo(p-1)=pred
  pour p de 1 a n                  |   si p == n alors rendre fibo
         tmp = fibo                |   sinon rendre Fibobis(n, p+1, fibo+pred, fibo)
         fibo = fibo + pred        |
         pred =  tmp               |   Fonction Fibo(int n) : int n
  rendre fibo                      |     rendre Fibobis(n,0,1,0)
```

Exercice : économiser un argument dans la sous-fonction Fibobis

Récursif ou itératif ?

Tout programme itératif peut être réécrit en récursif, il suffit de remplacer les codes de gauche par les codes de droite. La complexité de l'algo récursif sera du même ordre que celle de l'itératif qu'il simule.

```
  tant que test          |     ProcTantQue (variables de test et de truc)
       faire truc        |            si test alors  truc
                         |                           ProcTantQue(...)
-------------------------+-------------------------------------------------------
  pour i de x a y        |     ProcPourBis (i et autres variables de truc)
       faire truc        |            si i <= y alors  truc
                         |                              ProcPourBis( i+1, etc.)
                         |     ProcPour
                         |            ProcPourBis(x, etc.)
```

Réciproquement, tout programme récursif peut-il être réécrit en itératif ?

OUI. D'ailleurs le langage machine n'est pas récursif, or un compilateur transformera le programme récursif que vous avez écrit dans votre langage préféré, en un programme équivalent non récursif écrit en langage machine. Pour transformer un programme récursif, il "suffit" donc de singer le compilateur : "Tant que l'exécution n'est pas terminée, simuler un pas du compilateur". C'est un peu brutal ... Pour faire plus soft, essayons de comprendre ce que fait le compilateur. On ne va pas regarder tout ce qu'il fait, juste comment il change le récursif en itératif.

Reprenons l'image de l'exécution d'un programme récursif avec des piles de feuilles de papier. A chaque appel, on mémorise la ligne où l'on suspend le calcul, et on détermine le numéro de la première ligne de code de la fonctionnalité appelée. Par ailleurs, on mémorise les variables courantes, et on calcule les variables du prochain appel. Puis on continue l'exécution à la nouvelle ligne de code avec les nouvelles variables. Quand la fonctionnalité appelée se termine, on oublie la ligne de code courante et les variables courantes et on continue le calcul depuis la ligne code que l'on avait mémorisée avec les variables que l'on avait mémorisées. Les lignes de codes et les variables sont mémorisées avec les PILES de feuilles papier.

Une machine fait la même chose, sauf qu'à la place des piles des feuilles papier, il y aura une PILE de lignes de code et une PILE de variable. En gros, le compilateur fait :

Tant que ce n'est pas fini, exécuter la ligne de code courante, à savoir :

Si c'est un appel (récursif ou non), empiler la ligne de code courante sur la pile des appels et les variables courantes sur la pile des variables. Calculer les valeurs ou adresses des arguments et variables locales de la fonctionnalité appelée. Passer à la ligne de code du début de la fonctionnalité appelée.

Si c'est la fin d'une fonctionnalité, rendre les variables locales courantes, dépiler et reprendre pour le calcul courant les variables en sommet de pile des variables, dépiler la ligne de code en sommet de la pile des lignes de code, et se placer juste après cette ligne de code.

Si c'est la fin de la fonctionnalité principale, dire que c'est fini.

Dans les autres cas, exécuter la ligne de code courante et passer à la suivante.

On mémorise donc une pile de lignes de codes et une pile de variables En bref, on mémorise UNE PILE DES CHOSES RESTANT A FINIR, disant ce qu'il faut faire (pile des lignes de code) sur quelles données (pile des variables).

En pratique, on cherchera à simplifier et à rendre plus lisible : Par exemple, si ce qu'il faut faire (ligne de code) est toujours le même, il n'y a pas besoin de l'empiler (c'est ce qui se passera quand on dérécursifiera TriRapide (QuickSort)). Parfois, il sera plus pratique d'avoir une pile de paires (ligne de code, variables), ou au contraire d'avoir plusieurs piles pour plusieurs types de variables.

Notons que dérécursifier revient à simuler à la main ce qui est fait sinon par le compilateur. La complexité de la version dérécursifiée sera donc du même ordre que celle de la version récursive.

Exemple : Dérécursifions le code récursif naïf de Fibonacci. Rappelons que ce code récursif naïf est inefficace, les versions dérécursifiées illustreront la dérécursification, mais resteront tout aussi inefficaces que l'algo récursif de départ.

Quelle actions allons-nous effectuer ?: d'une part, calculer des nombres de Fibonacci, d'autre part additionner les deux derniers résultats pas encore traités. Pour faire `rendre F(n-1)+F(n-2)`, on va demander le calcul de `F(n-1)`, puis celui de `F(n-2)`, ce seront alors les deux derniers résultats pas encore additionnés. On additionnera ces deux nombre, la somme deviendra le dernier résultat pas encore additionné.

Nous aurons une pile $PileC$ de "ligne de code" contenant fib ou add. Une pile $PileN$ contenant les arguments des calculs demandés par fib et une pile $PileR$ contenant les résultats pas encore additionnés.

```
Fibonacci (N) : int
PileN = vide ; empiler N sur pileN :              Attention, code non efficace
PileC = vide ; empiler fib sur PileC ;
PileR = vide ;
tant que pileC est non vide
   depiler c de PileC, selon c :
       add : depiler r1 puis r2 de PileR, empiler r1+r2 sur PileR
       fib : depiler n de PileN :
             si n == 0 ou n == 1 alors empiler 1 sur PileR
                                 sinon empiler add puis fib puis fib sur PileC
                                       empiler n-2 puis n-1 sur PileN
depiler r de PileR et le rendre
```

En fin de compte, les valeurs r sont additionnées entre elles pour finir par faire le résultat. On peut aussi les ajouter progressivement au résultat à rendre. On ne va plus additionner les résultats entre eux, on va faire $Fibo{+}{+}$ quand on recontrera $n = 0$ ou $n = 1$

```
Fibonacci(N) : int
PileN = vide ; empiler N sur pileN ;              Attention, code non efficace
Fibo = 0
tant que PileN est non vide
    depiler n de PileN ; si n == 0 ou n == 1 alors Fibo++
                                             sinon empiler n-2 puis n-1 sur PileN
rendre Fibo
```

Exercices : Donner un algo récursif simple (non efficace) directement inspiré de ce dernier code.
Montrez que faire plutôt `empiler n-1 puis n-2` diminue la hauteur de la pile d'appel d'un facteur 2.

Récursif terminal

Si une procédure n'a plus rien à faire après un appel récursif, il est inutile d'empiler "plus rien à faire". On peut donc ne rien empiler, économie mémoire que fera un compilateur optimisé.

De la même manière, si une fonction termine son exécution par "rendre AppelRécursif", alors un compilateur optimisé ne va pas retenir qu'il faut récupérer le résultat de l'appelé et le transmettre à l'appelant, il va fermer l'appel courant, et demander à l'appelé de transmettre le résultat directement à l'appelant. En réalité, l'appelé et l'appelant risquent de faire de même, de sorte qu'en fin de compte, c'est un appel lointain devant qui va transmettre directement son résultat à un appel lointain derrière.

Puisqu'il n'y a rien à empiler, quand du récursif terminal est dérécursifié, que ce soit par un compilateur optimisé ou manuellement, on obtient un itératif sans pile. Par ailleurs, quand on récursifie de l'itératif, on trouve un récursif terminal. Ce qui explique que les fonctionnalités qui s'écrivent bien en itératif sont les mêmes que celles qui s'écrivent bien en récursif terminal.

Note 1 : Un appel récursif est terminal non pas si le texte du code se termine par l'appel mais si L'EXECUTION du code termine par l'appel. Dans `int fact(n) : si n == 0 alors rendre 1 sinon rendre n*fact(n-1)`, l'appel récursif n'est pas terminal car après en avoir reçu la réponse, il faut faire la multiplication par n. Dans `if test then AppelRécursif else AutreChose`, l'appel est terminal.

Note 2 : On peut donc transformer du récursif en récursif terminal, pour cela il suffit de le dérécursifier puis de le rerécursifier ? Certes, mais la dérécursification introduit des piles qui ne disparaitront pas à la rerécursification. On peut donc transformer automatiquement un récursif non terminal sans pile en un récursif terminal mais avec pile, ce qui n'a pas beaucoup d'intérêt.

Note 3 : Comment savoir si votre compilateur effectue l'optimisation ? Faites une fonctionnalité récursive terminale et lancer la avec une entrée sur laquelle elle ne termine pas. Si l'optimisation n'est pas faite, les appels vont s'empiler et le programme s'arrêtera avec un message "out of memory". Sinon il tournera indéfiniment et c'est vous qui allez devoir forcer manuellement son arrêt.

Evaluation paresseuse

Sachant que `"x mod 0"` plante, est-ce que le test `"b` $\neq$ `0 et a mod b == 0"` plante si $b = 0$? Cela dépend de la façon dont on calcule une expression de type `exp1 et exp2`.

Première méthode : On calcule les deux expressions quoiqu'il arrive, puis on rend un "et" des résultats, on fait `"a = exp1 ; b = exp2 ; si a alors rendre b sinon rendre faux"`. Ce type dévaluation est dit "courageux" (En réalité, il est plus stupide que courageux...)

Deuxième méthode : On observe que quand `exp1` est faux, l'expression `exp1 et exp2` va être fausse, il n'y a pas besoin de calculer `exp2` pour le dire, et on peut donc se passer ici du calcul de `exp2`, que l'on n'effectuera que si nécessaire. On va faire : `"si exp1 alors rendre exp2 sinon rendre faux"`. Ce type dévaluation est dit "paresseux" (En réalité, il est plus habile que paresseux...)

Dans le premier cas, l'échec du premier test `"b` $\neq$ `0"` n'empêche pas l'évaluation du second qui est pire qu'inutile car elle va faire planter. Dans le second cas, suite à l'échec du premier test, le second n'est pas évalué et ça ne plante pas.

A peu près tous les compilateurs effectuent l'évaluation paresseuse (c'est dans la norme d'à peu près tous les langages), ce qui va permettre une économie de calcul, mais va surtout éviter les plantages. En réalité, cela va surtout nous permette de coder plus léger (si les évaluations étaient courageuses, nous en tiendrions compte, ce qui nous obligerait à faire des codes plus lourds).

Notez que l'ordre devient important : `"a mod b == 0 et b` $\neq$ `0 "` plante si $b = 0$. Le "et" n'est pas commutatif ! Il l'est pour les logiciens pour qui les variables booléennes ne peuvent prendre que deux valeurs : vrai et faux. Il ne l'est pas pour les programmeurs car ils ont une troisième valeur "problème" (plantage, non-terminaison, etc.), qui ne commute pas : "faux et pbm" donne faux alors que "pbm et faux" donne pbm. (Le "et" est aussi non commutatif s'il y a des effets de bords.)

Nous supposerons systématiquement que l'évaluations du "et", et celle du "ou", sont paresseuses.

Variables

Une variable est un espace mémoire dans lequel vouz pouvez écrire et lire, et qui a :

- un TYPE (entier, booléen, etc.) qui sera pris en compte à la compilation
- une ADRESSE, attribuée à l'exécution, et fixée pour la durée de vie de la variable.

• une VALEUR, ce qui est écrit dans cette mémoire, qui évolue lors de l'exécution.

Certaines expressions ont un type et une valeur, mais pas d'adresse, ce ne sont pas des variables.
Ce sont des variables : `x, T[i], structure.champ, *p`
Ce ne sont pas des variables : `23, x+y, NULL, &n, fact(n), a==b`

Une égalité `e1 == e2` rend vrai ssi les VALEURS des expressions `e1` et `e2` sont égales. Une affectation `v = e` calcule la VALEUR de l'expression `e`, et écrit cette valeur à l'ADRESSE de la variable `v`. Le compilateur refusera `"0 = x"` car le côté gauche n'est pas une variable. Les affirmations précédentes enfoncent des portes ouvertes, mais il faudra s'y référer précisement pour bien comprendre les pointeurs.

L'espace mémoire des variables locales (et des arguments passés par valeur) d'un appel à une fonctionnalité, est fourni à l'exécution par le gestionnaire de mémoire au moment de l'ouverture de l'appel et il est repris par le gestionnaire au moment de la fermeture de l'appel. On peut donc dire que ces variables n'existent pas avant l'appel et n'existent plus après l'appel. Les variables globales existent du début à la fin de l'exécution du programme. L'utilisateur peut manuellement réserver et rendre des variables autres que celles déclarées, cf le chapitre sur les pointeurs, sections "malloc" et "free".

Une variable non initialisée n'est pas vide, elle contient n'importe quoi. Si vous faites afficher une variable entière non initialisée, vous verrez apparaître un entier loufoque.

Pointeurs

Un pointeur vers un truc, est un objet dont la valeur est l'adresse d'une variable de type truc.
Exemple, un pointeur vers une maison est une carte de visite sur laquelle est notée l'adresse de la maison.

Cf le prochain chapitre "Algo, Introduction 3 : Pointeurs"

Vous avez besoin d'avoir compris ce qu'est un pointeur et ce que sont les opérateurs & et * avant de lire ce qui suit (intervention des pointeurs dans le passage par adresse)

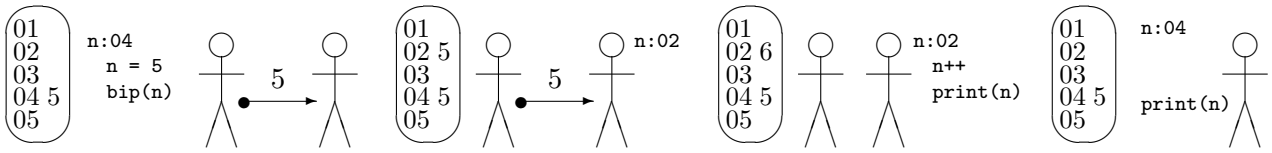
Modes de passage : par valeur, par variable, par adresse et quelques autres

Passage par valeur (C'est le passage par défaut)

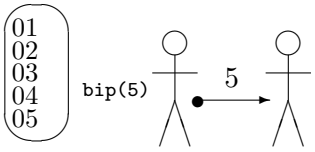
Ce mode de passage est utilisé pour les arguments destinés à être lus

L'appelant évalue la VALEUR de l'argument et l'envoie à l'appelé. Celui-ci demande au gestionnaire de mémoire un nouvel espace pour sa variable, et il y écrit la valeur transmise par l'appelant. Il rendra cet espace mémoire quand il se fermera. L'appelé travaille donc avec une COPIE de l'argument de l'appelant. Si l'appelé change la valeur de sa variable, cela n'a pas de répercussion pour la variable de l'appelant.

Dans l'exemple ci-dessous, `bip` a un argument par valeur. L'appelé fait imprimer un **6**, mais l'appelant fait imprimer un **5** car sa variable n'a pas changé malgré le `n++` de l'appelé.



Puisque seule la valeur de l'argument compte, il n'est pas nécessaire que ce soit une variable et on peut appeler `bip(5)`

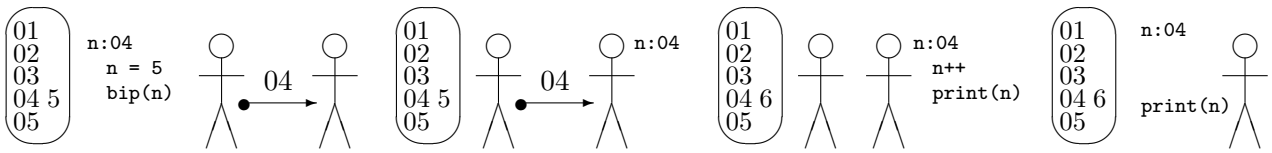


Passage par variable alias **Passage par référence**

Ce mode de passage est utilisé pour les arguments destinés à être modifiés ou initialisés.

L'appelant évalue l'ADRESSE de l'argument et l'envoie à l'appelé qui associe cette adresse à sa variable. L'appelé travaille donc avec l'ORIGINAL de l'argument de l'appelant. Si l'appelé change la valeur de sa variable, cela modifie la variable de l'appelant.

Dans l'exemple ci-dessous, `bip` a à présent un argument par variable. L'appelé fait imprimer un **6** et l'appelant fait aussi imprimer un **6** car le `n++` de l'appelé a modifié la valeur de la variable de l'appelant.



Puisque c'est l'adresse de l'argument qui est envoyé, il faut que ce soit une variable.

Passage par adresse

En C, il n'y a que le passage par valeur. Son usage avec des pointeurs remplace le passage par variable.

Pour faire modifier ou initialiser une variable n, par exemple entière, l'appelant fait pointer un pointeur p vers n et passe ce pointeur à l'appelé. Puisque c'est un passage par valeur, il comnunique la valeur de p, c.-à-d. l'adresse de n. In fine, l'information transmise est donc la même que dans un passage par variable. Simplement, le compilateur la prend comme valeur de p plutôt que comme adresse de n.

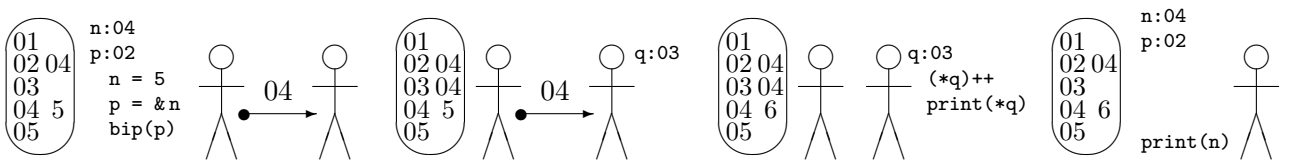
L'appelé receptionne cette valeur de pointeur, donc l'adresse de la variable entière n. Il demande au gestionnaire mémoire un espace pour un pointeur q et il y stoque la valeur reçue, soit l'adresse de n.

L'appelé travaille donc avec un pointeur q copie du pointeur p de l'appelant, mais original et copie pointent tous les deux vers n car copier un pointeur signifie recopier l'adresse. De la même manière que si je photocopie votre carte de visite, vous et moi aurons deux cartes de visite distinctes, mais les maisons pointées par nos cartes de visite respectives seront une seule et unique même maison.

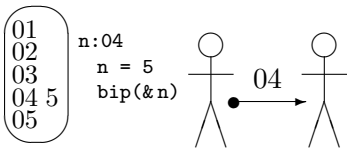
Si l'appelé modifie q, c'est sans conséquence pour p. Mais s'il modifie $*q$, cela modifie $*p$ puisque c'est la même variable. De la même façon, supposons que j'ai une photocopie de votre carte de visite. Si je rature ma photocopie, c'est sans effet pour vous. Mais si je fais repeindre les volets de la maison dont l'adresse figure sur ma photocopie, cela change l'apparence de la maison dont l'adresse figure sur votre carte de visite, puisque c'est la même maison.

Le but de toute cette opération est de faire modifier ou initialiser n par l'appelé, ce qu'il va pouvoir faire puisqu'il a accès à n sous le doux nom de $*q$.

Dans l'exemple ci-dessous, appelé puis appelant font imprimer **6**



Et puisque c'est un passage par valeur d'un pointeur vers n, il n'y a pas besoin de passer par une variable p, seule la valeur $\&n$ compte, et on peut appeler directement avec cette valeur.



Note : Si l'on appelle une procédure avec un passage par adresse, c'est pour transmettre indirectement une variable au travers d'un pointeur, il seait donc anormal (mais techniquement possible) d'appeler une telle procédure avec un pointeur NULL. Il est donc d'usage pour ce type de passage que l'appelé suppose que le pointeur reçu n'est pas NULL, et qu'il ne le vérifie pas.

Autre mode de passage : copie-in-copie-out. On peut trouver ce mode de passage en Ada.

L'appelé fait une copie de la variable de l'appelant comme dans un passage par valeur. Mais à la fin de la procédure, le contenu des variables de l'appelé est recopié dans la variable de l'appelant.

Exemple tordu : {`truc(x,y) : x++ ; y++ ;`} {`main: n = 5 ; truc(n,n) ; print(n) ;`} qui imprime 5 avec un passage par valeur, 7 avec un passage par variable, 6 avec un passage copie-in-copie-out.

Autre mode de passage : substitution. C'est le mode de passage des macros de C.

Un "préprocesseur" fait, au niveau de l'appel, un copié-collé du code de l'appelé en remplaçant les variables de l'appelé par les arguments de l'appelant. Exemple : le préprocesseur transformera {`Echange (x,y): tmp = x ; x = y ; y = tmp ;`} {`Main: a = 2 ; b = 3 ; Echange(a,b) ;`} en {`Main: a = 2 ; b = 3 ; tmp = a ; a = b ; b = tmp ;`}

Le copié-collé est généralement fait stupidement et a des effets indésirés. Avec le `Echange` ci-dessus, `Echange(T[i],i)` fait la même chose qu'avec un passage par variable, mais `Echange(i,T[i])` fait autre chose. Ou encore, {`Plus(x,y): x+y`} {`Main: print(3*Plus(6,5)) ;`} risque d'imprimer 23 !

Modes de passage in, out, inout

Nous utiliserons en pseudo-code les terminologies suivantes (inspirées de Ada) :

IN pour les arguments destinés à être lus par l'appelé. En pratique, ils seront passés par valeur.

INOUT pour les arguments destinés à être modifiés par l'appelé. Ces arguments doivent être des variables. En pratique, ils seront passés par variable ou par adresse.

OUT pour les arguments destinés à être initialisés par l'appelé. Ces arguments doivent être des variables déclarées par l'appelant qui les envoie non initialisées à l'appelé. En pratique, ils seront passés par variable ou par adresse. La valeur de sortie d'une fonction est une variable OUT implicite.

Exemples d'appels : `Fact(in n)` (5 en in, 120 en out), `Tri(inout T[])`, `Initialise(out x)`

Expressions et instructions

Une expression est quelquechose qui a un type (entier par exemple) à la compilation et une valeur à l'exécution. `0, x, x+y, T[i], a==b` sont des expressions. L'appel à une variable est une expression. Il y a des expressions qui ne sont pas des variables.

Une instruction est une action, une chose à faire. `print exp, var = exp, retourner exp, if exp then instr else instr` sont des instructions. Un programme impératif est une suite d'instructions (CAML n'est pas impératif mais fonctionnel).

Dans ce cours, vous devrez programmer proprement et bien distinguer expressions et instructions. Vous ne pouvez donc pas faire tout ce qu'il est possible de faire en C. Par exemple, il vous est interdit d'écrire `i = j++`. En effet, `j++` y est à la fois une instruction (incrémenter `j`, c'est donc une affectation `j = j+1`) et une expression (rend une valeur qui est affectée à i).

`variable++` est autorisé mais c'est une instruction, il est interdit comme expression.

Fonctions et procédures

Dans ce cours, vous devrez programmer proprement et bien distinguer fonctions et procédures. Vous ne pouvez pas faire tout ce que C laisse faire (mais que d'autres langages, Ada par exemple, empêchent)

Une fonction LIT des arguments et REND un résultat. Une fonction ne modifie pas ses arguments qui sont tous IN. Elle fait des `rendre` dans tous les cas (sauf sur les entrées illégales qui peuvent planter ou afficher un message d'erreur). Quand on la code, on répond à la question "Que dois-je rendre". Si f est une fonction, `f(Arguments)` est une expression.

Une procédure PREND des arguments et FAIT quelque chose, classiquement, elle imprime, elle modifie ou elle initialise ses arguments. Les arguments d'une procédure peuvent être IN, OUT ou INOUT. On n'y trouve aucun `rendre`. Quand on la code, on répond à la question "Que dois-je faire". Si p est une procédure, `p(Arguments)` est une instruction.

Exemple, pour passer de n à $n+1$, il y a une fonction et une procédure, ci-dessous à gauche, que l'on ne confondra pas. Le code ci-dessous au centre montre comment `succ` ne modifie pas p tandis que `incr` le fait. Ci-dessous à droite deux exemples de mauvaises utilisations (qui ne compilent pas dans un langage propre) de `succ` et `incr`, suite à une confusion expression VS instruction, fonction VS procédure.

```
            // successeur   |  p = 4 ;                |     Deux nonsenses dans :
fonction succ (in n)        |  q = succ(p) ;          |  if  test
   rendre n+1               |  print(q) ;     // 5    |  then  q = incr(p) ;
            // incremente   |  print(p) ;     // 4    |    // incr(p) pas une expression
procedure incr (inout n)    |  incr(p) ;              |  else  succ(p) ;
    n++                     |  print(p) ;     // 5    |    // succ(p) pas une instruction
```

Que peut-on faire avec ces fonctionnalités ? :

succ(succ(n)) : OUI, le `succ` intérieur attend une expression en argument, on lui donne une variable, ce qui est bien une expression. Par ailleurs, `succ` est une fonction, donc `succ(n)` est une expression. Le `succ` extérieur attend une expression en argument, c'est bien le cas, et `succ` est une fonction, donc `succ(succ(n))` est une expression. C'est la composition de fonctions des mathématiciens.

incr(succ(n)) : NON, `succ(n)` est une expression, mais n'est pas une variable. Or `incr` attend une variable comme argument.

succ(incr(n)) : NON, `incr(n)` est légal : `incr` attend une variable en argument, et on lui donne une variable, et `incr` est une procédure, donc `incr(n)` est une instruction. Mais `succ(n)` attend une expression comme argument et on lui donne une instruction.

incr(incr(n)) : NON, `incr(n)` est légal et c'est une instruction. Le `incr` extérieur attend une variable en argument. Or on ne lui donne pas une variable, même pas une expression, mais une instruction.

Il est probable que celui qui a écrit `incr(incr(n))` voulait incrémenter deux fois, c.-à-d. qu'il voulait incrémenter PUIS incrémenter une deuxième fois. Il ne sagit pas ici de composer des fonctions, mais de faire successivement deux instructions. Cela se fait grâce à l'instruction bloc : **{ incr(n); incr(n); }**.

Algo, Introduction 3 : Pointeurs

Algo, Listes 1 : Généralités

Une liste de trucs est donnée par un entier $N \in \mathbb{N}$, la longueur de la liste, et une suite finie $[\, e_1, ..., e_N \,]$ d'éléments de type truc.

L'entier N peut être égal à 0. C'est la liste vide $[\,]$.

UNE LISTE VIDE EST UNE LISTE PARFAITEMENT NORMALE ET LÉGALE

J'assassine au partiel celui qui fait afficher une erreur sur la liste vide pour n'importe quelle fonction, `Longueur` par exemple (Il y a cependant des fonctions qui donnent une erreur sur la liste vide, `PremierElement` par exemple)

Une liste est triée (croissante) ssi $\forall i, j \in [1, N], i < j \Rightarrow e_i \leq e_j$. Notez que la liste vide est triée.

Exemples : $[3, 4, 5]$ est une liste d'entiers. [] la liste vide, est une liste de ce que l'on veut. $[[3, 5], [], [7, 9]]$ est une liste de listes d'entiers. $[[]]$, singleton contenant la liste vide, est une liste de liste de ce que l'on veut (attention à ne pas confondre [] et $[[]]$). $[[], 6]$ n'est pas une liste car les éléments ne sont pas tous du même type.

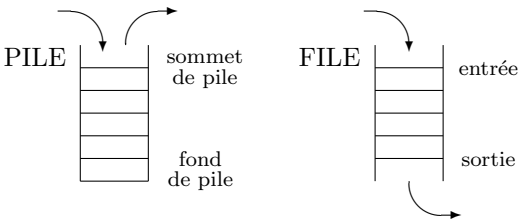
Taille d'une liste : formellement, c'est $N + \sum_i |e_i|$.

En pratique, les objets sont souvent de taille prédéfinie (booléen, ou bien entier qui ne sont pas les vrais entiers de $\mathbb{Z}$ mais plutôt ceux de $]-2^{31}, 2^{31}]$ codés sur 32 bits), ou bien on souhaite faire comme si, c.-à-d. que l'on on néglige le fait que les objets ont une taille individuelle.

On dira alors que la longueur N est la taille de la liste.

C'est parfaitement justifié pour des listes de booléen. Ca l'est à moitié pour des entiers. Ca ne l'est pas pour les listes de listes.

Piles et Files



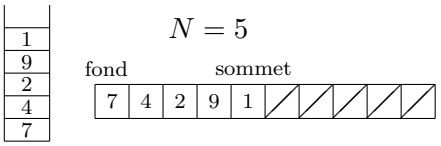
Les deux opérations clefs des piles et des files sont insérer un élément et sortir un élément. Ces opérations se font par les extrémités. C'est le même côté pour l'entrée et la sortie pour la pile. Ce sont des côtés opposés pour la file.

Dans une pile, si x a été inséré avant y, x sortira **après** y.

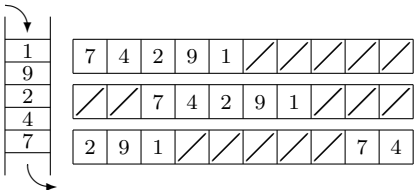
Dans une file, si x a été inséré avant y, x sortira **avant** y. On parle de FIFO "First-In-First-Out".

Représentations : Tableaux

Une liste peut être représentée par l'entier N et un tableau $T[1..M]$ (ou $T[0..M-1]$ dans certains langages) avec $M \geq N$. Les éléments son rangés dans les N premières case, les autres cases sont sans signification. Une pile est représentable ainsi avec le fond en première case et le sommet en N^{ieme} case. On fera {`rendre T[N];`} pour regarder l'élément en sommet de pile, {`N++; T[N] = e;`} pour empiler e, {`N--;`} pour dépiler.



Pour une file, les éléments sont insérés par la droite, comme pour une pile, et sortis par la gauche. Quand on sort un élément, on ne décale pas les éléments restants (ce serait trop coûteux), on se contente d' "effacer" la première case, de sorte que la file ne va plus nécessairement commencer à la première case du tableau. La file va progressivement glisser vers la droite. Quand elle atteint le bord droit du tableau, elle repart depuis le bord gauche. Le tableau est traité comme un ruban circulaire.



Il faudra garder en mémoire deux valeurs N, et S la case de sortie. On fera {`T[S`$\oplus_N$`N] = e; N++;`} {`N--; S = S`$\oplus_N$`1`} {`rendre T[S];`} (où $\oplus_N$est l'addition modulo N), pour insérer e, pour sortir, pour regarder l'élément en bout de file. (Garder S et E la case d'entrée est maladroit car si $E = S$, vous ne saurez pas si la pile est vide ou pleine. Garder S est plus logique que garder E car c'est l'élément en sortie que l'on consulte)

Dans un tableau avec chaînage des cases, chaque case du tableau contient deux parties : un élément de liste et un indice de tableau.

[23,89,72,92,51,17] Debut : 3

1	2	3	4	5	6	7	8	9	10
72		23		51	92		89	17	
6		8		9	5		1	0	

Une variable extérieure `Debut` dit dans quelle case du tableau se trouve le premier élément. `Debut` vaut 0 si la liste est vide. Chaque case utilisée du tableau contient un élément de la liste et l'indice de case contenant le prochain élément.

On est proche d'une structure de pile : on peut rapidement consulter l'élément de tête. On peut rapidement éliminer l'élément de tête. Par contre, il y a une difficulté pour insérer en tête car il n'est pas évident de trouver une case libre. La solution est de faire un chaînage des cases libres et d'en donner la tête dans `Libre`. Quand il n'y a plus de place, `Libre` vaut 0.

Libre : 4 Debut : 3

1	2	3	4	5	6	7	8	9	10
72		23		51	92		89	17	
6	7	8	10	9	5	0	1	0	2

Cette structure gère en réalité manuellement une liste chaînée, gestionnaire de mémoire inclus !

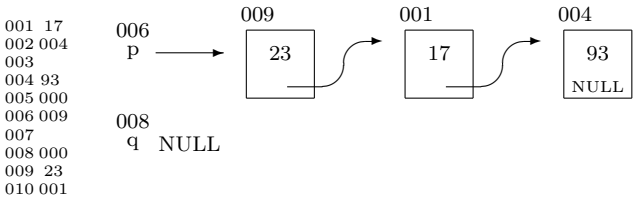
Représentations : Listes chaînées

Une liste chaînée consiste en

(1) une suite de blocs, un par élément de la liste, contenant d'une part l'élément de la liste, d'autre part un pointeur vers le bloc du prochain élément. Le pointeur du bloc du dernier élément est NULL

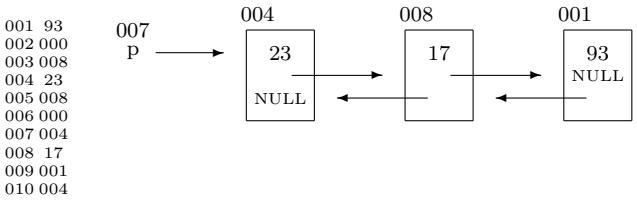
(2) Un pointeur vers le bloc du premier élément. Si la liste est vide, ce pointeur est NULL.

Dans l'exemple ci-contre, p est une variable pointeur représentant la liste $[23, 17, 93]$ et q est une variable pointeur représentant la liste vide.

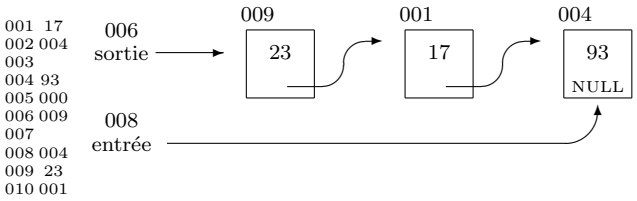


Cette strucuture permet d'implémenter les PILES.

Il est possible d'ajouter un pointeur vers le prédécesseur, on obtiendra alors une liste "doublement chaînée".



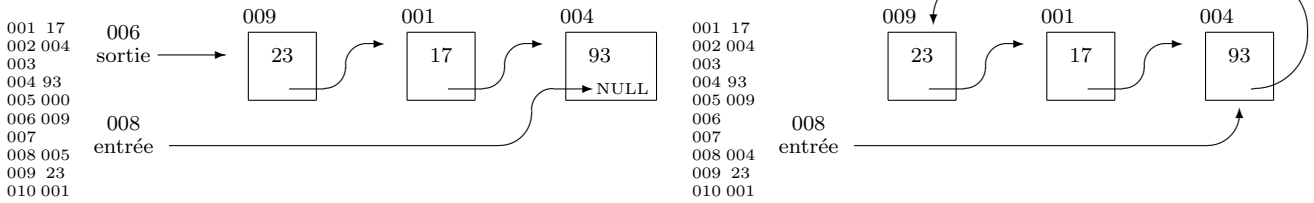
Pour une file, on ajoute un deuxième pointeur vers le dernier bloc de la liste (ou égal à NULL si la liste est vide). Il est alors facile d'insérer à chaque extrémité, mais il ne sera facile de mettre à jour la liste lors des sorties qu'en tête de liste. La tête de liste est donc la sortie et la fin de la liste est l'entrée.



Ci dessous à gauche, une première variante fait pointer `entrée` non pas vers le bloc, mais vers le pointeur dans le bloc (et donc `entrée` est de type pointeur vers pointeur vers bloc). Si la file est vide, le pointeur de pointeur `entrée` pointera alors vers le pointeur `sortie`.

Ci dessous à droite, une deuxième variante n'a que le pointeur entrée, mais la liste est circulaire, ce qui permet d'accéder à la tête de liste en passant par la fin de liste. La liste vide sera alors codée par un pointeur NULL et sera à traiter de façon spécifique.

Les deux variantes sont implémentables en même temps.



Différences entre listes chaînées et tableaux

D'une part, l'espace consommé est différent. Les listes chaînées gèrent mieux l'imprévisibilité éventuelle de la longeur de la liste mais consomment de l'espace en plus pour les pointeurs.

D'autre part, la complexité des opérations peut être très différente. Selon les opérations à effectuer, il sera plus rentable de prendre l'une ou l'autre représentation.

Algo, Listes 2 : Listes-Piles

Nous allons abstraire les listes chaînées en listes-piles. Il est préférable, même si nous ne manipulerons pas directement les pointeurs et les blocs, de comprendre les structures chaînées sous-jacentes.

Briques de base

Nous supposerons que sont disponibles les fonctions et procédures suivantes :

• **Fonction EstVide (in liste L) : booléen** qui rend vrai ssi la liste est vide.
Son code avec des listes chaînées est :

```
fonction EstVide (liste L) : booleen
                    return L == NULL ;
```

On pourra écrire `L == []` à la place de `EstVide(L)`

• **Procédure InitVide (out liste L)** qui initialise la liste à vide.
Avec des listes chaînées et le passage par variable, on appelle `InitVide(L)` et le code est :

```
procedure InitVide ( VAR liste L) : booleen
                    L = NULL ;
```

Avec des listes chaînées et le passage par adresse, on appelle `InitVide(&L)` et le code est :

```
procedure InitVide (liste* P) : booleen
                    *P = NULL ;    // P est un pointeur vers le pointeur L
```

On pourra écrire `L = []` à la place de `InitVide(L)`

• **Fonction Premier (in liste L) : élément** qui rend le premier élément de la liste.
Son code avec des listes chaînées est :

```
fonction EstVide (liste L) : booleen
                    return (*L).element ;
```

L est le pointeur de départ, $*L$ est le premier bloc, et $(*L).element$ est le champ élément du bloc.

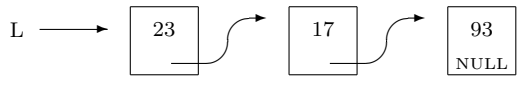
En C, `->` signifie $*$. et on pourra donc écrire `return L->element`

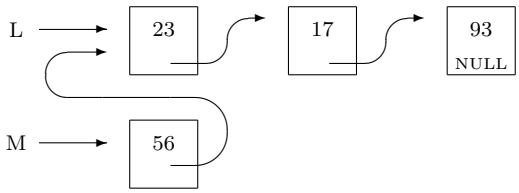
ATTENTION, `Premier` plante sur une liste vide, il faut donc impérativement la lancer avec un argument dont on est sûr qu'il est non vide. Classiquement : `si L == [] alors...sinon...Premier(L)...`

• **Inserton d'un élément**: une fonction **Ajoute** et une procédure **Empile**:

• **Fonction Ajoute (in élément x, in liste L) : liste** rend une liste dont le premier élément est x et qui continue par L. La mémoire est partagée avec L sauf bien sûr pour le bloc contenant x. C"est une fonction qui rend un pointeur liste et qui ne modifie pas L.

Exemple ci-dessous, `M = ajoute(56,L)` fait passer de la configuration de gauche à celle de droite.





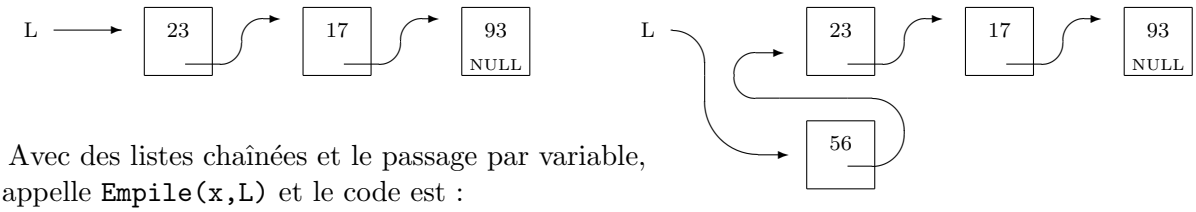
Son code avec des listes chaînées est :

```
Fonction Ajoute (element x, liste L) : liste
                    tmp = (liste) malloc(sizeof(Bloc)) ;
                    (*tmp).element = x ;
                    (*tmp).suite = L ;
                    return tmp ;
```

Cette fonctionnalié sert essentiellement à construire des listes récursivement. Exemple :

```
Fonction DeNaUn (int n) : liste  // rend la liste [n,n-1, ... ,2,1]
si n == 0  alors rendre [] ;
         sinon rendre ajoute(n,DeNaUn(n-1)) ;
```

• **Procédure Empile (in élément x, inout liste L)** modifie la liste L en y insérant x en tête.
Exemple ci-dessous, `empile(56,L)` fait passer de la configuration de gauche à celle de droite.



Avec des listes chaînées et le passage par variable, on appelle `Empile(x,L)` et le code est :

```
Procedure Empile (element x, VAR liste L)
              tmp = (liste) malloc(sizeof(Bloc)) ;
              (*tmp).element = x ;
              (*tmp).suite = L ;
              L = tmp ;
```

Avec des listes chaînées et le passage par adresse, on appelle `Empile(x,&L)` et le code est :

```
Procedure Empile (element x, liste* P)  // P est un pointeur vers le pointeur L
              tmp = (liste) malloc(sizeof(Bloc)) ;
              (*tmp).element = x ;
              (*tmp).suite = *P ;
              *P = tmp ;
```

En C, on peut remplacer `(*tmp).element` et `(*tmp).suite` par `tmp->element` et `tmp->suite`
`Empile(x,L)` est en fait équivalent à `L = Ajoute(x,L)`.
Cette fonctionnalité permet de construire des listes itérativement. Exemple :

```
Fonction DeNaUn (int n) : liste  // rend la liste [n,n-1, ... ,2,1]
      R = []
      pour i de 1 a n
             Empile(i,R) ;
      rendre R
```

Elle permet également de modifier (récursivement) une liste en y insérant des éléments (Cf ci-dessous `PointeurSuite`).

• **Retrait d'un élément**: la fonction **Suite**, la procédure **Dépile**, la pseudo-fonction **PointeurSuite**

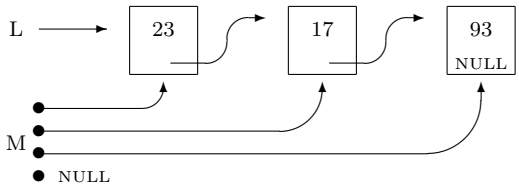
ATTENTION, les trois fonctionnalités `Suite`, `Dépile`, et `PointeurSuite` plantent sur une liste vide, il faut donc impérativement les lancer avec un argument dont on est sûr qu'il est non vide. Classiquement : `si L == [] alors ...  sinon ...  Suite(L) ...`

• **Fonction Suite (in liste L) : liste** rend un pointeur liste vers L sans son premier élément. La mémoire est partagée avec L et donc `Suite` rend un fait un pointeur vers le second bloc de la liste L, ou un pointeur NULL si la liste L est un singleton. C"est une fonction qui rend un pointeur liste et qui ne modifie pas L.
Exemple ci-dessous, `M = Suite(L)` fait passer de la configuration de gauche à celle de droite.

`Suite(L)` st un pointeur copie du pointeur `(*L).suite` et son code est tout simplement :

```
Fonction Suite (liste L) : liste
                    return(*L).suite ;
```

Faire `M = Suite(M)` permet d'avancer M d'un bloc, un peu comme si vous tourniez la page d'un livre, et `M = L ; tant que M` $\neq$ `NULL { M = Suite(M) ;}` permet de parcourir la liste L, un peu comme si lisiez toutes les pages d'un livre.

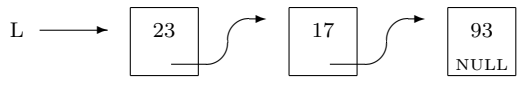


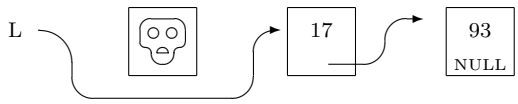
Cette fonctionnalité permet de parcourir des listes pour les lire, itérativement ou récursivement.

• **Procédure Dépile (inout liste L)** modifie la liste L en enlevant l'élément de tête.

Cet élément est **DETRUIT**, il est rendu à la mémoire par un free.

Exemple ci-dessous, `depile(L)` fait passer de la configuration de gauche à celle de droite.





`Depile(L)`, c'est arracher la première page du livre L.

Avec des listes chaînées et le passage par variable, on appelle `Depile(L)` et le code est :

```
Procedure Depile ( VAR liste L)
              tmp = L ;
              L = (*L).suite ;
              free(tmp) ;
```

```
variante :
              tmp = (*L).suite ;
              free(L) ;
              L = tmp ;
```

Avec des listes chaînées et le passage par adresse, on appelle `Depile(&L)` et le code est :

```
Procedure Depile (liste* P)
              tmp = *P ;
              *P = (**P).suite ;
              free(tmp) ;
```

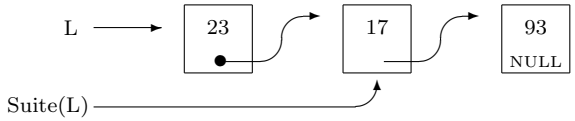
```
variante :
              tmp = (**P).suite ;
              free(*P) ;
              *P = tmp ;
```

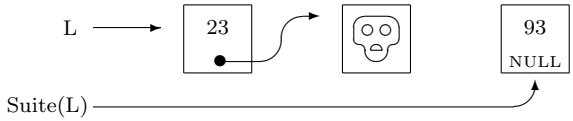
`depile(x,L)` est en fait équivalent à {`tmp = L ; L = Suite(L) ; free(tmp) ;`}.

Cette fonctionnalité permet de modifier (récursivement) une liste en y enlevant des éléments (Cf `PointeurSuite`).

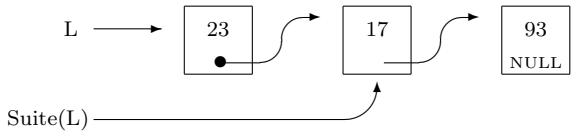
• **Pseudo-Fonction PointeurSuite (in liste L) : liste**

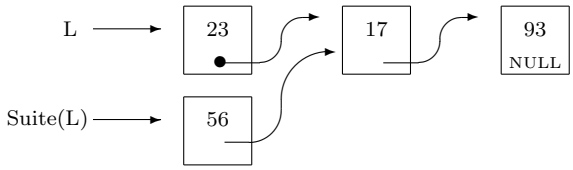
Supposons que vous vouliez enlever et détruire le second élément d'une liste (supposée n'être ni vide ni un singleton). `Depile(Suite(L))` ferait passer de la configuration de gauche à celle de droite





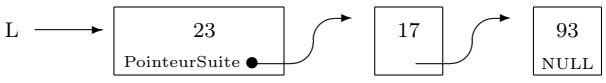
De la même manière, supposons que vous vouliez insérez un élément en seconde position d'une liste (supposée non vide). `Empile(56,Suite(L))` ferait passer de la configuration de gauche à celle de droite.





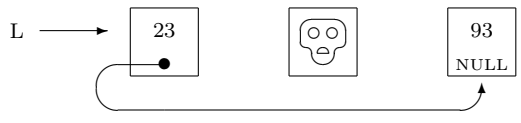
On constate qu'il y a un problème, les destructions et constructions ont bien eu lieu, mais le pointeur marqué d'un • n'a pas été mis à jour, c'est sa copie de pointeur `Suite(L)` qui l'a été. (Et en réalité, le compilateur devrait même refuser `Depile(Suite(L))` et `Empile(56,Suite(L))` car les procédures `Depile` et `Empile` attendent en argument une variable, ce que `Suite(L)` n'est pas.)

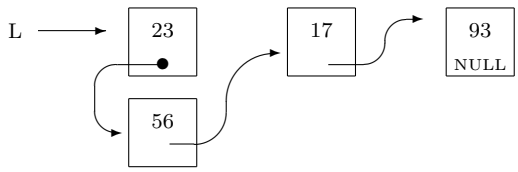
Il faut donc appeler `Depile`, `Empile`, et tout appel récursif destiné à modifier la liste, non pas avec `Suite(L)` mais avec la VARIABLE marquée d'un •, que l'on désignera par `PointeurSuite(L)`. Notez que `PointeurSuite` n'est pas une vraie fonction puisque son résultat est une variable.



Avec un passage par variable, il faudra faire `Appel( VAR (*L).suite)` et ne pas utiliser la fonction `Suite`. Avec un passage par adresse, il faudra appeler avec un pointeur vers cette variable, ce qui donnera `Appel(&((*L).suite))`, et si on est sur un appel récursif, L sera donné par un pointeur P qui pointe vers L, et du coup on fera `AppelRecursif(&((**P).suite)` (ou `AppelRecursif(&((*P)->suite))`

`Depile(PointeurSuite(L))` et `Empile(56,PointeurSuite(L))` auront alors les effets ci-dessous, qui sont les effets souhaités :





Cette fonctionnalité permet de parcourir récursivement les listes pour les modifier.

Exemple 1 : Fonction Longueur

Version récursive simple

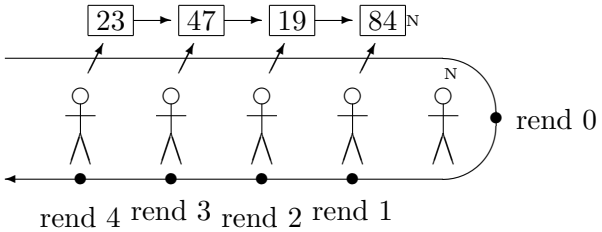
On raisonne en "diviser pour régner", comme si on faisait une récurrence. Cas de base : La liste vide, elle est de longueur 0. Sinon on peut toujours appeler récursivement . Est-ce que la connaissance de la longueur de la suite de L est pertinente pour donner la longueur de L ? Oui, il suffit d'ajouter 1 au résultat reçu pour obtenir le résultat à envoyer. D'où le code :

```
Fonction longueur (liste L) : entier          // version 1
        si L est vide
        alors rendre 0
        sinon rendre 1+longueur(Suite(L))
```

En mettant en avant les pointeurs, cela donne :

```
Fonction longueur (liste L) : entier          // version 1bis
        si L == NULL
        alors rendre 0
        sinon rendre 1+longueur((*L).suite)
```

Notons que cette fonction compte de façon contre-intuitive. Quelle est la longueur de [23,47,19,84] ? Vous dites 4, mais comment avez-vous compté ? A priori de gauche à droite : 1 sur le 23, 2 sur le 47, 3 sur le 19, 4 sur le 84. Cette version 1 compte dans l'autre sens, de droite à gauche : 1 sur le 84, 2 sur le 19, 3 sur le 47, 4 sur le 23. En effet, les appels se sont empilés jusqu'à celui qui a une liste vide et qui démarre le calcul en rendant 0. Puis au retour des appels, des 1 sont progressivement ajoutés.



La version dérécursifiée de cette version serait quelque chose comme :

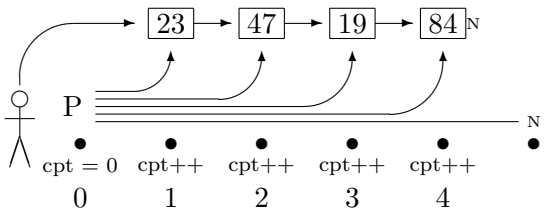
```
Fonction longueur (liste L) : entier          // version 2
        P = L
        H = vide
        tant que P est non vide
                empiler("ajouter 1") sur H
                P = Suite(P)
        cpt = 0
        tant que H est non vide
               cpt++
               depiler H
       rendre cpt
```

Pour `longueur`, il serait absurde de se prendre la tête comme ça en itératif. La technique peut servir pour faire en itératif des fonctionnalités qui demanderaient de lire la liste de droite à gauche.

Version itérative simple

On parcourt la liste, et tout en parcourant la liste, on incrémente un compteur. Il faut initialiser le compteur, puis il ne faudra pas oublier de le rendre :

```
Fonction longueur (liste L) : entier          // version 3
       P = L
       cpt = 0
       tant que P n'est pas vide
                cpt++
                P = Suite(P)
      rendre cpt
```



Remarque 1 : Ne doit-on pas commencer par tester si la liste est vide ? Non, on n'est pas en récursif. Il est prudent de vérifier que le code se comporte comme prévu sur liste vide et si oui, on ne va pas traiter la liste vide à part (et si on découvre que non parce qu'il y a une vraie bonne raison d'isoler la liste vide, alors on fera le test liste vide pour l'isoler). C'est comme pour une démo en math : Si je vous demande de démontrer que pour $\forall n \in \mathbb{N}, n + n^2$ est pair. Soit vous le faites par récurrence ($0 + 0^2$ est pair, puis si $n + n^2$ est pair alors $(n+1) + (n+1)^2 = (n + n^2) + 2(n + 1)$ aussi), soit vous ne le faites pas par récurrence, et vous direz par exemple que si n est pair, alors $n + n^2$ est pair comme somme de deux pairs, sinon $n + n^2$ est pair comme somme de deux impairs. La démo ci-avant fonctionne pour $n = 0$ et il serait inutilement lourd de dire "si n est nul alors $0 + 0^2 = 0$ est pair sinon si n est pair alors $n + n^2$ est pair comme somme de deux pairs, sinon $n + n^2$ est pair comme somme de deux impairs".

Bref, traiter la liste vide à part dans de l'itératif est occasionnellement nécessaire, mais très souvent, c'est juste une lourdeur.

Remarque 2 : Pourquoi passer par P et ne pas faire ? :

```
Fonction longueur (liste L) : entier          // version 3 variante Bof
       cpt = 0
       tant que L n'est pas vide
                cpt++
                L = Suite(L)
      rendre cpt
```

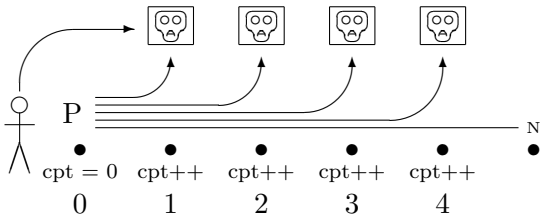
Parce que si vous faites cela, L est transformé et devient vide. Or si je vous confie L, ce n'est pas pour que vous le transformiez ou que vous le réduisiez à NULL. Si je vous confie les clefs de chez moi pour vous me fassiez un devis (fonction) ou des travaux (procédure), je ne vais pas être content si, pour faire ce que vous aviez à faire, vous avez chamboulé tout mon appartement. C'est pareil pour une fonctionnalité, on vous a vous confié des arguments, et, sauf pour les variables dont les spécifications disent qu'elles doivent changer, vous devez les rendre dans l'état où on vous les a données.

Oui mais, en réalité... On est sur un passage par valeur, donc le L de l'appelé est un pointeur copie du pointeur L de l'appelant. Donc je suis en train de faire un pointeur copie du pointeur copie. Oui, mais un jour vous ferez un passage par variable pour une autre fonctionnalité. Un jour, un collègue (ou vous-même !) reprendrez cette fonction et vous l'adapterez pour en faire autre chose, et le passage par valeur sera changé en un passage par variable (parce que il y a besoin pour cet autre chose, ou parce que votre collègue "aime" les passages par variable, allez savoir ce que les autres vont faire...). Si vous n'êtes pas passé par P, soit votre collègue ne s'en aperçoit pas et détruit l'argument par effet de bord sans s'en rendre compte. Soit il s'en aperçoit et il se trouve obligé de se replonger dans votre code et de l'adapter, et là il perd son temps et il vous hait.

Remarque 3 :
Puis-je utiliser `depile(P)` plutôt que `P = Suite(P)` ?

NON, car `Depile` fait un free et détruit (rend à la mémoire) les blocs qu'il traverse.

Si vous faite cela, à l'arrivée, tous les blocs de la liste ont été détruits, bref, vous avez modifié (par destruction) la liste qui vous a été confiée.



C'est comme si, pour les compter, vous arrachiez les pages d'un livre au lieu de les tourner.

Versions récursives terminales

Il suffit de dérécursifier la version itérative.

Une sous-PROCEDURE récursive va simuler les calculs faits par la boucle. Elle aura un argument `P` et un argument `cpt`.

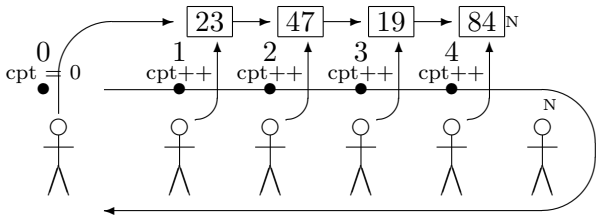
Le test `tant que P n'est pas vide` va devenir un `si P n'est pas vide` avec un appel récursif dans le `alors`.

`cpt` va être une variable globale que les appels vont se refiler via un mode de passage INOUT. L'instruction `cpt++` va rester et sera faite AVANT l'appel récursif: Il s'agit de MODIFIER une variable, on ne peut donc pas le faire (en programmation propre...) dans la liste des arguments de l'appel récursif.

`P` va être un argument lu. On pourrait faire `P = Suite(P)` avant l'appel récursif, puis appeler sur `P`, mais il s'agit de calculer une valeur, on peut donc plus simplement appeler récursivement sur `Suite(P)`

Une surfonction va effectuer toutes les opérations hors-boucle et faire l'interface.

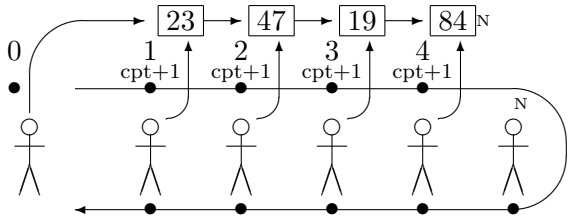
```
Fonction longueur (liste L) : entier          // version 4, recursive terminale
       cpt = 0                                // avec sous-procedure et cpt INOUT
       Bis(L,cpt)
       rendre cpt
Procedure Bis (liste P, INOUT int cpt)
       si P est non vide
       alors   cpt++
               Bis(Suite(P),cpt)
```



Variante :

On dédouble le `cpt` INOUT en deux morceeaux : un `cpt (cptIN)` pour la partie IN et un retour de fonction pour la partie OUT. On n'utilise plus une sous-procédure mais une sous-fonction. Pour la surfonction, il n'y a plus de variable `cpt`, mais une valeur envoyée, 0, et une valeur reçue, à transmettre. Le compteur n'etant plus une variable globale à modifier, mais une valeur à transmettre, on peut certes faire `cpt++` puis appeler avec `cpt` mais il est mieux d'appeler directement avec `cpt+1` (tout comme on appelle avec `Suite(P)`)

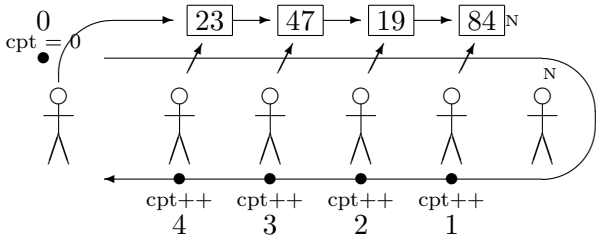
```
Fonction longueur (liste L) : entier          // version 5, recursive terminale
       rendre Bis(L,0)                        // avec sous-fonction et cpt IN
Fonction Bis (liste P, IN int cpt) : int      // et partie OUT en retour de fonction
       si P est vide
       alors rendre cpt
       sinon rendre Bis(Suite(P),cpt+1)
```



Autres versions récursives

Que se passe-t-il si l'on reprend la version 4 et que l'on déplace le `cpt++` après l'appel récursif ? Le compteur est initialisé au tout début par la sur-fonction, puis il ne bouge plus lors des ouvertures d'appels, et c'est au retour, à la fermeture des appels qu'il est incrémenté. Et ça marche, en se remettant à compter de droite à gauche, avec un initialisation faite très en avance.

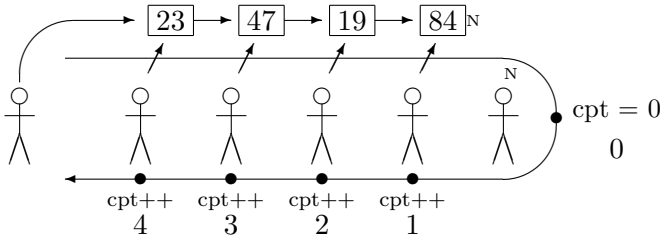
```
Fonction longueur (liste L) : entier          // version 6
       cpt = 0
       Bis(L,cpt)
       rendre cpt
Procedure Bis (liste P, INOUT int cpt)
       si P est non vide
       alors   Bis(Suite(P),cpt)
               cpt++
```



Finalement, on pourrait ne pas initialiser `cpt` autant à l'avance, on peut le faire initialiser par le dernier appel, celui avec `P == vide`. Du coup, `cpt` n'est plus transmis d'appelant vers appelé, mais uniquement d'appelé vers appelant, c.-à-d. qu'il devient une variable OUT.

En fait, on retouve la version récursive de base, sauf que la variable OUT implicite de retour de fonction devient une variable OUT explicite argument de procédure.

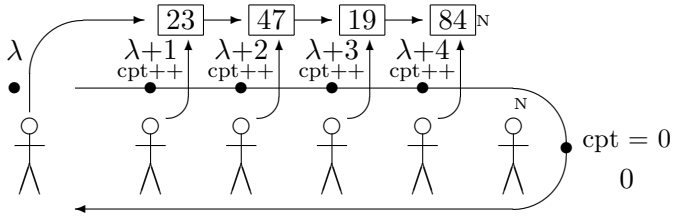
```
Fonction longueur (liste L) : entier          // version 7
       Bis(L,cpt)
       rendre cpt
Procedure Bis (liste P, OUT int cpt)
       si P est vide
       alors cpt = 0
       sinon   Bis(Suite(P),cpt)
               cpt++
```



Et si je prends la version 7 et que je redéplace le `cpt++` avant l'appel récursif ?

L'incrémentation se fait à l'aller, sur un compteur pas initialisé, qui vaut donc initialement une valeur farfelue λ, puis l'initialisation est faite, trop tard car après les incrémentations. Ce coup-ci, c'est une version bugée qui rend toujours 0.

```
Fonction longueur (liste L) : entier          // version 8  -- FAUSSE !!! --
       Bis(L,cpt)
       rendre cpt
Procedure Bis (liste P, OUT int cpt)
       si P est vide
       alors cpt = 0       FAUX !!!
       sinon   cpt++
               Bis(Suite(P),cpt)
```



Exemple 2 : Procédure VireDernier

La procédure `VireDernier` doit prendre une variable liste en argument et supprimer le dernier élément. Si la liste est vide, elle ne fait rien. Il faut utiliser un passage en INOUT puisque l'on modifie la liste, et appeler récursivement avec la variable `PointeurSuite`

```
Procedure VireDernier (inout liste L)
   si L est vide
   alors ne rien faire
   sinon   si Suite(L) est vide
           alors Depile(L)
           sinon VireDernier(PointeurSuite(L))
```

On peut alors remarquer que l'on ne relance jamais récursivement sur une liste vide. Le test `si L est vide` rendra toujours faux pour les appels récursifs. Il ne peut rendre vrai que pour le premier appel si l'utilisateur appelle sur une liste vide. On gagnerait donc du temps si on faisait ce test une fois en tête de liste, puis qu'on ne le fasse plus. La solution pour cela est de le faire dans une surfonction.

```
Procedure VireDernier (inout liste L)
      si L est vide
      alors ne rien faire
      sinon VD(L)
Procedure VD (inout liste L)  //  premisse : L est non vide
           si Suite(L) est vide
           alors Depile(L)
           sinon VD(PointeurSuite(L))
```

On vérifie que le premisse est bien vérifié à chaque appel. Pour l'appel dans la surfonction, c'est bien le cas puisque l'on appelle sur `L` dans le `sinon` du test `si L est vide`. Pour l'appel récursif, c'est bien le cas puisque l'on appelle sur `PointeurSuite(L)` dans le `sinon` du test `si Suite(L) est vide`.

Ce genre de vérification est faite en GL, Génie Logiciel. En gros, le GL cherche à prouver que les programmes font ce que l'on veut et ne plantent pas. C'est sympathique pour les programmes de jeux vidéos. C'est crucial pour les programmes de banque, d'armement, de centrale nucléaire, d'aviation...

Question : Pouvez-vous remplacer `VD(PointeurSuite(L))` par :
{ `x = Premier(L) ; Depile(L) ; VD(L) ; Empile(x,L) ;` } ?

NON, ce serait d'abord très lourd, vous désalloueriez pour réallouer plus tard la même chose. C'est comme si pour lire un livre que vous devez me rendre, vous répétiez "faire une photocopie puis arracher et jeter la page", et qu'une fois la lecture finie, vous me rendiez les photocopies au lieu du livre.

Outre la lourdeur, cela peut poser des problèmes, comme par exemple :

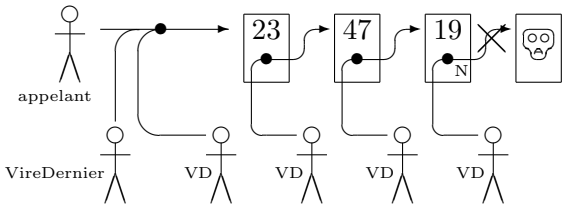
Je vous fait ajouter un champ au bloc. Il va alors falloir que vous repassiez sur tout votre code pour que les "photocopies" intègrent ce nouveau champ. Cela va être très pénible et vous risquez d'oublier des correctifs à faire, ce qui va vous faire buger le programme.

Si j'ai un pointeur que vous ne voyez pas sur un des blocs, vous ne pourrez pas le corriger pour qu'il pointe vers la nouvelle photocopie. Il continuera après la manip à pointer vers le vieux bloc que vous avez mis à la poubelle.

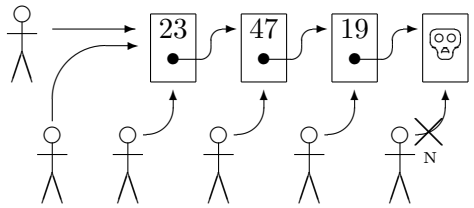
En mettant en avant les pointeurs, cela donne :

Avec un passage par variable, on appelle avec `VireDernier(L)` et le code est

```
Procedure VireDernier ( VAR liste L)
      si L == NULL
      alors ne rien faire
      sinon VD(L)
Procedure VD ( VAR liste L)
        si (*L).suite == NULL
        alors Depile(L)
        sinon VD( (*L).suite )
```

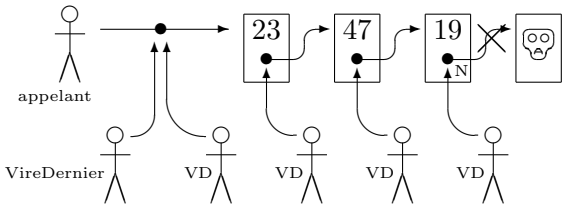


En apparence, l'appel avec `PointeurSuite(L)` est du même type que l'appel avec `Suite(L)` de la version 1bis de longueur. Mais attention, pour `VD`, il est impératif que ce soit la variable `(*L).suite` qui soit utilisée et pas une copie de ce pointeur, alors que pour longueur, seule la valeur de `(*L).suite` compte et on pourrait utiliser une copie de pointeur. On pourrait remplacer `rendre 1+longueur((*L).suite);` de la version 1bis de longueur par {`Pcopie = (*L).suite); rendre 1+longueur(Pcopie);`}. Par contre, la version ci-dessus deviendra fausse si vous remplacez `VD((*L).suite);` par {`Pcopie = (*L).suite); VD(Pcopie);`}. Si vous faites cela, ou si vous le faites implicitement en appelant `Suite(L)` c.-à-d. si vous faites des passages par valeur, c'est le pointeur copie qui est mis à NULL au lieu du champ `suite` de l'avant-dernier bloc.



Avec un passage par adresse, il faut utiliser des pointeurs vers les pointeurs listes, on appelle avec `VireDernier(&L)`, c.-à-d. avec un pointeur sur `L`, et le code est

```
Procedure VireDernier (liste* P)
      si *P == NULL
      alors ne rien faire
      sinon VD(P)
Procedure VD (liste* P)
        si (**P).suite == NULL
        alors Depile(P)
        sinon VD( &((**P).suite) )
```



Vous aurez observé que l'expression pour `PointeurSuite` est devenue explicitement plus complexe :
`P` est le pointeur qui pointe vers le pointeur liste qui pointe vers un bloc qui contient le champ suite.
`*P` est le pointeur liste qui pointe vers un bloc qui contient le champ suite.
`**P` est le bloc qui contient le champ suite.
`(**P).suite` est le champ suite du bloc. Ce champ a une adresse, c'est donc une variable
`&((**P).suite)` a pour valeur l'adresse de ce champ, c'est donc un pointeur vers le champ suite du bloc.

Ne pas se fier à la simplicité apparente de l'appel `VD(P)`. C'est `*P` que l'on veut envoyer à `VD`, mais comme `VD` prend les arguments par adresse, il faut donc appeler `VD(&(*P))`. Sauf qu'il se trouve que `&(*P)` et `P` ont la même valeur, et du coup, on peut simplifier `VD(&(*P))` en `VD(P)`. Mais moralement, ce `P` est un `&(*P)`. C'est exactement la même chose pour le `P` de `Depile(P)`

Si on veut le faire en itératif, avec un passage par variable, cela va donner le code ci-dessous à gauche. Avec le passage par adresse, seul le début et la fin seront changés `Procedure VireDernier (liste* P):` `si *P est non NULL alors tant que ...  Depile(P) // alias depile(&*P).`

En pseudo-code, il faudra faire intervenir * et & et donner le code de droite :

```
Procedure VireDernier ( VAR liste L)       | Procedure VireDernier ( INOUT liste L)
  si L est non NULL                        |   si L est non []
  alors P = & L                            |   alors P = & L
        tant que (**P).suite est non NULL  |         tant que Suite(*P) est non []
                 P = &((**P).suite)        |                 P = &(PointeurSuite(*P))
        Depile(*P)                         |         Depile(*P)
```

Diviser pour régner

Diviser pour régner, c'est sous-traiter le gros du travail. Les sous-traitants pourront eux-mêmes sous-traiter. Il y aura donc des sous-traitants de sous-traitants de sous-traitants, etc. Il faudra bien sûr que chaque sous-traitant fasse un petit quelque chose pour que le travail se fasse.

Diviser pour régner 1, le récursif : si le travail est vraiment très simple, je le fais. Sinon je fais appel à un sous-traitant clône de moi-même qui fera la majeure partie du travail et je complèterai son travail. On se pose la question : Et si je fais un appel récursif sur $n-1$, $n/2$, `suite(L)`, est-ce que j'obtiens quelque chose de pertinent et comment je complète ?

Exemple, longueur : Si je demande longueur(suite(l)), est-ce que je j'obtiens une information pertinente et comment dois-je compléter ? Oui, c'est pertinent, et je complèterai en ajoutant 1, d'où le premier code récursif pour longueur.

Diviser pour régner 2 : on peut faire appel à des sous-traitants qui ne sont pas des clônes.

Cela peut être découper le problème en deux morceaux que l'on traitera de façon indépendante.

Cela peut être faire appel à deux sous-traitants, un clône et un non clône.

On se pose la question : Et si je fais un appel récursif sur $n-1$, $n/2$, `suite(L)`, est-ce que j'obtiens quelque chose de pertinent et comment je complète ? Il apparait alors que "compléter" est un travail non élémentaire, différent de mon propre travail. Je vais donc faire appel à un sous-traitant d'un type nouveau, et ce sous-traitant pourra lui-même fonctionner en mode "diviser pour régner".

Exemple 3 : Fonction TousDifférents : Diviser pour régner, Complexité

La fonction TousDiff prend en argument une liste et rend vrai ssi tous les éléments sont différents.

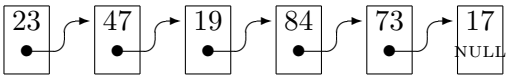
Question : Un appel TousDiff(suite(L)) est-il pertinent et comment compléter ? Je constate que oui et que je dois compléter en regardant si le premier élément est dans la suite de L. Donc je vais faire un appel récursif ET un appel à une nouvelle fonction `EstDans(x,L)` qui s'écrira facilement en récursif.

```
Fonction TousDiff (liste L) : bool
   si L est vide
   alors rendre vrai
   sinon  si EstDans(x,suite(L))
          alors rendre faux
          sinon rendre TousDiff(suite(L))
```

```
fonction EstDans(x,L) : bool
  si L est vide
  alors rendre faux
  sinon si x == premier(L)
        alors rendre faux
        sinon rendre EstDans(x, suite(L))
```

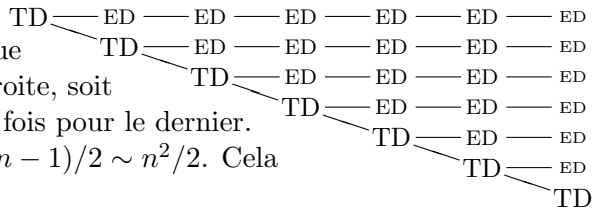
Complexité en nombre de tests "x == premier(L)" :

On peut commencer par faire l'arbre des appels :



L'arbre des appels fait un demi carré,

on "voit" que la complexité est quadratique, $\sim n^2/2$:



On peut le calculer de différentes façons :

Méthode 1 : C'est x qui paye pour chaque test. Chaque élément va payer autant de fois qu'il y a d'élément à sa droite, soit $n-1$ fois pour le premier, $n-2$ fois pour le second, ..., 0 fois pour le dernier. Le coût total est donc $(n-1)+(n-2)+...+1+0 = n(n-1)/2 \sim n^2/2$. Cela revient à compter les appels de l'arbre ligne par ligne.

Méthode 2 : C'est $premier(L)$ qui paye pour chaque test. Chaque élément va payer autant de fois qu'il y a d'élément à sa gauche, soit 0 fois pour le premier, 1 fois pour le second, ..., $n-1$ fois pour le dernier. Le coût total est donc $0+1+...+(n-2)+(n-1) = n(n-1)/2 \sim n^2/2$. Cela revient à compter les appels de l'arbre colonne par colonne.

Méthode 3 : Il y a une comparaison par ensemble de deux éléments, le nombre de comparaisons est donc $\begin{pmatrix} n \\ 2 \end{pmatrix} = n(n-1)/2 \sim n^2/2$.

Méthode 4 : on pose les équations sur TD_n et ED_n le combre de comparaisons faites lors d'un appel à TousDiff, resp. à EstDans. On obtient, au vu du code :

$TD_0 = 0$, $TD_n = TD_{n-1} + ED_{n-1}$

$ED_0 = 0$, $ED_n = 1 + ED_{n-1}$,

que l'on résoud mathématiquement en $ED_n = n$ puis $TD_n = n(n-1)/2$

Exemple 4 : Fonction Permutations : Diviser pour régner

La fonction Permutations prend en argument un entier n et rend la liste des permutations de [1..n]. Par exemple, Permutations(3) rendra [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]. (Les permutations peuvent être dans un autre ordre, [[2,3,1], [2,1,3], [3,1,2], [1,2,3], [1,3,2], [3,2,1]] convient aussi)

Le type de sortie est donc "liste de listes d'entiers"

Il y a plusieurs méthodes pour rendre ce résultat, nous allons procéder ici par "Diviser Pour Régner".

Pour faire Permutations(n), on peut appeler récursivement Permutations(n-1). Le résultat est-il pertinent et comment compléter ? On voit que pour compléter, il faudrait ajouter n dans toutes les listes à toutes les positions. On prévoit donc d'écrire ATPTL :

```
fonction ATLTP (int x, liste de liste d'entiers L) : liste de liste d'entiers
    // Ajoute Toute Liste Toute Position
   // ATLTP( 4, [[5,7,9],[2],[3,8],[]]  )
  // -> [[4,5,7,9],[5,4,7,9],[5,7,4,9],[5,7,9,4],[4,2],[2,4],[4,3,8],[3,4,8],[3,8,4],[4]]
```

Permutations s'écrit alors en récursif. Le cas de base, $n = 0$: il y a une permutation de la liste vide qui est la liste vide. Puisqu'il faut rendre la liste des permutations, on rend la liste singleton contenant la liste vide. En dehors du cas de base, on appele récursivement et on complète, grâce à ATLTP, en ajoutant n de toutes les manières possibles. Notez qu'au niveau de Permutations, on n'a pas besoin de savoir comment va fonctionner ATLTP.

```
fonction Permutations (int n) : liste de liste d'entiers
    si n == 0
    alors rendre [[]]
    sinon rendre ATLTP( n, Permutations (n-1) )
```

On écrit maintenant ATLTP. Notez que pour cela, je dois savoir ce que je dois faire, mais aucunement pourquoi. Si une personne fait Permutations et l'autre ATLTP, elles doivent se mettre clairement d'accord sur la "spécification" de Permutations sans se coucier du comment et ATLTP sans se soucier du pourquoi.

On se propose de faire ATLTP en diviser pour régner. On peut appeler récursivement sur suite(L), soit sur l'exemple ATLTP(4, [[2],[3,8],[]]), ce qui donnera [[4,2],[2,4],[4,3,8],[3,4,8],[3,8,4],[4]]. Est-ce pertinent et comment compléter ? Je vois qu'il me manque [[4,5,7,9],[5,4,7,9],[5,7,4,9],[5,7,9,4]] qui s'obtient en ajoutant 4 à toute position de [5,7,9] qui est ma première liste. Je ferai cela en appelant ATP(x,premier(L)), ATP(4,[5,7,9]) sur l'exemple, et je prévois d'écrire ATP :

```
fonction ATP (int x, liste d'entiers L) : liste de liste d'entiers
             // Ajoute Toute Position
            // ATP( 4, [5,7,9] ) ->[[4,5,7,9],[5,4,7,9],[5,7,4,9],[5,7,9,4]]
```

Puis je concaténerai [[4,5,7,9],[5,4,7,9],[5,7,4,9],[5,7,9,4]] et [[4,2],[2,4],[4,3,8],[3,4,8],[3,8,4],[4]]. Je prévois donc d'écrire

```
fonction Concat (liste de trucs L1, L2) : liste de trucs
            // Concat( [a,b,c], [x,y,z] ) -> [a,b,c,x,y,z]
```

J'écris à présent ATLTP :

```
fonction ATLTP (int x, liste de liste d'entiers L) : liste de liste d'entiers
    // Ajoute Toute Liste Toute Position
   // ATLTP( 4, [[5,7,9],[2],[3,8],[]]  )
  // -> [[4,5,7,9],[5,4,7,9],[5,7,4,9],[5,7,9,4],[4,2],[2,4],[4,3,8],[3,4,8],[3,8,4],[4]]
si L est vide
alors rendre []
sinon rendre Concat ( ATP( x, premier(L) ) ,
                      ATLTP( x, suite(L) )   )
```

De nouveau, ATLTP ne se soucie pas de comment seront écrites ATP et Concat. Et nous allons écrire Concat et ATP sans nous soucier de pourquoi nous les écrivons. L'important est que la communication soit claire entre l'utilisateur qui doit donner une spécification correcte et le fournisseur qui doit bien lire la spécification.

Écrivons Concat. On se propose de le faire en diviser pour régner. Il y a deux listes en arguments, on peut donc faire 3 appels récursifs distincts : (1) Concat(suite(L1),suite(L2)), (2) Concat(L1,suite(L2)), (3) Concat(suite(L1),L2). Sur l'exemple (Rappel L1=[a,b,c], L2=[x,y,z]), (1) donnera [b,c,y,z], il faudra ajouter un a devant -facile-, et un x : ouille, comment je sais où le mettre ?, ça risque d'être compliqué, il faut un argument retour supplémentaire pour détermianer la position d'insertion. Humm bof. Voyons (2) Cela donnera [a,b,c,y,z], il faut ajouter un x en cours de liste, même difficulté qu'avec (1). Regardons (3) : j'obtiens [b,c,x,y,z] et je n'aurai qu'à ajouter a devant, ce qui est facile. C'est la bonne option. Puisque le suite de l'appel récursif est sur L1, le cas de base est "Si L1 est vide". On est parti pour le code :

```
fonction Concat (liste de trucs L1, L2) : liste de trucs
             // Concat ( [a,b,c], [x,y,z] ) -> [a,b,c,x,y,z]
Si L1 est vide
alors rendre L2
sinon rendre ajoute( premier(L1), Concat( suite(L1), L2 ) )
```

Écrivons maintenant ATP. On se propose de le faire en diviser pour régner. On peut appeler récursivment, soit sur l'exemple ATP(4, [5,7,9]) (qui doit donner [[4,5,7,9],[5,4,7,9],[5,7,4,9],[5,7,9,4]]), on peut appeler ATP(4, [7,9]), ce qui donnera [[4,7,9],[7,4,9],[7,9,4]]. Est-ce pertinent, comment compléter ? On voit d'abord qu'il faudrait remettre le 5 en tête de toutes les listes, nous allons faire cela en appelant AETTL(premier(L), ATP(x,suite(L))), et l'on prévoit d'écrire AETTL

```
fonction AETTL (int x, liste de liste d'entiers L) : liste de liste d'entiers
    // Ajoute En Tete Toute Liste
   // AETTLTP( 6, [[3,7,9],[2],[4,8],[]]  )  -> [[6,3,7,9],[6,2],[6,4,8],[6]]
```

Puis il faudrait y ajouter devant (par un simple ajoute, d'une liste dans une liste de liste) la liste [4,5,7,9] qui n'est autre que ajoute(x,L). Pour le cas de base, si L est vide et que j'ajoute x, j'obtiens [x], et puisque je dois rendre la liste des listes possibilités, je rends [[x]].

D'où le code :

```
fonction ATP (int x, liste d'entiers L) : liste de liste d'entiers
              // Ajoute Toute Position
             // ATP( 4, [5,7,9] ) -> [[4,5,7,9],[5,4,7,9],[5,7,4,9],[5,7,9,4]]
Si L est vide
alors rendre [[x]]
sinon rendre ajoute ( ajoute (x,L),
                      AETTL( premier(L), ATP(x, suite(L))))
```

Une fois de plus, ATP a été écrite sans se soucier de comment AETTL sera écrite, et AETTL sera écrite sans se soucier de pourquoi on l'écrit.

Reste à coder AETTL. En diviser pour régner, on appelle récursivement, ce qui, sur l'exemple AETLTP(6, [[2],[4,8],[]]), donnera [[6,2],[6,4,8],[6]] , il suffira d'y ajouter [6,3,7,9] qui n'est autre que l'ajout de x dans la première des listes de L, d'où le code :

```
fonction AETTL (int x, liste de liste d'entiers L) : liste de liste d'entiers
    // Ajoute En Tete Toute Liste
   // AETTLTP( 6, [[3,7,9],[2],[4,8],[]]  )  -> [[6,3,7,9],[6,2],[6,4,8],[6]]
Si L est vide
alors rendre vide
sinon rendre ajoute ( ajoute( x, premier(L) ) ,
                      AETTL(x,suite(L))        )
```

Et c'est gagné...