1 Equivalents

Correction :
Rappels: *Les définitions ci-dessous ne sont valables que pour des fonctions f et g qui sont positives et ne s'annulent pas au voisinage de l'infini (Pour les fonctions non positives ou s'annulant au voisinage de l'infini, voir un cours de math).*

$f \sim g$, "f est équivalente à g" ssi $\frac{f}{g}$ tend vers 1 au voisinage de l'infini. L'idée est que f est quasiment égale à g.

$f = O(g)$ ssi $\frac{f}{g}$ est majorée au voisinage de l'infini $\Leftrightarrow \exists M, f(d) \leq M\, g(d)$ pour toute donnée d de taille suffisamment grande. L'idée est que f peut se majorer par du g à une constante multiplicative près (la majoration peut être pertinente $3n^2 + 1 = O(n^2)$ ou sur-évaluée $3n^2 + 1 = O(n^3)$)

$f = \Omega(g)$ ssi $\frac{f}{g}$ est minorée par un valeur strictement positive au voisinage de l'infini $\Leftrightarrow \exists m > 0, f(d) \geq m\, g(d)$ pour toute donnée d de taille suffisamment grande. L'idée est que f peut se MINORER par du g à une constante multiplicative près (de façon pertinente $3n^2 + 1 = \Omega(n^2)$ ou sous-évaluée $3n^2 + 1 = \Omega(n)$)

$f = \theta(g)$, "f est du même ordre de grandeur que g" ssi $\Leftrightarrow (f = O(g)$ and $g = O(f)) \Leftrightarrow \frac{f}{g}$ est minorée par une valeur strictement positive et est majorée au voisinage de l'infini $\Leftrightarrow \exists m > 0, M, m\, g(d) \leq f(d) \leq M\, g(d)$ pour toute donnée d de taille suffisamment grande. On rappelle que θ est une relation d'équivalence.

$f = o(g)$, "f est négligeable devant g", ssi $\frac{f}{g}$ tend vers 0 au voisinage de l'infini. L'idée est que f est bien plus petit que g.

- Donnez un équivalent simple puis un ordre de grandeur simple pour $f(n) = 4n^2 + 5n + 7$

Correction :
Moralité à faire passer : l'équivalent conserve la constante multiplicative, alors que l'ordre de grandeur ne la prend pas en compte. Un équivalent est donc plus précis.

L'équivalent le plus simple de f : $f \sim 4n^2$

L'ordre de grandeur le plus simple de f : $f = \theta(n^2)$.

Notez que $f \sim n^2$, c'est faux

Par contre, $f = \theta(4\, n^2)$, c'est vrai mais le 4 n'y joue aucun rôle. Il est tout aussi exact de dire que $f = \theta(23\, n^2)$.

On retiendra qu'au partiel ou à l'examen, quand une complexité est demandée, s'il est écrit "Donnez un équivalent", la constante multiplicative est demandée. S'il est écrit "Donnez un un ordre de grandeur", la constante multiplicative n'est pas demandée.

- Quelle est la complexité du code suivant : `if b then print(bonjour) else faire 100 fois print(bonsoir)`

Correction :
Si A est le temps de faire le test, B celui de dire bonjour et C celui de dire bonsoir, on a qqch comme $A + B \leq$ temps $\leq A + 100C$. Or $A + B$ et $A + 100C$ sont des constantes. On obtient donc temps $= \theta(1)$ i.e. $(A + B).1 \leq$ temps $\leq (A + 100C).1$, c'est donc "constant" au sens des ordres de grandeur et donc de la complexité. Cet exo sert juste à dire que constant ne veut pas dire constant au sens mathématique, mais être minoré et majoré (généralisation du constant mathématique). Par exemple, $3 + (-1)^n$ est constant, si la définiton de constant est être $\theta(1)$

- (facultatif) Montrez que $\frac{2^n}{n+1} \leq C_n^{n/2} \leq 2^n$. Donnez un équivalent simple de $C_n^{n/2}$ puis de $C_n^{n/2+\lambda\sqrt{n}}$. Rappel, $n! \sim \sqrt{2\pi n}(n/e)^n$,

---

Correction :
Note : détail pas très intéressant mais à donner si l'on veut être rigoureux : $n/2$ désigne ici au besoin la partie entière $\lfloor n/2 \rfloor$ de $n/2$. Notez que $\lfloor n/2 \rfloor \sim n/2$, donc on pourra ignorer ce détail dans les calculs.

On rappelle que $\sum_0^N C_N^p = 2^N$. Pour la majoration, on utilise le fait que notre terme y figure et que c'est une somme de termes positifs. Pour la minoration on utilise le fait qu'il y a $n + 1$ termes dans la somme et que $C_n^{\lfloor n/2 \rfloor} = C_n^{\lceil n/2 \rceil}$ est le plus grand des C_n^p (pour le justifier, observer que $C_n^{p+1}/C_n^p = (n!/(p+1)!(n-p-$

$1)!)/(n!/p!(n-p)!) = (n-p)/(p+1)$ *qui va être plus grand ou plus petit que 1 selon que p est plus grand ou plus petit que n/2 (détails n/2 vs $\lfloor n/2 \rfloor$ vs $\lceil n/2 \rceil$ fastidieux) doù la croissance jusqu'au milieu puis la décroissance)*

$$C_n^{n/2} = \frac{(n)!}{(n/2)!^2} \sim \frac{\sqrt{2\pi n}(n/e)^n}{[\sqrt{2\pi(n/2)}\ ((n/2)/e)^{n/2}]^2} = \frac{\sqrt{2\pi n}}{(\sqrt{\pi n})^2}\ \frac{n^n/e^n}{n^n/(2^n e^n)} = \sqrt{2/\pi}\ \frac{2^n}{\sqrt{n}}$$

bref c'est de taille exponentielle.

Note: on pouvait voir venir le coup du $1/\sqrt{n}$:

Tracer l'histogramme de $C_n^p/2^n$. C'est à peu de chose près une cloche (c'est une gaussienne) centrée en $p = n/2$. La surface est 1. La hauteur en p, c'est la proba que n pile ou face donnent p fois pile. Or si je fais n pile ou face, j'aurai n/2 pile en moyenne, et j'en aurai en pratique $n/2 + d$ avec d de l'ordre de l'écart type, soit du racine de n (rappel : la variance est la moyenne du carré de la différence à la moyenne, est aussi la différence entre la moyenne du carré et le carré de la moyenne. $V(X) = E((X - E(X))^2) = E(X^2)) + E(E(X)^2) - 2E(X.E(X)) = E(X^2) - E(X)^2$, et la variance de la somme de deux évènements indépendants X et Y (donc tels que $E(XY) = E(X)E(Y)$) est la somme des variances $V(X+Y) = E((X+Y)^2) - E(X+Y)^2 = E(X^2) + E(Y^2) + 2E(XY) - (E(X)^2 + E(Y)^2 - 2E(X)E(Y)) = V(X) + V(Y)$. La variance de la somme de n termes identiques indépendants est donc n fois la variance pour un terme. L'écart type est (définition) la racine de la variance. L'écart type de la somme de n termes indépendants identiques est donc $\sqrt{n}$ fois l'écart type pour un seul terme.

Bref, la largeur de la cloche est en $\sqrt{n}$ et puisque la surface fait 1, sa hauteur est de l'ordre de $1/\sqrt{n}$
On peut regarder ce que vaut C_n^p pour un p proche de n/2 à $\sqrt{n}$ près :

$$C_n^{n/2+\lambda\sqrt{n}} = \frac{(n)!}{(n/2+\lambda\sqrt{n})!)(n/2-\lambda\sqrt{n})!}$$

$$\sim \frac{\sqrt{2\pi n}(n/e)^n}{[\sqrt{2\pi(n/2+\lambda\sqrt{n})}\ ((n/2+\lambda\sqrt{n})/e)^{n/2+\lambda\sqrt{n}}] * [\sqrt{2\pi(n/2-\lambda\sqrt{n})}\ ((n/2-\lambda\sqrt{n})/e)^{n/2-\lambda\sqrt{n}}]}$$

$$= \frac{\sqrt{2\pi n}}{\sqrt{2\pi(n/2+\lambda\sqrt{n})} * \sqrt{2\pi(n/2-\lambda\sqrt{n})}} * \frac{(n/e)^n}{[((n/2+\lambda\sqrt{n})/e)^{n/2+\lambda\sqrt{n}}] * [((n/2-\lambda\sqrt{n})/e)^{n/2-\lambda\sqrt{n}}]}$$

$$\sim \sqrt{2/\pi n} * \frac{(n/e)^n}{(n/2e)^n(1+\frac{2\lambda}{\sqrt{n}})^{n/2+\lambda\sqrt{n}}(1-\frac{2\lambda}{\sqrt{n}})^{n/2-\lambda\sqrt{n}}}$$

$$= \sqrt{2/\pi}\ \frac{2^n}{\sqrt{n}} * \frac{1}{e^{(n/2+\lambda\sqrt{n})ln(1+\frac{2\lambda}{\sqrt{n}})} e^{(n/2-\lambda\sqrt{n})ln(1-\frac{2\lambda}{\sqrt{n}})}}$$

$$= \sqrt{2/\pi}\ \frac{2^n}{\sqrt{n}} * \frac{1}{e^{(n/2+\lambda\sqrt{n})(\frac{2\lambda}{\sqrt{n}}-(\frac{2\lambda}{\sqrt{n}})^2/2+o(\frac{1}{\sqrt{n}^3}))} e^{(n/2-\lambda\sqrt{n})(\frac{-2\lambda}{\sqrt{n}}-(\frac{2\lambda}{\sqrt{n}})^2/2+o(\frac{1}{\sqrt{n}^3}))}}$$

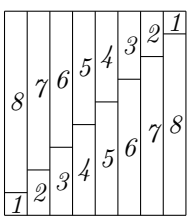
$$= \sqrt{2/\pi}\ \frac{2^n}{\sqrt{n}} * \frac{1}{(e^{\lambda\sqrt{n}+\lambda^2+o(1))} * (e^{-\lambda\sqrt{n}+\lambda^2+o(1)})}$$

$$\sim \sqrt{2/\pi}\ \frac{2^n}{\sqrt{n}} * e^{-2\lambda^2}$$

On reconnaît le terme en n/2 fois $e^{-2\lambda^2}$, on reconnaît notre gaussienne.

- Combien vaut $\sum_{i=1}^n i$? Donner un équivalent de $\sum_{i=1}^n i^2$, de $\sum_{i=1}^n i^k$, de $\sum_{i=1}^n 1/i$, de $\sum_{i=1}^n \ln i$.

Correction :



$\sum_{i=1}^n i = \frac{n(n+1)}{2}$. *On peut le voir comme la moyenne des nombres fois le nombre de termes pour une suite à progression affine, ou on fait un dessin : faire un histogramme de la fonction identité. Faire une copie de l'histogramme, lui faire faire un demi tour, et l'emboîter au dessus du premier de façon a avoir un rectangle. On compte la surface, c'est 2 fois la somme, et c'est longueur n fois hauteur $n+1$.*

que valent $\sum^N i^2$, $\sum^N i^3$. *Réponses : $n(n+1)(2n+1)/6$ et $(n(n+1)/2)^2$, d'où des équivalents en $n^3/3$ et $n^4/4$ (Démo à faire par un dessin de $\sum^N i^3 = (\sum^N i)^2$)*

quels équivalents pour $\sum^N i^{23}$, $\sum^N \sqrt{i}$? *conjecture : $N^{24}/24$ et $N^{3/2}/(3/2) = 2/3N\sqrt{N}$. On justifie en approximant par l'intégrale : $\sum^N i^k \sim \int^N i^k \sim N^{k+1}/(k+1)$*

Puis par approximation par intégrale, on aura aussi $\sum^N i^{-1} \sim \int^N i^{-1} \sim ln\, N$

$\sum^N ln\, i \sim \int^N ln\, i = [x\, ln\, x]^N - \int^N 1 \sim N\, ln\, N - N \sim N\, ln\, N$. Pour ce dernier, si on ne veut pas faire l'intégrale par partie, on peut dire que la fonction log est très plate, d'ou une approximation de l'intégrale par un rectangle de hauteur ln n et largeur n.

Remarque : pour justifier l'approximation, il faut majorer ou minorer l'histogramme par l'intégrale suivant que la fonction est croissante ou décroissante, puis décaler de 1 pour minorer ou majorer. Exemple, x^2 est croissante donc $N^2/2 = \int_0^N i^2 \leq \sum_1^N i^2 \leq \int_1^{N+1} i^2 = (N+1)^2/2 - 1$.

l'approximation est valide tant que la fonction ne croit pas trop vite. Avec un c^n, l'ordre de grandeur est bon mais pas l'équivalent, i.e. que la constante multiplicative est fausse. Au delà (c^{n^2}), ça ne va plus. la partie de l'histogramme qui dépasse de la courbe devient trop gros par rapport au reste.

---

2 Taille des entiers

Soit n est un entier strictement positif s'écrivant avec k chiffres en binaire. Encadrez n en fonction de k. Donnez (un équivalent de) k en fonction de n. Quelle est la taille de n ?

Quel rapport y-a-il entre le nombre de chiffres en base 2 de n et celui de $n\, div\, 2$? Combien d'itérations effectue le programme suivant : `i = n ; tant que i est non nul faire i = i div 2 ;` ?

Correction :

• *Si n s'écrit avec 7 chiffres en base 10, il vaut entre un million $= 10^6 = 10^{7-1}$ inclus et dix millions $= 10^7$ exclus. Plus généralement, si n s'écrit avec k chiffres en base 10, il vaut entre 10^{k-1} inclus et 10^k exclus. C'est le même principe en base 2: Si n s'écrit avec k chiffres en base 2, alors $2^{k-1} \leq n < 2^k$ d'où $k-1 \leq ln_2 n < k$ donc $k \leq 1 + ln_2 n < k+1$ donc $k = \lfloor ln_2 n \rfloor + 1$ (la notation $\lfloor\ \rfloor$ veut dire partie entière inférieure) La partie entière et le +1 sont sans importance, ce qu'il faut retenir, c'est $k \sim ln_2 n$. On remarquera au passage que $n = \theta(2^k)$ puisque $(1/2).2^k \leq n \leq 1.2^k$*

• *$chiffres(n) = chiffres(n\, div 2) + 1$ (détail technique peu intéressant : sauf pour le cas $n = 1$ où la formule n'est vraie que si l'on considère que 0 s'écrit avec 0 chiffres, et sauf pour $n = 0$ où la formule n'est pas vraie)*

• *environ $ln_2 n$ puisqu'à chaque itération, n perd l'un de ses chiffres et qu'il en a environ ln2 n au départ. Le code fait quelque chose comme* `i = environ ln2(n) bits ; tant que i a encore des bits faire i perd un bit ;` *On peut le voir aussi en disant qu'après k divisions par 2, notre nombre vaut $n/2^k$ et il vaut 1 pour $k = ln2(n)$.*

On retiendra que le nombre de fois qu'il faut diviser n par 2 (division entière) pour tomber sur 0 ou 1, c'est environ ln2(n). Cela sert sans arrêt pour évaluer des complexités. Exemple, la dichotomie : la taille du bout de tableau dans lequel on cherche est divisée par 2 à chaque itération, on part de n, on termine à 0 ou 1, il y a donc environ ln2(n) itérations.

---

3 Complexités de l'arithmétique

Quelle est la complexité des algos d'addition et de multiplication d'entiers vus au primaire ? Quelle est la complexité des algos utilisant les "bâtons" ? Quels sont les meilleurs algos ?

Correction :

Rappel : la taille d'un entier n, c'est son nombre de bits, soit $ln_2(n)$ environ. Si on écrit n dans une base b (exemple $b = 10$), il s'écrit avec environ $ln_b(n)$ chiffres, ce qui ne changera pas l'ordre de grandeur des complexités puisque deux log a bases différentes sont proportionnels dont θ l'un de l'autre ($ln_b(x) = ln\, x / ln\, b = (ln\, a / ln\, b) * (ln\, x / ln\, a) = (ln\, a / ln\, b) * ln_a(x)$)

On travaille sur deux entiers n et m à p et q. Quelle est la taille du problème ? On peut donner deux réponses : $p+q$ et $max(p,q)$. Si on est en base 2, $p+q$ est bien une taille car c'est le nombre de bits pour coder les deux nombres. Rappel, si f est une taille, alors g aussi à condition que $g = \theta(f)$. Or $1 * max(p,q) \leq p+q \leq 2 * max(p,q)$, donc max est θ de la somme, donc max est aussi une taille.

On rappelle donc que $p \sim ln2n$ et $q \sim ln2m$ et $n = \theta(2^p)$ et $m = \theta(2^q)$.

Le traitement des unités prend un temps constant : faire la somme puis une retenue éventuelle. Itou pour les deuzaines, faire une somme plus une retenue éventuelle, puis mettre une retenue éventuelle. Itou pour les quatraines. Pour les colonnes, il faudra au mieux recopier un nombre, au pire une somme avec plein de retenues. Il y a un temps min et un temps max par colonne, donc $\theta(1)$ par colonne, donc $\theta(max(p,q))$ au total. L'algo est linéaire, en $\theta(taille)$

Pour multiplier deux entiers à p et q chiffres, il faut faire q fois une multiplication d'un chiffre par un nombre à p chiffres, ce qui coûtera $p * q$ multiplications, et écrira q nombres à environ p chiffres qu'il faudra additionner, soit

environ pq chiffres à additionner pour un coût environ pq. L'algo est donc O(quadratique) (O d'une part parce que q peut être bien plus petit, exemple $q = 1$ et aussi parce qu'il se peut que parmi les q chiffres, il y ait beaucoup de 0, auquel cas on vient de surestimer la complexité). la complexité au pire est en θ(quadratique), cf cas de deux nombres à p chiffres avec peu de 0.

Algo des bâtons : pour additionner n et m à p et q chiffres, je trace n bâtons, puis m bâtons, puis je compte le nombre de bâtons. La complexité est donc en $n + m$, mais il faut l'exprimer en fonction des tailles soit p et q. Or $n + m = \theta(2^p + 2^q)$ et donc l'algo est exponentiel.

itou pour la multiplication si je procède avec des bâtons.

Comme n et n^2 sont o de n'importe quelle exponentielle, j'en déduis que les premiers algos sont meilleurs.

Notons que l'algo pour l'addition est optimal car il faut un temps au moins linéaire pour lire les chiffres. Par contre, l'argument ne tient pas pour montrer que l'algo de multiplication est optimal. Et de fait, il ne l'est pas. On verra un algo asymptotiquement meilleur à la fin du cours.

(facultatif) Déterminer les plus petits entiers m_k et n_k dont la recherche du PGCD nécessite k itérations par l'algorithme d'Euclide. En déduire la complexité en nombre d'itérations de cet algorithme.

Correction :

Une itération de l'algo d'Euclide passe de (n,m) à (m,r) avec $n = qm+r$ la division euclidienne. Pour que les nombres ne diminuent pas trop vite, il faudrait que q reste toujours petit, par exemple, soit toujours 1. Cela signifie que la suite est $(X_n, X_{n-1}), (X_{n-1}, X_{n-2}), ...(X_0, 0)$ avec $X_n = X_{n-1} + X_{n-2}$ puis il faudrait $X_0 = 1$, bref, on voit les nombres de Fibonacci arriver.

Prop1 : Euclide sur (F_k, F_{k-1}) fait k itérations.

Prop2 : si Euclide (n, m) fait k itérations et que $n > m$, alors $n \geq F_k$ et $m \geq F_{k-1}$.

Pour $k = 0$, alors voui, on a $m = 0 = F_{-1}$ et $n \geq 1 = F_0$.

Passage de $k - 1$ à k. Si (n,m) demande k itérations, alors la première itération donne (m,r) avec $n = qm + r$, y a plus qu'a utiliser l'hypothèse de récurrence sur (m,r) et utiliser que q vaut au moins 1, donc que $n \geq m + r$.

Le nombre d'itérations est donc le pire pour F_k et il vaut k. Puisque $F_k = \theta(((1 + \sqrt{5})/2)^k)$, on obtient que la complexité de Euclide pour PGCD de n et m est au pire en $\theta(ln_{(1+\sqrt{5})/2}(n))$

---

4 Complexités de codes pour la fonction racine entière

La racine entière (notation: `RE`) d'un entier n est le plus grand entier p tel que $p^2 \leq n$, par exemple $RE(16) = 4$, $RE(30) = 5$. On propose les codes ci-dessous pour calculer la fonction RE.

Donnez l'ordre de grandeur de la complexité de chacun d'eux en fonction de la VALEUR de l'entier n en argument. Les additions et multiplications d'entiers seront considérées de complexité constantes.

```
+--------------------------+--------------------------+-----------------------------------+
|                          | fonction RE2 (n:int): int | fonction RE3 (n:int): int         |
| fonction RE1 (n:int): int | si n == 0                | si n == 0                         |
| int cpt = 0              | alors rendre 0           | alors rendre 0                    |
| tant que cpt*cpt <= n    | sinon int y = RE2(n-1)   | sinon                             |
|      faire cpt++         |       si (y+1)*(y+1) == n |   si (RE3(n-1)+1)*(RE3(n-1)+1)==n |
| rendre cpt-1             |       alors rendre y+1   |   alors rendre RE3(n-1)+1         |
|                          |       sinon rendre y     |   sinon rendre RE3(n-1)           |
+--------------------------+-----+--------------------+-----------------------------+------+
| fonction RE4 (n:int): int       | fonction RE5 (n:int) :int                              |
| si n == 0                       | int j = 1                                              |
| alors rendre 0                  | tant que j*j <= n faire j = 2*j                        |
| sinon int y = RE4(n / 4)        | int i = j / 2                                          |
|       si (2*y+1)*(2*y+1) <= n   | tant que j>i+1 faire si ((i+j)/2)*((i+j)/2) <= n       |
|       alors rendre 2*y+1        |                       alors i = (i+j)/2                |
|       sinon rendre 2*y          |                       sinon j = (i+j)/2                |
|                                 | rendre i                                               |
+---------------------------------+--------------------------------------------------------+
```

Correction :

• *Principe de RE1, incrémenter cpt jusqu'à dépasser un peu. Exemple : combien vaut RE de 10 ? $0^2 \leq 10$ donc 0 en dessous, $1^2 \leq 10$ donc 1 en dessous, $2^2 \leq 10$ donc 2 en dessous, $3^2 \leq 10$ donc 3 en dessous, $4^2 > 10$ donc 4 strictement au dessus, donc c'est 3*

RE1 en $\sqrt{n}$ car cpt monte de 0 à environ racine(n) par pas de +1

• *Il n'est pas utile de comprendre comment RE2 fonctionne pour en donner la complexité. Le code local est en* $\theta(1)$*, on part avec argument n, on appelle sur Rec(n-1) et on finit à 0. L'arbre des appels est un arbre unaire de hauteur n, en quelque sorte, on compte á rebours de n à 0. Donc RE2 fait environ n appels.*

Comment fonctionne RE2 ? L'idée est que très souvent, RE(n)=RE(n-1), RE(34)=RE(33)=RE(32)... donc il suffit de rendre RE(n-1). Mais ça ne marche pas toujours, $RE(36) \neq RE(35)$*. On a RE(36)=RE(35)+1. En fait, RE(n)=RE(n-1) sauf si n est un carré auquel cas il faut ajouter 1. Donc on calcule récursivement RE(n-1) et on ajoute 1 si c'est un carré. Comment savoir si c'est un carré ? Si c'en est un, c'est le carré de RE(n-1)+1, on test donc si* $n=(RE(n-1)+1)^2$*.*

• *RE3 fonctionne sur le même principe que RE2, sauf qu'on ne mémorise pas RE(n-1) dans y, mais on le recalcule chaque fois que besoin. RE3(n) fait 3 appels à RE3(n-1) (deux dans le test puis un dans le "alors" ou exclusif dans le "sinon"). Ne pas croire naïvement que la complexité est juste multipliée par 3. Chacun des trois appels RE3(n-1) fait 3 appels RE3(n-2) donc y a 9 appels à RE3(n-2), puis 27 sur RE3(n-3) etc. et finalement* 3^n *appels à RE3(0). C'est donc exponentiel. Ne pas avoir stocké y engendre des répétitions catastrophiques de calculs ("Effet Fibonacci", cf poly pages 10 et 11). Une grosse erreur de débutant...*

Attention, ce piège vient très facilement et sans prévenir, il faut acquérir le réflexe de passer par une variable dès que l'on a besoin plusieurs fois d'une valeur récursive.

Si des tests ont été effectués correctement, de grands nombres ont été testés et l'on se rend compte qu'il y a un problème.

• *Il n'est pas utile de comprendre comment RE4 fonctionne pour en donner la complexité. Le code local est en* $\theta(1)$*, on part avec argument n, on appelle sur Rec(n/4) et on finit à 0. L'arbre des appels est un arbre unaire de hauteur n. On part de l'appel n et on descend jusqu'à 0 en effectuant de "/4". Combien de fois faut-il diviser par 4 pour passer de n à 0 ? Environ* $ln_4(n)(= 1/2\, ln_2(n))$ *(cf exo 2 de cette feuille).*

Comment marche RE4 ? L'idée est de faire un peu comme dans RE2, mais en se ramenant à RE(n/2) plutôt qu'á RE(n-1) pour aller plus vite. On cherche donc une relation entre $\sqrt{n}$ *et* $\sqrt{n/2}$*. On a* $\sqrt{n} = \sqrt{2}\sqrt{n/2}$*. Pas de bol,* $\sqrt{2}$ *est un méchant réel qui va m'embêter pour des calculs sur les entiers. Çela va mieux si on divise par 4 au lieu de 2 car* $\sqrt{4}$ *est un entier, c'est 2. On a* $\sqrt{n} = \sqrt{4}\sqrt{n/4} = 2\sqrt{n/4}$*. On rend donc 2 fois la valeur en n/4. En fait, il y a un petit ajustement à faire parce que c'est racine entière et pas racine tout court. si r est RE(n/4), on a* $r \leq \sqrt{n/4} < r+1$*, donc* $2r \leq \sqrt{n} < 2r+2$*, l'entier juste en dessous de* $\sqrt{n}$ *est donc 2r ou 2r+1, une comparaison avec* $(2r+1)^2$ *nous permet de savoir quelle est la bonne réponse.*

• *RE5 : Comment ça marche ? On repart sur l'idée de RE1, sauf qu'au lieu de monter par pas de +1 on va monter par pas de *2. Combien vaut RE(100) ?* $1^2 \leq 100$ *donc 1 en dessous,* $2^2 \leq 100$ *donc 2 en dessous,* $4^2 \leq 100$ *donc 4 en dessous,* $8^2 \leq 100$ *donc 8 en dessous,* $16^2 > 10$ *donc 16 strictement au dessus, donc c'est entre 8 (inclus) et 16 (exclus)*

Cette première boucle fait monter j à environ $\sqrt{n}$ *par pas de *2, donc il faudra environ* $ln_2(\sqrt{n})(= 1/2\, ln_2(n))$ *itérations (même principe qu'à l'exo 2, sauf qu'on le prend dans l'autre sens)*

La deuxième boucle est une dichotomie entre un i qui vaut de l'ordre de $\sqrt{n}$*, et le double de i. Il faut donc environ* $ln_2(\sqrt{n})(= 1/2\, ln_2(n))$ *itérations.*

Note la dichotomie est en général un peu pénible à écrire. Ici le code est très simple, mais ne pas se leurrer, on est sur une configuration très simple. Notez en particulier que la longueur de l'intervalle de départ est un puissance de 2, on ne tombera donc jamais sur un intervalle à découper de longueur impaire.

Bref, RE4 et RE5 sont les meilleurs $\theta(ln\, n)$ *puis RE1* $\theta(\sqrt{n}))$ *puis RE2* $\theta(n)$ *puis RE3* $\theta(3^n)$

---

eu de 0.

Algo des bâtons : pour additionner n et m à p et q chiffres, je trace n bâtons,

5 Faut-il trier ?

On a deux listes non triées de longueur N et n. On supposera que $N \geq n$. On cherche à savoir s'il y a un élément commun aux deux listes. Donnez 4 stratégies et comparer les ordres de grandeurs de leurs complexités. Après avoir rapidement expliqué comment elles fonctionnent, vous pouvez utiliser les fonctions et procédures suivantes (les complexités sont en nombre de comparaisons d'éléments des listes, l désigne la longueur de la liste L):

• $EstDans(x, L)$ qui teste si l'élément x apparaît dans la liste L, $EstDans(x, L)$ est de complexité $l + O(1)$

• $EstDansTriee(x, L)$ fait de même, mais en supposant la liste L triée. Sa complexité est $\ln_2 l + O(1)$

• $ElementCommunDansTriees(L1, L2)$ suppose que $L1$ et $L2$ sont triées et regarde si elles ont un élément commun. Cette fonction est de complexité $l_1 + l_2 + O(1)$.

• $Tri(L)$ qui trie la liste L. Cette fonction est de complexité $\sim l \ln_2 l$.

Correction :

Estdans : facile à comprendre.

EDT : Dichotomie. on tape au milieu puis au quart ou au trois quart selon le résultat, etc. On le fera proprement en cours. Le nombre d'itérations est environ $ln_2\, l$ *car la taille de l'espace de recherche est divisé par 2 à chaque fois.*

ECDT : plus fin, si les listes sont [2,6,7,12] et [3,4,8,9,17,21], que peut-on dire au seul vu des deux premiers 2 et 3 ? Qu'ils sont différents, que 2 est plus petit que 3 et que donc 2 n'apparaîtra pas dans la 2e liste mais que 3 pourrait apparaître dans la 1ère. On recommence donc avec [6,7,12] et [3,4,8,9,17,21]. La complexité est l1+l2 puisque un élément de l1 union l2 disparaît à chaque itération. Un code sera fait sur les listes piles puis sur les listes tableaux dans les prochaines feuilles de TD.

tri : Il y a des algos quadratiques, des en $\theta(n * ln\, n)$, des $\sim n * ln_2 n$, au pire ou en moyenne. On verra cela le long du semestre.

Les stratégies sont :

(A) Ne rien trier et faire pour tout x dans une L1, chercher x dans L2. Cela change-t-il quelque chose si on inverse le rôle des listes ? (A1) "pour tout x dans la longue, chercher x dans la courte" versus (A2) "pour tout x dans la courte, chercher x dans la longue". Intuitivement non, nous allons le confirmer. Nous dirons que c'est la même solution (A).

(B) Trier L1, et pour chaque élément x de L2, chercher x dans L1 par dichotomie. Cela change-t-il quelque chose si on inverse le rôle des listes ? (B1) "Trier la longue, et pour chaque élément x de la courte, chercher x dans la longue par dichotomie" versus (B2) "Trier la courte, et pour chaque élément x de la longue, chercher x dans la courte par dichotomie". C'est moins clair que c'est équivalent, nous allons donc considérer les deux versions (B1) et (B2).

(C) Trier les deux listes et utiliser ECDT.

Faire un sondage avant de faire les calculs, au doigt mouillé, pensez-vous que :

Si on n'en trie qu'une, il vaut mieux trier la petite ou la grande ou ça ne change rien?

Si on n'en trie qu'une, c'est le tri ou les recherches dichotomiques qui coûteront le plus cher ?

Quelle est la méthode la plus rentable ?

Vaut-il mieux tout trier ou ne rien trier ?

On fait les calculs : On regardera le terme le plus gros et le second plus gros.

(A) je ne trie rien : $N * n$ ou $n * N$ selon version (A1) ou (A2), ce qui ne change effectivement rien.

(C) je trie tout : $N * ln_2 N + n * ln_2 n + N + n$. On néglige les deux derniers termes, le plus gros terme est $N * ln_2 N$. Le plus petit est $n * ln_2 n$

(B1) je trie la grande : $N * ln_2 N + n * ln_2 N$. Le plus gros terme est $N * ln_2 N$. Le plus petit est $n * ln_2 N$ eu de 0.

Algo des bâtons : pour additionner n et m à p et q chiffres, je trace n bâtons,

(B2) je trie la petite : $n * ln_2 n + N * ln_2 n$. Le plus gros terme est $N * ln_2 n$. Le plus petit est $n * ln_2 n$

Si je ne garde que le plus grand terme, je trouve que c'est trier le plus petit qui est le plus rentable, on pourrait imaginer que si N et n sont assez proches alors le second terme renverse la vapeur (le gros terme de 5+1 est plus gros que celui de 4+3, mais le second renverse la vapeur) mais ce n'est pas le cas car le second terme de "trier la plus petite" est plus petit ou égal au second terme des autres cas.

Si on n'en trie qu'une, c'est le tri ou les recherches dichotomiques qui coûteront le plus ? Cela dépend. Si vous triez la plus petite, c'est les dichotomies sinon c'est le tri.

En bref, si on en trie qu'une, il vaut mieux trier la petite, et c'est la meilleure option (meilleur aussi que ne rien trier car $N * ln\, n$ est plus petit que $N * n$ puisque $ln\, n$ est plus petit que n.)

Et trier tout versus trier rien, on compare $N\, ln\, N$ et $N * n$. Le plus souvent, c'est trier tout qui est le plus rentable.

A MOINS QUE $n < ln\, N$, ie que $2^n < N$!!! auquel cas il vaut mieux ne pas trier.

Effectivement, si on veut savoir si un numéro parmi trois est dans l'annuaire, il vaut mieux passer trois fois dans l'annuaire (3N comparaisons) (ou bien passer l'annuaire une fois et faire trois comparaisons par entrée, ce qui revient au même, A1 et A2 sont équivalents) que trier l'annuaire. Et si l'on a 10 éléments ? 100 éléments ? Il faut commencer à trier quand $n = ln 2N$. L'annuaire de Paris contient quelques millions de noms, le ln2 vaut donc 20 et des poussières, disons 23.

Notons que le plus rentable de toutes les solutions reste de trier les trois et de regarder pour chaque numéro de l'annuaire si c'est un des trois en 2 comparaisons (médian puis grand ou petit suivant le résultat de la comparaison avec le médian), ce qui fera 2A comparaisons au total.

---

6 Complexité amortie

(1) Sur une année, vous devez payer un euro par jour sauf le 31 décembre où vous devez payer 366 euros. Quel est le coût quotidien au mieux, au pire ?

Si vous gagnez une somme fixe par jour, combien faut-il gagner pour vous en sortir ?

(2) Si la valeur de cpt est comprise entre 0 et N, quelles sont les complexité au mieux et au pire de l'opération "`cpt++`" en nombre de bits à modifier ? (les nombres sont écrits en base 2).

(3) Dans le programme suivant : "`cpt = 0 ; pour i de 1 à N faire cpt++`", quel est le coût amorti de `cpt++` ?

Correction :

(1) 1 au mieux, 366 au pire. On voit clairement que je m'en sors si je gagne 2 euros par jour, chaque jour je dépense 1 et j'économise 1, et le 31/12, je sors les économies de la banque et c'est bon.

Par contre, cela n'aurait pas fonctionné avec deux euros par jour s'il avait fallu payer les 366 euros le 1er janvier plutôt que le 31 décembre.

On veut définit la complexité amortie pour tenir compte de ce phénomène et dire que la complexité amortie est de 2 euros quotidiens.

On note c_k le coût du k^e jour.

À quelle condition n'êtes-vous pas endetté à la fin du k^e jour ? Si vous gagnez par jour au moins $(c_1+c_2+...+c_k)/k$

C'est la moyenne du coût des k premières opérations. Moyenne au sens du barycentre, pas des probas (il n'y a pas d'aléatoire dans cette histoire)

À quelle condition n'êtes-vous jamais endetté ? Si vous gagnez par jour au moins $max_k(c_1 + c_2 + ... + c_k)/k$.

C'est ce que l'on va appeler la complexité amortie.

complexité amortie $= max_k(c_1 + c_2 + ... + c_k)/k$.

Voici deux façons de prouver une complexité amortie :

En utilisant la définition. Pour $k < 365$, le taux vaut $k/k = 1$ (jusque là, un euro par jour suffit). Pour $k = 365$, le taux vaut $(364 + 366)/365 = 2$. Le max est donc 2.

En parlant de potentiel. Le potentiel, c'est l'argent qu'il vous reste à la banque. Par récurrence, au bout de k jours, si $k < 365$, il vous reste k euros, sinon 0. Vrai pour $k = 0$. Puis si vous avez k-1 euros au bout de k-1 jours, au bout de k jours, vous avez vos k-1 euros, plus deux reçus, moins un dépensé, donc k euros, sauf si k=365, auquel cas, vous avez 364+2-366=0.

Et vous n'êtes donc jamais endetté.

(2) au mieux 1, chaque fois que cpt finit par un 0 en base 2.

au pire, le nombre ne contient que des 1, on est alors en longueur de l'écriture de N en base 2, soit ln2 N

(3) On peut regarder le coût par jour, puis on calcule le coût amorti sur les k premiers jours (jour, jour en bas2, coût du jour, coût amorti des k premiers jours)

0	0		
1	1	1	1/1
2	10	2	3/2
3	11	1	4/3
4	100	3	7/4
5	101	1	8/5
6	110	2	10/6
7	111	1	11/7
8	1000	4	15/8
9	1001	1	16/9
10	1010	2	18/10
11	1011	1	19/11
12	1100	3	22/12
13	1101	1	23/13
14	1110	2	25/14
15	1111	1	26/15
16	10000	5	31/16

Au vu des premières valeurs, on a envie de conjecturer que le coût amorti est de 2. C'est bien le cas

Preuve 1 : on fait le calcul sur la formule de définition. Combien de bits a-t-on changé quand on est passé de 0 à k ? Le bit des unités est toujours changé, il est changé k fois, celui des deuzaines k/2 fois (plus exactement un peu moins : $\lfloor k/2 \rfloor$) celui des quatraines k/4 fois ($\lfloor k/4 \rfloor$) , etc

Total : $k + \lfloor k/2 \rfloor + \lfloor k/4 \rfloor + ... \leq k + k/2 + k/4 + ... = k(1 + 1/2 + 1/4 + ...) = 2k$ La moyenne barycentrique est majorée par 2, donc le coût amorti est inférieur à 2. Note, on peut calculer la moyenne barycentrique sur une puissance de 2, si $k = 2^p$, $2 - 2^p$ qui tend vers 2. La majoration est donc fine.

Note : Vous aurez peut-être dit que l'on change 1 bit une fois sur 2, 2 bits une fois sur 4, 3 bits une fois sur 8, etc., et que donc le coût est environ $\sum_i i/2^i$. Comment évaluer cette somme ? Si vous connaissez les séries entières, vous pouvez dire que $1/(1-t) = \sum_i i * t^i$ se dérive en $1/(1-t)^2 = \sum_i i * t^{i-1}$ et que donc $\sum_i i * t^i = t/(1-t)^2$ et que donc $\sum_i i/2^i = \frac{1}{2}/(1-\frac{1}{2})^2 = 2$. Sinon, dites que cette somme vaut

$$\begin{array}{ccccccccccccc}
\frac{1}{2} & + & \frac{1}{4} & + & \frac{1}{8} & + & \frac{1}{16} & + & \frac{1}{32} & + & \ldots & & 1 \\
 & & \frac{1}{4} & + & \frac{1}{8} & + & \frac{1}{16} & + & \frac{1}{32} & + & \ldots & & +\frac{1}{2} \\
 & & & & \frac{1}{8} & + & \frac{1}{16} & + & \frac{1}{32} & + & \ldots & = & +\frac{1}{4} \quad = 2 \\
 & & & & & & \frac{1}{16} & + & \frac{1}{32} & + & \ldots & & +\frac{1}{8} \\
 & & & & & & & & \frac{1}{32} & + & \ldots & & +\frac{1}{16}
\end{array}$$

Cela dit, faire une preuve rigoureuse n'est pas facile car les parties entières ne se posent pas de façon sympathique.

Preuve 2 : Avec un potentiel Changer un bit coûte 1 euro. Je vous donne 2 euros pour chaque cpt++, combien vous reste-t-il quand le compteur vaut k ? (k, k en base 2, coût quotidien, économie)

0	*0*			
1	*1*	*1*	*1*	*Je reçois 2 et dépense 1*
2	*10*	*2*	*1*	*J'ai 1, je reçois 2 et dépense 2*
3	*11*	*1*	*2*	*J'ai 1, je reçois 2 et dépense 1*
4	*100*	*3*	*1*	*J'ai 2, je reçois 2 et dépense 3*
5	*101*	*1*	*2*	
6	*110*	*2*	*2*	
7	*111*	*1*	*3*	
8	*1000*	*4*	*1*	
9	*1001*	*1*	*2*	
10	*1010*	*2*	*2*	
11	*1011*	*1*	*3*	
12	*1100*	*3*	*2*	
13	*1101*	*1*	*3*	
14	*1110*	*2*	*3*	
15	*1111*	*1*	*4*	
16	*10000*	*5*	*1*	

Il faut être observateur et voir ce qu'est l'économie en fonction de n.

Conjecture : votre économie est le nombre de 1 dans l'écriture en base 2 de k C'est vrai pour k=0 Si c'est vrai pour k, vous avez un euro par 1 dans l'écriture. k finit par p occurrences de 1 : .....01111111 : vous transformez les 1 en 0. Le nouveau nombre de 1 est l'ancien moins p (les 1 finals deviennent 0) plus 1 (un 0 devient 1).

Votre économie est l'ancienne plus deux (vous recevez deux) - (p+1) (vous dépensez p+1) soit l'ancienne moins p plus 1.

C'est la même relation de récurrence, donc ç marche encore pour k+1

---

7 Lourdeurs

Réécrire le code suivant de manière plus élégante :

```
Si p != 0 alors si (b == faux)
                    alors si q != 0 alors rendre vrai
                                    sinon rendre faux
                    sinon rendre (q==1)
              sinon rendre (b == vrai)
```

Correction :

3 lourdeurs dans ce code

- *b == vrai devrait être remplace par b, et b == faux par non(b)*
- `si test alors rendre vrai sinon rendre faux` *devrait être remplace par* `rendre test`
- `si non test alors 10 pages sinon 1 ligne` *devrait être remplacé par* `si test alors 1 ligne sinon 10 pages` *(quoi qu'ici il n'y a pas 10 pages)*

Notez que cette manie de faire le cas de base en fin de code donne

`si non test1 alors si non test2 alors si non test3 alors si non test4 alors v sinon w sinon x sinon y sinon z` *qui devrait être remplacé par* `si test1 alors z sinon si test2 alors y sinon si test3 alors x sinon si test4 alors w sinon v`. *Situation vraiment trouvée dans des copies, dans lesquelles vous ajouterez qu'il manque un sinon et que l'on ne sait donc plus quel sinon est supposé correspondre à quel si ...*

- *Le code réécrit donne :*

Si p == 0
alors rendre b
sinon si b
alors rendre (q==1)
sinon rendre q ≠ 0

---

8 Logique et codage

```
pour tout a dans A
     si     Q(a)
rendre
```

Complétez le code ci-contre pour rendre vrai ssi $\forall a \in A, Q(a)$.
Complétez le code ci-contre pour rendre vrai ssi $\exists a \in A, Q(a)$.
Donnez un code qui rend vrai ssi $\forall a \in A, \exists b \in B, \forall c \in C, P(a,b,c)$.

Correction :

(1)
r = vrai
pour tout a dans A
si non Q(a) alors r = faux (et break si possible)
rendre r

ou bien, si le return fait break comme en C

pour tout a dans A
si non Q(a) alors rendre faux
rendre vrai

(2)
r = faux
pour tout a dans A
si Q(a) alors r = vrai (et break si possible)
rendre r

ou bien, si le return fait break comme en C

pour tout a dans A
si Q(a) alors rendre vrai
rendre faux

(3)
Attention, l'ordre des quantificateurs n'est pas permutable, $\forall a \in A, \exists b \in B, R(a,b)$ n'est pas équivalent à $\exists b \in B, \forall a \in A, R(a,b)$. Exemple : $R(a,b)$ c'est $a > b$: La première phrase dit que pour tout entier, il en existe un plus grand (vrai : prendre l'entier plus un), le deuxième qu'il existe un entier plus grand que tous les autres (faux)
La formule est $\forall a \in A, \exists b \in B, \forall c \in C, P(a,b,c)$, i.e. $\forall a \in A, Q(a)$ avec $Q(a) : \exists b \in B, \forall c \in C, P(a,b,c)$ et donc le code va faire :

r1 = vrai
pour tout a dans A
si non $\exists b \in B, \forall c \in C, P(a,b,c)$
alors r1 = faux ; break si possible
rendre r1

Puisque Q1 va nécessiter un peu de code, on va passer par une variable
r1 = vrai
pour tout a dans A
r2 = $\exists b \in B, \forall c \in C, P(a,b,c)$
si non r2 alors r1 = faux ; break si possible
rendre r1

Q est $\exists b \in B, \forall c \in C, P(a,b,c)$, c'est un $\exists$, donc on utilise la technique du (2) et on utilise une variable
r1 = vrai
pour tout a dans A
r2 = faux
pour tout b dans B
r3 = $\forall c \in C, P(a,b,c)$
si r3 alors r2 = vrai ; break si possible
si non r2 alors r1 = faux ; break si possible
rendre r1

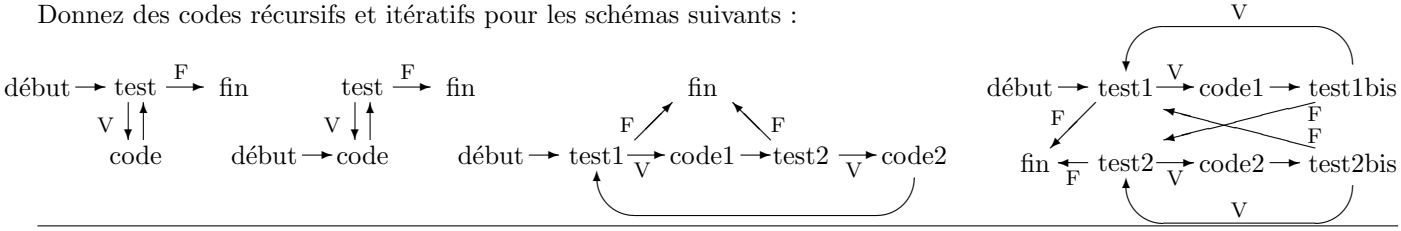
et on rejoue un dernier coup pour gérer le forall final :

```
r1 = vrai
pour tout a dans A
        r2 = faux
        pour tout b dans B
            r3 = vrai
            pour tout c dans C
                si non P(a,b,c) alors  r3 = faux ; break si possible
            si r3 alors  r2 = vrai ; break si possible
        si non r2 alors  r1 = faux ; break si possible
rendre r1

on peut aussi le gérer par des fonctions imbriquées

fct3 (a,b) : bool
pour tout c dans C
        si non P(a,b,c) alors rendre faux
rendre vrai

fct2 (a) : bool
pour tout b dans B
        si fct3(a,b) alors rendre vrai
rendre faux

fct1 (a) : bool
pour tout a dans A
        si non fct2(a) alors rendre faux
rendre vrai
```

---

9 Boucles

Donnez des codes récursifs et itératifs pour les schémas suivants :



Correction :

```
Q1rec : { si test
          alors { code ;
                  Q1rec() } }
Q1iter : { tant que test
           faire code }

Q2rec : { code ;
          si test alors Q2rec() }
Q2iter : { répéter code
          jusqu'á non test }
Q2iter : { répéter code
          tant que test } // variante suivant le langage

Q3rec : { si test1
          alors { code1 ;
                  si test2 alors { code2 ;
                                   Q3rec() } } }
```

Pour Q3iter, soit on a le break et on peut faire :

```
Q3iter : { tant que test1
           faire { code1 ;
                   si test2
```

alors code2
sinon break() } }

Le break est à utiliser avec parcimonie : il déstructure les programmes, il vaut donc mieux l'éviter, mais il y a des situations où il simplifie tellement la rédaction du code qu'il est alors toléré.

Tant qu'à utiliser le break, on peut l'utiliser à fond :

```
Q3iter : { tant que vrai
        si test1
                alors faire { code1 ;
                        si test2
                        alors code2
                        sinon break() }
                sinon break() }
```

Sans le break, il faut utiliser une variable booléenne, un "drapeau" qui va contrôler la boucle

```
Q3iter : { b = vrai
        tant que b
        faire { si non test1
                alors { b = faux }
                sinon { code1,
                        si test2
                        alors code2
                        sinon b = faux } } }
```

On peut aussi faire gérer la boucle partiellement par le break

```
Q3iter : { b = vrai
        tant que b et test1 // ne pas inverser l'ordre des deux tests
                faire { code1,
                        si test2
                        alors code2
                        sinon b = faux } } }
```

Il ne faut pas inverser l'ordre des deux tests car sinon, on effectue "test1" une fois de trop alors que b a dit que l'on s'arrêtait. Conséquence simple : sur la complexité. Plus embêtant si test1 a des effets de bord (impression ou modification de variables). Vraiment plus embêtant : code2 a modifié des variables avec pour effet que si on lance test1 à ce moment cela plante (cas typique : code2 a changé une liste l qui est devenue nulle, test2 c'est l non nul, et test1 plante sur l nul), donc votre simulation plante alors qu'elle ne devrait pas.

Pour faire Q4 en récursif, le plus simple est de faire deux fonctions qui s'appellent l'une l'autre :

```
Q4Rec : { si test1
        alors { code1 ;
                si test1bis
                alors Q4Rec()
                sinon Q4Bis() } }
Q4Bis : { si test2
        alors { code2 ;
                si test2bis
                alors Q4Bis()
                sinon Q4Rec() } }
```

Note : Pour avoir deux fonctions qui s'appellent l'une l'autre, il faudra faire un "and" en caml, une prédéclaration de Q4bis en C :

```
Q4Bis (arguments) ; // prédéclaration
Q4 (arguments) { code de Q4 }
Q4Bis (arguments) { code de Q4Bis }
```

Sans fonctions qui s'appellent l'une l'autre, il faut un drapeau qui en fait dit si l'on fait Q4Rec ou Q4Bis

```
Q4Rec : Q4Drap (vrai)
Q4Drap (b) : { si b
                alors { si test1 alors { code1 ; Q4Drap( test1bis ) } }
                sinon { si test2 alors { code2 ; Q4Drap( non test2bis ) } } }
```

En itératif, nous allons utiliser deux drapeaux, le premier dit si l'on doit continuer, l'autre si on est en haut ou en bas.

```
Q4Iter ; { stop faux, haut = vrai
        tant que non stop
        faire si haut alors { si test1 alors { code1 ; haut = test1bis } sinon stop = vrai }
```

sinon { si test2 alors { code2 ; haut = non test2bis } sinon stop = vrai } }

Si l 'on veut se passer du drapeau stop :

Q4Iter : { b = Vrai
tant que ((b et test1) ou (non b et test2))
faire { si b
alors { code1 ; b = test1bis }
sinon { code2 ; b = !test2bis } }

---

10 Petits programmes

- le log entier d'un entier strictement positif n est le nombre de fois qu'il faut le diviser par 2 pour obtenir 1.
  Donnez un code itératif, un code récursif et un code récursif terminal qui calcule le log entier.
  Que pensez-vous des trois codes ci-dessous ?

```
-------------------------------------------------------------------------------------------
|                            | LogEntier (in int n) : int    | LogEntier (in int n) : int    |
| LogEntier (in int n) : int |     int cpt                   |   int cpt = 0                 |
| int cpt = 0                |     LE(n,&cpt)                |   LE(n,cpt)                   |
| si n == 1 alors rendre cpt |     rendre cpt                |   rendre cpt                  |
| sinon cpt++                | LE (in int n, inout int *cpt) |                               |
|       LogEntier(n / 2)     | *cpt = 0                      | LE (in int n, in int cpt)     |
|       rendre cpt           | si n > 1 alors (*cpt)++       | si n > 1 alors cpt++          |
|                            |               LE(n / 2, cpt) |               LE(n / 2, cpt) |
-------------------------------------------------------------------------------------------
```

Correction :

Itératif

ln2 (n) : int
cpt = 0
tant que n>1 faire (cpt++, n = n/2)
rendre cpt

Récursif

ln2 (n) : int
si n=1 alors rendre 0 sinon rendre 1 + ln2(n/2)

Les trois codes proposés sont faux

Premier code faux : la personne qui a fait ce code (1) n'a pas compris la différence entre variable locale et variable globale, et chaque appel a sa propre variable cpt (2) tous les appels mettent leur variable à 0, il y a une sorte de reboot systématique du comptage à 0 (en fait, ce n'est pas vraiment un reboot puisque les variables sont locales donc séparées). La valeur rendue est celle du compteur du premier appel, donc 1, sauf si l'argument vaut 1 auquel cas c'est 0.

Notez d'ailleurs qu'il y a un souci avec l'appel récursif de LogEntier, qui est une fonction et appelé comme si c'était une procédure. Donc l'appel récursif est une expression alors qu'il faudrait une instruction. En C, ça compile parce que les fonctions et les procédures, les expressions et les instructions, sont mélangées. Dans un langage qui fait la différence, ça ne compilera pas.

Deuxième code faux : ici on a bien une seule variable globale cpt (qui est propagée à travers le passage inout), mais elle est reboutée à 0 à chaque appel. La fonction rendra 0, ce 0 étant celui qui a été mis par le tout dernier appel, celui avec n=1. L'initialisation de cpt est mal plaçée. Il faudrait la déplacer dans la surfonction.

Troisième code faux : cpt est passé en in, donc par copie, chaque appel a sa propre variable cpt. Le tout dernier appel, celui avec n=1 a le bon résultat dans sa variable, mais ce n'est pas la même variable que celle des appels précédents, ni celle de la surfonction qui est toujours à 0. La fonction rend toujours 0.

Pour le rec terminal, on crée une procédure récursive qui simule la boucle de l'itératif, le compteur sera passé en inout, car on s'intéresse à la valeur avant la boucle (in, initialisation avant la boucle) et à la valeur après la boucle (out, on rendra cpt après la boucle). La valeur n peut être passée en in, car on ne s'intéresse pas à ce qu'elle vaut à la sortie de la boucle.

Puis on crée une surfonction qui effectue ce qui est fait avant et après la boucle dans la version itérative, et lance la sous-procédure.

LogEntier (in int n) : int

cpt = 0

LE(n,cpt)

rendre cpt

LE (in int n, in int cpt)

si n > 1

alors cpt++

LE(n div 2,cpt)

Les versions 2 et 3 sont des versions bugées de cette réponse (2 : initialisation mal placée, 3: passage in au lieu de inout)

On peut aussi faire une sousfonction, avec un "accumulateur" qui compte :

LogEntier (in int n) : int

rendre LE(n,0)

LE (in int n, in int cpt)

si n == 1

alors rendre cpt

sinon rendre LE(n div 2,cpt +1)

En réalité, la deuxième version est la première dans laquelle le compteur inout a été séparé entre un morceau in, l'argument in de la fonction, et une partie out, le rendu de la fonction qui est bien une variable out implicite.

On notera la différence entre les deux versions.

Le compteur initialisé à 0 devient un simple appel à 0, plus besoin d'une variable pour lancer sur 0. Utiliser un cpt dans la version avec sous-fonction serait une lourdeur inutile. Par contre, ne pas utiliser une variable dans la version avec sous-procédure serait une erreur.

Le rendre cpt de la version avec sous-procédure devient rendre la valeur la fonction dans la version avec sous-fonction puisque c'est là qu'est la partie out du cpt.

Dans la procédure, on modifie le compteur d'où l'instruction cpt++. Dans la fonction on calcule la nouvelle valeur d'où l'expression cpt+1.

Si vous écrivez cpt++ à la place de cpt+1, (1) c'est sale car vous utilisez cpt++ comme expression alors que c'est moralement une instruction (2) c'est bugé car la valeur de cpt++ est celle de cpt AVANT l'incrémentation, et donc vous incrémentez votre variable locale, puis vous passez la vielle valeur du compteur, l'incrémentation n'a servi à rien. Si vous voulez programmer sale, mais correct, c'est ++cpt qu'il faut passer.

Quand n=1, dans l'instruction, il n'y a plus à modifier cpt, et c'est la surfonction qui va récupérer la valeur pour la rendre. Dans la fonction, il faut basculer la partie in du compteur en partie out, d'où le rendre n.

Dans la fonction, on REND des valeurs et on rend dans tous les cas, du coup, il n'y a pas de if sans else. Dans la procédure, on FAIT des choses, il n'y a pas de rendre, et il peut y avoir des if sans else, auquel cas c'est un "else ne rien faire"

- Donner des pseudo-codes pour la fonction `power(x,n)` qui prend en entrée x réel et n entier (supposé positif ou nul) et rend x^n: En se basant sur le principe que $x^n = x * x^{n-1}$, (1) en récursif simple, (2) En itératif, (3) en récursif terminal. (4) En se basant sur le principe que $x^p = (x^{p/2})^2$,

---

Correction :

```
power(x,n) ....  CODE 1 :  RÉCURSIF SIMPLE, PAR n-1
si n == 0
```

```
alors rendre 1
sinon rendre power(x,n-1) * x
```
fait n appels

Correction :

Itératif

Certains étudiants auront fait (erreurs constatées chez quelques étudiants):

```
power(x,n) ....FAUX....
r = x
pour i de 1 à n
     r = r * r
rendre r
```

*Erreur 1, ce n'est pas r*r mais r*x, le code ci-dessus calcule en fait* x^{2^n}.

```
power(x,n) ....FAUX....
r = x
pour i de 1 à n
     r = r * x
rendre r
```

Erreur 2, ce code fait une itération de trop et rend x^{n+1}.

```
power(x,n) ....FAUX....
r = x
pour i de 1 à n-1
     r = r * x
rendre r
```

Erreur 3, ce code rend x *au lieu de 1 si* $n = 0$

```
power(x,n) ....TRÈS LOURD ET MALADROIT....
si n == 0
alors rendre 1
sinon {
     r = x
     pour i de 1 à n-1
          r = r * x
     rendre r
     }
```

Lourd et maladroit,

il est logique d'isoler le cas de base en récursif,

mais en récursif, sauf exception, le cas de base devrait être géré par la boucle. Quand le code itératif commence par le cas de base, dans 95% des cas, c'est soit une lourdeur inutile, le cas est isolé alors que la boucle le gère très bien, soit que la boucle est mal pensée et qu'une façon plus propre permettrait de ne pas avoir à isoler le cas de base. Cette situation se retrouvera avec les listes-piles, listes chaînées. Isoler le cas liste vide est très souvent soit inutile, soit dû à un mauvais code (par exemple parce que le pointeur reste un cran en arrière avec un code de type "tant que suite(L) non vide, travail sur premier(suite(L))" plutôt que "tant que L non vide, travail sur premier(L)".

Ici, la maladresse et arrivée quand on a enlevé l'itération de trop. Au lieu de faire une itération de moins, il aurait mieux valu initialiser à 1.

```
power(x,n) ....LOURD ET MALADROIT....
si n == 0
alors rendre 1
sinon {
     r = 1
     pour i de 1 à n
          r = r * x
     rendre r
     }
```

Isoler le cas de base ne sert plus à rien, et on l'enlève pour arriver enfin à une version propre

```
power(x,n) ....  CODE 2a :  ITÉRATIF, PAR n-1
r = 1
```

```
pour i de 1 à n
     r = r * x
rendre r
```

On peut faire une variante avec un while

```
power(x,n) ....  CODE 2b :  ITÉRATIF, PAR n-1
r = 1
tant que n > 0
     r = r * x
     n--
rendre r
```

Notez que dans 2a, on compte jusqu'à n, alors que dans 2b, on compte à rebours ce qui économise une variable.

Ce code fait n itérations.

Correction :

Pour faire une version récursive terminale, on simule la version itérative. Une procédure va modifier le compteur r, comme dans la boucle. C'est du inout puisque l'on s'intéresse à la valeur de r avant et après la boucle (init avant, return après). Par contre, on peut gérer n avec un argument in puisque l'on ne s'intéresse pas à la valeur de n à la sortie de boucle. Une surfonction va faire le travail qui est fait hors de la boucle dans la version itérative et va appeler la sous-procédure pour simuler la boucle.

power (x,n) .... CODE 3 : RÉCURSIF TERMINAL AVEC SOUS-PROCÉDURE, PAR n-1
    int r = 1
    pow(x,n,& r)
    rendre r
*pow(x,n, inout *r)*
    si n == 0 alors ne rien faire
    *sinon { *r = (*r)*x*
        pow(x,n-1, r) }

Autre option, technique dite de l'accumulateur, on propage l'accumulateur dans une sous-fonction

power (x,n) .... CODE 4 : RÉCURSIF TERMINAL AVEC SOUS-FONCTION, PAR n-1
    rendre pow(x,n,1)
pow(x,n,r)
    si n == 0 alors rendre r
    *sinon rendre pow(x,n-1, r*x)*

La version 4 est en réalité la version 3 dans laquelle la variable inout a été coupée en deux parties, la partie in est l'argument de la sous-fonction, et la partie out est dans le rendu de fonction. Observez les différences entre ces deux versions. Ce sont les mêmes que dans les version récursives terminales de log entier. Cf la correction de log entier pour les remarques sur les deux versions.

Ramarque : la version itérative qui a été simulée est la 2b.

Si nous avions simulé la version 2a, nous aurions fait des sous-fonctionnalités avec deux argument (i,n) au lieu d'un. Le test n==0 serait devenu i==n, le n-1 serait remplaçé par i+1. Cela aurait été une lourdeur avec un argument supplémentaire de trop. Il est préférable de compter à rebours. Cf les codes maladroits ci-dessous (variantes des codes 3 et 4) qui ne comptent pas à rebours.

power (x,n) .... CODE MALADROIT UN ARGUMENT DE TROP CAR NE COMPTE PAS À REBOURS
    r = 1

pow(x,n,1,r)
rendre r
pow(x,n,i,r)
si n == i alors ne rien faire
*sinon { r = r*x*
pow(x,n, i+1, r) }

power (x,n) .... CODE MALADROIT UN ARGUMENT DE TROP CAR NE COMPTE PAS À REBOURS
rendre pow(x,n,1,1)
pow(x,n,r)
si n == i alors rendre r
*sinon rendre pow(x,n, i+1, r*x)*

Correction :

Toutes les versions vues jusquà présent sont de complexités $\theta(n)$, n itérations ou n appels car on appelle récursivement sur n-1.

On souhaite aller plus vite en appelant sur n/2. Nous devrions donc passer à une complexité en ln2 n.

Pour cela, nous utilisons $x^{2p} = (x^p)^2$ et $x^{2p+1} = (x^p)^2 * x$ ($x^{100} = (x^{50})^2$ et $x^{101} = (x^{50})^2 * x$)

On propose dans un premier temps :

```
power(x,n) ....  CODE 5 :  RÉCURSIF SIMPLE, PAR n/2, MAIS BOURDE
si n == 0 alors rendre 1
sinon
     si n est pair
     alors rendre power(x,n div 2) * power(x,n div 2)
     sinon rendre power(x,n div 2) * power(x,n div 2) * x
```

mais un smiley grimaçant apparaîtra sur votre copie si vous faites cela. En effet, vous appelez deux fois l'appel récursif au lieu d'un. Quelqu'un de naîf dira que la complexité est multipliée par deux, mais c'est bien pire que cela. Il y a deux appels récursif au lieu d'un, quatre appels d'appels au lieu d'un, et à profondeur k, 2^k appels au lieu d'un. Le surcoût est de type exponentiel avec un effet Fibonacci. Pour évaluer la complexité, je dessine l'arbre des appels, il est de profondeur $ln2(n)$ (car l'entier associé est divisé par deux à chaque étage) et y a 2^p nœuds à profondeur p, donc in fine, il y a $2^{ln2(n)} = n$ appels finals ! (soit exponentiel en la complexité attendue puisque n est une exponentielle de ln2(n)) recoit

On n'a rien gagné ! cette complexité se comprend si on regarde comment est calculé x^{10}. Dans la version récursive simple, c'est calculé comme $((((((((((1 * x) * x) * x) * x) * x) * x) * x) * x) * x) * x)$. À présent, c'est transformé en $x^5 * x^5$ puis en $(x^2 * x^2 * x) * (x^2 * x^2 * x)$ puis en $((x * x) * (x * x) * x) * ((x * x) * (x * x) * x)$. On voit que l'on effectue $x * x * x * x * x * x * x * x * x * x * x * x * x * x * x$ comme dans les premières versions, sauf qu'au lieu de parenthéser toujours à gauche, on fait une sorte de parenthésage dichotomique.

IL NE FAUT SURTOUT PAS APPELER DEUX FOIS UN MÊME APPEL RÉCURSIF. Sinon effet Fibonacci et coût exponentiel en ce qu'il devrait être. Dans cette configuration, il faut appeler une seule fois et stocker dans une variable, variable que l'on consulte par la suite.

Bonne version :

```
power(x,n) CODE 6 :  RÉCURSIF SIMPLE, PAR n/2, OK
si n == 0 alors rendre 1
sinon
     y = power(x,n div 2)
     si n est pair
     alors rendre y * y
     sinon rendre y * y * x
```

fait ln2(n) appels; x^{10} est calculé comme $((x^2)^2 * x)^2$

On peut toujours simuler cet algo en itératif, mais ce sera lourd, avec une pile, car le récursif n'est pas terminal.

Pour aller vers du récursif terminal ou de l'itératif en ln2(n), il faut changer légèrement l'idée, et au lieu de dire que $x^{100} = (x^{50})^2$, dire que $x^{100} = (x^2)^{50}$ (et $x^{101} = (x^2)^{50} * x$), d'où la variante du code 6

```
power(x,n) CODE 7 :  RÉCURSIF SIMPLE, PAR n/2, VARIANTE, OK
si n == 0 alors rendre 1
sinon
     si n est pair
     alors rendre power(x*x,n/2)
     sinon rendre power(x*x,n/2) * x
```

Remarque, dans les codes 6 et 7, on aurait pu calculer le carré en appelant power(...,2), ce qui aurait donné dans le code 6 : rendre power(y,2) (voire (power(power(x,n div 2),2)) en se passant de y) et dans le code 7 : power(power(x,2),n/2). GAG : dans ce cas, power(...,2) appelle power(...,2) et le code boucle ! Un mauvais patch serait de gérer le cas n==2 comme cas de base (mais pas le cas n==1 !), bof, le bon patch est de ne pas utiliser power pour calculer un carré.

*Ce dernier code permet de voir comment faire du récursif terminal en ln2(n). Il suffit de reprendre l'idée mais de mettre les facteurs "*x" dans un accumulateur :*

```
power(x,n) CODE 8 :  RÉCURSIF SIMPLE, PAR n/2, AVEC SOUS-FONCTION
     rendre pow(x,n,1)
pow(x,n,r)
si n == 0 alors rendre r
sinon
     si n est pair
     alors rendre pow(x*x,n/2, r)
     sinon rendre pow(x*x,n/2, r*x)
```

Le même, avec une sous-procédure

```
power(x,n) CODE 9 :  RÉCURSIF SIMPLE, PAR n/2, AVEC SOUS-PROCÉDURE
     r = 1
     pow(x,n, & r)
     rendre r
pow(x,n,inout *r)
si n == 0 alors { rien }
sinon
     si n est impair alors *r = (*r)*x
     pow(x*x,n/2, r)
```

Ce qui nous donne la version itérative en ln2(n) itérations. x^{10} sera calculé comme $x^8 * x^2$.

```
power(x,n) CODE 10 :  ITÉRATIF, PAR n/2
r = 1
tant que n n'est pas nul
     si n est impair alors r = r * x
     n = n div 2
     x = x * x
rendre res
```

- Le jeu de Hanoï: On a 3 piquets. En position initiale, on a n disques sur le premier piquet, chacun reposant sur un disque de taille plus grande ; sur les second et troisième piquets, il n'y a pas de disques. À chaque étape, on a le droit de déplacer un disque d'un piquet à l'autre, à condition qu'un disque ne soit jamais posé sur un disque plus petit. Le but du jeu est de déplacer les n disques du piquet 1 au piquet 3.

  Montrez qu'il y a une solution pour tout n. Puis écrire une procédure qui prend n en argument et qui affiche la liste des opérations à effectuer pour résoudre le problème des tours de Hanoï à n disques.

Correction :

Par récurrence sur n : il y a une solution.

$n = 0$: OK,

$P(n-1) => P(n)$: J'ai n disques, dis-je que c'est un petit et $n-1$ autres, ou bien que c'est un grand et $n-1$ autres ?

Je dis que n disques, c'est le grand disque plus les n-1 autres (Note : il est possible de faire une démonstration en isolant le petit ! Cf plus bas)

Par HR, je déplace les n-1 disques de 1 à 2, puis je déplace le grand disque de 1 à 3, puis à nouveau par HR, je déplace les n-1 disques de 2 à 3.

Ici, froncez les sourcils, et prenez la casquette d'un matheux et dites NON, car l'HR au rang n-1 me dit qu'il y a une solution pour n-1 pour aller de 1 à 3, mais pas qu'il y a une solution pour aller de 1 à 2, ni de 2 à 3.

Solutions pour faire la démo propre : renommer les piquets 1 en 1', 2 en 3', 3 en 2'. OU BIEN généraliser le résultat et montrer $Q(n)$ pour tous piquets $p \neq q$, il y a une solution avec n disques pour aller de p à q.

OK, c'est des maths, ça nous casse les pieds.

Sauf que : programme récursif :

On peut tenter de suivre la preuve par récurrence :

```
Hanoi (n) ....  FAUX ....
     if n>0
          Hanoi(n-1) ;
          printf(deplacez de 1 vers 3 ;
          Hanoi(n-1) ;
```

souci, si on fait un appel récursif avec n-1, ça envoie les disques sur 3 au lieu de 2.

In fine, le code ci-dessus n'écrit rien d'autre que des "1 vers 3"

solution ? et bien, on fait exactement comme pour la démo de math, on généralise, i.e. on écrit une procédure à 3 arguments (n,D,A) qui envoie de D(épart) à A(rrivée), puis on l'utilise avec une surprocédure pour envoyer de 1 à 3 (Notez qu'il est hors de question de demander à l'utilisateur d'appeler H avec argument 1,3. On me demande une procédure à un seul argument, je fournis une procédure à un argument. Ce qui ne veut pas dire qu'il m'est interdit d'utiliser une procédure de travail avec plus d'arguments. On peut calculer la coordonnée du piquet M(édian), $M = 6 - D - A$, mais il est moins lourd de l'intégrer dans les arguments.

```
Han (n,D,A,M)
     if n>0
          Han(n-1,D,M,A) ;
          print (déplacez de D à A) ;
          Han(n-1,M,A,D) ;

Hanoi (n)
     Han(n,1,3) ;
```

---

11 La fonction de Ackermann

Les fonctions de Ackermann sont définies comme suit : $A_0(n) = n+1$ puis :
$A_{m+1}(n) = A_m(A_m(A_m....(A_m(1))...))$ avec $n+1$ appels composés de A_m.

- Donnez un pseudo-code de Ackermann(m,n) $= A_m(n)$ avec de l'itératif et du récursif.
- Donnez un pseudo-code de Ackermann purement récursif.
- Donnez les premières valeurs de la suite $(A_m(0))_{m \in N}$

Correction :

On écrit A(m,n).

Le premier code suit la définition : si m == 0 alors n+1 sinon $A_{m-1}(A_{m-1}(A_{m-1}....(A_{m-1}(1))...))$ avec $n+1$ itérations.

Notez que la définition donne A_{m+1} en fonction de A_m mais que pour l'écrire, il faut la traduire avec A_m en fonction de A_{m-1}.

$A_m(n) = A_{m-1}(A_{m-1}(A_{m-1}....(A_{m-1}(1))...))$ avec $n+1$ appels composés de A_{m-1}.

Comment calculer λ itérations d'une fonction f sur x ? En faisant $r = x$ puis pour i de 1 à λ, $r = f(r)$.

La fonction itérée est $f(r) = A(m-1, r)$, on part de $r = 1$ et on fait $n+1$ itérations, d'où le code :

```
A(m,n) : int
     { si m == 0 alors rendre n+1
          sinon { r = 1
                  pour i de 1 à n+1
                     faire r = A(m-1,r)
```

rendre r } }

Pour faire du récursif pur, il faut remarquer que $A_{m-1}^{(n+1)}(1) = A_{m-1}(A_{m-1}(A_{m-1}....(A_{m-1}(1))...)) = A(m-1, X)$ *avec* $X = A_{m-1}(A_{m-1}(A_{m-1}....(A_{m-1}(1))...)) = A_{m-1}^{(n)}(1)$. *Puisque l'itération n'est faite que n fois, X est égal à A(m,n-1). D'où la formule de récurrence A(m,n) = A(m-1,A(m,n-1)). Cette formule est valable si* $n > 0$. *Pour* $n = 0$, *on a A(m,0)=A(m-1,1) (une itération). D'où le code :*

```
A(int m,n) : int
      { si m == 0 alors rendre n+1
          sinon
          si n == 0 alors rendre A(m-1,1)
          sinon rendre A(m-1,A(m,n-1)) }
```

Correction :

Au début, on est naïf (Les points de suspension ne refont pas les calculs qui ont été faits les lignes au dessus.)
Ack(0) = A(0,0) = 1
Ack(1) = A(1,0) = A(0,1) = A(0,0) + 1 = 2
Ack(2) = A(2,0) = A(1,1) = A(0,A(1,0)) = ... = A(0,2) = A(0,1) + 1 = ... = 2+1 = 3
Le suivant est-il 4 ?
Ack(3) = A(3,0) = A(2,1) = A(1,A(2,0)) = ... = A(1,3) = A(0,A(1,2)) = A(0,A(0,A(1,1))) = ... A(0,A(0,3)) = A(0,4) = 5
Devinez le suivant ? 8 parce que Fibonacci ? Déroulez le calcul (patience nécessaire) et vous trouverez 13. Ou bien programmez la fonction et profitez-en pour lui demander le suivant : il vous dit 65533.

Pour comprendre ce qui se passe, il faut essayer de voir ce que sont A(0,n), A(1,n), A(2,n), etc. On sait que A(0,n) = n+1, et ensuite ?

$Ack(1) = A(1,0) = A(0,1) = 2$
et
$A(1,n) = A(0, A(1,n-1)) = A(1,n-1) + 1$
et donc par récurrence (on résout $x_0 = 2, \forall n > 0, x_n = x_{n-1} + 1$*)*
$A(1,n) = n + 2$

$Ack(2) = A(2,0) = A(1,1) = 3$
et
$A(2,n) = A(1, A(2,n-1)) = A(2,n-1) + 2$
et donc par récurrence (on résout $x_0 = 3, \forall n > 0, x_n = x_{n-1} + 2$*)*
$A(2,n) = 2n + 3$

$Ack(3) = A(3,0) = A(2,1) = 5$
et
$A(3,n) = A(2, A(3,n-1)) = 2A(3,n-1) + 3$
On cherche donc à résoudre $y_0 = 3, \forall n > 0, y_n = 2 * y_{n-1} + 3$ *C'est une suite "arithmético-géometrique" ...*

D'une manière générale, on peut rapprocher les récurrences de type $x_{n+1} = a_n * x_n + b_n$ *(ici,* a_n *et* b_n *sont constants) des équations différentielles de type* $y'(x) = a(x) * y(x) + b(x)$. *(On peut généraliser ce qui suit aux ordres supérieurs). Les techniques pour les résoudre sont les mêmes. On cherche à résoudre l'équation de récurrence puis on fixe les paramètres avec les conditions initiales.*
Pour résoudre l'équation sans la condition initiale :

Technique 1 : Solution = solution de l'équation homogène + solution particulière
Il faut deviner une solution de $\forall n > 0, y_n = 2 * y_{n-1} + 3$. *On cherche une constante. La suite constante c fonctionne ssi* $c = 2c + 3$ *ie ssi* $c = -3$
Puis on résout l'équation homogène, $\forall n > 0, y_n = 2 * y_{n-1}$. *Ce qui donne* $y_n = A * 2^n$.
La solution générale de l'équation non homogène est donc $y_n = A * 2^n - 3$ *puis la condition initiale* $y_0 = 5$ *donne* $A = 8 = 2^3$ *et on a donc* $A(3,n) = 2^{n+3} - -3$

Technique 2 :
Si vous n'avez pas trouvé une solution particulière, vous pouvez utiliser la technique de la variation de la constante:

On résout l'équation homogène, puis on cherche une solution de la forme $A(n) * 2^n$, ce qui donne $A(n) = A(n-1) + 3/2^n$ soit $A(n) = A(0) + \sum_1^n 3/2^n = 5 + 3(1 - 1/2^n) = 8 - 3/2^n$ puis $A(3,n) = 2^{n+3} - 3$

Technique 3 : on déroule la récurrence

$$\begin{aligned} y_n &= 2y_{n-1} + 3 \\ &= 2(2y_{n-2} + 3) + 3 \\ &= 2^2 y_{n-2} + 2 * 3 + 3 \\ &= 2^2(2y_{n-3} + 3) + 2 * 3 + 3 \\ &= 2^3 y_{n-2} + 3(2^2 + 2 + 1) \\ &= 2^k y_{n-k} + 3(2^{k-1} + ... + 1) \\ &= 2^n y_0 + 3(2^n - 1) \\ &= 2^{n+3} - 3 \end{aligned}$$

Technique 4 : On peut aussi, et c'est un réflexe à prendre, regarder les premières valeurs et deviner :
$y_0 = 13, y_1 = 29, y_2 = 61, y_3 = 125, y_4 = 253, y_5 = 509, y_6 = 1021$
Il se voit que ce sont des puissances de 2 moins 3, reste à regarder quelle puissance de 2 et on voit la formule.

$Ack(4) = A(4,0) = A(3,1) = 2^4 - 3 = 13$
(tiens c'est pas 8...)
On remarque que chaque terme zéro d'un rang est en fait le terme 1 du rang d'avant, c'est ce que dit la formule $A(m,0) = A(m-1,1)$
et
$A(4,n) = A(3, A(4,n-1)) = 2^{A(4,n-1)-3} + 3$
On remarque que la formule en fonction de n au rang précédent donne la formule de récurrence au rang courant. Il suffit donc de remplacer n dans la formule du rang précédent (ici $x_n = 2^{n+3} - 3$*) par le terme précédent au courant (ici* $y_n = 2^{y_{n-1}+3} - 3$*).*
C'est ce que dit la formule $A(m,n) = A(m-1, A(m,n-1))$ *: on applique la formule du rang précédent A(m-1,...) au terme précédent A(m,n-1)*
revenons à A(4,...). En déroulant la récurrence (technique 3 ci-dessus), on trouve :

$$A(4,n) = (2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow n) - 3$$

$A(5,0) = A(4,1) = 2^{16} - 3 = 65533$

A partir de maintenant, nous allons ignorer le négligeable "-3"

$$A(5,n) = A(4, A(5,n-1)) \approx 2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow \text{A(5,n-1)}$$

Si je prends le premier terme, cela donne

$$A(6,0) = A(5,1) \approx 2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow 65533$$

Notons que $2^{65536} = 10^{\sim 20000}$*, c'est beaucoup plus que le nombre de particules dans l'univers.*
$2^{2^{65536}} = 2^{10^{\sim 20000}}$*, il n'y a pas assez de particules dans l'univers pour l'écrire.*
$2^{2^{2^{65536}}} = 2^{2^{10^{\sim 20000}}}$*, il n'y a pas assez de particules dans l'univers pour écrire combien de chiffres qu'il faut pour l'écrire.*
$2^{2^{2^{2^{65536}}}} = 2^{2^{2^{10^{\sim 20000}}}}$*, il n'y a pas assez de particules dans l'univers pour écrire combien de chiffres qu'il faut pour écrire combien de chiffres qu'il faut pour l'écrire.*
il n'y a pas assez de particules dans l'univers (pour écrire combien de chiffres qu'il faut)^{65530} *pour écrire A(5)*

et plus généralement :

$$A(5,n) = \underbrace{2^{2^{\cdots^{2^{16}}}} \updownarrow 2^{2^{\cdots^{2^{16}}}} \updownarrow ... \updownarrow 65533}_{n+1}$$

On obtient $A(7,0) = A(6,1) = A(5, A(6,0))$ *en iterant* $A(6,0)$ *fois à partir de 65535 le process "remplacer r par"*

$$2^{2^{\cdots^{2^{16}}}} \updownarrow r$$

soit

$$2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow 2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow \ldots \updownarrow 65533$$

$$2^{2^{2^{\cdots^{2^{2^{16}}}}}} \updownarrow 65533$$

puis A(6,n) =

$$2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow 2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow 2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow \ldots \updownarrow 65533$$

$$2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow 2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow \ldots \updownarrow 65533 \qquad n+1$$

$$\ldots$$

$$2^{2^{2^{\cdots^{2^{2^{16}}}}}} \updownarrow 65533$$

$A(8,0) = A(7,1) = A(6, A(7,0)) = A(6, A(6,1)) =$

$$2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow 2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow 2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow \ldots \updownarrow 65533$$

$$2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow 2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow \ldots \updownarrow 65533$$

$$\ldots$$

$$2^{2^{2^{\cdots^{2^{2^{16}}}}}} \updownarrow 65533$$

$$2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow 2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow 2^{2^{2^{\cdots^{2^{16}}}}} \updownarrow \ldots \updownarrow 2$$

$$2^{2^{2^{\cdots^{2^{2^{16}}}}}} \updownarrow 65536$$

et caetera

---

12 Les pièces

Que fait la fonction suivante ? :

```
function f (n : integer) : integer ;     (* n >= 0 ; n est une somme en francs *)
var     i,j,k,l,m,  sum  : integer ;
begin   sum := 0 ;
        for i := 0 to n        do
         for j := 0 to n div 2  do
          for k := 0 to n div 5  do
           for l := 0 to n div 10 do
            for m := 0 to n div 20 do
                if i + 2*j + 5*k + 10*l + 20*m = n then sum := sum +1 ;
        f := sum
end ;
```

Donnez un équivalent de sa complexité en nombre de tests. Peut-on améliorer cette complexité ?

Correction :

Oui, c'est un vieil exo, il parle de francs et le code est en Pascal...

$f(n)$ est le nombre de façons d'obtenir n francs avec des pièces de 1, 2, 5, 10 et 20 francs.

par exemple, f(6)=5 car il y a 5 façons de décomposer 6 francs : 6=5+1= 2+2+2 = 2+2+1+1 =2+1+1+1+1= 1+1+1+1+1+1

Il est facile de voir que la complexité est $\sim n^5/2000$ Le "for m ..." fait $n/20$ tests, puis "for l, for m ..." fait $n/10$ fois $n/20$ tests donc en fait $n^2/200$, etc.

Une amélioration naturelle consiste à arrêter la boucle des que les pièces prises font une somme supérieure à n. Facile à coder, il suffit de s'arrêter quand la somme obtenue avec les nouvelles pièces dépasse la somme restant à obtenir:

```
        for i := 0 to n do
    for j := 0 to (n - i) div 2 do
    for k := 0 to (n - i - 2*j ) div 5 do
    for l := 0 to (n - i - 2*j - 5*k ) div 10 do
```

```
  for m := 0 to (n - i - 2*j - 5*k - 10*l ) div 20 do
       if i + 2*j + 5*k + 10*l + 20*m = n then sum := sum +1 ;
```

ça améliore, mais de quoi ?

Intuitivement (??), on est toujours en n^5, mais on améliore d'un facteur constant.

Quel est ce facteur ? et pourquoi on serait toujours en n^5 ?

Regardons : si je fais

```
      for i := 0 to n do
  for j := 0 to n do truc
```

Je vais avoir combien de truc ? $\sim n^2$. On peut dire que c'est le nombre de points à coordonnées entières dans un carré de coté n.

Et si je fais

```
      for i := 0 to n do
  for j := 0 to i do truc
```

Ça va être $\sim n^2/2$. C'est le nombre de points entiers dans un certain triangle qui est un demi carré de coté n.

On applique ce principe à notre fonction f améliorée. On a un test pour chaque quintuplet qui respecte les bornes, La complexité est $card\{i,j,k,l,m \geq 0 | i+2j+5k+10l+20m \leq n\}$. C'est à peu près le nombre de points entiers dans un volume délimité par $i,j,k,l,m \geq 0, i+2j+5k+10l+20m \leq n$ dont le volume vaut

$\int\int\int\int\int_{\{i,j,k,l,m \geq 0 \wedge i+2j+5k+10l+20m \leq n\}} 1\, di\, dj\, dk\, dl\, dm$

on fait des changements de variables linéaires pour sortir les n et les constantes $i = \bar{i} * n, 2j = \bar{j} * n, ...$, ce qui nous donne

$n^5/2000 \int\int\int\int\int_{\{i,j,k,l,m \geq 0 | i+j+k+l+m \leq 1\}} dV$

Remarquez qu'ici, si on était parti de l'algo de la feuille de TD, on aurait $\int\int\int\int\int_{\{0 \leq i,j,k,l,m \leq 1\}} dV$ qui vaut 1 car c'est le volume d'un cube (en dimension 5 ... so what ?).

On voit ici déjà que c'est bien du n^5 avec une amélioration de type facteur multiplicatif, ce facteur étant notre intégrale. Sans l'optimisation, le facteur d'amélioration est 1, ce qui est normal.

Que vaut cette intégrale. On généralise à une dimension quelconque : $\int_{\forall i, x_i \geq 0 \wedge \sum_i x_i \leq 1} dV$

ici, on peut regarder le volume de l'intégrale I_N en dimension N

$N = 1$ ça fait 1, c'est une ligne.

$N = 2$ ça fait $1/2$, c'est un demi carré.

$N = 3$, ça fait $1/6$, car c'est un cône dont la base est un demi-carré, Constatez que l'on retrouve le demi carré de N=2 quand on a hauteur 0, que ça passe par le point (0,0,1) et que la limite est un plan (faire un dessin) et le volume d'un cône, c'est la surface de la base fois la hauteur, sur 3.

Quand on passe de $N-1$ à N, ça multiplie par $1/N$ en effet, un volume est l'intégrale sur la hauteur h, de la surface de la section du volume par le plan de hauteur h.

Ici, on fait varier de h=0 au point (0,0,.., 0,1) à h=1 aux points (.,.,...,0), autrement h, c'est $1-z$ ou z est la valeur de la dernière coordonnée

La section à $h = 1$, c'est notre intégrale mais en dimension -1, ça fait I_{N-1}

la section à $h = 0$, c'est un point, ça fait 0

La section à h, c'est notre I_{N-1} mais ayant subi une homothécie de facteur h (Nous sommes en présence d'un cône en dimension N). Comme on est en dimension $N-1$, on trouve que le volume est $h^{N-1} I_{N-1}$ et donc $I_N = \int_0^1 h^{N-1} I_{N-1} dh = I_{N-1} 1/N$ et donc par récurrence, on a $I_N = 1/N!$

Donc ici notre amélioration est d'un facteur 120.

Autre façon de calculer l'intégrale :

Je fais le changement de variable $i' = i$, $j' = i + j$, $k' = i + j + k$, etc. Je tombe sur $\int_{0 \leq i \leq j \leq k \leq l \leq m \leq 1} dV$ Ce truc revient à prendre un quintuplet de réels dans $[0,1]$ et a ne le garder que s'il est classé. La proba que deux éléments soient égaux est nulle. Tous les ordres sont possibles et équiprobables. L'ordre ordonné a donc une chance sur $N!$ d'arriver, ce qui veut dire que notre intégrale vaut $1/N!$

SI on inverse, l'ordre, ça ne change rien :

```
      for m := 0 to n div 20 do
  for l := 0 to n div 10 do
  for k := 0 to n div 5 do
  for j := 0 to n div 2 do
  for i := 0 to n do
       if i + 2*j + 5*k + 10*l + 20*m = n then sum := sum +1 ;
```

est en $n^5/2000$

```
      for m := 0 to n div 20 do
  for l := 0 to (n - 20*m ) div 10 do
  for k := 0 to (n - 20*m -10*l ) div 5 do
  for j := 0 to (n - 20*m -10*l -5*k ) div 2 do
  for i := 0 to n - 20*m -10*l -5*k - 2* j do
```

```
          if i + 2*j + 5*k + 10*l + 20*m = n then sum := sum +1 ;
```

est en $1/120 * n^5/2000$

mais dans la fin du dernier, je constate que je suis en train de faire

```
      for i := 0 to n - truc do
        if i + truc = n then sum := sum +1 ;
```

et la je dis, que sum s'incrémentera pour une seule valeur de i, la dernière. Dit autrement, j'ai pris des pièces de 2,5,10,20 sans dépasser n, il y a alors une seule façcon de compléter avec des pièces de 1 pour avoir n.

Ça veut dire que ce passage peut être remplace par `sum := sum +1 ;` *d'ou le code en* n^4 *:*

```
      for m := 0 to n div 20 do
    for l := 0 to (n - 20*m ) div 10 do
    for k := 0 to (n - 20*m -10*l ) div 5 do
    for j := 0 to (n - 20*m -10*l -5*k ) div 2 do
         sum := sum +1 ;
```

puis je dis que `for x := 1 to P do cpt ++` *équivaut à* `cpt := cpt + P` *d'ou algo en* n^3

```
      for m := 0 to n div 20 do
    for l := 0 to (n - 20*m ) div 10 do
    for k := 0 to (n - 20*m -10*l ) div 5 do
         sum := sum + 1+ (n - 20*m -10*l -5*k ) div 2
```

SI je veux progrsser, ça se complique, je me retrouve à avoir besoin d'une formule pour $\sum_k (n - 20 * m - 10 * l - 5 * k)\, div\, 2$ *qui existe, mais il faut discuter en fonction de parités, sèparer termes pairs et impairs, tout ça à cause de la division ENTIÈRE par 2, puis ...*

Et si je veux encore progresser, alors .... beuhh

L'algo du TD, si on le généralise à k pièces de valeurs $V[1], ..., V[k]$ *est de complexitè* $\theta(n^k/\Pi V[j])$

Si on veut améliorer sérieusement, il faut repenser le problème.

On essaye d'avoir une récurrence :

Soit X_n *le nombre de façons de décomposer n avec 1,2,5,10,20.*

Il y a deux sortes de décompositions : celles qui contiennent au moins une pièce de 20, et celles qui n'en contiennent aucune.

Des décompositions qui ont au moins une pièce de 20, il y en a X_{n-20} *(si* $n < 20$*, soit on dit que* $X_n = X_{n-20}$*, soit on garde la formule et on dit que* X_p *est nul quand p est négatif)*

Des décompositions qui n'ont pas de pièces de 20, ce n'est pas un $X_{...}$*.*

Ce sera Y_n *ou* Y_n *est le nombre de façons de décomposer n avec 1,2,5,10*

et on a donc $X_n = X_{n-20} + Y_n$

puis de la même façon, $Y_n = Y_{n-10} + Z_n$ *puis* $Z_n = Z_{n-5} + W_n$*, etc.*

De façon à généraliser, on va plutôt poser $H_{q,n}$ *le nombre de façons de décomposer n en utilisant les q premières pièces, de valeur* $V[1], ..., V[q]$*, et l'on a alors*

$H_{q,n} = H_{q,n-V[q]} + H_{q-1,n}$ *sauf dans les cas particuliers.*

Du coup, on peut faire une fonction récursive :

PS : leur demander ce que vaut H(0,m) et H(q,0). Voir que ça se contredit pour H(0,0), et leur demander quelle est la bonne réponse pour H(0,0)

```
fonction H(q,n) :  int
  si n == 0 alors return 1
  else
  si q == 0 alors return 0
  else
  si n < V[q] alors return H(q-1,n)
  else return H(q,n-V[q]) + H(q-1,n)
```

Quel en est la complexité ?

En fait, on tombe sur du n^k

En effet, pour $q = 0$*, c'est du* $\theta(1)$

Pour $q = 1$*, on a* $Comp_n = Comp_{n-V[1]} + \theta(1)$ *ce qui donne du n*

Pour $q = 2$*, on a* $Comp_n = Comp_{n-V[2]} + \theta(n)$ *ce qui donne du somme de n, donc du* n^2

Pour $q = 3$*, on a* $Comp_n = Comp_{n-V[3]} + \theta(n^2)$ *ce qui donne du somme de* n^2*, donc du* n^3

etc.

mais si on dessine l'arbre des appels de $H_{5,100}$

(DESSIN Á FAIRE AU TABLEAU; Pour cette correction, je code par des G et des D, par exemple GGDGD c'est le fils droit du gauche du droit du gauche du gauche de la racine)

on a un appel $H_{q,n}$ *sur le nœud x avec q et n qui valent :*

racine : 5,100

G : 5,80

D : 4,100
GG : 5,60
GD : 4,80
DG : 4,90
DD : 3,100
...
DGG : 4,80 que l'on a déjà calculé comme GD et que l'on recalcule.

Et puis après ça empire, par exemple H(3,70), on l'a comme GDGD, GDDGG, DGGGD, DGGDGG, DGDGGGG et DDGGGGGG ...

Que feriez-vous à la main ? Vous mémoriseriez les résultats déjà calculés. Ce que l'on se propose de faire. On introduit un tableau H[0..k,0..n]

Option 1 : on y met -1 partout, ce qui code "non calculé", puis on utilise la fonction récursive à laquelle on ajoute : commencer par regarder si le tableau contient autre chose que -1, si oui, rendre la valeur du tableau, sinon lancer le calcul puis sauver le résultat dans le tableau

Option 2 : on oublie le récursif et on remplit notre tableau. Sont faciles à calculer la première ligne et la première colonne. On souhaite avoir le résultat tout en bas à droite.

Pour calculer une case standard, on a besoin d'avoir déjà le résultat sur la case V[q] plus haut, et de la case juste à gauche.

DOnc, on peut remplir notre tableau de la première ligne à la dernière, en prenant les lignes de gauche a droite, ou bien de la première colonne à la dernière, en prenant les colonnes de gauche à droite.

On note que $H[q,n]$ se fait différement suivant qu'on a ou non $n < V[q]$. Si on le fait ligne après ligne, cela implique qu'avant de remplir chaque case du tableau, il faut faire un test $n < V[q]$. Si on le fait colonne après colonne, c'est bien plus simple à gérer, on le gère en manipulant les extrêmités les for.

Ce qui donne donc (on suppose que V[] est déjà rempli avec les valeurs des pièces. On généralise le code pour k pièces)

```
         for q := 0 to k do H[q,0] := 1 ;
      for m := 1 to n do H[0,m] := 0 ;
      for q := 1 to k do
           begin
           for m := 0 to V[q]-1 do H[m,q] := H[m,q-1] ;
           for m := V[q] to n do H[m,q] := H[m,q-1] + H[m-V[q],q] ;
           end ;
      rendre H[k,n]
end ;
```

L'algo est à présent en $\theta(n * k)$ au lieu de $\theta(n^k)$

On peut noter cependant que ça mange plus d'espace mémoire.

Complexité mémoire en $n * k$

Peut-on économiser sur l'espace mémoire ?

Oui : on observe que quand on calcule une case, on a encore besoin de la partie de la colonne au dessus de cette case, et de la partie de la colonne juste à gauche au dessous de cette case. Le reste à droite n'est pas encore calcule, le reste à gauche n'est plus utile et peut donc être oublié. Du coup, on peut n'utiliser qu'une colonne qui retient la partie utile, d'ou le code :

```
         H[0] := 1 ;
      for m := 1 to n do H[m] := 0 ;
      for q := 1 to k do
           begin
           for m := V[q] to n do H[m] := H[m] + H[m-V[q]] ;
           end ;
      rendre H[n]
end ;
```

La complexité mémoire est en n (complexité en temps inchangée)

On peut aussi voir que si on le fait ligne après ligne, on a besoin des V[q] cases au dessus de la case courante.

Donc il suffirait de garder en mémoire pour chaque colonne les V[q] derniers résultats.

Comment faire en pratique ? Faire un tableau H[0..k] de listes circulaires, la liste de H[q] ayant V[q] élément, celle de H[0] n'ayant qu'un élément.

Initialiser à 0 partout sauf sur l'élément $H[0]$ ou c'est 1.

Puis faire

```
Pour m de 1 à n
        pour q de k à 1 décroissant do
             H[q]->valeur := H[q]->valeur + H[q-1]->valeur
        pour q de 1 à k, avancer d'un cran dans la liste chaînée H[q]
rendre H[k]->valeur
```

L'espace mémoire en maintenant en $\sum V[q]$

Pour terminer, existe-il une formulaire toute faite pour nos 5 pièces 1 2 5 10 20 ?

Oui, mais elle est lourde à calculer.

La technique, c'est d'introduire pour un ensemble de pièces P la fonction

$f_P = \sum_0^\infty X_{P,n} t^n$

ou $X_{P,n}$ est le nombre de façons de décomposer n en utilisant uniquement des pièces prises dans P.

Si P est un singleton, une seule pièce de valeur v, on a que n se décompose ssi n est un multiple de v, et qu'alors il y a une seule décomposition.

$f_{\{v\}} = 1 + 0t + 0t^2 + .. + 0t^{v-1} + 1t^v + 0t^{v+1} + ... = 1 + t^v + t^{2v} + t^{3v} + t^{4v} + ... = 1/(1-t^v)$

puis on observe que si P et Q sont disjoints, alors décomposer n avec P et Q revient à choisir la somme m que l'on décompose avec P puis choisir une décomposition de m avec P et une décomposition de n-m avec Q, d'ou la formule

$X_{P+Q,n} = \sum_0^n X_{P,m} X_{Q,n-m}$

a droite, c'est le coefficient en n du produit des séries fP et fQ d'ou

$f_{P+Q} = f_P * f_Q$

en regroupant tout, on en déduit que

$f_{\{1,2,5,10,20\}} = \frac{1}{1-t}\frac{1}{1-t^2}\frac{1}{1-t^5}\frac{1}{1-t^{10}}\frac{1}{1-t^{20}}$

dont il ne reste plus qu'a trouver les coeff...

yaka (!!!) décomposer en éléments simples (horreur, 1 est racine 5eme, -1 racine triple, les racines 5eme de l'unité aussi, y a les racines 20e de l'unité. Berk !) et en déduire les coefficient.

On va d'abord avoir une formule avec des choses comme "$cos(2i\pi n/20)$"

donc même pas évident que c'est un entier.

EN fait, on aura des choses comme

$cos(2i\pi n/5) + cos(2*2i\pi n/5) + cos(3*2i\pi n/5) + cos(4*2i\pi n/5) + cos(5*2i\pi n/5)$ *qui vaut 5 si n est multiple de 5 et 0 sinon.*

Avec ce genre de considération, on peut tirer une formule sans cos (et manifestement entière) qui variera en fonction de la valeur modulo 20 de n....
