LYCEE CHARLEMAGNE
Jeudi 11 janvier
M.P.S.I.2

2023 2024
ITC

c =	21	35	16	12	5	25	31	10	Obj = 100
a =	0	1	1	0	0	1	1	0	réussie, stable
a =	1	0	1	1	1	1	0	0	pas réussie
a =	1	1	1	1	1	1	0	0	réussie, non stable
a =	1	1	0	1	0	0	1	1	réussie, stable

Exemple :

Le langage sera obligatoirement `Python`.
N'oubliez pas les spécifications comme `def truc(a, b) : #int x list of int -> float`

Question 1 : l'énoncé ne dit pas si la fonction `Python sum` est autorisée ou non.
Sachez toutefois que `sum(L)` si `L` est une liste de valeurs numériques est de complexité `len(L)`.

Question 2 : utilisez la question 1. C'est con à dire, mais le rapport du jury ricane parfois sur ceux qui recommencent tout.

Question 3 : est il besoin de tester toutes les sous-alliance de a ? Normalement un seul test suffira si vous choisissez le maillon faible.

Question 4 : et si déjà l'entreprise n-1 atteignait l'objectif toute seule ? Sinon ?

Question 5 : que faites vous quand vous posez une addition en binaire ?
Sinon, on peut imaginer un aller retour entre les listes et les entiers par justement le codage binaire.

Question 6 : `print` ou `return` ?

Question 7 : profitez de toutes vos procédures précédentes.
Et n'oubliez pas qu'il y a deux questions.

Question 7bis : Supplément spécial MPSI2 : donnez moi les six résultats de `imprimerStables([15, 20, 30, 40, 55, 75], 160)`

Question 8 : une seule boucle.

Question 9 : on fera une double boucle

> *Mettons en pratique pour comprendre l'exemple de la page 3 pour* `c = [15, 20, 30, 60, 75, 85]` *et* `Obj = 100`.
> *L'alliance croissante* `15+20+30+60` *dépasse 100. (est elle stable ?)*
> *On prend donc k = 3. La somme pour l'alliance {3} est donc 60.*
> *Le nouvel objectif Obj' est donc 40.*
> *Cette fois, on complète avec* `15+20+30` *et on prend* `k'=2`.
> *L'alliance partielle {k', k} a donc pour somme* `30+60`.

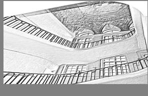
Question 9bis : Appliquez l'algorithme de l'énoncé pour `c = [15, 20, 30, 60, 75, 85]` et `Obj = 170`.

Question 10 :

Question 11 :

LYCEE CHARLEMAGNE
Jeudi 11 janvier
M.P.S.I.2

2023 2024
ITC
CORRECTION



Procédure **somme**(**c**, **a**) ITC

On crée un accumulateur `s`, on parcourt la liste `a`, et si le booléen `a[k]` est à `True`, on ajoute le terme `c[k]` dans la somme.

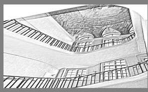
```
def somme(c, a) : #list of int x list of boolean -> int
....#calcule la somme des chiffres d'affaire d'une alliance
....#implicite : la liste a est de longueur n
....s = 0
....for i in range(n) :
........if a[i] :
............s += c[i]
....return s
```

On peut aussi extraire une liste `alliance`, puis sommer.

```
def somme(c, a) : #list of int x list of boolean -> int
....alliance = 0
....for i in range(n) :
........if a[i] :
............alliance.append(c[i])
....return sum(alliance)
```

On peut aussi jouer au one-liner sachant que les booléens sont assimilables à des entiers valant `0` ou `1` :

```
def somme(c, a) : #list of int x list of boolean -> int
....return sum(a[k]*c[k] for k in range(n))
```



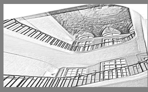
Procédure **estReussie**(**c**, **a**, **Obj**) ITC

On calcule la somme (la procédure précédente le fait) puis on teste.

```
def estReussie(c, a, Obj) : #list of int x list of boolean -> boolean
....#vérifie si l'alliance a un capital superieur a l'objectif
....return somme(c, a) >= Obj
```

Et si on ne fait pas de l'informatique, mais du truc lourd :

```
def estReussie(c, a, Obj) : #list of int x list of boolean -> boolean
....#vérifie si l'alliance a un capital superieur a l'objectif
....if somme(c, a) >= Obj :
........return True
....else :
........return False
```



Procédure **estStable**(**c**, **a**, **Obj**) ITC

Il faudra bien sûr déjà que l'alliance soit réussie.

Mais ensuite effectivement, la question est « si une entreprise part, l'alliance est elle encore réussie ».
En effet, si elle échoue avec le départ d'une, elle échouera avec le départ de plusieurs.
Et l'entreprise qui fait échouer est la plus petite en chiffre d'affaire.

Bref, pour que l'alliance soit stable il faut et il suffit que la liste a réussisse et que la liste a privée de sa « plus petite entreprise »

En effet, notons A l'alliance et A' l'alliance privée de sa plus petit entreprise.
La condition « alliance stable » est $reussie(A)$ *et* $\forall B \subsetneq A,\ \overline{reussie(B)}$
La condition que je propose est $reussie(A)$ *et* $\overline{reussie(A')}$.

Sens direct : si $\forall B \subsetneq A,\ \overline{reussie(B)}$ *alors en particulier* $A' \subsetneq A,\ \overline{reussie(A')}$.

Sens indirect : si $\overline{reussie(A')}$, *alors pour toute partie*
A terminer.

Dans tous les cas, il faut déjà parcourir la liste une fois pour voir si l'alliance est réussie.
On en profite pour déterminer durant ce parcours quelle est l'entreprise de chiffre d'affaire minimal.
Et on retourne le résultat des deux tests : total suffisant
et total moins petite entreprise insuffisant

```
def estStable(c, a, Obj) : #list of int x list of boolean x int -> boolean
....s, mini = 0, float('inf')
....for i in range(n) :
........if a[i] :
............s += c[i]
............if c[i] < mini :
................mini = c[i]
....return (s >= Obj and s-mini < Obj)
```

On peut aussi proposer de tester ce qu'il advient à chaque fois qu'on enlève une entreprise.
Si la sous-alliance est encore réussie, alors l'alliance initiale n'était pas stable.
Si avec les Card(A) essais on n'est pas sorti par False, alors c'est que tout est bon.

```
def estStable(c, a, Obj) :
....s = somme(c,a)
....if s < Obj :
........return False
....for k in range(n) :
........if a[k] and s-c[k] >= Obj :
............return False
....return True
```

On peut aussi reprendre l'idée de la liste des valeurs de l'alliance :

```
def EstStable(c, a, Obj) :#list of int x list of boolean -> boolean
....#vérifie si l'alliance a un capital superieur a l'objectif
....#puis si en enlevant la plus petite entreprise elle ne l'atteint plus
....alliance = [ ]
....for k in range(n) :
........if a[k] :
............alliance.append(c[k])
....s = sum(alliance)
....return s>= Obj and s-min(s) < Obj
```

L'idée est de prendre l'entreprise la plus riche, ou plus généralement les entreprises les plus riches ; et de les réunir jusqu'à ce que le chiffre d'affaire atteint soit supérieur ou égal à l'objectif.

```
def allianceMin(c, Obj) : #list of int x int -> list of int
....#implicite la liste c'est triée par ordre croissant
....#et la somme de toutes les entreprises depasse l'objectif
....a = [0 for k in range(n)]
....s, i = 0, n-1
....while s < Obj :
........s += c[i]
........a[i] = 1
........i -= 1
....return a
```

Et si finalement on lit que la demande est d'imprimer

```
def allianceMin(c, Obj) : #list of int x int -> None
....#implicite : la liste c'est triée par ordre croissant
....#et la somme de toutes les entreprises depasse l'objectif
....s, i = 0, n-1
....while s < Obj :
........s += c[i]
........print(i)
........i -= 1
```

Procédure **suivantDe**(**a**) ITC

Comment passer d'un entier au suivant ? Comment pose-t-on l'addition ?

Attention, dans ce qui suit, le chiffre des unités est à gauche.

	1	2	4	8	16
a = 18	0	1	0	0	1
a+1 = 19	1	1	0	0	1

	1	2	4	8	16
a = 19	1	1	0	0	1
a+1 = 20	0	0	1	0	1

	1	2	4	8	16
a = 5	1	0	1	0	0
a+1 = 6	0	1	1	0	0

	1	2	4	8	16
a = 23	1	1	1	0	1
a+1 = 19	0	0	0	1	1

	1	2	4	8	16
a = 6	0	1	1	0	0
a+1 = 7	1	1	1	0	0

	1	2	4	8	16
a = 31	1	1	1	1	1
a+1 = ?	0	0	0	0	0

Si le chiffre des unités est un 0 on le remplace par un 1.

Sinon, il y a une retenue et la question reprend avec le suivant.

En fait, tant que les chiffres sont des 1 ils passent à 0, et une fois qu'on arrive sur un 0, il devient un 1.
Avec le « tant que », on a une boucle `while`.
Et un index qui augmente de 1 à chaque fois.

```
while a[i] == 1 :
....a[i] = 0
....i += 1
```

Et il faut ensuite mettre à `1` le `a[i]` sur lequel on est arrivé.
Et on ne touche pas aux suivants.

Mais il faut voir aussi si on n'est pas arrivé bêtement tout au bout de notre liste.

```
def suivantDe(a) : #list of boolean -> boolean
....#modifie la liste a en passant au binaire suivant et retourne le booléen de depassement de capacite
....#list of boolean est modifie par la procedure
....i = 0
....while i < n and a[i] == 1 :
........a[i] = 0
........i += 1
....if i == n :
........return False
....a[i] = 1
....return True
```

Pour ceux qui préfèrent :

```
def binaireVersEntier(L) : #list of boolean -> int
....#convertit une chaine binaire en entier
....#implicite : la liste L est de longueur n
....p, d = 0, 1
....for k in range(n) :
........p += L[k]*d
........d *= 2
....return p
```

Solution de l'ingénieur non informaticien

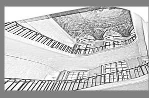
```
def binaireVersEntier(L) : #list of boolean -> int
....p = 0
....for k in range(n) :
........p += L[k]*2**k
....return p
```

```
def binaireVersEntier(L) : #list of boolean -> int
....return sum([L[k]*2**k for k in range(n)])
```

Et dans l'autre sens, un classique :

```
def entierVersBinaire(p) : #int -> list of boolean
....#convertit un entier en chaine binaire
....#implicite : p est plus petit que 2**n
....L = [ ]
....for k in range(n) :
........L.append(p%2)
........p //= 2
....return(L)
```

Il ne reste plus qu'à utiliser `entierVersBinaire(BinaitreVersEntier(a)+1)`



Procédure **imprimer**(**a**) ITC

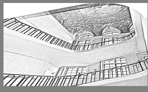
On va créer un mot. On y mettra les chiffres correspondant aux index où il y a un 1 (ou un `True`).
On pense aussi aux espaces (car 145 signifie-il que c'est 145 ou 1 puis 4 puis 5 ou 1 puis 45 ?).

```
def imprimer(a) : #list of boolean -> string
....#crée la chaine de caractere des entreprises de l'alliance
....mot = ""
....for i in range(n) :
........if a[i] :
............mot += str(i)+" "
....return mot
```

On pourra même conclure avec return `mot[ :-1]` pour ne pas afficher le dernier espace.

La question étant « imprimer », on pourra sortir avec un `print(mot)` au lieu d'un `return mot`.
Mais je préfère le `return` et je mets dans mon programme `print(imprimer(a))`.

Une solution en une ligne est sans nul doute possible.



Procédure **imprimerStables**(**c**, **Obj**) ITC

On va utiliser tout ce qui précède.
On crée une liste **a** de longueur n qui va représenter nos entiers de 0 à 2^n. On lui donne déjà la valeur $[0,\ 0,\ \ldots 0]$.

Ensuite tant qu'on n'est pas allé au bout, on passe au a suivant d'où un `while(suivantDe(a))`. (on rappelle que suivantDe(a) modifie a comme demandé, et retourne un booléen, ce qui est parfait pour notre boucle conditionnelle).

Et à chaque fois, on teste si a est une alliance stable : `if estStable(a)`.
Et dans ce cas, on affiche.

A la fin, on ne retourne rien. On a juste affiché des choses, comme l'indique le nom.

```
def imprimerStables(c, Obj) : #list of int x int -> None
....#print les alliances stables parmi les 2**n alliances possibles
....a = [0 for k in range(n)]
....while suivantDe(a) :
........if estStable(a) :
............imprime(a)
```

Je préfère pour ma part :

```
def imprimerStables(c, Obj) : #list of int x int -> list of string
....listeStables = [ ]
....a = [0 for k in range(n)]
....while suivantDe(a) :
........if estStable(a) :
............listeStables.append(imprime(a))
....return listeStables
```

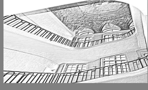
où `imprime(a)` est la procédure qui retourne un mot. La procédure retourne une liste de `string`, et je n'ai plus qu'à l'exploiter.

Mesure de la complexité.
Notre boucle `while` (en fait, on pouvait faire une boucle for et utiliser le convertisseur `entierVersBinaire` défini plus haut) va être exécutée 2^n fois.
Ensuite, le test `estStable(a)` est en $O(n)$.
De même l'affichage.

La complexité est $O(n.2^n)$. De quoi stopper l'élan de n'importe quel informaticien.

Exemple choisi à traiter à la main.
La liste `[15, 20, 30, 40, 55, 75]` a pour somme `235`. On sait qu'on peut atteindre `160`.

imprime(a)	0	1	2	3	4	5	val(a)	calcul						somme
'0 1 2 3 4 '	1	1	1	1	1	0	31	15	+20	+30	+40	+55		= 160
'0 2 3 5 '	1	0	1	1	0	1	45	15		+30	+40		+75	= 160
'1 2 3 5 '	0	1	1	1	0	1	46		20	+30	+40		+75	= 165
'0 1 4 5 '	1	1	0	0	1	1	51	15	+20			+55	+75	= 165
'2 4 5 '	0	0	1	0	1	1	52			30		+55	+75	= 160
'3 4 5 '	0	0	0	1	1	1	56				40	+55	+75	= 170



Procédure **cumul**(**c**, **t**) ITC

`c` est donné, et on détermine `t`, mais on ne le retourne pas. Pourquoi pas. On estime qu'on nous l'a donné et qu'on doit le modifier.

On peut certes à chaque valeur de k calculer $\sum_{i=0}^{k-1} c[k]$, mais ce serait idiot.

Le one line `[sum[c[i] for i in range(k)] for k in range(n)]` *est certes joli, mais calcule par exemple* `c[0]+c[1]+c[2]` *à un moment, puis à l'étape suivante calcule* `c[0]+c[1]+c[2]+c[3]` *au lieu de juste ajouter* `c[3]`*.*

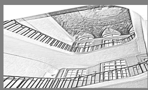
C'est pourquoi on nous demande linéaire.

```
def cumuls(c, t) : #list of int x list of int -> None
....#implicite  : les deux listes sont de longueur n
....t[0] = c[0]
....for k in range(1, n) :
........t[k] = t[k-1]+c[k]
```

C'est la formule définissant les séries en mathématiques :
a_n *terme général*
A_n *somme partielle*
$A_n = \sum_{i=0}^{n} a_i$
$A_{n+1} = A_{n-1} + a_n$
$a_n = A_n - A_{n-1}$

On utilisera le tableau t pour trouver la première alliance stable de petites entreprises

```
i = 0
while t[i] < Obj :
....i += 1
```

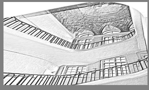


Algorithme pour c = [15, 20, 30, 60, 75, 85] et `Obj = 170.` ITC

	0	1	2	3	4	5
c	15	20	30	60	75	85
t	15	35	65	125	200	285

On veut atteindre `170`.
On part d'une alliance vide.

	nouvelle alliance	total	Obj' (ce qu'il manque)
C'est avec `15+20+30+60+75` qu'on dépasse 170.	{4}	75	95
C'est avec `15+20+30+60` qu'on dépasse 95.	{3, 4}	135	35
C'est avec `15+20` qu'on dépasse 35.	{0, 1, 3, 4}	170	0



Procédure **completer**(**t**, **c**, **a**, **Obj**, **k**)) ITC

En implicite, `k` est l'entreprise minimale appartenant à `a`.
Donc, les entreprises de `0` à `k-1` ne font pas partie de l'alliance `a`.
Les chiffres du début de `a` sont des `0`. Exemple `a = [0, 0, 0, 0, 1, 0, 1, 1, 0]`.
On peut être tenté de faire intervenir `suivant(a)`.

Mais comme en fait on va ajouter une seule entreprise, ce serait un peu idiot.

```
def completer(t, c, a, Obj, k) :
....Objprime = Obj - c[k]
....kprime = 0
....while t[kprime] < Obj :
........kprime += 1
....a[kprime] = 1
....Obj = Objprime - c[kprime]
```

Attention, dans l'exécution, les listes `a`, `c` et autres sont modifiées. Mais l'entier `Obj` ne l'est pas.

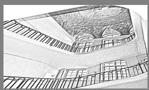
> *Les listes sont passés par adresse donc modifiées.*
> *Les entiers sont passés par valeur, donc non modifiés.*

De plus, cette procédure n'agit qu'une fois.
L'énoncé nous demande de calculer « quelques k' comme indiqués ».
On peut mettre en boucle ou solliciter de manière récursive.

```
def completer(t, c, a, Obj, k) : #list of int, list of int, list of int, int, int -> int
....#implicite : k < n et t(k) >= Obj
....while Obj > 0 :
........kprime = 0
........while t[kprime] < Obj :
............kprime += 1
........a [kprime] = 1
........Obj -= c[kprime]
....return kprime
```

A la fin, `Obj` est négatif, car on déborde un peu.

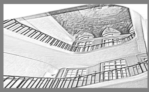
> *L'énoncé parle à chaque fois d'alliance stable. Mais par construction, elles le sont.*



Procédure **prelude**(**t**, **c**, **a**, **Obj**, **i**)) ITC

Comme graphiquement, les `i`, `l` et `1` se ressemble, la lettre `l` (ou plutôt $\mathscr{L}$) de l'énoncé sera appelée `elle`.
Je plains les correcteurs du sujet original.

```
def prelude(t, c, a, Obj, i) :
....elle = 0
....while i+elle+1 < n and a[i+elle+1] : #recherche de elle, sans déborder
........elle += 1
....k = i+elle+1
....if k == n : #plus d'autre alliance
........return n, 0
....else :
........for j in range(i, k) :
............if a[j] :
................Obj = Obj + c[j]
................a[j] = 0
........a[k] = 1 #on intègre k à l'alliance
........Obj -= c[k]
........return k , Obj
```



Procédure **imprimerAStables**(**c**, **Obj**))

ITC

```
def imprimerAStables1(c, Obj) : #list of int x int -> None
....t = [0 for k in range(n)]
....cumuls(c, t)
....a = [0 for k in range(n)]
....if t [-1] < Obj :
........print ('Pas assez de capital disponible')
....k = 0
....while t[k] < Obj :
.........k += 1
....a[k] = 1
....Obj -= c[k]
....i = completer(t, c, a, Obj, k)
....imprimer(a)
....while k < n :
........k , Obj = prelude(t, c, a, Obj, i)
........if k < n :
............i, Obj = completer(t, c, a, Obj, k)
............imprimer ( a )
```