Le devoir est traité sur le sujet fourni. N'oubliez pas d'indiquer vos noms et prénoms dès le début de l'épreuve. En cas d'erreur ou de manque de place, vous pouvez utilisez une feuille **double** complémentaire et indiquer sur le sujet les questions traitées sur la feuille complémentaire.

Exercice 1

Q 1). *Donner* ***le résultat et le type du résultat*** *des instructions suivantes.(Dans le cas où la ligne de commande comporte plusieurs instructions séparées par un ";", il s'agit du résultat et du type du résultat de la dernière instructon*

(a) `1+1.5`

(b) `l=[1,2,3];l.append(4);l`

(c) `s2="abc";s1="def";s1+s2`

(d) `s2="abc";s1="def";s1+"s2"`

Q 2). *Quelle est la taille mémoire en octets nécessaire pour stocker l'entier* (100!).

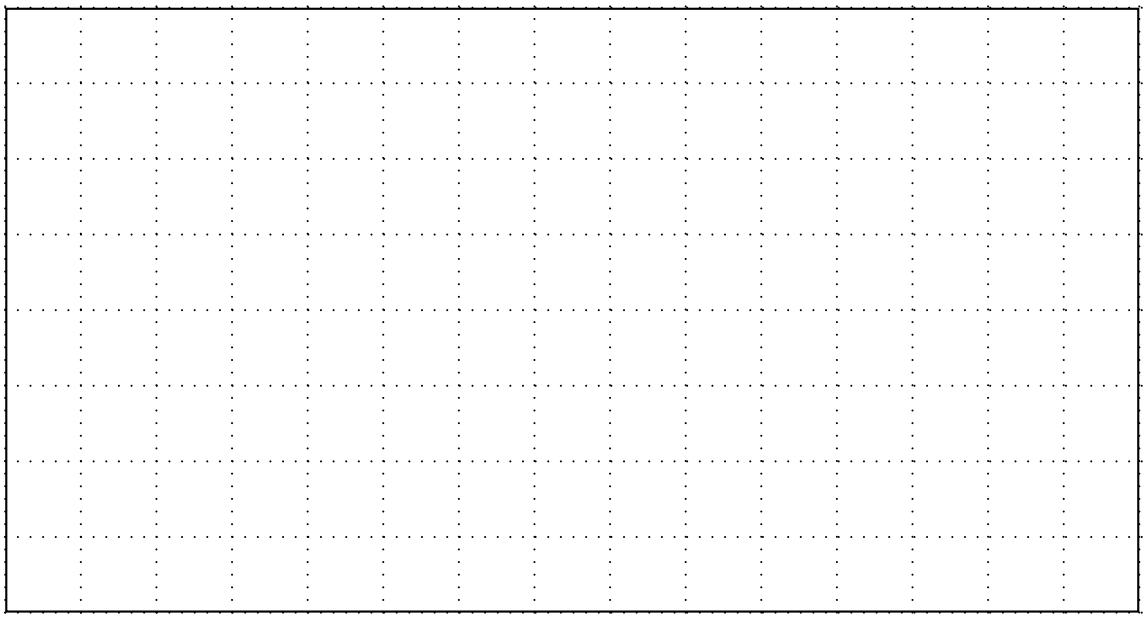
Q 3). *On considère le code suivant :*

```
1 def zero(n,p):
2     return [[0]*p]*n
3 a=zero(3,2)
4 print(a)
5 a[0][0]=1
6 print(a)
```

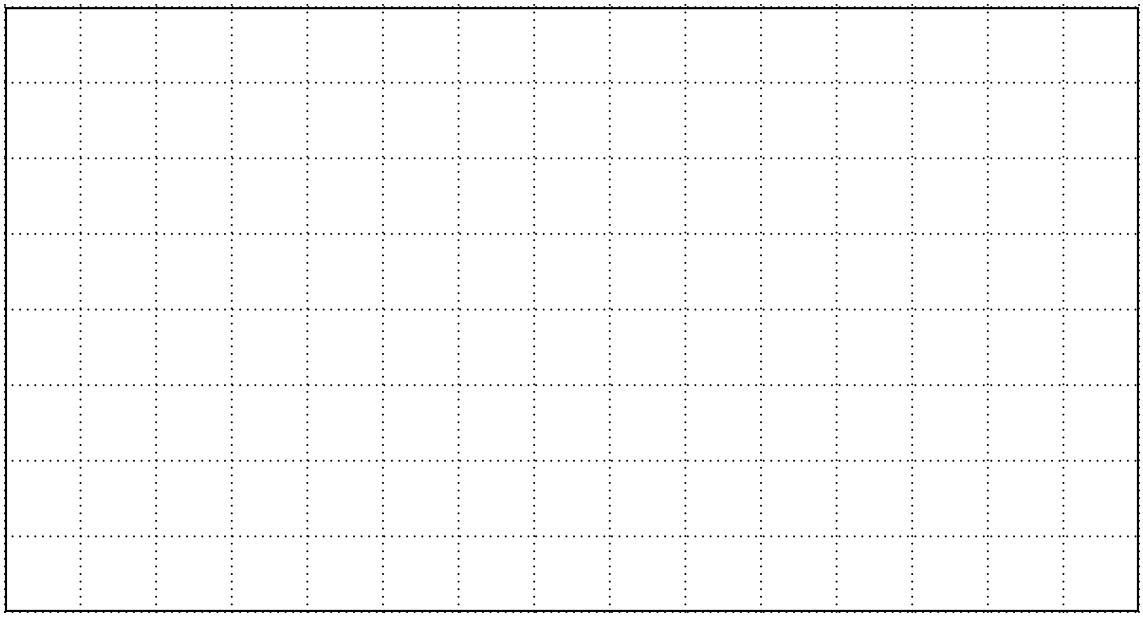
La fonction zero définie ligne 1 *a pour but de d'initialiser un tableau à deux dimensions (représenté par une liste de listes) avec des "0".*
L'instruction de la ligne 4 *ci-dessous affiche bien le résultat attendu* `[[0, 0], [0, 0], [0, 0]]`
L'instruction de la ligne 6 *affiche le résultat suivant :* `[[1, 0], [1, 0], [1, 0]]`, *au lieu du résultat attendu :* `[[1, 0], [0, 0], [0, 0]]`

(a) Expliquez pourquoi

(b) Donner une version correcte de la fonction ***zero***



Exercice 2 *Ecrire une fonction* ***binom*** *prenant en paramètre deux entiers positifs n et p tels que $p \leq n$, et renvoyant $\binom{n}{p}$. On n'utilisera que les opérateurs $*$, et $//$. On impose une complexité linéaire en p, c'est à dire telle que le nombre d'opérations pour calculer* ***binom(n,p)*** *est majoré par Kp où K est une constante. Le coût d'une multiplication et d'une division seront considérés comme constants. Attention la fonction* ***binom*** *doit renvoyer un entier.*



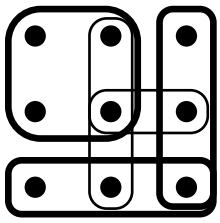
Exercice 3 *Cet exercice constitue un problème complet d'algorithmique. La partie III n'est à aborder que si tout le reste est terminé.*

Couverture optimale

Ce sujet a pour motivation principale le problème d'allocation de ressources, abordé explicitement dans la Partie III. L'approche choisie consiste ici à considérer ce problème comme une instance de celui de la couverture optimale d'ensemble.

Le problème s'énonce alors comme suit. Soit U un ensemble fini. Étant donnée une famille F constituée d'ensembles inclus dans U telle que F couvre U (c'est-à-dire que chaque élément de U

appartient à au moins un ensemble de F), de combien d'ensembles de F a-t-on besoin, au minimum, pour couvrir U ? Dans l'exemple ci-dessous, l'ensemble U est constitue de 9 points. La famille F contient 5 ensembles et recouvre U. Aucune sous-famille de F constituée d'au plus 2 ensembles n'est couvrante. En revanche, la sous-famille constituée des 3 ensembles en traits épais sur le dessin est couvrante. Une telle famille est dite optimale.



L'utilisation de fonctions puissance et logarithme prédéfinies en python est interdit

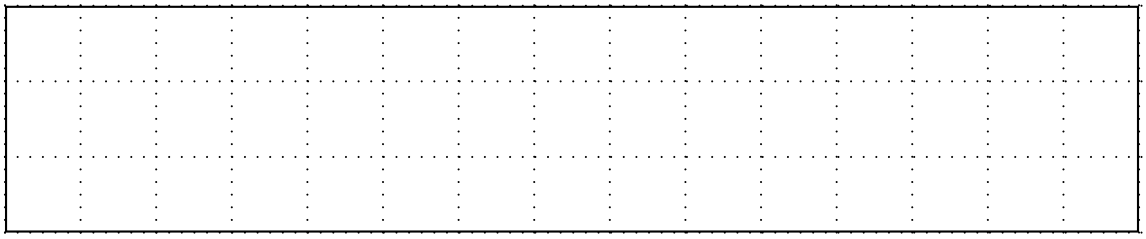
La complexité, ou le temps d'exécution, d'un programme P (fonction ou procédure) est le nombre d'opérations élémentaires (addition, soustraction, multiplication, division, affectation, etc...) nécessaires à l'exécution de P. Lorsque cette complexité dépend d'un paramètre n, on dira que P a une complexité en $O(f(n))$, s'il existe $K > 0$ tel que la complexité de P est au plus $K\,f(n)$, pour tout n. Lorsqu'il est demandé de garantir une certaine complexité, le candidat devra justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

Partie I. Autour des ensembles

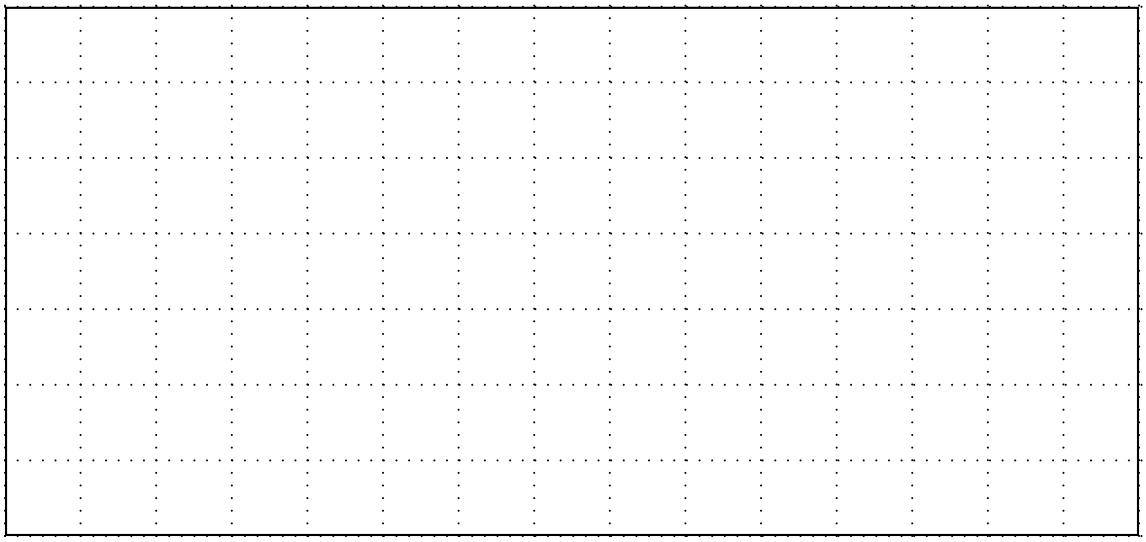
Dans cette partie, on représente les ensembles finis d'entiers positifs ou nuls par des listes de booléens. L'ensemble S correspondant au liste t contient exactement les entiers i tels que $t[i]$ est égal à `True`*; S ne contient pas les entiers i tels que est égal à* `False`*, ou tel que $t[i]$ n'est pas défini. Ainsi, le liste [*`False`*,* `True`*,* `True`*,* `False`*,* `True`*] représente l'ensemble $\{1, 2, 4\}$. On remarque qu'un même ensemble S peut être représenté de multiples façons, en jouant sur la longueur du liste. Par exemple [*`False`*,* `True`*,* `True`*,* `False`*,* `True`*,* `False`*] et [*`False`*,* `True`*,* `True`*,* `False`*,* `True`*,* `False`*,* `False`*,* `False`*,* `False`*] sont d'autres représentations de l'ensemble $\{1, 2, 4\}$.*

Q 4). *Écrire une fonction* `cardinal`*(t), qui prend en argument une liste t de booléens représentant l'ensemble S, et qui renvoie le cardinal de l'ensemble S .*

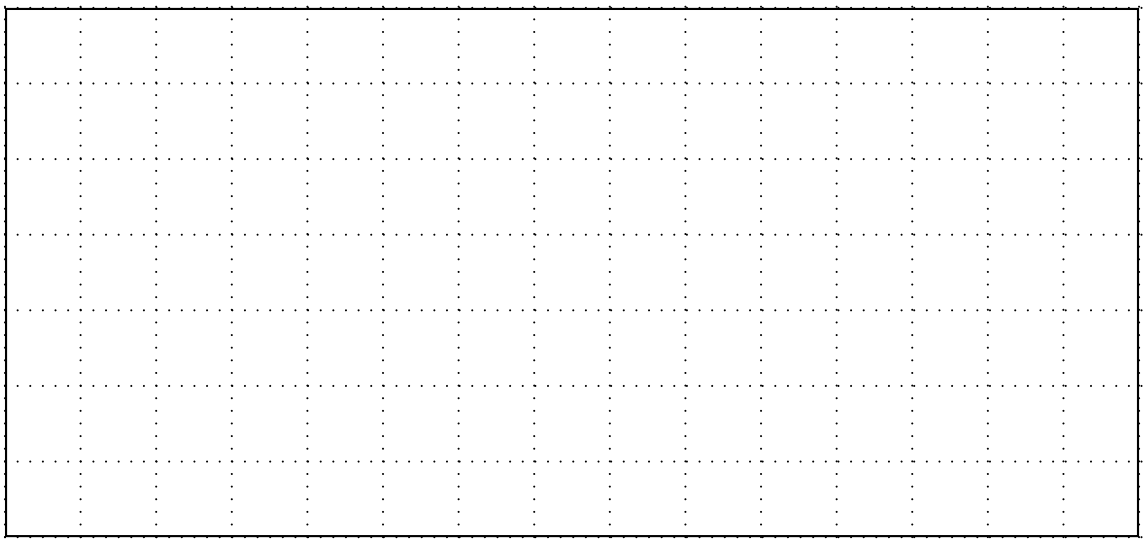
Q 5). *Écrire une fonction* `appartient`*(i, t), qui prend en argument un entier i et une liste de booléens t représentant l'ensemble S, et qui renvoie* `True` *si i appartient à S et* `False` *sinon.*



Q 6). *Écrire une fonction* `diff`(t_1, t_2)*, qui prend en argument deux listes de booléens* t_1 *et* t_2 *représentant les ensembles* S_1 *et* S_2*, et qui renvoie une liste représentant la différence ensembliste* $S_1 \setminus S_2$*. Ici* $S_1 \setminus S_2$ *désigne l'ensemble constitué des éléments de* S_1 *qui ne sont pas dans* S_2*.*



Q 7). *Écrire une fonction* `union`(t_1, t_2)*, qui prend en argument deux listes de booléens* t_1 *et* t_2 *représentant les ensembles* S_1 *et* S_2*, et qui renvoie une liste représentant l'union de* S_1 *et* S_2*.*



Partie II. Représentation par entiers

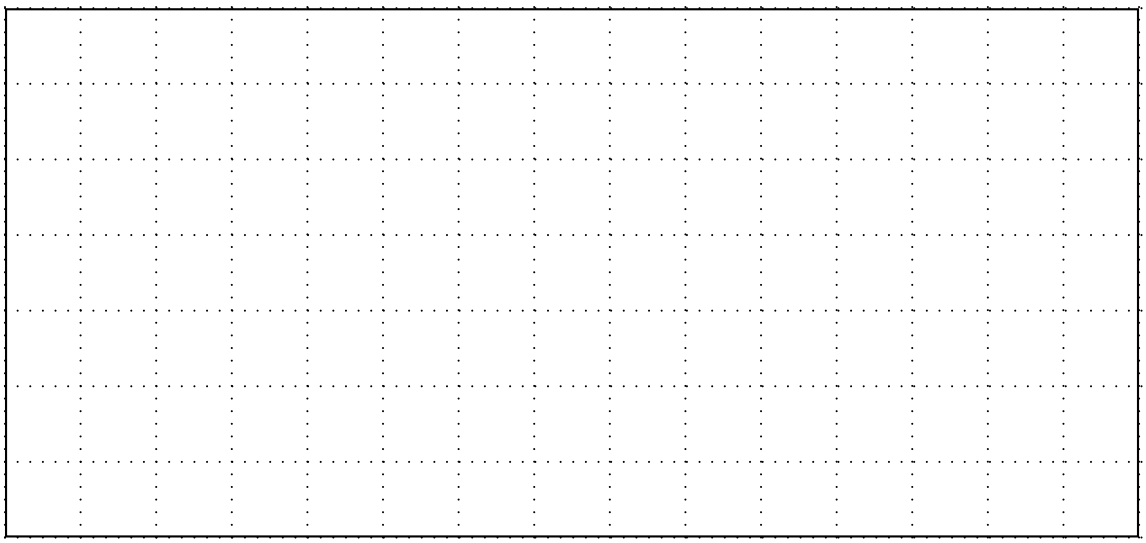
Afin de pouvoir manipuler simplement des familles d'ensembles, on souhaite maintenant représenter les ensembles par des entiers. Pour ce faire, on associe à un ensemble S *(d'entiers positifs ou nuls) l'entier* $s = \sum_{i \in S} 2^i$*. Par exemple, l'ensemble* $\{1, 2, 4\}$ *correspond à l'entier*

$2^1 + 2^2 + 2^4$, *soit 22. Contrairement à la représentation par liste de booléens, cette représentation est unique. On prendra garde à ne pas confondre les entiers représentant des ensembles avec les entiers contenus dans ces ensembles.*

Q 8). *Si* $U = \{0, 1, 2, ..., n-1\}$, *quel est le plus grand entier représentant un sous-ensemble de* U *? Quel est le plus petit ?*

Q 9). *Écrire une fonction* `set2int`*(t) qui prend en argument une liste* t *de booléens représentant un ensemble* S, *qui renvoie l'entier* s *représentant* S, *et dont le temps d'exécution est linéaire en la taille de* t.

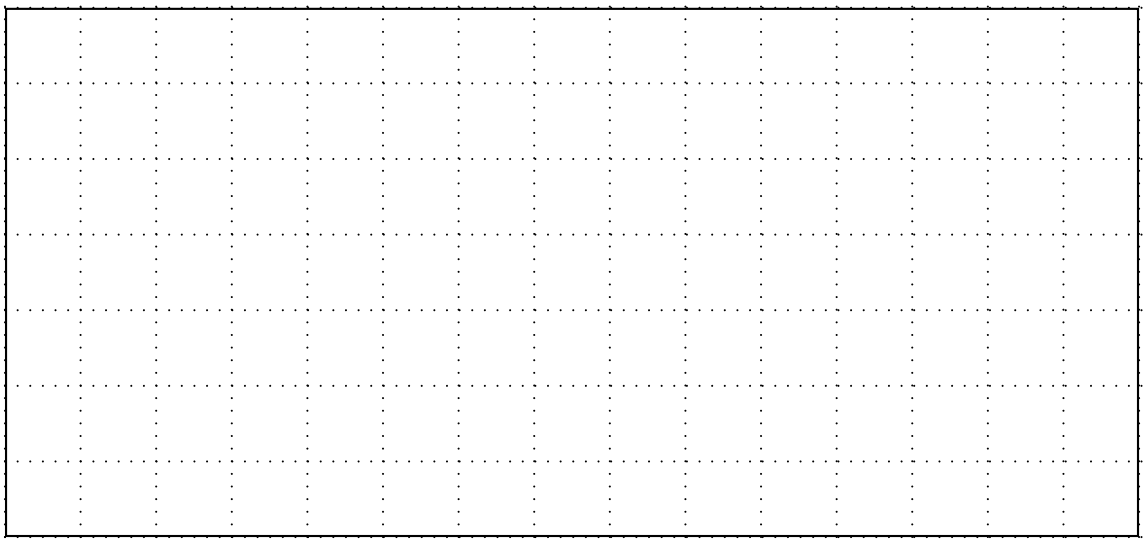
On prendra soin de s'assurer et de justifier brièvement que le temps d'exécution de la fonction est linéaire en la taille de t. *On rappelle que* ***l'utilisation de fonctions puissance et logarithme prédéfinies dans le langage est interdit****.*



Q 10). *Écrire une fonction* `int2set`*(s) qui prend en argument un entier* s *représentant un ensemble* S, *et qui renvoie une liste représentant* S. *En estimer le temps d'exécution. On prendra soin d'expliquer de quelle manière on choisit un vecteur particulier parmi les multiples représentations possibles.*

Grâce aux deux fonctions ci-dessus, il est possible d'étendre les fonctions de la partie précédente aux entiers représentant des ensembles.

Dans le reste du sujet, on supposera que toutes les fonctions de la partie précédente ont été réécrite afin de pouvoir opérer directement sur les entiers (arguments et résultat).



Partie III. Familles, sous-familles, et couvertures

On aborde maintenant le problème de couverture optimale, défini en préambule du sujet.

Q 11). *Une école souhaite proposer des activités sportives à ses élèves. Il y a plusieurs activités possibles, et chaque élève est invité à dire lesquelles lui conviennent (en en choisissant au moins une). L'école souhaite minimiser le nombre d'activités à organiser, tout en garantissant que chaque élève puisse suivre une activité qui lui convient. Montrer comment ce problème peut être vu comme un problème de couverture optimale.*

On peut représenter une famille finie $F = (F_0, F_1, ...)$ (sous-entendue ordonnée) constituée d'ensembles finis d'entiers (positifs ou nuls) par une liste f, dont chaque case f_ℓ est l'entier qui représente l'ensemble F_ℓ au sens de la partie II. Par exemple, la liste $[17, 3, 8, 22]$ représente la famille $(\{0, 4\}, \{0, 1\}, \{3\}, \{1, 2, 4\})$, puisque $2^0+2^4 = 17$, $2^0+2^1 = 3$, $2^3 = 8$ et $2^1+2^2+2^4 = 22$.

On fixe désormais $U = \{0, 1, 2, ..., n-1\}$ et une famille F d'ensembles inclus dans U telle que F couvre l'ensemble U. Dans tout le reste du sujet, on suppose que F est donnée par une liste f d'entiers.

Une sous-famille G de F est constituée de certains des ensembles de la famille F, et est donc de la forme $(F_{\ell_0}, F_{\ell_1}, ...)$, où $\ell_0, \ell_1, ...$ est une sous-suite finie strictement croissante des indices de f. Par conséquent G est définie par l'ensemble des indices $\{\ell_0, \ell_1, ...\}$. Le cardinal de G est donc le cardinal de l'ensemble $\{\ell_0, \ell_1, ...\}$. Au sens de la partie II, cet ensemble d'indices peut une nouvelle fois être représenté par un entier. Par exemple, si F est la famille $(\{0, 4\}, \{0, 1\}, \{3\}, \{1, 2, 4\})$, la sous-famille $G = (\{0, 4\}, \{3\})$, qui est constituée des ensembles F_0 et F_2, est de cardinal 2 et est représentée par l'entier 5, puisque $2^0 + 2^2 = 5$.

Noter que désormais trois objets distincts sont représentés par des entiers.

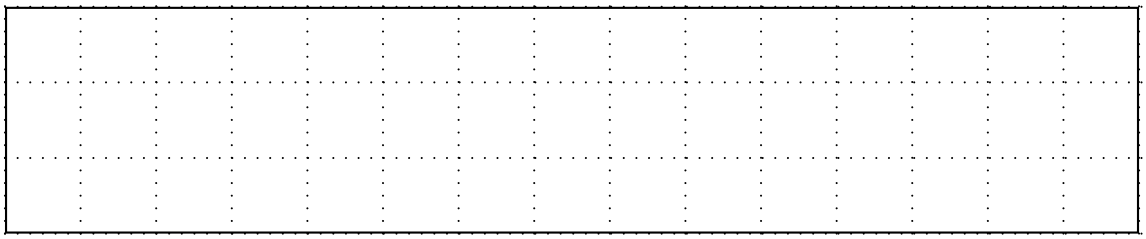
- *Les éléments de $U = \{0, 1, 2, ..., n-1\}$: dans ce cas la lettre i sera de préférence utilisée, et il n'y a pas de codage particulier, i représente l'entier i ;*

- *Les sous-ensembles de U : dans ce cas la lettre s sera de préférence utilisée, et s représente l'ensemble $S =$ `int2set`(s) ;*
- *Les sous-familles de F : dans ce cas la lettre g sera de préférence utilisée, et g représente la sous-famille définie par les ensembles F_ℓ pour $\ell \in$ `int2set`(g). De plus, chaque ensemble F_ℓ est lui-même représenté par l'entier $f[\ell]$.*

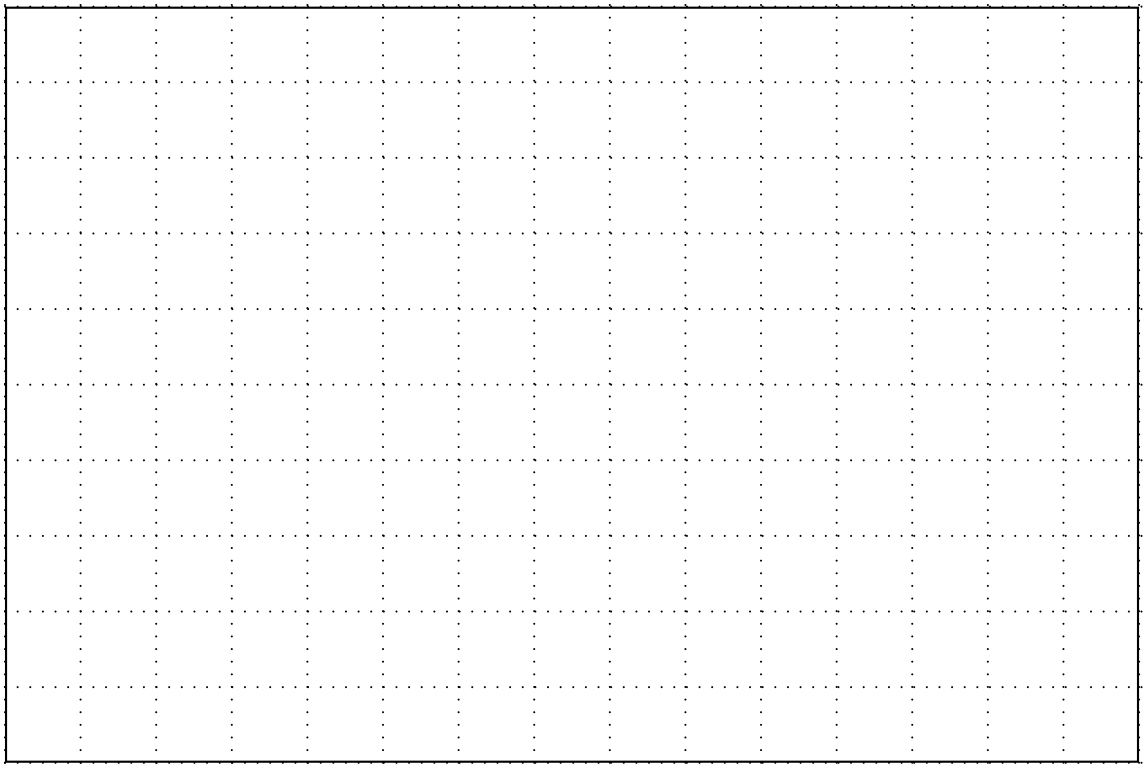
Une première façon de trouver une sous-famille couvrante de petit cardinal consiste à utiliser un algorithme dit "glouton". L'idée est de construire une sous-famille étape par étape, en gardant en mémoire l'ensemble des éléments déjà couverts, et en ajoutant à chaque fois l'ensemble qui permet de couvrir le plus de nouveaux éléments.

***Q 12)**. Proposer un exemple de famille où l'algorithme glouton ne renvoie pas une sous-famille couvrante optimale.*

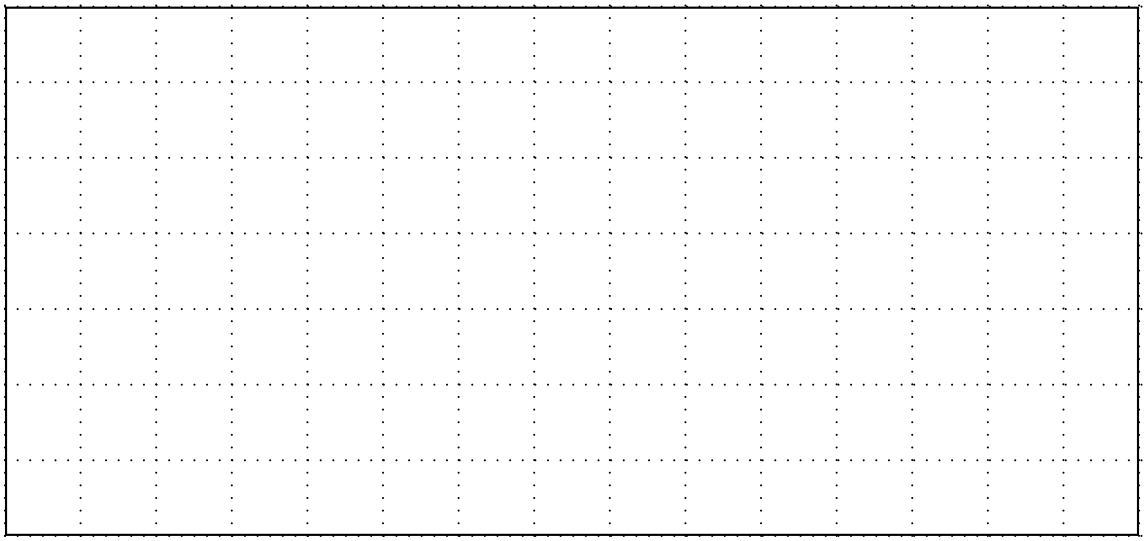
***Q 13)**. Écrire une fonction `reste`(s_1, s_2), qui prend en argument deux entiers s_1 et s_2 représentant les sous-ensembles S_1 et S_2 de $U = \{0, 1, ..., n-1\}$, et qui renvoie le nombre d'éléments de S_1 qui ne sont pas dans S_2.*



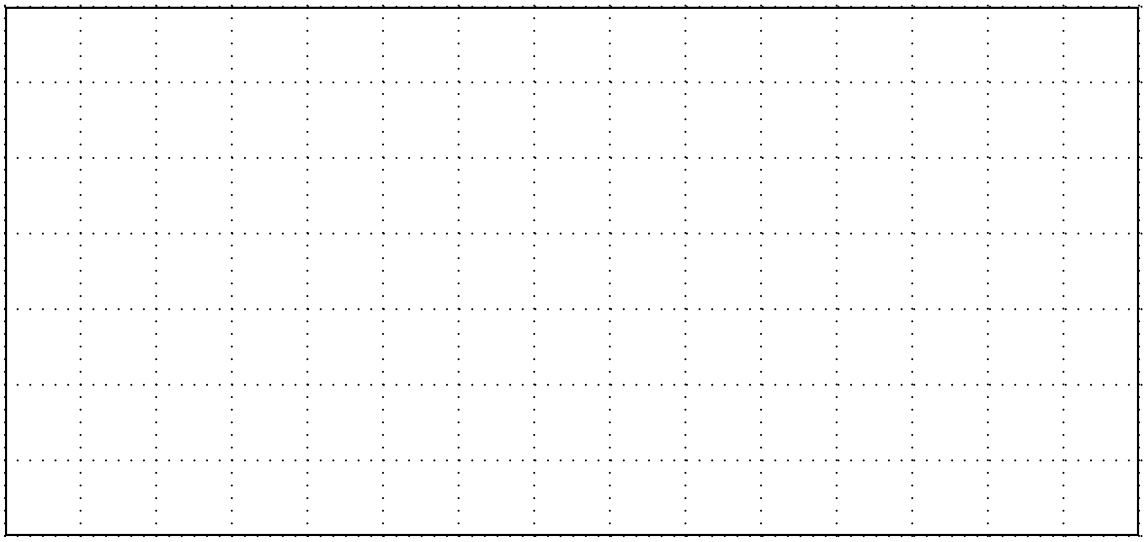
***Q 14)**. Écrire une fonction `glouton`(n,f), qui renvoie un entier g représentant une sous-famille couvrante G de F grâce à l'algorithme glouton décrit ci-dessus. Estimer son temps d'exécution.*



Q 15). *Écrire une fonction* `couverture`*(g,f,n), qui prend en argument un entier g représentant une sous-famille G de F et qui renvoie* `True` *si G est couvrante et* `False` *sinon.*



Q 16). *Écrire une fonction* `optimale`*(f,n) qui parcourt toutes les sous-familles possibles afin de renvoyer une sous-famille couvrante de F optimale. Estimer son temps d'exécution.*



Exercice 3

Q 4). Il s'agit de compter le nombre de `vrai` présents dans le liste. On parcourt la liste dans la boucle des lignes 3 à 5 et on incrémente le compteur `cmpt` si i appartient à l'ensemble représenté par la liste t (ligne 5).

```
1  def cardinal(t):
2      cmpt = 0;
3      for bool in t:
4          if bool:
5              cmpt +=1
6      return(cmpt)
```

Q 5). Si i est plus grand que la taille de la liste t alors i n'appartient pas à l'ensemble représenté par la liste t (ligne 5). Sinon, il suffit de retourner la valeur $t[i]$ (ligne 3).

```
1  def appartient(i,t):
2      if i < len(t):
3          return(t[i])
4      else:
5          return(False)
```

Q 6). : On commence par dupliquer le liste t_1 (boucle des lignes 4-6). Puis, par la boucle des lignes 7-8, on « supprime » les éléments de t_2 appartenant à t_1 à l'aide d'une opération booléenne. On prend en compte le fait que les deux listex ne sont pas nécessairement de la même longueur.
Remarque : la duplication du liste peut se faire en python grace à la méthode copy (`res=t1.copy()`). Par contre l'instruction `res=t1` ne créerait pas de copie de t1 car une liste est mutable. la fonction détruirait alors la liste fournie en entrée)

```
1  def diff(t1,t2):
2      n1 = len(t1)
3      n2 = len(t2)
4      res = []
5      for bool in t1:
6          res.append(bool)
7      for i in range(min(n1,n2)):
8          res[i] = res[i] and not(t2[i])
9      return(res)
```

Q 7). On commence par créer un liste correspondant à l'ensemble vide de la même longueur que le plus grand des listes (ligne 4). Dans la boucle des lignes 5-6, on ajoute les éléments présents dans t_1 et dans la boucle des lignes 7-8, on ajoute les éléments présents dans t_2. On retourne le résultat à la ligne 9.

```
1  def union(t1,t2):
2      n1 = len(t1)
3      n2 = len(t2)
4      res = [False]*max(n1,n2)
5      for i in range(n1):
6          res[i] = t1[i]
7      for i in range(n2):
8          res[i] = res[i] or t2[i]
9      return(res)
```

Partie II. Représentation par entiers

Q 8). L'entier $2^n - 1$ est le plus grand entier représentant un sous-ensemble de $\{0, 1, ..., n-1\}$, cet entier représentant le sous-ensemble $\{0, 1, ..., n-1\}$. L'entier 0 est le plus petit et représente l'ensemble vide.

Q 9). On parcourt la liste en partant de la fin. On initialise le résultat à 0 (ligne 3). À chaque passage dans la boucle des lignes 4 à 8, on multiplie le résultat par 2 et on ajoute 1 si l'élément $n-1-i$ appartient à l'ensemble S représenté par la liste t. Au final, on obtient l'entier représentant l'ensemble associé à la liste t en un temps linéaire (un seul parcours de boucle). (Il s'agit de l'algorithme de Hörner d'évaluation d'un polynôme)

```
1  def set2int(t):
2      res = 0
3      n = len(t)
4      for i in range(n):
5          if t[n-1-i]:
6              res = 1 + 2*res
7          else:
8              res = 2*res
9      return(res)
```

Q 10). Il s'agit de la décomposition d'un entier en base 2, les 1 et les 0 sont remplacés par **True** et **False**. La fonction est en $O(\log_2(n))$. (Si k vérifie : $2^k \leq n < 2^{k+1}$, la boucle while ligne 4 est réalisée $k+1$ fois, c'est à dire $\lceil \log_2(n) \rceil$ fois).

```
1  def int2set(n):
2      res=[]
3      while n>0:
4          if n % 2==0:
5              res.append(False)
6          else:
7              res.append(True)
8          n//=2
9      return(res)
```

Partie III. Familles, sous-familles, et couvertures

Q 11). On considère U l'ensemble des étudiants et pour une activité donnée a, l'ensemble F_a des élèves voulant pratiquer cette activité. On considère l'ensemble F de tous les F_a pour toutes les activités a. L'ensemble F constitue bien un recouvrement car chaque élève a choisi au moins une activité. Le problème de l'école est de donner au moins une activité à chaque élève tout en minimisant le nombre d'activités, c'est bien un problème de couverture optimale.

Q 12). On considère $n = 6$ et $F = (\{0, 2, 4\}, \{0, 1\}, \{2, 3\}, \{4, 5\})$. L'algorithme glouton conduit à choisir F comme sous-famille couvrante alors qu'une sous-famille couvrante optimale est $(\{0, 1\}, \{2, 3\}, \{4, 5\})$.

Q 13). On utilise les fonctions `diff` et `cardinal` agissant sur des entiers. Le résultat de la fonction `reste` n'est autre que le cardinal de l'ensemble $S_1 \setminus S_2$.

```
1       def reste(s1,s2):
2           return(cardinal(diff(s1,s2)))
```

Q 14). Dans la boucle conditionnelle des lignes 6 à 21, on applique l'algorithme glouton. On construit le sous ensemble couvrant de **F** représenté par **g** par ajout successif. À chaque passage dans la boucle, on détermine le sous-ensemble du complémentaire de **G** qui contient le plus d'éléments de non encore couverts. Ce sous-ensemble étant trouvé, on "ajoute" à **g** son indice dans la liste **f** (ligne 21) et on supprime ses éléments des éléments non encore couverts (ligne 20).

```
1       def glouton(n,f):
2           p = len(f)
3           compg=set2int([True]*p)
4           g = 0
5           noncouverts = set2int([True]*n)
6           while noncouverts > 0:
7               #recherche du meilleur candidat c
8               c=cardinal(noncouverts)
9               s=compg
10              i=0
11              puiss2i=1
12              while s>0:
13                  if s%2==1 and reste(noncouverts,f[i]) < c:
14                      c = reste(noncouverts,f[i])
15                      indice = i
16                      puiss2indice=puiss2i
17                  s//=2
18                  i+=1
19                  puiss2i*=2
20              noncouverts = diff(noncouverts,f[indice])
21              g+=puiss2indice
22          return(g)
```

Le coût de glouton en fonction de n et p, la longueur de la liste f est en $O(np)$. En effet :

Dans le pire des cas, dans la boucle conditionnelle (lignes 5-21), on supprime un seul élément dans **noncouverts** à chaque passage et donc cette boucle sera parcourue au plus n fois. La boucle interne des lignes 12 à 19 est parcourue au plus p fois ce qui nous donne un coût global en $O(np)$.

Q 15). Programme sans difficulté. Il suffit de parcourir l'ensemble des parties de **F** représenté par **g** en en "soustrayant" de l'ensemble des éléments non couverts les parties trouvées.

```
1       def couverture(g,f, n):
2           noncouverts=set2int([True]*n)
3           i=0
4           while g>0:
5               if g%2==1:
6                   noncouverts=diff(noncouverts,f[i])
7               g//=2
8               i+=1
9           return (noncouverts==0)
```

Q 16). Pour trouver la solution optimale, on étudie toutes les parties de F (boucle des lignes 5 à 9) et parmi celles qui recrouvrent l'ensemble $\{0, 1, ..., n-1\}$, on retourne une solution de cardinal minimal.

```
1  def optimal(n,f):
2      p = len(f)
3      cardPf=set2int([True]*p)+1
4      CardinalOptimal=p
5      for g in range(cardPf):
6          cardinalG = cardinal(g)
7          if couverture(g,f,n) and cardinalG < CardinalOptimal:
8              CardinalOptimal = cardinalG
9              res = g
10     return(res)
```

La boucle des lignes 5 à 9 est parcourue 2^p fois. Le coût de la fonction `cardinal` est en $O(n)$ (d'après la solution proposée à la question 1), le coût de la fonction `couverture` étant en $O(np)$ (On calcule la réunion d'au plus p ensembles de cardinal au plus n), on obtient un coût global en $O(np\,2^p)$.