CONCOURS GÉNÉRAL DES LYCÉES

—

SESSION 2024

—

Numérique et sciences informatiques

RAPPORT DE JURY

Rapport du jury de concours général des lycées

Épreuve de numérique et sciences informatiques

Session 2024

1 Composition du jury

- Marc de Falco, inspecteur général de l'éducation, du sport et de la recherche, président du jury.
- Pierre Hyvernat, maitre de conférences, université de Savoie Mont Blanc, vice-président du jury.
- Nathaniel Carré, professeur, académie de Paris.
- Florian Hatat, professeur, académie de Versailles.
- Anne Héam, professeur, académie de Besançon.
- Mathias Hiron, président de l'association France-ioi, organisateur des concours Algorea, Alkindi et Castor Informatique.
- Louis Jachiet, maitre de conférences, Télécom Paris.
- Laurent Sartre, professeur, académie de Bordeaux.
- Amélie Stainer, professeure, académie de Rennes.

2 Le rôle du concours général des lycées dans l'enseignement de NSI

La mise en place du concours général des lycées dans l'enseignement de NSI n'ayant pas été concomitante avec la mise en place de cette spécialité, il semble important au jury de rappeler le rôle structurant qu'un tel concours peut avoir dans les enseignements.

De nombreuses initiatives d'excellence existent en informatique. Certaines étant très anciennes, comme les Olympiades Internationales d'Informatique ou certains concours. D'autres, plus récentes, comme les Trophées NSI ou les Olympiades Académiques de NSI. Le concours général doit être vu comme un complément à celles-ci. Sa forme plus traditionnelle d'épreuve individuelle, reposant sur un sujet à traiter dans un temps limité sous format papier, permet d'offrir aux candidats une autre facette de l'informatique.

Tout en restant dans les limites induites par le programme officiel, le concours général permet, en visant l'excellence dans la maîtrise de l'ensemble des notions, de faire vivre ce programme d'une manière très différente. Ce faisant, sa préparation est très structurante pour les meilleurs élèves d'une classe. En visant une pleine compréhension des notions, les élèves peuvent ainsi commencer à envisager la discipline sous un angle qui pourra être suivi dans les études supérieures, notamment dans les classes préparatoires MP2I/MPI. Cette logique se retrouve dans les sujets du concours général dont la forme est parfois proche des sujets d'informatique des études supérieures.

3 Statistiques

Cette année, 549 candidats étaient inscrits et 531 étaient présents le jour de l'épreuve. Sur ces candidats inscrits, 513 l'étaient en France et 36 dans des lycées français à l'étranger.

Avec 56 personnes de genre féminin inscrites contre 493 de genre masculin, la proportion du genre féminin est très faible (10,2 %). Le jury regrette que le nombre de candidates ne soit pas plus élevé, même si cette proportion est comparable à celle des élèves du genre féminin sur l'ensemble des élèves inscrits en NSI en terminale.

Par rapport aux années précédentes Le nombre de candidats présents est en légère baisse, ce qui semble étonnant au regard du nombre de candidats potentiels.

Année	Candidats présents	Élèves suivant NSI en terminale
2022	576	16 158
2023	567	17 835
2024	531	17 612

Cette année, alors que le maximum de candidats est fixé à 8 %, seuls 3 % des élèves de NSI en terminale ont présenté le concours.

Les cartes choroplèthes présentées dans les figures 1 à 3 permettent d'identifier de grandes disparités selon les académies. Compte tenu du rôle rappelé plus haut du concours général dans la formation des élèves, le jury ne peut que regretter les très faibles proportions mesurées dans certaines académies.

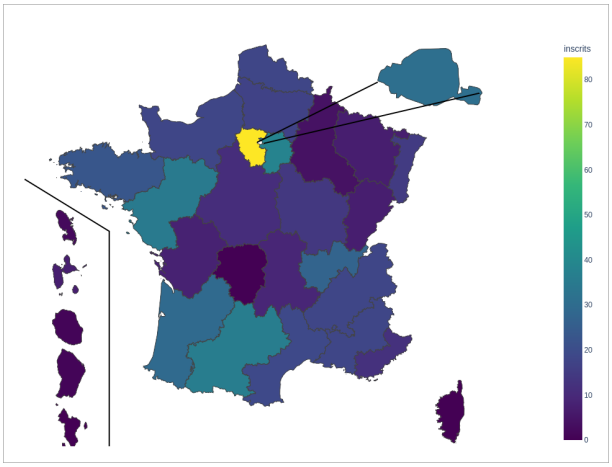

Figure 1: Répartition des inscrits par académie

4 Bilan de la session 2024

Le sujet comporte deux parties indépendantes. La première partie, sur la détermination de l'étage critique pour un lancer d'œufs, aborde un problème d'algorithmique et de programmation dont la résolution fait appel à des techniques habituelles de dichotomie et de programmation dynamique. La deuxième partie illustre la question de la représentation d'objets tridimensionnels et la résolution de quelques problèmes algorithmiques exploitant les représentations proposées. Cette résolution passe par la reconnaissance d'algorithmes de graphes. La partie se termine par une proposition de sérialisation de la structure de données pour permettre son stockage dans une base de données.

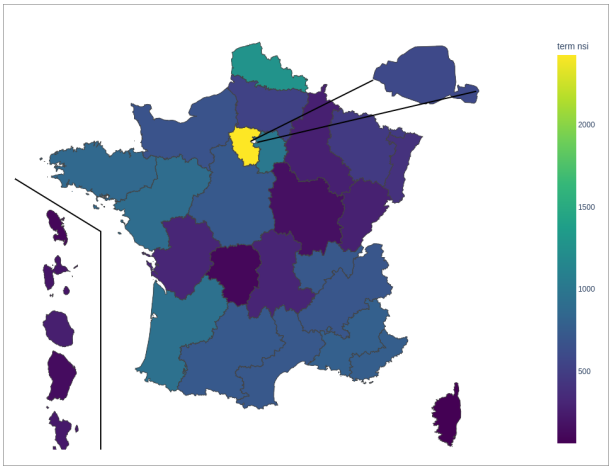

Figure 2: Répartition des élèves en terminale NSI par académie

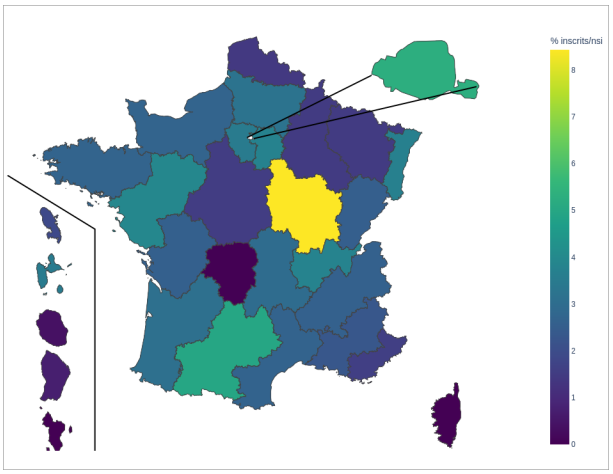

Figure 3: Répartition de la proportion des élèves en terminale NSI inscrits par académie

Chaque partie est assez longue, permettant aux candidates et aux candidats de s'investir dans la résolution de chaque partie grâce aux outils et connaissances du programme. Les meilleures copies ont pu traiter de très larges parties de chaque exercice.

5 Palmarès

Le jury a retenu 18 copies, conformément aux règles du Concours général des lycées, les estimant d'une qualité remarquable.

Le jury note que parmi les autres copies, un très grand nombre de copies étaient d'un niveau très bon ou excellent, ce qui témoigne d'un très bon niveau de préparation des candidates et des candidats.

Le jury félicite donc l'ensemble des candidates et des candidats pour leur participation et la bonne qualité générale des prestations. Il adresse sa reconnaissance aux professeurs pour la qualité de leur travail de préparation.

6 Conseils généraux aux futurs candidates et candidats

6.1 Présentation des copies

Le soin apporté à l'écriture, qui doit être lisible, et la présentation, en limitant notamment la quantité de ratures ou de rajouts entre les lignes, est essentiel pour faciliter le travail de lecture des copies et d'évaluation par le jury. Le jury conseille d'exploiter un brouillon pour aider à la réflexion et préparer les éléments de réponse avant leur rédaction.

En particulier, la rédaction de programmes en Python nécessite souvent, dès que la réponse attendue est un peu longue, de réfléchir au préalable à la structure générale du programme et à bien prendre en compte tous les cas.

Le jury recommande aux candidats de bien faire attention à la numérotation des questions. Celle-ci doit respecter *strictement* la numérotation de l'énoncé. Toute modification introduite par les candidats (absence de numéro, confusion de questions, passage de la numérotation en chiffres romains) présente le risque de tromper la vigilance du jury et de conduire à une mauvaise interprétation de la réponse donnée.

Les parties indépendantes peuvent être traitées dans l'ordre que les candidats préfèrent. À l'intérieur d'une partie, les questions sont rarement indépendantes. Plusieurs candidats ont choisi de traiter dans le désordre les questions d'une même partie. Cette stratégie est rarement efficace car elle n'aide pas à comprendre le raisonnement attendu, et le résultat est souvent difficilement lisible. De surcroit, le jury apprécie les copies présentant une progression nette dans les différentes parties, démontrant que l'idée d'ensemble de la partie est comprise.

Le jury considère qu'il est souvent plus facile d'aborder la rédaction des questions dans l'ordre de l'énoncé, quitte à parfois passer une question en laissant un blanc sur la copie, afin de se laisser la possibilité de compléter plus tard.

6.2 Programmation

Le jury a apprécié le fait que la majorité des codes Python étaient correctement présentés. Il restera attentif sur ce point et invite les candidats des sessions suivantes à bien faire attention à la présentation, notamment à bien indenter les blocs de code. Certains candidats ont fait le choix de marquer les niveaux d'indentation par des segments verticaux : ceci permet certes d'éviter des ambiguïtés d'écriture, mais ne devrait pas être nécessaire quand l'indentation est bien marquée.

En revanche, certains candidats ont choisi de marquer explicitement l'indentation avec, par exemple, une barre verticale suivie d'une flèche `|->` ou bien en marquant explicitement chaque espace par ␣. Ceci nuit à la lisibilité et est *fortement découragé*.

Plusieurs copies font usage d'éléments avancés de la bibliothèque standard Python. Cette pratique n'est pas sanctionnée par le jury, mais n'est pas non plus valorisée en tant que telle, les sujets étant conçus pour être traités avec les outils au programme de la spécialité NSI. Les candidats doivent s'assurer qu'ils ont la maitrise complète de ces éléments, toute erreur sur leur usage étant sanctionnée.

Parmi les erreurs générales fréquemment vues, on rappelle que l'expression `[] * n` crée une liste vide et pas une liste à `n` éléments.

7 Remarques détaillées par question

Question 1

On attend au maximum deux lancers, en faisant attention à bien lancer d'abord de l'étage $c-1$ puis de l'étage c. Dans l'autre sens, on risquerait de casser l'œuf sans pouvoir conclure sur l'étage critique.

Parmi les erreurs fréquentes, on note :

- des lancers effectués dans le mauvais ordre ;
- une recherche séquentielle en partant du premier étage, qui effectue trop de lancers d'œufs.

Il est demandé de respecter les noms des fonctions donnés par l'énoncé.

Question 2

La plupart des candidats ont bien écrit la recherche linéaire de l'étage.

Certains candidats ont écrit des boucles `while` avec des conditions trop compliquées, vraisemblablement pour éviter un `return` dans le corps de boucle. Ce faisant, ils ont introduit des erreurs qui empêchaient leur programme de terminer. Le jury estime que dans cette situation, la meilleure solution est d'utiliser une boucle `for` avec un `return` lorsque le cas recherché est atteint. Le risque d'oublier le `return` final (lorsque le cas recherché n'a pas été trouvé) est plus faible que le risque de se tromper dans la condition d'arrêt ou celui d'oublier l'initialisation ou l'incrémentation d'un compteur lorsqu'on utilise une boucle `while`.

Question 3-1

Dans le cas général, pour qu'une dichotomie soit possible, il faut s'assurer que le critère de recherche est monotone : c'est-à-dire qu'il faut expliquer que le résultat d'un lâcher à un étage donné est suffisant pour éliminer les étages supérieurs ou inférieurs. Ce critère essentiel a été très souvent oublié.

Une grande majorité des candidats s'assure bien par ailleurs que la quantité d'œufs est suffisante pour une dichotomie.

Question 3-2

Il s'agit d'un résultat de cours, qui ne demandait ici pas de justification particulière. En cohérence avec le programme de la spécialité NSI, tout logarithme, quelle que soit sa notation, est par défaut considéré comme étant le logarithme en base 2. Le jury suggère d'adopter la notation $\log_2(n)$ par cohérence avec les notations mathématiques usuelles.

Question 3-3

Il s'agit d'un algorithme de cours, qui n'est pas évident, mais dont l'écriture peut être attendue pour une épreuve de concours général. Il faut faire attention aux cas extrêmes (pas d'étage critique, étage critique à une extrémité) et à la condition d'arrêt (qui doit marcher également quand le nombre d'étapes n'est pas une puissance de 2).

Ce sujet donnait comme unique critère d'efficacité le nombre de lâchers d'œufs, c'est pourquoi il pouvait être envisageable d'effectuer une recherche dichotomique qui se rappelait récursivement sur un fragment de liste. Dans le cas général, cette technique est plutôt à décourager du fait du coût de la création de la sous-liste.

Une recherche itérative avec un `while` et deux indices de début et de fin, ou bien une recherche récursive avec une fonction auxiliaire prenant en paramètres l'indice de début et l'indice de fin convenaient pour cette question.

Le jury a noté deux erreurs fréquentes :

- utiliser la fonction `verif_etage_critique` à chaque étape de la dichotomie est contre productif car cela entraîne un lâcher d'œuf supplémentaire à chaque étage testé ;
- attention au parenthésage : `a + b // 2` ne renvoie pas le même résultat que `(a + b) // 2`.

Question 3-4

La notion de variant de boucle est au programme de la spécialité NSI. On peut fournir une réponse très convaincante en très peu de mots en exhibant précisément cette notion. Il est important dans cette question de s'appuyer précisément sur le code écrit à la question précédente : on ne demande pas un principe de terminaison de la dichotomie en général.

Question 4

Cette question a été peu traitée. Les réponses fournies étaient exactes dans leur très grande majorité, à l'exception de l'erreur fréquente suivante : certains candidats ont tenté d'appliquer la recherche dichotomique vue en question précédente, or elle ne convient pas par manque d'œufs.

Question 5

Cette question a été bien traitée dans la plupart des copies. Elle est facile, mais importante pour comprendre la suite.

Question 6

La formule proposée a souvent été bien comprise et expliquée.

Question 7

Cette question n'a quasiment jamais été réussie. La majorité des réponses fournies se contente d'écrire une fonction récursive, en suivant la formule donnée en question précédente, sans mémoïsation, ce qui donne une complexité temporelle beaucoup trop importante.

Il était possible de traiter cette question en utilisant les deux techniques habituelles en programmation dynamique (remplissage d'un tableau avec des boucles imbriquées, dans le bon ordre, ou bien récursivité avec mémoïsation).

L'usage du décorateur `functools.lru_cache`, par les candidats qui le connaissaient, ne répond pas complètement à la question car le cache utilisé est par défaut borné, ce qui ne garantit donc pas une complexité

polynomiale en fonction des paramètres. Il est préférable d'être très prudent dans l'utilisation d'objets de trop haut niveau de la bibliothèque standard, dont on se sert comme des boites noires sans vraiment en maitriser tous les aspects, surtout quand il existe une réponse simple avec des techniques plus élémentaires.

Question 8

Cette question est la suite habituelle en programmation dynamique de la question précédente. Il est quasiment impossible de la traiter correctement si la programmation dynamique n'est pas comprise.

Certain candidats ont souhaité réutiliser une partie de code déjà écrite à la question 7 sans tout réécrire. Il faut pour cela l'indiquer clairement, le jury ne peut pas deviner ce qui se cache derrière des points de suspension.

Question 9

Très peu de candidats ont abordé cette question. Elle peut être traitée en admettant l'existence de la fonction demandée en question précédente. La plupart des candidats ont préféré passer aux parties suivantes.

Question 10

La question peut sembler fastidieuse mais ne présente pas de difficulté particulière. Dans ce cas, l'énumération des 6 cas est plus simple et plus lisible (à condition d'introduire quelques variables) que des solutions "sophistiquées" qui augmentent le risque d'erreurs.

En particulier les propositions qui reposent sur l'idée suivante :

```
(x,y,z) = voxel
face = []
for dy in (0,1):
    for dz in (0,1):
        face.append((x,y+dy,z+dz))
```

ne sont pas valides car les sommets de la face ne sont pas dans le bon ordre, le polygone construit est croisé.

Question 11

La notion de "face visible" n'était pas précisée dans le sujet. Cet oubli n'a visiblement pas causé de problème dans les copies, mais il convient de préciser qu'une face est "visible" lorsqu'elle fait partie de la surface du volume complet. Une face d'un voxel est donc visible lorsque qu'elle n'appartient pas à un second voxel.

Pour tester si une face f d'un voxel v' est recouverte par un autre voxel (et donc, n'est pas visible), on peut :

- tester l'existence du seul autre voxel qui peut recouvrir f ;
- tester, pour chaque voxel $v \neq v'$, si f est une face de v.

Les deux solutions peuvent donner un coût linéaire. Cependant, la représentation d'une face n'est pas unique donc la deuxième version nécessite :

- de vérifier que la fonction `faces` de la question précédente renvoie toujours la même liste de sommets pour une face donnée, peu importe le voxel qui la contient. Cette condition ne fait pas partie de la spécification de l'énoncé, il faut donc au minimum la mentionner et s'assurer qu'elle est vérifiée ;
- ou bien programmer une fonction intermédiaire qui décide l'égalité de deux faces représentées différemment.

La première solution ne présentait pas cet inconvénient.

Question 12

Cette question laisse un peu de liberté aux candidats, permettant d'évaluer leur compréhension du problème et la qualité de leur approche scientifique. La réponse attendue était celle d'une soustraction de la surface externe à la surface totale trouvée en question 11, mais d'autres réponses étaient envisageables tant qu'elles étaient justifiées.

On note une interprétation imprévue de l'énoncé par de nombreux élèves, notamment la supposition qu'une géode est nécessairement sphérique, avec un trou également sphérique en son centre, comme dans l'exemple illustré. Une telle interprétation n'empêchait pas de répondre correctement aux autres questions, mais de nombreux candidats ont cherché, sans y parvenir, à s'appuyer dessus pour calculer la surface interne. Nombre d'entre eux ont fait l'erreur d'en déduire que la surface interne serait proportionnelle à la surface externe. Cette réponse n'est pas correcte.

Question 13

Une phrase rapide d'explication a été appréciée.

Question 14

L'énoncé précise qu'on ne considère que les faces visibles, donc répondre que toutes les faces accessibles sont visibles n'apporte aucune information supplémentaire.

Question 15

On attendait dans cette question que les candidats reconnaissent un parcours de graphe et donnent les détails concrets d'implémentation d'un tel parcours (partir d'une première face visible et découvrir les suivantes de proche en proche, comment ne pas visiter à nouveau des faces déjà vues). L'objectif était de valoriser les candidats ayant bien compris comment un tel parcours permettait de répondre à l'objectif de la partie, mais qui auraient était bloqués par l'implémentation en question 16.

Lorsque ces éléments n'étaient pas explicités en question 15 mais étaient clairement bien présents en question 16, le jury a considéré qu'ils étaient maitrisés par les candidats.

Question 16

Cette question technique de fin de partie est volontairement peu guidée.

Une construction explicite du graphe avant d'y faire le parcours n'était pas nécessaire ici. Sans créer le graphe, on a déjà suffisamment d'informations pour savoir quelle face est voisine de quelle autre.

Question 17

On n'attend pas une énumération manuelle de tous les cas, elle est ici bien trop fastidieuse à écrire. La plupart des candidats l'ont bien vu et ont proposé trois boucles `for`, en accord avec l'indication de l'énoncé.

Question 18

Le cas des voxels qui restent actifs est en général bien traité. L'ajout des voxels inactifs qui deviennent actifs est plus fréquemment omis. Une erreur fréquente vient du fait que la liste des voxels actifs ne peut pas être modifiée en place, car des voxels qui seront actifs à l'itération suivante sont alors à tort pris en compte pour déterminer le statut d'autres voxels de l'itération courante.

Question 19

Cette question vise à tester la culture acquise par les candidats, et plus particulièrement les problèmes habituels d'arrondis inhérents à la représentation flottante. Elle a été très largement bien traitée.

Question 20

Cette question a été majoritairement bien traitée.

Question 21

Cette question a été majoritairement bien traitée.

Question 22

De nombreuses erreurs proviennent du fait que les candidats considèrent que l'opérateur `in` s'exécute en temps constant sur des tableaux. Ceci n'est pas une hypothèse raisonnable étant donné l'implémentation naturelle du type `list` en Python.

Une justification rapide du calcul de complexité est souhaitable, en se référant explicitement au code des fonctions précédentes.

Annoncer, sans autre précision, qu'une complexité est « linéaire » ou « quadratique », sans indiquer selon quelle variable, n'est pas assez précis. Cette question faisant par ailleurs intervenir deux variables, seule une formule explicite pouvait convenir.

Question 23

L'énoncé aurait dû préciser que l'on ne souhaitait traiter que le cas des polygones convexes, un algorithme général étant délicat à obtenir (et n'a pas été proposé).

La principale source d'erreur se situe sur le découpage choisi : un petit dessin montre facilement que prendre itérativement des triangles de la forme `[face[i], face[i+1], face[i+2]]` n'est pas correct.

Il est maladroit de modifier le paramètre passé à cette fonction : des candidats ont construit une triangulation en prenant à chaque itération les trois premiers points d'une face puis en supprimant le point d'indice 1. Sauf si cela fait partie de la spécification de la fonction, un appelant ne s'attend pas à ce que les structures qu'il passe en arguments soient altérées.

Question 24

Plusieurs réponses étaient possibles, selon les variables que l'on considérait comme utilisables en entrée. La question aurait été plus pertinente en imposant que l'on devait partir de la demi-arête `a`.

Question 25

Les champs étant initialisés à `None` par les constructeurs, il faut penser à s'assurer qu'ils ont été modifiés avant d'accéder à leurs propriétés. La définition de la validité d'un sommet ou d'une face dépend de la validité d'une demi-arête, qui est plus compliquée et traitée en question suivante. Le jury a accepté les réponses qui ne tenaient pas compte de ce point, ou qui admettaient l'existence de la fonction `valide_demiarete`.

Question 26

Les remarques sont similaires à celles de la question 25.

Question 27

Des réponses non naturelles, par exemple « un graphe » ou « un arbre », qui ne donnaient aucune justification sur le fait que cela soit pertinent étant donné la restriction imposée par le sujet sur le seul attribut `suivant` de la structure, ont été refusées.

Question 28

Le jury encourage les candidats à s'entrainer à un rédiger un parcours correct de liste chainée. Cette question introduit une petite difficulté supplémentaire car la liste est cyclique. Il faut notamment faire attention au cas d'arrêt et ne pas oublier de traiter l'élément de départ lui-même.

Question 29

Il s'agit d'un parcours de graphe.

Question 30

Il ne faut créer qu'une seule face ici, car le nombre de faces augmente de 1, pas de 2. Par ailleurs, il faut bien identifier si la demi-arête de la face déjà existante se retrouvera dans la face contenant l'arête ab ou celle contenant l'arête ba.

Question 31

Cette question a été bien traitée dans la majorité des cas.

Question 32

Question souvent bien traitée, quand les candidats ont trouvé une manière correcte d'attribuer les valeurs des clés primaires.

8 Proposition de corrigé

Le jury propose un corrigé de l'épreuve afin d'aider les futurs candidats à préparer leur épreuve. De nombreuses autres réponses étaient possibles et ont été acceptées par le jury.

Question 1

```
def verif_etage_critique(c):
    # Il faut d'abord tester l'étage c-1, car si on teste d'abord l'étage c
    # alors on n'a plus d'œuf pour tester l'étage c-1 et on ne peut pas
    # conclure. On exploite la paresse de l'opérateur "and".
    return not lacher(c-1) and lacher(c)
```

Question 2

```
def critique_un_seul_oeuf(n):
    for i in range(1, n+1):
        if lacher(i):
            return i
    return n+1
```

Question 3-1

Si on jette un œuf depuis un étage i donné :

- soit il casse, dans ce cas on peut en déduire que l'étage critique est inférieur ou égal à i, il n'est alors pas utile de tester les étages supérieurs ;
- soit il ne casse pas, dans ce cas on peut en déduire que l'étage critique est strictement supérieur à i.

Si on effectue la recherche de l'étage critique entre deux étages a et b (initialement, $a = 1$ et $b = n$), alors on peut se contenter de tester l'étage du milieu et, en fonction du résultat, modifier a ou b et recommencer.

Cette stratégie dichotomique ne relance jamais d'œuf sur un étage déjà testé. Comme on dispose de $k = n$ œufs, on peut déterminer l'étage critique de cette façon.

Question 3-2

On effectue une recherche dichotomique sur un tableau de n cases, ce qui donne un nombre de tests de l'ordre de $\log_2(n)$.

Question 3-3

```python
def critique_dichotomie(n):
    bas = 1
    haut = n + 1
    # L'étage n+1 est hors de l'immeuble, on peut supposer fictivement qu'à
    # cet étage, l'œuf casse toujours. Ainsi à partir de maintenant,
    # l'invariant suivant est vérifié : lacher(haut) est vrai.
    # La boucle s'arrêtera quand haut == bas.
    while haut > bas:
        milieu = (bas + haut) // 2
        if lacher(milieu):
            haut = milieu
        else:
            bas = milieu + 1
    return haut
```

Question 3-4

On suppose que la fonction `lacher` termine. Dans ce cas, le contenu de la boucle `while` est uniquement composé d'opérations en temps constant. De même, le code avant et après la boucle est composé d'opérations qui terminent en temps constant. La terminaison de la fonction dépend donc uniquement de la terminaison de la boucle `while`.

On justifie que `haut` $-$ `bas` est un variant de boucle entier strictement décroissant et positif.

Quand on est dans la boucle, alors `bas` $<$ `haut`. Dans ce cas, on sait que `milieu` vérifie les inégalités `bas` $\leq$ `milieu` $<$ `haut`.

On note `haut`$'$ et `bas`$'$ les nouvelles valeurs de `haut` et `bas` à la fin de la boucle.

Dans le cas où l'œuf casse, `haut`$'$ $=$ `milieu` donc `haut`$'$ $<$ `haut`. L'écart entre `bas` et `haut` diminue d'au moins 1.

Dans le cas où l'œuf ne casse pas, `bas` $=$ `milieu` $+ 1$ donc `bas`$'$ $>$ `bas`. L'écart entre `bas` et `haut` diminue d'au moins 1.

Dans tous les cas, l'écart entre `haut` et `bas` diminue strictement à chaque tour de boucle, donc comme ce sont des entiers, l'inégalité `haut` $>$ `bas` finit par être fausse, donc la boucle termine.

En conclusion, la fonction termine.

Question 4

On peut faire la recherche en découpant les étages en p paquets de p étages :

- On commence par chercher le dernier paquet dont le premier étage ne casse pas l'œuf, il s'agit simplement d'une boucle `for` avec un pas de p ;
- En partant de cet étage, on cherche ensuite l'étage critique avec une boucle `for`.

```
def etage_critique_p2(p):
    etage = 0
    for e in range(0, p*p, p):
        if lacher(e+p):
            etage = e
            break

    for e in range(etage, etage+p):
        if lacher(e+1):
            return e

    return p*p
```

Si l'instruction `break` n'est pas connue, il est possible d'utiliser une fonction auxiliaire (avec un `return` dans la boucle `for`), ou une boucle `while`.

Question 5-1

Si l'œuf casse à l'étage i alors il casse à tous les étages supérieurs. Aucun d'entre eux n'est l'étage critique, donc l'étage critique est parmi les étages inférieurs ou égaux à i. Comme on a déjà testé l'étage i, il reste au plus les $i-1$ étages inférieurs à tester.

On n'a alors plus que $r-1$ œufs.

Question 5-2

Si l'œuf ne casse pas à l'étage i, alors il ne casse pas non plus aux étages inférieurs, donc il est inutile de les tester car l'étage critique n'est pas à ces étages. L'étage critique est parmi les étages strictement supérieurs à i (s'il existe). Si on a n étages au total, alors il reste $n-i$ étages supérieurs à tester. On dispose toujours de r œufs.

Question 6

Pour un i fixé, la question précédente montre que $L(n,k) \leq 1 + \max\{L(i-1,k-1), L(n-i,k)\}$: on ne sait pas laquelle des deux situations a lieu ni laquelle est la plus favorable (et il faut compter le lâcher de l'étage i). Cette relation est vraie pour tout i, donc $L(n,k) \leq 1 + \min\limits_{1 \leq i \leq n} \max\{L(i-1,k-1), L(n-i,k)\}$. Cependant toute stratégie doit commencer par faire un lâcher, depuis un certain étage i, donc l'inégalité ne peut être stricte.

Question 7

On utilise soit une matrice, soit la mémoïsation. On donne ci-dessous les deux versions (une seule version est attendue sur une copie).

Dans un cas comme dans l'autre, chaque $L(e, r)$ est calculé au plus une fois. Comme pour chacun, le calcul du minimum coûte de l'ordre de e opérations, le coût en temps est donc de l'ordre de $e^2 r$.

Avec mémoïsation :

```
def L_rec(e, r, cache):
    if (e,r) in cache:
        return cache[(e,r)]
    if e == 0:
        v = 0
    elif r == 0:
        v = Infty
    else:
        v = 1 + min([max(L_rec(i-1, r-1, cache),
                         L_rec(e-i, r, cache))
                     for i in range(1, e+1)])
        cache[(e,r)] = v
        return v

def L_memo(e, r):
    cache = {}
    return L_rec(e, r, cache)
```

Avec une matrice :

```
def L(e,r):
    # Création du tableau et valeurs initiales
    valeurs = [[Infty for j in range(r+1)]
               for i in range(e+1)]
    for j in range(r+1):
        valeurs[0][j] = 0

    # Calcul du reste du tableau
    for j in range(r+1):
        for k in range(1, e+1):
            valeurs[k][j] = \
                    1 + min([max(valeurs[i-1][j-1],
                                 valeurs[k-i][j])
                             for i in range(1, k+1)])
    return valeurs[e][r]
```

Question 8

```
def Lprec(e,r):
    # Création du tableau et valeurs initiales
    valeurs = [[Infty for j in range(r+1)]
               for i in range(e+1)]
    prec = [[-1 for j in range(r+1)]
            for i in range(e+1)]
    for j in range(r+1):
```

```
        valeurs[0][j] = 0

    for j in range(1, r+1):
        for k in range(1, e+1):
            m = Infty
            for i in range(1, k+1):
                suivant = 1 + max(valeurs[i-1][j-1], valeurs[k-i][j])
                if suivant < m:
                    prec[k][j] = i
                    m = suivant
            valeurs[k][j] = m

    return prec
```

Question 9

```
def aux(prec, k, niv_bas, niv_haut):
    if niv_haut == niv_bas:
        if lacher(niv_haut):
            return niv_haut
        else:
            return niv_haut + 1
    i = prec[niv_haut-niv_bas][k] + niv_bas
    if lacher(i):
        return aux(prec, k-1, niv_bas, i-1)
    else:
        return aux(prec, k, i+1, niv_haut)

def strategie(prec, n, k):
    return aux(prec, k, 1, n)
```

Question 10

```
def faces(v):
    x0, y0, z0 = v
    x1, y1, z1 = x0+1,y0+1,z0+1
    return [
        [(x0,y0,z0), (x1,y0,z0), (x1,y1,z0), (x0,y1,z0)], # face basse
        [(x0,y0,z1), (x1,y0,z1), (x1,y1,z1), (x0,y1,z1)], # face haute
        [(x0,y0,z0), (x1,y0,z0), (x1,y0,z1), (x0,y0,z1)], # face avant
        [(x0,y1,z0), (x1,y1,z0), (x1,y1,z1), (x0,y1,z1)], # face arrière
        [(x0,y0,z0), (x0,y1,z0), (x0,y1,z1), (x0,y0,z1)], # face gauche
        [(x1,y0,z0), (x1,y1,z0), (x1,y1,z1), (x1,y0,z1)]  # face droite
    ]
```

Question 11

```
def nb_faces_visibles(V):
    F = [] # tableau des faces vues
    n = 0 # nombre de faces
    for v in V:
        for f in faces(v):
```

```
            if f in F:
                # Si une face est vue 2 fois, c'est une face interne, il
                # ne faut donc pas la compter, or on l'avait comptée
                # précédemment la première fois qu'on l'avait vue.
                n -= 1
            else:
                n += 1
                F.append(f)
    return n
```

Pour améliorer le temps de calcul, on pourrait remplacer le tableau F par un ensemble. La recherche serait alors nettement plus rapide.

```
def nb_faces_visibles(V):
    F = set() # ensemble des faces déjà traitées
    n = 0 # nombre de faces
    for v in V:
        for f in faces(v):
            f = tuple(f) # pour que les faces ne soient pas mutables
            if f in F:
                n -= 1
            else:
                n += 1
                F.add(f)
    return n
```

(Attention, on ne peut pas directement faire des ensembles de tableaux, il faut utiliser des tuples, non modifiables.)

Question 12

On remarque que si le nombre de faces visibles est notée v, que la surface externe est notée S_e et que la surface interne est notée S_i, l'égalité $v = S_e + S_i$ est vérifiée.

On peut donc retrouver la surface interne en soustrayant la surface externe au nombre total de faces visibles.

Question 13

On note (x_m, y_m, z_m) la coordonnée qui représente le voxel de plus petite coordonnée en x. La face $[(x_m, y_m, z_m), (x_m, y_m, z_m + 1), (x_m, y_m + 1, z_m + 1), (x_m, y_m + 1, z_m)]$ est sur la surface externe. En effet, elle est visible, car le seul voxel qui pourrait la rendre invisible serait en abscisse $x_m - 1$, ce qui est impossible car x_m est la plus petite coordonnée de voxel. Les surfaces internes ont des faces situées plus à droite, c'est-à-dire avec des coordonnées en x plus grandes, donc la face est bien externe.

Question 14

On note f une face de la surface externe. Une face est sur la surface externe si et seulement s'il existe un chemin depuis f dans le graphe décrit par le sujet.

Question 15

On parcourt la liste des voxels afin de déterminer le voxels de plus petite coordonnée en x. On note ce voxel v_m. On construit une première face f_m à partir de ce voxel, comme indiqué en question 13. On réalise ensuite un parcours en profondeur (un parcours en largeur marcherait aussi). Pour cela, on marque initialement

toutes les faces comme non vues dans un tableau de booléens. On écrit une fonction récursive qui prend en paramètre une face. Si elle a déjà été vue, on ne fait rien, sinon on la marque comme vue et on rappelle récursivement le parcours sur chacune des faces voisines dans le graphe. L'appel initial se fait avec la face f_m. À la fin, une face est externe exactement quand elle a déjà été marquée comme vue par le parcours.

Question 16

```
def faces_visibles(V):
    # On reprend nb_faces_visibles mais on la modifie pour renvoyer les faces
    # visibles et pas uniquement leur nombre.
    F = set() # ensemble des faces déjà traitées
    Fv = set() # ensemble des faces visibles
    for v in V:
        for f in faces(v):
            f = tuple(f)
            if f in F:
                Fv.remove(f)
            else:
                F.add(f)
                Fv.add(f)
    return Fv

def aretes(f):
    """
    Fonction qui renvoie la liste des arêtes d'une face donnée.
    Les sommets dans chaque arête sont triés pour faciliter leur comparaison
    entre faces.
    """
    return [sorted([f[i-1],f[i]]) for i in range(4)]

def faces_coincidentes(f1, f2):
    """
    Fonction qui teste si deux faces sont reliées par une arête
    """
    a1 = aretes(f1)
    a2 = aretes(f2)
    return any(a in a2 for a in a1)

def faces_voisines(F, f):
    """
    Fonction qui détermine la liste des voisines de la face f
    """
    # Version très naïve, linéaire en le nombre de faces
    voisines = []
    for fp in F:
        if f != fp and faces_coincidentes(f, fp):
            voisines.append(fp)
    return voisines

def surface_externe(voxels):
    m = 0
    for i in range(1, len(voxels)):
```

```
        if voxels[i][0] < voxels[m][0]:
            m = i
    xm, ym, zm = voxels[m]
    F = faces_visibles(voxels)
    fm = ((xm, ym, zm), (xm, ym+1, zm), (xm, ym+1, zm+1), (xm, ym, zm+1))
    surface = set()
    a_visiter = [fm]
    while len(a_visiter) != 0:
        f = a_visiter.pop()
        if f in surface:
            continue
        surface.add(f)
        a_visiter += faces_voisines(F, f)
    return len(surface)
```

Question 17

```
def voisins(voxel):
    (x,y,z) = voxel
    res = []
    for dx in (-1,0,1):
        for dy in (-1,0,1):
            for dz in (-1,0,1):
                if dx != 0 or dy != 0 or dz != 0:
                    res.append((x+dx,y+dy,z+dz))
    return res
```

Question 18

```
def evolution(voxels_actifs):
    # Dictionnaire qui permet de savoir rapidement si un voxel est actif au début
    actif = {}
    # Dictionnaire qui pour chaque voxel compte le nombre de voisins actifs
    voisins_actifs = {}
    for voxel_actif in voxels_actifs:
        actif[voxel_actif] = True
        for voxel in voisins(voxel_actif):
            if voxel not in voisins_actifs:
                voisins_actifs[voxel] = 1
            else:
                voisins_actifs[voxel] += 1

    # Construction du résultat
    res = []
    for voxel in voisins_actifs:
        if voxel in actif:
            if 2 <= voisins_actifs.get(voxel, 0) <= 3:
                res.append(voxel)
        elif voisins_actifs.get(voxel, 0) == 3:
            res.append(voxel)

    return res
```

Question 19

Tous les calculs sur les sommets vont faire apparaitre des approximations, qui ne seront pas les mêmes suivant la manière utilisée pour faire les calculs. Par exemple, le barycentre du triangle (a, b, c) sera souvent différent du barycentre du triangle (c, b, a) ! On ne peut pas faire disparaitre le problème, mais on peut au minimum, remplacer la comparaison `==` entre flottants par une comparaison approximative :

```
def approx(x, y, epsilon=1e-6):
    return x-y < epsilon and y-x < epsilon
```

Question 20

```
def faces_adjacentes_sommet(F, p):
    res = [] # faces adjacentes trouvées
    for i in range(len(F)):
        if p in F[i]:
            res.append(i)
    return res
```

Question 21

```
def faces_adjacentes_cote(F, k):
    # Dictionnaire qui à chaque sommet associe les indices des faces qui le
    # contiennent
    faces = {}
    for face_id in range(len(F)):
        for sommet in F[face_id]:
            if sommet in faces:
                faces[sommet].append(face_id)
            else:
                faces[sommet] = [face_id]

    # Construction du résultat
    res = {}
    for sommet in F[k]:
        for face_id in faces[sommet]:
            res[face_id] = True
    return res.keys()
```

Question 22

Dans `faces_adjacentes_sommet`, le test d'appartenance nécessite un nombre d'opérations de l'ordre de s. Le reste du corps de la boucle est en temps constant. La boucle `for` tourne n fois, donc l'ensemble de la fonction s'exécute en temps de l'ordre de ns.

Dans `faces_adjacentes_cote`, en supposant que les opérations sur les dictionnaires sont en temps constant, la fonction s'exécute en temps de l'ordre de ns. (Une version plus naïve sans dictionnaire s'exécuterait plutôt en temps ns^2.)

Question 23

On fait l'hypothèse que le polygone donné en paramètre est convexe.

```python
def triangule(S):
    T = []
    for f in S:
        triangles_f = []
        a = f[0]
        for i in range(1, len(f)-1):
            triangles_f.append([a, f[i], f[i+1]])
        T.extend(triangles_f)
    return T
```

Question 24

1. `a.suivant.suivant.miroir.suivant.suivant.miroir.suivant.miroir.suivant.suivant`
2. `a.suivant.suivant.miroir.suivant.suivant.miroir.face`

Question 25

```python
def valide_face(face):
    if face.darete is None:
        return False
    return valide_darete(face.darete) and face.darete.face == face

def valide_sommet(sommet):
    if sommet.darete is None:
        return False
    return valide_darete(sommet.darete) and sommet.darete.source == sommet
```

Question 26

```python
def valide_darete(a):
    # Tous les champs ont été initialisés
    if a.face is None or a.source is None or a.but is None or a.miroir is None:
        return False

    # Le miroir va bien dans l'autre sens
    if a.miroir.source != a.but or a.miroir.but != a.source:
        return False

    # On cherche la demi-arête parmi celles de la face
    debut = True
    current_darete = a.face.darete
    while current_darete is not None and (debut or current_darete != a.face.darete):
        # On a l'égalité de demi-arête, il est inutile de tester le champ suivant,
        # c'est nécessairement le même.
        if current_darete == a:
            return True
        debut = False
        current_darete = current_darete.suivant

    return False
```

Question 27

Ceci s'apparente à une structure de liste chainée.

Question 28

```
def faces_voisines(F, f):
    """
    Fonction qui détermine la liste des voisines de la face f
    """
    # Version très naïve, linéaire en le nombre de faces
    voisines = []
    for fp in F:
        if f != fp and faces_coincidentes(f, fp):
            voisines.append(fp)
    return voisines
```

Question 29

```
def composante_connexe(face):
    faces_vues = {face: True}
    a_visiter = [face]

    while len(a_visiter) > 0:
        next_face = a_visiter.pop()
        for voisine in faces_voisines(face):
            if voisine in faces_vues:
                continue
            faces_vues[voisine] = True
            a_visiter.append(voisine)

    return faces_vues.keys()
```

Question 30

```
def separe_face(f, a, b):
    # On commence par trouver les demi-arêtes qui arrivent en a et en b
    da_vers_a = None
    da_vers_b = None
    da_actuelle = f.darete
    while da_vers_a is None or da_vers_b is None:
        if da_actuelle.but == a:
            da_vers_a = da_actuelle
        if da_actuelle.but == b:
            da_vers_b = da_actuelle
        da_actuelle = da_actuelle.suivant

    # On crée une nouvelle face
    nv_face = Face()
    # On crée les deux nouvelles demi-arêtes miroirs
    dab = DArete(a, b, f)
    dba = DArete(b, a, nv_face)
```

```
    nv_face.darete = dba
    f.darete = dab # Pour être sûr d'avoir une demi-arête bien sur la face
    dab.miroir = dba
    dba.miroir = dab
    dab.suivant = da_vers_b.suivant
    dba.suivant = da_vers_a.suivant

    # Celle qui va vers b continue avec dba, et on doit rattacher toutes
    # les arêtes après a à la nouvelle face.
    da_vers_b.suivant = dba
    da_actuelle = dba.suivant
    while da_actuelle != dba:
        da_actuelle.face = nv_face
        da_actuelle = da_actuelle.suivant

    # Celle qui va vers a continue avec dab. La face ne change pas.
    da_vers_a.suivant = dab

    return dab
```

Question 31

Dans les tables `sommet`, `face` et `demiarete`, chaque colonne `id` est une clé primaire.

Dans la table `sommet`, la colonne `darete` est une clé étrangère vers la colonne `id` de `demiarete`. Il en est de même pour la colonne `darete` de la table `face`.

Dans la table `demiarete` :

- la colonne `source` est une clé étrangère vers la colonne `id` de `sommet`,
- la colonne `but` est une clé étrangère vers la colonne `id` de `sommet`,
- la colonne `face` est une clé étrangère vers la colonne `id` de `face`,
- la colonne `miroir` est une clé étrangère vers la colonne `id` de `demiarete`,
- la colonne `suivant` est une clé étrangère vers la colonne `id` de `demiarete`.

Question 32

Le code suivant insère les données dans la base. En cours de construction, l'intégrité des clés étrangères n'est pas vérifiée mais elle l'est à la fin de la fonction. Pour que cette séquence d'insertions SQL fonctionne, il faut, selon le moteur de base de données, désactiver la vérification des clés étrangères ou déclarer ces contraintes comme vérifiables plus tard (`DEFERRED`).

```
def serialise(S, F, A):
    # Chaque objet va posséder l'identifiant correspondant à son rang dans les
    # tableaux S, F ou A.
    id_sommet = {sommet: id for id, sommet in enumerate(S)}
    id_face = {face: id for id, face in enumerate(F)}
    id_darete = {darete: id for ids, darete in enumerate(A)}

    # On insère les sommets
    values = []
    for id_sommet, sommet in enumerate(S):
```

```
    values.append("("
        + ",".join([id_sommet, sommet.x, sommet.y, sommet.z, id_darete[sommet.darete]])
        + ")")
values = ", ".join(values)
sql_execute(f"INSERT INTO sommet (id, x, y, z, darete) VALUES {values};")

# On insère les faces
values = []
for id_face, face in enumerate(F):
    values.append("("
        + ",".join([id_face, id_darete[face.darete]])
        + ")")
values = ", ".join(values)
sql_execute(f"INSERT INTO face (id, darete) VALUES {values};")

# On insère les demi-arêtes
values = []
for id_darete, darete in enumerate(A):
    values.append("("
        + ",".join([id_darete, id_sommet[darete.source], id_sommet[darete.but],
                    id_face[darete.face], id_darete[darete.miroir],
                    id_darete[darete.suivant]])
        + ")")
values = ", ".join(values)
sql_execute(f"INSERT INTO demiarete (id, source, but, face, miroir, suivant) VALUES {values};")
```