Prérequis pour entrer en terminale NSI

Cette fiche teste les acquis de première NSI indispensables pour aborder la terminale sereinement. Essaie chaque exercice **sans ton cours**, note tes réponses au brouillon, puis déplie la solution correspondante pour te corriger. Si tu bloques, c'est le signe qu'il faut reprendre le point correspondant avant d'aller plus loin.

Réussi sans hésitation : passe au suivant. **Réussi avec hésitation** : fais deux exercices supplémentaires pour consolider. **Échoué** : retravaille la leçon correspondante avant de poursuivre.

① Listes et tableaux

Exercice 1 *Questions*

1. Écrire une fonction `maximum(L)` qui renvoie le plus grand élément d'une liste non vide, sans utiliser `max()` (la liste peut contenir des nombres négatifs).
2. Réécrire la boucle suivante sous forme de liste en compréhension :

```
resultat = []
for x in range(1, 11):
    if x % 3 == 0:
        resultat.append(x ** 2)
```

3. Expliquer pourquoi le code suivant affiche `[99, 2, 3]` :

```
A = [1, 2, 3]
B = A
B[0] = 99
print(A)
```

   Proposer deux écritures qui, elles, préservent `A`.
4. Écrire une instruction qui crée une grille 3 lignes × 4 colonnes remplie de zéros (liste de listes), puis place la valeur 1 en ligne 2, colonne 3 (indices à partir de 0).

② Dictionnaires et p-uplets

Exercice 2 *Questions*

1. Créer un dictionnaire `planetes` associant `"Terre"` à 1, `"Mars"` à 2 et `"Jupiter"` à 5. Ajouter ensuite la clé `"Saturne"` avec la valeur 9.
2. Écrire une fonction `inverser(d)` qui échange clés et valeurs d'un dictionnaire (on suppose les valeurs distinctes et immuables).
3. Quelle est la différence entre `d["cle"]` et `d.get("cle", 0)` ?
4. Pourquoi ne peut-on pas utiliser une liste comme clé de dictionnaire, alors qu'un tuple est accepté ?

③ Recherche dans un tableau

Exercice 3 *Questions*

1. Écrire une fonction `recherche(L, x)` qui renvoie l'indice de la première occurrence de `x` dans `L`, ou -1 si `x` est absent.
2. Quelle est la précondition pour appliquer une recherche dichotomique ?
3. Donner la trace de la recherche dichotomique de 7 dans `L = [1, 3, 5, 7, 9, 11]` :

Étape	`gauche`	`droite`	`milieu`	Comparaison
1				
2				
3				

4. Sur un tableau trié de 1 000 000 d'éléments, combien de comparaisons fait la recherche dichotomique dans le pire cas ? La recherche séquentielle ?

④ Tris par comparaison

Exercice 4 *Questions*

1. Écrire une fonction `tri_selection(L)` qui trie la liste `L` en place (l'algorithme cherche, à chaque étape, le minimum de la partie non encore triée et le place à sa position).
2. Compléter le tableau d'état du tri par sélection appliqué à `L = [5, 2, 8, 1, 4]` :

Itération `i`	Minimum de `L[i:]`	Échange	État de `L` après échange
0			
1			
2			
3			

3. Décrire en quelques lignes le principe du tri par insertion.
4. Donner la complexité, dans le pire cas, du tri par sélection et du tri par insertion. Que devient la complexité du tri par insertion lorsque la liste est déjà triée ?

⑤ Algorithmes gloutons

Exercice 5 *Questions*

1. Quelles sont les deux caractéristiques d'un algorithme glouton ?
2. Écrire une fonction `rendu(somme, pieces)` qui rend `somme` en utilisant le minimum de pièces selon la stratégie gloutonne, en supposant `pieces` triée par ordre décroissant. La fonction renvoie la liste des pièces utilisées.
3. Tester mentalement `rendu(47, [200, 100, 50, 20, 10, 5, 2, 1])`. Combien de pièces la fonction renvoie-t-elle ?
4. Soit le système de pièces `[4, 3, 1]`. L'algorithme glouton donne-t-il toujours le nombre minimal de pièces pour obtenir 6 ? Justifier avec un contre-exemple.

⑥ k plus proches voisins

Exercice 6 *Questions*

1. Décrire en une phrase le principe de l'algorithme des k plus proches voisins (kNN) pour classer un nouveau point.
2. Calculer la distance euclidienne entre les points A(2 ; 3) et B(5 ; 7) du plan.
3. Que se passe-t-il si on choisit $k = 1$? si k est égal au nombre total de points d'apprentissage ?
4. Dans un jeu de données avec deux variables — âge (entre 15 et 80 ans) et revenu annuel (entre 10 000 et 100 000 euros) —, pourquoi faut-il normaliser les variables avant d'appliquer kNN ?

⑦ Complexité

Exercice 7 *Questions*

1. Donner la complexité de chacune des fonctions suivantes en fonction de la taille n de l'entrée :

```
def f(L):
    s = 0
    for x in L:
        s += x
    return s

def g(L):
    for x in L:
        for y in L:
            pass

def h(n):
    while n > 1:
        n = n // 2
    return n
```

2. Classer les complexités suivantes de la plus efficace à la moins efficace : $O(n^2)$, $O(1)$, $O(n \log n)$, $O(2^n)$, $O(n)$, $O(\log n)$.
3. Une opération élémentaire dure 1 microseconde. Estimer le temps d'exécution, pour $n = 10^4$: d'un algorithme en $O(n^2)$; d'un algorithme en $O(n \log n)$. Commenter.

Solutions des exercices

Corrigé de l'exercice 1.

1.
```
def maximum(L):
    maxi = L[0]
    for x in L:
        if x > maxi:
            maxi = x
    return maxi
```

   Initialiser `maxi` avec `L[0]` et non `0` : sinon, pour une liste de nombres tous négatifs, la fonction renverrait 0, qui n'est pas dans la liste.
2. `resultat = [x ** 2 for x in range(1, 11) if x % 3 == 0]`. Résultat : `[9, 36, 81]`.
3. `B = A` crée un alias, pas une copie : `A` et `B` désignent la même liste en mémoire. Modifier `B[0]` modifie donc aussi `A`. Pour une copie indépendante, on écrit `B = A[:]` ou `B = list(A)`.
4.
```
grille = [[0] * 4 for _ in range(3)]
grille[2][3] = 1
```

   Attention : `grille = [[0] * 4] * 3` donne trois fois la même ligne (alias), modifier une cellule modifie les trois lignes.

Si tu échoues : reprends la syntaxe des listes en compréhension, la distinction entre référence et copie, et la construction d'un tableau 2D. Ces trois points sont systématiquement utilisés en terminale (parcours de matrices, implémentation de structures, algorithmes sur les graphes). Fiches à revoir : Les listes, Tableaux à deux dimensions.

Corrigé de l'exercice 2.

1.
```
planetes = {"Terre": 1, "Mars": 2, "Jupiter": 5}
planetes["Saturne"] = 9
```

2.
```
def inverser(d):
    return {v: k for k, v in d.items()}
```

3. `d["cle"]` lève `KeyError` si la clé est absente ; `d.get("cle", 0)` renvoie 0 (valeur par défaut) sans erreur.
4. Les clés d'un dictionnaire doivent être *immuables* (on doit pouvoir calculer un haché stable). Une liste est mutable : son contenu peut changer, le haché deviendrait incohérent. Un tuple, lui, est immuable, il peut donc servir de clé (par exemple pour indexer par couple de coordonnées).

Si tu échoues : reprends la syntaxe `{cle: valeur}`, l'ajout par `d[cle] = valeur`, la méthode `items()` qui donne les paires parcourables, et la différence tuple (immuable) / liste (mutable). Les dictionnaires indexés par tuples sont omniprésents en terminale (matrices creuses, mémoïsation, graphes). Fiche à revoir : Les dictionnaires.

Corrigé de l'exercice 3.

1.
```
def recherche(L, x):
    for i in range(len(L)):
        if L[i] == x:
            return i
    return -1
```

2. La liste doit être triée (par ordre croissant ou décroissant).
3. Trace pour la dichotomie de 7 dans `[1, 3, 5, 7, 9, 11]` :

Étape	`gauche`	`droite`	`milieu`	Comparaison
1	0	5	2	L[2] = 5 < 7, on cherche à droite
2	3	5	4	L[4] = 9 > 7, on cherche à gauche
3	3	3	3	L[3] = 7, trouvé à l'indice 3

4. Dichotomie : au plus $\lceil \log_2(10^6) \rceil = 20$ comparaisons. Recherche séquentielle : jusqu'à 10^6 comparaisons. La dichotomie est environ 50 000 fois plus rapide dans le pire cas.

Si tu échoues : reprends la recherche séquentielle (avec sortie anticipée dès que l'élément est trouvé) et la dichotomie sur tableau trié (maintien des bornes `gauche` et `droite`, calcul du `milieu`, mise à jour par moitié). Retiens la précondition de tri : la dichotomie sur un tableau non trié renvoie n'importe quoi. Fiche à revoir : Recherche dans une liste.

Corrigé de l'exercice 4.

1.
```
def tri_selection(L):
    n = len(L)
    for i in range(n - 1):
        imin = i
        for j in range(i + 1, n):
            if L[j] < L[imin]:
                imin = j
        L[i], L[imin] = L[imin], L[i]
```

2. État du tri par sélection sur `[5, 2, 8, 1, 4]` :

Itération `i`	Minimum de `L[i:]`	Échange	État de `L` après échange
0	1 (en position 3)	`L[0]` ↔ `L[3]`	`[1, 2, 8, 5, 4]`
1	2 (en position 1)	pas d'échange	`[1, 2, 8, 5, 4]`
2	4 (en position 4)	`L[2]` ↔ `L[4]`	`[1, 2, 4, 5, 8]`
3	5 (en position 3)	pas d'échange	`[1, 2, 4, 5, 8]`

3. Le tri par insertion parcourt la liste de gauche à droite. À chaque étape, l'élément courant est inséré à sa bonne place dans la portion déjà triée située à sa gauche (on le décale vers la gauche tant qu'il est plus petit que son voisin).
4. Tri par sélection : $O(n^2)$ dans tous les cas (le nombre d'échanges peut être plus faible, mais les comparaisons restent quadratiques). Tri par insertion : $O(n^2)$ dans le pire cas (liste triée à l'envers), $O(n)$ dans le meilleur cas (liste déjà triée : une seule comparaison par élément).

Si tu échoues : reprends le schéma commun aux deux tris (invariant : la partie gauche de la liste est triée à chaque itération) et la différence de stratégie (sélection : on choisit le minimum ; insertion : on insère le courant). Écrire au moins une fois un tri à la main est indispensable avant de voir le tri fusion en terminale. Fiche à revoir : Tris naïfs.

Corrigé de l'exercice 5.

1. Un algorithme glouton fait à chaque étape le choix qui semble le meilleur localement et ne revient jamais en arrière sur une décision prise. Il est rapide mais ne garantit pas l'optimalité globale.

2.
```
def rendu(somme, pieces):
    resultat = []
    for p in pieces:
        while somme >= p:
            resultat.append(p)
            somme -= p
    return resultat
```

3. L'appel ci-dessous renvoie `[20, 20, 5, 2]`, soit 4 pièces.

Programme 1 · Appel testé

```
1 rendu(47, [200, 100, 50, 20, 10, 5, 2, 1])
```

4. Avec `[4, 3, 1]` pour obtenir 6 : le glouton prend d'abord 4, puis il faut faire 2 avec les pièces restantes, ce qui donne $4 + 1 + 1 = 3$ pièces. Or $3 + 3 = 6$ ne nécessite que 2 pièces. Le système n'est donc pas canonique : sur celui-ci, le glouton n'est pas optimal.

Si tu échoues : reprends la définition (choix local, pas de retour en arrière) et le principe du rendu de monnaie. Retiens surtout que le système de pièces de l'euro est canonique (le glouton est optimal), mais que changer le système peut rendre le glouton sous-optimal. En terminale, cette nuance revient pour justifier que la programmation dynamique est parfois nécessaire. Fiche à revoir : Algorithmes gloutons.

Corrigé de l'exercice 6.

1. On cherche les k points de la base d'apprentissage les plus proches du point à classer (pour une distance choisie), puis on lui attribue la classe majoritaire parmi ces k voisins.
2. $d(\mathrm{A}, \mathrm{B}) = \sqrt{(5-2)^2 + (7-3)^2} = \sqrt{9+16} = \sqrt{25} = 5$.
3. $k = 1$: le modèle est très sensible au bruit (un voisin mal étiqueté entraîne une mauvaise classification). k égal au nombre total de points : on prédit toujours la classe majoritaire globale, sans tenir compte de la position du point. Le bon k est un compromis entre ces deux extrêmes (souvent ajusté par validation).
4. Sans normalisation, la variable revenu (ordre de 10^4) domine totalement la variable âge (ordre de 10) dans le calcul de la distance. Deux personnes d'âge identique mais de revenus différents seront considérées comme très éloignées, alors que deux personnes de même revenu mais d'âges très différents seront vues comme proches. La normalisation (par exemple, ramener chaque variable à l'intervalle $[0\ ;\ 1]$) rétablit l'équilibre entre les dimensions.

Si tu échoues : reprends la distance euclidienne en dimension 2 puis d, le principe du vote majoritaire, et l'effet du choix de k. La normalisation des variables est un réflexe à acquérir : elle réapparaîtra en terminale dès qu'on mélange des grandeurs de nature différente. Fiche à revoir : Algorithme kNN.

Corrigé de l'exercice 7.

1. `f` : une boucle parcourt la liste, donc $\mathrm{O}(n)$. `g` : deux boucles imbriquées parcourent la liste, donc $\mathrm{O}(n^2)$. `h` : n est divisé par 2 à chaque itération, le nombre d'itérations est de l'ordre de $\log_2 n$, donc $\mathrm{O}(\log n)$.
2. Du plus efficace au moins efficace : $\mathrm{O}(1)$, $\mathrm{O}(\log n)$, $\mathrm{O}(n)$, $\mathrm{O}(n \log n)$, $\mathrm{O}(n^2)$, $\mathrm{O}(2^n)$.
3. Pour $n = 10^4$: un algorithme en $\mathrm{O}(n^2)$ effectue 10^8 opérations, soit environ 100 secondes. Un algorithme en $\mathrm{O}(n \log n)$ effectue environ $10^4 \times 14 \approx 1{,}4 \times 10^5$ opérations, soit environ 0,14 seconde. Le second est environ 700 fois plus rapide : c'est cet écart qui justifie, en terminale, le passage au tri fusion ($\mathrm{O}(n \log n)$) à la place des tris quadratiques.

Si tu échoues : reprends le comptage asymptotique (on ne garde que le terme dominant, les constantes multiplicatives disparaissent) et les trois patrons essentiels — une boucle simple donne $\mathrm{O}(n)$, deux boucles imbriquées donnent $\mathrm{O}(n^2)$, une division par deux à chaque tour donne $\mathrm{O}(\log n)$. La différence de comportement pratique entre $\mathrm{O}(n^2)$ et $\mathrm{O}(n \log n)$ est le fil conducteur du cours d'algorithmique de terminale. Fiche à revoir : Complexité algorithmique.