Informatique commune : Conseils pour les concours

Conseils d'ordre général

1. Faire attention aux types des éléments utilisés, renvoyer toujours un élément du type demandé par l'énoncé.
2. Écrire des programmes lisibles (par de ratures, de pâtés, de rayures...) et bien respecter l'indentation ! Après `def, for, if, elif, else, while`, mettre toujours des `:` puis indenter. On peut tracer des lignes verticales afin de mettre en évidence les différents blocs.
3. Ne jamais mettre de `print` dans une fonction pour renvoyer un résultat (sauf si l'énoncé demande explicitement d'imprimer quelque chose à l'écran), les résultats se renvoient avec un `return`, cela permet de composer les fonctions !
4. Pièges classiques : `range(a,b)` parcourt les entiers de a jusqu'à $b-1$, `randrange(a,b)` choisit au hasard un élément entre a et $b-1$.
5. L'énoncé impose parfois des contraintes : écrire une fonction récursive (ou au contraire une fonction non récursive), borne supérieure de la complexité, bibliothèques renommées (`import numpy as np`)... veiller à les respecter !
6. Bien justifier les complexités : ne pas oublier d'étape, de coûts cachés (voir pièges classiques plus loin), détailler la complexité de chaque fonction auxiliaire utilisée.

Programme de première année

1. Savoir décomposer un entier de la base 10 vers la base 2 (décomposition binaire) et réciproquement. Savoir aussi programmer les fonctions correspondantes.
2. Quelques éléments culturels : les entiers se codent en général sur 32 ou 64 bits, 1 octet = 8 bits (connaître aussi les multiples : kO, MO, GO, TO), un ordinateur de bureau fait environ 10^9 opérations par seconde.
3. Savoir montrer qu'une boucle `while` termine (avec un variant de boucle), savoir prouver un invariant de boucle (pour une boucle `for` ou `while`) par récurrence (l'invariant sera donné par l'énoncé).
4. Connaître quelques opérations sur les fichiers : ouvrir un fichier en lecture ou en écriture, écrire du texte, aller à la ligne, et les différentes manières de lire le fichier (`read, readline, readlines`).

Structures de données

1. Ne pas confondre les listes et les tableaux numpy (`np.array`) ! Chaque structure a ses avantages et inconvénients. Une liste est mutable (on peut modifier ses éléments) et dynamique : on peut utiliser par exemple `append` et `pop` pour ajouter ou retirer des éléments. On peut en outre mettre des éléments de types différents dedans. Un tableau numpy est

aussi mutable mais statique : sa taille est fixée dès sa construction et on ne peut le remplir qu'avec des éléments d'un même type. Par contre il est plus adapté aux opérations vectorielles (on peut appliquer directement une fonction à chacun de ses éléments par exemple).

2. Toutes les structures (listes, tableaux, chaînes de caractères...) sont numérotées de 0 à $n-1$ où n est la longueur. Ainsi la commande `L[len(L)]` n'a aucun sens ! Pour ne pas se tromper, sachez que si vous écrivez `n = len(L)`, pour parcourir toute la liste il suffit d'écrire `for i in range(n):` (pas de $n-1$ ou $n+1$). Pour un tableau bidimensionnel, on accède à ses deux dimensions avec `V.shape` (cela renvoie un couple d'entiers), tandis que `len(V)` renvoie le nombre de lignes.

3. Attention à ne pas confondre l'indice d'un élément dans une liste/un tableau et l'élément en lui-même. Il y a deux manières de parcourir une liste:

   ```
   for in in range(len(L)):
   for x in L:
   ```

   Dans le premier cas, on parcourt les indices et on devra appeler `L[i]`, dans l'autre cas on parcourt directement les éléments, on ne doit donc surtout pas écrire `L[x]` dans ce cas ! Il faut choisir le meilleur parcours selon la situation : dans une recherche dichotomique d'un élément par exemple, on manipule les indices (on prend l'indice du milieu, et non pas la moyenne entre les éléments extrémaux) donc on privilégie la première solution.

4. Connaître la complexité des fonctions classiques sur les listes. `L.append(x)` et `L.pop()` sont en coût constant. Pièges classiques : les tests `if x in L` ou `if x not in L` sont en coût linéaire, pareil pour les fonctions `L.insert(i,x)`, `L.pop(i)` et `L.remove(x)`.

5. Les chaînes de caractères et les tuples sont statiques et non mutables !

6. Savoir ce qu'est une pile, les opérations usuelles sur les piles (empiler un élément, dépiler le sommet de pile..) et comment en implémenter une en Python à l'aide d'une liste (il n'est pas nécessaire de connaître la définition de la classe `Pile`). Connaître le coût de chaque opération.

Récursivité

1. Savoir ce qu'est un algorithme récursif, la pile de récursion. Connaître les avantages et inconvénients de la récursivité (faire attention aux doubles appels comme dans le cas de Fibonacci).

2. Connaître quelques algorithmes classiques récursifs : PGCD, exponentiation rapide, coefficients binomiaux, recherche dichotomique, et savoir les programmer.

3. Savoir passer d'un algorithme récursif à un algorithme itératif (avec des boucles) dans des cas simples (factorielle, suite récurrente, PGCD...).

4. Savoir prouver qu'un algorithme récursif se termine (il y a des cas d'arrêt et ils sont toujours atteints) et qu'il est correct (par récurrence). Pas de variant de boucle ou d'invariant de boucle pour les algorithmes récursifs !

5. Pour la complexité, on a souvent une relation de récurrence qui apparaît. Connaître la solution sous forme d'un O des récurrences suivantes : $C(n) = C(n-1) + 1$, $C(n) = C(n-1) + n$, $C(n) = C(n/2) + 1$, $C(n) = 2 * C(n/2) + n$.

Algorithmes de tris

1. Connaître quelques situations où il est mieux d'avoir une liste triée (recherche de la médiane, recherche dichotomique, suppression des doublons).
2. Connaître les algorithmes de tris par insertion, fusion et rapide. Savoir les programmer (pour les tris par insertion et rapide, on les programme en place, c'est-à-dire sans créer de nouvelle liste). Les tris fusion et rapide font appel à des fonctions récursives.
3. Connaître pour chaque algorithme de tri la complexité dans le pire des cas, le meilleur des cas et en moyenne. Connaître les avantages et inconvénients de chaque tri (aucun n'est parfait !)

Analyse numérique

1. Savoir tracer une courbe à partir des vecteurs des abscisses et des ordonnées.
2. Savoir programmer la méthode des rectangles/des trapèzes pour le calcul approché d'une intégrale.
3. Savoir programmer la recherche du zéro d'une fonction par dichotomie (savoir justifier l'existence de ce zéro sous les bonnes hypothèses) et connaître une approximation de l'erreur commise au bout de n étapes, et combien d'étapes sont nécessaire pour avoir une précision $< \varepsilon$.
4. Savoir programmer la méthode de Newton. Connaître le théorème donnant une condition suffisante de convergence et l'estimation de l'erreur dans ce cas (on a une convergence quadratique donc meilleure que pour la dichotomie).
5. Savoir programmer les différentes étapes du pivot de Gauss et connaître les complexités.
6. Savoir programmer la méthode d'Euler pour les équations différentielles.
7. Il est bon de connaître aussi les fonctions intégrées à Python (pour les oraux de Centrale notamment).

Bases de données

1. Connaître le principe d'une base de données, les définitions de base : table, attribut, enregistrement, clé primaire.
2. Connaître les commandes de base : `SELECT, WHERE, ORDER BY...`
3. Savoir réaliser une jointure (et savoir ce que cela signifie, on fait un produit cartésien) : `JOIN ... ON ...`
4. Connaître les fonctions d'agrégation : `COUNT, SUM, MAX, MIN, AVG`. Après un `GROUP BY`, si l'on veut imposer une condition sur le groupe on utilise `HAVING`.
5. Il est bon de connaître aussi les opérations correspondantes en algèbre relationnelle : π pour une projection (correspond à `SELECT`), σ pour une sélection (correspond à `WHERE`), ρ pour un renommage (correspond à `AS`), $\bowtie$ pour une jointure.