Informatique — PCSI, 2021–2022

Corrigé du concours blanc

Exercice 1 – Parcours en profondeur et algorithme de Dijkstra

Question 1.1. L'algorithme de parcours en profondeur fournit le tableau suivant :

Pile	Sommets déjà marqués
A	A
B/A	A, B
$D/B/A$	A, B, D
$C/D/B/A$	A, B, D, C
$E/C/D/B/A$	A, B, D, C, E
$F/E/C/D/B/A$	A, B, D, C, E, F
$E/C/D/B/A$	A, B, D, C, E, F
$C/D/B/A$	A, B, D, C, E, F
$D/B/A$	A, B, D, C, E, F
B/A	A, B, D, C, E, F
A	A, B, D, C, E, F
	A, B, D, C, E, F

(avec ces notations, le sommet le plus à gauche de la pile est celui qui se trouve en haut). Comme l'énoncé donne des listes d'adjacences triées par ordre alphabétique, c'est ce critère qui joue en cas de choix à faire entre deux sommets.

Question 1.2. L'algorithme de Dijkstra fournit le tableau suivant (en indiquant entre parenthèses le nom du sommet ayant occasionné la dernière modification d'étiquette) :

A	B	C	D	E	F
0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
0	$2(A)$	$+\infty$	$5(A)$	$+\infty$	$+\infty$
0	2	$+\infty$	$4(B)$	$+\infty$	$+\infty$
0	2	$8(D)$	4	$+\infty$	$5(D)$
0	2	8	4	$7(F)$	5

Question 1.3. Le tableau de la question précédente permet de reconstruire à l'envers les plus courts chemins :

$A \to B \to D \to C$ est un plus court chemin de A à C.

$A \to B \to D \to F \to E$ est un plus court chemin de A à E.

Exercice 2 – Correction et complexité des algorithmes

Question 2.1.

— : Le précondition exprime juste le fait que n est un nombre d'étages et que l'immeuble a au moins un étage.

$P(n)$: n est un entier supérieur ou égal à 1.

— Pour la postcondition, il faut séparer deux cas : celui où la calculatrice n'est pas cassée à l'étage n et où l'algorithme retourne $n+1$, et celui où l'algorithme retourne le plus petit étage tel que la calculatrice est cassée. Si r est le résultat retourné par l'algorithme, cette post-condition peut être exprimée de la manière suivante.

$Q(n, r)$: (La calculatrice n'est pas cassée à l'étage n et $r = n+1$) ou (La calculatrice est cassé à l'étage r et n'est pas cassée à l'étage $r-1$)

Question 2.2. La terminaison de l'algorithme se montre simplement en prenant $n - i$ comme variant de boucle, où n est le numéro de l'étage ; $n - i$ ne peut être inférieur strictement à zéro en raison de la condition d'arrêt et $n - i$ diminue de 1 à chaque passage dans la boucle.

Question 2.3. Pour montrer la correction, il faut montrer que l'invariant de boucle suivant est vrai en fin de boucle : $\forall j \in [\![0; i-1]\!]$, la calculatrice n'est pas cassée à l'étage j.

Si $i = 0$, l'ensemble $[\![0; i-1]\!]$ est vide donc la condition est vérifiée.

Supposons que la condition soit vérifiée en début de boucle. À la fin de la boucle, i a pris la valeur $i+1$. D'après la condition d'arrêt, la boucle d'indice n'est par ailleurs exécutée que si la calculatrice n'est pas cassée à l'étage i. La condition $\forall j \in [\![0; (i+1)-1]\!]$ la calculatrice n'est pas cassée à l'étage j est donc vérifiée en fin de boucle.

Examinons maintenant la correction de l'algorithme. Celui s'arrête si $i > n$ ou la calculatrice est cassée à l'étage i. Dans le premier cas, il retourne $n+1$ et la première partie de la postcondition est vérifiée. Dans le second cas, il retourne i : la calculatrice est donc cassée à l'étage i, et, d'après la condition d'arrêt, elle n'est pas cassée à l'étage j pour tout $j \in [\![0; i-1]\!]$. La seconde partie de la postcondition est donc vérifiée.

Dans tous les cas, la postcondition est donc vérifiée.

En définitive, la postcondition $Q(n, r)$ est vérifiée donc l'algorithme est correct.

Question 2.4. Pour déterminer l'étage n à partir duquel les lancers sont fatals, il faut $\boxed{S(n) = \mathcal{O}(n)}$.

Question 2.5. Pour la première calculatrice, on fait au pire $\dfrac{n}{\lfloor\sqrt{n}\rfloor} = \mathcal{O}(\sqrt{n})$ essais.

Pour la deuxième, on fait au pire $\lfloor\sqrt{n}\rfloor = \mathcal{O}(\sqrt{n})$ essais.

Donc, par somme, il faut au pire $\dfrac{n}{\lfloor\sqrt{n}\rfloor} + \lfloor\sqrt{n}\rfloor \approx 2\lfloor\sqrt{n}\rfloor$ soit $\boxed{\mathcal{O}(\sqrt{n})}$ essais.

Question 2.6. On prend comme variant la largeur de l'intervalle $[\![m, n]\!]$, de cardinal $\text{Card}([\![m, n]\!]) = n - m + 1$.

— si $m \neq n$, on distingue deux cas :

— si la calculatrice est cassée à l'étage $p = \left\lfloor \dfrac{m+n}{2} \right\rfloor$, on appelle la fonction sur l'intervalle $[\![m, p]\!]$;

— sinon on appelle la fonction sur l'intervalle $[\![p, n]\!]$;

Ces deux intervalles sont deux fois plus petits que celui de départ $[\![m, n]\!]$.

— si $m = n$, l'intervalle contient un élément et correspond au dernier appel, qui correspond à l'étage fatal.

Donc l'algorithme termine.

Question 2.7. Comme l'algorithme de recherche par dichotomie consiste à diviser par deux l'intervalle à chaque fois, alors pour $n = 2^p$, il faut p lancers. Ainsi, en doublant l'intervalle (passage de $n = 2^p$ à $2n = 2^{p+1}$), il faut simplement rajouter un essai, d'où $S(2n) = S(n) + 1$.

Posant $S(1) = 1$ (un étage, donc un lancer), il vient par propriétés des suites arithmétiques :

$$\big(\forall k \geqslant 1, \quad S(2^{k+1}) = S(2^k) + 1\big) \iff \big(\forall k \geqslant 1, \quad S(2^k) = k + 1\big) \implies \boxed{S(n) = S(2^p) = p + 1}$$

Question 2.8. Ainsi, on a $\boxed{S(n) = \mathcal{O}\left(\log(n)\right)}$.

Exercice 3 – Permutations et tri par dénombrement

Question 3.1. Une liste de n éléments $(L_i)_{0 \leqslant i < n}$ est une progression si, et seulement si,

$$L_0 = 0 \qquad \text{et} \qquad \forall i \in [\![0, n-2]\!], \quad L_{i+1} = L_i + 1$$

ce que l'on implémente sous la forme :

```
def progression(L):
    i=0
    if L[i]==0:
        while i<len(L)-1 and L[i+1]==L[i]+1:
            i+=1
    return(i==len(L)-1)
```

Question 3.2. Pour déterminer si l'entier `p` est dans la liste `L`, il suffit de la parcourir et de s'arrêter dès qu'il est trouvé ou, à la fin de cette dernière. Ce qui se traduit par : « *tant que* `p` *n'est pas trouvé et qu'il reste des éléments à regarder, on essaie le suivant* ». Ce que l'on implémente directement :

```
def appartient(L,p):
    i=0
    while i<len(L) and L[i]!=p:
        i+=1
    return(i<len(L))
```

Question 3.3. Sachant qu'une permutation d'ordre n doit contenir tous les nombres de $[\![0, n-1]\!]$, il suffit de le vérifier successivement par appel de la fonction `appartient`. Pour éviter tout calcul inutile, on arrêtera la boucle au premier `False` témoin de non appartenance. Il vient alors le code suivant :

```
def permutation(L):
    i=0
    while i<len(L) and appartient(L,i):
        i+=1
    return(i==len(L))
```

Question 3.4. On note n la longueur de la liste `L`. La boucle `while` de la fonction `permutation` est réalisée au pire n fois et, dans chaque cas, la boucle `while` de la fonction `appartient` est aussi réalisée au pire n fois. On obtient alors par produit, dans le pire des cas, $\boxed{\mathcal{O}(n^2)}$.

Question 3.5. Pour utiliser la liste `presence` avec une liste de n éléments $(L_i)_{0\leqslant i<n}$, il suffit, pour tout indice $i \in [\![0, n-1]\!]$, d'affecter à l'indice L_i la valeur vrai, tout en sachant que L_i doit être inclus dans $[\![0, n-1]\!]$. Pour éviter tout calcul inutile, on s'arrêtera au premier indice L_i non adapté. Une fois les éléments de L parcourus, il s'agit de vérifier que `False` n'est pas dans la liste presence, ce que l'on traduit avec les fonction précédentes par :

```
not(appartient(presence,False))
```

que l'on met en sortie en conjonction avec `i==len(L)` pour ne faire cette recherche que si aucun indice inadapté L_i n'a été trouvé. Il vient alors le code suivant :

```
def permutationR(L):
    presence = [False for i in range(len(L))]
    i=0
    while i<len(L) and L[i]>=0 and L[i]<len(L):
        presence[L[i]]=True
        i+=1
    return(i==len(L) and not(appartient(presence,False)))
```

La complexité de cette fonction est bien, dans le pire des cas, en $\boxed{\mathcal{O}(n)}$. En effet, il faut n opérations pour la boucle `for` permettant de définir la liste `presence`, n passages dans la boucle `while` au pire des cas et n passages dans la boucle `while` de la fonction `appartient` aussi dans le pire des cas. Aucune de ces boucles n'est imbriquée, donc il vient par somme $3n = \mathcal{O}(n)$.

Notez qu'en utilisant un dictionnaire au lieu d'une liste pour la présence, avec une fonction de la forme :

```
def permutationD(L):
    d={i:False for i in range(len(L))}
    for i in range(len(L)):
        d.pop(L[i], None)
    return(len(d)==0)
```

on aurait une complexité de $2n = \mathcal{O}(n)$, ce qui permet de gagner 1/3 de temps...

Question 3.6. Pour construire l'histogramme, il faut faire attention au fait que l'on cherche à remplir une liste de longueur $N =$ `bsup` $-$ `binf`, d'indices compris dans $[\![0, N]\!]$ avec des entiers de $[\![$`binf`, `bsup`$]\!]$ et donc au décalage de $-$`binf`. On remplit donc l'histogramme avec le code :

```
for e in tab:
    histo[e-binf]+=1
```

Une fois construit l'histogramme, il suffit d'affecter successivement le bon nombre d'occurrences des entiers de $[\![$`binf`, `bsup`$]\!]$ à `tab` pour ne pas augmenter la complexité spatiale et ainsi faire le tri « en place ». Faisant de nouveau attention au décalage de $+$`binf`, il vient le code suivant.

```
def tri_comptage(tab, binf, bsup):
    histo = [0 for _ in range(bsup - binf + 1)]
    # on remplit l'histogramme
    for e in tab:
        histo[e-binf]+=1
    # on reconstitue le tableau
    i=0
    for e,n in enumerate(histo):
        for j in range(n):
            tab[i]=e+binf
            i+=1
```


On note n la longueur de `tab`. Avec 2 boucle de n termes, la complexité en temps de cet algorithme est linéaire, soit en $\mathcal{O}(n)$. La complexité en espace est en $n + N$.

Question 3.7. On peut définir une fonction qui vérifie si la liste donnée en argument est une permutation et renvoie la progression si c'est le cas. Le code suivant convient.

```
def regression(L):
    if permutation(L):
        tri_comptage(L,0,len(L)-1)
```

Par somme de deux fonctions de complexité en $\mathcal{O}(n)$, la complexité est bien en $\boxed{\mathcal{O}(n)}$.

Exercice 4 – Contrôle de parité croisé

Question 4.1. Pour définir la fonction `Parite` de façon récursive, il suffit de partir du principe que, pour deux bits a et b elle correspond au ou exclusif $a \oplus b$, c'est-à-dire l'union (cas $a + b \geqslant 1$) moins l'intersection (cas $a + b = 2$), que l'on peut implémenter sous la forme `(a+b)%2`. Ainsi, il vient :

$$\forall N \in \mathbb{N}, \quad \forall k \in [\![0, N]\!], \quad \text{Parité}\left((b_i)_{i\in[\![k,N]\!]}\right) = \begin{cases} b_k & \text{si } k = N \\ b_k \oplus \text{Parité}\left((b_i)_{i\in[\![k+1,N]\!]}\right) & \text{sinon.} \end{cases}$$

que l'on implémente sous la forme :

```
def parite(M):
    if len(M)==1:
        return(M[0])
    else:
        return((M[0]+parite(M[1:]))%2)
```

```
def parite(M):
    if len(M)==1:
        return(M[0])
    else:
        return(M[0]^parite(M[1:]))
```

Dans cette seconde version on utilise l'opérateur python `^` du *ou exclusif (XOR)* qui fonctionne bit à bit.

Question 4.2. Avant toute chose, la fonction `LRC` doit renvoyer un tableau (variable `S`) contenant autant de lignes mais une colonne de plus que le tableau `T`. Après avoir recpoié le tableau `T` avec la fonction `deepcopy` du module `copy`, on lui ajoute sur chaque ligne les bits de LRC par simple appel de la fonction `Parite`. Le code suivant convient.

```
import copy
def LRC(T):
    S=copy.deepcopy(T)
    for i in range(len(T)):
        S[i].append(parite(T[i]))
    return(S)
```

que l'on peut aussi écrire directement et beaucoup plus simplement :

```
def LRC(T):
    return([[e for e in L]+[parite(L)] for L in T])
```

où `L` du `L in T` désigne chaque ligne et où le `[e for e in L]` permet de recopier chaque ligne sans lier les éléments (ce que fait `deepcopy` à tous niveaux).

Question 4.3. Évidemment, en ayant remarqué que dans le cas du VRC on agit sur les colonnes de la même façon que le LRC agit sur les lignes, il suffit d'appeler la fonction `LRC` sur la transposée du tableau et de renvoyer la transposée du tableau résultat. On rappelle que transposer une matrice $T \in \mathcal{M}_{p,q}(\mathbb{K})$, avec $(p, q) \in \mathbb{N}_*^2$ consiste à définir une nouvelle matrice $S \in \mathcal{M}_{q,p}(\mathbb{K})$ telle que :

$$\forall i \in [\![1, p]\!], \quad \forall j \in [\![1, q]\!], \quad S_{ji} = T_{ij}$$

ce que l'on implémente sous la forme d'une fonction `transpose` avec une simple définition par compréhension :

```
def transpose(T):
    return([[T[i][j] for i in range(len(T))] for j in range(len(T[0]))])
```

et telle que la fonction `VRC` s'écrive alors :

```
def VRC(T):
    return(transpose(LRC(transpose(T))))
```

Question 4.4. Pour extraire le tableau de mots binaires initiaux, il suffit de faire une coupe pour garder toutes les lignes sauf la dernière et toutes les colonnes sauf la dernière. Le code suivant convient.

```
def extraire(T):
    return([L[:-1] for L in T[:-1]])
```

Question 4.5. Pour détecter s'il y a eu des erreurs de transmission, il est nécessaire de comparer le LRC et le VRC du bloc reçu à ceux issus de la reconstruction à partir de la composition `VRC∘LRC∘extraire`. Partant de ce point :

- il n'y pas d'erreur si :
  - le LRC et le VRC sont identiques ;
  - si le LRC présente un bit de différence mais pas le VRC (l'erreur ne porte que sur le LRC et pas sur le message) ;
  - si le VRC présente un bit de différence mais pas le LRC (l'erreur ne porte que sur le VRC et pas sur le message).
- il y a exactement une erreur si le LRC présente un bit de différence (indiquant la position du bit erroné dans le mot) et si le VRC présente 2 bits de différence : celui associé au mot binaire erroné et celui du LRC lui aussi erroné ;
- on ne peut pas conclure quant à l'intégrité du message si le LRC ou le VRC (sans le VRC de LRC) ont au moins de 2 bits de différence.

Dans le cas où il n'y a qu'une seule erreur, elle concerne le mot binaire associé au premier bit différent du VRC et le bit indiqué par le bit de différence du LRC.

Pour définir en pratique une fonction `decode`, il est nécessaire de :

- reconstruire un tableau codé à partir de la composition `VRC∘LRC∘extraire` ;
- contrôler le LRC, en comparant le dernier élément de chacune des lignes des deux tableaux et de garder en mémoire les différences ;
- contrôler le VRC en comparant le dernier élément de chacune des colonnes des deux tableaux et de garder en mémoire les différences ;
- corriger si besoin le tableau extrait.

Si une seule erreur est détectée, le LRC aura un seul bit faux, correspondant au bit du mot erroné. La position de ce mot dans le tableau correspond à l'indice du premier bit faux du VRC, le second étant le VRC du LRC dans le cas d'une seule erreur. Il est alors possible de corriger l'erreur. En notant e le bit erroné, nous avons ici défini le bit corrigé par le reste de la division euclidienne de $(e+1)$ par 2, noté ici `(E[l,c]+1)%2`.

Question 4.6. Le code suivant est une implémentation correcte de la fonction `decode` d'après la description que nous venons d'en faire dans la question précédente.

```
def decode(T):
    E=extraire(T)
    N=VRC(LRC(E))
    nl,nc=len(T),len(T[0])
    ne=[[],[]]
    # contrôle du LRC
    for l in range(nl-1):
        if T[l][nc-1]!=N[l][nc-1]:
            ne[0].append(l)
    # contrôle du VRC
    for c in range(nc):
        if T[nl-1][c]!=N[nl-1][c]:
            ne[1].append(c)
    # nombre d'erreurs & correction
    if len(ne[0])<=1 and len(ne[1])<=1:# pas d'erreur
        S=E
    elif len(ne[0])==1 and len(ne[1])==2 and ne[1][1]==nl-2:# une erreur
        l,c=ne[0][0],ne[1][0]
```



```
        S=E
        S[l][c]=(E[l][c]+1)%2
    else:# au moins deux erreurs
        S=False
    return(S)
```

Notez enfin que si, par malchance, les seuls trois bits faux sont un bit sur le LRC et deux sur le VRC (dont le dernier), alors le message qui malgré tout était correct sera décodé sans problème mais avec un bit d'erreur. La probabilité que cet évènement se produise est peut être faible mais belle et bien non nulle : le contrôle de parité croisé n'est donc pas une méthode infaillible !

— Fin du corrigé —