Correction du partiel

AP3 Algorithmique et programmation — L2 mathématiques

Automne 2016

Exercice 1.

(a)

```
def occ(li,x):
  cmpt = 0 # initially 0
  for y in li:
    # add 1 if y is actually x
    cmpt += 1 if x == y else 0
  return cmpt
```

(b)

```
def somme(li):
  s = 0 # sum of the empty list
  for x in li: s+=x # add each elt of li to s
  return s
```

(c)

```
def listeocc(ll,x):
  occs = []
  # for each list inside ll, use question (a)
  for li in ll: occs.append(occ(li,x))
  return occs
```

(d)

```
def totalocc(ll,x):
  return somme(listeocc(ll,x))
```

Exercice 2. On suit juste la définition mathématique récursive :

```
def mc(n):
  if n > 100: return n - 10
  else: return mc(mc(n + 11))
```

Exercice 3.

(a) La méthode propose ici de trier la liste entière, ce que l'on sait utiliser $\mathcal{O}(n \ln(n))$ comparaisons. Sélectionner les k derniers élements de la liste une fois triée n'effectue pas de comparaisons en plus, donc l'algorithme total fait $\mathcal{O}(n \ln(n))$ comparaisons.

(b) On fait bien attention : il s'agit de renvoyer l'*indice* d'un maximum, et non le maximum lui-même. On commence par identifier le cas de base, qui est le cas des listes de longueur 1 : dans ce cas, le maximum se situe nécessairement dans l'unique case de la liste, indicée par 0. Sinon, pour une liste `li` arbitraire, on recherche résursivement l'indice $i_{\max}$ d'un maximum dans la liste `li[1:]`. Ce maximum de `li[1:]` se situe donc à la case indicée par $i_{\max}+1$ dans la liste totale `li` : il suffit donc de comparer `li[0]` avec `li`[$i_{\max}$] et de renvoyer l'indice du plus grand des deux.

```
def recmaxi(li):
  # the maximum of a length 1 list is it unique element
  if len(li) == 1: return 0
  # otherwise (hereditary case):
  imax = recmaxi(li[1:]) # index in li[1:]
  imax += 1 # index in li of the potential maximum
  # now return index of max of li[0] and li[imax]
  return 0 if li[0] > li[imax] else imax
```

(c) Il suffit ici d'appliquer les mêmes idées que dans le tri par sélection mais de s'arrêter dès qu'on a selectionné k maximums successifs : on recherche l'indice d'un maximum avec `recmaxi` ; puis on échange cette position avec la dernière case de la liste ; on répète enfin l'opération avec la liste privée de sa dernière case ; etc. jusqu'à avoir répéter l'opération k fois. À ce moment, les k plus grands éléments de la liste sont les k derniers éléments de la liste, trié dans l'ordre croissant, que l'on retourne donc.

```
def maxi(li,k):
  last = len(li) - 1 # last index in li
  while k > 0:
    imax = recmaxi(li[:last+1]) # look for imax up to index
        last
    li[imax],li[last] = li[last],li[imax] # swap
    last -= 1 # new last index
    k -= 1
  return li[last+1:] # from the last written case to the end
```

(d) On cherche $\gamma(n\,,k)$ le nombre de comparaisons effectuées par la fonction `maxi(li,k)` quand `li` est de longueur n. Comme aucune comparaison n'est faite en dehors de la boucle `while`, on obtient :

$$\gamma(n,k)=\sum_{i=0}^{k-1}\delta(k-i)$$

où $\delta(j)$ est le nombre de comparaisons faites par la boucle quand `k` a la valeur j.

Bien entendu, $\delta(j)$ est entièrement déterminé par l'appel à la fonction `recmaxi` qui est la seule susceptible de faire des comparaisons. Notons $\rho(j)$ le nombre de comparaisons effectuées par `recmaxi` sur une liste de taille j. Alors, dans la boucle `while`, si `k` a pour valeur $k-i$, la liste `li[:last+1]` a une taille $n-i$. Ainsi :

$$\delta(k-i)=\rho(n-i)$$

et donc on a une expression de $\gamma(n\,,k)$ comme :

$$\gamma(n,k) = \sum_{i=0}^{k-1} \rho(n-i).$$

Reste alors à determiner ρ. C'est la complexité d'une fonction définie par récurrence : l'étude de la fonction devrait nous donner une relation de récurrence sur ρ qu'on cherchera alors à résoudre. Le nombre de comparaisons faites par `recmaxi(li)` est 0 si `li` est de longueur 1 ou celui nécessaire à l'obtention de `recmax(li[1:])` augmenté de 1 (la comparaison faite lors du `return`). C'est-à-dire :

$$\rho(1) = 0, \qquad \forall n > 1,\ \rho(n) = 1 + \rho(n-1)$$

C'est une suite arithmétique vite résolue : $\forall n \geq 1\,, \rho(n) = n-1$.

On peut enfin déterminer la complexité γ : $\forall n \in \mathbb{N}\,, \forall k \leq n$,

$$\gamma(n,k) = \sum_{i=0}^{k-1} n - i - 1 = k(n-1) - \frac{k(k-1)}{2}$$

On peut laisser le résultat sous cette forme, ou bien donner une forme plus asymptotique comme $\mathcal{O}(nk)$. En particulier, si $k = n$ (ce qui revient à trier la liste complètement), on retrouve la complexité $\mathcal{O}(n^2)$ du tri par sélection.

Remarque : la correction est ici très détaillée ; une réponse moins complète était très probablement acceptée.

(e) L'idée est la suivante :

- on commence par trier les k dernières cases de la liste `li` avec un tri avancé (fusion, ou tas), qui fait $\gamma_{\text{pre}}(n\,,k) = \mathcal{O}(k\ln(k))$ comparaisons ; à ce moment, `li[n-k:]` est triée dans l'ordre croissant,
- puis, pour i allant de 1 à $n-k$, on place par dichotomie l'élément `li[n-k-i]` dans `li[n-k:]` ; la liste `li[n-k:]` sur laquelle agit la dichotomie étant de taille k, on effectue $\gamma_i(n\,,k) = \mathcal{O}(\ln(k))$ comparaisons ; après cette opération, `li[n-k:]` est la liste triée des k plus grand éléments de `li[n-k-i:]`,
- en particulier quand $i = n-k$, `li[n-k:]` est la liste triée des k plus grand éléments de `li` tout entière.

Le nombre total de comparaisons est donc : $\forall n \in \mathbb{N}\,, \forall k \leq n$,

$$\begin{aligned}\gamma(n,k) &= \gamma_{pre}(n,k) + \sum_{i=1}^{n-k} \gamma_i(n,k) \\ &= \mathcal{O}(k\ln(k)) + \mathcal{O}((n-k-1)\ln(k)) \\ &= \mathcal{O}(n\ln(k))\end{aligned}$$

L'énoncé ne demande pas d'aller plus loin ! Pour la correction néanmoins, voici une implémentation `pyhton` de cette idée :

```python
# returns the index before which x should be inserted
def dich(li,start,end,x):
  if start > end: return start
  else:
    mid = (start + end) / 2
    if li[mid] == x: return mid
    elif li[mid] > x: return dich(li,start,mid-1,x)
    else: return dich(li,mid+1,end,x)

def maxi(li,k):
  # sort the last k element: O(k ln(k))
  n = len(li)
  li = li[:n-k] + sorted(li[n-k:])
  # insert each of the remaining element by dich
  for i in range(1,n-k+1):
    pos = dich(li,n-k,n-1,li[n-k-i])
    li = li[:n-k-i] + li[n-k-i+1:pos] + [li[n-k-i]] + li[pos:]
    # invariant here is: li[n-k:] is the sorted list of the k
        biggest elements of li[n-k-i:]
  return li[n-k:]
```

Exercice 4. Cet algorithme ne trie pas correctement. Le contre-exemple le plus simple est de prendre une liste de longueur $n = 2$ non triée, par exemple `[2,1]`. Dans ce cas, `range(n/2-1)` est la liste vide et le code interne à la boucle sur `k` n'est jamais exécuté. La liste de subit donc aucun changement et reste `[2,1]`, non triée, après l'appel à `TriBizarre`.

Quand à ce que fait l'algorithme, il trie par la méthode du tri bulle, simultanément mais indépendamment la sous-liste de `L` composée des éléments d'indices pairs et celle composée des éléments d'indices impairs.

Exercice 5.

(a) On suit la définition récursive mathématique. Il y a une petite subtilité : l'appel récursif à la fonction `det` sur les matrices de la forme $\widetilde{M_{i,k}}$ doit indiquer la colonne sur laquelle développer : la matrice $\widetilde{M_{i,k}}$ ayant une colonne de moins que M, le seul choix sûr est de développer sur la 1^{re} colonne.

```python
def det(M,k):
  d,n = 0,len(M)
  if n == 1: return M[0][0] # base case
  # otherwise use the recursive definition:
  for i in range(n):
    s = (1 if (i+k)%2 == 0 else -1) # (-1)^(i+k)
    # recursive call on tilde(M_i,k) along column 0
    dtilde = det([row[:k]+row[k+1:]
                  for row in M[:i]+M[i+1:]],0)
    d += s*M[i][k]*dtilde
  return d
```

(b) Notons $\gamma(n)$ la complexité de `det(M,k)` quand `M` est de taille $n \times n$.

- $\gamma(1)$ est juste le coût d'accès à la case mémoire `M[0][0]`, donc constant : $\gamma(1) = \mathcal{O}(1)$.

- supposons maintenant $n > 1$, alors

$$\gamma(n) = \sum_{i=0}^{n-1} \delta(n, i)$$

où $\delta(n\,, i)$ est la complexité du code interne à la boucle au tour i. Celle-ci est facile à déterminer, car outre les affectations et les opérations arithmétiques, toutes en temps constant, la seule opération est l'appel à `det` sur une matrice de taille $(n-1)\times(n-1)$: ainsi $\delta(i\,, n) = \gamma(n-1)+C$ avec C une constante. Ainsi, on trouve donc

$$\forall n \in \mathbb{N},\ \gamma(n) = \sum_{i=0}^{n-1} (\gamma(n-1) + C) = n\gamma(n-1) + nC.$$

Cette relation de récurrence est vite résolue : $\gamma(n) = \mathcal{O}(n\,!)$.

(c) Dans le calcul du déterminant de M, tous les appels récursifs au déterminant de $\widetilde{M}_{i,k}$ avec $m_{i,k} = 0$ sont en fait inutiles. Ainsi, plus la colonne k a de zéros, plus le calcul du déterminant est rapide. En tenant compte de cette remarque, on peut modifier notre code comme suit :

```
def det_better(M,k):
  d,n = 0,len(M)
  if n == 1: return M[0][0] # base case
  # otherwise use the recursive definition:
  for i in range(n):
    if M[i][k] != 0: # compute only if m_i,k is not 0
      # same as before, only with det_better
      s = (1 if (i+k)%2 == 0 else -1)
      dtilde = det_better([row[:k]+row[k+1:]
                           for row in M[:i]+M[i+1:]],0)
      d += s*M[i][k]*dtilde
  return d
```

Pour aller encore plus loin, on pourrait sélectionner à chaque appel récursif la colonne avec le moins de zéros dans $\widetilde{M}_{i,k}$ (au lieu de la colonne 0 choisie par défaut). Mais cela supposerait de pouvoir déterminer le nombres de zéros d'une colonne rapidement, nous emmenant bien au-delà de cet exercice.