Algorithmique Avancée (IN423) – Examen Session 1 2025-26

Durée totale : 1 heure 30 / Tiers Temps = 2 heures

Toute communication (orale, téléphonique, par messagerie, etc.) avec les autres étudiant·e·s est interdite. Aucun document autorisé (la fiche de syntaxe C est inutile).

— Il est **indispensable de rédiger vos réponses, et d'expliquer vos programmes.** Un programme non expliqué n'aura pas la totalité des points.

— Les questions ne **SONT PAS** de difficulté croissante.

— Cet examen est **très probablement trop long**, le barème en tiendra compte. Les points donnés dans les questions (pour un total de 20 points) sont des indications de pondération respective (difficulté, longueur) de chaque question.

— Les 2 parties de ce sujet sont indépendantes, et à peu près de longueur et points identiques. Il est demandé de **composer sur 2 copies séparées**.

Solution:
Le barème finalement utilisé est celui mentionné. Correction des copies par IP et YK cette année.

Note *Ce sujet d'examen est inspiré d'exercices tirés de l'épreuve de l'option Informatique de la filière MP du concours Centrale-Supelec de l'année 2025, et de l'épreuve d'informatique commune du concours Mines-Ponts de l'année 2025.*

1 Autour d'une structure de données : le *tas*

Rappels sur les listes chaînées Dans cet exercice, pour manipuler le type de données listes présenté en cours, on suppose données 5 opérations `liste_vide, est_liste_vide, ajoute, premier, enleve`.

La liste `(1, 2, 3)` peut être construite par l'expression :

```
ajoute(1, ajoute(2, ajoute(3, liste_vide()))).
```

On peut décrire le type "liste d'objets de type a" par :

```
type Liste<a> = La_liste_vide | Cellule(a, Liste<a>)
```

Si on veut manipuler des listes d'entiers, il suffit de prendre "a = Int", et ainsi on peut parler du type `Liste<Int>`.

Arbres n-aires, et tas Un arbre *n*-aire (cf. Figure 1) est un arbre dont chaque nœud peut avoir un nombre arbitraire d'enfants.

La structure de données considérée dans cette partie, un cas particulier d'abre *n*-aire sera nommée un *tas*.

Comme pour les autres arbres, il existe un arbre vide, construit par l'opération `tas_vide`. Les valeurs sont portées par les nœuds de l'arbre. Dans cet exercice, les valeurs seront des entiers.

Les enfants d'un nœud sont rangés dans une liste : ces enfants sont eux-mêmes des arbres, mais par convention, on ne met jamais l'arbre vide dans cette liste d'enfants

On décrit dans la suite ce type par la déclaration suivante :

```
type Tas = Le_tas_vide | Noeud(Int, Liste<Tas>)
```

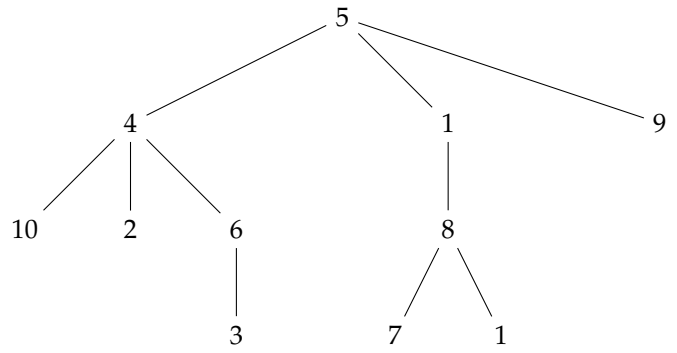

FIGURE 1 – Exemple d'arbre n-aire

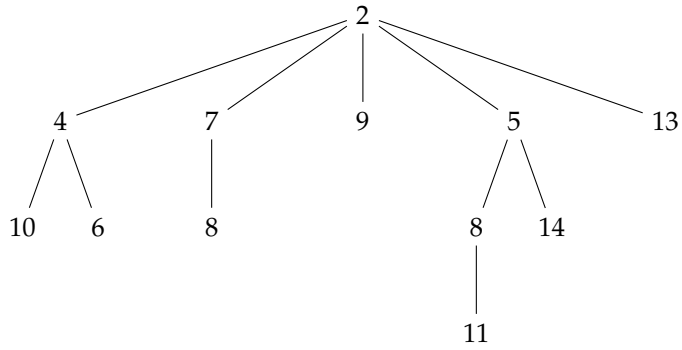

FIGURE 2 – Exemple de tas

Pour qu'un arbre n-aire soit un **tas**, il doit vérifier en plus la condition suivante : la valeur portée par un enfant d'un nœud doit être supérieure ou égale à la valeur portée par le nœud. Ceci doit être valable pour chaque enfant de chaque nœud. C'est le cas dans l'exemple décrit dans la Figure 2.

En termes d'opérations du type de données tas, outre l'opération/le constructeur `tas_vide` déjà mentionnée, on suppose disposer d'une opération `tas_est_vide` qui détermine si le tas passé en paramètre est vide (retournant un booléen) ; d'une opération `tas_noeud` prenant un entier, une liste de tas, et qui retourne le tas dont l'entier est à la racine, et la liste des enfants est la liste de tas passée en paramètre ; d'une opération `tas_valeur` qui renvoie la valeur (de la racine) d'un tas non-vide ; et d'une opération `tas_enfants` qui renvoie la liste des enfants d'un tas non-vide.

Remarque importante En raison de la propriété structurelle des tas, il est possible de construire avec `tas_vide` et `tas_noeud` des arbres n-aires qui ne sont pas des tas.

L'objet des questions suivantes est d'étudier un certain nombre d'opérations sur les tas et leur complexité algorithmique. On écrira du pseudo code algorithmique basé sur la structure des objets passés en paramètre, sur le modèle de la question suivante.

Question #1 ($\frac{1}{2}$ point)

Compléter la fonction suivante, qui calcule et renvoie la taille d'une liste d'entiers.

```
def longueurliste mylist:Liste<int> : int
  | La_liste_vide => ...
  | Cellule(x,l2) => ...
```

Solution:

```
def longueurliste mylist:Liste<int> : int
  | La_liste_vide => 0
  | Cellule(a,l2) => 1+ longueurliste(l2)
```

Question #2 ($\frac{1}{2}$ point)

Dessiner le tas construit par l'expression suivante :

```
tas_noeud(4,
          ajoute(tas_noeud(7, liste_vide()),
          ajoute(tas_noeud(5,
                    ajoute(tas_noeud(9, liste_vide()),
                    ajoute(tas_noeud(6, liste_vide()),
                    ajoute(tas_noeud(13, liste_vide()),
                    liste_vide()))),
                    ),
          liste_vide()))
          )
```

Solution: ...

Question #3 (1 point)

Combien y a-t-il d'arbres non-vides dans l'expression de la question précédente ?

Solution: Il y en a 6 - autant que d'entiers !

Question #4 (1 point)

Ecrire une fonction `tas_min()`, qui prend en paramètre un tas supposé non-vide, et retourne sa plus petite valeur.

Solution:

```
def tas_min montas:Tas : int
  | Le_tas_vide => "should not happen"
  | Noeud(x,_) => x
```

Question #5 (1 point)

Quelle est la complexité pire cas de la fonction que vous avez proposé à la question précédente ?

Solution: Constante : $O(1)$.

Il est possible de fusionner deux tas en un seul en appliquant la stratégie suivante. Si un des deux tas est un tas vide, le résultat de la fusion est l'autre tas. Lorsque les deux tas sont des tas non-vides, la racine du tas fusionné est celle du tas ayant la plus petite valeur à la racine. L'autre tas est alors inséré comme nouvel enfant de ce tas, via un ajout à la liste des enfants.

Deux exemples de fusion sont illustrés dans la Figure 3.

Question #6 (2 points)

Écrire une fonction `tas_fusion`, prenant deux tas en paramètre, et retournant le résultat de la fusion. *Prenez soin de traiter les cas où l'un, ou les deux, des tas passés en paramètre seraient des tas vides.*

Solution: En ocaml (correction prepa.org)

```
let fusion a1 a2 =
  match a1, a2 with
  | Vide, _ -> a2
  | _, Vide -> a1
  | Noeud (r1, fils1), Noeud (r2, fils2) ->
      if r1 < = r2
```

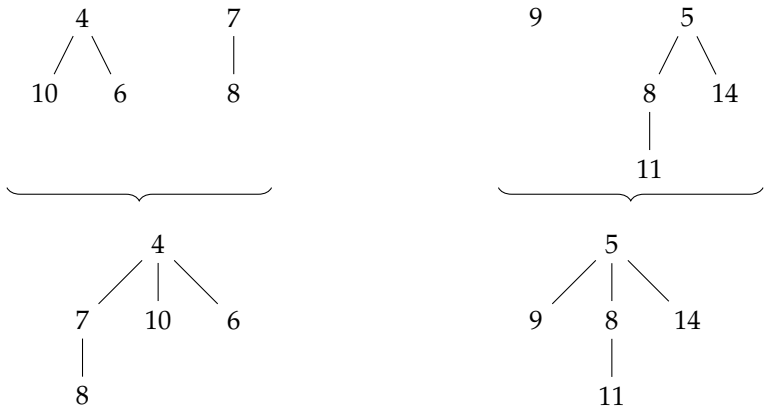

FIGURE 3 – Exemples pour l'opération fusion

```
      then Noeud(r1, a2 :: fils1)
      else Noeud (r2, a1 :: fils2)
```

Pour ajouter un élément à un tas, on crée un nouveau tas contenant cet élément, et on le fusionne avec le tas initial.

Question #7 (1 point)

Écrire une fonction `tas_insere`, qui insère un élément dans un tas.

Solution:

```
def insere x:int arbre:tas =
  fusion (Noeud (x, liste_vide())) arbre
```

Question #8 (2 points)

Écrire une fonction `tas_max`, qui prend un tas en paramètre, et retourne la plus grande valeur de ce tas.

Solution: La valeur maximale est la valeur maximale de l'enfant le plus à droite (c'est à dire en queue de liste.

```
def tas_max montas:Tas : int
  | Le_tas_vide => "should not happen"
  | Noeud(x,enfants) =>
        if est_vide(enfants) alors x
        sinon tas_max(queue(enfants))
```

il reste à écrire queue, flemme.

2 Algorithmique : retour au sac à dos !

Nous considérons le problème du sac à dos, avec la formulation et les notations ci-dessous.

Une grimpeuse souhaite préparer son sac à dos pour sa randonnée en montagne. Elle a devant elle n objets (**tous différents**) qu'elle pourrait emporter ; mais son sac ne peut emmener qu'un volume total de V, et c'est moins que la somme des volumes des n objets.

Elle attribue donc des valeurs de profit p_i aux objets (pour i allant de 1 à n), et choisit de maximiser la somme des valeurs des objets emportés. Le volume de l'objet i est noté r_i (le volume étant envisagé

comme une ressource) ; elle ne peut évidemment pas emporter plus d'objets que ce que lui permet le volume V de son sac !

Voici un exemple de problème de sac à dos : le volume total du sac est $V = 11$; les caractéristiques des 4 objets sont données dans le tableau suivant :

i	1	2	3	4
p_i	3	6	5	2
r_i	4	7	6	1

Question #1 (2 points)

Déterminez ce que va emporter la grimpeuse dans le cas de l'exemple fourni. Détaillez votre raisonnement.

Question #2 (3 points)

Écrire un algorithme glouton pour résoudre le problème du sac à dos, privilégiant les objets ayant le plus grand profit.

On propose dans la suite une routine de programmation dynamique qui résoud de manière exacte le problème du sac à dos.

Elle remplit un tableau A encodant dans la case (r, i) la valeur $A(r, i)$, représentant le meilleur profit d'un sac de volume maximum r, n'utilisant que des objets de numéros entre 1 et i.

Elle utilise la formule de récurrence :

$$A(r,i) = max(A(r,i-1), A(r-r_i, i-1) + p_i), \qquad \text{pour } i \text{ dans } \{1,\dots,n\} \text{ et } r \text{ dans } \{r_i,\dots,V\}$$

```
Fonction Sac_a_dos_programmation_dynamique(n, p, r, V)
    D : n, V : entiers
    D : p[1..n] : tableau d'entiers
    D : r[1..n] : tableau d'entiers
    L : i,k : entiers
    L : A[0..n][0..V] : tableau d'entiers
    R : m : entier
    Pour k de 0 à V Faire
        A[0][k] ← 0
    Fpour
    Pour k de 0 à V Faire
        Pour i de 1 à n Faire
            Si k < r_i alors
                A[i][k] ← A[i-1][k]
            Sinon
                A[i][k] ← max(A[i-1][k], A[i-1][k-r_i] + p_i)
            Fsi
        Fpour
    Fpour
    m ← A[n][V];
    Retourner m;
FFonction
```

Question #3 (1 point)

Justifier rapidement la formule de récurrence donnée.

Question #4 (1 point)

Dérouler l'algorithme ci-dessus sur l'exemple de problème de sac à dos donné plus haut.

Solution:

Question #5 (1 point)

Déterminer la complexité de cet algorithme de programmation dynamique.

Question #6 (1 point)

Démontrer que l'algorithme glouton proposé ci-dessus n'est pas optimal.

Solution: ici donner un Contre ex

Question #7 (2 points)

Enrichir l'algorithme de programmation dynamique, de sorte qu'il retourne la composition du sac à dos optimal.

Solution: En direct de l'énoncé des Mines :

```
def KPprogDynamique(obj,b):
    #
    #Lignes 2 à 14 précédentes conservées
    #
    S=n*[0]
    k=n-1
    r=b
    while r>0 and k>=0 and T[k+1][r]!=0:
        while k>0 and T[k+1][r]==T[k][r]:
            k-=1
        S[k]=1
        r=r-obj[k][0]
        k-=1
    return S,T[n][b]
```

Solution: