Examen d'algorithmique

jeudi 14 janvier 2016 15h30–18h30 / Aucun document autorisé

Mode d'emploi : Le barème est donné à titre indicatif. **La qualité de la rédaction des algorithmes et des explications sera fortement prise en compte pour la note.** On peut toujours supposer une question résolue et passer à la suite.

Exercice 1 : Dérouler des algorithmes (4 points)

1. On considère l'algorithme `P1` ci-dessous :

```
Def P1(entier x) :
Si x==0 Alors Retourner 0
Sinon :
  a=0
  b=1
  i=2
  tant que i <= x faire:
     aux = b
     b = a+b
     a = aux
     i=i+1
  Retourner b
```

   Décrire ce que fait l'algorithme `P1` appelé avec le paramètre 6. On décrira précisément l'état des variables `a` et `b` au cours de l'algorithme.

2. On considère l'algorithme `P2` ci-dessous :

```
Def P2(x) :
  Si x==0 ou x==1 Alors Retourner x
  Sinon Retourner P2(x-1)+P2(x-2)
```

   Décrire ce que fait l'algorithme `P2` appelé avec le paramètre 6. On décrira précisément tous les appels de fonctions.

3. Comparer ces deux algorithmes.

Exercice 2 : Tri pour deux valeurs - 4 points

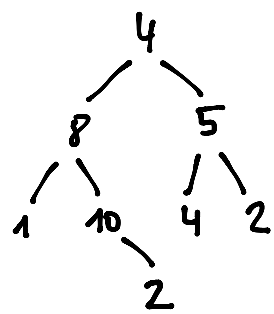
On veut définir un algorithme de tri pour des tableaux de taille n ne contenant que deux valeurs distinctes. On cherche à trier dans l'ordre croissant.

Par exemple pour le tableau suivant de taille 5 : `[2,4,4,2,2]`, on veut obtenir `[2,2,2,4,4]`

1. Ecrire un algorithme de tri basé sur une méthode de comptage.
2. Ecrire un algorithme qui trie le tableau en ne faisant qu'un seul parcours du tableau.

Exercice 3 : Algorithmes sur les arbres - 6 points

On considère des arbre binaires contenant des valeurs entières dans les noeuds, comme dans l'exemple ci-dessous :



On suppose que ces arbres sont représentés par des structures chaînées (comme en cours). Un noeud de l'arbre (type `noeud`) sera représenté par une structure ayant les champs de valeurs suivants :

— un champ de nom `val` et de type `entier` contenant la valeur stockée ;

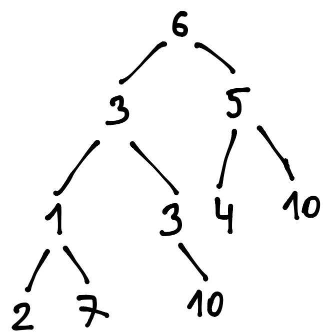
— un champ de nom `fg` et de type `arbre` contenant l'adresse du fils gauche ;

— un champ de nom `fd` et de type `arbre` contenant l'adresse du fils droit.

Et un `arbre` est un pointeur (adresse) vers un `noeud` (l'adresse 0 désigne un arbre vide). Lorsqu'un noeud n'a pas de fils gauche, son champ `fg` vaut 0 (et c'est pareil pour le fils droit avec `fd`). Un noeud qui n'a ni fils gauche, ni fils droit est une **feuille**.

Si `a` est un arbre non vide, `a->val` désigne la valeur stockée à sa racine (le premier noeud de l'arbre), `a->fg` désigne l'adresse du fils gauche (donc un arbre), et `a->fg` désigne l'adresse du fils droit, *etc.*

1. Dessiner la structure chaînée représentant l'arbre `test` ci-dessous :



2. Écrire un algorithme `Somme` qui étant donné un arbre `a` retourne la somme de toutes les valeurs stockées dans les noeuds de cet arbre (et 0 si l'arbre est vide).
   NB : Sur l'exemple, on doit renvoyer 36.
   Profil suggéré : `entier Somme(arbre a)`
   Appliquer votre algorithme sur l'arbre `test` (et décrire les éventuels appels de fonction, ou itérations. . .) .

3. Écrire un algorithme `CptFeuille` qui étant donné un arbre `a` retourne le nombre de feuilles de l'arbre `a`.
   NB : Sur l'exemple, on doit renvoyer 4.
   Profil : `entier CptFeuille(arbre a)`
   Quelle valeur retourne votre algorithme sur l'arbre `test` ?
4. Ecrire un algorithme `CptOcc` qui étant donné un arbre `a` et un entier `x` retourne le nombre d'occurrences de `x` dans l'arbre de racine `a`.
   NB : Sur l'exemple et avec $x = 4$, on doit renvoyer 2.
   Profil suggéré : `entier CptOcc(arbre a, entier x)`
   Quelle valeur retourne votre algorithme sur l'arbre `test` avec $x = 4$?
5. Ecrire un algorithme `Hauteur` qui étant donné un arbre `a` retourne la hauteur de l'arbre (la longueur du plus long chemin direct entre la racine et une feuille, et par convention on prendra -1 comme hauteur pour l'arbre vide).
   NB : Sur l'exemple, on doit renvoyer 3.
   Profil suggéré : `entier Hauteur(arbre a)`
   Quelle valeur retourne votre algorithme sur l'arbre `test` ?

Exercice 4 : Backtracking - 6 points

On s'intéresse ici aux mots construits à partir d'un alphabet Σ (un ensemble fini de lettres). Par exemple, si $\Sigma = \{a, b, c\}$, alors les mots *aaab*, *abc*, *abbbbbaaaab* sont des mots possibles. Le mot vide est noté ε et il est aussi un mot possible (de longueur 0). Mais le mot *abbdaab* n'est pas possible car d n'appartient pas à Σ.

Dans cet exercice, on pourra utiliser toutes les fonctions classiques sur les chaînes de caractères : concaténation (+), accès au i-ème caractère ($w[i]$), longueur ($|w|$), la répétition d'une lettre i fois (i*'a')...

1. Etant donné un alphabet Σ représenté par un tableau `T` de taille n (`T[i]` est la `i`-ème lettre) et un entier k, **écrire un algorithme qui affiche tous les mots de longueur k possibles avec Σ.**
   NB : Avec l'alphabet $\Sigma = \{a, b, c\}$ et $k = 2$, l'algorithme devra afficher : *aa*, *ab*, *ac*, *ba*, *bb*, *bc*, *ca*, *cb*, *cc*.
   Appliquer votre algorithme à l'alphabet $\Sigma = \{a, b\}$ (donc `T=[a,b]` et $k = 3$. On décrira avec précision le déroulé de l'algorithme.
   Profil suggéré si algorithme récursif : `void GenererMot(T,k,w)` où `w` est le mot en cours de construction (mot vide au premier appel). Et profil suggéré pour version itérative : `void GenererMot(T,k)`.
2. On reprend la question précédente mais cette fois, on remplace l'argument k par un tableau de caractères `m[-]` de longueur k qui va imposer un motif particulier aux mots recherchés : soit `m[i]` est une lettre de Σ et alors tous les mots affichés par l'algorithme devront avoir cette lettre à la position `i`, soit `m[i]` est `*` et alors n'importe quelle lettre de Σ peut se trouver à la position `i`. **Écrire un algorithme pour résoudre ce problème.**
   NB : Avec l'alphabet $\Sigma = \{a, b, c\}$ et `m=[a,*]`, l'algorithme devra afficher : *aa*, *ab*,

ac. Et si `m=[*,*]`, on retrouve tous les mots générés par l'algorithme de la question précédente pour $k = 2$.

Donner le résultat de l'application de votre algorithme pour l'alphabet $\Sigma = \{a, b, c\}$ et `m=[*,a,a,*]`.

3. On modifie le problème précédent en donnant un tableau d'entiers `Nb[-]` à la place du motif `m[-]` : `Nb` va décrire la taille des séquences de lettres identiques. Par exemple, si `Nb` est de taille 3 et que `Nb[0]` $= 3$, `Nb[1]` $= 2$ et `Nb[2]` $= 1$, alors les mots recherchés commenceront par une lettre répétée trois fois, puis une autre (pas la même !) sera répétée 2 fois, et le mot se terminera par un dernier changement de lettre (sans répétition). Donc avec $\Sigma = \{a, b, c\}$ et ce tableau `Nb`, on obtiendrait les mots suivants : *aaabba*, *aaabbc*, *aaacca*, *aaaccb*, *bbbaab*, *bbbaac*, *bbbcca*, *bbbccb*, *cccaab*, *cccaac*, *cccbba*, et *cccbbc*.

   Modifier l'algorithme précédent pour résoudre ce problème.

   Donner le résultat de l'application de votre algorithme pour l'alphabet $\Sigma = \{a, b\}$ et `Nb=[2,4,3]`.