# Algorithmique et Bases de la programmation

Durée : *1h55*
Documents autorisés : **AUCUN**

L'objectif de cet examen pratique est de compléter le benchmark de tri développé en TP en implantant le tri par tas étudié pendant le partiel et en TD.

## Informations

### Avant de commencer l'examen

Connectez vous à la machine et suivez les instructions suivantes :
- copiez l'archive `C.tar.gz` qui se trouve dans `/opt/files` dans votre répertoire (commande `cp`) ;
- après l'avoir décompressée (commande `tar` avec les options `zxvf`), renommez le répertoire en lui donnant comme nom : 'EXAM-ALGO-' suivi de votre nom et prenom, sans espace et en minuscule sauf pour les premières lettres du prénom et du nom. Si je devais passer l'examen, je nommerai le répertoire EXAM-ALGO-DelestreNicolas (je taperai donc dans le terminal `mv C EXAM-ALGO-DelestreNicolas`)

### À la fin de l'examen

Cinq minutes avant la fin de l'examen :
- assurez vous que votre projet compile ;
- créez une archive de votre répertoire (commande `tar` avec les options `zcvf`), que vous nommerez comme le répertoire (avec le préfixe `.tar.gz`) ;
- déposez sur moodle votre archive (cours « base de la programmation et algorithmique », section « examens pratiques »).

**Attention :**
- le lien de dépôt est actif uniquement 10 minutes, si vous dépassez le délai, votre note sera 0 ;
- on ne peut déposer qu'une seule fois le projet sur moodle.

### Quelques conseils

Pour réussir votre examen suivez les conseils suivants :
- respectez bien les consignes données ci dessus. **Le fait de ne pas les respecter vous fera perdre 2 points sur la note finale** ;
- par défaut le projet compile et s'exécute. Après le développement de chaque fonction C, compilez votre projet (`make`) et ne passez à la fonction suivante que lorsque votre projet compile : **le fait de rendre un projet qui ne compile pas ou ne s'exécute pas (par exemple à cause d'un *segmentation fault*), divisera par deux votre note finale** ;
- la note finale sera fonction :
  - du nombre de tests unitaires qui seront valides (exécution des programmes `bin/testTas` et `bin/testTriParTas`) ;

– du nombre de *Warning* (compilation avec l'option `-Wall`) qui seront affichés ;
– de la quantité, de la qualité et de la lisibilité du code (indentation et nom des identifiants).

# 1 Rappels algorithmiques : le tri par tas

## 1.1 Qu'est ce qu'un tas ?

Un tas est un arbre binaire particulier : la valeur de chaque noeud est supérieure aux valeurs contenues dans ses sous-arbres et l'arbre est rempli par niveau (de gauche à droite), un nouveau niveau n'étant commencé que lorsque le précédent est complet.

Un tas peut être représenté l'aide d'un tableau $t$ de telle sorte que les fils gauche et droit de $t[i]$ sont respectivement $t[2 * i]$ et $t[2 * i + 1]$.

La fonction suivante permet de savoir si un tableau $t$ composé de $n$ éléments significatifs est un tas à partir du rang $i$ :

**fonction** estUnTas (t : **Tableau**[1..MAX] **d'Entier** , i,n : **Naturel) : Booleen**
  ⌊**précondition(s)** i≤n
**debut**
  **si** 2*i > n **alors**
    **retourner** VRAI
  **sinon**
    **si** 2*i+1 > n **alors**
      **retourner** t[i]≥t[2*i]
    **sinon**
      **si** t[i]≥max(t[2*i],t[2*i+1]) **alors**
        **retourner** estUnTas(t,2*i,n) et estUnTas(t,2*i+1,n)
      **sinon**
        **retourner** FAUX
      **finsi**
    **finsi**
  **finsi**
**fin**

## 1.2 Pour transformer un tableau quelconque en tas : la procédure *tamiser*

Pour transformer un tableau quelconque en tas (objectif de la procédure *tamiser*) on a besoin de la procédure *faireDescendre*.

### 1.2.1 Procédure *faireDescendre*

À l'issue de l'appel à cette procédure *faireDescendre*, l'arbre (représenté par un tableau) dont la racine est en position $i$ sera un tas. On présuppose que les deux arbres dont les racines sont positionnées en $2i$ et $2i + 1$ sont des tas.

Son algorithme est en version itérative :

**procédure** faireDescendre **( E/S** t : **Tableau**[1..MAX] **d'Entier** , **E** i,n : **Naturel )**
  ⌊**précondition(s)** (2*i≤n et estUnTas(t,2*i,n)) ou (2*i+1≤n et estUnTas(t,2*i,n) et estUnTas(t,2*i+1,n))
  **Déclaration** elementBienPositionne :**Booleen**
    posMax :**Naturel**
**debut**

```
   elementBienPositionne ← FAUX
   tant que non elementBienPositionne faire
      si 2*i≤n alors
         posMax ← indiceMax(t,n,2*i,2*i+1)
         si t[i]<t[posMax] alors
            echanger(t[i],t[posMax])
            i ← posMax
         sinon
            elementBienPositionne ← VRAI
         finsi
      sinon
         elementBienPositionne ← VRAI
      finsi
   fintantque
fin
```

ou bien en version récursive :

**procédure** faireDescendre **( E/S** t : **Tableau**[1..MAX] **d'Entier** , **E** i,n : **Naturel )**

⌊**précondition(s)** (2*i≤n et estUnTas(t,2*i,n)) ou (2*i+1≤n et estUnTas(t,2*i,n) et estUnTas(t,2*i+1,n))

**Déclaration** posMax :**Naturel**

```
debut
   si 2*i≤n alors
      posMax ← indiceMax(t,n,2*i,2*i+1)
      si t[i]<t[posMax] alors
         echanger(t[i],t[posMax])
         si 2*posMax≤n alors
            faireDescendre(t,posMax,n)
         finsi
      finsi
   finsi
fin
```

Ses deux versions ont besoin de la fonction suivante :

**fonction** indiceMax **(**t : **Tableau**[1..MAX] **d'Entier** , n,i,j : **Naturel) : Naturel**

⌊**précondition(s)** i≤n et i≤j

```
debut
   si j≤n alors
      si t[i]>t[j] alors
         retourner i
      sinon
         retourner j
      finsi
   sinon
      retourner i
   finsi
fin
```

### 1.2.2 La procédure `tamiser`

**procédure** tamiser **( E/S** t : **Tableau**[1..MAX] **d'Entier** , **E** n : **Naturel )**

**Déclaration** i : **Naturel**

**debut**
  **pour** i ←n div 2 **à** 1 **pas de** -1 **faire**
    faireDescendre(t,i,n)
  **finpour**
**fin**

### 1.3 Le tri pas tas

Le principe du tri par tas est simple. Après avoir transformé le tableau $t$ à trier composé de $n$ éléments significatifs en un tas, cet algorithme est composé d'itérations $i$ (allant de $n$ jusqu'à 2) qui :
- échange $t[1]$ et $t[i]$ ;
- s'arrange pour que le tableau de $i-1$ éléments significatifs soit un tas.

## 2 Rappels C : le benchmark de tri

Le benchmark de tri développé en TP a pour objectif de comparer des tris. Ce projet C est composé initialement des fichiers suivants :
- `src/main.c` : le programme principal
- `include/echanger.h` et `src/echanger.c` : les fonctions permettant d'échanger deux variables de même type (entre autres pour le type `long int`)
- `include/triParMinimumSiccessif.h` et `src/triParMinimumSuccessif.c`
- `include/triParInsertion.h` et `src/triParInsertion.c`
- `include/triRapide.h` et `src/triRapide.c`
- `include/triFusion.h` et `src/triFusion.c`
- `lib/libechanger.a` : la librarie statique qui contient `echanger.o`
- `lib/libtris.a` : la librarire statique qui contient `triParMinSuccessif.o`, `triParInsertion.o`, `triRapide.o` et `triFusion.o`
- `makefile`

Nous lui avons ajouté les fichiers suivants pour le tas, le tri par tas et les tests unitaires y afférent :
- `include/tas.h` et `src/tas.c` ;
- `include/triParTas.h` et `src/triParTas.c` (la compilation de ce dernier est ajouté à la librairie `lib/libtris.a`) ;
- `src/testTas.c` ;
- `src/testTriParTas.c`.

## 3 Développement

**Travail à réaliser** En adaptant ces algorithmes au langage C (entre autres le fait que les tableaux en C commencent à l'indice 0), complétez les fichiers suivants (sans ajouter d'assertion pour les préconditions) :
- `src/tas.c`
- `src/triParTas.c`