A. Rappels sur les structures algorithmiques

On écrira des algorithmes ou des procédures en utilisant uniquement les trois actions de base lire, écrire et l'affectation représentée par une flèche, les trois structures conditionnelles si…alorts…finsi, si…alors…sinon…finsi, selon…dans…finselon et les trois structures répétitives tant que…faire…fintantque, répéter…jusqu'à et pour…de…à…faire…finpour . À cela s'ajoute la définition de fonctions et de procédures et des types construits (tableaux, enregistrements).

1. Écrire un algorithme qui affiche la première dans l'ordre alphabétique de trois chaînes de caractères données par l'utilisateur. On suppose que l'utilisateur donne des chaînes de caractères distinctes. Écrire deux versions (avec et sans "sinon") et les comparer (considérer par exemple le nombre de comparaisons). On suppose disposer des opérations usuelles de comparaison de chaînes.
2. Étudier le cout (la complexité) de chacune des procédures suivantes en remplissant les tableaux ci-dessous :

```
procédure decompte0(   entrée nb entier ;
                       sortie res entier )
locales cpt,tot  entiers
Début
      tot ← 0
      cpt ← 0
      tant que cpt ≤ nb faire
            tot ← tot + cpt
            cpt ← 1 + cpt
      fin tant que
      res ← tot
Fin
```

```
procédure decompte1(   entrée nb entier ;
                       sortie res entier )
locales cpt,tot  entiers
Début
      tot ← 0
      pour cpt de 1 à nb faire
            tot ← tot + cpt
      fin pour
      res ← tot
Fin
```

```
procédure decompte2(   entrée nb entier ;
                       sortie res entier )
locales cpt,com,tot  entiers
Début
      tot ← 0
      pour cpt de 1 à nb faire
            pour com de 1 à cpt faire
                  tot ← tot + 1
            fin pour
      fin pour
      res ← tot
Fin
```

```
procédure decompte3(   entrée nb entier ;
                       sortie res entier )
locales cpt,tot  entiers
Début
      tot ← 0
      cpt ← 0
      répéter
            cpt ← 1 + cpt
            tot ← tot + cpt
      jusqu'à tot > nb
      res ← tot
Fin
```

	nombre d'affectations	nombre de comparaisons	nombre d'opérations
pour nb=1 decompte0			
decompte1			
decompte2			
decompte3			
pour nb=2 decompte0			
decompte1			
decompte2			
decompte3			
pour nb=3 decompte0			
decompte1			
decompte2			
decompte3			
pour nb quelconque decompte0			
decompte1			
decompte2			
decompte3			

on rappelle que : $\sum_{i=1}^{i=n} i = \frac{n(n+1)}{2}$ *et* $\sum_{i=1}^{i=n} i^2 = \frac{n(n+1)(2n+1)}{6}$

B. À joug

Prévoir la valeur affichée par chacun de ces 4 programmes Pascal en fonction de la valeur donnée par l'utilisateur. En déduire jusqu'à quelle valeur donnée le programme affiche un résultat correct, les entiers étant limités en machine à un maximum de 16385.

```
Program de_con_te;
var n,cpt,i,j:integer;
begin
  writeln('valeur de n :');
  readln(n);
  cpt:=0 ;
  for i:=1 to n-1 do
      for j:= 1 to n-i do
            cpt:=cpt+1;
  writeln(n,'I-->',cpt);
end.
```

```
Program de_con_te2;
var n,cpt,i,j:integer;
begin
  writeln('valeur de n :');
  readln(n);
  cpt:=0 ;
  for i:=1 to n do
      for j:= 1 to n do
            cpt:=cpt+i;
  writeln(n,'I-->',cpt);
end.
```

```
Program de_con_te3;
var n,cpt,i,j:integer;
begin
  writeln('valeur de n :');
  readln(n);
  cpt:=0 ;
  for i:=1 to n do
      for j:= 1 to n do
            cpt:=cpt+j;
  writeln(n,'I-->',cpt);
end.
```

```
Program de_con_te4;
var n,cpt,i,j:integer;
begin
  writeln('valeur de n :');
  readln(n);
  cpt:=0 ;
  for i:=1 to n do
      for j:= 1 to i do
            cpt:=cpt+i+j;
  writeln(n,'I-->',cpt);
end.
```

C. À demi !

On considère un tableau de réels (de type NOTES) dont on utilise les cases 1 à N. On veut compter le nombre de valeurs supérieures ou égales à 10 dans ce tableau.

1. Écrire une fonction prenant en paramètre le tableau et N et donnant ce nombre.
2. Calculer le coût de l'appel de cette fonction sur le tableau ci-dessous.

11	8	15	9	7	3	12	9	17	5

3. Pour quel(s) tableau(x) à 10 éléments le coût de la procédure est-il le plus grand ? Et que vaut-il ?
4. Pour quel(s) tableau(x) à 10 éléments le coût de la procédure est-il le plus petit ? Et que vaut-il ?
5. Pour N quelconque, pour quelle forme des données aura-t-on le coût au pire ? Que vaut ce coût en fonction de N ?
6. Pour N quelconque, pour quelle forme des données aura-t-on le coût au mieux ? Que vaut ce coût en fonction de N ?

D. Grand écart

On considère un tableau d'entiers (de type TTAB) dont on n'utilise que les cases 1 à N.

1. Écrire une procédure donnant le plus petit écart que l'on puisse calculer en prenant deux éléments du tableau. Combien de comparaisons cette procédure effectue-t-elle ?
2. Peut-on écrire une procédure plus rapide en supposant le tableau trié ? Donner le nouveau nombre de comparaisons.
3. Écrire une procédure donnant le plus grand écart que l'on puisse calculer en prenant deux éléments du tableau. Combien de comparaisons cette procédure effectue-t-elle ?
4. Peut-on écrire une procédure plus rapide en supposant le tableau trié ? Donner le nouveau nombre de comparaisons.
5. Donner une version effectuant 2(N-1) comparaisons sur un tableau non trié.

E. Hou !

On considère la fonction suivante qui calcule le "ou" d'une suite de booléens stockés dans un tableau.

```
fonction ou(entrée p:tableau[1..MAX] de booléens; entrée N:entier):booléen
Début       i ← 0
            répéter                 i ← i+1
            jusqu'à i≥N ou p[i]
            ou ← p[i]
Fin
```

1. calculer les coûts au mieux, au pire et en moyenne pour un tableau de 2 éléments. Puis pour 3 et 4 éléments
2. Généraliser pour N éléments en précisant la forme des données dans chaque cas.

Remarque : $\sum_{k=0}^{k=n} x^k = \frac{x^{n+1}-1}{x-1}$ *et* $\sum_{k=0}^{k=n} k.x^k = x\frac{nx^n(x-1)-(x^n-1)}{(x-1)^2}$

F. Habit de recherche

On considère un tableau (de type TAB, le type des éléments n'est pas précisé — n'est pas important) dont on utilise les cases 1 à N. On désire écrire une procédure de recherche d'un élément dans ce tableau qui renvoie un booléen (valant VRAI si l'élément figure dans le tableau, FAUX sinon) et un entier : la position de l'élément dans le tableau.

1. Écrire une telle procédure en utilisant une boucle pour. Étudier son coût. Discuter du cas des
2. Améliorer la procédure précédente en utilisant une autre boucle. Étudier son coût. Discuter des occurences multiples.
3. Améliorer la procédure précédente en supposant le tableau trié. Étudier son coût. Discuter des occurences multiples.
4. Toujours dans le cas d'un tableau trié, on peut utiliser la recherche dichotomique : on compare l'élément cherché avec l'élément au centre du tableau, ce qui permet de restreindre la recherche à une moitié du tableau, puis on recommence avec cette moitié de tableau : on compare l'élément cherché avec celui au centre de cette moitié, ce qui permet de se restreindre à un quart du tableau, et ainsi de suite. Étudier le coût d'une telle procédure.

G. Notations asymptotiques

On classifie les fonctions dont la limite en $+\infty$ est $+\infty$ en fonction de leur comportement à l'infini : on définit

$$O(g) = \left\{ fonctions \quad f / \exists c \in R^{+*}, \exists n_0 \in N, \forall n \geq n_0, 0 \leq f(n) \leq c.g(n) \right\}$$
$$o(g) = \left\{ fonctions \quad f / \forall c \in R^{+*}, \exists n_0 \in N, \forall n \geq n_0, 0 \leq f(n) \leq c.g(n) \right\}$$
$$\theta(g) = \left\{ fonctions \quad f / \exists c_1, c_2 \in R^{+*}, \exists n_0 \in N, \forall n \geq n_0, c_1.g(n) \leq f(n) \leq c_2.g(n) \right\}$$
$$\Omega(g) = \left\{ fonctions \quad f / \exists c \in R^{+*}, \exists n_0 \in N, \forall n \geq n_0, 0 \leq c.g(n) \leq f(n) \right\}$$

On écrit traditionnellement f=O(g) au lieu de $f \in O(g)$, et on lit "f est un grand O de g". Intuitivement, $f=\theta(g)$ signifie que f et g varient de la même façon à l'infini, à une constante multiplicative près. Par exemple un polynôme $a_n x^n+a_{n-1}x^{n-1}+\ldots+a_0$ est un $\theta(x^n)$. On peut, sous certaines conditions de régularité sur f et g, écrire que :

$$f = \theta(g) \Leftrightarrow \lim_{n \to +\infty} \left(\frac{f(n)}{g(n)}\right) = c > 0 \quad \text{càd } f \text{ et } g \text{ varient de la même façon}$$
$$f = O(g) \Leftrightarrow \lim_{n \to +\infty} \left(\frac{f(n)}{g(n)}\right) = c \geq 0 \quad \text{càd } f \text{ ne varie pas plus vite que } g$$
$$f = o(g) \Leftrightarrow \lim_{n \to +\infty} \left(\frac{f(n)}{g(n)}\right) = 0 \quad \text{càd } f \text{ varie moins vite que } g$$
$$f = \Omega(g) \Leftrightarrow \lim_{n \to +\infty} \left(\frac{f(n)}{g(n)}\right) > 0 \ (\textit{éventuellement} \ +\infty) \quad \text{càd } f \text{ varie plus vite que } g$$

1. Comparer entre elles (avec les notations ci-dessus) les fonctions suivantes de la variable n :

   $\log(n), \sqrt{n}, e^n, n^2, n^3, n^4, \exp(n^2), e^{2^n}, e^{n+1}$

2. trouver des fonctions dans les classes suivantes :

   $o(\log(n)), \Omega(e^n), o(1/n)$

H. FlashBack

Donner les ordres de grandeur associés aux études des exercices précédents (A à F).

I. Étude d'algorithmes de tri

On s'intéresse au tri d' éléments d'un tableau, plus spécialement au tri sur place et par échanges. Le problème du tri est un des domaines les plus étudiés du point de vue de la complexité.

Pour généraliser un peu, considérons un tableau T dont les éléments sont de type ÉLÉMENT, un type quelconque mais sur lequel on paut définir une fonction CLÉ fournissant une valeur destiné à les ordonner. Le résultat ded la fonction CLÉ est de type TCLÉ, un type disposant d'opérations de comparaison ($<,>,=,\leq,\geq$) ; par exemple entier ou chaîne de caractères. On désire trier les éléments du tableau dont les indices sont compris entre deux valeurs données (souvent 1 et n, mais pas nécessairement).

Exemple de cas général : les éléments du tableau sont des enregistrements représentant des personnes par leur nom, prénom, et diverses informations, et la fonction CLÉ donne le nom (projection).

On va donc écrire une procédure TRI prenant en paramètre le tableau et les borenes des indices de la partie à trier et modifiant le tableau pour ordonner ses éléments selon les valeurs croissantes de leur CLÉ.

Remarque : il existe d'autres types de tri sur les tableaux (pas sur place — le tableau de départ n'est pas modifié — ou pas par échange — par exemple par comptage) ; il existe aussi d'autres structures de données permettant d'effectuer un tri (listes, arbres, tas).

1. Tri par sélection

On cherche dans la zone l'élément le "plus petit" (au sens des clés) que l'on échange avec le premier de la zone; puis on cherche le plus petit en dehors de celui-là, que l'on échange avec le second ; et ainsi de suite.

- Écrire la procédure TRI correspondante.
- Calculer son coût en comparaisons de CLÉ et en échanges d'ELEMENT.
- Quel est l'ordre de grandeur des coûts au mieux et au pire ?

2. Tri par insertion

La première case du tableau forme un sous-tableau trié ; on va donc insérer la seconde à sa place dans ce sous-tableau (c'est à dire soit on la laisse en seconde, soit on la permute avec la première). Puis les deux premières cases forment un sous-tableau trié, on va dons insérer la troisième à sa place dans ce sous tableau (toujours par échanges). Et ainsi de suite jusqu'à ce que le tableau soit trié.

Remarque : les termes "première", "seconde", etc. sont relatifs à la zone du tableau à trier, la première case est en fait celle d'indice la borne inférieure de la zone à trier.

- Écrire la procédure TRI correspondante.
- Calculer son coût en comparaisons de CLÉ et en échanges d'ELEMENT.
- Quel est l'ordre de grandeur des coûts au mieux et au pire ?

3. Tri rapide (ou tri pivot ou quicksort)

On sépare la zone à trier en deux parties en regroupant à gauche tous les éléments "plus petits" qu'une valeur fixe appelée pivot, et à droite tous les éléments "plus grands". Puis on trie récursivement la partie gauche et la partie droite.

- Discuter du choix du pivot.
- Écrire la procédure récursive TRI correspondante.
- Calculer son coût en comparaisons de CLÉ et en échanges d'ELEMENT.
- Quel est l'ordre de grandeur des coûts au mieux et au pire ?
- Discuter du choix du pivot.

4. Tri fusion

à rechercher

5. Tri par tas (ou heap sort)

à rechercher

J. Les problèmes suivants sont-ils NP, sont-ils P ?

Bien discuter de ce qu'est la "taille" des données.

1. Existe-t-il un diviseur comun strict de n entiers donnés ? Un diviseur strict d'un entier est un diviseur différent de 1 et de l'entier.
2. Un entier donné admet-il un diviseur strict ?
3. Deux entiers donnés sont-ils premiers entre eux ?
4. On appelle partition d'un entier une décomposition comme somme d'entiers strictement positifs, avec répétition possible mais sans ordre. On note p(n) le nombre de partition de n. Par exemple 5=4+1=3+2=3+1+1=2+2+1=2+1+1+1=1+1+1+1+1 et donc p(5) vaut 7. On s'intéresse au produit des éléments d'une partition et l'on note $\pi(n)$ le plus grand d'entre eux. Réécrire la recherche de $\pi(n)$ comme un problème de décision.
   Remarque : $p(n) \approx \frac{e^{\pi\sqrt{2n/3}}}{4n\sqrt{3}}$.
5. Soient n entiers $a_1,\ldots a_n$, et un entier K ; la médiane des a_i est-elle inférieure à K ? La médiane est une valeur de la série telle qu'il y ait autant de valeur supérieures que de valeurs inférieures dans la série ; éventuellement, la médiane peut être formée de deux valeurs (on en prend alors le milieu ou l'on parle de médiane inférieure et de médiane supérieure).

K. Bin-Packing

Soit un ensemble fini S d'entiers positifs, un entier K (la "bin capacity") et un entier N (le "nombre de bins").
Existe-t-il une partition de S en N sous-ensembles au plus telle que la somme des entiers de chaque sous-ensemble soit inférieure ou égale à K ?
Autrement dit, peut-on "empaqueter" les entiers de S dans au plus N "cases", sachant que la capacité de chaque case vaut K ?
Exemples d'applications concrètes :
• On veut ranger des boîtes rectangulaires de hauteur variées dans une armoire comportant N niveaux, les étagères étant à intervalle réguliers de K centimètres.
• On veut sauvegarder sur CD les vidéos contenues sur un disque dur. N est le nombre de CD dont on dispose, K est la capacité d'un CD.

1. Quelle est la taille des données ?
2. Ce problème est-il dans NP ?
3. Ce problème est-il dans P ?

L. Décomposé

Considérons l'algorithme suivant, supposé afficher la décomposition en facteurs premiers de n :

```
Début
    lire(n)      d ← 2
    répéter      si n mod d =0
                       alors        écrire(d)
                                n ← n div d
                       sinon d ← d + 1
          fin si
    jusqu'à d > n
Fin
```

1. Simuler son fonctionnement pour la valeur 105 puis pour 48.
2. Montrer que cet algorithme termine toujours.

M. Termine-t-on ?

Supposons que l'on dispose d'une fonction Pascal `termton(pgm,vals:string):string` qui prenne en entrée deux chaînes de caractères et renvoie "oui" ou "non" : la première chaîne est supposée contenir la définition d'une fonction Pascal ayant un unique paramètre chaîne et la fonction `termton` renvoie "oui" si la fonction Pascal contenue dans le fichier termine lorsqu'on lui donne `vals` comme paramètre, elle renvoie "non" si elle "boucle".

1. Écrire un exemple d'appel de la fonction `termton`.
2. Décrire (en français) ce que fait la fonction suivante :

```
function soimeme(pgm:string):string;
begin
      termton(pgm,pgm)
end;
```

3. En utilisant `soimeme`, écrire une fonction Pascal `boucleton` qui prenne en entrée la définition d'une fonction Pascal et qui renvoie "oui" si la fonction donnée comme paramètre "boucle" lorsqu'on l'applique sur sa propre définition ; `boucleton` doit boucler sinon.
4. Que renvoie `boucleton` si on l'appelle sur sa propre définition ? Conclure.

N. Et qui va lance ?

Supposons que l'on dispose d'une fonction Pascal `eqv(pgm1,pgm2:string):string` qui prenne en entrée deux chaînes de caractères et renvoie "oui" ou "non" : les chaînes de caractères sont supposées contenir des définitions de fonctions Pascal et la fonction `eqv` renvoie "oui" si ces deux fonctions sont équivalentes, elle renvoie "non" sinon. Deux fonctions seront dites équivalentes si elles renvoient toujours le même résultat (ou bouclent toutes les deux) quand on leur donne les mêmes paramètres (elles "font la même chose").

1. Écrire un exemple d'appel de la fonction `eqv`.
2. A quelle question la fonction `truc` suivante répond-elle par "oui" ou "non" ? On suppose que les entrées sont cohérentes : pgm contient une définition de fonction et nomfn est une fonction donnant le nom de la fonction définie dans son argument.

```
function truc(pgm:string):string;
var bidule:string;
begin bidule:='function t(s:string):string;';
      bidule:=bidule + 'begin  repeat until ' + nomfn(pgm)
      bidule:=bidule + '(s)="non"; t:="oui" ; end; ';
      truc:=eqv(pgm,bidule)
end;
```

3. Écrire deux exemples d'appel de la fonction `truc` , l'un répondant "oui" et l'autre ne s'arrêtant pas
4. Que vaut l'appel de `truc` appliquée à sa propre définition ? Conclure.

O. Spé s'y fier

Nous appellerons spécification d'un programme ou d'un algorithme la formalisation (par exemple en langage mathématique) de :

a) certaines conditions portant sur les données d'entrée nécessaires à l'algorithme (PRÉCONDITIONS) ;

b) le lien qui unit les résultats (sorties) de l'algorithme aux entrées (POSTCONDITION), sans nécessairement préciser comment ce lien est obtenu.

Par exemple, la recherche du plus petit élément d'un ensemble d'entiers peut se spécifier ainsi :

a) les entrées sont formées d'un ensemble d'entiers (donc sans répétition et sans ordre particulier) non vide ;

b) la sortie est l'un des élément de cet ensemble (donc un entier) qui est inférieur ou égal à tout élément de l'ensemble (on aurait aussi pu dire qui est strictement inférieur à toiut autre élément de l'ensemble).

Une autre formalisation possible est : $\forall E \in \mathrm{P}(\mathbf{Z}), E \neq \emptyset \Rightarrow \exists x \in E \,/\, \forall y \in E, x \leq y$ où ce qui est à gauche du signe d'implication correspond aux préconditions.

Formaliser mathématiquement les fonctions suivantes (en étudiant soigneusement les préconditions) :

1. f(x entier)=le plus grand diviseur de x
2. g(deux entiers a et b)=le plus petit multiple non nul de a qui soit aussi multiple de b
3. h(liste d'entiers l)=le plus petit élément de l
4. k(liste d'entiers l)=une nouvelle liste ayant les mêmes éléments que l (et en même quantité) mais par ordre croissant

P. What a proof !

Montrez que les algorithmes suivants vérifient les spécifications que vous avez écrites dans l'exercice précédent.

```
fonction f(x entier):entier
début
       d ← 2
       res ← 0
       tant que d≤x faire
              si x mod d = 0
                     alors res ← d
              fin si
              d ← d+1
       fin tant que
       retourner(res)
fin
```

```
fonction g(a,b entiers):entier
début
       m ← a
       tant que m mod b > 0 faire
              m ← m + a
       fin tant que
       retourner(m)
fin
```

```
fonction k(l  liste d'entiers):liste d'entiers
début
       % à vous de la définir %
fin
```

```
fonction h(l  liste d'entiers):entier
début
       m ← premier(l)
       tant que l ≠ liste_vide faire
              si m > premier(l)
                     alors m ← premier(l)
              fins si
              l ← reste(l)
       fin tant que
       retourner(m)
fin
```

Q. Boucle dort

On dit qu'un algorithme termine sur une entrée donnée s'il s'exécute en un temps fini avec cette entrée. On dit qu'il termine (absolument) s'il termine avec toutes les entrées vérifiant ses préconditions. Pour montrer qu'un algorithme termine, on exhibe en général une fonction (appelée taille) des entrées qui est entière (positive) et diminue strictement à chaque appel récursif ou à chaque tour de boucle. Comme il n'y a qu'un nombre fini d'entiers entre 0 et une valeur donnée, l'algorithme ne peut s'exécuter indéfiniment.

Montrez ainsi que les algorithmes précédents terminent.

R. Thermes

Étudier la terminaison des algorithmes suivants :

```
fonction xety(x,y entiers):entier
début
       tant que x * y ≠ 0 faire
              si y ≥ x
                     alors y ← y - x
                     sinon x ← x - y
              fin si
       fin tant que
       xety ← x + y
fin
```

```
fonction kap(n entier):entier
début
       tant que n > 1 faire
              si n mod 2 = 0
                     alors n ← n div 2
                     sinon n ← 3 * n + 1
              fin si
       fin tant que
fin
```