ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE – LAUSANNE
POLITECNICO FEDERALE – LOSANNA
SWISS FEDERAL INSTITUTE OF TECHNOLOGY – LAUSANNE

Faculté Informatique et Communications
Cours d'Information, Calcul et Communication, sections MA et PH
Chappelier J.-C.

Information, Calcul et Communication (SMA/SPH) : Correction de l'Examen I

3 novembre 2023

SUJET 1

INSTRUCTIONS (à lire attentivement)

IMPORTANT ! Veuillez suivre les instructions suivantes à la lettre sous peine de voir votre examen annulé dans le cas contraire.

1. Vous disposez de deux heures quarante-cinq minutes pour faire cet examen (13h15 – 16h00).

2. Vous devez **écrire à l'encre noire ou bleu foncée**, pas de crayon ni d'autre couleur.

   N'utilisez **pas non plus de stylo effaçable** (perte de l'information à la chaleur).

3. Vous avez droit à toute documentation papier.

   En revanche, vous ne pouvez pas utiliser d'ordinateur personnel, ni de téléphone portable, ni aucun autre matériel électronique.

4. Répondez aux questions directement sur la donnée, **MAIS** ne mélangez pas les réponses de différentes questions !

   Ne joignez aucune feuilles supplémentaires ; **seul ce document sera corrigé**.

5. Lisez attentivement et *complètement* les questions de façon à ne faire que ce qui vous est demandé. Si l'énoncé ne vous paraît pas clair, ou si vous avez un doute, demandez des précisions à l'un des assistants.

6. L'examen comporte 7 exercices indépendants sur 16 pages, qui peuvent être traités dans n'importe quel ordre, mais qui ne rapportent pas la même chose (les points sont indiqués, le total est de 145 points).

   Tous les exercices comptent pour la note finale.

Question 1 – Calcul approché. [23 points]

On cherche à écrire un programme C++ permettant de calculer une approximation du nombre $\frac{1}{e}$, en utilisant pour cela le développement en série entière de l'exponentielle :

$$\frac{1}{e} = \sum_{k=0}^{\infty} \frac{(-1)^k}{k!}$$

Note : $0! = 1$ et $(-1)^0 = 1$.

① **[5 points]** Écrire en C++ une fonction `factorielle()` qui prend en entrée un entier n sous forme de `double` et renvoie $n!$ (sous forme de `double`). Il **n'**est **pas** demandé de vérifier que la valeur n reçue en argument est bien une valeur entière.

Réponse : Voici une version possible :

```
double factorielle(double n)
{
  if (n <= 1.0) return 1.0;
  return n * factorielle(n - 1.0);
}
```

Note : une solution itérative est tout aussi valable.

Commentaire : Il ne faut pas oublier qu'un `double` peut être négatif.

Dans la version itérative, il ne faut pas oublier d'initialiser la variable d'accumulation.

② **[9 points]** On s'intéresse maintenant à l'approximation à l'ordre n de $\frac{1}{e}$: $\sum_{k=0}^{n} \frac{(-1)^k}{k!}$. Écrire une fonction `approx_inv_e()` qui prend en entrée un entier n et renvoie l'approximation de $\frac{1}{e}$ à l'ordre n en utilisant la formule ci-dessus.

Vous veillerez à minimiser la complexité de l'algorithme utilisé. (Voir aussi la question suivante.)

Réponse : Voici une solution efficace :

```
double approx_inv_e(double n)
{
  double term(1.0); // (-1)^0 / 0!
  double res(term);
  for (double i(1); i <= n; ++i) {
    term = -term; // (-1)^i = - (-1)^(i-1)
    term /= i; // i! = (i-1)! * i
    res += term;
  }
  return res;
}
```

Et voici une solution naïve, moins efficace :

```
double approx_inv_e(double n)
{
  double res(0.0);
  for (double k(0); k <= n; ++k) {
    res += pow(-1.0, k) / factorielle(k);
  }
  return res;
}
```

Note : `n` pourrait être `int` (le type de retour, par contre, doit nécessairement être `double`).

Commentaire : Malgré la consigne en gras, peu proposent la version efficace.

Parmi celles et ceux qui utilisent le calcul explicite de la puissance, beaucoup ne savent pas l'écrire correctement en C++ et ne semblent pas connaître (ou ont oublié) la fonction `pow()`.

③ **[3 points]** Quelle est la complexité de votre algorithme ? **Justifiez** votre réponse.

Réponse et justification : La première solution, qui calcule chacun des termes en $\Theta(1)$ (sorte de « programmation dynamique » simple, qui évite les recalculs), est en $\Theta(n)$.

La seconde solution, qui recalcule à chaque fois $k!$, lequel est en $\Theta(k)$, est en $\Theta(n^2)$.

Commentaire : Beaucoup de celles et ceux qui utilisent la version moins efficace oublient la complexité de la factorielle.

④ **[6 points]** Écrire une fonction `affiche_inv_e()` qui demande à l'utilisateur d'entrer un entier n strictement supérieur à 1 (répéter la question tant que ce n'est pas le cas), et qui affiche successivement l'approximation de $\frac{1}{e}$ à l'ordre 2, 3, ..., n.

Réponse : Voici une solution possible :

```
void affiche_inv_e()
{
  int n(-1);
  do {
    cout << "Entrez un entier strictement supérieur à 1 : ";
    cin >> n;
  } while (n <= 1);
  for (int i(2); i <= n; ++i) {
    cout << "ordre " << i << " : " << approx_inv_e(i) << endl;
  }
}
```

Commentaire : Attention à la portée de `n` (ne peut pas être dans le corps du `do`–`while` si apparaît dans la condition du `while`.)

Question 2 – Quelques algorithmes. [29 points]

Note : lisez la sous-question ③ (au dos) avant de répondre à ①.

Soient L_1 et L_2 deux listes quelconques de nombres entiers tous différents les uns des autres (dans chaque liste ; pas entre les listes). On cherche à écrire un algorithme ayant pour entrée ces deux listes et dont la sortie soit leur intersection (la liste des valeurs en commun), dans un ordre quelconque.

Par exemple, pour $L_1 = (-3, 12, 5, -2, 1, -16)$ et $L_2 = (7, 1, -9, 5, 4)$, un tel algorithme sortira alors p.ex. $(5, 1)$ (l'ordre peut être différent).

① **[6 points]** Écrivez un algorithme pour résoudre ce problème.

Réponse : Voici un premier algorithme, simple :

algo1
entrée : *L_1 et L_2 telles que décrites*
sortie : *liste L des valeurs communes*

```
n ⟵ taille(L1)
m ⟵ taille(L2)
L ⟵ () // liste vide
Pour i de 1 à n
    Pour j de 1 à m
        Si L1(i) = L2(j)
            L ⟵ L ⊕ L1(i) // ajout en fin de liste
Sortir : L
```

Notez qu'avec les conventions du cours, les boucles « **Pour** » ne se font pas si la liste est vide.

Commentaire : Globalement : cet exercices fait bien la part de la classe entre celles et ceux qui maîtrisent assez bien les algorithmes (en tout cas itératifs – cf commentaires plus loin) et celles et ceux qui doivent retravailler cette partie du cours : souvent soit l'exercice était globalement bien réussi, soit raté dès cette première question.

Quelques remarques cependant :

- tester son algorithme sur un exemple simple (par exemple $L_1 = (1, 2)$ et $L_2 = (2, 3)$) pourrait parfois éviter bien des erreurs ;
- attention aux variables dans les boucles « Tant que » : certain(e)s étudiant(e)s oublient de les incrémenter et leurs algorithmes ne terminent donc pas ;
- attention au dépassement d'indice par exemple :

```
Pour i de 1 à taille(L)
    Si L[i] = L[i + 1]
        ...
```

- quelques confusions entre l'affectation de la liste dans son ensemble ($L \longleftarrow x$) et l'ajout d'une valeur ($L \longleftarrow L \oplus x$)

② **[2 points]** Dans le cas où les deux listes ont la même taille n, quelle est la complexité de votre algorithme proposé en ① ? **Justifiez** votre réponse.

Réponse et justification : La complexité de l'algorithme précédent est en $\Theta(n^2)$. On parcourt en

effet la liste en entier et, pour chaque position de ce parcourt (i dans l'algorithme) on reparcourt toute la liste (j).

Commentaire : L'appartenance d'un élément à une liste n'est pas une opération élémentaire (c'est le problème de la recherche !).
De même, la concatenation de deux listes n'est pas une opération élémentaire.

③ **[6 points]** Écrivez un algorithme de complexité temporelle *strictement* moindre que $\Theta(n^2)$ pour résoudre le problème proposé.

Si votre réponse à ① est déjà en une complexité *strictement* moindre que $\Theta(n^2)$ lorsque les deux listes ont la même taille n, alors vous n'avez rien de plus à faire ici (et serez, bien entendu, noté(e) sur la somme des points des deux sous-questions).

Réponse : Voici un algorithme en $\Theta(n \log n)$ pour résoudre cette tâche :

```
algo1
entrée : L1 et L2 telles que décrites
sortie : liste L des valeurs communes
m ⟵ taille(L2)
L ⟵ () // liste vide
L'1 ⟵ tri(L1) // liste vide
Pour j de 1 à m
    Si recherche(L2(j), L'1) // dichotomie
        L ⟵ L ⊕ L2(j) // ajout en fin de liste
Sortir : L
```

Note : on peut faire un peu mieux pour la seconde partie, sans la recherche, mais en triant les *deux* listes : voir la solution récursive de ⑤ (qu'on peut aussi avoir implémentée ici sous forme itérative).

Commentaire : On ne peut pas faire une recherche par dichotomie si la liste n'est pas triée.

④ **[3 points]** Dans le cas où les deux listes ont la même taille n, quelle est la complexité de votre algorithme proposé en ③ ? **Justifiez** votre réponse.

Réponse et justification : La complexité de l'algorithme précédent est en $\Theta(n \log n)$. Le tri de L_1 est en effet en $\Theta(n \log n)$. Puis la boucle est en $\Theta(m) \times \Theta(\log n)$ puisque la recherche dans L'_1 peut être faite par dichotomie.

Et tout le reste est en $\Theta(1)$. Le tout reste donc en $\Theta(n \log n)$ pour $m = n$.

⑤ **[8 points]** On suppose maintenant que les deux listes sont *triées*.
Écrivez un algorithme *récursif* et sans boucle pour résoudre le problème proposé.

Par exemple, pour $L_1 = (-16, -3, -2, 1, 5, 12)$ et $L_2 = (-9, 1, 4, 5, 7)$, un tel algorithme sortira alors p.ex. $(1, 5)$ (l'ordre peut être différent).

Réponse :

Voici un algorithme récursif pour résoudre cette tâche :

intersecR

entrée : L_1 *et* L_2 *telles que décrites*
sortie : *liste* L *des valeurs communes*

Si L_1 est vide **ou** L_2 est vide
  Sortir : () // liste vide
Si $L_1(1) = L_2(1)$
  Sortir : $L_1(1) \oplus$ **intersecR**$(L_1(2,...), L_2(2,...))$ // ajout en début de liste
Si $L_1(1) < L_2(1)$
  Sortir : intersecR$(L_1(2,...), L_2)$
Sortir : intersecR$(L_1, L_2(2,...))$

en notant $L(2,...)$ la sous-liste de L sans son premier élément.

Commentaire : Très peu réussie, mais c'était la difficulté majeure de l'exercice. Il est important de retravailler cet aspect (récursivité) pour celles et ceux qui ont encore de la peine.

Voici quelques conseils (sur cet exercice) :
- plusieur(e)s étudiant(e)s renvoient le mauvais résultat pour les cas de condition d'arrêt (renvoient L_1 ou L_2 au lieu de la liste vide)
- attention au type de retour : renvoyer la liste vide, pas 0, ni une chaine de caractères vide
- appels récursifs : trop d'étudiant(e)s ne gèrent pas encore correctement le retour d'une fonction récursive ;
- pensez à vérifier/rester compatible avec les arguments lors de l'appel récursifs : même nombre, mêmes types ;
- beaucoup n'ont pas pensé à distinguer les cas $L_1[1] < L_2[1]$ et $L_1[1] > L_2[1]$.

⑥ **[4 points]** Dans le cas où les deux listes ont la même taille n, quelle est la complexité de votre algorithme proposé en ⑤ ? **Justifiez** votre réponse.

Réponse et justification : La complexité de l'algorithme précédent est en $\Theta(n)$.

Chaque appel récursif diminue la longueur d'au moins une des deux listes. Donc il y au plus $2n$ appels récursifs. Et tout le reste est en $\Theta(1)$. Le tout est donc en $\Theta(n)$.

suite au dos ☞

Question 3 – Tournez machines. [16 points]

On considère la machine de Turing ayant pour table de transition :

	0	1	ε
1	(3, ε, +)	(2, ε, +)	(8, 1, +)
2	(2, ε, +)	(2, ε, +)	(8, 0, +)
3	(3, 0, +)	(3, 1, +)	(4, ε, −)
4	(6, ε, −)	(5, ε, −)	(8, 0, +)
5	(5, 0, −)	(5, 1, −)	(1, ε, +)
6	(6, ε, −)	(6, ε, −)	(8, 0, +)

① **[5 points]** Quel est l'état de la bande et la position de la tête de lecture lorsque la machine s'arrête, si elle a démarré avec sa tête de lecture positionnée comme suit :

$\cdots\varepsilon$	0	0	0	0	1	1	1	1	$\varepsilon\cdots$
	↑								

② **[11 points]** Justifiez votre réponse en trois ou quatre phrase(s), puis dites en une phrase ce que fait cette machine.

Réponses :

$\cdots\varepsilon$	1	$\varepsilon\cdots$
	↑	

L'état 1 « réagit » au bit lu (choix du bloc à exécuter) en l'effaçant : si c'est un 0, alors on va chercher (états 3 et 4) s'il y a un 1 à la fin. Si c'est le cas, on l'efface, puis l'on revient au début de l'entrée de départ et on recommence (état 5), jusqu'à ne plus rien avoir ; on écrit alors 1.

Si ce n'est pas le cas (soit que le début n'est pas un 0 (état 2), soit qu'il n'y a pas de 1 à la fin (état(s) 5 (et 3)), on efface la bande (états 2 ou 6), et on met un 0.

Cette machine vérifie donc que l'entrée commence par un certain nombre (y compris aucun) de 0 et continue par autant de 1. (Note : on dit qu'elle vérifie 0^n1^n.)

Commentaire : Beaucoup ont bien compris l'aspect effacer d'un coté, aller au bout, effacer le bout, revenir au début, mais on mal conclu et on répondu que la machine effaçait tout. Or cela n'est pas possible simplement en regardant l'arrêt : les seuls arrêts de cette machine sont sur ε et ils écrivent tous quelque chose (0 ou 1). Il y a donc nécessairement quelque chose d'écrit lors de l'arrêt de la machine...

Par ailleurs, pour 11 points (et une place de réponde d'une page et demi), il y a trop de réponses très courtes, forcément très incomplètes.

suite au dos ☞

Question 4 – Satisfaits ? [29 points]

Le problème SAT (pour satisfaisabilité booléenne) est un problème de décision sur une formule de logique booléenne, qui cherche à déterminer s'il existe une assignation des variables telle que la formule soit vraie.

Par exemple (donné ici en C++ pour faciliter la lisibilité et la compréhension) :

```
bool x1(false);    bool x2(true);    bool x3(false);

bool f1( (x1 and x2) or (not x3) );
```

La formule `f1` est *satisfaite* (c.-à-d. est vraie) pour l'assignation des variables `x1`, `x2`, `x3` faite ci-dessus. Donc la réponse à cette instance de SAT est « oui ».

En revanche, la formule
`bool f2( x1 and x2 and (not x1) );`
n'est pas satisfiable : aucune affectation de la variable `x1` ne peut la rendre vraie (on dit « la *satisfaire* »). Donc la réponse à cette autre instance de SAT est « non ».

On s'intéresse ici à la résolution algorithmique de ce problème SAT.

① **[2 points]** Quelles sont la ou les entrée(s) d'un algorithme résolvant ce problème ?
Quelles sont sa ou ses sortie(s) ?

Entrée(s) : une formule de logique booléenne

Sortie(s) : oui ou non (la formule donnée est satisfiable).

Note : il serait aussi préférable, si la formule est satisfiable, de sortir une assignation (c.-à-d. un certificat) ; mais une telle réponse n'est pas attendue au niveau de ce cours.

Commentaire : Trop ne comprennent pas ce qu'est l'entrée du problème et parlent de variables ; certain(e)s même confondent l'exemple à 3 variables donné plus haut et le problème général.

Concernant la sortie : il faut aussi dire ce que cela représente (le sens (= la sémantique) donné(e)).

② **[3 points]** Qu'est-ce qu'un certificat pour une instance (positive) de ce problème ?

Réponse : une assignation des variables de la formule donnée, qui la rende vraie.

Commentaire : Beaucoup ne savent pas ce qu'est un certificat, ou ne savent pas appliquer concrètement la définition du cours.

③ **[2 points]** Si on a une formule consituée uniquement de `or` entre de simples variables (comme p.ex. `x1 or x2 or x3 or x4`) et une assignation de ses variables, quelle est la complexité du calcul de sa valeur (vraie ou fausse) ? **Justifiez** votre réponse.

Réponse et justification : C'est une complexité linéaire car il suffit de calculer la valeur du `or` de proche en proche ; le pire des cas étant qu'elles sont toutes fausses.

Commentaire : Assez bien réussi, mais il ne faut oublier de justifier, ou au moins mentionner, le pire cas (on calcule la complexité pire cas).

④ **[2 points]** Si on a une formule consituée uniquement de `and` entre de simples variables (comme p.ex. `x1 and x2 and x3`) et une assignation de ses variables, quelle est la complexité du calcul de sa valeur (vraie ou fausse) ? **Justifiez** votre réponse.

Réponse et justification : De façon identitique : c'est une complexité linéaire car il suffit de calculer la valeur du `and` de proche en proche ; le pire des cas étant qu'elles sont toutes vraies.

⑤ **[6.5 points]** D'après les deux questions précédentes, et sachant que toute formule booléenne peut être transformée en une séquence, de taille linéaire, de `and` ne portant chacun que sur des séquences (de taille linéaire) de `or` sur de simples variables[1], que pouvez-vous dire de la classe de complexité de ce problème ? **Justifiez pleinement** votre réponse.

Réponse et justification : On peut donc vérifier en un temps au plus quadratique si une formule est satisfaite pour une assignation donnée (certificat) : pour une formule initiale de taille n, la formule équivalente n'ayant pas plus que $\Theta(n)$ `and`, chaque terme étant un `or` n'ayant pas plus que $\Theta(n)$ variables, on pourra :

1. calculer en au pire $\Theta(n)$ la valeur de chacun des `or`
2. ce que l'on fera au pire $\Theta(n)$ fois
3. puis, en au pire $\Theta(n)$, calculer la valeur du `and`

On aura donc une vérification en « $\Theta(n) \times \Theta(n) + \Theta(n)$ », soit $\Theta(n^2)$.

Donc ce problème est **dans NP** (vérifiable en un temps polynomial).

Note : dans cet examen, « *simple variable* » désigne en fait un littéral sur les variables d'origine (mais ce terme n'est pas connu au niveau de ce cours). Ici on peut considérer avoir introduit des variables auxiliaires (liées) représentant les `not`. Par ailleurs, le `not` étant une opération élémentaire, cela ne change pas fondamentalement le raisonnement sur la complexité (de la vérification).

Commentaire : Enormes confusions entre « vérifier » et « résoudre » : beaucoup trop parlent ici de P (et se trompent).

⑥ **[2.5 points]** Pour écrire des algorithmes sur des assignations de variables booléennes, il est plus simple de voir de telles assignations comme l'écriture binaire d'un entier positif.

Si on a par exemple cinq variables booléennes `x1`, ..., `x5` et que l'on fait correspondre `x1` au bit de poids fort (et `x5` au bit de poids faible), quelle serait leur assignation correspondant à l'entier 13 ?

Justifiez votre réponse.

Note : une assignation est simplement une liste de valeurs booléennes. Par exemple l'assignation donnée dans le premier exemple tout au début est simplement la liste (`false`, `true`, `false`).

Réponse et justification : En binaire sur 5 bits, 13 s'écrit `01101`, ce qui correspond à l'affectation (`x1`, ..., `x5`) = (`false`, `true`, `true`, `false`, `true`).

Commentaire : Il faut donner toutes les valeurs : n'oubliez pas celles des variables correspondant aux bits de poids forts.

⑦ **[6 points]** En supposant que vous ayez :

— un algorithme « **assignation** », qui prend en entrée un nombre entier positif et une taille n, et qui sorte une assignation de n variables booléennes correspondant aux bits de l'écriture binaire de ce nombre (cf question précédente),

— un algorithme « **évalue** » qui prend en entrée une formule logique et une assignation de ses variables et qui, en sortie, donne la valeur (vraie ou faux) de la formule pour cette assignation,

proposez un algorithme qui, étant donné une formule logique, renvoie une assignation des variables satisfaisant la formule s'il en existe une, ou la liste vide sinon.

1. On parle de transformation linéaire en forme normale conjonctive équisatisfiable.

Note : vous pouvez considérer que l'on connait le nombre de variables d'une formule logique (exactement comme on connait la taille d'une liste) : $n \longleftarrow$ **nb_var**(formule).

Réponse :

```
SAT solver
entrée : formule booléenne F
sortie : assignation qui satisfait F
n ⟵ nb_var(F)
Pour i de 0 à 2^n − 1
    L ⟵ assignation(i, n)
    Si évalue(F, L)
        Sortir : L
Sortir : () // liste vide
```

Commentaire : Cette question (et la suivante – liée) n'est pas très bien réussie ; c'était la question difficile de l'examen.

Il faut bien faire attention à ne pas confondre n et 2^n, ceci plus largement qu'ici, p.ex. aussi ne pas confondre (en général) le nombre de bits et le nombre de valeurs possibles.

Pour celle et ceux qui ont la bonne boucle externe : il faut aussi parler des complexités des deux algorithmes à l'intérieur de la boucle.

⑧ **[5 points]** Quelle est la complexité de votre algorithme proposé en ⑦ ?
Justifiez *pleinement* votre réponse.

Réponse et justification :

- **nb_var** est au pire en $\Theta(M)$, où M est la taille de la formule – on pourra aussi dire ici : au pire en $\Theta(n)$;
- **assignation** est en $\Theta(n)$
- **évalue** est au pire en $\Theta(M^2)$ (où M est la taille de la formule ; cf question ⑤)

  on peut aussi dire ici : en supposant SAT dans NP, **évalue** est au pire polynomiale
- la boucle elle-même est clairement en $\Theta(2^n)$.

Note : un algorithme linéaire en n n'a aucun sens (confusion entre taille de la formule et taille de l'espace de recherche) et ne rapporte donc, bien sûr, aucun point ici.

Le tout est donc au mieux en « $\Theta(2^n)$ » (voire pire ; c'est en fait en $\Omega(2^n)$, mais on ne va pas aussi loin dans le cours) : c'est un algorithme (au moins) exponentiel.

Note : au niveau de ce cours, on pourra sans soucis confondre M (taille de la formule) et n (nombre de ses variables).

Question 5 – Un drôle de calcul. [8 points]

Considérez le code C++ suivant :

```
int g(int n, int x, int y)
{
  if (n <= 0) return x;
  return g(n-1, x+y, y);
}

int f(int n)
{
  return g(n, 0, 3);
}
```

① **[1 point]** Quelle est la valeur de `f(3)` ? **Réponse :** 9

② **[1 point]** Quelle est la valeur de `f(5)` ? **Réponse :** 15

③ **[1 point]** Quelle expression mathématique est réalisée par cette fonction ?
[2 points] Justifiez votre réponse par une brève démonstration.

Réponse et démonstration : $3 \times n$ si n est positif, et 0 sinon.

Voici trois exemples de démonstration :

1. Montrons par récurrence que, pour n entier positif, `g(n,x,y)` retourne $x + n \times y$:
   — c'est vrai pour $n = 0$;
   — supposons que ce soit vrai pour $n - 1$ ($n > 0$), alors `g(n,x,y)` retourne `g(n-1, x+y, y)`, qui, par hypothèse de récurrence, vaut $(x + y) + (n - 1) \times y = x + n \times y$.

2. preuve descendante : notons x_k et y_k les valeurs de `x` et `y` lors du k-ème appel récursif à `g` depuis `f`, c.-à-d. depuis `g(n,0,3)` :
   — On a $x_0 = 0$, $y_0 = 3$.
   — Pour tout k, $x_{k+1} = x_k + y_k$ et $y_{k+1} = y_k$.
   — Donc $y_k = y_0 = 3$ et $x_k = x_0 + 3k$ (suite arithmétique).
   — Donc $f(n) = x_n = 3n$.

3. preuve descendante moins formalisée :

   pour n entier positif :

$$g(n,x,y) = g(n-1, x+y, y) = g(n-2, x+2y, y) = \ldots = g(1, x+(n-1)y, y) = g(0, x+ny, y) = x+ny$$

Commentaire : Presque personne ne parle de n négatif.

Et beaucoup de « démonstrations » qui n'en sont pas mais restent des affirmations non justifiées (p.ex. paraphraser le code).

④ **[1 point]** Quelle est la complexité de l'algorithme implémenté par `f()` ?
[2 points] Justifiez votre réponse.

Réponse et justification : $\Theta(n)$. En effet, toutes les opérations sont en $\Theta(1)$, seul compte donc le nombre d'appels ; et on en fait exactement $n + 1$ (pour $n \geq 0$).

Commentaire : Beaucoup oublient de justifier que, hors des appels, la complexité est en $\Theta(1)$: ce n'est pas parce que l'on répète quelque chose n fois que la complexité est nécessairement en $\Theta(n)$ (elle pourrait être pire).

suite au dos ☞

Question 6 – Mauvais calcul ! [21 points]

Voici un programme C++ visant à implémenter deux algorithmes permettant de calculer a^n, pour $a \in \mathbb{R}$ et $n \in \mathbb{N}$:

```
1   #include <iostream>
2   using namespace std;
3
4   double exp_rec(double a, int n)
5   {
6     double res;
7     if (n <= 0) res = 0;
8     if (n == 1) res = a;
9     if (n%2 == 0) {
10      res = exp(a*a, n/2);
11    } else {
12      res = exp(a*a, n/2) * a;
13    }
14    return res;
15  }
16
17  double exp_iter(double a, int n)
18  {
19    double res;
20    while (n >= 0) {
21      if (n%2 == 1) {
22        res *= a;
23      }
24      a = a*a;
25      n = n/2;
26    }
27    return res;
28  }
29
30  int main()
31  {
32    double a;
33    int n;
34    cout << "Entrez un réel a : ";
35    cin >> a
36    cout << "Entrez un entier n : ";
37    cin >> n;
38    cout << "(rec)  a^n = " << exp_rec(a,n)  << endl;
39    cout << "(iter) a^n = " << exp_iter(a,n) << endl;
40    return 0;
41  }
```

Mais ce programme comporte plusieurs erreurs de programmation, possiblement de différente nature (syntaxe, déroulement, conception, méthodologie, ...).
Indiquez et **corrigez** toutes les erreurs (directement sur le code ci-dessus).
Expliquez *brièvement* les erreurs/corrections à droite du code ou sur la page ci-contre.
On ôtera 1 point pour toute indication d'une erreur qui n'en est pas une.

Explications :

Les six erreurs sont :

- ligne 7 : erreur de valeur : ce doit être 1 et non pas 0 ;
- lignes 7 et 8 : il faut des `return` et non pas des affectations (on pourrait aussi mettre des `else` lignes 8 et 9) ;
- lignes 10 et 12 : la fonction récursive s'appelle `exp_rec()` et non pas `exp()` ;
- ligne 19 : `res` n'est pas initialisée ;
- ligne 20 : l'inégalité doit être stricte (ou alors changer 0 en 1), sinon : boucle infinie ;
- ligne 35 : il manque le point-virgule à la fin.

Commentaire : Les erreurs sur les conditions d'arrêt (l.7 et 8) ont été peu vues ; de même que la boucle infinie (l.20). Et attention aux « fausses erreurs ».

suite au dos ☞

Question 7 – Un peu de calcul. [19 points]

① **[5 points]** Quelle est la valeur décimale du nombre binaire représenté en virgule flottante sur 10 bits avec (dans cet ordre) 1 bit de signe, 3 bits d'exposant et 6 bits de mantisse, par `0100100110` ?

Justifiez brièvement votre réponse (p.ex. en explicitant vos calculs).

Réponse et justification : `0100100110` se décompose alors en :

- signe : `0`, donc positif
- exposant : `100`, soit 4
- mantisse : `100110`, représentant donc `1,100110`,

  qui, en multipliant par 2^4, vaut (en binaire) : `11001,1`

  soit 25.5 ($= 16 + 8 + 1 + \frac{1}{2}$).

Commentaire : Bien réussie globalement. Erreurs les plus fréquentes :

- oubli du 1 initial sur la mantisse ;
- confusion entre la valeur de l'exposant et la valeur qu'il représente au final (2 puissance exposant).

② **[5 points]** Le nombre 2.875 (en base 10) peut-il être représenté exactement (sans erreur d'arrondi) en binaire à virgule flottante avec 2 bits d'exposant et 4 bits de mantisse ? **Justifiez** votre réponse.

Réponse et justification : oui : $2.875 = 2 + 0.5 + 0.25 + 0.125$, est représentable exactement en binaire (somme de puissances de 2), et ne nécessite pas plus que 4 bits de mantisse (de 1 à 1/8) et 1 bit d'exposant.

Commentaire : Aussi globalement bien réussie, mais plusieur(e)s se contentent de justifier qu'on peut l'écrire en binaire (dans l'absolu), sans justifier qu'on peut le faire sur la convention demandée ici (avec ses limites de taille).

③ **[4.5 points]** Combien vaut, en décimal, le résultat de l'opération en binaire signé par complément à deux, `11001001` + `10101101` ? **Justifiez** votre réponse.

Réponse et justification : L'addition de `11001001` + `10101101` donne `0111 0110`, qui, en représentation signée est un nombre positif (dépassement de capacité), de valeur décimale 118.

Note : la représentation étant « *en binaire signé par complément à deux* », il est clair qu'elle est nécessairement sur le nombre de bits donnés (huit ici) puisque dans cette représentation il est *nécessaire* de donner le bit le plus à gauche (c.-à.-d. le bit de signe).

Commentaire : Aussi globalement bien réussie, sauf par certain(e)s qui s'embrouillent avec un niveau de négatif de trop ou qui donnent une réponse sur 9 bits.

④ **[4.5 points]** Si les `int` sont stockés sur 8 bits en complément à deux, quel est le schéma binaire en mémoire de la valeur de la variable `i` suivante : `int i(-54);`

Justifiez brièvement votre réponse (p.ex. en explicitant vos calculs).

Réponse et justification : Il s'agit du complément à deux de 54.

La représentation binaire de 54 est `00110110` ($32 + 16 + 4 + 2$).

Son complément à deux est `11001010`.

On peut aussi directement rechercher la représentation binaire non signée de $256 - 54 = 202$.

Commentaire : Aussi bien réussie, sauf par certain(e)s qui ne restent pas sur 8 bits (typiquement oublient 1 ou 2 bits de poids fort(s)).