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, section PH
Chappelier J.-C.

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

31 octobre 2025

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 d'une heure quarante-cinq minutes pour faire cet examen (13h15 – 15h00).

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 feuille supplémentaire ; **seul ce document sera corrigé**.

   Si nécessaire, il y a deux pages de réponse supplémentaires en fin de copie.

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(e) des assistant(e)s.

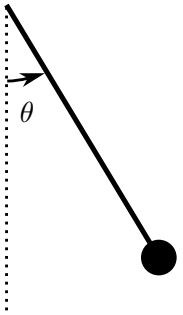
6. L'examen comporte 4 exercices indépendants sur 12 pages, qui peuvent être traités dans n'importe quel ordre, mais qui ne rapportent pas la même chose (les points sont indiqués, le total est de 90 points).

   Tous les exercices comptent pour la note finale.

Question 1 – Pendule simple [15 points]

On cherche à écrire des *parties* d'un programme C++ permettant de comparer avec la théorie un modèle numérique du pendule simple.

Le problème auquel on s'intéresse est donc un problème de simulation numérique, en 1D (l'angle du pendule), d'un objet oscillant autour de sa position d'équilibre.



Si l'on note θ l'angle du pendule par rapport à sa position d'équilibre stable, $\Omega = \frac{g}{L}$ (avec g l'accélération due à la pesanteur et L la longueur de la tige du pendule), et λ le coefficient de frottement fluide angulaire ($\lambda > 0$), l'équation d'évolution du pendule, linéarisé au voisinage de la position d'équilibre (θ petit) est :

$$\ddot{\theta} + 2\,\lambda\,\dot{\theta} + \Omega\,\theta = 0$$

(avec les notations usuelles $\ddot{\theta} = \frac{\mathrm{d}^2\theta}{\mathrm{d}t^2}$ et $\dot{\theta} = \frac{\mathrm{d}\theta}{\mathrm{d}t}$).

On cherche alors, à partir d'un angle initial θ_0 et d'une vitesse angulaire initiale $\dot{\theta}_0$, à approximer numériquement les valeurs de θ au cours du temps et à les comparer à la valeur théorique $\theta_{\text{théorie}}$

$$\theta_{\text{théorie}}(t) = e^{-\lambda\,t}\,(\theta_0\,\cos(\omega_d\,t) + \phi_0\,\sin(\omega_d\,t))$$

où $\lambda^2 < \Omega$, $\omega_d = \sqrt{\Omega - \lambda^2}$ et $\phi_0 = (\dot{\theta}_0 + \lambda\theta_0)/\omega_d$.

① **[3 points]** Écrire (en C++) la fonction `theorie()` qui prend en entrée t, θ_0, $\dot{\theta}_0$, λ et Ω et qui, si $\lambda^2 \geq \Omega$ retourne 0, et sinon, retourne la valeur théorique $\theta_{\text{théorie}}(t)$ explicitée ci-dessus.

Réponse : Voici une version possible :

```
double theorie(double t, double theta_0, double theta_p_0, double lambda, double Omega)
{
  if (lambda*lambda >= Omega) return 0.0;

  const double omega_d( sqrt(Omega - lambda*lambda) );
  const double phi_0( (theta_p_0 + lambda * theta_0) / omega_d );

  return exp(- lambda * t) * ( theta_0 * cos(omega_d * t) + phi_0 * sin(omega_d * t));
}
```

Commentaire : Simplifiez vous la vie en introduisant des expressions intermédiaires, et, surtout, surtout, **jamais de copié-collé !** (`omega_d`)

② **[2 points]** Écrire (en C++) la fonction `affiche()` qui prend en entrée t, une valeur arbitraire θ, puis θ_0, $\dot{\theta}_0$, λ et Ω, et qui affiche, séparés par des espaces :
t, la valeur arbitraire θ, la valeur théorique $\theta_{\text{théorie}}(t)$ et leur différence $\theta_{\text{théorie}}(t) - \theta$.

Par exemple, pour $t = 0.02$, $\theta = 0.093$ et une valeur théorique à cet instant t de 0.097, cette fonction affichera :

```
0.02 0.093 0.097 0.004
```

Réponse : Voici une version possible :

```
void affiche(double t, double theta,
             double theta_0, double theta_p_0, double lambda, double Omega)
{
  const double theta_theorie(theorie(t, theta_0, theta_p_0, lambda, Omega));

  cout << t << " " << theta << " " << theta_theorie
       << " " << (theta_theorie - theta) << endl;
}
```

Commentaire : Jamais, jamais de copié-collé ! (deux fois l'appel à `theorie()`)

③ **[4.5 points]** On s'intéresse maintenant à la résolution *numérique* d'une équation différentielle telle que celle donnée sur la page prédédente.
On utilise pour cela une fonction `integre()` qui prend en entrée θ, $\dot{\theta}$, $\ddot{\theta}$ et un « pas de temps » Δt (qui sont chacune simplement des variables numériques de type `double`) et qui *modifie* θ et $\dot{\theta}$ comme suit et dans cet ordre :

$$\dot{\theta} = \dot{\theta} + \ddot{\theta} \times \Delta t$$
$$\theta = \theta + \dot{\theta} \times \Delta t$$

Cette fonction ne retourne rien.

Par exemple, si l'on appelle cette fonction avec les arguments `integre(theta, theta_p, -0.2, 0.1)` et que `theta` valait 0.4 avant l'appel et `theta_p` valait 0.3 avant l'appel, alors, après l'appel `theta_p` vaudra 0.28 (parce que $0.28 = 0.3 - 0.2 \times 0.1$) et `theta` vaudra 0.428 (parce que $0.428 = 0.4 + 0.28 \times 0.1$; notez bien l'utilisation de la valeur 0.28 ici et non pas 0.3).

Écrire (en C++) la fonction `integre()`.

Réponse : Voici une version possible :

```
void integre(double& theta, double& theta_p, double accel, double dt)
{
  theta_p += accel   * dt;
  theta   += theta_p * dt;
}
```

Commentaire : N'oubliez pas les passages par référence.

④ **[5.5 points]** Pour finir, on veut comparer l'intégration numérique à la valeur théorique. Écrire pour cela (en C++) la fonction `teste()` qui prend en entrée un temps de fin t_{fin}, un « pas de temps » Δt, puis θ_0, $\dot{\theta}_0$, λ et Ω, et qui, pour chaque temps t entre 0 et t_{fin} en avançant de Δt à chaque fois, affiche la valeur calculée numériquement de proche en proche au moyen de la fonction `integre()`, la valeur théorique correspondante et leur écart.

Note : l'accéleration à passer à `integre()` est $\ddot{\theta} = -2\,\lambda\,\dot{\theta} - \Omega\,\theta$.

Voici un exemple du début de ce qu'afficherait cette fonction pour $\theta_0 = 0.4$, $\dot{\theta}_0 = 0.3$, $\Delta t = 0.1$, $\lambda = 0.01$ et $\Omega = 0.485$ (donc $\ddot{\theta}$ au départ vaut -0.2) :

```
 t   theta     theorie   écart
0    0.4       0.4       0
0.1  0.428     0.428977  0.000976838
0.2  0.453868  0.455818  0.00194996
0.3  0.477483  0.480398  0.00291484
```

Note : vous **n'**avez **pas** à vous préoccuper de l'affichage de la première ligne (`t theta theorie écart`), ni des alignements en colonnes.

Réponse : Voici une version possible :

```
void test(double t_fin, double dt,
          double theta_0, double theta_p_0, double lambda, double Omega)
{
  double theta(theta_0);
  double theta_p(theta_p_0);
  for (double t(0.0); t <= t_fin; t += dt) {
    affiche(t, theta, theta_0, theta_p_0, lambda, Omega);
    integre(theta, theta_p, - (2.0*lambda*theta_p + Omega*theta), dt);
  }
}
```

Commentaire : Jamais, jamais, *jamais* de copié-collé ! (Pensez à réutiliser les fonctions que vous avez faites.)
Pensez aussi à initialiser vos variables.

Question 2 – Quel rapport ? [27 points]

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

	`0`	`1`	ε
1	(1, `0`, +)	(2, `1`, +)	(3 , `1` , −)
2	(2, `0`, +)	(1, `1`, +)	(3 , `0` , −)
3	(3, `0`, −)	(3, `1`, −)	(4, ε, +)

① **[3 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	1	1	0	1	0	0	1	0	$\varepsilon\cdots$
	↑									

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

Réponses à ① et ② :

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

L'état 1 avance jusqu'au prochain '`1`' rencontré (« saute » les '`0`'), puis passe en l'état 2 ou, s'il n'y a plus rien (ε), écrit un '`1`' (à la fin de l'entrée initiale, donc).
L'état 2 est similaire à l'état 1, mais inscrit un '`0`' à la fin de l'entrée initiale, au lieu d'un '`1`'.
L'état 3 revient simplement au début de l'entrée (sans l'effacer).

Cette machine ajoute donc un bit à la fin, lequel indique si le nombre de '`1`' dans l'entrée est pair.

Commentaire : N'oubliez pas d'indiquer la position de la tête de lecture (c'était explicitement demandé).

③ **[3 points]** Quelle valeur décimale cette machine sortirait-elle pour la valeur 324 en entrée ? Autrement dit : si en entrée de cette machine, on a sur le ruban la représentation binaire (non signée) de 324, quelle est la valeur décimale correspondant à l'écriture binaire (non signée) obtenue en sortie ? **Justifiez** *brièvement* votre réponse.

Réponse et justification : 324 s'écrit `101000100` en binaire, qui a un nombre impairs de `1`, donc la machine va ajouter un `0` à la fin, c.-à-d. `1010001000`, lequel correspond à la valeur 648 (2×324).

Commentaire : N'oubliez pas de donner la valeur décimale comme explicitement demandé.

④ **[8 points]** Écrivez un algorithme qui prend en entrée une liste L de symboles binaires (`'0'` ou `'1'`) et qui en sortie répond « vrai » si le nombre de `'1'` dans L est pair ou nul, et « faux » sinon.

Réponse : Voici une solution possible, récursive :

Parité

entrée : *une liste L de 0 et de 1*
sortie : *« vrai » ou « faux » comme indiqué ci-dessus*

$n \longleftarrow$ **taille** (L)
Si $n = 0$ *// Si L est vide*
  Sortir : vrai
$a \longleftarrow$ **Parité** $(L[2], \cdots, L[n])$
Si $L[1] = 1$
  Sortir : **non** a
Sortir : a

Voici une autre solution possible, itérative :

Parité 2

entrée : *une liste L de 0 et de 1*
sortie : *« vrai » ou « faux » comme indiqué ci-dessus*

$a \longleftarrow$ vrai
Pour i de 1 à **taille** (L)
  Si $L[i] = 1$
    $a \longleftarrow$ **non** a
Sortir : a

On peut aussi compter :

Parité 3

entrée : *une liste L de 0 et de 1*
sortie : *« vrai » ou « faux » comme indiqué ci-dessus*

$a \longleftarrow 0$
Pour i de 1 à **taille** (L)
  Si $L[i] = 1$
    $a \longleftarrow a + 1$
Sortir : $a \mod 2 = 0$

Commentaire : 0 est pair, il n'est donc pas nécessaire d'écrire « $n = 0$ **OU** n est pair ».

⑤ **[8 points]** Écrivez une fonction **C++** qui prend en entrée un `int k`, que l'on supposera positif, et qui donne en sortie « `true` » si l'écriture binaire de `k` a un nombre pair (ou nul) de 1 et « `false` » sinon.

Réponse : Voici une version possible :

```
bool parite(int k)
{
  if (k == 0) return true;
  if (k % 2) return not parite(k/2);
  return parite(k/2);
}
```

Note : pour celles et ceux qui connaissent `unsigned`, `k` serait bien entendu mieux en `unsigned int`, mais ce n'est pas encore attendu à ce moment du semestre.

Commentaire : Cet exercice a étonnement été peu traité correctement (34% de réussite) alors que :

- la décomposition de l'écriture de nombres a été abordée plusieurs fois :
  - en binaire dans les slides du cours (slide 18 de la leçon I.4),
  - en décimal dans le devoir noté « lire et dire » du MOOC,
  - et à nouveau en binaire dans l'étude de cas du cours de C++ de la semaine 6 (`int2bin`).
- vous venez d'écrire l'algorithme à la question précédente...

Question 3 – Quelques algorithmes récursifs [36 points]

① **[12 points]** Proposez un algorithme **récursif** et sans boucle qui prend en entrée une liste L de nombres entiers et sort la liste des sommes des deux éléments de même rang à chaque bout : premier + dernier, second + avant-dernier, etc. ; autrement dit la liste $(L[1]+L[n],\ L[2]+L[n-1],\ L[3]+L[n-2],$ ...), où n est la taille de L.
Si la taille de la liste est impaire, $L[\frac{n+1}{2}]$ est additionné à lui-même dans la liste de sortie.

Par exemple, pour la liste $(7, 3, 6, 8, 5, 2)$ en entrée, on aurait la liste $(9, 8, 14)$ en sortie, puisque $7+2 = 9$, $3 + 5 = 8$ et $6 + 8 = 14$.

Autre exemple : pour la liste $(7, 3, 6, 5, 2)$ en entrée, on aurait la liste $(9, 8, 12)$ en sortie.

Réponse :

Voici une version possible :

Sommes
entrée : *Liste L*
sortie : *Liste des sommes des éléments de part et d'autre*

$n \longleftarrow$ **taille**(L)
Si $n = 0$
  Sortir : () // *liste vide*
Sortir : $\big(L[1] + L[n]\big) \oplus$ **Sommes**$\big(L[2], ..., L[n-1]\big)$

(sous-entendu que $\big(L[2], ..., L[n-1]\big)$ est la liste vide si $n \leq 2$.)

Commentaire : Ne perdez pas le début : trop d'oublis de combiner $\big(L[1] + L[n]\big)$ avec le résultat de l'appel récursif.
Quelques un(e)s oublient aussi de traiter le cas de la liste vide.

② **[4 points]** Quelle est la complexité de votre algorithme proposé en ① ? **Justifiez** votre réponse.

Réponse et justification : La complexité de l'algorithme précédent est en $\Theta(n)$ où n est la taille de la liste, et en supposant sans perte de généralité que **taille**() est en $\Theta(1)$ (il suffirait sinon simplement de la passer en argument supplémentaire).

On ne fait en effet rien de plus que de parcourrir la liste en entier par couple de valeurs (devant et derrière).

Commentaire : Il est aussi important de dire par rapport à quoi on mesure la complexité (c'est quoi « n » ?) si on ne l'a pas introduit clairement à la question précédente.
Certain(e)s confondent « enlever deux éléments » avec « enlever la moitié de la liste ».

③ **[6 points]** Que sort l'algorithme ci-dessous pour l'entrée $n = 6$?
Justifiez votre réponse.

Algo1

entrée : *un entier* $n \geq 0$
sortie : *? ? ?*

```
p ⟵ 1
Si n = 0
    Sortir : p
q ⟵ 2
Pour i de 1 à n − 1
    p ⟵ 2 × p
    q ⟵ p + n
p ⟵ n
Si p est impair
    K ⟵ 1
Sinon
    K ⟵ 1/2
Sortir : q − p + K × Algo1(⌊n/2⌋) × Algo1(⌊n/2⌋)
```

Réponse et justification :
Il sort $64 = 2^6$.
Cet algorithme sort 2^n. En effet :

- c'est vrai pour $n = 0$;
- la boucle ne fait que calculer $p = 2^{n-1}$;
- dans la dernière ligne, $q - p$ vaut toujours 2^{n-1} ;
- si n est pair, $\left\lfloor \frac{n}{2} \right\rfloor = \frac{n}{2}$ et la sortie vaut donc :

$$2^{n-1} + \frac{1}{2} \times 2^{\frac{n}{2}} \times 2^{\frac{n}{2}} = 2^n$$

- si n est impair, $\left\lfloor \frac{n}{2} \right\rfloor = \frac{n-1}{2}$ et la sortie vaut donc :

$$2^{n-1} + 2^{\frac{n-1}{2}} \times 2^{\frac{n-1}{2}} = 2^n$$

(Note : pour rappel, dans ce cours toutes les opérations arithmétiques sont considérées comme élémentaires.)

Commentaire : Cet exercice a été nettement moins bien réussi que ce que j'aurais pu penser. Trop « se jettent » directement « tête baissée » dans l'algorithme sans chercher à prendre un peu de recul pour se demander ce que chaque sous-partie fait effectivement.
Du coup, beaucoup d'erreurs de calcul, de compréhention, de report, liées à une complexité (cognitive) mal gérée...
Peut être avoir **lu TOUT l'exercice avant** de commencer, en particulier lu la sous-question ⑤, eût pu aider.

④ **[7 points]** Quelle est la complexité de l'algorithme **Algo1** proposé en ③ ? **Justifiez** votre réponse.

Réponse et justification (vous pouvez aussi donner des éléments de justification ci-dessus) **:**

Le plus simple pour répondre à cette question est peut être de dessiner le graphe des appels : un arbre binaire de profondeur $\log_2 n$ (passe de n à 0 en divisant par 2 (division euclidienne)).

Mais attention, ce simple dessin conclurait en une complexité en $\Theta\left(2^{\log_2 n}\right) = \Theta(n)$!
C'est sans tenir compte de la complexité interne de chaque appel, qui, en raison de la boucle, est en $\Theta\left(\frac{n}{2^k}\right)$ pour chaque nœud de profondeur k dans l'arbre. Comme on a 2^k tels nœuds (à chaque profondeur), la complexité de **chaque** profondeur est en $\Theta(n)$.
Comme l'arbre est de profondeur totale $\log_2 n$, on a donc une complexité totale en $\Theta(n \log_2 n)$.

La démonstration plus rigoureuse se fait en calculant l'équation vérifiée par la complexité. Si l'on note $C(n)$ la complexité pour une entrée n, alors en comptant on trouve pour n pair :

$$C(n) = \lambda + \mu\ (n-1) + 2\,C\left(\frac{n}{2}\right)$$

où λ correspond aux instructions élémentaires hors de la boucle et μ à celles dans la boucle.

On peut résoudre cette équation de proche en proche, soit en montant depuis 1, soit en descendant depuis n. On remarque alors que le terme dominant est le facteur de μ.

Si l'on ne se focalise que sur ce facteur — notons le $F(n)$ — on remarque qu'il suit l'équation :

$$F(n) = n - 1 + 2\,F(\frac{n}{2})$$

que l'on peut calculer pour des n puissances de 2 (poser $n = 2^k$). On voit alors qu'il croit comme $n \log_2 n$.

Notes (non attendu en examen) :

— la forme exacte de $F(n)$ est :

$$F(n) = n\,(\log_2(n) - 1) + 1$$

(Poser $G(n) = \frac{F(n)+n-1}{n}$ qui vérifie $G(n) - 1 = G(\frac{n}{2})$.)

— la forme exacte de $C(n)$ est :

$$C(n) = F(n)\,\mu + (n-1)\,\lambda + n\,C(1)$$

Commentaire : Beaucoup ont justifié en disant que la complexité de la boucle est en $\Theta(a)$ et que la prondeur de l'arbre est en $\Theta(\log a)$, *donc* la complexité totale est en $\Theta(a \log a)$.
Mais ce raisonnement est **faux**, car le « a » de la boucle change à chaque appel (il est a, puis $\frac{a}{2}$, puis $\frac{a}{4}$, ...) ! Il est important de bien voir qu'au niveau de profondeur k, on fera en tout k appels, chacun ayant une boucle de complexité en $\Theta(\frac{a}{k})$.

Certain(e)s multiplient les complexités des ré-appels (certainement parce que les résultats sont multipliés ?...) alors que « faire A *puis* A » coûte deux fois « faire A » et non pas « faire A » au carré.

⑤ **[7 points]** Simplifiez **Algo1** : proposez un algorithme équivalent, *récursif* et sans boucle, mais de complexité *linéaire* (et non constante ; c.-à-d. de complexité en $\Theta(n)$, mais pas en $\Theta(1)$).

Réponse : Je donne ici plusieurs pistes, au cas où l'on n'a pas vu en ③ ce que calcule effectivement l'algorithme.

Le double appel récursif n'apporte pas, en soit, de complexité ($\Theta\left(2^{\log_2 n}\right) = \Theta(n)$), mais autant le stocker directement dans une variable (version la plus élémentaire de la « programmation dynamique ») plutôt que de le laisser.

La vraie source complexité vient de la boucle, qu'il suffit de supprimer. Ce qui donnerait :

Algo1
entrée : *un entier* $n \geq 0$
sortie : *? ? ?*

$p \longleftarrow 1$
Si $n = 0$
  Sortir : p
$q \longleftarrow 2^{n-1} + n$
$p \longleftarrow n$
Si p est impair
  $K \longleftarrow 1$
Sinon
  $K \longleftarrow \frac{1}{2}$
$r \longleftarrow$ **Algo1**$(\lfloor \frac{n}{2} \rfloor)$
Sortir : $q - p + K \times r \times r$

qui est, du coup, en $\Theta(\log_2 n)$ (en considérant, comme supposé dans ce cours, que toutes les opérations arithmétiques sont élémentaires).

Après, on peut se rendre compte qu'il y a encore plein de choses inutiles dans cet algorithme :
- le calcul de q est inutile, puisque $q - p$, qui est la seule expression l'utilisant, vaut toujours la valeur de p à la sortie de la boucle, c.-à-d. maintenant la valeur de **Algo1**$(n-1)$;
- les affectations de p restantes sont aussi inutiles.

Tout ceci donnerait donc :

Algo1
entrée : *un entier* $n \geq 0$
sortie : *? ? ?*

Si $n = 0$
  Sortir : 1
Si n est impair
  $K \longleftarrow 1$
Sinon
  $K \longleftarrow \frac{1}{2}$
$r \longleftarrow$ **Algo1**$(\lfloor \frac{n}{2} \rfloor)$
Sortir : $2^{n-1} + K \times r \times r$

qui reste en $\Theta(\log_2 n)$ (sous les mêmes hypothèses).

Ce $\Theta(\log_2 n)$ résulte du fait que l'on a écrit directement 2^{n-1}, mais c'est un peu cacher la complexité d'origine sous une notation mathématique (d'où la remarque de la donnée sur « mais pas en $\Theta(1)$ »). Peut être que le mieux dans l'esprit de la donnée est d'aller plus loin et se rendre compte que cet algorithme calcule 2^n, que l'on peut donc calculer en $\Theta(n)$ de la façon suivante :

Algo1
entrée : *un entier* $n \geq 0$
sortie : 2^n

Si $n = 0$
  Sortir : 1
Sortir : $2 \times$ **Algo1**$(n-1)$

(qui me semblait plus simple à trouver si l'on avait compris dès ③ que l'algorithme retourne 2^n).

Commentaire : Cette question est très très peu réussie (22%), en raison de la mauvaise réussite de ③. Pourtant, comme indiqué dans le corrigé ci-dessus, on pouvait simplifier l'algorithme « mécaniquement » même sans l'avoir compris en détails.

Question 4 – Quelques nombres [12 points]

① **[3.5 points]** En représentation binaire non signée d'entiers sur 6 bits, quelle est la représentation binaire de `000100` × `011101` ?
Quelle est la valeur décimale correspondante ? **Justifiez** *brièvement* votre réponse.

Réponse puis justification : `110100` : puisque `000100` est une puissance de 2, il suffit simplement de décaler d'autant la représentation binaire vers la droite.

La valeur binaire correspondant est 52 ($32 + 16 + 4$).

Commentaire : Je suis très déçu de la réussite de cette question (53%) et surtout de la façon dont l'immense majorité de la classe l'a traitée : en faisant la multiplication en décimal.
J'ai donc échoué dans mon enseignement (`:_-(`) à vous faire *vraiment* comprendre deux choses :

- la numération positionnelle :
  en base machin-truc, combien vaut 100 × 83A7 ?
  réponse : 83A700 (ajouter deux zéros derrière son écriture)
- « débordement de capacité » ne veut pas dire « incapacité à faire » mais bien « on obtient *autre* chose » ; c'est d'ailleurs même l'idée fondamentale derrière la représentation en compléement à deux : on utilise justement cet « autre chose » pour faire que l'ajout de l'opposé donne 0.
  Bref, ne dites pas « on ne peut pas représenter », mais donnez ce qui est effectivement représenté.

---

② **[5 points]** Résoudre l'équation suivante sur l'ensemble des entiers *signés* représentés en binaire sur 7 bits (c.-à-d. donnez l'expression binaire de x sur 7 bits) :

$$x + \texttt{1001010} = \texttt{0110101}$$

Quelle est la valeur décimale correspondante ? **Justifiez** *brièvement* votre réponse.

Réponse puis justification : La réponse est `1101011`, qui correspond à la valeur décimale −21 (complément à deux de `0010101`).

Le plus simple à mon avis est de poser la soustraction comme à l'école élémentaire :

$$\begin{array}{rr} & \texttt{0110101} \\ - & \texttt{1001010} \\ \hline = & \texttt{1101011} \end{array}$$

Une autre solution consiste à ajouter l'opposé de `1001010`, donc ajouter `0110110` (obtenu par complément à deux) à `0110101`.

Finalement, la solution la moins efficace consiste à convertir `0110101` (53) et `1001010` (−54), puis à faire $53 + 54 = 107$, et à remettre ce résultat entre −64 et 63 : $107 - 128 = -21$.

Commentaire : Trop se compliquent la vie et surtout perdent le vue le point fondamental qu'en représentation signée, une écriture qui commence par `1` (à gauche) est forcément un nombre négatif.

---

③ **[3.5 points]** Quelle est l'erreur relative de représentation de 3.4 sur 6 bits avec 3 bits de mantisse et 2 bits d'exposant, en supposant une représentation par défaut (c.-à-d. *arrondi vers le bas*) et la représentation simplifiée non biaisée de l'exposant (donc toujours positif ou nul et sans comportement particulier pour 0) ? **Justifiez pleinement** votre réponse.

Réponse puis justification : Sur 3 bits de mantisse autour de la valeur recherchée on a comme puissances de 2 (ne pas oublier le 1 « automatique » devant la mantisse, donc $3+1$ bits utilisables) : 2, 1, 0.5 et 0.25, soient les valeurs 3, 3.25, 3.5 et 3.75

La valeur représentable par défaut de 3.4 est donc 3.25, soit une erreur relative de :

$$\frac{3.4-3.25}{3.4} = \frac{0.15}{3.4}$$

Commentaire : Beaucoup ne font pas le calcul exact du nombre représenté, mais donnent le *majorant* (erreur relative *maximale*) de $2^{-\text{taille mantisse}}$.