Concours e3a-Polytech, France

Session 2026, filière MP

Informatique

Durée : 4 heures

Ce que traite le sujet

- Problème 1 : multiplication de grands entiers
- Problème 2 : ensembles de mots implémentés par tries
- Problème 3 : logique implicationnelle

Sujet reproduit sans retouche à partir de la page suivante.

Résumé des parties, chapitres de cours liés et autres années :
prepahelp.com/annales/e3a/mp/2026/info

Prepa Help ne revendique aucun droit sur le sujet, propriété de l'organisateur du concours.

ÉPREUVE MUTUALISÉE AVEC E3A-POLYTECH

ÉPREUVE SPÉCIFIQUE - FILIÈRE MP

INFORMATIQUE

Durée : 4 heures

N.B. : le candidat attachera la plus grande importance à la clarté, à la précision et à la concision de la rédaction. Si un candidat est amené à repérer ce qui peut lui sembler être une erreur d'énoncé, il le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'il a été amené à prendre.

RAPPEL DES CONSIGNES

- *Utiliser uniquement un stylo noir ou bleu foncé non effaçable pour la rédaction de votre composition ; d'autres couleurs, excepté le vert, bleu clair ou turquoise, peuvent être utilisées, mais exclusivement pour les schémas et la mise en évidence des résultats.*
- *Ne pas utiliser de correcteur.*
- *Écrire le mot FIN à la fin de votre composition.*

Les calculatrices sont interdites.

Le sujet est composé de trois problèmes indépendants.

PROBLÈME 1
Multiplication de grands entiers

Pour manipuler des entiers, les ordinateurs manipulent à la place des suites de bits *encodant* ces entiers. On admet que si n est un entier naturel et b est un entier supérieur ou égal à 2, il existe une décomposition de n sous la forme :

$$n = \sum_{i=0}^{k} n_i b^i$$

appelée décomposition de n en base b. Les n_i sont appelés *chiffres de n en base b* et chaque n_i est dans $\{0, \ldots, b-1\}$. La base choisie de manière standard pour les entiers est habituellement $b = 2$ et pour cette valeur de b, on appelle les chiffres des *bits*. Par ailleurs, pour des raisons liées au matériel, on se limite à un nombre de chiffres borné, entre 32 et 64 sur les machines modernes.

Dans tout ce problème, on considère que la base est $b = 2$ et le seul langage utilisé est Python.

Q1. Quelle est la conséquence de cette contrainte matérielle sur l'ensemble des entiers que peut manipuler un programme ? Est-ce le cas en Python ?

Dans ce problème, on va manipuler des listes de taille non bornée de chiffres. Ces listes seront représentées par des listes Python dans lesquelles le terme de rang i vaudra le i-ème bit, n_i, c'est-à-dire que les bits associés à une puissance de 2 faibles seront en début de liste. Par exemple $[0, 1, 0, 1, 0, 1]$ représente l'entier $42 = 0 + 2 + 0 + 8 + 0 + 32$.

Q2. Donner le code d'une fonction `bit_to_int(bit_lst)` prenant en argument une liste de bits et renvoyant la valeur de l'entier correspondant. Cette fonction devra avoir une complexité linéaire en la taille de la liste qu'il n'est pas nécessaire de justifier.

Q3. Expliquer en français les grandes lignes d'un algorithme pour réaliser la somme de deux entiers représentés sous forme de listes de bits.

Q4. Donner le code d'une fonction `sum_bits(b_lst_1, b_lst_2)` réalisant l'addition de deux entiers encodés par liste de bits. Cette fonction prendra en argument deux listes de bits et renverra une nouvelle liste de bits correspondant à la somme des deux entiers encodés par les arguments. Il est interdit de convertir ces listes de bits en entiers, de faire la somme avec + et de reconvertir vers une liste de bits. On pourra supposer que les deux listes sont de même longueur.

Pour simplifier les opérations mettant en œuvre deux listes de bits (addition, soustraction, multiplication, etc.), on souhaite que ces deux listes soient de même taille.

Q5. Implémenter une fonction `complement(lst_bits, size)` qui complémente la liste de bits de manière à lui donner la taille `size` sans changer la valeur représentée. Votre programme vérifiera que l'entier `size` est supérieur à la taille de la liste des bits passée en argument et si cette assertion n'est pas vérifiée, interrompra l'exécution du programme. Votre fonction ne modifiera pas la liste passée en argument, mais en renverra une nouvelle.

Q6. Donner le code d'une fonction `int_to_bit` permettant d'obtenir la liste des chiffres de la décomposition en base $b = 2$ d'un entier passé en argument. On pourra s'appuyer sur la valeur de n modulo b dans l'expression de sa décomposition et regarder l'effet d'une division entière de n par b avec cette expression.

On souhaite à présent réaliser la multiplication de deux entiers u et v encodés par des listes de bits. On peut supposer, sans perte de généralité, que la taille k de ces listes est la même et est divisible par 2, quitte à utiliser la fonction `complement` pour que ce soit le cas. On peut s'appuyer sur la relation suivante en notant :

$$u = \sum_{i=0}^{\frac{k}{2}-1} u_i b^i + \sum_{i=\frac{k}{2}}^{k} u_i b^i = d_u + b^{\frac{k}{2}} f_u$$

et :

$$v = \sum_{i=0}^{\frac{k}{2}-1} v_i b^i + \sum_{i=\frac{k}{2}}^{k} v_i b^i = d_v + b^{\frac{k}{2}} f_v$$

on a alors :

$$\begin{aligned} u \times v &= \left(\sum_{i=0}^{k} u_i b^i\right) \times \left(\sum_{i=0}^{k} v_i b^i\right) \\ &= \left(\sum_{i=0}^{\frac{k}{2}-1} u_i b^i + \sum_{i=\frac{k}{2}}^{k} u_i b^i\right) \times \left(\sum_{i=0}^{\frac{k}{2}-1} v_i b^i + \sum_{i=\frac{k}{2}}^{k} v_i b^i\right) \\ &= \left(d_u + b^{\frac{k}{2}} f_u\right) \times \left(d_v + b^{\frac{k}{2}} f_v\right) \\ &= d_u \times d_v + b^{\frac{k}{2}} \left(f_u \times d_v + f_v \times d_u\right) + b^k \left(f_u \times f_v\right). \end{aligned}$$

Ce calcul assure la véracité de l'égalité :

$$u \times v = d_u \times d_v + b^{\frac{k}{2}} \left(f_u \times d_v + f_v \times d_u\right) + b^k \left(f_u \times f_v\right). \tag{1}$$

On peut donc calculer le produit $u \times v$ en :

a) décomposant les listes de bits de u et v en (d_u, f_u) et (d_v, f_v) respectivement,
b) calculant récursivement les 4 produits $d_u \times d_v$, $f_u \times d_v$, $f_v \times d_u$ et $f_u \times f_v$,
c) faisant les multiplications par les puissances de b : $b^{\frac{k}{2}} \left(f_u \times d_v + f_v \times d_u\right)$ et $b^k \left(f_u \times f_v\right)$,
d) additionnant les trois termes obtenus pour respecter l'égalité (1).

On note $C(k)$ la complexité de l'algorithme induit par cette méthode sur une liste de taille k. On admet que les opérations a, c et d peuvent se faire linéairement en k.

Q7. Établir une équation vérifiée par $C(k)$ et résoudre cette équation pour obtenir la complexité de la méthode. On pourra supposer que la taille de la liste est une puissance de 2 pour ne pas se préoccuper des arrondis lors de sa division par 2 et que la fonction C est croissante.

Le mathématicien soviétique Anatoly Karatsuba a proposé la relation suivante pour améliorer la complexité du calcul de la multiplication :

$$u \times v = d_u \times d_v + b^{\frac{k}{2}}(f_u \times f_v + d_u \times d_v - (f_u - d_u) \times (f_v - d_v)) + b^k \times f_u \times f_v \tag{2}$$

On admet que cette relation est correcte.

Q8. Combien de multiplications récursives distinctes sont nécessaires pour réaliser le produit $u \times v$ avec l'égalité (2) ? Donner en conséquence l'équation vérifiée par la complexité de cette méthode de multiplication.

On admet qu'en résolvant cette équation vérifiée par la complexité, on obtient une complexité temporelle en $O(k^{\log_2(3)}) \simeq O(k^{1,586})$, ce qui est mieux que l'approche précédente.

Q9. Indiquer ce à quoi correspond la multiplication par b^i d'un nombre vis-à-vis de sa représentation sous forme de la liste de chiffres en base b (étape c).

Q10. Implémenter une fonction `shift(lst_bits, i)` qui réalise cette opération sur une liste de bits passée en argument. Cette fonction renverra une nouvelle liste sans modifier celle passée en argument.

On considère le code Python d'une fonction `karatsuba(lst1, lst2)` pour réaliser la méthode proposée par Karatsuba.

On suppose disposer d'une fonction de somme `sum_bit_lst(lst1, lst2)` et de soustraction `sub_bit_lst(lst1, lst2)` qui réalisent les opérations attendues en temps linéaire en la taille de leurs entrées.

```
def karatsuba(lst1, lst2):
     n = max (len(lst1), len(lst2))
     lst1 = complement(lst1, n)
     lst2 = complement(lst2, n)
     if n == 0:
         # A : compléter ici
     if n == 1:
         # B : compléter ici
     else:
         h = n//2
         d1 = lst1[0:h]
         f1 = lst1[h:]
         d2 = lst2[0:h]
         f2 = lst2[h:]
         # C : compléter ici
```

Q11. Compléter les parties A, B et C du code proposé pour qu'il implémente la méthode de Karatsuba. Plusieurs lignes peuvent être nécessaires pour compléter certaines parties.

PROBLÈME 2
Ensembles de mots implémentés par *tries*

On s'intéresse dans ce problème à l'implémentation de structures permettant de représenter des ensembles de mots. Ce type de structure peut être utilisé pour la correction orthographique — si un mot n'est pas dans l'ensemble des mots connus, il y a probablement une faute — ou bien l'auto-complétion : s'il existe un mot dans notre ensemble qui commence comme celui qui vient d'être tapé, on le propose pour compléter.

Q12. Citer deux structures au programme pour implémenter un dictionnaire.

Q13. Comment pourrait-on utiliser un dictionnaire pour représenter des ensembles de mots lorsque l'objectif est de vérifier si un mot appartient à un ensemble ?

Dans la suite, on propose une première manière d'implémenter un ensemble de mots nommée *trie*. Il s'agit d'une structure d'arbre dans laquelle chaque nœud contient un booléen et a un enfant (éventuellement vide) par lettre de l'alphabet. Pour alléger les représentations, on n'affichera pas les enfants vides. Un chemin de la racine vers un nœud de l'arbre correspond donc à une suite de lettres, c'est-à-dire un mot. On dit qu'un mot appartient au *trie* si le dernier nœud du chemin étiqueté par ce mot depuis la racine contient `true`.

Par exemple, l'ensemble de mots {le, lettre, lettrine, liste} est représenté par le *trie* en **figure 1**. Le mot "let" n'est pas dans ce *trie* car le nœud sur lequel on arrive en lisant ce mot depuis la racine n'est pas étiquetté par `true`.

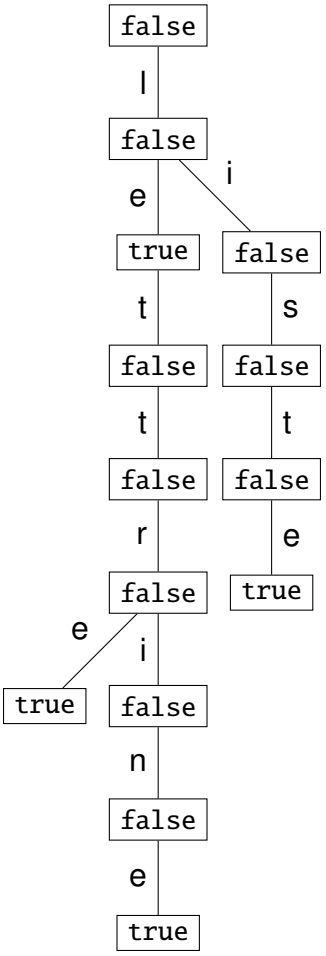


FIGURE 1 - Un exemple de *trie*

Q14. Dessiner le *trie* correspondant à l'ensemble de mots suivants : {en, entier, bot, bottier, bottine, routier, routine}.

Dans la suite du problème, le langage OCaml sera le seul langage utilisé.

Pour manipuler des mots en OCaml, on choisit d'utiliser des listes de lettres ; une lettre étant représentée par un `char`. On commence par se doter d'une fonction permettant de convertir un objet de type `string` en une liste de caractères. Il est rappelé qu'on peut accéder au caractère d'indice `i` d'une chaîne `s` en OCaml avec la syntaxe `s.[i]` et qu'on peut connaître la longueur de `s` via la fonction `String.length`.

Q15. Implémenter une fonction `string_to_list : string -> char list` qui s'évalue en la liste des caractères de la chaîne passée en argument. Par exemple, `string_to_list "lettre"` s'évaluera en `['l'; 'e'; 't';'t';'r';'e']`. On pourra s'appuyer sur une fonction auxiliaire si besoin, mais dans ce cas on prendra soin d'expliquer le rôle de cette fonction. La fonction aura une complexité linéaire en la taille de la chaîne qu'on justifiera.

On considère le type récursif OCaml suivant pour implémenter un *trie*.

```
type trie = Node of bool * (char * trie) list
```

On omet les arbres vides, un nœud peut donc avoir un nombre variable d'enfants. Les enfants d'un nœud sont stockés dans une liste de couples (lettre, arbre) et chaque nœud, y compris les feuilles, contient un booléen indiquant l'éventuelle présence d'un mot.

Par exemple, le *trie* en **figure 2** est représenté en OCaml par l'objet de type `trie` suivant :

```
Node (false,
   [('l',
     Node (false,
      [('e', Node (true, [('s', Node (true, []))])); ('a', Node (true,
    []))]))])
```

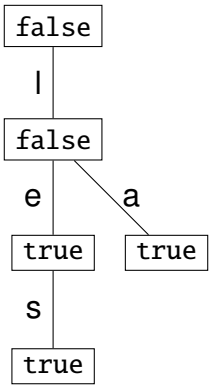


FIGURE 2 - Un exemple de petit *trie*

Q16. Implémenter une fonction `empty_trie : unit -> trie` qui renvoie le *trie* vide ayant le moins de nœuds possible.

Q17. Implémenter une fonction récursive `max_list : int list -> int` qui renvoie le plus grand élément d'une liste supposée non vide.

Q18. Implémenter une fonction récursive `list_map : ('a -> 'b) -> 'a list -> 'b list` qui prend en argument une fonction `f` et une liste `lst` et renvoie la liste des images par `f` des éléments de `lst`, dans le même ordre. Il est interdit d'utiliser la fonction `List.map` dans cette question.

Q19. Implémenter une fonction `height : trie -> int` qui calcule la hauteur d'un *trie* (la définition de la hauteur d'un *trie* est la même que pour un arbre quelconque). On pourra s'appuyer sur les deux fonctions précédentes.

Q20. Implémenter une fonction `size : trie -> int` qui calcule le nombre de nœuds dans un *trie* (attention, il ne s'agit pas du nombre de mots dans le *trie*, tous les nœuds comptent ici).

Considérons la fonction suivante sur les `trie` :

```
let rec f t =
  match t with
   |Node (true, []) -> 1
   |Node (false, []) -> 0
   |Node (b, (letter,child)::tail) -> (f child) + (f (Node (b, tail)))
```

Q21. Indiquer la grandeur que calcule cette fonction en expliquant pourquoi.

Q22. Implémenter une fonction `is_in_trie : char list -> trie -> bool` qui prend en entrée un mot sous forme d'une liste de ses caractères et un `trie` et s'évalue en `true` si le mot est présent dans la structure et `false` sinon.

Q23. Implémenter une fonction `add_to_trie : char list -> trie -> trie` qui prend un mot sous forme d'une liste de ses caractères et un *trie* et qui s'évalue en un *trie* contenant les mots contenus dans le *trie* passé en argument et le mot passé en argument.

Q24. Si on regarde l'exemple du *trie* obtenu en **Q14**, quel problème semble se poser en terme d'efficacité concernant la taille de la structure et d'une éventuelle redondance ?

On souhaite maintenant interpréter un *trie* comme un cas particulier d'automate de manière à ce que les mots reconnus par cet automate soient ceux présents dans le *trie*.

Q25. Préciser ce que sont les différents composants de l'automate (états, états initiaux, états finaux, transitions, alphabet) associé à un *trie* qu'on interprète comme un automate.

Q26. Dessiner l'automate associé au *trie* de **Q14**.

Pour tenter de résoudre le problème évoqué en **Q24**, on se propose de modifier l'automate obtenu à partir d'un *trie* pour y supprimer les états redondants. On peut noter que cet automate est déterministe et donc que la lecture d'une lettre c depuis un état q amène à au plus un état, noté $\delta(q, c)$ s'il existe. Si la lecture de c depuis q n'est pas possible, on parle de blocage.

Soient deux états q et q' de l'automate, Σ son alphabet et δ sa fonction de transition. On considère une relation binaire E sur les états. Deux états q et q' sont en relation selon E, ce que l'on note qEq', si :

- (q et q' sont tous deux finaux) ou (q et q' sont tous deux non finaux),
- $\forall c \in \Sigma$, soit c est un blocage pour q et q', soit $\delta(q, c)E\delta(q', c)$.

Q27. Donner un exemple d'un couple d'états sans transition sortante qui sont en relation pour E dans l'automate de **Q26**. Même chose pour un couple d'états avec transition sortante.

On admet que pour deux états qui sont en relation selon E, on reconnaît le même langage en commençant une lecture depuis l'un ou l'autre de ces états.

Q28. En déduire une manière de simplifier l'automate obtenu en convertissant un *trie* pour réduire au plus son nombre d'états tout en ne changeant pas le langage reconnu. En donner les grandes lignes en expliquant la démarche et illustrer cette démarche sur le *trie* contenant les mots {car, par, cal, pal}.

PROBLÈME 3
Logique implicationnelle

Un système logique est défini par :
- une syntaxe S, qui décrit la façon dont il faut écrire les formules du système,
- une sémantique M, qui explique le sens à leur donner,
- un système de déduction D, qui établit les règles syntaxiques permettant de faire des preuves.

On introduit tout d'abord un premier système logique, le calcul propositionnel $C_P = (S_P, M_P, D_P)$:
- S_P est l'ensemble des formules définies inductivement à partir de $\top$, $\bot$ et des variables propositionnelles à l'aide des connecteurs classiques $\wedge$, $\vee$, $\neg$ et $\rightarrow$,
- M_P est la sémantique standard construite à partir des tables de vérité pour $\wedge$, $\vee$, $\neg$ et $\rightarrow$,
- D_P est la déduction naturelle, dont les règles sont rappelées en annexe 1.

Q29. Si A et B sont deux formules de S_P, montrer que les formules $A \rightarrow B$ et $\neg A \vee B$ sont équivalentes à l'aide d'une table de vérité.

Q30. Si A et B sont deux formules de S_P, les formules $\neg A$ et $A \rightarrow \bot$ sont-elles équivalentes ?

Q31. Si A et B sont deux formules de S_P, établir en la détaillant la table de vérité de la formule :

$$\varphi = ((A \rightarrow B) \rightarrow A) \rightarrow A$$

Que peut-on dire de cette formule ?

On dit qu'un système Σ de connecteurs est complet si et seulement si toute formule de S_P est équivalente à une formule de S_P qui ne fait intervenir que les connecteurs de Σ. Par exemple, $\{\neg, \vee, \wedge, \rightarrow\}$ est un système complet de connecteurs.

Q32. Montrer par induction sur les formules de S_P que le système de connecteurs $\{\neg, \rightarrow\}$ est complet.

On introduit dans la suite du problème un nouveau système logique, le calcul purement implicationnel $C_I = (S_I, M_I, D_I)$:
- Les formules de S_I sont définies par induction de la façon suivante :
  - les symboles $\bot$, $\top$ et les variables propositionnelles sont des formules de S_I,
  - si A et B sont des formules de S_I, alors $A \rightarrow B$ également.

  Autrement dit, on restreint les formules de S_P à celles qui font intervenir uniquement le connecteur $\rightarrow$, d'où le nom du système.
- La sémantique M_I est la sémantique standard.
- Le système de déduction D_I est donné par les règles de déduction suivantes :

$$\frac{}{\Gamma, A \vdash A}\ \text{ax} \qquad \frac{}{\Gamma \vdash ((A \rightarrow B) \rightarrow A) \rightarrow A}\ \text{Peirce}$$

$$\frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \text{elim-}\rightarrow \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\ \text{intro-}\rightarrow$$

Autrement dit, on restreint les règles de la déduction naturelle à l'axiome, l'introduction et l'élimination du $\to$ et on y ajoute un axiome, appelé la loi de Peirce.

Q33. Justifier que pour toute formule de S_P, il existe une formule de S_I qui lui est équivalente.

Q34. Prouver dans le système D_I les séquents suivants :

1. $A \vdash (A \to B) \to B$
2. $A \to B, B \to C \vdash A \to C$
3. $(A \to B) \to C, A \to C \vdash C$. On pourra utiliser l'instance suivante de la loi de Peirce : $((C \to B) \to C) \to C$.

Q35. Rappeler ce que signifie, pour une règle d'un système de déduction, que d'être correcte. Montrer que les règles de D_I sont toutes correctes.

Q36. S'il existe une preuve du séquent $\Gamma \vdash F$ dans un système dont toutes les règles sont correctes, quelle relation sémantique existe entre F et Γ ? Justifier brièvement la réponse.

Q37. À l'aide d'un argument sémantique, expliquer pourquoi ce prétendu arbre de preuve est incorrect. L'ensemble Γ est défini par $\Gamma = \{A \to B, C \to B\}$:

$$\dfrac{\dfrac{\dfrac{}{\Gamma, A \vdash A}\ \text{ax} \qquad \dfrac{}{\Gamma, A \vdash A \to B}\ \text{ax}}{\Gamma, A \vdash B}\ \text{elim-}\to \qquad \dfrac{}{\Gamma, A \vdash C \to B}\ \text{ax}}{\dfrac{\Gamma, A \vdash C}{\Gamma \vdash A \to C}\ \text{intro-}\to}\ \text{elim-}\to$$

On rappelle que si :

$$\dfrac{\Gamma \vdash F_1 \quad ... \quad \Gamma \vdash F_k}{\Gamma \vdash F}$$

est une règle, les séquents $\Gamma \vdash F_1$, ..., $\Gamma \vdash F_k$ au-dessus de la barre se nomment les prémisses de la règle et le séquent $\Gamma \vdash F$ en est sa conclusion.

On dit qu'une règle est *dérivable* dans un système de déduction si on peut construire sa conclusion avec les règles du système de déduction en supposant qu'on a déjà une preuve de ses prémisses. Par exemple, la règle suivante est dérivable dans D_I :

$$\dfrac{\Gamma, A \vdash B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \text{cut}$$

En effet, on peut la prouver en n'utilisant que les règles de D_I via cet arbre :

$$\dfrac{\dfrac{\dfrac{\text{supposé}}{\Gamma, A \vdash B}}{\Gamma \vdash A \to B}\ \text{intro-}\to \qquad \dfrac{\text{supposé}}{\Gamma \vdash A}}{\Gamma \vdash B}\ \text{elim-}\to$$

Q38. Prouver le séquent $(A \to B) \to A, \neg A \vdash A$ dans le système D_P.

Q39. En déduire que la loi de Peirce est dérivable dans le système de déduction D_P. On pourra s'aider de la question précédente et utiliser la règle du raisonnement par l'absurde.

Q40. On note D le système de déduction constitué des règles de D_P auxquelles on a enlevé le raisonnement par l'absurde et ajouté la loi de Peirce. Montrer réciproquement que la règle du raisonnement par l'absurde est dérivable dans D. On pourra utiliser l'instance suivante de la loi de Peirce : $((A \to \bot) \to A) \to A$.

Q41. Que peut-on dire de D ? Justifier brièvement.

Annexe 1
Règles de la déduction naturelle

- Axiome :

$$\frac{}{\Gamma, A \vdash A}\ \text{ax}$$

- Affaiblissement :

$$\frac{\Gamma \vdash A}{\Gamma, B \vdash A}\ \text{aff}$$

- Introduction et élimination de l'implication :

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \text{intro-}\to \qquad \frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \text{elim-}\to$$

- Introduction et éliminations du et :

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \text{intro-}\wedge \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \text{elim-}\wedge \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \text{elim-}\wedge$$

- Introductions et élimination du ou :

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \text{intro-}\vee \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \text{intro-}\vee \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ \text{elim-}\vee$$

- Introdution et élimination du non :

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \text{intro-}\neg \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\ \text{elim-}\neg$$

- Élimination du $\bot$:

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \text{elim-}\bot$$

- Raisonnement par l'absurde :

$$\frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\ \text{ra}$$

FIN

IMPRIMERIE NATIONALE – 26 1005 – D'après documents fournis

Travailler les chapitres de ce sujet

Prepa Help suit le programme de MP chapitre par chapitre : cours détaillé, méthodes, pièges, exercices et problèmes corrigés pas à pas, tirés des concours. L'essai gratuit donne accès aux chapitres offerts de ta filière, sans carte bancaire.

Chapitres liés à ce sujet

- Architecture de l'ordinateur et représentation des nombres
- Bases de Python : expressions, types, instructions, fonctions
- Complexité des algorithmes itératifs et récursifs
- Récursivité
- Langage OCaml : expressions, types, filtrage, récursivité
- Logique, ensembles et applications

Ouvrir l'essai gratuit

Scanne le code ou clique sur le bouton.

prepahelp.com/#essai

Une question sur le concours ou sur ta filière :
écrire sur WhatsApp

Les autres années et épreuves : prepahelp.com/annales/e3a/mp