rédigé par
Fatoumata DIAL
Mame Diarra DIENG
Ndeye Aïta SECK
Anna NDOYE

Chapitre 2:

Indécidablité

Un cas d'indécidabilité désigne un problème mathématique ou informatique pour lequel il est impossible de concevoir un algorithme général qui puisse fournir une réponse correcte (oui ou non) dans un temps fini pour toutes les instances du problème. Ces problèmes sont dits indécidables.

Définitions clés :

1. Décidable : Un problème est décidable s'il existe un algorithme qui, pour toute entrée possible, termine toujours (dans un temps fini) et fournit la bonne réponse.
2. Indécidable : Un problème est indécidable s'il n'existe aucun algorithme capable de résoudre toutes les instances du problème dans un temps fini. Autrement dit,il est impossible de concevoir un programme qui détermine systématiquement la réponse correcte (oui ou non) pour toutes les entrées.

Exemple classique : Le problème de l'arrêt

L'un des exemples les plus connus d'indécidabilité est le problème de l'arrêt (*Halting Problem*), démontré par Alan Turing en 1936.

Enoncé :
On cherche à déterminer, pour un programme informatique PP et une entrée xx, si P(x) s'arrêtera (terminera) ou continuera à s'exécuter indéfiniment.

Résultat :
Il n'existe aucun algorithme universel capable de résoudre ce problème pour tous les programmes P et toutes les entrées x. C'est un problème indécidable.

Caractéristiques des problèmes indécidables

1. Inexistence d'algorithme général :
   Un problème est indécidable si aucune machine de Turing (ou programme informatique) ne peut résoudre toutes les instances du problème.
2. Réduction :
   Les problèmes indécidables sont souvent démontrés par réduction : si un problème A est indécidable, et si B peut être transformé en A de manière calculable, alors B est aussi indécidable.
3. Liens avec la récursivité :
   Les problèmes indécidables sont souvent associés à des ensembles ou des fonctions non récursives, qui ne peuvent pas être entièrement décrits par un algorithme.

Importance

1.Limites de l'informatique :
Elle montre qu'il existe des problèmes intrinsèquement impossibles à résoudre par des machines ou des algorithmes, quelle que soit leur puissance.

2.Théorie de la complexité :
L'indécidabilité fournit un cadre pour comprendre la distinction entre ce qui est calculable et ce qui ne l'est pas.

3.Applications pratiques :
De nombreux problèmes en sécurité informatique, vérification de programmes ou intelligence artificielle ont des aspects indécidables. Cela nécessite de développer des approximations.

Exemple :Fonction Morris

Démonstration : La fonction de Morris ne termine pas

Analyse de la fonction Morris

La fonction est définie comme suit :

· - Morris(0, n) = 1, avec n ∈ ℕ

· - Morris(m, n) = Morris(m-1, Morris(m, n)), si m > 0

Étape 1 : Étude de la récursivité

Pour m > 0, la définition de Morris(m, n) appelle récursivement Morris(m, n) lui-même comme un argument. Autrement dit, à chaque étape, la fonction s'auto-réutilise dans un processus qui ne diminue pas nécessairement les paramètres de manière strictement décroissante.

Prenons un exemple avec m = 1 et n = 2 :

Morris(1, 2) = Morris(0, Morris(1, 2))

Dans cette expression, pour évaluer Morris(1, 2), nous devons d'abord évaluer Morris(1, 2) dans son propre argument. Cela crée une boucle infinie.

Étape 2 : Absence de décroissance

Pour qu'une fonction récursive termine, il faut que les paramètres évoluent de manière strictement décroissante vers un cas de base. Cependant :

1. 1. La valeur de m reste constante dans l'appel Morris(m, n), et donc ne tend pas vers 0.

2. 2. L'argument n devient Morris(m, n), qui dépend lui-même d'une nouvelle instance de la fonction.

Ainsi, aucun des paramètres ne converge vers le cas de base m = 0.

Étape 3 : Généralisation et preuve par induction

Pour formaliser davantage, on peut essayer une démonstration par induction pour montrer que Morris(m, n) ne termine jamais pour m > 0.

· Base : m = 0

· Pour m = 0, Morris(0, n) = 1, ce qui termine en une seule étape.

· Hypothèse d'induction : Morris(m, n) ne termine pas pour un m > 0.

· Étape inductive :

· Pour m+1, on a :

Morris(m+1, n) = Morris(m, Morris(m+1, n))

L'évaluation de Morris(m+1, n) nécessite une évaluation complète de Morris(m+1, n) comme argument de Morris(m, ·), ce qui conduit à une récursion infinie par hypothèse.

Conclusion

La fonction de Morris est un exemple marquant qui met en lumière l'indécidabilité, un concept fondamental de l'informatique théorique. Elle démontre qu'il est impossible de concevoir un algorithme universel capable de déterminer, pour des entrées spécifiques mmm et nnn, si la fonction s'arrête ou s'exécute indéfiniment. Cela reflète directement le **problème de l'arrêt** formulé par Alan Turing.

Points clés :

1. **Problème de l'arrêt** : Turing a prouvé qu'aucun algorithme général ne peut résoudre le problème de l'arrêt pour tous les programmes possibles. La fonction de Morris en est une manifestation, illustrant cette limitation théorique.
2. **Indécidabilité** : L'impossibilité de prédire si la fonction termine pour toutes les valeurs de mmm et nnn met en évidence que certaines questions sont **non calculables**. Ces problèmes dépassent les capacités des algorithmes, quel que soit leur niveau de sophistication.

3. **Limitation de la calculabilité** : Ce comportement révèle une **barrière fondamentale** dans ce que les machines peuvent résoudre. Même pour des fonctions qui semblent simples, leur comportement peut être imprévisible.

Chapitre 3

1- Analyse des nombres de pesées dans le pire des cas

1. Pour P_1 :
Le nombre de pesées dans le pire des cas est donné par la formule :
$n(n-1)/2$

Démonstration :
- Si chaque pierre est comparée à toutes les pierres plus lourdes qu'elle, le processus correspond à effectuer une comparaison pour chaque paire unique de pierres.
- Le nombre total de paires dans un ensemble de n éléments est donné par la combinaison C(n, 2), qui se calcule comme : $C(n,2) = n(n-1)/2$.
- Cela représente le nombre maximum de comparaisons nécessaires.

Exemple :
Pour n = 4 (4 pierres) :
- Comparons chaque pierre avec les autres :
  - Comparer la 1ère pierre avec les 3 restantes.
  - Comparer la 2ème pierre avec les 2 restantes.
  - Comparer la 3ème pierre avec la dernière.
- Total : C(4, 2) = 4(4-1)/2 = 6 comparaisons.

---

2. Pour P_2 :
Le nombre de pesées dans le pire des cas est donné par la formule :
$(n-1)\cdot n!$

Démonstration :
- n! représente le nombre total de permutations possibles pour n éléments. Cela reflète toutes les manières d'organiser les pierres.
- Dans le pire des cas, on doit tester chaque permutation pour déterminer si c'est la bonne.
- Chaque permutation nécessite n-1 pesées (car il faut comparer successivement chaque pierre à la suivante dans l'ordre pour valider la permutation).
- Donc, au total : $(n-1)\cdot n!$

Exemple :
Pour n = 3 (3 pierres) :
- Nombre de permutations : 3! = 6.
- Pour tester une permutation, il faut 2 pesées (comparer 1ère ↔ 2ème, puis 2ème ↔ 3ème).
- Total : $(3-1)\cdot 3! = 2\cdot 6 = 12$ pesées.

---

3. Pour P_3 :
Le nombre de pesées dans le pire des cas est donné par la formule :
$2^n - 2$

Démonstration :
- Le problème suit une approche de recherche binaire où chaque pesée divise l'ensemble des possibilités en deux.
- Le nombre total de combinaisons possibles avec n éléments est 2^n (chaque pierre peut être dans l'un des deux tas lors de chaque pesée).
- Dans le pire des cas, toutes les combinaisons sauf la dernière doivent être testées.
- Par conséquent, le nombre de pesées est $2^n - 2$, car on exclut la combinaison initiale.

Exemple :
Pour n = 3 (3 pierres) :
- Nombre de combinaisons : $2^3 = 8$.
- Dans le pire des cas, il faut explorer $8 - 2 = 6$ pesées avant de trouver la solution.

2- Analyse des Complexités Algorithmiques

1. Complexité logarithmique (Θ(log n))

Description :

La complexité logarithmique signifie que le temps d'exécution d'un algorithme augmente proportionnellement au logarithme de la taille de l'entrée (à une base fixée, souvent 2). Cela se produit lorsque le problème est divisé en sous-problèmes plus petits, comme dans la recherche dichotomique.

Exemple : Recherche Dichotomique (Binary Search)
Trouver la position d'un élément dans un tableau trié.

Algorithme :

```
Procédure     RechercheDichotomique(tableau, n, cible)
        gauche ← 0    Initialiser l'indice gauche
        droite ← n - 1          Initialiser l'indice droit
                Tant que gauche ≤ droite    Faire
                        milieu ← (gauche + droite) // 2 Calculer l'indice du milieu
                                Si tableau[milieu] = cible    Alors
                                        Retourner milieu
                                        Sinon Si tableau[milieu] <  cible Alors
                                        gauche ← milieu + 1
                                        Sinon
                                        droite ← milieu - 1
                                Fin Si
                Fin Tant que
 Retourner -1


        Fin Procédure
```

Complexité :
Temps d'exécution : Θ(log n), car à chaque étape, l'espace de recherche est divisé par deux.

2. Complexité linéaire (Θ(n))

Description :

La complexité linéaire signifie que le temps d'exécution augmente proportionnellement à la taille de l'entrée. Cette complexité se retrouve souvent dans les algorithmes parcourant tous les éléments d'un tableau ou d'une liste.

Exemple : Recherche du Maximum
Trouver le plus grand élément d'un tableau.

Algorithme :

```
Procédure TrouverMaximum(tableau,      n)


 max ← tableau[0]
 Pour i ← 1 à n - 1 Faire
         Si tableau[i] > max   Alors
                  max ← tableau[i]
          Fin Si
 Fin Pour

 Retourner max


Fin Procédure
```

Complexité :
Temps d'exécution : Θ(n), car chaque élément est parcouru une fois.

3. Complexité quasi-linéaire (O(n log n))

Description:

Cette complexité est typique des algorithmes de tri efficaces. Elle reflète un temps d'exécution qui croît plus rapidement que linéairement mais moins rapidement que quadratiquement.

Exemple : Tri Rapide

Problème : Trier un tableau d'éléments.

Algorithme :

Procédure TriRapide(tableau, début, fin)

Si début < fin Alors

pivotIndex ← Partition(tableau, début, fin)

TriRapide(tableau, début, pivotIndex - 1)

TriRapide(tableau, pivotIndex + 1, fin)

Fin Si

Fin Procédure

Fonction Partition(tableau, début, fin)

pivot ← tableau[fin]

i ← début - 1

Pour j ← début à fin - 1 Faire

Si tableau[j] ≤ pivot Alors

i ← i + 1

Échanger tableau[i] et tableau[j]

Fin Si

Fin Pour

Échanger tableau[i + 1] et tableau[fin]

Retourner i + 1

Fin Fonction

Complexité :
Temps d'exécution moyen : O(n log n).
Temps d'exécution dans le pire cas (tableau déjà trié) : O(n²).

4. Complexité polynomiale ($O(n^k)$)

Description :

La complexité polynomiale reflète une croissance proportionnelle à une puissance de la taille de l'entrée. Ces algorithmes peuvent devenir rapidement inefficaces pour de grandes entrées.

Exemple :

Procédure MultiplierMatrices(A, B, n)

Initialiser une matrice C de dimensions n × n avec tous les éléments à 0

Pour i ← 0 à n - 1 Faire Parcourir les lignes de A

Pour j ← 0 à n - 1 Faire Parcourir les colonnes de B

C[i][j] ← 0 Initialiser l'élément C[i][j]

Pour k ← 0 à n - 1

C[i][j] ← C[i][j] + A[i][k] × B[k][j]

Fin Pour

Fin Pour

Fin Pour

Retourner C

Fin Procédure

Complexité :
Temps d'exécution : $O(n^3)$ car trois boucles imbriquées parcourent les éléments.

3- Démonstration des Propriétés de Domination et de Relation d'Équivalence

1. Prouver que la relation de domination est réflexive et transitive

Réflexivité

La relation $f(n) = O(g(n))$ signifie qu'il existe une constante $c > 0$ et un entier n_0 tel que pour tout $n \geq n_0$, $f(n) \leq c \cdot g(n)$. Si on considère $f(n) = g(n)$, alors on peut choisir $c = 1$ et $n_0 = 0$. Ainsi, $f(n) \leq 1 \cdot g(n)$ pour tout $n \geq n_0$, ce qui montre que $f(n)$ est dominée par elle-même.
Conclusion : la relation est réflexive.

Transitivité

Supposons que $f(n) = O(g(n))$ et $g(n) = O(h(n))$. Cela signifie :
- $\exists\ c_1 > 0, n_1 \in \mathbb{N}, \forall\ n \geq n_1, f(n) \leq c_1 \cdot g(n)$,
- $\exists\ c_2 > 0, n_2 \in \mathbb{N}, \forall\ n \geq n_2, g(n) \leq c_2 \cdot h(n)$.

Prenons $n_0 = \max(n_1, n_2)$. Pour $n \geq n_0$, on a :
$f(n) \leq c_1 \cdot g(n) \leq c_1 \cdot (c_2 \cdot h(n)) = (c_1 \cdot c_2) \cdot h(n)$.
Ainsi, $f(n) = O(h(n))$.
Conclusion : la relation est transitive.

2. Prouver que la relation Θ est une relation d'équivalence

Réflexivité

Par définition, $f(n) = \Theta(g(n))$ si $f(n) = O(g(n))$ et $g(n) = O(f(n))$. Comme $f(n) = O(f(n))$ (réflexivité prouvée ci-dessus), on a $f(n) = \Theta(f(n))$.
Conclusion : la relation est réflexive.

Symétrie

Si $f(n) = \Theta(g(n))$, cela signifie $f(n) = O(g(n))$ et $g(n) = O(f(n))$. Mais cette définition est symétrique : si $g(n) = O(f(n))$ et $f(n) = O(g(n))$, alors $g(n) = \Theta(f(n))$.
Conclusion : la relation est symétrique.

Transitivité

Supposons que $f(n) = \Theta(g(n))$ et $g(n) = \Theta(h(n))$. Cela signifie :
- $f(n) = O(g(n))$ et $g(n) = O(f(n))$,
- $g(n) = O(h(n))$ et $h(n) = O(g(n))$.

De la transitivité de O (prouvée ci-dessus), on obtient $f(n) = O(h(n))$ et $h(n) = O(f(n))$. Ainsi, $f(n) = \Theta(h(n))$.
Conclusion : la relation est transitive.

Conclusion

- La relation de domination O est réflexive et transitive.
- La relation Θ est une relation d'équivalence (réflexive, symétrique et transitive).

4-Exprimons tn (le temps d'exécution de cet algorithme) dans le pire des cas ainsi

que dans le meilleur des cas Analyse des Temps d’Exécution : Tri par Insertion

Le tri par insertion fonctionne en parcourant une liste d’éléments, un par un, en plaçant chaque élément dans sa position correcte parmi les éléments déjà triés. Cette analyse examine le temps d’exécution dans les deux scénarios possibles : le meilleur des cas et le pire des cas.

1. Meilleur des cas (T(n))

Le meilleur des cas survient lorsque les éléments de la liste sont déjà triés dans l’ordre croissant.

Déroulement :

- À chaque itération, l’algorithme compare l’élément courant avec le précédent.
- Aucune réorganisation n’est nécessaire, car tous les éléments sont déjà dans le bon ordre.
- Le nombre total de comparaisons effectuées est minimal.

Complexité :

- Pour chaque élément i, une seule cpmparaison est effectuée.
- Temps d’exécution total : $T(n) = \Theta(n)$.

2. Pire des cas (T(n))

Le pire des cas survient lorsque les éléments de la liste sont triés dans l’ordre décroissant.

Déroulement :

- Chaque nouvel élément doit être inséré à la première position de la liste triée.
- Cela nécessite de déplacer tous les éléments déjà triés d’une position vers la droite avant d’insérer l’élément courant.

Calcul du temps d’exécution :

- Pour le 2^e élément : 1 comparaison et 1 déplacement.
- Pour le 3^e élément : 2 comparaisons et 2 déplacements.
- Pour le i-ème élément : (i-1) comparaisons et déplacements.
- Nombre total d’opérations : $1 + 2 + 3 + ... + (n-1) = n(n-1)/2$.

Complexité :

- Temps d’exécution total : $T(n) = \Theta(n^2)$.

Résumé des Complexités

Scénario	Temps d’exécution T(n)	Description
Meilleur des cas	$\Theta(n)$	Liste déjà triée ; une seule comparaison par élément.
Pire des cas	$\Theta(n^2)$	Liste triée à l’envers ; maximum de comparaisons et déplacements.