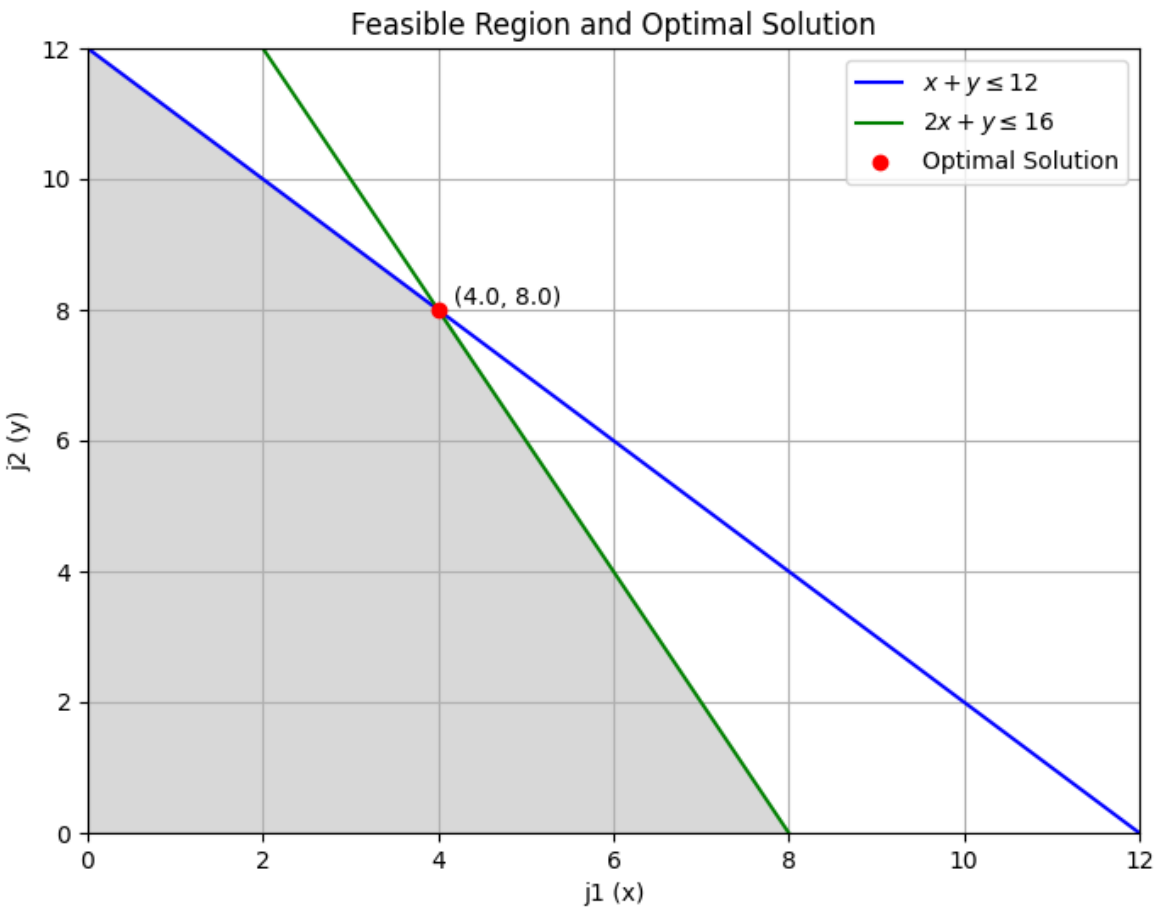
PROJET-AI-2

Programmation Linéaire

- La programmation linéaire permet de répondre à plusieurs besoins :
- Elle permet de :
    1. Trouver la solution optimale pour un problème ciblé.
    2. La programmation linéaire permet de réduire le nombre de ressources nécessaires.
    3. Réduire le nombre de décisions possibles.
    4. Permet de réduire le problème à sa plus simple équation.
- On utilise alors les fonctions linéaires pour déterminer la zone à cibler.
    - En délimitant la zone avec les fonctions (lignes), la programmation linéaire permet à notre modèle IA de ne pas gaspiller d'énergie sur des problèmes qui ne mènent pas à la résolution en temps optimal (meilleure résolution).
- Sur la figure suivante (fig. 1), on peut voir le résultat du code "ProgrammationLineaire.py".



- Comme on peut le voir sur le graphique ci-dessus, la solution optimale se situe à l'intersection des deux fonctions.
    - Dans le cas où on aurait calculé plusieurs fonctions, toute fonction qui se trouve en dehors de la délimitation devient inutile (redondance).
- **Ces éléments permettent de s'attaquer à des problèmes d'envergure !**

Temps algorithmique (Grand O)

1. Des boucles imbriquées une à l'intérieur de l'autre (3 boucles au total) ?

   Reponses : O(n^3)

   Justification : Parce que à chaque itération de la première boucle, la deuxième et la troisième vont s'exécuter également, on ne peut pas sélectionner une seule boucle et on doit plonger dans chacune d'elles. (Similaire au problème du cambrioleur)
2. Une variable directement utilisée ?

   Reponses : Complexite constante "O(1)"

   Justification : La raison étant que la complexité reste la même car la variable est directe et constante. Lorsqu'une variable ne bouge pas, sa réponse reste invariable.
3. Une recherche d'un arbre binaire ?

   Reponses : On utiliserait "O(log(n))"

   Justifications : Comme on a vu avec la suite de Fibonacci, lorsqu'une hiérarchie existe, il est possible de la "factoriser".

- Il ne faut surtout **PAS** employer un algorithme glouton dans le cas où l'arbre de décision est profond et large car la force brute, c'est le procédé d'essayer chacune des solutions (une par une) et déterminer la meilleure, ce qui prend un temps exponentiel O(n^e).

Récursivité

- Les fonctions récursives sont des fonctions qui sont en mesure de s'appeler elles-mêmes, ce qui signifie qu'elles peuvent résoudre des problèmes à l'intérieur de la boucle, puis, lorsque la boucle se termine, se rappeler elles-mêmes pour recommencer le processus.

Exemple "bash" de mon scraper/downloader Steam (programmé il y a un an) :

- Voici un exemple qui n'était pas dans le cours :

```
read -p "Press [ENTER] to Continue to : Automated Steam Download"
echo "Starting Automated Steam Download from : 'appid' file."
while read p; do
  echo $p > APP_ID
  awk -F' ' 'NR==8 {$2=""echo "'$(<APP_ID)'"""}1' steamcmd_buffer >
steamcmd_buffer.run
  steamcmd +runscript /home/qgp/Videos/steamcmd_buffer.run
wait
  sleep 3
done <appid
```

1. Le script itère à travers une liste (fichier) "appid" : `done <appid`
2. On "echo" les lignes une par une à chaque récursion du script et on stocke la présente dans "APP_ID", on recommence à partir de la ligne à laquelle on est rendu (Programmation Dynamique)
3. Le langage de script "AWK" permet d'insérer la variable qu'on a capturée dans la 8ème rangée, 2ème colonne du fichier "steamcmd_buffer" et on copie le résultat en tant que "steamcmd_buffer.run"
4. On exécute ensuite le script qui a été créé, celui-ci installe l'application qui était contenue dans la variable "APP_ID" qui correspond à l'identifiant de l'application.
5. La fonction effectue sa récursion jusqu'à ce que le fichier texte "appid" soit terminé.

Exemple "Python" relatif à notre cas de figure "Recursivite.py"

- Le parallèle avec le script précédent est très clair, mais je vais le réexpliquer.

```
# Fonction récursive qui additionne un nombre 'n' avec lui-même en faisant
des sauts de -1 :
def add_recursive(n):
    # Si le nombre arrive à valoir 1, on arrête la fonction récursive.
    if n == 1:
        return 1

    # On retourne un appel de la fonction elle-même avec son paramètre 'n'
diminué de 1, en additionnant ce résultat à un deuxième appel de la
fonction.
    return add_recursive(n - 1) + add_recursive(n - 1)
```

1. On définit la fonction "add_recursive(n)".
2. On additionne un nombre 'n' avec lui-même en faisant des sauts de -1.
3. Si le nombre vaut 1, on arrête la fonction récursive.
4. On fait un retour récursif où on appelle la fonction par elle-même avec le paramètre diminué de 1, en additionnant un deuxième appel de la même fonction.

- La fonction bash est plus complexe, mais revient au même (mis à part la sauvegarde dynamique).
- Une fonction récursive est une bonne façon de résoudre un sudoku.
- Lorsqu'on programme une fonction récursive, il faut aussi penser à sa condition d'arrêt, sans quoi, la récursion pourrait continuer à l'infini.

Programmation Dynamique

- La programmation dynamique est une façon de résoudre des problèmes par étapes en réutilisant les grands concepts récursifs : On divise les problèmes en sous-problèmes, puis on essaie de résoudre les problèmes les plus petits en premier (de l'intérieur vers l'extérieur), ce qui permet de créer des structures de résolution avec un potentiel pour l'automatisation.
- La décomposition de ces problèmes permet de stocker les résultats intermédiaires, ce qui permet la mémorisation des problèmes et une résolution "au cas par cas".
  - Elle permet de réagir au résultat obtenu durant le test récursif.
  - En ajoutant une fonction de mémorisation, cela nous permet de sauter par-dessus les résultats qui sont inutiles (non optimaux).
  - Lorsque nous avons besoin de la bonne réponse, nous pouvons demander au dictionnaire la "clé" plutôt que de recalculer notre jeu de données pour la trouver.
- Prenons le script "ProgrammationDynamique.py" en exemple :

```
def add_recursive_mem(n, memo={}):
    if n == 1:
        return 1
    elif n in memo:
        return memo[n]
    memo[n] = add_recursive_mem(n - 1) + add_recursive_mem(n - 1)
    return memo[n]
```

- Le script demande un chiffre à l'utilisateur et effectue ensuite l'opération.
- Celui-ci permet :
  1. D'**additionner** un nombre 'n' avec lui-même en faisant des sauts de -1. (Pris de "Recursivite.py")
  2. De **sauvegarder** l'opération dans un dictionnaire "memo"
  3. De vérifier le dictionnaire pour éviter la **répétition**
  4. Lorsqu'on arrive à 1, le script s'arrête
- Dans un cas réel, ceci permettrait d'économiser des ressources monstres là où l'information est rangée de façon séquentielle, car on n'aurait pas besoin de remonter la boucle au complet, juste de demander la réponse au dictionnaire.

Travail présenté par : Ottoniel Castaneda Aguilar et Sean Calcé

Github : https://github.com/OttonielCA/AI_TP2/

Sources/Documentation :

1. https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/
2. https://www.geeksforgeeks.org/recursive-neural-network-in-deep-learning/
3. Anthropic Claude 3 (Haiku)