UNIVERSITÉ SORBONNE PARIS NORD

Rapport du devoir de Conception d'Algorithmes

L2 double-licence mathématiques et informatique

Par :

- Wilen YAICI 12214142
- Mohammed Yanis TAKBOU 12206365

Partie 1 : Réponses aux Questions

Exercice 1 : Reproduction

Question 1: Format du fichier instances.csv

Le fichier `instances.csv` créé par le script est un fichier CSV qui contient différentes instances du problème du sac à dos. Chaque ligne représente une instance avec les colonnes suivantes :

- Capacité maximale du sac : La capacité maximale en termes de poids que le sac à dos peut contenir.
- Nombre d'objets : Le nombre total d'objets disponibles pour être placés dans le sac à dos.
- Pour chaque objet de cette instance s'accompagne la paire :
  - Poids des objets : Une liste séparée par des virgules indiquant le poids de chaque objet.
  - Valeurs des objets : Une liste séparée par des virgules indiquant la valeur de chaque objet.
- Ce format permet de facilement générer et analyser des instances de différentes tailles et complexités, nous permettant d'extensivement tester nos différents types d'algorithmes.

```
40,5,1,1,3,4,2,2,3,3,1,3
```

Exemple d'une instance du problème de sac à dos.

Dans cette instance, la capacité maximale est de 40, il y a 5 objets au total.
Le premier objet a pour poids 1 et pour valeur 1.
Le deuxième a pour poids 3 et valeur 4.
Le troisième objet a pour poids 2 et pour valeur 2.
Le quatrième a pour poids 3 et valeur 3.
Le cinquième objet a pour poids 1 et pour valeur 3.

Question 2: Complexité temporelle et utilisation de la mémoire des algorithmes de backtracking

Les deux versions de l'algorithme de backtracking ont les complexités temporelles et utilisations de mémoire suivantes :

- **Complexité temporelle :** La complexité temporelle de ces algorithme est de l'ordre de $\mathcal{O}(2^n)$, où n représente le nombre d'objets. Cela s'explique par le fait que les algorithmes explorent exhaustivement toutes les combinaisons possibles d'objets pour trouver la solution optimale au problème du sac à dos. Avec ***n*** objets, chaque objet peut être inclus ou exclu du sac, générant ainsi deux choix par objet. Par conséquent, il y a 2^n combinaisons distinctes à explorer pour évaluer toutes les possibilités. Cette exploration exhaustive est là pour garantir l'obtention de la solution optimale.
- **Utilisation de mémoire :** La complexité spatiale de ces algorithmes est principalement due à l'utilisation de la pile d'appels récursifs. Les variables locales (et globales pour la version 1) utilisées dans les fonctions récursives occupent un espace mémoire constant, indépendamment de la taille de l'entrée. Cependant, chaque appel récursif ajoute un nouveau cadre de pile à la mémoire, contenant les variables locales de chaque appel de fonction et l'adresse de retour, ce qui contribue à une complexité spatiale de $\mathcal{O}(n)$, où n est le nombre d'objets à considérer. La profondeur maximale de la récursion est égale à n, car chaque appel récursif explore une branche de décision pour chaque objet, aboutissant à une utilisation de la mémoire linéairement proportionnelle au nombre d'objets. Ainsi, même si les variables globales utilisent un espace constant, l'empreinte mémoire totale dépend directement de la taille de l'entrée en raison de la pile d'appels récursifs.
- **Variables globales :** Dans la première version, l'utilisation de variables globales facilite la programmation en évitant de passer de nombreux paramètres entre les fonctions. Cependant, cela peut rendre le code moins flexible et plus compliqué à comprendre car les changements aux variables globales peuvent affecter différentes parties du programme de manière inattendue. Dans la deuxième version, les variables globales ont été évitées, ce qui rend le code plus clair. En utilisant des paramètres de fonction pour transmettre les données nécessaires, cette approche rend le code plus indépendant et moins sujet à des erreurs imprévues. De plus, cela facilite le processus de débogage en rendant le code plus transparent et en facilitant l'identification des erreurs potentielles.

Question 3: Limite de l'algorithme de backtracking

L'algorithme de backtracking devient trop lent lorsque le nombre d'objets dépasse environ 30. À ce point, le temps nécessaire pour explorer toutes les combinaisons possibles devient impraticable en raison de la croissance exponentielle du nombre de solutions à examiner. Pour des instances avec plus de 30 objets, le nombre total de combinaisons possibles (2^{30} = 1 073 741 824) est déjà significatif, et prend un temps anormalement long (environ 4 minutes sur nos machines), ce qui rend l'algorithme pratiquement inutilisable pour de plus grandes instances.

Exercice 2 : Meilleur Backtracking ?

Question 4: Efficacité de la stratégie d'élagage

- **Stratégie d'élagage :**

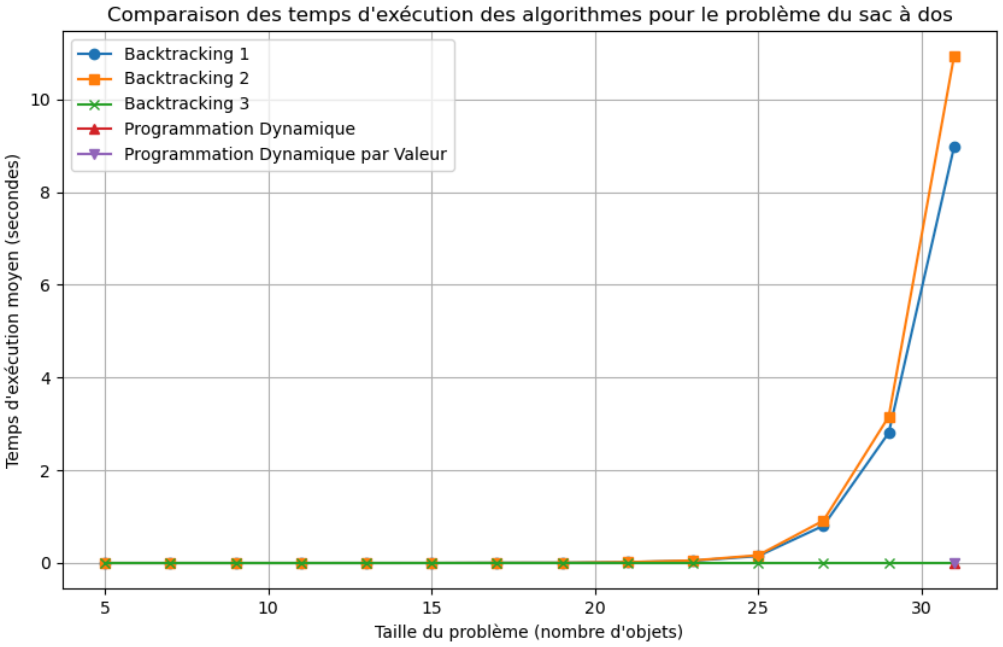
  Pour améliorer l'efficacité de l'algorithme de backtracking, nous avons introduit une borne supérieure. Cette borne est calculée en ajoutant à la valeur courante la somme des valeurs des objets restants qui pourraient potentiellement être inclus dans le sac à dos. Si cette borne supérieure est inférieure à la meilleure solution trouvée jusqu'à présent, la branche correspondante de l'arbre de recherche est élaguée (c'est-à-dire, elle n'est pas explorée davantage).

```
int upper_bound = g_currentValue;
for (int i = idx; i < g_n; ++i)
{
    upper_bound += g_itmes[i].value;
}

if (upper_bound <= g_bestValue)
{
    return;
}
```

- **Efficacité :**

  L'ajout de cette borne supérieure permet de réduire significativement le nombre de solutions à examiner, car de nombreuses branches de l'arbre de recherche peuvent être abandonnées tôt si elles ne peuvent pas mener à une meilleure solution que celle déjà trouvée. Empiriquement, cela se traduit par une réduction notable du temps d'exécution, en particulier pour des instances de taille moyenne à grande. Par exemple, pour des instances avec 15 à 20 objets, le temps d'exécution peut être réduit de plusieurs secondes, comme le montre le graphique ci dessous:



On voit que la fonction de backtracking prend autant de temps que la fonction de programmation dynamique pour des valeurs inférieures à 31.

Exercice 3 : Programmation Dynamique Basée sur la Valeur

Question 5: Propriété de la sous-structure optimale et définition récursive de dp[v]

Propriété de la sous-structure optimale :

- Si l'ensemble des objets Oi1, Oi2, ..., Oik est une solution de poids minimal pour atteindre une valeur optimale v pour un sac à dos supportant un poids maximum de W, alors Oi1, Oi2, ..., Oik-1 est une solution de poids minimal pour atteindre la valeur v - vi pour le sous-ensemble d'objets {Oj : 1 ≤ j ≤ ik-1} et pour un sac à dos supportant un poids maximum de W - wi.
- Démonstration : En raisonnant par l'absurde, supposons qu'il existe un sous-ensemble d'objets O' ⊆ {O1, O2, ..., Oik-1} qui soit une solution de poids minimal pour atteindre la valeur v - vi avec un poids inférieur à dp[v - vi]. En ajoutant l'objet Oik à cette solution, nous obtiendrions un poids total inférieur à dp[v - vi] + wi pour atteindre la valeur v, ce qui contredit notre hypothèse que dp[v] est le poids minimal pour atteindre v. Donc, dp[v - vi] est effectivement la solution optimale pour le sous-ensemble d'objets {Oj : 1 ≤ j ≤ ik-1}.
- De la propriété de sous-structure optimale précédente, on peut observer le suivant :
- Pour chaque valeur cible v, nous devons considérer toutes les combinaisons d'objets et les poids associés.
- En utilisant une table de programmation dynamique dp, où dp[v] représente le poids minimal pour atteindre la valeur v, nous pouvons définir récursivement cette relation.
- La relation de récurrence peut être formulée ainsi :
    - dp[v] = min(dp[v], dp[v - vi] + wi), où dp[v] est le poids minimal pour atteindre la valeur v, vi est la valeur de l'objet i, et wi est le poids de l'objet i.
- Ainsi, la fonction dp[v] pour chaque valeur v peut être construite en utilisant les solutions optimales des sous-problèmes. Cette approche nous permet de déterminer le poids minimal nécessaire pour obtenir chaque niveau de valeur possible jusqu'à la valeur maximale atteignable V.
- En conclusion, la solution optimale pour atteindre une valeur v peut être obtenue à partir des solutions optimales pour les valeurs v - vi, ce qui valide la propriété de sous-structure optimale du problème.

Définition récursive :

Comme vu précédemment, on a:

- La définition récursive pour $dp[v]$, où $dp[v]$ représente le poids minimal nécessaire pour obtenir une valeur v, est : $dp[v]=\min(dp[v],dp[v-vi]+pi)$
  où vi est la valeur de l'objet i et pi est son poids.
  Cette relation récursive signifie que pour atteindre la valeur v, nous pouvons soit :
  - Ne pas inclure l'objet i, auquel cas $dp[v]$ reste inchangé.
  - Inclure l'objet i, auquel cas nous devons ajouter son poids pi à la solution optimale pour la valeur $v-vi$.

Implémentation en C :

```c
int *knapsackdpv(int V, Item items[], int n)
{
    int *dpv = malloc((V + 1) * sizeof(int));
    dpv[0] = 0;
    for (int i = 1; i <= V; ++i)
    {
        dpv[i] = INT_MAX;
    }
    for (int i = 0; i < n; ++i)
    {
        for (int j = V; j >= items[i].value; --j)
        {
            if (dpv[j - items[i].value] != INT_MAX)
                dpv[j] = min(dpv[j - items[i].value] + items[i].weight, dpv[j]);
        }
    }
    return dpv;
}
int knapsackvalue(int W, Item items[], int n)
{
    int V = 0;
    for (int i = 0; i < n; ++i)
    {
        V += items[i].value;
    }
    int *dpv = knapsackdpv(V, items, n);
    int i = V;
    while (dpv[i] > W && i > 0)
    {
        i--;
    }
    int result = i;
    free(dpv);
    return result;
}
```

Question 6: Complexité temporelle de l'algorithme de programmation dynamique

- **Complexité temporelle :**

  L'algorithme de programmation dynamique basé sur la valeur a une complexité temporelle de $O(nV)$, où n est le nombre d'objets et V est la somme des valeurs de tous les objets. Cette complexité découle du fait que nous devons remplir une table de taille $n \times V$, en considérant chaque objet pour chaque valeur possible.

- **Complexité spatiale :**

  La complexité spatiale de cet algorithme est proportionnelle à $O(V)$, car nous utilisons une table pour stocker les poids minimaux nécessaires pour atteindre chaque valeur possible jusqu'à la valeur maximale V.

Le script graphique a été modifié pour inclure cette nouvelles fonction :

```
# Préparation du graphique pour comparer les temps d'exécution des trois algorithmes
plt.figure(figsize=(10, 6))
plt.plot(df['Taille'], df['BT1'], label='Backtracking 1', marker='s')
plt.plot(df['Taille'], df['BT2'], label='Backtracking 2', marker='o')
plt.plot(df['Taille'], df['BT3'], label='Backtracking 3', marker='x')
plt.plot(df['Taille'], df['DP'], label='Programmation Dynamique', marker='v')
plt.plot(df['Taille'], df['DPV'], label='Programmation Dynamique par Valeur', marker='^')
# Ajout de titres et légendes pour le graphique
plt.title("Comparaison des temps d'exécution des algorithmes pour le problème du sac à dos")
plt.xlabel('Taille du problème (nombre d\'objets)')
plt.ylabel('Temps d\'exécution moyen (secondes)')
plt.legend()
```

Exercice 4 : Sac à Dos 0/1 avec Poids Minimum

Question 7: Conception d'un Algorithme de Programmation Dynamique pour le Sac à Dos avec Poids Minimum

Valeur à stocker dans *dp[i][w]* :

La table dp[i][w] doit stocker la valeur maximale atteignable en utilisant les i premiers objets avec un poids total w. Cette table permet de suivre la meilleure combinaison d'objets pour chaque sous-problème défini par un sous-ensemble d'objets et une capacité de poids.

- **Définition récursive :**

  La définition récursive pour dp[i][w] est :

  dp[i][w] = max(dp[i−1][w], dp[i−1][w−pi] + vi)

  où vi et pi sont respectivement la valeur et le poids de l'objet i. Cela signifie que pour chaque objet i et chaque poids w, nous avons deux choix :

  - Ne pas inclure l'objet i, auquel cas la meilleure valeur atteignable reste dp[i−1][w].
  - Inclure l'objet i, auquel cas nous devons ajouter sa valeur vi à la meilleure valeur atteignable avec le poids w−pi.
- **Implémentation en C :**

  Nous avons écrit un programme en C pour implémenter cette solution de programmation dynamique. Le programme initialise la table dp avec des valeurs de départ conformes au cas de bas, puis la remplit en utilisant la relation récursive décrite ci-dessus. Le script generateData.py a été adapté pour générer des instances de ce problème, et les algorithmes ont été testés pour analyser leur performance.

```c
int knapsackDPMin(Item items[], int n, int C, int M)
{
    int **dp = (int **)malloc((n + 1) * sizeof(int *));
    for (int i = 0; i <= n; i++)
    {
        dp[i] = (int *)malloc((C + 1) * sizeof(int));
        for (int w = 0; w <= C; w++)
        {
            dp[i][w] = 0;
        }
    }

    for (int i = 1; i <= n; i++)
    {
        for (int w = 0; w <= C; w++)
        {
            if (items[i - 1].weight <= w)
            {
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - items[i - 1].weight] + items[i - 1].value);
            }
            else
            {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    int maxValue = 0;
    for (int w = M; w <= C; w++)
    {
        if (dp[n][w] > maxValue)
        {
            maxValue = dp[n][w];
        }
    }

    for (int i = 0; i <= n; i++)
    {
        free(dp[i]);
    }
    free(dp);

    return maxValue;
}
```

- **Complexité temporelle et spatiale :**

  La complexité temporelle de cette solution est O(n(C−M)), où C est la capacité maximale et M est le poids minimum requis. La complexité spatiale est également O(n(C−M)), car nous devons stocker des valeurs pour chaque combinaison d'objets et de poids entre M et C.

Exercice 5 : Approximation par un algorithme glouton

Question 8: Implémentation de l'algorithme glouton

- **Implémentation :**

  L'algorithme glouton pour le problème du sac à dos 0/1 trie les objets en fonction de leur rapport valeur/poids (valeur par unité de poids) et sélectionne les objets dans cet ordre jusqu'à ce que la capacité du sac soit atteinte. L'idée est de maximiser la valeur totale du sac à dos en ajoutant les objets avec le meilleur rapport valeur/poids en premier.

```c
int knapsack_glouton(Item items[], int n, int W)
{
    qsort(items, n, sizeof(Item), comparer_ratiobj);

    int poids_total = 0;   // Poids total des objets sélectionnés
    int valeur_totale = 0; // Valeur totale des objets sélectionnés

    for (int i = 0; i < n; i++)
    {
        if (poids_total + items[i].weight <= W)
        {
            poids_total += items[i].weight;
            valeur_totale += items[i].value;
        }
    }

    return valeur_totale;
}
```

- **Complexité :**

  La complexité temporelle de cet algorithme est O(nlogn) en raison de l'étape de tri des objets par leur rapport valeur/poids. La complexité spatiale est O(n), car nous devons stocker les objets triés et la solution finale.

- **Cas d'optimalité non atteinte :**

Nous avons identifié des instances où l'algorithme glouton n'atteint pas la solution optimale. Par exemple, considérons trois objets avec les caractéristiques suivantes :

- Objet 1 : poids = 5, valeur = 10 (rapport = 2)
- Objet 2 : poids = 4, valeur = 7 (rapport = 1.75)
- Objet 3 : poids = 6, valeur = 12 (rapport = 2)

Si la capacité du sac est 10, l'algorithme glouton choisira d'abord l'Objet 1, puis l'Objet 3, atteignant un poids total de 11, ce qui dépasse la capacité. La solution optimale serait de choisir l'Objet 1 et l'Objet 2 pour un poids total de 9 et une valeur totale de 17. L'algorithme glouton ne parvient pas à identifier cette combinaison optimale, bien qu’il soit très rapide et efficace.

Partie 2 : Rapport Général du Projet

Introduction :

Ce projet a pour objectif d'analyser et de comparer diverses méthodes algorithmiques pour résoudre le problème classique du sac à dos (Knapsack Problem). Nous avons implémenté et testé plusieurs algorithmes, notamment des techniques de backtracking, de programmation dynamique et une approche gloutonne. L'objectif principal est de déterminer l'efficacité et la performance de chaque méthode en termes de complexité temporelle et spatiale. Les algorithmes ont été évalués à l'aide de données générées automatiquement et leurs performances ont été comparées empiriquement à l'aide des outils fournis.

Description des Algorithmes

- **Backtracking :**
  - **Backtracking 1 :** Utilise des variables globales pour conserver l'état et simplifier la gestion des appels récursifs. Cela peut rendre le code moins flexible et plus difficile à comprendre.
  - **Backtracking 2 :** Utilise des variables locales et des paramètres de fonction pour une approche plus modulaire. Cette version évite les effets secondaires associés aux variables globales, ce qui rend le code plus propre et plus maintenable.
  - **Backtracking 3 :** Introduit des techniques d'élagage pour réduire le nombre de combinaisons explorées. En utilisant une borne supérieure calculée à partir de la somme des valeurs des objets restants, cette version peut éliminer plus tôt les branches non prometteuses de l'arbre de recherche.

- **Programmation Dynamique :**
  - **Programmation Dynamique par Capacité :** Cet algorithme remplit une table basée sur les poids des objets pour trouver la solution optimale. La table dp[i][w] stocke la valeur maximale atteignable en utilisant les i premiers objets avec un poids total w.
  - **Programmation Dynamique par Valeur :** Construit une table en fonction des valeurs des objets, minimisant le poids pour atteindre une

valeur donnée. La table dp[v] représente le poids minimal nécessaire pour obtenir une valeur v.

❖ **Algorithme Glouton**

Cet algorithme trie les objets en fonction de leur rapport valeur/poids (valeur par unité de poids) et sélectionne les objets dans cet ordre jusqu'à ce que la capacité maximale soit atteinte. Bien que rapide, cette méthode ne garantit pas toujours une solution optimale, notamment lorsque des objets avec un bon rapport valeur/poids excluent des combinaisons plus avantageuses.

Analyse des Performances

Complexité Temporelle et Spatiale

- **Backtracking :**
    - **Complexité Temporelle :** Les algorithmes de backtracking ont une complexité temporelle de l'ordre de $O(2^n)$, où n est le nombre d'objets. Cela s'explique par l'exploration exhaustive de toutes les combinaisons possibles d'objets pour trouver la solution optimale.
    - **Complexité Spatiale :** La complexité spatiale est $O(n)$ en raison de la profondeur de la pile de récursion. Chaque appel récursif ajoute un cadre à la pile, ce qui occupe de l'espace mémoire proportionnel au nombre d'objets.

- **Programmation Dynamique :**
    - **Complexité Temporelle :**
        - ***Par Capacité :*** $O(nC)$, où C est la capacité maximale du sac.
        - ***Par Valeur :*** $O(nV)$, où V est la somme des valeurs des objets.
    - **Complexité Spatiale :**
        - ***Par Capacité :*** $O(nC)$pour stocker la table des valeurs.
        - ***Par Valeur :*** $O(V)$ pour stocker les poids minimaux nécessaires pour chaque valeur possible.
- **Algorithme Glouton :**
    - **Complexité Temporelle :** $O(n\log n)$ en raison du tri initial des objets par leur rapport valeur/poids.
    - **Complexité Spatiale :** $O(n)$, car il nécessite de stocker les objets triés et la solution finale.

Résultats Expérimentaux

Les résultats expérimentaux montrent que :

- **Backtracking :**
    - Les algorithmes de backtracking marchent pour des petites tailles de problèmes (jusqu'à 20 objets), mais deviennent impraticables pour des instances plus grandes en raison de la croissance exponentielle du nombre de combinaisons à explorer.
    - L'ajout de techniques d'élagage (Backtracking 3) améliore significativement les performances en réduisant le nombre de branches explorées.

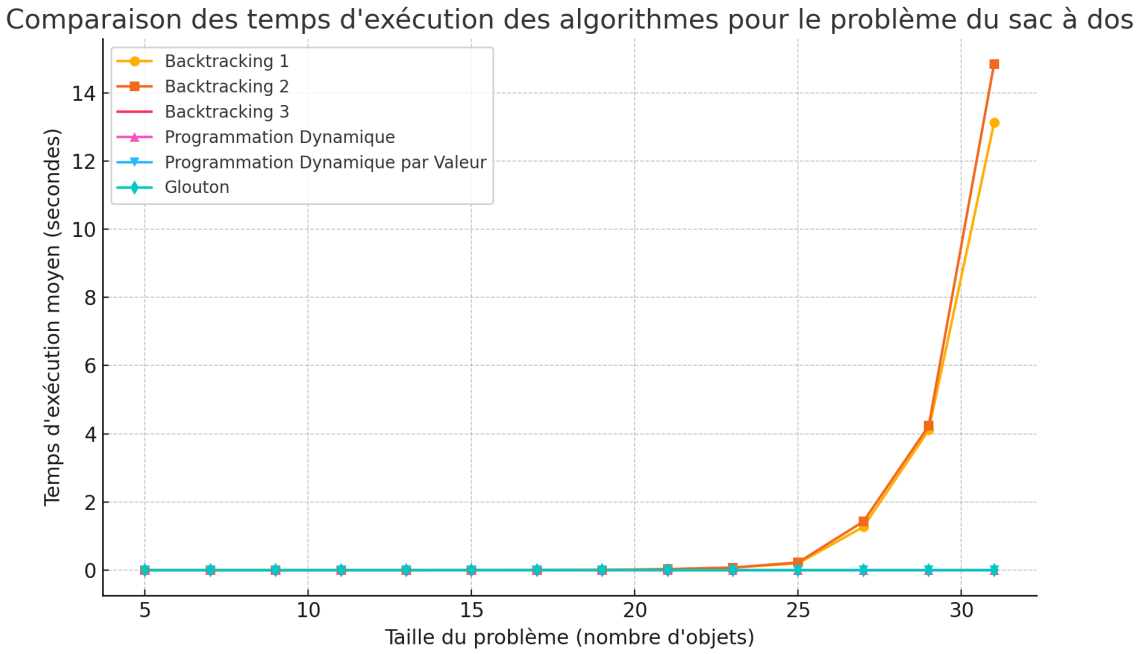
- **Programmation Dynamique :**
    - Ces algorithmes sont plus performants pour des problèmes de taille moyenne à grande. Ils offrent un compromis raisonnable entre temps de calcul et utilisation de la mémoire.
    - La version basée sur la capacité est particulièrement efficace lorsque la capacité du sac est relativement petite par rapport aux valeurs des objets.
    - La version basée sur la valeur est plus adaptée lorsque les valeurs des objets varient beaucoup et que la capacité n'est pas le facteur limitant.

- **Algorithme Glouton :**
    - Cet algorithme est extrêmement rapide et peut fournir des solutions approximatives en temps linéaire après le tri initial.
    - Cependant, il ne garantit pas toujours une solution optimale, surtout dans les cas où le choix d'objets basé uniquement sur le rapport valeur/poids peut exclure des combinaisons plus avantageuses.

Comparaison des Temps d'Exécution

Le graphique fourni illustre la comparaison des temps d'exécution moyens pour différents algorithmes en fonction de la taille du problème (nombre d'objets). Les algorithmes de backtracking montrent une croissance exponentielle du temps

d'exécution, tandis que les algorithmes de programmation dynamique et l'algorithme glouton restent beaucoup plus efficaces pour des tailles de problème plus grandes.



Conclusion

Ce projet a permis de mettre en évidence les forces et les faiblesses de chaque approche algorithmique pour le problème du sac à dos. Les algorithmes de backtracking, bien qu'optimaux pour les petites instances, sont limités par leur complexité exponentielle. Les algorithmes de programmation dynamique offrent une meilleure efficacité pour des problèmes de taille moyenne à grande, en fournissant des solutions optimales avec une complexité plus raisonnable. L'algorithme glouton, malgré ses limitations en termes d'optimalité, fournit des solutions rapides et souvent satisfaisantes, ce qui peut être utile pour des approximations rapides lorsque le temps de calcul est critique.