|  |  |  |
| --- | --- | --- |
| Numérique et Sciences Informatiques | | |
| 3h | **Algorithmiques gloutons** |  |
| **Objectif** : résoudre un problème grâce à un algorithme glouton | | |
| **Matériel**: papier, Python | | |

**Problème de livraisons**

Les livreurs d’un magasin partent de celui-ci, avec des produits à livrer à plusieurs clients, avant de revenir au magasin.

Par exemple, voici comment sont localisés différents clients à livrer par rapport au magasin :



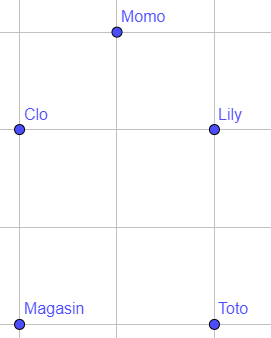
* D'après vous quel est le meilleur (le plus court) trajet de livraison ?

Mag 🡪 Toto 🡪 Lily 🡪 Clo

* D’après vous, quelle règle permet de choisir la prochaine destination après chaque client livré ?

La personne la plus proche

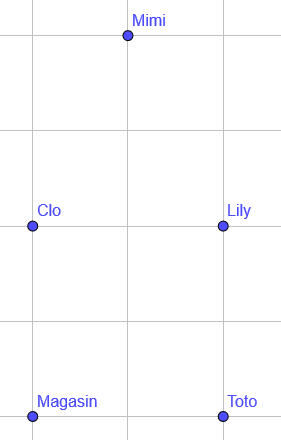
* Appliquez la règle définie à la situation suivante :



* Le trajet vous-semble-t-il optimum (le meilleur possible) ?

Mag 🡪 Toto 🡪 Lily 🡪 Momo 🡪 Clo

* Appliquez la règle définie à la situation suivante :



* Le trajet vous-semble-t-il optimum (le meilleur possible) ?

Mag 🡪 Toto 🡪 Lily 🡪 Clo 🡪 Mimi

Non

**Ce problème est un problème d’optimisation qui consiste à trouver le meilleur chemin.**

On parle d’optimisation lorsqu’on cherche à obtenir une solution qui se rapproche de la meilleure des solutions → **meilleure solution = optimal**.

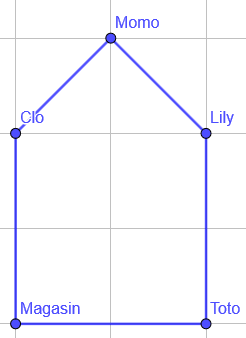
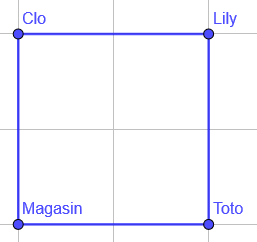
Résoudre ce problème est compliqué car il y a beaucoup de cas à comparer si on veut obtenir le meilleur résultat (optimal), ça peut prendre du temps.

Néanmoins, on s’aperçoit qu’on peut **avancer étape par étape**, en choisissant un **optimum local** (même si le dernier cas ne nous donne pas une solution optimale).

On obtient alors **rapidement** une **solution globale "souvent" proche de l’optimum global** (ce n’est pas toujours le cas).

Les règle pour trouver l’optimum vont être traduites dans un **algorithme glouton** (**greedy algorithm = algorithme cupide**).

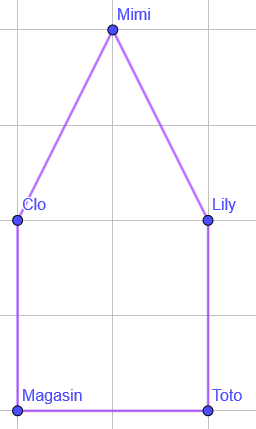
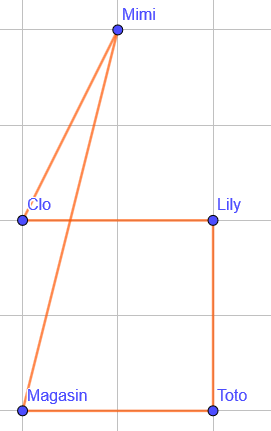
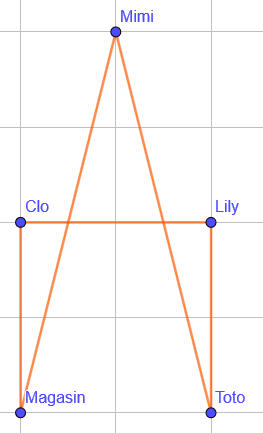
**En résumé :**



**Glouton → Optimal**

**Glouton → pas Optimal**

**Optimal sans Glouton**



# Problème du rendu de monnaie

* Vous achetez un paquet de bonbons valant 1,02€ avec un billet de 5€, combien doit-on vous rendre ?

3.98

* Remplissez le tableau, en appliquant les **règles** suivantes :
* *La somme rendue doit correspondre exactement à la somme due.*
* *Le nombre de pièces utilisées doit être le plus petit possible.*

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **2€** | **1€** | **50c€** | **20c€** | **10c€** | **5c€** | **2c€** | **1c€** | **Client satisfait** | **Caissier satisfait** | **Nombre de pièces** |
| **Cas 0** | 1 | 1 | 1 | 2 | 0 | 1 | 1 | 1 | Oui | Oui | 8 |
| **Cas 1** | 1 | 1 | 1 | 2 | 0 | 0 | 4 |  | Oui | Oui | 9 |
| **Cas 2** | 1 | 1 | 1 | 2 | 0 | 1 |  | 3 | Oui | Oui | 9 |
| **Cas 3** | 1 | 1 | 1 | 2 | 0 |  | 4 | 0 | Oui | Oui | 9 |
| **Cas 4** | 1 | 1 | 1 | 2 |  | 1 | 1 | 1 | Oui | Oui | 8 |
| **Cas 5** | 1 | 1 | 1 |  | 4 | 1 | 1 | 1 | Oui | Oui | 10 |
| **Cas 6** | 1 | 1 | 1 | 2 | 0 | 0 | 0 | 0 | non | oui | 5 |
| **Cas 7** | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | oui | non | 2 |
| **Cas 8** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 398 | non | oui | 398 |

* Quel est le cas le plus satisfaisant d’après les règles ?

Le cas 0 (donc 4)

**Rappel** : en Python, les calculs sur des flottants peuvent conduire à des erreurs (0.1 + 0.2 ≠ 0.3).

On va donc écrire la liste des valeurs des pièces en centimes d'€uros :

L\_valeurs = [200, 100, 50, 20, 10, 5, 2, 1] : centimes d'€uros

~~L\_valeurs = [2, 1, 0.5, 0.2, 0.1, 0.05, 0.02, 0.01] : €uros~~

Code\*\* : Écrivez en **pseudo-code** la fonction rendu\_monnaie() prenant en paramètres la somme à rendre et la liste des valeurs des différentes pièces disponnibles L\_valeurs.

La fonction doit renvoyer la liste L\_nb\_pieces permettant d'égaler (ou presque) somme en utilisant L\_valeurs et en minimisant le nombre total de pièces utilisées.

Exemples :

|  |  |  |  |
| --- | --- | --- | --- |
| **Entrées** | | **Sorties** |  |
| somme | L\_valeurs | L\_nb\_pieces | **Nombre de pièces** |
| 398 | [200, 100, 50, 20, 10, 5, 2, 1] | [1, 1, 1, 2, 0, 1, 1, 1] | 8 |
| 400 | [200, 100, 50, 20, 10, 5, 2, 1] | [2, 0, 0, 0, 0, 0, 0, 0] | 2 |

L’algorithme de la fonction rendu\_monnaie() sera de type **glouton** car :

* La somme rendue sera très proche de la somme exacte (mais pas forcément exacte).
* Le nombre de pièces utilisées sera proche de l’optimal (le moins de pièces possible).
* La fonction avance étape par étape, en choisissant un optimum local.
* Le calcul de la somme à rendre et du nombre de pièce sera rapide.

|  |
| --- |
| **Pseudo-code :** |
|  |

* Ecrivez en **Python** la fonction **glouton** rendu\_monnaie() puis testez-la sur les exemples donnés.

Pour chacun des 2 exemples, la fonction donne-t-elle un résultat optimal (le meilleur) ?

**Rappel sur le Cas 1 (rendre 398 sans pièce de 1c€) :**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **2€** | **1€** | **50c€** | **20c€** | **10c€** | **5c€** | **2c€** | **1c€** | **Client satisfait** | **Caissier satisfait** | **Nombre de pièces** |
| **Cas 1** | 1 | 1 | 1 | 2 | 0 | 0 | 4 |  | oui | oui | 9 |

**→ on s’attend donc à ce que la fonction rendu\_monnaie() donne L\_nb\_pieces=[1,1,1,2,0,0,4]**

* Testez la fonction **glouton** rendu\_monnaie() sur le **Cas 1**.

La fonction donne-t-elle un résultat optimal (le meilleur) ?

Non

* Testez la fonction **glouton** rendu\_monnaie() sur les exemples suivants.

Dans quel cas le résultat n’est pas optimal (le meilleur) ?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Entrées** | | **Sorties** |  |  |
| Somme | L\_valeurs | L\_nb\_pieces | **Somme rendue** | **Nombre de pièces** |
| 390 | [200, 100, 50, 20, 10, 5, 2, 1] | [1, 1, 1, 2, 0, 0, 0, 0] | 390 | 5 |
| 390 | [200, 100, 50, **25**, 20, 10, 5, 2, 1] | [1, 1, 1, 1, 0, 1, 1, 0, 0] | 390 | 6 |

**Si on veut être certain d’obtenir le rendu exact de la somme due, on peut utiliser une méthode de recherche exhaustive dite de force brute qui compare toutes les possibilités.**

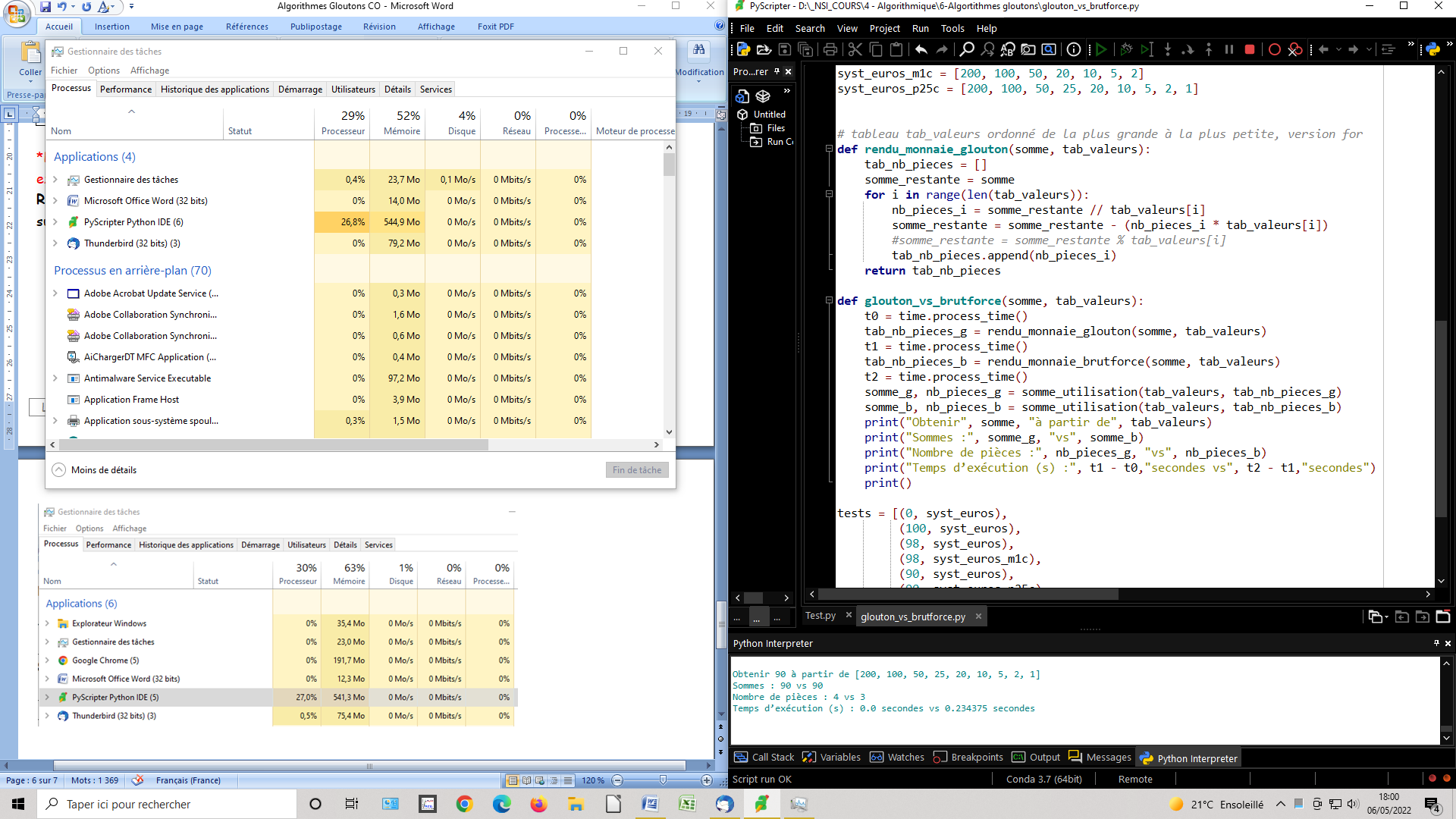
* Exécutez le fichier **glouton\_vs\_brutforce.py** pour comparer un **algorithme glouton** de rendu de monnaie avec un **algorithme en force brute** puis complétez le tableau suivant :

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Somme cible** | **L\_valeurs** | **Somme**  **glouton** | **Somme**  **brutforce** | **Nombre de pièces**  **glouton** | **Nombre de pièces**  **brutforce** | **Temps**  **glouton** | **Temps**  **brutforce** |
| 0 | [200, 100, 50, 20, 10, 5, 2, 1] |  |  |  |  |  |  |
| 100 | [200, 100, 50, 20, 10, 5, 2, 1] |  |  |  |  |  |  |
| 98 | [200, 100, 50, 20, 10, 5, 2, 1] |  |  |  |  |  |  |
| 98 | [200, 100, 50, 20, 10, 5, 2] |  |  |  |  |  |  |
| 90 | [200, 100, 50, 20, 10, 5, 2, 1] |  |  |  |  |  |  |
| 90 | [200, 100, 50, **25**, 20, 10, 5, 2, 1] |  |  |  |  |  |  |
| 200 | [200, 100, 50, 20, 10, 5, 2, 1] |  |  |  |  |  |  |
| 198 | [200, 100, 50, 20, 10, 5, 2, 1] |  |  |  |  |  |  |
| 198 | [200, 100, 50, 20, 10, 5, 2] |  |  |  |  |  |  |
| 190 | [200, 100, 50, 20, 10, 5, 2, 1] |  |  |  |  |  |  |
| 190 | [200, 100, 50, 20, 10, 5, 2, 1] |  |  |  |  |  |  |

* Remplissez le tableau de comparaison entre l’algorithme **glouton** et l’algorithme de **force brute** :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Rendu toujours juste ?** | **Résultat proche d’une solution optimale ?** | **Temps d’exécution** | **Coût en mémoire\*** |
| **Glouton** |  |  |  |  |
| **Brutforce** |  |  |  |  |

\*Pour évaluer le **coût en mémoire**, lancez le **programme glouton** puis celui **brutforce** et pour chaque exécution regardez dans le gestionnaire de tâches.



* Concluez sur l'algorithme (**glouton** ou **force brute**) qui vous semble être le meilleur compromis pour réaliser un rendu de monnaie.

Remarque : l’algorithme glouton de rendu de monnaie donne toujours le meilleur résultat (optimal) sur notre système monétaire car l'euro est un système canonique [2€, 1€, 50c, 20c, 10c, 5c, 2c, 1c].