4-karatsuba-sujet

November 2, 2022

1 Arithmétique de grands entiers

Dans ce TP : - on représente des *grands entiers*, c-a-d. des entiers qui ne sont pas des `int` de Python. - On code différents algorithmes d'arithmétique de tels entiers, en particulier la multiplication de Karatsuba étudié en cours. - Puis on effectue l'analyse expérimentale des performances de plusieurs versions de ces algorithmes afin d'optimiser leur performance. - On considère aussi des versions optimisées de la bibliothèques spécialisée GMP (`gmpy2` pour python).

Les questions de ce TP : - précisent la signature des fonctions à coder - sont (presque toutes) suivies de cellules de validation des traitements demandées.

Conseil. Lire *tout* le sujet au moins une fois avant de commencer.

Des "grands entiers" en Python ? Si une valeur entière n'est pas exactement représentable par un `int` python, le langage "bascule" automatiquement vers les entiers de précision arbitraire fournis par la bibliothèque GMP. Cette remarque sera utile pour écrire facilement des tests unitaires de nos traitements.

Exemple.

[ ]:
```
for i in range(60,70):
    x = 2 ** i
    print(i, x, type(x))
```

1.1 imports

On rassemble dans la cellule suivante **tous** les `import` dont on aura besoin au fur et à mesure de nos réponses. Ce qui permet de la relancer facilement pour d'éventuels redémarrage de noyau et exécution partielle du notebook.

[ ]:
```
from random import randint
import time
import numpy as np
import matplotlib.pyplot as plt
from numpy.polynomial.polynomial import polyfit
from math import floor, log, sqrt
from gmpy2 import mpz, mul
```

1.2 Représentation de grands entiers positifs

On représente un entier **positif** par la liste de ces chiffres (en base 10) stockés dans une liste par ordre croissant de son exposant en notation de position : le chiffre de plus faible exposant est le plus à droite de cette représentation .

Ainsi $123 = 1 \times 10^2 + 2 \times 10 + 3$ sera représenté par la liste `[1, 2, 3]`.

On va écrire les fonctions qui permettent les conversions entre ces "grands entiers", les `int` ou les `str` de Python.

1.2.1 `inttointlist0()`

Ecrire la fonction `inttointlist0()` qui convertit un `int`en un "grand entier".

[ ]:
```
def inttointlist0(n : int) -> list[int]:
    pass
```

[ ]:
```
assert inttointlist0(0) == [0]
assert inttointlist0(4) == [4]
assert inttointlist0(84) == [8, 4]
assert inttointlist0(123) == [1, 2, 3]
```

1.2.2 `intlisttoint0`

Ecrire la fonction `intlisttoint0()` qui convertit un "grand entier" en un `int`.

[ ]:
```
def intlisttoint0(l : list[int]) -> int:
    pass
```

[ ]:
```
assert intlisttoint0([0]) == 0
assert intlisttoint0([8, 9]) == 89
assert intlisttoint0([1, 2, 3]) == 123
```

1.2.3 `strtointlist()`

Ecrire la fonction `strtointlist()` qui convertit une `str` de Python en un "grand entier". La validation qui suit précise le format attendu de la `str`.

[ ]:
```
def strtointlist(s : str) -> list[int]:
    pass
```

[ ]:
```
assert strtointlist("0") == [0]
assert strtointlist("123") == [1, 2, 3]
```

1.2.4 `intlisttostr()`

Ecrire la fonction `intlisttostr()` qui convertit un "grand entier" en une `str` de Python.

[ ]:
```
def intlisttostr(l : list[int]) -> str:
    pass
```

[ ]:
```
assert intlisttostr([0]) == "0"
assert intlisttostr([1, 2, 3]) == "123"
```

1.3 `add0()`: l'addition simple de grands entiers positifs

Ecrire la fonction `add0()` qui effectue l'addition de 2 grands entiers **positifs**.

La gestion de la retenue est laissée à votre choix.

[ ]:
```
def add0(u: list[int], v: list[int]) -> list[int]:
    pass
```

[ ]:
```
assert add0([2], [1]) == [3]
assert add0([1], [2]) == [3]
assert add0([1], [2, 0]) == [2, 1]
assert add0([8], [3]) == [1, 1]
assert add0([8, 9], [1]) == [9, 0]
assert add0([9, 9], [2]) == [1, 0, 1]
assert add0([], [3]) == [3]
assert add0([0], [3]) == [3]
```

1.4 Quid des négatifs ?

Avant d'aller plus loin, comment représenter des grands entiers négatifs de façon cohérente avec la représentation choisie et dont dépendent les fonctions développées jusque-là, et en particulier l'addition `add0`?

On va donc convenir de stocker un grand entier négatif en notation "signe-entier". Ainsi le signe négatif est un `-1` **supplémentaire** situé en début de la représentation, c-a-d. comme chiffre de plus haut poids dans la représentation adoptée.

Ainsi l'entier négatif `-12345` sera représenté par la liste `[-1, 1, 2, 3, 4, 5]`.

Remarque importante. Précisons comment traiter la valeur nulle. - le "grand" entier 0 **est présenté par `[0]` ou `[]`** - la représentation de 0 par la liste vide `[]` est commode pour les traitements à venir. - il n'existe **pas** de zéro négatif, *ie.* il n'existe pas de `[-1, 0]`

On va étendre l'addition `add0()` pour traiter de tels opérandes négatifs.

Pour cela, on doit définir les fonctions suivantes : - `signe()` qui retourne le signe d'un grand entier - `neg()` qui retourne l'opposé d'un grand entier - `ge()` (pour *greater or equal* ou `=<`) qui compare 2 grands entiers positifs

On doit aussi compléter les fonctions de conversions pour traiter maintenant les “grands entiers” négatifs : - `inttointlist0()` est complétée en `inttointlist()`
- `intlisttoint0()` est complétée en `intlisttoint()`

`signe()`

[ ]:
```
def signe(u: list[int]) -> int:
    pass
```

[ ]:
```
assert signe([1, 2, 3]) == 0
assert signe([-1, 1]) == -1
assert signe([-1, 1, 2, 3]) == -1
assert signe([]) == 0
assert signe([0]) == 0
```

`neg()`

[ ]:
```
def neg(u: list[int]) -> list[int]:
    pass
```

[ ]:
```
assert neg([1, 2, 3]) == [-1, 1, 2, 3]
assert neg([-1, 1, 2, 3]) == [1, 2, 3]
assert neg([0]) == [0]
assert neg([]) == [0]
```

`ge()` Bien sûr, les deux représentations de 0 sont égales.

[ ]:
```
def ge(u: list[int], v: list[int]) -> bool:
    pass
```

[ ]:
```
# u ou v == 0
assert ge([0], [0]) == True
assert ge([], [0]) == True
assert ge([0], []) == True
assert ge([0], [1]) == False
assert ge([1], [0]) == True

# u > 0 et v > 0
assert ge([1,1], [3]) == True
assert ge([1,1], [1, 0]) == True
assert ge([1,1], [1, 1]) == True
assert ge([1,1], [3, 4]) == False
assert ge([1,1], [3, 4, 1]) == False
assert ge([2], [1]) == True
assert ge([2], [2]) == True
assert ge([2], [3]) == False
```

```
# u > 0 et v < 0
assert ge([1,1], [-1, 3]) == True
assert ge([1,1], [-1, 3, 4]) == True
assert ge([1,1,1], [-1, 1]) == True

# u < 0 et v > 0
assert ge([-1,1], [3]) == False
assert ge([-1,1], [1, 3, 4]) == False
assert ge([-1,1,1], [1, 1]) == False

# u > 0 et v > 0
assert ge([-1,1], [-1,3]) == True
assert ge([-1,1,1], [-1, 1, 0]) == False
assert ge([-1,1], [-1, 1]) == True
assert ge([-1,1], [-1, 3, 4]) == True
assert ge([-1,1,1], [-1, 3]) == False
assert ge([-1, 2], [-1, 1]) == False
assert ge([-1, 2], [-1, 2]) == True
assert ge([-1, 2], [-1, 3]) == True
```

1.4.1 inttointlist()

[ ]:
```
def inttointlist(n : int) -> list[int]:
    pass
```

[ ]:
```
assert inttointlist(0) == [0]
assert inttointlist(-4) == [-1, 4]
assert inttointlist(84) == [8, 4]
assert inttointlist(-123) == [-1,1, 2, 3]
```

1.4.2 intlisttoint()

[ ]:
```
def intlisttoint(l : list[int]) -> int:
    pass
```

[ ]:
```
assert intlisttoint([]) == 0
assert intlisttoint([0]) == 0
assert intlisttoint([8, 9]) == 89
assert intlisttoint([1, 2, 3]) == 123
assert intlisttoint([-1, 8, 9]) == -89
assert intlisttoint([-1, 1, 2, 3]) == -123
```

1.4.3 `add()` : addition de 2 grands entiers de signe quelconque

On peut maintenant écrire l'addition `add()` de 2 "grands entiers" de signe quelconque.

[ ]:
```python
def add(u: list[int], v: list[int]) -> list[int]:
    pass
```

[ ]:
```python
assert add([2], [1]) == [3]
assert add([1], [2]) == [3]
assert add([1], [2, 0]) == [2, 1]
assert add([8], [3]) == [1, 1]
assert add([8, 9], [1]) == [9, 0]
assert add([9, 9], [2]) == [1, 0, 1]

assert add([2], [-1, 1]) == [1]
assert add([1], [-1, 2]) == [-1, 1]
assert add([-1, 1], [2, 0]) == [1, 9]
assert add([8], [-1, 3]) == [5]
assert add([-1, 8, 9], [-1, 1]) == [-1, 9, 0]
assert add([-1, 9, 9], [-1, 2]) == [-1, 1, 0, 1]
assert add([1,2], [-1,5]) == [7]
assert add([5, 4], [-1, 4, 8]) == [6]

assert add([8], [-1, 8]) == [0]
assert add([-1, 8], [8]) == [0]
assert add([-1, 8], []) == [-1, 8]
```

1.5 Multiplication de grands entiers : méthode naïve

La multiplication naïve (celle "de la petite école" ou par multiplications-décalages) consiste à des multiplications d'un opérande par un nombre à un seul chiffre extrait de l'autre opérande, à des décalages des produits partiels ainsi obtenus et à leur addition finale.

La multiplication d'un grand entier par un nombre à un chiffre peut introduire des retenues à propager dans la représentation choisie : ici une liste de chiffres. On commence par la propagation de retenues présentes dans une liste d'entiers arbitraires de façon à obtenir la représentation d'un grand entier.

Remarque importante pour toute la suite du sujet.

- **La multiplication de grands entiers se limitera des opérandes positifs**
- Dans ce qui suit, on ne considérera pas le cas de la multiplication de grands entiers de signe quelconque.
  - Bien sûr en pratique, il n'est pas difficile d'étendre les développements qui vont venir aux cas de grands entiers de signe quelconque (par un pré-traitement du signe).

1.5.1 `propagerretenue()`

Cette fonction propage la retenue du "grand entier" `u` de façon conforme avec la représentation choisie. **Les exemples de validation explicitent le traitement attendu :** on suppose qu'on a obtenu une forme temporaire de la représentation d'un "grand entier". Cette forme temporaire comporte des **nombres** pas nécessairement inférieurs à 10. Cette fonction permet de générer l'écriture finale de ce "grand entier" en propageant les retenues nécessaires. Cette fonction facilitera l'écriture à venir des multiplications.

[ ]:
```python
def propagerretenue(u: list[int]) -> list[int]:
    pass
```

[ ]:
```python
assert propagerretenue([1, 2, 3]) == [1, 2, 3]
assert propagerretenue([11, 2, 33]) == [1, 1, 5, 3]
assert propagerretenue([11, 9, 33]) == [1, 2, 2, 3]
```

1.5.2 `multchiffre()`

On continue par la multiplication d'un grand entier par un nombre à un chiffre, propagation des retenues inclue sur le résultat.

[ ]:
```python
def multchiffre(u: list[int], c: int) -> list[int]:
    pass
```

[ ]:
```python
assert multchiffre([3, 2, 1], 0) == [0]
assert multchiffre([3, 2, 1], 1) == [3, 2, 1]
assert multchiffre([3, 2, 1], 4) == [1, 2, 8, 4]
assert multchiffre([3, 2, 1], 9) == [2, 8, 8, 9]
```

1.5.3 Décalage

On complète avec le décalage d'un grand entier (pour les résultats des multiplications partielles avant sommation).

1.5.4 `decaler()`

Cette fonction décalle le grand entier `u` de `p` positions vers la gauche – ce qui est utile pour la multiplication naïve.

Formellement cette fonction calcule $u \times 10^p$.

[ ]:
```python
def decaler(u: list[int], p: int) -> list[int]:
    pass
```

[ ]:
```python
assert decaler([1], 2) == [1, 0, 0]
assert decaler([1], 3) == [1, 0, 0, 0]
```

```
assert decaler([9, 9], 0) == [9, 9]
assert decaler([9, 9], 1) == [9, 9, 0]
```

1.5.5 `mult()`

On peut maintenant coder la multiplication naïve de deux grands entiers.

[ ]:
```
def mult(u: list[int], v: list[int]) -> list[int]:
    pass
```

[ ]:
```
assert mult([3, 2, 1], [0]) == [0]
assert mult([3, 2, 1], [1]) == [3, 2, 1]
assert mult([3, 2, 1], [1, 0]) == [3, 2, 1, 0]
```

1.6 Vérification

Pour des entiers aléatoires, on compare le résultat de `mult()` et des fonctions de conversions de format avec les résultats de l'opérateur natif de python.

[ ]:
```
pass
```

1.7 Complexité

Quelle est la complexité de cette multiplication ? Donner une expression précise puis son comportement asymptotique.

[ ]:
```

```

Coder cette complexité pour un ensemble de valeurs du paramètre de complexité `n`.

[ ]:
```
def comp_theorique(vect_n: list[int]) -> list[int]:
    pass
```

1.8 Timings

On va mesurer l'efficacité en temps de calcul de plusieurs algorithmes d'arithmétique "sur ces grands entiers". Ces mesures dépendent de la taille des opérandes, c-a-d. du nombre de chiffres décimaux de leur représentation. Nous allons donc effectuer plusieurs groupes de cas tests.

Définir de bonnes plages de test Les opérandes à tester seront choisies aléatoirement et dans un nombre à définir selon les 3 groupes suivants.

1. les opérandes avec des dizaines de décimales : entre 10 et 99 décimales
2. les opérandes avec des centaines de décimales : entre 100 et 999 décimales

3. les opérandes avec des milliers de décimales : entre 1000 et 9999 décimales

Bien sûr, ces intervalles seront parcourus de façon adaptée. Dans le cas 1 (dizaines), il est naturel de tester chaque dizaine. Mais ce choix n'est pas pertinent en pratique pour le cas 2 (centaines) et sans aucun intérêt pour le cas 3 (milliers).

Des opérandes aléatoires d'une taille fixée Ecrire un traitement simple qui génère des opérandes aléatoires d'un nombre de chiffres fixé. Ce traitement permettra de générer des échantillons d'opérandes pour chaque groupe de test.

```
[ ]: pass
```

Plusieurs dizaines de décimales : mesures et tracés Pour ce premier groupe de test, effectuer les mesures de performances de la multiplication naïve de deux grands entiers `mult()`. Auparavant, préciser rapidement votre stratégie de mesure.

Stratégie.

Mesures.

```
[ ]: pass
```

Tracés.

Proposer des tracés pertinents de ce premier groupe de mesures et de la complexité théorique.

```
[ ]: pass
```

Plusieurs centaines de décimales : mesures et tracés Effectuer un traitement (mesures et tracés) similaire à celui de la question précédente pour ce deuxième groupe.

Conclusion On gardera le troisième groupe (milliers de décimales) pour des algorithmes plus rapides.

Conclure brièvement sur ces premières mesures.

1.9 Algorithme de Karatsuba

On s'intéresse maintenant au produit de deux "grands entiers" avec l'algorithme de Karatsuba étudié en cours.

1.9.1 Introduction avec des opérandes `int`

On se "fait la main" sur cet algorithme en commençant avec des opérandes "petits", c-a-d. des `int` Python.

Coder cette multiplication.

[ ]:
```
def multK_int(u: int, v: int) -> int:
    pass
```

[ ]:
```
assert multK_int(123, 567) == 123 * 567
assert multK_int(1234, 567) == 1234 * 567
assert multK_int(12345, 567) == 12345 * 567
assert multK_int(12345, 4567) == 12345 * 4567

assert (12345678 * 56789012) == multK_int(12345678, 56789012)
```

1.9.2 Les opérandes sont des grands entiers, *ie.* des `list[int]`

On considère maintenant des opérandes "grands entiers" représentés comme précédemment, c-a-d. par une liste des chiffres en base dix.
On effectue la multiplication avec cette représentation : il ne s'agit pas de convertir ces "grands entiers" en des `int` Python (avec `intlisttoint()`).

Rappel. Les grands entiers sont stockés dans l'ordre de leur écriture en notation de position : 123 -> `[1, 2, 3]`.

Préalable : tranches de listes et duplication Bien regarder comment se dupliquent des listes ou des `ndarray` (types mutables) et comment se manipulent les tranches (*slicing*) de ce type d'objets.

[ ]:
```
pass
```

La solution de Karatsuba est récursive. On commence avec une récursion presque maximale, c-a-d. un arrêt de la récursion de Karastuba lorsqu'un des deux opérandes a 2 chiffres ou moins. Dans ce cas, on termine avec la multiplication des `int` de Python.

Conseils. - Attention aux éventuels résultats intermédiaires nuls et leur double codage dans notre représentation des "grands entiers". - Attention aux éventuels "zéros" inutiles qui peuvent apparaître dans les résultats.

Vous coderez 2 versions de cette multiplication. - `multK()` où les opérandes sont dupliquées avec des `.copy()` - `multK0()` où les opérandes sont manipulés directement avec des tranches (*slices*) de liste Python.

`multK()` : récursion maximale et recopie

[ ]:
```
def multK(u: list[int], v: list[int]) -> list[int]:
    pass
```

[ ]:
```
assert multK([1], [5,6,7]) == inttointlist(567)
assert multK([5,6,7], [1]) == inttointlist(567)
assert multK([1,2], [1,1]) == inttointlist(12 * 11)
assert multK([1,2,3], [1,1]) == inttointlist(123 * 11)
```

```
assert multK([1,2,3], [4,5,6]) == inttointlist(123 * 456)
assert multK([1,2,3,4,5,6,7,8], [4,5,6]) == inttointlist(12345678 * 456)
```

`multK0()`: une seconde récursion maximale en place

[ ]:
```
def multK0(u: list[int], v: list[int]) -> list[int]:
    pass
```

[ ]:
```
assert multK0([1], [5,6,7]) == inttointlist(567)
assert multK0([5,6,7], [1]) == inttointlist(567)
assert multK0([1,2], [1,1]) == inttointlist(12 * 11)
assert multK0([1,2,3], [1,1]) == inttointlist(123 * 11)
assert multK0([1,2,3], [4,5,6]) == inttointlist(123 * 456)
assert multK0([1,2,3,4,5,6,7,8], [4,5,6]) == inttointlist(12345678 * 456)
```

1.10 Timings et tracés

- Evaluer l'efficacité de ces multiplications, de façon similaire aux mesures effectuées plus haut avec `mult()`.
  - Cette fois-ci, le cas 3 (milliers de décimales) sera traité.
  - **Préalablement**, on pourra aussi parcourir ce troisième ensemble en se limitant aux nombre de décimales qui sont des puissance de 2. Vous justifierez pourquoi ce choix est proposé.
  - Si les temps de résolution restent raisonnables sur votre machine, vous pouvez reprendre le cas 3 sans cette contrainte.
- Tracer ces mesures de façon pertinente
  - Vous pouvez sauvegarder vos courbes dans des fichiers (`.png` par exemple).

1.10.1 Complexité théorique de Karastuba

Avant toutes choses, écrire une fonction qui calcule la complexité théorique de cette solution. Le tracé de cette fonction accompagnera les tracés des mesures qui suivent.

[ ]:
```
def comp_theo_Karatsuba(n):
    pass
```

1.10.2 Plusieurs dizaines de décimales

[ ]:
```
# Timings
pass
```

[ ]:
```
# Tracés
pass
```

1.10.3 Plusieurs centaines de décimales

[ ]:
```
# Timings
pass
```

[ ]:
```
# Tracés
pass
```

1.10.4 Des milliers de décimales puissances de 2

[ ]:
```
# Timings
pass
```

[ ]:
```
# Tracés
pass
```

1.11 Conclusion intermédiaire

Que conclure ?

1.12 Avec GMP

gmpy2 permet de travailler avec GMP et de manipuler des entiers arbitrairement longs de type `mpz`.

1.12.1 `gmpy2.mul()`

On peut raisonnablement supposer qu'un Karatsuba (bien optimisé) est derrière `gmpy2.mul()`, la multiplication de `gmpy2`. Regardons-ça …

On se fait la main avec quelques utilisations de cette fonction et les vérifications qui conviennent.

[ ]:
```
pass
```

1.12.2 Timings

On mesure son efficacité.

[ ]:
```
pass
```

1.12.3 Tracés

On traces ces résultats.

[ ]:
```
pass
```

1.12.4 Observations et conclusion

[ ]:
```

```

1.13 Améliorons notre solution

Il y a 2 pistes à explorer (au moins) pour tenter d'améliorer l'efficacité de notre solution – sans espérer toutefois rivaliser avec `gmpy2.mul()`.

1. Déterminer (expérimentalement) un seuil d'efficacité de la récursion de Karatsuba et l'exploiter pour une version plus efficace de ce produit.
   - Ce seuil dépend de notre représentation des "grands entiers" et de notre environnement de calcul.
2. Modifier la représentation actuelle des "grands entiers" et les traitements associés.
   - Cette modification peut être motivée par des contraintes de langage ou par des aspects algorithmiques. Nous détaillerons le premier cas le moment venu. Le second cas (choix d'une base de représentation adaptée au format `int` de l'environnement d'exécution) ne sera pas traité dans cette étude.

1.13.1 Intégrer le seuil d'efficacité de Karatsuba

Les mesures précédentes permettent d'identifier ce seuil. Ecrire une nouvelle version `multKopt` de la multiplication de Karastuba avec seuil. Effectuer rapidement (avec les tracés adaptés) une analyse expérimentale de son efficacité.

[ ]:
```
def multKopt(u: list[int], v: list[int]) -> list[int]:
    pass
```

[ ]:
```
pass
```

1.13.2 Une première autre représentation de grands entiers positifs

On va inverser la représentation précédente : un entier positif est maintenant représenté par la liste de ces chiffres (en base 10) stockés dans une liste par ordre **décroissant de son exposant** en notation de position.

Maintenant le chiffre de plus faible exposant est **en début de la liste** qui stocke ses chiffres.

Ainsi $123 = 1 + 2 \times 10 + 3 \times 10^2$ sera représenté par la liste `[3, 2, 1]`.

On espère que les manipulations de liste qui impactent les chiffres les plus à droite seront moins pénalisantes que si elles étaient effectuées en début de liste.

- On va reprendre les traitements précédents pour les adapter cette nouvelle représentation.

- On se limitera aux fonctions uniquement nécessaires à la multiplication de Karatsuba.
- Ne pas hésiter à recopier les cellules précédentes et effectuer les modifications pour cette nouvelle représentation de ces différentes fonctions.

1.13.3 Les nouvelles versions

Lister et coder les fonctions nécessaires à la multiplication de Karatsuba.

Notation : les identifiants de ces fonctions pour la nouvelle représentation seront les précédents suffixées par `2`, *eg.* `inttointlist2()`, ….

[ ]:
```
pass
```

1.13.4 `multK2` : algorithme de Karatsuba, autre version

On peut maintenant écrire cette nouvelle version.

[ ]:
```
def multK2(u: list[int], v: list[int]) -> list[int]:
    pass
```

[ ]:
```
assert multK2([1], [5,6,7]) == inttointlist2(765)
assert multK2([5,6,7], [1]) == inttointlist2(765)
assert multK2([2,2], [1,1]) == inttointlist2(22 * 11)
assert multK2([6,6,6], [1,1]) == inttointlist2(666 * 11)
assert multK2([1,2,3], [4,5,6]) == inttointlist2(321 * 654)
assert multK2([1,2,3,4,5,6,7,8], [4,5,6]) == inttointlist2(87654321 * 654)
```

1.13.5 Timings et tracés

On effectue les mesures d'efficacité de cette nouvelle version et les tracés correspondants comme précédemment.

Plusieurs centaines de décimales

[ ]:
```
# Timings et tracés
pass
```

Plusieurs milliers de décimales puissances de 2

[ ]:
```
# Timings et tracés
pass
```

Prise en compte du seuil Si les mesures précédentes, le justifie proposer une dernière version `multK2opt()` qui exploite le seuil d'efficacité de la récursion de Karatsuba.

[ ]:
```
def multK2opt(u: list[int], v: list[int]) -> list[int]:
    pass
```

Un dernier tracé pour résumer cette partie

[ ]:
```
pass
```

1.14 Conclusion

[ ]:
```

```