new idea

SOLUTION

Puzzle Gas Stations

Décomposez le problème en deux étapes :

1. Étant donné un indice `start`, vérifiez si `start` est une bonne solution.
2. Essayez avec `start = i` pour chaque `i = 0 ... n-1`.

1. Étant donné un indice `start`, vérifiez si `start` est une bonne solution.

```
success, current_gas = True, 0

for offset in range(0, n):
    current_gas = current_gas + gas[(start + offset) % n]
                              - cost[(start + offset) % n]
    if current_gas < 0:
        success = False
```

2. Essayez avec `start = i` pour chaque `i = 0 ... n-1`.

```
def find_valid_start(gas, cost):
    print("Gas: ", gas)
    print("Cost: ", cost)
    n = len(gas)

    for start in range(0, n):
        success, current_gas = True, 0
        for offset in range(0, n):
            current_gas = current_gas + gas[(start + offset) % n]
                                      - cost[(start + offset) % n]
            if current_gas < 0:
                success = False
        if success == True:
            return start

    return -1
```

new idea

RECURSION

Idée clé : Récursion = Une fonction qui s'appelle elle-même

Une fonction récursive résout un problème en le décomposant en **sous-problèmes identiques** mais plus petits.

Toute fonction récursive a deux parties:

- **Cas de base** : condition qui arrête la récursion
- **Pas récursif** : appel à elle-même avec des données plus petites

Problème: Calculer $n! = n \times (n-1) \times \cdots \times 1$

```
def factorial(n):
    # Cas de base
    if n <= 1:
        return 1

    # Cas recursif
    return n * factorial(n-1)

print( factorial(4) )
```

24

Trace d'appel: `factorial(4)`

```
factorial(4) = 4 * factorial(3)
             = 4 * (3 * factorial(2))
             = 4 * (3 * (2 * factorial(1)))
             = 4 * (3 * (2 * 1))
             = 4 * (3 * 2)
             = 4 * 6
             = 24
```

Problème: Suite de Fibonacci: $F_n = F_{n-1} + F_{n-2}$, avec $F_0 = 0, F_1 = 1$

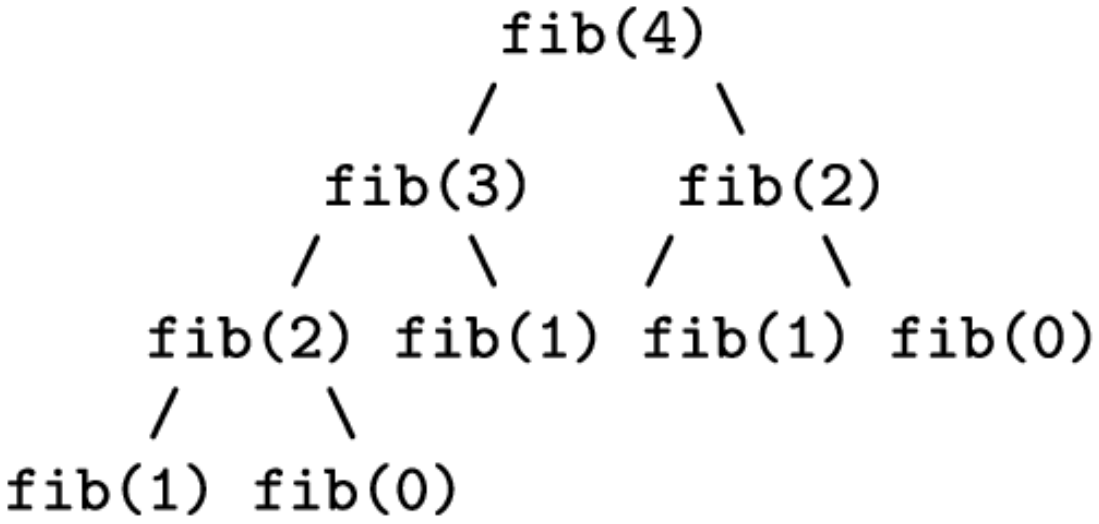
```python
def fibonacci(n):
    # Cas de base
    if n <= 0:
        return 0
    if n == 1:
        return 1

    # Cas recursif
    return fibonacci(n-1) + fibonacci(n-2)

print( fibonacci(8) )
```

21

Arbre d'appels pour `fibonacci(4)`



Problème: Calculer la somme d'une liste d'entiers

```python
def sum_list(lst):
    # Cas de base: liste vide
    if len(lst) == 0:
        return 0

    # Cas recursif
    return lst[0] + sum_list(lst[1:])

print(sum_list([1, 2, 3, 4, 5]))
```

```
15
```

Problème: Vérifier si une chaîne est un palindrome

```python
def is_palindrome(s):
    if len(s) <= 1:
        return True

    if s[0] != s[-1]:
        return False

    return is_palindrome(s[1:-1])

print(is_palindrome("radar"))
print(is_palindrome("python"))
```

```
True
False
```

Avantages:

Code élégant et concis

Solution naturelle pour structures récursives (arbres)

Facile à comprendre pour certains problèmes

Inconvénients:

Peut être plus lent (appels de fonction)

Risque de dépassement de pile

Parfois moins efficace qu'itération

Problèmes adaptés à la récursion:

Parcours d'arbres (répertoires, XML)

Algorithmes diviser-pour-régner (tri fusion, quicksort)

Problèmes mathématiques (factorielle, Fibonacci)

Backtracking (sudoku, labyrinthe)

Récursion vs Itération

Récursion	**Itération**
S'appelle elle-même	Utilise des boucles
Termine par cas de base	Termine par condition
Utilise la pile	Utilise des compteurs
Plus intuitif pour problèmes divisibles	Plus efficace en mémoire

À ÉVITER: Récursion infinie!

```
def bad_recursion(n):
    return n * bad_recursion(n-1)

def good_recursion(n):
    if n <= 1:
        return 1
    return n * good_recursion(n-1)
```

Toujours vérifier que le cas de base est atteignable!

Problem: Compter les chiffres

```
def count_digits(n):
    if abs(n) < 10:
        return 1

    return 1 + count_digits(n // 10)

print(f"12345 a {count_digits(12345)} chiffres")
print(f"-987 a {count_digits(-987)} chiffres")
```

Le cas de base a oublié de gérer les valeurs négatives !

```
12345 a 5 chiffres
-987 a 4 chiffres
```

La pile d'appels (Call Stack)

Chaque appel récursif empile:

- Les paramètres de la fonction
- Les variables locales
- L'adresse de retour

Quand le cas de base est atteint, on dépile.

L'Écueil Fréquent

'J'ai essayé de suivre tous les appels de fonction jusqu'au cas de base et retour
... et maintenant j'ai mal à la tête.'

Le Piège Mental :

Appel de `fibonacci(5)`.

Il appelle `fibonacci(4)` et `fibonacci(3)`.

Ils appellent `fibonacci(3)`, `fibonacci(2)`, `fibonacci(2)`, `fibonacci(1)`

... (On perd le fil).

Cela mène à la confusion et à la surcharge cognitive.

Faites confiance à la récursivité — elle fonctionne !

Astuce : Ne raisonnez qu'à un seul niveau d'abstraction.
"Faites comme si l'appel récursif fonctionnait déjà."

On raisonne par récurrence : on suppose que la fonction récursive fonctionne pour une taille $n - 1$ (hypothèse de récurrence), et on la construit pour n.

La Bonne Approche :

`factorial(5)` = $5 \times$ `factorial(4)`

Principe de la Boîte Noire :

Je **ne sais pas** comment `factorial(4)` calcule.

Je **sais** qu'il me renvoie la bonne réponse (ici 24).

Donc `factorial(5)` = $5 \times 24 = 120$.

Résumé: La Récursion

Fonction qui s'appelle elle-même

Doit avoir un **cas de base**

Doit progresser vers le cas de base

Utilise la pile d'appels

Élégant mais attention à la performance