# MPII
# ITC
# Séquence 6

## Algorithmes gloutons

# 1. Activités introductives

## 1.1. Optimiser : problème du représentant de commerce

| | Clermont-Ferrand | La Rochelle | Lille | Limoges | Lyon | Marseille | Nantes | Paris |
|---|---|---|---|---|---|---|---|---|
| La Rochelle | 462 | | | | | | | |
| Lille | 622 | 688 | | | | | | |
| Limoges | 173 1 | 222 | 608 | | | | | |
| Lyon | 165 | 614 | 681 | 385 2 | | | | |
| Marseille | 413 | 823 | 1001 | 593 | 314 | | | |
| Nantes | 534 | 137 | 600 | 320 | 655 | 986 | | |
| Paris | 423 | 472 | 225 | 392 | 466 3 | 775 | 385 | |
| Toulouse | 377 5 | 421 | 894 | 290 | 467 | 404 | 585 | 678 4 |

- Proposer quelques trajets possibles avec leur kilométrage respectif puis déterminer le trajet optimal que recherche le représentant.

Clermont-Ferrand -> Limoges(173) ->Lyon(558) -> Paris(1024) -> Toulouse (1702)-> Clermont-Ferrand (2079)

# 1. Activités introductives

## 1.1. Optimiser : problème du représentant de commerce

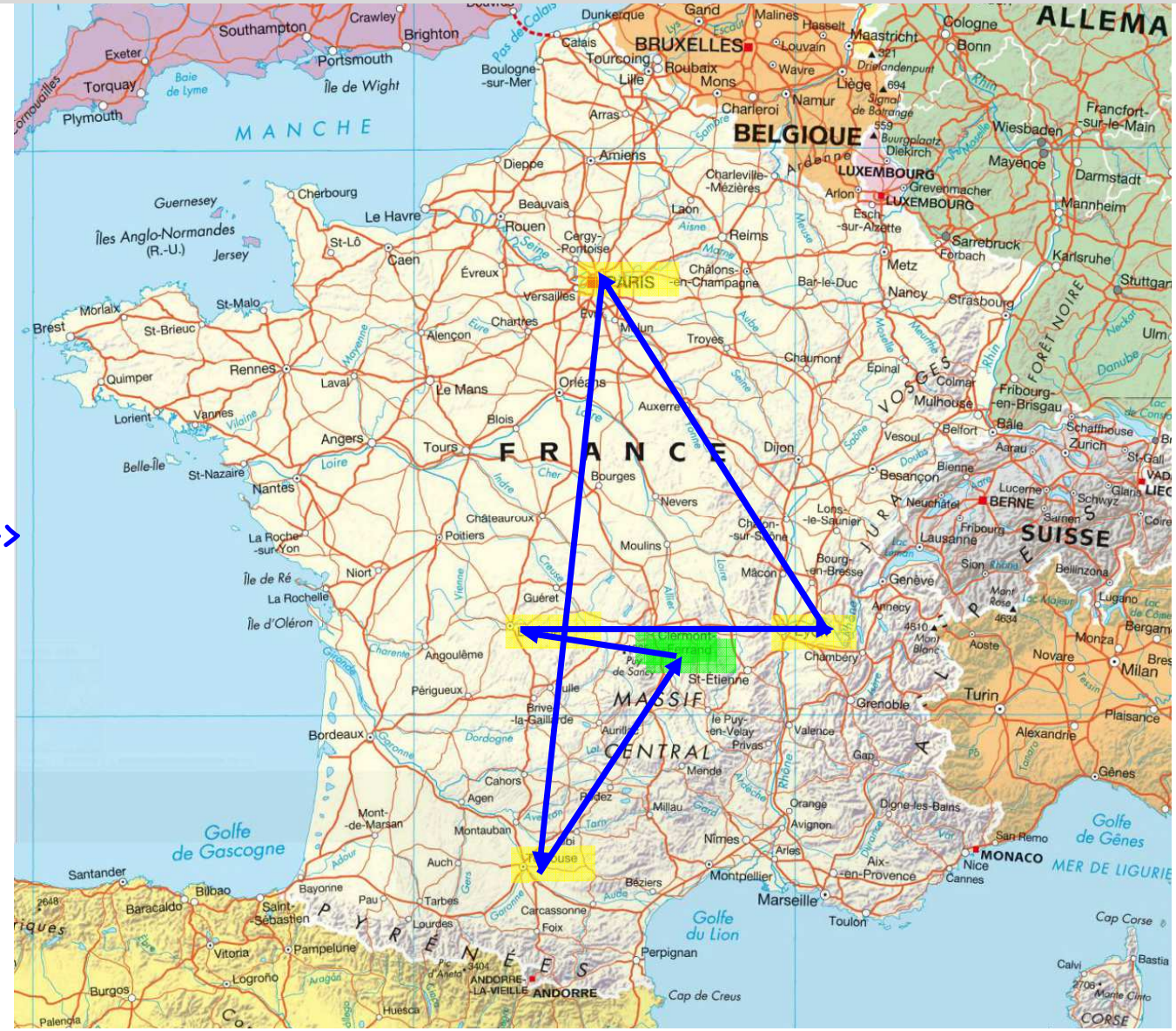
| | Clermont-Ferrand | La Rochelle | Lille | Limoges | Lyon | Marseille | Nantes | Paris |
|---|---|---|---|---|---|---|---|---|
| La Rochelle | 462 | | | | | | | |
| Lille | 622 | 688 | | | | | | |
| Limoges | 173 | 222 | 608 | | | | | |
| Lyon | 165 1 | 614 | 681 | 385 2 | | | | |
| Marseille | 413 | 823 | 1001 | 593 | 314 | | | |
| Nantes | 534 | 137 | 600 | 320 | 655 | 986 | | |
| Paris | 423 5 | 472 | 225 | 392 | 466 | 775 | 385 | |
| Toulouse | 377 | 421 | 894 | 290 3 | 467 | 404 | 585 | 678 4 |

- Proposer quelques trajets possibles avec leur kilométrage respectif puis déterminer le trajet optimal que recherche le représentant.

Clermont-Ferrand -> Lyon(165) -> Limoges (550)-> Toulouse((840)->Paris (1518)-> Clermont-Ferrand (1941)

# 1. Activités introductives

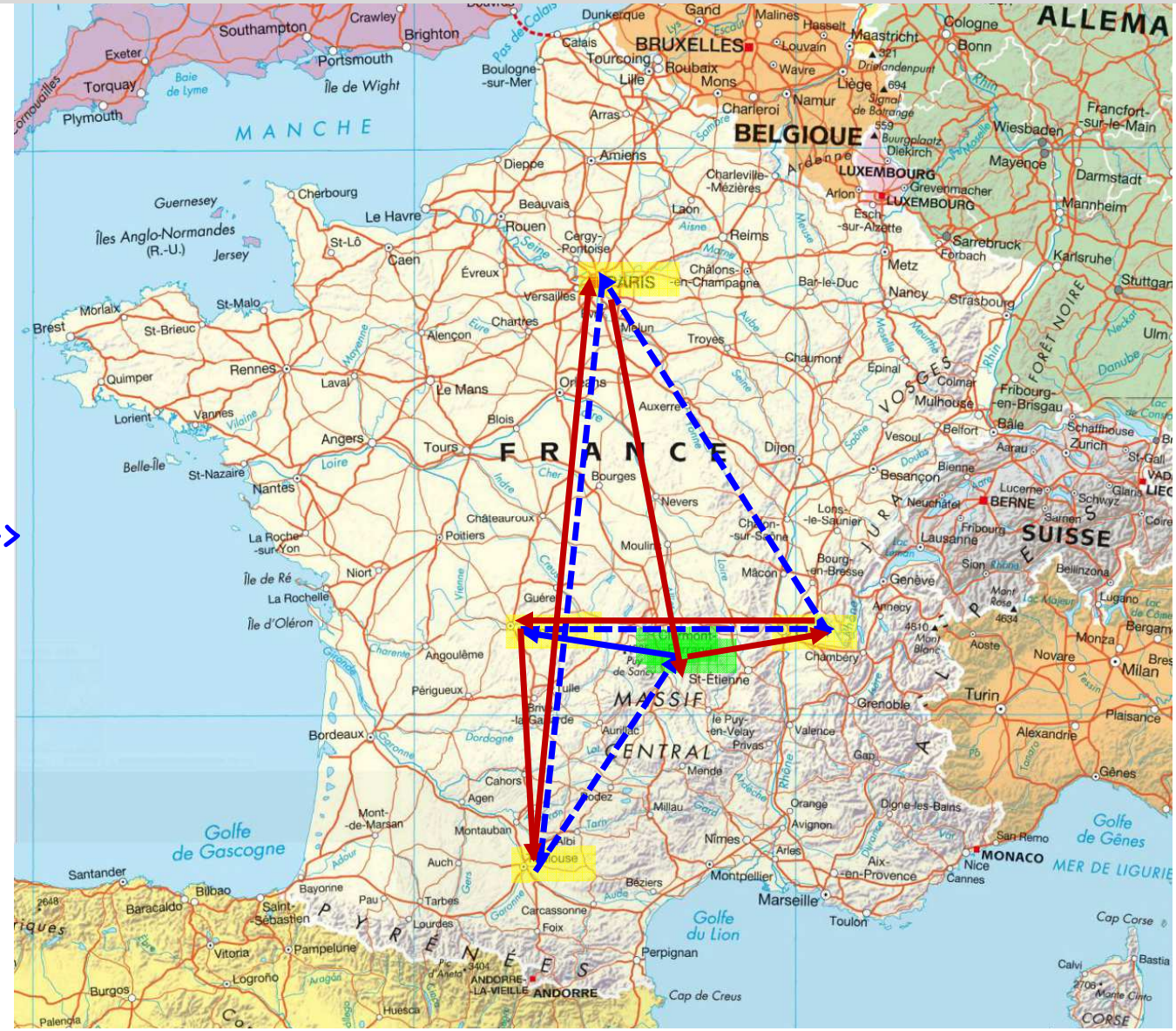
Clermont-Ferrand -> Limoges(173) -> Lyon(558) -> Paris(1024) -> Toulouse(1702) -> Clermont-Ferrand (2079)



# 1. Activités introductives

« Un » Choix :

Clermont-Ferrand -> Limoges(173) -> Lyon(558) -> Paris(1024) -> Toulouse(1702) -> Clermont-Ferrand (2079)




Choix localement optimal :

Clermont-Ferrand -> Lyon(165) -> Limoges(550) -> Toulouse(840) -> Paris (1518) -> Clermont-Ferrand (1941)

# 1. Activités introductives

Clermont-Ferrand -> Limoges(173) -> Lyon(558) -> Paris(1024) -> Toulouse(1702) -> Clermont-Ferrand (2079)

Clermont-Ferrand -> Lyon(165) -> Limoges (550) -> Toulouse((840) -> Paris (1518) -> Clermont-Ferrand (1941)


Choix optimal :

Clermont-Ferrand -> Lyon(165) -> Paris (631) -> Limoges(1023) -> Toulouse(1313) -> Clermont-Ferrand (1690)

# 1. Activités introductives

## 1.2. Optimiser : problème du rendu de monnaie

Lors d'un achat en espèces l'acheteur et le vendeur échangent des pièces et des billets, le problème consiste à rendre la monnaie avec un minimum de pièces.
Dans le système monétaire européen, les « pièces » prennent les valeurs entières suivantes $c_i$ d'une liste $coupures\_euro = [500, 200, 100, 50, 20, 10, 5, 2, 1]$, liste triée par ordre décroissant des valeurs.

- En prenant l'hypothèse que la quantité de coupures (pièces ou billets) pour chacune des valeurs $c_i$ est illimitée, alors, la monnaie peut être rendue avec de nombreuses combinaisons.
- Quantifier le nombre de chaque coupure du rendu permet d'évaluer la qualité de la solution.
- Il existe de nombreuses solutions de rendu dont l'une sera la meilleure.

➢ **Il s'agit bien d'un problème d'optimisation !**

# 1. Activités introductives

## 1.2. Optimiser : problème du rendu de monnaie

- *soit $s$ la somme à rendre, exprimer $s$ en fonction de $k_i$, nombre d'éléments de $c_i$ nécessaire.*

$$s = \sum_{i=1}^{n} k_i . c_i$$

- *Quelle expression cherche-t-on à minimiser ?*

$$\sum_{i=1}^{n} k_i$$

- *Compléter la fonction en version itérative qui cherche à minimiser le nombre de coupure du rendu de monnaie,*
- *Compléter la fonction en version récursive de cette même fonction.*

# 1. Activités introductives

## 1.2. Optimiser : problème du rendu de monnaie

➢ *Compléter la fonction en version itérative.*

```
def rendu_monnaie_it(somme, liste_coupures):
    """
    Renvoie le découpage glouton du rendu de monnaie.
    On considère que chaque coupure est disponible en quantité infinie.
    parameters:
        somme (int): la quantité de monnaie à rendre
        liste_coupures (list[int]): les coupures disponibles
    Returns:rendu (list[int]): la liste des coupures à rendre
    Préconditions: liste_coupures triée dans l'ordre décroissant"""
```

Documentation

# 1. Activités introductives

## 1.2. Optimiser : problème du rendu de monnaie

➢ *Compléter la fonction en version itérative.*

```
def rendu_monnaie_it(somme, liste_coupures):
    """
    Renvoie le découpage glouton du rendu de monnaie.
    On considère que chaque coupure est disponible en quantité infinie.
    parameters:
        somme (int): la quantité de monnaie à rendre
        liste_coupures (list[int]): les coupures disponibles
    Returns:rendu (list[int]): la liste des coupures à rendre
    Préconditions: liste_coupures triée dans l'ordre décroissant"""
```

Documentation

# 1. Activités introductives

## 1.2. Optimiser : problème du rendu de monnaie

➢ *Compléter la fonction en version itérative.*

```
def rendu_monnaie_it(somme, liste_coupures):
    """
    Renvoie le découpage glouton du rendu de monnaie.
    On considère que chaque coupure est disponible en quantité infinie.
    parameters:
        somme (int): la quantité de monnaie à rendre
        liste_coupures (list[int]): les coupures disponibles
    Returns:rendu (list[int]): la liste des coupures à rendre
    Préconditions: liste_coupures triée dans l'ordre décroissant"""
    rendu = [] #initialisation de la liste de rendu
    while somme != 0:
        compteur_coupures = 0 # Permet de choisir la coupure
        # recherche de la plus grande coupure
        while somme < liste_coupures[compteur_coupures]:
            compteur_coupures += 1
        somme = somme - liste_coupures[compteur_coupures]
        rendu.append(liste_coupures[compteur_coupures])
    return rendu
```

Documentation

➢ *Tester votre fonction pour un rendu de 12€, 41€ et 413€.*

# 1. Activités introductives

## 1.2. Optimiser : problème du rendu de monnaie

➢ *Compléter la fonction en version récursive.*

```
def rendu_monnaie_rec(somme, l_coupures):
    """
    Renvoie le découpage glouton du rendu de monnaie.
    On considère que chaque coupure est disponible en quantité infinie.
    parameters:
        somme (int): la quantité de monnaie à rendre
        liste_coupures (list[int]): les coupures disponibles
    Returns:rendu (list[int]): la liste des coupures à rendre
    Préconditions: liste coupures triée dans l'ordre décroissant """
```

Documentation

# 1. Activités introductives

## 1.2. Optimiser : problème du rendu de monnaie

➢ *Compléter la fonction en version récursive.*

```
def rendu_monnaie_rec(somme, l_coupures):
    """
    Renvoie le découpage glouton du rendu de monnaie.
    On considère que chaque coupure est disponible en quantité infinie.
    parameters:
        somme (int): la quantité de monnaie à rendre
        liste_coupures (list[int]): les coupures disponibles
    Returns:rendu (list[int]): la liste des coupures à rendre
    Préconditions: liste_coupures triée dans l'ordre décroissant """
    #condition d'arrêt
    if somme == 0:
        return []
    #recherche de la plus grande coupure à rendre
    compteur_coupures = 0
    while  somme < l_coupures[compteur_coupures]:
            compteur_coupures += 1
    #récupération des coupures à rendre
    return [l_coupures[compteur_coupures]] + rendu_monnaie_rec((somme-l_coupures[compteur_coupures]),l_coupures)
```

Documentation

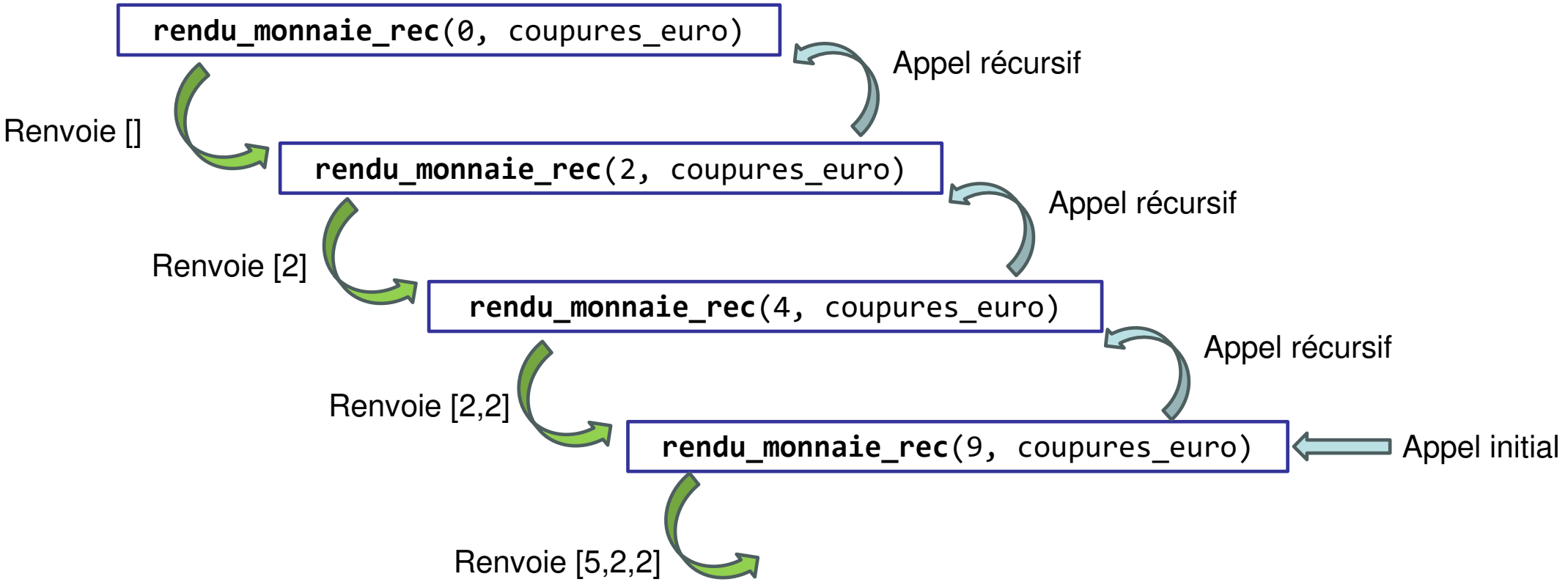
➢ *Etablir le schéma d'appel pour une somme de 9€*

# 1. Activités introductives

## 1.2. Optimiser : problème du rendu de monnaie

***Schéma d'appel pour une somme de 9€***

```
coupures_euro = [500, 200, 100, 50, 20, 10, 5, 2, 1]
```



# 1. Activités introductives

## 1.2. Optimiser : problème du rendu de monnaie

➢ *Compléter la fonction en version récursive.*

```
def rendu_monnaie_rec(somme, l_coupures):
    """
    Renvoie le découpage glouton du rendu de monnaie.
    On considère que chaque coupure est disponible en quantité infinie.
    parameters:
        somme (int): la quantité de monnaie à rendre
        liste_coupures (list[int]): les coupures disponibles
    Returns:rendu (list[int]): la liste des coupures à rendre
    Préconditions: liste_coupures triée dans l'ordre décroissant """
    #condition d'arrêt
    if somme == 0:
        return []
    #recherche de la plus grande coupure à rendre
    compteur_coupures = 0
    while  somme < l_coupures[compteur_coupures]:
            compteur_coupures += 1
    #récupération des coupures à rendre
    return [l_coupures[compteur_coupures]] + rendu_monnaie_rec((somme l_coupures[compteur_coupures]),l_coupures)
```

Documentation

➢ *Tester votre fonction pour un rendu de 12€, 41€ et 413€.*

# 1. Activités introductives

## 1.2. Optimiser : problème du rendu de monnaie

- *Tester votre fonction pour un système monétaire différent, par exemple, l'ancien système britannique : la liste des « pence » disponible était*

$$coupures\_pence = [240, 120, 60, 30, 24, 12, 6, 4, 3, 1] .$$

*Le rendu pour 48 est-il optimal ?*

# 1. Activités introductives

## 1.2. Optimiser : problème du rendu de monnaie

- Pour un système de monnaie non canonique:

**coupure_pence = [240, 120, 60, 30, 24, 12, 6, 4, 3, 1]**

***Rendu de monnaie pour 48 pence?***

| MODE GLOUTON | MODE OPTIMAL |
|---|---|
| 30 ; 12 ; 6<br><br>➢ 3 coupures | 24 ; 24<br><br>➢ 2 coupures |

Un système monétaire dans lequel l'algorithme glouton optimise le rendu de monnaie est dit canonique.

# 2. DÉFINITION

## 2.1. Contexte : un problème d'optimisation

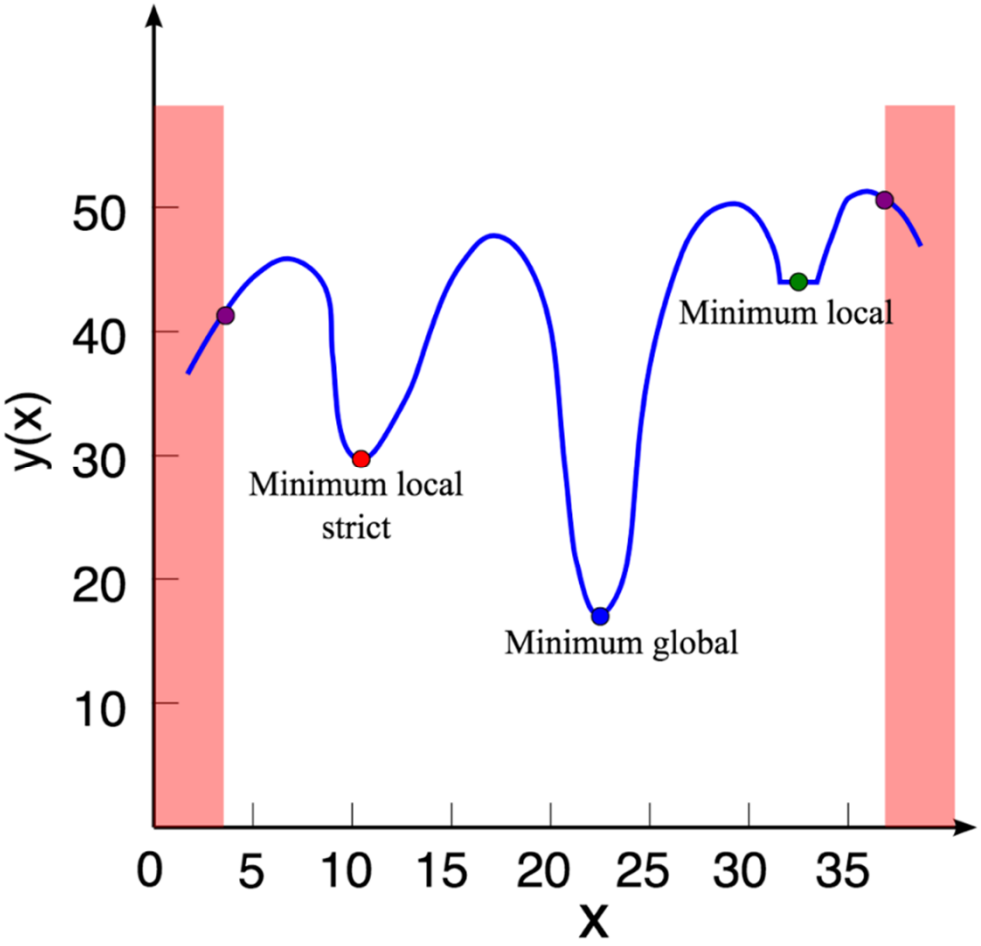
- Objectif:

**Trouver la « meilleure » solution au regard d'un critère tout en respectant une série de contraintes.**

- Contexte:
  - **un très grand nombre de solutions potentielles,**
  - **une fonction (à maximiser/minimiser) permettant d'évaluer la qualité de chaque solution,**
  - **l'existence d'une solution optimale, ou suffisamment bonne.**

# 2. DÉFINITION

## 2.2. Principe d'optimisation




- Les optimums:
  - optimum global,
  - optimum local strict (unique pour intervalle),
  - optimum local (non unique pour intervalle).

- Le raisonnement glouton:

Se précipiter « goulûment », étape par étape, sur un choix optimal localement, dans l'espoir d'obtenir un résultat optimal globalement.

# 2. DÉFINITION

## 2.3. Résolution par heuristique

Une heuristique est une méthode de résolution qui permet d'obtenir rapidement une solution approchée d'un problème, pas nécessairement la meilleure.

## 2.4. L'approche gloutonne

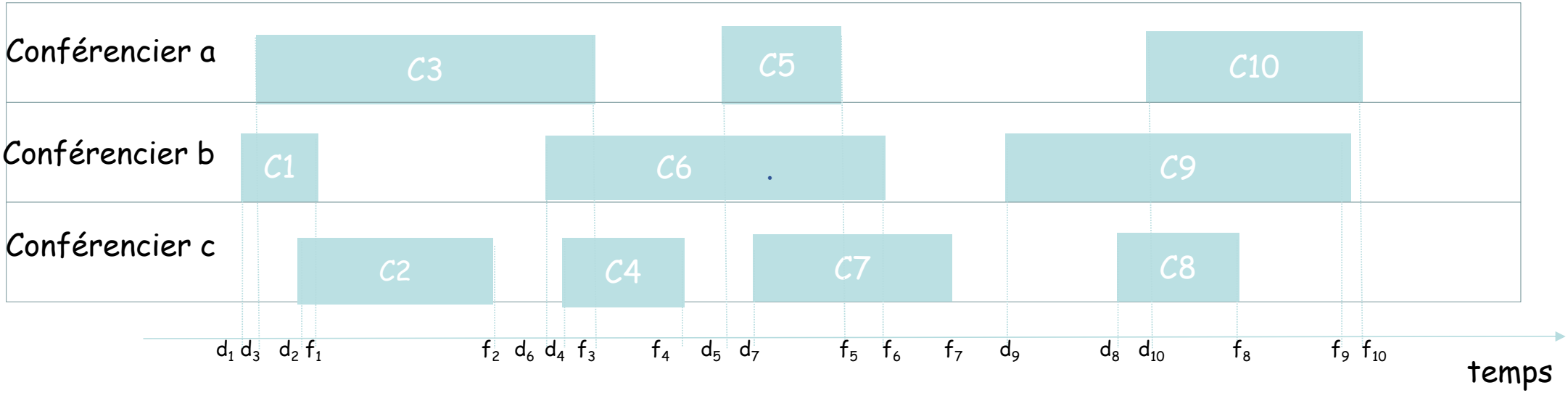
Algorithme d'optimisation au cours duquel:

- on choisit une **solution partielle** qui semble **optimale localement**,
- on considère le sous-problème restant et applique la même méthode;
- on ne remet **jamais** en cause une décision prise au cours des étapes précédentes.

# 3. ILLUSTRATION CLASSIQUE: UN CHOIX D'ACTIVITÉ

Allocation d'une salle à différents conférenciers pour une journée de colloque:

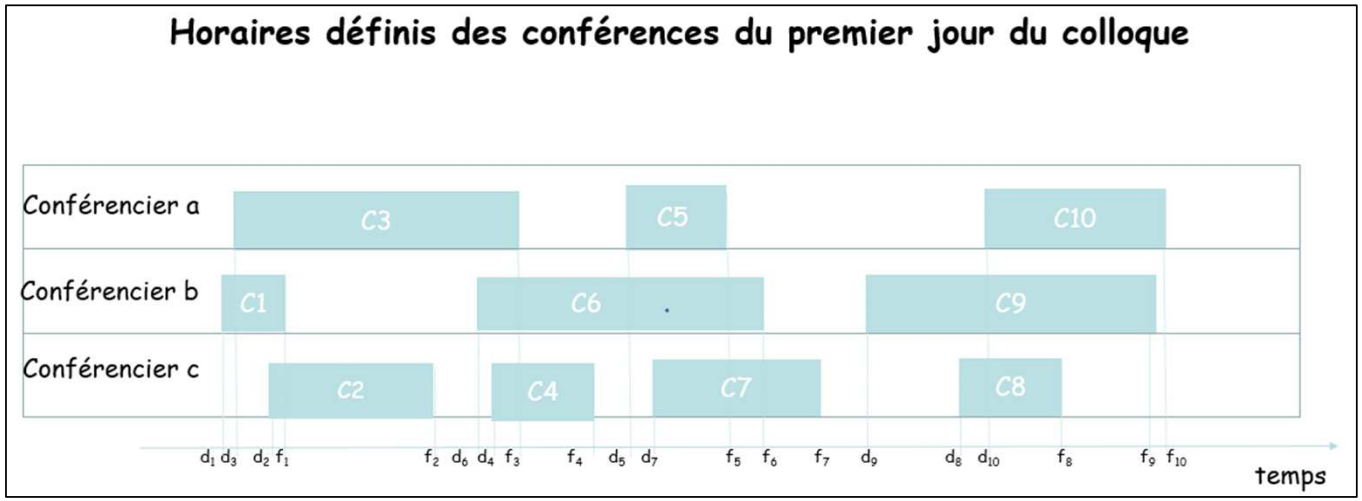
**Horaires définis des conférences du premier jour du colloque**




$$f(c_1) < f(c_2) < \cdots < f(c_{n-1}) < f(c_n)$$

# 3. ILLUSTRATION CLASSIQUE: UN CHOIX D'ACTIVITÉ

Allocation d'une salle à différents conférenciers pour une journée de colloque:




$$f(c_1) < f(c_2) < \cdots < f(c_{n-1}) < f(c_n)$$

➢ *Est-ce un problème d'optimisation ?*

- Présence de nombreuses combinaisons des conférences,
- Existence d'une fonction permettant de quantifier la qualité des solutions, elle compte le nombre de conférences: `len(solution)`.
- Optimisation possible en maximisant le nombre de conférences (en tenant compte de la contrainte, ici les conférences doivent être compatibles $f(s_i) \leq d(s_j)$).

## 3. ILLUSTRATION CLASSIQUE: UN CHOIX D'ACTIVITÉ

- *Le choix glouton consiste à choisir la conférence qui termine le plus tôt possible, préservant ainsi la disponibilité de la salle pour les conférences ultérieures.*

- Conférences référencées sous forme de tuples formés comme suit:

(NomConf, début, fin)

```
liste_confJ1 = [ ('C1',8,9) , ('C2',8.75,10.25) , ('C3',8.25,11.5) ,
('C4',11.25,12.25) , ('C5',13,14.5) , ('C6',11,15) , ('C7',13.25,15.5) , ('C8',17,18),
('C9',15.75,18.5) ,('C10',17.25,18.75)  ]
```

- ***Ecrire*** *le script d'une fonction* `choix_conferences(listeC: list)->list` *qui renvoie une liste de conférences en maximisant le nombre de celles-ci à partir d'une liste de tuples de conférences possibles,* `listeC`.

- ***Tester*** *votre fonction pour la journée 1 du colloque.*