# Algorithmes et Structures de données
# Problème de la monnaie rendue

INF TC1
2017-18 (S5)

Version Élève

# I Problème de monnaie

• Fonctionnement de la machine à rendre la monnaie.
• Le problème est connu sous le nom de *"Change making"* (rendre la monnaie)

### Définition du problème :

Dans une machine capable de rendre la monnaie, on a des pièces de 1c à 2€s, puis des billets. On suppose pour simplifier qu'il n'y a que des pièces. Un billet de 5€s sera représenté comme une pièce de 500 centimes.
De plus, on suppose qu'il existe un nombre suffisant (autant que nécessaire) de chaque pièces.

Un stock de pièces est alors un tuple $S = (v_1, v_2, ...., v_n)$ où l'entier $v_i > 0$ est la valeur de la ième pièce.
Pour refléter le fait qu'on a des pièces de 1,2 et 5c, $S$ contiendra $v_1 = 1$ (1 centime), $v_2 = 2, v3 = 5$.

Pour un système **S** et un entier positif **M**, *le problème de monnaie* est un problème d'optimisation combinatoire $(S, M)$ permettant de trouver le tuple $T = (x_1, x_2, ..., x_n)$ avec $x_i \geq 0$ qui minimise $\sum_{i=1}^{n} x_i$ sous la contrainte $\sum_{i=1}^{n} x_i.v_i = M$.
Autrement dit, sachant que $x_i$ est le nombre de pièces de valeur $v_i$ utilisées, on veut minimiser le nombre total de $x_i$ utilisés.

Exemple : pour rendre la somme *M*, on utilisera $x_i$ fois la pièce $v_i$. Pour un stock de pièces contenant :

| **i** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| $\mathbf{v_i}$ | 1 | 2 | 5 | 10 | 20 | 50 | 100 | 200 | 500 | ... |

donc avec $S = (1, 2, 5, 10, 20, 50, 100, 200, 500, ...)$, si on utilise 3 pièces de 2€s (200c : $v_8 = 200$), on aura dans la réponse $T$ la valeur $x_8 = 3$.

Dans ce problème d'optimisation, l'objectif à minimiser est le nombre total de pièces rendues.
Appelons $Q(S, M) = \sum_{i=1}^{n} x_i$ la quantité de pièces à rendre pour le montant *M* étant donné le système *S* décrit ci-dessus. Une solution optimale à ce problème est telle que *Q(S,M)* soit minimale : $Q_{opt}(S, M) = min \sum_{i=1}^{n} x_i$.

• **Un exemple :** pour rendre 9€s étant donné S ci-dessus, parmi les possibilités (en n'utilisant que les pièces ≥1€=100c) :

- 9 pièces de 1€ et 0 pour toutes les autres — **T=(0,0,0,0,0,0,0,9,0,0,0...0)** → 9 pièces, $Q(S, 9) = 9$
- 5 × 1€+2 × 2€s, 0 pour les autres — **T=(0,0,0,0,0,0,0,5,2,0,0...0)** → 7 pièces, $Q(S, 9) = 7$
- 1 × 1€+4 × 2€s, 0 pour les autres — **T=(0,0,0,0,0,0,0,1,4,0,0...0)** → 5 pièces, $Q(S, 9) = 5$
- 2 × 2€s+1 × 5€s, 0 pour les autres — **T=(0,0,0,0,0,0,0,0,2,1,0...0)** → 3 pièces, $Q(S, 9) = 3$
- 3 × 1 + 3 × 2, 0 pour les autres — **T=(0,0,0,0,0,0,0,3,3,0,0...0)** → 6 pièces,
- 4 × 1 + 1 × 5, 0 pour les autres — **T=(0,0,0,0,0,0,0,4,0,1,0...0)** → 5 pièces,
- etc. sans parler des solutions avec des centimes !

La solution *T=(0,0,0,0,0,0,0,0,2,1,0...0)* minimise le nombre total de pièces rendus est de 3 pièces.
Donc, $Q_{opt}(S, 9) = min\ Q(S, 9) = 3$.

• **Un autre exemple** : pour rendre la somme de à 1989€s pièces (sans les centimes), on aura :
$1989 = 500 \times 3 + 488 = 500 \times 3 + 200 \times 2 + 50 \times 1 + 20 \times 1 + 10 \times 1 + 5 \times 1 + 2 \times 2$
soit $3 + 2 + 1 + 1 + 1 = 8$ grosses pièces (billets) et $1 + 2 = 3$ pièces, tout est en euro.

### Résolution du problème

Pour résoudre ce problème (démontré NP-difficile relative à la taille de *S*), plusieurs approches sont possibles dont les approches :

I. Algorithmique : technique Gloutonne (pas toujours optimale)
II. Algorithmique : chemin de longueur minimale avec ou sans un arbre de recherche,
III. Algorithmique : Programmation Dynamique, ... et autres méthodes algorithmiques (cf. quasi-Dijkstra)
IV. Système d'équations à optimiser
Etc.

# II Approche algorithmique

Soit $M$ la somme à rendre et $S$ un tableau de pièces/billets disponibles. Un exemple de $S$ est donné ci-contre :

On se limite arbitrairement aux pièces/billets 100 €s max.

Dans sa forme complète (cas réel), on gère également le nombre de chaque pièce/billet disponible (la table **D**).

**d[i]=k** veut dire : il y a **k** pièces/billets du montant $v_i$ disponibles (pièces ou billets du montant *v[i]*) à l'indice *i* dans la table *S*.

Pour satisfaire la somme M, les solutions algorithmiques se déclinent sous différentes formes.
Nous abordons la technique dite Gloutonne, le chemin minimal dans un arbre de recherche et **la Programmation Dynamique**.
☞ Ci-dessous, on ne tient pas compte de la table D.

| | **la table S** | **la table D** |
|---|---|---|
| indice | Valeur ($v_i$) | Disponibilité ($d_i$) |
| 1 | 1c | nombre de pièces de 1c disponibles |
| 2 | 2c | nombre de pièces de 2c disponibles |
| 3 | 5c | |
| 4 | 10c | |
| 5 | 20c | |
| 6 | 50c | |
| 7 | 100 (1€ ) | 1€ |
| 8 | 200 (2€s ) | pièces 2€s |
| 9 | 500 (5€s ) | billets de 5€s |
| 10 | 1000 | billets de 10€s |
| 11 | 2000 | billets de 20€s |
| 12 | 5000 | billets de 50€s |
| 13 | 10000 | billets de 100€s |

On suppose donc qu'il y a un nombre suffisant de chaque pièce/billet dans la tableau S. On suppose également que S est ordonnée croissante.

## II-1 Technique Gloutonne

Cette méthode ne donne pas forcément un nombre minimal de pièces mais elle est simple à implanter.
Son principe est simple : on trouve la pièce la plus grande inférieure ou égale à $M$. Soit $V_i$ cette pièce. On utilise ($x_i = M\ div\ v_i$) fois la pièce $v_i$ ; le reste à traiter sera $M'$=($M\ mod\ v_i$). Ensuite, on recommence suivant le même principe pour satisfaire $M'$...
Ce qui donne l'algorithme de principe suivant (on ne tient pas compte ici de $D$ : $d_i = \infty$) :

```
Fonction Monnaie_Gloutonne
Entrées: la somme S, M
Sorties: le vecteur  T = Q(S,M) : le nombre de pièces nécessaires
    M'=M
    Répéter
        Chercher dans S l'indice i tel que Vi =< M'
        Ti = M div Vi                          % on utilisera Ti fois la pièce Vi
        M' = M mod Ti
    Jusqu'à M' = 0

    Constituer T avec les Ti utilisés
    Q(S,M) = sum_{i=1}^{n} Ti          % somme des valeurs du vecteur T
Fin Monnaie_Gloutonne
```

**Exemple :** pour satisfaire 236,65€s , cet algorithme se déroule de la manière suivante :

$23665 >= 10000$
$T_{13} = 23665\ div\ 10000 = 2$ % $T_{13}$ correspond à 100€s
$M' = 23665\ mod\ 10000 = 3665$ % on utilisera 2 billets de 100€s

$3665 >= 2000$
$T_{11} = 3665\ div\ 2000 = 1$ % $T_{11}$ correspond à 20€s
$M' = 3665\ mod\ 2000 = 1665$ ...

→ On obtient $T_{1..13} = (0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 2)$ et $Q(S, M) = 9$

☞ En Python, les indices commencent à zéro ! Faire -1 sur les indices ci-dessus.

Remarque :
Dans certains systèmes (cf. un parking), l'utilisateur paie une somme et attend sa monnaie (le reste). On peut d'emblée intégrer les pièces introduites (par un client) au tableau $S$. De cette sorte, on pourra toujours essayer de satisfaire une demande (même si $S$ est initialement vide auquel cas un échec suivra !).

### II-1-a Prise en compte de la table D

On a supposé disposer d'une quantité suffisante de chaque pièce (Donc $d_i$ illimité et suffisamment grande). Modifier la solution précédente pour tenir compte du nombre disponible de chaque pièce (table D en paramètre, avec $d_i$ limité).

## II-2 Version optimale

**Raisonnement** : admettons que l'on sache calculer une solution récursive pour *Gloutonne* qui sera utilisée dans la version optimale.

• Essayons le raisonnement suivant.
Pour un `indice` et pour un montant $M$ donné :
- ◦ on cherche une solution normale **EN UTILISANT** la pièce `S[indice]` (sous le contrôle de *D[indice]*).
- ➛ On appelle ce résultat *avec* dans le code ci-dessous.
- ◦ on cherche une solution **SANS UTILISER** la pièce `S[indice]` (en forçant provisoirement *D[indice]=0*).
- ➛ On appelle ce résultat *sans* dans le code ci-dessous.
- ◦ On retient le minimum des deux : vaut-il mieux *avec* ou *sans* ?

☞ Cette solution n'est pas demandée (dans le minimum du travail à rendre), mais connaître son principe facilite la compréhension de la solution par la programmation dynamique (section suivante).

# III Programmation Dynamique (PrD)

• Pour une introduction à la Programmation Dynamique, voir en Annexes (et le support du cours).

Explications :
Supposons que l'on puisse, pour le montant *M*, savoir calculer une solution optimale pour tout montant $M' < M$. Pour satisfaire *M*, il faudra alors prendre une (seule) pièce $v_i$ supplémentaire parmi les *n* pièces disponibles. Une fois cette pièce choisie, le reste $M' = M - v_i$ est forcément inférieur à *M* et on sait qu'on peut calculer un nombre optimal de pièces pour *M'*. Par conséquent : $Q_{opt}(S, M) = 1 + Min\ Q(S, M')$.

Le raisonnement ci-dessus est un raisonnement par Induction Mathématique.
L'inconvénient de cette méthode est que le nombre d'opérations nécessaires est proportionnel à *M*, donc exponentiel en la taille de *M* (sauf si *M* est représenté avec des bâtons - système monadique !).

Dans cette partie, on utilise la programmation dynamique dont les principes sont rappelés ici :
- ◦ identifier une famille de sous-problèmes,
- ◦ exprimer les équations qui relient ces sous-problèmes, puis résoudre ces problèmes dans un ordre ascendant.

**Propriétés de la solution :**
- ◦ Pour la somme $m \in [0, M]$, notons $Q_{opt}(i, m) = min\ Q(i, m)$ qui sera le nombre **minimal** de pièces nécessaires pour former la somme **m** lorsque seules les pièces de type $i..1, i \leq |S|$ sont disponibles ($|S|$ est la taille de $S$).

  Pour être *pratique*, sachant que le stock de pièces **S** ne varie pas, on fait varier $i \in 1..|S|$ en raisonnant sur la $i^{eme}$ pièce (le $i^{eme}$ élément de $S$ est une pièce de type $i$ : $S[i]$ vaut $v_i$ cents, cf. plus haut) :

  - ◦ Pour atteindre une somme nulle, aucune pièce n'est nécessaire.
    Donc, nous avons : $Q(i, 0) = 0, \forall i \in 1..|S|$ (autrement dit, quelque soit **S**, on a besoin d'aucune pièce si $M = 0$)
  - ◦ Atteindre une somme non nulle est impossible si l'on ne dispose d'aucune pièce (caractérisé ici par $i = 0$).
    Donc, nous avons : $Q(0, m) = \infty\ si\ 0 < m \leq M$
  - ◦ Enfin, pour atteindre une somme quelconque **M**,
    - soit on utilise au moins une pièce de type *i*, auquel cas les autres pièces sont de types *1* à *i* ($i$ est compris) et doivent constituer une manière optimale d'atteindre la somme $S - v_i$ ;
    - soit on n'en utilise aucune (du type $i$), auquel cas les pièces utilisées seront de types *1* à $i - 1$.

    ☞ Remarquons que $i$ décroit : ses valeurs vont de $|S|$ à $0$.

Nous avons par conséquent ($i : 1..|S|$ représente une pièce dans le stock $S$ de valeur $v_i$, la somme $m \in 1..M$) :

$$Q_{opt}(i, m) = min \begin{cases} 1 + Q(i, m - v_i) & si\ (m - v_i) \geq 0 \quad \textit{on utilise une pièce de type i de valeur } v_i \\ Q(i - 1, m) & si\ i \geq 1 \quad \textit{on n'utilise pas la pièce de type i, essayons i-1} \end{cases}$$

Pour être fidèle au principe de *mémorisation* de la PrD, on remplit une matrice $mat[|S|][M]$. Pour ce faire, la version *ascendante* (il existe également une version *descendante*) de la PrD calcule tous les $Q_{opt}(i, m)$, d'abord pour *i* croissant,

puis pour $m$ croissant (double itérations de l'algorithme ci-dessous).
Ainsi, pour $1 \leq i \leq |S|, 1 \leq m \leq M$, $mat[i][m]$ devient synonyme de $Q_{opt}(i,m)$ et pour calculer la valeur d'une case quelconque $mat[i][m]$, les optima pour toutes les cases depuis $mat[0][0]$ sont déjà calculés [1] [2].

Rappelons que les $v_i$ sont strictement positifs.

On stocke les différentes valeurs de $Q(i,m), m = 0..M, i = 0..|S|$ dans la matrice **mat** [3]. La table $S$ contient les pièces disponibles de valeurs $v_i$ dans l'ordre croissant (la dernière est la plus grosse ; en faisant $i-1$, on les consulte dans l'ordre décroissant).
La valeur recherchée finale est égale à *mat[ |S| ][M]* qui sera la toute dernière case de la matrice (en bas à droite).

Le pseudo-code peut s'écrire de la façon suivante :

```
fonction Monnaie(S,M) :                    S est le stock des pièces, M est le montant (la somme)
    soit mat la matrice d'indices [0, |S|] x [0, M]
    pour s = 1 à |S| faire                  – bornes incluses
        mat[0][s] = infini
    pour s = 1 à |S| faire                  – bornes incluses
        pour m = 0 à M faire                – bornes incluses
            si m = 0 alors
                mat[s][m] = 0
            sinon
                mat[s][m] = min(
                                si m − S_{s−1} >= 0  alors 1 + mat[s][m − S_{s−1}]  sinon infini
                                si s >= 1            alors mat[s−1][m]              sinon infini
    renvoyer mat[|S|][M]
```

La complexité de l'algorithme en espace est $O(M.|S|)$, c'est-à-dire le nombre d'éléments de la matrice.
Sa complexité en temps est la même, car il suffit d'un nombre constant d'opérations pour calculer la valeur de chaque case de la matrice.

**Proposer une solution Python** de cette méthode. Dans une première étape, développer trouver simplement le nombre minimal de pièces. Puis modifier votre solution pour donner les pièces utilisées.

# IV Solution par un système d'équations

Il suffit de résoudre le système défini ci-dessus (dans la définition du problème) par un solveur quelconque de système linéaires ($x_i$ est le nombre de la pièces $S_i$ utilisées, $M$ le montant) :

$$\text{Minimize} \sum_{i=1}^{|S|} x_i$$

Subject to

$$\sum_{i=1}^{|S|} x_i \, S_i = M$$

$$x_i \geq 0$$

$$S_i > 0$$

Cependant, en terme d'efficacité et disponibilité de ressources embarquées, la solution par la Programmation Dynamique semble mieux adaptée.
On note également que pour résoudre le système ci-dessus, nous avons besoin d'un solveur. Les plus curieux pourraient regarder du côté du : Matlab (commercial), LpSolve, GLPk, MiniZinc, et autres solveurs publics.

# V Travail à rendre

**• Le minimum à rendre :**

1. Rappel : $i$ est un indice et S[i] est une pièce du stock $S$ de valeur $v_i$. On préfère travailler avec les $i$ plutôt qu'avec les $v_i$ qui sont dis-contigües.
2. cf. la fonction $Fib(N) = Fib(N-1) + Fib(N-2)$ et le fait que le calcul de $Fib(N)$ est $O(1)$ si $Fib(N-1)$ et $Fib(N-2)$ sont déjà calculées et disponibles. Si on admet $Fib(0) = Fib(1) = 1$ par hypothèse, $Fib(2)$ n'aura qu'à additionner $Fib(0)$ et $Fib(1)$ déjà disponibles ; $Fib(3)$ d'utiliser $Fib(1)$ et $Fib(2)$, etc. La complexité de $Fib(.)$ devient ainsi linéaire (moyennant $O(N)$ en espace) versus une version récursive collant à la définition de $Fib(.)$ qui sera d'une complexité *exponentielle* quand $N \to \infty$. De plus, si on ne stock que $Fib(i-1)$ et $Fib(i-2), 0 \leq i \leq N$, la complexité en espace descend à $\Theta(2)$.
3. Pour les indices nuls, voir $Q(i,0)$ et $Q(0,m)$ des propriétés de $Q$.

◦ la solution Gloutonne.
◦ La version optimale avec la Programmation Dynamique (PrD), voir section VI-1 page 6.

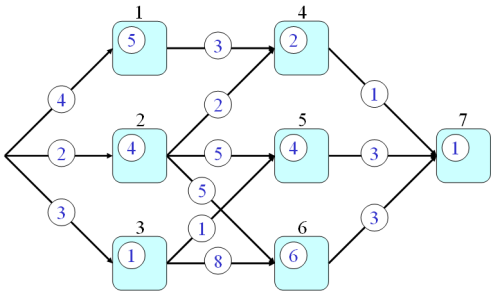
• **Bonus :**
◦ La version optimale avec un arbre (section VI-5 page 9) ou un Dico (section VI-3 page 7). Voir également le principe de la Prd en section VI-1 page 6.
◦ Nous avons vu (voir section II-2 page 3) qu'il était possible d'obtenir une version optimale sans avoir recours au principe PrD. Vous pouvez, si vous le voulez, l'ajouter à la partie bonus que vous rendrez.

# VI Annexes

## VI-1 Principes de la programmation dynamique (PrD)

• La programmation dynamique prend sa source dans le principe d'optimalité (de Richard Bellman).
• Une stratégie optimale pour résoudre un problème (représenté par un espace d'états dans lequel se trouvera un état = une solution optimale) est telle que quelque soit l'état initial et les premières décision prises, les décisions restantes doivent constituer une stratégie optimale par rapport à la toute première décision.
• Par exemple, pour trouver un chemin optimal d'un point de départ à un point d'arrivée (p.ex. le chemin le plus court), ce principe stipule que tout sous-chemin partiel de la solution finale est forcément le chemin le plus court entre les deux étapes reliées, étant donné le départ et l'arrivée (peut être démontré par l'absurde).
• Pour pouvoir appliquer le principe de la PrD, trois conditions / étapes doivent être réalisées :
  ◦ Définir une fonction optimale pour le problème à résoudre (p.ex. la longueur d'un chemin optimal ou le nombre minimal de pièces),
  ◦ Décrire une formulation récursive de la fonction optimale ainsi que la condition initiale,
  ◦ Exprimer la solution au problème en termes de la fonction optimale.

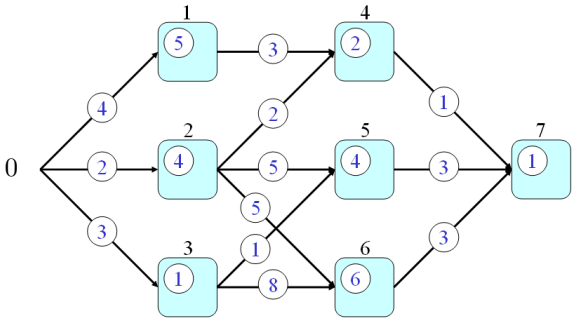
• Exemple : soit le graphe de villes ci-contre où les valeurs sur les arcs $route(x, y)$ sont les durées des trajets entre une ville $x$ et la ville voisine $y$ et où la valeur à l'intérieur de chaque noeud représente le temps de la traversé de chaque ville.




• Pour trouver le chemin le plus court (en terme de durée) entre le départ (à gauche) et le noeud 7, on pose :
  ◦ La *fonction optimale* $t(v)$ représente le minimum du temps pour aller du début au noeud $v$ (qui contiendra le temps de traverser la ville $v$)
  ◦ Pour arriver à la ville $v$ en passant par une des villes la précédents $p_1, ..., p_k$, la formulation récursive de la fonction optimale sera : $t(v) = min\{t(p_1) + route(p_1, v), t(p_2) + route(p_2, v), ..., t(p_k) + route(p_k, v)\}$
  ➛ $t(p_i)$ contient bien entendu le temps pour atteindre $p_i$ depuis le départ.

**Exemple :**

• Dans ce graphe, on aura (en partant de la ville de départ 0) :
  ◦ t(0) = 0
  ◦ t(1) = t(0)+4+5 = 9
  ◦ t(2) = t(0)+2+4 = 6
  ◦ t(3) = t(0)+3+1 = 4
  ◦ t(4) = min(9+3, 6+2)+2 = 10
  ◦ t(5) = min(6+5, 4+1)+4 = 9
  ◦ t(6) = min(6+5, 4+8)+6 = 17
  ◦ t(7) = min(10+1, 9+3, 17+3)+1 = 12




• L'efficacité de la PrD est basée sur le fait d'utiliser des résultats de calculs (partiels) précédents. Après avoir trouvé le chemin le plus court pour arriver à un noeud, cette information est stockée et réutilisée pour calculer le chemin optimal final.

## VI-2 Graphes en Python

• Un Dictionnaire en Python est représenté sous la forme de couples $clef \times valeur$. Par exemple (voir cours également) :

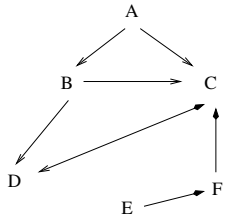
```
params = {"server":"CRI", "database":"master", "uid":"chef", "pwd":"secret"}
params.keys()
# ['server', 'uid', 'database', 'pwd']

params.values()
# ['CRI', 'chef', 'master', 'secret']

params.items()
# [('server', 'CRI'), ('uid', 'chef'), ('database', 'master'), ('pwd', 'secret')]
```

• Pour représenter le graphe suivant à l'aide d'un Dictionnaire en Python, on notera (voir cours également) :

A -> B
A -> C
B -> C
B -> D
C -> D
D -> C
E -> F
F -> C




On peut utiliser

```
graph = {'A': {'B': None, 'C' : None},
         'B': {'C' : None, 'D' : None},
         'C': {'D' : None},
         'D': {'C' : None},
         'E': {'F' : None},
         'F': {'C' : None}
        }
```

• Plusieurs autres représentations sont possibles. Par exemple, le même dictionnaire utilisant des listes donnera (remarquer que la valeur équivalent à `None` dans les listes est la liste vide `[]`) :

```
graph = {'A': ['B', 'C'],
         'B': ['C', 'D'],
         'C': ['D'],
         'D': ['C'],
         'E': ['F'],
         'F': ['C']}
```

• Différentes méthodes de parcours des graphes et arbres sont données dans le support du cours.

## VI-3 Solution par le chemin minimal dans un arbre (par un Dictionnaire)

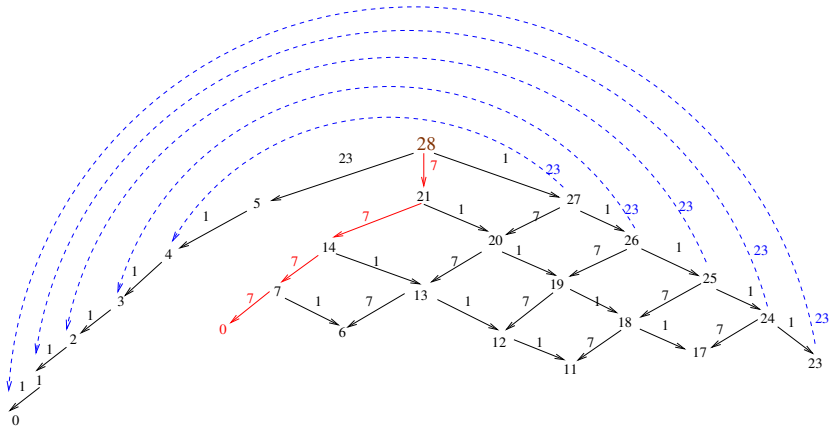
La méthode Gloutonne ne garantie pas que *Q(S,M)* soit minimal comme les tests l'ont montré pour M=28 et S=(1, 7, 23).
Pour ces données, la méthode Gloutonne choisira d'utiliser une pièce de 23 (la plus grande) puis 5 pièces de 1. Donc un total de 6 pièces. Mais, on peut aisément constater que 4 pièces de 7 auraient pu satisfaire la même demande !

**Remarque :** même si ces hypothèses ne correspondent pas à la réalité courante, du point de vue Mathématique, nous ne pouvons pas ne pas en tenir compte !
La méthode Gloutonne utilise un optimum local (choix de la plus grande pièce) qui ne débouche pas forcément sur un optimum global. Cependant, elle est très simple à calculer.

La méthode suivante est plus complexe à mettre en oeuvre mais donne une solution optimale.

Explications :
Pour illustrer cette méthode, on suppose *M=28* et *S=(1,7,23)*. C'est à dire, on a seulement des pièces de 1, 7 et 23 (€s ou cents) en nombre suffisant.
On construit un arbre de recherche (arbre des possibilités) dont la racine est $M$. Chaque noeud de l'arbre représente un montant : les noeuds autres que la racine initiale représentent $M' < M$ une fois qu'on aura utilisé une (seule) des pièces de S (parmi 1,7 ou 23 €s). La pièce utilisée pour aller de $M$ à $M'$ sera la valeur de l'arc reliant ces deux montants.
Cet arbre est donc développé par niveau (*en largeur*, voir cours). On arrête de développer un niveau supplémentaire dès qu'un noeud a atteint 0 auquel cas une solution sera le vecteur des valeurs (des arcs) allant de la racine à ce noeud.

Détails de M=28 :




Dès qu'on atteint 0, on remonte de ce 0 à la racine pour obtenir la solution minimale donnant le nombre minimal d'arcs entre ce 0 et la racine. Dans la figure, on remarque qu'on utilisera 4 pièces de 7c pour avoir 28c.
Par ailleurs, on remarque que le choix initial de 23 (à gauche de l'arbre) laisse le montant *M'=5c* à satisfaire lequel impose le choix de 5 pièces de 1c.
Le principe de l'algorithme correspondant à un parcours en largeur dont la version itérative utilise une *File d'attente* (comme devant un guichet de cinéma, voir cours, les parcours de graphes et arbres) est :

**Algorithme de principe du parcours en largeur :**

```
Fonction Monnaie_graphe
% on suppose qu'il y a un nombre suffisant de chaque pièce/billet dans la tableau S
Entrées: la somme M
Sorties: le vecteur  T et Qopt(S,M)  le nombre de pièces nécessaires
    File F = vide;
    Arbre A contenant un noeud racine dont la valeur = M
    Enfiler(M)                  % la file F contient initialement M
    Répéter
        M'= défiler()
        Pour chaque pièce vi =< M' disponible dans S :
              S'il existe dans l'arbre A un noeud dont la valeur est M' – vi
              Alors établir un arc étiqueté par vi allant de M' à ce noeud
              Sinon
                Créer un nouveau noeud de valeur M' – vi  dans A et lier ce noeud
                              à M' par un arc  étiqueté par vi
                Enfiler ce nouveau noeud
              Fin Si
    Jusqu'à (M' – vi = 0) ou (F = vide)

    Si (F est vide Et M' – vi  /= 0)
    Alors    il y a un problème dans les  calculs! STOP.
    Sinon  Le dernier noeud créer porte la valeur 0
           On remonte de ce noeud à la racine et on comptabilise dans T où Ti = le nombre d'occurrences des
            arcs étiquetés vi  de la racine au  noeud v de valeur 0
           Q(S, M) = ∑_{i=1}^{n} T_i                    % somme des valeurs du vecteur T
    Fin si
Fin Monnaie_graphe
```

☞ Voir en annexe une petite introduction aux graphes. Voir également le cours pour plus de détails.

☞ Contrairement à une implantation de graphes avec adressage dispersé (et pointeurs), l'utilisation d'un dictionnaire de Python (Dict) pour représenter le graphe permet de disposer en permanence de la totalité du graphe (moyennant un cout en espace évidemment !).
Dans l'exemple suivant, on a un graphe dont les noeuds de différents niveaux sont visibles et disponibles.

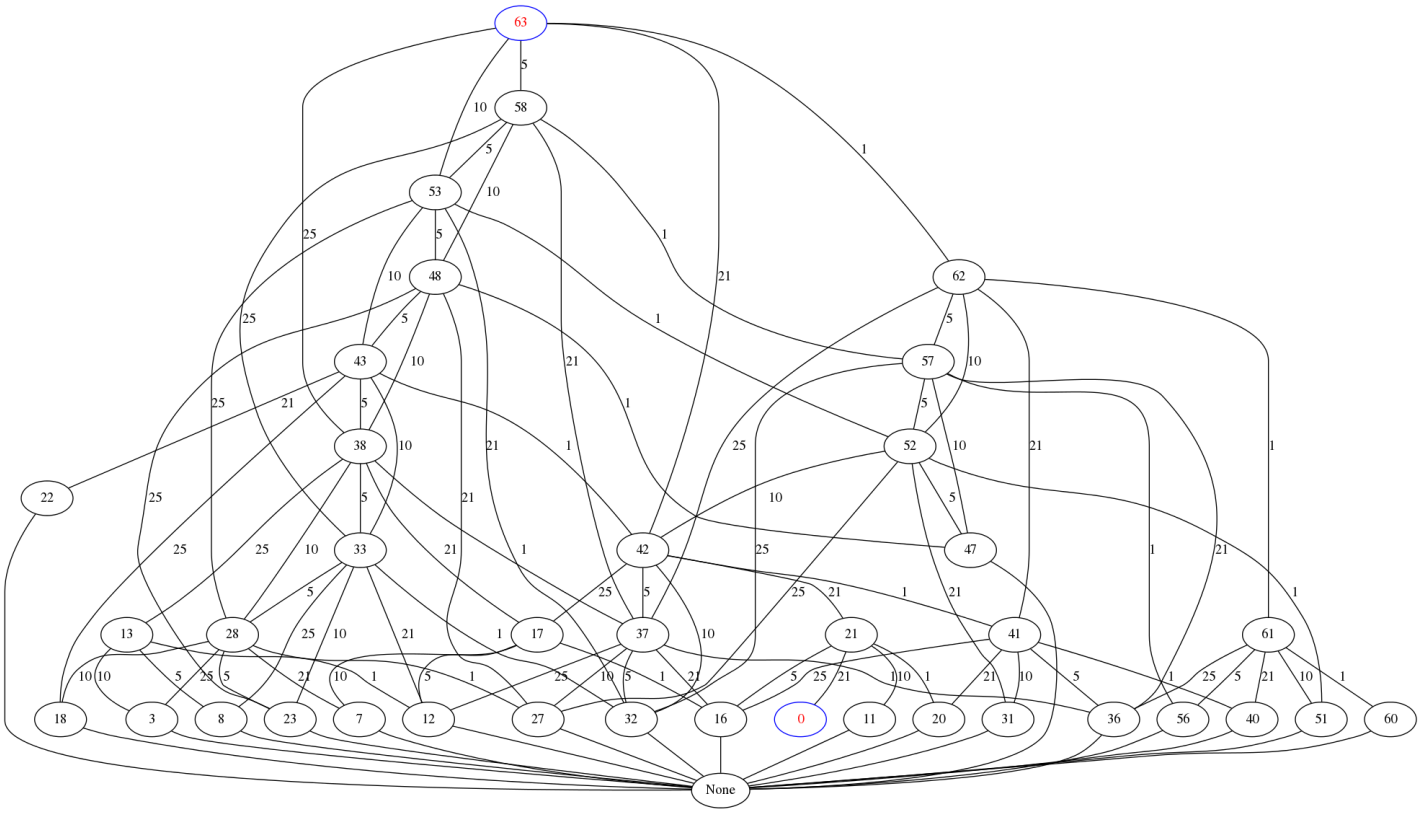
```
graph = { 'A': { 'B': None, 'C' : None},
          'B': { 'C' : None, 'D' : None},
          'C': { 'D' : None},
          'D': { 'C' : None},
          'E': { 'F' : None},
          'F': { 'C' : None}
        }
```

De ce fait, il n'est plus besoin d'une file d'attente (utilisée dans l'algorithme ci-dessous). L'accès à l'ensemble des fils d'un niveau est directement disponible dans un dictionnaire Python. Voir cours pour les implantations alternatives.

**Proposer une solution Python** de cette méthode. Dans une première étape, développer le graphe pour atteindre la feuille zéro puis construisez le chemin de la racine à cette feuille.

## VI-4 Arbre d'un autre exemple

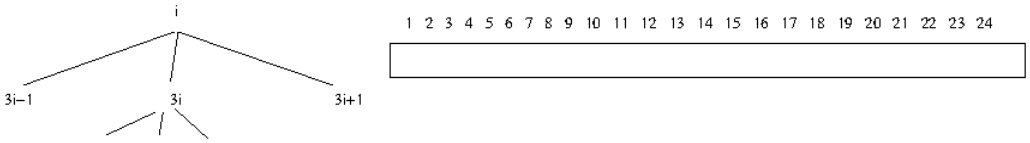
• L'arbre du système *Q(S,M) = Q([1,5,10,21,25], 63)* :




## VI-5 Utilisation d'un arbre ternaire représenté par une liste

• On peut appliquer la solution par un Dictionnaire Python à un arbre **n**-aire où **n** est la taille de S.
• Ci-dessous, on développe le cas de n=3. Voir ci-après pour n=4,5, ...

• Comme pour le BE2 (arbre de la partie "fusion"), il est possible d'utiliser un arbre représenté par une liste plate :




• Pour le cas particulier où on a seulement $k = 3$ pièces de monnaies différentes (l'ensemble $S$ donnant ici une arbre ternaire), la représentation ci-dessus permet de passer :
  ◦ de l'indice `pere` du père aux fils : `3*pere-1`, `3*pere`, `3*pere+1` (`for j in range(k) : 3*pere+j-1`)
  ◦ de l'indice `fils` au père : `pere= fils // 3;if fils % 3 ==2 : pere +=1`
  ◦ La somme initiale (ici 28 à rendre) est à l'indice 1 (l'indice 0 de la liste non utilisé).

☞ La faiblesse de cette méthode est que si l'ensemble $S$ contient $k$ pièces, l'équation ci-dessus devra être changée. Par exemple, pour k=2, les fils seront en $2i, 2i + 1$ (la racine à l'indice 1). Pour k=4 et k5, voir ci-après.

Pour $k > 4$, il faudra modifier l'équation (voir ci-après). Noter que Python permet de passer une fonction en paramètre et on pourra donc préparer et transmettre une fonction qui permet de retrouver les fils / père en fonction de $k$ à nos calculs.

• Pour plus de souplesse, on peut paramétrer la solution par :
  ◦ $k$ le nombre de pièces (taille de $S$) et le fait de savoir si pour cette valeur de $k$, l'emplacement d'indice 0 est utilisé ou pas (pour k=4, il est utilisé mais pas pour $k = 2, 3, 5$)
  ◦ Une fonction donnée en argument d'appel qui permet de calculer les indices des fils à partir de l'indice du père. Cette fonction donnera un couple si $k = 2$, un triplet si $k = 3$, un quadruplet si $k = 4$, etc.

◦ Une fonction donnée en argument d'appel qui permet de calculer l'indice du père à partir de l'indice d'un fils. Cette fonction accepte un entier et renvoie un entier.

➙ Par exemple, pour $k = 3$

```
pere2fils_k_is_3 = lambda p : (3*p+j-1 for j in range(3))
list(pere2fils_k_is_3(2)) # [5, 6, 7]
list(pere2fils_k_is_3(21)) # [62, 63, 64]

fils2pere_k_is_3 = lambda f : f//3+1  if f % 3 == 2 else f//3
fils2pere_k_is_3(37) # 12
fils2pere_k_is_3(21) #  7
fils2pere_k_is_3(20) # 7
```

• Pour $k = 4$ et l'indice $pere$ donné (cf. BE2), le premier fils est à l'indice $4pere + 1$, le 2e à l'indice $4pere + 2$, le 3e à $4pere + 3$ et le dernier à $4pere + 4$.

◦ Pour retrouver le parent (`pere`) d'un noeud d'indice *fils* :

```
pere=fils // 4
if fils % 4 == 0 : pere -= 1
```

☞ Rappelons que pour $k = 4$, l'indice 0 est utilisé (pas pour $k = 2$).

• Pour $k = 5$ et l'indice $pere$ donné, les fils sont en indice $5pere - 3, 5pere - 2, 5pere - 1, 5pere, 5pere + 1$

◦ Pour retrouver le parent (`pere`) d'un noeud d'indice *fils* :

```
pere=fils // 5
if fils % 5 > 1 : pere += 1
```

☞ Rappelons que pour certains k, l'indice 0 **n'est pas** utilisé.

• Vous pouvez chercher la relation pere-fils pour $k > 5$.

## VI-6 Arbre avec des tableaux

• Parmi les représentations possible des arbres et graphes, on choisit ici la représentation de l'arbre avec des tableaux.
• Le graphe **G(V,E)** (avec **V** : ensemble des noeuds, **E** : ensemble d'arcs/arêtes) est représenté par ces deux tables.

• Habituellement, la table **V** contient toutes les informations sur les noeuds (p. ex. une ville, sa population, sa superficie, ...).

La table **E** n'a pas besoin de répéter ces informations et se contente de contenir le nom du successeur, ou mieux, l'indice du successeur dans la table **V**.

Les flèches rouges dans la figure ci-contre renvoient vers le noeud successeur dans **V** : ce renvoi se fait via le nom du $successeur_i$ (le même nom que dans **V**) ou via l'indice du successeur dans **V**.

Si une pondération des arcs (arêtes) est présente, chaque case de E sera un couple *(noeud_succ, poids)*.

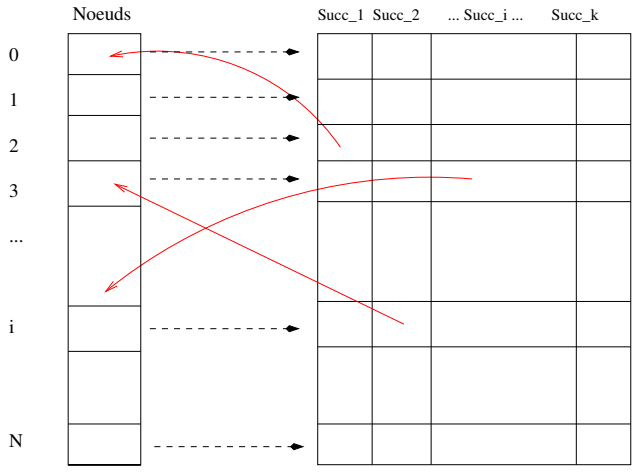



***Table V*** ***Table E (les arcs/aretes)***

• Dans le cas du problème de la monnaie, sachant que l'information d'un noeud est un simple entier (un montant), il est plus simple que chaque successeur contienne également un montant.

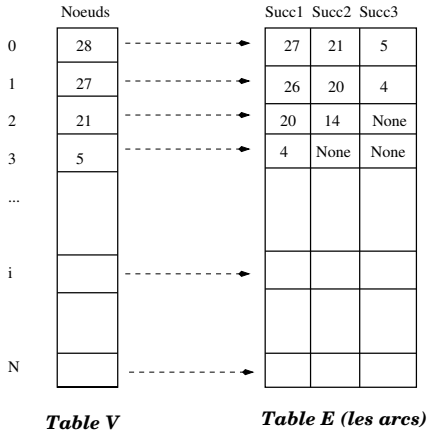
• Par exemple, pour *Q(S,M) = Q([1,7,23], 28)*, on peut avoir :

Les 3 colonnes de la table E correspondent aux 3 valeurs :
28-1, 28-7 et 28-23.

La table E contient 3 colonnes car $|S| = 3$.

La table E étant initialisée par *None*, ces valeurs ne se modifient pas si pour un montant (p. ex. 5), on ne peut pas utiliser certaines les pièces de **S**.

• Notons que cette représentation ne modifie en rien le type du parcours (en profondeur ou en largeur) des arbres/graphes.
• Rappelons que nous utilisons un parcours en largeur.




# Table des Matières