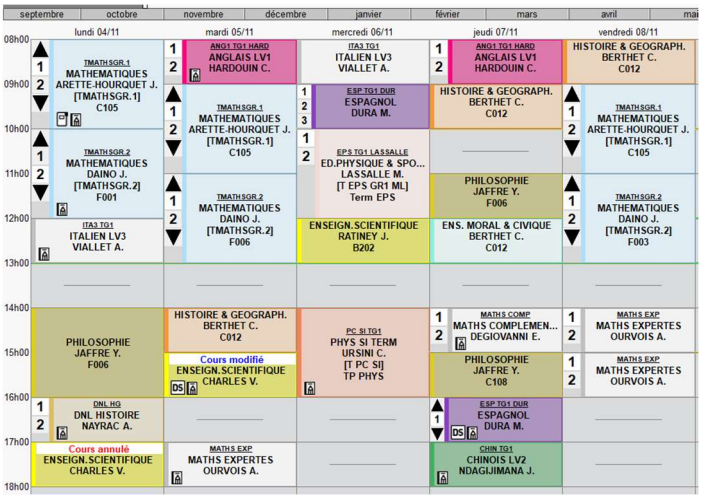
Chapitre 9 - Algorithmes Gloutons

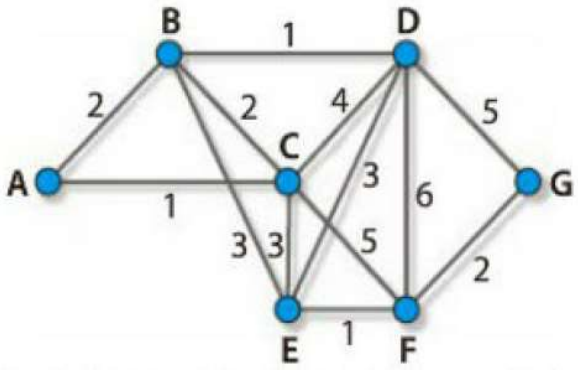
Dans certaines situations, l'informatique apporte une aide précieuse pour prendre des décisions. On peut citer les exemples suivants :

- Création d'un emploi du temps au lycée Branly : Le lycée compte 120 salles de cours, 61 classes. Les cours sont dispensés du lundi au vendredi, sur une plage horaire 8h – 18h, avec 1 heure pour la restauration entre 11h et 13h. L'emploi du temps créé en début d'année a pour vocation d'être optimal. Pour y parvenir, des logiciels comme Pronote proposent des solutions toutes faites. Comment fonctionnent ce type d'algorithmes ? Est-il possible pour un code, de tester toutes les possibilités d'organisation pour ensuite choisir la meilleure ?



- Optimiser le chargement d'un camion :
  Le chargement correct des marchandises dans le camion est un processus complexe, notamment en raison de l'hétérogénéité des emballages, des unités de charge et des types de transport. Les logiciels de gestion d'entrepôt et la transformation digitale sont la solution à cette difficulté, puisqu'ils aident à automatiser une grande partie du processus, en le rendant plus rapide et plus performant. Même question, comment fonctionnent ce type de code ? Est-il possible de tester toutes les possibilités d'organisation, pour ensuite choisir la meilleure ?

- Optimiser un trajet dans une ville : Le graphe ci-contre représente le plan d'une ville. Le sommet A désigne l'emplacement des services techniques. Les sommets B, C, D, E, F et G désignent les emplacements des jardins publics. Une arête représente l'avenue reliant 2 emplacements. Elle est pondérée par le nombre de feux tricolores situés sur le trajet. Des logiciels dits « *du plus court chemin* » permettent, par exemple, d'établir le trajet qui permet de relier A à G avec un minimum de feux tricolores. Comment fonctionnent ce type de code ? Est-il possible de tester toutes les possibilités de trajets, pour ensuite choisir le meilleur ?



Les algorithmes Gloutons qui font l'objet de ce chapitre, permettent d'apporter une première réponse à ce type de problème. C'est une réponse simple, mais non optimale. Ce type d'algorithme reste néanmoins assez efficace. On voit tout cela dans la suite.

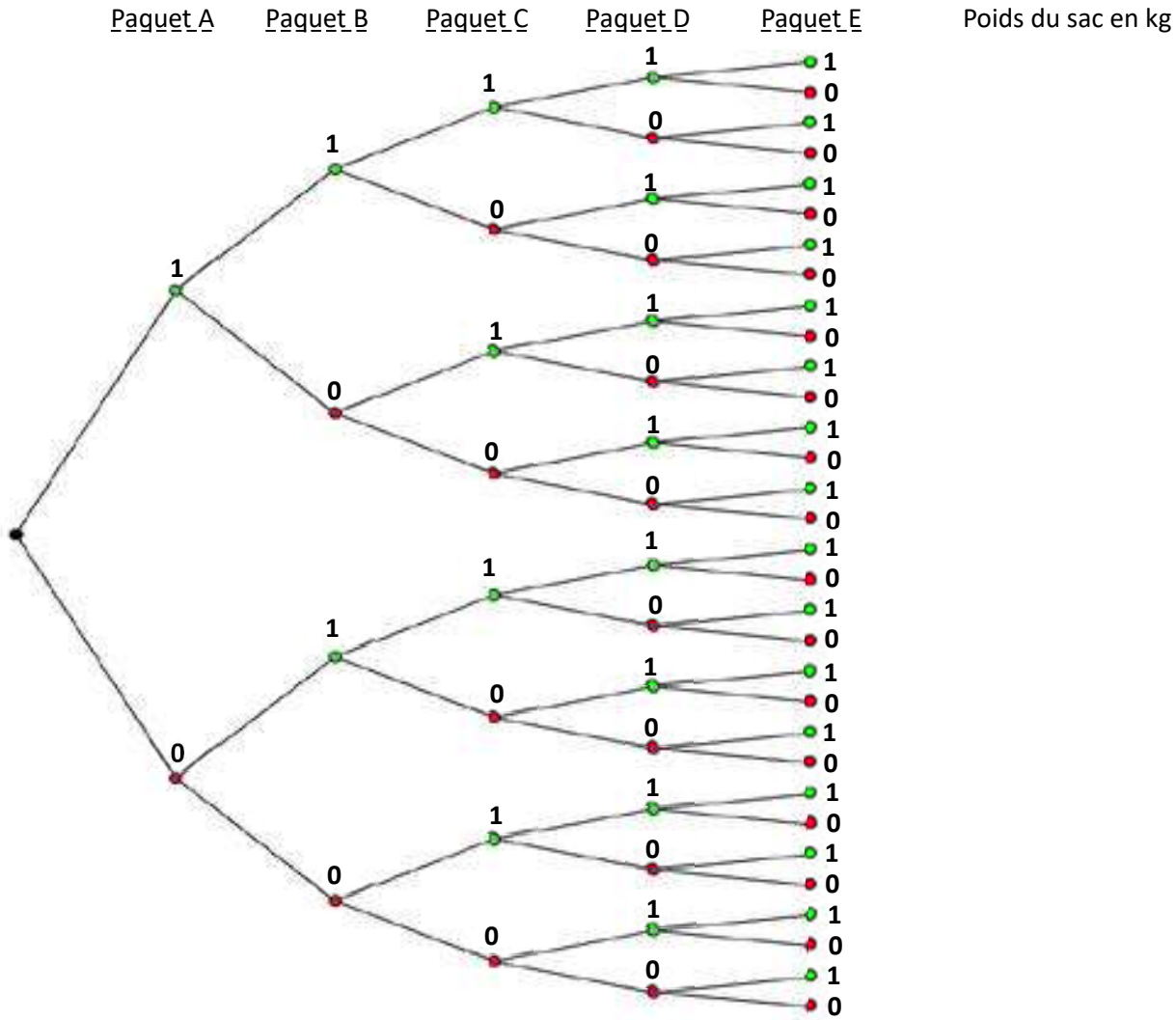
1- Exemple 1 : Probleme du sac a dos

Paquet A 2 kg	*Paquet C 5 kg*	*Paquet E 6 kg*
Paquet B 3 kg	*Paquet D 8 kg*	

On veut remplir un sac à dos avec plusieurs paquets pris parmi ceux A, B, C, D ou E dont les poids sont indiqués ci-contre. Le remplissage est considéré optimal si la somme des poids des paquets introduits s'approche au plus près de la limite des 15 kg admis, sans la dépasser.

a. Methode optimale : On explore toutes les solutions de remplissage possibles

Une technique qui permet d'explorer toutes les possibilités de remplissage est de construire l'arbre ci-dessous, afin de pourvoir calculer le poids du sac s'il contient le paquet repéré (noeud égal à 1) ou s'il ne le contient pas (noeud égal à 0).



Question : Combien de poids de sac différents doit-on calculer ?

Question : Quelle est la solution optimale de remplissage ?

b. METHODE OPTIMALE : CODE PYTHON ASSOCIE

D'un point de vue pratique, pour parcourir toutes les possibilités que présente cet arbre à 5 étages, on crée les 2^5 chaines de caractères données ci-contre. Elles sont composées de 5 chiffres égaux à 1 ou 0, qui repèrent la présence ou non de chacun des 5 paquets.

```
00000
00001
00010
00011
00100
00101
00110
00111
etc ….
11000
11001
11010
11011
11100
11101
11110
11111
```

Pour générer ces chaines de caractère, on peut utiliser la fonction suivante, pour laquelle on donne ci-dessous un exemple d'exécution :

```
def conversion(nDec,taille) :
    nBin = bin(nDec)
    nBin = nBin[2:]
    for i in range(taille-len(nBin)) :
        nBin = "0"+nBin
    return nBin
```

```
>>> conversion(14,7)
'0001110'
```

Pour ensuite calculer les poids pour toutes ces combinaisons, on peut utiliser la fonction donnée ci-contre, incomplète pour l'instant et pour laquelle on donne un exemple d'exécution ci-dessous avec seulement 2 paquets, l'un de 6 kg, l'autre de 8 kg.

```
def optimal(l) :
    n = len(l)
    sac = []
    for i in range(2**n) :
        nBin = conversion(i,n)
        sac.append([nBin,None])
    return sac
```

```
>>> optimal([6,8])
[['00', None], ['01', None], ['10', None], ['11', None]]
```

Question : Réécrire ci-dessous le code de la fonction *optimal()*, afin que l'exécution précédente donne cette fois-ci :

```
>>> optimal([6,8])
[['00', 0], ['01', 8], ['10', 6], ['11', 14]]
```

```
def optimal(l) :
    n = len(l)
    sac = []
    for i in range(2**n) :
        nBin = conversion(i,n)
```

En explorant toutes les possibilités, on est certain de trouver la solution optimale. Par contre le nombre de cas à traiter peut rapidement devenir prohibitif :

Questions :

⇨ nombre de possibilités à explorer pour un nombre $n = 10$ paquets :

⇨ nombre de possibilités à explorer pour un nombre $n = 50$ paquets :

⇨ nombre de possibilités à explorer pour un nombre $n = 1000$ paquets :

c. ALGORITHME GLOUTON : PRINCIPE

L'algorithme glouton qui fait l'objet de ce chapitre, est une alternative simple pour trouver une solution au problème posé.

Point Cours :

Un algorithme glouton est un algorithme qui effectue à chaque instant, « *le meilleur choix possible sur le moment* », sans retour en arrière ni anticipation des étapes suivantes, dans l'objectif d'atteindre au final un

Ce descriptif généraliste appliqué à notre problème de sac à dos donnera :

« *On remplit le sac en prenant toujours en priorité le paquet le plus lourd que le sac peut encore contenir. On arrête de le remplir lorsqu'il n'est plus possible d'en ajouter.*». « *Le meilleur choix possible sur le moment* » est ici de choisir le paquet le plus lourd.

d. ALGORITHME GLOUTON : CODE PYTHON ASSOCIE

Le code de la fonction *glouton()* donné ci-contre permet d'obtenir une solution de remplissage qui n'est pas optimale, mais qui dans la plupart des cas, fonctionne efficacement.

```
def glouton(l,poidsMax):
    sac = ""
    poids = 0
    for i in range(len(l)-1 , -1 , -1)
        if (poids + l[i]) <= poidsMax :
            sac = "1" + sac
            poids = poids + l[i]
        else : sac = "0" + sac
    return [sac , poids]
```

On constitue la liste `l = [2,3,5,6,8]` qui contient les poids des paquets triés par ordre croissant. L'exécution de la fonction *glouton()* pour cette liste et un poids maximum de sac de 15 kg, donne :

```
>>> glouton([2,3,5,6,8] , 15)
['00011', 14]
```

Pour bien comprendre comment fonctionne ce script, on le fait fonctionner « à la main » :

sac	poids	i	$l[i]$

```
def glouton(l,poidsMax):
    sac = ""
    poids = 0
    for i in range(len(l)-1 , -1 , -1)
        if (poids + l[i]) <= poidsMax :
            sac = "1" + sac
            poids = poids + l[i]
        else : sac = "0" + sac
    return [sac , poids]
```

2- EXEMPLE 2 : PROBLEME DU RENDU DE MONNAIE

Le distributeur « *snack* » ci-contre accepte les pièces de 10cts, 20cts, 50cts, 1€, 2€. Après achat, cette machine rend la monnaie de façon optimale. Un rendu optimum en minimisant le nombre de pièces rendues.

Question : Suite à un achat, la machine doit rendre 1,90 €. Donner le détail de ce rendu :

a. ALGORITHME GLOUTON : CODE PYTHON ASSOCIE

Le code de la fonction *glouton()* incomplet donné ci-dessous permet de retrouver le détail d'un rendu de monnaie. On constitue la liste `l = [0.1 , 0.2 , 0.5 , 1 , 2]` qui contient les valeurs des unités monétaires qui peuvent être rendues par le distributeur, valeurs triées par ordre croissant. L'exécution de cette fonction *glouton()* pour cette liste et pour une somme à rendre de 1.90 € donne :

```
>>> glouton([0.1 , 0.2 , 0.5 , 1 , 2] , 1.90)
[1, 0.5, 0.2, 0.2]
```

Question : Compléter ci-dessous le script de cette fonction glouton() :

```
def glouton(l,aRendre):
    rendu = []
    s = 0
    i = len(l)-1
```

Pour bien comprendre comment fonctionne ce script, on le fait fonctionner « à la main » pour l'exécution :

```
>>> glouton([0.1 , 0.2 , 0.5 , 1 , 2] , 1.90)
[1, 0.5, 0.2, 0.2]
```

N° itération	rendu	s	i	$l[i]$
initialisation				
1ère boucle				
2ième boucle				
3ième boucle				
4ième boucle				
5ième boucle				
6ième boucle				
7ième boucle				
8ième boucle				

Rappel du principe des algorithmes « *glouton* » :

Un algorithme glouton est un algorithme qui effectue à chaque instant, « *le meilleur choix possible sur le moment* », sans retour en arrière ni anticipation des étapes suivantes, dans l'objectif d'atteindre au final un

Pour ce rendu de monnaie, *« le meilleur choix possible sur le moment »* est ici de choisir la valeur monétaire la plus forte.

b. CET ALGORITHME EST-IL TOUJOURS OPTIMAL ?

Le système monétaire composé des pièces de 10cts, 20cts, 50cts, 1€, 2€ est dit canonique. Pour ce type de systèmes, rendre la monnaie en commençant par les grosses coupures donnera toujours la solution optimale. Tous les systèmes monétaires dans le monde sont canoniques.

On peut par contre, imaginer un système monétaire non canonique, par exemple celui composé des pièces de 10 cts, 30 cts, 40 cts, 1 € et 2 €.

Question : Que renvoie l'exécution suivante : `>>> glouton([0.1 , 0.3 , 0.4 , 1 , 2] , 1.60)`

Question : Si une fonction optimal() renvoie la solution optimale de ce rendu de monnaie, quel serait la valeur de ce revoi ? `>>> optimal([0.1 , 0.3 , 0.4 , 1 , 2] , 1.60)`