Exemples de projets en spécialité NSI en terminale

Auteur : Christophe LIGERET

Juin 2020

Extrait du BO

Démarche de projet
Un enseignement d'informatique ne saurait se réduire à une présentation de concepts ou de méthodes sans permettre aux élèves de se les approprier en développant des projets.
Un quart au moins de l'horaire total de la spécialité est réservé à la conception et à l'élaboration de projets conduits par les élèves.
Les projets réalisés par les élèves, sous la conduite du professeur, constituent un apprentissage fondamental tant pour l'appropriation des concepts informatiques que pour l'acquisition de compétences. En classe de première comme en classe terminale, ils peuvent porter sur des problématiques issues d'autres disciplines et ont essentiellement pour but d'imaginer des solutions répondant à un problème ; dans la mesure du possible, il convient de laisser le choix du thème du projet aux élèves. Il peut s'agir d'un approfondissement théorique des concepts étudiés en commun, d'une application à d'autres disciplines telle qu'une simulation d'expérience, d'exploitation de modules liés à l'intelligence artificielle et en particulier à l'apprentissage automatique, d'un travail sur des données socioéconomiques, du développement d'un logiciel de lexicographie, d'un projet autour d'un objet connecté ou d'un robot, de la conception d'une bibliothèque implémentant une structure de données complexe, d'un problème de traitement d'image ou de son, d'une application mobile, par exemple de réalité virtuelle ou augmentée, du développement d'un site Web associé à l'utilisation d'une base de données, de la réalisation d'un interpréteur d'un mini-langage, de la recherche d'itinéraire sur une carte (algorithme A*), d'un programme de jeu de stratégie, etc.
La conduite d'un projet inclut des points d'étape pour faire un bilan avec le professeur, valider des éléments, contrôler l'avancement du projet ou en adapter les objectifs, voire le redéfinir partiellement, afin de maintenir la motivation des élèves.
Les professeurs veillent à ce que les projets restent d'une ambition raisonnable afin de leur permettre d'aboutir.

Dans ce document, il ne sera présenté que la partie I/ exemples d'idées de projets autour de la thématique des télécommunications.

I/ Projets autour des télécommunications

I.1/ Gestion d'un répertoire téléphonique

On construra / gérera un document texte depuis Python. Dans ce document texte, il sera décrit les caractéristiques des différents contacts.

Chaque contact possédera au minimum le champ "nom"

On y rajoutera les champs optionnel : "adresse mail" ; "téléphone fixe" ; "téléphone portable" ; "adresse" et "entreprise"

Il faudra pouvoir mettre à jour le fichier texte : rajouter / modifier / consulter / supprimer un contact

I.1.1/ Version simple du répertoire

Il faudra pouvoir avoir accès aux contacts via leur nom.

Ensuite, le logiciel retournera les différents champs associé au contact.

I.1.2/ Version améliorée du répertoire

On pourra avoir accès aux contacts, non plus uniquement par leur nom mais aussi par leur(s) numéro(s) de téléphone.

I.1.3/ Version complète du répertoire

On pourra intégrer une fonction de recherche des contacts en entrant par exemple :
- les trois premières lettres du nom : le logiciel renverra une liste de contacts à consulter
- les 6 premiers numéros du téléphone
- le nom de l'entreprise

Les contacts trouvés seront affichés dans l'ordre alphabétique.

I.1.4/ Version graphique du répertoire

On pourra développer une interface graphique correspondant aux 3 points ci-dessus.

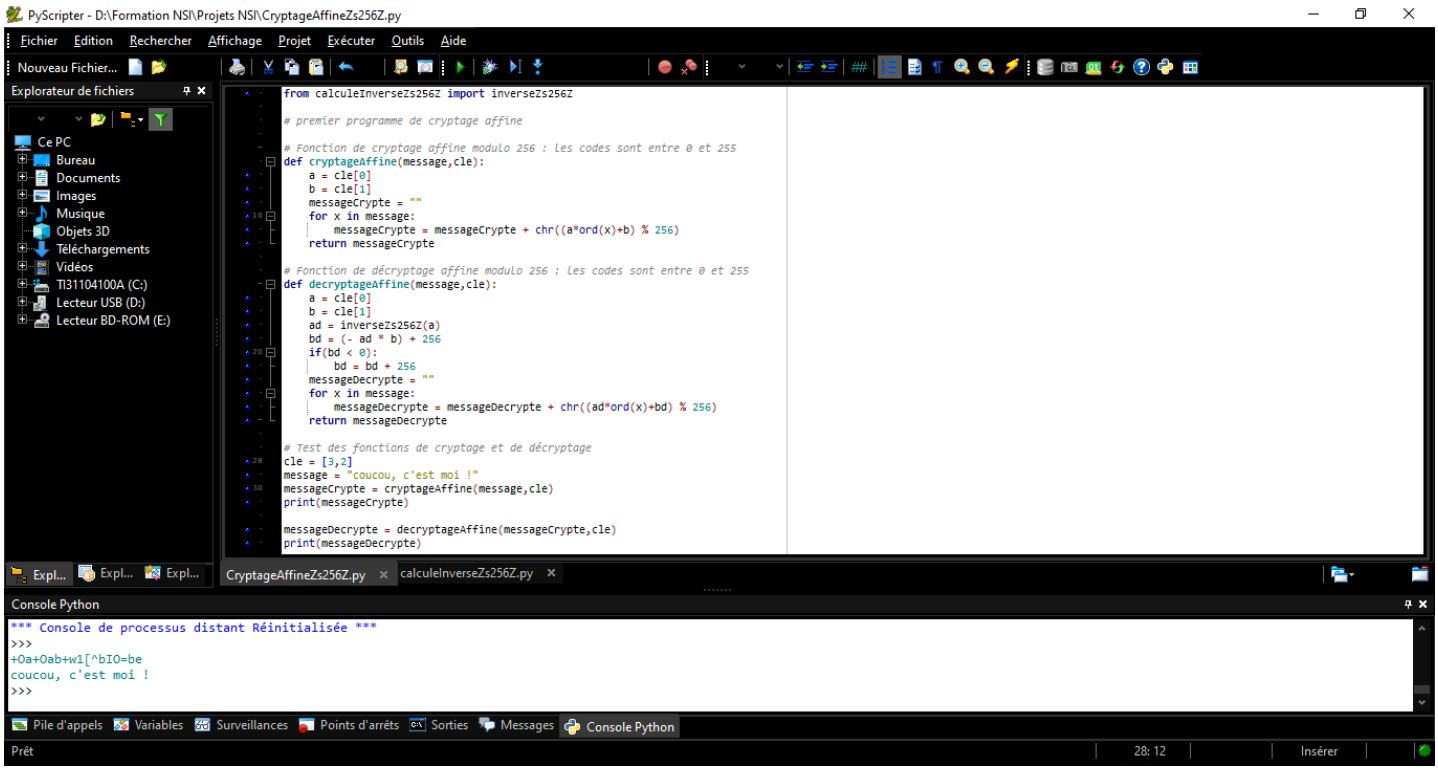
I.2/ Cryptage des données

On pourra proposer des projets autour du chiffrement des données. Ces projets pourront donner lieu au développement de code informatique en langage Python sur une machine informatique ou à l'échange de données entre différents ordinateurs ou microcontrôlleurs : on pourra ainsi par exemple établir une liaison sécurisée entre une télécommande et un robot téléguidé.

I.2.1 / Plateformes utilisées

a) Essais sur un seul ordinateur

Le groupe d'élèves développe des algorithmes et programmes en langage Python pour crypter / décrypter des messages.

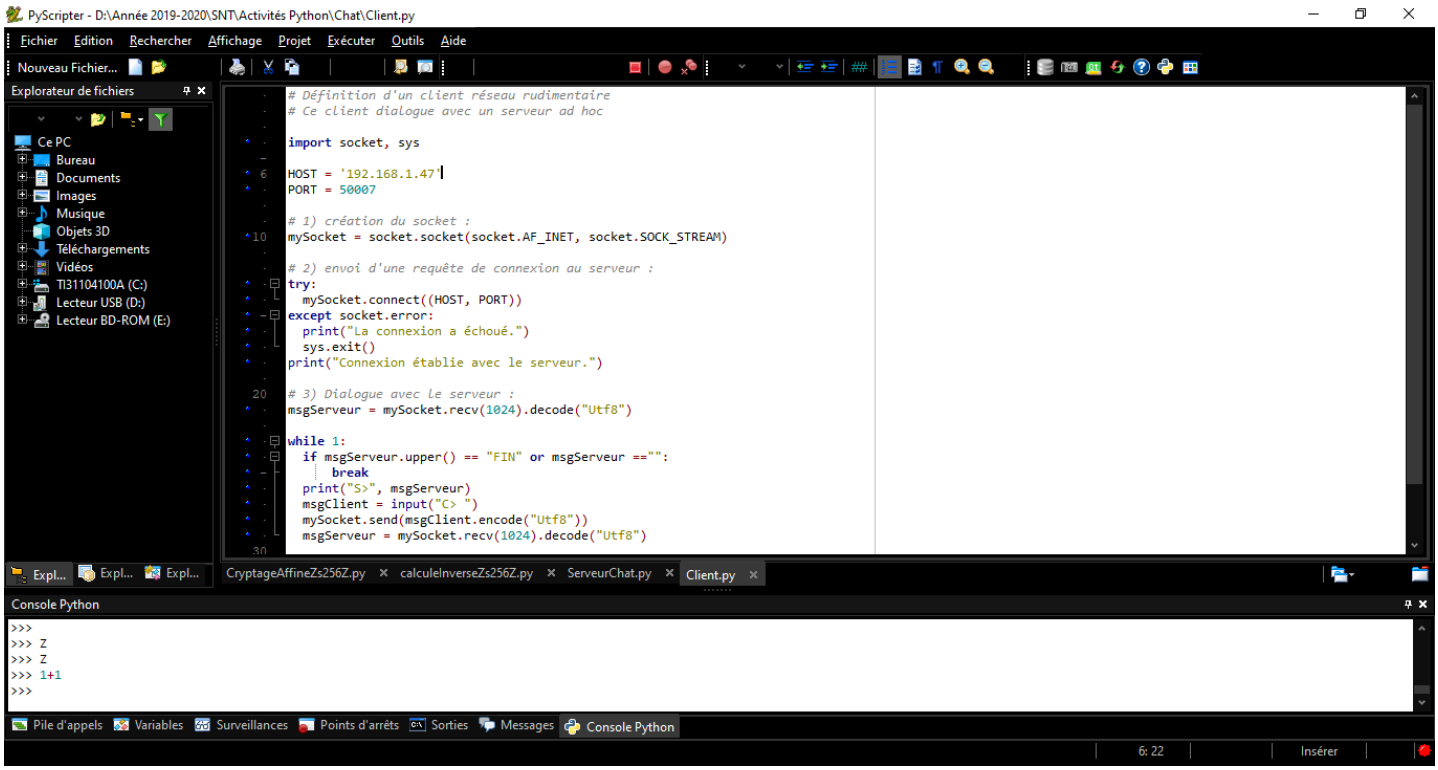


Ci-dessus : exemple de programmes de langage Python pour du cryptage affine

On peut développer une bibliothèque qui permet d'inverser un nombre entier dans $\mathbb{Z}/256\mathbb{Z}$

from calculeInverseZs256Z import inverseZs256Z

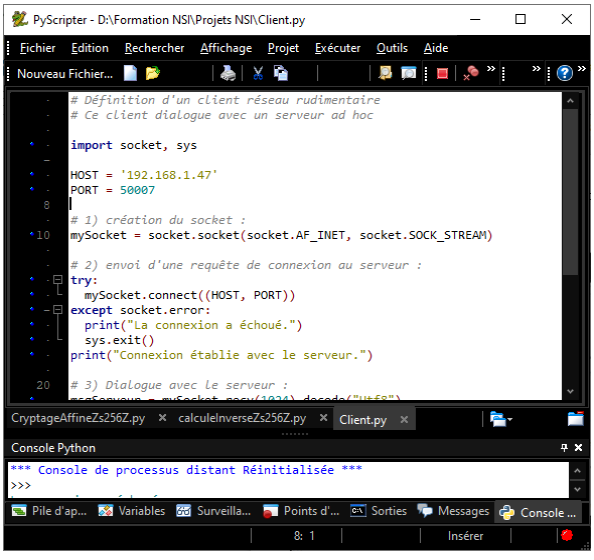
On pourra également développer deux applications en Python : un client et un serveur pour échanger des messages



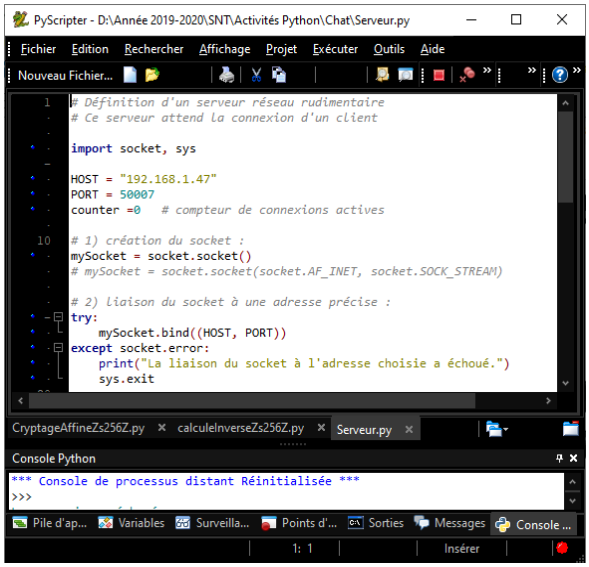
Ci-dessus : programmes Client et Serveur sur le même ordinateur

b) Essais sur deux ordinateurs

On place un ordinateur en Client et un autre ordinateur en serveur et on effectue des échanges de données sécurisées.

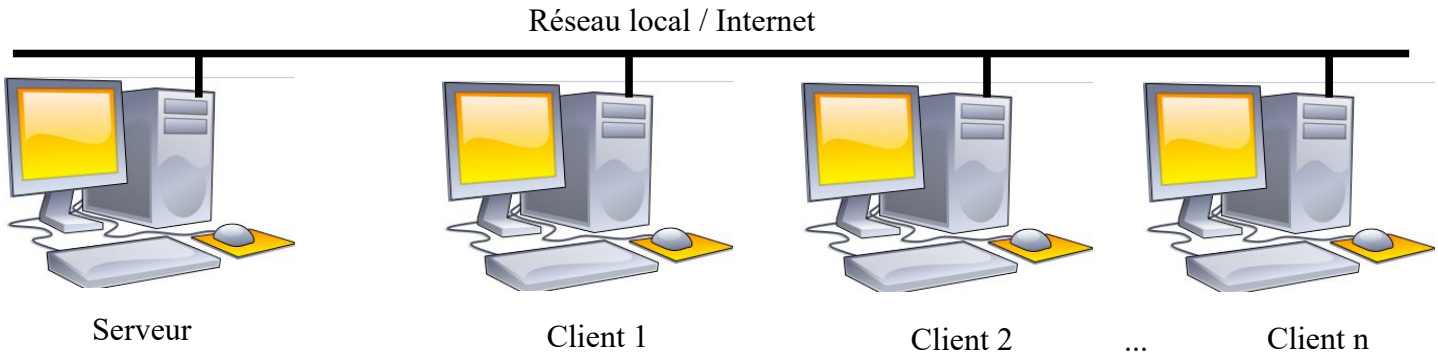


Ordinateur en mode Client avec Cryptage et décryptage



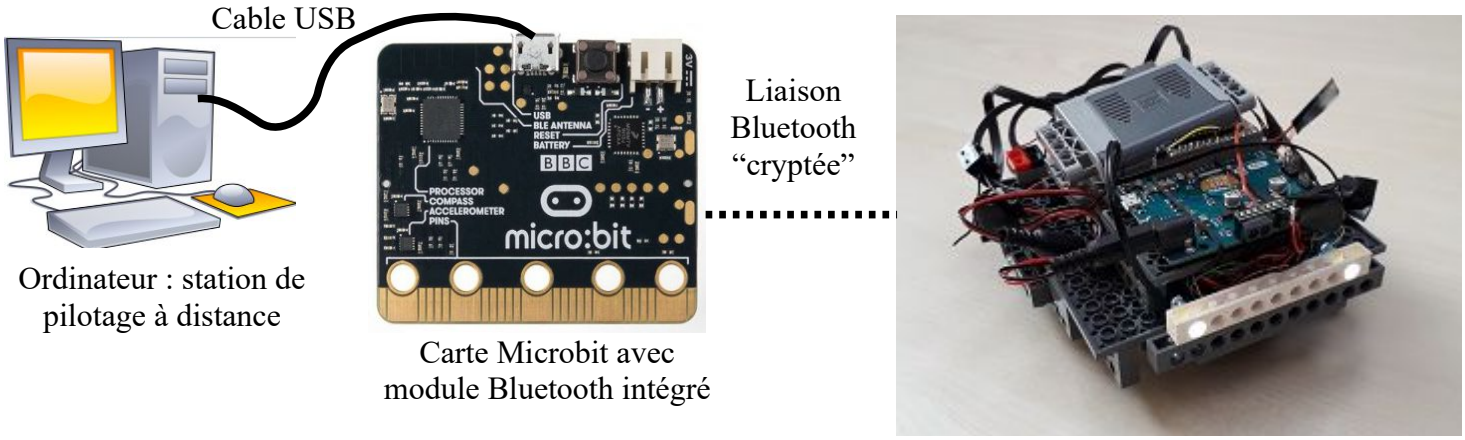
Ordinateur en mode Serveur avec Cryptage et décryptage

c) Essais sur plusieurs ordinateurs

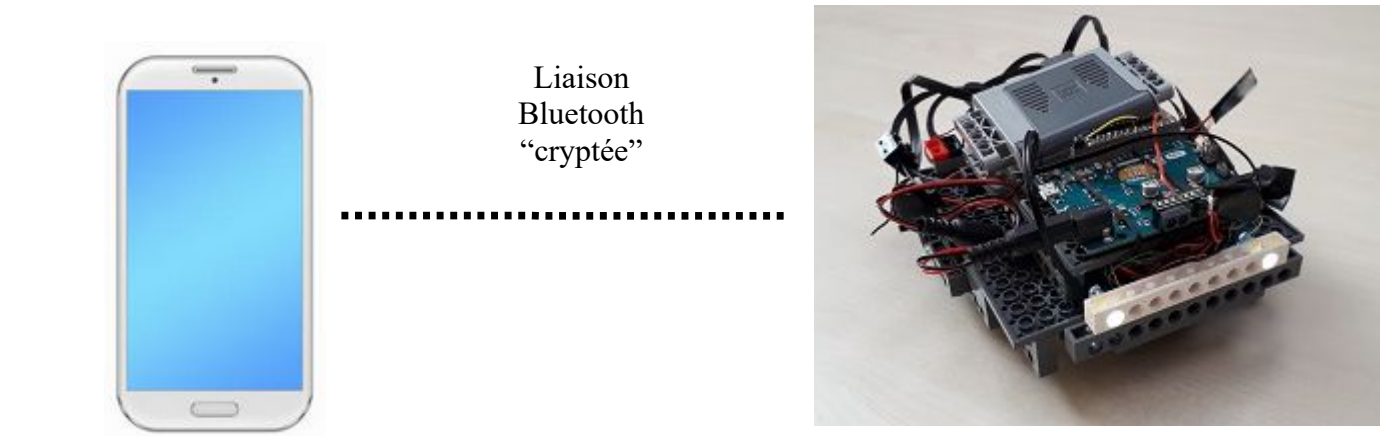


On pourra établir plusieurs connexions entre des clients et un serveur qui distribura des clés de cryptage et / ou des connexions entre des clients via le Serveur (comme un serveur mail)

d) Echange de données entre un ordinateur et un système embarqué



e) Echange de données entre un smartphone et un système embarqué

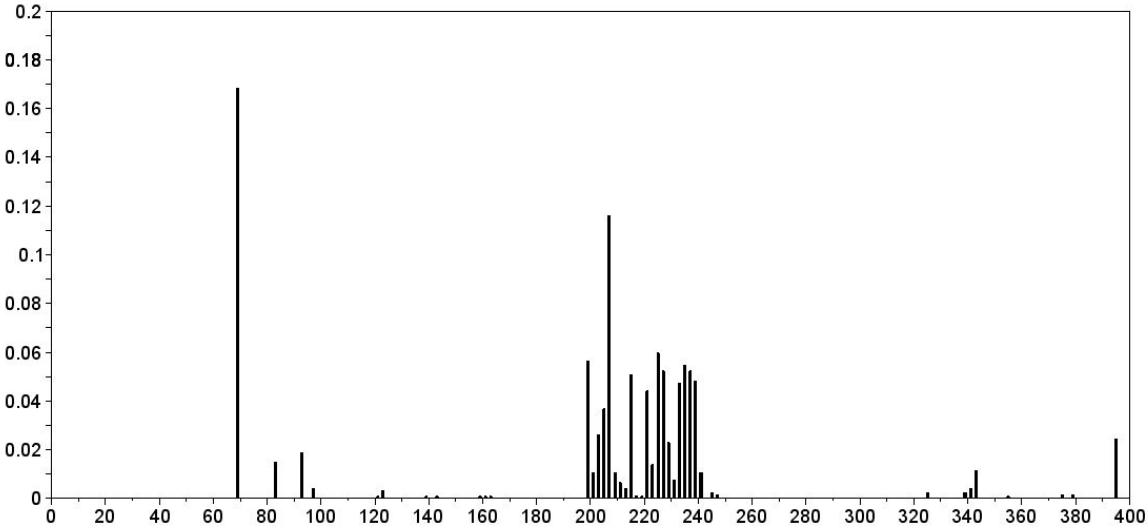


I.2.2 / Protocoles symétriques

Il vous est proposé quelques exemples de protocoles symétriques.
Les protocole symétriques utilisent la même clé pour le chiffrement et le déchiffrement.
Si on connait la clé pour crypter les données, on peut également les décrypter, ce qui pose des problèmes de sécurité car si deux entités ont la même clé de cryptage, elles peuvent chacunes décrypter leurs informations.

a) Cryptage affine

On pourra dans une première phase du projet utiliser un protocole de cryptage symétrique : on pourra démarrer avec un simple système de translation de caractères (code de César), puis un cryptage affine de la forme $a\,x+b$ sur 256 caractères (en considérant l'ensemble $\{0\,,\,1\,,\,2\,,\,...\,,\,255\} = \mathbb{Z}/256\mathbb{Z}$.
On demandera aux élèves de mettre en avant les limites de ce système de cryptage, notamment proposer une méthode simple de cassage de ce cryptage au moyen d'une analyse fréquentielle des caractères cryptés.



Ci-dessus : analyse fréquentielle d'un message crypté : on remarque que les caractères "espace" et "e" sont ceux qui sont les plus fréquents dans la langue française. On peut en déduire que le caractère "espace" est codé par le nombre 69 et le caractère "e" par le nombre 207. Comme le codage est de type affine, deux lettres suffisent à craquer ce type de cyptage : on peut ainsi décrypter le message sans avoir les clés.

b) Chiffrement par transposition

On pourra un cryptage par transposition (permutation) : les élèves devront construire une représentation (par exemple sous forme de liste ou de dictionnaire) de cette table et l'implémenter en langage Python.
Source : https://fr.wikipedia.org/wiki/Chiffrement_par_transposition

c) Chiffre de Vigenère

Il pourra être demandé d'implémenter un cryptage de Vigenère qui s'avère plus robuste que la cryptage affine dans la mesure où un même caractère crypté n'est pas nécessairement représenté par une même code.
Source : https://fr.wikipedia.org/wiki/Chiffre_de_Vigenère

I.2.3 / Protocole asymétrique

Les protocoles asymétriques règlent le problème d'un clé unique qui fait que si on sait crypter des données, on est aussi en mesure de décrypter les données du voisin.

Il y a deux clés distinctes : une pour le chiffrement : la clé publique et une pour le déchiffrement : la clé privé. L'émetteur ne connait que la clé publique pour émettre des messages cryptés. Par contre, le destinataire connait la clé privée pour décrypter les messages.

a) Protocole de Merkle-Hellman

Génération des clés [modifier | modifier le code]

On procède en 3 étapes[3]:

- Choix d'une séquence super-croissante $\{a_1, a_2, \cdots, a_n\}$ et d'un nombre $N, N > a_1 + a_2 + \cdots + a_n$
- Choix d'un nombre $A < N$ tel que $pgcd(A, N) = 1$
- Calcul des $b_i \equiv Aa_i \pmod{N}$

Dès lors, on obtient une clé publique $\{b_1, b_2, \cdots, b_n\}$ (un sac "dur") et une clé privée $(N, A, \{a_1, a_2, \cdots, a_n\})$ (un sac "facile" et deux nombres N et A).

Chiffrement [modifier | modifier le code]

La clé publique permet de chiffrer des messages de longueur n. Considérons un mot fini $w \in \{0,1\}^n$ de longueur n. Alors[3]:

$$c = \sum_{i=1}^{n} w_i b_i$$

est le message chiffré par la clé publique $\{b_1, b_2, \cdots, b_n\}$.

Déchiffrement [modifier | modifier le code]

Considérons le mot chiffré c. Posons $p \equiv A^{-1}c \pmod{N}$[note 1]. On peut alors écrire, l'instance du problème du sac à dos[3]:

$$p = \sum_{i=1}^{n} x_i a_i$$

qui a pour solution $i, x_i = w_i$. Le détenteur de la clef privée $(N, A, \{a_1, a_2, \cdots, a_n\})$ peut calculer calculer p et résoudre en temps polynomial l'instance du sac à dos pour retrouver le message original $(w_1, \ldots, w_n) \in \{0,1\}^n$.

Source : https://fr.wikipedia.org/wiki/Cryptosystème_de_Merkle-Hellman

b) Protocole RSA

Création des clés [modifier | modifier le code]

L'étape de création des clés est à la charge d'Alice. Elle n'intervient pas à chaque chiffrement car les clés peuvent être réutilisées. La difficulté première, que ne règle pas le chiffrement, est que Bob soit bien certain que la clé publique qu'il détient est celle d'Alice. Le renouvellement des clés n'intervient que si la clé privée est compromise, ou par précaution au bout d'un certain temps (qui peut se compter en années).

1. Choisir *p* et *q*, deux nombres premiers distincts ;
2. calculer leur produit *n* = *pq*, appelé *module de chiffrement* ;
3. calculer φ(*n*) = (*p* - 1)(*q* - 1) (c'est la valeur de l'indicatrice d'Euler en *n*) ;
4. choisir un entier naturel *e* premier avec φ(*n*) et strictement inférieur à φ(*n*), appelé *exposant de chiffrement* ;
5. calculer l'entier naturel *d*, inverse de *e* modulo φ(*n*), et strictement inférieur à φ(*n*), appelé *exposant de déchiffrement* ; *d* peut se calculer efficacement par l'algorithme d'Euclide étendu.

Comme *e* est premier avec φ(*n*), d'après le théorème de Bachet-Bézout il existe deux entiers *d* et *k* tels que *ed* = 1 + *k*φ(*n*), c'est-à-dire que *ed* ≡ 1 (mod φ(*n*)) : *e* est bien inversible modulo φ(*n*).

Le couple (*n*, *e*) — ou (*e*, *n*)[3] — est la *clé publique* du chiffrement, alors que sa *clé privée* est[4] le nombre *d*, sachant que l'opération de déchiffrement ne demande que la clef privée *d* et l'entier *n*, connu par la clé publique (la clé privée est parfois aussi définie comme le couple (*d*, *n*)[3] ou le triplet (*p*, *q*, *d*)[5]).

Chiffrement du message [modifier | modifier le code]

Si M est un entier naturel strictement inférieur à *n* représentant un message, alors le message chiffré sera représenté par

$$C \equiv M^e \pmod{n},$$

l'entier naturel *C* étant choisi strictement inférieur à *n*.

Déchiffrement du message [modifier | modifier le code]

Pour déchiffrer *C*, on utilise *d*, l'inverse de *e* modulo (*p* – 1)(*q* – 1), et l'on retrouve le message clair *M* par

$$M \equiv C^d \pmod{n}.$$

Source : https://fr.wikipedia.org/wiki/Chiffrement_RSA

Ce projet est l'occasion de développer des algorithmes autour de l'arithmétique et des nombres premiers : diviseurs, puissances, inverse d'un nombre dans $\mathbb{Z}/p\mathbb{Z}$ où p est un nombre premier. ; test de primalité et génération de grand nombres premiers ; représentation de grands entiers.

Quelques exemples de programmes en Python

```python
# Déterminer si un entier naturel a est multiple d'un entier naturel b
def estMultiple(a,b):
    q = a//b
    r = a%b
    if(r == 0):
        # print(str(a)+" est bien un multiple de "+str(b)+" car "+str(a)+"="+str(b)+"*"+str(q))
        return(1)
    else:
        # print(str(a)+" n'est pas un multiple de "+str(b)+" car "+str(a)+"="+str(b)+"*"+str(q)+"+"+str(r))
        return(0)


# Déterminer le plus grand multiple de a inférieur ou égal à b
def plusGrandMultiple(a,b):
    res = 1
    for i in range(2,(b+1)):
        if(estMultiple(a,i) == 1):
            res = i
    return(res)


# Déterminer si l'entier n est premier
def estPremier(n):
    e = plusGrandMultiple(n,n-1)
    if(e == 1):
        print(str(n)+" est un nombre premier ")
    else:
        print(str(n)+" n'est pas un nombre premier ")

n = int(input("Veuillez entrer la valeur de n svp : "))
estPremier(n)
```

I.3/ Codes détecteurs / correcteurs d'erreur

Lors de transmissions, l'information peut être altérée : par exemple, dans le cadre d'une communication radio, des interférences électrimagnétiques peuvent modifier les informations d'origine : par exemple l'octet 00100101 peut se transformer en l'octet 10100101 ou encore en pire en l'octet 10100001. Dans le premier cas, il y a une seul erreur sur le bit de poids fort (bit n°7 en comptant depuis la droite et en partant de 0). Par contre, dans le deuxième exemple, il y a 3 erreurs : sur le bit n°7 ; le bit n°2 et le bit n°1.
Les codes détecteurs / correcteurs d'erreurs permettent de détecter / corriger certains type d'erreurs.

I.3.1/ Bit de parité

Pour détecter une seule erreur, on rajoute 1 ou plusieurs bits de contrôle en plus de l'octet pour obtenir de la redondance : le système le plus commun est le bit de parité : on rajoute un neuvième bit de manière à obtenir une parité pair ou impaire.
Par exemple, si on a configuré une partité paire : le nombre total de bits à 1 doit être pair : 00100101 **1**
Si par contre, on a configuré une partité impaire : le nombre total de bits à 1 doit être impair : 00100101 **0**
Ce dispositif ne permet que de détecter une seule erreur : si on a un nombre pair d'erreurs (2 ; 4 ; 6 ou 8 bits altérés), on ne détecte rien. De plus, on se limite à détecter des erreurs : on n'est pas capable de retrouver l'octet d'origine

Exemple de programme en Python :

```
# Code détecteur d'erreur : bit de parité

def calculeParite(message,parite):
    somme = 0
    for x in message:
        somme = (somme + int(x)) % 2
    if parite == "pair":
        return message + str(somme)
    elif parite == "impair":
        return message + str((somme+1) % 2)
    elif parite == "aucune":
        return message
    else:
        print("Exception : parité non reconnue")
        return message

message = "00100101"

print(calculeParite(message,"pair"))
print(calculeParite(message,"impair"))
print(calculeParite(message,"aucune"))
print(calculeParite(message,"toto"))

def testeMessage(message,parite):
    messageSansParite = message[0:8]
    messageParitéRecalculee = calculeParite(messageSansParite,parite)
    if messageParitéRecalculee == message:
        print("message correct")
    else:
        print("message altéré")

testeMessage("001001011","pair")
testeMessage("101001011","pair")
testeMessage("101000011","pair")
testeMessage("001001010","impair")
```

I.3.2/ Code correcteur à répétition

Une technique simple pour détecter et corriger des erreurs est de répéter plusieurs fois le même bit.
Par exemple, en répétant 3 fois de suite le même bit, on est capable de reconstruire le message d'origine (lorsqu'il n'y a pas trop d'erreurs).

```
# Code correcteur d'erreur : répétition de bits (x3)

def messageCode(message):
    resultat = ""
    for x in message:
        resultat += str(x)*3
    return resultat

message = "00100101"

print(messageCode(message))

def messageDecode(messageCode):
    resultat = ""
    message0 = messageCode[0:24:3]
    message1 = messageCode[1:24:3]
    message2 = messageCode[2:24:3]
    for i in range(0,8):
        s = int(message0[i]) + int(message1[i]) + int(message2[i])
        if s > 1:
            resultat += "1"
        else:
            resultat += "0"
    return resultat

message = "00100101"
messageCode = messageCode(message)
messageCodeErreur = "010000101000100110010011"
messageDecode = messageDecode(messageCodeErreur)
print(message)
print(messageCode)
print(messageCodeErreur)
```

On remarque que ce système est très lourd car il augmente la taille des messages d'un facteur 3.

I.3.3/ Bit de Parité et matrice

Pour détecter et corriger des erreurs, on peut regrouper les bits en matrice (tableaux) et calculer des bits de parité par lignes et par colonnes

Exemple : ci-dessous bits de parité avec un regroupement de 4 x 4 bits correspondant aux données :
0011 0010 1011 1110

0	0	1	1	***0***
0	0	1	0	***1***
1	0	1	1	***1***
1	1	1	0	***1***
0	***1***	***0***	***0***	

Ci-contre, on transmet 00110 00101 10111 11101 0100

0	0	1	1	***0***	***0***
0	0	1	0	***1***	***1***
1	0	**0**	1	***1***	***0***
1	1	1	0	***1***	***1***
0	***1***	***0***	***0***		
0	***1***	***1***	***0***		

On reçoit 00110 00101 10011 11101 0100

On recalcule les bits de parité et on compare avec les bits de parité dans le message reçu.

On regarde la ligne et la colonne où les bits de parités reçus et calculés sont différents.

L'intersection de la ligne et de la colonne donne le bit erroné.

On corrige et on retrouve ainsi les données d'origine : 0011 0010 1011 1110

I.4/ Routage des données

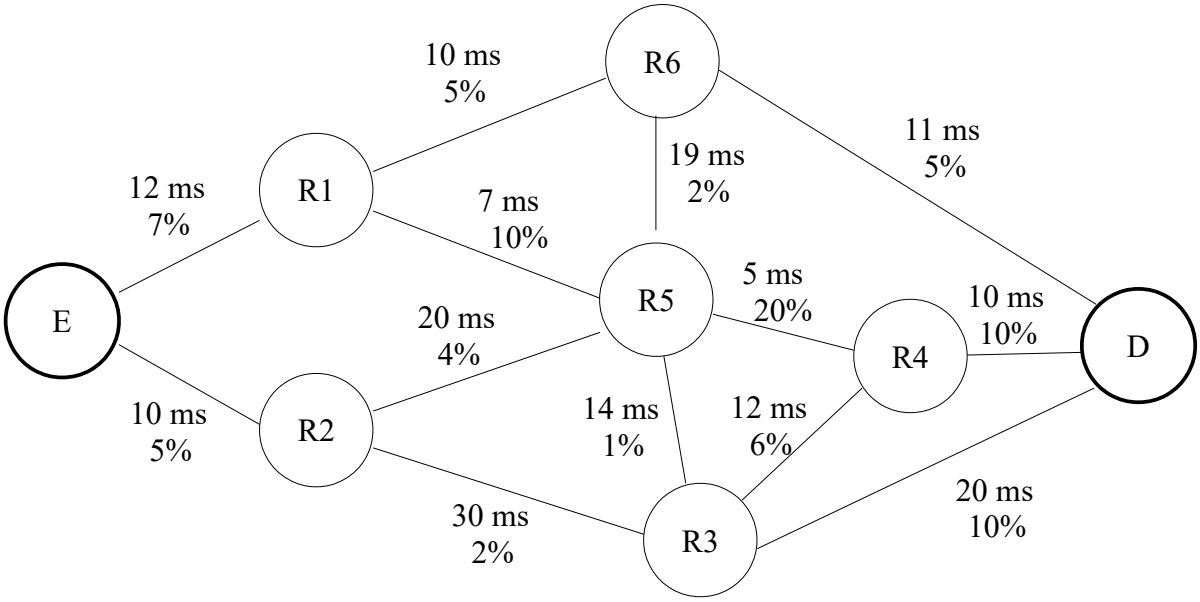
L'idée est de chercher un chemin optimal selon un critère défini pour faire transiter un message d'un émetteur E à un destinataire D au travers un réseau de routeurs.

Pour cela, on pourra utiliser des algorithmes naïfs qui parcourent tous les chemins possibles (sans repasser par un même sommet). Dans ce cas, on étudiera la complexité de tels algorithmes : complexité en n^2.

On pourra également utiliser l'algorithme de Dijkstra dont la complexité est en n ln(n)

Source : https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra

I.4.1/ Exemple général : optimisation selon un critère

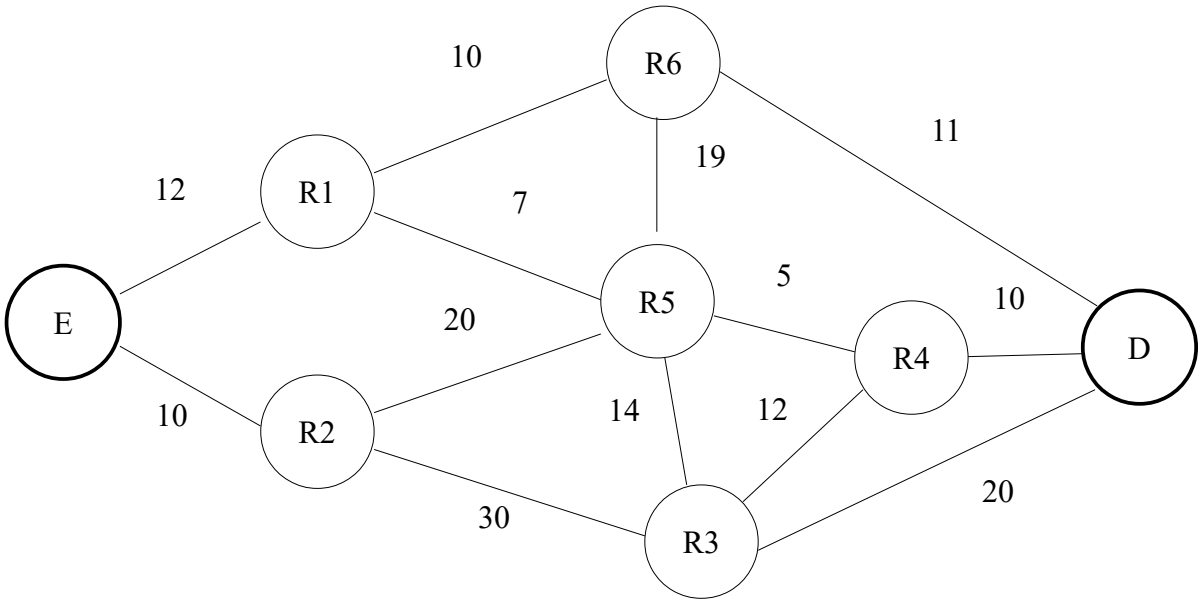


Ci-dessous : réseau Internet avec une émetteur (E) ; un destinataire (D) et des routeurs : Ri

Chaque noeud correspond soit à l'émetteur ; le destinataire ou les routeurs chargés de distribuer les paquets dans le réseau Internet. On peut cherche un chemin qui minimise le temps de parcours ou qui encombre le moins le réseau Internet. Chaque arc correspond à une liaison physique décrite par son temps de parcours et le pourcentage d'encombrement.

a) Temps de transfert minimum

On cherche le chemin qui minimise le temps de parcours. Pour cela, on peut utiliser l'algorithme de Dijkstra.



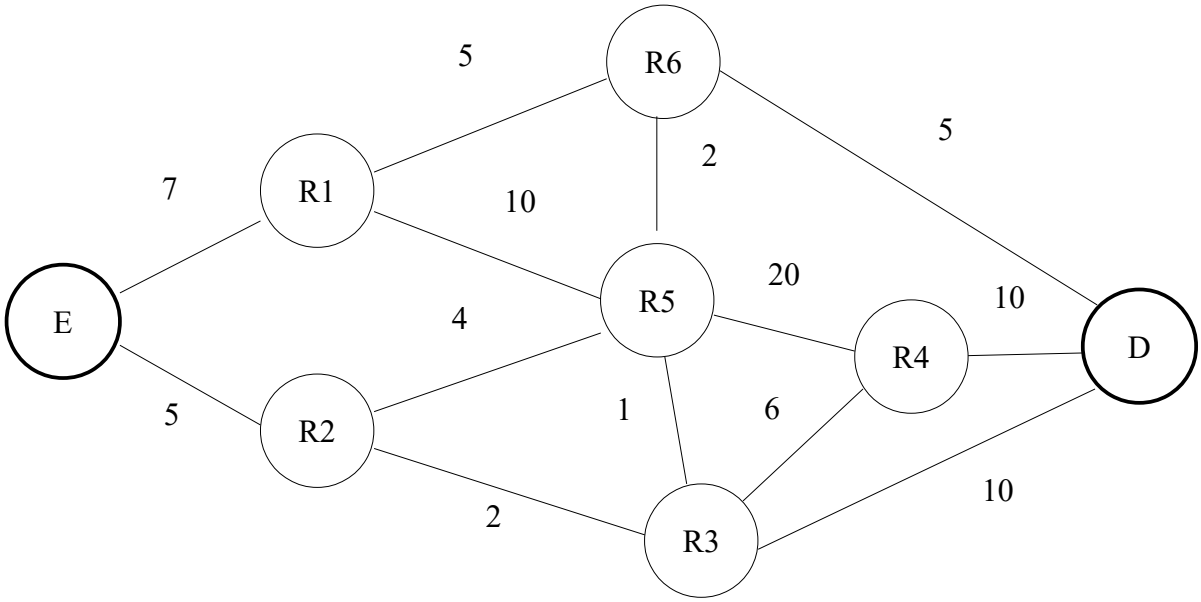
Exemple :

E	R1	R2	R3	R4	R5	R6	D
E-0	E-12	E-10					
		E-10	R2-40		R2-30		
	E-12				R1-19	R1-22	
					R1-19		
			R5-33	R5-24		R5-38	
						R1-22	R6-33
			R4-36	R5-24			R4-34
			R5-33				R3-53
							R6-33

On trouve ainsi que le chemin optimal d'un point de vue du temps de parcours est : E --> R1 --> R6 --> D parcouru en 33 ms.

b) Encombrement minimum

On cherche le chemin qui minimise l'encombrement global du réseau Internet. Pour cela, on peut utiliser de même l'algorithme de Dijkstra.



Exemple :

E	R1	R2	R3	R4	R5	R6	D
E-0	E-7	E-5					
		E-5	R2-7		R2-9		
	E-7				R1-17	R1-12	
			R2-7	R3-13	R3-8		R3-17
				R5-28	R3-8	R5-10	
						R5-10	R6-15
				R3-13			R6-23
							R6-15

On trouve ainsi que le chemin optimal d'un point de vue de l'emcombrement est :
E --> R2 --> R3 --> R5 --> R6 --> D : qui correspond à un total de 15%

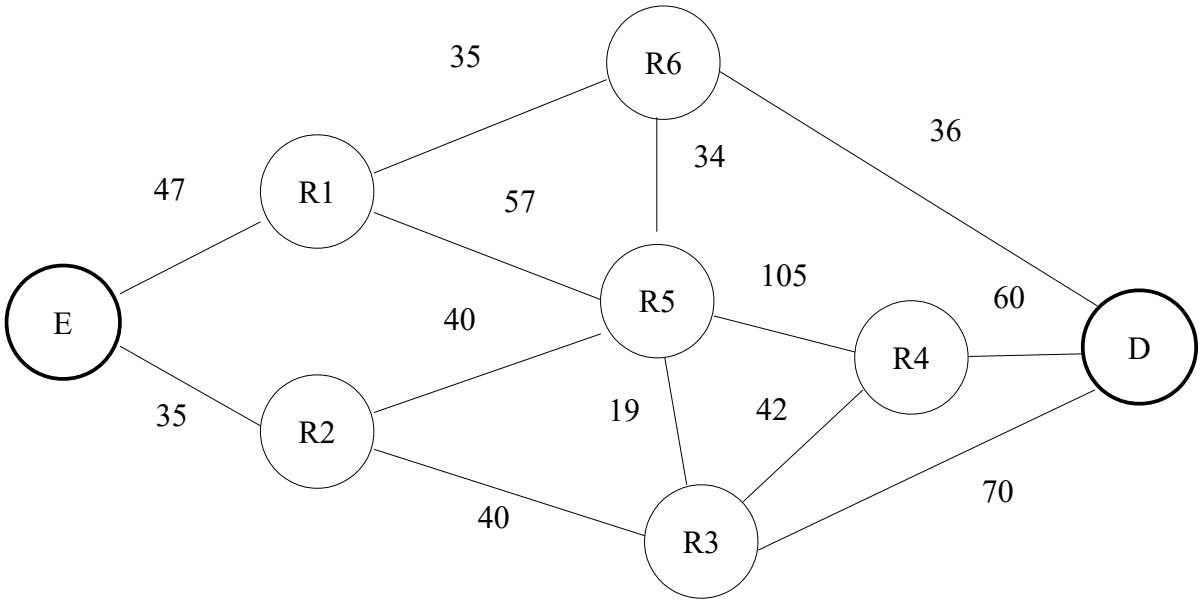
On remarque ici que le chemin optimal est différent de la celui trouvé dans la partie a) : E --> R1 --> R6 --> D car on n'a pas utilisé le même critère d'optimisation.

I.4.2/ Optimisation multicritère

a) Pondération temps + encombrement

On va pondérer chaque arc par un indicateur qui prend en compte le temps de parcours et l'encombrement du réseau : on peut par exemple prendre $\alpha T+\beta E$ où α et β sont des coefficients de pondération à déterminer et où T et E sont respectivement le temps et l'encombrement.

Par exemple, son on prend $\alpha=1$ et $\beta=5$, notre nouveau graphe devient :



Exemple :

E	R1	R2	R3	R4	R5	R6	D
E-0	E-47	E-35					
		E-35	R2-75		R2-75		
	E-47				R1-104	R1-82	
			R2-75	R3-117	R3-94		R3-145
				R5-180	R2-75	R5-109	
						R1-82	R6-118
				R3-117			R4-177
							R6-118

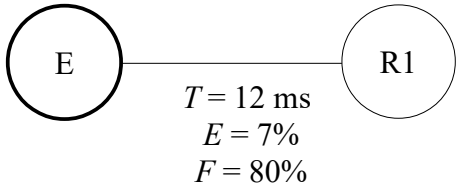
On trouve ainsi que le chemin optimal d'un point de vue du temps de parcours et de l'emcombrement est :
E --> R1 --> R6 --> D : qui correspond à un total de 118

On remarque bien évidemment que ce chemin optimal dépend des coefficients α et β

On peut également construire un nouveau graphe en prenant en compte non seulement le temps de parcours T et l'encombrement du réseau E mais également la fiabilité de la liaison : F

b) Pondération temps + encombrement + fiabilité

Pour faire de l'optimisation multicritère, on peut construire l'indicateur $\alpha T + \beta E + \gamma F$.



T = 12 ms
E = 7%
F = 80%