COURS DE NSI (Numérique et Sciences Informatiques)

Classe : NSIT^le

séquence 5

Mr BANANKO K.

LYCÉE INTERNATIONAL COURS LUMIÈRE

Programmation dynamique et Recherche textuelle

Objectifs :

Contenus	*Capacités attendues*	*Commentaires*
Programmation dynamique.	Utiliser la programmation dynamique pour écrire un algorithme.	Les exemples de l'alignement de séquences ou du rendu de monnaie peuvent être présentés. La discussion sur le coût en mémoire peut être développée
Recherche textuelle.	Étudier l'algorithme de Boyer-Moore pour la recherche d'un motif dans un texte.	L'intérêt du prétraitement du motif est mis en avant. L'étude du coût, difficile, ne peut être exigée.

I- Programmation dynamique

1- Suite de Fibonacci

Revenons sur ce qui a été vu dans le cours consacré à la récursivité. Dans « l'activité 5 » de ce cours, on vous demande d'écrire une fonction récursive qui permet de calculer le ***n ième*** terme de la suite de Fibonacci. Voici normalement ce que vous avez dû obtenir :

```
def fib(n) :
  if n < 2 :
    return n
  else :
    return fib(n-1)+fib(n-2)
```

Ce code, d'une grande simplicité, est malheureusement très inefficace

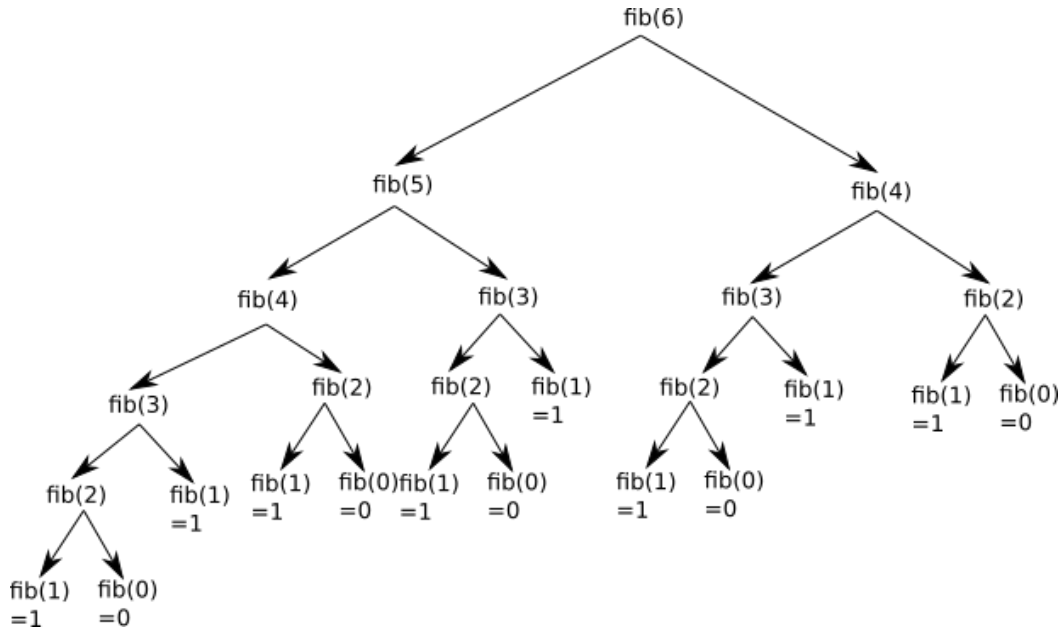
Exemple

Mesurer le temps de calcul de fib(40).

```
import time

def fib(n):
    if n <2 :
        return n
    else :
        return fib(n-1) + fib(n-2)

t0 = time.time()
fibo(40)
print(time.time() - t0)
```

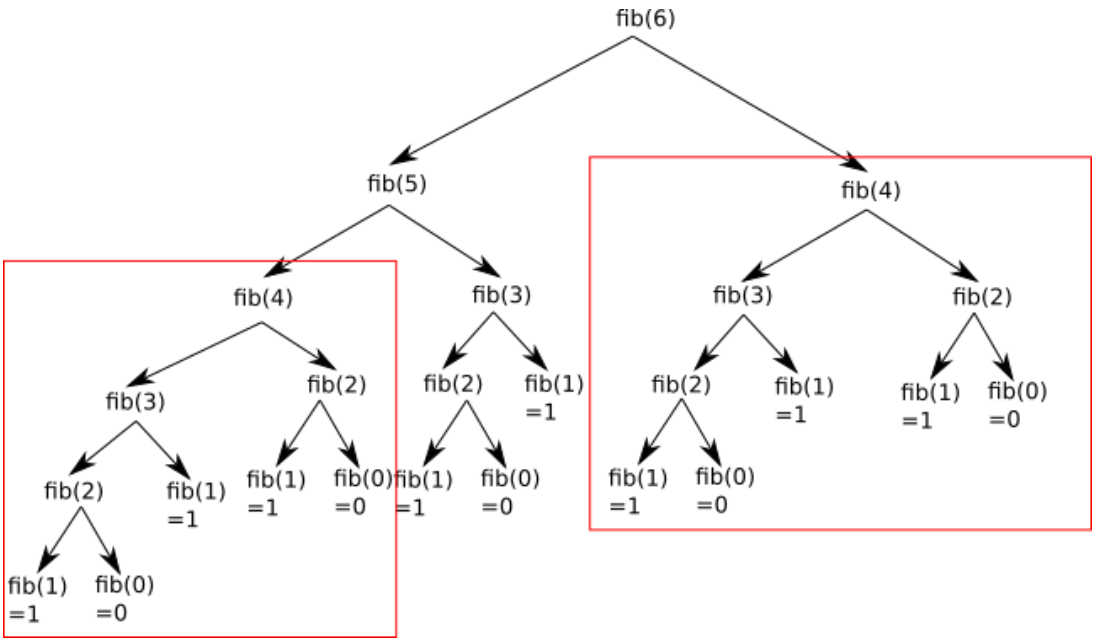
Le temps de calcul est de plusieurs dizaines de secondes, sur une machine récente. C'est très mauvais !

Pour n=6, il est possible d'illustrer le fonctionnement de ce programme avec le schéma ci-dessous :



Vous pouvez constater que l'on a une structure arborescente (***typique dans les algorithmes récursifs***), si on additionne toutes les feuilles de cette structure arborescente (***fib(1)=1 et fib(0)=0***), on retrouve bien 8.

En observant attentivement le schéma ci-dessus, vous avez remarqué que de nombreux calculs sont inutiles, car effectué 2 fois : par exemple on retrouve le calcul de ***fib(4)*** à 2 endroits (*en haut à droite et un peu plus bas à gauche*) :



On pourrait donc grandement simplifier le calcul en calculant une fois pour toutes ***fib(4),*** en "***mémorisant***" le résultat et en le réutilisant quand nécessaire :

Activité 1

Après avoir étudié attentivement le programme ci-dessous, testez-le :

```
def fib_mem(n):
  mem = [0]*(n+1)  #permet de créer un tableau contenant n+1 zéro
  return fib_mem_c(n,mem)


def fib_mem_c(n,m):
  if n==0 or n==1:
    m[n]=n
    return n
  elif m[n]>0:
    return m[n]
  else:
    m[n]=fib_mem_c(n-1,m) + fib_mem_c(n-2,m)
    return m[n]
```

En cas de difficultés, n'hésitez pas à poser des questions.

Dans le cas qui nous intéresse, on peut légitimement s'interroger sur le bénéfice de cette opération de "***mémorisation***", mais pour des valeurs de **n** beaucoup plus élevées, la question ne se pose même pas, le gain en termes de performance (***temps de calcul***) est évident. Pour des valeurs **n** très élevées, dans le cas du programme récursif "***classique***" (n'utilisant pas la "***mémorisation***"), on peut même se retrouver avec un programme qui "***plante***" à cause du trop grand nombre d'appels récursifs.

En réfléchissant un peu sur le cas que nous venons de traiter, nous divisons un problème "***complexe***" (***calcul de fib(6)***) en une multitude de petits problèmes faciles à résoudre (***fib(0) et fib(1)***), puis nous utilisons les résultats obtenus pour les "**petits problèmes**" pour résoudre le problème "***complexe***". Cela devrait vous rappeler la méthode "***diviser pour régner***" !

En faite, ce n'est pas tout à fait cela puisque dans le cas de la méthode "***diviser pour régner***", la "**mémorisation**" des calculs n'est pas prévue. La méthode que nous venons d'utiliser se nomme "***programmation dynamique***".

2- Programmation dynamique

a- Introduction

Comme nous venons de le voir, la programmation dynamique, comme la méthode diviser pour régner, résout des problèmes en combinant des solutions de sous-problèmes. Cette méthode a été introduite au début des années 1950 par Richard Bellman.

Il est important de bien comprendre que "**programmation**" dans "***programmation dynamique***", ne doit pas s'entendre comme "**utilisation d'un langage de programmation**", mais comme synonyme de planification et ordonnancement.

La programmation dynamique s'applique généralement aux problèmes d'optimisation. Nous avons déjà évoqué les problèmes d'optimisation lorsque nous avons étudié les algorithmes gloutons . N'hésitez pas, si nécessaire à vous replonger dans ce cours.

Comme déjà évoqué plus haut, à la différence de la méthode diviser pour régner, la programmation dynamique s'applique quand les sous-problèmes se recoupent, c'est-à-dire lorsque les sous-problèmes ont des problèmes communs *(dans le cas du calcul de fib(6) on doit calculer 2 fois fib(4). Pour calculer fib(4), on doit calculer 4 fois fib(2)...)*.

Un algorithme de programmation dynamique résout chaque sous-sous-problème une seule fois et mémorise sa réponse dans un **tableau** ou dans un dictionnaire, évitant ainsi le recalcul de la solution chaque fois qu'il résout chaque sous-sous-problème (***voir l' "Activité 1"***).

Exemple

1-

```
dict_fib = {0:0, 1:1}
def fib(n):
    if n in dict_fib:
        return …….
    dict_fib[n] = …… + …….
    return dict_fib[n]
```

2- Mesurer le temps de calcul de fib(40) et comparer avec la mesure de l'exemple précédent.

b- Approche descendante

Il existe 2 facettes équivalentes de la programmation dynamique :

- la programmation descendante aussi appelée **top-down** (c'est la méthode que nous avons utilisée ci-dessus pour Fibonacci et que nous allons utiliser immédiatement ci-dessous)
- la programmation ascendante aussi appelée **bottom-up** (que nous étudierons un peu plus tard)

Poursuivons donc notre étude de l'approche descendante avec le problème du rendu de monnaie :

Programmation dynamique et rendu de monnaie

Nous allons maintenant travailler sur un problème d'optimisation déjà rencontré l'année dernière : le problème du rendu de monnaie.

Petit rappel : vous avez à votre disposition un nombre illimité de pièces de 2 cts, 5 cts, 10 cts, 50 cts et 1 euro (100 cts). Vous devez rendre une certaine somme (rendu de monnaie). Le problème est le suivant : "***Quel est le nombre minimum de pièces qui doivent être utilisées pour rendre la monnaie***"

La résolution "***gloutonne***" de ce problème peut être la suivante :

1. on prend la pièce qui a la plus grande valeur (il faut que la valeur de cette pièce soit inférieure ou égale à la somme restant à rendre)

2. on recommence l'opération ci-dessus jusqu'au moment où la somme à rendre est égale à zéro.

Prenons un exemple :

nous avons 1 euro 77 cts à rendre :

- on utilise une pièce de 1 euro (plus grande valeur de pièce inférieure à 1,77 euro), il reste 77 cts à rendre
- on utilise une pièce de 50 cts (plus grande valeur de pièce inférieure à 0,77 euro), il reste 27 cts à rendre
- on utilise une pièce de 10 cts (plus grande valeur de pièce inférieure à 0,27 euro), il reste 17 cts à rendre
- on utilise une pièce de 10 cts (plus grande valeur de pièce inférieure à 0,17 euro), il reste 7 cts à rendre
- on utilise une pièce de 5 cts (plus grande valeur de pièce inférieure à 0,07 euro), il reste 2 cts à rendre
- on utilise une pièce de 2 cts (plus grande valeur de pièce inférieure à 0,02 euro), il reste 0 cts à rendre

L'algorithme se termine en renvoyant 6 (on a dû rendre 6 pièces)

Activité 2

Appliquez l'algorithme glouton vu ci-dessus avec la somme à rendre égale à 11 centimes.

Comme vous l'avez sans doute remarqué, si on applique l'algorithme glouton à cet exemple, nous ne trouvons pas de réponse. En effet :

- on utilise une pièce de 10 cts (plus grande valeur de pièce inférieure à 11 centimes), il reste 1 cts à rendre
- il n'y a pas de pièce de 1 cts => l'algorithme est "bloqué"

Cet exemple marque une caractéristique importante des algorithmes glouton : une fois qu'une "**décision**" a été prise, on ne revient pas "**en arrière**" (on a choisi la pièce de 10 cts, même si cela nous conduit dans une "***impasse***").

Rappel : dans certains cas, un algorithme glouton trouvera une solution, mais cette dernière ne sera pas "***une des meilleures solutions possible***" (une solution optimale), voir le cours sur les algorithmes glouton.

Évidemment, le fait que notre algorithme glouton ne soit pas "**capable**" de trouver une solution ne signifie pas qu'il n'existe pas de solution...en effet, il suffit de prendre 1 pièce de 5 cts et 3 pièces de 2 cts pour arriver à 11 cts. Recherchons un algorithme qui nous permettrait de trouver une solution optimale, quelle que soit la situation.

Afin de mettre au point un algorithme, essayons de trouver une relation de récurrence :

Soit X la somme à rendre, on notera Nb(X) le nombre minimum de pièces à rendre. Nous allons nous poser la question suivante : Si je suis capable de rendre X avec Nb(X) pièces, quelle somme suis-je capable de rendre avec 1+Nb(X) pièces ?
Si j'ai à ma disposition la liste de pièces suivante : p1, p2, p3, ..., pn et que je suis capable de rendre X cts, je suis donc aussi capable de rendre :

- X−p1
- X−p2
- X−p3
- ...
- X−pn

(à condition que pi (avec i compris entre 1 et n) soit inférieure ou égale à la somme restant à rendre)

Exemple : si je suis capable de rendre 72 cts et que j'ai à ma disposition des pièces de 2 cts, 5 cts, 10 cts, 50 cts et 1 euro, je peux aussi rendre :

- 72 - 2 = 70 cts
- 72 - 5 = 67 cts
- 72 - 10 = 62 cts
- 72 - 50 = 22 cts

Je ne peux pas utiliser de pièce de 1 euro.

Autrement dit, si Nb(X−pi) (avec i compris entre 1 et n) est le nombre minimal de pièces à rendre pour le montant X−pi, alors Nb(X)=1+Nb(X−pi) est le nombre minimal de pièces à rendre pour un montant X. Nous avons donc la formule de récurrence suivante :

- si X=0 : Nb(X)=0
- **si X>0 : Nb(X)=1+min(Nb(X−pi))** avec $1 \leq i < n$ et $pi \leq X$

Le "**min**" présent dans la formule de récurrence exprime le fait que le nombre de pièces à rendre pour une somme **X−pi** doit être le plus petit possible.

Activité 3

Étudiez attentivement le programme Python suivant :

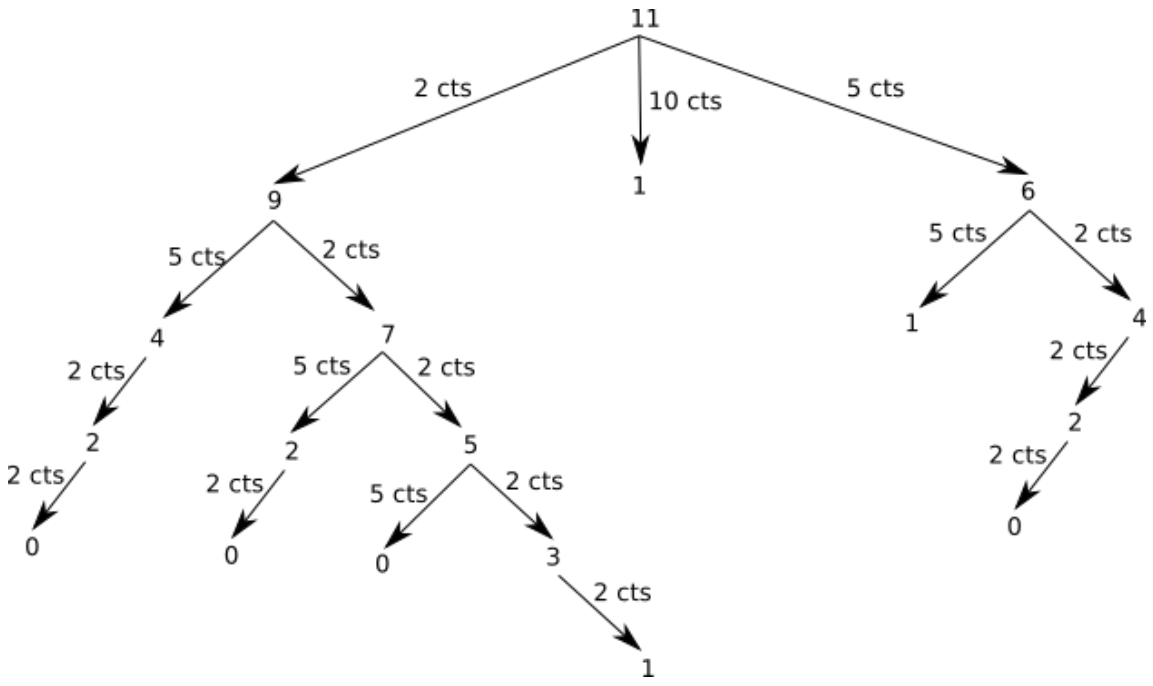
```
def rendu_monnaie_rec(P,X):
  if X==0:
    return 0
  else:
    mini = 1000
  for i in range(len(P)):
    if P[i]<=X:
      nb = 1 + rendu_monnaie_rec(P,X-P[i])
      if nb<mini:
        mini = nb
  return mini


pieces = (2,5,10,50,100)
```

Comme vous l'avez sans doute remarqué, pour être sûr de renvoyer le plus petit nombre de pièces, on attribue dans un premier temps la valeur 1000 à la variable mini (cette valeur 1000 est arbitraire, il faut juste une valeur suffisamment grande : on peut partir du principe que nous ne rencontrerons jamais de cas où il faudra rendre plus de 1000 pièces), ensuite, à chaque appel récursif, on "**sauvegarde"** le plus petit nombre de pièces dans cette variable mini.

Activité 4

Testez le programme de l'"Activité 3" en saisissant dans la console Python "`rendu_monnaie_rec(pieces,11)`" (***on recherche le nombre minimum de pièces à rendre pour une somme de 11 cts. Les pièces disponibles sont : 2 cts, 5 cts, 10 cts, 50 cts, 100 cts (1 euro)***)

Le programme de l' "Activité 3" n'est pas des plus simple à comprendre (même si on retrouve bien la formule de récurrence définit un peu au-dessus), voici un schéma (avec une somme à rendre de 11 centimes) qui vous permettra de mieux comprendre le principe de cet algorithme :



Plusieurs remarques s'imposent :

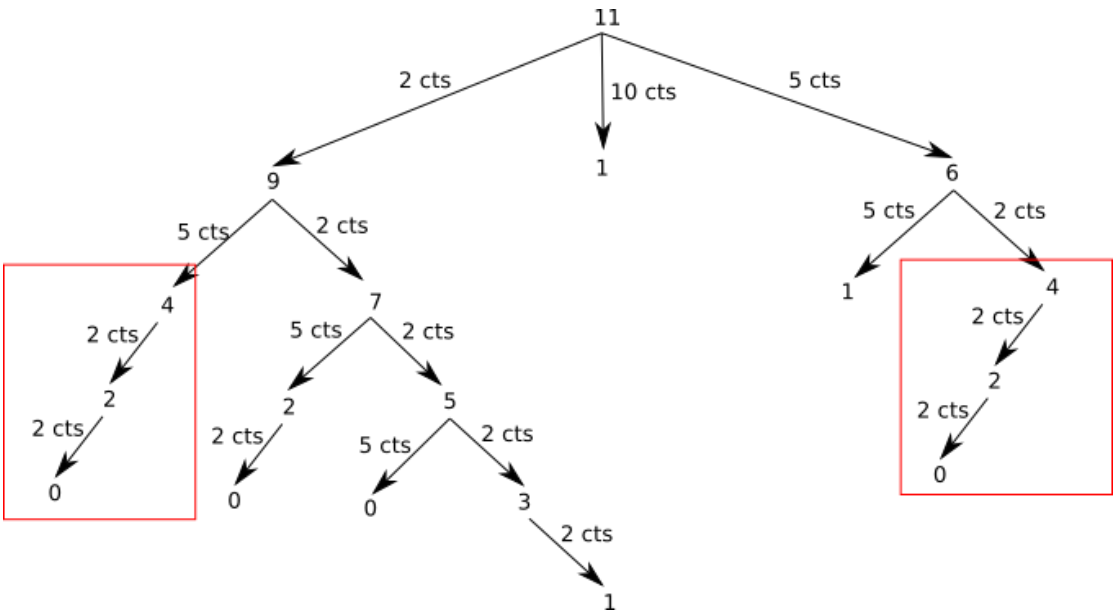
- comme vous pouvez le remarquer sur le schéma, tous les cas sont "**traités**" (quand un algorithme "**traite**" tous les cas possibles, on parle souvent de méthode "**brute force**").
- pour certains cas, on se retrouve dans une "**impasse**" (cas où on termine par un "**1**"), dans cette situation, la fonction renvoie "**1000**" ce qui permet de s'assurer que cette "**solution**" (qui n'en est pas une) ne sera pas "**retenue**".
- la profondeur minimum de l'arbre (avec une feuille 0) est de 4, la solution au problème est donc 4 (il existe plusieurs parcours : (5,2,2,2), (2,5,2,2)... qui donne à chaque fois 4)

Activité 5

Testez le programme de l' "Activité 3" en saisissant dans la console Python "***rendu_monnaie_rec(pieces,171)***" (on recherche le nombre minimum de pièces à rendre pour une somme de 1,71 euro. Les pièces disponibles sont : 2 cts, 5 cts, 10 cts, 50 cts, 100 cts (1 euro)).

Comme vous pouvez le constater le programme ne permet pas d'obtenir une solution, pourquoi ? Parce que les appels récursifs sont trop nombreux, on dépasse la capacité de la pile.

Comme vous avez peut-être déjà dû le remarquer, même dans le cas simple évoqué ci-dessus (11 cts à rendre), nous faisons plusieurs fois exactement le même calcul. Par exemple on retrouve 2 fois la branche qui part de "4" :



Il va donc être possible d'appliquer la même méthode que pour **Fibonacci** : la programmation dynamique.

À noter que dans des cas plus "**difficiles à traiter**" comme 1,71 euro, on va retrouver de nombreuses fois exactement les mêmes calculs, il est donc potentiellement intéressant d'utiliser la programmation dynamique.

Activité 6

Étudiez attentivement le programme Python suivant :

```
def rendu_monnaie_mem(P,X):
  mem = [0]*(X+1)
  return rendu_monnaie_mem_c(P,X,mem)

def rendu_monnaie_mem_c(P,X,m):
  if X==0:
    return 0
  elif m[X]>0:
    return m[X]
  else:
    mini = 1000
    for i in range(len(P)):
      if P[i]<=X:
        nb=1+rendu_monnaie_mem_c(P,X-P[i],m)
        if nb<mini:
          mini = nb
          m[X] = mini
  return mini

pieces = (2,5,10,50,100)
```

Ce programme ressemble beaucoup à programme utiliser pour la suite de **Fibonacci**, il ne devrait donc pas vous poser de problème.

Activité 7

Testez le programme de l' "Activité 6" en saisissant dans la console Python "**`rendu_monnaie_mem(pieces,171)`**" (on recherche le nombre minimum de pièces à rendre pour une somme de 1,71 euro. Les pièces disponibles sont : 2 cts, 5 cts, 10 cts, 50 cts, 100 cts (1 euro)).

Comme vous pouvez le constater, au contraire du cas précédent (programme de l' "Activité 3"), il suffit d'une fraction de seconde pour que le programme basé sur la programmation dynamique donne une réponse correcte.

c- L'approche ascendante

i- Introduction

Comme nous l'avons déjà vu, l'approche descendante utilise une fonction récursive (en mémorisant les valeurs calculées afin d'éviter d'être obligé de les recalculer plus tard et permettant ainsi d'éviter un grand nombre d'appels récursifs). Dans le cas de l'approche ascendante, nous n'utiliserons pas de fonction récursive (on parle de méthode itérative : on utilisera des boucles à la place des appels récursifs).

ii- Fibonacci avec l'approche ascendante

L'idée de base de l'approche ascendante est relativement simple : la relation de récurrence nous montre que pour calculer, par exemple, **fib(10)**, nous avons besoin de calculer **fib(9)** et **fib(8)**, que pour calculer **fib(9)** nous avons besoin de calculer **fib(8)** et **fib(7)...**

Au lieu de recalculer plusieurs fois la même chose, commençons par calculer **fib(2)** (*puisque par définition fib(0) = 0 et fib(1) = 1*), puis calculons **fib(3)** (*nous avons déjà fib(2) et fib(1)*), puis calculons **fib(4)** (*nous avons déjà fib(3) et fib(2)*), puis calculons **fib(5)** (*nous avons déjà fib(4) et fib(3))... afin, calculons fib(10) (nous avons déjà fib(9) et fib(8)*).

Le programme Python permettant de déterminer **fib(n)** en utilisant la méthode ascendante est relativement simple :

```
def fib(n):
    tab = [0]*(n+1)
    tab[1] = 1
    for i in range(2, n+1):
        tab[i] = tab[i-1] + tab[i-2]
    return tab[n]
```

OU

```
def fib(n):
    dict_fib = {}
    dict_fib[0] = 0
    dict_fib[1] = 1
    for k in range(2, n+1):
        dict_fib[k] = dict_fib[k-1] + dict_fib[k-2]
    return dict_fib[n]
```

Vous pouvez constater que ce programme est simple à comprendre :

- pour "**stocker**" les différentes valeurs, on utilise une liste Python que l'on initialise avec n+1 zéro (afin d'avoir un tableau *tab* qui a des indices allant de 0 à n).

- avant le début de la boucle on a le tableau *tab* suivant : `[0,1,0,0,...]`

- la boucle permet de modifier le tableau pour qu'à l'indice *i* on trouve bien la valeur de ***fib(i)***

On constate qu'avec cette approche ascendante, comme pour l'approche decendante, le calcul de ***fib(i)*** n'a été fait qu'une seule fois.

De manière plus générale, cette méthode est basée sur le fait de résoudre des problèmes de petite taille, puis de plus en plus gros, jusqu'au problème final.

iii- Rendu de monnaie avec l'approche ascendante

L'idée est la même que pour **Fibonacci**, on remplit un tableau ***tab***, avec cette fois l'indice *i* qui correspond à la somme à rendre et la valeur située à l'indice *i* qui correspond au nombre minimal de pièces à rendre :

- Pour rendre 0, il faut au minimum 0 pièce : nous aurons donc **tab[0] = 0**.
- Pour rendre 1, il faut au minimum 1 pièce (1 pièce de 1) : nous aurons donc **tab[1] = 1**.
- Pour rendre 2, il faut au minimum 1 pièce (1 pièce de 2) : nous aurons donc **tab[2] = 1**.
- Pour rendre 3, il faut au minimum 2 pièces (1 pièce de 1 et 1 pièce de 2) : nous aurons donc **tab[3] = 2.**
- ...

Si on désire rendre n, il faudra remplir le tableau jusqu'à l'indice **n (tab[n])**

Voici une fonction qui permet de remplir le tableau et donc de trouver une solution au rendu de monnaie :

```
def rendu_monnaie_asc(P,X):
    tab = [1000] * (X+1)
    tab[0] = 0
    for m in range(1, X+1):
        for piece in P:
            if m >= piece:
                tab[m] = min(tab[m], 1 + tab[m-piece])
    if tab[X] != 1000 :
        return tab[X]
```

Quelques remarques sur cette fonction :

- **tab = [1000] * (X+1)** on part du principe qu'il faudra toujours moins de 1000 pièces pour rendre une somme X.
- Nous avons 2 boucles imbriquées : une première boucle qui parcourt le tableau *tab* selon son indice (à chaque indice correspond une somme à rendre : indice 0, on veut rendre 0, indice 1 on veut rendre 1, ...). La deuxième boucle parcourt les pièces disponibles.
- Pour chaque montant, on parcourt toutes les pièces : si la valeur *piece* de la pièce courante est plus petite ou égale au montant courant *m*, *tab[m]* sera égale à ***min(tab[m], 1 + tab[m-piece])***

Prenons un exemple : on désire rendre 6 et nous avons à notre disposition les pièces suivantes : **`[1, 2, 5, 10, 20, 50, 100].`**

Avant le début de la boucle, nous avons le tableau suivant : **`tab = [0, 1000, 1000, 1000, 1000, 1000, 1000]`**

Commençons la boucle :

- m = 1 et piece = 1 : *tab[1] = min(tab[1], 1 + tab[0]) = min(1000, 1+0) = 1*
- m = 1 et piece = 2 : on ne fait rien car piece > m, donc *tab[1] = 1*
- m = 2 et piece = 1 : *tab[2] = min(tab[2], 1 + tab[1]) = min(1000, 1+1) = 2*
- m = 2 et piece = 2 : *tab[2] = min(tab[2], 1 + tab[0]) = min(2, 1+0) = 1*
- m = 2 et piece = 5 : on ne fait rien car piece > m, donc *tab[2] = 1*
- m = 3 et piece = 1 : *tab[3] = min(tab[3], 1 + tab[2]) = min(1000, 1+3) = 3*
- m = 3 et piece = 2 : *tab[3] = min(tab[3], 1 + tab[1]) = min(3, 1+1) = 2*
- m = 3 et piece = 5 : on ne fait rien car piece > m, donc *tab[3] = 2*
- m = 4 et piece = 1 : *tab[4] = min(tab[4], 1 + tab[3]) = min(1000, 1+2) = 3*
- m = 4 et piece = 2 : *tab[4] = min(tab[4], 1 + tab[2]) = min(3, 1+1) = 2*
- m = 4 et piece = 5 : on ne fait rien car piece > m, donc *tab[4] = 2*
- m = 5 et piece = 1 : *tab[5] = min(tab[5], 1 + tab[4]) = min(1000, 1+2) = 3*
- m = 5 et piece = 2 : *tab[5] = min(tab[5], 1 + tab[3]) = min(3, 1+2) = 3*
- m = 5 et piece = 5 : *tab[5] = min(tab[5], 1 + tab[0]) = min(3, 1+0) = 1*
- m = 5 et piece = 10 : on ne fait rien car piece > m, donc *tab[5] = 1*
- m = 6 et piece = 1 : *tab[6] = min(tab[6], 1 + tab[5]) = min(1000, 1+1) = 2*
- m = 6 et piece = 2 : *tab[6] = min(tab[6], 1 + tab[4]) = min(2, 1+2) = 2*
- m = 6 et piece = 5 : *tab[6] = min(tab[6], 1 + tab[1]) = min(2, 1+1) = 2*
- m = 6 et piece = 10 : on ne fait rien car piece > m, donc *tab[6] = 2*

La boucle se termine, et la fonction ***rendu_monnaie_asc([1, 2, 5, 10, 20, 50, 100],6)*** renvoie bien 2

d- Programmation dynamique et mémoire

Comme nous venons de le voir, la programmation dynamique permet de résoudre des problèmes beaucoup plus rapidement, on améliore donc la complexité en temps d'un algorithme en utilisant la programmation dynamique. Cependant, il y a une contrepartie à cette amélioration : l'utilisation de plus de mémoire. En effet, il nous faudra utiliser plus de mémoire pour stocker les résultats des sous-problèmes. La programmation dynamique permet donc d'échanger de l'efficacité en temps contre de l'utilisation de la mémoire.

Exercice

Exercice 1 du sujet 0 - version B 2024

Exercice

Largement inspiré du sujet pratique 24.2 de la BNS 2024.

On considère un tableau non vide de nombre entiers, positifs ou négatifs, et on souhaite déterminer la plus grande somme possible de ses éléments consécutifs.

Par exemple, dans le tableau [1, -2, 3, 10, -4, 7, 2, -5], la plus grande somme est 18 obtenue en additionnant les éléments 3, 10, -4, 7, 2.

Pour cela, on va résoudre le problème par programmation dynamique.

Le problème sera résolu en 2 temps.

Dans un premier temps, si on note tab le tableau considéré et i un indice dans ce tableau, on va chercher à déterminer la plus grande somme possible de ses éléments consécutifs se terminant à l'indice i. Voici la méthode :

Si on connait la plus grande somme possible de ses éléments consécutifs se terminant à l'indice i-1, on peut déterminer la plus grande somme possible de ses éléments consécutifs se terminant à l'indice i :

- soit on obtient une plus grande somme en ajoutant tab[i] à cette somme précédente ;
- soit on commence une nouvelle somme à partir de tab[i].

Dans un deuxième temps, il suffira de parcourir le tableau pour chercher la somme maximale d'éléments consécutifs.

Compléter la fonction **somme_max** ci-dessous qui réalise cet algorithme.

```
 1  def somme_max(tab):
 2      n = len(tab)
 3      sommes_max = [0]*n
 4      sommes_max[0] = tab[0]
 5      # on calcule la plus grande somme se terminant en i
 6      for i in range(1,n):
 7          if ... + ... > ...:
 8              sommes_max[i] = ...
 9          else:
10              sommes_max[i] = ...
11      # on en déduit la plus grande somme de celles-ci
12      maximum = 0
13      for i in range(1, n):
14          if ... > ...:
15              maximum = i
16
17      return sommes_max[...]
```

Exemples :

```
>>> somme_max([1, 2, 3, 4, 5])
15
>> somme_max([1, 2, -3, 4, 5])
9
>>> somme_max([1, 2, -2, 4, 5])
10
>>> somme_max([1, -2, 3, 10, -4, 7, 2, -5])
18
```

II- Recherche textuelle

1- Introduction

Les algorithmes qui permettent de trouver une sous-chaine de caractères dans une chaine de caractères plus grande sont des "**grands classiques**" de l'algorithmique. On parle aussi de recherche d'un motif (sous-chaine) dans un texte. Voici un exemple :

Soit le texte suivant :

"***Les sanglots longs des violons de l'automne blessent mon cœur d'une langueur monotone. Tout suffocant et blême, quand sonne l'heure, je me souviens des jours anciens et je pleure***."

Question : le motif "***vio***" est-il présent dans le texte ci-dessus, si oui, en quelle(s) position(s) ? (la numérotation d'une chaine de caractères commence à zéro et les espaces sont considérés comme des caractères)

Réponse : on trouve le motif "vio" en position 23

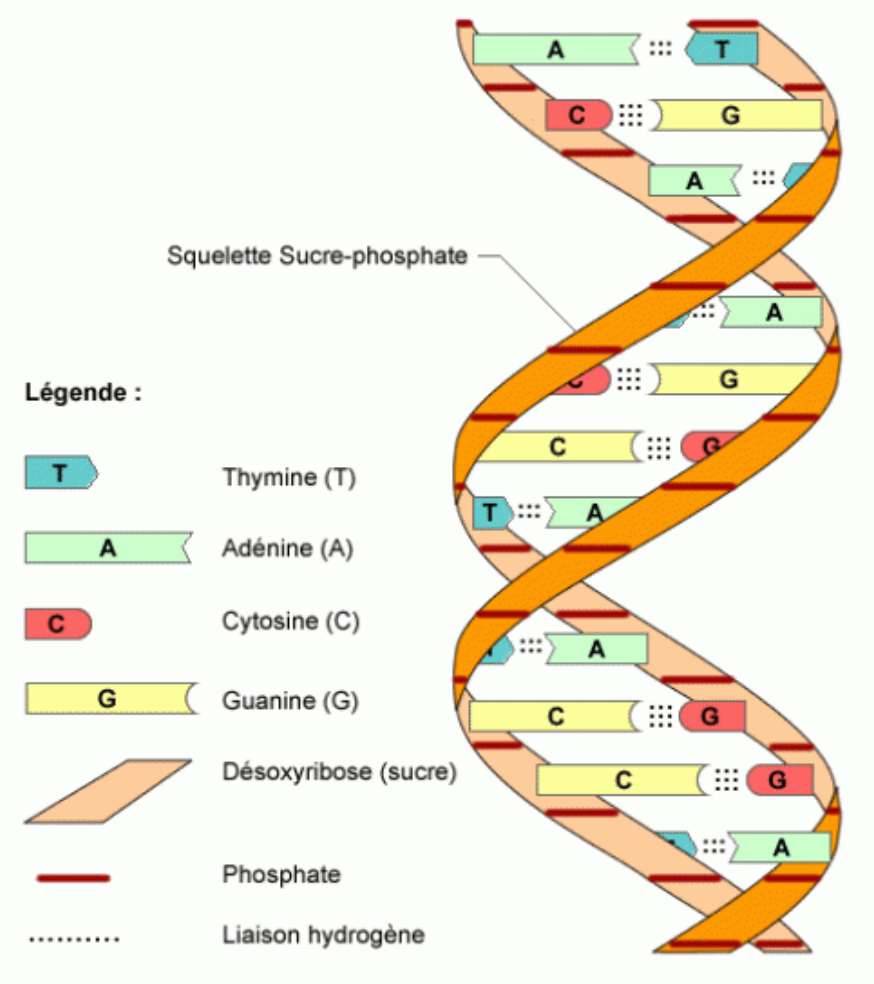
Les algorithmes de recherche textuelle sont notamment utilisés en bioinformatique.

2- Bioinformatique

Comme son nom l'indique, la bioinformatique est issue de la rencontre de l'informatique et de la biologie : la récolte des données en biologie a connu une très forte augmentation ces 30 dernières années. Pour analyser cette grande quantité de données de manière efficace, les scientifiques ont de plus en plus recourt au traitement automatique de l'information, c'est-à-dire à l'informatique.

a- Analyse de l'ADN

Comme vous le savez déjà, l'information génétique présente dans nos cellules est portée par les molécules d'ADN. Les molécules d'ADN sont, entre autres, composées de bases azotées ayant pour noms : **Adénine** (représenté par un A), **Thymine** (représenté par un T), **Guanine** (représenté par un G) et **Cytosine** (représenté par un C).



Molécule d'ADN

L'information génétique est donc très souvent représentée par de très longues chaines de caractères, composées des caractères **A, T, G et C**.

Exemple : CTATTCAGCAGTC...

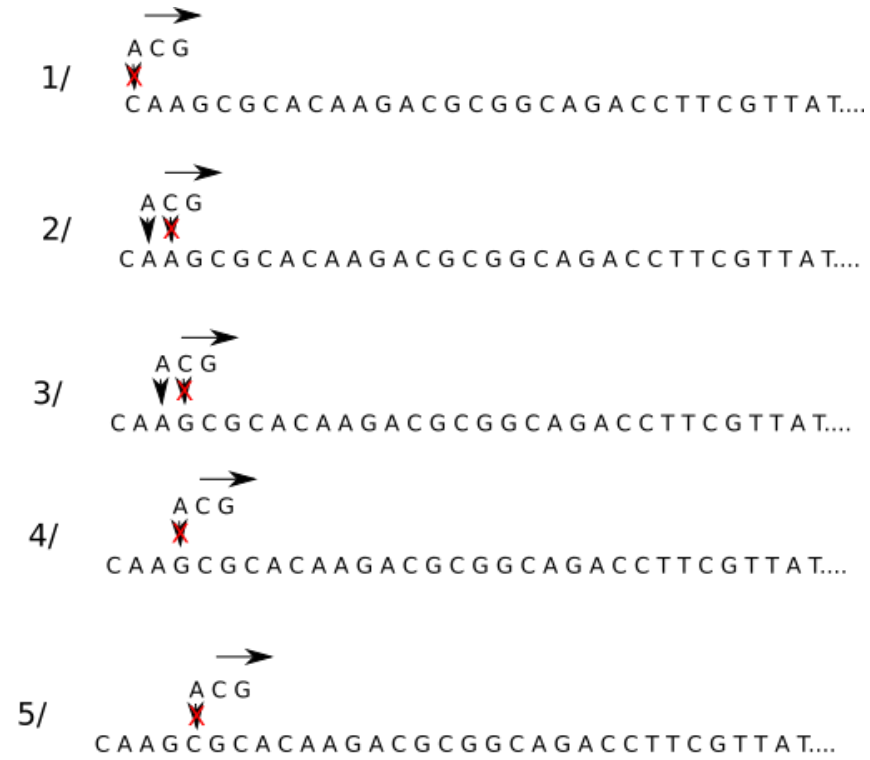
Il est souvent nécessaire de détecter la présence de certains enchainements de bases azotées (dans la plupart des cas un triplet de bases azotées code pour 1 acide aminé et la combinaison d'acides aminés forme une protéine).

Par exemple, on peut se poser la question suivante : trouve-t-on le triplet ACG dans le brin d'ADN suivant (et si oui, en quelle position ?):

```
CAAGCGCACAAGACGCGGCAGACCTTCGTTATAGGCGATGATTTCGAACCTACTAGTGGGTCTCTTAGGCCGAGC
GGTTCCGAGAGATAGTGAAAGATGGCTGGGCTGTGAAGGGAAGGAGTCGTGAAAGCGCGAACACGAGTGTGCG
CAAGCGCAGCGCCTTAGTATGCTCCAGTGTAGAAGCTCCGGCGTCCCGTCTAACCGTACGCTGTCCCCGGTACAT
GGAGCTAATAGGCTTTACTGCCCAATATGACCCCGCGCCGCGACAAAACAATAACAGTTTGCTGTATGTTCCATGGT
GGCCAATCCGTCTCTTTTCGACAGCACGGCCAATTCTCCTAGGAAGCCAGCTCAATTTCAACGAAGTCGGCTGTTGA
ACAGCGAGGTATGGCGTCGGTGGCTCTATTAGTGGTGAGCGAATTGAAATTCGGTGGCCTTACTTGTACCACAGCGA
TCCCTTCCCACCATTCTTATGCGTCGTCTGTTACCTGGCTTGGCAT
```

b- Utilisation d'un algorithme naïf (Recherche naïve)

Nous allons commencer par le premier algorithme qui nous vient à l'esprit (on parle souvent d'algorithme "**naïf**") :



Algorithme naïf

1. on place le motif recherché au même niveau que les 3 premiers caractères de notre chaine, le premier élément du motif ne correspond pas au premier élément de la chaine (A et C), on décale le motif d'un cran vers la droite.
2. le premier élément du motif correspond au premier élément de la chaine (A et A) mais pas le second (C et A), on décale d'un cran vers la droite
3. le premier élément du motif correspond au premier élément de la chaine (A et A) mais pas le second (C et G), on décale d'un cran vers la droite
4. le premier élément du motif ne correspond pas au premier élément de la chaine (A et G), on décale d'un cran vers la droite.
5. le premier élément du motif ne correspond pas au premier élément de la chaine (A et C), on décale d'un cran vers la droite.
6. ..... on continue le processus jusqu'au moment où les 3 éléments du motif correspondent avec les 3 éléments de la chaine situés au même niveau.

Cet algorithme naïf peut, selon les situations demander un très grand nombre de comparaisons, ce qui peut entraîner un très long temps de "**calcul**" avec des chaines très très longues. L'algorithme de **Boyer-Moore** permet de faire mieux en termes de comparaisons à effectuer

Algorithme de recherche naïve

```
def recherche_naive(txt, motif):
    '''
    renvoie la liste des indices (éventuellement vide) des occurrences de
    de la chaîne `motif` dans la chaîne `texte`.
    '''
    indices = []
    i = 0
    while i <= len(txt) - len(motif):
        k = 0
        while k < len(motif) and txt[i+k] == motif[k]:
            k += 1
        if k == len(motif):
            indices.append(i)
        i += 1

    return indices
```

Exemple d'utilisation :

```
>>> recherche_naive("une magnifique maison bleue", "maison")
[15]
>>> recherche_naive("une magnifique maison bleue", "nsi")
[]
>>> recherche_naive("une magnifique maison bleue", "ma")
[4, 15]
```

c- Algoritme de Boyer-Moore

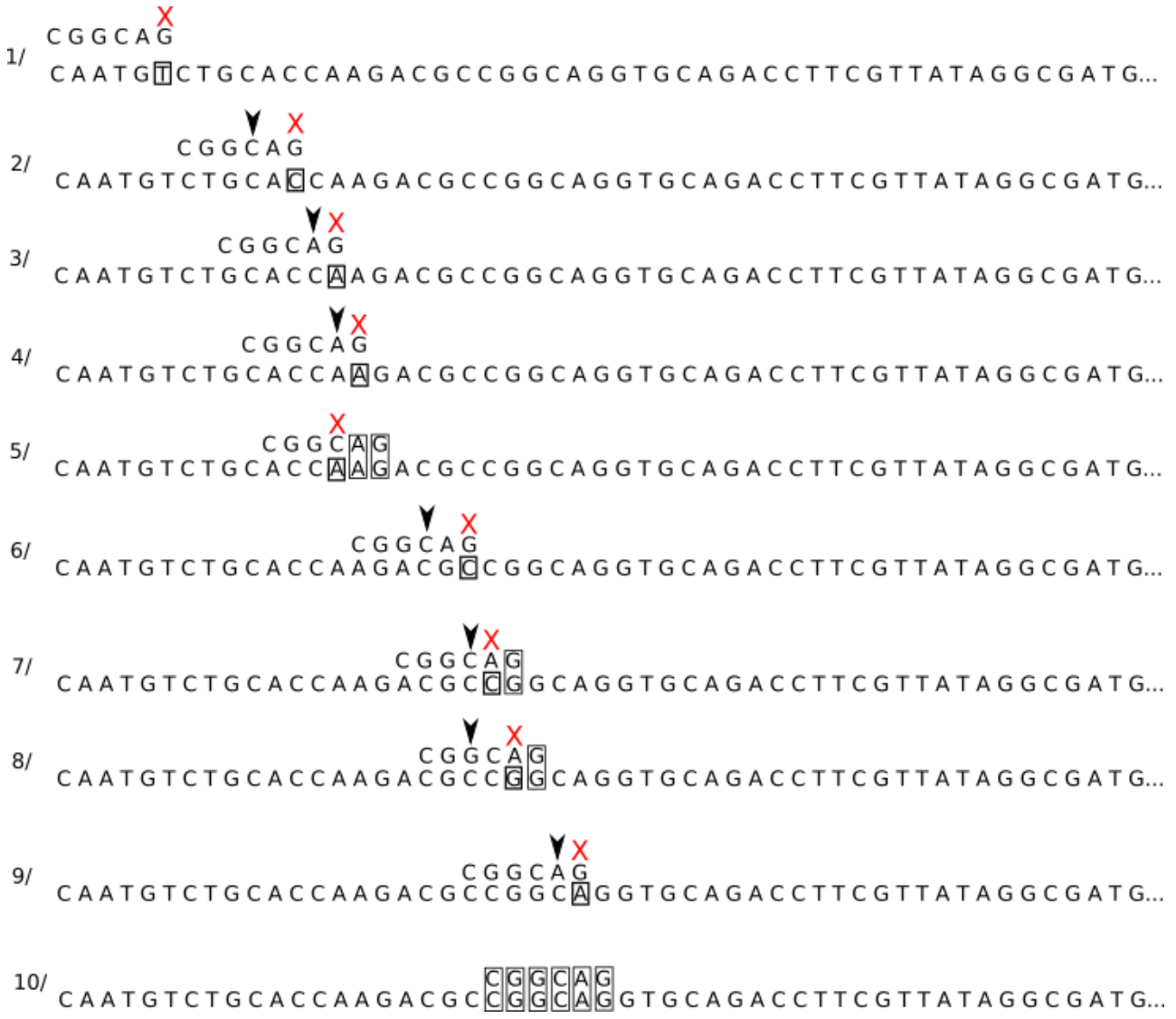
L'algorithme de Boyer-Moore se base sur les caractéristiques suivantes :

- L'algorithme effectue un prétraitement du motif. Cela signifie que l'algorithme "**connait**" les caractères qui se trouvent dans le motif
- On commence la comparaison motif-chaine par la droite du motif. Par exemple pour le motif CGGCAG, on compare d'abord le G, puis le A, puis C...on parcourt le motif de la droite vers la gauche
- Dans la méthode naïve, les décalages du motif vers la droite se faisaient toujours d'un "**cran**" à la fois. L'intérêt de l'algorithme de Boyer-Moore, c'est qu'il permet, dans certaines situations, d'effectuer un décalage de plusieurs crans en une seule fois.

Examinons un exemple. Soit la chaine suivante :

CAATGTCTGCACCAAGACGCCGGCAGGTGCAGACCTTCGTTATAGGCGATGATTTCGAACCTACTAGT
GGGTCTCTTAGGCCGAGCGGTTCCGAGAGATAGTGAAAGATGGCTGGGCTGTGAAGGGAAGGAGTC
GTGAAAGCGCGAACACGAGTGTGCGCAAGCGCAGCGCCTTAGTATGCTCCAGTGTAGAAGCTCCGG
CGTCCCGTCTAACCGTACGCTGTCCCCGGTACATGGAGCTAATAGGCTTTACTGCCCAATATGACCCC
GCGCCGCGACAAAACAATAACAGTTT

et le motif : CGGCAG



Algorithme de Boyer-Moore

1. on commence la comparaison par la droite, G et T ne correspondent pas. Le prétraitement du motif nous permet de savoir qu'il n'y a pas de T dans ce dernier, on peut décaler le motif de 6 crans vers la droite.

2. G et C ne correspondent pas, en revanche, on trouve 2 C dans le motif. On effectue un décalage du motif de 2 crans vers la droite afin de faire correspondre le C de la chaine (encadré sur le schéma) et le C le plus à droite dans le motif.
3. G et A ne correspondent pas, il existe un A dans le motif, on effectue un décalage d'un cran.
4. G et A ne correspondent pas, il existe un A dans le motif, on effectue un décalage d'un cran.
5. G et G correspondent, A et A correspondent, mais C et A ne correspondent pas. À gauche du C, il n'y a plus de A, on peut donc effectuer un décalage de 4 crans.
6. G et C ne correspondent pas, on effectue un décalage de deux crans pour faire correspondre les C.
7. G et G correspondent, A et C ne correspondent pas, on effectue un décalage d'un cran
8. G et G correspondent, A et G ne correspondent pas, on effectue un décalage de 2 crans (faire correspondre les G)
9. G et A ne correspondent pas, on effectue un décalage d'un cran
10. toutes les lettres correspondent, on a trouvé le motif dans la chaine.

On peut remarquer que l'on a bien, en fonction des cas, effectué plusieurs décalages en un coup, ce qui, au bout du compte, permet de faire moins de comparaison que l'algorithme naïf. On peut aussi remarquer que plus le motif est grand et plus l'algorithme de Boyer-Moore sera efficace.

Activité 1

Appliquez l'algorithme de Boyer-Moore au cas suivant :

chaine :

```
CAATGTCTGCACCAAGACGCCGGCAGGTGCAGACCTTCGTTATAGGCGATGATTTCGAACCTACTAGTGGGTCTCTT
AGGCCGAGCGGTTCCGAGAGATAGTGAAAGATGGCTGGGCTGTGAAGGGAAGGAGTCGTGAAAGCGCGAACA
CGAGTGTGCGCAAGCGCAGCGCCTTAGTATGCTCCAGTGTAGAAGCTCCGGCGTCCCGTCTAACCGTACGCTGT
CCCCGGTACATGGAGCTAATAGGCTTTACTGCCCAATATGACCCCGCGCCGCGACAAAACAATAACAGTTT
```

motif : ACCTTCG

Activité 2

Étudiez attentivement le programme Python suivant :

```
NO_CAR = 256
def recherche(txt, motif):
    m = len(motif)
    n = len(txt)
    tab_car = [-1]*NO_CAR
    for i in range(m):
        tab_car[ord(motif[i])] = i
    decalage = 0
    res = []
    while(decalage <= n-m):
        j = m-1
        while j>=0 and motif[j] == txt[decalage+j]:
            j = j - 1
        if j<0:
            res.append(decalage)
            if decalage+m<n :
                decalage = decalage + m-tab_car[ord(txt[decalage+m])]
            else :
                decalage = decalage + 1
        else:
            decalage = decalage + max(1, j-tab_car[ord(txt[decalage+j])])
```

```
    return res
```

Activité 3

Testez le programme de l'"Activité 2" avec l'exemple proposé à L' "Activité 1".