Séquence 14 – Programmation dynamique

Objectifs

1. Comprendre le principe de la programmation dynamique
2. Écrire un algorithme utilisant la programmation dynamique
3. Appliquer la programmation dynamique au problème du rendu de monnaie
4. Appliquer la programmation dynamique au problème de l' alignement de séquences

Cette séquence s'appuie sur :

- https://pixees.fr/informatiquelycee/n_site/nsi_term_algo_diviser_pour_regner.html
- https://pixees.fr/informatiquelycee/n_site/nsi_term_algo_progdyn.html
- https://pixees.fr/informatiquelycee/n_site/nsi_term_algo_boyer.html

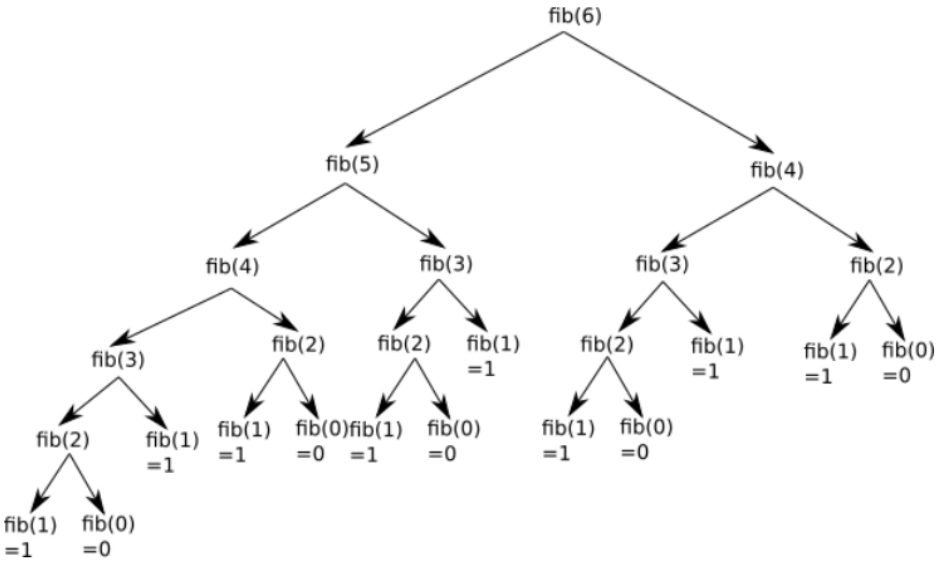
1 Suite de Fibonacci et récursivité explosive

1.1 Le problème

Revenons sur ce qui a été vu dans le cours consacré à la récursivité. 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)
```

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



Des valeurs élevées pour n donnent des calculs longs.

Si T_n désigne le nombre d'opérations pour calculer `fib(n)`, on obtient la relation de récurrence :

$$T_{n+2}=T_{n+1}+T_n+1 \quad .$$

Cette suite (et donc le nombre d'opérations) croit de façon exponentielle.

A faire vous même 1.

- Saisissez la fonction python `fib(n)`
- Testez cette fonction pour `n=10, n=20, n=30, …`
- Que pensez-vous des temps de calcul ?

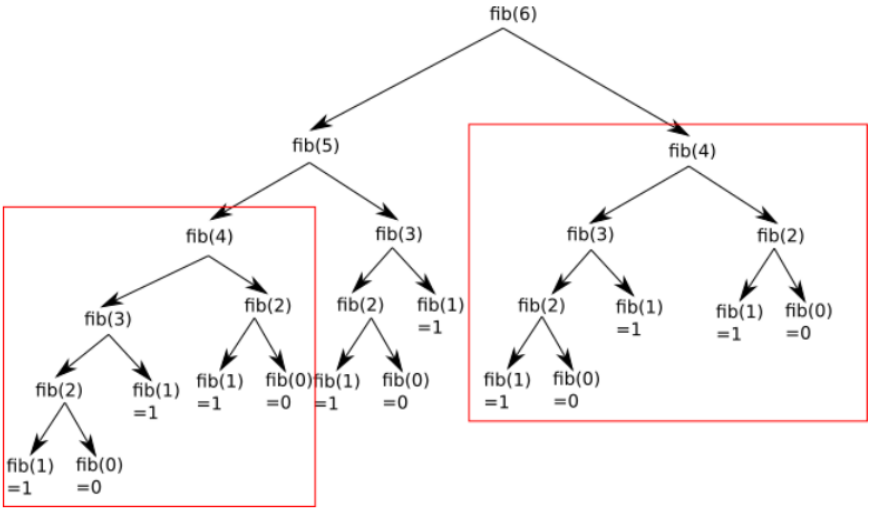
1.2 Une solution à la récursivité explosive : La mémoïsation

Lire P. 43

1.2.1 Principe

Si vous regardez le schéma ci-dessus, 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 :



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.

A faire vous même 2. Mise en application de la Mémoïsation

- P. 50 ex 10

A faire vous même 3. Utilisation d' un décorateur python pour la mémoïsation

- Lire P. 317 (mémoïsation d' une fonction quelconque)
- P. 326 ex 4

1.3 Conclusion

Pour des valeurs de `n` élevées, le gain en termes de performance (temps de calcul) est évident. Pour des valeurs `n` très élevées, dans le cas du programme récursif "classique", 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 fait, 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 est une approche "programmation dynamique".

2 Programmation dynamique

2.1 Principe

Comme nous venons de le voir, la programmation dynamique, comme la méthode diviser pour régner, résoud 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.
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, évitant ainsi le recalcul de la solution chaque fois qu'il résout chaque sous-sous-problème.

2.2 Appliqué au problème de rendu de monnaie

2.2.1 Principe :

On veut programmer une caisse automatique pour qu'elle rende la monnaie de façon optimale, c'est-à-dire avec le nombre minimal de pièces et billets.
La valeur des pièces et billets à disposition sont : 2, 3, 5, 10, 20, 50, 100 et 200 euros. On dispose d'autant d'exemplaires qu'on le souhaite de chaque pièce et billet.

2.2.2 Exemple :

Anaïs veut acheter un objet qui coûte 53 euros. Elle paye avec un billet de 100 euros. La caisse automatique doit lui rendre 47 euros.
La meilleure façon de rendre la monnaie est de le faire avec deux billets de 20, un billet de 5 et une pièce de 2 euros.

2.2.3 Rappel - Résolution « gloutonne »

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

1. On prend le billet/pièce qui a la plus grande valeur (il faut que la valeur de ce billet/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.

Reprenons notre exemple :
nous avons 47 euros à rendre :

- On utilise un billet de 20 € (plus grande valeur inférieure à 47 euro), il reste 27 € à rendre
- On utilise un billet de 20 € (plus grande valeur inférieure à 27 euro), il reste 7 € à rendre
- On utilise une pièce de 5 € (plus grande valeur inférieure à 7 euro), il reste 2 € à rendre
- On utilise une pièce de 2 € (plus grande valeur inférieure à 2 euro), il reste 0 € à rendre

L'algorithme se termine en renvoyant 4 (on a dû rendre 4 billets/pièces)

A faire vous même 4.

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

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 un billet de 10 € (plus grande valeur inférieure à 11 €), il reste 1 € à rendre
- il n'y a pas de pièce de 1 € => 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 le billet de 10 €, 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).
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 € et 2 pièces de 3 € pour arriver à 11 €.
Recherchons un algorithme qui nous permettrait de trouver une solution optimale, quelle que soit la situation.

2.2.4 Programmation dynamique

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 $Nb(X)-1$ pièces ?

Si j'ai à ma disposition la liste de pièces suivante : $p_1, p_2, p_3, \ldots, p_n$ et que je suis capable de rendre X, je suis donc aussi capable de rendre :

- $X - p_1$
- $X - p_2$
- $X - p_3$
- ...
- $X - p_n$

à condition que p_i (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 € et que j'ai à ma disposition des pièces de 2 €, 5 €, 10 €, 50 € et 100 €, je peux aussi rendre :

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

Autrement dit, si $Nb(X - p_i)$ (avec i compris entre 1 et n) est le nombre minimal de pièces à rendre pour le montant $X - p_i$, alors $Nb(X)=1+Nb(X - p_i)$ 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 $X>0 : Nb(X)=1+min(Nb(X - p_i))$ avec $1 \leq i < n$ et $p_i \leq X$

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

A faire vous même 5.

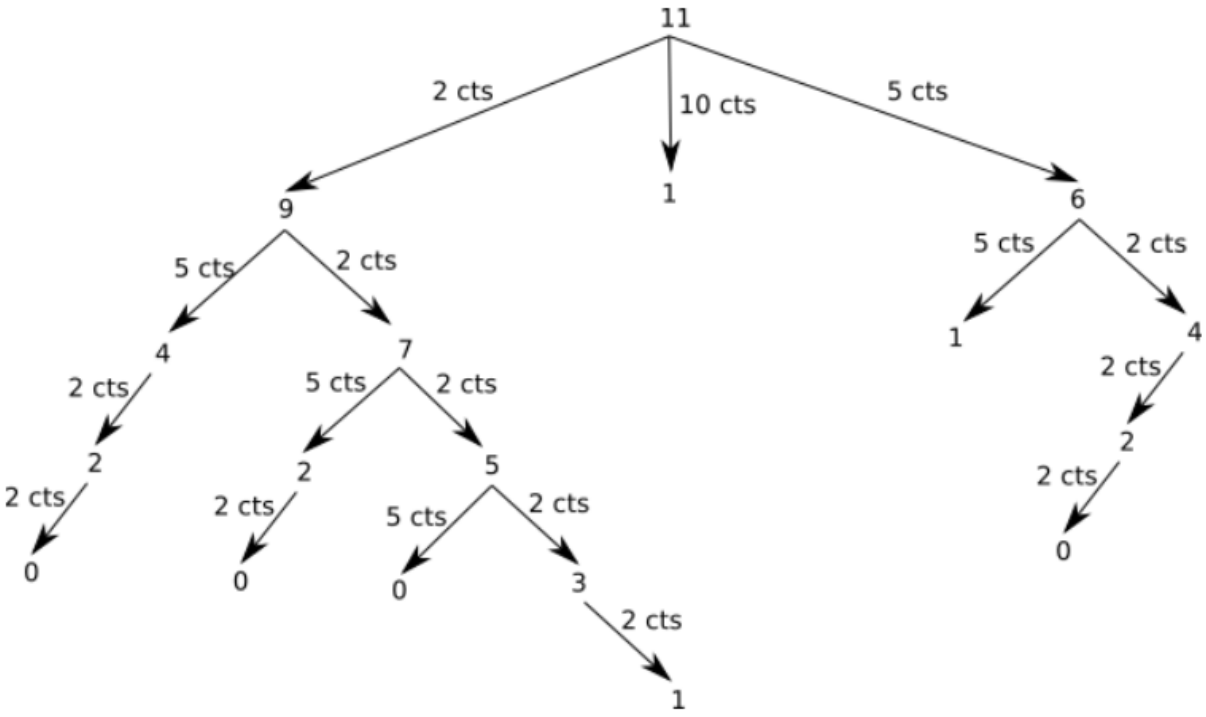
- Étudiez attentivement le programme Python suivant
- Testez le programme pour 11 € (on recherche le nombre minimum de pièces à rendre pour une somme de 11 €. Les pièces disponibles sont : 2 €, 5 €, 10 €, 50 € et 100 €

```
def rendu_monnaie_rec(liste_pieces, X):
    if X==0:
        return 0
    else:
        mini = 1000
    for piece in liste_pieces:
        if piece<=X:
            nb = 1 + rendu_monnaie_rec(liste_pieces,X-piece)
            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.

Voici un schéma (avec une somme à rendre de 11 centimes) qui vous permettra de mieux comprendre le principe de cet algorithme :



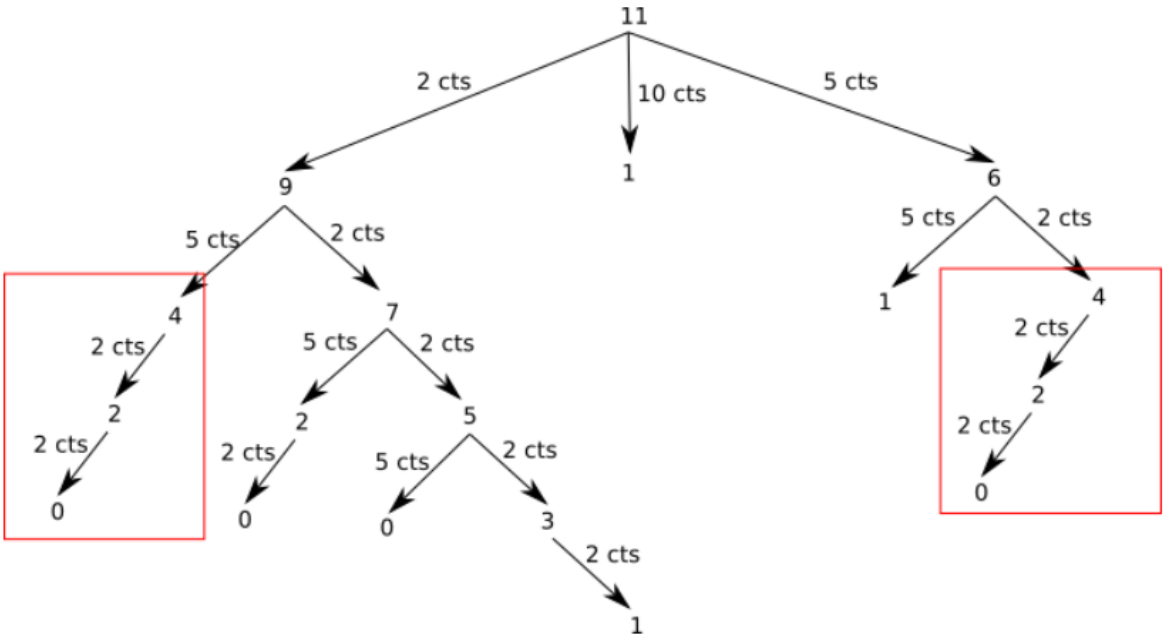
Plusieurs remarques s'imposent :

- 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)

A faire vous même 6.

- Recherchez le nombre minimum de pièces à rendre pour une somme de 171 euro. Les pièces disponibles sont : 2 €, 5 €, 10 €, 50 € et 100 €.

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 € 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 171 euro, on va retrouver de nombreuses fois exactement les mêmes calculs, il est donc potentiellement intéressant d'utiliser la programmation dynamique.

A faire vous même 7.

- Étudiez attentivement le programme Python suivant
- Testez-le pour 171 €

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

def rendu_monnaie_mem_c(liste_pieces,X,m):
    if X==0:
        return 0
    elif m[X]>0:
        return m[X]
    else:
        mini = 1000
        for piece in liste_pieces:
            if piece<=X:
                nb=1+rendu_monnaie_mem_c(liste_pieces,X-piece,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.
Comme vous pouvez le constater, il suffit d'une fraction de seconde pour que le programme basé sur la programmation dynamique donne une réponse correcte.

P. 327 ex 6

P. 327 ex 7

3 Recherche d'un motif dans un texte avec l'algorithme de Boyer-Moore

Lire P. 320-321

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

```
"Les sanglots longs des violons de l'automne blessent mon coeur 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.

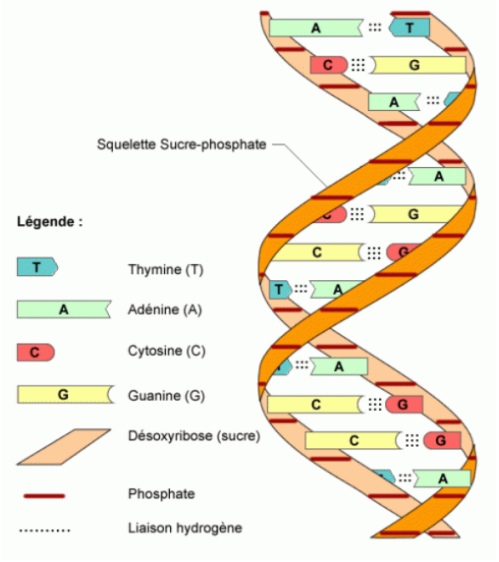
Les algorithmes de recherche textuelle sont notamment utilisés en bioinformatique.

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.

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 ?):

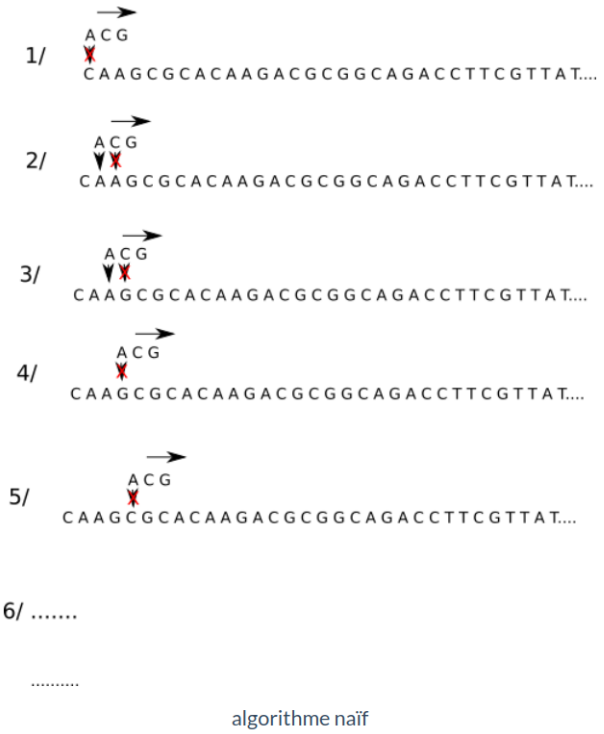
```
CAAGCGCACAAGACGCGGCAGACCTTCGTTATAGGCGATGATTTCGAACCTACTAGTGGGTCTCTTAGGCCGAGCGGTTCCGAGA
GATAGTGAAAGATGGCTGGGCTGTGAAGGGAAGGAGTCGTGAAAGCGCGAACACGAGTGTGCGCAAGCGCAGCGCCTTAGTATGC
TCCAGTGTAGAAGCTCCGGCGTCCCGTCTAACCGTACGCTGTCCCCGGTACATGGAGCTAATAGGCTTTACTGCCCAATATGACC
CCGCGCCGCGACAAAACAATAACAGTTTGCTGTATGTTCCATGGTGGCCAATCCGTCTCTTTTCGACAGCACGGCCAATTCTCCT
AGGAAGCCAGCTCAATTTCAACGAAGTCGGCTGTTGAACAGCGAGGTATGGCGTCGGTGGCTCTATTAGTGGTGAGCGAATTGAA
ATTCGGTGGCCTTACTTGTACCACAGCGATCCCTTCCCACCATTCTTATGCGTCGTCTGTTACCTGGCTTGGCAT
```

3.1 Algorithme naif

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

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.

1. 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
2. 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
3. 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.
4. 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.
5. ..... 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.



algorithme naïf

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

3.2 Algorithme de Boyer-Moore

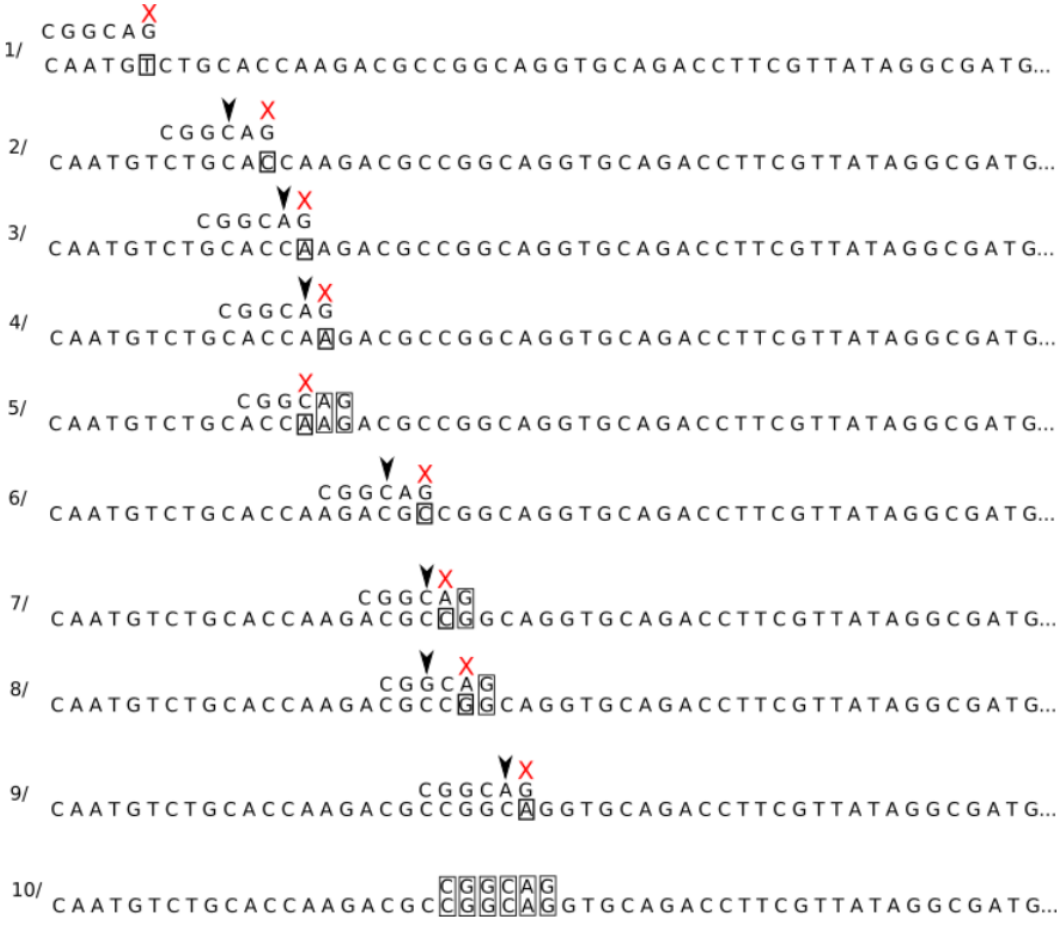
L'algorithme de Boyer-Moore se base sur les caractéristiques suivantes :

- L'algorithme effectue un prétraitement du motif. Cela signifie que l'algorithme "connaît" les caractères qui se trouvent dans le motif
- on commence la comparaison motif-chaîne 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 chaîne suivante :

```
CAATGTCTGCACCAAGACGCCGGCAGGTGCAGACCTTCGTTATAGGCGATGATTTCGAACCTACTAGTGGGTCTCTTAGGCCGAG
CGGTTCCGAGAGATAGTGAAAGATGGCTGGGCTGTGAAGGGAAGGAGTCGTGAAAGCGCGAACACGAGTGTGCGCAAGCGCAGCG
CCTTAGTATGCTCCAGTGTAGAAGCTCCGGCGTCCCGTCTAACCGTACGCTGTCCCCGGTACATGGAGCTAATAGGCTTTACTGC
CCAATATGACCCCGCGCCGCGACAAAACAATAACAGTTT
```

et le motif : `CGGCAG`



algorithme de Boyer-Moore

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.

1. 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 chaîne (encadré sur le schéma) et le C le plus à droite dans le motif.
2. G et A ne correspondent pas, il existe un A dans le motif, on effectue un décalage d'un cran.
3. G et A ne correspondent pas, il existe un A dans le motif, on effectue un décalage d'un cran.
4. 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.
5. G et C ne correspondent pas, on effectue un décalage de deux crans pour faire correspondre les C.
6. G et G correspondent, A et C ne correspondent pas, on effectue un décalage d'un cran
7. G et G correspondent, A et G ne correspondent pas, on effectue un décalage de 2 crans (faire correspondre les G)
8. G et A ne correspondent pas, on effectue un décalage d'un cran
9. toutes les lettres correspondent, on a trouvé le motif dans la chaîne.

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.

A faire vous même 8.

- Appliquer l'algorithme pour rechercher `papas` dans
  `un papou papa à poux a des poux papas et des poux pas papas`
- Compter le nombre de comparaisons de caractères dans cette recherche.

A faire vous même 9.

- Mêmes questions en recherchant le motif `avis` dans `Réfléchir est un bon moyen de progresser`.

A faire vous même 10. Pour les rapides

- Mêmes questions en recherchant le motif `001` dans `00000001`.

A faire vous même 11.

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

motif : `ACCTTCG`
chaîne :

```
CAATGTCTGCACCAAGACGCCGGCAGGTGCAGACCTTCGTTATAGGCGATGATTTCGAACCTACTAGTGGGTCTCTTAGGCCGAG
CGGTTCCGAGAGATAGTGAAAGATGGCTGGGCTGTGAAGGGAAGGAGTCGTGAAAGCGCGAACACGAGTGTGCGCAAGCGCAGCG
CCTTAGTATGCTCCAGTGTAGAAGCTCCGGCGTCCCGTCTAACCGTACGCTGTCCCCGGTACATGGAGCTAATAGGCTTTACTGC
CCAATATGACCCCGCGCCGCGACAAAACAATAACAGTTT
```

3.3 Implémentation en python

A faire vous même 12.

- Implémenter l' algorithme naif

Plutôt que de calculer la longueur du saut à chaque fois, on construit une table de sauts avant de lancer la recherche. L' algorithme « connaît » ainsi les caractères qui se trouvent dans le motif.
Pour le motif : `CGGCAG` on a :

C	G	A	autres
+2	+4	+1	+6

A faire vous même 13.

- Étudiez, implémentez, commentez et testez l' algorithme de Boyer-Moore

```
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
```

A faire vous même 14. Pour les rapides

- Avec matplotlib, timeit ou @chrono tracez un graphique avec les temps de calcul de chaînes de différentes longueurs.